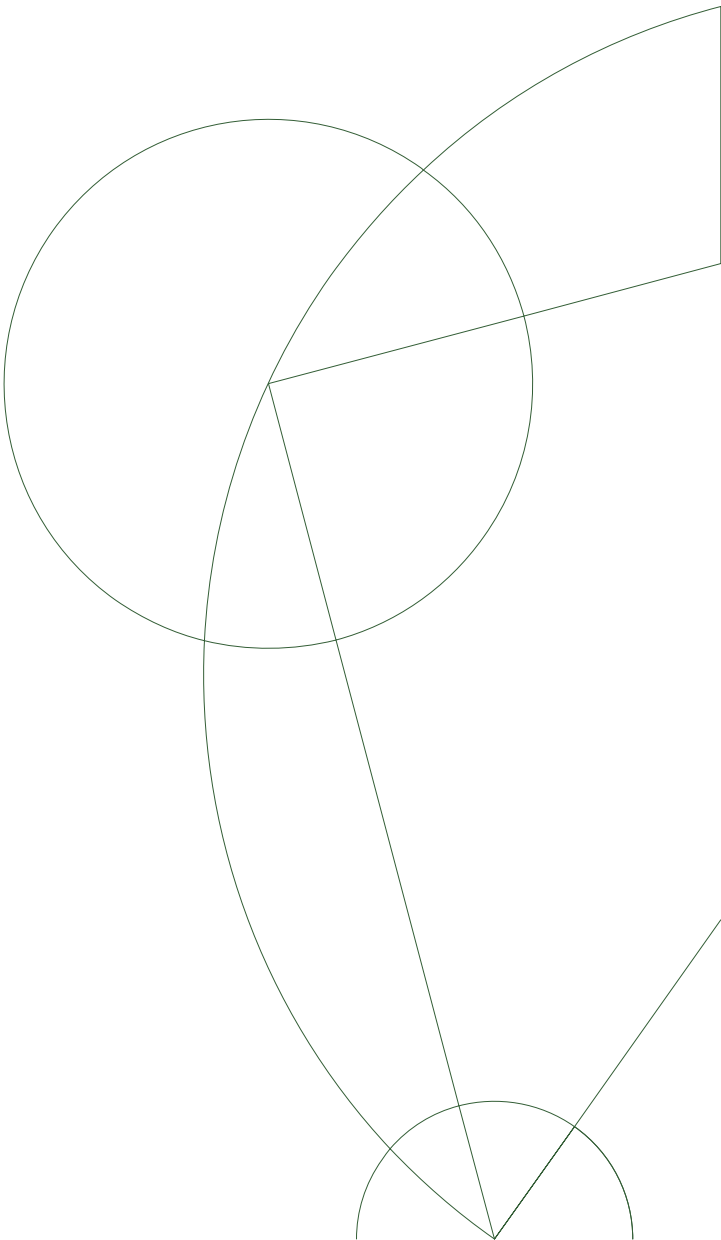
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Sune Andreas Dybro Debel

Deep Multi-Task Learning

For Relation Extraction



Dirk Hovy

July 31, 2017

Abstract

In this thesis we investigate the usefulness of a multi-task convolutional neural network architecture for relation classification. We review the relevant theoretical and practical research literature for supervised machine learning, convolutional neural networks, deep multi-task learning and relation classification. We test a state-of-the-art multi-task neural network architecture designed for relation classification on the SemEval 2010 Task 8 dataset, and investigate the sample complexity dynamics of learning this task simultaneously with other natural language processing tasks. We find that the convolutional architecture is ill-suited for this purpose. We discuss alternative approaches and make recommendations for further experimentation in this direction.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2

2 Background **4**
- 2.1 Information Extraction . . . 4
  - 2.1.1 Named Entity Recognition . . . 4
  - 2.1.2 Relation Extraction . . . 5
  - 2.1.3 Accuracy Measures . . . 6
- 2.2 Supervised Machine Learning . . . 7
  - 2.2.1 The Supervised Learning Problem . . . 7
  - 2.2.2 Statistical Learning Theory . . . 9
  - 2.2.3 Validation . . . 11
- 2.3 Summary . . . 12

3 Neural Networks **13**
- 3.1 Feed-Forward Neural Networks . . . 13
  - 3.1.1 Activation Functions . . . 15
  - 3.1.2 Objective Function . . . 17
- 3.2 Learning Algorithm . . . 19
  - 3.2.1 Gradient Descent . . . 19
  - 3.2.2 Adam . . . 20
  - 3.2.3 Backpropagation . . . 22
  - 3.2.4 Regularization . . . 25
- 3.3 Convolutional Neural Networks . . . 26
- 3.4 Word Vectors . . . 29

4 Multi-Task Learning **31**
- 4.1 Multi-Task and Single-Task Learning . . . 31
- 4.2 Bias Learning . . . 32
- 4.3 Representation Learning . . . 33
- 4.4 Task Relatedness . . . 34
- 4.5 Deep Multi-Task Learning . . . 36

5 Experiment **39**
- 5.1 Related Work . . . 39
- 5.2 Target Task . . . 40
- 5.3 Auxiliary Tasks . . . 42

5.3.1 ACE 2005 Relations . . . 43
5.3.2 CONLL2000 Part-of-Speech . . . 44
5.3.3 CONLL2000 Chunking . . . 45
5.3.4 GMB Named Entity Recognition . . . 46
5.4 Neural Network Architecture . . . 46
5.5 Algorithm . . . 51

6 Results **53**
6.1 Learning Surfaces . . . 53
6.2 Hypothesis Testing . . . 56

7 Discussion **58**
7.1 Impact of Limited Weight Sharing . . . 58
7.2 Why is ACE not a Useful Auxiliary task? . . . 60
7.3 Are the Sequence Classification Tasks Useful? . . . 60

8 Perspectives **61**
8.1 Alternative Neural Network Architectures . . . 61
8.1.1 Convolutional Neural Network with Argument Markers . . . 61
8.1.2 Multi-Channel Convolutional Neural Network . . . 62
8.1.3 Recurrent Neural Network . . . 62
8.2 Alternative Auxiliary Tasks . . . 63
8.2.1 Semantic Role Labeling . . . 63
8.2.2 Dependency Parsing . . . 64
8.3 Pipelining Vs. Multi-Task Learning . . . 64

9 Conclusion **65**

Part 1

Introduction

1.1 Motivation

The volume of digital text that exists in the world is rapidly increasing: The number of active science journals is growing with an annual rate of approximately 2.5%. The total number of published articles in these journals was approximately 2.5 million in 2015, up from 1.8 in 2009 (Mabe and Ware, 2009; Ware and Mabe, 2015). Similar staggering growth rates can be cited for content produced by online newspapers, social media and digital documents generated by businesses.

Finding the right information at the right time in the face of this data volume is challenging. A significant reason is that information contained in digital text is in the form of natural language which is often cited as being **unstructured** (despite the fact that natural language is actually highly structured). Unstructured data can be understood as essentially the opposite of the kind of data we find in relational databases where every data item is associated with metadata such as column names and column types. This metadata makes the structured data easy to search and analyze with computers. In contrast, unstructured data such as text, images, sound and video is not.

This problem has driven the development of so called **information extraction** techniques. The goal of these is to assign metadata to unstructured data, thereby giving some structure to it. This is an ambitious goal since in many cases it involves developing computer systems that perform tasks such as recognizing objects in images, converting speech to text or building data structures that capture the semantics of natural language, tasks we don't fully understand how humans are able to perform.

Relation extraction is an information extraction task the goal of which is to automatically identify a fixed set of semantic relationships of interest such as *Father-Son(Luke Skywalker, Darth Vader)* when they occur in natural language. Identifying these relations can significantly improve the quality of applications such as information retrieval or question answering systems.

Like most other natural language processing problems, relation extraction is extremely challenging because of the high degree of variation and ambiguity of natural language. Relation extraction systems are therefore usually composed of sub-

systems that each solve a sub-problem. Specifically, most relation extraction systems proceed by first identifying the potential arguments, such as *Luke Skywalker* and *Darth Vader* in our example, for any relation of interest. The system then detects whether or not a relation does in fact exist between them and finally classifies the relation if it exists.

Supervised machine learning in general, and **deep learning** in particular, are very successful techniques for solving information extraction problems. These approaches are based on the idea of supplying examples of inputs such as text, and corresponding correct outputs, so called labels, that we would like the system to reproduce given the input. The goal is to teach the system to give approximately correct output for new inputs that it hasn't seen before.

The conditions under which supervised machine learning systems can be expected to give approximately correct answers are fairly well understood. Theoretical analysis shows that the number of training examples provided for the learning system is one of the main ingredients in this guarantee. Producing these examples can be quite expensive however. It often requires a human annotator with specialized skills to provide the correct labels. This means there is significant motivation for reusing labeled training data to reduce the need to create large new collections of data for each new supervised machine learning problem.

Multi-task learning is a technique for reusing labeled data. In essence, it relies on the idea that it may be easier to learn related tasks simultaneously than in isolation, or in other words, that a learning system that learns from previously annotated data may require fewer examples for learning a new task, thereby reducing the cost of annotating new data.

1.2 Problem Statement

In this thesis we investigate multi-task learning for relation extraction. We focus on the relation classification step using deep learning techniques. Our goal is to investigate if and how learning an auxiliary task can benefit a target relation classification task that is learnt simultaneously, when the data available for the target task is limited. Specifically:

1. We survey the relevant research literature in order to formally answer questions such as *when is multi-task learning beneficial? How can we evaluate the usefulness of an auxiliary task?*
2. We implement a deep multi-task learning relation classification system based on our research.
3. We apply our theoretical understanding by analyzing our deep multi-task learning system for relation classification in order to make predictions about its performance.
4. We empirically test our predictions about the system's performance using appropriate accuracy measures.

In the following sections we detail the necessary background information necessary for understanding the relation extraction problem as well as our experiment with deep multi-task learning.

Part 2

Background

In this part, we describe the information extraction problem and the challenges it poses. Moreover, we formally describe the supervised machine learning setting. Specifically, we discuss the challenges of noise and overfitting, and show the usefulness of Vapnik-Chervonenkis analysis. We also cover validation techniques for supervised machine learning systems and appropriate accuracy measures for evaluating information extraction systems.

2.1 Information Extraction

In natural language processing, information extraction is the problem of extracting structured information from unstructured text. Many practical information extraction problems fall in one of two categories: **named entity recognition**, or **relation extraction** (Jurafsky and Martin, 2009). Here, we introduce each of them, and explain the challenges they pose.

2.1.1 Named Entity Recognition

A named entity is roughly anything that has a proper name. The goal of named entity recognition is to label mentions of entities such as people, organizations or places occurring in natural language. The list of things these systems are tasked with recognizing is often extended to include things that aren't technically named entities such as amounts of money or calendar dates.

As an example, consider the sentence:

Jim bought 300 shares of Acme Corp. in 2006.

A named entity recognition system designed to extract the entities *person* and *organization* should ideally assign the labels:

$$[\text{Jim}]_{person} \text{ bought 300 shares of } [\text{Acme Corp.}]_{organization} \text{ in 2006.}$$

This is a difficult problem because of two types of ambiguity. Firstly, two distinct entities may share the same name and category, such as *Francis Bacon* the painter and *Francis Bacon* the philosopher. Secondly, two distinct entities can have the same name, but belong to different categories such as *JFK* the former American president

Jim	bought	300	shares	of	Acme	Corp	.	in	2006	.
B-PER	O	O	O	O	B-ORG	I-ORG	I-ORG	O	O	O

Figure 2.1
A sentence labeled with BIO labels for named entity recognition.

and *JFK* the airport near New York.

Named entity recognition can be framed as a sequence labeling problem. A common approach is to apply so called tokenization to the text, i.e finding boundaries between words and punctuation, and associate each token with a label indicating which entity it belongs to. BIO (figure 2.1) is a widely used labelling scheme in which token labels indicate whether the token is at the **B**eginning, **I**nside, or **O**utside an entity mention.

2.1.2 Relation Extraction

The goal of relation extraction is to identify relationships such as *Family* or *Employment* in natural language. Most often, the relations of interest are limited to relations between named entities.

As an example, consider the sentence:

Yesterday, New York based Foo Inc. announced their acquisition of Bar Corp.

Imagine we have designed a relation extraction system that recognizes the relation *MergerBetween(organization, organization)* between two mentions of organizations. Ideally, we would like that system to extract the relation *MergerBetween(Foo Inc., Bar Corp.)* from the above sentence.

To simplify the relation extraction problem, it's often solved in three steps:

1. **Named entity recognition** Identify the named entities in the input text.
2. **Relation detection** For each pair of named entities in the input text, determine if a relation exists between them. This is a binary classification problem where the input is the text and the named entities detected in step 1, and the output is yes/no.
3. **Relation classification** Classify each of the detected relations in the previous step. This a multi-label classification problem where the input is the input text and the named entities for which a relation was detected in step 2, and the output is a relation label.

In this thesis we focus on step 3, assigning labels to detected relations. This is a difficult problem because of ambiguity. As an example, consider the sentence *Susan left JFK*. Imagine that we want to design a relation extraction system that can detect the relations *Physical(person, location): a person has a physical relation to a location* and *Personal-Social(person, person): two persons have a social relation*. Both can reasonably be assigned the previous sentence, depending on whether *JFK* refers to the airport near New York, or the former American president.

Relation extraction is often framed as a sentence classification problem, where the input to the relation extraction system is a sentence, and the output is a relation present in that sentence, if any.

2.1.3 Accuracy Measures

Information extraction systems are often evaluated empirically by applying them to collections of text, so called corpora, in which N mentions of named entities or relations are known. In these tests, accuracy measures for each class c of information we wish to extract are usually defined in terms of how many times the system predicted class c versus how many times c actually occurs in the corpus. Most metrics use the following terminology:

	predicted as c	**predicted as not** c
c	True positives (tp)	False negatives (fn)
not c	False positives (fp)	True negatives (tn)

Where for example tp is the number of true positives produced for class c.

The distribution of labels used in both named entity recognition and relation extraction is often highly imbalanced. Consider for example the BIO labelling scheme for named entity recognition in figure 2.1. Most words will be outside a mention of a named entity, and will have the label O. Using simple accuracy $\frac{tp+tn}{tp+tn+fn+fp}$ as a performance metric for a system that outputs bio labels for each token in the text is therefore not very informative, since a useless system which labels all tokens with O would achieve high performance.

Precision and **recall** are more appropriate performance metrics for this reason. Precision $\frac{tp}{tp+fp}$ is the fraction of correct information items extracted by the system. This is equal to 0 when none of the information extracted by the system was correct and 1 when all of it was correct.

Recall $\frac{tp}{tp+fn}$ is the fraction of information items in the corpora that the system correctly extracted. This is 0 when none of the extracted information was correct, and 1 when all of the extracted information was correct, and no item was incorrectly labeled by the system.

In a multi-class classification problem we are forced to decide how to average these metrics across classes. Specifically, there are two ways of averaging an accuracy measure across C classes: micro and macro averaging (Sokolova and Lapalme, 2009). In macro averaging, an accuracy measure is computed for each class c separately, and then averaged across all C classes. For example macro-precision p_M:

$$p_M = \frac{1}{C}\sum_{c=1}^{C} p_c$$

Where p_c is the precision of the system for class c. Micro averaging on the other hand, averages an accuracy by accumulating tp, tn, fp and fn across all C classes.

For example micro-precision p_μ:

$$p_\mu = \frac{\sum_{c=1}^{C} tp_c}{\sum_{c=1}^{C} tp_c + fp_c}$$

Where for example tp_c is the true positives a system produces for class c.

The main difference between macro and micro averages of accuracy measures is that micro averaging gives more weight to more frequent classes. In other words, micro averaging encodes the bias that infrequent classes are unimportant, and a misclassification of an example of such a class should not penalise the accuracy measure as much as a misclassification of a more frequent class. Whether on not this a reasonable bias depends on the problem.

To get a single number that summarizes the performance, precision p and recall r are often combined into a single metric, the $F1$ measure defined as the harmonic mean of precision and recall $\frac{2pr}{p+r}$. Variations that use the micro and macro versions of precision and recall can naturally be computed as the harmonic mean of the micro or macro precision and recall respectively.

2.2 Supervised Machine Learning

Most modern solutions to the information extraction problems in 2.1 are based on supervised machine learning techniques. In this setting a system learns to recognize the named entities or relations between them from examples provided by a human annotator. In this section we formally describe this approach and introduce important theoretical tools for understanding supervised machine learning.

2.2.1 The Supervised Learning Problem

A training set $\mathcal{D}$ of N examples $(\mathbf{x}_i, \mathbf{y}_i)$ of inputs $\mathbf{x}_i$ and corresponding labels $\mathbf{y}_i$ is created by a human annotator. Each $\mathbf{x}_i$ belongs to an input space $\mathcal{X}$, for example the set of all english sentences. Each $\mathbf{y}_i$ belongs to an output space $\mathcal{Y}$ of labels, for example the set of all sequences of BIO tags. As designers of the learning system we specify a set of functions $h : \mathcal{X} \mapsto \mathcal{Y}$, the so called **hypothesis space** $\mathcal{H}$. We want to find a function $h \in \mathcal{H}$, sometimes called a **model** or **hypothesis**, that can automatically assign labels to a new set of un-labeled inputs $\mathcal{D}_{test} = \{\mathbf{x}_i \mid \mathbf{x}_i \in \mathcal{X}\}$ at some point in the future.

Supervised machine learning is the science of how to use an algorithm to find a function h using $\mathcal{D}$ that performs well on $\mathcal{D}_{test}$, as measured by some performance measure e. In classification problems such as named entity recognition or relation extraction where $\mathcal{Y}$ is discrete, we typically use binary error $e(\mathbf{y}_1, \mathbf{y_2}) = \mathbb{I}[\mathbf{y}_1 \neq \mathbf{y}_2]$. Importantly, we are only interested in the performance of h on $\mathcal{D}$ to the extent that it's informative of how the system will perform on future data (Abu-Mostafa et al., 2012).

We can formalize the preference for functions h that perform well on examples outside of the training set with a quantity known as **generalization error**.

Definition 2.2.1 (generalization error)**.** Let $P(\mathbf{x}, \mathbf{y})$ be a joint probability distribution over inputs $\mathbf{x} \in \mathcal{X}$ and labels $\mathbf{y} \in \mathcal{Y}$. Let $e(\mathbf{y_1}, \mathbf{y_2})$ be an error function that measures agreement between labels $\mathbf{y}_1$ and $\mathbf{y}_2$. Then the generalization error E of a function $h : \mathcal{X} \mapsto \mathcal{Y}$ is defined as:

$$E(h) = \mathbb{E}_{\mathbf{x},\mathbf{y} \sim P(\mathbf{x},\mathbf{y})}[e(h(\mathbf{x}), \mathbf{y})]$$

Formally, the objective of supervised machine learning is to find a function h^* in a space of functions $\mathcal{H}$ that minimizes $E(h)$. We see the process generating the data as random, but with a behavior describable by a distribution $P(\mathbf{x}, \mathbf{y})$. Unfortunately, this distribution is unknown, which makes E unknown. However, we can use sampled data $\mathcal{S} = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{x}, \mathbf{y} \sim P(\mathbf{x}, \mathbf{y})\}$ to estimate $E(h)$ with a quantity known as **empirical error**:

Definition 2.2.2 (empirical error)**.** Let $\mathcal{S}$ be a set of N examples $\{(\mathbf{x}_i, \mathbf{y}_i) \mid \mathbf{x}_i, \mathbf{y}_i \sim P(\mathbf{x}, \mathbf{y})\}$. Then the empirical error $\hat{E}$ is defined as:

$$\hat{E}(h, \mathcal{S}) = \frac{1}{N} \sum_{i=1}^{N} e(h(\mathbf{x}_i), \mathbf{y}_i)$$

Because $\mathcal{S}$ is a random quantity, it's dangerous to use $\hat{E}$ to estimate E. We risk that the samples are not representative of $P(\mathbf{x}, \mathbf{y})$, leading us to believe that h is great, when in fact it's terrible. We can bound the probability that $\hat{E}$ is a bad estimate of E if we make two assumptions:

Firstly, we assume that the samples in $\mathcal{S}$ are drawn independently from $P(\mathbf{x}, \mathbf{y})$, that is observing any one sample did not change the probability of observing any other sample.

Secondly, we assume that h is independent of $\mathcal{S}$, in other words, that h was not specifically chosen based on the sample. These assumptions enable us to apply **Hoeffding's inequality** to bound the probability that $\hat{E}$ is far away from E:

Theorem 2.2.1 (Hoeffding's inequality)**.** let $E(h)$ be defined as in definition 2.2.1, and let $E(h, \mathcal{S})$ be defined as in definition 2.2.2. Then:

$$\mathbb{P}\left(|E(h) - \hat{E}(h, \mathcal{S})| \geq \epsilon\right) \leq 2e^{-2N\epsilon^2}$$

The inequality tells us that the probability that E is more than ϵ away from $\hat{E}$ decreases exponentially in ϵ and N. In other words, the more samples in $\mathcal{S}$, the less likely it is that E will be misleading. Estimating E with a sample that's independent of h is a technique called **validation** and will be discussed in section 2.2.3

Because $\mathcal{D}$ is used to select h, it cannot be used to estimate E by Hoeffding's inequality, and we need more sophisticated techniques to understand the relationship between $\mathcal{D}$ and E. The central question in supervised machine learning is *how can we best define $\mathcal{H}$ and use $\mathcal{D}$ to make E small?* Answering this question is the objective of a field of research known as **statistical learning theory**.

2.2.2 Statistical Learning Theory

We would like to know how best to define $\mathcal{H}$ and use $\mathcal{D}$ in order to make E small. $\mathcal{D}$ is the only information we have about $P(\mathbf{x}, \mathbf{y})$, and therefore also the only information we have about E. A straight-forward idea would be to find a function $g \in \mathcal{H}$ that minimizes the **training error** $\hat{E}(h, \mathcal{D})$ in the hope that g will also minimize E.

As we argued in section 2.2, using $\hat{E}$ to estimate E can be misleading. Moreover, because D is used to specifically choose g that makes $\hat{E}$ small, the guarantees provided by Hoeffding's inequality no longer holds, and therefore it may be possible to select g such that $\hat{E}(g, \mathcal{D})$ is small and $E(g)$ is large, even when we have a large number of training examples.

