Université de Montréal

Designing Regularizers and Architectures for Recurrent Neural Networks

par
David Krueger

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

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Janvier, 2016

© David Krueger, 2016.

RÉSUMÉ

Cette thèse contribue a la recherche vers l'intelligence artificielle en utilisant des méthodes connexionnistes. Les réseaux de neurones récurrents sont un ensemble de modèles séquentiels de plus en plus populaires capable en principe d'apprendre des algorithmes arbitraires. Ces modèles effectuent un apprentissage en profondeur, un type d'apprentissage machine. Sa généralité et son succès empirique en font un sujet intéressant pour la recherche et un outil prometteur pour la création de l'intelligence artificielle plus générale.

Le premier chapitre de cette thèse donne un bref aperçu des sujets de fonds: l'intelligence artificielle, l'apprentissage machine, l'apprentissage en profondeur et les réseaux de neurones récurrents. Les trois chapitres suivants couvrent ces sujets de manière de plus en plus spécifiques. Enfin, nous présentons quelques contributions apportées aux réseaux de neurones récurrents.

Le chapitre 5 présente nos travaux de régularisation des réseaux de neurones récurrents. La régularisation vise à améliorer la capacité de généralisation du modèle, et joue un role clé dans la performance de plusieurs applications des réseaux de neurones récurrents, en particulier en reconnaissance vocale. Notre approche donne l'état de l'art sur TIMIT, un benchmark standard pour cette tâche.

Le chapitre 6 présente une seconde ligne de travail, toujours en cours, qui explore une nouvelle architecture pour les réseaux de neurones récurrents. Les réseaux de neurones récurrents maintiennent un état caché qui représente leurs observations antérieures. L'idée de ce travail est de coder certaines dynamiques abstraites dans l'état caché, donnant au réseau une manière naturelle d'encoder des tendances cohérentes de l'état de son environnement. Notre travail est fondé sur un modèle existant; nous décrivons ce travail et nos contributions avec notamment une expérience préliminaire.

Mots clés: réseau de neurones, apprentissage machine, apprentissage profond, régularisation, intelligence artificielle, apprentissage supervisé, apprentissage non supervisé, reconnaissance vocale, modélisation du langage

ABSTRACT

This thesis represents incremental work towards artificial intelligence using connectionist methods. Recurrent neural networks are a set of increasingly popular sequential models capable in principle of learning arbitrary algorithms. These models perform deep learning, a type of machine learning. Their generality and empirical success makes them an attractive candidate for further work and a promising tool for creating more general artificial intelligence.

The first chapter of this thesis gives a brief overview of the background topics: artificial intelligence, machine learning, deep learning, and recurrent neural nets. The next three chapters cover these topics in order of increasing specificity. Finally, we contribute some general methods for recurrent neural networks.

Chapter 5 presents our work on the topic of recurrent neural network regularization. Regularization aims to improve a model's generalization ability, and is a key bottleneck in the performance for several applications of recurrent neural networks, most notably speech recognition. Our approach gives state of the art results on the standard TIMIT benchmark for this task.

Chapter 6 presents the second line of work, still in progress, exploring a new architecture for recurrent neural nets. Recurrent neural networks maintain a hidden state which represents their previous observations. The idea of this work is to encode some abstract dynamics in the hidden state, giving the network a natural way to encode consistent or slow-changing trends in the state of its environment. Our work builds on a previously developed model; we describe this previous work and our contributions, including a preliminary experiment.

Keywords: neural networks, machine learning, deep learning, regularization, artificial intelligence, supervised learning, unsupervised learning, speech recognition, language modeling.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AE	Autoencoder
AGI	Artificial General Intelligence
AI	Artificial Intelligence
ANN	Artificial Neural Network
BLEU	Bilingual Evaluation Understudy
BN	Batch Normalization
CAD	Computer Aided Design
CPGP	Conditional Predictive Gating Pyramid
CTC	Connectionist Temporal Classification
DL	Deep Learning
DNN	Deep Neural Network
FAI	Friendly Artificial Intelligence
FGAE	Factored Gated Autoencoder
FNN	Feedforward Neural Network
GAE	Gated Autoencoder
GOFAI	Good Old-Fashioned Artificial Intelligence GPU

Graphics Processing Unit

GRU	Gated Recurrent Unit
IRNN	Identity Recurrent Neural Network
IRL	Inverse Reinforcement Learning
LSTM	Long Short-Term Memory
LM	Language Model
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
NaN	Not a Number
NLP	Natural Language Processing
NTM	Neural Turing Machine
PGP	Predictive Gating Pyramid
RL	Reinforcement Learning
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SL	Supervised Learning
SRNN	Simple Recurrent Neural Network
SVM	Support Vector Machine
TRec	Threshold Rectified Linear Unit
UL	Unsupervised Learning

ACKNOWLEDGMENTS

I would like to thank everyone who helped me onto the path that lead me to complete this thesis, as well as those who made the journey more interesting, enjoyable, and/or manageable. In reverse chronological order, I gratefully thank: My sister Gretchen for proofreading. Alexandre de Brébisson for a great deal of help with the French text of the Résumé. Celine Begin for helping me navigate the administrative procedures of the university and the thesis process. My supervisors Yoshua Bengio and Roland Memisevic for creating a collaborative and open research atmosphere in the lab and for funding my studies and giving me the freedom to pursue interesting topics. Roland in particular for frequent impromptu discussions and guidance on how to make things actually work. Many amazing members of the LISA/MILA lab, including professors, visitors, and staff, for many fascinating and/or entertaining discussions, and invaluable help, advice, encouragement, and good times. The members of the RLLAB at McGill for the same and for welcoming me to their lab for talks and reading groups, or just to hang out and talk. The entire AI safety community for their work on this critical problem, and for helping bring existential risk into mainstream discourse in the machine learning community and society at large. Jelena Luketina in particular for inviting me to the 2015 CFAR reunion. All of my coauthors for the chance to work with and learn from them. Laurent Dinh in particular for patiently teaching me quite a bit about machine learning in the earlier stages of my degree. Daphne Koller and Andrew Ng for creating Coursera, and Geoff Hinton for his Coursera course, without which I almost certainly would not have heard about the phenomenon of Deep Learning in time to start this program when I did. My Freshman year housing advisor at Reed, Hassan Ghani, for inspiring me to seriously consider the possibility that artificial intelligence could be developed and the ethical significance of such an occurrence. All of the wonderful teachers I've had over the years who inspired me to keep studying. Finally, my family and friends for everything they've done for me over the years.

CHAPTER 1

INTRODUCTION

The subject matter of this thesis is artificial intelligence (AI), specifically machine learning (ML). More specifically, I focus on deep learning (DL) and in particular recurrent neural networks (RNNs). I'll introduce these topics briefly before delving into more detail in the following chapters, which provide the necessary background for the rest of the thesis, as well as a brief overview of these topics.

I'll attempt to describe the current state of AI, ML, DL, and RNN research, and how they relate to one another, making an effort to be informative as to current practices and recent trends. My goal is to give a glimpse of the world of artificial intelligence from the perspective of a deep learning researcher at the beginning of 2016, a time when artificial intelligence and deep learning are making headlines and capturing people's attention and imagination to an unprecedented extent. As such, broad statements about trends in these fields should be taken as my personal impressions which I feel confident enough to state as such. I hope that some degree of subjectivity can be accepted as a price for this insider's perspective, since I imagine it could be both interesting and informative to someone unfamiliar with the field.

I mostly use the "companionate we" for the rest of the thesis, occasionally slipping back into first-person singular for opinionative remarks. We assume some familiarity with calculus, linear algebra, probability, statistics, computer science, and graph theory. We attempt to be quite general in our definitions, but some of our terminology may be used in a more (or less) restrictive way in practice, depending on context.

1.1 Artificial Intelligence (AI)

We define artificial intelligence as the study of intelligent behaviour [1]. We distinguish ***narrow AI*** from ***artificial general intelligence (AGI)***. Narrow AI is about solving specific ***tasks*** which are considered indicative of intelligence. AGI is about solving *intelligence*. We will discuss what intelligence is in the next chapter, but to give some intuition: an entity which reaches or surpasses human performance in every task might be said to demonstrate AGI.

For some researchers, the goal of AI is to understand the common principles underlying all intelligent behaviour. For others, the goal is the creation of intelligent agents, and understanding the principles of intelligence is unnecessary, unimportant, or impossible. Historically, there has been a categorization of AI researchers into "scruffies", who view intelligence as a collection of (possibly problem specific) hacks, [2] and "neats", who believe in elegant and principled approaches. Similarly, I consider both engineering (scruffy) and science (neat) to be important components of AI research, and also juxtapose empirical (scruffy) and theoretical (neat) aspects of the study of intelligence. While Russell and Norvig reference the victory of the neats (since 1987) [56], I view the field as progressing via a continuous interaction of engineering, experiment, and theory. The importance of efficient hardware and software implementations makes engineering a constant influence on which methods are adapted and developed. The inventiveness and intuition of experimenters continues to produce unexpected breakthroughs resulting in new peaks in performance and efficiency. And theory continues to inform every researcher's intuition and practice.

[1] AI is generally thought of as trying to create *real* (not simulated) intelligence; In this spirit, Haugeland [22] proposed renaming it "synthetic intelligence". I prefer to think of it as a search for principles underlying intelligent behaviour (and/or its creation).

[2] A similar idea in neuroscience is the "massive modularity" hypothesis, which views the brain as a diverse collection of specialized modules rather than a general purpose learning machine.

1.2 Machine Learning (ML)

Machine learning is the dominant paradigm of modern artificial intelligence research. Although the field has often been more focused on narrow AI, interest in AGI has been increasing recently. The intuition of machine learning, from an AI perspective, is that intelligent behaviour can be learned from data. The AI designer becomes responsible for choosing the rules which govern the learning process, rather than directly specifying the rules that govern behaviours.

ML applies statistical tools to data to solve tasks, rather than to understand the data. ML also views its tools more algorithmically, and is interested in what is possible given a budget of computational resources, whereas statisticians are focused on what is possible in principle given a budget of statistical resources (i.e. data).

A type of ML that is particularly relevant to AI is ***Reinforcement Learning (RL)***. It has been argued [39, 62] that RL is the correct framework for AGI. RL formulates the problem of an ***agent*** interacting with an ***environment*** and seeking to maximize lifetime utility, or ***reward***. In RL, learning is, by definition, goal-directed, and the AI designer becomes responsible for specifying the goals of the AI via a ***reward function***.

While using ML may allow us to program an algorithm that generates desirable behaviour, it does not always give us a satisfactory understanding of how this behaviour is generated. Indeed, the algorithms learned by ML are usually ***opaque***, meaning that they defy human understanding to some extent. Machine learning researchers often aim to understand the algorithms which govern the learning process instead, but even at this higher-level, deep understanding remains elusive for many popular techniques, including much of deep learning.

1.3 Deep Learning (DL)

Deep learning is now the dominant paradigm in many areas of machine learning. Deep learning is perhaps best thought of as a trend in the machine learning community towards renewed interest in artificial neural networks (ANNs). Artificial neural networks (often simply called "neural networks" or "neural nets") are simple models that can learn complicated functions, and are loosely inspired by the networks of actual neurons and synapses that exist in the brain.

Most neural nets can be represented by a directed graph, called the ***computational graph*** [3]. Each node or ***unit*** in this graph represents a neuron, and is computed as a function of its parents. The roots and leaves are called input and output units, respectively, and all the rest are ***hidden units***. The hidden units are organized into ***(hidden) layers*** composed of units with the same distance from the roots [4]. ANNs with multiple hidden layers are called ***deep neural nets (DNNs)***.

Some theory and intuitions exist motivating DL methods [5], for instance, they are known to be universal function approximators [27]. Their recent popularity, however, can largely be attributed to their empirical triumphs. Despite the success of a few fairly simple models and learning principles across a wide range of tasks, to me, DL has a distinctly "scruffy" feel, and I've sometimes whinged that the field is too focused on engineering as opposed to science. Researchers from other areas of ML may find DL unprincipled, but DL researchers may retort that DL is necessary to make things work, and so cannot be abandoned in favor of methods that have nicer theoretical properties, but fail to perform (rather theoreticians should focus on understanding the methods that work instead of those that are easy to prove things about). Moreover, theories based on incorrect simplifying assumptions may give only the illusion of understanding. For example, much theory assumes that the model has been specified correctly (i.e. has the correct functional form); in practice this can assume an unrealistic amount of prior knowledge.

A recent trend in ML is combining deep learning and reinforcement learning (producing ***deep reinforcement learning***). ANNs have a long history of being used to approximate functions of interest in RL (e.g. value function, policy, or model), going back to Tesauro [65]. Many researchers at the forefront of this work today are cautiously optimistic that deep RL can be used to create AGI (i.e. that no further *conceptual* break-

[3] Some neural nets, such as Boltzmann machines, contain undirected edges; we do not consider them in this thesis.

[4] We define this distance as the maximum distance across all roots, although typically they would be equal.

throughs are required).

1.4 Recurrent Neural Networks (RNNs)

Recurrent Neural Nets differ from other ANNs (called ***feedforward nets (FNNs)***) in that their computational graph contains cycles, the edges of which are called ***recurrent connections***. RNNs are a natural model for sequential data. A single RNN can process multiple sequences of any length, and the lengths of the sequences can be different. Recently, RNNs have shown great promise in natural language processing (NLP) and speech recognition applications. Recurrent Neural Networks are more powerful than feedforward nets [59], capable, in fact, of simulating a universal Turing machine [61].

Recurrent Neural Networks are especially suited for real-world RL tasks, because their behaviour is a function of all of their previous inputs (or ***observations*** in RL terminology). This is important because in general the t-th element of an input sequence does not contain a complete representation of the state of the world at time t. For instance, a camera on the front of a robot will not detect something behind it. But just like a human, an RNN could remember what it had seen, then turn around and gather more information about its environment.

CHAPTER 2

MACHINE LEARNING

Modern machine learning differs from classical AI (sometimes called "Good Old-Fashioned AI (GOFAI)") in two big ways. One that we've already mentioned is learning instead of directly programming behaviours. The other is by focusing on probability instead of logic [1].

Machine Learning has been successful in many areas, such as Computer Vision, where GOFAI has failed. We postulate several reasons for this:

- The desired behaviours are too complicated for humans to code directly.
- The details of how the behaviours are generated are beyond humans' (conscious) knowledge.
- The knowledge involved is inherently probabilistic or fuzzy, and not well represented by dichotomies between true and false.
- There is too much relevant knowledge for it to be represented as logical statements and rules of inference/deduction.

Nonetheless, it is quite possible that logic and symbolic representations have a large role to play in AI; after all, they play a significant role in many parts of human intelligence.

2.1 A simple example

Much of machine learning can be thought of as designing algorithms that use data to make predictions. For instance, we might want to know what the value of the S&P500

[1] This is closely related to ***representation***, a topic we return to in section 3.2. Classical approaches represented concepts using specific symbols, such as words, properties, and relationships. Modern machine learning uses a ***bottom-up*** approach, where concepts are represented in terms of the data; for instance neural networks represent concepts via patterns of activity distributed across nodes in a graph. I think this is closer to how the brain operates, and we do not have so-called "grandmother cells", corresponding uniquely to specific concepts.

