Université de Montréal

On Deep Multiscale Recurrent Neural Networks

par Junyoung Chung

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

April, 2018

© Junyoung Chung, 2018.

Résumé

L'apprentissage profond a poussé l'étude des réseaux de neurones profonds et a conduit à des avancées significatives dans plusieurs domaines d'application de l'apprentissage automatique. Dans ce manuscrit, nous nous concentrons sur un sous-ensemble de ces modèles: les réseaux de neurones récurrents. La structure spécifique de ces réseaux fait de la modélisation de données temporelles, telles que les données textuelles ou de parole, leur point fort. Plusieurs domaines d'applications plus pratiques en font d'ailleurs leur composante essentielle, incluant la reconnaissance de parole, la synthèse de parole, la traduction automatique et l'apprentissage par renforcement. Cependent, notre compréhension des réseaux de neurones récurrents reste loin d'être complète, plusieurs problèmes spécifiques aux réseaux de neurones n'ont pas encore été résolus. Ce manuscrit inclut plusieurs pistes conduisant à des architectures de réseaux de neurones récurrents profond et multi-échelle.

Dans un premier article, nous présentons un réseau récurrent pouvant contrôler son propre schéma de connectivité entre couches représentant des indices temporels consécutifs. Ces connexions entre temps consécutifs ne se limitent pas juste à des connexions sur un même niveau mais permettent à des couches de haut niveau de communiquer avec des couches plus basses, et vice-versa. Un ensemble d'unités barrage paramétriques est appris afin d'ouvrir ou de fermer les connexions qui conduisent le signal des couches précédentes temporellement. Nous étudions comment les informations des couches ascendantes sont utiles dans la modélisation de dépendences temporelles.

Dans un deuxième article, nous étudions un système de traduction automatique neuronale reposant sur un décodeur par caractère. Ce travail est motivé par une question fondamentale: peut-on générer une suite de caractères en guise de traduction au lieu d'une suite de mots ? Afin de répondre à cette question, nous avons utilisé une architecture simple à deux niveaux et conçu un réseau de neurones plus complexe traitant les dynamiques rapides et lentes séparemment. Ce nouveau modèle se base sur l'idée d'utiliser des composantes évoluants à différentes échelles afin de traiter les dépendences temporelles.

Nous étudions dans un troisième article une architecture de réseau récurrent permettant la découverte des structures latentes d'une séquence. Cette nouvelle architecture s'appuie sur un ensemble d'unités limites permettant une segmentation en morceaux pertinents. Le réseau de neurones récurrent met à jour chaque couche cachée sur un rythme différent dépendant de l'état de ces unités limites. L'inclusion de ces unités limites nous permet de définir un nouveau mécanisme de miseàjour utilisant trois différents types d'opérations: chaque couche peut soit copier l'état précédent, mettre à jour cet état ou évacuer cet état vers l'état de plus haut niveau et réinitialiser le contexte.

Enfin, un quatrième article se penche sur l'utilisation de variables latentes dans un réseau de neurones récurrent. La complexité et le rapport signal-bruit de données séquentielles comme la parole rendent la découverte de structures pertinentes dans ces données difficiles. Nous proposons une extension récurrente de l'auto-encodeur variationel afin d'introduire ces variables latentes et améliorer la performance dans la modélisation séquentielle, incluant celle de la parole et de l'écriture manuscrite.

Mots-clefs: apprentissage profond, réseaux de neurones, réseaux de neurones récurrents, réseaux de neurones récurrents hiérarchiques, réseaux de neurones récurrents multi-échelle, modélisation du langage, traduction automatique, synthèse de parole, synthèse d'écriture manuscrite, auto-encodeur variationel.

Summary

Deep learning is a study of deep artificial neural networks that has led to several breakthroughs in many machine learning applications. In this thesis, a subgroup of deep learning models, known as recurrent neural networks is studied in depth. Recurrent neural networks are special types of artificial neural networks that possess more strength in modelling temporal structures of sequential data such as text and speech. Recurrent neural networks are used as the core module of many practical applications including speech recognition, text-to-speech, machine translation, machine comprehension, and question and answering. However, our understanding of recurrent neural networks is still limited, and some inherent problems with recurrent neural networks remain unresolved. This thesis includes a series of studies towards deep multiscale recurrent neural networks and novel architectures to overcome the inherent problems of recurrent neural networks.

In the first article, we introduce a deep recurrent neural network that can adaptively control the connectivity patterns between layers at consecutive time steps. The recurrent connections between time steps are not only restricted to self-connections as the conventional recurrent neural networks do, but a higher-level layer can connect to the lower-level layers, and vice-versa. A set of parametrized scalar gating units is learned in order to open or close the connections that carry the feedback from the layers at the previous time step. We investigate how the top-down information can be useful for modelling temporal dependencies.

In the second article, we study a neural machine translation system that exploits a character-level decoder. The motivation behind this work is to answer a fundamental question: can we generate a character sequence as translation instead of a sequence of words? In order to answer this question, we design a naive two-level recurrent neural network and a more advanced type of recurrent neural network that tries to capture faster and slower components separately with its layers. This proposed model is based on an idea of modelling time dependencies with multiple components that update with different timescales.

In the third article, we investigate a framework that can discover the latent hierarchical structure in sequences with recurrent neural networks. The proposed framework introduces a set of boundary detecting units that are used to detect terminations of meaningful chunks. The recurrent neural network updates each hidden layer with different timescales based on the binary states of these boundary detecting units. The inclusion of the boundary detectors enables us to implement a novel update mechanism using three types of different operations. Each layer of the recurrent neural network can choose either to completely *copy* the dynamic state, to *update* the state or to *flush* the state to the upper-level layer and reset the context.

Finally, in the fourth article, we study an inclusion of latent variables to recurrent neural networks. The complexity and high signal-to-noise ratio of sequential data such as speech make it difficult to learn meaningful structures from the data. We propose a recurrent extension of the variational auto-encoder in order to introduce high-level latent variables to recurrent neural networks and show performance improvements on sequences modelling tasks such as human speech signals and handwriting examples.

Keywords: deep learning, neural networks, recurrent neural networks, hierarchical recurrent neural networks, multiscale recurrent neural networks, language modelling, machine translation, speech generation, handwriting generation, variational auto-encoders

Contents

List of Figures

List of Tables

List of Abbreviations

AE	Auto-Encoder
ANN	Artificial Neural Network
BLEU	Bi-Lingual Evaluation Understudy
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
DAE	Denoising Auto-Encoder
DBN	Deep Belief Network
DNN	Deep Neural Network
GD	Gradient Descent
GMM	Gaussian Mixture Model
GRU	Gated Recurrent Unit
HMM	Hidden Markov Model
KL	Kullback-Leibler Divergence
LSTM	Long Short-Term Memory
MLP	Multilayer Perceptron
NMT	Neural Machine Translation
NTM	Neural Turing Machine
RBM	Restricted Boltzmann Machine
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
VAE	Variational Auto-Encoder

Acknowledgments

Mila has been such a stimulating environment to pursue an academic endeavor, and this doctoral program has been a great opportunity to work on exciting research ideas on the topics that I am truly passionate about. I could not possibly have completed this journey without the consistent support of many amazing people, friends, colleagues, advisors and my family.

I want to thank my advisor, Professor Yoshua Bengio, for welcoming me to his lab and guiding me to become an independent researcher. I feel lucky to have been part of his lab when great research break-throughs happened. Being Yoshua's student is one of the best things happened in my life. His passion for artificial intelligence and creativity has influenced me a lot. I would also like to thank Professor Aaron Courville and Professor Roland Memisevic who have always shared their knowledge and gave invaluable advice to me.

I would like to thank the organizers of the lab, Linda Penthière, Myriam Côte, Céline Bégin, Angela Fahdi, for helping to build such a great academic environment. I would like to thank the staff members of the lab, Frédéric Bastien, Arnaud Bergeron, Pascal Lamblin, Simon Lefrançois, for your infinite support and for taking care of all research tools and hardwares perfectly as one can imagine.

I am thankful to have met many amazing people, who are too numerous to list without the fear of omission. I want to first thank friends, collaborators, colleagues who have shared their time with me in Mila: Adriana Romero, Alexandre De Brebisson, Amjad Almahairi, Asja Fisher, Bart van Merriënboer, Çaglar Gülçehre, César Laurent, Daniel Jiwoong Lim, David Krueger, David Warde-Farley, Devon Hjelm, Dmitriy Serdyuk, Dustin Webb, Dzmitry Bahdanau, Faruk Ahmed, Francesco Visin, Guillaume Alain, Guillaume Dejardins, Harm De Vries, Heeyoul Choi, Inchul Song, Jacob Athul Paul, João Felipe Santos, Jörg Bornschein, José Rodriguez Sotelo, Kelvin Xu, Kratarth Goel, Kundan Kumar, Kyle Kastner, Kyunghyun Cho, Laurent Dinh, Li Yao, Mathieu Germain, Mehdi Mirza, Mohammad Pezeshski, Negar Rostamzadeh, Nicollas Ballas, Orhan Firat, Philemon Brakel, Razvan Pascanu, Saizheng Zhang, Samira Shabanian, Sina Honari, Soroush Mehri, Sungjin Ahn, Taesup Kim, Thomas Le Paine, Tim Cooijmans, Vincent Duoulin, Yann Dauphin, Yaroslav Ganin, Ying Zhang and Zhouhan Lin.

I am especially in debt of my collaborators: Kyunghyun Cho, Çaglar Gülçehre, Kyle Kastner, Laurent Dinh, Harm de Vries, Kratarth Goel and Sungjin Ahn.

I had opportunities to visit many different research groups and expand my experience during my doctoral program. Thanks to Hany Hassan Awadalla and Jacob Devlin for hosting me as an intern at Microsoft Research. Thanks to Wolfgang Macherey and Melvin Johnson for hosting me as an intern at Google Translate. Thanks to Kyunghyun Cho for inviting me as an academic visitor to his lab when he first joined the New York University as a faculty. It has been a pleasure to work with you.

Finally, I want to thank my parents and family for their endless support and encouragement to pursue my dream. Thank you for believing in me.

1 Introduction

Deep learning is a research field in machine learning, and a renewed name for neural networks, which has become an extremely popular and important technique to solve artificial intelligence (AI) problems. Ironically, the main building blocks for deep learning, neural networks, have existed for many years (McCulloch and Pitts, 1943; Rosenblatt, 1958; Rumelhart et al., 1988). In the past decade, deep learning has been popularized by a renewed interest that was raised by theoretical advance and breakthroughs (Hinton et al., 2006; Bengio et al., 2007). Deep learning has been adopted to many different applications such as computer vision, natural language processing, robotics, speech and bio-medicine. In computer vision, deep learning has brought significant improvements to object classification (Krizhevsky et al., 2012), object detection (He et al., 2016) and image caption generation (Vinyals et al., 2015; Xu et al., 2015). In machine translation, deep learning based approaches have marked large improvement in translation quality in terms of fluency, BLEU score[1] and the performance evaluated by humans. This deep learning based translation is known as neural machine translation (Kalchbrenner and Blunsom, 2013; Sutskever et al., 2014; Cho et al., 2014), and many companies are shifting, if not already transferred, from traditional statistical machine translation systems to ones based on neural machine translation (Wu et al., 2016; Johnson et al., 2016). Deep learning is also combined with reinforcement learning, bringing astonishing results for an artificial Go agent that can beat professional human Go players (Silver et al., 2016), an artificial agent that can play Atari games in super-human level (Mnih et al., 2016), and an agent that can control robots in physics environments (Lillicrap et al., 2015; Duan et al., 2016; Heess et al., 2016, 2017). Machines are able to generate human voice and audio that is difficult to distinguish from the real ones using deep learning models (Oord et al., 2016). Deep learning is also changing the fields of speech recognition (Amodei et al., 2016) and text-to-

1. A precision based metric used to evaluate language generation tasks such as machine translation and image caption generation.

speech (Wang et al., 2017)

This thesis focuses on advances in a particular type of deep learning model called a recurrent neural network. Recurrent neural networks are especially important in aforementioned problems because they can learn the dynamics of sequential data. The first chapter provides some background material on machine learning in general. The second chapter delves deeper into the fundamentals of deep learning. The rest of the thesis describes my contributions in recurrent neural networks modelling, using new architectures to better understand and model sequential data.

1.1 Introduction to Machine Learning

Machine learning is a research field focusing on designing learning algorithms, which learn to solve complex problems from examples rather than programming with explicit rules. This chapter will introduce the basic concepts of machine learning. It must be noted that this chapter will not fully cover the whole field of machine learning but will present the necessary metarial to understand the remainder of this thesis. *Learning* is perhaps the most important concept and is the main goal of machine learning. Learning can mean different things in different tasks, and the types of learning can be categorized into three groups:

— *Supervised learning* is a task of inferring a function with a desired behavior from labeled data. The labeled data means that each training example is a pair that consists of an input object and a desired output value.

— *Unsupervised learning* is a task of inferring a function to describe the hidden structure of data without labeled examples.

— *Reinforcement learning* is a task of learning a policy for how to act to maximize the expected rewards that could only be given sparsely and with delays.

In this chapter, the basic idea of the *learning from data* is described, along with definitions for supervised and unsupervised learning, which will be used in the subsequent chapters.

1.2 Learning

The main goal of machine learning is to find a model with a desired behavior f^* in solving a task T. The model can be obtained from data $\mathcal{D}$ by optimizing a properly chosen objective function to solve the task T (Mitchell, 1997). Learning makes machine learning fundamentally different from other approaches that rely on explicitly programming the desired behavior f^*. The model is equipped with limited resources, e.g., computational units, optimization algorithm, tunable parameters, and has to synchronize its behavior f to f^*. Here, we will cover parametric models, where a model has a finite number of parameters, and a parametric model family $\mathcal{F}$ can be defined as a set of parametric functions, $\mathcal{F} = \{f_\theta | \theta \in \Theta\}$, where $\theta \in \mathbb{R}^n$.

Learning is usually conducted as an optimization problem that minimizes a discrepancy measure between the behavior of the model f and the desired behavior f^*. The discrepancy measure is often called a loss function $\mathcal{L}$. The loss function $\mathcal{L}$ should be designed with care and should take into account the task T. Now learning can be defined in a clear way: searching for the best behavior $\hat{f}$ within a set of functions $\mathcal{F}$, by optimizing the loss function $\mathcal{L}$ to perform the task T given a set of examples from the data $\mathcal{D}$.

We can write the above statement in a mathematical form:

$$\mathcal{R}_{exp}(f_\theta) := \mathbb{E}_{(x,y)\sim p(X,Y)}[\mathcal{L}(f_\theta(x), y)], \tag{1.1}$$

$$\hat{f}_\theta \leftarrow \operatorname*{arg\,min}_{f_\theta \in \mathcal{F}} \mathcal{R}_{exp}(f_\theta) \tag{1.2}$$

where $\mathbb{E}_{(x,y)\sim p(X,Y)}$ stands for expectation over x and y sampled from the true distribution $p(X,Y)$ of the data $\mathcal{D}$, and θ denotes the parameters of the model. Here, x is an input and y is a desired outcome for x. For example, x is an input object, and y is a class label of the object given to the model. In this case, the model becomes a classifier, and the desired behavior f becomes predicting a correct class label of an input object. The loss function can vary depending on what kind of learning problem is being solved.

In practice, the model has only limited access to the data $\mathcal{D}$, hence, if there is a set of a finite number of examples $\tilde{\mathcal{D}}$, we can approximate Eq. 1.1 as:

$$\mathcal{R}_{emp}(f_\theta) := \frac{1}{|\tilde{\mathcal{D}}|} \sum_{(x,y)\in\tilde{\mathcal{D}}} \mathcal{L}(f_\theta(x), y), \tag{1.3}$$

$$\hat{f}_\theta \leftarrow \arg\min_{f_\theta \in \mathcal{F}} \mathcal{R}_{emp}(f_\theta), \tag{1.4}$$

where $|\tilde{\mathcal{D}}|$ is the number of examples in $\tilde{\mathcal{D}}$. $\mathcal{R}_{exp}$ is called an expected loss and $\mathcal{R}_{emp}$ is called an empirical loss (Vapnik, 2013). Almost all problems in supervised learning and unsupervised learning have limited access to $\mathcal{D}$, and operate on samples $\tilde{\mathcal{D}}$, instead of the true distribution. This procedure of finding the model that fits the desired behavior best is called *training* or *fitting* the data. Throughout the remainder of this thesis, we will use the terminology *training* to refer to this procedure.

In machine learning, we care about the generalization performance of the model to unseen data. This is because we have access to only a limited amount of data, and the model has to learn general behavior, without learning things which are specific only to the training examples. We measure the error between the behavior of the model and desired behavior on unseen data, which is known as generalization error. Back to the classification example, the generalization error is a measure of how accurately a model can predict correct labels for previously unseen data. In general, a new set of examples, different from the ones used in training, is used to measure the generalization performance. For this reason, the data $\tilde{\mathcal{D}}$ should be separated into three groups. The first group is a set of examples that is provided to train the model, and this set is called a *training set*. The second group is a set of examples that is provided to evaluate the training procedure of the model and to find the optimal values for tunable parameters that control the learning process to draw the best outcome. We call the second set of examples, the *validation set*. The last group is a set of examples that is never exposed to the model during the training, and because it is used to measure the generalization error, this set is called a *test set*.

Training a differentiable parametric model is usually done in an iterative fashion using optimization algorithms such as gradient descent [2]. We can assess how well

2. Each update in gradient descent is proportional to the negative of the gradient of the loss with respect to the parameter: $\theta_{\tau+1} = \theta_\tau - \eta \nabla_\theta f(\theta_\tau)$.

the model is fitting the data by observing the training error, $\mathcal{R}_{emp}$ on the training set, and the generalization error, $\mathcal{R}_{emp}$ on the test set. If both training error and the generalization error do not decrease, we can say that the model is *under-fitting* the data. That is, our choice of the model is rather too restricted, or difficult to search the function or solution that is close to f^*. If the training error decreases, but the generalization error increases, we say that the model is *over-fitting.* This is the case when the model has the ability to model complex solutions such that the model parameters can completely fit the training set instead of learning the true mapping from the input to the target objective. Here, when a model is 'highly flexible' means that the model has a large degree of freedom and enough parameters to approximate a complicated function. However, we cannot directly use the generalization error during training. The validation error, $\mathcal{R}_{emp}$ on the validation set, is used to find at which step during iterative training procedure, the generalization error increases. The validation set is also used to find a set of extra parameters which do not change during training but still influence the optimization and overall training of the model. These extra parameters are called *hyperparameters*, for example, the learning rate of the optimization algorithm.

The main goal of learning in machine learning is to obtain a model that has a good solution to the desired behavior which can generalize to unseen data. To have the best model, it is important not to over-fit to the training examples, or under-fit the possible solution. One way to prevent *over-fitting* is to use a low capacity model such that the model does not have enough flexibility to completely memorize the training set, so instead it is forced to learn underlying structure among the training examples. However, there is a chance for a simple model suffers from *under-fitting* if the model cannot approximate the function or hypothesis to match the desired behavior.

1.3 Regularization

If a model is sufficiently flexible, it may not *under-fit* the training data, however, the model can choose to match its parameters to fit all the specific details of the training examples instead of learning a general behavior for solving the task. Another option to prevent *over-fitting* is to introduce an extra term that restricts the model from exploring certain regions of the function space $\mathcal{F}$ leading to overfitting. Therefore, it can help the model to increase its generalization ability. This additional term is called a *regularization term*, and one can come up with different forms of regularization terms which are preferable for gradient based optimization. Now, the objective function with the additional regularization term can be written mathematically as:

$$\hat{f}_\theta \leftarrow \arg\min_{f_\theta \in \mathcal{F}} \mathcal{R}_{emp}(f_\theta) + \lambda\Omega(\theta), \tag{1.5}$$

where λ is a hyperparameter that weights the influence of the regularization term, and $\Omega(\theta)$ is the regularization term such as ℓ^2-norm of the model parameters. If the optimization algorithm conducts minimization over the objective function, e.g., gradient descent, $\lambda\Omega(\theta)$ should have a non-negative value. Adding a regularization term is just one way of regularization, others include Bayesian methods, adding noise to parameters, dropout, early stopping, et cetera.

1.4 Supervised Learning

Supervised learning is the process of finding a mapping function that takes input as a vector and outputs a desired response vector, $f_\theta : \mathbb{R}^d \rightarrow \mathbb{R}^t$. In supervised learning, examples are usually given as pairs:

$$\tilde{\mathcal{D}} = \left\{(x^i, y^i) \sim p(X, Y)\right\}_{i=1,\dots,|\tilde{\mathcal{D}}|},$$

where x is the input, and y is the desired output of x. The model is asked to predict a correct output value or label for an input object.

1.4.1 Probabilistic Classification

Classification is the task of predicting a correct class label Y of a given input object X. If the predicted value for Y is a probability distribution, then f_θ is said to be a probabilistic classifier. Usually, f_θ is modelled to learn a conditional probability $p(Y = y \mid X = x)$. That is, the output of the model $f_\theta(x)$ is a probability of x being an object of a class label, y. Classification is used as a whole or part of many applications in machine learning. For example, classification is used in image classification (Krizhevsky et al., 2012), speech recognition (Amodei et al., 2016), machine translation (Sutskever et al., 2014; Cho et al., 2014) and image caption generation (Vinyals et al., 2015; Xu et al., 2015).

Logistic Regression Logistic regression is a popular algorithm for binary classification. That is, the cardinality of the class label is two, i.e., $t = 2$. X is mapped into a probability of Y being 1 using an affine transformation followed by the logistic function:

$$p_\theta(Y = 1 \mid X = \mathbf{x}) = f_\theta(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b),$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$, $\mathbf{x} \in \mathbb{R}^d$ is the input, $\mathbf{w} \in \mathbb{R}^d$ is a weight vector and $b \in \mathbb{R}^1$ is a bias.

Other Classification Algorithms If the number of output classes is larger than 2, i.e., $|Y| > 2$, we consider using a softmax function instead of the logistic function. Now, the affine transformation is followed by the softmax function as below:

$$\psi(W\mathbf{x} + \mathbf{b}) = \frac{\exp(W\mathbf{x} + \mathbf{b})}{\sum_{k=1}^{K} \exp(W_k\mathbf{x} + \mathbf{b})},$$

where $W \in \mathbb{R}^{n \times d}$ is the weight matrix, $\mathbf{b} \in \mathbb{R}^n$ is the bias. Then, the probability of x being the j-th class is expressed as:

$$p_\theta(Y = j \mid X = \mathbf{x}) = \psi(W\mathbf{x} + \mathbf{b})_j,$$

where the subscript j denotes the j-th element of the vector $\psi(W\mathbf{x} + \mathbf{b})$.

1.4.2 Regression

Regression is used in problems where the desired outputs are continuous variables. For learning a regression function, it is common to make an assumption that $p_\theta(Y \mid X)$ follows a continuous distribution such as a Gaussian distribution. If we make an assumption that only the mean is predicted by the model f_θ and the variance is fixed to 1, the loss function simplifies to the mean squared error function:

$$\mathcal{L}(f_\theta(x), y) = (y - f_\theta(x))^2. \tag{1.6}$$

However, we can also model the variance σ_θ^2 using the probability density function of the Gaussian distribution:

$$\mu_\theta = f_\theta(x), \tag{1.7}$$

$$\sigma_\theta^2 = g_\theta(x), \tag{1.8}$$

$$\mathcal{L}(f_\theta(x), y) = \mathcal{N}(y - \mu_\theta, \sigma_\theta^2), \tag{1.9}$$

where $\mathcal{N}(\mu, \sigma)$ is a Gaussian distribution parmetrized by the mean μ and the standard deviation σ.

There are many other supervised learning algorithms such as nearest neighborhood or support vector machines that take a non-parametric approach, however, we will not cover them in this thesis.

1.5 Unsupervised Learning

Unsupervised learning is used when there is no label assigned to each example, but somehow the model has to discover the hidden structure underlying the set of training examples. Many machine learning algorithms are used for unsupervised learning. Because the target output is not specified, there is often a wider range of choice in the design of the model structure and objective functions.

One common task is *clustering* the given data points into a fixed number of groups. There are algorithms such as k-means clustering and Gaussian mixture models (GMMs), which perform this task. k-means clustering is one of the most popular machine learning algorithms for clustering In k-means clustering, centroids of clusters are learned, and a data point has to fall into one of these clusters. Clustering can provide an estimate of the number of major modes in the data.

Another popular task is *reconstruction*. Algorithms such as principal component analysis (PCA), auto-encoders (Baldi and Hornik, 1989) and denoising auto-encoders (Vincent et al., 2008) fall into this category. A good reconstruction under compression or noise requires a good representation of the data, hence, these algorithms are forced to learn representation of the data. Some algorithms are combined with additional regularization terms in order to enhance the ability to extract meaningful structures (Vincent et al., 2008; Rifai et al., 2011) when the compressed representation of the input has less dimensionality than the input, the algorithm is called a *dimensionality reduction* algorithm. Dimensionality reduction also includes algorithms such as t-SNE (Maaten and Hinton, 2008). Dimensionality reduction algorithms are very useful when we visualize the data or parameters of the model in a low-dimensional space, preferably a 2-D space.

In *density estimation*, the goal is to estimate the underlying distribution $p(X)$ in which the training examples were sampled from. Algorithms such as resctricted Boltzmann machines (RBMs) (Smolensky, 1986), variational auto-encoders (VAEs) (Kingma and Welling, 2013; Rezende et al., 2014) fall into this category. Some of the research projects in this thesis are unsupervised learning algorithms, which will be discussed in detail in the following chapters.

2 Deep Learning

Deep learning is a research field aiming at learning multiple levels of abstraction and feature representation for data using deep neural networks (Bengio, 2009; LeCun et al., 2015; Schmidhuber, 2015; Goodfellow et al., 2016). In the past decade, deep learning has been popularized as a powerful and scalable approach to solve complicated machine learning tasks. The main advantage of a deep learning approach is that deep architectures help the models[1] to better handle *the curse of dimensionality* (Bellman, 2013) by learning a composite and hierarchical feature representation of the data. The curse of dimensionality refers to a phenomenon of explosive increase of the complexity when the dimensionality of the data increases.

The effectiveness of deep architectures was first shown by the success of deep belief networks (DBNs) (Hinton et al., 2006) and stacked auto-encoders (Bengio et al., 2007). In these works, a greedy layer-wise learning strategy is used to train a deep architecture one layer at a time. This approach was first used for learning features in unsupervised learning tasks, however, it was also shown to be useful to initialize a deep neural network with the parameters of deep belief networks or stacked auto-encoders (Erhan et al., 2010). This technique is referred to pre-training, and it is used in many machine learning applications.

Deep neural networks are a key component of many machine learning systems, including speech recognition (Dahl et al., 2010; Mohamed et al., 2011; Graves et al., 2013; Graves and Jaitly, 2014), handwriting recognition (Graves, 2013), object classification (Krizhevsky et al., 2012), image caption generation (Vinyals et al., 2015), machine translation (Sutskever et al., 2014; Cho et al., 2014; Bahdanau et al., 2015) and playing games (Mnih et al., 2015; Silver et al., 2016).

Recent advance in deep neural networks is not limited to architectural improvements, but it also includes optimization methods (Amari, 1998; Martens, 2010; Pascanu and Bengio, 2013; Dauphin et al., 2014; Kingma and Ba, 2014; Desjardins

1. In deep learning, they are usually neural networks.

et al., 2015; Dauphin et al., 2015), development of activation functions[2] (Nair and Hinton, 2010; Glorot et al., 2011; Zeiler and Fergus, 2013; Goodfellow et al., 2013; Klambauer et al., 2017), initialization techniques (Glorot and Bengio, 2010; Saxe et al., 2013; He et al., 2015), regularization methods (Srivastava et al., 2014; Ioffe and Szegedy, 2015; Ba et al., 2016) and theoretical improvements (Bengio et al., 1994; Pascanu et al., 2012, 2013; Bengio et al., 2015; Gal and Ghahramani, 2015; Zhang et al., 2016; Dinh et al., 2017).

There are two other common operations in deep learning, beyond the linear transformations discussed in chapter 1, which are *convolution* and *recurrence*. *Convolutional neural networks* (Fukushima, 1980; LeCun et al., 1989) are specialized in modelling data with spatial relationships in the data such as images or video frames. *Recurrent neural networks* are specialized in modelling dynamics of sequences such as text and speech (Jordan, 1997; Elman, 1990; Hochreiter and Schmidhuber, 1997). These two operations can be combined in order to model spatio-temporal features in the data, such as video (Yao et al., 2015).

This thesis includes research projects that contribute ideas on constructing deep recurrent neural networks that can extract hierarchical and decomposable representation from sequential data. Techniques for capturing multiple timescale representation of sequences will be described in subsequent chapters.

2.1 Neural Networks

Artificial neural networks (ANNs) are in a family of models that share specific features, which are described in this section. The first ever neural network architecture appeared in Rosenblatt (1958), called the *Perceptron*. The Perceptron was later extended to the multilayer Perceptron in Rumelhart et al. (1988). After the Perceptron, various types of ANNs have been proposed, including convolutional neural networks (Fukushima, 1980; LeCun et al., 1989), Hopfield networks (Hopfield, 1982), self-organizing maps (Kohonen, 1982), Boltzmann machines (Ackley et al., 1985), restricted Boltzmann machines (Smolensky, 1986), auto-encoders (Baldi and Hornik, 1989), sigmoid belief networks (Neal, 1992) deep

2. The activation functions will be described with more details in the following sections.

belief networks (Hinton et al., 2006), deep Boltzmann machines (Salakhutdinov and Hinton, 2009), stacked denoising auto-encoders (Vincent et al., 2010), variational auto-encoders (Kingma and Welling, 2013; Rezende et al., 2014), adversarial networks (Goodfellow et al., 2014), highway networks (Srivastava et al., 2015) and residual networks (He et al., 2015). We will refer to ANNs as neural networks in the remainder of this thesis.

2.1.1 Neurons

A neuron is a basic computational unit of neural networks. A neuron h takes an input and applies an affine transformation to the input using its parameters, the weight vector $\mathbf{w} \in \mathbb{R}^d$ and the bias term $b \in \mathbb{R}^1$:

$$z = \mathbf{w}^\top \mathbf{x} + b, \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input, and z is the outcome of the affine transformation. A non-linear function $\phi(x)$ is applied to z:

$$h = \phi(z), \tag{2.2}$$

where h is the activation or state of the neuron h for the input x. The non-linear function $\phi(x)$ is especially important when neurons are stacked to form a hierarchy, which is able to represent complicated functions. The non-linearity function is also called an activation function. z is called the pre-activation since it is a value before applying the activation function. Figure 2.1 depicts a graphical view of a neuron.

2.1.2 Universal Approximator Theorem

There is a rich literature discussing the universality[3] of neural networks. It has been proven that a single layer feedforward neural network[4] with a sufficient number of neurons that uses a sigmoid function as the non-linearity is a universal approximator of any continuous function to arbitrary precision (Hornik et al., 1989; Cybenko, 1989; Funahashi, 1989; Barron, 1993). The universal approximator

3. If a computational model is universal, it can simulate any other computational model to arbitrary precision.

4. Feedforward neural networks will be discussed in more depth in section 2.2.

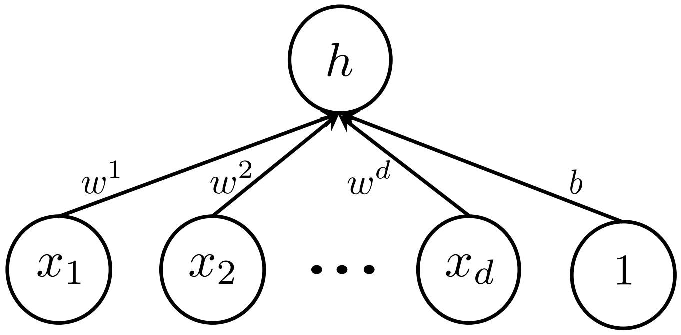


Figure 2.1: A neuron: a neuron is connected with the inputs using its weights. The last connection is the bias term.

theorem states that increasing the number of hidden units can decrease the error of the approximation.

