Semester project
„Coin Detection using PyTorch“

First name	Last name	Immatriculation number	Studies (NES/IST/ET)
Rohan Krishna	Vijayaraghavan	5040414	NES
Nitin Krishna	Venkatesan	5039262	NES
Keerthan Kopparam	Radhakrishna	5039224	NES
Prathamesh Vilas	Khairnar	5032738	NES

1. Introduction

An Artificial Neural Network is a computational learning system that uses a network of functions to understand and translate input data of one form into the desired output, usually in another form. Artificial neurons emulate the behavior of neurons in the brain. [1] They try to recognize the patterns, relationships, and information from the data provided. Humans cannot decipher ultrasound. The use of neural networks for detecting euro coins using their ultrasound time series data is a promising area of research. Recent studies have shown that neural networks can achieve high accuracy in detecting different types of euro coins even in challenging scenarios. The input data to a neural network can be roughly classified as either time-series data or spatial data. For the classification of coins, a dataset composed of time-series data has been provided. Time series data is a collection of observations obtained through repeated measurements over time. Time series metrics refer to a piece of data that is tracked at an increment in time [2].

2. State of art and research question

The project has been provided with ultrasound time series data for different euro coins, which have been sampled at a rate of 204kHz/s. The data comprises of CSV measurements for 7 out of 8 euro coin classes, namely 1ct, 2ct, 5ct, 20ct, 50ct, 1€, and 2€, while the data for 10-cent coins has been excluded to simulate the non-existence of such coins. The data encompasses a total of 7 distinct coin classes and 796024 discrete data points, providing a comprehensive dataset for training and testing an AI-based system for detecting euro coins based on ultrasound samples.

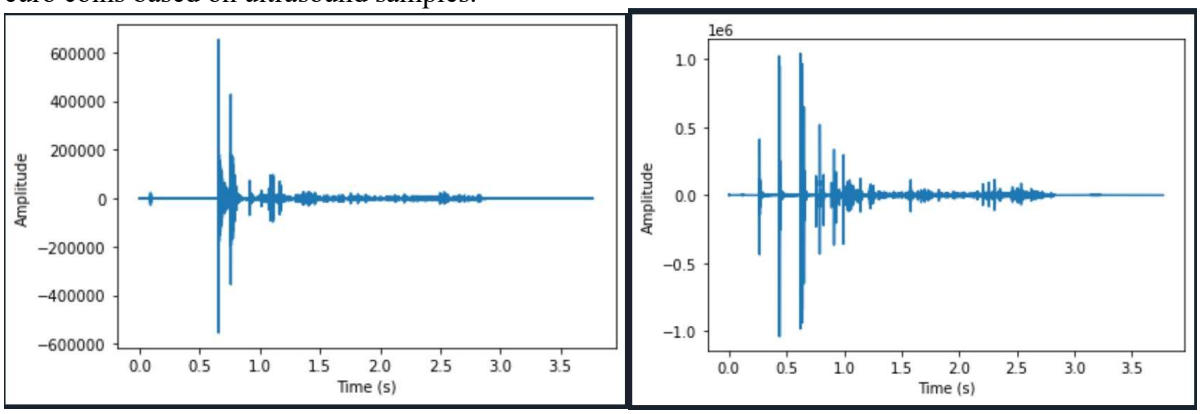

Figure 1 Input ultrasound waveform for 20ct coin(left) and 2euro coint(right)

“Is it possible for a neural network to accurately classify ultrasound audio, when humans cannot do so?”

3. Main chapters

3.1. Problem description

The accurate and efficient detection of euro coins falling onto a desk is a task that has practical applications in various domains. However, the manual detection of coins is time-consuming and error-prone. Hence an AI-based system that automatically detects euro coins falling onto a desk using ultrasound samples is developed. Our system will be trained using a handcrafted dataset of ultrasound samples of euro coins falling onto a desk, enabling the model to identify and differentiate between different types of coins accurately. Through this project, we aim to learn neural networks in designing and implementing AI-based systems practically.

3.2. Methods to solve the challenge.

The given data cannot be directly fed to the neural networks; therefore, pre-processing is required before feeding the data to the model. It also improves the quality and reliability of the data, thus enhancing the accuracy and efficiency of the model.

3.2.1. Data Imbalance Problem

In this case, the data is not undersampled or oversampled. The main reason being the delta between the class with the highest number of instances and the class with the lowest number of instances is less than 10%. Since the difference is insignificant, it is unnecessary to undertake any measures to address the data imbalance.

