Summative Assessment Section C

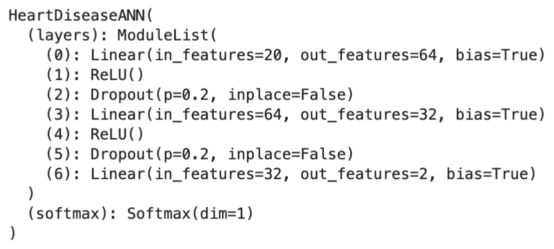
Keli Niu (ad21083@bristol.ac.uk)

1 ANN for Classification

The supervised learning technique chosen for this task is Artificial Neural Networks (ANN), which are widely used in classification and regression tasks. Supervised learning is a method of training a model by using labelled data [1]. Since the dataset used in this task is a binary classification problem on heart failure prediction involving non-linear relationships of multiple features, ANN is a suitable technique to solve this problem. ANN has an advantage over other supervised learning models in capturing non-linear and complex relationships between features. This enables it to handle high dimensional datasets and complex decision boundaries effectively [2].

1.1 Underlying Principles and Assumptions

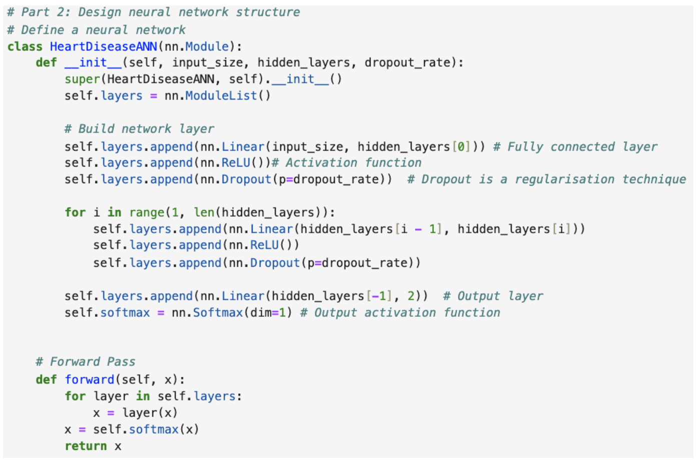
ANN is a computational system that mimics the way the human brain works, which can recognise relationships in data. A typical ANN consists of three main layers: the input layer, the hidden layers, and the output layer [3]. Each layer is composed of multiple neurons, which perform linear transformations on the input followed by a nonlinear activation function. This enables the model to capture both linear and nonlinear relationships in the data.



```
HeartDiseaseANN(
  (layers): ModuleList(
    (0): Linear(in_features=20, out_features=64, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=64, out_features=32, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=32, out_features=2, bias=True)
  )
  (softmax): Softmax(dim=1)
)
```

Figure 1: ANN structure.

Figure 1 illustrates the structure of a simple ANN constructed for this task. It comprises two hidden layers with 64 and 32 neurons, respectively. Figure 2 shows the code that builds the basic architecture of the ANN.



```python
# Part 2: Design neural network structure
# Define a neural network
class HeartDiseaseANN(nn.Module):
    def __init__(self, input_size, hidden_layers, dropout_rate):
        super(HeartDiseaseANN, self).__init__()
        self.layers = nn.ModuleList()

        # Build network layer
        self.layers.append(nn.Linear(input_size, hidden_layers[0])) # Fully connected layer
        self.layers.append(nn.ReLU())# Activation function
        self.layers.append(nn.Dropout(p=dropout_rate))  # Dropout is a regularisation technique

        for i in range(1, len(hidden_layers)):
            self.layers.append(nn.Linear(hidden_layers[i - 1], hidden_layers[i]))
            self.layers.append(nn.ReLU())
            self.layers.append(nn.Dropout(p=dropout_rate))

        self.layers.append(nn.Linear(hidden_layers[-1], 2))  # Output layer
        self.softmax = nn.Softmax(dim=1) # Output activation function


    # Forward Pass
    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        x = self.softmax(x)
        return x
```

Figure 2: Code of building ANN structure.

The key components chosen for the network are explained in detail below:

1. Activation Function (ReLU): The activation function is applied after the fully connected layers in each hidden layer to introduce nonlinearity, allowing the model to learn the complex relationships. In this task, the ReLU (Rectified Linear Unit) function is used, defined as [4]:

$$f(x) = \max(0, x),$$

where **x** is the input to the neuron.

ReLU enhances the sparsity of the network by setting negative inputs to zero, thereby speeding up training and mitigating the vanishing gradient problem. This makes the model more efficient in learning complex data structures [4].

2. Dropout Layer: Dropout is a simple regularisation technique used to solve overfitting. During training, it randomly drops a proportion of input units, thus reducing over-reliance on specific neurons [5]. Specifically this can be understood visually using the schematic diagram shown in Figure 3.

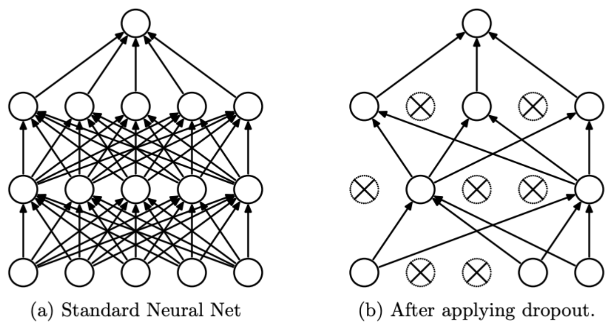


Figure 3: Dropout Neural Net Model [5].

Since the dataset used in this task has only 20 features, the model may easily overfit to the limited features, negatively affecting its generalisation ability. Setting the dropout rate to 0.2 effectively balances preventing overfitting with retaining the network's learning capacity.

3. Output Activation Function (Softmax): In the output layer, the Softmax function is used to transform logits into probabilities. Compared to the Sigmoid activation function, Softmax calculates the probability distribution over all possible classes, making it suitable for multi-class tasks. In this task, Softmax is chosen to output the probability distribution for the two possible classes (presence or absence of heart disease). This ensures the ANN structure is extensible to other multi-class tasks, providing scalability.

The Softmax function is defined as follows [6]:

$$P(y = j \mid \mathbf{x}) = \frac{\exp(\mathbf{x}^\top \mathbf{w}_j)}{\sum_{k=1}^{K} \exp(\mathbf{x}^\top \mathbf{w}_k)},$$

where $\mathbf{x}$ is the input vector representing the sample features; $\mathbf{w}_j$ is the weight vector for class j; K is the total number of classes. Figure 4 below shows a plot of the function of softmax.

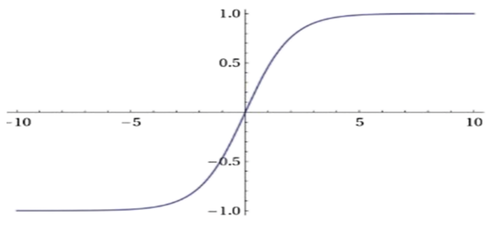


Figure 4: Softmax function plot [6].

4. Loss function: The CrossEntropyLoss function quantifies the difference between the predicted class probabilities and the true labels. It is a widely used loss function in classification tasks [7], grounded in the principle of Maximum Likelihood Estimation (MLE). The function operates by applying a negative logarithm to the predicted probability of the true class, effectively penalising the model more heavily when it assigns low confidence to the correct category. By averaging these penalties across all samples, the loss produces a single value that evaluates the model's performance and directs its optimisation. The specific formula is as follows:

$$L = -\frac{1}{N}\left[\sum_{j=1}^{N} (t_j \log(p_j) + (1 - t_j)\log(1 - p_j))\right],$$

where, $\mathbf{N}$ is the number of data points, $\mathbf{t_i}$ is the truth value taking a value of 0 or 1, and $\mathbf{p_i}$ is the Softmax probability for the $\mathbf{i}^{\text{th}}$ data point.

5. Optmiser (Adam): The optimiser adjusts the model parameters (such as weights and biases) to minimise the loss function, thereby improving the model's prediction performance. Adam is a computationally efficient and easy-to-implement stochastic gradient optimisation algorithm. It dynamically adjusts the learning rate for each parameter by calculating the first moment (mean) and the second moment (uncentered variance) of the gradients. Combined with bias correction and adaptive learning rates, Adam updates parameters in a smoother and more stable manner [8]. The update equations are as follows:

$$g_t = \nabla_{\theta_t} L(\theta_t) \tag{1}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{5}$$

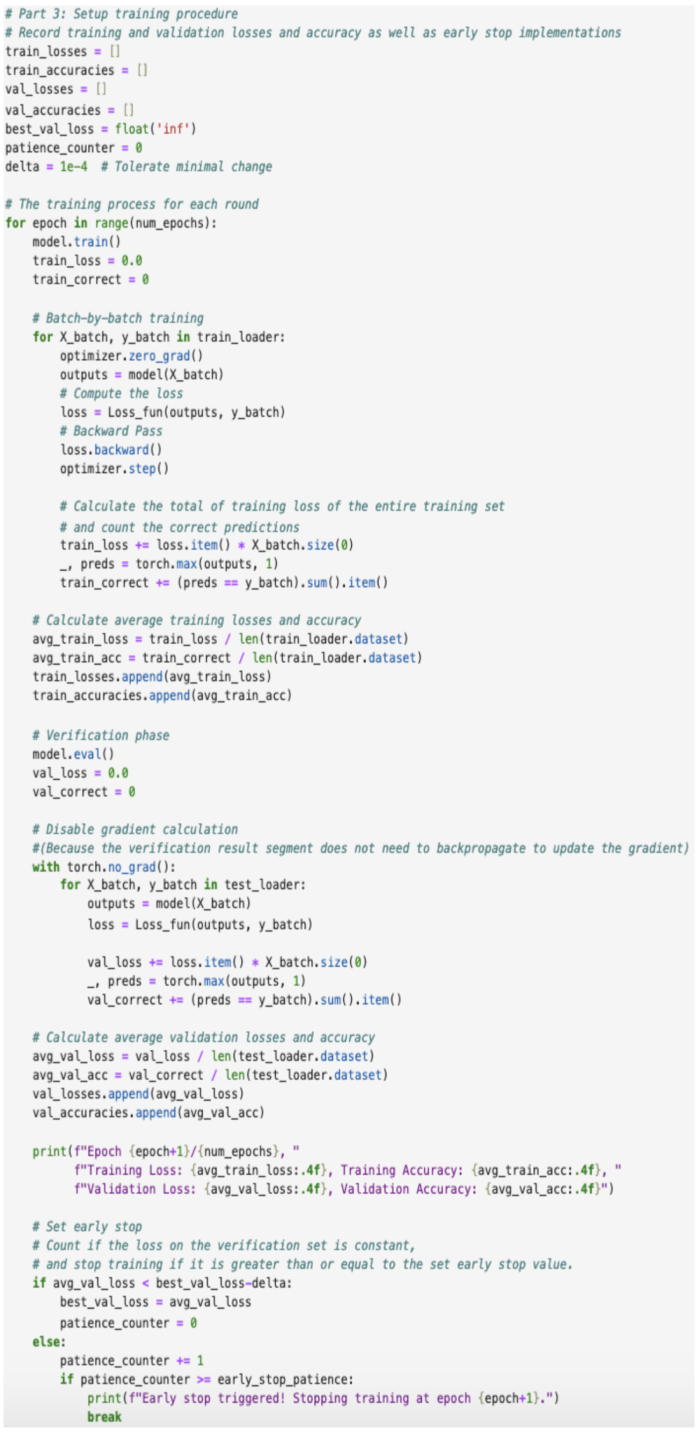
Specifically: (1) $\mathbf{g_t}$ represents the gradient of the loss function $\mathbf{L}(\theta_\mathbf{t})$ with respect to the parameters $\theta_\mathbf{t}$, indicating how sensitive the loss is to parameter changes. (2) $\mathbf{m_t}$ is the first moment (moving average of gradients), capturing the overall trend of the gradient. $\beta_\mathbf{1}$ is the momentum decay rate, and $\mathbf{m_t}$ denotes the current momentum. (3) $\mathbf{v_t}$ is the second moment (moving average of squared gradients), measuring the scale of gradient variations. $\beta_\mathbf{2}$ controls how past squared gradients influence the current update. (4) $\hat{\mathbf{m}}_\mathbf{t}$ and $\hat{\mathbf{v}}_\mathbf{t}$ are bias-corrected versions of the moments to counteract the initialization bias. (5) The final step uses the corrected moments to dynamically adjust the learning rate for each parameter and update them. Here, α is the learning rate, ϵ is a small constant to prevent division by zero, $\theta_\mathbf{t}$ and $\theta_\mathbf{t+1}$ are the current and the updated parameter, respectively.

