Introduction to Deep Learning
CS6910

Assignment 3

Submitted by
Chella Thiyagarajan N
ME17B179

Part B:

Aim:

You are required to classify movie reviews in one of the 5 classes based on the sentiment of the review. Make a feature representation of the review using pre-trained word embedding & your own embedding from Part-A and LSTM and then classify to one of the 5 classes.

Challenge during Implementation:

Handling Variable Length sequences: PyTorch comes with a useful feature ‘Packed Padding sequence‘ that implements Dynamic Recurrent Neural Network.

Padding is a process of adding an extra token called a *padding token* at the beginning or end of the sentence. As the number of the words in each sentence varies, we convert the variable length input sentences into sentences with the same length by adding *padding tokens*.

Padding is required since most of the frameworks support static networks, i.e. the architecture remains the same throughout the model training. Although padding solves the issue of variable length sequences, there is another problem with this idea – the architectures now process these *padding token* like any other information/data.

How to work with text data:

Torch package is used to define tensors and mathematical operations on it

TorchText is a Natural Language Processing (NLP) library in PyTorch. This library contains the scripts for preprocessing text and source of few popular NLP datasets.

Model Architecture Used:

The nn module from torch is a base model for all the models. This means that every model must be a subclass of the nn module.

I have defined 2 functions here: init as well as forward. Let me explain the use case of both of these functions-

1. Init: Whenever an instance of a class is created, init function is automatically invoked. Hence, it is called as a constructor. The arguments passed to the class are initialized by the constructor.We will define all the layers that we will be using in the model

2. Forward: Forward function defines the forward pass of the inputs.

Finally, let's understand in detail about the different layers used for building the architecture and their parameters-

Embedding layer: Embeddings are extremely important for any NLP related task since it represents a word in a numerical format. Embedding layer creates a look up table where each row represents an embedding of a word. The embedding layer converts the integer sequence into a dense vector representation. Here are the two most important parameters of the embedding layer –

1. num_embeddings: No. of unique words in dictionary
2. embedding_dim: No. of dimensions for representing a word

LSTM: LSTM is a variant of RNN that is capable of capturing long term dependencies. Following the some important parameters of LSTM that you should be familiar with. Given below are the parameters of this layer:

1. input_size : Dimension of input
2. hidden_size : Number of hidden nodes
3. num_layers : Number of layers to be stacked
4. batch_first : If True, then the input and output tensors are provided as (batch, seq, feature)
5. dropout: If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout. Default: 0
6. bidirection: If True, introduces a Bi directional LSTM

Linear Layer: Linear layer refers to dense layer. The two important parameters here are described below:

1. in_features : No. of input features
2. out_features: No. of hidden nodes

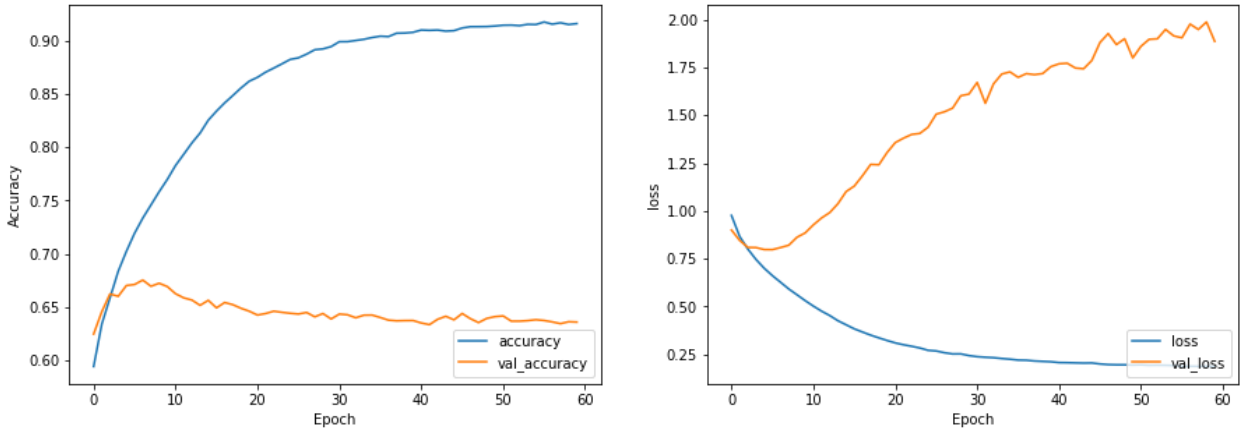
Pack Padding: As already discussed, pack padding is used to define the dynamic recurrent neural network. Without pack padding, the padding inputs are also processed by the rnn and returns the hidden state of the padded element. This an awesome wrapper that does not show the inputs that are padded. It simply ignores the values and returns the hidden state of the non padded element.

