UNIVERSITY OF TECHNOLOGY SYDNEY

FACULTY OF ENGINEERING AND INFORMATION TECHNOLOGY

APPLYING MODERN TECHNIQUES IN ARTIFICIAL INTELLIGENCE TO NEURAL ACTIVITY

WILLIAM SMITH

SUPERVISED BY YU-KAI WANG

ASSISTED BY SAI KALYAN RANGA SINGANAMALLA

A 12 CREDIT POINT PROJECT SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR OF ENGINEERING

STUDENT NUMBER: 12617066

PROJECT NUMBER: SUM-20-04231

MAJOR: SOFTWARE ENGINEERING

DECLARATION OF ORIGINALITY

I certify that to the best of my knowledge, the content of this project is my own work. This project has not been submitted for any degree or other purposes.

I certify that the intellectual content of this project is the product of my own work and that all the assistance received in preparing this project and sources have been acknowledged.

William Smith

30th January 2021

ABSTRACT

Humans and machines have never been so tightly connected as they are in the 21st century. The computer, once a handy tool for mathematicians, is for billions of people suddenly a permenant and very vital companion. This project is a study in how computers and humans can work closer than ever; how artificial intelligence can actually decipher human brain waves.

Neural networks, computing models based on the very design of the human brain, can help to understand real human brain signals using brain-computer interfaces. No screen; no keyboard; the brain and computer can communicate without impediment. In 2018 a group of researchers created the EEGNet, a neural network built explicitly for classifying all kinds of neural signals to a high degree of accuracy. This project seeks to understand their work, how brain-computer interfaces and artificial intelligence can work together and to recreate their success on a new dataset.

The potential for usages of brain-computer pairings in the modern world range far and wide, from the medical world to forms of leisure. It offers a non-invasive means by which handicapped people control their wheelchair by simply thinking about where they want to go, or pilots control their drones with only their brain as remote control. The use cases of such technology are limited only by the human imagination. This project breaks down some of the technology required, details the process of going from nothing to having a fully trained neural network model and successfully classifies new EEG data with an accuracy of 88%.

TABLE OF CONTENTS

1. INTRODUCTION ..... - 6 -

1.1 ELECTROENCEPHALOGRAPHY ..... - 6 -

1.2 ARTIFICIAL INTELLIGENCE IN THE 2010S ..... - 7 -

1.3 THE MODERN BRAIN-COMPUTER INTERFACE ..... - 8 -

1.4 EEGNET ..... - 9 -

2. LITERATURE REVIEW ..... - 10 -

2.1 THE XDAWN ALGORITHM ..... - 10 -

2.2 DATA NORMALISATION ..... - 11 -

2.3 BCI CHALLENGE NER2015 ..... - 11 -

2.4 SEPARABLE CONVOLUTIONS ..... - 13 -

2.5 EEGNET ..... - 15 -

2.6 MEASURING SUCCESS ..... - 16 -

3. METHODOLOGY ..... - 19 -

4. SETUP ..... - 20 -

4.1 BCI EEGNET ..... - 20 -

4.2 ARL EEGMODELS ..... - 22 -

5. DATA EXPLORATION ..... - 24 -

6. MODEL FITTING ..... - 28 -

7. MODEL IMPROVEMENTS ..... - 31 -

7.1 HUMAN ERROR ..... - 31 -

7.2 CLASS WEIGHTS ..... - 32 -

7.3 KERNEL LENGTH ..... - 33 -

7.4 BATCH SIZE ..... - 34 -

7.5 SUBJECT-WISE CLASSIFICATION ..... - 36 -

7.6 BUTTERWORTH FILTER ..... - 38 -

7.7 DOWNSAMPLING AND CHANNEL FILTERING ..... - 39 -

8. EXPERIMENTATION FRAMEWORK ..... - 41 -

8.1 BUILDING THE FRAMEWORK ..... - 41 -

8.2 INTERPRETING RESULTS ..... - 44 -

9. EXPERIMENTS ..... - 45 -

9.1 BATCH SIZE ..... - 45 -

9.2 EPOCHS ..... - 45 -

9.3 Sample rate ..... - 46 -
9.4 Shorter sample time ..... - 46 -
9.5 Stratify ..... - 47 -
9.6 Normalisation ..... - 47 -
9.7 Number of filters ..... - 48 -
10. RESULTS ..... - 49 -
11. CONCLUSION ..... - 50 -
12. REFERENCES ..... - 51 -
12. APPENDICES ..... - 53 -
Appendix A: Project Communication Log ..... - 53 -
Appendix B: ARL Model Library Dependencies ..... - 55 -
Appendix C: Cross-subject Results ..... - 57 -
Appendix D: Subject-wise Results ..... - 64 -

1. INTRODUCTION

The end-goal of this project is to apply state-of-the-art technologies and techniques in artificial intelligence (AI) to successfully analyse and classify the electrical signals generated by the brain as measured by an electroencephalogram (EEG). Using data measured from several subjects a Rapid Serial Visual Presentation (RSVP) experiment, a machine learning model was generated and trained to predict the actions and intents of the subjects using only their brain signals. The model is based off an existing model developed in 2018 named 'EEGNet', a type of convolutional neural network whose parameters and setup have been tuned specifically to successfully learn brain patterns as measured by an EEG (Lawhern et al. 2018). Varying based on the dataset used and the type of EEG experiment being carried out, Lawhern et al. found success rates of 80-90% in classifying EEG data.

In order to achieve this end-goal, the following goals were established as intermediary steps. First was gaining an understanding of EEGNet and the underlying technologies and frameworks, such as convolution neural networks (CNNs). Secondly, my goal was to understand other studies in the industry relating to the EEGNet and how other techniques perform on the various BCI paradigms. Thirdly I had to understand the common experimental procedures used in EEG across these paradigms, and fourth was understanding the specific format of the data I would be working with, which has already been gathered by my supervisor as part of another paper (Lin et al. 2015). Part of this fourth step was also to explore various state-of-the-art techniques which might find success performing classification on that specific dataset. Finally using the EEGNet on the data to achieve good classification performance would be, as mentioned, the end-goal.

1.1 Electroencephalography

Human brain cells communicate through different electrical impulses, which scientists have learnt to record using an EEG. Electroencephalography (EEG) is the study of human brain signals through the usage of small, metal electrodes attached to the human scalp, capable of detecting the electrical signals generated by the brain. Since the start of the 20th century scientists have found EEG to provide a convenient 'window on the mind' (Srinivasan 2012), a means of finally gaining quantifiable insights into the human consciousness. EEG to this day is a vital tool in the medical world, used for the diagnosis of epilepsy, sleep disorders and brain death (Chernecky & Berger 2013).

In an EEG experiment, as many as 256 metal electrodes can be attached to the subject's scalp. The amount and the placement of the electrodes varies depending on the experiment or procedure, and certain experiments might make do with as little as four electrodes. In most cases electrodes themselves do not make direct contact with the scalp, but a conductive electrode jelly paste bridges the gap, forming an electrical impedance between electrode and skin. The electrodes are connected to a computer and the neural oscillations, or brain waves, are recorded. Signals are often depicted visually on a time-domain graph plotting the relative changes in amplitude of the signals' voltages (see Fig. 1 below).

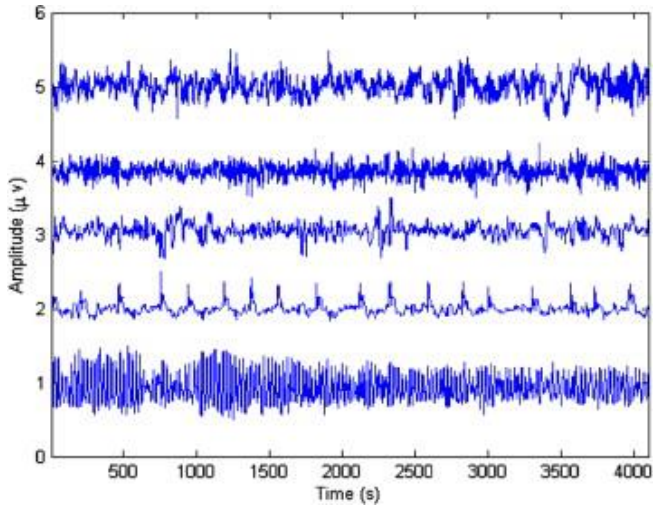


Fig. 1 - The signals from EEG an experiment run by the German University of Bonn

1.2 Artificial Intelligence in the 2010s

The idea of a developing an artificial intelligence based on the design of the human brain (a *neural network*) has been around since the early 1940s. The application of such a network on a real-world problem was performed successfully over fifty years ago, when Bernard Widrow and Marcian Hoff developed a model which reduced echoing in phone calls (Widrow & Hoff 1959). Through the early 2000s these models improved with increased computing power, access to graphics processing units (GPUs) and the usage of a subset of neural networks known as convolutional neural networks (CNNs). The CNN, due to its ability to generalise, found its way to success and fame as the AI model which could finally perform image recognition tasks (e.g., reading handwritten numbers and recognising human faces) at super-human levels (Ciresan et al. 2011). This would set the basis for what has been called for what has been called the third boom of AI, after the field endured two 'winters' in the late 1900s (Lloyd 1995). Today AI is used for everything from computer games to

facial recognition and medical breakthroughs including fighting the spread of the coronavirus (Broad 2020).

1.3 The Modern Brain-Computer Interface

A brain-computer interface (BCI) is a new means of building a bridge between the human brain and the world of computers. Most of us are aware of the typical human-computer interfaces which gained prevalence in the back half of the 20th century, those being the computer mouses and keyboards. These so-called interfaces are crucial to the modern world and provide, for most of us, an easy way of relating our thoughts and intents to a computer, but what about those of us without the physical capability to use such interfaces? A BCI can enable human to computer-interactions by monitoring a user's brain waves, potentially reducing the requirement for such physical devices as the mouse and keyboard. It provides a new channel of output for the brain connected directly to the computer, and as alluded to has clear potential use cases for helping disabled people. Particularly, BCIs are finding their places in usages for the rehabilitation of sensory and motor disabilities, exoskeletons and neurocommunication (Vallabhaneni 2005).

BCIs in more modern history have been looked at as a way not just in which to aid disabled people, but to enable a more seamless computer interaction for everybody. The field, with its modern developments, is regarded as one of the 'most exciting interdisciplinary areas of science and technology' (Pisarchik 2019). One of the most high-profile actors in the field is Elon Musk, who as CEO of his company Neuralink, strives to use the devices for 'transhumanism', the idea of enhancing the human body and mind to work or function past their natural biological limits. The ideas and goals of Musk and his company have faced much public criticism, but he's not the only one looking to make use of BCIs for human enhancement. For instance, in 2019 a group of researchers found participants were able to control a quadcopter using a BCI (Duan 2019). The interface correctly interpreted the users' intent with an accuracy of 86.5%. Another study as recently as 2020 found success enabling subjects to control home appliances with a BCI at a success rate of 92.8%. The application is pitched as a part of a potential 'smart home', enabling autonomy for elderly people within their homes (Park 2020). Clearly the field is wide open for new innovations.

1.4 EEGNet

In their seminal 2018 paper, Lawhern et al. proposed a CNN built specifically with the goal of learning and predicting those neural oscillations measured by an EEG. The model, dubbed the EEGNet, stitches together the scientific developments over the past several decades in artificial intelligence and BCIs and applies them to the EEG field. EEGNet enables other researchers in their own developments and applications of BCIs by providing a proven model which performs well across four different BCI paradigms and does so without requiring especially large amounts of training data. In each of the paradigms the model demonstrates classification performance at the level of or better than that of current state-of-the-art classification techniques. The paper offers also a way to graphically visualise the features learned by a trained neural network and an actual example of how anyone can write the code for their own EEGNet model. The EEGNet is the model which has been researched and applied in this project to the BCI paradigm of RSVP.

2. LITERATURE REVIEW

The following literature review is a brief summary of the reading undertaken as part of this project. Without understanding the theory it would have been impossible to analyse, optimise and improve the model at the heart of this project, and it is the readings and learnings which proved most crucial which are written up below.

2.1 The xDAWN Algorithm

The successful monitoring of neural signals using a BCI is complicated by the fact that brain is itself complicated and highly multi-dimensional. The brain works constantly and is never working on just one thing at a time. As such, attempting to listen to the signals generated by just one human intent or emotion is near impossible. EEG data comes from reading on the outside of the human scalp, meaning that signals of the over-ten billion currents generated between cells are simultaneously detected by each individual electrode. Attempts to interpret the raw signals are analogous to dangling a microphone over the roof of a sports stadium to hear a conversation between two football players. The observer will likely know when a team has scored by the cheers, but interpreting any more than that could be a fruitless task (Jensen et al. 2015). Fortunately this is where the xDAWN algorithm can help.

From their paper 2009 paper, Rivet et al. propose the model for a spatial filter which can work to increase the signal-to-noise ratio of raw EEG data. The model gets its name from the underlying model as depicted in Fig. 2 below. In this model, D is a matrix representing the time between when a subject is shown a stimulus and when the brain shows a measurable response for each stimulus (the stimulus onset), A' is the actual response after its dimensions have been reduced mathematically, W is the response's distribution over the various sensors/electrodes and N represents the input noise. In their paper the researchers compared how five models could classify EEG data, each using a different (or no) spatial filter on the data. Among the five tested methods, xDAWN achieved by far the best results. After only five 'repetitions' (the same stimulus shown to a target five times), the classifier with xDAWN filtering scored 80%, compared to the 2nd-best filter's score of just 71%.

$$X = D\,A'\,W^T + N'$$

Fig. 2 - The xDAWN model

2.2 Data Normalisation

In the machine learning world there are various types of data normalisation, which provide methods by which practitioners can smoothen or polish their data in a way that makes it easier for a machine learning model to 'learn' the data. The typical normalisation is Z Normalisation, also known as Standardisation, which is a transformation of the data in such a way that the new data's mean is zero and its standard deviation one. Another common normalisation method is the 'min-max', wherein all data points are fitted such that the minimum data point has a value of 0 and the maximum a value of 1. Finally, two common techniques used in machine learning are L1 and L2 normalisation. L2 normalisation, also known as the least squares algorithm, scales the input data such that the standard deviation is reduced and when the values of the squares of each item in a row are added together, the result is one. L1 normalisation, also known as a least absolute deviations regression, is similar but for the fact that the squares of each value don't add to one. Rather, a sum of the values themselves adds to one. For the L2 algorithm, data which strays far from the mean (outlier data points) has a great effect on the calculations, as the proportionate value of each item is squared. This makes L1 the more resilient algorithm, as it is not so impacted by the presence of outliers. L2, on the other hand, is more 'stable', meaning that in changing conditions (as the data changes and the same model is applied), the effective change in outcome will be minimised. Each algorithm clearly comes with its own benefits and drawbacks.

2.3 BCI Challenge NER2015

As part of the IEEE Neural Engineering Conference of 2015 (NER2015), a challenge was put out to the world in the form of a Kaggle tournament. Kaggle is a world-renowned company whose community host data analysis and machine learning challenges, competitions and discussions. The challenge went as follows. Twenty-six study participants were presented with a screen and a short word (see Fig. 3 below), before being presented with individual letters one-by-one. In most cases, the letters would spell out the word the users had previously been shown, however in some cases

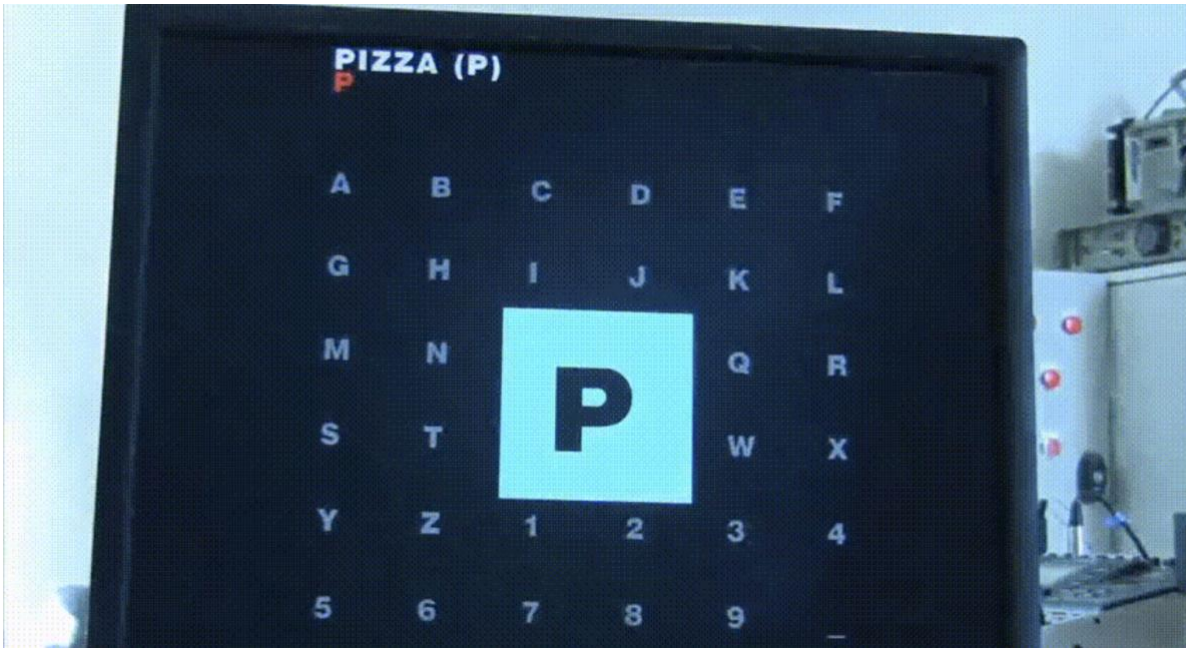


Fig. 3 - NER2015 Kaggle Challenge Spelling Test

the letter would not match. It was a 'spelling error' and served to trigger the 'oddball' effect on the human brain. The brain produces certain recognisable patterns in its brain waves when presented with an oddball – a stimulus which differs to the expected or regular input the brain was expecting. The goal of the challenge was to determine whether the letters on the screen correctly spelt out the word, given only the participants' brain waves. The idea is that participants' neural signals differ so much when presented with an oddball that a model should be able to learn to recognise these signals and, when presented with new data, determine when the subject was in fact presented with an oddball.

