EE5178 Modern Computer Vision Assignment 1

Vaibhav Mahapatra
Mechanical Engineering Department
Indian Institute of Technology, Madras
Chennai, India
me19b197@smail.iitm.ac.in

Abstract

The aim of this assignment is to experiment Please use Pytorch, an open source library for implementing deep learning models, to write your code. You can use any other libraries that you may feel are necessary.

In this project, I experimented with Multilayer Feedforward Neural Network (MLP) and Convolutional Neural Networks (CNN) architectures for image classification. The models were trained and evaluated on the CIFAR-10 dataset. This dataset contains 60000 32 × 32 color images in 10 different classes, with 6000 images per class.

Additionally, in the section, we build an RNN that tackles Language Transliteration problem using sequence to sequence models. Trained on the Google Dakshina Dataset, I'll analyse the performance of 2 variants of RNNs on translating English words to Hindi words.

Contents

I. CIFAR-10: Dataset for Image classification

The CIFAR-10 dataset. consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The 10 classes are namely: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

This dataset is a fairly common image dataset used for image classification. It can easily be downloaded as a PyTorch dataset from the torchvision.datasets. The MLP and CNN will be trained and evaluated on the training and test sets of this dataset

II. Multilayer Feedforward Neural Network (MLP)

A. Architecture

I created an NLP with the below architecture:

- Input layer: 32 x 32 x 3 = 3072 units
- Hidden1: 500 units
- Hidden2: 250 units
- Hidden3: 100 units
- Output: 10 units

Between each of the dense layers, I applied the ReLU activation function. It's a non-linearity described in Fig. 1

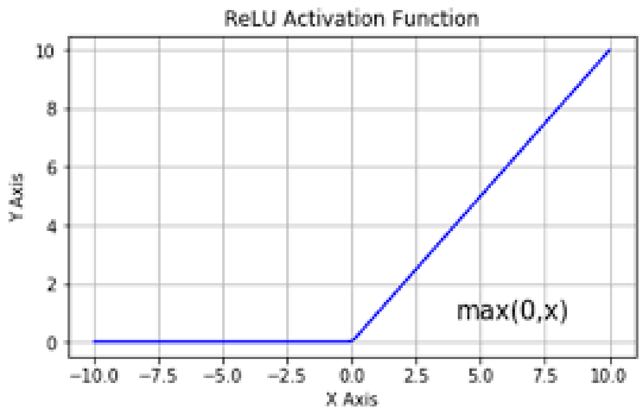
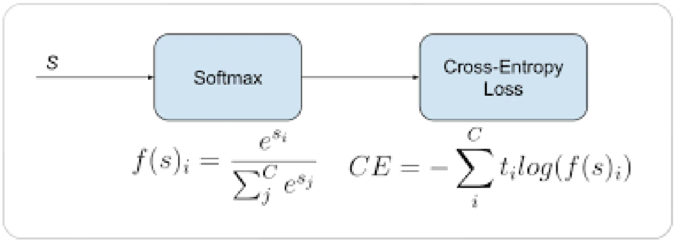


Figure 1. ReLU activation function

After the final dense layer, the softmax activation layer is used to generate a set of 10 probabilities, corresponding to each class.

B. Loss Function

The Cross entropy loss is used to optimize the Neural Network. For each datapoint, it compares the softmax output probabilities with a one-hot vector of the class the datapoint actually belongs to. Minimzing the cross entropy loss (denoted in Fig. 2) essentially maximizes the probability that each datapoint belongs to the assigned label.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

Figure 2. Cross Entropy Loss

C. Training Set-up

The training setup included the below hyperparameters:

- Optimizer: SGD Optimizer
- Epochs: 200
- Learning Rate: 1e-3 (0.001)

When trained with these hyperparameters, the model took 35 minutes to train.

D. Model performance

The growth of training and validation errors and accuracies are depicted in Fig. 3. We can see that after about 100 epochs, the model start overfitting. This is because the training loss keeps decreasing while the Validation loss saturates and oscillates.

