FYS-STK4155 - Applied data analysis and machine learning

Project 3

Even M. Nordhagen

November 29, 2018

- Github repository containing programs and results:

  `https://github.com/evenmn/FYS-STK4155`

Abstract

The aim of this project is to divide various sounds into ten categories, inspired by the Urban Sound Challenge. For that we have investigated the performance of logistic regression and various neural networks, like Feed-forward Neural Networks (FNNs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

When it comes to error estimation, accuracy score is a natural choice. Various activation functions were investigated, and regularization was added for the logistic case. Since we mostly used ADAM as optimization tool...

To extract features from our dataset (dimensionality reduction), we used a spectrogram in the CNN case, and Mel Frequency Ceptral Coefficients (MFCCs) for FNN and RNN. We also tried two RNN networks, mainly the Long Short-Term Memory (LSTM) network and the Gated Recurrent Unit (GRU) network.

All the neural networks were more or less able to recognize the training set, but it was a significantly difference in test accuracy. FNN provided the highest test accuracy of 94%, followed by LSTM (90%), GRU (88%) and CNN (85%). Our linear model, logistic regression, was only able to classify 59% of the test set correctly.

Contents

1 Introduction **3**

2 Sound analysis **3**
2.1 About the Urban Sound Challenge (USC) 3
2.2 Time domain 4
2.3 Frequency domain 5
2.4 Extract features 5
2.4.1 Spectrograms 5
2.4.2 Mel Frequency Cepstral Coefficients (MFCCs) 5
2.5 Examples from USC 6

3 Classification methods **6**
3.1 Logistic regression 7
3.2 Feed-forward Neural Networks (FNN) 11
3.3 Convolutional Neural Networks (CNN) 12
3.4 Recurrent Neural Networks (RNN) 14
3.4.1 Long Short-Term Memory (LSTM) 14
3.4.2 Gated Recurrent Unit (GRU) 14

4 Optimization **14**
4.1 Gradient Descent (GD) 14
4.2 Stochastic Gradient Descent (SGD) 15
4.3 ADAM 15

5 Activation **16**
5.1 Logistic 16
5.2 ReLU 16
5.3 ELU 16

6 Code **16**
6.1 Code structure 17
6.2 Implementation 17

7 Results **17**
7.1 Logistic regression 17
7.2 Feed-forward Neural Networks 17
7.3 Convolutional Neural Networks 18
7.4 Recurrent Neural Networks 18

8 Discussion **18**

9 Conclusion **18**

10 References **19**

1 Introduction

In the everyday life we are continuously surrounded by sounds, and usually the human brain is able to recognize what kind of sound it is based on experience. Artificial neural networks are again based on studies of the human brain, and if the artificial neurons work as the biological ones, there should be possible training a neural network recognizing sounds as well.

Lately, immense efforts have been put into this subject with the purpose of translating voice into text, leaded by technology companies like Google, Microsoft, Amazon and so on, who develop voice controlled virtual assistants. The technology is promising, the time saving using voice commands vs. keyboard commands is potentially huge and the market is enormous. Some even claim that virtual assistants will run our lives within 20 years. [`https://nordic.businessinsider.com/ai-assistants-will-run-our-lives-20-years-from-now-2018-1?r=US&IR=T`] If that is right is still uncertain, but what is certain is that the technology has great potential.

In this final project we will, based on the Urban Sound Challenge, sort sounds into classes using various classification methods. We use the same idea as when the great technology companies recognize voice, but we are going to differentiate between sounds made by **air conditioners**, **car horns**, **children**, **dogs**, **drills**, **engines**, **guns**, **jackhammers**, **sirens** and **street musicians**. In order to do that,

2 Sound analysis

In sound analysis, one can either analyze the raw sound files directly or convert them to simpler files and hope we have not lost any essential features. A big advantage with the latter, is that the dimensionality and complexity of the data set is significantly reduced, but we will also loose some information.

