Université de Montréal

# Sequence-to-sequence learning for machine translation and automatic differentiation for machine learning software tools

**par Bart van Merriënboer**

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Octobre, 2018

# Résumé

Cette thèse regroupe des articles d'apprentissage automatique et s'articule autour de deux thématiques complémentaires.

D'une part, les trois premiers articles examinent l'application des réseaux de neurones artificiels aux problèmes du traitement automatique du langage naturel (TALN). Le premier article introduit une structure codificatrice-décodificatrice avec des réseaux de neurones récurrents pour traduire des segments de phrases de longueur variable. Le deuxième article analyse la performance de ces modèles de 'traduction neuronale automatique' de manière qualitative et quantitative, tout en soulignant les difficultés posées par les phrases longues et les mots rares. Le troisième article s'adresse au traitement des mots rares et hors du vocabulaire commun en combinant des algorithmes de compression par dictionnaire et des réseaux de neurones récurrents.

D'autre part, la deuxième partie de cette thèse fait abstraction de modèles particuliers de réseaux de neurones afin d'aborder l'infrastructure logicielle nécessaire à leur définition et entraînement. Les infrastructures modernes d'apprentissage profond doivent avoir la capacité d'exécuter efficacement des programmes d'algèbre linéaire et par tableaux, tout en étant capable de différentiation automatique (DA) pour calculer des dérivées multiples. Le premier article aborde les défis généraux posés par la conciliation de ces deux objectifs et propose la solution d'une représentation intermédiaire fondée sur les graphes. Le deuxième article attaque le même problème d'une manière différente : en implémentant un code source par bande dans un langage de programmation dynamique par tableau (Python et NumPy).

**Mots-clés** apprentissage automatique, réseaux de neurones, apprentissage profond, traitement automatique du langage naturel, traduction automatique, différentiation automatique

# Summary

This thesis consists of a series of articles that contribute to the field of machine learning. In particular, it covers two distinct and loosely related fields.

The first three articles consider the use of neural network models for problems in natural language processing (NLP). The first article introduces the use of an encoder-decoder structure involving recurrent neural networks (RNNs) to translate from and to variable length phrases and sentences. The second article contains a quantitative and qualitative analysis of the performance of these 'neural machine translation' models, laying bare the difficulties posed by long sentences and rare words. The third article deals with handling rare and out-of-vocabulary words in neural network models by using dictionary coder compression algorithms and multi-scale RNN models.

The second half of this thesis does not deal with specific neural network models, but with the software tools and frameworks that can be used to define and train them. Modern deep learning frameworks need to be able to efficiently execute programs involving linear algebra and array programming, while also being able to employ automatic differentiation (AD) in order to calculate a variety of derivatives. The first article provides an overview of the difficulties posed in reconciling these two objectives, and introduces a graph-based intermediate representation that aims to tackle these difficulties. The second article considers a different approach to the same problem, implementing a tape-based source-code transformation approach to AD on a dynamically typed array programming language (Python and NumPy).

**Keywords** machine learning, neural networks, deep learning, natural language processing, machine translation, automatic differentiation

# Contents

# List of Figures

# List of Tables

# List of abbreviations

| | |
|---|---|
| **AD** | Automatic differentiation |
| **AI** | Artificial intelligence |
| **ANN** | Artificial neural network |
| **AST** | Abstract syntax tree |
| **BLEU** | Bilingual Evaluation Understudy |
| **BPE** | Byte pair encoding |
| **CSLM** | Continuous space language model |
| **DL** | Deep learning |
| **GPU** | Graphics processing unit |
| **GRU** | Gated recurrent unit |
| **i.i.d.** | Independent and identically distributed |
| **IL** | Intermediate language |
| **IR** | Intermediate representation |
| **JIT** | Just-in-time |
| **LSTM** | Long short-term memory |
| **ML** | Machine learning |
| **MT** | Machine translation |
| **MLP** | Multilayer perceptron |
| **NLP** | Natural language processing |
| **NNLM** | Neural net language model |
| **NMT** | Neural machine translation |
| **OO** | Operator overloading |
| **OOV** | Out-of-vocabulary |
| **PL** | Programming languages |
| **RNN** | Recurrent neural network |
| **SCT** | Source code transformation |
| **SGD** | Stochastic gradient descent |
| **SMT** | Statistical machine translation |
| **ST** | Source transformation |
| **SVM** | Support vector machine |

# Notation

**Vectors** are denoted by lower case bold Roman letters such as $\mathbf{x}$. The elements of a vector are written with subscripts as in $\mathbf{x} = (x_1, \ldots, x_n)$. Lower case bold letters and parentheses are also used to denote sequences. Individual elements are generally referred to with $x_i$, unless there is a temporal aspect in which case $x_t$ is used instead.

**Matrices** will be written with uppercase bold Roman letters such as $\mathbf{A}$. The identity matrix is denoted $\mathbf{I}$

**Scalars** are written with lowercase italics e.g. $b$. Any set element that is not explicitly a matrix or vector is also denoted with lowercase italics.

**Diagonal matrices** are denoted $\operatorname{diag}(\mathbf{x})$ where the vector $\mathbf{x}$ contains the diagonal entries of the matrix. Reversely, $\operatorname{diag}(\mathbf{A})$ is a vector containing the diagonal elements of the matrix $\mathbf{A}$.

**Derivatives** are denoted using either the Lagrange notation, where $f'$ is the derivative of $f$, or the Leibniz notation, in which case $f'(x)$ where $y = f(x)$ is written as $\frac{\mathrm{d}y}{\mathrm{d}x}$. The same notation is used for multivariate and vector-valued functions i.e. for $f : \mathbb{R}^n \to \mathbb{R}^m$ we use $f'$ to denote the Jacobian $m \times n$ matrix of partial derivatives and $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{x}}$ to denote the Jacobian matrix evaluated at $\mathbf{x}$.

**Iverson brackets** are written as $[P] := \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$

**Probability distributions** are denoted by a lowercase $p(x)$ or $p_X(x)$ where $X$ is the random variable (in uppercase Roman letters) of which $p$ is the probability mass or density function. The probability of a particular event happening is denoted $P(A)$, where $A$ is an event, or $P(X = x)$, where $x$ is the realization of a random variable $X$.

**Sets** are denoted with uppercase italic letters, and its individual elements are denoted with subscripts i.e. $X = \{x_1, \ldots, x_N\}$.

**Functions** are written with a variety of scripts. The notation $f : A \to B$ means that the function $f$ has the set $A$ as its domain and the set $B$ as its codomain. We write $f(x)$ where $x$ is the argument to $f$ and $f(x)$ the resulting value. We will write $f_\theta(x)$ or $f(x; \theta)$ to highlight the different treatment of the input $x$ and the parameters $\theta$, although both are technically arguments to the function $f$. If a function $f : \mathbb{R} \to \mathbb{R}$ is applied to a vector or matrix we assume it's applied element-wise.

**Estimators** will often be given a circumflex accent i.e. $\hat{f}$ is an estimator of the function $f$ while $\hat{y}$ is a model's estimate of the target $y$.

**Optimal values** can be recognised by the superscript asterisk i.e. $\theta^*$ is the best possible set of parameters according to some criterion.

**Expected value** is given by a blackboard $\mathbb{E}[X]$. Note that square brackets are used for functionals such as $\mathbb{E}$ and Var.

**Algebraic operations** with specific symbols that are used include $\odot$ for the element-wise product and $\oplus$ for the XOR function.

**Model** inputs and outputs will usually be denoted by $\mathbf{x}$ and $\mathbf{y}$ (or $\mathbf{x}_i$ and $\mathbf{y}_i$ to refer to specific examples in the dataset). The model parameters will be $\theta$ and the parameter space $\Theta$. In the context of neural networks we will use $\mathbf{W}$ or $\mathbf{w}$ to refer to the weights and $\mathbf{b}$ or $b$ for the biases. Intermediate layers are usually denoted by $\mathbf{h}$ for 'hidden'. We endeavour to reserve $N$ for the size of the dataset, $n$ for the size of the input space and $m$ for the size of the output space.

**Parenthesized superscripts** are used to make a distinction between different but similar variables e.g. $\mathbf{W}^{(h)}$ and $\mathbf{W}^{(y)}$ are two different weight matrices, and $\phi^{(h)}$ and $\phi^{(y)}$ are two different activation functions.

**Strings** are denoted with lowercase bold letters, but their elements are not parenthesized as in $\mathbf{s} = s_1, \dots, s_t$, where $s_i$ is a character. We write $\mathbf{s} = \mathbf{t}_1, \dots, \mathbf{t}_m$ when $\mathbf{s}$ is a concatenation of subsequences $\mathbf{t}_i$.

# 1 Background

*Machine learning* is the study of algorithms that can learn from data. *Learning* has been referred to as the "phenomenon of knowledge acquisition in the absence of explicit programming" (Valiant, 1984), which defines it in juxtaposition with rule-based systems, which perform a series of logical operations on premises in order to deduce facts. The importance of learning for artificial intelligence was recognised as early as 1950 in Alan Turing's seminal paper "Computing machinery and intelligence".

> Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. [. . . ] We have thus divided our problem into two parts. The child programme and the education process. These two remain very closely connected. We cannot expect to find a good child machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. [. . . ] We normally associate punishments and rewards with the teaching process. Some simple child machines can be constructed or programmed on this sort of principle. The machine has to be so constructed that events which shortly preceded the occurrence of a punishment signal are unlikely to be repeated, whereas a reward signal increased the probability of repetition of the events which led up to it.

Machine learning endeavours to find good 'child machines' and 'teaching methods'. It is closely related to statistics, sharing methodology and theoretical foundations. Seen through this prism, learning involves the estimation of a function operating on a probability distribution $p(x)$ given a series of samples from this distribution. The goal is for the learning algorithm to *generalize* (Bishop, 2006) and correctly estimate $f(x)$ for unseen samples after the learning process has completed.

*Deep learning* is a subfield of machine learning that focuses on algorithms that apply series of non-linear operations on the data in order to model high-level abstractions more efficiently (Y. Bengio, 2009). The most common deep architectures are *artificial neural networks* with multiple layers. Although modelled after biological neurons (McCulloch and Pitts, 1943), artificial neural networks are perhaps better understood as a series of interleaved linear transformations and element-wise non-linear functions applied to an input represented by a vector. Deep learning and artificial neural networks are closely related to the field of *representation learning*. Representation (or feature) learning is the process by which algorithms learn abstract representations of the data that they process (Y. Bengio, Courville, and Vincent, 2013). In the context of this work we consider *distributed representations*, where data is represented by a pattern of activity (e.g. a vector) as opposed to a localist representation (e.g. a single category). The activations of a layer in a neural network can be considered to be such a representation. Deep learning and representation learning are primary examples of a *connectionist* approach to artificial intelligence (Geoffrey E Hinton, James L McClelland, David E Rumelhart, et al., 1986). Connectionism contends that intelligent systems are best modelled as a form of emergence, a process in which patterns and behaviour arise through the interaction of many smaller, simpler units (i.e. the neurons). In Section 1.2 I will provide a short overview of the theoretical foundations of machine learning theory followed by a more practical overview of deep learning models.

Machine learning, neural networks and connectionism have a tumultuous history in artificial intelligence. Research in neural networks stagnated for several years after a book by Minsky and Papert (1969) discussed the limitations of single-layer neural networks (perceptrons). The AI community focused on symbolic, rule-based systems instead. An efficient training algorithm for deep neural networks, *back-propagation*, was introduced by Werbos (1974) but it was not until the mid-1980s that research in connectionist approaches fully resumed with the work of e.g. Hopfield (1982). Linear classifiers, such as support vector machines (SVMs), overshadowed neural networks for most of the 1990s and early 2000s, but the late 2000s saw the advent of deep learning methods. An important contribution was the development of a new pre-training method by Hinton (2006) that allowed for deeper neural networks to be trained. Turing foresightfully argued in 1950 that advancements in engineering were needed for his ‘learning machine’ to be feasible and it was the availability of efficient and cheap parallel computation in the form of graphics processing units (GPUs) that was one of the other main drivers behind the resurgence of artificial neural networks (Jürgen Schmidhuber, 2015).

## 1.1 Natural language understanding and artificial intelligence

The ability to read and understand natural language is by many considered a prerequisite for *general artificial intelligence* (Russel and Norvig, 2003). When Turing introduced the Turing test in 1950 he based his measure of machine intelligence on the ability to understand written natural language and reply appropriately.

The history of natural language processing (NLP) in many ways parallels that of artificial intelligence. In the 1970s and 1980s many NLP systems used sets of complex hand-written rules, grounded in the *rationalist* Chomskyan tradition of linguistics (Christopher D. Manning and Schütze, 1999). In the late 1980s an *empiricist* approach to language started taking hold, grounded in the belief that the structure of language can be learned from data, in many ways mirroring the growth of connectionism at the expense of computationalist/symbolic approaches in artificial intelligence. Examples of this approach are counting models, such as $n$-gram language models, as well as models that employ distributed representations of words and documents, as in latent semantic analysis. *Word embeddings* (Y. Bengio, Ducharme, et al., 2003) move beyond the surface level of the word; instead of considering 'cat' and 'dog' as entirely separate words, they are embedded in a vector space in such a way that distance relates to semantic similarity (Hill et al., 2014; Mikolov, Sutskever, et al., 2013).

In the way that learning for artificial intelligence can be traced back to Turing in the 1950s, when read from the perspective of a present-day machine learning researcher, Wittgenstein's *Philosophical Investigations* makes a case for statistical methods when proclaiming that "the meaning of a word is its use in the language" (1953), effectively going on to argue for the need of distributed representations:

> Consider for example the proceedings that we call "games" [to] look and see whether there is anything common to all. [...] And the result of this examination is: we see a complicated network of similarities overlapping and criss-crossing: sometimes overall similarities. I can think of no better expression to characterize these similarities than "family resemblances"; for the various resemblances between members of a family: build, features, colour of eyes, gait, temperament, etc. etc. overlap and criss-cross in the same way. — And I shall say: "games" form a family. [...] I can give the concept 'number' rigid limits [...] that is, use the word "number" for a rigidly limited concept, but I can also use it so that the extension of the concept is not closed by a frontier. And this is how we do use the word "game". For how is the concept of a game bounded? What still counts as a game and what no longer does? Can you give the boundary? No. You can draw one; for none has so far been drawn. (But that never troubled you before when you used the word "game".)

The idea that the meaning of a word can be derived from its use in language, the principle that underlies most models that learn distributed word representations, was more succinctly put by John Rupert Firth in 1957 as

> You shall know a word by the company it keeps.

## 1.2 Machine learning

A semi-formal definition of machine learning can be given as (T. M. Mitchell, 1997):

**Definition 1.2.1.** *A computer program is said to* learn *from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$.*

This definition is comprehensive but abstract. To make the definitions of 'computer program' (the model), $E$, $T$ and $P$ more concrete we first have to consider three commonly recognised categories of machine learning:

**Supervised learning** In supervised learning the model is provided with a set of input-output pairs (the experience $E$) and the goal is to learn a mapping between the space of inputs to the space of outputs. In *classification* each input is given a label from a finite set of classes, for example a set of images each of which is labelled as containing a cat or dog. When each target output is associated with a numeric value we refer to the task as *regression*, for example when predicting life expectancy given a set of health indicators. If the output is a structured object instead of a discrete or real value, the problem is referred to as *structured prediction*, for example when predicting a parse tree given a sentence.

**Unsupervised learning** In unsupervised learning the model is only given a set of inputs and is expected to find structure. If the task is to divide the inputs in a number of categories the problem is known as *clustering*. In *density estimation* the algorithm is expected to model the probability distribution from which the inputs were drawn. *Unsupervised representation learning* focuses on learning distributed representations of the inputs that can be used in downstream tasks. Other tasks that can be considered unsupervised learning are *dimensionality reduction*, where the inputs are to be mapped to a lower-dimensional space while minimizing information loss, and *anomaly detection*, where we need to find the outliers in the inputs that do not belong to some structure or pattern.

**Reinforcement learning** In the reinforcement learning setting an *agent* interacts with a dynamic environment. At each time step the agent observes (part of) the environment and then chooses from a (constrained) set of actions which affect the state of the environment, possibly in a stochastic manner. At each step the agent receives a reward which depends on the environment's state and the action taken. The goal for the agent is to maximize its future rewards.

Note that these categorisations and tasks are not mutually exclusive. For example, density estimation can be reframed as clustering by considering a mixture of distributions where we assign each example to the mixture component under which it has the highest likelihood. *Semi-supervised learning* studies models which use a combination of labelled and unlabelled data to improve performance.

Orthogonal to the categorization of tasks above, we can distinguish two types of models:

**Parametric models** These models are defined by a finite, fixed number of parameters $\theta$. For parametric learning the algorithm is expected to *fit* the parameters to the training data i.e. find the model parameters that maximize its performance.

**Nonparametric models** Any model which is not parametric. Examples include models which are fully defined by the data, such as $k$-nearest neighbors classification. Models which have both parametric and nonparametric components are sometimes referred to as *semiparametric*. For example, a neural network with a predefined structure is parametric, but if the number of parameters or its layers are selected based on the data (hyperparameter tuning) this introduces a non-parametric aspect.

The archetypical example of a non-parametric model is the $k$-nearest neighbours algorithm: Given a set of points in space with category labels as training data, the model classifies unseen samples by assigning it the class that the majority of its neighbours (by some distance metric) belong to. A simple example of a parametric model is performing density estimation by fitting a Gaussian, which is parametrized by its mean and variance, to a set of real valued numbers. Note that the $k$-nearest neighbours model is not constrained and has infinite capacity i.e. the function it describes can become arbitrarily complex given an infinite amount of training data. The Gaussian on the other hand is limited in its complexity.

## 1.3 Supervised learning with parametric models

The models discussed in the chapters to follow are all parametric or semiparametric models where the parameters are trained with supervision, which is the setting that is formalized in this section. For the remainder of this work we will disregard unsupervised learning and reinforcement learning.

We consider a set of possible inputs, $\mathcal{X}$, and outputs, $\mathcal{Y}$.

**Definition 1.3.1.** *Let $(\Omega, \mathcal{E}, P)$ be the probability space induced by sample space $\Omega = \mathcal{X} \times \mathcal{Y}$, a set of events $\mathcal{E}$, and a probability measure $P$. Let $X : \Omega \to I$ and $Y : \Omega \to O$ be a pair of co-occuring random elements. A* dataset *$\mathcal{D}$ consists* *of $N$ independent realizations of these random elements i.e. $\mathcal{D} = \{(x_i, y_i) : i = 1, \ldots, N\}$.*

As an example, let $\mathcal{X}$ be a group of patients and let $\mathcal{Y}$ be their diagnosis. The random variable $X$ could be a set of $n$ health indicators in $\mathbb{R}^n$, while $Y$ is a one-hot encoding of the $m$ possible diagnoses in $\mathbb{R}^m$. A one-hot encoding is defined as a vector with a single non-zero entry, $([1 = y], \ldots, [m = y])$ where $y$ is the index of the correct diagnosis. Note however that the definition allows for $I$ and $O$ to contain more complicated elements e.g. in the case of structured prediction.

**Definition 1.3.2.** *A* model *is a function $f_\theta$ over the domain $I$ in a function space $\mathcal{F}$ parametrized by a non-empty set of parameters $\theta \in \Theta$, i.e. there exists a surjective mapping $g : \Theta \to \mathcal{F}$.*

The codomain of $f_\theta$ is often $O$ or the space of distributions over $O$. Building on the previous example, a model could map the health indicators $\mathbf{x}_i$ to a vector $\hat{\mathbf{y}}_i = f_{\boldsymbol{\theta}}(\mathbf{x}_i)$ containing scores for each diagnosis.

**Definition 1.3.3.** *Let $A$ be a space of actions. A* decision function *is a function, $\delta_\theta : I \to A$ in a function space $\mathcal{F}'$, parametrized by the same set of parameters $\theta$ as the model, $f_\theta$.*

Often $A$ is the same as the codomain of $f_\theta$, or a function of it. For example, the space of actions could be to give the doctor a score for each diagnosis, or to issue a single diagnosis for a specific patient $\mathbf{x}_i$ by taking $(\delta_{\boldsymbol{\theta}})_i = \arg\max(f_{\boldsymbol{\theta}}(\mathbf{x}_i))$. One can also imagine cases where additional actions such as ‘unknown’ are possible.

**Definition 1.3.4.** *An* error function *or* loss function *is a function $\mathcal{L} : A \times O \to \mathbb{R}$ that assigns a cost to the decision taken, $\delta_\theta(x_i)$, given the correct answer, $y_i$.*

A loss function for the diagnosis example could be $[\delta_{\boldsymbol{\theta}}(\mathbf{x}_i) = \arg\max(\mathbf{y}_i)]$ i.e. whether the class decided by the decision function is the correct one. The mean of this, $\frac{1}{n}\sum_{i=1}^{n}[\delta_{\boldsymbol{\theta}}(\mathbf{x}_i) = \arg\max(\mathbf{y}_i)]$, is known as the *classification rate.* For regression a possible loss function is $\|\hat{\mathbf{y}}_i - \mathbf{y}_i\|$.

The goal of learning is finding a set of parameters, $\theta^*$, that minimizes the *expected risk*, $R$, of our decision function. Given the bivariate probability distribution $p_{X,Y}(x, y)$ we define

$$R(\delta_\theta) = \int_{\mathcal{X}} \int_{\mathcal{Y}} \mathcal{L}(\delta_\theta(x), y) p_{X,Y}(x, y) dx dy$$

Since we do not have access to the distribution $p_{X,Y}(x, y)$, we must consider the *empirical risk*, $\hat{R}$, instead, which is an unbiased estimator of the expected risk (since our samples are independent realizations of a single pair of random elements).

$$\hat{R}(\delta_\theta; \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\delta_\theta(x_i), y_i)$$

The principle of *empirical risk minimization* implies that learning takes the form of finding the set of parameters

$$\hat{\theta}^* = \operatorname*{arg\,min}_{\theta \in \Theta} \hat{R}(\delta_\theta; \mathcal{D})$$

However, this learning procedure results in the empirical risk no longer being an unbiased estimator of the expected risk, since $\hat{\theta}^*$ has been optimized to minimize the average loss over a strict subset of $\mathcal{X} \times \mathcal{Y}$.

$$\mathbb{E}_{\mathcal{D}}[R(\delta_{\hat{\theta}^*}) - \hat{R}(\delta_{\hat{\theta}^*}; \mathcal{D})] \geq 0$$

We refer to $\hat{R}(\delta_{\hat{\theta}^*}; \mathcal{D})$ as the *training error* or *empirical error* and $R(\delta_{\hat{\theta}^*})$ as the *expected error* or *generalization error*. Their expected difference is referred to as the *generalization gap*. Note that the generalization gap cannot, in general, be explicitly computed; in statistical learning theory the objective is usually to find upper bounds on this error. (A second concern in the field of statistical learning theory is proving for specific function spaces $\mathcal{F}'$ that the empirical risk and expected risk converge in probability to the same value as the number of examples $N \to \infty$.)

The fact that the empirical risk is a biased estimator means that we can not use it as a performance measure of our model. This is why in practice the dataset is split into two parts: the *training set* and the *test set*. The test set will not play any part in the training process, but is only used at the end as an unbiased estimator of the expected risk.

Let $\delta^* \colon I \to A$ be a decision function which minimizes $R$ in the space of all possible functions mapping inputs to actions. We assume that our set of parametrized decision functions $\mathcal{F}' \subsetneq I \to A$.[i] We can now distinguish two types of error in our

i. Some argue that this assumption must be explicitly stated because even if we choose $\Theta = (0, 1)$ the fact that this set is uncountably infinite means that any computable function could be parametrized by it.

attempt to minimize the expected risk, both non-negative:

$$R(\delta_{\hat{\theta}^*}) - R(\delta^*) = \underbrace{R(\delta_{\hat{\theta}^*}) - R(\delta_{\theta^*})}_{\text{estimation error}} + \underbrace{R(\delta_{\theta^*}) - R(\delta^*)}_{\text{approximation error}} \tag{1.1}$$

The *estimation error* occurs because we are optimizing over a finite training set and hence are unlikely to find the best function with respect to generalization within our function space. The estimation error is what our learning algorithm tries to minimize. The *approximation error* is due to the fact that the function space parametrized by $\Theta$ does not necessarily include the optimal decision function $\delta^*$.

Starting from a singleton function space, it is clear that the estimation error must start at 0. As we increase the size of the function space $\mathcal{F}'$, the value of $R(\delta_{\theta^*})$ can only decrease. The value of $R(\delta_{\hat{\theta}^*})$ is lower bounded by $R(\delta_{\theta^*})$. On the other hand, the approximation error can only decrease since $R(\delta^*)$ is constant.

### 1.3.1 Overfitting

Considering equation 1.1 it might look like a good idea to make our function space as large as possible since this decreases the approximation error. However, it turns out that this is a bad idea. As the function space grows, it is actually possible for $R(\delta_{\hat{\theta}^*})$ to start increasing. If the estimation error grows faster than the approximation error, the expected risk will actually start increasing. This is a process known as *overfitting.*

The *Vapnik-Chervonenkis (VC) dimension* formalizes the concept of *capacity*, the size or complexity of the function space $\mathcal{F}'$, and can be used to prove a probabilistic upper bound that shows that overfitting is not only possible, but likely.

We define the VC dimension by considering the case of binary classification. Similar definitions for capacity can be derived for e.g. regression and density estimation, but they are more involved. We consider binary classification since multiclass classification can be reduced to a series of one-vs.-rest classifications. Consider a set of points $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ where $\mathbf{x}_i \in \mathbb{R}^n$ and corresponding labels $Y = \{y_1, \ldots, y_N\}$ with $y_i \in \{0, 1\}$.

**Definition 1.3.5.** *A set of functions $\mathcal{F}$ is said to* shatter *the set $X$ if and only if $\forall Y \in \{0,1\}^N \; \exists f \in \mathcal{F}$ such that $f(\mathbf{x}_i) = y_i$ for all $i$.*

If $f$ shatters $X$ it means that it can classify all the points correctly for any set of labels.

**Definition 1.3.6.** *The* Vapnik-Chervonenkis (VC) dimension *$h$ is a measure of the* capacity *of a set of functions and is equal to the cardinality of the largest set $X$ that is shattered by $\mathcal{F}$.*

An example that is often given is the classification of points on the Euclidean plane by separating them using a single straight line, as a perceptron would do (see Section 1.4.1). Three points can always be separated (shattered) into two classes, regardless of their labelling, but this is not the case for 4 points (Radon's theorem). Hence, the VC dimension of this classifer is 3.

In V. N. Vapnik (1995, chapt. 3) it is proven that for a training set size $N$ and a set of functions bounded by $a$ and $b$ with VC dimension $h$

$$P\left(R(f) - \hat{R}(f) \leq \frac{b-a}{2}\sqrt{\frac{h(\log(2N/h)+1) - \log(\eta/4)}{N}}\right) \geq 1 - \eta \tag{1.2}$$

for all $\theta$ and with $\eta > 0$. The square root term increases monotonically. This shows that as the capacity of the model increases our generalization error is likely to increase and our empirical error might no longer be representative of the expected error.

Vapnik's bound on the generalization error shows that it is theoretically possible for the expected error to increase in the case where the empirical risk goes down more slowly than the probabilistic upper bound on the generalization gap.

## 1.3.2 Bias-variance tradeoff

We can analyze the occurence of overfitting by means of the bias-variance decomposition, analyzing the generalization error as a sum of three terms: the *bias*, *variance*, and *irreducible error*. Considering the following example following "The Elements of Statistical Learning" (Hastie, Tibshirani, and Friedman, 2001). We assume that our dataset consists of samples $\mathbf{x}_i \in \mathbb{R}^n$ and noised targets $y'_i = y_i + \epsilon$ with $y_i \in \mathbb{R}$, $\mathbb{E}[\epsilon] = 0$ and $\text{Var}[\epsilon] = \sigma^2$. Consider a squared loss function,

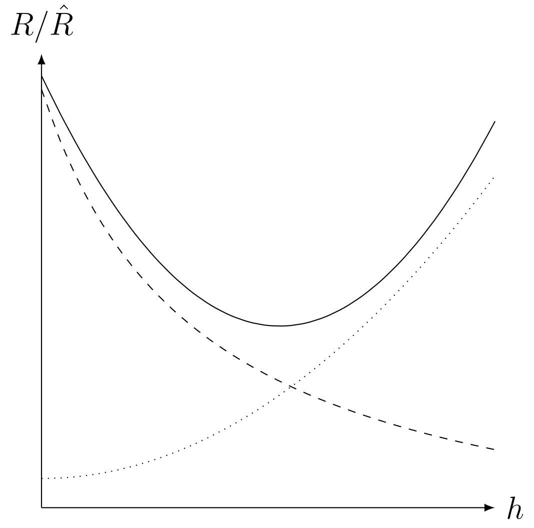


**Figure 1.1** – The expected error, $R(\delta_{\hat{\theta}^*})$, (solid), empirical error, $\hat{R}(\delta_{\hat{\theta}^*};\mathcal{D})$, (dashed) and the generalization error upper bound (see formula 1.2; dotted) as a function of the model capacity $h$. Note that this is an extreme example which assumes that the empirical error shrinks at a slower rate than the upper bound grows, resulting in the expected error increasing. From formula 1.2 alone it is not clear whether the expected error will start increasing or simply level off.

$\mathcal{L}(\delta_\theta(\mathbf{x}_i), y_i) = (y_i - \delta_\theta(\mathbf{x}_i))^2$. Let $\hat{y}_i = \delta_\theta(\mathbf{x}_i)$, then

$$\begin{aligned}
\mathbb{E}[(y_i' - \hat{y}_i)^2] &= \mathbb{E}[(y_i' - y_i + y_i - \hat{y}_i)^2] \\
&= \mathbb{E}[(y_i' - y_i)^2] + \mathbb{E}[(y_i - \hat{y}_i)^2] + 2\mathbb{E}[(y_i' - y_i)(y_i - \hat{y}_i)] \\
&= \sigma^2 + \mathbb{E}[(y_i - \hat{y}_i)^2] + 2(\underbrace{\mathbb{E}[(y_i' y_i)] - \mathbb{E}[y_i^2]}_{0} - \underbrace{\mathbb{E}[y_i'\hat{y}_i] + \mathbb{E}[y_i\hat{y}_i]}_{0}) \\
&= \sigma^2 + \mathbb{E}[(y_i - \mathbb{E}[\hat{y}_i] + \mathbb{E}[\hat{y}_i] - \hat{y}_i)^2] \\
&= \sigma^2 + \mathbb{E}[(y_i - \mathbb{E}[\hat{y}_i])^2] + \mathbb{E}[(\mathbb{E}[\hat{y}_i] - \hat{y}_i)^2] + 2\underbrace{\mathbb{E}[(y_i - \mathbb{E}[\hat{y}_i])(\mathbb{E}[\hat{y}_i] - \hat{y}_i)]}_{0} \\
&= \sigma^2 + \mathbb{E}[(y_i - \mathbb{E}[\hat{y}_i])^2] + \mathrm{Var}[\hat{y}_i]
\end{aligned}$$

The $\sigma^2$ is the irreducible error and forms a lower bound on the generalization error. The bias represents the error coming from the model. If the model is unable to capture some patterns in the data (because it does not have enough capacity) the bias will be high. The variance of the learning algorithm represents the sensitivity of our model to the random sampling of training examples from the data distribution. A high variance is likely to increase the generalization error because the model will be fit to a distribution that is likely different from the underlying data distribution. A similar decomposition can be derived for classification problems as well.

As we increase our model's capacity the bias will be reduced, but the variance will increase. If we increase capacity too much the variance will often increase faster than the bias, giving rise to overfitting; at this point the model is memorizing examples rather than modelling the underlying distribution. To avoid overfitting we limit the capacity of our model. For example, by considering a smaller parameter space, or through a *regularization* term which assigns a cost to each function, $\Omega : \mathcal{F}' \to \mathbb{R}$, and a regularization weighting $\lambda \in \mathbb{R}$.

$$\hat{\theta}^* = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\delta_\theta(x_i), y_i) + \lambda \Omega(\delta_\theta)$$

The regularization term penalizes the complexity of the function $\delta_\theta$ in some way. It can be seen as enforcing Occam's razor, encouraging the solution to be the simplest one that models the data. In many cases the regularization term can also be viewed from the Bayesian perspective as a prior on the model parameters. We will discuss regularization methods specific to neural networks in Section 1.5.1.

The theory discussed thus far underpins the training process that will be used throughout this work, namely with a separate training and test set and a variety of regularization methods in order to maximize the model performance.

## 1.4 Artificial neural networks

The term 'neural networks' has been used to describe a wide range of models. What these models have in common is that they can be interpreted as units (neurons) connected by weights (synapses) forming a network. A set of neurons is activated by the input data, after which they exchange information with the neurons that they are connected to. These neurons often respond to the incoming information in a non-linear way. The weights of the network are adaptive and can be tuned by a learning algorithm.

Neural networks have their origins in simple mathematical models of biological neural networks (McCulloch and Pitts, 1943; Rosenblatt, 1962), but the field has

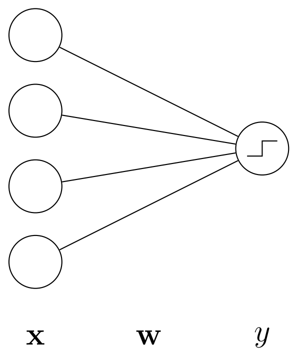


**Figure 1.2** – A visualization of the perceptron as a neural network with weights $\mathbf{w}$ going from the input $\mathbf{x}$ to the output $y$. Note that the bias was ommitted for illustrative purposes.

since diverged into those that endeavour to develop biologically plausible models of the nervous system (computational neuroscience), and those which develop efficient models for machine learning. This work falls entirely in the second category.

For all the models discussed henceforth the events space of the decision function is the same as the target space of the model, so we will use $f_\theta$ instead of $\delta_\theta$. Unless stated otherwise the models can be assumed to map from $\mathbb{R}^n$ to $\mathbb{R}^m$ and the parameter space $\Theta = \mathbb{R}^k$.

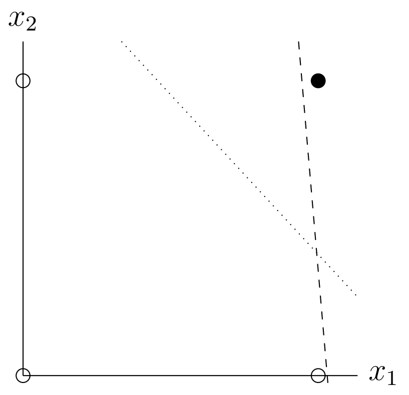
### 1.4.1 Perceptron

One of the first, and simplest, neural networks developed was the *perceptron*, visualized in figure 1.2. The model is a *linear classifier* parametrized by $\theta = (\mathbf{w}, b) \in \mathbb{R}^n \times \mathbb{R}$ which assigns one of two classes to $n$-dimensional inputs, $f_\theta \colon \mathbb{R}^n \to \{0, 1\}$. It is defined as $\mathbf{x}_i \mapsto [\mathbf{w}^T\mathbf{x}_i + b > 0]$.

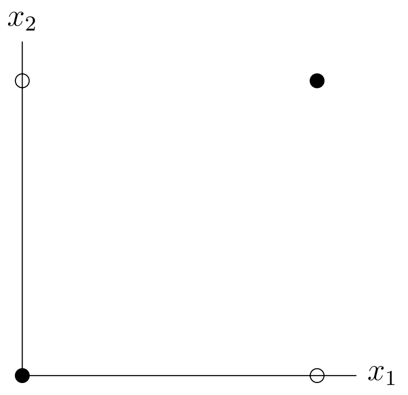
The elements of the vector $\mathbf{w}$ are referred to as the *weights*, connecting the inputs $\mathbf{x}_i$ to the output neuron. The parameter $b$ is referred to as a *bias.*[i] The weighted sum of the inputs plus the bias, $\mathbf{w} \cdot \mathbf{x}_i + b$, is sometimes known as the *pre-activation* of a neuron. An *activation function* transforms the pre-activation into the neuron's *output activation*, $\hat{y}_i = f_\theta(\mathbf{x}_i)$. In the case of the perceptron the activation function is the Heaviside step function $H(x) = [x \in (0, \infty)]$.

Geometrically the perceptron can be interpreted as drawing a hyperplane in $\mathbb{R}^n$, forming a decision boundary. The value of $f_\theta(\mathbf{x})$ depends on which side of

i. Note that one can define an equivalent network without biases by augmenting the inputs $\mathbf{x}' = [\mathbf{x}, 1]$ and using weights $\mathbf{w}' = [\mathbf{w}, b]$.



**(a)** The AND circuit in two dimensions. The two classes (circles and discs) are linearly separable by any number of lines.



**(b)** The XOR (exclusive-or) circuit in two dimensions. The two classes (circles and discs) are not separable by any line.

**Figure 1.3**

the hyperplane the point $\mathbf{x}_i$ lies. The bias $b$ allows the decision boundary to move away from the origin. This interpretation makes clear that the perceptron can only reach zero error if the two classes of points are linearly separable (i.e. if their convex hulls are disjoint). A typical example of a non-linearly separable set of points is $\mathbf{x} = (x_1, x_2) \in \{0, 1\}^2$ whose class labels are determined by the XOR function, $y = x_1 \oplus x_2$ (see figure 1.3b).

### 1.4.2 Stochastic gradient descent

Neural networks are most often formulated in such a way that they are differentiable and are trained with variations on the *stochastic gradient descent (SGD)* algorithm (see algorithm 1). SGD is an *online algorithm*, which means that for each

**Algorithm 1** Stochastic gradient descent (SGD) with mini-batches of size $M$

```
Initialize parameters $\boldsymbol{\theta}$
while not converged do
    Let $\sigma$ be a permutation of $(1, \ldots, N)$
    for $j$ in $0, \ldots, \frac{N}{M} - 1$ do
        $B = (\sigma(jM+1), \ldots, \sigma(jM+M))$                    ▷ Select a mini-batch
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \sum_{i \in B} \frac{\partial}{\partial \boldsymbol{\theta}} \mathcal{L}(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$   ▷ Estimate gradient
    end for
end while
return $\boldsymbol{\theta}$
```

iteration it looks at a single example or a small subset of the dataset and approximates the derivative of the loss function with respect to the parameters. Although theoretically each example should be taken at random from the training set, in practice SGD is often used by cyclically iterating over the training set to reduce the number of random memory accesses. Given the approximation of the gradient, the algorithm takes a gradient descent step in the parameter space in order to minimize the loss, and repeats this action until convergence. The scaling of the step taken, $\eta$, is known as the *learning rate*. A higher learning rate leads to faster minimization, but if the learning rate is too high the algorithm can become unstable. Online algorithms are feasible for our setting because our expected error is a linear combination of the loss for each sample in the dataset, $\hat{R}(f_\theta; \mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$, which makes it feasible to estimate $\hat{R}$ using a subset of the data, given that this subset was sampled i.i.d. from the data distribution.

The loss function of perceptron model previously discussed has a zero gradient almost everywhere because of the Heaviside step function, which makes it unsuitable for gradient optimization. However, we can apply SGD on the pre-activation value, $f'_\theta(\mathbf{x}_i) = \mathbf{w}^T\mathbf{x}_i + b$, instead. As a loss function we use $\mathcal{L}(f'_\theta(\mathbf{x}_i), y_i) = -y_i f'_\theta(\mathbf{x}_i)[f_\theta(\mathbf{x}_i) \neq y_i]$, where $y_i \in \{-1, 1\}$ instead of $y_i \in \{0, 1\}$ to simplify notation. This function assigns a non-zero loss to incorrectly classified examples equal to their distance to the hyperplane.

$$\begin{aligned}
\frac{\partial \mathcal{L}\left(f'_\theta(\mathbf{x}_i), y_i\right)}{\partial \mathbf{w}} &= -\frac{\partial}{\partial \mathbf{w}}\left(y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right)\right)[f_\theta(\mathbf{x}_i) \neq y_i] \\
&= -y_i\mathbf{x}_i[f_\theta(\mathbf{x}_i) \neq y_i] \\
\frac{\partial \mathcal{L}\left(f'_\theta(\mathbf{x}_i), y_i\right)}{\partial b} &= -y_i[f_\theta(\mathbf{x}_i) \neq y_i]
\end{aligned}$$

If we apply these updates only when the model misclassifies a point, we have derived the traditional perceptron training algorithm due to Rosenblatt (1958) as given in algorithm 2. Each update in effect moves the decision boundary towards the point. For the perceptron this learning algorithm is guaranteed to converge given that the points are linearly separable. Note that this convergence guarantee does not apply to neural networks in general since they are seldom convex. Optimization methods will be discussed further in Section 1.5.

