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/

Printed: Reprosentralen, University of Oslo

Abstract

Contents

List of Figures

List of Tables

Preface

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 Depression

2.1.1 Bipolar Disorder

Bipolar disorder is a disorder where a person experiences extreme mood swings. One day the person can feel amazing, and everything is fine, but the next day they feel like they do not belong anywhere in this universe. In general, one should be concerned about mood swings. It is, however, the extreme cases where the mind turns 180 degrees from day to day that is the main symptom of bipolar disorder. There is not 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 is not 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 would never have intended doing, like spending much money on items they do not need or abusing drugs/alcohol. They may also feel 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 do not belong here, or being guilty of something they may or may not have done. In some cases, depression may even end up with suicidality, where the person either spends much time thinking of death, or attempt suicide.

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].

2.1.2 Depression Rating: MADRS

Montgomery-Åsberg Depression Rating Scale, or MADRS for short, is a rating system for telling how depressed a patient is. Stuart A. Montgomery and Marie Åsberg designed it in 1979, and it is more sensitive to changes than the Hamilton Rating Scale for Depression (HRS) for patients that go through antidepressant medication. The process for

calculating a MADRS rating contains ten statements about the patient's behavior, where the topics are [17]:

- Apparent sadness
- Reported sadness
- Inner tension
- Reduced sleep
- Reduced appetite
- Concentration difficulties
- Lassitude
- Inability to feel
- Pessimistic thoughts
- Suicidal thoughts

The person doing the rating answers each question with a number between 0 and 6, where the higher the number, the more relevant the statement is for the patient. The numbers added together gives us the total MADRS score, which we split into four categories: normal/not depressed: 0-6, mild depression: 7-19, moderate depression: 20-34 and severe depression: 34-60 [9].

2.2 Machine learning

Machine learning is the field of computer science where we throw data into an algorithm and expect it to give answers to whatever the goal is, with as little work as possible. The process of performing machine learning was not as simple 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 tremendous and almost magical technology, but knowing how one should use it can be difficult without the experience. The amount of data needed to make an algorithm learn something makes it difficult to get started, and to be efficient when training the model on a large dataset, decent hardware is required. One can get away with using a CPU if they want to test machine learning on a small dataset, but if the goal is to build something useful, too much time is saved using a GPU.

The reason why GPUs are so much better than CPUs on this specific task is that the design choices of a CPU are for flexibility and general computing workloads. The GPU, on the other hand, is designed to do simple arithmetic instructions over and over again (easy to parallelize). These design choices make GPUs a lot more efficient for machine learning, and especially for deep neural networks [23]. Alternatively, if investing in a GPU is not the preferred choice, there are many cloud services available to us today where we can pay a small sum in order to use a system that is a better fit for the task.

Now how do we make a machine learn? Well, there are many different approaches to this, which we will discuss in the next sections, but let us say we want to use a neural network for achieving some goal. Then our next step should be to choose a framework. We can, of course, do everything from scratch, but why reinvent the wheel when there are so many good frameworks and tools already out there?

Python is a programming language perfect for machine learning in our opinion. The language looks a lot like pseudo-code, and this is perfect because we do not want to spend time on syntax rules in another language. A popular framework called **TensorFlow** is available to use in Python, and developers from Google have built it.

TensorFlow allows the programmer to build models quickly, and also execute the training and testing. Before we get started with TensorFlow, a smart choice would be to do a quick search on the Internet to see if someone else has already done something similar to what we want to achieve, and if you find something, the odds are that your neural network model can be similar, if not identical to it. If not, then we have to sit down and write the model by ourselves.

For the model implementation part, whether we found a model online or want to write it, we can, of course, use TensorFlow directly, but using **Keras** as an abstraction layer above TensorFlow is a popular choice. On

their documentation website [12], 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 [12], 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 2.1.

Source Code 2.1: 30 Seconds to Keras

```
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)
```

One thing that Keras does not make any easier is structuring the dataset so that the model can fit it. There is a high chance that we have to write some code ourselves to do this. Numpy is a package for Python built for math operations where everything happens optimally (Python by itself adds overhead to everything). All machine learning assumes that the input data is Numpy arrays, so experience with Numpy can be advantageous when structuring the dataset.

2.3 Machine learning strategies

Picking the right machine learning model can be quite tricky, especially if inexperienced. There are a couple of different *strategies* one can choose from when deciding on a machine learning model. These are called *supervised and unsupervised learning*, and we need to think about how the dataset is structured, and what we want to achieve 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

Supervised learning is the machine learning strategy where we provide both input and correct output data to the algorithm [22]. We may use supervised learning if the goal is to train a model to classify letters in the alphabet, or something else where we 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 we train this alphabet model, we will be able to input an image (not seen before by the algorithm) 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 [16].

Another kind of supervised learning is called *Regression* and is all about predicting (or estimating) a value. A classic example of regression learning is predicting income, using *features* like home, location, job title, the field of study, years of education and years of experience. We call these features *categorical* (first three) and *numerical* (last two) [15].

2.3.2 Unsupervised learning

Another strategy is unsupervised learning. We want to use this if we 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 we have unlabeled data and want to for example group data together in what we call a *cluster*. Unsupervised learning may not be as common as supervised learning, but unsupervised learning can be quite beneficial in some cases; for example when grouping addresses together in neighborhoods if we have an unsorted list of addresses as the dataset.

2.3.3 Semi-supervised learning

Now, we may not always want to use one of the strategies above. Looking at the dataset, we may want something in between like a combination of labeled and unlabeled data. Semi-supervised learning comes in handy when this is the case. For example, if we have many data samples to label in our dataset, it can be simply too much work. We will not go deep into details about how this works, but it is important to mention it because of its usefulness.

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

2.4 Machine learning approaches

When we know whether we want to use supervised learning, unsupervised learning or something in between, we need to select an approach. We call them approaches because we use these regardless of the strategy that we end up using; most of them (except for reinforcement learning) work in both supervised and unsupervised learning. There are a lot of different approaches available, and we will describe some of them.

