# Project 2 - Time Series and Representation Learning

## Interpretable and Explainable Classification for Medical Data

**Kári Rögnvaldsson** (kroegnvaldss@student.ethz.ch)
**Jacob Hunecke** (jhunecke@student.ethz.ch)

Group **26**
May 18, 2024

## 1 Part 1: Supervised Learning on Time Series

### 1.1 Question 1: Exploratory Data Analysis

This first part of the project was about classifying digital heart electrocardiograms (ECGs) into abnormal and normal diagrams. An ECG is a time series produced by recording a heart's electral activity in voltage over some time period. The dataset that will be used in this first part is a dataset compiled by the National Metrology Institute in Germany (PTB) of 14550 patients that are classified into 9 different categories, 8 of which are heart related problems, and the last one being a healthy baseline patient. The goal of this part is to be able to predict whether a patient has one of the 8 heart diseases or is healthy, i.e. only prediction of two classes. Figure 1 shows examples of two healthy patients and two patients diagnosed with heart disease. To the human eye, it is hard to see a difference between the two types of time series, which supports the idea of using machine learning models to learn to differentiate between the two.

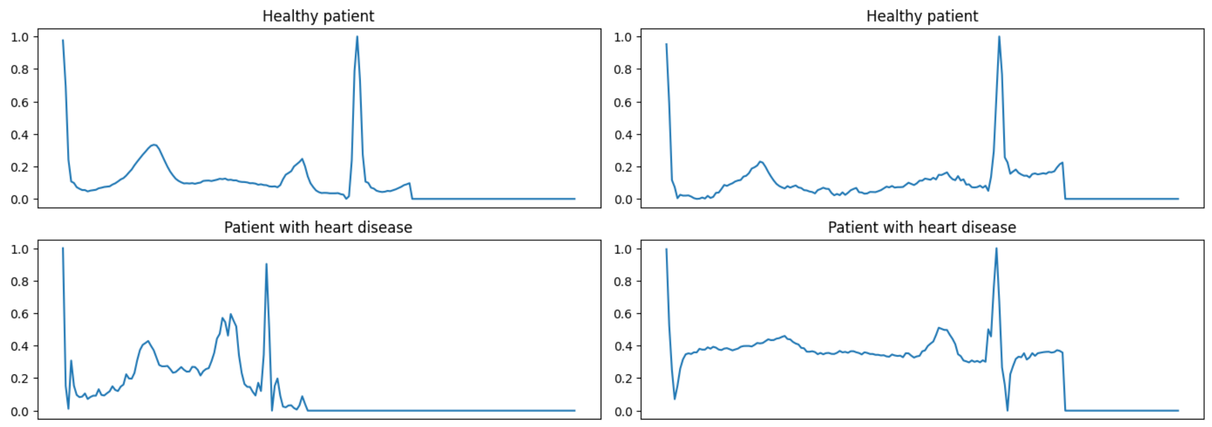


Figure 1: Example of ECGs from the dataset, two from healthy patients and two from patients with heart disease.

Figure 2 shows the distribution of the labels in the dataset, showing that the data is quite imbalanced, with around 72% of the examples being patients with heart disease. Due to the class imbalance, we will measure the quality of our models using the balanced accuracy metric.

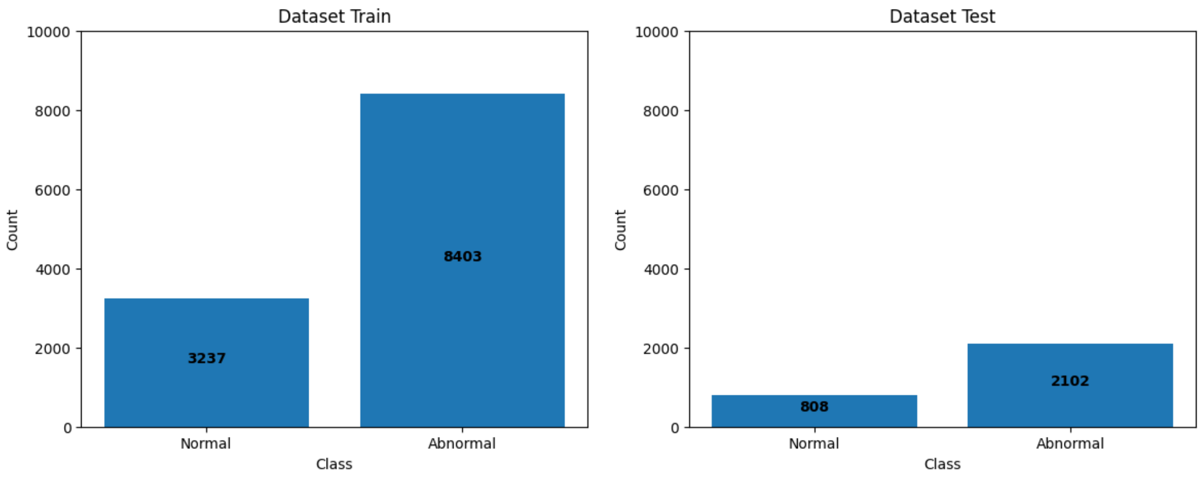


Figure 2: Label distribution of the PTB dataset.

## 1.2 Question 2: Classic Machine Learning Methods

Both logistic regression and random forest algorithms were trained on the training data, and the test set performance of those is summarized in Table 1 (Orig. feat.). We note that the random forest algorithm performs considerably better than the logistic regression model. The following additional features were added to the training data as an attempt to make it richer (ignoring zero-values as the data is padded):

- Mean of the time series values
- Standard Deviation
- Max
- Min
- Range (Min - Max)
- 25th quantile of the values in the time series
- 75th quantile of the values in the time series
- Inter-quartile range (75th quantile - 25th quantile)
- Kurtosis
- Skew

Using these new features, the same models were trained again, and the results are summarized in Table 1 (Under Add. feat.). It is interesting to see that the testing balanced accuracy of the logistic regression model was improved by almost 5%, while the balanced accuracy of the random forest algorithm increased by less than 1%.

Both random forests and logistic regression are classical ML methods that have been around for a long time. The main benefit of logistic regression is the extremely good interpretability of those models, being able to exactly tell how much each feature matters to the final prediction and how much you would need to increase a feature to change the classification of the models. The drawback of logistic regression models is that because they are so simple and interpretable, they are way less expressive and have less predictive power.

Random forests is a non-linear classification method and is very expressive. It is quite computationally inexpensive compared to deep learning models, and it is also quite robust, as it calculates predictions by averaging across multiple trees. The main drawback of the random forest models is that is is not very interpretable, since it is averaging across multiple trees. One can investigate feature importance in the models, but they are not nearly as interpretable as logistic regression models.

Table 1: Comparison of Balanced Accuracy of different classical ML models

| Model | Balanced Accuracy |
|---|---|
| **Logistic Regression** | |
| Orig. feat. | 74.8% |
| Add. feat. | 79.4% |
| **Random Forest** | |
| Orig. feat. | 95.0% |
| Add. feat. | 95.8% |

## 1.3 Question 3: Recurrent Neural Networks

An LSTM recurrent neural network with 50 hidden dimensions and 5 layers, ending with two linear layers of size 150 and 1 respectively was trained on the dataset. The architecture is depicted in the following:

```
LSTMModel(
  (lstm): LSTM(1, 50, num_layers=5, batch_first=True)
  (fc1): Linear(in_features=250, out_features=250, bias=True)
  (fc2): Linear(in_features=250, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

The model was trained for 200 epochs using binary-cross entropy loss, using minibatch training with minibatch size 100, and using the Adam optimizer with learning rate $5 \cdot 10^{-4}$. The LSTM model turned out to be very sensitive to learning rate, and therefore, the training data was shuffled and balanced before training the model. The training loop truncated each batch to be of the same length as the longest sequence in the batch (disregarding zero padding). The training loss and balanced accuracy over time are summarized in Figure 3, it is interesting to see that the model needed 50 epochs to learn anything better than random. This possibly indicates that the learning rate chosen for the model is a bit suboptimal. The test set performance of the LSTM methods is summarized in Table 2.

In ECG classification, it is very important to understand how the time series behaves around each data point, and therefore the unidirectional nature of LSTMs is not exactly optimal. From the way they are constructed, bidirectional LSTMs are built to support this. Additionally, bidirectional LSTMs can capture longer range dependencies in the time series as it has a bidirectional context window capturing previous and succeeding time steps.

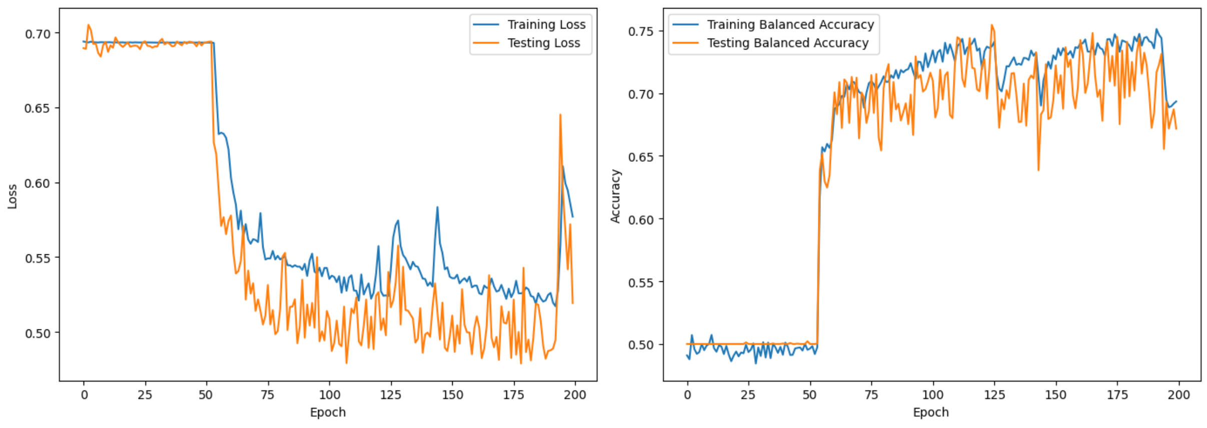


Figure 3: Training/Test Accuracy and Loss for 200 Epoch training of the unidirectional LSTM

A bidirectional LSTM with exactly the same architecture as the original LSTM except for the fact that the first linear layer had size 300 (due to the fact that the bidirectional LSTM has double the number of context vectors compared to the vanilla LSTM) was trained on the dataset with the same optimizer, loss function and learning rate. The architecture of the model was the following:

```
LSTMModel(
  (lstm): LSTM(1, 50, num_layers=5, batch_first=True, bidirectional=True)
  (fc1): Linear(in_features=500, out_features=250, bias=True)
  (fc2): Linear(in_features=250, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

The progression of the loss and balanced acuracy of the model is summarized in Figure 4 The test set balanced accuracy of the model is summarized in Table 2. We note that the bidirectional model was way less sensitive to learning rate than the vanilla LSTM and it achieved 98.1% balanced accuracy after 200 epochs. which is even better than the random forest model.

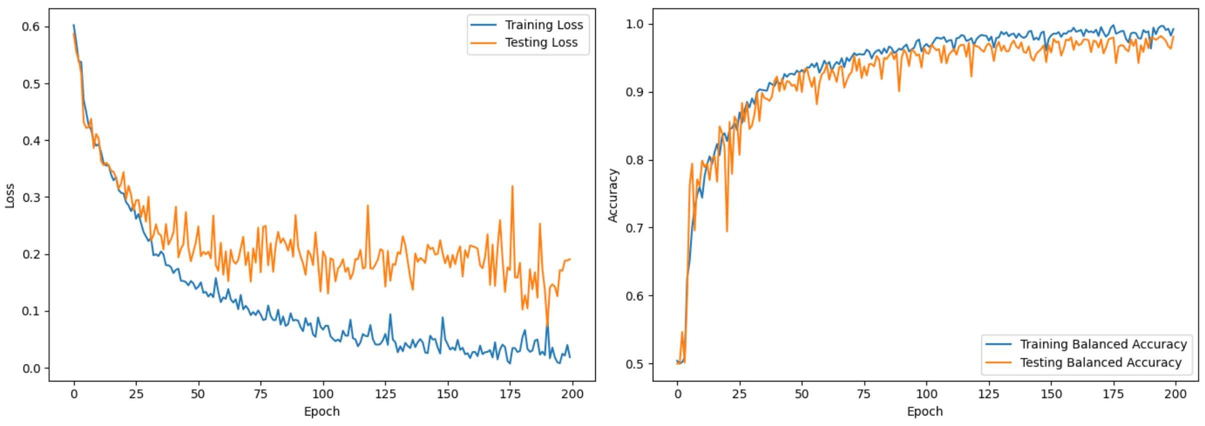


Figure 4: Training/Test Accuracy and Loss for 200 Epoch training of the bidirectional LSTM

Table 2: Comparison of balanced accuracy of the two different LSTM models

| Model | Balanced Accuracy | Epochs |
|---|---|---|
| Unidirectional LSTM | 72.7% | 200 |
| Bidirectional LSTM | 98.1% | 200 |

## 1.4 Question 4: Convolutional Neural Networks

RNNs are useful in time series prediction as they are built to handle sequential data, using hidden representations as a memory store, and are able to capture quite long range contexts. They are also useful as they can handle variable length sequences, while CNNs can only be trained on fixed length sequences. This can be problematic when the length of the time series are varying a lot. Even though CNNs are originally built to classify images, they are generally useful for identifying patterns in time series since they are almost shift invariant and therefore are able to understand anomalies in time series less dependent on where they are. CNNs are also more computationally efficient, as their training can be paralellized opposed to the recursive nature of LSTMs not allowing for paralellizability of training (because of backpropagation through time).

Both a standard CNN with two convolutional layers followed by max-pooling and ReLU activation and a CNN with two convolutional layers with residual blocks between them were trained on the PTB dataset, using binary cross-entropy loss and Adam optimization with learning rate $5 \cdot 10^{-4}$. The architecture of the standard CNN was the following:

```
CNNModel(
  (conv1): Conv1d(1, 16, kernel_size=(3,), stride=(1,), padding=(1,))
  (max_pool1): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (conv2): Conv1d(16, 32, kernel_size=(3,), stride=(1,), padding=(1,))
  (max_pool2): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (fc): LazyLinear(out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

The architecture of the CNN with residual connections was the following:

```
ResidualCNNModel(
  (conv1): Conv1d(1, 16, kernel_size=(3,), stride=(1,), padding=(1,))
  (max_pool1): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (residual_block1): ResidualBlock(
    (conv1): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (relu): ReLU()
  )
  (residual_block2): ResidualBlock(
    (conv1): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (relu): ReLU()
  )
  (conv2): Conv1d(16, 32, kernel_size=(3,), stride=(1,), padding=(1,))
  (max_pool2): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu): ReLU()
  (fc): LazyLinear(out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

The loss progression of the models are summarized in Figures 5 and 6 and the test set performance of each of the models is summarized in Table 3. The CNN with residual layers performed considerably better than the standard one, possibly because residual blocks allow for training deeper CNN classifiers and improve training stability and convergence since they improve the gradient flow of the model. Thus, CNNs with residual connections are able to achieve higher accuracy when both models are trained for the same amount of epochs.

Table 3: Comparison of balanced accuracy of the CNN models, with and without residual blocks.

| **Model** | **Balanced Accuracy** | **Epochs** |
|---|---|---|
| **CNN** | | |
| w/o res. conn. | 91.3% | 200 |
| w/ res. conn. | 94.7% | 200 |

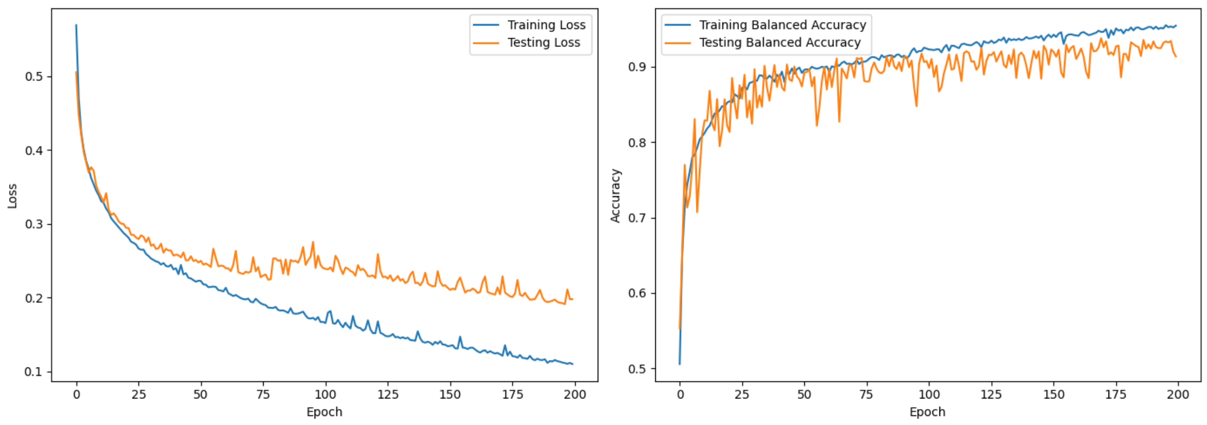


Figure 5: Training/Test Accuracy and Loss for 200 Epoch training of the CNN

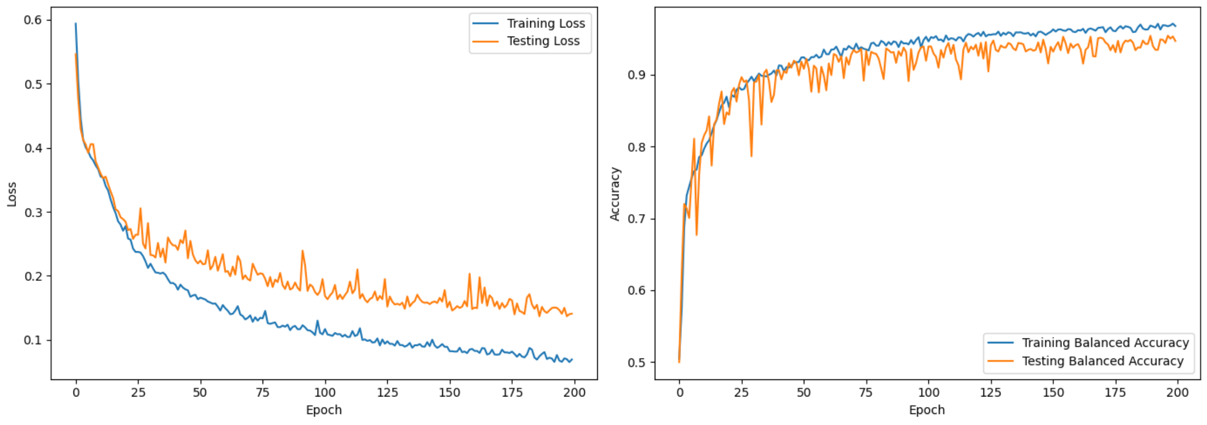


Figure 6: Training/Test Accuracy and Loss for 200 Epoch training of the CNN with residual connections