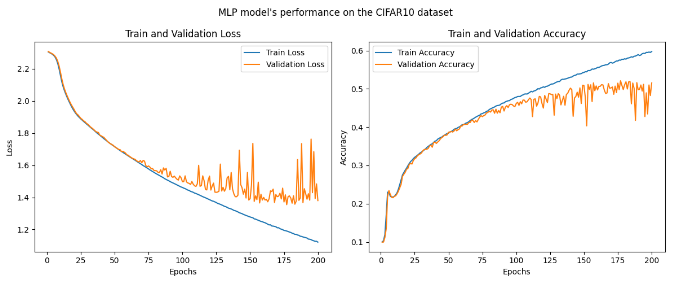


Figure 3. MLP loss growth

Test prediction's accuracy of this model = **47.9%**. Some sample predictions of this model are visualised in the Fig. 4

The confusion matrix depicting the class-wise prediction of the model can also be seen in Fig. 5. From the matrix, it is evident that the model makes errors like: predicting cat for dogs, frog for deer and ship for airplane. In general, this model performs best for the class "frog."

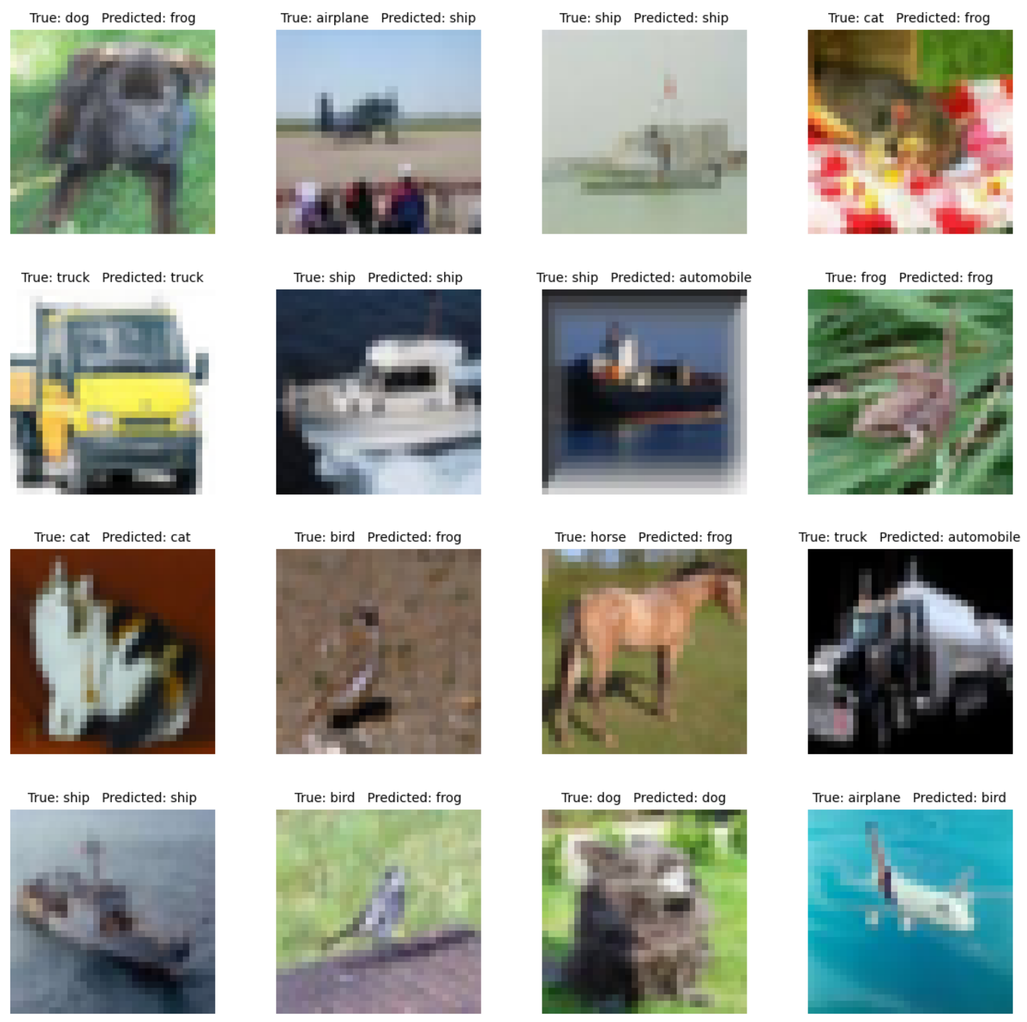

True: dog Predicted: frog
True: airplane Predicted: ship
True: ship Predicted: ship
True: cat Predicted: frog
True: truck Predicted: truck
True: ship Predicted: ship
True: ship Predicted: automobile
True: frog Predicted: frog
True: cat Predicted: cat
True: bird Predicted: frog
True: horse Predicted: frog
True: truck Predicted: automobile
True: ship Predicted: ship
True: bird Predicted: frog
True: dog Predicted: dog
True: airplane Predicted: bird

Figure 4. Sample Predictions of the MLP

E. MLP with Batch Normalization

I then implemented an MLP with BatchNorm1d after every dense layer. The loss and accuracy curves are much smoother than before, as can be seen in Fig. 6. Additionally, training this network took $1/10^{th}$ the time. I was able to achieve a 54% test accuracy by just training the network for 20 epochs in 3.5 minutes.

F. Key takeaways

1) Although MLPs are able to classify some image data, their performance saturates early. Deeper Neural Networks (NNs) may capture data better, but will take longer to train.
2) The oscillation observed in the vanilla MLP's validation accuracy may be due to the behaviour of the SGDOptimizer. More sophistication learning algorithms like Adam, which incorporate learning rate decay and momentum, can help mitigate this.
