In [ ]:
```
# If you are using Noteable, run this:
%pip install librosa
# Otherwise, if not using Noteable, ensure you have librosa installed in
```

In [ ]:
```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import librosa
from glob import glob
from torch.utils.data import Dataset, DataLoader
import sklearn.metrics
```

Python Libraries

You do not need to use any Python libraries in addition to those listed above. Do not import any other libraries. If you think I have missed out a library that you are meant to use during this assignment, please contact me.

Audio Machine Learning - Summative Assessment 1 - Machine Learning Challenge - Part 2

For the second part of the machine learning challenge, your task is to implement and train a spoken digit classifier using PyTorch

The model is intended to classify spoken digits, specifically the numbers 0 to 9 spoken in English.

There are sections in the Notebook which are left for you to complete, which will be assessed. This will be marked as below:

This will be marked as below:

Assessed Section

In [ ]:
```
# Any code cells in the 'Assessed Section' are part of the assessment.
# 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

Downloading the Dataset

The dataset for this assignment is available to download from GitHub at the following URL:

https://github.com/Alec-Wright/Digits

Downloading in Noteable

You can download this directly into Noteable by:

- Opening Noteable
- Clicking the 'Git' menu item at the top of the window
- Selecting 'Clone a Repository'
- Entering the url 'https://github.com/Alec-Wright/Digits'
- Clicking Clone

More information available here if you have trouble.

https://noteable.edina.ac.uk/user-guide/#up_4

Downloading to Computer

- Go to https://github.com/Alec-Wright/Digits
- Click the green '<> code' button
- Click 'Download ZIP'
- Unzip the downloaded .zip file

This should download a folder called 'Digits', within which are various subfolers containing '.wav' files of human speech.

Please contact me if you are unable to download the dataset.

Task 1 - Dataset Exploration

In the folder 'Digits', there is a folder called 'Data' that holds recordings of spoken digits, from various speakers, speaking the digits 0 to 9. This is from the 'Audio MNIST' dataset.

Each audio file in the dataset is a '.wav' file that contains a recording of one speaker saying one of the digits.

The folders are organised by speaker, so 'Data/01/' contains digits spoken by speaker '01' and 'Data/02/' contains digits spoken by speaker '02', and so on.

Assessed Section - 2.1

Your task is to:

- Determine how many audio files are in the dataset held in 'Digits/Data/' (including all subfolders). Save this to a variable called 'dataset_size'.
- Use Librosa to load each file and find its length in samples. Determine the length of the longest audio file, and the shortest audio file, in samples. Save these to variables called 'longest_length' and 'shortest_length'.
- For each of the 10 digits, count how many examples are contained in the dataset. Save this as a list to a variable called 'class_counts'

In [ ]:
```
# Your solution here
```

End of Assessed Section - 2.1

Task 2 - Zero Padding and Preprocessing

Each audio file is of a different length. We want them all to be the same length, for processing in batches with our neural network. We can do this by adding zeros to the end of shorter files, or by truncating longer files (cutting the end off).

We then want to create a time-frequency representation of the audio, which we will extract using Librosa.

Write a function, 'adjust_length', that adjusts the length of some input audio data. It should have the following arguments:

- audio_data: a tensor of shape $(t,)$, where t is the length of the audio in samples
- target_length: an Integer that describes the target length in samples of the audio

The function should:

- Either apply zero-padding or truncate the audio data such that it is of shape (target_length,)
- The function should then return the input audio, which is now of the shape (target_length,)

Write another function, 'preprocess', that normalises some input audio data, and then creates a time-frequency representation of it. It should take the following arguments:

- audio_data: a numpy array of shape $(t,)$, where t is the length of the audio in samples
- sample_rate: An integer that represents the sample rate of audio_data

The function should:

- Normalise the audio data. Multiply it by some scalar such that np.max(np.abs(audio_data)) = 1.
- Extract the mel spectrogram of the audio_data using Librosa. Use the librosa.feature.melspectrogram() function, ensuring to provide the correct sample rate to the function. You can use the default parameters for melspectrogram().
- Return the extracted mel-spectrogram

Assessed Section - 2.2

Your task is to:

- Implement the two functions described above, in the templates below:

In [ ]:
```
def adjust_length(audio_data, target_length):
    ## Implement the function here
    pass
    return 0 # Replace 0 with whatever you want this function to return