Results:

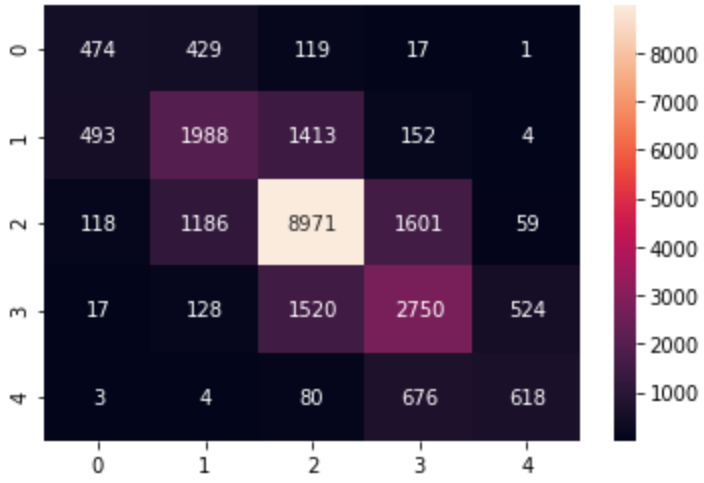
Parameters:

Embedding: GloVe - '6B'
Embedding Dimension: 200

Accuracy and Losses of Train and Val set during Training:



Confusion Matrix of our model's performance with Test set:

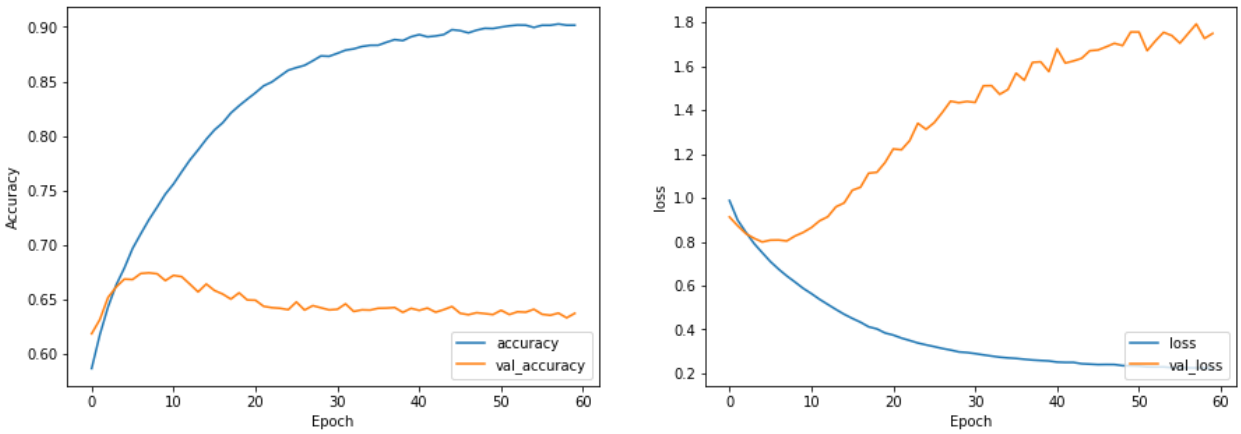


Observation:
From the observations it can be inferred that the train loss decreases much more than the validation loss and the accuracy of the test set grows much more than the validation set accuracy. Validation loss seems to grow after 10 epochs due to overfitting criteria. Class 0 and 4 seems to be minority classes in our dataset.