2.1.3 Why Deep Neural Networks?

The universal approximator property of a single layer neural network is one of the most cited theoretical results that justifies the use of neural networks as function approximators. However, there are some practical issues to train a single layer neural network with a large number of neurons due to the limited computational resource. Also, there is a lack of training data to train a neural network with a large number of parameters. Deep neural networks are argued as a more practical approach to function approximators, although there is only a few theoretical results for deep neural networks (Le Roux and Bengio, 2010; Montufar et al., 2014). In Eldan and Shamir (2016), it was shown that a two-layer feedforward neural network requires an exponential number of neurons in the input dimension d to approximate a simple function on $\mathbb{R}^d$ that can be approximated by a three-layer feedforward with a polynomial number of neurons in d.

2.1.4 Activation Functions

Activation functions introduce non-linearities to the neural networks. It is important to include non-linearities in deep neural networks to approximate complex functions. The most commonly used activation functions are the logistic function (sigmoid), hyperbolic tangent function (tanh), softplus function (softplus) and rectified linear units (ReLU, Nair and Hinton, 2010):

$$\mathrm{sigmoid}(x) = \frac{1}{1+\exp(-x)}, \tag{2.3}$$

$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}, \tag{2.4}$$

$$\mathrm{softplus}(x) = \log(1+\exp(x)), \tag{2.5}$$

$$\mathrm{ReLU}(x) = \begin{cases} x, \text{if } x > 0, \\ 0, \text{else } x <= 0. \end{cases} \tag{2.6}$$

Note that in recurrent neural networks, more complicated activation functions are commonly used (Hochreiter and Schmidhuber, 1997; Cho et al., 2014; Chung et al., 2014). These will be described in later sections.

2.1.5 Hidden Layers

Neurons which receive the same inputs can form a layer. We commonly call this set of neurons, a hidden layer, and each neuron in the layer, a hidden unit. Usually, there are no connections between neurons in the same layer. There are three types of layers in neural networks: an input layer, a hidden layer and an output layer (see Figure 2.2). The input layer is where the input values are placed. Sometimes the input data is preprocessed into a specific range or in a special format (e.g., one-hot encoding) depending on which kind of tasks is in our hands. The output layer is where the model can place its outputs. The outputs of the output layer should be in the same range with that of the targets. In deep neural networks, it is common to stack more than two hidden layers to form a deep architecture.

2.1.6 Backpropagation Algorithm

The error surface defined by the loss function of a highly non-linear function (e.g., deep neural networks) is often non-convex[5], and there is no guarantee that there is only one global solution to solve this problem. In addition, no analytical solution to the optimization problem is known. Therefore, one has to consider

5. Note that the error surface of highly non-linear loss function is not always non-convex.

using an iterative optimization algorithm such as gradient descent methods:

$$\theta_{\tau+1} = \theta_\tau - \eta \nabla_\theta f(\theta_\tau), \tag{2.7}$$

where η is a learning rate, and τ denotes the time step.

The backpropagation algorithm is used to compute the gradient of the loss with respect to each parameter of neural networks. It computes the gradient in different layers by propagating the error signal that originates from the loss function in a backward direction. The update of parameters are conducted after two steps: (1) a forward propagation and (2) a backward propagation. During the forward propagation, all the outputs of the neurons from the input layer to the output layer are computed and stored in a memory. During the backward propagation, the partial derivatives of the neurons are computed, and the gradients are propagated from the output layer to the input layer using the chain rule. When using the backpropagation algorithm, it is very important to ensure that all the operations conducted in the neural networks are differentiable.

In deep neural networks, there are a couple of issues with backpropagation when the number of hidden layers increases. One of the problems is vanishing gradients, where the magnitudes of the error signals diminish. This is because the non-linear functions map the inputs into a small output range, e.g., a sigmoid activation function maps the input into an output range of $[0, 1]$. For certain range of the inputs, the derivative of the corresponding output range is close to 0. Vanishing gradients become more problematic as the hidden layers are stacked. Piecewise linear functions such as rectified linear units (ReLU) (Nair and Hinton, 2010) can alleviate vanishing gradients. Other problems are related to optimization, due to the high non-convexity of the error function of deep neural networks. Developing and analyzing optimization routines for neural networks is an active field of research with a practical benefit for many applications (Zeiler, 2012; Dauphin et al., 2014; Kingma and Ba, 2014).

2.1.7 Pre-training

Pre-training was introduced in Hinton et al. (2006) to train deep belief networks (DBNs). In Hinton et al. (2006), DBNs are trained by a greedy layer-wise learning algorithm and fine-tuned with supervision using labeled data. While Bengio et al.

(2007) pre-trained a stack of auto-encoders and then fine-tuned them as a supervised MLP. They also introduced supervised greedy layer-wise pre-training for deep architectures. Pre-training is a two-step approach to train neural networks, and can be seen as a particular regularization scheme for transferring domain knowledge (Donahue et al., 2014). When it is too difficult to directly train a neural network on a given dataset, one can pre-train the model on the same dataset or on another dataset with more examples. Once the pre-training is done, the model can be fine-tuned on the target dataset. Each step can be used for unsupervised learning tasks or supervised ones.

2.1.8 Shortcuts or Linear Paths for Gradients

It is sometimes useful to introduce some linear connections between layers of deep neural networks, where gradients can flow without any obstacle (Raiko et al., 2012; He et al., 2015). It is also possible to connect the input layer directly to any intermediate hidden layer, and any hidden layer to the output layer (Graves, 2013). Srivastava et al. (2015) applied the gating mechanism used in gated recurrent neural networks (Hochreiter and Schmidhuber, 1997; Cho et al., 2014) to feedforward networks. Using gated feedforward connections, the model is much more resistant to vanishing gradients and is able to train with hundreds of hidden layers.

2.2 Feedforward Neural Networks

A feedforward neural network refers to a network of neurons that do not form feedback connections. It has a simple structure which maps the inputs to the target outputs. Multilayer Perceptrons (MLPs) (Rumelhart et al., 1988) and convolutional neural networks (CNNs) (Fukushima, 1980; LeCun et al., 1989) are typical examples of feedforward neural networks. Feedforward neural networks are usually used to process non-sequential data, however, they are frequently combined with recurrent neural networks to form a more complicated structure (Pascanu et al., 2013).

Figure 2.2 shows a graphical view of an MLP. The MLP is an extension to the Perceptron (Rosenblatt, 1958) with additional hidden layers for internal hidden

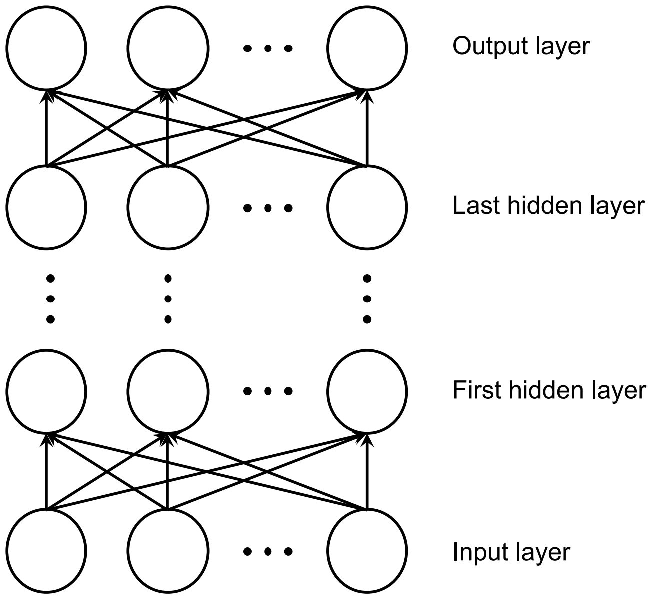


Figure 2.2: Multilayer Perceptron: multiple numbers of hidden layers are stacked to form a multilayer architecture. The activation functions of neurons can be specified by the practitioner, however, the neurons at the output layers should output a range that matches with the targets. The bias terms are omitted for brevity.

representation of the data. The neurons of the input layer are observed variables that represent the values of the input data. A hidden layer receives the output of the preceding layer as the input and computes the activations of its neurons. The activations output by the neurons are then propagated to the next layer. The neurons of the hidden layers (hidden units) are unobserved variables, and they are usually treated as deterministic variables. However, it is not necessary to assume that the hidden units are always deterministic. In Neal (1992); Tang and Salakhutdinov (2013), the hidden units are considered as stochastic variables. The output layer represents the final output of the model, and it is important to ensure that the range of the outputs matches the range of the targets. For instance, if the task is a binary classification problem, then the number of output neurons is one, and the MLP outputs a probability value in the range of $[0, 1]$. The MLP architecture with more than 1 or 2 hidden layers is what is typically known as deep feedforward neural networks. The activations functions of the intermediate hidden units and the output neurons can be modified by practitioners based on the tasks or the types of input data.

Feedforward neural networks are acyclic graphs, that is, the arrows from the input layer to the output layer define a computational graph without any cycles. Signals are propagated only once to the neurons, and there is no connectivity pattern that direct the outputs to the previous neurons, which would form a feedback loop.

A feedforward neural network with multiple hidden layers can learn a more complicated function than a shallow neural network. Each hidden layer contains many hidden units, therefore, the activation of ℓ-th hidden layer can be computed as:

$$\mathbf{h}^{\ell} = \phi^{\ell}(W^{\ell}\mathbf{h}^{\ell-1} + \mathbf{b}^{\ell}), \tag{2.8}$$

where $\phi^{\ell}(x)$ is the activation function of the layer ℓ, $W^{\ell} \in \mathbb{R}^{|\mathbf{h}^{\ell}|\times|\mathbf{h}^{\ell-1}|}$ is the weight matrix, $\mathbf{b}^{\ell} \in \mathbb{R}^{|\mathbf{h}^{\ell}|}$ is the bias term, $\mathbf{h}^{\ell-1} \in \mathbb{R}^{|\mathbf{h}^{\ell-1}|}$ is the activation of the previous layer, and $\mathbf{h}^0 = \mathbf{x}$. Each hidden unit in the layer ℓ detects features from its input and maps them into activation. The overall mapping function of the feedforward neural network becomes a composite function after computing each mapping function in a sequential manner.

2.3 Recurrent Neural Networks

A recurrent neural network (RNN) is a special type of neural network that can handle variable-length inputs and outputs. An RNN processes a sequence of an arbitrary length by recursively applying a state-transition function to its dynamic state whenever it reads each symbol from an input sequence. Unlike hidden Markov models (HMMs), which also have a dynamic state that is defined in a discrete state-space S, RNNs exploit a distributed representation[6]. Therefore, the expressive power of the dynamic state of RNNs is much richer than HMMs. For comparison, the state-space of HMMs is usually defined as a single set of mutually exclusive states, but in RNNs, the state-space is defined by a number of neurons. If there

6. Each input to a neural network is represented by many hidden units (features), and each hidden unit contributes in representing many possible inputs.

are N neurons, and each neuron can represent a value in $\{0, 1\}$, the size of the state-space of an RNN becomes $|S| = 2^N$. Since the value of each neuron in RNNs are typically in the range of $[-1, 1]$, the state-space can become even larger.

HMMs are probabilistic sequential models that interpret a sequence of observations as probabilistic outcomes of the dynamic state, which is hidden. One issue with HMMs is the difficulty of scaling up the algorithm since the size of the state-space is limited, otherwise the computational complexity of the inference can be dramatically increased as well as the number of the parameters in the transition matrix (Viterbi, 1967). In addition, the Markov property [7] restricts the dynamic state of HMMs to become only dependent on the last previous state, which is a disadvantage when modelling long-term time dependencies. In contrast, RNNs have a memory in the form of dynamic state that enables the model to preserve long-range context. Also, the expressive power of the model increases exponentially when the number of neuron increases, but the model can be trained efficiently via *backpropagation through time* (Rumelhart et al., 1986).

2.3.1 Simple Recurrent Neural Networks

Elman networks (Elman, 1990) are known as simple recurrent neural networks. A simple recurrent neural network consists of a hidden layer, a context layer and an output layer. These layers are expressed mathematically as follow:

$$\mathbf{z}_t = W^{in}\mathbf{x}_t + W^{rec}\mathbf{h}_{t-1} + \mathbf{b}, \tag{2.9}$$

$$\mathbf{h}_t = \phi(\mathbf{z}_t), \tag{2.10}$$

$$\mathbf{v}_t = W^{out}\mathbf{h}_t + \mathbf{c}, \tag{2.11}$$

$$\mathbf{y}_t = \phi(\mathbf{v}_t), \tag{2.12}$$

where $\mathbf{x}_t \in \mathcal{R}^{|\mathcal{X}|\times 1}$ is an input vector, $\mathbf{h}_t \in \mathcal{R}^{|\mathcal{H}|\times 1}$ is the hidden state. For the hidden layer, $W^{in} \in \mathcal{R}^{|\mathcal{H}|\times|\mathcal{X}|}$ is a weight matrix that transforms the input vector, $W^{rec} \in \mathcal{R}^{|\mathcal{H}|\times|\mathcal{H}|}$ is a recurrent transition matrix that transforms the previous hidden state, and $\mathbf{b} \in \mathcal{R}^{|\mathcal{H}|\times 1}$ is the bias term of the hidden layer. For the output layer, $W^{out} \in \mathcal{R}^{|\mathcal{Y}|\times|\mathcal{H}|}$ is a weight matrix connected to the hidden state, and $\mathbf{c} \in \mathcal{R}^{|\mathcal{Y}|\times 1}$

7. The Markov property assumes that the future state is independent of the past given the present state: $p(x_{t+1}|x_1, \ldots, x_t) = p(x_{t+1}|x_t)$.

is the bias term of the output layer. We use the subscript to index the time step, and the superscript to denote which layer the parameters belong to. In Jordan networks (Jordan, 1986), the previous output is used in Eq. 2.9 instead of the previous hidden state. The activation of the hidden layer at time step t is an outcome of a state-transition function that takes the current input symbol $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ as inputs. Here, ϕ is a hyperbolic tangent function, but other types of activation functions can be used as well.

The inputs and targets to RNNs are expected to be data points that are sequences. That is, an input example is a sequence given as $\mathbf{x} = \{x_1, x_2, \ldots, x_T\}$, and T is the length of the sequence. Let us think about a language modelling task, where $\mathbf{x}$ is a sequence of discrete symbols, and each symbol indicates either a character or a word. $\mathbf{x}$ is generated from a stochastic process $x_t \sim p(X_t \mid X_{1:t-1})$. The probability of the sequence $p(\mathbf{x})$ can be factorized using the chain rule:

$$p(\mathbf{x}) = p(X_1 = x_1)p(X_2 = x_2 \mid X_1 = x_1) \cdots p(X_T = x_T \mid X_1 = x_1, \cdots X_{T-1} = x_{T-1}).$$

An RNN reads each symbol x_t at each time step and predicts the next symbol x_{t+1}. That is, the conditional distribution $p(X_t \mid X_{1:t-1})$ is now modelled as a one-step process of the RNN using Eqs. 2.9–2.12. RNNs trained to model the probability of sequences are also called generative RNNs.

It is not necessary to assume that the data is always a time-series. RNNs can be applied to images as well, by treating pixels as symbols that occur sequentially in the spatial domain (Graves et al., 2007; van den Oord et al., 2016).

2.3.2 Backpropagation Through Time

Backpropagation Through Time (BPTT) (Rumelhart et al., 1986; Werbos, 1988, 1990; Williams and Peng, 1990) is an extension of backpropagation algorithm, which is a popular method to compute the gradients for parameters of RNNs. Figure 2.3 depicts an RNN with a recurrent connection in its hidden layer. An RNN can be unfolded in the temporal axis given a finite number of time steps. A graphical view of an unfolded RNN is shown in Figure 2.4. For each time step, there is a clone of the hidden state, and the recurrent connection of the RNN is replaced by direct connections. This results in a very deep neural network on the temporal axis, where the parameters are shared across time. The BPTT is performed by applying backpropagation algorithm to the unfolded RNN.

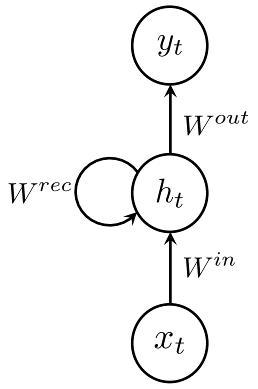


Figure 2.3: An RNN depicted as a folded form.

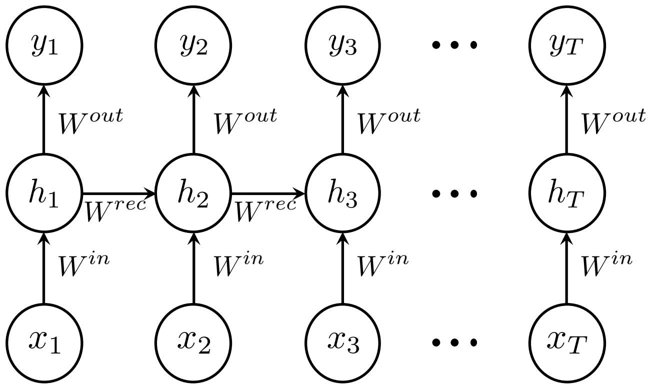


Figure 2.4: A folded RNN can be unfolded for finite number of time steps.

The BPTT computes the gradients of the RNNs in two stages as feedforward neural networks do. We start by computing the forward pass of the computational graph using Eqs. 2.9–2.12. The loss function is computed at every time step, and the error signal is propagated to y_t.

The gradients of pre-activations or activations $\mathbf{v}_t$, $\mathbf{h}_t$, $\mathbf{z}_t$ are defined as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t}\phi^{'}(\mathbf{v}_t), \tag{2.13}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{v}_t}W^{out} + \frac{\partial \mathcal{L}}{\partial \mathbf{z}_{t+1}}W^{rec}, \tag{2.14}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}\phi^{'}(\mathbf{z}_t). \tag{2.15}$$

The gradients of the parameters W^{in}, W^{rec} and W^{out} are computed as:

$$\frac{\partial \mathcal{L}}{\partial W^{out}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \mathbf{v}_t}\mathbf{h}_t^{\top}, \tag{2.16}$$

$$\frac{\partial \mathcal{L}}{\partial W^{in}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_t}\mathbf{x}_t^{\top}, \tag{2.17}$$

$$\frac{\partial \mathcal{L}}{\partial W^{rec}} = \sum_{t=1}^{T-1} \frac{\partial \mathcal{L}}{\partial \mathbf{z}_{t+1}}\mathbf{h}_t^{\top}. \tag{2.18}$$

$$\tag{2.19}$$

The gradients for the bias terms can be computed in the same way. From Eq. 2.18, we can draw an explanation why exploding and vanishing gradients are critical in recurrent neural networks.

2.3.3 Vanishing and Exploding Gradients

A major problem in training RNNs is that vanishing gradients can become much severe than feedforward neural networks. In addition to vanishing gradients, we start to observe a new problem, which is exploding gradients (Hochreiter, 1991; Bengio et al., 1994). It is straightforward to understand RNNs as extremely deep feedforward neural networks when they are unfolded in time. The final output of an RNN is a composition of a large number of non-linear transformations. Even though each non-linear transformation is a smooth function, the composition function is not necessarily a smooth function. Therefore, the derivatives of the composition function become either very small or very large.

In Bengio et al. (1994), exploding or vanishing gradients refers to an exponential increase or decrease of the norm of gradients due to a very long recurrence. Hochreiter (1991); Bengio et al. (1994) and Pascanu et al. (2012) provide theoretical analysis on these problems. The gradient computed at time step T is propagated to an arbitrary time step τ by following rule:

$$\frac{\partial \mathcal{L}_T}{\partial \mathbf{h}^{\tau}} = \frac{\partial \mathcal{L}_T}{\partial \mathbf{h}_{T-1}} \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{T-2}} \cdots \frac{\partial \mathbf{h}_{\tau+1}}{\partial \mathbf{h}_{\tau}}, \tag{2.20}$$

$$= \frac{\partial \mathcal{L}_T}{\partial \mathbf{h}_{T-1}} \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{\tau}}. \tag{2.21}$$

The term $\frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{\tau}}$ from Eq. 2.21 is computed as a product of Jacobians via the chain rule:

$$\frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{\tau}} = \prod_{i=\tau}^{T-2} \frac{\partial \mathbf{h}_{i+1}}{\partial \mathbf{h}_i} = \prod_{i=\tau}^{T-2} W^{rec} \mathrm{diag}(\phi^{'}(\mathbf{h}_i)), \tag{2.22}$$

where W^{rec} is the recurrent transition matrix, and $\phi^{'}(\mathbf{h}_i)$ are diagonalized derivatives of $\mathbf{h}_i$. Each Jacobian is equal to a multiplication of W^{rec} and $\phi^{'}(\mathbf{h}_i)$. If the spectral radius[8] of each Jacobian is smaller than 1, the gradient can decrease to 0 exponentially fast as $T - \tau$, if the spectral radius of each Jacobian is larger than 1,

8. The largest absolute value of eigenvalues of a square matrix.

the gradient can increase towards to an exponentially large number as $T - \tau$ (Pascanu et al., 2012). These phenomena are called vanishing and exploding gradients, respectively. For exploding gradients, there is an empirical solution to mitigate the problem, which is known as *gradient clipping* (Tomáš, 2012; Pascanu et al., 2012; Graves, 2013). For vanishing gradients, there are some approaches using orthogonality of the recurrent transition matrix in order to preserve the long-range context (Le et al., 2015; Arjovsky et al., 2016; Vorontsov et al., 2017). Vanishing gradients are more difficult to resolve, and RNNs can fail to capture the long-term temporal dependencies. The following sections will cover a few approaches to overcome vanishing gradients. However, both vanishing and exploding gradients are not the only bottlenecks of training RNNs. In subsequent chapters, the contributed articles will show the other challenges in training RNNs and discuss how to better handle these issues.

2.3.4 Long Short-Term Memory

The Long Short-Term Memory (LSTM) unit was initially proposed by Hochreiter and Schmidhuber (1997) in order to mitigate vanishing and exploding gradients problems. Since then, a number of minor modifications to the original LSTM unit have been made (Gers et al., 2000; Graves and Schmidhuber, 2005; Zaremba and Sutskever, 2014). The core idea of LSTMs is an introduction of memory cells with a self-loop connection with a constant value 1. Because a memory cell only runs through linear operations, it can store a bit of information for an arbitrary long period without suffering too much from vanishing gradients. Figure 2.5 depicts a graphical view of a single LSTM unit.

An LSTM unit has a set of sigmoidal gating units: an *input* gate i_t, a *forget* gate f_t, and an *output* gate o_t. For the memory cell, we use c_t as the notation. The memory cell conveys the context of an LSTM unit, while the gates control the amount of update and emission of the context stored in the memory cell. A context vector of the memory cells is updated as a weighted sum between the previous context vector and the new context vector. This can be expressed in a mathematical form as:

$$\mathbf{c}_t = \mathbf{f}_t \, \mathbf{c}_{t-1} + \mathbf{i}_t \, \tilde{\mathbf{c}}_t, \tag{2.23}$$

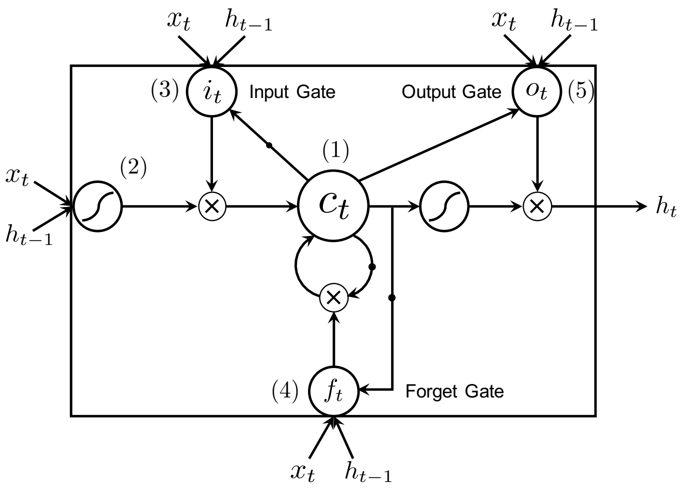


Figure 2.5: LSTM: (1) the memory cell is placed in the center. (2) The new context is computed and multiplied by the (3) input gate. The previous state of the memory cell is multiplied by the (4) forget gate and accumulated to the new context. The state of memory cell is updated by the new context. Finally, the new state of the memory cell is multiplied by the (5) output gate, and the hidden activation of the LSTM is computed. A closed dot is used to indicate shifting the time to backward by 1.

where $\tilde{\mathbf{c}}_t$ is the new context vector and can be computed as:

$$\tilde{\mathbf{c}}_t = \tanh\left(W^{xc}\mathbf{x}_t + W^{hc}\mathbf{h}_{t-1} + \mathbf{b}^c\right). \tag{2.24}$$

The input gates control how much the new context should be reflected to the memory cell, and the forget gates control how much an old context should be forgotten. These gates are computed from the previous hidden state, the current input and the last state of the memory cell:

$$\mathbf{i}_t = \sigma\left(W^{xi}\mathbf{x}_t + W^{hi}\mathbf{h}_{t-1} + W^{ci}\mathbf{c}_{t-1} + \mathbf{b}^i\right), \tag{2.25}$$

$$\mathbf{f}_t = \sigma\left(W^{xf}\mathbf{x}_t + W^{hf}\mathbf{h}_{t-1} + W^{cf}\mathbf{c}_{t-1} + \mathbf{b}^f\right), \tag{2.26}$$

where $\mathbf{i}_t$ denotes the input gates, and $\mathbf{f}_t$ denotes the forget gates. Here, $\sigma(x)$ is an element-wise sigmoid function. $\mathbf{x}_t$, $\mathbf{h}_{t-1}$ and $\mathbf{c}_{t-1}$ are the input vector, the previous hidden state and the previous cell context vector of the LSTM units, respectively. Note that W^{ci} and W^{cf} are the weight matrices that connect the previous context vector in the inference of the input and forget gates, and they are diagonal matrices.

This suggests that the context in each memory cell does not directly interact with the update of other cells.

Once the memory cell contexts of the LSTM units are updated, the hidden activations $\mathbf{h}_t$ are computed as:

$$\mathbf{h}_t = \mathbf{o}_t \tanh\left(\mathbf{c}_t\right).$$

The output gates $\mathbf{o}_t$ control to which degree the context in each memory cell is exposed. The output gates are dependent on the current input vector, the previous hidden state and the new context vector.

$$\mathbf{o}_t = \sigma\left(W^{xo}\mathbf{x}_t + W^{ho}\mathbf{h}_{t-1} + W^{co}\mathbf{c}_{t-1} + \mathbf{b}^o\right). \tag{2.27}$$

Here, W^{co} is a diagonal matrix.

The three gates and the memory cell enable the LSTM unit to adaptively *forget*, *memorize* or *emit* the context. If the context stored in the memory cell is deemed important, the forget gate will be closed (have a value close to 1) and hold the content across a long period of time, which is essentially capturing the long-range context. The unit can also decide to wipe out the memory content, if the content is not useful anymore. If the current input is not important, the input gate can be opened (have a value close to 0) and not change the context of the memory cell. Because of this adaptive memory capability and effective performance in practice, LSTMs are used in many sequence modelling tasks.

2.3.5 Gated Recurrent Units

A Gated Recurrent Unit (GRU) (Cho et al., 2014) is a lightweight version of an LSTM. Unlike LSTMs, GRUs do not have an explicit cell structure, and they are designed to adaptively *reset* or *update* the memory content. A GRU has a *reset* gate r_t and an *update* gate z_t. Because a GRU does not have a memory cell, the context is fully exposed at each time step, and the new context is determined by performing leaky integration between the previous context and the new context. Figure 2.6 shows a graphical view of a GRU.

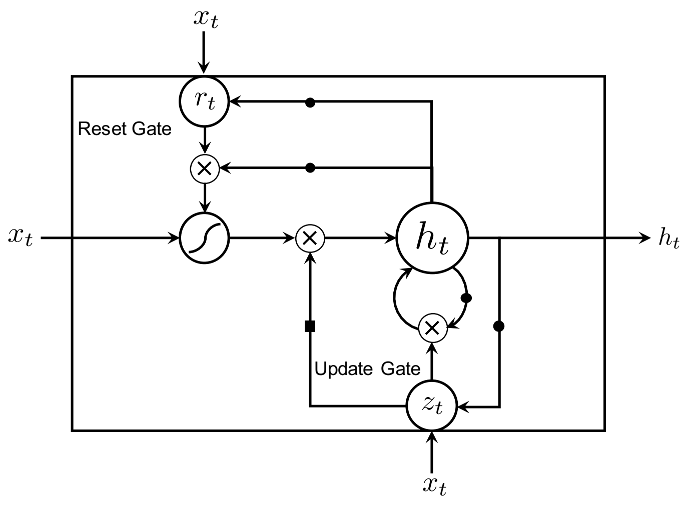


Figure 2.6: GRU: a GRU consists of a reset gate and an update gate. A closed circle is used to indicate a time-shift of 1 in backward. A closed square is an operation equivalent to $f(x) = 1 - x$.

At each time step t, the hidden state of GRUs is computed as:

$$\mathbf{h}_t = (1 - \mathbf{z}_t)\, \mathbf{h}_{t-1} + \mathbf{z}_t\, \tilde{\mathbf{h}}_t, \tag{2.28}$$

where $\mathbf{h}_{t-1}$ and $\tilde{\mathbf{h}}_t$ are the previous context and the new candidate context, respectively. The update gates $\mathbf{z}_t$ control in what amount the previous context should be forgotten and in what amount the new context should be added. The update gates are computed by taking the previous hidden state $\mathbf{h}_{t-1}$ and the current input vector $\mathbf{x}_t$ as inputs:

$$\mathbf{z}_t = \sigma \left(W^z \mathbf{x}_t + U^z \mathbf{h}_{t-1} + \mathbf{b}^z \right). \tag{2.29}$$

The new memory context $\tilde{\mathbf{h}}_t$ is computed similarly to the conventional state-transition function of RNNs shown in Eq. 2.9:

$$\tilde{\mathbf{h}}_t = \tanh \left(W \mathbf{x}_t + \mathbf{r}_t \odot U \mathbf{h}_{t-1} + \mathbf{b} \right), \tag{2.30}$$

where $\odot$ denotes an element-wise multiplication.

A major difference between the GRU and the state-transition function of the simple RNN is that the activations of the previous time step $\mathbf{h}_{t-1}$ are modulated by the reset gates $\mathbf{r}_t$. By using these reset gates, GRUs preserve the previous hidden state if they are deemed necessary or ignore the previous hidden state if they are no longer needed. The reset gates take the previous hidden state and the current input vector as inputs, and can be computed as:

$$\mathbf{r}_t = \sigma \left(W^r \mathbf{x}_t + U^r \mathbf{h}_{t-1} + \mathbf{b}^r \right). \tag{2.31}$$

This update mechanism helps GRUs to capture the long-range context as the LSTMs do. GRUs are often computationally more efficient than LSTMs due to the smaller number of parameters and computations. They perform well on many sequence modelling tasks (Chung et al., 2014).

2.3.6 Recurrent Neural Networks with External Memories

LSTMs and GRUs were proposed to alleviate vanishing gradients in order to capture long-term temporal dependencies. Another approach to capturing long-term temporal dependencies is using an external memory that can be read or written by a pre-defined access protocol. The external memory can be implemented as matrices (Weston et al., 2014; Graves et al., 2014), neural stack architectures (Grefenstette et al., 2015) or complex-valued vectors (Danihelka et al., 2016). An RNN is typically used as a controller that reads or writes the content of the external memory.