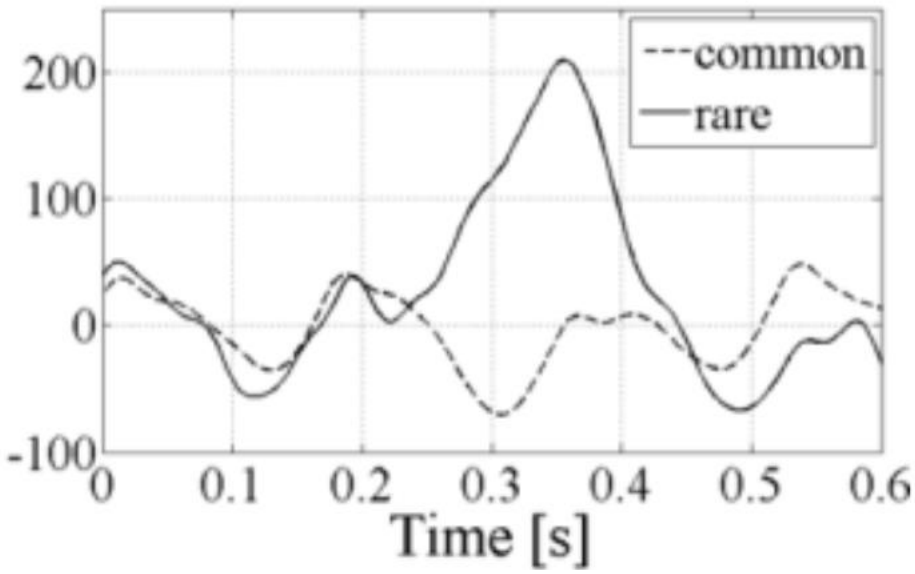


Fig. 4 - The oddball spike

The winners of the challenge were able to successfully predict whether a word was or wasn't spelt correctly at a rate of 85.1% with the following methodology as outlined in their GitHub project (Barachant et al. 2015). Firstly, four steps of data processing are executed. To start, for both the 'correct' spelling class and the 'error' class, xDAWN spatial filters are estimated as discussed in section 2.1 and applied to the raw data. Next, because the data was recorded using 56 electrodes, a channel selection process was applied by determining the Riemannian distance (see Barachant et al. 2011) between the mean of the covariances of each class (correct and error). EEG data can be so noisy that too many data channels (each electrode provides one channel) can actually be a detriment to a model's success. In the third step, covariance matrices of each class are projected into the tangent space (see Barachant et al. 2013), and in the final step of processing, the data is normalised using the L1 normalisation process previously described. Now that data processing is complete, a model is trained using the elastic-net algorithm. Elastic-net is a type of 'regularised' logistic regression, a type of logistic regression wherein overfitting of the model is reduced through usage of L1 and L2 normalisation during training. Elastic-net was chosen because it has a track-record of success on high dimensional data. To-date, the processing and classifying process followed by Barachant et al. appeared to provide a blueprint for the greatest found success in analysing and predicting EEG signals. All processes and code used by the team were published freely on GitHub for anyone in the world to recreate.

2.4 Separable Convolutions

The idea of a convolutional neural network has mostly been used for image classification tasks. The convolutional layer which gives the network its name performs a 'convolution' operation on the input, which involves the application of a filter to the input data (see Fig. 5 below). A spatially separable convolution is a simple iteration on this idea, wherein one filter is split into two, and each of the two new filters are one-at-a-time applied to the data. The technique can reduce the amount of computational complexity involved in a convolution by reducing the number of multiplication operations required between data and filter, however obviously this is only possible if the filter was in fact mathematically able to be divided into two. Oftentimes this is not the case. Another type of separable convolution, the depthwise separable convolution, is able to work with filters than are not so easily 'split' into two, and as such the depthwise variant is much more

commonplace in machine learning than the spatially separable convolution. The depthwise separable convolution deals with not two but three dimensions of data. For image classifiers the third dimension is the colour of the image; there are three channels in the third dimension – red, blue and green.

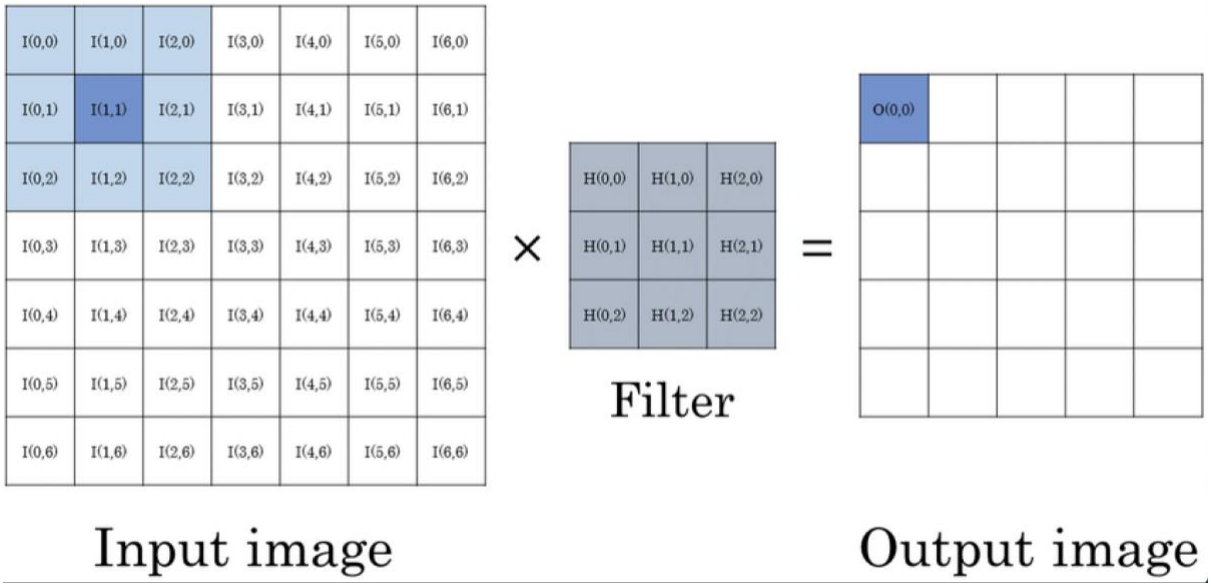


Fig. 5 - Convolution performed on an image

The depthwise separable convolution also splits the filter into two, however they are two entirely different convolutions themselves; one depthwise and one pointwise convolution. The depthwise convolution applies a filter to each layer of the data separately. Convolutions are not fully connected, which is of benefit with EEG data because this lets us use separate convolutions to learn separate spatial filters, which in turn enables us to learn frequency-specific spatial filters (Lawhern et al. 2018). Secondly the pointwise convolution, in contrast to the depthwise, iterates on a single point of data but throughout all channels, or the entire depth of the data. As many of these filters as required are applied such that one applies to every data-point, but again the important part is that this convolution will apply to the entire depth of the data at hand. These two depthwise and pointwise convolutions have iterated over the entire dataset as would a regular convolution, but with reduced computational complexity and most importantly, the process reduces the number of parameters in a convolution. Not every data science project would want to reduce its number of parameters, but in certain cases this can be hugely desirable.

2.5 EEGNet

The EEGNet paper published in 2018 led the way when it came to using AI in the field of EEG in a way that generalised across multiple BCI paradigms, did not require huge amounts of training data and saw highly successful results. Titled '*EEGNet: A Compact Convlutional Neural Network for EEG-based Brain-Computer Interfaces'*, it was the generalising across different paradigms which made the paper especially stand out. Before EEGNet, what was regarded as the best model for ERP modelling was the process undertaken in section 2.3 on NER2015. The elastic-net model uses a logistic regression algorithm using the logistic regression equation, which is relatively simple to implement and understand. The EEGNet is a multi-layer artificial neural network which is generally more suited for high dimensionality problems than logistic regression and is better able to analyse and determine more complex inter-variable relationships.

The project previously discussed from NER2015 described just what it took to get good EEG analysis results up until the EEGNet. As described, there were four quite highly complicated steps in processing the data before the logistic regression model was trained. These steps, at least, are highly documented and relatively easy to follow, but they apply to just one of several BCI paradigms – ERP. For the various other paradigms, of which in future there will only be more, similarly complex but different processing procedures must be carried out to find a good result. EEGNet provides a solution using deep learning to alleviate the need to find, understand and follow such procedures for each paradigm with a robust model agnostic to any one BCI paradigm.

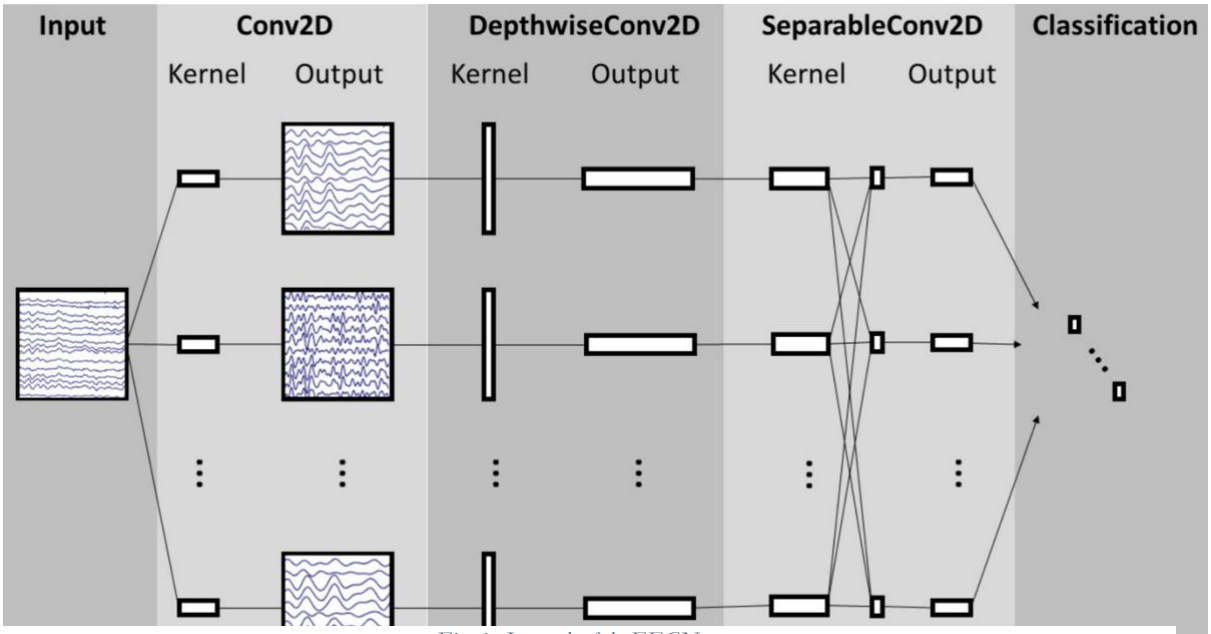


Fig. 6 - Internals of the EEGNet

The neural network applies the concepts behind the rapid rise in successful image classifiers in the early 2010s (as discussed in section 1.2), primarily the CNN with Depthwise and Separable convolutions. Such concepts are now commonplace in the deep learning image-classifying realm but had not previously been widely applied to EEG data.

The EEGNet architecture is comprised of two blocks followed by a third classification block. The first block is made up of a standard 2D convolution set to a length of half of the data's sampling rate, which the researchers found worked to work well in practise. The second part of the first block was is the depthwise convolution which learns a spatial filter. As noted, the depthwise convolution reduces the number of trainable parameters, and in this case, it enables us to learn separate filters for different recorded frequencies. In the second block, a depthwise separable convolution is used, the depthwise part of which learns patterns at different 'time-scales' independently, and then the pointwise part of which combines the learning of those separate feature maps. Finally, the softmax classifier takes the previous convolutions and outputs probabilities for each of the dataset's classes, which become the model's predictions. The researchers used their model on many datasets, but one of those in the ERP paradigm was relatively similar to the dataset used in this project. In their experiment, subjects were presented with a series of images, 20% containing vehicles or people and 80% containing neither. The task of the subjects was to press a button when shown an image containing a vehicle or a person. Obviously, all subjects were connected to a BCI for the experiment, and the recorded data was subsequently downsampled from 512 to 128Hz and filtered to contain only frequencies from 1-40Hz.

2.6 Measuring Success

There is no one golden metric with which one can measure the level of success of an AI model. Rather there are several, each of which bring their own pros and cons and are better suited to different projects. One metric which is most prominent in published studies is accuracy (Handelman et al. 2019). The accuracy of a model's predictions is simply the percentage of data points which have been correctly classified out of the entire dataset. Oftentimes accuracy is in fact a useful metric, and perhaps the best for a given project, however it comes with its flaws. The paper by Handelman et al. provides an example of the classic counterargument to usage of accuracy, an anecdote from an application of AI to the medical field. In their example the model is trained to identify blockages of the lungs, a rare medical issue presented in around 10% of subjects getting a scan done. The model successfully predicts the presence of blockages in test

participants' lungs with an accuracy of 90%, but all the model does is predict every patient to in fact be free of the blockages. All healthy patients were correctly cleared of medical issues, and all unhealthy patients were falsely advised their lungs were clear from blockage. Oftentimes, the paper writes, accuracy is not a helpful metric.

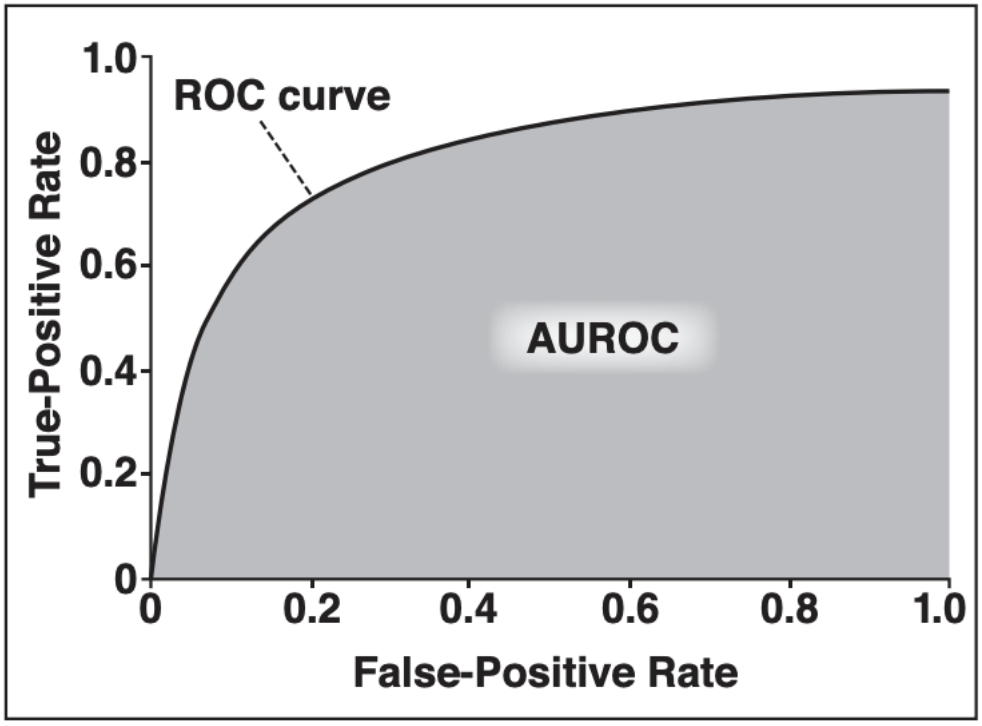


Fig. 7 - An example ROC curve with the AUC highlighted

A common method for avoiding the above issue is the usage of a receiver operating characteristic (ROC) curve (see Fig. 7 above). The RUC plots the relationship that a model's sensitivity has on its specificity. Sensitivity, or recall, measures the ability of an algorithm to correctly predict the actual positive cases (true positives). More strictly, it is the number of true positives divided by the sum of the true positives and false negatives, and is also known as the true-positive rate. In the prior example, sensitivity is 0 because none of the positive cases (patients with blockages) were correctly identified. The specificity of a model is the opposite; that is, it's the number of true negative predictions divided by the sum of the true negatives and false positives, and is also known as the false-positive rate. The model from the previous example had a specificity of 100; that is, it correctly predicted every negative case and didn't make any false positive predictions. Evidentially sensitivity and specificity play off of each other; generally as one rises the other will drop, and for most models some kind of balancing of the two measures is required. In the RUC curve both

sensitivity and specificity are measured, making the Area Under the Curve (AUC) a common means for evaluating the success of a good model. One perk of AUC is that it is agnostic to the classification threshold of a model. That threshold determines what probabilities turn into what predictions; a threshold of 0.5 would mean any input predicted to have a 50% or greater chance of being a positive is in fact predicted as a positive. The perk can, however, be a downside and take away from UAC as a helpful metric, because sometimes the costs of false positives and false negatives are not the same and that threshold should actually be much different. In the medical example, a false positive means a healthy patient will be falsely classified as suffering from one of the lung blockages and undergo further testing (not the worst thing). A false negative tells an unhealthy subject that they are in fact healthy and should not worry about such blockages (the worst thing). Clearly the cost of false negatives in this example is higher, and as such sensitivity would be more important than specificity. An ROC can be a great tool for choosing a threshold and it provides the researcher with the means by which to choose the classification threshold which makes sense for their own project.

