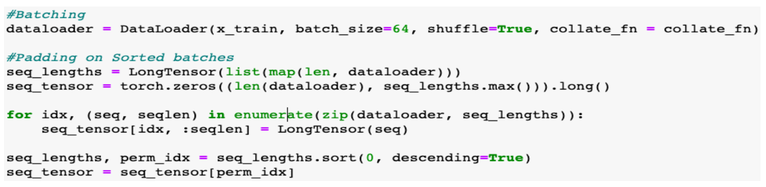
Deep Learning 2020 **(Due Date : 06.12.2020)**

Assignment # 4 - Part A

Instructor: Jakub Tomczak *Name:* Shwetambari Tiwari, *VUNetid:* sti860

1 Part 1 - Loading the data

The imdb data is successfully loaded and converted into batches with padding. I also did the bonus task of implementing variable sorted batch sizes followed by padding using the utility collate_fn. A code snippet (1) is shown in Figure 1. The batches are created by sorting the sequences in descending order.

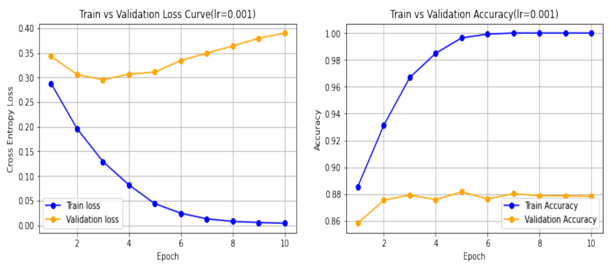


```
#Batching
dataloader = DataLoader(x_train, batch_size=64, shuffle=True, collate_fn = collate_fn)

#Padding on Sorted batches
seq_lengths = LongTensor(list(map(len, dataloader)))
seq_tensor = torch.zeros((len(dataloader), seq_lengths.max())).long()

for idx, (seq, seqlen) in enumerate(zip(dataloader, seq_lengths)):
    seq_tensor[idx, :seqlen] = LongTensor(seq)

seq_lengths, perm_idx = seq_lengths.sort(0, descending=True)
seq_tensor = seq_tensor[perm_idx]
```

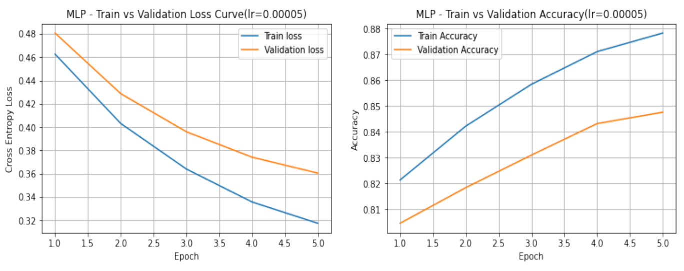
Figure 1: Variable Batch Size

2 Part B - Non Recurrent Model

The load_imdb method returns a list i2w and a dictionary w2i, the length of one of these will give the input dimension of the embedding layer. With a bigger batch size the amount of padding increases. Consider, a batch size of 128, with the majority of sequences in this list of length 38 and only 2 of them of length 102, then the whole batch needs to be padded with 0 values, to feed it into the sequence model. This increases the sparcity of the model and also the computation time. Here, a sequence length of 500 is taken for x_train padding, anything above that is sliced off. For the non recurrent model, Figure 2 (a) shows the results of the Training v/s Validation for loss and accuracy for 10 epochs, learning rate 0.001 with Adam optimizer and batch size of 64. Since, the network training time is high, different parameter setting is tested. A test was conducted for a higher batch size of 128, Figure 3 (a) and (b), show that a higher learning rate of 0.01 and a higher batch size of 128 are acceptable parameter for this architecture. Accuracy is above 80% both for training and validation set. The loss values and accuracy perform well on training set and not good on validation set. There is a possibility that the model is over-fitting for the chosen parameters. So, another experiment is conducted with a very low learning rate and a batch size of 128. The results can be observed from Figure 2 (b). With this result, we can say that the model has potential for improvements with further hyper-parameter tuning, drop-out layers and a variable batch size for computational efficiency.