3.2.2 Create a value/target pair for the data sample

In this case, the data is not undersampled or oversampled. The main reason being the delta between the class with the highest number of instances and the class with the lowest number of instances is less than 10%. Since the difference is insignificant, it is unnecessary to undertake any measures to address the data imbalance.

Value/target pairs are used in neural networks to enable the model to learn from labeled data through supervised learning, where the target values are used to train the model to predict accurate outputs for a given input.

The main pre-processing class is the **AudioDataset(Dataset)**, which has three methods. Init(), the first method, sorts out all the audio files in a specified order. The second method, **getitem(index),** loads the data from these files to a numpy array and does zero-padding for uneven time samples. The array is then reshaped and normalized, after which it is labeled using one-hot encoding. Finally, the method converts the numpy array to a tensor to create the target/value pair, and returns a tuple of <data, label>.

3.2.3 Normalize data set into Training, Validation and Test

Normalization of data into training, validation, and test sets is crucial in neural networks to ensure that the model can learn and generalize from the data effectively. By separating the data into these sets, the model can be trained on one set, validated on another set, and tested on a third set. This process helps to prevent overfitting and ensures that the model can accurately predict outputs for new, unseen data.

To randomly distribute the data into training, validation, and test sets with 75%, 10%, and 15% equity, we have developed a function called get_split_data(). This function returns the data into these sets and ensures that the model can learn and generalize from the data effectively

Multi-Layer Perceptron

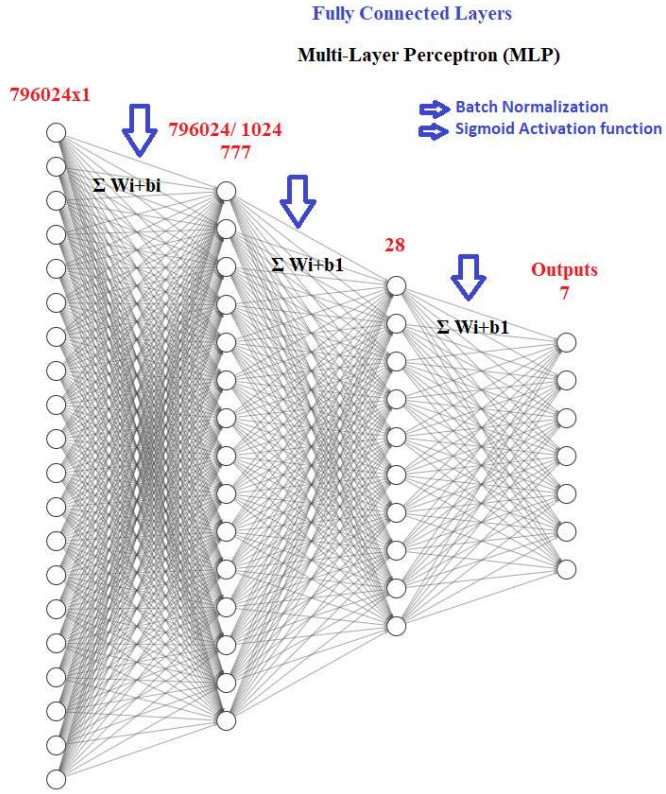

Fig2. Multi Layer Perceptron

In this project MLP is also used alongside CNN to perform classification of coins. MLP stands for Multi-layer perceptron. MLP consists of multiple layers of interconnected neurons or nodes that processes and transmits the information to solve a specific task. In this case we have 4 layers. Each neuron receives input, performs a dot product then activation function and later processed output is passed to the next layer. Batch normalization is also used to normalise the activations of each layer's neuron. It also reduces the chances of overfitting making the model stable or consistent. Overfitting occurs when the model is too well trained and as result it performs poorly on new or unsigned data. For activation, sigmoid activation function is used. it maps the input to the output in the range of 0 to 1, it allows the model to learn more complex inputs and outputs so this way outputs output neurons are able to classify the data based on the features extracted from previous data. In figure – it can be seen the number of neurons being drastically reduced due to batch normalization. At the output layer there are only 7 neurons containing specific features of coins.

