1D Convolutional Neural Networks for Biploar Disorder Classification and Prediction based on Activity Measurements

Joakim I. Frogner

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

1D Convolutional Neural Networks for Biploar Disorder Classification and Prediction based on Activity Measurements

Joakim I. Frogner

1D Convolutional Neural Networks for Biploar Disorder Classification
and Prediction based on Activity Measurements

http://www.duo.uio.no/

Abstract

Contents

List of Figures

List of Tables

Preface

x

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Statistics
- Data shows that 5,890,000 adults are diagnosed with bipolar disorder in the USA (2,65% of the adult population) [find better source].

Ways to use the results of this study?

1.2 Thesis overview

[Fill in later]

Chapter 2

Background

2.1 Bipolar disorder

Bipolar disorder is the disorder where you experience extreme mood swings. One day you can feel amazing and everything is fine, but the next day you feel like you don't belong anywhere in this universe. Mood swings in general is not something that you should be concerned about. It is however the extreme cases where your mind turns 180 degrees from day to day that is the main symptom of bipolar disorder. There is not really a specific type of people that get this; they can be of any age and any gender, but most people that suffer from it find out (by having an experience or episode) around age 25 [3].

When talking about bipolar disorder, we often separate between the states *normal*, *mania* and *depression*. The last two are the states we usually talk about, since a normal state isn't that interesting. These two states are very different, but they have some similarities, for example sleeping problems.

When a bipolar person is in a manic state, he/she may do things that they never would have intended doing, like spending a lot of money on items they really don't need, or abusing drugs/alcohol. They may also feel really excited or powerful [2].

A bipolar patient is in a depressive state when he or she is in a bad mood swing. They can stop doing everything they usually like to do, and lie down in bed all day with no motivation to do anything useful. They may feel useless and that they don't belong here, or being guilty of something they may or may not have done. In some cases, a depression may even end up with suicidality, where the person either just thinks of death, or actually attempt suicide (actually 20% of people diagnosed bipolar commit suicide [3]).

The frequency of these symptoms can vary. One year they can have these mood swings every day for several weeks at the time, and the next they get them less frequent, like once every month.

We also separate between bipolar disorder type I and II, with the main difference being that the manic episodes are way more aggressive in type I [1].

Statistics say that bipolarity is genetically inheritable, with 23% chance of getting a child with bipolar disorder if one parent is bipolar, and 66% if both parents are [3].

2.1.1 MADRS Score

TODO

2.2 Machine learning

Machine learning is the field of computer science where you basically throw a lot of data into an algorithm and expect it to give you answers to whatever you prefer, with as little work as possible. This was not the case in the early days of the technology, but nowadays it is a lot easier with all the different frameworks and tools available.

Machine learning is a great and almost 'magical' technology, but only if you do it right. First you need to have enough data to feed into the algorithm, and to be efficient when training the model on a large dataset, which you need to be if you want your result quickly, you need good hardware. You can get away with a decent CPU if you just want to test it out on a small dataset, but if you really want to do machine learning, then you need a good GPU. The reason why GPUs are so much better than CPUs on this specific task, is because the CPUs are designed for flexibility and general computing workloads. The GPUs on the other hand, are designed to do the same instructions over and over again in parallel. This makes GPUs a lot more efficient for machine learning, and especially for deep neural networks [20].

Now how do you do the actual machine learning? Well there are many different approaches to this, which I will discuss in the next sections, but say you want to use a neural network for your task. Then your next step should be to choose a framework. You can of course do everything yourself, but why reinvent the wheel when there are so many good frameworks and tools already out there?

The programming language **Python** is great for machine learning in my (and many other peoples) opinion. It is structured in a way such that it really looks like pseudo-code, and this is perfect because we don't want to spend time on weird syntax rules in another language. For Python, you have a popular framework called **TensorFlow** which is developed by Google. This allows you to build models easily, and also execute the training and testing. Before you get started with TensorFlow, do a quick google search to see if someone else has already done something similar to what you are trying to achieve, and if you find something, odds are that your neural network model can be similar, if not identical to it. If not, then you have to sit down and actually make the model yourself.

For the model implementation part, whether you found a model online or want to build it yourself, you can of course do it in TensorFlow, but there is an easier way. **Keras** is also a popular framework that is most commonly used together with TensorFlow. On their documentation website [11], they see their framework as 'A high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK or Teano'.

Following their '30 seconds to Keras' guide [11], you can create a 'sequential' model with 'dense' layers, configure its learning process (compile), then fit, train, evaluate and predict with just a few lines of code:

Source Code 2.1: 30 Seconds to Keras

```python
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5, batch_size=32)
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
classes = model.predict(x_test, batch_size=128)
```

So, as long as you know your theory, and can decide which model to use (and either find a good implementation of that model or create it yourself), you can easily do machine learning. One important task you have to do, is to make the dataset ready. This is the boring and tedious part of machine learning, but it has to be done in order for making it possible to train the model on it.

2.3   Machine learning strategies

Picking the right machine learning model can be quite difficult, especially if you don't have any experience from earlier. There are a couple of different *strategies* you can choose from when deciding on a machine learning model. These are called *Supervised and unsupervised learning*, and you need to look at your dataset and how it is structured to find out which one to use. The following sections will be a description of the strategies, to make the decision easier.

2.3.1   Supervised learning

This is the machine learning strategy where both input and desired output data are provided [19]. You can use this if you want to train a model to classify letters in the alphabet, or anything else where you have a dataset with both input and output data (for the alphabet, images of letters are input data and the actual letters are the output data). If you train this alphabet model, you will be able to input a completely new image of a letter, and the model will classify it to the letter it most likely fits. This kind

of supervised learning is called *Classification*, and is *the problem of assigning new observations to the class they most likely belong, based on a classification model built from labeled training data* [14].

Another kind of supervised learning is called *Regression*, and is all about predicting (or estimating) a value. A classic example for regression learning is predicting income, using *features* like home location, job title, field of study, years of education and years of experience. We call these features *categorical* (first three) and *numerical* (last two) [13].

2.3.2 Unsupervised learning

Another strategy is *Unsupervised learning*. This is what you want to use if you have a dataset without the same meaning as in a dataset for supervised learning. The items may not have a fixed answer, like the letters in the alphabet are. It is useful when you have *unlabeled* data, and want to for instance group data together in what we call a *cluster*. It may not be as commonly used as supervised learning, but unsupervised learning can also be very useful in some cases; like grouping addresses together in neighborhoods if you have a unsorted list of addresses as a dataset.

2.3.3 Semi-supervised learning

Now, you may not always want to use one of the strategies above. Looking at your dataset you may want something in between; a combination of labeled and unlabeled data. This is when semi-supervised learning comes in handy. For example if you have a lot of data to give labels to in your dataset, it can be simply too much work. We won't go deep into details about how this works, but it is important to mention it because of its usefulness.

2.4 Machine learning approaches

When you know whether you want to use supervised learning, unsupervised learning or something in between, you need to select an approach. We call them approaches because you use these regardless of the strategy you end up using; most of them (with the exception of *reinforcement learning*, which we will come back to) work in both supervised and unsupervised learning. There are a lot of diffent approaches available, and we will describe some of them, namely those we will use in the main parts of this thesis.

2.4.1 Decision tree learning

In computer science, trees are data structures commonly used to describe something that *branches out* based on different input. For example a tree can be a representation of how the frequency of letters in the alphabet are distributed in a text file, so that the text file can be compressed optimally.

