Classification of 12-Lead ECGs using Residual Neural Networks

A dissertation submitted to The University of Manchester for the degree of

Master of Science in Artificial Intelligence

in the Faculty of Science and Engineering

Year of submission

2021

Darsh Jadhav

Student ID

10806664

School of Computer Science

Contents

Word count: 14,143

List of figures

Abstract

Diagnosing cardiac arrhythmias in patients as quickly and accurately as possible is important when time is of the essence. Doctors go through several years of medical training in order to be able to make these diagnoses with great accuracy. Upon investigating surrounding literature, there is empirical evidence that the use of deep neural networks for ECG classification can speed up the process of making an initial diagnosis whilst being incredibly accurate, thus making this an important research area.

The aim of this project is to implement a residual neural network (ResNet) to classify cardiac arrhythmias by using patient data provided by the PhysioNet/-Computing in Cardiology Challenge 2020. Furthermore, this project will examine the performances of different ResNet architectures to improve the results of this project. ResNets have been used in many computer vision and signal classification tasks as they seem to perform well on these dataset. This is due to the skip connections in the architecture that allow for better flow of information throughout the network. In order to use deep neural networks on ECG classification, it is important that appropriate data preprocessing steps are researched and conducted as it will allow for the deep neural network to work effectively. The implementation of the ResNet34 model in this project has produced a weighted f-measure of 0.477 on the ECG training set, and a weighted f-measure of 0.419 on the ECG testing set. These results are similar to the performances of some other models produced by other researchers in this research space.

Declaration of originality

I hereby confirm that this dissertation is my own original work unless referenced clearly to the contrary, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Intellectual property statement

Acknowledgements

I would like to thank Dr. Giles Reger and Dr. David Wong for their continuous guidance and support throughout this project.

1 Introduction

1.1 Project Description and Motivations

An electrocardiogram (ECG) is a recording of the hearts electrical activity, using electrodes, a signal can be produced to show the rhythm of the heart pumping. If a medical practitioner suspects that a patient may be suffering from a cardiac arrhythmia, then they would commonly use ECGs in order to identify if a patient has a problem with their heart. Whilst medical practitioners are highly trained to spot these kinds of abnormalities, they may oversee an indication of a cardiac arrhythmia, or potentially misdiagnose the patient. A study by Cook et al. was conducted to analyse the accuracy of a medical physician's ability to interpret an ECG. They found deficiencies in the accuracy of an ECG interpretation by medical practitioners (68.5% and 74.9% for practicing physicians and cardiologists respectively) [1]. A study by Santos et al. investigated the accuracy of a general practitioner's (GP) interpretation of an ECG, finding that the GP accuracy for normal tests was 79.9%, and for more common abnormalities reaching upwards of 90% [2]. Whilst Santos et al. have managed to find that general practitioners have been able to find abnormalities with great success, there is the possibility for improvement and consistency. The introduction of machine learning for ECG interpretation (classification) has significantly improve the accuracy of diagnoses made, and has been proven to make very strong classifications [3]–[5]. Utilising labelled ECG data, we can propose a model that can be used for ECG classification. Furthermore, extensive testing will be conducted in order to maximise the predictive power of the model. If the model can make very accurate predictions, it can be used to make an initial classification that can be passed on to a medical practitioner to make a final interpretation of the ECG. Not only will this improve the accuracy of current ECG interpretations, it will save the time that each medical practitioner spends on diagnosing an ECG, which makes the health-care system more efficient.

Automated ECG classification has been attempted in the past using both traditional, and deep learning models. Traditional machine learning models are developed to use hand crafted features from the ECG. These traditional machine learning models require a feature extraction phase, requiring the researcher to investigate the features that will help classify the data. Deep learning methods such as neural networks can act as feature extractors, where the initial layers of the neural network act as the feature selection task. This project will be utilising deep learning methods for ECG classification, which allows the neural networks to find the most optimal set of features to extract information from, thus not settling with a feature set that may not be the most optimal set of features. Whilst machine learning models are able to make highly accurate predictions in ECG classification, a key issue that they encounter is the class imbalance problem. The class imbalance problem is when the data in each class is not equally distributed, meaning that some classes are underrepresented in the dataset. The issue with this is that the machine learning model does not have enough data to extract information from these underrepresented classes (the rare cases in the population), which results in a poor classification for that class. In the medical industry, it is important that the rare cases are accurately identified, as incorrectly diagnosing a patient may have fatal consequences. Researchers have developed data augmentation solutions to deal with the class imbalance problem, in order to boost the representation of an underrepresented class.

In order to make ECG classifications, this project will need to consider the possible deep learning method that would be appropriate to use. In addition to this, we will research surrounding literature for possible data augmentation techniques to reduce the class imbalance problem.

1.2 Aims and Objectives

In this project, we will be using deep learning methods for the classification of 12-lead ECGs. We will gather appropriate data for this project, implement data preprocessing that is needed for ECG classification, explore machine learning models that can be used for ECG classification, and discuss the possibilities of data augmentation for ECGs. In order to complete this project, we need to set a list of objectives that are achievable.

1. Conduct in-depth research to develop an appropriate research methodology for this project. Doing so will enable the ability to start designing an appropriate system that can be used for 12-lead ECG classification.

2. Develop an experimental system design that considers all aspects of the project, considering which data preprocessing techniques will be used, and the type of deep learning model that can be used for ECG classification. Doing so will give the project a good structure, allowing for future planning of technical details of the project.

3. Apply data preprocessing in order to prepare the raw sourced data, which will make it suitable for the classification task.

4. Implement a deep neural network that is able to classify the preprocessed 12-lead ECG data.

5. Visualise and discuss the results that the neural network has produced. This includes an evaluation of the developed deep neural network by using evaluation metrics that are suitable for this project.

6. Reflect on the whole project by considering future work that can be done to improve the findings of this project.

1.3 Dissertation Structure

In this dissertation, we will be covering the journey of this project. The following chapters of this dissertation is divided into 4 chapters: the background research and literature review; the research methodology, system design, and implementation; the evaluation and results; and the summary (conclusion). Chapter 2 compromises of the background research and literature review sections. The background research covers background information that would explain the technical and theoretical aspects of the project. This chapter also conducts a literature review of the surrounding work in this domain. Chapter 3 discusses the methodology of the project, which will present an overview and analysis of the methods required to complete a machine learning project, the system design, and the intricate technical details of the implementation of this project. Chapter 4 is an evaluation of the project, this compromises of the results produced from the classification tasks, and an analysis of the technical implementations of this project. This chapter will consider any improvements that can be implemented in order to improve the results of this project. Chapter 5 is the final chapter of this project, which summarises the entire project, the conclusions that are made, and any future work that further develops this project.

2 Background and Literature Review

This project requires a certain amount of prior knowledge in order to understand the following work. Therefore, it is expected that the reader has prior computer science and machine learning knowledge, specifically knowledge that would be taught to an undergraduate computer scientist. We will explore the background fundamentals that are required to understand this project, the technical background for the methods implemented in this project, and a literature review that covers related research within the ECG classification field.

2.1 Key Definitions

Some common phrases will be used throughout this project, where the definitions are to be assumed. These will be listed below, and will be further expanded on in future sections.

1. **Electrocardiogram (ECG):** An ECG is a signal recording of the hearts electrical activity [6].
2. **ECG Leads:** A lead is referred to as the graphical representation of the hearts electrical activity [6].
3. **Cardiac Arrhythmia:** Cardiac arrhythmia (or arrhythmia) is an irregularity in the hearts rhythm. An irregularity is diagnosed by conducting an

ECG.

4. **Machine Learning (ML):** Machine learning is the multi-disciplinary field that combines many different concepts from various different fields for the purpose of implementing algorithms to make predictions using training data.

5. **Artificial Neural Networks (ANNs):** ANNs (also referred as neural networks) is a type of machine learning model that consists of a collection of artificial neurones that loosely mimic the structure of the brain.

6. **Deep Learning (DL):** Deep learning is a sub-domain of machine learning that specifically uses neural networks (with multiple hidden layers).

7. **Object:** A row in the dataset, i.e, a patient's ECG sample.

8. **Features:** Features in machine learning are the independent variables that influence the outcome of the dependent variable (also known as the column in a dataset). In this project, the initial feature set would be Age, Sex, Gender, and 12-lead ECGs

9. **Classes:** Classes in machine learning is the dependent variable in a dataset. This is the column(s) that we are trying to predict. For this project, this would be the diagnosis (arrhythmia) of the patient. The terms diagnosis, arrhythmia, and classes may be used interchangeably depending on the context.

10. **Multi-class Classification:** Multi-class classification is a classification task where the predicted class (dependent variable) consists of more than 2 different class labels. In other words, the classification task is not binary.

11. **Multi-label Classification:** Multi-label classification is a classification task where multiple classes can belong to a single object in the dataset. In other words, the class labels are not mutually exclusive. This differs from multi-class classification as we consider the classes in multi-class to be mutually exclusive.

12. **Learning rate:** Learning rate is a hyperparameter that manages the changes of the model based on the response of the loss function.

13. **Momentum:** Momentum is a parameter that defines the aggregation of the exponential decay of the average gradients in a gradient descent algorithm. This allows for the gradient descent algorithm to converge at a faster rate.

14. **Epoch:** An epoch is referred as the number of times that a full training set of data has been passed through a machine learning model.

2.2 Electrocardiogram (ECG)

An electrocardiogram (ECG) is a recording of the electrical impulses that pump the heart [6]. Placing electrodes on the chest and the limbs allows us to monitor the electrical impulses that go through the heart, giving us an idea of how well the heart is functioning [6]. The position in which these electrodes are placed depends on the type of ECG that is being conducted, for example, a 12-lead ECG has a different electrode configuration to a 5-lead ECG. A ‘lead’ is referred to the graphical representation/view of the heart’s electrical activity [6]. A standard ECG typically consists of 12 leads, which produce 12 leads by using 12 electrodes. Six of the electrodes are attached to the limbs (these electrodes correspond to leads I, II, III, aVL, aVR, and aVF), and the remaining six electrodes are attached to different parts of the chest (these electrodes correspond to leads V1, V2, V3, V4, V5, and V6). The 12-lead ECG is the standard in relation to the type of ECG used to perform a complete evaluation of the heart's activity, allowing medical professionals to detect an arrhythmia or a disturbance in the electrical conduction throughout the heart [7].

To interpret each ECG lead, we need to understand the ECG complex. An ECG complex is the technical term for one ECG cycle, or one heartbeat. An ECG complex is typically a combination of the P wave, QRS complex, T waves, and U waves. The R peak of the QRS complex is the peak where the amplitude is the highest in the ECG complex. The ECG complex also contains different intervals that show the time taken for electrical discharge that spreads throughout the heart [6]. These intervals are measured by using different parts of the complex. Figure 1 shows a visualisation of the ECG complex and the intervals that can be derived from the complex.

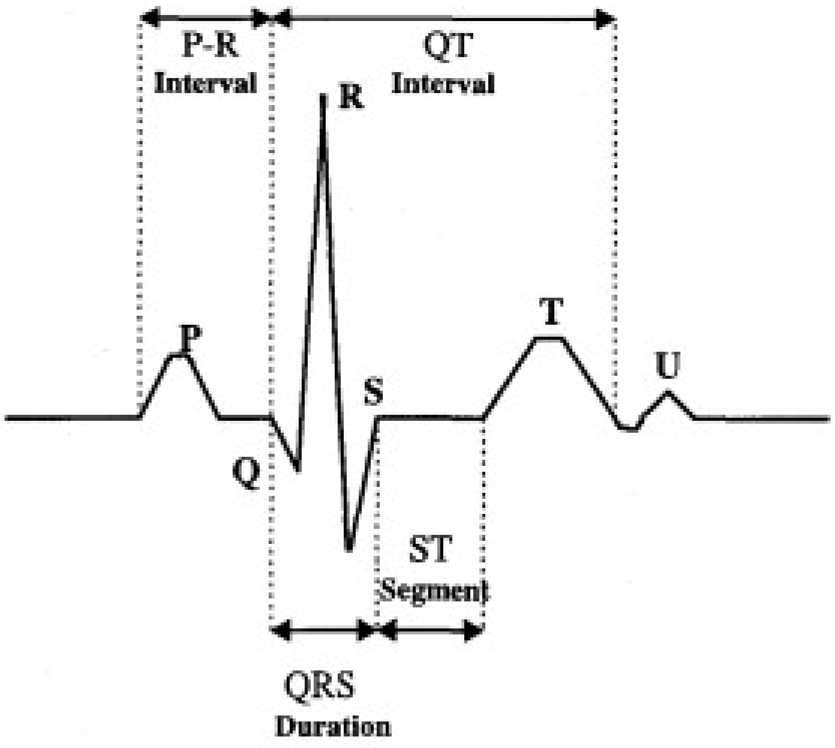


Fig. 1. A visualisation of an ECG complex. Source: H. Gholam-Hosseini and H. Nazeran [8]

Each part of the ECG complex corresponds to a different event occurring. The P wave shows the electrical activation of the atrial muscles, known as atrial depolarisation [6]. The QRS complex shows the electrical impulse spreading through the ventricles, known as ventricular depolarisation. Ventricular depolarisation results in the ventricles contracting, this is known as ventricular contraction. [6]. The T wave corresponds to the relaxation of the ventricles, which also reduces the pressure within the ventricles. This phenomenon is known as ventricular repolarisation [9]. The U wave is generally uncommon as the peak is very small, however, it can be seen in instances where a patient has a condition, such as Hypokalemia [10], a disease that results in a patient having low potassium levels due to electrical interactions between the ventricular myocardial layers (fibres found in the ventricles in the heart) [10].

The PR interval shows us the time taken for the excitation to spread from the sinoatrial (SA) node through the atrial muscles and the atrioventricular (AV) node [6]. The PR interval are measured by from the start of the P wave to the start of the QRS complex. The PR interval can tell us whether the conduction between the atria and ventricles is abnormal (a typical PR interval is between 120-200ms) [6]. The QT interval represents the time it takes for the ventricles to depolarise and repolarise. The QT interval is measured from the start of the QRS complex to the end of the T wave. Figure 2 shows the basic composition of the heart that is relevant to the processes that occur during an ECG complex.

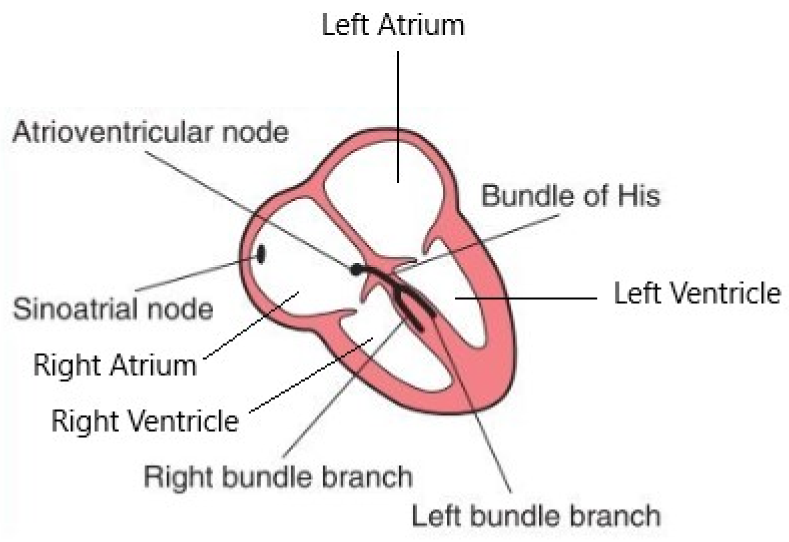


Fig. 2. A diagram of the heart. Source: J. Hampton [6]

The properties of an ECG complex is used to diagnose heart problems, where a slight irregularity can indicate that a patient is likely to have a particular disease [6]. A patient with a suspected cardiac arrhythmia (an abnormality with the heart's rhythm) may be subjected to an ECG, which will be analysed by a doctor in order to diagnose whether a patient has an issue with their heart. A correct diagnosis of a cardiac arrhythmia at an early stage can increase the chances of successful treatments [11]. The issue with manual interpretations of each ECG is that it is very time consuming, and requires individuals with a high level of expertise in the area. It is possible to train a machine learning model to identify patterns in an ECG that indicate a cardiac arrhythmia, like a experienced individual would, in order reduce the time it takes to diagnose an ECG, whilst maintaining a high level of accuracy.

2.3 Data Preprocessing

Data preprocessing is an important task in the data mining pipeline as it allows for improving the quality of the raw collected data [12]. The data collection process is often inconsistent as there can be numerous issues with the collected data, such as: missing data values, inappropriate sample, and noisy data. Data preprocessing consists of a variety of different techniques, some common techniques include data scrubbing, data normalisation, data transformation, and data reduction tasks; such as feature selection [13].

2.3.1 Data Normalisation

Data normalisation is an important practice as it addresses an algorithm using distribution to find patterns in the data. Furthermore, data normalisation is important step when the features in the dataset have different scales (e.g. age and weight). Min-max scaling is an example of data normalisation where the data is rescaled between 0 and 1. The equation for this technique is listed below.

$$x_{norm} = \frac{x - min}{max - min} \tag{1}$$

2.4 Machine Learning

Machine learning is a multi-disciplinary scientific field that combines concepts from multiple different fields of study, such as mathematics, physics, and neuroscience [14]. Machine learning is the use of algorithms to learn patterns from data, thus resulting in the ability to make predictions for classification [15]. Machine learning can be broken down into types of learning approaches, these learning approaches are commonly known as supervised, unsupervised, and reinforcement learning. The supervised learning paradigm is particularly useful for applications where we require the model to find patterns that associate to each class. This is done by training a machine learning model with labelled data. There are different approaches that can be adopted when choosing a model, using either a traditional machine learning method, or a neural network method.

A dataset of size $\{X_1, X_2\}$ has X_1 objects and X_2 features (or attributes). A feature is a measurable characteristic of the data, also known as a column in the dataset. An object is a group of values populating the data, also known as the rows in dataset. Machine learning has developed rapidly from theories to reality. The likes of Alan Turing contributing to many different subject areas such as mathematics, biology, philosophy, and to future areas now known as computer science, artificial intelligence, and machine learning [16]. Turing's paper talked about 'automatic machines' being machines which are "completely" determined by configuration at each step of motion [17], this was considered to be a foundation of computer science, these machines were first formally named by Alonzo Church, 1936, referring to them as ’Turing Machines’ [18]. The first reality of machine learning was in the form of representing how neurons of a brain work using electrical circuits, this was a simple model introduced by mathematician Walter Pitts and neurophysiologist Warren McCulloch [19], was a representation of neural networks. These foundations have outlined modern day machine learning, which has allowed us to use these machine learning algorithms, such as support vector machines (SVMs), for classification tasks in the medical domain. Traditional machine learning methods have been used to classify cardiac conditions. Celin et al. applied an SVM classifier along with feature selection and signal filters (to remove high frequency noise) to achieve an accuracy of 87.5% on the MIT-BIH arrhythmia and ECG–ID database [20]. Celin et al. also built an artificial neural network (ANN) using the back propagation algorithm to achieve an accuracy of 94% [20] on the MIT-BIH arrhythmia and ECG–ID database, showing that their ANN model performed better than their SVM model.

2.5 Evaluation Metrics and Loss Functions

2.5.1 Evaluation Metrics

An evaluation metric is a measurement that is used to evaluate how well a machine learning or statistical model performs, some examples are accuracy, precision, recall, f-measure, and loss. Some evaluation metrics used in this project are listed below:

A confusion matrix (of size $c \times c$, where c is the number of classes) is a representation (in tabular or matrix format, see Table 1) of how well a classification model performs, showing the true and predicted classifications for each object (instance) in the dataset. The basic elements of a confusion matrix include true positives, false positives, true negatives, false negatives.

True Positive (TP): The number of correctly predicted positives, meaning that the predicted class ($y_{pred} = 1$) is the same as the actual class ($y_{true} = 1$)

False Positive (FP): The number of incorrectly predicted positives, meaning that the predicted class ($y_{pred} = 1$) is not the same as the actual class ($y_{true} = 0$)

True Negative (TN): The number of correctly predicted negatives, meaning that the predicted class ($y_{pred} = 0$) is the same as the actual class ($y_{true} = 0$)

False Negative (FN): The number of incorrectly predicted negatives, meaning that the predicted class ($y_{pred} = 0$) is not the same as the actual class ($y_{true} = 1$)

True/Predicted	True (1)	False (0)
True (1)	TP	FN
False (1)	FP	TN

Table 1. This is an example of a Confusion Matrix for a binary class problem (Positive / Negative). The rows represent the true classes and the columns represent the predicted classes. The numbers in each cell represent how many instances were assigned by the classifier for that particular class. Each cell in a Confusion Matrix is linked have a meaning, these meanings are put in brackets in the Confusion Matrix. (TP = True Positive, FN = False Negative, FP = False Positive, TN = True Negative).

