AU

Artifact annotation

Artefakt annotering

Student Name	Student ID
Marc Jensen	201907421

Assistant professor (supervisor): Kaare Mikkelsen

Bachelorprojekt i datavidenskab,
Aarhus University, March 24, 2025

Resume

Artefakter er b.la. støj der generes i måling af EEG-data. Denne støj kan både opstå fra patienten som EEG-dataet bliver målt på, andre udefrakommende faktorer og det udstyr der anvendes til at måle EEG-dataet. At annoterer og at fjerne disse artefakter er yderst essentiel når EEG-dataet skal analyseres. Det at finde disse artefakter er dog ikke en let opgave. I denne rapport bliver et modificeret u-netværk, som er et deep learning netværk, trænet til at finde artefakter. Dette netværk sammenkobler et Convolutional Neural Network (CNN) og et Recurrent Neural Network (RNN). Den første version af netværket i denne raport trænes til at skelne mellem rent EEG-data og artefakter. Den anden version af netværket bliver også trænet til at kunne skelne mellem rent EEG, normale artefakter og elektrode artefakter. Evalueringen af de to før nævnte netværker viser lovende resultater. I sammenligning med andres arbejde, hvor både et CNN og et RNN anvendes (men hver for sig), opnår denne rapports netværker højere nøjagtighed på test-data. Disse resultater demonstrerer, at anvendelsen af deep learning netværker til automatisk at annotere artefakter i EEG-data er muligt. Det at opnå automatisk annotering af artefakter vil danne grundlag for, at færre menneskelige kræfter skal bruges på at annotatere EEG-data. Dette vil forhåbentligt også kunne forøge annoterings-hastigheden og derved mindske ventetiden for at opnå diagnosen for patienten.

Abstract

Artifacts are in this report defined as noise generated in EEG data. The artifacts can stem from the patient that the EEG data is measured from, other environmental factors or the equipment (electrodes) used for measuring the EEG data. To annotate and remove these artifacts are essential to prepare the EEG-data for further analysis. However, it is not an easy task to identify the artifacts. In this report a modified u-network (a deep learning architecture) will be trained to annotate artifacts. The network presented in this report use a combination of a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN). The first version of the network presented in this report is trained to distinguish between clean EEG-data and artifacts. The second version of the network is trained to distinguish between clean EEG-data, and to annotate the artifacts as either normal artifacts or electrode artifacts. The evaluation of the two versions of the network both shows promising results. Compared to networks using only a CNN or a RNN for annotating EEG-artifacts, the networks presented in this report show better accuracy on the test-data. These results indicate, that it will be beneficial to implement deep learning networks to automatically annotate artifact in EEG-data. For the patient this will hopefully reduce the waiting time to obtain a diagnosis.

Contents

1 | Introduction

ElectroEncephaloGraphy (EEG) is a very important method in diagnosing different neurological disorders. EEG is often used when diagnosing e.g. epilepsy. EEG might also be used in the diagnosis of brain tumors, sleeping disorders, and much more [1].
EEG is a noninvasive neurophysiological technique for measuring electrical activity inside the head of a patient [2]. These measurements are achieved by electrodes placed on the scalp of the patient [1].
The main point of an EEG experiment is to investigate the activity in the brain of the patient. The problem at hand is that the electrical signals from the brain are weak. This means that other processes from e.g. inside the head can contaminate the data measured from the patient.
Other measurement errors might also occur in the EEG recordings. A broad term for these errors are artifacts. A visual presentation of the most common artifacts will be displayed in subsection 2.3.
The first line of defense against artifacts is to limit the amount generated during the experiment. This is often done by giving the patient information about how artifacts are generated [2, page 98]. The equipment used to record EEG data can also generate artifacts. Examining and repairing equipment is thus important to ensure good quality EEG-data [3]. Removing all artifacts while recording the data is nearly impossible - just movement of the eye can cause an artifact. Thus afterwards cleaning the data of artifacts is very important. This is the main subject of the thesis.
You may ask: ”Why does artifacts create such a problem?”
The main challenge is for the further analysis of the EEG data. As noted, artifacts are noise in the EEG data. This noise might create confounding errors, and thus introduce bias and create false results [4, page 1]. A good saying go: Garbage in, garbage out. Thus if the artifacts are not removed, the conclusion based on the EEG recordings might be wrong. In the worst case the researchers might reach a wrong diagnosis for the patient.

1.1 | The goals of this report

The main goal of this report is to create a neural network that can do automatic artifact annotation. The following list will contain the main questions of research:

1. Can an deep-learning pipeline/network do automatic artifact annotation?
2. Would it be possible to make an neural network that provide information about the quality of the EEG equipment?
3. Does a neural network outperform a naive model in artifact annotation?
4. When training the neural network how should data be sampled?
5. How does the optimizers Adam and SGD perform when training the neural network?
6. How does the linear learning rate test compare to the exponential learning rate test?

2 | Theory

2.1 | Background for artifact removal/annotation

Before 1990 the go to tool for artifact removal was linear regression algorithms [4, 5]. These tools were great because they where not computationally heavy. But these methods have proven not sufficient. Mostly because of the fact that physiological processes are non-linear and non-stationary. Thus the linear methods does not fully adapt to the EEG-data [4, 6, 5]. Another issue with the linear regression algorithms is that they can only treat few specific types of artifacts [6, page 5].
Simple filtering is generally not the best option for removing artifacts. Unless it is for removing narrow band artifacts. This could for instance be environmental line noise created by the power grid [5, page 9]. The key issue that permits the usage of simple filtering is the fact that a variety of artifacts often overlap with the signal of interest. This often happens in the spectral and temporal domain, and might also happen in the spatial domain [6, page 288]. Using a filter would therefore remove some data of interest [4, page 2].
The issue at heart of artifact annotation is that an annotator can distinguish artifacts from data of interest. The issue is that the recognition of artifacts are hard to explain especially formally. This is a problem when trying to do artifact annotation automatically with a computer. The hope of using deep learning for this task, is to allow a machine to learn artifact annotation by experience [7]. Neural nets have shown great results in the fact that they have the ability to acquire their own knowledge by pattern extraction [8, 7].

2.2 | Data set

In this report the Temple University Hospital's EEG Artifact Corpus data will be used [9]. This data set consists of 310 EEG files. These files consists of actual clinical data. The artifacts in these files have been annotated by trained students. To ensure good quality data the students conducted an inter-rater agreement. This agreement is especially important for the uniformity between annotations [9, 10].
Since the data consists of actual clinical trials, the artifacts appearing are plenty and challenging to find [10].

The measured EEG signal is actually a digitized version of the electrical potential measured between an electrode and a reference point [11, page 2]. Three different configurations of references is used in the data set. Firstly the average reference (AR) which uses the average of certain electrodes as the reference. Thus the average is subtracted from the signals of each electrode. This is done for each sample and is to account for common noise [8, page 199]. Secondly the linked ears (LE) which uses the electrodes on the ears as reference. This is often used on the assumption that the ears have no electrical activity [8, page 199]. Some argue that this linked ears configuration reduces artifacts [11, page 4]. Thirdly a configuration of the average reference where the A1_ref and A2_ref channels are not in use [8, page 199]. Other reference configurations are possible e.g. just using a single electrode as reference [2, page 55]. One can also reference the different electrodes to each other. Meaning that no common electrode/average is used.
The model should be capable of working no matter the reference or the amount of signals. Thus all three configurations in the data set are kept for training and evaluation.

The EEG data is a time series and can be denoted as $x_{1:T} \in \mathbb{R}^{C \times T}$. Where C is the amount of recorded channels [11, page 2]. The time series length is denoted as T. The annotations can be written as $y_{1:T} \in \{0, 1\}^{M \times T}$. This is for the binary case where 0 means no artifact and 1 means an artifact is present. In this case M denote the number of bipolar montages imposed to the EEG data [11, page 3]. When neurologists view the EEG data, they look at derivations of the data. E.g. the channels FP1-REF and F7-REF can be turned into the derivation FP1-F7 [11, page 6 & 7].
The data is stored according to the reference used, in this case an identifier and the official patient number. In the data there are a total of 213 patients. In the patient folder there is at least one EEG recording, but multiple recordings might occur. For each recording an .edf file containing the EEG-data is stored. There is also a .csv file that contains the annotation of the artifacts. These annotations are stored on a time-interval format. These time-intervals refer the .edf file on when the artifacts start and end. Information about what channels the artifacts occurs on and the type of artifacts are stored in the .csv file. Additional data about the patient is stored in a .txt file.

The Corpus data also contain a montage of which electrodes are in accordance to each-other.

2.3 | Visual representation of data

In the following a plot of EEG data is presented:

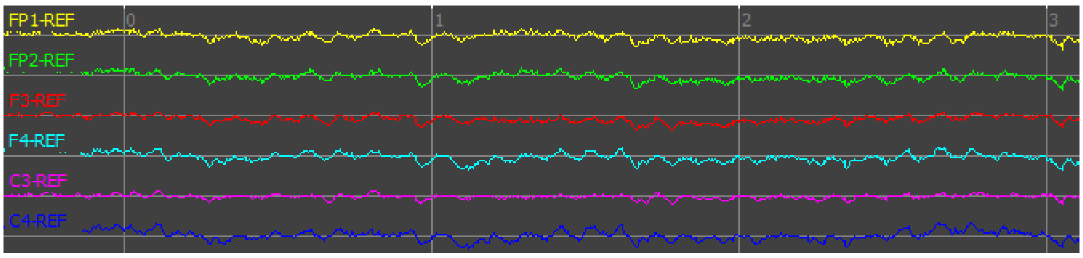


Figure 2.1: Example of clean EEG data
Plot made using TUH EEG dataset

This plot of EEG data has time on the x-axis (in seconds) and amplitude on the y-axis (using a scale of 100 µV). It can also be seen, that the data is recorded over multiple channels (each channel is represented by a different color). It is worth noting, that the horizontal white lines indicate 0 µV for the signals. Note that this part of the EEG recording does not contain artifacts. On the amplitude these signals are well in the bounds of -40 µV to 40 µV.

2.3.1 | Definition of the different artifacts

Lets explain some of the most common groups of artifacts according to the data set chosen [10].

- Chewing (CHEW): An artifact created by the patient tensing and relaxing his jaw muscles. This artifact type is a subset of the muscle artifact. Thus the same characteristic are present. This will be seen as high frequency sharp waves with 0.5 sec baseline periods between bursts on the plot.
- Electrode (ELEC): This artifact is created by various electrode errors. The lead artifact often occur when an electrode is moved by the patients head and/or poor connection of the electrode. Electrode pop (elpp) happens when different channels with the same electrode experience “spiking” with an electrographic phase reversal. The presence of electrode errors often result in disorganized and high amplitude slow waves.
- Eye Movement (EYEM): An artifact that is produced by the patient moving his eyes. The signal often become a spike-like waveform. These artifacts are mostly present in the electrode placed on the front of the patients head.
- Muscle (MUSC): This artifact are often produced by agitation in the patient. This type of artifact often occur with high frequency, sharp waves related to the patient moving. This type of waveform often occur above 30 Hz with no specific pattern.
- Shiver (SHIV): This artifact is created when the patient shivers. This is often seen on all or most channels. This artifact is characterized as being a specific and sustained sharp wave in the EEG.

These artifact types can also overlap in time. To register overlapping artifacts the annotators use combined artifact groups. For example does the class of eyem_musc exist. This is when both an eye movement and muscle artifact happen at the same time [10].

In the following plots some of the most common artifacts will be presented:

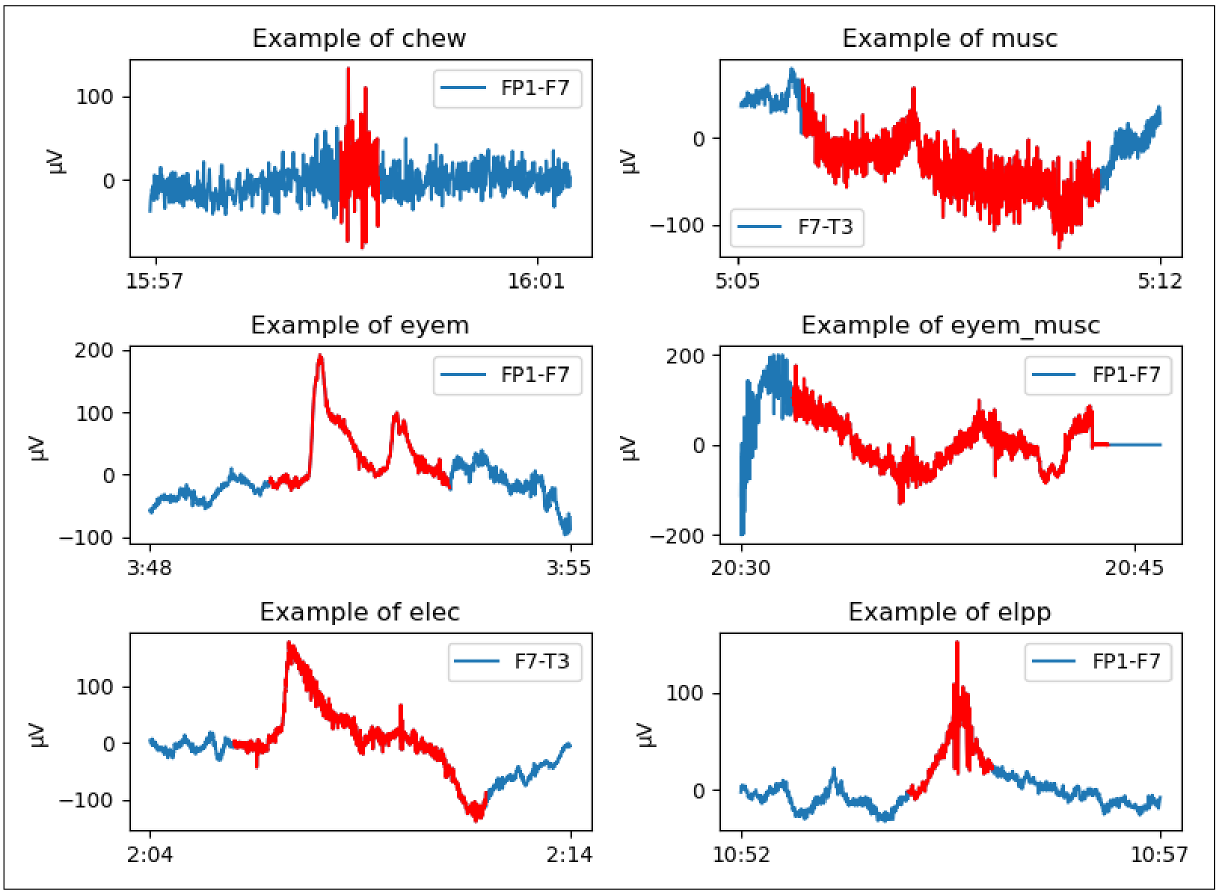


Figure 2.2: Example of artifacts
Plot made using TUH EEG data set

In each of the plots above a single signal is shown. On the x-axis is the time-interval and on the y-axis the amplitude in µV. As it can be seen from the plots, that the artifacts can have large amplitudes.

2.4 | More about the data

The EEG recordings have been collected from 2002 to the present day. The recordings has taken place at Temple University Hospital (TUH) [9].
There are a total of 310 EEG files on the .edf format. These files have been recorded from 213 patients in 259 sessions. Therefore some patients contributes multiple files to the data set. In some cases data from a single session might be stored in multiple files. The data set does not contain information about connection between files recorded in the same session. Thus the authors of the data set suggest treating each .edf file as an independent recording [11, page 7].
These 310 files amount to a total duration of 5,998.93 minutes (which is about 99.98 hours). In the table below, the total amount of different events are listed:

Artifact	Events	Artifact	Events
musc	51,052 (31.89%)	eyem	38,569 (24.09%)
elec	33,130 (20.70%)	eyem_musc	18,677 (11.67%)
musc_elec	7,651 (4.78%)	chew	6,482 (4.05%)
eyem_elec	2,422 (1.51%)	eyem_chew	864 (0.54%)
shiv	613 (0.38%)	chew_musc	243 (0.15%)
elpp	172 (0.11%)	chew_elec	152 (0.09%)
eyem_shiv	45 (0.03%)	shiv_elec	1 (0.00%)

The table above shows the number of occurrences of a specific type of artifact and how often this type of artifact occur in percentage. In the 310 EEG files are 160,073 artifact events present.

3 | Methods

3.1 | Code availablity

The code to produce the results in this report can be found in the github repository: https://github.com/marctimjen/Artefact-Rejection.

3.2 | Preprocessering

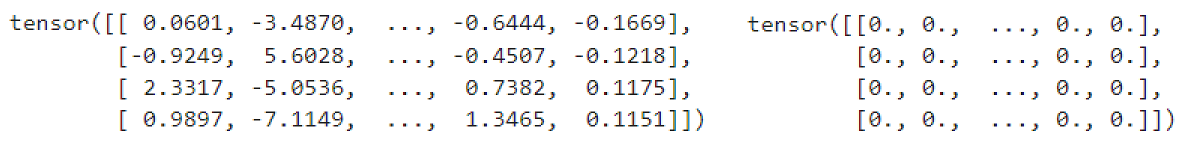
As explained in the theory chapter, the authors of the data set suggests treating each EEG-file as an independent recording. Therefore each .edf file is treated as its own recording in this report.
It is also described in the theory section, that a simple filtering of the EEG-data is a way to remove contamination from the electrical grid. The EEG-data has been recorded in the US. The utility frequency in the electrical grid in the US is 60 Hz. Thus the data is filtered with a notch-filter using a sampling rate of 60 Hz.
To get the area of interest, a band-pass filter with a sampling rate from 0.1 to 100 Hz is used. This is a typical step, to only look at the sampling region of interest. Secondly the filtering demeans the data. Both the notch-filter and the band-pass filter use the FIR-filter type. The notch-filter and band-pass filter utilize different filter length accordingly to the size of the transition regions [12].

The EEG-data is recorded with the different rates: 250 Hz, 440 Hz, 500 Hz 1000 Hz. The data has been band-pass filtered with an upper passband edge of 100 Hz. Thus the fastest frequency that data will contain is a rate of 200 Hz. Thus a re-sampling of 200 Hz has been used on the data [12]. The re-sampling step is important in order to not confuse the model. If the sampling rate would change between the inputs, the duration the recorded data will also change.

In some of the EEG-data experiments an EKG channel is present. This channel captures information on the electric signals from the heart [13]. The EKG data is vastly different from the EEG-data and there is no guarantee that this channel is present in the data. Thus the EKG channel has been removed in the prepossessing step of this project.

In the data only derivation with annotated artifacts have been selected. This step has been taken to make sure, that the derivation used for the model, have been checked by an annotator. This step results in a possible loss of data, as a channel might have no artifacts and thus be dropped. Discussion on correct annotations and data quality will be addressed in the discussion section.