Day	Temperature	Outlook	Humidity	Wind	Run
1	15 C	Sun	Low	Strong	Yes
2	6 C	Rain	High	Weak	No
3	15 C	Rain	Medium	Strong	Yes
4	6 C	Overcast	High	Medium	Yes
5	15 C	Sun	Low	Weak	No
6	12 C	Overcast	Medium	Weak	No
7	12 C	Sun	Medium	Medium	Yes

Table 2.1: Training data set: days a person went out for a run

I won't go into details about how this works, but my point is that tree-structures are very common in most fields of computer science.

In machine learning, we can apply the tree-structures as *decision tree learning*. And in this approach, we set up all the different outcomes (with the training data set) of a specific question in a tree. Let's say you want to predict whether or not a person will run outside on a specific day. Then it makes sense that the training set contains weather information. The different data in the training set is called attributes, and picking these correctly is important for the quality of the prediction.

Table 2.1 contains data about whether a person went outside for a run or not for a week (just an example, not real data). Here the first 4 (excluding "Day") columns (Temperature, Outlook, Humidity and Wind) is the "predictors" and the last column (Run) is the "target". To use this table in decision tree learning, we need to view it as a tree, with one of the predictors as root node and the targets as leaf nodes.

How we choose the tree structure is critical to the performance of the machine learning, and we need to use a good tree building algorithm. The most common algorithm to use in this situation, is the *ID3* algorithm made by J. R. Quinlan. It is a top-down, greedy search through the space of possible branches with no backtracking [7]. The way this happens is by calculating *Entropy* and *Information Gain*. The idea is to recursively choose the predictor that has the highest information gain and generate a tree structure. With an optimal tree, you can create decision rules by simply following the tree with new data.

Random Forest

One known problem with decision tree models, is that they often include a lot of *Variance*. This means that an algorithm is sensitive to small changes in the training set. One method to reduce the variance, is to use Random Forest.

Random Forest is a supervised machine learning strategy, which can be used for both classification and regression learning [18]. It essentially works by combining decision trees, where the tree building algorithm is heavily randomized for all trees. For example, if you want to get movie recommendations using machine learning, using one decision tree will

most likely be insufficient. Just think what happens when you ask one friend for movies to watch. What that friend recommends is purely based on movies you like and what he has already watched. You might be lucky and find your next favorite movie, but most likely, asking multiple people for recommendations is going to give a better result. The same goes for machine learning, and decision trees will most likely give a better answer if they are combined in a Random Forest.

2.4.2 Neural networks

General idea

The general idea of machine learning with neural networks, is to make the computer think like a human; it is inspired by they way biological neural networks in the human brain process information [15]. There are a lot of different neural networks, but all of them share the same underlying layer based architecture, where data is passed between layers where computation is done. The first layer is the input layer, which simply passes the data to the next layer, which is the hidden layers. The number of hidden layers is completely up to the model and the programmer, and this is where the intermediate processing/computation is done, before the data is passed to the output layer where we perform an activation function to define the output [16].

If you have a lot of hidden layers in a neural network, we call it a *deep* neural network. This is commonly used, and there is a lot of different deep networks with a lot of different use cases. Two of the most common neural network models are *Recurrent Neural Networks* and *Convolutional Neural Networks*. These two have their own use cases, which I will describe below.

Figure 2.1 is a visualization of a basic neural network with the input layer on the left, one hidden layer in the middle and the output layer on the right hand side. Every node in the input layer is connected with every node in the hidden layer, and every node in the hidden layer is connected to every node in the output layer.

Recurrent Neural Networks (RNN)

This type of neural network is good for predicting something based on a sequence of data, like for example predicting words in a sentence, which can be especially useful for typing on a phone. Also doing predictions based on historical data, like a forecast, is something an RNN can do effectively, which is something the dataset that I'm going to use in this thesis consist of.

One downside to regular Recurrent Neural Networks, is that if the sequence of data is long, the prediction will most likely be off if something that was for example typed in the beginning of a long text is a dependency for a prediction four chapters later, like the location of the main character. The workaround for this is something called *Long Short-Term Memory*

Figure 2.1: Neural network

Recurrent Neural Network (LSTM RNN), and is the idea of having additional logic to avoid the prediction model forgetting important facts.

Convolutional Neural Networks (CNN)

A Convolutional Neural Network, or *CNN* for short, can be used for identifying patterns in data, which can be used for predictions. A common use case for a CNN is image recognition. This is where you train your model to be really good at identifying objects in images, for example the difference between cats and dogs. Then you can input a completely different image to the model, and it will output whether the image is of a cat or a dog. This type of CNN is two-dimensional because an input image is really a two-dimensional array of pixel values, and it is most common to have a 2D CNN. Another way of constructing a CNN, is one-dimensionally, which can be useful for *one-dimensional* data, for example sensor data from gyroscopes or accelerometers [9].

	number	days	gender	age	afftype	melanch	inpatient	edu	marriage	work	madrs1	madrs2
0	condition_1	11	2	35-39	2.0	2.0	2.0	6-10	1.0	2.0	19.0	19.0
1	condition_2	18	2	40-44	1.0	2.0	2.0	6-10	2.0	2.0	24.0	11.0
2	condition_3	13	1	45-49	2.0	2.0	2.0	6-10	2.0	2.0	24.0	25.0
3	condition_4	13	2	25-29	2.0	2.0	2.0	11-15	1.0	1.0	20.0	16.0
4	condition_5	13	2	50-54	2.0	2.0	2.0	11-15	2.0	2.0	26.0	26.0

Figure 2.2: Demographics about participants (5 first rows)

	timestamp	date	activity
0	2003-05-07 12:00:00	2003-05-07	0
1	2003-05-07 12:01:00	2003-05-07	143
2	2003-05-07 12:02:00	2003-05-07	0
3	2003-05-07 12:03:00	2003-05-07	20
4	2003-05-07 12:04:00	2003-05-07	166

Figure 2.3: Activity measurements (5 first rows)

2.5 How can machine learning help people with bipolar disorder?

The usage of machine learning in the medical fields is growing exponentially these days. There are so many use cases of machine learning, and of course it can help in the bipolar field too! Let's say bipolar patients had a device that measured their heart rate among other things 24 hours a day could feed the data into a machine learning model that could give the user live feedback on which bipolar state they are currently in. I think that would be very useful, for both the patients and doctors/nurses. Another use case could be if medical institutions could know in advance how many new bipolar patients to expect the next day.

I believe that using machine learning in this field of study could help a lot of people get through their depression or mania, and potentially get rid of the condition completely.

2.6 The dataset

The dataset we will use in this project [8] was collected for another study for motor activity in schizophrenia and major depression. With the data about schizophrenia stripped out, this dataset is sufficient for my thesis. It contains activity level data for 23 bipolar and unipolar patients, and 32 non-depressed contributors. From now on, I will refer to the bipolar/unipolar group as the *condition group*, and the non-depressed group as the *control group*. This is also done in the dataset details [6].

The dataset is in two parts. One part includes the demographics of each participant 2.2. For the control group this only includes the number of days they were collecting data, their gender (1 for female and 2 for male) and age. For the condition group, it also includes their affliction type (1 for bipolar type II, 2 for unipolar depressive, 3 for bipolar type I), melanch (1 for melancholia, 2 for no melancholia), inpatient (1 for inpatient, 2 for outpatient), edu (education in years), marriage (1 for married / cohabiting, 2 for single), work (1 for working / studying, 2 for unemployed/sick/pension), madrs1 (MADRS score when measurement started) and madrs2 (MADRS score when measurement stopped) [6]. MADRS score (Montgomery-Asberg Depression Rating Scale) is used to grade the current severity of an ongoing depression [6].

The second part includes sensor data about the condition group and control group, as one file for each person 2.3. These files are in two folders: "condition" and "control" (for the two groups respectively), and one file for each person is inside the folders (filename is "GROUP_X.csv" where X is their id and GROUP is either condition or control. Inside the files, there is a list of activity measurements for every minute of the data collection period.