Assessing how well a model performed comes down to looking at the distribution of numbers in the main diagonal compared to the rest of the cells. A classifier that has performed perfectly will have all the predicted instances on the main diagonal, whilst being surrounding by zero objects predicted in the FP and FN cells. Using a confusion matrix, we are able to calculate some common evaluation metrics, these are listed in equations 2 to 6

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2}$$

$$ErrorRate = \frac{FP + FN}{TP + FP + TN + FN} \tag{3}$$

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall/Sensitivity = \frac{TP}{TP + FN} \tag{5}$$

$$F - Measure/F1Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6}$$

Evaluation metrics are an important consideration when assessing the performance of a model, as some evaluation metrics are do not give an accurate representation as to how well the model performed. Accuracy (eq. 2) and error rate (eq. 3) are commonly used evaluation metrics, where accuracy shows how many correct predictions a model has made. Error rate is the inverse of accuracy, where it measures the ratio of incorrect predictions made. Whilst these metrics can be used in cases where a dataset has an equivalent number of samples per class, it is not an appropriate metric when there is a class imbalance problem. The class imbalance problem is when there is not an equivalent number of objects in each class of the dataset, meaning that when a model incorrectly classifies an underrepresented class, an evaluation metric like accuracy would not accurately represent the performance of the model. Class imbalanced datasets are far more common in real world applications [21], therefore we need to use different evaluation metrics to assess how well our machine learning model has performed.

Precision (eq. 4) is used to describe how a classifier is able to predict the number of relevant objects from all the predicted values (for that specific class). Recall (eq. 5) describes the number of predicted objects that are in the the correct class. F-measure (also referred as the F1 Score, eq. 6) is the harmonic weighted mean of precision and recall [22], this means that this single metric is able to use both properties (precision and recall) to provide a better understanding of the classification problem, thus making f-measure a commonly adopted evaluation metric for imbalanced class problems [23].

2.5.2 Loss Functions

Loss functions, in the context of optimisation algorithms for neural networks, are functions that are used to evaluate the performance of a model given the set parameters. The type of loss function that you use depends on the type of problem that you aim to solve (e.g. regression, binary classification, and multi-classification problems). Some loss functions that are commonly used are described below:

Mean Squared Error (MSE)

MSE is a type of loss function that is commonly used in regression problems, where it is used to calculate the mean of the sum of squared distances between the true and target value. MSE is calculated using equation 7.

$$MSE = \frac{\Sigma_{i=1}^{n}(y_i^{true} - y_i^{pred})}{n} \tag{7}$$

where n is the number of objects in the dataset.

Binary Cross Entropy with Logits Loss

The loss function used in this project is known as Binary Cross-Entropy with logits loss (BCEWithLogitsLoss). This type of loss function is used for multi-class and multi-label classification problems, which is appropriate selection of loss function for this project as we are attempting a multi-label classification problem. This loss function consists of a Sigmoid layer function (an activation function) followed by Binary Cross-Entropy. The loss equation is shown in equation 8.

$$\ell(x,y) = L = \{l_1, ..., l_N\}^T, l_n = -w_n[y_n \cdot log\sigma(x_n) + (1-y_n) \cdot log(1-\sigma(x_n))], \tag{8}$$

where N is the batch size.

2.6 Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs), a subset of the machine learning domain, are algorithms that mimic the structure of the brain, consisting of thousands of connected nodes, formally known as artificial neurones (or neurones), arranged into 3 node layers: the input layer, one or more hidden layer(s), and the output layer[24].

The neurones in each layer are interconnected between layers, allowing data to pass through the neural network. Each of these artificial neurones has an associated weight and non-linear activation function (covered in chapter 2.8), see figure 3 for a visualisation of an artificial neurone. The ANN learns from the input data to find patterns that will help it perform its classification and recognition tasks. To understand how a neural network works, we can look at neural network known as a perceptron. The first artificial neurone, called the MCP neurone was developed by MuCulloch and Pitt (MCP). This basic form of neurone was a simple Boolean function that produced a 0 or 1 output [25]. The MCP neurone was adapted into the perceptron by Frank Rosenblatt [26]. The perceptron is the simplest form of a neural network, which consists of a single hidden (also known as intermediate) layer. The network has n inputs that have their individual weights, an activation function (expanded in chapter 2.8) that takes the sum of all these weighted inputs, and an output y, shown in figure 3. The perceptron is an example of a feed-forward neural network (see figure 4) that does not contain any cycles in the connections of the nodes. Since the perceptron only consists of one layer, it is only able to learn linear functions [27]. A more complex version of a perceptron is the multilayer perceptron (MLP). An MLP consists of multiple hidden layers between the input and output nodes. Since an MLP has multiple layers, it is able to learn non-linear functions, unlike the single layer perceptron mentioned earlier [27]. Another example of a feed-forward neural network is a residual neural network (ResNet), however, the ResNet consists of one fully connected layer at the output layer of the architecture. The ResNet utilises skip connections to get from shallow to deeper layers in the neural network. This will be covered in-depth in chapter 2.9.

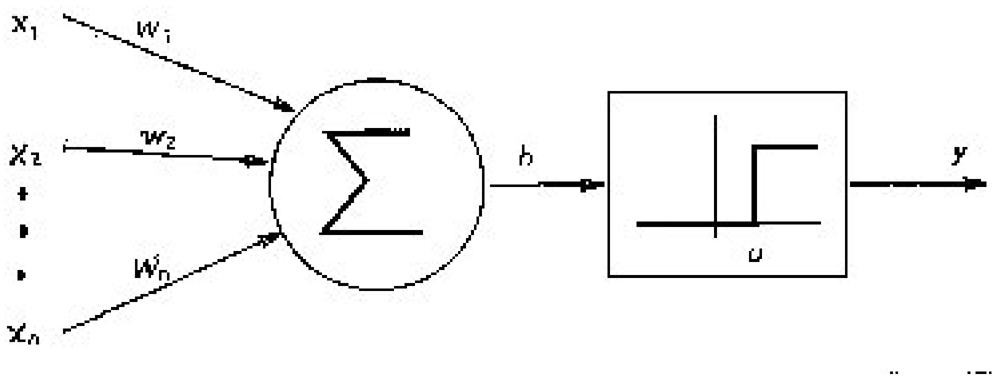


Fig. 3. A diagram of an artificial neurone. Source: A.K. Jain [28]

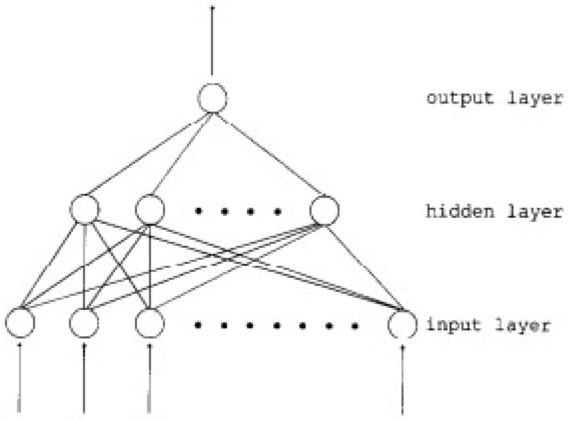


Fig. 4. An example of the structure of a feed-forward neural network. Source: D. Svozil [29]

2.7 Optimisation Algorithms

In order to improve the loss and convergence of an ANN, it is important to choose an optimisation algorithm. Optimisation algorithms change the parameters of an ANN until an optimum set of parameters are chosen, these parameters include weight and learning rate. As we construct deeper neural networks, the training time increases as it needs to pass multiple layers, therefore we adopt optimisation algorithms like gradient descent (GD) to improve the training efficiency of a neural network.

2.7.1 Gradient Descent (GD)

During training, an ANN uses an optimisation algorithm to optimise the weights of each layer in the neural network in order to minimise the error of the predictions made by the neural network. Gradient descent (GD) is an optimisation algorithm that is used to attain optimum values for the parameters of a function that minimises the cost function of the neural network. Figure 5 shows the steps in which GD takes in order to find an optima. GD utilises a learning algorithm known as backpropagation (algorithm covered in chapter 2.7.4) to calculate the gradient of the loss function (error gradient) with relation to the parameters of the ANN. The gradient descent algorithm is outlined below:

1. Randomly initialise weights$(w_1, w_2, ...w_n)$ and calculate the error using a cost function (f) of choice (e.g. mean squared error, MSE).
2. Calculate the error gradient using backpropagation. This is done by calculating the change in error from the cost function in relation to the updates of the weights.
3. Update the weights using the error gradient.
4. Using the new weights, calculate the new error value using the cost function.
5. Keep updating the weights and calculating the error value (steps 2 and 3) until the change in error is minimal, the error gradient ≈ 0.

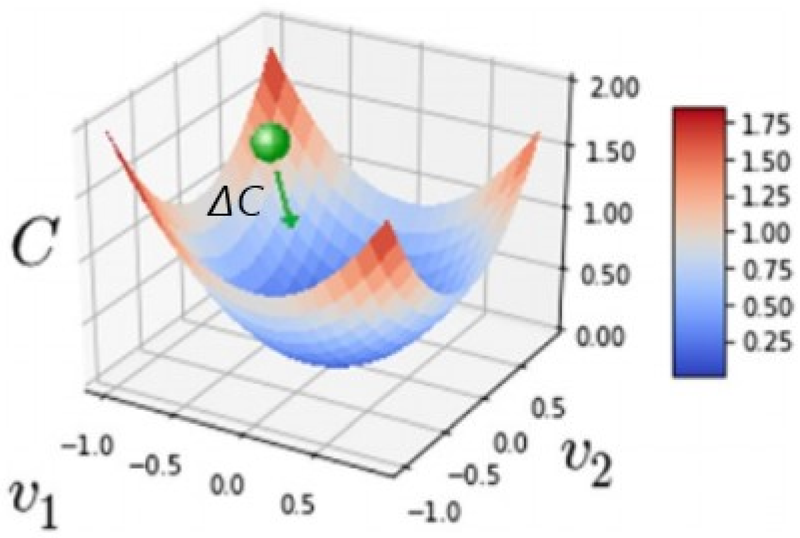


Fig. 5. A visualisation of GD where ΔC shows the change in the error gradient (the cost function). Source: P. Goyal [30]

2.7.2 Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) is another optimisation algorithm that is similar to gradient descent, however, it calculates an approximation of the error gradient by using a random subset of data (instead of using all of the data, as done in gradient descent). SGD is significantly reduces computation time as the iterations are faster, at the expense of a reduced convergence rate due to the noisy approximation of the true gradient [31]. This is beneficial when the training dataset is very large, resulting in the training time becoming a bottleneck. SGD has problems with traversing through surfaces where one dimension is much steeper than others [32]; this is common when there are local minima (optima). In order to solve this issue, SGD uses a method known as momentum to help it find the global minimum at a faster rate. Momentum works by increasing its momentum value for dimensions where the gradients are in the same direction, whilst reducing its momentum value for dimensions where gradients are often in different directions [32]. This results in faster convergence times, and reduced oscillations. Figure 6

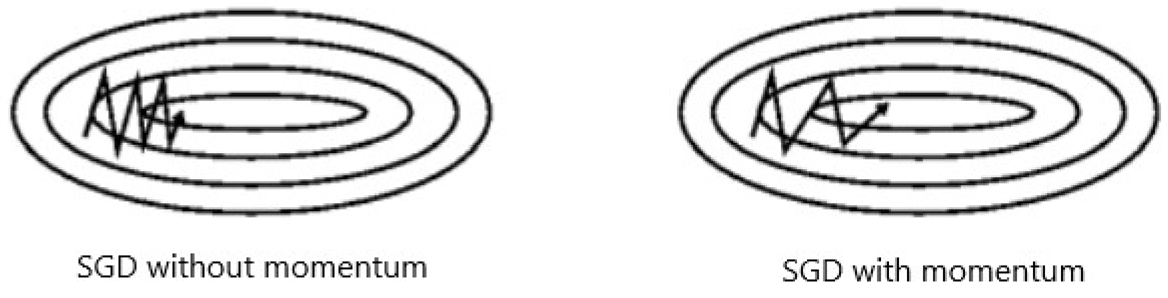


Fig. 6. A visualisation of SGD with and without momentum. Source: G. Orr [33]

2.7.3 Adaptive Momentum Estimation (Adam)

Adam is an extension of two SGD optimisation algorithm, known as AdaGrad and RMSProp, that is used to optimise deep learning networks. Adam computes individual learning rates for each parameter, which means that the learning rates are adaptive, whereas in SGD, the learning rate is a fixed to a specific value (α), in which the learning rate determines the rate in which the weights of a neural network are updated [34]. The Adam optimisation algorithm is explained in the steps below:

1. Requirements: Set the learning rate and the step size. Set the exponential decay rates ($[\beta_1, \beta_2]$) for moments. Set the a small numerical stabilisation constant (δ). Initialise moment values (these are momentum estimate values).

2. Take a mini-batch sub-sample of the training data ($X_1, ..., X_n$, where n is number of samples in the dataset).

3. Compute the gradients.
4. Update the first and second moment estimate values.
5. Make corrections to the biases of the first and second moment values.
6. Update the hyperparameters based on the steps above.

2.7.4 Backpropagation (BP)

As mentioned before, backpropagation (BP) is a learning algorithm that is used in neural networks to calculate gradient descent. The aim is to approximate the global minimum by using the steepest descent method. [35]. Backpropagation can be split into two mains steps: forwards propagation (forward pass) and backward gradient propagation. The forward propagation step calculates and stores the intermediate variables and outputs, in order from input to output layer, of an ANN. The backwards gradient propagation step refers to the method in which the gradients are calculated, in relation to the ANN's parameters. Unlike the forward pass, the backwards gradient propagation traverses through the ANN in the backwards order, from output to input layer [35]. The backpropagation algorithm is outlined below with the relevant mathematics:

1. Forward Propagation (Forward Pass) - Assume a multilayer perceptron (MLP; a type of ANN that consists of multiple hidden layers) has L hidden layers with neurones that have the activation function $f(\cdot)$ and the output layer of the activation function $g(\cdot)$. The training example is denoted as follows: $(x, y) \in \mathcal{B}_{k+1}$ (mini-batch), where x is the training data sample, y is the output,

(a) Input: $h^{(0)} \leftarrow x$ (h is the layer, where layer 0 is input layer)

(b) Compute the activate of neurones in hidden layers (for l = 1,2,...,L) by using the equation $y = wx + b$ (for w is weight, b is bias, x is data)

$$a^{(l)}(x) = W_k^{(l)} h^{(l-1)}(x) + b_k^{(l)}, h^{(l)} \leftarrow f\left(a^{(l)}(x)\right) \tag{9}$$

(c) Compute the activation of the output neurones (where the output layer is referred to as $L + 1$

$$a^{(L+1)}(x) = W_k^{(L+1)} h^{(L)}(x) + b_k^{(L+1)}, \hat{y} = h^{L+1} \leftarrow g\left(a^{(L+1)}(x)\right) \tag{10}$$

(d) Compute the loss of the training sample ($\mathcal{L}$ is Loss, Θ is parameters): $\mathcal{L}(\Theta_k; x, y)$, where $\Theta_k = \{W_k^{(l)}, b_k^{(l)}\}_{l=1}^{L}$

2. Backward Gradient Propagation - Now that the loss of the training samples are obtained, we can calculate the partial derivatives of the loss function $\mathcal{L}$ in respect the parameters (weight W and bias b) of the neural network.

(a) Compute the gradient at the output layer, $L + 1$.

$$\delta^{(L+1)}(x, y) \leftarrow \frac{\partial \mathcal{L}(\Theta_k; x, y)}{\partial a^{L+1}(x)} = \frac{\partial \mathcal{L}(\Theta_k; x, y)}{\partial h^{L+1}(x)} \odot g'\left(a^{(L+1)}(x)\right) \tag{11}$$

(b) Compute the gradients at all hidden layers (for $l = L, L+1, ..., 1$)

$$\frac{\partial \mathcal{L}(\Theta_k; x, y)}{\partial h^{(l)}(x)} \leftarrow \left(W_k^{(l+1)}\right)^T \delta^{(l+1)}(x, y), \delta^{(l)}(x, y)$$
$$\leftarrow \frac{\partial \mathcal{L}(\Theta_k; x, y)}{\partial h^{(l)}(x)} \odot f'\left((a^{(l)})(x)\right)$$

(c) Update the parameters on the mini-batch (for $l = L, L+1, ..., 0$)

$$b_{k+1}^{(l+1)}(x) \leftarrow b_k^{(l+1)}(x) - \frac{\eta}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}_{k+1}} \delta^{l+1}(x, y), \tag{12}$$

$$W_{k+1}^{(l+1)} \leftarrow W_k^{(l+1)} - \frac{\eta}{|\mathcal{B}|} \sum_{(x,y)\in\mathcal{B}_{k+1}} \delta^{l+1}(x, y)\left(h^{(l)}(x)\right)^T \tag{13}$$

2.8 Activation Functions

An activation function of a node takes an input and will produce a given output depending on the mathematical properties of the function, thus deciding whether an artificial neurone will be activated [36]. There are two types of activation functions, linear and non-linear. A linear activation function can be described in the form of equation 14.

$$f(x) = w^T x + b, \tag{14}$$

where w = weight, x = input, b = bias.

Using linear activation functions in neural networks only work when the data is linear, meaning that the neural network will not be able to learn on complex (non-linear) data [37]. The output of models using linear activation functions is defined in equation 15

$$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b \tag{15}$$

Non-linear activation functions are the most commonly used activation functions as they are suitable for deep neural networks that use non-linear data. Non-linear activation functions are useful as they allow for backpropagation, this is because they are differentiable, meaning that there is a derivative at each point in its domain [36]. The output of models using non-linear activation functions are defined in equation 16.

$$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b) \tag{16}$$

where f is the activation function.

The choice of activation function used in a neural network is important as each have their own advantages and disadvantages. We will now highlight some important non-linear activation functions, in which we will discuss how they work, and their advantages and disadvantages.

2.8.1 Sigmoid Activation Function

The sigmoid activation function has output values between the bounds of 0 and 1, making it useful for models with probability outputs, ash shown in figure 7. The sigmoid activation function can be expressed in the following equation:

$$f(x) = \left(\frac{1}{(1 + exp^{-x}} \right) \tag{17}$$

where x = input, exp = Euler's number ≈ 2.71828.

A disadvantage of the sigmoid activation function is the vanishing gradients problem during backpropagation [36]. This vanishing gradient problem arises when the absolute value of the input increases in very small steps, resulting in the gradient to become very small. The vanishing gradients problem is an issue as weight updates become negligible, meaning that when the neural network is training, it will find it difficult to converge.

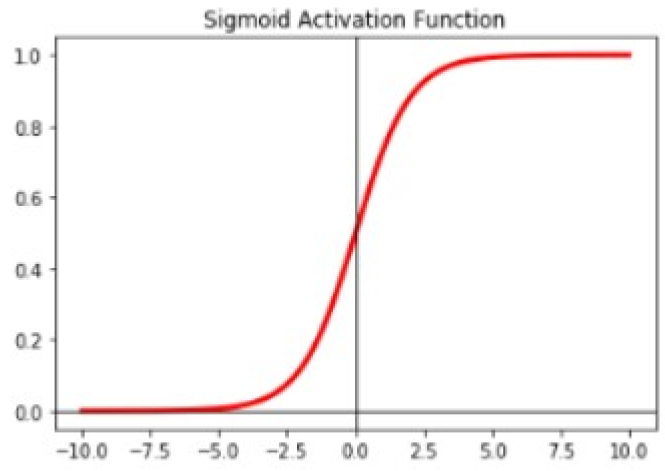


Fig. 7. Sigmoid activation function Source: I. Kandel [38]

2.8.2 Rectified Linear Unit (ReLU) Activation Function

The rectified linear unit (ReLU) activation function is the most extensively used activation function for deep learning applications [36]. The ReLU very nearly represents a linear function, thus the activation function is able to retain its linear properties that make them easier to optimise in conjunction with optimisation methods, such as SGD [36]. The ReLU activation function applies a threshold to the input (x), as shown in a graph in figure 8, and in a mathematical representation in equation 18.

$$f(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases} \tag{18}$$

where x = input.