The EEG-data has now been filtered, re-sampled and only the appropriate derivations have been chosen for further analysis. According to which derivation have been selected, the targets will be generated. The targets are produced from the time-intervals present in the .csv file. The target will be a tensor containing binary indicator values. Thus a 0 means no artifact and a 1 means that an artifact is present. All this processing of data are done in the script "0 Transform_data.py" [1]. There is also implemented other versions of this script (see the repository). The most notable change is that these scripts clamp the interval of the EEG between -200 and 200 µV (so if the EEG signal is larger than 200 µV, the value is set to be 200 µV). After the processing the data will have this form:



```
tensor([[ 0.0601, -3.4870,  ..., -0.6444, -0.1669],
        [-0.9249,  5.6028,  ..., -0.4507, -0.1218],
        [ 2.3317, -5.0536,  ...,  0.7382,  0.1175],
        [ 0.9897, -7.1149,  ...,  1.3465,  0.1151]])
```

```
tensor([[0., 0.,  ..., 0., 0.],
        [0., 0.,  ..., 0., 0.],
        [0., 0.,  ..., 0., 0.],
        [0., 0.,  ..., 0., 0.]])
```

Figure 3.1: Plot of the experiment EEG on the left and the annotations on the right
Plot made using the processed data

In this plot, the input data can be seen on the left, and the target data can be seen on the right. Note that the targets (annotations of artifacts) might be different accordingly to which derivation is in use. The electrodes on the frontal part of the head eg. Fp1 or Fp2 would be more exposed to eye artifacts, than the electrodes on the back of the head eg. O1/Oz/O2. Thus there is a connection between the different type

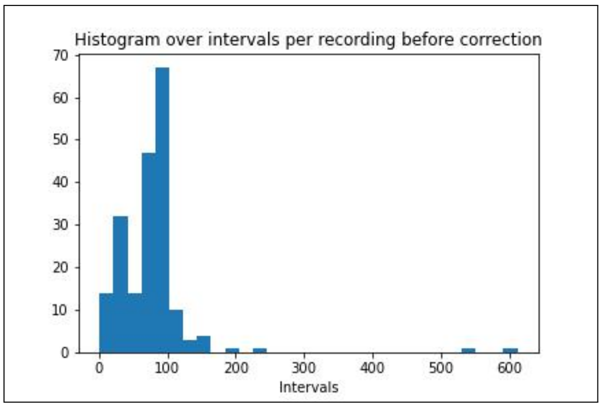
[1] https://github.com/marctimjen/Artefact-Rejection/blob/main/Preprocess/0%20Transform_data.py

of artifacts and the derivations.

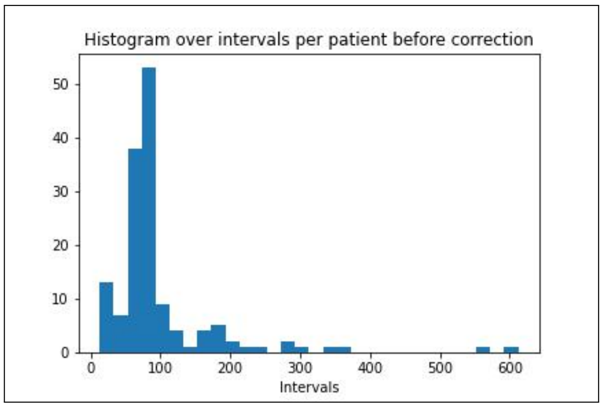
Plotting the data revealed that some of the EEG-recordings have an unstable beginning. Because of this the first 30 seconds of the data is removed. In the training phase of the neural network 5 minute intervals will be drawn at random from the EEG-data. These intervals consists of the cleaned EEG data and the associated target values. Some of the recordings are extremely short. Therefore only EEG-recordings longer than 5 minutes and 30 seconds can be used for training the model.

It is noted that EEG-data contains idiosyncratic features. This means that the artifacts appearances might change from person to person [2, page 101]. As explained in the theory chapter some patients contributed more than one recording to the data set. Therefore it is important that data from the same patient does not appear in both the train set and/or the validation/test-set. This is important otherwise the test results might not indicate if the networks have generalized well to the data.
After the data processing up to this point, a total of 207 patients are represented. These 207 patients have a total of 279 recorded sessions that can be used in the analysis. To optimize the model the data should be split into 70 % train, 20 % test and 10 % validation. This will allocate 195, 56 and 28 recordings to the train-, test- and validation-sets respectively. If perfect distribution was used the number patients would be represented as 145, 41, and 21 in train-, test-, and validation-set respectively. Because of randomness and the structure of the data, the final cut came out to be 145, 39 and 23 patients in the train-, test- and validation-set.
It is noted that the 8 patients who contributed the most sessions to the data set contributed a total of 41 sessions. This amounts to an average of 5.1 sessions per patient. These patients were forced into the train-set, as to not over-represent a single patient in the test-/validation-set. The splitting of data is done in the python script "1 train-test-split.py" [2]

In the training of the network 5 minutes intervals of EEG-data is loaded into the model at a time. Firstly it is checked, if there is a problem in the distribution of the data. In order to do this a histogram over the amount of intervals per recorded session is made. This is to indicate how much data is loaded per EEG-recording. Since some patients have multiple data-series recorded, the amount of intervals sampled per patients is aggregated. Another histogram is made for this data:



(a) Histogram of intervals per recording



(b) Histogram of intervals per patient

Figure 3.2: Histogram of the intervals per recordings and patients before any correction
plot created using the processed data

The left histogram shows that most intervals are centered around 0-100 intervals per recording. The histogram also shows that a few recordings are extremely long. Therefore many intervals can be drawn from these long recordings. One with about 550 intervals drawn and one with a little under 700 intervals. On the right is a histogram over the amount of intervals drawn per person. This lead to the same conclusion. The data of intervals per recording has a mean and median equal to 74 and 66. Therefore it can be concluded that the distribution of data is right skewed as the median is lower than the mean. To illustrate this point, the top 25 % of patients with the longest recordings are collected. This is about 36

[2] https://github.com/marctimjen/Artefact-Rejection/blob/main/Preprocess/1%20train-test-split.py

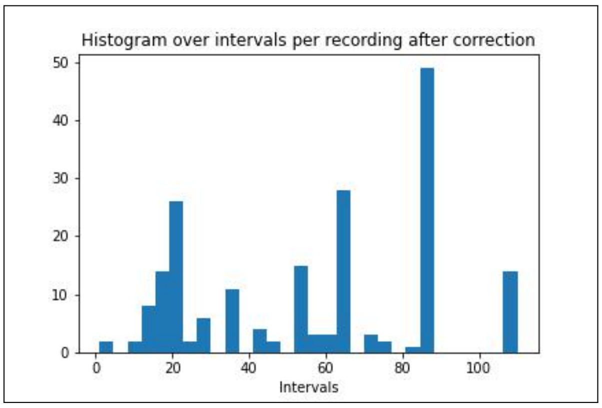
out of 145 patients. These patients contribute about 48 % of the data. In Figure 3.4 a Lorenz Curve is plotted. The Lorenz Curve also illustrates that a relatively minor amount of patients contribute a great amount of data.

Based on the above it is necessary to correct for the over representation of some patients. In this report a python dictionary is created to keep track of how many intervals should be drawn from which series. This is done in the script "2 series_dict.py" [3]. Especially line 70-71 is important:

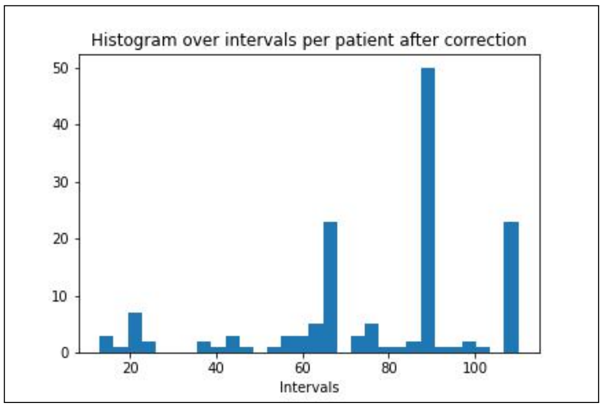
```
res = int(min(torch.div(110, len(values), rounding_mode='trunc'), length))
```

A value of 110 is the maximum amount of intervals a patient can contribute across all the persons recordings. This is in order to not over represent a few individuals in the data set. The "len(values)" indicates how many recorded sessions the patient has contributed. If a patient contributed multiple recordings, the script will divide how many intervals can be drawn per session. The "length" variable indicates how many possible intervals in the given experiment is possible. This is to ensure that a single recording is not over-sampled. An extreme example would be if a patient contributed a recording with only 1 channel, and the recording is 5 minutes and 30 seconds long. If the minimum on the length parameter was not implemented, the exact 5 minutes would be loaded into the model 110 times.

The following histograms shows the same data as Figure 3.2 but after the implementation of the series dictionary:



(a) Histogram of intervals per recording



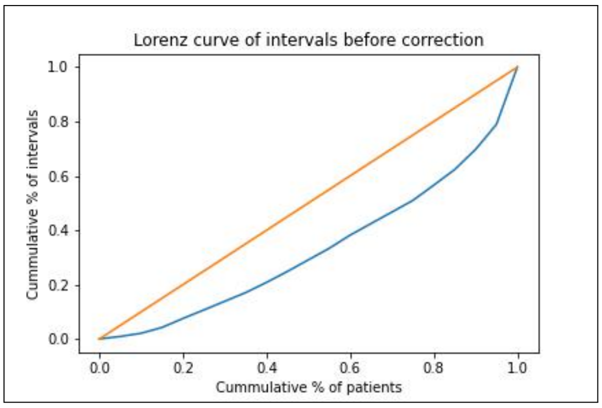
(b) Histogram of intervals per patient

Figure 3.3: Histogram of the intervals per recordings and patients after correction
plot created using the processed data

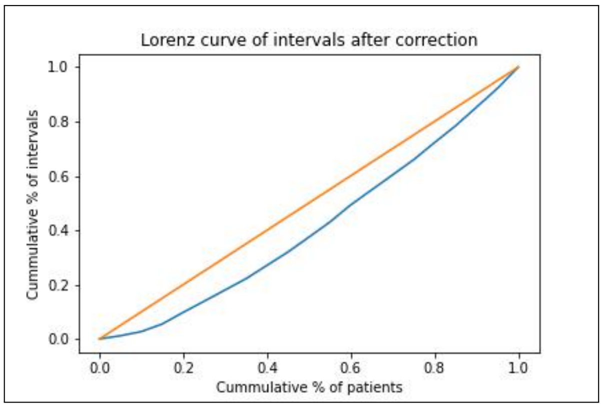
The left plot shows a histogram over the amount of intervals drawn from the different recordings. Now no more than 110 intervals is drawn from one recording. On the right is a histogram plotted where the data is aggregated per patient. Now no patient contribute more than 110 intervals. The mean and median of this data is 61 and 64 respectively. This indicates first of all that less intervals are drawn than before the correction (which had a mean of 74). But the data is also less skewed than before this correction. Using the data of intervals per patient, it can be found that the top 25 % of the patients (which totals 39 patients) contributed 32 % of the intervals. This is not perfect equality (which will be discussed later) but it is an improvement from before.

[3]https://github.com/marctimjen/Artefact-Rejection/blob/main/Preprocess/2%20series_dict.py

Lorenz Curves' has been plotted to illustrate how the equality has changed from before the correction to after:



(a) Lorenz Curve for the intervals per patient before the correction



(b) Lorenz Curve for the intervals per patient after the correction

Figure 3.4: Lorenz curve before and after the correction
plot created using the processed data

A Lorenz Curve have the cumulative % of the population of patients plotted on the x-axis. On the y-axis the cumulative % of contributed intervals is plotted. If perfect equality was achieved, every 10 % of patients would contribute 10 % of data. The diagonal line thus shows total equality. The distance between the blue curve and the perfect equality line indicates how equal/un-equal the distribution of intervals per population of patients. The closer the curve is to the diagonal line the more equal the distribution of intervals are. Plotting the Lorenz Curve before and after the correction will indicate how the equality has changed.
The left plot shows the Lorenz Curve before the correction, and the right plot shows a Lorenz Curve after the correction. It is noticeable, that with the correction more equality has been achieved.

Now changing the code above, from a value of 110 to 10 (very low value). The following Lorenz Curve is achieved:

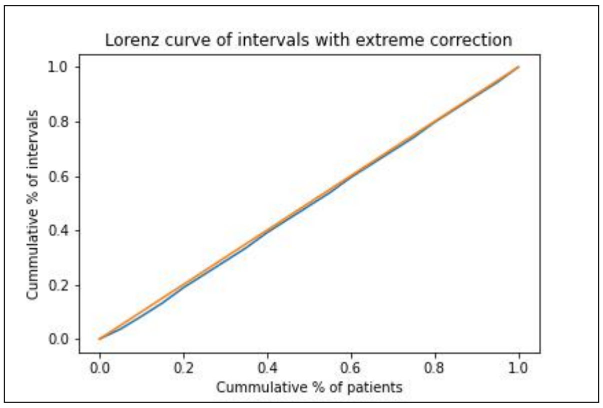


Figure 3.5: Lorenz curve with extreme correction
plot created using the processed data

Setting the number very low (as low as the patient with the fewest possible intervals or lower) near perfect equality can be achieved. This is shown in the plot above. One could think that this would be a perfect setting when loading the data. But the problem is that the data loader samples the same amount of intervals from the shorter recordings as from the longer recordings. This means that intervals from the short recordings would be more likely to overlap (in the EEG-recordings). This can be an issue, when making the model generalize to the data.

3.3 | Further data treatment

As noted in the introduction, the goal of this report is to train models that can distinguish artifacts from clean EEG-data. In this case the model should react on any type of artifact, whether it be for example a muscle or eye artifact. In this case the targets with binary indications will be used. To further develop this type of model, it will be trained to distinguish electrode artifacts from other types of artifacts. As mentioned in the introduction it is important to ensure good quality of the equipment used to collect EEG-data. This is to minimize the amount of artifacts created by the EEG measurement hardware.

To build this new model, a new labeling of target data is needed. In this case the clean EEG data is labeled 0, the other types of artifacts are labeled 1 and the electrode artifacts are labeled 2. Any labeling of electrode errors are put into this class of artifacts. This stem from the fact, that the model should be capable of annotating the electrode error no matter if another artifact is affecting the EEG recording.

To further investigate which types of artifacts the model can find, a multiple of classes will be created. In the following is the code to label the artifacts differently. Note this code is taken from the script ”0 Transform_data_cutoff_mclass.py” [4].

```python
elec_flag = re.search("elec", rows[3].lower()) or re.search("elpp", rows[3].lower())
if elec_flag: # test if string is in the elec class
    tar_anno = 2
elif rows[3].lower() == "musc":
    tar_anno = 3
elif rows[3].lower() == "eyem":
    tar_anno = 4
else:
    tar_anno = 1
```

It should be noted that the ”re” is an import of the regex package. Regex is used to make sure, that any type of electrode artifact annotation is caught. The targets label is saved in the variable tar_anno. It can also be seen, that a class for muscle and a class for eye movement artifacts are created. These are respectively labeled as 3 and 4. Any artifact that is not an electrode, muscle or eye movement artifact will be labeled with 1. This is a catch all class for the artifacts. It should be noted that even the eyem_muscle artifact is being labeled 1. The classes for muscle and eye movement artifacts are created because these artifacts are the most common (this can be seen in the table on page 4).

3.4 | The network architecture

The u-net architecture has achieved promising results on distinct biomedical segmentation applications [14, page 7]. The u-net is a fully convolutions network, and has the advantage that it does classification on localization [14]. This for instance mean, that the network can be trained to classify a certain class to the pixels of an image (also known as semantic segmentation).
In this paper the goal is also to annotate which locations (samples) of the EEG-signal that are contaminated with artifacts. The u-net was originally build for image input (that are 2 dimensional). In the task of artifact annotation, the input EEG-recording is of one dimension. Thus some slight modifications will be needed. Some inspiration for the implementation of u-net in pytorch has been taken from Nikolas Adaloglou [15].
The u-net architecture has a contracting phase, where the input is shrunken in the size of dimensions, but increased in the size of feature channels. Subsequently the network then has an expansive phase, where the data is increased in dimensions but decreased in numbers of feature channels. Additionally information from the expansion phase concatenated with a chopped version of the previous outputs from the shrinking phase [14]. The contracting and expansive phase are nearly opposites. When visualizing this, the network looks like an ”u”, hence the name u-net.

[4]https://github.com/marctimjen/Artefact-Rejection/blob/main/Preprocess/0%20Transform_data_cutoff_mclass.py

The u-net shape becomes apparent in the following figure:

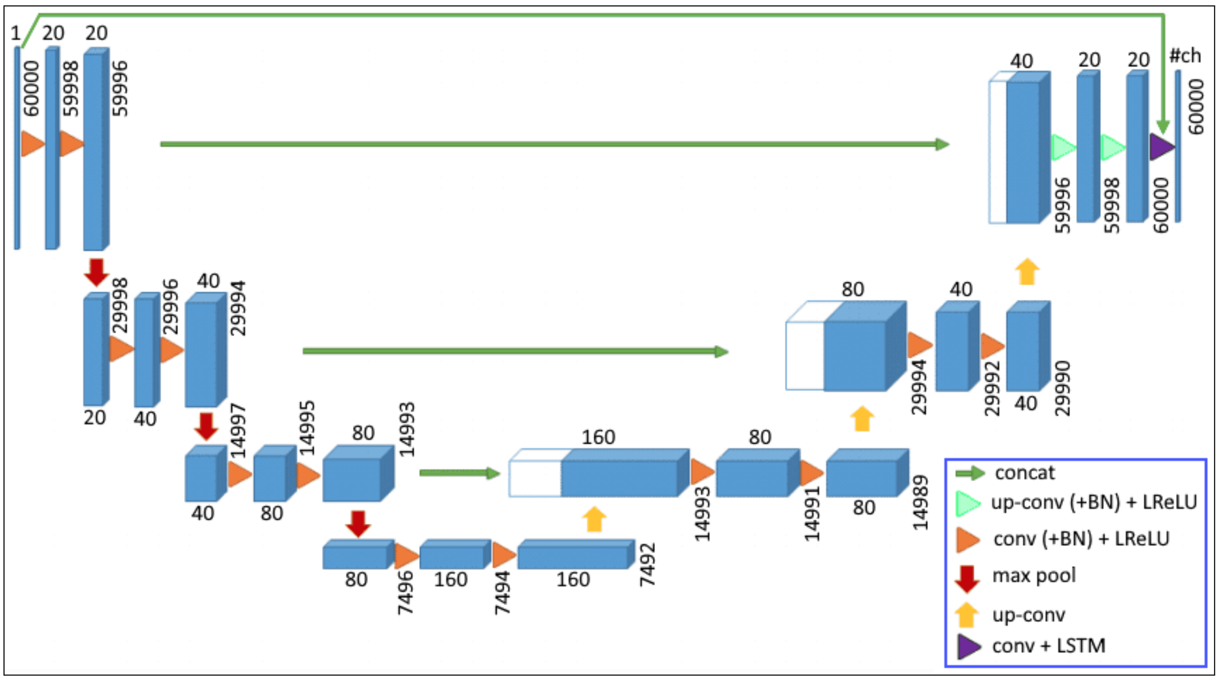


Figure 3.6: Plot of the network architecture
Augmented plot from Nikolas Adaloglou [15]

