Assignment 3

Student Details: Name: Sudhanva Satish Roll No: DA24M023

WANDB Link: https://wandb.ai/da24m023-indian-institute-of-technology-madras/DL_A3/reports/DA6401-Assignment-3-Transliteration-System--VmlldzoxMjgyNTE1Ng?accessToken=ir2v79i5ek8c60jqo6fqdhs52pgffpag8fjxq5espeaqqd5sw464y4g4269gtfjr

Github Link: https://github.com/suddu21/Deep-Learning-Assignment-3

Submission Instructions : kindly format your doc like this page (with your details and links) and then submit it.

Important Instructions:

- Students **must follow the updated submission format strictly**. Non-compliance will result in penalties.
- **Submitting non-modular, unstructured, or unreadable code will cause penalties**.
- Please upload all your `.py` files and the original `.ipynb` notebooks to GitHub, along with a detailed README file that explains the assignment structure, functionality, and instructions for running the code. Any deviation from it will cause penalties.
- You will be heavily penalized if we can't access your GitHub repo and W&B links.
- Any plagiarism will be reported and heavily penalized.
- **Note that the deadlines are stringent for this assignment, as grades need to be finalized before the institute-mandated schedule.**

DA6401 Assignment 3 - Transliteration System

Use recurrent neural networks to build a transliteration system.

Sudhanva Satish da24m023

Created on May 18 | Last edited on May 20

Instructions

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)

(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)

Answer

Model Assumptions

- **Embedding dimension**: m
- **Hidden dimension**: k
- **Sequence length**: T
- **Vocabulary size**: V
- **Architecture**: Single-layer encoder and decoder
- **Sequence**: Equal-length input and output sequences

---

1. Parameter Breakdown

The model consists of embedding, recurrent, and output layers. Below, we define the parameters for each component.

Embedding Layer

- Parameters: $V \cdot m$ (vocabulary size × embedding dimension)

Recurrent Layer Matrices

- **Matrix U (input-to-hidden)**: $k \cdot m$
- **Matrix W (hidden-to-hidden)**: k^2

- **Bias (hidden)**: k

Output Layer

- **Matrix V (hidden-to-output)**: $V \cdot k$
- **Bias (output)**: V

---

2. Simple RNN Model

Total Parameters

For a Simple RNN, the encoder and decoder each have one recurrent layer.

- **Encoder**:
  - Embedding: $V \cdot m$
  - Matrix U: $k \cdot m$
  - Matrix W: k^2
  - Bias: k
  - Total: $V \cdot m + k \cdot m + k^2 + k$
- **Decoder**:
  - Embedding: $V \cdot m$
  - Matrix U: $k \cdot m$
  - Matrix W: k^2
  - Matrix V: $V \cdot k$
  - Bias (hidden): k
  - Bias (output): V
  - Total: $V \cdot m + k \cdot m + k^2 + V \cdot k + k + V$

Total parameters:

$$\boxed{2V \cdot m + 2k \cdot m + 2k^2 + V \cdot k + 2k + V}$$

Total Computations

Parameters are shared across timesteps. The encoder processes T timesteps, and the decoder processes T timesteps.

- **Complexity per timestep**: $O(V \cdot m + k \cdot m + k^2 + k)$ for encoder; $O(V \cdot m + k \cdot m + k^2 + V \cdot k + k + V)$ for decoder
- **Total complexity**:

$$\boxed{O((2T+1)(V \cdot m + k \cdot m + k^2 + k) + (T)(V \cdot k + V))}$$

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).
- 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.

Answer

I have chosen the Hindi language for this project. Below are the hyperparameter value combos I explored during the sweeps, along with the strategy employed to minimize the number of runs.

