CS7015 Deep Learning : Programming Assignment 3 Report

Anupama S (EE15B009) and Sahana Ramnath (EE15B109)

April 17, 2019

Checklist :

- ✓ We have read all the instruction carefully and followed them to our best ability.
- ✓] We have written the name, roll no in report.
- ✓ Run sanity_check.sh.
- ✓ We will be submitting only single submission on behalf of our team.
- ✓ We have not included unnecessary text, pages, logos in the assignment.
- ✓ We have not used any high level APIs(Keras, Estimators for e.g.).
- ✓ We have not copied anything for this assignment.

Team Mate 1 : **Anupama S**
Roll No : **EE15B009**
Team Mate 2 : **Sahana Ramnath**
Roll No : **EE15B109**

1 Problem Statement

The goal of this assignment is to model a Recurrent Neural Network (LSTM) and implement a text transliteration system that converts a sequence of characters from one language to a sequence of characters in another language, in this case from English to Hindi. The code for this assignment is written in python using libraries `tensorflow`, `numpy`, `pandas`, `argparse`,`matplotlib`.

We have implemented the DECODER and ATTENTION MECHANISM using basic tensorflow operations and not with the tensorflow seq2seq module.

Dataset : NEWS 2012 (Named Entities Workshop) shared task dataset, containing input words of different lengths. The training set has 13122 datapoints, validation set has 997 datapoints, test set (partial) has 400 datapoints and test set (final) has 1000 datapoints.

2 Contents of the submitted folder

- `create_vocab.py` - python code to create and save english and hindi vocabulary
- `train.py` - python code for training and testing the RNN model
- `train_best_val.py` - python code containing model which gave best validation accuracy
- `train_best_test.py` - python code containing model which gave best public test accuracy
- `train_uni.py` - python code containing model with unidirectional encoder
- `run.sh` - command for running the code with the best hyperparameter configurations
- `plot_loss.py` - python code to plot loss and accuracy plots
- `attention_plots.py` - python code to plot the attention weights on the given test set
- `report.pdf` - report with the required plots, observations and inferences of the results
- `Kaggle_subs/` - predictions `firstbest.csv`, `secondbest.csv` for two best Kaggle submissions

3 Calculation of Loss

The maximum length of output sequence in the train data is 62. However, most data points have between 5-10 characters only, and so cross entropy loss as such will result in the model learning to predict only pad tokens in the output, in order to reduce the loss significantly.

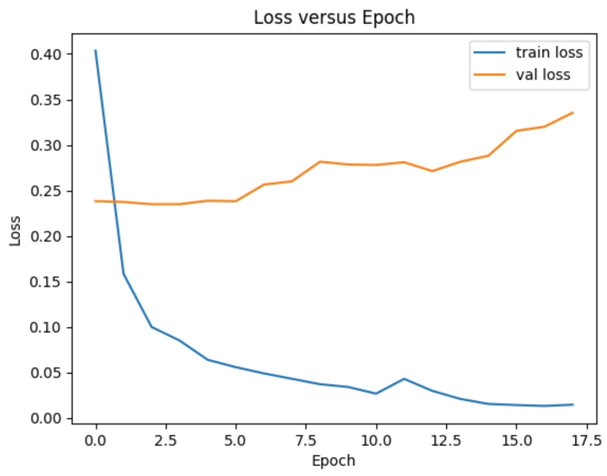
Hence the loss is calculated only till the actual output sequence length, plus one pad token which indicates that the decoder has finished prediction. During test time, the decoder predicts the required output sequence followed by a pad token (which indicates completion of prediction), followed by random tokens till the max sequence length which we can ignore.

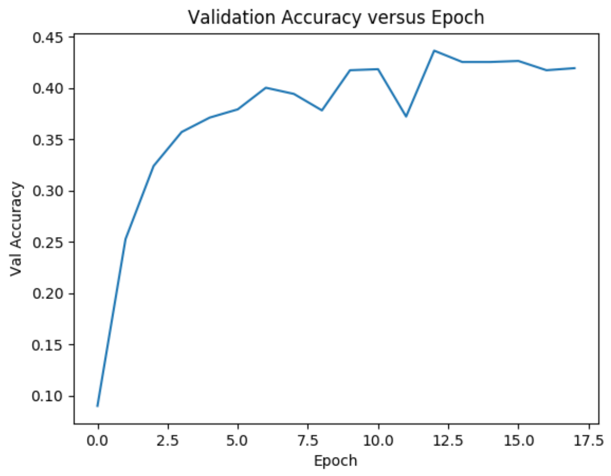
4 Observation and Results

4.1 Plot of training loss and validation loss

This was done for the hyperparameter setting given in the problem statement. Dropout was applied only to the outputs of the LSTM. Attention weights were masked according to actual and pad tokens in the encoder's outputs.