The plot above illustrate the architecture of the network that will be trained and tested in this report. It should be noted, that quite a few steps happens at the last arrow - both using a LSTM layer and concatenation. Note that the network takes an input of 5 minutes of EEG data. Thus the form of the input is (1, 60000) - 60000 comes from $5 * 60 * 200$. The size of the data is given as number of channels above or below the tensor, and the length given alongside the tensor. In the implementation of the network, unit tests have been written to ensure, that the correct shapes and values are returned.
The implementation of the network is done in blocks, since some parts of the network is repeated versions of itself. In the figure two orange arrows indicate a building block called "double convolution". In this block the data is fed though a convolutions layer then a batch normalization and lastly a LReLU. This is done twice. After each double convolution the data is saved (for concatenation later - see the green arrow). The double convolution is utilized in the block called "down scale". This block utilize max pooling using a kernel of 2 and then a double convolution. Since every two values are pooled into one the data length is halved. In the contracting phase of the network, a double convolution block and three down sample blocks are used.
For the expansion phase the usage of the "up scale" block is at play. This block contain more complicated code. Therefore the following is the python code used in the Up_Scale class. This class is depicted as one yellow arrow, one green arrow and a double convolution. This code can be seen in the script "Unet.py" [5]. Note comments and unit tests have been removed from the code. This is to make it more readable.

```python
class Up_Scale(nn.Module):
    # yellow arrow + green arrow + double_conv
    def __init__(self, in_channels, out_channels, up_conv = False):
        super().__init__()
        self.up = nn.Upsample(scale_factor=2, mode='nearest')
        self.up_conv1 = nn.ConvTranspose1d(in_channels, out_channels, kernel_size = 2)
        self.doub = Double_Convolution(in_channels, out_channels, up_conv)
    def forward(self, x, y):
        x = self.up(x)
        x = self.up_conv1(x)
        diffy = y.size()[2] - x.size()[2]
        x = F.pad(x, [diffy // 2, diffy - diffy // 2]) # make the dimentions fit
        x = torch.cat([y, x], dim=1)
        x = self.doub(x)
        return x
```

[5]https://github.com/marctimjen/Artefact-Rejection/blob/main/LoaderPACK/Unet.py

Note that the packages needed to create the building blocks and the network are torch, torch.nn called nn and torch.nn.functional called F.
The Up_Scale class use the __init__ clause to initialize the different functions needed. The up-sample function, one-dimensional transposed convolution and double convolution is initialized. The input to the function is the input channels, output channels and a boolean value indicating if the function should perform an up double convolution.
As an explanation of the code please look at Figure 3.6. The first usage of the Up_Scale would be at the lowest point (of the U) at the tensor of (160, 7492). The Up_Scale class would return the tensor of (80, 14989). The input to the class would be in_channels equal to 160, out_channels equal to 80 and up_conv set as False. The input to the forward function would be the tensor of (160, 7492) as the x-value and the tensor of (80, 14989) as the y-value.
Firstly the Up_Scale would take the x-value tensor of (160, 7492) and use the upsample function. This function works by posting the values twice in the output. Eg. $upsample([[[1, 2, 3]]]) = [[[1, 1, 2, 2, 3, 3]]]$. This would scale the tensor to (160, 14984). Then the transposed convolution is used with kernel size 2, and the number of input channels is set to 160 and output channels to 80. This would create an additional value and it would half the number of channels to 80. Therefore the new size of the x-value tensor is (80, 14985).
To concatenate the x-value with the y-value tensor of size (80, 14989), additional padding is needed. Thus the value diffy is calculated to $14993 - 14985 = 8$. These 8 additional values are padded with 4 values on either end of the data-series of the x-value tensor.
Now the x-value and y-value can be concatenated, and this result in a tensor with shape (160, 149993). This is the tensor after the yellow arrow in the figure. The Up_Scale function then use the double convolution and the resulting tensor is of size (80, 14989). See the box after two orange arrows.

In the original u-net paper, the output of the final annotations are smaller in the dimensions width and height than the input image [14]. This can easily work for different tasks. But in this report it is opted for an equal length of annotations and targets. Therefore two transposed convolutions are used in the final steps of the network. The up_conv value indicate if this should be activated. This option is only activated in the last use of the Up_Scale class.
In the very last layer, a convolution with 1 output channel is used. In the original u-net, this last convolution could be tuned with the amount of output channels. This would be used to return a one-hot encoded like output with the number of channels equal to the amount of classes that was to be annotated [14]. In this report only one feature channel is used. This is to gain the most compact version of the data, before it can be concatenated with a the input to the network. Then this data with two channels is fed to a LSTM cell. The bidirectional setting is set as true. This means that the LSTM cell go sequentially trough the data in both direction. The num_layers is to be 1, this means that only one LSTM cell is used. The value of hidden_size is set to 5. The hidden_size indicate the amount of features in the hidden state. The input_size is set to 2, since two time-series is given to the LSTM network. The proj_size parameter is set to 1, this is to project the data into 1 series. Since the bidirectional is set true, a total of 2 series is produced. If the projection is not used a total of 10 series would be returned. To make the network return the correct states and values, further the data might be altered additionally. The proj_size is going to be changed in accordance with how many different classes the network should annotate. Lastly the information from the LSTM cell is run trough the soft-max function. This will return pseudo-probabilities for which class the network believe the samples to belong to.

3.4.1 | Explanation of the settings in the network

In the network made in this report 1 dimensional convolutions are performed. In the double convolutions a kernel size of 3 is used. A kernel size of 3 is also what the original u-net report used [14]. Another study by Lopes, Leal, Medeiros, Pinto, Dourado, Dümpelmann and Teixeira where the goal was to de-noise EEG-signals found that a kernel size of 3 was as good as kernel size of 5 [4, page 4].
In the original u-net paper the ReLU (Rectified Linear Unit) activation function was used [14]. The ReLU activation function is a combination of two linear functions. The first returns 0 if the input is less than or equal to 0. And the second return the input value if the value is larger than 0. Apart from its simplicity and fast computation it also has benefits because it is non-saturated [16, 4]. It should namely increase the convergence speed of the network [16, page 1]. One of the issues regarding the ReLU function is that it turns every negative value to 0. This mean that for different negative inputs it will return a zero value. Therefore this activation function might crate so called ”dead neurons” in the network [4]. To try and

combat this issue, and keep the benefits of ReLU, another function called LReLU is suggested. This function is called leaky ReLU, which introduce a negative slope. Mathematically it can be written as:

$$y_i = \begin{cases} x_i & if\ x_i \geq 0 \\ \frac{x_i}{\alpha} & if\ x_i < 0 \end{cases}$$

The value of α is fixed and should be in the range of $(1, +\infty)$ [16, page 1 and 2]. Xu, Wang, Chen and Li has tested the LReLU function and found that the LReLU outperformed ReLU. They got good results using α equal to 5.5 [16]. As a result of this a α value of 5.5 will be used in the network created in this report.

A LSTM-cell has been chosen because of its promising abilities in sequence labeling tasks [17, page 1]. This module does also have some nice properties that the convolutional neural network does not have. This will be elaborated upon in the discussion Chapter. Kim and Keene found that the LSTM unit was the best at detecting artifacts [8, page 205].

The number of feature channels in the network is set to double in every down scale and subsequently to be halved in each up scale. This is how the behavior is set in the original u-net [14]. So the amount of feature channels chosen in the first double convolution will determine how many feature channels are used though-out the network. Brief tests using 10, 20 and 64 feature channels in the first double convolution was conducted. It seemed that the network of 10 feature channels where to shallow and the network of 64 where to big. Thus the final setting of the feature channels where 20 at the first double convolution. The main concerns of choosing the size of the network would be that a to large network would tend to over-fit the training data, and a to shallow network would never learn the features of the data [18].

3.5 | The optimizers and loss function used for training

In this report the optimizers stochastic gradient descent (SGD) and Adam will be tested. The optimizer SGD is used to update the weights (θ) of the network. This happens using the formula:

$$\theta^t = \theta^{t-1} - \epsilon_t \cdot \frac{\partial L}{\partial \theta}$$

The learning rate parameter (ϵ_t) is the most important parameter to tune [19, page 1]. Using a learning rate that is to low can lead to slow learning of the network, and a to large learning rate can result in divergence of the network [19, page 1].
SGD provides a few other parameters that can be tuned namely the momentum and weight decay. The use of momentum is to hopefully speed up the convergence of the network when using SGD [7, Chapter 8].
The Adam optimizer use whats called adaptive moments (thus the name). The Adam optimizer can be seen as a combination of RMSProp and momentum [7, Chapter 8]. The Adam optimizer is seen as robust to the setting of hyper-parameters. The default learning rate parameter of Adam might need to be changed to gain optimal training [7, Chapter 8]. Therefore the learning rate and weight decay will be optimized according to subsection 3.6.
The loss function crossentropyloss is used [20]. This function can be used when one deals with a classification problem (which annotation is about). The weight parameter is useful when dealing with the imbalance of the classes in the data set [20].

3.6 | Hyper-parameter tuning

The network presented contain convolutions. Therefore the cyclical learning rate methods explained by Smith might be beneficial [19, page 10]. Smith argue that these methods allow for less guesswork when tuning the learning rate parameter. Smith suggest using the cyclical learning rate scheduler. This type of scheduler makes the learning rate increase and decrease over the training of the network. This results in an interesting phenomenon: The rise of the learning rate has an overall benefit to the training of the network. Even though the large learning rate might cause temporary harm to the performance of the network [19, page 2].

A visual representation of the simple cyclical learning rate can be seen here:

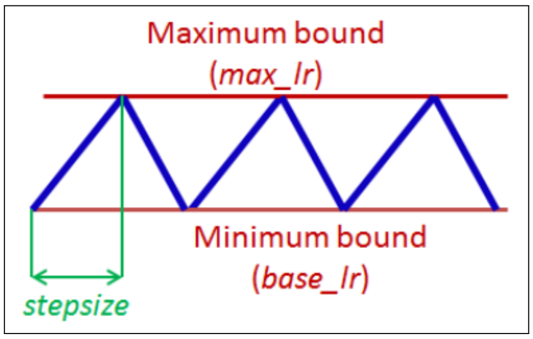


Figure 3.7: Plot of the simple cyclical learning rate scheduler
Plot from Smith [19, page 3]

Even though three new parameters should be optimized, these can easily be found according to Smith. To estimate the parameter of maximum bound (also called max_lr) and the minimum bound (called base_lr) a learning rate test is conducted. The learning rate test suggested by Smith, is to linearly increase the learning rate from a small to a rather large value. If one has already tested models on the data, one should have an rough idea of what these values would be. Otherwise Smith suggest trying the ranges of 0.0001-0.01 and 0.01-0.1 [19, page 5]. These learning rate tests are to be done over a few epochs (4-8) [19, page 5].
The learning rate should be updated after each batch of the training data. Validation of the network should also be performed during training of the network. After the short learning rate test, one should plot the validation accuracy against the learning rate [19, page 5].

A few example of learning rate test can be seen in the plot below:

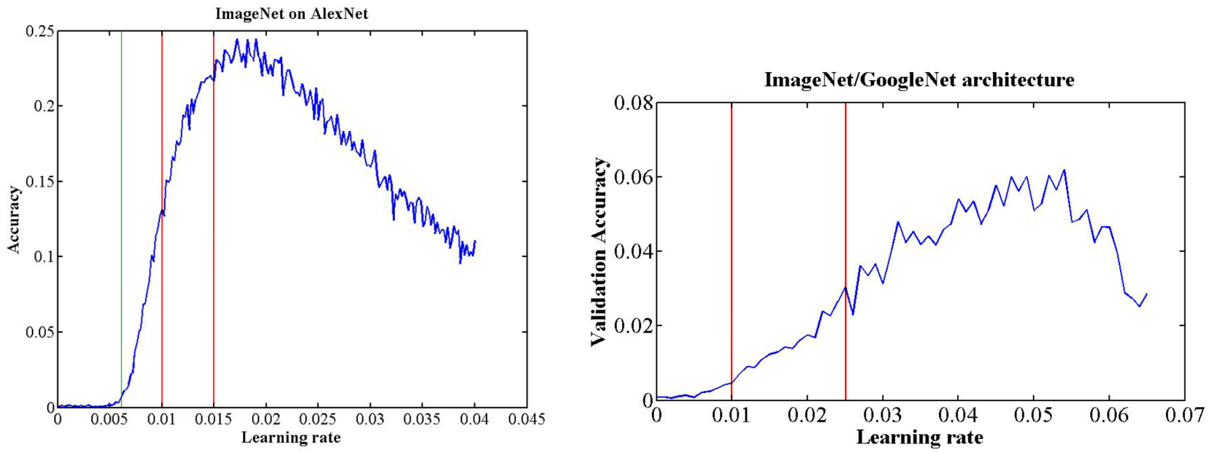


Figure 3.8: Examples of learning rate tests
Augmented plots from Smith [19, page 8 and 9]

It can be seen that when the network starts training, the small learning rate makes no significant improvement of the model. When the learning-rate has increased enough to start the training of the model, then the validation accuracy will begin to increase. At this learning rate the model is trainable. The momentum of the validation accuracy will eventually start to stall, become ragged or starts to decrease. At this point a good maximum learning rate value has been found [19, page 5]. In the graphs above, the two red lines indicate the learning rate boundaries for the cyclical learning rate scheduler. The plot on the left illustrates training on the AlexNet and on the right plot training on GoogLeNet. Both these nets are used for image classification [19].

The red lines indicate the lower and upper bound settings for the cyclical learning rate scheduler. These lines are set by Smith. On the left plot Smith argue that an learning rate of 0.06 (the green line) is a

possible base_lr learning rate. Though he remarks that the majority of the apparent improvement in accuracy would come from the lower end of the learning rate [19, page 7]. Also this stem from the fact that a large learning rate is a form of regularization. Smith remarks that reducing other types of regularization in favor of larger learning rates should make the training of the model significantly more efficient [21, page 7].
Smith suggests using a step-size of 2-8 epochs [19, page 4]. The tests by Smith showed that 3 cycles would train the model most of the way, but more cycles would further increase the performance [22, page 3].
To estimate the base_lr and max_lr, Dehaene suggest using a learning rate test where the learning rate is increased exponentially [23]. Dehaene suggest running the network though two training epochs. After each batch the loss and learning rate should be saved and the gradients should be updated [23]. When optimizing for the best learning rate, Dehaene suggests using the smoothed training loss against the log of the learning rate. The loss is smoothed using the equation $smoothed\_loss = \alpha \cdot loss + (1 - \alpha) \cdot previous\_loss$ [23]. Looking at this plot, one should find the lowest point, and divide the value by 10. This should be a good upper-bound. The lower bound is found by dividing the upper bound by 6 [23]. This can also be seen in the following plot:

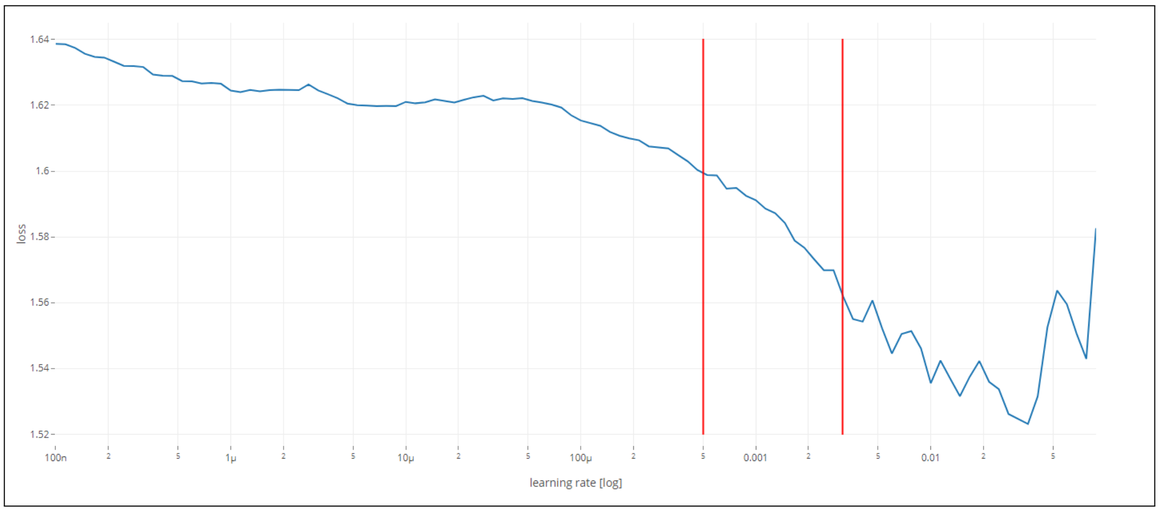


Figure 3.9: Examples of exponential learning rate test
Augmented plot from Dehaene [23]

The lowest point on the smoothed loss curve was determined to be 3e-2, then the max_lr is set to be 3e-3, and thus the base_lr become 5e-4. Dehaene tested the Adam optimizer (with a constant learning rate) and the SGD with a cyclical learning rate (CLR) with the parameters found before. The conclusion was that Adam was constantly good and that SGD with CLR achieved much faster convergence [23]. The data in this report is drawn at random when training the models. Therefore the smoothed loss for the hyper parameter tuning is found using a validation set (where the same samples are drawn).

The experiments by Smith have shown that a constant weight decay should be used throughout the training [21, page 11]. The data set in this report is rather complex since it is made up of recorded EEG-data (please refer to section 2). For complex data Smith argue that less regularization is needed, and thus lower weight decay values [21, page 14]. In this report the weight decay values 0, 1e-4 and 1e-5 will be tested. According to Smith a simple grid search is enough and differences between the weight decay values will be visible on the validation loss curve [21, page 11].
When increasing the learning rate, a decreasing momentum will make the training with the higher learning rate more stable [21, page 10]. This thus allow for larger learning rates [21, page 10]. In this report cyclical momentum that decrease while the learning rate increase will be tested. No momentum will also be tested. The range for the momentum will be 0.9 to 0.99 and 0.8 to 0.9 that is default for the pytorch function [24]. In this report the parameters are set one at a time, but according to Smith some parameters are possible to determine simultaneously [21, page 12].
Smith argue that the use of larger batch size, adds stability to the training. Therefore a large batch size allow for larger learning rates [21, page 14]. Thus in this report, the largest batch size the GPU hardware permits will be used.

4 | Results

4.1 | Logs of network training

The software "Neptune.AI" has been used to log information during the training, learning-rate/momentum/weight-decay tests of the networks. The logs can be accessed using this link:[6]. Each log is for one initialization and training/testing of the network. The number in the log sequence is an unique identifier for each log. This identifier is incremented for each run of the network.

4.2 | The models that are going to be trained and tested

In this report different models will be trained and tested. The first model is a naive model, that use a threshold value to annotate artifacts. The network in the methods chapter will also be used. The network will be used to do binary classification of the artifacts (0 = clean EEG data and 1 = artifact). This model will be named the "binary model". A multi class of the network is also trained and tested. This model should annotate the data as: 0 = clean EEG, 1 = normal artifact, 2 = electrode artifact.