(a stock market index) will be at some point in the future. A human making such a prediction might consider all sorts of different sources of information:

- the current value of the S&P500
- past values of the S&P500
- indicators of the state of financial markets, such as stock charts and indexes
- media articles about the state of financial markets
- their belief about trends in the global economy

Similarly, an algorithm could attempt to use this kind of data when trying to predict the future value of the S&P500. For instance, the algorithm can use a formula to make its predictions; specifically, we could predict the value of the S&P500 at some future time t seconds in the future, based on t and its the current price, p_{now}.

$$prediction = p_{now} + \theta t$$

Where θ is a real number, to be specified. We could then try to find the best values of θ by looking at what values do the best job of predicting past data.

The form of this equation and the choice of which data is used to make the prediction reflect our ***prior*** assumptions about the way the world operates. This equation essentially says that the value of the S&P500 will increase or decrease steadily (linearly) with time, and all the other factors we've mentioned have no use in determining the future value. This is probably not the case, and such a simple equation might not be very useful for active investing, but it could perhaps be useful, e.g. for planning for retirement.

Making some prior assumptions is essential to making machine learning problems tractable, and having a good prior is essential to getting useful results. Imagine trying to predict changes in the stock market based on the outcome of the super-bowl; such a strategy seems patently absurd [2]!

[2] ...and yet a striking correlation exists which is famous enough to have its own wikipedia page: https://en.wikipedia.org/wiki/Super_Bowl_indicator

2.2 What is machine learning?

Machine learning algorithms aim to discover generalizable patterns in data. Formally, we consider data living in some space $\mathscr{X}$, and devise a machine learning algorithm to produce some function $f : \mathscr{X} \to \mathscr{Y}$. We can view the algorithm as searching for an appropriate function f within some space of functions $\mathscr{F}$. This search process is called ***learning*** or ***training***, and algorithms which perform such a search are ***learning algorithms***.

We call $\mathscr{F}$ the ***hypothesis class***, as it represents the set of allowed hypotheses about what patterns exist in the data. The space of functions considered is shaped and often further restricted by a ***model***. In the process of specifying a model, prior assumptions about which types of functions yield good solutions are formalized mathematically, giving the model an ***inductive bias***. The inductive bias tells a model how to generalize to unseen data; without some prior assumptions, the problem of learning is ill-posed, since the value of the function at some unobserved point could be anything in its range. A model may give good results because the hypotheses it encodes are correct, but good results may also be due to other reasons, such as making good use of the data and computational resources available. For instance, a model of a weighted coin toss could involve the force applied by the tosser, which is certainly one of the factors determining the result. But fewer tosses and less computation would be required to learn a simpler model which imagines that the coin has a fixed probability of being heads. Inductive biases can reflect specific knowledge of the task at hand, called ***domain knowledge***; but they can also reflect more generic ideas about the nature of the universe or of learning. The most common inductive bias in ML is ***locality***, which states that similar inputs should produce similar outputs.

2.2.1 The data generating process

In many cases, learning is performed on a finite ***dataset*** $\mathscr{D}$ of ***data points***. Models typically make some basic assumptions about the ***data generating process***, that is, the real-world process which generates the data that the algorithm observes. A common assumption is that the data being modeled is independently and identically distributed (i.i.d.). That is, we assume that each data point is independently sampled from the same ***data distribution***.

This assumption allows us to think of our data as multiple objects inhabiting a space of data points, rather than a single object inhabiting a space of datasets. Successive screenshots from a video game are typically highly correlated and hence not i.i.d. But if we consider the entire video of the screen during a complete game to be one data point, then these data points might be i.i.d. (although not if the player was changing their strategy between games).

The i.i.d. assumption can never be guaranteed to hold in the real world, where data is sampled from a partially observed environment. There could always be some part of the data generating process that is unknown to us; for instance, a video game's programmers might have the game change in a predictable way each time it is played. More generally, there might be very complicated relations of cause and effect that are currently unknown to us, and thus appear as noise in our dataset, but which could in principle be modeled.

2.2.2 Parametric and nonparametric ML

A common way of restricting the space of functions considered is to specify a functional form with some free variables, θ, called parameters, and search for parameter values that yield a good function. This ***parametric*** approach contrasts with ***nonparametric*** approaches, which adjust the number of parameters throughout learning in a data-dependent way. In addition to parameters, models often have ***hyperparameters***, which are free variables that are not learned, but rather specified in advance by the model designer. Choosing hyperparameters can be thought of as part of the process of specifying the model. An example hyperparameter is the number of units in the first hidden layer of an MLP (see section 3.1.2 and figure 3.2 for MLPs).

2.2.3 Supervised and unsupervised learning

Machine learning is typically broken down into three subfields: ***supervised learning (SL)***, ***unsupervised learning (UL)***, and ***reinforcement learning (RL)***, although this categorization is imperfect. Ignoring reinforcement learning for the time being, there are at least two ways of distinguishing supervised and unsupervised learning: in terms of the kind of function learned, or in terms of the data used.

In terms of the function learned, (un)supervised learning can be thought of as learning something about an (un)conditional probability distribution (see Table 2.I) [3]. In supervised learning, the conditional probability distribution can be used to make ***predictions*** about the value of Y for any particular value of X, e.g. by sampling from the conditional distribution, or taking the argmax [4].

In terms of data, (un)supervised learning uses (un)labeled data. A data point x is unlabeled by default, and becomes labeled when a (typically human) labeler assigns a ***label*** y to it, creating a labeled pair $\{x, y\}$.

Table 2.I: Different ways of defining supervised vs. unsupervised learning.

defined by	supervised learning (SL)	unsupervised learning (UL)
function learned	$P(Y\|X)$	$P(X)$
data used	labeled	unlabeled

The uses are related: generally with labeled data, the ***target*** (i.e. what we try to predict) is the label that the labeler assigned. But we can also simply treat the label as part of the data and perform unsupervised learning on the labeled pairs. Similarly, with unlabeled multi-dimensional data, we can treat some subset of the dimensions as

[3] For convenience, we allow ***generalized functions***, such as the ***Dirac delta function*** in our definition of a probability distribution.

[4] The argmax is the argument of a function which produces a maximum valued output:

$$\arg\max(f) = x : f(x) \geq f(x')\,\forall x' \in X$$

(we can break ties by ordering the inputs and using this and the original ordering to induce a new strict ordering on f's outputs). In the case of a finite vector, the argmax is usually taken to be the first index containing its maximum value.

targets and learn their conditional distribution given the values of the other dimensions. For example, given a dataset of unlabeled images, we might use the pixel values in the bottom half of each image as a target, and try to predict them from the pixels in the top half of the same image. In this thesis, we use the terms to refer to the type of function learned, unless otherwise specified.

2.2.4 Generative models

A ***generative model*** can be used to produce samples from a learned probability distribution, often a non-trivial or even intractable task. Conventionally, generative modeling refers to UL, and "conditional generative modeling" to SL.

Supervised learning is sometimes distinguished from ***conditional generative modeling***, although as we've defined supervised learning, it covers both definitions. The distinction is about whether the target is a value $y \in Y$ (supervised learning) , or a distribution $P(Y)$ (conditional generative modeling).

As a simple example, consider a labeled dataset with 10 identical data points, 3 of which are labeled 0, and 7 of which are labeled 1. An idealized supervised learner aiming to minimize errors would learn to output 1 for this example, whereas an idealized conditional generative model would learn a Bernoulli distribution with parameter $p = .7$.

A conditional generative model might be more appropriate in many cases; for instance in the task of machine translation, there may exist several good translations of a given document. While even then we might be satisfied with a model that gives us one very good translation, there are applications where we are interested in producing multiple outputs for each input. For instance, we might like a model that can generate many songs in the style of a given artist.

2.2.5 Cost functions

We typically use some scalar ***cost function***, $\mathscr{L} : \mathscr{F} \to \mathbb{R}$ to evaluate candidate functions, aiming to find functions with progressively lower ***cost*** as learning progresses. The cost function should reflect the ability of the learned function to perform some ***task*** of interest, We generally think of the cost as a known function that can be computed.

2.2.5.1 Choice of cost function

It can be extremely difficult to devise a cost function that does a good job of capturing all the relevant aspects of a task. Using translation as an example, the best way of evaluating the quality of translation is to let human experts specify the cost for each French/English pair of documents. But the function which human experts implement is unknown, and it is impractical to have humans evaluating a function's outputs in real-time. In practice, a cost for this task is typically computed using the Bilingual Evaluation Understudy (BLEU) score, a function which compares the algorithm's translation to some previously completed expert translations. There are known issues with using BLEU as a cost function, but it is widely used nonetheless, because it is easy to compute from a dataset of French and English versions of the same documents.

2.2.5.2 Cost functions for AGI

An unconstrained search for an algorithm with low cost can return unexpected solutions, which may have undesirable behaviours, if the cost does not properly reflect everything that is or isn't considered desirable [5]. As AI becomes more powerful, the choice of cost becomes increasingly important, since more extreme solutions can be found. For instance, Bostrom [8] argues that obvious choices of cost for achieving seemingly innocuous goals like optimizing paperclip production could lead to disastrous outcomes in which an advanced AI transforms all available resources into paperclip manufacturing facilities, including all of the earth and humanity. More generally, ***instrumental rationality*** refers to the tendency of an advanced AI to develop certain ***instrumental goals***, such as self-preservation, rationality, and resource acquisition, regardless of the ***terminal goals*** encoded in its cost function. An AGI which is much more intelligent than any

[5] The classic example here is Bird and Layzell [7], in which an evolutionary algorithm designed to create an oscillator in a computer chip instead created a radio and amplified radio signals carrying oscillations. The authors of [7] concluded by highlighting the practical impossibility of simulating such a result.

human could plausibly find ways of accomplishing its goals in the face of human opposition, thus it should have a cost function which not only captures the important aspect of a single task, but rather all the complexities of human values. The problem of finding a cost function that reflects human values has been termed ***Friendly AI (FAI)*** [77]. Different humans have very different values, of course; ***coherent extrapolated volition*** [76] is one idea about how to resolve this issue. While I consider these issues critically important to the field of AI and humanity in general, they are out of the scope of this thesis, and rarely considered in current practice. In this work, we pursue the standard research paradigm of minimizing the cost, without considering whether it is an appropriate measure of performance for real world applications.

2.2.5.3 Inverse reinforcement learning

A cost function encoding human values could potentially be learned from data such as human behaviour and cultural artifacts. One proposed way to learn human values is inverse reinforcement learning (IRL) [55**?**] [6]. Inverse reinforcement learning algorithms observe some behaviour and seek to uncover the reward function motivating that behaviour. IRL could be used to learn about human values by observing human behaviour. There are many outstanding problems in IRL which would be relevant to such a project.

2.2.6 Types of SL: classification, regression, and structured output

2.2.6.1 Classification

A classic example of a classification task is to correctly label pictures of different animals with the name of the animal. In classification problems, the targets express which of a set of categories (e.g. "dog", "cat", "pig", ...) the inputs belong to. The number of categories, k, is typically finite and can thus be identified with the integers $\{1, 2, ..., k\}$. Another common way to represent categories in a classification problem is called the

[6] I've imprecisely conflated ***cost*** with ***reward*** in this discussion; while there is a significant difference between these concepts, a reward function can be translated into a cost function, given a perfect model of the world.

one-hot encoding. Here, we represent each category by a different k-dimensional binary vector with a 1 in one dimension and 0s in all the others (see table 2.II).

Table 2.II: Encoding categories as mathematical objects

categories	integer encoding	one-hot encoding
"cat"	1	$\{1,0,0,...\}$
"dog"	2	$\{0,1,0,...\}$
"pig"	3	$\{0,0,1,...\}$
...	...	...

A common cost function for classification problems is called ***categorical cross-entropy***. This is the cross entropy between the target and the prediction, considered as categorical (a.k.a "multinoulli") distributions. Cross entropy is defined as:

$$H(p,q) = -\int p(x)\log q(x)dx$$

which in the categorical case simply becomes $-\log q(t)$ where t is the target category; this is also called the negative log-likelihood [7]. The special case in which there are two categories is called ***binary classification***.

2.2.6.2 Regresssion

In ***regression*** tasks, the targets are real-valued vectors (typically finite-dimensional). The example of predicting the value of the S&P500, from the beginning of this chapter, is a regression problem. The example of predicting the pixel values of the top half of an image given the bottom half is as well. A common cost function for regression is the ***mean squared error (MSE)***, given by the (average) squared distance between each prediction and its corresponding target:

$$MSE(X,Y) = \mathbb{E}(f(x)-y)^2$$

[7] A ***likelihood*** is simply a probability distribution at some point in its domain, considered as a function of the parameters of the distribution.

2.2.6.3 Structured output

Structured output refers to problems where the space of targets is structured (e.g. not categorical or scalar real-values). Predicting the pixel values of the top half of an image given the bottom half is a structured output problem, since the values of nearby pixels in images are correlated. Predicting multiple categorical variables is also a structured output problem (for instance, predicting which animal is in a picture, and which direction it is facing). Predicting a graph, e.g. a parse tree is another example.

2.2.7 Examples of UL: density estimation, clustering

2.2.7.1 Density estimation

Density Estimation involves learning a function that maps inputs to their probability mass or density. Such a function can be used, e.g. in structured prediction to "clean up" the outputs of a supervised learning model. For instance, in translation, if we have a small number of documents translated from French to English (labeled data), but a much larger number of documents in English (unlabeled data), we can learn a translation model using SL on the labeled data, and learn a ***language model*** to perform density estimation on the unlabeled data. The language model simply tells us how probable each English sentence is, so we can use this to select translations of a given French sentence that make more sense in English. Concretely, we might use the translation model to generate a list of candidate translations and then choose the one which is the most probable English sentence.

2.2.7.2 Clustering

A standard example of unsupervised learning is ***clustering***. An exception to our definition of unsupervised learning above, Clustering methods don't necessarily learn a probability distribution. Rather, they learn a function that assigns data points to corresponding ***clusters***. Clusters can be thought of as an equivalence relation over data points; data points which are assigned to the same cluster are meant to be similar in some sense.

For instance, customers of some company might be clustered together based on their consumer habits and advertisements designed for each cluster.

One popular method for clustering is called ***k-means clustering***. In ***k-means clustering***, we decide in advance the number of clusters, k, and assign them each some point in data space. Then we assign data points to the nearest cluster, then move each cluster to the mean of the points assigned to it. We repeat this process until the assignments and means stop changing, which is guaranteed to happen after a finite number of iterations.

2.3 Optimization (minimizing cost)

We've defined learning as a search process over a space of functions. We call the process of searching for functions that minimize the cost function ***optimization***, and a technique for performing optimization an ***optimizer*** [8].

