NLP Assignment 2 Report

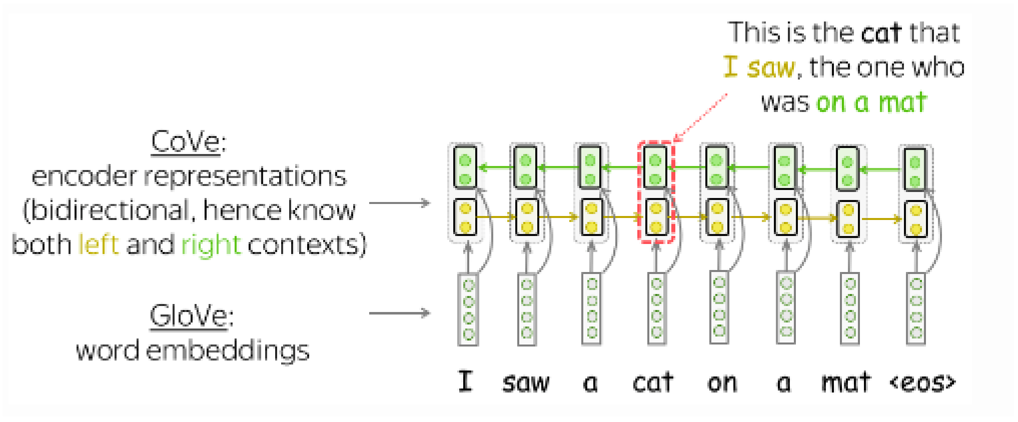
Name: Harshit Gupta

Roll No: 2020114017

ANSWERS TO SECTION 2.1

Question 1: How does ELMo differ from CoVe? Discuss and differentiate both the strategies used to obtain the contextualized representations with equations and illustrations as necessary.

Answer:



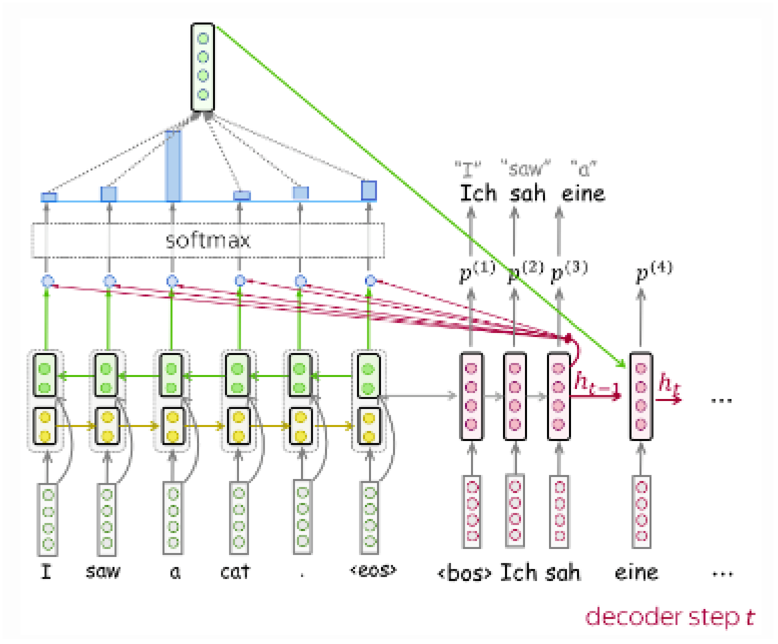
Some of the major differences between CoVe and ELMo are:

1. CoVe requires label data to generate contextual word vectors, whereas ELMo uses an unsupervised approach. Thus, CoVe uses the supervised machine translation dataset, whereas ELMo uses unsupervised language model data.
2. For contextual word representations, CoVe uses only the final layer, whereas ELMo uses numerous layers.
3. The CoVe vectors are fixed during task training, whereas the biLM model weights are fine-tuned with task-specific data before being fixed for task training.
4. CoVe vectors are fixed during task training while the biLM model's weights are adjusted using data specific to the task before being fixed for the task training.

Understanding CoVe better:

1. CoVe uses a deep LSTM encoder from an attentional sequence-to-sequence model trained for the purpose of machine translation (MT) to get Contextualized word vectors (CoVe).

2. The trained encoder outputs CoVe for other tasks.



Math behind it:

Consider a sequqnce of words in source language: $w^x = [w^{x1}, w^{x2}, ..., w^{xn}]$

The sequence of words in the target language are: $w^z = [w^{z1}, w^{z2}, ..., w^{zn}]$

1. Let $GloVe(wx)$ be a sequence of GloVe vectors for the same. This is then fed to a standard, two-layer, bidirectional, long short-term memory network (MT-LSTM). This is used to compute a sequence of hidden states. $h = MT - LSTM(GloVe(w^x))$

2. An attentional decoder is used. It uses a two-layer, unidirectional LSTM to produce a hidden state based on previous target embeddings and a context-adjusted hidden state.

   $h_t^{dec} = LSTM([z_{t-1}; h_{t-1}, h_{t-1}^{dec}])$

3. The decoder then computes a vector of attention weights α representing the relevance of each encoding time-step to current decoder state.

   $\alpha_t = softmax(H(W_1 h_t^{dec} + b_1)$, H = elements of h stacked along time dimension.

4. The decoder then uses these weights as coefficients in an **attentional sum** that is **concatenated** with the decoder state and passed through a **tanh** layer to **form the context-adjusted hidden state**.

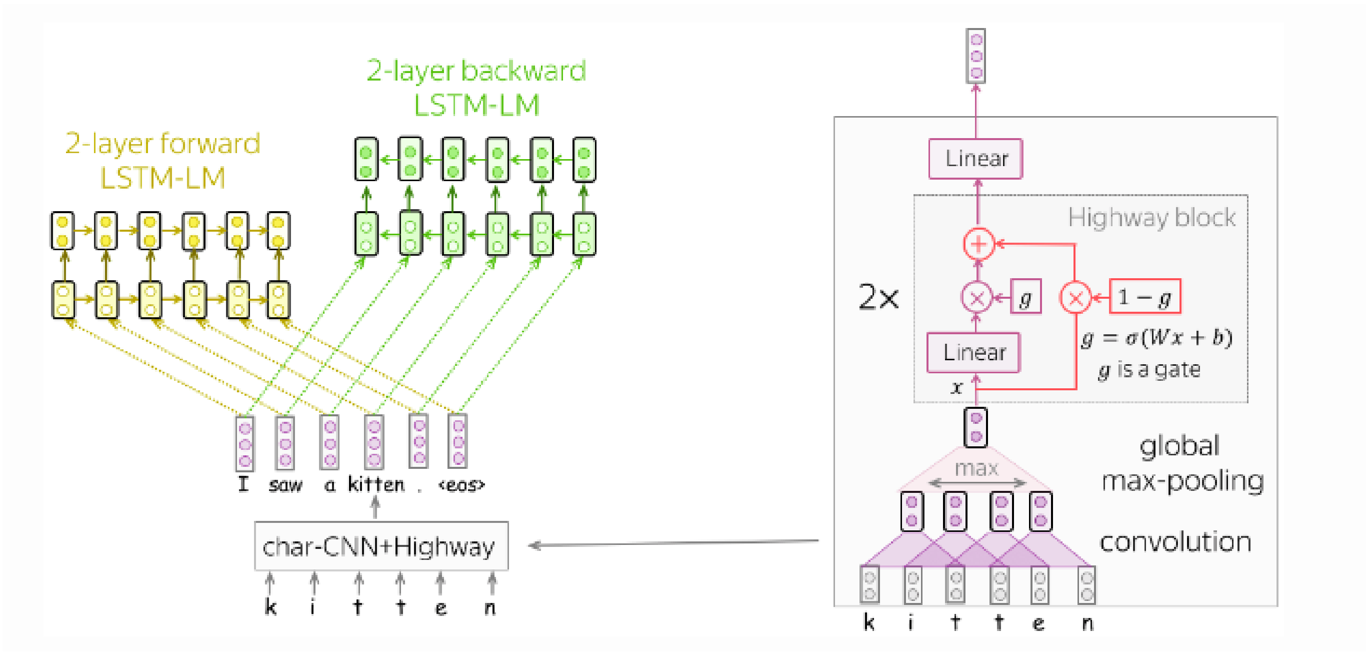
$$\tilde{h_t} = tanh(W_2[H^T \alpha_t; h_t^{dec}] + b_2)$$

5. Finally, we apply softmax to it:

$$p(\hat{w_t^z}|X, w_1^z, ..., w_{t-1}^z) = softmax(W_{out}\tilde{h_t} + b_{out})$$

This is how we get context vectors.

Understanding GloVe better:



The forward LM is a deep LSTM that goes over the sequence from start to end to predict token t_k based on the prefix t_1 to t_{k-1}:

$p(t_k|t_1, ..., t_{k-1}; \theta_x, \overrightarrow{\theta_{LSTM}}, \theta_s)$ Parameters being the θ related symbols.