4.3 | Testing of the naive model

This naive model builds on the idea that artifacts are large in amplitude. This model is to produce outputs between 0-1. The model predicts close to 1 if it believes an artifact is present, and close to 0 otherwise. The model works by taking the absolute value of the EEG data. This result in the EEG data being bounded from 0 to 200. The model then multiply the data by $\frac{1}{200}$. The model is tested on the validation data set using a threshold of 0.5. The outputs from the model is saved in two histograms. One histogram for when an artifact is present, and one for when no artifact is present. The amount of bins in the histograms is 1000. In the following the two histograms are plotted:

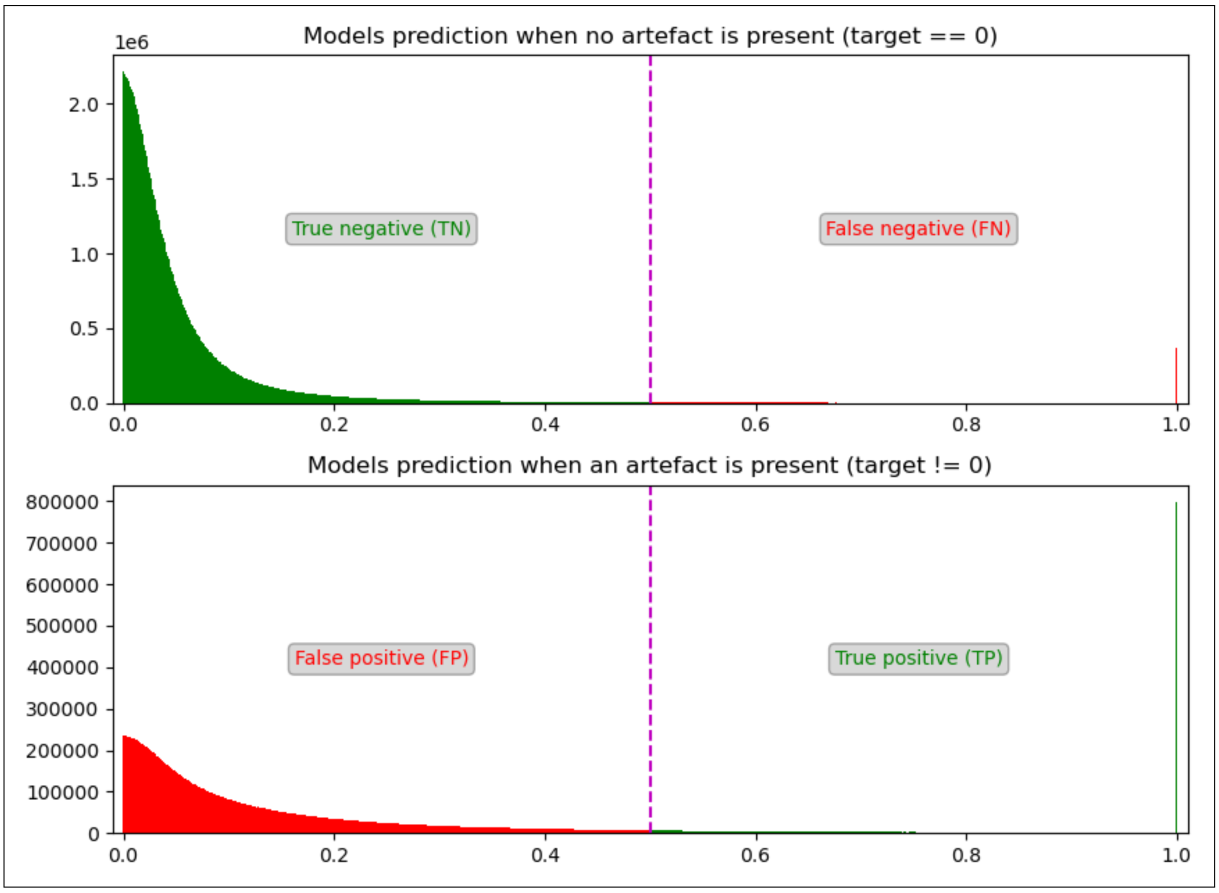


Figure 4.1: Histograms over naive model predictions
Plot made using the validation data set

[6] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/experiments?split=tbl&dash=charts&viewId=standard-view

Note the true positives denote when the model correctly identify an artifact. Setting the threshold value equal 0.5 means that the model classify any sample above 100 µV (in amplitude) as an artifact. The threshold is marked as the magenta line in the plot above. As can be seen by the plot, the model find a lot of the true negatives. Therefore the model correctly classify a lot of the clean EEG data. But because of the assumptions of the model, it does not find any artifacts below 100 µV in amplitude. The area of interest (for further EEG analysis) is in this case set to be below 100 µV in amplitude. As mentioned in the theory chapter, the issue of artifact annotation is that the artifacts can lie in the area of interest. This plot also display that some of the EEG data above 199.8 µV is not marked as artifacts. This will be investigated later.
This model is tested on the test-set and the following confusion matrices can be made:

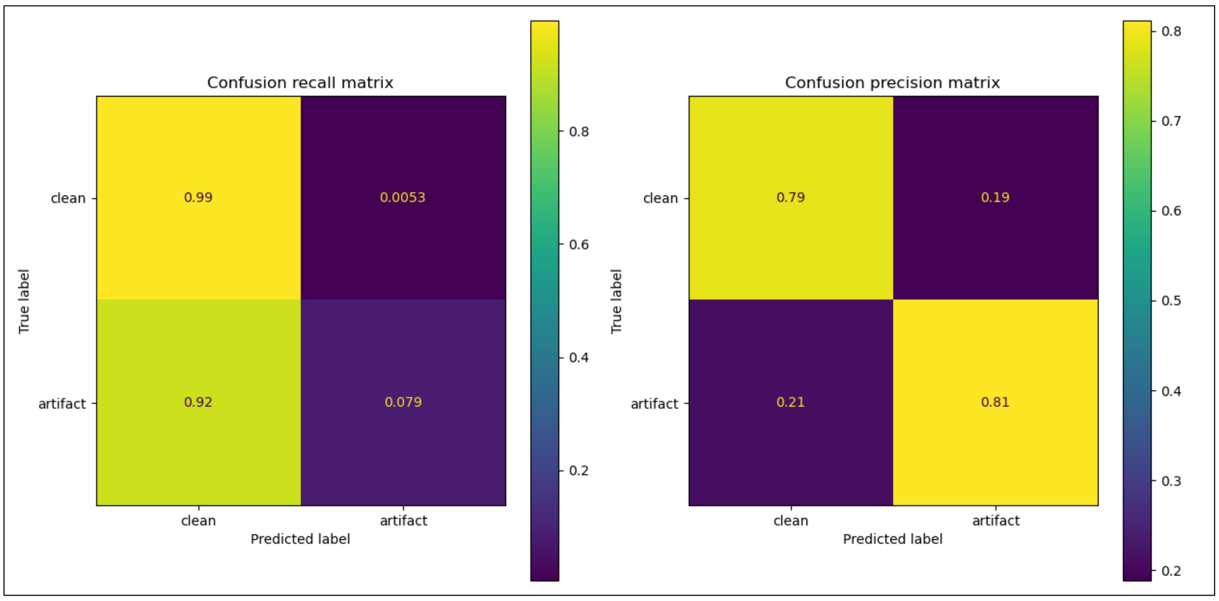


Figure 4.2: Confusion matrices for the naive model
Plot made using the test data set

From these matrices it can be seen that the naive model has high precision (79 % and 81 %) when annotating the clean and artifact data. But the model suffer from the fact that it only recall about 7.9 % of the artifacts. The model recall a lot of the clean EEG-data, this is about 99 % of the samples. It should be noted, that moving the threshold of the model will change the results. A lower threshold will allow the model to recall more artifacts (higher true positive), but at a cost of the model predicting more false negatives too.

4.4 | Investigating the quality of the data

During the testing of the naive model it was found that some clean EEG data had an amplitude above 199.8 µV. In the following plot EEG data with large amplitude and annotations is shown.

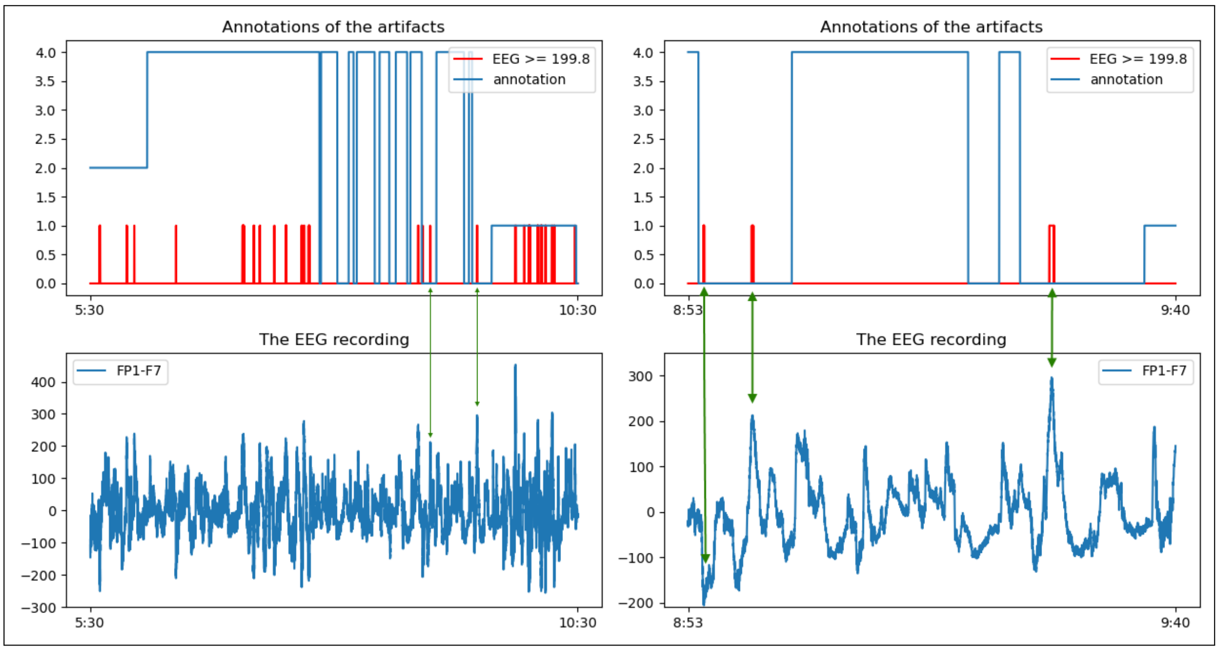


Figure 4.3: Plot of EEG data and the associated annotations
Plot made using TUH EEG data

In the plot the artifact annotations are marked in the top row and the EEG signals are plotted in the bottom row. Note that the right plot is a zoomed in version of the left plot. The different values (0, 1, 2 and 3) on the top row of plots denote different types of artifacts (for further information please refer to the methods chapter). In the annotations plots, the blue line indicates which artifacts are annotated, and the red line indicate if the underlying EEG signal has an absolute amplitude above 199.8 µV. The green arrows indicate some of the areas, where the underlying signal is above 199.8 µV and yet no artifact is annotated.
In the plot below a "clean" EEG signal is plotted. This is for the same deviation and from the same person, just from a different recording.

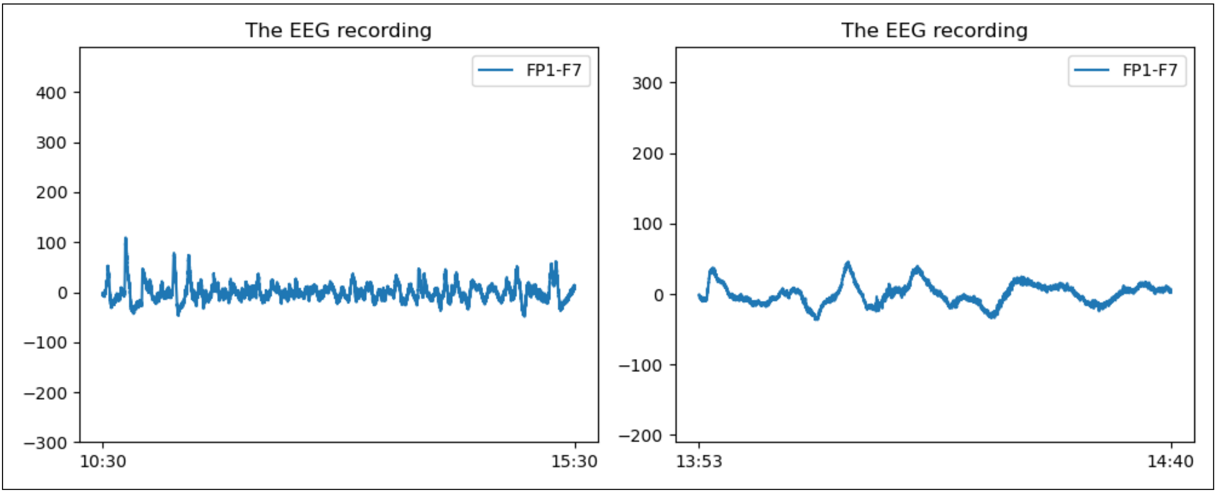


Figure 4.4: Plot of clean EEG data
Plot made using TUH EEG data

In the plot the same x- & y-axis length is used as in Figure 4.3. It is clear that this plot show way less volatility and a more "clean" time series. These signals are also well in the bounds of [-150 µV, 150 µV]. As to why the signals above 199.8 µV is not annotated as artifacts is unclear. No additional information is given for this specific recording. EEG signals with absolute value above 199.8 µV (and yet no artifact is marked) also happen in other EEG recordings. If these annotations are wrong, the evaluation and training of the model will be impacted.

4.5 | Training of the binary networks

In the training of the networks it was found that a batch size of 10 is optimal. It should be noted that the training of the different networks happens on a GPU-cluster. Therefore the models were trained on different GPUs, that had different amount of memory available (which is the determining factor of the batch_size). A batch size of 10 was found to fit all GPUs, therefore this was used.
If the hyper-parameters are way off, it will show early on in the training process according to Smith [21]. Therefore the hyper parameters are tested using the network, but with less training and validation data. This will make the testing process quicker. The training data used is randomly drawn. The network is tested on the same validation data.
The weighting of the cross-entropy loss is found. This is done with the script: "weights_in_crossentropyloss.py" [7]. With the script it is found that on average every 5'th sample is annotated as an artifact. Therefore the weights are set as [1., 5.]. This result in a higher loss for false positives than false negatives. The learning rate test can now be conducted for the networks that use SGD and Adam as optimizers. The plot below show the results of the linear learning rate tests:

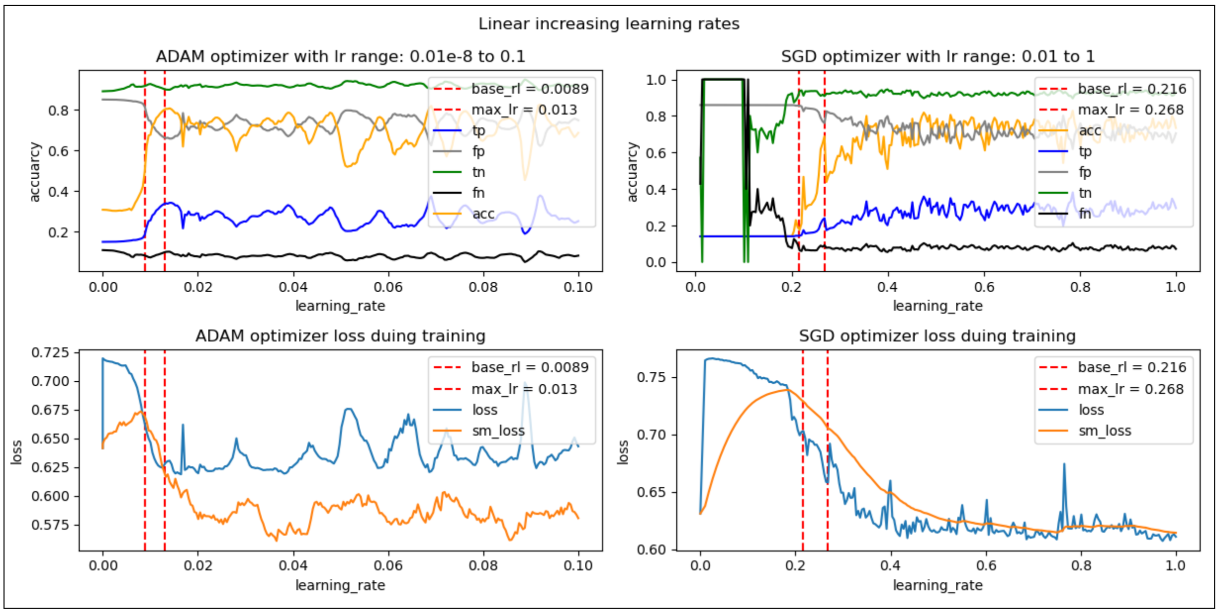


Figure 4.5: Linear learning rate test for the binary network
Plot made using Neptune data

For the learning rate tests, the top row will contain validation accuracy and the positive and negative recall. Here a true positive refer to the model correctly recalling an artifact. The bottom row of plots display the validation loss (sm_loss is the smoothed loss). In the plots the red lines indicates the bounds that has been found. For the linear test this follows Smith's instructions. The plot show the boundaries in the label boxes. The boundaries for the Adam optimizer has been set as: base_lr = 0.0089 and max_lr = 0.013. The boundaries for the SGD optimizer has been set as: base_lr = 0.216 and max_lr = 0.268. The following plots shows the results of the exponential learning rate tests:

[7] https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/1.%20hyperoptimize/weights_in_crossentropyloss.py

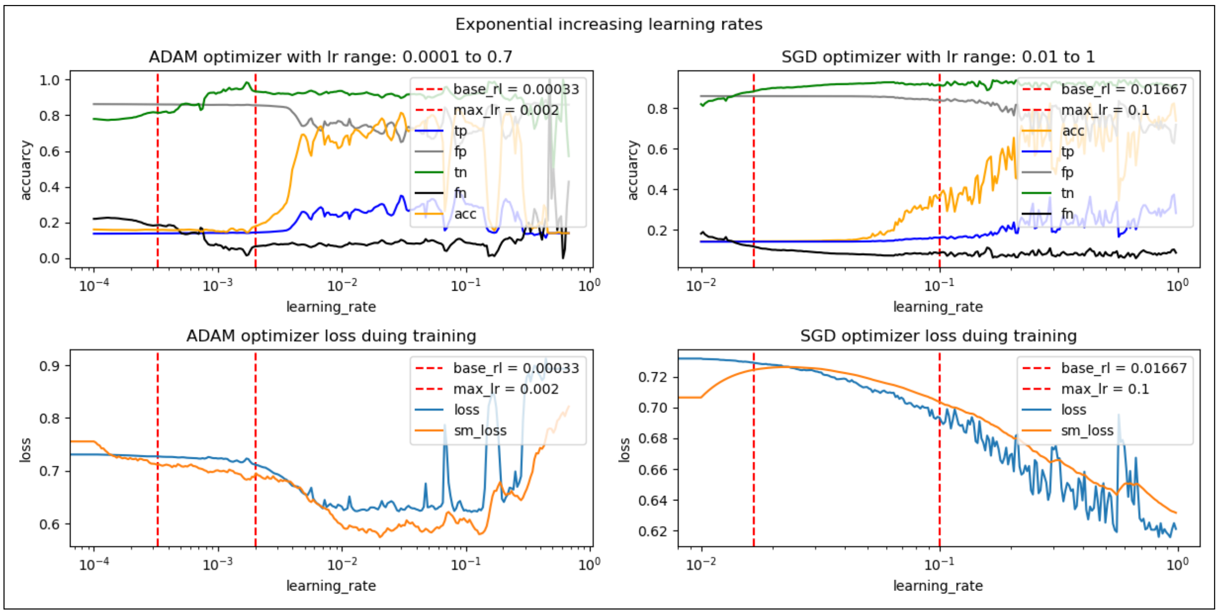


Figure 4.6: Exponential learning rate test for the binary networks
Plots made using Neptune data

In this case the upper and lower bounds are found according to the instructions by Dehaene. For the Adam optimizer the lowest learning rate point for the smoothed loss is 0.02 according to the plot. This results in a max_lr of 0.002 and a base_lr of 0.00033. For the SGD optimizer the lowest point is at 1, thus max_lr = 0.1 and base_lr = 0.01667. Note that this is the edge of the learning rate range of this test. If the test was conducted with a larger learning rate range, the smoothed loss might have a minimum at a even higher learning rate. Though this was not assumed to be the case. The learning also seemed to be more volatile towards the larger learning rates in the test. Therefore a larger learning rate might not be preferred.