2.1 About the Urban Sound Challenge (USC)

The Urban Sound Challenge is a classification contest provided by Analytics Vidhya with the purpose of introducing curious people to a real-world classification problem. After registered, one is provided with a dataset containing sounds from ten classes. For the training data set, the classes (targets) are given, but there is also a test dataset where the targets are unknown. Our task is to classify the test dataset correctly, and by uploading our results to Analytics Vidhya's webpage, they will return the accuracy score. Participants are expected to summit their answers by 31st of December 2018, and there will be a leaderboard. We are allowed to use all possible tools, including open-source libraries like TensorFlow, Keras and Scikit-Learn. For more practical information, see [`https://datahack.analyticsvidhya.com/contest/practice-problem-urban-sound-classification/`]

The data sets consists of a total number of 8732 sound samples (5434 training sets and 3298 test sets) with a constant sampling rate of 22050Hz. The length of each sampling is maximum 4 seconds, and they are distributed between ten classes, namely

- air conditioner
- car horn
- children playing
- dog bark
- drilling
- engine idling
- gun shot
- jackhammer
- siren
- street music

The error is evaluated with the accuracy score, which is just how much of the data set that is classified correctly:

$$\text{Accuracy} = \frac{\sum_{i=1}^{N} I(y_i = t_i)}{N}. \tag{1}$$

2.2 Time domain

Sounds are longitudinal waves which actually are different pressures in the air. They are usually represented as a function in the time domain, which means how the pressure vary per time. This function is obviously continuous, but since computers represent functions as arrays, we cannot save all the information.

INSERT FIGURE

How much information we loose depends on the sampling rate, which is the number of sampling points per second. A rule of thumb is that one should have twice as high sampling rate as the highest sound frequency to keep the most important information. For instance, a human ear can perceive frequencies in the range 20-20000Hz, so around a sampling rate around 40kHz should be sufficient to keep all the information. Ordinary CD's use a sampling rate of 44.1kHz, but for speech recognition, 16kHz is enough to cover the frequency range of human speech. [https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a]

2.3 Frequency domain

Sometimes, a frequency domain gives a better picture than the time domain. On one hand, one looses the time dependency, but on the other one gets information about which frequencies that are found in the wave. One goes from the time domain to the frequency domain using Fourier transformations. Which one that gives the best results in our case is not clear.

INSERT FIGURE

2.4 Extract features

There are many techniques

2.4.1 Spectrograms

As we have seen above, we are missing the frequency information in the time domain, and time information in the frequency domain. Is it possible to get the best of both worlds? This is what we try when creating a spectrogram, which is a time-frequency representation.

The procedure goes like this: we divide the spectrum into N frames and Fourier transform each frame. Thereafter we collect all the frequency domains in a NxM-matrix, where M is the length of each frequency domain. We are then left with an image which hopefully hold the important information, and since Hidden Markov Models are known to implicity model spectrograms for speech recognition, we have a reason to believe it will work. [https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=11&ved=2ahUKEwi3__LqlPfeAhUEDCwKHQStD_gQFjAKegQICBAC&url=http%3A%2F%2Fwww.speech.cs.cmu.edu%2F15-492%2Fslides%2F03_mfcc.pdf&usg=AOvVaw1WHnRzY9KrtWTnsmR9_SBf]

2.4.2 Mel Frequency Cepstral Coefficients (MFCCs)

MFCCs are features widely used in automatic speech and speaker recognition. [http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/] They are inspired by how the human ear extracts features, and has been the main feature extraction tool for sound recognition since it was introduced in the 1980's.

It actually takes advantage of the spectrogram, and use it to reveal the most dominant frequencies. Thereafter, we sum the energy in each frame and take the logarithm, which is the way an ear works. Those energies are then what we call the Mel Frequency Ceptral Coefficients. Ulike the spectrogram, we are now left with a single array of information from the spectrum.

