FYS-STK4155 - Applied data analysis and machine learning

Project 3

Even M. Nordhagen

December 9, 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 60% of the test set correctly.

Contents

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. [Fel18] 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. In this section we will first introduce the Urban Sound Challenge, and then look at the sounds from a time perspective, frequency perspective and see how to extract information. Finally, we take a look at some of the samplings and see if we can classify them with the naked eye.

2.1 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 [Vid17].

The data sets consist of a total number of 8732 sound samplings (5434 training samplings and 3298 test samplings) 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.

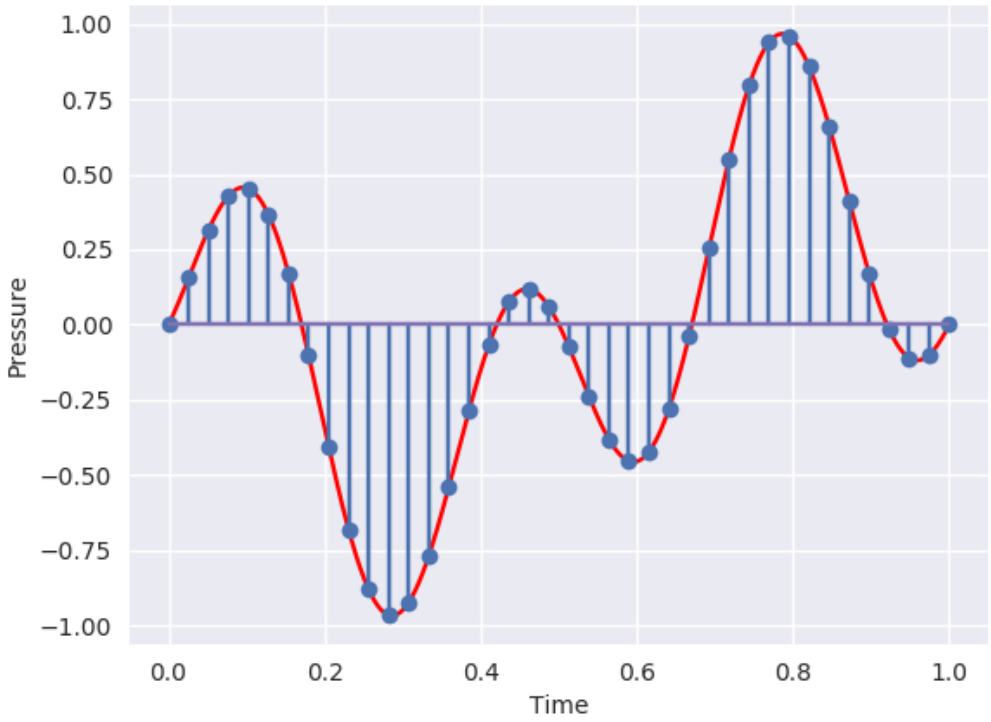


Figure 1: When sampling sound, we need to choose a high enough sampling. Here, the red line is the true sound and the blue dots are sampling points. The sampling rate here is 40 Hz, which is quite low and just for illustration.

How much information we loose depends on the sampling rate, which is the number of sampling points per second (see figure (1)). 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. [Gei16]

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. To go from the time domain to the frequency domain, one needs to use Fourier transformations, defined by

$$\hat{f}(x) = \int_{-\infty}^{\infty} f(t)\exp(-2\pi i x t)dx \tag{2}$$

where $f(t)$ is the time function and $\hat{f}(x)$ is the frequency function. Fortunately we do not need to solve this analytically, Fast Fourier Transformations (FFT) does the job numerically in no time. In figure (2), FFT is used on a time picture to obtain the corresponding frequency picture.