4.5.1 | Results from the network with Adam optimizer

The linear learning rate test resulted in base_lr equal 0.0089 and a max_lr equal 0.013. Using this range the hyper-parameter of weight decay is found:

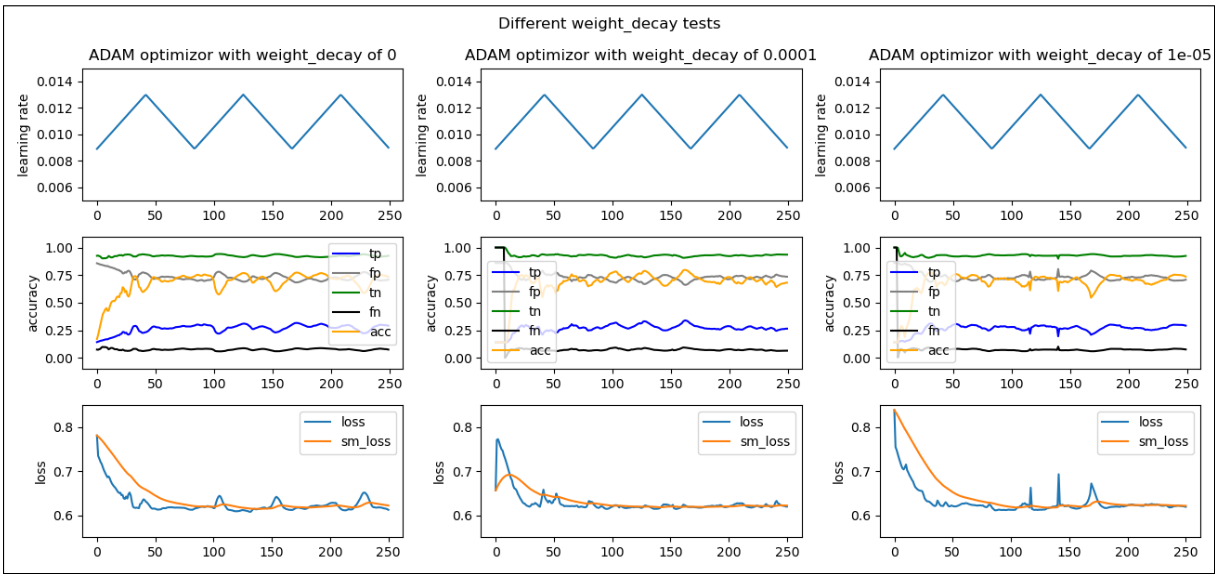


Figure 4.7: Weight decay tests for the Adam network
Plot made using Neptune data

These tests indicate that A weight decay of 0.0001 result in the network having a stable and good convergence during training. This network was trained using these parameters for a 100 epochs and with 10 cycles using the cyclical learning rate scheduler. The network showed fast convergence, and fewer epochs/cycles could probably have been used. The neptune logs can be found at [8]. The final train accuracy came out to be 83 % and the validation accuracy resulted in 75 %. The network does not seem to over-fit the training data (please refer to Appendix B). The histogram testing script was used:

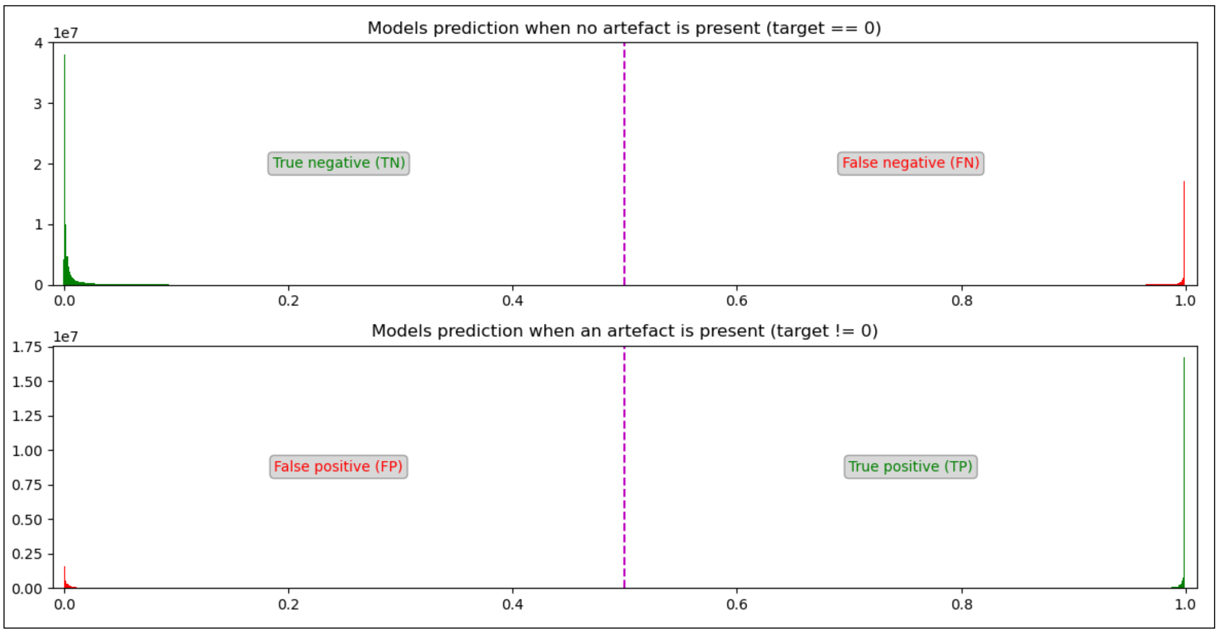


Figure 4.8: Histogram made using the Adam network
Plot made using the validation data set

It can be seen from this plot that the network tried to minimize the loss. This is done by separating the probabilities as much as possible. Thus a lot of values end up at 0 and 1. Therefore setting the threshold higher or lower, is not going to change many of the models annotations.
The following plot shows how the Adam network annotates artifacts.

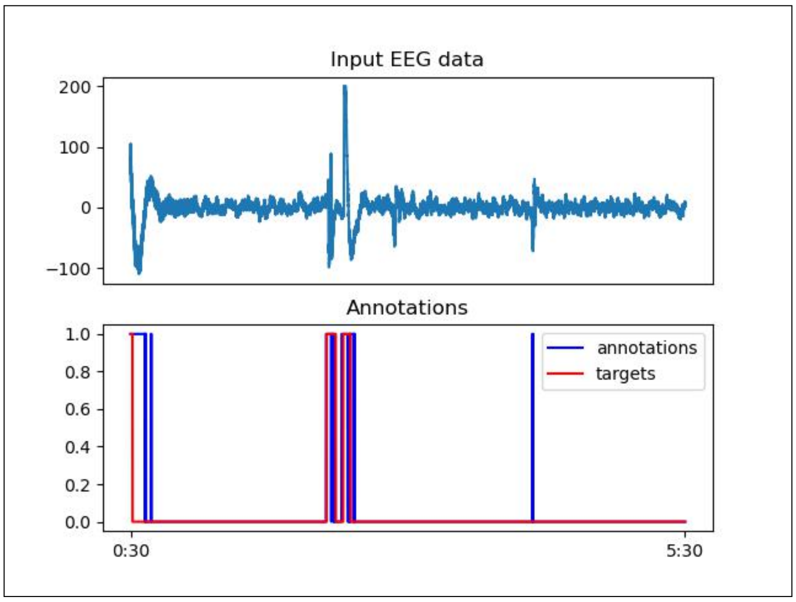


Figure 4.9: Plot of the annotations by the Adam network
Plot made using the validation data set

[8]https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-470/all

When the red line hits the value 1 an artifact is present in the EEG data. The blue line indicates the Adam network's predictions. From the plot it can also be seen, that the network can annotate EEG-data as artifacts when the amplitude of the signal is in the range of +/- 100 µV.
This network was used to annotate the test-data. The results can be summarized in the confusion matrices:

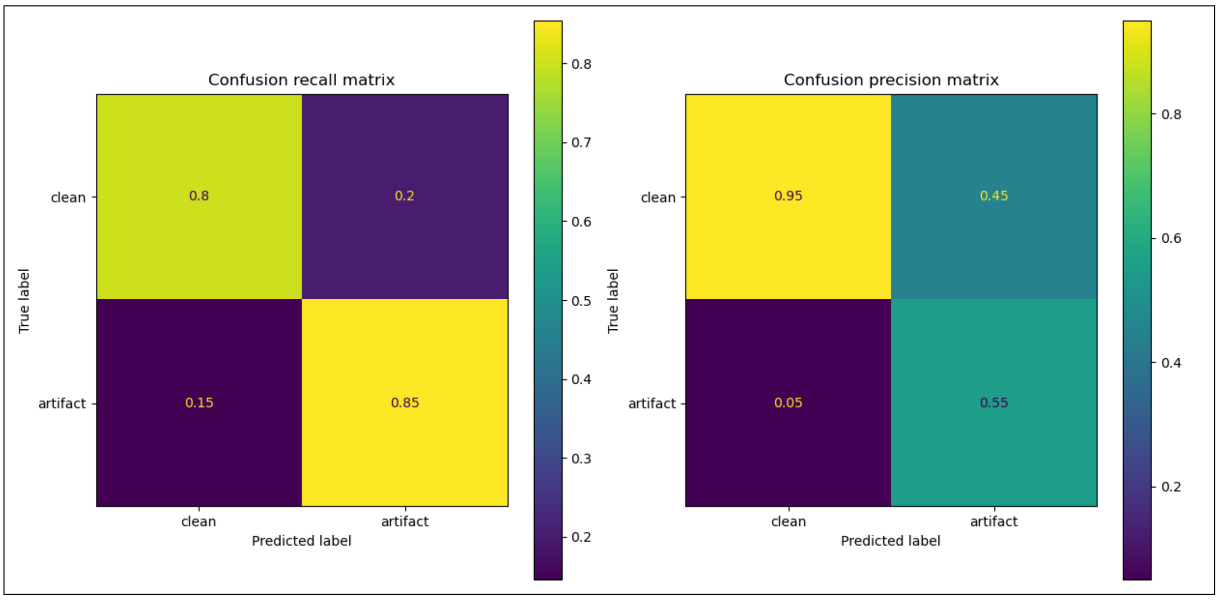


Figure 4.10: Confusion matrices for the Adam network
Plots made using the test data set

The test-result indicates that the network can recall about 85 % of the artifacts in the test data set. The network does this with a precision of 55 %. With this configuration the network recall about 80 % of all the clean EEG samples, and the network does this with a 95 % precision. The resulting test accuracy of this network is 81.3 %. For more in-depth comparison of the models please refer to subsubsection 4.5.3.

4.5.2 | Results from the network with SGD optimizer

The network using SGD as optimizer was trained with a learning rate range of (0.216, 268). This was found in the the linear learning rate test. The other parameters resulted in a cyclical momentum from 0.8 to 0.9 and a weight decay of 0. This network initially showed good training behavior. But at epoch 21 the network diverged and started to predict all samples as artifacts. This can be seen in the neptune log[9]. The increasing and decreasing learning rate could not make the network predict otherwise. The network was then trained with a modification of the results from the exponential learning rate test. Even though the maximum learning rate could have turned out a little larger, the training seemed to become unstable. The previous result indicated that a to large learning rate might cause to much harm in the training (please refer to subsection 3.6). In this training the base_lr was set to 0.07 and the max_lr to 0.103. The reason for a higher base_lr was that the loss and accuracy showed slow convergence at these low values (please refer to Figure 4.5).

[9]https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-490/all?path=network_SGD%2F&attribute=val_acc_pr_file

Cyclical momentum was then tested and can be seen in the plot:

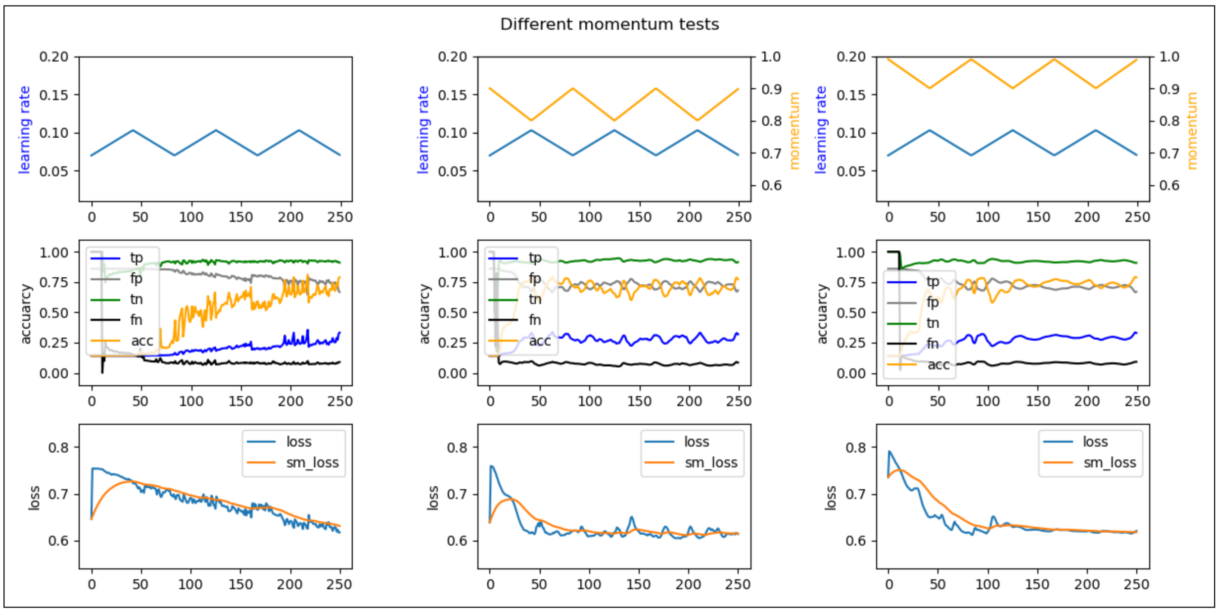


Figure 4.11: Momentum tests of the SGD network
Plot made using Neptune data

When determining the value of cyclical momentum a few tests of the network are performed. The network is tested using no momentum, using a momentum range of (0.8 - 0.9) and a range of (0.9 - 0.99). When finding the right parameter, one look at the stability (volatility) and convergence (how fast the loss drop) of the networks. The conclusion from the tests is that the training with momentum (0.9 - 0.99) was the most successful.
After implementing this result in the training, a constant weight decay can be tested:

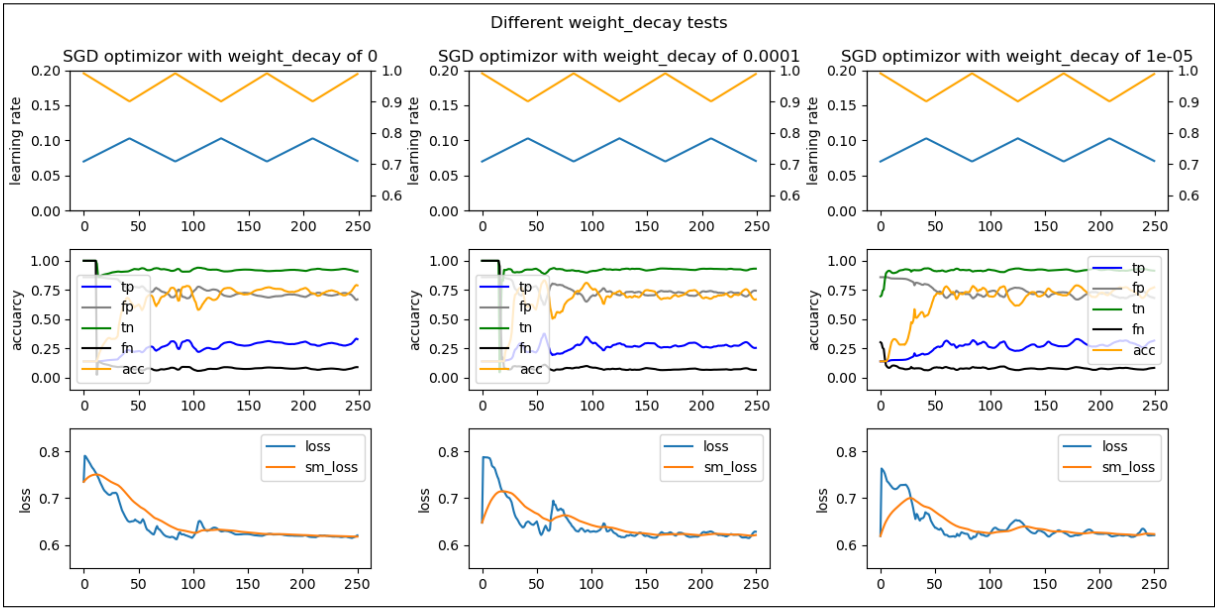


Figure 4.12: Weight decay tests of the SGD network
Plot made using Neptune data

When the momentum and learning rate have been set, the weight decay can be tested. The same criteria as above is used to determine the weight decay. In this case no weight decay seem to be the most optimal solution. With these settings the network is trained for a longer time with more data. The resulting network can be viewed here: [10]. This network showed promising learning at the beginning. Though it hit a major drawback around the peak of the second cycle. The network never got the loss lower than at epoch 15. The training and test accuracy at this epoch can be found to be 0.78 and 0.77 respectively. The version of the network at epoch 15 was thus used in the pipeline to annotate the test-EEG data. Comparing the targets with the annotated data the following confusion matrices were achieved:

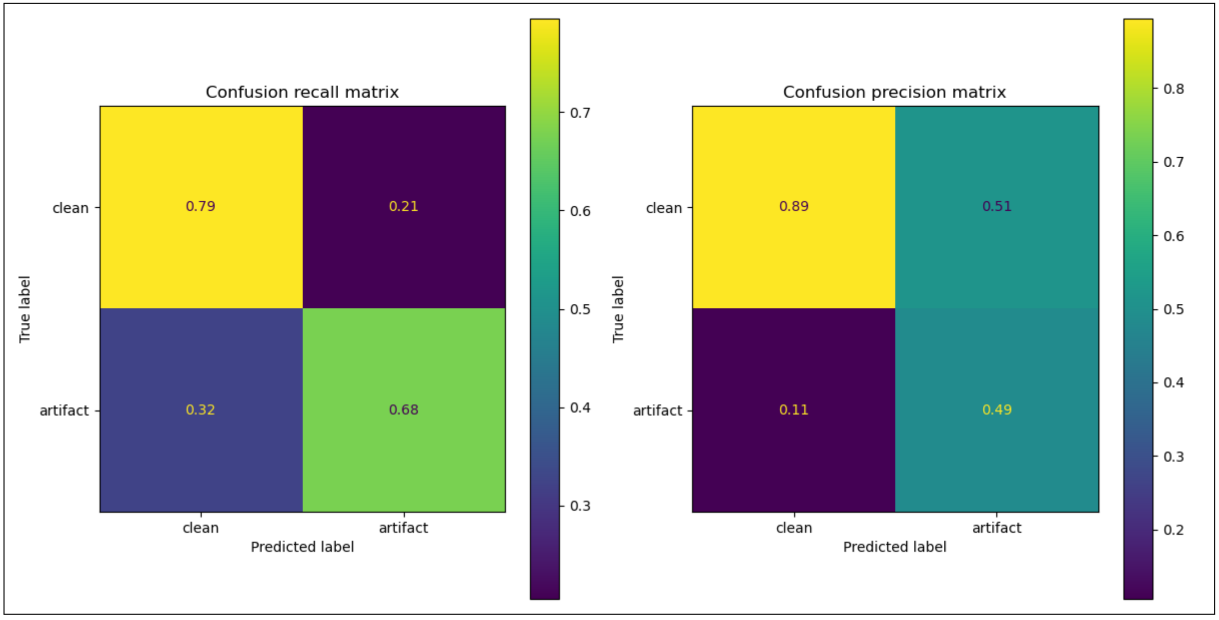


Figure 4.13: Confusion matrices for the SGD network
Plots made using the test data set

It can be concluded that this network can predict an artifact with 68 % precision and recall about 49 % of the artifacts in the test data. The network can also distinguish about 79 % of the clean data from the unclean data. With this precision the network recalls about 89 % of the clean EEG samples. The final precision of this network was found to be 76.8 %. To better compare models further tests of the networks can be seen at subsubsection 4.5.3.