Hyperparameter Ranges

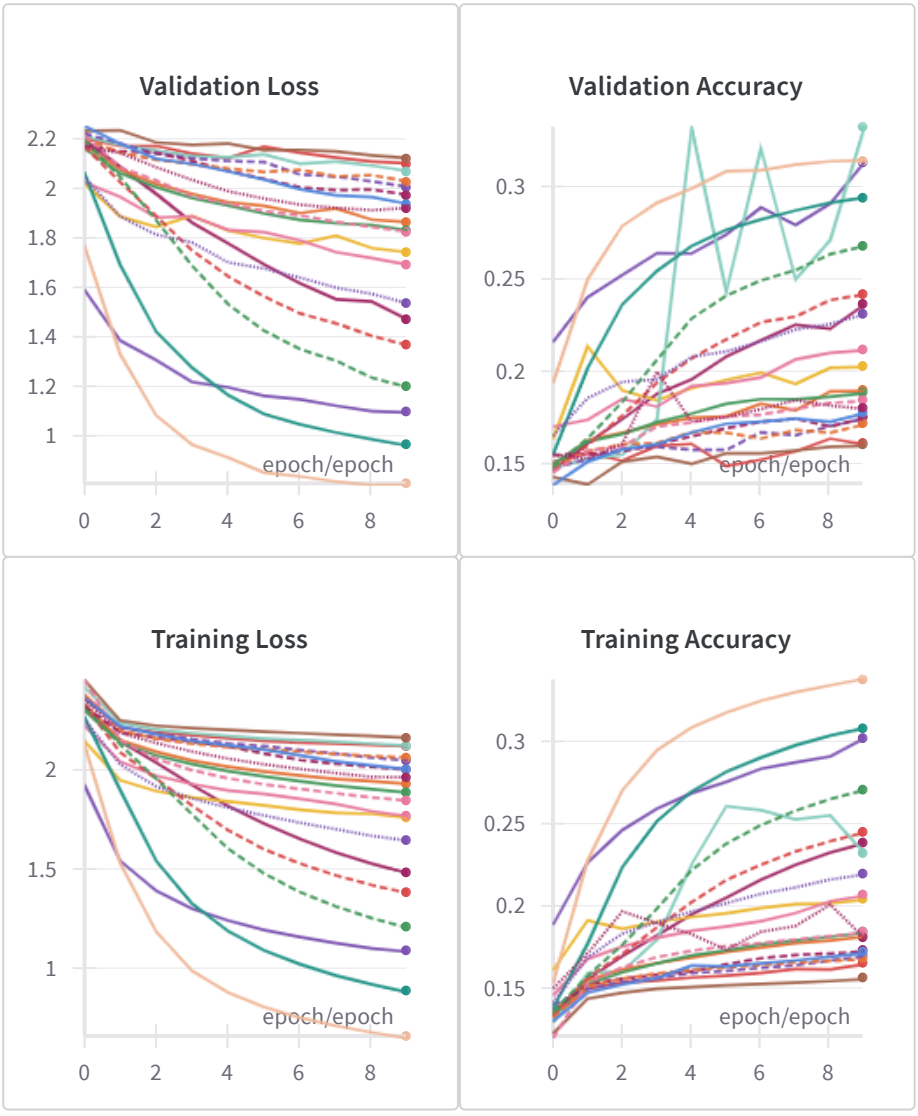
```
Input embedding size: [16, 32, 64, 256]
Hidden dimension: [16, 32, 64, 256]
Cell type: [RNN, GRU, LSTM]
Number of encoder layers: [1, 2, 3]
Number of decoder layers: [1, 2, 3]
Dropout rate: [0.2, 0.3]
Batch size: [32, 64]
Epochs: 10
Beam size: [1, 3, 5]
```

