ECG classification using deep neural networks: Investigating architecture optimization and transfer learning

Pawel Kudzia
Simon Fraser University

Abstract — Electrocardiogram (ECG) interpretation is an important medical diagnostic tool for diagnosing conditions of the heart. The use of neural networks aiming to assist cardiologists in the interpretation of ECG signals show great potential and could lead to improved patient outcomes. Here, I compared and developed several neural network architectures for classifying ECG signals from widely available datasets. Additionally, I compared modelling results when using augmented data and evaluated how well the models trained on one dataset classified data from another dataset. I found that the presented neural network architectures accurately classified pathological ECG signals, comparable to that of several published articles. I also found that data augmentation diminished overall performance, suggesting the approach taken was too simple. Finally, I found that transfer learning is possible but, classification performance was poor on pathological ECG.

Keywords —ECG, deep neural networks, architecture optimization, transfer learning, classification problems, error matrix

1.0 Introduction

An electrocardiogram (ECG) is a measurement of the electrical activity of the heart muscle. Each time the heart pumps blood it produces a characteristic time series waveform with distinct and identifiable features (right, Figure 1). In medicine, the ECG signal is used as a diagnostic tool to determine if the heart is functioning as expected. When someone has a pathology, such as heart valve leaks or tears, there can be subtle but detectable changes in the ECG signal (left, Figure 1). The ability to detect the presence of a pathology early could lead to quick diagnosis and improved outcomes for patients, not to mention reductions in long term healthcare costs [1]–[4].

Computer-aided interpretation of ECG signals has been an increasingly important tool in the clinical workflow, with first attempts dating back to the late 1950s [3]. Although sophisticated computer algorithms have shown promise, high rates of misdiagnosis are common (~50%). The presence of noise, variability in wave morphology between patients, and the mixture of both subjective and objective characteristics makes ECG classification a difficult problem, where even experienced cardiologists can disagree on prognosis [5]. The use of deep neural networks has led to major advances in image classification, speech recognition, and many medical diagnostics [1], [6], [7]. The ability of deep neural networks to recognize patterns and learn features of time varying signals without extensive preprocessing makes them particularly well suited for ECG classification.

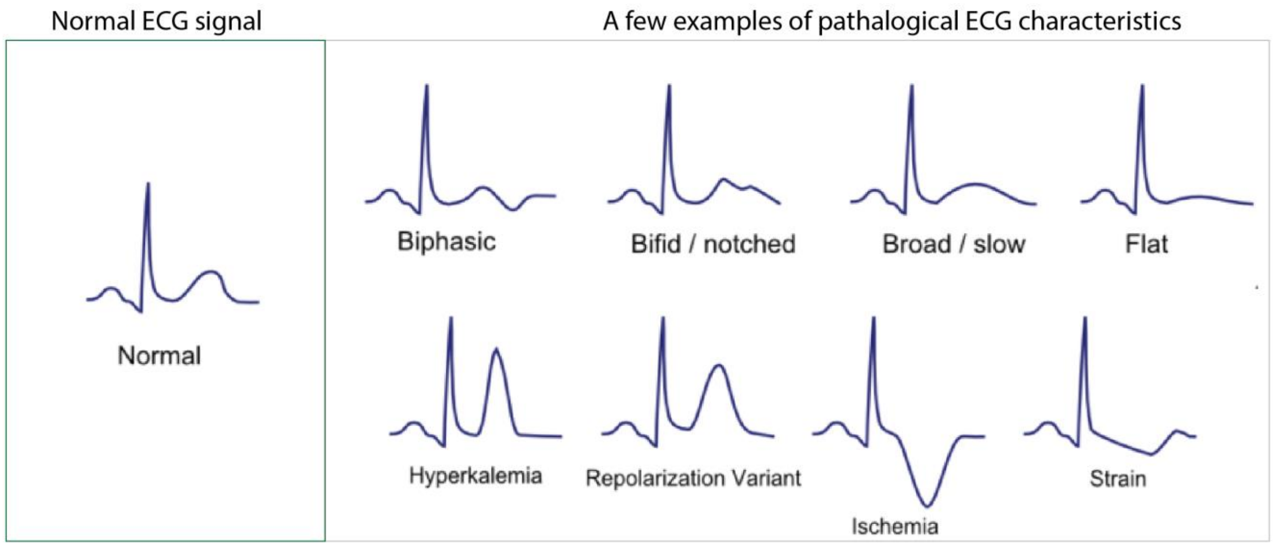


Figure 1: *Example characteristics of normal and pathological ECG signals.*

There are many possible pathologies that can be observed in ECG waveforms. It is common for researchers to study only certain more common pathologies, or group pathologies into categories due to limitations in the amount of data on pathological ECG signals [1], [4], [8]. In general, many research groups have shown great success with ECG classification using neural networks, achieving low misclassification errors, and accuracy's nearing 99% [2], [7], [9]–[13]. Although the results are profound and have led to startup companies and major advances, the caveats are small representative test sets, with limited patients, and limited variations in the ECG pathologies themselves. This increases the chances of missing abnormalities that may exist warranting expert cardiologist option (who can also miss these suitable features). Recently, a monumental effort led by Andrew Ng's group at Stanford university, proposed a neural network architecture for evaluating up to 12 different pathologies (more then has ever been proposed before) [3], [7]. This effort was only possible due to the extensive amount of data the group was able to collect (over 50,000 patients with all ECG annotations performed by cardiologists) which made it possible to train and evaluate their model.

This project here aims to use deep neural networks to classify normal and pathological ECG waveforms from publically available datasets. Specifically, there are several objectives. The first objective is build, develop, and train a convolutional neural network using a previously established architecture and compare its ability to classify normal and 4 pathological ECG signals to the ability of two author developed architectures. The second objective is characterize the performance variations when a simple data augmentation is performed on the dataset. Finally, the third objective is to quantify the performance of the models established in aim 1 and 2 on a novel ECG dataset via transfer learning.

2.0 Methods

2.1 Data Curation

ECG data was taken from the MIT-BIH arrhythmia (48 subjects, f = 360 Hz) and the PTB Diagnostic ECG (290 subjects, f= 1000 Hz) datasets as made available through Kaggle courtesy of [1]. Both of these ECG datasets have been labelled by trained cardiologists. The dataset has some preprocessing steps that were taken which modify it from its raw time-series form. In brief, the continuous ECG voltage signal was divided into 10s windows, voltage normalized to be between 0-1, peaks were identified using the first derivative, and the median time interval between peaks was determined (*T_interval*). Using the *T_interval* determined for each 10 second window, the signal was then divided up such that each frame starts from a peak and continues for 1.2**T_interval*. Lastly, to ensure the divisions of each new frame have a fixed length, each frame is padded with zeros to make its length equal to an array of 187 in length. Each interval is then given the corresponding label which the cardiologist has previously provided. This label is added to each array as the 188th value. These labels range are either normal or pathological as expressed in Table 1. This type of beat extraction does not filter the signal or make any assumption about the morphology, which has and been shown as an effective method for ECG processing. A further and more detailed overview of these preprocessing steps is provided elsewhere [1].

Table 1: The two datasets explored in this project MIT-BIH and PTB Diagnostic

MIT-BIH ECG				PTB Diagnostic ECG			
ECG Class	Label	# samples	% of Total	ECG Class	Label	# samples	% of Total
Normal	0	90589	82.8%	Normal	0	10506	72.2
Fusion of Paced and Normal	1	2779	2.5%	Abnormal	1	4046	27.8
Premature Ventricular Contraction	2	7236	6.6%				
Atrial Premature	3	803	0.7%				
Fusion of Ventricular and Normal	4	8039	7.3%				

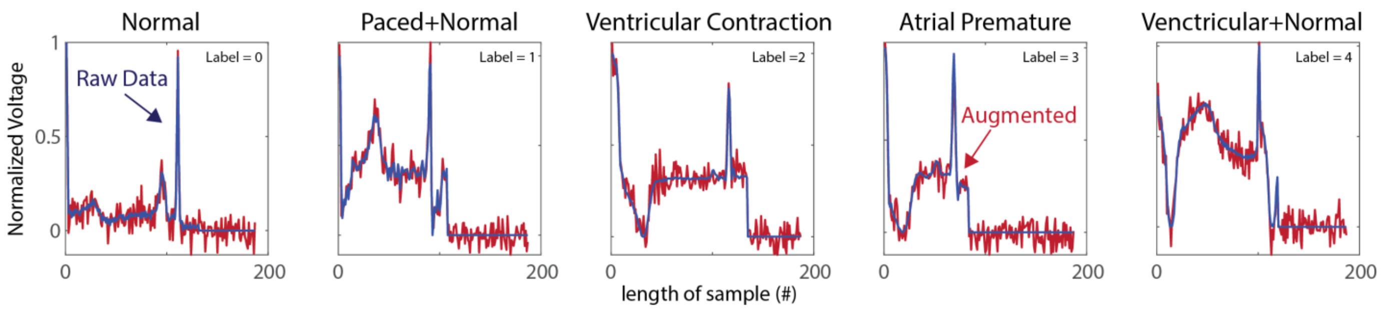
2.2 Data Augmentation

The number of representative samples for each class shows a large class imbalance (Table 1), where the majority of all data is of normal non-pathological ECG. To address this class imbalance I decided to perform a data augmentation on the minority classes [14]. It is generally suggested that augmenting just the training data increases reliability and generalization of the model [13]. With this in mind, the sample data was first split into 80% training, 20% testing. A *StratifedShuffleSplit* approach was taken such that the data was split equally amongst the classes. Next, the training data was further split such that 16% went into a separate validation set. The validation and testing sets were then set aside well a very simple data augmentation was performed on the training set for the MIT-BIH dataset (the main dataset explored in the bulk of this project). The majority class (normal ECG) was first reduced down to 10,000 samples and the minority classes were *resampled* such that each class now contained the same amount of observations. Random Gaussian noise was injected to each sample of the resampled data to further increase generalizability. The amplitude of this noise was small (0.03-0.05) to help ensure each label kept its class (it is possible though that this could change the class, this is why a small amount of noise was used). A figurative representation of the data is shown Figure 2 and Table 2 shows the summary of the data after splitting and augmenting.

