NOVIA
UNIVERSITY OF APPLIED SCIENCES

INTELLIGENT SYSTEMS

Johan Westö

SERIES L: TEACHING MATERIALS, 2/2014

Abstract

This course material is meant as an introduction to neural networks. Based on a mathematical representation of neurons, the material will try to present 1) how these neurons can be used to build models, 2) how the models are dependent on the network's structure, and 3) how we can make the networks learn from data. During the course, neural networks will be trained to solve simple examples problems as well as large scale real problems in the form of image classification. The goal is to give students a basic understanding for how neural networks can be used to solve both regression and classification problems.

Sammanfattning

Denna kurs är tänkt som en introduktion till neurala nätverk. Materialet kommer att påvisa hur en matematisk representation av nervceller kan användas för att representera modeller, samt hur dessa modeller är beroende av nätverkets struktur och hur de kan fås att lära från data. Under kursen kommer neurala nätverk att användas för att lösa såväl fiktiva exempel problem som riktiga problem i form av bild klassificering. Målet är att de studerande skall erhålla en baskunskap för hur neurala nätverk kan användas för både regressions- och klassificeringsproblem.

Publisher: Novia University of Applied Sciences, Wolffskavägen 35 B, 65200 Vasa, Finland
© Johan Westö & Novia University of Applied Sciences
Novia Publications and Productions, series L: Teaching materials 2/2014
ISSN: 1799-4195,
ISBN: 978-952-5839-86-9 (online)
Layout: Johan Westö

Contents

1

Course information

This is a course on intelligent systems with a focus on Artificial Neural Networks (ANNs). ANNs are currently experiencing something like a new golden age due to their recent successes on problems related to image and speech recognition (Bengio, Courville, & Vincent, 2012). The purpose of this course is to provide the necessary background information needed about ANNs in order to understand the kind of thinking that has led to their recent successes. Hence, the course targets students in fields where models needs to be learned from data, such as computer science and electrical engineering. Upon completing the course, participating students are expected to have obtained knowledge about 1) how ANNs can solve both regression and classification problems, 2) how gradient based optimization methods can be used for training ANNs, 3) how layering allows networks to solve non-linear problems, and 4) why deep ANNs are thought to be useful.

Most of the material used is taken from Haykin (2009), but today it is also possible to find excellent free courses online. I would recommend visiting Coursera's homepage. There you can find a really good course on "Machine Learning" taught by Andrew Ng (co-founder of Coursera by the way), and an equally good course on "Artificial Neural Networks" taught by Geoffrey Hinton (a.k.a. the godfather of neural networks). I would also recommend an article about deep neural networks and the human brain by Laserson (2011) (found here) as inspiration.

Prerequisites: This material assumes that students are familiar with 1) linear algebra (matrix operations), 2) calculus (the gradient and partial derivatives), 3) system modelling, and 4) MATLAB.

2

Introduction

Today more and more appliances connect to the internet all the time, and this increased connectivity is accompanied by an increased ability to collect and store data. Factories have more sensors collecting data about their processes, and companies such as Facebook and Google sit on a wealth of information about their users; but how do we make sense of all this data? One way is to let computers build models from it, which is what this course is about. More specifically, this course will give an introduction to how we can teach machines to learn from data using ANNs. We will start by first looking at how linear regression can be represented by one artificial neuron and how we can use gradient descent to solve optimization problems. From here, we will move on to see how softmax regression can complement linear regression in solving problems related to classification. Finally, we will look into how multilayer perceptron networks emerges out of both linear and softmax regression models by first first making a non-linear projection of the input data.

2.1 Intelligence

What is meant by intelligence? Different people will probably have different opinions, but what follows is one that is currently receiving a lot of attention. Karl Friston, who is a famous neuroscientist, proposed that several different brain theories could be gathered under one concept (Friston, 2010). Within this concept, the brain's functionality could be looked upon as minimizing surprise. If the brain is thought to be intelligent, one meaning of intelligence would then be the ability to make correct predictions. This definition is by no means a new one, and when asked, "What is intelligence if it is not defined by behaviour?" Jeff Hawkins replied (Hawkins, 2004, p. 6):

> The brain uses vast amounts of memory to create a model of the world. Everything you know and have learned is stored in this model. The brain uses this memory-based model to make continuous predictions of future events. It is the ability to make predictions about the future that is the crux of intelligence. I will describe the brain's predictive ability in depth; it is the core idea in the book.

Similarly, in a translation on Sun Tzu's "The Art of War" by Cleary (1988, p. xi), the following two statements are found.

According to an old story, a lord of ancient China once asked his physician, a member of a family of healers, which of them was the most skilled in the art. The physician, whose reputation was such that his name became synonymous with medical science in China, replied, “My eldest brother sees the spirit of sickness and removes it before it takes shape, so his name does not get out of the house. My elder brother cures sickness when it is still extremely minute, so his name does not get out of the neighbourhood. As for me, I puncture veins, prescribe potions, and massage skin, so from time to time my name gets out and is heard among the lords.”

Just as the eldest brother in the story was unknown because of his acumen and the middle brother was hardly known because of his alacrity, Sun Tzu also affirms that in ancient times those known as skilled warriors won when victory was still easy, so the victories of skilled warriors were not known for cunning or rewarded for bravery.

Despite that these Chinese stories refer to old texts (2500 years) they still agree with the interpretation of intelligence being the ability to foresee events, even if it is disguised as skill in this particular case. Going back to Friston’s idea that brain functionality tries to minimize surprise, we see that predictions should not be restricted to only foreseeing the future. A definition of intelligence based on predictive capability could also include the ability to make correct judgements of data. That is, if an object recognized to be a car actually is a car, then this also corresponds to a situation were surprise is minimized. So, if a system is able to either make correct judgements of data or predict future events, we could call it an intelligent system. However, people have discovered during the last 50 years that programming intelligent systems explicitly is very difficult, one often fails to note the complexity of a task. For example, humans easily recognize Figure 2.1a as a car, but it is very difficult to tell a computer how to infer the same thing from the RGB matrices representing an image (see Figure 2.1b), and the task gets even more difficult if one has to also allow for all different types of cars and viewpoints. One way to solve this problem would then be to write code for instructing a computer on how to learn the task, instead of trying to tell it explicitly how to do it. This way of thinking leads us straight to the next section on machine learning.

2.2 Machine learning

The material in this course will mainly relate to a branch of Artificial Intelligence (AI) called machine learning, a term defined by Mitchell (1997, p. 2) as:

Definition

“A computer program is said to **learn** from experience E with respect to some task T and performance measure P, if its performance at task T, as measured by P, improves with experience E.”

In practise, this means that machine learning applications include among other things: face detection, object recognition, cluster analysis, recommender systems, fault detection, spam detection, and automatic speech recognition systems. In all these situations, a machine learning algorithm has learned to perform the task from experience (old data).

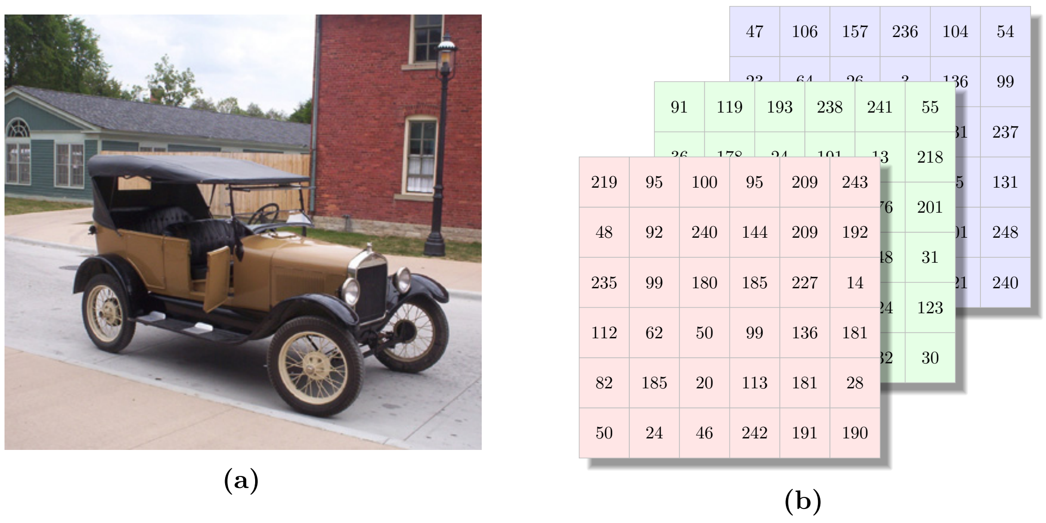


Figure 2.1: Image recognition: a) a image of a T-Ford as seen by a human, adapted from Wikipedia (n.d.-a), and b) a color image as seen by a computer.

Despite the wide spread use, machine learning algorithms can normally be classified as belonging to one of the following three categories:

Supervised learning
Represent situations were each data point is associated with a desired output. Normal tasks include classification when the answer is a label (face / not a face) and regression when the answer is a real value (function fitting).

Unsupervised learning
Seeks to find structure in unlabelled data, examples include cluster analysis and dimensionality reduction.

Reinforcement learning
Focuses on on-line trial and error learning where a computer program tries to improve its performance on a task by testing different actions and evaluates the responses observed.

This course will only look at methods belonging to the first category, and focus is put on methods based on ANNs. The reason for this is “deep learning” which is a collection name for different types of ANNs with a “deep” hierarchical structure. These networks are especially good at image and speech recognition, and they are currently receiving a lot of attention from big companies, such as Microsoft, Google, and Facebook. Both Geoffrey Hinton and Yuan LeCun, who are big names within the field, have recently been hired by Google and Facebook respectively (Mcmilan, 2013; Metz, 2013).

2.2.1 Traditional machine learning versus deep learning

Several problems within machine learning faces high dimensional data, e.g. the dimensionality of image data corresponds to the number of pixels in the image. Richard Bellman pointed out already 50 years ago that the learning complexity increases exponentially as the dimensionality of the data increases linearly, and he named this problem the “curse of dimensionality”. Traditionally machine learning methods have tried to avoid this problem by first performing dimensionality reduction, or feature extraction as it is also commonly called (Arel, Rose, & Karnowski, 2010).

In simpler terms, the above means that machine learning traditionally have operated in two stages. First, the dimensionality of the problem have been decreased by extracting features, whereupon these features have been fed to a machine learning algorithm selected for the task at hand. Examples of feature extraction methods are "Bag of Words" for text data and Fourier transforms for temporal and spatial data. As feature extraction is performed first, the success of many machine learning algorithms is strongly dependent on how well the extracted features can represent variation in the original data. Furthermore, the feature extraction process is normally labour intensive and often performed by humans. Taken together, the above can be seen as a serious weakness or obstacle in reaching a wide spread deployment of intelligent systems (Bengio et al., 2012).

Deep learning methods differ in the way that they try to handle the curse of dimensionality. Instead of relying on human ingenuity, these methods strive to incorporate the feature extraction process in the machine learning algorithm by taking inspiration from the neocortex (Arel et al., 2010). This is the wrinkled outermost layer of the brain, and it is thought to be responsible for our cognitive abilities. Hence, this is also where the visual and auditory cortices are located. Investigations of these have revealed that sensory information is progressed trough a hierarchical structure were higher level information is extracted as one moves up the hierarchy. For vision, this means that edges are detected at the lowest levels, whereas more complex objects such as cars and faces would detected at higher levels (Poggio & Ullman, 2013). In hierarchical systems, depth can also be used as a measure of the number of levels, and hence, deep learning, as we shall see, corresponds to ANNs with several levels or layers as it is also normally called.

2.3 Neurons and neural networks

Your brain consist of around 86 billion connected neurons, recently verified by Herculano-Houzel (2012), and each neuron can connect to up to 10^4 other neurons (Squire & Kandel, 2009). In more detail, each neuron is a type of cell capable of sending and receiving impulses. Functionally, the neuron receives impulses from other neurons on its dendrites, and these can trigger the neuron to send an electrical impulse of its own down the axon, which in turn connects to other neurons (see Figure 2.2). The sites that connect different neurons to each other are called synapses, and contradictory to the neuron's electrical impulse, signals are transmitted chemically within the synapse using substances called neurotransmitters.

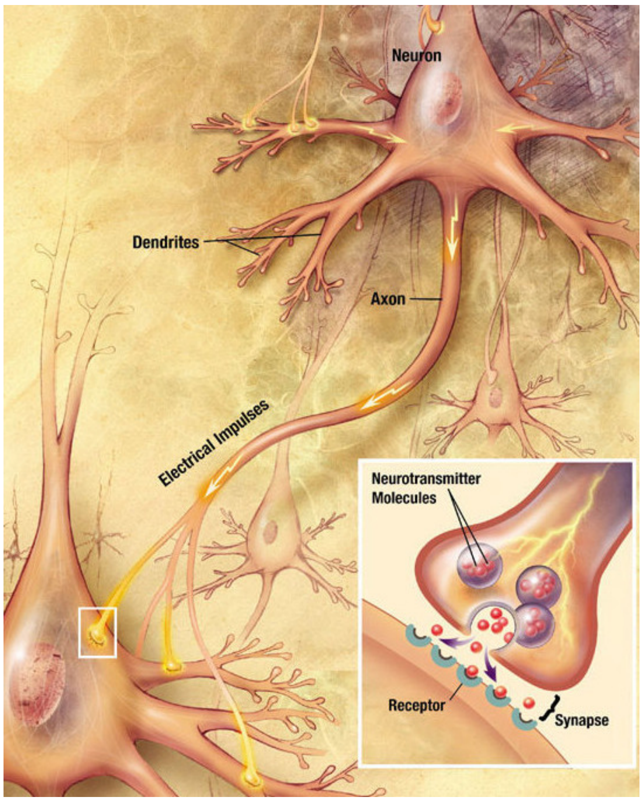


Figure 2.2: Biological neuron, adapted from Wikipedia (n.d.-b).

The term neural network is used to describe networks of connected neurons, and one example of a neural network is therefore the brain. But what is it about these types of networks that make it possible to store memories or information? Ramón Cajal, a Nobel laureate in Physiology and Medicine 1906, proposed that memories are stored trough alterations of the synapses connecting neurons. In other words, he proposed that synapses could form connections of various strengths, and that these strengths were plastic in the sense that they could change over time. In 2000, Eric Kandel received the Nobel prize in Physiology and Medicine for his work on verifying the above statement and for describing how this process occurs in real synapses (Squire & Kandel, 2009).

