Share

CS6910 Assignment-3: Recurrent Neural Networks

Done as part of CS6910, Jan-May 2023, under Professor Mitesh Khapra.
Submitted by Dhananjay Balakrishnan, ME19B012.

Dhananjay Balakrishnan

All the code written for this assignment can be found in the following github repository: https://github.com/Dhananjay42/cs6910-assn3.

Question 1:

Refer to the code for the implementation of these.

Implementational Note: We have taken the number of layers in the encoder and the decoder to be equal, instead of 2 separate quantities. We have also used the same embedding size and the same hidden layer size for both the networks. This enables us to easily copy the final hidden state of the encoder into the initial hidden state of the decoder, and also makes hyperparameter tuning so much easier (i.e. by reducing the dimension of the search).

For the calculations: Given, input embedding size = m, number of layers in the encoder and decoder = 1, length of the input and output sequence = T, size of the vocabulary of both the input and output language = V, and the hidden cell state size is k.

a) Total number of computations done by the network.

Encoder:

- Embedding Layer: The embedding layer is of size m x V , it operates on incoming sequences of size V x T, to give an output of dimensions m x T. The input sequences are one-hot encoded, which means that instead of doing a matrix multiplication, we just need to look up the corresponding column of the embedding layer, and is of order $O(1)$ for each sequence in the input. We need to do this for T different sequences, so overall, this operation is of the order $O(T)$.
- We're dealing with a standard RNN cell. The mathematical equations for it are given by:

$$s_i = \sigma(Ux_i + Ws_{i-1} + b)$$

$$y_i = O(Vs_i + c)$$

- In general, when you multiply an a x b matrix, and a b x c matrix, the number of computations = $ac(2b - 1)$, as we need to compute a dot product for each of the ac elements, and each dot product consists of b multiplications and $b - 1$ additions. Therefore, corresponding to Ux_i - we have $k(2m - 1)$ computations; corresponding to Ws_{i-1} - we have $k(2k - 1)$ computations. Finally, due to the bias term, we have an extra k computations, corresponding to the addition.
- Similarly, corresponding to Vs_i, we have $m(2k - 1)$ computations, and an extra m computations, corresponding to the addition of the bias term c.
- Now, for σ, i.e. the sigmoidal layer, let us assume we perform $O(k)$ computations, assuming each sigmoidal computation is of the order $O(1)$. Similarly, for the activation at y_i, we take it to be of order $O(m)$ computations.
- We do this for each timestep and there are T timesteps.

Therefore, the total number of computations corresponding to the encoder is:

$$T(k(2m - 1) + k(2k - 1) + m(2k - 1) + k + m + O(k) + O(m)) + O(T) = T(2k^2 + 4mk - k) + O(Tk) + O(Tm) + O(T)$$

Decoder:

- The embedding layer and the RNN of the decoder will have essentially the same structure, and hence, have the same number of computations as what we calculated for the encoder.
- Finally, we have a linear layer. The equation is given by:

$$out = softmax(V'y_i + b')$$

- Therefore, the number of computations corresponding to Vs_i is $V(2m - 1)$, and corresponding to the bias, we have V extra computations, and a further $O(V)$ computations corresponding to the softmax operation.
- Again, we repeat this for T timesteps.

Total Encoder-Decoder Model:

Therefore, the total number of computations in one forward pass of the encoder-decoder model is:

$$2(T(2k^2 + 4mk - k) + O(Tk) + O(Tm) + O(T)) + T(V(2m - 1) + V + O(V))$$

$$= T(4k^2 + 8mk + 2mV - 2k) + O(Tk) + O(Tm) + O(T) + O(V)$$

Therefore, we have computed the order of the total number of computations for the forward pass of the encoder-decoder model.

b) Total number of parameters in the network:

Encoder:

- Embedding Layer: it is of size m x V.
- Coming to the RNN cells: U is a k x m matrix, V is a m x k matrix, and W is a k x k matrix. b is a k x 1 vector, and c is a m x 1 vector. Therefore, the total number of parameters is $2mk + k^2 + k + m$.

Decoder:

- Both the embedding layer and the RNN cells have the same number of weights as that of the encoder.
- Dense Layer: V' is a V x m matrix, and b' is a V x 1 matrix. Therefore, the total number of parameters is $mV + V$.

Total Encoder-Decoder Model:

Therefore, the total number of parameters in the model is:

$$2(mV) + 2(2mk + k^2 + k + m) + mV + V$$

$$= 2k^2 + 4mk + 2m + 2k + 3mV + V$$

Question 2:

Before we get into the hyperparameters, we will elaborate about what metrics we have used to quantify performance. We have been asked to report the test accuracy - in which a prediction is counted as correct only if it exactly matches with the target, character-to-character. However, on looking at a few sample predictions, I saw that in a lot of cases, the word was somewhat accurately predicted with mistakes that even a human being might end up making.

This is especially the case in a somewhat confusing script like Tamil, where writing Tamil words in English definitely results in the loss of information due to insufficient English consonants being there to accurately represent the various sounds and inflections associated with a language like that. For example, ணா, னா, and ந can only be written in English as 'na' while they are pronounced very differently. I will elaborate more about the language-specific observations we've made later.

Anyways, considering our given problem statement, I felt that reporting the validation accuracy alone might not be a good representation of how well the model has actually learnt. This led me to think whether we could come up with a more logical metric as well, to keep track of our training process -> and that was implemented as a character-wise 'similarity' score, which was based on the edit distance.

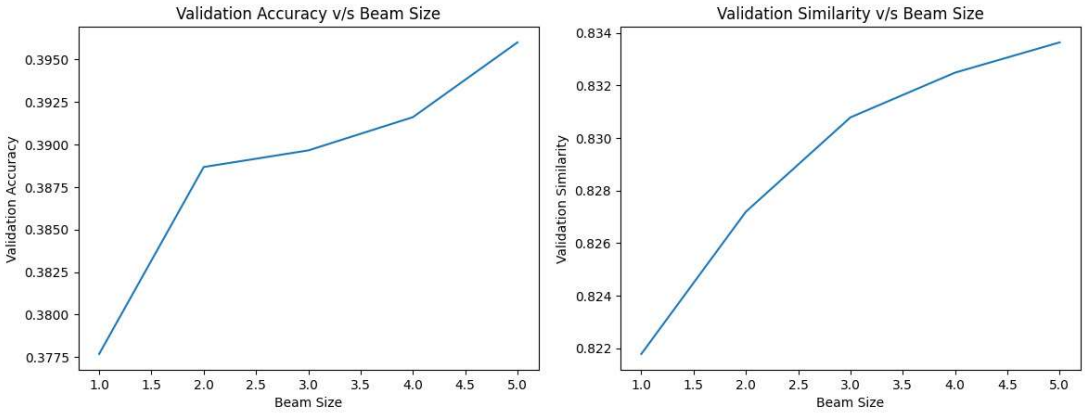
Short Note on the Similarity Metric:

- First, we computed the Levenshtin Distance between 2 words. This is basically the edit distance that tells you how many changes (replacements, additions, and deletions) you need to make in the predicted string to get to the target string. We normalize this by dividing it by the number of the letters of the prediction/target, depending on which one has more characters.
- We compute this score for each word pair and average them to get an average metric. If the predictions are good, the metric will be close to 0, and as the predictions get worse, the metric will get closer to 1.
- We would like to have a linearly increasing scale like accuracy, so we report (1 - above score) as the character-wise similarity on a

particular dataset. We notice from our sweeps that this correlates with the accuracy and the training loss quite well, and also gives us a better idea of how the model is performing.