2.7 Challenges and ethical concerns

In most projects in the medical fields, there are going to be ethical concerns and challenges with privacy. What happens if someone that are not authorized for the data gets access to it? What if the database gets hacked? With new regulations (GDPR), which basically means that users have the right to be "forgotten". However, in this project all data is anonymized (only referenced by an id), so there will be no persons mentioned. If the dataset were not to be anonymized, and the patient's names were in it, things could get problematic if it got into the wrong hands.

Part II

The project

Chapter 3

Planning and preparing data

3.1 Goals

Our goal in this thesis is to create machine learning models for three different tasks:

- Classify whether a participant belongs to the **control** group or **condition** group.

- Classify a participant's depression class (by MADRS score).

- Predict a participant's MADRS score.

The model is going to be a One-Dimensional Convolutional Neural Network. We will solve all these using each participant's activity data as input, and we will create a model that, with few changes, can be used for all three goals. For the different goals, only the last few layers (preferably only the output layer) should be changed. For the first goal, classifying **control** or **condition** group, the output data should be a matrix with two columns (**control**, **condition**), with a 1 in one of the columns and a 0 in the other for each row.

But first we wanted to see if these problems could be solved with simple regression. The idea was to simply throw in the columns from the demographics dataset 2.2. We did not expect much from this, as there are only 55 rows in the table. Anyone having a little bit experience with machine learning will know that this is not nearly enough data. But we wanted to do it regardless, and see how a simple and stupid model performed. Doing this, we established some sort of benchmark for performance; the Convolutional Neural Network model had to *at least* better than this one.

3.1.1 Learning experiments

We needed to learn more about CNNs. CNNs are used in image recognition, so we proceeded to implement one. We found a tutorial on how to make a 2D CNN for classifying cats and dogs from images [12], and thought it would be a good way to learn.

It was both a fun and informative experience implementing this. Especially when we extended the script to allow an image url to predict on. Then we could browse for any image of a cat or a dog, and find out if the model could handle it (in most cases it did!). We even tried inputting images of humans to the model for fun. This experiment resulted in a lot of motivation for our task.

However as mentioned before, our data is one-dimensional, so a two-dimensional CNN would not be useful.

> *A 1D CNN is very effective when you expect to derive interesting features from shorter (fixed-length) segments of the overall data set and where the location of the feature within the segment is not of high relevance. This applies well to the analysis of time sequences of sensor data (such as gyroscope or accelerometer data).* [9]

To learn more about 1D CNNs, we followed a tutorial [9], which used a dataset containing accelerometer data from a smartphone on the participants waists. The goal for this CNN is to predict what a given person is doing at the time, given the accelerometer data for that time slice. What the given person is doing is one of the following:

- Standing

- Walking

- Jogging

- Sitting

- Upstairs

- Downstairs

As we followed the tutorial and implemented the model, we learned a lot about how 1D CNNs work and how we should structure our own data.

However, we also learned where our dataset could provide more data. What if the dataset contained the current mental state of the bipolar patient? Then someone could make some automated system that always can tell a patient whether they are normal, manic or depressive. However data collection for this kind of task would be difficult because we can't always know what the patient thinks, nor does the patient themselves. The "tutorial" dataset is different because it is easy to differentiate physical states of the body like standing or walking.

3.2   Preparing Input Data

3.2.1   Input data

As we said before, we wanted the input data to be exactly the same for each goal, since we wanted to use a similar model on all three. The tutorial [9]

sliced up the measurements with overlap, and labelled the slices. We are also did this.

We created a list where for each participant in the demographics table 2.2, measurements for *N* hours were grouped. Another choice we learned from the tutorial was to overlap the sequences, so we made the next group of *N* hours start *M* hours after, and not *N* hours after the group before, as one might think. When this list was complete with sequences from all participants, we had to **reshape** it so that it could fit into a neural network. We ended up with a feature list (which we called **segments**), where each element was a list of activity measurements for 4 hours:

    segments[0] = [[0], [143], [0], [20], [166], [160], [306], [277], [439], ...]

3.2.2 Output data

A second list was created simultaneously, where the value here was different on each goal.

Classifying participant group

For classifying control / condition group, this list was built to contain the values **0** or **1** for the labels **[CONTROL]** and **[CONDITION]**, which was chosen according to the group the participants were in. Using a helper function from Keras, **to_categorical**, we transformed this list of labels into the matrix we described. Transforming the values to a categorical matrix is required for the neural network that we ended up building, to be able to select a *category* for the result. This list, which we called **labels**, can look like this:

    labels[i] = [0, 1]

Meaning that segment *i* is labeled as **[CONDITION]**.

Classifying depression

Here we wanted to classify which depression class a participant belongs to, and as described in the background part about Bipolar Disorder, we divide MADRS scores into some *cutoff* points, which we will use as *classes* in our prediction:

- 0-6: normal

- 7-19: mild depression

- 20-34: moderate depression

- >34: severe depression

So instead of labelling the segments as **[CONTROL]** or **[CONDITION]**, we label them as 0, 1 or 2 (we ignore 3 as there is no participant with MADRS score above 34). One element in **labels** in this case could be, after making it a categorical matrix:

    labels[i] = [0, 1, 0]

Meaning that segment *i* is labelled as **[MILD DEPRESSION]**.

	Actual: Negative	Actual: Positive
Predicted: Negative	True Negative (TN)	False Negative (FN)
Predicted: Positive	False Positive (FP)	True Positive (TP)

Table 3.1: Confusion Matrix

	Actual: Control group	Actual: Condition
Predicted: Control group	95	5
Predicted: Condition	7	**93**

Table 3.2: Confusion Matrix Example: Control vs Condition group

Predicting MADRS Score

Instead of classifying one of three or one of two classes which we have done earlier, this time we want to predict the actual MADRS score value. Creating the output data for this goal is easier, we simply add the MADRS score for the corresponding participant to the list. For example:
  scores[i] = [18]

3.3   Performance Metrics

When you are done with a training session for a machine learning model, you need to use different metrics to be able to tell how the model performed based on some testing data. To begin with, you will focus on one performance metric which is called accuracy. This one makes sense to the human brain; telling a friend that has no experience in machine learning or data science that you have made your computer able to predict something with 98% accuracy is actually something that they would understand. Other metrics include *confusion matrix*, *precision*, *Recall*, *specificity* and *F1 score*.

3.3.1   Confusion Matrix

Table 3.1 shows a *confusion matrix*. It is a visual metric for classification models in machine learning and is the basis for the other performance metrics. It can tell you how well your model is performing by having correlation values for the different classes. Let's say we have 200 samples in our test data to use on our model that classifies control vs condition group. A *confusion matrix* for a good model would look like table 3.2, with high numbers in **True Positive** and **True Negative** and as low numbers as possible in **False Positive** and **False Negative**. Having a high number in **True Positive** means that the model is able to classify that a participant is in the condition group if he or she actually is, and having a high number in **True Negative** means that the model is able to classify that a participant is in the control group if this is the case. The other cases, **False Positive**

and **False Negative**, is where the model made a wrong classification, and therefore as close these numbers are to zero the better our model is.

3.3.2 Accuracy

'Accuracy is a good measure when the target variable classes in the data are nearly balanced.' ([17])

When calculating the *accuracy*, we sum up the correct predictions and divide that with the total number of predictions ($\frac{TP+TN}{TP+TN+FN+FP}$). For our example (3.2), the *accuracy* would be $\frac{93+95}{93+95+5+7} = 94\%$. It is a good metric to use for our example because the number of samples for each variable class is well balanced ($93 + 5 = 98$ samples where **condition group** was the correct option, and $7 + 95 = 102$ samples where **control group** was correct).