1.2 Training process and prediction

As shown in the code in Figure 5, the process of training an ANN can be divided into a training phase and a validation phase as follows: First, the training phase processes the data by batch. For each batch of data, the model calculates the output by Forward Pass. It generates predictions by linearly transforming the input data according to the current weights and biases, and then introduces a nonlinear mapping through the activation function. After of that, the error between the predicted and true values is calculated using a loss function, which serves to quantify the performance of the model. Afterwards, the gradient of the loss function for each weight and bias is calculated by Backward Pass to represent the degree of influence of each parameter on the error. At the same time, the parameters are updated using the optimiser so that the value of the loss function gradually decreases, thus improving the model's ability to fit the data. After all batches are completed, the average loss and accuracy of the training set are calculated to measure the performance of the current model on the training data.

Then for the validation phase, where predictions are made in the validation set. It only requires calculating the output and loss by forward propagation to evaluate the performance of the model on unseen data. This process is used to test the generalisation ability of the model.

In addition, an early stopping mechanism is incorporated to monitor the trend of the validation set loss. If the loss no longer decreases within multiple consecutive rounds, the training is terminated early to prevent model overfitting.



```python
# Part 3: Setup training procedure
# Record training and validation losses and accuracy as well as early stop implementations
train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []
best_val_loss = float('inf')
patience_counter = 0
delta = 1e-4  # Tolerate minimal change

# The training process for each round
for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0
    train_correct = 0

    # Batch-by-batch training
    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        outputs = model(X_batch)
        # Compute the loss
        loss = Loss_fun(outputs, y_batch)
        # Backward Pass
        loss.backward()
        optimizer.step()

        # Calculate the total of training loss of the entire training set
        # and count the correct predictions
        train_loss += loss.item() * X_batch.size(0)
        _, preds = torch.max(outputs, 1)
        train_correct += (preds == y_batch).sum().item()

    # Calculate average training losses and accuracy
    avg_train_loss = train_loss / len(train_loader.dataset)
    avg_train_acc = train_correct / len(train_loader.dataset)
    train_losses.append(avg_train_loss)
    train_accuracies.append(avg_train_acc)

    # Verification phase
    model.eval()
    val_loss = 0.0
    val_correct = 0

    # Disable gradient calculation
    #(Because the verification result segment does not need to backpropagate to update the gradient)
    with torch.no_grad():
        for X_batch, y_batch in test_loader:
            outputs = model(X_batch)
            loss = Loss_fun(outputs, y_batch)

            val_loss += loss.item() * X_batch.size(0)
            _, preds = torch.max(outputs, 1)
            val_correct += (preds == y_batch).sum().item()

    # Calculate average validation losses and accuracy
    avg_val_loss = val_loss / len(test_loader.dataset)
    avg_val_acc = val_correct / len(test_loader.dataset)
    val_losses.append(avg_val_loss)
    val_accuracies.append(avg_val_acc)

    print(f"Epoch {epoch+1}/{num_epochs}, "
          f"Training Loss: {avg_train_loss:.4f}, Training Accuracy: {avg_train_acc:.4f}, "
          f"Validation Loss: {avg_val_loss:.4f}, Validation Accuracy: {avg_val_acc:.4f}")

    # Set early stop
    # Count if the loss on the verification set is constant,
    # and stop training if it is greater than or equal to the set early stop value.
    if avg_val_loss < best_val_loss-delta:
        best_val_loss = avg_val_loss
        patience_counter = 0
    else:
        patience_counter += 1
        if patience_counter >= early_stop_patience:
            print(f"Early stop triggered! Stopping training at epoch {epoch+1}.")
            break
```

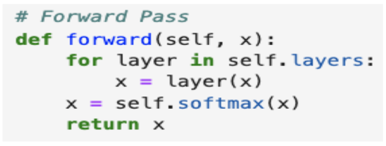
Figure 5: The process of training an ANN.

1.3 Forward Pass and Backward Pass

Forward Pass is the computation phase of a neural network, where input data is processed through the layers to generate output predictions. Each layer applies a linear transformation to the inputs followed by a non-linear activation function $g(\cdot)$, as expressed below [9]:

$$z^{(l)} = W^{(l)}A^{(l-1)} + b^{(l)}, \quad A^{(l)} = g(z^{(l)})$$

where,$\mathbf{z}^{(l)}$: Linear output at layer **l**; $\mathbf{W}^{(l)}$ is weight matrix of layer **l**; $\mathbf{A}^{(l-1)}$ is the activation from the previous layer or input data; $\mathbf{b}^{(l)}$ is the bias of layer. The code is shown in Figure 6.



```python
# Forward Pass
def forward(self, x):
    for layer in self.layers:
        x = layer(x)
    x = self.softmax(x)
    return x
```

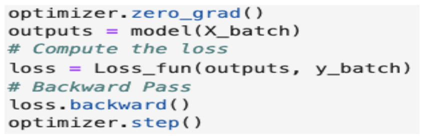
Figure 6: The forward pass.

Backward Pass calculates the gradients of the loss function with respect to the model parameters to optimise the network. Update the weight and bias by using the chain rule [9, 10]:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}},$$

$$\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial b^{(l)}},$$

where, $\mathbf{z}^{(l)}$ is the output of the linear combination of the $\mathbf{l}^{\mathbf{th}}$ layer; $\mathbf{W}^{(l)}$ is the weight matrix; $\mathbf{b}^{(l)}$ is the bias vector. The code is shown in Figure 7.



```python
optimizer.zero_grad()
outputs = model(X_batch)
# Compute the loss
loss = Loss_fun(outputs, y_batch)
# Backward Pass
loss.backward()
optimizer.step()
```

Figure 7: The forward pass.

1.4 Test Data Prediction Process

According to the previous sections, the model makes predictions about the test data through forward propagation. Specifically, the input features are passed through the network layer by layer, applying linear transformations and nonlinear activation functions at each layer. The final probability distribution for each category is generated by the Softmax function in the output layer. The model will select the category with the highest probability as the prediction, thus making the final classification decision for the test data.

1.5 Training Results Visualisation

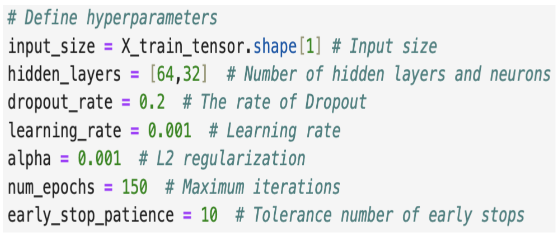

```
# Define hyperparameters
input_size = X_train_tensor.shape[1] # Input size
hidden_layers = [64,32]  # Number of hidden layers and neurons
dropout_rate = 0.2  # The rate of Dropout
learning_rate = 0.001  # Learning rate
alpha = 0.001  # L2 regularization
num_epochs = 150  # Maximum iterations
early_stop_patience = 10  # Tolerance number of early stops
```

Figure 8: Hyperparameters for training the model.

The ANN model was trained using the parameter settings shown in Figure 8. The results and the code to present the results are shown in Figure 9 and Figure 10, respectively. The left plot shows the training and validation loss, where both decrease during training. However, the validation loss decreases more slowly than the training loss, indicating a slight overfitting problem. The right plot shows the training and validation accuracy. While the training accuracy continues to increase, the validation accuracy stabilizes, also reflecting overfitting. Future improvements could involve tuning regularisation techniques to improve the generalisation performance of the model.

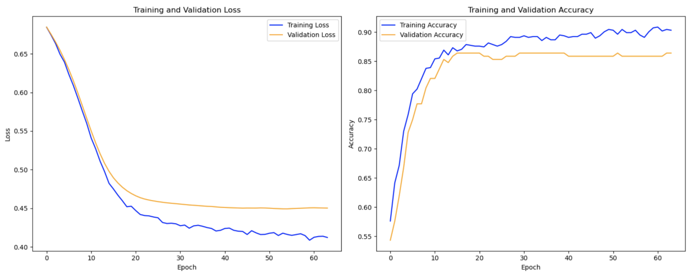


Figure 9: The training result: on the left is the loss curve and on the right is the accuracy curve. Where blue is the training set and yellow is the validation set.

