Report: Assignment 3

Report on how to use recurrent neural networks to build a transliteration system.

Rigved Sah cs20m053

In this assignment we modelled sequence to sequence learning problems using Recurrent Neural Networks, compared different cells such as vanilla RNN, LSTM and GRU, understood how attention networks overcome the limitations of vanilla seq2seq models and visualized the interactions between different components in a RNN based model.

The dataset used to train and evaluate the model is **Dakshina** dataset by Google.

Specifically, given as input a Romanized string **(ghar)**, the network is trained to produce the corresponding word in Devanagari **(घर)**.

Below panel contains sample word pairs present in Dakshina dataset.

Devnagri Word	Romanized Word
अं	an
अंकगणित	ankganit
अंकल	uncle
अंकुर	ankur
अंकुरण	ankuran
अंकुरित	ankurit
अंकुश	aankush
अंकुश	ankush
अंग	ang
अंग	anga
अंगद	agandh
अंगद	angad
अंगने	angane
अंगभंग	angbhang
अंगरक्षक	angarakshak
अंगरक्षक	angrakshak
अंगारा	angara
अंगारे	angaare
अंगारे	angare

We implemented a RNN which takes a Romanized input string from Dakshina dataset as input and produce the corresponding word in Devanagari. The code was written from scratch using libraries such as Keras, Tensorflow and Numpy.

Our RNN based seq2seq model consists of the input layer for character embeddings followed by an 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).

Our model consists of 5 Convolution Layers. Each convolution layer is followed by a ReLU activation and a Max Pooling layer. After 5 such conv-relu-maxpool blocks of layers we have one dense layer followed by the output layer containing 10 neurons (1 for each of the 10 classes). We used Softmax as output function and Cross-Entropy as the loss function.

In addition to above layers we have also used Batch Normalization & Dropout layers in various different positions in some sweep runs to check the effect of these layers on convergence, accuracy, loss etc.

The code is written in a very flexible manner that allows changing any hyperparameters such as filters, size of filters and activation functions etc. at just one place.

Train-Val-Test Split

We used the standard train/ val/ test split as provided by the Dakshina dataset and used the validation dataset for hyperparameter tuning.

The summary for different datasets is as:

Training Data:	44,204 word pairs
Validation Data:	4,358 word pairs
Test Data:	4,502 word pairs

The total numbers of input (Romanized) and target (Devanagari) characters across all datasets (training, validation and testing) are 28 and 66 respectively. The maximum length for input and target sequences across all datasets are 20 and 21 respectively.

Total computations done by the network

Given,

Input Vocabulary Size	=	Output Vocabulary Size	=	V
Input Sequence Length	=	Output Sequence Length	=	T
Size of Embedding	=	m		
Hidden Cell State of Encoder	=	Hidden Cell State of Decoder	=	k

- Dimension of Word Representation = T * V
  After Embedding layer, dimension = T * m
- State Computations of Encoder:

  $s_i = \sigma(W s_{i-1} + U x_i + b)$, where s_i and x_i are the state and input to encoder at i^{th}timestep.

  1.) Computation for calculating state $s_1 = \sigma(Ux + b)$:

  Dimensions of $U = k * m,\ x_1 = m * 1,\ b = k * 1$

  Number of multiplications in $Ux = k$= km

  Number of additions in $Ux + b = k(m - 1) + k$

  Number of additions in sigmoid of vector, $Ux + b = k$

  Number of divisions in sigmoid of vector, $Ux + b = k$

  Total number of computations = $2k(m + 1)$

  2.) Computation for calculating states from $s_2 = \sigma(W s_1 + U x_1 + b)$ to s_t:

  Dimensions of $U = k * m,\ x_i = m * 1,\ b = k * 1, s_{i-1} = k * 1, W = k * k$

  Number of multiplications in $Ux + Ws + b = km + k^2$

  Number of additions in $Ux + Ws + b = k(m - 1) + k * (k - 1) + 2k$

  Number of additions in sigmoid of vector, $Ux + Ws + b = k$

Number of divisions in sigmoid of vector, $Ux + Ws + b = k$

Total number of computations = $2k(m + k + 1)$

Total computations for encoder states for all timesteps = $2k(m + 1 + (T - 1)(m + k + 1))$

- Output computations of Encoder:

  $y_i = softmax(Ps_i + c)$, where s_i is the state of encoder at i^{th}timestep.

  1.) Dimensions of $P = V * k,\ s_i = k * 1,\ c = k * 1$

  Number of multiplications in $Ps + c = Vk$

  Number of additions in $Ps + c = V(k - 1) + V$

  Number of additions in softmax of vector, $Ps + c = V$

  Number of divisions in softmax of vector, $Ps + c = V$

  Total number of computations = $2V(k + 1)$

- Total computations of Encoder for all timesteps = $2k(m + 1 + (T - 1)(m + k + 1)) + 2TV(k + 1)$
- State computations of Decoder= T * Computation of Encoder State $s_2 = 2Tk(m + k + 1)$
- Output computations of Decoder = Output computations of Encoder = $2V(k + 1)$
- Total computations of Decoder for all timesteps = $2Tk(m + k + 1) + 2V(k + 1)$
- Total computations = $2k(m + 1 + (T - 1)(m + k + 1)) + 2Tk(m + k + 1) + 2V(k + 1)$

Total parameters of the network

- **Parameter in Encoder:**

1.) Between T inputs and T cells, there are T number of U matrices of size $k * m \rightarrow T * k * m$ parameters.

2.) Between T inputs and T cells, there are T number of b vectors of size $k * 1 \rightarrow T * k$ parameters.

3.) Among T cells, there are T -1 number of W matrices of size $k * k \rightarrow (T - 1) * k^2$ parameters.

4.) Among T cells and output layer there are T number of P matrices of size $V * k \rightarrow T * V * k$ parameters.

5.) Between T cells and output layers, there are T number of b vectors of size $V * 1 \rightarrow T * V$ parameters.

- **Parameter in Decoder:**

1.) Same number of parameters as in Encoder.