Regarding Beam Search:

- While we did integrate Beam Search into our codebase, we found that it was very computationally intensive, and hence, if we wanted to get any meaningful results with the limited time and resources we have, it would be difficult to have a beam search step as part of the training process.
- Instead, what we did is run beam search on the validation step, particularly for the vanilla encoder-decoder model with the best set of hyperparameters, the results of which are shown below.
- We notice that increasing the beam size (here 1 is the standard evaluation, by just considering the best prediction), we are guaranteed better predictions - as reflected in the validation accuracy (which goes up by over 4.85%), and the validation similarity (which increases by over 1.45%). However, this is at the cost of increased computational time. While the evaluation for the validation data takes over a minute normally, with beam size = 5, it takes over 20 minutes. (On average, the time taken is higher by an order of B^2) where B is the beam size.
- An interesting observation after looking at the various alternate predictions is the model is often confused between different types of 'na's (there are 3 in Tamil, as elaborated later), and 'la's (there are 2). This is quite interesting to note, as this implies the model has learnt a somewhat right mapping between the source language (English) to the target language (Tamil).



Hyperparameter Tuning:

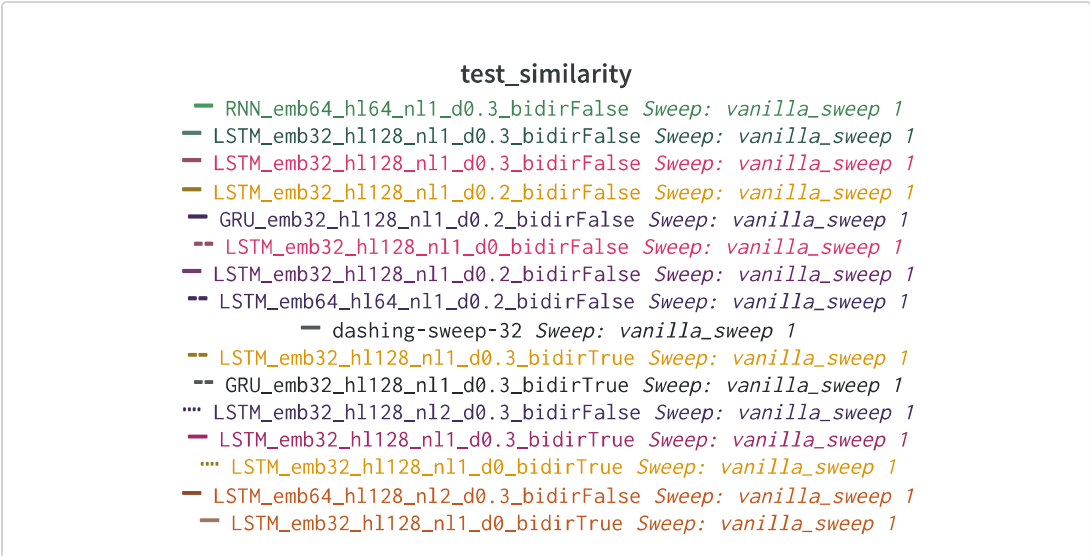
Implementational Note: We ran each model for 75k iterations. This was inspired by the training process used in the seq2seq model documentation on the pytorch website. Each iteration is performed on one word pair, and these 75k word pairs are randomly chosen from the train data with replacement. While this does not guarantee us using all the words in the training dataset, it does randomize the process - making the model sufficiently robust, and on average - training it over a large variety of training samples while occasionally repeating other samples.

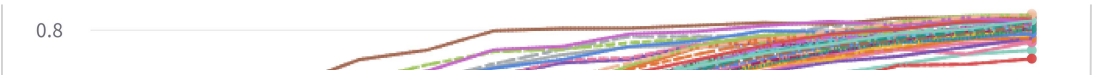
The hyperparameters and the values we swept over were:

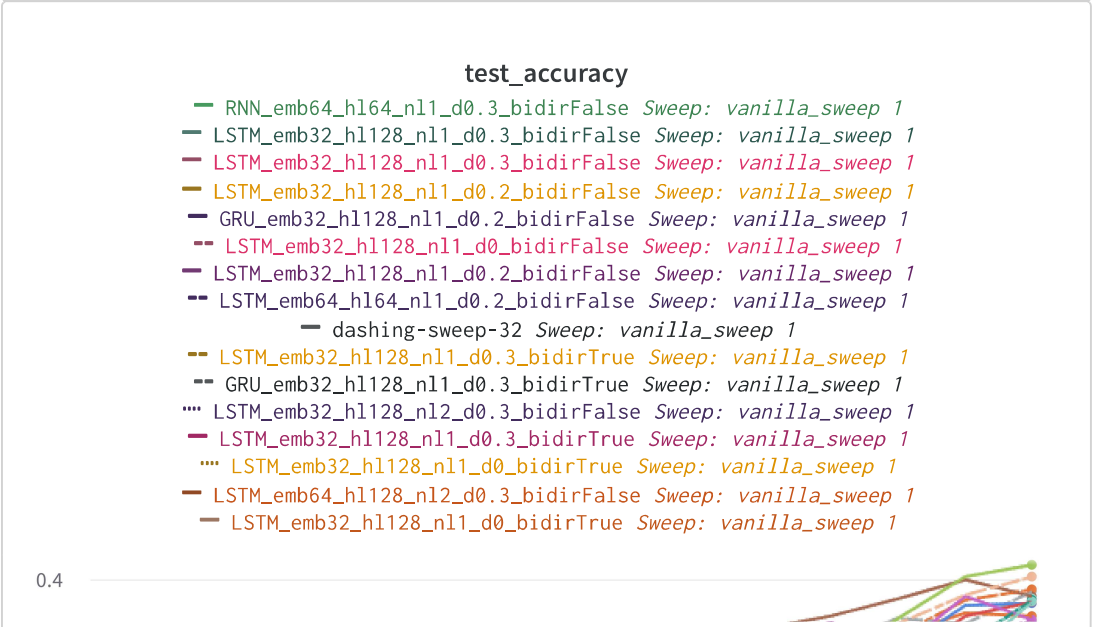
```
hyperparameters = {
        "embedding_size": {
            'values': [32, 64, 128]
        },
        "n_layers": {
            'values' : [1, 2, 3]
        },
        "hl_size": {
            'values' : [32, 64, 128]
        },
        "bidir": {
            'values': [True, False]
        },
        "dropout":{
          'values': [0, 0.2, 0.3]
        },
        "cell_type": {
            'values' : ['RNN', 'GRU', 'LSTM']
        }
    }
```

You can see the various hyperparameter values we've swept over in this image.