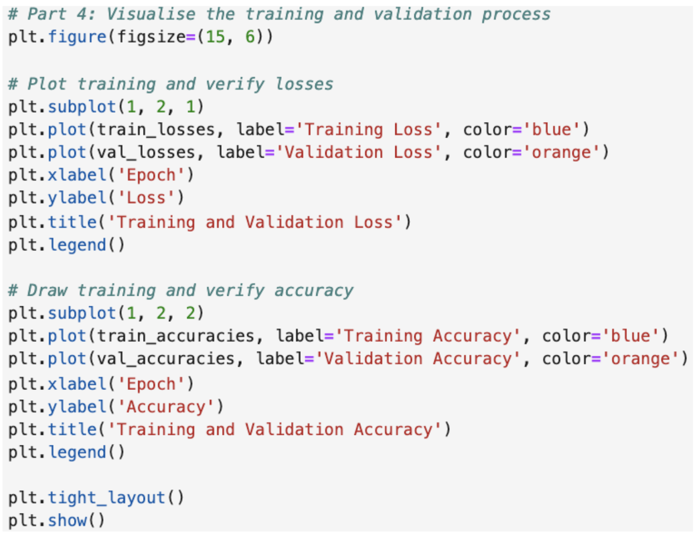

```
# Part 4: Visualise the training and validation process
plt.figure(figsize=(15, 6))

# Plot training and verify losses
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss', color='blue')
plt.plot(val_losses, label='Validation Loss', color='orange')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()

# Draw training and verify accuracy
plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training Accuracy', color='blue')
plt.plot(val_accuracies, label='Validation Accuracy', color='orange')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

Figure 10: Visualise the training and validation process.

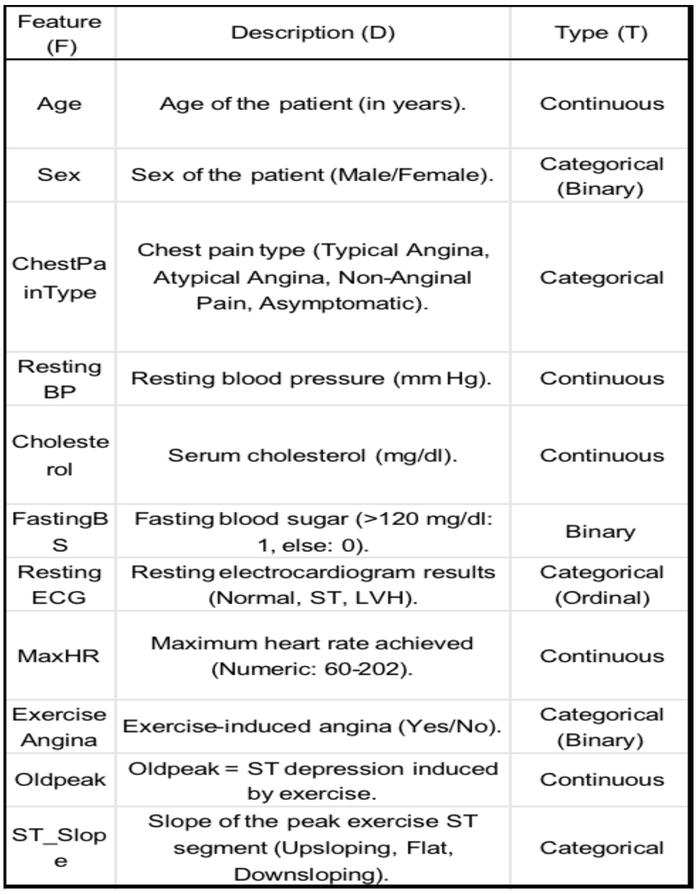
1.6 Application Areas of ANN

Artificial Neural Networks (ANNs) excel at capturing complex non-linear relationships between features, making them particularly suitable for tasks where traditional linear models struggle. They are widely used in applications such as image recognition, natural language processing, and medical diagnosis [11].

The present task involves Heart Failure Prediction, a binary classification problem that exhibits non-linear relationships among multiple medical features, including both continuous (e.g., age, cholesterol level) and categorical variables (e.g., chest pain type). ANNs are capable of identifying and leveraging these complex patterns, which makes them highly effective for this type of problem.

However, there are some limitations to consider. ANNs often require large amounts of labeled training data and substantial computational resources to perform well. Furthermore, their decision-making process lacks transparency and is often referred to as a ”black box” model. Despite these challenges, their unparalleled ability to approximate complex functions positions ANNs as a powerful tool for classification and prediction tasks, particularly in high-stakes areas like healthcare, where accuracy is critical.

2 Description of the Data Set

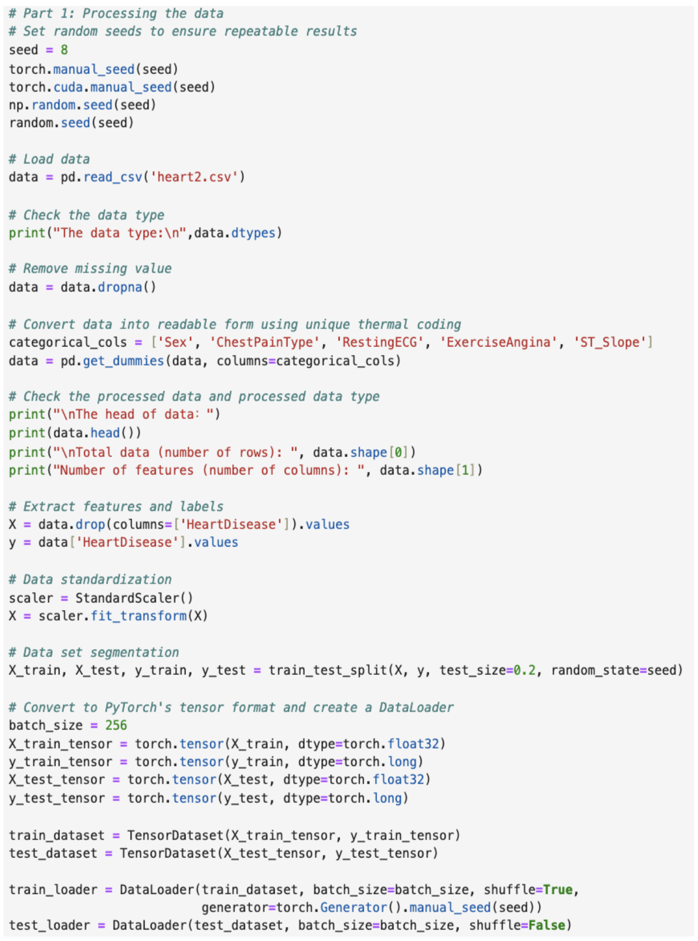


Feature (F)	Description (D)	Type (T)
Age	Age of the patient (in years).	Continuous
Sex	Sex of the patient (Male/Female).	Categorical (Binary)
ChestPainType	Chest pain type (Typical Angina, Atypical Angina, Non-Anginal Pain, Asymptomatic).	Categorical
Resting BP	Resting blood pressure (mm Hg).	Continuous
Cholesterol	Serum cholesterol (mg/dl).	Continuous
FastingBS	Fasting blood sugar (>120 mg/dl: 1, else: 0).	Binary
Resting ECG	Resting electrocardiogram results (Normal, ST, LVH).	Categorical (Ordinal)
MaxHR	Maximum heart rate achieved (Numeric: 60-202).	Continuous
Exercise Angina	Exercise-induced angina (Yes/No).	Categorical (Binary)
Oldpeak	Oldpeak = ST depression induced by exercise.	Continuous
ST_Slope	Slope of the peak exercise ST segment (Upsloping, Flat, Downsloping).	Categorical

Figure 11: Description and Types of Features.

The dataset used in this task is the Heart Failure Prediction Dataset [12]. It predicts the likelihood of heart disease based on 11 features that are potential indicators. The dataset comprises five subsets: 303 Cleveland observations, 294 Hungarian observations, 123 Switzerland observations, 200 Long Beach VA observations, and 270 Stalog (Heart) Dataset observations. After removing 272 duplicate samples, the final dataset includes 918 samples and 12 features, including the target variable ”HeartDisease.” The meanings and types of the other 11 features are shown in Figure 11.

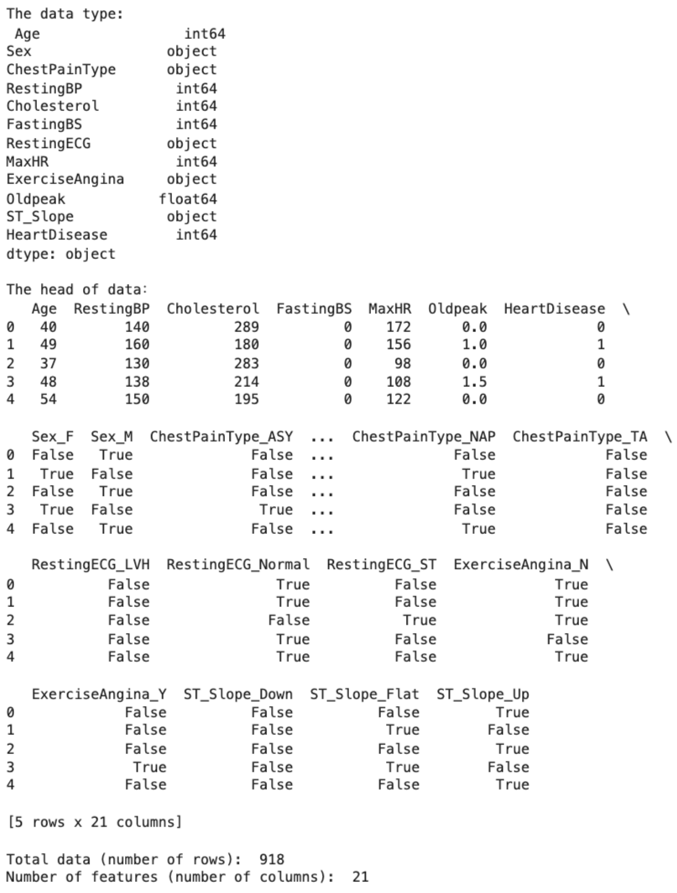
2.1 Data Processing



```python
# Part 1: Processing the data
# Set random seeds to ensure repeatable results
seed = 8
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
random.seed(seed)

# Load data
data = pd.read_csv('heart2.csv')

# Check the data type
print("The data type:\n",data.dtypes)

# Remove missing value
data = data.dropna()

# Convert data into readable form using unique thermal coding
categorical_cols = ['Sex', 'ChestPainType', 'RestingECG', 'ExerciseAngina', 'ST_Slope']
data = pd.get_dummies(data, columns=categorical_cols)

# Check the processed data and processed data type
print("\nThe head of data: ")
print(data.head())
print("\nTotal data (number of rows): ", data.shape[0])
print("Number of features (number of columns): ", data.shape[1])

# Extract features and labels
X = data.drop(columns=['HeartDisease']).values
y = data['HeartDisease'].values

# Data standardization
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Data set segmentation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)

# Convert to PyTorch's tensor format and create a DataLoader
batch_size = 256
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
                          generator=torch.Generator().manual_seed(seed))
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Figure 12: Data processing.