We view this search process as constructing sequence $F = f_1, f_2, \ldots$ of candidate functions, from which we take the element with minimum cost.

When the space $\mathscr{F}$ is finite, it is possible to search using brute-force; this is also called ***exhaustive search***. When $\mathscr{F}$ is infinite, a simple strategy would be to place a probability distribution on $\mathscr{F}$, and then sample functions from it; this is called ***random search***.

Random search and exhaustive search are quite limited. Better search methods are the main thing that makes machine learning techniques seem intelligent. For example, the game of Go is the best-known classic game in which AI has not achieved super-human performance [9], and has received a large amount of attention among AI researchers. Because there is a maximum number of turns in a game of Go, and a finite number of possible moves per game, there are a finite number of possible games, and a finite number of strategies. Thus the best strategy could be found via exhaustive search. However, the number of strategies is also significantly larger than the number of parti-

[8] This is inspired by the idea of finding a local or global optimum (in this case a minimum) of the cost function, although in practice we are simply trying to improve upon the best function found so far, and don't necessarily care if we have reached an optimum.

[9] Although recently, **?**] demonstrated significant progress, and super-human performance appears immanent.

cles in the known universe, so this exhaustive search could not be performed in practice using current technology.

A more interesting idea is ***local search***, which we can think of as taking small steps through the space of functions, and adding the function at each step to the sequence F. This requires $\mathscr{F}$ to have some structure; for instance if $\mathscr{F}$ is parametrized by k real numbers, then we can think of the space of functions as $\mathbb{R}^k$; we will assume this is the case from here on. Local search methods include hill-climbing (adjusting a single parameter at each step in a way that decreases cost), and ***gradient descent***.

Gradient descent uses the gradient of the cost with respect to the parameters: $\frac{d\mathscr{L}}{d\theta}$, to find the direction of steepest descent. A ***learning rate***, α_t helps determine the step-size, and can be constant, changed according to some predetermined schedule, or chosen by some more sophisticated method that aims to find the optimal step-size, such as a ***line search*** [10]. The formula for gradient descent, then, is:

$$\theta_{t+1} = \theta_t - \alpha_t \frac{d\mathscr{L}}{d\theta_t} \tag{2.1}$$

Gradient descent methods are far and away the most popular way of optimizing ANNs. Initialization (how the first candidate function is chosen) is often important in local search, and has been shown to play a critical role in deep learning. Another natural idea is to use multiple paths, initialized at different locations, to search the space of functions.

Local search is an example of an iterative method, in which each new function is expected to be an improvement over all previous functions considered. Consistently decreasing the cost at each time-step is an attractive property, but may not be enough to find a good solution. The best solutions are ***global minima***, functions whose cost is minimal. A simple property that guarantees convergence to a global minimum is

[10] When training ANNs, it is common for the experimenter to tune the learning rate interactively. This is part of what has been called the "black art" of deep learning, since experienced researchers are often better at making these adjustments.

convexity. A function $f : X \to \mathbb{R}$ is convex if

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2), \forall x_1, x_2 \in X, \forall t \in [0,1].$$

A great deal of optimization research focuses on the convex case, which has a well-developed theory. Non-convex optimization is not understood nearly as well, although the success of deep learning has generated some interest. Methods developed for convex optimization are often used in non-convex optimization, despite the lack of theoretic guarantees that they enjoy in the convex case.

We say an optimizer converges if the sequence of functions it produces is convergent. When convergence to a global optimum cannot be proven, it is often possible to prove convergence to a local optimum, but local optima may be very suboptimal. Recent work [10] in deep learning suggests this may not be the case for DNNs, however.

The geometry of function space and parameter space may not correspond exactly; the ***natural gradient*** can correct for this discrepancy, but can also be more costly to compute. Techniques for approximating the natural gradient have been used successfully in DL.

2.4 Generalization, overfitting and regularization

A machine learning algorithm finds a function that does a good job of explaining the data it has observed, as measured by the cost. This function reflects the mathematical patterns that it has found in its observations. But apparent patterns are often coincidental, a result of random chance that doesn't properly reflect the data generating process. [11].

We don't want our model to focus on such patterns; in other words, we want it to ***generalize*** to new data generated by the same process as its previous observations, not just memorize the right answers for the training set. Informally, a function which performs well on inputs that it has encountered, but fails to generalize well is said to be ***overfitting***. The not-quite-opposite problem of ***underfitting*** occurs when a function fails

[11] ***Apophenia*** is the human tendency to perceive patterns in random data [74]. One notable example is the ***clustering illusion***, in which randomly distributed points appear to form clumps or clusters. Historically, Londoners developed theories about bombing patterns in WWII. The clustering illusion has also been offered (controversially) as an explanation for apparent regional clusters of cancer cases [12]

to capture important patterns in the data it *has* encountered.

My favorite example for explaining overfitting and underfitting is polynomial regression. In this setting, our model is an n-th degree polynomial. A higher degree polynomial can do a better job, even fitting the training data perfectly when $n-1$ equals the number of data points, but this almost always is due to extreme overfitting; see figure 2.1.

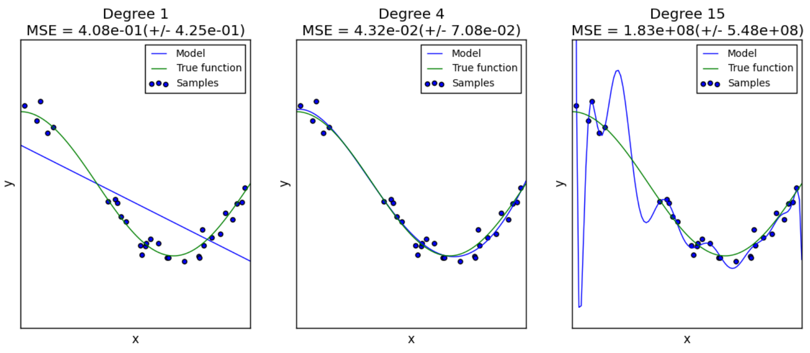

Figure 2.1: Polynomial regression: The degree 1 polynomial (left) is too simple and underfits. The degree 15 polynomial (right) is too complicated and overfits. The degree 4 polynomial (middle) strikes a good balance between overfitting and underfitting. This figure is taken from [50].

2.4.1 Early stopping

One popular way of preventing overfitting with a finite dataset is ***early stopping***. To perform early stopping, we randomly split our dataset into a three sets, called the ***training***, ***validation***, and ***test*** sets. We only train our model using the training set and perform model selection by evaluating cost on the validation set. Only once we have settled on a model and trained it (optionally using the validation set, as well) do we evaluate on the test set. The cost on the test set is then an unbiased evaluation of the trained model's performance on unseen data from the data generating process.

2.4.2 Regularization

Early stopping is a form of ***regularization***. A ***regularizer*** is a technique for preventing overfitting in a model by expressing a preference among hypotheses. A common form of regularizer adds a regularization term depending only on the hypothesis (and not the data) to the cost function.

The choice, made when selecting a model, of which hypotheses to consider, implicitly expresses a very strong prior, namely absolute certainty that the hypotheses not considered are incorrect. Just like the choice of model, regularization techniques express an inductive bias. Unlike model selection, regularization makes a ML algorithm favor certain hypotheses *a priori*, without completely *removing* any from consideration. Most regularizers introduce a new scalar hyperparameter specifying how strong the penalty should be, in effect expressing the confidence we have in the motivating inductive bias.

2.4.2.1 L_1 and L_2 regularization

Besides early stopping, another common way of regularizing is to encourage the parameters to be small. For real-valued parameters, larger values typically correspond to more extreme hypotheses, which are more likely to be overfit. The most common regularizers of this form penalize the L_1 or L_2 norm of the (real-valued) parameter vectors. The L_1 penalty has a tendency to encourage ***sparsity***, meaning a large number of parameters are chosen to be 0. This means the trained model can often be compressed by removing these parameters.

When training a model with gradient descent, early stopping can be viewed as an adaptive form of L_2 regularization, which chooses the strength of the penalty based on the validation cost [12].

These penalties have been used for neural nets, but are less common since the invention of dropout regularization [25].

2.4.3 Occam's razor and model complexity

Perhaps the most general prior is a preference for simpler models. Occam's Razor states that between two theories which explain our observations, we should choose the simpler theory. More specifically, there is a trade-off between how well a pattern

[12] This also requires reparametrizing so that all parameters are initialized to 0.

explains the data, and how simple the model is [13]. As suggested by the polynomial example, a simpler hypothesis often does a better job of generalizing. While regularization expresses a preference among hypotheses, we can also enforce a hard limit on the complexity of the hypotheses we consider. Removing more complex hypotheses from consideration should reduce overfitting, but what we ultimately care about in machine learning is generalization, not simplicity. Simplicity can also hurt generalization; if we do not consider the correct hypothesis because it is too complex, then our model is bound to underfit.

The concept of ***model complexity*** or ***model capacity*** corresponds to (inverse) simplicity in the sense of Occam's Razor. A model with more parameters is often more complex, but more sophisticated methods of evaluating complexity may be necessary; models can be ***over-parametrized***, meaning equivalent to a model with fewer parameters. A simple example is the function $f(x) = \theta_1 \theta_2 x$, with parameters $\theta_1, \theta_2 \in \mathbb{R}$; any such function could also be expressed with a single real-valued parameter $\theta = \theta_1 \theta_2$. In fact, considering the same function, now with $\theta_1 \in [0,1]$, $\theta_2 \in \{-1,1\}$, this model is *less* expressive than the model given by $f(x) = \theta x$, where θ can take on any real value, despite having *more* parameters.

There are more sophisticated theoretical notions of complexity, such as ***minimum description length (MDL)***, which measures how many bits it takes to encode a complete description of a model, and ***VC dimension***, which measures how many points a binary classification model can ***shatter***, i.e. classify correctly for any possible labeling, but there is no universally applicable method of comparing model complexity in practice.

When we have little prior knowledge about the data generating process, limiting the complexity of the model can be one of the best ways to prevent overfitting. For instance, a smaller neural network (i.e. one with fewer parameters) sometimes gives better performance, even with early stopping.

[13] One way of formalizing this trade-off is Bayesian Model Comparison.

2.4.4 Risk minimization

We've defined learning as a search for functions with low cost. The principle of ***risk minimization*** states that the cost we should seek to minimize should depend on the data generating process, not the observed data. Since the data generating process is generally unknown, this cost cannot be evaluated directly. ***Structural risk minimization (SRM)*** provides a theoretical justification for using the cost on the training data (called the ***empirical risk***) as a proxy for risk The trick is to control model complexity using the VC dimension. Specifically, the risk can be bounded (with some probability) by the empirical risk and the VC confidence. The VC confidence is a function of the VC dimension and the desired confidence that the bound holds (more confidence results in a looser bound).

Support Vector Machines (SVMs) are a popular algorithm based on the principles of SRM. Calculating the VC dimension for ANNs is not straightforward or common in current practice.Early stopping is similar in spirit to SRM, however; the complexity of an ANN generally increases through training, as it has more time to grow its parameter values [14].

2.4.5 Bias and variance

Formally, we can view overfitting and underfitting as describing the bias and variance of the outputs of a learning procedure as a function of the training data. We view the learned function as a statistical estimator of a true function, and ask ourselves: if we were to receive a new dataset from the same generating process, how would our results change? High variance means that the function learned would be very different if it were trained on another dataset. High bias means that the result is suboptimal in some consistent way, regardless of what training data has been observed. This decomposition demonstrates how overfitting and underfitting are not opposites: a particularly bad model (e.g. with high complexity and a poor inductive bias) can suffer from both, whereas a good model will not suffer too much from either, given sufficient training data. When

[14] ANN parameters are usually initialized to values close to 0. Absent any regularization, the parameters generally grow throughout training.

there is not enough data to observe the trends one hopes to learn about, it is still possible to have a good model, with sufficient prior knowledge; in the limit of no data, this means knowing the correct function in advance.

2.5 The ML research process

The process of ML research tends to progress as follows:

```
repeat
    choose: task, training cost, evaluation cost, model, learning algorithm
    train the model using the learning algorithm and training cost
    compute the evaluation cost of the learned function
until satisfied or out of time/patience
```

Depending on the goal of the research project, frequently only one of the task, cost functions, model, or learning algorithm will be manipulated in the research loop, while the others are held fixed. It is important to remember that components can interact to determine which function is learned, however.

It is common to use good results from previous work as a ***baseline*** to compare against. Common research objectives are:

- evaluating the generality of a method (by comparing across tasks)
- finding which of several methods performs better on a specific task
- gauging the robustness of one component to changes in the others (e.g. asking: how does my learning algorithm work with a different model?)

This approach has the advantage of allowing relatively straightforward comparison of different methods. However, each time we repeat the process, the learning algorithm starts over from scratch. This contrasts sharply with the life-long learning that animals exhibit, and that is desirable for many systems in practice. There is some work that attempt to ***transfer*** knowledge across tasks, but very few projects simulate an ongoing perpetual learning process. Also, as mentioned above, for real systems, it is necessary to consider the appropriateness of the evaluation cost as a measure of task performance; the algorithm with the best evaluation cost might not be the one you want to use in your real-world system.

CHAPTER 3

DEEP LEARNING AND REPRESENTATION LEARNING

The phrase "deep learning" has been the subject of some controversy [59]. Bengio [5] defines depth via "the number of levels of composition of non-linear operations in the function learned"; According to this definition, which we call the ***technical definition***, training any model with depth greater than one is deep learning. In practice, the phrase has been applied imprecisely to a wide range of work, starting with ***deep belief networks (DBNs)*** [24]. Most but not all of this work uses deep artificial neural networks (ANNs).

In this chapter, we'll describe feedforward ANNs (FNNs) and their relation to representation learning. We'll also cover some of the most basic and important methods of deep learning.

3.1 Artificial neural networks

3.1.1 Biological inspirations and analogies

Artificial neural networks (ANNs) encompass a variety of machine learning models inspired by neuroscientific knowledge of how real biological neurons work [1]. In particular, the ***activation*** of the "neurons" in an ANN can be thought of as representing the ***firing rate*** of a biological neuron [2].

ANNs are, however, a gross simplification of biological neural networks. There are many different types of neurons and neurotransmitters in the brain, with different properties. But the most glaring difference in my mind is that ANNs operate in discrete

[1] ANNs are usually referred to simply as "neural nets" by deep learning researchers, but should be distinguished from the biological neural networks that exist in the brain, as well as ***spiking neural networks***, a more detailed model inspired by biological neural networks that operates in continuous time.

[2] Neurons are believed to communicate via binary electro-chemical signals called ***action potentials***, or ***spikes***; the firing rate is the number of spikes per unit of time. Some neuroscientists believe that all the information a neuron transmits is encoded in the firing rate, but assessing this claim is not straightforward, and others believe the precise timing of the spikes also carries information. In particular, how to convert a spike-train (a signal of electric potential at the neuronal membrane) into a signal specifying the instantaneous firing rate is controversial [75].

time; I know of no justification for modeling a biological neural network this way, other than convenience.