Terrible use of the *accuracy* metric would be when one of the classes strongly dominates the samples. For example, if a model predicts **cancer** vs **no cancer**, and the samples contain 5 people with cancer and the 95 remaining people do not. The model would be terrible at predicting cancer and still have an accuracy score of 95% [17].

3.3.3 Precision

This performance metric operates entirely on the predicted positives, and it tells us how many **true positives** there is among **predicted positives** ($\frac{TP}{TP+FP}$) [17].

Precision is better to use on *unbalanced* classes than accuracy. The *cancer vs no cancer* example, assuming it predicts no-one to have **cancer**, would yield a precision score of $\frac{5}{5+95} = 5\%$. And our *control vs condition* example would result in a precision score of $\frac{93}{93+7} = 93\%$.

3.3.4 Recall

Recall is another useful performance metric. It tells us the relationship between **true positives** and **actual positives**, for example how many participants classified to be in the condition group there were among total participants in the condition group.

The calculation of recall is done by dividing **true positives** by **true positives + false negatives** ($\frac{TP}{TP+FN}$), which translates into $\frac{93}{93+5} \approx 95\%$ for table 3.2.

Choosing a metric to use from *precision* or *recall* depends on your goal. Try to achieve close to 100% *recall* if you want to reduce **false negatives**, and likewise with *precision* if you want to reduce **false positives** [17].

3.3.5 Specificity

As *recall* operates on **actual positives**, *specificity* is the exact opposite metric. It tells us the relationship between **true negatives** and **actual negatives**. So if your goal is to reduce **false positives**, specificity is a valid choice.

Specificity is calculated by dividing **true negatives** by **true negatives + false positives** ($\frac{TN}{TN+FP}$). For table 3.2, the *specificity* score equals $\frac{95}{95+7} \approx$ 93%.

3.3.6 F1 Score

The metrics that we have described in this section are all useful when determining whether your classification model is good enough. But the relationship between *recall* and *precision* and knowing when to use which can be confusing, at least if the different classes are somewhere between completely unbalanced and perfectly balanced (for example a 35% split).

Therefore another metric called *F1 Score* was created, which gives us a balanced value combining *recall (R)* and *precision (P)*. The basic idea is to return the *mean* value of the two scores (F1 = $\frac{P+R}{2}$), but that would not be balanced if one score is much lower than the other. F1 score actually uses something called *harmonic mean* instead of the standard *arithmetic mean*, and is calculated as $2 \cdot \frac{P \cdot R}{P+R}$ [17].

Following this formula, the F1 score for confusion matrix 3.2 becomes:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} = 2 \cdot \frac{0,93 \cdot 0,95}{0,93 + 0,95} \approx 94\%$$

3.3.7 Classification Report

A machine learning framework for Python, *sklearn*, includes a package with functions to calculate most of these scores. Simply import and use them like this:

```
from sklearn.metrics import confusion_matrix,
                            precision_score,
                            recall_score,
                            f1_score,
                            classification_report

# ... set up model, make predictions ...

print(confusion_matrix(y, y_class_predicted))
print(precision_score(y, y_class_predicted))
print(recall_score(y, y_class_predicted))
print(f1_score(y, y_class_predicted))
print(classification_report(y, y_class_predicted))
```

Source Code 3.1: sklearn metrics

The classification report returns a matrix (see table 3.3) with *precision*, *recall*, *F1 score* and another column called *support* (simply how many samples of data there are for this class). The rows describe each class used for prediction [4].

	precision	recall	f1-score	support
class 0	0.05	1.00	0.67	1
class 1	0.00	0.00	0.00	1
class 2	1.00	0.67	0.80	3
micro avg	0.60	0.60	0.60	5
macro avg	0.50	0.56	0.49	5
weighted avg	0.60	0.60	0.60	5

Table 3.3: Classification Report

Chapter 4

Implementing the project

4.1   Regression

Regression is used a lot in statistics, where its main purpose is to predict a target value based on independent predictors and is useful to find cause and effect relationships between variables [10]. One experiment that might be interesting is to do linear regression on some of the columns in the demographics dataset 2.2, in order to see whether there are columns that tell more about the participant's group (control or condition).

Not all of the columns are relevant though, for example, **number** is unique for each participant so it does not make sense to use. The column **melanch** also does not make sense to use, as there is only one participant with Melancholia. **Inpatient** cannot be used because the participants in the control group are not patients, and can't be either inpatient or outpatient. The same goes for **edu**, **work** and **marriage**, but we don't have these data values for control group participants. We don't have their **afftype** or any of the **MADRS** scores either, but we already know them: **afftype** should be 0 (not bipolar) and both **MADRS** scores should be 0 (not depressed).

We used the **afftype** column as a target in the regression. This way we could classify whether a participant is in the control or the condition group, by setting the **afftype** value to either 0 or 1 instead of 0, 1, 2 or 3 (values above 1 are reduced to 1), and run regression on all of the remaining columns one by one:

- **Gender**

- **Age**

- **Days**

- **MADRS1**

- **MADRS2**

The model for this task is really simple. We created a sequential model 4.1 with one input layer and one output layer. The input is a **dense** layer which takes one value (the value of the current column), activates using

```
1    def regression_model():
2        model = Sequential()
3        model.add(Dense(5, input_dim=1, activation='relu'))
4        model.add(Dense(1))
5
6        model.compile(loss='mse', optimizer='adam', metrics=['mse'])
7        return model
```

Source Code 4.1: Regression Model

Figure 4.1: Feature Detector / Filter "sliding" over input data

relu and outputs 5 neurons. In the output layer, we simply activate using a **linear** function (default activation function when none are specified) and return one value: the prediction. The model is compiled with the loss function **Mean Squared Error** and the optimizer **Adam**. More on loss functions and optimizers will be described later.

Since we were predicting the **afftype** of a participant, which is a binary value (0 or 1) and a prediction from the model yields a value between 0 and 1, we simply rounded the prediction value to the nearest integer after running predictions on the test data. Doing it this way instead of making a classification model, makes the **loss** value more useful as a metric while training rather than accuracy.

When testing the model, we wanted to achieve an accuracy of 100% when using the **MADRS** scores. This is because of how easy it is using a simple check: if the MADRS score is 0, then the participant is in the control group. If not, the participant is in the condition group. We didn't really need to use machine learning on these columns but was interesting to train the model to find this relationship without telling it the simple rule. For the other columns, we didn't really know what to expect; maybe there was a relationship, maybe not.

4.2   1D Convolutional Neural Network

4.2.1   Convolution

The main ingredient in a CNN is the *convolutional* layer. It is responsible for, as you may have guessed, the convolution. It consists of a number of *filters* 4.1, which are the sliding windows that go through the input data. They are also called *feature detectors*, and using 100 of them means that the layer can detect 100 features. The size of a filter is called *kernel size*. The

output of convolutional layer is a matrix with one column for each filter, and one row for each step in the convolution. How many steps there are, is given by the length of the input data (also called *height*) minus the kernel size plus 1.

4.2.2 Creating the Model

Following the tutorial on 1D CNNs [9], we came up with two models. One model for the first two goals was built for classification 4.5, and for the third goal we came up with another model built for prediction 4.6. We built the models a lot like how it was in the tutorial, but we also had to tweak some of the parameters to make the model work on our dataset.

To make a convolutional neural network, you need some *convolutional* and *pooling* layers. Which layers you add in between and the ordering of them, together with the parameters you pass to the layers, is what makes the model perform differently. This, of course, requires your input data to be well structured.