3) BatchNorm significantly improves the training speed and performance of an MLP. It also introduced a regularization effect which ensure that the model does not overfit on training data
4) Flattening the image comes at the cost of losing spacial information. Information regarding the relative position of different objects in an image gets lost in the process. Hence, to better target this problem, a Convolutional Neural Network will be a better architecture.

III. VGG11: CNN for Classification

I assembled the architecture denoted in Fig. 7 Subsequently, I played around with parameters like learning_rate, no. of hidden layers and units in the Feed-forward section of the architecture to build an effective VGG11 architecture.

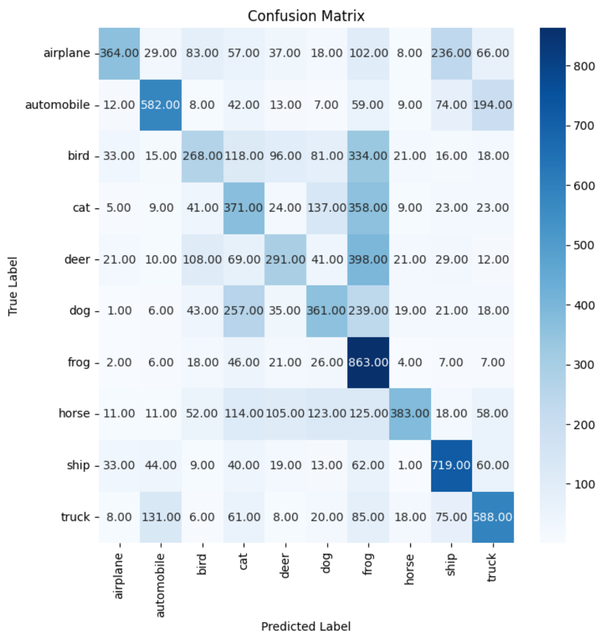


Figure 5. Confusion Matrix of MLP

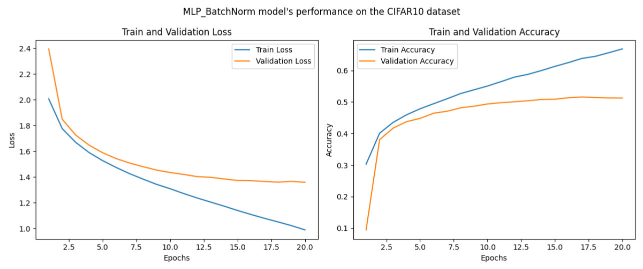


Figure 6. Performance of MLP with BatchNorm

A. Experiment 1: Learning Rate tuning

I chose to evaluate the model's performance on 4 learning rates: [0.0005, 0.001, 0.005, 0.01]. I trained each model for 150 epochs and observed the growth of training and validation accuracy of each of the models. The graph Fig. **??** showcases just this. My main takeways from the graphs are:

- The model doesn't seem to be learning for the first 50% of training. This might be due to the fact that the data is not distributed evenly in the parameter space. As a result of this, gradients in the starting epochs are low and the parameters is negligibly updated.
- Among all the learning rates, **0.01** stand out by learning the fastest
- The 0.01 lr model learn fast but also saturates and overfits quickly. Some sort of regularizer should help mitigate this.