4.5.3 | Results from the binary models

The following tables contain results from the evaluation of the models utilizing the test set. These tables can be used to investigate which types of artifacts the models is successful in annotating. Note that the "True Positive" (tp) refer to the class of interest. E.g. a tp of "No artifact" would be a correctly labeled "clean EEG" samples.

[10] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-525/all?path=.

Networks	No artifacts				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Linear model	0.790	0.789	0.812	0.995	0.079
Network using SGD	0.768	0.895	0.487	0.795	0.676
Network using Adam	0.813	0.950	0.553	0.800	0.855

Networks	All artifacts				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Linear model	0.933	0.110	0.951	0.048	0.980
Network using SGD	0.703	0.103	0.973	0.635	0.707
Network using Adam	0.692	0.128	0.991	0.878	0.682

Networks	Elec				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Linear model	0.921	0.533	0.930	0.145	0.989
Network using SGD	0.727	0.190	0.969	0.735	0.727
Network using Adam	0.713	0.201	0.984	0.866	0.699

Networks	Musc				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Linear model	0.915	0.116	0.933	0.037	0.980
Network using SGD	0.716	0.153	0.970	0.698	0.718
Network using Adam	0.708	0.176	0.990	0.900	0.694

Networks	Eyem				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Linear model	0.955	0.053	0.975	0.046	0.979
Network using SGD	0.690	0.042	0.982	0.510	0.695
Network using Adam	0.661	0.047	0.986	0.648	0.662

Based on the tables it can be concluded that the networks are better at recalling the artifacts. This can be seen by the higher "True negative recall" in the "No artifacts" table. Though the network model sacrifice some precision when finding more artifacts (some of which are false positives). This can be seen in the "True negative precision" column in the "No artifacts" table. It would be preferred to see an increase in both recall and precision. The network that use Adam is better at recalling the different types of artifacts. This network does not lose precision as a trade off (compared to the network using SGD).
The macro-average F1 scores have been found for the different models using the test-data set. The linear model achieved 0.643, the SGD network achieved 0.712 and the Adam network achieved 0.788. From these results it seems that the network using Adam has the best trade off between recall and precision. It should be noted, that the linear model can be tuned to find more/less artifacts using a higher/lower threshold value. The network models do not offer the same option due to how the models minimize loss. Take a look at the histograms: Figure 4.8.

4.6 | Training of the multi classification network

As mentioned earlier, this network is to distinguish between clean EEG samples, normal artifacts and electrode artifacts. The network used for this tasked was trained with the maximum batch_size of 10. The setting of the LSTM-cell make the network either predict 2 or 4 classes. Keeping the extra class should not influence the model that much. The network should quickly learn, that no annotations belong in this class. Therefore the extra class is kept.
The weighting of the cross-entropy loss function is found. This is done by running the script: "weights_in_crossentropyloss.py" [11]. Here it is found that the total training data amounts to 1401 hours. This is drastically greater than the 100 hours described in the theory chapter. The reason being, that the hours listed in the theory chapter is across multiple channels. In the description of the data, a 10 hour recording could for example mean 10 hours times 22 channels. In the model training only a single channel is looked at, so the 10 hour recording would amount to 220 hours of training data. Of these 1401 hours of recorded data, a total of about 90 hours are contaminated with electrode artifacts. A total of 206 hours are contaminated with other types of artifacts. Thus the weighting of the different classes becomes: $[\frac{1401}{1401-90-206}, \frac{1401}{206}, \frac{1401}{90}, 1.] = [1.27, 6.80, 15.57, 1.]$. The network was trained using the weights:

[11] https://github.com/marctimjen/Artefact-Rejection/blob/main/mulitclass_net/1.%20hyperoptimize/weights_in_crossentropyloss.py

[1.25, 6.67, 16.67, 1.]. Where the intention is that a higher value for the electrode artifacts, would force the network to learn how annotate this type of artifact.
The setting of the base and maximum learning rate, has been done in accordance with the instruction described by Smith [19]. Please refer to Appendix C. The linear learning rate test resulted for the SGD optimizer in a base_lr of 1.2 and max_lr of 1.3. For the Adam optimizer it is found that base_lr = 0.007 and max_lr = 0.0138. The exponential learning rate test was also conducted. The results are plotted in Appendix C. For the Adam optimizer the lowest loss is found to be 0.05. Therefore the max_lr is set to 0.005. The base_lr is found to be $\frac{0.005}{6} = 0.00083$. The network using SGD had the lowest smoothed loss at 8. Thus max_lr become 0.8 and base_lr become $\frac{0.8}{6} = 0.1333$.

4.6.1 | Results from the multi class SGD network

Firstly the learning rate parameters were set to go from 1.2 to 1.3 (these bounds were found in the linear learning rate test). This network was hyper optimized to have 0 momentum and 0 weight decay. Unfortunately this resulted in the network failing (the network only predicted one class). Take a look at the logs [12].
After this model failed, the parameters from the exponential learning rate is tested. Here the max_lr become 0.8 and base_lr become 0.1333. With these parameters a cyclical momentum test was conducted see Figure C.3 in Appendix C.
Using this plot it was found that the large cyclical momentum was very unstable (middle column). The validation accuracy even plateaus, which is an indication that the network only annotate all data as one class. This is an example of a badly set momentum. The other two configuration no momentum and momentum in range of (0.8-0.9) were more stable. Both of these configuration could have been used for further training. The configuration with no momentum seemed to have a more stable learning. Using this as hyper-parameters, the weight decay can be tested with a grid search. The results can be seen in Appendix C (Figure C.4).
From this plot it can be concluded that the left column has the fastest convergence and less volatility. Therefore a weight decay of 1e-5 is used in the training of the network with the SGD optimizer.
Therefore the final hyper parameters for the network ended up being set as a base_lr of 0.133 and a max_lr of 0.8, no momentum and a weight decay of 0.00001.
Unfortunately this network has a very unstable training, and after epoch 17 it has fully diverged. See the logs [13]. This can also be seen in the plot:

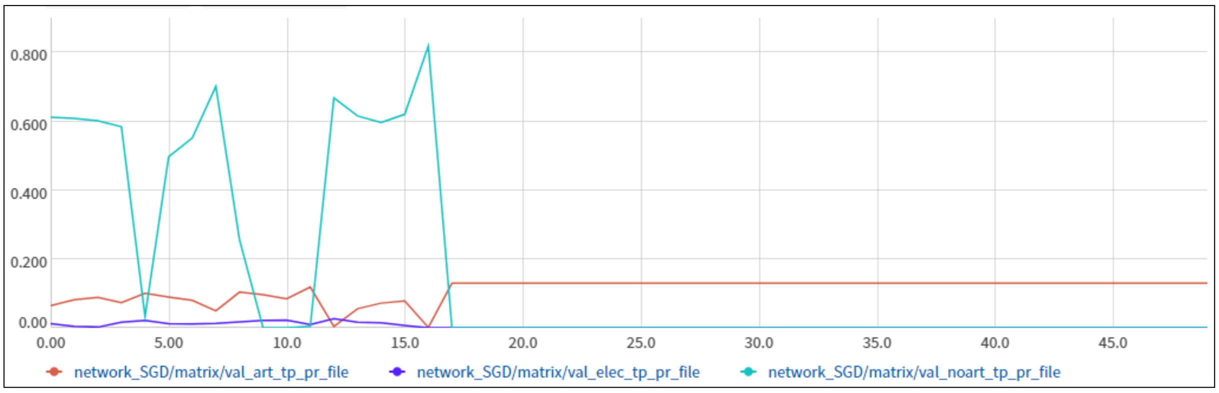


Figure 4.14: Failed training of the SGD network
Plot made using Neptune logs

This plot show the recall for the different classes. It can be seen from the plot, that when recall from one class spike, the recall from another class drop. Therefore it seems that the network were trying to figure out how it should annotate the different artifacts. But at epoch 17 the network has diverged and only annotate all samples as "normal artifacts". The lowest loss for this network was achieved at epoch 16. But at this point the model pretty much only guessed every sample as clean EEG. This network was tested on the test-set, and the model did not find a substantial amount of artifacts.
From this it seem that the network were learning the correct features, and that the training where heading in the right direction. A training with a lower learning rate was conducted. Here the parameters where set

[12] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-617/all?path=.
[13] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-660/all?path=.

to be base_lr of 0.007 and a max_lr of 0.013, no momentum and a weight decay of 0.00001. This resulted in a more stable run. The neptune logs for this training can be found here [14]. The training ended up having the lowest loss at epoch 30. In the following a plot over the recall of the different classes be seen.

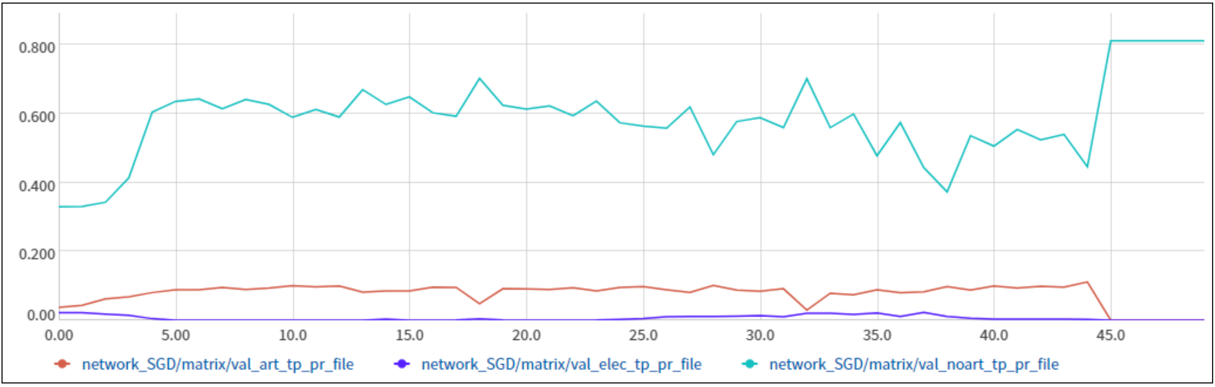


Figure 4.15: Successful SGD network training
Plot made using Neptune logs

The plot showed a more stable learning than before. Though the network still diverge - this happens at epoch 45. The network from epoch 30 was used to annotate the test data. This resulted in the confusion matrices:

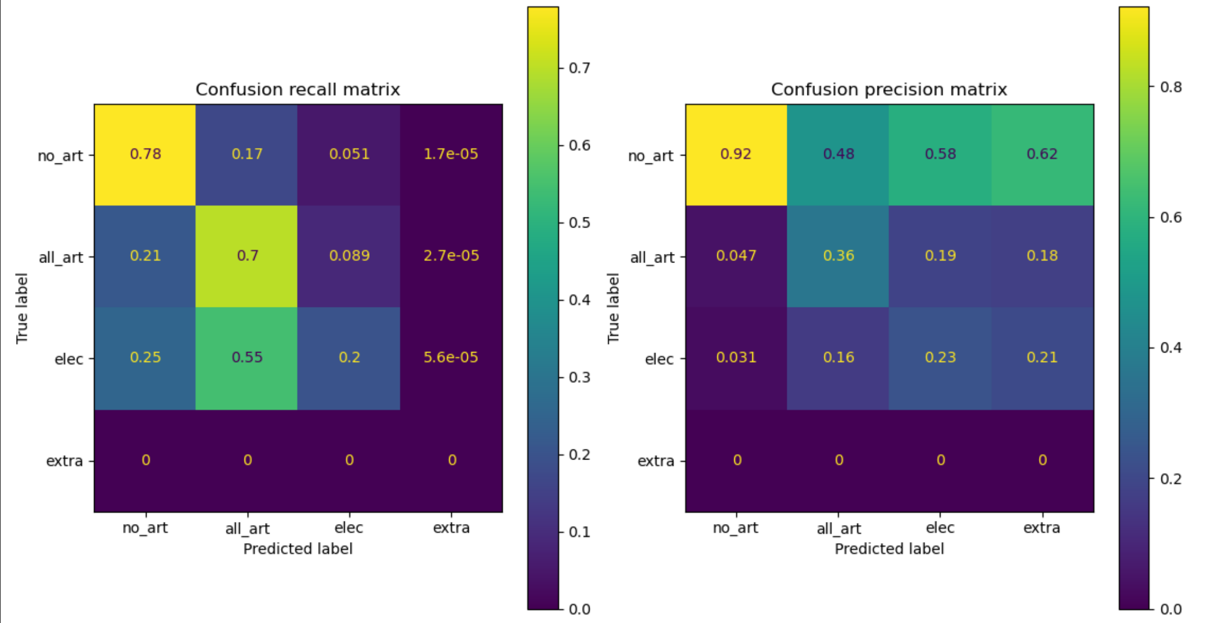


Figure 4.16: Confusion matrices for multi-class SGD network
Plot made using the test data set

From this plot one can see that the notwork can recall 78 % of the clean EEG-data, 70 % of the normal artifacts and about 20 % of the electrode artifacts. This is with the precision of 92 % on the clean EEG data, 36 % on the normal artifact data and lastly 23 % on the electrode artifact data. It should be noted that the network annotate very few samples with the ”extra” channel. Since none of the data is classified with the extra channel, the network should have learned to not use this channel. It is though interesting as to why the network use this channel. One explanation would be that the network, is unsure of which annotation a specific sample should have. Therefore the network end up returning a tensor with values close to 0.25 for all classes. In this case the network just so happen to give the last class a higher probability (and thus annotate a sample with the extra class).

[14] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-634/dashboard/plot-9675fe2e-436b-4259-ab79-2bc976db89ba

4.6.2 | Results from the Adam network

Firstly the values obtained from the linear learning rate test was used. The hyper optimized values was set as: base_lr = 0.007 and max_lr = 0.013 and weight decay of 0. After a few epochs this model diverged resulting in the model annotating all the samples as no artifact [15].
After this the model was tested with the exponential learning rate bounds: base_lr = 0.00083 and max_lr = 0.005. The results of optimizing the weight decay can be seen in Figure C.5 Appendix C.
From this plot the weight decay of 0.00001 seemed to result in the most stable training of the network. Therefore the network was trained using the mentioned hyper-optimized values. The logs from the training of the network can be found here: [16]. From the valdiation loss curve in Neptune, it was found, that the lowest loss was achieved at epoch 23.
This network was used to annotate the EEG data from the test-data. This resulted in the following confusion matrices:

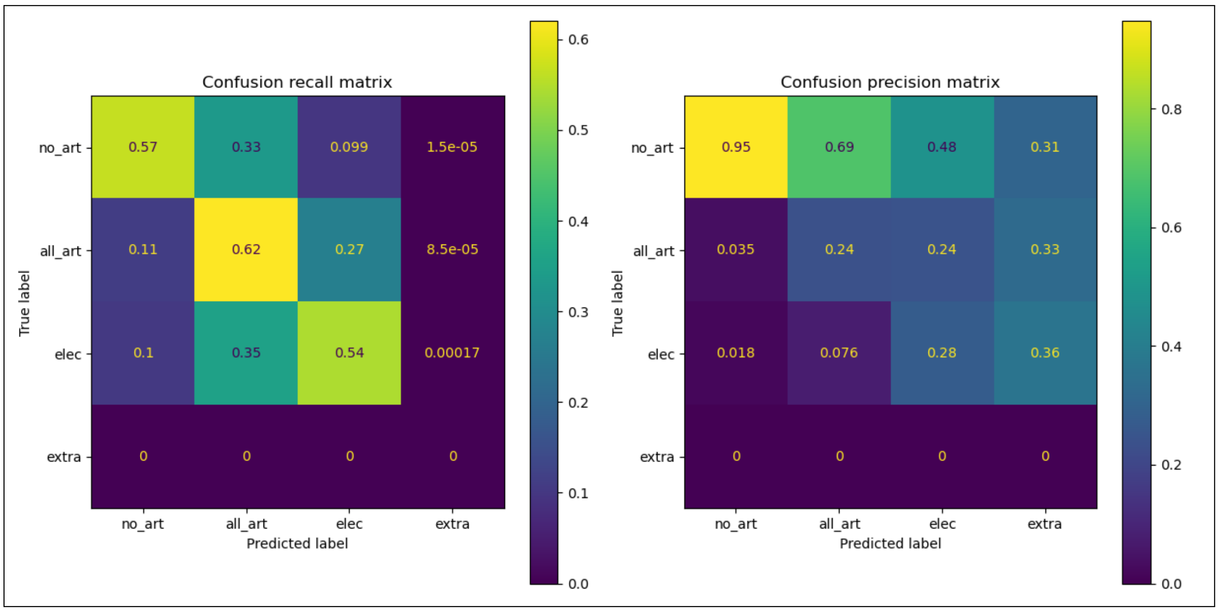


Figure 4.17: Confusion matrices for the multi-class Adam network
Plots made using the test data set

From these matrices it can be seen that this network is capable of recalling about 62 % and 56 % of respectively the artifacts and electrode artifacts. These artifacts are found with a precision of 24 % and 28 % for the normal artifacts and the electrode artifacts respectively. It should be noted, that about 24 % of the predicted electrode artifacts was instead normal artifacts. The other way around it was about 7.6 %. The model has a precision of 95 % on the clean EEG data and manage to recall about 57 % of this data. Further analysis of the networks is done in subsubsection 4.6.3.

4.6.3 | Test results for the multi class networks

It should be noted, that the following tables have more strict standards than the ones presented in subsubsection 4.5.3. In the tables presented in subsubsection 4.5.3 the network does not distinguish between the different classes of artifacts. In the following tables the correct annotation for electrode artifacts is needed. This means that if the model annotates an electrode artifact as 1 (the class for normal artifacts) the evaluation would count this as a false negative.