Even if changes in the brain's synapses constitute the basis for memories, one should not imagine memories as being stored in any specific location. That is, there is no specific place where a memory is stored, instead memories are stored in a distributed fashion throughout the network. This property is maybe best captured by Lashley (1950) who spent his carrier searching for a specific memory trace, or engram as he called it, and arrived at the following conclusion:

> This series of experiments has yielded a good bit of information about what and where the memory trace is not. It has discovered nothing directly of the real nature of the engram. I sometimes feel, in reviewing the evidence on the localization of the memory trace, that the necessary conclusion is that learning just is not possible.

The message one should take home from the above is then not that learning is not possible, but rather that finding one or more synapses that specifically code for a memory trace might very well be.

Before moving on to ANNs, there is still one more detail of interest. As the brain performs so many different tasks, neural networks clearly posses interesting features from an intelligent systems viewpoint; but what is the chance of us being able to utilize them to the same degree as the brain is doing? And, how do we know that we are not just going to end up copying lots of different learning mechanism? That is, is it possible that there could be one general learning method for neural networks, utilized by the brain, that also could be implemented for ANNs? The real answer is that we do not know yet, but findings such as the one by Von Melchner, Pallas, and Sur (2000) provide hope. In this study, the signals coming from the eyes were rerouted to the auditory cortex in newborn ferrets. The purpose was to investigate if the auditory cortex could learn to process visual information and this seems to be the case. Therefore, there is some hope for the existence of one general learning algorithm for neural networks.

2.3.1 Artificial neuron

Similarly to real neural networks, ANNs are built up from connected neurons; but in this case from artificial neurons. Different models for artificial neurons exist, but for the purpose of this course all neurons will be of the model described in Figure 2.3. This model uses a vector ($\mathbf{w}$) with connection weights to indicate synaptic strengths to inputs in a vector ($\mathbf{x}$), a summation for determining the induced field (v), and an activation function ($\varphi(.)$) to calculate the output ($\hat{y}$). These calculations can be described mathematically as:

$$\begin{aligned}\hat{y} &= \varphi\left(\sum_{m=0}^{M} w_m x_m\right)\\ &= \varphi(\mathbf{w}^\intercal \mathbf{x}) \qquad (2.1)\end{aligned}$$

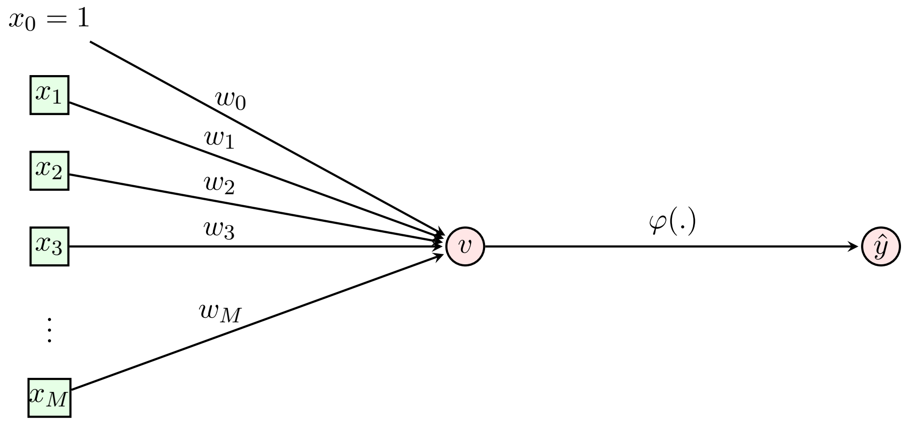


Figure 2.3: Artificial neuron.

2.4 Used notation

The notation used is as follows: upper case bold letters represent matrices, lower case bold letters represents vectors, lower case letters with subscripts represent elements in matrices or vectors (if the vector only contains one element the subscript is left out), and finally, superscripts with Roman numerals represents depth in a hierarchy when not obvious from context. A complete list of all used notations and symbols is found on page 43.

2.5 MATLAB

The Matrix Laboratory (MATLAB) environment is widely used within both academia and industry, and it will be used throughout this course to illustrates different learning examples. MATLAB is, as the name implies, well suited for matrix and vector operations; and these type of operations should be favoured over loops whenever possible. One neat feature of our used artificial neuron is therefore that the induced field is determined by the dot product[1] of the inputs and the weights. This means that the model output ($\hat{\mathbf{Y}}$) for all data points ($\mathbf{X}$) can be easily determined as:

```
% Calculating the locally induced field
% Please note that W is transposed!
V = W'*X;
% Calculating yHat (assuming that the activation function
% can handle matrices
YHat = phi(V)
```

and these calculations still work even if there are several neurons attached to the inputs. In the chapters that follow, we will use this model of a neuron to see 1) how we can represent models, 2) how the model depends of the network's structure, and 3) how we can make the network learn from data.

[1]**The dot product** between two vectors **a** and **b** is obtained as the sum of an element wise multiplication: $\sum_{i=1}^{|a|} a_i b_i$.

3

Linear regression

In linear[2] regression, the aim is to fit a hyperplane[3] to a set of data points. Such models can be visualized as a single neuron connected to every element in the input vector. We will begin by looking at a simple example with one dimensional input and output data, and we will assume that the process generating the data can be described with the following model:

$$\hat{y} = w_0 + x_1 w_1 + \varepsilon \tag{3.1}$$

where ε is an error term representing the influence of unknown or not measured terms and noise. This model can be described by a single neuron, of the same form as the one in Figure 2.3, if we assume that the activation function ($\varphi(.)$) simply returns the argument unchanged. In this case, we could then model the process with a neuron looking like the one in Figure 3.1.

At this stage however, our input data ($\mathbf{x}$) only has dimensionality one, whereas the dimensions for the weight vector ($\mathbf{w}$) would require it to have a dimensionality of two. As can be seen from Equation 3.1 though, w_0 should always be multiplied with one. w_0 hence represent what is called a bias term, and it is present in all models we will look at. For this reason, the simplest fix is to always add a row of ones to the input data matrix ($\mathbf{X}$) ($\mathbf{X}$ then gets the dimensions $M + 1$ by N). This is also the reason why the indexing here start from zero when all other indexes starts from one.

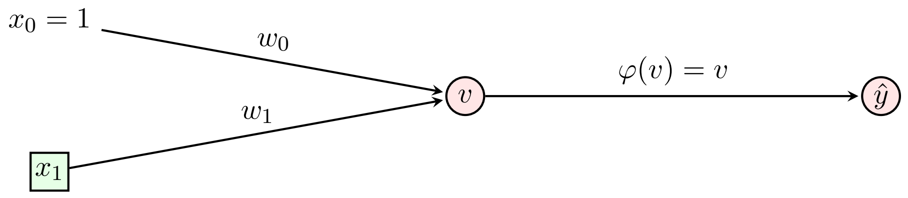


Figure 3.1: Linear regression neuron ($x_0 = 1$ and $w_0 =$ bias).

[2]**A linear function** must satisfy $f(x+y) = f(x)+f(y)$ and $f(ax) = af(x)$ for all a (Lay, 2012, p. 65).
[3]**A hyperplane** is a plane with dimensionality $D - 1$ where D is the dimensionality of space (Lay, 2012, p. 440). In two dimensional space, the hyperplane then becomes a line.

Important

Always remember to add a row of ones to the top of your observed data matrix ($\mathbf{X}$) (x_0 should always equal 1).

3.1 A quadratic error measure

The problem in linear regression then boils down to choosing values for the weights ($\mathbf{w}$), so that the model can explain the observed output (y) as good as possible. In order to know what is good and what is bad, we have to define some kind of error measure that scores each possible combination of weight values. To this end, lets start of by defining an error signal (e) as the difference between each desired output (y), indexed with n, and the corresponding model output ($\hat{y}$).

$$e(n) = y(n) - \hat{y}(n) \tag{3.2}$$

Using the error signal, we then define an instantaneous error energy ($\mathscr{E}$) as:

$$\mathscr{E}(n) = \frac{1}{2}e^2(n) \tag{3.3}$$

where the term $\frac{1}{2}$ is added for mathematical convenience (simpler derivative). Finally, we define the average error energy ($\mathscr{E}_{av}$) to be:

$$\begin{aligned}
\mathscr{E}_{av} &= \frac{1}{N}\sum_{n=1}^{N}\mathscr{E}(n) \\
&= \frac{1}{2N}\sum_{n=1}^{N}e^2(n) \\
&= \frac{1}{2N}\sum_{n=1}^{N}[y(n) - \hat{y}(n)]^2 \\
&= \frac{1}{2N}\sum_{n=1}^{N}\left[y(n) - \sum_{m=0}^{M}w_m x_m(n)\right]^2 \qquad (3.4)
\end{aligned}$$

where it has been assumed that a linear regression output neuron is used with the activation function $\varphi(v) = v$. As $\mathscr{E}$ squares the observed error signal, $\mathscr{E}_{av}$ will obtain positive contributions from all errors signals, independently if they have a positive or negative sign. Hence, the lower the value of $\mathscr{E}_{av}$ the better the model can explain the observed data. The most optimal choice of model weights is therefore obtained at the minima of $\mathscr{E}_{av}$ with respect to different weight combinations. However, one should keep in mind that these model weights are only optimal for the currently used quadratic error measure.

3.2 Example data (part 1)

Lets look at an example to illustrate what have been said so far. Figure 3.2a plots 100 data points generated by the process:

$$y = 1 + 0.5x + r \tag{3.5}$$

where r is a random normally distributed variable with mean zero and standard deviation 0.5. To this data, we will try to fit the model in Figure 3.1. We are not yet in a position to determine what the optimal weight values are, but as there are only two weights in this particular case, we can plot a surface illustrating how $\mathscr{E}_{av}$ varies for different weight combinations. This is done in MATLAB by calculating $\mathscr{E}_{av}$ as:

```
E = Y - Yhat;
En = 1/2*E.*E;
En_av = mean(En);
```

over a grid of different weight combinations and visualizing the result as a surface plot. Such a surface is shown in Figure 3.2b, and despite that the curvature along the w_0 axes is small, it is still possible to get an idea of where the minima is located.

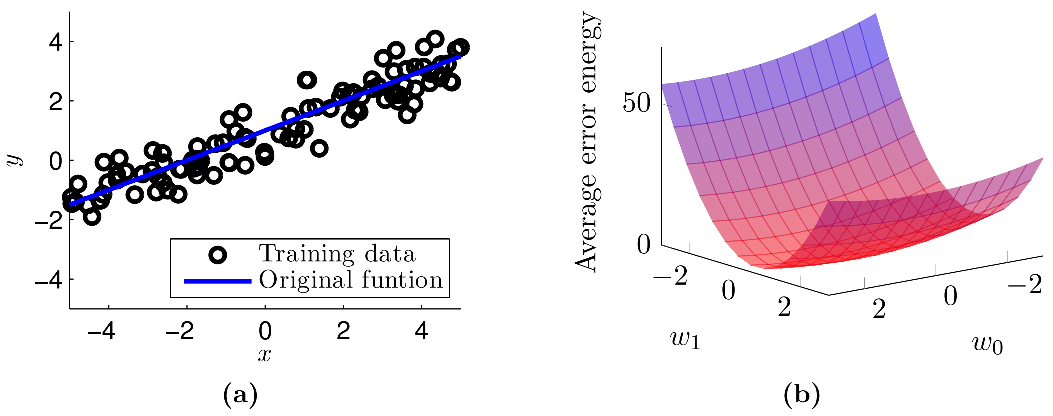


Figure 3.2: Example data: a) 100 data points, selected randomly in the interval [-5 5], generated by the process in Equation 3.5 (black circles) together with a blue line representing the same process without the stochastic variable r, b) $\mathscr{E}_{av}$ as a function of w_0 and w_1 in the interval [-3 3].

Important

Before proceeding, one should ask if this is the only minima that can be found. It this case it is, but it is not always the case and we will therefore return to this question later.

3.3 Finding optimal weight values

For simple problems, it is often possible to analytically determine the exact weight values where $\mathscr{E}_{av}$ has its minima. Linear regression belongs to this group of simpler problems, but in order to prepare for more difficult challenges ahead, we will look at a gradient based iterative method called gradient descent (also known as the method of steepest descent). Given any set of weights, this method evaluates the curvature of the energy surface (Figure 3.2b); and adjusts the parameters so that a step is taken in the direction of steepest descent. From calculus, we know that the gradient[4] of a function gives the direction of steepest ascent. Hence, with gradient descent we therefore want to update our current weights so that a step is taken in the negative direction of the gradient. Mathematically, we define this parameter updating process as:

[4]**The gradient** to a function is a vector where each element is the partial derivatives of the function with respect to a certain variable. In our case, the variables are represented by our model's weights.

$$w_m^{\text{new}} = w_m^{\text{old}} - \eta \frac{\partial \mathscr{E}_{av}}{\partial w_m^{\text{old}}} \tag{3.6}$$

where η is a parameter that determines the step size. In order to simplify things further ahead, we will at this point also introduce the following shorthand notation for the weight change ($\Delta \mathbf{w}$) in Equation 3.6.

$$\Delta w_m = \eta \frac{\partial \mathscr{E}_{av}}{\partial w_m} \tag{3.7}$$

The next step is then to obtain an expression for the partial derivatives of E_{av}. We begin by noting the similarity:

$$\frac{\partial \mathscr{E}_{av}}{\partial w_m} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial \mathscr{E}(n)}{\partial w_m(n)} \tag{3.8}$$

From here, we derive partial derivatives of $\mathscr{E}(n)$ by implementing the chain rule[5] as:

$$\frac{\partial \mathscr{E}(n)}{\partial w_m(n)} = \frac{\partial \mathscr{E}(n)}{\partial e(n)} \frac{\partial e(n)}{\partial \hat{y}(n)} \frac{\partial \hat{y}(n)}{\partial v(n)} \frac{\partial v(n)}{\partial w_m(n)}$$

where

$$\begin{aligned}
\frac{\partial \mathscr{E}(n)}{\partial e(n)} &= \frac{\partial \frac{1}{2} e^2(n)}{\partial e(n)} = e(n) \\
\frac{\partial e(n)}{\partial \hat{y}(n)} &= \frac{\partial y(n) - \hat{y}(n)}{\partial \hat{y}(n)} = -1 \\
\frac{\partial \hat{y}(n)}{\partial v(n)} &= \frac{\partial \varphi(v(n))}{\partial v(n)} = \frac{\partial v(n)}{\partial v(n)} = 1 \\
\frac{\partial v(n)}{\partial w_m(n)} &= \frac{\partial \sum_{m=0}^{M} w_m(n) x_m(n)}{\partial w_m(n)} = x_m(n)
\end{aligned}$$

therefore

$$\frac{\partial \mathscr{E}(n)}{\partial w_m(n)} = -e(n) x_m(n)$$

and

$$\frac{\partial \mathscr{E}_{av}}{\partial w_m} = -\frac{1}{N} \sum_{n=1}^{N} e(n) x_m(n) \tag{3.9}$$

Equation 3.9 then tells us in which direction, in weight space, we should move our weights so that $\mathscr{E}_{av}$ decreases.