Classification

The following model 4.5 is used to achieve our first two goals; classifying whether a participant belongs to the **control** or **condition** group, and classifying the participant's **depression class**. The only difference between these goals is the number of classes we are trying to classify, and therefore only the output layer needs to be changed.

1. We start by defining a **Sequential** model. This is an easy to understand and readable way of defining a model. Alternatively we could use a **functional** model, which gives more control of inputs and outputs of the layers. A **functional** model would be useful if we wanted to debug and optimize each layer within the model.

2. **Reshape**: In the first layer we need to reshape the input data so that it becomes an *X* by 1 matrix, where *X* is the length of each segment. The reason for this reshape step is because the next layer (**Conv1D**) requires the input to contain (*batch, steps, channels*). *Batch* will be set to **None**, *steps* will be the segments, and *channels* will be 1 (because we only have one measurement value for each minute).

3. **Conv1D**: This is the first *convolutional* layer, where the required parameters are how many *filters* you want, and how big the *kernel* should be. As in the tutorial, we will also start by using 100 filters and a kernel size of 10. Having less or more might impact the performance, which we will experiment on later. There are many different parameters that you can use on a layer like this, for example *padding* and *strides*, but using the default values is good for now. This is also something that we can experiment with later. However using the default parameters, the output of this layer will be a $(X - 10 + 1) \times 100$ matrix, and X is the length of each segment here

as well. All the convolutional layers in this model will be using the activation function *relu*.

4. **Conv1D**: The second convolutional layer will look exactly like the first one, and the output will be a $(X - 10 + 1 - 10 + 1) \times 100$ matrix.

5. **MaxPooling1D**: Pooling is important in a convolutional neural network to reduce complexity [9]. The basic idea of *max pooling* is to reduce to only the maximum value for each *window* of size $N \times N$. We are using 2 as window size (N), resulting in matrix that is half the size of the input: $\frac{X-10+1-10+1}{2} \times 100$. Pooling may also help reduce *overfitting*, which is when the model learns its own training data too well, and performs worse on unseen data.

6. **Conv1D**: Two more convolutional layers are added, and after these the input to the next layer will be a matrix of size $\left(\frac{X-10+1-10+1}{2} - 10 + 1 - 10 + 1\right) \times 160$.

7. **GlobalAveragePooling1D**: This is another pooling layer, and this one takes the average of weights within the network instead of the maximum. After doing this, the matrix will be of size 1×160.

8. **Dropout**: Dropout is used to reduce overfitting, by randomly ignoring units in the neural network.

9. **Dense**: The final layer in the model is a dense layer (fully connected) which reduces the matrix from 160 values to either 2 or 3 (for goal one and two), with the activation function **softmax**. Then the output (a 1 in one of the neurons) is mapped to the corresponding label.

After creating the model, we *compile* it as a model that is ready to be *fit* to the dataset, giving it a **loss function** and an **optimizer**.The loss function is the function that evaluates how well the model *models* the given data [5], and the optimizer is the function that attempts to lower the output of the loss function.

For the first two goals, the loss function *categorical crossentropy* calculates a probability over the number of classes that are supplied (number of classes equals the number of neurons in the output layer).

For the optimizer, there are many different choices available, but to keep things simple, we will start out using an optimizer called **Adam** for training all of our models. Using a different optimizer can make the model fit to the dataset faster or slower.

Prediction

To make the model work for our third goal, where we will predict the actual value for the participant's MADRS score, we have to change a few layers. To simplify it a bit, we removed two of the *Conv1D* layers, and after the *GlobalAveragePooling1D* layer, this is applied:

Not bipolar	Bipolar
0	1
1	0
0	1
1	0
1	0
0	1
0	1
1	0

Table 4.1: Categorical Labels

8. **Flatten**: The matrix needs to be flat (one dimensional) before proceeding to the final layers.

9. **Dense**: A dense layer with 10 neurons, and *relu* as the activation function.

10. **Dense**: The output layer is a dense layer of size 1, since we are predicting *one* value. Also, the activation on this layer is going to be *linear* instead of softmax.

We compile this model with the loss function *Mean Squared Error*, which is measured as the average (mean) of squared difference between predictions and actual observations ($loss = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$) [5]. The optimizer is the same as before.

4.2.3 Creating Input and Output Data

One function (*create_segments_and_labels()* 4.4) is responsible of creating the data that is sent into the neural network. We start by defining a *segment length L*, which is how much data (minutes) we want inside each segment. We will experiment with the value of *L*, but let's say we use segments of 4 hours at a time ($L = 4 * 60 = 240$). Next we need a value for how many values to step after each iteration, *S*. We will keep this value at one hour, meaning $S = 60$. Between the different goals, this function will only be different in how it determines the output values. The code in the Source Code section 4.4 is simplified to only generate input and output for classifying control / condition groups (goal one).

- First we read the *global* dataset, where we find each participant and whether they are bipolar or not. As there is no *afftype* value for non-bipolar participants, we simply set this to 0. This is fine because the other possible values are 1, 2 and 3.

- Then we iterate over all participant activity data files:

29

– Append a **segment** that is of length L to the list of segments (using default parameters in the *create_segments_and_labels* function 4.4).

– Append the target value for the current goal, so:

  * Append a 1 or 0 for classifying control/condition group.
  * Append a 0, 1 or 2 for classifying depression class (normal/mild/moderate).
  * Append the MADRS score (after measurement period) when the goal is to predict MADRS score.

– Skip S indexes, and repeat until we have added all segments.

• Make the list of labels into a *categorical* 2D matrix 4.1 with a **1** in only one of the columns, instead of a single-dimensional list. This is only needed in the first two goals, for the **softmax** activation function.

• Also we need the list of segments to be restructured. We do this with the **reshape** function, and after this the data is ready to be passed into the neural network.

Train and test data

One last step is done before we can start training the model. We need to split the data in two parts; training and test data. This way we can calculate the performance of the model after training has finished, and also prevent overfitting by evaluating on data that is unseen to the model.

The function *train_test_split* from the *sklearn* package is useful here, where you input the segments and labels, plus how large you want the training and test sets to be (number between 0 and 1, which determines the size of the test partition). The function also randomizes the data, preventing model to accidentally learn something that is correct for segments in a row that also are chronologically in order. After calling the function 4.2, you end up with two arrays for input data (*X_train, X_test*), and two arrays for output data (*y_train, y_test*). In this case the *test_size* is set to 0.4, meaning that the test sets contains 40% of the total dataset and the remaining 60% are in the training sets.

Source Code 4.2: Sklearn train and test split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(segments,
                                                    labels,
                                                    test_size=0.4)
```

Another option is to use something called *K-fold cross-validation*. It works by splitting the dataset into K train and test sets, and for each of them, train the model on the training set and validate on the test set. This is a good way of checking if your model's performance truly is independent on which data

it trained on. The higher number of splits (*K*) means less data to train on, so you need to keep that in mind (also applies to *train_test_split* if you set the *test_size* too high). Sklearn has an implementation of *K-fold* splitting of data, called *StratifiedKFold*. For example, code 4.3 shows how you can use *StratifiedKFold* to perform a 3-fold cross-validation in your project ($K = 3$).

Source Code 4.3: Sklearn K-Fold

```python
from sklearn.model_selection import StratifiedKFold

segments, labels = create_segments_and_labels(...)

skf = StratifiedKFold(n_splits=3, shuffle=True)

for train_index, test_index in skf.split(segments, labels):
    X_train, X_test = segments[train_index], segments[test_index]
    y_train, y_test = labels[train_index], labels[test_index]

    model = create_model(...)
    model.fit(X_train, y_train, ...)
    results = model.evaluate(X_test, y_test)