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. We will not go into details about how this works, but our 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*. In this approach, we set up all the different outcomes (with the training data set) of a specific question in a tree. Let us say we 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 correctly picking these is essential 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 the 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, we can create decision rules by merely following the tree with new data.

Random Forest

One known problem with decision tree models is that they often include much *variance*. Variance in an algorithm means that it 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 useful for both classification and regression learning [21]. It mainly works by combining decision trees, where the tree building algorithm is heavily randomized for all trees.

For example, if we want to get movie recommendations using machine learning, using one decision tree will most likely be insufficient. Just think what happens when we ask a friend for movies to watch. What that friend recommends is purely based on movies we like ourselves and what the friend likes. We might be lucky and find our next favorite movie, but most likely, asking multiple people for recommendations is going to yield better results. The same goes for machine learning, and decision trees will most likely give a better answer if we combine them 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, inspired by the way biological neural networks in the human brain process information [18]. There are a lot of different neural networks, but all of them share the same underlying layer-based architecture, where data get passed between layers where computation happens. The first layer is the input layer, which passes the data to the next layer, which is the hidden layers. The number of hidden layers is entirely up to the model and the programmer, and this is where the intermediate processing/computation is done before the data get passed to the output layer where we perform an activation function to define the output [19].

If we have a lot of hidden layers in a neural network, we call it a *deep* neural network. Deep networks can be useful for anything, and only the programmer's creativity sets the limit. Two common ways to use neural

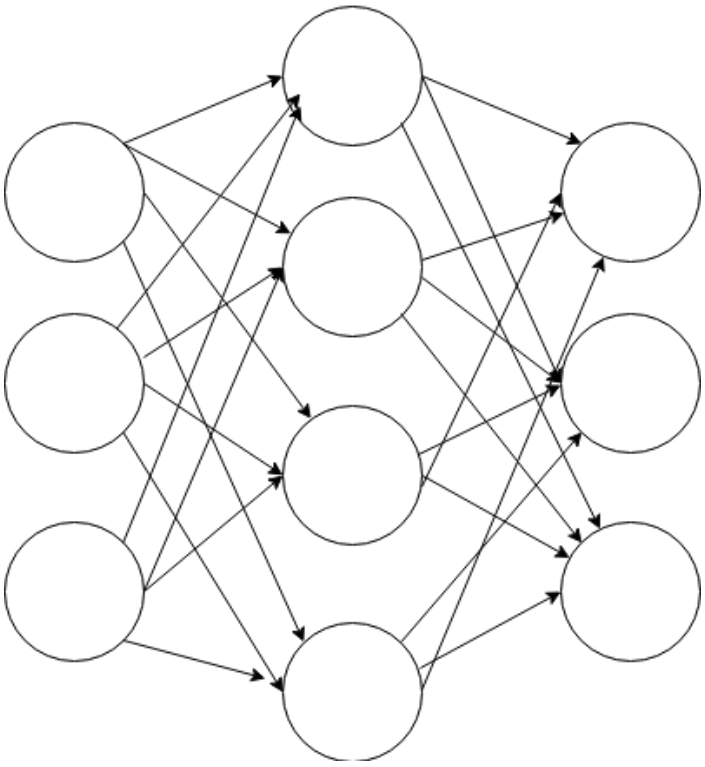

Figure 2.1: Neural network

networks are *Recurrent Neural Networks* and *Convolutional Neural Networks*. These two have their use cases, which we will describe below.

Figure 2.1 is a visual representation of a shallow neural network with the input layer on the left, one hidden layer in the middle and the output layer on the right-hand side. For each layer, we have fully connected nodes, which means that each node has a connection to all nodes in the next layer.

Recurrent Neural Networks (RNN)

This type of neural network is useful 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 the phone. Also making predictions based on historical data, like a forecast, is something an RNN can do effectively.

One downside to Recurrent Neural Networks used on large sequences of data is that the prediction will most likely be off if a word written at the beginning of a long text is a dependency for a prediction four chapters later, for example, the home location of the main character. The workaround for this is something called *Long Short-Term Memory Recurrent Neural Network (LSTM RNN)*, and is the idea of having additional logic to

avoid the prediction model forgetting essential facts.

Convolutional Neural Networks (CNN)

A Convolutional Neural Network, or *CNN* for short, can be used for identifying patterns in data, which then is the underlying calculations for either prediction or classification. A common use case for CNN's is image recognition. In image recognition, we train our models to be good at identifying objects in images, for example, the difference between cats and dogs. Then we 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 a two-dimensional array of pixels, so the network also needs to have two dimensions in the convolutional layers. 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 [10].

2.5 How can machine learning help people with bipolar disorder?

The use of machine learning in the medical fields is growing exponentially. There is no limit to the usefulness of machine learning, and it can certainly help in the medical/psychological field too. Let us say bipolar patients had a device that measured their heart rate among other things 24 hours a day could feed the data into a neural network that gave the user's bipolar state as output. That would be useful for both the patients and doctors/nurses. Another use case could be if medical institutions knew in advance how many new bipolar patients to expect the next day.

Using machine learning in this field of study could help many people get through their depression or mania, and potentially get rid of the condition altogether.

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, the dataset is sufficient for this thesis. It contains activity level data for 23 bipolar patients, one unipolar patient, and 32 non-depressed contributors. From now on, we will refer to the bipolar/unipolar group as the *condition group*, and the non-depressed group as the *control group*. The dataset details [6] also follows this convention.