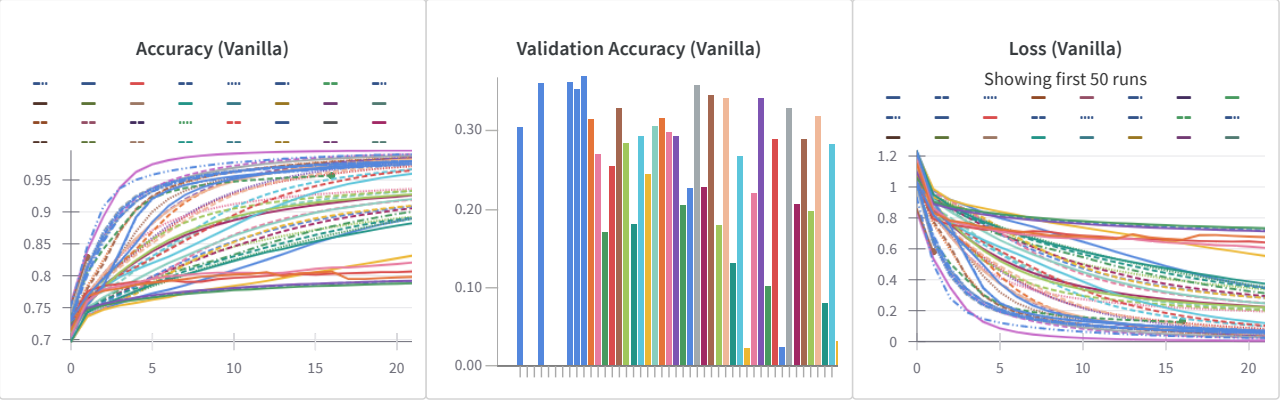
- **Total Parameter** = $2 * (Tkm + Tk + (T - 1)k^2 + TVk + TV)$

Hyperparameter Tuning

The hyperparameters considered for sweep as as:

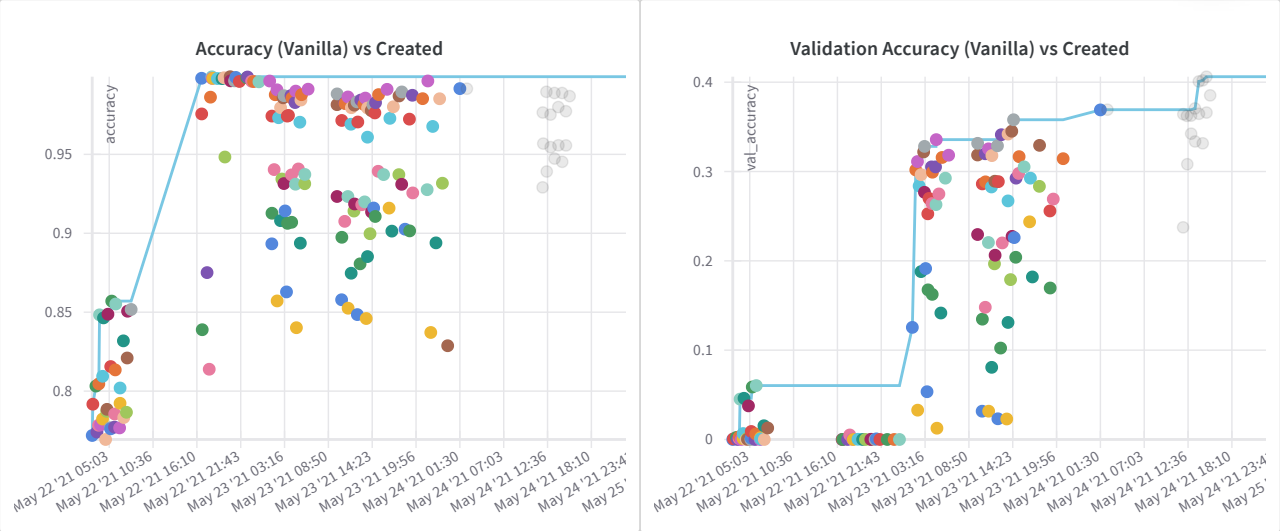
Input Embedding Size	**16, 32, 64, 256**
Number of Encoder Layers	**1, 2, 3**
Number of Decoder Layers	**1, 2, 3**
Hidden Layer Size	**16, 32, 64, 256**
Cell Type	**RNN, LSTM, GRU**
Dropout	**0.2, 0.3**
Beam Width	**0, 4**