def preprocess(audio_data, sample_rate):
    ## Implement the function here
    pass
    return 0 # Replace 0 with whatever you want this function to return
```

End of Assessed Section - 2.2

Task 3 - Creating the Datasets

We will create a dataset using the recordings from speaker '01'. The audio from speaker 01 is contained in the folder 'Data/01/'

Each audio file is named in the following format:

'digit_speaker_i.wav'

What appears before the first underscore is the digit being spoken. What appears after the first underscore is the speaker identity. What appears after the second underscore is the index.

So:

```
'0_01_0.wav'
```

is the digit '0'
spoken by speaker '01'
and it is the first recording of speaker '01' saying '0' in the dataset.

```
'0_01_1.wav'
```

is the digit '0'
spoken by speaker '01'
and it is the SECOND recording of speaker '01' saying '0' in the dataset.

and so on.

The dataset will be split into three subsets. For each digit, the first 40 examples of that digit should be placed in the *training* dataset. The next 5 examples should be placed in the *validation* dataset, and the final 5 should be placed in the *test* dataset.

Assessed Section - 2.3

Your task is to:

- Create the three data subsets as described above
- Create a list of the filenames of the audio to be contained in each subset
- For each file:
  - Load the audio using Librosa, resampling to a sample_rate of 22050
  - Use the adjust_length function you defined earlier to adjust the length of the audio data to exactly 1-second
  - Use the preprocess function you defined earlier to extract the time-frequency representation
  - Determine the label of the file from the filename (using the filename.split('_') method might be useful here)
- For each subset:

- Extract the time-frequency representation of all the audio files contained in that subset, as well as the labels
- Create a PyTorch tensor that holds all the time-frequency representations of the data in that subset. Ensure it is of datatype 'float32'. It should be a tensor of shape (N, C, T), where N is the number of datapoints in that subset, C is the number of frequency bins in the mel-spectrogram, and T is the length of the mel-spectrogram in frames.
- Create a PyTorch tensor that holds the labels corresponding to each of the N examples held in the tensor of features. It should be of shape (N,)

Save the features to the variables: 'train_features', 'val_features' and 'test_features'

Save the labels to the variables: 'train_labels', 'val_labels', 'test_labels'

In [ ]:
```
# Your solution here
```

End of Assessed Section - 2.3

Task 4 - Datasets and DataLoaders

We will now create the Dataset class, which will hold our extracted features and labels

Assessed Section - 2.4

Create an AudioDataSet class, using the template below:

- It should take PyTorch tenors holding features and corresponding labels of a dataset as its constructor arguments
- It should save these as attributes in the constructor
- The __getitem__ method should take the integer 'i' as its argument, and return both the features and corresponding label for the $i-th$ item from the dataset
- The __len__ method should return an integer, that is equal to the number of data points held in the dataset

In [ ]:
```
class AudioDataSet(Dataset):                # Don't change this line
    def __init__(self, features, labels): # Don't change this line
        pass # Define the Class constructor here

    def __getitem__(self, i):             # Don't change this line
        pass # Define the __getitem__ method here
```