	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)

The dataset is in two parts. One part includes the demographics of each participant 2.2, where the fields are [6]:

- **number**: a unique id for each participant
- **days**: number of days of activity data collection
- **gender**: 1 for female and 2 for male
- **age**: participant's age (grouped by four years)
- **afftype**: affliction type, where 1 is for bipolar type II, 2 equals unipolar depressive, and 3 for participants with bipolar type I
- **melanch**: 1 means a participant has melancholia, 2 means no melancholia
- **inpatient**: whether the patient is inpatient (1) or outpatient (2)
- **edu**: participant's education in years (grouped by four years)
- **marriage**: married or cohabiting (1) or single (2)
- **work**: whether the participant is working or studying (1) or not (2)
- **madrs1**: MADRS score before activity measurement started
- **madrs2**: MADRS score after activity measurements ended

The second part includes sensor data about the condition group and control group, as one file for each participant 2.3. These files are in two folders for the two groups (control/condition) respectively, and one file for each person 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 unauthorized for the data gets access to it? What if the database gets hacked? With new regulations (GDPR), which means that users have the right to be forgotten or deleted. 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 was 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 models were going to be One-Dimensional Convolutional Neural Networks. We solved all these using each participant's activity data as input, and we created a model that, with few changes, could be used for all three goals. For the different goals, only the last few layers needed to change.

For the first goal, classifying **control** or **condition** group, the output data was a matrix with two columns (**control**, **condition**), with a 1 in one of the columns and a 0 in the other for each row.

However, first, we wanted to see if these problems were solvable with simple regression. The idea was to 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 of experience with machine learning will know that this is not nearly enough data. However, we wanted to do it regardless, and see how a simple and stupid model performed. Doing this, we established some 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 convolutional neural networks, so we proceeded to implement an image recognition algorithm. We found a

tutorial on how to make a 2D CNN for classifying cats and dogs from images [14] and thought it would be an excellent way to learn.

It was both a fun and informative experience in implementing this. Especially when we extended the script to allow an image URL to predict on, we could browse for images of cats and dogs on the Internet, and find out if the model could classify them (in most cases it did). We even tried inputting images of humans to the model for fun, to see if they looked more like a dog or a cat.

However as mentioned before, the activity measurements are of one dimension, so we could not use this network.

> *A 1D CNN is very effective when you expect to derive interesting features from shorter (fixed-length) segments of the overall dataset 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 data from gyroscopes or accelerometers).* [10]

To learn more about one-dimensional convolutional neural networks, we followed a tutorial [10], which used a dataset containing accelerometer data from a smartphone on the participant's waists. The goal here was 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 one-dimensional convolutional neural networks work and how we should structure our data to make it work for our dataset.

We also got some ideas about where our dataset could provide more data. What if the dataset containing the current mental state of the bipolar patient? Then someone could make some automated system that always can tell a patient whether they are not depressed, manic or depressive. However data collection for this kind of task would be difficult because we cannot always know what the patient thinks, nor does the patient. 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 the same for each goal, since we wanted to use a similar model on all three. The tutorial [10] sliced up the measurements with overlap, and labeled the slices. We 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 for 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. We had to transform the values to a categorical matrix in order to be able to select a *category* for classification. This list, which we called **labels**, looked like this:

labels[i] = [0, 1]

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

Depression Classes

Here we wanted to classify which depression class a participant belonged to, and as described in the earlier, we divide MADRS scores into some cutoff-points, which we used as *classes* in our classification:

- 0-6: normal
- 7-19: mild depression
- 20-34: moderate 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

- 34-60: severe depression

So instead of labelling the segments as **[CONTROL]** or **[CONDITION]**, we labeled them as 0, 1 or 2 (we ignored 3 as there are no participants with MADRS scores this high). One element in **labels** after making it a categorical matrix looked like this:

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

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

Predicting MADRS Score

Instead of classifying one of three or one of two classes which we have done earlier, this time we wanted to predict the actual MADRS score value. Creating the output data for this goal is easier; we appended the MADRS score for the corresponding participant to the list. For example:

scores[i] = [18]

3.3 Performance Metrics

When finishing with a training session for a machine learning model, we need to use different metrics to be able to tell how the model performed based on some testing data. For prediction models, we only consider the loss, but evaluating classification models happens differently.

To begin with, we 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 we have made our computer able to classify something with 98% accuracy is something that they would understand. Other metrics include *confusion matrix*, *precision*, *Recall*, *specificity* and *F1 score*, which will be described below.

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 how well our model is performing by having correlation values for the different classes. Let us say we have 200 samples in our test data to use on our model that classifies control vs. condition group.

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

Table 3.2: Confusion Matrix Example: 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 can classify that a participant is in the condition group if he or she is, and having a high number in **True Negative** means that the model can 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.' ([20])

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 five 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% [20].

3.3.3 Precision

Precision operates entirely on the predicted positives, and it tells us how many **true positives** there is among **predicted positives** ($\frac{TP}{TP+FP}$) [20].

This performance metric 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** [20].

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 our classification model is good enough. However, the relationship between *recall* and *precision* and knowing when to use which can be confusing, at least if the different classes are somewhere between entirely 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 $2 \cdot \frac{P \cdot R}{P+R}$ [20]. 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. Just import and use them

	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

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].

Chapter 4

Implementing the Project

4.1 Regression

Regression is used a lot in statistics, where its primary purpose is to predict a target value based on independent predictors and is useful to find cause and effect relationships between variables [11]. One experiment that might be interesting is to perform 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 cannot 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 reduced to 1), and ran the regression on all of the remaining columns one by one:

- **Gender**
- **Age**
- **Days**
- **MADRS1**
- **MADRS2**

```
def regression_model():
  model = Sequential()
  model.add(Dense(5, input_dim=1, activation='relu'))
  model.add(Dense(1))

  model.compile(loss='mse', optimizer='adam', metrics=['mse'])
  return model
```

Source Code 4.1: Regression Model

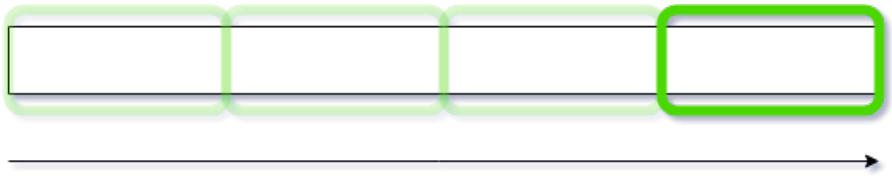

Figure 4.1: Feature Detector / Filter "sliding" over input data

The model for this task is straightforward. 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 **relu** and outputs 5 neurons. In the output layer, we activate using a **linear** function (default activation function when none are specified) and return one value: the prediction. We compiled the model with the loss function **Mean Squared Error** and the optimizer **Adam**.