First, a random seed is set to ensure reproducibility during training. After removing missing values, categorical variables such as ”Sex” are converted into boolean variables using one-hot encoding [13], making them suitable for ANN processing. The dataset is then checked for missing values and data types. Subsequently, the target variable y is separated from the dataset, while the remaining features are set as X. The features are standardized to improve training efficiency. Finally, the dataset is split into 80% training data and 20% testing data. Then convert the data into PyTorch tensor format to efficiently handle high-dimensional data and support GPU acceleration. This is followed by creating DataLoader objects, which streamline the process of batching, shuffling, and loading data during training and evaluation. The processing code and outputs are shown in Figure 12 and Figure 13, separately.



```
The data type:
 Age                 int64
Sex                object
ChestPainType      object
RestingBP           int64
Cholesterol         int64
FastingBS           int64
RestingECG         object
MaxHR               int64
ExerciseAngina     object
Oldpeak           float64
ST_Slope           object
HeartDisease        int64
dtype: object

The head of data:
   Age  RestingBP  Cholesterol  FastingBS  MaxHR  Oldpeak  HeartDisease  \
0   40        140          289          0    172      0.0             0
1   49        160          180          0    156      1.0             1
2   37        130          283          0     98      0.0             0
3   48        138          214          0    108      1.5             1
4   54        150          195          0    122      0.0             0

   Sex_F  Sex_M  ChestPainType_ASY  ...  ChestPainType_NAP  ChestPainType_TA  \
0  False   True              False  ...              False             False
1   True  False              False  ...               True             False
2  False   True              False  ...              False             False
3   True  False               True  ...              False             False
4  False   True              False  ...               True             False

   RestingECG_LVH  RestingECG_Normal  RestingECG_ST  ExerciseAngina_N  \
0           False               True          False              True
1           False               True          False              True
2           False              False           True              True
3           False               True          False             False
4           False               True          False              True

   ExerciseAngina_Y  ST_Slope_Down  ST_Slope_Flat  ST_Slope_Up
0             False          False          False         True
1             False          False           True        False
2             False          False          False         True
3              True          False           True        False
4             False          False          False         True

[5 rows x 21 columns]

Total data (number of rows):  918
Number of features (number of columns):  21
```

Figure 13: Checking the output of the data.

2.2 Applicability of ANN for Heart Failure Prediction

This study focuses on the Heart Failure Prediction problem, a binary classification task. The objective is to determine whether a patient is at risk of heart failure based on medical attributes such as age, blood pressure, cholesterol levels, and ECG results.

Heart failure prediction is vital in healthcare, as early detection can enable timely interventions and reduce mortality. The dataset includes 11 features, a mix of continuous (e.g., cholesterol, age) and categorical variables (e.g., chest pain type, ECG results). The target variable is binary, indicating the presence or absence of heart failure.

ANNs are well-suited for this problem, as they can capture the complex, non-linear interactions among medical features that simpler models might overlook. This makes ANN a powerful tool for developing accurate data-driven systems to support critical decisions for healthcare professionals.

3 Evaluating and Analysing Model Performance

3.1 Model Performance Metric

In this task, there are 508 samples in class 1 (heart disease patients) and 410 samples in class 0 (non-patients), resulting in a relatively balanced dataset. Therefore, accuracy is chosen as the primary evaluation metric. Accuracy represents the proportion of samples correctly classified by the model, and its calculation formula is as follows [14]:

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

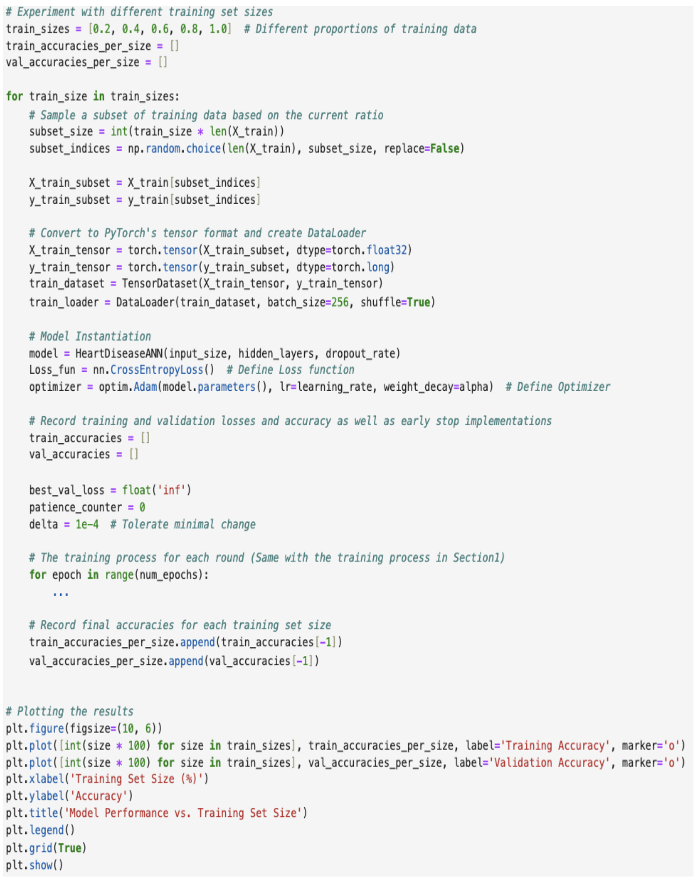
Where:

- TP represents True Positives (the number of samples correctly predicted as positive by the model).
- TN represents True Negatives (the number of samples correctly predicted as negative by the model).
- FP represents False Positives (the number of samples incorrectly predicted as positive by the model).
- FN represents False Negatives (the number of samples incorrectly predicted as negative by the model).

For this binary classification task with a relatively balanced dataset, accuracy serves as a quick and intuitive evaluation standard. However, in scenarios where class distributions are heavily imbalanced, accuracy might fail to fully capture the model's performance. In such cases, alternative metrics like the F1-score are more appropriate for evaluation.

3.2 Impact of Training Data Volume on Model

To explore the impact of training data size on model performance, 20% of the dataset is first fixed as the validation set. The remaining 80% is used as the training set, which is then divided into different proportions: 20%, 40%, 60%, 80% and 100%. The model is trained on each subset, and its performance on both the training and validation sets is recorded. Figure 14 is the implementation code, where the training process is the same as in Figure 5.



```python
# Experiment with different training set sizes
train_sizes = [0.2, 0.4, 0.6, 0.8, 1.0]  # Different proportions of training data
train_accuracies_per_size = []
val_accuracies_per_size = []

for train_size in train_sizes:
    # Sample a subset of training data based on the current ratio
    subset_size = int(train_size * len(X_train))
    subset_indices = np.random.choice(len(X_train), subset_size, replace=False)

    X_train_subset = X_train[subset_indices]
    y_train_subset = y_train[subset_indices]

    # Convert to PyTorch's tensor format and create DataLoader
    X_train_tensor = torch.tensor(X_train_subset, dtype=torch.float32)
    y_train_tensor = torch.tensor(y_train_subset, dtype=torch.long)
    train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
    train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True)

    # Model Instantiation
    model = HeartDiseaseANN(input_size, hidden_layers, dropout_rate)
    Loss_fun = nn.CrossEntropyLoss()  # Define Loss function
    optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=alpha)  # Define Optimizer

    # Record training and validation losses and accuracy as well as early stop implementations
    train_accuracies = []
    val_accuracies = []

    best_val_loss = float('inf')
    patience_counter = 0
    delta = 1e-4  # Tolerate minimal change

    # The training process for each round (Same with the training process in Section1)
    for epoch in range(num_epochs):
        ...

    # Record final accuracies for each training set size
    train_accuracies_per_size.append(train_accuracies[-1])
    val_accuracies_per_size.append(val_accuracies[-1])


# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot([int(size * 100) for size in train_sizes], train_accuracies_per_size, label='Training Accuracy', marker='o')
plt.plot([int(size * 100) for size in train_sizes], val_accuracies_per_size, label='Validation Accuracy', marker='o')
plt.xlabel('Training Set Size (%)')
plt.ylabel('Accuracy')
plt.title('Model Performance vs. Training Set Size')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 14: Code for impact of training data ratio on model accuracy.

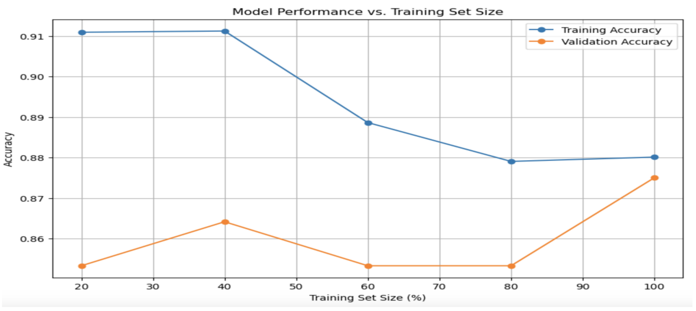
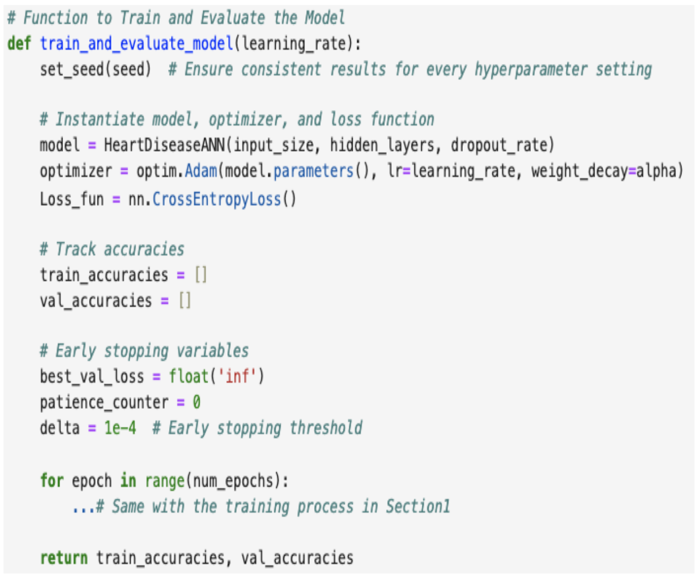


Figure 15: Impact of training data ratio on model accuracy: Orange is the verification set accuracy and blue is the training set accuracy.

The resulting accuracy is visualised in the Figure 15. It can be observed that the performance of the model under different training data proportions shows a clear phase change. At lower training size (20%-40%), the training accuracy is higher while the validation accuracy is lower. It indicates that the model may have overfitting phenomenon, which better fits a small amount of training data but has a weak generalisation ability. With the training size at 40%-60%, the training accuracy gradually decreases and the validation accuracy also decreases slightly. This may be due to the added data increasing the sample complexity, resulting in the model not being able to learn the features well and showing a slight underfitting. Interestingly, as the data is further increased to 80%-100%, the model's validation accuracy increases significantly, while the training accuracy also improves slightly. This is likely due to the fact that with more training data, the models capture more comprehensive information about the features, thus improving their generalisation ability. Overall, increasing the amount of data significantly improves the model's ability to learn and predict.

4 Explore Model Performance with Varying Hyperparameters

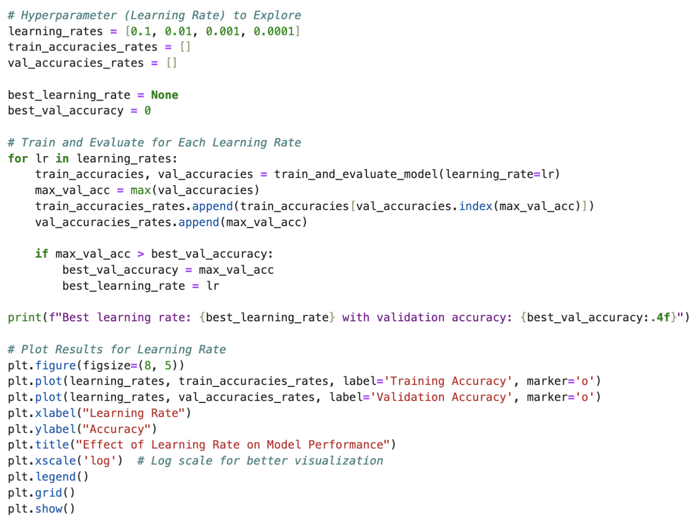
This task explored how adjusting the learning rate, a crucial hyperparameter, affects the performance of an ANN on the training and validation datasets. The learning rate directly influences the step size of gradient descent, which in turn controls the convergence rate and performance of the model [15]. To investigate its effects, I adjusted the learning rate exponentially by powers of 10 and performed a uniform search on a logarithmic scale. This approach ensured coverage of a wide range of learning rate values for a thorough analysis [16].