Convolutional Neural Networks:

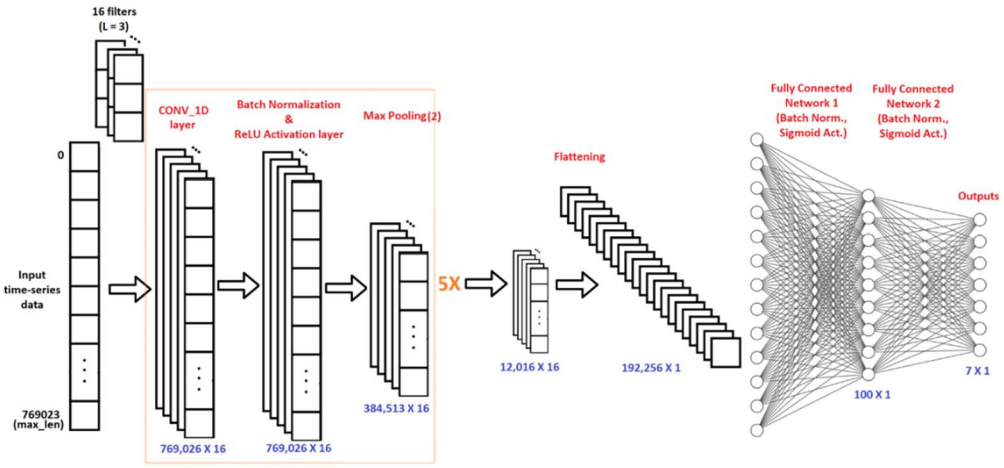

Figure 3. CNN Model for the classification of coins

One of the tasks assigned was to also implement the classification of coins using a Convolutional Neural Network model. For the CNN model, six CONV-1D networks and two fully connected networks are used. For the CONV-1D layer, 16 filters with a kernel size of 3 are being used. Each of these filters extracts a certain feature or aspect of the input time-series data. With a stride of 1, the filters convolve the input data. The convoluted data are then subjected to batch normalization, ReLU activation function, and later pooled by a factor of 2. These stages are repeated again for another 5 times, when the dimensions of the data become 12,016x16. These values are flattened to a 1D array that can be passed to the fully connected networks. Each of the 12,016x16 values is fed into a neuron in the input layer of the fully connected network. The size of the input layer has been drastically reduced as a result of Max Pooling. In comparison with the MLP model implemented earlier, the memory size of the CNN model has been reduced significantly. Within the fully connected network, batch normalization and Sigmoid activation function are used as earlier. At the output, there are seven neurons, each indicating a certain class. The Cross-Entropy function has been used to compute the loss between the predicted and expected classes. Adam optimizer class function has been used for the optimization of weights and biases.

3.3. Results and discussion

Now we train both the models with the same hyperparameters, as shown in the table below.

Hyperparameter	Value
Batch size	64
Number of epochs	8
Learning rate	0.002

While no regularization method is implemented in the training loop, the number of epochs were manually adjusted based on the accuracies in every epoch. The results of training the model can be seen in the below figure.

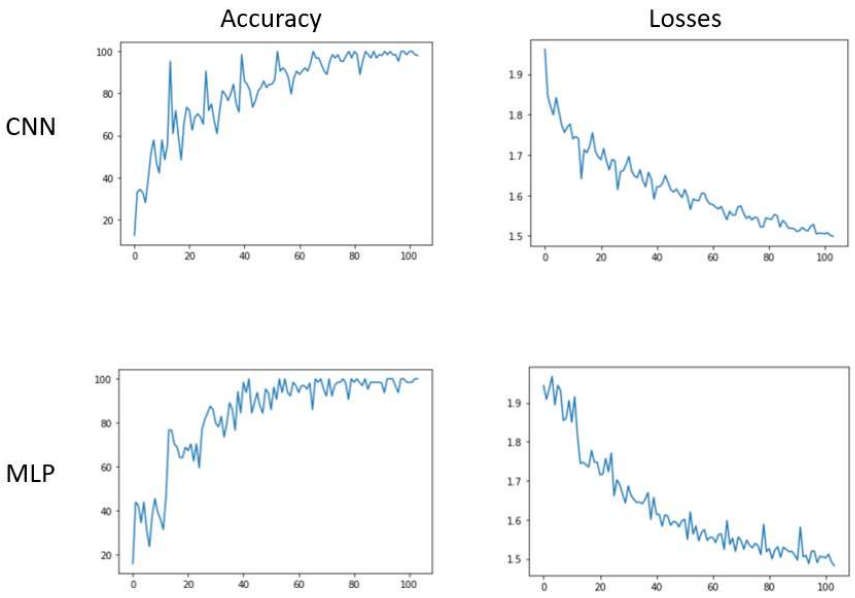

Fig 4. Plot of training accuracies and losses for CNN and MLP

It is to be noted here that the accuracies and losses are plotted with respect to iteration, i.e., value per batch per epoch. This gives us a good idea not only of how the values change every epoch, but also how well the model is trained to classify every individual batch. The test accuracies and the model sizes of the two models are shown below.