3.1.2 Neural net basics

An ANN is a weighted (directed or undirected) graph, whose nodes represent ***neurons*** (and are often called ***units***), and whose edges represent synapses and are called ***weights*** or ***connections***. The ANN architectures we consider in this thesis are directed graphs with inputs as roots and outputs as leaves. Each neuron is computed as a (possibly random, but usually deterministic) function of its parents.

Each neuron computes its ***activation*** as a function of its parents' activations. The activations are transient and input-dependent; they are *not* parameters. The parameters of the network determine which functions the neurons compute. The activation of a neuron is typically computed as a linear function (i.e. a weighted sum) of its inputs (called the ***preactivation***) followed by a nonlinearity; see figure 3.1 for a depiction of this computation.

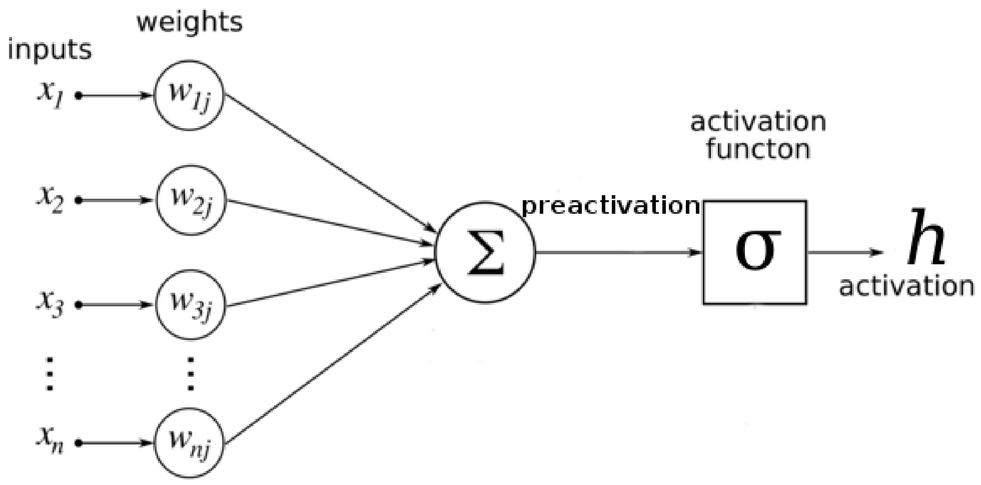

Figure 3.1: The computation performed by a single neuron in a neural network. This figure is adapted from [73].

In the most basic standard ANN architecture, called the ***multi-layer perceptron (MLP)***, neurons are organized into ***fully connected*** layers, which share parents and children (see figure 3.2). Since each unit typically has a (nonlinear) ***activation function***, σ, (also called a ***nonlinearity***), each layer of units increases the depth of the architecture.

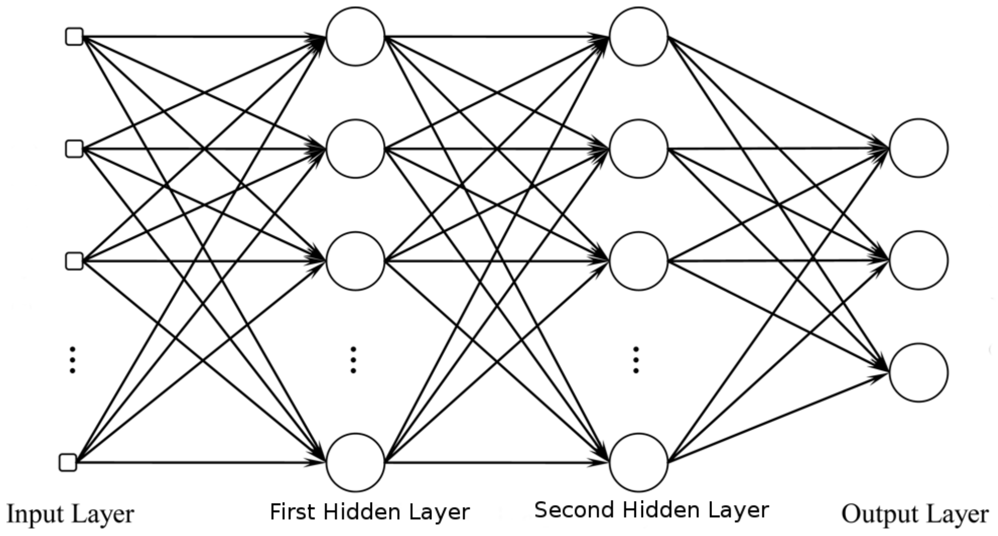

Figure 3.2: The computational graph of an MLP with two hidden layers and three output units. This figure is adapted from [58].

The weights on an ANN specify the strength of the connections between neurons. We typically add a constant bias term to the preactivation of each neuron; this is equivalent to giving all neurons a parent with a fixed activation of 1. The weights for each layer are organized into a ***weight matrix***, with rows corresponding to parents and columns, children. Each entry in this matrix then corresponds to one edge of the computational graph. This allows the computations each layer of neurons performs to be written compactly as:

$$h = \sigma(Wx + b)$$

where h are the activations of the child layer, x the parents' activations, and the parameters are W and b: the weight matrix and vector of bias terms, respectively (σ is the layer's activation function). When we have have more than two layers, we denote the input layer as x, the output layer, y, and the i-th hidden layer as h_i. We index nonlinearities and parameters similarly, so we can write the computation a FNN with l hidden layers performs in closed form as follows:

$$y = \sigma_y(W_y h_l + b_y) \tag{3.1}$$

$$y = \sigma_y(W_y \sigma_l(W_l h_{l-1} + b_l) + b_y) \tag{3.2}$$

$$y = \sigma_y(W_y \sigma_l(W_l \sigma_{l-1}(W_{l-1}(\ldots\sigma_1(W_1 x + b_1)\ldots + b_{l-1}) + b_l) + b_y) \tag{3.3}$$

For layers that are not fully connected (i.e. missing some edges between the parent and child layers), we can simply set the corresponding entries of the weight matrices to 0. Computational graphs are sometimes depicted with nodes representing layers as opposed to individual units.

3.1.3 Activation functions

Nonlinear activation functions are what make deep nets deep architectures. The composition of linear (or affine) functions is itself a linear (or affine) function, so without nonlinearities, stacking multiple layers is redundant. The output nonlinearity must be compatible with the cost function, and the choice of hidden layer activation functions can have a dramatic effect on performance. Some activation functions, such as the softmax, cannot be computed elementwise.

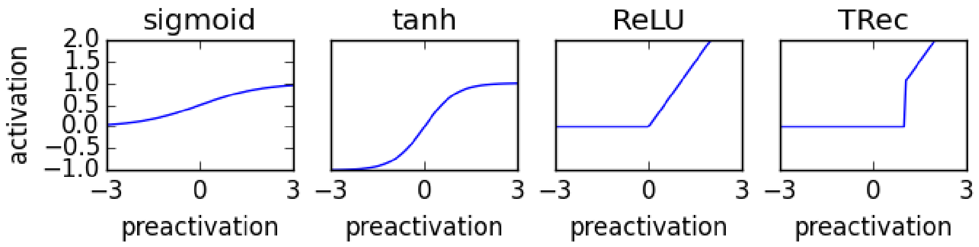

Figure 3.3: Common activation functions for neural nets, and the TRec activation function with threshold 1.

3.1.3.1 Logistic sigmoid

The classical activation function is the logistic sigmoid (usually just called sigmoid), given by:

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

The range of the sigmoid is $(0,1)$, and it is the default choice of output nonlinearity in binary classification, where we typically interpret the output as representing the parameter of a Bernoulli distribution.

3.1.3.2 Hyperbolic tangent

The hyperbolic tangent (tanh) function is another s-shaped curve, but its range is $(-1,1)$. This activation function was traditionally used for recurrent connections, where it was found to outperform sigmoid. The equation for tanh is:

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

3.1.3.3 Softmax

The softmax activation function is used almost exclusively in the output layer. It converts a set of real numbers into the parameters of a categorical distribution, in such a way that increasing the preactivation of a unit always increases its activation (i.e. it is monotonic in every component). This is accomplished by first converting all the preactivations into positive real numbers with the exponential function, and then normalizing the result to sum to 1:

$$\text{softmax}(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

3.1.3.4 Rectified linear and variants

Widespread use of the rectified linear (ReLU) activation function is one of the more recent trends in training neural nets. This function is computed as

$$\text{ReLU}(x) = \max(0, x)$$

ReLU generally significantly outperforms so called ***saturating nonlinearities***, such as sigmoid and tanh [3]. This may be because the gradients for saturating nonlinearities become very small when they are saturating (i.e. near their asymptotes). Since weights tend to increase during training, preactivations would typically increase as well, pushing these units into their saturating regime. For ReLU, negative preactivations give a gradient of 0, but in practice this doesn't seem to be as much of an issue; since activations are input-dependent, most units should be active (i.e. non-zero) for some inputs, in which cases they have a significant gradient.

Due to its empirical success, several variants of ReLU have been explored [14, 23, 35, 40]. In this thesis, we use the ***TRec*** activation function [35], which is used without bias parameters, and instead has a "threshold" hyperparameter (set to 1 in the original paper and 0 in this thesis). The activation is then just given by:

$$\text{TRec}(x) = x(x > thresh).$$

3.2 Features and representations

The way data is presented to a learning algorithm, called a ***representation***, has a large effect on results. Informally, we think of different mathematical objects as representing the same data. Formally, any (potentially random) function of a dataset can be thought of as transforming the dataset into a new representation. A ***feature map*** or ***feature detector*** is a function that computes a new representation for each data point independently. The dimensions of such a representation are called ***features*** (although this term is also

[3] Having bounded activations is desirable in many settings, however, and so saturating nonlinearities remain widely used.

commonly used to refer to feature maps). Not all representations can be decomposed in this way, however; for instance, a dataset of M N-dimensional real vectors might be transformed by subtracting the global mean of all MN scalar values from each vector. The process of using feature maps to create a new representation of a data point is called ***feature extraction***.

There is no single metric for the quality of a given representation, and in practice, quality is task-dependent. Nonetheless, there are several ways in which representations can be evaluated, and proven to be more or less useful.

- *human-interpretability*: Facilitating human understanding or visualization is useful. Descriptive statistics is mostly used for this purpose. Projecting high-dimensional data into 2-dimensional space for visualization using t-SNE [68] is one example.
- *independence of features*: Statistically independent features can be thought of as ***disentangling underlying factors of variation***, i.e. the higher-level concepts that are important for understanding the world. The idea of ***natural kinds*** in philosophy expresses a similar idea. A classic example is expression and pose for pictures of human faces.
- *compression*: Compression of data reduces storage and communication costs.
- *sparsity*: A sparse representation, with fewer active components, could greatly reduce computation, and facilitate compression.
- *performance in pipeline*: Often, the ultimate measure of the utility of a representation is simply the performance in some task of interest we get using that representation. We call the entire system including the transformation of raw data into a new representation and the algorithm that learns to perform the task a ***pipeline***.

Machine learning practitioners often use domain knowledge to specify the choice of representation directly, a process called ***feature engineering***. Using machine learning to search for good representations is called ***representation learning*** [4]. Bengio et al. [6]

[4] Unfortunately, the acronym RL in ML is already claimed by reinforcement learning. Still, sometimes RL is used to refer to representation learning, but I avoid it.

argue that generic inductive biases for representation learning are important for developing more general AI, and give several examples of such inductive biases, some of which can be used to motivate deep learning techniques. Powerful and generic representation learning techniques would allow an algorithm to learn task-relevant representations with minimal human guidance.

While most common representation learning techniques learn feature maps, the feature learning *process* typically depends very much on the entire dataset, since we want the features to do a good job of representing any data that is likely to be sampled from the data distribution.

3.2.1 Deep learning and representation learning

In ANNs, every unit/neuron represents a feature map. Most of the power of DNNs comes from the representations they learn. DNNs learn a ***feature hierarchy***, where ***higher order features*** (those computed after the first hidden layer) are computed as a function of features from the layer below. In most recent work, all features are learned jointly using supervised learning and SGD. The top layers of DNNs generally use the same basic linear classification or regression functions as previous methods; DNNs outperform other methods because they have better features.

A DNN learns a representation that is both ***deep*** and ***distributed*** [6]. These characteristics of their architecture encode a prior almost as generic as locality, but considerably more powerful.

Traditional clustering techniques (see 2.2.7.2) have a many-to-one mapping from inputs to features, and are not distributed. Distributed representations represent different inputs with a combination or ***code*** of features, as opposed to just one. This is related to the independence prior we mentioned above; different aspects of an input can often be varied (almost) independently. For instance we could represent a car in terms of its color, condition, etc. As a human, when we see a model of car we've never seen before, it might be somewhat surprising, but seeing that same model in a different color or condition afterwards is not. As this example is meant to demonstrate, distributed representations allow for non-local generalization.

Distributed representations can be very expressive with a small number of features, giving a richer description of each input, and making nuanced distinctions between similar inputs. If we think of binary features, there are exponentially many more combinations of features than there are individual features.

A deep representation composes features in a hierarchy, allowing higher-level features to reuse the same lower-level features. This can also result in exponential gains in efficiency, since each path through the hierarchy (from inputs to outputs) represents a different computation [6]. A classic example with vision is edges being composed into objects. Most objects can be represented well as a combination of many edges in specific locations, so low level features which say where there exist edges of each orientation and thickness in an image is useful regardless of the image content. Likewise, an object, which is a high-level feature, could be in any location and still have similar semantics, although it would contain a completely different set of edges. So the mapping of feature containment is many-to-one in both directions. This example demonstrates the synergy between deep and distributed representations.

3.2.1.1 Unsupervised and semi-supervised deep representation learning

Representations can be learned with UL or SL. Early successes of deep learning (e.g. Hinton et al. [24]) combined UL and SL; this is known as ***semi-supervised learning***. In modern work, supervised learning predominates. In 2015, new semi-supervised learning techniques with impressive results [45, 53] renewed interest in this topic. There is vastly more unlabeled than labeled data, making semi-supervised learning very attractive [5].

[5] Some prominent researchers also believe that most human learning is unsupervised, although I think reinforcement learning better explains the way humans learn. The disagreement may be partially semantic; most people seem to agree on the importance of having a good model of the world. The question is how do humans learn what information is important to represent? There is much too much information to represent everything, and I believe that reward signals are a main determinant of what humans remember or forget.

3.2.1.2 Autoencoders and PCA

A classic unsupervised representation learning algorithm is principal component analysis (PCA). PCA finds the most important directions in the data space by performing an eigen-decomposition of the empirical covariance matrix, X. This is equivalent to finding a linear projection, W, into a lower-dimensional space such that reconstructing the data as $W^T WX$ minimizes reconstruction error. This linear projection provides a compressed representation of the data. PCA can often reduce the dimensionality of data points by half or more with almost no reconstruction error.