```
    def __len__(self):                    # Don't change this line
        pass # Define the __len__ method here
```

End of Assessed Section - 2.4

Task 5 - Neural Network Creation

We want to make a multi-class classifier, that takes in the time-frequency representation of the audio as input, and predicts the digit that is being spoken.

We will do this using a convolutional neural network.

First, define a Convolutional Block Class. The Convolutional block will consist of a 1D-Convolutional layer, followed by a ReLU activation function. The convolutional block should process inputs through these two layers, and then return the output. The number of input channels, output channels, and kernel size should be determined by arguments provided to the class constructor. The convolutional block should **not** apply any zero-padding.

Then, define a Convolutional Neural Network (CNN) Class. It should consist of 8 of the Convolutional blocks defined previously. It should process inputs sequentially through each of the eight blocks.

After the eighth block, the network should apply global average pooling, which should apply pooling in the time-dimension only, reducing the output to a single value in the time-dimension, but not applying averaging over the channel or batch dimensions. Finally, a linear layer should be applied, converting the final output to a tensor with the appropriate shape for this multi-class classification problem.

Assessed Section - 2.5

Your task it to:

- Create the ConvBlock class described above, using the template provided
- Create the ConvNet class described above, using the template provided
- Create an instance of your ConvNet class, with kernel size of 5. The number of input channels for the first convolutional block should be chosen based on the dimensionality of the time-frequency representation of your data. Otherwise, the number of channels for convolutional layers should be 16.
- Process some data through the neural network class, and compare the shapes of the input and output Tensors. You may use data extracted earlier, or you may create random data using torch.randn(N,C,T), where N is the batch size, C is the

channel dimensions and T is the number of frames in the time-frequency representation.

In [ ]:
```
class ConvBlock(torch.nn.Module):                                # Don't change th
    def __init__(self, your_arguments):                          # Add arguments t
        super(ConvBlock, self).__init__()                        # Don't change th
        pass # Define your class constructor here

    def forward(self, x):                                        # Don't change th
        # Define forward method here
        return block_output
```

In [ ]:
```
class ConvNet(torch.nn.Module):                                  # Don't change th
    def __init__(self):                                          # Add arguments t
        super(ConvNet, self).__init__()                          # Don't change th
        pass # Define your class constructor here

    def forward(self, x):                                        # Don't change th
        # Define forward method here
        return network_output
```

In [ ]:
```
# Create an instance of your network, as process some input with it as de
```

End of Assessed Section - 2.5

Task 6 - Neural Network Training

Now you will train your Neural Network Digit Classifier.

The Training Loop. One iteration of the Training Loop should carry out Stochastic Gradient Descent over the full training dataset, in random batches of batch size 50. Iterating over the complete training dataset once is commonly known as 'one training epoch'. You may use the PyTorch DataLoader class when creating the training loop. Each training epoch must use every item in the training dataset exactly once. Use the Adam optimiser, with default settings, available as 'torch.optim.Adam()'. Ensure you pass the model parameters to the optimiser. Use the Cross-Entropy Loss function. You may use the PyTorch cross-entropy loss function class. After each training epoch, save the average loss over the full training subset to a list of losses.

The Validation Loop. The Validation Loop should calculate the average loss and accuracy of the model over the whole Validation subset. The validation loss should be saved to a list of validation losses and the validation accuracy should be saved to a list of validation accuracies.

The Test Loop. The Test Loop should calculate the average loss and accuracy of the model over the whole test subset. The test loop should also calculate the accuracy

for each class, and generate a confusion matrix. You may you sklearn.metrics to create the confusion matrix.

Assessed Section - 2.6

Your task it to:

- Create the Training, Validation and Test loops described above.
- Train your neural network for 100 epochs, calculating the validation loss and accuracy every 5th epoch.
- After training is complete, run the test loop. Display the confusion matrix, and save the resulting figure to a file called 'DigitClassifierConfusion.png'. Include this file with your submission.

In [ ]:

End of Assessed Section - 2.6

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:

- Does the trained classifier generalise to other speakers? Test this using other data from the dataset provided.
- Try increasing the diversity of the data used during training, by including more speakers. Train a model using this dataset, and test if this model generalises better to speakers that weren't included in the training set.
- Adjust your ConvNet class, so it can receive arguments determining the number of blocks and the activation function used.
- There are many choices you could make to change the neural network architecture, or the training process generally. For example, adjusting the number of blocks, changing the time-frequency representation used or adjusting the number of channels in the convolutional layers. Pick one or two of these and run experiments to compare how making changes to these influences the performance of the trained neural network model. Write in a markdown cell a summary of your findings and any conclusions you can draw about the adjustments you made.

In [ ]: