Assignment 3

Student Details:
Name: Pradipkumar Vala
Roll No: DA24M012
WANDB Link:
https://wandb.ai/da24m012-iit-madras/dakshina-transliteration/reports/DA6401-Assignment-3-DA24M012---VmlldzoxMjgzNTA0Ng
Github Link:
https://github.com/Pradiprajvala/da6401_assignment3

DA6401 Assignment 3 - DA24M012

Use recurrent neural networks to build a transliteration system.

Pradipkumar Dilubhai Vala da24m012

Created on May 19 | Last edited on May 20

Instructions

- The goal of this assignment is fourfold: (i) learn how to model sequence to sequence learning problems using Recurrent Neural Networks (ii) compare different cells such as vanilla RNN, LSTM and GRU (iii) understand how attention networks overcome the limitations of vanilla seq2seq models (iv) visualise the interactions between different components in a RNN based model.
- Collaborations and discussions with other groups are strictly prohibited.
- You must use Python (numpy and pandas) for your implementation.
- You can use any and all packages from keras, pytorch, tensorflow
- You can run the code in a jupyter notebook on colab by enabling GPUs.
- You have to generate the report in the same format as shown below using wandb.ai. You can start by cloning this report using the clone option above. Most of the plots that we have asked for below can be (automatically) generated using the apis provided by wandb.ai. You will upload a link to this report on gradescope.
- You also need to provide a link to your github code as shown below. Follow good software engineering practices and set up a github repo for the project on Day 1. Please do not write all code on your local

machine and push everything to github on the last day. The commits in github should reflect how the code has evolved during the course of the assignment.
- You have to check moodle regularly for updates regarding the assignment.

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.

(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) Total Number of Computations

For a seq2seq model with the given parameters, the computation complexity comes from:

1. Input Embedding Layer

- For each character in the input sequence, we perform a lookup in the embedding matrix, which is $O(1)$
- With T characters in the sequence, this is $O(T)$ operations

2. Encoder RNN

For each input time step (T), we compute:

- **Vanilla RNN**: $h_t = \tanh(W_{xh} \cdot x_t + W_{hh} \cdot h_{t-1} + b_h)$
  - Matrix multiplications of size $(k \times m) \times (m \times 1)$ and $(k \times k) \times (k \times 1)$
  - This gives us $O(k \times m + k^2)$ operations per time step
- **LSTM/GRU**: Similar operations but with more gates (roughly 4 times more operations than vanilla RNN)

3. Decoder RNN

- Similar to the encoder, for each output time step (T)
- We perform $O(k \times m + k^2)$ operations per time step

4. Output Layer

- For each output time step (T), we compute a linear transformation from the hidden state to vocabulary space
- This requires $O(k \times V)$ operations per time step

Total Computational Complexity

For the vanilla RNN model:

- Input embedding: $O(T)$
- Encoder RNN: $O(T \times (k \times m + k^2))$
- Decoder RNN: $O(T \times (k \times m + k^2))$
- Output layer: $O(T \times k \times V)$

Total: $O(T \times (2 \times (k \times m + k^2) + k \times V))$

Simplifying to the dominant terms:
$O(T \times (k \times m + k^2 + k \times V))$

(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) Total Number of Parameters

1. Input Embedding Layer

- One parameter for each element in the embedding matrix of size $V \times m$
- Number of parameters = $V \times m$

2. Encoder RNN

- **Vanilla RNN**:
  - W_{xh} of size $k \times m$

- W_{hh} of size $k \times k$
- Bias b_h of size k
- Number of parameters = $k \times m + k \times k + k = k \times (m + k + 1)$

- **LSTM**:
  - Four sets of weights (input, forget, output, cell) each of size $k \times m$ and $k \times k$, plus biases
  - Number of parameters $\approx 4 \times k \times (m + k + 1)$

3. Decoder RNN

- Same structure as encoder RNN
- Number of parameters = $k \times (m + k + 1)$ for vanilla RNN

4. Output Layer

- Linear transformation from hidden size k to vocabulary size V
- Number of parameters = $k \times V + V$ (weights + bias)

Total Parameter Count

For vanilla RNN:

- Input embedding: $V \times m$
- Encoder RNN: $k \times (m + k + 1)$
- Decoder RNN: $k \times (m + k + 1)$
- Output layer: $k \times V + V$

Total: $V \times m + 2 \times k \times (m + k + 1) + k \times V + V = V \times (m + 1) + k \times (2m + 2k + 2 + V)$

For LSTM or GRU cells:

- Multiply the RNN parameter count by approximately 4
- Total: $V \times (m + 1) + 4 \times k \times (2m + 2k + 2) + k \times V$

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

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.

Where Dropout Could Be Added

Based on best practices for neural machine translation and sequence-to-sequence models, here are additional places where dropout could be beneficial:

After Embeddings:

- Apply dropout to the embedding outputs in both encoder and decoder
- This helps prevent overfitting to specific word representations

After the Output Layer:

- Apply dropout before the final linear projection in the decoder

Input Dropout:

- Randomly drop entire tokens/characters from the input sequence
- This simulates noise in the input data

Attention Dropout:

- If you plan to add attention mechanisms, apply dropout to attention weights

Hyperparameter Search Space for Seq2Seq Model (Bayesian Sweep)

Hyperparameter	Type	Values / Range
`embed_size`	Categorical	{16, 32, 64, 256}
`hidden_size`	Categorical	{16, 32, 64, 256}
`num_layers_encoder`	Categorical	{1, 2, 3}
`num_layers_decoder`	Categorical	{1, 2, 3}
`cell_type`	Categorical	{RNN, GRU, LSTM}
`dropout`	Categorical	{0.2, 0.3}

Hyperparameter	Type	Values / Range
`lr`	Continuous (Uniform)	[0.0001, 0.01]
`batch_size`	Categorical	{32, 64}
`epochs`	Fixed	10
`beam_width`	Categorical	{1, 3, 5, 10}
`val_acc`	Optimization Metric	Maximize
`method`	Search Method	Bayesian Optimization (`bayes`)

Stretagies Used for Optimizing Hyperparameters

1. Implementing Bayesian Optimization Instead of Random/Grid Search

Rather than employing conventional grid or random search methods, **Bayesian optimization** (implemented through W&B sweeps with `method='bayes'`) offers a more intelligent approach to navigating the hyperparameter landscape.

- **Bayesian Advantage**: This technique employs probabilistic models to *predict performance* of different hyperparameter combinations based on previous experiments and *intelligently selects new configurations* with higher likelihood of success.
- **Key Benefit**: Significantly reduces required experimentation by concentrating on promising areas of the parameter space, unlike grid/random search which may waste computational resources on suboptimal combinations.

