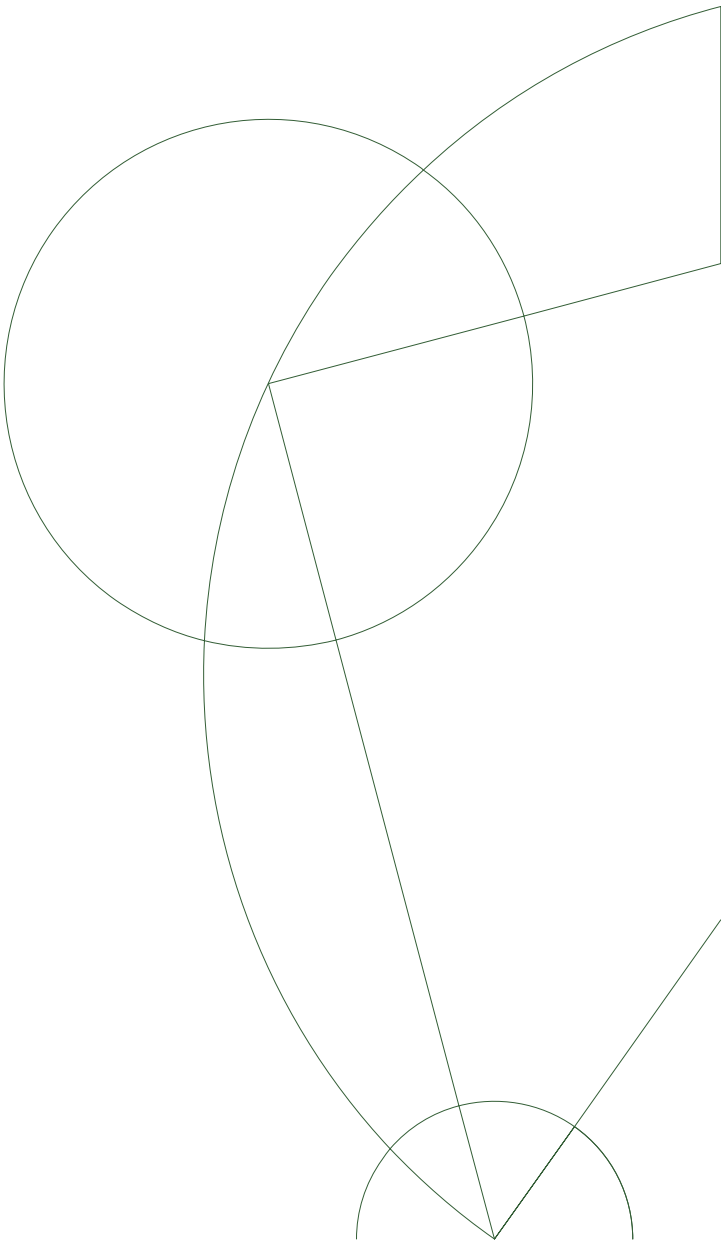
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master Thesis

Sune Andreas Dybro Debel

Deep Multi-Task Learning

For Relation Extraction



Dirk Hovy

July 20, 2017

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

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

2 Background **3**
2.1 Information Extraction . . . 3
2.1.1 Named Entity Recognition . . . 3
2.1.2 Relation Extraction . . . 4
2.1.3 Accuracy Measures . . . 4
2.2 Supervised Machine Learning . . . 5
2.2.1 The Supervised Learning Problem . . . 6
2.2.2 Statistical Learning Theory . . . 7
2.2.3 Validation . . . 9
2.3 Summary . . . 10

3 Neural Networks **11**
3.1 Feed-Forward Neural Networks . . . 11
3.1.1 Activation Functions . . . 12
3.1.2 Objective Function . . . 15
3.2 Learning Algorithm . . . 16
3.2.1 Gradient Descent . . . 16
3.2.2 Adam . . . 17
3.2.3 Backpropagation . . . 19
3.2.4 Regularization . . . 21
3.3 Convolutional Neural Networks . . . 23
3.4 Word Vectors . . . 25

4 Multi-Task Learning **27**
4.1 Multi-Task and Single-Task Learning . . . 27
4.2 Bias Learning . . . 27
4.3 Representation Learning . . . 29
4.4 Task Relatedness . . . 30
4.5 Deep Multi-Task Learning . . . 31

5 Experiment **33**
5.1 Target Task . . . 33
5.2 Auxiliary Tasks . . . 34
5.2.1 ACE 2005 Relations . . . 35
5.2.2 CONLL2000 Part-of-Speech . . . 35
5.2.3 CONLL2000 Chunking . . . 36
5.2.4 GMB Named Entity Recognition . . . 37
5.3 Neural Network Architecture . . . 37
5.4 Algorithm . . . 39

6 Results **43**
6.1 Learning Surfaces 43
6.2 Hypothesis Testing 43

7 Discussion **44**

8 Perspectives **45**

9 Conclusion **46**

Part 1

Introduction

1.1 Motivation

Information available as digital text 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 online newspapers, social media content and digital documents generated by businesses.

Finding the right information at the right time in the face of this data volume is challenging. One of the main reasons is that information contained in digital text is natural language data, which is often cited as being **unstructed** (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 relational databases for example, where every data item is associated with metadata such as column names and column types, which makes the structured data easy to search and analyze with computers. Unstructured data such as text, images, sound and video in contrast is not.

This problem has driven the development of so called **information extraction** techniques. The goal of these is essentially 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 and building data structures that capture the semantics of natural language, tasks which we currently do not fully understand how humans are able to perform.

Supervised machine learning in general, and **deep learning** in particular, is a very successful technique for solving information extraction problems. This approach is 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, in the hope that the system can learn 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. One of the main ingredients in this guarantee is the number of training examples provided for the learning system. Producing these examples can be quite expensive however. It often requires a human annotator with specialized skills to provide the correct labels, for example a trained linguist or a doctor. 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. For a new target supervised machine learning problem, it may be the case that already annotated datasets for related auxiliary tasks exists, in which case learning from all the available data may reduce the cost of creating a new annotated dataset.