**Algorithm 2** Perceptron algorithm

```
$\mathbf{w} \hookleftarrow \vec{0}$, $b \hookleftarrow 0$
$i \hookleftarrow 1$
while $\exists i$ s.t. $[\hat{y}_i \neq y_i]$ do                ▷ Until all examples are correctly classified
    $\hat{y}_i \hookleftarrow \mathbf{w}^T\mathbf{x}_i + b$
    if $[\hat{y}_i \neq y_i]$ then                              ▷ $y_i$ is incorrectly classified
        $\mathbf{w} \hookleftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$                            ▷ $0 < \eta <\leq 1$
        $b \hookleftarrow b + \eta y_i$
    end if
    $i \hookleftarrow (i \bmod N) + 1$
end while
```

### 1.4.3 Logistic and multinomial logistic regression

The perceptron model has its shortcomings. One issue is that the perceptron training algorithm does not distinguish between different hyperplanes that separate the training data. Given the two lines in figure 1.3a for example, we would prefer the dotted line over the dashed since it is more likely to generalize to unseen data. Efforts to develop algorithms that maximize the margins between the hyperplane and the training points resulted in what is now known as the linear *support vector machine (SVM)* (Cortes and V. Vapnik, 1995). Another approach that maximises the margins is *logistic regression* (Cox, 1958; Walker and Duncan, 1967). This approach can be considered a variation of the perceptron model which allows for a probabilistic interpretation by replacing the Heaviside step function with the *sigmoid function* (Verhulst, 1845), $\sigma : \mathbb{R} \to (0, 1)$, also called the *logistic function* (see figure 1.4)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

To simplify notation, we will consider a classification problem with labels $y_i \in \{0, 1\}$. The sigmoid function allows for a probabilistic interpretation of our model with $f_\theta(\mathbf{x}_i)$ estimating the probability that our sample belongs to class 1, $f_\theta(\mathbf{x}_i) = \hat{P}(y_i = 1 \mid \mathbf{x}_i; \theta)$. Note that $1 - f_\theta(\mathbf{x}_i) = \hat{P}(y_i = 0 \mid \mathbf{x}_i; \theta)$. Since we assume the samples in our dataset to be i.i.d. we can say that the conditional likelihood under our model of the pairs of $\mathbf{x}_i$ and $y_i$ in the dataset, $\hat{P}(\mathcal{D})$, is equal to $\prod_{i=1}^{N} f_\theta(\mathbf{x}_i)^{y_i}(1 - f_\theta(\mathbf{x}_i))^{1-y_i}$. A reasonable thing to do is to maximize the likelihood of the data, a principle known as *maximum-likelihood estimation (MLE)*. In

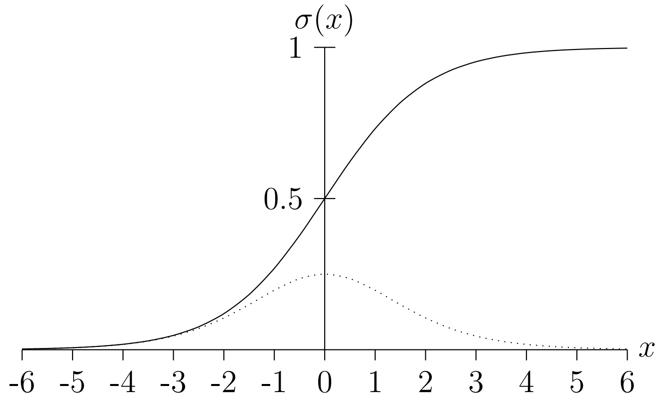


**Figure 1.4** – The logistic function (solid) and its derivative (dotted)

order for our cost function to be a linear combination of the samples we consider the log-likelihood of our model instead, and to maintain the convention that we want to minimize our cost function, we will use the *negative log-likelihood* as our cost function.[i]

$$\begin{aligned}\mathcal{L}(f_\theta(\mathbf{x}_i), y_i) &= -\log(f_\theta(\mathbf{x}_i)^{y_i}(1 - f_\theta(\mathbf{x}_i))^{1-y_i}) \\ &= -y_i \log(f_\theta(\mathbf{x}_i)) - (1 - y_i)\log(1 - f_\theta(\mathbf{x}_i)).\end{aligned}$$

Note that since the logarithm is a monotonically increasing function, it does not change the location of the minimum of our cost function.

The perceptron and logistic regression model can both be considered feedforward neural networks without hidden layers. A neural network layer is of the form

$$f_\theta(\mathbf{x}_i) = \phi\left(\mathbf{W}\mathbf{x}_i + \mathbf{b}\right)$$

where $\phi\left(\cdot\right)$ is an activation function which is applied element-wise[ii] (see figure 1.5).

i. In the context of neural networks it is often said that we minimize *cross-entropy*. The cross-entropy of the empirical data distribution, $p$, and our model's output distribution, $\hat{p}$, is defined as $H(p, \hat{p}) = \mathbb{E}_p[-\log \hat{p}]$. Let $p$ be a discrete distribution that gives the same probability to each sample in our dataset $\mathbf{x} = \{x_1, \ldots, x_n\}$ i.e. $p(x) = \frac{1}{n}\sum_{i=1}^n \delta(x - x_i)$. Then $H(p, \hat{p}) = -\int_X p(x)\log \hat{p}(x)dx = -\frac{1}{n}\sum_{i=1}^n \log \hat{p}(x_i)$, which is the negative log-likelihood under our model. Also note that $H(p, \hat{p}) = H(p) + D_{KL}(p\|\hat{p})$ where $H$ is the entropy and $D_{KL}$ is the Kullback-Leibler divergence (also called relative entropy). Since $H(p)$ is fixed, minimizing the cross-entropy is equivalent to minimizing the KL-divergence.

ii. Note that convolutions are linear and can be represented by using Toeplitz matrices for $\mathbf{W}$. However, some activation functions in common use are not applied element-wise (e.g., max pooling or maxout units).

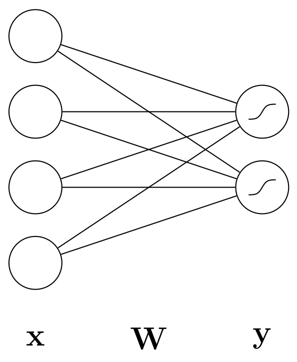


**Figure 1.5** – A single-layer neural network without hidden layer. If the softmax function is used as the activation function, this is equivalent to multinomial logistic regression.

In order to use gradient descent methods like SGD we require $\phi$ to be differentiable almost everywhere and not have a zero gradient almost everywhere. The input layer $\mathbf{x}$ is fully connected to the output layer $\mathbf{y}$ through the matrix multiplication with $\mathbf{W}$. In the case of the perceptron and logistic regression $\mathbf{y} \in \mathbb{R}^1$ and $\mathbf{W} \in \mathbb{R}^{1 \times n}$, and the Heaviside step function and logistic function were used as activation functions respectively. In the case of *regression* the output does not need to be in the interval $[0, 1]$ and other activation functions, including the identity, can be used for the last layer.

In general a neural network can be vector-valued, $\mathbf{y} \in \mathbb{R}^m$. For multiclass classification a generalized version of the logistic function called the *softmax function* (Bridle, 1990) (or *normalized exponential function*) is often used in the output layer. The softmax of a vector $\mathbf{x} = (x_1, \ldots, x_n)$ is defined by

$$\operatorname{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}.$$

This produces a valid categorical probability distribution as the output. In fact, a single-layer perceptron with a softmax output is equivalent to multinomial logistic regression. The softmax function is differentiable and has the same desireable properties as the logistic function for gradient-based optimization methods.

Neural network models can also be used for *multi-label classification*, where $\mathbf{y} \in \mathbb{R}^m$ and multiple answers are correct, for example when predicting which objects are present in a picture both 'house' and 'tree' could be correct at the same time. In these cases the problem is often approached as a series of binary

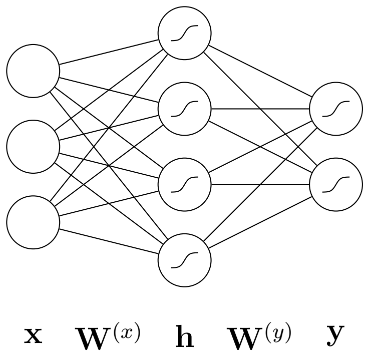


**Figure 1.6** – A neural network with a single hidden layer.

classifications by applying the sigmoid function to each output independently. Note that this assumes conditional independence of the labels. However, since the binary classifiers share the majority of the model parameters, the neural network can model the shared factors (i.e. representations of labels that are highly correlated will end up close together) (M.-L. Zhang and Zhou, 2006).

### 1.4.4 Multilayer feedforward neural networks

Layers can be stacked to form neural networks with multiple hidden layers, the output of one layer forming the input to another (see figure 1.6).[i]

In 1974 Paul Werbos first proposed using the *backpropagation algorithm* to train these multilayer networks (Werbos, 1974). The use of backpropagation for the training of neural networks gained further recognition in the 1980s (David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, 1986), leading to a renewed interest in the field. The backpropagation algorithm is effectively an efficient method of calculating the partial derivatives of the cost function with respect to the model's parameters (the weights and biases of each layer) using the chain rule. The discovery of this algorithm can be traced back to the 1960s (Bryson and Denham, 1962; Dreyfus, 1962; Pontryagin et al., 1962). Backpropagation is in fact a special case of reverse-mode automatic differentiation (AD), which is discussed in detail in Chapter 9.

Multilayer neural networks are able to classify non-linearly separable data such

i. Often referred to as a *multilayer perceptron (MLP)*, but this is technically a misnomer since the method is only applicable to networks that unlike the perceptron use activation functions that do not have a zero gradient almost everywhere.

as the XOR problem in figure 1.3b. Their power is given theoretical grounding in the *universal approximation theorem* (Hornik, Stinchcombe, and White, 1989) (sometimes called Cybenko theorem, after the person who proved one of the first versions in 1989 (Cybenko, 1989)).

**Theorem 1.4.1.** *Let $\phi$ be a non-constant, bounded, and monotonically-increasing continuous function. Given any $\varepsilon > 0$ and continuous function $f(\mathbf{x}) : [0,1]^n \to \mathbb{R}$ there exists a layer-size $N$, weight matrices $\mathbf{W}$ and $\mathbf{w}$, and biases $\mathbf{b}$ such that a neural network, $F(\mathbf{x})$, with a single hidden layer and linear output,*

$$F(\mathbf{x}) = \sum_{i=1}^{N} \mathbf{w}^T \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

*can approximate $f(\mathbf{x})$ to within $\varepsilon$, that is, $\forall \mathbf{x} \in [0,1]^n$*

$$|F(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$$

More recent theoretical work has shown that the representational power of neural networks grows much faster by adding depth than by adding width (Montufar et al., 2014; Poole et al., 2016). Despite their high complexity and non-convexity, the learning dynamics for deep networks are relatively stable (Saxe, James L. McClelland, and Ganguli, 2014; Choromanska et al., 2015; Dauphin et al., 2014).

### 1.4.5 Recurrent neural networks

Multilayer perceptrons are generally stateless; each data sample is processed in isolation. Recurrent neural networks (RNNs) address this issue by introducing feedback connections in the hidden layers. Given a sequence of inputs $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$, the hidden state and output are for example calculated as follows

$$\begin{aligned}
\mathbf{h}_t &= \phi^{(h)}\left(\mathbf{W}^{(x)}\mathbf{x}_t + \mathbf{W}^{(h)}\mathbf{h}_{t-1} + \mathbf{b}^{(h)}\right) \\
\mathbf{y}_t &= \phi^{(y)}\left(\mathbf{W}^{(y)}\mathbf{h}_t + \mathbf{b}^{(y)}\right)
\end{aligned}$$

RNNs are able to deal with variable length inputs and outputs, making them appropriate for time series prediction, signal processing, or natural language processing. Note that in the ideal case there is a causal relationship between the subsequent

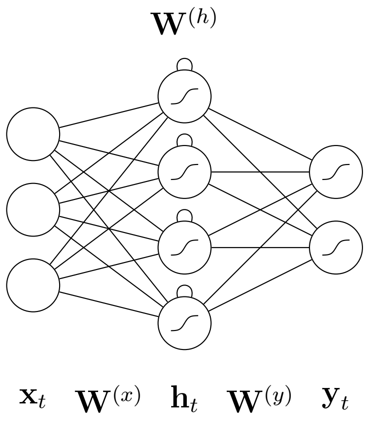


**Figure 1.7** – A recurrent neural network. Note that the recurrent layer is normally fully connected i.e. there should be arrows from all nodes in $\mathbf{h}_t$ to all nodes in $\mathbf{h}_t$. For the sake of clarity these lines were omitted here.

elements. Training is performed by *backpropogation through time (BPTT)* which effectively "unfolds" the RNN as if it were a very deep MLP and performing regular SGD. This can cause optimization problems such as the *vanishing and exploding gradient* problem (Razvan Pascanu, Mikolov, and Y. Bengio, 2013; Y. Bengio, Simard, and Frasconi, 1994; Hochreiter, 1991). Consider the gradient flow from $\mathbf{h}_t$ to $\mathbf{h}_1$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_1} = \prod_{i=2}^{t} \mathbf{W}^{(h)} \operatorname{diag} \phi'(\mathbf{W}^{(x)}\mathbf{x}_i + \mathbf{W}^{(h)}\mathbf{h}_{i-1} + \mathbf{b}^{(h)})$$

If $\exists\gamma$ s.t. $\rho(\phi'(x)) < \gamma$ for all $x$, where $\rho$ is the spectral radius, we can bound

$$\rho\left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_1}\right) \leq \gamma^{t-1} \rho\left(\mathbf{W}^{(h)}\right)^{t-1}$$

and use the properties of the spectral radius to conclude that $\rho\left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_1}\right) \to 0$ if $\rho(\mathbf{W}^{(h)}) < \frac{1}{\gamma}$ and $\rho\left(\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_1}\right) \to \infty$ if $\rho(\mathbf{W}^{(h)}) > \frac{1}{\gamma}$ as $t \to \infty$. If we assume that Note that we have $\gamma = 1$ for $\phi = \tanh$ and $\gamma = \frac{1}{4}$ for $\phi = \sigma$. Exploding gradients can be addressed by clipping the norm of the gradient (Razvan Pascanu, Mikolov, and Y. Bengio, 2013).

Recurrent networks are sometimes trained on long sequences with *truncated backpropagation through time (TBPTT)* (Ronald J Williams and Peng, 1990), which means that gradient flows involving more than a certain number of transitions are ignored. This reduces the amount of memory that is required and also reduces the chance of exploding gradients.

In applications where we can assume that the entire sequence is available before our model needs to produce an output it is possible to run a recurrent neural network in two directions. This is known as a *bidirectional recurrent neural network* (M. Schuster and Paliwal, 1997; Alex Graves and Jürgen Schmidhuber, 2005). It involves two independent RNNs processing the sequence $(x_1, \ldots, x_T)$ and its reverse $(x_T, \ldots, x_1)$ respectively. The networks will produce two series of hidden states, $(h_1, \ldots, h_T)$ and $(h'_T, \ldots, h'_1)$. Two states $h_t$ and $h'_t$ are then combined, often by concatenating them or summing an affine transformation of them. The result can be seen as a representation of the entire sequence, but possibly focused around a single element of it.

### LSTM

Besides being an optimization problem, vanishing gradients impede the learning procedure since they force the RNN to forget information over time when the transition's operator norm is smaller than 1. Long short-term memory (LSTM) units were developed (Hochreiter and Jürgen Schmidhuber, 1997) in order to address this problem. The core difference is that instead of overriding the state, $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$, the old state is partially carried over, allowing both the information and the gradient to flow unaffected by the transition operator

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{x}_t) \odot f(\mathbf{h}_{t-1}, \mathbf{x}_t) + (1 - g(\mathbf{h}_{t-1}, \mathbf{x}_t)) \odot \mathbf{h}_{t-1}$$

The precise structure of the LSTM is more complicated (and different authors tend to use slight variations). It is perhaps best understood graphically (see figure 1.8). LSTMs have been battle-tested and studied extensively (Greff et al., 2017; Jozefowicz, Zaremba, and Sutskever, 2015) and are still the architecture of choice for most RNN models. A commonly used, and computationally cheaper, alternative is the GRU, which will be presented in Chapter 4.

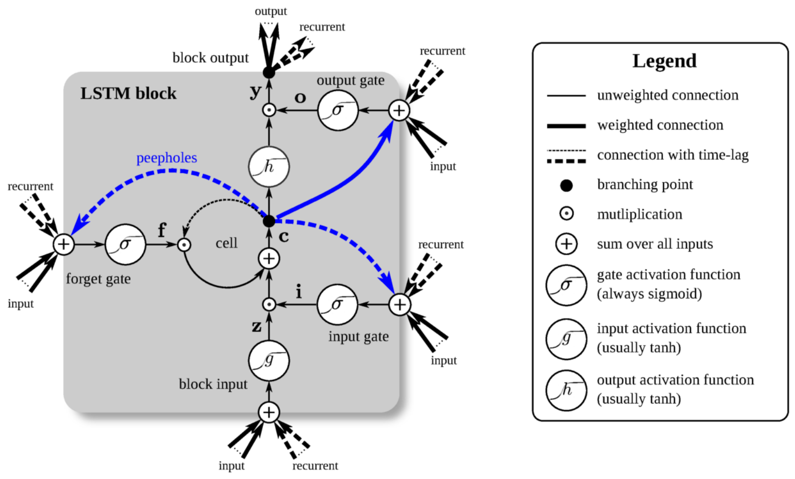


**Figure 1.8** – A long short-term memory (LSTM) unit. The cell, **c**, is the main hidden state of the unit. A weights matrix and a recurrent weights matrix are used to combine the previous cell state and the input into a proposal state. Then three weights and recurrent weights matrices each are used with a sigmoid activation to determine which elements from the cell to forget, which elements to incorporate from the proposal state, and which elements to output. Image from Greff et al. (2017).

## 1.5 Optimization

To simplify notation we will write $f(\boldsymbol{\theta})$ to mean the loss function as a function of the parameters alone, while considering the inputs fixed i.e. $\nabla f(\boldsymbol{\theta})$ is the gradient with respect to the parameters. Stochastic gradient descent (SGD) as introduced in Section 1.4.2 still forms the basis of most neural network training algorithms. Many variations and extensions to SGD have been proposed, with a variety of justifications: momentum (David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams, 1986), Nesterov momentum (Nesterov, 1983), Adagrad (Duchi, Hazan, and Singer, 2011), Adadelta (Zeiler, 2012), RMSprop (G. Hinton, 2012), and Adam (Kingma and J. Ba, 2015), are just some of the more common variants. For the sake of brevity we will only introduce Adagrad and its extension Adadelta in this section, which is the algorithm used for training some of the models described in Chapter 4.

For convex functions SGD has a sublinear convergence rate. In the batch case, the convergence rate for functions $f_\theta \in C^2$ can usually be increased by using *curvature* information i.e. the second derivative (the Hessian matrix, $\mathbf{H}$). In classical optimization this is a common approach known as *Newton's method*, where

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - (\mathbf{H}(f)(\boldsymbol{\theta}))^{-1}\nabla f(\boldsymbol{\theta})$$

Unlike gradient descent, which has linear convergence, Newton's method converges quadratically for convex functions. For a simple graphical interpretation, see figure 1.9.

Moreover, in the case of neural networks and large-scale optimization in general calculating the Hessian explicitly is often infeasible. It requires $\frac{1}{2}N(N+1)$ storage and $O(N^2)$ computation, where $N$ is the number of parameters. Inverting the Hessian requires a further $O(N^3)$ computation.[i] Many of the methods previously mentioned can be interpreted as performing some diagonal approximation of the function's curvature.

### 1.5.1 Regularization

In Section 1.3.2 we discussed the concept of overftting and explained that the bias-variance tradeoff can be controlled by adding *regularization*. John von Neumann famously said (Dyson, 2004)

> With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.

This makes regularization particularly important for neural networks, which can have billions of parameters (Jeffrey Dean et al., 2012) and are heavily overparametrized (Denil et al., 2013; G. Hinton, Vinyals, and Jeff Dean, 2015). An overview of the many regularization techniques used in neural networks (Goodfellow, Y. Bengio, and Courville, 2016) is beyond the scope of this work, but we will give a brief overview of the regularization methods that will be mentioned and used in later sections:

**Norm constraints** Adding the $L^2$-norm of parameters is a common form of regularization known as weight decay, limiting the function space $\mathcal{F}$ to those defined by parameters lying more closely to the origin. Adding an $L^1$-norm penalty is a common way of promoting sparsity in the activations when needed.

i. Inversion is the same complexity as matrix multiplication, so technically $O(n^{2.37\cdots})$ with Coppersmith-Winograd algorithms, but in practice no faster than $O(n^{2.807})$ using the Strassen algorithm.

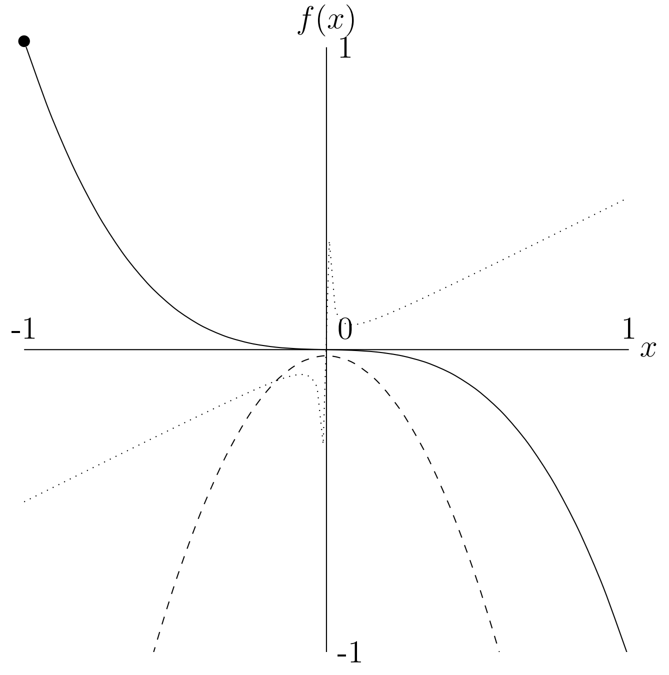


**Figure 1.9** – Consider the function $f(x) = -(x^3 + \epsilon x)$ with $\epsilon \ll 1$ (solid) being minimized using a gradient method starting from the point $(-1, 1 + \epsilon)$. The norm of the gradient, $f'(x) = -(3x^2 + \epsilon)$ determines the speed at which we progress i.e. the size of the step we take to the right (dashed). Note that $f'(0) = \epsilon$, which can be arbitrarily small, so our optimization is likely to progress very slowly around this point. Second order methods rely on the inverse of the second derivative, $f''(x) = -6x$, to scale the step size. This means that we now take steps of size $f'(x)/f''(x) = \frac{1}{2}x + \frac{\epsilon}{6x}$ (dotted). Note the very different behaviour of this function: If $x$ is not near 0, we progress at a more even rate through parameter space. In the flat region around 0 we do not slow down, instead the size of the steps actually increases, allowing us to escape this region quickly. However, we can also see that at $x = 0$ our algorithm could become unstable. Moreover, the use of the second derivative has reversed our step direction when $x > 0$. Instead of minimizing the function by finding $x \to \infty$, Newton's method for non-convex functions converges to any criticial point i.e. where $f''(x) = 0$. We can remediate this by only using the magnitude of the curvature, $f'(x)/|f''(x)|$. Note that in the multivariate case, this implies using the 'absolute Hessian', $|H| = Q|\Lambda|Q^T$ where $H = Q\Lambda Q^T$ is the eigendecomposition (Gould and Nocedal, 1998; Greenstadt, 1967; Razvan Pascanu, Dauphin, et al., 2014). Trust region methods and line searches with Wolfe conditions are other ways of improving Newton's method's stability and avoiding convergence to saddle points (Nocedal and Wright, 2006).

**Data augmentation** Is a heuristic where we augment our dataset with samples that were transformed in a way that we believe our model should be agnostic to. For example, an object recognition model should be robust to images being rotated slightly or flipped horizontally.

**Noise** Noise can be injected in many stages, and justified in multiple ways. When noise is added to the inputs it can be considered a form of data augmentation. If Gaussian noise is added to the weights it can be interpreted as a variational Bayesian method with a uniform prior on the weights (Alex Graves, 2011). Adding noise to the gradients themselves has also been shown to improve performance, and could perhaps be likened to simulated annealing (Neelakantan et al., 2015).

**Dropout** A special case of weight noise where all the incoming weights to half the units are set to 0 during training (Srivastava et al., 2014). This can be interpreted as training an exponential number of networks, $\prod_{i=1}^{L} \binom{n_i}{n_i/2}$ where $n_i$ is the number of units in the $i$th layer, and averaging their predictions at test time. Dropout can be used in RNNs, but is generally not applied to the recurrent weights (V. Pham et al., 2014) or the same mask is applied across time steps (Gal, 2016; Moon et al., 2015; Semeniuta, Severyn, and Barth, 2016).

**Low rank approximation** Instead of learning a weights matrix $\mathbf{W} \in \mathbb{R}^{n\times m}$ we learn a low rank approximation in the form of $\mathbf{W}^{(1)}\mathbf{W}(2)$ with $\mathbf{W}^{(1)} \in \mathbb{R}^{n\times k}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{k\times m}$ and $k(n+m) < nm$, forcing the network to represent the linear transformation from $n$ to $m$ with fewer parameters.

### 1.5.2 Parameter initialization

The way in which parameters are initialized can be important. If the network is deep and the gradient operator have a spectral radius far away from one, the gradient signal can quickly vanish or explode; the explanation of the vanishing gradient problem in 1.4.5 applies to deep networks as well. Glorot and Y. Bengio (2010) analyze the variance of the gradients in a feedforward network and suggest sampling the weights going from layer $i$ to $i+1$ from the uniform distribution $\mathcal{U}(\frac{\sqrt{6}}{\sqrt{n_i+n_{i+1}}})$.

Saxe, James L. McClelland, and Ganguli (2014) analyze the learning dynamics as the number of layers goes to infinity and discover that a particular class of orthogonal weights matrices leads to stable learning dynamics. They show experimentally that random orthogonal matrices too show such stable behaviour. An intuitive explanation is that orthogonal matrices are norm-preserving. Following our derivation for the exploding and vanishing gradient problem in Section 1.4.5 we see that this is a class of weights matrices for which the gradient norm remains stable, assuming that our activation function's gradient is also approximately one.

# 2 Sequence-to-sequence learning for machine translation

The field of natural language processing has the goal of allowing computers to derive meaning from natural language. The field comprises many well-studied subtasks. In this work we will consider two: *language modelling* and *machine translation*. Language modelling is the task of assigning a probability to a text (string). In the usual approach a string of characters is *tokenized*, splitting it up into a sequence of words and punctuation (collectively referred to as *tokens*) using heuristics. The set of distinct tokens $E$ is referred to as the *vocabulary* or *dictionary*. Our task is now to estimate $p(\mathbf{e})$ where $\mathbf{e} = e_1, \ldots, e_n$ with $e_i \in E$. This technically is density estimation, an unsupervised learning task. However, we can apply the probability chain rule to factorize $P(\mathbf{e})$ into a series of conditional probabilities, allowing us to use supervised models and auto-regressive architectures instead:

$$P\left(e_1, \ldots, e_n\right) = \prod_{i=1}^{n} P\left(e_i \mid e_1, \ldots, e_{i-1}\right)$$

Machine translation can abstractly be seen as conditional language modelling, where we try to model the likelihood of a translation $\mathbf{e}$ in the target language conditioned on the source (foreign) language $\mathbf{f}$, $P\left(\mathbf{e} \mid \mathbf{f}\right)$ (Philipp Koehn, Och, and Marcu, 2003). The task of translation is then finding

$$\mathbf{e}^* = \underset{\mathbf{e} \in E}{\arg\max}\, P\left(\mathbf{e} \mid \mathbf{f}\right)$$

The next two sections will briefly discuss the traditional approaches to language modelling and machine translation in statistical (empirical) natural language processing.

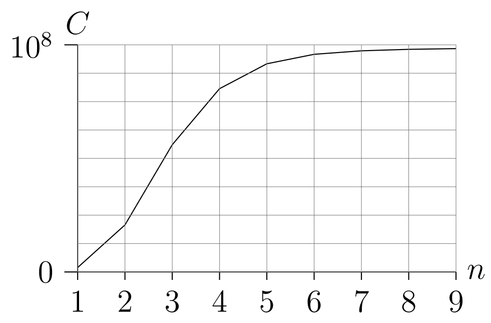


**Figure 2.1** – Number of distinct $n$-grams $C$ of a given order $n$ in the first $10^8$ words of the Common Crawl monolingual English corpus. For higher order $n$-grams we see that there as many as there are words in the corpus, which means that most higher order $n$-grams only appear once. Hence in practice language models do not employ $n$-grams above the order of ∼5.

## 2.1 $n$-gram language models

A naive approach is to estimate $P(e_i \mid e_1, \ldots, e_{i-1})$ by counting the number of times the sequence $(e_1, \ldots, e_i)$ appears in a given training text, relative to all other sequences $(e_1, \ldots, e_{i-1}, e'_i)$ that appear in the text

$$\hat{P}(e_i \mid e_1, \ldots, e_{i-1}) = \frac{C(e_1, \ldots, e_i)}{\sum_{e'_i \in E} C(e_1, \ldots, e_{i-1}, e'_i)}$$

where $C(\mathbf{e})$ is the number of times the sequence $\mathbf{e}$ appeared in a training set (often referred to as a *corpus*). However, as the length of the sequence grows we quickly run into problems of data sparsity (see figure 2.1). A sequence of more than ∼5 words is very unlikely to repeat itself, which is problematic. For example, if we didn't come across any other sentence containing the words "cat sat on the" our model now estimates:

$$\begin{aligned} \hat{P}(\text{mat} \mid \text{cat}, \text{sat}, \text{on}, \text{the}) &= 1 \\ \hat{P}(e_i \mid \text{cat}, \text{sat}, \text{on}, \text{the}) &= 0, \quad \forall e_i \in E \setminus \{\text{mat}\} \end{aligned}$$

Note that this is an extreme case of overfitting. $n$-gram models partially mitigate this by limiting the length of the sequence considered

$$\hat{P}(e_i \mid e_1, \ldots, e_{i-1}) \approx \hat{P}(e_i \mid e_{i-n+1}, \ldots, e_{i-1})$$

A variety of other methods are used to handle this sparsity. Most methods involve some combination of smoothing/discounting, interpolation and backing-off. Discounting simply means assigning a fixed non-zero count to unseen $n$-grams so that they are assigned a non-zero probability. Backing-off is a popular method that estimates the probability of unseen $n$-grams by using their constituent lower order $n$-grams (Katz, 1987). For example, given the two unseen 5-grams “dog sat on the mat” and “dog sat on the moon” we estimate

$$\hat{P}(\text{mat} \mid \text{dog}, \text{sat}, \text{on}, \text{the}) \approx \hat{P}(\text{mat} \mid \text{sat}, \text{on}, \text{the})$$
$$\hat{P}(\text{moon} \mid \text{dog}, \text{sat}, \text{on}, \text{the}) \approx \hat{P}(\text{moon} \mid \text{sat}, \text{on}, \text{the})$$

where most likely $\hat{P}(\text{mat} \mid \text{sat}, \text{on}, \text{the}) > \hat{P}(\text{moon} \mid \text{sat}, \text{on}, \text{the})$.

The most popular smoothing approach for $n$-gram models is modified Kneser-Ney smoothing (S. F. Chen and Goodman., 1996). It combines discounting with backing-off, but when performing back off it takes into account the word histories. To see why this is relevant consider the word “York”, which appears many times in most English corpora as part of the bigram “New York”. When the $n$-gram model needs to score the bigram “cat York” it is most likely forced to back-off to scoring the unigrams “cat” and “York”, both of which are common words and hence the bigram will be given a high likelihood. Modified Kneser-Ney scoring addresses this by penalizing lower order $n$-grams which only appear as part of a limited set of higher order $n$-grams. It was later shown that several $n$-grams models, such as the interpolated Kneser-Ney model, can be derived from a Bayesian perspective as an approximation to a Pitman-Yor process (a generalization of the Dirichlet process) (Goldwater, Johnson, and Griffiths, 2005; Y. W. Teh, 2006).

The performance of language models is often measured using *perplexity*, $2^{-\frac{1}{|\mathbf{e}|}\log_2 \hat{p}(\mathbf{e})}$. Note that this is a simple transformation of the negative log-likelihood, $-\log \hat{p}(\mathbf{e})$.

## 2.2 Neural language models

A shortcoming of $n$-gram models is that they do not take into account word similarity. If ‘the cat sat on the mat’ appears many times in a corpus, it does not increase the probability of ‘the dog sat on the mat’ because to the $n$-gram model the words ‘cat’ and ‘dog’ are as distinct as ‘cat’ and ‘summer’. Neural networks such as the one introduced in Y. Bengio, Ducharme, et al. (2003) can learn word embeddings i.e. they will learn a distinct set of parameters which can be understood to represent the semantics of each word. This allows neural networks to generalize better than count-based models.

Note that in Section 1.4 we mentioned that neural networks usually take vector-valued inputs with $\mathbf{x} \in \mathbb{R}^n$. We can map distinct words to $\mathbb{R}^n$ using a *one-hot encoding*. Given a dictionary of words $E = \{e_1, \ldots, e_n\}$ the one-hot mapping $f : E \to \mathbb{R}^n$ is defined as

$$f(e_i) = ([i = 1], \ldots, [i = n])$$

Note that the vector-matrix product $f(e_i)\mathbf{W}$ is equivalent to extracting the $i$th row from $\mathbf{W} \in \mathbb{R}^{n \times m}$. The row corresponding to a word is called its *embedding*. Note that a task like language modelling is often used as a proxy for learning these embeddings, which can then be used for a variety of other tasks.

A neural network model that takes several words as inputs can perform a level of *semantic composition*, deriving the meaning of a sentence or $n$-gram from the semantics of the constituent words (represented by the embeddings) and rules that combine them (the function learned by the network).

### 2.2.1 Feedforward language model

The neural net language model introduced in Y. Bengio, Ducharme, et al. (2003) models $P(e_i \mid e_{i-n+1}, \ldots, e_{i-1})$, similar to an $n$-gram model. More specifically, its parameters consist of an embedding matrix $\mathbf{W}^{(\text{emb})} \in \mathbb{R}^{|E| \times m}$, where $m$ is the dimension of our word embeddings, a weights matrix for the hidden layer, $\mathbf{W}^{(h)} \in \mathbb{R}^{nm \times h}$, and a weights matrix for the softmax, $\mathbf{W}^{(y)} \in \mathbb{R}^{h \times |E|}$, and corresponding biases. Note that the rows of the last matrix can also be considered word embeddings.

$$\begin{aligned}
\mathbf{p}_j &= \mathbf{e}_j \mathbf{W}^{(\text{emb})} && \text{for } j = i-n+1, \ldots, i-1 \\
\mathbf{p} &= [\mathbf{p}_{i-n+1} | \ldots | \mathbf{p}_{i-1}] \\
\mathbf{h} &= \tanh\left(\mathbf{W}^{(h)}\mathbf{p} + \mathbf{b}^{(h)}\right) \\
\hat{\mathbf{y}} &= \operatorname{softmax}\left(\mathbf{W}^{(y)}\mathbf{h} + \mathbf{b}^{(y)}\right)
\end{aligned}$$

This approach showed a significant improvement over traditional $n$-gram models.

For computational considerations most neural net language models do not learn embeddings for every single word, instead using a shortlist of the most frequent ones, $E_{\mathrm{SL}} \subset E$. All other words are referred to as out-of-vocabulary (OOV) words, and during training they are replaced by an unknown word token, The embedding of this word is then learned as usual.[i]

An advantage of feedforward language models compared to the RNN models discussed in the next section is that their computation can be parallelized i.e. $\hat{P}\left(e_i \mid e_{i-n+1}, \ldots, e_{i-1}\right)$ and $\hat{P}\left(e_{i+1} \mid e_{i-n+2}, \ldots, e_i\right)$ can be calculated simultaneously. This computational efficiency makes it possible to integrate them in the stack search used in statistical machine translation (Vaswani, Zhao, et al., 2013) (see Section 2.3.1). The log-likelihood given by the neural network language model to a partial translation is added as a new feature in the log-linear model.

### 2.2.2 Recurrent language model

For counting $n$-gram models the length of the history is limited in order to avoid data sparsity. Neural networks generalize better, which means that they can take into account a longer history. The parameter space of the previously discussed feedforward models scales linearly with the history size though, making it unfeasible to consider very large contexts. Recurrent neural networks (discussed in Section 1.4.5) on the other hand are an excellent option for language modelling, since the number of parameters is independent of the length of the input.

Originally vanilla RNNs were introduced as language models, showing large improvements over $n$-gram models (Mikolov, 2012; Mikolov, Karafiát, et al., 2010). Soon after LSTM versions were shown to lead to even more significant improvements (Sundermeyer, Schlüter, and Ney, 2012). Although many variations have

i. Note that this approach is common (out of necessity) but should be used with caution, because the performance of language models is only comparable on the exact same vocabulary. It is easy to see that a smaller vocabulary will inflate the accuracy model since by taking $\hat{P}\left(e_i \notin E_{\mathrm{SL}}\right) = \hat{P}\left(\mathrm{UNK}\right)$ we are actually introducing an upper bound since $\hat{P}\left(\mathrm{UNK}\right) = \sum_{e_i \notin E_{\mathrm{SL}}} \hat{P}(e_i) \geq \hat{P}(e_j)$ for all $e_j \notin E_{\mathrm{SL}}$. Consider the extreme case in which our dictionary has a size of 1. For the English language this is most likely the word 'the' which accounts for 7% of all word occurences. This means that the constant model $\hat{P}(\mathrm{UNK}) = 0.93$ achieves a phenomenal perplexity of 1.3, even though it has not actually learnt anything. The task of estimating $\hat{P}(\mathrm{UNK})$ also means that we are modelling whether a word's frequency is below the threshold required to be part of the dictionary, regardless of its semantics. This means we have unintentionally turned our problem into a multi-task one, which could hurt performance on the task we care about.

been explored since, the current state of the art is still a large 2-layer LSTM model (with thousands of hidden units) (Jozefowicz, Vinyals, et al., 2016).

## 2.3 Phrase-based statistical machine translation

The most succesful approach in statistical machine translation up to circa 2015 was phrase-based machine translation. Its underpinning theory begins by applying Bayes' theorem in order to split the problem of generating the translation **e** given the source sentence **f** into two subproblems

$$\arg\max_{\mathbf{e}\in E} P\left(\mathbf{e} \mid \mathbf{f}\right) = \arg\max_{\mathbf{e}\in E} P\left(\mathbf{f} \mid \mathbf{e}\right) P\left(\mathbf{e}\right)$$

The models for $P\left(\mathbf{e}\right)$ and $P\left(\mathbf{f} \mid \mathbf{e}\right)$ are called the language model and *translation model* respectively. The translation probability $P\left(\mathbf{f} \mid \mathbf{e}\right)$ is difficult to estimate. Hence the problem is traditionally approached by constructing a variety of feature functions (mostly heuristics) and combining them in a log-linear model

$$\arg\max_{\mathbf{e}\in E} \log P\left(\mathbf{f} \mid \mathbf{e}\right) \approx \arg\max_{\mathbf{e}\in E} \sum_{i=1}^{n} \lambda_i f_i\left(\mathbf{f}, \mathbf{e}\right)$$

where $f_i$ and $\lambda_i$ are the different feature functions and their weights in the log-linear model. The weights are then tuned to maximize the translation quality of the system.

Most features depend on a mapping between words in the source and target language. For example, a sentence where the German "Haus" is translated to the English "house" should score highly. However, not each word in the source sentence has a corresponding word in the target translation: The German "Hauskatze" corresponds to the English compound noun "domestic cat". Hence a phrase-based model (Philipp Koehn, Och, and Marcu, 2003) is used, where we try to find a mapping between groups of words in both languages instead (see figure 2.2). Note that sometimes words in one language don't have a translation in the other language, in which case we map it to an "empty phrase".

These alignments are learned from parallel corpora, also called bitexts, which consist of sentence-aligned texts in two languages. This is popularly done using the

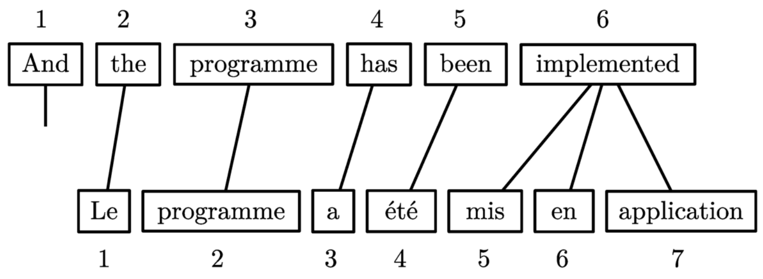


**Figure 2.2** – A phrase-mapping between French and English

| Source phrase | Target phrase | $\hat{P}(\mathbf{e} \mid \mathbf{f})$ | $\hat{P}(\mathbf{f} \mid \mathbf{e})$ |
|---|---|---|---|
| der | the | 0.3 | 0.2 |
| das | the | 0.4 | 0.1 |
| das | it | 0.2 | 0.3 |
| das ist | this is | 0.8 | 0.6 |
| das ist | this | 0.1 | 0.2 |

**Table 2.1** – A short example of the basic structure of a phrase table. The direct translation probability, $\hat{P}(\mathbf{e} \mid \mathbf{f})$, and inverse translation probability, $\hat{P}(\mathbf{f} \mid \mathbf{e})$, are calculated by counting the number of times the source and target phrases were aligned, divided by the number of times the phrases appeared in the source and target text respectively.

tool GIZA++[i] which combines IBM Models 1–5, statistical models that are trained using the expectation-minimization (EM) algorithm, and an hidden Markov model (HMM) in an attempt to find a word-level source-to-target alignment, as well as a word-level target-to-source alignment. These two alignments are then merged using certain heuristics (Och and Ney, 2003) to create a phrase-level alignment that will form the basis of our phrase table (see table 2.1 for an example).