3. METHODOLOGY

In brief, the following is my intended methodology, describing what I should do to end up with a successful project. It should be noted that before any of this methodology could be performed, copious amounts of research on the AI concepts, EEG concepts and the applications of AI on EEGs was necessary. As stated, this project's objective was to learn and apply state of the art techniques in machine learning to EEG data with the goals of better understanding the processes for myself and developing a model capable of learning and classifying the data. Before starting development of this model, a fair amount of work is required just in order to have the right working development environment to be able to start exploring and understanding these processes. In this step it would be great to successfully run a model using data found online and get good results classifying EEG signals. The next step would be to get access to my supervisor's data, perform data exploration and analysis, and really understand both the data itself and exactly what experimental procedures were executed in its gathering. The next step would be going back to the model from the setup and doing everything required to apply it to the data. This would mean refactoring parts of the model to suit the shape of the new data, and getting it working on that new data (working meaning it could output a result, but not necessarily a good one). The final step of the methodology is the fine-tuning of the model and exploration of tweaks and modifications which would help the model to get the best classification results possible. The last step was anticipated to be the biggest and most cumbersome.

4. SETUP

4.1 BCI EEGNet

The first milestone was to get a working development environment wherein I could train a neural network on data from a Kaggle competition and recreate the results achieved by those participants of the contest. The model I would be implementing was that used in this GitHub repository - https://github.com/cbhanu/BCI_EEGNet - a model developed for a Kaggle challenge which used the previously described spelling challenge. The repository was a great starting point, however it wouldn't prove a silver bullet solution to getting a model up-and-running easily. Unfortunately the code was already outdated by the time I started looking at it, having been written in March of 2018. Obviously, things move fast in the software world, and new versions of libraries are always being published. Normally this would not be too much of an issue, so long as the writers of the repository provide the exact versions of every library they used when the code was working correctly. The authors of this repository were focused on the Kaggle competition and uploaded their code afterwards but failed to list which versions of what libraries they had used. They did provide some instructions on using their model, but by the time I got my hands on it the code wouldn't actually even compile. It took many hours of research and trial-and-error of mixing and matching different library versions (the authors had used over twenty external libraries) before I finally had something that would compile. Once the program was compiling, I gathered the dataset the authors had used from the Kaggle competition with the goal of recreating the processes and success they had found. At first the program wouldn't run on the data provided; the data format the model was expecting didn't match the data I had gathered. After confirming I had the same data which the authors had used, I set to work debugging the program and eventually found the inconsistencies. Presumably some of the code the authors had used on the Kaggle dataset had not been updated in the repository, but with changes in two lines finally I was able to run the program.

Running the program on my old 2012 MacBook Pro seemed an impossible task. Training the model with just ten epochs took over forty minutes. As mentioned above, part of the huge increase in AI performance in the last decade came from the usage of increasingly powerful GPUs. My laptop had none. Fortunately, three years prior I had built a desktop PC which had much better specifications than my laptop, including its own GPU. Transferring the development environment to the desktop, I found the time required to run ten epochs of the program cut down by 60% to sixteen minutes. It was a huge improvement but still probably not good enough to get the kind of

quick feedback required to develop, train and improve a model. With the ten epochs, the model classified the two classes of data with an accuracy of just 30%. This result was not surprising considering the number of epochs. In a change of strategy, I opted to run the program overnight, this time for 300 epochs. I calculated this would take my machine eight hours, given the ten epochs took just over sixteen. Although I turned the 'sleep' mode off of my computer such that it would stay awake all night, a separate power saving mode kicked in two hours into the learning process, and as such after eight hours only 170 epochs had completed. I changed this setting and carried out the experiment again the next night and finally the model had completed its training and testing process. Using 50% of the data to train, 25% to validate its trainings, and 25% to test the model after the training process, the model achieved an accuracy of 68.3% and AUC of 61.6%. It was slow and not very accurate, but the model was classifying data at a rate better than a random guess. Clearly it was learning something, and I was on the right track. Unfortunately, with eight hours required just to run one experiment, it was going to take a very long time to make any kind of optimisations and fine-tuning of the model. That's not to mention the power costs and computer wear that come with leaving the machine working at 100% capacity every night for eight hours at a time.

At this time through communication with my supervisor I was advised of UTS' own High Powered Computing (HPC) facility which provides staff and researchers with complimentary access to over forty high-powered computers online all of the time. Having already gained experience using the command line through a few years of work experience, the idea of being able to remotely connect via the command line to a high powered computer in lieu of using my own was highly appealing. Once I was granted access to the system by the administrators, I saw immediately that the HPC was a game-changer for me. I could now work on the project from anywhere using only my old laptop, because the UTS computers were doing all of the hard work and all I had to do was send a few commands to tell them what to do. It took some time to get the environment working again, more so than when I had moved to the desktop because this time it was a totally new system I was working with and I had to learn what the HPC had pre-installed for users. Other problems were how to get the program and data I had been using onto the HPC system, and how to get it to persist there so that I wouldn't have to get it there again every time I reconnected to the system. Fortunately UTS' own documentation is quite extensive and eventually I found instructions on how to do just that. I used the Secure File Transfer Protocol (SFTP) to send my program and data over to the UTS computers, stored it in my own persisted storage directory there and was able to run the model there. Finally, the model's training-and-classifying process which had taken all night - eight hours - was reduced in time by 97% down to just twelve minutes. The benefits of such a

reduction in time are hard to understate. With that improvement I would be able to run the entire process multiple times per hour, allowing for quick experimentation with the model's parameters. It opened up the possibility to use an even high number of epochs, smaller batch size or training on bigger datasets, all of which would increase the amount of time taken for the model to train itself.

4.2 ARL EEGModels

I had shown I could perform the machine learning process in my own environment using the HPC to ensure the process didn't take too long and had trained a model on real EEG data showing the model improved with time (i.e., it was really learning). The issue was that part of my goal in the setup was to recreate the results achieved by the authors of the project in their

Kaggle competition. Unfortunately, I soon realised, the authors had not actually listed their results as part of their repository, and looking through the competition leaderboard I struggled to find any entrants whose names matched those of the repository authors. Surely their results were better than my 68% accuracy, but how much better I did not know. That was when my supervisor pointed me to another repository developed by American Army Research Laboratory (ARL) (https://github.com/vlawhern/arl-eegmodels).

Finding the repository was like striking gold. Every external library the authors had used was documented alongside the exact versions they had used. The repository provided a relatively extensive documentation on how to use their code. Two of the stated goals of their project were to 'facilitate reproducible research' and to 'enable other researchers to use… these models… on their data'. The repository, from my experience, did exactly that. Using a real EEG dataset as the sample, the repository provides instructions on how to achieve a good result in classifying the data and also notes exactly the results the authors were able to achieve on the data (93%). The code is also heavily commented with notes and general documentation to assist the reader. Using their model and sample data, I was able to run the training process in around ten minutes (similar to the other project) and recreate the results of 93% accuracy see Fig. 8 below). The primary tools and libraries used were Python3, Keras and Tensorflow, but for a full list of the dependencies I used to get the ARL working see Appendix B. Having recreated the result of the authors, I had achieved my goal and as such the setup phase of the project was complete.

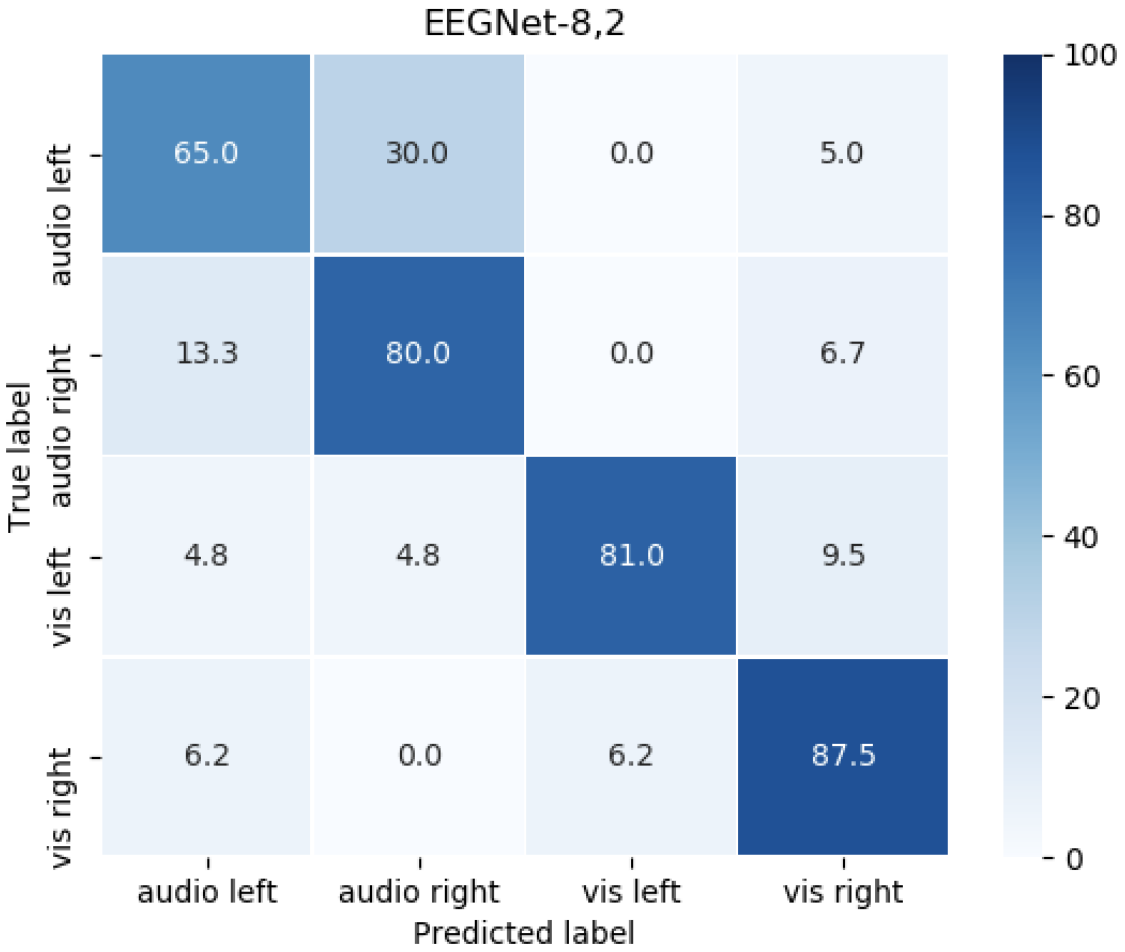


Fig. 8 - Confusion matrix of results on sample data

5. DATA EXPLORATION

Finally, at this point I was ready to being working with my supervisor's data. The dataset was used in 2015 in one of his papers (Lin et al. 2015) titled '*Extracting patterns of single-trial EEG using an adaptive learning algorithm*'. The research paper studies the rapid serial visual presentation (RSVP) paradigm, wherein participants are presented with rapid sequences of images including one 'target' image. In the experiment at hand, subjects were presented with series' of letters on a screen, each letter being presented for a duration of 200 milliseconds, i.e. five letters per second (see Fig. 9). Roughly 5% (one in twenty) of the letters presented were designated a 'target', and the subjects' task was to, upon recognising a target, press a button. The recognition of the target image triggers an oddball response in the brain as described previously, and the action of moving the finger to press a button fires even more neural signals.

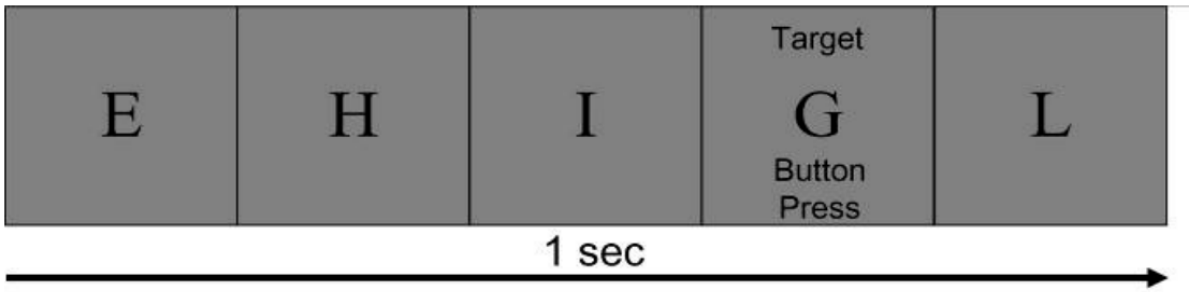


Fig. 9 - A visual representation of the experiment by Lin et al. showing letter 'G' as a target

The six study participants were fitted with an EEG cap composed of 32 electrodes to be placed over the skull. Each subject participated in either three or four sessions, each of which consisted of 80 target letters and over 1200 non-targets. Thus, in total the dataset consists of 19,215 trials, each labelled as either a target or non-target event, and the job of the prediction model is to classify signals into one of these two classes. The data distribution is represented below in Fig. 10. The most striking attribute is the class imbalance, a consequence of the ~20 non-target letters presented to subjects between their being shown a target. It stands in contrast to the data on which the model scored a 93% accuracy rate, which is presented below in Fig. 11. The four classes of that dataset are split almost exactly into four even chunks of 25%.

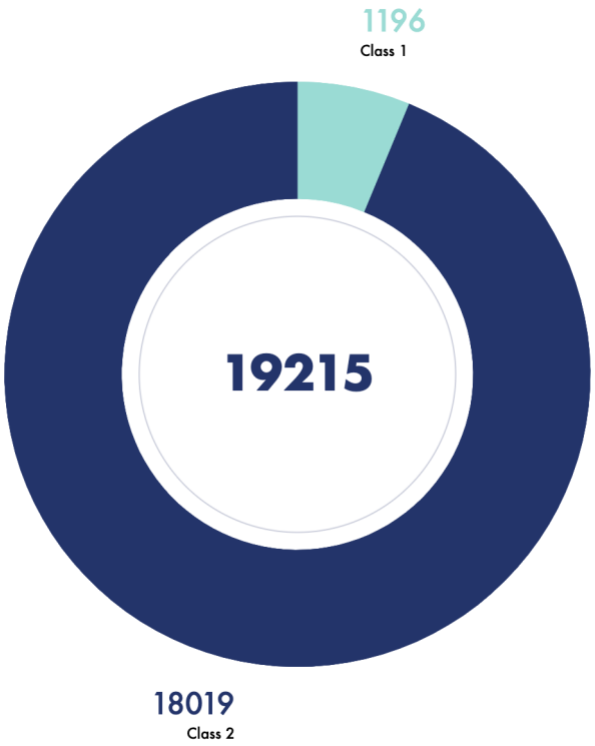


Fig. 10 - Data distribution between target (class 1) and non-target (class 2)

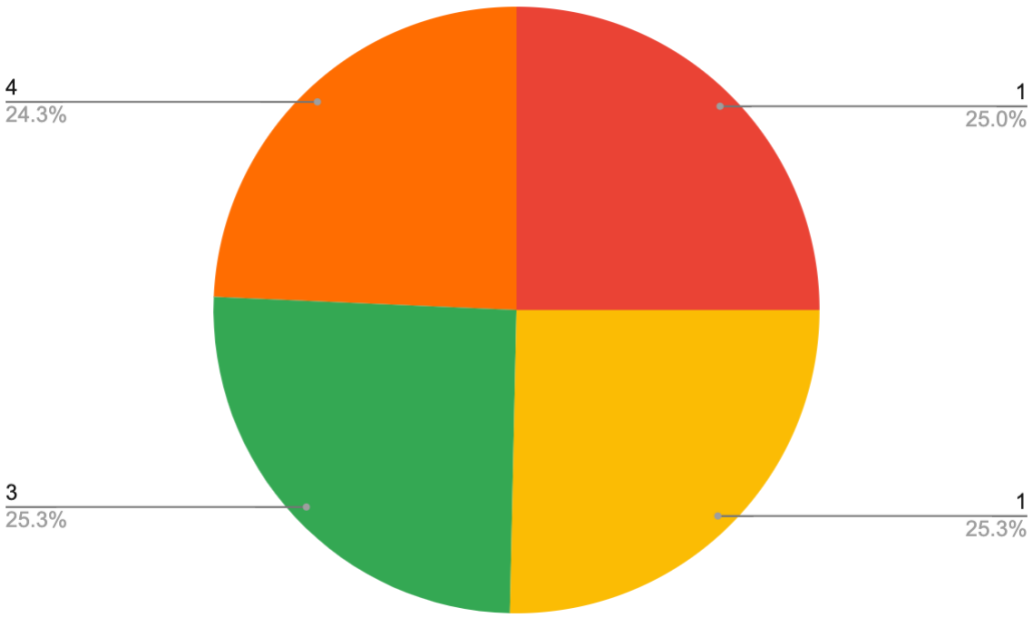


Fig. 11 - ARL sample data distribution

Immediately it's obvious that the RSVP dataset (that of my supervisor) is far more fleshed out than the example found in the ARL code. Its trial count alone of 19,215 puts it far above and beyond that of the sample data's count of 288. Even if the RSVP data was balanced like the sample one by trimming non-target data from the dataset, the dataset would still contain 2392 trials (1192 each of target and non-target). The other obvious difference is that the sample has four classes. That's because in the sample dataset the study participants were being subject to a stimulus to their left ear, right ear, left eye or right eye, hence the four classes. The RSVP dataset is simpler in that there are only two classes to learn: target and non-target. The final major difference in datasets I've observed was actually the paradigm which was being explored with the EEG. While my supervisor's data, as mentioned, works in the RSVP paradigm, the sample data was in the paradigm of Event Related Fields (ERF).