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 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. The task should be easy 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 did not 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 did not 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 Convolutional Neural Networks [10], 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.

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 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* we 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 with 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 [10]. 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 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 applied this after the *GlobalAveragePooling1D* layer:

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).

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

Table 4.1: Categorical Labels

- 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:
  - 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.

4.2.4 Train and test data

We did one last step before we started training the model. We needed to split the data ion 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 data that is unseen to the model.

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)
```

train_test_split

The function *train_test_split* from the *sklearn* package was useful here; 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.

Cross-validation

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. Cross-validation is a good way of checking if your model's performance truly is independent on which data samples it trained on. The higher number of splits (K) means fewer data samples to test on, so you need to keep that in mind (the same for *train_test_split* if you set the *test_size* too low). Sklearn has an implementation of *K-fold*, 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

```
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 models

Out of the box, we didn't think that the models were going to perform perfectly. Therefore we needed to experiment with each parameter that we passed to the models 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 discussed 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 essential 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 afterward. We can't use all the data in both cases, so the most balanced split needs to be determined.

4.4 Source Code

4.4.1 Create Input Data

Source Code 4.4: Read Dataset, Create Input Data

```
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

```
def create_classification_model(segment_length, output_layers):
  model = Sequential()

  model.add(Reshape((segment_length, 1), input_shape=(segment_length,)))
  model.add(Conv1D(100, 10, activation='relu', input_shape=(segment_length, 1)))
  model.add(Conv1D(100, 10, activation='relu'))
  model.add(MaxPooling1D(2))
  model.add(Conv1D(160, 10, activation='relu'))
  model.add(Conv1D(160, 10, activation='relu'))
  model.add(GlobalAveragePooling1D())
  model.add(Dropout(0.5))
  model.add(Dense(output_layers, activation='softmax'))

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

  return model
```

Source Code 4.6: 1D CNN Model for Prediction

```
def create_prediction_model(segment_length):
  model = Sequential()

  model.add(Reshape((segment_length, 1), input_shape=(input_shape,)))
  model.add(Conv1D(128, 2, activation='relu', input_shape=(segment_length, 1)))
  model.add(MaxPooling1D(pool_size=2, strides=1))
  model.add(Conv1D(64, 2, activation='relu'))
  model.add(GlobalAveragePooling1D())
  model.add(Flatten())
  model.add(Dense(10, activation='relu'))
  model.add(Dense(1))

  model.compile(loss='mean_squared_error',
                optimizer='adam',
                metrics=['mse'])