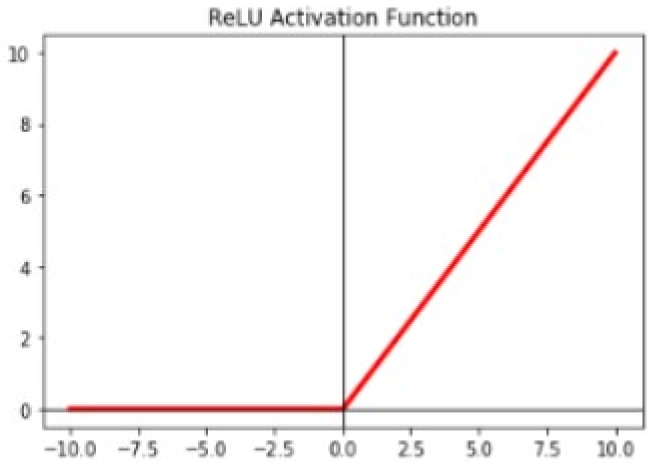


Fig. 8. ReLU activation function Source: I. Kandel [38]

The computation when using the ReLU activation function is considerably faster as it does not compute exponentials [36]. A disadvantage with the ReLU can be its tendency to overfit, in comparison to the sigmoid activation function.

2.8.3 Softmax Activation Function

The softmax activation function is used in multi-class classification models (when an object in the dataset can belong to more than one class). The softmax returns the probabilities of the object belonging to each class, where all the probabilities in each class sum to 1. The softmax function is mathematically written in equation 19.

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{19}$$

2.9 Residual Neural Networks (ResNets)

A Residual neural network (ResNet) is a type of artificial neural network that are commonly adopted for various different applications. A ResNet consists of residual blocks that use skip connections allow for easier information traversal throughout the neural network. A common problem known as vanishing gradients occur amongst neural networks that use gradient-based learning methods. This problem occurs when the number of layers in the network increase, therefore the gradients become very small (vanishing), which in turn prevents the weights of the neural network from being updated [39]. ResNets address this problem by utilising skip connections in order to provide an alternative route for information to pass through the neural network. These skip connections bypass intermediate layers in order to connect shallow layers to deeper layers [40]. Figure 9 visualises the residual block of a ResNet, showing the skip connection that bypasses the intermediate layers.

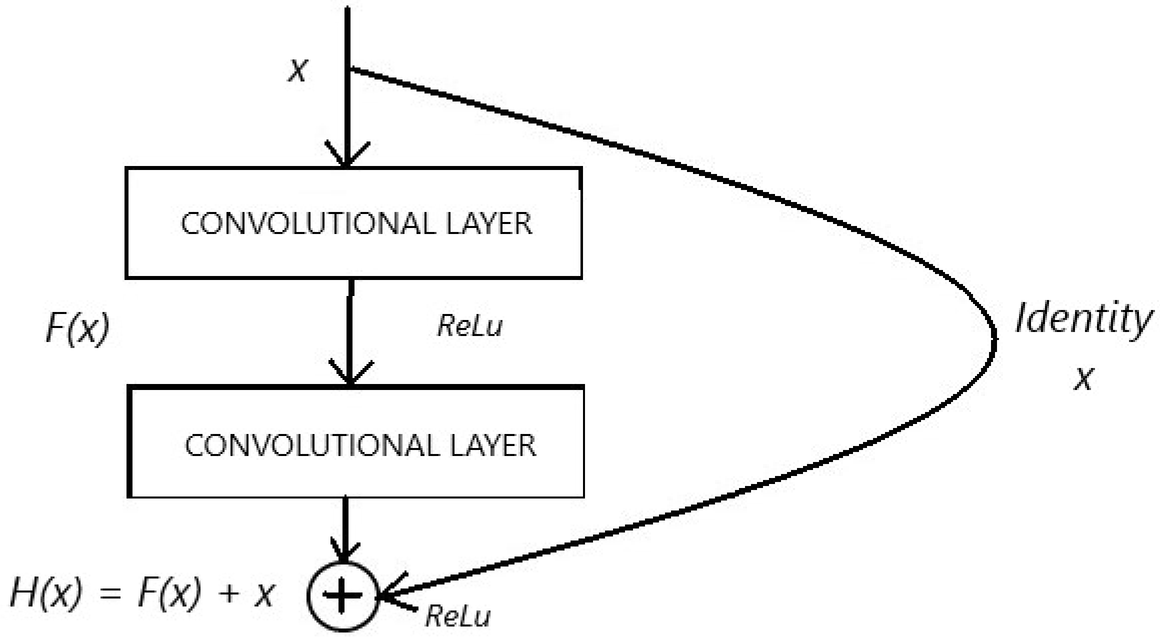


Fig. 9. A visualisation of a residual block

2.10 Data Augmentation

Data augmentation via the creation of synthetic data has been used in the past to deal with class imbalance problems [41], [42]. Class imbalance is a frequent problem in which some classes in the dataset are over-represented, and some classes are under-represented [43]. This results in a poor distribution of training instances for each class, which can affect the ability of the model to generalise as the model as there is a large variation in the objects that fall under each class in the dataset. A solution to this class imbalance problem is generating synthetic data. Synthetic data is created by using information from existing data in order to recreate a synthetic sample, meaning that this synthetic sample is not real, however, it is able to represent the information in the under-represented classes, provided that synthetic data generation is done appropriately. A method in which synthetic data can be generated is by using generative adversarial networks (GANs). GANs use a backpropagation algorithm with two neural networks to produce synthetic data [44]. The creation of synthetic ECG signals have been explored in the past. Golany et al. explored the idea of improving ECG classification using an LTSM (type of ANN) model with a deep convolutional generative adversarial network (DCGAN), finding that the performance of their LTSM model improved when including their synthetic data generated by the DCGAN [45].

Generative Adversarial Networks (GANs)

Generative adversarial networks (GANs) are used to address generative modelling problems [46], which is commonly related to creating synthetic data for class imbalance problems. The GAN consists of two neural networks, the first neural network is used to generate the synthetic data, known as the generator, and the second is used to discriminate the real and synthetic data, known as the discriminator. The generator is initialised with random variables, creating a piece of synthetic data. This synthetic data is then fed to the discriminator, where the discriminator is using training data of the real data as a ground truth to make a prediction as to whether the synthetic data is real or synthetically generated. Once a prediction is made, the loss is calculated, which is then used in the backpropagation algorithm to obtain the gradients. These gradients are then used to make adjustments to the weights of the generator [44]. Figure 10 visualises the components of a GAN.

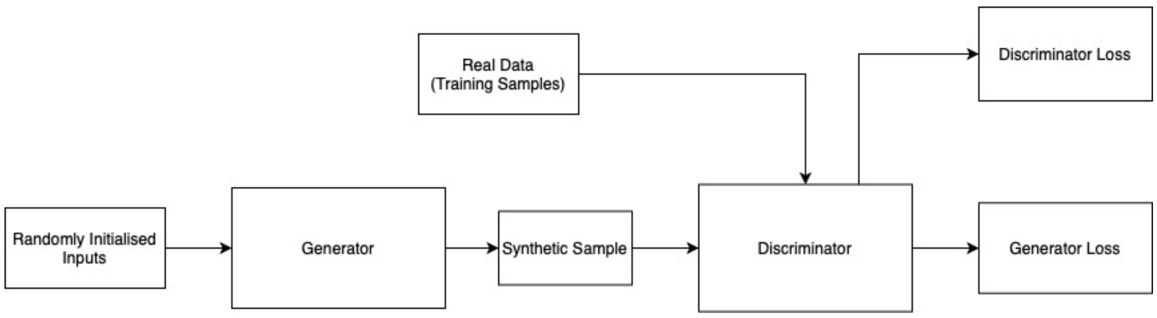


Fig. 10. A diagram of a Generative Adversarial Network.

McSharry and Clifford's Dynamic Model for synthetic ECG generation

McSharry and Clifford propose a dynamic model that utilise differential equations for a realistic ECG generation. The dynamic model uses operators that can specify the characteristics of the signal, such as the standard deviation and mean heart rate, and the PQRST morphology of the ECG complex [47]. This method has shown that it is able to generate a realistic ECG signal by using the characteristics of a real ECG. Figure 11 shows a comparison of a synthetic and real ECG signal generated by McSharry and Clifford.

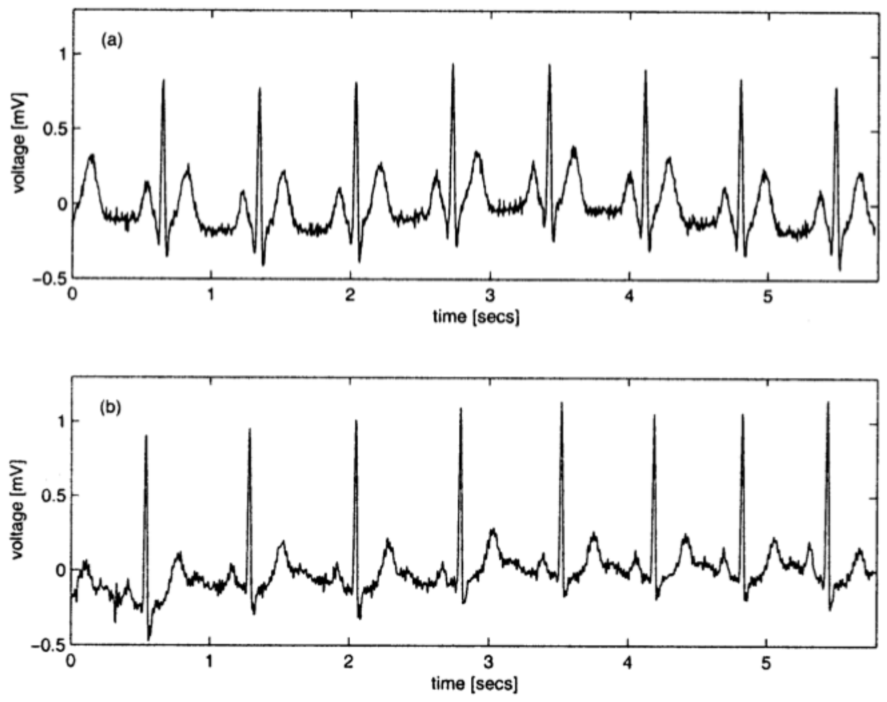


Fig. 11. A comparison of a synthetically generated ECG signal (top) and a real ECG from a human (bottom). Source: P. McSharry [47]

2.11 Machine Learning Applications for ECG Classification

When applying neural networks to a classification task, the key problem is defining an appropriate architecture for the neural network to work well with the dataset that is being used [48]. ResNets perform well on many different tasks, including computer vision related problems such as image classification and object detection [49]. In addition to computer vision, ResNets have been used in the past to predict the abnormalities in electrocardiograms [7], [50], [51]. Li et al. constructed a deep residual convolutional neural network in order to perform classification of different types of arrhythmias [52]. They managed to achieve an accuracy of 99.06% and 99.38% [52] for 1 and 2-lead ECG recordings respectively (using the

MIT-BIH arrhythmia database). Jia et al. built a SE-ResNet34 deep neural network on a collection of four different datasets (41,101 patients) to achieve a validation score of 0.653 in the PhysioNet/Computing in Cardiology Challenge 2020. Jia et al. found that the ResNet they built performed well, however they address that generalisation of the model may be a potential problem due to the lack of appropriate data [51]; the 'appropriate data' is referencing to the severe class imbalance problem. This class imbalance problem may be addressed by using data augmentation techniques. Nonaka and Seita address this issue by applying signal processing techniques to change the hyperparameters of existing ECG recordings to create slight variations in the existing ECG recordings, which augment the current dataset that is provided. Nonaka and Seita focus on improving classification for atrial fibrillation, where they managed to increase the F1-score by 3.17% from the baseline performance of their residual convolutional deep neural network [53]. Another type of ANN used for ECG classification is the convolutional neural network (CNN). Kiranyaz et al. proposed a 1-dimensional 3-layer CNN that performs feature extraction and classification using raw ECG data from the MIT/BIH arrhythmia dataset. Kiranyaz et al. focused on ventricular ectopic beat (VEB) and supraventricular ectopic beat (SVEB) detection, resulting in an accuracy of 99% (VEB) and 97.6% (SVEB) [54]. Pyakillya et al. mention that there is an abundance of traditional machine learning applications for ECG classification, but that the current applications only utilise engineered features, which is seen as suboptimal due to the problem of finding the most appropriate features [55]. Pyakillya et al. propose using a deep CNN for ECG classification, where the first convolutional layers conduct the feature extraction process. This allows the CNN to find the most appropriate set of features to produce the most accurate results. The CNN resulted in an accuracy of 86% on the validation dataset (CinC/Challenge

2017 dataset), but mentioned that the accuracy of their model could be improved with the assistance of data augmentation using GANs to reduce the class imbalance problem [55], as mentioned by Jia et al. [51]. Zhou et al. combine a ResNet with a Bidirectional long short-term memory (BiLSTM), a type of recurrent neural network, for ECG classification [56]. The dataset they use comes from the MIT-BIH arrhythmia database, and Physionet's PTB diagnostic database. When implementing the ResNet, they mention that the depth of the ResNet should be carefully chosen since the ResNet will suffer from vanishing or exploding gradients. The combination of the ResNet and the BiLSTM resulted in an accuracy of 96.2% on the MIT-BIH dataset, and 99.6% on the PTB dataset [56]. Zhou et al. mentioned that their model is very deep, resulting in the training time being very long. They mention that a future improvement can be to make the model more computationally efficient [56].

2.12 Dataset

The PhysioNet/Computing in Cardiology Challenge 2020 [57] has provided labelled training data from multiple sources that will enable the option for supervised learning. This data consists of the patient's age, sex, 12-lead ECG data, and the disease the patient is suffering from. This data will be subjected to data preprocessing in order improve the quality of the data. The data is compromised of 37,749 labelled samples, with 111 different classes. For this project, we will be modelling on 27 of the 111 classes (see figure 12 for list), as these diagnoses are most common. A heat map of the distribution of the data for each class (diagnosis) is shown in figure 13.

Diagnosis	Code	Abbreviation
1st degree AV block	270492004	IAVB
Atrial fibrillation	164889003	AF
Atrial flutter	164890007	AFL
Bradycardia	426627000	Brady
Complete right bundle branch block	713427006	CRBBB
Incomplete right bundle branch block	713426002	IRBBB
Left anterior fascicular block	445118002	LAnFB
Left axis deviation	39732003	LAD
Left bundle branch block	164909002	LBBB
Low QRS voltages	251146004	LQRSV
Nonspecific intraventricular conduction disorder	698252002	NSIVCB
Pacing rhythm	10370003	PR
Premature atrial contraction	284470004	PAC
Premature ventricular contractions	427172004	PVC
Prolonged PR interval	164947007	LPR
Prolonged QT interval	111975006	LQT
Q wave abnormal	164917005	QAb
Right axis deviation	47665007	RAD
Right bundle branch block	59118001	RBBB
Sinus arrhythmia	427393009	SA
Sinus bradycardia	426177001	SB
Sinus rhythm	426783006	NSR
Sinus tachycardia	427084000	STach
Supraventricular premature beats	63593006	SVPB
T wave abnormal	164934002	TAb
T wave inversion	59931005	TInv
Ventricular premature beats	17338001	VPB

Fig. 12. The list of diagnoses and their CT-Codes. Source: PhysioNet/Computing in Cardiology Challenge 2020 [57]

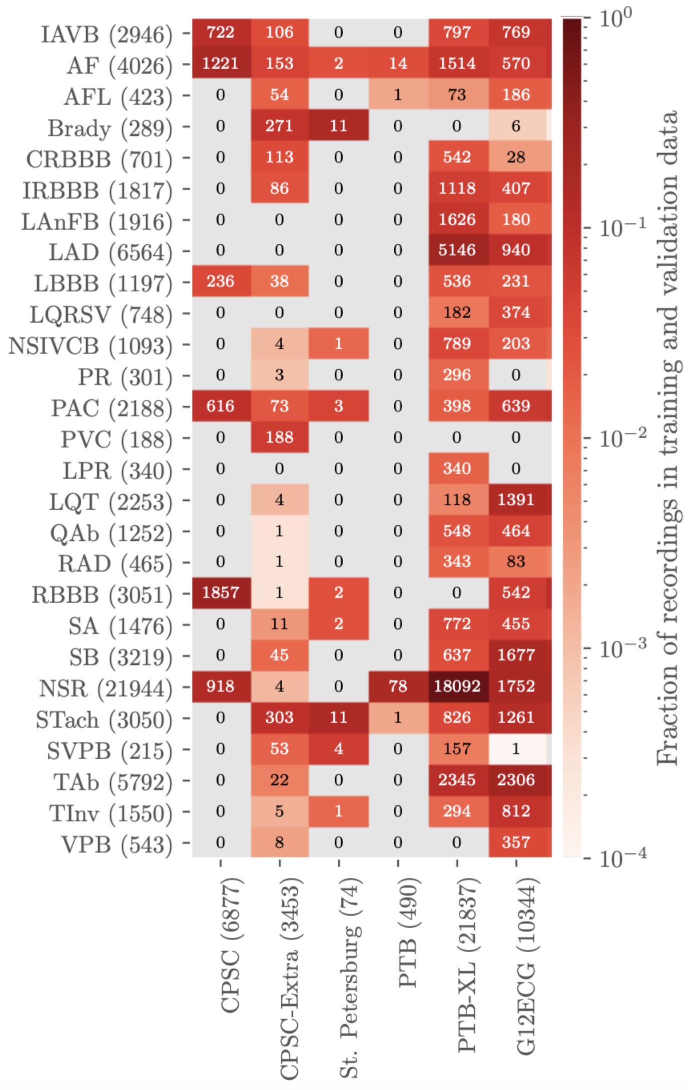


Fig. 13. The distribution of the classes in the dataset. Source: PhysioNet/Computing in Cardiology Challenge 2020 [47]

3 Research Methodology and Implementation

In order to execute a successful project, it is paramount that a suitable methodology is chosen. The design of the experiment must be chosen appropriately by researching the techniques that may improve the experiment. This chapter will discuss the research methodology of how we will predict cardiac arrhythmias using ResNets. This will propose solutions in order to implement each part of the project. This section will start by introducing the data preparation processes. The next part will outline an architecture of the ResNet that is going to be developed. Finally, we will discuss some data augmentation techniques that were researched but not pursued (reasons will be expanded on later on).

3.1 Methodology

Approaching a classification problem is commonly approached in a similar manner. The first step to approaching a problem starting with doing background research and reviewing surrounding literature to understand what the project entails. This step involves considering which data we will use, the type of models that would favour the project, and the type of evaluation metrics we can use to determine whether the developed model is reliable. Once these steps are complete, we can start to move onto the technical steps in the project. The initial technical steps involve starting with the data sourcing and preparation processes, this includes data gathering, standard data preprocessing, and data transformations to suit the requirements of future modelling. In order to meet the aims of the project, we would need to consider the type of data we will need to source, this means that the initial feature set needs to be considered. Most often the raw data that is obtained is not fit for immediate modelling, therefore increasing the quality of the data will be a key step in order to produce reliable results. Improving data quality can be done by cleaning the data, which can compromise of: dealing with incomplete rows in the dataset, removing anomalies, and changing data types. Data transformation is an important step in the data preprocessing stage as it makes it easier for both the programmer and the model to understand the data [58]. The initial data preparation phase is followed by the process of applying a machine learning model to the processed data. The first step in this process is to select the type machine learning model that will fit the data. The type of model that is appropriate for a project is determined by a large number of factors. Some common factors may include: some models may perform better on a specific type of data mining problem, whether the problem requires a regression (mapping a function (f) from an input variable (x) to a continuous output variable (y)) or classification model, which model works best on a features set ($X_f = \{X_1, X_2, ..., X_n\}$), and the training time of the model. Once an appropriate model is chosen, trained, and tested; the final step is to use the results from the testing phase to extract knowledge from the model. This is done by using an appropriate evaluation metric that accurately represents the performance of the model.

The goal of this project is to experiment with neural networks in order to classify 12-lead ECGs, furthermore, research into the options of data augmentation in order to improve the classification performance of neural networks. The pipeline of this project is shown in figure 14.