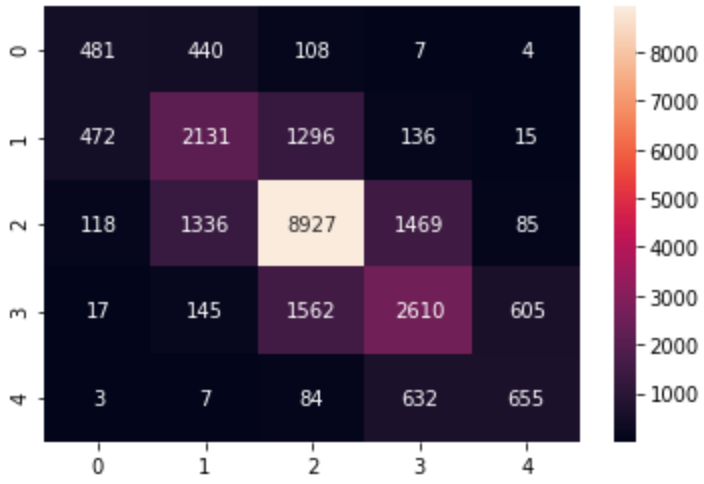
Parameters:

Embedding: GloVe - ‘6B’
Embedding Dimension: 100

Accuracy and Losses of Train and Val set during Training:



Confusion Matrix of our model’s performance with Test set:



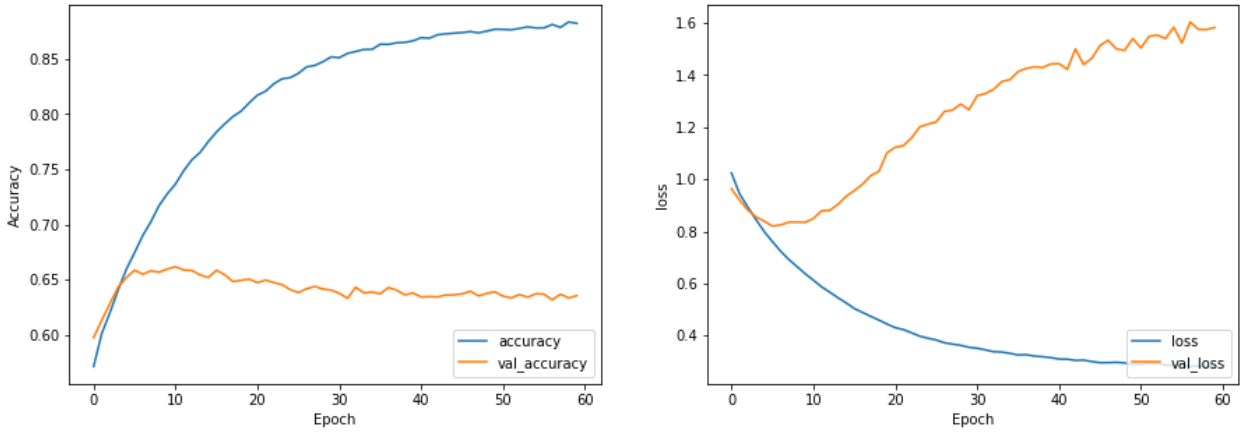
Observation:
From the observations it can be inferred that the train loss decreases much more than the validation loss and the accuracy of the test set grows much more than the validation set

accuracy. Validation loss seems to grow after 10 epochs due to overfitting criteria. Class 2 seems to have majority in our dataset which creates imbalances in our dataset.

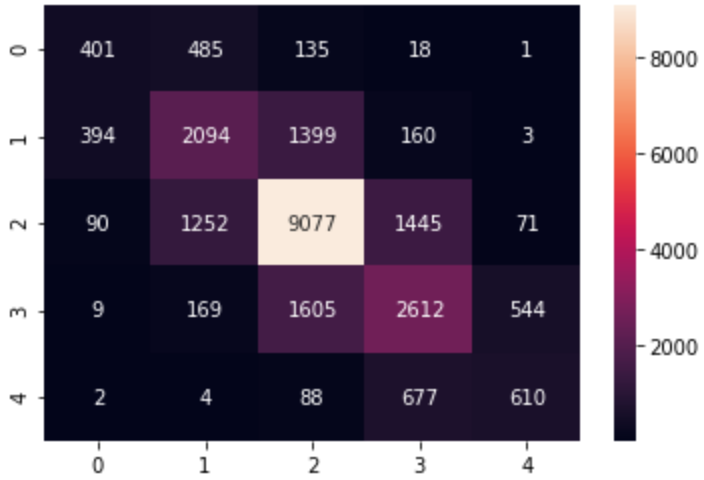
Parameters:

Embedding: GloVe - ‘6B’
Embedding Dimension: 50

Accuracy and Losses of Train and Val set during Training:



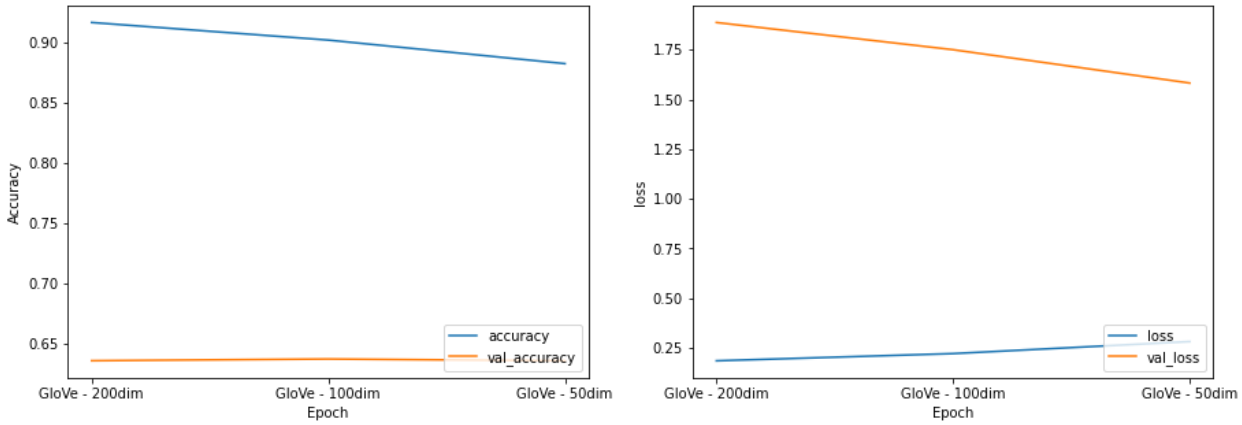
Confusion Matrix of our model’s performance with Test set:



Observation:

From the observations it can be inferred that the train loss decreases much more than the validation loss and the accuracy of the test set grows much more than the validation set accuracy. Validation loss seems to grow after 10 epochs due to overfitting criteria. By comparing confusion matrices Glove embedding dim - 50 seems to perform better than others.

Comparison of all the 3 above-mentioned embeddings:



Observation:

From the above graphs it is evident that training accuracies decrease as we decrease our embedding dimensions, It is also supported by training loss which increases over decrease in dimensions.

Whereas Validation accuracy remains flat over all kinds of embeddings. But, Validation loss seems to be decreasing as the dimensions decrease which is a positive response, This also hints us that higher embedding vectors try to overfit our dataset.

Source of problems mentioned in above observations:

More than 55 % of the data points belong to class 2 and of the remaining data points classes 1 and 3 are more in number than 0 and 4. Thus the models are overfitted on class 2 data points. Also the cross class accuracies are pretty similar for different embedding dimensions. One possible solution for this is oversampling the data points of classes in lesser frequencies such that in each epoch every class is trained equal number of times.

Part A:

Aim:
We are required to train your own Word2Vec model for the “text8” dataset. Implement (a) Bag-of-words, (b) skip-gram and (c) LSTM-based models.

Data Preparation:
we will prepare the data using the following method:

- Split tokens on white space.
- Remove all punctuation from words.
- Remove all words that are not purely comprised of alphabetical characters.
- Remove all words that are known stop words.
- Remove all words that have a length <= 1 character

Prepare vocabulary:

It is important to define a vocabulary of known words when using a bag-of-words model.