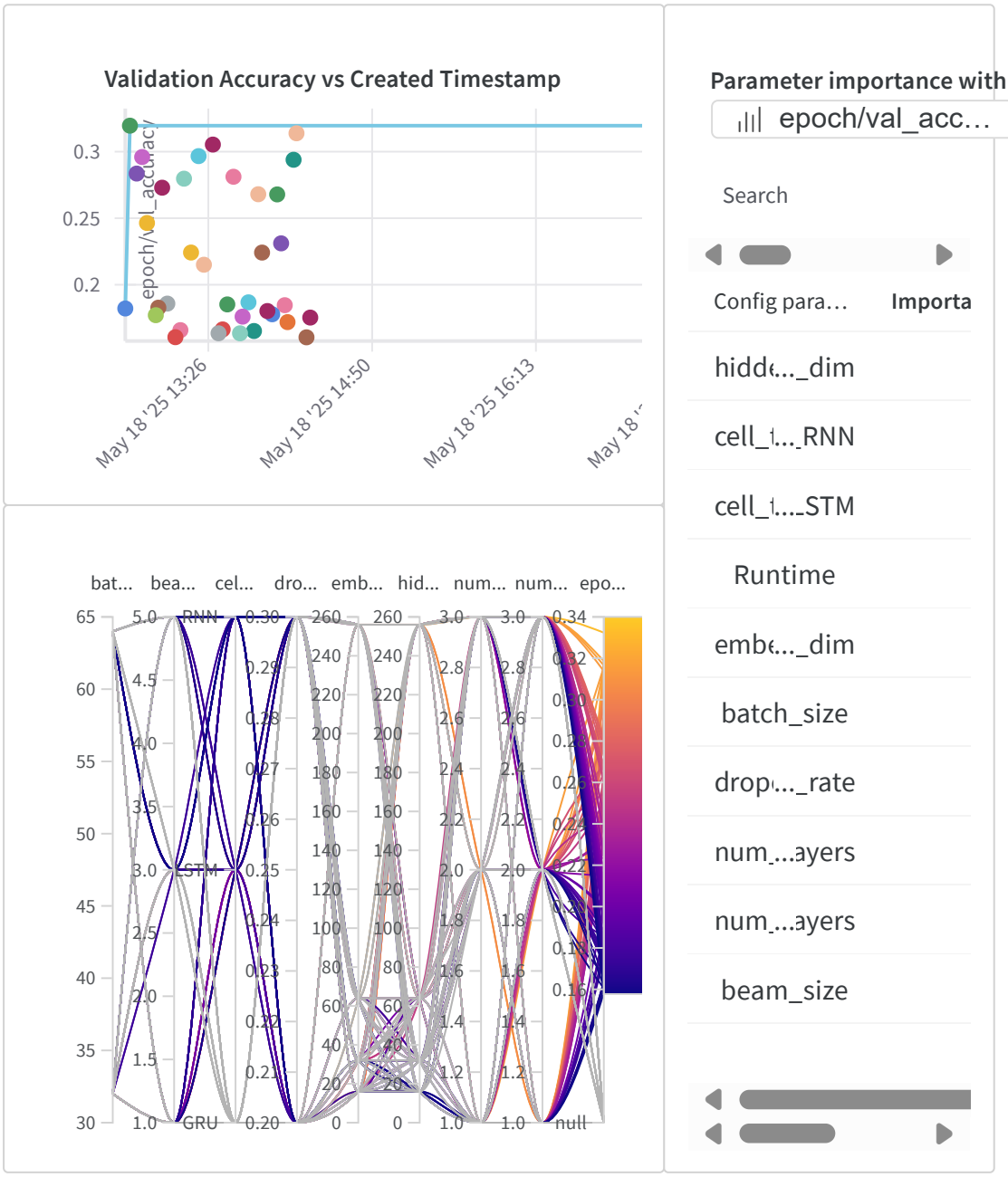
Strategy to Reduce Experiments

To efficiently explore the hyperparameter space, I implemented the following approach:

1. **Bayesian Optimization**: We utilized Bayesian search to minimize the number of experiments required. Unlike grid or random search, Bayesian optimization intelligently selects hyperparameter combinations based on the performance of previous trials.
2. **How Bayesian Search Works**: This method models the hyperparameter space as a probabilistic function, updating its predictions after each experiment. By prioritizing promising regions of the search space and avoiding less effective ranges, Bayesian search converges to optimal hyperparameters faster. This is particularly advantageous for computationally expensive models or large datasets, as it reduces training time while maintaining high performance.

- Also I fixed the number of epochs to 10 as I noticed that after 8-9 epochs the accuracy starts plateauing





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.

