EE5178 - Modern Computer Vision
Programming Assignment - 1
Assignment Report

Bersilin C - CS20B013

March 12, 2024

OBJECTIVE

Implement and train models to classify images from the **CIFAR-10** dataset into 10 classes. Use **PyTorch** for the implementation. Different models include Multi-layer Feed-forward Neural Network (MLP) and Convoluted Neural Network (CNN). The **MLP** model acts as a baseline for the **CNN** model. The ten classes in the CIFAR-10 datasets are airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Implement and train an Encoder-Decoder model for transliteration from English (Latin script) to any of the local Indian scripts.

1 DESCRIPTION OF THE DATASET

The CIFAR-10 dataset consists of 60,000 images split into *train* and *test*, each consisting of 50,000 and 10,000, respectively. Each class contains exactly 5,000 images in the training set.

Each image is of the shape $(3, 32, 32)$. The first index corresponds to a colour channel *(Red, Blue, Green)*, each with a resolution of 32 x 32. Some example images with their labels are as follows:

(a) Sample Images

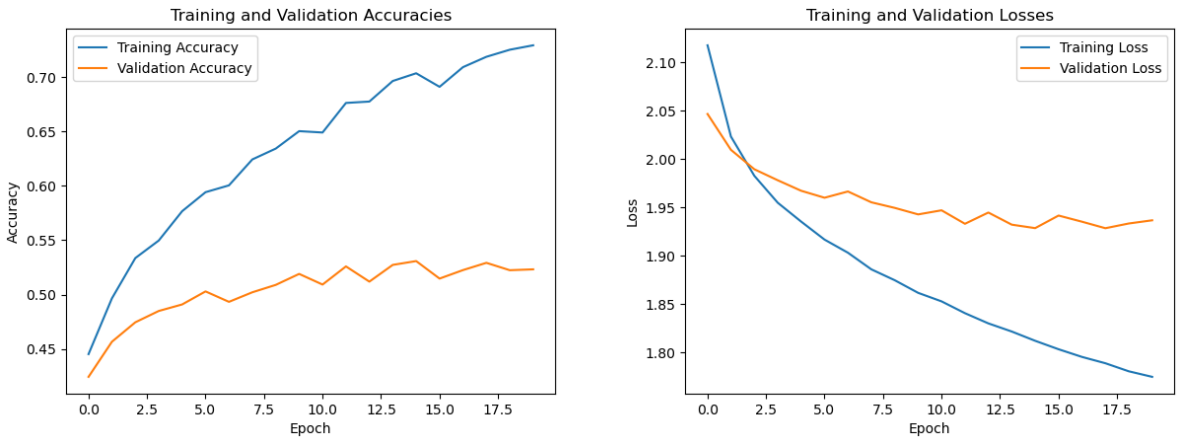
2 Multi-layer Feed-forward Neural Network

2.1 Architecture & Training

- The neural network consists of three hidden layers, each with a neuron count of 500, 250 and 100, respectively.
- The activation function used in the hidden layers is the Rectified Linear Unit *(ReLU)* function. The activation function used in the output layer is the linear function, and the softmax function is applied to obtain the probabilities for each of the ten classes.
- The loss function used to train the model is the Cross-Entropy loss function from the PyTorch library. The optimizer used to learn the parameters is Stochastic Gradient Descent *(SGD)*.
- The training of the model was done with various hyper-parameters.
  - Batch Size: [50, 80, 100, 120, 150, 200]
  - Learning Rate: [0.0005, 0.001, 0.003, 0.005, 0.007, 0.01, 0.03, 0.05]
  - Batch Normalization: [True, False]
  - Momentum (for SGD): [0.87, 0.9, 0.93, 0.99]
- The hyper-parameter tuning was done for 10 epochs for each model, and 100 models were trained. The results of the hyper-parameter tuning can be found here: MLP-Sweep
- The best set of hyper-parameters gave a training accuracy of **0.7293** and a validation accuracy of **0.5232** after 20 epochs.
- The test set consists of 10,000 images, and the accuracy in the test set is **0.5249**.
- The best set of hyper-parameters were: {'batch_norm': **True**, 'learning_rate': **0.007**, 'batch_size': **50**, 'momentum': **0.87**}.