The phenomena where training error is small but generalization error is large is known as **overfitting**. As the name implies, it's caused by harmful idiosyncrasies of $\mathcal{D}$ that, when used to minimize $\hat{E}(h, \mathcal{D})$, leads us to a g with a larger E than other functions in $\mathcal{H}$. These idiosyncrasies of $\mathcal{D}$ are ultimately the product of **noise**.

In general, noise comes in two forms. The first form is known as **stochastic noise**. This type of noise is introduced by variation in the relationship between $\mathbf{x}$ and $\mathbf{y}$ that is irrelevant to the problem we are trying to solve. For example, human error is a common source of stochastic noise in information extraction where an annotator incorrectly labels a piece of text. Selecting a g that repeats this error is a case of overfitting, because g will have lower training error but larger generalization error than another h that doesn't predict the incorrect annotation, since presumably the error is the exception to the rule.

The second type of noise is called **deterministic noise**. This type of noise may be introduced when the relationship between $\mathbf{x}$ and $\mathbf{y}$ is deterministic, but $\mathcal{H}$ doesn't have the capacity to represent this relationship exactly.

To understand deterministic noise, imagine that even h^* can't represent the deterministic relationship $\mathbf{y} = f(\mathbf{x})$ exactly. Suppose that we get a $\mathcal{D}$ that contains a sample $(\mathbf{x}_i, \mathbf{y}_i)$ that falls outside the capacity of h^*, that is, $h^*(\mathbf{x}_i) \neq \mathbf{y}_i$. Now further imagine that in order to minimize $\hat{E}$, we select a g that predicts this sample, such that $g(\mathbf{x}_i) = \mathbf{y}_i$. This is a case of overfitting since we know that there is at least one function in $\mathcal{H}$ with lower generalization error than g, namely h^*.

The risk of overfitting is linked to the diversity of $\mathcal{H}$. When we say that $\mathcal{H}$ is diverse, we roughly mean that the functions $h \in \mathcal{H}$ are very different from each other. The more diverse $\mathcal{H}$ is, the greater the risk that there exists a $h \in \mathcal{H}$ that will overfit $\mathcal{D}$.

A **dichotomy** is a central concept in measuring the diversity of $\mathcal{H}$. A dichotomy is a specific sequence of N labels. For simplicity, most theoretical analyses of $\mathcal{H}$ assume a binary output space $\mathcal{Y} = \{0, 1\}$ and we will too. In that case, if $N = 3$ then (0 1 0) is a dichotomy and so is (1 0 0). We have listed all dichotomies for $N = 3$ in figure 2.2.

Dichotomies allow us to group similar functions. By simple combinatorics the number of dichotomies for N must be smaller than or equal to 2^N if $\mathcal{Y}$ is binary. There may be infinitely many functions in $\mathcal{H}$, but on a specific $\mathcal{D}$, many of them will produce the same dichotomy since the number of training examples in $\mathcal{D}$ is finite. This allows us to quantify the diversity of $\mathcal{H}$ in terms of the number of dichotomies

(0 0 0)
(1 0 0)
(0 1 0)
(0 0 1)
(1 1 0)
(0 1 1)
(1 0 1)
(1 1 1)

Figure 2.2
All dichotomies for $\mathcal{Y} = \{0, 1\}$ and $N = 3$. There are $2^3 = 8$ ways to choose a sequence of 3 labels from 2 possibilities.

it's able to realize on a set of N points. This is achieved by a measure known as the **growth function**.

Definition 2.2.3 (growth function)**.** Let $\mathcal{H}(N) = \{(h(\mathbf{x}_1), \ldots, h(\mathbf{x}_N)) \mid h \in \mathcal{H}, \mathbf{x}_i \in \mathcal{X}\}$ be the set of all dichotomies generated by $\mathcal{H}$ on N points, and let $|\cdot|$ be the set cardinality function. Then the growth function m is:

$$m(N, \mathcal{H}) = \max |\mathcal{H}(N)|$$

In words, the growth function measures the maximum number of dichotomies that are realizable by $\mathcal{H}$ on N points. To compute $m(N, \mathcal{H})$, we consider any choice of N points from the whole input space $\mathcal{X}$, select the set that realizes the most dichotomies and count them.

The growth function allows us to account for redundancy in $\mathcal{H}$. If two functions $h_i \in \mathcal{H}$ and $h_j \in \mathcal{H}$ realise the same dichotomy on $\mathcal{D}$, then any statement based only on $\mathcal{D}$ will be either true or false for for both h_i and h_j. This makes it possible to group the events $\hat{E}(h_i, \mathcal{D})$ *is far away from* $E(h_i)$ and $\hat{E}(h_j, \mathcal{D})$ *is far away from* $E(h_j)$, and thereby avoiding to overestimate the probability of the union of both events occurring.

If $\mathcal{H}$ is infinite, the number of redundant functions in $\mathcal{H}$ will also be infinite since the number of dichotomies on N points is finite. If $m(N, \mathcal{H})$ is much smaller than 2^N, the number of redundant functions in $\mathcal{H}$ will be so large as to make the probability that $\hat{E}$ is far away from E very small.

This line of reasoning is the basis of the Vapnik-Chervonenkis bound, which bounds $E(h)$ in terms of $\hat{E}(h, \mathcal{D})$:

Theorem 2.2.2 (Vapnik-Chervonenkis bound)**.** Let $m(N, \mathcal{H})$ be defined as in definition 2.2.3, $E(h)$ as 2.2.1, and $\hat{E}(h, \mathcal{D})$ as in 2.2.2. Then, with probability $1 - \delta$:

$$E(h) \leq \hat{E}(h, \mathcal{D}) + \sqrt{\frac{8}{N} \ln \frac{4m(2N, \mathcal{H})}{\delta}}$$

The bound tells us that $E(h)$ will be close to $\hat{E}(h, \mathcal{D})$ if $m(N, \mathcal{H})$ is small and N is large. Intuitively, this tells us that a set $\mathcal{H}$ that contains "simple" functions will make

it easier to choose g such that generalization error will be close to training error, where simple means: functions that realize a small number of dichotomies.

On the other hand, having a set $\mathcal{H}$ that can realize a large number of dichotomies on N points, will make it easier to find a function that will make $\hat{E}(h, \mathcal{D})$ small. Using a $\mathcal{H}$ with functions that are too simple is called **underfitting**. It occurs when we search for a function in the set of functions $\mathcal{H}$, when there is another, more diverse set of functions $\mathcal{G}$ which contain a function with lower generalization error.

This analysis tells us that an optimally diverse $\mathcal{H}$ balances the tradeoff between the risk of overfitting, represented in the bound by m, and the risk of underfitting, represented by $\hat{E}$. In practice, underfitting is less of a problem than overfitting, since modern supervised machine learning algorithms search in extremely diverse spaces of functions $\mathcal{H}$. In fact, most $\mathcal{H}$ are so diverse that steps must be taken to avoid using all of $\mathcal{H}$ when learning from it. These techniques are known as **regularization**, which we will see an instance of in section 3.2.4.

A simple rewrite of theorem 2.2.2 leads to a very popular equivalent formulation: a **sample complexity** bound. Sample complexity denotes the number of training examples required for a certain generalization performance. Exercising a bit of algebra on the Vapnik-Chervonenkis bound leads to the insight that in order for E to be no more than ϵ away from $\hat{E}$ with probability $1 - \delta$, it requires:

$$N \geq \frac{1}{\epsilon^2} \ln \frac{4m(2N, \mathcal{H})}{\delta}$$

We will see several statements of this type throughout this thesis, since the sample complexity when learning multiple tasks is precisely our main concern.

2.2.3 Validation

Statistical learning theory tells us how to design $\mathcal{H}$ given a dataset by revealing the relationship between $\hat{E}(\mathcal{D}, h)$ and $E(h)$. While Vapnik-Chervonenkis analysis gives us a theoretical bound on $E(h)$, we may be interested in getting a concrete empirical estimate of E, for example in order to decide whether a system is good enough to be put in to production.

In general, $\mathcal{D}$ is unsuited for this estimation because of **bias**: we use $\mathcal{D}$ to specifically select g to minimize $\hat{E}(\mathcal{D}, h)$, and so the performance of g on $\mathcal{D}$ is likely an optimistic estimate of E.

In order to get an unbiased empirical estimate of E we can split $\mathcal{D}$ into two datasets: $\mathcal{D}_{val}$ containing V samples and $\mathcal{D}_{train}$ containing $N - V$ samples. $\mathcal{D}_{train}$ is used by the learning system to find a function $g^- \in \mathcal{H}$. The minus superscript indicates that the function was selected using only a subset of $\mathcal{D}$. $\mathcal{D}_{val}$ is used to compute $\hat{E}(g^-, \mathcal{D}_{val})$ as an unbiased estimate of E.

We can use Vapnik-Chervonenkis analysis to bound the error of $\hat{E}(g^-, \mathcal{D}_{val})$ as an estimate of $E(g^-)$. We can view $\mathcal{D}_{val}$ as a training set, which we use to search a hypothesis space containing just g^-. This leads to:

$$E(g^-) \leq \hat{E}(g^-, \mathcal{D}_{val}) + O(\frac{1}{\sqrt{V}})$$

The inequality tells us that V should be large in order for $\hat{E}(g^-, \mathcal{D}_{val})$ to be close to $E(g^-)$. This presents a problem since increasing the size of V decreases the number of examples available for training. Though hard to prove theoretically, it's empirically well documented that more training data lead to lower generalization error (Abu-Mostafa et al., 2012). In other words, making V large will lead to a very accurate estimate of a very poor hypothesis.

Cross validation is a technique that may be used to overcome this dilemma. In this setting, $\mathcal{D}$ is split into K parts called **folds** each containing $\frac{N}{K}$ samples. $\mathcal{D}_{train}$ is then composed of $K-1$ folds, and the remaining fold is used as $\mathcal{D}_{val}$. This leads to K iterations of the learning procedure, yielding K hypotheses g_k^-, and K estimates of generalization error $e_k = \hat{E}(g_k^-, \mathcal{D}_{val})$. We can then define the cross-validation error as the average of these estimates:

$$E_{cv} = \frac{1}{K} \sum_{k=1}^{K} e_k$$

We want to know E, but it would be almost as useful to know the expected E of our learning system when trained on any dataset $\mathcal{D}$ of size N. For this purpose, we can define

$$\tilde{E}(N) = \mathbb{E}_{\mathcal{D}}[E(g)]$$

In words, the expected generalization error with respect to datasets of size N. The expected value of E_{cv} is $\tilde{E}(N - \frac{N}{K})$. To see why, consider the expected value of a single estimate e_k:

$$\mathbb{E}[e_k] = \mathbb{E}_{\mathcal{D}_{train}} \mathbb{E}_{\mathcal{D}_{val}}[\hat{E}(g_k^-, \mathcal{D}_{val})] = \mathbb{E}_{\mathcal{D}_{train}}[E(g_k^-)] = \tilde{E}(N - \frac{N}{K})$$

Since the equality holds for a single estimate, it also holds for the average E_{cv} (Abu-Mostafa et al., 2012).

In words, the cross validation error estimates the expected generalization error of the learning system when trained on $N - \frac{N}{K}$ samples. We can control the number of samples available for finding each function g_k^- by increasing K, at no cost of estimation accuracy. Increasing K increases computation time however, since we have to search the hypothesis space K times.

2.3 Summary

In this section we have seen that the purpose of named entity recognition is to identify mentions of entities such as people, organizations and places in natural language. The purpose of relation extraction systems is to identify relationships between them.

We have seen that simple accuracy is uninformative as an evaluation measure in information extraction, and described precision and recall as alternative metrics.

We have described the formal setting of supervised machine learning. We have discussed concepts such as overfitting and noise, diversity of the set of functions $\mathcal{H}$ from which to choose h, and its impact on training and generalization error. Finally we have discussed validation as a technique for estimating E empirically.

Part 3

Neural Networks

In this part we describe how to define $\mathcal{H}$ using functions called **neural networks** which have the advantage of being easy to adapt to multi-task learning. We begin by describing how to design a $\mathcal{H}$ with neural networks. We then turn to the issue of how to use $\mathcal{D}$ to search this hypothesis space. Lastly, we introduce a specialized neural network often used for text classification problems such as relation extraction.

3.1 Feed-Forward Neural Networks

A feed-forward neural network is a function $h : \mathcal{X} \mapsto \mathcal{Y}$. To understand how it works, it's instructive to look at each part of its name in isolation.

h is called a **network** because it's a composition of L **layers** of other functions $f^{(l)}$. Each $f^{(l)}$ receives input from $f^{(l-1)}$. For example if $L = 2$, then $h(\mathbf{x}) = f^{(2)}(f^{(1)}(\mathbf{x}))$. We denote the input to $f^{(1)}$ as $\mathbf{x}^{(0)}$, which is identical to the input vector $\mathbf{x}$ with an added **bias** component of 1, as described later in this section. each $f^{(l)}$ outputs a vector $\mathbf{x}^{(l)}$ of dimension $d^{(l)}$. The dimensionality of these vectors determine the **width** of the network. The number of layers L is called the **depth** of the network. $f^{(L)}$ is called the **output layer**. The remaining functions $f^{(1)}$ to $f^{(L-1)}$ are called **hidden layers**.

The functions $f^{(1)}$ to $f^{(L)}$ are ordered by their index l such that the index of the earliest layers are smaller than the index of the later layers. h is called a **feed-forward** network because each $f^{(l)}$ can receive input only from functions $f^{(i)}$ if $l > i$. In other words, it's not possible for a function $f^{(l)}$ to feed its own output into itself, or any other function that it receives input from.

Finally h is called a **neural** network since its design is loosely based on neurons in the brain (Goodfellow et al., 2016). Each component x_i of the vector $\mathbf{x}^{(l)}$ can be seen as the output of a unit similar to a neuron. Each unit in layer l receives input from units in layer $l-1$. The output $x_i^{(l-1)}$ of unit i in layer $l-1$ is multiplied by a weight $w_{ij}^{(l)}$ that gives the strength of the connection between unit i in $l-1$ and unit j in l. Unit j sums all of the input it receives from units in layer $l-1$ to obtain its **activation**

$a_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$. To compute its output $x_j^{(l)}$, it applies an **activation function** $\sigma(a_j^{(l)})$ to its activation.

Activation functions model the behavior of biological neurons by outputting a signal only when the activation is above a certain threshold. To make it possible to learn this threshold for each unit using the same activation function, we introduce a special **bias** unit that always outputs 1. The index of the bias unit in layer l is 0 by convention. Figure 3.1. shows how a unit j computes its output $x_j^{(l)}$ by combining the outputs of units in layer $l-1$.

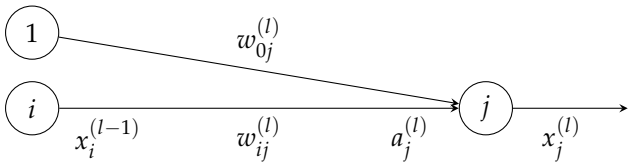


Figure 3.1
A visual representation of the connections between unit i in layer $l-1$, the bias unit in $l-1$, and unit j in layer l. The connection strength between these units is given by the weight $w_{ij}^{(l)}$ between i and j, and $w_{0j}^{(l)}$ between the bias unit and j. The activation $a_j^{(l)}$ at unit j is computed by $a_j^{(l)} = w_{ij}^{(l)} x_i^{(l)} + w_0^{(l)}$. The output $x_j^{(l)}$ of unit j is given by $x_j^{(l)} = \sigma(a_j^{(l)})$

Keeping track of the indices l, i and j quickly becomes confusing. By collecting all of the weights of connections going into unit j in layer l in a vector $\mathbf{w}_j^{(l)}$, the activation at unit j can be computed as a dot product $a_j^{(l)} = \mathbf{w}_j^{(l)} \cdot \mathbf{x}^{(l-1)}$. Moreover, we can compute the entire vector $\mathbf{a}^{(l)}$ of activations at layer l, by organising the weight vectors $\mathbf{w}_j^{(l)}$ in a matrix $\mathbf{W}^{(l)} = \begin{bmatrix} \mathbf{w}_1^{(l)} & \dots & \mathbf{w}_{d^{(l)}}^{(l)} \end{bmatrix}^T$, which leads to $\mathbf{a}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$.

By gathering the weights in matrices $\mathbf{W}^{(l)}$, we have simplified our view of h into a composition of matrix-vector products and element-wise application of activation functions. Figure 3.2 shows the parallel views of neural networks as networks of units and matrix-vector operations.

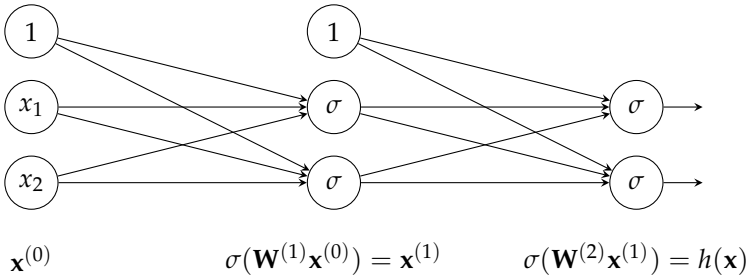


Figure 3.2
A visual representation of $h = f_2(f_1(\mathbf{x}^{(0)}))$. The activation at each layer $\mathbf{a}^{(l)}$ is computed by $\mathbf{W}^{(l)}\mathbf{x}^{(l-1)}$. The output at each layer is computed by element-wise application of the activation function of $\sigma(\mathbf{a}^{(l)})$.

We now have all the components we need to specify $\mathcal{H}$ as a set of neural networks. The set is defined by the depth of the networks L, the number of units in each layer d_l and the activation function σ. For a particular L, d_l, and σ, each $h \in \mathcal{H}$ corresponds exactly to a unique assignment of real numbers to all of its weights. We can make the dependence of h on its weights explicit by defining a vector $\mathbf{w} = \begin{bmatrix} w_{ij}^{(1)} & \dots & w_{ij}^{(L)} \end{bmatrix}$ and writing $h(\mathbf{x}, \mathbf{w})$ which means *the function h parameterised by the weight vector* $\mathbf{w}$. In the next section we discuss how to choose the activation functions at the layers of the network.

3.1.1 Activation Functions

Activation functions mimic the behaviour of neurons in the brain. A neuron emits a signal when the combined input it receives from other neurons exceeds a certain threshold. Activation functions achieve this by a variation of the step function, where an activation signal $a_j^{(l)}$ below the threshold is mapped to a value near zero and an activation signal above the threshold is mapped to a value greater than zero. From a mathematical perspective the role of activation functions is to introduce non-linearity in h which allows $\mathcal{H}$ to model a larger class of functions.

Many networks use **sigmoid** activation functions such as the classical sigmoid function $\sigma(a) = \frac{1}{1+e^{-a}}$. These functions have the advantage of being differentiable everywhere. As we will see in section 3.2, differential calculus is the fundamental tool for finding a good $h \in \mathcal{H}$ which makes differentiability a desirable quality. One drawback of sigmoid activation functions is that their derivates are small, as seen in figure 3.3. For example, we can show that $\max \frac{d\sigma}{da} = \frac{1}{4}$. As we will see in section 3.2, neural networks are trained by multiplying chains of derivatives. When these derivatives are smaller than 1, the magnitude of the derivative shrinks in the length of the chain of terms which can make learning from $\mathcal{D}$ extremely slow.

Because of this shrinking problem, the default recommendation today is to use **rectified linear units**. These units use the activation function $\sigma(a) = \max(0, a)$ depicted in figure 3.4. This function has the advantage that its derivative $\frac{d\sigma}{da} = 1$ when $a > 0$, and $\frac{d\sigma}{da} = 0$ when $a < 0$. This activation function is not strictly differentiable when $a = 0$. In practice however, this is not a big problem because a is rarely exactly 0 and since neural networks are trained through an iterative process in which we can skip iterations where units have zero activation.

Often we would like the output of h to be a probability distribution over values in the label space $\mathcal{Y}$, since this makes it possible to design the learning algorithm with a principled technique called **maximum likelihood** in which the appropriateness of h is measured by the probability it assigns to the training data. For this reason it's common to use different activation functions in the output layer that enables us to interpret the output of h as a probability.