### 1.5 Question 5: Attention and Transformers

A transformer model consisting of 3 encoder layers, model dimensionality 16, with the final layer average pooling over each of the heads, was trained on the dataset. The data was prepared to be split into 47 context windows of length 4 (one last dimension was padded to the data so it would have dimensionality divisible by 4). This was done for it to be easier to visualize attention maps of the transformer network. Each batch was truncated to the maximum sequence length within the batch. The architecture of the transformer network was the following:

```
TransformerModel(
  (embedding): Linear(in_features=4, out_features=16, bias=True)
  (pos_encoder): PositionalEncoding(
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (transformer_encoder): TransformerEncoder(
    (layers): ModuleList(
      (0-2): 3 x TransformerEncoderLayer(
        (self_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(in_features=16, out_features=16, bias=True)
        )
        (linear1): Linear(in_features=16, out_features=512, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
        (linear2): Linear(in_features=512, out_features=16, bias=True)
        (norm1): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
        (norm2): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
        (dropout1): Dropout(p=0.1, inplace=False)
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
  )
  (fc): Linear(in_features=16, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

The accuracy/loss progression of the trainsformer model is summarized in Figure 7 and the test set balanced accuracy after training for 200 epochs was 92.7%. Transformers, handle sequences in parallel using attention mechanisms, enabling them to capture long-range dependencies effectively and accelerate computation. RNNs, in contrast, process sequential data by iterating through elements one at a time, making them less efficient for capturing long-term dependencies and parallelizing computation due to the recursive nature of the models.

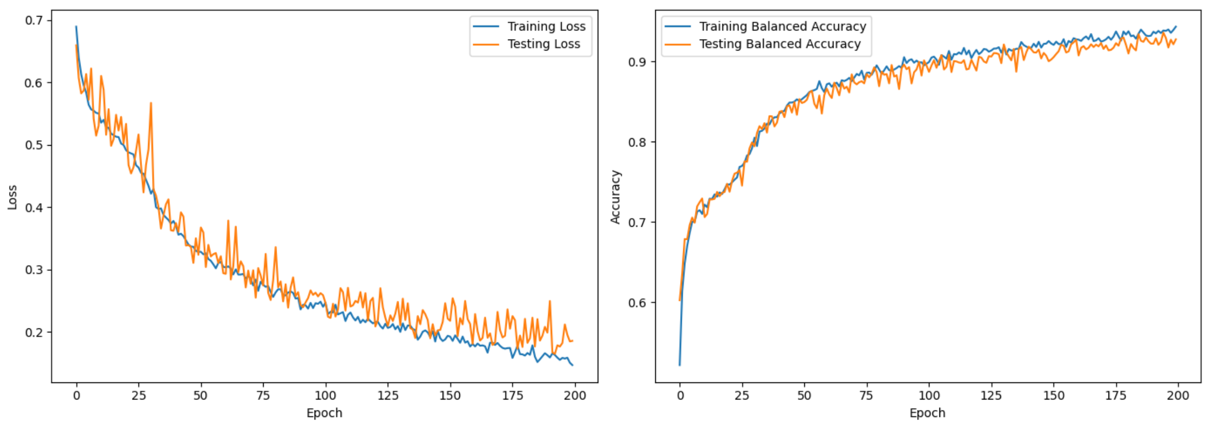


Figure 7: Training/Test Accuracy and Loss for 200 Epoch training of the Transformer

### 1.5.1 Attention Maps

For each of the 12 heads in the transformer (3 layers, 4 heads in each layer), an attention map was computed and compared for two healthy and two diagnosed patients, and the attention maps are summarized in Figures 8 and 9. The attention maps show how much attention each part of the sequence (x-axis) places to each other part of the sequence (y-axis).

We see that some of the heads, like head 2 in layer 1 seem to focus very clearly on the one peak of the time series in every series, and this peak seems to be important in many time series, which seems natural as it seems like a crucial point in the time series. The model also seems to place some emphasis on the padded part of the time series (after it goes down to zero), but since the model was trained on the padded data, it was impossible to remove the padding when computing the attention maps.

The most notable difference between healthy patients and patients with heart disease is the fact that for healthy patients, head 1 in layer 2 places emphasis on a bump in the time series in the beginning and not much else, but for patients with heart disease, it seems to focus on other parts of the sequence. Other than that, the heads seem to be looking at similar things for both types of patients, which also seems natural.

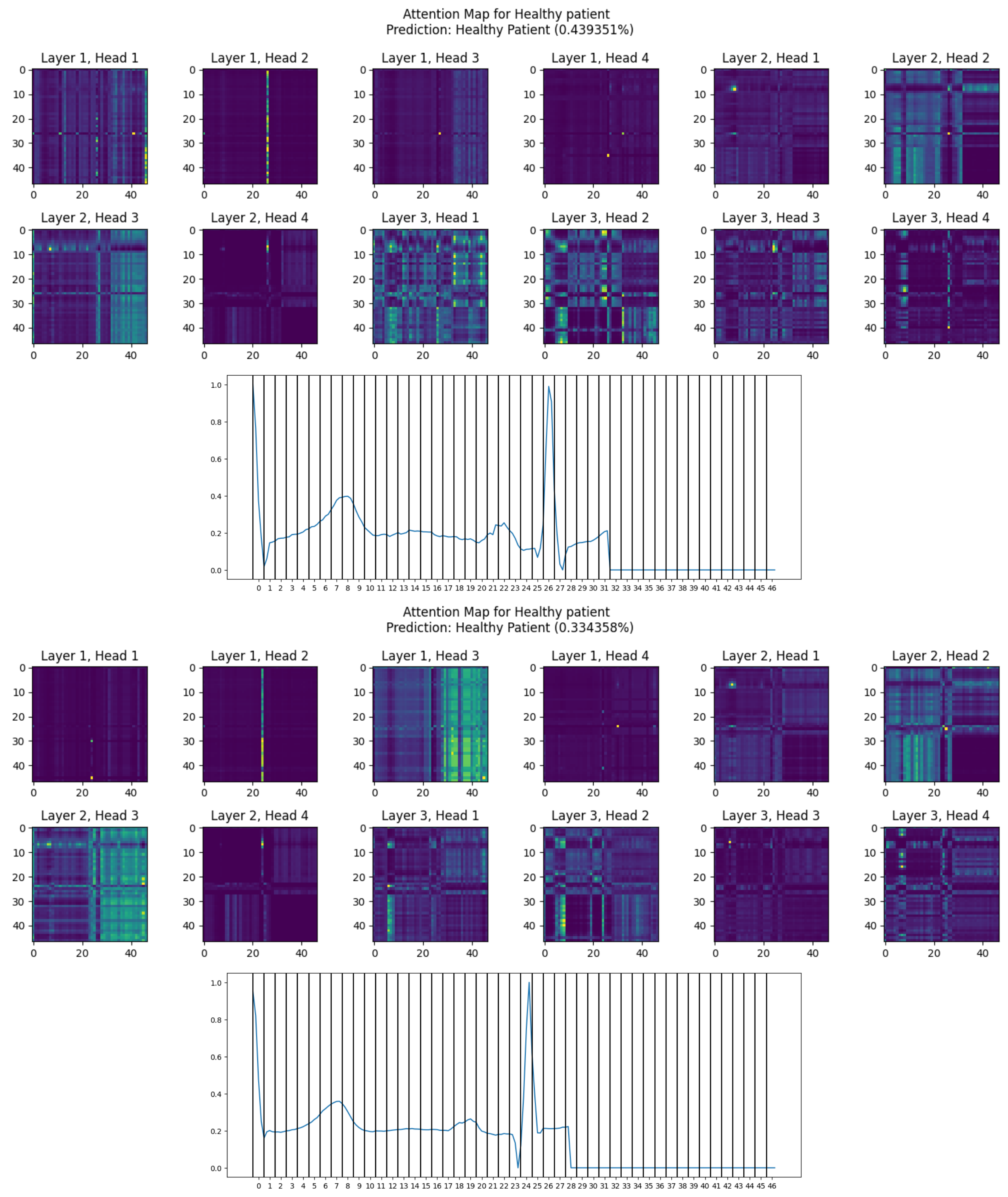