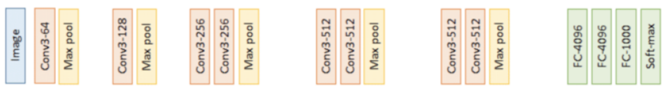


Figure 7. VGG11 architecture

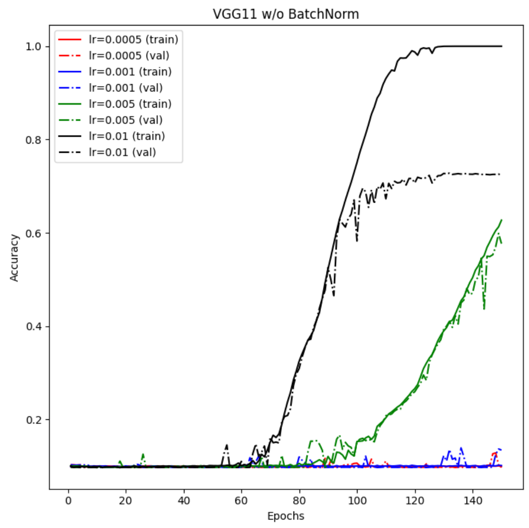


Figure 8. Hyperparameter tuning: Learning rate

B. Experiment 2: Addition of BatchNorm

In the VGG11 architecture, I've added BatchNorm throughout the architecture to act as a regularizer while simultaneously tackle the low gradients. As a result, the model trains within 25 epoch, which is much faster than the earlier model. Among various learning rates, **0.01** proves to be the best one in Fig. 9

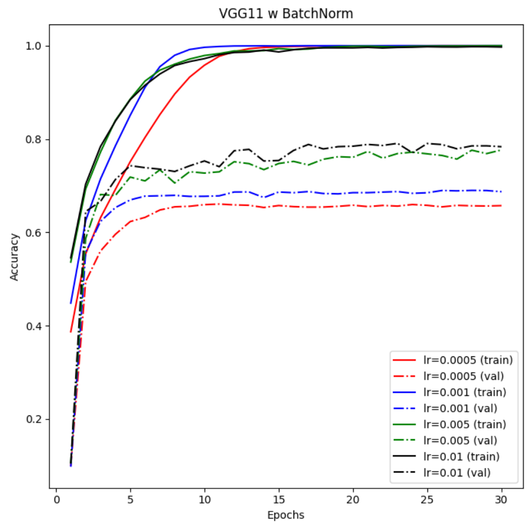


Figure 9. Adding BatchNorm: Effect on training speed

C. Experiment 3: No. of hidden layers in FCN

In the Fully connected network region of the VGG11 architecture, I've tested 3 networks of various depths: [1, 5, 10]. There doesn't seem to be a drastic difference in the model's performance in Fig. 10. I'll choose **n_hidden = 5** as the best parameter for this case.

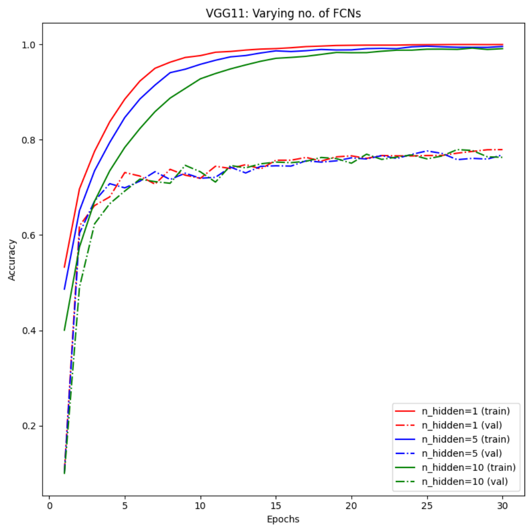


Figure 10. Hyperparameter tuning: No. Hidden layers

D. Experiment 4: No. of units in hidden layers

In the Fully connected network region of the VGG11 architecture, I've tested 4 sets of neurons: [1024, 2048, 4096, 8192]. This factor also doesn't seem to drastically affect the model's performance, as seen in Fig. 11. I'll choose **n_units = 2048** as the best dimension for the final model