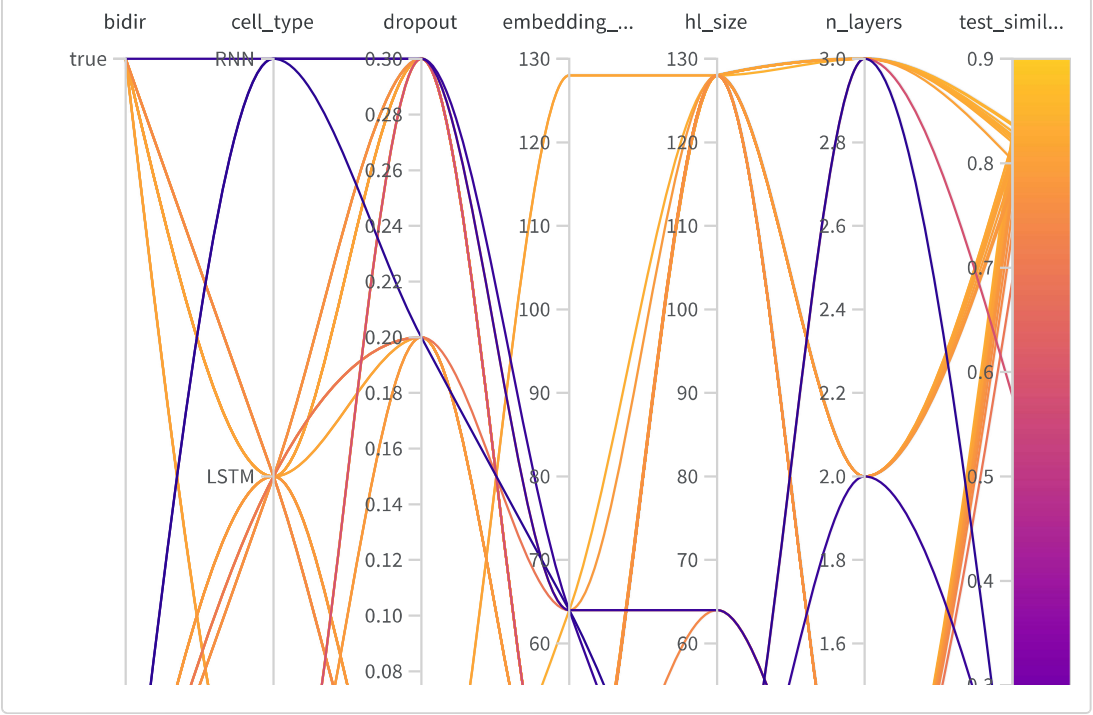
As we have a large number of possible hyperparameter combinations, it does not make sense to use something like a grid-search. So, for this, we used a Bayesian sweep using the aforementioned character-wise similarity as a metric. We have pasted the various plots from the sweep below.




0.8


test_accuracy
RNN_emb64_hl64_nl1_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
GRU_emb32_hl128_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb64_hl64_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
dashing-sweep-32 Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirTrue Sweep: vanilla_sweep 1
GRU_emb32_hl128_nl1_d0.3_bidirTrue Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl2_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirTrue Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0_bidirTrue Sweep: vanilla_sweep 1
LSTM_emb64_hl128_nl2_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0_bidirTrue Sweep: vanilla_sweep 1
0.4


train_loss
RNN_emb64_hl64_nl1_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
GRU_emb32_hl128_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb64_hl64_nl1_d0.2_bidirFalse Sweep: vanilla_sweep 1
dashing-sweep-32 Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirTrue Sweep: vanilla_sweep 1
GRU_emb32_hl128_nl1_d0.3_bidirTrue Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl2_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0.3_bidirTrue Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0_bidirTrue Sweep: vanilla_sweep 1
LSTM_emb64_hl128_nl2_d0.3_bidirFalse Sweep: vanilla_sweep 1
LSTM_emb32_hl128_nl1_d0_bidirTrue Sweep: vanilla_sweep 1
2.5



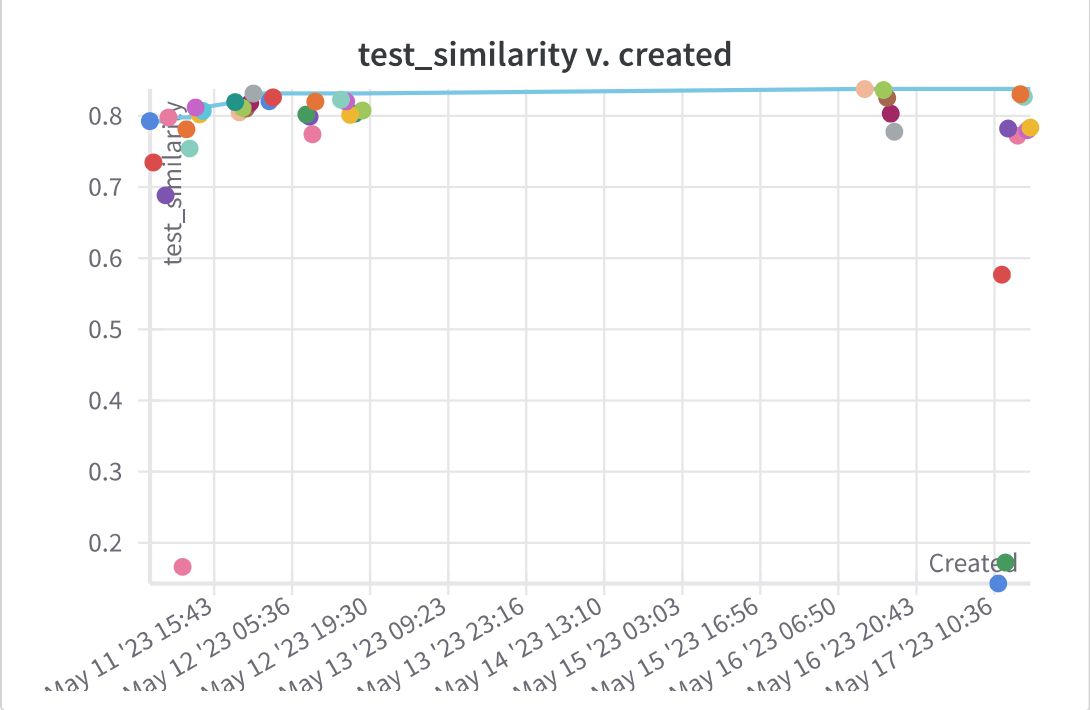
Parameter importance with respect to test_similarity

Search

Parameters

1-9 of 9

Config parameter	Importance	Correlation
cell_type.value_RNN		
hl_size		
Runtime		
n_layers		
bidir		
cell_type.value_LSTM		



Note regarding Bayesian Sweep and further Hyperparameter tuning:

As I performed this sweep, I noticed that often the same combination of hyperparameters would be repeated. On checking the documentation, I learnt that this was a pitfall associated with doing a Bayesian sweep, and however by that time, I was too far along the sweep process that it wouldn't have made sense for me to redo the whole process. Going forward, one will need to take extra precautions to avoid such problems. Also, due to this, all the hyperparameters have not been varied amply, and so we do not have the best correlations between the hyperparameters and the evaluated similarity metrics, however, we've made use of whatever information we could gather from our experiments.