Figure 8: Attention maps for 2 healthy patients. For each patient, two plots are provided. The upper one is the attention map and the lower one is the ECG time series for the patient, split into the context windows of size 4 and labeled accordingly.

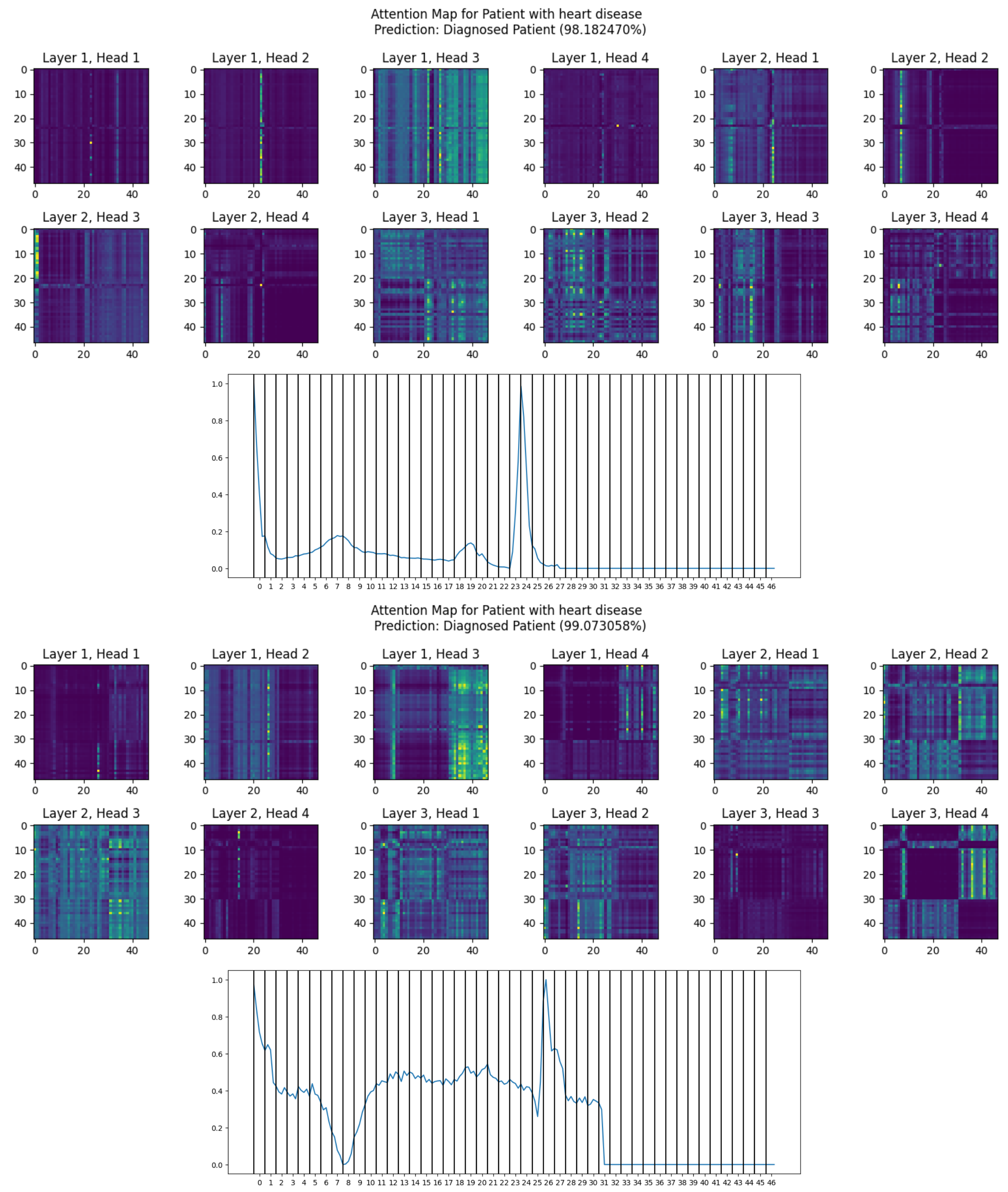


Figure 9: Attention maps for 2 patients with heart disease. For each patient, two plots are provided. The upper one is the attention map and the lower one is the ECG time series for the patient, split into the context windows of size 4 and labeled accordingly.

## 1.6 Conclusion of Part 1

To conclude this part of the report, the results of each of the models trained on the PTB dataset are summarized in Table 4.

Table 4: Comparison of balanced accuracy of different classical and deep ML models

| **Model** | **Balanced Accuracy** | **Epochs** |
|---|---|---|
| **Logistic Regression** | | |
| Orig. feat. | 74.8% | - |
| Add. feat. | 79.4% | - |
| **Random Forest** | | |
| Orig. feat. | 95.0% | - |
| Add. feat. | 95.8% | - |
| **LSTM** | | |
| Unidirectional | 72.7% | 200 |
| Bidirectional | 98.1% | 200 |
| **CNN** | | |
| w/o res. conn. | 91.3% | 200 |
| w/ res. conn. | 94.7% | 200 |
| **Transformer** | 92.7% | 200 |

## 2 Part 2: Transfer and Representation Learning

In this second part of the project, we aim at creating models to perform transfer-learning from another ECG dataset, called the MIT-BIH Arrhythmia Database dataset, onto the dataset from part 1 (the PTB dataset). The MIT-BIH Arrhythmia Database is a dataset that was compiled in 1980, consisting of ECG recordings from multiple patients in the Boston's Beth Israel Hospital. The dataset has 5 labels and the label distribution of the labels in the training and test sets is visualized in Figure 10.

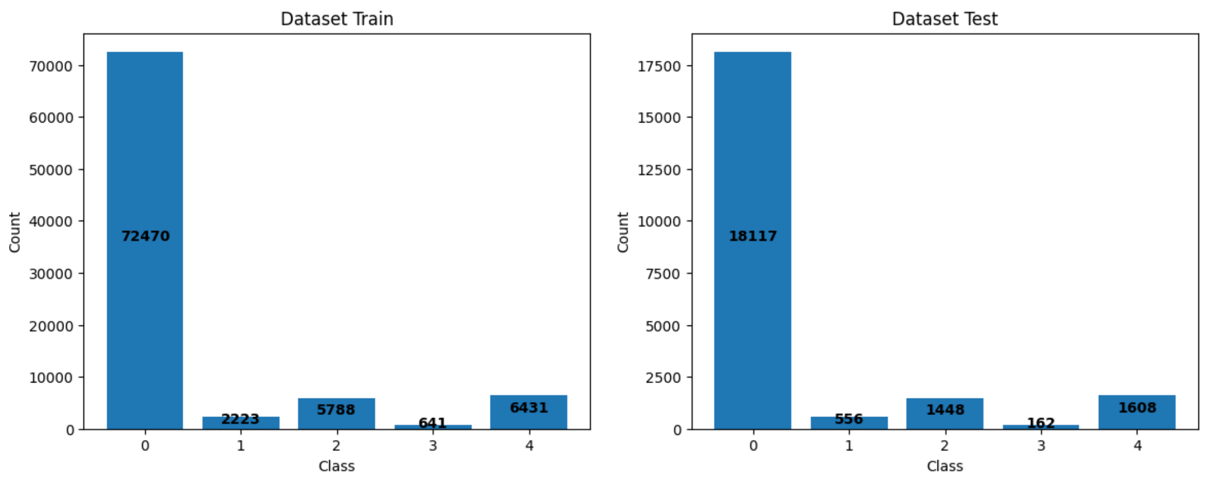


Figure 10: Label distribution of the MIT dataset.

### 2.1 Question 1: Supervised Model for Transfer

A model with the same architecture as the residual CNN in part 1 of the project was trained for 100 epochs on the MIT-BIH dataset, with 5 neurons in the output layer instead of 1, and using standard cross-entropy loss as the loss function. The progression of the loss and balanced accuracy are summarized in Figure 11. The test set balanced accuracy of the final model was 88.1%. Note that this is quite good accuracy, as the dataset is quite imbalanced.

The reason the residual CNN model was chosen for this part of the task is the fact that even though the classification accuracy of the bidirectional LSTM from part 1, the CNN took under 3 minutes to train for 200 epochs as opposed to 36 minutes for the bidirectional LSTM, while achieving comparable accuracy. The transformer model could also be a good candidate model, but in our case the residual CNN model performed slightly better than the transformer model in part 1. In this part of the project, balanced accuracy score will be reported on each of the models, for the same reason as it was used in part 1, because the dataset is quite imbalanced ($\sim 83\%$ of the training data coming from class 0).

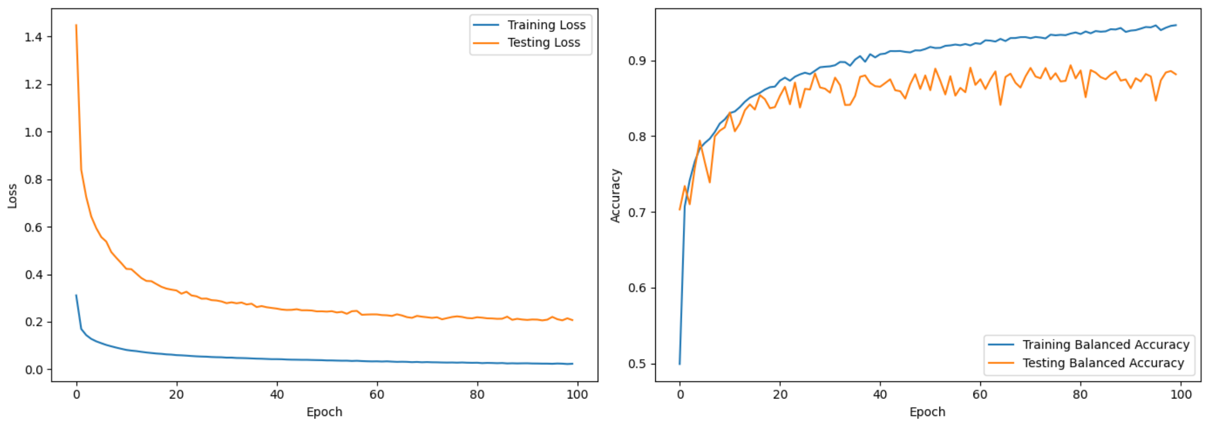


Figure 11: Training/Test Accuracy and Loss for 100 Epoch training of the CNN with residual connections

## 2.2 Question 2: Representation Learning Model

An autoencoder was trained on the dataset, having the same residual CNN as the one from Question 1 as an encoder (see architecture above), and decoding with a reversed version of the same model (using upsampling instead of max-pooling). The model architecture was the following:

```
AutoEncoder(
  # Encoder
  (conv1_enc): Conv1d(1, 16, kernel_size=(3,), stride=(1,), padding=(1,))
  (max_pool1): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu1): ReLU()
  (residual_block1_enc): ResidualBlock(
    (conv1): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (relu): ReLU()
  )
  (residual_block2_enc): ResidualBlock(
    (conv1): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (relu): ReLU()
  )
  (conv2_enc): Conv1d(16, 32, kernel_size=(3,), stride=(1,), padding=(1,))
  (max_pool2): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu2): ReLU()
  (fc_encoder_enc): LazyLinear(out_features=16, bias=True)

  # Decoder
  (fc_encoder_dec): LazyLinear(out_features=11776, bias=True)
  (upsample1): Upsample(scale_factor=2.0, mode='nearest')
  (conv2_dec): Conv1d(32, 16, kernel_size=(3,), stride=(1,), padding=(1,))
  (residual_block2_dec): ResidualBlock(
    (conv1): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (relu): ReLU()
  )
  (residual_block1_dec): ResidualBlock(
    (conv1): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (conv2): Conv1d(16, 16, kernel_size=(3,), stride=(1,), padding=(1,))
    (relu): ReLU()
  )
  (upsample2): Upsample(size=187, mode='nearest')
  (conv1_dec): Conv1d(16, 1, kernel_size=(3,), stride=(1,), padding=(1,))
)
```

The model objective is to reconstruct the time series that was input into it, and the mean square error (MSE) loss was used to monitor the pretraining step to compare the predicted outcome to the original input. A bottle neck of dimension 16 was used to embed each time series in the encoder. The loss progression of the model is visualized in Figure 12. On top of the 16-dimensional encodings of the time series, a random forest model was implemented, achieving 79.1% balanced accuracy on the test set.

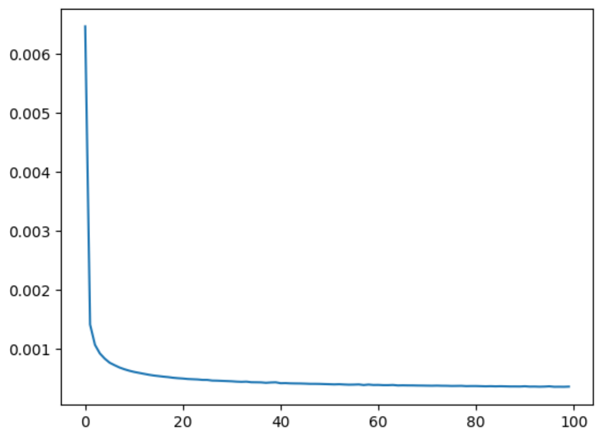


Figure 12: Training/Test Accuracy and Loss for 100 Epoch training of the Autoencoder

Table 5 summarizes the accuracies of the pretraining methods on the MIT-BIH dataset. Since the Residual CNN is specifically trained to classify the points into the 5 classes, it is natural that it has a higher accuracy than the autoencoder, which is trained to reconstruct the time series and knows nothing about the labels.

Table 5: Comparison of balanced accuracy on the test set of the two models

| **Model** | **Balanced Accuracy** | **Epochs** |
|---|---|---|
| Residual CNN | 88.1% | 100 |
| Autoencoder + Random Forest | 79.1% | 100 |

## 2.3 Question 3: Visualising Learned Representations

The output layer of the model from question 1 was used as an encoder that the training data from both the PTB and MIT datasets was passed through, and Figure 13 displays the UMAP representations of the datapoints, colored by label. We note that for the MIT training data, the data seems to split the data points from each of the 5 classes quite distinctively. The orange class (class 1) however, seems to completely overlap the blue class (negative samples). For the PTB dataset, the data points seem to overlap quite a lot and are not as identically distributed.

For the encoder, the UMAP visualizations can be seen on Figure 14, colored by label. The visualization of the embeddings are shaped quite weirdly. We cannot see a lot of structure in the UMAP visualization, but the red points and purple points seem to be quite distinctive from the others. In general, the data points seem to not be as identically distributed as the embeddings from the CNN, but not quite random.

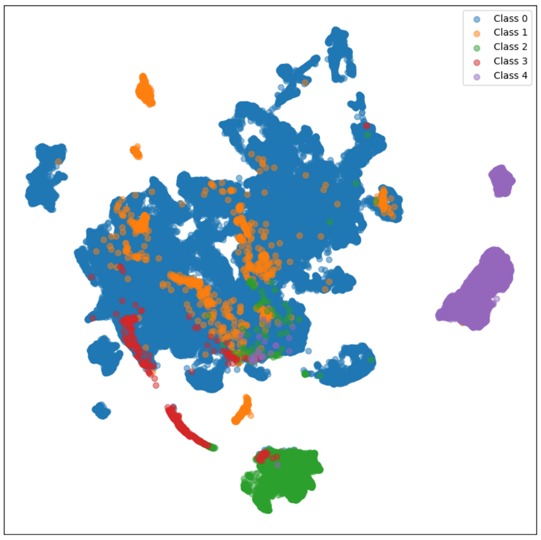
The figures on Figure 15 display a UMAP visualization of the embeddings produced by each model fit to the embeddings of both datasets simultaneously. It is very interesting to see that on this figure, we see that the representations learned from the CNN classifier on the PTB dataset seem out-of-distribution compared to the representations learned on the MIT dataset, while for the autoencoder, they seem to be very similarily distributed (which is a positive). This showcases the fact that the autoencoder model seems to learn a more general and robust representation of the data points.

### 2.3.1 Quantitative analysis of embeddings

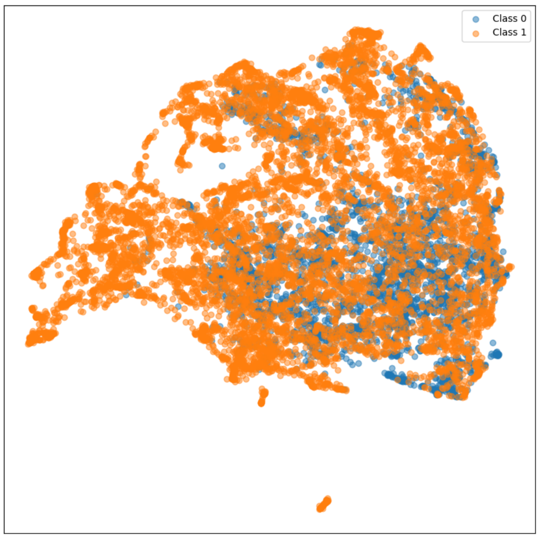
In this section, multiple different values of Kullback-Leibler divergence (KL divergence) are calculated. As a refresher, formally, KL divergence between two probability distributions $p$ and $q$ is defined as

$$KL(p,q) = \int p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right].$$

The KL divergence of $p$ and $q$ is 0 if and only if $p = q$. The Gibbs' inequality shows that $KL(p,q) \geq 0$ for all values of $p$ and $q$. It is important to note that KL divergence is not symmetric.

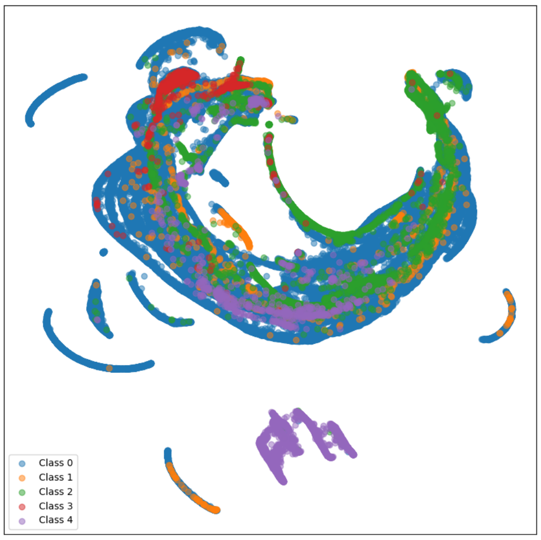


(a) UMAP representation of CNN encoded MIT data