2.2 Results

2.2.1 Accuracy and Error Plots

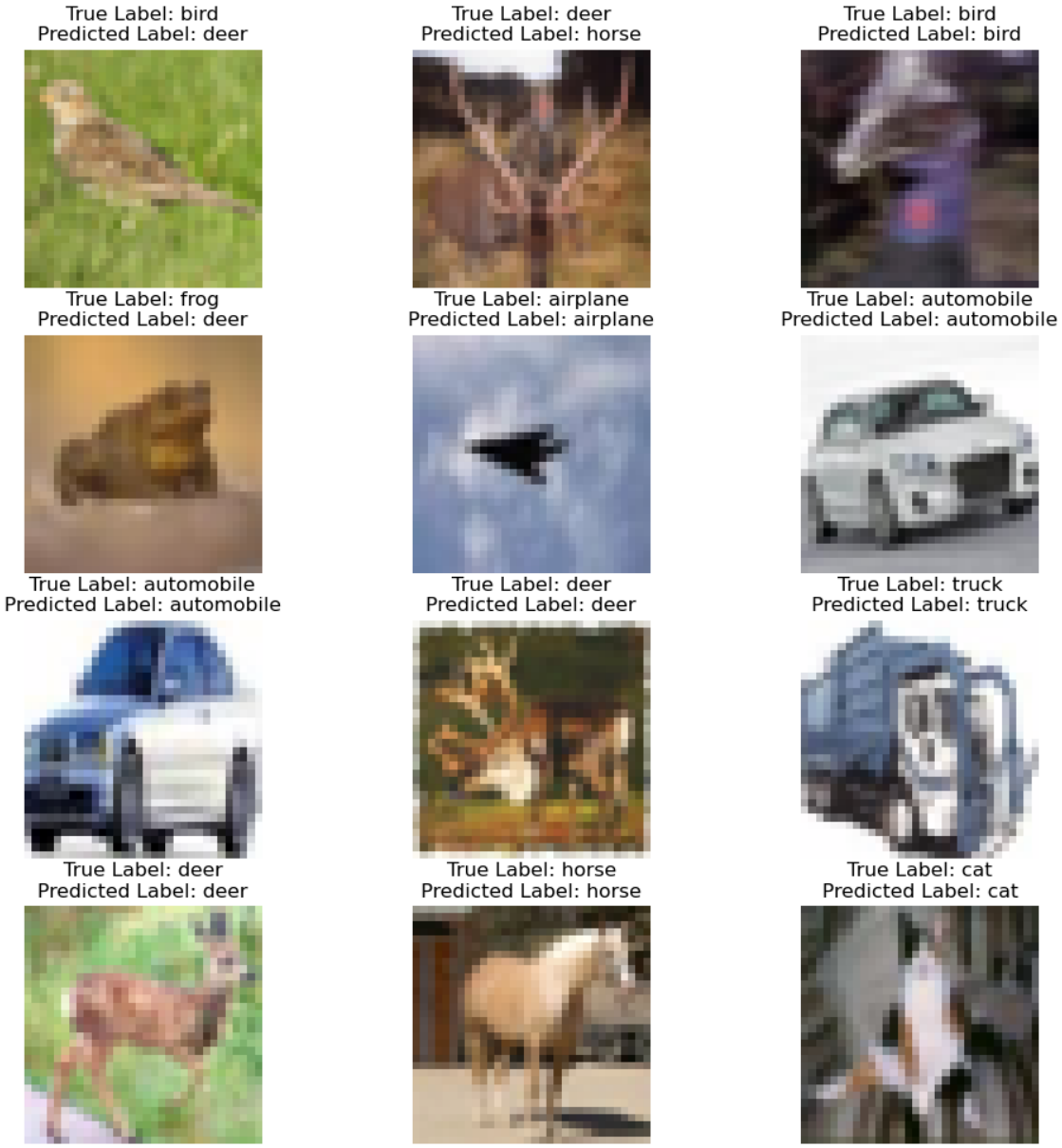


(a) Accuracy Plot

(b) Error Plot

Figure 2.1: vs Epochs Plots

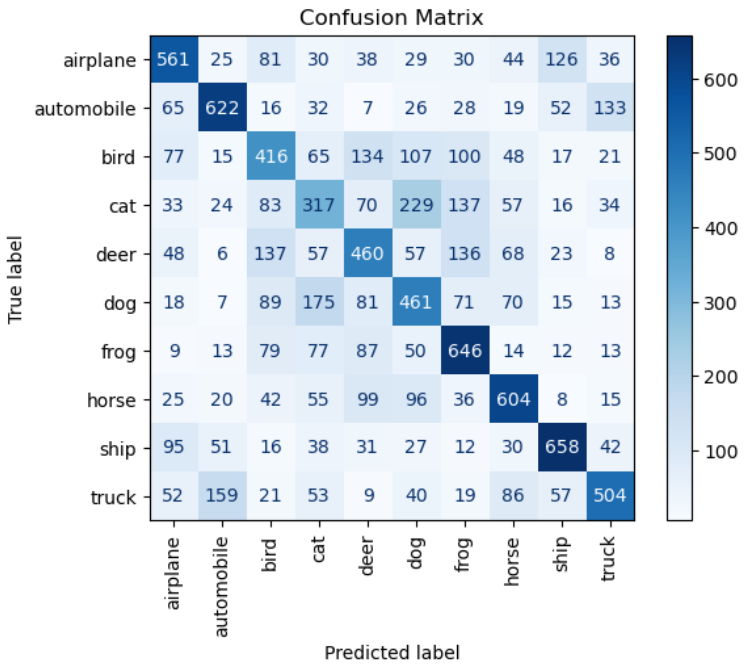
2.2.2 PERFORMANCE ON THE TEST SET



(a) Samples chosen from test set with their predicted label

Figure 2.2: Randomly chosen samples

2.2.3 Confusion Matrix



(a) Confusion Matrix

2.2.4 Effect of Batch Normalization

- For the same set of hyper-parameters like batch size, the time taken by the model with batch normalization enabled is higher than the model without batch normalization.
- The accuracy of the models is also higher on average. The best 20 models in the 100 models trained were with batch normalization enabled. This could be due to the added stability of the values during training.

2.3 The Model

```
MLP(
  (fc): Sequential(
    (0): Linear(in_features=3072, out_features=500, bias=True)
    (1): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=500, out_features=250, bias=True)
    (4): BatchNorm1d(250, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=250, out_features=100, bias=True)
    (7): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): Linear(in_features=100, out_features=10, bias=True)
  )
  (softmax): Softmax(dim=1)
)
```