1.2 Problem Statement

In this thesis we investigate multi-task learning for information extraction. We focus on an information extraction task called relation classification using deep multi-task learning techniques. Specifically:

1. We will 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 will apply our theoretical understanding by analysing a deep multi-task learning system for relation classification, in order to make predictions about its performance.
3. We implement a deep multi-task learning relation classification system based on our analysis.
4. We empirically test our predictions about the system's performance using appropriate accuracy measures.

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.

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 why they are difficult.

2.1.1 Named Entity Recognition

A named entity is roughly anything that has a proper name. The task in named entity recognition is to label mentions of entities such as people, organisations or places occurring in natural language. 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 *organisation* should ideally assign the labels:

$[\text{Jim}]_{person}$ bought 300 shares of $[\text{Acme Corp.}]_{organisation}$ 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 and *JFK* the airport near New York.

Named entity recognition can be framed as a sequence labelling problem. A common approach is to apply so called tokenisation 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.

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 the BIO labels for named entity recognition.

2.1.2 Relation Extraction

Relation extraction refers to the problem of identifying relationships such as *Family* or *Employment* between 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 recognises the relation *MergerBetween(organisation, organisation)* between two mentions of organisations. 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, its 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 have 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 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 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 true named entities or relations of all named entities or relations that were 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 true named entities or relations that were extracted by the system. This is 0 when none of the extracted information was correct, and 1 when all of the extracted information was correct, and no true named entities or relations were incorrectly labeled.

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 $\frac{2pr}{p+r}$.

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.

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 recognise 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 set $\mathcal{D}$ of N training 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 a space $\mathcal{Y}$ of labels, for example the set of all sequences of BIO tags. As designers of the learning system, we specify the so called **hypothesis space** $\mathcal{H}$, a set of functions $h : \mathcal{X} \mapsto \mathcal{Y}$. 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 not explicitly interested in the performance of h on $\mathcal{D}$ (Abu-Mostafa et al., 2012).

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

Definition 2.2.1 (generalisation 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 $e(\mathbf{y}_1, \mathbf{y_2}) = \mathbb{I}[\mathbf{y}_1 \neq \mathbf{y}_2]$ be the binary error function that measures agreement between labels $\mathbf{y}_1$ and $\mathbf{y}_2$. Then the generalisation error E of a function $h : \mathcal{X} \mapsto \mathcal{Y}$ is defined as:

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

Now, formally, the objective of supervised machine learning is to find a function h^* in a space of functions $\mathcal{H}$ that minimises $E(h)$. We see the process generating the data as random, but with a behaviour 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 is independent of h is a technique called **validation**. In 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 minimises the **training error** $\hat{E}(h, \mathcal{D})$ in the hope that g will also minimise 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 generalisation error is large is known as **overfitting**. As the name implies, it's caused by harmful idiosyncrasies of $\mathcal{D}$ that, when used to minimise $\hat{E}(h, \mathcal{D})$, 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 inherently unpredictable. 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 generalisation 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 minimise $\hat{E}$, we select a g that predicts this sample, such that $h(\mathbf{x}_i) = \mathbf{y}_i$. This is a case of overfitting since we know that there is at least one function in $\mathcal{H}$ with lower generalisation error than g, namely h^*.

The risk of overfitting is linked to the diversity of $\mathcal{H}$. By diversity of $\mathcal{H}$, we roughly mean how different any function in $\mathcal{H}$ is from any other function in $\mathcal{H}$. 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 example, if $\mathcal{Y} = \{0, 1\}$, and $N = 3$, 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. In the rest of this section, let's assume that $\mathcal{Y} = \{0, 1\}$. By simple combinatorics the number of dichotomies for N must be smaller than or equal to 2^N. 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

(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.

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 it's able to realise 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 realisable 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 realises 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 generalisation error will be close to training error, where simple means: functions that realise a small number of dichotomies.

On the other hand, having a set $\mathcal{H}$ that can realise 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 generalisation 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 minimising $\hat{E}$ as much as is actually possible. These techniques are known as **regularisation**, which we will see an instance of in section 3.2.4.

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 minimise $\hat{E}(\mathcal{D}, h)$, 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 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^-:

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

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 generalisation error. 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 generalisation 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

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

That is, the expected generalisation 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}.

In words, the cross validation error estimates the expected generalisation 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 train K different models.

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, organisations 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 the alternative precision and recall.

We have described the formal setting of on 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 generalisation error.

Part 3

Neural Networks

In this part we describe how to define $\mathcal{H}$ using functions called **neural networks**. These functions 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 specialised neural networks that are useful for natural language processing.

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 $\mathbf{x}$, except for an added **bias** component of 1, as described later in this section. each $f^{(l)}$ outputs a vector $\mathbf{x}^{(l)}$ of dimension $d^{(l)}$ which is the input of $f^{(l+1)}$. 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. By ordered, we mean that the index of the innermost functions are smaller than the index of the outermost. 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 behaviour 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 $l-1$.

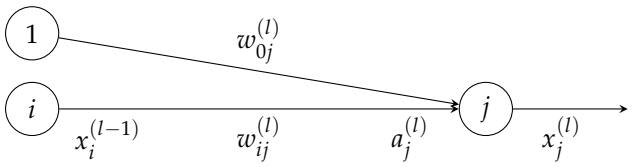


Figure 3.1
A visual representation of the connections between unit i in layer $l-1$, the bias unit in $l-1$, and unit j in layer l. The connection strength between these units is given by the weight $w_{ij}^{(l)}$ between i and j, and $w_{0j}^{(l)}$ between the bias unit and j. The activation $a_j^{(l)}$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

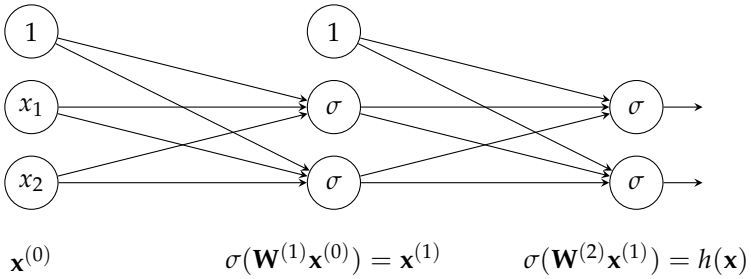


Figure 3.2
A visual representation of $h(\mathbf{x}) = f_2(f_1(\mathbf{x}^{(0)}))$. The activation at each layer $\mathbf{a}^{(l)}$ is computed by $\mathbf{W}^{(l)}\mathbf{x}^{(l-1)}$. The output at each layer is computed by element-wise application of the activation function of $\sigma(\mathbf{a}^{(l)})$.