Answer

Based on the hyperparameter sweeps conducted for my model, I analyzed the impact of various configurations on model performance. Some key observations are as follows

1. **Beam Search Size Trade-off**: Larger beam sizes consistently improved validation accuracy by exploring more decoding paths, but increased computational cost significantly. A beam size of 3 offered a balanced trade-off, achieving strong performance but it still increased the compute requirements considerably.
2. **Cell Type Performance**: LSTM and GRU outperformed Simple RNN across all configurations. Simple RNN struggled to capture long-term dependencies, resulting in lower validation accuracy even with additional layers. LSTM and GRU achieved higher accuracies.
3. **LSTM Efficiency**: LSTM models achieved highest validation accuracy (~35%) with fewer encoder and decoder layers which suggests LSTM's gating mechanism effectively captures sequence patterns without requiring deep architectures.
4. **GRU Layer Requirements**: GRU models benefited from additional layers, achieving performance closer to LSTM when configured with higher layer counts. GRU's simpler architecture required more layers to match LSTM's

efficiency. Besides, LSTM performed better so I went ahead with that.

5. **Hidden Dimension size**: Larger hidden dimensions (e.g., 256) were strongly correlated with improved accuracy, as they allowed the model to capture more complex patterns. Smaller hidden dimensions (e.g., 16 or 32) led to underfitting, with low validation accuracies. This is expected because higher dimension -> more parameters -> better learning of patterns.
6. **Dropout Rate**: A dropout rate of 0.2 provided the best balance, improving generalization and preventing overfitting. A rate of 0.3 was probably too much info loss, reducing model capacity and lowering accuracy, while no dropout (0.0) led to overfitting.
7. **Batch Size Effects**: A batch size of 64 outperformed 32, enabling more stable gradient updates and higher validation accuracy. Smaller batch sizes increased training variance thus slowing convergence.
8. **Embedding Size**: Higher embedding sizes (e.g., 256) enhanced performance by providing richer representations of input tokens, leading to validation accuracies higher than smaller sizes. Same logic here as hidden dimension size.
9. **Layer Count**: Increasing the number of encoder and decoder layers improved performance only when paired with optimal settings for hidden dimension, embedding size, and cell type. For instance, the best-performing model achieved ~35% validation accuracy because it had all combos of best values, highlighting the synergy of these parameters.
10. **Optimizer**: I used good old Adam optimizer because it performs well and ensured fast and stable convergence compared to alternatives like SGD.

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).

(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.

(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!
- ...

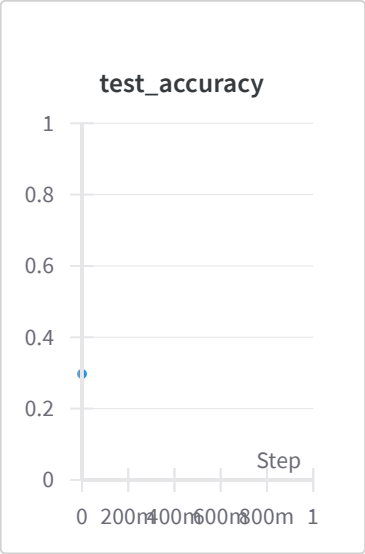
Answer

Part (a)

- The best model from my sweep had the config

```
Input embedding size: 256
Hidden dimension: 256
Cell type: LSTM
Number of encoder layers: 2
Number of decoder layers: 2
Dropout rate: 0.2
Batch size: 64
Epochs: 10 (I used 20 to train further)
```