The RSVP dataset, as mentioned, displayed each image for a total of just 200 milliseconds before moving onto another image. The oddball paradigm as described before famously triggers what's known as the P300 response, its name coming from the fact that an obvious disruption occurs in subjects' brain signals not at the time of sensing an oddball, but around 300 milliseconds after the fact. In this RSVP dataset each image is shown for only 200 milliseconds, so to get a good reading of the effect that seeing a target image and pressing a button had on subjects, clearly we would have to look past the end of the time the target was shown by at least 100 milliseconds, such that at least 300 milliseconds had passed since the subject first registered the presence of a target. In fact, fortunately for me, the dataset had already undergone some heavy pre-processing to take all of this into account. As it was provided to me, the dataset was composed of the 19,215 trials previously mentioned. I had at first assumed each trial's datapoints to include only the 200 milliseconds measured while the subject was looking at a particular image. With the BCI measuring data at a rate of 256Hz, this would leave me with 51 datapoints per trial, or 51 'samples. In fact, this was not the case. Exploring the data, I had 231 samples per trial. The data had already been processed such that the data for each trial started 100 milliseconds before the image was shown up until 800 milliseconds after the fact. Thus, I had 900 milliseconds of data per trial, which with the sampling rate of 256Hz left me with the 231 samples.

Another presumption I made was in thinking that, even though the data had already clearly been processed in a beneficial way, the ordering of the trials would represent the ordering from the real experiment. That is to say, the data would consist primarily of non-target classes with sporadic target classes roughly every twenty trials. In fact, as I will discuss later on, this presumption would go on to burn me in the attempt of fitting a model to this data, because the presumption was not correct. For every session, comprising around 1300 trials, all eighty of the target classes were right

at the end. That is to say, each group of session data consisted of around 1200 trials of non-targets followed by 80 targets. This was a very important discovery in the process of getting to a good result using the EEGNet.

In section 2.3 on how the winners of the competition from NER2015 came to their winning model I wrote of the issues that can come with analysing and learning EEG data when the data is recorded with a high number of electrodes. The recordings can just be too noisy to properly learn, and the extra noise can make generalising across subjects near impossible. A model trained on one subject might anyway have a hard time classifying signals from a different subject, and that is only truer when a high number of channels is used. With more channels, it's more likely that the model could come to learn patterns which are specific to that specific subject, and as such the model will be 'overfitted'. In this case the number of electrodes used was 32, so that's the number of data channels in the dataset. In their entry for the NER2015 contest the authors of the project I reviewed reduced the number of channels in their data from fifty-six using a selection process based on the Riemannian distance algorithm. In this case, I started off using all channels in case the result was good, with the decision to consult later with my supervisor if the results weren't good about any advice he might have for choosing channels which would get the best result.

6. Model Fitting

At this point it was time to use the RSVP data and fit the EEGNet implementation from the ARL to that dataset. Compared to working with the initial 2018 repository (BCI_EEGNet) the process was rather simple and straightforward. The code and data were transferred onto the UTS HPC system using SFTP ready to go. I had to get the data ready to go in the format that the ARL model wanted it. The data had 19,215 samples, 32 channels and 231 samples per trial, in a data shape of [32, 231, 19,215] whereas the ARL model wanted it in a shape of [trials, kernels, channels, samples]. The only real work to do to get the model actually running with the data was reshaping the data into this format. Previously the shapes of the data were hard-coded based on the sample data the authors had used. Instead of changing these figures to match the data I had, I refactored the code such that the dimensions (channels, trials, samples) were calculated dynamically based off of the input data (see Fig. 12). Later on, once I started experimenting with changing the data shape, this saved what would have been a lot of time changing the hard-coded values every time.

```
51
52    def channelsSamplesTrialKernels(data):
53        return data.shape[0], data.shape[1], data.shape[2], 1
54
```

```
self.chans, self.samples, self.trials, self.kernels = channelsSamplesTrialKernels(data)
X = X.reshape(self.trials, self.kernels, self.chans, self.samples)
```

Fig. 12 - Modifying the ARL code to dynamically determine the paramters of the data's shape

The model then was ready to run with the RSVP data. On the HPC system I was just using the command line rather than the graphical user interface (GUI) so all of these code changes I actually made on my own computer using an integrated development environment (IDE). I experimented with using a Jupyter Notebook as is the norm in machine learning. The notebook is handy in that it presents a very user-friendly way to run Python scripts in little chunks and with repeatability. For me, I am very comfortable with IDEs and wanted a way to do my development in one, save every modification with version control, and then run it on the HPC. It was crucial that I could do all of this quickly; make a change, save it and run the code. Therefore, at this point I turned my modifications of the ARL code into a GitHub repository (https://github.com/WillSmithTE/arl-eegmodels). The benefits of this approach were that I could do all development on my own machine using the setup I am comfortable with. Then once I was ready to run the code, I could

simply 'save' it by making a commit, push it to my repository, pull the changes on the HPC and run the code. The process of saving it on my computer and getting it on the HPC took only ~15 seconds and let me track every change I made. It created a clear history and log of every change I made and let me have the freedom to change the code however I wanted always knowing everything was saved and I could get back to a previous state at any moment. The setup also meant my project lived online, and as such I wasn't limited to just one computer which proved handy when I did in fact have to switch between computers due to hardware failure on one. The GitHub repository also served as a backup, giving me the confidence that at no point could I lose my progress.

Running the model with exactly the same parameters as were used on the sample ERF data was my first attempt at learning the RSVP data. The accuracy on the ERF was high and, even though the data was very different, I thought it would be a good place to start. I had again split the data into 50%/25%/25% sections for training, validating and testing respectively, using all subjects together (known as cross-subject validation). Although cross-subject validation is often seen as more difficult, in this case it would allow me to use all over-nineteen-thousand trials and would anyway serve as a starting point for making sure the setup of the model and the development environment were correct. With so much data, a batch size of just sixteen and 300 epochs the training process took around twenty minutes on the HPC. The accuracy of the model from the first run was 92.9% which seems fantastic, however the confusion matrix in Fig. 13 tells the underlying story.

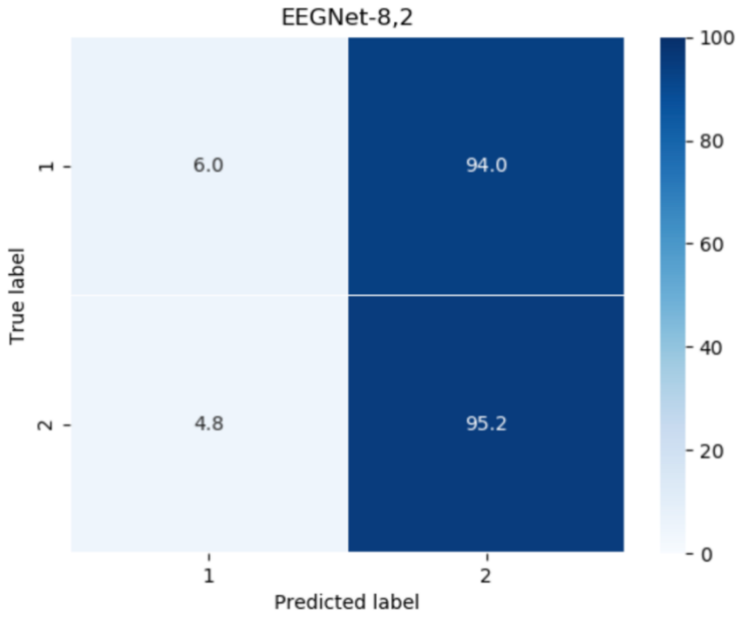


Fig. 13 - Confusion matrix from first run of EEGNet on RSVP data

In the confusion matrix class 1 is the target and class 2 is the non-target. I would later switch these numbers to be in line with common practise, but the important takeaway from the confusion matrix is that 'true positives' in the top left, wherein the target class was correctly predicted, made up just 6% of the total positive results. All 94% of the other target classes were predicted as non-targets. In raw numbers, just 18/288 target classes were predicted successfully. The overall accuracy of 92.9% masked the fact that the target class was predicted successfully at an accuracy of just 6%. Clearly, and as expected, plugging the new data into the ARL model with the same parameters would not prove a magic formula for learning the RSVP data, however running the model on the RSVP was a success.

7. Model Improvements

7.1 Human Error

Here I have to confess a significant human error I made early in the project which had effects carrying over right all the way until a week before this report was submitted. It affected the most important part of what I was spending all of my time on: measuring the successes and failures of the model in response to changing parameters. When I first started computing the AUC, the ROC I had calculated from which the AUC score was determined was not quite right, and I only realised the error when I decided to actually plot the ROC for my own reference.

The ROC, I have already described, plots true positive rate against false positive rate for various thresholds of predictions. In my code, the threshold I was using for predictions was 0.5, but in fact unbeknownst to me I was using the same threshold in my ROC. Rather than plotting 200 different thresholds, I plotted just one, resulting in a graph with just three data points (see Fig. 14). The correct ROC plotted 200 different thresholds to provide an idea about the balance we have to manage between sensitivity and specificity (see Fig. 15).

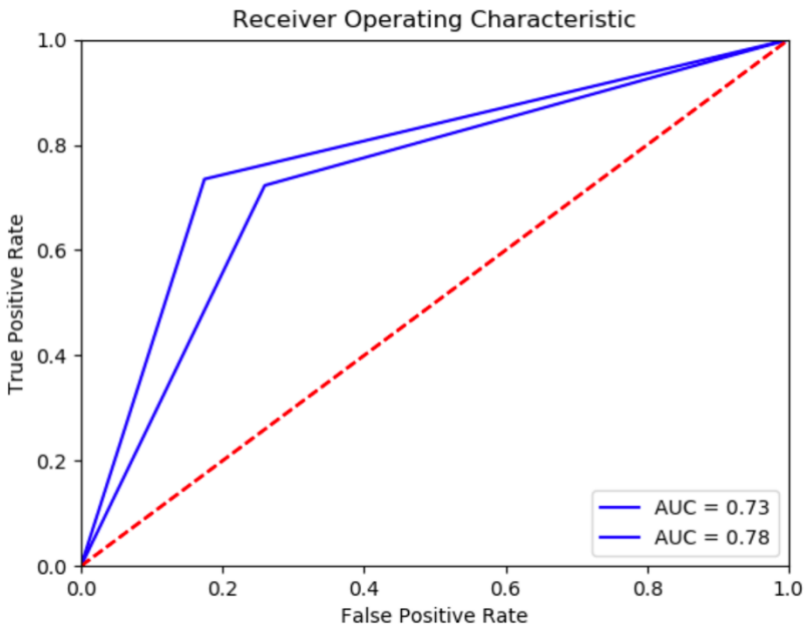


Fig. 14 - The incorrect ROC used to calculate the AUCs

I executed about 200 experiments before realising my mistake. As such, the AUC scores for these 200 experiments were lower than they should be. In this report, I decided to continue to use these

AUC scores to compare the models. As far as the experimenting and tweaking goes, the only goal is to improve performance, so the error should have minimal impact on the conclusions drawn from the following experiments. I plan to carry out all experiments as planned and complete them using the lower, incorrect AUC score. Then I will use the takeaways from the experiments to try to maximise the *real* AUC score and use that for my final result. I admit this was a sloppy mistake from me and acknowledge that the most scientific thing to do would be to restart all experimentation using the real AUC result.

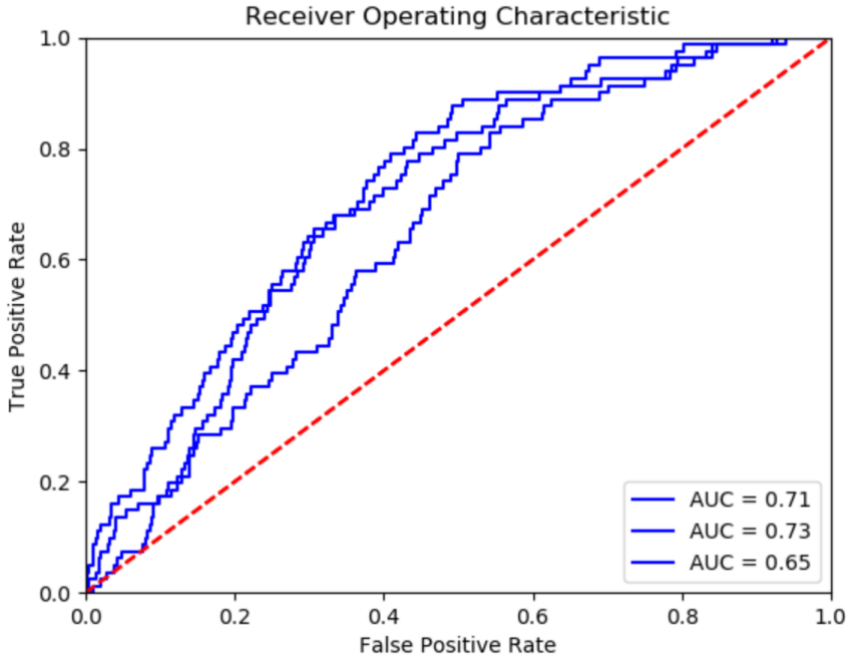


Fig. 15 - The correct ROC

7.2 Class Weights

Clearly work had to be done to improve the model. The problem ties back to the issue described back in section 2.6 on the woes of accuracy as a metric. With such an imbalance between target and non-target classes, the model quickly learns it can score high accuracies simply by predicting the non-target nearly every time. As the model tries to learn through backpropagation, any progress on learning patterns which might help with predicting the target classes is nullified because the accuracy goes down, the loss from the loss function goes up and the model assumes there's no further improvement to be made that way. In a way it's right. As long as the loss function treats classifying target and non-target datapoints in the same way its methodology is the best one. That's

why I had to change how the loss function determined the success of the model's predictions. By adding 'weightings' to the class, the model would know in learning that despite their making up only 6% of the data, predicting the targets was just as important as predicting the non-targets. I added weights equal to the opposite of the distribution of the data dynamically in Python, such that correctly or incorrectly classifying a target image was fifteen times more influential to the classifier's loss than classifying a non-target. See the code in Fig. 16.

```
def getClassWeights(data):
    return dict(enumerate(class_weight.compute_class_weight('balanced', np.unique(data), data)))
```

Fig. 16 - Calculating weights for each class based on the data distribution

Running the model again with the same parameters except for the change in weights, there was a notable improvement. Running the program four times and averaging the results, the accuracy had dropped to 81% but reading the confusion matrix showed the model to be better at classifying the target class than it was before. This is when I came to the conclusion that I needed a proper metric to determine whether the model was getting better or worse. I had made a tweak which I knew improved the model, but my only metric (accuracy) told me performance had decreased. If I was going to be able to continue to perform extensive tweaking and tuning of the model, I had to find a quantitative way to track progress.

From the research I had done up until this point I thought that a metric which would help as a good start would be AUC, as discussed in section 2.6. Unlike accuracy, it would actually take into account the true positive rate and the false positive rate, which would mean the first model (predicting only non-targets) would not score well, and predicting more successful targets would score better. I tested it on the two results I had so far and was happy with the result. Both of the AUC scores were low obviously, but at least the second was better than the first. The first had scored 52.3 and the other 56.0.

7.3 Kernel Length

At this point I went back to the EEGNet paper and the documentation for the ARL project. My supervisor's continual advice was to get familiar with the theory and he was 100% correct. Every time I went back and re-read the paper or the documentation something new stood out to me that I had missed. This time it was the 'kernel length'. What the ARL authors referred to as kernel

length was the size of the temporal convolution in the first layer of the CNN. In their example, and in the code I had used, they had set this kernel length to a value of 64. Upon reading through their documentation I found this to be because of the sampling rate of the ERF data they were working with. The data had a sample rate of 128, and the researchers had found best success when the kernel length was set to exactly half of this figure. I was also using the kernel length of 64, however I had 231 samples in my dataset. Clearly this was incorrect and against their recommendation. To avoid repeating the same mistake, I made this value instead dynamically determined at runtime by the program based on the dataset's sample rate (see Fig. 17). In my case, it would be set to 115.

```
if kernLength is None:
    kernLength = int(self.samples/2)
```

Fig. 17 - Dynamically determining the kernel length based on the sample rate

Again over four runs of the model, I averaged the AUC scores of the predictions with this change. The score was better (56.8 compared to 56.0) but not by enough to rule out the possibility that the improvement was nothing more than random luck (obviously each run of the program gets a different result). The model was still having terrible issues predicting the target classes, despite the class weightings and this kernel length fix.