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 cer-

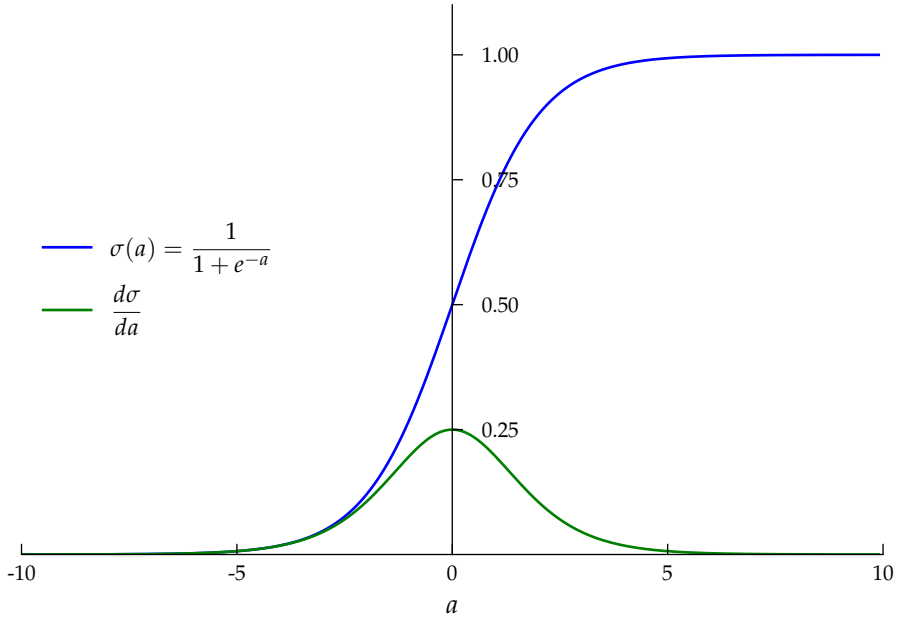


Figure 3.3
*Sigmoid activation and its derivate. Sigmoid activation units have the disadvantage of **saturating**, meaning that they become flat when a is large or small. This makes the derivative smaller than 1 everywhere, and much smaller than 1 almost everywhere.*

tain 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 it to approximate a much 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**, depicted in figure 3.4. These units use the activation function $\sigma(a) = \max(0, a)$. 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 we may arbitrarily choose $\sigma(0)$ to be either 0 or 1, and still successfully train our networks.

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 so called **cost functions** with a principled technique **maximum likelihood**. For this reason it's common to use different activation functions in the output layer.

For example, named entity recognition can be seen as a multi-class classification

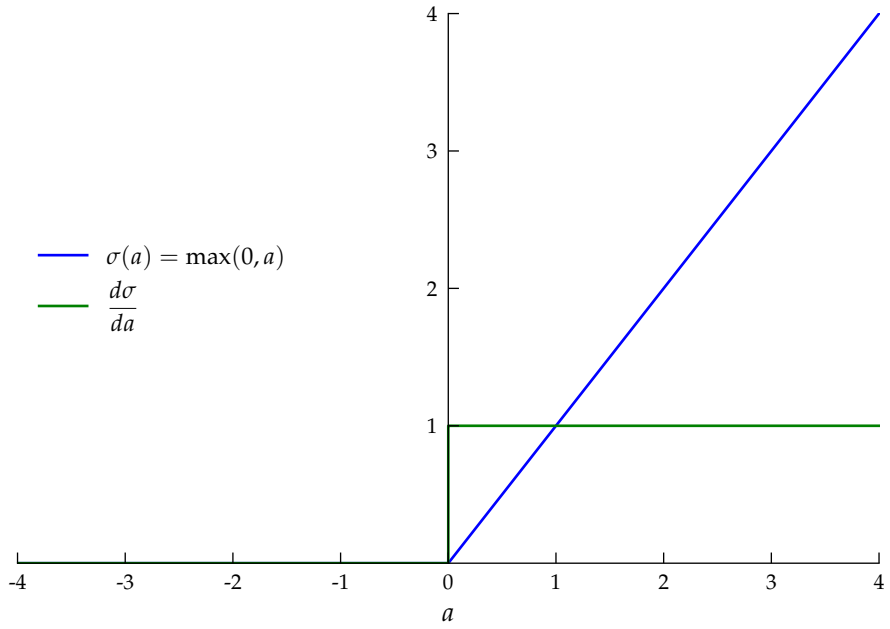


Figure 3.4
ReLU activation and its derivate. Unlike sigmoid activation, ReLU activation doesn't saturate. This means that the derivative of a unit remains large whenever that unit produces output.

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 $\mathbf{y}_i \in \mathcal{D}_{train}$ is equal to 1 if $\mathbf{x}_i \in \mathcal{D}_{train}$ 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 $h(\mathbf{x})_c = P(y = c|\mathbf{x})$.

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

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

In words, the soft-max function makes sure that the output of h is a valid probability distribution, firstly 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 $\hat{E}(h, \mathcal{D}_{train})$, 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 $\mathcal{D}_{train}$. 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 $\mathcal{D}_{train}$. Such a function is often called an objective function, borrowing terminology from the mathematical field of optimisation.

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:

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

Where y is the true label for $\mathbf{x}_i$ and 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 $P(y \mid \mathbf{x}) = (0.1^0)(0.8^1)(0.1^0) = 0.8$.