```
# Function to Train and Evaluate the Model
def train_and_evaluate_model(learning_rate):
    set_seed(seed)  # Ensure consistent results for every hyperparameter setting

    # Instantiate model, optimizer, and loss function
    model = HeartDiseaseANN(input_size, hidden_layers, dropout_rate)
    optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=alpha)
    Loss_fun = nn.CrossEntropyLoss()

    # Track accuracies
    train_accuracies = []
    val_accuracies = []

    # Early stopping variables
    best_val_loss = float('inf')
    patience_counter = 0
    delta = 1e-4  # Early stopping threshold

    for epoch in range(num_epochs):
        ...# Same with the training process in Section1

    return train_accuracies, val_accuracies
```

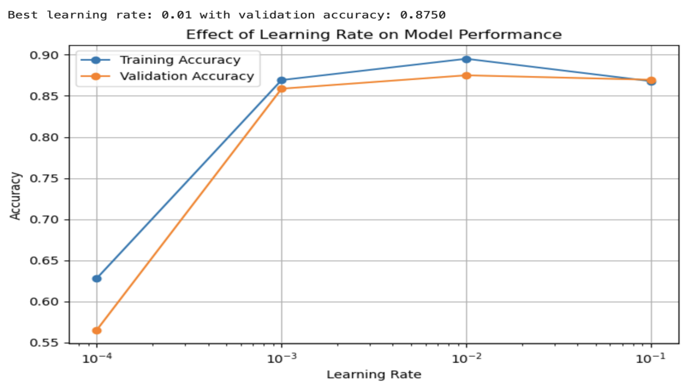
Figure 16: Function of "train_and_evaluate_model (learning_rate)".

To simplify the repeated training of the model, I defined a "`train_and_evaluate_model(learning_rate)`" function. The logic of the code within the function is the same as in Figure 16. As shown in Figure 17, this function loops through different learning rates to train the models, and records the best validation accuracy along with the corresponding training accuracy for visualisation. Besides, for better visualisation, I applied a logarithmic scale to the learning rate axis using "`plt.xscale('log')`".



```
# Hyperparameter (Learning Rate) to Explore
learning_rates = [0.1, 0.01, 0.001, 0.0001]
train_accuracies_rates = []
val_accuracies_rates = []

best_learning_rate = None
best_val_accuracy = 0

# Train and Evaluate for Each Learning Rate
for lr in learning_rates:
    train_accuracies, val_accuracies = train_and_evaluate_model(learning_rate=lr)
    max_val_acc = max(val_accuracies)
    train_accuracies_rates.append(train_accuracies[val_accuracies.index(max_val_acc)])
    val_accuracies_rates.append(max_val_acc)

    if max_val_acc > best_val_accuracy:
        best_val_accuracy = max_val_acc
        best_learning_rate = lr

print(f"Best learning rate: {best_learning_rate} with validation accuracy: {best_val_accuracy:.4f}")

# Plot Results for Learning Rate
plt.figure(figsize=(8, 5))
plt.plot(learning_rates, train_accuracies_rates, label='Training Accuracy', marker='o')
plt.plot(learning_rates, val_accuracies_rates, label='Validation Accuracy', marker='o')
plt.xlabel("Learning Rate")
plt.ylabel("Accuracy")
plt.title("Effect of Learning Rate on Model Performance")
plt.xscale('log')  # Log scale for better visualization
plt.legend()
plt.grid()
plt.show()
```

Figure 17: Code for exploring learning rates.

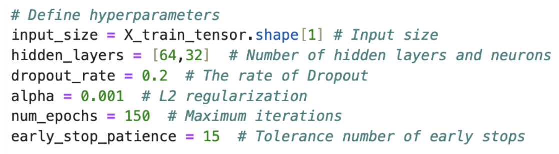


```
Best learning rate: 0.01 with validation accuracy: 0.8750
```

Figure 18: Results for exploring learning rates: Orange is the verification set accuracy and blue is the training set accuracy.

The results are shown in Figure 18. When the learning rate was 10^{-4}, both training and validation accuracies were low. This occurred because the small learning rate prevented the model from fully converging within the given number of iterations. As a result, the model failed to learn enough features, leading to underfitting. At 10^{-3}, the model's performance improved significantly. Because the learning rate is high enough to converge efficiently, which greatly improves the training and validation accuracy. At 10^{-2}, the validation accuracy reached its peak at 0.8750. This indicates that the model learned the patterns in the training data well while maintaining strong generalisation on the validation set. However, at 10^{-1}, the training accuracy dropped significantly, and the validation accuracy also declined slightly. Interestingly, the validation accuracy was slightly higher than the training accuracy at this learning rate. This uncommon phenomenon may result from the high learning rate causing instability in the optimization process, preventing the model from minimizing training loss effectively. Additionally, regularisation techniques such as Dropout was applied during training but not during validation. This might also explain why the validation accuracy appeared higher than the training accuracy.

To ensure a fair comparison across different learning rates, I kept all other hyperparameters fixed, as shown in Figure 19. Additionally, I increased the early stopping patience from 10 to 15. This change allowed models with smaller learning rates to have sufficient time to converge, preventing premature stopping, especially at 10^{-4}.



```
# Define hyperparameters
input_size = X_train_tensor.shape[1] # Input size
hidden_layers = [64,32]  # Number of hidden layers and neurons
dropout_rate = 0.2  # The rate of Dropout
alpha = 0.001  # L2 regularization
num_epochs = 150  # Maximum iterations
early_stop_patience = 15  # Tolerance number of early stops
```

Figure 19: The fixed hyperparameters when exploring the learning rate.

In conclusion, the results show that 10^{-2} is the optimal learning rate for this task. It achieved the best balance between training and validation performance. Extremely small or large learning rates significantly affected the model's performance, highlighting the importance of carefully tuning the learning rate during the training of ANNs.

5 Exploring the Impact of Cross-Validation on Hyperparameter Optimisation

In this task, learning rate was chosen as the hyperparameter to explore the effect of cross-validation on model performance. By comparing the results of single-split validation and cross-validation, the aim is to understand the advantages of cross-validation in hyperparameter selection.

5.1 The Process of Cross-Validation

Cross-validation is a widely used method for evaluating model performance [17]. The method involves splitting the dataset into multiple folds, with one fold being selected as the validation set while the remaining folds are used for training. This process, illustrated in Figure 20, evaluates the model over multiple training-validation splits. The final performance is calculated as the average of the validation metrics across all folds. Compared to a single validation split, cross-validation reduces variability caused by dataset partitioning, offering a more robust evaluation of hyperparameters.

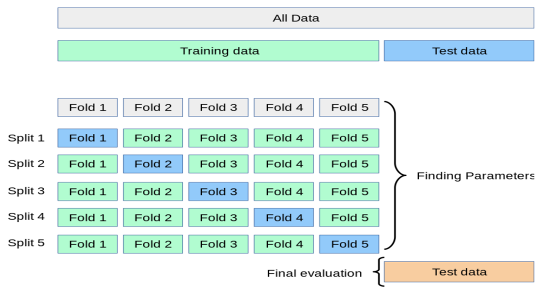
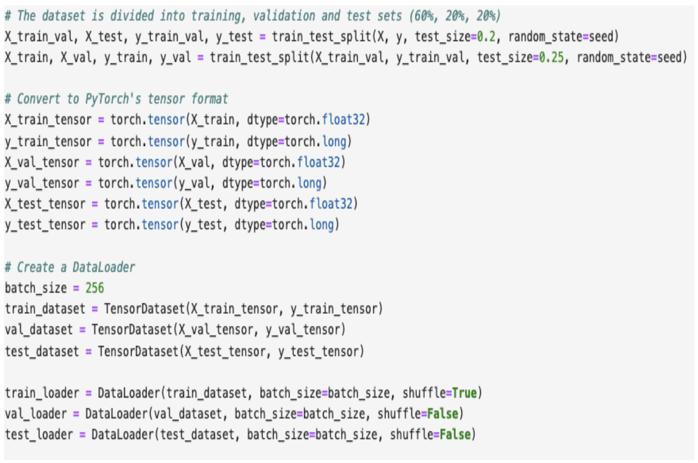


Figure 20: The process of Cross-validation [17].

5.2 Experimental Design

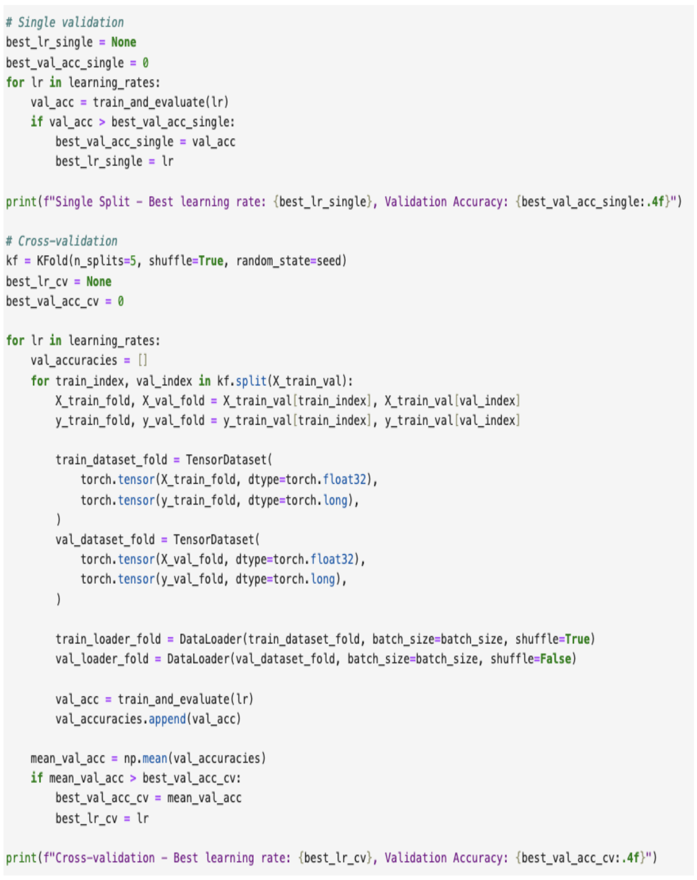
As shown in Figure 21, the dataset was divided into 60%, 20%, and 20% proportions for training, validation, and testing, respectively. Then, these subsets were changed into PyTorch tensors for model training. To compare the performance of hyperparameters selected through single-split validation and cross-validation, I first searched for the learning rate that achieved the highest validation accuracy. This was done separately for both methods. The best learning rate was then used to train the entire training dataset (original training and validation sets combined). Finally, the model was evaluated on the test set to measure its performance.