    # do something with results...
```

4.3   Optimizing the model

Out of the box, we didn't think that the model was going to perform perfectly. Therefore we needed to experiment with each parameter that we passed to the model, and find the best ones. We had some ideas of what to change:

- Use different segment lengths
- Tweak hyper-parameters

The first idea, using different segment lengths, was the one we thought was going to impact the results the most. Having more data inside each segment will give the neural network more opportunities to learn features, and then be better at its job of either classifying or predicting something. It each epoch of training took more time though, since each layer had to process more data. Whether it was worth waiting the extra training time when increasing the segment length, and if it was one specific segment length that was superior to the others, is something to be discuss later in another section.

Hyper-parameters are all the *higher-order* parameters we compile/train the model with. They are different from the parameters learned by training the model, and need to be fixed for one session of model training. Finding the perfect ones can be crucial for a well performing model. Hyper-parameters that we use in our models include:

- **Optimizer and learning rate**
  The **optimizer** function (as said earlier) is important to lower the loss value when fitting the dataset to the model. Different optimizers have different input parameters, and the *learning rate* is one that all optimizers use. Tweaking the learning rate can yield better results, but the default learning rate for the chosen optimizer is always a good starting point because it's what the author of the optimizer set as default.

- **Epochs**
  Defines how many iterations of training that are to be executed, and in most cases more epochs yield better results up to a certain point.

- **Batch size**
  This is how much data that is processed at the same time each epoch, and the best batch size to use can be completely different on two different models.

- **Train/test split size**
  When building a neural network, we want as much data as possible to both train and to test on afterwards. We can't use all the data in both cases, so the most balanced split size needs to be determined.

4.4 Source Code

4.4.1 Create Input Data

Source Code 4.4: Read Dataset, Create Input Data

```python
def create_segments_and_labels(segment_length, output_classes=2, step=60):
    scores = pd.read_csv('scores.csv')
    scores['afftype'].fillna(0, inplace=True)

    segments = []
    labels = []

    for person in scores['number']:
        p = scores[scores['number'] == person]
        df_activity = pd.read_csv(f'{person}.csv')

        for i in range(0, len(df_activity) - segment_length, step):
            segment = df_activity['activity'].values[i : i + segment_length]
            segments.append([segment])

            if p['afftype'].values[0] == 0:
                labels.append(0)
            else:
                labels.append(1)

    segments = np.asarray(segments).reshape(-1, segment_length, 1)
    segments = segments.reshape(segments.shape[0], segment_length)

    labels = to_categorical(np.asarray(labels), output_classes)

    return segments, labels
```

4.4.2   The Models

Source Code 4.5: 1D CNN Model for Classification

```
1   def create_classification_model(segment_length, output_layers):
2     model = Sequential()
3
4     model.add(Reshape((segment_length, 1), input_shape=(segment_length,)))
5     model.add(Conv1D(100, 10, activation='relu', input_shape=(segment_length, 1)))
6     model.add(Conv1D(100, 10, activation='relu'))
7     model.add(MaxPooling1D(2))
8     model.add(Conv1D(160, 10, activation='relu'))
9     model.add(Conv1D(160, 10, activation='relu'))
10    model.add(GlobalAveragePooling1D())
11    model.add(Dropout(0.5))
12    model.add(Dense(output_layers, activation='softmax'))
13
14    model.compile(loss='categorical_crossentropy',
15                  optimizer='adam',
16                  metrics=['accuracy'])
17
18    return model
```

Source Code 4.6: 1D CNN Model for Prediction