A neural turing machine (NTM) (Graves et al., 2014) consists of a controller network and a memory bank. The controller network reads and writes heads using a fully differentiable addressing mechanism, which is based on normalized weights over all memory addresses. NTMs can be trained by gradient descent and can perform tasks such as copying, sorting and associative recall from examples. NTMs and LSTMs can perform comparably on these tasks, however, NTMs that have external memory show better generalization ability to longer sequences.

In Associative LSTMs (Danihelka et al., 2016), the memory term is implemented as a sum of key-value pairs, where each key-value pair is a complex multiplication between a key and an input vector. The context in the memory can be retrieved by multiplying a complex conjugate of the key, that is associated with the target context, to the associative memory. In order to reduce the noise in retrievals, associative LSTMs make multiple copies for each context item in the memory and the final retrieval is an average of these copies.

2.3.7 Sequence to Sequence Models

Broadly speaking, a sequence-to-sequence model is an RNN based model that is used to learn a mapping function from a sequence to another sequence. Sequence-to-sequence models were first proposed in neural machine translation (NMT) (Cho et al., 2014; Sutskever et al., 2014), but their applications are not only limited to NMT. Technically, sequence-to-sequence models can be applied to any task requiring multiple outputs for an input such as image caption generation (Xu et al., 2015), video caption generation (Yao et al., 2015), text-to-speech (Wang et al., 2017; Sotelo et al., 2017) and speech recognition (Chan et al., 2015), where the goal is to transform a source sequence into a target sequence.

A sequence-to-sequence model is constructed as a composite of an encoder RNN and a decoder RNN. The encoder RNN first reads a source sequence $\mathbf{x} = \{x_1, \ldots, x_{T_x}\}$ and summarizes it into a context vector $\mathbf{c}$. The decoder RNN is a conditional generative RNN that models a conditional distribution $P(\mathbf{y} \mid \mathbf{x})$, where the context is provided instead of the source sequence $\mathbf{x}$. The last hidden state of the encoder RNN is usually given as the context (Cho et al., 2014; Sutskever et al., 2014). The decoder RNN reads each symbol from a target sequence $\mathbf{y} = \{y_1, \ldots, y_{T_y}\}$ and predicts the next target symbol. In sequence-to-sequence model, the lengths of the source sequence and the target sequence can be different. Sometimes, the difference can be quite large, e.g., the source side is a sequence of words, and the target side is a sequence of characters (Chung et al., 2016). Figure 2.7 depicts a graphical view of a sequence-to-sequence model.

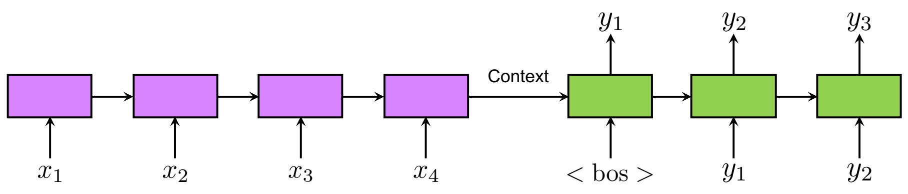


Figure 2.7: Sequence-to-Sequence Model: a sequence-to-sequence model with a single layer encoder RNN and a single layer decoder RNN.

One issue with the sequence-to-sequence model is that the long-range context is not well captured due to the practical difficulties of training RNNs. It was shown in NMT that the translation quality drops quickly when the length of the source sequences increases (Bahdanau et al., 2015).

Attention Mechanism In order to overcome this limitation, one can use a content-based *attention mechanism*, which is also called a *soft alignment mechanism* (Bahdanau et al., 2015). Unlike the naive sequence-to-sequence model, a sequence-to-sequence model with the attention mechanism retains the encoder states. Whenever the decoder processes each symbol in the target sequence, attention coefficients over the encoder states are computed at each time step, and the context is obtained as a weighted sum of the encoder states using the attention coefficients as weights. Figure 2.8 visualizes a graphical view of a sequence-to-sequence model with an attention mechanism. More details on the model architecture and equations are described in chapter 6.

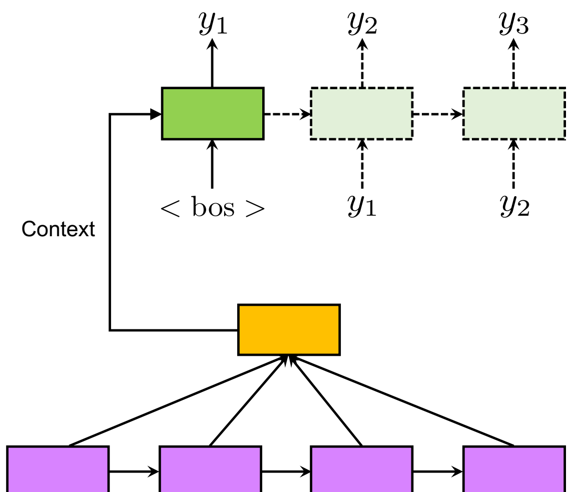


Figure 2.8: Sequence-to-Sequence Model with Attention Mechanism: a sequence-to-sequence model with a single layer encoder RNN, a single layer decoder RNN and an attention module. Whenever the decoder processes each symbol, the context is obtained as a weighted sum of the encoder states.

2.3.8 Hierarchical Recurrent Neural Networks

One of the desired properties of deep neural networks is learning a decomposable and hierarchical representation of data. Unfortunately, it is not clear how RNNs can learn such hierarchical representation of sequences since there are two directions to consider, one from the input layer to the output layer, and another along the temporal axis. For images, deep convolutional neural networks can capture different levels of spatial relationships. The lower-level layers tend to detect fine-scale features such as edges and corners, and higher-level layers tend to capture more abstract and coarse-scale features such as partial or full object shapes (Lee et al., 2009). The same thing could happen for the sequences and RNNs, but now the spatial relationships are replaced by temporal dependencies. In El Hihi and Bengio (1995), the authors propose stacking multiple RNNs on top of each other, and let them update the hidden states in different timescales. The motivation behind this is to show that the temporal dependencies form a hierarchical structure. This type of RNN, which assigns different update frequency to each of the hidden layers, is called a multiscale RNN. The model has one or a small number of time-delay factors that prevent the hidden layers from changing their hidden states at the same frequency. The key idea is to separate the temporal dependencies by length and model them differently.

Multiscale RNNs can provide a few advantages compared to standard RNNs that do not assign explicit timescales for updating the hidden layers. For example, multiscale RNNs compute fewer matrix multiplications by updating the higher-level layers less frequently, allowing multiscale RNNs to be computationally cheaper than standard RNNs. Secondly, by updating the hidden state less frequently, especially at the higher-level layers, which model long-term temporal dependencies, multiscale RNNs can mitigate the vanishing gradients problem. Multiscale RNNs also allow more predictable separation of explanatory factors than standard RNNs since different layers can be set to run on different explicit timescales. In some cases, the timescales can be parametrized and learned meaning the timescales are no longer fixed variables and explicitly given, however, we still know that the upper-level layer is always changing the hidden state slower than the lower-level ones due to the structure of the model.

There are mainly two approaches to implement multiscale RNNs. The first approach is to consider the timescales as hyperparameters. That is, each hidden layer of the RNN is updated with a pre-defined schedule, e.g., at every n time steps. One issue with this approach is that fixed timescales are not suited to the case where different segments in the hierarchical structure have variable lengths. The second approach is to learn the timescales with parameters. Learning the timescales may fit the purpose of the multiscale approach better, however, it can introduce more challenges than the first approach. It is difficult to obtain the training signal for learning the timescales. Imagine implementing a detector associated with each hidden layer that finds the right times to update the hidden state. This detector needs boundary information in order to learn a termination of a meaningful chunk. For some domains such as text, it is relatively easy to obtain the boundary information. We can use word tokenizers to obtain the word-level boundaries, punctuation marks to obtain the sentence-level boundaries, and line changes to obtain the paragraph-level boundaries. However, in most cases, the boundary information is not trivially extracted, meaning one has to invent an algorithm that can discover the boundaries by itself. The detected boundaries have to be represented as discrete variables in order to perform the conditional computation when updating the hidden state. This leads to the more economic use of computational resources, but introducing discrete variables makes it harder to compute gradients using backpropagation through time. The multiscale approach by learning the timescales is a challenging problem. This approach aligns with the main contribution of this thesis, and it will be studied in depth over the chapter 4,6 and 8.

3 Prologue to First Article

3.1 Article Details

Gated Feedback Recurrent Neural Networks. Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho and Yoshua Bengio, *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015).*

Personal Contribution.
Yoshua Bengio provided me his intuition on multiscale RNNs before starting this work. Inspired by Yoshua Bengio and his student's earlier work, and the clockwork RNN, I came up with this idea of allowing all possible recurrent connections between hidden layers of deep RNNs and learning them. Kyunghyun Cho helped me with shaping the idea with more concrete plans from the initial stage. Caglar Gulcehre spent a lot of time with me for the experimental design. I implemented the algorithms and did all of the experiments reported in the paper. I also participated heavily to the writing with Yoshua Bengio and Kyunghyun Cho, who contributed significantly to the introductory part.

3.2 Context

This paper was motivated by two earlier works, which are El Hihi and Bengio (1995) and Koutník et al. (2014). The idea of allowing the top-down connection was inspired by the clockwork RNN (Koutník et al., 2014), then later it was extended in this work to allow all recurrent connections between layers. This idea can be also related to skip-connections (Graves, 2013). In skip-connections, intermediate hidden layers can be connected to the output layer, and the input layer can be connected to any intermediate hidden layer. Skip-connections are helpful when the network is deep by providing a side road to the gradients. Here in this work, skip-connections are applied in a recurrent fashion with gating units that control the connections, and also top-down connections are allowed.

3.3 Contributions

Introducing a sigmoidal switch unit for each connection between layers was inspired by the gated RNNs. The gated RNNs refer to RNNs that use GRUs or LSTMs as the recurrent hidden units. A switch unit can control the traffic of the information between layers and can increase the modelling power compared to a model that has a deterministic connectivity pattern between hidden layers.

We observed that these extra connections with gating units help the model to perform well on the language modelling and program evaluation tasks (Zaremba and Sutskever, 2014). We achieved state-of-the-art results on the Hutter dataset at the moment this paper was published. One drawback of GF-RNNs is that it is difficult to explicitly show whether each layer is updating at a different timescale. Later with the hierarchical multiscale RNNs (Chung et al., 2016), we demonstrated a more improved model that shows an empirical evidence that the hidden layers are updated at different timescales.

4 Gated Feedback Recurrent Neural Networks

4.1 Introduction

Recurrent neural networks (RNNs) have been widely studied and used for various machine learning tasks which involve sequence modeling, especially when the input and output have variable lengths. Recent studies have revealed that RNNs using gating units can achieve promising results in both classification and generation tasks (see, e.g., Graves, 2013; Bahdanau et al., 2015; Sutskever et al., 2014).

Although RNNs can theoretically capture any long-term dependency in an input sequence, it is well-known to be difficult to train an RNN to actually do so (Hochreiter, 1991; Bengio et al., 1994; Hochreiter, 1998). One of the most successful and promising approaches to solve this issue is by modifying the RNN architecture e.g., by using a gated activation function, instead of the usual state-to-state transition function composing an affine transformation and a point-wise nonlinearity. A gated activation function, such as the long short-term memory (LSTM, Hochreiter and Schmidhuber, 1997) and the gated recurrent unit (GRU, Cho et al., 2014), is designed to have more persistent memory so that it can capture long-term dependencies more easily.

Sequences modeled by an RNN can contain both fast changing and slow changing components, and these underlying components are often structured in a hierarchical manner, which, as first pointed out by El Hihi and Bengio (1995) can help to extend the ability of the RNN to learn to model longer-term dependencies. A conventional way to encode this hierarchy in an RNN has been to stack multiple levels of recurrent layers (Schmidhuber, 1992; El Hihi and Bengio, 1995; Graves, 2013; Hermans and Schrauwen, 2013). More recently, Koutník et al. (2014) proposed a more explicit approach to partition the hidden units in an RNN into groups such that each group receives the signal from the input and the other groups at a separate, predefined rate, which allows feedback information between these partitions to be propagated at multiple timescales. Stollenga et al. (2014) recently showed the importance of feedback information across multiple levels of feature hierarchy, however, with feedforward neural networks.

In this paper, we propose a novel design for RNNs, called a gated-feedback RNN (GF-RNN), to deal with the issue of learning multiple adaptive timescales. The proposed RNN has multiple levels of recurrent layers like stacked RNNs do. However, it uses gated-feedback connections from upper recurrent layers to the lower ones. This makes the hidden states across a pair of consecutive time steps fully connected. To encourage each recurrent layer to work at different timescales, the proposed GF-RNN controls the strength of the temporal (recurrent) connection adaptively. This effectively lets the model to adapt its structure based on the input sequence.

We empirically evaluated the proposed model against the conventional stacked RNN and the usual, single-layer RNN on the task of language modeling and Python program evaluation (Zaremba and Sutskever, 2014). Our experiments reveal that the proposed model significantly outperforms the conventional approaches on two different datasets.

4.2 Background

4.2.1 Revisiting Recurrent Neural Network

An RNN is able to process a sequence of arbitrary length by recursively applying a transition function to its internal hidden states for each symbol of the input sequence. The activation of the hidden states at time step t is computed as a function f of the current input symbol $\mathbf{x}_t$ and the previous hidden states $\mathbf{h}_{t-1}$:

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

It is common to use the state-to-state transition function f as the composition of an element-wise nonlinearity with an affine transformation of both $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$:

$$\mathbf{h}_t = \phi\left(W\mathbf{x}_t + U\mathbf{h}_{t-1}\right), \tag{4.1}$$

where W is the input-to-hidden weight matrix, U is the state-to-state recurrent weight matrix, and ϕ is usually a logistic sigmoid function or a hyperbolic tangent function.

We can factorize the probability of a sequence of arbitrary length into:

$$p(x_1, \cdots, x_T) = p(x_1)p(x_2 \mid x_1) \cdots p(x_T \mid x_1, \cdots, x_{T-1}).$$

Then, we can train an RNN to model this distribution by letting it predict the probability of the next symbol x_{t+1} given hidden states $\mathbf{h}_t$ which is a function of all the previous symbols $x_1, \cdots, x_{t-1}$ and current symbol x_t:

$$p(x_{t+1} \mid x_1, \cdots, x_t) = g\left(\mathbf{h}_t\right).$$

This approach of using a neural network to model a probability distribution over sequences is widely used, for instance, in language modeling (see, e.g., Bengio et al., 2001; Mikolov, 2012).

4.2.2 Revisiting Gated Recurrent Neural Network

The difficulty of training an RNN to capture long-term dependencies has been known for long (Hochreiter, 1991; Bengio et al., 1994; Hochreiter, 1998). A previously successful approaches to this fundamental challenge has been to modify the state-to-state transition function to encourage some hidden units to adaptively maintain long-term memory, creating paths in the time-unfolded RNN, such that gradients can flow over many time steps.

Long short-term memory (LSTM) was proposed by Hochreiter and Schmidhuber (1997) to specifically address this issue of learning long-term dependencies. The LSTM maintains a separate memory cell inside it that updates and exposes its content only when deemed necessary. More recently, Cho et al. (2014) proposed a gated recurrent unit (GRU) which adaptively remembers and forgets its state based on the input signal to the unit. Both of these units are central to our proposed model, and we will describe them in more details in the remainder of this section.

Long Short-Term Memory

Since the initial 1997 proposal, several variants of the LSTM have been introduced (Gers et al., 2000; Zaremba et al., 2014). Here we follow the implementation provided by Zaremba et al. (2014).

Such an LSTM unit consists of a memory cell c_t, an *input* gate i_t, a *forget* gate f_t, and an *output* gate o_t. The memory cell carries the memory content of an LSTM unit, while the gates control the amount of changes to and exposure of the memory content. The content of the memory cell c_t^j of the j-th LSTM unit at time step t is updated similar to the form of a gated leaky neuron, i.e., as the weighted sum of the new content $\tilde{c}_t^j$ and the previous memory content c_{t-1}^j modulated by the input and forget gates, i_t^j and f_t^j, respectively:

$$c_t^j = f_t^j c_{t-1}^j + i_t^j \tilde{c}_t^j, \tag{4.2}$$

where

$$\tilde{\mathbf{c}}_t = \tanh\left(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1}\right). \tag{4.3}$$

The input and forget gates control how much new content should be *memorized* and how much old content should be *forgotten*, respectively. These gates are computed from the previous hidden states and the current input:

$$\mathbf{i}_t = \sigma\left(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1}\right), \tag{4.4}$$

$$\mathbf{f}_t = \sigma\left(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1}\right), \tag{4.5}$$

where $\mathbf{i}_t = \left[i_t^k\right]_{k=1}^p$ and $\mathbf{f}_t = \left[f_t^k\right]_{k=1}^p$ are respectively the vectors of the input and forget gates in a recurrent layer composed of p LSTM units. $\sigma(\cdot)$ is an element-wise logistic sigmoid function. $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are the input vector and previous hidden states of the LSTM units, respectively.

Once the memory content of the LSTM unit is updated, the hidden state h_t^j of the j-th LSTM unit is computed as:

$$h_t^j = o_t^j \tanh\left(c_t^j\right).$$

The output gate o_t^j controls to which degree the memory content is exposed. Similarly to the other gates, the output gate also depends on the current input and the previous hidden states such that:

$$\mathbf{o}_t = \sigma\left(W_o\mathbf{x}_t + U_o\mathbf{h}_{t-1}\right). \tag{4.6}$$

In other words, these gates and the memory cell allow an LSTM unit to adaptively *forget*, *memorize* and *expose* the memory content. If the detected feature, i.e., the memory content, is deemed important, the forget gate will be closed and carry the memory content across many time steps, which is equivalent to capturing a long-term dependency. On the other hand, the unit may decide to reset the memory content by opening the forget gate. Since these two modes of operations can happen simultaneously across different LSTM units, an RNN with multiple LSTM units may capture both fast-moving and slow-moving components.

Gated Recurrent Unit

The GRU was recently proposed by Cho et al. (2014). Like the LSTM, it was designed to adaptively *reset* or *update* its memory content. Each GRU thus has a *reset* gate r_t^j and an *update* gate z_t^j which are reminiscent of the forget and input gates of the LSTM. However, unlike the LSTM, the GRU fully exposes its memory content each time step and balances between the previous memory content and the new memory content strictly using leaky integration, albeit with its adaptive time constant controlled by update gate z_t^j.

At time step t, the state h_t^j of the j-th GRU is computed by:

$$h_t^j = (1 - z_t^j)h_{t-1}^j + z_t^j\tilde{h}_t^j, \tag{4.7}$$

where h_{t-1}^j and $\tilde{h}_t^j$ respectively correspond to the previous memory content and the new candidate memory content. The update gate z_t^j controls how much of the previous memory content is to be forgotten and how much of the new memory content is to be added. The update gate is computed based on the previous hidden states $\mathbf{h}_{t-1}$ and the current input $\mathbf{x}_t$:

$$\mathbf{z}_t = \sigma\left(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1}\right). \tag{4.8}$$

The new memory content $\tilde{h}_t^j$ is computed similarly to the conventional transition function in Eq. (4.1):

$$\tilde{\mathbf{h}}_t = \tanh\left(W\mathbf{x}_t + \mathbf{r}_t \odot U\mathbf{h}_{t-1}\right), \tag{4.9}$$

where $\odot$ is an element-wise multiplication.

One major difference from the traditional transition function (Eq. (4.1)) is that the states of the previous step $\mathbf{h}_{t-1}$ is modulated by the reset gates $\mathbf{r}_t$. This behavior allows a GRU to ignore the previous hidden states whenever it is deemed necessary considering the previous hidden states and the current input:

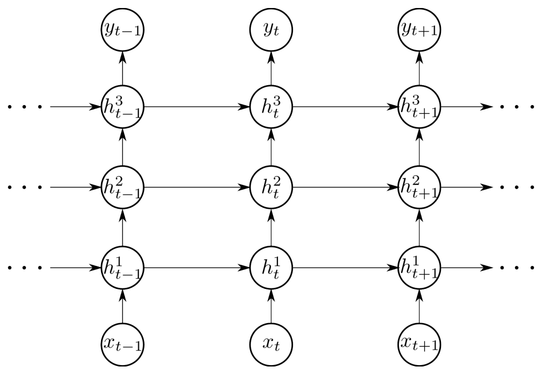
$$\mathbf{r}_t = \sigma\left(W_r\mathbf{x}_t + U_r\mathbf{h}_{t-1}\right). \tag{4.10}$$

The update mechanism helps the GRU to capture long-term dependencies. Whenever a previously detected feature, or the memory content is considered to be important for later use, the update gate will be closed to carry the current memory content across multiple time steps. The reset mechanism helps the GRU to use the model capacity efficiently by allowing it to reset whenever the detected feature is not necessary anymore.

4.3 Gated Feedback Recurrent Neural Network

Although capturing long-term dependencies in a sequence is an important and difficult goal of RNNs, it is worthwhile to notice that a sequence often consists of both slow-moving and fast-moving components, of which only the former corresponds to long-term dependencies. Ideally, an RNN needs to capture both long-term and short-term dependencies.

El Hihi and Bengio (1995) first showed that an RNN can capture these dependencies of different timescales more easily and efficiently when the hidden units of the RNN is explicitly partitioned into groups that correspond to different timescales. The clockwork RNN (CW-RNN) (Koutník et al., 2014) implemented this by allowing the i-th module to operate at the rate of 2^{i-1}, where i is a positive integer, meaning that the module is updated only when $t \bmod 2^{i-1} = 0$. This makes each module to operate at different rates. In addition, they precisely defined the connectivity pattern between modules by allowing the i-th module to be affected by j-th module when $j > i$.



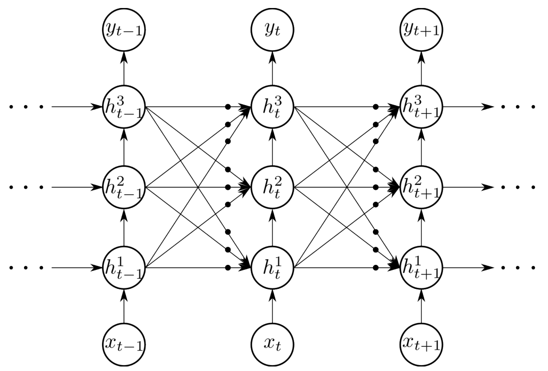


Figure 4.1: Illustrations of (a) conventional stacking approach and (b) gated-feedback approach to form a deep RNN architecture. Bullets in (b) correspond to global reset gates. Skip connections are omitted to simplify the visualization of networks.

Here, we propose to generalize the CW-RNN by allowing the model to adaptively adjust the connectivity pattern between the hidden layers in the consecutive time steps. Similar to the CW-RNN, we partition the hidden units into multiple modules in which each module corresponds to a different layer in a stack of recurrent layers.

Unlike the CW-RNN, however, we do not set an explicit rate for each module. Instead, we let each module operate at different timescales by hierarchically stacking them. Each module is fully connected to all the other modules across the stack and itself. In other words, we do not define the connectivity pattern across a pair of consecutive time steps. This is contrary to the design of CW-RNN and the conventional stacked RNN. The recurrent connection between two modules, instead, is gated by a logistic unit ($[0, 1]$) which is computed based on the current input and the previous states of the hidden layers. We call this gating unit a *global reset* gate, as opposed to a unit-wise reset gate which applies only to a single unit (See Eqs. (4.2) and (4.9)).

The global reset gate is computed as:

$$g^{i \to j} = \sigma\left(\mathbf{w}_g^{i \to j}\ \mathbf{h}_t^{j-1} + \mathbf{u}_g^{i \to j}\ \mathbf{h}_{t-1}^{*}\right),$$

where $\mathbf{h}_{t-1}^{*}$ is the concatenation of all the hidden states from the previous time step $t-1$. The superscript ${}^{i\to j}$ is an index of associated set of parameters for the transition from layer i in time step $t-1$ to layer j in time step t. $\mathbf{w}_g^{i\to j}$ and $\mathbf{u}_g^{i\to j}$ are respectively the weight vectors for the current input and the previous hidden states. When $j=1$, $\mathbf{h}_t^{j-1}$ is $\mathbf{x}_t$.

In other words, the signal from $\mathbf{h}_{t-1}^{i}$ to $\mathbf{h}_t^{j}$ is controlled by a single scalar $g^{i\to j}$ which depends on the input $\mathbf{x}_t$ and all the previous hidden states $\mathbf{h}_{t-1}^{*}$.

We call this RNN with a fully-connected recurrent transitions and global reset gates, a *gated-feedback RNN* (GF-RNN). Fig. 4.1 illustrates the difference between the conventional stacked RNN and our proposed GF-RNN. In both models, information flows from lower recurrent layers to upper recurrent layers. The GF-RNN, however, further allows information from the upper recurrent layer, corresponding to coarser timescale, flows back into the lower recurrent layers, corresponding to finer timescales.

In the remainder of this section, we describe how to use the previously described LSTM unit, GRU, and more traditional tanh unit in the GF-RNN.

4.3.1 Practical Implementation of GF-RNN

tanh **Unit.** For a stacked tanh-RNN, the signal from the previous time step is gated. The hidden state of the j-th layer is computed by

$$\mathbf{h}_t^{j} = \tanh\left(W^{j-1\to j}\mathbf{h}_t^{j-1} + \sum_{i=1}^{L} g^{i\to j} U^{i\to j}\mathbf{h}_{t-1}^{i}\right),$$

where L is the number of hidden layers, $W^{j-1\to j}$ and $U^{i\to j}$ are the weight matrices of the current input and the previous hidden states of the i-th module, respectively. Compared to Eq. (4.1), the only difference is that the previous hidden states are from multiple layers and controlled by the global reset gates.

Long Short-Term Memory and Gated Recurrent Unit. In the cases of LSTM and GRU, we do not use the global reset gates when computing the unit-wise gates. In other words, Eqs. (4.4)–(4.6) for LSTM, and Eqs. (4.8) and (4.10) for GRU are not modified. We only use the global reset gates when computing the new state (see Eq. (4.3) for LSTM, and Eq. (4.9) for GRU).

The new memory content of an LSTM at the j-th layer is computed by

$$\tilde{\mathbf{c}}_t^j = \tanh\left(W_c^{j-1\to j}\mathbf{h}_t^{j-1} + \sum_{i=1}^{L} g^{i\to j} U_c^{i\to j}\mathbf{h}_{t-1}^i \right).$$

In the case of a GRU, similarly,

$$\tilde{\mathbf{h}}_t^j = \tanh\left(W^{j-1\to j}\mathbf{h}_t^{j-1} + \mathbf{r}_t^j \odot \sum_{i=1}^{L} g^{i\to j} U^{i\to j}\mathbf{h}_{t-1}^i \right).$$

4.4 Experiment Settings

4.4.1 Tasks

We evaluated the proposed GF-RNN on character-level language modeling and Python program evaluation. Both tasks are representative examples of discrete sequence modeling, where a model is trained to minimize the negative log-likelihood of training sequences:

$$\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} -\log p\left(x_t^n \mid x_1^n, \ldots, x_{t-1}^n; \boldsymbol{\theta}\right),$$

where $\boldsymbol{\theta}$ is a set of model parameters.

Language Modeling

We used the dataset made available as a part of the human knowledge compression contest (Hutter, 2012). We refer to this dataset as the *Hutter dataset*. The dataset, which was built from English Wikipedia, contains 100 MBytes of characters which include Latin alphabets, non-Latin alphabets, XML markups and special characters. Closely following the protocols in (Mikolov et al., 2012; Graves, 2013), we used the first 90 MBytes of characters to train a model, the next 5 MBytes as a validation set, and the remaining as a test set, with the vocabulary of 205 characters including a token for an unknown character. We used the average number of bits-per-character (BPC, $E[-\log_2 P(x_{t+1}|\mathbf{h}_t)]$) to measure the performance of each model on the Hutter dataset.

Python Program Evaluation

Zaremba and Sutskever (2014) recently showed that an RNN, more specifically a stacked LSTM, is able to execute a short Python script. Here, we compared the proposed architecture against the conventional stacking approach model on this task, to which refer as *Python program evaluation*.

Scripts used in this task include addition, multiplication, subtraction, for-loop, variable assignment, logical comparison and if-else statement. The goal is to generate, or predict, a correct return value of a given Python script. The input is a program while the output is the result of a print statement: every input script ends with a print statement. Both the input script and the output are sequences of characters, where the input and output vocabularies respectively consist of 41 and 13 symbols.

The advantage of evaluating the models with this task is that we can artificially control the difficulty of each sample (input-output pair). The difficulty is determined by the number of nesting levels in the input sequence and the length of the target sequence. We can do a finer-grained analysis of each model by observing its behavior on examples of different difficulty levels.

In Python program evaluation, we closely follow (Zaremba and Sutskever, 2014) and compute the test accuracy as the next step symbol prediction given a sequence of correct preceding symbols.

4.4.2 Models

We compared three different RNN architectures: a single-layer RNN, a stacked RNN and the proposed GF-RNN. For each architecture, we evaluated three different transition functions: tanh + affine, long short-term memory (LSTM) and gated recurrent unit (GRU). For fair comparison, we constrained the number of parameters of each model to be roughly similar to each other.

For each task, in addition to these capacity-controlled experiments, we conducted a few extra experiments to further test and better understand the properties of the GF-RNN.

Table 4.1: The sizes of the models used in character-level language modeling. Gated Feedback L is a GF-RNN with a same number of hidden units as a stacked RNN (but more parameters). The number of units is shown as `(number of hidden layers)` $\times$ `(number of hidden units per layer)`.

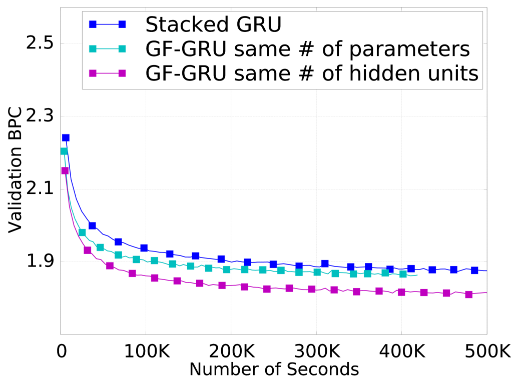
Unit	Architecture	# of Units
tanh	Single	1×1000
	Stacked	3×390
	Gated Feedback	3×303
GRU	Single	1×540
	Stacked	3×228
	Gated Feedback	3×165
	Gated Feedback L	3×228
LSTM	Single	1×456
	Stacked	3×191
	Gated Feedback	3×140
	Gated Feedback L	3×191

Language Modeling

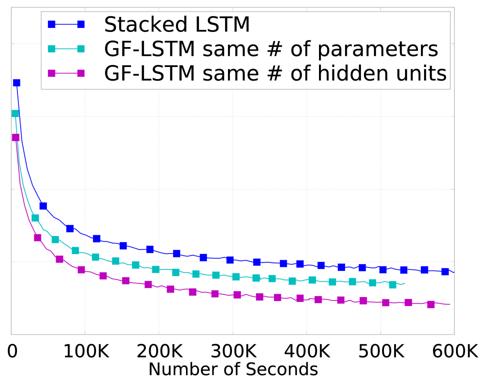
For the task of character-level language modeling, we constrained the number of parameters of each model to correspond to that of a single-layer RNN with 1000 tanh units (see Table 4.1 for more details). Each model is trained for at most 100 epochs.

We used RMSProp (Tieleman and Hinton, 2012) and momentum to tune the model parameters (Graves, 2013). According to the preliminary experiments and their results on the validation set, we used a learning rate of 0.001 and momentum coefficient of 0.9 when training the models having either GRU or LSTM units. It was necessary to choose a much smaller learning rate of 5×10^{-5} in the case of tanh units to ensure the stability of learning. Whenever the norm of the gradient explodes, we halve the learning rate.