- This model gave a decent test accuracy of ~30% on the test set



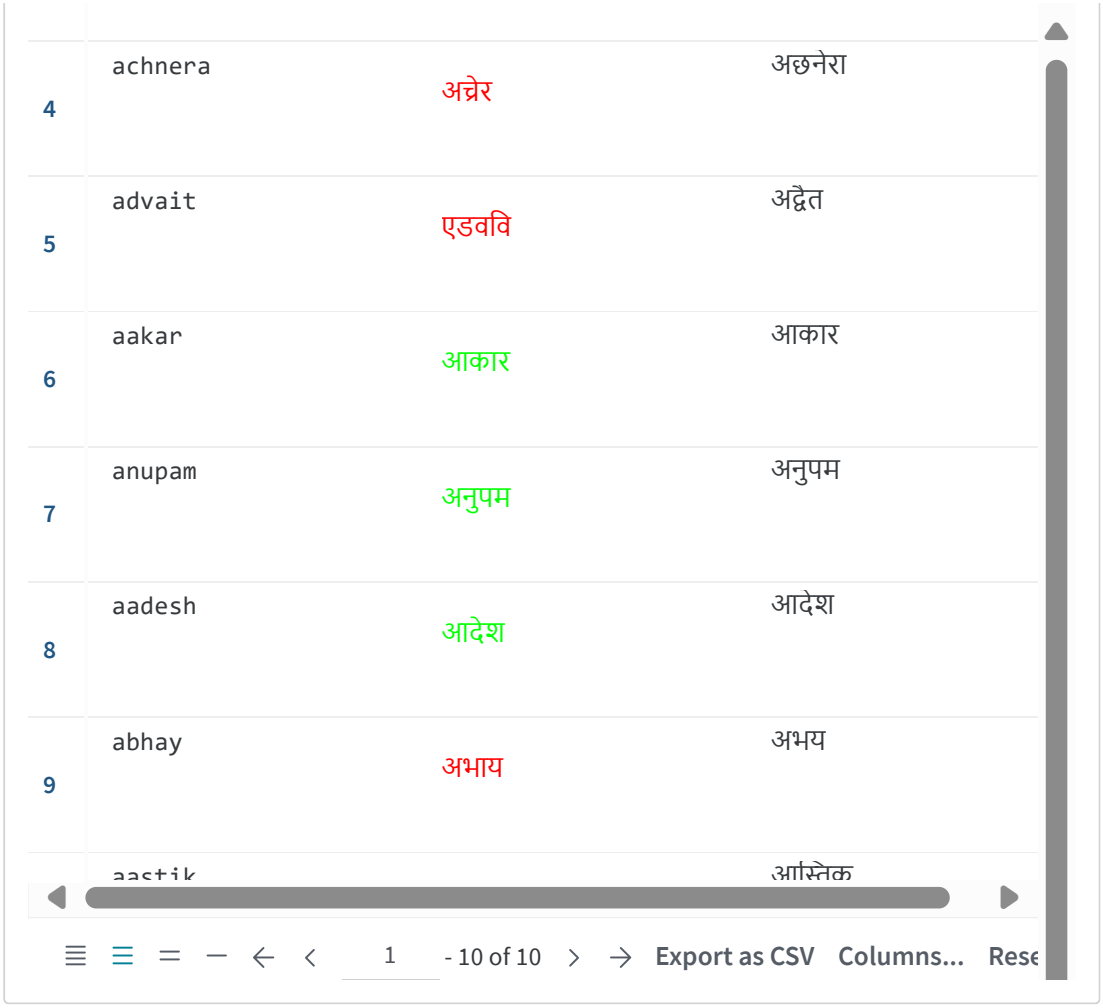
Part (b)

- Some sample predictions for words in the test dataset are shown below
- Correct predictions are written in Green text whereas incorrect predictions are written in Red text

```
runs.summary["Prediction Samples Colored Table"]
```

Filter

	Input Word	Predicted Word	Target Word
1	ankit	अंकीत	आंकेत
2	angreji	अगग्ेजी	अंग्रज़ी
3	andhapan	अंधापन	अंधापन



4	achnera	अच्रेर	अछनेरा
5	advait	एडवि	अद्वैत
6	aakar	आकार	आकार
7	anupam	अनुपम	अनुपम
8	aadesh	आदेश	आदेश
9	abhay	अभाय	अभय
	aastik		आस्तिक

1 - 10 of 10 Export as CSV Columns... Rese

Part (c)

By analyzing the generated predictions I was able to see the following patterns in the mistakes the model was making:

- **Consonant Errors**: More errors occur on consonants (e.g., `angreji`: अगग्ेजी vs. अंग्रज़ी, wrong ग्) than vowels.
- **Vowel Errors Less Common**: Vowel errors are minor (e.g., `ankit`: अंकीत vs. अंकित, ी vs. ि), with correct vowel predictions in cases like `aakar` (आकार).
- **Longer Sequences Error-Prone**: Longer words like `angreji` and `achnera` show multiple errors indicating context loss in extended sequences.