```
# The dataset is divided into training, validation and test sets (60%, 20%, 20%)
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25, random_state=seed)

# Convert to PyTorch's tensor format
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_val_tensor = torch.tensor(X_val, dtype=torch.float32)
y_val_tensor = torch.tensor(y_val, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Create a DataLoader
batch_size = 256
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Figure 21: Code for splitting data.

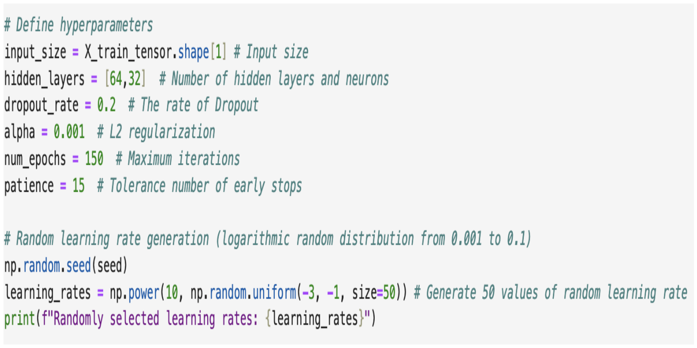
Figure 22 shows the implementation of single-split validation and cross-validation. For cross-validation, I used the "`KFold(n_splits=5, shuffle=True, random_state=seed)`" function to divide the dataset into five folds. The model was trained and evaluated on various training-validation combinations across all folds.



```python
# Single validation
best_lr_single = None
best_val_acc_single = 0
for lr in learning_rates:
    val_acc = train_and_evaluate(lr)
    if val_acc > best_val_acc_single:
        best_val_acc_single = val_acc
        best_lr_single = lr

print(f"Single Split - Best learning rate: {best_lr_single}, Validation Accuracy: {best_val_acc_single:.4f}")

# Cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=seed)
best_lr_cv = None
best_val_acc_cv = 0

for lr in learning_rates:
    val_accuracies = []
    for train_index, val_index in kf.split(X_train_val):
        X_train_fold, X_val_fold = X_train_val[train_index], X_train_val[val_index]
        y_train_fold, y_val_fold = y_train_val[train_index], y_train_val[val_index]

        train_dataset_fold = TensorDataset(
            torch.tensor(X_train_fold, dtype=torch.float32),
            torch.tensor(y_train_fold, dtype=torch.long),
        )
        val_dataset_fold = TensorDataset(
            torch.tensor(X_val_fold, dtype=torch.float32),
            torch.tensor(y_val_fold, dtype=torch.long),
        )

        train_loader_fold = DataLoader(train_dataset_fold, batch_size=batch_size, shuffle=True)
        val_loader_fold = DataLoader(val_dataset_fold, batch_size=batch_size, shuffle=False)

        val_acc = train_and_evaluate(lr)
        val_accuracies.append(val_acc)

    mean_val_acc = np.mean(val_accuracies)
    if mean_val_acc > best_val_acc_cv:
        best_val_acc_cv = mean_val_acc
        best_lr_cv = lr

print(f"Cross-validation - Best learning rate: {best_lr_cv}, Validation Accuracy: {best_val_acc_cv:.4f}")
```

Figure 22: The implementation of single-split validation and cross-validation.

Besides, as shown in Figure 23, I used a simulated random search method to construct the parameter space. The random search is an efficient method for hyperparametric optimisation [18]. Specifically, 50 exponentially scaled learning rate values were randomly sampled within the range [0.001, 0.1]. In continuous parameter spaces, random search is better than grid search. It allows to explore a broader range of potential values more efficiently. By focusing on randomly selected configurations, random search can reduce the computational cost while still identifying hyperparameters that are close to optimal. This makes it an effective method for high-dimensional or computationally intensive optimisation tasks.



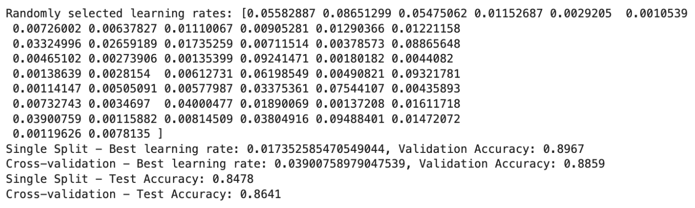
```python
# Define hyperparameters
input_size = X_train_tensor.shape[1] # Input size
hidden_layers = [64,32]  # Number of hidden layers and neurons
dropout_rate = 0.2  # The rate of Dropout
alpha = 0.001  # L2 regularization
num_epochs = 150  # Maximum iterations
patience = 15  # Tolerance number of early stops

# Random learning rate generation (logarithmic random distribution from 0.001 to 0.1)
np.random.seed(seed)
learning_rates = np.power(10, np.random.uniform(-3, -1, size=50)) # Generate 50 values of random learning rate
print(f"Randomly selected learning rates: {learning_rates}")
```

Figure 23: The implementation of single-split validation and cross-validation.

5.3 Results and Analysis

The results of the experiment are presented in Figure 24:

- The single-split method found an optimal learning rate of `0.01735`, achieving a validation accuracy of **89.67%** and a test accuracy of **84.78%**.
- The cross-validation method identified an optimal learning rate of `0.03901`, with an average validation accuracy of **88.59%** and a test accuracy of **86.41%**.



```
Randomly selected learning rates: [0.05582887 0.08651299 0.05475062 0.01152687 0.0029205  0.0010539
 0.00726002 0.00637827 0.01110067 0.00905281 0.01290366 0.01221158
 0.03324996 0.02659189 0.01735259 0.00711514 0.00378573 0.08865648
 0.00465102 0.00273906 0.00135399 0.09241471 0.00180182 0.0044082
 0.00138639 0.0028154  0.00612731 0.06198549 0.00490821 0.09321781
 0.00114147 0.00505091 0.00577987 0.03375361 0.07544107 0.00435893
 0.00732743 0.0034697  0.04000477 0.01890069 0.00137208 0.01611718
 0.03900759 0.00115882 0.00814509 0.03804916 0.09488401 0.01472072
 0.00119626 0.0078135 ]
Single Split - Best learning rate: 0.017352585470549044, Validation Accuracy: 0.8967
Cross-validation - Best learning rate: 0.03900758979047539, Validation Accuracy: 0.8859
Single Split - Test Accuracy: 0.8478
Cross-validation - Test Accuracy: 0.8641
```

Figure 24: The results of single-split validation and cross-validation.

These results can be observed that the validation accuracy of the single-split method is 89.67%, which is slightly higher than that of the cross-validation 88.59%. However, its accuracy is lower than 86.41% of cross-validation. This shows that the single-split approach may overfit the special validation set, resulting in poorer generalisation on unseen data. In contrast, cross-validation showed better generalisation by considering multiple training-validation splits. This makes it more reliable to evaluate the hyperparameter.

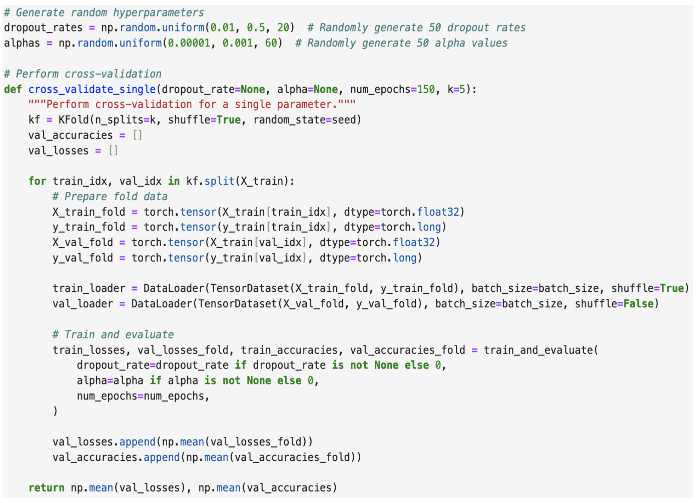
In conclusion, this experiment shows the advantages of cross-validation in hyperparameter optimisation. Because, compared to the single split validation, cross-validation reduces the impact of random dataset segmentation by averaging the performance of multiple folds. Future research could explore a wider range of parameters or integrate other optimisation techniques to further improve model performance.

6 Further Exploring

In this experiment, regularisation techniques were studied to evaluate their impact on the performance of Artificial Neural Networks (ANN). The study followed an ablation experimental design and employed the control variable method for analysis. Two widely-used regularisation methods, L2 regularisation (weight decay *alpha*) and Dropout, were examined, both individually and in combination. The main aim was to evaluate the model's training and validation performance under various regularisation settings and analyse its effectiveness in reducing overfitting.

