In [ ]:
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

Audio Machine Learning - Summative Assessment 1 - Machine Learning Challenge - Part 1

For the first part of the machine learning challenge, your task is to implement and train a binary classifier neural network using PyTorch.

There are sections in the Notebook which are left for you to complete, which will be assessed to determine your overall grade for this assessment. These will be marked as below:

Assessed Section

In [ ]:
```
# Any code cells in the 'Assessed Section' will be graded as part of the
# There will be instructions for you to follow within these parts of the
# Your submission will be the Notebook file, with the Assessed Sections f
# This one doesn't count, as it is just a demonstration!
```

End of Assessed Section

Task 1 - Dataset Loading and Visualisation

The dataset is currently stored in two numpy files included with this Notebook, 'class_0.npy' and 'class_1.npy'. If you are using Notebable, you will need to upload these files to Noteable.

Each numpy file contains a dataset of examples of a class, either 'class 0' or 'class 1'. The data is held in a numpy array of the following shape:

(N, j)

Where the N denotes how many data points are held in the dataset, and j denotes how many features describe each data point.

Before training a machine learning model, it is important to split your dataset into three subsets: a *training* set, a *validation* set, and a *test* set.

The *training* subset should contain 60% of the datapoints, the *validation* subset should contain 20% of the datapoints, and the *test* subset should contain the remaining 20% of the datapoints.

Each subset should consist of a torch tensor holding the features, and a torch tensor holding the corresponding labels. For example, if the first datapoint of the subset contains an example of class 0, then 'subset_features[0, :]' should return the features of the first datapoint in the subset, and subset_labels[0,:] should return the label of the first datapoint in the subset, in this case '0'.

Each feature tensor should be a torch tensor of shape (N, j), where N is however many datapoints are in this subset. Each label tensor should be a torch tensor of shape $(N, 1)$.

- Ensure that there is no overlap between the subsets. Each datapoint should appear exactly once in one of the three subsets. For example, a datapoint in the training subset should not appear in either the validation or test subsets.
- Ensure that each subset contains the same number of examples from both classes. Each subset should consist of exactly 50% datapoints of class 'class_0' and exactly 50% datapoints of class 'class_1'
- Use the torch.from_numpy() function to convert numpy arrays into Torch tensors

Assessed Section - 1.1

Your task is to:

- Load the data held in the files 'class_0.npy' and 'class_1.npy'.
- Create a scatter plot using MatPlotLib (imported in this NoteBook as 'plt'). Each datapoint should appear on the scatter plot, with the x-axis indicating the value of the first feature, and the y-axis indicating the value of the second feature. The two classes should be labelled using the plt.legend() method.
- Split the dataset into the three subsets described in the previous cell.

In [ ]:
```
# Your Solution Here
```

In [ ]:
```
# You may add as many cells as you like for your solution. Just ensure th
```

End of Assessed Section - 1.1

Task 2 - Neural Network Initialisation

We want to make a classifier, that attempts to predict the class of a datapoint, from the datapoint's features.

We will do this using a neural network. It should take the input features of any number of datapoints as input, and process them to produce a prediction of the class corresponding to the input features of each datatpoint.

The neural network should consist of:

- A Linear Layer
- A non-linear activation function layer
- A second Linear Layer.
- A final non-linear activation function layer

The neural network should have a 'forward' method that processes input through the layers, in the order they are listed above.

The final non-linear activation function layer should force the outputs to be in range [0, 1]. (We learned about an appropriate function for doing this in Lecture 3, on logistic regression!)

Define the layers of the neural network in the 'constructor function' of the neural network class. Each layer should be a PyTorch layer from the 'torch.nn' library.

Assessed Section - 1.2

Your task is to complete the Neural Network class template, 'NNet' below:

- Define the layers of the Neural Network in the class constructor, as described in the previous cell.
- Define the forward function of the Neural Network, as described in the previous cell.
- Create an instance of your Neural Network, and process some data with it.