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 $\mathcal{D}_{train}$ by interpreting h as a probability distribution parameterised by $\mathbf{w}$:

Definition 3.1.1 (likelihood function)**.** Let $\mathcal{D}_{train} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$ 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 y_{ic} denote component c of the one-hot label for example i. Then the likelihood $P(\mathcal{D}_{train} \mid \mathbf{w})$ is:

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

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

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** $\ln P(\mathcal{D}_{train} \mid \mathbf{w})$ 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 minimisation 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**. Minimising the average negative log-likelihood is clearly identical to maximising 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 $\mathcal{D}_{train}$ is an optimisation problem. Optimisation 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, finding the h which maximises the likelihood by analytical differentiation is impossible. Neural network optimisation is therefore solved using an iterative algorithm called **gradient descent**, which we describe in this section. We then explore an algorithm for computing the gradient of $\hat{E}$ called **backpropagation**. Finally, we look into **regularisation** 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 $\mathcal{D}_{train}$. 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 of gradient descent which addresses this problem. In stochastic gradient descent, a single training example $(\mathbf{x}_i, \mathbf{y}_i)$ is sampled from $\mathcal{D}_{train}$. Instead 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 $\mathcal{D}_{train}$ 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 minimises $\hat{E}$. To counter this problem, 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 vector 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.

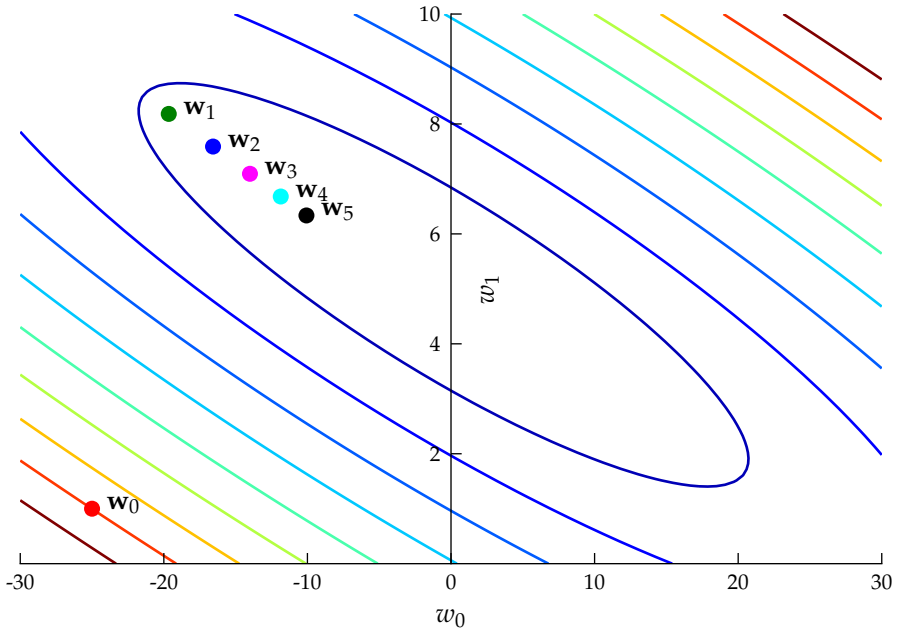


Figure 3.5
Level curves of squared training error $\hat{E}(h, \mathcal{D}_{train}) = \frac{1}{N}\sum_{i=1}^{N}(h(\mathbf{x}_i) - y_i)^2$ *for a toy* $\mathcal{D}_{train}$ *shown in 3.6, and the simple* $\mathcal{H} = \{h = \mathbf{w}^T\mathbf{x}^{(0)} \mid \mathbf{w} \in \mathbb{R}^2\}$. $\hat{E}$ *has 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.*

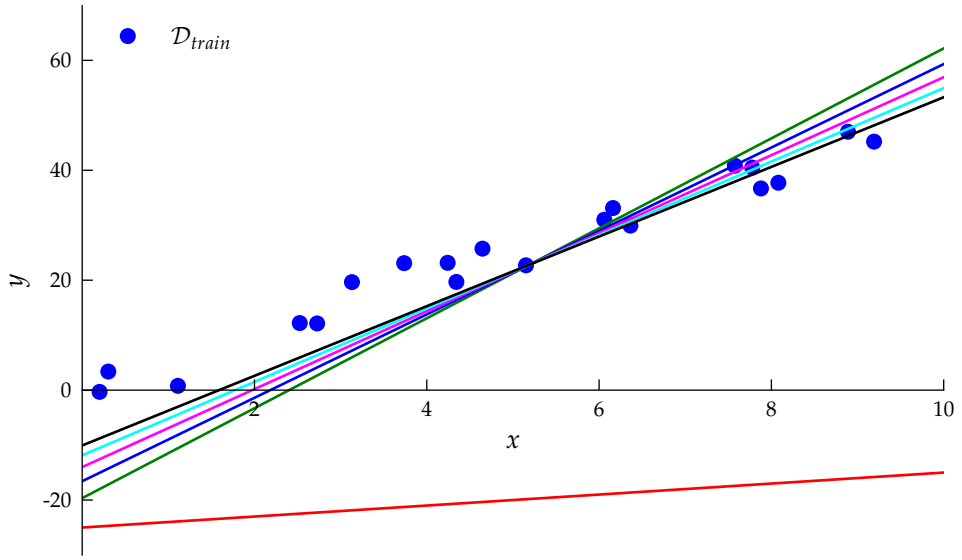


Figure 3.6
The training data $\mathcal{D}_{train}$ *used in figure 3.5. The colored lines correspond to* $h(\mathbf{x}, \mathbf{w}_i) = 0$ *for each weight vector* $\mathbf{w}_i$ *found by gradient descent in figure 3.5, such that for example* $h(\mathbf{x}, \mathbf{w}_0) = 0$ *is given by the red line.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

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 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 element $\frac{\partial e}{\partial w_{ij}^{(l)}}$ 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 summarise, 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 $\hat{E}(h, \mathcal{D}_{train})$ is bounded by a function that's roughly a measure 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 regularisation.