Weight Initialization	xavier
Learning Rate	0.001
Batch Size	60
Dropout Probability	30%
Decoder	2 layers
Validation Accuracy	43.63%





4.2 Best parameter setting (validation data)

Weight Initialization	Xavier
Learning Rate	0.001
Batch Size	60
Dropout Probability	30%
Decoder	1 layer
Validation Accuracy	46.74%

Here, dropout was applied for inputs, outputs and states of the LSTM. Other configurations were default as in the problem statement. Attention weights were masked according to actual and pad tokens in the encoder's outputs.

4.3 Best parameter setting (kaggle final public test data)

Weight Initialization	Xavier
Learning Rate	0.001
Batch Size	60
Dropout Probability	20%
Decoder	2 layers
Public Test Accuracy	48.66%

Here, dropout was applied for only outputs of the LSTM. Other configurations were default as in the problem statement. Attention weights were not masked according to actual and pad tokens in the encoder's outputs. First decoder time step's input to the attention module was the encoder's final state itself and not the projected encoder's final state(which is used as decoder intial state).

4.4 Hyperparameter tuning

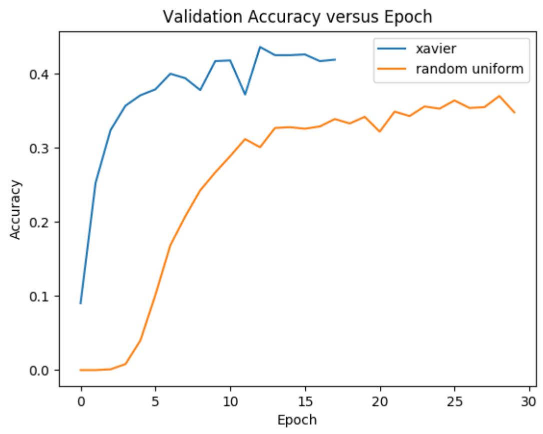
All the following experiments had the following common configuration

- Optimizer : Adam
- Loss function : Cross Entropy
- Early Stopping : Patience of 5 epochs based on validation accuracy
- INEMBED size : 256
- OUTEMBED size : 256
- Encoder Type : Bidirectional
- encsize : 512
- decsize : 512
- Decoder layers : 2 layer stacked decoder used
- FC between encoder and decoder : Was used and the projected encoder state was used as initial state of decoder as well as in the first attention weight calculation
- Dropout : Applied only for the output of the LSTM cell
- Attention : Equations implemented as in slides, and pad tokens were masked while calculating attention weights.
- Non Linearity Used : tanh everywhere

4.4.1 Tuning Weights initialization

Batch size: 60 Learning rate: 0.001 Dropout: 30% Decoder: stacked

Weight Initialization	Validation Accuracy (%)
Xavier	**43.63**
Random Uniform	37.01 (30 epochs)

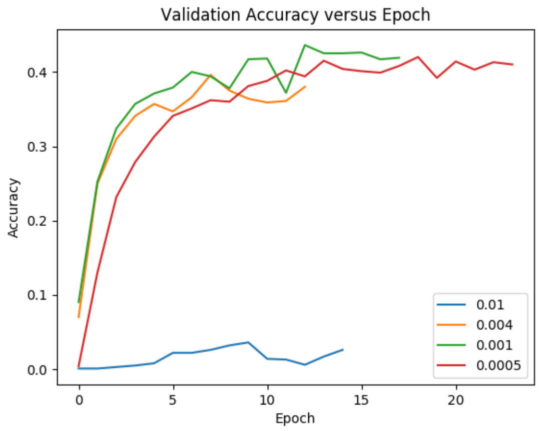


Here, as expected, Xavier performs much better than uniform random initialisation.