### 2.3.1 Decoder

Given a language model and a phrase-based translation model, we can search for a suitable translation $\mathbf{e}$ of source sentence $\mathbf{f} = f_1, \ldots, f_n$ as follows: Consider all possible segmentations of the source sentence

$$(\mathbf{f}_1, \ldots, \mathbf{f}_m) = (f_1, \ldots, f_{i_1}, \ldots, f_{i_m}, \ldots, f_n)$$

i. https://code.google.com/p/giza-pp/

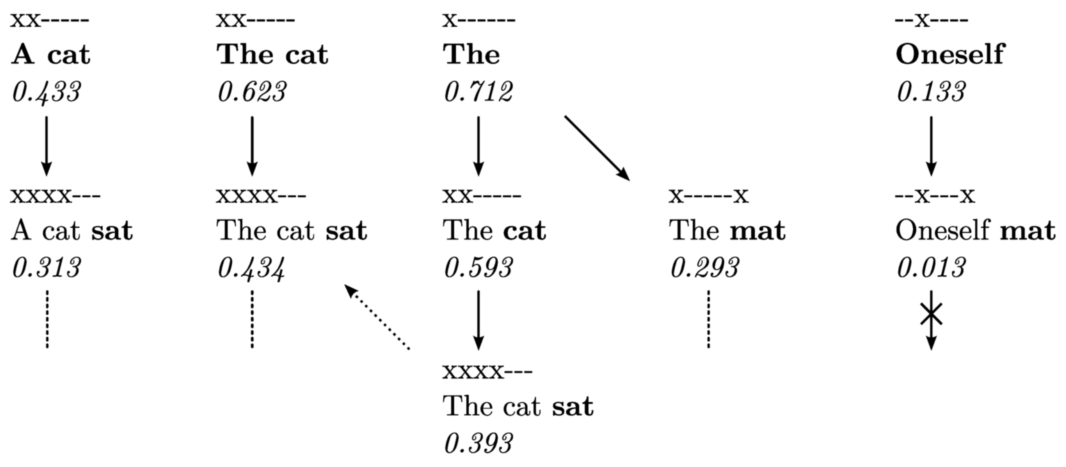


**Figure 2.3** – An illustration of a stack search as performed by a phrase-based decoder. The translation is developed iteratively by considering each untranslated source phrase which has an entry in the phrase table. In this diagram, the crosses on the first line of each node represent the words from the source sentence that have been translated. The second line is the translation so far. The third line is the score given by the log-linear model. Translations with low scores such as on the far right ("Oneself mat") are pruned. Nodes that result in the same translation ("The cat sat", bottom node) are recombined in order to reduce the search space further.

such that each segment $\mathbf{f}_i$ contains at least one entry in the phrase table (i.e. has at least one possible translation). For a given segmentation we can now consider all possible combinations of translations, in any order. An exhaustive search through this space is often not feasible, as it is of the order

$$\text{segmentations} \times \text{translations per phrase}^{\text{num segments}} \times \text{num segments!}$$

Instead, we perform a stack search, which is similar to a beam search, to limit our search space. During the stack search millions of partial translations are evaluated. See figure 2.3 for a small part of the stack search performed to translate the sentence "The cat sat on the mat" from Spanish:

$$\mathbf{f} = \text{El}, \text{gato}, \text{se}, \text{sentó}, \text{en}, \text{la}, \text{alfombra}$$

If implemented naively, the stack search algorithm is likely to translate all the 'easy' words in the sentence first (i.e. words for which translations with high scores can be found in the phrase table), pruning out all translations that begin with words that are difficult to translate. To counteract this bias the future cost of fully developing a translation must be estimated. This is done by assuming that the sentence will be translated without reordering. In this case finding the minimum cost translation path can be solved efficiently using dynamic programming.

### 2.3.2 Translation evaluation

In practice many of the feature functions used in the log-linear model do not output probabilities, instead outputting general numerical values such as the difference in length between the source and target sentence or the number of reorderings performed. Our estimation of $\hat{P}(\mathbf{f} \mid \mathbf{e})$ is no longer normalized, which means we cannot evaluate our translation system based on the likelihood, necessating the use of other evaluation metrics.

One of the most common metrics is the Bilingual Evaluation Understudy (BLEU) score (Papineni et al., 2002). Theoretically, the BLEU score can use multiple reference texts to reduce any bias caused by a single human translator. However, in practice the cost of obtaining multiple high-quality human-translated texts is often too high. A machine-translated text is a sequence of sentences $(\mathbf{e}_1, \ldots, \mathbf{e}_m)$, where each sentence $\mathbf{e}_i$ is a sequence of words $\mathbf{e}_i = e_i^1, \ldots, e_i^{k_i}$. Define the set of $n$-grams in sentence $\mathbf{e}_i$ as

$$w_i^n = \left\{ \left(e_i^1, \ldots, e_i^n\right), \ldots, \left(e_i^{k_i - n + 1}, \ldots, e_i^{k_i}\right) \right\}$$

So $w_i^1$ is the set of all unigrams (words) in sentence $\mathbf{e}_i$, $w_i^2$ the set of all bigrams, etc. Given a human-translated reference text $(\mathbf{e}'_1, \ldots, \mathbf{e}'_m)$ define the set of $n$-grams in each sentence $\mathbf{e}'_i$ likewise

$$w_{i'}^n = \left\{ \left(e_{i'}^1, \ldots, e_{i'}^n\right), \ldots, \left(e_{i'}^{k'_i - n + 1}, \ldots, e_{i'}^{k'_i}\right) \right\}$$

The BLEU score of a text $(\mathbf{e}_1, \ldots, \mathbf{e}_m)$ considers the fraction of $n$-grams in the machine translation that also appear in the reference translation for $n = 1, \ldots, 4$ and takes the geometric mean of these values.

$$\mathrm{BLEU} = \mathrm{BP} \cdot \sqrt[\frac{1}{4}]{\prod_{n=1}^{4} \left( \frac{\sum_{i=1}^{m} \left| w_i^n \cap w_i^{n\prime} \right|}{\sum_{i=1}^{m} \left| w_i^n \right|} \right)}$$

BP is the *brevity penalty*, which is only applied if the total length of the reference translation is longer than the machine translation, $\sum_{i=1}^{m} m'_i > \sum_{i=1}^{m} m_i$, in which case it is defined to be

$$\text{BP} = \exp\left(1 - \frac{\sum_{i=1}^{m} m'_i}{\sum_{i=1}^{m} m_i}\right)$$

Without the brevity penalty a translation model could inflate its BLEU score by producing short sentences with high precision but low recall i.e. produce short sentences with only the words that it is sure will match. A BLEU score is strictly speaking a value between 0 and 1, but it is often reported on a scale of 0 to 100 instead. Note that the BLEU score is often 0 when calculated on a sentence level, because correct 4-grams are uncommon.

# 3 Prologue to First Article

## 3.1 Article Details

**Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation**.

Cho, KyungHyun, Bart van Merriënboer, Çağlar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, Doha, Qatar, October 25–29: 1724–1734.

*Personal Contribution.*

As second author my primary contributions to this paper were in the application area of statistical machine translation (SMT) whereas the primary author, KyungHun Cho, must be credited for the development of the encoder-decoder model and the gated recurrent unit (GRU). I researched and conceptualized ways of integrating generative language models into the SMT pipeline, which resulted in the idea of scoring phrase pairs in the phrase table (the central application of the model in this paper). I performed the majority of the experiments that involved data preprocessing and the SMT system, Moses.

## 3.2 Context

Following the introduction of neural network language models in Y. Bengio, Ducharme, et al. (2003) and recurrent neural network languages in Mikolov, Karafiát, et al. (2010), neural networks were applied to a variety of other tasks in NLP such as speech recognition, paraphrase detection, and reasoning. Machine translation is an important goal in NLP, and neural network language models were quickly adopted to score the quality of translations, either in isolation (Schwenk, Costa-Jussà, and

Fonollosa, 2006) or with the source sentence as additional context (Schwenk, 2012; H.-S. Le, Allauzen, and Yvon, 2012; Zou et al., 2013). A convolutional encoder-decoder model was used as a translation model in Kalchbrenner and Blunsom (2013), and was evaluated by rescoring $n$-best translation lists.

The goal of this research was to evaluate neural network models which were not only able to rescore translations proposed by a phrase-based system, but to generate different translations by being more tightly integrated into the stack search.

## 3.3 Contributions

The contributions of this work are threefold.

Firstly, it introduced the combination of two recurrent neural networks in order to model $p(x|y)$ where both $x$ and $y$ are variable-length sequences. We refer to this model as the RNN encoder-decoder. The use of neural networks to model variable length inputs and outputs is presently more commonly known as 'sequence-to-sequence learning'.

The second contribution of this paper is a new RNN cell, which we refer to as the *gated recurrent unit* (GRU) in later work. Similarly to the well-known LSTM unit, the GRU unit helps learning long-term dependencies by sidestepping the exploding and vanishing gradient problem. However, the GRU is computationally simpler and hence scales more easily.

Lastly, it introduced the scoring of a phrase table as an application of this new model and RNN cell. This deeper integration into the translation pipeline allowed us to achieve performance improvements in machine translation beyond what is possible by rescoring the $n$-best lists.

## 3.4 Recent Developments

This work was one in a series of closely related papers, starting with Kalchbrenner and Blunsom (2013), which founded the field of neural machine translation (NMT). In work that was performed in parallel to ours (Sutskever, Vinyals, and Q.

Le, 2014) it was shown that RNN encoder-decoder models can reach competitive translation performance compared with phrase-based methods, given large enough model sizes and the use of beam search.

The year following the publication of this work the first NMT system competed in the EMNLP 2015 Workshop on Machine Translation (WMT) translation task. By 2016 over 90% of the top translations systems in WMT16 were NMT systems. The improvement in translation quality over legacy systems saw NMT systems adopted quickly by industry, and Google's Neural Machine Translation (GNMT) system was launched before the end of 2016.

A large body of follow up work on NMT has since been produced. Directions of research include character-level translation instead of word-level translation (Chung, Cho, and Y. Bengio, 2016; Ling et al., 2015; Lee, Cho, and Hofmann, 2017), training the model and the sampling technique (beam search) jointly (Wiseman and Rush, 2016), curriculum learning (S. Bengio et al., 2015), adressing the quadratic memory complexity of attention models (Luong, H. Pham, and Christopher D Manning, 2015), alternatives to RNN-based encoder-decoder models such as convolutional based models (Kalchbrenner, Espeholt, et al., 2016; Gehring et al., 2017) and feed-forward models (Vaswani, Shazeer, et al., 2017), the use of monolingual data (Gulcehre et al., 2015), multilingual translation (Firat, Cho, and Y. Bengio, 2016), and the rare word problem (Luong, Sutskever, et al., 2014; Sennrich, Haddow, and Birch, 2015).

# 4 Learning Phrase Representations using RNN Encoder-Decoder for SMT

## 4.1 Introduction

Deep neural networks have shown great success in various applications such as objection recognition (see, e.g., Krizhevsky, Sutskever, and Geoffrey E Hinton (2012)) and speech recognition (see, e.g., Dahl et al. (2012)). Furthermore, many recent works showed that neural networks can be successfully used in a number of tasks in natural language processing (NLP). These include, but are not limited to, language modeling (Y. Bengio, Ducharme, et al., 2003), paraphrase detection (Socher et al., 2011) and word embedding extraction (Mikolov, Sutskever, et al., 2013). In the field of statistical machine translation (SMT), deep neural networks have begun to show promising results. Schwenk (2012) summarizes a successful usage of feedforward neural networks in the framework of phrase-based SMT system.

Along this line of research on using neural networks for SMT, this paper focuses on a novel neural network architecture that can be used as a part of the conventional phrase-based SMT system. The proposed neural network architecture, which we will refer to as an *RNN Encoder-Decoder*, consists of two recurrent neural networks (RNN) that act as an encoder and a decoder pair. The encoder maps a variable-length source sequence to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence. The two networks are trained jointly to maximize the conditional probability of the target sequence given a source sequence. Additionally, we propose to use a rather sophisticated hidden unit in order to improve both the memory capacity and the ease of training.

The proposed RNN Encoder-Decoder with a novel hidden unit is empirically evaluated on the task of translating from English to French. We train the model to learn the translation probability of an English phrase to a corresponding French phrase. The model is then used as a part of a standard phrase-based SMT system by scoring each phrase pair in the phrase table. The empirical evaluation reveals that this approach of scoring phrase pairs with an RNN Encoder-Decoder improves the translation performance.

We qualitatively analyze the trained RNN Encoder-Decoder by comparing its phrase scores with those given by the existing translation model. The qualitative analysis shows that the RNN Encoder-Decoder is better at capturing the linguistic regularities in the phrase table, indirectly explaining the quantitative improvements in the overall translation performance. The further analysis of the model reveals that the RNN Encoder-Decoder learns a continuous space representation of a phrase that preserves both the semantic and syntactic structure of the phrase.

## 4.2 RNN Encoder-Decoder

### 4.2.1 Preliminary: Recurrent Neural Networks

A recurrent neural network (RNN) is a neural network that consists of a hidden state $\mathbf{h}$ and an optional output $\mathbf{y}$ which operates on a variable-length sequence $\mathbf{x} = (x_1, \ldots, x_T)$. At each time step $t$, the hidden state $\mathbf{h}_{\langle t \rangle}$ of the RNN is updated by

$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, x_t\right), \tag{4.1}$$

where $f$ is a non-linear activation function. $f$ may be as simple as an element-wise logistic sigmoid function and as complex as a long short-term memory (LSTM) unit (Hochreiter and Jürgen Schmidhuber, 1997).

An RNN can learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence. In that case, the output at each timestep $t$ is the conditional distribution $p(x_t \mid x_{t-1}, \ldots, x_1)$. For example, a multinomial distribution (1-of-$K$ coding) can be output using a softmax activation function

$$p(x_{t,j} = 1 \mid x_{t-1}, \ldots, x_1) = \frac{\exp\left(\mathbf{w}_j \mathbf{h}_{\langle t \rangle}\right)}{\sum_{j'=1}^{K} \exp\left(\mathbf{w}_{j'} \mathbf{h}_{\langle t \rangle}\right)}, \tag{4.2}$$

for all possible symbols $j = 1, \ldots, K$, where $\mathbf{w}_j$ are the rows of a weight matrix $\mathbf{W}$. By combining these probabilities, we can compute the probability of the sequence $\mathbf{x}$ using

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \ldots, x_1). \tag{4.3}$$

From this learned distribution, it is straightforward to sample a new sequence by iteratively sampling a symbol at each time step.

### 4.2.2 RNN Encoder-Decoder

In this paper, we propose a novel neural network architecture that learns to *encode* a variable-length sequence into a fixed-length vector representation and to *decode* a given fixed-length vector representation back into a variable-length sequence. From a probabilistic perspective, this new model is a general method to learn the conditional distribution over a variable-length sequence conditioned on yet another variable-length sequence, e.g. $p(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T)$, where one should note that the input and output sequence lengths $T$ and $T'$ may differ.

The encoder is an RNN that reads each symbol of an input sequence $\mathbf{x}$ sequentially. As it reads each symbol, the hidden state of the RNN changes according to Eq. (4.1). After reading the end of the sequence (marked by an end-of-sequence symbol), the hidden state of the RNN is a summary $\mathbf{c}$ of the whole input sequence.

The decoder of the proposed model is another RNN which is trained to *generate* the output sequence by predicting the next symbol $y_t$ given the hidden state $\mathbf{h}_{\langle t \rangle}$. However, unlike the RNN described in Sec. 4.2.1, both $y_t$ and $\mathbf{h}_{\langle t \rangle}$ are also conditioned on $y_{t-1}$ and on the summary $\mathbf{c}$ of the input sequence. Hence, the hidden state of the decoder at time $t$ is computed by,

$$\mathbf{h}_{\langle t \rangle} = f\left(\mathbf{h}_{\langle t-1 \rangle}, y_{t-1}, \mathbf{c}\right),$$

and similarly, the conditional distribution of the next symbol is

$$P(y_t | y_{t-1}, y_{t-2}, \ldots, y_1, \mathbf{c}) = g\left(\mathbf{h}_{\langle t \rangle}, y_{t-1}, \mathbf{c}\right).$$

for given activation functions $f$ and $g$ (the latter must produce valid probabilities,

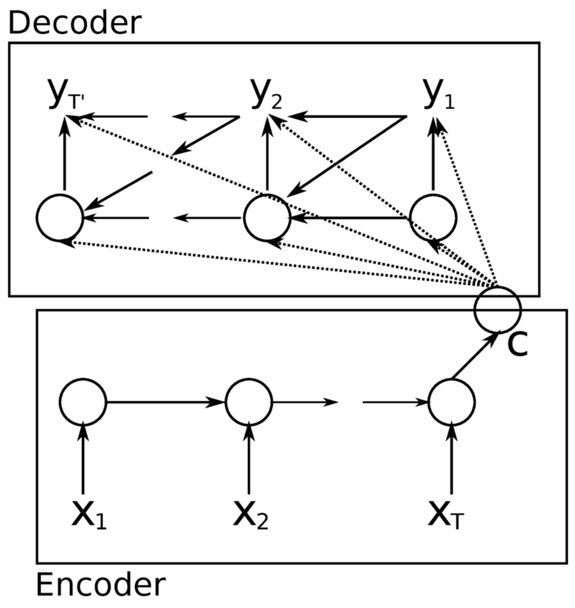


**Figure 4.1** – An illustration of the proposed RNN Encoder-Decoder.

e.g. with a softmax).

See Fig. 4.1 for a graphical depiction of the proposed model architecture.

The two components of the proposed *RNN Encoder-Decoder* are jointly trained to maximize the conditional log-likelihood

$$\max_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{y}_n \mid \mathbf{x}_n), \tag{4.4}$$

where $\boldsymbol{\theta}$ is the set of the model parameters and each $(\mathbf{x}_n, \mathbf{y}_n)$ is an (input sequence, output sequence) pair from the training set. In our case, as the output of the decoder, starting from the input, is differentiable, we can use a gradient-based algorithm to estimate the model parameters.

Once the RNN Encoder-Decoder is trained, the model can be used in two ways. One way is to use the model to generate a target sequence given an input sequence. On the other hand, the model can be used to *score* a given pair of input and output sequences, where the score is simply a probability $p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x})$ from Eqs. (4.3) and (4.4).

### 4.2.3 Hidden Unit that Adaptively Remembers and Forgets

In addition to a novel model architecture, we also propose a new type of hidden unit ($f$ in Eq. (4.1)) that has been motivated by the LSTM unit but is much simpler to compute and implement.[i] Fig. 4.2 shows the graphical depiction of the proposed hidden unit.

Let us describe how the activation of the $j$-th hidden unit is computed. First, the *reset* gate $r_j$ is computed by

$$r_j = \sigma\left(\left[\mathbf{W}_r \mathbf{x}\right]_j + \left[\mathbf{U}_r \mathbf{h}_{\langle t-1\rangle}\right]_j\right), \tag{4.5}$$

where $\sigma$ is the logistic sigmoid function, and $[.]_j$ denotes the $j$-th element of a vector. $\mathbf{x}$ and $\mathbf{h}_{t-1}$ are the input and the previous hidden state, respectively. $\mathbf{W}_r$ and $\mathbf{U}_r$ are weight matrices which are learned.

Similarly, the *update* gate $z_j$ is computed by

$$z_j = \sigma\left(\left[\mathbf{W}_z \mathbf{x}\right]_j + \left[\mathbf{U}_z \mathbf{h}_{\langle t-1\rangle}\right]_j\right). \tag{4.6}$$

The actual activation of the proposed unit $h_j$ is then computed by

$$h_j^{\langle t\rangle} = z_j h_j^{\langle t-1\rangle} + (1 - z_j)\tilde{h}_j^{\langle t\rangle}, \tag{4.7}$$

where

$$\tilde{h}_j^{\langle t\rangle} = \phi\left(\left[\mathbf{W}\mathbf{x}\right]_j + \left[\mathbf{U}\left(\mathbf{r} \odot \mathbf{h}_{\langle t-1\rangle}\right)\right]_j\right). \tag{4.8}$$

In this formulation, when the reset gate is close to 0, the hidden state is forced to ignore the previous hidden state and reset with the current input only. This effectively allows the hidden state to *drop* any information that is found to be irrelevant later in the future, thus, allowing a more compact representation.

On the other hand, the update gate controls how much information from the previous hidden state will carry over to the current hidden state. This acts similarly

i. The LSTM unit, which has shown impressive results in several applications such as speech recognition, has a memory cell and four gating units that adaptively control the information flow inside the unit, compared to only two gating units in the proposed hidden unit. For details on LSTM networks, see, e.g., Alex Graves (2012b).

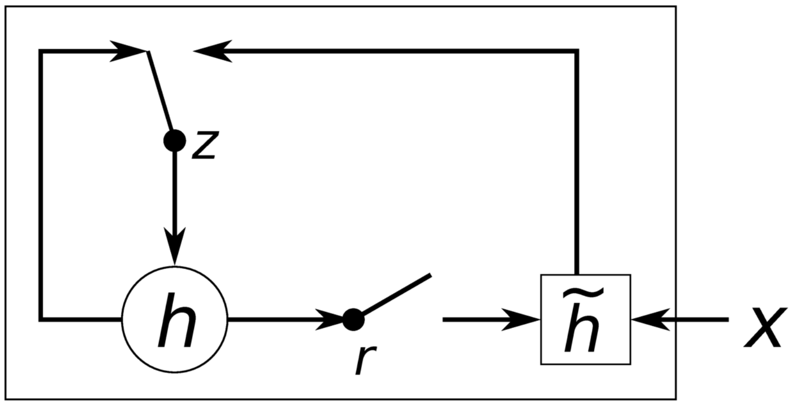


**Figure 4.2** – An illustration of the proposed hidden activation function. The update gate $z$ selects whether the hidden state is to be updated with a new hidden state $\tilde{h}$. The reset gate $r$ decides whether the previous hidden state is ignored. See Eqs. (4.5)–(4.8) for the detailed equations of $r$, $z$, $h$ and $\tilde{h}$.

to the memory cell in the LSTM network and helps the RNN to remember long-term information. Furthermore, this may be considered an adaptive variant of a leaky-integration unit (Y. Bengio, Boulanger-Lewandowski, and Razvan Pascanu, 2013).

As each hidden unit has separate reset and update gates, each hidden unit will learn to capture dependencies over different time scales. Those units that learn to capture short-term dependencies will tend to have reset gates that are frequently active, but those that capture longer-term dependencies will have update gates that are mostly active.

In our preliminary experiments, we found that it is crucial to use this new unit with gating units. We were not able to get meaningful result with an oft-used tanh unit without any gating.

## 4.3 Statistical Machine Translation

In a commonly used statistical machine translation system (SMT), the goal of the system (decoder, specifically) is to find a translation **f** given a source sentence **e**, which maximizes

$$p(\mathbf{f} \mid \mathbf{e}) \propto p(\mathbf{e} \mid \mathbf{f}) p(\mathbf{f}),$$

where the first term at the right hand side is called *translation model* and the latter *language model* (see, e.g., P. Koehn (2005)). In practice, however, most SMT systems model $\log p(\mathbf{f} \mid \mathbf{e})$ as a log-linear model with additional features and corresponding weights:

$$\log p(\mathbf{f} \mid \mathbf{e}) = \sum_{n=1}^{N} w_n f_n(\mathbf{f}, \mathbf{e}) + \log Z(\mathbf{e}), \tag{4.9}$$

where $f_n$ and $w_n$ are the $n$-th feature and weight, respectively. $Z(\mathbf{e})$ is a normalization constant that does not depend on the weights. The weights are often optimized to maximize the BLEU score on a development set.

In the phrase-based SMT framework introduced in Philipp Koehn, Och, and Marcu (2003) and Marcu and Wong (2002), the translation model $\log p(\mathbf{e} \mid \mathbf{f})$ is factorized into the translation probabilities of matching phrases in the source and target sentences.[i] These probabilities are once again considered additional features in the log-linear model (see Eq. (4.9)) and are weighted accordingly to maximize the BLEU score.

Since the neural net language model was proposed in Y. Bengio, Ducharme, et al. (2003), neural networks have been used widely in SMT systems. In many cases, neural networks have been used to *rescore* translation hypotheses ($n$-best lists) (see, e.g., Schwenk, Costa-Jussà, and Fonollosa (2006)). Recently, however, there has been interest in training neural networks to score the translated sentence (or phrase pairs) using a representation of the source sentence as an additional input. See, e.g., Schwenk (2012), H.-S. Le, Allauzen, and Yvon (2012) and Zou et al. (2013).

### 4.3.1 Scoring Phrase Pairs with RNN Encoder-Decoder

Here we propose to train the RNN Encoder-Decoder (see Sec. 4.2.2) on a table of phrase pairs and use its scores as additional features in the log-linear model in Eq. (4.9) when tuning the SMT decoder.

When we train the RNN Encoder-Decoder, we ignore the (normalized) frequencies of each phrase pair in the original corpora. This measure was taken in order

i. Without loss of generality, from here on, we refer to $p(\mathbf{e} \mid \mathbf{f})$ for each phrase pair as a translation model as well.

(1) to reduce the computational expense of randomly selecting phrase pairs from a large phrase table according to the normalized frequencies and (2) to ensure that the RNN Encoder-Decoder does not simply learn to rank the phrase pairs according to their numbers of occurrences. One underlying reason for this choice was that the existing translation probability in the phrase table already reflects the frequencies of the phrase pairs in the original corpus. With a fixed capacity of the RNN Encoder-Decoder, we try to ensure that most of the capacity of the model is focused toward learning linguistic regularities, i.e., distinguishing between plausible and implausible translations, or learning the "manifold" (region of probability concentration) of plausible translations.

Once the RNN Encoder-Decoder is trained, we add a new score for each phrase pair to the existing phrase table. This allows the new scores to enter into the existing tuning algorithm with minimal additional overhead in computation.

As Schwenk points out (2012), it is possible to completely replace the existing phrase table with the proposed RNN Encoder-Decoder. In that case, for a given source phrase, the RNN Encoder-Decoder will need to generate a list of (good) target phrases. This requires, however, an expensive sampling procedure to be performed repeatedly. In this paper, thus, we only consider rescoring the phrase pairs in the phrase table.

### 4.3.2 Related Approaches: Neural Networks in Machine Translation

Before presenting the empirical results, we discuss a number of recent works that have proposed to use neural networks in the context of SMT.

Schwenk (2012) proposed a similar approach of scoring phrase pairs. Instead of the RNN-based neural network, he used a feedforward neural network that has fixed-size inputs (7 words in his case, with zero-padding for shorter phrases) and fixed-size outputs (7 words in the target language). When it is used specifically for scoring phrases for the SMT system, the maximum phrase length is often chosen to be small. However, as the length of phrases increases or as we apply neural networks to other variable-length sequence data, it is important that the neural network can handle variable-length input and output. The proposed RNN Encoder-Decoder is well-suited for these applications.

Similar to Schwenk (2012), Devlin et al. (2014) proposed to use a feedforward neural network to model a translation model, however, by predicting one word in a target phrase at a time. They reported an impressive improvement, but their approach still requires the maximum length of the input phrase (or context words) to be fixed a priori.

Although it is not exactly a neural network they train, the authors of Zou et al. (2013) proposed to learn a bilingual embedding of words/phrases. They use the learned embedding to compute the distance between a pair of phrases which is used as an additional score of the phrase pair in an SMT system.

In Chandar A P et al. (2014), a feedforward neural network was trained to learn a mapping from a bag-of-words representation of an input phrase to an output phrase. This is closely related to both the proposed RNN Encoder-Decoder and the model proposed in Schwenk (2012), except that their input representation of a phrase is a bag-of-words. A similar approach of using bag-of-words representations was proposed in J. Gao et al. (2013) as well. Earlier, a similar encoder-decoder model using two recursive neural networks was proposed in Socher et al. (2011), but their model was restricted to a monolingual setting, i.e. the model reconstructs an input sentence. More recently, another encoder-decoder model using an RNN was proposed in Auli et al. (2013), where the decoder is conditioned on a representation of either a source sentence or a source context.

One important difference between the proposed RNN Encoder-Decoder and the approaches in Zou et al. (2013) and Chandar A P et al. (2014) is that the order of the words in source and target phrases is taken into account. The RNN Encoder-Decoder naturally distinguishes between sequences that have the same words but in a different order, whereas the aforementioned approaches effectively ignore order information.

The closest approach related to the proposed RNN Encoder-Decoder is the Recurrent Continuous Translation Model (Model 2) proposed in Kalchbrenner and Blunsom (2013). In their paper, they proposed a similar model that consists of an encoder and decoder. The difference with our model is that they used a convolutional $n$-gram model (CGM) for the encoder and the hybrid of an inverse CGM and a recurrent neural network for the decoder. They, however, evaluated their model on rescoring the $n$-best list proposed by the conventional SMT system and computing the perplexity of the gold standard translations.

## 4.4 Experiments

We evaluate our approach on the English/French translation task of the WMT'14 workshop.

### 4.4.1 Data and Baseline System

Large amounts of resources are available to build an English/French SMT system in the framework of the WMT'14 translation task. The bilingual corpora include Europarl (61M words), news commentary (5.5M), UN (421M), and two crawled corpora of 90M and 780M words respectively. The last two corpora are quite noisy. To train the French language model, about 712M words of crawled newspaper material is available in addition to the target side of the bitexts. All the word counts refer to French words after tokenization.

It is commonly acknowledged that training statistical models on the concatenation of all this data does not necessarily lead to optimal performance, and results in extremely large models which are difficult to handle. Instead, one should focus on the most relevant subset of the data for a given task. We have done so by applying the data selection method proposed in Moore and Lewis (2010), and its extension to bitexts (Axelrod, He, and J. Gao, 2011). By these means we selected a subset of 418M words out of more than 2G words for language modeling and a subset of 348M out of 850M words for training the RNN Encoder-Decoder. We used the test set `newstest2012 and 2013` for data selection and weight tuning with MERT, and `newstest2014` as our test set. Each set has more than 70 thousand words and a single reference translation.

For training the neural networks, including the proposed RNN Encoder-Decoder, we limited the source and target vocabulary to the most frequent 15,000 words for both English and French. This covers approximately 93% of the dataset. All the out-of-vocabulary words were mapped to a special token ([UNK]).

The baseline phrase-based SMT system was built using Moses with default settings. This system achieves a BLEU score of 30.64 and 33.3 on the development and test sets, respectively (see Table 4.1).

| Models | BLEU | |
|---|---|---|
| | dev | test |
| Baseline | 30.64 | 33.30 |
| RNN | 31.20 | 33.87 |
| CSLM + RNN | 31.48 | 34.64 |
| CSLM + RNN + WP | 31.50 | 34.54 |

**Table 4.1** – BLEU scores computed on the development and test sets using different combinations of approaches. WP denotes a *word penalty*, where we penalizes the number of unknown words to neural networks.

### RNN Encoder-Decoder

The RNN Encoder-Decoder used in the experiment had 1000 hidden units with the proposed gates at the encoder and at the decoder. The input matrix between each input symbol $x_{\langle t \rangle}$ and the hidden unit is approximated with two lower-rank matrices, and the output matrix is approximated similarly. We used rank-100 matrices, equivalent to learning an embedding of dimension 100 for each word. The activation function used for $\tilde{h}$ in Eq. (4.8) is a hyperbolic tangent function. The computation from the hidden state in the decoder to the output is implemented as a deep neural network (Razan Pascanu et al., 2014) with a single intermediate layer having 500 maxout units each pooling 2 inputs (Goodfellow, Warde-Farley, et al., 2013).

All the weight parameters in the RNN Encoder-Decoder were initialized by sampling from an isotropic zero-mean (white) Gaussian distribution with its standard deviation fixed to 0.01, except for the recurrent weight parameters. For the recurrent weight matrices, we first sampled from a white Gaussian distribution and used its left singular vectors matrix, following Saxe, James L. McClelland, and Ganguli (2014).

We used Adadelta and stochastic gradient descent to train the RNN Encoder-Decoder with hyperparameters $\epsilon = 10^{-6}$ and $\rho = 0.95$ (Zeiler, 2012). At each update, we used 64 randomly selected phrase pairs from a phrase table (which was created from 348M words). The model was trained for approximately three days.

Details of the architecture used in the experiments are explained in more depth in the supplementary material.

| Source | Translation Model | RNN Encoder-Decoder |
|---|---|---|
| at the end of the | a la fin de la<br>ŕ la fin des années<br>être supprimés à la fin de la | à la fin du<br>à la fin des<br>à la fin de la |
| for the first time | **r** Ⓒ pour la premi**r**ëre fois<br>été donnés pour la première fois<br>été commémorée pour la première fois | pour la première fois<br>pour la première fois ,<br>pour la première fois que |
| in the United States and | **?** aux **?**tats-Unis et<br>été ouvertes aux États-Unis et<br>été constatées aux États-Unis et | aux Etats-Unis et<br>des Etats-Unis et<br>des États-Unis et |
| , as well as | **?**s , qu'<br>**?**s , ainsi que<br>**?**re aussi bien que | , ainsi qu'<br>, ainsi que<br>, ainsi que les |
| one of the most | **?**t **?**l' un des plus<br>**?**l' un des plus<br>être retenue comme un de ses plus | l' un des<br>le<br>un des |

**(a)** Long, frequent source phrases

## Neural Language Model

In order to assess the effectiveness of scoring phrase pairs with the proposed RNN Encoder-Decoder, we also tried a more traditional approach of using a neural network for learning a target language model (CSLM) (Schwenk, 2007). Especially, the comparison between the SMT system using CSLM and that using the proposed approach of phrase scoring by RNN Encoder-Decoder will clarify whether the contributions from multiple neural networks in different parts of the SMT system add up or are redundant.

We trained the CSLM model on 7-grams from the target corpus. Each input word was projected into the embedding space $\mathbb{R}^{512}$, and they were concatenated to form a 3072-dimensional vector. The concatenated vector was fed through two rectified layers (of size 1536 and 1024) (Glorot, Bordes, and Y. Bengio, 2011–2014). The output layer was a simple softmax layer (see Eq. (4.2)). All the weight parameters were initialized uniformly between $-0.01$ and $0.01$, and the model was trained until the validation perplexity did not improve for 10 epochs. After training,

| Source | Translation Model | RNN Encoder-Decoder |
|---|---|---|
| , Minister of Communications and Transport | Secrétaire aux communications et aux transports :<br>Secrétaire aux communications et aux transports | Secrétaire aux communications et aux transports<br>Secrétaire aux communications et aux transports : |
| did not comply with the | vestimentaire , ne correspondaient pas à des<br>susmentionnée n' était pas conforme aux<br>présentées n' étaient pas conformes à la | n' ont pas respecté les<br>n' était pas conforme aux<br>n' ont pas respecté la |
| parts of the world . | ⓒ gions du monde .<br>régions du monde considérées .<br>région du monde considérée . | parties du monde .<br>les parties du monde .<br>des parties du monde . |
| the past few days . | le petit texte .<br>cours des tout derniers jours .<br>les tout derniers jours . | ces derniers jours .<br>les derniers jours .<br>cours des derniers jours . |
| on Friday and Saturday | vendredi et samedi à la<br>vendredi et samedi à<br>se déroulera vendredi et samedi , | le vendredi et le samedi<br>le vendredi et samedi<br>vendredi et samedi |

**(b)** Long, rare source phrases

**Table 4.2** – The top scoring target phrases for a small set of source phrases according to the translation model (direct translation probability) and by the RNN Encoder-Decoder. Source phrases were randomly selected from phrases with 4 or more words. **?** denotes an incomplete (partial) character. **r** is a Cyrillic letter *ghe*.

the language model achieved a perplexity of 45.80. The validation set was a random selection of 0.1% of the corpus. The model was used to score partial translations during the decoding process, which generally leads to higher gains in BLEU score than n-best list rescoring (Vaswani, Zhao, et al., 2013).

To address the computational complexity of using a CSLM in the decoder a buffer was used to aggregate n-grams during the stack-search performed by the decoder. Only when the buffer is full, or a stack is about to be pruned, the n-grams are scored by the CSLM. This allows us to perform fast matrix-matrix multiplication on GPU using Theano (Bergstra et al., 2010; Frédéric Bastien et al., 2012).

| Source | Samples from RNN Encoder-Decoder |
|---|---|
| at the end of the | à la fin de la (×11) |
| for the first time | pour la première fois (×24)<br>pour la première fois que (×2) |
| in the United States and | aux États-Unis et (×6)<br>dans les États-Unis et (×4) |
| , as well as | , ainsi que<br>,<br>ainsi que<br>, ainsi qu'<br>et UNK |
| one of the most | l' un des plus (×9)<br>l' un des (×5)<br>l' une des plus (×2) |

**(a)** Long, frequent source phrases

### 4.4.2 Quantitative Analysis

We tried the following combinations:

1. Baseline configuration
2. Baseline + RNN
3. Baseline + CSLM + RNN
4. Baseline + CSLM + RNN + Word penalty

The results are presented in Table 4.1. As expected, adding features computed by neural networks consistently improves the performance over the baseline performance.

The best performance was achieved when we used both CSLM and the phrase scores from the RNN Encoder-Decoder. This suggests that the contributions of the CSLM and the RNN Encoder-Decoder are not too correlated and that one can expect better results by improving each method independently. Furthermore, we tried penalizing the number of words that are unknown to the neural networks (i.e. words which are not in the shortlist). We do so by simply adding the number of unknown words as an additional feature the log-linear model in Eq. (4.9).[i] However,

i. To understand the effect of the penalty, consider the set of all words in the 15,000 large

| Source | Samples from RNN Encoder-Decoder |
|---|---|
| , Minister of Communications and Transport | , ministre des communications et le transport (×13) |
| did not comply with the | n' tait pas conforme aux<br>n' a pas respect l' (×2)<br>n' a pas respect la (×3) |
| parts of the world . | arts du monde . (×11)<br>des arts du monde . (×7) |
| the past few days . | quelques jours . (×5)<br>les derniers jours . (×5)<br>ces derniers jours . (×2) |
| on Friday and Saturday | vendredi et samedi (×5)<br>le vendredi et samedi (×7)<br>le vendredi et le samedi (×4) |

**(b)** Long, rare source phrases

**Table 4.3** – Samples generated from the RNN Encoder-Decoder for each source phrase used in Table 4.2. We show the top-5 target phrases out of 50 samples. They are sorted by the RNN Encoder-Decoder scores.

in this case we were not able to achieve better performance on the test set, but only on the development set.

### 4.4.3 Qualitative Analysis

In order to understand where the performance improvement comes from, we analyze the phrase pair scores computed by the RNN Encoder-Decoder against the

shortlist, SL. All words $x^i \notin$ SL are replaced by a special token [UNK] before being scored by the neural networks. Hence, the conditional probability of any $x_t^i \notin$ SL is actually given by the model as

$$\begin{aligned} p\left(x_t = \text{[UNK]} \mid x_{<t}\right) &= p\left(x_t \notin \text{SL} \mid x_{<t}\right) \\ &= \sum_{x_t^j \notin SL} p\left(x_t^j \mid x_{<t}\right) \geq p\left(x_t^i \mid x_{<t}\right), \end{aligned}$$

where $x_{<t}$ is a shorthand notation for $x_{t-1}, \ldots, x_1$.

As a result, the probability of words not in the shortlist is always overestimated. It is possible to address this issue by backing off to an existing model that contain non-shortlisted words (see Schwenk (2007)) In this paper, however, we opt for introducing a word penalty instead, which counteracts the word probability overestimation.

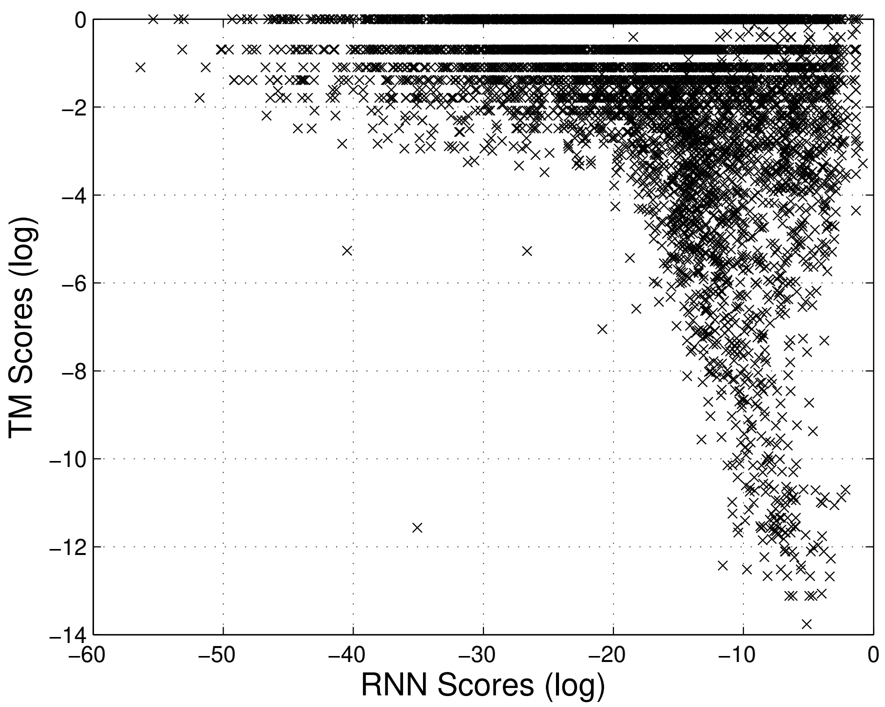


**Figure 4.3** – The visualization of phrase pairs according to their scores (log-probabilities) by the RNN Encoder-Decoder and the translation model.

corresponding $p(\mathbf{f} \mid \mathbf{e})$ from the translation model. Since the existing translation model relies solely on the statistics of the phrase pairs in the corpus, we expect its scores to be better estimated for the frequent phrases but badly estimated for rare phrases. Also, as we mentioned earlier in Sec. 4.3.1, we further expect the RNN Encoder-Decoder which was trained without any frequency information to score the phrase pairs based rather on the linguistic regularities than on the statistics of their occurrences in the corpus.