Plots from Hyperparameter Tuning

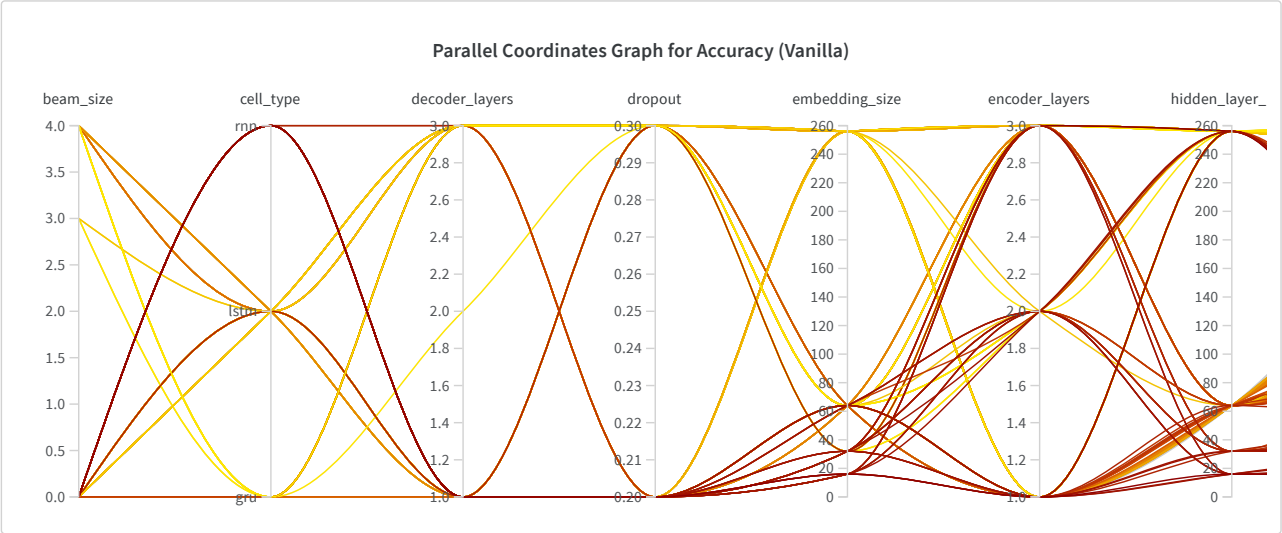


Add panel

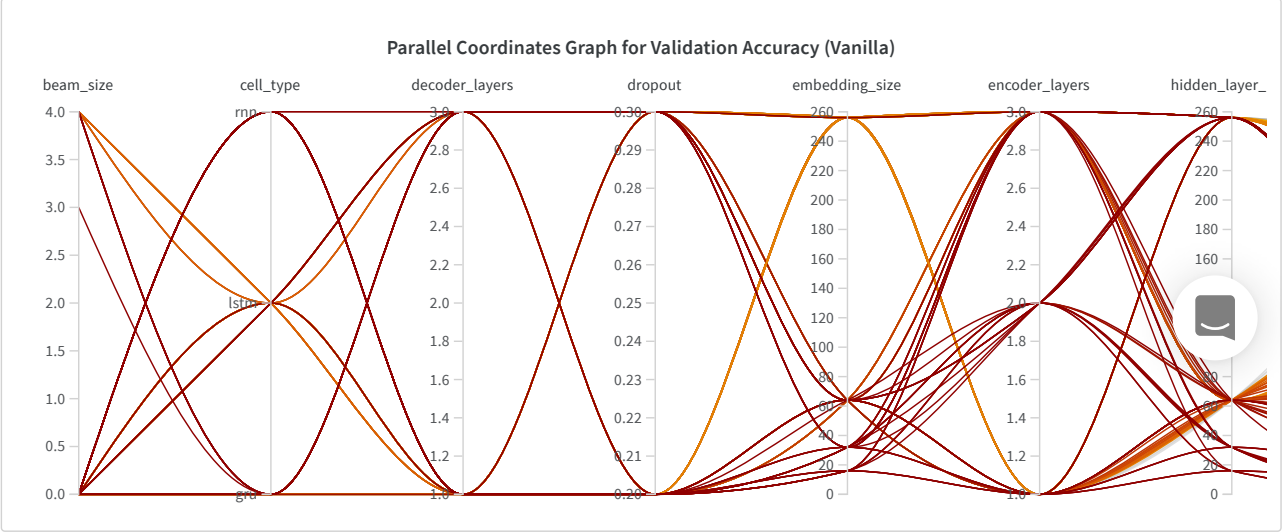
Here accuracy and loss are calculated automatically by Keras while training the model and are therefore wrt to each character being predicted correctly or not, whereas the validation accuracy is manually calculated by us after training by going over complete validation data and matching the complete predicted sequence (target word) with the true word, as we are supposed to do for test data.



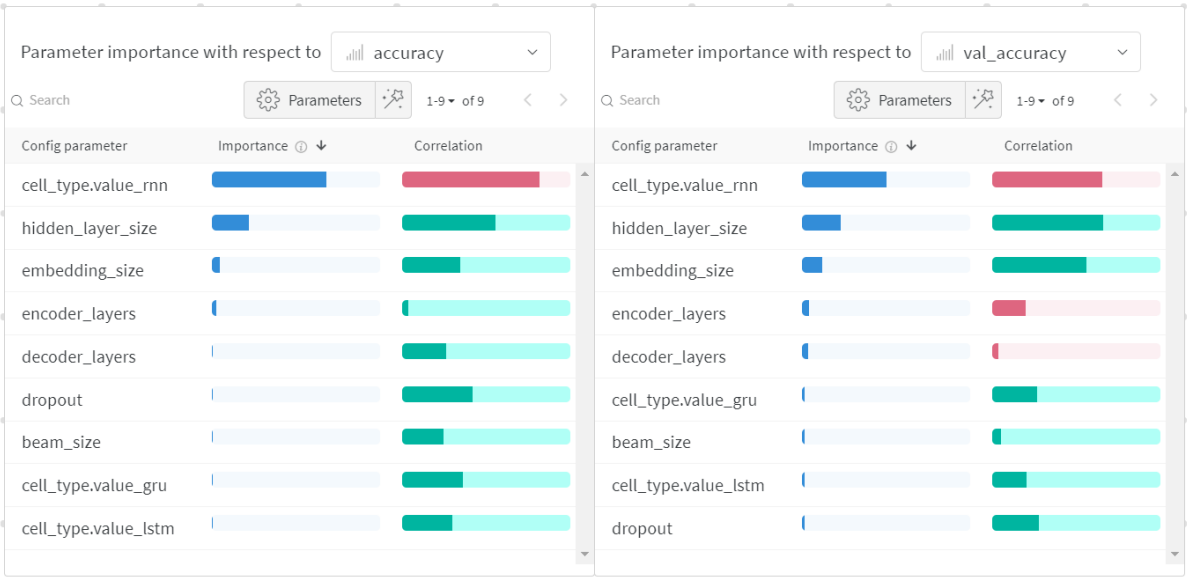
Add panel



Add panel



Add panel



Strategies to Reduce Runs

As we can see above, we will have exponential number of hyperparameters combinations therefore, to perform hyperparameter tuning efficiently we used,

1. Wandb's **Bayes Optimization** search strategy which uses gaussian function to model the function and then chooses parameter to optimize probability of improvement. Here we are optimizing (maximizing) the **validation accuracy.**
2. Wandb's **Hyperband** stopping criteria. It speeds up hyperparameter search by stopping poorly performing runs.

Observations & Inferences

- For same number of epochs RNN based model gives worst accuracy among GRU and LSTM. The highest recorded accuracy and validation accuracy for RNN based models are 0.857 and 0.06035 respectively.
- Hidden layer size play a crucial role in terms of both accuracy and validation accuracy as can be seen from above correlation summary table. Also we can see that all the top performing models have hidden layer size equal to 256.
- Dropouts leads to better performance as it reduces the overfitting on data. From above graphs we can see that nearly for every cell whether RNN, LSTM or GRU as we increased dropout from 0.2 to 0.3, th was a slight increase in validation accuracy.
- GRU gave the best performance in terms of accuracy as well as validation accuracy amongst all cells. It might be due to parameter tying in GRU which further prevents overfitting. Validation accuracy of Vanilla GRU (without attention) is around 0.3609, for LSTM it is around 0.3357 and for RNN it is around 0.06035.
- As we use beam search with width 4, we observed a slight increase in validation accuracy, however the time taken to train the model increases significantly.
- Small Embedding size of around 16, 32 gives very poor performance as can be seen from parallel coordinates graphs (brown curves).
- Number of Encoder and Decoder layers increases the performance considerable. On observing parallel coordinates graph as well as accuracy vs created graph we can see that in Vanilla case (without attention) the highest accuracy and validation accuracy models have around 3 encoder and decoder layers. It might be due to more neurons we are able to learn complex patterns in the data.
- From accuracy and loss graph above we can observe that the model saturates around 15 epochs (as the graph becomes almost horizontal), however here we have ran 25 epochs.