4.4.2 Tuning learning rate

Weights init: Xavier Batch size: 60 Dropout: 30% Decoder: stacked

Learning Rate	Validation Accuracy (%)
0.01	3.61
0.004	39.62
0.001	**43.63**
0.0005	42.02

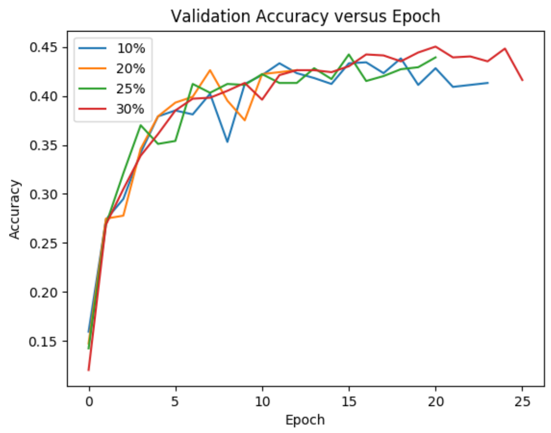


Learning rate of 0.001 along with Adam Optimizer was observed to be the best, significantly.

4.4.3 Tuning dropout

Weights init: Xavier   Learning rate: 0.001   Batch size: 60   Decoder : One layer

Dropout Probability (%)	keep_prob	Validation Accuracy (%)
10	0.9	43.83
20	0.8	42.63
25	0.75	44.23
30	0.7	**45.13**



For this set of experiments alone, we have used a single-layered decoder since it gave more prominent difference between the validation accuracies. As seen above, dropout of 30% gives the best result. This hyperparameter's effects is explained more in Section 4.10.

4.4.4 Tuning batch size

Weights init: Xavier   Learning rate: 0.001   Dropout: 30%   Decoder: stacked

Batch Size	Validation Accuracy (%)
10	41.52
30	43.32
60	43.63
80	**43.8**

Though not very significant, batch size 80 gave the beat validation accuracy. In this assignment, change of batch size did not result in huge changes of final accuracy.

4.5 Seq2Seq Model Structure

Input character sequence → INEMBED → ENCODER → FFNN (optional to project state from *encsize* to *decsize*) → DECODER with ATTENTION → SOFTMAX → OUTEMBED → Output character sequence

4.6 Input and Output dimensions

The following are the input and output dimensions for the GIVEN MODEL.

Layer	Input dim	Output dim
INEMBED	batchsize×input_seq_len	batchsize×input_seq_len×256
ENCODER	batchsize×input_seq_len×256	batchsize×input_seq_len×1024 (fw, bw concatenated)
FFNN(optional)	batchsize×512 (encoder state)	batchsize×512 (c,h states of fw,bw)
ATTENTION	batchsize×1024 (decoder state) batchsize×input_seq_len×1024 (encoder output)	batchsize×256 (c_t)
DECODER	batchsize×512	batchsize×output_seq_len×512
SOFTMAX	batchsize×512	batchsize×output_vocab_size
OUTEMBED	batchsize×output_vocab_size	batchsize×256

4.7 Comparison of unidirectional and bidirectional Encoder

Same common configuration as in Section 4.4.

Batch size: 60   Learning rate: 0.001   Dropout: 30%   Weights init : Xavier
Decoder : 2 layers

Encoder Type	Validation Accuracy (%)
Unidirectional	37.8
Bidirectional	**44.13**

Here, as expected, bidirectional encoder performs **much** better than a unidirectional encoder. This works as in theory; bidirectional encoders contain information about the sequence from both directions; this is useful especially in cases of comparatively long input sequences which we do have in our dataset.

4.8 Attention mechanism

Attention over the encoder's outputs was implemented with **basic tensorflow operations** (**not** with the inbuilt `AttentionWrapper`), with equations as in slides, but dimensions of weights modified in order to to use with a bidirectional LSTM.