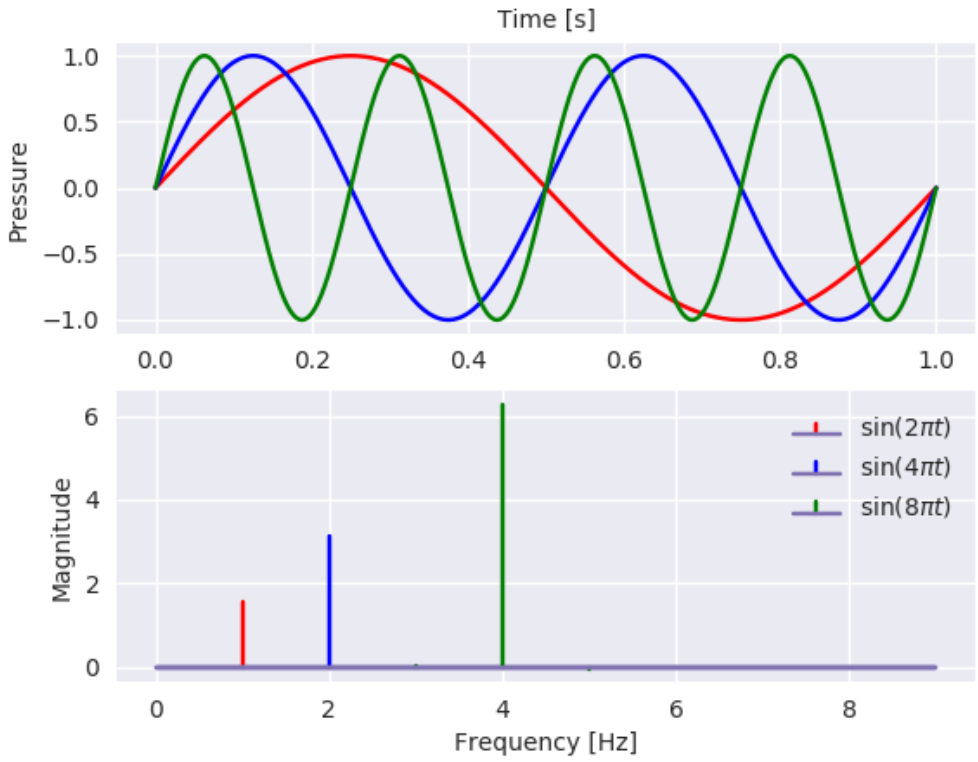


Figure 2: Time picture of three harmonic functions; $\sin(2\pi t)$, $\sin(4\pi t)$ and $\sin(8\pi t)$ (upper part) and the corresponding frequency pictures (lower part).

2.4 Extract features

There are many ways we can extract features from sound, but the most interesting are those that are based on how humans extract information. First the spectrogram method will be described briefly, and we thereafter turn to the mel frequency ceptral coefficients method.

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. [Pra15]

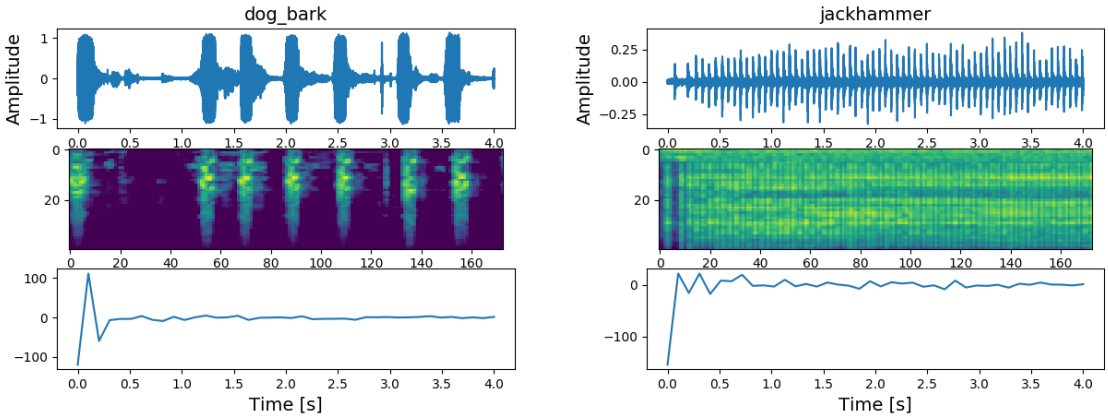
Mel Frequency Cepstral Coefficients (MFCCs)

MFCCs are features widely used in automatic speech and speaker recognition. [Lyo13] 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. Unlike the spectrogram, we are now left with a single array of information from the spectrum.