2.5 Examples from USC

ADD SOME EXAMPLES

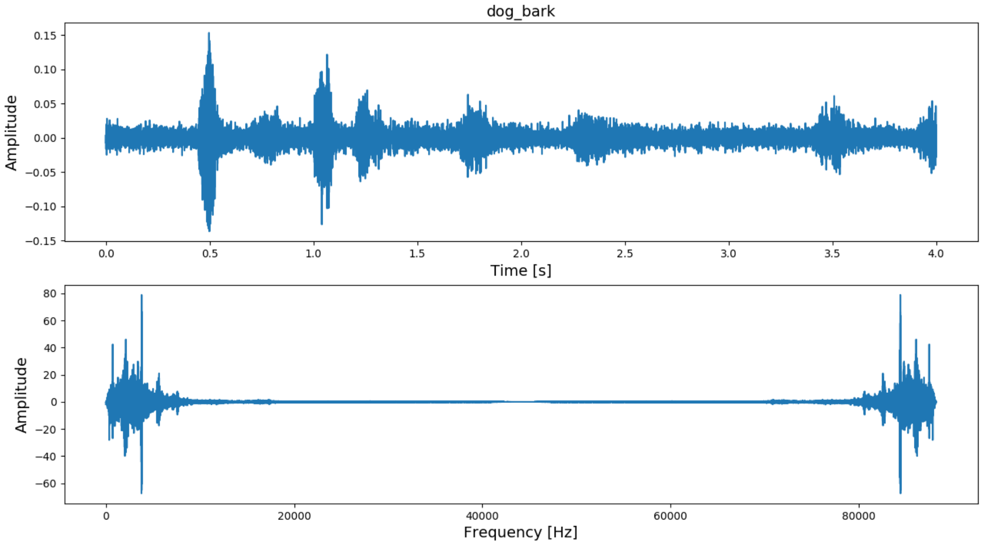

Figure 1

3 Classification methods

Classification is very important in everyday life, and we often use classification even without thing about it. A simple example is when we distinguish between people, which is usually an easy task for a human. For a computer, on the other hand, classification is difficult, but fortunately some great methods are developed mainly based on neural networks. In this contest we will investigate several methods, spanning from logistic regression to various neural networks like **Feed-forward Neural Networks**, **Convolutional Neural Networks** and **Recurrent Neural Networks**.

3.1 Logistic regression

Logistic regression is a linear model, where...

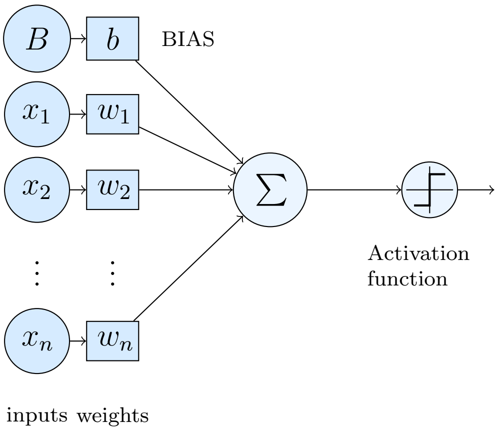

Figure 2: Logistic regression model with n inputs.

In logistic regression, we usually have one binary output node for each class, but for two categories one output node is sufficient, which can be fired or not fired.

Initially, one needs to train the perceptron such that it knows which outputs are correct, and for that one needs to know the outputs that correspond to the inputs. Every time the network is trained, the weights are adjusted such that the error is minimized.

The very first step is to calculate the initial outputs (forward phase), where the weights usually are set to small random numbers. Then the error is calculated, and the weights are updated to minimize the error (backward phase). So far so good.

Forward phase

Let us look at it from a mathematical perspective, and calculate the net output. The net output seen from an output node is simply the sum of all the "arrows" that point towards the node, see figure (2), where each "arrow" is given by the left-hand node multiplied with its respective weight. For example, the contribution from input node 2 to the output node follows from $X_2 \cdot w_2$, and the total net output to the output O is therefore

$$\boxed{net = \sum_{i=1}^{I} x_i \cdot w_i + b \cdot 1.} \tag{2}$$

Just some notation remarks: x_i is the value of input node i and w_i is the weight which connects input i to the output. b is the bias weight, which we will discuss later.