Let h_j represent the encoder's output for each input character time step $j \in \{1, 2, .., encoder\_time\_steps\}$. Let s_t represent the decoder's hidden state at the current (decoding) time step t.

$$e_{jt} = V^T \tanh(Uh_j + Ws_{t-1})$$
$$\alpha_{jt} = softmax(e_t)_j$$
$$c_t = \sum_{j=1}^{T} \alpha_{jt} h_j$$

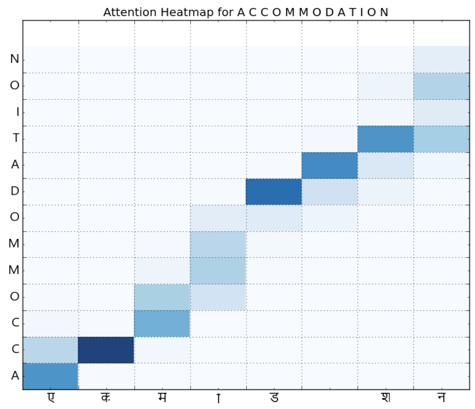
The c_t calculated above is concatenated with the previous decoder output's embedding and is used as the input for the next time step of the decoder.

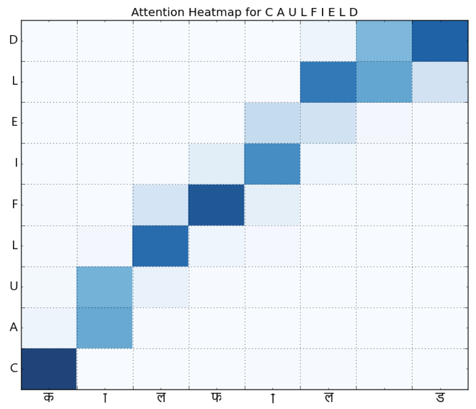
Now, $U \in \mathrm{R}^{2d_1 \times d_2}$, $W \in \mathrm{R}^{2d_3 \times d_2}$, $V \in \mathrm{R}^{d_2}$, where d_1 is the number of units in the encoder, d_3 is the number of units in the decoder and d_2 can be any embedding size. In our code, we have used d_2 as *outembed* itself. Note : For the first time step of the decoder, the encoder's final hidden state must be projected to *decsize* for the dimensions to work out.

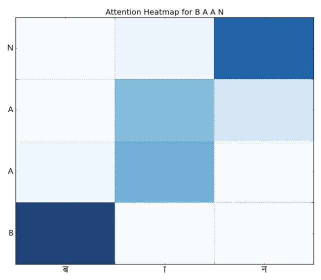
Explanation of Attention Mechanism

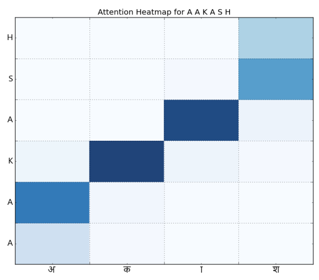
In the above set of equations, at each time step in the decoder, the decoder state (which depicts the output sequence so far) is used to attend the encoder's outputs to assign relative importance to each of the encoder's outputs. The encoder's outputs are then weighed and summed according to these assigned probabilities to give a relevant context vector c_t for the next time step's input.