2.5 Examples from USC

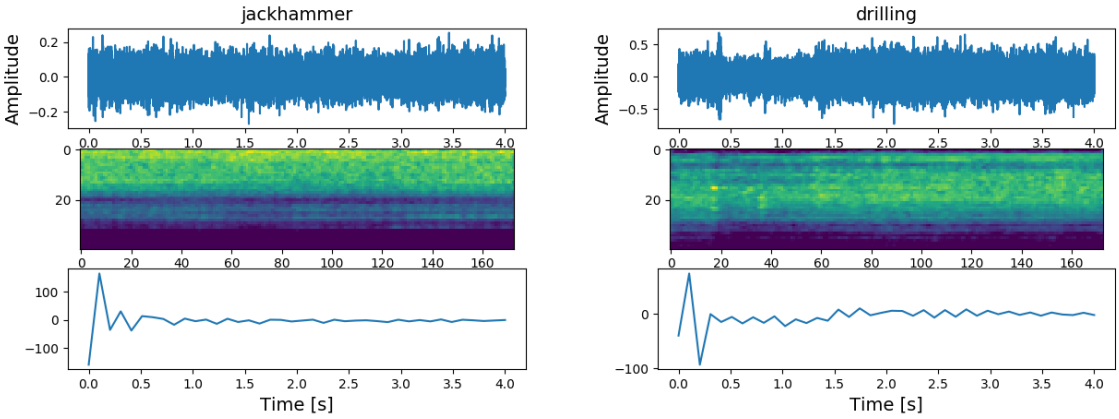
We will end this section with a few examples of what a sound file may look like. In figure (3), we compare one of the dog bark sounds with one of the jackhammer sounds. On the top, we have the time domain, in the middle the spectrogram is presented and on the bottom we have plotted the MFCCs. As we can see, they are quite different, and distinguish between those two should not be too hard. The spectrograms also really makes sense, but it is hard for a human eye to extract information from the MFCCs.



(a) Dog bark (b) Jackhammer

Figure 3: Time domains (top), spectrograms (middle) and MFCCs (bottom) for two random sound files, more specifically dog bark on the left and jackhammer on the right.

However, sometimes two time domains in the same category look totally different, for example if one compares the jackhammer in (4) with the one in (3). Will it be possible for our program to understand that they are in the same category? Also sometimes two time domain from different categories look similar, as we can see in (4). Can we really distinguish between them?



(a) Jackhammer (b) Drilling

Figure 4: Time domains (top), spectrograms (middle) and MFCCs (bottom) for two random sound files, more specifically jackhammer on the left and drilling on the right.

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, which means that each input node is multiplied with only one weight to get the net output. Since we want the probability of each class as a number between 0 and 1, we typically use the logistic function as activation function, and one hot encoder is often used to represent classes as arrays. In figure (5), logistic regression is illustrated as a single layer perceptron, with inputs on the left and side and outputs on right hand side.

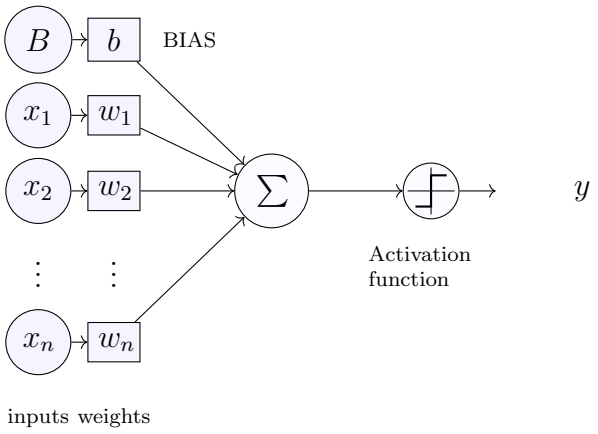


Figure 5: Logistic regression model with n inputs.

The first thing one might observe, is the bias on top, which is added to avoid the weights to explode or vanish. For instance, if we want the output to be 1, but the inputs are really small, the weights need to be large to give 1 without the bias. When adding the bias, we do not need that large weights, and the perceptron turns out to be more stable. The bias node value needs to be given, but the value does not really matter since the bias weight will be adjusted with respect to it.

The net output is then found to be

$$z = \sum_{i=1}^{n} x_i \cdot w_i + b \equiv \sum_{i=1}^{n} \mathrm{x}_i \cdot \mathrm{w}_i \tag{3}$$

where $\mathrm{x} \equiv [1, x]$, $\mathrm{w} \equiv [b, w]$ and the bias node value is set to 1. We get the real output by sending the net output through the activation function f, $y = f(z)$.

What we really want to find is the optimal weight values, which is not necessary unambiguous. Initially the weight are set to random, preferably small values, and they are updated with the formula

$$\mathrm{w}_i^+ = \mathrm{w}_i - \eta \cdot [f(\mathrm{x}_i \cdot \mathrm{w}_i) - t_i]\mathrm{x}_i \tag{4}$$

with η as the learning rate, t_i as a target and w_i^+ as an updated weight. This formula was derived from the cross entropy cost function and makes use of gradient descent, see [Nor18].