- **Spurious Characters**: Incorrect additions (e.g., virama matra in `आस्तिक्`) suggest issues with sequence termination.
- **Multiple Matras**: Errors in words with multiple complex matras (e.g., `advait`: `एडवि` vs. `अद्वैत`) reflect challenges with character and matra distinctions.
- **Letter Phonetics**: Some words like `angreji` (`अगग्ेजी` vs. `अंग्रज़ी`) are error-prone, because the English `'ji'` is actually written as `'zi' (ज़ी)` in Hindi which causes errors
- **Simple Patterns**: Correct predictions (`andhapan`, `aakar`, `anupam`, `aadesh`) involve common, straightforward patterns, showing model strength.

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.

(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.

(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)

(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).

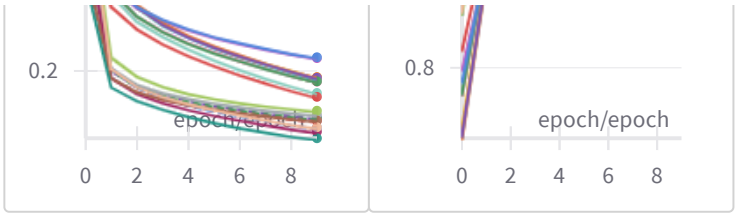
Answer

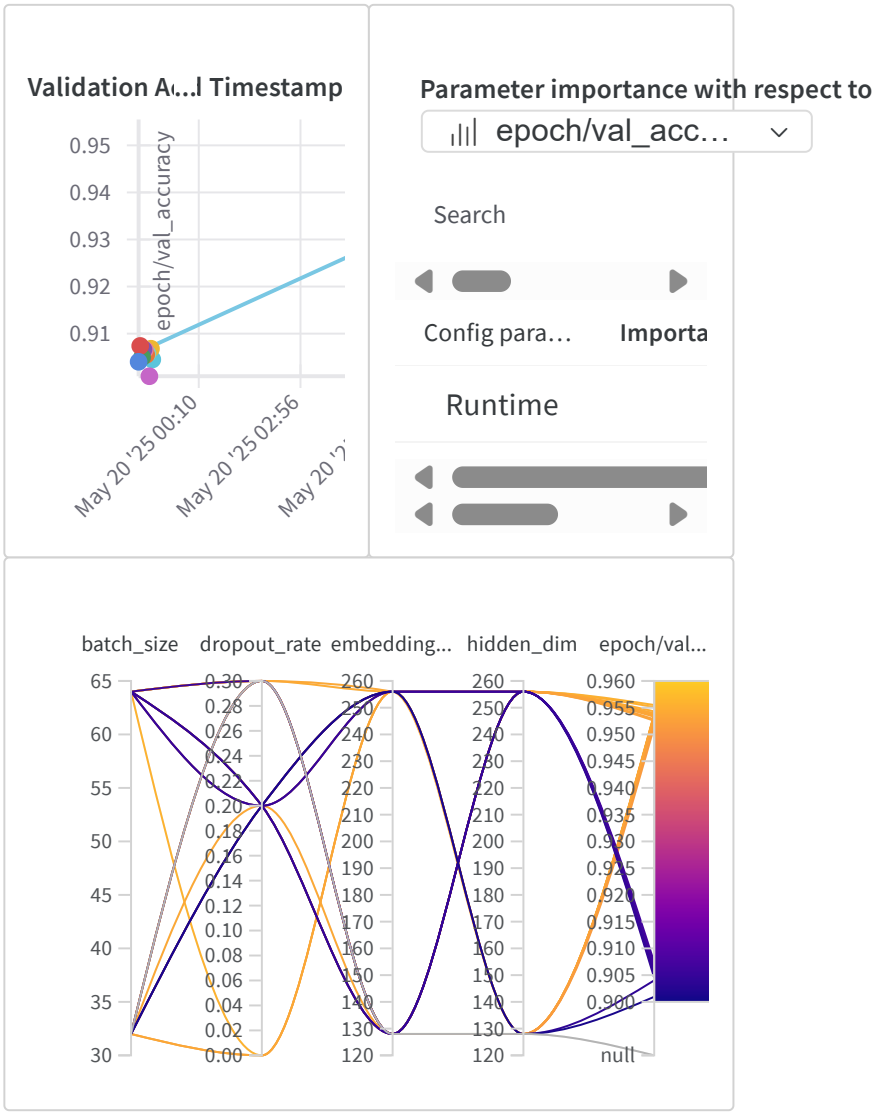
Part (a)