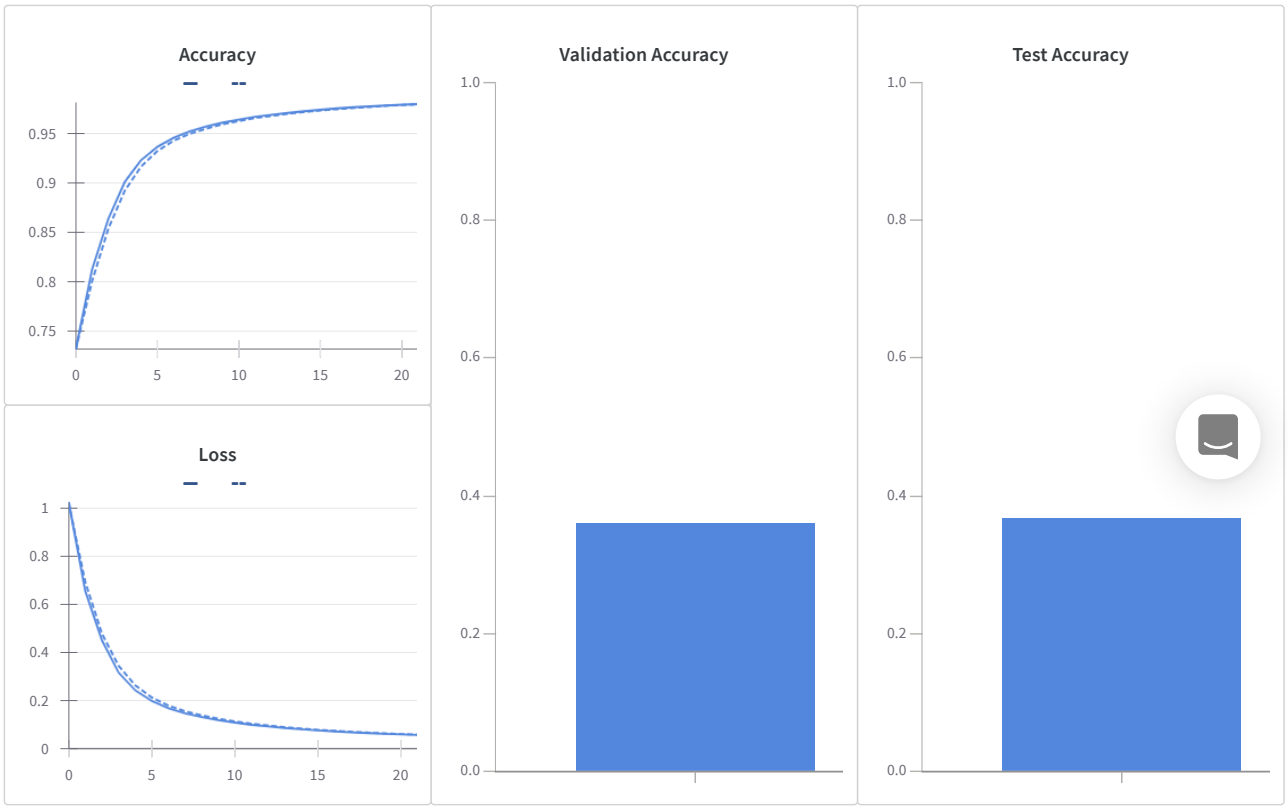
- On observing the outputs (predicted sequences) we found that longer sequences are often predicted incorrectly. It might be because the embedding size is fixed and for long sequence it might not hold good representations of initial characters in the input sequence. This can be overcome by using attention networks.
- We also ran our Vanilla model on other languages such as Tamil where we got much higher accuracy as compared to Hindi. It might be due because there are more characters in Hindi (around 67) than in Tamil (around 48). Varied characters such as **क** and **क़** (very close to each other but different) could often contribute in incorrect prediction since probability is now distributed over more classes.

Vanilla Model Performance on Test Data

We trained our best model with the following configuration,

Embedding Layer Size:	**256**
Number of Encoder Layers	**3**
Number of Decoder Layers	**3**
Hidden Layer Size (Latent Dim)	**256**
Cell Type	**GRU**
Dropout:	**0.3**
Beam Width:	**0**
Epochs:	**25**
Batch Size:	**128**

Statistics of our Model



Add panel

Results of our model are as,

Training Accuracy:	**0.9815**
Validation Accuracy:	**0.3596**
Test Accuracy:	**0.3692**
Training Loss:	**0.05498**

We have saved all the predictions made by the model on the test dataset in the `predictions_vanilla.csv` file and uploaded the same on our Github page. This file contains three columns namely `Input Sequence`, `Predicted Sequence`, `Original Sequence` which denotes the input sequence we are given, the predicted (decoded) sequence as given by our model and the true sequence to which our prediction tends to.

Predictions by the Model

Input	Predicted	True
antrmukh	अंतर्मुख	अंतर्मुख
engineer	इंजीनियर	इंजिनियर
aahnan	आह्नान	आह्नान
email	इमेल	ईमेल
udghosh	उद्घोष	उदघोष
erica	एरिका	एरिका
association	एसोसिएसन	एसोसिएशन
kahaniyon	कहानियों	कहानियों
clock	क्लॉक	क्लाक
gaumutr	गौमुत्र	गौमूत्र

Fig 1. Sample Predictions Grid

Comments on the Errors Made

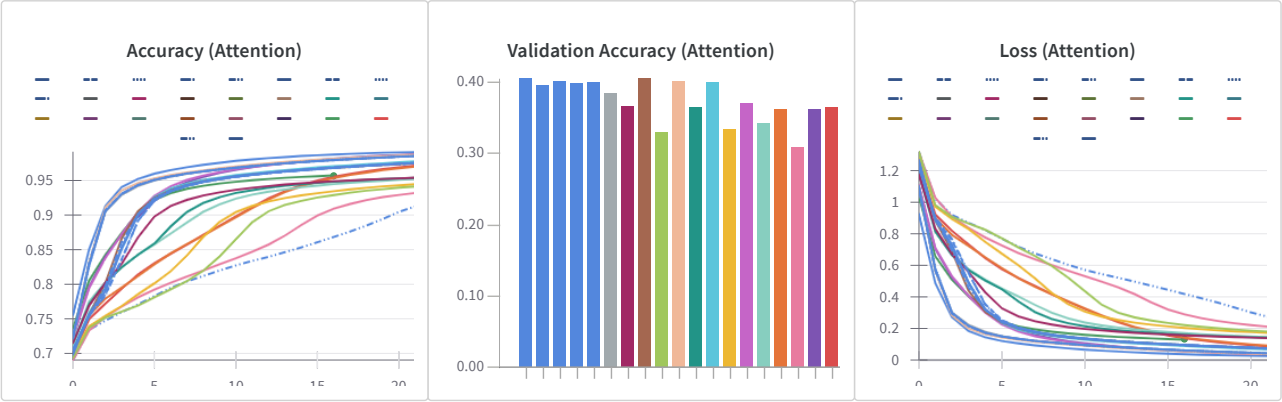
- Model is not able to recognize **ra** well as **र** or **द्र**. It has made many such mistakes in various instance such as dra, kra etc.
- The model is making more mistakes on longer sequences such as **augustustine, adhyayanarat** etc.,
- Model is outputting **क़** instead of **क** in both **q** as well as **kav,** when in case of kav it should give **क**.
- **i** is most of the times predicted as **ी**, which is a source of major mistakes.
- Similarly for **a,** which is wrongly assumes as **आ** instead of **अ**.