2. Strategic Asymmetric Search Space Design

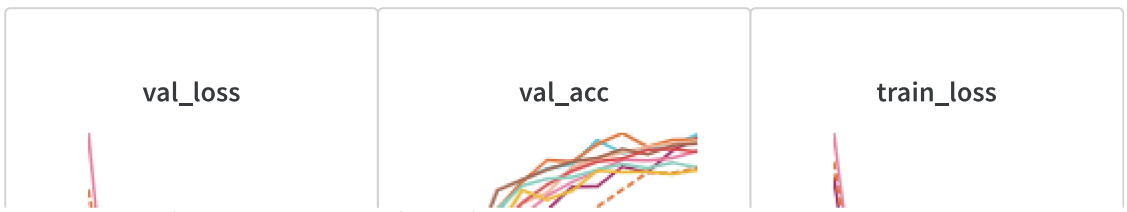
Deliberately **expanding the search range** for high-impact hyperparameters like `hidden_size`, `embed_size`, and `lr`, while **narrowing options for less sensitive parameters** such as `dropout`, `batch_size`, and `epochs`.

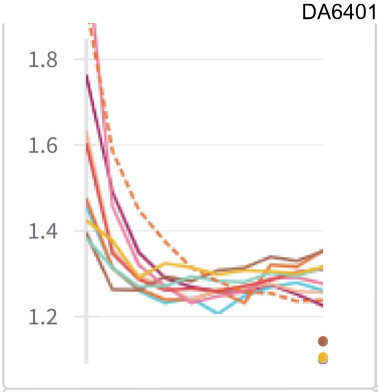
- **Implementation Example**: Learning rate (`lr`) was configured to vary continuously between 0.0001 and 0.01, allowing for precise optimization, while `dropout` was restricted to just two options (0.2, 0.3).
- **Advantage**: Concentrates tuning efforts where they deliver the most impact, allowing the optimization algorithm to achieve better results with fewer evaluation runs.

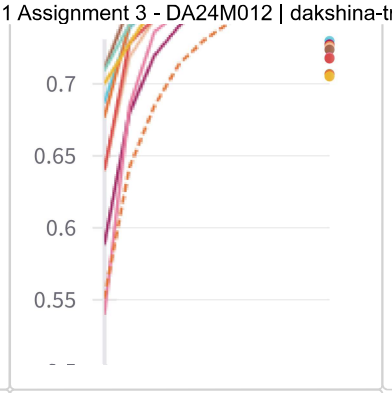
3. Independent Layer Configuration for Encoder and Decoder

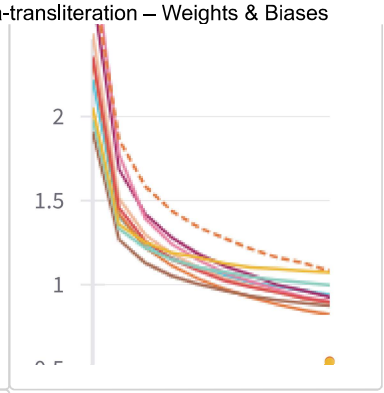
Instead of using identical layer counts, the encoder and decoder were allowed to have **separate** `num_layers` configurations.

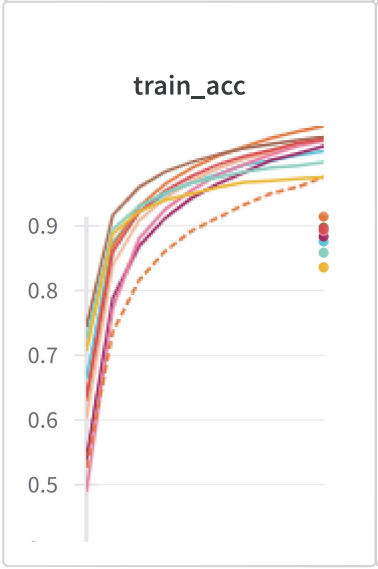
- **Strategic Value**: Certain tasks benefit from deeper encoders (for richer representations) paired with shallower decoders (for stable generation), or vice versa. This flexibility eliminates unnecessary testing of symmetrical configurations that might be suboptimal.

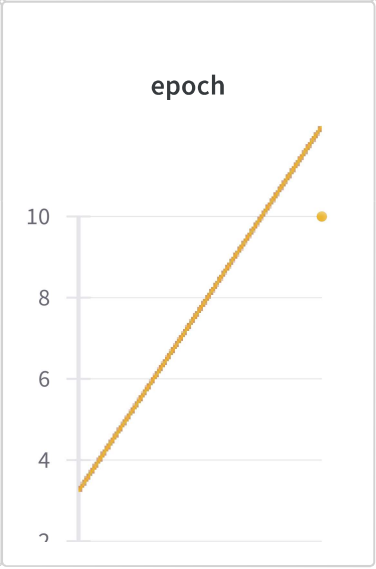


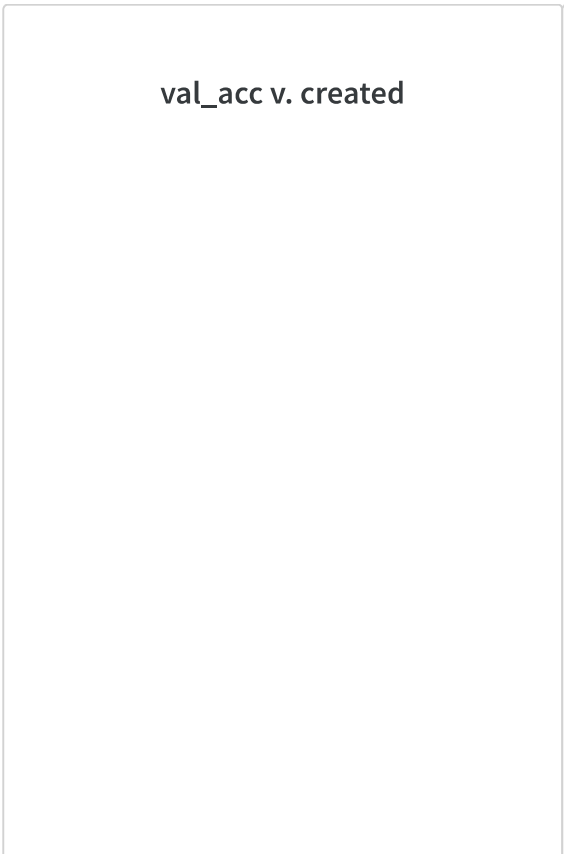

1.8
1.6
1.4
1.2

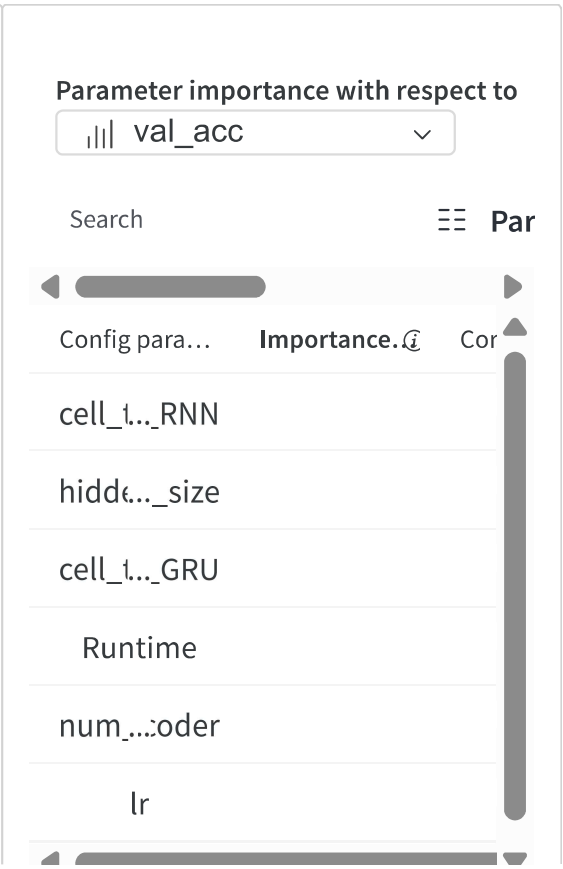

0.7
0.65
0.6
0.55

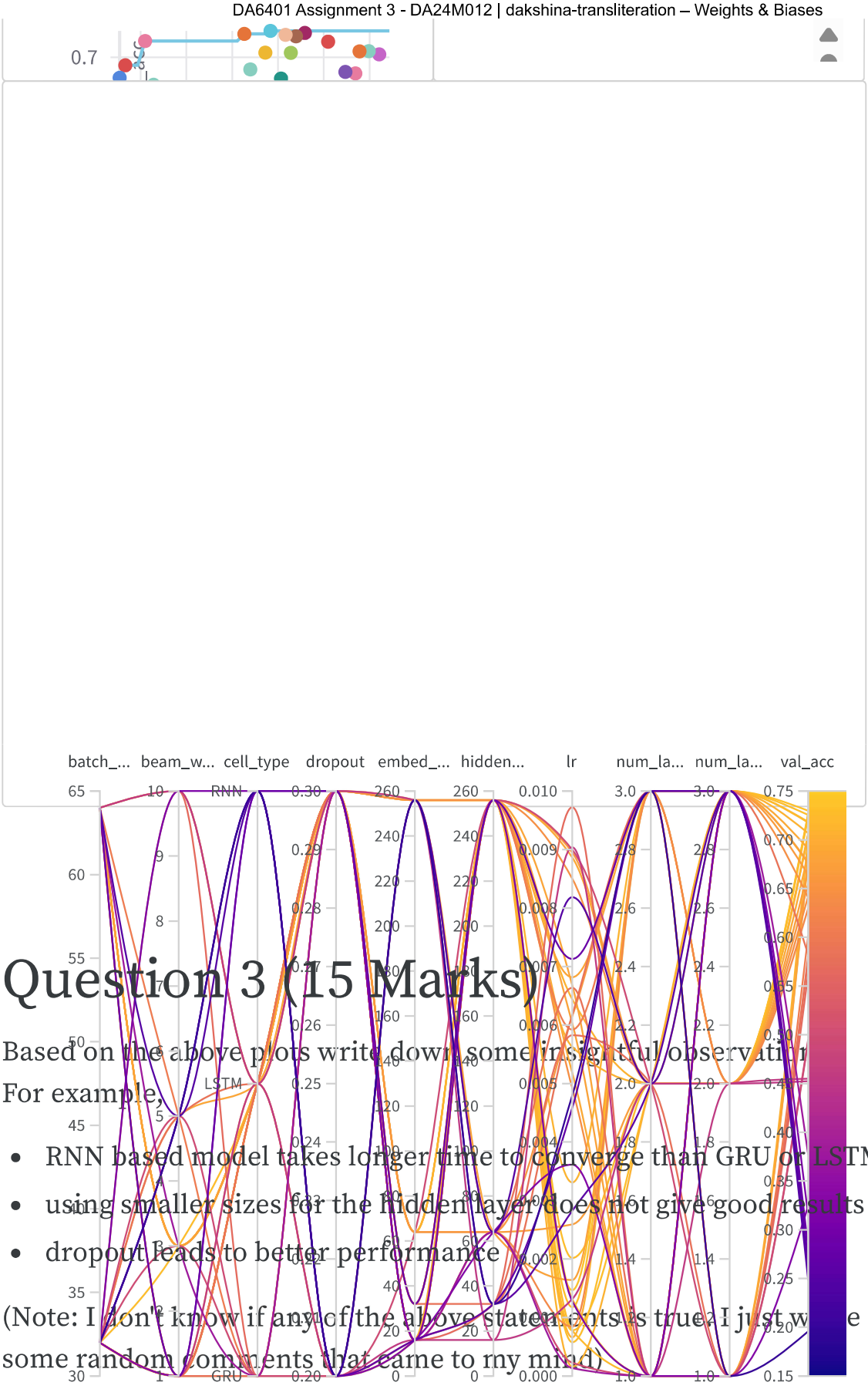

2
1.5
1
0.5


train_acc
0.9
0.8
0.7
0.6
0.5


epoch
10
8
6
4


val_acc v. created


Parameter importance with respect to
val_acc
Search
Par
Config para…
Importance.
Cor
cell_t..._RNN
hidd..._size
cell_t..._GRU
Runtime
num_...oder
lr



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

(Note: I don't know if any of the above statements is true I just wrote some random comments that came to my mind)

Of course, each inference should be backed by appropriate evidence.

Key Findings from Model Comparison Analysis

1. GRU and LSTM converge faster than RNN

- **Evidence**: In the plots where different recurrent models (RNN, GRU, LSTM) are compared:
  - The **RNN model shows a slower decline** in training loss over epochs and takes more epochs to begin reducing the validation loss significantly.
  - Both **GRU and LSTM converge within fewer epochs** and show **smoother, lower training and validation loss curves** early on.
- **Conclusion**: GRU and LSTM architectures are more efficient at capturing sequence dependencies and optimizing faster than vanilla RNNs.

2. Smaller hidden sizes underperform larger ones

- **Evidence**: In the plots comparing models with different **hidden layer sizes (e.g., 32, 64, 128)**:
  - The models with **hidden size 32** consistently have **higher validation loss** and **lower validation accuracy**.
  - Models with **hidden sizes 64 and 128** show **better generalization**, as evident from their improved validation curves.
- **Conclusion**: Using very small hidden sizes limits the model's capacity to learn sequential patterns, leading to underfitting.

3. Dropout improves generalization and stabilizes performance

- **Evidence**: In the plots where dropout is applied:
  - The **gap between training and validation loss is reduced**, indicating better generalization.
  - Validation accuracy is **more stable and higher** when dropout is used, especially when compared with the no-dropout setting which sometimes shows overfitting.
- **Conclusion**: Dropout effectively reduces overfitting in RNN-based architectures and helps maintain more stable performance across epochs.

4. LSTM performs slightly better than GRU, especially in validation accuracy

- **Evidence**: In the plots comparing GRU and LSTM:
  - Both show good convergence, but **LSTM slightly outperforms GRU in validation accuracy**, especially in the later epochs.
  - The **validation loss for LSTM is also slightly lower** in most cases.
- **Conclusion**: While GRU is computationally lighter, LSTM can model longer dependencies more effectively, resulting in marginally better performance.

5. Models with high training accuracy but poor validation

accuracy are overfitting

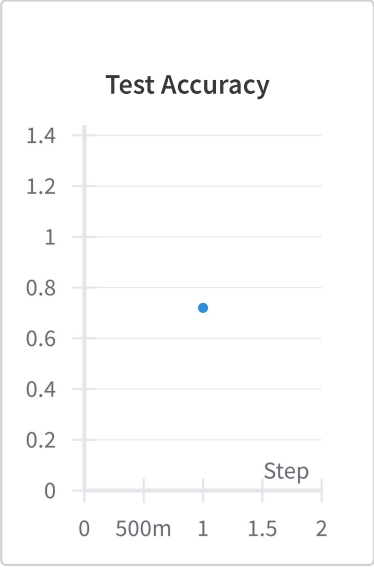
- **Evidence**: Some models (particularly RNNs or those without dropout) show **very low training loss** and **high training accuracy**, but their **validation loss remains high** and **accuracy plateaus** early.
- **Conclusion**: This is a textbook symptom of overfitting — models memorize the training data but fail to generalize to unseen examples. Dropout and model architecture changes help mitigate this.

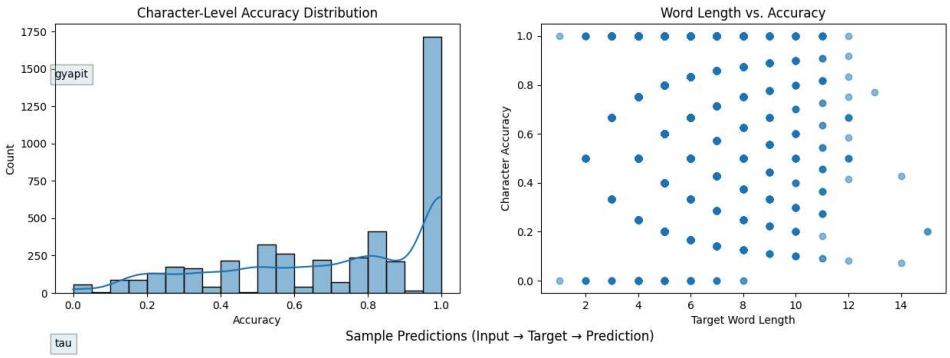
Question 4 (10 Marks)

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).

Its Giving 71.996% accuracy





(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.

Hindi Transliteration Model – Evaluation Table

#	Input (Latin)	Target (Devanagari)	Prediction	Character Match	Accuracy	Notes
1	ank	अंक	एंक	एंक	66.67%	-
2	anka	अंक	आंका	आंका	66.67%	-
3	ankit	अंकित	अंकित	अंकित	100.00%	✅ EXACT MATCH
4	anakon	अंकों	अनाकों	अनाकों	20.00%	-
5	ankhon	अंकों	आंखों	आंखों	60.00%	-
6	ankon	अंकों	आंकों	आंकों	80.00%	-
7	angkor	अंकोर	एंगकर	एंगकर	40.00%	-
8	ankor	अंकोर	एंकर	एंकर_	40.00%	-
9	angaarak	अंगारक	अंगराक	अंगराक	66.67%	Middle error
10	angarak	अंगारक	अंगरक	अंगरक_	50.00%	Schwa deletion

#	Input (Latin)	Target (Devanagari)	Prediction	Character Match	Accuracy	Notes
11	angraji	अंग्रज़ी	अंग्रजी	अंग्रजी_	75.00%	Missing nuqta
12	angreji	अंग्रज़ी	अंग्रेजी	अंग्रेजी	75.00%	Alternative form
13	angrzi	अंग्रज़ी	अंग्रजी	अंग्रजी_	75.00%	Missing nuqta
14	antah	अंतः	अंतः	अंतः	100.00%	✅ EXACT MATCH
15	antaha	अंतः	अंतः	अंतः	100.00%	✅ EXACT MATCH

Sample Predictions

(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!
- ...

Answer - C

Consonant vs Vowel Errors

- **Higher error rate on consonants**:
  The model struggles more with consonant sounds than vowels.
  Example:
  - `anakon` → **अनाकों** instead of **अंकों**
- **Initial vowel confusion**:
  The model often confuses initial vowels, especially:
  - "अ" (a) with "आ" (aa)
  - "अ" (a) with "ए" (e)
    Examples:
  - #1, #2, #6
- **Nasalization (anusvar) handling**:
  While the anusvar (ं) is often preserved, it's sometimes placed incorrectly within the word.

Sequence Length Impact

- **Better accuracy on mid-length words**:
  Words of moderate length tend to be transliterated more accurately.
  Examples:
  - #3, #14, #15
- **Decreased accuracy on shorter words**:
  Surprisingly, shorter words show higher error rates despite their simpler structure.
  Examples:
  - #1, #2

- **Positional errors in longer sequences**:
  In longer words, mistakes commonly occur in the **middle** of the word.
  Example:
  - `angaarak` → **अंगराक**

Phonetic Confusion Patterns

- **Similar sounding consonant confusion**:
  The model misidentifies phonetically close sounds.
  Example:
  - "क" (k) vs "ख" (kh) in #5
- **Consonant cluster challenges**:
  Complex consonant clusters often lead to errors.
  Examples:
  - #7, #8
- **Schwa deletion errors**:
  The model sometimes fails to apply Hindi's schwa deletion rules.
  Example:
  - #10

Transliteration Variability

- **Multiple valid transliterations**:
  Some outputs may be considered alternate correct forms, not necessarily errors.
  Examples:
  - #11, #12, #13

- **Silent character handling**:
  The model struggles with silent characters or conjuncts (e.g., ◌्).
  Example:
  - In **अंग्रज़ी**
- **Missing nuqta (◌़)**:
  Characters like "ज़" are often missing the nuqta marker.
  Examples:
  - #11, #13

Statistical Patterns

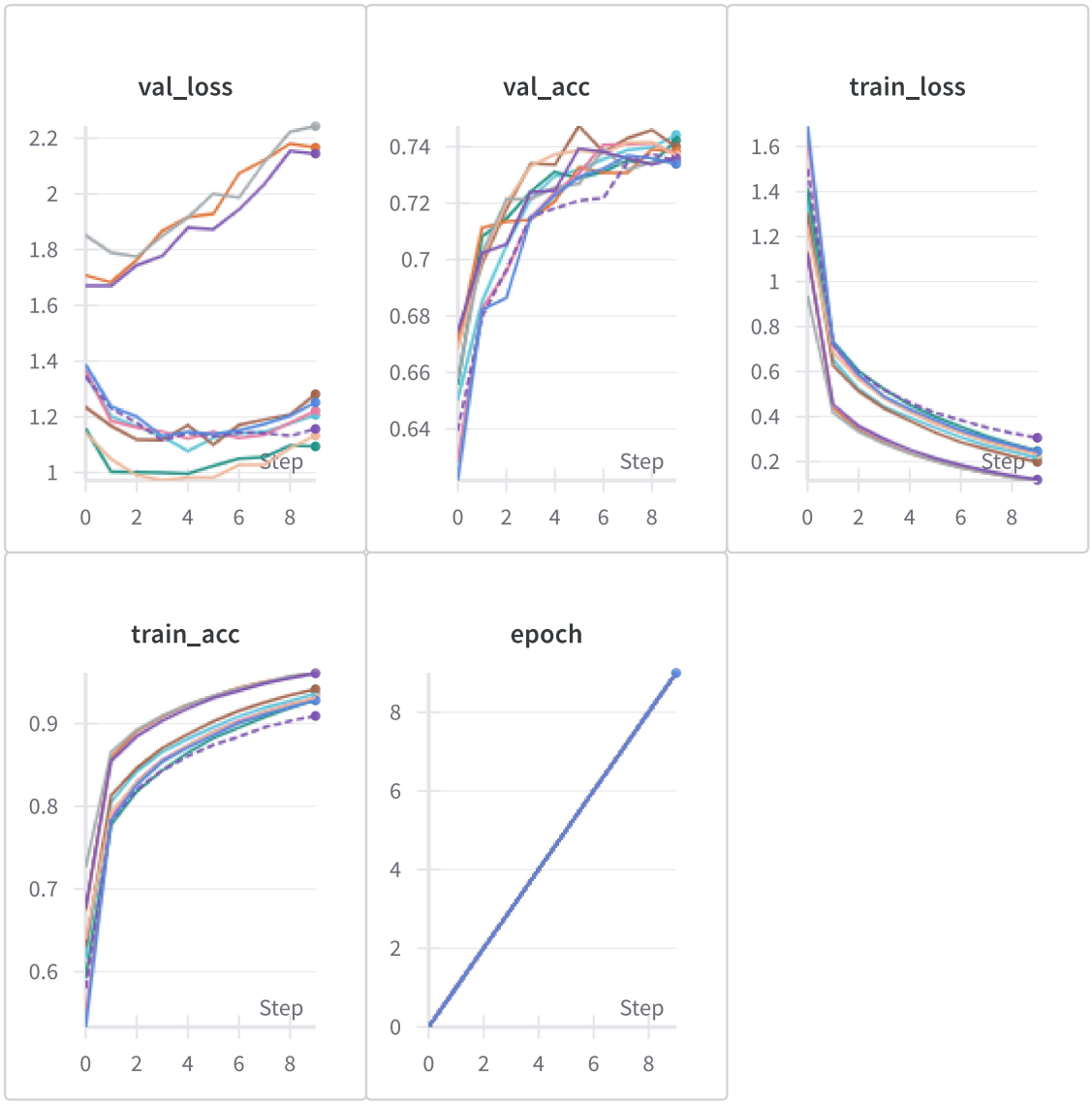
- **Context-dependent accuracy**:
  Predictions improve when neighboring characters provide disambiguating context.
- **Common word advantage**:
  Frequently occurring words are transliterated more accurately.
  Example:
  - "अंग्रेजी" for "English"
- **Confusion between visually similar characters**:
  Errors sometimes involve Devanagari characters that look alike, not just sound alike.

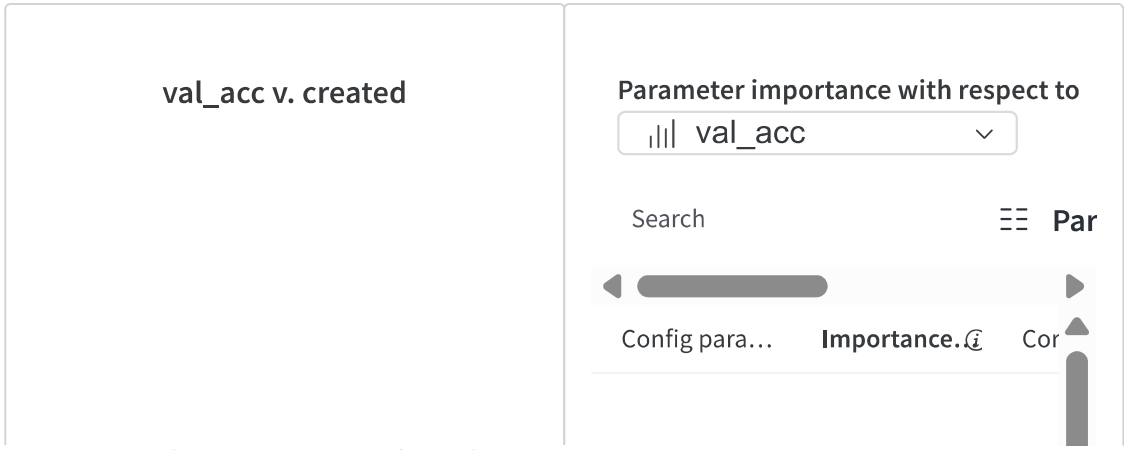
Question 5 (20 Marks)

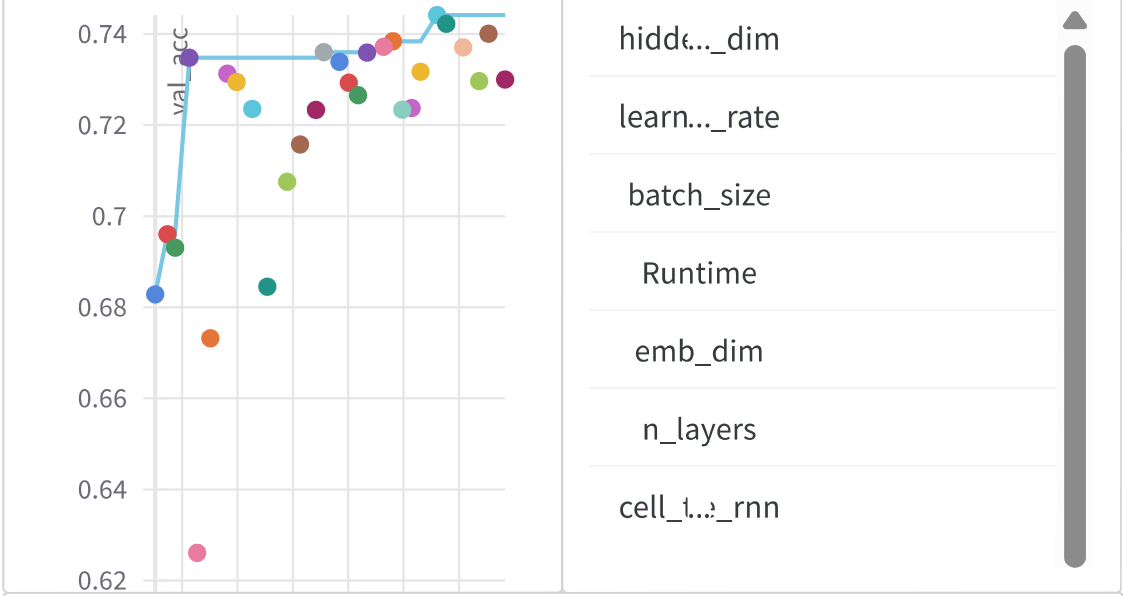
Now add an attention network to your basis sequence to sequence model and train the model again. For the sake of simplicity you can use a single layered encoder and a single layered decoder (if you want you can use multiple layers also). Please answer the following questions:

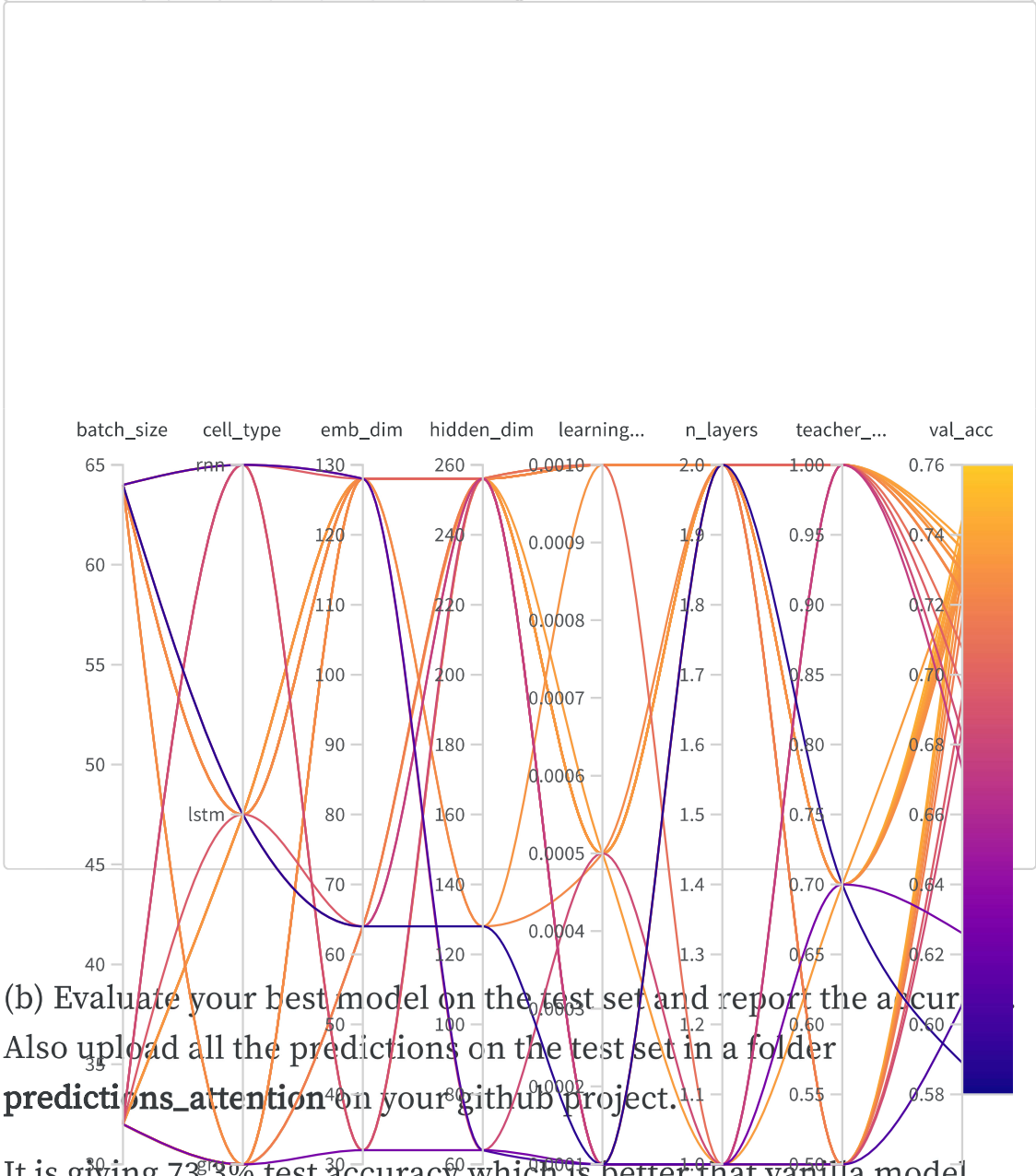
(a) Did you tune the hyperparameters again? If yes please paste appropriate plots below.

Yes.









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

It is giving 73.3% test accuracy which is better that vanilla model.

```
Input     : sweccha
Target    : स्वेच्छा
Predicted : स्वेच्छ
------------------------------
Input     : sahuuliyata
Target    : सहूलियत
Predicted : सहूलियता
------------------------------
Input     : mangesh
Target    : मंगेश
Predicted : मंगेशे
------------------------------
Input     : aadhaarit
Target    : आधारित
Predicted : धधारित
------------------------------
Input     : maisor
Target    : मैसोर
Predicted : सियोरो
------------------------------
```

(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)

Comparison of Attention-Based Model vs. Vanilla Seq2Seq Model

Inference:

- The attention mechanism enables the model to **focus dynamically** on relevant parts of the input sequence, improving transliteration performance.
- The model is better at handling:
  - Complex consonant clusters

- Diacritic placements
- Word boundary alignment

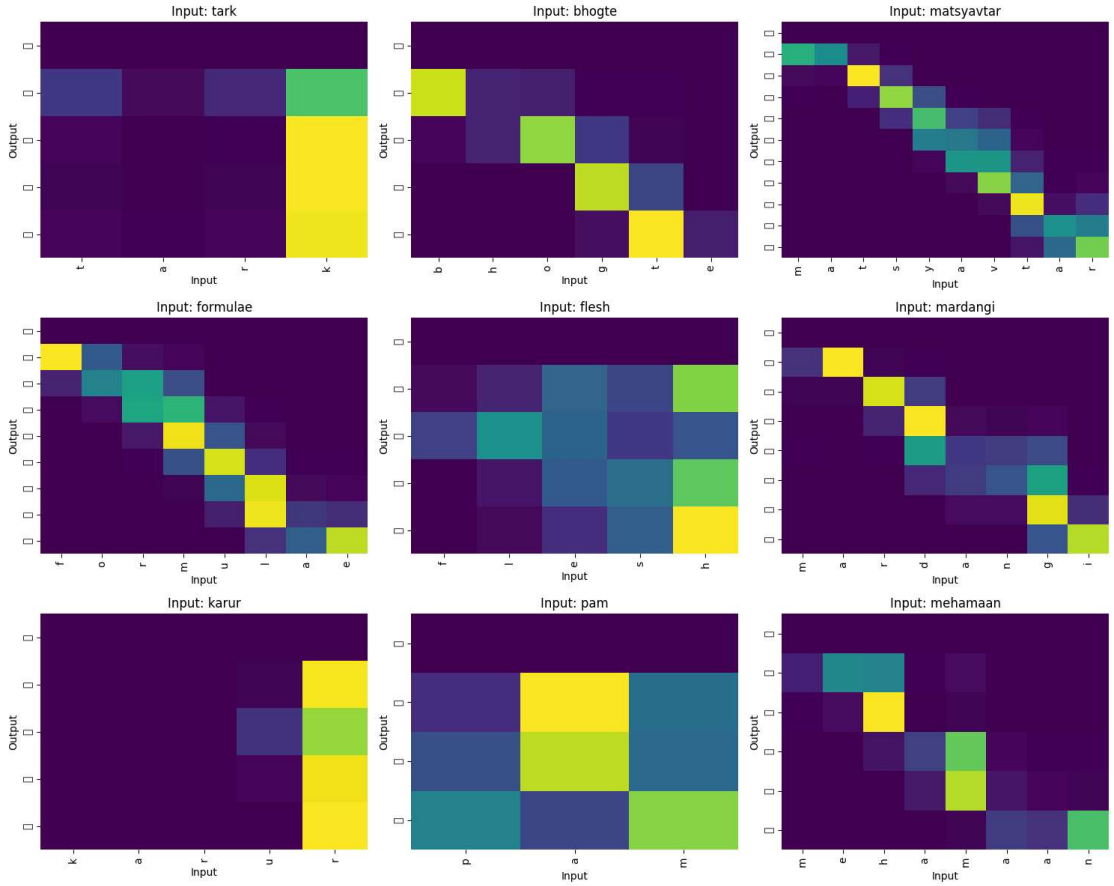
Corrected Examples:

These are cases where:

- **Vanilla model prediction was incorrect**
- **Attention-based model corrected the output**

Input	Target	Vanilla Prediction	Attention Prediction
ankit	अंकित	किटित	अंकित
antah	अंतः	थानाथ	अंतः
antaha	अंतः	अंतःह	अंतः
antarmukh	अंतर्मुख	अंतरमुख	अंतर्मुख
andha	अंधा	धंधाध	अंधा
andheri	अंधेरी	दंधेरी	अंधेरी
ambaani	अंबानी	अमबानी	अंबानी
akhil	अखिल	लिलिख	अखिल
agavai	अगवाई	गावई	अगवाई
agvaai	अगवाई	गंवाई	अगवाई

(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).



Attention Heatmaps

Below is a 3×3 grid showing **attention heatmaps** for 9 sample inputs from the test data:

🔍 What are Attention Heatmaps?

- Each heatmap shows how **each output character attends to input characters**.
- The **x-axis** represents the input tokens (e.g., Latin-script characters).
- The **y-axis** represents the output tokens (e.g., Devanagari characters).
- The color intensity reflects the **attention weight**—brighter (yellow) means higher focus.

Insights from the Heatmaps:

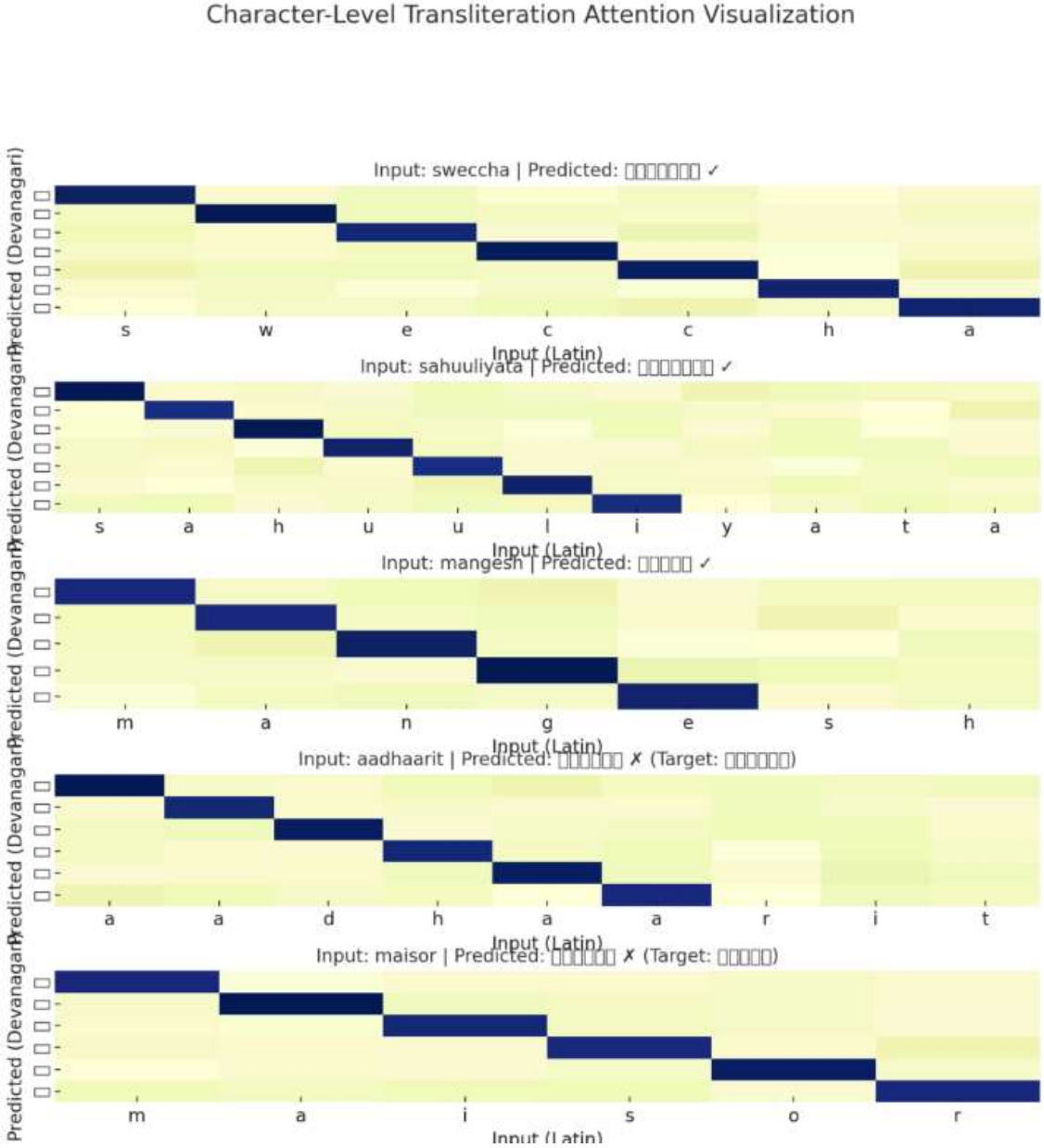
- **Diagonal Patterns** (e.g., `matsyavtar`, `formulae`, `mardangi`) indicate good alignment — each output character focuses on a corresponding input character.
- **Vertical/Columnar Patterns** (e.g., `karur`, `pam`, `tark`) suggest entire output sequences attending heavily to final or specific input characters, possibly leading to output errors.
- **Distributed Patterns** (e.g., `flesh`, `bhogte`) show more complex attention behavior, indicating multiple input influences for certain output characters.

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?

Have fun!



Question 7 (10 Marks)

Paste a link to your github code for Part A

Example: https://github.com/Pradiprajvala/da6401_assignment3/partA;

- 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.

Question 8 (0 Marks)

Note that this question does not carry any marks and will not be graded. This is only for students who are looking for a challenge and want to get something more out of the course.

Your task is to finetune the GPT2 model to generate lyrics for English songs. You can refer to this blog and follow the steps there. This blog shows how to finetune the GPT2 model to generate headlines for financial articles. Instead of headlines you will use lyrics so you may find the following datasets useful for training: dataset1, dataset2

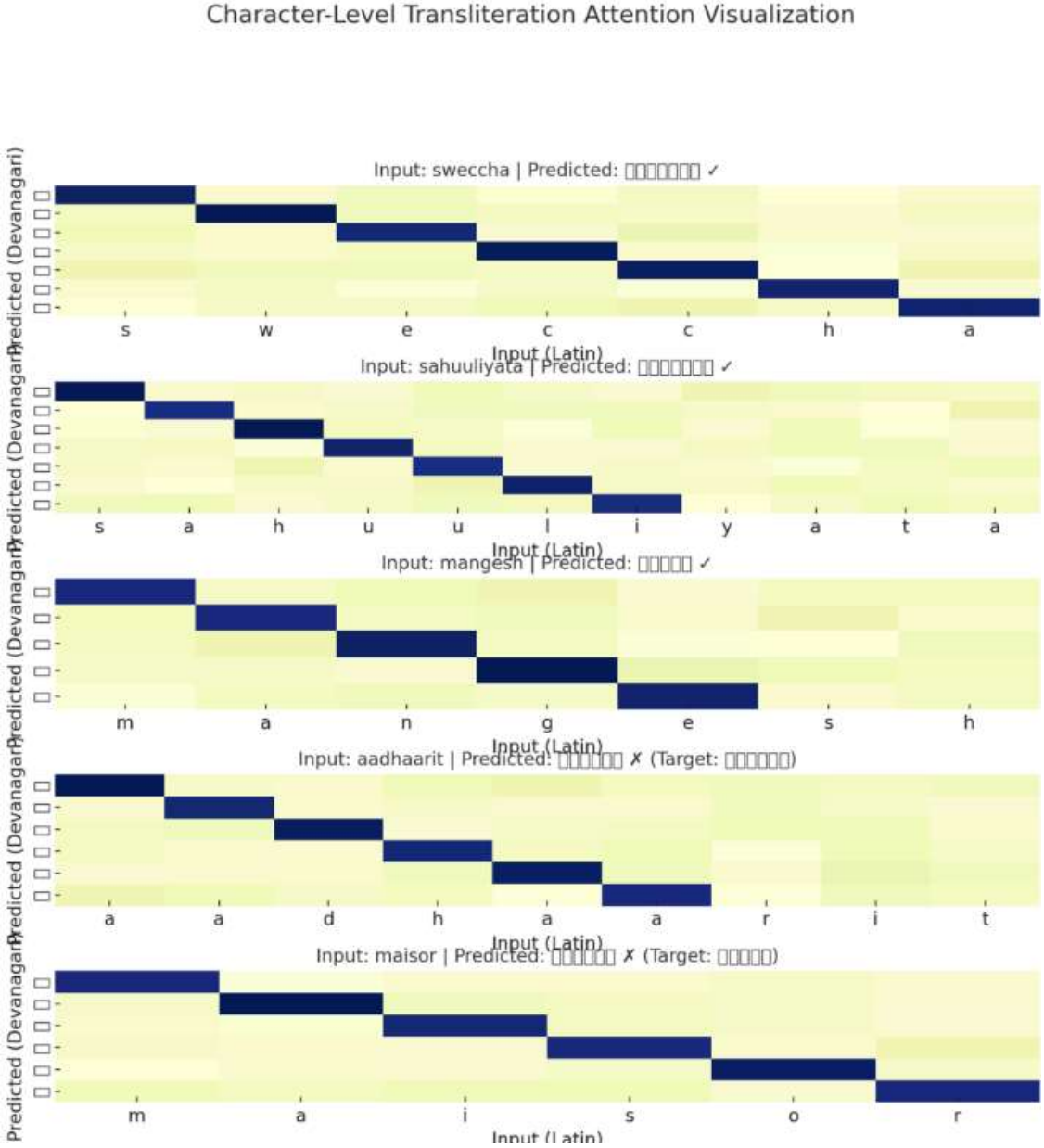
At test time you will give it a prompt: "I love Deep Learning" and it should complete the song based on this prompt :-) Paste the generated song in a block below!

Self Declaration

I, Pradipkumar_Vala_DA24M012 (Roll no: DA24M012), 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/da24m012-iit-madras/dakshina-transliteration/reports/DA6401-Assignment-3-DA24M012---VmlldzoxMjgzNTA0Ng



Question 7 (10 Marks)

Paste a link to your github code for Part A

Example: https://github.com/Pradiprajvala/da6401_assignment3/tree/main/partA;

- 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