I chose LSTM to use for the final attention model as it was performing best out of the 3 types and ran a hyperparameter sweep over the following values

```
Input embedding size: [128, 256]
Hidden dimension: [128, 256]
Dropout rate: [0.0, 0.2, 0.3]
Batch size: [32, 64]
Epochs: 10
```

- The best model gave a great Validation Accuracy of ~95% which is a huge improvement over the regular LSTM based model

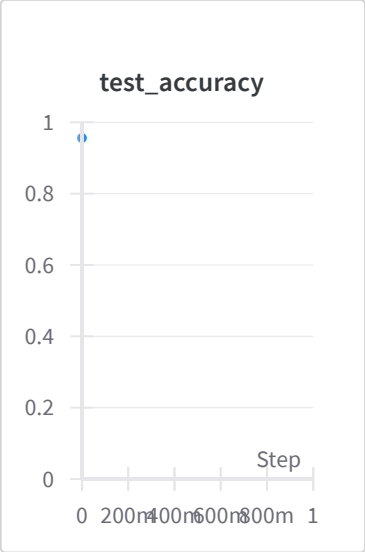




Part (b)

- The final attention based LSTM model gave an accuracy of 95.59% on the test dataset
- This large improvement in performance can be attributed to the fact that English (Subject-Verb-Object) and Hindi

(Subject-Object-Verb) have different syntactical structures and Attention mechanism helps the model correctly focus on the required words during decoding.




```
runs.summary["Prediction Samples Colored Table"]
```

Filter

	Input Word	Predicted Word	Target Word
1	ankit	अंकित	अंकेत
2	angreji	अंग्रजी	अंग्रज़ी
3	andhapan	अंधापन	अंधापन
4	achnera	अछोरा	अछनेरा
5	advait	अद्वैत	अद्वैत
	aradas		अरदास

6		अरदास	
7	avasthaayen	अवस्थायं	अवस्थायं
8	unthak	उंथक	अनथक
9	angaarak	अंगारक	अंगारक
	ashaktataa		अशक्तता