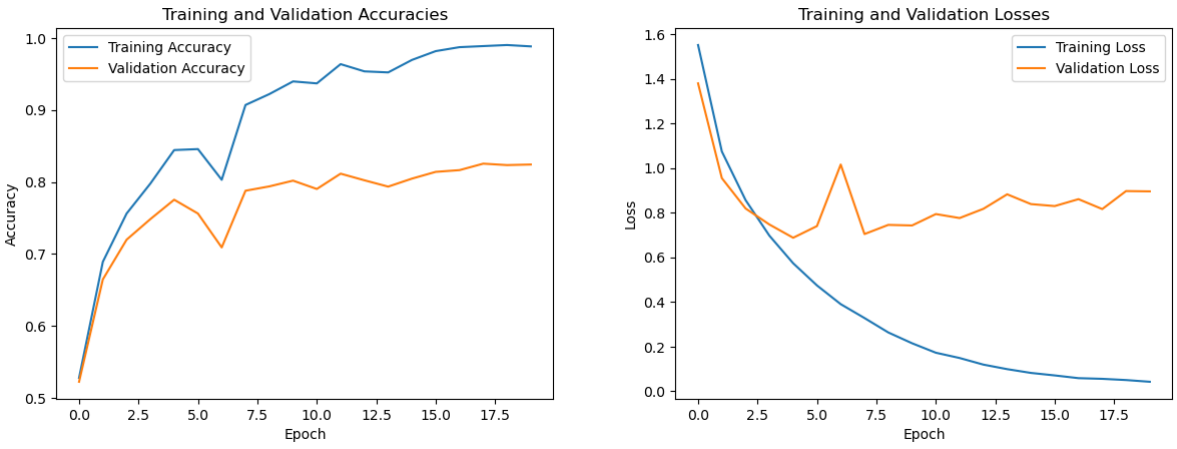
3 Convoluted Neural Network

3.1 Architecture and Training

- The CNN architecture for VGG-11 consists of repeating convolution with kernel (3, 3) and max pooling layers, reducing the image resolution by 2 in each axis but increasing the number of channels. The size of the image changes from (32, 32, 3) to (1, 1, 512) by the end of the convolution layers.
- The output from the convolution layers are fed into the fully connected layers of two hidden layers, each with a neuron count of 4096. The activation function for all the convolution and fully connected layers is ReLU. The loss function is Cross-Entropy and optimizer used in SGD.
- For hyper-parameter tuning, a total of 200 models were trained. The hyper-parameter tuning was done for 10 epochs for each model. The results are here: CNN-Sweep. The training was done with the following hyper-parameters:
  - Batch Size: [50, 80, 100, 120, 150, 200]
  - Learning Rate: [0.0005, 0.001, 0.003, 0.005, 0.007, 0.01, 0.03, 0.05]
  - Momentum (for SGD): [0.87, 0.9, 0.93, 0.99]
- The best set of hyper-parameters gave a training accuracy of **0.9883** and a validation accuracy of **0.8242** after 20 epochs. The accuracy of the test set is **0.815**.
- The best set of hyper-parameters were: {'learning_rate': **0.05**, 'batch_size': **100**, 'momentum': **0.87**}.
- **Bottleneck/Challenge**: Without the addition of batch normalization after the convolution layer, the model didn't seem to train even after 10 epochs. This could be due to an explosion of values after a few convolution layers. Introducing batch normalization avoided this, and the model started to learn. Normalizing the image using the mean and standard deviation of the pictures of the train set seems to improve the model's performance.