	MLP	CNN
Model size	2.2 GB	73.3 MB
Test accuracy	44.848%	74.545%

As is seen, the CNN performs much better than the MLP, while also having a much smaller model size, proving that it is optimal for this dataset.

We now perform t-SNE on the results of the last convolutional layer of the CNN. t-SNE [4] is a tool to visualize high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. The implementation of t-SNE was largely derived from the source in [5].

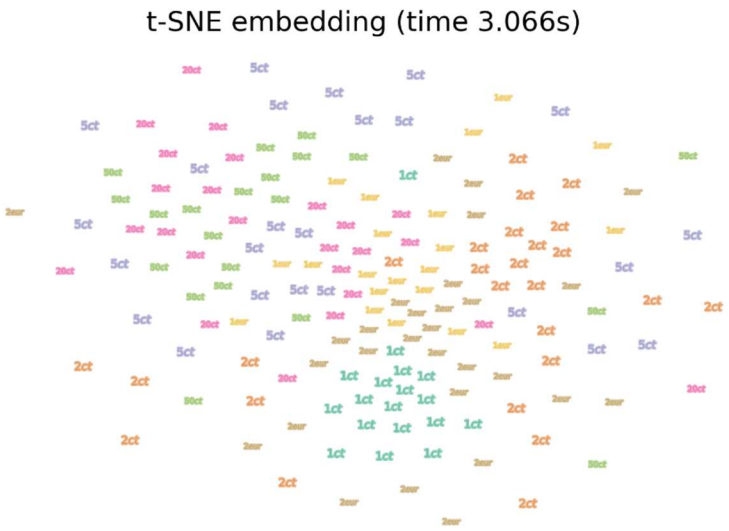

Fig 5. t-SNE performed on last convolution layer output in CNN

4. Summary

In this project, we aim to detect different euro coins based on their ultrasound waveform produced when dropping them on a table. We design two different models, a Multi-layer perceptron network and a convolutional neural network, and compare the performance of both for this dataset. We try to obtain optimal performance purely through the functioning of the network, without performing any signal processing such as down-sampling on the initial data. The results show us that the MLP trains faster on the data but performs poorly for unseen data, suggesting that the model memorizes the sequences. The CNN however takes longer to train, but performs much better for unseen data, showing that it learns to extract necessary features from the data for classification. Performing t-SNE on the CNN output also shows that there is a decent amount of clustering of data corresponding to the classes, therefore suggesting that the CNN is the optimal model between the two.

5. Appendix with implementation code

Pre-processing functions

```
def sort_files():

  file_path = "/content/gdrive/My Drive/data_raw/"#go to path of files

  '''
```

```
    can you check how to implement the next 2 lines here because we arent using the "os" library
    '''
    file_names = os.listdir(file_path)#get file names from path
    file_names.sort()#sort names

    file_list = []
    for name in file_names:
        file_list.append(file_path+name)#append file path to file names



    return file_list#return list of files
```

dataset Class

```
class AudioDataset(Dataset):#dataset class to hold data for loading into model
    def     init    (self, transform=None):

        self.scaler = StandardScaler()#normalizing function
        self.max  len = 769024
        self.paths = sort_files()#store paths of all data instead of storing actual data
        print(self.paths[1])

    def __getitem__(self, index):#output data & label given input index

        d = np.loadtxt(self.paths[index],delimiter=',', dtype=float)#get data from files to numpy array
        d = np.pad(d, (0, self.max_len - len(d)), 'constant', constant_values=0)#zero padding for uneven time samples

        d = d.reshape(-1,1)
        d = d/STD_DEV #normalize, div by stddev is enough



        if self.paths[index].startswith('/content/gdrive/My Drive/data_raw/200'):#add labels
            y = np.array([0, 0, 0 ,0, 0, 0, 1])
        elif self.paths[index].startswith('/content/gdrive/My Drive/data_raw/100'):
            y = np.array([0, 0, 0, 0, 0 ,1, 0])
        elif self.paths[index].startswith('/content/gdrive/My Drive/data_raw/50'):
            y = np.array([0, 0, 0, 0, 1, 0 ,0])
        elif self.paths[index].startswith('/content/gdrive/My Drive/data_raw/20'):
            y = np.array([0 ,0 ,0 ,1, 0 ,0 ,0])
        elif self.paths[index].startswith('/content/gdrive/My Drive/data_raw/5'):
            y = np.array([0, 0, 1, 0, 0, 0, 0])
        elif self.paths[index].startswith('/content/gdrive/My Drive/data_raw/2'):
            y = np.array([0, 1 ,0 ,0, 0, 0 ,0])
        elif self.paths[index].startswith('/content/gdrive/My Drive/data_raw/1'):
            y = np.array([1, 0, 0,0 ,0 ,0, 0])



        return torch.tensor(d).float(), torch.LongTensor(y).float()

    def __len__(self):#output lengths of dataset
```

```
        return len(self.paths)
```