Attention Network

We added an attention network to our basic seq2seq model and trained the model again. For the sake of simplicity we have used a single layered encoder and decoder.

For this assignment we used **Bahdanau's attention** (Additive attention).

Hyperparameter Tuning

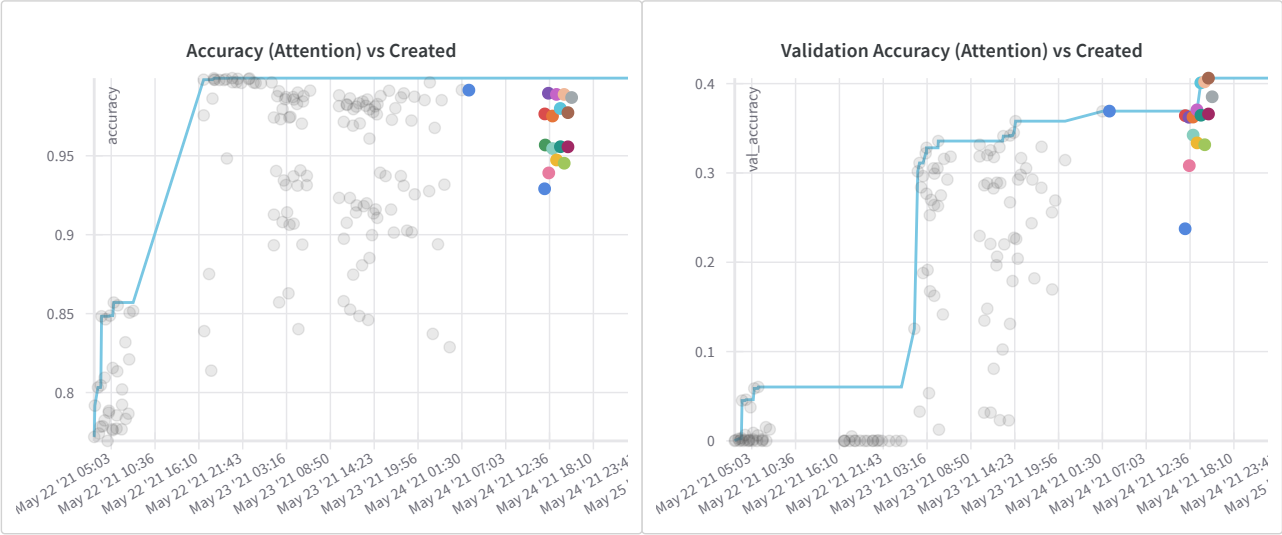


Add panel

Draft autosaved just now

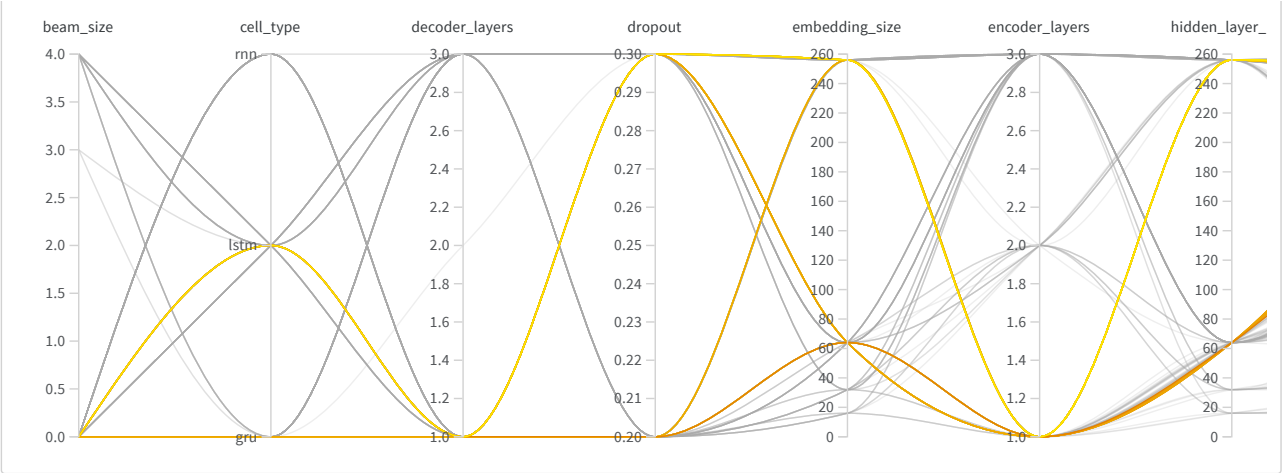
Save to report

Here accuracy and loss are calculated automatically by Keras while training the model and are therefore wrt to each character being predicted correctly or not, whereas the validation accuracy is manually calculated by us after training by going over complete validation data and matching the complete predicted sequence (target word) with the true word, as we are supposed to do for test data.