For example, named entity recognition can be seen as a multi-class classification problem, where each token in a sentence must be assigned one of a fixed set of C labels. To frame this as a probabilistic problem, we can encode each token label $\mathbf{y}$ as a vector of C probabilities such that component y_c of $\mathbf{y}_i \in \mathcal{D}$ is equal to 1 if $\mathbf{x}_i \in \mathcal{D}$

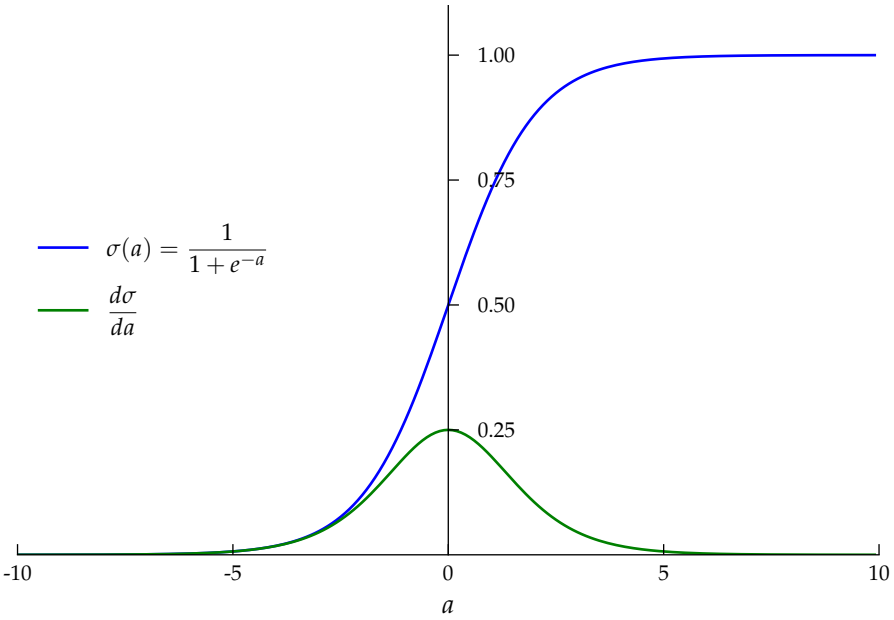


Figure 3.3
*Sigmoid activation and its derivate. Sigmoid activation units have the disadvantage of **saturating**, meaning that they become flat when a is large or small. This makes the derivative smaller than 1 everywhere, and much smaller than 1 almost everywhere.*

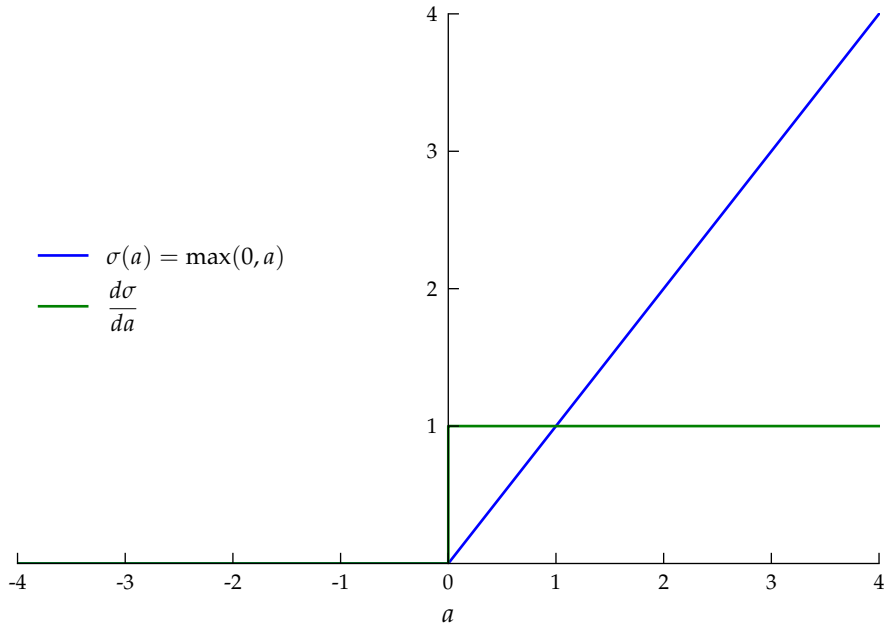


Figure 3.4
ReLU activation and its derivate. Unlike sigmoid activation, ReLU activation doesn't saturate. This means that the derivative of a unit remains large whenever it produces output.

belongs to class c. All other components $y_{j \neq c}$ in $\mathbf{y}_i$ are equal to 0. This is known as **one-hot** encoding. $\mathbf{y}$ can be seen as a conditional probability distribution over each possible label given $\mathbf{x}_i$, that places all of the probability mass on label c.

With one hot encoding, we can design h to output vector with C components, where each component $c \in h$ gives the probability that $\mathbf{x}$ has class c. More formally, we can interpret $h(\mathbf{x})$ as conditional probability distribution such that $h(\mathbf{x})_c = P(Y = c \mid X = \mathbf{x})$ where X and Y are random variables over $\mathcal{X}$ and $\mathcal{Y}$.

This type of output can be achieved by using the so-called **soft-max** activation function in the output layer of a neural network. The soft-max activation is given by

$$\sigma(\mathbf{a})_c = \frac{e^{\mathbf{a}_c}}{\sum_{i=1}^{C} e^{\mathbf{a}_i}}$$

Where the notation $\mathbf{a}_c$ denotes the c'th component of the vector $\mathbf{a}$. In words, the soft-max function makes sure that the output of h is a valid probability distribution by making sure that each component of $h(\mathbf{x})$ is positive by taking the exponent, and by making sure that $\sum_{c=1}^{C} h(\mathbf{x})_c = 1$ by dividing by the sum of all the exponentiated components. The last point means that unlike the other activation functions we have seen in this section, the soft-max must receive as input the vector $\mathbf{a}^{(L)}$ of all activations in layer L.

Having designed the output layer of h so that we can interpret its output as a conditional probability distribution, we can define the training error $\hat{E}(h, \mathcal{D})$, sometimes also called the **objective function** by the maximum likelihood principle, that quantifies the appropriateness of a weight vector $\mathbf{w}$ as a probability using the samples in $\mathcal{D}$. This function is crucial for finding $g \in \mathcal{H}$. We study it in the next section.

3.1.2 Objective Function

We would like a function that lets us compare functions in $\mathcal{H}$ in terms of how well they predict the samples in $\mathcal{D}$. Such a function is often called an objective function, borrowing terminology from the mathematical field of optimization.

In section 3.1.1 we saw that the combination of one-hot encoding of the labels in $\mathcal{Y}$ and soft-max activation in the output layer of h allows us to interpret $h(\mathbf{x})$ as a conditional probability distribution. In the following, we will use a convenient rewrite of the formula given in 3.1.1:

$$P(Y = y \mid X = \mathbf{x}) = \prod_{c=1}^{C} h(\mathbf{x}, \mathbf{w})_c^{\mathbf{y}_c}$$

Where y is the true label for $\mathbf{x}_i$ and $\mathbf{y}_c$ is component c of the one-hot vector $\mathbf{y}$. This formulation works because $\mathbf{y}$ is a one-hot vector, which means exactly one component of $\mathbf{y}$ is equal to 1, and all other components are equal to 0. So if $\mathbf{y} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$ and $h(\mathbf{x}) = \begin{bmatrix} .1 & .8 & .1 \end{bmatrix}^T$, then $P(Y = y \mid X = \mathbf{x}) = (0.1^0)(0.8^1)(0.1^0) = 0.8$.

If we design $\mathcal{H}$ in such a way that every h outputs a probability, we can use the

principle of maximum likelihood to derive a plausible objective function. Maximum likelihood estimation uses the likelihood function to compute the probability of $\mathcal{D}$ by interpreting h as a probability distribution parameterized by $\mathbf{w}$:

Definition 3.1.1 (likelihood function)**.** Let $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$ be a set of N training examples, where each $\mathbf{y}_i$ is a C dimensional one-hot vector. Let $h(\mathbf{x}, \mathbf{w})$ be a neural network which outputs conditional probability distributions over the C possible classes, such that $\sum_{c=1}^{C} h(\mathbf{x}, \mathbf{w})_c = 1$ and $0 \leq c \leq 1 \, \forall c \in h(\mathbf{x}, \mathbf{w})$. Furthermore, let the notation $\mathbf{y}_{ic}$ denote component c of the one-hot label for example i. Then the likelihood $P(\mathcal{D} \mid \mathbf{w})$ is:

$$P(\mathcal{D} \mid \mathbf{w}) = \prod_{i=1}^{N} \prod_{c=1}^{C} h(\mathbf{x}_i, \mathbf{w})_c^{\mathbf{y}_{ic}}$$

Informally, we can think of the likelihood function as asking the question: *assuming that $h(\mathbf{x})$ is the true conditional distribution from which $\mathcal{D}$ was sampled, what is the probability of observing the samples in $\mathcal{D}$?* Using the likelihood function to find a good $h \in \mathcal{H}$ is a matter of finding a weight vector $\mathbf{w}$ that maximize the likelihood of observing $\mathcal{D}$.

Computing a large number of products of probabilities on a computer can be problematic because of **numerical underflow**. Since computers have limited precision, small positive numbers may be actually be represented as small negative numbers, which is bad because the likelihood function is a probability.

To avoid numerical underflow, the **log-likelihood** $\ln P(\mathcal{D} \mid \mathbf{w})$ is often used instead. The logarithm turns the products into sums, which are entirely unproblematic for computers. Since the natural logarithm is a monotonic function, applying it to the likelihood function does not change the properties we are interested in.

Finally, most other objective functions for supervised machine learning are defined in terms of training error $\hat{E}(h, \mathcal{D})$. In this view, searching for a good $h \in \mathcal{H}$ becomes a minimization problem. For consistency, maximum likelihood estimation is often turned into a minimisation problem by using the **negative log-likelihood** $-\ln P(\mathcal{D}_{train} \mid \mathcal{W})$. In addition, most error measures are invariant to dataset size which makes it easy to compare the performance of a model on different data sets. To give the negative log-likelihood this property, it's common to divide by N, giving what is called the **average negative log-likelihood**. Minimizing the average negative log-likelihood is clearly identical to maximizing the likelihood, since $\max f(\mathbf{x}) = \min -f(\mathbf{x})$, and dividing by N doesn't change the optimum.

Definition 3.1.2 (average negative log-likelihood)**.** Let $\mathcal{D}_{train}$ and $h(\mathbf{x}; \mathcal{W})$ be defined as in definition 3.1.1. Then the negative log likelihood $-\ln P(\mathcal{D}_{train} \mid \mathcal{W})$ is:

$$\hat{E}(\mathbf{w}, \mathcal{D}_{train}) = -\frac{1}{N} \ln P(\mathcal{D}_{train} \mid \mathbf{w}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \ln h(\mathbf{x}_i, \mathbf{w})_c$$

This error measure is also known as **cross-entropy error**, in which the term $-\sum_{c=1}^{C} y_{ic} \ln h(\mathbf{x}_i, \mathbf{w})_c$ is taken as the error measure $e(h(\mathbf{x}_i), \mathbf{y}_i)$, which allows us to write $\hat{E}$ in the familiar form used in section 2.2.2: $\hat{E}(h, \mathcal{D}_{train}) = \frac{1}{N} \sum_{i=1}^{N} e(h(\mathbf{x}_i), \mathbf{y}_i)$.

In the next section, we will see how to use the average negative log-likelihood to find a good $h \in \mathcal{H}$.

3.2 Learning Algorithm

Finding a function $h \in \mathcal{H}$ that maximises the likelihood of $\mathcal{D}$ is an optimization problem. Optimization is solved by answering the question: *how does $\hat{E}$ change when we change h?* We answer questions of this type with differential calculus. Sadly, there is no known method for finding the h which maximizes the likelihood by analytical differentiation. Neural network optimization is therefore solved using an iterative algorithm called **gradient descent**, which we describe in this section. We go on to explore an algorithm for computing the gradient of $\hat{E}$ called **backpropagation**. Finally, we look into **regularization** which are tools for constraining the learning algorithm in order to avoid overfitting. Lastly, we describe a specific learning algorithm called **Adam**, an efficient variation on gradient descent.

3.2.1 Gradient Descent

We want to find a $h \in \mathcal{H}$ that minimises $\hat{E}$ as described in section 3.1.2. Each h is defined exactly by the weight vector $\mathbf{w}$. $\hat{E}$ can't be minimised analytically, since its derivative with respect to $\mathbf{w}$ is a system of non-linear equations, which in general does not have an analytical solution. We therefore look for h by choosing an initial weight vector $\mathbf{w}_0$, and iteratively reduce $\hat{E}$: In iteration i, the weight vector $\mathbf{w}_i$ is found by taking a small step η in a direction given by a vector $\mathbf{v}$, or more formally: $\mathbf{w}_i = \mathbf{w}_{i-1} + \eta\mathbf{v}$. The main question is, which direction should we choose?

$\hat{E}$'s direction of steepest descent at each $\mathbf{w}_i$ is given by the gradient $\nabla\hat{E}$. $\nabla\hat{E}$ is a vector where each component is a partial derivative $\frac{\partial}{\partial w}\hat{E}$ with respect to a weight $w \in \mathbf{w}$:

Definition 3.2.1 (gradient)**.** Let $w_{ij}^{(l)} \in \mathbf{w}$ be every weight in h, and let $\hat{E}$ be defined as in definition 3.1.2. Then the gradient $\nabla\hat{E}$ is:

$$\nabla\hat{E} = \begin{bmatrix} \frac{\partial}{\partial w_{ij}^{(1)}}\hat{E} \\ \vdots \\ \frac{\partial}{\partial w_{ij}^{(L)}}\hat{E} \end{bmatrix}$$

The gradient can be used for computing the rate of change of $\hat{E}$ in the direction of a unit vector $\mathbf{u}$ by taking the dot product $\mathbf{u}^T\nabla\hat{E}$. We would like to know in which direction $\mathbf{u}$ we should change $\mathbf{w}_i$ in order to make $\hat{E}$ as small as possible. The dot product of $\mathbf{u}^T\nabla\hat{E}$ is equal to $|\nabla\hat{E}||\mathbf{u}|\cos\theta$ where θ is the angle between $\nabla\hat{E}$ and $\mathbf{u}$. The direction $\mathbf{u}$ with the greatest positive rate of change of $\hat{E}$ is the direction in which $\theta = 0°$, in other words, the same direction as $\nabla\hat{E}$. The direction with the greatest negative rate of change of $\hat{E}$ is the direction in which $\theta = 180°$, in other words, the direction $-\nabla\hat{E}$. This means that we can make $\hat{E}$ smaller by taking a small step η in the direction $-\nabla\hat{E}$, such that $\mathbf{w}_i = \mathbf{w}_{i-1} - \eta\nabla\hat{E}$. A small example is given in figure 3.5 and 3.6.

One challenge of gradient descent is that $\nabla\hat{E} = \frac{1}{N}\sum_{i=1}^{N}\nabla e(h(\mathbf{x}_i), \mathbf{y}_i)$ is based on

all the examples in $\mathcal{D}$. This means that computing $\nabla \hat{E}$ requires one full iteration over the training set. If the training set is large, this means that every update to the weights $\mathbf{w}$ takes a long time which makes learning slow. **Stochastic gradient descent** is a common variation on gradient descent which addresses this problem. In stochastic gradient descent, a single training example $(\mathbf{x}_i, \mathbf{y}_i)$ is sampled from $\mathcal{D}$. Instead of updating $\mathbf{w}_i$ by the gradient $-\nabla \hat{E}$ over all the training examples, we update the weights based on the gradient of a single example $\mathbf{w}_i = \mathbf{w}_{i-1} - \eta \nabla e(h(\mathbf{x}_i), \mathbf{y}_i)$. Since each sample in $\mathcal{D}$ can be drawn with probability $\frac{1}{N}$, stochastic gradient descent is identical to gradient descent in expectation:

$$\mathbb{E}(-\nabla e(h(\mathbf{x}_i), \mathbf{y}_i)) = \frac{1}{N} \sum_{i=1}^{N} -\nabla e(h(\mathbf{x}_i), \mathbf{y}_i) = -\nabla \hat{E}$$

In traditional gradient descent, $\nabla \hat{E}$ approaches $\mathbf{0}$ when $\mathbf{w}_i$ approaches a local or global minimum for $\hat{E}$. This prevents the algorithm from stepping far away from this minimum once it's close to a solution. When using stochastic gradient descent however, each update to the weights is based on just a single example and is therefore noisy, which means that $\nabla e(h(\mathbf{x}_i, \mathbf{y}_i)$ may be large, even if $\mathbf{w}_i$ is close to a value that minimizes $\hat{E}$. To reduce the noise in the gradient estimate, its common to sample a small mini-batch from $\mathcal{D}$ and perform gradient descent on that. In addition, its common to shrink the learning rate η as the algorithm progresses. We will see a strategy for shrinking η systematically in the next section.

3.2.2 Adam

The Adam algorithm is a variation on stochastic gradient descent that attempts to shrink the learning rate η automatically in each iteration. Since the learning rate varies from iteration to iteration, we will denote the learning rate in iteration i as η_i. Moreover, Adam adapts the learning rate for each parameter w individually, by using a learning rate vector $\boldsymbol{\eta}$ instead of a scalar in the update rule, such that $\mathbf{w}_i = \mathbf{w}_{i-1} - \boldsymbol{\eta}_i \nabla e(h(\mathbf{x}_i), \mathbf{y}_i)$

Adam uses the following heuristic: the learning rate for parameters w for which $\frac{\partial}{\partial w} e(h(\mathbf{x}_i), \mathbf{y}_i)$ is frequently large should decrease more quickly than parameters that consistently have small derivatives. To achieve this, Adam scales η by $\mathbf{v}_i$ such that:

$$\mathbf{v}_i = \beta_1 \mathbf{v}_{i-1} + (1 - \beta_1)(\nabla e(h(\mathbf{x}_i, \mathbf{y}_i))^2$$

Where $\mathbf{v}_0 = \mathbf{0}$. In words, $\mathbf{v}_i$ is an exponentially decaying average of past squared gradients, where β_1 is the decay rate, usually set to a value near .9. To cancel the bias introduced by initialising $\mathbf{v}_i$ to $\mathbf{0}$, a bias corrected value $\hat{\mathbf{v}}_i = \mathbf{v}_i / (1 - \beta_1^i)$ is computed. The learning rate is then computed as:

$$\boldsymbol{\eta}_i = \frac{\eta}{\sqrt{\hat{\mathbf{v}}_i} + \epsilon}$$

Where ϵ is small value introduced to prevent division by 0.

In addition to scaling the learning rate, Adam uses the idea of **momentum** to speed up stochastic gradient descent. Momentum is designed to make stochastic gradient

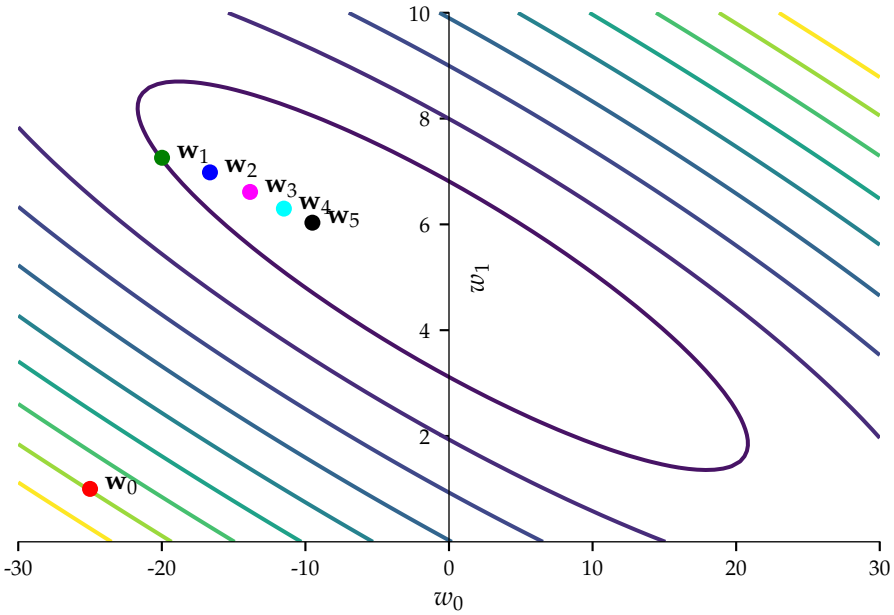


Figure 3.5
Level curves of squared training error $\hat{E}(h, \mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}(h(\mathbf{x}_i) - y_i)^2$ for a toy $\mathcal{D}$ shown in 3.6, and the simple $\mathcal{H} = \{h = \mathbf{w}^T\mathbf{x}^{(0)} \mid \mathbf{w} \in \mathbb{R}^2\}$. $\hat{E}$ has its minimum at $(0, 5)$. Each colored dot corresponds to a step $\mathbf{w}_i$ in gradient descent using a fixed learning rate η. The first step from $\mathbf{w}_0$ to $\mathbf{w}_1$ makes a lot of progress towards the minimum, and each subsequent update to $\mathbf{w}_i$ is much less dramatic.

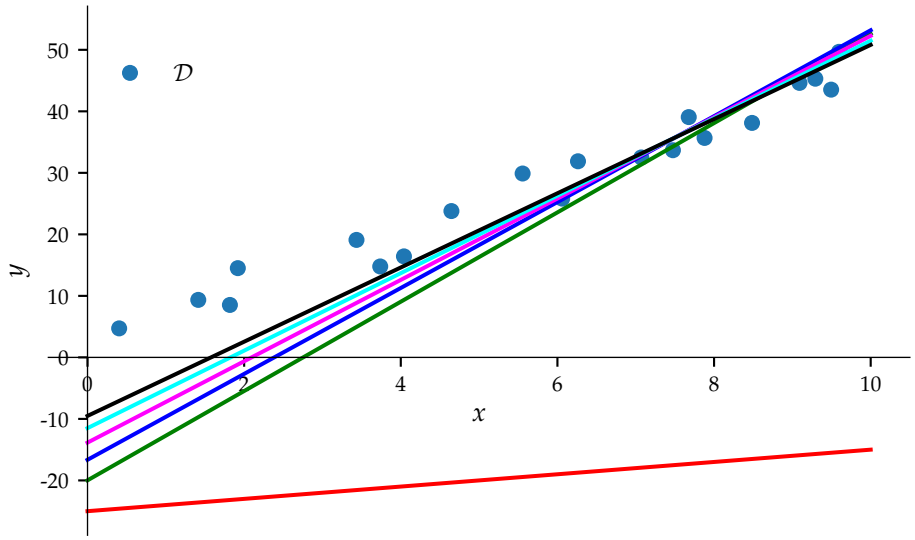


Figure 3.6
The training data $\mathcal{D}$ used in figure 3.5. The colored lines correspond to $h(\mathbf{x}, \mathbf{w}_i) = 0$ for each weight vector $\mathbf{w}_i$ found by gradient descent in figure 3.5, such that for example $h(\mathbf{x}, \mathbf{w}_0) = 0$ is given by the red line. We see as gradient descent makes $\hat{E}$ smaller, the lines fit $\mathcal{D}_{train}$ better.

descent more robust to high curvature in $e(h(\mathbf{x}_i), \mathbf{y}_i)$ and noisy gradients. This is achieved by changing the update rule, such that the parameters $\mathbf{w}_i$ are updated not in the direction of $-\nabla e(h(\mathbf{x}_i), \mathbf{y}_i)$, but in the direction of an exponentially decaying average of past gradients $\mathbf{m}_i$:

$$\mathbf{m}_i = \beta_2 \mathbf{m}_{i-1} + (1 - \beta_2) \nabla e(h(\mathbf{x}_i), \mathbf{y}_i)$$

where $\mathbf{m}_0 = 0$ and β_2 is the decay rate. Just as before, the initialisation bias is corrected by computing $\hat{\mathbf{m}}_i = \mathbf{m}_i / (1 - \beta_2^i)$.

The full update rule for Adam is thus:

$$\mathbf{w}_i = \mathbf{w}_{i-1} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_i} + \epsilon} \hat{\mathbf{m}}_i$$

Gradient descent gives us an algorithm for minimising $\hat{E}$ using $\nabla \hat{E}$. In the next section we explore an algorithm for computing $\nabla \hat{E}$ called backpropagation.

