FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

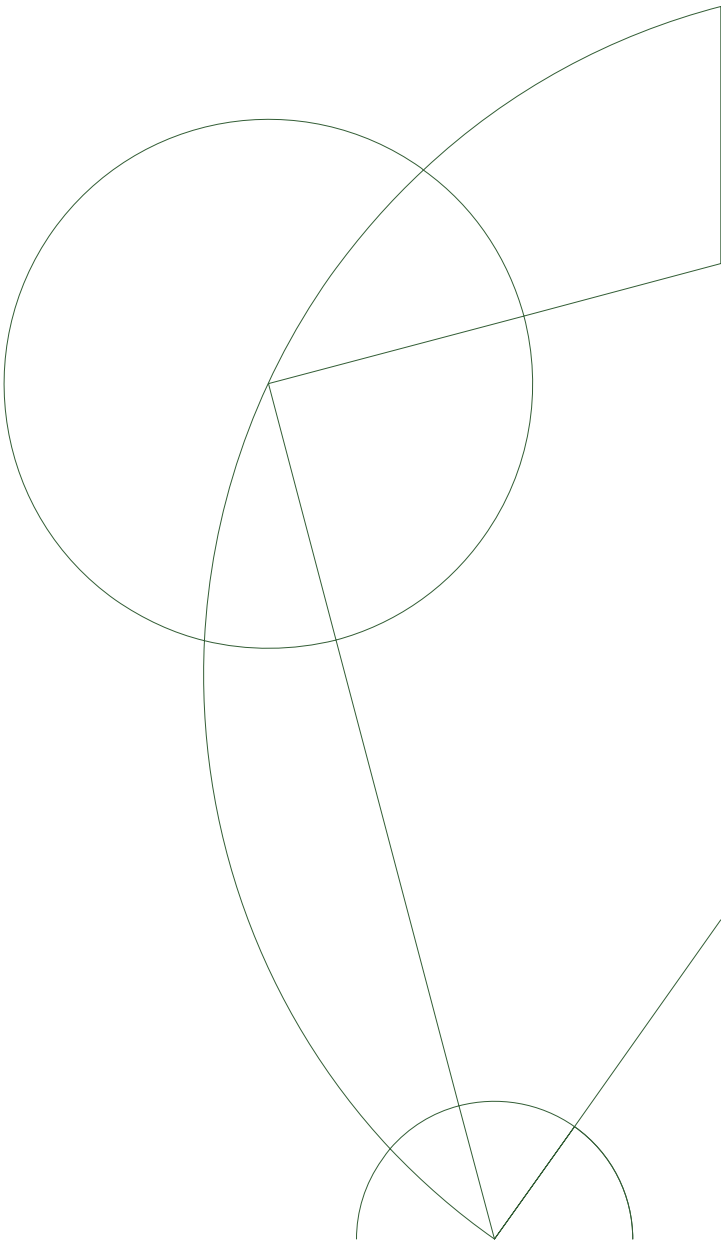
418.25368pt

Master Thesis

Sune Andreas Dybro Debel

Deep Multi-Task Learning

For Relation Extraction



Dirk Hovy

March 16, 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

Part 1

Introduction

1.1 Motivation

1.2 Problem Statement

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.

Because relation extraction is concerned with relationships between named entities, many systems that perform relation extraction applies named entity recognition first as a pre-processing step. This approach is sometimes called **pipelining**. An alternative to pipelining is **end-to-end** relation extraction, where relations and entities are extracted simultaneously. Pipeline approaches can suffer from the problem of **error propagation**, where the system erroneously assigns a label in the named entity recognition step, which later causes it to make an error in the relation extraction step.

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. Accuracy measures used in such tests are usually defined in terms of:

True positives (tp) The number of true named entities or relations correctly labeled by the system.

True negatives (tn) The number of true non-entities or non-relations correctly labeled by the system.

False positives (fp) The number of true non-entities or relations incorrectly labeled by the system.

False negatives (fn) The number of true named entities or relations incorrectly labeled by the system.

The distribution of labels used in both named entity recognition and relation extraction is often highly imbalanced. Consider for example the BIO labelling scheme in figure 2.1. Most words will be outside a mention of a named entity, and will have the label O. Using simple accuracy $\frac{tp+tn}{N}$ 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 summarises 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 $\frac{2pr}{p+r}$.