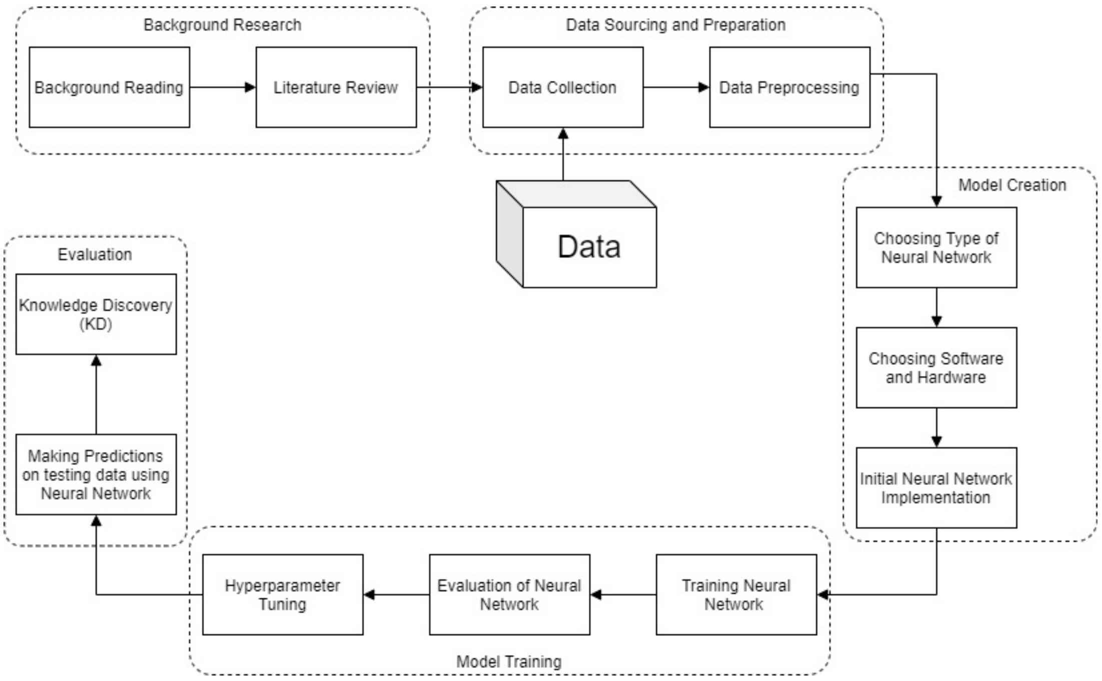

Background Research
Background Reading
Literature Review
Data Sourcing and Preparation
Data Collection
Data Preprocessing
Data
Model Creation
Choosing Type of Neural Network
Choosing Software and Hardware
Initial Neural Network Implementation
Model Training
Training Neural Network
Evaluation of Neural Network
Hyperparameter Tuning
Evaluation
Making Predictions on testing data using Neural Network
Knowledge Discovery (KD)

Fig. 14. An overview of the project implementation process.

Data Augmentation

During the research methodology phase, the issue of class imbalance was noted, as surrounding literature suggested that classifying underrepresented classes was difficult, which was seen from the submissions to the PhysioNet/Computing in Cardiology Challenge 2020. Upon doing research, two methods that could be used to addressed this problem was encountered, the McSharry and Clifford dynamic model, and the use of GANs. The McSharry and Clifford model utilised differential equations for ECG generation, which was found to be a fitting solution for the data augmentation problem. However, this project aims to utilise all 12-leads for ECG classification, which would mean generating synthetic data for all 12 leads for each individual lead. This approach would favour a project which solely investigates boosting the performance of a classification model of all underrepresented classes. Due to this reason, the next step was to evaluate the use of

GANs for data augmentation. As mentioned previously, GANs are very commonly used for generating synthetic data via neural networks. The main issue with this is that the GAN would require enough training data to be able to generate a realistic ECG which manages to represent all the characteristics of an irregular ECG. Initial implementations of the GAN had started, however, it seemed unfeasible to implement this due to the lack of training data in the underrepresented classes, which posed a risk of ruining the results of the entire project if the GAN could not recreate the characteristics of the cardiac arrhythmia in the ECG.

3.2 Implementation

In the methodology that was outlined previously, a brief overview was given to explain the pipeline of a machine learning project. The key steps were data sourcing and preparation, model creation, model training, and model evaluation. Applying the methodology to this project will be outlined, furthermore, showing the implementation of each step in the project.

The experimental design has outlined an overview of the steps that will be taken to complete the project. The following section will show the experimental design and the implementation of the step in the project. The programming of each step will be broken down into detail.

3.2.1 Software and Hardware

For this project, there are some key software and hardware used in order to complete this project. A list of the main software used for this project is listed below.

1. **Python (v.3.8.5)**

   For this project, we will be programming in Python. Python is an open source, high level programming language that is commonly used for machine learning applications. Python has an abundance of useful libraries that will be used to complete this project.

2. **PyTorch (v.1.8.0)**

   PyTorch is an open source, tensor optimised machine learning library [59] that is used for deep neural network applications. PyTorch will be used to develop the residual neural network for this project.

3. **NumPy (v.1.20.2)**

   NumPy is a scientific computing package for Python, which provides a wide range of mathematical computing tools [60].

4. **pandas (v.1.2.4)**

   The pandas library is an open source data analysis and manipulation tool that is built on the Python programming language. The use of pandas tools will help prepare the raw data for modelling.

5. **Git**

   Git is an open source distributed version control system that allows for keeping track of source code history [61]. Git will be used to ensure that all the source code is managed. The source code for this project is in a private repository that is on the cloud, which mitigates the risk of losing source code due to hardware failures.

6. **CUDA and CUDA Deep Neural Network (cuDNN)**

For this project, we will be utilising CUDA, a parallel computing platform developed by NVIDIA that utilises the graphics processing unit (GPU) for processing [62]. CUDA can be utilised to speed up training time as performs operations in parallel.

In order to utilise CUDA for neural network training, it is recommended to have hardware that is able to cope with the tasks it will have to perform. Table 2 lists the specifications of the hardware for this project.

Component	Model
CPU	Intel i7-7700 @ 3.6 GHz
GPU	NVIDIA GeForce GTX 1080 8GB
RAM	16GB DDR4 @ 2133 MHz

Table 2. Hardware specifications for residual neural network training.

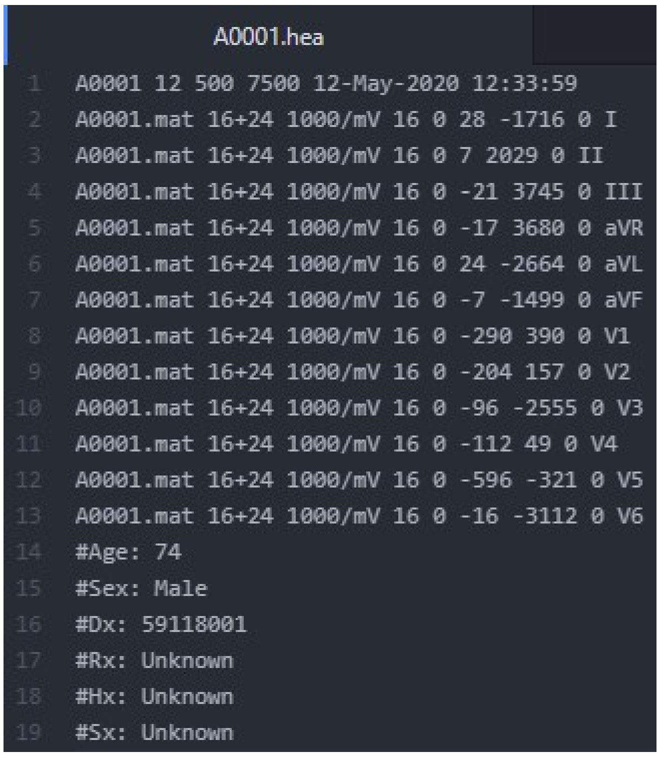
3.2.2 Data Sourcing

In order for the ANN to successfully predict cardiac arrhythmias, it is essential that we source enough appropriate data for training as it will improve the predictive power of the ANN. The data that will be used in this project is sourced from the PhysioNet/Computing in Cardiology Challenge 2020 [57], where they have gathered 6 datasets from the following 4 different sources (An in-depth breakdown of the data from each source is given in Table 3):

- China Physiological Signal Challenge 2018 (CPSC and CPSC Extra)
- St Petersburg INCART 12-lead Arrhythmia (St. Petersburg)

- Physikalisch Technische Bundesanstalt (PTB and PTB XL)
- Georgia 12-lead ECG Challenge (Georgia)

The datasets provided by the consisted of a header text file (see figure 15, which gave information about the patient (age, sex, disease), the details of the ECG recording (e.g. sampling frequency), and an ECG recording of the patient's heart (see figure 16. The header text file has a large amount of information regarding the patient and their ECG recording. Using the header file shown in figure 15, we can see that the recording number is referred to as A0001, thus the file name is "A0001.mat". The first line of the header file provides information about the number of leads (line 1: "12"), the sampling frequency (line 1: "500"), and the number of samplings points in the lead (line 1: "7500"). The following lines provide information regarding the way the lead was recorded (lines 2-13). Lastly, the remaining lines provide the demographic (line 14/15: "Age" and "Sex") and diagnosis (line 16: "Dx") information.



```
A0001 12 500 7500 12-May-2020 12:33:59
A0001.mat 16+24 1000/mV 16 0 28 -1716 0 I
A0001.mat 16+24 1000/mV 16 0 7 2029 0 II
A0001.mat 16+24 1000/mV 16 0 -21 3745 0 III
A0001.mat 16+24 1000/mV 16 0 -17 3680 0 aVR
A0001.mat 16+24 1000/mV 16 0 24 -2664 0 aVL
A0001.mat 16+24 1000/mV 16 0 -7 -1499 0 aVF
A0001.mat 16+24 1000/mV 16 0 -290 390 0 V1
A0001.mat 16+24 1000/mV 16 0 -204 157 0 V2
A0001.mat 16+24 1000/mV 16 0 -96 -2555 0 V3
A0001.mat 16+24 1000/mV 16 0 -112 49 0 V4
A0001.mat 16+24 1000/mV 16 0 -596 -321 0 V5
A0001.mat 16+24 1000/mV 16 0 -16 -3112 0 V6
#Age: 74
#Sex: Male
#Dx: 59118001
#Rx: Unknown
#Hx: Unknown
#Sx: Unknown
```

Fig. 15. File "A0001.hea" is a header text file containing basic information about the patient and the associated ECG recording.

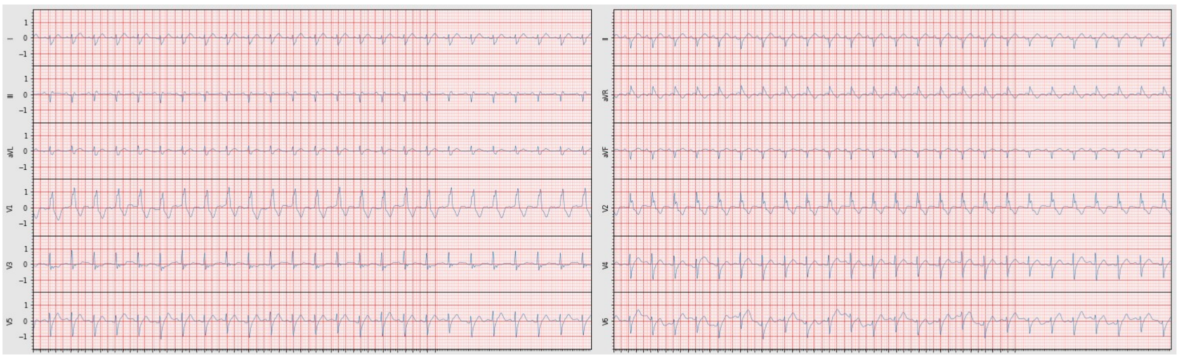


Fig. 16. A recording of a 12-lead ECG.

The different sources provide ECG recordings from 4 different regions around the world, meaning that the diversity in the data reduces the results being based on one demographic. The raw data compromises of demographic information about the patient (age, sex, gender), information on the ECG recording (sampling frequency, length of signal, and number of samples), the 12-lead ECG recording, and the diagnosis of the patient. The data is anonymised so there is not an implication with sensitive information. For this project, the only data that will be used to train on the neural network is the 12-lead ECG recordings, whilst there have been implementations to use less than 12-leads (some have used 1-lead [63], [64]), using a deep neural network on all 12-leads of the ECG results in better performances [65].

3.2.3 Data Preprocessing

In order for the ANN to make predictions on the ECG recordings, in needs to undergo data preprocessing. The raw format of the header text file and ECG recordings need to be stored in a usable format, such as a dataframe or NumPy array. Once the raw data is loaded into a data manipulation tool, it needs to be processed and inspected as the quality of the data could be low. Missing values, anomalies, and corrupted files will be checked and dealt with to increase the quality of the data.

Extracting Data from Header Text File Implementation:
In order to extract the relevant information from the header text files, we need to implement the two functions listed below (see figure 17 for a flow chart that visualises the process):

- **file_parse():** In order to extract the patient data from the header file, I have created a simple python script that can scrape the data that we need from the header file. The important information that we need is the age, sex, and diagnosis. The data scraped from the header files is stored in a table form data structure known as a dataframe. The code for this function is shown below:

```python
# Parse header file
def file_parse(file_locs, df, keys):
    file_no = -1
    for path in file_locs:
        with open(path, 'r') as file_object:
            file_no += 1
            temp_arr = []
            for line in file_object:
                    for key in keys:
                        if key in line:
                            temp_arr.append(line.split(key,1)[1][:-1])

            df.loc[file_no] = [temp_arr[0]] + [temp_arr[1]] + [temp_arr[2]]

    return df
```

Listing 1. A file parser function for the header text file data.

- **extract_hea_data():** This function will output a comma-separated values (CSV) file containing all the data that we need from the header file. This function calls the **file_parse()** function in order to scrape the header file data. The function also replaces the diagnosis codes with the CT codes. These CT codes are abbreviations of the diagnosis, making it easier to identify the patient's diagnosis. The code for this function is shown below:

```python
# Extract all header data and diagnosis classes
```

```python
def extract_hea_data(inputdir, outputdir, fs):
    # Initialise dataframe with necessary column names
    patientrecords = pd.DataFrame(columns=['Age','Sex','Dx'])
    # Get list of all files in the input directory
    all_files = glob.glob(inputdir)
    # Parse the header file and storing it in dataframe
    df1 = file_parse(all_files, patientrecords, ['Age: ','Sex: ','Dx: '])
    # Insert sampling frequency column
    fscol = [fs] * df1.shape[0]
    df1["FS"] = fscol
    # Read csv file containing all the diagnosis codes (CT Codes)
    ct_codes = pd.read_csv("../Data/Dx_map.csv")
    replace_df = ct_codes.iloc[:, 1:]
    # Replace numerical with abbreviation CT Codes
    for i in range(df1.shape[0]):
        for j in range(0, len(list(replace_df.iloc[:,0]))):
            if str(replace_df.iloc[j,0]) in str(df1['Dx'].iloc[i]):
                df1['Dx'].iloc[i] = df1['Dx'].iloc[i].replace(str(replace_df.iloc[j,0]), str(replace_df.iloc[j,1]))
    print(df1.shape)
    # Store dataframe as csv in output directory
    df1.to_csv(outputdir)
```

Listing 2. Extracting header data from file and storing it as a CSV file for later use.

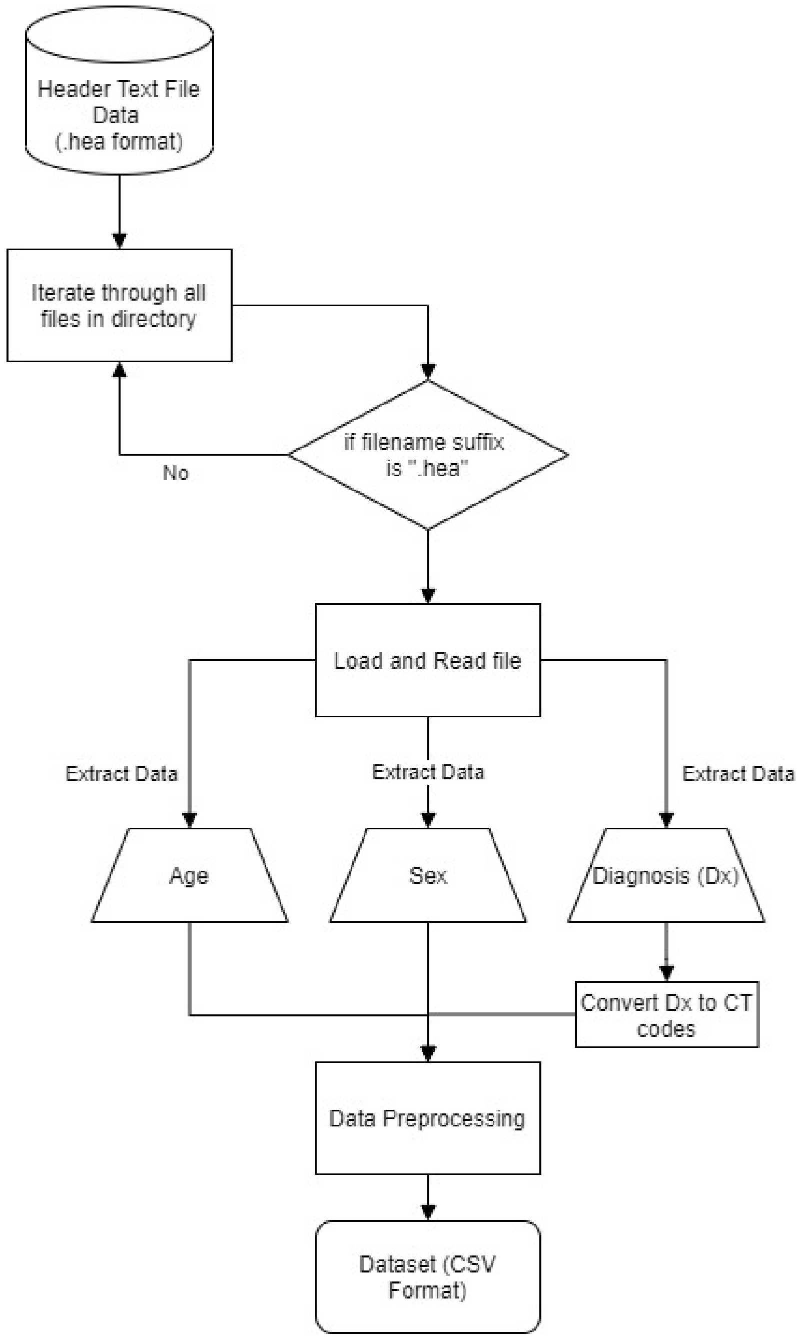


Fig. 17. A flow chart to show the process of extracting essential data from the header text files.

3.2.4 12-lead ECG Data Preprocessing

Once the header text files have been processed and cleaned, the next step is to implement a function that will take the raw ECG recordings and convert it into a usable format. Since the ECG recordings are from different sources, they come in various different signal lengths and sampling frequencies, therefore the ECG recordings need to be standardised in order to feed it through the neural network. An appropriate method for standardising the sampling frequency is to use a technique known as data resampling.

Data Resampling

Data resampling has been a common preprocessing technique when dealing with ECG data [53] as the ECG recordings can be sampled at different frequencies. The dataset that we are using compromises of ECG recordings from multiple different sources, meaning that our sampling frequencies will vary from different datasets. The data extracted from these datasets have a sampling frequency range between 257Hz - 1000Hz, and the length of the ECG leads range from 6 seconds - 30 minutes. This will require us to resample the frequency of the recordings to one common frequency. The datasets and their properties are listed in Table 3. An important note to make is that the total number of samples initially extracted was 41,101, however, as we are only using ECG samples that correspond the the 27 common classes, we will be using a total of 37,749 samples.

For our dataset, we have ECG recordings that have been sampled at 3 different frequencies: 257Hz, 500Hz, 1000Hz. In order to ensure that the data is as consistent as possible, the ECG recordings of 257Hz and 1000Hz are resampled to

500Hz. The resampling methods were implemented using a resample function from the scipy.signal library (open source python library for scientific computing).

Table 3. Information on each dataset being used for this project

Dataset	Number of Samples	Sampling Frequency (Hz)	Length of Recordings
CPSC	6877	500	6 seconds - 60 seconds
CPSC Extra	3453	500	6 seconds - 60 seconds
St. Petersburg	74	257	30 minutes
PTB	516	1000	30 seconds - 130 seconds
PTB-XL	21837	500	10 seconds
Georgia	10344	500	10 seconds

Data Transformation

The length of the ECG recordings vary from different sources; ranging from 6 seconds to 30 minutes. In order feed these recordings through the neural network, we need to make sure that the length of the data is the same. As the majority of the recordings last for 10 seconds, we have decided to make this our standard. Choosing a standard for the length of each recording means that the ECG recordings will have to undergo data truncation or zero padding. The process of data truncation is to clip the ECG recording to 10 seconds to make the recording shorter (if the recording is too long), and the process of zero padding is adding zeros to the end of the recording in order to make the recording longer (if the ECG recording is too short). The methods mentioned are said to have destructive effects on the information in the ECG [50] as we are removing information from the ECG recording. The data preparation process of the ECG recordings are shown in figure 18.