3.2.3 Backpropagation

We want to compute $\nabla \hat{E}$ in order to use gradient descent to make $\hat{E}$ small. Because of the sum and product rules of differential calculus, we can simplify our analysis by computing $\nabla \hat{E}$ of a single example $(\mathbf{x}, \mathbf{y})$:

$$\nabla \hat{E} = \nabla \frac{1}{N} \sum e(h(\mathbf{x}_i), \mathbf{y}_i) = \frac{1}{N} \sum \nabla e(h(\mathbf{x}_i), \mathbf{y}_i)$$

In our explanation we will use the cross-entropy error $e(h(\mathbf{x}), \mathbf{y}) = -\sum_{c=1}^{C} y_c \ln h(\mathbf{x})_c$ as an example.

If we can derive a generic formula for a single component $\frac{\partial e}{\partial w_{ij}^{(l)}}$ of ∇e, we can compute all of ∇e. The partial derivative is asking the question *how does e change if we change* $w_{ij}^{(l)}$*?* The weight $w_{ij}^{(l)}$ influences e only through the activation $a_j^{(l)}$. We can therefore decompose the derivative using the chain rule of calculus:

$$\frac{\partial e}{\partial w_{ij}^{(l)}} = \frac{\partial e}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}}$$

The term $\frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}}$ is easy to compute, because $a_j^{(l)}$ depends directly on $w_{ij}^{(l)}$ in a simple sum:

$$\frac{\partial a_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial}{\partial w_{ij}^{(l)}} \sum_{k=0}^{d^{(l-1)}} w_{kj}^{(l)} x_k^{(l-1)} = x_i^{l-1}$$

The term $\frac{\partial e}{\partial a_j^{(l)}}$ is more involved since $a_j^{(l)}$ influences e through units in layers $m > l$ that directly or indirectly receives input from unit j in layer l. Computing $\frac{\partial e}{\partial a_j^{(l)}}$ therefore requires a number of applications of the chain rule that depend on the number

of layers between $a_j^{(l)}$ and the output. The backpropagation algorithm solves this problem by defining $\delta_j^{(l)} = \frac{\partial e}{\partial a_j^{(l)}}$, and deriving a recursive formula for $\delta_j^{(l)}$ that relates it to $\delta_j^{(l-1)}$.

We start by computing $\delta_j^{(L)}$, since the activation in the output layer $a_j^{(L)}$ influences e directly and can therefore be used as a base case for the recursion, that doesn't depend on any other $\delta_j^{(l)}$

Lets start by rewriting e in terms of the output of layer L:

$$e(h(\mathbf{x}), \mathbf{y}) = -\sum_{c=0}^{C} y_c \ln x_c^{(L)}$$

Where $x_c^{(L)}$ is the output of unit c in the output layer. When using soft-max activation in the output layer would mean that $x_c^{(L)} = \sigma(\mathbf{a}^{(L)})_c = \frac{e^{a_c^{(l)}}}{\sum_{i=1}^{C} e^{a_i^{(l)}}}$.
Since $a_j^{(L)}$ affects e through the soft-max activation, we will need to compute the derivative of the soft-max activation with respect to the activation $\frac{\partial x_i^{(L)}}{\partial a_j^{(L)}}$ in order to compute $\delta_j^{(L)}$. This derivative is different depending on which output $x_i^{(L)}$, and which activation $a_j^{(L)}$ we consider.

If $i = j$, that is, we are taking the derivative of the output of a unit with respect to its activation, we get:

$$\begin{aligned}
\frac{\partial x_i^{(L)}}{\partial a_i^{(L)}} &= \frac{\partial}{\partial a_i^{(L)}} \frac{e^{a_i^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}} = \frac{e^{a_i^{(L)}} \sum_{c=1}^{C} e^{a_c^{(L)}} - e^{a_i^{(L)}} e^{a_i^{(L)}}}{\left(\sum_{c=1}^{C} e^{a_c^{(L)}}\right)^2} = \frac{e^{a_i^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}} \frac{\left(\sum_{c=1}^{C} e^{a_c^{(L)}}\right) - e^{a_i^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}} \\
&= \frac{e^{a_i^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}} \left(1 - \frac{e^{a_i^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}}\right) \\
&= x_i^{(L)}(1 - x_i^{(L)})
\end{aligned}$$

If $i \neq j$, in other words, if we are taking the derivative of the output of a unit with respect to the activation of another unit, we get:

$$\frac{\partial x_i^{(L)}}{\partial a_j^{(L)}} = \frac{0 - e^{a_i^{(L)}} e^{a_j^{(L)}}}{\left(\sum_{c=1}^{C} e^{a_c^{(L)}}\right)^2} = -\frac{e^{a_i^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}} \frac{e^{a_j^{(L)}}}{\sum_{c=1}^{C} e^{a_c^{(L)}}} = -x_i^{(L)} x_j^{(L)}$$

Armed with $\frac{\partial x_i^{(L)}}{\partial a_j^{(L)}}$, we can go on to compute $\delta_j^{(L)}$:

$$\begin{aligned}
\delta_j^{(L)} = \frac{\partial e}{\partial a_j^L} &= -\sum_{c=1}^{C} y_c \frac{\partial}{\partial a_j^L} \ln x_c^{(L)} = -\sum_{c=1}^{C} y_c \frac{1}{x_c^{(L)}} \frac{\partial x_c^{(L)}}{\partial a_j^{(L)}} = -\frac{y_j}{x_j^{(L)}} \frac{\partial x_j^{(L)}}{\partial a_j^{(L)}} - \sum_{c \neq j}^{C} \frac{y_c}{x_c^{(L)}} \frac{\partial x_c^{(L)}}{\partial a_j^{(L)}} \\
&= -\frac{y_j}{x_j^{(L)}} x_j^{(L)}(1 - x_j^{(L)}) - \sum_{c \neq j}^{C} \frac{y_c}{x_c^{(L)}}(-x_c^{(L)} x_j^{(L)}) = -y_j + y_j x_j^{(L)} + \sum_{c \neq j}^{C} y_c x_j^{(L)} \\
&= -y_j + \sum_{c=1}^{C} y_c x_j^{(L)} = -y_j + x_j^{(L)} \sum_{c=1}^{C} y_c \\
&= x_j^{(L)} - y_j
\end{aligned}$$

Finally, we see that the derivative of the error with respect to the activation of unit j in the output layer is simply $x_j^{(L)} - y_j$.

Having derived a formula for $\delta_j^{(L)}$, we can go on to recursively derive $\delta_i^{(l-1)}$. Since e depends on $a_i^{(l-1)}$ only through $x_i^{(l-1)}$, we can use the chain rule to decompose $\delta_i^{(l-1)}$:

$$\delta_i^{(l-1)} = \frac{\partial e}{\partial a_i^{(l-1)}} = \frac{\partial e}{\partial x_i^{(l-1)}} \frac{\partial x_i^{(l-1)}}{\partial a_i^{(l-1)}}$$

The derivative of the output of unit i with respect to its input is simply the derivative of the activation function σ. We leave this generic here:

$$\frac{\partial x_i^{(l-1)}}{\partial a_i^{(l-1)}} = \sigma'(a_i^{(l-1)})$$

Since e depends on $x_i^{(l-1)}$ through the activation of every unit j that i is connected to, the chain rule tells us that we must sum the effects on e of changing $x_i^{(l-1)}$:

$$\frac{\partial e}{\partial x_i^{(l-1)}} = \sum_{i=1}^{d^{(l)}} \frac{\partial a_j^{(l)}}{\partial x_i^{(l-1)}} \frac{\partial e}{\partial a_j^{(l)}} = \sum_{i=1}^{d^{(l)}} w_{ij}^{(l)} \delta_j^{(l)}$$

We now finally have a recursive formula for $\delta_i^{(l-1)}$:

$$\delta_i^{(l-1)} = \frac{\partial e}{\partial a_i^{(l-1)}} = \sigma'(a_i^{(l-1)}) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)} \delta_j^{(l)}$$

To summarize, we now have a recursive formula for every weight component of the gradient $\frac{\partial e}{\partial w_{ij}^{(l)}}$ given by:

$$\frac{\partial e}{\partial w_{ij}^{(l)}} = x_i^{(l-1)} \delta_j^{(l)}, \quad \delta_j^{(l)} = \sigma'(a_i^{(l)}) \sum_{i=1}^{d^{(l+1)}} w_{ij}^{(l+1)} \delta_j^{(l+1)}$$

3.2.4 Regularization

In section 2.2.2 we saw that the distance between E and $\hat{E}(h, \mathcal{D})$ is bounded by, among other things, a function of the diversity of $\mathcal{H}$. In this section we discuss techniques for restricting the learning algorithm to search only in subsets of $\mathcal{H}$, with the aim of reducing E. These techniques are collectively known as regularization.

For a $\mathcal{H}$ thats parameterized by a weight vector $\mathbf{w}$ such as the hypothesis space given by a particular neural network architecture, we can limit the region of weight space that our learning algorithm is allowed to consider by imposing the constraint that the norm of $\mathbf{w}$ must be smaller than some constant C. This has the effect that the weights can be selected only from a limited spherical region around the origin. This reduces the effective number of different hypotheses available during learning, and the Vapnik-Chervonenkis bound gives us confidence that this should improve generalisation.

If the weights $\mathbf{w}^*$ that minimises the unconstrained training error $\hat{E}(\mathbf{w}, \mathcal{D}_{train})$ lie outside this ball, then the weights $\bar{\mathbf{w}}$ that minimises $\hat{E}$ while still satisfying the constraint $\bar{\mathbf{w}}^T\bar{\mathbf{w}} \leq C$, must have norm equal to C, in other words lie on the surface of the sphere with radius C. The normal vector to this surface at any $\mathbf{w}$ is $\mathbf{w}$ itself. At $\bar{\mathbf{w}}$, the normal vector must point in the exact opposite direction of $\nabla\hat{E}$, since otherwise $\nabla\hat{E}$ would have a component along the border of the constraint sphere, and we could decrease $\hat{E}$ by moving along the border of the sphere in the direction of $\nabla\hat{E}$ and still satisfy the constraint. In other words, the following equality holds for $\bar{\mathbf{w}}$:

$$\nabla\hat{E}(\bar{\mathbf{w}}, \mathcal{D}) = -2\lambda\bar{\mathbf{w}}$$

Where λ is some proportionality constant. Equivalently, $\bar{\mathbf{w}}$ satisfy:

$$\nabla(\hat{E}(\bar{\mathbf{w}}, \mathcal{D}) + \lambda\bar{\mathbf{w}}^T\bar{\mathbf{w}}) = \mathbf{0}$$

Because $\nabla(\bar{\mathbf{w}}^T\bar{\mathbf{w}}) = 2\bar{\mathbf{w}}$. In other words, for some $\lambda > 0$, $\bar{\mathbf{w}}$ minimizes a new error function which we will call **augmented error** $\bar{E}(\mathbf{w}, \mathcal{D})$:

$$\bar{E}(\mathbf{w}, \mathcal{D}) = \hat{E}(\mathbf{w}, \mathcal{D}) + \lambda\mathbf{w}^T\mathbf{w}$$

This means that the problem of minimizing $\hat{E}(\mathbf{w}, \mathcal{D})$ constrained by $\mathbf{w}^T\mathbf{w} \leq C$ is equivalent of minimizing $\bar{E}(\mathbf{w}, \mathcal{D})$. This is useful because minimizing $\bar{E}(\mathbf{w}, \mathcal{D})$ can be done by gradient descent which makes it a useful regularization scheme for neural networks where analytical solutions are not possible in general.

This particular form of regularization where a penalty on the norm of the weight vector is added to the minimization objective is called **weight decay**. To see why, lets consider a single step of the gradient descent algorithm when minimizing $\bar{E}(\mathbf{w}, \mathcal{D})$. In iteration i the weight vector $\mathbf{w}_i$ is given by:

$$\mathbf{w}_i = \mathbf{w}_{i-1} - \eta\nabla\bar{E}(\mathbf{w}, \mathcal{D}) = \mathbf{w}_{i-1}(1 - 2\eta\lambda) - \eta\nabla\hat{E}(\mathbf{w}_{i-1}, \mathcal{D})$$

In words, the added norm penalty of $\bar{E}(\mathbf{w}, \mathcal{D})$ has the effect of pulling the vector $\mathbf{w}_i$ towards $\mathbf{0}$, by multiplying by $1 - 2\eta\lambda$ in each iteration. In this way, weight decay is limiting the region that gradient descent can explore in a finite number of iterations,

and is therefore limiting the diversity of $\mathcal{H}$.

Early stopping is a form of regularization for iterative optimization methods that is particularly straight-forward to implement, and as an added bonus gives a reasonable stopping criterion for gradient descent. It works very similarly to weight decay, by limiting the region of $\mathcal{H}$ that can be explored in a finite number of iterations.

For a single iteration i of gradient descent with step size η, gradient descent explores all weights in a radius of η around $\mathbf{w}_i$, since a step in the direction of the negative gradient minimizes $\hat{E}(\mathbf{w}, \mathcal{D})$ among all weights with $||\mathbf{w} - \mathbf{w}_i|| \leq \eta$. In other words, we can think of an effective hypothesis space $\mathcal{H}_i$ for each iteration thats limited by η:

$$\mathcal{H}_i = \{\mathbf{w} \mid ||\mathbf{w} - \mathbf{w}_i|| \leq \eta\}$$

We can think of the hypothesis space $\mathcal{H}$ explored by gradient descent in a finite number of steps I as the union of these sets:

$$\mathcal{H} = \bigcup_{i=1}^{I} \mathcal{H}_i$$

As I increases, $\mathcal{H}$ becomes more diverse, and Vapnik-Chervonenkis theory tells us that the risk of selecting $\mathbf{w} \in \mathcal{H}$ that fits the noise in $\mathcal{D}$ increases. In practice, it is consistently observed that both $E(\mathbf{w_i})$ and $\hat{E}(\mathbf{w}_i, \mathcal{D})$ is decreased as a function of i, until a certain point i^*, after which only $\hat{E}(\mathbf{w}_i, \mathcal{D})$ is decreased, as a consequence of fitting the noise in $\mathcal{D}$, which causes $E(\mathbf{w}_i)$ to increase. See figure 3.7 for a visualization. When using early stopping, we treat i^* the optimal number of iterations of gradient descent as a parameter we want to estimate. This is done through validation, as described in section 2.2.3. Specifically, after each gradient descent iteration, $\hat{E}(\mathbf{w}_i, \mathcal{D}_{val})$ is computed as an estimate of E. When this quantity is no longer improving, gradient descent is halted, and the the parameters $\mathbf{w}_{i^*}$ are returned.

When using stochastic gradient descent, $\hat{E}(\mathbf{w}_i, \mathcal{D}_{val})$ may vary slightly from iteration to iteration due to the noise introduced by the stochastic gradient. This means that the simple heuristic stopping criterion described above may fail when using stochastic gradient descent. A so called **patience** parameter is a simple solution to this problem. When using patience p, stochastic gradient descent is only halted when no improvement on $\hat{E}(\mathbf{w}_i, \mathcal{D}_{val})$ has been observed for p iterations.

3.3 Convolutional Neural Networks

A convolution $f * k$ is a mathematical operation that takes as input two functions f and k.

Definition 3.3.1 (convolution)**.** Let $f(x) \in \mathbb{R}$ and $k(x) \in \mathbb{R}$ be two real-valued functions defined for the entire real number line. Then the convolution $f * k$ is defined as

$$(f * k)(x) = \int f(y)k(x - y)dy$$

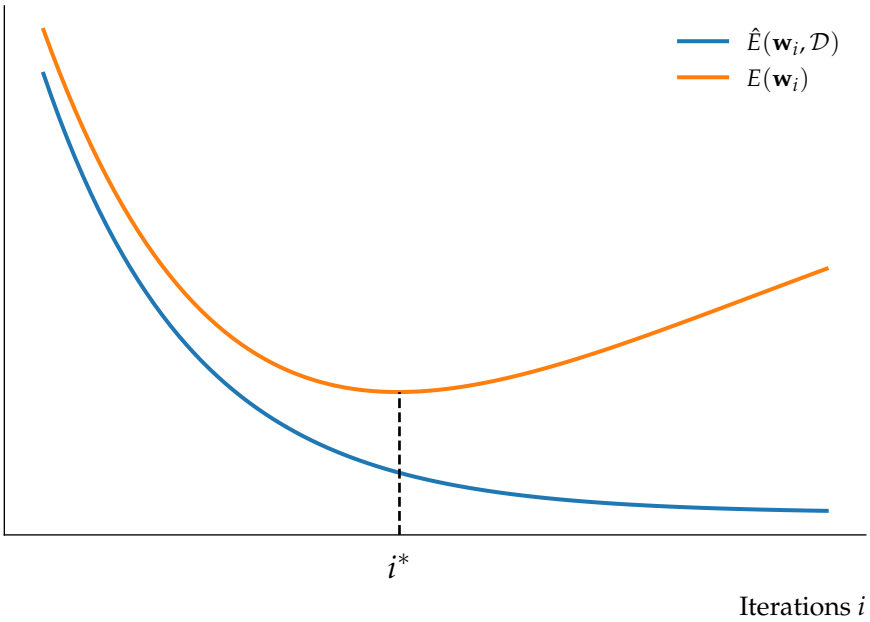


Figure 3.7
Typical behavior of E and $\hat{E}$ as a function of the number of iterations i of gradient descent. Both errors are reduced until a point i^* beyond which the training error is reduced, but generalization error increases.

In practical applications involving computers, f and k are discrete, and the integral turns into a sum:

$$(f * k)(x) = \sum_{y=-\infty}^{\infty} f(y)k(x-y)$$

Most functions in practical applications of convolutions represent signals such as images, sound or text, which are only defined over a limited range of indices x. In these cases it's assumed that whenever x is beyond the domain of f or k, the output of either function is 0.

We can think of a convolution as a weighted sum of the output of f where the output of k acts as the weights. This view of convolution is used heavily in signal processing applications where k is chosen to produce certain properties in the convolution output, such as reducing noise in f. In this setting k is often referred to as a **kernel**. As an example consider the noisy signal convolved with a gaussian kernel in figure 3.8.

The kernel k can also act as a **feature detector**. When the output of f is closely correlated with the output of k, the output of the convolution spikes. See for example figure 3.9.

Convolutional neural networks are neural networks that take advantage of convolutions as feature detectors. By arranging the layers and weights in the network in specific ways, we can construct a network such that the output of each layer l is the output of layer $l-1$ convolved with a kernel k, where the weights of k are exactly

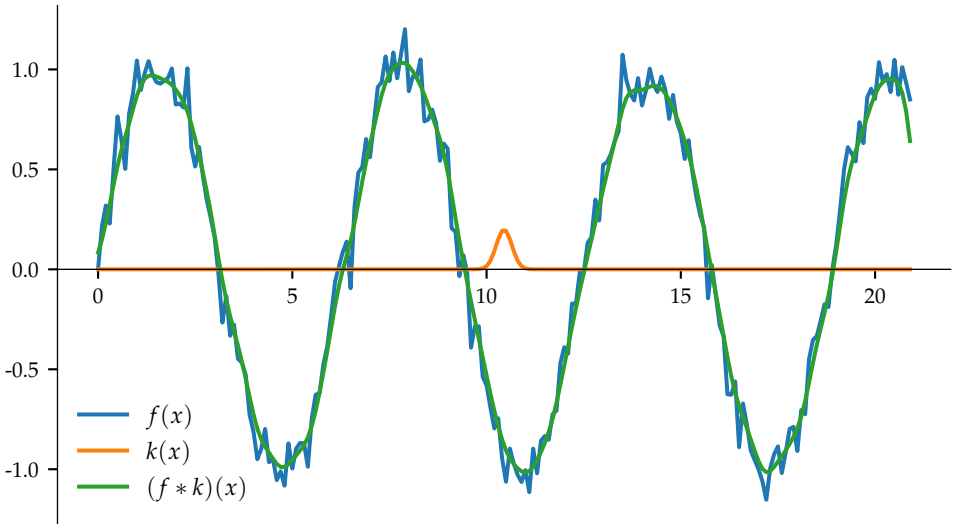


Figure 3.8
Visualisation of a noisy signal f convolved with a small Gaussian kernel k. The output of the convolution $f * k$ captures the general trend of f by averaging the outputs of f at every x, such values of f of inputs close to x contribute more to the output of the convolution, than inputs far away from x thanks to the weights of the Gaussian kernel.

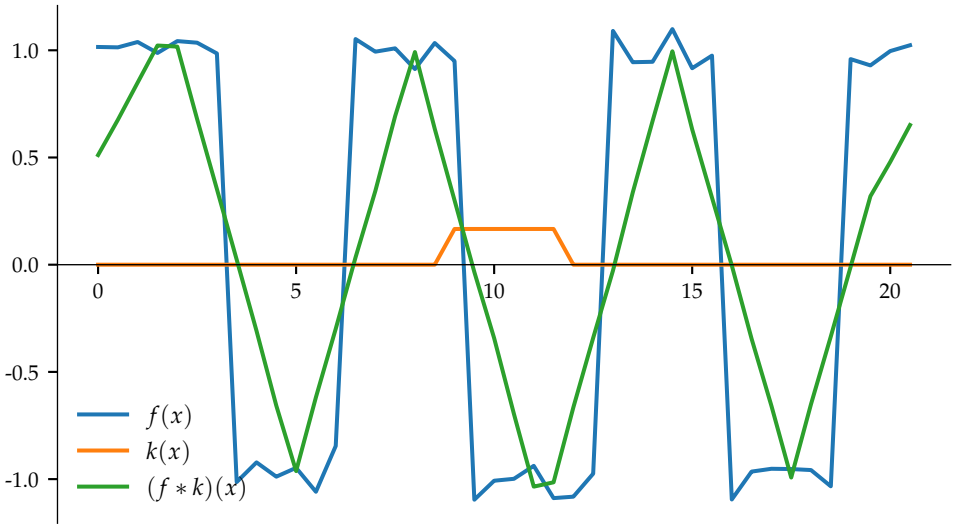


Figure 3.9
Visualisation of convolutional kernel as feature detector. When the signal f is similar to the kernel, the output of the convolution is maximally positive.

the neural network weights connecting the units in layer l and $l-1$.