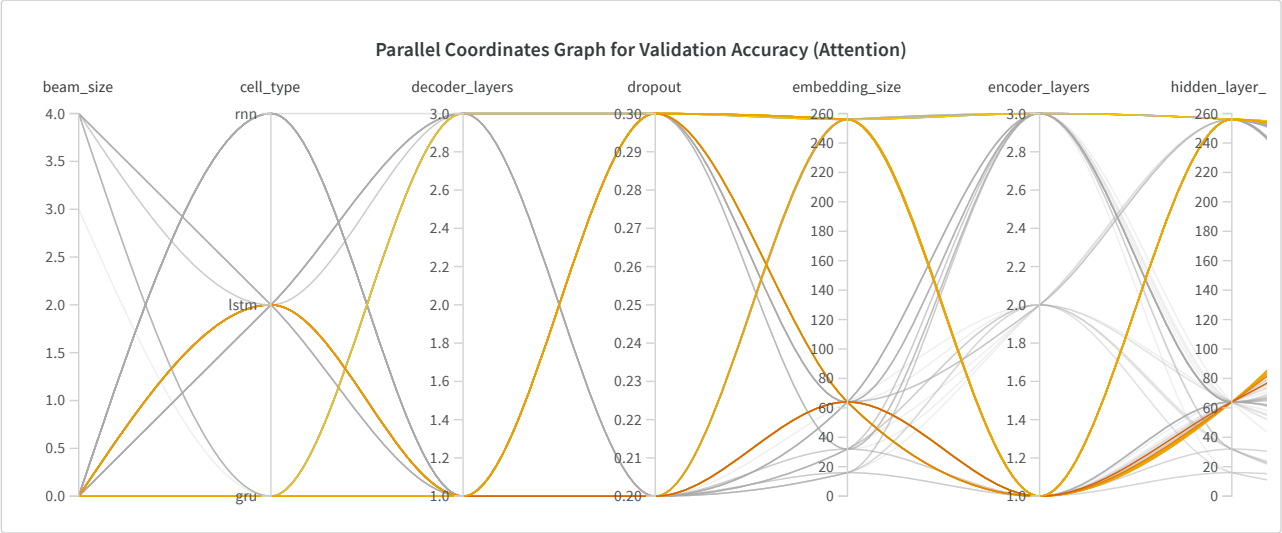


Add panel

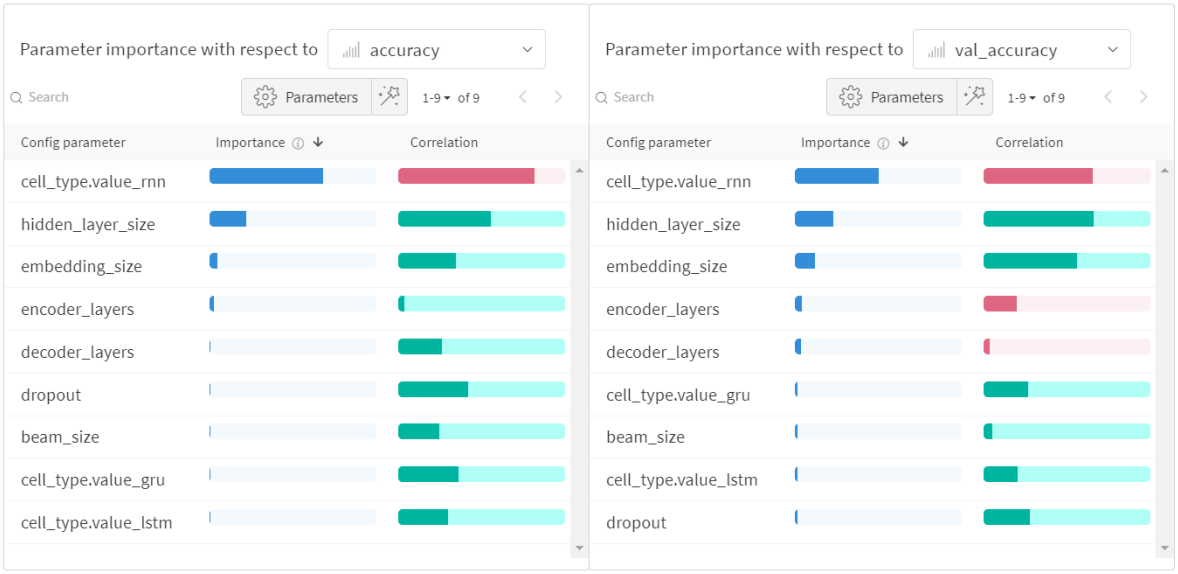
Type '/' for commands



Add panel



Add panel

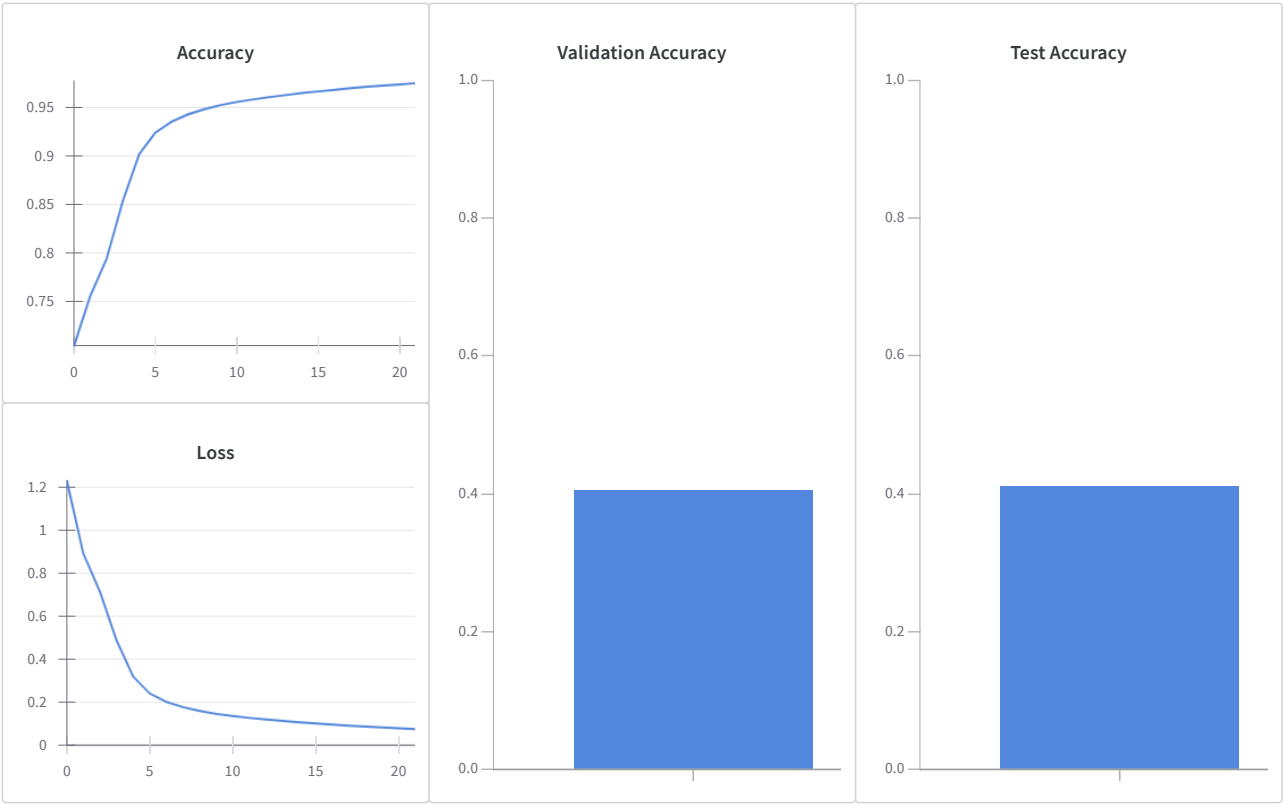


Performance on Test Data

We trained our best model with the following configuration,

Embedding Layer Size:	**64**
Number of Encoder Layers	**1**
Number of Decoder Layers	**1**
Hidden Layer Size (Latent Dim)	**256**
Cell Type	**GRU**
Dropout:	**0.3**
Beam Width:	**0**
Epochs:	**25**
Batch Size:	**128**