The more words, the larger the representation of documents, therefore it is important to constrain the words to only those believed to be predictive. This is difficult to know beforehand and often it is important to test different hypotheses about how to construct a useful vocabulary.

We have already seen how we can remove punctuation and numbers from the vocabulary in the previous section. We can repeat this for all documents and build a set of all known words.

We can develop a vocabulary as a *Counter*, which is a dictionary mapping of words and their count that allows us to easily update and query.

Word2Vec:

Using the position of each word in these unique words list a dictionary mapping words to their indices was generated. These indices were passed to the embedding layer in the different word2vec models to generate the output for each particular word2vec model.

Model Architectures:

For the Bag of Words model, around each word in the dataset a window of two is considered on either side. For each word in the window except the center word, the embeddings are generated and its mean is passed to a linear layer with an output for each word in the vocabulary to predict which could be the ideal centre word.

CBOW:

```
class CBOW(torch.nn.Module):

    def __init__(self, vocab_size, embedding_dim):

        super(CBOW, self).__init__()



        self.embeddings = nn.Embedding(vocab_size, embedding_dim)

        self.linear1 = nn.Linear(embedding_dim, vocab_size)

        self.activation_function = nn.LogSoftmax(dim = -1)



    def forward(self, inputs):

        out = torch.mean(self.embeddings(inputs),1)

        out = self.linear1(out)

        return self.activation_function(out)



    def get_word_emdedding(self, word):

        word = torch.tensor([wordToIndexDictionary[word]])

        return self.embeddings(word)
```

In the SkipGram Model, for each word in the window the centre word is used as the input to the embedding layer followed by a linear layer with the output dimension being the vocabulary size, to predict the ideal word which would lie in the window of the centre word.

SkipGram:

```
class SkipGram(nn.Module):

    def __init__(self, vocab_size, embedding_dim):

        super(SkipGram, self).__init__()



        self.embeddings = nn.Embedding(vocab_size, embedding_dim)

        self.linear1 = nn.Linear(embedding_dim, vocab_size)

        self.activation_function = nn.LogSoftmax(dim = -1)



    def forward(self, inputs):

        out = self.embeddings(inputs)

        out = self.linear1(out)

        return self.activation_function(out)



    def get_word_embedding(self, word):

        word = torch.tensor([wordToIndexDictionary[word]])

        return self.embeddings(word)
```

In the LSTM based model the embeddings of a sequence of words is passed through an LSTM followed by a linear layer with the output dimension of vocabulary size to predict the next word after the sequence.

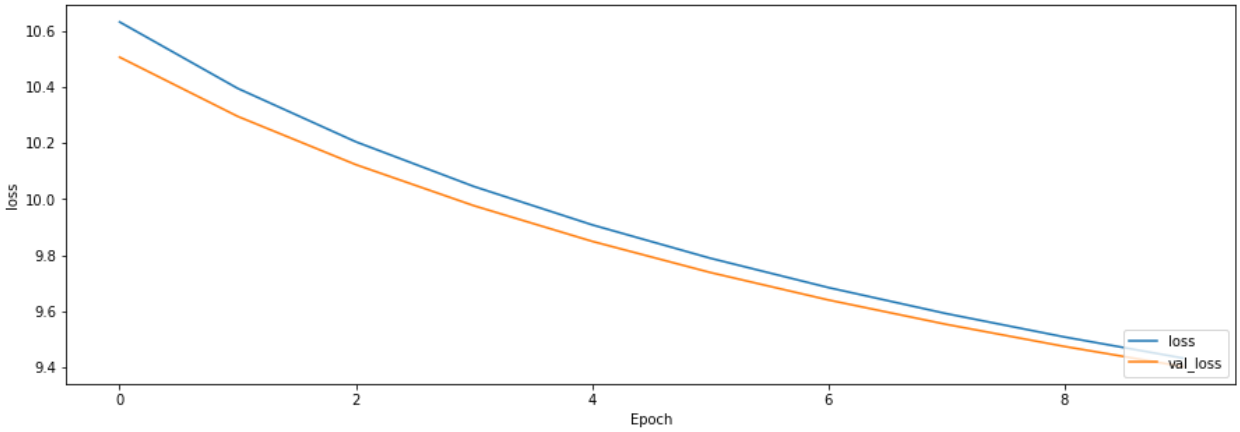
LSTM:

```
class LSTM(torch.nn.Module):

    def __init__(self, vocab_size, embedding_dim, hidden_dim,
window_size):

        super(LSTM, self).__init__()



        self.embeddings = nn.Embedding(vocab_size, embedding_dim)

        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True,
bidirectional=True)

        self.linear = nn.Linear((window_size*4*hidden_dim), vocab_size)

        self.activation_function = nn.LogSoftmax(dim = -1)

        self.window_size = window_size

        self.hidden_dim = hidden_dim



    def forward(self, inputs):

        out = self.embeddings(inputs)

        out, _ = self.lstm(out)

        out =
self.linear(out.reshape([out.shape[0],(self.window_size*4*self.hidden_dim)
]))

        return self.activation_function(out)
```