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 in 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 $\lambda > 0$, $\bar{\mathbf{w}}$ minimises a new error function which we will call **augmented error** $\bar{E}(\mathbf{w}, \mathcal{D}_{train})$:

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

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

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

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

In words, the added norm penalty of $\bar{E}(\mathbf{w}, \mathcal{D}_{train})$ 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

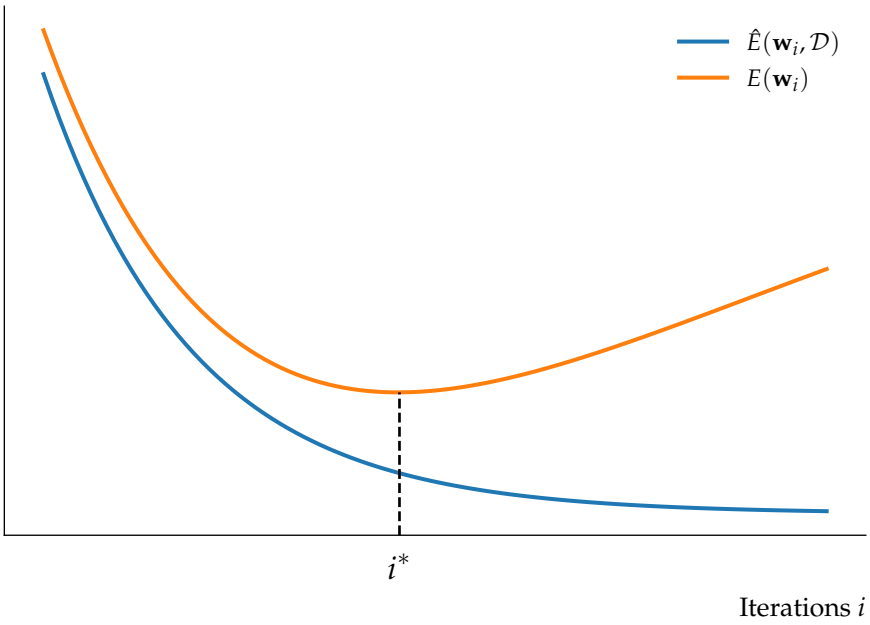


Figure 3.7
Typical behavior of E and $\hat{E}$ as a function of the number of iterations i of gradient descent. Both errors are reduced until a point i^* beyond which the training error is reduced, but generalization error increases.

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, which means that the simple heuristic described above may fail. A so called **patience** parameter is a simple solution to this problem. When using patience p, 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 is a mathematical operation that takes as input two functions f and k, commonly written as $f * k$.

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

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

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.

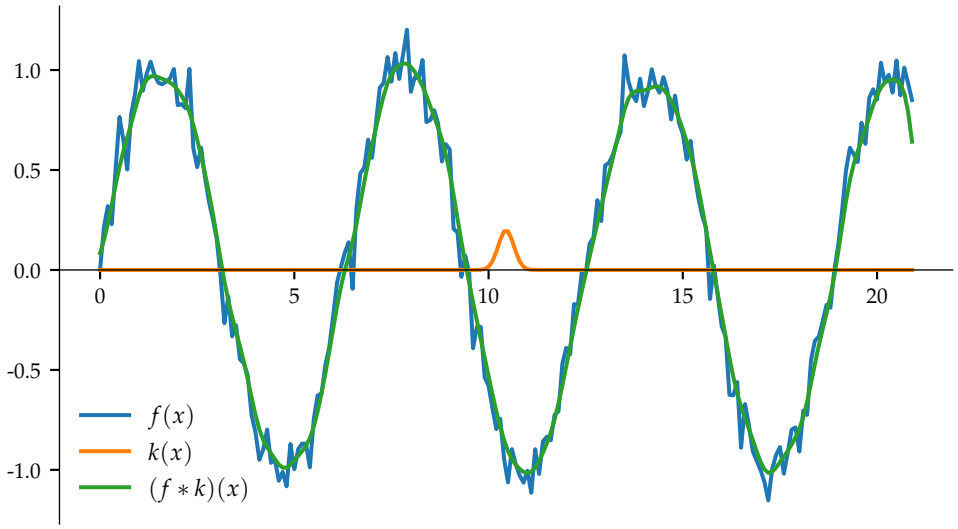


Figure 3.8
Visualisation of a noisy signal f convolved with a small Gaussian kernel k. The output of the convolution $f * k$ captures the general trend of f by averaging the outputs of f at every x, such values of f of inputs close to x contribute more to the output of the convolution, than inputs far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

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 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 used at every index x in the convolution operation. 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.

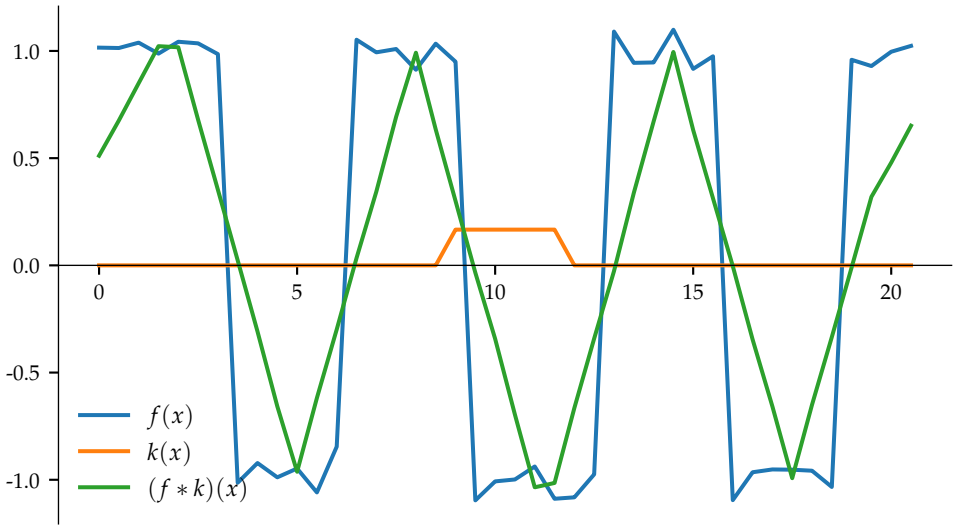


Figure 3.9
Visualisation of convolutional kernel as feature detector. When the signal f is similar to the kernel, the output of the convolution is maximally positive.

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.