```
1   def create_prediction_model(segment_length):
2     model = Sequential()
3
4     model.add(Reshape((segment_length, 1), input_shape=(input_shape,)))
5     model.add(Conv1D(128, 2, activation='relu', input_shape=(segment_length, 1)))
6     model.add(MaxPooling1D(pool_size=2, strides=1))
7     model.add(Conv1D(64, 2, activation='relu'))
8     model.add(GlobalAveragePooling1D())
9     model.add(Flatten())
10    model.add(Dense(10, activation='relu'))
11    model.add(Dense(1))
12
13    model.compile(loss='mean_squared_error',
14                  optimizer='adam',
15                  metrics=['mse'])
16
17    return model
```

Chapter 5

Training the models

For training large neural networks you often need expensive hardware if you want to do it on your own. The way to do it these days is to use cloud services such as Google or Amazon to run the machine learning once you have prepared the dataset and the model. These services are often paid by the hour, but it's not expensive as long as you remember to shut it down after use, and only run it when you really need to.

Our dataset, however, only contains 55 participants and only a few days of activity measurements for each participant, so using a service to train the model would not be necessary. We decided to train them on one of our own personal computers, with an Nvidia GTX 1070. Our guess is that this GPU will do the job just fine.

5.1 Regression

We ended up training the regression model 4.1 for 1000 epochs using a batch size of 16. Because of the simplicity of the model, 1000 epochs did not take that much time and the training loss graph 5.1 shows that more epochs won't yield any better results. The loss does not reduce any more after the first 100 epochs for any of the columns. As we said earlier, we wanted the model to predict flawlessly based on the MADRS scores, and the loss results from training these are clearly better than for the other columns.

We used the **train_test_split** function to create training and testing data, and because of how few rows there are in this dataset, we used an 80/20 split for training and testing data. 20% of the dataset in the test data seemed to be good enough for this experiment because we wanted as many rows as possible to train on. Note that the data elements that were in the test and train splits were different for each column.

5.1.1 Results

The confusion matrices 5.2 displays the results when running the prediction on the test data (and then using it as classification). We said we wanted 100% on all metrics for the MADRS score columns, and this is clearly achieved because the model guessed everything right. The other columns
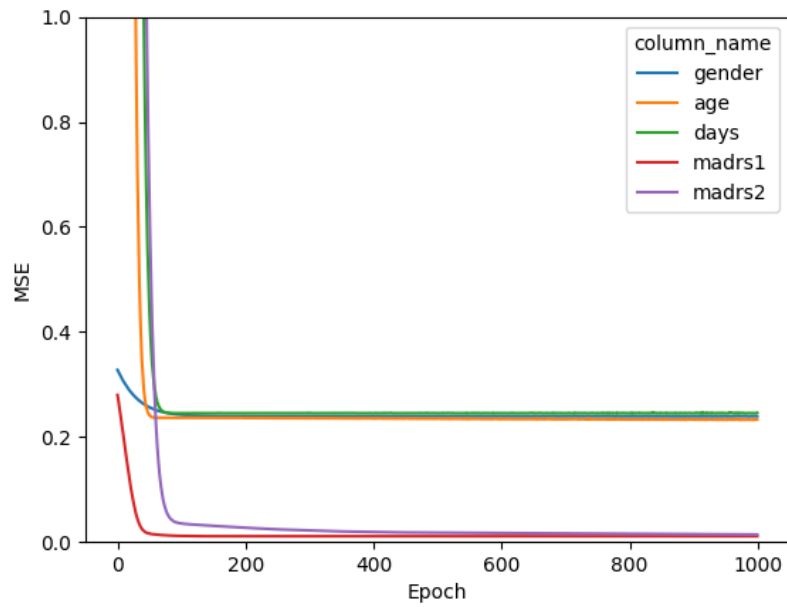
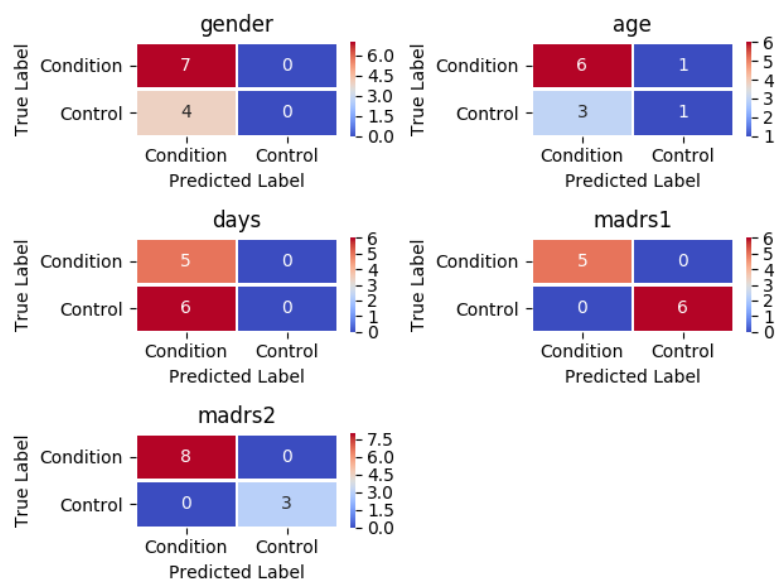Figure 5.1: Regression Training Loss (MSE) by Epoch

Figure 5.2: Confusion Matrices for Regression

have these performance scores (using **condition** as positive and **control** as negative):

5.1.2 Accuracy

$A = \frac{TP+TN}{TP+TN+FN+FP}$

- **Gender**: $\frac{7+0}{7+0+4+0} = 7/11 \approx 63\%$

- **Age**: $\frac{6+1}{6+1+3+1} = 7/11 \approx 63\%$

- **Days**: $\frac{5+0}{5+0+6+0} = 5/11 \approx 45\%$

5.1.3 Precision

$P = \frac{TP}{TP+FP}$

- **Gender**: $\frac{7}{7+0} = 7/7 = 100\%$

- **Age**: $\frac{6}{6+1} = 6/7 \approx 86\%$

- **Days**: $\frac{5}{5+0} = 5/5 = 100\%$

5.1.4 Recall

$R = \frac{TP}{TP+FN}$

- **Gender**: $\frac{7}{7+4} = 7/11 \approx 63\%$

- **Age**: $\frac{6}{6+3} = 6/9 \approx 67\%$

- **Days**: $\frac{5}{5+6} = 5/11 \approx 45\%$

5.1.5 Specificity

$S = \frac{TN}{TN+FP}$

- **Gender**: $\frac{0}{0+0} = 0/0$

- **Age**: $\frac{1}{1+1} = 1/2 = 50\%$

- **Days**: $\frac{0}{0+0} = 0/0$

5.1.6 F1 Score

$F1 = 2 \cdot \frac{P \cdot R}{P+R}$

- **Gender**: $2 \cdot \frac{1 \cdot 0,63}{1+0,63} \approx 77\%$

- **Age**: $2 \cdot \frac{0,86 \cdot 0,67}{0,86+0,67} \approx 75\%$

- **Days**: $2 \cdot \frac{1 \cdot 0,45}{1+0,45} \approx 62\%$

Looking away from the results of **madrs1** and **madrs2**, because they were obviously going to be good, these performance scores tell us that the model is good at predicting that someone is in the condition group (see precision). Other than that, we can't really learn that much from them. Also, we have undefined specificity for both gender and days because we don't have any predictions for the values required to calculate them.

However, we learned a lot about machine learning and regression doing this experiment, and we look at it more like a warmup for the more complex models that we built for the next goals.

5.2 1D CNN: Control vs Condition Group

Next up was our goal to classify whether a participant belongs to the control group or the condition group. As we said in the section about optimizing the model, the segment length is what we thought was going to impact the result the most. To test this theory, we trained the model to fit input data created with segment lengths of 1, 2, 4, 8, 16, 24, 48 and 96 hours.

The input data was split using **train_test_split** so that the training data was 80% of the total and the rest was going to be used for testing after the training was complete. Of the training data, 40% was set to be used as validation data, which is what the model uses in each epoch to determine if the results are getting better or worse.

We did the training for each of the 8 sets for 10 epochs. The objective of this experiment was to find the best segment length to use, and not to train the models to be perfect, so 10 epochs seemed fine for this purpose. Our guess before we started with the experiment was that the more hours of data we used, the better, meaning that 96 hours of data in each segment was going to give us the best model (from what was possible with only 10 epochs). To keep it simple, we used a batch size of 16 throughout this experiment.