3.2 Feed-forward Neural Networks (FNN)

Feed-forward neural networks work mostly in the same way as the single layer perceptron, but the difference is that the perceptron model is not single anymore. With two sets of weights, we call it a double layer perceptron, and with more layers it is called a multi layer perceptron model. In figure (6), we have illustrated a double layer perceptron.

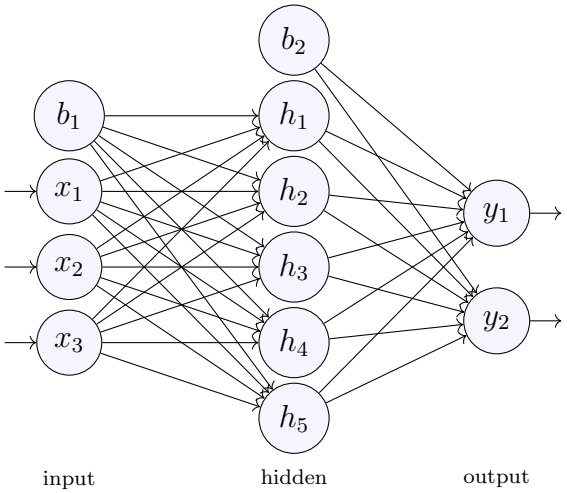


Figure 6: Neural network with 3 input nodes, 5 hidden nodes and 2 output nodes, in addition to the bias nodes.

As one can see, it differs from the single perceptron model in the way that it has one more layer. This makes the model nonlinear, and we are able to solve more complex problems than with the linear one. Actually, the double layer perceptron is able to approximate any continuous function according to the **the universal approximation theorem**.

Generally, each layer, included the output layer, consists of multiple nodes which gives us a matrix of weights between all the layers. The outputs are found in the same way as for the single perceptron, where the output at layer l is the dependent on the output at layer $l-1$:

$$z_j^{(l)} = \sum_{i=1}^{h_{l-1}} \mathrm{z}_i^{(l-1)} \cdot \mathrm{w}_{ij}. \tag{5}$$

where h_{l-1} is the number of nodes in layer $l-1$ and the outputs z^{l-1} again are assumed to take the bias node values. We activate each layer with an activation function, which can be layer unique, similarly as for the single perceptron: $y^l = f(z^l)$. There is a bunch of activation function to choose from, and we consider it so important that we branched it into its own section, see section 5.

The remaining part now is the weight updates, which are based on backward propagation. We then need to update the last set of weights first, and work back to the input,

$$\mathrm{w}_{ij}^{(3)} = \mathrm{w}_{ij}^{(3)} - \eta \cdot \delta_j \cdot \mathrm{z}_i^{(3)}$$

$$\mathrm{w}_{ij}^{(2)} = \mathrm{w}_{ij}^{(2)} - \eta \sum_{k=1}^{h_3} \delta_k \cdot \mathrm{w}_{jk}^{(3)} f'(\mathrm{z}_j^{(3)}) \cdot \mathrm{z}_i^{(2)}$$

$$\mathrm{w}_{ij}^{(1)} = \mathrm{w}_{ij}^{(1)} - \eta \sum_{k=1}^{h_3} \sum_{l=1}^{h_2} \delta_k \cdot \mathrm{w}_{lk}^{(3)} f'(\mathrm{z}_l^{(3)}) \cdot \mathrm{w}_{jl}^{(2)} f'(\mathrm{z}_j^{(2)}) \cdot \mathrm{z}_i^{(1)}$$

where we have used the short hand

$$\delta_j = (t_j - \mathrm{z}_j^{(3)}) \cdot f'(\mathrm{z}_j^{(3)})$$

and the $\mathrm{z}_i^{(1)}$ are just the input nodes. If we take a close look at the weight updating equations, we observe that there is a pattern. By defining a unique function $\delta_{ij}^{(l)} = \mathrm{w}_{ij}^{(l)} f'(\mathrm{z}_i^{(l)})$ for each layer, we can generalize the equations above, and find an expression for an arbitrary number of layers. It might be easier to vectorize first.

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.. [Kar13]

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. [Sri+14]

Fully connected layer

The last part of a convolutional network is often refereed to as a fully connected layer, which is just a FNN. The output from the other layers needs to be flatten out before it is sent into the FNN.

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 make the RNNs applicable when dealing with sequential data, such as handwriting recognition, speech recognition and other data sets where the input location matters.

