Deep Learning

DL basic concepts with CNN and LSTM model

Table of Contents

Question 1

Develop a neural network to classify, whether a cell is infected or not using Convolution Neural Network. Perform splitting into training, testing and validation sets with 80:10:10 ratio.

1. Visualize 5 random images from both the classes. [5 marks]

2. Implement a CNN architecture with blockA [9x9] followed by fully connected layer, blockB [6x6] followed by max pooling, blockC [3x3] followed by fully connected layer and finally a sigmoid layer. [5 marks]

3. Initialize your neural network weights by using following initialization methods: [3 marks]
   a. Zero initialization
   b. Random Initialization
   c. He initialization

Which initialization approach is best and why?

4. Implement Dropout and use i) After convolutional layers, ii) Between fully connected layers. Compare the performance in both the cases. [6 marks]

5. Keeping the above architecture same, implement the following regularizations and do a thorough analysis on the output of each one of them: [6 marks]
   a. L1 Regularization
   b. L2 Regularization

Compare the performance in both the cases. Which regularization is better and why?

Approach

- Understood that the images are divided into two classes, hence it is a classification problem.
- Data was processed, splitted, normalized and converted to required form (tensors). (More in Pre processing)
- At last the dense layer is used with sigmoid activation function and 2 neurons.
- The plots are formed of accuracy vs epochs and loss vs epochs.

Epochs = 20

Model compile =

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr = learning_rate)
```

- (Fact:) SparseCategoricalCrossEntropy doen't require one hot encoding in labels, makes our life easier.

Pre-Processing

- The images were resized to (32,32,3) shape along with cv2.INTER_AREA interpolation technique.
- The image matrices are normalized by dividing it by 255.0.
- The dataset is then divided into train, validation and test dataset in the ratio 8:1:1.
- The numpy image matrices are then converted to tensors before feeding into the network.

Grayscaling was not done to not lose anyimportant information.

Outputs and Analysis

1.1

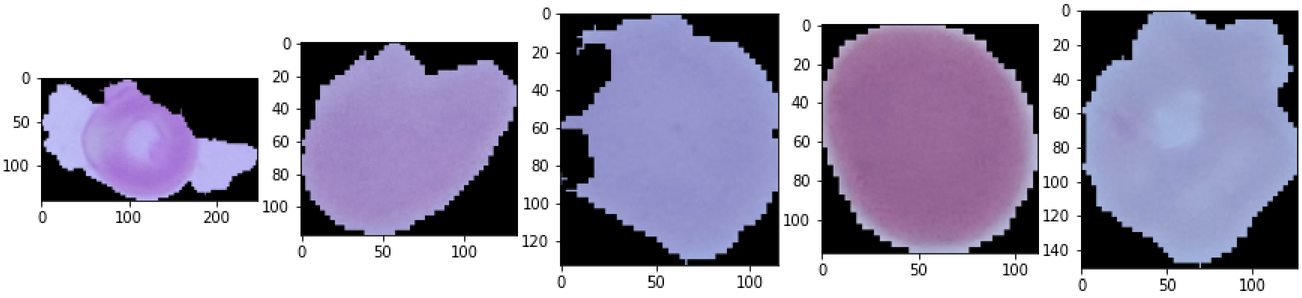


Fig: Uninfected Images

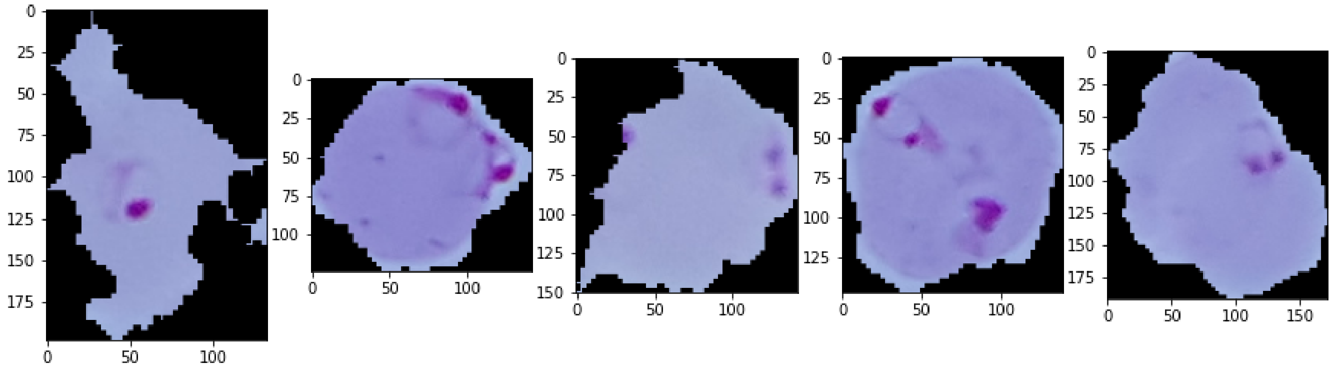


Fig: Parasitized Images

Analysis -

- Spots (some dark and some light) can be seen in the parasitized images
- Uninfected images look pretty clear

1.2

```
class CNN(nn.Module):
   def __init__(self,dropout = '',drate = 0.4,weight_ini=None):
       super(CNN, self).__init__()
       self.dropout = dropout
       self.drate = drate
       self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3)
       if dropout == 'afterconv':
         self.conv1_drop = nn.Dropout2d(p=drate)
       self.conv2 = nn.Conv2d(10, 20, kernel_size=3)
       if dropout == 'afterconv':
         self.conv2_drop = nn.Dropout2d(p=drate)
       self.conv3 = nn.Conv2d(20, 20, kernel_size=3)
       if dropout == 'afterconv':
         self.conv3_drop = nn.Dropout2d(p=drate)
       self.fc1 = nn.Linear(80, 30)
       if dropout == 'betweendense':
         self.fc1_drop = nn.Dropout(p=drate)
       self.fc2 = nn.Linear(30, 2)

       if(weight_ini is not None):
           for m in self.modules():
               if isinstance(m, nn.Linear):
                 if weight_ini=='zero':
                   nn.init.zeros_(m.weight)
                 elif weight_ini=='random':
                   nn.init.normal_(m.weight)
                 else:
                   nn.init.xavier_uniform_(m.weight)

   def forward(self, x):
       if self.dropout=='afterconv':
         x = F.relu(F.max_pool2d(self.conv1_drop(self.conv1(x)), 2))
       else:
         x = F.relu(F.max_pool2d(self.conv1(x), 2))
       if self.dropout=='afterconv':
         x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
       else:
         x = F.relu(F.max_pool2d(self.conv2(x), 2))
       if self.dropout=='afterconv':
         x = F.relu(F.max_pool2d(self.conv3_drop(self.conv3(x)), 2))
       else:
         x = F.relu(F.max_pool2d(self.conv3(x), 2))
       x = x.view(x.shape[0],-1)
       x = F.relu(self.fc1(x))
       if self.dropout == 'betweendense':
         x = self.fc1_drop(x)
       x = F.sigmoid(self.fc2(x))
       return x