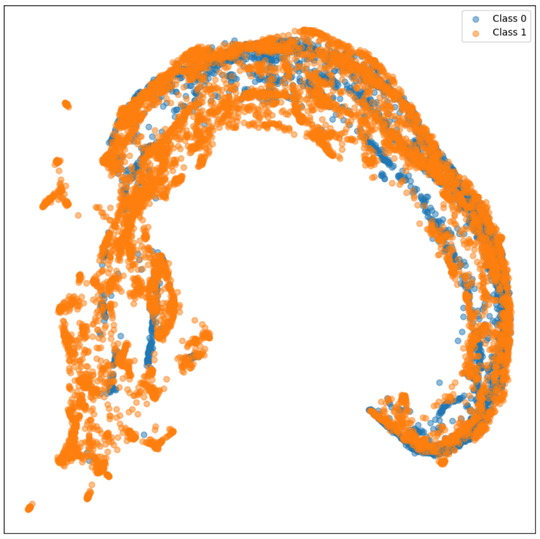


(b) UMAP representation of CNN encoded PTB data

Figure 13: UMAP representation of the embeddings produced by the autoencoder

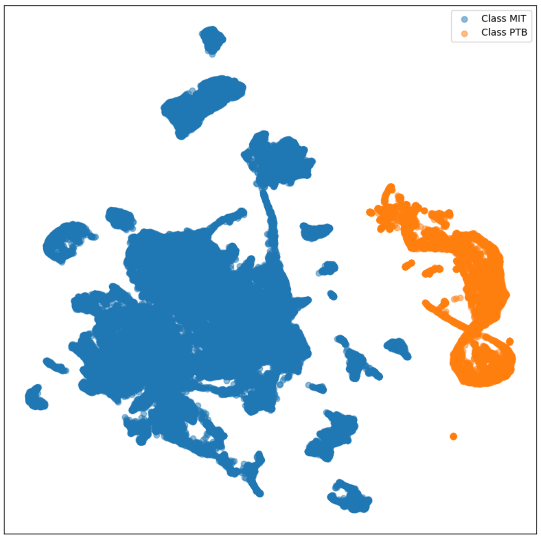


(a) UMAP representation of Autoencoder encoded MIT data

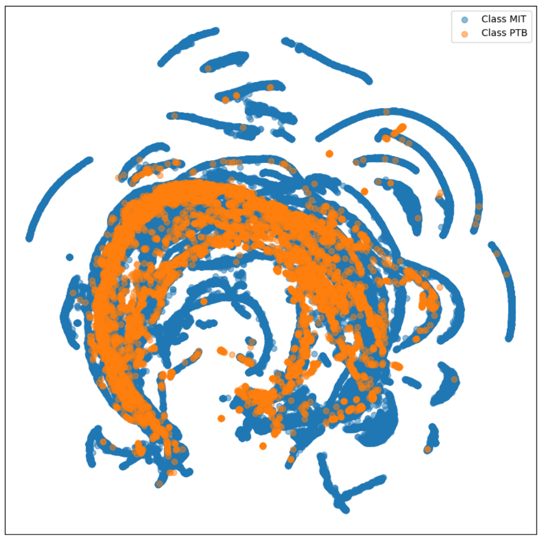


(b) UMAP representation of Autoencoder encoded PTB data

Figure 14: UMAP representation of the embeddings produced by the autoencoder



(a) Visualization of the embeddings produced by the CNN model fit on both datasets simultaneously



(b) Visualization of the embeddings produced by the autoencoder model fit on both datasets simultaneously

Figure 15: Visualization of the embeddings produced by each of the models fit on both datasets simultaneously.

Since we do not have access to the specific probability distributions of the embeddings, we will estimate the KL divergence of two embedding types $d_a$ and $d_b$ ($d_a$ being data points of type $a$ and $d_b$ being data points of type $b$) using Monte Carlo estimation. More formally, we will first map the data points embeddings to 4 dimensions using principal component analysis fit on the concatenation of both datasets. Then, a Gaussian Kernel Density approximation is fit of the data points of each type, let us call the approximations $K_a$ and $K_b$. The KL divergence $KL(d_a, d_b)$ is then estimated as

$$\frac{1}{|d_a|}\sum_{x\in d_a}\log\frac{K_a(x)}{K_b(x)}.$$

It is easy to show that this is an unbiased estimate, as it is an unbiased estimate of the expectation of $\log(p_a/p_b)$ of points sampled from $d_a$.

To quantitively evaluate the differences between how the CNN embeds each dataset to how the autoencoder embeds each dataset, each dataset was fed through the relevant model, and the data was mapped to four dimensions using principal component analysis on the concatenation of the two datasets of embeddings. Furthermore, the KL divergence between the embeddings of the MIT dataset and the embeddings of the PTB datasets, i.e. $KL(d_{MIT}, d_{PTB})$ where $d_{MIT}$ denotes the embeddings of the MIT dataset mapped to four dimensions and $d_{PTB}$ denotes the embeddings of the PTB dataset mapped to four dimensions. The steps described in the previous paragraph were followed, the results of this analysis is summarized in Table 6. We see that as we expected from the plots on Figure 15, the autoencoded sequences have multiple orders of magnitude lower KL divergence, meaning that they are from a more similar distribution than the CNN embeddings across datasets.

To quantitatively evaluate the differences between each of the label distributions, the embeddings were mapped to four dimensions using principal component analysis. Furthermore, to compare whether different labels are distributed identically, the KL divergence between each of the classes against every other class for each dataset was calculated, after fitting a kernel estimator to the distribution of the first class (similar approach as described in the previous paragraph). The results are summarized in Tables 7 through 10. The results are in general quite similar for the MIT dataset, the CNN embeddings seem to have about two times higher KL divergence between classes, meaning that the CNN model separates data points from different classes better. For the PTB dataset however, the autoencoder embeddings have 3-15 times higher KL divergence between the labels, meaning that the autoencoder separates points from different classes better. This seems to further support the claim that the autoencoder creates more robust and general embeddings for the time series.

Table 6: KL divergence between encoded MIT and PTB data distributions

| **Model** | $KL(d_{MIT}, d_{PTB})$ |
|---|---|
| **CNN** | 6454124 |
| **Autoencoder** | 0.849 |

Table 7: KL divergence between classes for the CNN encodings of the MIT dataset. Here, row $i$, column $j$ includes the value of $KL(d_i, d_j)$ where $d_i$ denotes the embeddings of class $i$ mapped to 4 dimensions and $d_j$ denotes the embeddings of class $j$ mapped to 4 dimensions

| | Class 0 | Class 1 | Class 2 | Class 3 | Class 4 |
|---|---|---|---|---|---|
| Class 0 | - | 15.30 | 39.62 | 11.17 | 66.20 |
| Class 1 | 9.55 | - | 17.09 | 10.94 | 50.71 |
| Class 2 | 29.08 | 61.78 | - | 12.03 | 41.71 |
| Class 3 | 86.43 | 148.78 | 85.60 | - | 413.25 |
| Class 4 | 69.85 | 103.29 | 60.26 | 34.06 | - |

Table 8: KL divergence between classes for the autoencoder encodings of the MIT dataset. Here, row $i$, column $j$ includes the value of $KL(d_i, d_j)$ where $d_i$ denotes the embeddings of class $i$ mapped to 4 dimensions and $d_j$ denotes the embeddings of class $j$ mapped to 4 dimensions

| | Class 0 | Class 1 | Class 2 | Class 3 | Class 4 |
|---|---|---|---|---|---|