You might wonder why we talk about the net output all the time, do we have other outputs? If we look at the network mathematically, what we talk about as the net output should be our only output. Anyway, it turns out to be convenient mapping the net output to a final output using an activation function, which is explained further in section 3.1. The activation function, f, takes in the net output and gives the output,

$$out = f(net). \tag{3}$$

If not everything is clear right now, it is fine. We will discuss the most important concepts before we dive into the maths.

BIAS

As mentioned above, we use biases when calculating the outputs. The nodes, with value B, are called the bias nodes, and the weights, b, are called the bias weights. But why do we need those?

Suppose we have two inputs of value zero, and one output which should not be zero (this could for instance be a NOR gate). Without the bias, we will not be able to get any other output than zero, and in fact the network would struggle to find the right weights even if the output had been zero.

The bias value B does not really matter since the network will adjust the bias weights with respect to it, and is usually set to 1 and ignored in the calculations. [2]

Learning rate

In principle, the weights could be updated without adding any learning rate ($\eta = 1$), but this can in practice be problematic. It is easy to imagine that the outputs can be fluctuating around the targets without decreasing the error, which is not ideal, and a learning rate can be added to avoid this. The downside is that with a low learning rate the network needs more training to obtain the correct results (and it takes more time), so we need to find a balance.

Cost function

The cost function is what defines the error, and in logistic regression the cross-entropy function is a naturally choice. [3] It reads

$$\boxed{c(\boldsymbol{W}) = -\sum_{i=1}^{n}\Big[y_i \log f(\boldsymbol{x}_i^T\boldsymbol{W}) + (1-y_i)\log[1-f(\boldsymbol{x}_i^T\boldsymbol{W})]\Big]} \tag{4}$$

where $\boldsymbol{W}$ contains all weights, included the bias weight ($\boldsymbol{W} \equiv [b, \boldsymbol{W}]$), and similarly does $\boldsymbol{x}$ include the bias node, which is 1; $\boldsymbol{x} \equiv [1, \boldsymbol{x}]$. Further, the $f(x)$ is the activation function discussed in next section.

The cross-entropy function is derived from likelyhood function, which famously reads

$$p(y|x) = \hat{y}^y \cdot (1-\hat{y})^{1-y}. \tag{5}$$

Working in the log space, we can define a log likelyhood function

$$\begin{aligned} \log\Big[p(y|x)\Big] &= \log\Big[\hat{y}^y \cdot (1-\hat{y})^{1-y}\Big] && (6)\\ &= y\log\hat{y} + (1-y)\log(1-\hat{y}) && (7) \end{aligned}$$

which gives the log of the probability of obtaining y given x. We want this quantity to increase then the cost function is decreased, so we define our cost function as the negative log likelyhood function. [7]

Additionally, including a regularization parameter λ inspired by Ridge regression is often convenient, such that the cost function is

$$c(\boldsymbol{W})^+ = c(\boldsymbol{W}) + \lambda||\boldsymbol{W}||_2^2. \tag{8}$$

We will later study how this regularization affects the classification accuracy.

Activation function

Above, we were talking about the activation function, which is used to activate the net output. In binary models, this is often just a step function firing when the net output is above a threshold. For continuous outputs, the logistic function given by

$$\boxed{f(x) = \frac{1}{1+e^{-x}}.} \tag{9}$$

is usually used in logistic regression to return a probability instead of a binary value. This function has a simple derivative, which is advantageous when calculating a large number of derivatives. As shown in section **??**, the derivative is simply

$$\frac{df(x)}{dx} = x(1-x). \tag{10}$$

$tanh(x)$ is another popular activation function in logistic regression, which more or less holds the same properties as the logistic function.