An ***autoencoder (AE)*** is a feedforward neural net where the targets are the inputs. Autoencoders can be thought of as a non-linear version of PCA. The simplest autoencoder is the ***bottleneck autoencoder***, which uses a hidden representation with less dimensions than its inputs. This prevents it from learning the identity mapping, and makes the connection with PCA clear; with one hidden layer, the bottleneck AE reconstruction becomes $W^T \sigma(WX)$. Other kinds of autoencoders can learn ***overcomplete*** representations (with more hidden than input dimensions), and use regularization (such as a sparsity penalty on the hidden activations) to avoid learning the trivial identity map.

An autoencoder is often viewed as the composition of an ***encoder***, which transforms data into some representation, and a ***decoder*** which transforms points in representation space back into the original data space. Generative models such as the variational autoencoder (VAE) [34] and nonlinear independent component estimation (NICE) [11] can be described in this way. These models aim to transform the data distribution into a representation where features are independent; samples can then be generated by sampling in the representation space and decoding.

We will return to autoencoders in Chapter 6, where we propose an RNN architecture involving AEs.

3.3 Some important deep learning techniques

In this section, we cover a few popular techniques of deep learning. These are standard elements of the deep learning toolkit that all DL researchers should be familiar with. While the basic models and techniques of deep learning have existed since at least the 1980s, and much of the progress made since then can be attributed to faster computers and larger datasets, there are some new techniques which have seen rapid and widespread adoption due to their generality, simplicity, and ability to significantly improve performance and/or cut down on training time.

3.3.1 Faster computers

One of the key events in the history of deep learning is the adoption of graphics processing units (GPUs) for training DNNs. GPUs are massively parallel processors developed for video games and other graphics intensive processes, such as computer aided design (CAD). By themselves, GPUs tend to give a speed-up of 10-50x over CPUs. In industrial applications it is common to use hundreds of Graphics Processing Units (GPUs) to train a network in parallel with near linear speed-up. Someone who started training a model in the year 2000 could potentially start training all over again today and overtake the older model by tomorrow. But optimization of state of the art models can still takes days to weeks on state of the art hardware.

3.3.2 Optimization techniques: stochastic gradient descent, momentum, RMSProp, Adam

Optimization of DNNs is poorly understood, and it is not clear why training models takes so long. If we understood what the obstacles to optimizing DNNs were, it is possible they could be trained much faster.

DNNs are almost always trained with methods based on gradient descent (section 2.1). Gradient descent can be seen as approximating a function by its first-order Taylor series expansion. Since this approximation is generally not accurate, we can only trust it locally, and so we make a small change to the parameters and re-evaluate the gradient.

Using a higher-order Taylor series expansion allows us to better approximate a function; this is the motivation for ***second-order methods***. For reasonably sized DNNs, computing the second derivative at a point (called the Hessian) is computationally infeasible, but some methods attempt to approximate it.

While the techniques presented here are popular today, they could very well be replaced by new methods at any time. Optimization is an active area of research in deep learning. New optimizers are regularly devised, and theoretical work attempts to understand the nature of the neural net optimization problems, e.g. the shape of the cost surface. Some researchers are also exploring methods of improving optimization via smarter initialization or parametrizations. Batch normalization (BN) [29] (see 3.3.3) is an example of the latter which has resulted in much faster optimization as well improving performance.

3.3.2.1 Stochastic gradient descent (SGD)

The standard technique for optimizing ANNs is (mini-batch) stochastic gradient descent (SGD). SGD is a classic technique which replaces the gradient of the training cost with a stochastic estimate given by the cost of a single randomly sampled data point. For training ANNs, a ***mini-batch*** of, e.g. 128 examples, is used instead of just one, and the mini-batches are sampled without replacement. Algorithmically, this amounts to repeatedly shuffling the data, then looping over the entire training set; each repeat is called an ***epoch***. Minibatch sizes typically range from around 16 to 128, with batch-size up to about 1000 being used in some settings. Smaller batch-size often results in better performance, but below 128 examples usually starts to slow down training time for computational reasons. As with regular ("full-batch") gradient descent, the learning rate is often decreased as training progresses. Other optimization hyper-parameters may also be adjusted automatically or by a researcher during training.

3.3.2.2 Momentum

Momentum is another classic technique [52], with an intuitive physical explanation: We simply think of our steps in parameter space as changes in velocity instead of position, and add some friction to exponentially decay the influence of past steps. This results in the following update rule:

$$v_{t+1} = \gamma v_t + \frac{d\mathscr{L}}{d\theta_t} \tag{3.4}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1} \tag{3.5}$$

Here v represents the velocity, and $\gamma \in [0,1]$ is the decay due to friction. A related method is called ***Nesterov's accelerated gradient*** [46] (or "Nesterov momentum" in the DL community), and can be seen as an approximate second-order method.

3.3.2.3 RMSProp and Adam

RMSProp [66] is another heuristic method that can be seen as using approximate second-order information. It was motivated as a mini-batch version of RProp [54], a method that only uses the sign of the gradient. While this changes the direction of steepest descent, it can help avoid plateaus in parameter space, where the gradient in one direction dominates, necessitating smaller step-sizes and slowing training. RMSProp simulates RProp using mini-batches by dividing the gradient for each parameter by a decaying average of its norms.

Adam [33] is a recently developed optimizer that has quickly become one of the standard tools of DL. It can be viewed as combining RMSProp and momentum (although they are both computed on the gradient rather than being applied sequentially). Adam goes one step further than RMSProp+momentum by correcting for the bias they exhibit as estimators of the gradient's distribution's (second and first, respectively) moments.

3.3.3 Deep learning regularization: dropout and batch normalization

Dropout [25] is one of the most powerful and popular regularizers for feedforward neural networks. Dropout consists of multiplicative noise applied to a neural network's units, and can be viewed as training an ensemble of networks that share parameters. Traditionally this noise would be Bernoulli (so some units are "dropped out" of the computational graph at random), but this has been generalized to other noise distributions, e.g. Gaussian [63].

Recently, ***batch normalization (BN)*** [29] has challenged dropout's position as the pre-eminent ANN regularizer. Batch normalization is motivated by the desire to reduce ***internal covariate shift***. ***Covariate shift*** refers to changes in the distribution $P(X)$ over inputs between training and test time. The idea of internal covariate shift is that the distribution of inputs *to each layer* of a network change *throughout training*. Applying BN is thought to help with optimization by allowing the network to keep the distribution of inputs to each layer approximately constant during training. What is surprising is that it also tends to have a regularization effect. This is particularly surprising since BN does not restrict the model's capacity. Thus we can view this regularization as due entirely to the effect BN has on the optimization process.

CHAPTER 4

RECURRENT NEURAL NETWORKS

All the work in this thesis deals with Recurrent Neural Networks. Recurrent Neural Networks (RNNs) are a type of ANN strongly associated with Deep Learning. In particular, RNNs using Long-Short-Term Memory (LSTM-RNNs, or simply LSTMs) have recently become very popular in a variety of machine learning tasks.RNNs are precisely those ANNs whose computational graphs contain cycles, called ***recurrent connections*** (acyclic ANNs are referred to as ***feedforward networks (FNNs)***).

RNNs take as input a sequence x with an arbitrary number of terms (each term, x_t being a vector in some finite dimensional space). RNNs process their inputs sequentially and recursively, computing a new hidden-state, h_t, (a fixed-length vector) after each subsequent term is processed. RNNs naturally handle sequences of variable length by repeatedly using the same ***transition function*** to compute h_{t+1} from h_t and x_{t+1}. The transition function can also depend on the time-step, for instance there could be a different transition function for even and odd time-steps. The hidden state can be seen as a compressed representation of the input sequence up to time t.

For a given input RNN is ***unrolled*** into a feedforward net by specifying a number of time-steps (see 4.1). The cycles in an RNN represent the dependencies of h_{t+1} on h_t in the transitions function; their computations specify how the current hidden state influences the next hidden state. Unrolling creates an acyclic computational graph that explicitly represents the hidden states at each time-step and the recurrent connections between them.

RNNs may also compute an output vector at each time-step and/or after processing a complete input sequence. These outputs can be a function of the hidden-state at that time-step and/or preceding time-steps. In examples of previous work:

- The final hidden-state was used as a compressed representation of the input sequence [64]

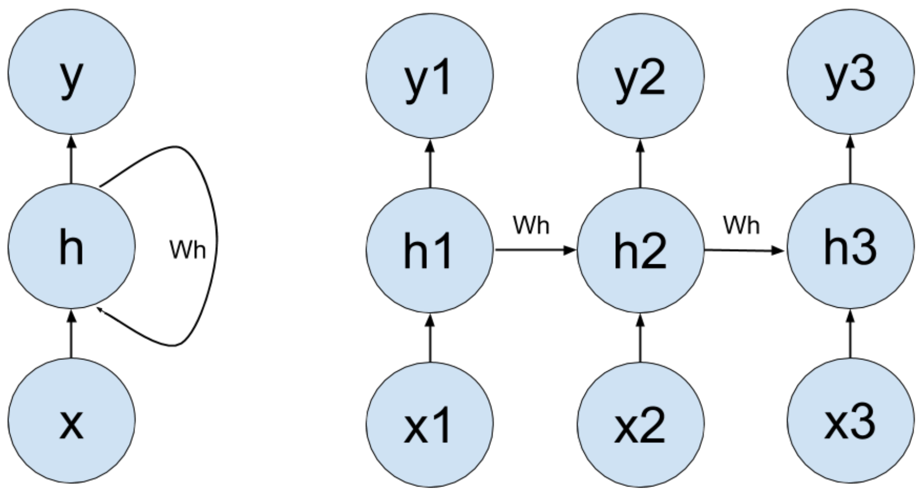

Figure 4.1: Unrolling a simple recurrent neural network for three time-steps. Left: the original computational graph, including a cycle. Right: the unrolled, acyclic graph. We use the same input, output, and transition (W_h) matrices at each time-step. This figure represents the traditional RNN architecture.

- The state at each time-step was used to predict an aligned time-series [15]
- An "attention mechanism" was used to create a soft mapping of terms in the inputs sequence to terms in an (unaligned) output sequence [3]
- The same attention mechanism is used to predict an output with the same number of terms as the input [70]

This thesis includes two threads of work on RNNs. The first work introduces and evaluates a novel regularization technique for RNNs. This work is currently under review for the 2016 International Conference on Learning Representations. The second work, in progress, develops a novel RNN architecture for supervised learning in the case where both inputs and targets are sequences.

4.1 Recurrent neural network architectures

4.1.1 Simple RNNs

Without qualification, "recurrent neural network" is often taken to refer to a specific architecture of RNN, where the transition function computes an affine transformation followed by a nonlinearity. We refer to this architecture, which also includes an output map composed of affine transformation and nonlinearity, as a ***simple RNN (SRNN)***. These operations are parametrized by eight matrices W_x, W_h, W_y and biases b_h, b_y, as follows:

$$h_{t+1} = \sigma_h(W_h h_t + W_x x_{t+1}) \tag{4.1}$$

$$y_{t+1} = \sigma_y(W_y h_{t+1}) \tag{4.2}$$

$$\tag{4.3}$$

Traditionally, the nonlinearity σ_h was chosen to be the hyperbolic tangent function (tanh), while σ_y is task dependent. More recently, using a rectified linear (ReLU) nonlinearity for σ_h was shown to be give much better performance on some tasks, provided the matrix W_h is initialized as the identity or a multiple thereof [38].

4.1.2 Long short-term memory

The currently dominant RNN architecture, Long Short-Term Memory, or LSTM [26], is significantly more complicated. In LSTM, the hidden state as we've defined it is decomposed as $h_t = \tilde{h}_t, c_t$, where $\tilde{h}_t$ is commonly (and confusingly) called the hidden state and c_t are called ***memory cells***. There are several variants of the LSTM architecture, the equations below describe the variant currently considered standard.

$$\begin{aligned}
i_{t+1} &= \tanh(W_{hi}h_t + W_{xi}x_{t+1} + b_i) && (4.4)\\
m_{t+1} &= \text{sigm}(W_{hm}h_t + W_{xm}x_{t+1} + b_m) && (4.5)\\
f_{t+1} &= \text{sigm}(W_{hf}h_t + W_{xf}x_{t+1} + b_f) && (4.6)\\
o_{t+1} &= \text{sigm}(W_{ho}h_t + W_{xo}x_{t+1} + b_o) && (4.7)\\
c_{t+1} &= f_{t+1}c_t + i_{t+1}m_{t+1} && (4.8)\\
h_{t+1} &= o_{t+1}\tanh(c_t) && (4.9)\\
y_{t+1} &= \sigma_y(W_y h_{t+1}) && (4.10)\\
& && (4.11)
\end{aligned}$$

Here "sigm" stands for the sigmoid nonlinearity. The first four computations represent (in order) the ***input***, ***input-modulation***, ***forget***, and ***output*** gates. After computing the gates, the memory cells and then the rest of the hidden state are computed.

LSTM networks typically outperform SRNNs by a significant margin.

4.1.3 Gated recurrent units

A somewhat simpler architecture with similar features to LSTM was proposed by Cho et al. [9]. This architecture, called Gated Recurrent Unit (GRU) has shown similar performance to LSTM on nearly every task where they have been compared. GRU, like LSTM, uses multiplicative gating to construct a new candidate hidden state, and has an element-wise ***reset gate*** which determines how much of the previous hidden state is preserved, similarly to the forget gate in LSTM. Unlike LSTM, GRU has no explicit memory cells, which makes it faster to compute.

4.1.4 Deep RNNs

While unrolled RNNs are already arbitrarily deep, the computations performed at each time-step are not. Stacked-RNNs [15, 60] have recently become the default architecture for Speech Recognition as well as Machine Translation and other NLP tasks.

Pascanu et al. [49] also proposed several other methods for making the computation performed at each time-step deep.

4.1.5 Attention modules

Graves [16] used a differentiable form of attention over an input sequence of characters to generate handwriting from a string of text. Following quickly on the heals of the first successful application of RNNs to Machine Translation [9, 64], Bahdanau et al. [3] demonstrated the utility of attention mechanisms for the same task; in this case, the attention was applied to the hidden states of an "encoder" RNN. The attention module produces a ***glimpse*** [3] or ***reading*** [2] of the input, with some aspects captured at higher resolutions. In a static model, an attention module would simply learn a prior over where important information is located, but a sequential model can use these readings to inform future decisions of where to attend.

4.1.6 Memory interfaces

Several recent works [18, 19, 32, 36, 72] have used RNN architectures involving more durable forms of memory inspired by computer architectures or data-structures. These models frequently use attention mechanisms to focus on various parts of their memory. These innovations provide tremendous advantages over architectures with transient memory dynamics (such as SRNN or LSTM) in several toy-tasks such as learning algorithms for sorting [18]. Large improvements on real-world tasks has not yet been observed, although Kumar et al. [36] do report state-of-the-art performance on several datasets.