In its simplest form, a so-called vanilla recurrent neural network, values on hidden nodes are fed back to the same nodes at the next time step, which creates a short-term memory. This is known to give better results on sequential data than ordinary FNNs, but it also has some cons that makes it useless in our case, especially the vanishing gradient problem. Instead, we will turn to more advanced methods which have a long-term memory as well, namely long short-term memory and gated recurrent units.

Long Short-Term Memory (LSTM)

Long short-term memory is an extension of the vanilla network, and is able to remember multiple previous time steps at the same time. It is therefore suited to extract important features which repeat occasionally. The way it works, is that it has three gates: an input gate, forget gate and output gate. Based on the targets, the network decides whether the input is important (pass to output gate) or noise (send to forget gate). Because of the forget gate, the gradients will remain steep and the vanishing gradient problem will not occur. Even though an input is sent to the forgot gate, the information will still be stored in the long time memory. [Don18]

Gated Recurrent Unit (GRU)

Gated recurrent unit is a relatively fresh method, which has shown similar performances as the more traditional LSTM. GRUs come with an input gate, update gate and a reset gate, which work in a similar way as the LSTM gates. The update gate determines how much of the past information is relevant for the future and the reset gate decides how much of the past information to forget. [Cho+14]

4 Optimization

In data science, we are surrounded by large data sets and functions to extract information from them. After a function describing the data set is found, one typical wants to find its minimum or maximum to optimize various parameters. Since the extremes seldomly are available analytically, one has to rely on numerical optimization methods. There exist loads of methods, but we will only look at two gradient methods: Gradient Descent and ADAM, where the latter is preferred.

4.1 Gradient Descent (GD)

The gradient descent method is the simplest optimization method available. It simply moves the steepest slope towards the extreme with constant step length (learning rate), calculating the gradient based on the entire data set. Mathematically, we can write it as

$$\theta_{ij}^{+} = \theta_{ij} - \eta \cdot \frac{\partial c(\boldsymbol{\theta})}{\partial \theta_{ij}} \tag{6}$$