Table 2: The split MIT-BIH dataset showing the number of samples in each class as well the new formed augmented data set which down sampled the majority class and up sampled the minority.

		MIT-BIH ECG Dataset			
		Training Set 1			Training Set 2
		Non-Augmented Training	Validation Set	Test Set	Augmented Training
	% samples	64%	16%	20%	64%
	Label number	# samples	# samples	# samples	# samples
Normal	0	57976	14495	18118	10000
Pathological	1	1778	445	556	10000
	2	4631	1158	1447	10000
	3	514	128	161	10000
	4	5145	1286	1608	10000

Figure 2: Raw data (MIT-BIH) from each category and data with Gaussian noise (augmented data).



2.3 Network Architectures

All networks were trained using the following set parameters: batch size = 64, patience = 20, epochs = 500. Model weights were set to be balanced meaning that for the augmented data these weights were just 1 but for the unbalanced data, the weights were balanced following the default approach proposed in Keras using callback functions. Learning rate was programed to have an exponential decay such that it started at 0.001, had 10,000 decay steps at a decay rate of 0.75. The Adam optimizer was used with beta_1 = 0.9 and beta_2, 0.999 set. The optimizer optimized for *spare categorical cross entropy* which is the recommended loss for this type of data [1].

Model 1 is a convolutional neural network with 55,013 parameters based on the architecture proposed elsewhere [1]. Model 2 is a recurrent neural network (RNN) with long short term memory (LSTM) having 54,661 parameters. This model was developed following recommendations for building RNN's and was hand tuned by trial and error during pilot experiments to include dropout [12]. RNN's are well suited for time series data of this type and should work well in classifying the ECG signals [12], [15]. Model 3 is a deep neural network with variable layers that I developed. Since developing neural network architecture is not my expertise, I wanted to use numerical optimization to help develop a simple deep neural network. This process is further explained in section **2.4 Optimization**. The model that trained on the non-augmented data converged on an architecture having 65,669 parameters. The model that trained on the augmented data converged on an architecture having 75,333 parameters (specific models details are provided in the Appendix).

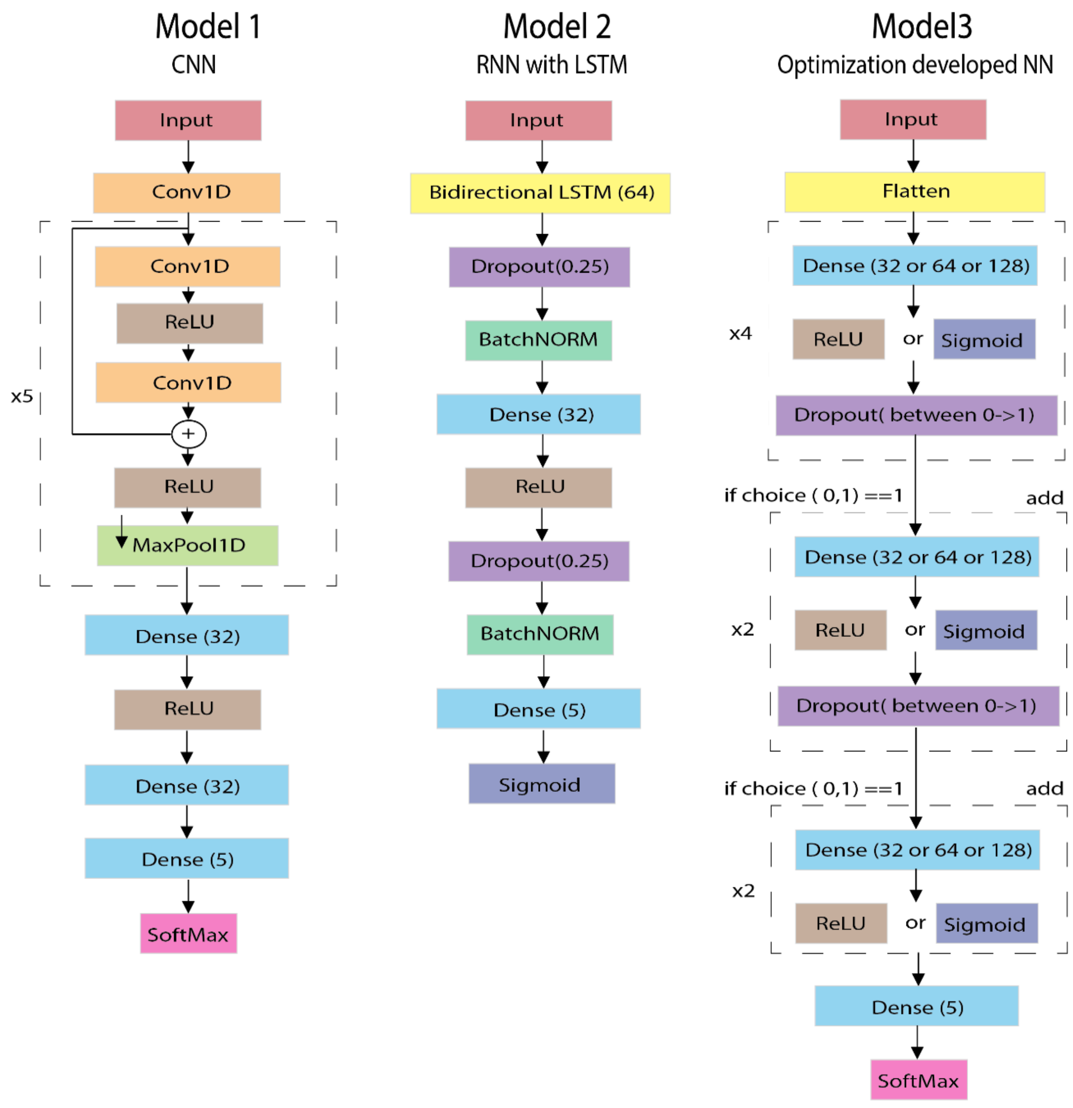


Figure 3: **Model 1**: Convolutional neural network based on [1], **Model 2:** recurrent neural network (RNN) with long short term memory (LSTM), and **Model 3**: an optimization developed neural network with variables parameter and number of layers developed using package *hyperopt* [16].

2.4 Optimization

An architecture optimization approach was implemented for Model 3 (Figure 3). A numerical optimization was setup to minimize negative test accuracy as depicted by equation 1. The optimization routine first evaluated a possible model architecture based on possible options, then model parameters were generated and optimized by minimizing the spare *categorical cross entropy*, same as for the other models. The optimization model was improved over 40 epochs of optimization (i.e. 40 different possible architectures were explored and optimized). The best combination of model architecture and model parameters resulting in the lowest objective function (Equation 1) was chosen for further evaluation (resulting model specifics shown in Appendix). This optimization routine was setup up using *hyperopt* [16].

Equation 1. $$\textbf{Objective Function} = \min(-(test_{set}\ Accuracy))$$

2.5 Transfer Learning

After training and evaluating the models on MIT-BIH dataset, I wanted to see how well the trained model would perform on a novel dataset namely the PTB Diagnostic, which has only two classes: normal and pathological (all pathological signals grouped into one) (Table 1). To do this, I used the best trained model from each architecture and added two dense layers and an output node such that the models now went from the 5 output nodes (5 classes in the MIT-BIH dataset) through two dense layers and outputted either 0 or 1 (normal or pathological) as depicted in Figure 4. This bottlenecking approach was based on methodology proposed in literature [1]. The two added dense layers where then trained on the new data while the other model weights were left untouched.

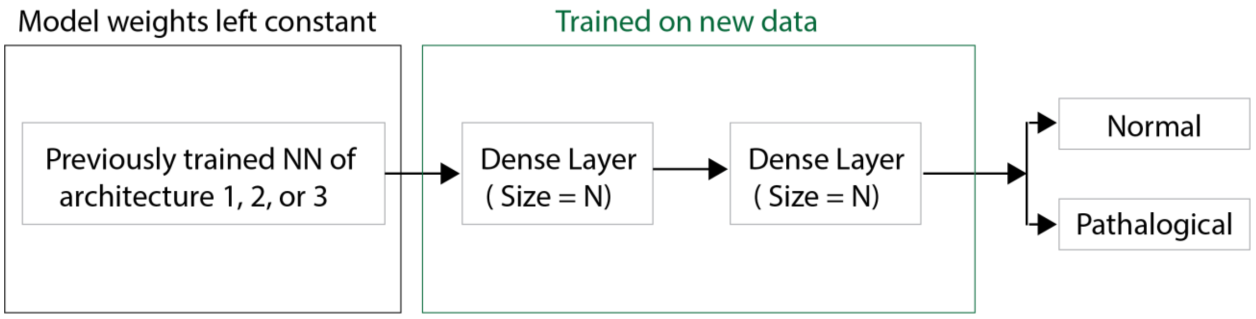


Figure 4: Approach used to add two additional dense layers to previously evaluated models. After some pilot experiments, the added dense layer was set to be a size of N =32 as is also recommend elsewhere [1].

In addition to this approach, I also evaluated how training deep layers after adding the two dense layers would affect the transfer learning. To evaluate this in a systematic way I also evaluated the effects of training 5 layers deep and training 8 layers deep and present this in my results.

2.6 Metrics

The following metrics were utilized in this study. Recall which reflects the accurate positive identification of the ECG label as defined by equation 2.

Equation 2.
$$\textbf{Recall} = \frac{\text{True Positive}}{\text{True Positive+False Negative}}$$

Precision which is the accurate positive identification—when the model predicted positive was it correct? This is defined by equation 3.