Backward phase

Now all the tools for finding the outputs are in place, and we can calculate the error. If the outputs are larger than the targets (which are the exact results), the weights need to be reduced, and if the error is large the weights need to be adjusted a lot. The weight adjustment can be done by any minimization method, and we will look at a couple of gradient methods. To illustrate the point, we will stick to the **gradient descent** (GD) method in the calculations, even though other methods will be used later. The principle of GD is easy: each weight is "moved" in the direction of steepest slope,

$$\boldsymbol{w}^+ = \boldsymbol{w} - \eta \cdot \frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}}, \tag{11}$$

where η is the learning rate and $c(\boldsymbol{w})$ is the cost function. We use the chain rule to simplify the derivative

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial c(\boldsymbol{w})}{\partial out} \cdot \frac{\partial out}{\partial net} \cdot \frac{\partial net}{\partial \boldsymbol{w}} \tag{12}$$

where the first is the derivative of the cost function with respect to the output. For the cross-entropy function, this is

$$\frac{\partial c(\boldsymbol{w})}{\partial out} = -\frac{y}{out} + \frac{1-y}{1-out}. \tag{13}$$

Further, the second derivative is the derivative of the activation function with respect to the output, which is given in (10)

$$\frac{\partial out}{\partial net} = out(1-out). \tag{14}$$

The latter derivative is the derivative of the net output with respect to the weights, which is simply

$$\frac{\partial net}{\partial \boldsymbol{w}} = \boldsymbol{x}. \tag{15}$$

If we now recall that $out = f(\boldsymbol{x}^T\boldsymbol{w})$, we can write

$$\frac{\partial c(\boldsymbol{w})}{\partial \boldsymbol{w}} = [f(\boldsymbol{x}^T\boldsymbol{w}) - \boldsymbol{y}]\boldsymbol{x} \tag{16}$$

and obtain a weight update algorithm

$$\boldsymbol{w}^{+} = \boldsymbol{w} - \eta \cdot [f(\boldsymbol{x}^T \boldsymbol{w}) - \boldsymbol{y}]^T \boldsymbol{x}. \tag{17}$$

where the bias weight is included implicitly in $\boldsymbol{w}$ and the same applies for $\boldsymbol{x}$.

3.2 Feed-forward Neural Networks (FNN)

If you have understood logistic regression, understanding a neural network should not be a difficult task. According to **the universal approximation theorem**, a neural network with only one hidden layer can approximate any continuous function. [8] However, often multiple layers are used since this tends to give fewer nodes in total.

In figure (3), a two-layer neural network (one hidden layer) is illustrated. It has some similarities with the logistic regression model in figure (2), but a hidden layer and multiple outputs are added. In addition, the output is no longer probabilities and can take any number, which means that we do not need to use the logistic function on the outputs anymore.

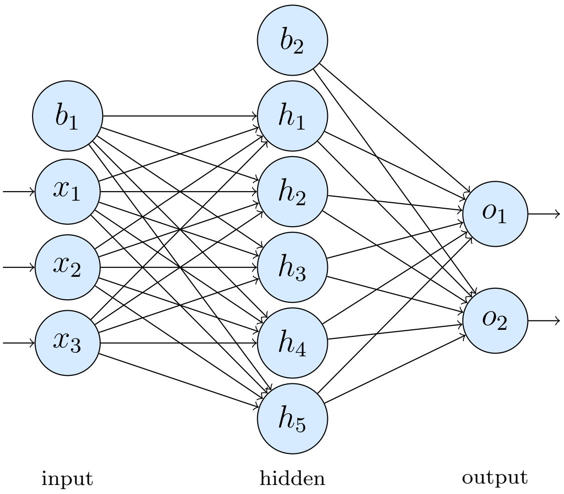

Figure 3: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

Without a hidden layer, we have seen that the update of weights is quite straight forward. For a neural network consisting of multiple layers, the question is: how do we update the weights when we do not know the values of the hidden nodes? And how do we know which layer causing the error? This will be explained in section **??**, where one of the most popular techniques for that is discussed. Before that we will generalize the forward phase presented in logistic regression.