where θ_{ij}^{+} is the updated θ_{ij} and η is the learning rate, in the same way as in section 3. GD has some major cons: it is likely to be stuck in a local extreme and because of its lack of adaptiveness, it is either really slow or likely to walk past the extreme. 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])
        theta -= self.eta * gradient
```

4.2 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 [KB14], 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 $c(\boldsymbol{\theta})$ and initial weights $\boldsymbol{\theta}$, 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

The activation function is a term that we repeatedly have used, but not really detailed. It is used to activate the outputs, which often need to take some certain properties. For instance, when doing classification, the outputs are probabilities and therefore between 0 and 1. A nonlinear activation function is often preferred to reinforce the non-linearity of the neural network.

Traditionally, the logistic function and the tanh function have been used as activation functions, since they where believed to work in the same way as the human brain. However, in 2012 a new network was introduced known as AlexNet, taking image recognition to a new level. They used a function named Rectified Linear Units (ReLU), which is zero for negative values. Based on this, some successful activation functions have been innovated in the recent years, such as Leaky ReLU and Exponential Linear Unit (ELU). They will be examined in turn below.

5.1 Logistic

The logistic activation function, also called the sigmoid function, is already mentioned several times in this report. It is widely used in classification, because it maps the outputs between 0 and 1 such that they represent probabilities for the different classes. The function reads

$$f(x) = \frac{1}{1 + \exp(-x)}. \tag{7}$$

Another important property is that the derivative has a simple form, which is just

$$\frac{\partial f(x)}{\partial x} = f(x)\big(1 - f(x)\big) \tag{8}$$

5.2 ReLU

The ReLU function is a modification of the pure linear function, where the function is pure linear for positive values and 0 else. This makes the function able to recognize the non-linearity in the model. [KSH12] The standard ReLU function is given by

$$f(x) = \begin{cases} x & \text{if} \quad x \geq 0 \\ 0 & \text{if} \quad x < 0. \end{cases} \tag{9}$$

This function obviously has a clean derivative, given by the step function

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ 0 & \text{if} \quad x < 0. \end{cases} \tag{10}$$

5.3 Leaky ReLU

The Leaky ReLU function is again a modification of the ReLU function, but where the gradient for negative values not necessarily is zero. The function can be written as

$$f(x) = \begin{cases} x & \text{if} \quad x \geq 0 \\ \alpha x & \text{if} \quad x < 0. \end{cases} \tag{11}$$

This is a more general activation function, where $\alpha = 1$ gives a pure linear function and $\alpha = 0$ gives the standard ReLU function. Usually one wants a small derivative for negative values, setting a to a small number. [Xu+15] The derivative reads

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ \alpha & \text{if} \quad x < 0. \end{cases} \tag{12}$$

5.4 ELU