[15] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-618/all?path=.
[16] https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-647/all

The tables can be seen here:

Networks	No artifacts				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Network using SGD	0.778	0.922	0.502	0.779	0.772
Network using Adam	0.639	0.948	0.373	0.567	0.892

Networks	All artifacts				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Network using SGD	0.746	0.130	0.980	0.713	0.748
Network using Adam	0.631	0.078	0.966	0.581	0.634

Networks	Elec				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Network using SGD	0.883	0.234	0.931	0.199	0.943
Network using Adam	0.848	0.275	0.956	0.543	0.875

Networks	Musc				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Network using SGD	0.766	0.198	0.982	0.806	0.763
Network using Adam	0.657	0.136	0.973	0.753	0.650

Networks	Eyem				
	mean acc.	tp prec.	tn prec.	tp recall	tn recall
Network using SGD	0.718	0.034	0.978	0.373	0.727
Network using Adam	0.615	0.023	0.973	0.342	0.622

In the table above it seems that the network that utilize Adam has a higher recall on the artifacts than the SGD network. Though it seems that the SGD network has a better recall on the clean EEG data. The true positive percentage varies between the SGD network and Adam network in the different classes. It seems as if the SGD network has a higher mean accuracy in the different classes. It should be noted that the Adam network has a higher recall and precision on electrode artifacts. The macro-average F1 scores is 0.528 for the Adam network and 0.531 for the SGD network. Based on this it can not be concluded which model is better. This is also often dependent on the usage of the network.

4.7 | Summery of learning rates

The following table lists the learning rate test values:

Networks	Using Adam optimizor			Using SGD optimizor		
	base_lr	max_lr	diff.	base_lr	max_lr	diff.
Exp. binary model	0.00033	0.002	0.0017	0.0167	0.1	0.0833
Exp. multi model	0.00083	0.005	0.0042	0.8	0.133	0.677
Linear binary model	0.0089	0.013	0.0041	0.216	0.268	0.052
Linear multi model	0.007	0.0138	0.0068	1.2	1.3	0.1

The red color indicates learning rates for models that diverge (training was unsuccessful). The orange color indicates models that was not tested. The lime color indicates models that were trained, but the base_lr was changed. The green color indicates models with successful training. From the table above it can be seen that the network using Adam was more robust to the hyper parameter settings.
From the table it can be seen that the Adam optimizer would train at a lower learning rate. The difference between the maximum and base learning rate is smaller for the network using Adam than the difference for the network using the SGD optimizer. It can be seen that the exponential learning rate tests would result in lower learning rate values than the linear learning rate tests. The multi class network would seem to train at a higher learning rate, according to the learning rate tests.

5 | Discussion

As it can be seen from the result chapter, it is possible to use a neural network to automatically annotate artifacts. The Adam and SGD networks outperformed the naive model. However the usage of the networks can be discussed. On one hand it could be argued, that the network models does not recall all artifacts, and the networks have low precision when detecting specific artifacts. Therefore one could argue, that if the networks were deployed to do real artifact annotation, a human annotator of EEG-data should afterwards correct both the false positives and false negatives of the network. On the other hand one can argue that a human annotator is bound to make mistakes too. As it was also mentioned in section 2, the annotators of the data set have entered an agreement on how to annotate the different artifacts. This indicates that subjectivity can effect the annotations. Thus different annotators might conclude different annotations on the same EEG-data.
To put things into perspective Kim and Keene [8] have done artifact annotation on an older version of the TUH artifact data set. Their best binary model was a deep CNN that achieved a test accuracy of 75 %. Kim and Keene also implemented a LSTM (RNN) network, which achieved a test accuracy of 71.2 %. The binary networks presented in this report had a test accuracy of 76.8 % when using SGD and 81.3 % when using Adam as optimizer. It should be noted, that Kim and Keene loaded data differently than how data is loaded in this report. This might influence the accuracies of the models. Another interesting point, is that Kim and Keene used an older version of the data set. Some updates have been made, and thus also new annotations. This is another factors that can effect the accuracy of the models. As can be seen from Figure 4.3 and Figure 4.4, some annotations might still be wrong.
There is a trade-off when data loading is performed. One of the difficulties encountered in this report, was that different patients have different length of recorded EEG data. On one hand, the goal is to present the model with as much and diverse data as possible. On the other hand the model should not over-fit the training data. Mainly the issue was that idiosyncratic features made it important that a diverse group of patients were represented. Using the Lorenz curves it was found that a perfect equality between patients could be achieved. Please refer to subsection 3.2. The perfect equality means, that the same amount of intervals is sampled from the different patients. This on one hand make sure that the models trained does not over-fit the idiosyncratic features of a single person. On the other hand this might cause the model to run the same EEG-data over and over again. This might cause the network to over-fit these EEG input samples. On one side the model should not over-fit a patients features. On the other side a model should not over-fit to a specific EEG-recording. In this report the equality parameter was set to 110 and this is shown in Figure 3.4 (plot (b)).
In subsubsection 4.6.3 it was found that the network can do multi-class annotation. It was tested if the network could distinguish electrode artifacts apart from normal artifacts and clean EEG-data. The network using Adam as optimizer recalled 54.3 % of the electrode artifacts in the test data. This was with a precision of 27.5 %. It can be discussed if this model can be used to determine the quality of the EEG-recording equipment. It would make sense, that the equipment needs attention and care if more and more electrode artifacts are generated. As stated in the introduction, it is important that as few artifacts as possible is produced in the recording process. On one side it could be argued that the network using Adam has a to low recall and precision for the electrode artifacts to be deployed. The argument being that the model does not show good enough performance. It would also be necessary to test if the models consistently obtain the recall and precision percentage of the data. On the other side it could be argued that an automatic model is faster at annotating than the human annotators. Therefore the network could be handy if the equipment is to be tested in a short times notice.
The input EEG-signal is recorded between two electrodes (or an electrode and a reference). If one of these are broken the signal from the derivation will be contaminated with electrode artifacts. The model or a human annotator will be capable of detecting that the derivation has gone bad. To be able to tell which electrode(s) is at fault, one need to use re-referencing. This should not be a problem in the TUH data set, since linked ears and average references have been used. But if other configurations are used, re-referencing needs to be possible if one wants to detect which electrodes are broken.
In this report it was found that the Adam optimizer trained the best. The only time this optimizer made the network diverge was for the learning rates presented by the linear learning rate test for the multi-class network. Overall this optimizer seemed to be robust when trained with the parameters found. The final networks often ended in divergence when using the hyper-optimized parameters of the SGD optimizer. Based on the results it is believed that a to large base and maximum learning rate is used for the network using SGD. The learning rate boundaries found might be to large because not enough training and validation data have been used in the learning rate tests.

The hyper parameter tuning instructions from Smith is for convolutional neural networks only. The LSTM-cell (that is a RNN) implemented in the network might influence how the parameters should be set (it might cause the learning rate test to show the wrong learning rate boundaries). In the training sessions it was sometimes necessary to backtrack and use an earlier train version of some of the networks. This was because some of the networks would diverge later in the training or the performance of the network was impacted to much by the large learning rates of the scheduler.

The way of correctly setting up network training lead to the discussion on how to approach a deep learning task such as the one presented in this report. On one side the goal is to achieve the most optimal model. This would require the perfect architecture and the correct settings of hyper parameters such as learning rate. The big question is how this is done. An argument could be that a lot of testing of different models, activation functions, loss functions and so on is needed. This would result in a brute force approach to the problem. On the other side one could argue, that using methods/architectures that has achieved good results on similar problems is an advantageous place to start. This is especially useful, if one can reason why these methods are superior. The issue is that these assumptions often lack theoretical proof. Xu, Wang, Chen and Li note in their report, that the reason for LReLU (and other versions) outperforming ReLU still need a justification on a theoretical level [16, page 4]. So the issue in this debate is how to achieve the best model. In this report both techniques have been implemented.
Testing of the architecture is important. In this report some peculiarities about the standard implementation of the u-net was found. This happened during testing of the u-network. In the testing constant values (E.g. only zero values) were given to the u-net as input. The first convolutional layer would perform its convolutions. The convolution function does add a bias value after the multiplication with the parameters. For the zero input, the first convolutional layer would return a tensor filled with the bias value. The result is therefore constant for all values. Applying batch-normalization thus de-mean the data (setting all values to zero). This would mean that information about the initial input value have been lost. This means that the u-net would not be capable of classifying constant input samples differently. E.g. the network could not perform the task: Return values of 1's for 0 constant inputs and return 0's if a constant value of 1 were given.
The network should be capable of returning one specific class for any constant input. This is because the last operation is a convolution. The examples above does not need to be solved using machine learning. A simple "if" statement could nearly do the job of this application.
In this report the data is processed in such a way that a constant input value can appear at the values of 0, -200 or 200. These cases are extremely rare. With the input length set as 5 minutes, the raw EEG amplitude needs to constantly be above 200 µV (or below -200 µV) for the duration of the 5 minutes for a constant of +/- 200 to be given. In the cases of constant value of -200, 0 and 200 the binary network should return only artifacts. The multi-class network should most likely return electrode artifact only. It would make no sense if a person produced e.g. a muscle artifact for a duration of 5 minutes. Since all constant inputs should be annotated as artifacts the standard u-net model should be capable of learning this. Testing of the u-network revealed that it had some difficulties learning to output artifacts if the input EEG-signal suddenly became constantly 0. Annotating constant 0 values as artifacts is not that difficult, and implementing a neural network to do this would clearly be overkill. Though it got me thinking about how to improve the model, and make it learn the correct features. The LSTM-cell was mostly added for its ability in sequence labeling tasks (please refer the section 3). As an added bonus, this network should be capable of learning that a constant input value of 0 should be annotated as 1 (as a normal artifact) or 2 (as an electrode artifact).
To further develop an artifact annotation model that achieve even higher accuracy, one could look into making a model that utilize transfer learning. This would require one to train the u-net (or another model) to hopefully generate a network that can annotate the most common artifacts across different patients. Utilizing this model, one could then make it learn the idiosyncratic features of a specific patient. This would hopefully create a model that is more accurate at annotating artifact in the EEG-data for this specific person. To implement this an human annotator first have to annotate a patients EEG-recording. Then use transfer learning to train the network to find the artifacts for this specific person. At this point the model will hopefully be good enough to do the artifact annotations for the future/other EEG-recordings from this patient.

6 | References

[1] Mayo Clinic. Eeg (electroencephalogram). Accessed: 2022-5-31. URL: https://www.mayoclinic.org/tests-procedures/eeg/about/pac-20393875.

[2] Riitta Hari and Aina Puce. *MEG-EEG primer*. Oxford University Press, New York, NY, 2021 - 2017.

[3] Kaare B. Mikkelsen, Yousef R. Tabar, Simon L. Kappel, Christian B. Christensen, Hans O. Toft, Martin C. Hemmsen, Mike L. Rank, Marit Otto, and Preben Kidmose. Accurate whole-night sleep monitoring with dry-contact ear-EEG. *Scientific Reports*, 9(1):16824, November 2019. doi:10.1038/s41598-019-53115-3.

[4] Fábio Lopes, Adriana Leal, Júlio Medeiros, Mauro F. Pinto, António Dourado, Matthias Dümpelmann, and César Teixeira. Automatic electroencephalogram artifact removal using deep convolutional neural networks. *IEEE Access*, 9:149955–149970, 2021. doi:10.1109/ACCESS.2021.3125728.

[5] Jose Urigüen and Begoña Zapirain. Eeg artifact removal – state-of-the-art and guidelines. *Journal of neural engineering*, 12:031001, 04 2015. doi:10.1088/1741-2560/12/3/031001.

[6] Md Kafiul Islam, Amir Rastegarnia, and Zhi Yang. Methods for artifact detection and removal from scalp eeg: A review. *Neurophysiologie Clinique/Clinical Neurophysiology*, 46(4):287–305, 2016. URL: https://www.sciencedirect.com/science/article/pii/S098770531630199X, doi:https://doi.org/10.1016/j.neucli.2016.07.002.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] Dong Kyu Kim and Sam Keene. Fast automatic artifact annotator for eeg signals using deep learning. In *Biomedical Signal Processing*, pages 195–221. Springer International Publishing, Cham, 2021.

[9] Neural Engineering Data Consortium. Electroencephalography (eeg) resources. Accessed: 2022-5-28. URL: https://isip.piconepress.com/projects/tuh_eeg/html/downloads.shtml.

[10] A. Hamid, K. Gagliano, S. Rahman, N. Tulin, V. Tchiong, I. Obeid, and J. Picone. The temple university artifact corpus: An annotated corpus of eeg artifacts. In *2020 IEEE Signal Processing in Medicine and Biology Symposium (SPMB)*, pages 1–4, 2020. doi:10.1109/SPMB50085.2020.9353647.

[11] Sean Ferrell, Vineetha Mathew, Matthew Refford, Vincent Tchiong, Tameem Ahsan, Iyad Obeid, and Joseph Picone. The temple university hospital eeg corpus: Electrode location and channel labels. *Neural Engineering Data Consortium*, April 6, 2022. URL: https://isip.piconepress.com/publications/reports/2020/tuh_eeg/electrodes/.

[12] MNE. mne.io.raw. Accessed: 2022-5-27. URL: https://mne.tools/stable/generated/mne.io.Raw.html#mne.io.Raw.notch_filter.

[13] Maxim integrated. Ekg explained and intro to electrocardiographs. Accessed: 2022-5-27. URL: https://www.maximintegrated.com/en/design/technical-documents/tutorials/4/4693.html.

[14] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL: http://arxiv.org/abs/1505.04597, arXiv:1505.04597.

[15] Nikolas Adaloglou. An overview of unet architectures for semantic segmentation and biomedical image segmentation. *AI SUMMER*, 04 2021. Accessed: 2022-6-13. URL: https://theaisummer.com/unet-architectures/.

[16] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015. URL: http://arxiv.org/abs/1505.00853, arXiv:1505.00853.

[17] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *CoRR*, abs/1402.1128, 2014. URL: http://arxiv.org/abs/1402.1128, arXiv:1402.1128.

[18] Ajitesh Kumar. Model complexity & overfitting in machine learning. *Data Analytics Data, Data Science, Machine Learning, AI*, 05 2022. Accessed: 2022-6-9. URL: https://vitalflux.com/model-complexity-overfitting-in-machine-learning/.

[19] Leslie N. Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186v2, 2015. URL: https://arxiv.org/abs/1506.01186v2, `arXiv:1506.01186v2`.

[20] pytorch. Crossentropyloss. Accessed: 2022-6-12. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[21] Leslie N. Smith. A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. *CoRR*, abs/1803.09820, 2018. URL: http://arxiv.org/abs/1803.09820, `arXiv:1803.09820`.

[22] Leslie N. Smith. Cyclical learning rates for training neural networks, 2015. URL: https://arxiv.org/abs/1506.01186, `doi:10.48550/ARXIV.1506.01186`.

[23] Thomas Dehaene. Adaptive - and cyclical learning rates using pytorch. *Towards Data Science*, Mar 20, 2019. Accessed: 2022-5-30. URL: https://towardsdatascience.com/adaptive-and-cyclical-learning-rates-using-pytorch-2bf904d18dee.

[24] pytorch. Cycliclr. Accessed: 2022-6-13. URL: https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CyclicLR.html.

7 | Figures

Figure 2.1 Example of clean EEG data. This plot was made with the program EDF-browser. The data is from the time frame 60 secs to 63 secs. The data is from the patient: 00000254_s005_t000 from the TUH EEG Artifact Corpus.

Figure 2.2 Example of artifacts. This plot is made using matplotlib in python. See both the jupyter lab and python file (called artiplot) at: https://github.com/marctimjen/Artefact-Rejection/tree/main/plotting%20of%20artifacts.

Figure 3.1 Plot of the experiment EEG on the left and the annotations on the right. This plot is made using pytorch in jupyter lab. See the jupyter lab at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20loader/test_of_loader.ipynb

Figure 3.2 Histogram of the intervals per recordings and patients before any correction. This plot is made using the data loaders and matplotlib in jupyter lab. See the jupyter lab at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20loader/test_of_loader.ipynb

Figure 3.3 Histogram of the intervals per recordings and patients after correction. This plot is made using the data loaders and matplotlib in jupyter lab. See the jupyter lab at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20loader/test_of_loader.ipynb

Figure 3.4 Lorenz curve before and after the correction. This plot is made using the data loaders and matplotlib in jupyter lab. See the jupyter lab at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20loader/test_of_loader.ipynb

Figure 3.5 Lorenz curve with extreme correction. This plot is made using the data loaders and matplotlib in jupyter lab. See the jupyter lab at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20loader/test_of_loader.ipynb

Figure 3.6 Plot of the network architecture. This plot is an augmented version of the plot in: https://theaisummer.com/unet-architectures/

Figure 3.7 Plot of the simple cyclical learning rate scheduler. This plot is taken from [19, page 3].

Figure 3.8 Examples of learning rate test. This plot is an augmented version of the plots taken from [19, page 8 and 9].

Figure 3.9 Examples of exponential learning rate test. This plot is an augmented version of the plots taken from [23].

Figure 4.1 Histogram over naive models guesses. This plot is made using the validation data set and the file https://github.com/marctimjen/Artefact-Rejection/blob/main/Small%20tests/val_testing.py.

Figure 4.2 Confusion matrices for the naive model. This plot has been created by using the test data set. See the code at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20the%20models/binary%20linear%20model.py.

Figure 4.3 Plot of EEG data and the associated annotations. This plot has been made using the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/plotting%20of%20artifacts/range%20plot.py. There is also an associated jupyter file. The data is from the patient s007_2013_03_25 and specifically the recording 00000254_s007_t000.edf.

Figure 4.4 Plot of clean EEG data. This plot has been made using the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/plotting%20of%20artifacts/range%20plot.py. There is also an associated jupyter file. The data is from the patient s007_2013_03_25 and specifically the recording 00000254_s005_t000.edf.

Figure 4.5 Linear learning rate tests of the binary network. This plot is made using the logs from Neptune and the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/plotting/1.learn_plotter.py

Figure 4.6 Exponential learning rate tests of the binary network. This plot is made using the logs from Neptune and the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/plotting/1.learn_plotter.py

Figure 4.7 Weight decay tests of the Adam network. This plot is made using the logs from Neptune and the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/plotting/3.weight_decay_plotter_adam.py

Figure 4.8 Histograms made using the Adam network. This hisogram has been produced from the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/Small%20tests/val_testing.py

Figure 4.9 Plot of the annotations by the Adam network. The data is from the time frame 30 secs to 330 secs. The data is from the recording: 00000254_s005_t000 from the TUH EEG Artifact Corpus.
This plot has been produced from the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/Small%20tests/val_testing.py

Figure 4.10 Confusion matrices for the Adam network. This plot is made using the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/plotting/3.weight_decay_plotter_sgd.py

Figure 4.11 Momentum tests of the SGD network. This plot is made using the logs from Neptune and the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/plotting/2.momentum_plotter.py

Figure 4.12 Weight decay tests of the SGD network. This plot is made using the logs from Neptune and the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/binary_net/plotting/3.weight_decay_plotter_adam.py

Figure 4.13 Confusion matrices for the SGD network. This plot is made using the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20the%20models/binary%20sgd%20unet.py

Figure 4.14 Failed training of the SGD network. This plot is made using the logs from Neptune at https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-660/all

Figure 4.15 Successful SGD network training. This plot is made using the logs from Neptune at https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-634/all

Figure 4.16 Confusion matrices for multi-class SGD network. This plot is made using the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20the%20models/elec%20sgd%20unet.py

Figure 4.17 Confusion matrices for multi-class Adam network. This plot is made using the script: https://github.com/marctimjen/Artefact-Rejection/blob/main/Testing%20the%20models/elec%20adam%20unet.py

A | Software versions

The network and training of the network have been utilizing: Python 3.9.6, pytorch 1.11.0 with cudatoolkit 11.3.1, numpy 1.21.6, neptune-client 0.16.2, pandas 1.4.2, matplotlib 3.5.2, pandas 1.4.2 and mne 1.0.3. The full list of software can be found at: https://github.com/marctimjen/Artefact-Rejection/blob/main/Software%20versions.ipynb
This software has mostly been used in the developing of the networks, training and test scripts. Most of the training and hyper-optimization have been done on a gpucluster at Aarhus university. The same software versions of software should have been used, but older versions might unintentionally have been used.