Also, we have performed somewhat of a "mini-sweep" for the attention part of the assignment, where we varied a small number of hyperparameters, enabling us to do a comprehensive grid search. In fact going forward, it would be a smart thing to do comprehensive grid searches on smaller subsets of the hyperparameters. That was

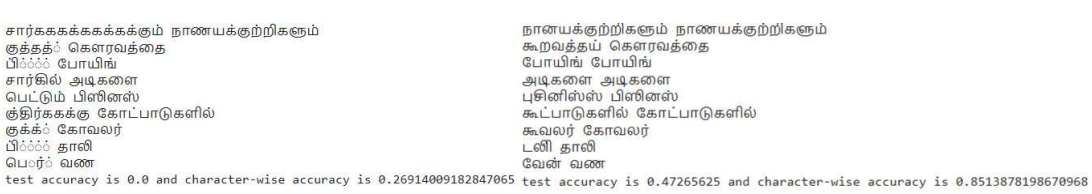
an oversight by me while performing these experiments, and I will make sure to not let it happen again.

Question 3:

- From the above sweep, both the GRU and the LSTM models are much superior in their performance, and both are more or less equally good. The RNN models on the other hand, simply fail to converge.
- It should also be noted that, on average, GRU and LSTM models take significantly more time for one forward/backward step than RNN models, and LSTM models are marginally slower than GRU. We also notice that as the number of hidden layers increase, or as the number of neurons per hidden layer increase - the time for training increases significantly, ranging from 30-ish minutes for simpler models to 45-ish minutes for more complex models.
- There is a good positive correlation between the hidden layer size and the test similarity score. Therefore, increasing the hidden layer size should improve the model performance.
- Although you would expect a good correlation to exist between the number of layers and the model performance, we don't really observe anything of the sort.
- The reported correlation between the embedding size and the model performance is negative, implying that increasing embedding size is detrimental to the model performance. However, on inspecting the sweep data we have, we see that the RNN models were the only ones with high embedding sizes (by chance), and as their performance was very poor, they are less likely to get picked (Bayesian search), and hence, we don't have enough data to conclude. So, based on the existing literature and domain knowledge, we will go ahead and roughly assume that a higher embedding size will give marginally better results; although it isn't expected to change the performances by too much.
- Bi-directionality is seen to consistently improve the model performance. However, this is at the cost of slowing down the

training process.

- We don't see a very strong correlation between the dropout value and the model performance, so going forward, we'll go with the median value (0.2), just to make sure our models don't overfit. This will definitely help when we are dealing with extremely complex models, like when we integrate attention into the vanilla encoder-decoder model.
- Initially, on using a learning rate of 0.001, I noticed that the model barely learnt anything. I had to spike the learning rate up to 0.01 for there to actually be any learning. Otherwise, the learning was too slow.
- Increasing the beam size guarantees an improvement in the performance, at the cost of it being more computationally expensive.
- This isn't an observation about the hyperparameters per se, but just something interesting about the training process. It was very interest to see the quality of the predictions of the model evolve over time. If you look at the predictions initially, you see that they just don't make sense semantically. In the image given below, you can see how predictions have evolved with time. However, it's not just the correctness of the predictions that have improved, it is the underlying semantics that have improved. For example, the set of predictions on the left has a lot of words that well, don't make sense, for instance - you can't have an accent (a maatra in Hindi) or a contractor without there being an actual letter, but these predictions have that. However, with time, while the predictions get better, the model also learns the underlying semantics associated with the language. That is something I found very interesting and worth discussing.



On the left, you see the predictions initially (after 5k iterations), and on the right, you see the predictions finally (after 75k iterations).

Question 4:

a) Best Model

The best model that we obtain from the vanilla sweep has the following hyperparameters:

```
best_params = {
    'cell_type':'LSTM', 'dropout':0, 'embedding_size':128, 'hl_size':128, 'n_layers':3, 'bidirectional_flag':False
}
```

Best Hyperparameters for the Vanilla Model

This model has a validation accuracy of 0.4177 and a validation similarity of 0.8362.

On the test dataset, it has a test accuracy of 0.320556640625 and a test similarity score of 0.7873471098288715.

Based on the disparity between this scores, we can conclude that we might be overfitting, and improving regularization might help improve the model performance.

b and c) Sample Predictions and Observations

- We want to understand where the model is making mistakes and where the model is performing well. To do this, I computed the Levenshtein Distance based metric as discussed earlier, and sorted the predictions in ascending order of the score. After this, I looked at the 20 best and the 20 worst predictions, and I made an interesting observation (although it's not about the model itself :P). If you look at the words it got correct, they are mostly simple enough - straightforward examples, and the model does well.
- However, if you look at what it is getting wrong, we see that it is bad in some short forms/proper nouns that are taken from english/abbreviations. This is seen in examples such as 'okay', 'baee', 'iowa', 'ina', or 'dmk'.
- On close inspection at the examples on the left though, we see that a lot of the ground truths are just wrong. In fact, they look like they have been completely wrongly annotated, and there has been a misalignment of sorts in whatever annotated corpus they have been