Each update is done using a minibatch of 100 subsequences of length 100 each, to avoid memory overflow problems when unfolding in time for backprop. We approximate full back-propagation by carrying the hidden states computed at the previous update to initialize the hidden units in the next update. After every 100-th update, the hidden states were reset to all zeros.



(a) GRU



(b) LSTM

Figure 4.2: Validation learning curves of three different RNN architectures; stacked RNN, GF-RNN with the same number of model parameters and GF-RNN with the same number of hidden units. The curves represent training up to 100 epochs. Best viewed in colors.

Table 4.2: Test set BPC (lower is better) of models trained on the Hutter dataset for a 100 epochs. ($*$) The gated-feedback RNN with the global reset gates fixed to 1 (see Sec. 4.5.1 for details). Bold indicates statistically significant winner over the column (same type of units, different overall architecture).

	tanh	GRU	LSTM
Single-layer	1.937	1.883	1.887
Stacked	**1.892**	1.871	1.868
Gated Feedback	1.949	**1.855**	**1.842**
Gated Feedback L	–	**1.813**	**1.789**
Feedback*	–	–	1.854

Python Program Evaluation

For the task of Python program evaluation, we used an RNN encoder-decoder based approach to learn the mapping from Python scripts to the corresponding outputs as done by Cho et al. (2014); Sutskever et al. (2014) for machine translation. When training the models, Python scripts are fed into the encoder RNN, and the hidden state of the encoder RNN is unfolded for 50 time steps. Prediction is performed by the decoder RNN whose initial hidden state is initialized with the last hidden state of the encoder RNN. The first hidden state of encoder RNN h_0 is always initialized to a zero vector.

For this task, we used GRU and LSTM units either with or without the gated-feedback connections. Each encoder or decoder RNN has three hidden layers.

For GRU, each hidden layer contains 230 units, and for LSTM each hidden layer contains 200 units.

Following Zaremba and Sutskever (2014), we used the *mixed* curriculum strategy for training each model, where each training example has a random difficulty sampled uniformly. We generated $320,000$ examples using the script provided by Zaremba and Sutskever (2014), with the nesting randomly sampled from $[1, 5]$ and the target length from $[1, 10^{10}]$.

We used Adam (Kingma and Ba, 2014) to train our models, and each update was using a minibatch with 128 sequences. We used a learning rate of 0.001 and β_1 and β_2 were both set to 0.99. We trained each model for 30 epochs, with early stopping based on the validation set performance to prevent over-fitting.

At test time, we evaluated each model on multiple sets of test examples where each set is generated using a fixed target length and number of nesting levels. Each test set contains $2,000$ examples which are ensured not to overlap with the training set.

4.5 Results and Analysis

4.5.1 Language Modeling

It is clear from Table 4.2 that the proposed gated-feedback architecture outperforms the other baseline architectures that we have tried when used together with widely used gated units such as LSTM and GRU. However, the proposed architecture failed to improve the performance of a vanilla-RNN with tanh units. In addition to the final modeling performance, in Fig. 4.2, we plotted the learning curves of some models against wall-clock time (measured in seconds). RNNs that are trained with the proposed gated-feedback architecture tends to make much faster progress over time. This behavior is observed both when the number of parameters is constrained and when the number of hidden units is constrained. This suggests that the proposed GF-RNN significantly facilitates optimization/learning.

Effect of Global Reset Gates After observing the superiority of the proposed gated-feedback architecture over the single-layer or conventional stacked ones, we

Table 4.3: Generated texts with our trained models. Given the seed at the left-most column (bold-faced font), the models predict next 200 ∼ 300 characters. Tabs, spaces and new-line characters are also generated by the models.

Seed	Stacked LSTM	GF-LSTM
`[[pl:Icon]]` `[[pt:Icon]]` `[[ru:Icon]]` `[[sv:Programspraket Icon]]</text>` `</revision>` `</page>` `<page>` `<title>Iconology</title>` `<id>14802</id>` `<revi`	`<revision>` `<id>15908383</id>` `<timestamp>` `2002-07-20T18:33:34Z` `</timestamp>` `<contributor>` `<username>The Courseichi</userrand` `vehicles in [[enguit]].` `==The inhibitors and alphabetsy and moral/` `hande in===In four [[communications]] and`	`<revision>` `<id>41968413</id>` `<timestamp>` `2006-09-03T11:38:06Z` `</timestamp>` `<contributor>` `<username>Navisb</username>` `<id>46264</id>` `</contributor>` `<comment>The increase from the time`
`<title>Inherence relation</title>` `<id>14807</id>` `<revision>` `<id>34980694</id>` `<timestamp>` `2006-01-13T04:19:25Z` `</timestamp>` `<contributor>` `<username>Ro`	`<username>Robert]]` `[[su:20 aves]]` `[[vi:10 Februari]]` `[[bi:16 agostoferosín]]` `[[pt:Darenetische]]` `[[eo:Hebrew selsowen]]` `[[hr:2 febber]]` `[[io:21 februari]]` `[[it:18 de februari]]`	`<username>Roma</username>` `<id>48</id>` `</contributor>` `<comment>Vly’’’ and when one hand` `is angels and [[ghost]] borted and` `’’mask r:centrions]], [[Afghanistan]],` `[[Glencoddic tetrahedron]], [[Adjudan]],` `[[Dghacn]], for example, in which materials` `dangerous (carriers) can only use with one`

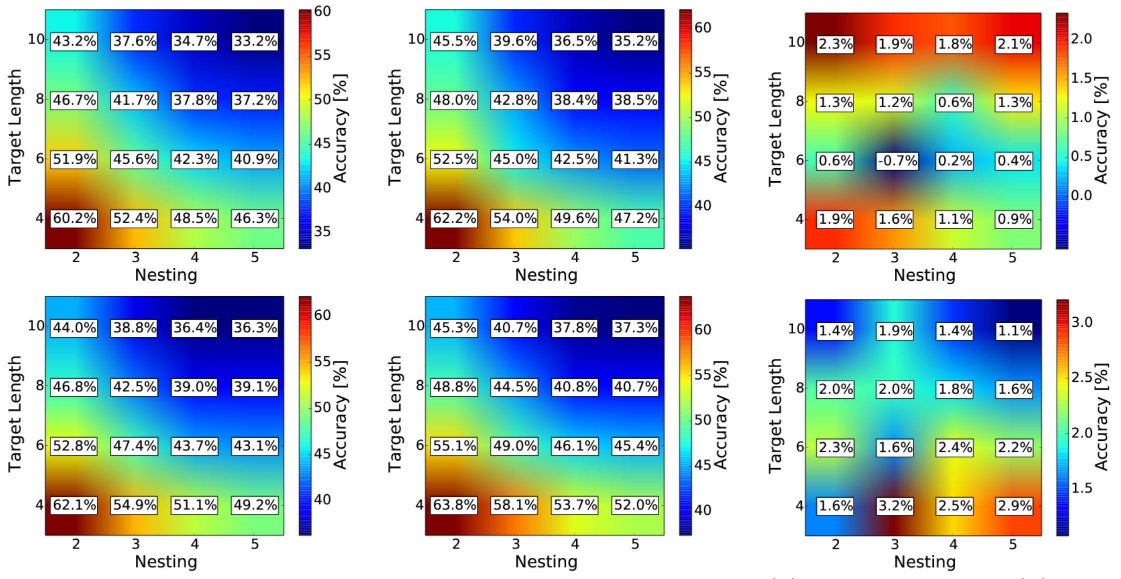
further trained another GF-RNN with LSTM units, but this time, after fixing the global reset gates to 1 to validate the need for the global reset gates. Without the global reset gates, feedback signals from the upper recurrent layers influence the lower recurrent layer fully without any control. The test set BPC of GF-LSTM without global reset gates was 1.854 which is in between the results of conventional stacked LSTM and GF-LSTM with global reset gates (see the last row of Table 4.2) which confirms the importance of adaptively gating the feedback connections.

Qualitative Analysis: Text Generation Here we qualitatively evaluate the stacked LSTM and GF-LSTM trained earlier by generating text. We choose a subsequence of characters from the test set and use it as an initial seed. Once the model finishes reading the seed text, we let the model generate the following characters by sampling a symbol from *softmax* probabilities of a time step and then provide the symbol as next input.

Given two seed snippets selected randomly from the test set, we generated the sequence of characters ten times for each model (stacked LSTM and GF-LSTM). We show one of those ten generated samples per model and per seed snippet in Table 4.3. We observe that the stacked LSTM failed to close the tags with `</username>` and `</contributor>` in both trials. However, the GF-LSTM succeeded to close both of them, which shows that it learned about the structure of XML tags. This type of behavior could be seen throughout all ten random generations.

Table 4.4: Test set BPC of neural language models trained on the Hutter dataset, MRNN = multiplicative RNN results from Sutskever et al. (2011) and Stacked LSTM results from Graves (2013).

MRNN	Stacked LSTM	GF-LSTM
1.60	1.67	**1.58**



(a) Stacked RNN (b) Gated Feedback RNN (c) Gaps between (a) and (b)

Figure 4.3: Heatmaps of (a) stacked RNN, (b) GF-RNN, and (c) difference obtained by substracting (a) from (b). The top row is the heatmaps of models using GRUs, and the bottom row represents the heatmaps of the models using LSTM units. Best viewed in colors.

Large GF-RNN We trained a larger GF-RNN that has five recurrent layers, each of which has 700 LSTM units. This makes it possible for us to compare the performance of the proposed architecture against the previously reported results using other types of RNNs. In Table 4.4, we present the test set BPC by a multiplicative RNN (Sutskever et al., 2011), a stacked LSTM (Graves, 2013) and the GF-RNN with LSTM units. The performance of the proposed GF-RNN is comparable to, or better than, the previously reported best results. Note that Sutskever et al. (2011) used the vocabulary of 86 characters (removed XML tags and the

Wikipedia markups), and their result is not directly comparable with ours. In this experiment, we used Adam (Kingma and Ba, 2014) instead of RMSProp to optimize the RNN. We used learning rate of 0.001 and β_1 and β_2 were set to 0.9 and 0.99, respectively.

4.5.2 Python Program Evaluation

Fig. 4.3 presents the test results of each model represented in heatmaps. The accuracy tends to decrease by the growth of the length of target sequences or the number of nesting levels, where the difficulty or complexity of the Python program increases. We observed that in most of the test sets, GF-RNNs are outperforming stacked RNNs, regardless of the type of units. Fig. 4.3 (c) represents the gaps between the test accuracies of stacked RNNs and GF-RNNs which are computed by subtracting (a) from (b). In Fig. 4.3 (c), the red and yellow colors, indicating large gains, are concentrated on top or right regions (either the number of nesting levels or the length of target sequences increases). From this we can more easily see that the GF-RNN outperforms the stacked RNN, especially as the number of nesting levels grows or the length of target sequences increases.

4.6 Conclusion

We proposed a novel architecture for deep stacked RNNs which uses gated-feedback connections between different layers. Our experiments focused on challenging sequence modeling tasks of character-level language modeling and Python program evaluation. The results were consistent over different datasets, and clearly demonstrated that gated-feedback architecture is helpful when the models are trained on complicated sequences that involve long-term dependencies. We also showed that gated-feedback architecture was faster in wall-clock time over the training and achieved better performance compared to standard stacked RNN with a same amount of capacity. Large GF-LSTM was able to outperform the previously reported best results on character-level language modeling. This suggests that GF-RNNs are also scalable. GF-RNNs were able to outperform standard stacked RNNs and the best previous records on Python program evaluation task with varying difficulties.

We noticed a deterioration in performance when the proposed gated-feedback architecture was used together with a tanh activation function, unlike when it was used with more sophisticated gated activation functions. More thorough investigation into the interaction between the gated-feedback connections and the role of recurrent activation function is required in the future.

5 Prologue to Second Article

5.1 Article Details

A Character-Level Decoder without Explicit Segmentation for Neural Machine Translation. Junyoung Chung, Kyunghyun Cho and Yoshua Bengio, *Proceedings of the 54th Association for Computational Linguistics (ACL 2016).*

Personal Contribution.
The idea of taking a character-level approach in neural machine translation was initially proposed by Yoshua Bengio and Kyunghyun Cho. Kyunghyun Cho and I designed the biscale RNNs. I implemented the algorithm and performed all of the experiments. I wrote the major part of the paper, Kyunghyun Cho did a general editing afterward, and Yoshua Bengio did the final editing.

5.2 Context

The character-level approach has been believed to be a difficult path to perform a machine translation task due to the data sparsity issue. In fact, for neural machine translation systems, a large dictionary size has been the difficulty, and researchers were struggling to solve this problem (Jean et al., 2015; Luong et al., 2015). Before this paper was published, neural machine translation systems using subwords as tokens have shown some promising results (Sennrich et al., 2015). This success has dramatically reduced the size of the dictionary that is used in neural machine translation systems, and partially removed the rare-word problem. The rare-word problem arises due to the long-tailed distribution of words in text corpora that are used for training machine tranlsation systems, where some infrequent words are very sparse compared to other frequent words. However, even for the subword-based translation systems, it was still necessary to build a dictionary from the training corpus before training. Also, the rare-word problem could not be completely resolved since the dictionary contains a limited number of tokens (subwords) that are built from the training corpus.

5.3 Contributions

In this work, we proposed to replace only the decoder to use characters instead of subword units. The motivation of this setting was to provide a transparent comparison between two different types of input units. We did an exhaustive experiments using four shared tasks from the WMT'15 parallel corpora except for En$\leftrightarrow$Fr. We proposed a biscale RNN, which consists of two hidden layers with different update speeds. The first layer is a fast layer, which is expected to model the components that change fast, such as characters. The second layer is a slow layer, which is expected to model the components that change slower such as words or phrases. The new architecture was inspired by the GF-RNN (Chung et al., 2015). Here, unlike to the GF-RNN, a biscale RNN exploits a set of gating units that are tied. One drawback of the biscale RNN is that the gating units are continuous variables in the range of $[0, 1]$. This ends up being like updating the hidden state at every time step, but the amount of change is scaled down by the gating units. The biscale RNN was extended to the hierarchical multiscale RNN (Chung et al., 2016), which exploits discrete gating units.

6 Character-Level Neural Machine Translation

6.1 Introduction

The existing machine translation systems have relied almost exclusively on word-level modelling with explicit segmentation. This is mainly due to the issue of data sparsity which becomes much more severe, especially for n-grams, when a sentence is represented as a sequence of characters rather than words, as the length of the sequence grows significantly. In addition to data sparsity, we often have a priori belief that a word, or its segmented-out lexeme, is a basic unit of meaning, making it natural to approach translation as mapping from a sequence of source-language words to a sequence of target-language words.

This has continued with the more recently proposed paradigm of neural machine translation, although neural networks do not suffer from character-level modelling and rather suffer from the issues specific to word-level modelling, such as the increased computational complexity from a very large target vocabulary (Jean et al., 2015; Luong et al., 2015). Therefore, in this paper, we address a question of whether *neural machine translation can be done directly on a sequence of characters without any explicit word segmentation*.

To answer this question, we focus on representing the target side as a character sequence. We evaluate neural machine translation models with a character-level decoder on four language pairs from WMT'15 to make our evaluation as convincing as possible. We represent the source side as a sequence of subwords extracted using byte-pair encoding from Sennrich et al. (2015), and vary the target side to be either a sequence of subwords or characters. On the target side, we further design a novel recurrent neural network (RNN), called *biscale recurrent network*, that better handles multiple timescales in a sequence, and test it in addition to a naive, stacked recurrent neural network.

On all of the four language pairs–En-Cs, En-De, En-Ru and En-Fi–, the models with a character-level decoder outperformed the ones with a subword-level decoder. We observed a similar trend with the ensemble of each of these configurations, outperforming both the previous best neural and non-neural translation systems on En-Cs, En-De and En-Fi, while achieving a comparable result on En-Ru. We find these results to be a strong evidence that neural machine translation can indeed learn to translate at the character-level and that in fact, it benefits from doing so.

6.2 Neural Machine Translation

Neural machine translation refers to a recently proposed approach to machine translation (Forcada and Ñeco, 1997; Kalchbrenner and Blunsom, 2013; Cho et al., 2014; Sutskever et al., 2014). This approach aims at building an end-to-end neural network that takes as input a source sentence $X = (x_1, \ldots, x_{T_x})$ and outputs its translation $Y = (y_1, \ldots, y_{T_y})$, where x_t and $y_{t'}$ are respectively source and target symbols. This neural network is constructed as a composite of an encoder network and a decoder network.

The encoder network encodes the input sentence X into its continuous representation. In this paper, we closely follow the neural translation model proposed in Bahdanau et al. (2015) and use a bidirectional recurrent neural network, which consists of two recurrent neural networks. The forward network reads the input sentence in a forward direction: $\overrightarrow{\mathbf{z}}_t = \overrightarrow{\phi}(e_x(x_t), \overrightarrow{\mathbf{z}}_{t-1})$, where $e_x(x_t)$ is a continuous embedding of the t-th input symbol, and ϕ is a recurrent activation function. Similarly, the reverse network reads the sentence in a reverse direction (right to left): $\overleftarrow{\mathbf{z}}_t = \overleftarrow{\phi}(e_x(x_t), \overleftarrow{\mathbf{z}}_{t+1})$. At each location in the input sentence, we concatenate the hidden states from the forward and reverse RNNs to form a context set $C = \{\mathbf{z}_1, \ldots, \mathbf{z}_{T_x}\}$, where $\mathbf{z}_t = \left[\overrightarrow{\mathbf{z}}_t; \overleftarrow{\mathbf{z}}_t\right]$.

Then the decoder computes the conditional distribution over all possible translations based on this context set. This is done by first rewriting the conditional probability of a translation: $\log p(Y|X) = \sum_{t'=1}^{T_y} \log p(y_{t'}|y_{<t'}, X)$. For each conditional term in the summation, the decoder RNN updates its hidden state by

$$\mathbf{h}_{t'} = \phi(e_y(y_{t'-1}), \mathbf{h}_{t'-1}, \mathbf{c}_{t'}), \tag{6.1}$$

where e_y is the continuous embedding of a target symbol. $\mathbf{c}_{t'}$ is a context vector computed by a soft-alignment mechanism:

$$\mathbf{c}_{t'} = f_{\text{align}}(e_y(y_{t'-1}), \mathbf{h}_{t'-1}, C)). \tag{6.2}$$

The soft-alignment mechanism f_{align} weights each vector in the context set C according to its relevance given what has been translated. The weight of each vector $\mathbf{z}_t$ is computed by

$$\alpha_{t,t'} = \frac{1}{Z} e^{f_{\text{score}}(e_y(y_{t'-1}), \mathbf{h}_{t'-1}, \mathbf{z}_t)}, \tag{6.3}$$

where f_{score} is a parametric function returning an unnormalized score for $\mathbf{z}_t$ given $\mathbf{h}_{t'-1}$ and $y_{t'-1}$. We use a feedforward network with a single hidden layer in this paper.[1] Z is a normalization constant: $Z = \sum_{k=1}^{T_x} e^{f_{\text{score}}(e_y(y_{t'-1}), \mathbf{h}_{t'-1}, \mathbf{z}_k)}$. This procedure can be understood as computing the alignment probability between the t'-th target symbol and t-th source symbol.

The hidden state $\mathbf{h}_{t'}$, together with the previous target symbol $y_{t'-1}$ and the context vector $\mathbf{c}_{t'}$, is fed into a feedforward neural network to result in the conditional distribution:

$$p(y_{t'} \mid y_{<t'}, X) \propto e^{f_{\text{out}}^{y_{t'}}(e_y(y_{t'-1}), \mathbf{h}_{t'}, \mathbf{c}_{t'})}. \tag{6.4}$$

The whole model, consisting of the encoder, decoder and soft-alignment mechanism, is then tuned end-to-end to minimize the negative log-likelihood using stochastic gradient descent.

6.3 Towards Character-Level Translation

6.3.1 Motivation

Let us revisit how the source and target sentences (X and Y) are represented in neural machine translation. For the source side of any given training corpus, we scan through the whole corpus to build a vocabulary V_x of unique tokens to

1. For other possible implementations, see (Luong et al., 2015).

which we assign integer indices. A source sentence X is then built as a sequence of the indices of such tokens belonging to the sentence, i.e., $X = (x_1, \ldots, x_{T_x})$, where $x_t \in \{1, 2, \ldots, |V_x|\}$. The target sentence is similarly transformed into a target sequence of integer indices.

Each token, or its index, is then transformed into a so-called one-hot vector of dimensionality $|V_x|$. All but one elements of this vector are set to 0. The only element whose index corresponds to the token's index is set to 1. This one-hot vector is the one which any neural machine translation model sees. The embedding function, e_x or e_y, is simply the result of applying a linear transformation (the embedding matrix) to this one-hot vector.

The important property of this approach based on one-hot vectors is that the neural network is oblivious to the underlying semantics of the tokens. To the neural network, each and every token in the vocabulary is equal distance away from every other token. The semantics of those tokens are simply *learned* (into the embeddings) to maximize the translation quality, or the log-likelihood of the model.

This property allows us great freedom in the choice of tokens' unit. Neural networks have been shown to work well with word tokens (Bengio et al., 2001; Schwenk, 2007; Mikolov et al., 2010) but also with finer units, such as subwords (Sennrich et al., 2015; Botha and Blunsom, 2014; Luong et al., 2013) as well as symbols resulting from compression/encoding (Chitnis and DeNero, 2015). Although there have been a number of previous research reporting the use of neural networks with characters (see, e.g., Mikolov et al. (2012) and Santos and Zadrozny (2014)), the dominant approach has been to preprocess the text into a sequence of symbols, each associated with a sequence of characters, after which the neural network is presented with those symbols rather than with characters.

More recently in the context of neural machine translation, two research groups have proposed to directly use characters. Kim et al. (2015) proposed to represent each word not as a single integer index as before, but as a sequence of characters, and use a convolutional network followed by a highway network (Srivastava et al., 2015) to extract a continuous representation of the word. This approach, which effectively replaces the embedding function e_x, was adopted by Costa-Jussà and Fonollosa (2016) for neural machine translation. Similarly, Ling et al. (2015) use a bidirectional recurrent neural network to replace the embedding functions e_x

and e_y to respectively encode a character sequence to and from the corresponding continuous word representation. A similar, but slightly different approach was proposed by Lee et al. (2015), where they explicitly mark each character with its relative location in a word (e.g., "B"eginning and "I"ntermediate).

Despite the fact that these recent approaches work at the level of characters, it is less satisfying that they all rely on knowing how to segment characters into words. Although it is generally easy for languages like English, this is not always the case. This word segmentation procedure can be as simple as tokenization followed by some punctuation normalization, but also can be as complicated as morpheme segmentation requiring a separate model to be trained in advance (Creutz and Lagus, 2005; Huang and Zhao, 2007). Furthermore, these segmentation[2] steps are often tuned or designed separately from the ultimate objective of translation quality, potentially contributing to a suboptimal quality.

Based on this observation and analysis, in this paper, we ask ourselves and the readers a question which should have been asked much earlier: *Is it possible to do character-level translation without any explicit segmentation?*

6.3.2 Why Word-Level Translation?

(1) Word as a Basic Unit of Meaning A word can be understood in two different senses. In the abstract sense, a word is a basic unit of meaning (lexeme), and in the other sense, can be understood as a "concrete word as used in a sentence." (Booij, 2012). A word in the former sense turns into that in the latter sense via a process of morphology, including inflection, compounding and derivation. These three processes do alter the meaning of the lexeme, but often it stays close to the original meaning. Because of this view of words as basic units of meaning (either in the form of lexemes or derived form) from linguistics, much of previous work in natural language processing has focused on using words as basic units of which a sentence is encoded as a sequence. Also, the potential difficulty in finding a mapping between a word's character sequence and meaning[3] has likely contributed to this trend toward word-level modelling.

2. From here on, the term *segmentation* broadly refers to any method that splits a given character sequence into a sequence of subword symbols.

3. For instance, "quit", "quite" and "quiet" are one edit-distance away from each other but have distinct meanings.

(2) Data Sparsity There is a further technical reason why much of previous research on machine translation has considered words as a basic unit. This is mainly due to the fact that major components in the existing translation systems, such as language models and phrase tables, are a count-based estimator of probabilities. In other words, a probability of a subsequence of symbols, or pairs of symbols, is estimated by counting the number of its occurrences in a training corpus. This approach severely suffers from the issue of data sparsity, which is due to a large state space which grows exponentially w.r.t. the length of subsequences while growing only linearly w.r.t. the corpus size. This poses a great challenge to character-level modelling, as any subsequence will be on average 4–5 times longer when characters, instead of words, are used. Indeed, Vilar et al. (2007) reported worse performance when the character sequence was directly used by a phrase-based machine translation system. More recently, Neubig et al. (2013) proposed a method to improve character-level translation with phrase-based translation systems, however, with only a limited success.

(3) Vanishing Gradient Specifically to neural machine translation, a major reason behind the wide adoption of word-level modelling is due to the difficulty in modelling long-term dependencies with recurrent neural networks (Bengio et al., 1994; Hochreiter, 1998). As the lengths of the sentences on both sides grow when they are represented in characters, it is easy to believe that there will be more long-term dependencies that must be captured by the recurrent neural network for successful translation.

6.3.3 Why Character-Level Translation?

Why *not* Word-Level Translation? The most pressing issue with word-level processing is that we do not have a perfect word segmentation algorithm for any one language. A perfect segmentation algorithm needs to be able to segment any given sentence into a sequence of lexemes and morphemes. This problem is however a difficult problem on its own and often requires decades of research (see, e.g., Creutz and Lagus (2005) for Finnish and other morphologically rich languages and Huang and Zhao (2007) for Chinese). Therefore, many opt to using either a rule-based tokenization approach or a suboptimal, but still available, learning based segmentation algorithm.

The outcome of this naive, sub-optimal segmentation is that the vocabulary is often filled with many similar words that share a lexeme but have different morphology. For instance, if we apply a simple tokenization script to an English corpus, "run", "runs", "ran" and "running" are all separate entries in the vocabulary, while they clearly share the same lexeme "run". This prevents any machine translation system, in particular neural machine translation, from modelling these morphological variants efficiently.

More specifically in the case of neural machine translation, each of these morphological variants–"run", "runs", "ran" and "running"– will be assigned a d-dimensional word vector, leading to four independent vectors, while it is clear that if we can segment those variants into a lexeme and other morphemes, we can model them more efficiently. For instance, we can have a d-dimensional vector for the lexeme "run" and much smaller vectors for "s" and"ing". Each of those variants will be then a composite of the lexeme vector (shared across these variants) and morpheme vectors (shared across words sharing the same suffix, for example) (Botha and Blunsom, 2014). This makes use of distributed representation, which generally yields better generalization, but seems to require an optimal segmentation, which is unfortunately almost never available.

In addition to inefficiency in modelling, there are two additional negative consequences from using (unsegmented) words. First, the translation system cannot generalize well to novel words, which are often mapped to a token reserved for an unknown word. This effectively ignores any meaning or structure of the word to be incorporated when translating. Second, even when a lexeme is common and frequently observed in the training corpus, its morphological variant may not be. This implies that the model sees this specific, rare morphological variant much less and will not be able to translate it well. However, if this rare morphological variant shares a large part of its spelling with other more common words, it is desirable for a machine translation system to exploit those common words when translating those rare variants.

Why Character-Level Translation? All of these issues can be addressed to certain extent by directly modelling characters. Although the issue of data sparsity arises in character-level translation, it is elegantly addressed by using a parametric approach based on recurrent neural networks instead of a non-parametric count-based approach. Furthermore, in recent years, we have learned how to build and train a recurrent neural network that can well capture long-term dependencies by using more sophisticated activation functions, such as long short-term memory (LSTM) units (Hochreiter and Schmidhuber, 1997) and gated recurrent units (Cho et al., 2014).

Kim et al. (2015) and Ling et al. (2015) recently showed that by having a neural network that converts a character sequence into a word vector, we avoid the issues from having many morphological variants appearing as separate entities in a vocabulary. This is made possible by sharing the character-to-word neural network across all the unique tokens. A similar approach was applied to machine translation by Ling et al. (2015).

These recent approaches, however, still rely on the availability of a good, if not optimal, segmentation algorithm. Ling et al. (2015) indeed states that "[m]uch of the prior information regarding morphology, cognates and rare word translation among others, should be incorporated".

It however becomes unnecessary to consider these prior information, if we use a neural network, be it recurrent, convolution or their combination, directly on the unsegmented character sequence. The possibility of using a sequence of unsegmented characters has been studied over many years in the field of deep learning. For instance, Mikolov et al. (2012) and Sutskever et al. (2011) trained a recurrent neural network language model (RNN-LM) on character sequences. The latter showed that it is possible to generate sensible text sequences by simply sampling a character at a time from this model. More recently, Zhang et al. (2015) and Xiao and Cho (2016) successfully applied a convolutional net and a convolutional-recurrent net respectively to character-level document classification without any explicit segmentation. Gillick et al. (2015) further showed that it is possible to train a recurrent neural network on unicode bytes, instead of characters or words, to perform part-of-speech tagging and named entity recognition.

These previous works suggest the possibility of applying neural networks for the task of machine translation, which is often considered a substantially more difficult problem compared to document classification and language modelling.

6.3.4 Challenges and Questions

There are two overlapping sets of challenges for the source and target sides. On the source side, it is unclear how to build a neural network that learns a highly nonlinear mapping from a spelling to the meaning of a sentence.

On the target side, there are two challenges. The first challenge is the same one from the source side, as the decoder neural network needs to summarize what has been translated. In addition to this, the character-level modelling on the target side is more challenging, as the decoder network must be able to generate a long, coherent sequence of characters. This is a great challenge, as the size of the state space grows exponentially w.r.t. the number of symbols, and in the case of characters, it is often 300-1000 symbols long.

All these challenges should first be framed as questions; whether the current recurrent neural networks, which are already widely used in neural machine translation, are able to address these challenges as they are. In this paper, we aim at answering these *questions* empirically and focus on the challenges on the target side (as the target side shows both of the challenges).

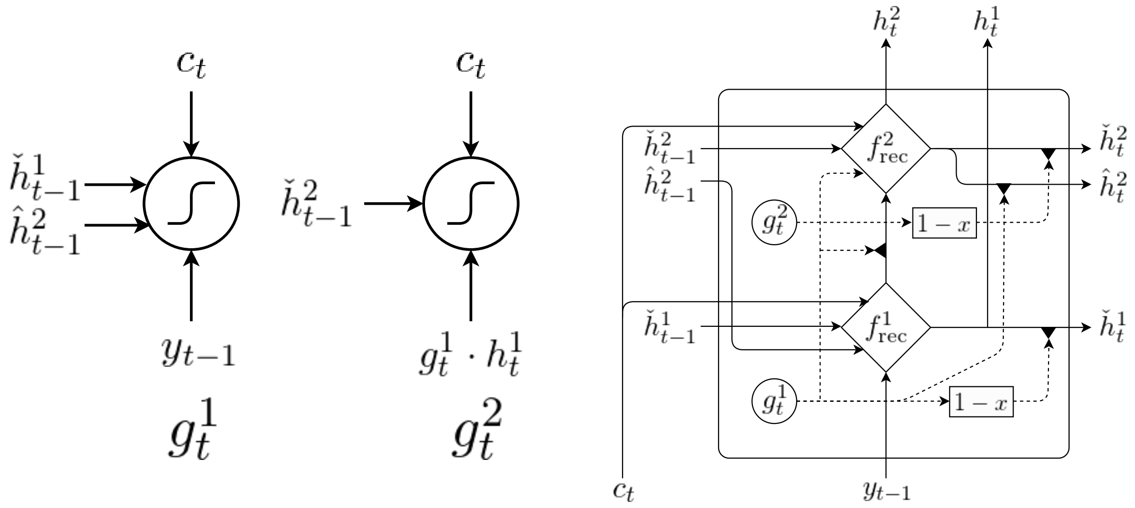
6.4 Character-Level Translation

In this paper, we try to answer the questions posed earlier by testing two different types of recurrent neural networks on the target side (decoder).