3.3.1 Running gradient descent

Using partial derivatives, we have a method for updating the model's weights so that $\mathscr{E}_{av}$ decreases; but this requires that we already have a set of weights that we are trying to improve. To get started, we therefore need to select a set of initial weight values. Previous knowledge about the problem can here be used, but if no such knowledge exist, it is common to simply generate a set of random values. Based on this, the complete algorithm for gradient descent is summarized in Algorithm 1, where it has been assumed that the iterative process continues until convergence. That is, until a point where $\mathscr{E}_{av}$ is no longer decreasing.

[5] **The chain rule** says that $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$, assuming z to be a function of a variable y which int turn is a function of a variable x (Croft, Davison, & Hargreaves, 2001, p. 368).

```
$\mathbf{w} \leftarrow$ random initial weights [-1 1]
$i \leftarrow 1$ {epoch[a] counter}
Iteration loop for gradient descent
**repeat**
  Loop over all training examples (replace with matrix operation in MATLAB)
  **for** $n = 1$ **to** $N$ **do**
    $v(n) \leftarrow \mathbf{w}^\intercal \mathbf{x}(n)$
    $\hat{y}(n) \leftarrow v(n)$
    $e(n) \leftarrow y(n) - \hat{y}(n)$
    $\mathscr{E}(n) \leftarrow \frac{1}{2} e^2(n)$
    $\mathscr{E}_{av} \leftarrow \mathscr{E}_{av} + \frac{1}{N}\mathscr{E}(n)$
    $\Delta\mathbf{w} \leftarrow \Delta\mathbf{w} - \frac{\eta}{N}\mathbf{x}(n)e(n)$
  **end for**
  $\mathscr{E}_{av}(i) \leftarrow \mathscr{E}_{av}$
  Plot progress {check that $\mathscr{E}_{av}$ is decreasing}
  $\mathbf{w} \leftarrow \mathbf{w} - \Delta\mathbf{w}$
**until** convergence {$\mathscr{E}_{av}$ is no longer decreasing}
```

[a] **An epoch** is when training networks referred to one complete run trough the training set.

Algorithm 1: Linear regression using gradient descent.

We have seen earlier that the for loop, in Algorithm 1, can be replaced with matrix multiplications when calculating $\mathbf{V}$, $\hat{\mathbf{Y}}$, $\mathbf{E}$, and $\mathscr{E}_{av}$. Similarly, $\Delta\mathbf{w}$ can also be calculated directly in MATLAB using

```
E = Y - Yhat;
dw = -eta * 1/N * X*E';
```

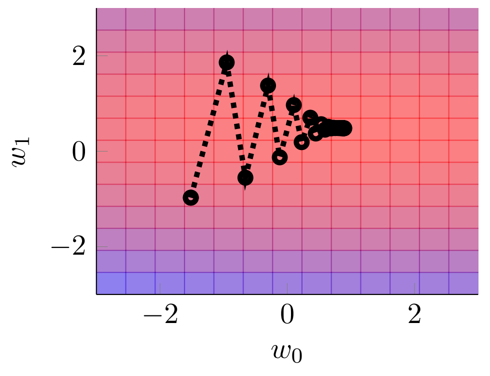
This speeds up the calculations drastically when large datasets are used. Finally, using Equation 3.6 we update the weights in MATLAB as:

```
w = w - dw;
```

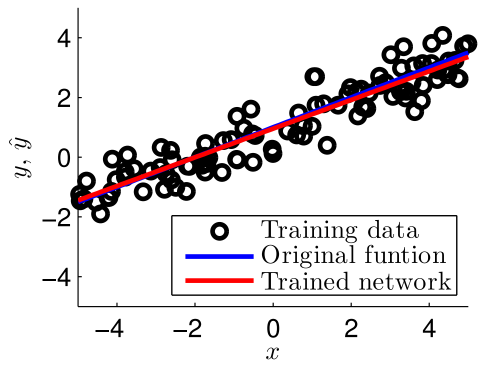
It could here be noted that MATLAB also has built in optimization routines and that several of these are gradient based. It is therefore possible to use these routines instead of gradient descent, but we will stick with gradient descent throughout this course.

3.4 Example data (part 2)

Now, when we have a general method for finding the minima of $\mathscr{E}_{av}$, we can implement it on the example studied earlier. Starting from random initial weight values, the result after 30 iterations with gradient descent is shown in Figure 3.3a. As the energy surface has the form of a long valley, the direction of gradient descent will not necessary point towards the minima. Too large η values can therefore make the algorithm take big leaps that end up increasing $\mathscr{E}_{av}$. In Figure 3.3a, η is on the verge of becoming too large and this is illustrated by the zigzag path taken. Nevertheless, the algorithm was able to find good model parameters after only 30 iterations (red line in Figure 3.3b).



(a)



(b)

Figure 3.3: Gradient descent: a) 30 iterations from random initial parameters with $\eta = 0.15$. b) 100 data points from Equation 3.5 (black circles), a blue line representing the same process without the stochastic variable r, and a red line representing the model obtained from the trained neuron.

Important

The gradient descent algorithm might become unstable and diverge if η is chosen to large.

3.5 Final remarks on gradient descent

One might wonder about why we are using gradient descent for finding the minima when several algorithms exist that are more efficient. The reason is that gradient descent is very simple and intuitive, and it can also be implemented in batch or online mode. Batch mode corresponds to the description given above where the contributions from all training examples are summed up before the weights are updated. That is, the output for each data point is evaluated using the same weights before the update is performed. In online mode, the weights are instead updated continuously using the individual partial derivatives obtained from each data point.

Batch and online mode are therefore both extreme cases. One uses all training examples to update the weights, whereas the other uses only one. Between these two extremes we find something called mini batch, and this is one of the main reasons why gradient descent is still sometimes used. Imagine having a million training examples. If you now implement gradient descent using batch mode, you will have to do a lot of calculations before any progress can be done; but you will know exactly in which direction to move the weights. Online mode requires a lot less calculations before any weights are updated, but here you have only considered one data point; and the direction suggested is not likely to be same as the one suggested by the whole batch. However, if you randomly selected a subset consisting of ten to a hundred thousand training examples (a mini batch). The direction suggested by this mini batch would most likely point in approximately the direction suggested by the batch, but you would get this information at a fraction of the cost that it would take to evaluate all data points. As Equation 3.9 includes a sum over all data points, the presented gradient descent algorithm can be used very easily with mini batches, as this only requires that the sum is restricted to a subset of the data points.

4

Softmax regression

UNFORTUNATELY softmax regression is a bit of a misnomer. We saw already in chapter 2 that the term regression is used for real valued outputs, whereas the term classification is used for labelled outputs. Softmax regression is, however, a classification algorithm despite its name.

In softmax regression,[6] each class (k) is represented by one neuron; and each neuron represent the probability that a given input vector belongs to its corresponding class. The vectors $\mathbf{y}$ and $\hat{\mathbf{y}}$ must therefore both sum up to 1 (probabilities have to sum to 1). For $\mathbf{y}$, the class label is known and this vector therefore contains a 1 (indicating 100 % confidence) for the correct class and zeros for the rest. For $\hat{\mathbf{y}}$, the summation constraint is in turn satisfied by using the following activation function:

$$\phi(v_k) = \frac{e^{v_k}}{\sum_{k=1}^{K} e^{v_k}} \tag{4.1}$$

The probability, given by our model, that an input vector belongs to class k is therefore given by:

$$\hat{y}_k = p(\text{class} = k|\mathbf{x}) = \frac{e^{\mathbf{w}_k^\intercal \mathbf{x}}}{\sum_{k=1}^{K} e^{\mathbf{w}_k^\intercal \mathbf{x}}} \tag{4.2}$$

In MATLAB, we can make use of built in functions and matrix operations to calculate $\hat{\mathbf{Y}}$ from $\mathbf{V}$ as:

```matlab
% Assuming V has dimensions K by N
Yhat = exp(V) ./ ( ones(size(V,1),1) * sum(exp(V)) );
```

The summation over all induces fields in Equation 4.2 requires that each output is connected to all the induced fields. Hence, we will represent softmax regression with the structure in Figure 4.1.

[6]**Softmax regression** is a generalization of the more common logistic regression algorithm to more than two classes. This generalization is also known as multinomial logistic regression.

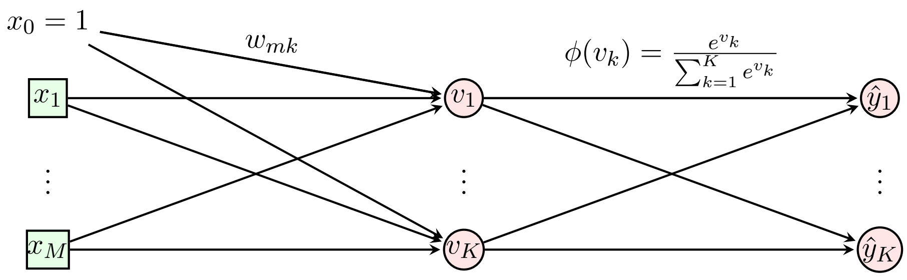


Figure 4.1: Softmax regression model for classification between K classes.

4.1 Maximum likelihood

Softmax regression output probabilities for different classes, and hence, weight selection should be based upon how likely the observed combination of inputs and desired outputs would be for a given set of weights. As an example, imagine that you are trying to model human height by determining the mean (μ) and the standard deviation (σ), and that you have been given a sample containing the heights of 1000 randomly selected persons from the entire human population. Based upon our sample, we can calculate how likely we are to observe it for different values of μ and σ with a likelihood function ($\mathscr{L}$), defined as $\mathscr{L}(\mu, \sigma|\text{sample})$. With this definition, the most most logical choice of model parameters would be found at the maxima of $\mathscr{L}(\mu, \sigma|\text{sample})$; and the task hence becomes an optimization problem, which again could be solved using gradient descent.

Similarly, in softmax regression we are interested in finding the weights for our neurons that would be the most likely ones given the available data, and for our case, the likelihood that our model generated one data point is given by:

$$\mathscr{L}(\mathbf{w}|\mathbf{x}(n)) = \prod_{k=1}^{K} p_k^{y_k(n)}(n) \tag{4.3}$$

Assuming independence between data points, the likelihood function over $\mathbf{X}$ then becomes the product over all observed data points.

$$\mathscr{L}(\mathbf{w}|\mathbf{X}) = \prod_{n=1}^{N} \prod_{k=1}^{K} p_k^{y_k(n)}(n) \tag{4.4}$$

Large products are, however, cumbersome to work with. A normal trick is therefore to work with the mean log likelihood function instead (ℓ), or the mean negative log likelihood function[7] as in this case. We obtain this function by simply taking the logarithm of Equation 4.4 and multiplying the expression by $-\frac{1}{N}$, which gives:

$$\begin{aligned}\ell &= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k(n) \ln(p_k(n)) \\ &= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} y_k(n) \ln(\hat{y}_k(n)) \end{aligned} \tag{4.5}$$

[7]**The negative log likelihood function** is used in order to transform a maximization problem into a minimization problem. Multiplying a function with negative one flips its surface so that the previous maxima becomes a minima.

In order to avoid unnecessary loops, it is a lot faster to calculate ℓ directly in MATLAB using:

```
% Mean negative log likelihood
l = -1/N * sum(sum( Y.*log(Yhat) ));
```

Just as $\mathscr{E}_{av}$ for the linear regression problem, ℓ only have one minima in softmax regression; following the gradient is therefore guaranteed to lead to a global minima. Differentiating Equation 4.5 with respect to the model parameters is somewhat tricky. A complete derivation can be found in Bishop (1995), but we will here just conclude that the derivation in the end gives:

$$\frac{\partial \ell}{\partial w_{mk}} = -\sum_{n=1}^{N} e_k(n) x_m(n) \tag{4.6}$$

where $e_k(n)$ is defined as

$$e_k(n) = y_k(n) - \hat{y}_k(n) \tag{4.7}$$

Quite interestingly, Equation 4.6 is actually identical to what we obtained for linear regression in the previous chapter. At this point, we now have all the information needed to fit models, as the one in Figure 4.1, to labelled data. Using gradient descent, the complete procedure is summarized in Algorithm 2

```
$\mathbf{W} \leftarrow$ random initial weights [-1 1]
$i \leftarrow 1$ {epoch counter}
Iteration loop for gradient descent
**repeat**
  Loop over all training examples (replace with matrix operation in MATLAB)
  **for** $n = 1$ **to** $N$ **do**
    $\mathbf{v}(n) \leftarrow \mathbf{W}^\mathsf{T}\mathbf{x}(n)$
    **for** $k = 1$ **to** $K$ **do**
      $\hat{y}_k(n) \leftarrow \frac{e^{\mathbf{v}_k(n)}}{\sum_{k=1}^{K} e^{\mathbf{v}_k(n)}}$
    **end for**
    $\mathbf{e}(n) \leftarrow \mathbf{y}(n) - \hat{\mathbf{y}}(n)$
    $\ell(n) \leftarrow -\sum_{k=1}^{K} y_k(n) \ln(\hat{y}_k(n))$
    $\ell \leftarrow \ell + \frac{1}{N}\ell(n)$
    $\Delta\mathbf{W} \leftarrow \Delta\mathbf{W} - \frac{\eta}{N}[\mathbf{x}(n)\mathbf{e}^\mathsf{T}(n)]$
  **end for**
  $\ell(i) \leftarrow \ell$
  Plot progress {check that $\ell$ is decreasing}
  $\mathbf{W} \leftarrow \mathbf{W} - \Delta\mathbf{W}$
**until** convergence {$\ell$ is no longer decreasing}
```

Algorithm 2: Softmax regression using gradient descent.