Equation 3.
$$\textbf{Precision} = \frac{\text{True Positive}}{\text{True Positive+False Positive}}$$

The F1- score is a metric for comparing the performance as a ratio of the mean of the recall and precision. The higher the F1-score the better the model did is the best way to look at it. F1-score is convenient to look at as it can be calculated for each label. In the context of our problem here this is a great metric to use to compare model performance as expressed in equation 4.

Equation 4.
$$\textbf{F1 Score} = \frac{2*\text{Recall}*\text{Precision}}{\text{Recall+ Precision}}$$

The accuracy is the ratio between correctly predicted outcomes and the sum of all predictions. A relatively easy metric to understand for looking at global performance of the model as depicted by equation 5.

Equation 5.

$$\textbf{Accuracy} = \frac{\text{True Positive}+\text{True Negative}}{\text{True Positive}+\text{True Negative}+\text{False Positive}+\text{False Negative}}$$

Finally, the error matrix also known as a confusion matrix is a convenient table that shows the percentage of classified or misclassified labels in a matrix (it is the recall measure expressed for every label). Figure 5 provides an example of how this table will be presented in the results of this research. The diagonals of the matrix express the accurate identification of the correct label and the closer this accuracy is to 100% the better the model did. Any numbers shown off the diagonal can be thought of as unfavourable as this expresses misclassification of the model.

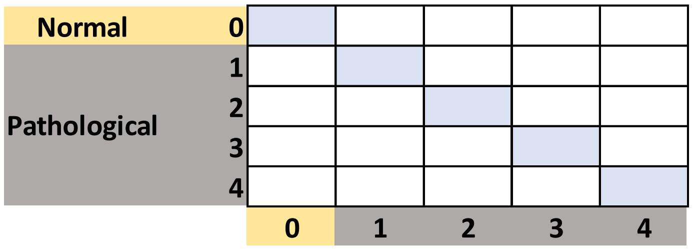


Figure 5: Example of an error matrix used for evaluating performance of a classification model. High accuracy scores across the diagonal indicates good classification.

3.0 Results

3.1 Network architecture influences performance

Network architecture has a noticeable effect on the F1-score of each model, as can be seen in Table 3. We can see that Model A has the highest overall F1-score when the data that the network is trained on is augmented (Model A, F1-Score = 0.954) but when the data the network is trained on is not augmented Model B performs best (Model B, F1-Score = 0.993). We can further see that the models all have high accuracies meaning that their ability to perform this classification task is good. The one caveat here is that the testing data is heavily skewed towards normal ECG signals. If the model correctly classifies the normal ECG it can still achieve a high accuracy without correctly classifying pathological ECG.

To better visualize the errors and misclassification occurring in the models, Figure 6 shows the error matrixes. Here we can see that all the models had a difficult time with the pathological label 1(fusion of paced and normal) as indicated by the higher misclassifications (10-35%) of this label. The models performed exceptionally well at classifying the normal ECG signal which was especially noticeable when the data was trained on the non-augmented training set (achieving almost 100% for all three architectures). The ability to well classify the larger normal set also explains why we see such large model accuracies. In general, all the architectures where good at classifying the data regardless of the training dataset howevr, Model 1 one performed noticeably better than the other two models. Model 3 had misclassifications errors higher than other models when it came to classifying pathological ECG signals vs. normal.

Table 3: F1-scores for the 3 models trained on augmented and non-augmented datasets MIT BIT. Green shading indicates best performance. The closer the F1-score is to 1 the better the model performed at classifying.

	F1-Score			
	Augmented Training	**Model 1**	**Model 2**	**Model 3**
Normal	Normal	0.974	0.960	0.966
Pathological	Fusion of Paced and Normal	0.680	0.588	0.679
	Premature Ventricular Contraction	0.934	0.891	0.847
	Atrial Premature	0.474	0.393	0.484
	Fusion of Ventricular and Normal	0.975	0.972	0.910
	Weighted Average	**0.960**	**0.944**	**0.943**
	Macro Average	**0.807**	**0.761**	**0.777**
	Non-Augmented Training	**Model 1**	**Model 2**	**Model 3**
Normal	Normal	0.992	0.997	0.989
Pathological	Fusion of Paced and Normal	0.843	0.925	0.770
	Premature Ventricular Contraction	0.961	0.984	0.936
	Atrial Premature	0.799	0.895	0.757
	Fusion of Ventricular and Normal	0.989	0.997	0.982
	Weighted Average	**0.985**	**0.993**	**0.978**
	Macro Average	**0.917**	**0.960**	**0.887**

Table 4: Global accuracy metrics for the 3 models trained on augmented and non-augmented datasets. Green shading indicates best performance. If the model correctly classifies the normal ECG it can still achieve a high accuracy without correctly classifying pathological ECG.

Accuracy	**Model 1**	**Model 2**	**Model 3**
Augmented Training	95.4%	93.2%	97.9%
Non-Augmented Training	98.5%	99.3%	94.0%

Training on Augmented Data

	# of samples		**Model 1 CNN**					**Model 2 RNN**					**Model 3 Optimization**				
Normal	**18118**	**0**	95%	2%	1%	2%	0%	93%	3%	1%	2%	0%	96%	1%	1%	1%	1%
Pathological	**556**	**1**	10%	89%	1%	0%	0%	10%	88%	2%	0%	0%	29%	68%	3%	0%	0%
	1447	**2**	1%	0%	96%	2%	0%	1%	1%	94%	3%	0%	11%	0%	86%	2%	0%
	161	**3**	2%	1%	6%	92%	0%	4%	0%	1%	94%	0%	12%	0%	4%	84%	0%
	1608	**4**	2%	0%	1%	0%	97%	1%	0%	0%	0%	99%	4%	0%	2%	0%	94%

Training on Non-Augmented Data

			Model 1 CNN					**Model 2 RNN**					**Model 3 Optimization**				
Normal	**18118**	**0**	99%	0%	0%	0%	0%	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
Pathological	**556**	**1**	13%	87%	0%	0%	0%	11%	88%	1%	0%	0%	35%	64%	1%	0%	0%
	1447	**2**	3%	1%	94%	1%	0%	1%	0%	98%	1%	0%	6%	0%	91%	3%	0%
	161	**3**	12%	2%	6%	80%	0%	7%	0%	6%	88%	0%	20%	0%	3%	76%	0%
	1608	**4**	1%	0%	0%	0%	99%	0%	0%	0%	0%	99%	3%	0%	0%	0%	97%
			0	**1**	**2**	**3**	**4**	**0**	**1**	**2**	**3**	**4**	**0**	**1**	**2**	**3**	**4**

Figure 6: Error matrixes for the 3 models explored showing the results for models trained on augmented and non-augmented training sets. Blue shading indicates diagonal squares (100% is perfect classification), red shaded color indicates noticeable misclassifications, grey shading indicates low error (<2%).

3.2 Data augmentation decreased performance

Data augmentation had a determinable effect on overall model performance as can be seen by looking at accuracy and loss performance for the three evaluated models shown in Figure 7. The model accuracy was lower and validation loss was higher for models that were trained on the augmented data set resulting in poorer F1-scores (Table 3), lower test accuracy (Table 4) and higher misclassification rates (Figure 6).

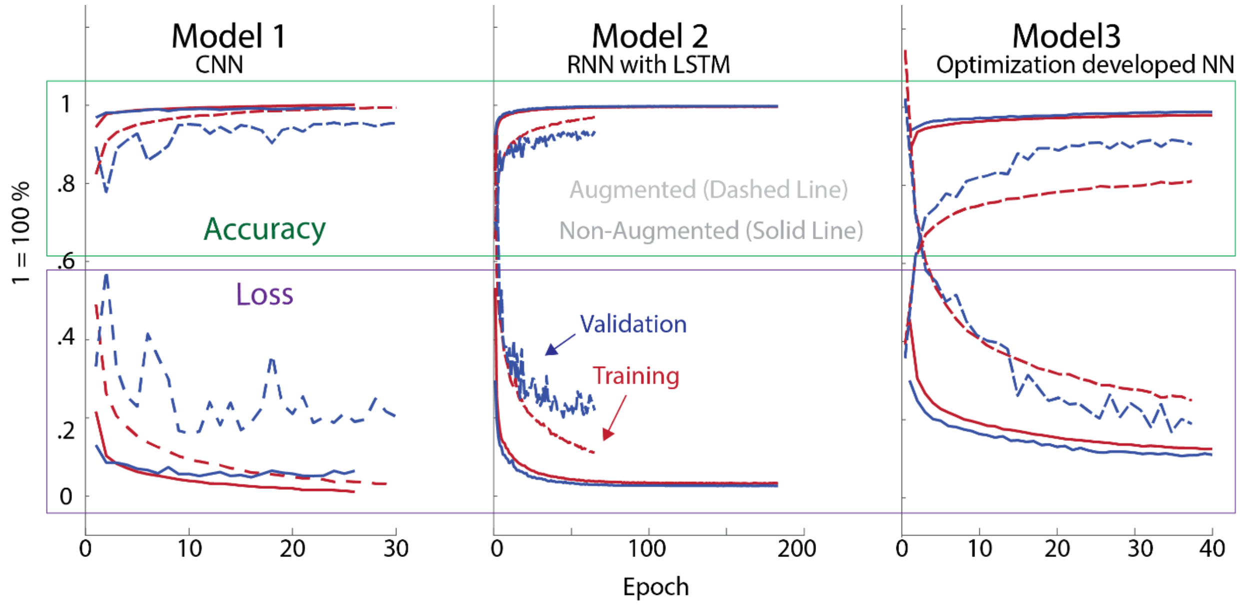


Figure 7: Accuracy (top section) and objective function loss (lower section) for validation and training sets using augmented (dashed lines) and non-augmented (solid line) training sets.