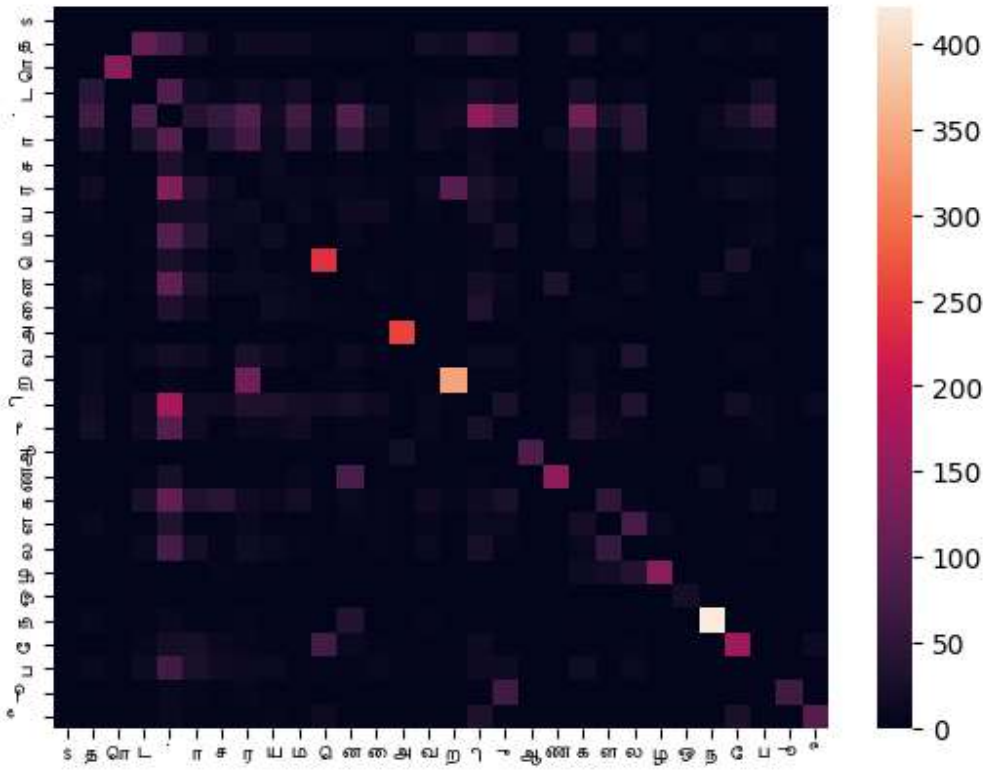
obtained from, because it looks like the ground truths correspond to a word that would appear before the given input word.

```
[['kartu', 'சிம்', 'கற்று', 0.0],
 ['vendum', 'குளிக்க', 'வெண்டும்', 0.0],
 ['varukinrana', 'உலா', 'வருகின்றன', 0.0],
 ['vitum', 'கலந்து', 'விடும்', 0.0],
 ['okay', 'ஓகே', 'கசோய்', 0.0],
 ['bee', 'பீ', 'ீய', 0.0],
 ['vazhi', 'நல்ல', 'வழி', 0.0],
 ['kei', 'சாவி', 'கெய்', 0.0],
 ['veentum', 'குளிக்க', 'வீண்டும்', 0.0],
 ['nava', 'நவ', 'வா', 0.0],
 ['shankar', 'பிரபு', 'நாங்கர்', 0.0],
 ['civinki', 'ஒட்டகச்', 'வினிங்கி', 0.0],
 ['pirakaasa', 'சைவப்', 'பிராசசச', 0.0],
 ['qey', 'சாவி', 'ஜெய்', 0.0],
 ['baee', 'பை', 'ீமே', 0.0],
 ['iowa', 'ஐயோவா', 'ஹேய', 0.0],
 ['payanikalai', 'சுற்றுலாப்', 'பயனிகளை', 0.0],
 ['high', 'ஹை', 'விகி', 0.0],
 ['ina', 'ஐநா', 'னன', 0.0],
 ['sankar', 'பிரபு', 'சங்கர்', 0.0]]
```

```
[['chopra', 'சோப்ரா', 'சோப்ரா', 1.0],
 ['nanmai', 'நன்மை', 'நன்மை', 1.0],
 ['seidhen', 'செய்தேன்', 'செய்தேன்', 1.0],
 ['arivathu', 'அறிவது', 'அறிவது', 1.0],
 ['pali', 'பலி', 'பலி', 1.0],
 ['senti', 'சென்டி', 'சென்டி', 1.0],
 ['vaangkum', 'வாங்கும்', 'வாங்கும்', 1.0],
 ['thoalviyaith', 'தோல்வியைத்', 'தோல்வியைத்', 1.0],
 ['thimuka', 'திமுக', 'திமுக', 1.0],
 ['uyirinam', 'உயிரினம்', 'உயிரினம்', 1.0],
 ['ulagaip', 'உலகைப்', 'உலகைப்', 1.0],
 ['kattamaikka', 'கட்டமைக்க', 'கட்டமைக்க', 1.0],
 ['saerkkaiyaal', 'சேர்க்கையால்', 'சேர்க்கையால்', 1.0],
 ['puthumaigalai', 'புதுமைகளை', 'புதுமைகளை', 1.0],
 ['viradham', 'விரதம்', 'விரதம்', 1.0],
 ['thattaan', 'தட்டான்', 'தட்டான்', 1.0],
 ['scotch', 'ஸ்காட்ச்', 'ஸ்காட்ச்', 1.0],
 ['veechuthal', 'வீசுதல்', 'வீசுதல்', 1.0],
 ['thaathup', 'தாதுப்', 'தாதுப்', 1.0],
 ['sattama', 'சட்டமா', 'சட்டமா', 1.0]]
```

On the left, you have the bad predictions. On the right, you have the good ones.

- When we try to plot the confusion matrix, it is quite useless, only the diagonal values are meaningful. To better see how the predictions are, we only consider the populated part of the confusion matrix with between 0 and 500 entries (so that we aren't offset by the high values on the diagonal). Confusion Matrix for predictions by Vanilla.



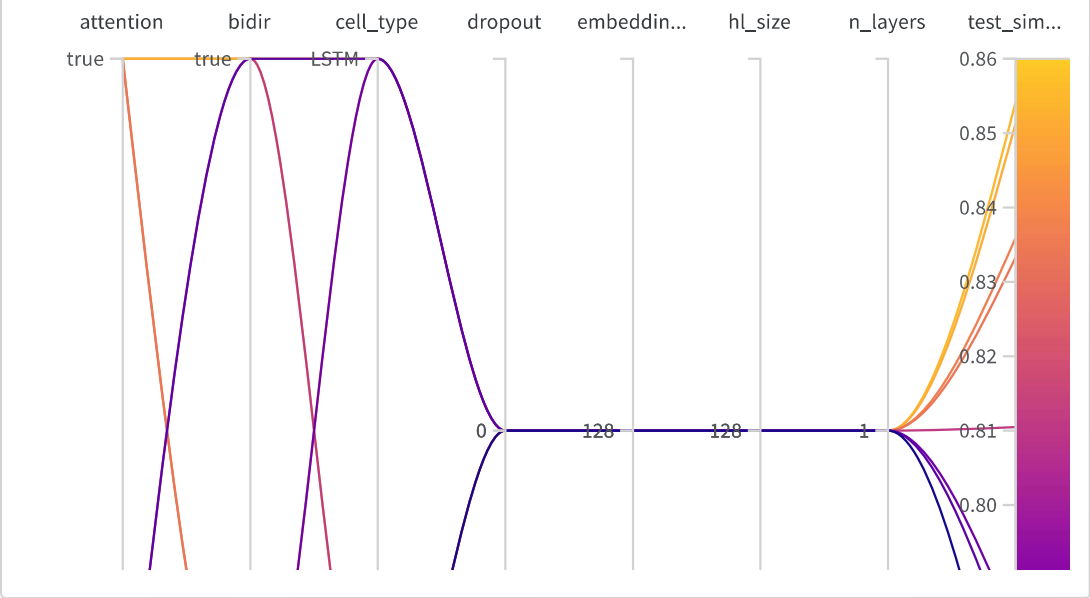
Partial Confusion Matrix for predictions by Vanilla.