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 $\mathcal{D}_{train}$ 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 $\mathcal{D}_{train}$ 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 $\mathcal{D}_{train}$ (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 $\mathcal{D} = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{x}, \mathbf{y} \sim P(\mathbf{x}, \mathbf{y})\}$ to estimate $E(h)$ with a quantity known as **empirical error**:

Definition 2.2.2 (empirical error)**.** Let $\mathcal{D}$ 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:

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

Because $\mathcal{D}$ 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 $\mathcal{D}$ 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 $\mathcal{D}$, 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 $E(h, \mathcal{D})$ be defined as in definition 2.2.2. Then:

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

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 D, 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, the sample provided by a human annotator is split into two datasets, $\mathcal{D}_{train}$, which we intend to use to search for h^*, and $\mathcal{D}_{validate}$, which saved until we are done searching. Since $\mathcal{D}_{validate}$ is independent of whichever h we selected, Hoeffding's inequality applies and $\mathcal{D}_{validate}$ can be used to estimate E.

Because $\mathcal{D}_{train}$ 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 $\mathcal{D}_{train}$ and E. The central question in supervised machine learning is *how can we best define $\mathcal{H}$ and use $\mathcal{D}_{train}$ 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 $\mathcal{D}_{train}$ in order to make E small. $\mathcal{D}_{train}$ 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** $\hat{E}(g, \mathcal{D}_{train})$ 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_{train} 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 $\hat{E}(g, \mathcal{D}_{train})$ 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 $\mathcal{D}_{train}$ that, when used to minimise $\hat{E}(h, \mathcal{D}_{train})$, leads us to a g with a larger E than other functions in $\mathcal{H}$. These idiosyncrasies of $\mathcal{D}_{train}$ 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 $\mathcal{D}_{train}$ 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 $\mathcal{D}_{train}$.

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.

(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.

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 $\mathcal{D}_{train}$, many of them will produce the same dichotomy since the number of training examples in $\mathcal{D}_{train}$ 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 $\hat{E}(h, \mathcal{D}_{train})$:

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 $1 - \delta$:

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

The bound tells us that $E(h)$ will be close to $\hat{E}(h, \mathcal{D}_{train})$ 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 $\hat{E}(h, \mathcal{D}_{train})$ 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 3.2.3.

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 $\mathcal{D}_{train}$ 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 $h(\mathbf{x}) = f_2(f_1(\mathbf{x}))$. 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**. Each f_l outputs a vector $\mathbf{z}_l$ of dimension d_l. The size of these vectors determine the **width** of the network. We denote the input to f_1 as $\tilde{\mathbf{x}}$, which is identical to the input vector $\mathbf{x}$, except for an added **bias** component of 1, as described later in this section.

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. Networks where this restriction is removed are called **recurrent neural networks**, and will be discussed in section 3.4.

Finally h is called a **neural** network since its design is loosely based on neurons in the brain (Goodfellow et al., 2016). Each component z of the vector $\mathbf{z}_l$ 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 $z_{l-1,v}$ of unit v in layer $l-1$ is multiplied by a weight $w_{l,v,u}$ that gives the strength of the connection between unit v in $l-1$ and unit u in l. Unit u sums all of the input it receives from units in layer $l-1$ to obtain its **activation** $a_{l,u} = \sum w_{l,v,u} z_{l-1,v}$. To compute its output $z_{l,u}$, it applies an **activation function** $\sigma(a_{l,u})$ 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 u computes its output $z_{l,u}$ by combining

the outputs of units in layer $l-1$.

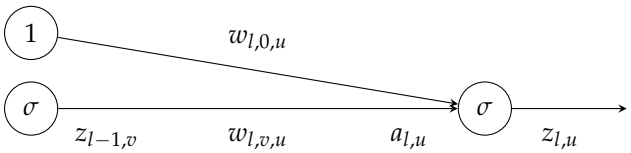


Figure 3.1
A visual representation of the connections between unit v in layer $l-1$, the bias unit in $l-1$, and unit u in layer l. The connection strength between these units is given by the weight $w_{l,v,u}$ between v and u, and $w_{l,0,u}$ between the bias unit and u. The activation $a_{l,u}$ at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

Keeping track of the indices l, u and v quickly becomes confusing. By collecting all of the weights of connections going into unit u in layer l in a vector $\mathbf{w}_{l,u}$, the activation at unit u can be computed as a dot product $a_{l,u} = {\mathbf{w}_{l,u}}^T\mathbf{z}_{l-1}$. Moreover, we can compute the entire vector $\mathbf{a}_l$ of activations at layer l, by organising the weight vectors $\mathbf{w}_{l,u}$ in a matrix $\mathbf{W}_l = [\mathbf{w}_{l,1} \dots \mathbf{w}_{l,d_l}]^T$, which leads to $\mathbf{a}_l = \mathbf{W}_l\mathbf{z}_{l-1}$.