The augmented dataset required down sampling the majority class and up sampling the minority class. Perhaps the approach taken using *resample* from the Keras toolbox and adding Gaussian noise was too simple with too many similar cases for the network to train on. It is important to note that there were some improvements on classifying the pathological classes when the models trained on the augmented data (Figure 6) but in general the models performed more poorly. More sophisticated augmentation techniques appear to be beneficial for ECG data classification and should be explored in future work [11].

3.3 Transfer learning classification is strong on normal ECG but not on pathological

Figure 8 shows the F1-scores for the transfer learning results comparing: training the 2 added layers, training 5 layers deep, and training up to 8 layers deep. Applying the trained MIT-BIH models on a novel dataset resulted in a strong ability to classify the normal ECG signals but the ability to classify the pathological signal varied widely between models. Classifying the pathological data was particularly poor for the models trained on the augmented dataset. In terms of the F1-score, we can see in Figure 8 that model 3 had the best generalizability to this new dataset, resulting in the highest overall F1-score. It is interesting to note that model 3 had the poorest (although was still good at) classification performance for the MIT-BIH dataset.

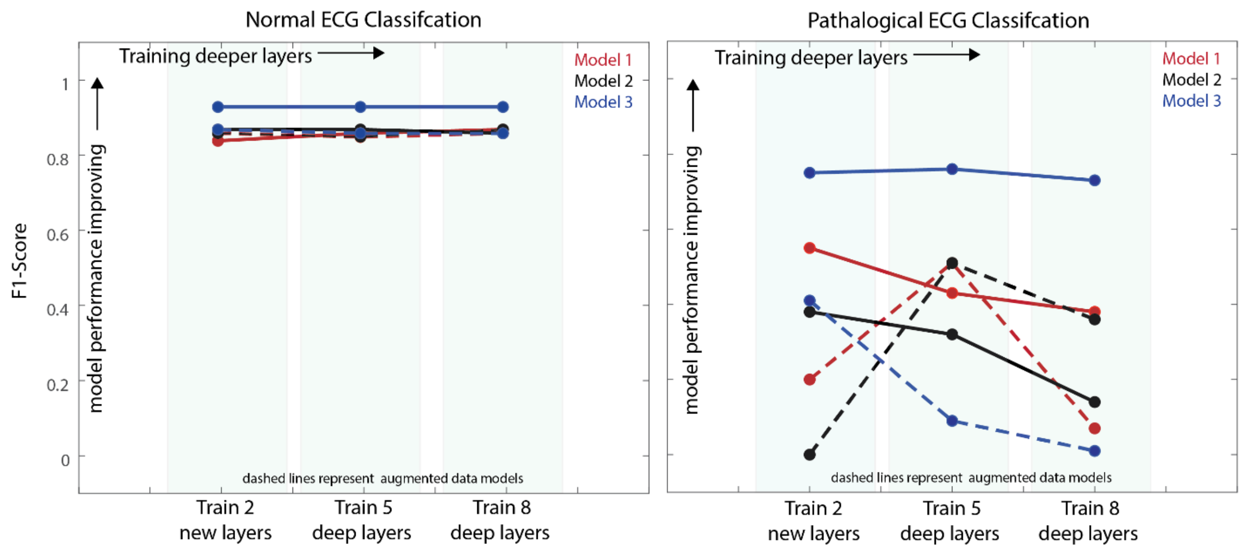


Figure 8: F1-scores for classification on PTB dataset after transfer learning from MIT_BIT trained models. PTB has two labels datasets either Normal (right) or Pathological (left). Results for model trained on augmented (dashed lines) and non-augmented (solid lines) training set are shown.

Looking further at the resulting error matrix just for the models trained on the 2 new added layers, we can see that model 3 shows a strong ability to correctly classifying the data, with Model 2 performing poorly (Figure 9). Training more layers did not appear to improve Model 3's performance and in most cases it deteriorated all models ability to classify the pathological data. In general though these results are encouraging and with further tuning improvements could further be made [1], [17].

Training on Augmented Data								
	# of samples		Model 1 CNN		Model 2 RNN		Model 3 Opt.	
Normal	2102	0	98%	2%	100%	0%	93%	7%
Pathological	809	1	12%	88%	100%	0%	69%	31%

Training on Non-Augmented Data								
			Model 1 CNN		Model 2 RNN		Model 3 Opt.	
Normal	2102	0	82%	18%	94%	6%	92%	8%
Pathological	809	1	45%	55%	73%	27%	29%	71%
			0	1	0	1	0	1

Figure 9: Error matrixes for the 3 models explored showing the transfer learning results for models trained on augmented and non-augmented training sets. Blue shading indicates diagonal squares, red shaded color indicates noticeable misclassifications, grey shading indicates low error (<2%). The PTB dataset has two classes normal and pathological. This requires the MIT-BIH trained models to bottle neck down to 2 outputs. Note: these error matrixes are only representative of results from training 2 new added dense layers.

3.4 Hyper parameters effect individual model performance

Although all of the results presented in this project used one set of hyper parameters as described in the methods, various other hyper parameter combinations were evaluated during the pilot testing of developing this systematic experiment. It warrants to present the effects of hyper parameter tuning to increase the transparency and further possibility of improving the presented results. Changes in parameters such as batch size, augmentation sample set or patience each resulted in differences in the final outcomes, as presented by the error matrix of an arbitrary model architecture with changes made to these parameters in Figure 10. Although further tuning and exploring the best combinations of these parameters is possible, there were limitations in the hardware the network was running on and the time constraints possible to evaluate long duration computations (Anaconda using a single-core 3.6 GHz Dell OptiPlex 7020 computer with 16 Gb of RAM). I am acknowledging that it is possible that the results presented her could be further improved upon, but I feel that the conclusions and end outcomes would likely not vary drastically from what was presented here. Nevertheless, further work could lead to improvements in performance and should be explored.

Evaluation of hyper-paramter tuning on outcomes																	
			Batch size = 256					Augmentation = 20,000					Patience = 10				
Normal	18118	0	82%	14%	1%	2%	1%	98%	2%	1%	0%	0%	93%	5%	1%	0%	0%
Pathological	556	1	5%	94%	1%	0%	0%	9%	90%	1%	0%	0%	3%	96%	1%	0%	0%
	1447	2	2%	3%	91%	4%	1%	2%	1%	96%	1%	0%	1%	1%	97%	1%	0%
	161	3	0%	4%	2%	94%	0%	9%	1%	7%	83%	0%	5%	2%	11%	82%	0%
	1608	4	0%	0%	0%	0%	99%	1%	0%	0%	0%	99%	0%	0%	0%	0%	99%
			0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

Figure 10: Error matrixes for 1 example model using variations in hyper parameters. Results show sensitivity to user selected hyper parameters, further work is warranted.

3.5 Comparison of results to literature

The results presented in this project were compared to results from other sources. Specially, the accuracy and F1-score was compared as shown in Table 5. The performance metrics achieved in this work are strong and compare well against what is found in recent literature. Although the classification ability of the proposed work is certainly good, caution must be taken when making direct comparisons of this sort. The test set used in this project was unbalanced, small, and only had several pathological signals (for example [3][7] evaluated 12 classes on over 50,000 patients). Making comparison to other models which may achieve lower F1-score and lower accuracy for this reason is deceptive and most be taken in to account. For this specific problem misclassification of pathological ECG has larger consequences then misclassifying a healthy normal ECG. Although this was not specifically taken into account in this project it is important to consider when evaluating the models ability to classify ECG signals.

Table 4: Comparing the accuracy and F1-scores obtained in this project to that of literature

Authors	Citation	**accuracy %**	Architecture	Authors	Citation	**F1-score**	**F1-score (cardiologist)**
2019 Mousavi et al.	[13]	99.92*	RNN+CNN	2020 Steenkiste et al.	[17]	0.80-0.98	
2017 Achaya et al.	[11]	97.40	CNN	2019 Hannun et al.	[3][7]	0.57-0.94**	0.53-0.91**
2016 Kiranyaz et al.	[6]	99.10*	CNN 2D	2019 Alfaras et al.	[10	0.78-0.98	
2010 Ye et al.	[8]	99.91*	Support Vector Machine				
Presented here		**93.2-99.3**		**Presented here**		**0.39-0.99**	

* more ECG classes explored then in this dataset and large sample size with augmentation

** Over 12 classes and dataset composed of over 50,000+ patients

4.0 Conclusions

In summary, the work presented here successfully met the objectives of the project. First, I was able to implement 3 neural network architectures for classifying ECG signals. All the proposed architectures had strong performance metrics and were able to classify the data to a high degree of accuracy. Second, I augmented the training set data and evaluated the performance effects. I found that the simple data augmentation approach led to an overall decrease in prediction performance. Lastly, I evaluated how well the models would perform when tested on a new data set. I found that the classification of normal ECG was strong but the classifying the pathological data proved to be more difficult. Further work is warranted in optimizing hyper parameters which would lead to improvements in the outcomes.

References

[1] M. Kachuee, S. Fazeli, and M. Sarrafzadeh, "ECG Heartbeat Classification: A Deep Transferable Representation," *2018 IEEE Int. Conf. Healthc. Inform. ICHI*, pp. 443–444, Jun. 2018, doi: 10.1109/ICHI.2018.00092.

[2] S. Kiranyaz, T. Ince, and M. Gabbouj, "Real-Time Patient-Specific ECG Classification by 1-D Convolutional Neural Networks," *IEEE Trans. Biomed. Eng.*, vol. 63, no. 3, pp. 664–675, Mar. 2016, doi: 10.1109/TBME.2015.2468589.

[3] A. Y. Hannun *et al.*, "Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network," *Nat. Med.*, vol. 25, no. 1, pp. 65–69, Jan. 2019, doi: 10.1038/s41591-018-0268-3.