B | Training of binary Adam network

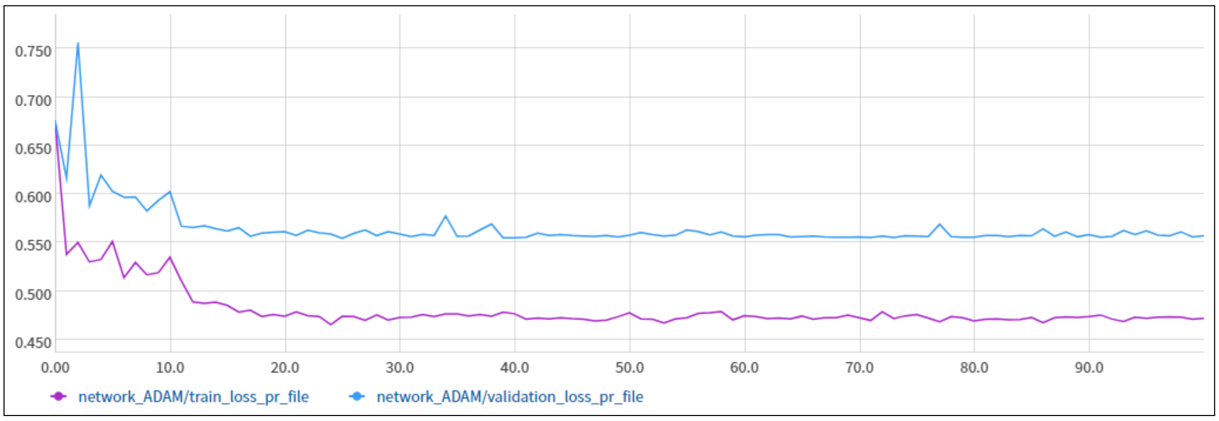


Figure B.1: Training and validation loss for the network with Adam

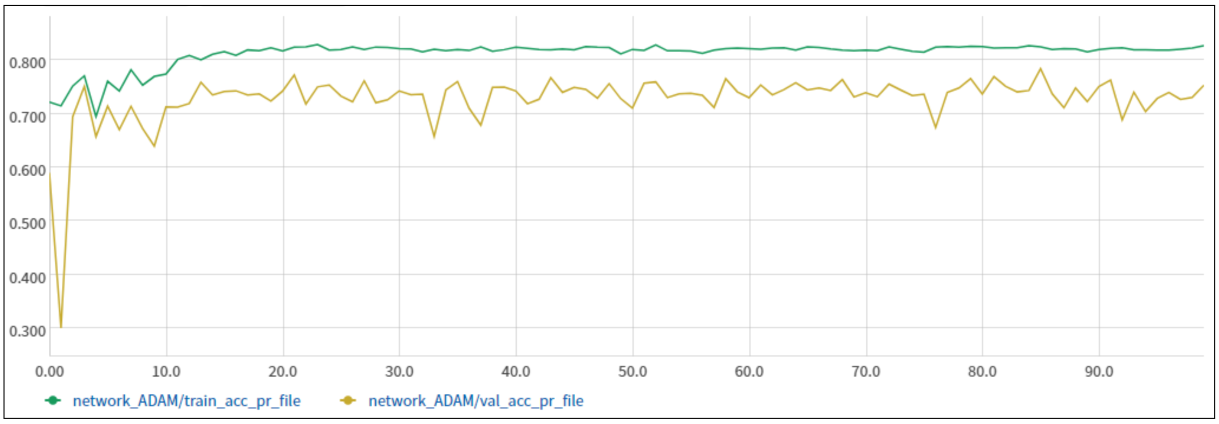


Figure B.2: Training and validation accaurcy for the network with Adam

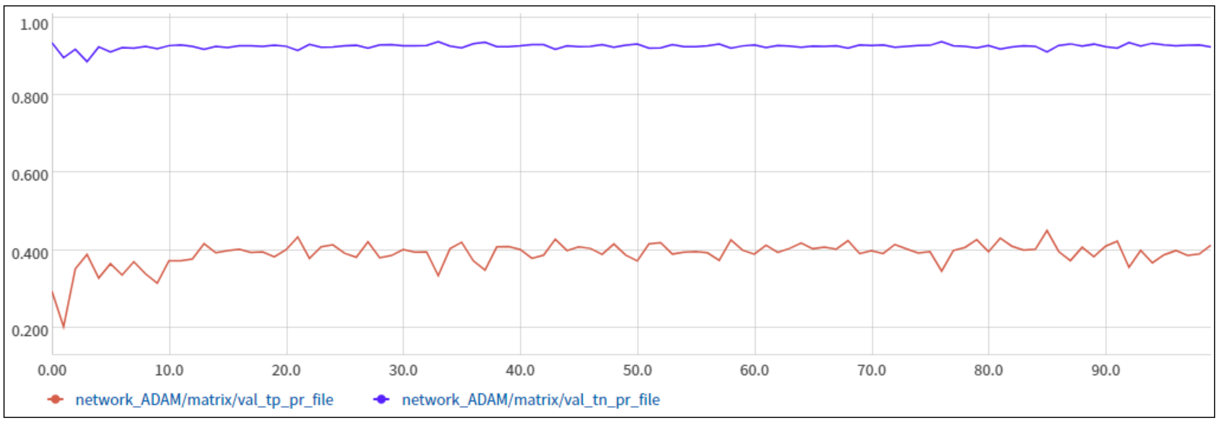


Figure B.3: Validation recall of true positive and negatives for the network with Adam

See the Neptune logs at: https://app.neptune.ai/o/NTLAB/org/artifact-rej-scalp/e/AR1-470/all

C | Multi-class hyper-optimization

These plots have been obtained by running the scripts in https://github.com/marctimjen/Artefact-Rejection/tree/main/mulitclass_net/plotting
The linear learning rate test:

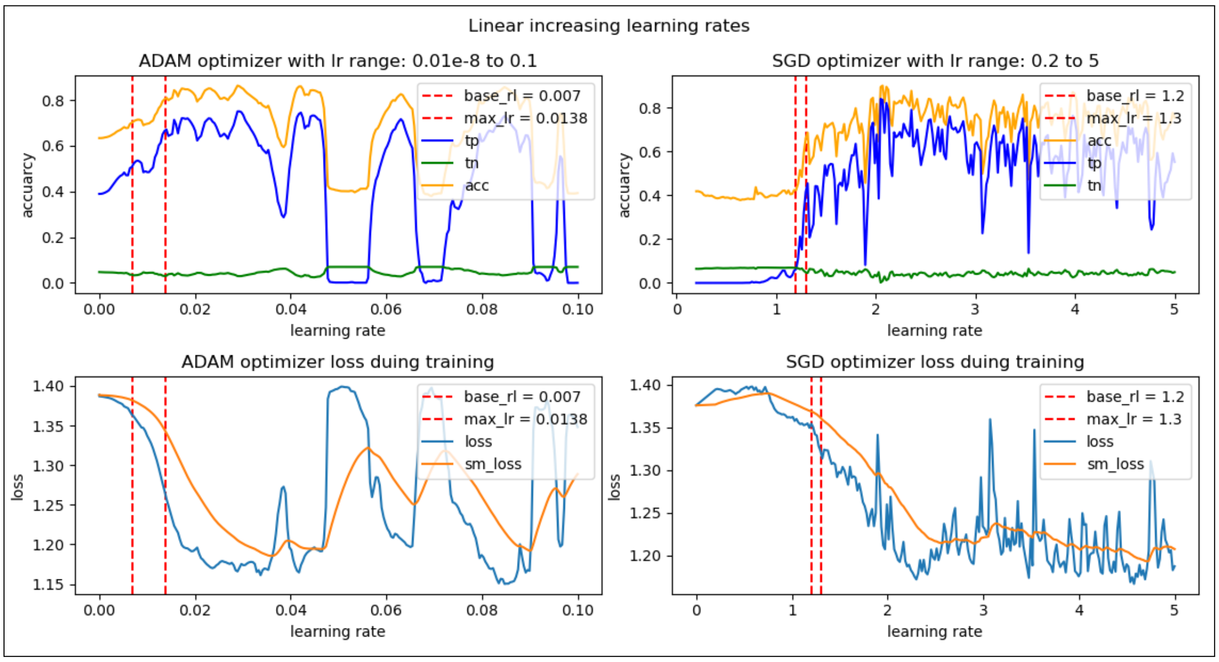


Figure C.1: Linear learning rate test for multi-class networks

The exponential learning rate test:

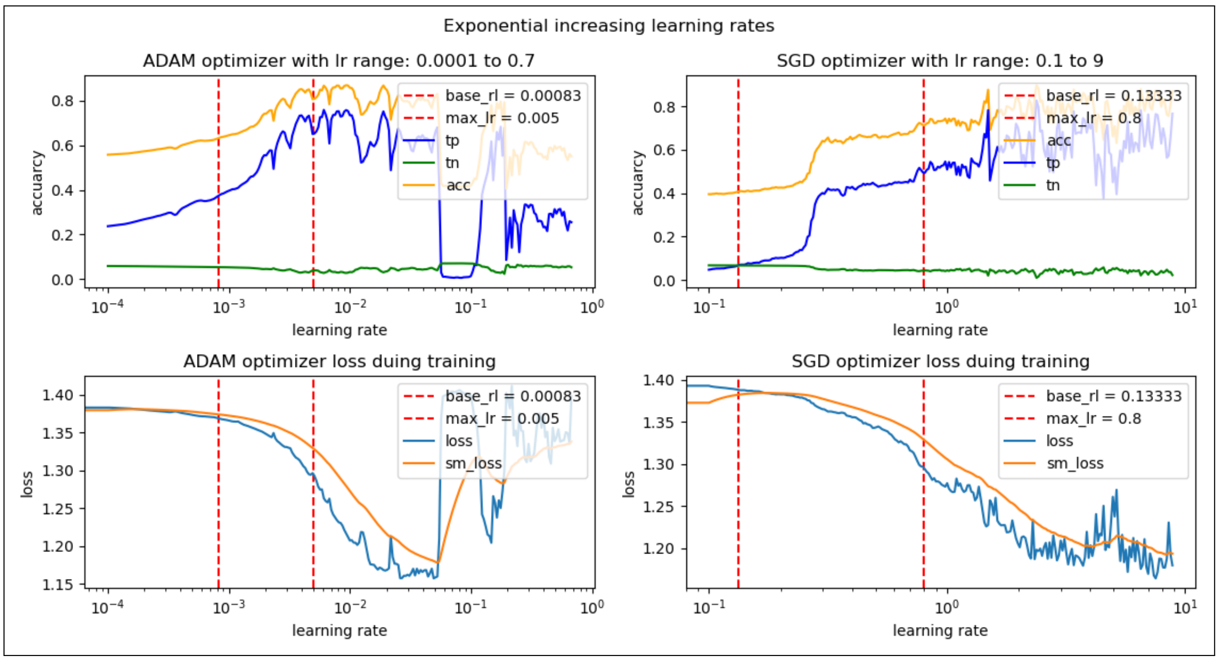


Figure C.2: Exponential learning rate test for multi-class networks

The momentum test for SGD:

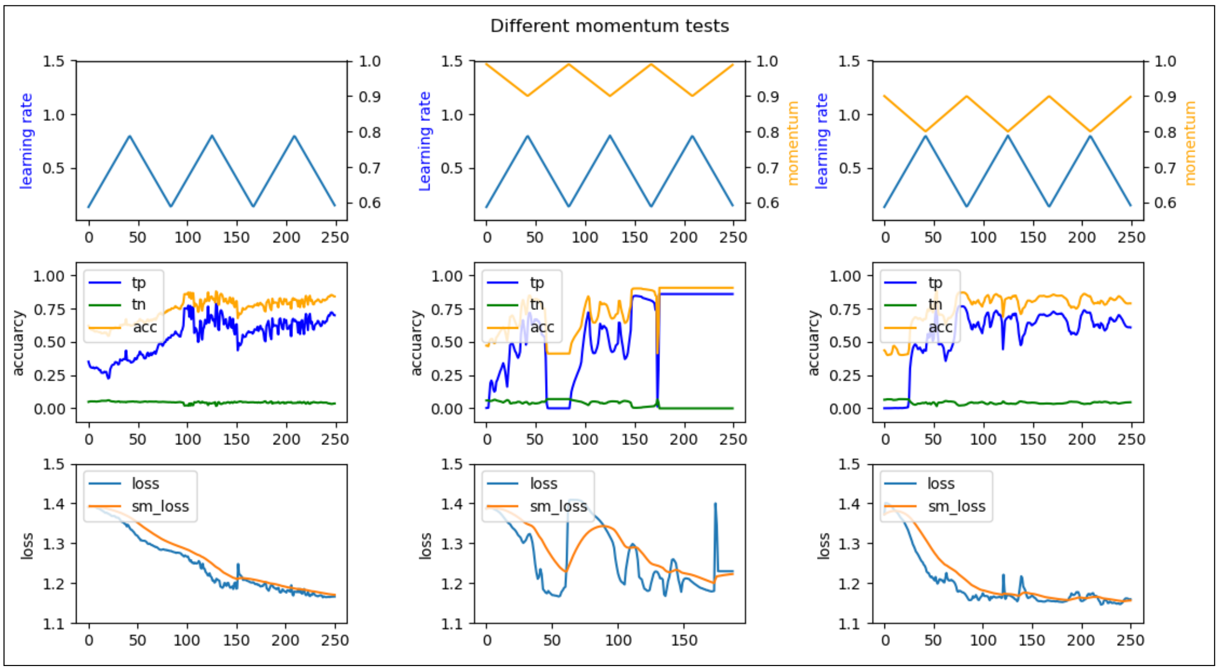


Figure C.3: Momentum tests for the multi-class SGD network

Weight decay test for SGD:

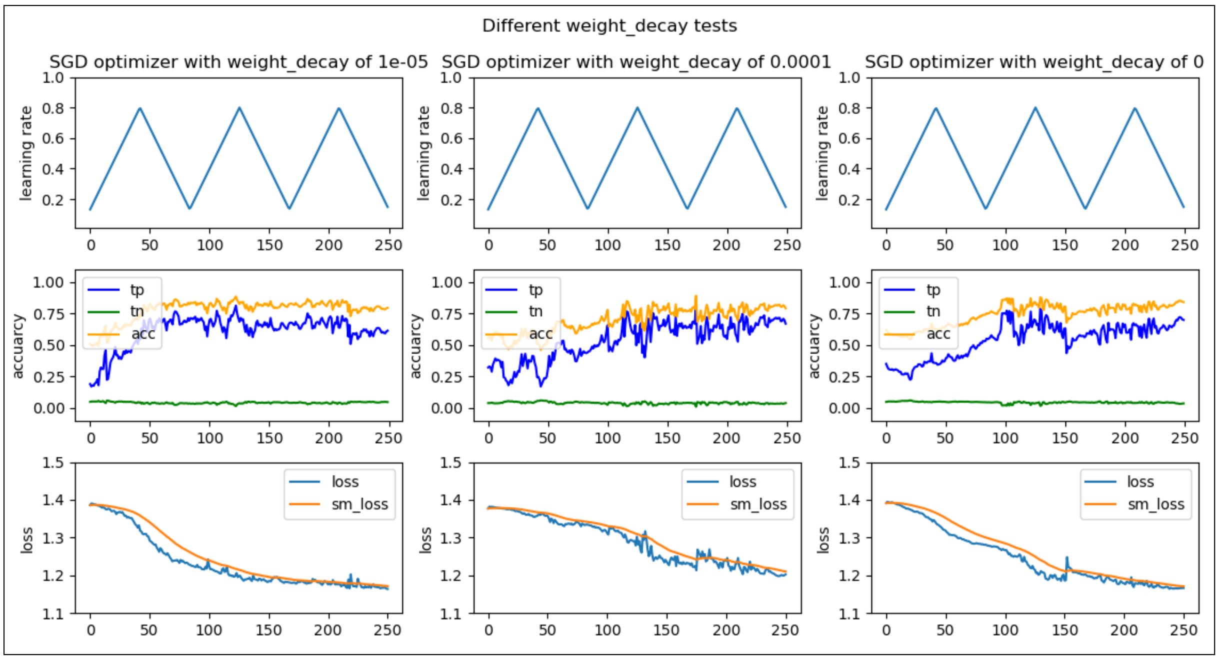


Figure C.4: Weight decay tests for the multi-class SGD network

Weight decay test for Adam:

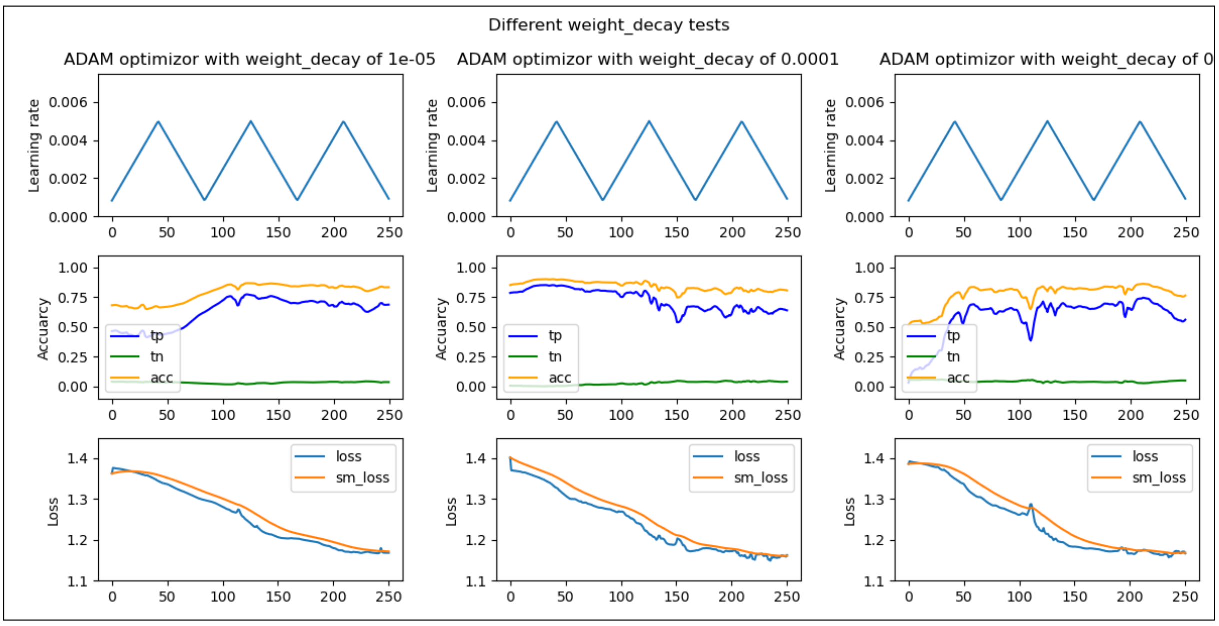


Figure C.5: Weight decay tests for the multi-class Adam network