Finally, we will look at the ELU function, which is another modification of the standard ReLU function. For the Leaky ReLU function, the zero gradient problem was avoided by adding a linear function with a small inclination. The idea behind ELU is similar, but the function added is an exponential function instead of a linear one. [CUH15] The function can be written as

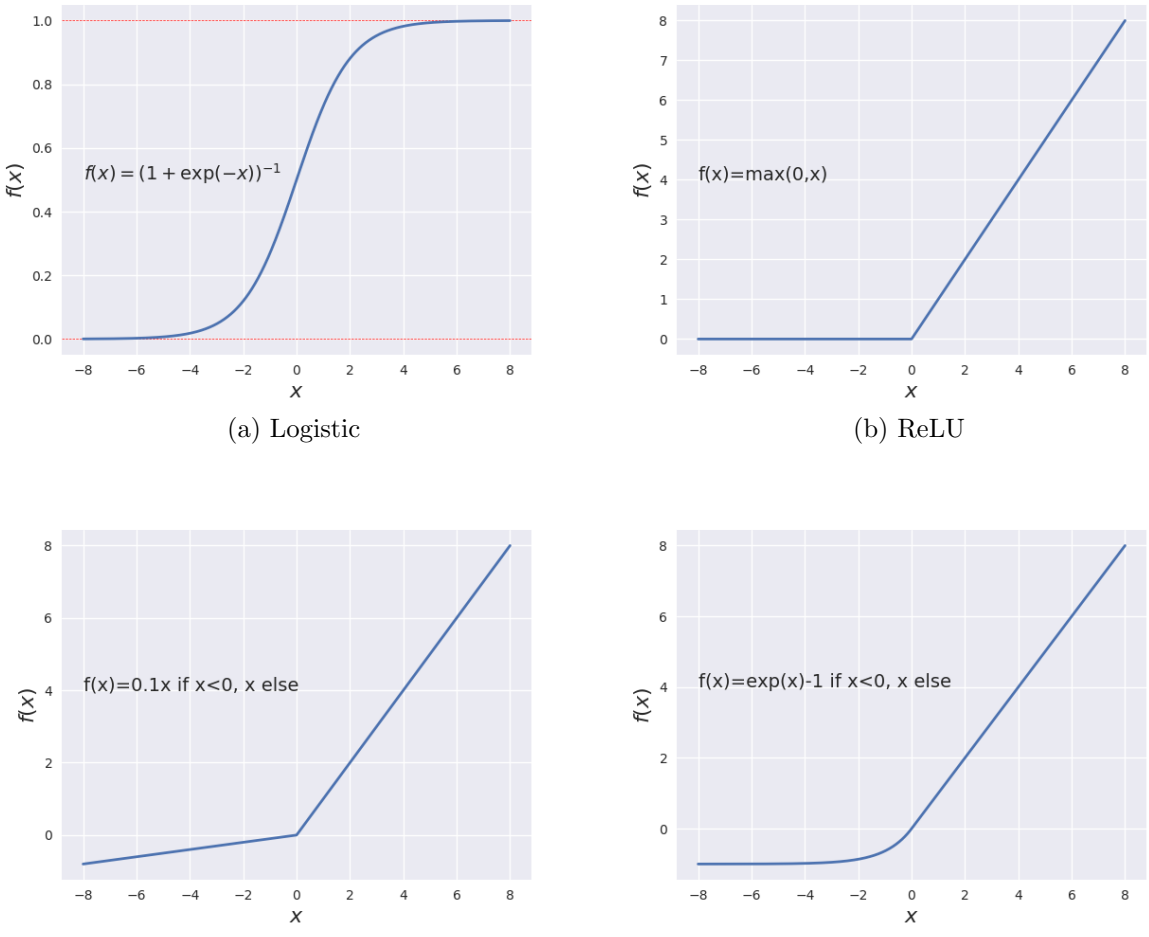
$$f(x) = \begin{cases} x & \text{if} \quad x \geq 0 \\ \alpha\big(\exp(x) - 1\big) & \text{if} \quad x < 0 \end{cases} \tag{13}$$

and also this has a simple derivative,

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1 & \text{if} \quad x \geq 0 \\ f(x) + \alpha & \text{if} \quad x < 0. \end{cases} \tag{14}$$

5.5 Illustration

In figure (7), *standard RELU, leaky RELU* and *ELU* are plotted along with the logistic function.



(a) Logistic

(b) ReLU

(c) Leaky ReLU

(d) ELU

Figure 7: Some popular activation functions: the logistic activation function (a), ReLU function (b), Leaky ReLU function (c) and the ELU function (d).

6 Code

For this project, the choice of programming language was not very difficult. Firstly, Python is a high-level language which is easy to work with and supports large data operations. Secondly, Python has a huge library of useful packages, which we will dive into very soon. Those two factors have made Python the language of machine learning.

6.1 Packages

When talking about packages in Python for scientific work, the **NumPy** package cannot be omitted. Due to its performance and simple implementation, NumPy is preferred for data storing, math operations and matrix modifications. [Oli06]

The first thing we had to do, was to transform the samplings to Python-friendly objects. For that, the **LibROSA** package is excellent, and was used to transform the sound clips to NumPy arrays. The same package was used to extract features from the clips, including creating spectrograms and finding MFCCs. [McF+15] To read the CSV files linking samples to classes, we used the **Pandas** package. [McK10]

After the features are extracted, we are ready for classification. In principle, we could have used the neural network which we implemented in Project 2, [Nor18], but it obviously cannot compete with optimized packages considering time performance and flexibility. For that reason, we used **TensorFlow** with **Keras** interface for all the classification methods, which is the considered the most feature-rich machine learning tool in Python. [Aba+15]

6.2 Code structure

The code structure is quite simple, and the entire operation is done by three functions *spectrogram.py*, *mfcc.py* and *classifier.py*. In addition, some scripts are written to generate plots used in this report. *classifier.py* consists of five classification functions, namely `Logistic`, `FNN`, `Convolutional`, `LSTM` and `Gated`. The code structure is illustrated in figure (8).

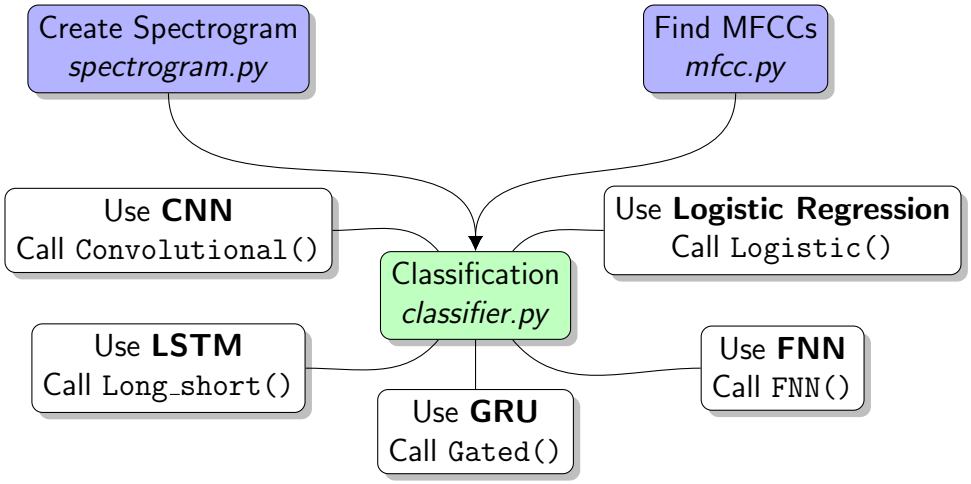


Figure 8: Code structure.

6.3 Implementation

The interface provided by **Keras** is very user-friendly and intuitive. Below a FNN with three hidden layers of 1024 nodes each is presented. The activation function on hidden nodes is the logistic function, while for the output we used the softmax function. A dropout of 50% is used on all layers and the ADAM optimizer is used with a batch size of 32 and 100 epochs.