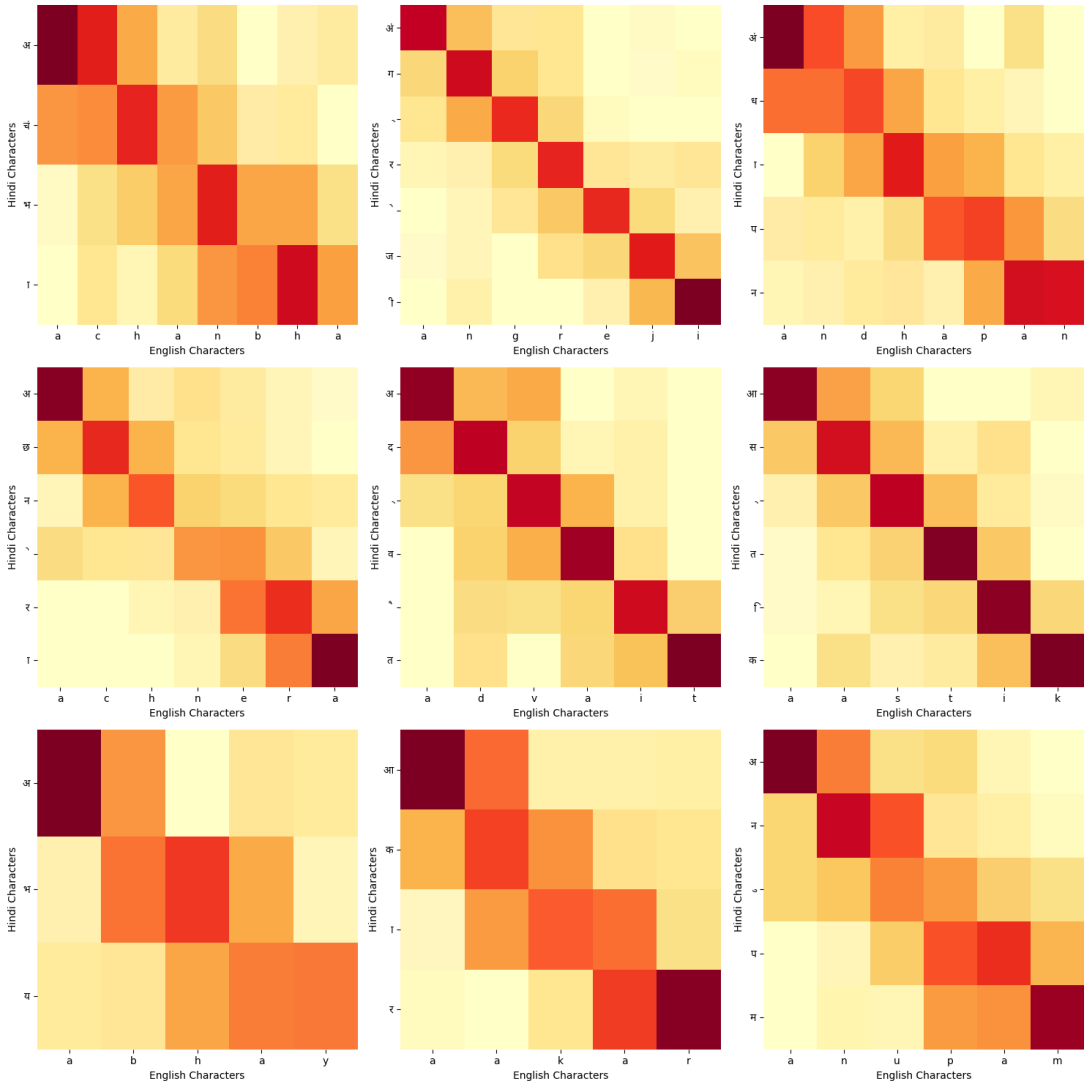
≣ ☰ = – ← ‹ 1 - 10 of 10 › → Export as CSV Columns... Rese

Part (c)

- Attention model outperforms vanilla: Achieves ~96% accuracy vs. vanilla's 30%
- Corrected ankit: Vanilla (`अंकीत`, wrong matra) vs. Attention (`अंकित`, correct), showing better vowel sign handling.
- Corrected aastik: Vanilla (`आस्तिक्`, spurious matra) vs. Attention (`आस्तिक`, correct), indicating reduced over-generation.
- Common correct predictions: Both correctly predict andhapan, aakar, anupam, aadesh, indicating robust learning of simpler patterns.
- Inference: Attention improves matras and character alignment. It could be improved even further with more diverse training data.
- The Attention mechanism allows the model to see the relationship between the current character and those around it which is a very key aspect of the Hindi language to assign the correct matras to the letters. In this way the Attention based model works much better

Part (d)

- The attention heatmaps generated for a few samples from the test dataset are plotted below in a 3x3 grid
- Darker cell color indicates greater attention weight



Attention heatmaps from sample test set predictions

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!

Answer

- I have used the attention values at each decoding timestep to visualize the attention at each decoder output for a sample prediction from the test dataset
- The input characters with highest attention while generating a particular decoder output are highlighted strongly in green
- With context to the visualization in the article mentioned, this table is basically equivalent to a snapshot of it, i.e, if you were to hover your mouse over the character in the rows of the table (decoder output character) the corresponding characters in the columns of that row (input characters) would be highlighted according to how much attention is on them.

Output ↓	a	n	k	i	t
अ	a	n	k	i	t
ं	a	n	k	i	t
क	a	n	k	i	t
ि	a	n	k	i	t
त	a	n	k	i	t

Attention Connectivity map for one sample from test dataset prediction

Question 7 (10 Marks)

Paste a link to your github code for Part A

Example: https://github.com/<user-id>/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.

Answer

Github link - https://github.com/suddu21/Deep-Learning-Assignment-3

Question 8 (0 Marks)

Self Declaration

I, Sudhanva Satish (DA24M023), 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.

https://wandb.ai/da24m023-indian-institute-of-technology-madras/DL_A3/reports/DA6401-Assignment-3-Transliteration-System--VmlldzoxMjgyNTE1Ng