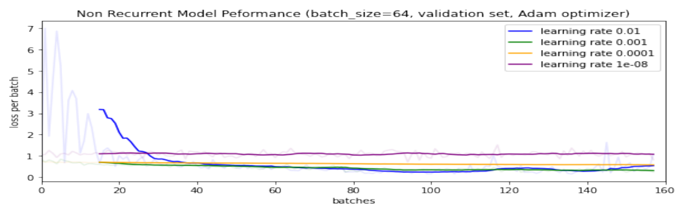


(a) Train vs Validation for lr = 0.001

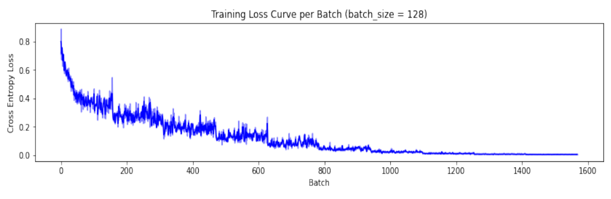


(b) Train vs Validation for lr = 0.00005

Figure 2: Performance of Non Recurrent Model



(a) Learning Rate Test

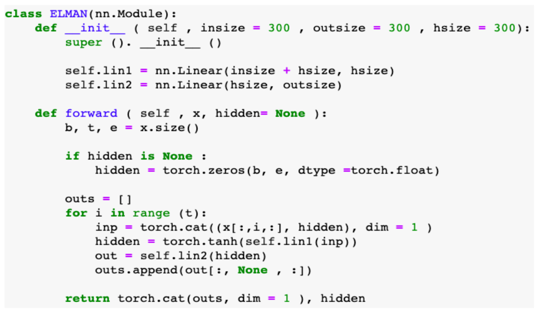


(b) Batch Size-128

Figure 3: Parameter Tuning of Non Recurrent Network

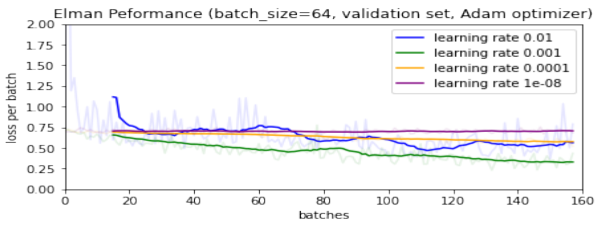
3 Part C - Writing own Elman Network

The Elman network was implemented. The training time for this model is higher in comparison to MLP implemented in part B. The class Elman implemented is shown in the Figure 4. A similar test of learning rate is conducted with a batch size of 64 on validation set. Learning rate of 0.001 performs the best with this analysis. Accuracy obtained is 86% at epoch1. This is higher than the accuracy obtained for MLP.



```
class ELMAN(nn.Module):
    def __init__ ( self , insize = 300 , outsize = 300 , hsize = 300):
        super (). __init__ ()

        self.lin1 = nn.Linear(insize + hsize, hsize)
        self.lin2 = nn.Linear(hsize, outsize)

    def forward ( self , x, hidden= None ):
        b, t, e = x.size()

        if hidden is None :
            hidden = torch.zeros(b, e, dtype =torch.float)

        outs = []
        for i in range (t):
            inp = torch.cat((x[:,i,:], hidden), dim = 1 )
            hidden = torch.tanh(self.lin1(inp))
            out = self.lin2(hidden)
            outs.append(out[:, None , :])

        return torch.cat(outs, dim = 1 ), hidden
```

(a) class Elman

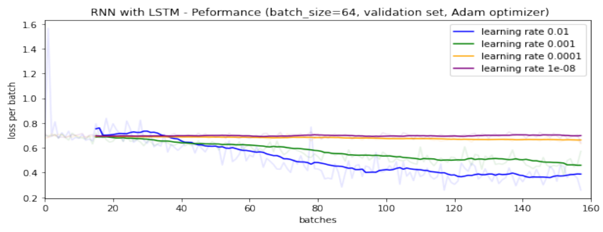


(b) Elman - Learning Rate Test

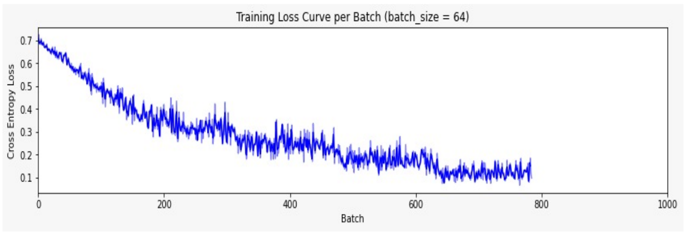
Figure 4: Performance of Elman Network

4 Part D - Using Torch's RNN

The model LSTM and RNN (2) is introduced at layer 2 of the given architecture. For LSTM model, analysis is conducted for learning rate and training loss per batch on validation set after training. Results are shown in Figure **??** It shows that the loss decreases per iteration and learning rate of 0.01 and 0.001 perform similar. A higher learning rate is chosen for further analysis and faster results.



(a) Learning Rate Comparison RNN with LSTM



(b) Batch Loss - LSTM

Figure 5: Parameter Tuning of RNN with LSTM

The 3 models (MLP, RNN and LSTM) introduced at layer 2 in the architecture are tested on test set for 5 epochs, learning rate =0.01, Adam optimizer and batch size of 128. Also, the length of each sequence of x_train and x_test is shortened to 250 for faster execution.

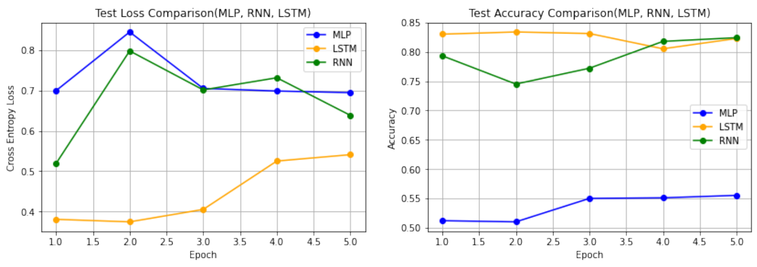


Figure 6: Parameter Tuning on Non Recurrent Network

These results are compromised but they do show a trend. At epoch 1, the loss obtained from LSTM is the lowest 38042, followed by RNN 0.51904, then MLP 0.6995. This behaviour is inline with the expected accuracy these models usually give. The accuracy obtained here for LSTM and RNN is higher than 08%, while for MLP it much lower around 50%. With a better chosen hyperparameters, these results can be improved. We might show a smoother trend, as is seen for the experiment conducted on validation set for batched loss for these architecture.

References

Harsh trivedi, pad_packed_demo.py. https://gist.github.com/HarshTrivedi/.
Pytorch -. https://pytorch.org/docs/stable/torchvision/datasets.html.

Appendix

Question 2

```
class NoRNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(300, 2)

    def forward(self, text):
        embedded = self.embedding(text)
        output = self.linear1(embedded)
        output = F.relu(output)
        output, _ = torch.max(output, dim = 1)
        output = self.linear2(output)

        return output
```

Question 3

```
class RNN_LSTM(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super().__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        embedded = self.embedding(text)
        output, hidden = self.lstm(embedded)
        output, _ = torch.max(output, dim = 1)
        output = self.linear2(output)

        return output
```