   def compute_l1_loss(self, w):
     return torch.abs(w).sum()

   def compute_l2_loss(self, w):
     return torch.square(w).sum()
```

```
CNN(
  (conv1): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=80, out_features=30, bias=True)
  (fc2): Linear(in_features=30, out_features=2, bias=True)
)
```

Analysis -

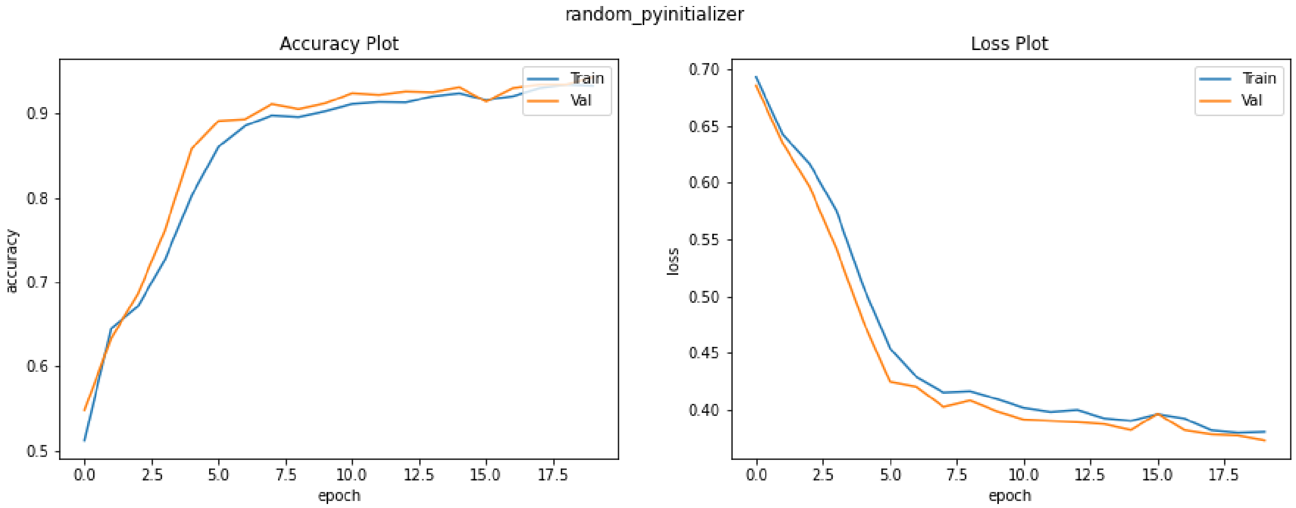
- The number of kernels in CNN layers keeps on increasing as we go deeper in the network. This is because there are more kernels needed as the number of combinations from previous kernels will be more.
- Flatten layer was necessary as it makes the 2D input to it into 1D. Without this the model was not performing very well. This layer made around 20% jump the accuracy.

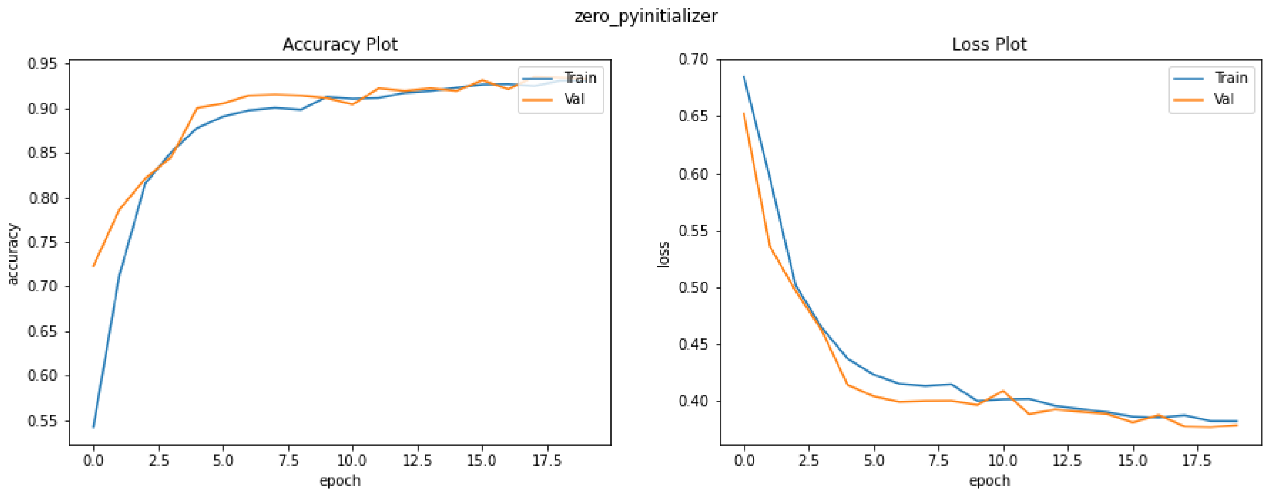
1.3

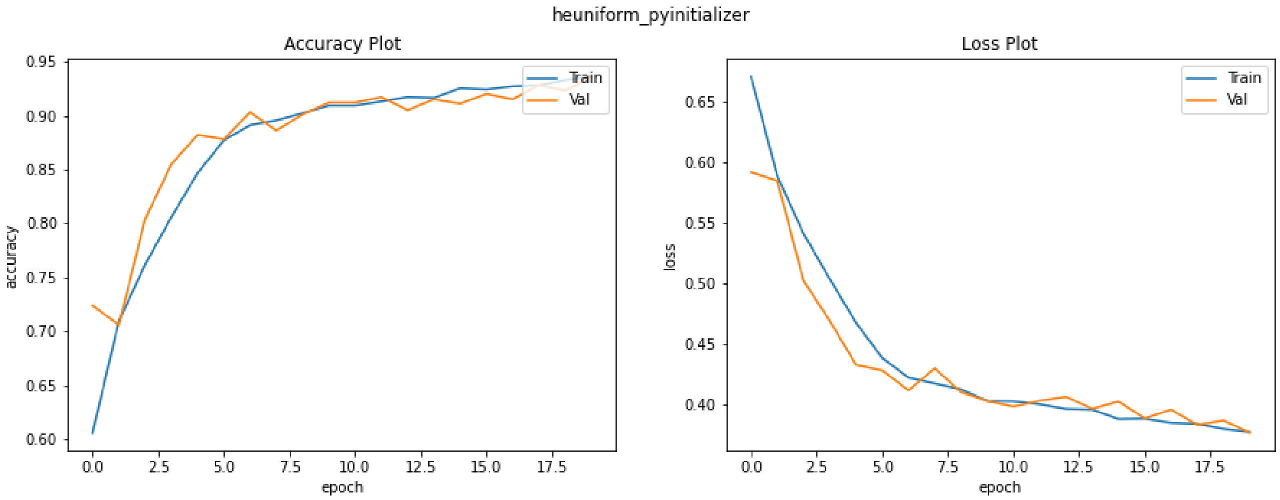
Initializers

	random initializer	zero initializer	heuniform initializer
Pytorch	Test Accuracy = 0.933 Test Loss = 0.380	Test Accuracy = 0.923 Test Loss = 0.388	Test Accuracy = 0.935 Test Loss = 0.380
Tensorflow model with more parameters (just for display of effect)	Test Accuracy = 0.783 Test Loss = 0.48	Test Accuracy = 0.5 Test Loss = 0.69	Test Accuracy = 0.82 Test Loss = 0.40

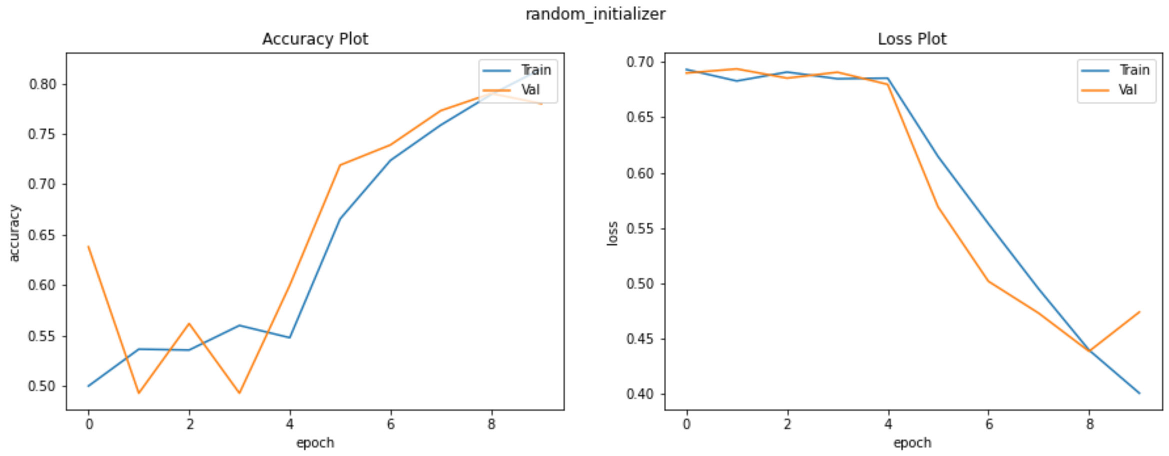
Pytorch:

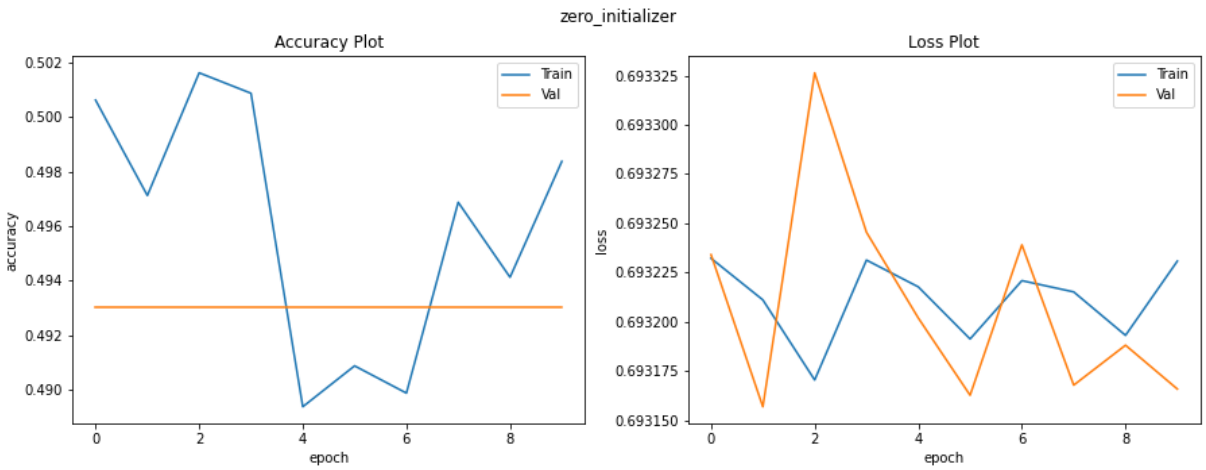


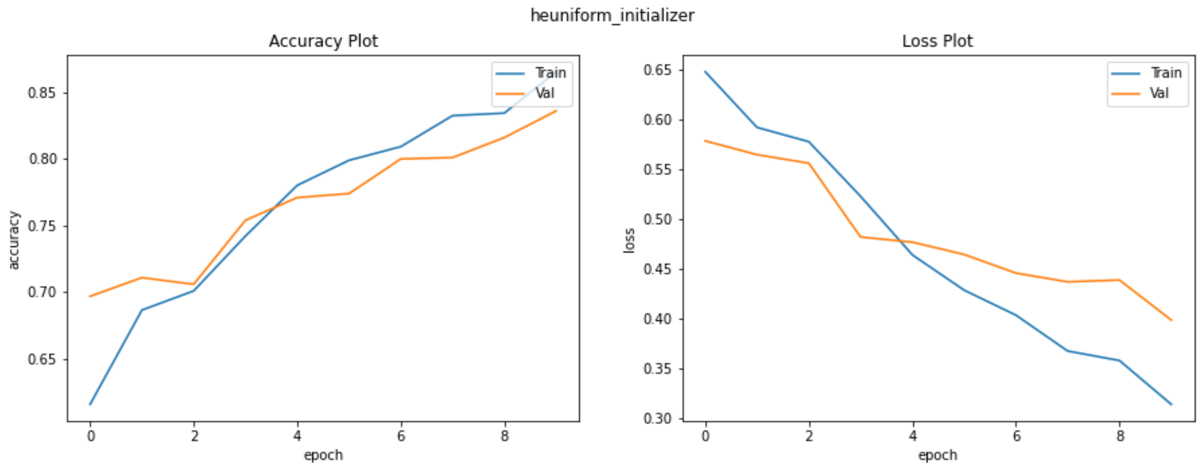




Tensorflow (with keras):







Analysis -

- We have a winner and it is - he uniform initializer. This was also expected beforehand as it overcomes the shortcomings of the other two initialization techniques. It takes into account the previous layer size and hence the model architecture. The weights are still random but differ in range depending on the size of the previous layer of neurons. This provides a controlled initialization hence the faster and more efficient gradient descent.