Specifically, the weights connecting layers l and $l-1$ in a convolutional neural network should be arranged such that they are:

sparse each unit in layer l receives input from a small number of units layer $l-1$.

shared the weights connecting units in layer l and $l-1$ are shared across the layer, in the same way that the same kernel weights are re-used around every index of f. See figure 3.10.

These restrictions on the network architecture reduces the number of unique weights of the model. This has the effect of reducing both the memory requirements of storing the network, but also limits the number of operations required to compute the output of the network for a given input.

Intuitively, the output at each unit u in l in a convolutional layer indicates how strongly the feature detected by the kernel given by its connecting weights is present in the output of units that u connects to in layer $l-1$. Since the weights are learned by gradient descent, the feature detected by units in layer l is learnt as well.

Often, the simple presence or absence of a feature in the output of layer $l-1$ is very informative for the classification task the convolutional network was built to solve. The exact position of a detected feature in layer $l-1$ is often less informative however. For this reason, convolutional layers are often interleaved with so called **pooling layers**. The output of a pooling layer can be thought of as a summary how strongly a feature is detected in layer l, that discards information about the exact position at which the features was detected. Very commonly, max-pooling is used which simply outputs the maximum value over all outputs of units in layer l.

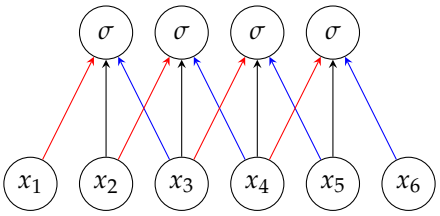


Figure 3.10
Visual representation of a one-dimensional convolution implemented as the first layer of a convolutional neural network. The connections between the input layer and the convolutional layer are sparse in that each unit is connected only to three of six inputs. The colors of the connections indicate how the weights are shared.

3.4 Word Vectors

The way text is represented in a computer doesn't in general encode any information about semantic similarities between words or sentences. Instead, text is most often represented as sequences of discrete symbols. Learning a h that maps from discrete input space where distances between points don't encode similarity, such as words, to a prediction, such as the presence of a named entity, may be more difficult than

learning a mapping from continuous input space to a prediction, since a continuous function can be expected to have some smoothness properties, i.e similar inputs should have similar outputs (Bengio et al., 2003).

For this reason some effort has been devoted to designing real-valued vector representation of words, so called **word vectors**, that encode semantic similarities such that words with similar meaning are close to each other in word-vector space. The notion of "meaning" of a word is a philosophically challenging one. A simple definition which leads to simple but useful algorithms is that words have similar meaning if they are used in similar contexts.

This leads to the idea of representing words as vectors of co-occurrence counts. Two words w_i and w_j co-occur in a context of c words if w_j appears somewhere in a window of c words from w_i in some piece of text. By representing w_i as a vector $\mathbf{w}_i \in \mathbb{R}^V$ of co-occurrence counts for the V words in some vocabulary, words that occur in similar contexts will be close to each other in co-occurrence vector space.

The main problems with this representation is that V may be very large, and $\mathbf{w}_i$ may be very sparse, that is, most of its components are 0 since most words never co-occur together. Recent solutions to this problem learn lower dimensional word vectors using co-occurrence statistics. **GloVe** is a recent and successful technique for learning word vectors that encode much useful syntactic and semantic information (Pennington et al., 2014). In GloVe, each word w_i is represented by a word vectors $\mathbf{w}_i$, and a context word vector $\tilde{\mathbf{w}}_i$. The vectors are initialized randomly

Glove vectors are learned by minimizing the objective function:

$$\sum_{i=1}^{V}\sum_{i=1}^{V} f(\mathbf{X}_{ij})(\mathbf{w}_i^T\tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \ln \mathbf{X}_{ij})^2$$

Where $\mathbf{X}_{ij}$ is the co-occurrence count for word w_i and w_j, and b_i and $\tilde{b}_j$ are bias terms. f is a weighting function that gives low weight to infrequent terms and caps extremely frequent terms, defined as:

$$f(x) = \begin{cases} (x/x_{max})^{3/4} & \text{when } x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

Minimizing this objective leads to word vectors whose dot products are close to log-co-occurrence counts for the words they represent. It can be shown that this has the effect that word vector differences encode information about ratios of log-co-occurrence probabilities which are highly informative of semantic similarity.

It is now common practice to incorporate word vectors in neural network models for natural language processing tasks in a so called embedding layer. In this scheme, the components of the word vectors are parameters that can be trained by back propagation to yield word vector representations that are informative for a given task. These word embedding vectors, or simply word embeddings, can be initialized with small random components as any other neural network parameter, or they can be initialized with pre-learned word vectors, for example GloVe vectors.

Part 4

Multi-Task Learning

In this section we introduce an extension to the supervised machine learning framework called multi-task learning. We first cover the main ideas and motivation for multi-task learning. We then summarize different variations on statistical learning theory for multi-task learning to gain an intuition of how and when multi-task learning works.

4.1 Multi-Task and Single-Task Learning

In our description of supervised machine learning so far we have assumed that the input for the learning system was an annotated dataset $\mathcal{D}$ in which all samples $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}$ are drawn independently from the same distribution $P(\mathbf{x}, \mathbf{y})$. In the real world however, it's often possible to combine data from disparate sources if we relax this assumption: We may have access to a set $\mathcal{D}_M$ of M datasets $\mathcal{D}_m \in \mathcal{D}_M$, drawn from a set $\mathcal{P}$ of M different distributions $P_m(\mathbf{x}, \mathbf{y}) \in \mathcal{P}$. Since creating new labels for a machine learning task is often both cumbersome and costly, it would be desirable if re-using previously labeled data could reduce the need for data annotation

In many cases, we are not interested in implementing a learning system that performs well on all M learning tasks. We really only care about one **target** task defined by a distribution $P_t \in \mathcal{P}$ and $\mathcal{D}_t \in \mathcal{D}_M$ in which case we consider the other datasets $\mathcal{D}_A = \{\mathcal{D}_m \mid \mathcal{D}_m \in \mathcal{D}_M, m \neq t\}$ to be **auxiliary**. Since we are dealing with more than one probability distribution, it becomes useful to think of generalization error with respect to a particular distribution. Thus, we extend our notation for generalization error from E to E_m to mean:

$$E_m(h) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim P_m(\mathbf{x},\mathbf{y})}[e(h(\mathbf{x}), \mathbf{y})]$$

We can speculate that if $\mathcal{D}_t$ and $\mathcal{D}_A$ are related somehow, and if the learning system is able to share what is learnt between the learning tasks, learning the tasks simultaneously may improve generalization for the target task relative to learning from $\mathcal{D}_t$ in isolation (Caruana, 1997). To distinguish the two approaches, the traditional approach to supervised machine learning as described in section 2.2 is called **single-task learning**, and the new approach, which learns from all of $\mathcal{D}_M$ is called **multi-task learning**.

In the following sections we introduce contributions from statistical learning theory that shed some light on when and how learning from $\mathcal{D}_M$ is beneficial.

4.2 Bias Learning

Selecting the hypothesis space $\mathcal{H}$, sometimes referred to as **biasing** the hypothesis space, is often the hardest problem in supervised machine learning (Baxter, 2000). Vapnik-Chervonenkis analysis tells us that $\mathcal{H}$ must be large enough to contain a good solution to the learning problem of interest, yet small that the selected model can generalize from a small sample. This motivates developing techniques that can learn a good $\mathcal{H}$ from the data.

Baxter (2000) formalizes this idea by introducing a model of **bias learning**, in which the learning system is tasked with learning a hypothesis space $\mathcal{H}$ from a family of hypothesis spaces $\mathbb{H} = \{\mathcal{H}\}$. The system is supplied with M datasets $\mathcal{D}_m$ each drawn from M distributions P_m over $\mathcal{X} \times \mathcal{Y}$. The goal of the system is then to first select a good hypothesis space $\mathcal{H} \in \mathbb{H}$, and then to select a vector $\mathbf{h}$ of M hypothesis $h_m \in \mathcal{H}$. In his framework, the goal of the learning system is to minimize the multi-task generalization error defined as the average generalization error over the M learning problems:

$$E(\mathbf{h}) = \frac{1}{M} \sum_{m=1}^{M} E_m(h_m)$$

Similarly, we can generalize the empirical single-task error to an average multi-task empirical error $\hat{E}(\mathbf{h}, \mathcal{D}_M)$:

$$\hat{E}(\mathbf{h}, \mathcal{D}_M) = \frac{1}{M} \sum_{m=1}^{M} \hat{E}(h_m, \mathcal{D}_m)$$

The bias learning model of Baxter (2000) extends Vapnik-Chervonenkis analysis to the multi-task learning problem. To this end, he defines $\mathcal{H}(N, M)$ to be the set of all matrices of dichotomies, that can be formed from selecting M hypothesis from $\mathcal{H}$ and applying them to the to the N samples of the M datasets in $\mathcal{D}_M$:

$$\mathcal{H}(N, M) = \left\{ \begin{bmatrix} h_1(\mathbf{x}_{11}) & \cdots & h_1(\mathbf{x}_{1N}) \\ \vdots & \ddots & \vdots \\ h_M(\mathbf{x}_{M1}) & \cdots & h_M(\mathbf{x}_{MN}) \end{bmatrix} : h_1, \ldots, h_M \in \mathcal{H} \right\}$$

This allows him to define a concept of dichotomies on multi-task samples $\mathcal{D}_M$ for hypothesis space families, $\mathbb{H}(N, M)$:

$$\mathbb{H}(N, M) = \bigcup_{\mathcal{H} \in \mathbb{H}} \mathcal{H}(N, M)$$

And extend the growth function m to the multi-task setting:

$$m(N, M, \mathbb{H}) = \max |\mathbb{H}(N, M)|$$

With a binary label space, the maximum size of $\mathbb{H}(N, M)$ is 2^{NM}. Baxter uses this to define the Vapnik-Chervonenkis dimension $d(M, \mathbb{H})$ of the hypothesis space family $\mathbb{H}$:

$$d(M, \mathbb{H}) = \max\{N : m(N, M, \mathbb{H}) = 2^{NM}\}$$

In words, the Vapnik-Chervonenkis dimension of the hypothesis space family $\mathbb{H}$, is the largest number of samples N for which the family can generate all possible binary dichotomy matrices, when learning from M datasets of size N.

Using the same reasoning as is the basis of the original Vapnik-Chervonenkis bound, Baxter is able to show that in order for the average true error $E(\mathbf{h})$, to be within ϵ of the average empirical error $\hat{E}(\mathbf{h}, \mathcal{D}_M)$ with probability $1 - \delta$, it requires that the number of samples N for each task is:

$$N = O\left(\frac{1}{\epsilon^2}\left(d(M, \mathbb{H}) \log \frac{1}{\epsilon} + \frac{1}{M} \log \frac{1}{\delta}\right)\right)$$

Ignoring the confidence parameters ϵ and δ, we see that the number of examples N depends inversely on the number of tasks M. This means that we can reduce the number of samples required to keep E close to $\hat{E}$, if we can increase the number of learning tasks. This is an important result since it shows that multi-task bias learning can improve our confidence that E_m is close to $\hat{E}(h_m, \mathcal{D}_m)$ at least on average.

On the other hand, it's also a limited result in the sense that it doesn't tells anything about how $\hat{E}(h_m, \mathcal{D}_m)$ behaves in multi-task learning relative to single-task learning. In other words, it may be possible that bias learning leads to a hypothesis space $\mathcal{H}$ where $\hat{E}(\mathbf{h}, \mathcal{D}_M)$ is close to $E(\mathbf{h})$, but every $\hat{E}(h_m, \mathcal{D}_m)$ is much larger than would have been possible to achieve if the tasks had been learned separately.

4.3 Representation Learning

Representation learning is a special case of bias learning that's especially relevant for deep learning techniques. In this view, the bias is modeled specifically as a transformation of the input data, a representation, that is shared across the M learning tasks.

Baxter (1995) provides a basic framework for formally understanding the mechanics of representation learning. To enable the learning system to take advantage of the disperse datasets, the hypothesis space $\mathcal{H}$ is split into two parts $\mathcal{F} = \{f \mid f : \mathcal{X} \to \mathcal{V}\}$ and $\mathcal{G} = \{g \mid g : \mathcal{V} \to \mathcal{Y}\}$ where $\mathcal{V}$ is an arbitrary set. We achieve this by defining each function $h \in \mathcal{H}$ as a composition of two functions, i.e $h = g \circ f$ where $f \in \mathcal{F}$ and $g \in \mathcal{G}$. $\mathcal{F}$ is called the **representation space**, and $\mathcal{G}$ is called the **output space**.

According to Baxter (2000), the objective of representation learning is to find a good representation $f \in \mathcal{F}$, which is shared between each of the output functions g_1 to g_M. To formalize this, he introduces the notation $\mathbf{g} \circ f$ which denotes the composition of a vector $\mathbf{g}$ of M output functions with the representation function f as $\mathbf{g} \circ f = [g_1 \circ f, \ldots, g_M \circ f)]$. A good representation f is then a function which re-

duces the generalization error $E(\mathbf{g} \circ f)$:

$$E(\mathbf{g} \circ f) = \frac{1}{M} \sum_{m=1}^{M} E_m(g_m \circ f)$$

In words, the generalization error of f is the average single task generalization error over the M tasks, where the tasks share a common representation f.

Since $\mathcal{P}$ is unknown, we can only estimate the true generalization by an empirical error measure, $\hat{E}(\mathbf{g} \circ f, \mathcal{D}_M)$:

$$\hat{E}(\mathbf{g} \circ f, \mathcal{D}_M) = \frac{1}{M} \sum_{m=1}^{M} \hat{E}(g_m \circ f, \mathcal{D}_m)$$

In words, the average empirical error over each task and each training sample for each task, using a common representation for all tasks.

Baxter (1995) is able to show that if the tasks are learnt by minimizing $\hat{E}$ with a common representation f, we can decrease the number of examples N for each task required to ensure that $\hat{E}$ will be close to E with high probability by increasing the number of tasks M. In other words, learning with a common representation reduces the gap between training error and generalization error. As is the case with the bound presented in Baxter (1995) however, this result only bounds the average distance between $\hat{E}$ and E. In other words, it doesn't tell us when multi-task representation learning is beneficial in an absolute sense, in other words, if representation learning can reduce the complexity of $\mathcal{H}$ and $\hat{E}$ simultaneously as compared to single task learning.

As we will see in section 4.5, the potential advantages of multi-task learning with neural networks are enabled precisely by a shared representation. This makes the contribution of Baxter (1995) important since it provides a theoretical basis for deep multi-task learning.

4.4 Task Relatedness

Our presentation of multi-task learning so far has been limited to the statistical properties of learning multiple tasks simultaneously without consideration as to how the tasks are related to one another. Intuitively, we expect that learning related tasks should yield better results than learning unrelated tasks.

Ben-David et al. (2003) attempts to quantify this intuition by extending the work of Baxter (2000) with a notion of task "relatedness". They focus on modeling similarity between the M distributions P_1 to P_M from which the M datasets $D_m \in \mathcal{D}_M$ are drawn.

Specifically, they consider two learning tasks, defined by the probability distributions P_1 and P_2 on the input space $\mathcal{X}$, to be related if P_1 and P_2 are identical up to a transformation $f : \mathcal{X} \to \mathcal{X}$. To formalize this, they define a set of such transformations $\mathcal{F}$, and say that two learning tasks are $\mathcal{F}$-related if for some fixed distribution,

the data in each of these tasks are generated by applying some $f \in \mathcal{F}$ to that distribution.

Formally, let $\mathcal{F}$ be a set of transformations $f : \mathcal{X} \to \mathcal{X}$, and P_1, P_2 be probability distributions over $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{Y} = \{0, 1\}$. P_1 and P_2 are $\mathcal{F}$-related if there exists some $f \in \mathcal{F}$ such that for any $T \subseteq \mathcal{X} \times \mathcal{Y}$, T is P_1-measurable iff $f[T] = \{(f(\mathbf{x}), \mathbf{y}) \mid (\mathbf{x}, \mathbf{y}) \in T\}$ is P_2-measurable and $P_1(T) = P_2(f[T])$. Two samples are $\mathcal{F}$-related if they are sampled from $\mathcal{F}$-related distributions (Ben-David et al., 2003).

In the framework of Ben-David et al. (2003), they assume that the learning system knows the set $\mathcal{F}$ but doesn't know which function $f \in \mathcal{F}$ relates the distributions the system is learning from. Therefore, the ease with which the learner can transfer information about the underlying distributions from one learning task to another depends on the size of $\mathcal{F}$. The larger this set is, the looser the notion of relatedness between the learning tasks.

In order to let the learning system take advantage of the multiple datasets $\mathcal{D}_M$, Ben-David et al. (2003) uses their notion of task relatedness to reduce the complexity of the hypothesis space $\mathcal{H}$ by first using all the data $\mathcal{D}_M$ to select a subspace of $\mathcal{H}$ which is likely to contain good solutions to the set of learning problems. After this initial biasing of $\mathcal{H}$, M functions h_m are selected from this subspace for each learning problem. Specifically, from the hypothesis space $\mathcal{H}$, create a family of hypothesis spaces H of sets of hypotheses $h \in \mathcal{H}$ that are equivalent up to transformations in $\mathcal{F}$, assuming that for each $f \in \mathcal{F}$ and $h \in \mathcal{H}$, we have $h \circ f \in \mathcal{H}$.

To formalize this, Ben-David et al. (2003) define an equivalence relation $\sim \mathcal{F}$ on $\mathcal{H}$. This means that h_1 and h_2 are equivalent if there exists $f \in \mathcal{F}$ such that $h_2 = h_1 \circ f$. They use the notation $[h]_{\sim \mathcal{F}}$ to mean the equivalence class of h under $\mathcal{F}$.

Ben-David et al. (2003) uses the notion of equivalence classes to partition $\mathcal{H}$ into the family H of equivalence classes of $\mathcal{H}$ under $\mathcal{F}$, i.e $H = \mathcal{H} \backslash \sim \mathcal{F}$

Note that if two learning tasks defined by the distributions P_1 and P_2 are $\mathcal{F}$-related, then there exists $f \in \mathcal{F}$ such that the generalization errors of any function $h \in \mathcal{H}$ on both tasks are equal. In other words, there exists $f \in \mathcal{F}$ such that:

$$E_1(h) = E_2(h \circ f)$$

This means that the equivalence classes of $\mathcal{H}$ perform equally well on the different tasks, when measured by:

$$E_m(H) = \inf_{h \in H} E(h)$$

Ben-David et al. (2003) uses this fact of equivalence classes to build on Baxter (2000) and shows that if the number of examples N in each learning task satisfy:

$$N = O\left(\frac{1}{\epsilon^2}\left(d(M, H)\log\frac{1}{\epsilon} + \frac{1}{M}\log\frac{1}{\delta}\right)\right)$$

Then, with probability $1 - \delta$, for any $1 \leq i \leq M$:

$$\left| E_i([h]_{\sim \mathcal{F}}) - \inf_{h_1, \ldots, h_M \in [h]_{\sim \mathcal{F}}} \frac{1}{M} \sum_{m=1}^{M} \hat{E}(h_m, \mathcal{D}_m) \right| \leq \epsilon$$

The main difference between this result and the one obtained by Baxter (2000) is that Ben-David et al. (2003) bounds the distance between $E_m([h]_{\sim\mathcal{F}})$, i.e the generalization error of the equivalent functions $[h]_{\sim\mathcal{F}}$ for *any* task m, and the functions that minimizes the training errors for each $\mathcal{D}_m$, whereas Baxter (2000) bounds the distance between the *average* generalization error and training error.

This is an important result because it gives credence to our intuition that learning related tasks improves the guarantees that can be made on the distance between training and generalization error over learning unrelated tasks. However, just as the bound provided by Baxter (2000), this bound does not reveal anything about how learning from $\mathcal{D}_M$ might improve $\hat{E}(h_m, \mathcal{D}_m)$. Moreover, the range of domains where tasks are $\mathcal{F}$-related are limited. Specifically, the notion of $\mathcal{F}$-relatedness is limited to domains where two tasks are essentially two different views of the same data, for example video footage of the same scenery from two different perspectives. This may not be an appropriate assumption for natural language processing tasks.

4.5 Deep Multi-Task Learning

Neural networks have the advantage of being easy to adapt from single-task learning to multi-task learning. The simplest way of turning two single task learning problems into a multi-task learning problem is by hard weight sharing of the weights of layers of a neural network architecture between learning tasks and learning them simultaneously (Caruana, 1997). As an example, consider figure 4.1 and 4.2.

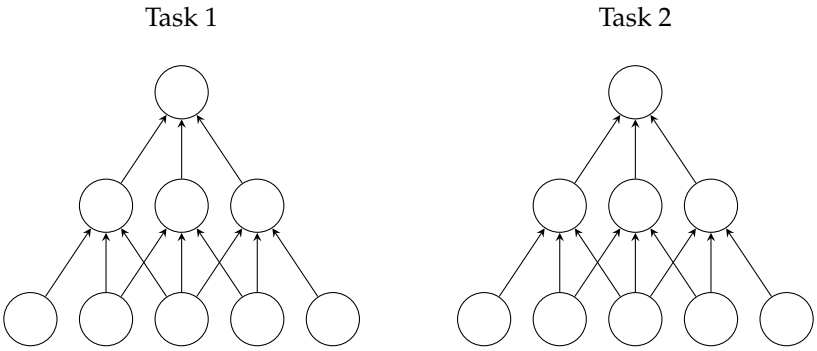


Figure 4.1
Visual representation of single-task learning with neural networks. A set of neural network weights are learnt separately for Task 1 and Task 2.