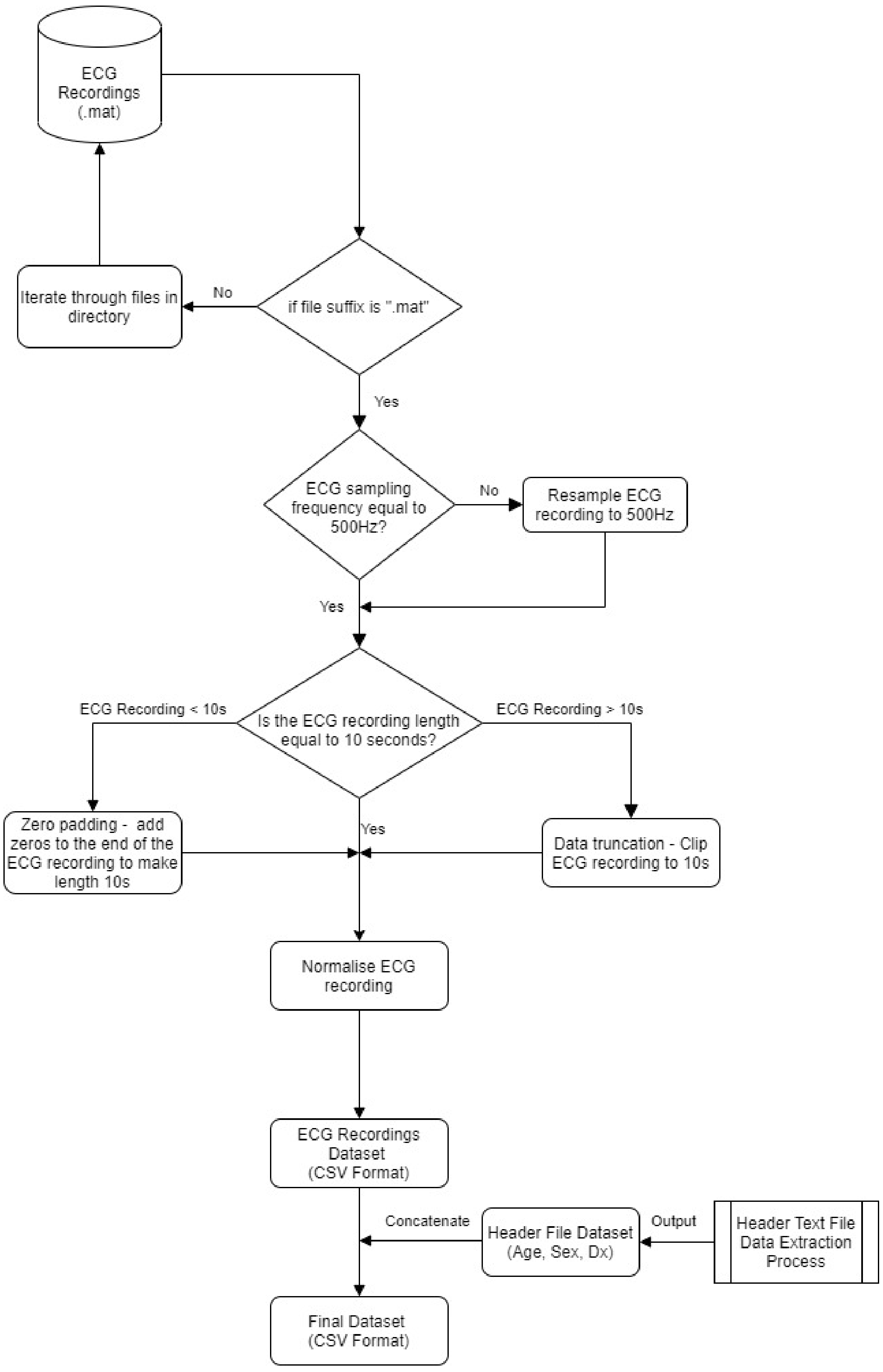

ECG Recordings (.mat)
if file suffix is ".mat"
No
Iterate through files in directory
Yes
ECG sampling frequency equal to 500Hz?
No
Resample ECG recording to 500Hz
Yes
ECG Recording < 10s
Is the ECG recording length equal to 10 seconds?
ECG Recording > 10s
Zero padding - add zeros to the end of the ECG recording to make length 10s
Yes
Data truncation - Clip ECG recording to 10s
Normalise ECG recording
ECG Recordings Dataset (CSV Format)
Concatenate
Header File Dataset (Age, Sex, Dx)
Output
Header Text File Data Extraction Process
Final Dataset (CSV Format)

Fig. 18. A flowchart of how the ECG recordings are prepared.

Data Resampling and Transformation Implementation (extract_leads()):

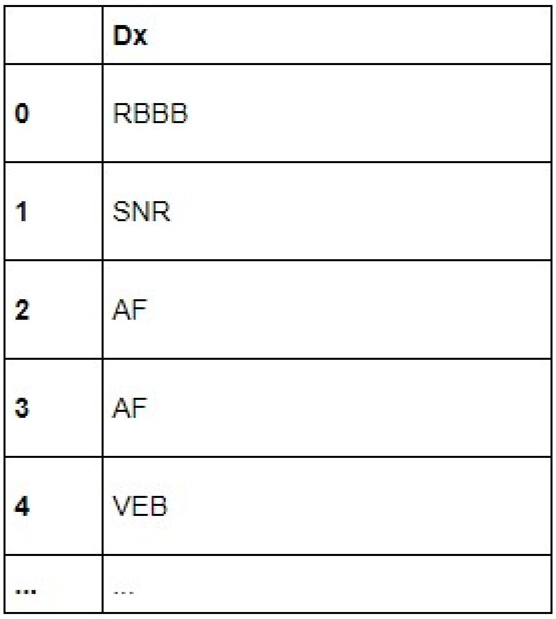
This function implements the flowchart shown in figure 18. The function is broken down into steps, which is outlined below:

1. The function loads each ECG recording and stores it into a multidimensional NumPy array ($arraysize = 12 \times L$, where L is length of ECG) using the NumPy library (a library for scientific computing).
2. The function uses the resample function from the SciPy library in order to resample the ECG to the desired sampling frequency.
3. Since the length of the ECG needs to be at a standard length, we need to implement data truncation and zero padding. The function will check whether the length of the ECG recording is equal to the desired length (10 seconds). If the length of the ECG recording is longer than 10 seconds, then it will be trimmed so that the first 10 seconds of the recording is retained. If the ECG recording is shorter than the desired length, then we add zeros to the end of the ECG until the length is equal to 10 seconds.
4. The function stores the ECG recordings along with its header data in one large dataframe for later use.

Implementing these steps in Python was done in one function, the code for this implementation is shown in the appendix (B.0.2).

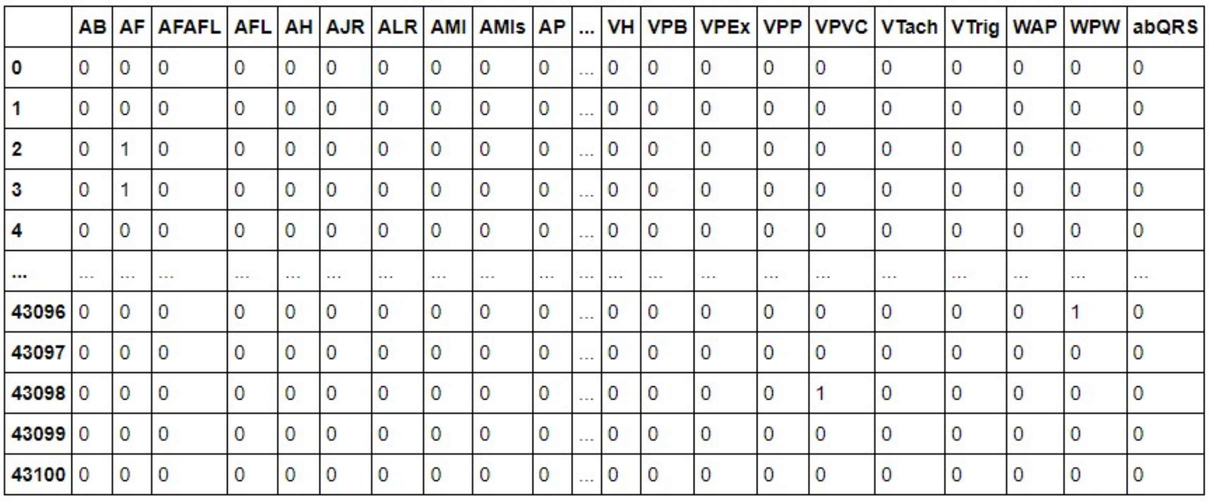
One Hot Encoding for Classes (Diagnosis)

Each patient has an associated cardiac arrhythmia (diagnosis) in the dataset. The diagnosis is referred to as classes (or known as the dependent variable), meaning that we are trying to predict the class using the ECG recordings (the independent variable). Each patient may have one or more cardiac arrhythmias, which means that we need to split all possible entries in the class column into its own column, as this is a multi-label classification problem. In order to do this, we can employ a method known as one-hot encoding. One-hot transforms a singular multi-class column into an binary integer representation, which allows us to compare the actual classes to the predicted classes from the neural network, with taking into account multi-labelled objects in the dataset. Using the pandas library, we can use the a function called ”get_dummies” in order to implement this. Figures 19 and 20 show the multi-class formats before an after the one-hot encoding has been applied.



	Dx
0	RBBB
1	SNR
2	AF
3	AF
4	VEB
...	...

Fig. 19. The classes (diagnosis) column before one-hot encoding.



	AB	AF	AFAFL	AFL	AH	AJR	ALR	AMI	AMIs	AP	...	VH	VPB	VPEx	VPP	VPVC	VTach	VTrig	WAP	WPW	abQRS
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
43096	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	1	0
43097	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
43098	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1	0	0	0	0	0
43099	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
43100	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Fig. 20. The one-hot encoded classes columns (27 columns for 27 unique classes).

3.2.5 ECG Recordings into Tensors

A tensor is a type of data structure that can hold mathematical representations of data with N dimensions [66]. This data structure is used to train neural networks as it can represent higher dimensional data. Currently the ECG recordings that we have are in the form of a nested dataframe, and in order to feed this data through a neural network, we will have to convert the data into a tensor (this is done by using the PyTorch library). Figure 21 shows the ECG recordings being converted from a dataframe to a tensor.

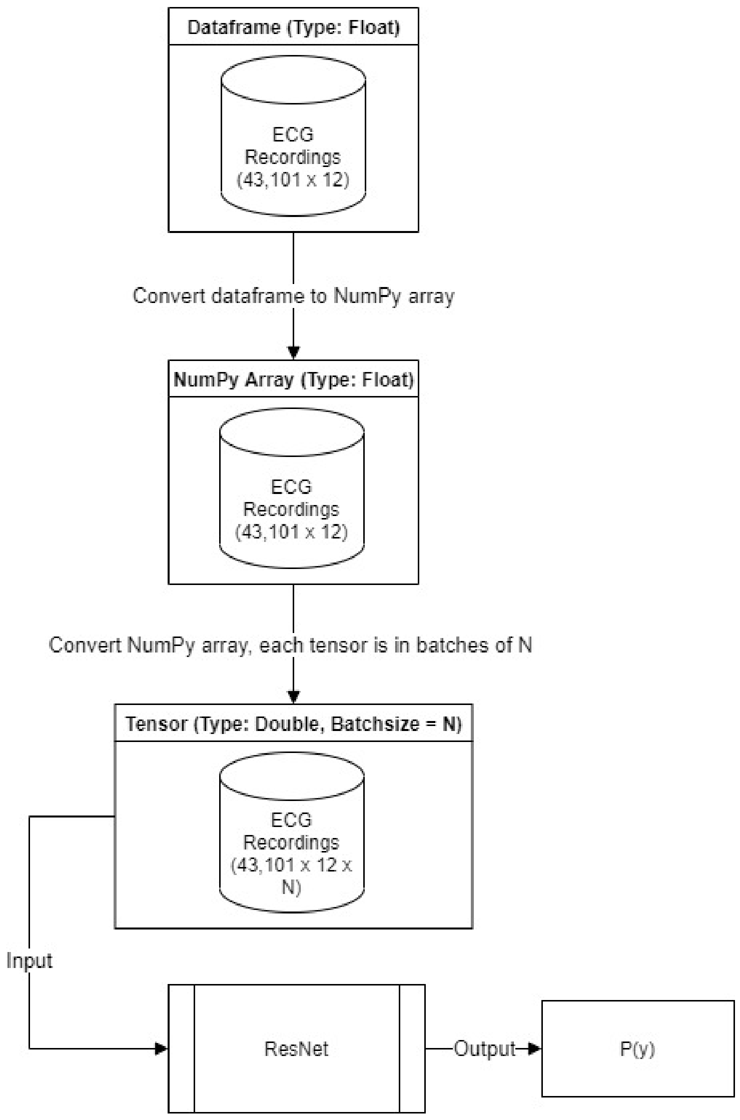

Dataframe (Type: Float)
ECG Recordings (43,101 x 12)
Convert dataframe to NumPy array
NumPy Array (Type: Float)
ECG Recordings (43,101 x 12)
Convert NumPy array, each tensor is in batches of N
Tensor (Type: Double, Batchsize = N)
ECG Recordings (43,101 x 12 x N)
Input
ResNet
Output
P(y)

Fig. 21. Converting ECG recordings into Tensors.

3.2.6 Sampling Technique

The dataset consists of 37,749 labelled samples, where I have chosen to use a hold-out sampling technique of an 60-20-20 train-validation-test split, meaning that 60% of the dataset will be used for training the neural network, 20% of the dataset is used as the validation set (the validation set is used to optimise the hyperparameters of the neural network), and the remaining 20% will be used for testing (implementation of this can be found in section B.0.1 in the appendix). Due to the large dataset, the hold-out sampling technique is an appropriate method as a large amount of data will result in the machine learning model being less prone to overfitting. The cross-validation (or leave-one-out cross validation, LOOCV) sampling techniques is commonly used as it reduces the model's chances of overfitting, however, this is computationally expensive when using a large dataset (making this an unfeasible option with limited hardware).

3.2.7 Residual Neural Network (ResNet) Architecture

Once the data preprocessing steps are completed, we move onto the developing the ResNet. The architecture of a neural network can be simplified into three main parts, the input layer, the hidden layer(s), and the output layer. When implementing a neural network, it is important to consider the type of functions that compose each individual layer, this includes the type of activation functions used in each layer, the type of layer, and the depth of the neural network.

Activation Function

Activation functions allow neural networks to extract patterns from a dataset. Various different types of activation functions have been covered in section 2.8 as they can be used for this project. Choosing the most appropriate activation function is very important when constructing a neural network as it tends to impact the performance of the neural network. The three activation functions covered in section 2.8 were the Sigmoid, ReLU, and Softmax functions. As discussed previously, the Sigmoid function suffers from the vanishing gradients problem. This is where the gradual increase in the absolute values of the inputs produce very small gradients. This means that the neural network will find it difficult to converge during training (if used in the hidden layers). In contrast, the ReLU activation function does not suffer from the vanishing gradients problem, this is due to the properties of the ReLU function. ReLU is a non-linear activation function, but since its output is a linear function of the input, the gradient flows well on the paths of the neurones [67].

Output Layer Activation Function

For the output layer of the neural network, we have a choice between the Sigmoid and Softmax function. The Softmax function is commonly used for multi-class classification problems as it produces a multinomial probability distribution, meaning that the output of the function is the probability of the input data belonging to each individual class. This is used along with a threshold value to assign all classes with a probability value higher than the threshold value to the input object, meaning that we are able to predict multiple classes. Whilst this is good for multi-class problems, it is not an appropriate output layer activation function for multi-label problems. The classes in the ECG dataset are not mutually exclusive, meaning that each ECG recording can be assigned multiple labels (as a patient can have more than one arrhythmia at the same time). Due to this criteria, we have opted to use a sigmoid function. Whilst it has been mentioned that the sigmoid activation function suffers from vanishing gradients, the sigmoid function will only be used as an output layer function to give the class labels.

Loss Function

Loss functions are a type of function that calculates the cost of predicted class in relation to the actual class (how far the predicted class is from the actual class). In order to improve the performance of a model, we use loss functions to make updates on hyperparameters of a model, in attempt to minimise the error of the model. As mentioned in section 2.5.2, we are using a loss function known as Binary Cross-Entropy with logits loss (BCEWithLogitsLoss). This function is a combination of the sigmoid activation function and Binary Cross-Entropy, which calculates the distance of the predicted class probabilities from the actual class. BCEWithLogitsLoss is used in this project as it can be used for multi-label classification problems.

Optimisation Algorithm

In order for the neural network to learn, it requires an optimisation algorithm that uses a loss function to optimise the weights of the neural network. The two main optimisation algorithms that can be used for this project is the adaptive moment estimation (Adam), and stochastic gradient descent (SGD) algorithms. Adam is a popular optimisation algorithm that takes advantage of two optimisation algorithms known as Adaptive Gradient (AdaGrad) algorithm, and the Root Mean Square Propagation (RMSProp). It utilises the per-parameter learning rates to mitigate the problem of vanishing gradients [34], meaning that the neural network will be able to converge. Adam is known to be a faster optimisation algorithm in comparison to SGD, however, some papers indicate that using SGD can improve the generalisation of a neural network [68], meaning that model is not overfitting. For this project, we will testing both optimisation algorithms to see which one will improve the performance of the ResNet.

ResNet Depth

The depth of the ResNet is important as increasing the number of layers can have advantages and disadvantages. Finding the optimal depth of the neural network is important as a very deep neural network can improve predictive power, however, it results in longer training times, and the possibility of overfitting the data.

In this project, we will be implementing a ResNet, where we will be experimenting with the depth of the ResNet. The two ResNets used to evaluate the performance when depth is increased (or decreased) is the ResNet18 and ResNet34 models. The ResNet18 is a residual neural network that consists of 18 hidden layers, whereas the ResNet34 consists of 34 hidden layers. The choice of functions for this project are determined by the type of data that is fed into the neural network. Each lead in the 12-lead ECG recording is a one dimensional signal, in which each data point in the signal gives us the amplitude of the signal, and the time at which that data point was recorded. As this is a one dimensional signal, we will be using one dimensional kernel functions in each layer of the neural network. The ResNet34 model consists of residual blocks, and the residual network stage. The ResNet starts with a 1-D convolutional kernel, followed by a 1-D batch normalisation layer (provides a normalised signal), a ReLU activation function, and a 1-D max pooling layer (used for extracting low level features). The residual block consists of a 1-D convolution kernel, with a kernel size of 7; a 1-D batch normalisation (BN) layer that normalises the signal for evaluation; a ReLU activation function; a dropout layer that reduces the ResNet from overfitting; another 1-D convolution kernel; a 1-D BN layer; an identity connection, which is used for skip connections; and a ReLU activation function. The residual block is depicted in figure 22, and an example of the overview architecture of the ResNet model is depicted in figure 23 (specifically the ResNet34 model, the ResNet18 model has a different hidden layer configuration). To see the in-depth architecture of the ResNet18 and ResNet34 model, check figures 26 and 27 in the appendix (A.0.1 and A.0.1).

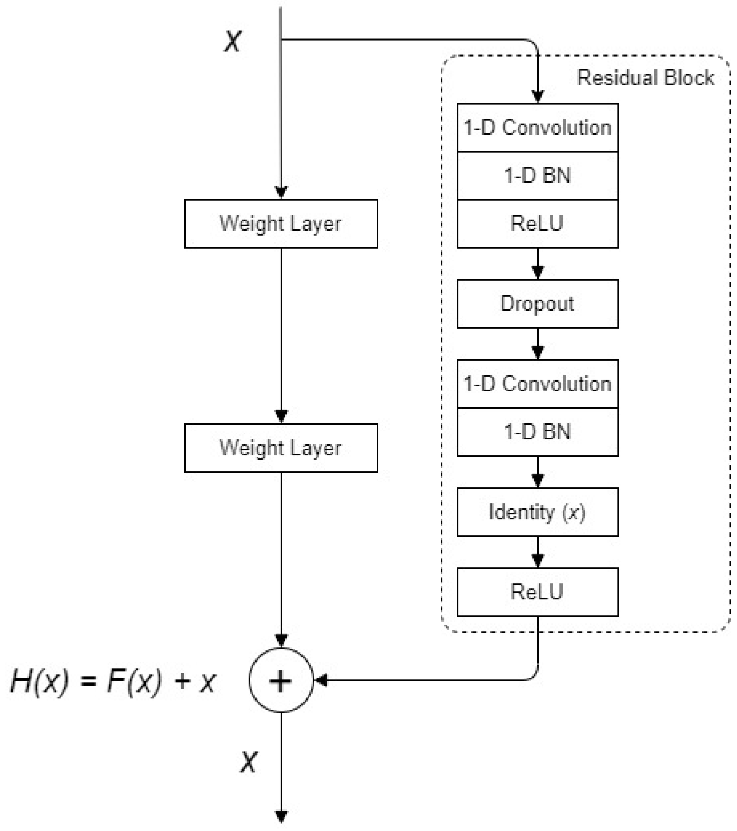


Fig. 22. The architecture of the residual blocks in the ResNet34 model.