We focus on those pairs whose source phrase is *long* (more than 3 words per source phrase) and *frequent*. For each such source phrase, we look at the target phrases that have been scored high either by the translation probability $p(\mathbf{f} \mid \mathbf{e})$ or by the RNN Encoder-Decoder. Similarly, we perform the same procedure with those pairs whose source phrase is *long* but *rare* in the corpus.

Table 4.2 lists the top-3 target phrases per source phrase favored either by the translation model or by the RNN Encoder-Decoder. The source phrases were randomly chosen among long ones having more than 4 or 5 words.

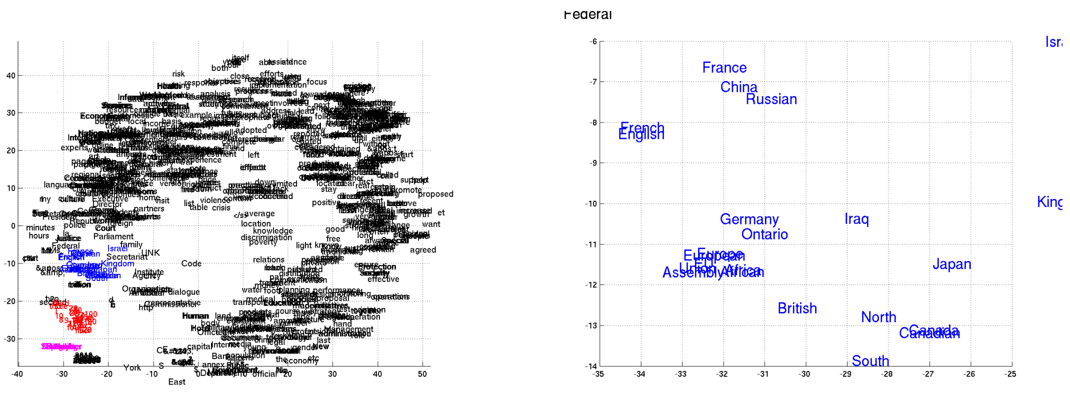


**Figure 4.4** – 2–D embedding of the learned word representation. The left one shows the full embedding space, while the right one shows a zoomed-in view of one region (color-coded). For more plots, see the supplementary material.

In most cases, the choices of the target phrases by the RNN Encoder-Decoder are closer to actual or literal translations. We can observe that the RNN Encoder-Decoder prefers shorter phrases in general.

Interestingly, many phrase pairs were scored similarly by both the translation model and the RNN Encoder-Decoder, but there were as many other phrase pairs that were scored radically different (see Fig. 4.3). This could arise from the proposed approach of training the RNN Encoder-Decoder on a set of unique phrase pairs, discouraging the RNN Encoder-Decoder from learning simply the frequencies of the phrase pairs from the corpus, as explained earlier.

Furthermore, in Table 4.3, we show for each of the source phrases in Table 4.2, the generated samples from the RNN Encoder-Decoder. For each source phrase, we generated 50 samples and show the top-five phrases accordingly to their scores. We can see that the RNN Encoder-Decoder is able to propose well-formed target phrases without looking at the actual phrase table. Importantly, the generated phrases do not overlap completely with the target phrases from the phrase table. This encourages us to further investigate the possibility of replacing the whole or a part of the phrase table with the proposed RNN Encoder-Decoder in the future.

### 4.4.4 Word and Phrase Representations

Since the proposed RNN Encoder-Decoder is not specifically designed only for the task of machine translation, here we briefly look at the properties of the trained

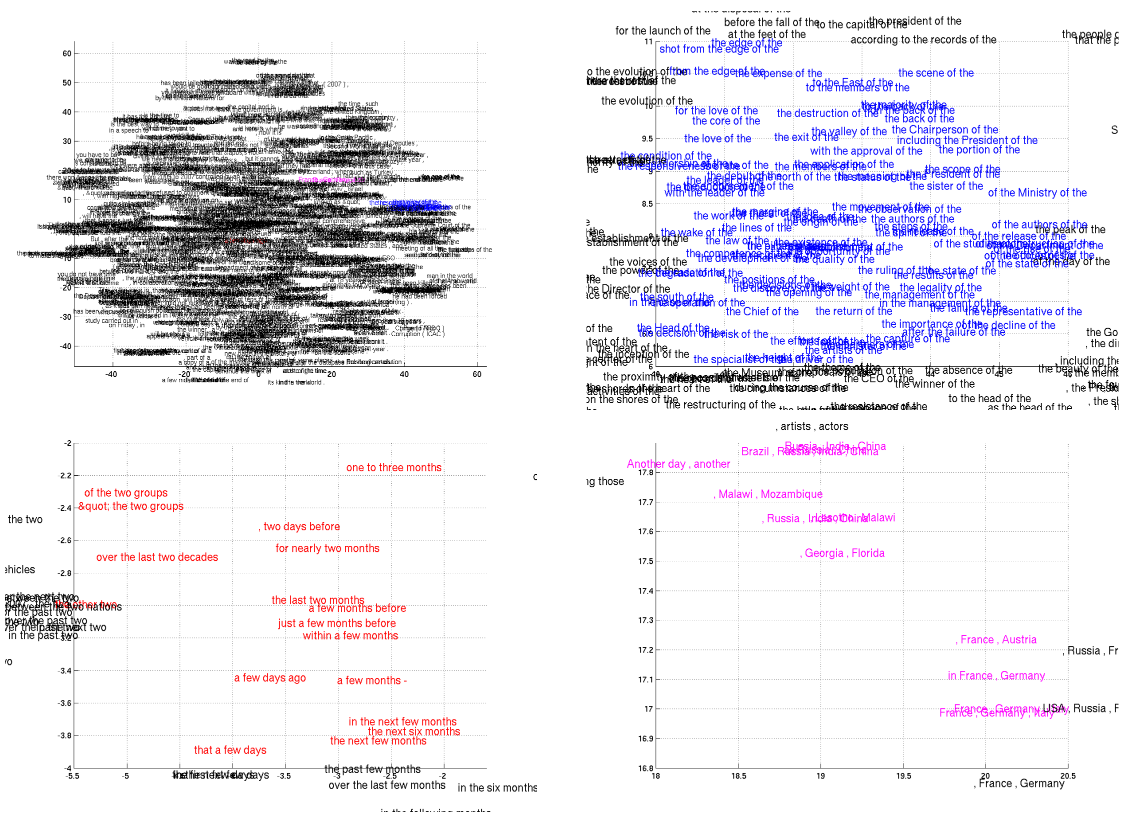


**Figure 4.5** – 2–D embedding of the learned phrase representation. The top left one shows the full representation space (5000 randomly selected points), while the other three figures show the zoomed-in view of specific regions (color-coded).

model.

It has been known for some time that continuous space language models using neural networks are able to learn semantically meaningful embeddings (See, e.g., Y. Bengio, Ducharme, et al. (2003) and Mikolov, Sutskever, et al. (2013)). Since the proposed RNN Encoder-Decoder also projects to and maps back from a sequence of words into a continuous space vector, we expect to see a similar property with the proposed model as well.

The left plot in Fig. 4.4 shows the 2–D embedding of the words using the word embedding matrix learned by the RNN Encoder-Decoder. The projection was done by the recently proposed Barnes-Hut-SNE (Maaten, 2013). We can clearly see that semantically similar words are clustered with each other (see the zoomed-in plots in Fig. 4.4).

The proposed RNN Encoder-Decoder naturally generates a continuous-space representation of a phrase. The representation (**c** in Fig. 4.1) in this case is a 1000-dimensional vector. Similarly to the word representations, we visualize the representations of the phrases that consists of four or more words using the Barnes-Hut-SNE in Fig. 4.5.

From the visualization, it is clear that the RNN Encoder-Decoder captures *both semantic and syntactic* structures of the phrases. For instance, in the bottom-left plot, most of the phrases are about the duration of time, while those phrases that are syntactically similar are clustered together. The bottom-right plot shows the cluster of phrases that are semantically similar (countries or regions). On the other hand, the top-right plot shows the phrases that are syntactically similar.

## 4.5 Conclusion

In this paper, we proposed a new neural network architecture, called an *RNN Encoder-Decoder* that is able to learn the mapping from a sequence of an arbitrary length to another sequence, possibly from a different set, of an arbitrary length. The proposed RNN Encoder-Decoder is able to either score a pair of sequences (in terms of a conditional probability) or generate a target sequence given a source sequence. Along with the new architecture, we proposed a novel hidden unit that includes a reset gate and an update gate that adaptively control how much each hidden unit remembers or forgets while reading/generating a sequence.

We evaluated the proposed model with the task of statistical machine translation, where we used the RNN Encoder-Decoder to score each phrase pair in the phrase table. Qualitatively, we were able to show that the new model is able to capture linguistic regularities in the phrase pairs well and also that the RNN Encoder-Decoder is able to propose well-formed target phrases.

The scores by the RNN Encoder-Decoder were found to improve the overall translation performance in terms of BLEU scores. Also, we found that the contribution by the RNN Encoder-Decoder is rather orthogonal to the existing approach of using neural networks in the SMT system, so that we can improve further the performance by using, for instance, the RNN Encoder-Decoder and the neural net language model together.

Our qualitative analysis of the trained model shows that it indeed captures the linguistic regularities in multiple levels i.e. at the word level as well as phrase level. This suggests that there may be more natural language related applications that may benefit from the proposed RNN Encoder-Decoder.

The proposed architecture has large potential for further improvement and analysis. One approach that was not investigated here is to replace the whole, or a part of the phrase table by letting the RNN Encoder-Decoder propose target phrases. Also, noting that the proposed model is not limited to being used with written language, it will be an important future research to apply the proposed architecture to other applications such as speech transcription.

# 5 Prologue to Second Article

## 5.1 Article Details

**On the Properties of Neural Machine Translation: Encoder-Decoder Approaches**.

Cho, KyungHyun, Bart van Merriënboer, Dzimitry Bahdanau, and Yoshua Bengio. 2014. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, Doha, Qatar, October 25, 2014: 103–111.

*Personal Contribution.*

As second author my primary contribution was performing the experiments (data preprocessing, model training) and analysis looking at the properties of NMT models, whereas the primary author developed the new gated recursive convolutional neural network presented in this paper.

## 5.2 Context

Soon after the publication of the first article, Sutskever, Vinyals, and Q. Le (2014) introduced a larger-scale model which was used to translate full sentences instead of phrases. This second article seeks to analyze encoder-decoder models that are trained on full sentences in more detail: How does the fixed-size sentence representation in NMT models affect translation quality for long sentences? How does the NMT model handle rare and out-of-vocabulary words? By gaining a deeper understanding of the behavior of NMT models, and comparing this behavior to traditional phrase-based models, the goal is to identify future directions of research on NMT models.

## 5.3 Contributions

This work helped improve the understanding of NMT models and identified several directions of research in order to improve their performance. In particular, it made explicit the difficulty that fixed-size sentence representations pose when dealing with long sentences, as well as the inability of NMT models to effectively handle rare or out-of-vocabulary words.

This work also introduced and explored the use of a recursive neural network for machine translation in order to compare its performance to the recurrent approaches.

## 5.4 Recent Developments

In follow-up work from the same year (Bahdanau, Cho, and Y. Bengio, 2015) an alignment model is introduced that helps overcome the information bottleneck present in the encoder-decoder model. A larger body of research has since focused on addressing the rare word problem in NMT e.g. through the use of subword tokens (Sennrich, Haddow, and Birch, 2015), backing off to phrase tables (Luong, Sutskever, et al., 2014), and character-level models (Ling et al., 2015).

# 6 On the Properties of Neural Machine Translation: Encoder–Decoder Approaches

## 6.1 Introduction

A new approach for statistical machine translation based purely on neural networks has recently been proposed (Kalchbrenner and Blunsom, 2013; Sutskever, Vinyals, and Q. Le, 2014). This new approach, which we refer to as *neural machine translation*, is inspired by the recent trend of deep representational learning. All the neural network models used in Sutskever, Vinyals, and Q. Le (2014) and Cho et al. (2014) consist of an encoder and a decoder. The encoder extracts a fixed-length vector representation from a variable-length input sentence, and from this representation the decoder generates a correct, variable-length target translation.

The emergence of the neural machine translation is highly significant, both practically and theoretically. Neural machine translation models require only a fraction of the memory needed by traditional statistical machine translation (SMT) models. The models we trained for this paper require only 500MB of memory in total. This stands in stark contrast with existing SMT systems, which often require tens of gigabytes of memory. This makes the neural machine translation appealing in practice. Furthermore, unlike conventional translation systems, each and every component of the neural translation model is trained jointly to maximize the translation performance.

As this approach is relatively new, there has not been much work on analyzing the properties and behavior of these models. For instance: What are the properties of sentences on which this approach performs better? How does the choice of source/target vocabulary affect the performance? In which cases does the neural machine translation fail?

It is crucial to understand the properties and behavior of this new neural machine translation approach in order to determine future research directions. Also, understanding the weaknesses and strengths of neural machine translation might lead to better ways of integrating SMT and neural machine translation systems.

In this paper, we analyze two neural machine translation models. One of them is the RNN Encoder-Decoder that was proposed recently in Cho et al. (2014). The other model replaces the encoder in the RNN Encoder-Decoder model with a novel neural network, which we call a *gated recursive convolutional neural network* (grConv). We evaluate these two models on the task of translation from French to English.

Our analysis shows that the performance of the neural machine translation model degrades quickly as the length of a source sentence increases. Furthermore, we find that the vocabulary size has a high impact on the translation performance. Nonetheless, qualitatively we find that the both models are able to generate correct translations most of the time. Furthermore, the newly proposed grConv model is able to learn, without supervision, a kind of syntactic structure over the source language.

## 6.2 Neural Networks for Variable-Length Sequences

In this section, we describe two types of neural networks that are able to process variable-length sequences. These are the recurrent neural network and the proposed gated recursive convolutional neural network.

### 6.2.1 Recurrent Neural Network with Gated Hidden Neurons

A recurrent neural network (RNN, Fig. 6.1 (a)) works on a variable-length sequence $x = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$ by maintaining a hidden state $\mathbf{h}$ over time. At each timestep $t$, the hidden state $\mathbf{h}^{(t)}$ is updated by

$$\mathbf{h}^{(t)} = f\left(\mathbf{h}^{(t-1)}, \mathbf{x}_t\right),$$

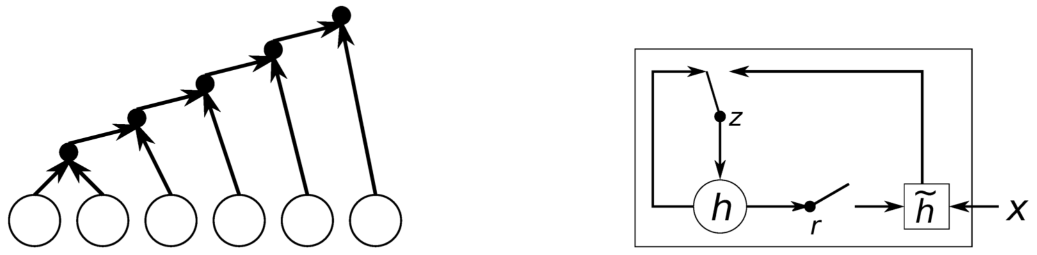


**Figure 6.1** – The graphical illustration of (a) the recurrent neural network and (b) the hidden unit that adaptively forgets and remembers.

where $f$ is an activation function. Often $f$ is as simple as performing a linear transformation on the input vectors, summing them, and applying an element-wise logistic sigmoid function.

An RNN can be used effectively to learn a distribution over a variable-length sequence by learning the distribution over the next input $p(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \ldots, \mathbf{x}_1)$. For instance, in the case of a sequence of 1-of-$K$ vectors, the distribution can be learned by an RNN which has as an output

$$p(x_{t,j} = 1 \mid \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1) = \frac{\exp\left(\mathbf{w}_j \mathbf{h}_{\langle t \rangle}\right)}{\sum_{j'=1}^{K} \exp\left(\mathbf{w}_{j'} \mathbf{h}_{\langle t \rangle}\right)},$$

for all possible symbols $j = 1, \ldots, K$, where $\mathbf{w}_j$ are the rows of a weight matrix $\mathbf{W}$. This results in the joint distribution

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \ldots, x_1).$$

Recently, in Cho et al. (2014) a new activation function for RNNs was proposed. The new activation function augments the usual logistic sigmoid activation function with two gating units called reset, $\mathbf{r}$, and update, $\mathbf{z}$, gates. Each gate depends on the previous hidden state $\mathbf{h}^{(t-1)}$, and the current input $\mathbf{x}_t$ controls the flow of information. This is reminiscent of long short-term memory (LSTM) units (Hochreiter and Jürgen Schmidhuber, 1997). For details about this unit, we refer the reader to Cho et al. (2014) and Fig. 6.1 (b). For the remainder of this paper, we always use this new activation function.

## 6.2.2 Gated Recursive Convolutional Neural Network

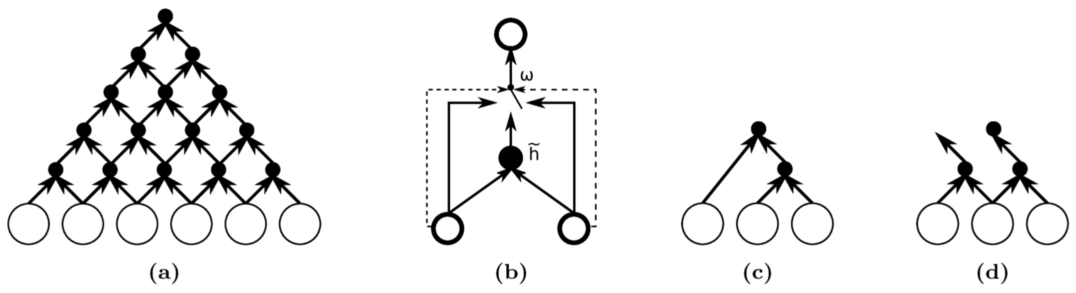


**Figure 6.2** – The graphical illustration of (a) the recursive convolutional neural network and (b) the proposed gated unit for the recursive convolutional neural network. (c-d) The example structures that may be learned with the proposed gated unit.

Besides RNNs, another natural approach to dealing with variable-length sequences is to use a recursive convolutional neural network where the parameters at each level are shared through the whole network (see Fig. 6.2 (a)). In this section, we introduce a binary convolutional neural network whose weights are recursively applied to the input sequence until it outputs a single fixed-length vector. In addition to a usual convolutional architecture, we propose to use the previously mentioned gating mechanism, which allows the recursive network to learn the structure of the source sentences on the fly.

Let $x = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$ be an input sequence, where $\mathbf{x}_t \in \mathbb{R}^d$. The proposed gated recursive convolutional neural network (grConv) consists of four weight matrices $\mathbf{W}^l$, $\mathbf{W}^r$, $\mathbf{G}^l$ and $\mathbf{G}^r$. At each recursion level $t \in [1, T-1]$, the activation of the $j$-th hidden unit $h_j^{(t)}$ is computed by

$$h_j^{(t)} = \omega_c \tilde{h}_j^{(t)} + \omega_l h_{j-1}^{(t-1)} + \omega_r h_j^{(t-1)}, \tag{6.1}$$

where $\omega_c$, $\omega_l$ and $\omega_r$ are the values of a gater that sum to 1. The hidden unit is initialized as

$$h_j^{(0)} = \mathbf{U}\mathbf{x}_j,$$

where $\mathbf{U}$ projects the input into a hidden space.

The new activation $\tilde{h}_j^{(t)}$ is computed as usual:

$$\tilde{h}_j^{(t)} = \phi\left(\mathbf{W}^l h_{j-1}^{(t)} + \mathbf{W}^r h_j^{(t)}\right),$$

where $\phi$ is an element-wise nonlinearity.

The gating coefficients $\omega$'s are computed by

$$\begin{bmatrix} \omega_c \\ \omega_l \\ \omega_r \end{bmatrix} = \frac{1}{Z} \exp\left(\mathbf{G}^l h_{j-1}^{(t)} + \mathbf{G}^r h_j^{(t)}\right),$$

where $\mathbf{G}^l, \mathbf{G}^r \in \mathbb{R}^{3 \times d}$ and

$$Z = \sum_{k=1}^{3} \left[\exp\left(\mathbf{G}^l h_{j-1}^{(t)} + \mathbf{G}^r h_j^{(t)}\right)\right]_k.$$

According to this activation, one can think of the activation of a single node at recursion level $t$ as a choice between either a new activation computed from both left and right children, the activation from the left child, or the activation from the right child. This choice allows the overall structure of the recursive convolution to change adaptively with respect to an input sample. See Fig. 6.2 (b) for an illustration.

In this respect, we may even consider the proposed grConv as doing a kind of unsupervised parsing. If we consider the case where the gating unit makes a hard decision, i.e., $\omega$ follows an 1-of-K coding, it is easy to see that the network adapts to the input and forms a tree-like structure (See Fig. 6.2 (c-d)). However, we leave the further investigation of the structure learned by this model for future research.

## 6.3 Purely Neural Machine Translation

### 6.3.1 Encoder-Decoder Approach

The task of translation can be understood from the perspective of machine learning as learning the conditional distribution $p(f \mid e)$ of a target sentence (trans-

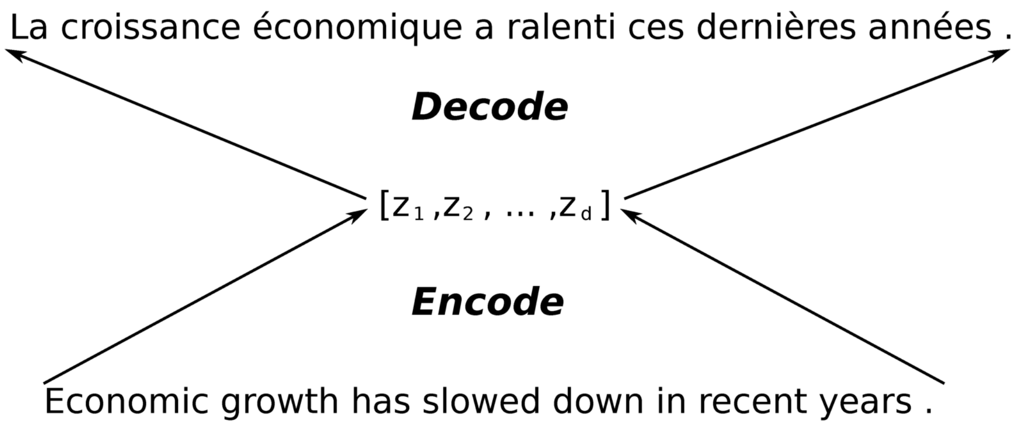

La croissance économique a ralenti ces dernières années .
Decode
[z1 ,z2 , ... ,zd ]
Encode
Economic growth has slowed down in recent years .

**Figure 6.3** – The encoder-decoder architecture

lation) $f$ given a source sentence $e$. Once the conditional distribution is learned by a model, one can use the model to directly sample a target sentence given a source sentence, either by actual sampling or by using a (approximate) search algorithm to find the maximum of the distribution.

A number of recent papers have proposed to use neural networks to directly learn the conditional distribution from a bilingual, parallel corpus (Kalchbrenner and Blunsom, 2013; Cho et al., 2014; Sutskever, Vinyals, and Q. Le, 2014). For instance, the authors of Kalchbrenner and Blunsom (2013) proposed an approach involving a convolutional $n$-gram model to extract a vector of a source sentence which is decoded with an inverse convolutional $n$-gram model augmented with an RNN. In Sutskever, Vinyals, and Q. Le (2014), an RNN with LSTM units was used to encode a source sentence and starting from the last hidden state, to decode a target sentence. Similarly, the authors of Cho et al. (2014) proposed to use an RNN to encode and decode a pair of source and target phrases.

At the core of all these recent works lies an encoder-decoder architecture (see Fig. 6.3). The encoder processes a variable-length input (source sentence) and builds a fixed-length vector representation (denoted as $\mathbf{z}$ in Fig. 6.3). Conditioned on the encoded representation, the decoder generates a variable-length sequence (target sentence).

Before Sutskever, Vinyals, and Q. Le (2014) this encoder-decoder approach was used mainly as a part of the existing statistical machine translation (SMT) system. This approach was used to re-rank the $n$-best list generated by the SMT system in Kalchbrenner and Blunsom (2013), and the authors of Cho et al. (2014) used this approach to provide an additional score for the existing phrase table.

In this paper, we concentrate on analyzing the direct translation performance, as in Sutskever, Vinyals, and Q. Le (2014), with two model configurations. In both models, we use an RNN with the gated hidden unit (Cho et al., 2014), as this is one of the only options that does not require a non-trivial way to determine the target length. The first model will use the same RNN with the gated hidden unit as an encoder, as in Cho et al. (2014), and the second one will use the proposed gated recursive convolutional neural network (grConv). We aim to understand the inductive bias of the encoder-decoder approach on the translation performance measured by BLEU.

## 6.4 Experiment Settings

### 6.4.1 Dataset

We evaluate the encoder-decoder models on the task of English-to-French translation. We use the bilingual, parallel corpus which is a set of 348M selected by the method in Axelrod, He, and J. Gao (2011) from a combination of Europarl (61M words), news commentary (5.5M), UN (421M) and two crawled corpora of 90M and 780M words respectively.[i] We did not use separate monolingual data. The performance of the neural machien translation models was measured on the news-test2012, news-test2013 and news-test2014 sets ($\tilde{3}000$ lines each). When comparing to the SMT system, we use news-test2012 and news-test2013 as our development set for tuning the SMT system, and news-test2014 as our test set.

Among all the sentence pairs in the prepared parallel corpus, for reasons of computational efficiency we only use the pairs where both English and French sentences are at most 30 words long to train neural networks. Furthermore, we use only the 30,000 most frequent words for both English and French. All the other rare words are considered unknown and are mapped to a special token ([UNK]).

i. All the data can be downloaded from `http://www-lium.univ-lemans.fr/~schwenk/cslm_joint_paper/`.

### 6.4.2 Models

We train two models: The RNN Encoder-Decoder (RNNenc)(Cho et al., 2014) and the newly proposed gated recursive convolutional neural network (grConv). Note that both models use an RNN with gated hidden units as a decoder (see Sec. 6.2.1).

We use minibatch stochastic gradient descent with AdaDelta (Zeiler, 2012) to train our two models. We initialize the square weight matrix (transition matrix) as an orthogonal matrix with its spectral radius set to 1 in the case of the RNNenc and 0.4 in the case of the grConv. tanh and a rectifier ($\max(0, x)$) are used as the element-wise nonlinear functions for the RNNenc and grConv respectively.

The grConv has 2000 hidden neurons, whereas the RNNenc has 1000 hidden neurons. The word embeddings are 620-dimensional in both cases.[i] Both models were trained for approximately 110 hours, which is equivalent to 296,144 updates and 846,322 updates for the grConv and RNNenc, respectively.

| Model | Development | Test |
|---|---|---|
| RNNenc | 13.15 | 13.92 |
| grConv | 9.97 | 9.97 |
| Moses | 30.64 | 33.30 |
| Moses+RNNenc$^\star$ | 31.48 | 34.64 |
| Moses+LSTM$^\circ$ | 32 | 35.65 |
| No UNK | | |
| RNNenc | 21.01 | 23.45 |
| grConv | 17.19 | 18.22 |
| Moses | 32.77 | 35.63 |

**(a)** All lengths

| Model | Development | Test |
|---|---|---|
| RNNenc | 19.12 | 20.99 |
| grConv | 16.60 | 17.50 |
| Moses | 28.92 | 32.00 |
| No UNK | | |
| RNNenc | 24.73 | 27.03 |
| grConv | 21.74 | 22.94 |
| Moses | 32.20 | 35.40 |

**(b)** 10–20 words

**Table 6.1** – BLEU scores computed on the development and test sets. The top three rows show the scores on all the sentences, and the bottom three rows on the sentences having no unknown words. ($\star$) The result reported in Cho et al. (2014) where the RNNenc was used to score phrase pairs in the phrase table. ($\circ$) The result reported in Sutskever, Vinyals, and Q. Le (2014) where an encoder-decoder with LSTM units was used to re-rank the $n$-best list generated by Moses.

i. In all cases, we train the whole network including the word embedding matrix. The embedding dimensionality was chosen to be quite large, as the preliminary experiments with 155-dimensional embeddings showed rather poor performance.

## Translation using Beam-Search

We use a basic form of beam-search to find a translation that maximizes the conditional probability given by a specific model (in this case, either the RNNenc or the grConv). At each time step of the decoder, we keep the $s$ translation candidates with the highest log-probability, where $s = 10$ is the beam-width. During the beam-search, we exclude any hypothesis that includes an unknown word. For each end-of-sequence symbol that is selected among the highest scoring candidates the beam-width is reduced by one, until the beam-width reaches zero.

The beam-search to (approximately) find a sequence of maximum log-probability under RNN was proposed and used successfully in Alex Graves (2012a) and Boulanger-Lewandowski, Y. Bengio, and Vincent (2013). Recently, the authors of Sutskever, Vinyals, and Q. Le (2014) found this approach to be effective in purely neural machine translation based on LSTM units.

When we use the beam-search to find the $k$ best translations, we do not use a usual log-probability but one normalized with respect to the length of the translation. This prevents the RNN decoder from favoring shorter translations, behavior which was observed earlier in, e.g., Alex Graves (2013).

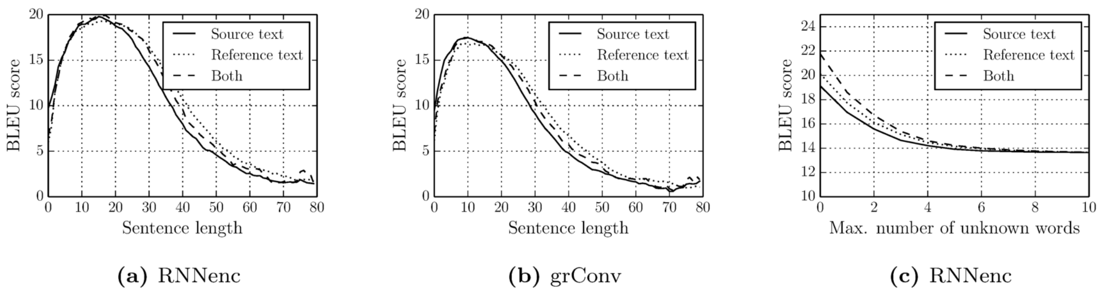


**Figure 6.4** – The BLEU scores achieved by (a) the RNNenc and (b) the grConv for sentences of a given length. The plot is smoothed by taking a window of size 10. (c) The BLEU scores achieved by the RNN model for sentences with less than a given number of unknown words.

**Source** She explained her new position of foreign affairs and security policy representative as a reply to a question: “Who is the European Union? Which phone number should I call?”; i.e. as an important step to unification and better clarity of Union’s policy towards countries such as China or India.

**Reference** Elle a expliqué le nouveau poste de la Haute représentante pour les affaires étrangères et la politique de défense dans le cadre d’une réponse à la question: “Qui est qui à l’Union européenne?” “A quel numéro de téléphone dois-je appeler?”, donc comme un pas important vers l’unicité et une plus grande lisibilité de la politique de l’Union face aux états, comme est la Chine ou bien l’Inde.

**RNNenc** Elle a décrit sa position en matière de politique étrangère et de sécurité ainsi que la politique de l’Union européenne en matière de gouvernance et de démocratie.

**grConv** Elle a expliqué sa nouvelle politique étrangère et de sécurité en réponse à un certain nombre de questions: “Qu’est-ce que l’Union européenne?”.

**Moses** Elle a expliqué son nouveau poste des affaires étrangères et la politique de sécurité représentant en réponse à une question: “Qui est l’Union européenne? Quel numéro de téléphone dois-je appeler?”; c’est comme une étape importante de l’unification et une meilleure lisibilité de la politique de l’Union à des pays comme la Chine ou l’Inde.

---

**Source** The investigation should be complete by the end of the year when the findings will be presented to Deutsche Bank’s board of managing directors — with recommendations for action.

**Reference** L’examen doit être terminé d’ici la fin de l’année, ensuite les résultats du conseil d’administration de la Deutsche Bank doivent être présentés — avec recommandation, d’ habitude.

**RNNEnc** L’étude devrait être terminée à la fin de l’ année, lorsque les conclusions seront présentées au conseil d’administration de la Deutsche Bank, conseil d’association avec des mesures.

**grConv** L’enquête devrait être terminée à la fin de l’année où les conclusions seront présentées par le conseil d’administration de la BCE à la direction des recommandations.

**Moses** L’enquête devrait être terminé d’ici la fin de l’année lorsque les résultats seront présentés à la Deutsche Bank conseil des directeurs généraux — avec des recommandations.

| |
|---|
| **Source** And there are thorny mechanical questions that must be resolved during that time, like how to balance the state's mandate of "adequate access" to licensed marijuana with its prohibitions on cannabis businesses within 1,000 feet of a school, park, playground or child care center. |
| **Reference** Pendant ce temps, des questions pratiques restent en suspens: comment équilibrer le mandat de l'état qui garantit un accès approprié à la marijuana agréée et interdit l'installation de commerces de vente de cannabis dans un rayon de 30 km autour d'une école, d'un parc, d'un terrain de jeu ou d'une crèche. |
| **RNNEnc** Il y a des questions préventives qui se posent quant à l'équilibre des droits de l'enfant dans les limites d'une entreprise de collecte de sang. |
| **grConv** De façon générale, il y a des raisons de sécurité pour que les entreprises aient accès à des milliers de centres de pêche, d'eau ou de recherche. |
| **Moses** Et il y a des problèmes mécaniques complexes qui doivent être résolues au cours de cette période, comme la manière d'équilibrer le mandat de "l'accès adéquat" permis de marijuana avec l'interdiction du cannabis aux entreprises de 1000 pieds d'une école de jeu ou de parc, le service de garde. |

**(a)** Long Sentences

## 6.5 Results and Analysis

### 6.5.1 Quantitative Analysis

In this paper, we are interested in the properties of the neural machine translation models. Specifically, the translation quality with respect to the length of source and/or target sentences and with respect to the number of words unknown to the model in each source/target sentence.

First, we look at how the BLEU score, reflecting the translation performance, changes with respect to the length of the sentences (see Fig. 6.4). Clearly, both models perform relatively well on short sentences, but suffer significantly as the length of the sentences increases.

We observe a similar trend with the number of unknown words, in Fig. 6.4 (c). As expected, the performance degrades rapidly as the number of unknown words increases. This suggests that it will be an important challenge to increase the size of vocabularies used by the neural machine translation system in the future. Although we only present the result with the RNNenc, we observed similar behavior for the grConv as well.

| |
|---|
| **Source** There is still no agreement as to which election rules to follow. |
| **Reference** Toutefois il n'existe toujours pas d'accord selon quel règlement de vote il faut procéder. |
| **RNNEnc** Il n'y a pas encore d'accord sur les règles électorales. |
| **grConv** Il n'y a pas encore d'accord sur la question des élections à suivre. |
| **Moses** Il y a toujours pas d'accord sur l'élection des règles à suivre. |

| |
|---|
| **Source** Many of these ideas may have been creative, but they didn't necessarily work. |
| **Reference** Beaucoup de ces idées étaient créatives mais elles n'ont pas forcément fonctionné. |
| **RNNEnc** Bon nombre de ces idées ont peut-être été créatrices, mais elles ne s'appliquaient pas nécessairement. |
| **grConv** Beaucoup de ces idées peuvent être créatives, mais elles n'ont pas fonctionné. |
| **Moses** Beaucoup de ces idées ont pu être créatif, mais ils n'ont pas nécessairement. |

| |
|---|
| **Source** There is a lot of consensus between the Left and the Right on this subject. |
| **Reference** C'est qu'il y a sur ce sujet un assez large consensus entre gauche et droite. |
| **RNNEnc** Il existe beaucoup de consensus entre la gauche et le droit à la question. |
| **grConv** Il y a un consensus entre la gauche et le droit sur cette question. |
| **Moses** Il y a beaucoup de consensus entre la gauche et la droite sur ce sujet. |

| |
|---|
| **Source** According to them, one can find any weapon at a low price right now. |
| **Reference** Selon eux, on peut trouver aujourd'hui à Moscou n'importe quelle arme pour un prix raisonnable. |
| **RNNEnc** Selon eux, on peut se trouver de l'arme à un prix trop bas. |
| **grConv** En tout cas, ils peuvent trouver une arme à un prix très bas à la fois. |
| **Moses** Selon eux, on trouve une arme à bas prix pour l'instant. |

**(b)** Short Sentences

**Table 6.2** – The sample translations along with the source sentences and the reference translations.

In Table 6.1 (a), we present the translation performances obtained using the two models along with the baseline phrase-based SMT system.[i] Clearly the phrase-based SMT system still shows the superior performance over the proposed purely neural machine translation system, but we can see that under certain conditions (no unknown words in both source and reference sentences), the difference diminishes quite significantly. Furthermore, if we consider only short sentences (10–20 words per sentence), the difference further decreases (see Table 6.1 (b)).

Furthermore, it is possible to use the neural machine translation models together with the existing phrase-based system, which was found recently in Cho et al. (2014) and Sutskever, Vinyals, and Q. Le (2014) to improve the overall translation performance (see Table 6.1 (a)).

This analysis suggests that that the current neural translation approach has its weakness in handling long sentences. The most obvious explanatory hypothesis is that the fixed-length vector representation does not have enough capacity to encode a long sentence with complicated structure and meaning. In order to encode a variable-length sequence, a neural network may "sacrifice" some of the important topics in the input sentence in order to remember others.

This is in stark contrast to the conventional phrase-based machine translation system (Philipp Koehn, Och, and Marcu, 2003). As we can see from Fig. 6.5, the conventional system trained on the same dataset (with additional monolingual data for the language model) tends to get a higher BLEU score on longer sentences.

In fact, if we limit the lengths of both the source sentence and the reference translation to be between 10 and 20 words and use only the sentences with no unknown words, the BLEU scores on the test set are 27.81 and 33.08 for the RNNenc and Moses, respectively.

Note that we observed a similar trend even when we used sentences of up to 50 words to train these models.

## 6.5.2 Qualitative Analysis

Although BLEU score is used as a de-facto standard metric for evaluating the performance of a machine translation system, it is not the perfect metric (see, e.g., Song, Cohn, and Specia (2013) and C. Liu, Dahlmeier, and Ng (2011)). Hence,

i. We used Moses as a baseline, trained with additional monolingual data for a 4-gram language model.

here we present some of the actual translations generated from the two models, RNNenc and grConv.

In Table. 6.2 (a-b), we show the translations of some randomly selected sentences from the development and test sets. We chose the ones that have no unknown words. (a) lists long sentences (longer than 30 words), and (b) short sentences (shorter than 10 words). We can see that, despite the difference in the BLEU scores, all three models (RNNenc, grConv and Moses) do a decent job at translating, especially, short sentences. When the source sentences are long, however, we notice the performance degradation of the neural machine translation models.

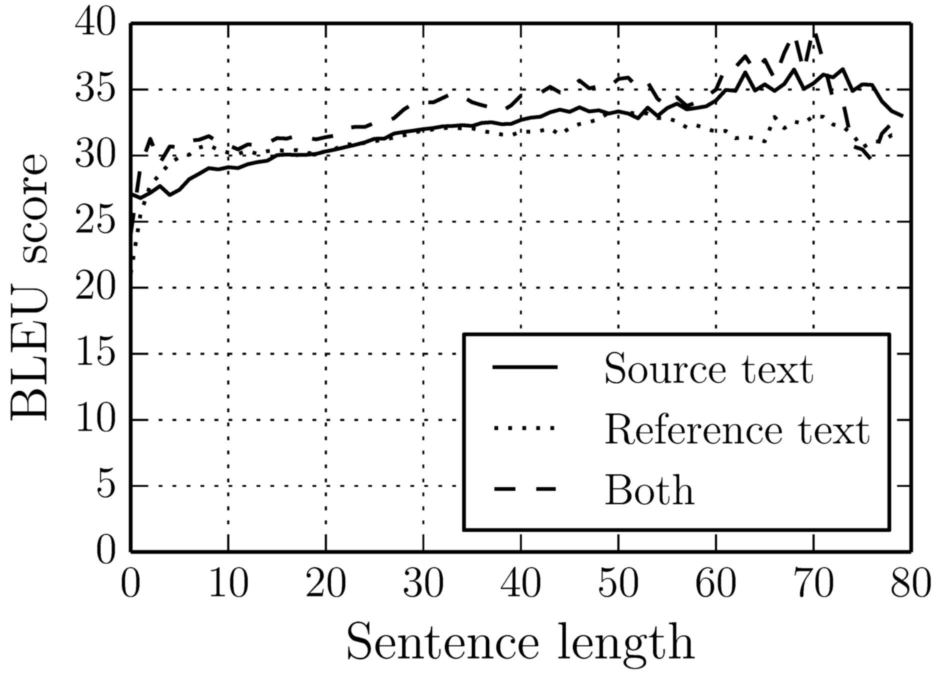
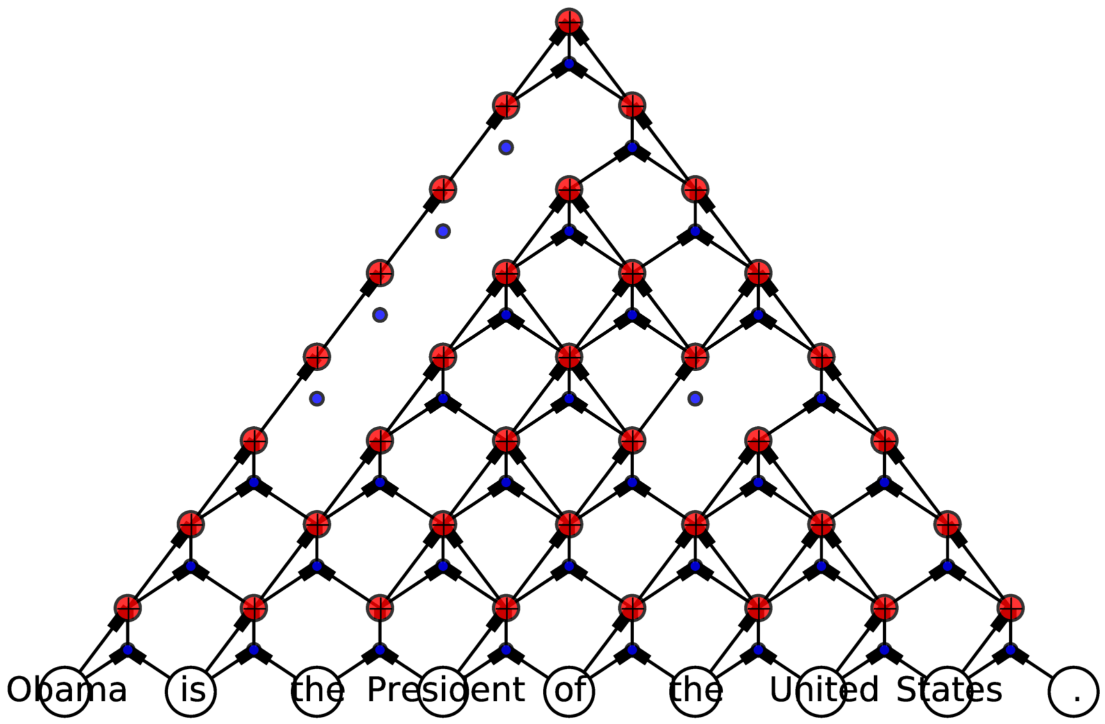


**Figure 6.5** – The BLEU scores achieved by an SMT system for sentences of a given length. The plot is smoothed by taking a window of size 10. We use the solid, dotted and dashed lines to show the effect of different lengths of source, reference or both of them, respectively.