Multi-task learning techniques that are based on sharing neural network weights between tasks are collectively known as **deep multi-task learning** techniques. Deep multi-task learning is closely associated with the idea of representation learning presented in section 4.3 and the more general framework of bias learning presented in section 4.2. The network represents a shared representation $f(\mathbf{x})$ represented by S shared layers, often the first layers, and hypotheses $h_1 = (g_1 \circ f)(\mathbf{x})$ to $h_M = (g_M \circ f)(\mathbf{x})$ for each learning task, where g_m is $L - S$ neural network layers specific

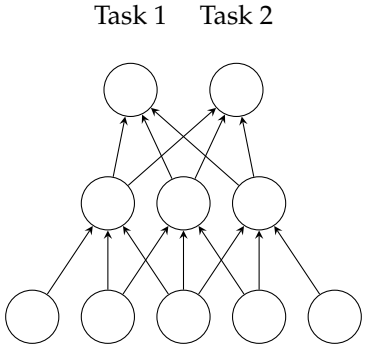


Figure 4.2
Visual representation multi-task learning with neural networks. The weights of the first layer is shared between the two tasks.

to each task.

The exact circumstances under which deep multi-task learning leads to lower overall generalization error compared to deep single-task learning are not yet theoretically well understood. Caruana (1997) lists 3 suggestions for how multi-task learning can reduce generalization error:

Statistical Data Amplification The effective number of training examples available to a deep multi-task learning system is increased due to the examples in the auxiliary data. The extensions to the Vapnik-Chervonenkis bound seen in the preceding sections gives us confidence that this reduces the risk that generalization error is far away from training error.

Eavesdropping If a hidden layer feature is useful to both Task 1 and Task 2, but much easier to learn when learning Task 2, sharing the hidden layer between the two tasks is likely to reduce generalization error for Task 1.

Representation Bias If Task 1 and Task 2 share a common minimum in weight-space, learning the tasks with weight sharing biases the learning system to choose the shared minimum. This is effectively a form of regularization that forces the learning system to search for a good hypothesis in a hypothesis space that is restricted to hypotheses that are useful for more than one task.

Baxter (2000) applies his bias learning framework to the case where the hypothesis space family $\mathbb{H}$ is constructed of neural networks where the first two layers are shared between tasks.

Specifically, a feature map $\phi_{\mathbf{w}} : \mathbb{R}^d \mapsto \mathbb{R}^{d^{(2)}}$ is a two layer neural network parameterized by $\mathbf{w}$ that maps an input vector $\mathbf{x} \in \mathbb{R}^d$ to feature vector $\phi_{\mathbf{w}}(\mathbf{x})$. Each feature $\phi_{\mathbf{w},j} \in \phi_{\mathbf{w}}$ is defined by:

$$\phi_{\mathbf{w},j}(\mathbf{x}) = \sigma\left(w_{0j} + \sum_{i=1}^{d^{(1)}} w_{ij} h_i(\mathbf{x})\right)$$

h_i is the output of unit i in the first layer, w_{ij} is the weight connecting unit $\phi_{\mathbf{w},j}$ and i and w_{0j} is the bias weight. For simplicity, Baxter (2000) considers only the binary

threshold activation function:

$$\sigma(a) = \begin{cases} +1 & \text{when } a \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

The output of each unit in the first layer h_j is computed as

$$h_j(\mathbf{x}) = \sigma\left(v_{0j} + \sum_{i=1}^{d} v_{ij} x_i\right)$$

where v_{ij} is the weight connecting the input feature x_i to unit j in the first layer and v_{0j} is a bias weight. The total number of weights W in these two layers is thus $W = d^{(1)}(d^{(0)} + 1) + d^{(0)}(d + 1)$. The space of all such feature maps $\{\phi_{\mathbf{w}} \mid \mathbf{w} \in \mathbb{R}^W\}$ can be thought of as the representation space $\mathcal{F}$ in the representation learning framework of Baxter (1995).

Baxter (2000) defines a hypothesis space $\mathcal{H}_{\mathbf{w}}$ as a set of binary decision functions on top the feature maps. Specifically:

$$\mathcal{H}_{\mathbf{w}} = \left\{ \sigma\left(a_0 + \sum_{i=1}^{d^{(2)}} a_i \phi_{\mathbf{w},i}\right) \;\middle|\; a_0, \ldots, a_{d^{(2)}} \in \mathbb{R} \right\}$$

where a_i is the weight connecting feature $\phi_{\mathbf{w},i}$ and the output unit and a_0 is a bias weight. The set of all such hypothesis spaces can be considered a hypothesis space $\mathbb{H}$ family such that:

$$\mathbb{H} = \{\mathcal{H}_{\mathbf{w}} \mid \mathbf{w} \in \mathbb{R}^W\}$$

Recall that the goal in bias learning is to select a vector $\mathbf{h}$ of M hypotheses h_m that minimizes the average generalization error $E(\mathbf{h}) = \frac{1}{M}\sum_{m=1}^{M} E_m(h_m)$. With the limitations described above, Baxter (2000) is able to show that in order for the average empirical error $\hat{E}(\mathbf{h}, \mathcal{D}_M)$ to be within ϵ of the average generalization error $E(\mathbf{h})$ with probability $1 - \delta$, it suffices that the number of examples N per task satisfies:

$$N \geq O\left(\frac{1}{\epsilon^2}\left(\frac{W}{M} + d^{(2)} + 1\right)\log\frac{1}{\epsilon} + \frac{1}{M}\log\frac{1}{\delta}\right)$$

Ignoring the confidence parameters ϵ and δ, the sample complexity bound tells us that learning complicated neural network representations where $d^{(1)}$ and $d^{(2)}$ and therefore also W are large, is harder than learning simple representations in the sense that it requires more samples to succeed. The benefit gained by multi-task learning is that we can reduce N by increasing M, an option we don't have in the single task learning setting. This means we can afford to learn more complicated representations in the hope that this can lead to lower training error and still have high confidence that generalization is possible by increasing M.

Part 5

Experiment

In this section we explain our experimentation with multi-task learning for relation classification. We begin by describing related work on deep multi-task learning for natural language processing tasks and relation extraction. Our focus is to investigate the impact of multi-task learning on generalization error on a target task introduced in the first section. We continue by describing each auxiliary task. Finally, we describe the neural network architecture and algorithm used for generating empirical data.

5.1 Related Work

To our knowledge, there hasn't been any experimental work done on deep multi-task learning for relation classification despite the popularity of these techniques in natural language processing in general. The work of Jiang (2009) is perhaps most closely related to our own. She demonstrates a technique for sharing weights between logistic regression classifiers trained for binary classification of a target relation for which there is only a small number of seed training instances.

The input to these binary classifiers are engineered linguistic features based on syntactic and dependency parse trees. She finds that sharing components of the weight vectors for each of these classifiers improves the F1 score on a validation set compared to learning the weight vectors in a single-task fashion.

Moreover, she finds that the benefit of weight sharing decreases as the number of seed instances is increased, in other words, weight sharing actually hurts performance in this setting when the number of training examples is greater than 1000.

Collobert et al. (2011) is one of the earliest works that detail how to solve natural language processing tasks with neural networks. They focus on neural networks for sequence prediction problems. Specifically, they use two different neural network architectures for predicting tags for words by considering a window of neighboring words around the target word, transform this window into a matrix of concatenated word-vectors and feed this into a neural network. The major contribution of this technique was to show that state-of-the-art performance was possible for these tagging tasks without manual feature engineering, but simply by learning word vectors and convolutional filters directly from the words.

Collobert et al. (2011) also experimented with deep multi-task learning through hard weight sharing and found that weight sharing in general increased performance on most of the tasks they considered.

Bingel and Søgaard (2017) performs a thorough experiment on deep multi-task learning using recurrent neural networks for sequence prediction problems in natural language processing. They run experiments of 90 different configurations of target and auxiliary task and compare generalization error for the target task when compared to single-task learning. Their goal is to investigate correlation between dataset statistics and characteristics in single-task learning that are good predictors for gains in multi-task learning.

They find that the best predictor for multi-task learning gains is single-task learning curves over gradient descent iterations. Specifically, if the target task generalization error quickly plateaus, learning it simultaneously with an auxiliary task that doesn't plateau is likely to improve generalization. They speculate this happens because the inclusion of an auxiliary task can move neural network weights out of local minima during training.

Neural networks have also been tested for sentence classification problems such as relation classification. Kim (2014) design a convolutional neural network for sentence classification and tests it on a number of sentence classification tasks such as sentiment analysis and semantic category prediction in a single-task learning setting. His architecture achieves state-of-the-art performance on a number of these tasks without any manual feature engineering.

Nguyen and Grishman (2015) and adapts the work of Kim (2014) to the relation classification problem. Their addition to the architecture is a mechanism for marking the relation arguments in the input. This leads to state-of-the-art performance. They do not consider multi-task learning.

Zhang and Wang (2015) investigate the performance of recurrent neural networks on the relation extraction problem. They suggest a simple bi-directional recurrent architecture where the relation arguments are marked simply by special tokens that indicate the beginning and end of each argument. With this architecture they achieve almost state-of-the-art results close to the results of Nguyen and Grishman (2015). They find that for sentences where the relation arguments are far apart, the recurrent architecture outperforms the convolutional architecture.

5.2 Target Task

We have chosen a target relation classification dataset which will act as a benchmark across our experiments in order to empirically investigate the dynamics of sample complexity for multi-task relation classification. The SemEval 2010 Task 8 dataset has arguably become somewhat of a standard for relation classification papers, and so is a reasonable choice for the role of target task in this context (Hendrickx et al., 2009).

The SemEval 2010 Task 8 dataset consists of 10,717 English sentences. Each sentence is annotated with exactly one of the following semantic relations:

Cause-Effect An event or object leads to an effect. Example: *those [cancers] were caused by radiation [exposures].*

Instrument-Agency An agent uses an instrument. Example: *[phone] [operator].*

Product-Producer A producer causes a product to exist. Example: *a [factory] manufactures [suits].*

Content-Container An object is physically stored in a delineated area of space. Example: *a [bottle] full of [honey] was weighed*

Entity-Origin An entity is coming or is derived from an origin (e.g., position or material). Example: *[letters] from foreign [countries].*

Entity-Destination An entity is moving towards a destination. Example: *the [boy] went to [bed].*

Component-Whole An object is a component of a larger whole. Example: *my [apartment] has a large [kitchen].*

Member-Collection A member forms a nonfunctional part of a collection. Example: *there are many [trees] in the [forest].*

Message-Topic A message, written or spoken, is about a topic. Example: *the [lecture] was about [semantics].*

Other Any other relation.

SemEval 2010 Task 8 is not a traditional relation classification task. In particular, the objective of traditional relation extraction is to identify semantic relationships between named entities. In the SemEval dataset however, the annotated relationships are between head words of nominal phrases, for example *Message-Topic(lecture, semantics)* in the sentence *the lecture was about semantics* where neither *lecture* nor *semantics* are named entities.

Moreover, the annotation process for the SemEval dataset includes some restrictions which are designed to make the resulting learning problem easier. Firstly, the annotators exclude sentences where relationships depend on discourse knowledge, for example when one of the arguments are pronouns. Secondly, sentences where the relation arguments occur in different sentential clauses are excluded. For example, we could argue that the relationship *Instrument-Agency(man, unicycle)* exists in the sentence *the man, who rides a unicycle, came to see me.*, but since the arguments occur in different sentential clauses it would be excluded from the SemEval dataset.

The annotators aimed for a uniform distribution of relations in the SemEval dataset. To this end, they initially collected approximately 1200 sentences for each relation category by pattern based web search. This ultimately lead to the distribution shown in figure 5.1. One consequence of this selection procedure is that the label distribution does not follow the distribution of relations found in natural language data in the wild.

Ideally, we'd like to be able to make conclusions about the usefulness of multi-task

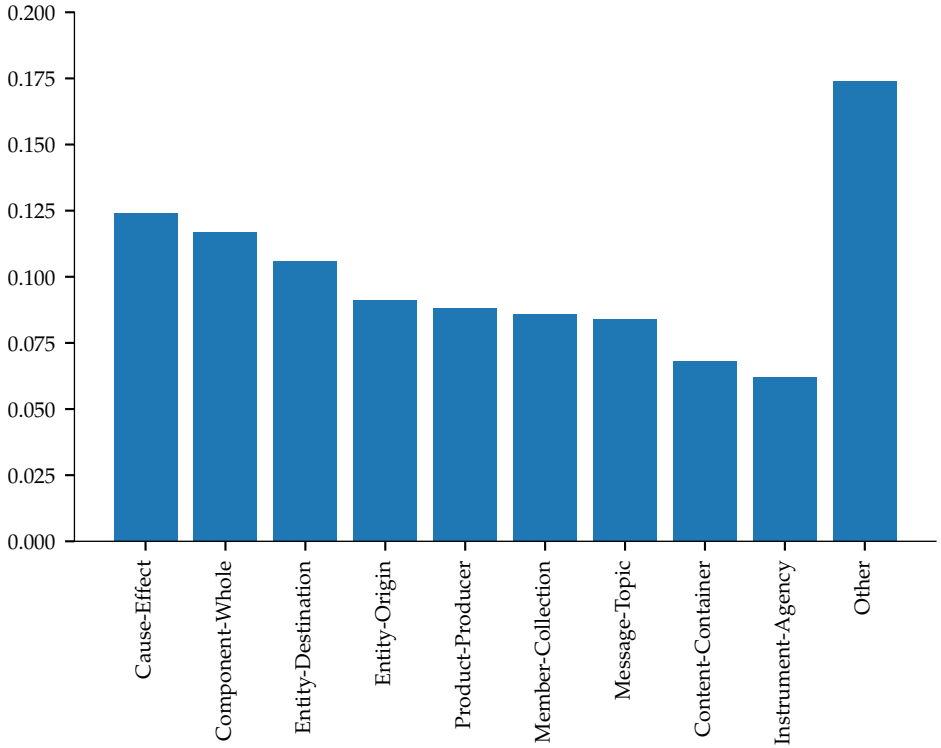


Figure 5.1
Label distribution of SemEval 2010 Task 8.

learning for relation classification in general and not just on SemEval 2010 Task 8 based on the results in this thesis. This is possible to the extent that the SemEval data is a realistic sample and that the target relations are general enough that they are useful for a wide range of domains. As discussed above, both of these qualities can be called into question.

Handschuh et al. (2016) provides a brief discussion of the generality of SemEval 2010 Task 8. In addition to the limitations we have already pointed out, they highlight that the relations in the SemEval dataset are mainly concerned with relations between concrete, physical objects. Taken together, the limitations of SemEval 2010 Task 8 could indicate that a need exists for authoring a more general relation classification task that is more appropriate as a benchmark. At this time however, we use the SemEval dataset as target task in our experiments because of its prevalence in the research literature, despite these points of criticism.

5.3 Auxiliary Tasks

Here we describe each of the datasets used as auxiliary tasks in our multi-task learning experiment. We describe the goal of each task in some detail in order to reason about its potential benefit as an auxiliary task for SemEval 2010 Task 8.

5.3.1 ACE 2005 Relations

Intuitively, we expect that a feature transformation in the early layers of a neural network that's useful for one relation classification task should also be useful to another, assuming the relations of interest in one task are semantically related to the relations in another.

We therefore test the usefulness of incorporating an auxiliary relation extraction task, as measured by generalization error on the SemEval dataset. Next to SemEval, the ACE 2005 relation classification dataset is the most widely used in contemporary literature (Walker, 2006). Unlike the SemEval 2010 dataset, the ACE 2005 relation classification task is concerned with identifying relations between named entities. Specifically, the relations in the ACE 2005 dataset are defined between the entity types: person, organization, location, facility, weapon, vehicle and geo-political entity. Unlike the SemEval annotation process however, the annotation guide for ACE 2005 contains no restrictions on the complexity of the sentences in terms of dependence on discourse knowledge or sentential clauses.

The ACE 2005 dataset contains 8,365 english sentences collected from various sources such as transcribed news broadcasts and phone conversations, as well as Usenet discussion forums and Newswire. Each sentence is annotated with exactly one of the following relations:

> **Physical** Two entities are physically related. Example: *[Donald Trump] lives in [The White House].*
>
> **Part-Whole** One entity constitutes part of another. Example: *[Gibraltar] is territory of the [UK].*
>
> **Personal-Social** Entities are people with a social relation. Example: *[Darth Vader] is the father of [Luke Skywalker].*
>
> **Organization-Affiliation** A person is affiliated with an organization. Example: *[Ray Kroc] founded [McDonald's].*
>
> **Agent-Artifact** An entity is the agent of an artifact. Example: *[James Bond] drives an [Aston Martin DB5].*
>
> **Gen-Affiliation** General affiliation between two entities. Example: *[Mitt Romney] is a member of [the Mormon church].*

In truth, each of the relations above are further sub-categorized in the ACE 2005 corpus. For example,

There is clear semantic overlap between the relation categories of the SemEval dataset and the ACE dataset, for example for the categories *Agent-Artifact* and *Physical* in the ACE corpus and *Agent-Artifact* and *Entity-Origin* in the SemEval corpus. We can speculate that a neural network representation that's useful for one task may be useful for the other which may lead to improved generalization error on the target task.

Specifically, the first layer of a neural network for a natural language processing task is usually a word embedding layer which maps words into word-vectors that encode semantic similarities between words. We can speculate that if a word is highly informative for predicting the *Agent-Artifact* relation in the ACE corpus, this

will be encoded in the word embedding or a neural unit on top of the word embedding through neural network training, and can eventually benefit the SemEval task if this representation is shared between them.

5.3.2 CONLL2000 Part-of-Speech

Part-of-speech tagging is the task of assigning part-of-speech tags such as noun, verb etc. to word tokens (Jurafsky and Martin, 2009). Part-of-speech tags are known to be a useful input feature for a number of other supervised machine learning systems for natural language processing tasks, here-among named entity recognition and relation extraction. This is believed to be the case since word classes are highly informative of a word's semantic role in a sentence (Jurafsky and Martin, 2009).

Several part-of-speech tagging schemes exists. The universal tag set is a simple and commonly used scheme which contains 12 different tags:

VERB Verbs (all tenses and modes)

NOUN Nouns (common and proper)

PRON Pronouns

ADJ Adjectives

ADV Adverbs

ADP Adpositions (prepositions and postpositions)

CONJ Conjunctions

DET Determiners

NUM Cardinal numbers

PRT Particles or other function words

X Other - foreign words, typos, abbreviations.

. Punctuation

Part-of-speech tagging can be seen as sequence labeling problem. The goal is to assign a tag to each token in a sentence. For example:

I	saw	the	man	with	the	telescope	.
PRON	VERB	DET	NOUN	ADP	DET	NOUN	.

Figure 5.2
A sentence tagged with universal part-of-speech tags.

The CONLL2000 dataset was produced as a shared task for the year 2000 Conference on Computational Natural Language Learning (Tjong Kim Sang and Buchholz, 2000). It contains 10,948 sentences with 259,104 tokens from the Wall Street Journal section of the Penn Treebank (et al., 1999). The part-of-speech tag for each token is

supplied not by a human annotator, but from an automatic tagging system called the Brill tagger (Brill, 1992).

We speculate that a neural network representation that's useful for part-of-speech tagging will also be useful for relation classification. In particular, if word vectors or neural network features encode information about part-speech-tags it may help to resolve ambiguity for words that are crucial for identifying semantic relations, such as words that are verbs in some contexts but nouns in others.

5.3.3 CONLL2000 Chunking

Assigning structure to a sentence is generally known as parsing. Syntactic parsing is a fundamental task in natural language processing which involves segmenting a sentence into a hierarchical structure that captures its syntactic elements (Jurafsky and Martin, 2009). Consider figure 5.3 as an example.

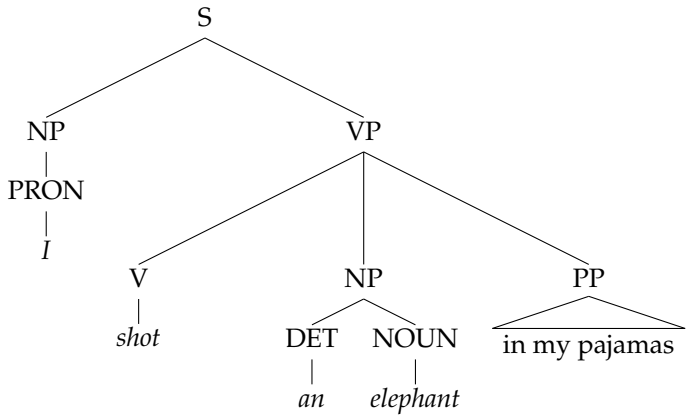


Figure 5.3
Syntactic parse tree for the sentence: I shot an elephant in my pajamas.

Many practical applications do not require full syntactic parses. **Chunking** is a simpler partial parsing technique that can often be used as an alternative. The goal of chunking is to identify the flat, non-overlapping parts of a sentence that constitute its major non-recursive phrase structures. Se for example figure 5.4.

The CONLL2000 dataset is annotated with chunking information in the BIO-labelling scheme introduced in section 2.1.1 in addition to part-of-speech tags.

A neural network representation that's useful for predicting syntactic chunks may

I	shot	an	elephant	in	my	pajamas	.
B-NP	B-VP	B-NP	I-NP	B-PP	I-PP	I-PP	O

Figure 5.4
Chunks of the sentence: I shot an elephant in my pajamas, *annotated with BIO-labelling.*

benefit a relation classification task since the syntactic structure of a sentence is highly informative for how nominals, including named entities, are semantically related to each other. For example, in order to determine whether the relationship between *The Ford Motor Company* and *Dearborn, Michigan* in the sentence *The Ford Motor Company produces cars in Dearborn, Michigan* is *Entity-Origin* and not for example *Product-Producer*, it's useful to know that *cars* is a noun-phrase where as *Dearborn, Michigan* is prepositional phrase, and therefore not the object of *produces*.

5.3.4 GMB Named Entity Recognition

We described the named entity recognition problem in section 2.1.1. Groningen Meaning Bank is a corpus annotated with various semantic information such as named entity information, developed at University of Groningen (Basile et al., 2012). The corpus contains 62,010 sentences annotated with the following named entity types:

Person Individuals that are human or have human characteristics, such as divine entities.

Location Geographical entities such as geographical areas and landmasses, bodies of water, and geological formations.

Organization Corporations, agencies, and other groups of people defined by an established organizational structure.

Geo-Political Entity Geographical regions defined by political and/or social groups. A GPE entity subsumes and does not distinguish between a city, a nation, its region, its government, or its people.

Artifact Manmade objects, structures and abstract entities, including buildings, facilities, art and scientific theories.

Natural Object Entities that occur naturally and are not manmade, such as diseases, biological entities and other living things.

Event Incidents and occasions that occur during a particular time.