3.2 Results

3.2.1 Accuracy and Error Plots

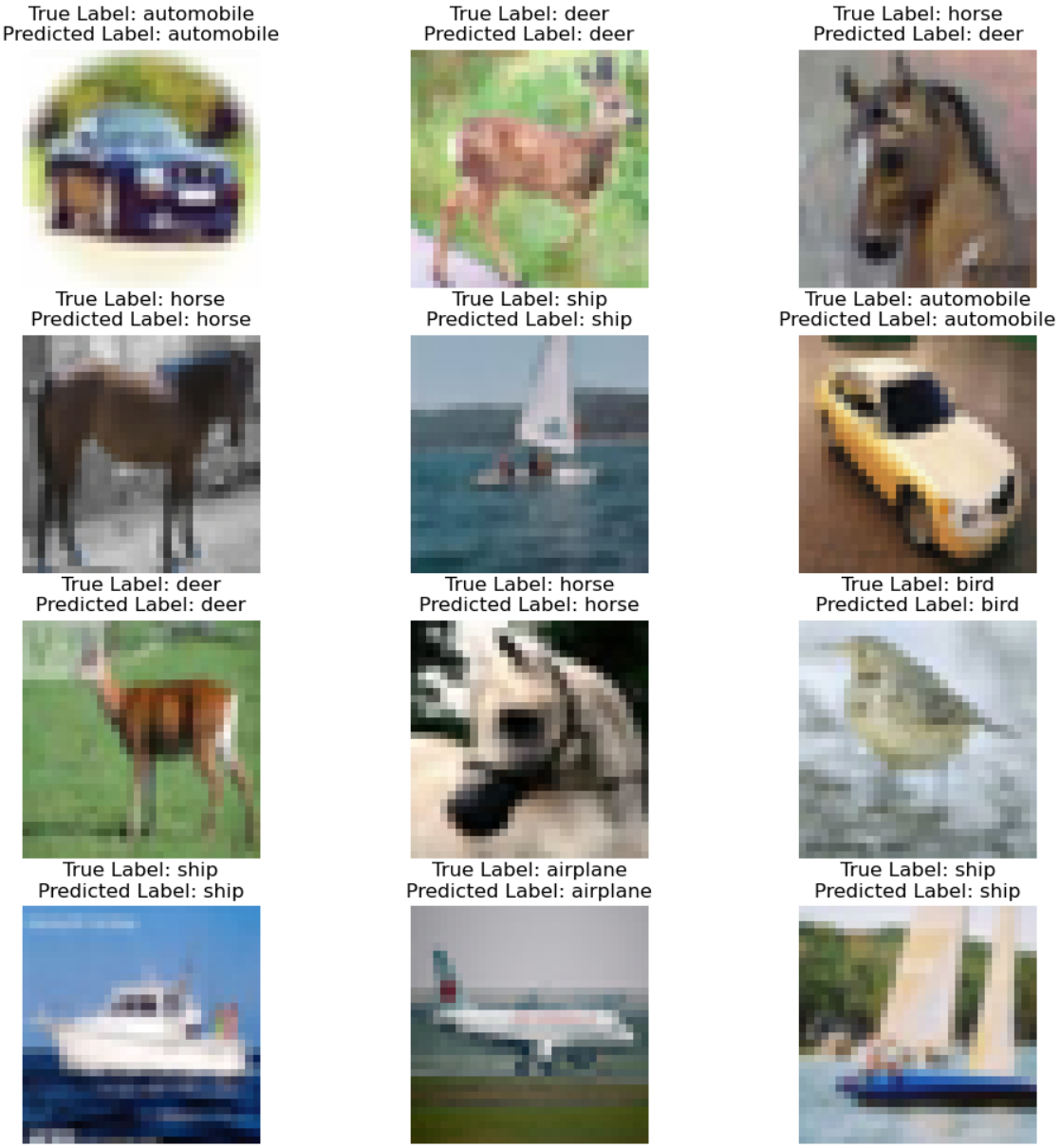


(a) Accuracy Plot

(b) Error Plot

Figure 3.1: vs Epochs Plots

3.2.2 PERFORMANCE ON THE TEST SET



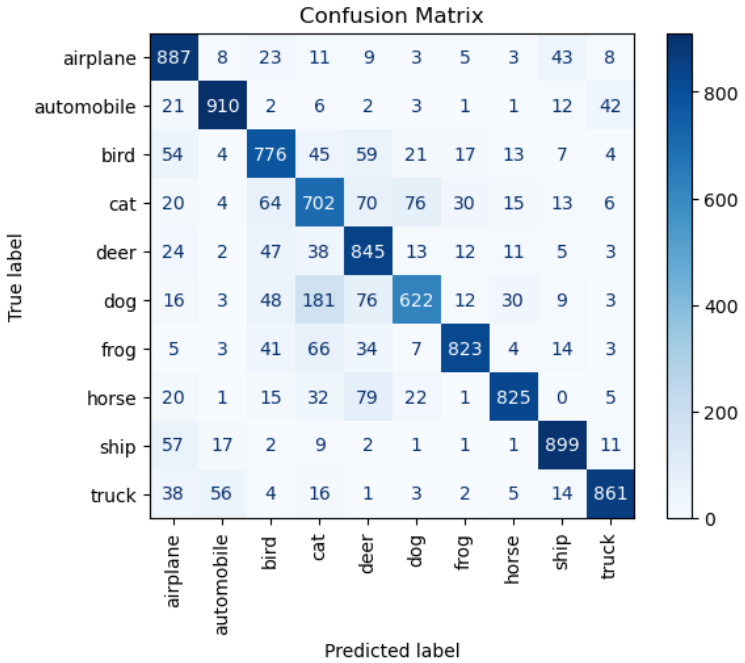
(a) Samples chosen from test set with their predicted label

Figure 3.2: Randomly chosen samples

3.2.3 CUSTOM IMAGES TEST

(a) These images were downloaded from the internet and compressed to (32, 32)

3.2.4 CONFUSION MATRIX



(b) Confusion Matrix

3.3 THE MODEL

```
VGG_11(
  (conv_layers): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False
    )
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False
    )
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU()
    (14): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=
    False)
    (15): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (17): ReLU()
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (20): ReLU()