7.4 Batch size

At this point I started to fiddle around with the batch size. I was training the model in batches of 16 but for no particular reason. Having re-read the EEGNet paper and ARL documentation, I found no mention of the batch size there like I had for the kernel length, so figured there was no particular reason why a batch size of 16 should work for me compared to something else. At first, I tried the extremes. I tried a stochastic approach with a batch size of one. I was still working with all subjects' data from all of their sessions, so for the model to train in batch sizes of one this training process took a very, very long time so I reduced the number of epochs down to fifty. Even still the process took over forty minutes, but finally I had a result. The accuracy looked great, at 94.9%, but the AUC had dropped like a rock down to 50.1 (just about the worst AUC one can get). Digging into the result, I found the confusion matrix as per Fig. 18. Notably the confusion matrix here is using the raw values rather than the percentages in the previous confusion matrix. This was a deliberate choice, as I was finding myself converting the percentages into raw figures

anyway. From this point on in the project I preferred the matrix showing just the raw values, even if it doesn't look as aesthetic as Fig. 12.

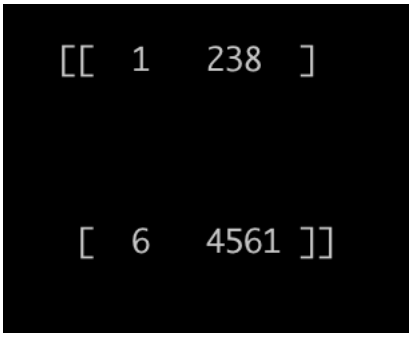


Fig. 18 - Stochastic descent's confusion matrix

One target class had been correctly classified, for a target accuracy of 0.4%, effectively 0%. I tried to find examples of the stochastic gradient approach in use with CNNs but did not find much, so I was confidently able to write that approach off. To try the other side of the extreme, I experimented with a batch gradient descent wherein the batch size makes up the entire dataset. I anticipated the learning process to be extremely quick here so increased the number of epochs back to 300 and otherwise kept all parameters the same. The result was an accuracy of 93.2% and an improved but still unacceptable AUC of 52.0. Finally, I tried a batch size of 80 and got my best result yet, with an AUC of 57.2 and a true positive rate of 27.6% (Fig. 19).

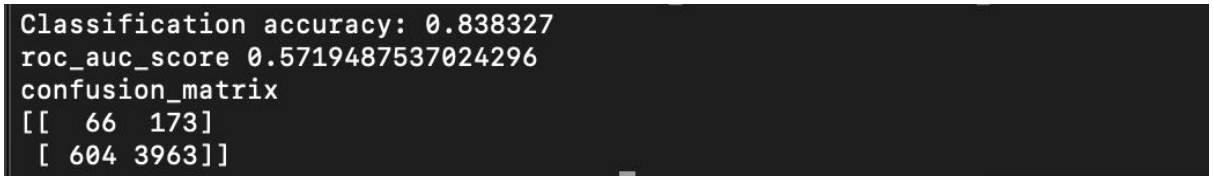


Fig. 19 - First attempt of batch size 80 resulting in the highest AUC yet

Seeing this result I decided to do more exploration of the RSVP dataset. Up until this point I had thought the target classes to be spread evenly throughout the dataset. This is the moment where I realised they were not. That would explain the terrible results with the targets with a batch size of 16, and I was in fact surprised that using a batch size of 80 even managed to get the AUC result it did (admittedly it was still not good). With the target classes grouped at the end of each session's data, the model was going through hundreds of batches before even seeing the existence of any

target classes. As such, the model learnt quickly that it could achieve great accuracy by predicting only non-targets. When the targets finally entered the fray, they weren't able to have enough impact on the model's loss to drastically change its internal weightings and model. Thus, it would stick to predicting non-target classes.

7.5 Subject-wise classification

The small tweaks which were having such negligible improvements on performance up until this point were clearly helping but not in a way big enough that performance was going to get to truly good or even acceptable levels. Authors of papers I had read were able to achieve AUC scores in the mid 80s. My supervisor and assistant advised me that it would be wise to try the classification on just one subject rather than all five together. In deep learning the size of a dataset is often extremely important, which is why I wanted to combine the data of all subjects initially, but fortunately they advised me that often cross-subject classification can be more difficult that subject-wise (one subject) despite the reduced size of the dataset because the brain signals across subjects can vary so much. In order to switch from all subjects to just one I of course had to make some changes to the code. I wated to make the changes but in a way that would easily let me switch back to all subjects, or just subject two, or any other such combination, with minimal work required. Therefore I created common functions in separate files for different datasets. As such I did some minor refactoring of the all-subjects code such that from the EEG code, I could simply call a *getDataAndLabels* function and get back all of the data I needed (see Fig. 20). This way I could write the subject one code in a separate file also with a *getDataAndLabels* function, and easily between datasets by changing the name of the file imported (i.e., change *getDataAndLabels1* to *getDataAndLabels1Subject1*. All other code would stay the same.

```
from getDataAndLabels1 import getDataAndLabels, channelsSamplesTrialKernels
```

Fig. 20 - New way of switching between datasets

At first I did the same classification process from before but just on subject one and with a batch size of 16. Immediately I found the same issues I had faced before with such a low batch size; the target class was almost never predicted (Fig. 21).

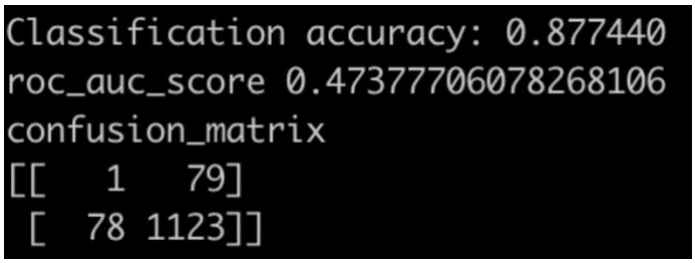


Fig. 21 - Subject 1 classification with batch size 16

This was to be expected, however switching back to the batch size of 80 I had used for all subjects got a better result. The AUC went up to 58.1, right around the same score as was found for all subjects. I knew at this point I was pointlessly collating all of the data together for every run of the model, when I could just collate it once and save it. For that reason, I wrote a basic 'save' and 'read' functionality so that I could cache the data and save time on every run (Fig. 20).

```
import pickle

def save(data, filename):
    file = open(filename, 'ab')
    pickle.dump(data, file)
    file.close()


def read(filename):
    try:
        file = open(filename, 'rb')
        data = pickle.load(file)
        file.close()
        return data
    except:
        return None
```

Fig. 22 - Helper functions for caching data run-to-run

One issue identified which could be harming the success of the model was the fact that the target values were so concentrated together (all 80 put at the end of each session). In an attempt to remediate that I tried to shuffle all of the data before performing any kind of training or classifying. For all subjects the shuffling significantly decreased the performance, perhaps due to the fact that the subjects' data was now interspersed. The AUC went down to around 50. To the contrary, the subject-wise classification actually improved the AUC score, so from here on I always shuffled the data for subject-wise classification for subject-wise data but not when classifying cross-subjects.

7.6 Butterworth Filter

More advice from my professor was to try applying a filter to the data. I had read about usages, specifically in EEG, where machine learning experts had applied a Butterworth filter to reduce noise picked up in EEG detection. The technique would be especially useful for the cross-subject classification, because it would help to block out those signals unrelated to the experiment at hand and which would otherwise cause the model to overfit on the training data and potentially learn the signals of specific subjects. Obviously learning the signals of specific subjects would be useful in subject-wise classification, but it would make the model incapable at generalising across subjects.

I later found a great implementation by an external library for the filter in Python, but the first few implementations I found were highly tailed to the individual dataset they were working on, so at first I wrote one instead in MATLAB (see Fig. 23). The program would take in my dataset, apply the filter between frequencies of 0.5 and 30Hz, and save the result to a MATLAB file. I could then take that MATLAB file and use it as the input for my data.

filter_dataset.m

```
load 'data.mat'
Signal=double(data);

Filtered_signal=zeros(32, 231, 19215);

figure;
for char=1:19215

    for chan=1:32

        x11=Signal(chan,:,char);

        disp([char,chan]);

        sampleRate = 256; % Hz
        lowEnd = 0.5; % Hz
        highEnd = 30; % Hz
        filterOrder = 2;
        [b, a] = butter(filterOrder, [lowEnd highEnd]/(sampleRate/2)); % Gene
        Filtered_signal(chan,:,char) = filtfilt(b, a, x11); % Apply filter to


    end
end

save('filtered', 'Filtered_signal')
```

Fig. 23 - MATLAB Butterworth Filter

The filter seemed to have some success. Over four experiments, the AUC score hit 60 for the first time. My supervisor advised I could make the filtering even tighter, at a range of 0.5-20Hz now. The tweak also had immediate results, with an AUC score of 60.9. I tried the same filter on data from all subjects and had, again, the highest AUC score yet of 64.0, with an accuracy of 82.4%. Fig. 24 below displays a graph of the loss throughout the training process to achieve this result. Visibly, the loss varies greatly throughout the 300 epochs, which is a part of the reason why the model has so much variability in its results run-to-run.

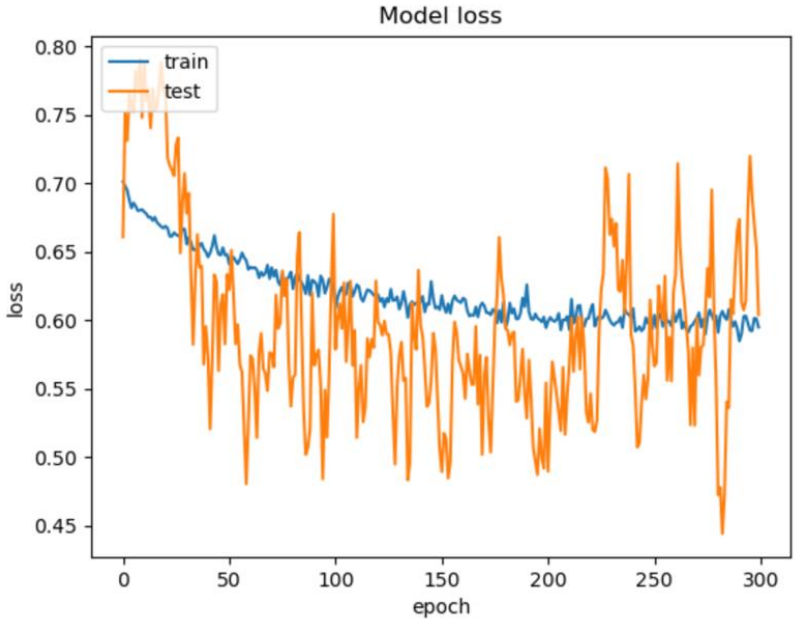


Fig. 24 - All subjects filtered loss graph

7.7 Downsampling and channel filtering

What came next were two improvements which helped the performance of the model hugely. I knew about the effects that too many channels can have on the model's performance as I had read about them before, and I found another EEG project wherein they selected 19 out of the 64 electrodes used, using only these 19 for the classification. In the RSVP experiment, 16 of those 19 electrodes were used so I tried filtering my data such that only these 16 channels of data were included. Again the results were good, with a small increase in the AUC score. I thought about determining optimal channel selection using the Riemannian distance as discussed previously, but just as I was looking into it my supervisor advised me of six channels he thought would likely contain the most relevant and least noisy signals. These were:

- Pz
- P7
- P8
- Oz
- O1
- O2

Before trying the Riemannian distance calculation, I tried using manual selection with these suggested channels and my professor was right. The results were even better.

One of the final significant improvements I saw in the AUC score was made by downsampling the data. Reading through the EEGNet paper I saw multiple references to an ideal sampling rate of 128Hz. Using the rule of the ARL repository, this would put the length of the first convolution at 64 (half of the sample rate), making up 500 milliseconds of recorded data. The more I read the more it seemed that the other parameters (especially the numbers of filters/convolutions chosen) all related to this 128 or 64. Because the RSVP data was 900 milliseconds of data sampled at 256Hz, I had the uneven number of samples of 231.

I decided to try downsampling the data to 128 data points. I could have altered the number and size of filters the model was using but figured that would leave more room for error than downsampling, acknowledging that by downsampling from 231 to 128 I was losing a lot of information (nearly half of the data). The downsampling proved to improve the AUC yet again to just below 70 and, on some runs, just over 70. For the downsampling procedure I used a popular Python library called 'scipy' which provides a module for working with data signals (see Fig. 25).

```
def downSample(data):
    reshaped = data.reshape(data.shape[2], data.shape[1], data.shape[0])
    samples = 128

    Signal_A=np.zeros([reshaped.shape[0],samples,reshaped.shape[2]])

    for i in range(0,reshaped.shape[0]):
        Signal_A[i,:,:] = scipy.signal.resample(reshaped[i,:,:], int(samples))

    Signal_A = Signal_A.reshape(data.shape[0], samples, data.shape[2])
    return Signal_A
```

Fig. 25 - Data downsampling code extract

8. Experimentation Framework

8.1 Building the framework

At this point between tinkering with the batch size, the number of epochs and whether or not data was filtered, I was spending a lot of time setting up and then monitoring these 'experiments'. The whole goal was to maximise the UAC by experimenting with various parameters, so I wanted to find a better way of running more experiments for a longer period of time. It was too time-consuming to wait the ~10 minutes for one to finish, manually record the AUC and accuracy (as I did in the above screenshots), and then run another one. The problem was even more pronounced because I needed to run each experiment multiple times, usually four, just because the results varied so much run-to-run and I wanted to flatten out some of the variability.

```
1   from MyERP import ERPExperiment
2   from csvUtil import writeRow
3
4   experiment = ERPExperiment()
5   experiment.multiTrainAndPredict(F1=16)
6   experiment.multiTrainAndPredict(batchSize=500, F1=16)
7   experiment.multiTrainAndPredict(batchSize=2000, F1=16)
8   experiment.multiTrainAndPredict(batchSize=3000, F1=16)
9   experiment.multiTrainAndPredict(batchSize=4000, F1=16)
10  experiment.multiTrainAndPredict(batchSize=10000)
11  experiment.multiTrainAndPredict(batchSize=10000, F1=16)
12
13  experiment.multiTrainAndPredict(F1=16, D=1)
14  experiment.multiTrainAndPredict(batchSize=500, F1=16, D=1)
15  experiment.multiTrainAndPredict(batchSize=2000, F1=16, D=1)
16  experiment.multiTrainAndPredict(batchSize=3000, F1=16, D=1)
17  experiment.multiTrainAndPredict(batchSize=4000, F1=16, D=1)
18  experiment.multiTrainAndPredict(batchSize=10000, F1=16, D=1)
19
```

Fig. 26 - Using the experimentation framework

Once it was complete, Fig. 26 shows the experimentation framework in practise with configured variable batch sizes. I had been running so-called experiments up until this point (in fact, I had run over 100 of them), but from this point I strove to formalise the process. I explicitly stated my hypothesis, ran the experiment which would keep all variables constant except for that being

tested, saved the results, and compared them with a table or graph. The constructor of the ERPExperiment class simply got the data and put it on the class, as per Fig. 27.

```
class ERPExperiment():
    def __init__(self):
        # get the data and put it on the
        # class so we don't have to get it
        # for every run of the experiment
        self.initialiseData()
```

Fig. 27 - ERPExperiment Constructor

For actually performing the experiments, the method *trainAndPredict* trains the model with the given parameters and executes the full classification process. The parameters available for experimentation are as follows:

- epochs – number of epochs to run
- batchSize – the size of the batches to run
- class_weights – the weightings to attach to each predicted class, as discussed
- F1 – the number of temporal filters to learn on the model
- D – the number of spatial filters to learn within each temporal filter
- kernLength – the size of the temporal convolution in the first layer
- dropoutRate – the percentage of datapoints not included in the training process
- learningRate – the rate at which the model adjusts itself during training

I provided all parameters with default values as per Fig. 28.

```
def trainAndPredict(
    self,
    epochs = 300,
    batchSize = 1000,
    class_weights = None,
    F1 = 8,
    D = 2,
    kernLength = None,
    dropoutRate = 0.5,
    learningRate = 0.001,
):
```

Fig. 28 - Experimentation framework parameter default values

Finally I had the framework I wanted and could run experiments much more easily. In the final step, I wrapped the above *trainAndPredict* in a *multiTrainAndPredict* which was a method with the same definition except for an additional *numberExperiments* property which defined how many times the experiment should run. In this way, for flattening the run-to-run variability, I could easily run the experiment multiple times in one go. For tracking the results of the experiment, I created a master spreadsheet and an ephemeral CSV file where results and parameters were logged (see Fig. 29), and from which I would fill the master spreadsheet once the program had run its course.

```
1  date,epochs,dataset,batchSize,sampleRate,kernLength,dropout,learning,roc_auc,accuracy,f1,D
2  19/02/21,300,all,16,231,115,0.5,0.001,0.707848061877667,0.64,8,2
3  19/02/21,300,all,16,231,115,0.5,0.001,0.7190722795163584,0.6333333333333333,8,2
4  19/02/21,300,all,16,231,115,0.5,0.001,0.7585570768136557,0.68,8,2
5
```

Fig. 29 - Results CSV filled automatically from the experimentation framework

Using this system I had a totally new way to run long experiments. Each individual experiment could take anywhere from less than one minute to over ten. At one point, as I had when using my desktop, I left the program running overnight, but this time instead of running through the classification process just once, the program executed more than 80 experiments and logged them all into the results file.