Time References to certain temporal entities that have a name, such as the days of the week and months of a year.

Even though SemEval 2010 Task 8 is not explicitly concerned with classifying relationships between named entities, we can speculate that neural network features that are useful for predicting named entity types is also useful for predicting semantic relations for the SemEval task. For example, a representation that predicts that *forest* is more likely to be a location than a person in the sentence *there are many trees in the forest* is likely to be useful by increasing confidence for appropriate relation types for locations.

5.4 Neural Network Architecture

Our neural network architecture for relation classification is based on Nguyen and Grishman (2015). We initially chose this architecture since it achieves the best results

that we've encountered in the research literature on SemEval 2010 Task 8, which would make our findings relevant to state-of-the-art research. This architecture is not appropriate for sequence labeling tasks however. As a consequence, we use a related but slighter different architecture based on Collobert et al. (2011) for the sequence labeling tasks and share neural network weights of the early layers between the two architectures.

In the architecture described in Nguyen and Grishman (2015) each word s_i of an input sentence s is first mapped to a word-vector $\mathbf{v}_i \in \mathbb{R}^d$ through a word-embedding matrix to form a sentence matrix $\mathbf{S}$. In our experiment, we initialize the word-embedding matrix with d dimensional GloVe vectors trained on the Common Crawl corpus (commoncrawl.org). To ensure that the dimensionality of $\mathbf{S}$ is consistent across sentences, we compute the longest sentence length K of the sentences in the SemEval and ACE corpora and pad shorter sentences with a padding token as a preprocessing step, such that $\mathbf{S} = [\mathbf{v}_1, \ldots, \mathbf{v}_K]^T$.

To indicate which words in the input sentence are the relation arguments, we compute the distance between each word index i and the index of the first relation argument words $e1$ and $e2$ as $i - e1$ and $i - e2$. These distances are mapped into real valued position vectors $\mathbf{p}_{1i}$ for the distance $i - e1$ for each word, and $\mathbf{P}_{2i}$ for the distance $i - e2$ for each word, both in $\mathbb{R}^{d'}$. From these we construct the position matrices $\mathbf{P}_1 = [\mathbf{p}_{11}, \ldots, \mathbf{p}_{1K}]^T$ and $\mathbf{P}_2 = [\mathbf{p}_{21}, \ldots, \mathbf{p}_{2K}]^T$. We use the three matrices $\mathbf{S}$, $\mathbf{P}_1$ and $\mathbf{P}_2$ to form the augmented sentence matrix $\mathbf{S}' = [\mathbf{S} \mid \mathbf{P}_1 \mid \mathbf{P}_2] \in \mathbb{R}^{K \times (d+2d')}$.

The $K \times (d + 2d')$ dimensional augmented sentence matrix is used as input for a convolutional neural network layer. The convolution filters are applied over the full height of augmented sentence matrix in windows of n tokens over the K dimensional axis. In other words, each convolution filter is a weight matrix $\mathbf{W} \in \mathbb{R}^{n \times (d+2d')}$. The output of the convolutional neural at position i is:

$$\sigma\left(w_0 + \sum_{j=i}^{i+n} \mathbf{S}'_i \mathbf{W}_i^T\right)$$

Where σ is the ReLU activation function, $\mathbf{W}_i$ and $\mathbf{S}'_i$ is the i'th row of the convolutional filter matrix and augmented sentence matrix respectively, and w_0 is a bias term. In our experiment we use 150 filters of each window size, and window sizes n of 2, 3, 4 and 5 for a total of 600 convolutional filters.

We apply max-pooling to the output of each convolutional filter yielding a 600 dimensional feature vector, which is used as input for a soft-max output layer. See figure 5.5 for a diagram.

For the sequence classification tasks we use a convolutional neural network architecture based on Collobert et al. (2011). This architecture is virtually identical to the architecture used for the relation classification task, with the exception of the position features. To predict the tag for word s_i in the sentence s, a window of K tokens around s_i is transformed into a sentence matrix $\mathbf{S} = [\mathbf{v}_{i-\frac{K}{2}}, \ldots, \mathbf{v}_i, \ldots, \mathbf{v}_{i+\frac{K}{2}}]^T \in \mathbb{R}^{K \times d}$ where $\mathbf{v}_i \in \mathbb{R}^d$ is taken from a word-embedding matrix. For words where $i \pm K/2$ exceeds the sentence border a padding token is used. The sentence matrix $\mathbf{S}$ is used directly as input to the convolutional layer. In other words, the convolutional

filter weights $\mathbf{W}$ for sequence classification tasks have dimensionality $n \times d$, where n is the convolution filter window size.

The subtle differences between the network architecture used for sequence and relation classification leads to some practical difficulties. Since most symbolic differentiation software used for neural network training such as TensorFlow or Theano use matrix-vector formulations, neural network weights must be expressed as matrices in these frameworks (Abadi et al., 2016; Theano Development Team, 2016). This means that the convolutional filter weights $\mathbf{W}$ cannot be shared between the sequence classification tasks and the relation classification tasks, since the filters have dimensionality $K \times d$ in one and $K \times d + 2d'$ in the other.

For this reason, we adopt the strategy of Collobert and Weston (2008) and share only the word-embedding matrix between the relation classification tasks and sequence classification tasks. A diagram of neural network weights are shared between tasks can be seen in figure 5.6

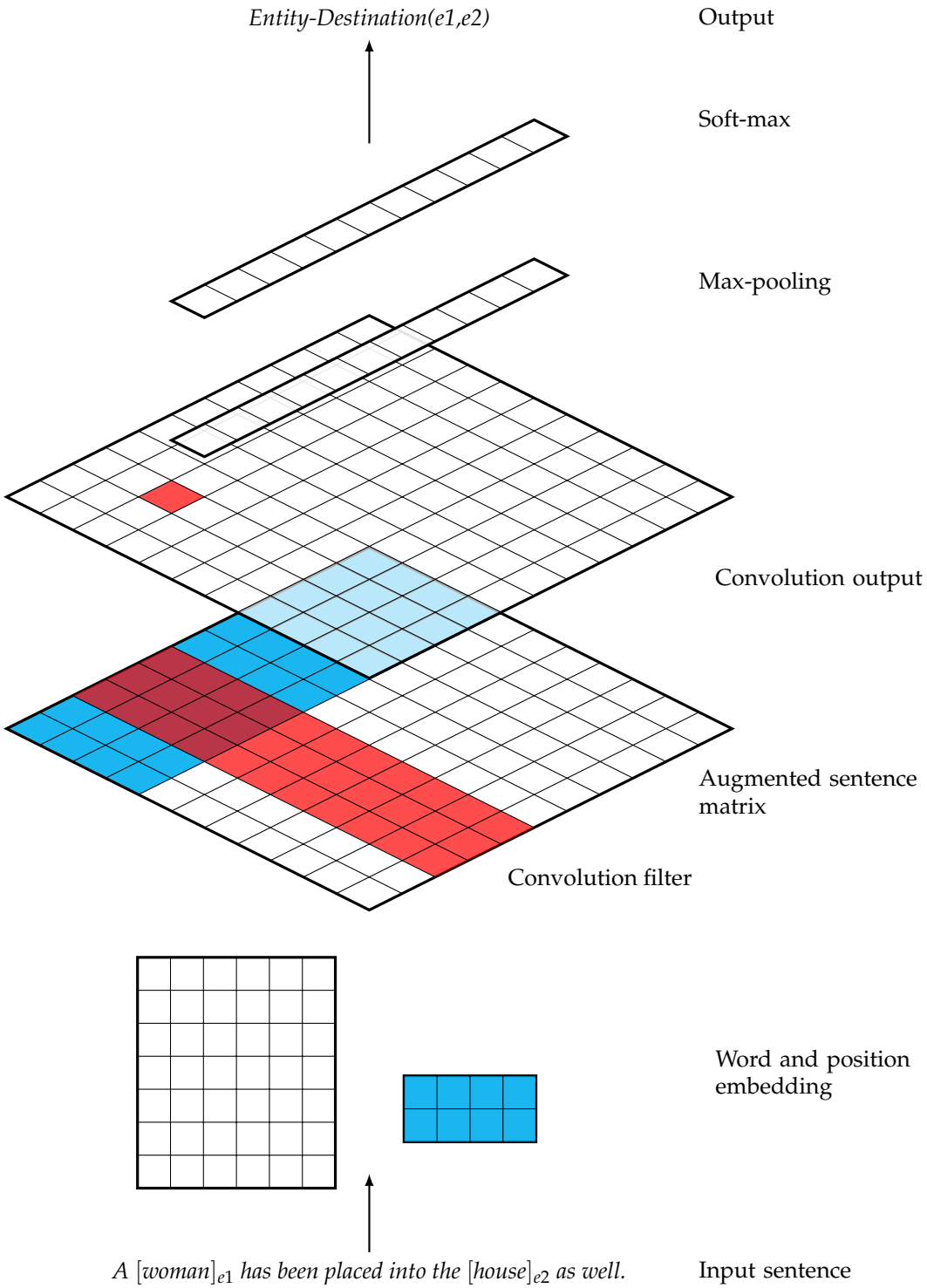


Figure 5.5
Diagram of the convolutional neural network for relation classification. The input sentence is mapped to a sentence matrix by concatenating the word-vector for each word in the word embedding matrix and the position-vector for each word-position in the position embedding matrix for both entities. The position embedding components of the architecture is shown in blue. This forms a $K \times d + 2d'$ matrix. Convolution filters are applied along the K-axis of the sentence matrix to produce the convolution output. Each element in the convolution output matrix corresponds to one convolution filter applied at one position of the sentence matrix. An example convolutional filter and its corresponding output unit is shown in red. Max-pooling is applied to the convolution output to obtain a feature-vector. This vector is used as input to a soft-max output layer.

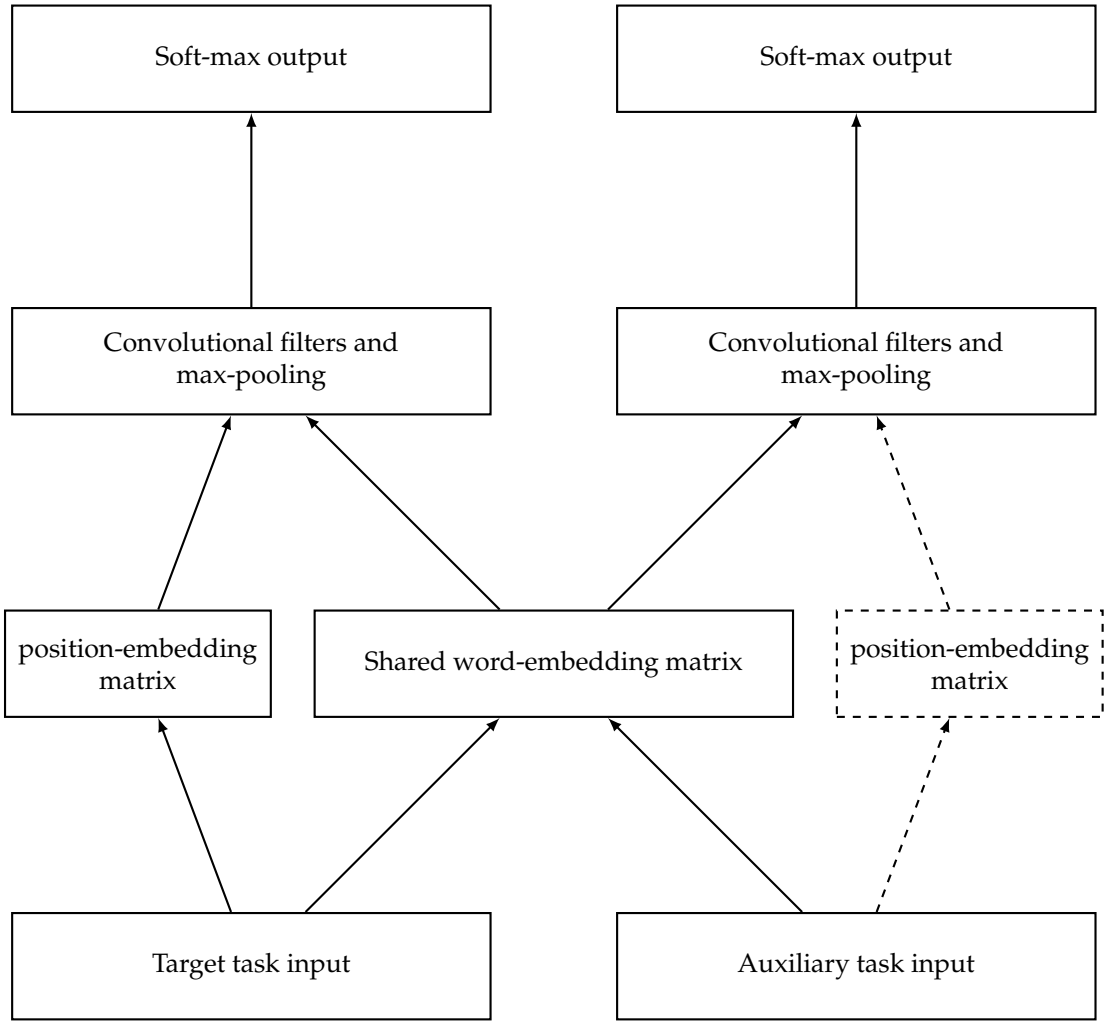


Figure 5.6
Diagram of how neural network weights are shared between the auxiliary and target task. The word-embedding matrix is shared between tasks, but the convolution filters and potential position embeddings for the auxiliary task are not.

5.5 Algorithm

Our main goal is to investigate the sample complexity dynamics of learning a relation classification task in a multi-task learning setting. To this end, we compare the generalisation error of a deep learning model trained only on SemEval 2010 Task 8, the target task, with the generalisation error of a deep learning model trained jointly on the SemEval data and one of the auxiliary tasks described in 5.3.

We proceed as follows: We vary the amount of data from the target task and auxiliary task in turn by a set of fractions. For every combination of fractional target and auxiliary data, we perform 5-fold cross validation on the target data to yield 5 macro-F1 scores. We use the training data from the 4 training folds of target data and the auxiliary data to train the architecture described in 5.4. This is done by uniformly selecting one of the two tasks, sampling a mini-batch of the fractional training data from that task, and performing one gradient descent update with respect to cross-entropy error using the Adam algorithm described in section 3.2.2.

This process is iterated until an early stopping criterion on the target training data is met. Specifically, 1/10 of target training data is set aside for early stopping validation. When the cross-entropy error on the early stopping dataset has not improved for 200 iterations of mini-batch gradient descent, training is halted, and the model weights are reset to their best recorded value.

When the patience is exceeded we record the cross-validation macro-F1 on the target task test fold using the best recorded weights. Since neural network training is a random search procedure with respect to weight initialization and mini-batch sampling, we run this experiment for each combination of target and auxiliary fractional data 5 times, yielding a total of 25 random cross-validation splits for each combination. We have provided the algorithm used in our experiments as pseudocode in algorithm 1.

Algorithm 1 *Pseudocode for our deep multi-task learning experiment.*

Require: $miniBatchSize$: an integer giving the mini-batch size
Require: $targetData = \{(\mathbf{x}_{t1}, \mathbf{y}_{t1}), \ldots, (\mathbf{x}_{tN}, \mathbf{y}_{tN})\}$
Require: $auxiliaryData = \{(\mathbf{x}_{a1}, \mathbf{y}_{a1}), \ldots, (\mathbf{x}_{aN}, \mathbf{y}_{aN})\}$
Require: $sample(data, fraction)$: A routine that can sample $|set| \cdot fraction$ samples from the set $data$ uniformly.
Require: $crossValidation(data)$: A routine that returns a set of cross-validation folds over the set $data$.
Require: $initializeWeights()$: A routine that can initialize a neural network weight vector.
Require: $gradientDescent(data, w)$: A routine that performs gradient descent on the neural network weights w.
Require: $macroF1(w, data)$: A routine that computes the macro F1 score on $data$ using neural network weights w.
Require: $report(score, targetFraction, auxiliaryFraction)$: A reporting routine.

```
fractions ← {0/5, 1/5, 2/5, 3/5, 4/5, 1}
for iteration ∈ {1, ..., 5} do
    for targetFraction ∈ fractions do
        for auxiliaryFraction ∈ fractions do
            targetFractionalData ← sample(targetData, targetFraction)
            auxiliaryFractionalData ← sample(auxiliaryData, auxiliaryFraction)
            earlyStoppingData ← sample(targetFractionalData, 1/10)
            targetTrainData = targetFractionalData \ earlyStoppingData
            for trainFold, testFold ∈ crossValidation(targetTrainData) do
                w ← initializeWeights()
                while patience not exceeded do
                    task ← sample({trainFold, auxiliaryFractionalData}, 1/2)
                    miniBatch ← sample(task, |task| / miniBatchSize)
                    w ← gradientDescent(miniBatch, w)
                end while
                score = macroF1(w, testFold)
                report(score, targetFraction, auxiliaryFraction)
            end for
        end for
    end for
end for
```

Part 6

Results

In this section we present the results obtained in the experiment detailed in the previous sections. We begin by visualizing the results across all our experiments using a visualization technique called learning surfaces. We then examine specific results in details using hypothesis testing.

6.1 Learning Surfaces

A learning curve is a common technique for visualizing the empirical sample complexity dynamics for a single task. It's generated by varying the sample size N and recording either training or generalization error for each configuration of N. In the multi-task setting, the sample size may be varied in both the target and auxiliary direction and the learning curve becomes a learning surface.

In this section we present learning surfaces for each of our multi-task learning experiments. The x-axes show the fraction of auxiliary data used for model training and the y-axes show the fraction of target data used for model training. The z-axes show the mean macro-F1 computed for the validation target data for the 25 random splits created for each combination of fractional target and auxiliary data.

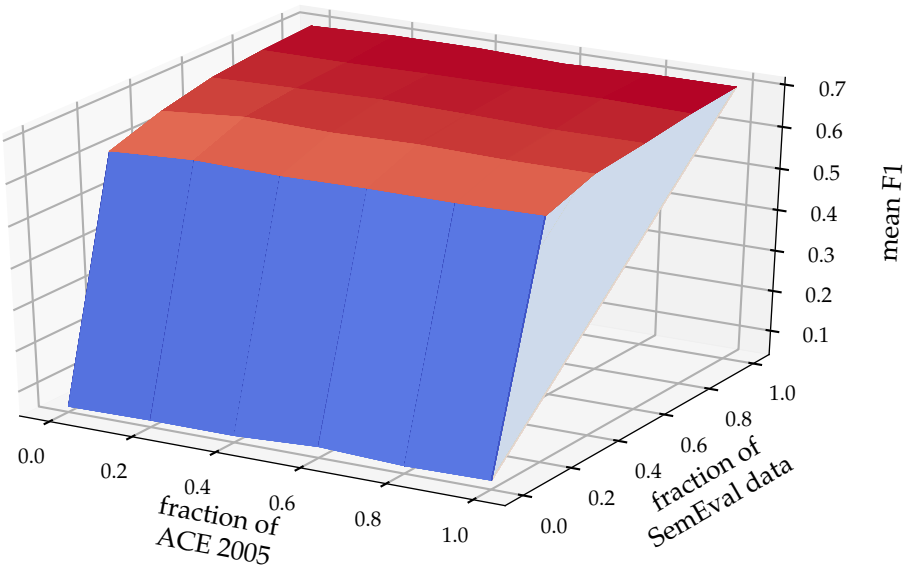


Figure 6.1
Learning surface for SemEval and CONLL2000 Chunk.

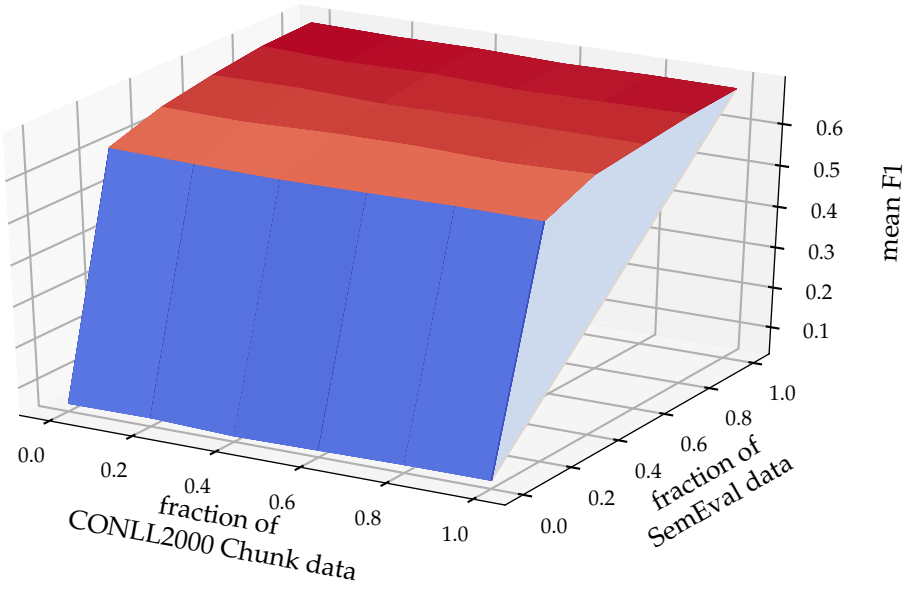


Figure 6.2
Learning surface for SemEval and CONLL2000 Chunk.

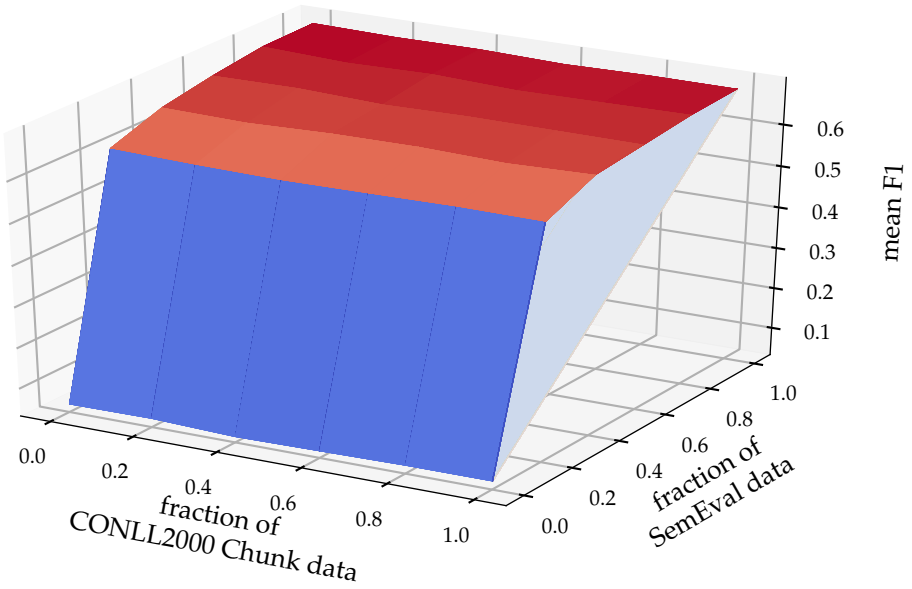


Figure 6.3
Learning surface for SemEval and CONLL2000 Chunk.