  return model
```

Chapter 5

Training the models

For training large neural networks, we often need expensive hardware if we want to do it on our own. The way to do it these days is to use cloud services such as Google or Amazon to run the machine learning once we have prepared the dataset and the model. We often pay for these services paid by the hour, but they are not expensive as long as we remember to shut them down after use, and only use them when we 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 personal computers, with an Nvidia GTX 1070. We guess 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, a thousand epochs did not take that much time, and the training loss graph 5.1 shows that more epochs will not 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 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 to train on as many rows as possible. Note that the data elements that were in the test and train splits were different for each column.

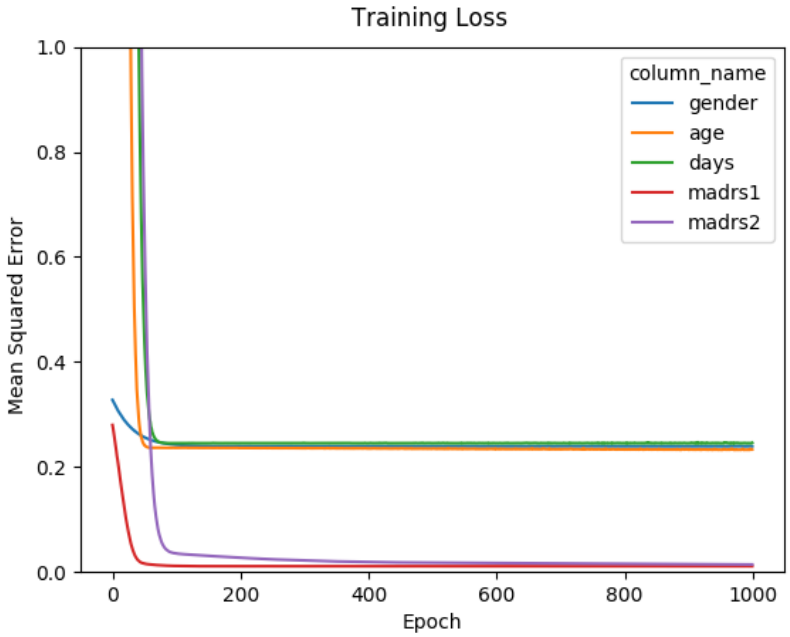

Figure 5.1: Regression Training Loss (MSE) by Epoch

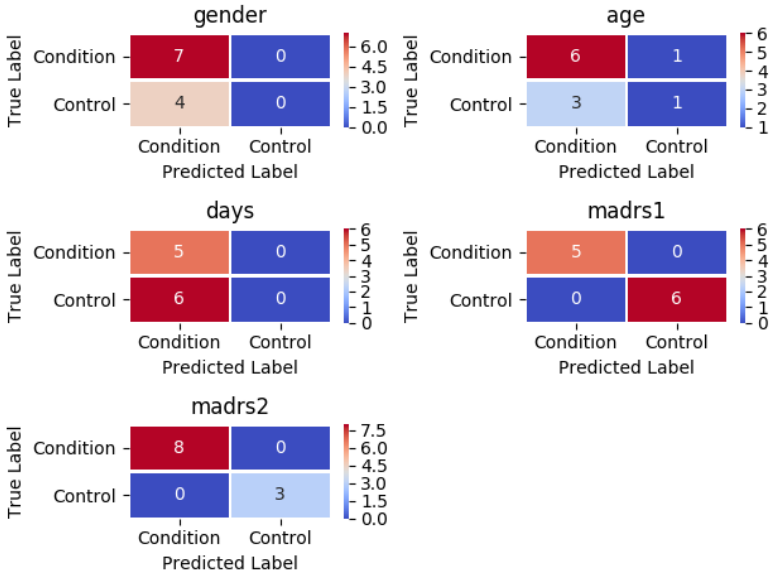

Figure 5.2: Confusion Matrices for Regression

5.1.1 Results

The confusion matrices 5.2 display 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 we can see that this is achieved because the model guessed everything right. The other columns 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, the performance scores tell us that the model is good at predicting that someone is in the condition group (precision). Other than that, we cannot learn that much from them. Also, we have undefined specificity for both

gender and days because we do not 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 looked 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 groups

5.2.1 Training and finding the optimal segment length

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. We used 40% the training data as validation data, which made it easy for us to tell if the model was learning from epoch to epoch or not.

We did the training for each of the eight different input sets for ten epochs. To keep it simple, we used a batch size of 16 and the Adam optimizer with a default learning rate of 0,001 throughout this experiment. The primary objective here was to find the best segment length to use, and not to train the models to be perfect, so these hyper-parameters 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 ten epochs).

Looking at the history graphs 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 48, and then it did not seem to be any better for 96 hours. This was the case for both training and testing, as the evaluation graphs (results after testing the model with the test-split) to the right also show straight lines from 48 hours to 96 hours. The question was whether 48 hours was our magic number, or if we just needed to train it more.

To find the best segment length, we needed to experiment with more epochs. We reran the same experiment for 50 epochs, with 48, 72 and 96 hour long segments. However, from the results for that 5.4, it is 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 close to perfect. The additional

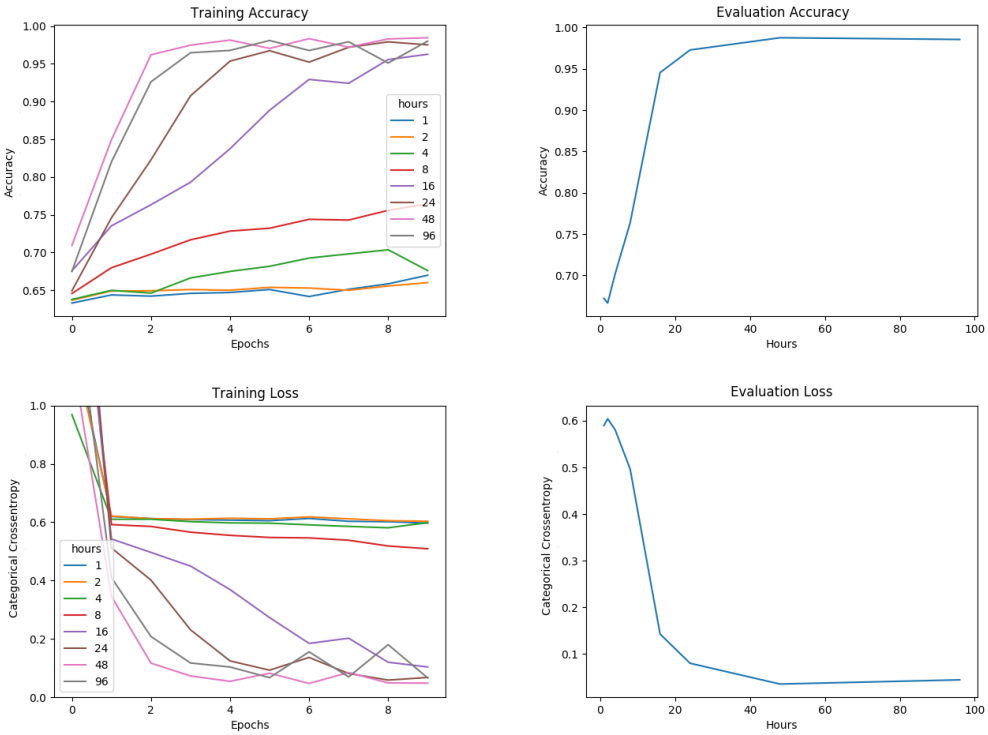

Figure 5.3: Training (left) and evaluation (right) results (10 epochs)

40 epochs of training reduced the false positive classifications by a little bit, which was worth the time in our opinion as we want the value of wrong classifications to be as close to 0 as possible. For this relatively small dataset, we are happy with a classification model with scores above 99% on all performance metrics previously discussed.