4.2 Gradient checking

Programming mistakes easily occur during implementation of the presented algorithms. In the example we looked at in chapter 3, it was possible to confirm that our selected parameter values moved in the intended direction by looking at how our positioned changed on the $\mathscr{E}_{av}$ surface. As the dimensionality of $\mathbf{X}$ grows larger (more weights) this is no longer possible, and we therefore need another method to verify our calculations. A nice way of verifying that everything is working as intended is to compare the calculated partial derivatives to approximate numerical estimates.

The partial derivative for each weight tells us how much $\mathscr{E}_{av}$ or ℓ should change when that weight is varied. We can easily verify this by adding or subtracting a small number (κ) from a weight and calculating the induced change in $\mathscr{E}_{av}$ or ℓ afterwards. To get an unbiased estimate, its a good idea to take the average of the observed change when κ is both added and subtracted. Mathematically we can describe the estimated partial derivative as:

$$\text{estimated value for } \frac{\partial \ell}{\partial w_{mk}} = \frac{\ell(w_{mk}^{+}) - \ell(w_{mk}^{-})}{2\kappa} \tag{4.8}$$

where

$$w_{mk}^{+} = w_{mk} + \kappa \quad \text{and} \quad w_{mk}^{-} = w_{mk} - \kappa$$

Equation 4.8 can be repeated for every weight and the results can be compared to what is obtained from either Equation 3.9 or Equation 4.6. For small values on κ (10^{-4}), the estimated and computed partial derivatives should be identical for atleast the first three or four decimals.

4.3 2 class example

Lets start off with a two class example so we can get a feel for how softmax regression works. Our problem consist of learning to correctly classify the 100 data points in Figure 4.3a. These data points are generated by the processes:

$$\text{Class 1} \begin{cases} x_1 = 2 + r \\ x_2 = 2 + s \end{cases} \qquad \text{Class 2} \begin{cases} x_1 = -2 + r \\ x_2 = -2 + s \end{cases} \tag{4.9}$$

where both r and s are random normally distributed numbers with mean 0 and standard deviation 1. After adding a row of ones to $\mathbf{X}$, its dimensions are now 3 by 100. Labels are coded using the desired outputs as:

$$\mathbf{y} = \begin{cases} [1\ 0]^{\intercal} & \text{if class} = 1 \\ [0\ 1]^{\intercal} & \text{if class} = 2 \end{cases} \tag{4.10}$$

Hence, the model that we will try to fit looks like the one illustrated in Figure 4.2. Even if Equation 4.5 provides an error measure, it is still interesting to know how often data points are classified incorrectly. For this purpose, we introduce a classification error ratio ($\mathscr{C}$) as:

$$\mathscr{C} = \frac{1}{N} \sum_{n=1}^{N} \begin{cases} 0 & \text{if the predicted class is equal to the assigned class} \\ 1 & \text{otherwise} \end{cases} \tag{4.11}$$

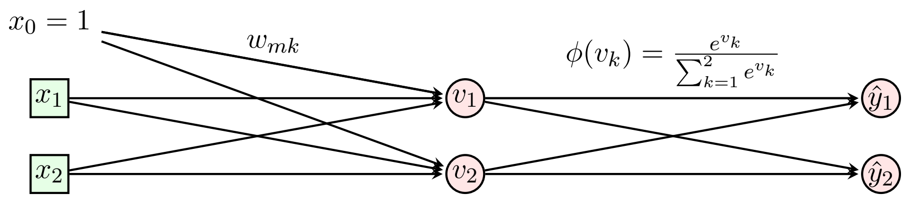


Figure 4.2: Softmax regression model for a two dimensional two class problem.

When calculating $\mathscr{C}$ in MATLAB, we can make use of built in relational operators, but first we need a vector with class labels for both $\mathbf{Y}$ and $\hat{\mathbf{Y}}$. As our model outputs probabilities, we would like to assign a data point to the class with the highest probability. With our matrix definitions, this corresponds to selecting the row in $\mathbf{Y}$ or $\hat{\mathbf{Y}}$ with the maximum value in each column, and taking the row number as the defined or predicted label for each data point. In the end, we can implement all of this using the following three lines of code in MATLAB:

```matlab
% Labels from Y
[~, Y_labels] = max(Y, [], 1);
% Labels from Yhat
[~, Yhat_labels] = max(Yhat, [], 1);
% The classification error ratio
C = mean( Y_labels ~= Yhat_labels );
```

Before running the procedure in Algorithm 2, we will check that our implementation is correct by comparing the calculated and estimated gradient. For random initial weights [-1 1], this comparison showed that both the calculated and estimated partial derivatives are identical for the first four decimals (see Table 4.1).

Table 4.1: Comparison between calculated and estimated partial derivatives for random initial weights [-1 1] and with $\kappa = 10^{-4}$

	$\frac{\partial \ell}{\partial w_{01}}$	$\frac{\partial \ell}{\partial w_{02}}$	$\frac{\partial \ell}{\partial w_{11}}$	$\frac{\partial \ell}{\partial w_{12}}$	$\frac{\partial \ell}{\partial w_{21}}$	$\frac{\partial \ell}{\partial w_{22}}$
Calculated	-0.0142	0.0142	-0.3747	0.3747	-0.4990	0.4990
Estimated	-0.0142	0.0142	-0.3747	0.3747	-0.4990	0.4990

Figure 4.3b illustrates training progress for 30 epochs while running the procedure in Algorithm 2 with $\eta = 0.25$. After these 30 epochs ℓ is hardly decreasing any more, and it was concluded that the algorithm had converged. The decision surface found by the algorithm is illustrated in Figure 4.3c together with a decision surface found using linear regression in Figure 4.3d. Linear regression is clearly not suited for classification tasks, whereas softmax regression is meant to handle labelled data.

4.4 Training and testing

Our hope when training a model is that it will be able to generalize to unseen data. However, as one starts to fit models with more and more weights, there is always a risk that the model starts to learn patterns that are just found in the data used for training. This phenomena is called overfitting, and it can make the model perform significantly worse on unseen data. Even if overfitting has not occurred, models normally perform worse on unseen data because it might contain patterns that were not present in the data used for training. Model performance on training data is therefore not a good estimate on how well the model will perform. Available data is therefore normally divided up into a training set and a test set. The training set is used for training the model, whereas the test set is used for getting an unbiased measure on how well the model performs. A common division is to randomly select around 80 % of the data for training and leave the rest for testing.

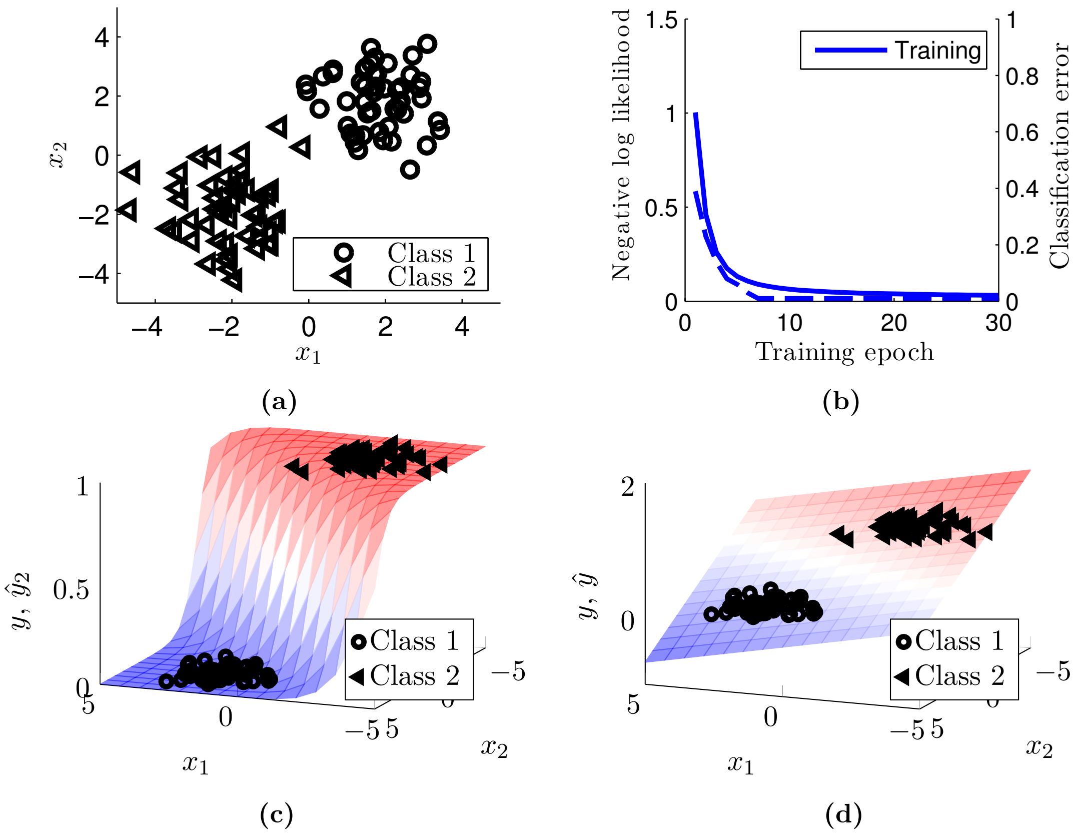


Figure 4.3: a) 50 data points from both class 1 and class 2 generated in accordance with Equation 4.9. b) Training results with random initial weights [-1 1] and $\eta = 0.25$. The blue line represents ℓ and the dashed blue line is the classification error as given by Equation 4.11. c) Decision surface for the softmax regression network trained in Figure 4.3b. d) Decision surface for a linear regression model trained on the data in Figure 4.3a.

4.5 MNIST

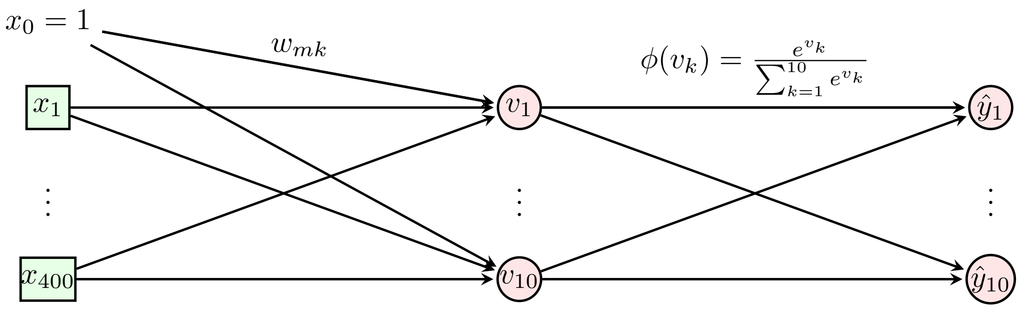
Different people and groups all over the world develop new or improve existing algorithms, and it is often difficult to know which ones are the best. However, there exist several “famous” datasets that people try out their algorithms on and comparisons are hence possible. One well know dataset is the MNIST database of handwritten digits (LeCun, Bottou, Bengio, & Haffner, 1998). This dataset contains 60 000 images (20 by 20 pixels) that are to be used for training and 10 000 more for testing. As can be noted in Benenson (2013), the best reported classification error ratio on the test set is 0.21 %. It could here also be noted that deep learning networks are currently at the top for both this and the other image classification datasets.

The MNIST datasets can be freely downloaded from the MNIST homepage and a function for reading them into MATLAB can be found from Matlab central. Figure 4.4b illustrates 16 images taken from the training set.

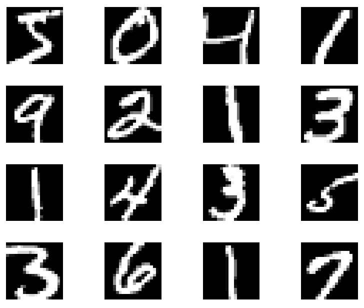
Data in image format have to be concatenated into vector format before we can fit a model using softmax regression. This is accomplished by simply taking each column in the image matrix and placing them on top of each other to form a vector $\mathbf{x}$ with dimensionality 400. As before, we also have to add a row of ones; the dimensions for $\mathbf{X}_{\text{training}}$ then become 401 by 60 000. A similar process for the test set gives $\mathbf{X}_{\text{test}}$ with dimensions 401 by 10 000. Labels are coded using the desired outputs as:

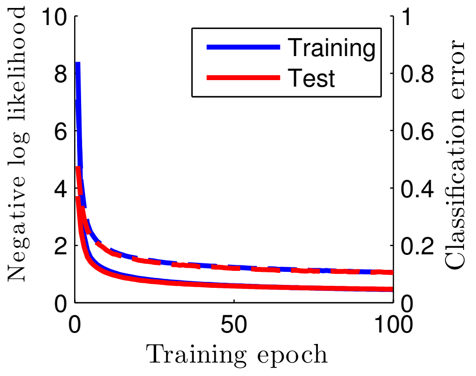
$$\mathbf{y} = \begin{cases} [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^{\mathsf{T}} & \text{if digit} = 1 \\ [0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^{\mathsf{T}} & \text{if digit} = 2 \\ \quad\vdots & \quad\vdots \\ [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1]^{\mathsf{T}} & \text{if digit} = 0 \end{cases} \tag{4.12}$$

$\mathbf{Y}_{\text{training}}$ and $\mathbf{Y}_{\text{test}}$ therefore have the dimensions 10 by 60 000 and 10 by 10 000 respectively. In the end, this means that the model we will try to fit looks like the one illustrated in Figure 4.4a. Training results after 100 epochs of gradient descent using random initial weights [-1 1], $\eta = 2$, and mini batches of 20 % are shown in Figure 4.4c. Running gradient descent for additional epochs could improve the results slightly, but overall it looks like the algorithm has converged, and the final results are summarized in Table 4.2.



(a)





(b) **(c)**

Figure 4.4: a) Softmax regression model for classifying the MNIST data set. b) 16 images from the MNIST training set. c) Training progress, on the MNIST dataset, using softmax regression with random initial weights [-1 1], $\eta = 2$, and mini batches of size 20 %. Solid lines represent ℓ and the dashed line is $\mathscr{C}$.

Table 4.2: Numerical summary of the final training results from Figure 4.4c

ℓ_{training}	ℓ_{test}	$\mathscr{C}_{\text{training}}$	$\mathscr{C}_{\text{test}}$
0.4561	0.4695	0.1051	0.1053

Correctly classified training images	53696
Incorrectly classified training images	6304
Correctly classified test images	8947
Incorrectly classified test images	1053

4.6 Restrictions for linear models

Both linear and softmax regression represent artificial neural networks with a single layer of neurons. This simplicity has the advantage that both problems are convex,[8] but it also brings restrictions on what is possible to achieve. Linear regression can only fit hyperplanes to the observed data and softmax regression can only separate classes using hyperplanes. Softmax regression can therefore only classify all data points correctly if the data is linearly separable. In two dimensions, linearly separable then means that the classes should be separable by a straight line. A simple example that is not linearly separable is the XOR problem.