Statistics of our Model



Add panel

Results of our model are as,

Training Accuracy:	**0.9778**
Validation Accuracy:	**0.4061**
Test Accuracy:	**0.4125**
Training Loss:	**0.06809**

We have saved all the predictions made by the model on the test dataset in the `predictions_attention.csv` file and uploaded the same on our Github page. This file contains three columns namely `Input Sequence, Predicted Sequence, Original Sequence` which denotes the input sequence we are given, the predicted (decoded) sequence as given by our model and the true sequence to which our prediction tends to.

Comparison with Vanilla Model

Yes the attention based models performed better than Vanilla models. Infact keeping the other configurations such as batch size, epochs, embedding layer size, hidden layer size, cell type, dropout same and using single layer encoder and decoder (since we are only using single layers in attention models) we got around **0.05 - 0.08** increase in validation accuracy.

The average accuracy using above configurations was averages around **0.32** whereas the average validation accuracy for attention based models is around **0.4**. Now even for high end models having multiple layers such as 3 layers in encoder and decoder the maximum validation accuracy reported was around **0.3692** which is still less than average of attention models. The maximum validation accuracy reported for attention based model is **0.415,** (without using multiple layers).

We also ran our model on other languages such as Tamil where using attention we got much higher accuracy as compared to Hindi (around 0.58).

One drawback of attention is model takes more time to train.

While checking the predictions of our best attention and best vanilla based models we found that the attention based models corrected some of the errors made by vanilla seq2seq models which are as,

- As we mentioned in above section (Vanilla Model Performance on Test Data) that vanilla model is not able to recognize **ra** well as **र** or **द्र.** Attention based model is more able in distinguishing this. For eg. vanilla model decoded **अंद्राबी** as **अंदरबी** whereas attention model decoded as **अंद्रबी**. It still is not equal but more close to true sequence.
- Attention based models are better at decoding long sequences. It may be because they also take into account the state vectors at each timestep, which helps to overcome the problem of finite size of final state vector. For e.g. vanilla model decoded **अनावश्यक** as **अनावैश्यक** whereas attention model correctly decoded it as **अनावश्यक**.
- Again as mentioned in above section (Vanilla Model Performance on Test Data) that vanilla model is outputting **क़** instead of **क** in both **q** as well as **kav,** attention model rectified it and mostly gives the corrected decoding **क**.

Attention Heatmaps

Attention heatmaps for 10 random inputs from test dataset are represented in a 3 x 3 grid as below. y-Axis of heatmaps represents the target (decoded) characters and the x-Axis represents the input characters.

In below heatmap color at (x, y) coordinates represents the influence of x^{th} input character while decoding the y^{th} target character. Here light yellow denotes very high influence while dark purple denotes very low influence. These color values are calculated according to the values of α_i of the context vectors.

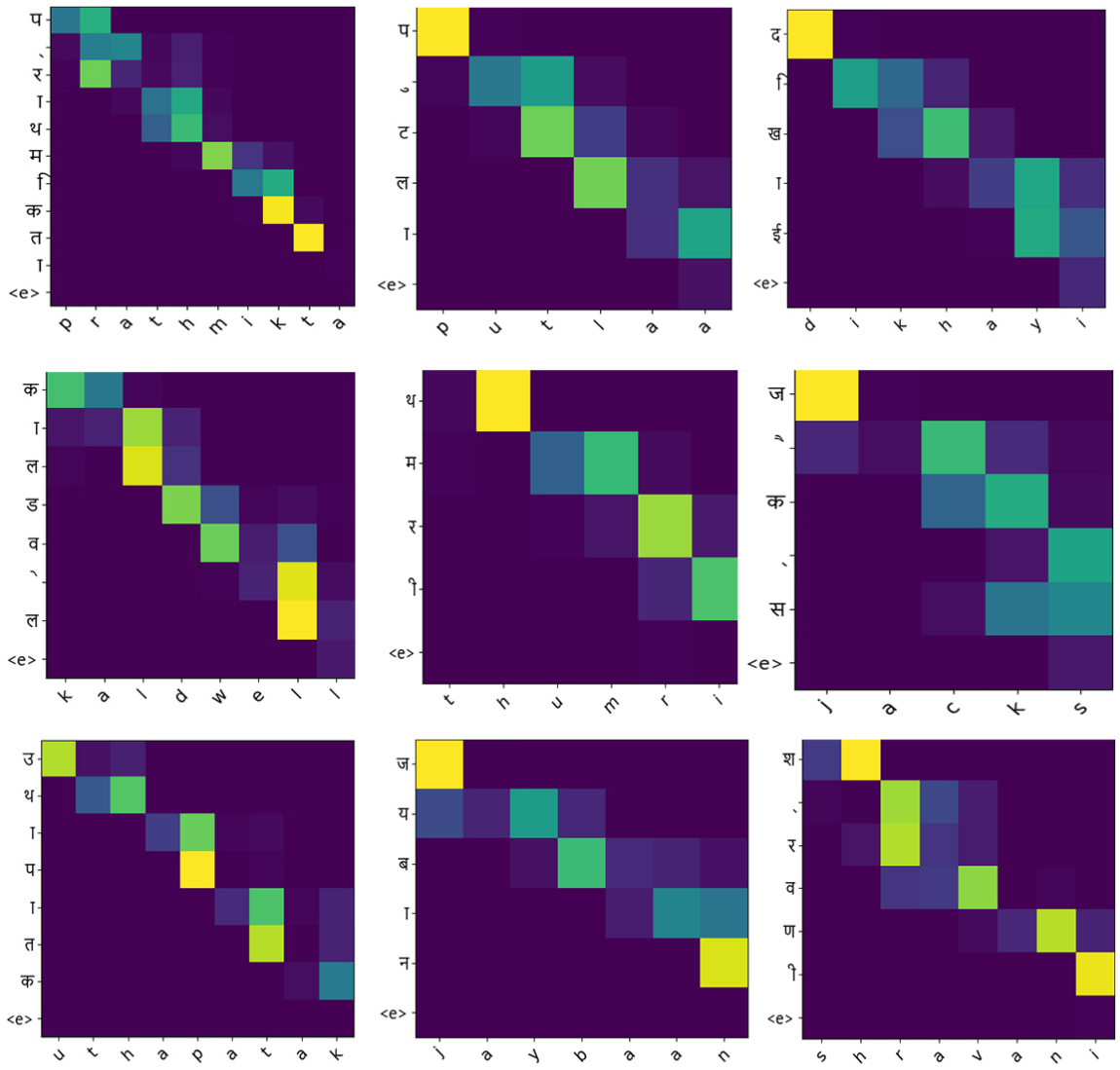


Fig. 1 Attention Heatmaps

Visualization

Deep Learning is often considered something as a "Black Box" where we know that it does the work but often do not know exactly how.