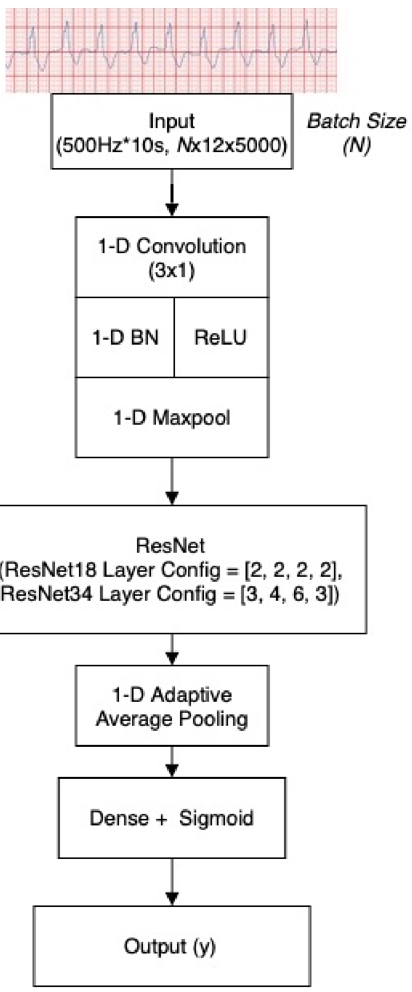


Fig. 23. The architecture of the ResNet34 model.

Implementation of the ResNet34 Architecture

Figures 22 and 27 visualise the architecture of the ResNet34 model. Implementing the ResNet34 architecture can be broken down into 3 main classes:

- The residual block (BasicBlock class).
- The bottleneck layer (Bottleneck class)
- The ResNet34 architecture (ResNet class)

The code for all the classes and functions used to implement the ResNet34 model is in the appendix (B.1).

4 Evaluation

Chapter 3 discussed the research methodology and implementation of the steps in this project. For this chapter, we will discuss the results produced by the implemented ResNet, providing an analysis of how each individual process has effected the results.

4.1 Evaluation Metrics

In order to assess how well a machine learning model performs, we need to have the appropriate evaluation metrics in place. In chapter 2.5.1, we discussed some potential evaluation metrics and loss functions that will show how well the model is performing. We discussed how the use of the accuracy metric is inadequate due to the severe class imbalance problem that this dataset suffers from. Therefore, we will be using precision, recall, f-measure, and the loss function (BCEWithLogitsLoss). F-measure is a popular evaluation metric that considers the performance of all classes individually, meaning that a poor classification result in some classes will be apparent in the results.

4.2 Experimental Results

The dataset was split into training, validation, and testing subsets. The training set is used to train the model, the validation set is used in training to tune the hyperparameters, and the testing set is an unseen set of data that is used to test the performance of the model, meaning that this set is unbiased. We will explore the performance of the model at all three stages of training, validation, and testing.

4.2.1 Results Comparison of Different Models

In order to evaluate how well the training process has been, we need to investigate the loss of the model. The aim of the ResNet is to minimise the loss when training, as it indicates that the model is improving. This loss function optimises the training process, allowing for small weight adjustments to be made. For this project, we have used the BCEWithLogitsLoss function. As mentioned in the methodology section, we will be testing both ResNet18 and ResNet34 architectures to see how the ResNet models perform when classifying ECGs, furthermore, we are comparing the performance of different optimisation algorithms (Adam and SGD) to see if one out performs the other. The performance of these results are evaluated using weighted f-measure values. The reason to use weighted f-measure values is to account for the severe class imbalance problem. Weighted f-measures are calculated by producing a support value that is based on the number of instances that belong to each class. This means that the results will account for some of the class imbalance. The results of these tests are shown in table 4.

Model No.	ResNet Architecture	Learning Rate (LR)	Optimiser	Training Weighted F-Measure	Training Loss	Validation Loss	Testing Weighted F-Measure
1	ResNet34	0.01	Adam	0.477	0.119	0.0013	0.419
2	ResNet18	0.1	SGD	0.471	0.184	0.0019	0.402
3	ResNet34	0.001	Adam	0.464	0.181	0.0014	0.438
4	ResNet34	0.1	Adam	0.463	0.203	0.0013	0.392
5	ResNet18	0.01	Adam	0.463	0.149	0.0014	0.376
6	ResNet18	0.001	Adam	0.462	0.157	0.0017	0.372
7	ResNet18	0.1	Adam	0.457	0.168	0.0019	0.413
8	ResNet34	0.01	SGD	0.448	0.192	0.0012	0.081
9	ResNet18	0.01	SGD	0.434	0.193	0.0027	0.349
10	ResNet34	0.1	SGD	0.405	0.186	0.0013	0.385
11	ResNet34	0.001	SGD	0.383	0.234	0.0014	0.382
12	ResNet18	0.001	SGD	0.371	0.242	0.0026	0.041

Table 4. ECG classification results using a combination of different architectures and optimisers (ordered by training weighted f-measure). (Additional Parameters: Adam - [weight_decay=1e^{-5}], SGD - [momentum=0.9]

When training these models, there is a large amount of data that the model will use for training, which meant that some hardware limitations were encountered that restricted the number of epochs that could be executed. Due to these limitations, we chose to use 5 epochs as this would allow the model to have multiple passes on the dataset, and complete training in a reasonable time period. The results that are presented shows the model in its highest performing state, meaning that we saved the best performing model (out of all the epochs) and evaluated the performance on the testing set.

As we can see in table 4, there is a large variation in the weighted f-measures. Upon looking at the f-measures, the results seem reasonable as there was a major class imbalance problem. Model 1 had performed the best, which utilised 34 layer architecture, in-conjunction with the Adam optimiser (lr=0.01). This model had a training and testing f-measure of 0.477 and 0.419 respectively (which was found on the 4_{th} epoch). Figure 24 shows the training loss as each training batch is passed through the network. When looking at the distribution of model performance in relation to which optimiser was used, Adam tended to outperform the SGD optimiser, apart from the one scenario where SGD (lr=0.1) combined with the ResNet18 architecture (model 2) was the second best performing model that was tested. Model 2 managed to produce a weighted f-measure of 0.471 and 0.402 for training and testing respectively (which was found on the 2_{nd} epoch). When looking worst performing models, we can see that the bottom two models (model 11 and 12) both utilised a SGD at a learning rate of 0.001. This evidently suggests that the learning rate was too small, which means that the error rate stays too high. This is evidently seen in the testing f-measure result for model 12, where it only managed to achieve 0.041. Model 3 utilised a ResNet34 architecture with the Adam optimiser (lr=0.001), which resulted in the best performing model on the testing set. with a weighted f-measure value of 0.438 (which was found on the 1_{st} epoch).

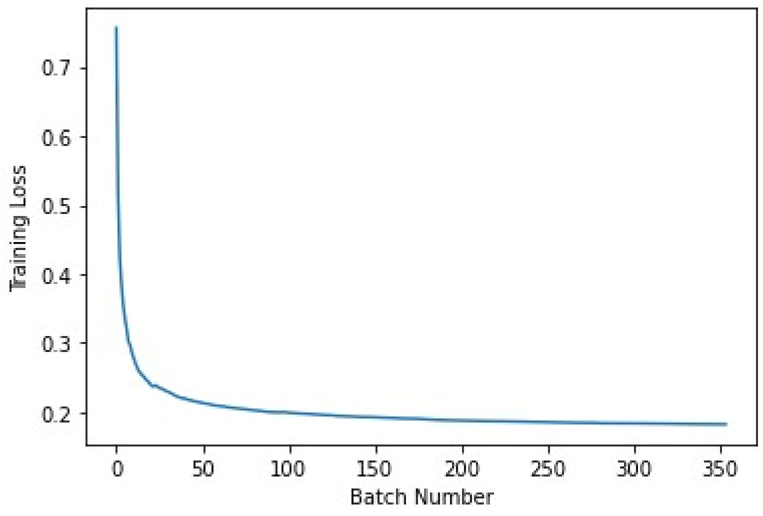
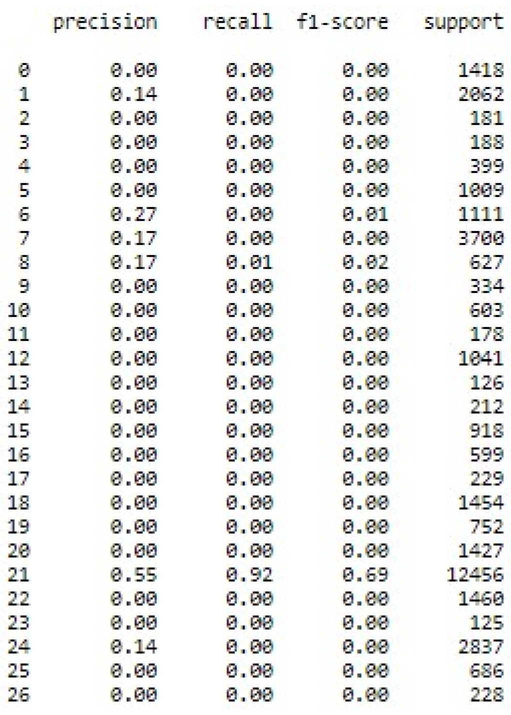


Fig. 24. A plot to show the improvement in loss as the ResNet34 model is fed each training batch. Config: ResNet34, BCEWithLogitsLoss, Adam Optimiser $(lr = 0.01, weight_decay = 1e^-5$

.

When observing the training and validation loss for all models, we can see that the training and validation loss is very similar across all models, where the lowest training loss was from model 1 (0.119), and the lowest validation loss was from model 8 (0.0012). When observing the loss results for all models, we can see that the training loss is higher than the validation loss by a considerable margin. Training and validation loss is generally similar, but large difference can indicate that the model is underfitting. Underfitting is when the model is unable to extract enough information from the training data, which means that it is unable to form relationships between the features and classes. This is evident as a large portion of the models have not performed well. Underfitting can be caused by class imbalance, or lack of training. In order to address this problem, running more epochs would allow the model to train for longer, which may improve the loss and f-measure results. In addition, data augmentation is a possibility that can reduce the class imbalance, which would allow the classifier to extract more information from classes that were originally underrepresented. This consideration was also mentioned by Jia et al., where they state that the reason that classification results are not very high is due to the lack of appropriate data [51], where lack of appropriate data references the class imbalance problem. Figure 25 shows the classification performance of model 12 on each individual class. As we can see, class 21 (sinus rhythm, SNR) is the majority of the the training data, which results in the model being able to make correct prediction for that class, but some classes with very few training samples are not predicted correctly as the model is not able to extract enough information from those leads.



```
      precision    recall  f1-score   support

 0       0.00      0.00      0.00      1418
 1       0.14      0.00      0.00      2062
 2       0.00      0.00      0.00       181
 3       0.00      0.00      0.00       188
 4       0.00      0.00      0.00       399
 5       0.00      0.00      0.00      1009
 6       0.27      0.00      0.01      1111
 7       0.17      0.00      0.00      3700
 8       0.17      0.01      0.02       627
 9       0.00      0.00      0.00       334
10       0.00      0.00      0.00       603
11       0.00      0.00      0.00       178
12       0.00      0.00      0.00      1041
13       0.00      0.00      0.00       126
14       0.00      0.00      0.00       212
15       0.00      0.00      0.00       918
16       0.00      0.00      0.00       599
17       0.00      0.00      0.00       229
18       0.00      0.00      0.00      1454
19       0.00      0.00      0.00       752
20       0.00      0.00      0.00      1427
21       0.55      0.92      0.69     12456
22       0.00      0.00      0.00      1460
23       0.00      0.00      0.00       125
24       0.14      0.00      0.00      2837
25       0.00      0.00      0.00       686
26       0.00      0.00      0.00       228
```

Fig. 25. The precision, recall, and f-measure results for model 12 (overall results in table 4).

4.3 Comparing Results from other Researchers to Ours

The PhysioNet/CinC Challenge 2020 had many researchers attempting to classify ECGs using machine learning. For that challenge, there was a strict set of hidden validation and testing sets that were used to rank the performance of every team's model. This project adopted a different approach where we used all of the available data to train and test my models, which means that the results will not be directly comparable, however, it will certainly give an indication as to how well the implemented models in this project had performed. Upon reviewing the results of other researchers, the best performing f-measure on a specific test set was 0.487, which was done by implementing an SE-ResNet. The same team achieved an f-measure of 0.683 on an offline test set (20% of whole dataset) [69]. Some other teams that utilised ResNet architectures achieved f-measure scores of 0.520 [70], and 0.462 [71] on a specified test set. Some possible improvements that could be made to boost the performance of this projects models are implementing some further data preprocessing to remove noise from the ECGs, and use the header file features (age, sex, and gender) to give the neural network some more information to find patterns in the dataset.

5 Summary

This chapter provides the conclusions to this project. Covering the experiments implemented, the achievements of the project, and the scope for future progression for this project.

5.1 Conclusions

The aim of this project was to produce a deep learning model that is able to classify cardiac arrhythmias using 12-lead ECGs. Upon mapping out the requirements for this project, it was apparent that using a powerful programming language for machine learning was necessary. Considering the available options for machine learning, it was decided that Python would be the programming language of choice, due to the abundance of data manipulation and machine learning libraries that are available. The main libraries that were used in this project were pandas and NumPy for data manipulation, sci-kit learn for the evaluation metrics, and PyTorch for developing a deep neural network. In order to successfully implement a deep learning model, we needed to consider all data preprocessing steps that would prepare the ECGs for training. Upon doing research, an interesting model known as a residual neural network was discovered that seemed to produce strong predictions on image and signal datasets. This influenced the decision to implement a residual neural network to make ECG classifications.

During the course of the ResNet development, there were challenges that needed to be faced, as the initial implementation produced very poor classification results. In order to overcome this issue, a large portion of the implementation's time frame was designated to experimenting with different ResNet architectures, and finding optimal hyperparameters that will maximise the performance of the model. A ResNet34 model that was implemented managed to achieve a training and testing weighted f-measure of 0.477 and 0.419 respectively. The results from the model performed well on classes that were well represented in the dataset, however, it performed poorly underrepresented classes in the dataset. Future work is considered in order to boost the performance of the ResNet models implemented, which will be discussed in the final section.

5.2 Future Work

In this project, we implemented a residual neural network that is able to classify 12-lead ECGs to a certain extent. The great efforts that have been put into this project have resulted in the development of a residual neural network that is able to classify ECGs with respectable results, however, there is scope for further improvements that would help improve some aspects of the project. These considerations are going to be discussed.

1. During the initial stages of the project, data augmentation techniques have been discussed in order to boost the performance of the ResNet on underrepresented classes. The project workflow initially started out to generate synthetic data using generative adversarial networks. An issue discussed was the lack of training data for the underrepresented classes. Even though there would have been an issue of generating realistic ECG signals that replicate a specific cardiac arrhythmia, there is more scope to assess the quality of these generated synthetic ECGs.

2. Using the dynamic model proposed my McSharry and Clifford, we can focus on augmenting data to improve ECG classification, presenting a feasible way of utilising the dynamic model in a larger scale (due to the complexities of generating 12-lead synthetic ECG data for 27 different classes).

3. The results produced by the ResNet models have scope for improvement.

Using the knowledge gained from this project, we can further develop the model by implementing ensemble classifiers that use hand crafted features from the dataset to improve the results produced.

4. As with many problems that are solved with deep learning, the classifications made are difficult to reverse engineer in order to explain the exact characteristics that correspond to a classification. Further research can allow for improved explainability (the ability to explain the output of a machine learning model to a human) in relation to the specific morphology of an ECG that results in the corresponding classifications.

References

[1] D. A. Cook, S.-Y. Oh, and M. V. Pusic, "Accuracy of physicians' electrocardiogram interpretations a systematic review and meta-analysis," *JAMA*, vol. 180, no. 11, pp. 1461–1471, 2020.

[2] P. Santos, P. Pessanha, M. Viana, M. Campelo, J. Nunes, A. Hespanhol, F. Macedo, and L. Couto, "Accuracy of general practitioners' readings of ecg in primary care," *Open Medicine*, vol. 9, no. 3, pp. 431–436, 2014.

[3] S. M. Mathews, C. Kambhamettu, and K. E. Barner, "A novel application of deep learning for single-lead ecg classification," *Computers in biology and medicine*, vol. 99, pp. 53–62, 2018.

[4] E. H. Houssein, M. Kilany, and A. E. Hassanien, "Ecg signals classification: A review," *International Journal of Intelligent Engineering Informatics*, vol. 5, no. 4, pp. 376–396, 2017.

[5] P. De Chazal, M. O'Dwyer, and R. B. Reilly, "Automatic classification of heartbeats using ecg morphology and heartbeat interval features," *IEEE transactions on biomedical engineering*, vol. 51, no. 7, pp. 1196–1206, 2004.

[6] J. Hampton and J. Hampton, *The ECG made easy e-book*. Elsevier Health Sciences, 2019.

[7] A. H. Ribeiro, M. H. Ribeiro, G. M. Paixão, D. M. Oliveira, P. R. Gomes, J. A. Canazart, M. P. Ferreira, C. R. Andersson, P. W. Macfarlane, W. Meira Jr, *et al.*, "Automatic diagnosis of the 12-lead ecg using a deep neural network," *Nature communications*, vol. 11, no. 1, pp. 1–9, 2020.

[8] H. Gholam-Hosseini and H. Nazeran, "Detection and extraction of the ecg signal parameters," in *Proceedings of the 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society. Vol. 20 Biomedical Engineering Towards the Year 2000 and Beyond (Cat. No. 98CH36286)*, IEEE, 1998, pp. 127–130.

[9] F. Monitillo, M. Leone, C. Rizzo, A. Passantino, and M. Iacoviello, "Ventricular repolarization measures for arrhythmic risk stratification," *World journal of cardiology*, vol. 8, no. 1, p. 57, 2016.

[10] T. Hlaing, T. DiMino, P. R. Kowey, and G.-X. Yan, "Ecg repolarization waves: Their genesis and clinical implications," *Annals of noninvasive electrocardiology*, vol. 10, no. 2, pp. 211–223, 2005.

[11] H. P. Adams, G. del Zoppo, M. J. Alberts, D. L. Bhatt, L. Brass, A. Furlan, R. L. Grubb, R. T. Higashida, E. C. Jauch, C. Kidwell, P. D. Lyden, L. B. Morgenstern, A. I. Qureshi, R. H. Rosenwasser, P. A. Scott, and E. F. Wijdicks, "Guidelines for the early management of adults with ischemic stroke," *Stroke*, vol. 38, no. 5, pp. 1655–1711, 2007. DOI: 10.1161/STROKEAHA.107.181486. [Online]. Available: https://www.ahajournals.org/doi/abs/10.1161/STROKEAHA.107.181486.

[12] J. Jaumot, C. Bedia, and R. Tauler, *Data Analysis for Omic Sciences: Methods and Applications*. Elsevier, 2018.

[13] S. García, J. Luengo, and F. Herrera, *Data preprocessing in data mining*. Springer, 2015, vol. 72.

[14] S. Marsland, *Machine learning: an algorithmic perspective*. CRC press, 2015.

[15] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," 1988.

[16] B. J. Copeland, *The essential turing*. Clarendon Press, 2004.

[17] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[18] L. De Mol, "Turing machines," 2018.

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[20] S. Celin and K. Vasanth, "Ecg signal classification using various machine learning techniques," *Journal of medical systems*, vol. 42, no. 12, pp. 1–11, 2018.

[21] J. M. Johnson and T. M. Khoshgoftaar, "Survey on deep learning with class imbalance," *Journal of Big Data*, vol. 6, no. 1, pp. 1–54, 2019.

[22] D. M. Powers, "Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[23] C. Ferri, J. Hernández-Orallo, and R. Modroiu, "An experimental comparison of performance measures for classification," *Pattern Recognition Letters*, vol. 30, no. 1, pp. 27–38, 2009.

[24] S.-C. Wang, "Artificial neural network," in *Interdisciplinary computing in java programming*, Springer, 2003, pp. 81–100.

[25] S. Chakraverty, D. M. Sahoo, and N. R. Mahato, "Mcculloch–pitts neural network model," in *Concepts of Soft Computing*, Springer, 2019, pp. 167–173.

[26] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[27] Y. Hayashi, M. Sakata, and S. I. Gallant, "Multi-layer versus single-layer neural networks and an application to reading hand-stamped characters," in *International Neural Network Conference*, Springer, 1990, pp. 781–784.

[28] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: A tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, 1996.

[29] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[30] P. Goyal, S. Pandey, and K. Jain, "Introduction to natural language processing and deep learning," in *Deep Learning for Natural Language Processing*, Springer, 2018, pp. 1–74.

[31] L. Bottou, "Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*, Springer, 2012, pp. 421–436.