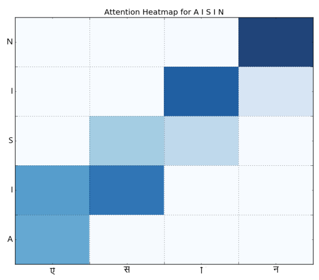
4.8.1 Attention visualization plot

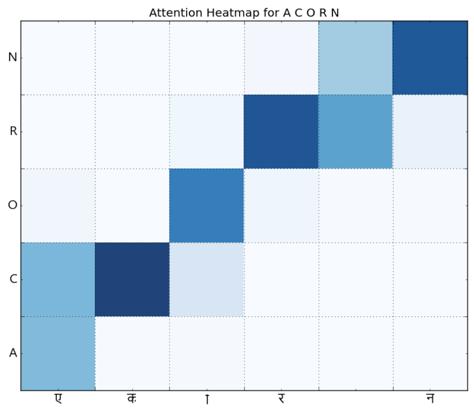


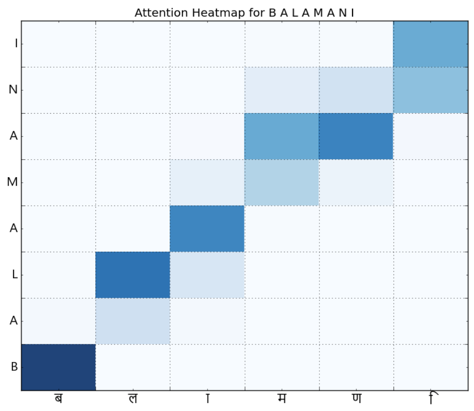












Refer above figure for visualisation of attention weights for different words in the test set. As can be seen in the first two and last two plots, the implemented attention mechanism works well(almost perfectly!) even for relatively long sequences. Most of the attention plots we saw had meaningful character alignments, with only a few characters having the wrong alignments(such as 'NI' in the last plot). There no non-contiguous alignments observed. A lot of the characters have perfect one-one or one-may alignments(see 'AU' in CAULFIELD or 'CO' in ACORN). Even when the one-many alignments are not perfect, the higher probability is almost always assigned to the correct english character.
Note : In some of the plots above, the hindi characters are not perfectly seen due to matplotlib plotting issues; however, we are able to guess the correct hindi character which is predicted there.

4.9 Comparison of 2-layered and single-layered decoder

Same common configuration as in Section 4.4.
Batch size: 60, Learning rate: 0.001, Dropout: 30%, Weights init: Xavier,
Encoder: Bidirectional

Number of layers in Decoder	**Validation Accuracy (%)**
1	**45.13**
2	44.13

Here, using a single layered decoder gives a better validation accuracy than a stacked decoder. In general, we expect a deeper network to perform better, but in this case, the stacked decoder model overfits the given training data and hence performs slightly worse. We observed this trend across different dropout probabilities, batch sizes and learning rates.

4.10 Effect of dropout

Dropout probability tuning can be seen in the table in Section 4.4.3 In our model, we very clearly observed heavy overfitting with less dropout probabilities. In the cases of dropout 0% or 10%, the train accuracy increases very fast as compared to the validation accuracy and by the time val accuracy reaches 40-43%, train accuracy goes to 90-100%. However, in the case of higher dropout probability (till around 30%), the train and val accuracies increase together in the same range, and train data overfits only after a long time.

4.11 Effect of early stopping

Early stopping prevents overfitting of the model beyond a particular point. We gave our model a patience parameter of 5 epochs (based on increasing validation accuracy or decreasing validation loss). In our experiments, after the patience parameter reached its limit, continuing to run the model till the maximum set epoch **rarely** gave better accuracies.

4.11.1 Early stopping methods

Early stopping can be done either based on the validation accuracy or the validation loss at the end of each training epoch.

Early Stopping Methods	**Validation Accuracy (%)**
Based on Validation Accuracy	**44.13**
Based on Validation Loss	34.5

As seen in the above table, early stopping based on accuracy results in much better final saved model than loss. Even though the validation accuracy increased till 39.5% in the second case, when the patience parameter reached 5 due to validation loss increase, the algorithm only recognises the model with the lowest validation loss (with accuracy 34.5%) as the best one.

Though in theory, loss and accuracy are supposed to be inversely correlated, in practice, in both this and the previous CNN assignment, we found out that after a point, validation loss remains either constant or increases slightly (probably due to overfitting), but the validation accuracy still increases significantly. Since our final evaluation of a model is the accuracy on the validation/test data, early stopping based on accuracy is a better method to choose.

5 Conclusion

In this assignment, we implemented a recurrent neural network using tensorflow for various hyperparameter settings and other techniques such as early stopping, dropout, attention, uni/bi-directional encoders and stacked/non-stacked decoders. We have analysed and commented upon the trends observed with changing all of the above. We have also plotted the observed attention weights and commented on the patterns in it.

As expected, best results were given by using Xavier initialization, early stopping based on validation accuracy, bidirectional encoder and with attention over the encoder's outputs. Again, just making the network deeper, for example by using a decoder of 2 layers instead of 1 did not help; it led to more overfitting. Dropout of 30% gives the best results by preventing heavy overfitting.

We have written our own decoder and attention mechanism. Our attention mechanism's equations have been mentioned above as well. Observing the attention plots, we can see that, for the most part, the model has learnt proper correspondence between the english and hindi letters.

6 Versions of Libraries used

Following are the versions of libraries needed to run and reproduce the above results.

- python 2.7.12
- numpy 1.13.3
- tensorflow 1.10.1
- pandas 0.22.0
- matplotlib 1.5.3