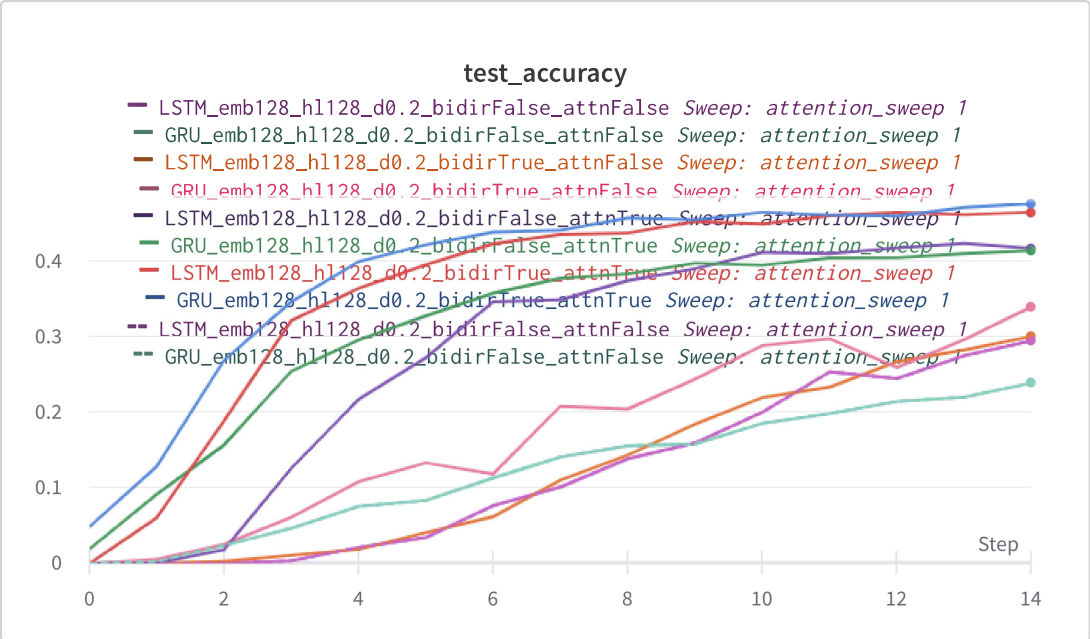
- By observing the confusion matrix, one thing we see is the column and the row corresponding to the dot matra (which is used above a letter to shorten the sound - the 5th entry in the axes) have relatively higher counts of errors, implying that the model often misses predictions involving this category.
- Surprisingly, we see that the model does exceedingly well on ந (soft na), while it is confused and makes mistakes between ண and ன, as you would expect from the model.
- You also see that the model does well with predicting ற (hard ra) correctly. We see that the model makes mistakes with ர (soft ra) and often ends up predicting it to be the former instead.
- The model is seen to do well in cases involving vowel matras, such as: அ (a), ெ (matra indicating ey), ே (matra indicating eyy) and ு (matra indicating uu).
- The model also shows some confusion in cases involving the letters ல (soft la), ள (hard la), or ழ (retroflex la/zha depending on how you know it).

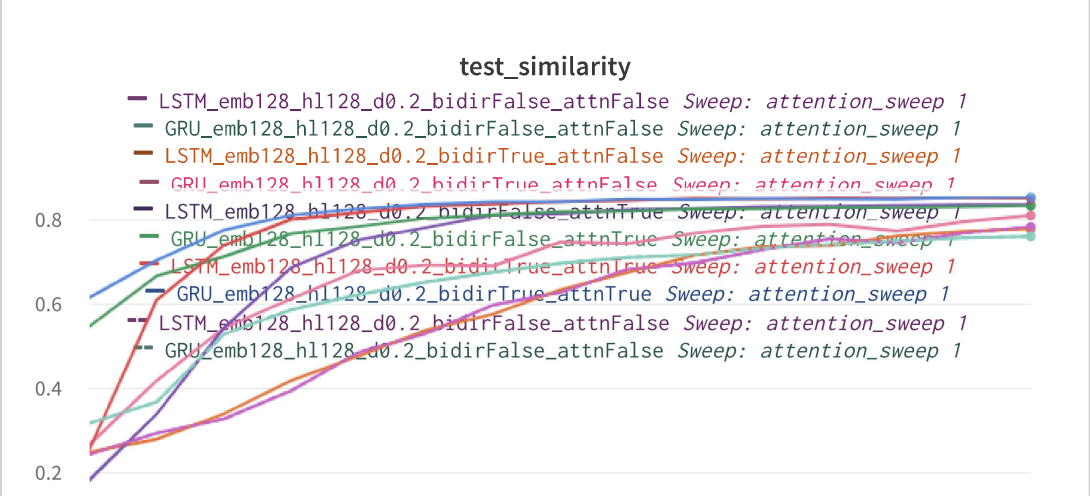
Question 5:

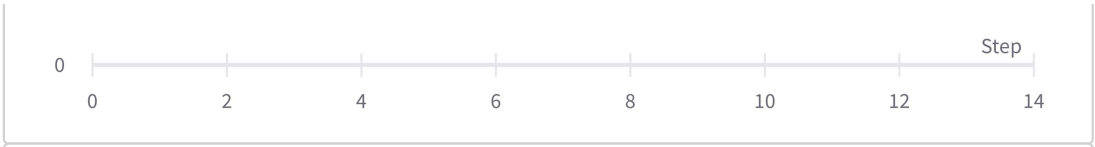
a) Training Attention Model

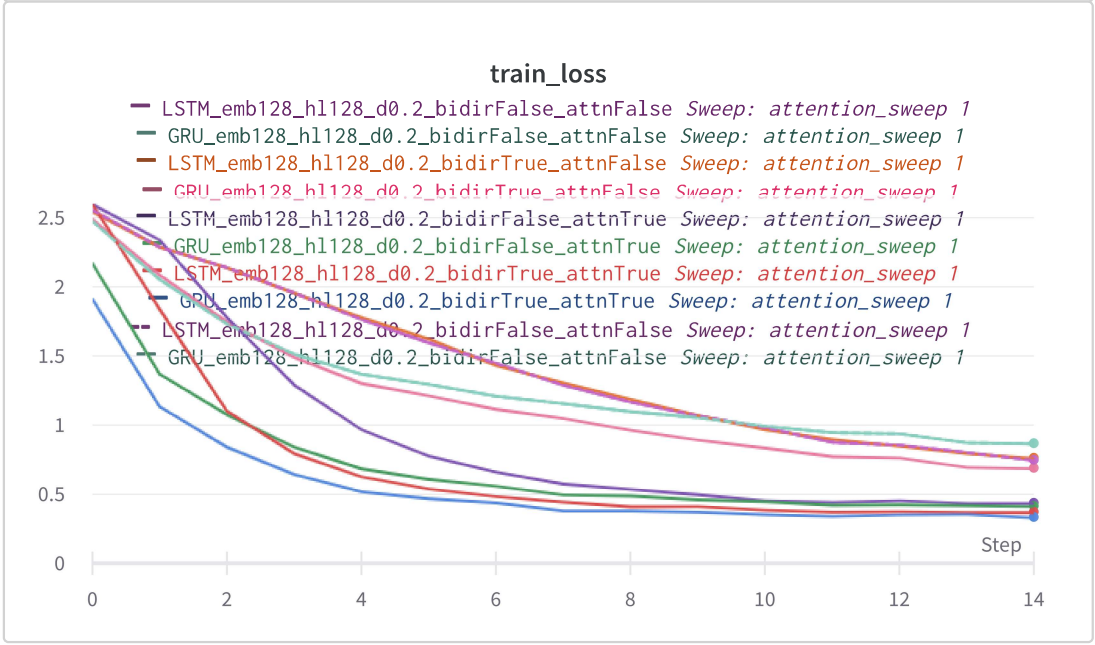
While I didn't really do a full-blown hyperparameter sweep, I did a small "mini-sweep" over a limited number of parameters exhaustively, as mentioned previously. I took the embedding size and the hidden layer size to be 128, and just varied the cell type (between LSTM and GRU), the bidirectionality (True and False), and with and without attention - so as to understand how attention helps the model. The plots from the sweep have been attached below:

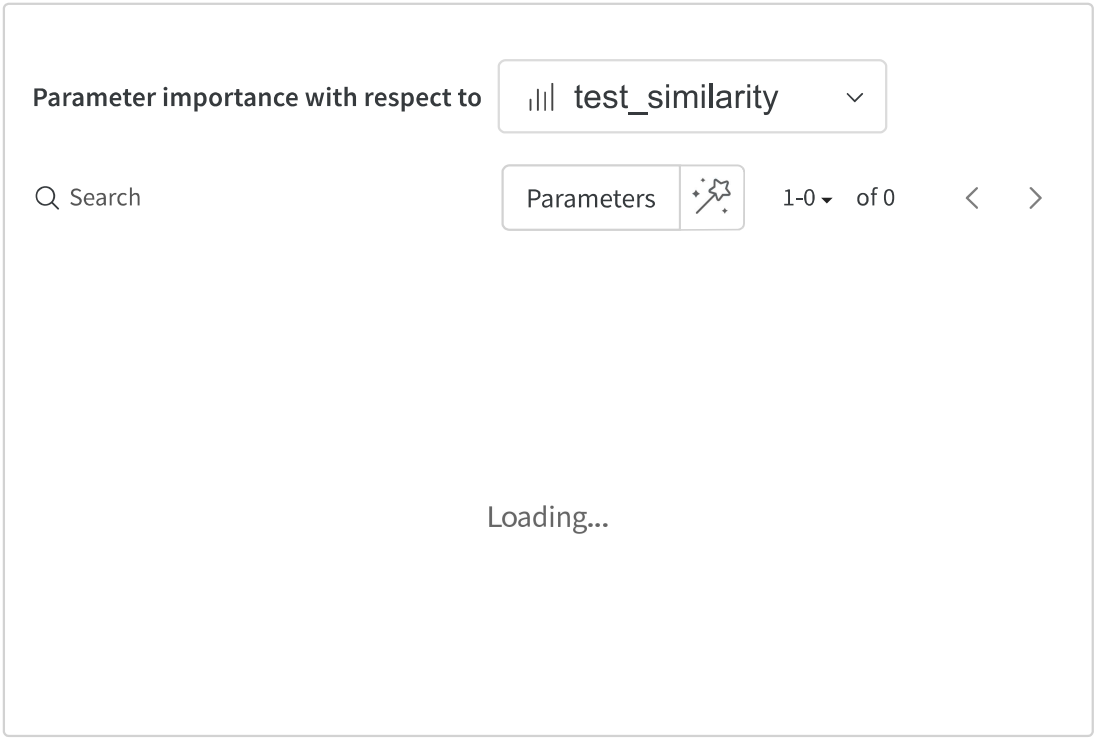

attention
bidir
cell_type
dropout
embeddin...
hl_size
n_layers
test_sim...
true
true
LSTM
0
128
128
1
0.86
0.85
0.84
0.83
0.82
0.81
0.80


test_accuracy
LSTM_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirTrue_attnFalse Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirTrue_attnFalse Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirFalse_attnTrue Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirFalse_attnTrue Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirTrue_attnTrue Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirTrue_attnTrue Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
0.4
0.3
0.2
0.1
0
Step
0
2
4
6
8
10
12
14


test_similarity
LSTM_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirTrue_attnFalse Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirTrue_attnFalse Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirFalse_attnTrue Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirFalse_attnTrue Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirTrue_attnTrue Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirTrue_attnTrue Sweep: attention_sweep 1
LSTM_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
GRU_emb128_hl128_d0.2_bidirFalse_attnFalse Sweep: attention_sweep 1
0.8
0.6
0.4
0.2







Observations:

- Right off the bat, the importance of attention is clearly visible. The model performance improves drastically, and there is a very strong correlation between the model performance and including attention.
- One more thing that should be noted is that the speed with which the model learns is high, i.e. there are very sharp jumps in performance, especially towards the beginning of the training process. In fact, if you end up using the same learning rate as you did for the vanilla network, after a while, the loss will start increasing, owing to how quickly it converges. If you use a much lower learning rate, the model is very slow to converge, and you don't utilize the fast learning that comes with an attention network. What is the fix?
- To address this, we used a learning rate with a linear decay, starting at 0.005 (as opposed to the 0.01 we used for the vanilla network) and linearly decaying to 0.001 after 30k epochs. This decay rate was chosen empirically, after observing that the loss started to increase slightly after these many iterations.
- Also apparent from this training process is the importance of bi-directionality, with it giving a marked improvement in performance. We also notice that the GRU and the LSTM cells do more or less similarly, with the GRU cell doing marginally better.
- We also see that the results we've got with the attention model (even though just 1 layer), exceed everything we got with the vanilla network by far. This just goes to show how effective the attention mechanism is to do the task of transliteration.

b) Best Model

The best model that we obtain has the following hyperparameters:

```
best_params = {
    'cell_type':'GRU', 'dropout':0.2, 'embedding_size':128, 'hl_size':128, 'bidirectional_flag':True
}
```

The best hyperparameters obtained for the attention model.

By using these hyperparameters, we get a validation accuracy of 0.47509765625 and a validation similarity score of

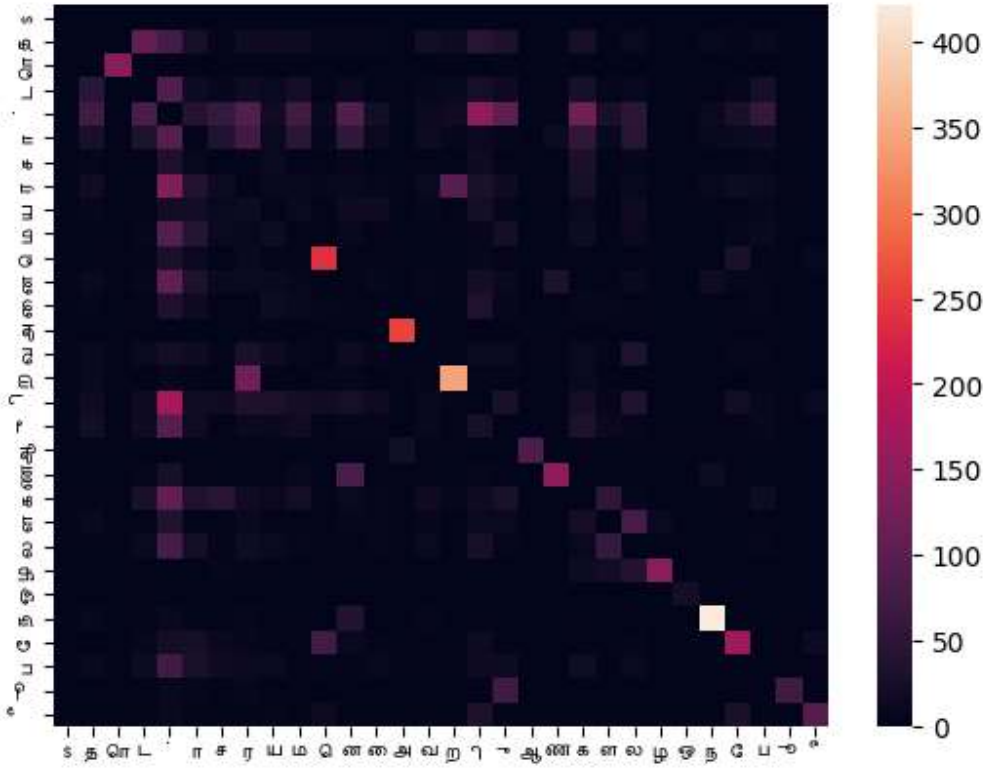
0.8572123446253838.

On evaluating this model on the given test dataset, we get a test accuracy 0.372802734375 and a test similarity score of 0.818051479780717.