| Class 0 | - | 8.49 | 14.52 | 11.66 | 27.83 |
| Class 1 | 10.98 | - | 22.41 | 18.78 | 46.23 |
| Class 2 | 10.97 | 12.78 | - | 14.19 | 21.77 |
| Class 3 | 45.03 | 61.38 | 66.61 | - | 103.84 |
| Class 4 | 44.35 | 55.28 | 66.61 | 40.28 | - |

Table 9: KL divergence between classes for the CNN encodings of the PTB dataset. Here, row $i$, column $j$ includes the value of $KL(d_i, d_j)$ where $d_i$ denotes the embeddings of class $i$ mapped to 4 dimensions and $d_j$ denotes the embeddings of class $j$ mapped to 4 dimensions

| | Class 0 | Class 1 |
|---|---|---|
| Class 0 | - | 1.10 |
| Class 1 | 2.62 | - |

Table 10: KL divergence between classes for the autoencoder encodings of the PTB dataset. Here, row $i$, column $j$ includes the value of $KL(d_i, d_j)$ where $d_i$ denotes the embeddings of class $i$ mapped to 4 dimensions and $d_j$ denotes the embeddings of class $j$ mapped to 4 dimensions

| | Class 0 | Class 1 |
|---|---|---|
| Class 0 | - | 15.80 |
| Class 1 | 6.63 | - |

### 2.4 Question 4: Finetuning Strategies

All of the finetuning strategies were implemented transferring the models from questions 1 and 2 onto the PTB dataset, using random forest as the classic ML method. The two-step finetuning process included training the output layers of the network for 50 epochs while freezing the encoder and then training the whole model for 50 epochs. Figures 16 through 21 show the loss and accuracy progression of the models through the 100 epochs of training and the performance on the test set is summarized in Table 11.

We see that the table clearly shows that the two-step finetuning process seems to be the best way to train the model, yielding in a very high accuracy similar to the bidirectional LSTM from part 1 and over

3% better than the residual CNN from part 1. It can be expected that the model trained on the MIT dataset and the PTB dataset performs slightly better than a model only trained in the PTB dataset as it is trained on more data. It is natural that the 2-step finetuning process performs the best as it gets the best of both worlds. Even though finetuning the entire model seems like a good idea, finetuning the output layer first results in an easier optimization problem for the whole model, as it gives a better starting point. When finetuning the entire model from scratch, it is initialized further from the global minimum and has a harder time finding the optimal set of parameters.

We see that the autoencoder embeddings were seemingly better than the residual CNN up until the final 2-step finetuning, where both embedding models performed similarily (and very close too being 100% accurate). This is most likely because the encoders themselves have almost exactly the same architecture and thus have the same predictive power. It seems as though the embeddings from the autoencoder (representation learning) were better for the PTB dataset than the embeddings from the residual CNN (transfer learning) even though they were slightly worse for the MIT dataset (as was seen in the analysis in Question 3 of this part), as they resulted in better accuracy in the parts before the 2-step finetuning process. This shows that the representation learning approach seems to be a better and more robust way of learning general representation of time series, while the transfer learning approach, which was specifically trained on the classification of the MIT-BIH dataset, does not generalize as well.

Table 11: Comparison of Accuracy and Loss after finetuning of Transformer and Autoencoder following 3 different finetuning strategies

| **Model** | **Balanced Accuracy** | **Epochs** |
|---|---|---|
| **Residual CNN (Transfer learning)** | | |
| Random Forest on encoding | 83.0% | - |
| Finetuning of output layer | 76.4% | 100 |
| Finetuning of entire model | 87.5% | 100 |
| 2-stage finetuning | 98.4% | 50/50 |
| **Autoencoder (Representation learning)** | | |
| Random Forest on encoding | 91.8% | - |
| Finetuning of output layer | 80.0% | 100 |
| Finetuning of entire model | 91.4% | 100 |
| 2-stage finetuning | 97.8% | 50/50 |

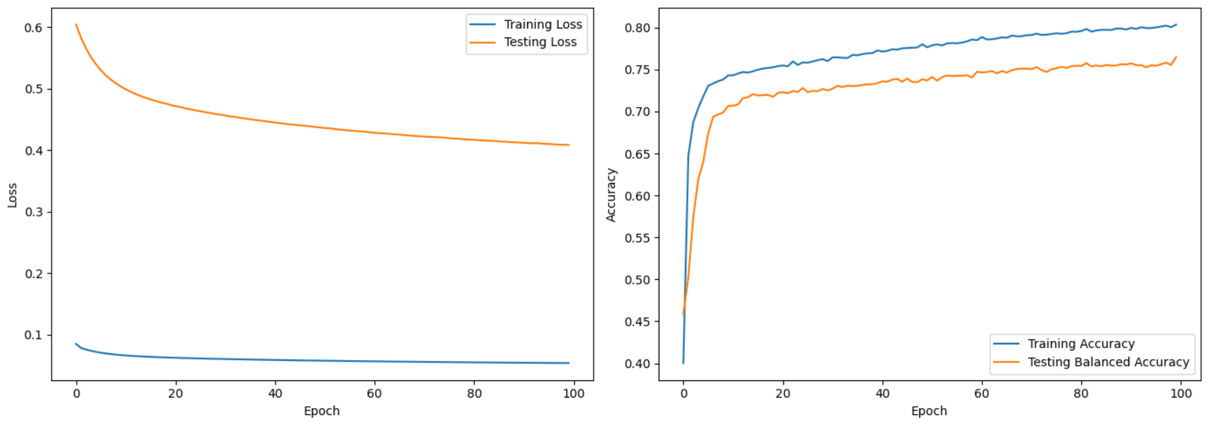


Figure 16: Training/Test Accuracy and Loss for 100 Epoch finetuning of the output layer of the CNN with residual connections

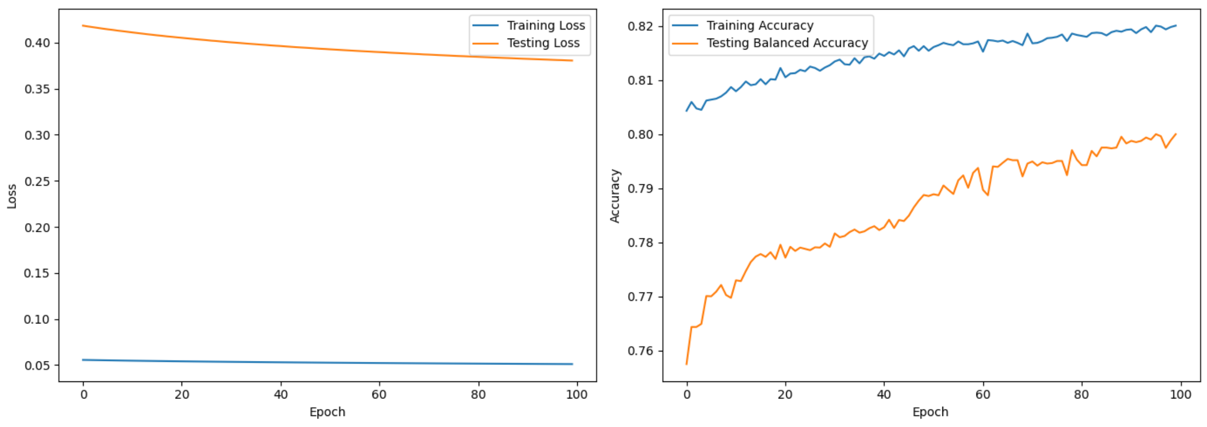


Figure 17: Training/Test Accuracy and Loss for 100 Epoch finetuning of the output layer of the Autoencoder

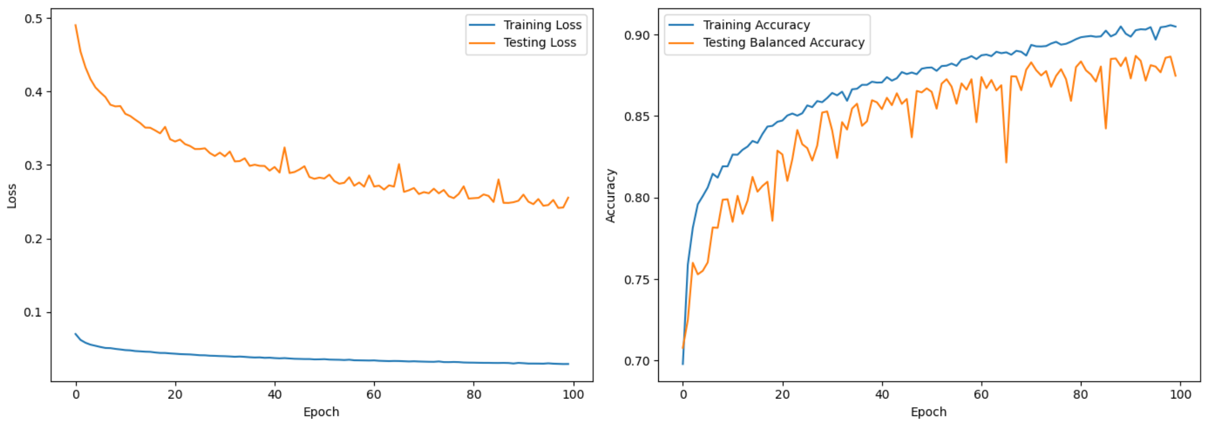


Figure 18: Training/Test Accuracy and Loss for 100 Epoch finetuning of the entire CNN with residual connections

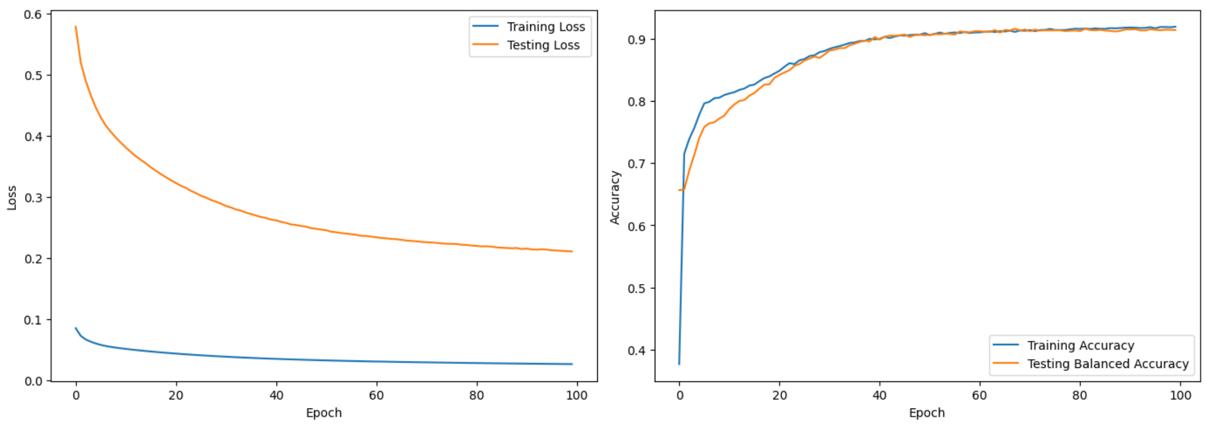


Figure 19: Training/Test Accuracy and Loss for 100 Epoch finetuning of the entire Autoencoder

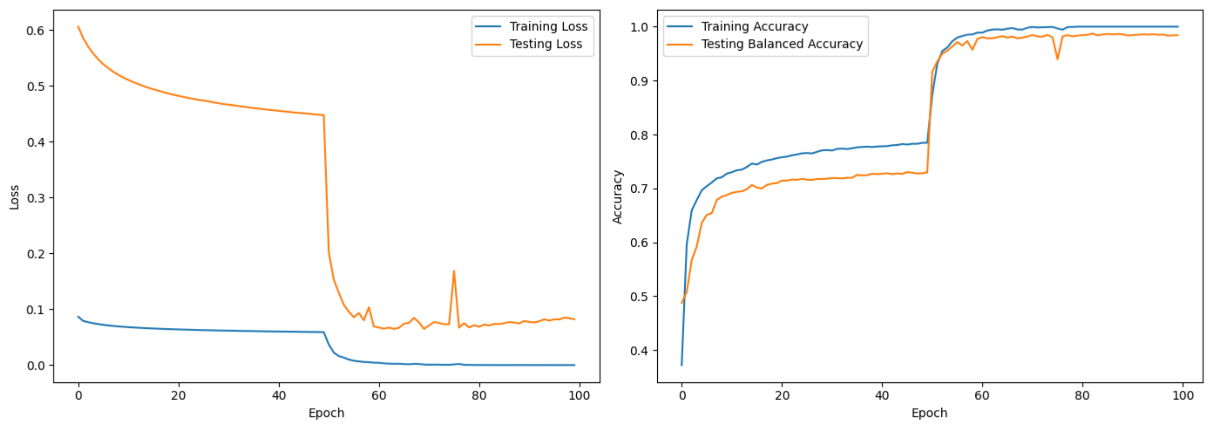