[4] S. M. Jadhav, S. L. Nalbalwar, and A. A. Ghatol, "ECG arrhythmia classification using modular neural network model," in *2010 IEEE EMBS Conference on Biomedical Engineering and Sciences (IECBES)*, Nov. 2010, pp. 62–66, doi: 10.1109/IECBES.2010.5742200.

[5] J. Schläpfer and H. J. Wellens, "Computer-Interpreted Electrocardiograms," *J. Am. Coll. Cardiol.*, vol. 70, no. 9, pp. 1183–1192, Aug. 2017, doi: 10.1016/j.jacc.2017.07.723.

[6] 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)*, Milan, Aug. 2015, pp. 2608–2611, doi: 10.1109/EMBC.2015.7318926.

[7] P. Rajpurkar, A. Y. Hannun, M. Haghpanahi, C. Bourn, and A. Y. Ng, "Cardiologist-Level Arrhythmia Detection with Convolutional Neural Networks," *ArXiv170701836 Cs*, Jul. 2017, Accessed: Mar. 18, 2020. [Online]. Available: http://arxiv.org/abs/1707.01836.

[8] Can Ye, M. T. Coimbra, and B. V. K. Vijaya Kumar, "Arrhythmia detection and classification using morphological and dynamic features of ECG signals," in *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*, Buenos Aires, Aug. 2010, pp. 1918–1921, doi: 10.1109/IEMBS.2010.5627645.

[9] T. Li and M. Zhou, "ECG Classification Using Wavelet Packet Entropy and Random Forests," *Entropy*, vol. 18, no. 8, p. 285, Aug. 2016, doi: 10.3390/e18080285.

[10] M. Alfaras, M. C. Soriano, and S. Ortín, "A Fast Machine Learning Model for ECG-Based Heartbeat Classification and Arrhythmia Detection," *Front. Phys.*, vol. 7, p. 103, Jul. 2019, doi: 10.3389/fphy.2019.00103.

[11] U. R. Acharya *et al.*, "A deep convolutional neural network model to classify heartbeats," *Comput. Biol. Med.*, vol. 89, pp. 389–396, Oct. 2017, doi: 10.1016/j.compbiomed.2017.08.022.

[12] S. Saadatnejad, M. Oveisi, and M. Hashemi, "LSTM-Based ECG Classification for Continuous Monitoring on Personal Wearable Devices," *IEEE J. Biomed. Health Inform.*, vol. 24, no. 2, pp. 515–523, Feb. 2020, doi: 10.1109/JBHI.2019.2911367.

[13] S. Mousavi, F. Afghah, and U. R. Acharya, “Inter- and intra- patient ECG heartbeat classification for arrhythmia detection: a sequence to sequence deep learning approach,” *ArXiv181207421 Phys. Q-Bio*, Mar. 2019, Accessed: Mar. 18, 2020. [Online]. Available: http://arxiv.org/abs/1812.07421.

[14] “Resampling strategies for imbalanced datasets.” https://kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets (accessed Apr. 12, 2020).

[15] M. Zihlmann, D. Perekrestenko, and M. Tschannen, “Convolutional Recurrent Neural Networks for Electrocardiogram Classification,” in *Computing in Cardiology*, Sep. 2017, doi: 10.22489/CinC.2017.070-060.

[16] J. Bergstra, D. Yamins, and D. D. Cox, “Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures,” p. 9.

[17] G. Van Steenkiste, G. van Loon, and G. Crevecoeur, “Transfer Learning in ECG Classification from Human to Horse Using a Novel Parallel Neural Network Architecture,” *Sci. Rep.*, vol. 10, no. 1, p. 186, Dec. 2020, doi: 10.1038/s41598-019-57025-2.

4.0 Appendix

Model 3: Trained on non-augmented data set

Options resulting from optimzation see Figure 3 for model characteristics

```
{'Activation': 0,
'Activation_1': 0,
'Activation_2': 0,
'Activation_3': 0,
'Activation_4': 0,
'Activation_5': 1,
'Activation_6': 1,
'Activation_7': 0,
'Activation_8': 0,
'Dense': 2,
'Dense_1': 2,
'Dense_2': 2,
'Dense_3': 1,
'Dense_4': 1,
'Dense_5': 1,
'Dense_6': 1,
'Dense_7': 0,
'Dense_8': 1,
'Dense_9': 3,
'Dropout': 0.3265904623792442,
'Dropout_1': 0.3431590875721917,
'Dropout_2': 0.25572580783477367,
'Dropout_3': 0.6975060560039738,
'Dropout_4': 0.3423233980953824,
'Dropout_5': 0.32317538374749766,
'a': 0,
'if': 0,
'if_1': 0,
'if_2': 0,
'if_3': 0}
```

```
Model: "sequential_103"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_107 (Flatten)        (None, 187)               0
_________________________________________________________________
dense_677 (Dense)            (None, 128)               24064
_________________________________________________________________
activation_636 (Activation)  (None, 128)               0
_________________________________________________________________
dropout_456 (Dropout)        (None, 128)               0
_________________________________________________________________
dense_678 (Dense)            (None, 128)               16512
_________________________________________________________________
activation_637 (Activation)  (None, 128)               0
_________________________________________________________________
dropout_457 (Dropout)        (None, 128)               0
_________________________________________________________________
dense_679 (Dense)            (None, 128)               16512
_________________________________________________________________
activation_638 (Activation)  (None, 128)               0
_________________________________________________________________
dropout_458 (Dropout)        (None, 128)               0
_________________________________________________________________
dense_680 (Dense)            (None, 64)                8256
_________________________________________________________________
activation_639 (Activation)  (None, 64)                0
_________________________________________________________________
dropout_459 (Dropout)        (None, 64)                0
_________________________________________________________________
dense_681 (Dense)            (None, 5)                 325
=================================================================
Total params: 65,669
```

Model 3: Trained on augmented data set

Options resulting from optimzation see Figure 3 for model characteristics

```
{'Activation': 0,
'Activation_1': 1,
'Activation_2': 0,
'Activation_3': 0,
'Activation_4': 0,
'Activation_5': 0,
'Activation_6': 0,
'Activation_7': 0,
'Activation_8': 1,
'Dense': 2,
'Dense_1': 0,
'Dense_2': 2,
'Dense_3': 1,
'Dense_4': 2,
'Dense_5': 1,
'Dense_6': 0,
'Dense_7': 0,
'Dense_8': 0,
'Dense_9': 1,
'Dropout': 0.4959326042904314,
'Dropout_1': 0.5162300963476361,
'Dropout_2': 0.4169915806152422,
'Dropout_3': 0.25301754583882874,
'Dropout_4': 0.2515810749300903,
'Dropout_5': 0.516131118961639,
'a': 1,
'if': 0,
'if_1': 0,
'if_2': 1,
'if_3': 0}
```

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten_135 (Flatten)        (None, 187)               0
_________________________________________________________________
dense_917 (Dense)            (None, 128)               24064
_________________________________________________________________
activation_815 (Activation)  (None, 128)               0
_________________________________________________________________
dropout_582 (Dropout)        (None, 128)               0
_________________________________________________________________
dense_918 (Dense)            (None, 32)                4128
_________________________________________________________________
activation_816 (Activation)  (None, 32)                0
_________________________________________________________________
dropout_583 (Dropout)        (None, 32)                0
_________________________________________________________________
dense_919 (Dense)            (None, 128)               4224
_________________________________________________________________
activation_817 (Activation)  (None, 128)               0
_________________________________________________________________
dropout_584 (Dropout)        (None, 128)               0
_________________________________________________________________
dense_920 (Dense)            (None, 64)                8256
_________________________________________________________________
activation_818 (Activation)  (None, 64)                0
_________________________________________________________________
dropout_585 (Dropout)        (None, 64)                0
_________________________________________________________________
dense_921 (Dense)            (None, 32)                2080
_________________________________________________________________
activation_819 (Activation)  (None, 32)                0
_________________________________________________________________
dense_922 (Dense)            (None, 5)                 165
=================================================================
Total params: 75,333
```

```python
# Written by Pawel Kudzia

import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

from sklearn.utils import resample
from sklearn.utils import class_weight

from sklearn.model_selection import (TimeSeriesSplit, KFold, ShuffleSplit,
                                     StratifiedKFold, GroupShuffleSplit,
                                     GroupKFold, StratifiedShuffleSplit)

from sklearn.metrics import classification_report, roc_auc_score, accuracy_score, precision_score, recall_score

from tensorflow.keras.layers import Input, Conv1D, MaxPool1D, Activation, Add, Dense, Flatten
from tensorflow.keras.models import Model,load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import plot_model
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.metrics import Precision
from tensorflow.keras.layers import Input,Dropout, LSTM, Bidirectional,BatchNormalization, Dense,Activation
from tensorflow.keras.models import Sequential, Model

from tensorflow.keras.callbacks import EarlyStopping,LearningRateScheduler, ModelCheckpoint

from sklearn.metrics import roc_curve, auc,roc_auc_score

from sklearn.metrics import confusion_matrix

#  CUSTOM Functions imported from folder
from plot_Conf_Matrix import plot_Conf_Matrix
from make_model_1 import make_model_1 # This is the biorix Architeture  Model 1 in submitted report
from make_model_2_RNN import make_model_2_RNN# this is a simple RNN model


from hyperopt import Trials, STATUS_OK, tpe
from hyperas import optim
from hyperas.distributions import choice, uniform


import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter(action='ignore', category=FutureWarning)

# %% Modelling Parameters and Variables CHANGE PARAMETERS HERE AS NEEDED
saveData = 0

epochs = 1
batch = 64
patience = 10
verbose = 1
ModelSaveName = 'Model1CNN_UsingNotAugentedData.h5' # Change this to what you want to save the modoel as
Model2_RNN = 0
Model1_CNN =1
Model3_OptimizationModel = 0