Additionally, we present here what type of structure the proposed gated recursive convolutional network learns to represent. With a sample sentence *"Obama is the President of the United States"*, we present the parsing structure learned by the grConv encoder and the generated translations, in Fig. 6.6a. The figure suggests that the grConv extracts the vector representation of the sentence by first merging *"of the United States"* together with *"is the President of"* and finally combining this



**(a)** The visualization of the grConv structure when the input is *"Obama is the President of the United States."*. Only edges with gating coefficient $\omega$ higher than 0.1 are shown.

with *"Obama is"* and *"."*, which is well correlated with our intuition. Note, however, that the structure learned by the grConv is different from existing parsing approaches in the sense that it returns *soft* parsing.

Despite the lower performance the grConv showed compared to the RNN Encoder-Decoder,[i] we find this property of the grConv learning a grammar structure automatically interesting and believe further investigation is needed.

i. However, it should be noted that the number of gradient updates used to train the grConv was a third of that used to train the RNNenc. Longer training may change the result, but for a fair comparison we chose to compare models which were trained for an equal amount of time. Neither model was trained to convergence.

| Translations | NLL |
|---|---|
| Obama est le Président des États-Unis. | 2.06 |
| Obama est le président des États-Unis. | 2.09 |
| Obama est le président des Etats-Unis. | 2.61 |
| Obama est le Président des Etats-Unis. | 3.33 |
| Barack Obama est le président des États-Unis. | 4.41 |
| Barack Obama est le Président des États-Unis. | 4.48 |
| Barack Obama est le président des Etats-Unis. | 4.54 |
| L'Obama est le Président des États-Unis. | 4.59 |
| L'Obama est le président des États-Unis. | 4.67 |
| Obama est président du Congrès des États-Unis. | 5.09 |

**(b)** The top-10 translations generated by the grConv. The numbers given are the negative log-probability.

**Figure 6.6**

## 6.6 Conclusion and Discussion

In this paper, we have investigated the property of a recently introduced family of machine translation system based purely on neural networks. We focused on evaluating an encoder-decoder approach, proposed recently in Kalchbrenner and Blunsom (2013), Cho et al. (2014), and Sutskever, Vinyals, and Q. Le (2014), on the task of sentence-to-sentence translation. Among many possible encoder-decoder models we specifically chose two models that differ in the choice of the encoder; (1) RNN with gated hidden units and (2) the newly proposed gated recursive convolutional neural network.

After training those two models on pairs of English and French sentences, we analyzed their performance using BLEU scores with respect to the lengths of sentences and the existence of unknown/rare words in sentences. Our analysis revealed that the performance of the neural machine translation suffers significantly from the length of sentences. However, qualitatively, we found that the both models are able to generate correct translations very well.

These analyses suggest a number of future research directions in machine translation purely based on neural networks.

Firstly, it is important to find a way to scale up training a neural network both in terms of computation and memory so that much larger vocabularies for both source and target languages can be used. Especially, when it comes to languages with rich morphology, we may be required to come up with a radically different approach in dealing with words.

Secondly, more research is needed to prevent the neural machine translation system from underperforming with long sentences. Lastly, we need to explore different neural architectures, especially for the decoder. Despite the radical difference in the architecture between RNN and grConv which were used as an encoder, both models suffer from *the curse of sentence length*. This suggests that it may be due to the lack of representational power in the decoder. Further investigation and research are required.

In addition to the property of a general neural machine translation system, we observed one interesting property of the proposed gated recursive convolutional neural network (grConv). The grConv was found to mimic the grammatical structure of an input sentence without any supervision on syntactic structure of language. We believe this property makes it appropriate for natural language processing applications other than machine translation.

# 7 Prologue to Third Article

## 7.1 Article Details

**Multi-scale sequence modeling with a learned dictionary**.

Van Merriënboer, Bart, Amartya Sanyal, Hugo Larochelle, and Yoshua Bengio. 2017. Paper presented at the *Workshop on Machine Learning in Speech and Language Processing*, Sydney, Australia, August 11, 2017.

*Personal Contribution.*

As primary author I was the main contributor to the conceptualization of this paper, its experiments, and its writing.

## 7.2 Context

The neural machine translation (NMT) models proposed in the previous two articles operate on sequences of words in both the encoder and decoder. However, unlike the traditional phrase-based machine translation approach, the runtime of NMT models grows linearly with the size of their input and output vocabularies. Moreover, these models are unable to deal with a long tail of out-of-vocabulary (OOV) words. These concerns revived an interest in character-level language modelling (Chung, Cho, and Y. Bengio, 2016; Kim et al., 2015) and subword-level language modelling (Sennrich, Haddow, and Birch, 2015). Since character-level counterparts are generally less performant and harder to train than their word-level equivalents (Sutskever, Martens, and Geoffrey E. Hinton, 2011) the subword-level language modelling approach, where rare words are split into smaller subword units, was adopted in large-scale production systems such as Google's GNMT system (Yonghui Wu et al., 2016).

## 7.3 Contributions

The work in this article was inspired by the subword-level machine translation approach, which uses a simple form of non-parametric data compression called byte pair encoding (BPE) to determine subword tokens. In this work we extend this semi-parametric approach, using a BPE-variant over the entire sentence for the application of language modelling. Further, although BPE compression outputs a single segmentation of a string, there are in reality many valid segmentations. In this work, we take a more principled approach and marginalize over all possible segmentations. This requires the implementation of custom GPU kernels which are able to handle the irregular access patterns presented by this model.

## 7.4 Recent Developments

The work in Buckman and Neubig (2018) was published soon after this workshop paper and introduces a nearly identical model. To avoid the need for writing custom GPU kernels able to handle irregular data access, the authors restrict themselves to regular lattices, which means that all character n-grams up to a certain order are part of the dictionary. This work considers a wider range of strategies for averaging incoming states, and the model is evaluated on several languages. Encouraging is that their research shows similar results on English (a small improvement over the baseline) but much more significant improvements on Chinese.

# 8 Multiscale sequence modeling with a learned dictionary

## 8.1 Introduction

Sequence modeling is the task of learning a probability distribution over a set of finite sequences. We consider sequences of elements drawn from a finite set of symbols, $s_t \in \Sigma$. In the context of language modeling this is the probability mass function of a set of strings given some alphabet of characters.

$$p(s_1 \dots s_n), \quad s_t \in \Sigma \tag{8.1}$$

Most approaches in language modeling follow Shannon (1948) and model sequences as acyclic Markov chains, exploiting the fact that natural languages have strong temporal dependencies (see figure 8.1).

$$p(s_1 \dots s_n) \approx \prod_{t=1}^{n} p(s_t \mid s_1 \dots s_{t-1}), \quad s_t \in \Sigma \tag{8.2}$$

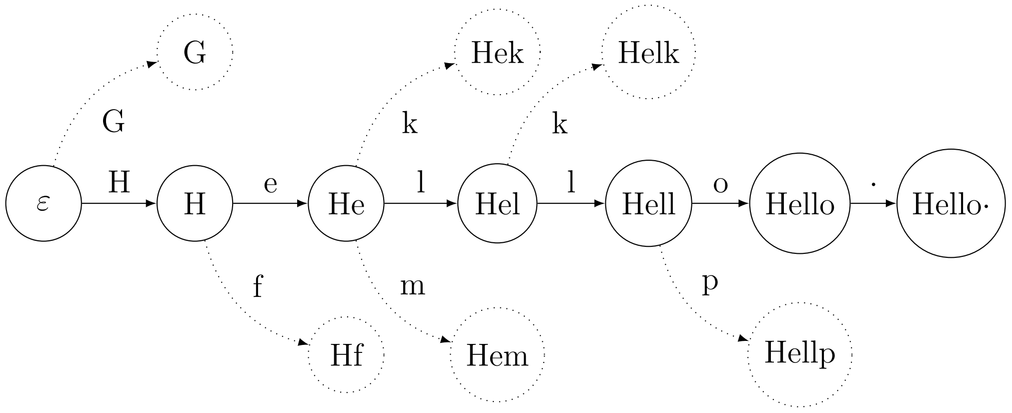


**Figure 8.1** – Diagrammatic representation of a character-level language model as a Markov chain. The probability of the string "Hello" is the probability of reaching the absorbing state "Hello·" starting from the empty string ($\varepsilon$), where $\cdot$ is a special end-of-string (EOS) token. Each state transition is analogous to concatenating a token from the dictionary $\Sigma$ to the state.

Recurrent neural networks (RNNs) can be used to efficiently model these Markov chains (Mikolov, 2012; Mikolov, Karafiát, et al., 2010). The hidden state of the network can encode the subsequence that is conditioned on $(s_1 \ldots s_{t-1})$ using constant memory. Let $\boldsymbol{x}_t$ be an embedding of symbol $s_t$, then an RNN model is of the form

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \tag{8.3}$$

$$\boldsymbol{y}_t = g(\boldsymbol{h}_t) \tag{8.4}$$

Typically the function $f$ is a long-short term memory (LSTM) unit or gated recurrent unit (GRU), and $g$ is a linear transformation followed by a softmax activation.

### 8.1.1 Tokenization

Natural language is naturally represented as a sequence of characters (Sutskever, Martens, and Geoffrey E. Hinton, 2011; Mikolov, 2012; Alex Graves, 2013). However, in practice text is usually 'tokenized' and modeled as a sequence of words instead of characters (see figure 8.2). Word-level models display superior performance to character-level models, which we argue can be explained by several factors.

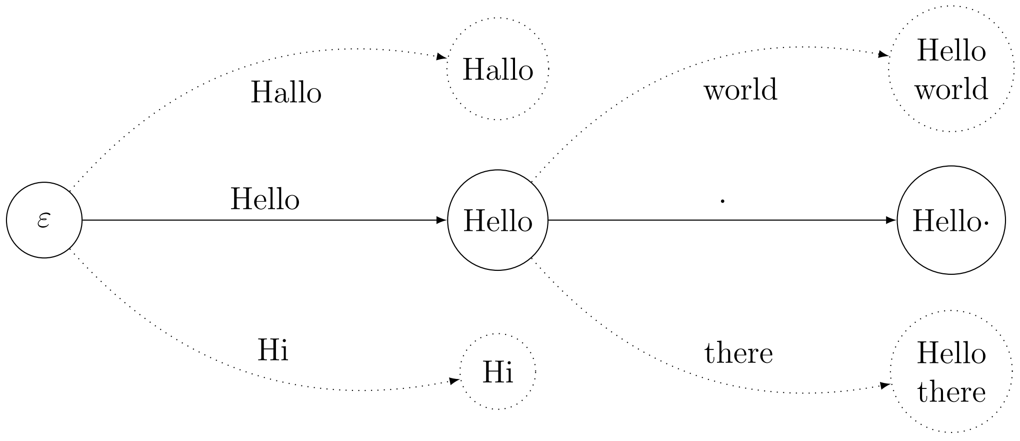


**Figure 8.2** – A word-level language model, requiring fewer transitions in order to reach the state "Hello·". The state space is significantly reduced, which means that many strings cannot be modeled.

## Training difficulties

Tokenization reduces the length of the sequences to model. Learning long-term dependencies with RNNs can be difficult (Razvan Pascanu, Mikolov, and Y. Bengio, 2013; Y. Bengio, Simard, and Frasconi, 1994; Hochreiter, 1991), and in natural language dependencies such as agreement in number or case can span tens or even hundreds of characters.

Furthermore, the softmax generally used in neural networks can never assign a probability of exactly 1 to the correct token. The product of many probabilities less than 1 in equation 8.2 causes the probability of a sequence to decay quickly as it grows longer. To counteract this behaviour a network will quickly learn to fully saturate the softmax. However, this slows down learning (LeCun, Bottou, et al., 1998).

## Compositionality

In the context of natural language it can be argued that character-level and word-level language modeling are fundamentally different.

Word-level models rely on the principle of compositionality (Szabó, 2017) and can learn to represent the meaning of a sentence by combining the semantic representations of its constituent words (J. Mitchell and Lapata, 2008). This mapping is arguably 'smooth': If words have similar semantics, the sentences they form are likely to have a similar meaning and representation as well.

A character-level model performs a second, different task: Mapping a sequence of characters to the semantic representation of a morpheme. The principle of compositionality does not apply in this case. It is a lookup operation which is entirely non-linear: 'the', 'then' and 'they' are entirely unrelated. It is possible that character-level RNN models perform worse than word-level models because RNNs are ill-suited to perform this lookup operation. It is feasible that other application domains have a similar hierarchical structure in their sequential data e.g. sequence motifs in genetics.

## 8.2 Multi-scale sequence modeling

In typical sequence models, we model the likelihood of the next symbol individually. A single symbol (e.g. a character) is selected from a dictionary of mutually exclusive options. In this paper we propose a more general setting in which at each step we make predictions over multi-symbol tokens, potentially multiple of which are correct if they share a prefix (see figure 8.3).

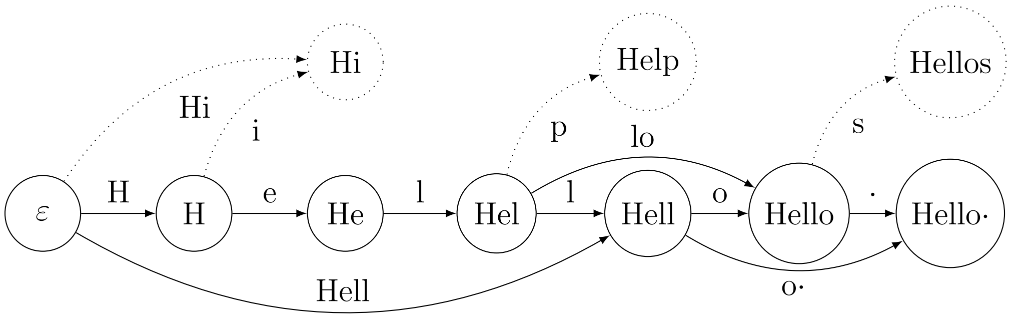


**Figure 8.3** – The multi-scale model allows multiple outgoing transitions, maintaining the flexibility of a character-level model while incorporating many of the benefits of word-level models. Any path through the Markov chain from $\varepsilon$ to Hello· is a segmentation of the string Hello using the tokens in the dictionary. The probability of the state Hello· is the sum of the likelihood of each segmentation. When modeled using an RNN, each state corresponds to a hidden state $\boldsymbol{h}_t$, and each arrow corresponds to the application of the transition function $f$ which takes inputs $\boldsymbol{h}_t$ and token embedding $\boldsymbol{x}_i$.

Formally, given a set of symbols $\Sigma$, consider a dictionary of multi-symbol tokens $T$, where $\Sigma \subset T$. (This condition guarantees that the space of sequences we can model is the same as for typical symbol-level models.) Let $|t_i|$ denote the number of symbols in token $t_i$. The Markov chain (see figure 8.3) for a sequence $s_1 \dots s_n$, $s_t \in \Sigma$, can be modeled using an RNN as follows:

$$\boldsymbol{h}_t = \frac{1}{|T_t|} \sum_{T_t} f(\boldsymbol{x}_i, \boldsymbol{h}_{t-|t_i|}), \tag{8.5}$$

$$T_t = \{t_i : t_i \in T, t_i = s_{t-|t_i|+1} \dots s_t\} \tag{8.6}$$

$$\boldsymbol{y}_t = g(\boldsymbol{h}_t) \tag{8.7}$$

where $\boldsymbol{x}_i$ is an embedding of token $t_i$. Note that a typical RNN model (e.g. a character-level language model) is a special case of this model where $T = \Sigma$.

The likelihood of this model is tractable and can be easily calculated using dynamic programming. We can optimize this likelihood directly using gradient descent. This is similar to the forward-backward algorithm used in hidden Markov models and connectionist temporal classification (CTC) (A. Graves et al., 2006).

$$p\left(s_1 \ldots s_t\right) = \sum_{T_t} p\left(t_i | s_1 \ldots s_{t-|t_i|}\right) p\left(s_1 \ldots s_{t-|t_i|}\right) \tag{8.8}$$

This approach can be used in general for the modeling of Markov chains without cycles in the case of a finite set of transitions (even if the state space is infinite). The recurrent neural network predicts the transition probabilities over this finite set of transitions for each state using a representation of the state and a learned representation of each transition. We believe this is a novel approach to modelling acyclical Markov chains using RNNs.

In this work we consider a multiscale generalization of LSTM networks. For transition functions, $f$, with multiple operations we can perform the averaging at any point. We choose to average the cell states and output gates. Note that performing the averaging earlier on reduces the amount of computation.

$$\begin{pmatrix} \boldsymbol{f_i} \\ \boldsymbol{i_i} \\ \boldsymbol{o_i} \\ \boldsymbol{g_i} \end{pmatrix} = \boldsymbol{W}_h \boldsymbol{h}_{t-|t_i|} + \boldsymbol{W}_x \boldsymbol{x}_i + \boldsymbol{b}$$

$$\boldsymbol{c}_t = \frac{1}{N} \sum_{i=1}^{N} \left(\sigma(\boldsymbol{f}_i) \odot \boldsymbol{c}_{t-|t_i|} + \sigma(\boldsymbol{i}_i) \odot \tanh(\boldsymbol{g}_i)\right)$$

$$\boldsymbol{h}_t = \sigma(\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{o}_i) \odot \tanh(\boldsymbol{c}_t)$$

### 8.2.1 Model characteristics

The computational complexity of a regular RNN model grows as a function of the sequence length, $O(T)$. The multiscale model's complexity instead grows as a function of the number of arcs. The number of arcs in a sequence is theoretically bounded by $\frac{T(T+1)}{2}$, but in practice it grows sublinearly with the size of the dictionary. For example, for a dictionary with 16384 tokens we find an average of 2.7 arcs per time step for the text8 dataset.

It should be noted that the computation of arcs can be entirely parallelized, so on a parallel computer (e.g. a GPU) the span (depth) of the computation is equivalent to that of a normal RNN, $O(T)$.

During training time the memory usage of an RNN model grows as $O(T)$ because of the need to store the hidden states for the backward propagation. The multiscale model's memory usage grows the same and does not depend on the number of arcs, since the averages (see formula 8.5) can be calculated by accumulating values in-place. The need to keep token embeddings in memory means that the memory usage grows as $O(T + D)$ where $D$ is the dictionary size.

In conclusion, the multiscale model is both computationally and memory efficient. On a parallel architecture it has a the same computational complexity as a regular RNN and only requires a small amount of extra memory in order to store the token embeddings.

### 8.2.2 Dictionary learning

The formulation of our multi-scale model requires the construction of a dictionary of multi-symbol tokens. Heuristically speaking, we would simplify our modeling problem if we construct a dictionary which allows each sequence to be segmented into a short sequence of tokens, minimizing the shortest path length through the graph (see figure 8.3).

In natural language processing, word-level models usually construct a dictionary by splitting strings on whitespace and punctuation. The dictionary then consists of the $N$ most frequent tokens, with the rest of the words replaced with a special out-of-vocabulary (OOV) token. Note that many other application domains (e.g. modeling DNA sequences) don't have any straightforward heuristics to tokenize the data.

Even in language modeling this type of tokenization is problematic for a variety of reasons. The number of words in natural language is effectively infinite for synthetic languages, which means there will always be OOV tokens. Furthermore, it is arguably arbitrary from a linguistic perspective. Whereas English is a rather isolating language, with ∼1.67 morphemes per word (Greenberg, 1960) on average, synthetic languages such as Turkish or Eskimo have ∼2.33 and ∼3.70 morphemes per word respectively. For example, the Dutch word *meervoudigepersoonlijkhei-*

*dsstoornis* (multiple personality disorder) contains 10 morphemes. For these types of languages, we might want to consider a tokenization that contains subword units. On the other hand, for highly isolating languages we might want to model several words as a single token e.g. *chúng tôi*, Vietnamese for 'we'.

### 8.2.3 Dictionary coders

Instead of arbitrarily splitting on whitespace, a more principled approach is to to 'learn' the tokens to be modeled. Here we propose an approach which is grounded in text compression and inspired by the byte-pair encoding (BPE) algorithm. BPE has been used in the domain of neural machine translation to learn subword units, reducing the number of OOV tokens (Sennrich, Haddow, and Birch, 2015).

Dictionary coder algorithms like BPE learn dictionaries of tokens with the purpose of representing strings with as few tokens as possible, increasing the level of compression. This reduces the effective depth of the unrolled RNN network (i.e. the shortest path through the graph in figure 8.3), which is a reasonable learning objective for our dictionary.

**Algorithm 3** Adapted byte-pair encoding algorithm

```
Require: $T_{\max}, T = \{s_1, \ldots, s_m\}, \boldsymbol{s} = s_{i_1}, \ldots, s_{i_n}$    ▷ Initial dictionary $T = \Sigma$,
  string $\boldsymbol{s}$
  while true do
      $j, k \hookleftarrow \arg\max_{j,k} \mathit{paircount}(s_j, s_k)$
      $s_{\text{new}} \hookleftarrow [s_j | s_k], \quad T \hookleftarrow T \bigcup \{s_{\text{new}}\}$
      if $|T| = T_{\max}$ then
          break
      end if
      Substitute each occurrence of $s_j, s_k$ in $\boldsymbol{s}$ with $s_{\text{new}}$
      for all $l \in \{l : \mathit{count}(s_l) < \mathit{count}(s_{new})\}$ do
          $T \hookleftarrow T \setminus \{s_l\}$
          Substitute each occurrence of $s_l$ in $\boldsymbol{s}$ with $s_o, s_p$ s.t. $[s_o | s_p] = s_l$
      end for
  end while
  return $T$
```

Regular BPE starts with a dictionary of characters and consecutively replaces the most frequent pairs of tokens with a single new token, until a given dictionary size $T_{\max}$ is reached. We extend the algorithm by reversing the merger of two tokens whenever a token becomes too rare. As a motivating example consider the string *abcabcabc*.... In this case $a$ and $b$ are merged into $ab$, followed by a merger between $ab$ and $c$ into $abc$. Our extension makes sure that the token $ab$, which now occurs zero times, is removed from the dictionary. This removal prevents us from wasting space in the dictionary on rare tokens.

Our implementation of this algorithm uses a bit array of the size of the input data, where each element signifies whether the corresponding character is merged with the subsequent character. We maintain two $d$-ary heaps of tokens and token pairs sorted by their frequency. The algorithm proceeds by repeatedly popping the most common pair from the heap and searching the text and bit array for occurences. If an occurence is found, the bit array is updated to represent the merge, and the $d$-ary heaps are updated to reflect the new token and pair counts. This requires a minimum of $D$ passes over the data to construct a dictionary of size $D$ but uses a relatively small amount of memory.

## 8.3 Experiments

### 8.3.1 Implementation

The irregular, data-dependent access pattern makes the multiscale model difficult to implement in a performant manner using existing GPU-accelerated deep learning frameworks such as Theano and Torch. Hence, experiments were performed with a hand-written CUDA implementation of both the model (including layer normalization) and the dynamic programming forward and backward sweeps. Our implementation was able to exploit the parallelism inherent in the model, fully utilizing the K20 and K80 NVidia GPUs that the models were trained on.

### 8.3.2 Penn Treebank

We evaluate the multiscale model on the widely used Penn Treebank dataset using the training, validation and test split proposed by Mikolov, Karafiát, et al. (2010). Our baseline is an LSTM with 1024 hidden units and embeddings of dimension 512 trained with truncated backpropagation-through-time (TBPTT) on

| # | Token |
|---|---|
| 1 | and· |
| 2 | a· |
| 3 | the· |
| 7 | s· |
| 8 | of·the· |
| 11 | in·the· |
| 12 | ed· |
| 17 | ing· |
| 65 | ation· |
| 239 | people· |
| 245 | man |
| 296 | ed·by·the· |
| 525 | external·links· |
| 540 | at·the·end·of·the· |
| 565 | at·the·university·of· |
| 608 | united·states· |
| 1468 | in·the·united·states· |
| 2727 | one·of·the·most· |

**Table 8.1** – A sample from the tokens in the dictionary of size 8192 constructed using the text8 dataset by our adapted BPE algorithm. Spaces are visualized with the · character. The most common tokens are similar to the ones traditionally found in word-level models e.g. 'and ', 'the ', and 'a '. The dictionary also contains common suffixes such as 's ' (for plural nouns and third person singular verbs), 'ing ' (for gerunds and verbal actions), and 'ed ' (for adjectives, past tenses and past participles), as well as multi-word tokens e.g. 'of the', 'and the', 'in the', etc. and longer phrases.

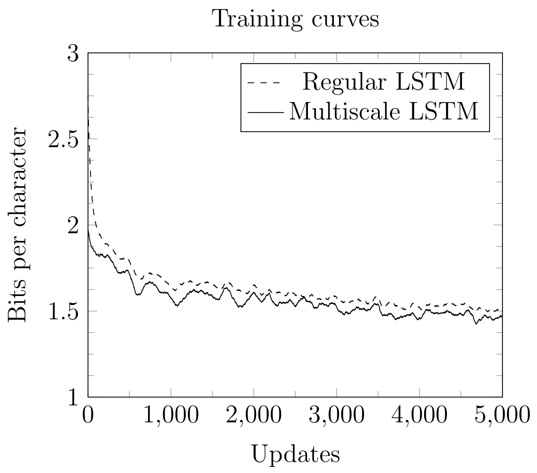


**Figure 8.4** – Training curves of the regular and multiscale LSTM. Note how the multiscale LSTM training loss starts lower because of the learned dictionary, which shows that the use of compression algorithms for dictionary construction is effective.

| Samples |
|---|
| the ·independ·ence ·in the ·third quarter ·the ·chief ·ex·port ·stock ·prices ·for the ·year· ·and ·into the ·disa·ster·l·and |
| gains ·so ·on the ·economy ·because ·in addition ·to a ·compl·ex ·closed ·higher ·comm·ut·e ·pres·sure ·of ·his ·company |
| meeting ·in ·the ·trust ·is ·expected to ·be ·an·ticip·ated · $ · ·offic·es ·during the ·past |
| actu·ally ·have ·spok·es·man ·with ·hous·ing ·their ·junk ·bond ·due · $ N billion from ·most ·important ·next ·day ·at |

**Table 8.2** – Samples from the multiscale model trained on Penn Treebank. Token boundaries are marked with the · symbol. The samples show the model's ability to model a sentence by predicting entire words or phrases ('in addition', 'into the') at a time, while also being able to exploit subword structure ('comm·ut·e') and maintaing the flexibility of character language models to output unseen words ('disa·ster·l·and').

sequences of length 400. These optimal values were found using a grid search. We train using the Adam (Kingma and J. Ba, 2015) optimizer (learning rate of 0.001) and to increase convergence speed we use layer normalization (J. L. Ba, Kiros, and Geoffrey E Hinton, 2016).

Our baseline model achieves a score of 1.43 bits per character. The multiscale model is trained using the exact same configuration but using a dictionary of 2048 tokens. It achieves a test score of 1.42 bits per character. Note that the multiscale models improvements are orthogonal to what can be achieved by straightforwardly increasing the capacity of the network. The regular LSTM networks with more than 1024 units showed decreased performance in our experiments due to overfitting.

Moreover, our network is able to achieve better performance with far fewer parameters. The multiscale model with 512 hidden units, embeddings of size 256, and 2048 tokens has 51% fewer parameters compared to our baseline, but achieves a score of 1.41 bpc, compared to 1.48 bpc for a regular LSTM with the same embedding and hidden state size.

### 8.3.3 Text8

Text8 (Mahoney, 2006) is a text dataset of 100 million characters built from the English Wikipedia. The characters are limited to the 26-letter alphabet and spaces.

We use the traditional split of 90, 5 and 5 million for the training, validation and test set respectively.

We compare the performance of our multi-scale model with a single-layer character-level language model with 2048 units. The same training procedure as described in the previous subsection is used. The baseline achieves a score of 1.45 bits per character. The multiscale model improves on this performance using a dictionary of 16,384 tokens, achieving a test score of 1.41 bits per character.

## 8.4 Related work

In language modeling a variety of approaches have attempted to bridge the gap between character and word-level models. The approach in Kim et al. (2015) is to apply a convolutional neural network (CNN) with temporal pooling over the constituent characters of a word. The CNN filters of this network can be interpreted as character n-gram detectors. The output of this network is used as the input to an LSTM network which models the word-level dynamics. Note that the resulting model still requires information about word-boundaries.

Other approaches use multi-scale RNN architectures. The model in Bojanowski, Joulin, and Mikolov (2016) uses both a word-level and character-level RNN, the latter being conditioned on the former. This model too still requires knowledge of word boundaries. The approach in Chung, Ahn, and Y. Bengio (2017) does not require word boundaries, and instead uses the straight-through estimator to learn the latent hierarchical structure directly. Their model does not learn separate embeddings for the segments however, and can only output a single character at a time.

The latent sequence decomposition (LSD) model introduced in Chan et al. (2017) is related to our multiscale model, and was shown to improve performance on a speech recognition task. Instead of using compression algorithms the LSD model uses a dictionary of all possible $n$-grams. Since the number of $n$-grams grows exponentially, this limits the the dictionary to very short tokens only. The LSD model uses a regular RNN which is trained on a set of sampled segmentations instead of averaging the hidden states using dynamic programming. This complicates training and makes the likelihood of the model intractable. The recent Gram-CTC model (H. Liu et al., 2017) is also related and does use dynamic programming but still uses a dictionary of character n-grams.

Although our model is competitive with recent methods such as MI-LSTM (Yuhuai Wu et al., 2016) and td-LSTM (X. Zhang, Lu, and Lapata, 2016-06), which achieve 1.44 and 1.63 bits per character on the text8 dataset respectively, other recent models such as HM-LSTM (Chung, Ahn, and Y. Bengio, 2017) have achieved lower scores (1.29 bpc). Since many of the LSTM variations in the literature can be extended to the multiscale model, we believe it is possible to improve the performance of multiscale models further in the future. Similarly, deeper multi-layer extensions to our model are feasible.

## 8.5 Discussion

Through arithmetic encoding it can be shown that modeling data is equivalent to compressing it Mahoney (1999). Using neural networks to improve upon text compression algorithms is a common technique (Mahoney, 2000), but as far as we are aware the reverse has not been researched. One can see our model as a mix between non-parametric and parametric approaches: As discussed in Section 8.1.1, character-level models learn a parametric mapping from constituent characters to semantic representations of morphemes. Word-level models avoid learning this highly non-linear function by constructing a dictionary and learning a representation for each word, which is non-parametric. Our multiscale model generalizes this approach, combining non-parametric dictionary coders and parametric RNN models. The size of the dictionary allows us to choose the balance between the two approaches.

A rough parallel can be drawn between our multiscale approach for sequences and superpixels in the computer vision domain (Ren and Malik, 2003), where pixels are clustered in order to improve computational and representational efficiency

The multiscale model can also be related to work on text segmentation. The hierarchical Pitman-Yor language model in Mochihashi, Yamada, and Ueda (2009) learns how to segment a string of characters into words, while simultaneously learning a word-level $n$-gram model. Each path through the graph of the multiscale model (figure 8.3) can be considered a single segmentation of the text, with the likelihood of the string being the marginalization over all possible segmentations.

A large number of combinations of data compression and neural network sequence modeling are still open to investigation. Besides BPE, there are many other dictionary coder algorithms out there. Another consideration would be to learn the dictionary and the sequence model jointly. Subsequently, a variety of neural network models can conceivably be adapted to work with the multi-scale representation of text used in this paper e.g. bag-of-words (BOW) models could be replaced with bag-of-token models instead, similar to the approach in Bojanowski, Grave, et al. (2017) which uses character $n$-grams.

# 9 Automatic differentiation for machine learning

All of the machine learning models discussed so far were optimized using variations of the gradient descent algorithm that was introduced in Section 1.4.2. These algorithms require efficient access to the gradient of the loss with respect to the parameters. The second half of this thesis will focus on *automatic differentiation* (AD, sometimes called *algorithmic differentiation*), a set of techniques and algorithms for calculating derivatives of mathematical functions defined by computer programs.

Automatic differentiation is not to be confused with numerical differentiation (finite differences), which computes an approximation $f'(x) \approx \frac{f(x+\Delta)-f(x)}{\Delta}$. It is also distinct from symbolic differentiation, which is the differentiation of formulae represented as data structures whereas AD is concerned with the differentiation of computer programs[i].

## 9.1 Chain rule

To bridge the gap from calculus to numerical computing, we will begin by deriving the gradient of a simple network using the chain rule while paying particular attention to the dimensionality of the variables involved. The model is a binary classifier with a single $m$-dimensional hidden layer with inputs $\mathbf{x} \in \mathbb{R}^n$ and labels $y \in \{0, 1\}$:

$$\begin{aligned} \mathbf{a} &= \mathbf{W}\mathbf{x} \\ \mathbf{h} &= \sigma(\mathbf{a}) \\ o &= \mathbf{w}^T\mathbf{h} \\ \hat{y} &= \sigma(o) \end{aligned} \tag{9.1}$$

i. Although this distinction between symbolic formulae and computer programs is commonly made, it is quite tenuous when considering purely functional languages (Elliott, 2018).

where $\sigma$ is the logistic function $\frac{1}{1+e^{-x}}$. The loss $\mathcal{L}$ we aim to minimize is the cross-entropy between the model output, $\hat{y}$, and the target label, $y$.

$$\mathcal{L} = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right)$$

The gradient of the loss with respect to the input can be derived by hand through application of the chain rule.

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\hat{y}}\frac{\mathrm{d}\hat{y}}{\mathrm{d}o}\frac{\mathrm{d}o}{\mathrm{d}\mathbf{h}}\frac{\mathrm{d}\mathbf{h}}{\mathrm{d}\mathbf{a}}\frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{x}} \tag{9.2}$$

The first two terms on the right hand side are scalar derivatives given by

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\hat{y}} &= -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \\ &= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \\ \frac{\mathrm{d}\hat{y}}{\mathrm{d}o} &= \sigma(o)\sigma(-o)\end{aligned}$$

The next term, $\frac{\mathrm{d}o}{\mathrm{d}\mathbf{h}}$, is no longer a scalar derivative. Since $o \in \mathbb{R}$ and $\mathbf{h} \in \mathbb{R}^m$ it is in fact a gradient with a constant value, $\frac{\mathrm{d}o}{\mathrm{d}\mathbf{h}} = \mathbf{w}^T$. The subsequent term, $\frac{\mathrm{d}\mathbf{h}}{\mathrm{d}\mathbf{a}}$, is $J_\sigma(\mathbf{a})$ where $J_\sigma$ is the $m \times m$ Jacobian matrix of the element-wise logistic function

$$J_\sigma(\mathbf{x}) = \begin{pmatrix} \sigma'(x_1) & 0 & \cdots & 0 \\ 0 & \sigma'(x_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma'(x_m) \end{pmatrix}$$

The last term, $\frac{\mathrm{d}\mathbf{a}}{\mathrm{d}\mathbf{x}}$, is a Jacobian matrix of size $m \times n$, which in this case has the constant value $\mathbf{W}$.

## 9.2 Automatic differentiation

We will now consider how to evaluate equation 9.2 for a given $\mathbf{x}$ and $y$ using automatic differentiation. AD is based on the observation that any numerical program can be broken down into a series of *primitives* (*elementary operations*) such as addition and multiplication. The original program (*primal computation*) is transformed (or interpreted with non-standard semantics) as to produce a new program with different semantics which calculates derivatives. To calculate gradients, this transformation requires each numerical primitive to have an accompanying vector-Jacobian product (VJP) or Jacobian-vector product (JVP). For example, to support a 2D rotation as a primitive either of the functions given in listing 9.1 would have to be defined.

```
def rot(x, theta):    def rot_vjp(x, theta,  def rot_jvp(x, theta,
    R = array([                   v):                    v):
      [ cos(theta),       R = ...                R = ...
       -sin(theta)],      return v @ R           return R @ v
      [sin(theta),
       cos(theta)]])
    return R @ x
```

**Listing 9.1** – The rotation primitive and its JVP and VJP functions. Note that the infix operator `@` signifies matrix-vector multiplication.

Mathematically speaking, one could replace the JVP and VJP functions with a single function that calculates the Jacobian matrix. However, for many functions (e.g. element-wise functions or convolutions) this would be inefficient. For example, the JVP $J_\sigma(\mathbf{x})\mathbf{y}$ is more efficiently expressed as $\operatorname{diag}(J_\sigma(\mathbf{x})) \odot \mathbf{y}$.

Note that any function can be part of the numeric basis of the language, i.e., any function can be a primitive. In traditional AD systems primitives were usually limited to elementary arithmetic and, e.g., trigonometric, logarithmic, and exponential functions. In modern machine learning frameworks primitive functions include matrix multiplications, convolutions, batch normalization, or even entire RNNs with LSTM units.

### 9.2.1 Forward mode

Note that in order to evaluate $\frac{\mathrm{d}\hat{y}}{\mathrm{d}x_i}$ we must have evaluated the primal computation, $\hat{y} = \sigma(o)$. This is true for most non-linear functions. Given this observation we can adopt a greedy evaluation strategy in which we calculate the derivatives as soon as possible, in lockstep with the original program evaluation.

$$\begin{array}{clll c}
 & & \dfrac{\mathrm{d}\mathbf{x}}{\mathrm{d}x_i} = ([i=1], \ldots, [i=n]) & & \\
 & \mathbf{a} = \mathbf{W}\mathbf{x} & \dfrac{\mathrm{d}\mathbf{a}}{\mathrm{d}x_i} = \mathbf{W}\dfrac{\mathrm{d}\mathbf{x}}{\mathrm{d}x_i} & & \\
\Big\downarrow & \mathbf{h} = \sigma(\mathbf{a}) & \dfrac{\mathrm{d}\mathbf{h}}{\mathrm{d}x_i} = J_\sigma(\mathbf{a})\dfrac{\mathrm{d}\mathbf{a}}{\mathrm{d}x_i} & \Big\downarrow & \\
 & o = \mathbf{w}^T\mathbf{h} & \dfrac{\mathrm{d}o}{\mathrm{d}x_i} = \mathbf{w}^T\dfrac{\mathrm{d}\mathbf{h}}{\mathrm{d}x_i} & & \\
 & \hat{y} = \sigma(o) & \dfrac{\mathrm{d}\hat{y}}{\mathrm{d}x_i} = \sigma(o)\sigma(-o)\dfrac{\mathrm{d}o}{\mathrm{d}x_i} & & \\
 & \mathcal{L} = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right) & \dfrac{\mathrm{d}\mathcal{L}}{\mathrm{d}x_i} = \dfrac{\hat{y}-y}{\hat{y}(1-\hat{y})}\dfrac{\mathrm{d}y}{\mathrm{d}x_i} & &
\end{array} \tag{9.3}$$

Note that the equations on the right-hand side can be evaluated using Jacobian-vector products. This is referred to as performing AD in the *forward accumulation mode* (Wengert, 1964) or simply *forward mode*.

### 9.2.2 Reverse mode

Note that to calculate $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}}$ using forward mode we have to calculate $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}x_i}$ for $i = 1, \ldots, n$, which means that calculating our gradient is $O(n)$ times as expensive as evaluating the model. Can we do better?

Looking at equation 9.2 we evaluated the terms from right to left. We could instead evaluate the terms from left to right, which gives rise to *reverse accumulation mode* AD (*reverse mode*). In the context of reverse mode AD we often refer to the the *forward pass*, which is the primal computation augmented with machinery to store intermediate values for use in the backward pass. The *backward pass* (*reverse pass*) computes derivatives using a series of vector-Jacobian products and the values stored during the forward pass.

$$
\begin{array}{cll}
 & \mathbf{a} = \mathbf{W}\mathbf{x} & \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{a}}\mathbf{W} \\
 & \mathbf{h} = \sigma(\mathbf{a}) & \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{a}} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{h}}J_\sigma(\mathbf{a}) \\
\Bigg\downarrow & o = \mathbf{w}^T\mathbf{h} & \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{h}} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}o}\mathbf{w}^T \quad \Bigg\uparrow \\
 & \hat{y} = \sigma(o) & \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}o} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\hat{y}}\sigma(o)\sigma(-o) \\
 & \mathcal{L} = -\left(y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right) & \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\hat{y}} = \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathcal{L}}\frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \\
 & & \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathcal{L}} = 1
\end{array}
\tag{9.4}
$$

### 9.2.3 Runtime and memory complexity

Looking at our reverse mode AD calculation we note that the backward pass has approximately the same number of operations as the forward pass. This is true in general: For functions $f : \mathbb{R}^n \to \mathbb{R}$ which involve $n$ operations, reverse mode AD can evaluate the gradient $f'$ in $O(cn)$ operations where $c \leq 3$ or $c \leq 5$ (depending on whether we consider memory accesses) (Griewank and Walther, 2008).