5.2.2 Performance metrics

Accuracy

$$A = \frac{TP+TN}{TP+TN+FN+FP} = \frac{3122+1577}{3122+1577+5+13} \approx 99,6\%$$

Precision

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

Recall

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

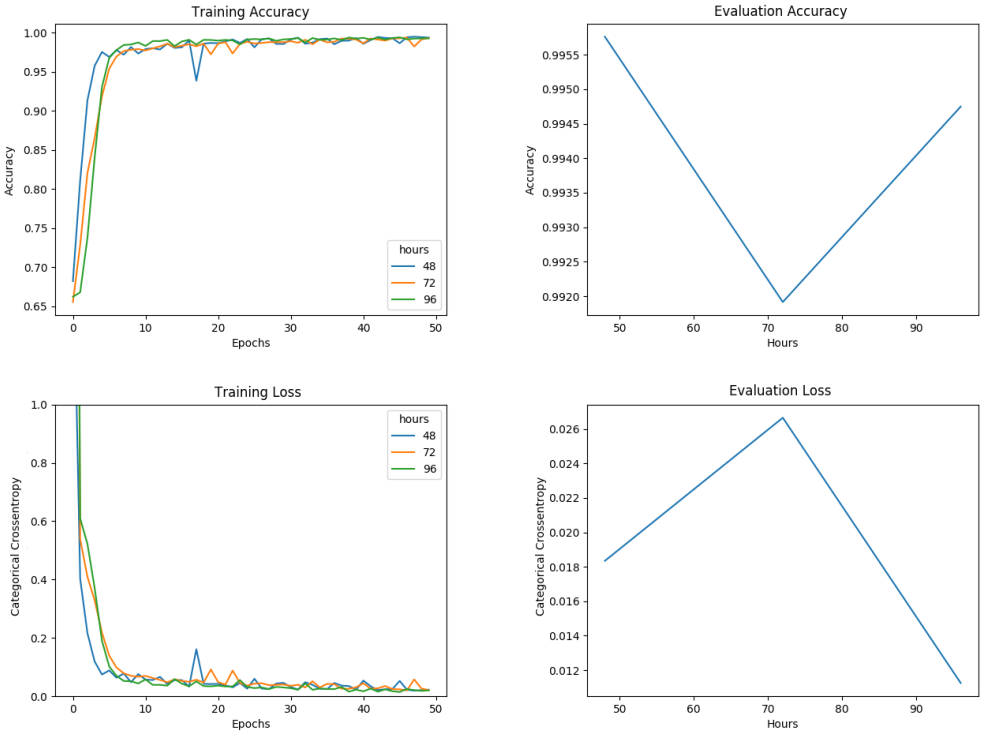

Figure 5.4: Training (left) and evaluation (right) results (50 epochs)

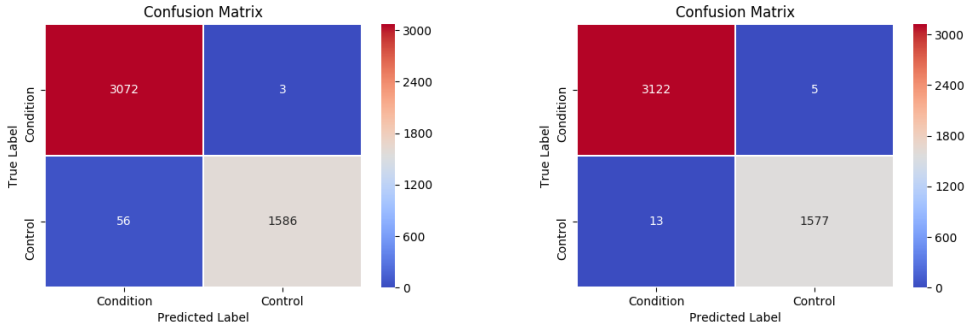

Figure 5.5: 10 epochs (left) and 50 epochs (right) for 48 hour segments

Fold	Loss	Accuracy
1	0.059426798584136714	0.9807080771929636
2	0.07039665617483107	0.9813440746237015
3	0.05903829742228553	0.9809200763323308
Mean	**0.0629539173937511**	**0.9809907427163319**

Figure 5.6: 3-Fold Cross validation

Specificity

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

F1 Score

$F1 = 2 \cdot \frac{P \cdot R}{P+R} = 2 \cdot \frac{0,996 \cdot 0,998}{0,996+0,998} \approx 99,7\%$

5.2.3 Cross-validation

To ensure that our model is not overfitting and did not get lucky when classifying the samples from the testing set, we proceeded to use 3-fold cross-validation. First, we split the dataset in two like before, into a train and test set (80/20 split here as well). Then we generated three folds containing training and validation parts, where for each fold a model was trained to fit the inputs. Each epoch the model was validated against the validation split. After training a model for a fold, we evaluated them by looking at the mean accuracy/loss against the global test split. If the accuracy was still high and the loss was still low, the model would have a good chance of doing correct classifications on unseen data.

To make this process quick, we trained the model for each fold only for ten epochs. The goal was to prove consistency in the model and not achieve high performance, so it seemed enough. As one can see in the cross-validation results 5.6, we have a mean loss of 0.06 and a mean accuracy of 0.98, which means that the model is consistently correct in most classifications.

5.3 1D CNN: Depression Classes

5.3.1 Training and finding the optimal segment length

The next neural network we trained was for classifying how depressed the participants were. Depression classes are, as we said before, based on their MADRS score which is 0 for participants in the control group and differs between 11 and 28 in the condition group 2.2. We labeled participants with

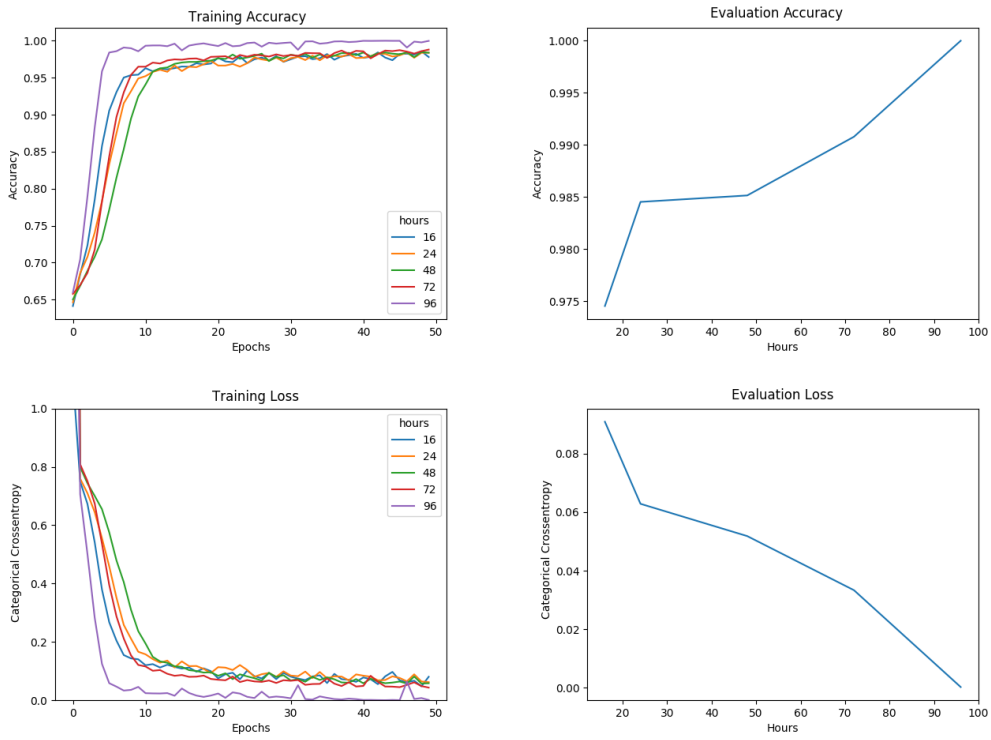

Figure 5.7: Depression classes: Training (left) and evaluation (right) results (50 epochs)

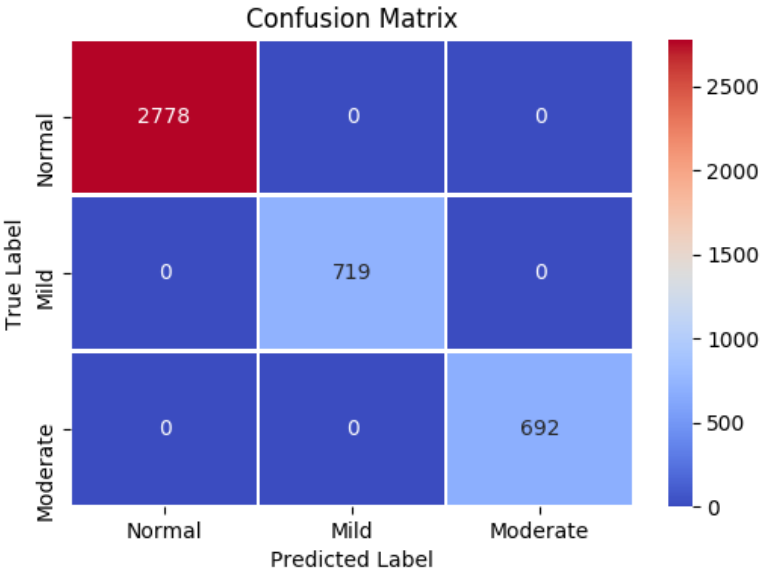

Figure 5.8: 50 epochs for 96 hour segments

MADRS score 0 as not depressed, between 11 and 19 as mildly depressed, and above 20 as moderately depressed.

Using what we previously discovered from classifying control vs. condition groups, we knew how we wanted to achieve this goal. First, find the optimal segment length, then use the best segment length in cross-validation, and guarantee that the performance is consistent.

In the previous experiment, there was a clear threshold after segment lengths of 24 hours where the performance did not improve that much if increased more 5.3. The results were just a little bit better for 48 than 24 hours and worse for longer segments. The question was whether the threshold existed for depression classes also, or if we could use longer segments to achieve better results.

We experimented the same way as before, except that we went straight for 50 epochs and skipped the shortest segments of 1, 2, 4 and 8 hours, as we were positive these segments would not be any good. We proceeded to train segment lengths of 16, 24, 48, 72 and 96 hours. Looking at the training and testing graphs 5.7, we can see that the results for 96-hour segments were outstanding. We achieved an accuracy of 100% on the testing set. The confusion matrix 5.8 shows not a single error in classification.