First, we test an existing recurrent neural network with gated recurrent units (GRUs). We call this decoder a *base* decoder.

Second, we build a novel two-layer recurrent neural network, inspired by the gated-feedback network from Chung et al. (2015), called a *biscale* recurrent neural network. We design this network to facilitate capturing two timescales, motivated by the fact that characters and words may work at two separate timescales.

We choose to test these two alternatives for the following purposes. Experiments with the base decoder will clearly answer whether the existing neural network is enough to handle character-level decoding, which has not been properly answered in the context of machine translation. The alternative, the biscale decoder, is tested



(a) Gating units (b) One-step processing

Figure 6.1: Biscale recurrent neural network

in order to see whether it is possible to design a better decoder, if the answer to the first question is positive.

6.4.1 Biscale Recurrent Neural Network

In this proposed biscale recurrent neural network, there are two sets of hidden units, $\mathbf{h}^1$ and $\mathbf{h}^2$. They contain the same number of units, i.e., $\dim(\mathbf{h}^1) = \dim(\mathbf{h}^2)$. The first set $\mathbf{h}^1$ models a fast-changing timescale (thereby, a *faster layer*), and $\mathbf{h}^2$ a slower timescale (thereby, a *slower layer*). For each hidden unit, there is an associated gating unit, to which we refer by $\mathbf{g}^1$ and $\mathbf{g}^2$. For the description below, we use $y_{t'-1}$ and $\mathbf{c}_{t'}$ for the previous target symbol and the context vector (see Eq. (6.2)), respectively.

Let us start with the faster layer. The faster layer outputs two sets of activations, a normal output $\mathbf{h}^1_{t'}$ and its gated version $\check{\mathbf{h}}^1_{t'}$. The activation of the faster layer is computed by

$$\mathbf{h}^1_{t'} = \tanh\left(\mathbf{W}^{h^1}\left[e_y(y_{t'-1}); \check{\mathbf{h}}^1_{t'-1}; \hat{\mathbf{h}}^2_{t'-1}; \mathbf{c}_{t'}\right]\right),$$

where $\check{\mathbf{h}}^1_{t'-1}$ and $\hat{\mathbf{h}}^2_{t'-1}$ are the gated activations of the faster and slower layers respectively. These gated activations are computed by

$$\check{\mathbf{h}}^1_{t'} = (1 - \mathbf{g}^1_{t'}) \odot \mathbf{h}^1_{t'}, \quad \hat{\mathbf{h}}^2_{t'} = \mathbf{g}^1_{t'} \odot \mathbf{h}^2_{t'}.$$

In other words, the faster layer's activation is based on the adaptive combination of the faster and slower layers' activations from the previous time step. Whenever the faster layer determines that it needs to reset, i.e., $\mathbf{g}^1_{t'-1} \approx 1$, the next activation will be determined based more on the slower layer's activation.

The faster layer's gating unit is computed by

$$\mathbf{g}^1_{t'} = \sigma\left(\mathbf{W}^{g^1}\left[e_y(y_{t'-1}); \check{\mathbf{h}}^1_{t'-1}; \hat{\mathbf{h}}^2_{t'-1}; \mathbf{c}_{t'}\right]\right),$$

where σ is a sigmoid function.

The slower layer also outputs two sets of activations, a normal output $\mathbf{h}^2_{t'}$ and its gated version $\check{\mathbf{h}}^2_{t'}$. These activations are computed as follows:

$$\begin{aligned}\mathbf{h}^2_{t'} &= (1 - \mathbf{g}^1_{t'}) \odot \mathbf{h}^2_{t'-1} + \mathbf{g}^1_{t'} \odot \tilde{\mathbf{h}}^2_{t'}, \\ \check{\mathbf{h}}^2_{t'} &= (1 - \mathbf{g}^2_{t'}) \odot \mathbf{h}^2_{t'},\end{aligned}$$

where $\tilde{\mathbf{h}}^2_{t'}$ is a candidate activation. The slower layer's gating unit $\mathbf{g}^2_{t'}$ is computed by

$$\mathbf{g}^2_{t'} = \sigma\left(\mathbf{W}^{g^2}\left[(\mathbf{g}^1_{t'} \odot \mathbf{h}^1_{t'}); \check{\mathbf{h}}^2_{t'-1}; \mathbf{c}_{t'}\right]\right).$$

This adaptive leaky integration based on the gating unit from the faster layer has a consequence that the slower layer updates its activation only when the faster layer resets. This puts a soft constraint that the faster layer runs at a faster rate by preventing the slower layer from updating while the faster layer is processing a current chunk.

The candidate activation is then computed by

$$\tilde{\mathbf{h}}^2_{t'} = \tanh\left(\mathbf{W}^{h^2}\left[(\mathbf{g}^1_{t'} \odot \mathbf{h}^1_{t'}); \check{\mathbf{h}}^2_{t'-1}; \mathbf{c}_{t'}\right]\right). \tag{6.5}$$

$\check{\mathbf{h}}^2_{t'-1}$ indicates the *reset* activation from the previous time step, similarly to what happened in the faster layer, and $\mathbf{c}_{t'}$ is the input from the context.

According to $\mathbf{g}^1_{t'} \odot \mathbf{h}^1_{t'}$ in Eq. (6.5), the faster layer influences the slower layer,

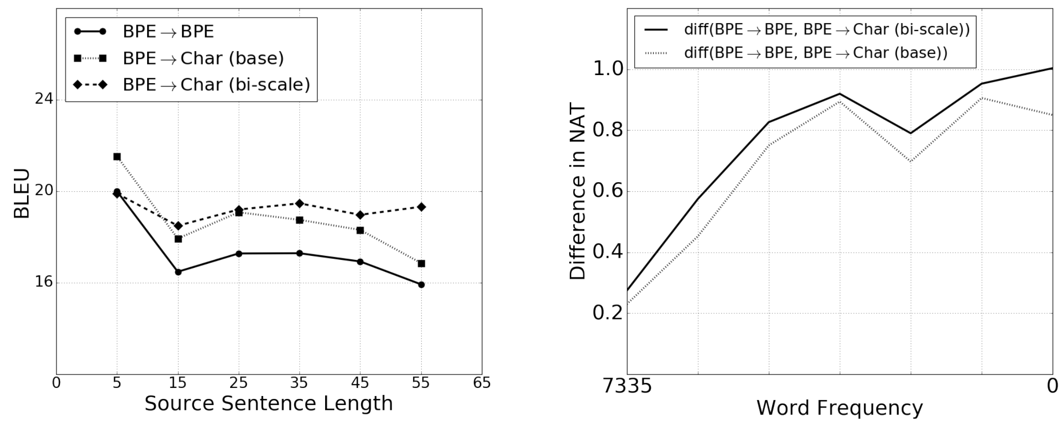


Figure 6.2: (left) The BLEU scores on En-Cs w.r.t. the length of source sentences. (right) The difference of word negative log-probabilities between the subword-level decoder and either of the character-level base or biscale decoder.

only when the faster layer has finished processing the current chunk and is about to *reset* itself ($\mathbf{g}^1_{t'} \approx 1$). In other words, the slower layer does not receive any input from the faster layer, until the faster layer has quickly processed the current chunk, thereby running at a slower rate than the faster layer does.

At each time step, the final output of the proposed biscale recurrent neural network is the concatenation of the output vectors of the faster and slower layers, i.e., $[\mathbf{h}^1; \mathbf{h}^2]$. This concatenated vector is used to compute the probability distribution over all the symbols in the vocabulary, as in Eq. (6.4). See Fig. 6.1 for graphical illustration.

		Src	Trgt	Depth	Attention		Model	Development		Test$_1$		Test$_2$	
					$\mathbf{h}^1$	$\mathbf{h}^2$		Single	Ens	Single	Ens	Single	Ens
En-De	(a)	BPE	BPE	1	✓		Base	20.78	–	19.98	–	21.72	–
	(b)			2	✓	✓		$21.26^{21.45}_{20.62}$	23.49	$20.47^{20.88}_{19.30}$	23.10	$22.02^{22.21}_{21.35}$	24.83
	(c)		Char	2		✓	Base	$21.57^{21.88}_{20.88}$	23.14	$\mathbf{21.33}^{21.56}_{19.82}$	$\underline{23.11}$	$\mathbf{23.45}^{23.91}_{21.72}$	25.24
	(d)			2	✓	✓		20.31	–	19.70	–	21.30	–
	(e)			2		✓	Bi-S	$21.29^{21.43}_{21.13}$	23.05	$21.25^{21.47}_{20.62}$	23.04	$23.06^{23.47}_{22.85}$	$\underline{25.44}$
	(f)			2	✓	✓		20.78	–	20.19	–	22.26	–
	(g)			2	✓			20.08	–	19.39	–	20.94	–
	State-of-the-art Non-Neural Approach*							–		$20.60^{(1)}$		$24.00^{(2)}$	
En-Cs	(h)	BPE	BPE	2	✓	✓	Base	$16.12^{16.96}_{15.96}$	19.21	$17.16^{17.68}_{16.38}$	20.79	$14.63^{15.09}_{14.26}$	17.61
	(i)		Char	2		✓	Base	$17.68^{17.78}_{17.39}$	19.52	$19.25^{19.55}_{18.89}$	21.95	$\mathbf{16.98}^{17.17}_{16.81}$	18.92
	(j)			2		✓	Bi-S	$17.62^{17.93}_{17.43}$	19.83	$\mathbf{19.27}^{19.53}_{19.15}$	$\underline{22.15}$	$16.86^{17.10}_{16.68}$	$\underline{18.93}$
	State-of-the-art Non-Neural Approach*							–		$21.00^{(3)}$		$18.20^{(4)}$	
En-Ru	(k)	BPE	BPE	2	✓	✓	Base	$18.56^{18.70}_{18.26}$	21.17	$25.30^{25.40}_{24.95}$	29.26	$19.72^{20.29}_{19.02}$	22.96
	(l)		Char	2		✓	Base	$18.56^{18.87}_{18.39}$	20.53	$\mathbf{26.00}^{26.07}_{25.04}$	$\underline{29.37}$	$\mathbf{21.10}^{21.24}_{20.14}$	23.51
	(m)			2		✓	Bi-S	$18.30^{18.54}_{17.88}$	20.53	$25.59^{25.76}_{24.57}$	29.26	$20.73^{21.02}_{19.97}$	$\underline{23.75}$
	State-of-the-art Non-Neural Approach*							–		$28.70^{(5)}$		$24.30^{(6)}$	
En-Fi	(n)	BPE	BPE	2	✓	✓	Base	$9.61^{10.02}_{9.24}$	11.92	–	–	$8.97^{9.17}_{8.88}$	11.73
	(o)		Char	2		✓	Base	$11.19^{11.55}_{11.09}$	13.72	–	–	$\mathbf{10.93}^{11.56}_{10.11}$	$\underline{13.48}$
	(p)			2		✓	Bi-S	$10.73^{11.04}_{10.40}$	13.39	–	–	$10.24^{10.63}_{9.71}$	13.32
	State-of-the-art Non-Neural Approach*							–		–		$12.70^{(7)}$	

Table 6.1: BLEU scores of the subword-level, character-level base and character-level biscale decoders for both single models and ensembles. The best scores among the single models per language pair are bold-faced, and those among the ensembles are underlined. When available, we report the median value, and the minimum and maximum values as a subscript and a superscript, respectively. (∗) http://matrix.statmt.org/ as of 11 March 2016 (constrained only). (1) Freitag et al. (2014). (2, 6) Williams et al. (2015). (3, 5) Durrani et al. (2014). (4) Haddow et al. (2015). (7) Rubino et al. (2015).

6.5 Experiment Settings

For evaluation, we represent a source sentence as a sequence of subword symbols extracted by byte-pair encoding (BPE, Sennrich et al. (2015)) and a target sentence either as a sequence of BPE-based symbols or as a sequence of characters.

Corpora and Preprocessing We use all available parallel corpora for four language pairs from WMT'15: En-Cs, En-De, En-Ru and En-Fi. They consist of 12.1M, 4.5M, 2.3M and 2M sentence pairs, respectively. We tokenize each corpus using a tokenization script included in Moses.[4] We only use the sentence pairs, when the source side is up to 50 subword symbols long and the target side is either up to 100 subword symbols or 500 characters. We do not use any monolingual corpus.

For all the pairs other than En-Fi, we use newstest-2013 as a development set, and newstest-2014 (Test$_1$) and newstest-2015 (Test$_2$) as test sets. For En-Fi, we use newsdev-2015 and newstest-2015 as development and test sets, respectively.

Models and Training We test three models settings: (1) BPE→BPE, (2) BPE→Char (base) and (3) BPE→Char (biscale). The latter two differ by the type of recurrent neural network we use. We use GRUs for the encoder in all the settings. We used GRUs for the decoders in the first two settings, (1) and (2), while the proposed biscale recurrent network was used in the last setting, (3). The encoder has 512 hidden units for each direction (forward and reverse), and the decoder has 1024 hidden units per layer.

We train each model using stochastic gradient descent with Adam (Kingma and Ba, 2014). Each update is computed using a minibatch of 128 sentence pairs. The norm of the gradient is clipped with a threshold 1 (Pascanu et al., 2013).

Decoding and Evaluation We use beamsearch to approximately find the most likely translation given a source sentence. The beam widths are 5 and 15 respectively for the subword-level and character-level decoders. They were chosen based on the translation quality on the development set. The translations are evaluated

4. Although tokenization is not necessary for character-level modelling, we tokenize the all target side corpora to make comparison against word-level modelling easier.

using BLEU.[5]

Multilayer Decoder and Soft-Alignment Mechanism When the decoder is a multilayer recurrent neural network (including a stacked network as well as the proposed biscale network), the decoder outputs multiple hidden vectors–$\left\{\mathbf{h}^1, \ldots, \mathbf{h}^L\right\}$ for L layers, at a time. This allows an extra degree of freedom in the soft-alignment mechanism (f_{score} in Eq. (6.3)). We evaluate using alternatives, including (1) using only $\mathbf{h}^L$ (slower layer) and (2) using all of them (concatenated).

Ensembles We also evaluate an ensemble of neural machine translation models and compare its performance against the state-of-the-art phrase-based translation systems on all four language pairs. We decode from an ensemble by taking the average of the output probabilities at each step.

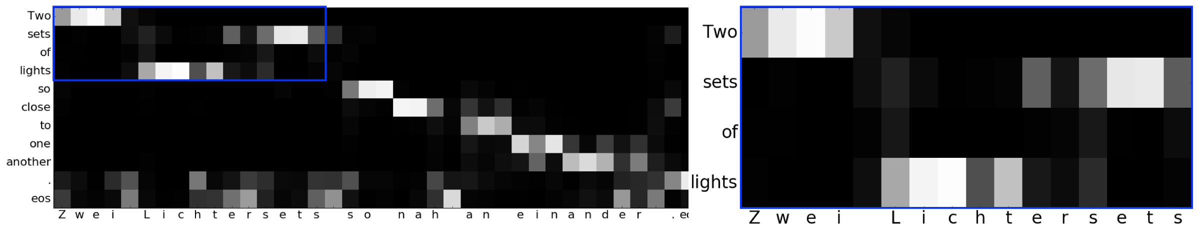


Figure 6.3: Alignment matrix of a test example from En-De using the BPE→Char (biscale) model.

6.6 Quantitative Analysis

Slower Layer for Alignment On En-De, we test which layer of the decoder should be used for computing soft-alignments. In the case of subword-level decoder, we observed no difference between choosing any of the two layers of the decoder against using the concatenation of all the layers (Table 6.1 (a–b)) On the other hand, with the character-level decoder, we noticed an improvement when only the slower layer ($\mathbf{h}^2$) was used for the soft-alignment mechanism (Table 6.1 (c–g)). This suggests that the soft-alignment mechanism benefits by aligning a larger chunk in

5. We used the multi-bleu.perl script from Moses.

the target with a subword unit in the source, and we use only the slower layer for all the other language pairs.

Single Models In Table 6.1, we present a comprehensive report of the translation qualities of (1) subword-level decoder, (2) character-level base decoder and (3) character-level biscale decoder, for all the language pairs. We see that the both types of character-level decoder outperform the subword-level decoder for En-Cs and En-Fi quite significantly. On En-De, the character-level base decoder outperforms both the subword-level decoder and the character-level biscale decoder, validating the effectiveness of the character-level modelling. On En-Ru, among the single models, the character-level decoders outperform the subword-level decoder, but in general, we observe that all the three alternatives work comparable to each other.

These results clearly suggest that *it is indeed possible to do character-level translation without explicit segmentation.* In fact, what we observed is that character-level translation often surpasses the translation quality of word-level translation. Of course, we note once again that our experiment is restricted to using an unsegmented character sequence at the decoder only, and a further exploration toward replacing the source sentence with an unsegmented character sequence is needed.

Ensembles Each ensemble was built using eight independent models. The first observation we make is that in all the language pairs, neural machine translation performs comparably to, or often better than, the state-of-the-art non-neural translation system. Furthermore, the character-level decoders outperform the subword-level decoder in all the cases.

6.7 Qualitative Analysis

(1) Can the character-level decoder generate a long, coherent sentence? The translation in characters is dramatically longer than that in words, likely making it more difficult for a recurrent neural network to generate a coherent sentence in characters. This belief turned out to be false. As shown in Fig. 6.2 (left), there is no significant difference between the subword-level and character-level decoders, even though the lengths of the generated translations are generally 5–10 times longer in characters.

(2) Does the character-level decoder help with rare words? One advantage of character-level modelling is that it can model the composition of any character sequence, thereby better modelling rare morphological variants. We empirically confirm this by observing the growing gap in the average negative log-probability of words between the subword-level and character-level decoders as the frequency of the words decreases. This is shown in Fig. 6.2 (right) and explains one potential cause behind the success of character-level decoding in our experiments (we define $\text{diff}(x, y) = x - y$).

(3) Can the character-level decoder soft-align between a source word and a target character? In Fig. 6.3 (left), we show an example soft-alignment of a source sentence, "Two sets of light so close to one another". It is clear that the character-level translation model well captured the alignment between the source subwords and target characters. We observe that the character-level decoder correctly aligns to "lights" and "sets of" when generating a German compound word "Lichtersets" (see Fig. 6.3 (right) for the zoomed-in version). This type of behaviour happens similarly between "one another" and "einander". Of course, this does not mean that there exists an alignment between a source word and a target character. Rather, this suggests that the internal state of the character-level decoder, the base or biscale, well captures the meaningful chunk of characters, allowing the model to map it to a larger chunk (subword) in the source.

(4) How fast is the decoding speed of the character-level decoder? We evaluate the decoding speed of subword-level base, character-level base and character-level biscale decoders on newstest-2013 corpus (En-De) with a single Titan X GPU. The subword-level base decoder generates 31.9 words per second, and the character-level base decoder and character-level biscale decoder generate 27.5 words per second and 25.6 words per second, respectively. Note that this is evaluated in an online setting, performing consecutive translation, where only one sentence is translated at a time. Translating in a batch setting could differ from these results.

6.8 Conclusion

In this paper, we addressed a fundamental question on whether a recently proposed neural machine translation system can directly handle translation at the level of characters without any word segmentation. We focused on the target side, in which a decoder was asked to generate one character at a time, while soft-aligning between a target character and a source subword. Our extensive experiments, on four language pairs–En-Cs, En-De, En-Ru and En-Fi– strongly suggest that it is indeed possible for neural machine translation to translate at the level of characters, and that it actually benefits from doing so.

Our result has one limitation that we used subword symbols in the source side. However, this has allowed us a more fine-grained analysis, but in the future, a setting where the source side is also represented as a character sequence must be investigated.

7 Prologue to Third Article

7.1 Article Details

Hierarchical Multiscale Recurrent Neural Networks. Junyoung Chung, Sungjin Ahn and Yoshua Bengio, *Proceedings of the 5th International Conference on Learning Representations (ICLR 2017).*

Personal Contribution.
Introducing discrete gating units to implement multiscale RNNs was my own idea. Yoshua Bengio suggested using the straight-through estimator to compute the gradients for the discrete variables. I implemented the algorithm and performed all of the experiments. Sungjin Ahn participated in the development of the update rule and contributed to the writing. I wrote the major part of the paper with Sungjin Ahn, and Yoshua Bengio did a general editing.

7.2 Context

The main difference between this work and the previous work on multiscale RNNs (El Hihi and Bengio, 1995; Koutník et al., 2014) is that the timescales are learned from the data instead of being treated as hyper parameters. This poses a significant challenge since computing the gradients of a discontinuous function is intractable. In this work, we used a trick called straight-through estimator (Hinton, 2012; Bengio et al., 2013; Courbariaux et al., 2016) to compute the gradients. The straight-through estimator is a biased estimator, and in order to reduce its bias, we introduced a technique called slope-annealing.

7.3 Contributions

Learning a decomposable and hierarchical feature representation of sequences has been one of the long-standing challenges of RNNs. It is yet unclear how to explicitly ask each hidden layer to model different level of abstract terms in RNNs. There are huge potential benefits of being able to decompose the learned representation of sequences at multiple timescales. For example, one can imagine a controller that makes decisions in different levels depending on the context or progress of the task. Now, this controller can be used in a program execution task, where programs form a hierarchy. In most of the time, sub-programs are executed, but once in a while a main program has to be executed to collect the outputs of the sub-programs.

In this work, we showed that the proposed RNN architecture can capture the underlying multiscale structures of sequences. The proposed update mechanism using a set of boundary detecting units whose states are binary, allows using three types of state-transition operations. The first operation is COPY, where the dynamic state of an RNN is copied from the previous state without any loss of information. The second operation is UPDATE, which is identical to the state-transition equation of an RNN, be it a simple RNN, a GRU or an LSTM. The third operation is FLUSH, where the current state is propagated to the upper-level layer, and the state is reset into a zero vector.

The proposed hierarchical multiscale RNN (HM-RNN) otbtained state-of-the-art results on the Hutter dataset and Text8 dataset, and a comparable result to the state-of-the-art result on the Penn Treebank dataset. The HM-RNN outperformed a standard deep RNN on a handwriting generation task using IAM-OnDB dataset (Graves et al., 2008).

7.4 Future Directions

Segments of a sequence can be generated by a segmentation algorithm by considering the nearby tokens as the inputs, if not the whole sequence, and the generation process does not necessarily need to be unidirectional. In some tasks, the system needs to operate as an online system, for example, in speech recognition. In a few cases, the system has to predict the targets without knowing the future information. The HM-RNN learns to perform the prediction of the boundaries using the data only given up to the present step. However, in certain circumstances, it would be beneficial to incorporate the very least of the future information. In speech recognition, the prediction of phonemes can be delayed by a few time steps to use more tokens as the inputs. The same approach can be applied to HM-RNN for predicting the boundaries.

8 Hierarchical Multiscale Recurrent Neural Networks

8.1 Introduction

One of the key principles of learning in deep neural networks as well as in the human brain is to obtain a hierarchical representation with increasing levels of abstraction (Bengio, 2009; LeCun et al., 2015; Schmidhuber, 2015). A stack of representation layers, learned from the data in a way to optimize the target task, make deep neural networks entertain advantages such as generalization to unseen examples (Hoffman et al., 2013), sharing learned knowledge among multiple tasks, and discovering disentangling factors of variation (Kingma and Welling, 2013). The remarkable recent successes of the deep convolutional neural networks are particularly based on this ability to learn hierarchical representation for spatial data (Krizhevsky et al., 2012). For modelling temporal data, the recent resurgence of recurrent neural networks (RNN) has led to remarkable advances (Mikolov et al., 2010; Graves, 2013; Cho et al., 2014; Sutskever et al., 2014; Vinyals et al., 2015). However, unlike the spatial data, learning both hierarchical and temporal representation has been among the long-standing challenges of RNNs in spite of the fact that hierarchical multiscale structures naturally exist in many temporal data (Schmidhuber, 1991; Mozer, 1993; El Hihi and Bengio, 1995; Lin et al., 1996; Koutník et al., 2014).

A promising approach to model such hierarchical and temporal representation is the multiscale RNNs (Schmidhuber, 1992; El Hihi and Bengio, 1995; Koutník et al., 2014). Based on the observation that high-level abstraction changes slowly with temporal coherency while low-level abstraction has quickly changing features sensitive to the precise local timing (El Hihi and Bengio, 1995), the multiscale RNNs group hidden units into multiple modules of different timescales. In addition to the fact that the architecture fits naturally to the latent hierarchical structures in many temporal data, the multiscale approach provides the following advantages that resolve some inherent problems of standard RNNs: (a) computational efficiency obtained by updating the high-level layers less frequently, (b) efficiently delivering long-term dependencies with fewer updates at the high-level layers, which mitigates the vanishing gradient problem, (c) flexible resource allocation (e.g., more hidden units to the higher layers that focus on modelling long-term dependencies and less hidden units to the lower layers which are in charge of learning short-term dependencies). In addition, the learned latent hierarchical structures can provide useful information to other downstream tasks such as module structures in computer program learning, sub-task structures in hierarchical reinforcement learning, and story segments in video understanding.

There have been various approaches to implementing the multiscale RNNs. The most popular approach is to set the timescales as hyperparameters (El Hihi and Bengio, 1995; Koutník et al., 2014; Bahdanau et al., 2016) instead of treating them as dynamic variables that can be learned from the data (Schmidhuber, 1991, 1992; Chung et al., 2015, 2016). However, considering the fact that non-stationarity is prevalent in temporal data, and that many entities of abstraction such as words and sentences are in variable length, we claim that it is important for an RNN to dynamically adapt its timescales to the particulars of the input entities of various length. While this is trivial if the hierarchical boundary structure is provided (Sordoni et al., 2015), it has been a challenge for an RNN to discover the latent hierarchical structure in temporal data without explicit boundary information.

In this paper, we propose a novel multiscale RNN model, which can learn the hierarchical multiscale structure from temporal data without explicit boundary information. This model, called a *hierarchical multiscale recurrent neural network* (HM-RNN), does not assign fixed update rates, but adaptively determines proper update times corresponding to different abstraction levels of the layers. We find that this model tends to learn fine timescales for low-level layers and coarse timescales for high-level layers. To do this, we introduce a binary boundary detector at each layer. The boundary detector is turned on only at the time steps where a segment of the corresponding abstraction level is completely processed. Otherwise, i.e., during the within segment processing, it stays turned off. Using the hierarchical boundary states, we implement three operations, UPDATE, COPY and FLUSH, and choose one of them at each time step. The UPDATE operation is similar to the usual update rule of the long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997), except that it is executed sparsely according to the detected boundaries. The COPY operation simply copies the cell and hidden states of the previous time step. Unlike the leaky integration of the LSTM or the Gated Recurrent Unit (GRU) (Cho et al., 2014), the COPY operation retains the whole states without any loss of information. The FLUSH operation is executed when a boundary is detected, where it first ejects the summarized representation of the current segment to the upper layer and then reinitializes the states to start processing the next segment. Learning to select a proper operation at each time step and to detect the boundaries, the HM-RNN discovers the latent hierarchical structure of the sequences. We find that the straight-through estimator (Hinton, 2012; Bengio et al., 2013; Courbariaux et al., 2016) is efficient for training this model containing discrete variables.

We evaluate our model on two tasks: character-level language modelling and handwriting sequence generation. For the character-level language modelling, the HM-RNN achieves the state-of-the-art results on the Text8 dataset, and comparable results to the state-of-the-art on the Penn Treebank and Hutter Prize Wikipedia datasets. The HM-RNN also outperforms the standard RNN on the handwriting sequence generation using the IAM-OnDB dataset. In addition, we demonstrate that the hierarchical structure found by the HM-RNN is indeed very similar to the intrinsic structure observed in the data. The contributions of this paper are:

— We propose for the first time an RNN model that can learn a latent hierarchical structure of a sequence without using explicit boundary information.
— We show that it is beneficial to utilize the above structure through empirical evaluation.
— We show that the straight-through estimator is an efficient way of training a model containing discrete variables.
— We propose the *slope annealing* trick to improve the training procedure based on the straight-through estimator.

8.2 Related Work

Two notable early attempts inspiring our model are Schmidhuber (1992) and El Hihi and Bengio (1995). In these works, it is advocated to stack multiple layers of RNNs in a decreasing order of update frequency for computational and learning efficiency. In Schmidhuber (1992), the author shows a model that can self-organize a hierarchical multiscale structure. Particularly in El Hihi and Bengio (1995), the advantages of incorporating a priori knowledge, "*temporal dependencies are structured hierarchically*", into the RNN architecture is studied. The authors propose an RNN architecture that updates each layer with a fixed but different rate, called a hierarchical RNN.

LSTMs (Hochreiter and Schmidhuber, 1997) employ the multiscale update concept, where the hidden units have different forget and update rates and thus can operate with different timescales. However, unlike our model, these timescales are not organized hierarchically. Although the LSTM has a self-loop for the gradients that helps to capture the long-term dependencies by mitigating the vanishing gradient problem, in practice, it is still limited to a few hundred time steps due to the leaky integration by which the contents to memorize for a long-term is gradually diluted at every time step. Also, the model remains computationally expensive because it has to perform the update at every time step for each unit. However, our model is less prone to these problems because it learns a hierarchical structure such that, by design, high-level layers learn to perform less frequent updates than low-level layers. We hypothesize that this property mitigates the vanishing gradient problem more efficiently while also being computationally more efficient.

A more recent model, the clockwork RNN (CW-RNN) (Koutník et al., 2014) extends the hierarchical RNN (El Hihi and Bengio, 1995) and the NARX RNN (Lin et al., 1996) [1]. The CW-RNN tries to solve the issue of using soft timescales in the LSTM, by explicitly assigning hard timescales. In the CW-RNN, hidden units are partitioned into several modules, and different timescales are assigned to the modules such that a module i updates its hidden units at every $2^{(i-1)}$-th time step. The CW-RNN is computationally more efficient than the standard RNN including the LSTM since hidden units are updated only at the assigned clock rates. However, finding proper timescales in the CW-RNN remains as a challenge whereas our model *learns* the intrinsic timescales from the data. In the biscale RNNs (Chung et al., 2016), the authors proposed to model *layer-wise* timescales adaptively by having additional gating units, however this approach still relies on the *soft* gating mechanism like LSTMs.

Other forms of Hierarchical RNN (HRNN) architectures have been proposed in

1. The acronym NARX stands for Non-linear Auto-Regressive model with eXogenous inputs.

the cases where the explicit hierarchical boundary structure is provided. In Ling et al. (2015), after obtaining the word boundary via tokenization, the HRNN architecture is used for neural machine translation by modelling the characters and words using the first and second RNN layers, respectively. A similar HRNN architecture is also adopted in Sordoni et al. (2015) to model dialogue utterances. However, in many cases, hierarchical boundary information is not explicitly observed or expensive to obtain. Also, it is unclear how to deploy more layers than the number of boundary levels that is explicitly observed in the data.

While the above models focus on online prediction problems, where a prediction needs to be made by using only the past data, in some cases, predictions are made after observing the whole sequence. In this setting, the input sequence can be regarded as 1-D spatial data, convolutional neural networks with 1-D kernels are proposed in Kim (2014) and Kim et al. (2015) for language modelling and sentence classification. Also, in Chan et al. (2016) and Bahdanau et al. (2016), the authors proposed to obtain high-level representation of the sequences of reduced length by repeatedly merging or pooling the lower-level representation of the sequences.

Hierarchical RNN architectures have also been used to discover the segmentation structure in sequences (Fernández et al., 2007; Kong et al., 2015). It is however different to our model in the sense that they optimize the objective with explicit labels on the hierarchical segments while our model discovers the intrinsic structure only from the sequences without segment label information.