Function to split dataset

```
def get  split  data():

    full  data = AudioDataset()#create dataset object


    proportions = [.75, .10, .15]
    lengths = [int(p * len(full  data)) for p in proportions]
    lengths[-1] = len(full  data) - sum(lengths[:-1])
    train_data,val_data, test_data = torch.utils.data.random_split(full_data, lengths)#split into train,val, test dataset

    return train_data, test_data, val_data
```

CNN model

```
class CNNModel(nn.Module):# CNN 6x (1x3 conv -> batch norm -> ReLU -> maxpool 2) -> 100 x 7 FC layer

    def __init__(self,input_size, filter_size, kernel_size, num_channels, num_classes):#initialize layers of model
        super(CNNModel, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv1d(num_channels ,filter_size, kernel_size=kernel_size, stride=1, padding=1),
            nn.BatchNorm1d(filter_size),
            nn.ReLU(),
            nn.MaxPool1d(kernel  size=2, stride=2))

        self.layer2 = nn.Sequential(
            nn.Conv1d(filter  size ,filter  size, kernel  size=kernel  size, stride=1, padding=1),
            nn.BatchNorm1d(filter  size),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2))

        self.layer3 = nn.Sequential(
            nn.Conv1d(filter_size ,filter_size, kernel_size=kernel_size, stride=1, padding=1),
            nn.BatchNorm1d(filter  size),
            nn.ReLU(),
            nn.MaxPool1d(kernel  size=2, stride=2))

        self.layer4 = nn.Sequential(
            nn.Conv1d(filter  size ,filter  size, kernel  size=kernel  size, stride=1, padding=1),
            nn.BatchNorm1d(filter  size),
            nn.ReLU(),
            nn.MaxPool1d(kernel  size=2, stride=2))

        self.layer5 = nn.Sequential(
            nn.Conv1d(filter_size ,filter_size, kernel_size=kernel_size, stride=1, padding=1),
            nn.BatchNorm1d(filter  size),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2))
```

```
        self.layer6 = nn.Sequential(
            nn.Conv1d(filter_size ,filter_size, kernel_size=kernel_size, stride=1, padding=1),
            nn.BatchNorm1d(filter_size),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2))

        self.fc1 = nn.Sequential(
            nn.Linear(int(input_size/(2**6))*filter_size, 100, bias=True),
            nn.BatchNorm1d(100),
            nn.Sigmoid())
        self.fc2 = nn.Sequential(
            nn.Linear(100, num_classes, bias=True),
            nn.BatchNorm1d(num_classes),
            nn.Sigmoid())


    def forward(self, x):#forward path of model
        x = x.transpose(1, 2).contiguous()#reshape data for compatibility with torch conv layers
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x = self.layer6(x)
        x = x.view(x.size(0), -1)#flatten
        tsnedata.append(x)  #pass data for tsne later
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

MLP model

```
class MLPModel(nn.Module):# MLP network 769024 x 751 x 28 x 7
    def __init__(self,input_size, num_classes):#initialize layers of the model
        super(MLPModel, self).__init__()

        self.fc1 = nn.Sequential(
            nn.Linear(input_size, int(input_size/2**10), bias=True),
            nn.BatchNorm1d(int(input_size/2**10)),
            nn.Sigmoid())
        self.fc3 = nn.Sequential(
            nn.Linear(int(input_size/2**10), int(num_classes*4), bias=True),
            nn.BatchNorm1d(int(num_classes*4)),
            nn.Sigmoid())
        self.fc4 = nn.Sequential(
            nn.Linear(int(num_classes*4), num_classes, bias=True),
            nn.BatchNorm1d(num_classes),
            nn.Sigmoid())
```

```
def forward(self, x):#forward path for model
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.fc3(x)
    x = self.fc4(x)
    return x
```

6. References

[1] Bayat, F. Merrikh, et al. “Implementation of Multilayer Perceptron Network with Highly Uniform Passive Memristive Crossbar Circuits.” Nature Communications, vol. 9, no. 1, 1, June 2018, p. 2331. www.nature.com, https://doi.org/10.1038/s41467-018-04482-4

[2] https://www.analyticsvidhya.com/blog/2021/05/beginners-guide-to-artificial-neural-network/

[3] https://www.influxdata.com/what-is-time-series-data/

[4] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

[5] https://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html#sphx-glr-auto-examples-manifold-plot-lle-digits-py