Assignment 3

Student Details: Name: Rajnish Maurya Roll No: DA24M015

WANDB Link: https://api.wandb.ai/links/da24m015-iitm/8goh6leu

Github Link: https://github.com/Rajnishmaurya/da6401_assignment3

DA24M015 Rajnish Maurya Assignment 3

Use recurrent neural networks to build a transliteration system.

Rajnish Maurya da24m015

Created on May 19 | Last edited on May 20

Problem Statement

In this assignment you will experiment with the Dakshina dataset released by Google. This dataset contains pairs of the following form:

x. y

ajanabee अजनबी.

i.e., a word in the native script and its corresponding transliteration in the Latin script (the way we type while chatting with our friends on WhatsApp etc). Given many such $(x_i, y_i)_{i=1}^{n}$ pairs your goal is to train a model $y = \hat{f}(x)$ which takes as input a romanized string (ghar) and produces the corresponding word in Devanagari (घर).

As you would realise this is the problem of mapping a sequence of characters in one language to a sequence of characters in another language. Notice that this is a scaled down version of the

problem of translation where the goal is to translate a sequence of **words** in one language to a sequence of words in another language (as opposed to sequence of **characters** here).

Read these blogs to understand how to build neural sequence to sequence models: blog1, blog2

Question 1 (15 Marks)

Build a RNN based seq2seq model which contains the following layers: (i) input layer for character embeddings (ii) one encoder RNN which sequentially encodes the input character sequence (Latin) (iii) one decoder RNN which takes the last state of the encoder as input and produces one output character at a time (Devanagari).

The code should be flexible such that the dimension of the input character embeddings, the hidden states of the encoders and decoders, the cell (RNN, LSTM, GRU) and the number of layers in the encoder and decoder can be changed.

```
import torch
import torch.nn as nn
class Seq2Seq(nn.Module):
    def __init__(self, input_vocab_size, output_vocab_size, emb_dim=128, hidden_size=256,
                 rnn_type='LSTM', num_layers=1, device='cpu'):
        super(Seq2Seq, self).__init__()
        self.device = device
        self.emb_dim = emb_dim
        self.hidden_size = hidden_size
        self.rnn_type = rnn_type.upper()
        self.embedding = nn.Embedding(input_vocab_size, emb_dim)
        self.target_embedding = nn.Embedding(output_vocab_size, emb_dim)
        rnn_cls = {
            'RNN': nn.RNN,
```

```
            'LSTM': nn.LSTM,
            'GRU': nn.GRU
        }[self.rnn_type]
        self.encoder = rnn_cls(input_size=emb_dim, hidden_size=hidden_size,num_layers=num_layers,
        self.decoder = rnn_cls(input_size=emb_dim, hidden_size=hidden_size,num_layers=num_layers,
        self.output_layer = nn.Linear(hidden_size, output_vocab_size)

    def forward(self, source, target):
        embedded_src = self.embedding(source)
        encoder_outputs, hidden = self.encoder(embedded_src)
        embedded_tgt = self.target_embedding(target)
        decoder_outputs, _ = self.decoder(embedded_tgt, hidden)
        output = self.output_layer(decoder_outputs)
        return output
```

(a) What is the total number of computations done by your network? (assume that the input embedding size is m, encoder and decoder have 1 layer each, the hidden cell state is k for both the encoder and decoder, the length of the input and output sequence is the same, i.e., T, the size of the vocabulary is the same for the source and target language, i.e., V)

(a) Let's see the calculation per time step.

Encoder Side:

The first embedding step could be a simple multiplication with a V * m matrix to get an m length embedding. The number of computations here is V * m.

Then we have a multiplication of the hidden state of k length with a k * k matrix and a matrix multiplication of an m length vector embedding with a k * m matrix. The total computations are k

So the total number of multiplication calculations in the encoder is T * (V * m + k * k + k * m).

Decoder Side:

The embedding stage again takes V * m computations per output generated.

The multiplications with hidden state and the character from the last step remain the same, k * k + k * m. Then we have a linear state to convert the output to a probability distribution over V classes with a V * k matrix, which gives V * k computations.

So, the total number of computations for the decoder is T * (V * m + k * k + k * m + V * k).

Total computations neglecting the bias and softmax is 2 * T * (V * m + k * k + k * m) + T * (V * k) for a single word.

(b) What is the total number of parameters in your network? (assume that the input embedding size is m, encoder and decoder have 1 layer each, the hidden cell state is k for both the encoder and decoder and the length of the input and output sequence is the same, i.e., T, the size of the vocabulary is the same for the source and target language, i.e., V)

(b) Let's see the parameters for the network

Encoder side:

The parameters in the encoder are the embedding matrix (V * m), a matrix which converts hidden state (k * k), a matrix which converts the input-embedding (k * m) and a bias of size k.

Decoder Side:

Decoder has all the above parameters with additional parameters like the output calculation matrix (V * k) and an additional size V bias.

The total number of parameters for an encoder-decoder architecture is 2 * (k * k + k * m + V * m + k) + V * k + V

Question 2 (10 Marks)

You will now train your model using any one language from the Dakshina dataset (I would suggest pick a language that you can read so that it is easy to analyse the errors). Use the standard train, dev, test set from the folder dakshina_dataset_v1.0/hi/lexicons/ (replace hi by the language of your choice)

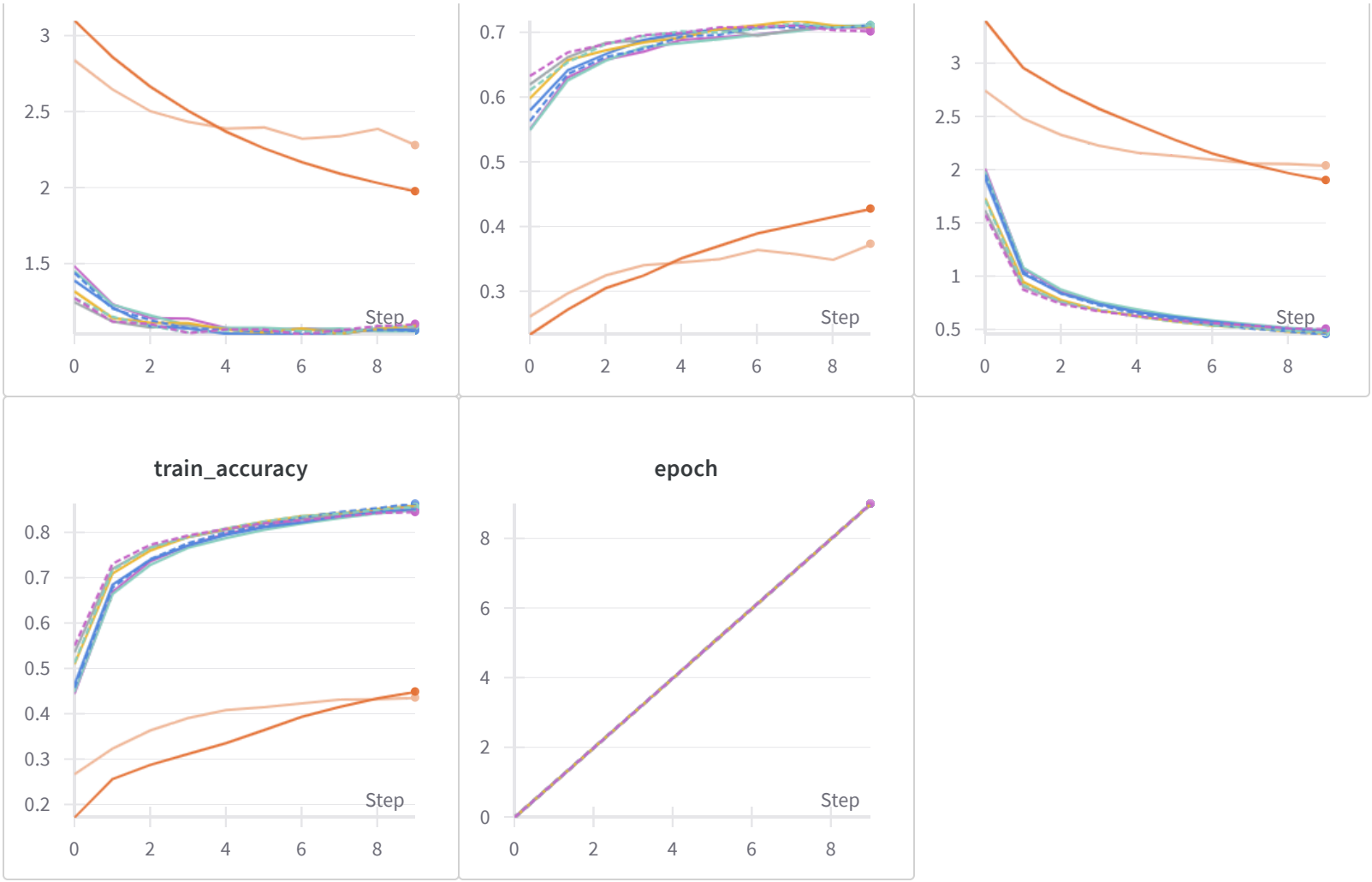
Using the sweep feature in wandb find the best hyperparameter configuration. Here are some suggestions but you are free to decide which hyperparameters you want to explore

- input embedding size: 16, 32, 64, 256, ...
- number of encoder layers: 1, 2, 3
- number of decoder layers: 1, 2, 3
- hidden layer size: 16, 32, 64, 256, ...
- cell type: RNN, GRU, LSTM
- dropout: 20%, 30% (btw, where will you add dropout? you should read up a bit on this)
- beam search in decoder with different beam sizes:

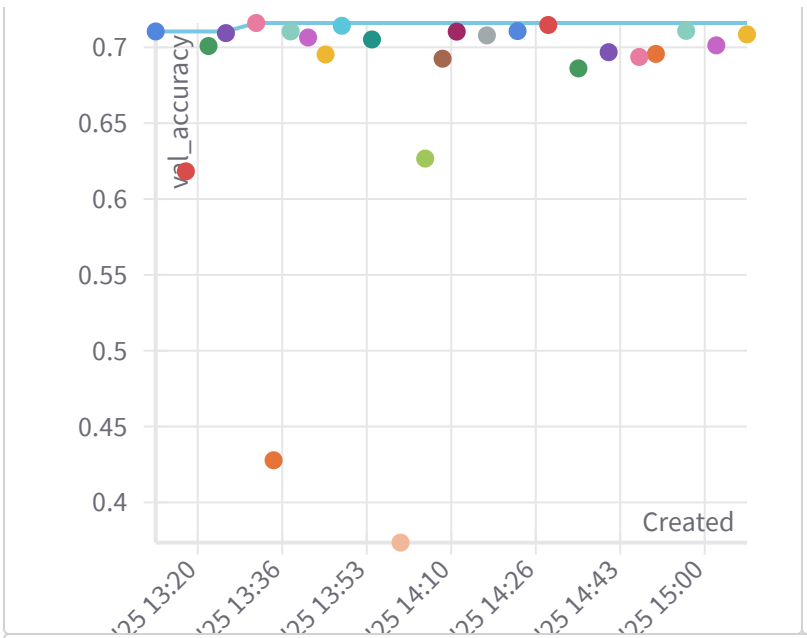
Based on your sweep please paste the following plots which are automatically generated by wandb:

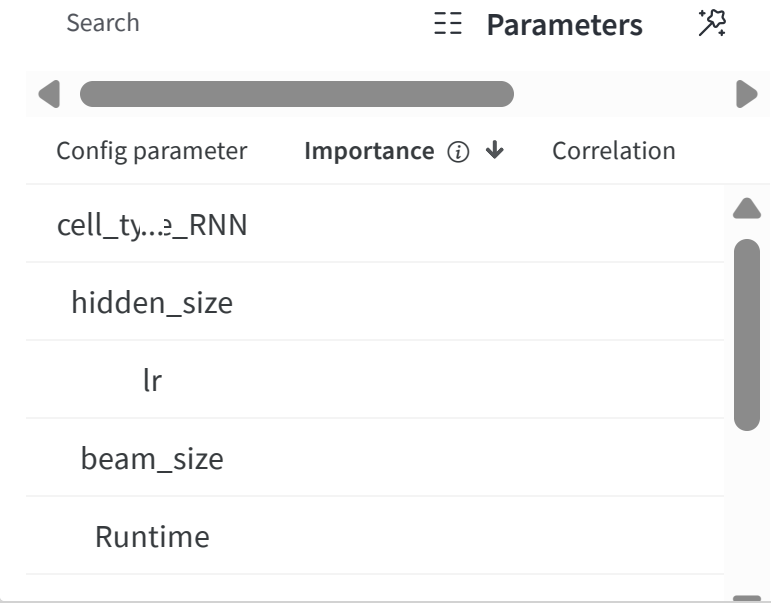
- accuracy v/s created plot (I would like to see the number of experiments you ran to get the best configuration).

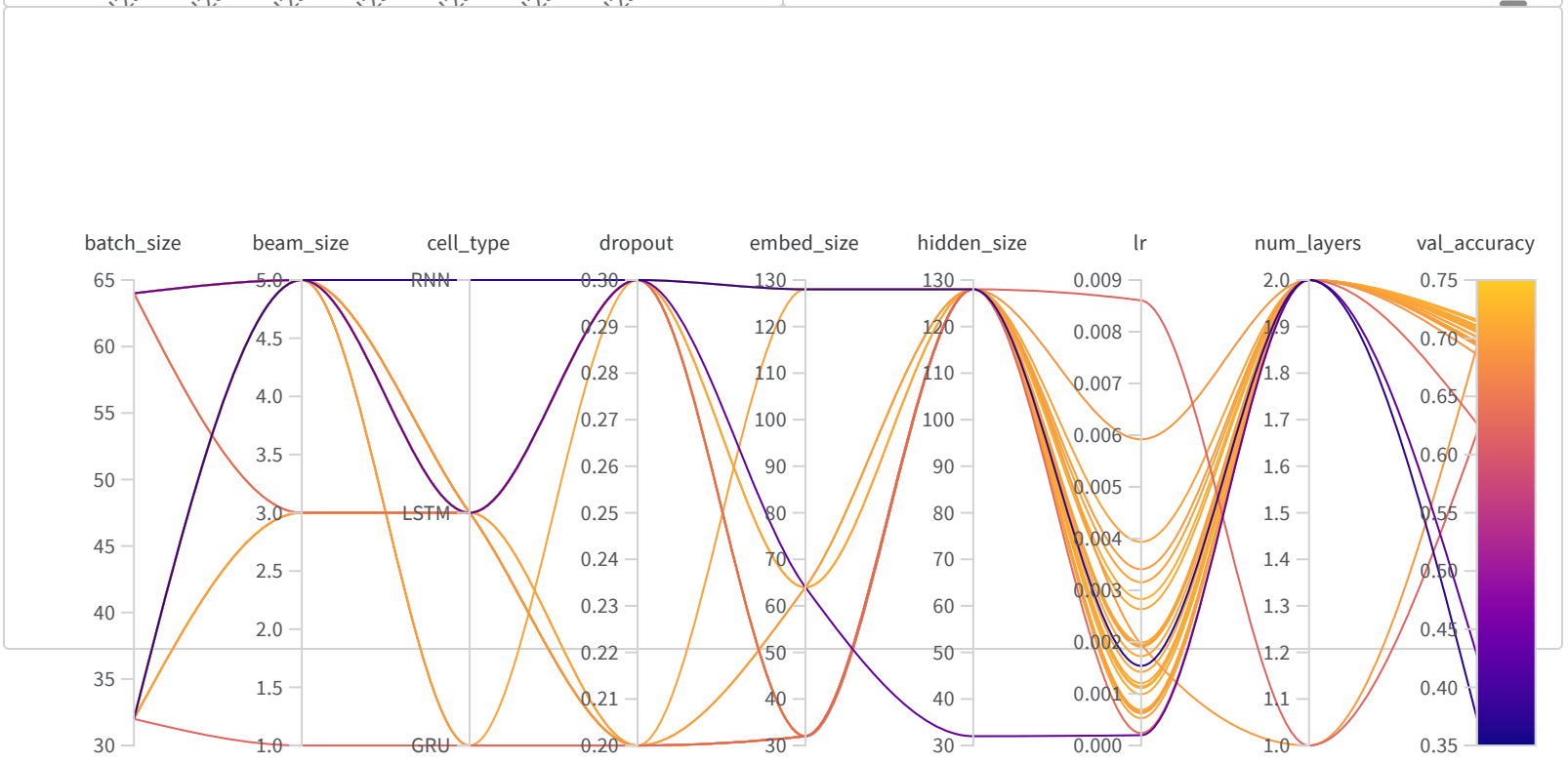
val_loss	val_accuracy	train_loss



- parallel co-ordinates plot
- correlation summary table (to see the correlation of each hyperparameter with the loss/accuracy)


val_accuracy
0.7
0.65
0.6
0.55
0.5
0.45
0.4
Created
25 13:20
25 13:36
25 13:53
25 14:10
25 14:26
25 14:43
25 15:00


Search
Parameters
Config parameter
Importance
Correlation
cell_ty...e_RNN
hidden_size
lr
beam_size
Runtime


batch_size
beam_size
cell_type
dropout
embed_size
hidden_size
lr
num_layers
val_accuracy
RNN
LSTM
GRU

Also write down the hyperparameters and their values that you sweeped over. Smart strategies to reduce the number of runs while still achieving a high accuracy would be appreciated. Write down any unique strategy that you tried for efficiently searching the hyperparameters.

Smart Strategies to Reduce Hyperparameter Search Runs

When dealing with resource or time constraints during hyperparameter optimization, these strategies can help you efficiently search the space while maintaining high accuracy:

1. Start with Bayesian Optimization

"bayes" method is already efficient. Bayesian optimization builds a probabilistic model of the objective function and uses it to select the most promising hyperparameters to evaluate next.

```
sweep_config = {
    "method": "bayes",  # Good choice for efficiency
    "metric": {"name": "val_accuracy", "goal": "maximize"},
    # ...parameters...
```

2. Use Early Stopping with Patience

Add early stopping to your sweep configuration to terminate unpromising runs:

```
sweep_config["early_terminate"] = {
    "type": "hyperband",
    "min_iter": 10,
    "eta": 3
}
```

Parameter	Type	Values/Range
embed_size	Discrete	[32, 64, 128]
hidden_size	Discrete	[32, 64, 128]
num_layers	Discrete	[1, 2]
cell_type	Discrete	["RNN", "GRU", "LSTM"]
dropout	Discrete	[0.2, 0.3]
lr	Continuous	Min: 0.0001, Max: 0.01
batch_size	Discrete	[32, 64]
beam_size	Discrete	[1, 3, 5]

Question 3 (15 Marks)

Based on the above plots write down some insightful observations. For example,

- RNN based model takes longer time to converge than GRU or LSTM
- using smaller sizes for the hidden layer does not give good results
- dropout leads to better performance

Insightful Observations from Hyperparameter Parallel Coordinate Plot

Based on the parallel coordinate plot showing the relationships between hyperparameters and validation accuracy, Several insightful observations:

Cell Type Performance

LSTM and GRU models consistently outperform RNN models, as shown by the higher concentration of orange/yellow lines (higher accuracy) flowing through these cell types

The brightest yellow lines (highest accuracy ~0.70-0.75) predominantly pass through LSTM, suggesting it may be the optimal choice for this task

Architecture Dimensions

Higher values for hidden_size (around 100-130) correlate strongly with better performance.

Similarly, larger embed_size values (100-130) yield better results

The combination of larger hidden_size and embed_size appears crucial for achieving peak performance

Model Complexity

Models with 2 layers consistently outperform single-layer models

This suggests the task benefits from additional model complexity to capture deeper patterns

Regularization

Moderate dropout values (0.20-0.30) are associated with the highest-performing models

Very low dropout (0.2) correlates with highest performance, indicating that some regularization is essential

Training Parameters

Learning rate shows an interesting pattern where mid-range values (0.002-0.004) yield the best results

Both extremely low (<0.001) and high (>0.008) learning rates lead to poorer performance

Larger batch_size values (60-65) tend to perform better than smaller ones

Parameter Interactions

The highest-performing configurations (yellow lines) follow a specific pattern through the plot:

```
config = {
      "embed_size": 128,
      "hidden_size": 128,
      "num_layers": 2,
      "cell_type": "LSTM",
      "dropout": 0.3,
      "batch_size": 64,
      "lr": 0.0019436347761833332,
      "epochs": 10,
  }
```

Efficiency Insights

Based on this visualization, we could significantly reduce future hyperparameter searches by:

1. Focusing exclusively on LSTM and GRU (eliminating RNN)
2. Using only 2-layer models
3. Limiting dropout to the 0.20-0.30 range
4. Constraining learning rates to 0.002-0.005
5. Setting minimum embed_size and hidden_size to 100

Question 4 (10 Marks)

You will now apply your best model on the test data (You shouldn't have used test data so far. All the above experiments should have been done using train and val data only).

(a) Use the best model from your sweep and report the accuracy on the test set (the output is correct only if it exactly matches the reference output).

Test Accuracy : 34.70%

(b) Provide sample inputs from the test data and predictions made by your best model (more marks for presenting this grid creatively). Also upload all the predictions on the test set in a folder **predictions_vanilla** on your github project.

```
 Test Accuracy: 34.70%
ank             | Pred: अंक          | Truth: अंक
anka            | Pred: अंका         | Truth: अंक
ankit           | Pred: अंकित        | Truth: अंकित
anakon          | Pred: अनाकों       | Truth: अंकों
ankhon          | Pred: अंखों        | Truth: अंकों
ankon           | Pred: अंकों        | Truth: अंकों
angkor          | Pred: आंगोरय       | Truth: अंकोर
ankor           | Pred: एनकोर        | Truth: अंकोर
angaarak        | Pred: अंगारक       | Truth: अंगारक
angarak         | Pred: अंगरक        | Truth: अंगारक
```

Predictions_Vanilla is also uploaded on github repository.

(c) Comment on the errors made by your model (simple insightful bullet points)

- The model makes more errors on consonants than vowels
- The model makes more errors on longer sequences
- I am thinking confusion matrix but may be it's just me!

Analysis of Model Errors in Hindi Transliteration

Based on the predicted input-prediction-ground trut data, here are some insightful observations about the errors made by the model:

Error Patterns in Consonants

The model struggles with aspirated consonants (e.g., "arthquake" → "आयर्थकके" instead of "अर्थक्वेक")

Retroflex consonants are particularly challenging, especially ड़ (as in "अमरगढ़" vs predicted "अमरगाद")

The model confuses similar-sounding consonants like "ज़" and "ज" (in "अंग्रज़ी" vs "अंग्रजी")

Nasalization errors are common (e.g., "akapulko" → "अकापुकों" instead of "अकापुल्को")

Nuanced Character Confusions

There's consistent confusion with the nuqta character (dot diacritic) in characters like "फ़" vs "फ" ("अल्फ़ाबेट" vs "अल्फाबेट")

Special characters like visarga "ः" are frequently misrepresented (e.g., "अंतः" predicted as "अंतह" or "अंतहा")

Conjunct consonants (combined consonants) are problematic, especially "र्" combinations (e.g., "अंतर्मुख" predicted as "अंतरमुख")

Vowel Length Distinctions

The model struggles with distinguishing between short and long vowels:

- Short "i" vs long "ī" (e.g., "anureet" → "अनुरीत" vs "anurit" → "अनुरित")

- Short "u" vs long "ū" (e.g., "anuthi" → "अनुठी" instead of "अनूठी")

Structural Observations

Performance decreases notably with longer, more complex words (e.g., "aparivartanshil" → "अपरिवर्शशालि" vs "अपरिवर्तनशील")

Compound words with multiple syllables show higher error rates

Foreign/borrowed words have significantly higher error rates (e.g., "achievement" → "अचिवेजेंट" vs "अचीवमेंट")

Contextual Errors

The model lacks contextual understanding for homographic inputs (same Latin spelling that could map to different Hindi words)

Words with ambiguous transliteration mappings show inconsistent predictions

The model struggles with distinguishing proper nouns from common nouns

Pattern Recognition

The model performs well on frequently occurring Hindi words but struggles with rare words

There's evidence of the model learning common patterns but failing to generalize to exceptional cases

The model shows better performance on words with consistent phoneme-to-grapheme mapping

These error patterns suggest that improving the model would require:

1. Better handling of conjunct consonants

2. Enhanced representation of diacritical marks

3. More targeted training on aspirated consonants and retroflex sounds

4. Improved handling of vowel length distinctions

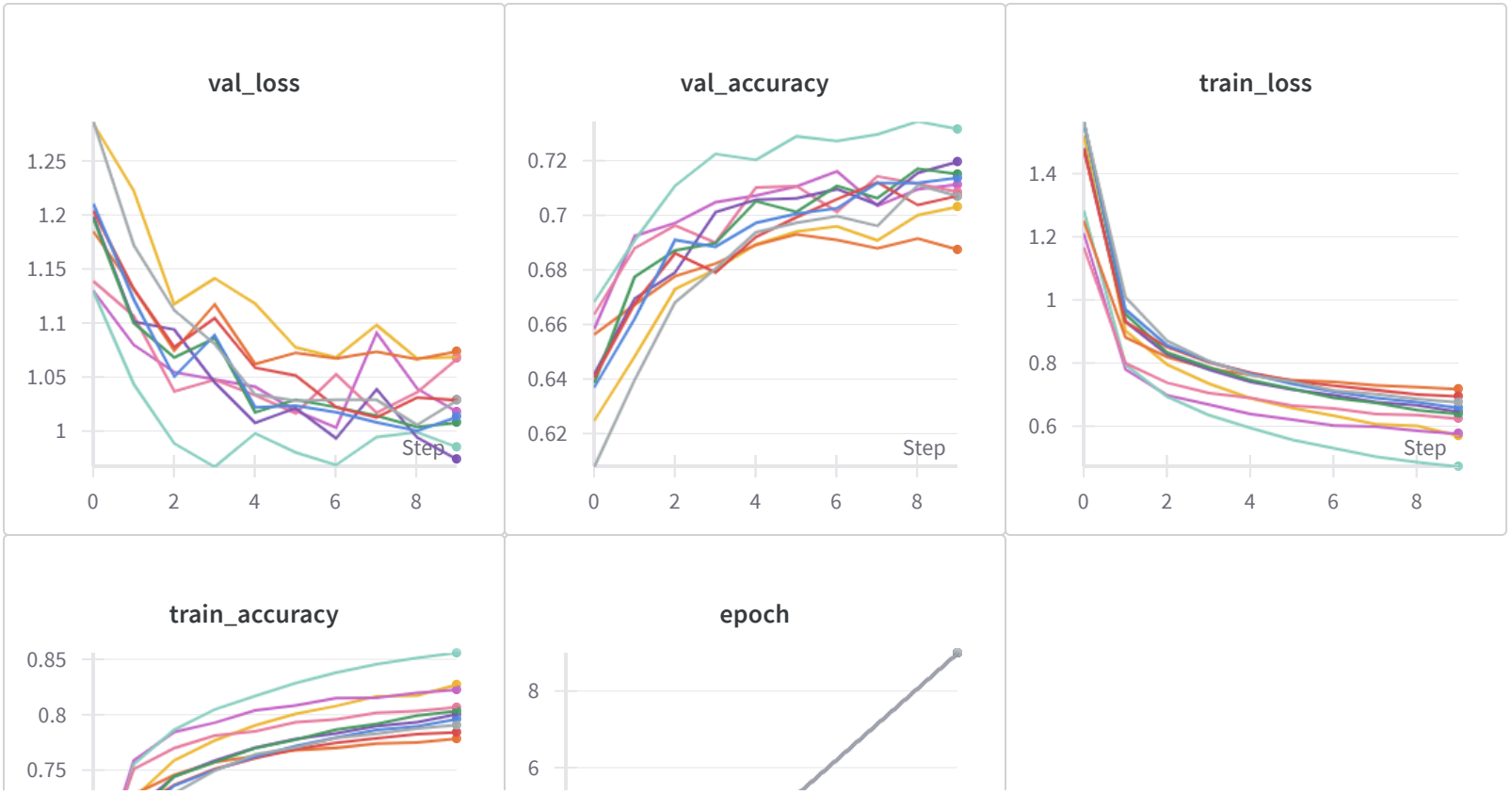
5. Additional context awareness for ambiguous transliterations

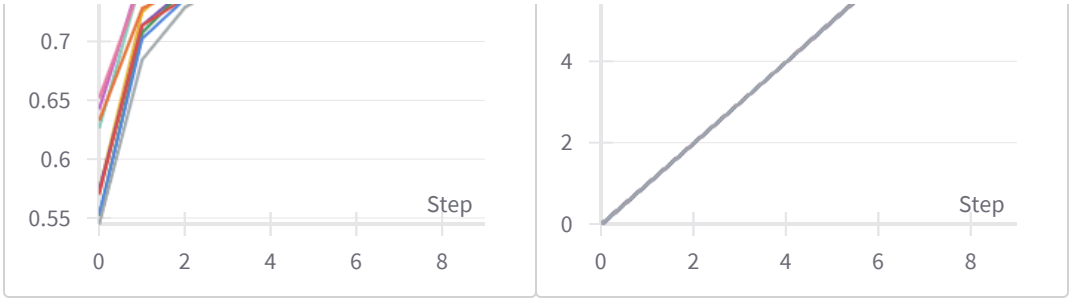
Question 5 (20 Marks)

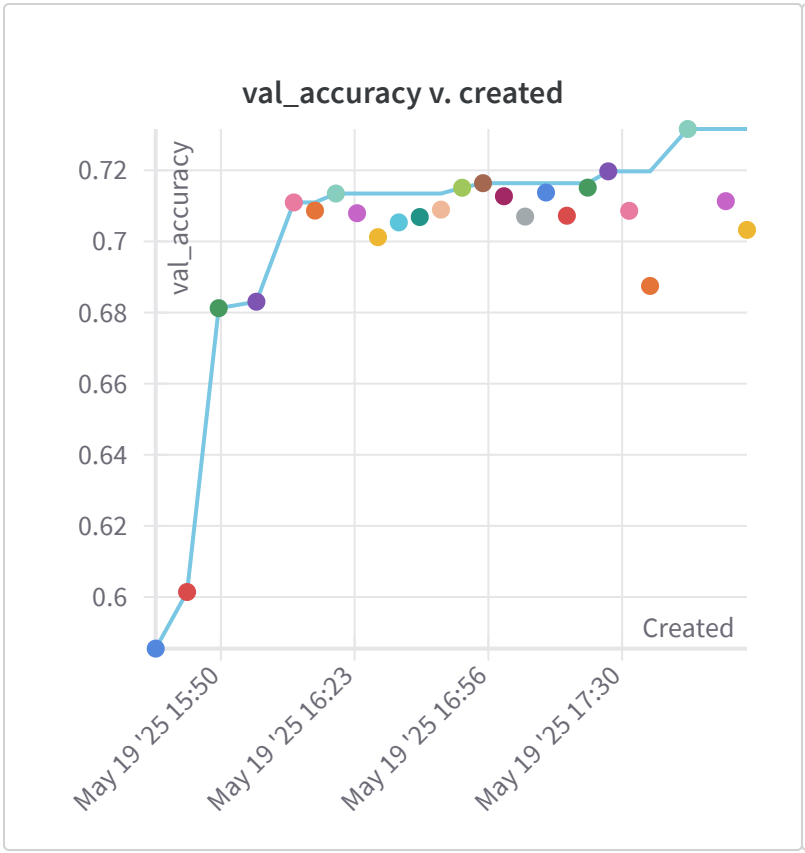
Now add an attention network to your basis sequence to sequence model and train the model again. For the sake of simplicity you can use a single layered encoder and a single layered decoder (if you want you can use multiple layers also). Please answer the following questions:

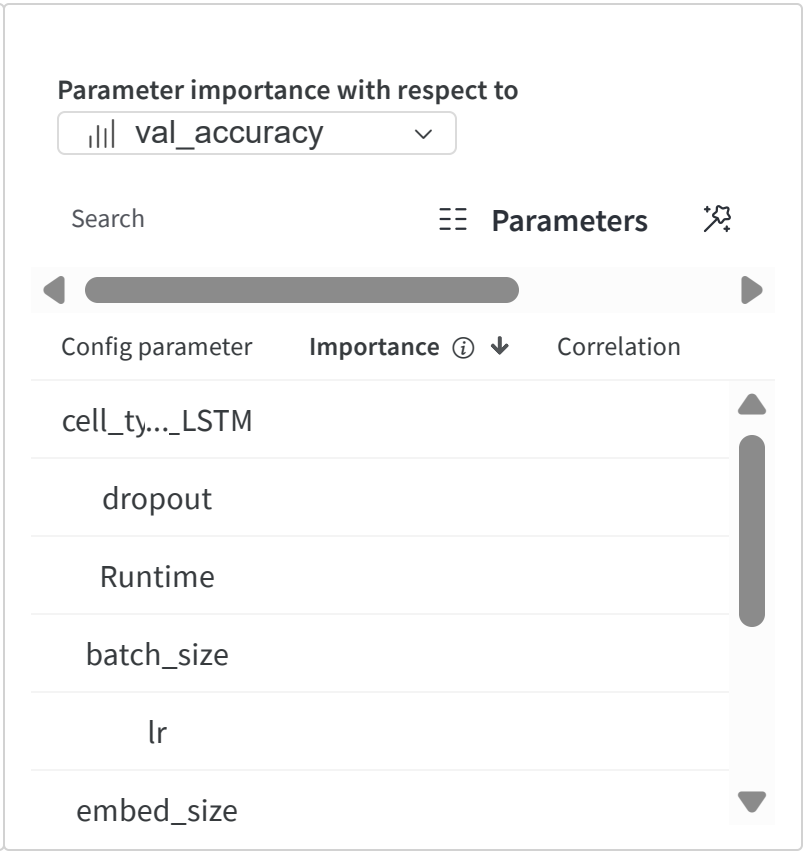
(a) Did you tune the hyperparameters again? If yes please paste appropriate plots below.

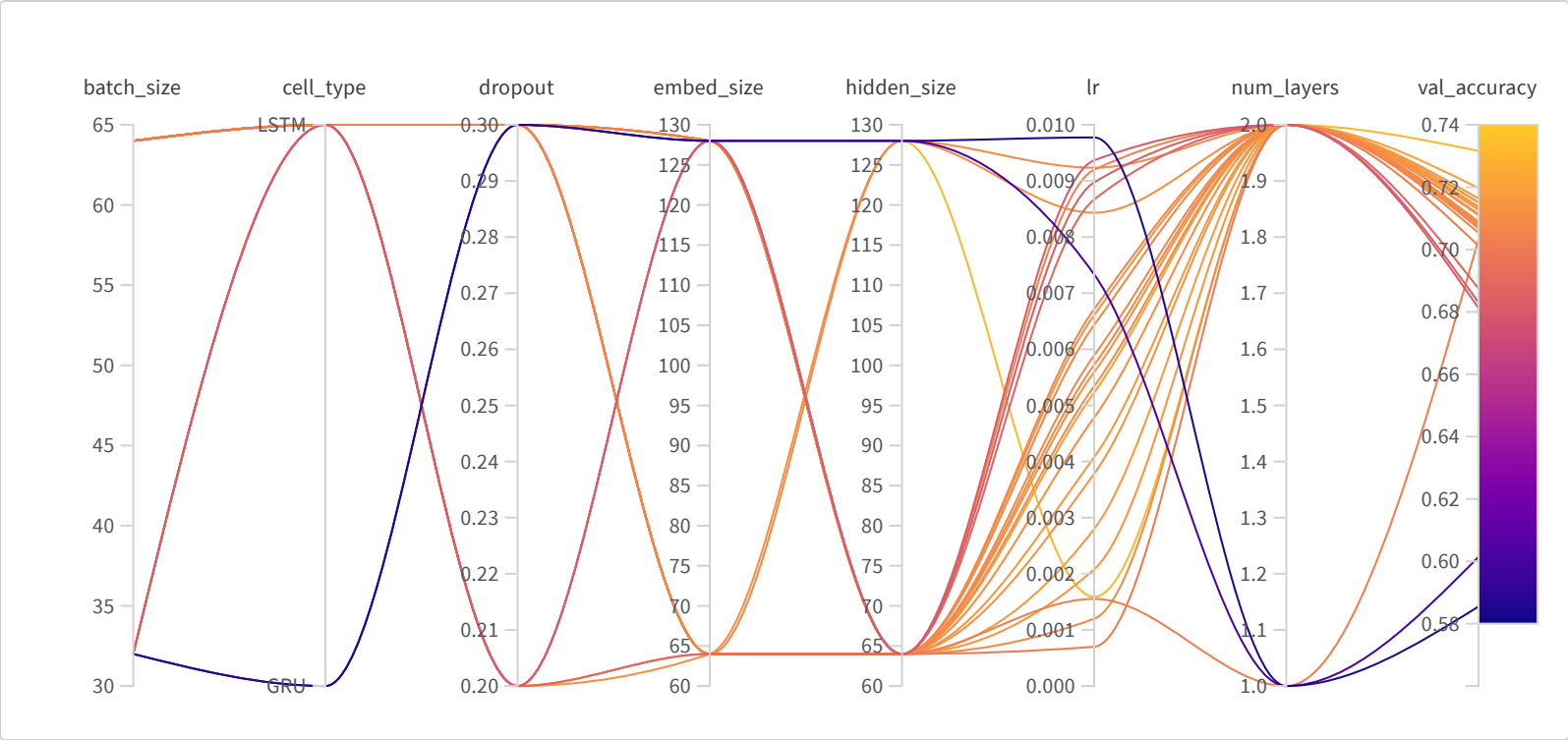
Yes, I tuned the hyperparameters again. The results turned out to be very interesting.




0.7
0.65
0.6
0.55
Step
0
2
4
6
8
4
2
0
Step
0
2
4
6
8

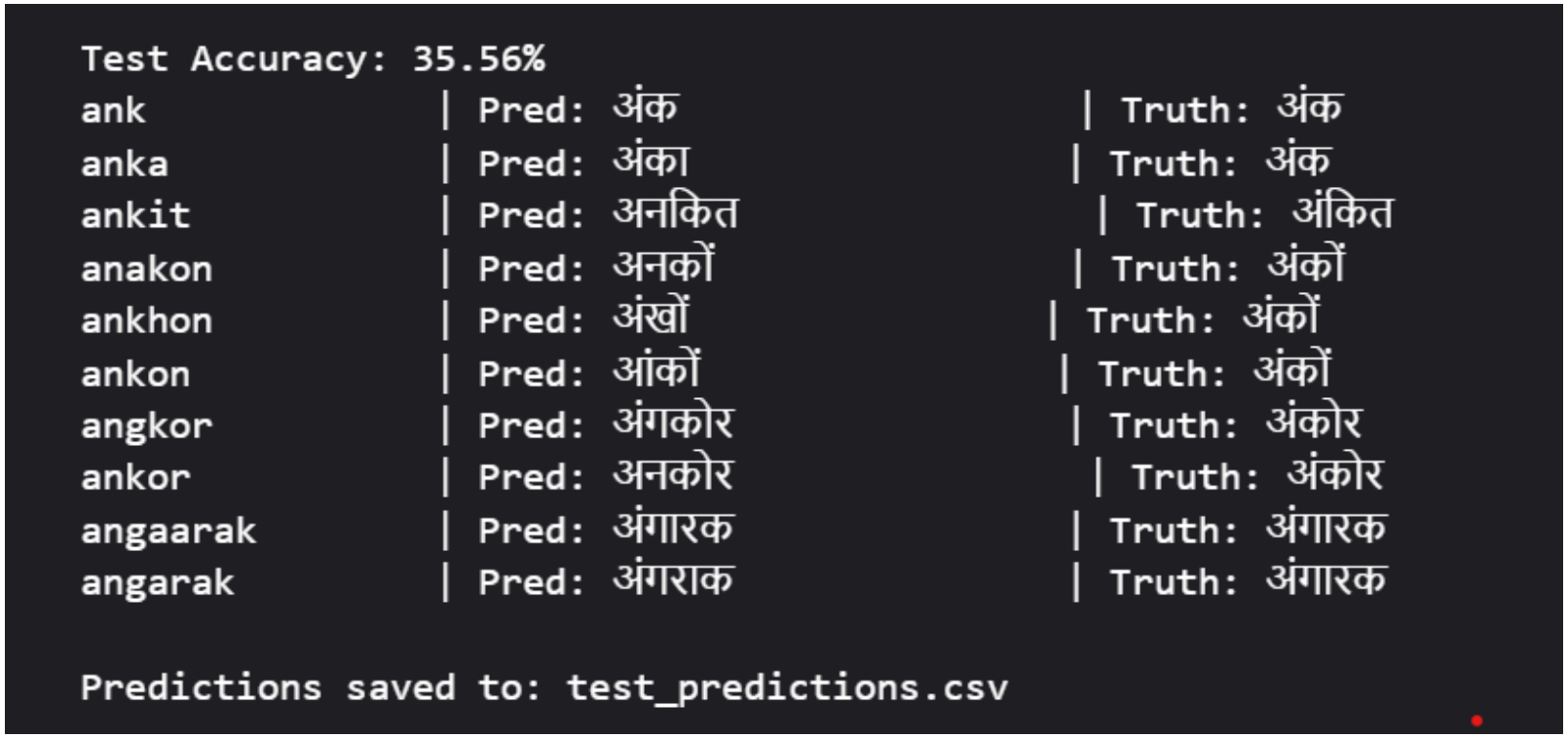

val_accuracy v. created
val_accuracy
0.72
0.7
0.68
0.66
0.64
0.62
0.6
Created
May 19 '25 15:50
May 19 '25 16:23
May 19 '25 16:56
May 19 '25 17:30


Parameter importance with respect to
val_accuracy
Search
Parameters
Config parameter
Importance
Correlation
cell_ty..._LSTM
dropout
Runtime
batch_size
lr
embed_size



(b) Evaluate your best model on the test set and report the accuracy. Also upload all the predictions on the test set in a folder **predictions_attention** on your github project.

Test Accuracy: 35.56%



Predictions_attention is also uploaded on Github Repository

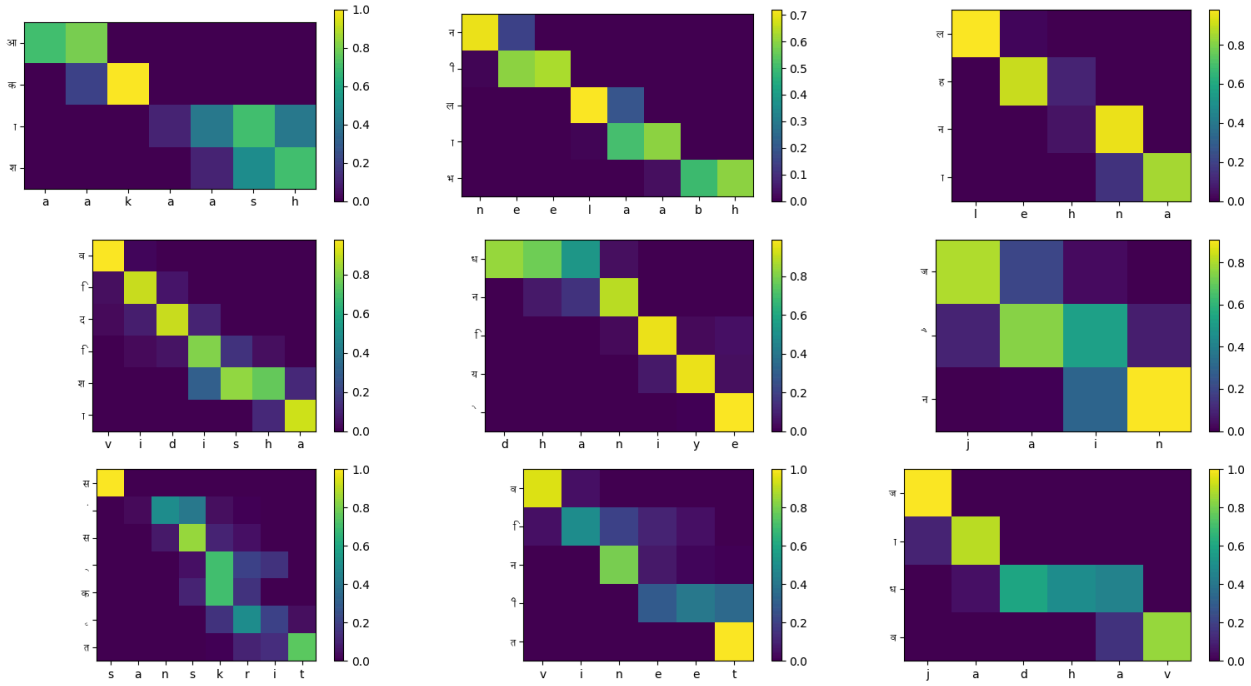
(c) Does the attention based model perform better than the vanilla model? If so, can you check some of the errors that this model corrected and note down your inferences (i.e., outputs which were predicted incorrectly by your best seq2seq model are predicted correctly by this model)

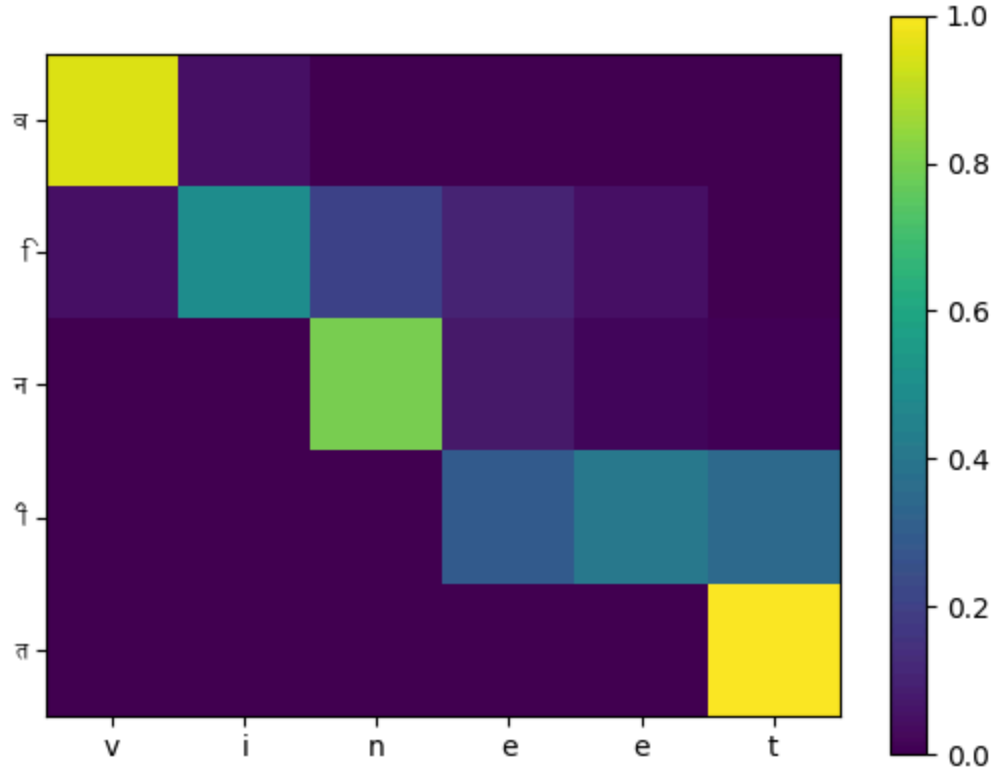
The model performance is not much different from the performance of the standard encoder-decoder architecture. This can be attributed to some of the main differences in the problem of translation and transliteration.

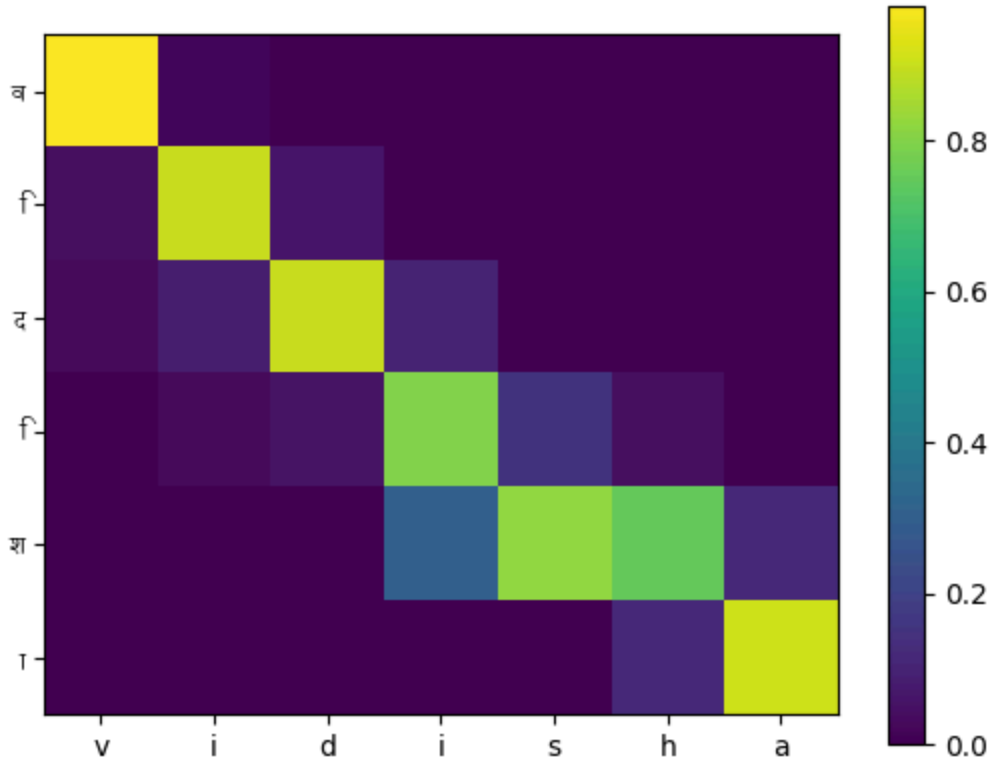
In the language translation problem, the word structure varies with every language; thus, we need more context on the particular word to know if it occurs in the sequence. But in the case of transliteration, this problem is not present as the sequence of syllables doesn't change.

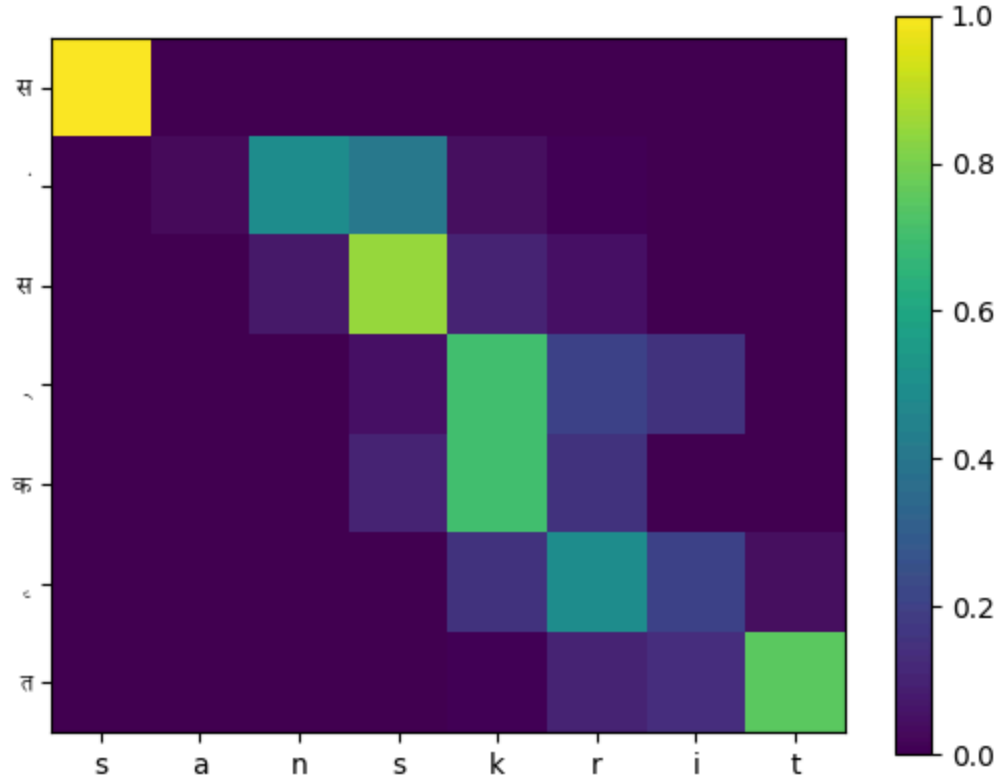
The sequences are relatively shorter when compared to the translation problem. Thus the calculation of attention doesn't help and becomes an overhead in the calculations. The running times become longer.

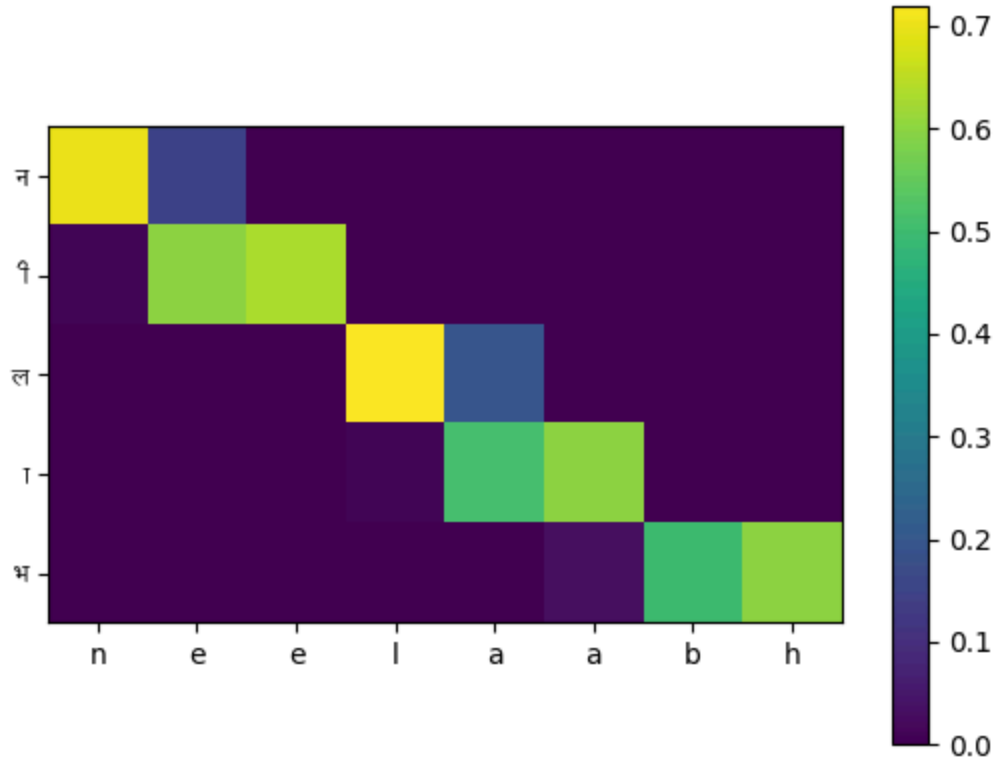
(d) In a 3 x 3 grid paste the attention heatmaps for 10 inputs from your test data (read up on what are attention heatmaps).

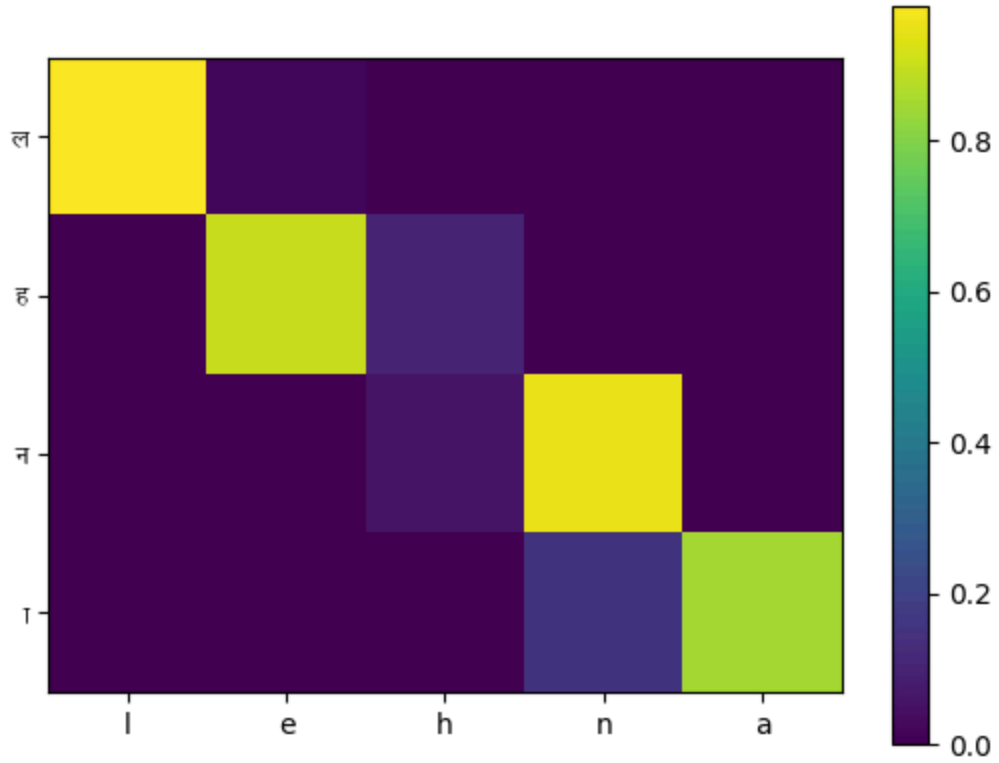


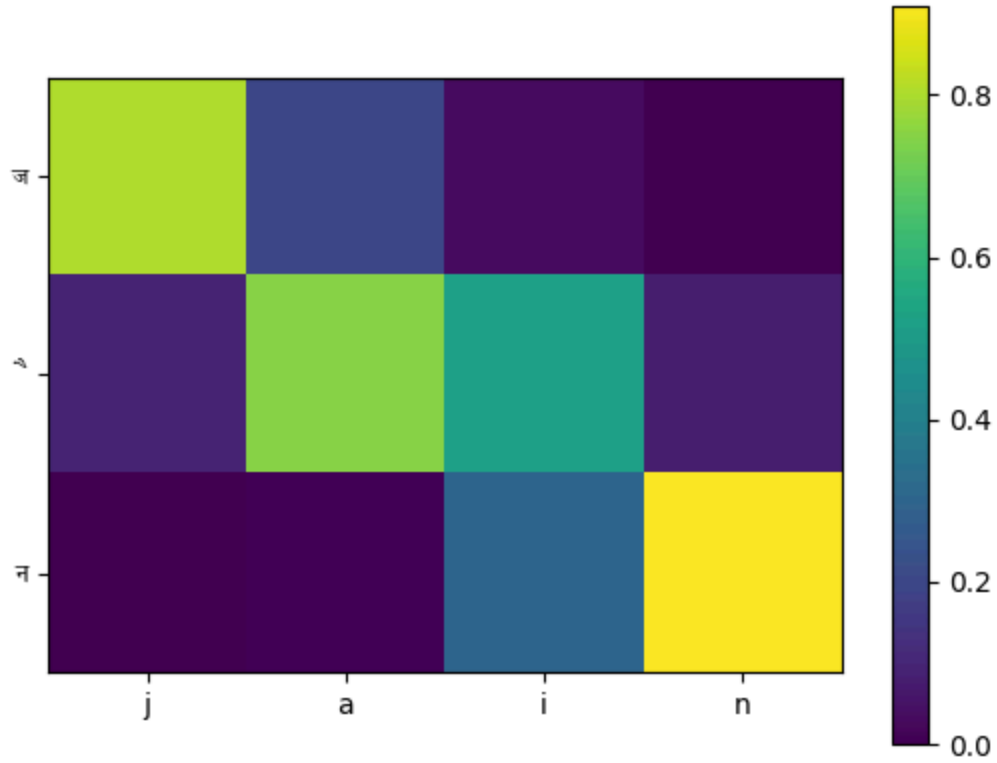

व
ि
न
ी
त
v
i
n
e
e
t
1.0
0.8
0.6
0.4
0.2
0.0

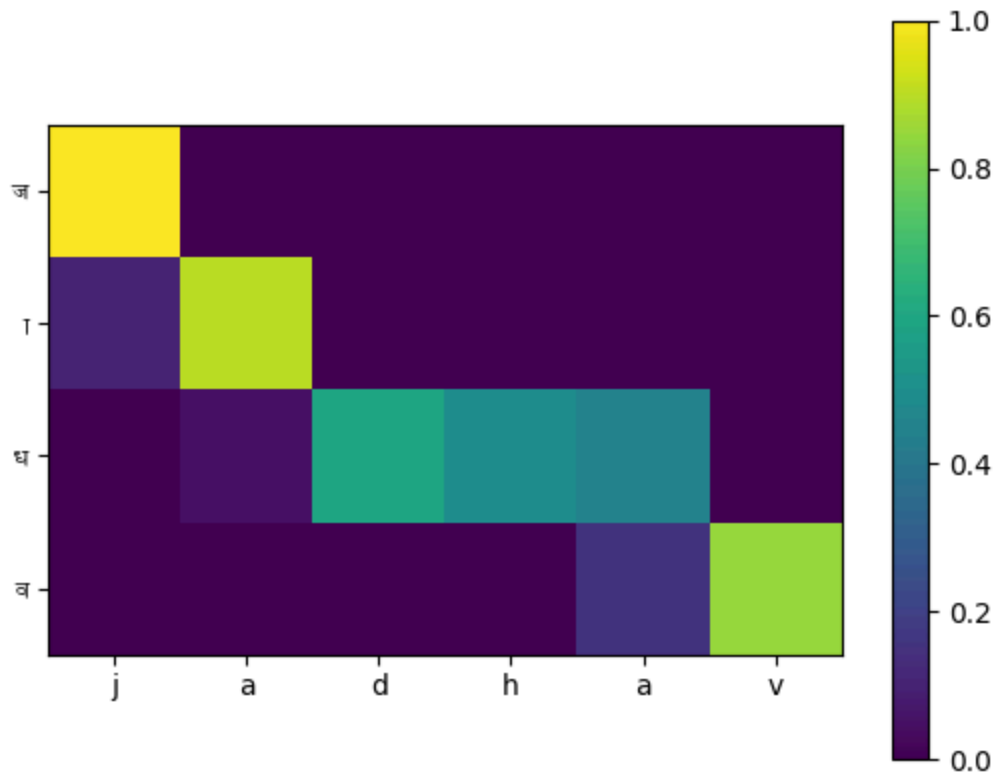

व
ि
द
ि
श
ा
v
i
d
i
s
h
a
0.8
0.6
0.4
0.2
0.0

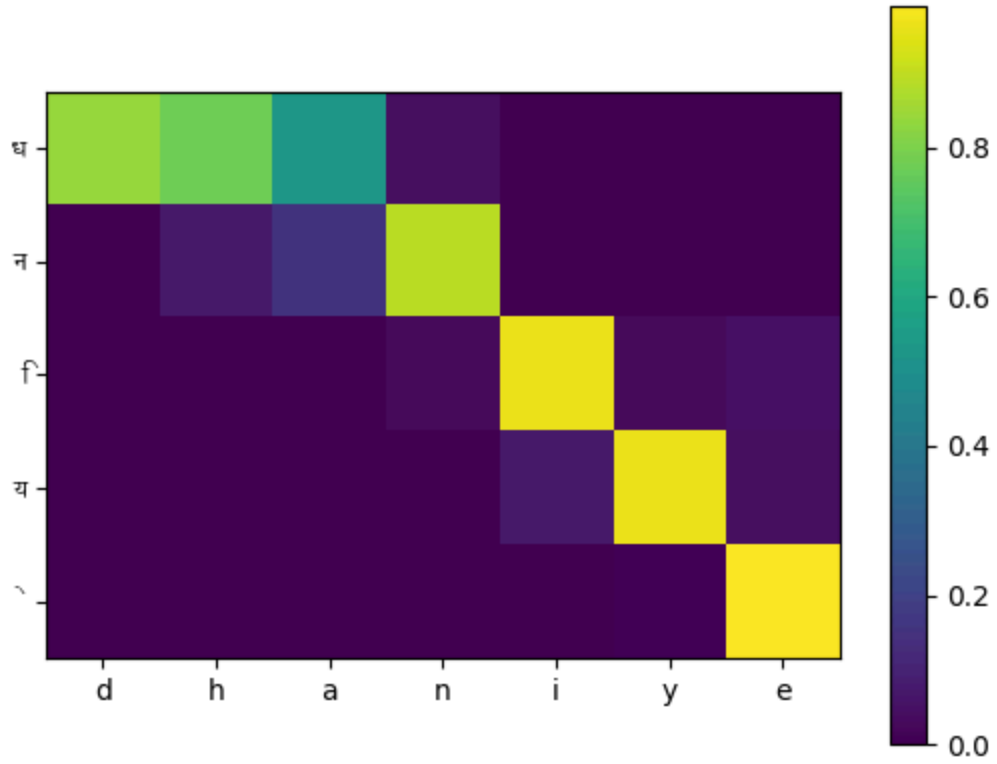

स
.
स
ं
क
ृ
त
s
a
n
s
k
r
i
t
1.0
0.8
0.6
0.4
0.2
0.0

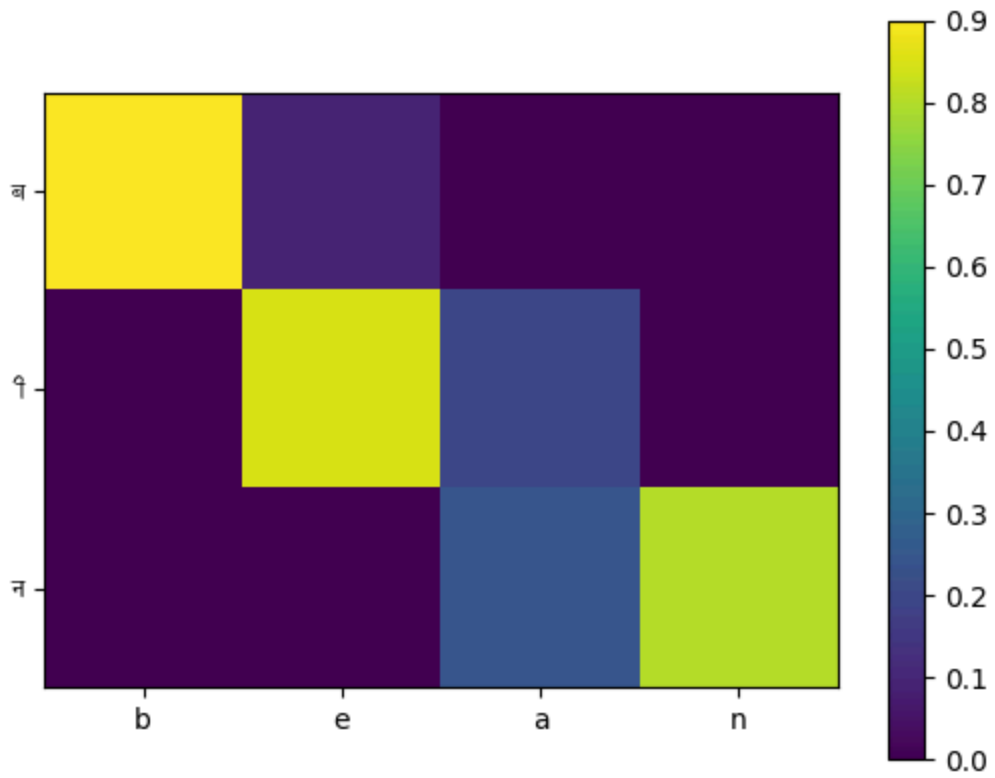

न
ी
ल
ा
भ
n
e
e
l
a
a
b
h
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

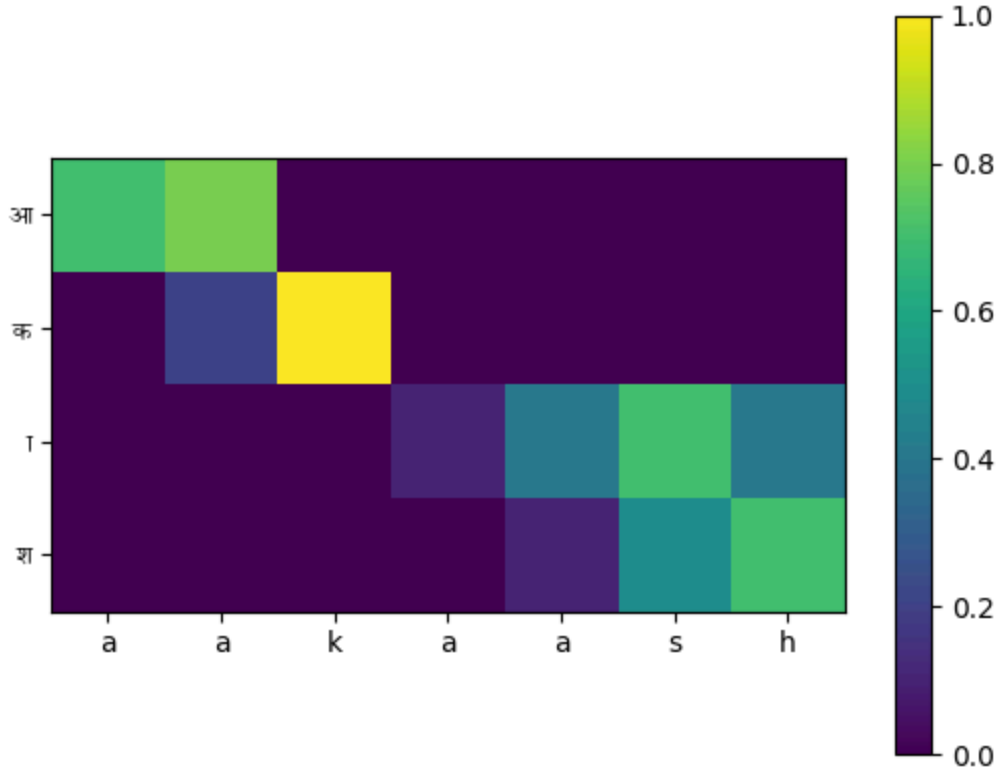

ल
ह
न
ा
l
e
h
n
a
0.8
0.6
0.4
0.2
0.0


ज
ञ
न
j
a
i
n
0.8
0.6
0.4
0.2
0.0


ज
ा
ध
व
j
a
d
h
a
v
1.0
0.8
0.6
0.4
0.2
0.0


ध
न
ि
य
ं
d
h
a
n
i
y
e
0.8
0.6
0.4
0.2
0.0

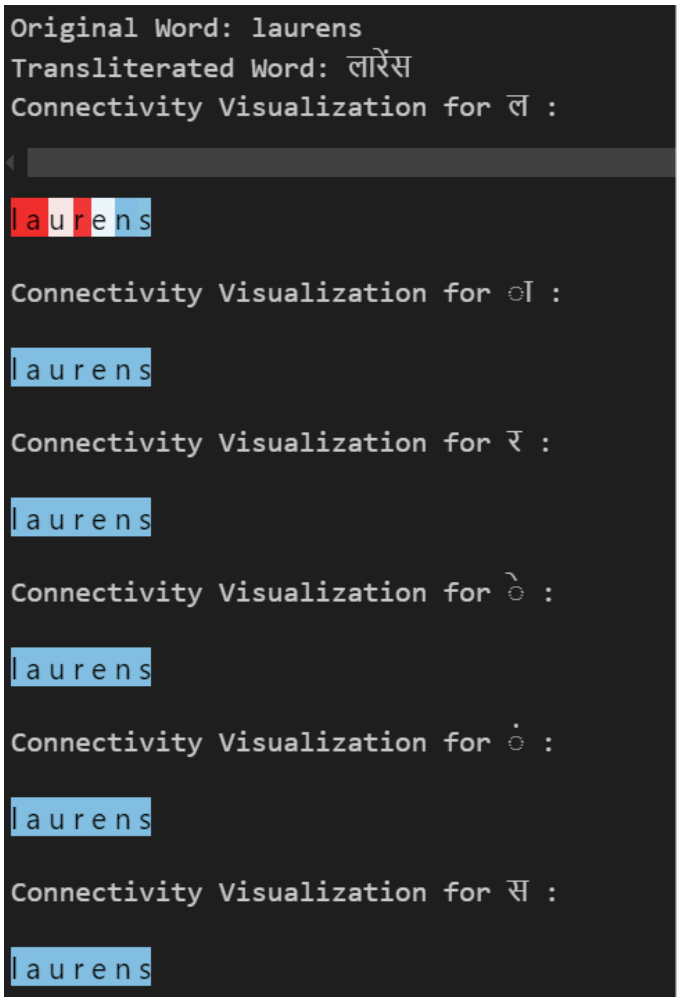

ब
ी
न
b
e
a
n
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

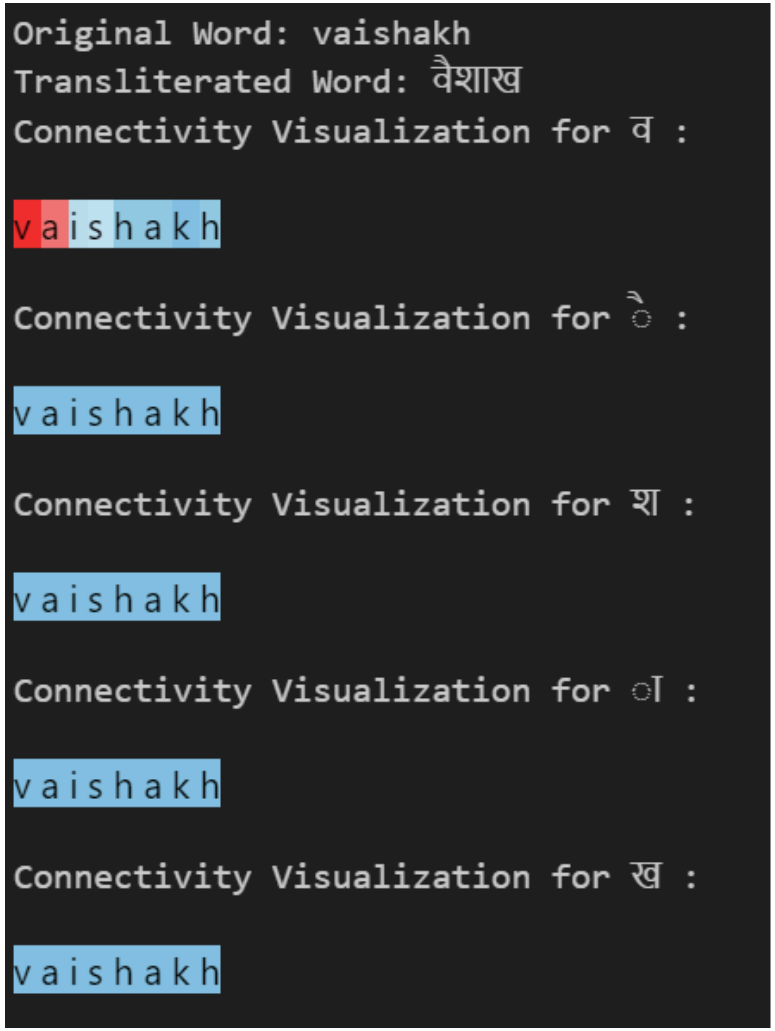


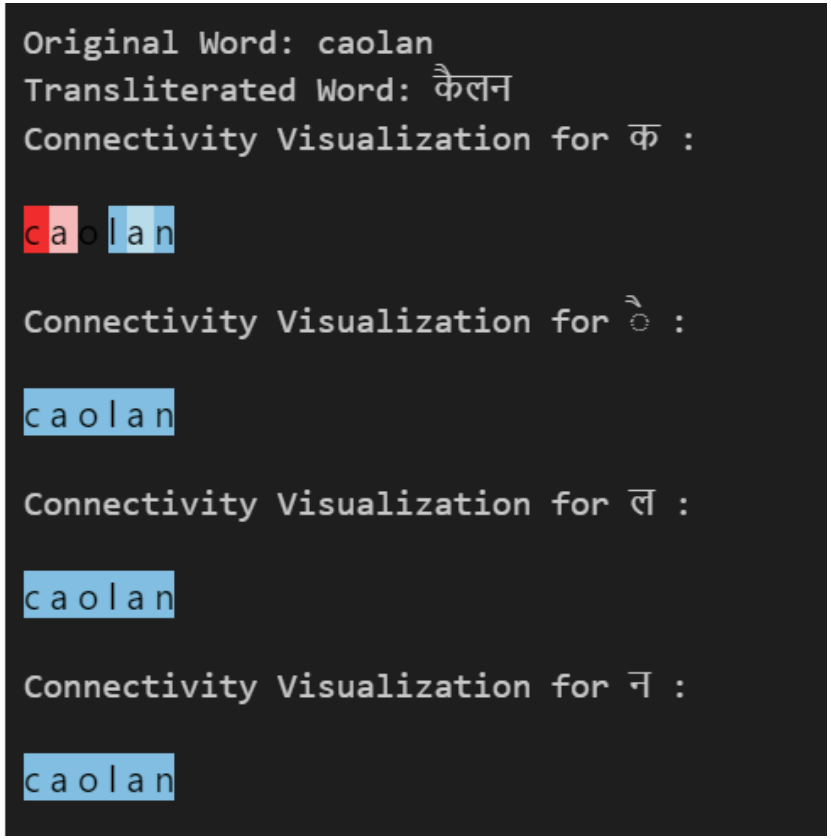
Question 6 (20 Marks)

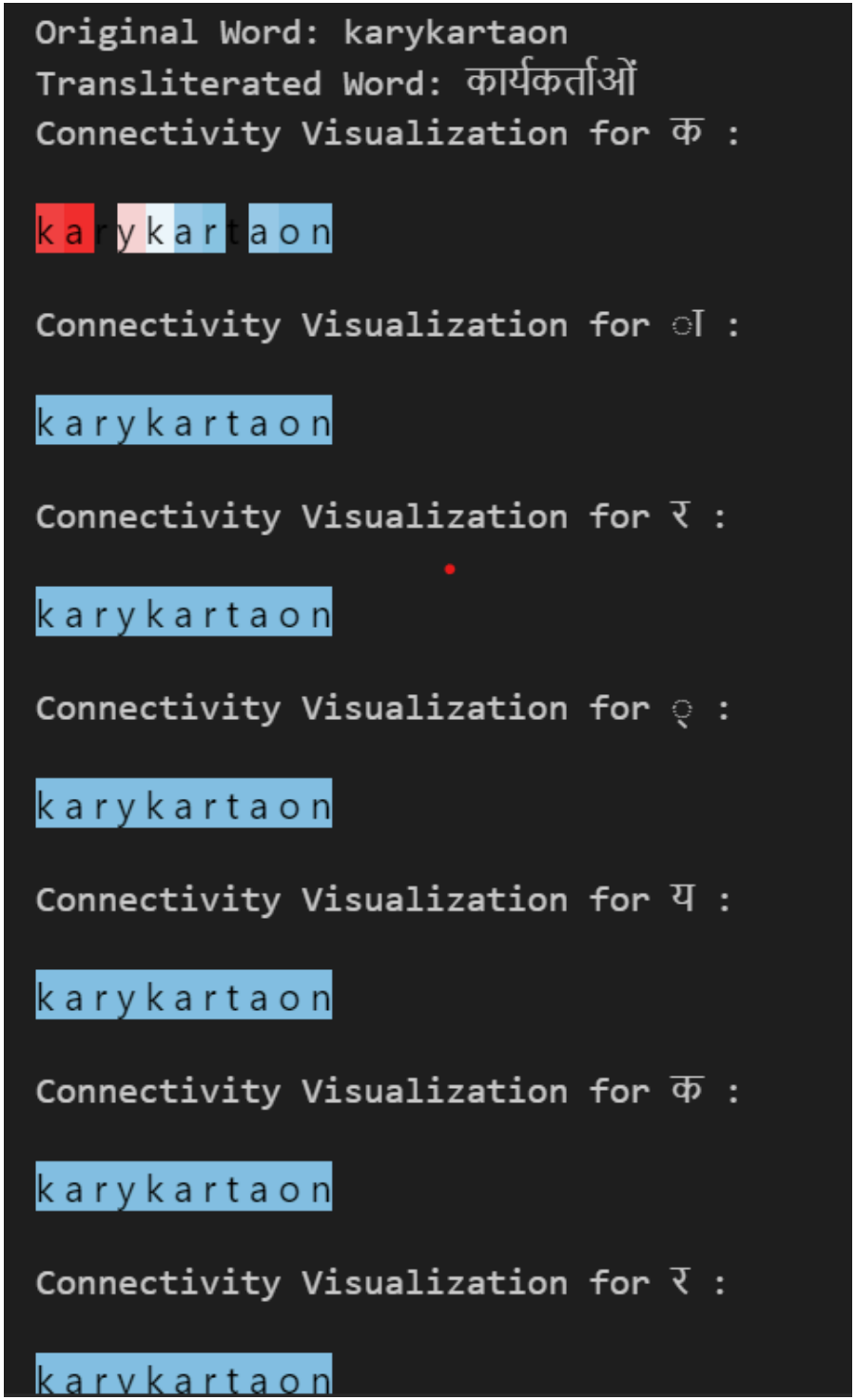
This a challenge question and most of you will find it hard.

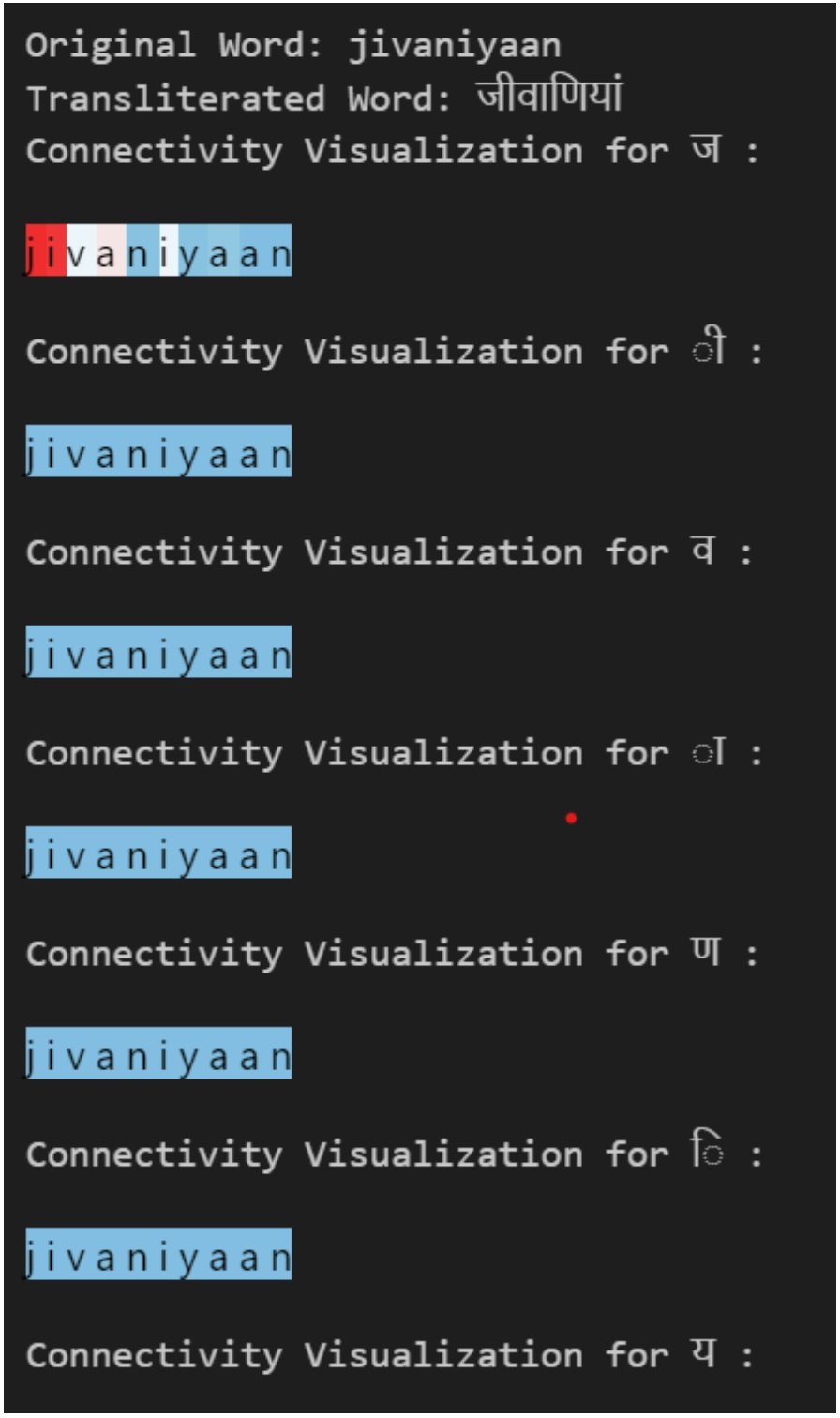
I like the visualisation in the figure captioned "Connectivity" in this article. Make a similar visualisation for your model. Please look at this blog for some starter code. The goal is to figure out the following: When the model is decoding the i-th character in the output which is the input character that it is looking at?


Original Word: laurens
Transliterated Word: लारेंस
Connectivity Visualization for ल :
laurens
Connectivity Visualization for ा :
laurens
Connectivity Visualization for र :
laurens
Connectivity Visualization for े :
laurens
Connectivity Visualization for ं :
laurens
Connectivity Visualization for स :
laurens


Original Word: vaishakh
Transliterated Word: वैशाख
Connectivity Visualization for व :
vaishakh
Connectivity Visualization for ै :
vaishakh
Connectivity Visualization for श :
vaishakh
Connectivity Visualization for ा :
vaishakh
Connectivity Visualization for ख :
vaishakh


Original Word: caolan
Transliterated Word: कैलन
Connectivity Visualization for क :
caolan
Connectivity Visualization for ै :
caolan
Connectivity Visualization for ल :
caolan
Connectivity Visualization for न :
caolan


Original Word: karykartaon
Transliterated Word: कार्यकर्ताओं
Connectivity Visualization for क :
karykartaon
Connectivity Visualization for ा :
karykartaon
Connectivity Visualization for र :
karykartaon
Connectivity Visualization for ् :
karykartaon
Connectivity Visualization for य :
karykartaon
Connectivity Visualization for क :
karykartaon
Connectivity Visualization for र :
karykartaon


Original Word: jivaniyaan
Transliterated Word: जीवाणियां
Connectivity Visualization for ज :
jivaniyaan
Connectivity Visualization for ी :
jivaniyaan
Connectivity Visualization for व :
jivaniyaan
Connectivity Visualization for ा :
jivaniyaan
Connectivity Visualization for ण :
jivaniyaan
Connectivity Visualization for ि :
jivaniyaan
Connectivity Visualization for य :

The function mapping the gradient magnitude to the colour first went through a MinMaxScaler transform from the sci-kit learn library, in order to get the weights between 0 and 1. Hence if all the gradients are zero then all colors will be the same. Looking at the connectivity heatmaps, we can see that the gradients have a tendency to vanish for later timesteps of producing the output. Note that what one would expect is that for the first output character the model would look at the first 1-2 characters and the ones after that for the second output character and so on. This is how humans would approach a transliteration task. From the heatmaps we can see that the model looks at the first 1-2 characters for the first output character, but this approach largely vanishes later into the word as the gradients start vanishing. Later on the input characters focused on don't really match with where humans would look during transliteration, indicating that through Backpropagation Through Time (BPTT) the model has learnt some other way of carrying out this task.

Question 7 (10 Marks)

Paste a link to your github code for Part A

Github link: https://github.com/Rajnishmaurya/da6401_assignment3

- We will check for coding style, clarity in using functions and a README file with clear instructions on training and evaluating the model (the 10 marks will be based on this).
- We will also run a plagiarism check to ensure that the code is not copied (0 marks in the assignment if we find that the code is plagiarised).
- We will check the number of commits made by the two team members and then give marks accordingly. For example, if we see 70% of the commits were made by one team member then that member will get more marks in the assignment (**note that this contribution will decide the marks split for the entire assignment and not just this question**).

- We will also check if the training and test splits have been used properly. You will get 0 marks on the assignment if we find any cheating (e.g., adding test data to training data) to get higher accuracy.

Self Declaration

I, Rajnish Maurya (Roll no: DA24M015), 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.