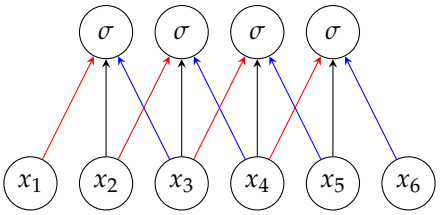


Figure 3.10
Visual representation of a one-dimensional convolution implemented as the first layer of a convolutional neural network. The connections between the input layer and the convolutional layer are sparse in that each unit is connected only to three of six inputs. The colors of the connections 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 features without any concept of distance, such as words, to a prediction, such as the presence of a named entity, may be more difficult than learning a mapping from continuous features 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 initialised randomly

Glove vectors are learned by minimising 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:

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

Minimising 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 initialised with small random components as any other neural network parameter, or they can be initialised 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 multitask 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 $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}$ belong to the same input/output space $\mathcal{X}$ and $\mathcal{Y}$, and 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 one or more of these assumptions: 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 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:

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

In many cases, we are not interested in implementing a learning system that performs well on all M learning tasks, but 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**. 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 bias 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 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**.

The objective of representation learning, according to Baxter, 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 formalise 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 reduces 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 generalisation error of f is the average single task generalisation 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 we will see in section 4.5, the potential advantages 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 the equivalence relation $\sim \mathcal{F}$ $[h]_{\sim \mathcal{F}}$ on $\mathcal{H}$, where h_1 and h_2 are equivalent if there exists $f \in \mathcal{F}$ such that $h_2 = h_1 \circ f$. 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 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:

$$\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim P_1(\mathbf{x},\mathbf{y})}[e(h(\mathbf{x}), \mathbf{y})] = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim P_2(\mathbf{x},\mathbf{y})}[e((h \circ f)(\mathbf{x}), \mathbf{y})]$$

This means that the equivalence classes of $\mathcal{H}$ perform equally well on the different tasks, when measured by $E(H) = \inf_{h \in H} E(h)$.

Building on the result from Baxter (2000), Ben-David et al. (2003) is able to show

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 $1 \leq m \leq M$:

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

4.5 Deep Multi-Task Learning

Neural networks have the advantage of being easy to adapt from single-task learning to multi-task learning by sharing the weights of early 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.

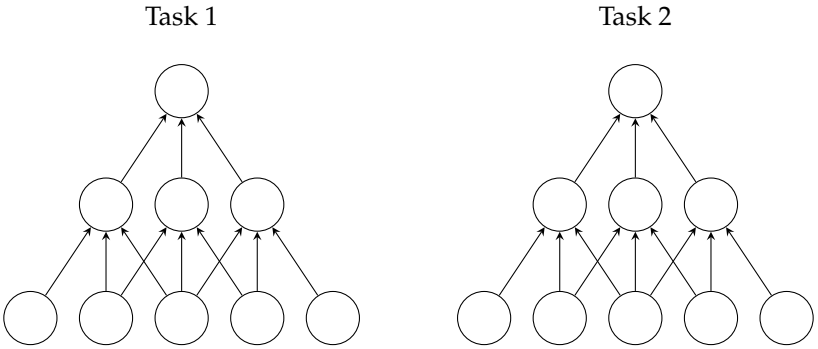


Figure 4.1
Visual representation of single-task learning with neural networks. A set of neural network weights are learnt separately for Task 1 and Task 2.

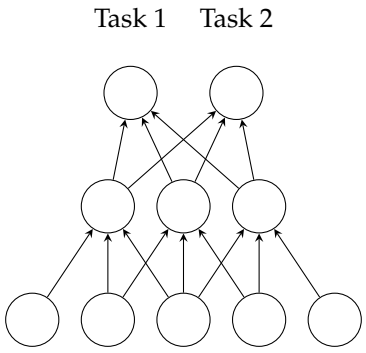


Figure 4.2
Visual representation multi-task learning with neural networks. The weights of the first layer is shared between the two tasks.

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 to each task.

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

Galanti et al. (2016) develops a framework based on the bias learning framework by Baxter (2000) that may be used to gain some intuition on when and how deep multi-task learning is beneficial.

Part 5

Experiment

In this section we explain our experimentation with multi-task learning for relation classification. We explain each of the learning tasks included in the experiment. We then describe the neural network architecture used in the experiments. Finally we describe the learning algorithm used in the experiments.

5.1 Target Task

In order to empirically investigate the dynamics of sample complexity for multi-task relation classification, we have chosen a target relation classification dataset which will act as a baseline across our experiments. 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.

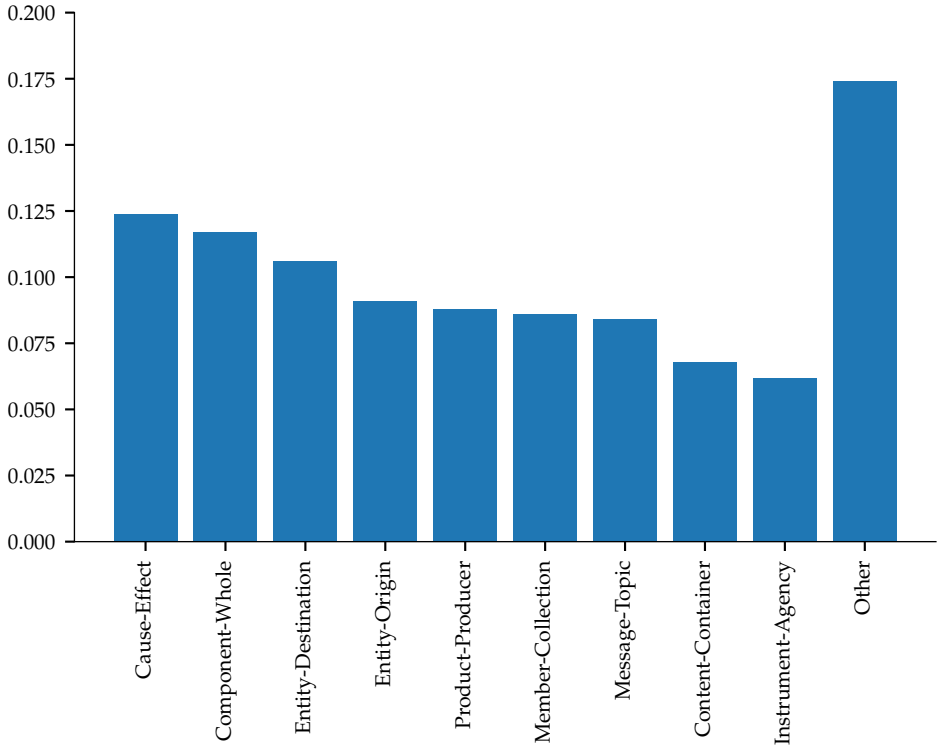


Figure 5.1
Label distribution of SemEval 2010 Task 8.