Since it will be quite a lot calculations, I will just express the results here, and move the calculations to Appendix B. The forward phase in a three-layer perceptron is

$$\begin{aligned}
net_{hi} &= \sum_{j} w_{ji}^{(1)} \cdot x_j \\
out_{hi} &= \mathrm{f}(net_{hi}) \\
\\
net_{ki} &= \sum_{j} w_{ji}^{(2)} \cdot out_{hj} \\
out_{ki} &= \mathrm{f}(net_{ki}) \\
\\
net_{oi} &= \sum_{j} w_{ji}^{(3)} \cdot out_{kj} \\
out_{oi} &= \mathrm{f}(net_{oi})
\end{aligned} \tag{18}$$

which can easily be turned into vector form. The backward propagation follows from the two-layer example, and we get

$$\begin{aligned}
w_{ij}^{(3)} &= w_{ij}^{(3)} - \eta \cdot \delta_{oj} \cdot out_{ki} \\
\\
w_{ij}^{(2)} &= w_{ij}^{(2)} - \eta \sum_{k=1}^{O} \delta_{ok} \cdot w_{jk}^{(3)} \cdot f'(out_{kj}) \cdot out_{hi} \\
\\
w_{ij}^{(1)} &= w_{ij}^{(1)} - \eta \sum_{k=1}^{O} \sum_{l=1}^{K} \delta_{ok} \cdot w_{lk}^{(3)} \cdot f'(out_{kl}) \cdot w_{jl}^{(2)} f'(out_{hj}) \cdot x_i
\end{aligned}$$

where we again use the short hand

$$\delta_{oj} = (t_j - out_{oj}) \cdot f'(out_{oj}).$$

If we compare with the weight update formulas for the two-layer case, we recognize some obvious connections, and it is easy to imagine how we can construct a general weight update algorithm, no matter how many layers we have.

3.3 Convolutional Neural Networks (CNN)

Convolutional neural networks are known to be good at image classification, but how can we use it on sound classification? The idea is to turn the samples into images, and for that one can for example use a mel spectrogram, as described in the theory section.

CNNs are based on FNNs, but the image is initially feed through a few layers that extract the most important information.

Convolutional layer

Initially, the image is sent into a convolutional layer, which is meant to reveal structures and shapes in the image. The way we do it, is to introduce a filter that we slide over the entire image and multiply with all pixels (with overlap). Every time the filter is multiplied with a set of pixels, we sum all the multiplications and add the value to an activation map. The activation map is completed after we have multiplied the filter with the entire image. A typical filter has dimensions 16x16, but depends on the image shape. It is important to choose a filter that is big enough to cover structures.

Pooling layer

It is common to insert a pooling layer in-between convolutional layers, but why do we do that? A pooling layer is just a way to reduce the dimensionality of the representation such that we do not need to optimize that many parameters, but also helps to control overfitting. It works the way that we divide the representation (usually an activation map) into regions of equal size and represent each region with one single number. Max pooling is apparently the most popular technique, which just represents each region with the largest number in that region, but it is also possible to use average pooling, min pooling etc.. [`http://cs231n.github.io/convolutional-networks/`]

As an example, we can divide the image into 2x2 regions, which will reduce the size of the representation with 75%.

Dropout

Dropout is widely used in neural network layers, and is another method to prevent overfitting. The way it works is just to drop out units in the current layer.

To understand the idea, we need to take a close look at why overfitting occurs. Overfitting occurs when neighbor nodes collaborate, and become a powerful cluster which fits the model too specifically for the training set. To brake up those evil clusters, we can simply set random nodes to zero. [`https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropou`

Fully connected layer

The last part of a convolutional network is often refereed to as a fully connected layer, which is just a FNN. To be able to feed this layer with a input, we need to flatten out the reference.

3.4 Recurrent Neural Networks (RNN)

Last, but not least, we will examine recurrent neural networks. In the time domain, the sound at one time is dependent on sounds at other times, but the methods above are not able to utilize that because they lack memory.

In RNNs, information is shared between the different nodes, in that sense they have a short memory. This has

3.4.1 Long Short-Term Memory (LSTM)

3.4.2 Gated Recurrent Unit (GRU)

4 Optimization

In many cases we are not able to obtain an analytical expression for the derivative of a function, and to minimize it we therefore need to apply numerical minimization methods. An example is the minimum of the Lasso cost function, which does not have an analytical expression. In this section we will present two simple and similar methods, first ordinary gradient descent and then stochastic gradient descent.