What are the neurons learning? What features are they capturing? Which output is dependent on which input?????

These are the question that prompted researchers to want to understand how deep learning do what it do. To answer these questions various visualization techniques were discovered that provided us with a sensible way to interpreting what is going on behind the scenes and in essence unboxing this black box.

Since we are dealing with RNNs here we present two visualization techniques that gives us an insight on the working of RNNs and how predictions are made. The two visualizations are as,

Connectivity Visualization

The goal of this visualization is to figure out that when the model is decoding the i^{th} character in the output which is the input character that it is looking at, or in other words which input characters(s) influences the the decoded character at i^{th} timestep. Not only this using the color intensities, we will also visualize the importance (influence) of input characters in predicting the output character.

In attention networks, instead of taking the final encoder state like we do in normal encoder decoder RNNs we take the encoder states at each time step and using different methods, such as Luong's (Dot Product) or Bahdanau's (Additive) we calculate the scores which in turn after taking softmax gives us the α_is denoting the importance of each state (and in essence the inputs). Higher the α_i for a timestep, greater is the influence of the input at that time step in calculating the target character.

Using above concept we visualized some of the input and predicted sequence pairs such as,

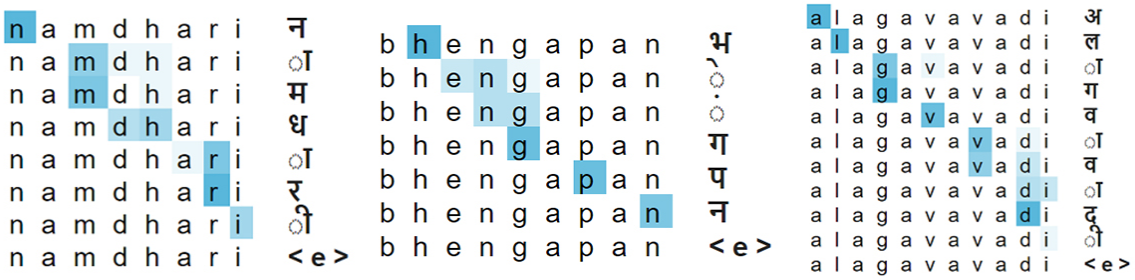


Fig. 1 Namdhari, Fig. 2 Bhengapan & Fig. 3Alagavavadi

Here darker the blue blackground of a character more is the influence of that character on the rightmost decoded character. For e.g. in Fig. 1, **ध** is looks at **d & h** while decoding which makes sense as we typically write ध as dh in Hindi (also the sound they make is similar). Similarly in Fig. 2, **ग** looks at **g,** again we typically write ग as g in Hindi.

LSTM Cell Visualization

The goal of this visualization is to figure out that what does a neuron in a given layer captures about the character it is decoding, or in other words given a neuron at some layer will it fire when a particular character is decoded or not. Not only this using the color intensities, we will also visualize the probability with which it fires.

Here while decoding a character at a particular time step we also get the state at that timestep infact, the state goes on to determine the probability distribution across the vocabulary. Now we intercept the state at last layer (we can do it for any layer, for this assignment we considered last layer, it can be simple RNN, LSTM or GRU). After getting the state which is actually a vector of size equal to the number of neurons in that layer, we apply sigmoid so as to get the probability of each neuron firing. Higher the value value of an element of the vector, more active is the neuron at that particular position.

Using above concept we can uncover interesting information about that neurons. We visualized some of the predicted words as,

Fig.1 Activations of Neuron 230

Fig.2 Activations of Neuron 78

Red denotes the active neuron and blue denotes silent neuron. The intensities of colors denotes the intensities of activeness of neurons. All the visualizations are done for a single GRU layered RNN having hidden layer size (latent dim or number of neurons) equal to 256.

Fig. 1 tells us about the activations of neuron 230. Here we can see that the first character of every word is red i.e. this neuron fires for every first character of a word. This gives us a feeling that this neuron might be trained to recognize the first characters of the words. Contrastively Fig. 2 tells a different story. It tells us about the activations of neuron 78, here the first character of every word is blue i.e. this neuron does not fires for first character of any word. This gives us a feeling that this neuron might be trained to ignore the first characters of the words.

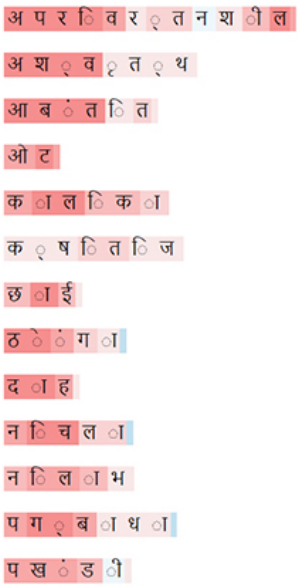


Fig.3 Activations of Neuron 25 Fig.4 Activations of Neuron 128

Here Fig. 3 tells that this neuron (25) fires for every character and thus is not so useful for prediction. Fig. 4 tells us about the activations of neuron 128 and as we can see we could not understand the pattern, there might be some complex pattern that the neuron learnt but we (Rigved & Sumit) as a human are unable to recognize it.

Github Link

The code for this part can be found at
https://github.com/rigvedsah000/CS6910/tree/master/Assignment%203

We will include a README file at root of Assignment 2 folder on how to run the code.

Self-Declaration

Contributions of the two team members are as,

CS20M053 (50% Contribution)

- Implemented multilayer RNN
- Implemented attention
- Implemented visualization
- Wrote report
- Ran wandb sweeps
- Drew observation and inferences

CS20M067 (50% Contribution)

- Implemented Multilayer RNN
- Implemented attention
- Implemented visualization
- Made photos
- Ran wandb sweeps
- Drew observation and inferences

We, **RIGVED SAH, CS20M053** and **SUMIT NEGI, CS20M067**, swear on our honor that the above declaration is correct.