# %% Class labels
classesMIT={0:"Normal",
        1:"Artial Premature",
        2:"Premature Ventricular Contraction",
        3:"Fusion of Centricular and Normal",
        4:"Fusion of paced and Normal"}


classePTB={0:"Normal", 1:"Abnormal (MI)"}

# %% Let define all our smaller nested functions here

def plot_learning(history):
    plt.subplot(211)
    plt.plot(history.history['accuracy'])
    plt.legend(["accuracy"])
    plt.subplot(212)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'], label = "val_loss")
    plt.legend(["loss", "val_loss"])
    plt.show()

def outputMetrics(X_test,Y_Test,ModelX, batch) :

    if 1:
        X_test = X_test.reshape(len(X_test), X_test.shape[1],1)
    else:
        X_test =X_test

    Y_Test=to_categorical(Y_Test)

    PredictionModel=ModelX.predict(X_test, batch_size=batch, verbose=1)

    TrueValue=np.argmax(Y_Test, axis=1)
    Prediction=np.argmax(PredictionModel,axis=1)

    print(classification_report(TrueValue, Prediction))

    return (Prediction,TrueValue)


def roc_auc_score_multiclass(actual_class, pred_class, average = "macro"):

  #creating a set of all the unique classes using the actual class list
  unique_class = set(actual_class)
  roc_auc_dict = {}
  for per_class in unique_class:
    #creating a list of all the classes except the current class
    other_class = [x for x in unique_class if x != per_class]

    #marking the current class as 1 and all other classes as 0
    new_actual_class = [0 if x in other_class else 1 for x in actual_class]
    new_pred_class = [0 if x in other_class else 1 for x in pred_class]

    #using the sklearn metrics method to calculate the roc_auc_score
    roc_auc = roc_auc_score(new_actual_class, new_pred_class, average = average)
    roc_auc_dict[per_class] = roc_auc

    return roc_auc_dict

def classification_report_csv(report):
    report_data = []
    lines = report.split('\n')
    for line in lines[2:-3]:
        row = {}
        row_data = ' '.join(line.split())
        row_data = row_data.split(' ')
        row['class'] = row_data[0]
        row['precision'] = float(row_data[1])
        row['recall'] = float(row_data[2])
        row['f1_score'] = float(row_data[3])
        row['support'] = float(row_data[4])
        report_data.append(row)
    dataframe = pd.DataFrame.from_dict(report_data)
    dataframe.to_csv('classification_report.csv', index = False)

    return()

def NetworkModel(ModelInput,TrainingDataX,TrainingDataY,ValidationX,ValidationY,TestSetX,TestSetY, epochs,batch,patience,verbose,ModelSaveName,ComputeWeight, RNN,CNN,Optimzation):

    # he learning hyper paramters are just set to what is  reccomnded adjusting these paramters will not be the goal of this project.
    # these specifc paramaters were taken from the paper "ECG Heartbeat Classification: A Deep Transferable Representation"
    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(0.001, decay_steps=10000, decay_rate=0.75)
    adam = Adam(learning_rate=lr_schedule, beta_1=0.9, beta_2=0.999, amsgrad=False)


    if ComputeWeight:
        class_weights = class_weight.compute_class_weight('balanced',
                                                          np.unique(TrainingDataY),
                                                          TrainingDataY)
    else:
        class_weights =[]

    # We will use the sparse_C_C and the optimzation function and look at accruacy for now
    ModelInput.compile(optimizer=adam, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Define some early stopping criteria here
    callb1= EarlyStopping(monitor='val_loss', mode='min', verbose=verbose,patience=patience) # lets monitor the loss in the opt function
    callb2 = ModelCheckpoint(filepath=ModelSaveName, monitor='val_loss', save_best_only=True) # Lets save it everytime it imrpoves as well

    if RNN: # fit setup for RNN network

                history = ModelInput.fit(TrainingDataX,
                                         TrainingDataY,
                                         epochs=epochs,
                                         batch_size=batch,
                                         validation_data=(ValidationX, ValidationY),
                                         verbose=verbose,
                                         class_weight = class_weights,
                                         callbacks=[callb1,callb2])

    elif CNN: # fit setup for CNN network

                history = ModelInput.fit(np.expand_dims(TrainingDataX, axis=2),
                                        TrainingDataY,
                                        epochs=epochs,
                                        batch_size=batch,
                                        validation_data=(np.expand_dims(ValidationX, axis=2), ValidationY),
                                        verbose=verbose,
                                        class_weight = class_weights,
                                        callbacks=[callb1,callb2])

    plot_learning(history)
    ModelInput.save(ModelSaveName)
    print('The model has been saved into your directory')


    return(ModelInput,history)

def optimization_model(TrainingDataX, TrainingDataY, ValidationX, ValidationY,X_Test_MIT,Y_Test_MIT):

    ModelForTraining = Sequential()
    ModelForTraining.add(Flatten(input_shape=(187,1)))

    ModelForTraining.add(Dense({{choice([32,64,128])}}, input_shape=(187,1)))
    ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))
    ModelForTraining.add(Dropout({{uniform(0.25, 0.75)}}))

    ModelForTraining.add(Dense({{choice([32,64,128])}}, input_shape=(187,1)))

    ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))
    ModelForTraining.add(Dropout({{uniform(0.25, 0.75)}}))

    ModelForTraining.add(Dense({{choice([32,64,128])}}, input_shape=(187,1)))
    ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))
    ModelForTraining.add(Dropout({{uniform(0.25, 0.75)}}))

    ModelForTraining.add(Dense({{choice([32,64,128])}}, input_shape=(187,1)))
    ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))
    ModelForTraining.add(Dropout({{uniform(0.25, 0.75)}}))
```

```python


    if ({{choice(['two', 'three'])}}) == 'three':
        ModelForTraining.add(Dense({{choice([32,64,128])}}))
        ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))
        ModelForTraining.add(Dropout({{uniform(0.25, 0.75)}}))


    if ({{choice(['two', 'three'])}}) == 'three':
        ModelForTraining.add(Dense({{choice([32,64,128])}}))
        ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))
        ModelForTraining.add(Dropout({{uniform(0.25, 0.75)}}))


    if ({{choice(['two', 'three'])}}) == 'three':
        ModelForTraining.add(Dense({{choice([32,64,128])}}))
        ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))


    if ({{choice(['two', 'three'])}}) == 'three':
        ModelForTraining.add(Dense({{choice([32,64,128])}}))
        ModelForTraining.add(Activation({{choice(['relu', 'sigmoid'])}}))


    ModelForTraining.add(Dense(5,activation='softmax'))

    # in pilot testing i was experimenting with optimizing Model 1 to understand if the authors in the paper used the best paramters i.e if i could achieve better results
    # although this was a good effort the processing was simply too slow on my hardware to make this a

    # input_shape = (187, 1)
    # I = Input(input_shape)
    # C = Conv1D(filters=32, kernel_size=5 ,strides=1)(I)

    # a = {{choice(['relu', 'sigmoid','softmax'])}}

    # C11 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(C)
    # R11 = Activation(activation=a)(C11)
    # C12 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R11)
    # A11 = Add()([C12, C])
    # R12 = Activation(activation=a)(A11)
    # M11 = MaxPool1D(pool_size=5, strides=2)(R12)

    # C21 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M11)
    # R21 = Activation(activation= a)(C21)
    # C22 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R21)
    # A21 = Add()([C22, M11])
    # R22 = Activation(activation=a)(A21)
    # M21 = MaxPool1D(pool_size=5, strides=2)(R22)

    # C31 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M21)
    # R31 = Activation(activation=a)(C31)
    # C32 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R31)
    # A31 = Add()([C32, M21])
    # R32 = Activation(activation=a)(A31)
    # M31 = MaxPool1D(pool_size=5, strides=2)(R32)

    # C41 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M31)
    # R41 = Activation(activation=a)(C41)
    # C42 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R41)
    # A41 = Add()([C42, M31])
    # R42 = Activation(activation=a)(A41)
    # M41 = MaxPool1D(pool_size=5, strides=2)(R42)

    # C51 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M41)
    # R51 = Activation(activation=a)(C51)
    # C52 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R51)
    # A51 = Add()([M41,C52])
    # R52 = Activation(activation=a)(A51)
    # M51 = MaxPool1D(pool_size=5, strides=2)(R52)

    # F1 = Flatten()(M51)

    # D1 = Dense({{choice([32,64,128,256])}})(F1)
    # R6 = Activation(activation={{choice(['relu', 'sigmoid'])}})(D1)
    # D2 = Dense({{choice([32,64,128,256])}})(R6)
    # D3 = Dense(5)(D2)
    # O = Activation(activation='softmax')(D3)

    # ModelForTraining = Model(inputs=I, outputs=O)

    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(0.001, decay_steps=10000, decay_rate=0.75)
    adam = Adam(learning_rate=lr_schedule, beta_1=0.9, beta_2=0.999, amsgrad=False)

    class_weights = class_weight.compute_class_weight('balanced',
                                                     np.unique(TrainingDataY),
                                                     TrainingDataY)

    # class_weights = {0: 1.,1: 1.5,2: 1.5, 3:1.5,4:1.5} Experimenting here with different weights

    # We will use the sparse_C_C and the optimzation function and look at accruacy for now
    ModelForTraining.compile(optimizer=adam, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Define some early stopping criteria here (this was not used for optimzation as it resulted in poor results)
    # callb1= EarlyStopping(monitor='val_loss', mode='min', verbose=1,patience=10) # lets monitor the loss in the opt function
    # callb2 = ModelCheckpoint(filepath=ModelSaveName, monitor='val_loss', save_best_only=True) # Lets save it everytime it imrpoves as well

    history = ModelForTraining.fit(np.expand_dims(TrainingDataX, axis=2),
                                   TrainingDataY,
                                   epochs=30,
                                   batch_size=64,
                                   validation_data=(np.expand_dims(ValidationX, axis=2), ValidationY),
                                   verbose=0,
                                   class_weight = class_weights)
                                   # callbacks=[callb1])

    test_loss, test_acc = ModelForTraining.evaluate(np.expand_dims(X_Test_MIT,axis=2), Y_Test_MIT, batch_size=64)
    #get the highest validation accuracy of the training epochs
   #  validation_acc = np.amax(history.history['val_accuracy']) #  Not used

    print('Best validation acc of epoch:', test_acc)

    return {'loss': -test_acc, 'status': STATUS_OK, 'model': ModelForTraining}

def CalculateModelMetrics(ModelOutput,History,TestSetX,TestSetY,batch, Naming):