The COPY operation used in our model can be related to Zoneout (Krueger et al., 2016) which is a recurrent generalization of stochastic depth (Huang et al., 2016). In Zoneout, an identity transformation is randomly applied to each hidden unit at each time step according to a Bernoulli distribution. This results in occasional copy operations of the previous hidden states. While the focus of Zoneout is to propose a regularization technique similar to dropout (Srivastava et al., 2014) (where the regularization strength is controlled by a hyperparameter), our model learns (a) to dynamically determine when to copy from the context inputs and (b) to discover the hierarchical multiscale structure and representation. Although the main goal of our proposed model is not regularization, we found that our model also shows very good generalization performance.

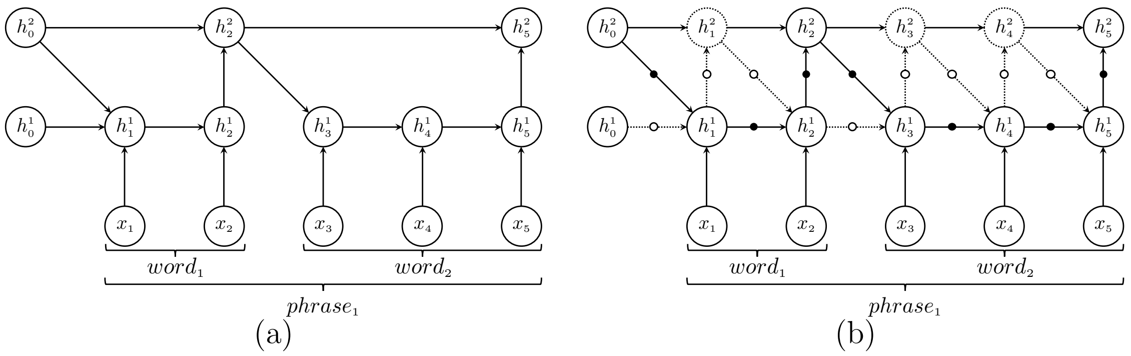


Figure 8.1: (a) The HRNN architecture, which requires the knowledge of the hierarchical boundaries. (b) The HM-RNN architecture that discovers the hierarchical multiscale structure in the data.

8.3 Hierarchical Multiscale Recurrent Neural Networks

8.3.1 Motivation

To begin with, we provide an example of how a stacked RNN can model temporal data in an ideal setting, i.e., when the hierarchy of segments is provided (Sordoni et al., 2015; Ling et al., 2015). In Figure 8.1 (a), we depict a hierarchical RNN (HRNN) for language modelling with two layers: the first layer receives characters as inputs and generates word-level representations (C2W-RNN), and the second layer takes the word-level representations as inputs and yields phrase-level representations (W2P-RNN).

As shown, by means of the provided end-of-word labels, the C2W-RNN obtains word-level representation after processing the last character of each word and passes the word-level representation to the W2P-RNN. Then, the W2P-RNN performs an update of the phrase-level representation. Note that the hidden states of the W2P-RNN remains unchanged while all the characters of a word are processed by the C2W-RNN. When the C2W-RNN starts to process the next word, its hidden states are reinitialized using the latest hidden states of the W2P-RNN, which contain summarized representation of all the words that have been processed by that time step, in that phrase.

From this simple example, we can see the advantages of having a hierarchical multiscale structure: (1) as the W2P-RNN is updated at a much slower update rate than the C2W-RNN, a considerable amount of computation can be saved,

(2) gradients are backpropagated through a much smaller number of time steps, and (3) layer-wise capacity control becomes possible (e.g., use a smaller number of hidden units in the first layer which models short-term dependencies but whose updates are invoked much more often).

Can an RNN discover such hierarchical multiscale structure without explicit hierarchical boundary information? Considering the fact that the boundary information is difficult to obtain (for example, consider languages where words are not always cleanly separated by spaces or punctuation symbols, and imperfect rules are used to separately perform segmentation) or usually not provided at all, this is a legitimate problem. It gets worse when we consider higher-level concepts which we would like the RNN to discover autonomously. In Section 8.2, we discussed the limitations of the existing RNN models under this setting, which either have to update all units at every time step or use fixed update frequencies (El Hihi and Bengio, 1995; Koutník et al., 2014). Unfortunately, this kind of approach is not well suited to the case where different segments in the hierarchical decomposition have different lengths: for example, different words have different lengths, so a fixed hierarchy would not update its upper-level units in synchrony with the natural boundaries in the data.

8.3.2 The Proposed Model

A key element of our model is the introduction of a parametrized boundary detector, which outputs a binary value, in each layer of a stacked RNN, and learns when a segment should end in such a way to optimize the overall target objective. Whenever the boundary detector is turned on at a time step of layer ℓ (i.e., when the boundary state is 1), the model considers this to be the end of a segment corresponding to the latent abstraction level of that layer (e.g., word or phrase) and feeds the summarized representation of the detected segment into the upper layer $(\ell+1)$. Using the boundary states, at each time step, each layer selects one of the following operations: UPDATE, COPY or FLUSH. The selection is determined by (1) the boundary state of the current time step in the layer below $z_t^{\ell-1}$ and (2) the boundary state of the previous time step in the same layer z_{t-1}^{ℓ}.

In the following, we describe an HM-RNN based on the LSTM update rule. We call this model a hierarchical multiscale LSTM (HM-LSTM). Consider an HM-

LSTM model of L layers ($\ell = 1, \ldots, L$) which, at each layer ℓ, performs the following update at time step t:

$$\mathbf{h}_t^\ell, \mathbf{c}_t^\ell, z_t^\ell = f_{\text{HM-LSTM}}^\ell(\mathbf{c}_{t-1}^\ell, \mathbf{h}_{t-1}^\ell, \mathbf{h}_t^{\ell-1}, \mathbf{h}_{t-1}^{\ell+1}, z_{t-1}^\ell, z_t^{\ell-1}). \tag{8.1}$$

Here, $\mathbf{h}$ and $\mathbf{c}$ denote the hidden and cell states, respectively. The function $f_{\text{HM-LSTM}}^\ell$ is implemented as follows. First, using the two boundary states z_{t-1}^ℓ and $z_t^{\ell-1}$, the cell state is updated by:

$$\mathbf{c}_t^\ell = \begin{cases} \mathbf{f}_t^\ell \odot \mathbf{c}_{t-1}^\ell + \mathbf{i}_t^\ell \odot \mathbf{g}_t^\ell & \text{if } z_{t-1}^\ell = 0 \text{ and } z_t^{\ell-1} = 1 \text{ (UPDATE)} \\ \mathbf{c}_{t-1}^\ell & \text{if } z_{t-1}^\ell = 0 \text{ and } z_t^{\ell-1} = 0 \text{ (COPY)} \\ \mathbf{i}_t^\ell \odot \mathbf{g}_t^\ell & \text{if } z_{t-1}^\ell = 1 \text{ (FLUSH)}, \end{cases} \tag{8.2}$$

and then the hidden state is obtained by:

$$\mathbf{h}_t^\ell = \begin{cases} \mathbf{h}_{t-1}^\ell & \text{if COPY}, \\ \mathbf{o}_t^\ell \odot \texttt{tanh}(\mathbf{c}_t^\ell) & \text{otherwise}. \end{cases} \tag{8.3}$$

Here, $(\mathbf{f}, \mathbf{i}, \mathbf{o})$ are forget, input, output gates, and $\mathbf{g}$ is a cell proposal vector. Note that unlike the LSTM, it is not necessary to compute these gates and cell proposal values at every time step. For example, in the case of the COPY operation, we do not need to compute any of these values and thus can save computations.

The COPY operation, which simply performs $(\mathbf{c}_t^\ell, \mathbf{h}_t^\ell) \leftarrow (\mathbf{c}_{t-1}^\ell, \mathbf{h}_{t-1}^\ell)$, implements the observation that an upper layer should keep its state unchanged until it receives the summarized input from the lower layer. The UPDATE operation is performed to update the summary representation of the layer ℓ if the boundary $z_t^{\ell-1}$ is detected from the layer below but the boundary z_{t-1}^ℓ was not found at the previous time step. Hence, the UPDATE operation is executed sparsely unlike the standard RNNs where it is executed at every time step, making it computationally inefficient. If a boundary is detected, the FLUSH operation is executed. The FLUSH operation consists of two sub-operations: (a) EJECT to pass the current state to the upper layer and then (b) RESET to reinitialize the state before starting to read a new segment. This operation implicitly forces the upper layer to absorb the summary information of the lower layer segment, because otherwise it will be lost. Note that the FLUSH operation is a *hard* reset in the sense that it completely erases all the previous states of the same layer, which is different from the *soft* reset or *soft* forget operation in the GRU or LSTM.

Whenever needed (depending on the chosen operation), the gate values $(\mathbf{f}_t^\ell, \mathbf{i}_t^\ell, \mathbf{o}_t^\ell)$, the cell proposal $\mathbf{g}_t^\ell$, and the pre-activation of the boundary detector $\tilde{z}_t^\ell$[2] are then obtained by:

$$\begin{pmatrix} \mathbf{f}_t^\ell \\ \mathbf{i}_t^\ell \\ \mathbf{o}_t^\ell \\ \mathbf{g}_t^\ell \\ \tilde{z}_t^\ell \end{pmatrix} = \begin{pmatrix} \texttt{sigm} \\ \texttt{sigm} \\ \texttt{sigm} \\ \texttt{tanh} \\ \texttt{hard sigm} \end{pmatrix} f_{\texttt{slice}} \left(\mathbf{s}_t^{\text{recurrent}(\ell)} + \mathbf{s}_t^{\text{top-down}(\ell)} + \mathbf{s}_t^{\text{bottom-up}(\ell)} + \mathbf{b}^{(\ell)} \right) \tag{8.4}$$

where

$$\begin{aligned} \mathbf{s}_t^{\text{recurrent}(\ell)} &= U_\ell^\ell \mathbf{h}_{t-1}^\ell, & (8.5) \\ \mathbf{s}_t^{\text{top-down}(\ell)} &= z_{t-1}^\ell U_{\ell+1}^\ell \mathbf{h}_{t-1}^{\ell+1}, & (8.6) \\ \mathbf{s}_t^{\text{bottom-up}(\ell)} &= z_t^{\ell-1} W_{\ell-1}^\ell \mathbf{h}_t^{\ell-1}. & (8.7) \end{aligned}$$

Here, we use $W_i^j \in \mathbb{R}^{(4dim(\mathbf{h}^\ell)+1)\times dim(\mathbf{h}^{\ell-1})}, U_i^j \in \mathbb{R}^{(4dim(\mathbf{h}^\ell)+1)\times dim(\mathbf{h}^\ell)}$ to denote state transition parameters from layer i to layer j, and $\mathbf{b} \in \mathbb{R}^{4dim(\mathbf{h}^\ell)+1}$ is a bias term. In the last layer L, the top-down connection is ignored, and we use $\mathbf{h}_t^0 = \mathbf{x}_t$. Also, we do not use the boundary detector for the last layer. The `hard sigm` is defined by $\texttt{hard sigm}(x) = \max\left(0, \min\left(1, \frac{ax+1}{2}\right)\right)$ with a being the slope variable.

Unlike the standard LSTM, the HM-LSTM has a top-down connection from $(\ell + 1)$ to ℓ, which is allowed to be activated only if a boundary is detected at the previous time step of the layer ℓ (see Eq. 8.6). This makes the layer ℓ to be initialized with more long-term information after the boundary is detected and execute the FLUSH operation. In addition, the input from the lower layer $(\ell - 1)$ becomes effective only when a boundary is detected at the current time step in the layer $(\ell - 1)$ due to the binary gate $z_t^{\ell-1}$. Figure 8.2 (left) shows the gating mechanism of the HM-LSTM at time step t.

Finally, the binary boundary state z_t^ℓ is obtained by:

$$z_t^\ell = f_{\texttt{bound}}(\tilde{z}_t^\ell). \tag{8.8}$$

2. $\tilde{z}_t^\ell$ can also be implemented as a function of $\mathbf{h}_t^\ell$, e.g., $\tilde{z}_t^\ell = \texttt{hard sigm}(U\mathbf{h}_t^\ell)$.

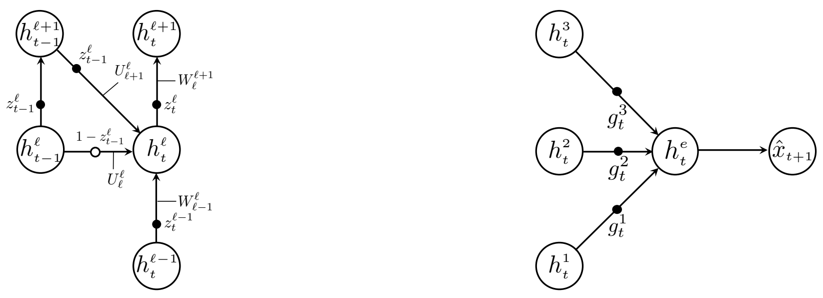


Figure 8.2: Left: The gating mechanism of the HM-RNN. Right: The output module when $L = 3$.

For the binarization function $f_{\text{bound}} : \mathbb{R} \rightarrow \{0, 1\}$, we can either use a deterministic step function:

$$z_t^{\ell} = \begin{cases} 1 & \text{if } \tilde{z}_t^{\ell} > 0.5 \\ 0 & \text{otherwise,} \end{cases} \tag{8.9}$$

or sample from a Bernoulli distribution $z_t^{\ell} \sim \text{Bernoulli}(\tilde{z}_t^{\ell})$. Although this binary decision is a key to our model, it is usually difficult to use stochastic gradient descent to train such model with discrete decisions as it is not differentiable.

8.3.3 Computing Gradient of Boundary Detector

Training neural networks with discrete variables requires more efforts since the standard backpropagation is no longer applicable due to the non-differentiability. Among a few methods for training a neural network with discrete variables such as the REINFORCE (Williams, 1992; Mnih and Gregor, 2014) and the straight-through estimator (Hinton, 2012; Bengio et al., 2013), we use the straight-through estimator to train our model. The straight-through estimator is a biased estimator because the non-differentiable function used in the forward pass (i.e., the step function in our case) is replaced by a differentiable function during the backward pass (i.e., the hard sigmoid function in our case). The straight-through estimator, however, is much simpler and often works more efficiently in practice than other unbiased but high-variance estimators such as the REINFORCE. The straight-through estimator has also been used in Courbariaux et al. (2016) and Vezhnevets et al. (2016).

The Slope Annealing Trick. In our experiment, we use the slope annealing trick to reduce the bias of the straight-through estimator. The idea is to reduce the discrepancy between the two functions used during the forward pass and the backward pass. That is, by gradually increasing the slope a of the hard sigmoid function, we make the hard sigmoid be close to the step function. Note that starting with a high slope value from the beginning can make the training difficult while it is more applicable later when the model parameters become more stable. In our experiments, starting from slope $a = 1$, we slowly increase the slope until it reaches a threshold with an appropriate scheduling.

8.4 Experiments

We evaluate the proposed model on two tasks, character-level language modelling and handwriting sequence generation. Character-level language modelling is a representative example of discrete sequence modelling, where the discrete symbols form a distinct hierarchical multiscale structure. The performance on real-valued sequences is tested on the handwriting sequence generation in which a relatively clear hierarchical multiscale structure exists compared to other data such as speech signals.

8.4.1 Character-Level Language Modelling

A sequence modelling task aims at learning the probability distribution over sequences by minimizing the negative log-likelihood of the training sequences:

$$\min_{\theta} -\frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T^n} \log p\left(x_t^n \mid x_{<t}^n; \theta\right), \tag{8.10}$$

where θ is the model parameter, N is the number of training sequences, and T^n is the length of the n-th sequence. A symbol at time t of sequence n is denoted by x_t^n, and $x_{<t}^n$ denotes all previous symbols at time t. We evaluate our model on three benchmark text corpora: (1) Penn Treebank, (2) Text8 and (3) Hutter Prize Wikipedia. We use the bits-per-character (BPC), $\mathbb{E}[-\log_2 p(x_{t+1} \mid x_{\leq t})]$, as the evaluation metric.

Penn Treebank		
Model		BPC
Norm-stabilized RNN	(Krueger and Memisevic, 2015)	1.48
CW-RNN	(Koutník et al., 2014)	1.46
HF-MRNN	(Mikolov et al., 2012)	1.41
MI-RNN	(Wu et al., 2016)	1.39
ME n-gram	(Mikolov et al., 2012)	1.37
BatchNorm LSTM	(Cooijmans et al., 2016)	1.32
Zoneout RNN	(Krueger et al., 2016)	1.27
HyperNetworks	(Ha et al., 2016)	1.27
LayerNorm HyperNetworks	(Ha et al., 2016)	**1.23**
LayerNorm CW-RNN†		1.40
LayerNorm LSTM†		1.29
LayerNorm HM-LSTM	Sampling	1.27
LayerNorm HM-LSTM	Soft*	1.27
LayerNorm HM-LSTM	Step Fn.	1.25
LayerNorm HM-LSTM	Step Fn. & Slope Annealing	1.24

Table 8.1: BPC on the Penn Treebank test set. (∗) This model is a variant of the HM-LSTM that does not discretize the boundary detector states. (†) These models are implemented by the authors to evaluate the performance using layer normalization (Ba et al., 2016) with the additional output module.

Model We use a model consisting of an input embedding layer, an RNN module and an output module. The input embedding layer maps each input symbol into 128-dimensional continuous vector without using any non-linearity. The RNN module is the HM-LSTM, described in Section 8.3, with three layers. The output module is a feedforward neural network with two layers, an output embedding layer and a softmax layer. Figure 8.2 (right) shows a diagram of the output module. At each time step, the output embedding layer receives the hidden states of the three RNN layers as input. In order to adaptively control the importance of each layer at each time step, we also introduce three scalar gating units $g_t^\ell \in \mathbb{R}$ to each of the layer outputs:

$$g_t^\ell = \mathtt{sigm}(\mathbf{w}^\ell[\mathbf{h}_t^1; \cdots; \mathbf{h}_t^L]), \tag{8.11}$$

Hutter Prize Wikipedia	
Model	**BPC**
Stacked LSTM (Graves, 2013)	1.67
MRNN (Sutskever et al., 2011)	1.60
GF-LSTM (Chung et al., 2015)	1.58
Grid-LSTM (Kalchbrenner et al., 2015)	1.47
MI-LSTM (Wu et al., 2016)	1.44
Recurrent Memory Array Structures (Rocki, 2016a)	1.40
SF-LSTM (Rocki, 2016b)‡	1.37
HyperNetworks (Ha et al., 2016)	1.35
LayerNorm HyperNetworks (Ha et al., 2016)	1.34
Recurrent Highway Networks (Zilly et al., 2016)	1.32
LayerNorm LSTM†	1.39
HM-LSTM	1.34
LayerNorm HM-LSTM	1.32
PAQ8hp12 (Mahoney, 2005)	1.32
decomp8 (Mahoney, 2009)	**1.28**

Table 8.2: BPC on the Hutter Prize Wikipedia test set (right). (†) These models are implemented by the authors to evaluate the performance using layer normalization (Ba et al., 2016) with the additional output module. (‡) This method uses test error signals for predicting the next characters, which makes it not comparable to other methods that do not.

where $\mathbf{w}^\ell \in \mathbb{R}^{\sum_{\ell=1}^{L} dim(\mathbf{h}^\ell)}$ is the weight parameter. The output embedding $\mathbf{h}_t^{\mathrm{e}}$ is computed by:

$$\mathbf{h}_t^{\mathrm{e}} = \texttt{ReLU}\left(\sum_{\ell=1}^{L} g_t^\ell W_\ell^{\mathrm{e}} \mathbf{h}_t^\ell\right), \tag{8.12}$$

where $L = 3$ and $\texttt{ReLU}(x) = \max(0, x)$ (Nair and Hinton, 2010). Finally, the probability distribution for the next target character is computed by the softmax function, $\texttt{softmax}(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$, where each output class is a character.

Penn Treebank We process the Penn Treebank dataset (Marcus et al., 1993) by following the procedure introduced in Mikolov et al. (2012). Each update is done by using a mini-batch of 64 examples of length 100 to prevent the memory overflow problem when unfolding the RNN in time for backpropagation. The last hidden state of a sequence is used to initialize the hidden state of the next sequence to approximate the full backpropagation. We train the model using Adam (Kingma

Text8	
Model	**BPC**
td-LSTM (Zhang et al., 2016)	1.63
HF-MRNN (Mikolov et al., 2012)	1.54
MI-RNN (Wu et al., 2016)	1.52
Skipping-RNN (Pachitariu and Sahani, 2013)	1.48
MI-LSTM (Wu et al., 2016)	1.44
BatchNorm LSTM (Cooijmans et al., 2016)	1.36
HM-LSTM	1.32
LayerNorm HM-LSTM	**1.29**

Table 8.3: BPC on the Text8 test set.

and Ba, 2014) with an initial learning rate of 0.002. We divide the learning rate by a factor of 50 when the validation negative log-likelihood stopped decreasing. The norm of the gradient is clipped with a threshold of 1 (Mikolov et al., 2010; Pascanu et al., 2012). We also apply layer normalization (Ba et al., 2016) to our models. For all of the character-level language modelling experiments, we apply the same procedure, but only change the number of hidden units, mini-batch size and the initial learning rate.

For the Penn Treebank dataset, we use 512 units in each layer of the HM-LSTM and for the output embedding layer. In Table 8.1, we compare the test BPCs of four variants of our model to other baseline models. Note that the HM-LSTM using the step function for the hard boundary decision outperforms the others using either *sampling* or *soft* boundary decision (i.e., hard sigmoid). The test BPC is further improved with the slope annealing trick, which reduces the bias of the straight-through estimator. We increased the slope a with the following schedule $a = \min(5, 1 + 0.04 \cdot N_{epoch})$, where N_{epoch} is the maximum number of epochs. The HM-LSTM achieves test BPC score of 1.24. For the remaining tasks, we fixed the hard boundary decision using the step function without slope annealing due to the difficulty of finding a good annealing schedule on large-scale datasets.

Text8 The Text8 dataset (Mahoney, 2009) consists of 100M characters extracted from the Wikipedia corpus. Text8 contains only alphabets and spaces, and thus we have total 27 symbols. In order to compare with other previous works, we follow the data splits used in Mikolov et al. (2012). We use 1024 units for each HM-LSTM layer and 2048 units for the output embedding layer. The mini-batch size and the

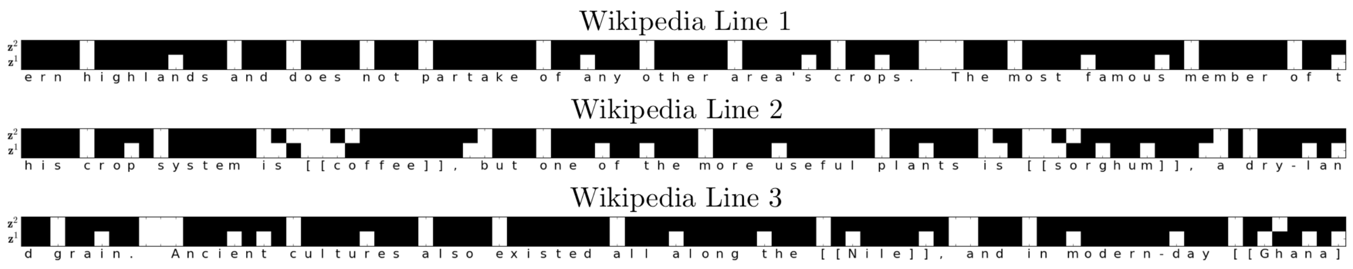


Figure 8.3: Hierarchical multiscale structure in the Wikipedia dataset captured by the boundary detectors of the HM-LSTM.

initial learning rate are set to 128 and 0.001, respectively. The results are shown in Table 8.3. The HM-LSTM obtains the state-of-the-art test BPC 1.29.

Hutter Prize Wikipedia The Hutter Prize Wikipedia (`enwik8`) dataset (Hutter, 2012) contains 205 symbols including XML markups and special characters. We follow the data splits used in Graves (2013) where the first 90M characters are used to train the model, the next 5M characters for validation, and the remainders for the test set. We use the same model size, mini-batch size and the initial learning rate as in the Text8. In Table 8.2, we show the HM-LSTM achieving the test BPC 1.32, which is a tie with the state-of-the-art result among the neural models. Although the neural models, show remarkable performances, their compression performance is still behind the best models such as PAQ8hp12 (Mahoney, 2005) and decomp8 (Mahoney, 2009).

Visualizing Learned Hierarchical Multiscale Structure In Figure 8.3 and 8.4, we visualize the boundaries detected by the boundary detectors of the HM-LSTM while reading a character sequence of total length 270 taken from the validation set of either the Penn Treebank or Hutter Prize Wikipedia dataset. Due to the page width limit, the figure contains the sequence partitioned into three segments of length 90. The white blocks indicate boundaries $z_t^\ell = 1$ while the black blocks indicate the non-boundaries $z_t^\ell = 0$.

Interestingly in both figures, we can observe that the boundary detector of the first layer, z^1, tends to be turned on when it sees a space or after it sees a space, which is a reasonable breakpoint to separate between words. This is somewhat surprising because the model self-organizes this structure without any explicit boundary information. In Figure 8.3, we observe that the z^1 tends to detect the boundaries of the words but also fires within the words, where the z^2 tends to fire when it sees either an end of a word or $2, 3$-grams. In Figure 8.4, we also see flushing in the middle of a word, e.g., "tele-FLUSH-phone". Note that "tele" is a prefix after which a various number of postfixes can follow. From these, it seems that the model uses to some extent the concept of *surprise* to learn the boundary. Although interpretation of the second layer boundaries is not as apparent as the first layer boundaries, it seems to segment at reasonable semantic / syntactic boundaries, e.g., "consumers may" - "want to move their telephones a" - "little closer to the tv set <unk>", and so on.

Another remarkable point is the fact that we do not pose any constraint on the number of boundaries that the model can fire up. The model, however, learns that it is more beneficial to delay the information ejection to some extent. This is somewhat counterintuitive because it might look more beneficial to feed the fresh update to the upper layers at every time step without any delay. We conjecture the reason that the model works in this way is due to the FLUSH operation that poses an implicit constraint on the frequency of boundary detection, because it contains both a reward (feeding fresh information to upper layers) and a penalty (erasing accumulated information). The model finds an optimal balance between the reward and the penalty.

To understand the update mechanism more intuitively, in Figure 8.4, we also depict the heatmap of the ℓ^2-norm of the hidden states along with the states of the boundary detectors. As we expect, we can see that there is no change in the norm value within segments due to the COPY operation. Also, the color of $\|\mathbf{h}^1\|$ changes quickly (at every time step) because there is no COPY operation in the first layer. The color of $\|\mathbf{h}^2\|$ changes less frequently based on the states of z_t^1 and z_{t-1}^2. The color of $\|\mathbf{h}^3\|$ changes even slowly, i.e., only when $z_t^2 = 1$.

A notable advantage of the proposed architecture is that the internal process of the RNN becomes more interpretable. For example, we can substitute the states of z_t^1 and z_{t-1}^2 into Eq. 8.2 and infer which operation among the UPDATE, COPY and FLUSH was applied to the second layer at time step t. We can also inspect the update frequencies of the layers simply by counting how many UPDATE and FLUSH operations were made in each layer. For example in Figure 8.4, we see that the first layer updates at every time step (which is 270 UPDATE operations), the second layer updates 56 times, and only 9 updates has made in the third layer.

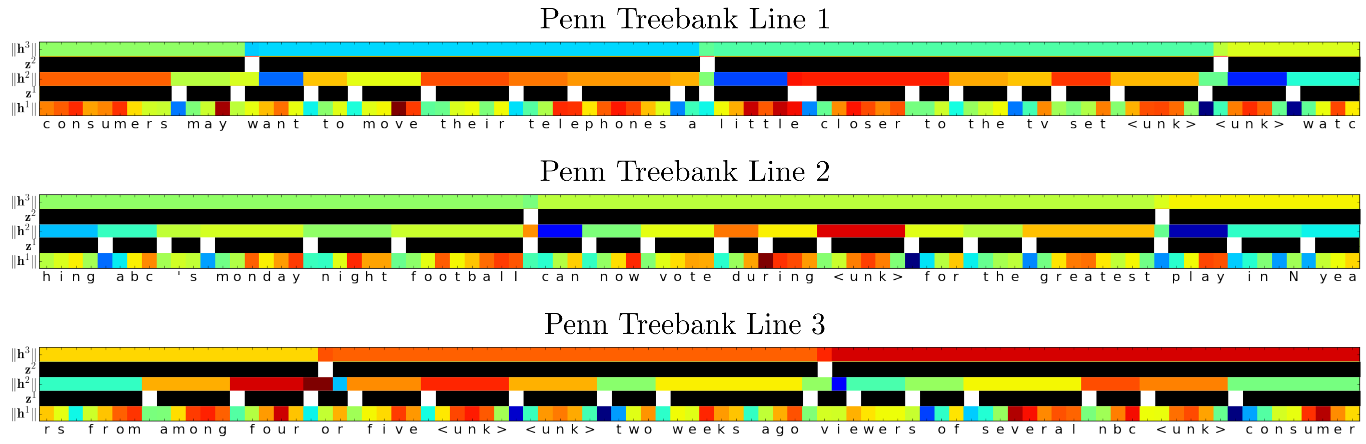


Figure 8.4: The ℓ^2-norm of the hidden states shown together with the states of the boundary detectors of the HM-LSTM.

Note that, by design, the first layer performs UPDATE operation at every time step and then the number of UPDATE operations decreases as the layer level increases. In this example, the total number of updates is 335 for the HM-LSTM which is 60% of reduction from the 810 updates of the standard RNN architecture.

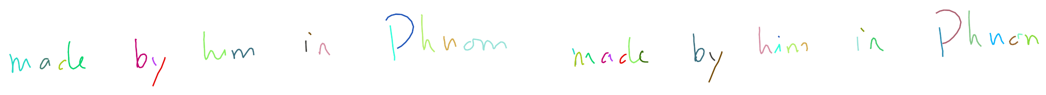
8.4.2 Handwriting Sequence Generation

We extend the evaluation of the HM-LSTM to a real-valued sequence modelling task using IAM-OnDB (Liwicki and Bunke, 2005) dataset. The IAM-OnDB dataset consists of $12,179$ handwriting examples, each of which is a sequence of (x, y) coordinate and a binary indicator p for pen-tip location, giving us $(x_{1:T^n}, y_{1:T^n}, p_{1:T^n})$, where n is an index of a sequence. At each time step, the model receives (x_t, y_t, p_t), and the goal is to predict $(x_{t+1}, y_{t+1}, p_{t+1})$. The pen-up $(p_t = 1)$ indicates an end of a stroke, and the pen-down $(p_t = 0)$ indicates that a stroke is in progress. There is usually a large shift in the (x, y) coordinate to start a new stroke after the pen-up happens. We remove all sequences whose length is shorter than 300. This leaves us $10,465$ sequences for training, 581 for validation, 582 for test. The average length of the sequences is 648. We normalize the range of the (x, y) coordinates separately with the mean and standard deviation obtained from the training set. We use the mini-batch size of 32, and the initial learning rate is set to 0.0003.

We use the same model architecture as used in the character-level language model, except that the output layer is modified to predict real-valued outputs. We use the mixture density network as the output layer following Graves (2013), and

IAM-OnDB	
Model	**Average Log-Likelihood**
Standard LSTM	1081
HM-LSTM	1137
HM-LSTM & Slope Annealing	**1167**

Table 8.4: Average log-likelihood per sequence on the IAM-OnDB test set.



Visualization by segments using the ground truth of pen-tip location

Visualization by segments using the states of z^2

Figure 8.5: The visualization by segments based on either the given pen-tip location or states of the z^2.

use 400 units for each HM-LSTM layer and for the output embedding layer. In Table 8.4, we compare the log-likelihood averaged over the test sequences of the IAM-OnDB dataset. We observe that the HM-LSTM outperforms the standard LSTM. The slope annealing trick further improves the test log-likelihood of the HM-LSTM into 1167 in our setting. In this experiment, we increased the slope a with the following schedule $a = \min(3, 1 + 0.004 \cdot N_{epoch})$. In Figure 8.5, we let the HM-LSTM to read a randomly picked validation sequence and present the visualization of handwriting examples by segments based on either the states of z^2 or the states of pen-tip location [3].