- Zero initializer was expected to perform worse as all the weights are zeros, hence it makes all the hidden layers identical and symmetric as all the weights will have the same value with subsequent iterations. This defeats the purpose of learning and generalizing.
- Random Initializer performed better than zero one. It assigns random value to all the weights, hence breaking out of the symmetry problem. But he does have the problematic cases when assigned value is too large or too small (it also depends on the activation function though). In our case we have used ReLU which reduces this problem, but sigmoid is used in the last layer. There can be a

problem of vanishing gradient in some cases. But overall these problems affect learning and hence the performance suffers.

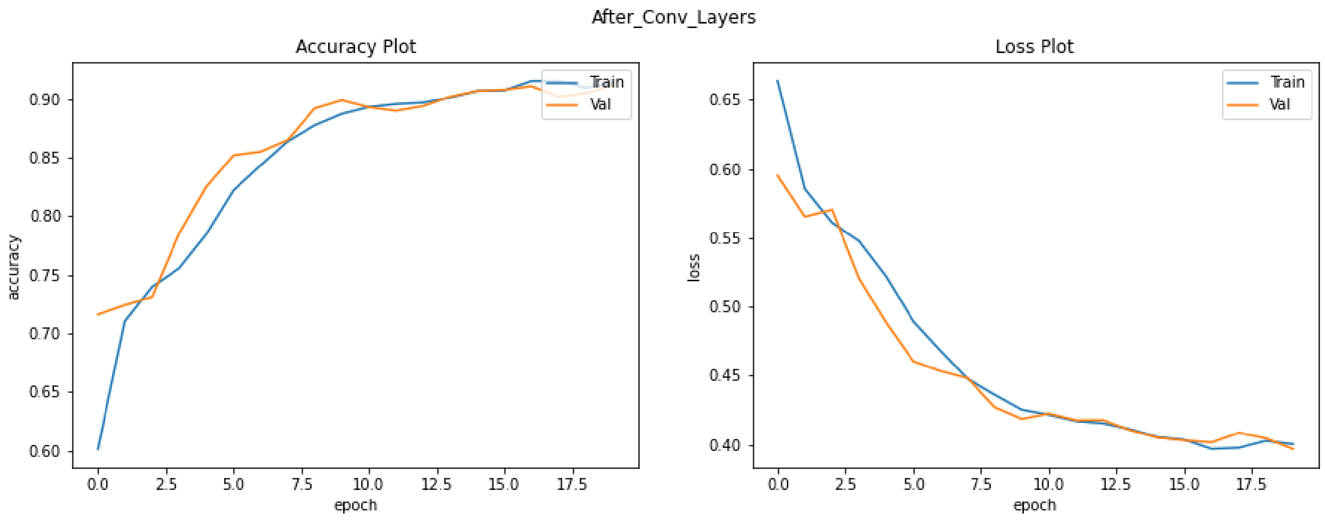
- Keeping above points in mind, we see a nice downwards trend in Loss vs epoch for he uniform > random > zero (is all over the place).

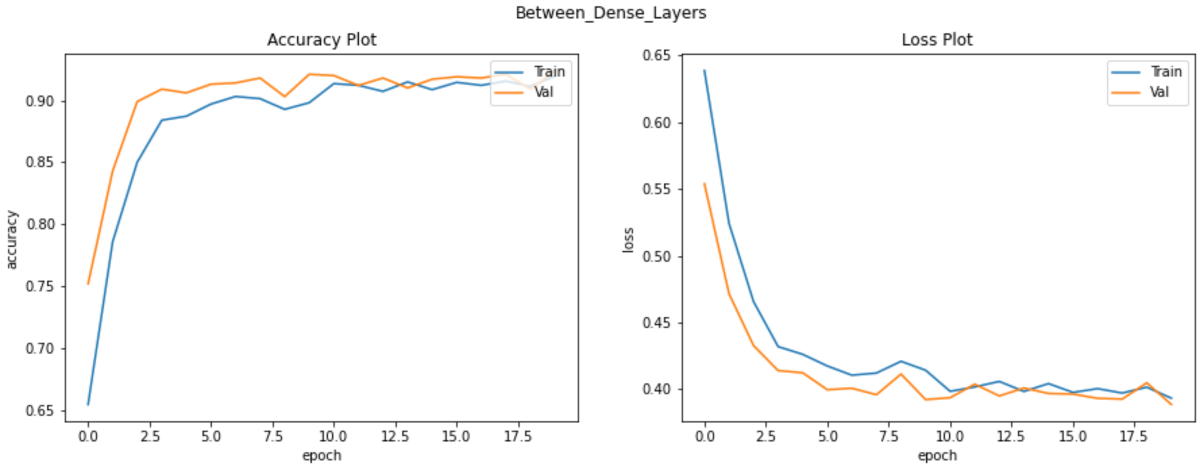
1.4

Dropout rate = 0.2

After Convolution Layers	Between Dense Layers
Test Accuracy = 0.75 Test Loss = 0.57 Test Accuracy = 0.916 Test Loss = 0.40	Test Accuracy = 0.86 Test Loss = 0.34 Test Accuracy = 0.912 Test Loss = 0.3990833146572113

CNN(
  (conv1): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1))
  (conv1_drop): Dropout2d(p=0.4, inplace=False)
  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.4, inplace=False)
  (conv3): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1))
  (conv3_drop): Dropout2d(p=0.4, inplace=False)
  (fc1): Linear(in_features=80, out_features=30, bias=True)
  (fc2): Linear(in_features=30, out_features=2, bias=True)
)
	CNN(
	  (conv1): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1))
	  (conv2): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1))
	  (conv3): Conv2d(20, 20, kernel_size=(3, 3), stride=(1, 1))