8.2 Interpreting Results

Now I had a way to quickly run large amounts of tests with varying parameters, I had to be able to interpret the results of the CSV file. For this I used the 'pandas' Python library which is fantastic at working with CSV data. I used two methods depending on the experiment (see Fig. 30).

```
import pandas as pd

def getCorrelations():
    df = pd.read_csv('~/Downloads/results_all.csv')
    return df.corr()

def getGroupedStats(data, column = 'Batch size', sortColumn = 'roc_auc'):
    return data.groupby(column).mean().reset_index().sort_values(sortColumn, ascending=False)
```

Fig. 30 - Interpreting experiment results

The first listed correlations of different columns in the CSV. I used this when running many experiments and generating lots of data. It was less exact, but allowed me to correlate, for example, a bigger batch size with a higher AUC. The second was more precise and I used it much more. *getGroupedStats* would take in the CSV, group the results for a particular column (taking the average), and provide the results in a list ordered by the AUC score. For an example of the output see Fig. 31. The experiment compares batch sizes of 80 with those of 160 and orders the result by the AUC score.

	batchSize	epochs	sampleRate	kernLength	dropout	learning	roc_auc	accuracy	f1	D
1	160	100	231	115	0.5	0.001	0.787283	0.681440	8	2
0	80	100	231	115	0.5	0.001	0.780144	0.686434	8	2

Fig. 31 - Example of grouped stats after an experiment

9. EXPERIMENTS

Below is a sample of some experiments I ran along with my hypotheses and the results.

9.1 Batch Size

Test batch sizes of 16, 32, 64, 128 and 256 on just one subject. For each batch size run the program four times and average the result. It's believed that 256 will have the best score because it will allow for a healthy of target and non-target classes in each batch.

Batch size	Average AUC
16	52.1
32	60.1
64	70.1
128	69.9
256	68.2

The hypothesis was incorrect. The model using batches of 64 scored the highest. It's believed that this is because, due to the low batches, the training process was noisier and thus the model was able to better generalise onto the new data.

9.2 Epochs

Test epochs of 100, 200, 300 and 400 on just one subject. For each epoch run the program four times and average the result. It's believed that 400 will score the best result. From experience, anything above 400 and the model will be overfitting, while 400 while allow the model to iterate enough times to develop a solid model.

Epochs	Average AUC
100	66.3
200	69.0
300	69.1
400	69.8

The hypothesis was correct. Up until 400, more epochs meant a better result.

9.3 Sample rate

Test downsampling the data again to 64Hz this time. For each epoch run the program four times and average the result. It's believed that the 128Hz downsampling will achieve the best result because the model's parameters were already tuned to that sample rate.

Sample rate	Average AUC
64	69.8
128	73.4
231	70.1

The hypothesis was correct. 64Hz scored worse than the 231-sample rate, likely because of how much data was lost in the downsampling.

9.4 Shorter sample time

Previously I had tested downsampling the data. In this experiment I wanted to test what happens if we narrow in the data closer to the time the image was shown to the subject and the time the P300 response would expectedly by triggered. The current data spans from -0.1 to 0.8s with 0.0s being the first moment the subject sees the image. In this experiment shifted the data such that only 0.0 to 0.5s are recorded. The 0.9s is quite long and the thought is there could be outside noise related to other images which make the signals harder to learn. The hypothesis is the 0.9s will be more successful, but this is just an idea which I'd like to prove wrong.

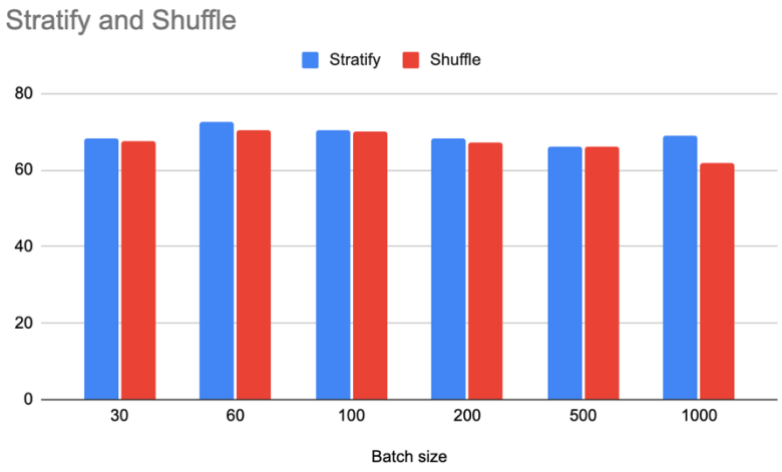
Sample time	Average AUC
-0.1s => 0.8s	72.8
0.0s => 0.5s	56.1

The hypothesis in this case was correct, and I will keep using the data as it was provided to me with the entire -0.1 to 0.8s span of data.

9.5 Stratify

I had already tried one technique to make the target data more interspersed amongst the dataset, and it had worked for one subject but not for cross-subject training. Another strategy wished to try was a strategic stratification of the data such that targets are interspersed evenly throughout the dataset, as they were in the actual experiments (roughly one per every twenty images).

Batch size	Stratify	Shuffle
30	68.2	67.5
60	72.4	70.5
100	70.4	69.9
200	68.3	67.2
500	66.2	66.3
1000	69.0	61.8



In fact, stratifying and shuffling the data scored nearly exactly the same results.

9.6 Normalisation

Up until this point I had performed no normalisation on the data. On this kind of data (EEG), where a bit of noise and messiness is to be expected, normalisation could play a big role in helping

performance of the classifier. I want to try a Z-score normalisation (or standardisation), an L1 normalisation as used in the NER2015 paper, and compare that to no normalisation, all for the single subject trials. I hypothesise that the L1 normaliser will get the best result; it is resilient to outliers which can make it an especially good normaliser for making sense of noisy data.

Normalising	**Result**
Z-score	73.2
L1 normaliser	77.8
None	70.1

This was the greatest result to date. Clearly the normalisation was helping. L1 normalisation proved most successful, but even the standardisation helped enormously.

9.7 Number of filters

The numbers of filters were already optimised by the EEGNet authors for the data with sample rate of 128, but I want to still try my own experimentations. I will compare the researchers' recommendation of eight temporal filters and two spatials with sixteen temporal filters and just one spatial. I hypothesise that the original 8,2 configuration will prove more accurate.

Model-type	**Result**
8,2	70.3
16,1	70.8

In fact, my hypothesis was proven wrong. The 16,1 model proved ever so slightly more accurate over the four repetitions.

10. RESULTS

The final performance of the EEGNet on the RSVP data successfully matched the performance of the EEGNet authors and beat the performance of the winning paper from NER2015. The final performance of the model in cross-subject classification was an AUC score of 86.2, and with a classification threshold of 0.5 an accuracy of 86.7%. The ROC for this AUC is depicted in Fig. 32.

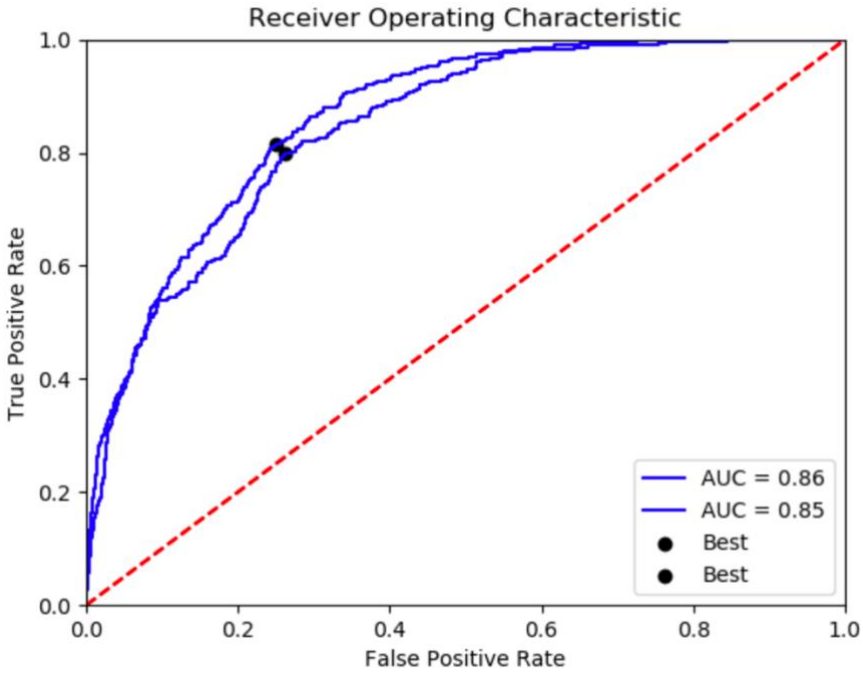


Fig. 32 - ROC from final cross-subject classification

For classification on just one subject the results were similar, with a repeatable AUC of 89.6 and an accuracy at the 0.5 threshold of 86.3%.

Results from all experiments found be found in Appendixes C & D, and all results are replicable using the GitHub repository (https://github.com/WillSmithTE/arl-eegmodels).

11. CONCLUSION

The GitHub repository ended up with over 550 commits. Some were minor fixes after finding out the code I'd written was wrong, but most were the setup for various experiments. Many such experiments and many commits could have been saved with more reading and research before the coding began. The most important lesson and takeaway for me is to really understand the subject matter and the data at hand before digging into the specifics. Having a solid understanding of the theory leaves for less guesswork and fiddling around. Apart from that key takeaway, I learnt so much throughout this project that it's hard to put into words.

The process felt slow-going at the time, but by the end I hope it is clear for the reader the kind of success that EEGNet can have in working with EEG signals. The development of such a successful model which is capable at learning neural signals seemingly regardless of the specific paradigm has clear real-world implications, with the potential to help and improve lives. With the way things are trending, the integration of EEG and AI is just getting started, and many of the most brilliant applications of such technologies are most likely still to be developed. It certainly as a field with an open and exciting future.

12. REFERENCES

Barachant, A. & Bonnet, S. 2011 "Channel Selection Procedure Using Riemannian Distance for BCI Applications." 2011 5th International IEEE/EMBS Conference on Neural Engineering, 2011, doi:10.1109/ner.2011.5910558.

Barachant, A., et al. 2012, "Classification of Covariance Matrices Using a Riemannian-Based Kernel for BCI Applications." *Neurocomputing*, vol. 112, 2013, pp. 172–78. *Crossref*, doi:10.1016/j.neucom.2012.12.039.

Barachant, A., Rafal Cycon, et al. 2015 "Signal Processing & Classification Pipeline." BCI Challenge @ NER 2015, 2015. GitHub, github.com/alexandrebarachant/bci-challenge-ner-2015.

Broad, W. 2020, "A.I. Versus the Coronavirus." The New York Times, 26 Mar. 2020, www.nytimes.com/2020/03/26/science/ai-versus-the-coronavirus.html?

Chernecky, Cynthia C., & Berger, B. 2013. Laboratory Tests and Diagnostic Procedures. United States, Elsevier/Saunders, 2013.

Ciresan, D. et al. 2011, Cornell University, arxiv.org/pdf/1102.0183.pdf.

Duan, Xu, et al. "Quadcopter Flight Control Using a Non-Invasive Multi-Modal Brain Computer Interface." *Frontiers in Neurorobotics*, vol. 13, 2019. *Crossref*, doi:10.3389/fnbot.2019.00023.

Handelman, G., et al. 2012, "Peering Into the Black Box of Artificial Intelligence: Evaluation Metrics of Machine Learning Methods." *American Journal of Roentgenology*, vol. 212, no. 1, 2019, pp. 38–43. *Crossref*, doi:10.2214/ajr.18.20224.

Jensen, M. et al. 2015, "Brain Oscillations, Hypnosis, and Hypnotizability." *American Journal of Clinical Hypnosis*, vol. 57, no. 3, 2015, pp. 230–53. *Crossref*, doi:10.1080/00029157.2014.976786.

Lloyd, J. 1995, "Surviving the AI Winter," in Logic Programming: The 1995 International Symposium , MIT Press, 1995, pp.33-47

Lawhern, V. et al. 2018, "EEGNet: A Compact Convolutional Neural Network for EEG-Based Brain–Computer Interfaces." *Journal of Neural Engineering*, vol. 15, no. 5, 2018, p. 056013. *Crossref*, doi:10.1088/1741-2552/aace8c.

Lin, C. et al. 2015, "Extracting patterns of single-trial EEG using an adaptive learning algorithm." *Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Annual International Conference* vol. 2015 (2015): 6642-5. doi:10.1109/EMBC.2015.7319916

Lobo, J. et al. 2007, "AUC: A Misleading Measure of the Performance of Predictive Distribution Models." *Global Ecology and Biogeography*, vol. 17, no. 2, 2007, pp. 145–51. *Crossref*, doi:10.1111/j.1466-8238.2007.00358.x.

Park, S., et al. 2020, "Development of an Online Home Appliance Control System Using Augmented Reality and an SSVEP-Based Brain–Computer Interface." *IEEE Access*, vol. 7, 2020, pp. 163604–14. *Crossref*, doi:10.1109/access.2019.2952613.

Pisarchik, Alexander N., et al. 2019, "From Novel Technology to Novel Applications: Comment on 'An Integrated Brain-Machine Interface Platform With Thousands of Channels' by Elon Musk and Neuralink." *Journal of Medical Internet Research*, vol. 21, no. 10, 2019, p. e16356. *Crossref*, doi:10.2196/16356.

Rivet, B. et al. 2009, "xDAWN algorithm to enhance evoked potentials: application to brain-computer interface." *IEEE transactions on bio-medical engineering* vol. 56,8 (2009): 2035-43. doi:10.1109/TBME.2009.2012869

Srinivasam, R. & Nunez, P. 2012, "Electroencephalography." *Encyclopedia of Human Behavior (Second Edition)*, 2012, pp. 15–23. *ScienceDirect*, www.sciencedirect.com/science/article/pii/B9780123750006003955.

Torang, Arezo, et al. 2019, "An Elastic-Net Logistic Regression Approach to Generate Classifiers and Gene Signatures for Types of Immune Cells and T Helper Cell Subsets." *BMC Bioinformatics*, vol. 20, no. 1, 2019. *Crossref*, doi:10.1186/s12859-019-2994-z.

Vallabhaneni, A. & Wang T., 2005, "Brain—Computer Interface." *ResearchGate*, 2005, pp. 85–121. *ResearchGate*, doi:10.1007/0-306-48610-5_3.

Widrow, B., & Hoff M. 1960, "ADAPTIVE SWITCHING CIRCUITS." 1960, doi:10.21236/ad0241531.

Niedermeyer, E, & Da, SFL 2004, Electroencephalography : Basic Principles, Clinical Applications, and Related Fields, Wolters Kluwer, Philadelphia

12. APPENDICES

Appendix A: Project Communication Log

Project Title:	Applying Modern Techniques in Artificial Intelligence to Neural Activity		
Student Name:	William Smith	**Supervisor Name:**	Dr YK Wang
Date	**Event**	**Topic of Communication**	**Outcome**
15/05/2020	Online message	Defining the project	Clearer idea of what will go into the project and what the basic subject matter is
11/09/2020	Online message	Project timeline	Confirmed to finish the project over summer
15/09/2020	Audio call	Online repository I was to use (BCI EEGNet) and making sure I had the right idea of the general project	Repository to work off and clear goals to work towards (getting the model working from that repository)
06/10/2020	Online message	Discussing me being stuck with the current result using the BCI EEGNet repository	Just a status update
17/10/2020	Online message	Update on a new online repository to work off with an update of progress on classifying online data	YK advised me to continue focusing on the theory as well
19/10/2020	Online message	Getting in touch with YK's PHD student (Sai) who will assist and provide the data	Sai to provide the RSVP dataset

23/10/2020	Online message	Checking in on the status of getting the RSVP dataset	YK passed the message onto his assistant Sai who provided me with the data
11/03/2020	Online message	Status update and request for assurance with imbalanced data	Supervisor advised me that imbalanced data is normal and provided a good example
18/12/2020	Online message	Overall guidance with bigger picture, looking at the project scope	Reaffirmation of clear outcomes and goals of the project
21/12/2020	Online message	YK and Sai request a quick update on the project	I agree to prepare some quick basic slides sharing progress
15/01/2021	Audio call	Deliver 10 slides and short presentation/discussion on the status of the project and results so far	Clear 8-point list of things to try and ways to both improve the model and improve my understanding of EEG and AI
08/02/2021	Email	Sai provides me with a report by YK describing the RSVP data	I have more helpful information on the data which helps with training the model and writing my report