4.1.7 Grammar cells

Michalski et al. [44] propose a novel RNN structure involving gating that models *transitions* between states as opposed to modelling the states directly. These transitions are assumed to be linear, but the model learns a ***product of experts***, allowing it to represent exponentially many linear maps and decide which one to apply. In Chapter 6 of this thesis, we describe this model in detail and propose a novel generalization for supervised learning.

4.2 Regularization for recurrent neural networks

Since our main contribution is a regularization technique for RNNs, we provide a summary of previous regularization techniques that have been applied to RNNs. As with other ANNs, RNNs are almost always trained with early stopping, but can also frequently benefit from additional regularization. Traditional regularizers, such as L_1 and L_2 penalties can of course be applied to RNNs, but in contemporary work, this is uncommon.

Additive Gaussian ***weight noise*** has also been a popular regularizer for RNNs [16]. As the name suggests, this amounts to adding random noise to the weights of a network at each training step. More recently, dropout and batch normalization have been applied to RNNs (see the previous chapter for a discussion of these techniques). Applying dropout in RNNs is straightforward, but Pham et al. [51] found that this makes training difficult and suggest only noising the weights between successive layers at the same time-step ("up the stack"), not between time-steps ("forward through time"). This idea was produced independently and popularized by Zaremba et al. [78]). Just like with dropout, applying BN to RNNs is straightforward, but has only been shown to work when applied up the stack and not forward through time [1, 37].

None of the regularizers we've mentioned so far were designed with RNNs in mind. Designing a regularizer specifically for RNNs is the motivation of the work we present in the next chapter.

CHAPTER 5

REGULARIZING RNNS BY STABILIZING ACTIVATIONS

5.1 Prologue to the paper

This paper was joint work with Roland Memisevic. It has been submitted as a conference paper to the International Conference on Learning Representations, 2016, and been favorably reviewed [1]. I proposed the idea, performed the experiments, and wrote the article. Roland provided some seed code, provided useful discussions, and advised on what to focus on in the experiments and write-up.

Our contribution is to propose and validate a novel approach to regularization for recurrent neural networks (RNNs). Deep learning models are often heavily over-parameterized, and require some form of regularization (e.g. early stopping) for most applications. Regularization can sometimes be foregone for large enough datasets. But it remains crucially important for real-world tasks such as speech recognition.

The most commonly used regularization methods for RNNs are not specific to RNNs or even sequential models. We view this as an opportunity to devise regularizers that target RNNs in particular. We consider stability of the hidden state norms as a regularization objective, and demonstrate that it can improve performance substantially and outperforms previously used regularization techniques on the popular TIMIT benchmark.

Our experiments demonstrate the promise of this technique and suggest further work on devising regularizers specific to sequential models such as RNNs could be a very fruitful avenue of research. Much exploration remains to be done in this space of ideas. Although we compare different architectures, tasks, and regularization penalties on the sequence of hidden states, many more experiments can and should be done. Whether through experiment or theory, a better understanding of the reasons behind the success of this regularizer would be welcome.

[1] The three reviewers gave it scores of 6, 7, and 8 out of 10.

5.2 Abstract

We stabilize the activations of Recurrent Neural Networks (RNNs) by penalizing the squared distance between successive hidden states' norms. This penalty term is an effective regularizer for RNNs including LSTMs and IRNNs, improving performance on character-level language modelling and phoneme recognition, and outperforming weight noise and dropout. We achieve state of the art performance (17.5% PER) for RNNs on the TIMIT phoneme recognition task, without using beam search or an RNN transducer. With this penalty term, IRNN can achieve similar performance to LSTM on language modelling, although adding the penalty term to the LSTM results in superior performance. Our penalty term also prevents the exponential growth of IRNN's activations outside of their training horizon, allowing them to generalize to much longer sequences.

5.3 Introduction

Overfitting in machine learning is addressed by restricting the space of hypotheses (i.e. functions) considered. This can be accomplished by reducing the number of parameters or using a regularizer with an inductive bias for simpler models, such as early stopping. More effective regularization can be achieved by incorporating more sophisticated prior knowledge. Keeping an RNN's hidden activations on a reasonable path can be difficult, especially across long time-sequences. With this in mind, we devise a regularizer for the state representation learned by temporal models, such as RNNs, that aims to encourage stability of the path taken through representation space. Specifically, we propose the following additional cost term for Recurrent Neural Networks (RNNs):

$$\beta \frac{1}{T} \sum_{t=1}^{T} (\|h_t\|_2 - \|h_{t-1}\|_2)^2$$

Where h_t is the vector of hidden activations at time-step t, and β is a hyperparameter controlling the amounts of regularization. We call this penalty the ***norm-stabilizer***, as it successfully encourages the norms of the hiddens to be stable (i.e. approximately constant across time). Unlike the "temporal coherence" penalty of Jonschkowski and

Brock [31], our penalty does *not* encourage the state representation to remain constant, *only its norm*.

In the absence of inputs and nonlinearities, a constant norm would imply orthogonality of the hidden-to-hidden transition matrix for simple RNNs (SRNNs). However, in the case of an orthogonal transition matrix, inputs and nonlinearities can still change the norm of the hidden state, resulting in instability. This makes targeting the hidden activations directly a more attractive option for achieving norm stability. Stability becomes especially important when we seek to generalize to longer sequences at test time than those seen during training (the "training horizon").

The hidden state in LSTM [26] is usually the product of two squashing nonlinearities, and hence bounded. The norm of the memory cell, however, can grow linearly when the input, input modulation, and forget gates are all saturated at 1. Nonetheless, we find that the memory cells exhibit norm stability far past the training horizon, and suggest that this may be part of what makes LSTM so successful.

The activation norms of simple RNNs (SRNNs) with saturating nonlinearities are bounded. With ReLU nonlinearities, however, activations can explode instead of saturating. When the transition matrix, W_{hh} has any eigenvalues λ with absolute value greater than 1, the part of the hidden state that is aligned with the corresponding eigenvector will grow exponentially to the extent that the ReLU or inputs fails to cancel out this growth.

Simple RNNs with ReLU [38] or clipped ReLU [21] nonlinearities have performed competitively on several tasks, suggesting they can learn to be stable. We show, however, that IRNNs performance can rapidly degrade outside of their training horizon, while the norm-stabilizer prevents activations from exploding outside of the training horizon allowing IRNNs to generalize to much longer sequences. Additionally, we show that this penalty results in improved validation performance for IRNNs. Somewhat surprisingly, it also improves performance for LSTMs, but not tanh-RNNs.

To the best of our knowledge, our proposal is entirely novel. Pascanu et al. [48] proposed vanishing gradient regularization, which encourages the hidden transition to preserve norm in the direction of the cost derivative. Like the norm-stabilizer, their cost depends on the path taken through representation space, but the norm stabilizer does not prioritize cost-relevant directions, and accounts for the effects of inputs as well. A hard constraint (clipping) on the activations of LSTM memory cells was previously proposed by [57]. Hannun et al. [21] use a clipped ReLU, which also has the effect of limiting activations. Both of these techniques operate element-wise however, whereas we target the activations' norms. Several other works have used penalties on the difference of hidden states rather than their norms [31, 71]. Other regularizers for RNNs that do not target norm stability include weight noise [30], and dropout [47, 51, 78].

5.4 Experiments

5.4.1 Character-level language modelling on PennTreebank

We show that the norm-stabilizer improves performance for character-level language modeling on PennTreebank [41] for LSTM and IRNNs, [2] but *not* tanh-RNNs. We present results for $\beta \in \{0, 50, 500\}$. We found that values of $\beta > 500$ could slightly improve performance, but also resulted in much longer training time on this task. Scheduling β to increase throughout training might allow for faster training. Unless otherwise specified, we use 1000/1600 units for LSTM/SRNN, and SGD with learning rate=.002, momentum=.99, and gradient clipping=1. We train for a maximum of 1000 epochs and use sequences of length 50 taken without overlap. When we encounter a NaN in the cost function, we divide the learning rate by 2, and restart with the previous epoch's parameters.

For LSTMs, we either apply the norm-stabilizer penalty only to the memory cells, or only to the hidden state (in which case we remove the output tanh, as in [13]). Although Greff et al. [20] found the output tanh to be essential for good performance, removing it gave us a slight improvement in this task. We compare to tanh and ReLU (with and without bias), with a grid search across cost weight, gradient clipping, and learning rate. For simple RNNs, we found that the zero-bias ReLU (i.e. TRec [35] with threshold 0) gave the best performance. The best performance for ReLU activation functions is

[2] As in Le et al. [38], we initialize W_{hh} to be an identity matrix in our experiments

obtained with the penalty applied. For tanh-RNNs, the best performance is obtained without any regularization. Results are better with the penalty than without for 9 out of 12 experiment settings.

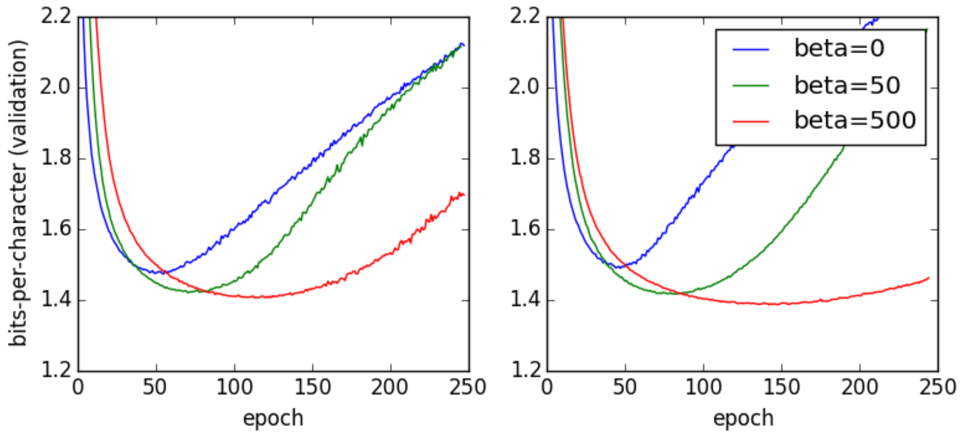

Figure 5.1: Learning Curves for LSTM with different values of β. Penalty is applied to the hidden state (Left), or the memory cells (Right).

5.4.1.1 Alternative costs

We compare 8 alternatives to the norm-stabilizer cost on PennTreeBank for IRNNs without biases (see Table 5.III), using the same setup as in 5.4.1. These include relative error, L_1 norm, absolute difference, and penalties that don't target successive time-steps. The following two penalties performed very poorly and were not included in the table: $|\Delta \|h_t\|_2|$, $\|h_t\|_2^2$. We find that our proposal of penalizing successive states' norms gives the best performance, but some alternatives seem promising and deserve further inves-

Table 5.I: LSTM Performance (bits-per-character) on PennTreebank for different values of β.

	$\beta = 0$	$\beta = 50$	$\beta = 500$
penalize hidden state	1.47	1.41	**1.39**
penalize memory cell	1.49	1.42	**1.40**

tigation. In particular, the relative error could be more appropriate; unlike the norm-stabilizer cost, it cannot be reduced simply by dividing all of the hidden states by a constant. The value 5 was chosen as a target for the norms based on the value found by our proposed cost; in practice it would be another hyperparameter to tune. The success of the other regularizers which encourage (L_2) norm stability indicates that our inductive bias in favor of stable norms is useful.

Table 5.II: Performance with and without norm-stabilizer penalty for different activation functions. Gradients are clipped at 1 in the first and third, and 10^6 in the second and fourth columns.

	$lr = .002, gc = 1$	$lr = .002$	$lr = .0002, gc = 1$	$lr = .0002$
tanh, $\beta = 0$	1.71	**1.55**	2.15	2.15
tanh, $\beta = 500$	1.57	2.70	1.79	1.80
ReLU, $\beta = 0$	1.78	1.69	1.93	1.93
ReLU, $\beta = 500$	1.74	1.73	**1.65**	2.04
TRec, $\beta = 0$	1.62	1.63	1.95	1.88
TRec, $\beta = 500$	**1.48**	1.49	1.56	1.56

Table 5.III: Performance (bits-per-character) of zero-bias IRNN with various penalty terms designed to encourage norm stability.

	$(\Delta h_t)^2$	$(\Delta \|h_t\|_2)^2$	$(\frac{\Delta\|h_t\|_2}{\|h_t\|_2})^2$	$(\Delta \|h_t\|_1)^2$	$(\|h\|_2 - 5)^2$	$(\|h_0\|_2 - \|h_T\|_2)^2$
$\beta = 50$	1.84		1.60	2.96	1.49	3.81
$\beta = 500$	2.19	1.48	1.50	3.18	1.50	1.54

5.4.2 Phoneme recognition on TIMIT

We show that the norm-stabilizer improves phoneme recognition on the TIMIT dataset, outperforming networks regularized with weight noise and/or dropout. For these experiments, we use a similar setup to the previous state of the art for an RNN on this task [15], with CTC [17] and bidirectional LSTMs with 3 layers of 500 hidden units (for each direction). We train with Adam [33] using learning rate=.001, and gradient clipping=200.

Unlike Graves et al. [15], we do not use beam search or an RNN transducer. We early stop after 25 epochs without improvement on the development set.

We apply norm-stabilization to the hidden activations (in this case we *do* use the output tanh as is standard) with $\beta \in \{0, 50, 500\}$, and use standard deviation .05 for weight noise, and p=.5 for dropout. We try all pair-wise combinations of the regularization techniques. We run 5 experiments for each of these 10 settings, and report the average phoneme error rate (PER). Combining weight noise and norm-stabilization gave poor performance, with some networks failing to train, these results are omitted. Adding dropout had a minor effect on results. Norm-stabilized networks had the best performance (see figure 5.2 and table 5.IV). Inspired by these results, we decided to train larger networks with more regularization, and observed further performance improvements (see table 5.V). We also used a higher "patience" for our early stopping criterion here, terminating after 100 epochs without improvement. Unlike previous experiments, we only ran one experiment with each of these settings. The network with 1000 hidden units and $\beta = 1000$ achieved dev/test PER of 16.7%/17.5%. This is the state of the art test set performance for RNNs on this task, although Tóth [67] achieved 13.9%/16.7% using convolutional neural networks.

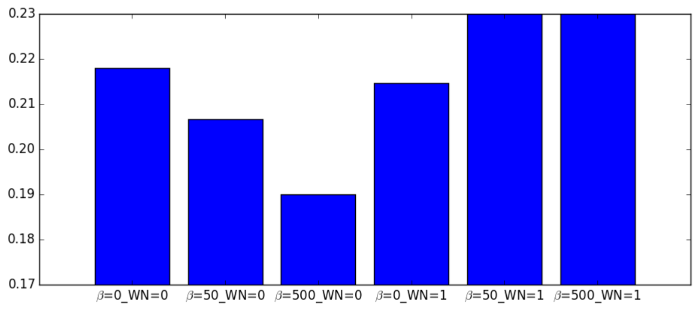

Figure 5.2: Average PER on TIMIT core test set for different combinations or regularizers. The norm-stabilizer (β) shows a clear positive effect on performance. Weight noise (WN) also improves performance but less so. Combining weight noise with norm-stabilization gives poor results.