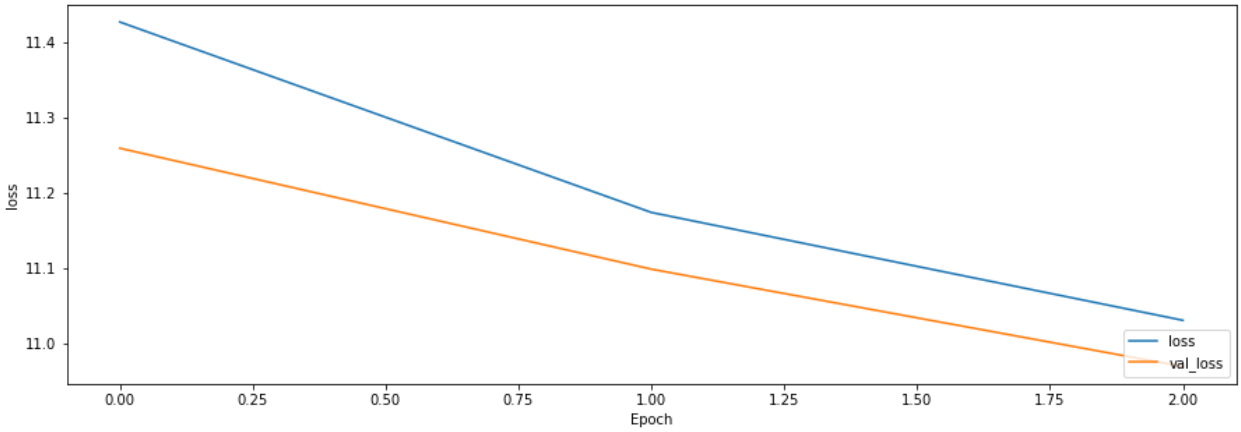
```
    def get_word_emdedding(self, word):

        word = torch.tensor([wordToIndexDictionary[word]])

        return self.embeddings(word)
```

Results:

Case: CBOW



Case: SkipGram



Case: LSTM

Epoch	Train Loss	Val Loss
1	11.23969738	10.68458324

I could only run for only one epoch as I got Error: `Buffered data was truncated after reaching the output size limit.` Often in Colab and couldn't be able to run more.

Observations:

CBOW was run for 6 epochs , skipgram were run for 3 epochs and LSTM was run for only 1 epochs, The reason due to very small number of epochs is due to an Error: `Buffered data was truncated after reaching the output size limit.` This error limited further experimenting and getting good representational knowledge of these models.

From the above graphs and table it is evident that all the above models have the ability to learn word embeddings as we can observe decrease in both training and validation set losses.

For the SkipGram Model the training time is large and in the limited compute time available 3 epochs were run. The loss was still decreasing in both the train and test sets indicating that learning was still happening and the minima hadn't been reached yet. In the LSTM based Model the train set loss keeps decreasing with each epoch but the test set loss increases for 2 epochs. This might be because the lstm output dimension of 25 used in the model might be high which could have started over fitting from early on. Since the compute resource was low further experiments couldn't be performed to verify this.