Important

Softmax regression can only try to separate classes using hyperplanes as boundaries and can therefore only obtain $\mathscr{C} = 0$ on linearly separable problems.

If the models found using linear or softmax regression are unsatisfactory, more complex models will have to be used. One such example are multilayer perceptron networks that are presented in the next chapter.

[8] **Convex** problems only have one minima, and hence gradient descent is guaranteed to find the global minima for either $\mathscr{E}_{av}$ or ℓ.

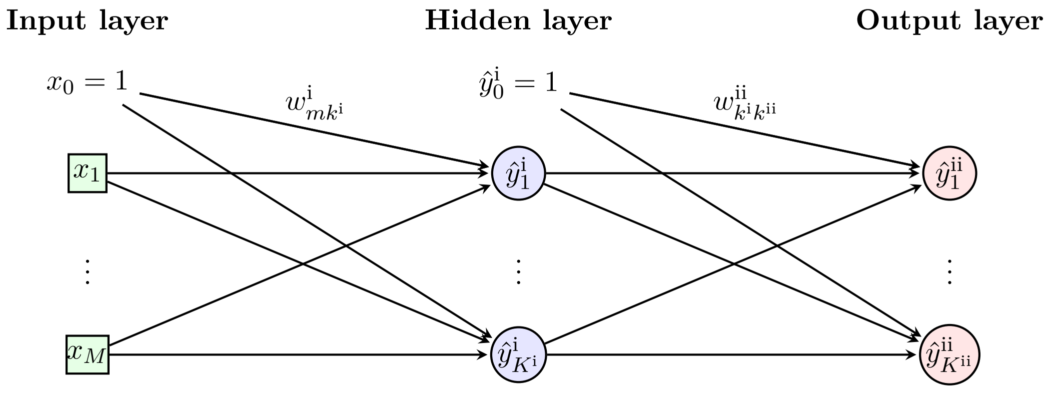
5

Multilayer Perceptrons

The previous chapter ended with a discussion about limitations for one layer networks. It turns out that we can get around these limitations by adding another layer of neurons. This gives us a Multilayer Perceptron (MLP) network (another misnomer unfortunately)[9] and an example is given in Figure 5.1. Previously, we only had an input layer with inputs and an output layer with neurons; but now we have added a hidden layer with neurons in between these two layers (one could also add more than one hidden layer). These neurons in the hidden layer will now function as inputs to the output layer. It is, however, important that neurons in the hidden layer have a non-linear activation function. If not, the network can be truncated down to a single layer and there is no gain in using a hidden layer. At the same time, the activation functions must be differentiable, otherwise, as we shall see, we will not be able to train the network using gradient descent.

Important

Neurons located in hidden layers must have differentiable non-linear activation functions.



Input layer
Hidden layer
Output layer
$x_0 = 1$
$w^{\text{i}}_{mk^{\text{i}}}$
$\hat{y}^{\text{i}}_0 = 1$
$w^{\text{ii}}_{k^{\text{i}}k^{\text{ii}}}$
x_1
$\hat{y}^{\text{i}}_1$
$\hat{y}^{\text{ii}}_1$
x_M
$\hat{y}^{\text{i}}_{K^{\text{i}}}$
$\hat{y}^{\text{ii}}_{K^{\text{ii}}}$

Figure 5.1: MLP neural network with one hidden layer (Roman numerals are used to indicate depth).

[9]**The perceptron** is actually a type of learning algorithm for linear classifiers and does not unfortunately have anything to do with MLP networks.

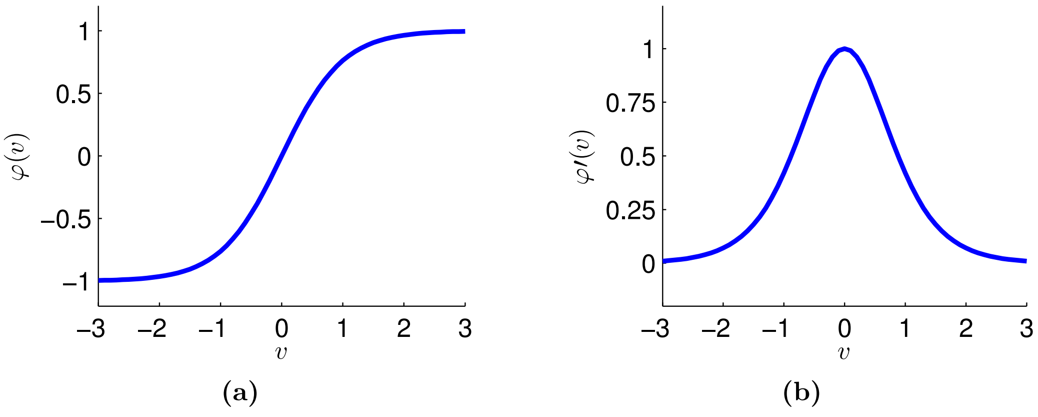


Figure 5.2: a) The hyperbolic tangent function. b) The derivative to the hyperbolic tangent function.

What are then suitable activation functions? A common choice that fulfil the requirements is the hyperbolic tangent function. This function and its derivative are shown in Figure 5.2a and Figure 5.2b, and they can be represented mathematically as:

$$\varphi_{\text{tanh}}(v) = \tanh(v) = -1 + \frac{2}{1+e^{-2v}} \tag{5.1}$$

$$\varphi\prime_{\text{tanh}}(v) = \text{sech}^2(v) = \frac{4e^{2v}}{(e^{2v}+1)^2} \tag{5.2}$$

Independently on if we are performing regression or classification, we will here always use the hyperbolic tangent function as activation function for all neurons in the hidden layer. The output layer, on the other hand, will either consist of a linear regression neuron or softmax regression neurons. We can therefore look upon the hidden layer as a non-linear projection of the inputs before performing linear or softmax regression. Thinking about the XOR problem, our objective then becomes to select weights for the hidden layer so that the classes, after the non-linear projection, becomes linearly separable.

With multiple layers we have to calculate the outputs sequentially. That is, we first determine the outputs for the neurons in the hidden layer, whereupon we determine the outputs for the neurons in the output layer. Similarly, just as we always add a row of ones to $\mathbf{X}$ before calculating $\hat{\mathbf{Y}}^{\text{i}}$, we also have to add a row of ones to $\hat{\mathbf{Y}}^{\text{i}}$ before calculating $\hat{\mathbf{Y}}^{\text{ii}}$. In MATLAB, this is done by:

```matlab
% Calculating the locally induced field in the hidden layer
% Please note that Wi is transposed!
Vi = Wi'*X;
% Calculating outputs from the hidden layer
YiHat = tanh(Vi);

% Adding a row of ones to yiHat
YiHat = [ones(1,size(YiHat,2)); YiHat];

% Calculating induced fields in the output layer
% Please note that Wii is transposed!
Vii = Wii'*YiHat;
```

```
% phi(Vii) is determined by the output layer type
YiiHat = phi(Vii);
```

Now when we know how to calculate the output from the network, the next step is to look at how we should update the weights.

5.1 Backpropagation

As we will use either a linear regression neuron or softmax neurons in the output layer, the layer (ii) weights can be updated as before using the partial derivatives from either Equation 3.9 or Equation 4.6. However, the output layer now obtains its inputs from the hidden layer; we therefore have to change $x_m(n)$ to $\hat{y}^{\text{i}}_{k^{\text{i}}}(n)$ so that we get:

$$\frac{\partial \mathscr{E}_{av}}{\partial w^{\text{ii}}_{k^{\text{i}}}} = -\frac{1}{N}\sum_{n=1}^{N} \hat{y}^{\text{i}}_{k^{\text{i}}}(n) e(n) \tag{5.3}$$

$$\frac{\partial \ell}{\partial w^{\text{ii}}_{k^{\text{i}}k^{\text{ii}}}} = -\frac{1}{N}\sum_{n=1}^{N} \hat{y}^{\text{i}}_{k^{\text{i}}}(n) e_{k^{\text{ii}}}(n) \tag{5.4}$$

Getting the partial derivatives for the layer (i) weights is a bit trickier, but, assuming we are using a linear regression neuron in the output layer, we can again expand the partial derivatives for $\mathscr{E}(n)$ using the chain rule as:

$$\frac{\partial \mathscr{E}(n)}{\partial w^{\text{i}}_{mk^{\text{i}}}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e(n)} \frac{\partial e(n)}{\partial \hat{y}^{\text{ii}}(n)} \frac{\partial \hat{y}^{\text{ii}}(n)}{\partial v^{\text{ii}}(n)} \frac{\partial v^{\text{ii}}(n)}{\partial \hat{y}^{\text{i}}_{k^{\text{i}}}(n)} \frac{\partial \hat{y}^{\text{i}}_{k^{\text{i}}}(n)}{\partial v^{\text{i}}_{k^{\text{i}}}(n)} \frac{\partial v^{\text{i}}_{k^{\text{i}}}(n)}{\partial w^{\text{i}}_{mk^{\text{i}}}(n)}$$

where

$$\frac{\partial \mathscr{E}(n)}{\partial e(n)} = e(n)$$

$$\frac{\partial e(n)}{\partial \hat{y}^{\text{ii}}(n)} = -1$$

$$\frac{\partial \hat{y}^{\text{ii}}(n)}{\partial v^{\text{ii}}(n)} = 1$$

$$\frac{\partial v^{\text{ii}}(n)}{\partial \hat{y}^{\text{i}}_{k^{\text{i}}}(n)} = w^{\text{ii}}_{k^{\text{i}}}$$

$$\frac{\partial \hat{y}^{\text{i}}_{k^{\text{i}}}(n)}{\partial v^{\text{i}}_{k^{\text{i}}}(n)} = \text{sech}^2\left[v^{\text{i}}_{k^{\text{i}}}(n)\right]$$

$$\frac{\partial v^{\text{i}}_{k^{\text{i}}}(n)}{\partial w^{\text{i}}_{mk^{\text{i}}}(n)} = x_m$$

therefore

$$\frac{\partial \mathscr{E}(n)}{\partial w^{\text{i}}_{mk^{\text{i}}}(n)} = -x_m(n)\text{sech}^2\left[v^{\text{i}}_{k^{\text{i}}}(n)\right] e(n) w^{\text{ii}}_{k^{\text{i}}}$$

and (using Equation 3.8)

$$\frac{\partial \mathscr{E}_{av}}{\partial w^{\text{i}}_{mk^{\text{i}}}} = -\frac{1}{N}\sum_{n=1}^{N} x_m(n)\text{sech}^2\left[v^{\text{i}}_{k^{\text{i}}}(n)\right] e(n) w^{\text{ii}}_{k^{\text{i}}} \tag{5.5}$$

Differentiating ℓ is more difficult, but we again obtain an almost identical results (you can verify it using gradient checking). As shown in Equation 5.6, the only difference is a sum over all the neurons in the output layer.

$$\frac{\partial \ell}{\partial w^{\mathrm{i}}_{mk^{\mathrm{i}}}} = -\frac{1}{N}\sum_{n=1}^{N} x_m(n)\mathrm{sech}^2\left[v^{\mathrm{i}}_{k^{\mathrm{i}}}(n)\right]\sum_{k^{\mathrm{ii}}=1}^{K^{\mathrm{ii}}} e_{k^{\mathrm{ii}}}(n) w^{\mathrm{ii}}_{k^{\mathrm{i}}k^{\mathrm{ii}}} \tag{5.6}$$

From both Equation 5.5 and Equation 5.6 we see that the observed error, in the output layer, is propagated backwards along the connecting weights to the hidden layer; it is this mechanism that has given the backpropagation learning algorithm its name. At the same time, we should note that the derivative of the activation function for the neurons in the hidden layer is present in both equations. This is then the reason for the previous statement that the selected activation function must be differentiable. Finally, when summarizing all the above, we obtain the MLP learning method in Algorithm 3.

```
$\mathbf{W}^{\mathrm{i}} \leftarrow$ random initial weights
$\mathbf{W}^{\mathrm{ii}} \leftarrow$ random initial weights
$i \leftarrow 1$ {epoch counter}
Iteration loop for gradient descent
**repeat**
  Loop over all training examples (replace with matrix operation in MATLAB)
  **for** $n = 1$ **to** $N$ **do**
    Forward pass
    $\mathbf{v}^{\mathrm{i}}(n) \leftarrow \mathbf{W}^{\mathrm{i}\mathsf{T}}\mathbf{x}(n)$
    $\hat{\mathbf{y}}^{\mathrm{i}}(n) \leftarrow \tanh(\mathbf{v}^{\mathrm{i}})$
    $\mathbf{v}^{\mathrm{ii}}(n) \leftarrow \mathbf{W}^{\mathrm{ii}\mathsf{T}}\hat{\mathbf{y}}^{\mathrm{i}}(n)$
    $\hat{\mathbf{y}}^{\mathrm{ii}}(n) \leftarrow \varphi(\mathbf{v}^{\mathrm{ii}})$ {select activation function based on used output layer}
    Error
    $\mathbf{e}(n) \leftarrow \mathbf{y}(n) - \hat{\mathbf{y}}^{\mathrm{ii}}(n)$
    **if** Regression **then**
      $\mathscr{E}(n) \leftarrow \frac{1}{2}e^2(n)$
      $\mathscr{E}_{av} \leftarrow \mathscr{E}_{av} + \frac{1}{N}\mathscr{E}(n)$
    **else**
      $\ell(n) \leftarrow -\sum_{k^{\mathrm{ii}}=1}^{K^{\mathrm{ii}}} y_{k^{\mathrm{ii}}}(n)\ln(\hat{y}^{\mathrm{ii}}_{k^{\mathrm{ii}}}(n))$
      $\ell \leftarrow \ell + \frac{1}{N}\ell(n)$
    **end if**
    Backward pass
    $\Delta\mathbf{W}^{\mathrm{ii}} \leftarrow \Delta\mathbf{W}^{\mathrm{ii}} - \frac{\eta}{N}\hat{\mathbf{y}}^{\mathrm{i}}(n)\mathbf{e}^{\mathsf{T}}(n)$
    $\Delta\mathbf{W}^{\mathrm{i}} \leftarrow \Delta\mathbf{W}^{\mathrm{i}} - \frac{\eta}{N}\mathbf{x}(n)\left[\mathrm{sech}^2\left(\mathbf{v}^{\mathrm{i}}(n)\right) \odot \left(\mathbf{W}^{\mathrm{ii}\mathsf{T}}\mathbf{e}(n)\right)\right]^{\mathsf{T}}$
  **end for**
  $\mathscr{E}_{av}(i) \leftarrow \mathscr{E}_{av}$ **or** $\ell(i) \leftarrow \ell$ {regression or classification}
  Plot progress {check that $\ell$ or $\mathscr{E}_{av}$ is decreasing}
  $\mathbf{W}^{\mathrm{ii}} \leftarrow \mathbf{W}^{\mathrm{ii}} - \Delta\mathbf{W}^{\mathrm{ii}}$
  $\mathbf{W}^{\mathrm{i}} \leftarrow \mathbf{W}^{\mathrm{i}} - \Delta\mathbf{W}^{\mathrm{i}}$
**until** convergence {$\ell$ or $\mathscr{E}_{av}$ is no longer decreasing}
```

Algorithm 3: Training MLP networks with gradient descent.