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.

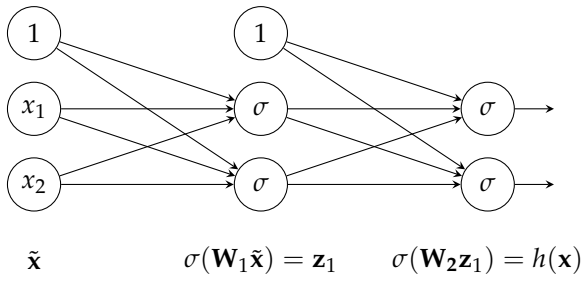


Figure 3.2
A visual representation of $h(\tilde{\mathbf{x}}) = f_2(f_1(\tilde{\mathbf{x}}))$. The activation at each layer $\mathbf{a}_l$ is computed by $\mathbf{W}_l\mathbf{z}_{l-1}$. The output at each layer is computed by element-wise application of the activation function of $\sigma(\mathbf{a}_l)$.

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 $\mathbf{w} = \left[w_{1,u,v} \dots w_{L,u,v}\right]$ and writing $h(\mathbf{x}, \mathbf{w})$ which means *the function h parameterised by the weight vector* $\mathbf{w}$. In the next section we discuss how to choose the activation functions at the layers of the network.

3.1.1 Activation Functions

Activation functions mimic the behaviour of neurons in the brain. A neuron emits a signal when the combined input it receives from other neurons exceeds a certain threshold. Activation functions achieve this by a variation of the step function, where an activation signal $a_{l,u}$ 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.

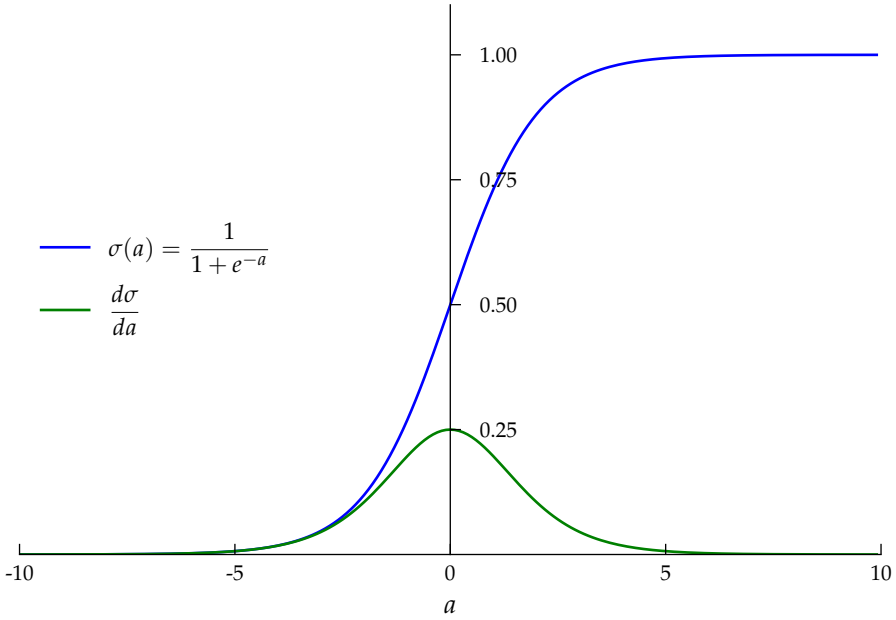


Figure 3.3
*Sigmoid activation and its derivate. Sigmoid activation units have the disadvantage of **saturating**, meaning that they become flat when a is large or small. This makes the derivative smaller than 1 everywhere, and much smaller than 1 almost everywhere.*

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 $\mathcal{D}_{train}$ 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 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 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 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.