Appendix B: ARL Model Library Dependencies

Library	Version
_libgcc_mutex	0.1
_openmp_mutex	4.5
_tflow_select	2.1.0
absl-py	0.10.0
astor	0.8.1
backcall	0.2.0
blas	2.17
bleach	1.5.0
c-ares	1.16.1
ca-certificates	2020.12.5
certifi	2020.12.5
cudatoolkit	9
cudnn	7.6.5
cupti	9.0.176
cycler	0.10.0
dbus	1.13.6
decorator	4.4.2
enum34	1.1.10
expat	2.2.9
fontconfig	2.13.1
freetype	2.10.3
gast	0.2.2
gettext	0.19.8.1
glib	2.66.1
google-pasta	0.2.0
grpcio	1.31.0
gst-plugins-base	1.14.5
gstreamer	1.14.5
h5py	2.10.0
hdf5	1.10.6
html5lib	0.9999999
icu	58.2
importlib-metadata	2.0.0
ipykernel	5.4.3
ipython	7.16.1
ipython-genutils	0.2.0
jedi	0.18.0
joblib	0.17.0
jpeg	9d
jupyter-client	6.1.11
jupyter-core	4.7.1
keras-applications	1.0.8
keras-preprocessing	1.1.0
kiwisolver	1.2.0
krb5	1.17.1
lcms2	2.11
ld_impl_linux-64	2.35
libblas	3.8.0
libcblas	3.8.0
libcurl	7.71.1
libedit	3.1.20191231
libev	4.33
libffi	3.2.1
libgcc-ng	9.3.0
libgfortran-ng	7.5.0
libgfortran4	7.5.0
libglib	2.66.1
libiconv	1.16
liblapack	3.8.0
liblapacke	3.8.0
libnghttp2	1.41.0
libopenblas	0.3.10
libpng	1.6.37
libprotobuf	3.13.0.1
libssh2	1.9.0
libstdcxx-ng	9.3.0
libtiff	4.1.0

libuuid	2.32.1
libwebp-base	1.1.0
libxcb	1.13
libxml2	2.9.9
llvm-openmp	11.0.0
lz4-c	1.9.2
markdown	3.3.1
matplotlib	2.2.3
matplotlib-base	3.3.2
metakernel	0.27.5
mne	0.21.0
ncurses	6.2
numpy	1.19.2
oct2py	5.2.0
octave-kernel	0.32.0
olefile	0.46
openssl	1.1.1i
opt-einsum	3.3.0
pandas	1.1.3
parso	0.8.1
patsy	0.5.1
pcre	8.44
pexpect	4.8.0
pickleshare	0.7.5
pillow	8.0.0
pip	20.2.3
prompt-toolkit	3.0.14
protobuf	3.13.0.1
pthread-stubs	0.4
ptyprocess	0.7.0
pygments	2.7.4
pyparsing	2.4.7
pyqt	5.9.2
pyriemann	0.2.5
python	3.6.11

python-dateutil	2.8.1
python_abi	3.6
pytz	2020.1
pyzmq	22.0.2
qt	5.9.7
readline	8
scikit-learn	0.20.1
scipy	1.5.2
seaborn	0.11.0
seaborn-base	0.11.0
setuptools	49.6.0
sip	4.19.8
six	1.15.0
statsmodels	0.12.0
tensorboard	1.12.2
tensorflow	1.12.0
tensorflow-base	1.12.0
tensorflow-estimator	1.15.1
tensorflow-gpu	1.12.0
tensorflow-tensorboard	0.4.0
termcolor	1.1.0
tk	8.6.10
tornado	6.0.4
traitlets	4.3.3
wcwidth	0.2.5
werkzeug	1.0.1
wheel	0.35.1
wrapt	1.12.1
xorg-libxau	1.0.9
xorg-libxdmcp	1.1.3
xz	5.2.5
zipp	3.3.1
zlib	1.2.11
zstd	1.4.5

Appendix C: Cross-subject Results

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
100	1000	128	64	0.25	0.01	0.6170	0.8270	8	2	
100	1000	128	64	0.25	0.01	0.5950	0.8710	8	2	
300	1000	128	64	0.25	0.01	0.6520	0.6450	8	2	
300	1000	128	64	0.25	0.01	0.5330	0.9270	8	2	
1000	1000	128	64	0.25	0.01	0.6260	0.8280	8	2	
1000	1000	128	64	0.25	0.01	0.5860	0.7780	8	2	
300	2000	128	64	0.25	0.01	0.5760	0.8300	8	2	
300	2000	128	64	0.25	0.01	0.6100	0.8080	8	2	
300	100	128	64	0.25	0.01	0.6080	0.7790	8	2	
300	100	128	64	0.25	0.01	0.6460	0.5340	8	2	
1000	100	128	64	0.25	0.01	0.6640	0.7370	8	2	
1000	100	128	64	0.25	0.01	0.5440	0.9020	8	2	
1000	100	231	64	0.25	0.01	0.5820	0.4270	8	2	got 75% of target
500	100	231	115	0.25	0.01	0.5420	0.8300	8	2	
500	100	231	115	0.25	0.01	0.5450	0.2970	8	2	
500	100	231	115	0.25	0.1	0.5030	0.9450	8	2	1% of target right
500	100	231	115	0.25	0.001	0.5940	0.7530	8	2	
500	100	231	115	0.25	0.01	0.5600	0.8150	8	2	
500	100	231	115	0.25	0.01	0.5970	0.5590	8	2	
500	100	128	115	0.25	0.01	0.6370	0.5890	8	2	
2000	100	128	115	0.25	0.005	0.6170	0.4790	8	2	
300	1000	128	64	0.25	0.01	0.6400	0.8240	8	2	filter 0.5-20 instea
300	1000	128	64	0.25	0.01	0.6710	0.7820	8	2	
600	1000	128	64	0.25	0.01	0.5620	0.8910	8	2	
600	1000	128	64	0.25	0.001	0.6330	0.7850	8	2	
600	1000	128	64	0.25	0.001	0.6410	0.7670	8	2	
300	1000	128	64	0.25	0.001	0.6450	0.7650	8	2	
300	1000	128	64	0.25	0.001	0.6640	0.7490	8	2	
300	1000	128	64	0.5	0.001	0.6610	0.6980	8	2	
300	1000	128	64	0.5	0.001	0.6970	0.7250	8	2	
300	1000	128	64	0.5	0.001	0.6860	0.7050	8	2	
600	1000	128	64	0.5	0.001	0.6540	0.7040	8	2	
600	1000	128	64	0.5	0.001	0.6820	0.6860	8	2	
300	1000	128	64	0.5	0.001	0.6653	0.6954	8	2	
300	1000	128	64	0.5	0.001	0.6895	0.6850	8	2	
300	500	128	64	0.5	0.001	0.6924	0.7093	8	2	
300	500	128	64	0.5	0.001	0.6784	0.6563	8	2	
300	2000	128	64	0.5	0.001	0.6736	0.6548	8	2	
300	2000	128	64	0.5	0.001	0.6904	0.7168	8	2	
300	3000	128	64	0.5	0.001	0.6809	0.6648	8	2	
300	3000	128	64	0.5	0.001	0.7064	0.6756	8	2	
300	4000	128	64	0.5	0.001	0.6897	0.6854	8	2	
300	4000	128	64	0.5	0.001	0.6946	0.7097	8	2	
300	1000	128	64	0.5	0.001	0.6614	0.7258	16	2	
300	1000	128	64	0.5	0.001	0.6761	0.7085	16	2	
300	500	128	64	0.5	0.001	0.6719	0.6854	16	2	
300	500	128	64	0.5	0.001	0.6546	0.7316	16	2	
300	2000	128	64	0.5	0.001	0.6437	0.7751	16	2	
300	2000	128	64	0.5	0.001	0.6648	0.7397	16	2	
300	3000	128	64	0.5	0.001	0.6859	0.7347	16	2	
300	3000	128	64	0.5	0.001	0.6843	0.7052	16	2	
300	4000	128	64	0.5	0.001	0.6867	0.7022	16	2	
300	4000	128	64	0.5	0.001	0.6921	0.7050	16	2	
300	10000	128	64	0.5	0.001	0.6919	0.5689	8	2	

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
300	10000	128	64	0.5	0.001	0.6536	0.4020	8	2	
300	10000	128	64	0.5	0.001	0.6605	0.4640	16	2	
300	10000	128	64	0.5	0.001	0.6716	0.5718	16	2	
300	1000	128	64	0.5	0.001	0.6824	0.6941	16	1	
300	1000	128	64	0.5	0.001	0.6807	0.7020	16	1	
300	500	128	64	0.5	0.001	0.6824	0.6979	16	1	
300	500	128	64	0.5	0.001	0.6819	0.6931	16	1	
300	2000	128	64	0.5	0.001	0.6748	0.7097	16	1	
300	2000	128	64	0.5	0.001	0.6743	0.7050	16	1	
300	3000	128	64	0.5	0.001	0.6864	0.7168	16	1	
300	3000	128	64	0.5	0.001	0.6825	0.6754	16	1	
300	4000	128	64	0.5	0.001	0.6958	0.7422	16	1	
300	4000	128	64	0.5	0.001	0.6980	0.6973	16	1	
300	10000	128	64	0.5	0.001	0.6764	0.5395	16	1	
300	10000	128	64	0.5	0.001	0.6739	0.5724	16	1	
300	3000	128	64	0.5	0.001	0.7049	0.6954	8	2	
300	3000	128	64	0.5	0.001	0.7030	0.6502	8	2	
300	3000	128	64	0.65	0.001	0.7139	0.6296	8	2	
300	3000	128	64	0.65	0.001	0.7136	0.6140	8	2	
300	3000	128	64	0.8	0.001	0.7073	0.4588	8	2	
300	3000	128	64	0.8	0.001	0.7085	0.4648	8	2	
300	3000	128	64	0.5	0.0005	0.6802	0.6409	8	2	
300	3000	128	64	0.5	0.0005	0.6957	0.6063	8	2	
300	3000	128	64	0.5	0.0005	0.6907	0.6683	8	2	
300	3000	128	64	0.5	0.0005	0.7062	0.6226	8	2	
300	3000	128	64	0.5	0.001	0.6681	0.7459	16	2	
300	3000	128	64	0.5	0.001	0.6851	0.6804	16	2	
300	3000	128	64	0.5	0.001	0.6004	0.7794	32	2	
300	3000	128	64	0.5	0.001	0.6463	0.7799	32	2	
300	3000	128	64	0.5	0.001	0.6332	0.7776	64	2	
300	3000	128	64	0.5	0.001	0.6251	0.7547	64	2	
300	3000	128	64	0.5	0.001	0.5818	0.7892	128	2	
300	3000	128	64	0.5	0.001	0.6101	0.7790	128	2	
300	3000	128	64	0.5	0.001	0.7017	0.6893	8	1	
300	3000	128	64	0.5	0.001	0.6905	0.6040	8	1	
300	3000	128	64	0.5	0.001	0.7026	0.5893	8	1	
300	3000	128	64	0.5	0.001	0.6929	0.6613	8	1	
300	4000	64	32	0.5	0.001	0.7227	0.6388	8	2	
300	4000	64	32	0.5	0.001	0.7018	0.6367	8	2	
300	4000	64	32	0.5	0.001	0.6934	0.5830	8	2	
300	4000	64	32	0.5	0.001	0.6967	0.6648	8	2	
300	4000	64	32	0.5	0.001	0.6846	0.6003	8	4	
300	4000	64	32	0.5	0.001	0.6924	0.6829	8	4	
300	4000	64	32	0.5	0.001	0.6445	0.7690	32	4	
300	4000	64	32	0.5	0.001	0.6151	0.7659	32	4	
300	4000	64	32	0.5	0.001	0.6647	0.6415	8	8	
300	4000	64	32	0.5	0.001	0.6816	0.7603	8	8	
300	4000	64	32	0.5	0.001	0.6838	0.7457	16	8	
300	4000	64	32	0.5	0.001	0.6554	0.7333	16	8	
300	4000	64	32	0.5	0.005	0.6380	0.7039	8	8	
300	4000	64	32	0.5	0.005	0.6353	0.7778	8	8	
300	4000	64	32	0.5	0.005	0.6170	0.7545	16	8	
300	4000	64	32	0.5	0.005	0.6285	0.8177	16	8	
600	4000	64	32	0.5	0.005	0.6280	0.7526	8	8	

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
600	4000	64	32	0.5	0.005	0.6250	0.7168	8	8	
600	4000	64	32	0.5	0.005	0.6129	0.7353	16	8	
600	4000	64	32	0.5	0.005	0.6174	0.7890	16	8	
600	4000	64	32	0.5	0.001	0.6685	0.7054	8	8	
600	4000	64	32	0.5	0.001	0.6726	0.7056	8	8	
600	4000	64	32	0.5	0.001	0.6243	0.7720	16	8	
600	4000	64	32	0.5	0.001	0.6465	0.7426	16	8	
1000	4000	64	32	0.5	0.001	0.6629	0.7362	8	8	
1000	4000	64	32	0.5	0.001	0.6699	0.7343	8	8	
1000	4000	64	32	0.5	0.001	0.6273	0.7551	16	8	
1000	4000	64	32	0.5	0.001	0.6434	0.7744	16	8	
300	500	64	32	0.65	0.001	0.6887	0.6419	16	1	
300	500	64	32	0.65	0.001	0.6776	0.5757	16	1	
100	500	64	32	0.65	0.001	0.7067	0.5593	16	1	
100	500	64	32	0.65	0.001	0.7200	0.5356	16	1	
300	500	64	32	0.7	0.001	0.6964	0.5662	16	1	
300	500	64	32	0.7	0.001	0.7241	0.5737	16	1	
100	500	64	32	0.7	0.001	0.7097	0.5726	16	1	
100	500	64	32	0.7	0.001	0.7233	0.5608	16	1	
300	500	64	32	0.6	0.001	0.6957	0.6213	16	1	
300	500	64	32	0.6	0.001	0.6852	0.5938	16	1	
100	500	64	32	0.6	0.001	0.6939	0.5162	16	1	
100	500	64	32	0.6	0.001	0.7091	0.6167	16	1	
300	5000	64	32	0.65	0.001	0.6988	0.4802	16	1	
300	500	64	32	0.6	0.001	0.6894	0.6471	16	1	
300	500	64	32	0.6	0.001	0.6874	0.6207	16	1	
300	500	64	32	0.6	0.001	0.6893	0.6394	16	2	
300	500	64	32	0.6	0.001	0.6935	0.6097	16	2	
300	500	64	32	0.6	0.005	0.6716	0.5907	16	1	
300	500	64	32	0.6	0.005	0.6911	0.6013	16	1	
300	500	64	32	0.6	0.005	0.6705	0.6941	16	2	
300	500	64	32	0.6	0.005	0.6716	0.6359	16	2	
300	300	64	32	0.6	0.001	0.7094	0.6473	16	1	
300	300	64	32	0.6	0.001	0.6923	0.6640	16	1	
300	300	64	32	0.6	0.001	0.6942	0.6600	16	2	
300	300	64	32	0.6	0.001	0.6825	0.6904	16	2	
300	300	64	32	0.6	0.005	0.6779	0.6440	16	1	
300	300	64	32	0.6	0.005	0.6846	0.6417	16	1	
300	300	64	32	0.6	0.005	0.6770	0.6084	16	2	
300	300	64	32	0.6	0.005	0.6687	0.6380	16	2	
300	300	64	32	0.6	0.0005	0.6977	0.6854	16	2	
300	300	64	32	0.6	0.0005	0.7039	0.6521	16	2	
300	300	64	32	0.6	0.0005	0.6898	0.6402	16	2	
300	300	64	32	0.6	0.0005	0.6935	0.6850	16	2	
500	300	64	32	0.6	0.0005	0.6777	0.6248	16	2	
500	300	64	32	0.6	0.0005	0.6644	0.6448	16	2	
500	300	64	32	0.6	0.0005	0.6966	0.6871	16	2	
500	300	64	32	0.6	0.0005	0.6832	0.6579	16	2	
500	300	64	32	0.6	0.0005	0.6667	0.7133	16	4	
500	300	64	32	0.6	0.0005	0.6966	0.6985	16	4	
500	300	64	32	0.6	0.0005	0.6880	0.6746	16	4	
500	300	64	32	0.6	0.0005	0.6692	0.7293	16	4	
500	300	64	32	0.6	0.0005	0.5660	0.5481	16	4	shuffle all
500	300	64	32	0.6	0.0005	0.5661	0.5512	16	4	

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
500	300	64	32	0.6	0.0005	0.5747	0.5449	16	4	
500	300	64	32	0.6	0.0005	0.5707	0.5262	16	4	
500	16	64	32	0.6	0.0005	0.5355	0.8414	16	4	
500	16	64	32	0.6	0.0005	0.5596	0.8111	16	4	
500	16	64	32	0.6	0.0005	0.5249	0.8529	16	4	
500	16	64	32	0.6	0.0005	0.5508	0.8165	16	4	
300	64	64	32	0.6	0.0005	0.5805	0.6192	16	1	
300	64	64	32	0.6	0.0005	0.5999	0.5780	16	1	
300	64	64	32	0.6	0.0005	0.6135	0.6465	16	1	
300	64	64	32	0.6	0.0005	0.6025	0.5685	16	1	
300	128	64	32	0.6	0.0005	0.5945	0.5075	16	1	
300	128	64	32	0.6	0.0005	0.5838	0.5248	16	1	
300	128	64	32	0.6	0.0005	0.5867	0.5360	16	1	
300	128	64	32	0.6	0.0005	0.5875	0.5547	16	1	
300	256	64	32	0.6	0.0005	0.5852	0.5275	16	1	
300	256	64	32	0.6	0.0005	0.6065	0.5329	16	1	
300	256	64	32	0.6	0.0005	0.5981	0.5314	16	1	
300	256	64	32	0.6	0.0005	0.6043	0.5345	16	1	
300	100	64	32	0.6	0.001	0.5971	0.5670	16	1	
300	100	64	32	0.6	0.001	0.6043	0.5633	16	1	
300	100	64	32	0.6	0.001	0.5923	0.5753	16	1	
300	100	64	32	0.6	0.001	0.5958	0.4985	16	1	
300	200	64	32	0.6	0.001	0.5942	0.5329	16	1	
300	200	64	32	0.6	0.001	0.6139	0.5410	16	1	
300	200	64	32	0.6	0.001	0.5873	0.5171	16	1	
300	200	64	32	0.6	0.001	0.5852	0.5160	16	1	
300	300	64	32	0.6	0.001	0.5844	0.4973	16	1	
300	300	64	32	0.6	0.001	0.5953	0.5493	16	1	
300	300	64	32	0.6	0.001	0.5736	0.5144	16	1	
300	300	64	32	0.6	0.001	0.6038	0.5680	16	1	
300	400	64	32	0.6	0.001	0.5951	0.5345	16	1	
300	400	64	32	0.6	0.001	0.5977	0.5164	16	1	
300	400	64	32	0.6	0.001	0.5924	0.5323	16	1	
300	400	64	32	0.6	0.001	0.6097	0.5963	16	1	
300	64	64	32	0.6	0.001	0.6765	0.7355	16	1	don't shuffle
300	64	64	32	0.6	0.001	0.6791	0.6991	16	1	
300	64	64	32	0.6	0.001	0.7016	0.6854	16	1	
300	64	64	32	0.6	0.001	0.7063	0.6866	16	1	
300	128	64	32	0.6	0.001	0.7058	0.6556	16	1	
300	128	64	32	0.6	0.001	0.6782	0.6635	16	1	
300	128	64	32	0.6	0.001	0.7020	0.6710	16	1	
300	128	64	32	0.6	0.001	0.6807	0.7097	16	1	
300	256	64	32	0.6	0.001	0.7027	0.5820	16	1	
300	256	64	32	0.6	0.001	0.7121	0.5847	16	1	
300	256	64	32	0.6	0.001	0.6940	0.6219	16	1	
300	256	64	32	0.6	0.001	0.7043	0.6076	16	1	
100	4000	64	32	0.6	0.001	0.6987	0.4840	16	1	
100	4000	64	32	0.6	0.001	0.6965	0.5287	16	1	
100	4000	64	32	0.6	0.001	0.7085	0.5214	16	1	
100	4000	64	32	0.6	0.001	0.6975	0.5495	16	1	
200	4000	64	32	0.6	0.001	0.7064	0.5362	16	1	
200	4000	64	32	0.6	0.001	0.7012	0.5452	16	1	
200	4000	64	32	0.6	0.001	0.7036	0.5535	16	1	
200	4000	64	32	0.6	0.001	0.7131	0.5641	16	1	