5.3.2 Cross-validation

To make sure this was not just lucky, we needed to cross-validate here as well. We split the dataset into a training and testing set (80%/20%), then did three-fold cross-validation on the training set, just as before. If all three folds had similar accuracy and loss, and the mean values were good (close to 1.0 for accuracy and close to zero for loss), we had achieved a consistent model, and the model would be fit perfectly, at least for the 55 participants in the dataset.

We trained the three models for 15 epochs each. We knew that was not enough to give us 100% accuracy (need around 50 epochs for that), but we aimed for somewhere around 98-99% for all folds. Looking at the cross-validation results 5.9, we can see that the lowest accuracy was 98,5% and the highest was 99,8%. The mean accuracy for all three folds was 99%, which is what we wanted to see.

The model is indeed perfect for classifying depression classes, at least for all the participants in the dataset. We know that the model will not classify with 100% accuracy on data outside the participants in the dataset since 55 persons cannot possibly tell everything about human activity behavior, but we will go more in-depth on this later.

Fold	Loss	Accuracy
1	0.04258962207290729	0.9890188589162091
2	0.008786056019605889	0.998328956791597
3	0.046309486162426576	0.9847218906660301
Mean	**0.032561721418313254**	**0.990689902124612**

Figure 5.9: 3-Fold Cross validation

5.4 1D CNN: MADRS Score Prediction

Now that we have models for classifying both whether a participant belongs to the control group or the condition group and how depressed participants are, we can get started on our last goal which is predicting the MADRS score of our participants.

Training a prediction model is significantly more computationally heavy than training a classification model. We noticed this as soon as we started the training when the loss started on around 100 which is very high, and prediction on test data after a few epochs was poor. In classification, we were used to training and validation loss between 1 and 0 and reaching a high accuracy after only a few epochs. We had to train for some epochs to compare segment lengths, which would only show which are best in the first epochs.

5.4.1 Segment length

The plan was to perform 100 epochs of training for different segment lengths. We used the same segment lengths as we did for the previous model: 16, 24, 48, 72 and 96 hours. Since the two classification experiments had a different optimal segment length, it was an interesting question whether this was the case for regression as well.

After finding the best segment length, we used the one with the most promising results to train the model for a more extended period. We wanted to see how good performance we could achieve if we increased the number of epochs by a lot and let it learn overnight.

5.4.2 Cross-validation

Before training the model to be as perfect as possible, we did cross-validation to check its consistency. Just as we had done before, we performed 3-fold cross-validation. For each fold, we trained the model for 100 epochs to fit the corresponding training data and validated on the corresponding validation data. Then, after a model had completed its training, it was evaluated against the global test data (same procedure as