[32] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[33] G. Orr, N. Schraudolph, and F. Cummins, *Momentum and learning rate adaptation*, 1999. [Online]. Available: `https://www.willamette.edu/~gorr/classes/cs449/momrate.html#top`.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[35] H. Leung and S. Haykin, "The complex backpropagation algorithm," *IEEE Transactions on signal processing*, vol. 39, no. 9, pp. 2101–2104, 1991.

[36] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378*, 2018.

[37] S. Sharma, *Activation functions in neural networks*, Jul. 2021. [Online]. Available: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

[38] I. Kandel and M. Castelli, "Transfer learning with convolutional neural networks for diabetic retinopathy image classification. a review," *Applied Sciences*, vol. 10, no. 6, p. 2021, 2020.

[39] J. Kolbusz, P. Rozycki, and B. M. Wilamowski, "The study of architecture mlp with linear neurons in order to eliminate the "vanishing gradient" problem," in *International Conference on Artificial Intelligence and Soft Computing*, Springer, 2017, pp. 97–106.

[40] D. Wu, Y. Wang, S.-T. Xia, J. Bailey, and X. Ma, "Skip connections matter: On the transferability of adversarial examples generated with resnets," *arXiv preprint arXiv:2002.05990*, 2020.

[41] J. Fan, C. Sun, C. Chen, X. Jiang, X. Liu, X. Zhao, L. Meng, C. Dai, and W. Chen, "Eeg data augmentation: Towards class imbalance problem in sleep staging tasks," *Journal of Neural Engineering*, vol. 17, no. 5, p. 056 017, 2020.

[42] X. Jiang and Z. Ge, "Data augmentation classifier for imbalanced fault classification," *IEEE Transactions on Automation Science and Engineering*, 2020.

[43] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, pp. 249–259, 2018.

[44] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, “Generative adversarial networks: An overview,” *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[45] T. Golany, G. Lavee, S. T. Yarden, and K. Radinsky, “Improving ecg classification using generative adversarial networks,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 13 280–13 285.

[46] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[47] P. E. McSharry, G. D. Clifford, L. Tarassenko, and L. A. Smith, “A dynamical model for generating synthetic electrocardiogram signals,” *IEEE transactions on biomedical engineering*, vol. 50, no. 3, pp. 289–294, 2003.

[48] M. Ibnkahla, “Applications of neural networks to digital communications–a survey,” *Signal processing*, vol. 80, no. 7, pp. 1185–1215, 2000.

[49] M. S. Hanif and M. Bilal, “Competitive residual neural network for image classification,” *ICT Express*, vol. 6, no. 1, pp. 28–37, 2020.

[50] S. Yang, H. Xiang, Q. Kong, and C. Wang, “Multi-label classification of electrocardiogram with modified residual networks,” in *2020 Computing in Cardiology*, IEEE, 2020, pp. 1–4.

[51] W. Jia, X. Xu, X. Xu, Y. Sun, and X. Liu, “Automatic detection and classification of 12-lead ecgs using a deep neural network,” in *2020 Computing in Cardiology*, IEEE, 2020, pp. 1–4.

[52] Z. Li, D. Zhou, L. Wan, J. Li, and W. Mou, “Heartbeat classification using deep residual convolutional neural network from 2-lead electrocardiogram,” *Journal of electrocardiology*, vol. 58, pp. 105–112, 2020.

[53] N. Nonaka and J. Seita, "Data augmentation for electrocardiogram classification with deep neural network," *arXiv preprint arXiv:2009.04398*, 2020.

[54] S. Kiranyaz, T. Ince, R. Hamila, and M. Gabbouj, "Convolutional neural networks for patient-specific ecg classification," in *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, IEEE, 2015, pp. 2608–2611.

[55] B. Pyakillya, N. Kazachenko, and N. Mikhailovsky, "Deep learning for ecg classification," in *Journal of physics: conference series*, IOP Publishing, vol. 913, 2017, p. 012 004.

[56] Y. Zhou, H. Zhang, Y. Li, and G. Ning, "Ecg heartbeat classification based on resnet and bi-lstm," in *IOP Conference Series: Earth and Environmental Science*, IOP Publishing, vol. 428, 2020, p. 012 014.

[57] E. A. P. Alday, A. Gu, A. J. Shah, C. Robichaux, A.-K. I. Wong, C. Liu, F. Liu, A. B. Rad, A. Elola, S. Seyedi, *et al.*, "Classification of 12-lead ecgs: The physionet/computing in cardiology challenge 2020," *Physiological measurement*, vol. 41, no. 12, p. 124 003, 2020.

[58] *What is data transformation: Definition, benefits, and uses: Stitch resource.* [Online]. Available: `https://www.stitchdata.com/resources/data-transformation/`.

[59] [Online]. Available: `https://pytorch.org/`.

[60] *Numpy.* [Online]. Available: `https://numpy.org/`.

[61] [Online]. Available: `https://git-scm.com/`.

[62] *Cuda zone*, Jul. 2021. [Online]. Available: `https://developer.nvidia.com/cuda-zone`.

[63] X. Fan, Q. Yao, Y. Cai, F. Miao, F. Sun, and Y. Li, “Multiscaled fusion of deep convolutional neural networks for screening atrial fibrillation from single lead short ecg recordings,” *IEEE journal of biomedical and health informatics*, vol. 22, no. 6, pp. 1744–1753, 2018.

[64] V. Sujadevi, K. Soman, and R. Vinayakumar, “Real-time detection of atrial fibrillation from short time single lead ecg traces using recurrent neural networks,” in *The International Symposium on Intelligent Systems Technologies and Applications*, Springer, 2017, pp. 212–221.

[65] D. Zhang, S. Yang, X. Yuan, and P. Zhang, “Interpretable deep learning for automatic diagnosis of 12-lead electrocardiogram,” *Iscience*, vol. 24, no. 4, p. 102 373, 2021.

[66] P. Etingof, S. Gelaki, D. Nikshych, and V. Ostrik, *Tensor categories*. American Mathematical Soc., 2016, vol. 205.

[67] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[68] N. S. Keskar and R. Socher, “Improving generalization performance by switching from adam to sgd,” *arXiv preprint arXiv:1712.07628*, 2017.

[69] Z. Zhu, H. Wang, T. Zhao, Y. Guo, Z. Xu, Z. Liu, S. Liu, X. Lan, X. Sun, and M. Feng, “Classification of cardiac abnormalities from ecg signals using se-resnet,” in *2020 Computing in Cardiology*, IEEE, 2020, pp. 1–4.

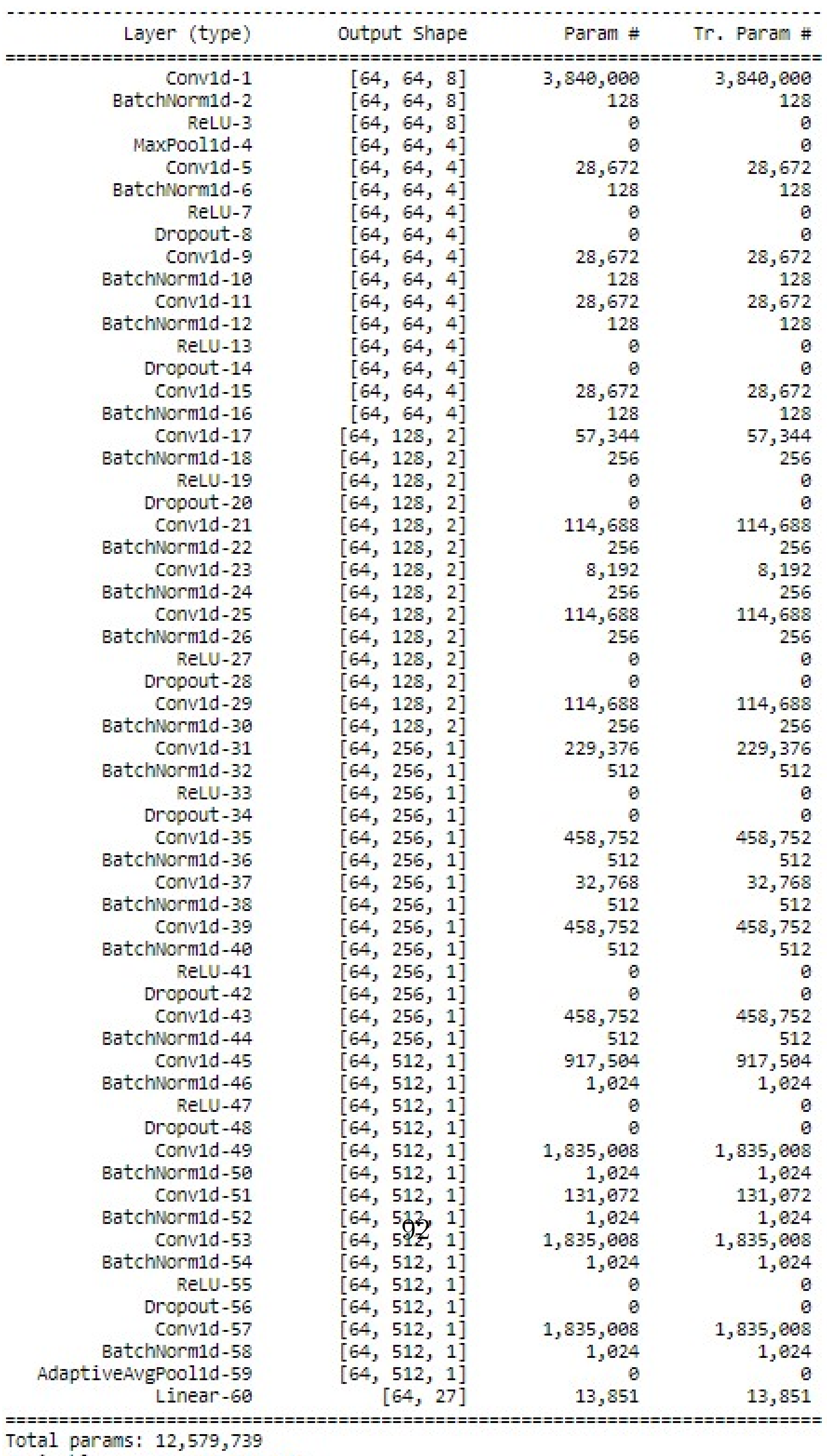
[70] Z. Zhao, H. Fang, S. D. Relton, R. Yan, Y. Liu, Z. Li, J. Qin, and D. C. Wong, “Adaptive lead weighted resnet trained with different duration signals for classifying 12-lead ecgs,” in *2020 Computing in Cardiology*, IEEE, 2020, pp. 1–4.

[71] M. P. Oppelt, M. Riehl, F. P. Kemeth, and J. Steffan, "Combining scatter transform and deep neural networks for multilabel electrocardiogram signal classification," in *2020 Computing in Cardiology*, IEEE, 2020, pp. 1–4.

Appendices

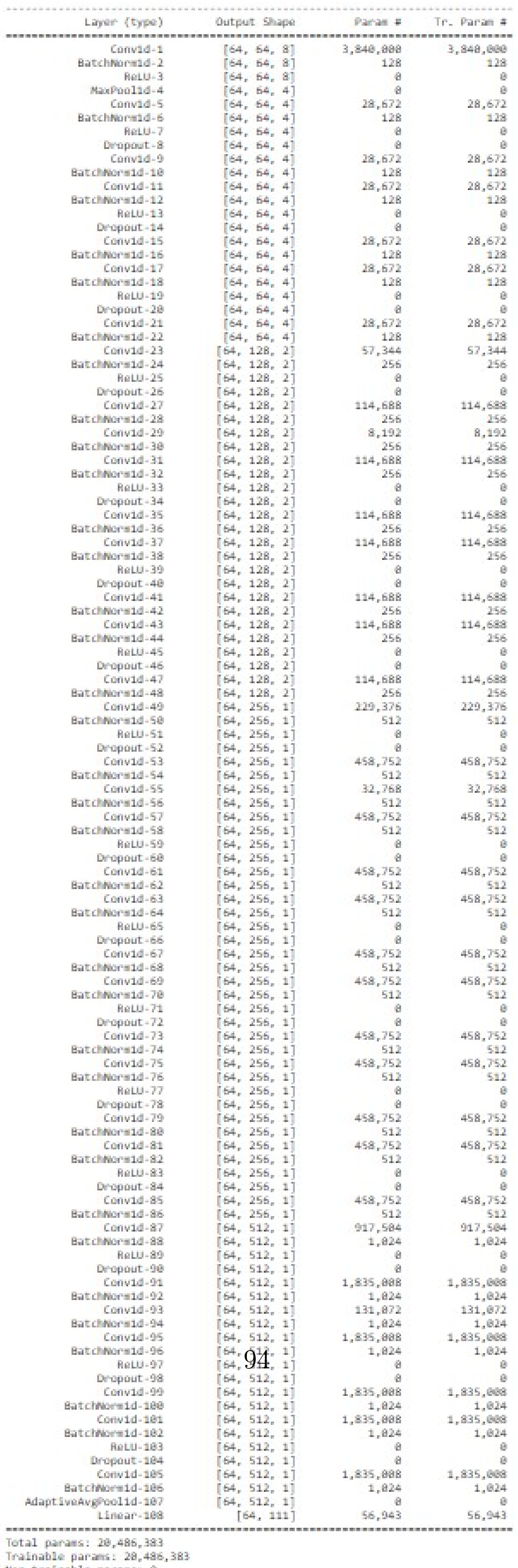
A Figures

A.0.1 In-depth ResNet18 Architecture



```
--------------------------------------------------------------------------------
            Layer (type)        Output Shape         Param #     Tr. Param #
================================================================================
               Conv1d-1         [64, 64, 8]       3,840,000       3,840,000
          BatchNorm1d-2         [64, 64, 8]             128             128
                 ReLU-3         [64, 64, 8]               0               0
            MaxPool1d-4         [64, 64, 4]               0               0
               Conv1d-5         [64, 64, 4]          28,672          28,672
          BatchNorm1d-6         [64, 64, 4]             128             128
                 ReLU-7         [64, 64, 4]               0               0
              Dropout-8         [64, 64, 4]               0               0
               Conv1d-9         [64, 64, 4]          28,672          28,672
         BatchNorm1d-10         [64, 64, 4]             128             128
              Conv1d-11         [64, 64, 4]          28,672          28,672
         BatchNorm1d-12         [64, 64, 4]             128             128
                ReLU-13         [64, 64, 4]               0               0
             Dropout-14         [64, 64, 4]               0               0
              Conv1d-15         [64, 64, 4]          28,672          28,672
         BatchNorm1d-16         [64, 64, 4]             128             128
              Conv1d-17        [64, 128, 2]          57,344          57,344
         BatchNorm1d-18        [64, 128, 2]             256             256
                ReLU-19        [64, 128, 2]               0               0
             Dropout-20        [64, 128, 2]               0               0
              Conv1d-21        [64, 128, 2]         114,688         114,688
         BatchNorm1d-22        [64, 128, 2]             256             256
              Conv1d-23        [64, 128, 2]           8,192           8,192
         BatchNorm1d-24        [64, 128, 2]             256             256
              Conv1d-25        [64, 128, 2]         114,688         114,688
         BatchNorm1d-26        [64, 128, 2]             256             256
                ReLU-27        [64, 128, 2]               0               0
             Dropout-28        [64, 128, 2]               0               0
              Conv1d-29        [64, 128, 2]         114,688         114,688
         BatchNorm1d-30        [64, 128, 2]             256             256
              Conv1d-31        [64, 256, 1]         229,376         229,376
         BatchNorm1d-32        [64, 256, 1]             512             512
                ReLU-33        [64, 256, 1]               0               0
             Dropout-34        [64, 256, 1]               0               0
              Conv1d-35        [64, 256, 1]         458,752         458,752
         BatchNorm1d-36        [64, 256, 1]             512             512
              Conv1d-37        [64, 256, 1]          32,768          32,768
         BatchNorm1d-38        [64, 256, 1]             512             512
              Conv1d-39        [64, 256, 1]         458,752         458,752
         BatchNorm1d-40        [64, 256, 1]             512             512
                ReLU-41        [64, 256, 1]               0               0
             Dropout-42        [64, 256, 1]               0               0
              Conv1d-43        [64, 256, 1]         458,752         458,752
         BatchNorm1d-44        [64, 256, 1]             512             512
              Conv1d-45        [64, 512, 1]         917,504         917,504
         BatchNorm1d-46        [64, 512, 1]           1,024           1,024
                ReLU-47        [64, 512, 1]               0               0
             Dropout-48        [64, 512, 1]               0               0
              Conv1d-49        [64, 512, 1]       1,835,008       1,835,008
         BatchNorm1d-50        [64, 512, 1]           1,024           1,024
              Conv1d-51        [64, 512, 1]         131,072         131,072
         BatchNorm1d-52        [64, 512, 1]           1,024           1,024
              Conv1d-53        [64, 512, 1]       1,835,008       1,835,008
         BatchNorm1d-54        [64, 512, 1]           1,024           1,024
                ReLU-55        [64, 512, 1]               0               0
             Dropout-56        [64, 512, 1]               0               0
              Conv1d-57        [64, 512, 1]       1,835,008       1,835,008
         BatchNorm1d-58        [64, 512, 1]           1,024           1,024
   AdaptiveAvgPool1d-59        [64, 512, 1]               0               0
              Linear-60            [64, 27]          13,851          13,851
================================================================================
Total params: 12,579,739
```