Looking at the forward mode algorithm, we note that it would require $O(cn^2)$ operations to evaluate $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}}$. This is in fact also a general observation: The runtime complexity of forward mode grows with the number of inputs respectively, whereas reverse mode's runtime complexity grows with the number of outputs. Hence reverse mode is much more efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $m \ll n$, and vice versa forward mode is preferable when $m \gg n$.

In theory, the terms in equation 9.2 can be evaluated in any order by mixing forward and reverse mode. However, finding the optimal evaluation order is an NP-complete problem (Naumann, 2008) and rarely used in practice, although there are some models which are handcrafted to allow for the mixing of the two modes (Gori, Y. Bengio, and Mori, 1989).

A second important consideration is the memory complexity. In forward mode, each variable additionally requires its partial derivative to be stored. This means that the memory complexity of the differentiated program can be expected to approximately double. For reverse mode AD, however, all of the intermediate variables on the left hand side of equation 9.4 must be stored before the partial

Table 9.1 – Overview of checkpointing algorithms. Non-adaptive algorithms require the number of operations to be known in advance. Some algorithms assume that the execution time of each operation, $t_i$, is uniform. Note that each of these algorithms assumes that the number of checkpoints is fixed in advance.

| | Non-adaptive | Adaptive |
|---|---|---|
| Uniform $t_i$ | Binomial checkpointing[a](Griewank, 1992) | a-revolve (Hinze and Sternberg, 2005) |
| Non-uniform $t_i$ | Dynamic programming (Walther, 2004)<br>Heuristic (Sternberg, 2002) | Heuristic (Sternberg, 2002) |

[a] Proven optimal in Grimm, Pottier, and Rostaing-Schmidt (1996) and implemented in Griewank and Walther (2000). Rediscovered in the context of neural networks in Gruslys et al. (2016)

derivatives can be calculated. Hence, the memory complexity of reverse mode AD grows with the number of primitive operations in the original function.

### Checkpointing

For long-running programs this increase in memory complexity can be prohibitive. *Checkpointing* is the ability to trade off a decrease in memory complexity for an increase in runtime complexity by deleting intermediate variables and recomputing them when they are needed during the backward pass. An alternative approach is to move variables to higher levels in the memory hierarchy i.e. moving the variables from memory to disk, or from GPU to CPU memory.

Determining which variables to checkpoint (i.e. keep in memory) is a well studied problem in the AD literature (see table 9.1 for an overview). However, these approaches assume that all variables are scalars which allows the number of checkpoints to be fixed. In the context of deep learning more heuristic techniques are used such as moving variables from GPU to CPU with a least-recently used (LRU) cache, or determining which variables to recompute by considering their computation-to-memory ratio (Linnan Wang et al., 2018).

## 9.2.4 Higher-order differentiation and generalizations

Some AD implementations are closed under their own operations, which means that forward and reverse mode can be applied multiple times in order to calculate higher-order derivatives. For example, the second-order derivative of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ can be calculated using forward-over-forward which would have an overhead of $O(n^2)$. Alternatively reverse-over-forward, forward-over-reverse, or reverse-over-reverse could be used. These methods are theoretically equivalent (Christianson, 2012) with an overhead of $O(nm)$, but ease of implementation and differences in the efficiency of the generated derivative programs means that certain combinations are preferred over others (Naumann, 2012). In particular, for scalar-valued functions the forward-over-reverse algorithm is preferred to calculate the Hessian matrix. Advanced algorithms such as edge pushing (Gower and Mello, 2012) and the live variable-centered Hessian algorithm (M. Wang, Gebremedhin, and Pothen, 2016) improve on the naive forward-over-reverse algorithm by exploiting the symmetry of the Hessian matrix.

The principles of AD can be used not only to calculate $n$-th order derivatives, but also to efficiently calculate Hessian-vector products (Barak A Pearlmutter, 1994), the Gauss-Newton matrix times a vector (Schraudolph, 2002; Martens, Sutskever, and Swersky, 2012), and the uncentered covariance matrix of the gradients[i] times a vector (Schraudolph, 2002).

## 9.3 Implementations

### 9.3.1 Forward mode

The use of forward and reverse mode tells us which computations to perform and in which order. We will now consider how to turn this algorithm into actual code.

Forward mode can be implemented in a straightforward manner through the use of *dual numbers* (Rall, 1986): Each variable $\mathbf{y}$ is augmented with a second variable which holds the value of the partial derivative $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x_i}$ to form the dual number $\mathbf{y} + \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x_i}\epsilon$. Primitives are then overloaded to operate on dual numbers, returning a new dual number constructed by performing the regular computation as well as the Jacobian-vector product. See listing 9.2 for a minimal example.

i. When the gradients are weighted according to the samples' errors this matrix is sometimes referred to as the 'empirical Fisher matrix', which would imply that these products can be used to implement the natural gradient descent algorithm (Le Roux, Manzagol, and Y. Bengio, 2008). In fact, this is a common misunderstanding since the Fisher matrix and uncentered covariance matrix are only equivalent when the model distribution and data distribution are same, which is not the case in general during training.

```
@dataclass
class Dual:
    val: float
    eps: float

    def __mul__(x, y):
        eps = x.val * y.eps + x.eps * y.val
        return Dual(x.val * y.val, eps)

    def __add__(x, y):
        return Dual(x.val + y.val, x.eps + y.eps)

x = Dual(2, 1)  # x = 2, d/dx(x) = 1
y = Dual(3, 0)  # y = 3, d/dx(y) = 0

# d/dx (x^2 + x * y)
z = x * x + x * y
assert z.eps == 7

# d/dx(x * d/dy(x + y)) (fails)
x = y = Dual(1, 1)
assert (x * Dual((x + y).eps, 0)).eps == 1
```

**Listing 9.2** – A minimal example of a forward mode AD implementation.

It must be noted that the implementation of higher-order forward mode requires more care since a naive implementation will lead to a common class of bugs called *perturbation confusion* (Siskind and Barak A Pearlmutter, 2005). For example, when one tries to naively evaluate $\frac{\mathrm{d}}{\mathrm{d}x}\left(x\left(\frac{\mathrm{d}}{\mathrm{d}y}x+y\right)\right)$ for $x, y = 1$ using our minimal implementation in listing 9.2 the result will be 2 instead of 1 because the implementation cannot distinguish between the partial derivatives of the first and second application. A variable tagging system is normally used to address this issue.

The concept of dual numbers was extended to hyper-dual numbers which allow for the calculation of higher order derivatives in a single pass (Karczmarczuk, 2001; Barak A Pearlmutter and Siskind, 2007; Fike and Alonso, 2012).

### 9.3.2 Reverse mode

Forward mode is relatively easy to implement because the partial derivatives are calculated in step with the original computation. Reverse mode is more complicated, since it involves reversing the control flow of the original program. Moreover, during the backward pass the program might need access to the intermediate variables from the forward pass.

Different implementation methods will have trade-offs in terms of their ease of implementation and performance. The paper introduced in the next chapter contains a review of the different methods and their trade-offs.

# 10 Prologue to First Article

## 10.1 Article Details

**Automatic differentiation in ML: Where we are and where we should be going**.

Van Merriënboer, Bart, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, Montréal, Canada, December 2–8, 2018.

*Personal Contribution.*

The Myia compiler and framework is the result of a research and development project that I initiated summer 2016. As the project leader I was responsible for the vision and conceptualization of Myia and the writing of several early prototypes. In the spring of 2017 the project was joined by Olivier Breuleux and Maxime Chevalier-Boisvert who wrote two more prototypes. In the fall of 2017 I worked with Olivier to synthesise the different prototypes into a preliminary software package that integrated dataflow programming with a functional closure based approach to AD. The development of Myia has since been taken over by Olivier Breuleux and Arnaud Bergeron. With regards to the paper I was the main author of the review section and jointly authored the section on Myia with Olivier Breuleux.

## 10.2 Context

TensorFlow (Abadi et al., 2016) was introduced in late 2015, building on the source code transformation and dataflow programming approach pioneered by Theano (Al-Rfou et al., 2016). However, shortcomings of this approach in handling models with large amounts of control flow had started becoming apparent, leading to the popularization of operator-overloading libraries such as Chainer (Tokui et al., 2015) and torch-autograd.

Myia was born out of the desire to reconcile the flexibility and usability of operator overloading frameworks with the performance of the dataflow programming and source code transformation (SCT) approaches. It attempts to do so by combining and generalizing the dataflow programming approaches from TensorFlow and Theano with the powerful closure-based AD approach introduced in Barak A Pearlmutter and Siskind (2008).

## 10.3 Contributions

This paper consists of two parts: The first part is a short literature review of reverse mode AD in the context of ML, which attempts to bridge the gap between the ML, AD, and compiler/PL communities by relaying some of the lesssons that were learned from implementing Myia.

The second part of the paper introduces Myia, a compiler pipeline which uses a novel graph-based representation inspired by A-normal form which is tailored towards supporting reverse mode AD through the use of closures as well as reaching high-performance in typical deep learning workloads through vectorization and parallelization.

Although the focus of Myia is the exploration of closure-based AD transformations on a new graph-based representation, the development of a prototype pipeline required the implementation of several other components: It performs type inference and optimizations, and includes a runtime that eagerly evaluates the program. Note that Myia's optimizations give preference to speed over semantic equivalence, e.g., unused computations such as multiplications with zero are aggressively removed, similar to deep learning frameworks such as Theano. Note that this can change the runtime behavior of the program in a similar way as GCC's `-ffast-math` optimizations.

Since Myia can be compiled using backends such as XLA and NNVM which are used by the TensorFlow and MXNet frameworks, theoretically Myia can achieve similar performance. However, achieving such performance was left to later work, with this paper focusing most strongly on the trade-offs involved when choosing a IR for ML frameworks.

## 10.4 Recent Developments

Implementing general purpose AD algorithms that are able to reach high-performance for deep learning workloads is still an active area of research. Recent additions include Lantern, which uses delimited continuations in Scala to implement reverse mode AD using callbacks (F. Wang, X. Wu, et al., 2018; F. Wang and Rompf, 2018).

# 11 Automatic differentiation in ML: Where we are and where we should be going

## 11.1 Introduction

Recent advances in ML, and deep learning in particular, have in part been driven by advances in hardware (LeCun, Y. Bengio, and G. Hinton, 2015; Jürgen Schmidhuber, 2015). This increase in computational power has spurred the development of a large number of software libraries, compute kernels, programming languages, and compiler toolchains in order to exploit it. We distinguish some features and objectives that separate these ML frameworks from traditional array programming frameworks.

Firstly, many machine learning models use optimization algorithms which require access to derivatives of the model. Automatic differentiation (Griewank and Walther, 2008) comprises a collection of techniques that can be employed to calculate the derivatives of a function specified by a computer program, and is a central feature of popular ML frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017).

ML frameworks also put heavy emphasis on being able to iterate quickly on new models using high-level, dynamically typed languages, while maintaining high performance through aggressively exploiting resources (e.g., through parallelism, distributed computing, accelerators, static optimization). Moreover, since the derivative code is generated programmatically using AD, frameworks cannot always rely on users writing hand-tuned code and must instead provide compiler optimizations.

Despite the growth in ML frameworks, many have been developed in isolation of the AD community, and many of their insights regarding language design and interactions between source transformation and compiler optimizations have gone largely ignored. Moreover, although many ML frameworks have slowly been adopting functional language concepts (such as pure functions, immutable variables, lazy evaluation) many of the standard approaches in use by functional language compilers to guarantee high performance (A-normal form and continuation passing style representations, persistent data structures, heap recycling, etc.) have not been applied.

In some cases popular ML frameworks have sacrificed flexibility and generality compared to popular array programming packages such as NumPy (Walt, Colbert, and Varoquaux, 2011) in order to provide AD and achieve high performance. On the one hand, frameworks relying on computation graphs such as TensorFlow and Theano (Al-Rfou et al., 2016) do not support higher-order functions or recursion, even though some ML models (e.g. Tai, Socher, and Christopher D Manning (2015)) are more naturally expressed using recursion than loops. On the other hand, frameworks relying on operator overloading such as PyTorch and Autograd (Maclaurin, Duvenaud, and Adams, 2015) see performance degradation for models with scalars or small vectors.[i]

## 11.2 Background and prior work

The development of ML frameworks has been driven by a wide range of fields and perspectives—systems programming, automatic differentiation, programming languages, compiler design, applied machine learning, etc.–which has lead to duplicated research and confused terminology (e.g. *define-by-run* and *operator overloading*). To contextualize our proposed framework, the first half of this paper consists of a review which aims to synthesise these different perspectives. We will begin with explaining the nature of AD and the various challenges associated with it. Then we will review the different approaches to AD and relevant prior work from different domains, such as graph representations from the compiler literature, and language and IR design from functional languages. We will discuss the uses of these approaches in existing frameworks and how they affect performance, expressive power, and usability.

Given this insight, our goal is to outline in the subsequent sections a proof of concept of a high-performance ML framework with first-class support for AD, but which has the flexibility and expressive power of a generic, high-level programming

i. `https://github.com/pytorch/pytorch/issues/2518` language so that it does not restrict the ability of ML researchers to explore novel models and algorithms.

### 11.2.1 Automatic differentiation

Automatic differentiation (AD, also called algorithmic differentiation) relies on the ability to decompose a program into a series of elementary operations (primitives) for which the derivatives are known and to which the chain rule can be applied. AD allows for the calculation of derivatives of any order up to working precision.

AD has been studied since the 60s and 70s and has been employed in fields such as computational fluid dynamics, astronomy, and mathematical finance (Griewank and Walther, 2008). Both its implementation and its theory are still an active area of research (e.g., Siskind and Barak A. Pearlmutter (2016) and M. Wang, Gebremedhin, and Pothen (2016)). We recommend Griewank and Walther (2008) and Baydin, Barak A. Pearlmutter, et al. (2018) for a review of AD in general and in the context of machine learning respectively. From an application perspective, AD affects and interacts with the entire toolchain, from language design through intermediate representations, static analysis, to code generation and program execution.

The runtime and memory complexity of AD depends on the order in which the chain rule is evaluated. Evaluating the chain rule from right to left (from inputs to outputs) is referred to as *forward mode*, whereas evaluating it from left to right (from outputs to inputs) is called *reverse mode*. Forward mode has constant memory requirements and its runtime complexity scales with the number of inputs. Reverse mode's runtime complexity scales with the number of outputs, and its memory complexity grows with the number of intermediate variables. In principle, forward and reverse mode can be mixed, but finding the optimal way of doing so is NP-complete (Naumann, 2008).

In forward mode, the partial derivatives of intermediate variables are calculated in step with the original program. As such, forward mode is relatively straightforward to implement, e.g. using dual numbers (Clifford, 1873). In reverse mode, the chain rule is evaluated in reverse order of the original program. This is a more complex program transformation: an *adjoint* program must be constructed whose control flow is the reverse of the original (or *primal*) program. First, the primal program is run to obtain the output, and then the adjoint program is run to compute the gradient, starting from that output and going backwards. In order to do so efficiently, each statement in the adjoint must have access to the intermediate variables of the original program. Hence, the AD transformation must guarantee that the intermediate variables are not destroyed or mutated.

In ML applications, large matrices of input parameters are typically updated using gradient descent on a scalar output cost. Since the number of inputs is significantly larger than the number of outputs, reverse mode AD is to be preferred. The term 'backpropagation' is used to refer to the specialized application of reverse mode AD in machine learning.

Two implementation methods of AD are generally distinguished: operator overloading (OO) and source transformation (ST, also called source code transformation). Each method has its advantages and disadvantages in terms of usability, implementation, and efficiency (C. H. Bischof and Bücker, 2000). We will briefly discuss them in the context of reverse mode AD.

**Operator overloading**

OO relies on a language's ability to redefine the meaning of functions and operators. All primitives are overloaded so that they additionally perform a tracing operation: The primitive is logged onto a 'tape', along with its inputs to ensure that those intermediate variables are kept alive. At the end of the function's execution, this tape contains a linear trace of all the numerical operations in the program. Derivatives can be calculated by walking this tape in reverse.

The main advantage of OO is that it is straightforward to implement. Because the tracing passes through function calls and control flow, the AD logic is simplified. A significant downside is that a separate 'derivative interpreter' is needed for the adjoint program. Having an embedded interpreter inside of the host language can complicate debugging and performance analysis. Moreover, since the program is traced and reversed at runtime, OO incurs overhead on each function call which can be particularly problematic if the primitives are fast to execute relative to the tracing operation. OO also does not allow for ahead-of-time optimizations on the adjoint program.

OO is the technique used by PyTorch, Autograd, and Chainer (Tokui et al., 2015). Non-ML oriented AD frameworks using OO include ADOL-C (Griewank, Juedes, and Utke, 1996) and CppAD (Bell, 2003).

## Source transformation

ST explicitly constructs the adjoint program. Unlike OO, ST needs to explicitly construct a program with a reversed control flow, which means that it needs transformation rules for function calls and control flow statements such as loops and conditionals. Whereas OO operates within the language, ST requires tooling such as parsers, tools to manipulate intermediate representations, and unparsers. The advantage of ST is that the AD transformation is done only once per program and hence doesn't incur overhead at runtime, which makes ST performant for a wider range of workloads. Moreover, the full adjoint program is available during compilation and can therefore be optimized ahead of time.

Although ST does not have to deal with the AD transformation at runtime, it must still ensure that intermediate variables from the forward pass are accessible by the adjoint. There are a variety of approaches to deal with this.

**Tape-based** Frameworks such as ADIFOR (C. Bischof et al., 1996) and Tapenade (Hascoët and Pascual, 2013) for Fortran and C use a global stack also called a 'tape'[i] to ensure that intermediate variables are kept alive. The original (primal) function is augmented so that it writes intermediate variables to the tape during the forward pass, and the adjoint program will read intermediate variables from the tape during the backward pass. More recently, tape-based ST was implemented for Python in the ML framework Tangent (Merriënboer, Moldovan, and Wiltschko, 2018).

A problem of this approach is that the tape is a data structure constructed at runtime, analysis of which requires custom compiler passes (Hascoët, Naumann, and Pascual, 2003; Hascoët and Pascual, 2013). Moreover, adjoint programs have a particular symmetric structure where intermediate variables from the first primal statements are used by the last adjoint statements. This highly non-local structure is unsuitable for traditional compiler optimizations which act locally. Ways of addressing this interaction between AD and compiler optimizations is an ongoing

i. The tape used in ST stores only the intermediate variables, whereas the tape in OO is a program trace that stores the executed primitives as well.

research topic (Siskind and Barak A. Pearlmutter, 2016; Hascoët, 2017). Finally, reading and writing to the tape need to be made differentiable in order to compute higher-order derivatives which involve multiple applications of reverse mode. For this reason most tape-based systems do not support reverse-over-reverse.

**Closure-based** To address some of the shortcomings of the tape-based approach, alternative approaches have been proposed which employ closures (Barak A Pearlmutter and Siskind, 2008) or delimited continuations (F. Wang and Rompf, 2018). In both cases, tools from functional programming are used which can capture the environment of a statement during the forward pass, and execute the corresponding adjoint within that environment. The advantage of this approach is that no AD-specific compiler passes are needed: a functional language compiler will recognize the non-local use of the intermediate variables by the fact that they are free variables in the generated closure or continuation. This avoids the need for custom compiler passes, and allows for the application of all the tooling from functional compilers on the generated adjoint program (Shivers, 1991; Siskind and Barak A Pearlmutter, 2008).

### 11.2.2 Dataflow programming

Popular ML frameworks such as Theano, TensorFlow, and MXNet (T. Chen, Li, et al., 2015) follow the dataflow programming paradigm (Johnston, Hanna, and Millar, 2004) and use computation graphs as their intermediate representation. These graph representations do not have scoping or recursive function calls, which means that AD is much easier to implement with ST. Since the adjoint program is part of the same dataflow graph, it can access the intermediate variables from the forward pass directly from the global scope, so neither tapes nor closures are required. Additionally, a simple liveness analysis makes it easy to keep intermediate values from the primal alive only for as long as required by the adjoint computation.

Using dataflow graphs without function calls[i] nor scoping[ii] introduces limitations. Some of these limitations are addressed by the use of metaprogramming,

i. TensorFlow and Theano implement a type of subroutine through their `Defun` and `OpFromGraph` constructs, but these must be explicitly constructed by the user and don't support recursion.

ii. TensorFlow has a concept it refers to as 'scoping', but these scopes are not lexical and can be reentered at any time, so the lifetime of a value is not affected by its scope.

but others affect the end-user (e.g., the lack of recursion and higher-order functions reduces the expressiveness of the language) and the compiler pipeline (e.g., loops cannot be represented in a principled way, which complicates their implementation).

An advantage of dataflow programming is that graphs are a natural representation for distributed computing (Akidau et al., 2015). This allows different operations to be easily distributed across different hosts, devices, and cores.

Graph-based IRs are generally useful for compilers, since the absence of an explicit ordering can simplify certain optimizations and scheduling. Theano's graph representation in particular was based on the representations used by computer algebra systems (CAS), enabling aggressive algebraic simplification and pattern matching. An SSA [i]-based graph representation (Click and Paleczny, 1995; Lindenmaier et al., 2005), sometimes referred to as *sea-of-nodes*, is used by the HotSpot Java compiler and the V8 TurboFan JavaScript compiler, and a graph representation using continuation-passing style (CPS, an IR commonly used in functional languages) called *Thorin* also exists (Leißa, Köster, and Hack, 2015).

### 11.2.3 Programming languages and compilers

Theano was one of the first software packages to refer to itself as a 'linear algebra compiler'. Since then, more frameworks started approaching the definition and execution of ML models as a compiler problem. In the case of Theano and TensorFlow, they can be considered compilers of a custom language which must be metaprogrammed using Python as a metalanguage. The dataflow graph is an intermediate representation which is optimized using a series of compiler passes. The resulting program is compiled (e.g., XLA) and/or interpreted (e.g., the TensorFlow/Theano runtimes). Similarly, PyTorch has started optimizing its traced Python programs using just-in-time (JIT) compiler approaches.

More recently, projects such as DLVM (Wei, Schwartz, and Adve, 2017) and Swift for TensorFlow [ii] have attempted to extend existing compiler toolchains such as LLVM and Swift's intermediate language (SIL) with array programming and AD in order to create frameworks better suited for ML workflow needs.

i. Static single assignment, which essentially means each variable is assigned to exactly once.
ii. https://www.tensorflow.org/community/swift

Viewing ML frameworks as compiler toolchains raises several questions. For example, on what intermediate representations is it the easiest to apply AD and aggressive optimizations? IRs with closures as first-class objects will be able to use closure-based approaches to AD, whereas traditional SSA-based representations (such as SIL) would need to use a tape-based approach. And which IRs are most suitable for the heavy use of parallelism and distributed computing in ML?

Secondly, what should the source language be? The ML community is highly invested in Python, an interpreted, dynamically typed programming language which does not have built-in support for multidimensional arrays. More recently, frameworks have suggested using Swift (DLVM) or Julia (JuliaDiff, Revels, Lubin, and Papamarkou, 2016), languages with static typing and built-in multidimensional arrays respectively. On the other hand, frameworks such as Theano and TensorFlow do not have an exposed source language but can only be metaprogrammed. In the AD community, there has been strong push away from traditional imperative languages such as Fortran and C to purely functional languages, since they simplify the implementation of AD and are easier to optimize. Examples of this are VLAD, a dialect of Lisp which is compiled with the Stalin$\nabla$ compiler (Siskind and Barak A. Pearlmutter, 2016; Barak A Pearlmutter and Siskind, 2008; Siskind and Barak A Pearlmutter, 2008), DVL[i], and DiffSharp (Baydin, Barak A Pearlmutter, and Siskind, 2016).

## Python

Because Python plays an important role in the ML community many popular ML frameworks are Python-based. However, the language's characteristics make it difficult to implement a high-performance AD-enabled ML framework in Python directly. The reference implementation of Python, CPython, has effectively no support for concurrency, and the interpreter is relatively slow. Moreover, its highly dynamic nature makes source transformation difficult (Tangent imposes several restrictions on the use of Python in order for it to perform ST). Python does not have built-in support for multidimensional arrays, which are only supported through third-party frameworks such as NumPy.

How to reconcile users' desire to work in Python because of its flexibility with the need for high performance and speed is an open question. ML frameworks have

i. https://github.com/axch/dysvunctional-language focused on metaprogramming and using C extensions, but other approaches are possible. For example, Cython (Behnel et al., 2011) is a superset of Python which compiles to Python modules, whereas Numba (Lam, Pitrou, and Seibert, 2015) can compile individual Python functions using LLVM.

## 11.3 Graph-based direct intermediate representation

We endeavor to combine several of the aforementioned techniques and insights from the compiler and AD literature in order to provide a flexible basis for an ML framework. This requires a well-tailored intermediate representation which avoids the pitfalls of previous methods, while keeping their strengths. Concretely, we propose an IR with the following properties:

**Graph based** Similar to Theano or TensorFlow, programs are represented as graphs. Graphs have the advantage of being easy to optimize and flexible about execution order, as operations that do not depend on each other in the graph may be executed in any order, or in parallel. Unlike Theano and TensorFlow, however, functions may be called recursively and they are first-class objects. Functions may be passed as parameters to other functions, or returned from a function and then called. A large variety of control flow constructs, ranging from simple loops to graph traversals, can be implemented using these capabilities. Other graph frameworks tend to implement only a few of these as specialized operators, such as Theano's `scan` or TensorFlow's `while`, leading to an IR which is both more complex and less powerful than the general one we are proposing. A general IR does require more work to transform and optimize in a provably correct way in the context of automatic differentiation, but this work only needs to be done once.

**Purely functional** Mutation and side effects are problematic for reverse mode AD, where the backward pass requires access to the unchanged intermediate variables from the forward pass. They also interact poorly with complex optimizations because of aliasing. Restricting our language to be purely functional therefore allows us to implement more robust AD and more advanced optimizations compared to imperative languages.

Note that Myia's intended use case is not the writing of efficient low-level kernels, which often requires fine-grained memory control. Similarly to, e.g., TensorFlow, the user can write efficient low-level kernels and their derivatives in a low-level language such as CUDA or XLA, and expose them to Myia as primitives.

**Closure representation** AD on functional languages involves storing the primal's intermediate results into closures which are then connected together to form the adjoint. It is therefore important to have a natural representation for closures. As in Thorin, we represent a function's graph's free variables as direct pointers to nodes that belong to other graphs, thereby creating an implicit nesting relationship between them (a graph $G_c$ is "nested" in $G_p$ if it points to a node in $G_p$, or to a graph nested in $G_p$, or to a node in a graph nested in $G_p$). This facilitates joint optimization of a closure with the functions it is nested in. Closures are also a great means for abstraction and a natural way to represent the methods of objects, so there is a concrete advantage in expressiveness from the user's perspective, which cannot be found in other frameworks.

**Strongly typed** In its canonical form, every node must be associated with a concrete type. This is important to maximize performance. This is also important in ML applications, because operations tend to be very costly and it is best to catch errors as early as possible. In addition to data types, there is also a need to infer other properties such as the dimensions of vectors and matrices so that we can guarantee that the inputs of all operations have compatible dimensions prior to executing them. Type and shape inference are more complex and powerful on our proposed IR than in dataflow graphs because of the need to support recursive calls and higher order functions.

### 11.3.1 IR specification

Concretely, our representation represents a function as a graph object with a list of parameter nodes and a single return node (multiple return values are supported through tuples). A node represents a function application and has an ordered list of incoming edges. The first incoming edge is a pointer to the function to apply, and the rest point to the arguments. Constants are represented as nodes with no incoming edges and a value field. Links between nodes are bidirectional, so that graphs can be traversed in either direction. Each non-constant node belongs to a single graph. See Figure 11.1 for a visual representation of the IR.

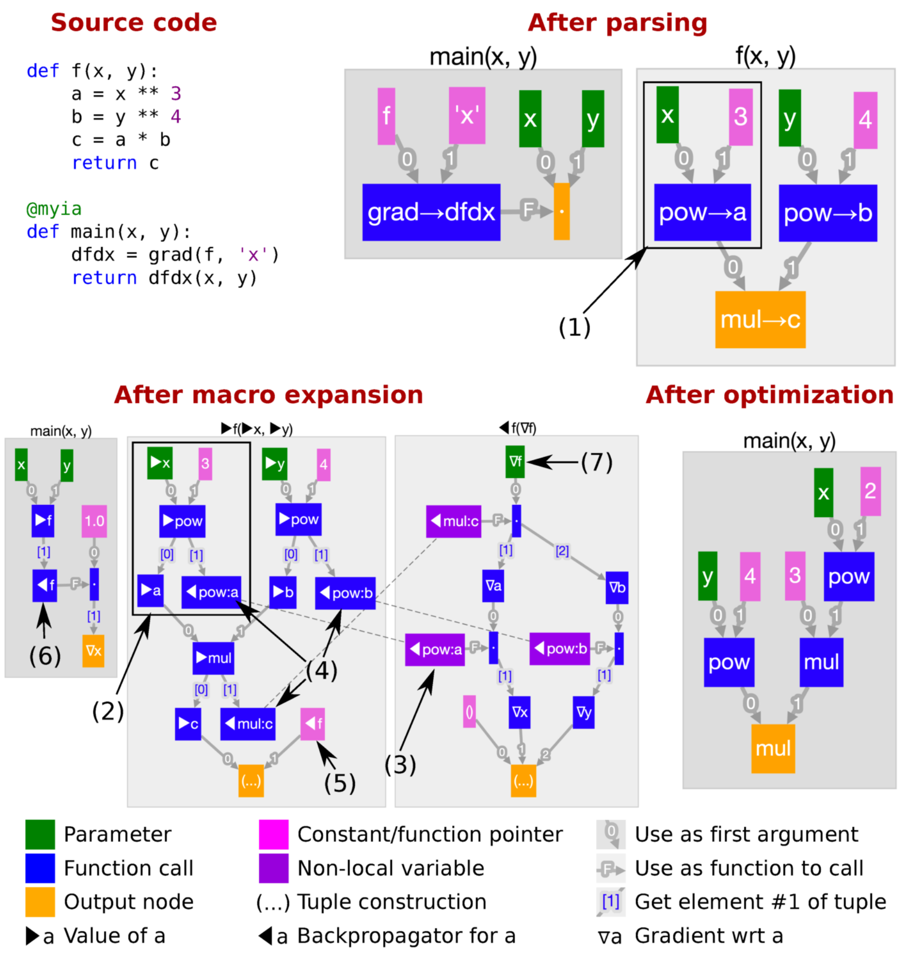
Compared to other representations, our representation is more expressive than dataflow graphs, and more flexible than SSA or CPS representations which tend to be rigid about execution order. It is closest to A-normal form (ANF, Flanagan et al., 1993), where every intermediate computation is assigned a unique name, but it is graphical rather than syntactic and therefore easier to manipulate algorithmically.

### 11.3.2 Source transformation

AD can be implemented for this IR using ST with a closure-based method. We closely follow the approach described in Barak A Pearlmutter and Siskind (2008). The transformed program constructs a chain of closures during the forward computation. These closures contain the adjoint code required to compute the derivatives along with the intermediate variables from the forward pass that are needed.

The transformation proceeds as follows: Each function call is transformed to return an additional value, which is a closure called the 'backpropagator'. The backpropagator computes the derivative with respect to the inputs given the derivatives with respect to the outputs. The backpropagators of primitives are known, whereas the backpropagators of user-defined functions can be easily constructed by calling the backpropagators of the function calls in the body in reverse order.

In order to ensure that our transformation can be applied again on the transformed program (so we can use reverse-over-reverse to compute second-order derivatives), it must be able to handle functions with free variables. To this end, each backpropagator will return the partial derivatives with respect to the inputs of the original function, as well as an ordered set of partial derivatives with respect to the free variables. The backpropagator of the function that built the closure is responsible for unpacking these partial derivatives so that it can add contributions to the free variables that belong to it, this unpacking being the adjoint of closure creation. Closures are first class functions: when given as inputs of other closures,


```python
def f(x, y):
    a = x ** 3
    b = y ** 4
    c = a * b
    return c

@myia
def main(x, y):
    dfdx = grad(f, 'x')
    return dfdx(x, y)
```

**Figure 11.1** – Transform of a simple Python program into Myia's representation. The program computes the gradient of `f` with respect to `x`. (1) identifies the part of the graph that implements `x ** 3`, or `pow(x, 3)`. After the `grad` macro is expanded, a new graph, ▶`f` is built. In that graph, `a = pow(x, 3)` becomes ▶`a`, ◀`pow` = ▶`pow(`▶`x, 3)` (2). The ▶`pow` operation thus returns two values instead of one, ▶`a` being equal to the original value `a`, and ◀`pow` being the *backpropagator* for this operation. That backpropagator is used in (3). It is applied on the gradient wrt the output, ∇`a`, and produces the gradient wrt the input ∇`x` (it also produces a gradient wrt the constant `3`, but that gradient is not used). (4) points to all the backpropagators created while executing ▶`f`. These are closures that retain pointers to all the information necessary to perform reverse mode AD. ◀`f`, the backpropagator we construct for `f`, is returned by ▶`f` (5), alongside the main result (notice that this mirrors the interface in (2)). ◀`f` retains pointers to all backpropagators in (4), as indicated by the dashed lines. ◀`f` is retrieved in (6), and immediately called with the value `1.0` for the parameter ∇`f` (7). In this context, ∇`f` corresponds to $\partial f/\partial f$, hence why we give it a value of one. After optimization, all functions and backpropagators end up being inlined. All unused computations are cut, and what remains is an expression for $\partial f/\partial x$ that is essentially identical to what one would have written by hand.

they are treated like any other input.

## 11.4 Myia

Myia is a functioning proof of concept of a toolchain that uses the proposed graph representation[i]. Myia performs type inference given the input types, and applies a series of optimizations such as inlining, common expression elimination, constant propagation, closure conversion, and algebraic simplifications. The final code can be executed using an interpreter, and we also implemented a prototype which compiles the straight-line parts of the graph using TVM (T. Chen, Moreau, et al., 2018).

### 11.4.1 Python front end

Due to Python's popularity in the ML community, we feel it is important to offer a front end in that language. Users can write models in a subset of Python 3.6 and have them compiled to our IR. This requirement is ostensibly at odds with our IR being pure and strongly typed, for Python is neither of these things. We solve that apparent contradiction by selecting a pure subset of Python, and running an advanced type inference algorithm on the functions the user asks to compile. In that sense, our approach is similar to that of Numba and Cython, or the recently introduced `@script` decorator in PyTorch[ii]. Functions that should be compiled with Myia are denoted using the `@myia` decorator, and can be freely mixed with Python code in the same file.

Most of Python's features, such as functions, conditionals, and loops, can readily be parsed into our functional representation. However, Python does include some statements such as index assignment (`x[i] = v`) and augmented assignment statements (`x += y`) which imply mutability. We currently forbid these statements in Myia, although it may be possible to support principled use of them in the future through techniques like uniqueness typing (Barendsen and Smetsers, 1993; Vries, Plasmeijer, and Abrahamson, 2007).

i. Code available at `https://github.com/mila-udem/myia`
ii. `https://pytorch.org/2018/05/02/road-to-1.0.html`

Myia uses Python's `inspect` module to parse the function into an abstract syntax tree (AST), and converts that AST into the graph representation we previously described. Source transformation as described in Section 11.3.2 is used to generate the code for derivatives. See Figure 11.1 for an illustration of how a Python function is parsed into the proposed IR, its adjoint program is created using ST, and finally optimized to produce an efficient derivative function.

### 11.4.2 Type inference

Python is a dynamically typed language, but for the sake of optimization and eager error reporting, it is important to be able to infer concrete types for all expressions. While it is possible to write optional type annotations in Python 3.6, they are not widely used in practice, and we wish to minimize the amount of work one has to do in order to port existing code to Myia.

When a Myia function is called, we use the types of the user-provided arguments as a starting point for type inference, which allows us to compile a specialized version of the function for these types. No type annotations are required, even when using higher order functions such as `map` or `grad`. Myia functions can be polymorphic: Myia will specialize each use of a function according to the input type signature for that call site. This means users can write highly dynamic programs just as they are used to in Python, and Myia will check them.

The inferrer operates on an untyped version of the IR. It can infer types as well as values (constant propagation) and shapes. Inference for other properties can easily be added in the future. The inferrer is implemented using coroutines: to infer a certain property through a certain primitive, one may write a coroutine (`async def` in Python) that asynchronously requests any number of properties from any number of nodes and combines the results using arbitrary logic.

### 11.4.3 Optimization

Reverse mode AD in Myia poses a few specific challenges for optimization that we have to tackle. As may be seen in Figure 11.1, the AD transform produces graphs that are substantially larger than the original source. These graphs typically contain many computations that are not necessary, such as gradients with respect to constants, and a lot of tuple packing and unpacking. These graphs can be simplified using inlining and local optimizations. Figure 11.1 demonstrates the resulting simplification.

## 11.5 Conclusion

In this work we examined the different approaches and techniques used in developing AD-enabled ML frameworks, drawing insights from functional languages, graph-based IRs, and AD. To address some of the shortcomings in existing frameworks, we propose a novel graph-based intermediate representation and describe a proof of concept toolchain called Myia to show its advantages.

The result is a system that can achieve performance similar to compiled frameworks such as TensorFlow, while providing the flexibility of OO frameworks such as PyTorch with e.g. support for recursion and higher-order functions.

We believe that as AD frameworks will slowly move towards being full-fledged languages and compilers, developers will benefit from building on many other ideas from these fields. For example, other techniques from functional languages that could be beneficial include the use of monads to handle random number generators, and using higher-order functions for kernel programming (similar to Tensor Comprehensions[i]).

i. https://facebookresearch.github.io/TensorComprehensions/

# 12 Prologue to Second Article

## 12.1 Article Details

**Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming**.

Van Merriënboer, Bart, Alexander B. Wiltschko, and Dan Moldovan. 2018. In *Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*, Montréal, Canada, December 2–8, 2018.

*Personal Contribution.*

The idea of applying SCT was due to Alex Wiltschko, the second author. As primary author, I architected and wrote the majority of the Tangent framework. I conceptualized and implemented the use of persistent arrays in AD.

## 12.2 Context

Tangent was motivated by similar considerations as the Myia project introduced in Chapter 10, but does so by adapting traditional tape-based SCT approaches to Python and ML workloads, which implies handling dynamic typing and array programming. It was also an exploration of the application of traditional AD techniques such as source code preprocessors to a deep learning setting with the aim of discovering application specific problems that have to be addressed.

## 12.3 Contributions

This work introduces Tangent, the first successful application of AD in the form of a source code preprocessor to a dynamically typed array programming language. Previous applications of SCT to dynamically typed languages such as Scheme took the form of compiler plugins or extensions (Siskind and Barak A. Pearlmutter, 2016; Barak A Pearlmutter and Siskind, 2008; Siskind and Barak A Pearlmutter, 2008). As such, Tangent addresses several problems specific to this setting involving gradient initialization, higher-order differentiation, and dynamic types.

As a result, Tangent is the first framework to successfully separate the reverse mode AD transformation entirely from the execution of Python code, allowing it to integrate into the wider Python ecosystem. Tangent focuses in particular on the CPython interpreter for the Python language, since array programming packages such as TensorFlow Eager and NumPy are not well supported in other Python implementations such as PyPy.

Tangent is also one of the first frameworks to propose the use of persistent arrays in a machine learning context. In the regular use case where a small part of a single array is repeatedly mutated, persistent arrays can lead to significant speedups. It must be noted that, like persistent data structures in general, persistent arrays can be significantly slower in some edge cases (e.g., when two versions of an array are updated in alternating fashion). This can be seen as trading runtime performance for lower memory usage. Since Tangent supports both regular and persistent arrays, the user is able to make decisions regarding these trade-offs.

## 12.4 Recent Developments

The recent developments in the space of AD frameworks for ML were discussed in Section 10.4.

In retrospect, the lazy evaluation of zero gradients used by Tangent is similar to the bold-faced zero used in Barak A Pearlmutter and Siskind (2008).

# 13 Tangent: AD using SCT for dynamically typed array programming

## 13.1 Introduction

Many applications in machine learning rely on gradient-based optimization, or at least the efficient calculation of derivatives of models expressed as computer programs. Researchers have a wide variety of tools from which they can choose, particularly if they are using the Python language (Paszke et al., 2017; Maclaurin, Duvenaud, and Adams, 2015; Tokui et al., 2015; Al-Rfou et al., 2016; Abadi et al., 2016). These tools can generally be characterized as trading off *research* or *production* use cases, and can be divided along these lines by whether they implement automatic differentiation using operator overloading (OO) or SCT. SCT affords more opportunities for whole-program optimization, while OO makes it easier to support convenient syntax in Python, like data-dependent control flow, or advanced features such as custom partial derivatives. We show here that it is possible to offer the programming flexibility usually thought to be exclusive to OO-based tools in an SCT framework.