	  (fc1): Linear(in_features=80, out_features=30, bias=True)
	  (fc1_drop): Dropout(p=0.4, inplace=False)
	  (fc2): Linear(in_features=30, out_features=2, bias=True)
	)





Analysis -

- Dropout layer after dense layers performed much better than the dropout layers after the convolution layers.
- Dropout layers should not be used after the convolutional layers as we slide the filter over the width and height of the input image, we produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. As the dropout layer neutralizes (makes it zero) random neurons, we might end up losing important features in an image in our training process.
- As discussed above, we see nice downwards progression in the loss plot for Between dense layers while the other one almost shoots up as we would be making wrong predictions, due to loss of important features. (as checked in tensorflow model)

- In this pytorch model, both models show similar metrics. But we can see that in case of dense layers, the train loss converges faster and is certain (always below val loss) and also in the accuracy plot we see a similar trend. As the model is not very big, the loss of information is not seen prominently.

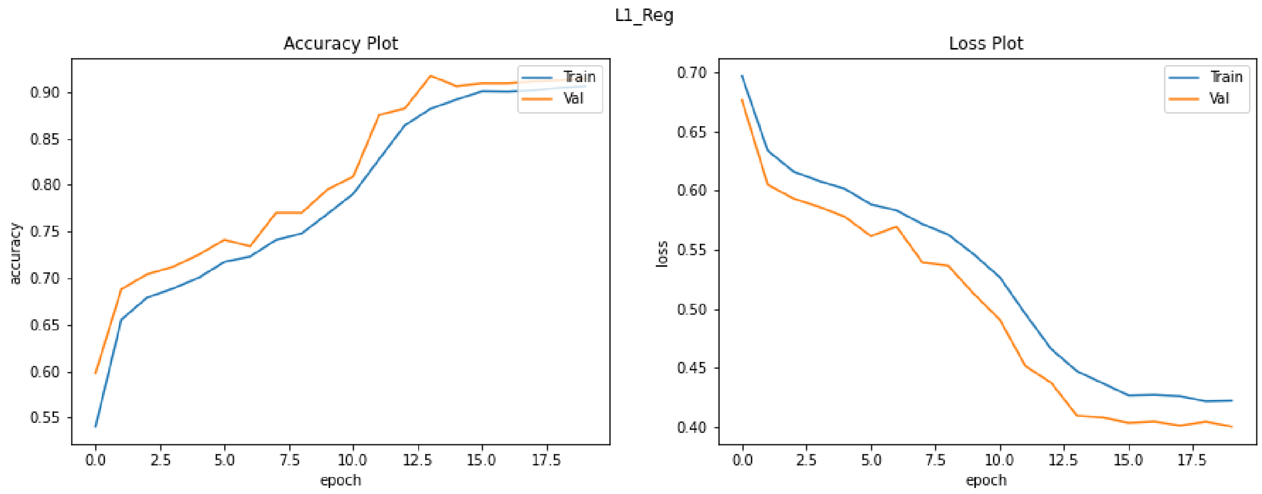
1.5

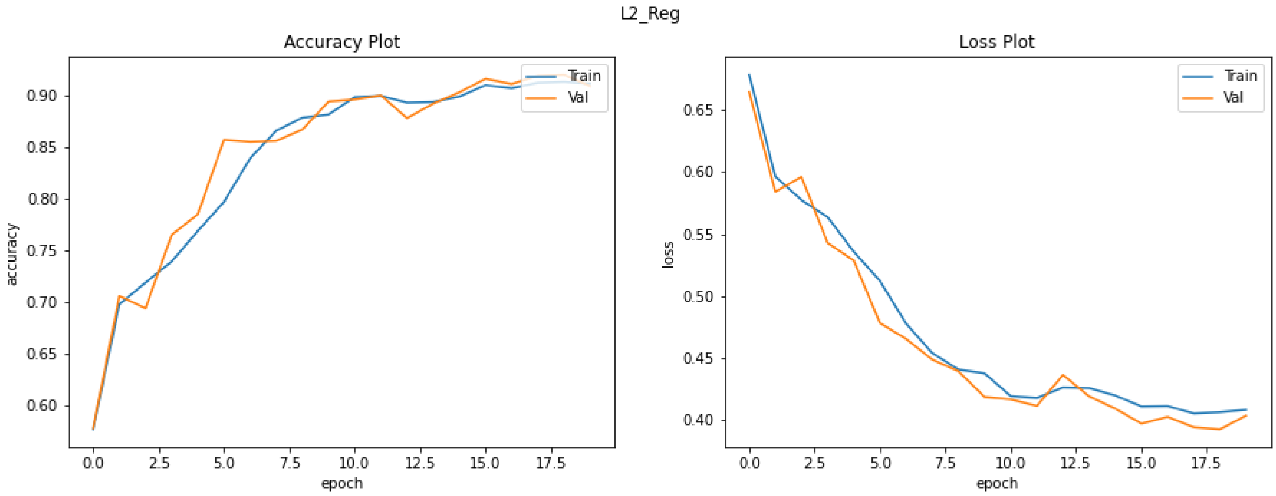
Regularization is applied on the dense layers only. Before it was applied to all the layers (conv and dense) but it was giving accuracy results close to (60%).

L1 Regularizer	L2 Regularizer

`def compute_l1_loss(self, w):`
`    return torch.abs(w).sum()`
`l1 = l1_weight * model.compute_l1_loss(torch.cat(l1_parameters))`
`loss += l1`
	`def compute_l2_loss(self, w):`
	`    return torch.square(w).sum()`
	`l2 = l2_weight * model.compute_l2_loss(torch.cat(l1_parameters))`
	`loss += l2`

`Test Accuracy =  0.903`
`Test Loss =  0.411`