However, looking at the graphs for loss and accuracy 5.3 for how the training went epoch by epoch for each of the experiment's datasets, we noticed that the results were better when increasing the number of hours up to around 48, and it did not seem to be any better for 96 hours. It's even easier to see this in the evaluation graphs for loss and accuracy 5.3, where there is a straight line after 48 hours. We didn't know that this was true for better-trained models though, it could be that we needed to train for more epochs when the segments are above this size, which made sense.

To really find the best segment length, we needed to experiment with more epochs. We ran the same experiment again for 50 epochs, with 48, 72 and 96 hour long segments. However, from the results for that 5.4, it's clear that nothing more was achieved with segments longer than 48 hours. The classifications for the models trained to fit 48-hour segments you can see in the confusion matrices 5.5 are really close to perfect. The extra 40 epochs of training reduced the false positive classifications by a little bit, which was worth the time in our opinion as you want the value of wrong classifications to be as close to 0 as possible. For a relatively small dataset, we are very happy with a classification model with scores above 99% on all performance metrics previously discussed.

5.2.1 Performance metrics

Accuracy

$$A = \frac{TP+TN}{TP+TN+FN+FP} = \frac{3122+1577}{3122+1577+5+13} \approx 99,6\%$$
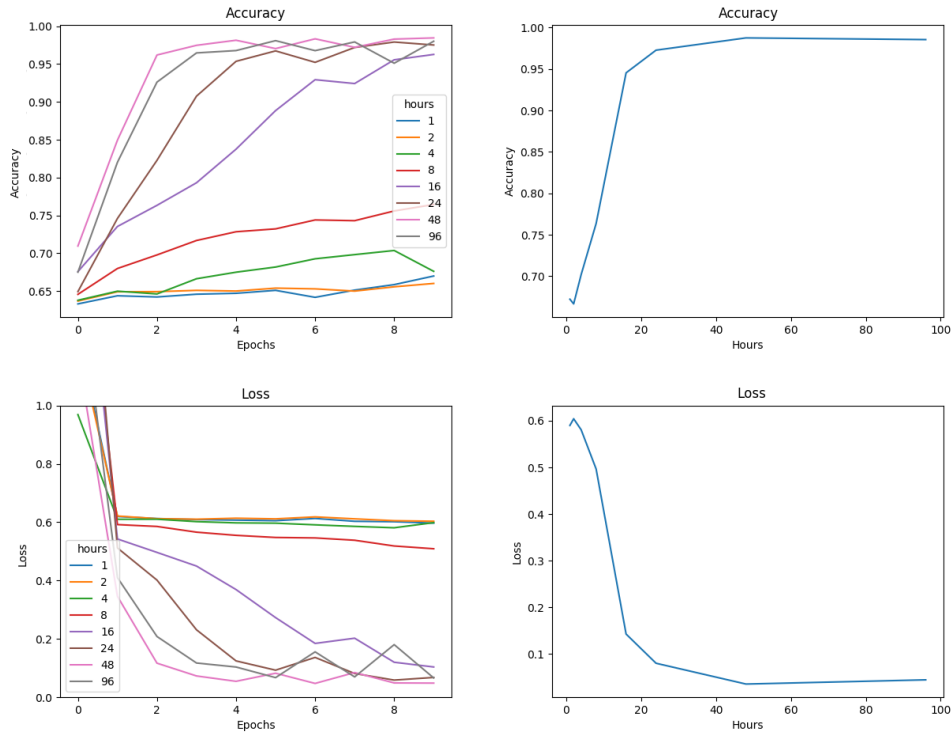
Figure 5.3: Training (left) and testing (right) results (10 epochs)

Precision

$P = \frac{TP}{TP+FP} = \frac{3122}{3122+13} \approx 99,6\%$

Recall

$R = \frac{TP}{TP+FN} = \frac{3122}{3122+5} \approx 99,8\%$

Specificity

$S = \frac{TN}{TN+FP} = \frac{1577}{1577+13} \approx 99,2\%$

F1 Score

$F1 = 2 \cdot \frac{P \cdot R}{P+R} = 2 \cdot \frac{0,996 \cdot 0,998}{0,996+0,998} \approx 99,7\%$
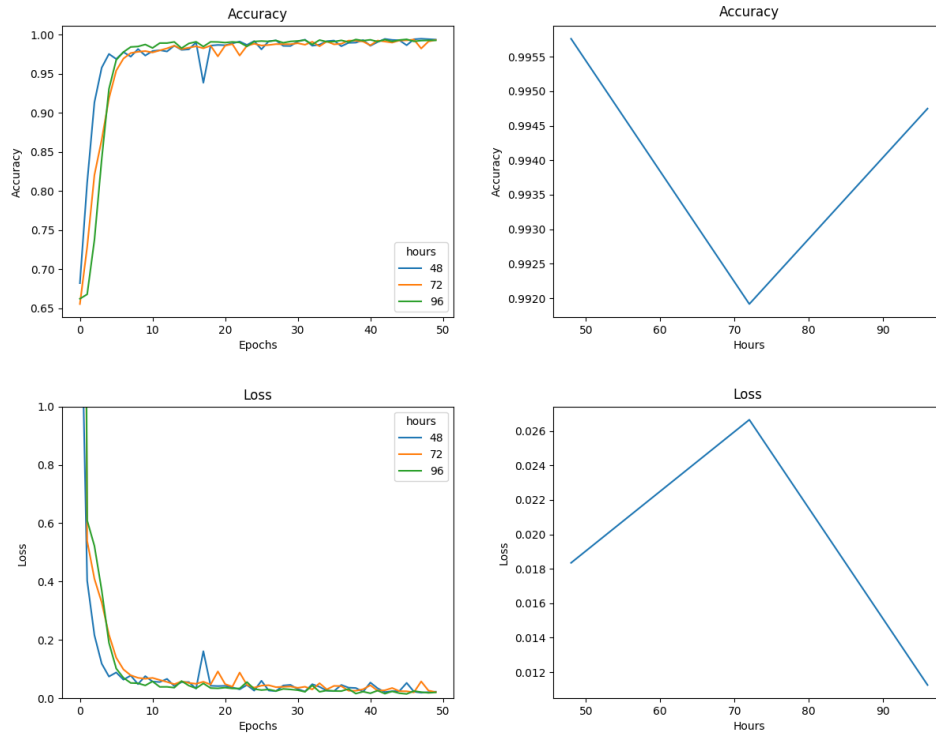
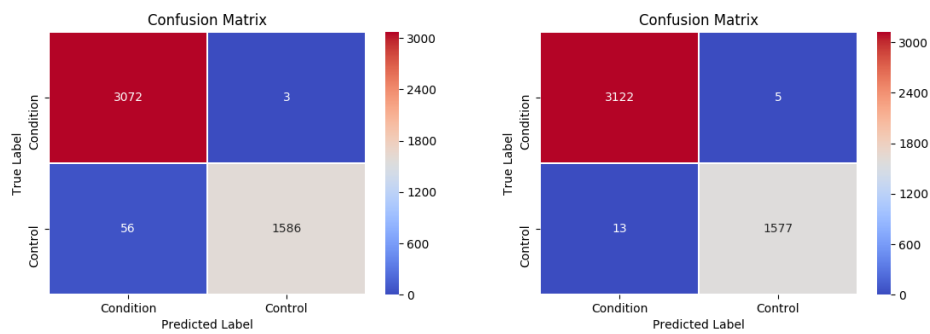Figure 5.4: Training (left) and testing (right) results (50 epochs)

Figure 5.5: 10 epochs (left) and 50 epochs (right) for 48 hour segments

Part III

Conclusion

Chapter 6

Results

Bibliography

[1] *Bipolar 1 Disorder and Bipolar 2 Disorder: What Are the Differences?* Accessed: 14.03.2018. URL: https://www.healthline.com/health/bipolar-disorder/bipolar-1-vs-bipolar-2.

[2] *Bipolar disorder*. Accessed: 14.03.2018. URL: https://familydoctor.org/condition/bipolar-disorder/.

[3] *Bipolar disorder statistics*. Accessed: 14.03.2018. URL: https://www.statisticbrain.com/bipolar-disorder-statistics/.

[4] *Classification Report*. Accessed: 07.02.2019. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html.

[5] *Common Loss functions in machine learning*. Accessed: 01.03.2019. URL: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23.

[6] *Dataset details*. Accessed: 08.05.2018. URL: http://datasets.simula.no/depresjon/.

[7] *Decision Tree*. Accessed: 08.05.2018. URL: http://www.saedsayad.com/decision_tree.htm.

[8] Enrique Garcia-Ceja et al. 'Depresjon: A Motor Activity Database of Depression Episodes in Unipolar and Bipolar Patients'. In: *Proceedings of the 9th ACM on Multimedia Systems Conference*. MMSys'18. Amsterdam, The Netherlands: ACM, 2018. DOI: 10.1145/3204949.3208125. URL: http://doi.acm.org/10.1145/3204949.3208125.

[9] *Introduction to 1D Convolutional Neural Networks*. Accessed: 07.12.2018. URL: https://blog.goodaudience.com/introduction-to-1d-convolutional-neural-networks-in-keras-for-time-sequences-3a7ff801a2cf.

[10] *Introduction to Machine Learning Algorithms: Linear Regression*. Accessed: 15.03.2019. URL: https://towardsdatascience.com/introduction-to-machine-learning-algorithms-linear-regression-14c4e325882a.

[11] *Keras documentation*. Accessed: 15.03.2018. URL: https://keras.io.

[12] *Loading in your own data - Deep Learning basics with Python, TensorFlow and Keras p.2*. Accessed: 07.12.2018. URL: https://pythonprogramming.net/loading-custom-data-deep-learning-python-tensorflow-keras/?completed=/introduction-deep-learning-python-tensorflow-keras/.

[13] *Machine Learning for Humans, Part 2.1: Supervised Learning.* Accessed: 11.04.2018. URL: https://medium.com/machine-learning-for-humans/supervised-learning-740383a2feab.

[14] *Machine Learning for Humans, Part 2.2: Supervised Learning II.* Accessed: 11.04.2018. URL: https://medium.com/machine-learning-for-humans/supervised-learning-2-5c1c23f3560d.

[15] *Neural Networks.* Accessed: 21.05.2018. URL: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/.

[16] *Neural Networks Introduction.* Accessed: 21.05.2018. URL: https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc.

[17] *Performance Metrics for Classification problems in Machine Learning.* Accessed: 07.02.2019. URL: https://medium.com/greyatom/performance-metrics-for-classification-problems-in-machine-learning-part-i-b085d432082b.

[18] *Random forest.* Accessed: 21.05.2018. URL: https://medium.com/Synced/how-random-forest-algorithm-works-in-machine-learning-3c0fe15b6674.

[19] *What is supervised learning?* Accessed: 11.04.2018. URL: https://searchenterpriseai.techtarget.com/definition/supervised-learning.

[20] *Why are GPUs well-suited to deep learning?* Accessed: 15.03.2018. URL: https://www.quora.com/Why-are-GPUs-well-suited-to-deep-learning.