the two previous experiments). Finally, the Mean Squared Error for each fold was saved and compared with other folds to see how they averaged. Table 5.10 contains the cross-validation results, and it seems that the model is consistent.

Fold	**Mean Squared Error**
1	30.177846447198807
2	33.61265733228388
3	30.40483367657899
Mean	**31.398445818687225**

Figure 5.10: 3-Fold Cross validation

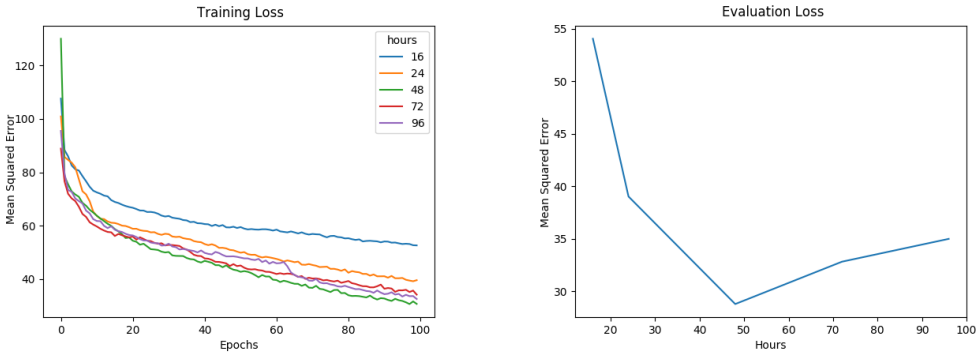

Figure 5.11: MADRS Prediction: Training (left) and evaluation (right) results (100 epochs)

5.4.3 Hyper-parameters

From the training and evaluation graphs 5.11, it seemed like the way to go was 48-hour segments. We don't know what's best for training the model further, but this was what we settled on using.

Also, unlike the classification experiments, we experimented more with hyper-parameters. The Adam optimizer with default learning rate did the job for classification, but now that we had to train for thousands of epochs, these parameters had the potential to affect the results by a lot. We proceeded to compare three different optimizers: Adam, SGD, and Nadam. We had to change the learning rate to get SGD to work (the default which is 0.01 [13] resulted in NAN MSE for some reason). All optimizers were set to use a learning rate of 0.0001. The optimizer comparison graph 5.12 shows that Adam was the best choice for prediction as well.

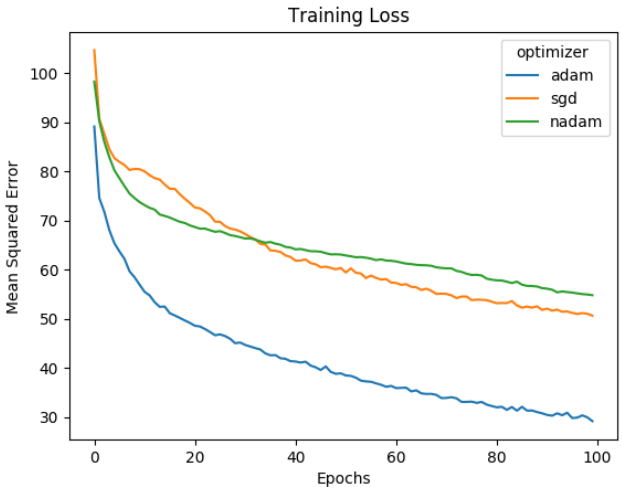

Figure 5.12: Optimizer comparison (100 epochs)

5.4.4 Training the model

To summarize, this model was trained to fit segments of length 48 hours (2880 minutes). It was trained with the optimizer Adam using a learning rate of 0.0001. We did not change the batch size; we kept this at 16. We split the dataset into 60% training data and 40% testing data. Based on the time it took to train each of the 100-epoch experiments, we calculated that we had time for 5000 epochs training it from the evening until the next morning.

When we checked for results the next day, we could see the training had resulted in a mean squared error of 3.24, and predictions on the test data 5.13 looked very promising. This graph shows correct MADRS scores in the x-axis and the predicted MADRS scores in the y-axis. Each blue dot is a prediction, and the dotted black line is a linear guideline for the perfect predictions (where the predicted and correct scores are the same).

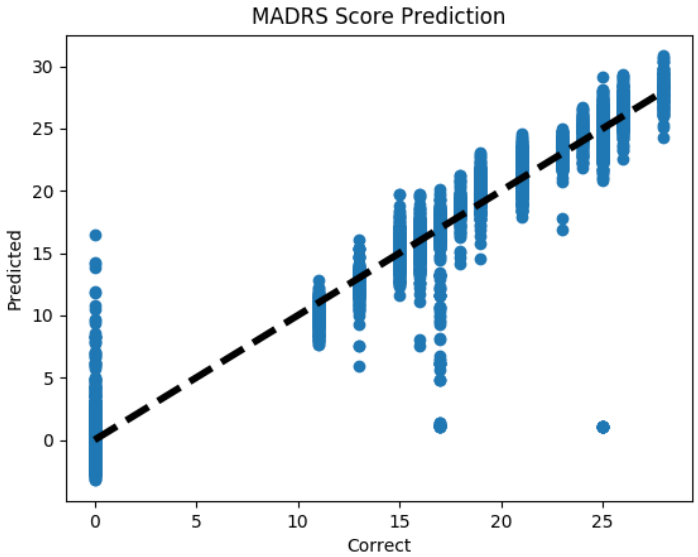

Figure 5.13: Predicting MADRS Scores from test-set

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] N. Herrmann et al. 'The Sunnybrook Stroke Study'. In: *Stroke* 29.3 (1998), pp. 618–624. DOI: 10.1161/01.STR.29.3.618. eprint: https://www.ahajournals.org/doi/pdf/10.1161/01.STR.29.3.618. URL: https://www.ahajournals.org/doi/abs/10.1161/01.STR.29.3.618.

[10] *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.

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

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

[13] *Keras documentation*. Accessed: 28.03.2018. URL: https://keras.io/optimizers.

[14] *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/.

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

[16] *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.

[17] S.A. Montgomery and Marie Åsberg. 'A New Depression Scale Designed to be Sensitive to Change'. In: *Br J Psychiatry* 134 (May 1979), pp. 382–389.

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

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

[20] *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.

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

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

[23] *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.