	`Test Accuracy =  0.906`
	`Test Loss =  0.406`





Analysis -

- L2 Regularizer performs slightly better than the L1 regularizer based on the above results.
- The above graphs show that L2 helps in a better convergence than L1 and also based the accuracy result of val and test.
- As the almighty google says, L1 regularization penalizes the sum of absolute values of the weights, whereas L2 regularization penalizes the sum of squares of the weights. L2 regularization only shrinks the weights to values close to 0, rather than actually being 0. On the other hand, L1 regularization shrinks the values to 0. Even though L1 is said to be more robust to outliers, in our case there is no such case.

Question 2

Dialogue-act Classification: For a given dialogue students need to develop a program to predict act of utterance at time T with the help of previous X utterances as context.

1. Visualize dialogue corpus and show stats of the train and test file. [2 Marks]

2. Implement a program using just LSTM and linear layers to predict act of utterance at time T considering previous X utterances' context. [8 Marks]
Note: Students need to propose an architecture for this.

3. Now, show plots for accuracy and weighted F1 scores for X = {0,1,2,3,4} [10 Marks]

4. Does the performance of the model increase with increase in X? Justify. [5 Marks]

Approach

- This is a text classification task. There are 4 types of acts associated with the utterance. Hence it is a multiclass classification.
- X previous occurrences context were seen taken as list [0,1,2,3,4]. For every value a new dataset was formed considering the X previous contexts and then the model is trained and plots are created.
- Plots of accuracy vs epochs, loss vs epochs, f1 score vs epochs are created for analysis purposes.

Pre-Processing

- The words are converted to lowercase