SemEval 2010 Task 8 differs somewhat from what is traditionally meant by relation classification. 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 noun-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 achieve this, they initially collected approximately 1200 sentences for each 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. This may lead to optimistic performance of trained models.

5.2 Auxiliary Tasks

Here we describe each of the datasets used as auxiliary tasks in our multi-task learning experiment.

5.2.1 ACE 2005 Relations

Intuitively, we expect that a low-level representation that's useful for one relation classification task should also be useful to another, assuming the targeted relations in one task are related to the relations in another.

We would therefore like to 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].*

5.2.2 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 tagging is known to be a useful input feature for a number of other learning tasks, here-among named entity recognition and relation extraction. This believed to be the case since word classes are highly informative of a word's semantic role in a sentence.

Several 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 labelling scheme. 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).

5.2.3 CONLL2000 Chunking

Identifying the structure of sentences 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.

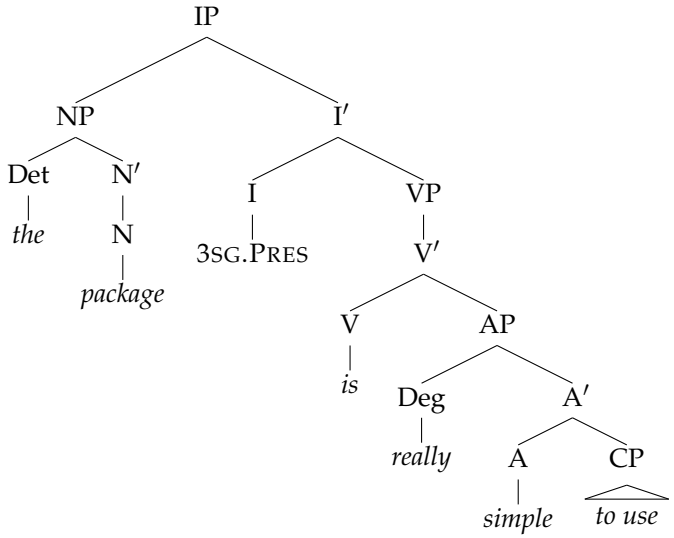


Figure 5.3
Syntactic parse tree for the sentence: I shot an elephant in my pajamas.

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

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.*

chunking is to identify identifying the flat, non-overlapping parts of a sentence that constitute the major its basic non-recursive phrases. Se for example figure 5.4.

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

5.2.4 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 othe r 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.

5.3 Neural Network Architecture

For our deep multi-task learning experiments, we use two closely related convolutional neural network architectures: one architecture for the sequence classification problems CONLL2000 part-of-speech, CONLL2000 chunking and GMB named entity recognition, and another architecture for the relation classification problems SemEval 2010 Task 8 and ACE 2005. This is necessary since the relation classification tasks require special mechanisms for representing which words in the input sentence constitute the arguments for the target relation.

For the relation classification tasks we use a state-of-the-art convolutional neural network architecture based on Nguyen and Grishman (2015). In this architecture, 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 $\mathbf{S}$. 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 head words of relation arguments, we compute the distance between each word index i and the index of the head word of the relation arguments $e1$ 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. 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 lead to some technical 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 share only the word-embedding matrix between the relation classification tasks and sequence classification tasks. In our experiment, we initialize the word-embedding matrix with GloVe vectors trained on the Common Crawl corpus (commoncrawl.org). A diagram of neural network weights are shared between tasks can be seen in figure 5.6

5.4 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 5.2.

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 5.3. 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 of the parameters 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 recorded 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. The condition on the maximum number of iterations without improvement is commonly known as the patience.