Table 5.IV: Phoneme Error Rate (PER) on TIMIT for different experiment settings, average of 5 experiments. Norm-stabilized networks achieve the best performance. The regularization parameters are: β - norm stabilizer, p - dropout probability, σ - standard deviation of additive Gaussian weight noise.

	$\beta=0$ $\sigma=0$ $p=0$	$\beta=50$ $\sigma=0$ $p=0$	$\beta=500$ $\sigma=0$ $p=0$	$\beta=0$ $\sigma=.05$ $p=0$	$\beta=0$ $\sigma=0$ $p=.5$	$\beta=50$ $\sigma=0$ $p=.5$	$\beta=500$ $\sigma=0$ $p=.5$	$\beta=0$ $\sigma=.05$ $p=.5$
test	21.8	20.7	**19.0**	21.5	21.9	20.9	19.4	21.1
dev	19.6	18.6	**16.9**	19.1	19.5	18.5	17.0	18.9

Table 5.V: Phoneme Error Rate (PER) on TIMIT for experiments with n hidden units and more norm-stabilizer regularization (β). Networks regularized with weight noise $\sigma = .05$ when $\beta = 0$.

	$\beta=0$ $n=750$	$\beta=500$ $n=750$	$\beta=1000$ $n=750$	$\beta=1500$ $n=750$	$\beta=0$ $n=999$	$\beta=500$ $n=999$	$\beta=1000$ $n=999$	$\beta=1500$ $n=999$
test	21.9	18.8	18.6	18.0	21.8	19.5	**17.5**	18.6
dev	19.6	16.8	**16.2**	**16.2**	19.1	17.4	16.7	16.7

5.4.3 Adding task

The adding task [26] is a toy problem used to test an RNN's ability to model long-term dependencies. The goal is to output the sum of two numbers seen at random time-steps during training; inputs at other time-steps carry no information. Each element of an input sequence consists of a pair $\{n, i\}$, where $n \in [0, 1]$ is chosen at uniform random and $i \in \{0, 1\}$ indicates which two numbers to add. We use sequences of length 400. In Le et al. [38], none of the models were able to reduce the cost below the "short-sighted" baseline set by predicting the first (or second) of the indicated numbers (which gives an expected cost of $\frac{1}{12}$) for this sequence length. We are able to solve this task more successfully. We use uniform initialization in $[-.01, .01]$, learning rate=.01, gradient clipping=1. We compare across nine random seeds with and without the norm-stabilizer (using $\beta = 1$). The norm-stabilized networks reduced the test cost below $\frac{1}{12}$ in 8/9 cases, averaging .059 MSE. The unregularized networks averaged .105 MSE, and only outperformed the short-sighted baseline in 4/9 cases, also failing to improve over a constant predictor in 4/9 cases.

5.4.4 Visualizing the effects of norm-stabilization

To test our hypothesis that stability helps networks generalize to longer sequences than they were trained on, we examined the costs and hidden norms at each time-step.

Comparing identical SRNNs trained with and without norm-stabilizer penalty, we found LSTMs and RNNs with tanh activation functions continued to perform well far beyond the training horizon. Although the activations of LSTM's memory cells could potentially grow linearly, in our experiments they are stable. Applying the norm-stabilizer does significantly decrease their average norm and the variability of the norm, however (see figure 5.3). IRNNs, on the other hand, suffered from exploding activations, resulting in poor performance, but the norm-stabilizer effectively controls the norms and maintains a high level of performance; see figure 5.4. Norm-stabilized IRNNs' performance and norms were both stable for the longest horizon we evaluated (10,000 time-steps).

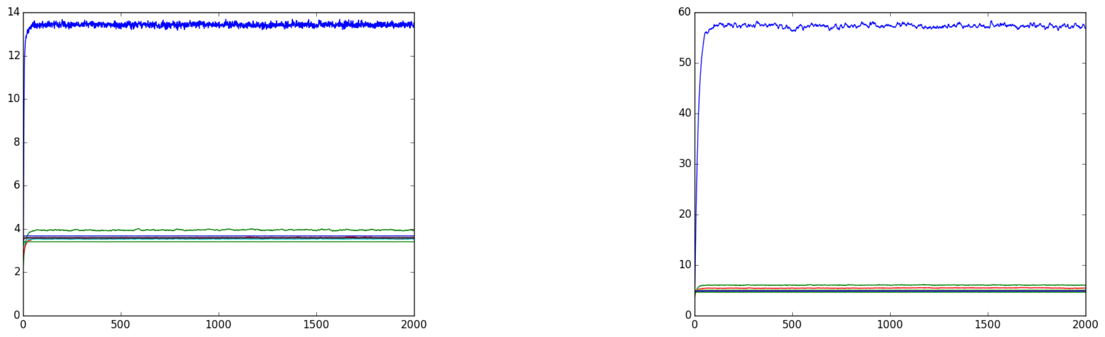

Figure 5.3: Norm (y-axis) of LSTM hidden states (Left) and memory cells (Right) for different values of β, across time-steps (x-axis). The blue curve at the top is when $\beta = 0$. Non-zero values dramatically reduce the mean and variance of the norms. LSTM memory cells have the potential to grow linearly, but instead exhibit natural stability.

For more insight on why the norm-stabilizer outperforms alternative costs, we examined the hidden norms of networks trained with values of β ranging from 0 to 800 on a dataset of 1000 length-50 sequences taken from wikipedia [28]. When we penalize the difference of the initial and final norms, or the difference of the norms from some fixed value, increasing the cost results in norms that grow less, but it does *not* change the *shape* of the norms; they still begin to explode within the training horizon (see figure 5.5). For the norm-stabilizer, however, increasing the penalty significantly delayed (but did not completely eradicate) activation explosions on this dataset.

We also noticed that the distribution of activations was more concentrated in fewer hidden units when applying norm-stabilization on PennTreebank. Similarly, we found that the forget gates in LSTM networks had a more peaked distribution (see figure 5.6), while the average across dimensions was lower (so the network was forgetting more on average at each time step, but a small number of units were forgetting less). Finally, we found that the eigenvalues of regularized IRNN's hidden transition matrices had a larger number of large eigenvalues, while the *unregularized* IRNN had a much larger number of eigenvalues closer to 1 in absolute value (see figure 5.6). This supports our hypothesis that orthogonal transitions are not inherently desirable in an RNN. By explicitly encouraging stability, the norm-stabilizer seems to favor solutions that maintain stability via selection of active units, rather than restricting the choice of transition matrix.

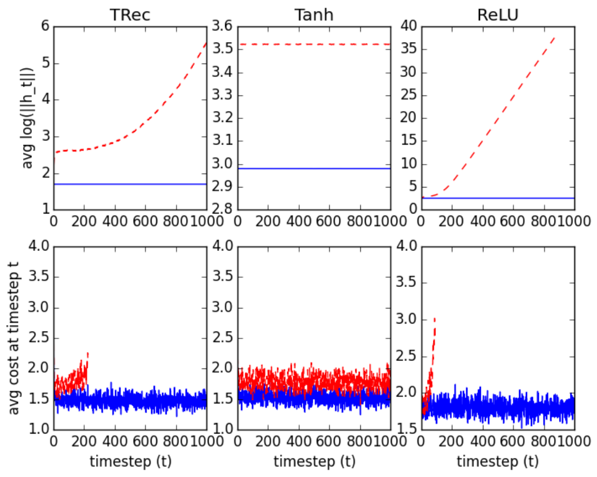

Figure 5.4: Top: average logarithm of hidden norms as a function of time-step. Bottom: average cost as a function of time-step. Solid blue - $\beta = 500$, dashed red - $\beta = 0$. Notice that IRNN's activations explode exponentially (linearly in the log-scale) within the training horizon, causing cost quickly go to infinity outside of the training horizon (50 time-steps).

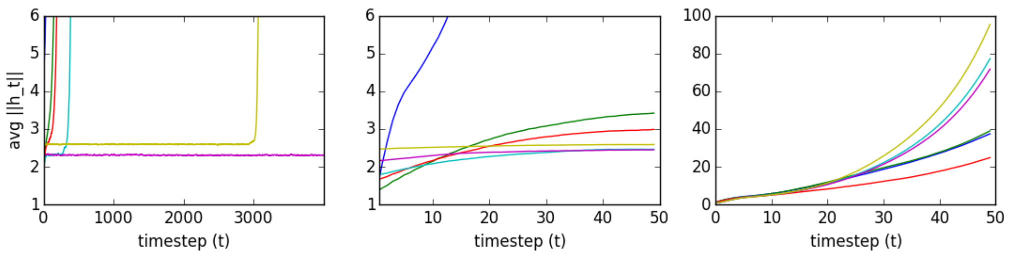

Figure 5.5: Hidden norms as a function of time-step for values from 0 to 400 of the norm-stabilizer (Left and Center) vs. a penalty on the initial and final norms (Right). The norm-stabilizer delays the explosion of activations by changing the shape of the curve, extending the flat region.

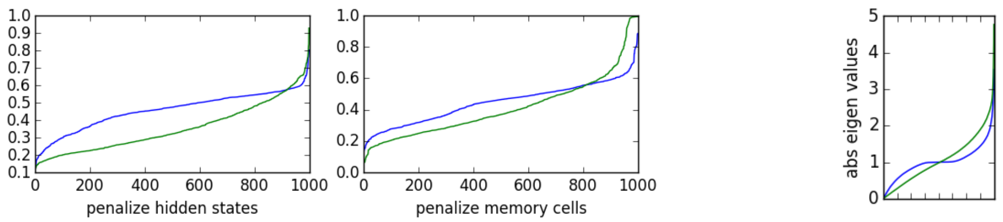

Figure 5.6: Left: sorted distribution of average forget-gates for different memory cells in LSTM. Right: sorted absolute value of eigenvalues of W_{hh} in IRNN. Blue - $\beta = 0$, Green - $\beta = 500$

5.5 Conclusion

We introduced norm-based regularization of RNNs to prevent exploding or vanishing *activations*. We compared a range of novel methods for encouraging or enforcing norm stability. The best performance is achieved by penalizing the squared difference of subsequent hidden states' norms. This penalty, the ***norm-stabilizer***, improved performance on the tasks of language modelling and addition tasks, and gave state of the art RNN performance on phoneme recognition on the TIMIT dataset.

Future work could involve:

- Exploring the relationship between stability and generative modelling with RNNs
- Applying norm-regularized IRNNs to more challenging tasks
- Applying similar regularization techniques to feedforward nets

Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laborotory (AFRL) . The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. We appreciate the many k80 GPUs provided by ComputeCanada. The authors would like to thank the developers of Theano [4] and Blocks [69]. Special thanks to Alex Lamb, Amar Shah, Asja Fischer, Caglar Gulcehre, Cesar Laurent, Dmitriy Serdyuk, Dzmitry Bahdanau, Faruk Ahmed, Harm de Vries, Jose Sotelo, Marcin Moczulski, Martin Arjovsky, Mohammad Pezeshki, Philemon Brakel, and Saizhen Zhang for useful discussions and/or sharing code.

CHAPTER 6

CONDITIONAL PREDICTIVE GATING PYRAMIDS

This chapter contains a brief report on some work in progress. We explore the idea of directly modelling dynamics as part of the hidden state in an RNN. Our first step in this direction is to extend the model of Michalski et al. [44] to handle supervised learning.

The original motivation of this work was ***speech synthesis***, that is, creating a conditional generative model of speech conditioned on content (and possibly other factors, such as characteristics of the speaker). Speech is encoded as a very high-dimensional sequence of real-valued audio samples, e.g. 16000/s. At the level of audio samples or ***frames*** of several samples (e.g. 160), speech exhibits slowly-changing dynamics. This is especially true of ***voiced*** speech (e.g. of vowels), which contains a periodic signal whose frequency changes slowly.

6.1 Predictive gating pyramids

6.1.1 Gated autoencoders

A gated autoencoder (GAE) [42, 43] takes two separate inputs, x and y, and uses its hidden units (called ***mapping units***) to model the relationship between them [1]. For instance, for a pair of images, the hidden state might represent "rotated by 30 degrees clockwise". Since the representation only represents the *relationship between images*, and not the images themselves, such a representation can be used as a conditional generative model to predict what an arbitrary image would look like when rotated by 30 degrees.

Concretely, we can consider relationships in terms of multiplicative interactions, and create a new representation for a pair of inputs given by the pairwise products of their dimensions (i.e. outer product of the vectors x and y). A gated autoencoder can be

[1] "Gating" in DL just refers to multiplication. We can view one of the inputs to a multiplication operation as a gate valve in a faucet. How far this gate is opened determines how much information from the other input is able to "flow" through. The analogy fails somewhat in that the two inputs are symmetric.

viewed as a regular autoencoder taking these outer products as inputs. If we flatten the outer product into a vector, then we have a weight matrix as usual, otherwise, we have a weight tensor, since each mapping unit takes a matrix as input.

6.1.1.1 Factored gated autoencoders

Gated models are often ***factored*** to reduce the number of parameters [43]. This involves replacing the pairwise product of input dimensions with an element-wise product of features extracted from the inputs. The features are represented as another layer of hidden units. This is equivalent to factoring the weight tensor into a product of three matrices, and results in the following equation for the mapping units, h (with $\odot$ representing elementwise multiplication):

$$h = \sigma(W_{hf}(W_{fx}x \odot W_{fy}y)) \tag{6.1}$$

The reconstruction of y given x and h is given by:

$$y = W_{fy}^T(W_{fx}x \odot W_{hf}^T h) \tag{6.2}$$

While the number of parameters of a gated model is quadratic in the input dimension, for a ***factored gated autoencoder (FGAE)***, it is linear in the input dimension (and the number of features).

6.1.2 Predictive gating pyramids

Predictive Gating Pyramids (PGPs) [44] use FGAEs to model the relationship between successive time-steps in an input sequence. In the case of speech, these time-steps would be frames of audio samples. Once we have inferred the relationship between two successive frames, we can predict future frames by assuming the same relationship holds, i.e. that the first-order dynamics (or “abstract velocity”) are constant. In the image example, this would allow us to create a video of an object rotating at some constant angular velocity.

PGPs stack the architecture of the FGAE to model *n*-th order dynamics, e.g. "abstract acceleration". While the first-order dynamics represented the relationship between input frames, the second-order dynamics represent the relationship *between relationships*, and so on. With second-order dynamics, we could create a video of an object rotating with constant angular acceleration, so that the rotation slows down and eventually changes direction.

To the extent that the PGP is able to capture the true dynamics of the system it models, it can generalize perfectly simply by keeping the *n*-th order dynamics constant. However, for signals whose dynamics do not change smoothly or predictably, a more powerful model is needed.

6.2 Conditional predictive gating pyramids

PGPs were introduced for unsupervised learning; we extend them to perform supervised learning by adding inputs which directly affect the hidden state. Since the hidden state represents the dynamics up to level *n*, the inputs affect the outputs indirectly by changing which transition is applied. We call this model the ***conditional predictive gating pyramid (CPGP)***. Unlike PGPs, CPGPs *can* handle arbitrary changes to the dynamics, so long as these changes can be predicted from their inputs.