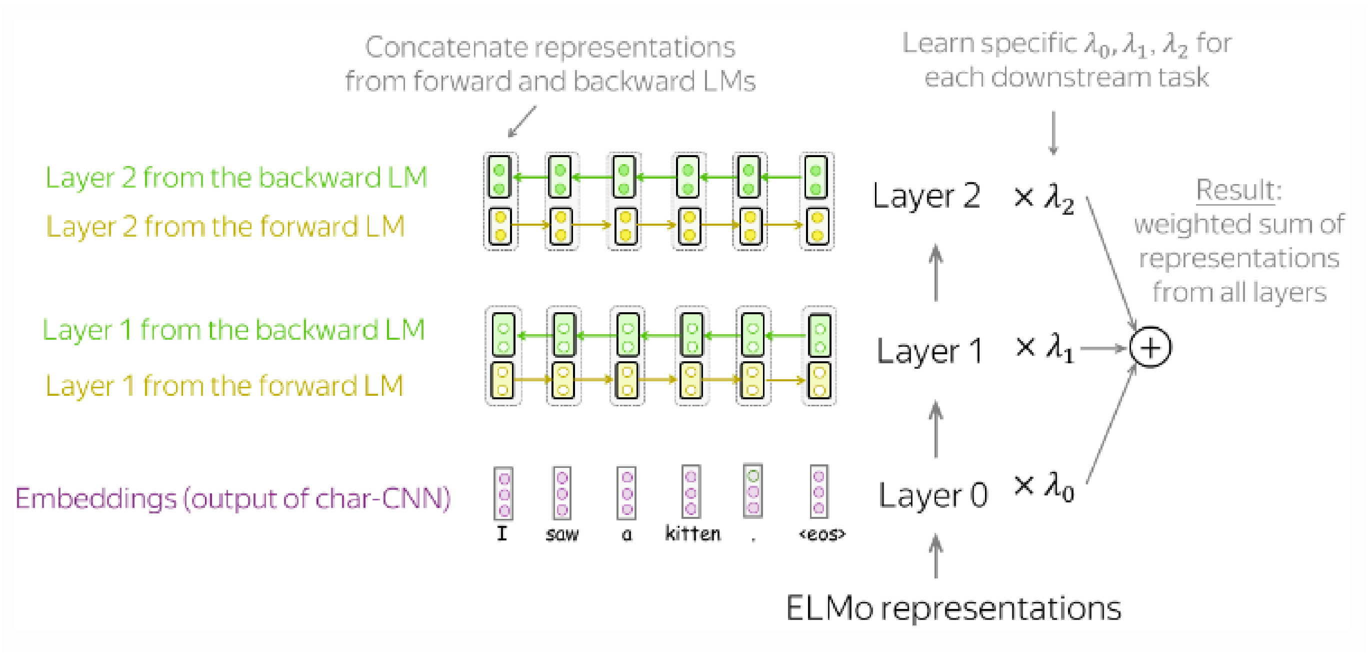
The backward LM is a deep LSTM that goes over the sequence from end to start to predict token t_k based on the suffix $t_{k+1}, \ldots t_N$:

$$p(t_k|t_1, ..., t_{k-1}; \theta_x, \overleftarrow{\theta_{LSTM}}, \theta_s)$$

We then train these LMs jointly, with the same parameters for the token representations and the softmax layer.

ELMo learns softmax weights s_j^{task} to collapse these vectors into a single vector and a task specific scalar γ^{task}.

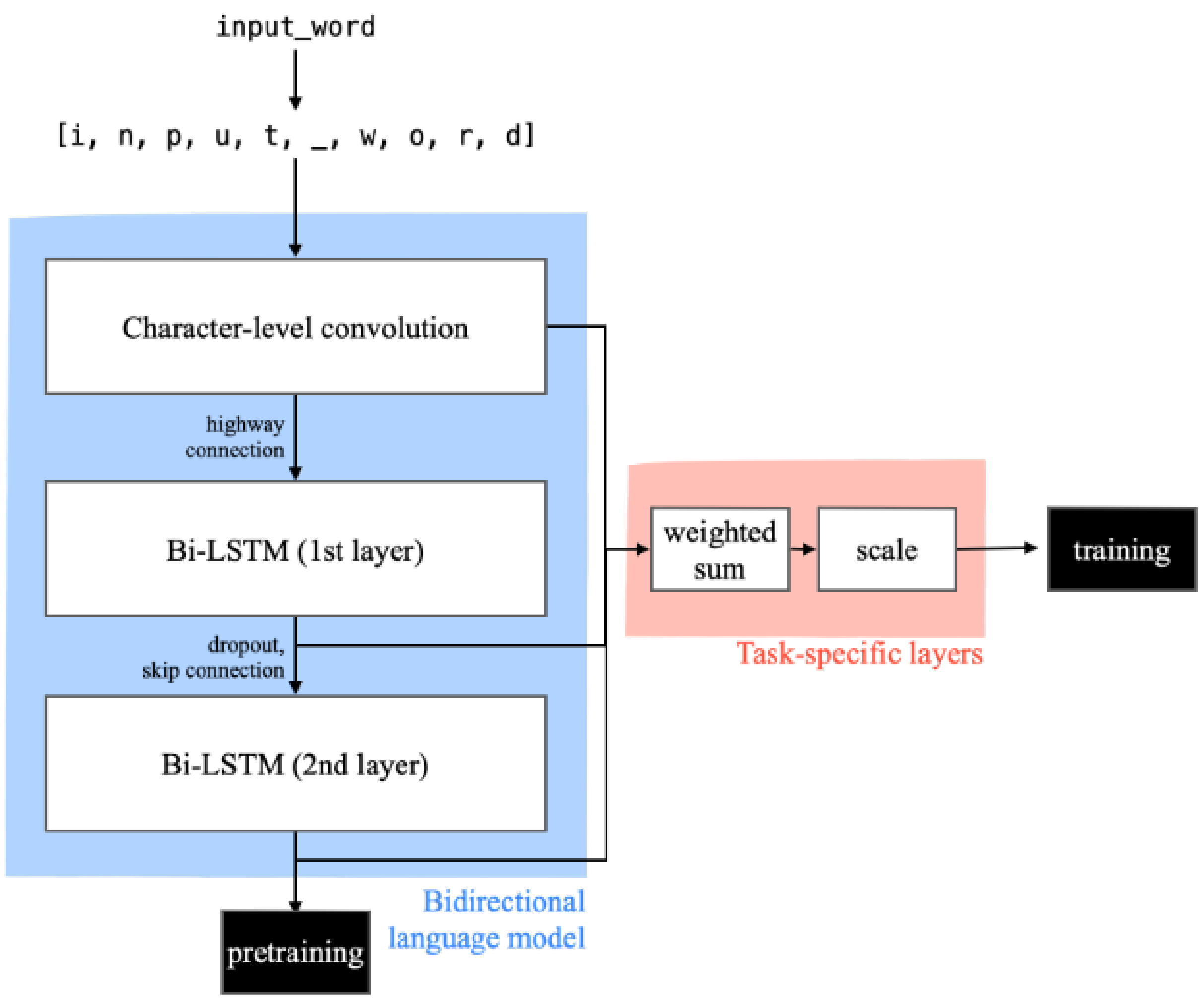
$$ELMo_k^{task} = E(R_k; \theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{k,j}^{LM}$$


Concatenate representations from forward and backward LMs
Learn specific $\lambda_0, \lambda_1, \lambda_2$ for each downstream task
Layer 2 from the backward LM
Layer 2 from the forward LM
Layer 1 from the backward LM
Layer 1 from the forward LM
Embeddings (output of char-CNN)
I
saw
a
kitten
.
<eos>
Layer 2
$\times \lambda_2$
Layer 1
$\times \lambda_1$
Layer 0
$\times \lambda_0$
Result: weighted sum of representations from all layers
ELMo representations

Question 2: The architecture described in the ELMo paper includes a character convolutional layer at its base. Find out more on this, and describe this layer. Why is it used? Is there any alternative to this? [Hint: Developments in word tokenization]

Answer:

The majority of words have vector representations when GloVe Embeddings are taken into account. The GloVe renderings, however, fall short. GloVe just assigns these OOV words some random vector values to handle them. This random assignment would end up confusing our model if it weren't fixed. By analysing the character-level compositions of words, character level embedding uses a one-dimensional convolutional neural network (1D-CNN) to find words' numerical representations.



When we consider the advantage and reason for using character level tokenization over word level, it is to account for a large number of OOV tokens. An alternative to this is sub-word tokenization done in transformer based models.

Reason for advancing from character tokenization: However, character tokenization makes it considerably more difficult for the model to learn useful input representations, even though it is quite simple and would greatly reduce memory and temporal complexity.

Subword tokenization, a cross between word-level and character-level tokenization, is used in transformers models to achieve the best of both worlds.

Subword tokenization methods are based on the idea that uncommon words should be broken down into meaningful subwords rather than frequently used words being divided into smaller subwords. For instance, the word "*annoyingly,*" which may be broken down into the parts "*annoying*" and "*ly*," may be rare. The stand-alone subwords "*annoying*" and "*ly*" would appear more frequently, while the meaning of "*annoyingly*" is maintained by the combined meaning of "*annoying*" and "*ly*".