A.0.2 In-depth ResNet34 Architecture



```
-----------------------------------------------------------------------------
             Layer (type)        Output Shape         Param #     Tr. Param #
=============================================================================
                 Conv1d-1         [64, 64, 8]       3,840,000       3,840,000
            BatchNorm1d-2         [64, 64, 8]             128             128
                   ReLU-3         [64, 64, 8]               0               0
              MaxPool1d-4         [64, 64, 4]               0               0
                 Conv1d-5         [64, 64, 4]          28,672          28,672
            BatchNorm1d-6         [64, 64, 4]             128             128
                   ReLU-7         [64, 64, 4]               0               0
                Dropout-8         [64, 64, 4]               0               0
                 Conv1d-9         [64, 64, 4]          28,672          28,672
           BatchNorm1d-10         [64, 64, 4]             128             128
                Conv1d-11         [64, 64, 4]          28,672          28,672
           BatchNorm1d-12         [64, 64, 4]             128             128
                  ReLU-13         [64, 64, 4]               0               0
               Dropout-14         [64, 64, 4]               0               0
                Conv1d-15         [64, 64, 4]          28,672          28,672
           BatchNorm1d-16         [64, 64, 4]             128             128
                Conv1d-17         [64, 64, 4]          28,672          28,672
           BatchNorm1d-18         [64, 64, 4]             128             128
                  ReLU-19         [64, 64, 4]               0               0
               Dropout-20         [64, 64, 4]               0               0
                Conv1d-21         [64, 64, 4]          28,672          28,672
           BatchNorm1d-22         [64, 64, 4]             128             128
                Conv1d-23        [64, 128, 2]          57,344          57,344
           BatchNorm1d-24        [64, 128, 2]             256             256
                  ReLU-25        [64, 128, 2]               0               0
               Dropout-26        [64, 128, 2]               0               0
                Conv1d-27        [64, 128, 2]         114,688         114,688
           BatchNorm1d-28        [64, 128, 2]             256             256
                Conv1d-29        [64, 128, 2]           8,192           8,192
           BatchNorm1d-30        [64, 128, 2]             256             256
                Conv1d-31        [64, 128, 2]         114,688         114,688
           BatchNorm1d-32        [64, 128, 2]             256             256
                  ReLU-33        [64, 128, 2]               0               0
               Dropout-34        [64, 128, 2]               0               0
                Conv1d-35        [64, 128, 2]         114,688         114,688
           BatchNorm1d-36        [64, 128, 2]             256             256
                Conv1d-37        [64, 128, 2]         114,688         114,688
           BatchNorm1d-38        [64, 128, 2]             256             256
                  ReLU-39        [64, 128, 2]               0               0
               Dropout-40        [64, 128, 2]               0               0
                Conv1d-41        [64, 128, 2]         114,688         114,688
           BatchNorm1d-42        [64, 128, 2]             256             256
                Conv1d-43        [64, 128, 2]         114,688         114,688
           BatchNorm1d-44        [64, 128, 2]             256             256
                  ReLU-45        [64, 128, 2]               0               0
               Dropout-46        [64, 128, 2]               0               0
                Conv1d-47        [64, 128, 2]         114,688         114,688
           BatchNorm1d-48        [64, 128, 2]             256             256
                Conv1d-49        [64, 256, 1]         229,376         229,376
           BatchNorm1d-50        [64, 256, 1]             512             512
                  ReLU-51        [64, 256, 1]               0               0
               Dropout-52        [64, 256, 1]               0               0
                Conv1d-53        [64, 256, 1]         458,752         458,752
           BatchNorm1d-54        [64, 256, 1]             512             512
                Conv1d-55        [64, 256, 1]          32,768          32,768
           BatchNorm1d-56        [64, 256, 1]             512             512
                Conv1d-57        [64, 256, 1]         458,752         458,752
           BatchNorm1d-58        [64, 256, 1]             512             512
                  ReLU-59        [64, 256, 1]               0               0
               Dropout-60        [64, 256, 1]               0               0
                Conv1d-61        [64, 256, 1]         458,752         458,752
           BatchNorm1d-62        [64, 256, 1]             512             512
                Conv1d-63        [64, 256, 1]         458,752         458,752
           BatchNorm1d-64        [64, 256, 1]             512             512
                  ReLU-65        [64, 256, 1]               0               0
               Dropout-66        [64, 256, 1]               0               0
                Conv1d-67        [64, 256, 1]         458,752         458,752
           BatchNorm1d-68        [64, 256, 1]             512             512
                Conv1d-69        [64, 256, 1]         458,752         458,752
           BatchNorm1d-70        [64, 256, 1]             512             512
                  ReLU-71        [64, 256, 1]               0               0
               Dropout-72        [64, 256, 1]               0               0
                Conv1d-73        [64, 256, 1]         458,752         458,752
           BatchNorm1d-74        [64, 256, 1]             512             512
                Conv1d-75        [64, 256, 1]         458,752         458,752
           BatchNorm1d-76        [64, 256, 1]             512             512
                  ReLU-77        [64, 256, 1]               0               0
               Dropout-78        [64, 256, 1]               0               0
                Conv1d-79        [64, 256, 1]         458,752         458,752
           BatchNorm1d-80        [64, 256, 1]             512             512
                Conv1d-81        [64, 256, 1]         458,752         458,752
           BatchNorm1d-82        [64, 256, 1]             512             512
                  ReLU-83        [64, 256, 1]               0               0
               Dropout-84        [64, 256, 1]               0               0
                Conv1d-85        [64, 256, 1]         458,752         458,752
           BatchNorm1d-86        [64, 256, 1]             512             512
                Conv1d-87        [64, 512, 1]         917,504         917,504
           BatchNorm1d-88        [64, 512, 1]           1,024           1,024
                  ReLU-89        [64, 512, 1]               0               0
               Dropout-90        [64, 512, 1]               0               0
                Conv1d-91        [64, 512, 1]       1,835,008       1,835,008
           BatchNorm1d-92        [64, 512, 1]           1,024           1,024
                Conv1d-93        [64, 512, 1]         131,072         131,072
           BatchNorm1d-94        [64, 512, 1]           1,024           1,024
                Conv1d-95        [64, 512, 1]       1,835,008       1,835,008
           BatchNorm1d-96        [64, 512, 1]           1,024           1,024
                  ReLU-97        [64, 512, 1]               0               0
               Dropout-98        [64, 512, 1]               0               0
                Conv1d-99        [64, 512, 1]       1,835,008       1,835,008
          BatchNorm1d-100        [64, 512, 1]           1,024           1,024
               Conv1d-101        [64, 512, 1]       1,835,008       1,835,008
          BatchNorm1d-102        [64, 512, 1]           1,024           1,024
                 ReLU-103        [64, 512, 1]               0               0
              Dropout-104        [64, 512, 1]               0               0
               Conv1d-105        [64, 512, 1]       1,835,008       1,835,008
          BatchNorm1d-106        [64, 512, 1]           1,024           1,024
    AdaptiveAvgPool1d-107        [64, 512, 1]               0               0
               Linear-108           [64, 111]          56,943          56,943
=============================================================================
Total params: 20,486,383
Trainable params: 20,486,383
```

B Code

B.0.1 Holdout Sampling Technique

```
def dataloader(url):
    print("Loading Data")
    dataset = pd.read_pickle(url)
    labels = dataset.iloc[:, 16:]
    dataset = dataset.iloc[:, 4:16]

    _, weightindex = weight_metric()
    labels = labels[weightindex].to_numpy() #[:27]
    del weightindex
    gc.collect()


    print("Test Train Split 1")
    X_train, X_test, y_train, y_test = train_test_split(dataset, labels
    , test_size=0.2, random_state=42)
    del dataset
    del labels


    # train-validation-test split = 60/20/20
    print("Test Train Split 2")
    X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
     test_size=0.25, random_state=42)

    print("Loading Leads")
    X_train = lead_loader(X_train)
    X_val = lead_loader(X_val)
    X_test = lead_loader(X_test)

    print("Delete Missing Rows")
    X_train, y_train = delete_missing_rows(X_train, y_train)
    X_val, y_val = delete_missing_rows(X_val, y_val)
    X_test, y_test = delete_missing_rows(X_test, y_test)

    gc.collect()

    return X_train, X_test, y_train, y_test, X_val, y_val

# Convert ECG leads from Dataframe to NumPy array (used to convert to
   tensor later)
def lead_loader(dataset):

    dataset = dataset.to_numpy()
    print("Dataset shape: ", dataset.shape)
    testingleads = np.zeros([dataset.shape[0], dataset.shape[1], 5000])

    for i in range(0,12):
        for j in range(0, dataset.shape[0]):
            testingleads[j, i, :] = dataset[j, i]

    del dataset
    gc.collect()

    return testingleads

# Loading weight metric from challenge to use the 27 classes
def weight_metric():
    weights = pd.read_csv("../Data/weights.csv", index_col=0)
    ctcodes = pd.read_csv("../Data/Dx_map.csv")
    ctcodes = ctcodes.iloc[:, 1:]
    replacedict = dict(zip(ctcodes.iloc[:,0], ctcodes.iloc[:,1]))

    weights.columns = weights.columns.astype(int)
    weights.rename(columns=replacedict, index=replacedict, inplace=True
)
    weightindex = np.array(weights.index)
    return weights, weightindex


# Delete rows where labels are missing
def delete_missing_rows(leads, labels):
    print("Deleting Rows/n")
    indexvals=[]
    for i in range(0, labels.shape[0]):
        if len(np.unique(labels[i,:], axis=0)) == 2:
            indexvals.append(i)
    leads = np.take(leads, indexvals, axis=0)
    labels = np.take(labels, indexvals, axis=0)

    del indexvals
    gc.collect()

    return leads, labels


trainingleads, testingleads, traininglabels, testinglabels,
    validationleads, validationlabels = dataloader("../Data/fullecgdata
    .pkl")
```

Listing 3. A function that utilises the "train_test_split" function in the sci-kit learn library to perform holdout sampling in order to produce a dataset for ResNet training.

B.0.2 Data Resampling and Transformation on the ECG recordings

```python
def extract_leads():
    # Directories containing ECG recordings
    dirs = ["../Data/CPSC2018/Training_WFDB/",
    "../Data/CPSC2018_UNUSED/Training_2/",
    "../Data/GEORGIA/WFDB/",
    "../Data/PTB/WFDB/",
    "../Data/PTBXL/WFDB/",
    "../Data/STPETERSBURG/WFDB/"]
    # Sampling frequency for each dataset source
    fsarray = [500, 500, 500, 1000, 500, 257]

    i=0
    fsval = -1
    tarfs = 500 # Target sampling frequency value (500Hz)

    for directory in dirs:
        print(directory)
        fsval+=1
        for filename in os.listdir(directory):
            if filename.endswith(".mat"):
                # Load ECG recording in temporary variable
                temp = sp.loadmat(str(directory)+str(filename))
                I = pd.DataFrame(temp['val'])
                # Resample frequency based on dataset source (using "fsarray" variable)
                for j in range(I.shape[0]):
                    # Resample to 500Hz
                    if fsval == 3 or fsval == 5:
                        tempI = sig.resample(I.iloc[j,:].to_numpy(), 500)
                    else:
                        tempI = I.iloc[j,:].to_numpy()

                    # Truncate or Zero Pad Signals based on number of samples
                    if len(tempI) > 5000:
                        tempI = tempI[0:5000]
                    elif len(tempI) < 5000:
                        zeropadtemp = [0] * (5000 - len(tempI))
                        tempI = np.concatenate([tempI, zeropadtemp])
                    fulldataset.at[i, str(leads[j])] = tempI
                i+=1
            else:
                continue
```

Listing 4. A function to implement data resampling and transformation on the ECG recordings.

B.1 ResNet Implementation and Training

```python
# Using a function to apply 1d convolution where kernel_size=7
def conv3x1(in_planes, out_planes, stride=1):
    return nn.Conv1d(in_planes, out_planes, kernel_size=7, stride=stride,
                     padding=3, bias=False)

# Using a function to apply 1d convolution where kernel_size=1
def conv1x1(in_planes, out_planes, stride=1):
    return nn.Conv1d(in_planes, out_planes, kernel_size=1, stride=
   stride, bias=False)

# Residual Block
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x1(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm1d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x1(planes, planes)
        self.bn2 = nn.BatchNorm1d(planes)
        self.downsample = downsample
        self.stride = stride
        self.dropout = nn.Dropout(.2)

    # Feed data through the network
    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.dropout(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out

# Bottleneck layer to reduce overfitting
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = conv1x1(inplanes, planes)
        self.bn1 = nn.BatchNorm1d(planes)
        self.conv2 = conv3x1(planes, planes, stride)
        self.bn2 = nn.BatchNorm1d(planes)
        self.conv3 = conv1x1(planes, planes * self.expansion)
        self.bn3 = nn.BatchNorm1d(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride
        self.dropout = nn.Dropout(.2)

    # Feed data through the network
    def forward(self, x):
        identity = x
```

```python

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)
        out = self.dropout(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out

# ResNet model (defining architecture)
class ResNet(nn.Module):
    # in_channel=number of leads, out_channel=number of classes
    def __init__(self, block, layers, in_channel=12, out_channel=27,
    zero_init_residual=False):
        super(ResNet, self).__init__()
        self.inplanes = 64 # Batch size
        self.conv1 = nn.Conv1d(in_channel, 64, kernel_size=5000, stride
    =2, padding=7,
                               bias=False)
        self.bn1 = nn.BatchNorm1d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool1d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(512 * block.expansion, out_channel)

        for m in self.modules():
            if isinstance(m, nn.Conv1d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm1d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)


        if zero_init_residual:
            for m in self.modules():
                if isinstance(m, Bottleneck):
                    nn.init.constant_(m.bn3.weight, 0)
                elif isinstance(m, BasicBlock):
                    nn.init.constant_(m.bn2.weight, 0)


    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion, stride
   ),
                nn.BatchNorm1d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)


    # Feed data through the network
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

```python

# ResNet18 model
def resnet18(pretrained=False, **kwargs):
    model = ResNet(BasicBlock, [2, 2, 2, 2], **kwargs)
    return model

# ResNet34 model
def resnet34(pretrained=False, **kwargs):
    model = ResNet(BasicBlock, [3, 4, 6, 3], **kwargs)
    return model

# Training ResNet
def train(data, labels, validationleads, validationlabels, epochnum,
   threshval, opt, resnettype):
    if torch.cuda.is_available():
        dev = "cuda:0"
    else:
        dev = "cpu"

    # Batch size
    batchval=64
    batchind = np.array(range(0, data.shape[0], batchval))
    #Initialise Model
    if resnettype == 18:
        model = resnet18().double()
    elif resnettype == 34:
        model = resnet34().double()
    model.cuda()
    # Array to store trained models
    models = []
    # Optimiser
    if opt == 0:
        optimiser = torch.optim.Adam(filter(lambda p: p.requires_grad,
model.parameters()), lr=0.001, weight_decay=1e-5)
    elif opt == 1:
        optimiser = torch.optim.SGD(model.parameters(), lr=0.001,
momentum=0.9)

    # Initialise loss function
    criterion = nn.BCEWithLogitsLoss()
    # Model Save path
    model_save_path = "torch_models/"
    # Ndarray to store predicted labels
    pred_labels = np.zeros(shape=labels.shape)
    # Used to store models based on f-measure values
    prev_f_measure_val = 0.0
    # Arrays to store history of results
    f_measure_hist = np.ones(epochnum)
    training_loss_hist = []
    validation_loss_hist = []
    training_acc_hist = []
    validation_acc_hist = np.ones(epochnum)
    loss_batches = 0.0
    # Training phase
    for epoch in range(0, epochnum):
        model.train()
        print("TRAINING EPOCH {}".format(epoch))
        batchcounter = 0
        loss_epoch = 0.0
        loss = 0.0
        pred_labels = np.zeros(shape=labels.shape)
        epoch_training_loss = []
        # For each epoch, train on the training set
        with torch.set_grad_enabled(True):
            for i in batchind[:-1]:
                # training data - Numpy to Tensor
                x = torch.from_numpy(data[i:i+batchval,:,:] / 1000)
                x = x.cuda()

                # training labels - Numpy to Tensor
                trainlabels = torch.from_numpy(labels[i:i+batchval,:]).
    double()
                trainlabels = trainlabels.to(device=dev)

                # Reset gradients
                optimiser.zero_grad()

                # Make prediction
                y = model(x).to(dev)
                # Apply output layer function to predictions
                y_prob = torch.sigmoid(y)

                # Calculate loss
                loss = criterion(y, trainlabels)
                temploss = loss.item() * x.size(0)
                loss_epoch += temploss

                # Backwards step
                loss.backward()
                optimiser.step()

                # Calculate loss
                loss_batches += temploss
                batchcounter += x.size(0)
                current_loss = loss_batches / batchcounter

                # Store predictions and labels
                if i == 0:
                    all_train_labels = trainlabels
                    all_pred_prob = y_prob
                else:
                    all_train_labels = torch.cat((all_train_labels,
trainlabels), 0)
                    all_pred_prob = torch.cat((all_pred_prob, y_prob),
0)

                # Covert predictions from tensor to numpy
                pred_labels[i:i+batchval,:] = y_prob.cpu().detach().
numpy()

                # Apply threshold to predictions to recieve binary
output
                pred_labels[pred_labels < threshval] = 0
                pred_labels[pred_labels > threshval] = 1

                # Calculate Training Accuracy
                training_acc = accuracy_score(labels[i:i+batchval,:],
pred_labels[i:i+batchval,:])

                # Store Loss
                epoch_training_loss.append(current_loss)

                # Output current training information
                print("{}".format(time.strftime("%H:%M:%S", time.
localtime())), ", Epoch: {} [{}/{}],".format(epoch,i+batchval,data.shape[0]),
            "Loss: {}".format(current_loss), "Accuracy: {}".format(training_acc))

        training_acc_hist.append(training_acc)
        training_loss_hist.append(epoch_training_loss)

    # Generate confusion matrix
    conf_mat_train = multilabel_confusion_matrix(labels[:pred_labels.shape[0],:], pred_labels)
    class_report = classification_report(labels[:pred_labels.shape[0],:], pred_labels)



    # Evaluation mode (using validation set)
    model.eval()
    # Batch size
    val_batchval=64
    batchindval = np.array(range(0, validationleads.shape[0], val_batchval))
    valid_loss = 0.0 # Loss variable
    batchcounterval = 0 # batch counter variable
    val_pred_labels = np.zeros(shape=validationlabels.shape)
    epoch_validation_loss = []
    # Initialise loss function
    criterion = nn.BCEWithLogitsLoss()
    with torch.set_grad_enabled(False):
        for j in batchindval[:-1]:
            # validation leads - numpy to tensor
```

```
                x_val = torch.from_numpy(validationleads[j:j+
val_batchval,:,:] / 1000)
                x_val = x_val.cuda()
                # validation labels - numpy to tensor
                val_labels = torch.from_numpy(validationlabels[j:j+
val_batchval,:]).double().cuda()
                val_labels = val_labels.to(device=dev)

                # Make prediction
                val_y = model(x_val).to(device=dev)
                # Apply output layer function
                val_y_prob = torch.sigmoid(val_y)

                # Calculate loss
                val_loss = criterion(val_y, val_labels)
                valid_loss = val_loss.item() * x_val.size(0)
                batchcounterval += x_val.size(0)
                current_val_loss = valid_loss / batchcounterval

                # Store raw predictions
                if j == 0:
                    all_val_labels = val_labels
                    all_val_prob = val_y_prob
                else:
                    all_train_labels = torch.cat((all_val_labels,
val_labels), 0)
                    all_pred_prob = torch.cat((all_val_prob, val_y_prob
), 0)

                # Output predicted values from tensor to numpy
                val_pred_labels[j:j+val_batchval,:] = y_prob.cpu().
```

```
detach().numpy()

                # Ouput current validation loss
                epoch_validation_loss.append(current_val_loss)
                print("Epoch: {}".format(epoch), "Validation Loss: {}".
format(current_val_loss))

        # Calculating and Storing f-measure and validation accuracy
        validation_loss_hist.append(epoch_validation_loss)
        models.append(model)

        tempfmeasure = []
        tempaccmeasure = []



        val_pred_labels[val_pred_labels > threshval] = 1
        val_pred_labels[val_pred_labels < threshval] = 0

        val_pred_labels = val_pred_labels.astype(int)
        f_measure_val = f1_score(validationlabels, val_pred_labels,
average='samples')
        validation_acc = accuracy_score(validationlabels,
val_pred_labels)
        tempfmeasure.append(f_measure_val)
        tempaccmeasure.append(validation_acc)

        validation_acc_hist[epoch] = validation_acc

        f_measure_hist[epoch] = f_measure_val

        print("F-MEASURE ({}): {}".format(threshval, f_measure_val), "
ACCURACY: {}".format(validation_acc))

        # Save model to path
        if f_measure_val > prev_f_measure_val:
            newmodelpath = model_save_path + "model_e" + str(epoch) + "
_f" + str(f_measure_val) + ".pth"
            torch.save(model.state_dict(), newmodelpath)
            prev_f_measure_val = f_measure_val

    # print(challenge_metric_hist)
    print("F-measure History: ", f_measure_hist)
    print("Accuracy History: ", validation_acc_hist)

    return training_loss_hist, validation_loss_hist, f_measure_hist,
validation_acc_hist, conf_mat_train, class_report, models



training_loss_hist, validation_loss_hist, f_measure_hist,
validation_acc_hist, conf_mat_train, class_report, models = train(
trainingleads, traininglabels, validationleads, validationlabels,
epochnum=1, threshval=0.35, opt=0, resnettype=34)
```

Listing 5. Implementation and training of the ResNet model

B.2 ResNet Testing

```
def test_eval(model, testingleads, testinglabels, threshval):
    if torch.cuda.is_available():
        dev = "cuda:0"
    else:
        dev = "cpu"
    with torch.no_grad():
        model.cuda()
        # Batch size
        batchval=64
        batchind = np.array(range(0, testingleads.shape[0], batchval))
        test_pred_labels = np.zeros(shape=testinglabels.shape)

        for i in batchind[:-1]:
            # testing leads - numpy to tensor
            x = torch.from_numpy(testingleads[i:i+batchval,:,:] / 1000)
            x = x.cuda()

            # testing labels - numpy to tensor
            test_labels = torch.from_numpy(testinglabels[i:i+batchval
,:]).double().cuda()

            # Make prediction
            y = model(x).to(device=dev)
            # Apply output layer function
            test_y_prob = torch.sigmoid(y)

            # Store predictions
            if i == 0:
                all_test_labels = test_labels
                all_test_pred_prob = test_y_prob
            else:
                all_test_labels = torch.cat((all_test_labels,
test_labels), 0)
                all_test_pred_prob = torch.cat((all_test_pred_prob,
test_y_prob), 0)
```

```python

        # Get predicted labels in numpy format
        test_pred_labels[i:i+batchval,:] = test_y_prob.cpu().detach().numpy()

        test_pred_labels = test_pred_labels.astype(int)

        test_pred_labels[test_pred_labels > threshval] = 1
        test_pred_labels[test_pred_labels < threshval] = 0

        # Calculate Accuracy
        f_measure_temp_val = accuracy_score(testinglabels[i:i+batchval,:], test_pred_labels[i:i+batchval,:])

        # Print testing accuracy per batch
        print("Accuracy: {}".format(f_measure_temp_val))
    finalpred = all_test_pred_prob.cpu().detach().numpy()
    finalpred[finalpred > threshval] = 1
    finalpred[finalpred < threshval] = 0

    # Calculate weighted f-measure
    FMEASUREVAL = f1_score(testinglabels[:all_test_pred_prob.shape[0],:], finalpred, average='samples')
    # Ouput confusion matrix and classification report
    conf_mat_test = multilabel_confusion_matrix(testinglabels[:finalpred.shape[0],:], finalpred)
    test_class_report = classification_report(testinglabels[:finalpred.shape[0],:], finalpred)
    print(FMEASUREVAL)

    return conf_mat_test, test_class_report
```

```python


testingmodel = models[0]

conf_mat_test, test_class_report = test_eval(testingmodel, testingleads
    , testinglabels, threshval=0.3)
```

Listing 6. Testing the ResNet model