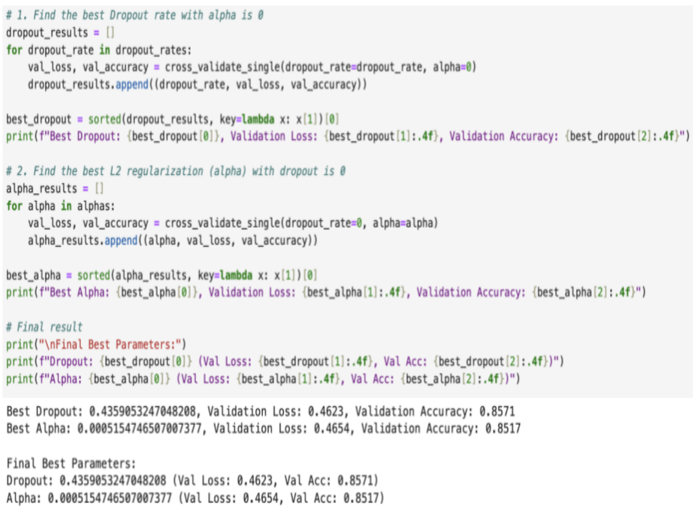
6.1 Methods and Experimental Design

First, I used random search and cross-validation to determine the optimal hyperparameters for Dropout and *alpha*, respectively. Specifically, I generated 20 random dropout rates within the range of 0.01 to 0.5 and 60 *alpha* values within the range of 0.00001 to 0.001. As shown in Figure 25 and Figure 26, the performance of each parameter was evaluated using a 5-fold cross-validation function. The parameter with the lowest mean validation loss was selected as the optimal choice. The application of cross-validation helps reduce the randomness introduced by a single data split and improves the reliability of the results.



```python
# Generate random hyperparameters
dropout_rates = np.random.uniform(0.01, 0.5, 20)  # Randomly generate 50 dropout rates
alphas = np.random.uniform(0.00001, 0.001, 60)  # Randomly generate 50 alpha values

# Perform cross-validation
def cross_validate_single(dropout_rate=None, alpha=None, num_epochs=150, k=5):
    """Perform cross-validation for a single parameter."""
    kf = KFold(n_splits=k, shuffle=True, random_state=seed)
    val_accuracies = []
    val_losses = []

    for train_idx, val_idx in kf.split(X_train):
        # Prepare fold data
        X_train_fold = torch.tensor(X_train[train_idx], dtype=torch.float32)
        y_train_fold = torch.tensor(y_train[train_idx], dtype=torch.long)
        X_val_fold = torch.tensor(X_train[val_idx], dtype=torch.float32)
        y_val_fold = torch.tensor(y_train[val_idx], dtype=torch.long)

        train_loader = DataLoader(TensorDataset(X_train_fold, y_train_fold), batch_size=batch_size, shuffle=True)
        val_loader = DataLoader(TensorDataset(X_val_fold, y_val_fold), batch_size=batch_size, shuffle=False)

        # Train and evaluate
        train_losses, val_losses_fold, train_accuracies, val_accuracies_fold = train_and_evaluate(
            dropout_rate=dropout_rate if dropout_rate is not None else 0,
            alpha=alpha if alpha is not None else 0,
            num_epochs=num_epochs,
        )

        val_losses.append(np.mean(val_losses_fold))
        val_accuracies.append(np.mean(val_accuracies_fold))

    return np.mean(val_losses), np.mean(val_accuracies)
```

Figure 25: Define a 5-fold cross-validation function.

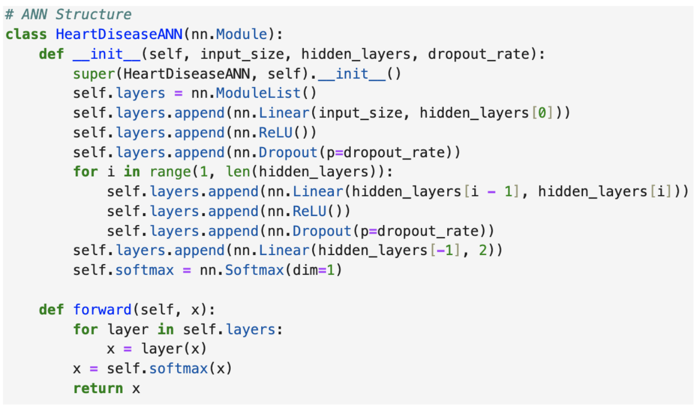


```python
# 1. Find the best Dropout rate with alpha is 0
dropout_results = []
for dropout_rate in dropout_rates:
    val_loss, val_accuracy = cross_validate_single(dropout_rate=dropout_rate, alpha=0)
    dropout_results.append((dropout_rate, val_loss, val_accuracy))

best_dropout = sorted(dropout_results, key=lambda x: x[1])[0]
print(f"Best Dropout: {best_dropout[0]}, Validation Loss: {best_dropout[1]:.4f}, Validation Accuracy: {best_dropout[2]:.4f}")

# 2. Find the best L2 regularization (alpha) with dropout is 0
alpha_results = []
for alpha in alphas:
    val_loss, val_accuracy = cross_validate_single(dropout_rate=0, alpha=alpha)
    alpha_results.append((alpha, val_loss, val_accuracy))

best_alpha = sorted(alpha_results, key=lambda x: x[1])[0]
print(f"Best Alpha: {best_alpha[0]}, Validation Loss: {best_alpha[1]:.4f}, Validation Accuracy: {best_alpha[2]:.4f}")

# Final result
print("\nFinal Best Parameters:")
print(f"Dropout: {best_dropout[0]} (Val Loss: {best_dropout[1]:.4f}, Val Acc: {best_dropout[2]:.4f})")
print(f"Alpha: {best_alpha[0]} (Val Loss: {best_alpha[1]:.4f}, Val Acc: {best_alpha[2]:.4f})")
```

```
Best Dropout: 0.4359053247048208, Validation Loss: 0.4623, Validation Accuracy: 0.8571
Best Alpha: 0.0005154746507007377, Validation Loss: 0.4654, Validation Accuracy: 0.8517

Final Best Parameters:
Dropout: 0.4359053247048208 (Val Loss: 0.4623, Val Acc: 0.8571)
Alpha: 0.0005154746507007377 (Val Loss: 0.4654, Val Acc: 0.8517)
```

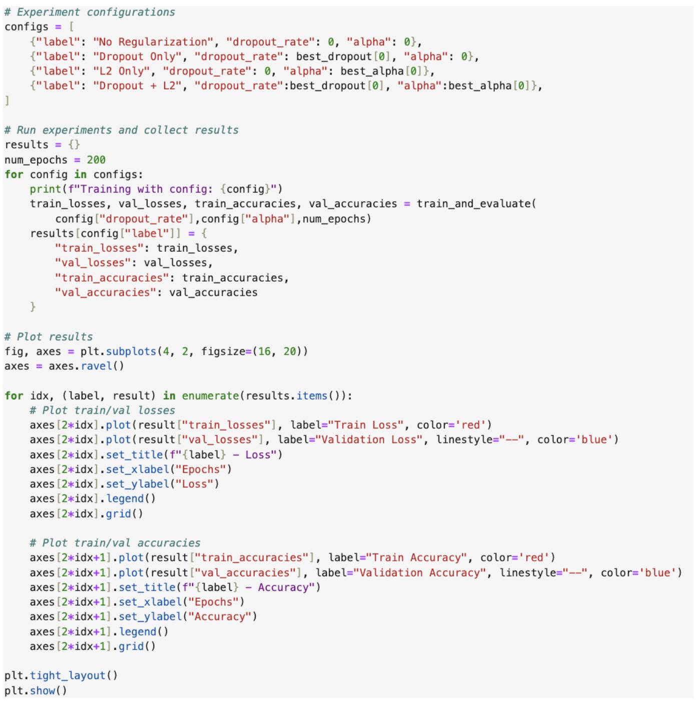
Figure 26: Printing the Best Alpha and Dropout Values.

Next, I trained the model under four different configurations: (1) no regularisation; (2) using only Dropout with the optimal dropout rate; (3) L2 regularisation only (using the best *alpha*); and (4) a combination of Dropout and L2 regularisation. All configurations used the same neural network structure (detailed in 27). The learning rate was fixed at 0.001, hidden layers were set to [64, 32], and each configuration was trained for 200 epochs without early stopping. Removing early stopping aimed to fully observe the effect of regularisation over prolonged training. This approach allows us to identify whether the model overfits during extended training or if validation reaches a steady level of performance.



```python
# ANN Structure
class HeartDiseaseANN(nn.Module):
    def __init__(self, input_size, hidden_layers, dropout_rate):
        super(HeartDiseaseANN, self).__init__()
        self.layers = nn.ModuleList()
        self.layers.append(nn.Linear(input_size, hidden_layers[0]))
        self.layers.append(nn.ReLU())
        self.layers.append(nn.Dropout(p=dropout_rate))
        for i in range(1, len(hidden_layers)):
            self.layers.append(nn.Linear(hidden_layers[i - 1], hidden_layers[i]))
            self.layers.append(nn.ReLU())
            self.layers.append(nn.Dropout(p=dropout_rate))
        self.layers.append(nn.Linear(hidden_layers[-1], 2))
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        x = self.softmax(x)
        return x
```

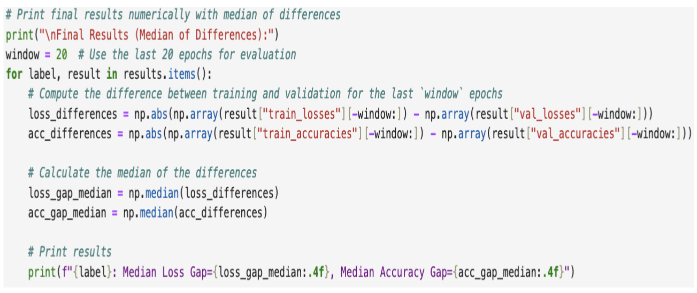
Figure 27: ANN structure used to explore regularisation techniques.



```python
# Experiment configurations
configs = [
    {"label": "No Regularization", "dropout_rate": 0, "alpha": 0},
    {"label": "Dropout Only", "dropout_rate": best_dropout[0], "alpha": 0},
    {"label": "L2 Only", "dropout_rate": 0, "alpha": best_alpha[0]},
    {"label": "Dropout + L2", "dropout_rate":best_dropout[0], "alpha":best_alpha[0]},
]

# Run experiments and collect results
results = {}
num_epochs = 200
for config in configs:
    print(f"Training with config: {config}")
    train_losses, val_losses, train_accuracies, val_accuracies = train_and_evaluate(
        config["dropout_rate"],config["alpha"],num_epochs)
    results[config["label"]] = {
        "train_losses": train_losses,
        "val_losses": val_losses,
        "train_accuracies": train_accuracies,
        "val_accuracies": val_accuracies
    }

# Plot results
fig, axes = plt.subplots(4, 2, figsize=(16, 20))
axes = axes.ravel()

for idx, (label, result) in enumerate(results.items()):
    # Plot train/val losses
    axes[2*idx].plot(result["train_losses"], label="Train Loss", color='red')
    axes[2*idx].plot(result["val_losses"], label="Validation Loss", linestyle="--", color='blue')
    axes[2*idx].set_title(f"{label} - Loss")
    axes[2*idx].set_xlabel("Epochs")
    axes[2*idx].set_ylabel("Loss")
    axes[2*idx].legend()
    axes[2*idx].grid()

    # Plot train/val accuracies
    axes[2*idx+1].plot(result["train_accuracies"], label="Train Accuracy", color='red')
    axes[2*idx+1].plot(result["val_accuracies"], label="Validation Accuracy", linestyle="--", color='blue')
    axes[2*idx+1].set_title(f"{label} - Accuracy")
    axes[2*idx+1].set_xlabel("Epochs")
    axes[2*idx+1].set_ylabel("Accuracy")
    axes[2*idx+1].legend()
    axes[2*idx+1].grid()

plt.tight_layout()
plt.show()
```

Figure 28: Code for plotting loss curves and accuracy curves.