```
def FNN(N=3):
    X_train, t_train, X_val, t_val = load_mfcc()

    num_labels = t_train.shape[1]

    model = Sequential()

    model.add(Dense(1024, input_shape=(40,)))
    model.add(Activation('sigmoid'))
    model.add(Dropout(0.5))

    for i in range(N-1):
        model.add(Dense(1024))
        model.add(Activation('sigmoid'))
        model.add(Dropout(0.5))

    model.add(Dense(num_labels))
    model.add(Activation('softmax'))

    model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer
        ↪ ='adam')
    model.fit(X_train, t_train, batch_size=32, epochs=100, validation_data=(X_val,
        ↪  t_val))
```

The other implementations are quite similar, and we will therefore not detail them here. For the entire code, see `https://github.com/evenmn/FYS-STK4155`.

7 Results

Finally we are ready to present the results, where the accuracy score is in our focus. We will evaluate each method separately, starting with the logistic regression and moving on to the neural network based methods. What is common for all methods, is that the softmax function was used as output activation function and ADAM was used as minimization tool with a batch size of 32 and a various number of epochs.

7.1 Logistic regression

For logistic regression, we examined how the accuracy was dependent on the regularization parameter with the values 10^{-1}, 10^{-3}, 10^{-5} and 0. In addition, the number of epochs was varied in the range $[10, 50]$. The results can be found below in table (1).

Table 1: The accuracy-score for the training set (Train) and test set (Test) with a changing regularization parameter. The number of epochs was set to 10, 20, 30, 40 and 50. As optimization tool, ADAM was used, and we used the softmax activation function.

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

As we can see, the more epochs the higher accuracy, but when increasing the number of epochs further, the score does not improve much. Apparently, there is no such linear dependency when it comes to the regularization, but we observe that 0.1 works best.

7.2 Feed-forward Neural Networks

We found the FNNs to work best without regularization, and will not present results with various regularization parameters. Instead, we vary the number of

hidden layers, where each has 1024 nodes. Also here we present the accuracy-score for different number of epochs, going from 20 to 100. See table (2) for the actual accuracy.

Table 2: The accuracy-score for the training set (Train) and test set (Test) with 1-4 hidden layers with 1024 nodes each. The number of epochs was set to 20, 40, 60, 80 and 100. As optimization tool, ADAM was used, with a batch size of 32. On the output layer we used softmax activation, and on the hidden layers the logistic function was used. A dropout of 50% was used in all layers.

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**

Again we see that the more epochs the higher accuracy. We went further increasing the number of epochs to 1000, and got a training accuracy of 0.99 and a test accuracy of 0.94 for a network with three hidden layers. Apparently, the test accuracy is more sensitive to the number of hidden layers then the training accuracy.

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

References

[Oli06] E.T. Oliphant. "A guide to NumPy". In: *Trelgol Publishing* (2006).

[McK10] W. McKinney. "Data Structures for Statistical Computing in Python". In: *Proceeings of the 9th Python in Science Conference* (2010), pp. 51–56.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Commun. ACM* 60.6 (May 2012), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: http://doi.acm.org/10.1145/3065386.

[Kar13] A. Karpathy. *Convolutional Neural Networks for Visual Recognition.* http://cs231n.github.io/convolutional-networks/. [Online; accessed 06-December-2018]. 2013.

[Lyo13] J. Lyons. *Mel Frequency Cepstral Coefficient (MFCC) tutorial.* http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/. [Online; accessed 06-December-2018]. 2013.

[Cho+14] K. Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[KB14] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[Sri+14] N. Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[Aba+15] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[CUH15] D. Clevert, T. Unterthiner, and S. Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *CoRR* abs/1511.07289 (2015). arXiv: 1511.07289. URL: http://arxiv.org/abs/1511.07289.

[McF+15] B. McFee et al. "LibROSA: Audio and Music Signal Analysis in Python". In: (2015).

[Pra15] K. Prahallad. "Speech Technology: A Practical Introduction." In: (2015).

[Xu+15] B. Xu et al. “Empirical Evaluation of Rectified Activations in Convolutional Network”. In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853.

[Gei16] A. Geitgey. *How to do Speech Recognition with Deep Learning.* https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a. [Online; accessed 06-December-2018]. 2016.

[Vid17] Analytics Vidhya. *Practice Problem: Urban Sound Challenge.* https://datahack.analyticsvidhya.com/contest/practice-problem-urban-sound-classification/. [Online; accessed 06-December-2018]. 2017.

[Don18] N. Donges. *Recurrent Neural Networks and LSTM.* https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5. [Online; accessed 07-December-2018]. 2018.

[Fel18] R. Feloni. *20 years from now, regular people will have the personal assistants currently reserved for the rich and famous - through tech.* https://nordic.businessinsider.com/ai-assistants-will-run-our-lives-20-years-from-now-2018-1?r=US&IR=T”. [Online; accessed 06-December-2018]. 2018.

[Nor18] E. M. Nordhagen. *FYS-STK4155 - Applied data analysis and machine learning.* 1nd Edition. 2018.