Tangent is the first AD framework using SCT in a dynamically typed language. We produce efficient derivatives using a novel combination of multiple dispatch, lazy evaluation, and static optimizations. Further, Tangent has mutable multidimensional arrays as first class objects, implemented using persistent data structures for performance in the context of reverse mode AD. By operating directly on Python source code, Tangent is able to achieve a separation of concerns that other AD libraries do not. Specifically, we achieve compositionality with tools in the Python ecosystem, such as debuggers, profilers and other compilers. Tangent makes it easy and efficient to express machine learning models, and is open source[i].

i. Source code and documentation available at https://github.com/google/tangent

## 13.2 Background

Automatic differentiation (AD) is a set of techniques to evaluate derivatives of mathematical functions defined as programs (Griewank and Walther, 2008), and is heavily used in machine learning (Baydin, Barak A. Pearlmutter, et al., 2018). It is based on the insight that the chain rule can be applied to the elementary arithmetic operations (primitives) performed by the program. This allows derivatives to be calculated up to machine precision (Naumann, 2012) with only a constant overhead per operation. AD is different from symbolic differentiation (which applies to mathematical expressions instead of programs) and numerical differentiation (where the gradient is approximated using finite differences).

For multidimensional functions, $f : \mathbb{R}^n \to \mathbb{R}^m$, where $f$ is a composition of primitives with known derivatives, the application of the chain rule results in a series of matrix-vector multiplications involving the primitives' Jacobians and partial derivatives of intermediate values. The order in which these multiplications are evaluated determines the runtime complexity. Forward-mode AD evaluates the chain rule from inside to outside and is efficient for functions where $m > n$. The implementation of forward mode is relatively straightforward, since the partial derivatives are evaluated in step with the primitives. Forward mode is commonly implemented by replacing numbers with *dual numbers*, which can be interpreted as a variable's value along with its partial derivative with respect to one of the inputs. Reverse-mode AD, where the chain rule is evaluated from outside to inside, is more efficient in the case where $n > m$. Reverse mode is more complex to implement because evaluation of the partial derivatives requires reversing the execution order of the original program. This reversal gives rise to a non-local program transformation where the beginning of the original program interacts with the generated derivative program.

Two methods of implementing reverse-mode AD are commonly distinguished: operator overloading (OO) and source code transformation (SCT). In the OO approach primitives are overloaded so that at runtime each numerical operation is logged onto a tape (a linear trace) along with its inputs. The chain rule can then be evaluated by walking this tape backward. SCT, on the other hand, explicitly transforms the original program (primal) prior to execution to produce a separate derivative function (adjoint) whose control flow is the reverse of the original program. Both approaches have different implementation, performance, and usability trade-offs (C. H. Bischof and Bücker, 2000).

OO is easier to implement and since it only requires tracing, it naturally supports all the features of the host language such as higher-order functions, recursion, and classes. If the control flow of the program is data dependent, the function must be retraced for each function call, which can cause significant overhead when the runtime of the primitives is small compared to the cost of tracing. Since the adjoint program is run by a separate 'derivative interpreter' (the algorithm that walks the tape in reverse), there is no adjoint program that can be inspected, optimized or compiled.

SCT is harder to implement, since it requires tooling to transform intermediate representations of computer programs. Further, the AD tool must explicitly support all of the features of the host language, including function calls, loops, classes, etc. If a language feature is not explicitly handled by the AD system, the user cannot take derivatives of code using those features. For some languages like C and C++ this requires a separate toolchain, but reflective languages such as Lisp and Python contain the necessary tools to capture, transform, and output program representations. The advantage of SCT is that there is no runtime overhead, and that generated derivative code can be statically analyzed and optimized.

## 13.3 Prior work

AD packages using either approach have long existed for, e.g., C, C++, Fortran, (see Baydin, Barak A. Pearlmutter, et al., 2018, for an overview) and have been used in fields such as computational fluid dynamics, atmospheric sciences, and astronomy. In the machine learning community different needs have led to the development of a separate set of tools. In particular, the community has a strong attachment to Python and its models rely heavily on multidimensional arrays.

Theano (Al-Rfou et al., 2016) and TensorFlow (Abadi et al., 2016) are two popular machine learning frameworks with support for SCT AD. Although Python-based, they do not perform AD on the Python code. Instead, Python is used as a metaprogramming language to define a dataflow graph (computation graph) on which SCT is performed. Since these dataflow graphs only operate on immutable values and do not have function calls or lexical scoping, the AD logic is simplified. The same graph representation is then used for static analysis, optimizations, and code generation.

OO has been used to implement AD in Python in packages such as Autograd (Maclaurin, Duvenaud, and Adams, 2015), Chainer (Tokui et al., 2015), and PyTorch (Paszke et al., 2017).

Although OO frameworks are easier to implement, their runtime performance falls short of that of frameworks using SCT for workloads that do not spend most of their time in hand-optimized compute primitives. On the other hand, existing frameworks that use SCT require the user to metaprogram computation graphs, significantly complicating the definition of ML models. Tangent applies SCT directly on the Python language in order to combine the performance achieved by SCT with the usability of programming directly in Python.

## 13.4 Features

Tangent supports reverse mode and forward mode, as well as function calls, loops, and conditionals. Higher-order derivatives are supported, and reverse and forward mode can readily be combined. To our knowledge, Tangent is the first SCT-based AD system for Python and moreover, it is the first SCT-based AD system for a dynamically typed language. As a consequence of performing SCT directly on the Python source code, the generated programs can be run, inspected, profiled, and debugged with standard Python tools. Tangent supports array programming on both CPU and GPU through the NumPy (Oliphant, 2006) and TensorFlow Eager libraries. A modular design makes it possible to extend Tangent to support other numeric libraries.

The ability to write code directly in Python makes Tangent less verbose and more idiomatic than the metaprogramming approach used by Theano and Tensorflow (see Listing 13.1a). Moreover, the metaprogrammed code requires a separate compiler and/or runtime, separate debugging tools, etc.

The OO approach can be problematic for debugging and usability as well as performance (see Listing 13.2). When an adjoint function `grad(f)` is called, the

```
x = tf.placeholder(tf.float32)
y = x * x
dx, = tf.gradients(y, x)

with tf.Session() as sess:
    dx_ = sess.run(
        dx, feed_dict={x: 3})
```

**(a)** TensorFlow requires the programmer to define the variable `x` as part of the dataflow graph. After the program (dataflow graph) has been constructed, its evaluation must be triggered by creating a session and providing values for the arguments.

```
def f(x):
    return x * x

df = grad(f)
dx = df(3)
```

**(b)** Tangent and libraries such as Autograd allow the user to write pure Python.

**Listing 13.1** – Comparison between metaprogramming and direct programming approaches.

function `f` is executed with non-standard semantics, since each function and operator has been overloaded to log onto a tape, after which the tape is walked in reverse using a loop that is internal to the framework. This means that each function call incurs tracing overhead, and errors that occur during execution will potentially have tracebacks involving tracing logic that can be hard for a user to decipher.

```
def f(x):
    while x < 10000:
        x = x + 1
    return x
```

**Listing 13.2** – In the case that `x` is a scalar, this trivial program and its derivative contain a tight loop. Since it does not require tracing, Tangent's derivative of this function is approximately 30% faster than PyTorch's, even though PyTorch is given type information about `x` whereas Tangent's derivative is dynamically typed.

```
# Generated gradient function
def dfdx(x, by=1.0):
    # Grad of: y = x * x
    _bx = tangent.unbroadcast(by * x, x)
    _bx2 = tangent.unbroadcast(by * x, x)
    bx = _bx
    bx = tangent.add_grad(bx, _bx2)
    return bx
```

**Listing 13.3** – Source code of the gradient of `def f(x): return x * x` in Tangent. The `unbroadcast` function is responsible for reversing the broadcasting performed by NumPy when performing element-wise operations on differently-sized multidimensional arrays.

The adjoint code generated by Tangent is regular Python (see Listing 13.3), which means that it can be debugged using standard debuggers such as `pdb`, profiled using, e.g., `line_profiler`, optimized by JIT compilers such as Numba (Lam, Pitrou, and Seibert, 2015) and Pythran (Guelton et al., 2015). The adjoint code can readily be inspected by users, and Tangent tries to ensure that is human-readable and commented, which is useful for debugging as well as for didactic purposes.

Unlike most existing ML frameworks, arrays in Tangent are mutable without incurring unnecessary performance loss (see Section 13.5.4 for implementation details).

### 13.4.1 Backward pass inlining

Many algorithms use approximations or modifications of the gradient. For example, for performance reasons recurrent neural networks (RNNs) are often trained using truncated backpropagation through time (Ronald J Williams and Peng, 1990) (TBPTT) and/or gradient clipping (Razvan Pascanu, Mikolov, and Y. Bengio, 2013). In other cases, custom gradients are used to train models with discontinuous functions (e.g. straight-through estimators) or for many other applications (Y. Bengio, Léonard, and Courville, 2013; Ganin et al., 2016; Oord, Vinyals, and Kavukcuoglu, 2017; Heess et al., 2015; Jang, Gu, and Poole, 2017; Nøkland, 2016; Lillicrap et al., 2016). A user might also be interested in accessing the values of gradients for logging or debugging.

Existing AD frameworks support this functionality by allowing the user to define custom adjoints for functions. Tangent provides this functionality as well, but uses Python's context manager syntax to introduce a second, novel way of allowing the user to inject arbitrary code into the gradient computation (see Listing 13.4). We believe this syntax provides a more succinct and readable way of modifying the adjoint code in many cases.

## 13.5 Implementation

Tangent uses Python's built-in machinery to inspect and transform the abstract syntax tree (AST) of parsed source code. AD can be performed line by line Griewank and Walther, 2008, Proposition 4.2. Hence, for each piece of supported Python syntax we have implemented a rule indicating how to rewrite an AST node into its primal and adjoint. We have defined adjoints for e.g. mathematical operators, function calls to NumPy methods, and constructs such as if-statements and

```python
# Original function
def f(x):
    with insert_grad_of(x) as dx:
        if dx > 10:
            print('Clipping', dx)
            dx = 10
    return x * x
```

```python
# Generated gradient function
def dfdx(x, bx_times_x=1.0):
    x_times_x = x * x
    # Grad of: dx = 10
    _bx = tangent.unbroadcast(
        bx_times_x * x, x)
    _bx2 = tangent.unbroadcast(
        bx_times_x * x, x)
    bx = _bx
    bx = tangent.add_grad(bx, _bx2)
    # Inserted code
    if bx > 10:
        print('Clipping', bx)
        bx = 10
    return bx
```

**Listing 13.4** – Gradient clipping implemented using Tangent. The code inside of the context manager is inserted directly into the derivative function.

for-loops. The adjoints are defined using a custom template programming syntax (see Listing 13.5) which makes it easy for users to add new or custom derivatives.

```python
# Templates are Python functions
@adjoint(numpy.multiply)
def adjoint_multiply(z, x, y):
    d[x] = y * d[z]
    d[y] = x * d[z]
# If the primal contains...
c = numpy.multiply(a, b)
```

```python
# ...Tangent will expand the template...
new_ast = tangent.template.replace(
    adjoint_multiply,
    z='c', x='a', y='b')
# ...generating the following adjoint
b_a = b * b_c
b_b = a * b_c
```

**Listing 13.5** – Tangent's source generation uses templating. The template takes the form of a Python function which is parsed into its AST. The variable names in the AST are substituted and variables for the partial derivatives are constructed, before the AST is inserted into the code of the adjoint function.

Generated derivative code is constructed using the built-in Python AST. The alternative program representations are Python bytecode, which changes across Python versions, and a formatting-aware AST used in the Python 2-to-3 conversion tool, 2to3, which has little tooling and is more cumbersome to use. We acquire and manipulate the Python AST with the inspect and ast modules from the standard library, and standardize small differences between the Python 2 and Python 3 AST with gast and use astor to invert ASTs into readable source code.

To support dynamic typing and array programming while maintaining efficiency, Tangent relies on a novel combination of multiple dispatch, lazy evaluation, persistent data structures, and static optimizations.

### 13.5.1 Multiple dispatch

Python is a dynamic language which uses dynamic typing, late binding and operator overloading. These fundamental features of the language make it impossible to determine ahead of time how a statement will be executed, which means it is impossible to determine ahead of time what the adjoint program should be. Instead of enforcing static types (for example by using type annotations and MyPy[i]), Tangent embraces late binding and generates adjoints that will use the runtime types to determine what derivative computation to execute.

For example, `x * y` where `x` and `y` are scalars at runtime results in a scalar multiplication. However, if either of the two variables is a NumPy `ndarray` object, the multiplication operator is dispatched to perform broadcasting followed by element-wise multiplication. The adjoint of this operation requires summing over the broadcasted axes. Tangent will generate code that uses type checking to ensure that the correct adjoint calculation is performed based on the runtime types.

Similarly, the initialization and addition of gradients cannot be generated statically. We introduce `add_grad` and `init_grad` operators which use multiple dispatch. For example, `init_grad(x)` will return `0` if `x` is a scalar, but will return `numpy.zeros_like(x)` if `x` is an `ndarray`.

### 13.5.2 Lazy evaluation

A common performance bottleneck in the context of AD and array programming is that initializing the gradient of a large array results in allocating a large zero array. When gradients are accumulated later on this large array of zeros is added to a partial gradient, which is effectively a no-op. In general, the gradient initialization and addition might happen in different functions, making it non-trivial to statically optimize this case. To address this issue, Tangent lazily initializes gradients: Instead of allocating an array of zeros, Tangent returns a special `ZeroGrad` object. The `add_grad` operator uses multiple dispatch to return the other argument when either argument is of the type `ZeroGrad`.

i. http://mypy-lang.org/

### 13.5.3 Static optimizations

When constructing the adjoint of a function, some of the code of the forward pass might become dead code. The opportunity for removing unused code only grows when taking higher order derivatives. One of the advantages of SCT is that the resulting code can be optimized by an optimizing compiler whose dead code elimination (DCE) pass would address this problem. However, Python is an interpreted language, and very few optimizations are applied before its execution. For this reason, Tangent includes a small Python optimizing compiler toolchain which constructs a control-flow graph (CFG) on which forward dataflow analysis is performed. Tangent uses this to perform dead code elimination on generated adjoints. The same machinery is used to perform algebraic simplifications and constant propagation. Note that although these optimizations are hard to perform on Python in general, we can exploit the fact that Tangent operates on a more limited subset of Python which is more amenable to analysis (see Section 13.6 for details).

Note that these optimizations are aimed at removing dead code or simplifying trivial expressions (such as multiplication by 1) generated by the AD algorithm. Unlike frameworks such as XLA and TVM (T. Chen, Moreau, et al., 2018), we expressly do not attempt to optimize the numerical kernels themselves. Since Tangent outputs regular Python code, functions can be passed to an optimizing Python compiler such as Numba for this purpose.

A central problem in reverse mode AD is that intermediate values are required to be kept alive after they go out of scope since they might be needed by their adjoint. For example, if a function contains `z = x * y` the variables `x` and `y` cannot be deleted after the function returns since the backward pass requires their values to calculate `dx = dz * y` and `dy = dz * x`. Tangent, like most SCT frameworks, uses a global stack (tape) to store intermediate variables on in order to ensure they are kept alive. Hence, before the function returns, `x` and `y` are pushed onto this stack and they will be popped off the stack right before the adjoint calculation. Note that the trace used in OO is also referred to as a tape, the difference being that the tape in OO stores not only the intermediate variables, but also the operations performed.

In order to perform DCE effectively on the generated code, our dataflow analysis follows variable uses through their respective pushes (reads) and pops (writes) in

```python
# Raw generated code
def dfdx(x, by=1.0):
    # Initialize the tape
    _stack = tangent.Stack()
    y = None
    # Beginning of forward pass
    tangent.push(_stack, y, '_19429e9f')
    y = x
    # Beginning of backward pass
    _y = y
    # Grad of: y = x
    y = tangent.pop(_stack, '_19429e9f')
    _bx = tangent.copy(by)
    by = tangent.init_grad(y)
    bx = _bx
    return bx
```

```python
# Optimized generated code
def dfdx(x, by=1.0):
    y = x
    # Grad of: y = x
    _bx = tangent.copy(by)
    bx = _bx
    return bx
```

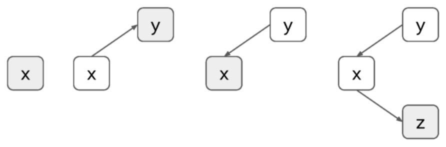
**Listing 13.6** – A simple example of Tangent's optimization capabilities as applied to the gradient function of `def f(x): y = x; return y`. Note that the original transformation includes the writing and reading of `y` to and from the tape, and contains dead code in initializing the gradient of `y` which is never returned. Tangent's dataflow analysis is able to match the tape reads and writes and understands that the value of `y` is the same, allowing it to aggressively optimize the function.

the primal and adjoint code. This highlights the close interaction required between the optimizing compiler and the AD machinery for maximum performance. To enable the dataflow analysis to match reads and writes they are augmented in the source code with unique hashes (see Listing 13.6).

### 13.5.4 Persistent data structures

AD is problematic in the context of mutability. If `x` and `y` from the previous example are mutable arrays, their value could have been changed by an in-place operation, resulting in an incorrect adjoint calculation. For this reason, arrays are in principle immutable in existing AD frameworks for ML such as TensorFlow, Autograd, and Theano. PyTorch allows users to mutate arrays if they can guarantee that the previous version will not be needed by the backward pass, otherwise an error will be thrown. This makes algorithms which rely on mutating arrays in place inefficient and difficult to express.

Persistent data structures (Driscoll et al., 1989) are data structures that are



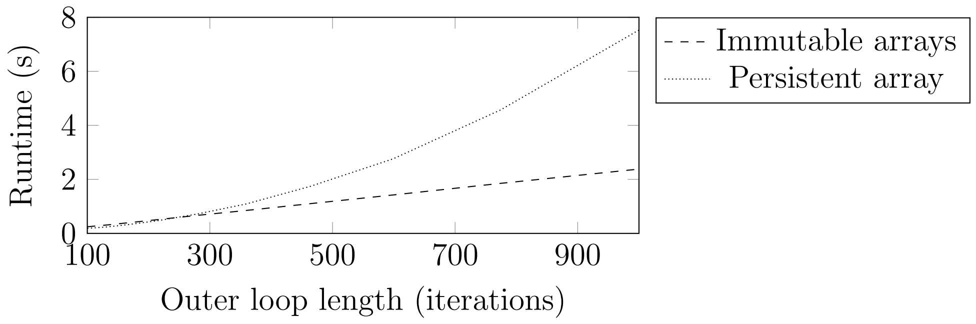
```python
# Create handle to original version
x_copy = copy.copy(x)
# Create new node y
x[i] = v
# Restore original version
print(x_copy)
# Modify old version to create z
x_copy[i] = v
```

**Listing 13.7** – Illustration of the persistent array data structure. Root nodes are gray and edges represent deltas.

effectively immutable: They are mutable data structures where all previous versions can be accessed. Unlike truly immutable data structures, different versions of persistent data structures may share memory and hence can be more memory-efficient, although accessing previous versions might require extra work. Functional languages often use persistent data structures for implementing, e.g., linked lists, trees, stacks. We note that persistent array data structures can be used to support mutability efficiently in the context of AD.

By default, Tangent handles index assignments (`x[i] = y`) efficiently by copying only the affected subarray `x[i]` onto the tape. To deal with mutability in full generality Tangent also introduces a persistent array data structure with support for index assignment as well as inserting and deleting rows at the end. Each time the array is modified, the delta with the previous version is stored. Since previous versions can be modified as well, this results in a version tree where the root contains the current array in memory and other versions of the array are represented by leaf nodes (see Listing 13.7). If the user attempts to read a specific version of the array, the deltas on the path from the leaf to the root of the version tree are applied in order to reconstruct the array in memory. When the handle to a specific array version is destroyed, the deltas are garbage collected. We note that in the context of reverse mode AD the most common access pattern is a linear series of mutations during the forward pass, followed by accessing the arrays in reverse order during the backward pass. In this case, our persistent array results in optimal memory and time complexity.

As an example, consider the double loop from Listing 13.8, which is a simplification of a neural lattice language model from Buckman and Neubig (2018). Given an outer loop with $n$ iterations, an inner loop with $m$ iterations, and a vector



```
def f(x, OUTER):
    r = numpy.zeros(DIM)
    for _ in range(OUTER):
        x = append(x, r)
        for _ in range(INNER):
            y = numpy.add(x[-1], 1.)
            x = setitem(x, -1, y)
    return numpy.mean(x)
```

**Listing 13.8** – Runtime for a simplified version of a lattice language model with dimension 2000 and inner loop of 15 iterations. Results are an average of 10 runs.

dimensionality of $d$, the complexity of this algorithm is $O(n^2md)$ for immutable arrays. When using regular NumPy arrays, Tangent will intelligently handle index assignments and only copy the affected subarray onto the tape, bringing the complexity down to $O(n^2d + ndm)$. When a persistent array is used, the complexity goes down to $O(ndm)$. When using persistent arrays, Tangent's runtime and memory complexity is determined only by the amount of data that is inserted, deleted or modified. In contrast, most libraries will have the gradient's runtime and memory complexity grow linearly with the number of times an array is modified. The technique described in Rae et al. (2016) for memory-augmented networks is also a special case of using persistent arrays.

## 13.6 Limitations

SCT relies on the ability to perform dataflow analysis to determine which variables are 'active' i.e. which variables affect the output of the function whose derivative we are constructing. To this end, Tangent is restricted to a subset of Python where these analyses are feasible. Note that these restrictions only apply to statements involving active variables.

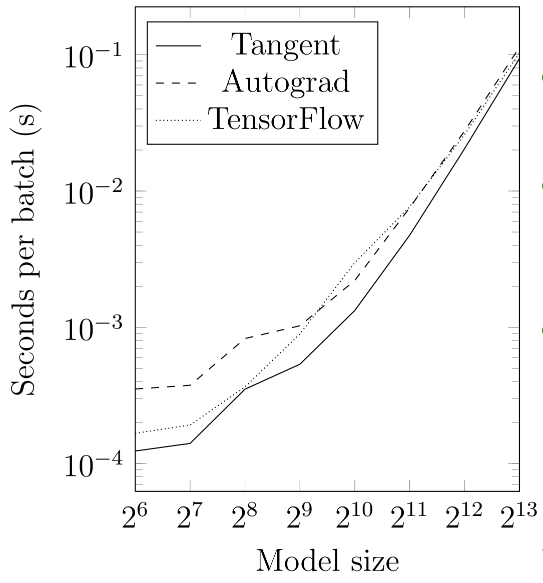
1. Functions that modify a variable in-place must also return that variable. Hence, `numpy.add(a, b, out=a)` is disallowed and should be written as `a = numpy.add(a, b)`. Likewise, a user-defined function that modifies `x` in-place using `x[i] = v`, must have `x` as a returned value.
2. Closures are not supported since closures with free variable references lead to a problem sometimes referred to as 'perturbation confusion' (Siskind and Barak A Pearlmutter, 2005), which is non-trivial to address. Additionally, Python uses lexical, not dynamic scoping, so writing adjoint values into the same scope where primal values are read is not straightforward.
3. Object methods are not currently supported because it is non-obvious what the partial derivative with respect to a member variable is.
4. In order to perform AD, the function and its source code must be resolvable at the time that the AD transformation is applied. This means that higher-order functions and nested function definitions are not supported. Tangent could apply additional AD passes at runtime to avoid this limitation.
5. Some Python syntax is not (yet) supported[i] e.g. `try` and `except` statements, as well as `break` and `continue`.

If the return value of a function is not used, it is assumed that its inputs were unchanged. This allows statements such as `print(numpy.mean(x))` to be used without interfering with the AD transformation.

## 13.7 Performance

Tangent was not designed with raw performance in mind. Instead, it intends to strike a balance between usability and good software design practices, while

i. For an up to date overview of supported AST nodes please refer to the code in `tangent/fence.py`.



```
def logsumexp(x):
    return numpy.log(numpy.sum(numpy.exp(x),
                                axis=-1, keepdims=True))

def logsoftmax(logits):
    return logits - logsumexp(logits)

def softmax_xent(logits, y):
    return -numpy.sum(
        logsoftmax(logits) * y, axis=-1)

def mlp(x, w1, b1, wout, bout, label):
    h1 = numpy.tanh(numpy.dot(x, w1) + b1)
    out = numpy.dot(h1, wout) + bout
    loss = numpy.mean(softmax_xent(out,label))
    return loss

autograd_dmlp = autograd.multigrad(
    mlp, argnums=(1, 2, 3, 4))
tangent_dmlp = tangent.grad(
    mlp, wrt=(1, 2, 3, 4))
```

**Listing 13.9** – Runtime for a simple feedforward neural network with a single hidden layer. We vary the input size and hidden layer size, which are set to the same value. The reported runtime is averaged over 50 runs with a batch size of 16. Run on a Xeon E5-1650 v3 @ 3.5 GHz, 64GB of RAM, with Ubuntu 14.04 on Python 2.7 with MKL. Note that for sufficiently large models the runtime of the numerical kernels dominates, which means that the frameworks have similar runtimes irrespective of their AD implementation.

exploring the feasibility and implementation details of applying SCT to dynamically typed languages. That said, Tangent's lack of runtime overhead combined with static optimizations and lazy gradient initialization means that its runtime performance is competitive with existing frameworks (see Listing 13.9).

## 13.8 Conclusion

In this work we introduced the AD library Tangent. Tangent is the first application of source-code transformation on a dynamically typed language such as Python. It uses several novel approaches, such as persistent data structures and lazy evaluation to ensure good performance. Machine learning models are natural and easy to express and debug in Tangent using many features that are not available in other frameworks e.g. mutable arrays, inspectable derivative code, and modifying gradients by injecting arbitrary code in the backward pass.

We believe Tangent is an important step on the path to fully general differentiable programming. Instead of an ML-framework, Tangent can be seen as the addition of the gradient operator to the Python language, without the need for metaprogramming or separate derivative interpreters (OO). This means that the user can write normal Python code while the entire Python ecosystem including debuggers, profilers, and introspection capabilities, become part of the ML toolkit. This allows users to express models more naturally and debug them more easily.

## Appendix: Performance

In the performance comparison between Tangent, Autograd, and TensorFlow Eager, it should be considered that the performance of a machine learning framework consists of several components.

**Numerical kernels** The actual numerical computation is performed by third-party libraries. TensorFlow Eager uses the Eigen numerical library, whereas Tangent and Autograd are linked to BLAS libraries through NumPy. The runtime of these kernels will dominate for large models with relatively few primitives.

**Overhead** A variety of implementation details can impact the performance. For example, frameworks that are written using CPython's C-API can have significant overhead if they repeatedly release and acquire the global interpreter lock (GIL). For very small models, this cost can easily dominate. On the other hand, libraries such as TensorFlow Eager can perform the backward pass entirely outside of the Python interpreter. In the case of long-running models, this can lead to significant speedups.

**AD implementation** Naive AD implementations can display bad performance, e.g., when explicitly initializing zero arrays, not reusing buffers of partial derivatives, or having inefficient gradients of primitives defined.

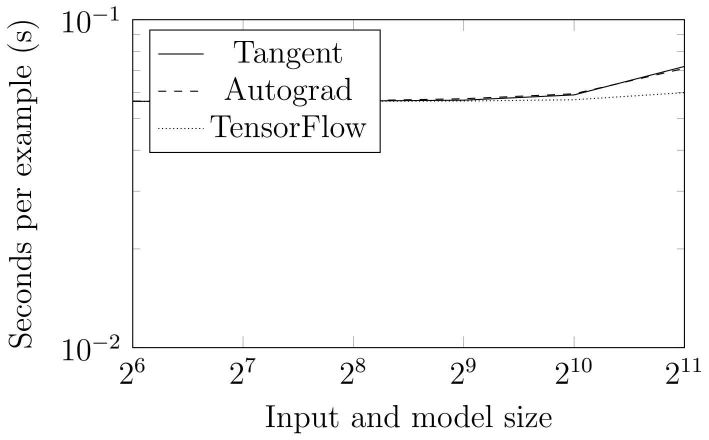


**Figure 13.1** – Runtime for a simple Elman RNN where the network receives a single input at time step $t = 1$, and produces a single softmax prediction at time step $t = 128$. The benchmark was run on a MacBook Pro (15 inch, 2017 model) with a 2.8 GHz Intel Core i7 using a batch size of 16 and the OpenBLAS library. Results were averaged over 10 runs.

When considering Tangent's performance, we are primarily interested in showing that its AD transformation does not give rise to unnecessary slowdowns. The original benchmarks in the paper highlight that this is indeed the case.

Additional benchmarks show that these performance results should not be interpreted as Tangent consistently outperforming other frameworks. In Figure 13.1 we can see that for larger models TensorFlow Eager outperforms Autograd and Tangent, which perform identically. This could be because Eigen outperforms the OpenBLAS library in this case, or possibly because TensorFlow Eager's execution engine in C++ is faster than the Python-based backward pass in Tangent and Autograd.

The results in Figures 13.1 and 13.2 suggest that for models where the numerical kernels do not dominate, the three frameworks can often perform identically.

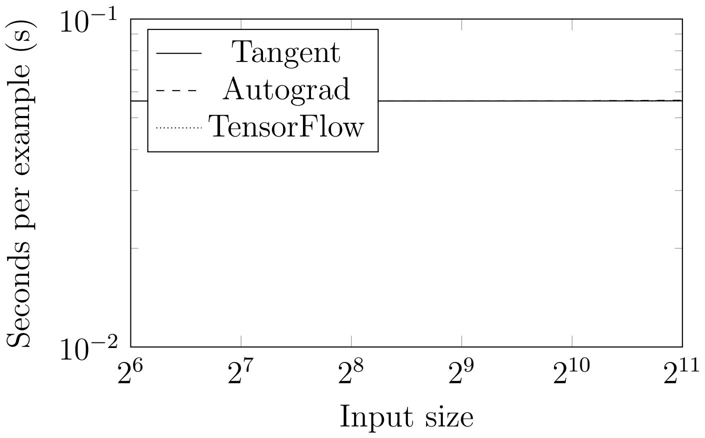


**Figure 13.2** – Runtime for a simple autoencoder using an L2 loss and a hidden dimension of 16. The benchmark was run on a MacBook Pro (15 inch, 2017 model) with a 2.8 GHz Intel Core i7 using a batch size of 16 and the OpenBLAS library. Results were averaged over 10 runs.

# 14 Discussion

The first set of articles presented in this thesis represent a natural scientific progression.

The first article squarely aims to push the boundaries of neural networks on the task of machine translation. This endeavor was very successful. Neural networks are now the industry standard for machine translation. Moreover, it led to the development of new architectures such as sequence-to-sequence models and gated recurrent units. Deep learning has seen several other endeavors which aim solely to achieve state-of-the-art performance on specific tasks such as ImageNet (Krizhevsky, Sutskever, and Geoffrey E Hinton, 2012) or Go (Silver et al., 2016).

However, achieving good performance on a specific task is engineering rather than science if we do not ask the question 'Why?' The second article in this thesis asks this question and aims to improve our understanding of the behaviour and limitations of sequence-to-sequence models. We should aim to use this understanding not only to improve our models' performance on specific tasks, but also to simplify our models and make them more generally applicable. The third article seeks to achieve exactly this by removing some of the common tokenization preproccessing steps used in language and translation models.

Much room remains for future research in machine translation and language modelling. Neural language models still have difficulty dealing with low frequency words and long-range dependencies. Moreover, the application of sequence models in different domains (e.g., gene sequences) represents a new set of challenges in terms of sequence length and tokenization that will be exciting to explore.

As discussed in the first article, many state-of-the-art neural network models are computationally intensive. To make training of these models feasible, a wide range of highly specialized kernels, frameworks, and software packages has been developed. For example, our machine translation research led us to develop the GroundHog library[i] and subsequently the Blocks and Fuel libraries (Van Merriën-

i. https://github.com/lisa-groundhog/GroundHog boer et al., 2015). In general, the need for fast computation in deep learning has resulted in the development of highly specialized tools. However, this can be a hindrance to the development of novel models: If a model cannot readily be expressed in an existing framework, a significant amount of low-level programming can be required to implement a model efficiently. The third article in this thesis is an example of this.

The desire to develop a set of tools which is simultaneously performant, usable, and general purpose enough for cutting-edge machine learning research is what drives the research presented in the second half of this thesis. A lot of work remains in this space, as deep learning presents a unique combination of requirements. Practitioners desire high-level interfaces in dynamic languages such as Python to be able to iterate quickly over models, but simultaneously want their code to be executed efficiently on accelerators. In addition, machine learning requires full program transformations such as automatic differentiation and vectorization. Solutions for many of these components exist, but a single stack that integrates all of these parts has yet to emerge.

## Acknowledgments

I would like to thank my advisor, prof. Yoshua Bengio, whose passion for science and research is an inspiration, and who supported my research even as I changed research directions repeatedly.

Thanks also goes to my parents, Jeroen van Merriënboer and Ankie Hersbach, and my siblings who have always supported and encouraged me in my academic studies. I would also like to thank my partner, Julio Méndez, for being patient and supporting throughout even the most stressful periods of my studies.

This thesis also would not have been possible without my collaborators, my internship hosts, my thesis committee, and the many students and researchers I crossed paths and had fruitful discussions with at Mila, Facebook, Twitter and Google.

# Bibliography

Abadi, Martín et al. (Nov. 2016). "TensorFlow: A System for Large-Scale Machine Learning." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, Georgia, USA, Nov. 2–4, 2016), pp. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Akidau, Tyler et al. (Aug. 2015). "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12. Ed. by Chen Li and Volker Markl, pp. 1792–1803. URL: http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf.

Auli, Michael et al. (Oct. 2013). "Joint Language and Translation Modeling with Recurrent Neural Networks". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA). Ed. by David Yarowsky et al., pp. 1044–1054. URL: http://aclweb.org/anthology/D13-1106.

Axelrod, Amittai, Xiaodong He, and Jianfeng Gao (2011). "Domain Adaptation via Pseudo In-Domain Data Selection". In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing* (Edinburgh, Scotland, UK, July 27–31, 2011). Ed. by Regina Barzilay and Mark Johnson, pp. 355–362. URL: http://aclweb.org/anthology/D11-1033.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton (2016). *Layer normalization*. arXiv: 1607.06450.

Bahdanau, Dzmitry, KyungHyun Cho, and Yoshua Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *Proceedings of the 2015 International Conference on Learning Representations* (San Diego, California, USA, May 7–9, 2015). arXiv: 1409.0473.

Barendsen, Erik and Sjaak Smetsers (1993). "Conventional and uniqueness typing in graph rewrite systems". In: *Proceedings of the 13th Conference on Foun-*

*dations of Software Technology and Theoretical Computer Science* (Bombay, India, Dec. 15–17, 1993). Ed. by Rudrapatna K. Shyamasundar. Berlin and Heidelberg, pp. 41–51. DOI: 10.1007/3-540-57529-4_42.

Bastien, Frédéric et al. (2012). *Theano: new features and speed improvements*. Presented at the Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop. arXiv: 1211.5590.

Baydin, Atılım Güneş, Barak A. Pearlmutter, et al. (Apr. 2018). "Automatic differentiation in machine learning: a survey". In: *Journal of Machine Learning Research* 18.153, pp. 1–43. URL: http://jmlr.org/papers/v18/17-468.html.

Baydin, Atılım Güneş, Barak A Pearlmutter, and Jeffrey Mark Siskind (2016). *DiffSharp: An AD Library for .NET Languages*. Presented at the 7th International Conference on Algorithmic Differentiation. arXiv: 1611.03423.

Behnel, Stefan et al. (Mar.–Apr. 2011). "Cython: The best of both worlds". In: *Computing in Science & Engineering* 13.2, pp. 31–39. DOI: 10.1109/MCSE.2010.118.

Bell, Bradley M (2003). *CppAD: a package for C++ algorithmic differentiation*. URL: https://coin-or.github.io/CppAD/.

Bengio, Samy et al. (2015). "Scheduled sampling for sequence prediction with recurrent neural networks". In: *Advances in Neural Information Processing Systems 28* (Montréal, Canada, Dec. 7–12, 2015). Ed. by Corinna Cortes et al., pp. 1171–1179. URL: https://papers.nips.cc/paper/5956-scheduled-sampling-for-sequence-prediction-with-recurrent-neural-networks.

Bengio, Yoshua (2009). "Learning Deep Architectures for AI". In: *Foundations and Trends in Machine Learning* 2.1, pp. 1–127. DOI: 10.1561/2200000006.

Bengio, Yoshua, Nicolas Boulanger-Lewandowski, and Razvan Pascanu (2013). "Advances in Optimizing Recurrent Networks". In: *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (Vancouver, British Columbia, Canada, May 26–31, 2013). DOI: 10.1109/ICASSP.2013.6639349.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013). "Representation Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8, pp. 1798–1828. DOI: 10.1109/TPAMI.2013.50.

Bengio, Yoshua, Réjean Ducharme, et al. (2003). "A neural probabilistic language model". In: *Journal of Machine Learning Research* 3.Feb, pp. 1137–1155. URL: http://www.jmlr.org/papers/v3/bengio03a.html.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville (2013). *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. arXiv: 1308.3432.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi (1994). "Learning Long-Term Dependencies with Gradient Descent is Difficult". In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166. DOI: 10.1109/72.279181.

Bergstra, James et al. (2010). "Theano: a CPU and GPU Math Expression Compiler". In: *Proceedings of the 9th Python in Science Conference* (Austin, TX, June 28–July 3, 2010). Ed. by Stéfan van der Walt and Jarrod Millman, pp. 3–10. URL: http://conference.scipy.org/proceedings/scipy2010/bergstra.html.

Bischof, Christian H and H Martin Bücker (2000). "Computing derivatives of computer programs". In: *Modern Methods and Algorithms of Quantum Chemistry*. Proceedings, Second Edition (Jülich, Germany, Feb. 21–25, 2000). Vol. 3. Jülich, pp. 315–327. URL: http://juser.fz-juelich.de/record/44658.

Bischof, Christian et al. (1996). "ADIFOR 2.0: Automatic differentiation of Fortran 77 programs". In: *IEEE Computational Science and Engineering* 3.3, pp. 18–32. DOI: 10.1109/99.537089.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer-Verlag. ISBN: 978-0-387-31073-2.

Bojanowski, Piotr, Edouard Grave, et al. (2017). "Enriching word vectors with subword information". In: *Transactions of the Association for Computational Linguistics* 5. Ed. by Hinrich Schutze, pp. 135–146. URL: http://aclweb.org/anthology/Q17-1010.

Bojanowski, Piotr, Armand Joulin, and Tomáš Mikolov (2016). "Alternative structures for character-level RNNs". In: *Proceedings of the 2016 International Conference on Learning Representations*. Workshop Track (San Juan, Puerto Rico, May 2–4, 2016). URL: https://openreview.net/forum?id=wVqzL1ypocG0qV7mtLqm.

Boulanger-Lewandowski, Nicolas, Yoshua Bengio, and Pascal Vincent (2013). "Audio Chord Recognition with Recurrent Neural Networks". In: *Proceedings of the 14th International Society for Music Information Retrieval Conference* (Curitiba, Brazil, Nov. 4–8, 2013). Ed. by Alceu de Souza Britto Jr., Fabien Gouyon, and Simon Dixon, pp. 335–340. URL: http://www.ppgia.pucpr.br/ismir2013/wp-content/uploads/2013/09/243%5C_Paper.pdf.

Bridle, John S (1990). "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition". In: *Neurocomputing. Algorithms, Architectures and Applications*. Proceedings of the NATO Conference on Neurocomputing (Les Arcs, France, Feb. 1989). Ed. by Françoise Fogelman Soulié and Jeanny Hérault. Vol. 68. NATO ASI Series. Berlin and Heidelberg, pp. 227–236. DOI: 10.1007/978-3-642-76153-9_28.

Bryson, Arthur E and Walter F Denham (1962). "A steepest-ascent method for solving optimum programming problems". In: *Journal of Applied Mechanics* 29.2, pp. 247–257. DOI: 10.1115/1.3640537.

Buckman, Jacob and Graham Neubig (2018). "Neural Lattice Language Models". In: *Transactions of the Association for Computational Linguistics* 6. Ed. by Holger Schwenk, pp. 529–541. URL: https://transacl.org/ojs/index.php/tacl/article/view/1261.

Chan, William et al. (2017). "Latent sequence decompositions". In: *Proceedings of the 2017 International Conference on Learning Representations* (Toulon, France, Apr. 24–26, 2017). URL: https://openreview.net/forum?id=SyQq185lg.

Chandar A P, Sarath et al. (2014). "An Autoencoder Approach to Learning Bilingual Word Representations". In: *Advances in Neural Information Processing Systems 27* (Montréal, Canada, Dec. 8–13, 2014). Ed. by Zoubin Ghahramani et al., pp. 1853–1861. URL: https://papers.nips.cc/paper/5270-an-autoencoder-approach-to-learning-bilingual-word-representations.

Chen, Stanley F. and Joshua T. Goodman. (1996). "An Empirical Study of Smoothing Techniques for Language Modeling". In: *34th Annual Meeting of the Association for Computational Linguistics*. Proceedings of the Conference (Santa Cruz, California, USA, June 24–27, 1996), pp. 310–318. URL: http://aclweb.org/anthology/P96-1041.

Chen, Tianqi, Mu Li, et al. (2015). *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. Presented at the NIPS 2015 Workshop on Machine Learning Systems. arXiv: 1512.01274.

Chen, Tianqi, Thierry Moreau, et al. (2018). “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, California, USA, Oct. 8–10, 2018), pp. 578–594. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

Cho, KyungHyun et al. (2014). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Proceedings of the 2014 Conference on Empiricial Methods in Natural Language Processing* (Doha, Qatar, Oct. 25–29, 2014).