Earlier, we have seen how we can implement the forward pass and the error caclulations in MATLAB without having to do a for loop over the whole training set. As before, we can also do the backward pass without a for loop using:

```
% Error signal
E = Y - YiiHat;
% Weight change for layer (ii)
dWii = -eta * 1/N * YiHat*E';
% Weight change for layer (i)
dWi = -eta * 1/N * X*( (Wii(2:end,:)*E) .* sech(Vi).^2 )';

% Updating weights
Wi = Wi - dWi;
Wii = Wii - dWii;
```

independently on if the output layer consists of a linear regression neuron or of softmax regression neurons. At this stage, we are now then ready to start solving problems using MLP networks.

5.2 The XOR problem

The XOR problem is the simplest non-linearly separable problem, and it is therefore a nice starting point for trying to understand the capabilities of MLP networks. In order to solve the problem, we will try to fit the model in Figure 5.3 to the data in Figure 5.4a.

Running Algorithm 3 for 100 epochs with random initial weights $[-1\ 1]$ and $\eta = 1.5$ resulted in the progress shown in Figure 5.4b. We stated earlier that, in order to solve the problem, the hidden layer have to perform a non-linear projection that makes the data linearly separable. Figure 5.4c shows the actual projections performed by the trained network, and as can be seen, the classes are now linearly separable in the space spanned by $\hat{y}_1^{\text{i}}$ and $\hat{y}_2^{\text{i}}$. This makes it possible for the sotfmax regression neurons in the output layer to learn a model that can classify all the data points correctly. We can also see from the dotted line in Figure 5.4b that this is indeed the case, and furthermore, Figure 5.4d shows how the network generalizes by plotting its decision surface.

The addition of a hidden layer have also brought with it an unwanted consequence, the surface formed by $\mathscr{E}_{av}(\mathbf{W})$ or $\ell(\mathbf{W})$ is no longer convex. This means that there is no guarantee that the gradient will guide us towards a global minima. In Figure 5.4e and Figure 5.4f, both the training progress and the decision surface are shown for a case when the algorithm got stuck in a suboptimal minima. This is unfortunately a built in feature of gradient based algorithms and, therefore, a problem that we have to be aware of.

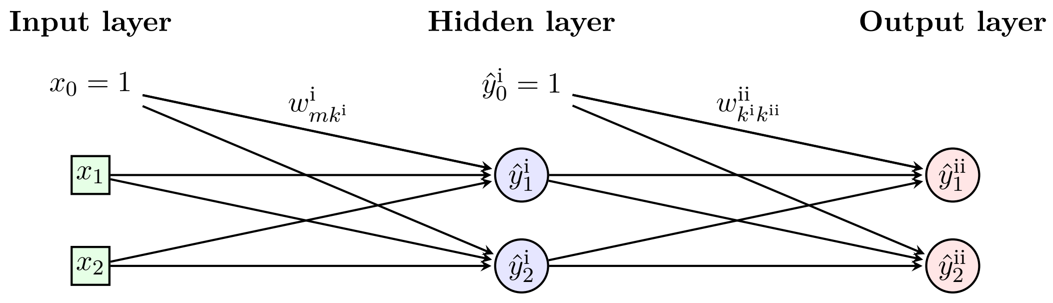


Figure 5.3: MLP network used for solving the XOR problem (Roman numerals are used to indicate depth).

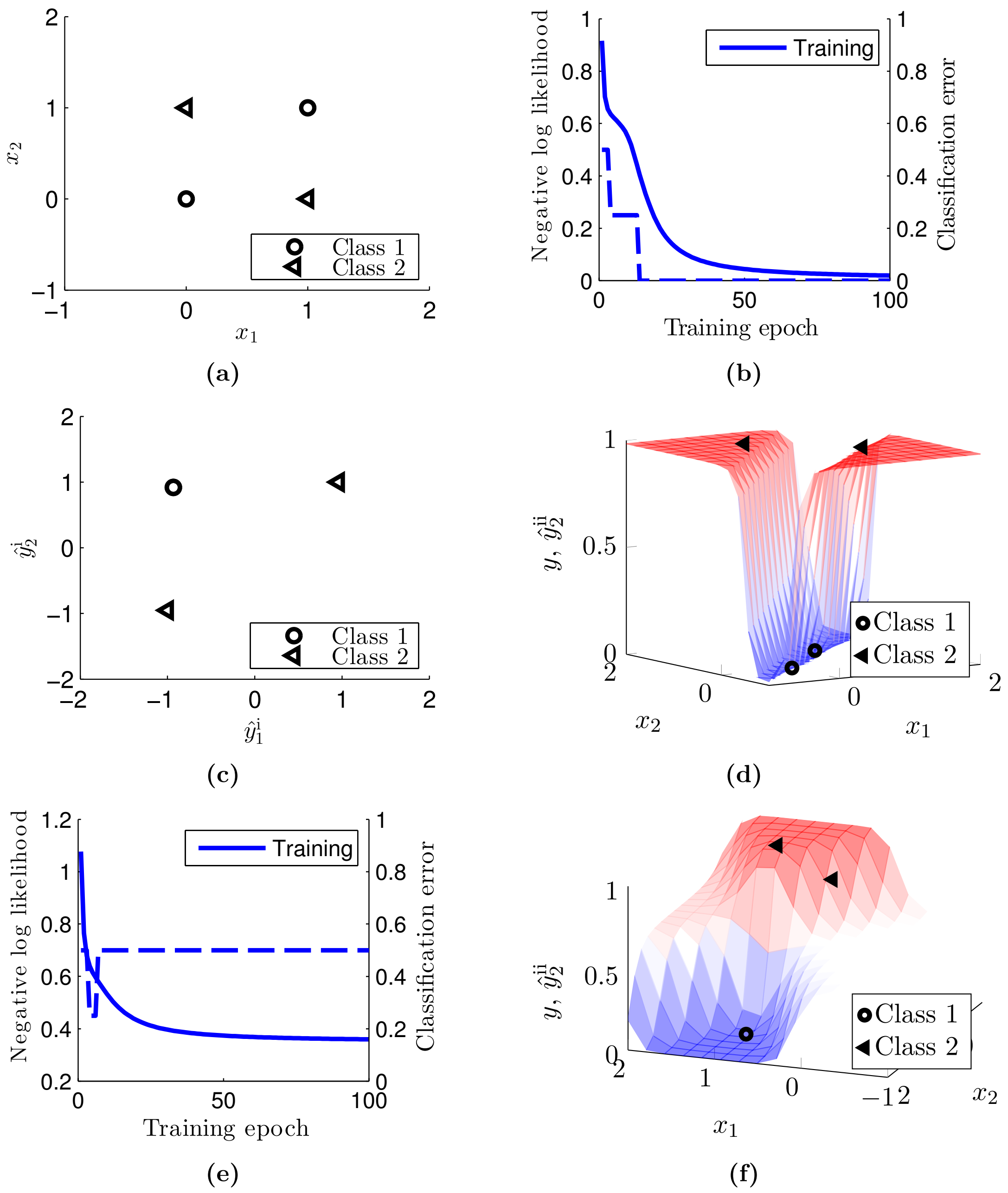


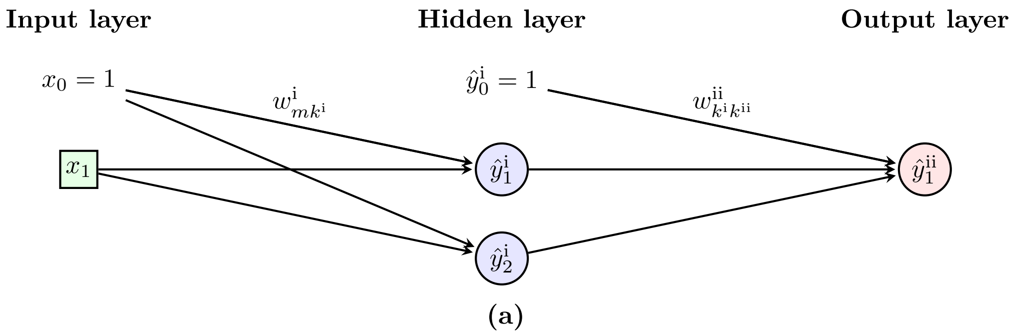
Figure 5.4: a) The XOR problem. b) Training progress for the XOR problem after 100 epochs using Algorithm 3 with 2 hidden neurons, 2 softmax output neurons, random initial weights [-1 1], and $\eta = 1.5$. c) Learned projections by the hidden layer after training (the two data points belonging to class 1 actually lie on top of each other). d) Decision surface for the trained MLP network. e) Training progress for network stuck in a suboptimal minima. f) Decision surface corresponding to a suboptimal minima.

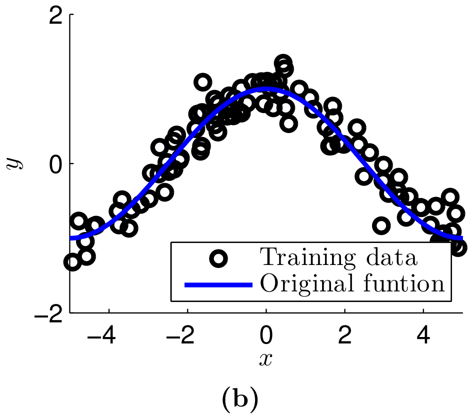
5.3 Non-linear regression

In the previous case, we saw that the hidden layer could learn to perform a non-linear mapping that turned a previously non-linearly separable problem into linearly separable one. An interesting question is therefore if the hidden layer also could learn to perform a mapping that mapped non-linear data onto a hyperplane. We know from chapter 3 that a linear regression output neuron can fit hyperplanes. So, if the hidden layer could map non-linear input data onto a hyperplane, it would be possible for a linear regression output neuron to model this hyperplane. In order to test this, we will fit the simple model in Figure 5.5a to the data points shown in Figure 5.5b. These data points have been generated by the non-linear process:

$$y = \cos\left(x\frac{\pi}{5}\right) + r \tag{5.7}$$

where r is random normally distributed variable with mean zero and standard deviation 0.25. Training the network, using Algorithm 3, for 300 epoch with random initial weights in the interval $[-1\ 1]$ and $\eta = 0.25$ resulted in Figure 5.5c.


Input layer
Hidden layer
Output layer
$x_0 = 1$
$w^{\mathrm{i}}_{mk^{\mathrm{i}}}$
$\hat{y}^{\mathrm{i}}_0 = 1$
$w^{\mathrm{ii}}_{k^{\mathrm{i}}k^{\mathrm{ii}}}$
x_1
$\hat{y}^{\mathrm{i}}_1$
$\hat{y}^{\mathrm{ii}}_1$
$\hat{y}^{\mathrm{i}}_2$
(a)



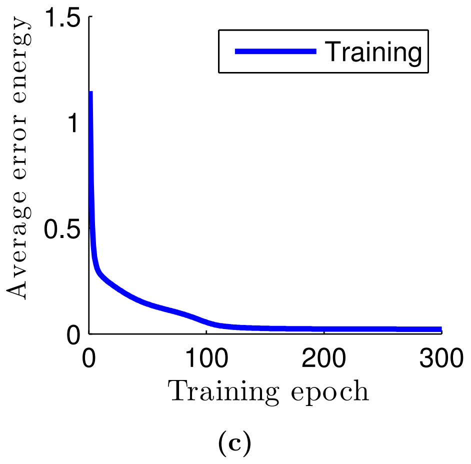
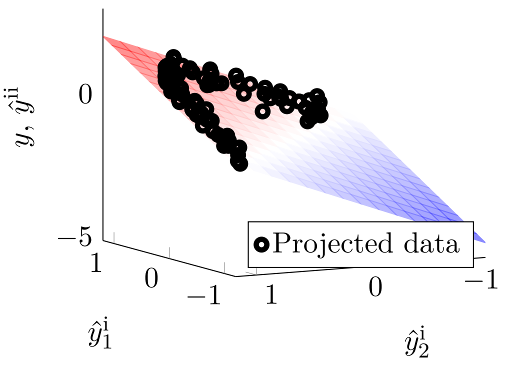
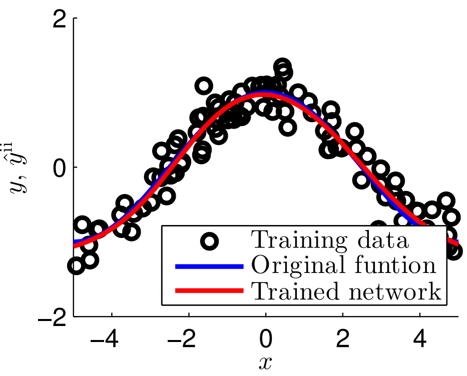


Figure 5.5: a) MLP network used for non-linear regression (Roman numerals are used to indicate depth). b) 100 data points generated by Equation 5.7 together with a blue line representing the same process with its stochastic element excluded. c) Training results for fitting the model in Figure 5.5a to the data in Figure 5.5b, using Algorithm 3 with $\eta = 0.25$ and starting from random initial weights in the interval $[-1\ 1]$.

When we plot the outputs from the hidden layer ($\hat{y}^{\mathrm{i}}$) for each x value together with the desired outputs (y), we notice that the hidden layer has indeed learned to map the inputs onto a hyperplane. This is shown in Figure 5.6a together with the hyperplane learned by the linear regression output neuron. Finally, in Figure 5.6b the learned model for the whole network is illustrated together with the training data.