    (21): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=
    False)
    (22): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (24): ReLU()
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (27): ReLU()
    (28): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=
    False)
  )
  (fcs): Sequential(
    (0): Linear(in_features=512, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

4 Recurrent Neural Networks

4.1 Normal Decoder Architecture

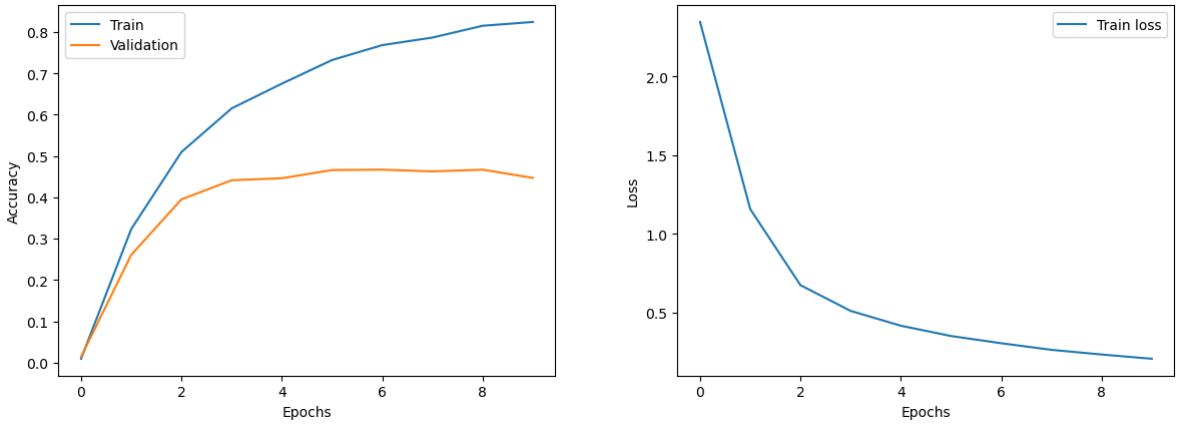
4.1.1 Architecture and Training

- In this section, the decoder takes the previous output of the decoder and the current character to decode the next output character. It doesn't take all the information from the input into account at every step.
- I have picked the Tamil language to train and test the model.
- The hyper-parameters used to train the model are:

```
params = {
    "embed_size": 8,
    "hidden_size": 256,
    "cell_type": "LSTM", #options: ['RNN', 'LSTM', 'GRU']
    "num_layers": 3,
    "dropout": 0.2,
    "learning_rate": 0.005,
    "optimizer": "SGD", #options: ['SGD', 'ADAM']
    "teacher_forcing_ratio": 0.5,
    "max_length": 50
}
```

- The loss used to train the model is the NLLLoss (Negative log-likelihood loss). The optimizers tested were ADAM and SGD, and the relative performance of SGD seems to be better than ADAM.
- The model was trained for 10 epochs. The loss seems to be decreasing even then, but it has plateaued mostly. The average training loss obtained is: **0.2041**
- The training accuracy and validation accuracy are **0.8242** and **0.4470** respectively.
- The accuracy in the test set is **0.4496**

4.1.2 Accuracy and Error Plots



(a) Accuracy Plot

(b) Error Plot

Figure 4.1: vs Epochs Plots

4.2 Attention-based Decoder Architecture

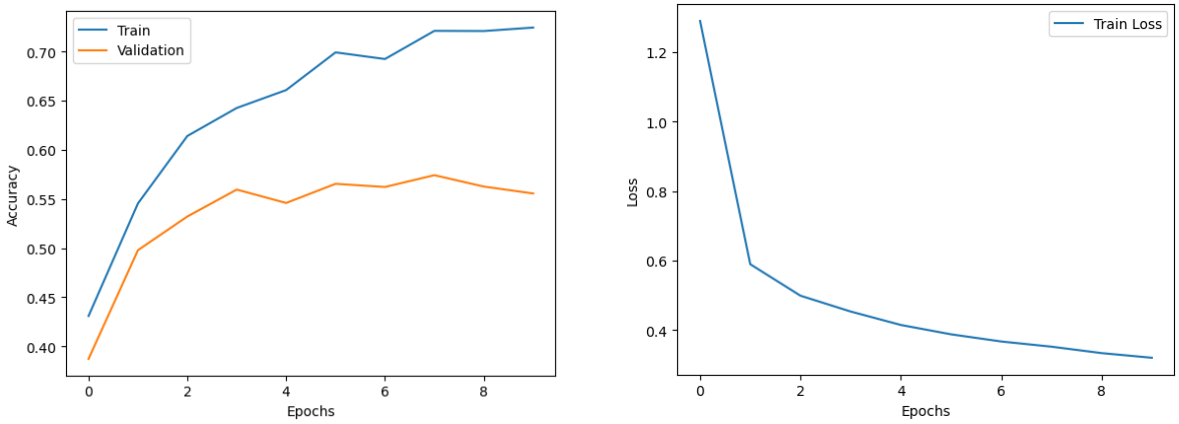
4.2.1 Architecture and Training

- In this section, the decoder takes the current character, the hidden state of the previous output and the encoded output of the encoder at every step of the transliteration.
- The hyper-parameters used to train the model are:

```
params = {
    "embed_size": 32,
    "hidden_size": 256,
    "cell_type": "RNN",
    "num_layers": 2,
    "dropout": 0,
    "learning_rate": 0.001,
    "optimizer": "SGD",
    "teacher_forcing_ratio": 0.5,
    "max_length": 50,
    "weight_decay": 0.001
}
```

- The accuracy was calculated based on string matching of the entire word. The loss is calculated letter-wise.
- The model was trained for 10 epochs. The loss seems to be decreasing even then, but it has plateaued mostly. The average training loss obtained is: **0.3210**
- The training accuracy and validation accuracy are **0.7244** and **0.5556** respectively.
- The accuracy in the test set is **0.5345**

4.2.2 Accuracy and Error Plots



(a) Accuracy Plot

(b) Error Plot

Figure 4.2: vs Epochs Plots

4.3 THE MODELS

4.3.1 THE ENCODER

```
class Encoder(nn.Module):
    def __init__(self,
                 in_sz: int,
                 embed_sz: int,
                 hidden_sz: int,
                 cell_type: str,
                 n_layers: int,
                 dropout: float):

        super(Encoder, self).__init__()
        self.hidden_sz = hidden_sz
        self.n_layers = n_layers
        self.dropout = dropout
        self.cell_type = cell_type
        self.embedding = nn.Embedding(in_sz, embed_sz)

        self.rnn = get_cell(cell_type)(input_size = embed_sz,
                                       hidden_size = hidden_sz,
                                       num_layers = n_layers,
                                       dropout = dropout)

    def forward(self, input, hidden, cell):
        embedded = self.embedding(input).view(1, 1, -1)

        if(self.cell_type == "LSTM"):
            output, (hidden, cell) = self.rnn(embedded, (hidden, cell))
        else:
            output, hidden = self.rnn(embedded, hidden)

        return output, hidden, cell

    def initHidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_sz, device=device)
```

4.3.2 THE NORMAL DECODER

```
class Decoder(nn.Module):
    def __init__(self,
                 out_sz: int,
                 embed_sz: int,
                 hidden_sz: int,
                 cell_type: str,
                 n_layers: int,
                 dropout: float,
                 device: str):

        super(Decoder, self).__init__()
        self.hidden_sz = hidden_sz
        self.n_layers = n_layers
        self.dropout = dropout
        self.cell_type = cell_type
        self.embedding = nn.Embedding(out_sz, embed_sz)
        self.device = device

        self.rnn = get_cell(cell_type)(input_size = embed_sz,
                                       hidden_size = hidden_sz,
                                       num_layers = n_layers,
                                       dropout = dropout)

        self.out = nn.Linear(hidden_sz, out_sz)
        self.softmax = nn.LogSoftmax(dim=1)

```

```
    def forward(self, input, hidden, cell):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)

        if(self.cell_type == "LSTM"):
            output, (hidden, cell) = self.rnn(output, (hidden, cell))
        else:
            output, hidden = self.rnn(output, hidden)

        output = self.softmax(self.out(output[0]))
        return output, hidden, cell

    def initHidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_sz, device=self.device)
```

4.3.3 THE ATTENTION DECODER

```
class AttentionDecoder(nn.Module):
    def __init__(self,
                 out_sz: int,
                 embed_sz: int,
                 hidden_sz: int,
                 cell_type: str,
                 n_layers: int,
                 dropout: float):

        super(AttentionDecoder, self).__init__()
        self.hidden_sz = hidden_sz
        self.n_layers = n_layers
        self.dropout = dropout
        self.cell_type = cell_type
        self.embedding = nn.Embedding(out_sz, embed_sz)

        self.attn = nn.Linear(hidden_sz + embed_sz, 50)
        self.attn_combine = nn.Linear(hidden_sz + embed_sz, hidden_sz)

        self.rnn = get_cell(cell_type)(input_size = hidden_sz,
                                       hidden_size = hidden_sz,
                                       num_layers = n_layers,
                                       dropout = dropout)

        self.out = nn.Linear(hidden_sz, out_sz)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden, cell, encoder_outputs):
        embedding = self.embedding(input).view(1, 1, -1)

        attn_weights = F.softmax(self.attn(torch.cat((embedding[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

        output = torch.cat((embedding[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        if(self.cell_type == "LSTM"):
            output, (hidden, cell) = self.rnn(output, (hidden, cell))
        else:
            output, hidden = self.rnn(output, hidden)

        output = self.softmax(self.out(output[0]))
        return output, hidden, cell, attn_weights

    def initHidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_sz, device=device)
```