When the patience is exceeded we record the cross-validation macro-F1 on the target task test fold. 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
```

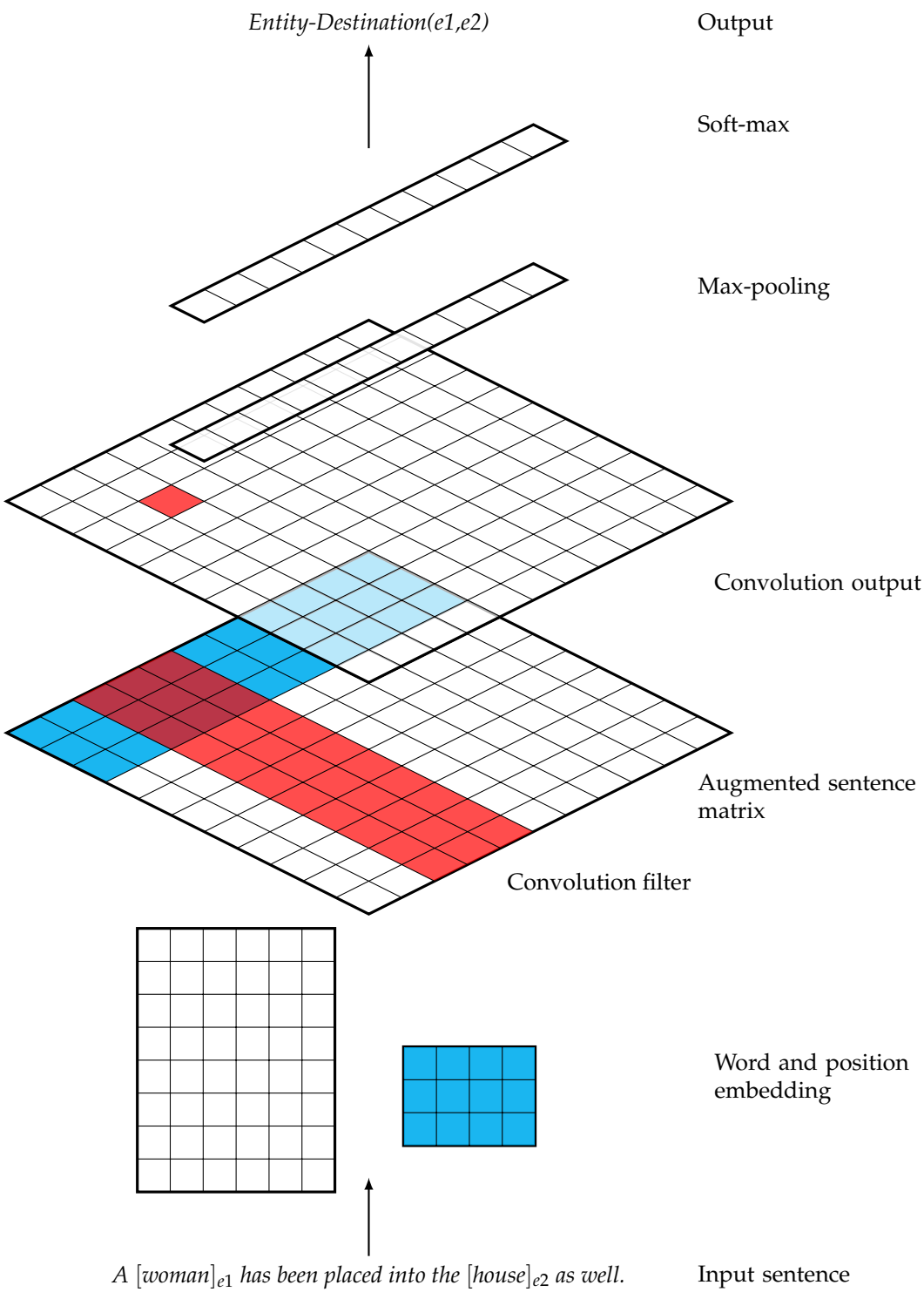


Figure 5.5
Diagram of convolutional neural network for relation classification. The input sentence is mapped to a sentence matrix by concatenating the word-vector for each word in the word embedding matrix and the position-vector for each word-position in the position embedding matrix. This forms an $s \times d + d'$ matrix. Convolution filters are applied along the s-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. 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.

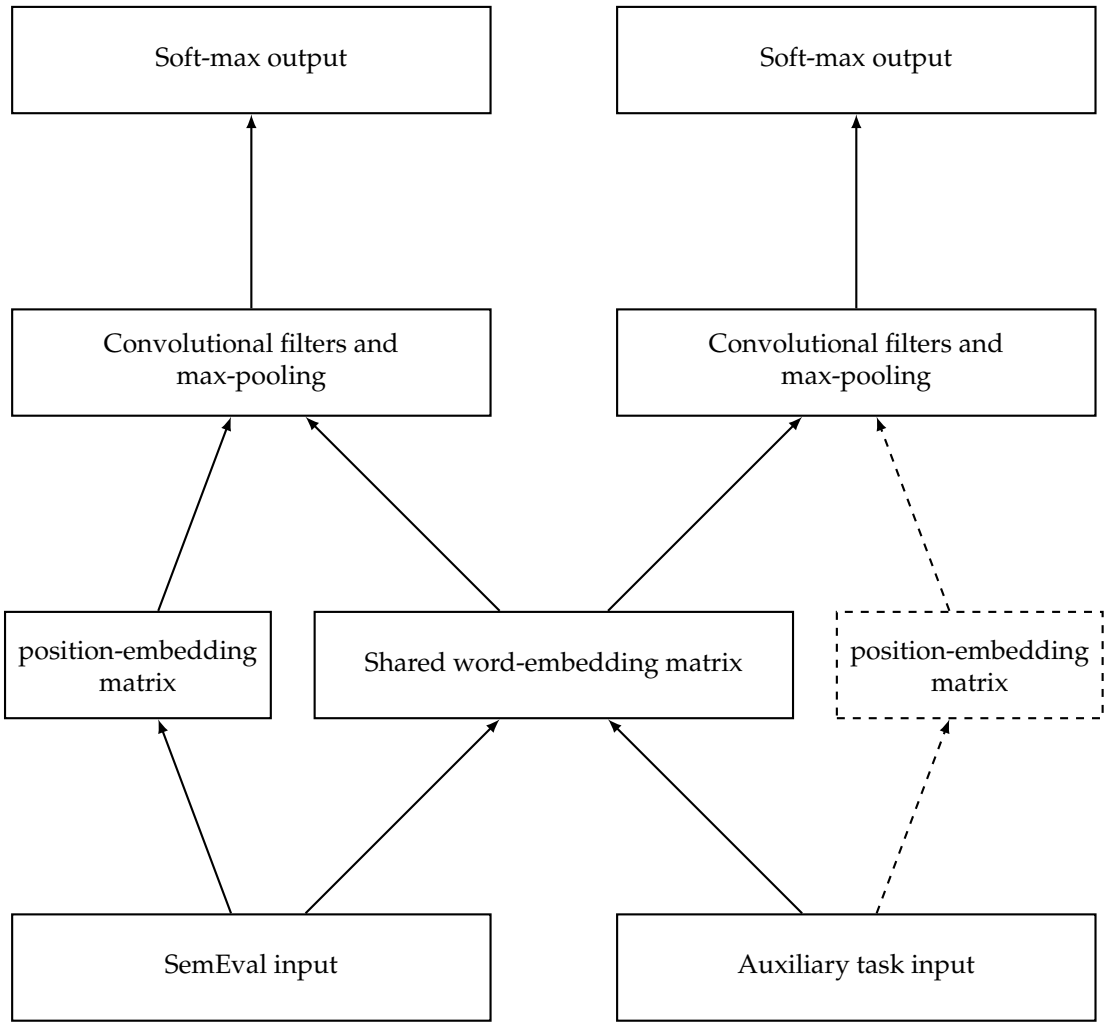


Figure 5.6
Diagram of how neural network weights are shared between auxiliary tasks and SemEval 2010 Task 8. The word-embedding matrix is shared between tasks, but the convolution filters are not.

Part 6

Results

6.1 Learning Surfaces

6.2 Hypothesis Testing

Part 7

Discussion

Part 8

Perspectives

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.

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.

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, 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.

Tomer Galanti, Lior Wolf, and Tamir Hazan. A theoretical framework for deep transfer learning. *Information and Inference: A Journal of the IMA*, 5(2):159–209, 2016.

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

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

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.

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.