(a)



(b)

Figure 5.6: a) All data points, in Figure 5.5b, mapped to the space spanned by the neurons in the hidden layer. The hidden neurons in the trained network has here learned to map non-linear input data onto hyperplane that the output neuron in turn has learned to represent. b) The obtained model by the MLP network together with the data used for training.

5.4 MNIST revisited

In chapter 4, we obtained a classification error ratio of just above 0.1 on the MNIST test set using softmax regression. As softmax regression only can separate linearly separable classes, it makes sense to assume that we should be able to obtain a better classification error ratio with an MLP network. In order to test this, we will attempt to fit the model in Figure 5.7 to the MNIST training set. Notice that this model actually contains 1000 hidden neurons.

Starting from random initial weights in the interval $[-10^{-4}\ 10^{-4}]$, Algorithm 3 produces the learning progress in Figure 5.8a with $\eta = 0.5$ and mini batches of size 0.2. After 300 epochs it still looks like both ℓ and $\mathscr{C}$ are decreasing, but at this stage $\mathscr{C}_{\text{test}}$ has already decreased to 0.036 which is far better than what we obtained with softmax regression alone. A complete summary of the results after 300 epoch is given in Table 5.1, and Figure 5.8b illustrates 16 images that were wrongly classified in the test set.

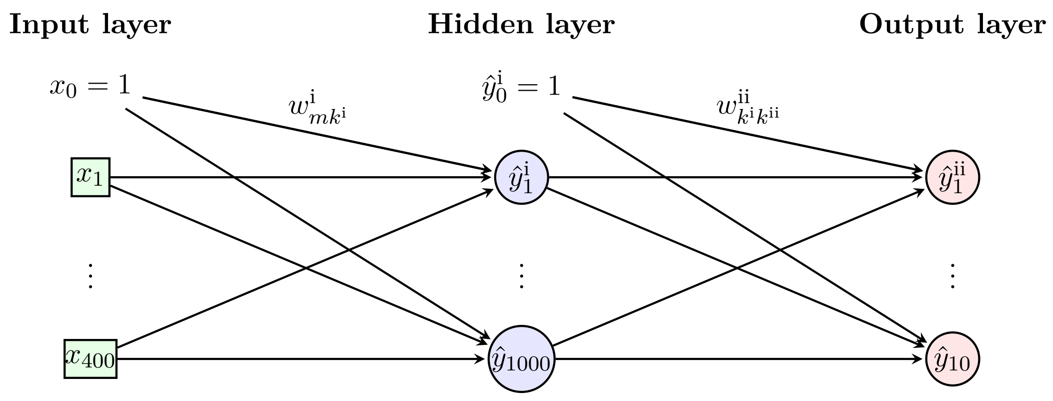
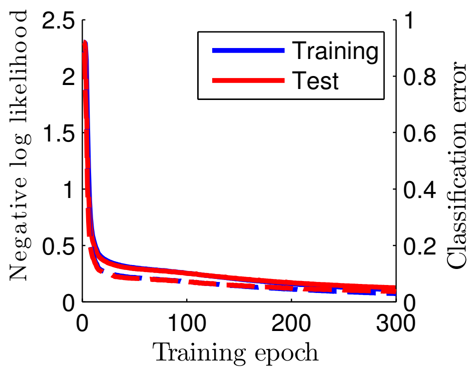
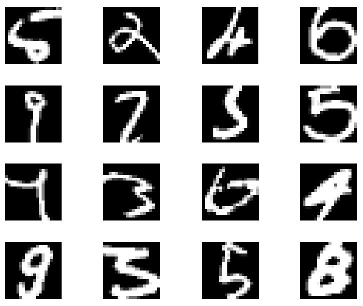


Figure 5.7: MLP network used for non-linear regression (Roman numerals are used to indicate depth).



(a)



(b)

Figure 5.8: a) Fitting the model in Figure 5.7 to the MNIST training set using Algorithm 3 with $\eta = 0.5$, mini batches of size 0.2, and randomly generated initial weights in the interval $[-10^{-4}\ 10^{-4}]$. b) 16 out of 360 wrongly classified images from the test set.

Table 5.1: Numerical summary of the final training results from Figure 5.8a

ℓ_{training}	ℓ_{test}	$\mathscr{C}_{\text{training}}$	$\mathscr{C}_{\text{test}}$
0.1056	0.1243	0.0291	0.0360

Correctly classified training images	58254
Incorrectly classified training images	1746
Correctly classified test images	9640
Incorrectly classified test images	360

5.5 Deeper architectures

So far in this chapter we have seen that the incorporation of a hidden layer gives us non-linear models, but with the additional complexity that both ℓ and $\mathscr{E}_{av}$ now have multiple minima. An interesting question to ask is, therefore, are there any constraint on what MLP networks can model. Hornik, Stinchcombe, and White (1989) have actually answered this question by proofing that a MLP network, with only one hidden layer, can actually approximate any function to any degree of accuracy. This sounds like good news but there is a catch. Even if a MLP network can approximate any function, the proof says nothing about how many hidden neurons are needed, if it is possible to learn such a network using gradient based methods, or most importantly, if such a model is an efficient representation of the function. Many think that deeper networks with more layers can give rise to more efficient representations, and this idea with additional inspiration from the human brain (remember that hierarchical structures are thought to exist in the cortex) has lead people to try to train deeper neural networks. Methods that can successfully train networks with more than two hidden layers exist today and these are generally referred to as “Deep Learning” methods (Bengio, 2009).

6

What is next?

We have now seen how both regression and classification models can be visualized as ANNs, and we have also seen that these models can be trained using gradient descent. As you continue to learn more and more about machine learning, you will notice that this is only the beginning. Several different types of ANNs exist as well as completely different techniques. However, the presented material should have given you basic knowledge to continue your path with any of the following subjects.

Support Vector Machines: In chapter 5, we used a MLP network with depth 2 (hidden and output layer of neurons) and concluded that such a network is capable of solving classification problems that are not linearly separable. We also noticed that this added complexity came with a cost that prevented us from finding optimal weights. Support Vector Machines (SVMs) can also be looked upon as a network of depth 2, but with added constraints that allow us to find an optimal solution. This has made SVMs very popular and a nice introduction is given in Hearst, Dumais, Osman, Platt, and Scholkopf (1998).

Convolutional Neural Networks: The ANNs examined here did not take spatial information into account when classifying images. Contrary to this, neurons in the visual cortex have receptive fields, and this property is copied by Convolutional Neural Networks (CNNs). These are currently state of the art for image classification and LeCun et al. (1998) provide a good introduction.

Hessian free optimization: Gradient descent, which we used throughout this course, is a very crude optimization method. It completely ignores the curvature of the error surface and simply looks a the gradient for determining new weights. Clearly, it would be useful if the curvature also could be taken into account. This way one could estimate how far one should proceed in any direction along the surface. Such information, however, comes at a cost, and might not always computationally feasible to obtain. Recently though, Martens (2010) proposed an efficient Hessian free optimization method for deep neural networks that do take the curvature into account.

References

Arel, I., Rose, D. C., & Karnowski, T. P. (2010). Deep machine learning-a new frontier in artificial intelligence research [research frontier]. *Computational Intelligence Magazine, IEEE*, *5*(4), 13–18.

Benenson, R. (2013, December 18). What is the class of this image? [GitHub]. Retrieved December 21, 2013, from http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

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

Bengio, Y., Courville, A., & Vincent, P. (2012). Representation learning: a review and new perspectives. arXiv: 1206.5538 `[cs.LG]`

Bishop, C. M. (1995). *Neural networks for pattern recognition.* Oxford university press.

Cleary, T. (1988). *The art of war.* Shambhala.

Croft, A., Davison, R., & Hargreaves, M. (2001). *Engineering mathematics.* Pearson Education.

Friston, K. (2010). The free-energy principle: a unified brain theory? *Nature Reviews Neuroscience*, *11*(2), 127–138.

Hawkins, J. (2004). *On intelligence.* Macmillan.

Haykin, S. S. (2009). *Neural networks and learning machines.* Prentice Hall New York.

Hearst, M., Dumais, S., Osman, E., Platt, J., & Scholkopf, B. (1998, July). Support vector machines. *Intelligent Systems and their Applications, IEEE*, *13*(4), 18–28.

Herculano-Houzel, S. (2012). The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *Proceedings of the National Academy of Sciences*, *109*(Supplement 1), 10661–10668.

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

Laserson, J. (2011). From neural networks to deep learning: zeroing in on the human brain. *XRDS: Crossroads, The ACM Magazine for Students*, *18*(1), 29–34.

Lashley, K. S. (1950). In search of the engram. In *Symposia of the society for experimental biology* (Vol. 4, *454-482*, p. 30).

Lay, D. C. (2012). *Linear algebra and its applications*. Pearson.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the ieee* (Vol. 86, *11*, pp. 2278–2324).

Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th international conference on machine learning (icml)*.

Mcmilan, R. (2013, February 18). How google retooled android with help from your brain. *Wired*. Retrieved December 21, 2013, from http://www.wired.com/wiredenterprise/2013/02/android-neural-network/

Metz, C. (2013, December 12). Facebook's 'deep learning' guru reveals the future of ai. *Wired*. Retrieved December 21, 2013, from http://www.wired.com/wiredenterprise/2013/12/facebook-yann-lecun-qa/

Mitchell, T. (1997). *Machine learning*. McGraw-Hill.

Poggio, T. & Ullman, S. (2013). Vision: are models of object recognition catching up with the brain? *Annals of the New York Academy of Sciences*.

Squire, L. R. & Kandel, E. R. (2009). *Memory from mind to molecules*. Roberts and Company Publishers.

Von Melchner, L., Pallas, S. L., & Sur, M. (2000). Visual behaviour mediated by retinal projections directed to the auditory pathway. *Nature*, *404*(6780), 871–876.

Wikipedia. (n.d.-a). Automobile. Retrieved December 22, 2013, from http://en.wikipedia.org/wiki/Automobile

Wikipedia. (n.d.-b). Neuron. Retrieved December 20, 2013, from http://en.wikipedia.org/wiki/Neuron

A Matlab code for solving the XOR problem

```
clear all; close all; clc;

% Paramters
eta = 0.25;
nEpochs = 200;

% Generating data
X = [1 1 1 1; 1 0 1 0; 1 0 0 1];
Y = [1 1 0 0; 0 0 1 1];

% Number of data points
N = size(X,2);

% Generating random initial weights for 5 hidden neurons
Wi = randn(3,5);
Wii = randn(6,2);

% Defining l and C vectors
l = nan(1,nEpochs);
C = nan(1,nEpochs);

% Initializing progress figure
f1 = figure();
[ah,ph1,ph2] = plotyy(0:(nEpochs-1), l, 0:(nEpochs-1), C);
set(ph1, 'LineWidth', 2); set(ph2, 'LineWidth', 2);
set(ah(2), 'YLim', [0 1]);
set(ah(1), 'XLim', [0 nEpochs]); set(ah(2), 'XLim', [0 nEpochs]);

% Gradient descent
for i = 1:nEpochs
  % Calculating the locally induced field in the hidden layer
  % Please note that Wi is transposed!
  Vi = Wi'*X;
  % Calculating outputs from the hidden layer
  YiHat = tanh(Vi);
  % Adding a row of ones to yiHat
  YiHat = [ones(1,size(YiHat,2)); YiHat];
  % Calculating induced fields in the output layer
  % Please note that Wii is transposed!
  Vii = Wii'*YiHat;
  % phi(Vii) is determined by the output layer type
  YiiHat = exp(Vii) ./ ( ones(size(Vii,1),1) * sum(exp(Vii)) );

  % l and C
  l(i) = -1/N * sum(sum( Y.*log(YiiHat) ));
  % Labels from Y
  [~, Y_labels] = max(Y, [], 1);
  % Labels from Yhat
  [~, Yhat_labels] = max(YiiHat, [], 1);
  % The classification error ratio
  C(i) = mean( Y_labels ~= Yhat_labels );

  % Updating progress figure
  set(ph1, 'YData', l); set(ph2, 'YData', C);
  drawnow

  % Error signal
  E = Y - YiiHat;
  % Weight change for layer (ii)
  dWii = -eta * 1/N * YiHat*E';
  % Weight change for layer (i)
  dWi = -eta * 1/N * X*( (Wii(2:end,:)*E) .* sech(Vi).^2 )';

  % Updating weights
  Wi = Wi - dWi;
  Wii = Wii - dWii;
end

% Fixing labels after training is complete (faster)
xlabel('Epochs', 'Interpreter', 'LaTex');
ylabel(ah(1), 'Mean negative log likelihood ($l$)',...
    'Interpreter', 'LaTex');
ylabel(ah(2), 'Classification error ratio ($C$)',...
    'Interpreter','LaTex');
leg = legend('$l$', '$C$'); set(leg, 'Interpreter', 'latex');

% Calculating a decision surface for output neuron 1
xLim = [-1 2];
dx = 0.1;
[X1, X2] = meshgrid( xLim(1):dx:xLim(2), xLim(1):dx:xLim(2) );

for i = 1:size(X1,1)
  for j = 1:size(X1,2)
    % Creating a datapoint
    X_tmp = [1; X1(i,j); X2(i,j)];
    % Calculating the locally induced field in the hidden layer
    % Please note that Wi is transposed!
    Vi = Wi'*X_tmp;
    % Calculating outputs from the hidden layer
    YiHat = tanh(Vi);
    % Adding a row of ones to yiHat
    YiHat = [ones(1,size(YiHat,2)); YiHat];
    % Calculating induced fields in the output layer
    % Please note that Wii is transposed!
    Vii = Wii'*YiHat;
    % phi(Vii) is determined by the output layer type
    YiiHat = exp(Vii) ./ ( ones(size(Vii,1),1) * sum(exp(Vii)) );
    % Storing the obtained output
        YiiHat1(i,j) = YiiHat(1);
  end
end

% Plotting a decision surface and data points
f2 = figure();
hold on
% Class 1
```

```
c1 = plot3(X(2,1:2), X(3,1:2), Y(1,1:2), 'ko',...
    'MarkerSize', 10, 'LineWidth', 2);
% Class 2
c2 = plot3(X(2,3:4), X(3,3:4), Y(1,3:4), 'kx',...
    'MarkerSize', 10, 'LineWidth', 2);
% Decision surface
sh = surf(X1, X2, YiiHat1, 'EdgeColor', 'none');
% Transparency and colormap
alpha(0.5)
colormap(spring);
% Legend and labels
leg = legend('Class 1', 'Class 2', 'Decision surface');
set(leg, 'Interpreter', 'latex');
xlabel('$x_1$', 'Interpreter', 'LaTex');
ylabel('$x_2$', 'Interpreter', 'LaTex');
zlabel('$y$, $\hat{y}_1^{ii}$', 'Interpreter', 'LaTex');
```

B Suggested course structure

What here follows is a suggestion for how the course could be structured.