Subword tokenization enables the model to acquire meaningful context-independent representations while maintaining a manageable vocabulary size. Additionally, by breaking down words into their constituent subwords, subword tokenization enables the model to process words it has never encountered before. *Example:*

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
tokenizer.tokenize("I have a new GPU!")
["i", "have", "a", "new", "gp", "##u", "!"]
```

`"##"` means that the rest of the token should be attached to the previous one, without space (for decoding or reversal of the tokenization).

Dataset Handling:

Tokenizer Used: `from torchtect.data import get_tokenizer('basic_english')`

Vocabulary Size: `44816`

Word Embeddings Used: `GloVe dim = 100`

Batch Size: `8` (Smaller batch size ensures that validation is done properly and account for memory issues.)

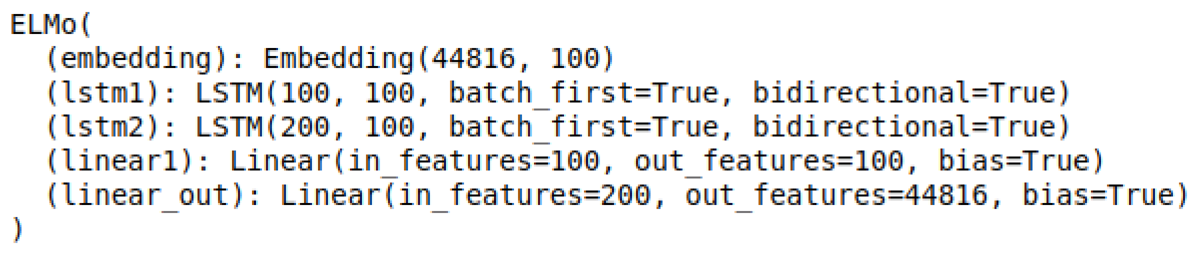
Pretraining Process:

We create an ELMo Model with the following hyperparameters:

1. Loss Function: `nn.CrossEntropyLoss()`
2. No of epochs: `4`
3. Learning Rate: `0.001`
4. Optimizer: `Adam Optimizer`

We truncate the sentences at the length of `100`. This is because the longest sentence is 1203 tokens long which cause memory issues. Smaller sentences are padded and longer sentences are truncated before that.

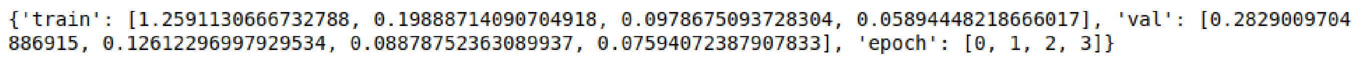
The figure below whos the model architecture for the task.



```
ELMo(
  (embedding): Embedding(44816, 100)
  (lstm1): LSTM(100, 100, batch_first=True, bidirectional=True)
  (lstm2): LSTM(200, 100, batch_first=True, bidirectional=True)
  (linear1): Linear(in_features=100, out_features=100, bias=True)
  (linear_out): Linear(in_features=200, out_features=44816, bias=True)
)
```

The figure below shows the Training Losses, Validation Losses for the corresponding epochs.

NOTE: We save the model with the least validation loss.

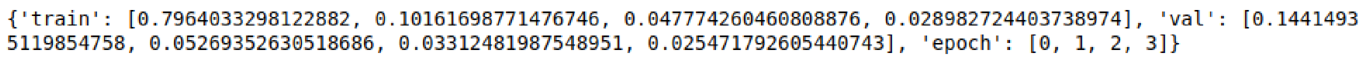


```
{'train': [1.2591130666732788, 0.19888714090704918, 0.0978675093728304, 0.05894448218666017], 'val': [0.2829009704
886915, 0.12612296997929534, 0.08878752363089937, 0.07594072387907833], 'epoch': [0, 1, 2, 3]}
```

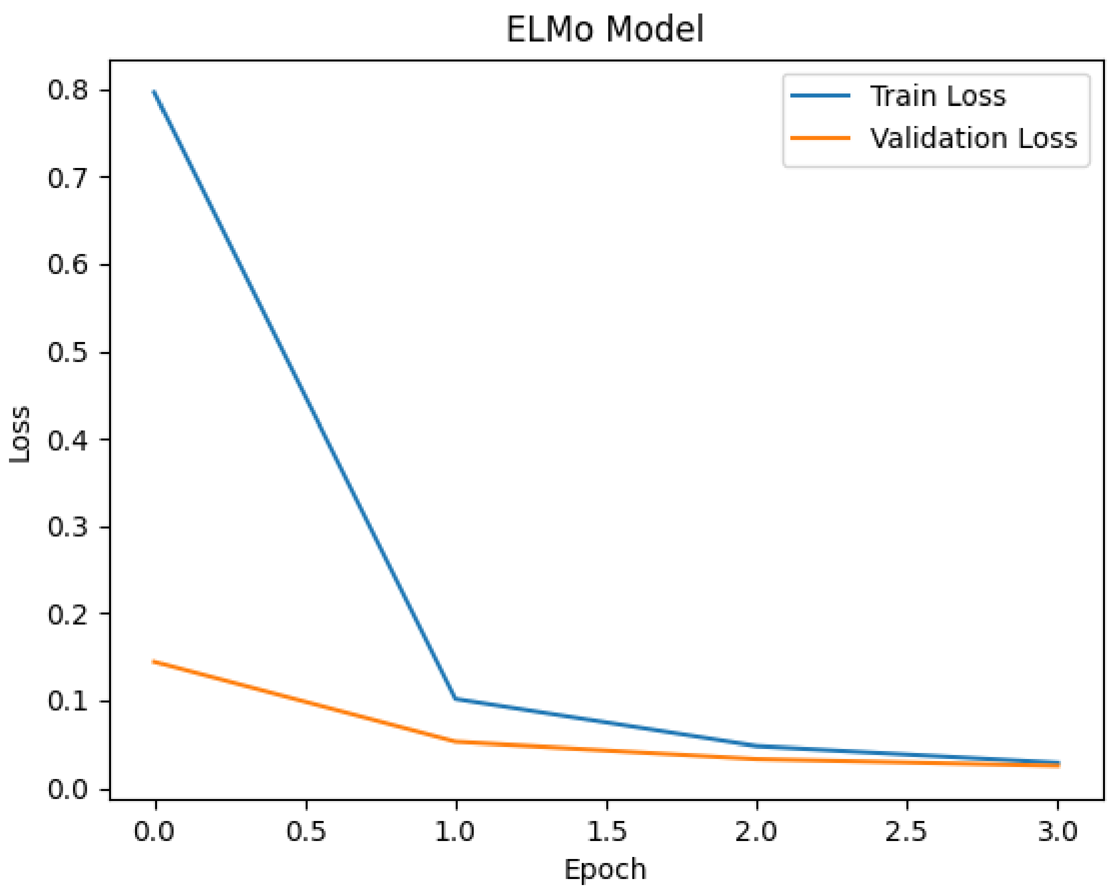
We see that both the training losses and validation losses keep

Suppose we change the sentence length to `250`. The model takes much longer to train due to the increase in dimensions of the input data.

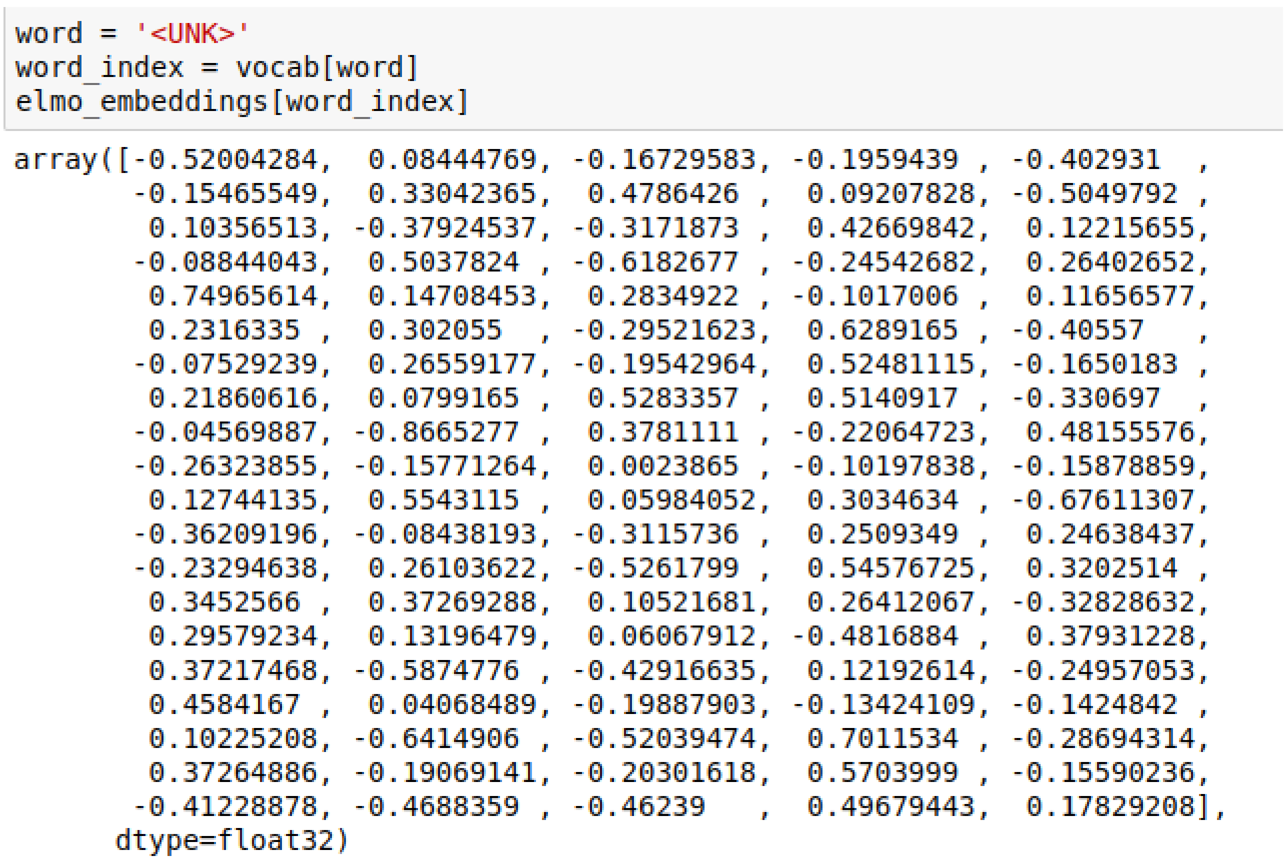
Here we see that the losses can be shown in these images:



```
{'train': [0.7964033298122882, 0.10161698771476746, 0.047774260460808876, 0.028982724403738974], 'val': [0.1441493
5119854758, 0.05269352630518686, 0.03312481987548951, 0.025471792605440743], 'epoch': [0, 1, 2, 3]}
```



We stop by the 4^{th} epoch since it doesnt make much of a difference in model performance. In order to see that word embeddings are being generated, we analyse embeddings for the `<UNK>` token. Standard GloVe Embeddings give us a matrix of only 0's. However, after pretraining we notice that the Embeddings for the same token are different as seen in the figure below.



```
word = '<UNK>'
word_index = vocab[word]
elmo_embeddings[word_index]
```

```
array([-0.52004284,  0.08444769, -0.16729583, -0.1959439 , -0.402931  ,
       -0.15465549,  0.33042365,  0.4786426 ,  0.09207828, -0.5049792 ,
        0.10356513, -0.37924537, -0.3171873 ,  0.42669842,  0.12215655,
       -0.08844043,  0.5037824 , -0.6182677 , -0.24542682,  0.26402652,
        0.74965614,  0.14708453,  0.2834922 , -0.1017006 ,  0.11656577,
        0.2316335 ,  0.302055  , -0.29521623,  0.6289165 , -0.40557   ,
       -0.07529239,  0.26559177, -0.19542964,  0.52481115, -0.1650183 ,
        0.21860616,  0.0799165 ,  0.5283357 ,  0.5140917 , -0.330697  ,
       -0.04569887, -0.8665277 ,  0.3781111 , -0.22064723,  0.48155576,
       -0.26323855, -0.15771264,  0.0023865 , -0.10197838, -0.15878859,
        0.12744135,  0.5543115 ,  0.05984052,  0.3034634 , -0.67611307,
       -0.36209196, -0.08438193, -0.3115736 ,  0.2509349 ,  0.24638437,
       -0.23294638,  0.26103622, -0.5261799 ,  0.54576725,  0.3202514 ,
        0.3452566 ,  0.37269288,  0.10521681,  0.26412067, -0.32828632,
        0.29579234,  0.13196479,  0.06067912, -0.4816884 ,  0.37931228,
        0.37217468, -0.5874776 , -0.42916635,  0.12192614, -0.24957053,
        0.4584167 ,  0.04068489, -0.19887903, -0.13424109, -0.1424842 ,
        0.10225208, -0.6414906 , -0.52039474,  0.7011534 , -0.28694314,
        0.37264886, -0.19069141, -0.20301618,  0.5703999 , -0.15590236,
       -0.41228878, -0.4688359 , -0.46239   ,  0.49679443,  0.17829208],
      dtype=float32)
```

Furthermore, we get lower losses for sentence length `250` in comparison to sentence length of `100` due to more context and data being given to train on.

DownStream Task:

We create a sentence classification model using the following parameters.

The model is named `SenClass`. The model was made by getting the lstm models, `lstm1` and `lstm2` from the `ELMo Model` and using it to get hidden layers for the downstream task. We get the pretrained word embeddings using the embeddings from the pretrained model. The command used for the same is:

```
elmo_embeddings = list(elmo.parameters())[0].to(device)
```

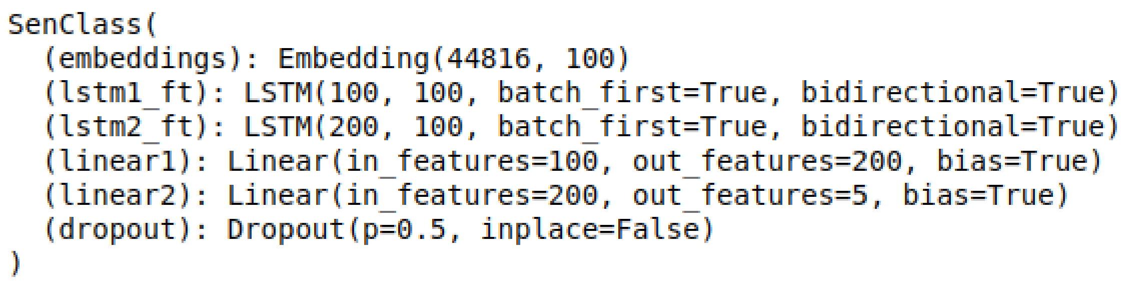
We then use `nn.Parameters()` to store α, β, γ which are the trainable parameters for weighing in the embeddings layer and the hidden layers from the lstm models.

We change them to experiment and allow them to update themselves in order to get the embeddings representations. This can be seen in the command:

```
(self.weights[0]*hidden1 + self.weights[1]*hidden2 +
self.weights[2]*embeds_change)/(self.weights[0]+self.weights[1]+self.weights[2])
```

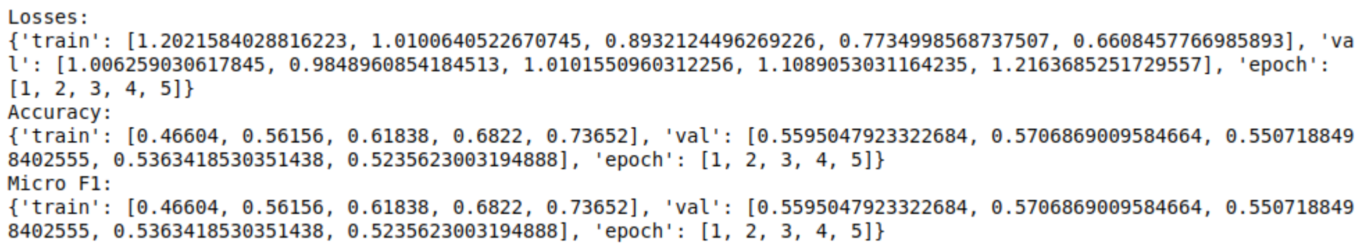
We then apply `max_pooling` to the representations as well as dropout of `Dropout(0.5)`. The reason we apply dropout is because while the training loss was decreasing, the validation loss was increasing which is an indication of overfitting on the train data.

The model architecture of the sentence classification model is:

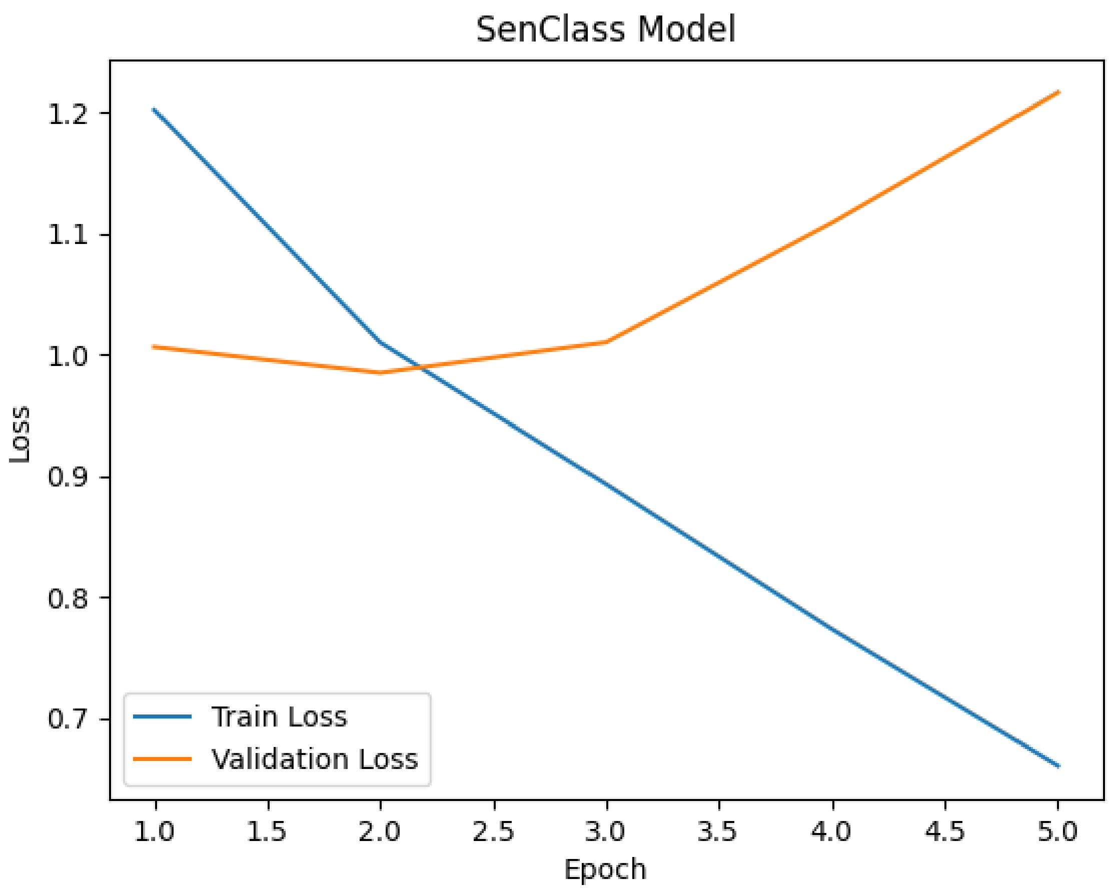


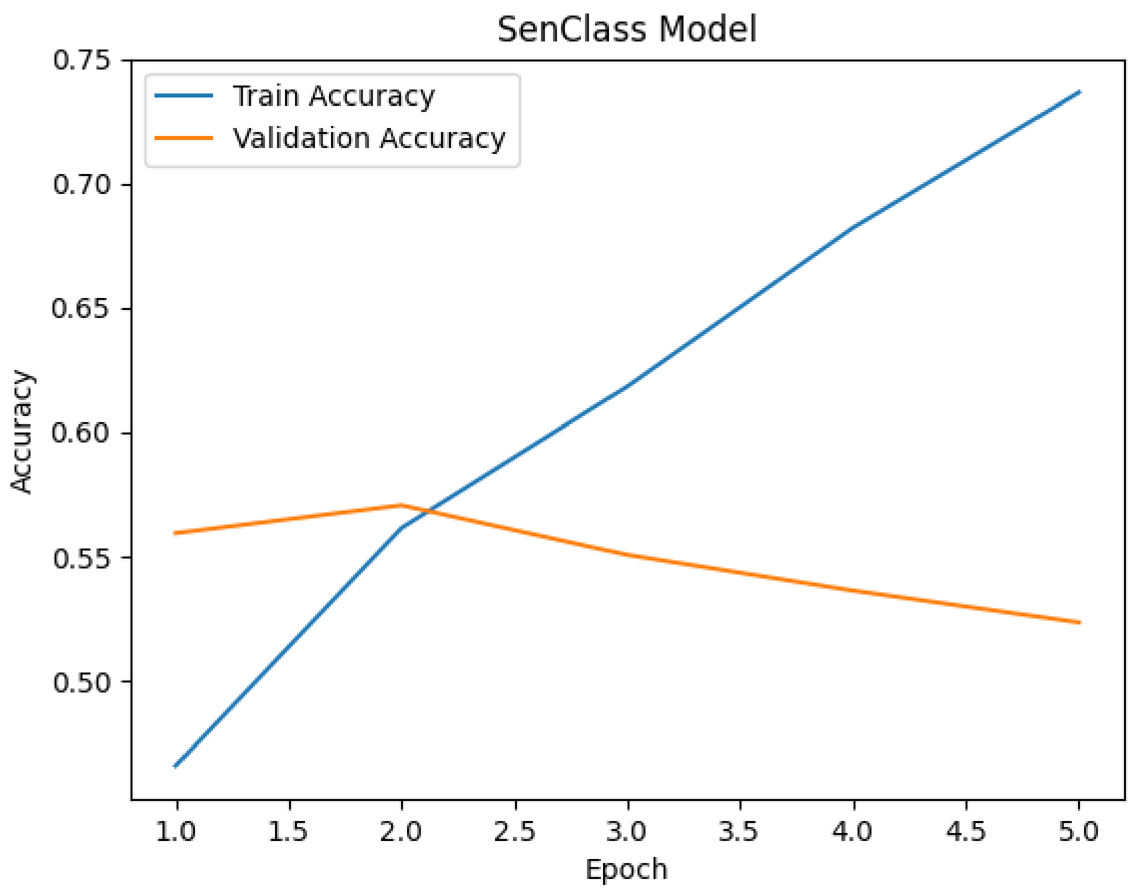
```
SenClass(
  (embeddings): Embedding(44816, 100)
  (lstm1_ft): LSTM(100, 100, batch_first=True, bidirectional=True)
  (lstm2_ft): LSTM(200, 100, batch_first=True, bidirectional=True)
  (linear1): Linear(in_features=100, out_features=200, bias=True)
  (linear2): Linear(in_features=200, out_features=5, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
)
```

After training over 5 epochs, we get the primary metrics as:



```
Losses:
{'train': [1.2021584028816223, 1.0100640522670745, 0.8932124496269226, 0.7734998568737507, 0.6608457766985893], 'va
l': [1.006259030617845, 0.9848960854184513, 1.0101550960312256, 1.1089053031164235, 1.2163685251729557], 'epoch':
[1, 2, 3, 4, 5]}
Accuracy:
{'train': [0.46604, 0.56156, 0.61838, 0.6822, 0.73652], 'val': [0.5595047923322684, 0.5706869009584664, 0.550718849
8402555, 0.5363418530351438, 0.5235623003194888], 'epoch': [1, 2, 3, 4, 5]}
Micro F1:
{'train': [0.46604, 0.56156, 0.61838, 0.6822, 0.73652], 'val': [0.5595047923322684, 0.5706869009584664, 0.550718849
8402555, 0.5363418530351438, 0.5235623003194888], 'epoch': [1, 2, 3, 4, 5]}
```


SenClass Model
Loss
Epoch
1.2
1.1
1.0
0.9
0.8
0.7
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Train Loss
Validation Loss



Thus, the Epoch `2` gives the most optimal validation accuracy. Hence, we load model parameters at epoch `2` and perform the sentence classification over the test data. When we run predictions over the test data, we get:

```
F1 Micro:
0.4112
Accuracy:
0.4112
Confusion Matrix:
[[172 236 197 189 206]
 [175 221 223 169 212]
 [183 224 220 173 200]
 [178 226 196 190 210]
 [153 246 205 171 225]]
```

We notice that while the tranining loss decreases, the validation loss increases after Epoch 2. This is due to overfitting on the training data. We add dropout, however, it does not give any better results. Adding more data points for training and validation can help the model to prevent this.

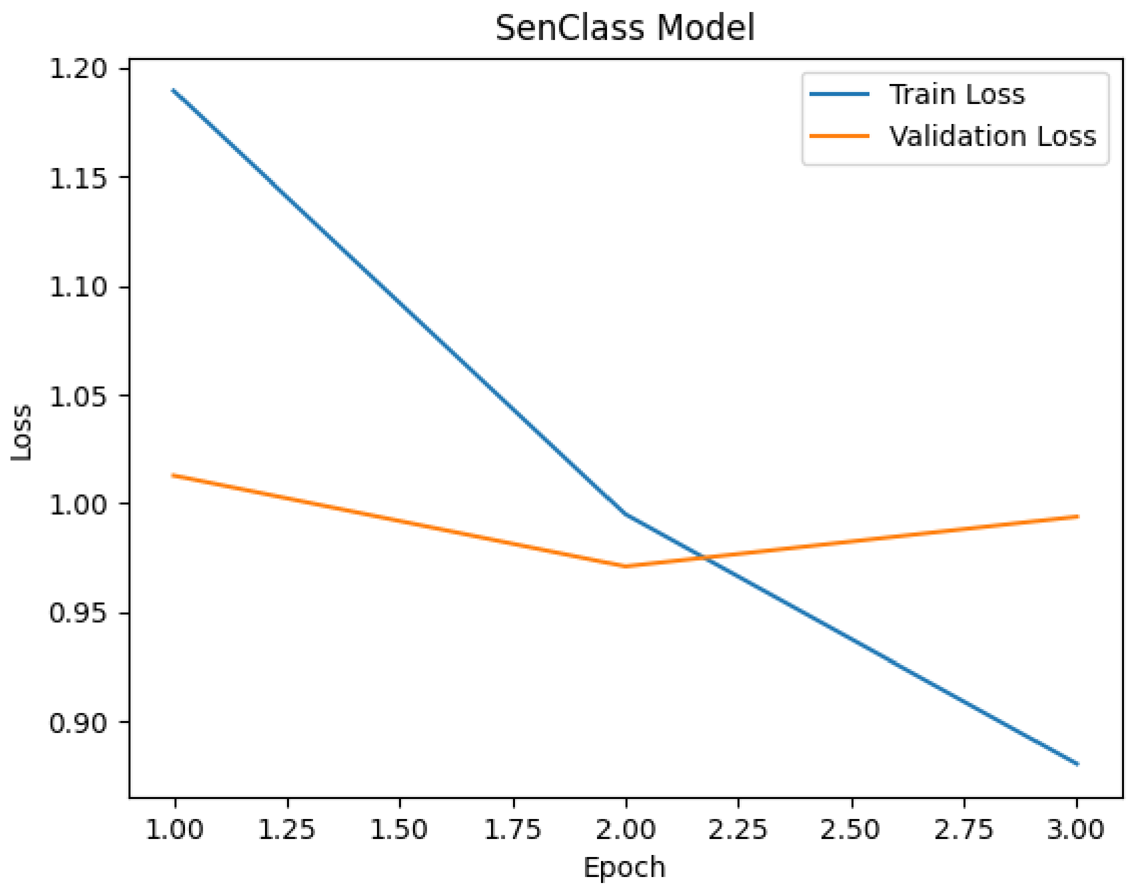
Bonus Task:

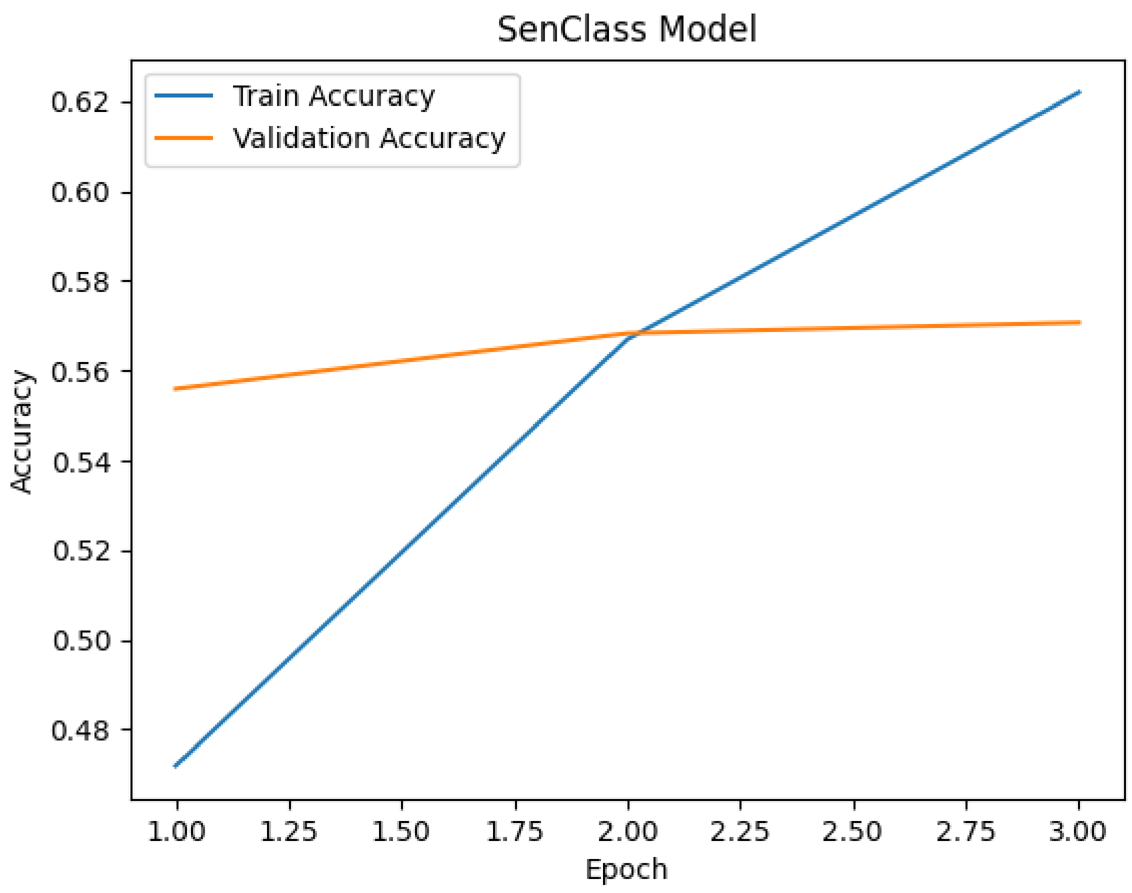
BATCH SIZE: 8

Configuration 1:

We set α, β, γ as `0.9, 0.9, 0.9`. We get the following Losses and Accuracy graphs:

```
Losses:
{'train': [1.1894300786733627, 0.9949418982505799, 0.880409590251445 7], 'val': [1.0127189775434926, 0.9709633030830
481, 0.9938064203285181], 'epoch': [1, 2, 3]}
Accuracy:
{'train': [0.47198, 0.56696, 0.62198], 'val': [0.5559105431309904, 0.5682907348242812, 0.5706869009584664], 'epoc
h': [1, 2, 3]}
Micro F1:
{'train': [0.47198, 0.56696, 0.62198], 'val': [0.5559105431309904, 0.5682907348242812, 0.5706869009584664], 'epoc
h': [1, 2, 3]}
```





Since the weights are traininable, after training, the weights change to:

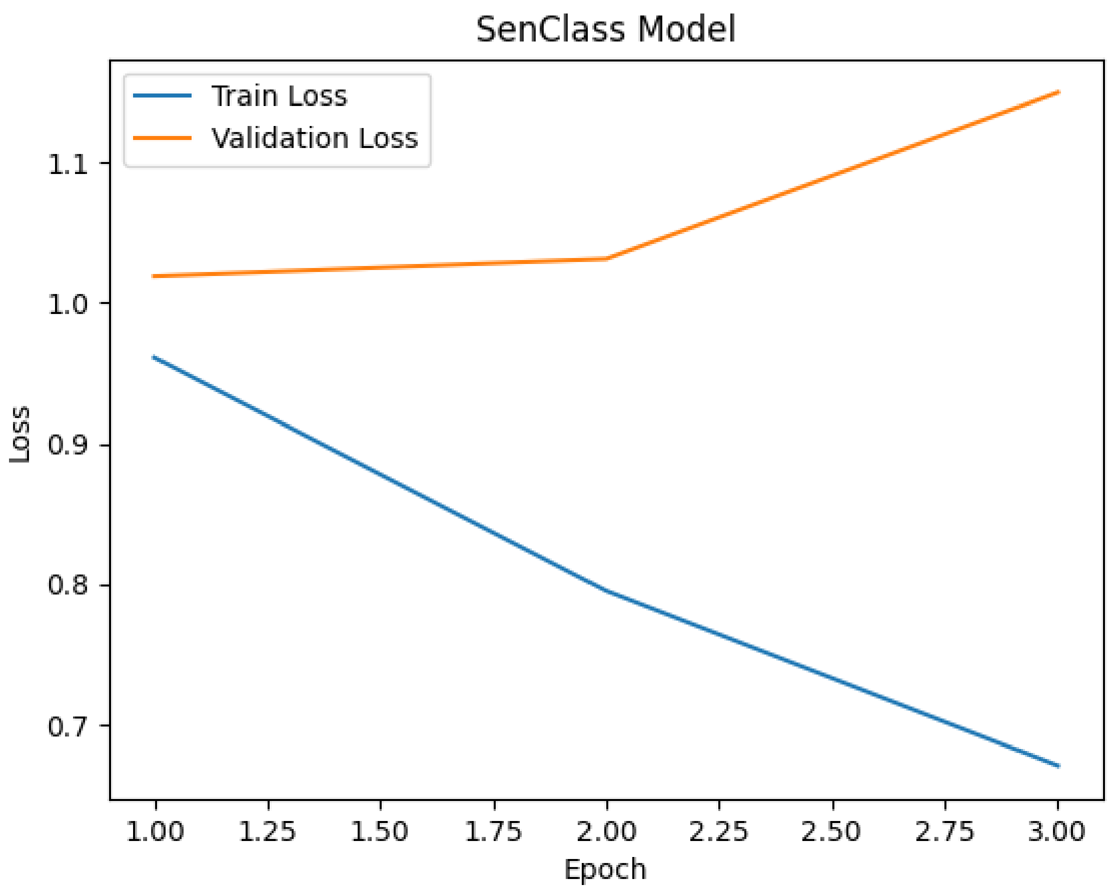
```
weights tensor([1.1851, 0.8163, 0.7646], device='cuda:0') torch.Size([3])
```

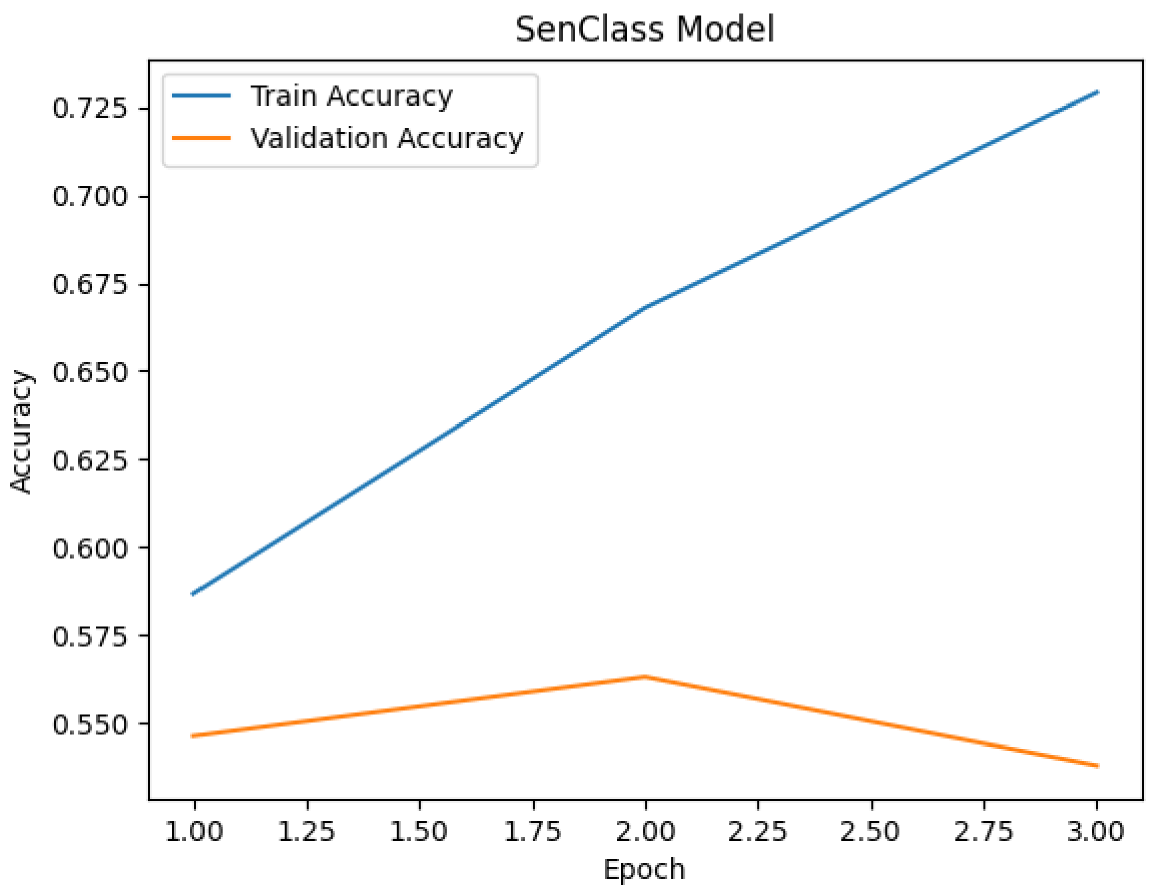
Analysis: For higher values of weights, we observe that the weight changes in favour of using the first lstm more compared to the rest. In fact it depends lesser on pretrained word embeddings.

Configuration 2:

We set α, β, γ as `0.2, 0.2, 0.2`. We get the following Losses and Accuracy graphs:

```
Losses:
{'train': [0.9609358486509323, 0.7951988686609268, 0.6709609821653366], 'val': [1.018830865383529, 1.03118213658896
5, 1.1494162891048212], 'epoch': [1, 2, 3]}
Accuracy:
{'train': [0.58678, 0.66798, 0.72914], 'val': [0.5463258785942492, 0.5630990415335463, 0.5379392971246006], 'epoc
h': [1, 2, 3]}
Micro F1:
{'train': [0.58678, 0.66798, 0.72914], 'val': [0.5463258785942492, 0.5630990415335463, 0.5379392971246006], 'epoc
h': [1, 2, 3]}
```


SenClass Model
Train Loss
Validation Loss
Loss
1.1
1.0
0.9
0.8
0.7
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
Epoch



Since the weights are traininable, after training, the weights change to:

```
weights tensor([0.2270, 0.2403, 0.2031], device='cuda:0') torch.Size([3])
```

The accuracy and confusion matrix on the test data is as follows:

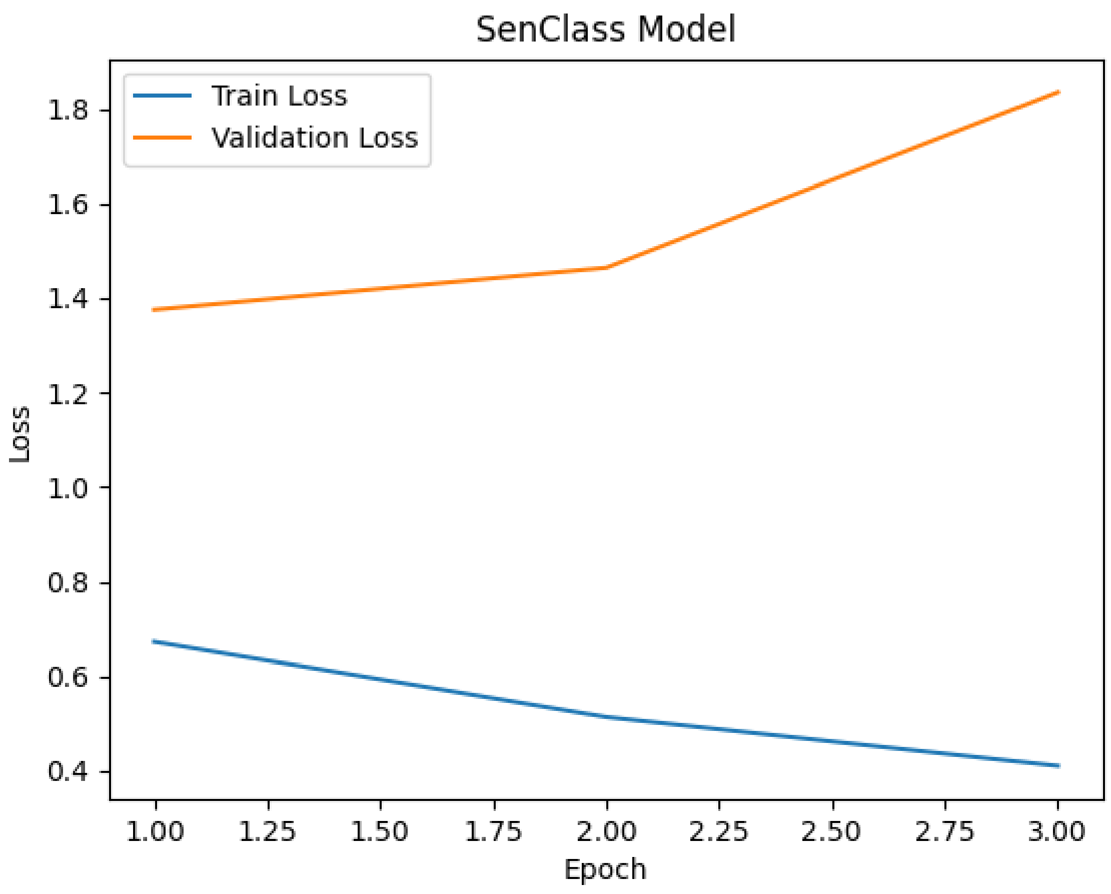
```
0.402
[[212 271 162 132 223]
 [171 273 164 153 239]
 [203 274 168 148 207]
 [202 248 179 136 235]
 [187 248 206 143 216]]
```

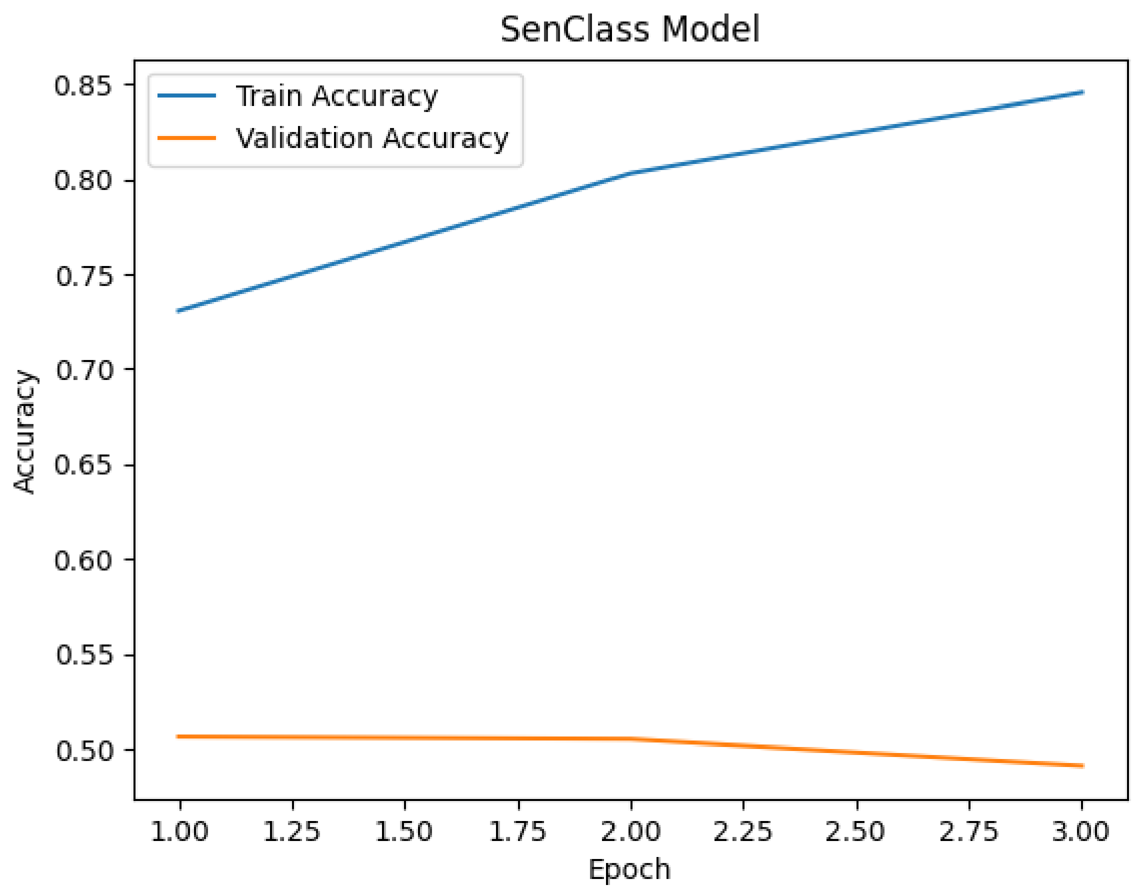
Analysis: When we use lower weights, the weights tend to tune and favour the sconds lstm model compared to others.

Configuration 3:

In this configuration we only train using the second layer of the LSTM Model. We set weights as `0,1,0`. We get:

```
Losses:
{'train': [0.6724043889713287, 0.5138920419788361, 0.4105797599864006], 'val': [1.3752493858337402, 1.4640295216991
643, 1.834415672121534], 'epoch': [1, 2, 3]}
Accuracy:
{'train': [0.73064, 0.8031, 0.8456], 'val': [0.5063694267515924, 0.5051751592356688, 0.4912420382165605], 'epoch':
[1, 2, 3]}
Micro F1:
{'train': [0.73064, 0.8031, 0.8456], 'val': [0.5063694267515924, 0.5051751592356688, 0.4912420382165605], 'epoch':
[1, 2, 3]}
```





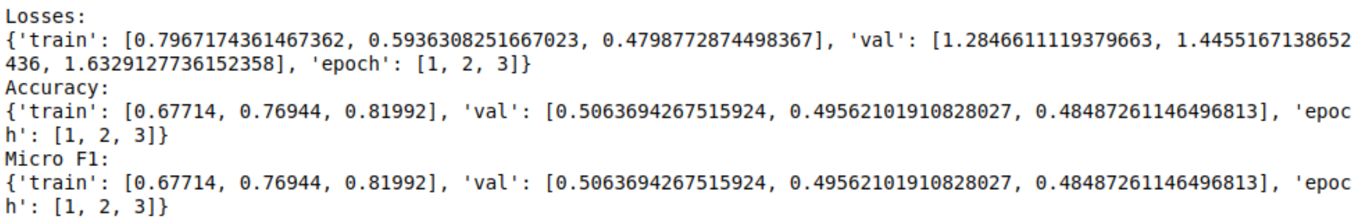
On generating predictions on the test set, we get:

```
F1 Micro:
0.39879999999999993
Accuracy:
0.3988
Confusion Matrix:
[[184 212 194 269 141]
 [177 176 194 269 184]
 [184 197 222 222 175]
 [169 201 219 259 152]
 [172 195 205 272 156]]
```

Compared to other configurations, we notice that just using the second lstm layer for the downstream task does not give the best accuracy or f1-score. This shows that however small, there is imprivement in using other layers as well along with the second lstm layer.

Configuration 4:

We set α, β, γ as `0.33, 0.33, 0.33`. they are fixed weights. We get the following metrics:

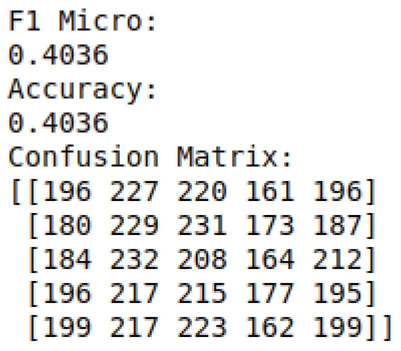


```
Losses:
{'train': [0.7967174361467362, 0.5936308251667023, 0.4798772874498367], 'val': [1.2846611119379663, 1.4455167138652
436, 1.6329127736152358], 'epoch': [1, 2, 3]}
Accuracy:
{'train': [0.67714, 0.76944, 0.81992], 'val': [0.5063694267515924, 0.49562101910828027, 0.48487261146496813], 'epoc
h': [1, 2, 3]}
Micro F1:
{'train': [0.67714, 0.76944, 0.81992], 'val': [0.5063694267515924, 0.49562101910828027, 0.48487261146496813], 'epoc
h': [1, 2, 3]}
```

We notice that the weights change to:

```
weights tensor([0.4606, 0.8395, 0.8514]
```

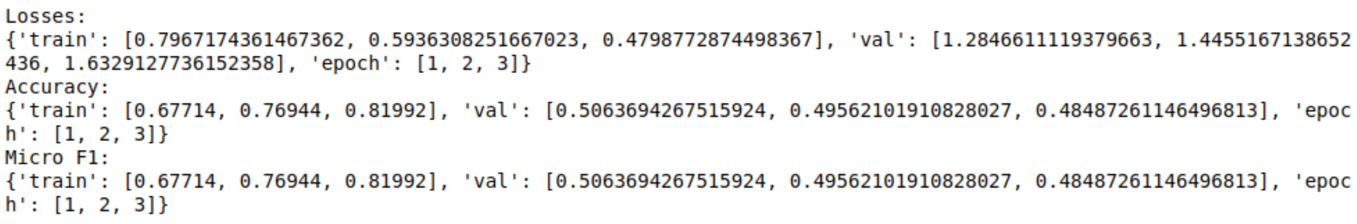
And when run on the test set, we get the following results:



```
F1 Micro:
0.4036
Accuracy:
0.4036
Confusion Matrix:
[[196 227 220 161 196]
 [180 229 231 173 187]
 [184 232 208 164 212]
 [196 217 215 177 195]
 [199 217 223 162 199]]
```

Configuration 5:

We set α, β, γ as `0.33, 0.33, 0.33` but they are NOT fixed weights and then observe the results.

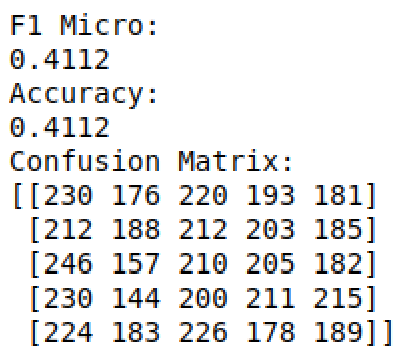


```
Losses:
{'train': [0.7967174361467362, 0.5936308251667023, 0.4798772874498367], 'val': [1.2846611119379663, 1.4455167138652
436, 1.6329127736152358], 'epoch': [1, 2, 3]}
Accuracy:
{'train': [0.67714, 0.76944, 0.81992], 'val': [0.5063694267515924, 0.49562101910828027, 0.48487261146496813], 'epoc
h': [1, 2, 3]}
Micro F1:
{'train': [0.67714, 0.76944, 0.81992], 'val': [0.5063694267515924, 0.49562101910828027, 0.48487261146496813], 'epoc
h': [1, 2, 3]}
```

The weights change to:

```
weights tensor([0.4606, 0.8395, 0.8514]
```

On running on the test data, we get the foloowing results:



```
F1 Micro:
0.4112
Accuracy:
0.4112
Confusion Matrix:
[[230 176 220 193 181]
 [212 188 212 203 185]
 [246 157 210 205 182]
 [230 144 200 211 215]
 [224 183 226 178 189]]
```

Analysis: If we compare having fixed weights to gradient weights, we observe that static weights while hardcoded to a certain configuration provide great results, they do not provide results as accurate compared to allowing them to constantly update themselves.

Currently the best configuration of weights to get **maximum accuracy** and **f1-score** is:

`0.33` $\leq \alpha, \beta, \gamma \leq$ `0.5`.