- Punctuations are removed
- Alphanumeric words like 'amc78' are replaced with jus alphabets 'amc.
- Stopwords are removed
- The words are lemmatized
- 'Extra spaces are removed

- After this the text is tokenized and converted into word embedding using glove and then converted to data iterables using data loaders for pytorch.
- The labels are converted to one hot encodings.

Outputs and Analysis

2.1

Train Data

No of utterances in Train data = 5090
Avg length of sentences (in chars) = 62
Avg length of sentences (in words) = 15
Max length of sentence = 550
Min length of sentence = 5
Vocab size = 5819
No of sentences for act 1 are = 2859
No of sentences for act 2 are = 1497
No of sentences for act 3 are = 426
No of sentences for act 4 are = 308

Test Data

No of utterances in Train data = 722
Avg length of sentences (in chars) = 61
Avg length of sentences (in words) = 15
Max length of sentence = 474
Min length of sentence = 6
Vocab size = 1781
No of sentences for act 1 are = 363
No of sentences for act 2 are = 217
No of sentences for act 3 are = 83
No of sentences for act 4 are = 59

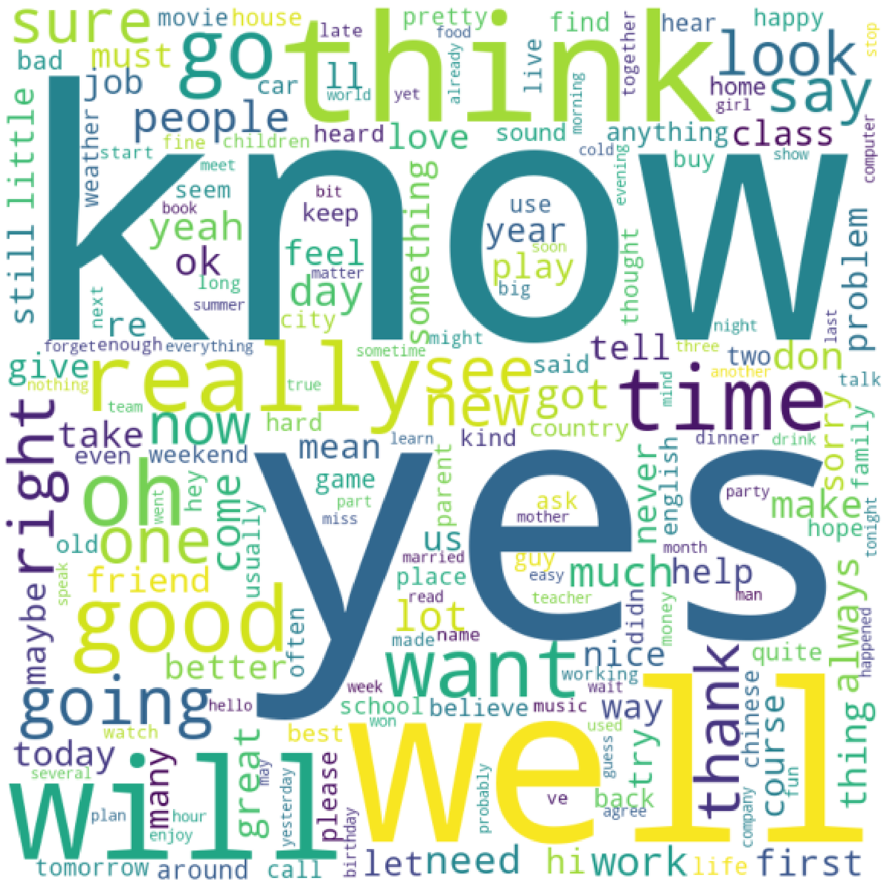


Fig: Word Cloud for Training Data

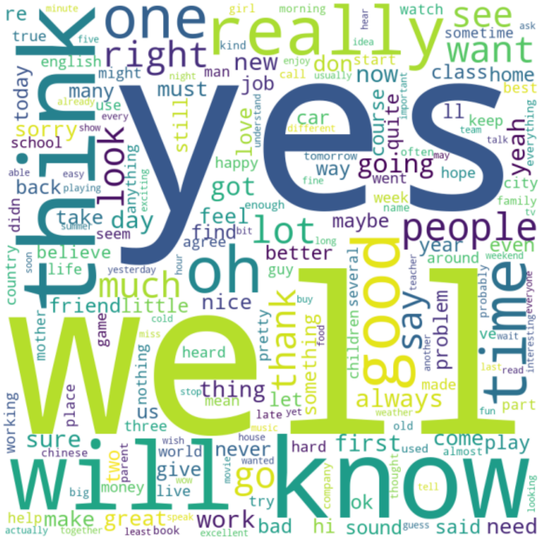


Fig: Act 1

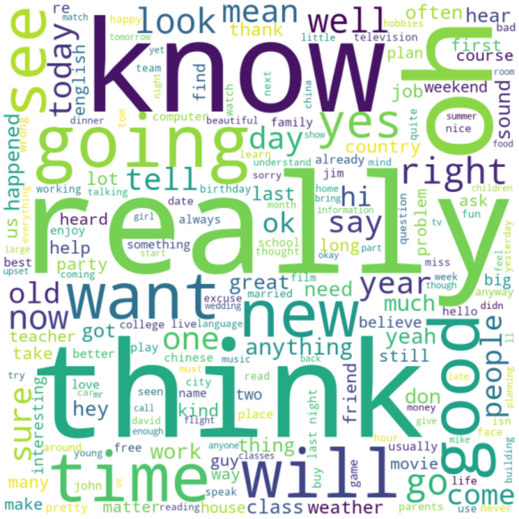


Fig: Act 2

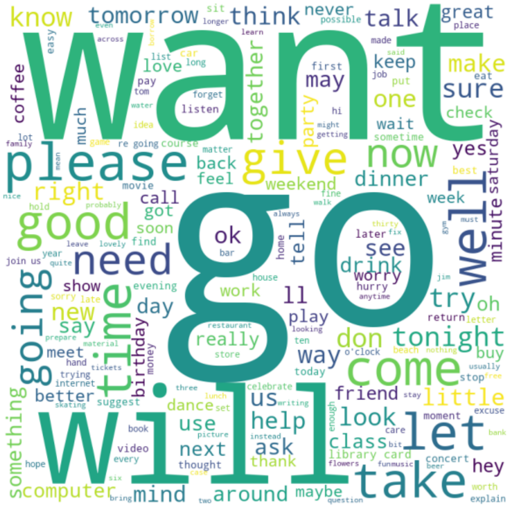


Fig: Act 3

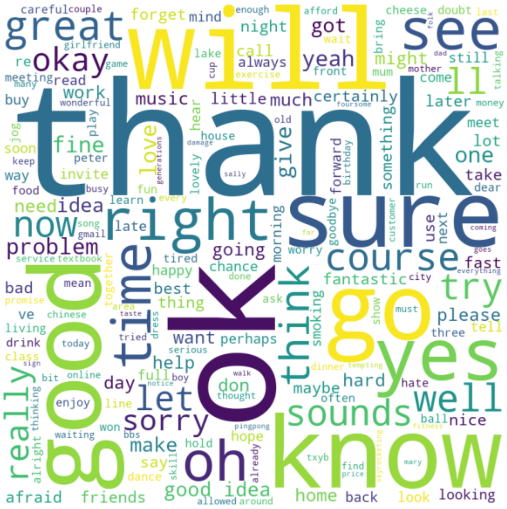


Fig: Act 4

Analysis -

- Word clouds give us an idea of the type of words to expect in each case.
- There are 4 dialog acts in the data and encoded as shown below:
  - inform : 1 - yes, will know, yeah, maybe, think, right….
  - question : 2 - really, going, know….
  - directive : 3 - go, will, want, take, let…..
  - commissive : 4 - thank, sure, ok…..
- The statistics of the length, no of utterances etc above are self explanatory.

2.2