8.5 Conclusion

In this paper, we proposed the HM-RNN that can capture the latent hierarchical structure of the sequences. We introduced three types of operations to the RNN, which are the COPY, UPDATE and FLUSH operations. In order to implement these operations, we introduced a set of binary variables and a novel update rule that is dependent on the states of these binary variables. Each binary variable is learned to find segments at its level, therefore, we call this binary variable, a boundary detector. On the character-level language modelling, the HM-LSTM achieved

3. The plot function could be found at blog.otoro.net/2015/12/12/handwriting-generation-demo-in-tensorflow/.

state-of-the-art result on the Text8 dataset and comparable results to the state-of-the-art results on the Penn Treebank and Hutter Prize Wikipedia datasets. Also, the HM-LSTM outperformed the standard LSTM on the handwriting sequence generation. Our results and analysis suggest that the proposed HM-RNN can discover the latent hierarchical structure of the sequences and can learn efficient hierarchical multiscale representation that leads to better generalization performance.

9 Prologue to Fourth Article

9.1 Article Details

A Recurrent Latent Variable Model for Sequential Data. Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarh Goel, Aaron Courville and Yoshua Bengio, *Proceedings of the 28th Advances in Neural Information Processing Systems (NIPS 2015).*

Personal Contribution.
The idea of combining a variational auto-encoder with an RNN came out during a meeting between Yoshua Bengio and myself. I proposed to condition the prior distribution of the latent variables on the hidden state of a context RNN and to use an RNN encoder and RNN decoder, which replace the feedforward encoder and decoder of a variational auto-encoder (VAE) (Kingma and Welling, 2013; Rezende et al., 2014). Yoshua Bengio suggested using a single RNN that provides the temporal context to the encoder, decoder and the conditional prior distribution. Laurent Dinh contributed on reviewing the theoretical aspect of the proposed model. Kyle Kastner handled most of the data engineering part, which was crucial for speech datasets that can easily occupy a lot of space in the memory and file systems. I programmed the main algorithm, and Kratarth Goel and Kyle Kastner contributed a significant amount of efforts on writing necessary software and pipelines. I did the most of the experiments on the speech generation tasks, and Kratarth Goel did the experiment on the handwriting generation task. Kyle Kastner contributed on the visualization of the examples shown in the paper. The introductory part was written by Aaron Courville, and I wrote the rest of the paper with Laurent Dinh and Kyle Kastner. Yoshua Bengio did the general editing. This work was also contributed indirectly by the members of MILA [1] speech synthesis team.

1. Montreal Institute of Learning Algorithms

9.2 Context

Generative RNNs (Graves, 2013) showed promising results on many tasks that involve unsupervised learning of sequential data such as natural language (Sutskever et al., 2011; Graves, 2013) or handwriting examples (Graves et al., 2008; Graves, 2013). However, in speech modelling tasks, such as generating spoken utterances, the RNNs struggled with the under-fitting problem. Using more complicated output functions such as mixture density networks (Graves, 2013) can be helpful to some extent, however, the sample quality of the generated examples are not as good as the original examples.

In feedforward neural networks, a latent variable model named after the auto-encoder, the *variational auto-encoder* (VAE) (Kingma and Welling, 2013; Rezende et al., 2014), showed promising performance on modelling toy image datasets. Latent variable models were also extended to RNNs. A popular RNN-based latent variable model is the deep recurrent attentive writer (DRAW) (Gregor et al., 2015) which was tested on the toy image datasets that VAEs were tested on. There are also *variational recurrent auto-encoders* (Fabius et al., 2014) and *stochastic recurrent neural networks* (Bayer and Osendorfer, 2014) that were evaluated on small audio datasets or motion capture dataset.

9.3 Contributions

In this work, we proposed a new type of latent variable model that is based on an RNN and can model large-scale sequential datasets. There are some challenges with directly applying VAEs to sequences. The first issue is handling variable-length examples, which is often the case in sequence modelling tasks. The second issue is that VAEs have originally tried on toy image datasets with a size of 784 pixels (the dimensionality of an example from the MNIST dataset) or 1024 pixels (the dimensionality of a gray-scale image from the CIFAR10 dataset). Therefore, whether VAEs could scale-up to model raw waveforms was unclear since the dimensionality of a speech signal can be orders of magnitude larger than the toy image data and of a very different nature. The proposed VRNN is a straightforward extension of a VAE, where each component of the VAE such as the encoder and decoder are conditioned by the temporal context, which is the hidden state of an RNN.

10 Variational Recurrent Neural Networks

10.1 Introduction

Learning generative models of sequences is a long-standing machine learning challenge and historically the domain of dynamic Bayesian networks (DBNs) such as hidden Markov models (HMMs) and Kalman filters. The dominance of DBN-based approaches has been recently overturned by a resurgence of interest in recurrent neural network (RNN) based approaches. An RNN is a special type of neural network that is able to handle both variable-length input and output. By training an RNN to predict the next output in a sequence, given all previous outputs, it can be used to model joint probability distribution over sequences.

Both RNNs and DBNs consist of two parts: (1) a transition function that determines the evolution of the internal hidden state, and (2) a mapping from the state to the output. There are, however, a few important differences between RNNs and DBNs.

DBNs have typically been limited either to relatively simple state transition structures (e.g., linear models in the case of the Kalman filter) or to relatively simple internal state structure (e.g., the HMM state space consists of a single set of mutually exclusive states). RNNs, on the other hand, typically possess both a richly distributed internal state representation and flexible non-linear transition functions. These differences give RNNs extra expressive power in comparison to DBNs. This expressive power and the ability to train via error backpropagation are the key reasons why RNNs have gained popularity as generative models for highly structured sequential data.

In this paper, we focus on another important difference between DBNs and RNNs. While the hidden state in DBNs is expressed in terms of random variables, the internal transition structure of the standard RNN is entirely deterministic. The only source of randomness or variability in the RNN is found in the conditional output probability model. We suggest that this can be an inappropriate way to model the kind of variability observed in highly structured data, such as natural speech, which is characterized by strong and complex dependencies among the output variables at different time steps. We argue, as have others (Boulanger-Lewandowski et al., 2012; Bayer and Osendorfer, 2014), that these complex dependencies cannot be modelled efficiently by the output probability models used in standard RNNs, which include either a simple unimodal distribution or a mixture of unimodal distributions.

We propose the use of high-level latent random variables to model the variability observed in the data. In the context of standard neural network models for non-sequential data, the variational auto-encoder (VAE) (Kingma and Welling, 2013; Rezende et al., 2014) offers an interesting combination of highly flexible non-linear mapping between the latent random state and the observed output and effective approximate inference. In this paper, we propose to extend the VAE into a recurrent framework for modelling high-dimensional sequences. The VAE can model complex multimodal distributions, which will help when the underlying true data distribution consists of multimodal conditional distributions. We call this model a *variational RNN* (VRNN).

A natural question to ask is: how do we encode observed variability via latent random variables? The answer to this question depends on the nature of the data itself. In this work, we are mainly interested in highly structured data that often arises in AI applications. By highly structured, we mean that the data is characterized by two properties. Firstly, there is a relatively high signal-to-noise ratio, meaning that the vast majority of the variability observed in the data is due to the signal itself and cannot reasonably be considered as noise. Secondly, there exists a complex relationship between the underlying factors of variation and the observed data. For example, in speech, the vocal qualities of the speaker have a strong but complicated influence on the audio waveform, affecting the waveform in a consistent manner across frames.

With these considerations in mind, we suggest that our model variability should induce *temporal dependencies across time steps*. Thus, like DBN models such as HMMs and Kalman filters, we model the dependencies between the latent random variables across time steps. While we are not the first to propose integrating random variables into the RNN hidden state (Boulanger-Lewandowski et al., 2012;

Bayer and Osendorfer, 2014; Fabius et al., 2014; Gregor et al., 2015), we believe we are the first to integrate the dependencies between the latent random variables at neighboring time steps.

We evaluate the proposed VRNN model against other RNN-based models – including a VRNN model without introducing temporal dependencies between the latent random variables – on two challenging sequential data types: natural speech and handwriting. We demonstrate that for the speech modelling tasks, the VRNN-based models significantly outperform the RNN-based models and the VRNN model that does not integrate temporal dependencies between latent random variables.

10.2 Background

10.2.1 Generative Sequence modelling with Recurrent Neural Networks

An RNN can take as input a variable-length sequence $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$ by recursively processing each symbol while maintaining its internal hidden state $\mathbf{h}$. At each time step t, the RNN reads the symbol $\mathbf{x}_t \in \mathbb{R}^d$ and updates its hidden state $\mathbf{h}_t \in \mathbb{R}^p$ by:

$$\mathbf{h}_t = f_\theta\left(\mathbf{x}_t, \mathbf{h}_{t-1}\right), \tag{10.1}$$

where f is a deterministic non-linear transition function, and θ is the parameter set of f. The transition function f can be implemented with gated activation functions such as long short-term memory (LSTM, Hochreiter and Schmidhuber, 1997) or gated recurrent unit (GRU, Cho et al., 2014). RNNs model sequences by parameterizing a factorization of the joint sequence probability distribution as a product of conditional probabilities such that:

$$\begin{aligned} p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T) &= \prod_{t=1}^{T} p(\mathbf{x}_t \mid \mathbf{x}_{<t}), \\ p(\mathbf{x}_t \mid \mathbf{x}_{<t}) &= g_\tau(\mathbf{h}_{t-1}), \end{aligned} \tag{10.2}$$

where g is a function that maps the RNN hidden state $\mathbf{h}_{t-1}$ to a probability distribution over possible outputs, and τ is the parameter set of g.

One of the main factors that determines the representational power of an RNN is the output function g in Eq. (10.2). With a deterministic transition function f, the choice of g effectively defines the family of joint probability distributions $p(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ that can be expressed by the RNN.

We can express the output function g in Eq. (10.2) as being composed of two parts. The first part φ_τ is a function that returns the parameter set ϕ_t given the hidden state $\mathbf{h}_{t-1}$, i.e., $\phi_t = \varphi_\tau(\mathbf{h}_{t-1})$, while the second part of g returns the density of $\mathbf{x}_t$, i.e., $p_{\phi_t}(\mathbf{x}_t \mid \mathbf{x}_{<t})$.

When modelling high-dimensional and real-valued sequences, a reasonable choice of an observation model is a Gaussian mixture model (GMM) as used in (Graves, 2013). For GMM, φ_τ returns a set of mixture coefficients α_t, means $\boldsymbol{\mu}_{\cdot,t}$ and covariances $\Sigma_{\cdot,t}$ of the corresponding mixture components. The probability of $\mathbf{x}_t$ under the mixture distribution is:

$$p_{\boldsymbol{\alpha}_t, \boldsymbol{\mu}_{\cdot,t}, \Sigma_{\cdot,t}}(\mathbf{x}_t \mid \mathbf{x}_{<t}) = \sum_j \alpha_{j,t} \mathcal{N}\left(\mathbf{x}_t; \boldsymbol{\mu}_{j,t}, \Sigma_{j,t}\right).$$

With the notable exception of (Graves, 2013), there has been little work investigating the structured output density model for RNNs with real-valued sequences.

There is potentially a significant issue in the way the RNN models output variability. Given a deterministic transition function, the only source of variability is in the conditional output probability density. This can present problems when modelling sequences that are at once highly variable and highly structured (i.e., with a high signal-to-noise ratio). To effectively model these types of sequences, the RNN must be capable of mapping very small variations in $\mathbf{x}_t$ (i.e., the only source of randomness) to potentially very large variations in the hidden state $\mathbf{h}_t$. Limiting the capacity of the network, as must be done to guard against over-fitting, will force a compromise between the generation of a clean signal and encoding sufficient input variability to capture the high-level variability both within a single observed sequence and across data examples.

The need for highly structured output functions in an RNN has been previously noted. Boulanger-Lewandowski et al. (2012) extensively tested NADE and RBM-based output densities for modelling sequences of binary vector representations of music. Bayer and Osendorfer (2014) introduced a sequence of independent latent variables corresponding to the states of the RNN. Their model, called *STORN*, first generates a sequence of samples $\mathbf{z} = (\mathbf{z}_1, \ldots, \mathbf{z}_T)$ from the sequence of independent latent random variables. At each time step, the transition function f from Eq. (10.1) computes the next hidden state $\mathbf{h}_t$ based on the previous state $\mathbf{h}_{t-1}$, the previous output $\mathbf{x}_{t-1}$ and the sampled latent random variables $\mathbf{z}_t$. They proposed to train this model based on the VAE principle (see Sec. 10.2.2). Similarly, Pachitariu and Sahani (2012) earlier proposed both a sequence of independent latent random variables and a stochastic hidden state for the RNN.

These approaches are closely related to the approach proposed in this paper. However, there is a major difference in how the prior distribution over the latent random variable is modelled. Unlike the aforementioned approaches, our approach makes the prior distribution of the latent random variable at time step t dependent on all the preceding inputs via the RNN hidden state $\mathbf{h}_{t-1}$ (see Eq. (10.5)). The introduction of temporal structure into the prior distribution is expected to improve the representational power of the model, which we empirically observe in the experiments (See Table 10.1). However, it is important to note that any approach based on having stochastic latent state is orthogonal to having a structured output function, and that these two can be used together to form a single model.

10.2.2 Variational Auto-Encoder

For non-sequential data, VAEs (Kingma and Welling, 2013; Rezende et al., 2014) have recently been shown to be an effective modelling paradigm to recover complex multimodal distributions over the data space. A VAE introduces a set of latent random variables $\mathbf{z}$, designed to capture the variations in the observed variables $\mathbf{x}$. As an example of a directed graphical model, the joint distribution is defined as:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x} \mid \mathbf{z})p(\mathbf{z}). \tag{10.3}$$

The prior over the latent random variables, $p(\mathbf{z})$, is generally chosen to be a simple Gaussian distribution and the conditional $p(\mathbf{x} \mid \mathbf{z})$ is an arbitrary observation model whose parameters are computed by a parametric function of $\mathbf{z}$. Importantly, the VAE typically parameterizes $p(\mathbf{x} \mid \mathbf{z})$ with a highly flexible function approximator such as a neural network. While latent random variable models of the form given in Eq. (10.3) are not uncommon, endowing the conditional $p(\mathbf{x} \mid \mathbf{z})$ as a potentially highly non-linear mapping from $\mathbf{z}$ to $\mathbf{x}$ is a rather unique feature of the VAE.

However, introducing a highly non-linear mapping from $\mathbf{z}$ to $\mathbf{x}$ results in intractable inference of the posterior $p(\mathbf{z} \mid \mathbf{x})$. Instead, the VAE uses a variational approximation $q(\mathbf{z} \mid \mathbf{x})$ of the posterior that enables the use of the lower bound:

$$\log p(\mathbf{x}) \geq -\mathrm{KL}(q(\mathbf{z} \mid \mathbf{x}) \| p(\mathbf{z})) + \mathbb{E}_{q(\mathbf{z}|\mathbf{x})} \left[\log p(\mathbf{x} \mid \mathbf{z}) \right], \tag{10.4}$$

where $\mathrm{KL}(Q \| P)$ is Kullback-Leibler divergence between two distributions Q and P.

In (Kingma and Welling, 2013), the approximate posterior $q(\mathbf{z} \mid \mathbf{x})$ is a Gaussian $\mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2))$ whose mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\sigma}^2$ are the output of a highly non-linear function of $\mathbf{x}$, once again typically a neural network.

The generative model $p(\mathbf{x} \mid \mathbf{z})$ and inference model $q(\mathbf{z} \mid \mathbf{x})$ are then trained jointly by maximizing the variational lower bound with respect to their parameters, where the integral with respect to $q(\mathbf{z} \mid \mathbf{x})$ is approximated stochastically. The gradient of this estimate can have a low variance estimate, by reparameterizing $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ and rewriting:

$$\mathbb{E}_{q(\mathbf{z}|\mathbf{x})} \left[\log p(\mathbf{x} \mid \mathbf{z}) \right] = \mathbb{E}_{p(\boldsymbol{\epsilon})} \left[\log p(\mathbf{x} \mid \mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}) \right],$$

where $\boldsymbol{\epsilon}$ is a vector of standard Gaussian variables. The inference model can then be trained through standard backpropagation technique for stochastic gradient descent.

10.3 Variational Recurrent Neural Network

In this section, we introduce a recurrent version of the VAE for the purpose of modelling sequences. Drawing inspiration from simpler dynamic Bayesian networks (DBNs) such as HMMs and Kalman filters, the proposed *variational recurrent neural network* (VRNN) explicitly models the dependencies between latent random variables across subsequent time steps. However, unlike these simpler DBN models, the VRNN retains the flexibility to model highly non-linear dynamics.

Generation The VRNN contains a VAE at every time step. However, these VAEs are conditioned on the state variable $\mathbf{h}_{t-1}$ of an RNN. This addition will help the VAE to take into account the temporal structure of the sequential data. Unlike a standard VAE, the prior on the latent random variable is no longer a standard Gaussian distribution, but follows the distribution:

$$\mathbf{z}_t \sim \mathcal{N}(\boldsymbol{\mu}_{0,t}, \operatorname{diag}(\boldsymbol{\sigma}_{0,t}^2)) \text{ , where } [\boldsymbol{\mu}_{0,t}, \boldsymbol{\sigma}_{0,t}] = \varphi_\tau^{\text{prior}}(\mathbf{h}_{t-1}), \tag{10.5}$$

where $\boldsymbol{\mu}_{0,t}$ and $\boldsymbol{\sigma}_{0,t}$ denote the parameters of the conditional prior distribution. Moreover, the generating distribution will not only be conditioned on $\mathbf{z}_t$ but also on $\mathbf{h}_{t-1}$ such that:

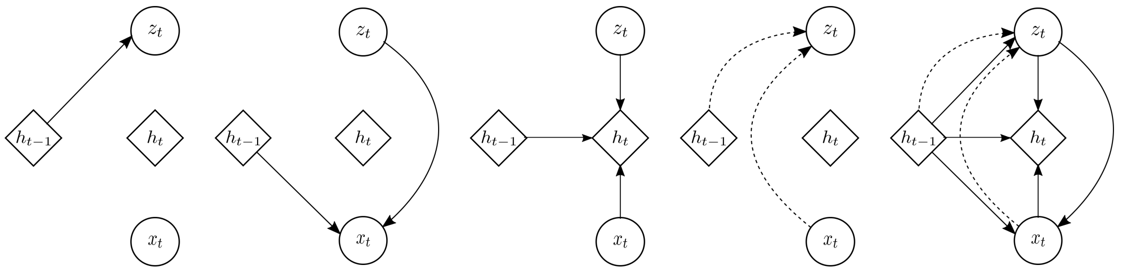
$$\mathbf{x}_t \mid \mathbf{z}_t \sim \mathcal{N}(\boldsymbol{\mu}_{x,t}, \operatorname{diag}(\boldsymbol{\sigma}_{x,t}^2)) \text{ , where } [\boldsymbol{\mu}_{x,t}, \boldsymbol{\sigma}_{x,t}] = \varphi_\tau^{\text{dec}}(\varphi_\tau^{\mathbf{z}}(\mathbf{z}_t), \mathbf{h}_{t-1}), \tag{10.6}$$

where $\boldsymbol{\mu}_{x,t}$ and $\boldsymbol{\sigma}_{x,t}$ denote the parameters of the generating distribution, $\varphi_\tau^{\text{prior}}$ and $\varphi_\tau^{\text{dec}}$ can be any highly flexible function such as neural networks. $\varphi_\tau^{\mathbf{x}}$ and $\varphi_\tau^{\mathbf{z}}$ can also be neural networks, which extract features from $\mathbf{x}_t$ and $\mathbf{z}_t$, respectively. We found that these feature extractors are crucial for learning complex sequences. The RNN updates its hidden state using the recurrence equation:

$$\mathbf{h}_t = f_\theta \left(\varphi_\tau^{\mathbf{x}}(\mathbf{x}_t), \varphi_\tau^{\mathbf{z}}(\mathbf{z}_t), \mathbf{h}_{t-1} \right), \tag{10.7}$$

where f was originally the transition function from Eq. (10.1). From Eq. (10.7), we find that $\mathbf{h}_t$ is a function of $\mathbf{x}_{\leq t}$ and $\mathbf{z}_{\leq t}$. Therefore, Eq. (10.5) and Eq. (10.6) define the distributions $p(\mathbf{z}_t \mid \mathbf{x}_{<t}, \mathbf{z}_{<t})$ and $p(\mathbf{x}_t \mid \mathbf{z}_{\leq t}, \mathbf{x}_{<t})$, respectively. The parameterization of the generative model results in and – was motivated by – the factorization:

$$p(\mathbf{x}_{\leq T}, \mathbf{z}_{\leq T}) = \prod_{t=1}^{T} p(\mathbf{x}_t \mid \mathbf{z}_{\leq t}, \mathbf{x}_{<t}) p(\mathbf{z}_t \mid \mathbf{x}_{<t}, \mathbf{z}_{<t}). \tag{10.8}$$



(a) Prior (b) Generation (c) Recurrence (d) Inference (e) Overall

Figure 10.1: Graphical illustrations of each operation of the VRNN: (a) computing the conditional prior using Eq. (10.5); (b) generating function using Eq. (10.6); (c) updating the RNN hidden state using Eq. (10.7); (d) inference of the approximate posterior using Eq. (10.9); (e) overall computational paths of the VRNN.

Inference In a similar fashion, the approximate posterior will not only be a function of $\mathbf{x}_t$ but also of $\mathbf{h}_{t-1}$ following the equation:

$$\mathbf{z}_t \mid \mathbf{x}_t \sim \mathcal{N}(\boldsymbol{\mu}_{z,t}, \mathrm{diag}(\boldsymbol{\sigma}^2_{z,t})) \text{ , where } [\boldsymbol{\mu}_{z,t}, \boldsymbol{\sigma}_{z,t}] = \varphi^{\mathrm{enc}}_{\tau}(\varphi^{\mathbf{x}}_{\tau}(\mathbf{x}_t), \mathbf{h}_{t-1}), \tag{10.9}$$

similarly $\boldsymbol{\mu}_{z,t}$ and $\boldsymbol{\sigma}_{z,t}$ denote the parameters of the approximate posterior. We note that the encoding of the approximate posterior and the decoding for generation are tied through the RNN hidden state $\mathbf{h}_{t-1}$. We also observe that this conditioning on $\mathbf{h}_{t-1}$ results in the factorization:

$$q(\mathbf{z}_{\leq T} \mid \mathbf{x}_{\leq T}) = \prod_{t=1}^{T} q(\mathbf{z}_t \mid \mathbf{x}_{\leq t}, \mathbf{z}_{<t}). \tag{10.10}$$

Learning The objective function becomes a time step-wise variational lower bound using Eq. (10.8) and Eq. (10.10):

$$\mathbb{E}_{q(\mathbf{z}_{\leq T} \mid \mathbf{x}_{\leq T})}\left[\sum_{t=1}^{T}(-\mathrm{KL}(q(\mathbf{z}_t \mid \mathbf{x}_{\leq t}, \mathbf{z}_{<t}) \| p(\mathbf{z}_t \mid \mathbf{x}_{<t}, \mathbf{z}_{<t})) + \log p(\mathbf{x}_t \mid \mathbf{z}_{\leq t}, \mathbf{x}_{<t}))\right]. \tag{10.11}$$

As in the standard VAE, we learn the generative and inference models jointly by maximizing the variational lower bound with respect to their parameters. The schematic view of the VRNN is shown in Fig. 10.1, operations (a)–(d) correspond to Eqs. (10.5)–(10.7), (10.9), respectively. The VRNN applies the operation (a) when computing the conditional prior (see Eq. (10.5)). If the variant of the VRNN (VRNN-I) does not apply the operation (a), then the prior becomes independent across time steps. STORN (Bayer and Osendorfer, 2014) can be considered as an instance of the VRNN-I model family. In fact, STORN puts further restrictions on the dependency structure of the approximate inference model. We include this version of the model (VRNN-I) in our experimental evaluation in order to directly study the impact of including the temporal dependency structure in the prior (i.e., conditional prior) over the latent random variables.

10.4 Experiment Settings

We evaluate the proposed VRNN model on two tasks: (1) modelling natural speech directly from the raw audio waveforms; (2) modelling handwriting generation.

Speech modelling We train the models to directly model raw audio signals, represented as a sequence of 200-dimensional frames. Each frame corresponds to the real-valued amplitudes of 200 consecutive raw acoustic samples. Note that this is unlike the conventional approach for modelling speech, often used in speech synthesis where models are expressed over representations such as spectral features (see, e.g., Tokuda et al., 2013; Bertrand et al., 2008; Lee et al., 2009).

We evaluate the models on the following four speech datasets:

1. **Blizzard**: This text-to-speech dataset made available by the Blizzard Challenge 2013 contains 300 hours of English, spoken by a single female speaker (King and Karaiskos, 2013).
2. **TIMIT**: This widely used dataset for benchmarking speech recognition systems contains 6,300 English sentences, read by 630 speakers.
3. **Onomatopoeia**[1]: This is a set of 6,738 non-linguistic human-made sounds such as coughing, screaming, laughing and shouting, recorded from 51 voice actors.

1. This dataset has been provided by Ubisoft.

Table 10.1: Average log-likelihood on the test (or validation) set of each task.

Models	Speech modelling				Handwriting
	Blizzard	TIMIT	Onomatopoeia	Accent	IAM-OnDB
RNN-Gauss	3539	-1900	-984	-1293	1016
RNN-GMM	7413	26643	18865	3453	1358
VRNN-I-Gauss	$\geq$ 8933	$\geq$ 28340	$\geq$ 19053	$\geq$ 3843	$\geq$ 1332
	$\approx$ 9188	$\approx$ 29639	$\approx$ 19638	$\approx$ 4180	$\approx$ 1353
VRNN-Gauss	$\geq$ 9223	$\geq$ 28805	$\geq$ 20721	$\geq$ 3952	$\geq$ 1337
	$\approx$ **9516**	$\approx$ **30235**	$\approx$ **21332**	$\approx$ 4223	$\approx$ 1354
VRNN-GMM	$\geq$ 9107	$\geq$ 28982	$\geq$ 20849	$\geq$ 4140	$\geq$ 1384
	$\approx$ 9392	$\approx$ 29604	$\approx$ 21219	$\approx$ **4319**	$\approx$ **1384**

4. **Accent**: This dataset contains English paragraphs read by $2,046$ different native and non-native English speakers (Weinberger, 2015).

For the Blizzard and Accent datasets, we process the data so that each sample duration is $0.5s$ (the sampling frequency used is 16kHz). Except the TIMIT dataset, the rest of the datasets do not have predefined train/test splits. We shuffle and divide the data into train/validation/test splits using a ratio of 0.9/0.05/0.05.

Handwriting generation We let each model learn a sequence of (x, y) coordinates together with binary indicators of pen-up/pen-down, using the IAM-OnDB dataset, which consists of $13,040$ handwritten lines written by 500 writers (Liwicki and Bunke, 2005). We preprocess and split the dataset as done in (Graves, 2013).

Preprocessing and training The only preprocessing used in our experiments is normalizing each sequence using the global mean and standard deviation computed from the entire training set. We train each model with stochastic gradient descent on the negative log-likelihood using the Adam optimizer (Kingma and Ba, 2014), with a learning rate of 0.001 for TIMIT and Accent and 0.0003 for the rest. We use a minibatch size of 128 for Blizzard and Accent and 64 for the rest. The final model was chosen with early-stopping based on the validation performance.

Models We compare the VRNN models with the standard RNN models using two different output functions: a simple Gaussian distribution (Gauss) and a Gaussian mixture model (GMM). For each dataset, we conduct an additional set of experiments for a VRNN model without the conditional prior (VRNN-I).

We fix each model to have a single recurrent hidden layer with 2000 LSTM units (in the case of Blizzard, 4000 and for IAM-OnDB, 1200). All of φ_τ shown in Eqs. (10.5)–(10.7), (10.9) have four hidden layers using rectified linear units (Nair and Hinton, 2010) (for IAM-OnDB, we use a single hidden layer). The standard RNN models only have $\varphi_\tau^{\mathbf{x}}$ and $\varphi_\tau^{\text{dec}}$, while the VRNN models also have $\varphi_\tau^{\mathbf{z}}$, $\varphi_\tau^{\text{enc}}$ and $\varphi_\tau^{\text{prior}}$. For the standard RNN models, $\varphi_\tau^{\mathbf{x}}$ is the feature extractor, and $\varphi_\tau^{\text{dec}}$ is the generating function. For the RNN-GMM and VRNN models, we match the total number of parameters of the deep neural networks (DNNs), $\varphi_\tau^{\mathbf{x},\mathbf{z},\text{enc},\text{dec},\text{prior}}$, as close to the RNN-Gauss model having 600 hidden units for every layer that belongs to either $\varphi_\tau^{\mathbf{x}}$ or $\varphi_\tau^{\text{dec}}$ (we consider 800 hidden units in the case of Blizzard). Note that we use 20 mixture components for models using a GMM as the output function.

For qualitative analysis of speech generation, we train larger models to generate audio sequences. We stack three recurrent hidden layers, each layer contains 3000 LSTM units. Again for the RNN-GMM and VRNN models, we match the total number of parameters of the DNNs to be equal to the RNN-Gauss model having 3200 hidden units for each layer that belongs to either $\varphi_\tau^{\mathbf{x}}$ or $\varphi_\tau^{\text{dec}}$.

10.5 Results and Analysis

We report the average log-likelihood of test examples assigned by each model in Table 10.1. For RNN-Gauss and RNN-GMM, we report the exact log-likelihood, while in the case of VRNNs, we report the variational lower bound (given with $\geq$ sign, see Eq. (10.4)) and approximated marginal log-likelihood (given with $\approx$ sign) based on importance sampling using 40 samples as in (Rezende et al., 2014). In general, higher numbers are better. Our results show that the VRNN models have higher log-likelihood, which support our claim that latent random variables are helpful when modelling complex sequences. The VRNN models perform well even with a unimodal output function (VRNN-Gauss), which is not the case for the standard RNN models.