Lectures and assignments

A big portion of the course is reserved for implementations in MATLAB; lecture time is therefore divided 50/50 to presentation of the material and programming assistance. However, assigned programming time during lectures will not be enough to complete the homework assignments. Students are therefore also expected to work on their own.

Requirements for passing the course

In order to pass the course every student will have to complete 3 homework assignments and give a short presentation (no exam). Each homework assignment will be worth 10 points in total, and atleast 3.5 points is needed from each one in order to pass the course. 10.5 points therefore gives the grade 1, and from here the scale goes linearly upwards to the grade 5. Homework problems can be discussed freely with other student, but every one have to hand in a report of their own. The report should always present answers to solved questions together with the code used to solve the problem (if the answer required programming). If there are cases with clear plagiarism, all students involved will have to redo their report and the best attainable grade for the course will be lowered to 3. Besides the homework problems, everyone will also have to give a short presentation (10 min) in a group with one or two other students. More information about the homework assignments and the presentation is found in accompanying appendices.

B.1 Guidelines for reporting answers to homework problems

There is no need to write an essay in order to present your answers, but you should present your answers in a well structured manner. That is, each question should be answered separately, and the answer should include comments on what you have done together with the results you obtained. Any MATLAB code used to reach the results should either be appended after the answer or at the end of your report (choose the method that makes everything as clear as possible). The code may **not** include any of your answers as comments, all obtained results that you want to present should be presented in your answer to the question. The person reading your answers should not have to go trough your m-code in detail in order to see what you have done. The m-code is simply appended so that there is a possibility to check what kind of errors you have done if your answers are looking strange.

Figures make up a big part of your answers and it is therefore important that they are well presented. You should therefore make sure that each figure fulfil the following requirements:

- There are labels on all axes.
- Axes limits are chosen in such a way that what you want to show is clearly visible.
- Use legends if necessary to clear out any ambiguities.
- Each figure or table should always be accompanied with a descriptive text.

As a rule of thumb, each figure description should include all the information needed for another person to redo the experiment and validate your results. Finally, always hand in your answers as **one** pdf file, and name the file **first name_surname_HomeworkX.pdf** where X is the number of the homework assignment (1, 2, or 3).

B.2 Homework 1

The purpose of this homework is to solve a linear regression problem in order to get familiar with gradient descent. You are free to discuss the homework tasks with the rest of the class, but you have to hand in a report with answers (code appended as well) individually. Please make sure that the report is well structured and follows the guidelines in Appendix B.1. Points will be subtracted for poorly structured reports and ambiguous answers.

Generate data (1 p)

Generate and plot 100 data points, where x is drawn randomly in the interval [-5 5], from the process:

$$y = a + bx + r \tag{B-1}$$

where r is a normally distributed random number with mean 0 and variance 0.5, a is the month you where born divided by ten, and b the day of the month you were born divided by ten. Complement your plot with a straight line given by Equation B-1 when r is ignored.

Surface plot of $\mathscr{E}_{av}$ (2 p)

Assuming that you will try to fit a model as the one in Figure 3.1, plot a surface plot of $\mathscr{E}_{av}$ as function of both w_0 and w_1 in a suitable interval so that the minima can be seen.

Gradient descent (3 p)

Implement Algorithm 1 and find the minima of $\mathscr{E}_{av}$. Plot the algorithms progress on top of the surface plot obtained from the previous task. Verify that the algorithms seem to be moving in the direction of steepest descent.

Varying η (1 p)

When varying η, you should see that for some values the algorithms diverges, for other values it moves in a zigzag pattern towards the minima, and yet for other values it moves in a smooth trajectory towards the minima. Examine the limits for η where these three different regimes occur and plot an example from each.

Linear algebra (1 p)

Linear algebra provides the following method to determine the optimal values for the weights.

$$\mathbf{w} = (\mathbf{X}\mathbf{X}^{\intercal})^{-1}\mathbf{X}\mathbf{y}^{\intercal} \tag{B-2}$$

Why does the above calculation give us the right answer? You probably have to google up where this equations comes from, or find a book on linear algebra. The above definition is adapted to how matrices are defined here, so the equation you will find in books or online most likely looks something like:

$$\hat{\mathbf{x}} = (\mathbf{A}^\intercal\mathbf{A})^{-1}\mathbf{A}^\intercal\mathbf{y} \tag{B-3}$$

Preprocessing (2 p)

Data is often preprocessed before any attempts to fit a model are done. A normal preprocessing stage consists of mean centering followed up by variance scaling. In our case, this means that when you calculate the means for the rows in **X** you should get a vector of zeros as your answer. Similarly, when calculating the variance for the rows you should get a vector of ones. However, this procedure does not have to be done for x_0, so here you just have to do it for x_1.

Incorporate preprocessing into your previous solutions and do another surface plot of $\mathscr{E}_{av}$. What has now changed and how does this change affect how the gradient descent algorithm moves towards the minima?

B.3 Presentation

This presentation will strive to introduce you to people whose names are encountered quite often when reading material on both artificial and real neural networks. In groups of 2 to 3 persons, you are to select one of the persons in the list below and give a short presentation (10 min) about what this persons has done.

1. **Donald Hebb**
2. **Geoffrey Hinton**
3. **Andrew Ng**
4. **Yann LeCun**
5. **Tomaso Poggio**
6. **Eric Kandel**
7. **Henry Markram**
8. **Yoshua Bengio**
9. **Jeff Hawkins**
10. **Ramón Cajal**

B.4 Homework 2

In this homework you will implement softmax regression to solve different classification problems. As before, you are free to discuss your solutions with other students, but everyone have to hand in an individual report containing their answers. Please make sure that the report is well structured and follows the guidelines in Appendix B.1. Points will be subtracted for poorly structured reports and ambiguous answers.

Basic math

a Explain how matrix multiplications work, and illustrate your understanding by calculating an example of your own without a computer. **(0.25 p)**

b Explain and illustrate why matrix multiplications are useful for determining the induced field for the artificial neuron in Figure 2.3. **(0.25 p)** c Why are we not affecting the localization of the a minima when we take the logarithm of the likelihood function ($\mathscr{L}$)? **(0.25 p)**

d In what way are your models restricted if you forget to add a row of ones to $\mathbf{X}$? **(0.25 p)**

A two dimensional two class example

a Generate two dimensional data with 100 data points for two classes (50 data points per class) so that the classes are linearly separable. Besides from the previously mentioned constraint, you are free to come up with your own function for generating the data, just remember that $\mathbf{Y}$ now is two dimensional (one output for each class). Finally, plot the data you generated. **(1 p)**

b Implement Algorithm 2 and make sure that you have calculated the partial derivatives correctly by also implementing gradient checking. **(1 p)**

c Run Algorithm 2 to fit your model to the data and plot ℓ and $\mathscr{C}$ for each epoch. **(0.5 p)**

d Visualize the obtained model by plotting a decision surface together with the data you used for training. **(1 p)**

e Use your knowledge from the previous homework to also fit a model using linear regression to your data. Compare the obtained decision surfaces for these two models. **(0.75 p)**

f Adjust your function for generating the two dimensional data so that the classes no longer are linearly separable. Train an additional model for this new data and see what kind of decision surface you obtain. **(0.75 p)**

Handwritten digits

a Download the MNIST dataset and the corresponding function for reading the data files into MATLAB (see section 4.5 for details).

b Use the downloaded function to store the MNIST datasets into the matrices $\mathbf{X}_{\text{training}}$, $\mathbf{X}_{\text{test}}$, $\mathbf{Y}_{\text{training}}$, and $\mathbf{Y}_{\text{test}}$. The original data only contains a label for y so you have to generate both $\mathbf{Y}$ matrices your self. Show that this step is accomplished by plotting the first 16 images from the training set. **(1 p)**

c Fit a softmax regression model to the MNIST data, illustrate how the training progresses, and present how well your best found model performs. Use your implementation of gradient checking here as well to convince your self that your code is correct. **(2.5 p)** d Visualize 16 wrongly classified images. **(0.5 p)**

B.5 Homework 3

In this homework you will investigate how multilayer perceptron networks perform a non-linear projection of your data, followed up by either linear or softmax regression inside a new space spanned by neurons in the hidden layer. As before, you are free to discuss your solutions with other students, but everyone have to hand in an individual report containing their answers. Please make sure that the report is well structured and follows the guidelines in Appendix B.1. Points will be subtracted for poorly structured reports and ambiguous answers.

The XOR problem (3 p)

You are to replicate the experiment presented in section 5.2. Points are handed out for the following steps:

a Plot your data points in the space spanned by your inputs (x_1 and x_2), and in the space spanned by the two neurons in your hidden layer ($\hat{y}^{\text{i}}_1$ and $\hat{y}^{\text{i}}_2$) for random initial weights. **(0.5 p)**

b Write a function that returns the gradient and the mean negative log likelihood, and verify that the gradient is calculated correctly using gradient checking. **(0.5 p)**

c Train the network using gradient descent and plot ℓ and $\mathscr{C}$ for each epoch. **(0.5 p)**

d Plot a decision surface for one of your output neurons and see how the model generalises. **(0.5 p)**

e Show that your model has learnt to do a non-linear mapping from (x_1, x_2) to ($\hat{y}^{\text{i}}_1$, $\hat{y}^{\text{i}}_2$) so that the classes now are linearly separable. **(0.5 p)**

f Increase the number of hidden neurons and see if you can get a different decision surface. **(0.5 p)**

Non-linear regression (2 p)

You are to replicate the experiment presented in section 5.3. Points are handed out for the following steps:

a Plot your original data points (x, y) and the non-linear projection ($\hat{y}^{\text{i}}_1$, $\hat{y}^{\text{i}}_2$, y) for random initial weights. **(0.5 p)**

b Train the network using gradient descent and plot $\mathscr{E}$ for each epoch. **(0.5 p)** c Plot your data points again after the non-linear projection ($\hat{y}_1^{\text{i}}$, $\hat{y}_2^{\text{i}}$, y), and add the plane learnt by your linear regression neuron in the output layer. **(0.75 p)**

d Visualize the model learned together with your data points, (x, y) and (x, $\hat{y}^{\text{ii}}$). **(0.25 p)**

Is this an eye? (5 p)

Your task is to tell if an image represents an eye or not (positive = image centred on the eye, negative = eye not in centre or not at all in the image). To this end, you can implement any method of your choosing (presented in this course), but you will get points for how well you succeed. That is, points are awarded for presenting the data, presenting your model, verifying that your model works, and for presenting how well it works.

You will receive a mat file containing the data. This file contains two matrices **X** (15x15x2255) and **Y** (2x2255). Hence, each image is of size 15 by 15 pixels and there are 2255 images in total. Divide the images into training and testing sets in order to get an unbiased estimate of how well your model works. Good luck!

Notation

$\mathbf{X}$	Matrix with input data (dimensions: $M+1$ by N)
$\mathbf{x}$	One column from $\mathbf{X}$ (represents a data point as a vector)
x_{mn}	One element in $\mathbf{X}$
M	The dimensionality of $\mathbf{x}$
m	Index for input dimensionality, going from 0 to M
N	The number of data points in $\mathbf{X}$
n	Index for data points, going from 1 to N
$\mathbf{V}$	Matrix with induced fields (dimensions: K by N)
$\mathbf{v}$	One column from $\mathbf{V}$ (induced fields by $\mathbf{x}$)
v_{kn}	One element in $\mathbf{V}$
$\mathbf{Y}$	Matrix with desired output data (dimensions: K by N)
$\mathbf{y}$	One column from $\mathbf{Y}$ (desired output vector for $\mathbf{x}$)
y_{kn}	One element in $\mathbf{Y}$
$\hat{\mathbf{Y}}$	Matrix with model output data (dimensions: K by N
$\hat{\mathbf{y}}$	One column from $\hat{\mathbf{Y}}$ (obtained model output vector for $\mathbf{x}$)
$\hat{y}_{kn}$	One element in $\hat{\mathbf{Y}}$
$\mathbf{E}$	Error signals as defined by Equation 3.2 (dimensions: K by N)
$\mathbf{e}$	One column from $\mathbf{E}$ (represents the observed error signal vector for $\mathbf{x}$)
e_{kn}	One element in $\mathbf{E}$
K	The dimensionality of $\mathbf{y}$, $\hat{\mathbf{y}}$, $\mathbf{v}$, and $\mathbf{e}$
k	Index for output dimensionality, going from 1 to K
$\mathbf{W}$	Weight matrix (dimensions: # of layer inputs by # of layer outputs)
$\mathbf{w}$	One column from $\mathbf{W}$ (represents weights to a single neuron)
w_{mk}	One element in $\mathbf{W}$
$\Delta\mathbf{W}$	Matrix with weight changes, same dimensions as $\mathbf{W}$
$\Delta\mathbf{w}$	One column from $\Delta\mathbf{W}$ (represents weights changes for a single neuron)
Δw_{mk}	One element in $\Delta\mathbf{W}$
L	Network depth
l	Index for depth, going from 1 to L
φ	Activation function
ε	Error term
$\mathscr{E}$	Error energy
$\mathscr{E}_{av}$	Average error energy
η	Learning rate parameter
$\mathscr{L}$	Likelihood function
ℓ	Mean negative log likelihood
$\mathscr{C}$	Classification error ratio

Acronyms

AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
MATLAB	Matrix Laboratory
MLP	Multilayer Perceptron
SVM	Support Vector Machine

Index

NOVIA
UNIVERSITY OF APPLIED SCIENCES

NOVIA UNIVERSITY OF APPLIED SCIENCES
Tel +358 (0)6 328 5000
Fax +358 (0)6 328 5110
www.novia.fi

ADMISSIONS OFFICE
PO BOX 6
FI-65201 Vaasa, Finland
Tel +358 (0)6 328 5055
Fax +358 (0)6 328 5117
admissions@novia.fi

ISBN 978-952-5839-86-9
9 789525 839869