Choromanska, Anna et al. (2015). “The loss surface of multilayer networks”. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics* (San Diego, California, USA, May 9–12, 2015). Ed. by Guy Lebanon and S. V. N. Vishwanathan. Vol. 38. Proceedings of Machine Learning Research, pp. 192–204. URL: http://proceedings.mlr.press/v38/choromanska15.html.

Christianson, Bruce (2012). “A Leibniz Notation for Automatic Differentiation”. In: Lecture Notes in Computational Science and Engineering 87. Ed. by Shaun Forth et al., pp. 1–9. DOI: 10.1007/978-3-642-30023-3_1.

Chung, Junyoung, Sungjin Ahn, and Yoshua Bengio (2017). “Hierarchical Multiscale Recurrent Neural Networks”. In: *Proceedings of the 2017 International Conference on Learning Representations* (Toulon, France, Apr. 24–26, 2017). URL: https://openreview.net/forum?id=S1di0sfgl.

Chung, Junyoung, KyungHyun Cho, and Yoshua Bengio (2016). “A character-level decoder without explicit segmentation for neural machine translation”. In: *54th Annual Meeting of the Association for Computational Linguistics*. Proceedings of the Conference (Berlin, Germany, Aug. 7–12, 2016), pp. 1693–1703. DOI: 10.18653/v1/P16-1160.

Click, Cliff and Michael Paleczny (1995). “A simple graph-based intermediate representation”. In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate Representations* (San Francisco, California, USA, Jan. 22, 1995). New York, pp. 35–49. DOI: 10.1145/202529.202534.

Clifford, William Kingdon (1873). “Preliminary sketch of biquaternions”. In: *Proceedings of the London Mathematical Society* 4.1, pp. 381–395. DOI: 10.1112/plms/s1-4.1.381.

Cortes, Corinna and Vladimir Vapnik (1995). “Support Vector Networks”. In: *Machine Learning* 20.3, pp. 273–297. DOI: 10.1007/BF00994018.

Cox, David R (1958). “The regression analysis of binary sequences”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 20.2, pp. 215–242. URL: http://www.jstor.org/stable/2983890.

Cybenko, George (1989). “Approximation by Superpositions of a Sigmoidal Function”. In: *Mathematics of Control, Signals and Systems* 2.4, pp. 303–314. DOI: 10.1007/BF02551274.

Dahl, George E. et al. (2012). “Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 20.1, pp. 30–42. DOI: 10.1109/TASL.2011.2134090.

Dauphin, Yann N et al. (2014). “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”. In: *Advances in Neural Information Processing Systems 27* (Montréal, Canada, Dec. 8–13, 2014). Ed. by Zoubin Ghahramani et al., pp. 2933–2941. URL: https://papers.nips.cc/paper/5486-identifying-and-attacking-the-saddle-point-problem-in-high-dimensional-non-convex-optimization.

Dean, Jeffrey et al. (2012). “Large Scale Distributed Deep Networks”. In: *Advances in Neural Information Processing Systems 25* (Lake Tahoe, Nevada, USA, Dec. 3–8, 2012). Ed. by Fernando Pereira et al., pp. 1223–1231. URL: https://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.

Denil, Misha et al. (2013). “Predicting Parameters in Deep Learning”. In: *Advances in Neural Information Processing Systems 26* (Lake Tahoe, Nevada, USA, Dec. 5–10, 2012). Ed. by Christopher J. C. Burges et al., pp. 2148–2156. URL: https://papers.nips.cc/paper/5025-predicting-parameters-in-deep-learning.

Devlin, Jacob et al. (2014). “Fast and Robust Neural Network Joint Models for Statistical Machine Translation”. In: *52nd Annual Meeting of the Association for Computational Linguistics*. Proceedings of the Conference (Baltimore, Maryland, USA, June 22–27, 2014). Ed. by Kristina Toutanova and Hua Wu, pp. 1370–1380. DOI: 10.3115/v1/P14-1129.

Dreyfus, Stuart E. (1962). “The numerical solution of variational problems”. In: *Journal of Mathematical Analysis and Applications* 5.1, pp. 30–45. DOI: 10.1016/0022-247X(62)90004-5.

Driscoll, James R et al. (1989). “Making data structures persistent”. In: *Journal of computer and system sciences* 38.1, pp. 86–124. DOI: 10.1016/0022-0000(89)90034-2.

Duchi, John, Elad Hazan, and Yoram Singer (2011). “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12, pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

Dyson, Freeman (2004). “A meeting with Enrico Fermi”. In: *Nature* 427, pp. 297–297. DOI: 10.1038/427297a.

Elliott, Conal (2018). “The simple essence of automatic differentiation”. In: 2, p. 70. DOI: 10.1145/3236765.

Fike, Jeffrey A and Juan J Alonso (2012). “Automatic differentiation through the use of hyper-dual numbers for second derivatives”. In: Lecture Notes in Computational Science and Engineering 87. Ed. by Shaun Forth et al., pp. 163–173. DOI: 10.1007/978-3-642-30023-3_15.

Firat, Orhan, KyungHyun Cho, and Yoshua Bengio (2016). “Multi-way, multilingual neural machine translation with a shared attention mechanism”. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (San Diego, California, USA, June 12–17, 2016). Ed. by Kevin Knight, Ani Nenkova, and Owen Rambow, pp. 866–875. DOI: 10.18653/v1/N16-1101.

Flanagan, Cormac et al. (1993). “The essence of compiling with continuations”. In: *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (Albuquerque, New Mexico, USA, June 21–25, 1993), pp. 237–247. DOI: 10.1145/155090.155113.

Gal, Yarin (2016). “A Theoretically Grounded Application of Dropout in Recurrent Neural Networks”. In: *Advances in Neural Information Processing Systems 29* (Barcelona, Spain, Dec. 5–10, 2016). Ed. by Daniel D. Lee et al., pp. 1019–1027. URL: https://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks.

Ganin, Yaroslav et al. (2016). “Domain-adversarial training of neural networks”. In: *Journal of Machine Learning Research* 17, pp. 2096–2030.

Gao, Jianfeng et al. (2013). *Learning Semantic Representations for the Phrase Translation Model*. arXiv: 1312.0482.

Gehring, Jonas et al. (2017). “Convolutional Sequence to Sequence Learning”. In: *Proceedings of the 34th International Conference on Machine Learning* (Sydney, Australia, Aug. 6–11, 2017). Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1243–1252. URL: http://proceedings.mlr.press/v70/gehring17a.html.

Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (Sardinia, Italy, May 13–15, 2010). Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. PMLR, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (Apr. 11, 2011–Apr. 13, 2014). “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (Fort Lauderdale, FL, USA). Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. PMLR, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a.html.

Goldwater, Sharon, Mark Johnson, and Thomas L Griffiths (2005). “Interpolating between types and tokens by estimating power-law generators”. In: *Advances in Neural Information Processing Systems 18* (Whistler, British Columbia, Canada, Dec. 5–8, 2005). Ed. by Yair Weiss, Bernhard Schölkopf, and John C. Platt, pp. 459–466. URL: https://papers.nips.cc/paper/2941-interpolating-between-types-and-tokens-by-estimating-power-law-generators.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press. URL: http://www.deeplearningbook.org.

Goodfellow, Ian, David Warde-Farley, et al. (2013). “Maxout Networks”. In: *Proceedings of the 30th International Conference on Machine Learning* (Atlanta, Georgia, USA, June 17–19, 2013). Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. PMLR, pp. 1319–1327. URL: http://proceedings.mlr.press/v28/goodfellow13.html.

Gori, Marco, Yoshua Bengio, and Renato De Mori (1989). "BPS: a learning algorithm for capturing the dynamic nature of speech". In: *International 1989 Joint Conference on Neural Networks* (Washington, District of Columbia, USA, June 18–22, 1989). Vol. 2, pp. 417–423. DOI: 10.1109/IJCNN.1989.118276.

Gould, Nicholas I. M. and Jorge Nocedal (1998). *The modified absolute-value factorization norm for trust-region minimization*. Ed. by Renato De Leone et al. Springer, pp. 225–241. DOI: 10.1007/978-1-4613-3279-4_15.

Gower, Robert Mansel and Margarida P Mello (2012). "A new framework for the computation of Hessians". In: *Optimization Methods and Software* 27.2, pp. 251–273. DOI: 10.1080/10556788.2011.580098.

Graves, A. et al. (2006). "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks". In: *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA, June 25–29, 2006), pp. 369–376. DOI: 10.1145/1143844.1143891.

Graves, Alex (2011). "Practical Variational Inference for Neural Networks". In: *Advances in Neural Information Processing Systems 24* (Grenada, Spain, Dec. 12–17, 2011). Ed. by J. Shawe-Taylor et al., pp. 2348–2356. URL: https://papers.nips.cc/paper/4329-practical-variational-inference-for-neural-networks.

— (2012a). "Sequence Transduction with Recurrent Neural Networks". In: *Proceedings of the 29th International Conference on Machine Learning. Representation Learning Workshop*. arXiv: 1211.3711.

— (2012b). *Supervised Sequence Labelling with Recurrent Neural Networks*. Vol. 385. Studies in Computational Intelligence. Springer. DOI: 10.1007/978-3-642-24797-2.

— (2013). *Generating Sequences With Recurrent Neural Networks*. arXiv: 1308.0850.

Graves, Alex and Jürgen Schmidhuber (2005). "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". In: *Neural Networks* 18.5, pp. 602–610. DOI: 10.1016/j.neunet.2005.06.042.

Greenberg, Joseph H (1960). "A quantitative approach to the morphological typology of language". In: *International Journal of American Linguistics* 26.3, pp. 178–194. URL: http://www.jstor.org/stable/1264155.

Greenstadt, John (1967). “On the relative efficiencies of gradient methods”. In: *Mathematics of Computation* 21.99, pp. 360–367. URL: http://www.jstor.org/stable/2003238.

Greff, Klaus et al. (2017). “LSTM: a search space odyssey”. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10, pp. 2222–2232. DOI: 10.1109/TNNLS.2016.2582924.

Griewank, Andreas (1992). “Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation”. In: *Optimization Methods and software* 1.1, pp. 35–54. DOI: 10.1080/10556789208805505.

Griewank, Andreas, David Juedes, and Jean Utke (1996). “Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++”. In: *ACM Transactions on Mathematical Software (TOMS)* 22.2, pp. 131–167. DOI: 10.1145/229473.229474.

Griewank, Andreas and Andrea Walther (2000). “Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation”. In: *ACM Transactions on Mathematical Software* 26.1, pp. 19–45. DOI: 10.1145/347837.347846.

— (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. 2nd ed. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898717761.

Grimm, José, Loïc Pottier, and Nicole Rostaing-Schmidt (1996). *Optimal time and minimum space-time product for reversing a certain class of programs*. research report RR-2794. INRIA. URL: https://hal.inria.fr/inria-00073896.

Gruslys, Audrunas et al. (2016). “Memory-efficient backpropagation through time”. In: *Advances in Neural Information Processing Systems 29* (Barcelona, Spain, Dec. 5–10, 2016). Ed. by Daniel D. Lee et al., pp. 4125–4133. URL: https://papers.nips.cc/paper/6221-memory-efficient-backpropagation-through-time.

Guelton, Serge et al. (2015). “Pythran: Enabling static optimization of scientific python programs”. In: *Computational Science & Discovery* 8.1, p. 014001.

Gulcehre, Caglar et al. (2015). *On using monolingual corpora in neural machine translation*. arXiv: 1503.03535.

Hascoët, Laurent (2017). *Some highlights on Source-to-Source Adjoint AD*. Presented at NIPS Autodiff Workshop. URL: https://openreview.net/pdf?id=r1tLym8T-.

Hascoët, Laurent, Uwe Naumann, and Valérie Pascual (2003). *TBR Analysis in Reverse-Mode Automatic Differentiation*. research report RR-4856. INRIA. URL: https://hal.inria.fr/inria-00071727.

Hascoët, Laurent and Valérie Pascual (2013). "The Tapenade Automatic Differentiation tool: principles, model, and specification". In: *ACM Transactions on Mathematical Software* 39.3, p. 20. DOI: 10.1145/2450153.2450158.

Hastie, Trevor, Rober Tibshirani, and Jerome Friedman (2001). *The elements of statistical learning. Data mining, inference and prediction*. Springer Series in Statistics. Springer. DOI: 10.1007/978-0-387-84858-7.

Heess, Nicolas et al. (2015). "Learning continuous control policies by stochastic value gradients". In: *Advances in Neural Information Processing Systems 28*. Ed. by Corinna Cortes et al., pp. 2944–2952. URL: https://papers.nips.cc/paper/5796-learning-continuous-control-policies-by-stochastic-value-gradients.

Hill, Felix et al. (2014). "Not All Neural Embeddings are Born Equal". In: *NIPS 2014 Workshop on Learning Semantics*. arXiv: 1410.0718.

Hinton, Geoffrey (2012). *Neural Networks for Machine Learning*. Coursera.

Hinton, Geoffrey E., Simon Osindero, and Yee Whye Teh (2006). "A fast learning algorithm for deep belief nets". In: *Neural Computation* 18.7, pp. 1527–1554. DOI: 10.1162/neco.2006.18.7.1527.

Hinton, Geoffrey E, James L McClelland, David E Rumelhart, et al. (1986). "Distributed representations". In: *Parallel distributed processing. Explorations in the microstructure of cognition*. Ed. by Jerome A. Feldman, Patrick J Hayes, and David E Rumelhart. Vol. 1. Computational Models of Cognition and Perception. The MIT Press. Chap. 3, pp. 77–109. ISBN: 9780262181204.

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean (2015). "Distilling the knowledge in a neural network". In: *NIPS 2014 Deep Learning Workshop*. arXiv: 1503.02531.

Hinze, Michael and Julia Sternberg (2005). "A-revolve: an adaptive memory-reduced procedure for calculating adjoints; with an application to computing adjoints of the instationary Navier–Stokes system". In: *Optimization Methods and Software* 20.6, pp. 645–663. DOI: 10.1080/10556780410001684158.

Hochreiter, Josef (1991). “Untersuchungen zu dynamischen neuronalen Netzen”. PhD thesis. Technische Universität München.

Hochreiter, Josef and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

Hopfield, John J. (1982). “Neural Networks and Physical Systems with Emergent Collective Computational Abilities”. In: *Proceedings of the National Academy of Sciences* 79.8, pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). “Multilayer Feedforward Networks Are Universal Approximators”. In: *Neural Networks* 2.5, pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

Jang, Eric, Shixiang Gu, and Ben Poole (2017). “Categorical reparameterization with Gumbel-softmax”. In: *Proceedings of the 2017 International Conference on Learning Representations* (Toulon, France, Apr. 24–26, 2017). URL: https://openreview.net/forum?id=rkE3y85ee.

Johnston, Wesley M., J. R. Paul Hanna, and Richard J. Millar (2004). “Advances in dataflow programming languages”. In: *ACM Computing Surveys* 36.1, pp. 1–34. DOI: 10.1145/1013208.1013209.

Jozefowicz, Rafal, Oriol Vinyals, et al. (2016). *Exploring the limits of language modeling*. arXiv: 1602.02410.

Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever (2015). “An Empirical Exploration of Recurrent Network Architectures”. In: *Proceedings of The 32nd International Conference on Machine Learning* (Lille, France, July 6–11, 2015), pp. 2342–2350.

Kalchbrenner, Nal and Phil Blunsom (2013). “Recurrent Continuous Translation Models”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA, Oct. 18–21, 2013). Ed. by David Yarowsky et al., pp. 1700–1709. URL: http://aclweb.org/anthology/D13-1176.

Kalchbrenner, Nal, Lasse Espeholt, et al. (2016). *Neural machine translation in linear time*. arXiv: 1610.10099.

Karczmarczuk, Jerzy (2001). “Functional differentiation of computer programs”. In: *Higher-Order and Symbolic Computation* 14.1, pp. 35–57. DOI: 10.1023/A:1011501232197.

Katz, Slava M. (1987). “Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35.3, pp. 400–401. DOI: 10.1109/TASSP.1987.1165125.

Kim, Yoon et al. (2015). “Character-aware neural language models”. In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona, USA, Feb. 12–17, 2016), pp. 2741–2749. arXiv: 1508.06615.

Kingma, Diederik and Jimmy Ba (2015). “Adam: A Method for Stochastic Optimization”. In: *Proceedings of the 2015 International Conference on Learning Representations* (San Diego, California, USA, May 7–9, 2015). DOI: 11245/1.505367.

Koehn, P. (2005). “Europarl: A Parallel Corpus for Statistical Machine Translation”. In: *Proceedings of the Tenth Machine Translation Summit* (Phuket, Thailand, Sept. 12, 2005–Sept. 16, 2009), pp. 79–86. URL: http://www.mt-archive.info/MTS-2005-Koehn.pdf.

Koehn, Philipp, Franz Josef Och, and Daniel Marcu (2003). “Statistical Phrase-based Translation”. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Edmonton, Canada, May 27–June 1, 2003), pp. 48–54.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by Fernando Pereira et al., pp. 1097–1105. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert (2015). “Numba: A LLVM-based Python JIT compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas, USA, Nov. 15, 2015), p. 7. DOI: 10.1145/2833157.2833162.

Le Roux, Nicolas, Pierre-Antoine Manzagol, and Yoshua Bengio (2008). “Topmoumoute online natural gradient algorithm”. In: *Advances in Neural Information Processing Systems 20*. Ed. by John C. Platt et al., pp. 849–856. URL: http://papers.nips.cc/paper/3234-topmoumoute-online-natural-gradient-algorithm.

Le, Hai-Son, Alexandre Allauzen, and François Yvon (2012). "Continuous Space Translation Models with Neural Networks". In: *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Montréal, Canada, June 3–8, 2012). Ed. by Eric Fosler-Lussier, Ellen Riloff, and Srinivas Bangalore, pp. 39–48. URL: http://aclweb.org/anthology/N12-1005.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521, pp. 436–444. DOI: 10.1038/nature14539.

LeCun, Yann, Léon Bottou, et al. (1998). "Efficient Backprop". In: *Neural Networks: Tricks of the Trade*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Vol. 7700. Lecture Notes in Computer Science, pp. 9–48. DOI: 10.1007/978-3-642-35289-8_3.

Lee, Jason, KyungHyun Cho, and Thomas Hofmann (2017). "Fully character-level neural machine translation without explicit segmentation". In: *Transactions of the Association for Computational Linguistics* 5. Ed. by Adam Lopez, pp. 365–378.

Leißa, Roland, Marcel Köster, and Sebastian Hack (2015). "A graph-based higher-order intermediate representation". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (San Francisco, California, USA, Feb. 7–11, 2015), pp. 202–212.

Lillicrap, Timothy P et al. (2016). "Random feedback weights support learning in deep neural networks". In: *Nature Communications* 7, p. 13276. DOI: 10.1038/ncomms13276.

Lindenmaier, Götz et al. (2005). *Firm, an intermediate language for compiler research.* technical report. Fakultät für Informatik, Universität Karlsruhe. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003172.

Ling, Wang et al. (2015). *Character-based neural machine translation.* arXiv: 1511.04586.

Liu, Chang, Daniel Dahlmeier, and Hwee Tou Ng (2011). "Better evaluation metrics lead to better machine translation". In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing* (Edinburgh, United Kingdom, July 27–31, 2011), pp. 375–384.

Liu, Hairong et al. (2017). "Gram-CTC: Automatic Unit Selection and Target Decomposition for Sequence Labelling". In: *Proceedings of the 34th International*

*Conference on Machine Learning* (Sydney, Australia, Aug. 6–11, 2017). Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 2188–2197. URL: http://proceedings.mlr.press/v70/liu17f.html.

Luong, Minh-Thang, Hieu Pham, and Christopher D Manning (2015). "Effective approaches to attention-based neural machine translation". In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (Lisbon, Portugal, Sept. 17–21, 2015). Ed. by Lluís Màrquez, Chris Callison-Burch, and Jian Su, pp. 1412–1421. DOI: 10.18653/v1/D15-1166.

Luong, Minh-Thang, Ilya Sutskever, et al. (2014). "Addressing the rare word problem in neural machine translation". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing* (Beijing, China, July 26–31, 2015), pp. 11–19. DOI: 10.3115/v1/P15-1002.

Maaten, Laurens van der (2013). "Barnes-Hut-SNE". In: *Proceedings of the 2013 International Conference on Learning Representations* (Scottsdale, Arizona, USA, May 2–4, 2013). arXiv: 1301.3342.

Maclaurin, Dougal, David Duvenaud, and Ryan P Adams (2015). *Autograd: Effortless Gradients in NumPy.* Presented at the ICML 2015 AutoML Workshop.

Mahoney, Matthew V (1999). "Text compression as a test for artificial intelligence". In: *Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence* (Orlando, Florida, USA, July 18–22, 1999), p. 970.

— (2000). "Fast Text Compression with Neural Networks". In: *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference* (Orlando, Florida, USA, May 21–23, 2000), pp. 230–234.

— (2006). *Large text compression benchmark.* URL: http://mattmahoney.net/dc/text.html (visited on 10/25/2018).

Manning, Christopher D. and Hinrich Schütze (1999). *Foundations of Statistical Natural Language Processing.* 6th ed. Cambridge, Massachusetts, USA: MIT Press. ISBN: 0-262-13360-1.

Marcu, Daniel and William Wong (2002). "A Phrase-based, Joint Probability Model for Statistical Machine Translation". In: *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing* (Philadelphia, Pennsylvania, USA, July 6–7, 2002), pp. 133–139. URL: https://aclanthology.info/papers/W02-1018/w02-1018.

Martens, James, Ilya Sutskever, and Kevin Swersky (2012). “Estimating the Hessian by back-propagating curvature”. In: *Proceedings of the 29th International Conference on Machine Learning* (Edinburgh, Scotland, United Kingdom, June 25–July 1, 2012).

McCulloch, Warren S. and Walter Pitts (1943). “A Logical Calculus of Ideas Immanent in Nervous Activity”. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. DOI: 10.1007/BF02478259.

Merriënboer, Bart van, Dan Moldovan, and Alexander B. Wiltschko (2018). “Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming”. In: *Advances in Neural Information Processing Systems 31* (Montréal, Canada, Dec. 2–8, 2018). Ed. by Samy Bengio et al. URL: https://papers.nips.cc/paper/7863-tangent-automatic-differentiation-using-source-code-transformation-for-dynamically-typed-array-programming.

Mikolov, Tomáš (2012). “Statistical Language Models based on Neural Networks”. PhD thesis. Brno University of Technology. URL: http://www.fit.vutbr.cz/~imikolov/rnnlm/thesis.pdf.

Mikolov, Tomáš, Martin Karafiát, et al. (2010). “Recurrent neural network based language model”. In: *Proceedings of the 11th Annual Conference of the International Speech Communication Association* (Makuhari, Chiba, Japan, Sept. 26–30, 2010), pp. 1045–1048.

Mikolov, Tomáš, Ilya Sutskever, et al. (2013). “Distributed representations of words and phrases and their compositionality”. In: *Advances in Neural Information Processing Systems 26* (Stateline, Nevada, USA, Dec. 5–10, 2013). Ed. by Christopher J.C. Burges et al., pp. 3111–3119. URL: https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.

Minsky, Marvin L. and Seymour A. Papert (1969). *Perceptrons. An Introduction to Computational Geometry*. Cambridge, Massachusetts, USA: MIT Press. ISBN: 0-262-63022-2.

Mitchell, Jeff and Mirella Lapata (2008). “Vector-based Models of Semantic Composition”. In: *Proceedings of the 2008 Conference of the North American Chapter* *of the Association for Computational Linguistics: Human Language Technologies* (Columbus, Ohio, USA, June 16–18, 2008). Ed. by Johanna D. Moore et al., pp. 236–244. URL: https://aclanthology.info/papers/P08-1028/p08-1028.

Mitchell, Tom M. (1997). *Machine Learning.* New York City, New York, USA: McGraw-Hill. ISBN: 0070428077.

Mochihashi, Daichi, Takeshi Yamada, and Naonori Ueda (2009). "Bayesian unsupervised word segmentation with nested Pitman-Yor language modeling". In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing* (Suntec, Singapore, Aug. 2–7, 2009), pp. 100–108.

Montufar, Guido F. et al. (2014). "On the Number of Linear Regions of Deep Neural Networks". In: *Advances in Neural Information Processing Systems 27* (Montréal, Canada, Dec. 8–13, 2014). Ed. by Zoubin Ghahramani et al. URL: https://papers.nips.cc/paper/5422-on-the-number-of-linear-regions-of-deep-neural-networks.

Moon, Taesup et al. (2015). "RNNDROP: A novel dropout for RNNs in ASR". In: *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)* (Scottsdale, Arizona, USA, Dec. 13–17, 2015). DOI: 10.1109/ASRU.2015.7404775.

Moore, Robert C. and William Lewis (2010). "Intelligent Selection of Language Model Training Data". In: *Proceedings of the ACL 2010 Conference Short Papers* (Upssala, Sweden, July 11–16, 2010). Ed. by Jan Hajič et al., pp. 220–224. URL: https://aclanthology.info/papers/P10-2041/p10-2041.

Naumann, Uwe (2008). "Optimal Jacobian accumulation is NP-complete". In: *Mathematical Programming* 112.2, pp. 427–441. DOI: 10.1007/s10107-006-0042-z.

— (2012). *The art of differentiating computer programs. An introduction to algorithmic differentiation.* Philadelphia, Pennsylvania, USA: Society for Industrial and Applied Mathematics. ISBN: 978-1-61197-206-1. DOI: 10.1137/1.9781611972078.

Neelakantan, Arvind et al. (2015). *Adding Gradient Noise Improves Learning for Very Deep Networks.* arXiv: 1511.06807.

Nesterov, Yu (1983). "A method for solving the convex programming problem with convergence rate $O(1/k^2)$". In: *Dokl Akad Nauk* 269, pp. 543–547.

Nocedal, Jorge and Stephen J. Wright (2006). *Numerical Optimization*. 2nd ed. New York City, New York, USA: Springer. ISBN: 978-0-387-30303-1. DOI: 10.1007/978-0-387-40065-5.

Nøkland, Arild (2016). "Direct feedback alignment provides learning in deep neural networks". In: *Advances in Neural Information Processing Systems 29* (Barcelona, Spain, Dec. 5–10, 2016). Ed. by Daniel D. Lee et al., pp. 1045–1053.

Och, Franz Josef and Hermann Ney (2003). "A systematic comparison of various statistical alignment models". In: *Computational linguistics* 29.1, pp. 19–51.

Oliphant, Travis E (2006). *A guide to NumPy*. 2nd ed. CreateSpace Independent Publishing Platform. ISBN: 978-1517300074.

Oord, Aaron van den, Oriol Vinyals, and Koray Kavukcuoglu (2017). "Neural Discrete Representation Learning". In: *Advances in Neural Information Processing Systems 30* (Long Beach, California, USA, Dec. 4–9, 2017). URL: https://papers.nips.cc/paper/7210-neural-discrete-representation-learning.

Papineni, Kishore et al. (2002). "BLEU: A method for automatic evaluation of machine translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (Philadelphia, Pennsylvania, USA, July 7–12, 2002), pp. 311–318.

Pascanu, Razan et al. (2014). "How to Construct Deep Recurrent Neural Networks". In: *Proceedings of the 2014 International Conference on Learning Representations* (Banff, Canada, Apr. 14–16, 2014).

Pascanu, Razvan, Yann N Dauphin, et al. (2014). *On the saddle point problem for non-convex optimization*. arXiv: 1405.4604.

Pascanu, Razvan, Tomáš Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks". In: *Proceedings of the 30th International Conference on Machine Learning* (Atlanta, Georgia, USA, June 17–19, 2013). Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. 3. PMLR, pp. 1310–1318. URL: http://proceedings.mlr.press/v28/pascanu13.html.

Paszke, Adam et al. (2017). *Automatic differentiation in PyTorch*. Presented at the NIPS 2017 Autodiff Workshop.

Pearlmutter, Barak A (1994). “Fast exact multiplication by the Hessian”. In: *Neural Computation* 6.1, pp. 147–160.

Pearlmutter, Barak A and Jeffrey Mark Siskind (2007). “Lazy multivariate higher-order forward-mode AD”. In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Nice, France, Jan. 17–19, 2007), pp. 155–160.

— (2008). “Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator”. In: *ACM Transactions on Programming Languages and Systems* 30.2, pp. 1–30. DOI: 10.1145/1330017.1330018.

Pham, Vu et al. (2014). “Dropout improves recurrent neural networks for handwriting recognition”. In: *Proceedings of the 14th International Conference on Frontiers in Handwriting Recognition* (Heraklion, Greece, Sept. 1–4, 2014). IEEE, pp. 285–290. DOI: 10.1109/ICFHR.2014.55.

Pontryagin, LS et al. (1962). *The mathematical theory of optimal processes*. Gordon and Breach Science Publishers. ISBN: 2-88124-077-1.

Poole, Ben et al. (2016). “Exponential expressivity in deep neural networks through transient chaos”. In: *Advances in Neural Information Processing Systems 29* (Barcelona, Spain, Dec. 5–10, 2016). Ed. by Daniel D. Lee et al., pp. 3360–3368. URL: https://papers.nips.cc/paper/6322-exponential-expressivity-in-deep-neural-networks-through-transient-chaos.

Rae, Jack et al. (2016). “Scaling memory-augmented neural networks with sparse reads and writes”. In: *Advances in Neural Information Processing Systems 29* (Barcelona, Spain, Dec. 5–10, 2016). Ed. by Daniel D. Lee et al., pp. 3621–3629. URL: https://papers.nips.cc/paper/6298-scaling-memory-augmented-neural-networks-with-sparse-reads-and-writes.

Rall, Louis B (1986). “The arithmetic of differentiation”. In: *Mathematics Magazine* 59.5, pp. 275–282. DOI: 10.2307/2689402.

Ren, Xiaofeng and Jitendra Malik (2003). “Learning a Classification Model for Segmentation”. In: *Proceedings Ninth IEEE International Conference on Computer Vision* (Nice, France, Oct. 13–16, 2003), pp. 10–17. DOI: 10.1109/ICCV.2003.1238308.

Revels, Jarrett, Miles Lubin, and Theodore Papamarkou (2016). *Forward-Mode Automatic Differentiation in Julia*. Presented at the 7th International Conference on Algorithmic Differentiation. arXiv: 1607.07892.

Al-Rfou, Rami et al. (2016). *Theano: A Python framework for fast computation of mathematical expressions*. arXiv: 1605.02688.

Rosenblatt, Frank (1958). “The perceptron: A probabilistic model for information storage and organization in the brain”. In: *Psychological Review* 65.6, pp. 386–408.

— (1962). *Principles of Neurodynamics. Perceptrons and the theory of brain mechanisms*. New York City, New York, USA: Spartan Books.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). “Learning Representations by Back-Propagating Errors”. In: *Nature* 323, pp. 533–536. DOI: 10.1038/323533a0.

Russel, Stuart J. and Peter Norvig (2003). *Artificial Intelligence. A Modern Approach*. 3rd ed. Prentice Hall. ISBN: 0-13-604259-7.

Saxe, Andrew M., James L. McClelland, and Surya Ganguli (2014). “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks”. In: *Proceedings of the 2014 International Conference on Learning Representations* (Banff, Canada, Apr. 14–16, 2014).

Schmidhuber, Jürgen (2015). “Deep Learning in Neural Networks: An Overview”. In: *Neural Networks* 61, pp. 85–117. DOI: 10.1016/j.neunet.2014.09.003.

Schraudolph, Nicol N (2002). “Fast curvature matrix-vector products for second-order gradient descent”. In: *Neural Computation* 14.7, pp. 1723–1738. DOI: 10.1162/08997660260028683.

Schuster, M. and K. Paliwal (1997). “Bidirectional recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 45.11, pp. 2673–2681. DOI: 10.1109/78.650093.

Schwenk, Holger (2007). “Continuous Space Language Models”. In: *Computer Speech and Language* 21.3, pp. 492–518. DOI: 10.1016/j.csl.2006.09.003.

— (2012). “Continuous Space Translation Models for Phrase-Based Statistical Machine Translation”. In: *Proceedings of the 24th International Conference on Computational Linguistics* (Mumbai, India, Dec. 8–15, 2012). Ed. by Martin Kay and Christian Boitet, pp. 1071–1080. URL: https://aclanthology.info/papers/C12-2104/c12-2104.

Schwenk, Holger, Marta R. Costa-Jussà, and José A. R. Fonollosa (2006). *Continuous space language models for the IWSLT 2006 task*. Presented at the International Workshop on Spoken Language Translation.

Semeniuta, Stanislau, Aliaksei Severyn, and Erhardt Barth (2016). “Recurrent Dropout without Memory Loss”. In: *Proceedings of the 26th International Conference on Computational Linguistics* (Osaka, Japan, Dec. 11–16, 2016). Ed. by Yuji Matsumoto and Rashmi Prasad, pp. 1757–1766. URL: https://aclanthology.info/papers/C16-1165/c16-1165.

Sennrich, Rico, Barry Haddow, and Alexandra Birch (2015). “Neural machine translation of rare words with subword units”. In: *54th Annual Meeting of the Association for Computational Linguistics*. Proceedings of the Conference (Berlin, Germany, Aug. 7–12, 2016), pp. 1715–1725. DOI: 10.18653/v1/P16-1162.

Shannon, Claude E. (1948). “A Mathematical Theory of Communication”. In: *Bell System Technical Journal* 27.3, pp. 379–423.

Shivers, Olin (1991). “Control-flow analysis of higher-order languages”. PhD thesis. Carnegie Mellon University.

Silver, David et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587, p. 484.

Siskind, Jeffrey Mark and Barak A Pearlmutter (2005). *Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD*. Ed. by Andrew Butterfield. Presented at the 17th International Workshop on Implementation and Application of Functional Languages.

— (2008). *Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code*. Tech. rep. 367. Purdue University.

— (2016). *Efficient implementation of a higher-order language with built-in AD*. Presented at the 7th International Conference on Algorithmic Differentiation.

Socher, Richard et al. (2011). “Dynamic Pooling and Unfolding Recursive Autoencoders for Paraphrase Detection”. In: *Advances in Neural Information Processing Systems 24* (Grenada, Spain, Dec. 12–17, 2011). Ed. by J. Shawe-Taylor et al., pp. 801–809. URL: https://papers.nips.cc/paper/4204-dynamic-pooling-and-unfolding-recursive-autoencoders-for-paraphrase-detection.

Song, Xingyi, Trevor Cohn, and Lucia Specia (2013). “BLEU deconstructed: Designing a Better MT Evaluation Metric”. In: *Proceedings of the 14th International Conference on Intelligent Text Processing and Computational Linguistics* (Samos, Greece, Mar. 24–30, 2013).

Srivastava, Nitish et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Sternberg, Julia (2002). “Adaptive Umkehrschemata für Schrittfolgen mit nicht-uniformen Kosten”. PhD thesis. Diplomarbeit, 2002. Techn. Univ. Dresden.

Sundermeyer, Martin, Ralf Schlüter, and Hermann Ney (2012). “LSTM Neural Networks for Language Modeling”. In: *Proceedings of the 13th Annual Conference of the International Speech Communication Association*, pp. 194–197.

Sutskever, Ilya, James Martens, and Geoffrey E. Hinton (2011). “Generating text with recurrent neural networks”. In: *ICML'2011*, pp. 1017–1024.

Sutskever, Ilya, Oriol Vinyals, and Quoc Le (2014). “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems 27* (Montréal, Canada, Dec. 8–13, 2014). Ed. by Zoubin Ghahramani et al. URL: https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.

Szabó, Zoltán Gendler (2017). “Compositionality”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2017. Metaphysics Research Lab, Stanford University.

Tai, Kai Sheng, Richard Socher, and Christopher D Manning (2015). “Improved semantic representations from tree-structured long short-term memory networks”. In: *Proceedings of the Joint Conference of the 43rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing* (Beijing, China, July 26–31, 2015). Ed. by Chengqing Zong and Michael Strube, pp. 1556–1566. DOI: 10.3115/v1/P15-1150.

Teh, Yee Whye (2006). “A hierarchical Bayesian language model based on Pitman-Yor processes”. In: *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics* (Sydney, Australia, July 17–18, 2006), pp. 985–992. DOI: 10.3115/1220175.1220299.

Tokui, Seiya et al. (2015). *Chainer: a next-generation open source framework for deep learning*. Presented at the NIPS 2015 LearningSys Workshop. URL: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

Turing, Alan M (1950). "Computing machinery and intelligence". In: *Mind* 59.236, pp. 433–460.

Valiant, Leslie G. (1984). "A Theory of the Learnable". In: *Communications of the ACM* 27.11, pp. 1134–1142. DOI: 10.1145/1968.1972.

Van Merriënboer, Bart et al. (2015). *Blocks and fuel: Frameworks for deep learning.* arXiv: 1506.00619.

Vapnik, Vladimir Naumovich (1995). *The Nature of Statistical Learning Theory.* 2nd ed. New York: Springer. DOI: 10.1007/978-1-4757-3264-1.

Vaswani, Ashish, Noam Shazeer, et al. (2017). "Attention is all you need". In: *Advances in Neural Information Processing Systems 30* (Long Beach, California, USA, Dec. 4–9, 2017), pp. 5998–6008.

Vaswani, Ashish, Yinggong Zhao, et al. (2013). "Decoding with Large-Scale Neural Language Models Improves Translation". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA, Oct. 18–21, 2013). Ed. by David Yarowsky et al., pp. 1387–1392. URL: http://aclweb.org/anthology/D13-1140.

Verhulst, Pierre-François (1845). "Recherches mathématiques sur la loi d'accroissement de la population." In: *Nouveaux mémoires de l'académie royale des sciences et belles-lettres de Bruxelles* 18, pp. 14–54.

Vries, Edsko de, Rinus Plasmeijer, and David M. Abrahamson (2007). "Uniqueness Typing Simplified". In: (Freiburg, Germany, Sept. 27–29, 2007). Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók, pp. 201–218. DOI: 10.1007/978-3-540-85373-2.

Walker, Strother H and David B Duncan (1967). "Estimation of the probability of an event as a function of several independent variables". In: *Biometrika* 54.1-2, pp. 167–179. DOI: 10.1093/biomet/54.1-2.167.

Walt, Stéfan van der, S. Chris Colbert, and Gael Varoquaux (2011). "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2, pp. 22–30. DOI: 10.1109/MCSE.2011.37.

Walther, Andrea (2004). "Program reversals for evolutions with non-uniform step costs". In: *Acta Informatica* 40.4, pp. 235–263.

Wang, Fei and Tiark Rompf (2018). "A Language and Compiler View on Differentiable Programming". In: URL: https://openreview.net/forum?id=SJxJtYkPG.

Wang, Fei, Xilun Wu, et al. (2018). *Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator*. arXiv: 1803.10228.

Wang, Linnan et al. (2018). "Superneurons: dynamic GPU memory management for training deep neural networks". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austra, Feb. 24–28, 2018), pp. 41–53.

Wang, Mu, Assefaw Gebremedhin, and Alex Pothen (2016). "Capitalizing on live variables: new algorithms for efficient Hessian computation via automatic differentiation". In: *Mathematical Programming Computation* 8.4, pp. 393–433. DOI: 10.1007/s12532-016-0100-3.

Wei, Richard, Lane Schwartz, and Vikram Adve (2017). "DLVM: A modern compiler framework for neural network DSLs". In: *Proceedings of the 2018 International Conference on Learning Representations* (Vancouver, Canada, Apr. 30–May 3, 2018). URL: https://openreview.net/forum?id=HJxPq4ywz.

Wengert, Robert Edwin (1964). "A simple automatic derivative evaluation program". In: *Communications of the ACM* 7.8, pp. 463–464. DOI: 10.1145/355586.364791.

Werbos, Paul J. (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University.

Williams, Ronald J and Jing Peng (1990). "An efficient gradient-based algorithm for on-line training of recurrent network trajectories". In: *Neural computation* 2.4, pp. 490–501. DOI: 10.1162/neco.1990.2.4.490.

Wiseman, Sam and Alexander M Rush (2016). "Sequence-to-sequence learning as beam-search optimization". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (Austin, Texas, Nov. 1–5, 2016). Ed. by Jian Su, Kevin Duh, and Xavier Carreras, pp. 1296–1306. DOI: 10.18653/v1/D16-1137.

Wittgenstein, Ludwig (1953). *Philosophical Investigations*. Oxford, England, United Kingdom: Macmillan Publishing Company. ISBN: 978-1405159289.

Wu, Yonghui et al. (2016). *Google's neural machine translation system: Bridging the gap between human and machine translation*. Tech. rep. arXiv: 1609.08144.

Wu, Yuhuai et al. (2016). "On multiplicative integration with recurrent neural networks". In: *Advances in Neural Information Processing Systems 29* (Barcelona, Spain, Dec. 5–10, 2016). Ed. by Daniel D. Lee et al., pp. 2856–2864.

Zeiler, Matthew D. (2012). *ADADELTA: an adaptive learning rate method*. arXiv: 1212.5701.

Zhang, Min-Ling and Zhi-Hua Zhou (2006). “Multilabel neural networks with applications to functional genomics and text categorization”. In: *IEEE Transactions on Knowledge and Data Engineering* 18.10, pp. 1338–1351. DOI: 10.1109/TKDE.2006.162.

Zhang, Xingxing, Liang Lu, and Mirella Lapata (2016-06). “Top-down Tree Long Short-Term Memory Networks”. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (San Diego, California, USA, June 12–17, 2016), pp. 310–320. URL: http://www.aclweb.org/anthology/N16-1035.

Zou, Will Y. et al. (2013). “Bilingual Word Embeddings for Phrase-Based Machine Translation”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (Seattle, Washington, USA, Oct. 18–21, 2013). Ed. by David Yarowsky et al., pp. 1393–1398. URL: http://aclweb.org/anthology/D13-1141.