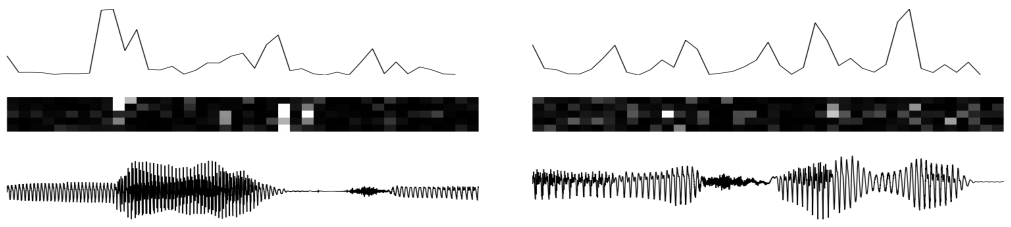
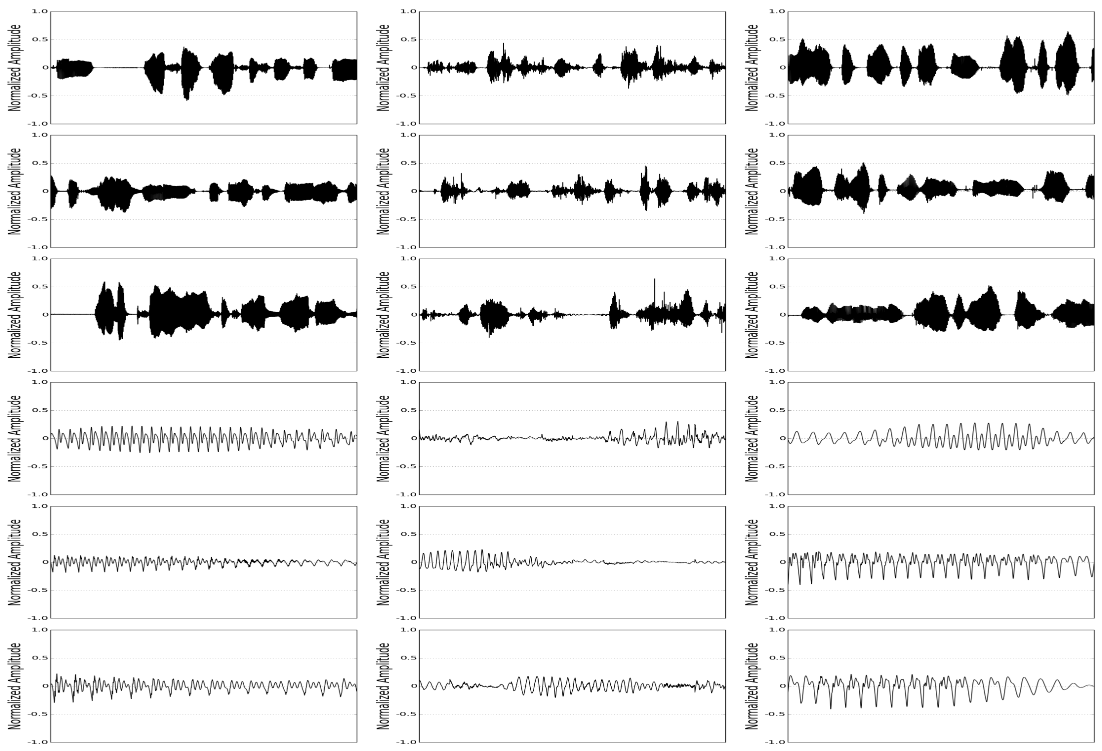


Figure 10.2: The top row represents the difference δ_t between $\boldsymbol{\mu}_{z,t}$ and $\boldsymbol{\mu}_{z,t-1}$. The middle row shows the dominant KL divergence values in temporal order. The bottom row shows the input waveforms.

Latent space analysis In Fig. 10.2, we show an analysis of the latent random variables. We let a VRNN model read some unseen examples and observe the transitions in the latent space. We compute $\delta_t = \sum_j (\boldsymbol{\mu}^j_{z,t} - \boldsymbol{\mu}^j_{z,t-1})^2$ at every time step and plot the results on the top row of Fig. 10.2. The middle row shows the KL divergence computed between the approximate posterior and the conditional prior. When there is a transition in the waveform, the KL divergence tends to grow (white is high), and we can clearly observe a peak in δ_t that can affect the RNN dynamics to change modality.

Speech generation We generate waveforms with $2.0s$ duration from the models that were trained on Blizzard. From Fig. 10.3, we can clearly see that the waveforms from the VRNN-Gauss are much less noisy and have less spurious peaks than those from the RNN-GMM. We suggest that the large amount of noise apparent in the waveforms from the RNN-GMM model is a consequence of the compromise these models must make between representing a clean signal consistent with the training data and encoding sufficient input variability to capture the variations across data examples. The latent random variable models can avoid this compromise by adding variability in the latent space, which can always be mapped to a point close to a relatively clean sample.

Handwriting generation Visual inspection of the generated handwriting (as shown in Fig. 10.4) from the trained models reveals that the VRNN model is able to generate more diverse writing style while maintaining consistency within samples.

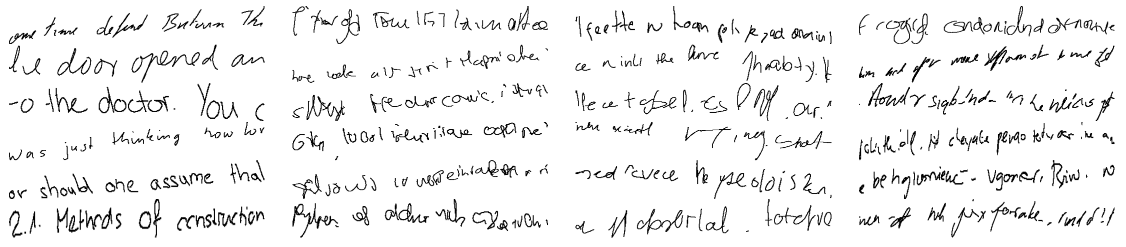


(a) Ground Truth (b) RNN-GMM (c) VRNN-Gauss

Figure 10.3: Examples from the training set and generated samples from RNN-GMM and VRNN-Gauss. Top three rows show the global waveforms while the bottom three rows show more zoomed-in waveforms. Samples from (b) RNN-GMM contain high-frequency noise, and samples from (c) VRNN-Gauss have less noise. We exclude RNN-Gauss, because the samples are almost close to pure noise.

10.6 Conclusion

We propose a novel model that can address sequence modelling problems by incorporating latent random variables into a recurrent neural network (RNN). Our experiments focus on unconditional natural speech generation as well as handwriting generation. We show that the introduction of latent random variables can provide significant improvements in modelling highly structured sequences such as natural speech sequences. We empirically show that the inclusion of randomness into high-level latent space can enable the VRNN to model natural speech sequences with a simple Gaussian distribution as the output function. However, the standard RNN model using the same output function fails to generate reasonable samples. An RNN-based model using more powerful output function such as a GMM can



(a) Ground Truth (b) RNN-Gauss (c) RNN-GMM (d) VRNN-GMM

Figure 10.4: Handwriting samples: (a) training examples and unconditionally generated handwriting from (b) RNN-Gauss, (c) RNN-GMM and (d) VRNN-GMM. The VRNN-GMM retains the writing style from beginning to end while RNN-Gauss and RNN-GMM tend to change the writing style during the generation process. This is possibly because the sequential latent random variables can guide the model to generate each sample with a consistent writing style.

generate much better samples, but they contain a large amount of high-frequency noise compared to the samples generated by the VRNN-based models.

We also show the importance of temporal conditioning of the latent random variables by reporting higher log-likelihood numbers on modelling natural speech sequences. In handwriting generation, the VRNN model is able to model the diversity across examples while maintaining consistent writing style over the course of generation.

11 Conclusion of the Thesis

In this thesis, we have proposed a series of new architectures to explore multiscale RNNs. Performing both temporal and hierarchical abstraction has been a long standing challenge of RNNs, and there is a huge reward for having a robust model that can learn multiscale representation of sequences. In chapter 4, we proposed a framework to build deep RNNs, where connectivity patterns between hidden layers in consecutive time steps are not deterministic, but can be learned by a set of scalar gating units. We learned that top-down information is often helpful when modelling sequences since it conveys more global information or structure of the data. This lesson led us to develop biscale RNNs introduced in chapter 6.

Biscale RNNs were used for neural machine translation as a building block of the character-level decoder. A biscale RNN has a fast layer that updates the hidden state quickly to model the characters and a slow layer that updates the hidden state slower than the fast layer in order to model the words. It is a simplified version of multiscale RNNs that only consists of two timescales. One drawback of biscale RNNs is that boundary detecting units used to open and close the connections between layers are continuous variables in the range $[0, 1]$, instead of having binary states. This results in the leakage of gradients, and the hidden state of the slow layer keeps getting updated at every time step, which is not ideal. Making a binary decision is important in order to introduce useful properties to multiscale RNNs.

In chapter 8, we introduced a more generalized framework to build multiscale RNNs. The architecture itself has not changed much from biscale RNNs, however, now the boundary detecting units are implemented as discrete variables. In order to compute the gradients for the discrete variables, we used the straight-through estimator. With boundary detecting units being discrete and making binary decisions, we can implement a novel update rule that consists of three operations: COPY, UPDATE and FLUSH. The new approach shows empirical evidence that the model is capturing meaningful temporal structure in the data. The proposed model is more resistant against over-fitting, suggesting the multiscale representation learned by the model yields better generalization.

Chapter 10 introduced an inclusion of latent variables to RNNs. This work was a direct extension of the VAE into a recurrent form in order to process sequences. We let the latent variables become dynamic by repeating a VAE at every time step and conditioning it by the hidden state of an RNN. The prior distribution for the latent variables was set to a conditional Gaussian distribution, which is dependent on the hidden state of the RNN. In this work, we showed that injecting noise in a higher-level latent space is more efficient than lower-level space that is close to the data.

We have reviewed three articles that are related to building RNN architectures that can extract hierarchical and decomposable representation of sequences, and one article about introducing latent variables to RNNs that can help to explain the complicated structure of the data. Learning a decomposable representation is important not only for supervised or unsupervised learning of sequences, but also for reinforcement learning. RNNs have an important role to solve partially observable Markov decision process (POMDP) problems in deep reinforcement learning. If RNNs can learn the decomposable and multiscale representation of the input sequence, we can design a policy that plans and draws actions in a hierarchical manner. This approach is essential for solving hierarchical reinforcement learning, where sparse reward is a big issue. The decision search space can be reduced by exploring with higher-level actions from a higher-level policy. By extracting decomposable representation, the higher-level layers of RNNs can be used as the representation for the higher-level policy. Changing the RNN architecture cannot solely overcome the issue with sparse rewards, but this kind of RNN architecture can be an important building block for many problems that require a hierarchical decision making process.

Bibliography

Ackley, D. H., G. E. Hinton, and T. J. Sejnowski (1985). A learning algorithm for boltzmann machines. *Cognitive science 9*(1), 147–169.

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation 10*(2), 251–276.

Amodei, D., S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, et al. (2016). Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pp. 173–182.

Arjovsky, M., A. Shah, and Y. Bengio (2016). Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pp. 1120–1128.

Ba, J. L., J. R. Kiros, and G. E. Hinton (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., K. Cho, and Y. Bengio (2015). Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR).*

Bahdanau, D., J. Chorowski, D. Serdyuk, Y. Bengio, et al. (2016). End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4945–4949. IEEE.

Baldi, P. and K. Hornik (1989). Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks 2*(1), 53–58.

Barron, A. R. (1993). Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory 39*(3), 930–945.

Bayer, J. and C. Osendorfer (2014). Learning stochastic recurrent networks. *arXiv preprint arXiv:1411.7610*.

Bellman, R. (2013). *Dynamic programming.* Courier Corporation.

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning 2*(1), 1–127.

Bengio, Y., R. Ducharme, and P. Vincent (2001). A neural probabilistic language model. In *Advances in Neural Information Processing Systems*, pp. 932–938.

Bengio, Y., P. Lamblin, D. Popovici, H. Larochelle, et al. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems 19*, 153.

Bengio, Y., D.-H. Lee, J. Bornschein, and Z. Lin (2015). Towards biologically plausible deep learning. *arXiv preprint arXiv:1502.04156*.

Bengio, Y., N. Léonard, and A. Courville (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

Bengio, Y., P. Simard, and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks 5*(2), 157–166.

Bertrand, A., K. Demuynck, V. Stouten, et al. (2008). Unsupervised learning of auditory filter banks using non-negative matrix factorisation. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 4713–4716. IEEE.

Booij, G. (2012). *The grammar of words: An introduction to linguistic morphology.* Oxford University Press.

Botha, J. A. and P. Blunsom (2014). Compositional morphology for word representations and language modelling. In *ICML 2014*.

Boulanger-Lewandowski, N., Y. Bengio, and P. Vincent (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*.

Chan, W., N. Jaitly, Q. Le, and O. Vinyals (2016). Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4960–4964. IEEE.

Chan, W., N. Jaitly, Q. V. Le, and O. Vinyals (2015). Listen, attend and spell. *arXiv preprint arXiv:1508.01211*.

Chitnis, R. and J. DeNero (2015). Variable-length word encodings for neural translation models. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 2088–2093.

Cho, K., B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734.

Cho, K., B. Van Merriënboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Cho, K., B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio (2014, October). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*.

Chung, J., S. Ahn, and Y. Bengio (2016). Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*.

Chung, J., K. Cho, and Y. Bengio (2016). A character-level decoder without explicit segmentation for neural machine translation. *Association for Computational Linguistics (ACL)*.

Chung, J., C. Gulcehre, K. Cho, and Y. Bengio (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS Workshop on Deep Learning*.

Chung, J., C. Gulcehre, K. Cho, and Y. Bengio (2015). Gated feedback recurrent neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*.

Cooijmans, T., N. Ballas, C. Laurent, and A. Courville (2016). Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*.

Costa-Jussà, M. R. and J. A. Fonollosa (2016). Character-based neural machine translation. *arXiv preprint arXiv:1603.00810*.

Courbariaux, M., I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*.

Creutz, M. and K. Lagus (2005). *Unsupervised morpheme segmentation and morphology induction from text corpora using Morfessor 1.0.* Helsinki University of Technology.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS) 2*(4), 303–314.

Dahl, G., A.-r. Mohamed, G. E. Hinton, et al. (2010). Phone recognition with the mean-covariance restricted boltzmann machine. In *Advances in neural information processing systems*, pp. 469–477.

Danihelka, I., G. Wayne, B. Uria, N. Kalchbrenner, and A. Graves (2016). Associative long short-term memory. *arXiv preprint arXiv:1602.03032*.

Dauphin, Y., H. de Vries, and Y. Bengio (2015). Equilibrated adaptive learning rates for non-convex optimization. In *Advances in Neural Information Processing Systems*, pp. 1504–1512.

Dauphin, Y. N., R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pp. 2933–2941.

Desjardins, G., K. Simonyan, R. Pascanu, et al. (2015). Natural neural networks. In *Advances in Neural Information Processing Systems*, pp. 2071–2079.

Dinh, L., R. Pascanu, S. Bengio, and Y. Bengio (2017). Sharp minima can generalize for deep nets. *arXiv preprint arXiv:1703.04933*.

Donahue, J., Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell (2014). Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pp. 647–655.

Duan, Y., X. Chen, R. Houthooft, J. Schulman, and P. Abbeel (2016). Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*.

Durrani, N., B. Haddow, P. Koehn, and K. Heafield (2014). Edinburgh's phrase-based machine translation systems for wmt-14. In *Proceedings of the ACL 2014 Ninth Workshop on Statistical Machine Translation, Baltimore, MD, USA*, pp. 97–104.

El Hihi, S. and Y. Bengio (1995). Hierarchical recurrent neural networks for long-term dependencies. In *Advances in Neural Information Processing Systems*, pp. 493–499. Citeseer.

Eldan, R. and O. Shamir (2016). The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pp. 907–940.

Elman, J. L. (1990). Finding structure in time. *Cognitive science 14*(2), 179–211.

Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (2010). Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research 11*(Feb), 625–660.

Fabius, O., J. R. van Amersfoort, and D. P. Kingma (2014). Variational recurrent auto-encoders. *arXiv preprint arXiv:1412.6581*.

Fernández, S., A. Graves, and J. Schmidhuber (2007). Sequence labelling in structured domains with hierarchical recurrent neural networks. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pp. 774–779. Morgan Kaufmann Publishers Inc.

Forcada, M. L. and R. P. Ñeco (1997). Recursive hetero-associative memories for translation. In *International Work-Conference on Artificial Neural Networks*, pp. 453–462. Springer.

Freitag, M., S. Peitz, J. Wuebker, H. Ney, M. Huck, R. Sennrich, N. Durrani, M. Nadejde, P. Williams, P. Koehn, et al. (2014). Eu-bridge mt: Combined machine translation.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics 36*(4), 193–202.

Funahashi, K.-I. (1989). On the approximate realization of continuous mappings by neural networks. *Neural networks 2*(3), 183–192.

Gal, Y. and Z. Ghahramani (2015). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. *arXiv preprint arXiv:1506.02142*.

Gers, F. A., J. Schmidhuber, and F. A. Cummins (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation 12*(10), 2451–2471.

Gillick, D., C. Brunk, O. Vinyals, and A. Subramanya (2015). Multilingual language processing from bytes. *arXiv preprint arXiv:1512.00103*.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, Volume 9, pp. 249–256.

Glorot, X., A. Bordes, and Y. Bengio (2011). Deep sparse rectifier neural networks. In *Aistats*, Volume 15, pp. 275.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep learning.* MIT press.

Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pp. 2672–2680.

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. C. Courville, and Y. Bengio (2013). Maxout networks. *ICML (3) 28*, 1319–1327.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., S. Fernández, and J. Schmidhuber (2007). Multi-dimensional recurrent neural networks. In *Artificial Neural Networks - ICANN 2007, 17th International Conference, Porto, Portugal, September 9-13, 2007, Proceedings, Part I*, pp. 549–558.

Graves, A. and N. Jaitly (2014). Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1764–1772.

Graves, A., M. Liwicki, H. Bunke, J. Schmidhuber, and S. Fernández (2008). Unconstrained on-line handwriting recognition with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 577–584.

Graves, A., A.-R. Mohamed, and G. Hinton (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6645–6649. IEEE.

Graves, A. and J. Schmidhuber (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks 18*(5), 602–610.

Graves, A., G. Wayne, and I. Danihelka (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Grefenstette, E., K. M. Hermann, M. Suleyman, and P. Blunsom (2015). Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1828–1836.

Gregor, K., I. Danihelka, A. Graves, and D. Wierstra (2015). Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*.

Ha, D., A. Dai, and Q. V. Le (2016). Hypernetworks. *arXiv preprint arXiv:1609.09106*.

Haddow, B., M. Huck, A. Birch, N. Bogoychev, and P. Koehn (2015). The edinburgh/jhu phrase-based machine translation systems for wmt 2015. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pp. 126–133.

He, K., X. Zhang, S. Ren, and J. Sun (2015). Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*.

He, K., X. Zhang, S. Ren, and J. Sun (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Heess, N., S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. Eslami, M. Riedmiller, et al. (2017). Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*.

Heess, N., G. Wayne, Y. Tassa, T. Lillicrap, M. Riedmiller, and D. Silver (2016). Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*.

Hermans, M. and B. Schrauwen (2013). Training and analysing deep recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 190–198.

Hinton, G. (2012). Neural networks for machine learning. Coursera, video lectures.

Hinton, G. E., S. Osindero, and Y.-W. Teh (2006). A fast learning algorithm for deep belief nets. *Neural computation 18*(7), 1527–1554.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 6*(02), 107–116.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural computation 9*(8), 1735–1780.

Hoffman, J., E. Tzeng, J. Donahue, Y. Jia, K. Saenko, and T. Darrell (2013). One-shot adaptation of supervised deep convolutional models. *arXiv preprint arXiv:1312.6204*.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences 79*(8), 2554–2558.

Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer feedforward networks are universal approximators. *Neural networks 2*(5), 359–366.

Huang, C. and H. Zhao (2007). Chinese word segmentation: A decade review. *Journal of Chinese Information Processing 21*(3), 8–20.

Huang, G., Y. Sun, Z. Liu, D. Sedra, and K. Weinberger (2016). Deep networks with stochastic depth. *arXiv preprint arXiv:1603.09382*.

Hutter, M. (2012). The human knowledge compression contest.

Ioffe, S. and C. Szegedy (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Jean, S., K. Cho, R. Memisevic, and Y. Bengio (2015). On using very large target vocabulary for neural machine translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics: Short Papers-Volume 2*.

Johnson, M., M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, et al. (2016). Google's multilingual neural machine translation system: Enabling zero-shot translation. *arXiv preprint arXiv:1611.04558*.

Jordan, M. (1986). Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. Technical report, California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science.

Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. *Advances in psychology 121*, 471–495.

Kalchbrenner, N. and P. Blunsom (2013). Recurrent continuous translation models. In *EMNLP*, Volume 3, pp. 413.

Kalchbrenner, N., I. Danihelka, and A. Graves (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kim, Y., Y. Jernite, D. Sontag, and A. M. Rush (2015). Character-aware neural language models. *arXiv preprint arXiv:1508.06615*.

King, S. and V. Karaiskos (2013). The blizzard challenge 2013. In *The Ninth annual Blizzard Challenge.*

Kingma, D. P. and J. Ba (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P. and M. Welling (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Klambauer, G., T. Unterthiner, A. Mayr, and S. Hochreiter (2017). Self-normalizing neural networks. In *Advances in Neural Information Processing Systems*, pp. 972–981.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological cybernetics 43*(1), 59–69.

Kong, L., C. Dyer, and N. A. Smith (2015). Segmental recurrent neural networks. *arXiv preprint arXiv:1511.06018*.

Koutník, J., K. Greff, F. Gomez, and J. Schmidhuber (2014). A clockwork rnn. In *Proceedings of the 31st International Conference on Machine Learning (ICML 2014).*

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1097–1105.

Krueger, D., T. Maharaj, J. Kramár, M. Pezeshki, N. Ballas, N. R. Ke, A. Goyal, Y. Bengio, H. Larochelle, A. Courville, et al. (2016). Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*.

Krueger, D. and R. Memisevic (2015). Regularizing rnns by stabilizing activations. *arXiv preprint arXiv:1511.08400*.

Le, Q. V., N. Jaitly, and G. E. Hinton (2015). A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*.

Le Roux, N. and Y. Bengio (2010). Deep belief networks are compact universal approximators. *Neural computation 22*(8), 2192–2207.

LeCun, Y., Y. Bengio, and G. Hinton (2015). Deep learning. *Nature 521*(7553), 436–444.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation 1*(4), 541–551.

Lee, H., R. Grosse, R. Ranganath, and A. Y. Ng (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pp. 609–616. ACM.

Lee, H., P. Pham, Y. Largman, and A. Y. Ng (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pp. 1096–1104.

Lee, H.-G., J. Lee, J.-S. Kim, and C.-K. Lee (2015). Naver machine translation system for wat 2015. In *Proceedings of the 2nd Workshop on Asian Translation (WAT2015)*, pp. 69–73.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, T., B. G. Horne, P. Tino, and C. L. Giles (1996). Learning long-term dependencies in narx recurrent neural networks. *IEEE Transactions on Neural Networks 7*(6), 1329–1338.

Ling, W., T. Luís, L. Marujo, R. F. Astudillo, S. Amir, C. Dyer, A. W. Black, and I. Trancoso (2015). Finding function in form: Compositional character models for open vocabulary word representation. *arXiv preprint arXiv:1508.02096*.

Ling, W., I. Trancoso, C. Dyer, and A. W. Black (2015). Character-based neural machine translation. *arXiv preprint arXiv:1511.04586*.

Liwicki, M. and H. Bunke (2005). Iam-ondb-an on-line english sentence database acquired from handwritten text on a whiteboard. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pp. 956–961. IEEE.

Luong, M.-T., H. Pham, and C. D. Manning (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Luong, M.-T., I. Sutskever, Q. V. Le, O. Vinyals, and W. Zaremba (2015). Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*.

Luong, T., R. Socher, and C. D. Manning (2013). Better word representations with recursive neural networks for morphology. In *CoNLL*, pp. 104–113.

Maaten, L. v. d. and G. Hinton (2008). Visualizing data using t-sne. *Journal of Machine Learning Research 9*(Nov), 2579–2605.

Mahoney, M. V. (2005). Adaptive weighing of context models for lossless data compression.

Mahoney, M. V. (2009). Large text compression benchmark. *URL: http://www. mattmahoney. net/text/text. html*.

Marcus, M. P., M. A. Marcinkiewicz, and B. Santorini (1993). Building a large annotated corpus of english: The penn treebank. *Computational linguistics 19*(2), 313–330.

Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 735–742.

McCulloch, W. S. and W. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics 5*(4), 115–133.

Mikolov, T. (2012). Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*.

Mikolov, T., M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur (2010). Recurrent neural network based language model. In *INTERSPEECH*, Volume 2, pp. 3.

Mikolov, T., I. Sutskever, A. Deoras, H.-S. Le, S. Kombrink, and J. Cernocky (2012). Subword language modeling with neural networks. *Preprint*.

Mitchell, T. M. (1997). *Machine Learning* (1 ed.). New York, NY, USA: McGraw-Hill, Inc.

Mnih, A. and K. Gregor (2014). Neural variational inference and learning in belief networks. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1791–1799.

Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu (2016). Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. (2015). Human-level control through deep reinforcement learning. *Nature 518*(7540), 529–533.

Mohamed, A.-r., T. N. Sainath, G. Dahl, B. Ramabhadran, G. E. Hinton, and M. A. Picheny (2011). Deep belief networks using discriminative features for phone recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 5060–5063. IEEE.

Montufar, G. F., R. Pascanu, K. Cho, and Y. Bengio (2014). On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pp. 2924–2932.

Mozer, M. C. (1993). Induction of multiscale temporal structure. *Advances in neural information processing systems*, 275–275.

Nair, V. and G. E. Hinton (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814.

Neal, R. M. (1992). Connectionist learning of belief networks. *Artificial intelligence 56*(1), 71–113.

Neubig, G., T. Watanabe, S. Mori, and T. Kawahara (2013). Substring-based machine translation. *Machine translation 27*(2), 139–166.

Oord, A. v. d., S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Pachitariu, M. and M. Sahani (2012). Learning visual motion in recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 1322–1330.

Pachitariu, M. and M. Sahani (2013). Regularization and nonlinearities for neural language models: when are they needed? *arXiv preprint arXiv:1301.5650*.

Pascanu, R. and Y. Bengio (2013). Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*.

Pascanu, R., C. Gulcehre, K. Cho, and Y. Bengio (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

Pascanu, R., T. Mikolov, and Y. Bengio (2012). On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*.

Raiko, T., H. Valpola, and Y. LeCun (2012). Deep learning made easier by linear transformations in perceptrons. In *AISTATS*, Volume 22, pp. 924–932.

Rezende, D. J., S. Mohamed, and D. Wierstra (2014). Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*.

Rifai, S., P. Vincent, X. Muller, X. Glorot, and Y. Bengio (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 833–840.

Rocki, K. M. (2016a). Recurrent memory array structures. *arXiv preprint arXiv:1607.03085*.

Rocki, K. M. (2016b). Surprisal-driven feedback in recurrent networks. *arXiv preprint arXiv:1608.06027*.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review 65*(6), 386.

Rubino, R., T. Pirinen, M. Espla-Gomis, N. Ljubešic, S. Ortiz Rojas, V. Papavassiliou, P. Prokopidis, and A. Toral (2015). Abu-matran at wmt 2015 translation task: Morphological segmentation and web crawling. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pp. 184–191.

Rumelhart, D. E., G. E. Hinton, J. L. McClelland, et al. (1986). A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition 1*, 45–76.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1988). Learning representations by back-propagating errors. *Cognitive modeling 5*(3), 1.

Salakhutdinov, R. and G. E. Hinton (2009). Deep boltzmann machines. In *AISTATS*, Volume 1, pp. 3.

Santos, C. D. and B. Zadrozny (2014). Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1818–1826.

Saxe, A. M., J. L. McClelland, and S. Ganguli (2013). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*.

Schmidhuber, J. (1991). Neural sequence chunkers.

Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation 4*(2), 234–242.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks 61*, 85–117.

Schwenk, H. (2007). Continuous space language models. *Computer Speech & Language 21*(3), 492–518.

Sennrich, R., B. Haddow, and A. Birch (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature 529*(7587), 484–489.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Technical report, DTIC Document.

Sordoni, A., Y. Bengio, H. Vahabi, C. Lioma, J. Grue Simonsen, and J.-Y. Nie (2015). A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 553–562. ACM.

Sotelo, J., S. Mehri, K. Kumar, J. F. Santos, K. Kastner, A. Courville, and Y. Bengio (2017). Char2wav: End-to-end speech synthesis.

Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*(1), 1929–1958.

Srivastava, R. K., K. Greff, and J. Schmidhuber (2015). Training very deep networks. In *Advances in Neural Information Processing Systems*, pp. 2368–2376.

Stollenga, M. F., J. Masci, F. Gomez, and J. Schmidhuber (2014). Deep networks with internal selective attention through feedback connections. In *Advances in Neural Information Processing Systems*, pp. 3545–3553.

Sutskever, I., J. Martens, and G. E. Hinton (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML'11)*, pp. 1017–1024.

Sutskever, I., O. Vinyals, and Q. V. Le (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112.

Tang, Y. and R. R. Salakhutdinov (2013). Learning stochastic feedforward neural networks. In *Advances in Neural Information Processing Systems*, pp. 530–538.

Tieleman, T. and G. Hinton (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning 4*(2).

Tokuda, K., Y. Nankaku, T. Toda, H. Zen, J. Yamagishi, and K. Oura (2013). Speech synthesis based on hidden markov models. *Proceedings of the IEEE 101*(5), 1234–1252.

Tomáš, M. (2012). *Statistical language models based on neural networks.* Ph. D. thesis, PhD thesis, Brno University of Technology.

van den Oord, A., N. Kalchbrenner, and K. Kavukcuoglu (2016). Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*.

Vapnik, V. (2013). *The nature of statistical learning theory.* Springer science & business media.

Vezhnevets, A., V. Mnih, J. Agapiou, S. Osindero, A. Graves, O. Vinyals, K. Kavukcuoglu, et al. (2016). Strategic attentive writer for learning macro-actions. *arXiv preprint arXiv:1606.04695*.

Vilar, D., J.-T. Peter, and H. Ney (2007). Can we translate letters? In *Proceedings of the Second Workshop on Statistical Machine Translation*, pp. 33–39. Association for Computational Linguistics.

Vincent, P., H. Larochelle, Y. Bengio, and P.-A. Manzagol (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103. ACM.

Vincent, P., H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research 11*(Dec), 3371–3408.

Vinyals, O., A. Toshev, S. Bengio, and D. Erhan (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3156–3164.

Viterbi, A. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory 13*(2), 260–269.

Vorontsov, E., C. Trabelsi, S. Kadoury, and C. Pal (2017). On orthogonality and learning recurrent networks with long term dependencies. *arXiv preprint arXiv:1702.00071*.

Wang, Y., R. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio, et al. (2017). Tacotron: Towards end-to-end speech syn. *arXiv preprint arXiv:1703.10135*.

Weinberger, S. (2015). The speech accent archieve. http://accent.gmu.edu/.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural networks 1*(4), 339–356.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE 78*(10), 1550–1560.

Weston, J., S. Chopra, and A. Bordes (2014). Memory networks. *arXiv preprint arXiv:1410.3916*.

Williams, P., R. Sennrich, M. Nadejde, M. Huck, and P. Koehn (2015). Edinburgh's syntax-based systems at wmt 2015. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pp. 199–209.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8*(3-4), 229–256.

Williams, R. J. and J. Peng (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation 2*(4), 490–501.

Wu, Y., M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Wu, Y., S. Zhang, Y. Zhang, Y. Bengio, and R. Salakhutdinov (2016). On multiplicative integration with recurrent neural networks. *arXiv preprint arXiv:1606.06630*.

Xiao, Y. and K. Cho (2016). Efficient character-level document classification by combining convolution and recurrent layers. *arXiv preprint arXiv:1602.00367*.

Xu, K., J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio (2015). Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044 2*(3), 5.

Yao, L., A. Torabi, K. Cho, N. Ballas, C. Pal, H. Larochelle, and A. Courville (2015). Describing videos by exploiting temporal structure. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4507–4515.

Zaremba, W. and I. Sutskever (2014). Learning to execute. *arXiv preprint arXiv:1410.4615*.

Zaremba, W., I. Sutskever, and O. Vinyals (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zeiler, M. D. and R. Fergus (2013). Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*.

Zhang, C., S. Bengio, M. Hardt, B. Recht, and O. Vinyals (2016). Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*.

Zhang, S., Y. Wu, T. Che, Z. Lin, R. Memisevic, R. Salakhutdinov, and Y. Bengio (2016). Architectural complexity measures of recurrent neural networks. *arXiv preprint arXiv:1602.08210*.

Zhang, X., J. Zhao, and Y. LeCun (2015). Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems*, pp. 649–657.

Zilly, J. G., R. K. Srivastava, J. Koutník, and J. Schmidhuber (2016). Recurrent highway networks. *arXiv preprint arXiv:1607.03474*.