```
class RNN(nn.Module):
   def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim,
n_layers, bidirectional, dropout):
       super().__init__()
       self.embedding = nn.Embedding(vocab_size, embedding_dim)
       self.lstm =
nn.LSTM(embedding_dim,hidden_dim,num_layers=n_layers,bidirectional=bidirectiona
l,dropout=dropout,batch_first=True)
       self.fc = nn.Linear(hidden_dim * 2, output_dim)
       self.act = nn.Sigmoid()

   def forward(self, text, text_lengths):
       embedded = self.embedding(text)
       packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded,
text_lengths,batch_first=True,enforce_sorted=False)
       packed_output, (hidden, cell) = self.lstm(packed_embedded)
       hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1)
       dense_outputs=self.fc(hidden)
       outputs=self.act(dense_outputs)
       return outputs
```

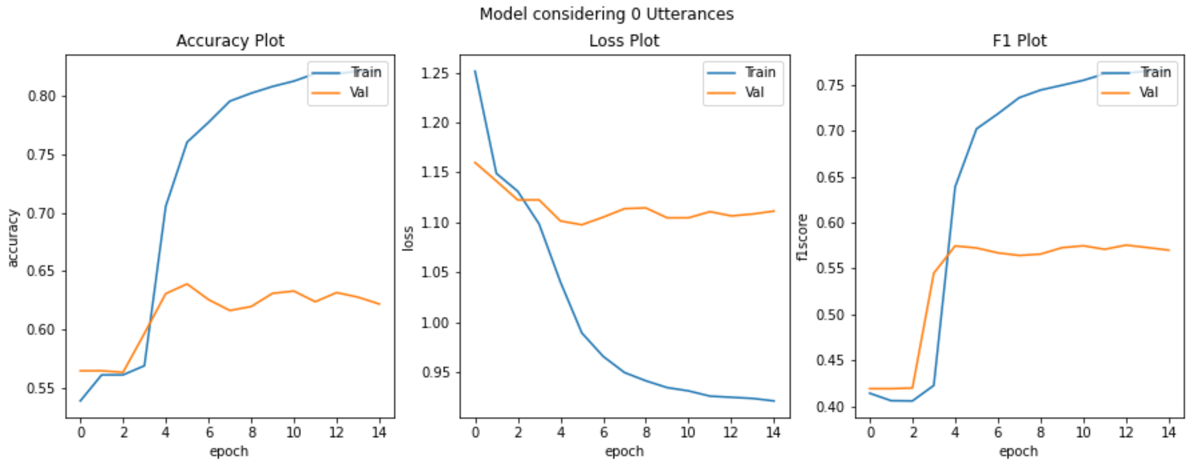
```
RNN(
  (embedding): Embedding(5816, 200)
  (lstm): LSTM(200, 32, num_layers=2, batch_first=True, dropout=0.4,
bidirectional=True)
  (fc): Linear(in_features=64, out_features=4, bias=True)
  (act): Sigmoid()
)
```

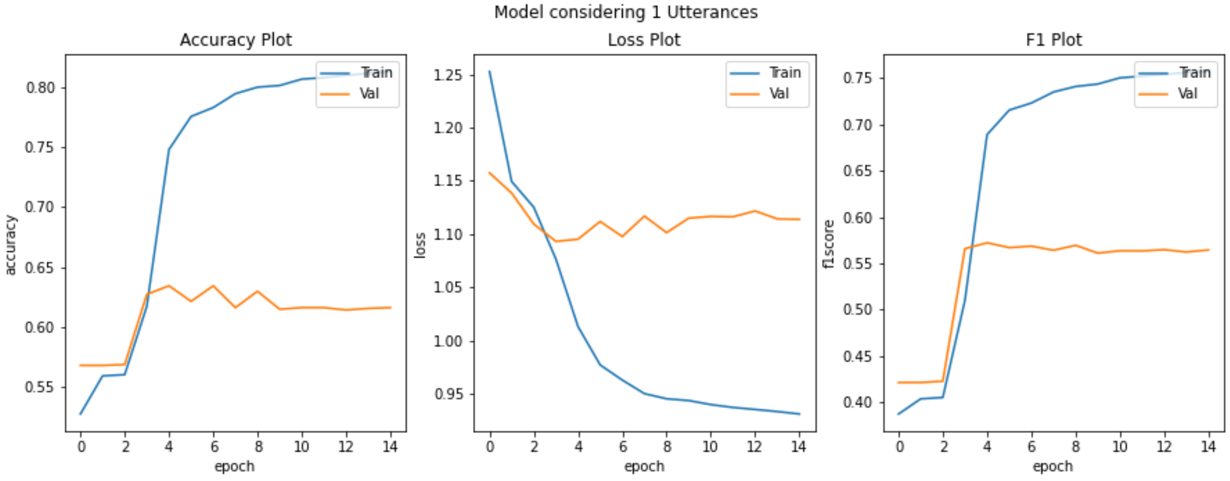
Analysis - A simple structure was designed as the data is sparse (less in no). A deeper complicated model will not work efficiently and will tend to overfit

2.3

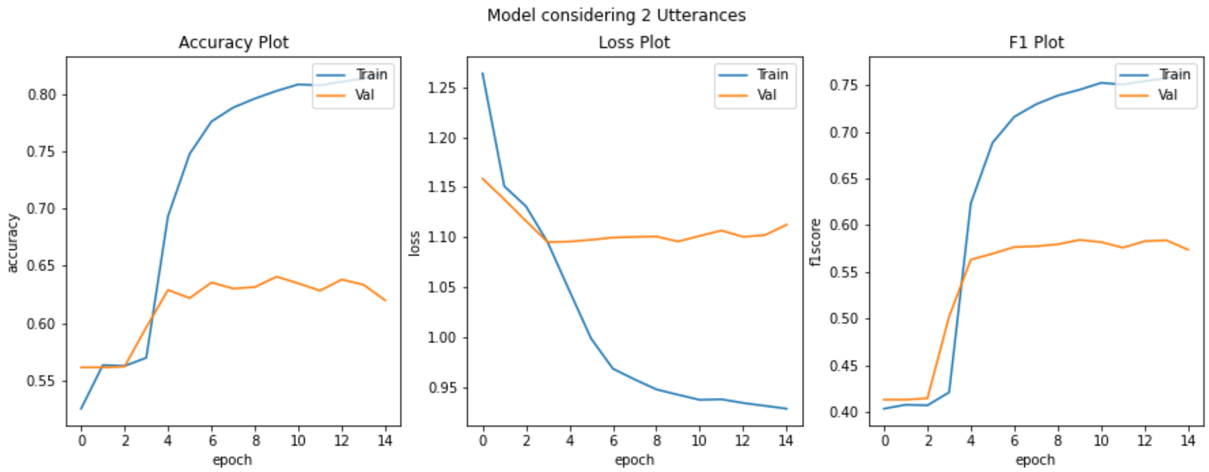
X	Test Stat

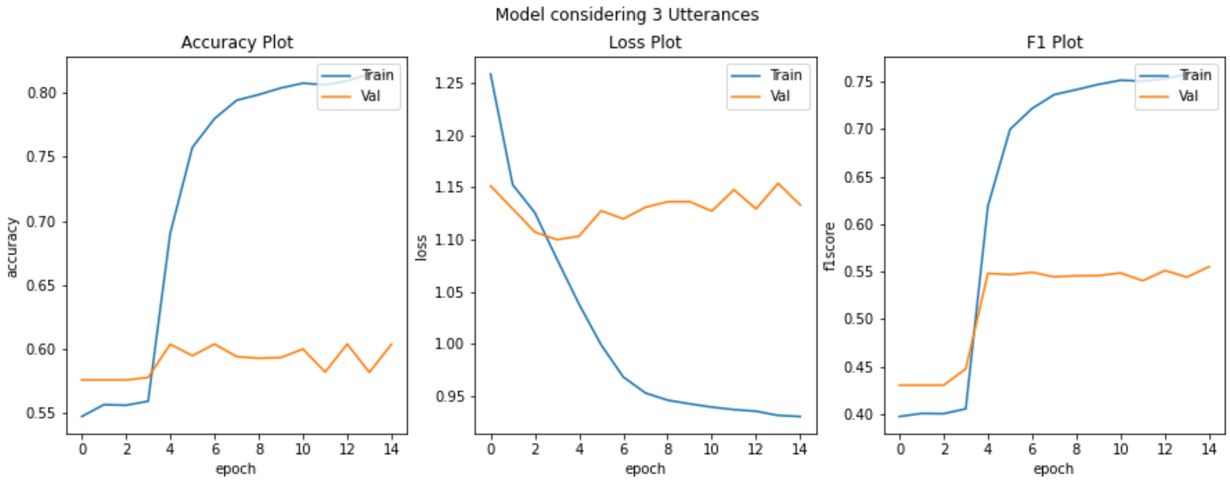
0	Test Accuracy = 0.5886863420406977 Test Loss = 1.149266133705775 Test f1 = 0.5200124234156559
1	Test Accuracy = 0.5877757370471954 Test Loss = 1.1541457275549571 Test f1 = 0.5131300730186233
2	Test Accuracy = 0.57421875 Test Loss = 1.1635022858778636 Test f1 = 0.5164106772280782