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
300	4000	64	32	0.6	0.001	0.6938	0.6140	16	1	
300	4000	64	32	0.6	0.001	0.6826	0.5776	16	1	
300	4000	64	32	0.6	0.001	0.7043	0.5737	16	1	
300	4000	64	32	0.6	0.001	0.7014	0.5643	16	1	
400	4000	64	32	0.6	0.001	0.6946	0.6382	16	1	
400	4000	64	32	0.6	0.001	0.6774	0.6733	16	1	
400	4000	64	32	0.6	0.001	0.6982	0.5433	16	1	
400	4000	64	32	0.6	0.001	0.6920	0.6067	16	1	
400	128	64	32	0.6	0.001	0.6822	0.6862	16	2	
400	128	64	32	0.6	0.001	0.6778	0.6854	16	2	
400	128	64	32	0.6	0.001	0.6486	0.7014	16	2	
400	128	64	32	0.6	0.001	0.6645	0.7129	16	2	
400	64	64	32	0.6	0.001	0.6666	0.7131	16	2	
400	64	64	32	0.6	0.001	0.6565	0.7466	16	2	
400	64	64	32	0.6	0.001	0.6654	0.6956	16	2	
400	64	64	32	0.6	0.001	0.6891	0.7370	16	2	
400	32	64	32	0.6	0.001	0.6105	0.8023	16	2	
400	32	64	32	0.6	0.001	0.6408	0.7996	16	2	
400	32	64	32	0.6	0.001	0.6126	0.8215	16	2	
400	32	64	32	0.6	0.001	0.6155	0.8005	16	2	
400	16	64	32	0.6	0.001	0.5290	0.9186	16	2	
400	16	64	32	0.6	0.001	0.5517	0.9166	16	2	
400	16	64	32	0.6	0.001	0.5238	0.9238	16	2	
400	16	64	32	0.6	0.001	0.5334	0.9195	16	2	
300	4000	64	32	0.5	0.001	0.6990	0.6841	16	1	
300	4000	64	32	0.5	0.001	0.6717	0.6663	16	1	
300	4000	64	32	0.5	0.001	0.7003	0.6527	16	1	
300	4000	64	32	0.5	0.001	0.6789	0.6573	16	1	
300	4000	64	32	0.5	0.001	0.6814	0.6506	16	1	
300	4000	64	32	0.5	0.001	0.6865	0.6567	16	1	
300	4000	64	32	0.5	0.001	0.6864	0.6375	16	1	
300	4000	64	32	0.5	0.001	0.6776	0.6284	16	1	
300	4000	128	64	0.5	0.001	0.6691	0.6989	16	1	
300	4000	128	64	0.5	0.001	0.6780	0.7083	16	1	
300	4000	128	64	0.5	0.001	0.6448	0.6300	16	1	
300	4000	128	64	0.5	0.001	0.7002	0.6939	16	1	
300	4000	128	64	0.5	0.001	0.6791	0.6991	16	1	
300	4000	128	64	0.5	0.001	0.6822	0.7389	16	1	
300	4000	128	64	0.5	0.001	0.7050	0.6881	16	1	
300	4000	128	64	0.5	0.001	0.6543	0.7122	16	1	
300	4000	231	115	0.5	0.001	0.6552	0.7366	16	1	
300	4000	231	115	0.5	0.001	0.6349	0.7582	16	1	
300	4000	231	115	0.5	0.001	0.6338	0.7110	16	1	
300	4000	231	115	0.5	0.001	0.6305	0.7160	16	1	
300	4000	231	115	0.5	0.001	0.6556	0.8013	16	1	
300	4000	231	115	0.5	0.001	0.6816	0.7603	16	1	
300	4000	231	115	0.5	0.001	0.6754	0.6921	16	1	
300	4000	231	115	0.5	0.001	0.6511	0.6384	16	1	
500	60	128	64	0.5	0.001	0.6206	0.5314	8	2	stratify data
500	60	128	64	0.5	0.001	0.6291	0.5181	8	2	stratify data
500	80	128	64	0.5	0.001	0.6234	0.5104	8	2	stratify data
500	80	128	64	0.5	0.001	0.6132	0.5352	8	2	stratify data
500	100	128	64	0.5	0.001	0.6257	0.5087	8	2	stratify data
500	100	128	64	0.5	0.001	0.6353	0.5912	8	2	stratify data

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
300	4000	128	64	0.5	0.001	0.5912	0.5612	8	2	stratify data
300	4000	128	64	0.5	0.001	0.5988	0.5462	8	2	stratify data
300	4000	128	64	0.5	0.001	0.5731	0.6064	8	2	stratify data
300	4000	128	64	0.5	0.001	0.5911	0.5171	8	2	stratify data
300	4000	128	64	0.5	0.001	0.5929	0.5878	16	1	stratify data
300	4000	128	64	0.5	0.001	0.5915	0.4682	16	1	stratify data
300	4000	128	64	0.5	0.001	0.5920	0.5862	16	1	stratify data
300	4000	128	64	0.5	0.001	0.5930	0.5558	16	1	stratify data
300	4000	128	64	0.5	0.001	0.6929	0.6914	8	2	
300	4000	128	64	0.5	0.001	0.6941	0.6937	8	2	
300	4000	128	64	0.5	0.001	0.6964	0.6717	8	2	
300	4000	128	64	0.5	0.001	0.6911	0.6918	8	2	
300	4000	128	64	0.5	0.001	0.6918	0.7270	16	1	
300	4000	128	64	0.5	0.001	0.6878	0.7270	16	1	
300	4000	128	64	0.5	0.001	0.6884	0.7054	16	1	
300	4000	128	64	0.5	0.001	0.6969	0.7104	16	1	
300	60	128	64	0.5	0.001	0.6446	0.7615	8	2	
300	60	128	64	0.5	0.001	0.6965	0.7397	8	2	
300	500	128	64	0.5	0.001	0.7258	0.6916	8	2	z-score all
300	500	128	64	0.5	0.001	0.7390	0.6887	8	2	z-score all
300	500	128	64	0.5	0.001	0.7117	0.6773	8	2	z-score all
300	500	128	64	0.5	0.001	0.7201	0.7297	8	2	z-score all
300	500	128	64	0.5	0.001	0.7448	0.6457	8	2	l1-norm all
300	500	128	64	0.5	0.001	0.7335	0.6602	8	2	l1-norm all
300	500	128	64	0.5	0.001	0.7442	0.5435	8	2	l1-norm all
300	500	128	64	0.5	0.001	0.7317	0.7093	8	2	l1-norm all
300	500	128	64	0.5	0.001	0.7367	0.6796	8	2	
300	500	128	64	0.5	0.001	0.7231	0.6970	8	2	
300	500	128	64	0.5	0.001	0.7288	0.7399	8	2	
300	500	128	64	0.5	0.001	0.7344	0.7089	8	2	
300	80	231	115	0.5	0.001	0.8425	0.7253	8	2	
300	80	231	115	0.5	0.001	0.8220	0.7268	8	2	
300	160	231	115	0.5	0.001	0.8351	0.6490	8	2	
300	160	231	115	0.5	0.001	0.8006	0.7168	8	2	
300	320	231	115	0.5	0.001	0.8131	0.6831	8	2	
300	320	231	115	0.5	0.001	0.8155	0.6925	8	2	
300	640	231	115	0.5	0.001	0.8084	0.6841	8	2	
300	640	231	115	0.5	0.001	0.7783	0.6176	8	2	
300	1280	231	115	0.5	0.001	0.8331	0.6692	8	2	
300	1280	231	115	0.5	0.001	0.8372	0.7428	8	2	
300	80	128	64	0.5	0.001	0.6172	0.6082	8	2	
300	80	128	64	0.5	0.001	0.6209	0.6061	8	2	
300	80	231	115	0.5	0.001	0.8337	0.7686	8	2	
300	80	231	115	0.5	0.001	0.8202	0.7505	8	2	
300	80	128	64	0.5	0.001	0.7377	0.7029	8	2	
300	80	128	64	0.5	0.001	0.8059	0.7289	8	2	
300	80	231	115	0.5	0.001	0.6427	0.5766	8	2	
300	16	128	64	0.5	0.001	0.7597	0.7815	8	2	
300	16	128	64	0.5	0.001	0.7852	0.6998	8	2	
300	40	128	64	0.5	0.001	0.7498	0.6975	8	2	
300	40	128	64	0.5	0.001	0.7605	0.8000	8	2	
300	4000	128	64	0.5	0.001	0.7605	0.6987	8	2	
300	4000	128	64	0.5	0.001	0.7602	0.7249	8	2	
300	80	231	115	0.5	0.001	0.7987	0.9361	8	2	l1 norm

Epochs	Batch size	Sample ra	Kern length	Dropout	Learning	roc_auc	accuracy	F1	D	notes
300	80	231	115	0.5	0.001	0.8090	0.4444	8	2	l1 norm
300	80	231	115	0.5	0.001	0.8265	0.7333	8	2	z-norm
300	80	231	115	0.5	0.001	0.7985	0.7453	8	2	z-norm
300	80	231	115	0.5	0.001	0.8623	0.8670	8	2	stratify no norm, ' [[3391 1114],[70
300	80	231	115	0.5	0.001	0.8450	0.7535	8	2	stratify no norm,E

Appendix D: Subject-wise Results

epochs	batchSize	sampleRate	kernLength	dropout	learning	roc_auc	accuracy	f1	D	
300	1000	128	64	0.5	0.01	0.3320	0.5250	8	2	
300	1000	128	64	0.5	0.01	0.3700	0.6280	8	2	
300	100	128	64	0.5	0.01	0.3800	0.5500	8	2	
300	100	128	64	0.5	0.01	0.3800	0.6300	8	2	
300	50	128	64	0.5	0.01	0.3540	0.5660	8	2	
300	50	128	64	0.25	0.01	0.3670	0.6670	8	2	
300	60	128	64	0.5	0.001	0.6399	0.6628	8	2	shuffle
300	60	128	64	0.5	0.001	0.6570	0.7205	8	2	
300	30	128	64	0.5	0.001	0.6035	0.7244	4	2	
300	30	128	64	0.5	0.001	0.5592	0.8072	4	2	
300	60	128	64	0.5	0.001	0.6212	0.6956	4	2	
300	60	128	64	0.5	0.001	0.5921	0.6308	4	2	
300	120	128	64	0.5	0.001	0.6177	0.6269	4	2	
300	120	128	64	0.5	0.001	0.6078	0.5980	4	2	
300	30	128	64	0.5	0.001	0.6137	0.6971	8	2	
300	30	128	64	0.5	0.001	0.6056	0.7713	8	2	
300	16	128	64	0.5	0.001	0.5724	0.8486	8	2	
300	16	128	64	0.5	0.001	0.5821	0.7970	8	2	
500	30	128	64	0.5	0.001	0.6402	0.7861	8	2	
500	30	128	64	0.5	0.001	0.7068	0.7713	8	2	[[39 38],[260 943]]
300	30	128	64	0.3	0.001	0.6585	0.6714	8	2	
300	30	128	64	0.3	0.001	0.6493	0.7237	8	2	
800	30	128	64	0.5	0.001	0.6919	0.7377	8	2	
800	30	128	64	0.5	0.001	0.5891	0.8267	8	2	
800	30	128	64	0.6	0.001	0.6616	0.7041	8	2	
800	30	128	64	0.6	0.001	0.6454	0.6854	8	2	
800	30	128	64	0.7	0.001	0.6053	0.7158	8	2	
800	30	128	64	0.7	0.001	0.6582	0.6979	8	2	
500	30	125	62	0.5	0.001	0.5033	0.7088	8	2	use just 0-0.5s
500	30	125	62	0.5	0.001	0.5166	0.7564	8	2	use just 0-0.5s
500	30	128	64	0.5	0.001	0.6158	0.7978	8	2	
500	30	128	64	0.5	0.001	0.5754	0.7869	8	2	
500	30	128	64	0.5	0.001	0.5793	0.6815	8	2	stratify data
500	30	128	64	0.5	0.001	0.6472	0.7650	8	2	stratify data
500	60	128	64	0.5	0.001	0.7131	0.6807	8	2	stratify data
500	60	128	64	0.5	0.001	0.6309	0.7564	8	2	stratify data
500	100	128	64	0.5	0.001	0.6622	0.6620	8	2	stratify data
500	100	128	64	0.5	0.001	0.7156	0.6635	8	2	stratify data
500	200	128	64	0.5	0.001	0.6289	0.6760	8	2	stratify data
500	200	128	64	0.5	0.001	0.6451	0.6737	8	2	stratify data
500	500	128	64	0.5	0.001	0.7006	0.6464	8	2	stratify data
500	500	128	64	0.5	0.001	0.5931	0.5761	8	2	stratify data
500	1000	128	64	0.5	0.001	0.6543	0.6690	8	2	stratify data
500	1000	128	64	0.5	0.001	0.6556	0.6495	8	2	stratify data
500	60	128	64	0.5	0.001	0.7139	0.6503	8	2	
500	60	128	64	0.5	0.001	0.6692	0.6760	8	2	
500	100	128	64	0.5	0.001	0.6856	0.6518	8	2	
500	100	128	64	0.5	0.001	0.6674	0.6589	8	2	
500	200	128	64	0.5	0.001	0.6770	0.5941	8	2	
500	200	128	64	0.5	0.001	0.6531	0.5902	8	2	
500	500	128	64	0.5	0.001	0.6201	0.6105	8	2	
500	500	128	64	0.5	0.001	0.6887	0.5886	8	2	
500	1000	128	64	0.5	0.001	0.6731	0.5730	8	2	
500	1000	128	64	0.5	0.001	0.6211	0.5847	8	2	
500	100	128	64	0.5	0.001	0.6255	0.7026	8	2	
500	100	128	64	0.5	0.001	0.6360	0.6674	8	2	
500	100	128	64	0.5	0.001	0.6697	0.7198	8	2	
500	100	128	64	0.5	0.001	0.6489	0.6151	8	2	
500	100	128	64	0.5	0.001	0.6118	0.7096	16	1	
500	100	128	64	0.5	0.001	0.6689	0.6526	16	1	
500	100	128	64	0.5	0.001	0.6060	0.6659	16	1	
500	100	128	64	0.5	0.001	0.6863	0.7510	16	1	
500	100	128	64	0.5	0.001	0.6888	0.7424	8	2	z-score all
500	100	128	64	0.5	0.001	0.7021	0.7205	8	2	z-score all
500	100	128	64	0.5	0.001	0.7046	0.6963	8	2	z-score all

epochs	batchSize	sampleRate	kernLength	dropout	learning	roc_auc	accuracy	f1	D	
500	100	128	64	0.5	0.001	0.7051	0.6745	8	2	z-score all
500	100	128	64	0.5	0.001	0.6782	0.7377	8	2	l1-norm all
500	100	128	64	0.5	0.001	0.6894	0.7783	8	2	l1-norm all
500	100	128	64	0.5	0.001	0.6823	0.8025	8	2	l1-norm all
500	100	128	64	0.5	0.001	0.7028	0.8017	8	2	l1-norm all
500	100	128	64	0.5	0.001	0.6570	0.6729	8	2	
500	100	128	64	0.5	0.001	0.6300	0.6378	8	2	
500	100	128	64	0.5	0.001	0.6120	0.7057	8	2	
500	100	128	64	0.5	0.001	0.6705	0.7065	8	2	
300	80	231	115	0.5	0.001	0.8960	0.8806	8	2	Best Threshold=0.309149, stratify
300	80	231	115	0.5	0.001	0.8961	0.8478	8	2	Best Threshold=0.446272, stratify