4.1 Gradient Descent (GD)

The Gradient Descent method was mentioned already in the theory part, in equation (11), and this will therefore just be a quick reminder of the idea. The though is that we need to move in the direction where the cost function is steepest, which will take us to the minimum. The gradient always points in the steepest direction, and since we want to move in opposite direction of the gradient, the gradient is subtracted from the weights in every iteration. Updating the weights using gradient descent therefore reads

$$\boxed{W_{ij}^+ = W_{ij} - \eta \cdot \frac{\partial c(\boldsymbol{W})}{\partial W_{ij}}} \tag{19}$$

where W_{ij}^+ is the updated W_{ij} and η is the learning rate. An example implementation looks like

```
for iter in range(T):
        for i in range(N):
                y[i] = feedforward(X[i])
                gradient = (y[i] - t[i]) * df(y[i])
               W -= self.eta * gradient
```

4.2 Stochastic Gradient Descent (SGD)

We now turn to the stochastic gradient descent, which hence the name is stochastic. The idea is to calculate the gradient of just a few states, and hope that the gradient will work for the remaining states as well. It will probably not converge in fewer steps, but each step will be faster. We can express the SGD updating algorithm as

$$W_{ij}^{+} = W_{ij} - \frac{\eta}{N}\sum_{k=1}^{N}\frac{\partial c_k(\boldsymbol{W})}{\partial W_{ij}} \tag{20}$$

where we sum over all states in a so-called minibatch. After going through all minibatches, we say that we have done an epoch.

Because of the stochasticity, we are less likely to be stuck in local minima, and the minimization will be significantly faster because we calculate just a fraction of the gradients compared to ordinary gradient descent. Anyway, we expect this method to require more iterations than standard gradient descent.

```
for epoch in range(T):
        for i in range(m):
                random_index = np.random.randint(m)
                Xi = self.X[random_index:random_index+1]
                ti = self.t[random_index:random_index+1]
                yi = feedforward(Xi)
                gradient = (yi - ti) * df(yi)
               W -= self.eta * gradient
```

4.3 ADAM

ADAM is often the preferred optimization method in machine learning, due to its computational efficiency, little memory occupation and implementation ease. Since it was introduced in a paper by D. P. Kingma and J. Ba in 2015 [`https://arxiv.org/abs/1412.6980`], the paper has collected more than 15000 citations! So what is so special about ADAM?

ADAM is a momentum based method that only rely on the first derivative of the cost function. Required inputs are exponential decay rates β_1 and β_2, cost function $c(\theta)$ and initial weights θ, in addition to the learning rate η and eventually a regularization λ. Interatively, we first calculate the gradient of the cost function, and use this to calculate the first and second moment estimates. Further, we bias-correct the moments before we use this to update the weights. An implementation may look like this

```
def ADAM(eta, b1, b2, c, theta):
    m = 0
    v = 0
    for i in range(100):
        grad = d(c)/d(theta)
        m = b1*m + (1 - b1)*grad
        v = b2*v + (1 - b2)*grad*grad
        m_ = m/(1 - b1**i)
        v_ = v/(1 - b2**i)
        theta -= eta * m_/(sqrt(v_) + lambda)
```

5 Activation

Before 2012, the logistic, the tanh and the pur linear functions where the standard activation functions, but then Alex Krizhevsky published an article where he introduced a new activation function called *Rectified Linear Units (ReLU)* which outperformed the classical activation functions. [4] The network he used is now known as AlexNet, and helped to revolutionize the field of computer vision. [5] After that, the ReLU activation function has been modified several times (avoiding zero derivative among others), and example of innovations are *leaky ReLU* and *Exponential Linear Unit (ELU).* All those networks are linear for positive numbers, and small for negative numbers. Often, especially in the output layer, a straight linear function is used as well.

In figure (**??**), *standard RELU, leaky RELU* and *ELU* are plotted along with the logistic function.