    # function to make excel sheets and plot some of the data for further inspection

    #       results = ModelOutput.evaluate(np.expand_dims(TestSetX, axis=2), TestSetY, batch_size=batch)
    #       print(f"The accuracy on the testing set is {np.round(results[1]*100,1)}%")

    [Prediction,TrueValue,] =outputMetrics(TestSetX,TestSetY,ModelOutput,batch)
    clsf_reportClassification = pd.DataFrame(classification_report(y_true =TrueValue, y_pred = Prediction, output_dict=True)).transpose()

    confusionM  = confusion_matrix(TrueValue, Prediction)
    clsf_report = pd.DataFrame(confusionM)

    plt.figure()
    ReportName = '%s' %(Naming)

    plot_Conf_Matrix(confusionM,classes=['N', 'S', 'V', 'F', 'Q'],normalize=True,title=ReportName)
    #plot_Conf_Matrix(confusionM,classes=['N', 'S'],normalize=True,title=ReportName)
    plt.show()

    print("\nLogistic Regression")
    # assuming your already have a list of actual_class and predicted_class from the logistic regression classifier
    lr_roc_auc_multiclass = roc_auc_score_multiclass(TrueValue, Prediction)
    print(lr_roc_auc_multiclass)

    hist_df = pd.DataFrame(History.history)

    hist_csv_file = ' History_%s'  %(Naming)
    with open(hist_csv_file, mode='w') as f:
        hist_df.to_csv(f)

    jsonName = Naming.replace('.csv', '')
    hist_json_file =  ' History_%s.json'  %(jsonName)
    with open(hist_json_file, mode='w') as f:
        hist_df.to_json(f)

    ReportName = 'ConfusionMatrix%s' %(Naming)
    clsf_report.to_csv(ReportName, index= True)

    ReportName = 'ClassificationReport%s' %(Naming)
    clsf_reportClassification.to_csv(ReportName, index= True)
    return()


def data_curation_MIT():
    # this function imports the data, performs the augmentation etc.

    SampleCounts= 10000
    noiseLevel = 0.03

    csv_MIT_train = pd.read_csv(r'E:\1_PAWEL\ENSC_project\heart_beatdata\mitbih_train.csv',header=None)
    csv_MIT_test= pd.read_csv(r'E:\1_PAWEL\ENSC_project\heart_beatdata\mitbih_test.csv',header=None)
    DataSet_MIT= pd.concat([csv_MIT_train, csv_MIT_test], axis=0) # put it all into one structure
    Values_MIT = DataSet_MIT.values

    X_Data_MIT = Values_MIT[:,:-1]# remove the last column which is the calssification
    Y_Data_MIT = Values_MIT[:,-1] # keep the last column which will now be used for the Class Categroization

    StratifyData = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=100)
    #  what we are trying to accomplish here is to split up the data such that the number of samples in each class is somewhat evenly distrubuted
    #  Since we already have a large class imbalance , this might be an issue if it is not addressed from the start. Lets split it up such that the
    #  test_size get 20% of the data (but catigorigcal samples evenly distrubuted)

    StratifyData.get_n_splits(X_Data_MIT, Y_Data_MIT)

    print(StratifyData)

    for train_index, test_index in StratifyData.split(X_Data_MIT, Y_Data_MIT):
        X_temp, X_Test_MIT = X_Data_MIT[train_index], X_Data_MIT[test_index]
        y_temp, Y_Test_MIT = Y_Data_MIT[train_index], Y_Data_MIT[test_index]

    # Now lets further split up our training data such that 20% of it goes to the validation sample
    #[X_Train_MIT, X_Valid_MIT, Y_Train_MIT, Y_Valid_MIT] = train_test_split(X_temp, y_temp, test_size=0.2)

    StratifyData = StratifiedShuffleSplit(n_splits=5, test_size=0.20, random_state=100)
    #  what we are trying to accomplish here is to split up the data such that the number of samples in each class is somewhat evenly distrubuted
    #  Since we already have a large class imbalance , this might be an issue if it is not addressed from the start. Lets split it up such that the

    #  test_size get 20% of the data (but catigorigcal samples evenly distrubuted)
    StratifyData.get_n_splits(X_temp, y_temp)

    for train_index, test_index in StratifyData.split(X_temp, y_temp):
        X_Train_MIT, X_Valid_MIT = X_temp[train_index], X_temp[test_index]
        Y_Train_MIT, Y_Valid_MIT = y_temp[train_index], y_temp[test_index]

    # TrainingDataX = X_Train_MIT # TRAINING DATA
    # TrainingDataY = Y_Train_MIT# TRAINING DATA Labels

    ValidationX =X_Valid_MIT
    ValidationY =Y_Valid_MIT

    temp_X_Train_MIT = pd.DataFrame(X_Train_MIT)
    temp_Y_Train_MIT = pd.DataFrame(Y_Train_MIT)

    tempMIT_TrainingDataSet= pd.concat([temp_X_Train_MIT,temp_Y_Train_MIT], axis=1,ignore_index=True) # put it all into one structure, ignore the index and autogenerate a new one

    print(tempMIT_TrainingDataSet[187].astype(int).value_counts()) # Lets see how many there are in each category. # so we have ~50k in the majority and >5k in the minoritys
    # at this point i think it makes sense to slightly reduce the sample count on the majority and then upsample the minoristy so ...
    temp1=tempMIT_TrainingDataSet[tempMIT_TrainingDataSet[187]==1]
    temp2=tempMIT_TrainingDataSet[tempMIT_TrainingDataSet[187]==2]
    temp3=tempMIT_TrainingDataSet[tempMIT_TrainingDataSet[187]==3]
    temp4=tempMIT_TrainingDataSet[tempMIT_TrainingDataSet[187]==4]

    temp0=(tempMIT_TrainingDataSet[tempMIT_TrainingDataSet[187]==0]).sample(n=SampleCounts,random_state=1) # We can downsample this
    temp1_upsample=resample(temp1,replace=True,n_samples=SampleCounts,random_state=2) # and we can upsample all of the minority classes
    temp2_upsample=resample(temp2,replace=True,n_samples=SampleCounts,random_state=3) # one note here is the random_state =var ; this will let us reproduce the results everytime we run this
    temp3_upsample=resample(temp3,replace=True,n_samples=SampleCounts,random_state=4)
    temp4_upsample=resample(temp4,replace=True,n_samples=SampleCounts,random_state=5)

    # note that this resample approach will just randomly duplicate observations from the minority class. we might want to add some noise to
    # this data to avoid the nerual network just learning the samples. However, we need to be careful here as to not physically change he sample and its
    # classication. Therefore we will just add a bit of guassian noise

    MIT_Training_Resampled=pd.concat([temp0,temp1_upsample,temp2_upsample,temp3_upsample,temp4_upsample])
    print(MIT_Training_Resampled[187].astype(int).value_counts())

    MIT_Training_Resampled = MIT_Training_Resampled.values
    X_Data_MIT_resampled_Training = MIT_Training_Resampled[:,:-1]# remove the last column which is the calssification
    Y_Data_MIT_resampled_Training  = MIT_Training_Resampled[:,-1] # keep the last column which will now be used for the Class Categroization

    X_Data_MIT_resampled_Noise_Training = np.empty(X_Data_MIT_resampled_Training.shape)

    for i in range(0,len(X_Data_MIT_resampled_Training)):
        noise1=np.random.normal(0,noiseLevel,187)
        noise2=np.random.normal(0,noiseLevel+0.01,187)+noise1
        noise=np.random.normal(0,noiseLevel-0.01,187)+noise2
        X_Data_MIT_resampled_Noise_Training[i] = X_Data_MIT_resampled_Training[i]+noise# lets loop through each sample and add some random noise


    TrainingDataX = X_Data_MIT_resampled_Training # choose what training Dataset you want to USE (with augmnetaion or not??)
    TrainingDataY = Y_Data_MIT_resampled_Training

    return TrainingDataX, TrainingDataY, ValidationX, ValidationY,X_Test_MIT,Y_Test_MIT


# %% ------------------Plotting of data for inspection----------------

# # %% Plotting of the Initial Data for Visulization
# fig = plt.figure(figsize=(15,4))

# for i in range(0,5):
#     plt.subplot(2,3,i + 1)
#     all_samples_indexes = np.where(Y_Data_MIT == i)[0]
# #    rand_samples_indexes = np.random.randint(0, len(all_samples_indexes), 1)
#     rand_samples = X_Data_MIT[all_samples_indexes[5]] # plot the 5th of each class
#     plt.plot(rand_samples.transpose())
#     plt.ylim(-0.1, 1.3)

#     if  i ==0 :
#         plt.ylabel('Norm Amplitude')
#     elif i ==3:
#         plt.ylabel('Norm Amplitude')
#     plt.title("Sample:  " + classesMIT[i], loc='left', fontdict={'fontsize':10}, x=0.01, y=0.80)

# if saveData == 1:
#     fig.savefig('MIT_Data_Methods.png', format='png', dpi=800)

# %%---------- Train and TEST Model on MIT DataSET -------------------

# lets load up the data for the model now
TrainingDataX, TrainingDataY, ValidationX, ValidationY,X_Test_MIT,Y_Test_MIT = data_curation_MIT()

n_classes = len(np.unique(TrainingDataY))  # how many classes are there in our data