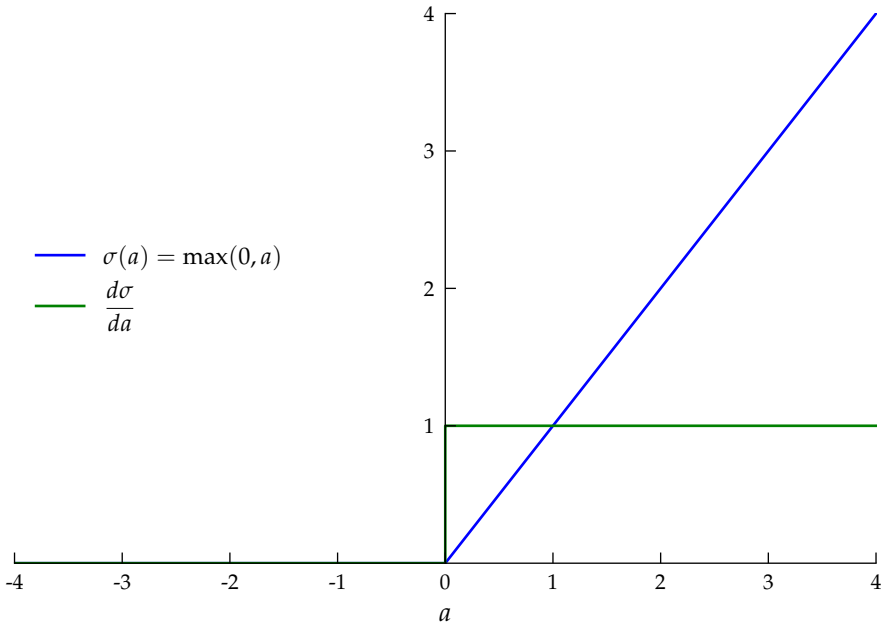


Figure 3.4
ReLU activation and its derivate. Unlike sigmoid activation, ReLU activation doesn't saturate. This means that the derivative of a unit remains large whenever that unit produces output.

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

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

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 set of weights $\mathcal{W}$ 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:

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

This formulation works because $\mathbf{y}$ is a one-hot vector, which means just component of $\mathbf{y}$ is equal to 1, and all other components are 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 $P(\mathbf{y} \mid \mathbf{x}) = (0.1^0)(0.8^1)(0.1^0) = 0.8$.

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 $\mathbf{y}_{i,c}$ denote the c'th component of the one-hot label for example i. Then the likelihood $P(\mathcal{D}_{train} \mid \mathbf{w})$ is:

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

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** $\ln P(\mathcal{D}_{train} \mid \mathcal{W})$ 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 $\hat{E}(h, mathcal D_{train}$. 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:

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

This error measure is also known as **cross-entropy error**, in which the term $-\sum_{c=1}^{C} \mathbf{y}_{i,c} \ln h(\mathbf{x}_i, \mathbf{w})_c$ is taken as the error measure $e(\mathbf{y}_i, h(\mathbf{x}_i))$, 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 be 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 $w_{l,u,v} \in \mathbf{w}$ be the component at index u, v of $\mathbf{W}_l$ in h, and let $\hat{E}$ be defined as in definition 3.1.2. Then the gradient $\nabla\hat{E}$ is:

$$\nabla\hat{E} = \begin{bmatrix} \frac{\partial}{\partial w_{1,v,u}}\hat{E} \\ \vdots \\ \frac{\partial}{\partial w_{L,v,u}}\hat{E} \end{bmatrix}$$

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:

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

Another practical issue of using gradient descent is how to choose the learning rate η. We investigate a method called Adam that uses a heuristic to select η dynamically in each iteration in section 3.2.4.

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.