5.1 Logistic

5.2 ReLU

5.3 ELU

6 Code

Using the language of machine learning: Python. Useful packages sklearn, keras, TensorFlow. Parallel processing using Dask. Pandas used in reading. Librosa

6.1 Code structure

6.2 Implementation

7 Results

7.1 Logistic regression

Table 1: Accuracy, softmax output

Epochs	Regularization							
	0.1		0.001		0.00001		0	
	Train	Test	Train	Test	Train	Test	Train	Test
10	0.3925	0.4112	0.2000	0.2042	0.2543	0.2447	0.2308	0.2429
20	0.4572	0.4802	0.2927	0.2843	0.3320	0.3238	0.4135	0.3919
30	0.5741	0.5842	0.2948	0.2815	0.3387	0.3395	0.4393	0.4186
40	0.5780	0.5603	0.2948	0.2861	0.3433	0.3385	0.4402	0.4103
50	0.5916	0.5998	0.3007	0.2833	0.3389	0.3395	0.4590	0.4453

7.2 Feed-forward Neural Networks

Table 2: Accuracy, sigmoid function was used in hidden layers, softmax on output. Dropout was used.

Epochs	Hidden nodes							
	1024		2x1024		3x1024		4x1024	
	Train	Test	Train	Test	Train	Test	Train	Test
20	0.8684	0.8464	0.8603	0.8602	0.8541	0.8602	0.8341	0.8298
40	0.9245	0.8924	0.9204	0.8942	0.9195	0.8868	0.8990	0.8924
60	0.9416	0.9062	0.9508	0.9154	0.9317	0.9016	0.9254	0.8960
80	0.9572	0.8942	0.9556	0.9163	0.9508	0.9172	0.9418	0.9016
100	0.9607	0.9016	0.9579	0.9126	0.9627	0.9181	0.9510	0.9200

We could go further and increase the number of epochs. With 3x1024 hidden nodes and 1000 epochs, we were able to get an accuracy of 0.99 for the training set and 0.94 for the test set.

7.3 Convolutional Neural Networks

Table 3

Epochs	Hidden nodes			
	1024	2x1024	3x1024	4x1024
100	0.4371	0.4381	0.3333	
200	0.4379	0.4367	0.3333	
300	0.4371	0.4382	0.3333	
400	0.9930	0.9929	0.9655	
500	0.9935	0.9934	0.9679	

7.4 Recurrent Neural Networks

Table 4

Epochs	Hidden nodes			
	1024	2x1024	3x1024	4x1024
100	0.4371	0.4381	0.3333	
200	0.4379	0.4367	0.3333	
300	0.4371	0.4382	0.3333	
400	0.9930	0.9929	0.9655	
500	0.9935	0.9934	0.9679	

8 Discussion

9 Conclusion

10 References

[1] E. M. Nordhagen. Project 1 - FYS-STK4155 - Applied data analysis and machine learning. https://github.com/evenmn/FYS-STK4155/tree/master/doc/Project1. (2018).

[2] S. Marsland. Machine Learning: An Algorithmic Perspective. Second Edition (Chapman & Hall/Crc Machine Learning & Pattern Recognition). (2014).

[3] P. Mehta, C.H. Wang, A.G.R. Day, C. Richardson, M.Bukov, C.K.Fisher, D.J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. (2018).

[4] ImageNet Classification with Deep Convolutional Neural Networks. A. Krizhevsky, I. Sutskever, G. E. Hinton. (2012).

[5] K. Allen. How a Toronto professor's research revolutionized artificial intelligence. thestar.com, retrieved November 3rd, 2018. (2015).

[6] ELU-Networks: Fast and Accurate CNN Learning on ImageNet. M.Heusel, D.A.Clevert, G.Klambauer, A.Mayr, K.Schwarzbauer, T.Unterthiner, S.Hochreiter. http://image-net.org/challenges/posters/JKU_EN_RGB_Schwarz_poster.pdf

[7] Logistic Regression-Detailed Overview. S.Swaminathan. https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

[8] Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning and convolutional networks. M.Hjorth-Jensen. https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/NeuralNet.html