3	Test Accuracy = 0.5786458353201548 Test Loss = 1.1580149829387665 Test f1 = 0.5094619792650147
4	Test Accuracy = 0.584 Test Loss = 1.155 Test f1 = 0.517

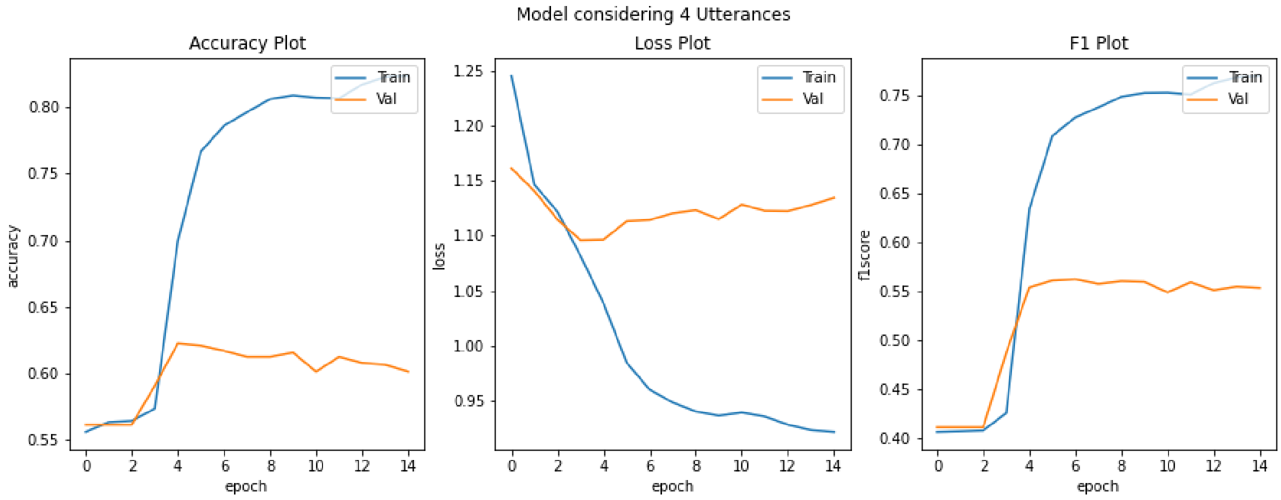




2.4


Model considering 2 Utterances
Accuracy Plot
Loss Plot
F1 Plot
Train
Val
accuracy
loss
f1score
epoch
0.55
0.60
0.65
0.70
0.75
0.80
0.95
1.00
1.05
1.10
1.15
1.20
1.25
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0
2
4
6
8
10
12
14


Model considering 3 Utterances
Accuracy Plot
Loss Plot
F1 Plot
Train
Val
accuracy
loss
f1score
epoch
0.55
0.60
0.65
0.70
0.75
0.80
0.95
1.00
1.05
1.10
1.15
1.20
1.25
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0
2
4
6
8
10
12
14


Model considering 4 Utterances
Accuracy Plot
Loss Plot
F1 Plot
Train
Val
accuracy
loss
f1score
epoch
0.55
0.60
0.65
0.70
0.75
0.80
0.95
1.00
1.05
1.10
1.15
1.20
1.25
0.40
0.45
0.50
0.55
0.60
0.65
0.70
0.75
0
2
4
6
8
10
12
14

2.4

Analysis - The accuracy,f1 score increases and loss decreases for X (2 to 4) as we are getting more data/text to predict the labels and important co-occurrences for that label might be captured in this situation.
The model performs better for X=0 and X=1 as the context window is small, LSTM will mostly be learning within the conversation only. Also bidirectional layer helps in improving this performance.

The overfitting is due to less number of data points. As the model is very simple, shrinking the model for reducing the overfitting is not preferred. Dropout rates of 0.2 to 0.4 are tested but there is not much difference. Early stopping can be implemented to stop, but not preferred as the overfitting happens at just 3 epochs.