The results of the experiments are presented through the code in Figure 28, which plots the training and validation loss curves and accuracy curves for each regularisation configuration. It is also analysed more visually by outputting the median of the difference between the performance of the last 20 rounds of training and validation sets through the code in Figure 29. The median is used to reduce the impact of late fluctuations on the results and to more accurately reflect the difference in model performance during the stabilisation phase.

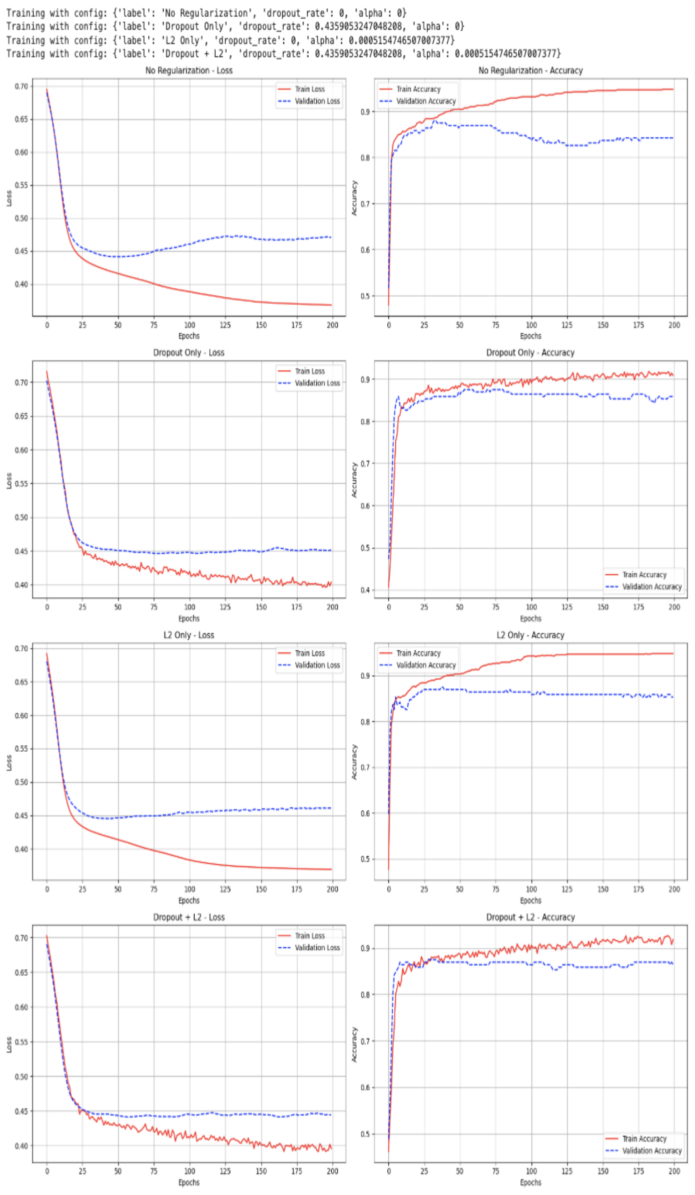


```python
# Print final results numerically with median of differences
print("\nFinal Results (Median of Differences):")
window = 20  # Use the last 20 epochs for evaluation
for label, result in results.items():
    # Compute the difference between training and validation for the last `window` epochs
    loss_differences = np.abs(np.array(result["train_losses"][-window:]) - np.array(result["val_losses"][-window:]))
    acc_differences = np.abs(np.array(result["train_accuracies"][-window:]) - np.array(result["val_accuracies"][-window:]))

    # Calculate the median of the differences
    loss_gap_median = np.median(loss_differences)
    acc_gap_median = np.median(acc_differences)

    # Print results
    print(f"{label}: Median Loss Gap={loss_gap_median:.4f}, Median Accuracy Gap={acc_gap_median:.4f}")
```

Figure 29: Code for median of loss and accuracy gap.

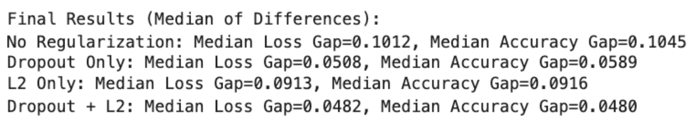
6.2 Results and Analysis

From the plots (Figure 30 and Figure 31), the following key observations were made:



```
Training with config: {'label': 'No Regularization', 'dropout_rate': 0, 'alpha': 0}
Training with config: {'label': 'Dropout Only', 'dropout_rate': 0.4359053247048208, 'alpha': 0}
Training with config: {'label': 'L2 Only', 'dropout_rate': 0, 'alpha': 0.0005154746507007377}
Training with config: {'label': 'Dropout + L2', 'dropout_rate': 0.4359053247048208, 'alpha': 0.0005154746507007377}
```

Figure 30: Loss and accuracy curves for each method: on the left is the loss curve and on the right is the accuracy curve. Where red is the training set and blue is the validation set.



```
Final Results (Median of Differences):
No Regularization: Median Loss Gap=0.1012, Median Accuracy Gap=0.1045
Dropout Only: Median Loss Gap=0.0508, Median Accuracy Gap=0.0589
L2 Only: Median Loss Gap=0.0913, Median Accuracy Gap=0.0916
Dropout + L2: Median Loss Gap=0.0482, Median Accuracy Gap=0.0480
```

Figure 31: Results for median loss and accuracy gap.

No regularisation: The model without regularisation showed a significant gap between training and validation performance. The median loss gap was 0.1012, and the median accuracy gap was 0.1045, showing severe overfitting. While the training performance was excellent, the validation accurance was lower. This may be due to poor generalisation caused by the model overlearning some details in the training set. Furthermore, the validation accuracy peaks at an early stage and gradually decreases as the number of rounds increases, further emphasising the overfitting issue.

Dropout Only: Dropout significantly reduced the performance gap between training and validation sets. The median loss gap decreased to 0.0508, and the median accuracy gap to 0.0589, showing improved generalisation ability. This is likely because Dropout forces the model to learn a different combination of features in each training by randomly masking some of the neurons. This makes the model less reliant on specific features, which improves generalisation. However, the randomness introduced by Dropout may lead to some instability, which may require further studies when combined with other regularisation techniques.

L2 regularisation Only: L2 regularisation showed limited improvement. The median loss gap was 0.0913, and the median accuracy gap was 0.0916. Compared to no regularisation, the improvement was minor. The possibly reason is that the dataset was small and the model's learning capacity was relatively high. In this case, L2 regularisation, which works by limiting the size of model weights, had less of an impact.

Dropout and L2 Combined: This configuration achieved the best performance, with the median loss gap reduced to 0.0482 and the median accuracy gap to 0.0480. It suggests that the combination of Dropout and L2 regularisation method amplifies their individual benefits. Dropout reduces overfitting by adding randomness, while L2 further smoothens the weight updates, resulting in superior performance.

Additional, a curious observation was that when regularisation, especially Dropout was applied, validation accuracy often exceeded training accuracy during the early stages of training. This may happen because regularisation limits the model's ability to fit the training data too quickly. However, since regularisation is not applied during validation, the validation performance improves faster at first. This highlights how regularisation improves the generalisation ability of the model by restricting overfitting to the training data.

6.3 Conclusion

Based on the results of this experiment, the following conclusions can be drawn:

- Models without regularisation are easy to overfitting, resulting in poor validation performance.
- Dropout is better than L2 regularisation for avoiding overfitting and improving validation.
- The combination of Dropout and L2 regularisation achieves the best performance, with the smallest gap between training and validation performance, demonstrating superior generalisation ability.

This experiment also raises interesting questions about the complementary effects of regularisation techniques. The combination of Dropout and L2 regularisation might involve not only parameter interactions but also structural diversity and balance within the model. Future research could delve deeper into the theoretical basis and practical implications of combining different regularisation methods. For instance, studies could examine how these techniques interact with other hyperparameters or evaluate their effectiveness on larger, more complex datasets to uncover broader patterns and insights.

References

[1] GeeksforGeeks, n.d. Supervised vs Unsupervised Learning. [Online]. Available: https://www.geeksforgeeks.org/supervised-unsupervised-learning/. [Accessed: 16-Nov-2024].

[2] Hastie, T., Tibshirani, R., & Friedman, J., 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer. [Online]. Available: https://link.springer.com/book/10.1007/978-0-387-84858-7.

[3] Tzzdq, n.d. [Online]. Available: https://tzzdq.com/1397/. [Accessed: 16-Nov-2024].

[4] Agarap, A. F., 2018. Deep Learning using Rectified Linear Units (ReLU). *CoRR*, vol. abs/1803.08375. [Online]. Available: http://arxiv.org/abs/1803.08375.

[5] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R., 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html.

[6] Papers with Code, n.d. Softmax. [Online]. Available: https://paperswithcode.com/method/softmax. [Accessed: 16-Nov-2024].

[7] DataCamp, n.d. The Cross-Entropy Loss Function in Machine Learning. [Online]. Available: https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning. [Accessed: 16-Nov-2024].

[8] Kingma, D. P., & Ba, J., 2017. Adam: A Method for Stochastic Optimization. [Online]. Available: https://arxiv.org/abs/1412.6980.

[9] Zhihu, n.d. Understanding Cross-Entropy Loss Function. [Online]. Available: https://zhuanlan.zhihu.com/p/447113449. [Accessed: 16-Nov-2024].

[10] MathCentre, 2009. Chain Rule. [Online]. Available: https://www.mathcentre.ac.uk/resources/uploaded/mc-ty-chain-2009-1.pdf. [Accessed: 16-Nov-2024].

[11] Baidu Cloud, n.d. Understanding Cloud Computing. [Online]. Available: https://cloud.baidu.com/article/1896200. [Accessed: 16-Nov-2024].

[12] Fedesoriano, September 2021. Heart Failure Prediction Dataset. [Online]. Available: https://www.kaggle.com/fedesoriano/heart-failure-prediction. [Accessed: 12-Nov-2024].

[13] GeeksforGeeks, n.d. ML — One Hot Encoding of Datasets in Python. [Online]. Available: https://www.geeksforgeeks.org/ml-one-hot-encoding-of-datasets-in-python/. [Accessed: 19-Nov-2024].

[14] Google Developers, n.d. Accuracy, Precision, and Recall: Classification. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall. [Accessed: 16-Nov-2024].

[15] Smith, L. N., 2017. Cyclical Learning Rates for Training Neural Networks. In *Proceedings of the 2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472. [Online]. Available: https://doi.org/10.1109/WACV.2017.58.

[16] Amazon Web Services, n.d. Best Practices for Hyperparameter Tuning. Retrieved December 3, 2023, from Amazon SageMaker Documentation: https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning-define-ranges.html.

[17] Scikit-learn, n.d. Cross-validation: evaluating estimator performance. [Online]. Available: https://scikit-learn.org/1.5/modules/cross_validation.html. [Accessed: 21-Nov-2024].

[18] Bergstra, J., & Bengio, Y., 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, vol. 13, no. 2.