In [ ]:
```
class NNet(nn.Module):                  # Don't change this line
    def __init__(self):                 # Don't change this line
        super(NNet, self).__init__() # Don't change this line
        # This is the constructor function
        # define your neural network layers here
        # Don't forget to save them as class attributes using 'self.'

    def forward(self, x):               # Don't change this line
        # This is the forward pass of the neural network
        # It should receive an input tensor, x, which contains N datapoin
        # The shape of the input tensor x should be 'x.shape = (N, j)'
        # The forward pass should use the layers defined in the construct
        # predictions of the class.
```

```
        # Finally, it should return the predicted class labels using the
        return # Return the neural network's predicted labels here
```

In [ ]:
```
# You may add as many cells as you like for your solution. Just ensure th
```

End of Assessed Section - 1.2

Task 3 - Binary Cross Entropy Loss

The binary cross entropy loss function measures the difference between neural network predictions, and the true class labels.

It is given by the following formula:

$$L(y^{(i)},\hat{y}^{(i)})=-(y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)}))$$

Where log is the natural logarithm function, $y^{(i)}$ is the target label for the $i-th$ datapoint, and $\hat{y}^{(i)}$ is the model's predicted label for $i-th$ datapoint.

This calculates the loss for a single datapoint, i. To find the loss for a batch of datapoints, calculate the loss for each individual datapoint, and take the mean of the loss over the batch.

Assessed Section - 1.3

Your task is to implement the Binary Cross-Entropy loss function in the template below:

- The loss function should take two PyTorch tensors as inputs, a batch of N model predictions, y_hat, and the corresponding N target labels, y.
- Both input tensors should be of shape $(N, 1)$.
- As the input to the natural log function tends to 0, the output approaches negative infinity. In your BCE loss function implementation you can avoid negative infinity terms by applying the torch.clamp() function to inputs to the log function.
- You may **only** use the torch.log(), torch.clamp(), and torch.mean() functions from the torch library for your implementation.
- The loss function should return a tensor, that is a scalar, that corresponds to the mean BCE loss over the batch that was input to the function.

In [ ]:
```
class BCELoss(nn.Module):                # Don't change this line
    def __init__(self):                  # Don't change this line
        super(BCELoss, self).__init__()# Don't change this line
        pass

    def forward(self, y, y_hat): # Don't change this line
        # Implement binary cross entropy loss here
        return  # Return the binary cross entropy loss here
```

End of Assessed Section - 1.3

Task 4 - Neural Network Training

Implement a neural network training loop that carries out batch gradient descent on your training dataset.

Create an instance of the PyTorch torch.optim.SGD() optimiser to update your model parameters.

For each training iteration:

- Zero the gradients in the optimiser
- Pass the features of the full training dataset to the neural network to get the model's predictions
- Calculate the loss (you may use the PyTorch BCE Loss function if you're uncertain about your implementation - you will not lose any marks for doing this)
- Call the backward method on the loss
- Update the neural network model's parameters using the optimiser.step() method
- Append the training loss to a list of training losses

After every 20th training iteration, find the validation loss and accuracy:

- Pass the features of the full validation dataset to the neural network to get the model's predictions
- Calculate the loss
- Find the model's classification accuracy on the validation set (assume a decision boundary at y_hat=0.5)
- Append the validation loss to a list of validation losses, and append the validation accuracy to a list of validation accuracies.

Assessed Section - 1.4

Impletment the neural network training and validation as descibed above:

- Run the training for 1000 iterations, whilst monitoring the validation loss and accuracy
- Plot the training and validation losses against the training iteration number
- Plot the validation accuracy against the training iteration number
- Calculate the model's loss and accuracy on the test subset

In [ ]:
```
# Put your neural network training code here
# You will not assessed on the accuracy or loss achieved by your model, b
```

End of Assessed Section - 1.4

Further Work

The above is sufficient to achieve a mark of 60 for Postgraduate students and 67 for Undergraduate Students. if you wish to go further to get a higher grade, you can try and implement some of the following:

- Implement Stochastic Gradient Descent in the training loop, so instead of processing the whole dataset in each training iteration, you process a random batch instead
- Make a plot of the decision boundary of the neural network before and after training.
- Implement the training using a Mean-Squared-Error loss function instead. Train one model with the MSE loss, and one with the BCE loss, and compare the accuracy on the validation and test sets over the duration of the training.

In [ ]:
```
# Further work here
```