As a proof of concept, we experiment with modelling a sine wave whose frequency changes according to a random walk. We use a soft one-hot encoding of the frequency as an input to the model, and frames of 10 audio samples as outputs. For this experiment, we use a 3-layer PGP (with abstract velocity, acceleration, and jerk). Our inputs affect the jerk units additively via a 1-layer neural network with sigmoid nonlinearity. We find that the model is able to learn this task and to generalize to new frequencies and beyond the training horizon (see 6.1). The model was trained for about a week, and showed no sign of nearing convergence or overfitting at that point.

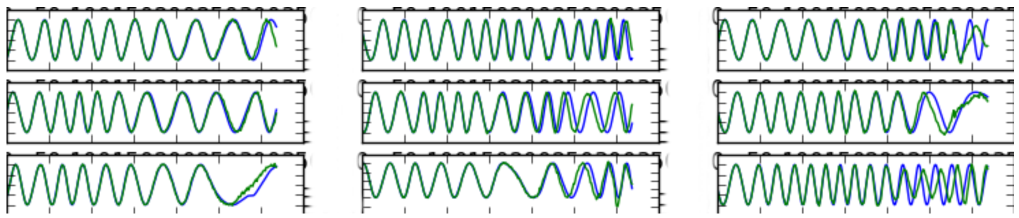

Figure 6.1: Conditional predictive gating pyramid trained on sine waves with frequencies modulated by a random walk. Blue is the ground truth sequence, green is the model's prediction. The frequency is input in a soft one-hot encoding. The model was trained on sequences half this length, but generalizes well to longer sequences at test time.

6.3 Future work

Much remains to be done. As a first step, more experiments are needed, especially comparing CPGPs with other models. Practically speaking, finding a way to significantly decrease the training time for PGP/CPGP would be helpful.

In terms of applications:

- Although we worked on applying this technique to the problem of speech synthesis, our results there were invalidated by bugs in the code.
- We also began work using CPGPs as a ***model*** of the environment [2]; we believe this could be a good fit, in particular for robotics tasks.

There are also many avenues to explore with the PGP family of models.

- We would like to construct a generative PGP with ***proprioception***, the ability to look at its outputs and learn how to change its dynamics in order to avoid a compounding of errors. We proposed the ***forward-down-up*** procedure in which the top-level dynamics are re-inferred at each time-step based on what was just generated, as one approach to proprioception. Another is to use the CPGP architecture taking the previously generated output as input.
- The input model for CPGP could be made recurrent, so that inputs can be used to directly manipulate dynamics at future time-steps.

[2]In RL, a model predicts the next state (and possibly reward) from previous state and action

- We could also learn a representation for the outputs, which could also be recurrent.
- We could explore the role of factoring and consider ***partially factored*** architectures which learn multiple "channels" of features to represent their inputs. Each channel would be multiplied elementwise with all the other channels in the representation of the other input.

Furthermore, this particular architecture is just one approach to the general idea of modelling dynamics. Dynamics play an important role in describing most real-world systems; for instance a complete description of a pendulum requires both position and velocity.

While here we learn representations of the dynamics, this might not be necessary. We could also fix some dynamic elements and produce the hidden state directly via their dynamics, using new information (e.g. from the inputs, or the results of computations being performed by the evolution of the hidden state) to change the influence of each of these dynamic elements. It would also be interesting to view these dynamic elements as a memory reservoir external to the hidden state, with the dynamics serving a clock-like function. The network would then interact with this memory reservoir via some interface, as in the models from section 4.1.6. More speculatively, the network could potentially learn how to "program" the dynamics of the memory reservoir, as well.

CHAPTER 7

CONCLUSION

In this thesis, I introduced the field of deep learning and discussed some of the key conceptual elements and practices of contemporary deep learning research. My main goal was to provide an unique and personal overview of deep learning and representation learning, and their relevance to the goals of artificial intelligence research. In order to make this work accessible and useful, I aimed to explain fundamental machine learning concepts at a high-level of abstraction and precision. I also attempted to motivate the now common perspective that learning plays a central role in intelligence. I placed special emphasis on recurrent neural networks, one of the most popular and powerful families of deep learning models, and the subject of my highlighted research contributions: norm-stabilization, a successful and novel approach to regularization in recurrent neural networks, and conditional predictive gating pyramids, a supervised learning version of an existing recurrent model.

As I write this thesis, the popularity of deep learning within the artificial intelligence research community continues its spectacular ascent. Meanwhile, artificial intelligence itself is an increasingly hot topic in society at large, thanks in part to the successes of deep learning (although it remains to be seen how far deep learning techniques can take us towards general artificial intelligence). In my experience, more people are aware of the idea that artificial intelligence could pose an existential risk to humanity than recognize the terms “deep learning” or “machine learning”. With this in mind, I hope that my thesis can help educate people about the current state and direction of artificial intelligence research.

In particular, I view the rise of machine learning, and now deep learning as part of an historic progression in artificial intelligence towards approaches that can make more decisions with less human input and oversight. Reinforcement learning and unsupervised learning, which require less human supervision than the presently popular supervised learning techniques, are commonly thought to be the next frontiers in artificial intelligence research. Ultimately, learning to learn (“meta-learning”) may result in algorithms capable of performing the challenging intellectual tasks involved in current research and implementation, leaving humans (hopefully) the role of deciding the purpose or goal of a learning algorithm, which will itself give rise to interesting intellectual and ethical challenges. How we approach this task of setting goals for the intelligent systems we create may ultimately be the most important project for the field of artificial intelligence.

BIBLIOGRAPHY

[1] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, and others. Deep Speech 2: end-to-end speech recognition in English and Mandarin. ***arXiv:1512.02595***, 2015.

[2] Philip Bachman, David Krueger, and Doina Precup. Testing visual attention in dynamic environments. ***arXiv:1510.08949***, 2015.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. ***arXiv:1409.0473***, 2014.

[4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. ***CoRR***, abs/1211.5590, 2012.

[5] Y. Bengio. Learning deep architectures for AI. ***Foundations and Trends in Machine Learning***, 2(1):1–127, 2009.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: a review and new perspectives. ***IEEE Transactions on Pattern Analysis and Machine Intelligence***, 35(8):1798–1828, 2013.

[7] Jon Bird and Paul Layzell. The evolved radio and its implications for modelling the evolution of novel sensors. In ***Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC'02.***, volume 2, pages 1836–1841. IEEE, 2002.

[8] Nick Bostrom. ***Superintelligence: Paths, Dangers, Strategies***. Oxford University Press, Oxford, UK, 1st edition, 2014. ISBN 0199678111, 9780199678112.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. ***arXiv:1406.1078***, 2014.

[10] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks. ***arXiv:1412.0233***, 2014.

[11] Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: non-linear independent components estimation. ***CoRR***, abs/1410.8516, 2014.

[12] Atul Gawande. The cancer-cluster myth. ***The New Yorker***, February 1999. URL `http://www.newyorker.com/magazine/1999/02/08/the-cancer-cluster-myth`.

[13] Felix A. Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In ***International Joint Conference on Neural Networks (IJCNN)***, pages 189–194, 2000.

[14] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. ***arXiv:1302.4389***, 2013.

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

[16] Alex Graves. Generating sequences with recurrent neural networks. ***arXiv:1308.0850***, 2013.

[17] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In ***Proceedings of the 23rd International Conference on Machine Learning***, ICML '06, pages 369–376, 2006.

[18] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. ***arXiv:1410.5401***, 2014.

[19] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In ***Advances in Neural Information Processing Systems***, pages 1819–1827, 2015.

[20] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: a search space odyssey. ***CoRR***, abs/1503.04069, 2015.

[21] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. ***arXiv:1412.5567***, 2014.

[22] John Haugeland. ***Artificial Intelligence: the very idea***. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985. ISBN 978-0-262-08153-5.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: surpassing human-level performance on imagenet classification. ***arXiv:1502.01852***, 2015.

[24] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. ***Neural Computation***, 18(7):1527–1554, 2006.

[25] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. ***arXiv:1207.0580***, 2012.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. ***Neural Computation***, 9(8):1735–1780, 1997.

[27] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. ***Neural Networks***, 4(2):251–257, 1991.

[28] Marcus Hutter. The human knowledge compression contest. 2012. URL `http://prize.hutter1.net/`.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. ***arXiv:1502.03167***, 2015.

[30] Kam-Chuen Jim, C. Lee Giles, and Bill G. Horne. An analysis of noise in recurrent neural networks: convergence and generalization. ***IEEE Transactions on Neural Networks***, 7(6):1424–1438, 1996.

[31] Rico Jonschkowski and Oliver Brock. Learning state representations with robotic priors. ***Autonomous Robots***, 39(3):407–428, 2015.

[32] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. ***arXiv:1503.01007***, 2015.

[33] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. ***CoRR***, abs/1412.6980, 2014.

[34] Diederik P. Kingma and Max Welling. Auto-Encoding variational Bayes. ***arXiv:1312.6114***, 2013.

[35] Kishore Konda, Roland Memisevic, and David Krueger. Zero-bias autoencoders and the benefits of co-adapting features. ***arXiv:1402.3337***, 2014.

[36] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: dynamic memory networks for natural language processing. ***arXiv:1506.07285***, 2015.

[37] César Laurent, Gabriel Pereyra, Philémon Brakel, Ying Zhang, and Yoshua Bengio. Batch normalized recurrent neural networks. ***arXiv:1510.01378***, 2015.

[38] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. A simple way to initialize recurrent networks of rectified linear units. ***CoRR***, abs/1504.00941, 2015.

[39] Shane Legg and Marcus Hutter. A collection of definitions of intelligence. ***arXiv:0706.3639***, 2007.

[40] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In ***Proceedings of the 30th International Conference on Machine Learning***, volume 30 of ***ICML '13***, pages 1–6, 2013.

[41] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. ***Comput. Linguist.***, 19(2): 313–330, 1993.

[42] Roland Memisevic. ***Non-linear latent factor models for revealing structure in high-dimensional data***. dissertation, University of Toronto, 2008.

[43] Roland Memisevic. Gradient-based learning of higher-order image features. In ***IEEE International Conference on Computer Vision (ICCV)***, pages 1591–1598, 2011.

[44] Vincent Michalski, Roland Memisevic, and Kishore Konda. Modeling deep temporal dependencies with recurrent grammar cells"". In ***Advances in neural information processing systems***, pages 1925–1933, 2014.

[45] Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, and Shin Ishii. Distributional smoothing with virtual adversarial training. ***arXiv:1507.00677***, 2015.

[46] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). ***Soviet Mathematics Doklady***, 27:372–376, 1983.

[47] M. Pachitariu and M. Sahani. Regularization and nonlinearities for neural language models: when are they needed? ***arXiv: 1301.5650***, 2013.

[48] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. ***CoRR***, abs/1211.5063, 2012.

[49] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. ***arXiv:1312.6026***, 2013.

[50] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: machine learning in Python. ***Journal of Machine Learning Research***, 12:2825–2830, 2011.

[51] Vu Pham, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. ***CoRR***, abs/1312.4569, 2013.

[52] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. ***USSR Computational Mathematics and Mathematical Physics***, 4(5):1–17, 1964.

[53] Antti Rasmus, Mathias Berglund, Mikko Honkala, Harri Valpola, and Tapani Raiko. Semi-supervised learning with ladder networks. In ***Advances in Neural Information Processing Systems***, pages 3532–3540, 2015.

[54] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In ***IEEE International Conference on Neural Networks***, pages 586–591, 1993.

[55] Stuart Russell. Learning agents for uncertain environments (Extended Abstract). In ***Proceedings of the Eleventh Annual Conference on Computational Learning Theory***, COLT' 98, pages 101–103, New York, NY, USA, 1998.

[56] Stuart J. Russell and Peter Norvig. ***Artificial intelligence: a modern approach***. Prentice Hall series in artificial intelligence. Prentice Hall, Englewood Cliffs, N.J, 1995. ISBN 978-0-13-103805-9.

[57] Hasim Sak, Andrew Senior, Kanishka Rao, Ozan Irsoy, Alex Graves, Françoise Beaufays, and Johan Schalkwyk. Learning acoustic frame labeling for speech recognition with recurrent neural networks. In ***2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)***, pages 4280–4284. IEEE, 2015.

[58] Ömer Galip Saracoglu and Hayriye Altural. Color regeneration from reflective color sensor using an artificial intelligent technique. ***Sensors***, 10(9):8363–8374, 2010.

[59] Juergen Schmidhuber. Deep learning in neural networks: an overview. ***Neural Networks***, 61:85–117, 2015.

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

[61] H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. ***Applied Mathematics Letters***, 4(6):77–80, 1991.

[62] David Silver. Deep reinforcement learning, 5 2015. URL `http://www.iclr.cc/lib/exe/fetch.php?media=iclr2015:silver-iclr2015.pdf`. Keynote talk at the International Conference on Learning Representations, 2015.

[63] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. ***The Journal of Machine Learning Research***, 15(1):1929–1958, 2014.

[64] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In ***Advances in neural information processing systems***, pages 3104–3112, 2014.

[65] Gerald Tesauro. Practical issues in temporal difference learning. ***Machine Learning***, 8:257–277, 1992.

[66] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[67] László Tóth. Combining time- and frequency-domain convolution in convolutional neural network-based phone recognition. In ***IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP)***, pages 190–194, 2014.

[68] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. ***Journal of Machine Learning Research***, 9:2579–2605, 2008.

[69] Bart van Merriënboer, Dzmitry Bahdanau, Vincent Dumoulin, Dmitriy Serdyuk, David Warde-Farley, Jan Chorowski, and Yoshua Bengio. Blocks and fuel: frameworks for deep learning. ***CoRR***, abs/1506.00619, 2015.

[70] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. ***arXiv:1506.03134***, 2015.

[71] Tsung-Hsien Wen, Milica Gasic, Nikola Mrksic, Pei-hao Su, David Vandyke, and Steve J. Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. ***CoRR***, abs/1508.01745, 2015.

[72] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. ***arXiv:1410.3916***, 2014.

[73] Wikibooks. Artificial neural networks/activation functions - Wikibooks, open books for an open world. URL `https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Activation_Functions`.

[74] Wikipedia. Apophenia, 2015. URL `https://en.wikipedia.org/w/index.php?title=Apophenia&oldid=694943277`. Page Version ID: 694943277.

[75] Wikipedia. Neural coding, 2015. URL `https://en.wikipedia.org/w/index.php?title=Neural_coding&oldid=696917988`. Page Version ID: 696917988.

[76] Eliezer Yudkowsky. Coherent extrapolated volition. ***The Singularity Institute***, 2004.

[77] Eliezer Yudkowsky. Artificial intelligence as a positive and negative factor in global risk. ***Global catastrophic risks***, 1:308, 2008.

[78] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. ***CoRR***, abs/1409.2329, 2014.