Figure 20: Training/Test Accuracy and Loss for 100 Epoch two-step finetuning of the CNN with residual connections - Epoch 1-50 finetuning output layers only and Epcoh 51-100 finetuing the entire model

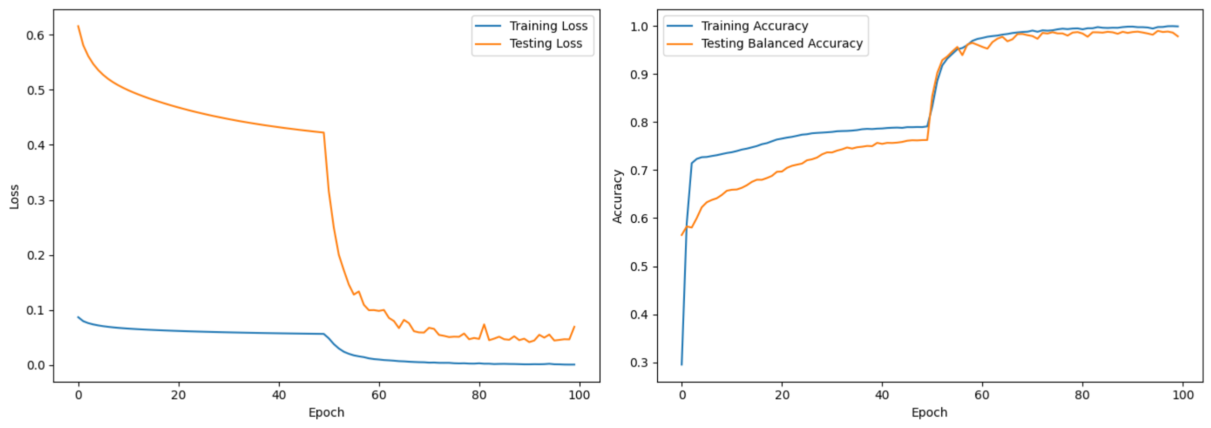


Figure 21: Training/Test Accuracy and Loss for 100 Epoch two-step finetuning of the Autoencoder - Epoch 1-50 finetuning output layers only and Epoch 51-100 finetuning the entire model

## 3 Part 3: General Questions

### 3.1 Question 1

There are many machine learning settings where classic methods are still competitive with deep learning architectures. Have you observed this in this project? Why is this (not) the case? (2 pts)

**Answer:** Looking at the models from this project, we see that the Random Forest algorithm produces comparable results to that of the deep learning models in the project, beating every classifier in part 1 except for a bidirectional LSTM. The reason for that in this case is probably because the task was not so complex, each observation only having 187 variables (many of them potentially having value 0). For more complex time series, like speaker classification, we suspect that random forest models would not perform as well, even though the random forest algorithm is in general a good way to get good results with minimum overhead. In general, for simple tasks, classic ML methods can be more robust than deep learning methods as they are less prone to overfitting.

### 3.2 Question 2

When modeling time series using deep learning architectures, when might you want a causal/unidirectional architecture? Give an example of a task. (1 pt)

**Answer:** In a next-step prediction model, where we want to predict one step into the future at a time, using a bidirectional architecture is not as useful as it is for classification of a whole time series, since the hidden representation of the most recent time step is the most important one for next-step prediction.

Examples of tasks where a unidirectional model would be more applicable are weather forecasting and stock price prediction, since these tasks are next-step predictions that only should depend on the previous time steps.

### 3.3 Question 3

Can you think of an attention-related bottleneck regarding very (very) long time series? Conceptually, which deep methods from above are more suitable for such long time series? (2 pt)

**Answer:** An attention-related bottleneck for very long time series is the fact that if the sequence length is $N$, the attention matrices will have size $N \times N$, and multiplying matrices of this size can be quite computationally expensive if $N$ is very large (the size of the attention matrices grow quadratic as $N$ grows). The same problem also goes for LSTMs, as the gradients could become very small/very large if the input sequence is very long. A more computationally efficient method more suitable for such a task could be a CNN (possibly using dilated convolutions to increase context window), as the convolutional layers have the same number of parameters (parameter sharing) independent of the input size.

### 3.4 Question 4

What are some challenges in using self-supervised representation learning? What difficulties have you observed in your approach? Can you think of additional ones? (2 pt)

**Answer:** Self-supervised learning is generally a hard problem, as the goal of it is to understand the distribution of the dataset without knowing the underlying labels. It is not as easy to measure the quality of a self-supervised model during training compared to classifiers which are very easy to monitor and easy to know if the quality of the model is good or bad. The embeddings that the pretrained autoencoder produced seemed to be distributed in a strange way and it was hard to visualize them. However, the embeddings of the autoencoder model turned out to be quite good, as can bee seen from the results from Question 3 and 4 in Part 2 of the report, as the embeddings generalized quite nicely to the PTB dataset.

There exist better ways of performing self-supervised learning, like the Contrastive Predictive Coding approach (using the InfoNCE loss). However, these methods are generally very computationally heavy, have complex architecture and implementation, and have non-trivial and random loss functions. The InfoNCE loss involves sampling negative data points for every data point during training, which makes the training of the model slow. However, for this relatively simple task of classifying 187-dimensional ECG data, this approach turned out to be an overkill, as we ran into a lot of problems when trying to implement the approach.

## 4 Project Code

All the code including images produced for the project can be found in the GitHub repository for the project, but the notebooks, README and requirements.txt were handed in as .zip files to Moodle