Based on the disparity between the scores, we can conclude that we might be overfitting - although the performance on the test dataset is good, nevertheless.

c) Improvements over Vanilla

We see a significant jump in the score in the attention model as compared to the vanilla model.



Partial Confusion matrix for the predictions by the attention model.

On simply glancing at the confusion matrix, we don't really see anything very different from that of the vanilla model, and we probably won't. The confusion matrix in these cases are good to make generic observations about the limitations of these models in the tasks, but not the best for understanding how one model does

better, because intrinsically, the way we form a confusion matrix is something that is a bit of an approximation in itself.

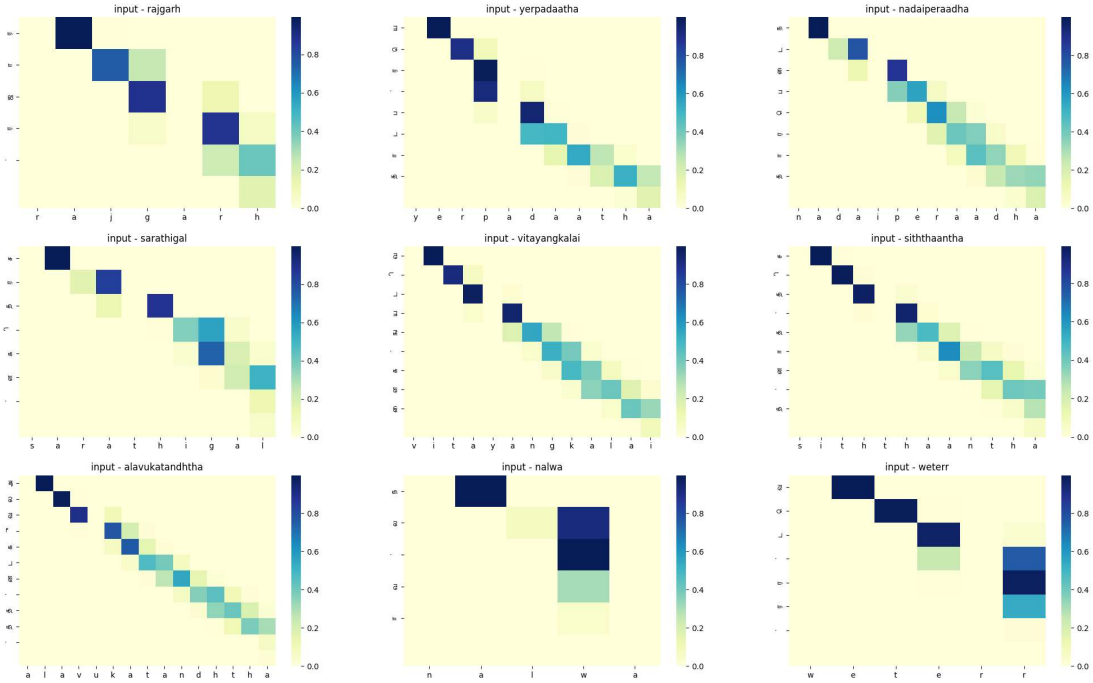
So, what do we do now? Let us look at the predictions that the attention model got right that the vanilla model got wrong.

```
['vealaiyaiyum', 'வேலையையும்', 'வேளையையும்', 'வேலையையும்']
['kaithal', 'கைதல்', 'கைத்ல', 'கைதல்']
['oththuzhaikkinranar', 'ஒத்துழைக்கின்றனர்', 'ஒத்துழைக்கின்றர்', 'ஒத்துழைக்கின்றனர்']
['maruththuvarai', 'மருத்துவரை', 'மறுத்துவரை', 'மருத்துவரை']
['aayiraththai', 'ஆயிரத்தை', 'ஆயிர்ததை', 'ஆயிரத்தை']
['kidaikkakkoodiya', 'கிடைக்கக்கூடிய', 'கிடைக்கட்கூடிய', 'கிடைக்கக்கூடிய']
['piramukar', 'பிரமுகர்', 'பிரமுக்◌', 'பிரமுகர்']
['mutharkattamaaga', 'முதற்கட்டமாக', 'முதர்கட்டாக', 'முதற்கட்டமாக']
['seiyappattiruppaar', 'செய்யப்பட்டிருப்பார்', 'செய்யப்பட்டுப்பார்', 'செய்யப்பட்டிருப்பார்']
['mandabaththinullea', 'மண்டபத்தினுள்ளே', 'மண்டதத்தினில்லே', 'மண்டபத்தினுள்ளே']
['karumpalakaiyil', 'கரும்பலகையில்', 'கர்ம்பகளைகளி', 'கரும்பலகையில்']
['magizhunthu', 'மகிழுந்து', 'கசிழுந்து', 'மகிழுந்து']
['ontruthirandulla', 'ஒன்றுதிரண்டுள்ள', 'ஒன்றுதி◌ர்◌◌ல்', 'ஒன்றுதிரண்டுள்ள']
['seimadhi', 'செய்மதி', 'செய்மாி', 'செய்மதி']
['melton', 'மெல்டன்', 'மெல்ட்ன்', 'மெல்டன்']
['kodukkappattikkum', 'கொடுக்கப்பட்டிக்கும்', 'கொடுக்கப்படிட்குக்ம்', 'கொடுக்கப்பட்டிக்கும்']
['purinthum', 'புரிந்தும்', 'புறிந்தும்', 'புரிந்தும்']
['seithiyaalargalukum', 'செய்தியாளர்களுக்கும்', 'செய்தியாளர்களுகும்', 'செய்தியாளர்களுக்கும்']
['purivadhu', 'புரிவது', 'புறிவது', 'புரிவது']
['guha', 'குஹா', 'கு', 'குஹா']
```

Here are 20 randomly chosen predictions that attention got right. The order displayed is [input, target, vanilla prediction, attention prediction].

- One of the main things to notice from this is that the changes learnt by the attention model are subtle, yet they make all the difference. We see that in almost all of these cases, the vanilla based model almost had it except for maybe a couple of slightly mismatched syllables, or slightly wrong consonants (not that the vanilla predictions don't make sense, can be thought of as the prediction being just slightly wrong). This is where the attention-based model snatches it away, with it consistently performing very well on the confusing consonants and getting all the wrong syllables of the vanilla based prediction right.
- This is induced by the additional information we give in the form of attention, telling the model which parts it should be focusing on at a given time instant, and using that to avoid making these tiny mistakes that bog the vanilla model down.

d) Attention Heatmaps



The Attention Heatmaps for 9 random inputs

We see that the model has given attention to the right parts of the input, corresponding to the output tamil characters.

Question 6 (Optional):

Not Attempted.

Question 7:

All code can be found in the following github repository: https://github.com/Dhananjay42/cs6910-assn3.

Reference: The skeleton code has heavily been developed based on the provided seq2seq blog code https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html.

Question 8 (Optional):

Not Attempted.

Self Declaration:

I, Dhananjay Balakrishnan (Roll Number: ME19B012), swear on my honour that I have written the code and the report by myself and have not copied it from the internet or other students.

Created with 💔 on Weights & Biases.

https://wandb.ai/clroymustang/cs6910-assignment-3/reports/CS6910-Assignment-3-Recurrent-Neural-Networks--Vmlldzo0MzI4NjY3