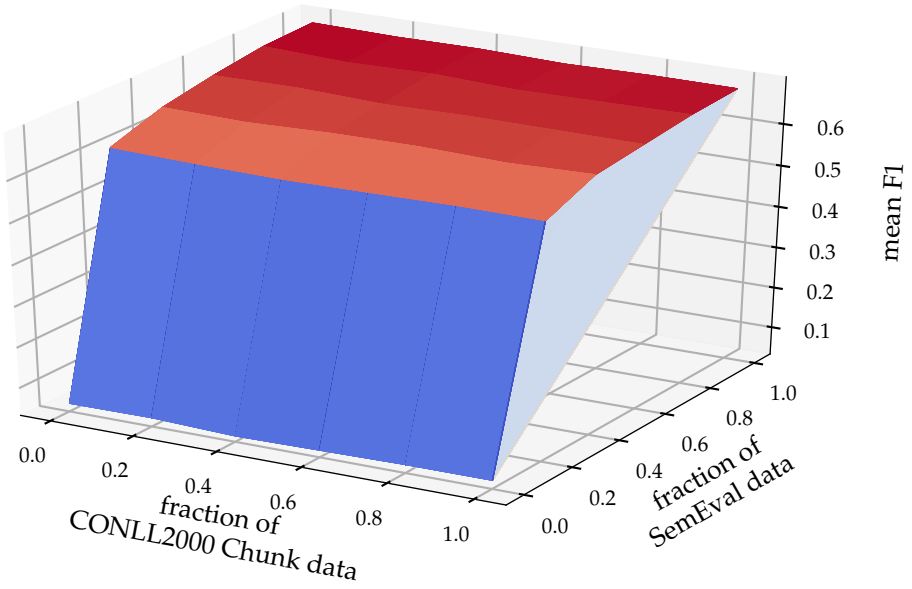


Figure 6.4
Learning surface for SemEval and CONLL2000 Chunk.

6.2 Hypothesis Testing

In section we compare the learning curves of single-task and multi-task learning. Specifically, for each fraction of target data in our experiment, we compare the mean F1 μ_{F1} of single-task learning to the mean F1 of multi-task learning using all the auxiliary data for each of the auxiliary tasks.

ACE 2005 Relations

Fraction of target data	.0	.2	.4	.6	.8	1.0
μ_{F1} **Single task**	0	0	0	0	0	0
σ_{F1} **Single task**	0	0	0	0	0	0
μ_{F1} **Multi-Task**	0	0	0	0	0	0
σ_{F1} **Multi-Task**	0	0	0	0	0	0
p**-value**	0	0	0	0	0	0

Figure 6.5
Significance test for difference between SemEval 2010 Task 8 learned as a single task and learned simultaneously with CONL2000 Chunk

CONLL2000 Chunk

Fraction of target data	.0	.2	.4	.6	.8	1.0
μ_{F1} **Single task**	0	0	0	0	0	0
σ_{F1} **Single task**	0	0	0	0	0	0
μ_{F1} **Multi-Task**	0	0	0	0	0	0
σ_{F1} **Multi-Task**	0	0	0	0	0	0
p**-value**	0	0	0	0	0	0

Figure 6.6
Significance test for difference between SemEval 2010 Task 8 learned as a single task and learned simultaneously with CONL2000 Chunk

CONLL2000 Part-of-speech

Fraction of target data	.0	.2	.4	.6	.8	1.0
μ_{F1} **Single task**	0	0	0	0	0	0
σ_{F1} **Single task**	0	0	0	0	0	0
μ_{F1} **Multi-Task**	0	0	0	0	0	0
σ_{F1} **Multi-Task**	0	0	0	0	0	0
p**-value**	0	0	0	0	0	0

Figure 6.7
Significance test for difference between SemEval 2010 Task 8 learned as a single task and learned simultaneously with CONL2000 Chunk

GMB Named Entity Recognition

Fraction of target data	.0	.2	.4	.6	.8	1.0
μ_{F1} **Single task**	0	0	0	0	0	0
σ_{F1} **Single task**	0	0	0	0	0	0
μ_{F1} **Multi-Task**	0	0	0	0	0	0
σ_{F1} **Multi-Task**	0	0	0	0	0	0
p**-value**	0	0	0	0	0	0

Figure 6.8
Significance test for difference between SemEval 2010 Task 8 learned as a single task and learned simultaneously with CONL2000 Chunk

Part 7

Discussion

In the previous section we saw that the particular type of weight sharing tested in our experiments did not lead to significant improvements in generalization error for relation classification. In this section, we reflect on this result in order to outline the conclusions we can draw.

7.1 Impact of Limited Weight Sharing

As discussed, the neural network architecture adopted from Nguyen and Grishman (2015) puts certain limitations on how neural network weights can be shared between tasks in practice. We believe the solution we chose, sharing only the word embedding weights, may reduce the effectiveness of multi-task learning. This choice was motivated by Collobert and Weston (2008) in which they show that sharing the word vector weights leads to significant improvements in generalization error for a semantic role labeling task.

However, all their learning tasks are derived from annotations of the same sections of the PropBank corpus (Kingsbury and Palmer, 2002). When annotations for auxiliary tasks are taken from different corpora, the potential benefits made possible by sharing only the word embedding is limited by the degree of overlap of words occurring in both corpora. The set of words that occur in a corpora is commonly referred to as the **vocabulary**. The only way an auxiliary can benefit the target task, is if the weights that are updated while learning from an auxiliary task are also used by the target task. When sharing only the word embedding, this happens only when a word is in both the auxiliary vocabulary and in the target vocabulary.

In Collobert and Weston (2008) the vocabulary overlap is maximal since all annotations for all tasks pertain to the same text. This is not the case for the corpora used in our experiments as seen in figure 7.1. We speculate that when combining text from different corpora for multi-task learning using word-vectors, it's more beneficial to share the convolutional filters than the word-embedding since the filters are guaranteed to be used by both tasks. Intuitively, one task may then benefit the other if the features detected by a filter learnt by one task is useful for the other.

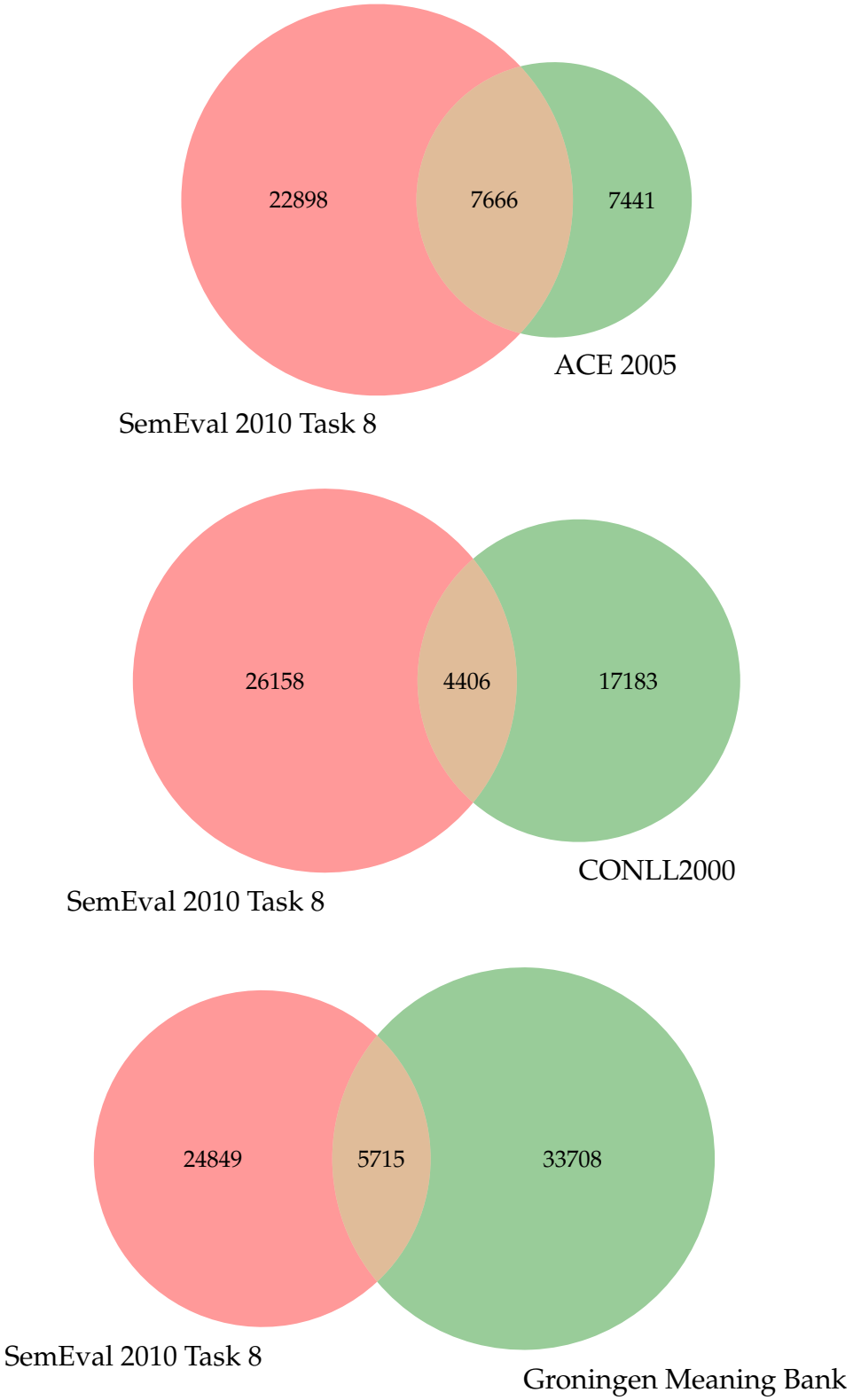


Figure 7.1
Vocabulary overlap between the corpora used in our experiments. The Venn diagrams show the number of tokens occurring in both the SemEval 2010 Task 8 vocabulary and each of the vocabularies of the auxiliary tasks.

7.2 Why is ACE not a Useful Auxiliary task?

The reasoning in the previous section does not explain why there is no observable improvement when sharing both the word embeddings and the convolutional filters between between the hypotheses for SemEval 2010 Task 8 and the ACE 2005 relation classification task.

7.3 Are the Sequence Classification Tasks Useful?

In the absence of a full fledged theory of how tasks should be related and neural network weights be shared in order for multi-task learning to improve generalization, the most reliable way to investigate the theses dynamics for a specific application is trial and error. In other words, there is no real way of knowing whether our informal reasoning on why the auxiliary tasks should benefit a target relation classification task may be totally wrong.

Something about tests...

Part 8

Perspectives

In this section we reflect on the points highlighted in the previous section. We focus on suggesting multi-task learning strategies for relation classification that we believe still needs to be explored in order to determine if multi-task learning can be an effective tool for relation classification tasks. We begin by exploring alternative neural network architectures that remove the weight sharing limitations of the architecture proposed by Nguyen and Grishman (2015). We then turn to suggestions for other possible auxiliary tasks. We end the section with a discussion of the the pros and cons of multi-task learning vs. feature engineering.

8.1 Alternative Neural Network Architectures

As discussed, the architecture suggested by Nguyen and Grishman (2015) puts limitations on how neural network weights can be shared in practice. In this section, provide basic outlines for how to construct a neural network architecture that removes these limitations.

8.1.1 Convolutional Neural Network with Argument Markers

It is generally accepted that a successful relation classification system needs as input some representation of which words of the input sentence constitute the relation arguments (Nguyen and Grishman, 2015; Zhang and Wang, 2015; Jiang, 2009). The solution proposed by Nguyen and Grishman (2015) is to augment the sentence matrix formed from the concatenated word vectors pertaining to the input with position vectors that encode distances to the relation arguments.

This has the unfortunate effect of hindering weight sharing of the convolutional filter weights between the relation classification network and sequence classification networks in practice, since it induces a mismatch of the dimensionality of the respective convolutional filter matrices.

Other researchers have found that it's possible to remove the position features completely if each sentence is pruned such that the first and last word constitute the first and last relation argument words (Santos et al., 2015). This suggests that the specific representation of relation argument positions is unimportant.

One strategy for equalizing the sentence matrix dimensionality is therefore to adapt a relation argument position representation that does away with the position embedding. The pruning strategy of (Santos et al., 2015) is one path forward. We speculate that simply marking the relation arguments with special beginning and end tokens would also work. By equalizing the dimensionality of the sentence matrix for the relation classification network and the sequence prediction networks sharing the convolutional filter weights is made possible, which may lead to better results than we have reported here.

8.1.2 Multi-Channel Convolutional Neural Network

Kim (2014) introduces the idea of using multiple "channels" in convolutional neural networks for sentence classification. The idea of a channel in this context is similar to a color channel in a digital image. An image is often represented as a stack of three matrices where each matrix component denotes color intensity in a color channel.

Similarly, we can construct a stack of sentence matrices by using more than one word embedding matrix. A number of sentence matrices are produced by concatenating the word vectors of each word in each word embedding matrix with each other such that each word embedding induces a sentence matrix.

We can use the idea of multi-channel convolutional neural networks to modify the architecture proposed by Nguyen and Grishman (2015) to accommodate sharing convolutional filter weights between the relation classification network and the sequence classification networks. Specifically, we can add another word embedding matrix to the architecture that forms a sentence matrix for an input sentence which is not augmented with the position vectors. This word embedding and convolutional filters on the sentence matrix matrix induced by it can then be readily shared with the sequence classification networks.

8.1.3 Recurrent Neural Network

Zhang and Wang (2015) describes a recurrent neural network architecture for relation classification that can also be used for a multi-task learning. In this architecture, the word vector for each word is fed into a bi-directional recurrent network layer. This produces a sequence of feature vectors that are the concatenation of the output of each application of the recurrent layer in both directions. Zhang and Wang (2015) apply max-pooling on the components of these feature vectors, and feed the resulting vector into a logistic regression layer. The relation arguments are marked simply by adding special tokens before and after each argument.

It's possible to share the weights of the recurrent layer with a sequence classification model by adding an output layer to each feature vector produced by the bidirectional recurrent layer as is done in for example Bingel and Søgaard (2017). Since the dimensionality of the weight matrices depend only on the dimensionality of the word embedding matrix, this does away the limitation induced by the position features in the architecture proposed by Nguyen and Grishman (2015).

8.2 Alternative Auxiliary Tasks

In this section we investigate other auxiliary natural language processing tasks that have the potential for improving relation classification generalization which were not tested in our experiment. We discuss the goal of each task and investigate neural network architectures that permit hard parameter sharing with a relation classification network.

8.2.1 Semantic Role Labeling

The goal of relation classification is ultimately to produce a compact representation of the semantic roles of words in text when those roles relate to specific relations of interest. **Semantic role labeling** can be seen as a generalization of this problem. The goal in semantic role labeling is to label the arguments for so called predicates in a sentence. The term *predicate* stems from logic and roughly means a function that performs a logical test on its arguments, i.e maps it to a truth value.

In natural language processing, the term predicate is often used to refer to words, often nouns, that expect certain arguments so to speak. A particular example of this is the idea of transitive and intransitive verbs. A transitive verb has a direct and possibly an indirect object, for example *brought* in *he brought her a glass of water* where *her* is the indirect object and *a glass of water* is the direct object. Intransitive verbs takes no objects, such as *laughs* in *she laughs*. We can express the transitivity of these verbs by representing them as predicates that take a fixed number of arguments, for example *brought(her, a glass of water* and *laughs(she)*.

The goal of semantic role labeling is to predict the predicate arguments given a sentence and a predicate in that sentence (Jurafsky and Martin, 2009). PropBank and FrameNet are two datasets frequently used for this task (Kingsbury and Palmer, 2002; Baker et al., 1998). See figure 8.1 for an example.

[The San Fransisco Examiner]	issued	[a special edition]	yesterday
ARG1	PREDICATE	ARG2	

Figure 8.1
Example predicate labeling derived from PropBank.

Semantic role labeling tasks can be solved with neural networks. Collobert et al. (2011) describe a convolutional architecture very similar to the architecture of Nguyen and Grishman (2015). Specifically, they approach semantic role labeling as a sequence labeling task where the goal is to assign BIO labels indicating whether a token is a the beginning, inside or outside of a predicate argument. To indicate which word is the predicate, they augment the window matrix formed by the word vectors of the window with position features that encode the distance to the predicate. It's possible to share the convolutional filters of this architecture by using the multi-channel or argument marker strategy described in section 8.1.2 and 8.1.1.

As discussed, semantic role labeling and relation classification are highly related tasks.

8.2.2 Dependency Parsing

8.3 Pipelining Vs. Multi-Task Learning

In this thesis we have investigated the possibility of re-using labeled data by using it to automatically learn a representation that improves generalization for a target task when data for this task is limited. In truth, labeled natural language data for auxiliary tasks can be re-used for this purpose in a different manner: by training a system that predicts labels for target task data. These predictions can be used as input features to the system for the target task. For example, we could use the CONLL2000 data to train the chunk and part-of-speech tags for the SemEval data, and then use these tags as input to the relation classification network.

The approach described above is in fact the standard mode of operation for natural language practitioners. Using all of the linguistic knowledge available to us to manually produce a good representation may let us use a smaller hypothesis space for the target task without penalizing the training error. Vapnik-Chervonkis analysis gives us confidence that learning with a smaller hypothesis space reduces the need for training data. Therefore, using the auxiliary data not as output information used to automatically find a good representation, but as input information, by manually creating a **pipeline** of natural language processing systems that produces a good representation for the target task, we may be able to use the auxiliary data to learn with a small hypothesis space and thereby reduce the need for labeled training data for the target task.

The main issue with such a pipeline method is **error propagation**, where classification errors early in the pipeline lead to classification errors on the target task (Collobert et al., 2011). Whether multi-task learning is preferable to pipelining in the context of relation classification is a question that, with our current understanding, may only be investigated empirically. Such a comparative study would be a significant undertaking, and we therefore suggest it as a possible future experiment that may improve our understanding of, not only when pipelining is preferable to multi-task learning, but also what makes an auxiliary task beneficial.

Part 9

Conclusion

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLbook.com, 2012.

Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The berkeley framenet project. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*, ACL '98, pages 86–90, Stroudsburg, PA, USA, 1998. Association for Computational Linguistics. doi: 10.3115/980845.980860. URL `http://dx.doi.org/10.3115/980845.980860`.

Valerio Basile, Johan Bos, Kilian Evang, and Noortje Venhuizen. Developing a large semantically annotated corpus. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC 2012)*, pages 3196–3200, Istanbul, Turkey, 2012.

Jonathan Baxter. Learning internal representations. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 311–320. ACM, 1995.

Jonathan Baxter. A model of inductive bias learning. *J. Artif. Intell. Res.(JAIR)*, 12 (149-198):3, 2000.

Shai Ben-David, Reba Schuller, et al. Exploiting task relatedness for multiple task learning. *Lecture notes in computer science*, pages 567–580, 2003.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

Joachim Bingel and Anders Søgaard. Identifying beneficial task relations for multi-task learning in deep neural networks. *arXiv preprint arXiv:1702.08303*, 2017.

Eric Brill. A simple rule-based part of speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, ANLC '92, pages 152–155, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics. doi: 10.3115/974499.974526. URL `http://dx.doi.org/10.3115/974499.974526`.

Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

Marcus et al. Treebank-3 ldc99t42., 1999.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, first edition, 2016.

Siegfried Handschuh, Vivian S Silva, Manuela Hürliman, André Freitas, and Brian Davis. Semantic relation classification: task formalisation and refinement. *CogAlex-V@ COLING 2016*, 2016.

Iris Hendrickx, Su Nam Kim, Zornitsa Kozareva, Preslav Nakov, Diarmuid Ó Séaghdha, Sebastian Padó, Marco Pennacchiotti, Lorenza Romano, and Stan Szpakowicz. Semeval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals. In *Proceedings of the Workshop on Semantic Evaluations: Recent Achievements and Future Directions*, pages 94–99. Association for Computational Linguistics, 2009.

Jing Jiang. Multi-task transfer learning for weakly-supervised relation extraction. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2*, pages 1012–1020. Association for Computational Linguistics, 2009.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Pearson Education, international edition, 2009.

Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

Paul Kingsbury and Martha Palmer. From treebank to propbank. In *LREC*, pages 1989–1993, 2002.

Michael Mabe and Mark Ware. The stm report: An overview of scientific and scholarly journals publishing. 2009.

Thien Huu Nguyen and Ralph Grishman. Relation extraction: Perspective from convolutional neural networks. 2015.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

Cicero Nogueira dos Santos, Bing Xiang, and Bowen Zhou. Classifying relations by ranking with convolutional neural networks. *arXiv preprint arXiv:1504.06580*, 2015.

Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

Erik F. Tjong Kim Sang and Sabine Buchholz. Introduction to the conll-2000 shared task: Chunking. In *Proceedings of the 2Nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning - Volume 7*, ConLL '00, pages 127–132, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics. doi: 10.3115/1117601.1117631. URL http://dx.doi.org/10.3115/1117601.1117631.

et al. Walker, Christopher. Ace 2005 multilingual training corpus ldc2006t06. Philadelphia: Linguistic Data Consortium, 2006.

Mark Ware and Michael Mabe. The stm report: An overview of scientific and scholarly journal publishing. 2015.

Dongxu Zhang and Dong Wang. Relation classification via recurrent neural network. *arXiv preprint arXiv:1508.01006*, 2015.