# now pick which model we want to evaluate
if Model1_CNN:
    print('Using Model 1')
    ModelForTraining = make_model_1(n_classes) # Using the archetecture CHANGE HERE

elif Model2_RNN:
    print('Using Model 2')
    ModelForTraining = make_model_2_RNN(n_classes) # Using the archetecture CHANGE HERE

    # for the RNN model we need to reshape the input a bit for it to work
    TrainingDataX_RNN = np.reshape(TrainingDataX,(TrainingDataX.shape[0],TrainingDataX.shape[1],1))
    ValidationX_RNN= np.reshape(ValidationX,(ValidationX.shape[0],ValidationX.shape[1],1))
    TestDataX_RNN = np.reshape(X_Test_MIT,(X_Test_MIT.shape[0],X_Test_MIT.shape[1],1))

    del TrainingDataX, TestSetX, ValidationX

    TrainingDataX =TrainingDataX_RNN
    ValidationX =ValidationX_RNN
    X_Test_MIT =TestDataX_RNN

    n_classes = len(np.unique(TrainingDataY))

elif Model3_OptimizationModel:
    print('Using Model 3')
    trials = Trials()


from os import path
pathFile = (r'E:\1_PAWEL\ENSC_project\%s' % (ModelSaveName))
var = path.exists(pathFile)
if not var:

      if Model3_OptimizationModel:
      # Here is the setup for the optimized model, we set max_evals to 40 mainly as a hardware contraint as it takes quite a long time.
          best_run, ModelOutput,return_space = optim.minimize(model=optimization_model,
                                                      data = data_curation_MIT,
                                                      algo=tpe.suggest,
                                                      max_evals=40,
                                                          trials=trials,
                                                          return_space = True)

            # saving it in this manner as to deal with issues of loading .h5 file after the optization. this seems to work now.
            ModelSave = trials.results[np.argmin([r['loss'] for r in trials.results])]['model']
            ModelSave.save(ModelSaveName)

        else:

            [ModelOutput, History]= NetworkModel(ModelForTraining,TrainingDataX,TrainingDataY,ValidationX,ValidationY,X_Test_MIT,Y_Test_MIT,
                                             epochs,batch,patience,verbose,ModelSaveName,ComputeWeight=False,RNN=False, CNN= True, Optimzation = False )
else:

    print('Loading preexcisting model from folder .....')
    ModelOutput = load_model(r'E:\1_PAWEL\ENSC_project\RESULTS\Model1_NoAUG\%s' % (ModelSaveName))

    jsonName = ModelSaveName.replace('.h5', '')
    hist_json_file =  ' History_%s.json'  %(jsonName)
    with open(hist_json_file, 'r') as json_file:
        History= json_file.read()

 # this will not spit out the results, csv files etc into the folder to further look at
History = ModelOutput.history
CalculateModelMetrics(ModelOutput,History,X_Test_MIT,Y_Test_MIT,batch, Naming='MIT_resampled_Training_Weights.csv')  #


plt.figure(figsize=(5, 5))
plt.plot(History.history['accuracy'])
plt.plot(History.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# summarize history for loss
# Plot non-normalized confusion matrix
plt.figure(figsize=(5, 5))
plt.plot(History.history['loss'])
plt.plot(History.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()

# %% Exploring Transfer Learning

# lets load in the other data to explore transfer learning
csv_PTB_Normal = pd.read_csv(r'E:\1_PAWEL\ENSC_project\heart_beatdata\ptbdb_normal.csv',header=None)
csv_PTB_AbNormal = pd.read_csv(r'E:\1_PAWEL\ENSC_project\heart_beatdata\ptbdb_abnormal.csv',header=None)
DataSet_PTB = pd.concat([csv_PTB_Normal, csv_PTB_AbNormal], axis=0)
Values_PTB = DataSet_PTB.values # get the values

X_Data_PTB = Values_PTB[:,:-1] # remove the last column which is the calssification
Y_Data_PTB = Values_PTB[:,-1] # keep the last column which will now be used for the Class Categroization

StratifyData = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=50)
#  what we are trying to accomplish here is to split up the data such that the number of samples in each class is somewhat evenly distrubuted
#  Since we already have a large class imbalance , this might be an issue if it is not addressed from the start. Lets split it up such that the
#  test_size get 20% of the data (but catigorigcal samples evenly distrubuted)
StratifyData.get_n_splits(X_Data_PTB, Y_Data_PTB)

print(StratifyData)

for train_index, test_index in StratifyData.split(X_Data_PTB, Y_Data_PTB):
    X_temp, X_Test_PTB = X_Data_PTB[train_index], X_Data_PTB[test_index]
    y_temp, Y_Test_PTB = Y_Data_PTB[train_index], Y_Data_PTB[test_index]

uniqueValues, occurCountTrain = np.unique(y_temp, return_counts=True)
uniqueValues, occurCountValid = np.unique(Y_Test_PTB, return_counts=True)

print(occurCountTrain)
print(occurCountValid)

for train_index, test_index in StratifyData.split(X_temp, y_temp):

    X_Train_PTB, X_Valid_PTB = X_temp[train_index], X_temp[test_index]
    Y_Train_PTB, Y_Valid_PTB = y_temp[train_index], y_temp[test_index]

# # Plotting PTB database 1 sample from each cat

# plt.figure(figsize=(15,4))
# for i in range(0,2):
#     plt.subplot(1,2,i + 1)
#     all_samples_indexes = np.where(Y_Data_PTB == i)[0]
#     rand_samples_indexes = np.random.randint(0, len(all_samples_indexes), 1)
#     rand_samples = X_Data_PTB[all_samples_indexes[5]]
#     plt.plot(rand_samples.transpose())
#     plt.title("Sample: " + classePTB[i], loc="left", fontdict={'fontsize':10})


# change this line to change which model we want to evaluate
ModelOutputTransfer = ModelOutput # CHANGE THIS TO WHAT YOU WANT TO TEST

# lets build the new transfer learning layers ...
D1 = Dense(32)(ModelOutputTransfer.output)
D2 = Dense(32)(D1)

O = Dense(2, activation='softmax')(D2)
modelTransfer = Model(inputs=ModelOutputTransfer.input, outputs=O) # we have our new model

for layer in modelTransfer.layers[:-3]:  # change this number to explore changing more layers in case of report i looked at -3,-5, -8
    layer.trainable = False

for layer in modelTransfer.layers[-3:]: # change this number to explore changing more layers in case of report i looked at -3,-5, -8
    layer.trainable = True
```

```python
# we can use the function developed before and run it no on the new transfer learning model.
[modelTransferOutput, History]= NetworkModel(modelTransfer,X_Train_PTB,Y_Train_PTB,X_Valid_PTB,Y_Valid_PTB,X_Test_PTB,Y_Test_PTB,
                                    epochs,batch,patience,verbose,ModelSaveName,ComputeWeight=False,RNN=False, CNN= True, Optimzation = False)

# lets output and look at how the transfer learning did on the novel dataset...
CalculateModelMetrics(modelTransferOutput,History,X_Test_PTB,Y_Test_PTB,batch, 'Transfer')
```

```python
from tensorflow.keras.layers import Input, Conv1D, MaxPool1D, Activation, Add, Dense, Flatten
from tensorflow.keras.models import Model

def make_model_1(final_layer_size):
    input_shape = (187, 1)
    I = Input(input_shape)
    C = Conv1D(filters=32, kernel_size=5,strides=1)(I)

    C11 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(C)
    R11 = Activation(activation='relu')(C11)
    C12 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R11)
    A11 = Add()([C12, C])
    R12 = Activation(activation='relu')(A11)
    M11 = MaxPool1D(pool_size=5, strides=2)(R12)

    C21 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M11)
    R21 = Activation(activation='relu')(C21)
    C22 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R21)
    A21 = Add()([C22, M11])
    R22 = Activation(activation='relu')(A21)
    M21 = MaxPool1D(pool_size=5, strides=2)(R22)

    C31 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M21)
    R31 = Activation(activation='relu')(C31)
    C32 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R31)
    A31 = Add()([C32, M21])
    R32 = Activation(activation='relu')(A31)
    M31 = MaxPool1D(pool_size=5, strides=2)(R32)

    C41 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M31)
    R41 = Activation(activation='relu')(C41)
    C42 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R41)
    A41 = Add()([C42, M31])
    R42 = Activation(activation='relu')(A41)
    M41 = MaxPool1D(pool_size=5, strides=2)(R42)

    C51 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(M41)
    R51 = Activation(activation='relu')(C51)
    C52 = Conv1D(filters=32, kernel_size=5,strides=1,padding='same')(R51)
    A51 = Add()([M41,C52])
    R52 = Activation(activation='relu')(A51)
    M51 = MaxPool1D(pool_size=5, strides=2)(R52)

    F1 = Flatten()(M51)

    D1 = Dense(32)(F1)
    R6 = Activation(activation='relu')(D1)
    D2 = Dense(32)(R6)
    D3 = Dense(final_layer_size)(D2)
    O = Activation(activation='softmax')(D3)

    return Model(inputs=I, outputs=O)
```

```python
from tensorflow.keras.layers import Input,Dropout, LSTM, Bidirectional,BatchNormalization, Dense
from tensorflow.keras.models import Sequential


def make_model_2_RNN(final_layer_size):


    Model = Sequential()
    Model.add(Dense(32, input_shape=(187,1)))
    Model.add(Bidirectional(LSTM(64,input_shape = (187,1))))
    Model.add(Dropout(rate =0.25))
    Model.add(BatchNormalization())
    Model.add(Dense(32, activation='relu'))
    Model.add(Dropout(rate =0.25))
    Model.add(BatchNormalization())
    Model.add(Dense(final_layer_size,activation = 'sigmoid'))



    return Model
```