3.2.2 Backpropagation

3.2.3 Regularisation

3.2.4 Adam

3.3 Convolutional Neural Networks

3.4 Recurrent and Recursive Neural Networks

3.4.1 LSTM

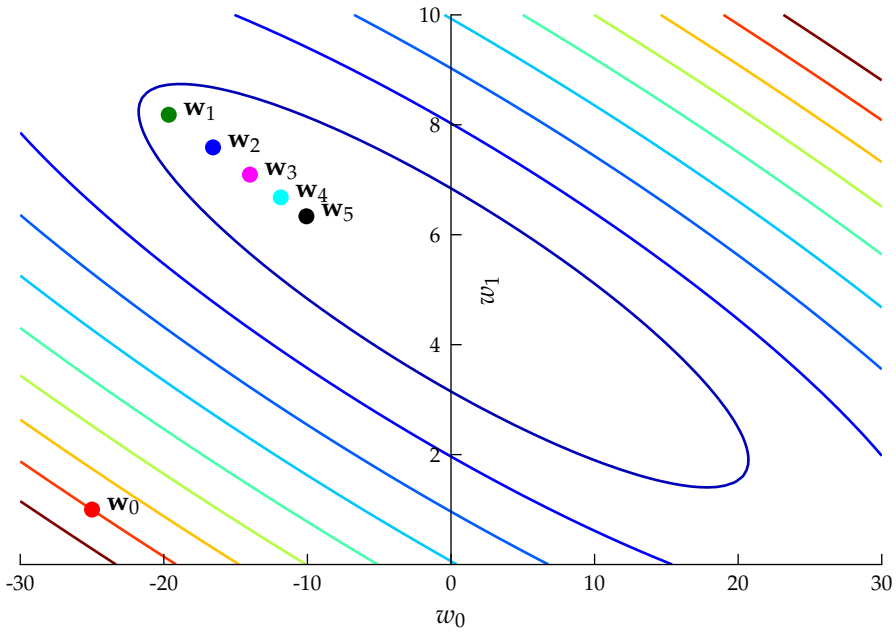


Figure 3.5
Level curves of squared training error $\hat{E}(h, \mathcal{D}_{train}) = \frac{1}{N}\sum_{i=1}^{N}(h(\tilde{\mathbf{x}}) - y_i)^2$ *for a toy* $\mathcal{D}_{train}$ *shown in 3.6, and the simple* $\mathcal{H} = \{h = \mathbf{w}^T\tilde{\mathbf{x}} \mid \mathbf{w} \in \mathbb{R}^2\}$. $\hat{E}$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

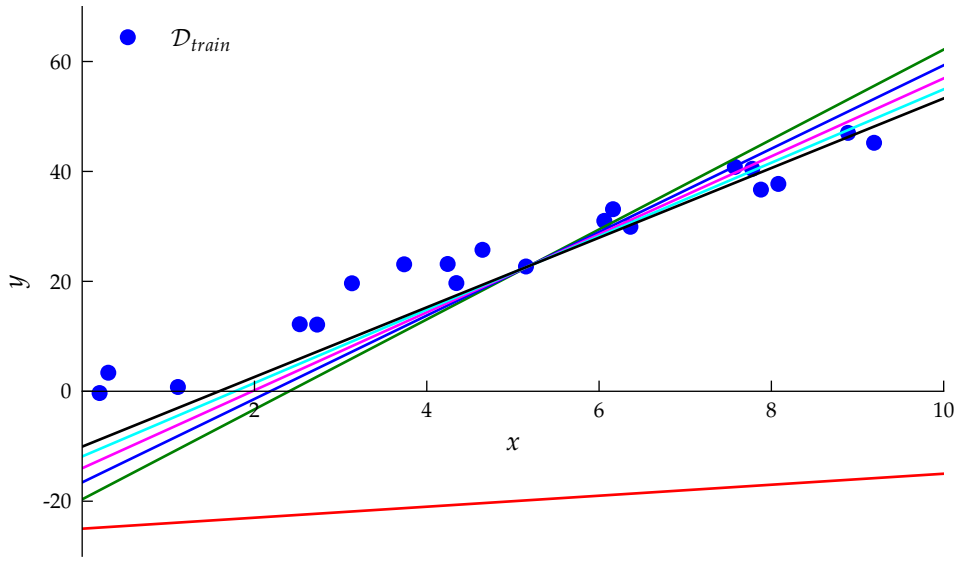


Figure 3.6
The training data $\mathcal{D}_{train}$ *used in figure 3.5. The colored lines correspond to* $h(\tilde{\mathbf{x}}, \mathbf{w}_i) = 0$ *for each weight vector* $\mathbf{w}_i$ *found by gradient descent in figure 3.5, such that for example* $h(\tilde{\mathbf{x}}, \mathbf{w}_0) = 0$ *is given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

Part 4

Multi-Task Learning

4.1 Representation Learning

4.1.1 Auto Encoders

4.1.2 Word Embeddings

4.2 Deep Multi-Task Learning

Part 5

Experiment

5.1 Auxiliary Tasks

5.2 Network Architecture

Part 6

Results

Part 7

Discussion

Part 8

Conclusion

Bibliography

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

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

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

Appendix A

Appendix

A.1 First