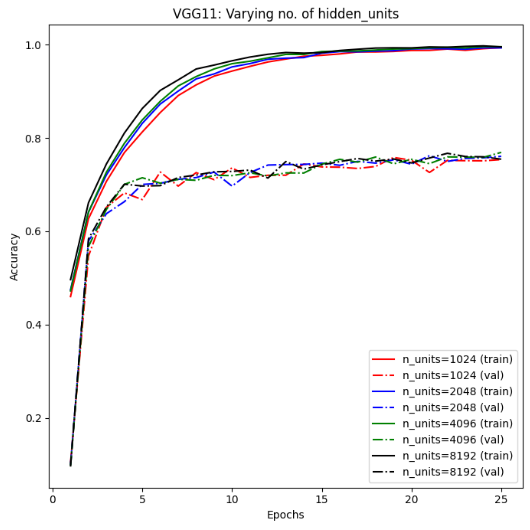


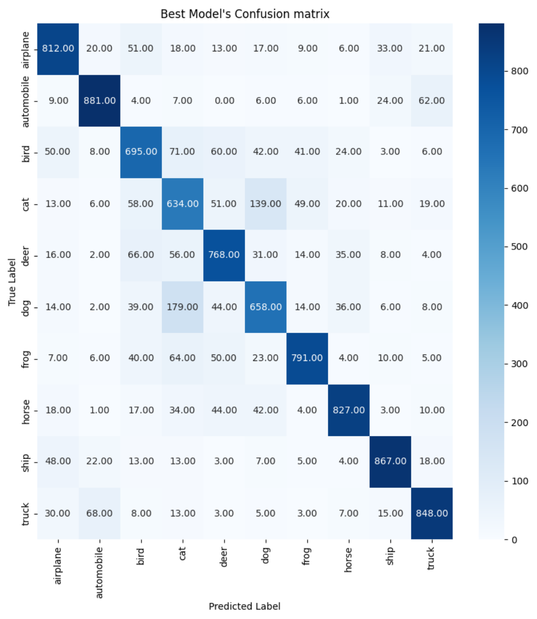
Figure 11. Hyperparameter tuning: No. Hidden units

E. Final Model

The best VGG11 model based on my tests has the following hyperparameters:

- Includes Batch Normalization
- Learning rate = 0.01
- No. Hidden Layers in FCN = 5
- No. of Hidden units in each hidden layer = 2048

This configuration attains a test accuracy of 70% and yields the confusion matrix in Fig. 12. This confusion matrix when compared with Fig. 5 shows superior and more accurate results.



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	812.00	20.00	51.00	18.00	13.00	17.00	9.00	6.00	33.00	21.00
automobile	9.00	881.00	4.00	7.00	0.00	6.00	6.00	1.00	24.00	62.00
bird	50.00	8.00	695.00	71.00	60.00	42.00	41.00	24.00	3.00	6.00
cat	13.00	6.00	58.00	634.00	51.00	139.00	49.00	20.00	11.00	19.00
deer	16.00	2.00	66.00	56.00	768.00	31.00	14.00	35.00	8.00	4.00
dog	14.00	2.00	39.00	179.00	44.00	658.00	14.00	36.00	6.00	8.00
frog	7.00	6.00	40.00	64.00	50.00	23.00	791.00	4.00	10.00	5.00
horse	18.00	1.00	17.00	34.00	44.00	42.00	4.00	827.00	3.00	10.00
ship	48.00	22.00	13.00	13.00	3.00	7.00	5.00	4.00	867.00	18.00
truck	30.00	68.00	8.00	13.00	3.00	5.00	3.00	7.00	15.00	848.00

Figure 12. Confusion Matrix of the best VGG11

F. Performance on each class

G. Predictions on real images

IV. Sequence to Sequence modeling with Recurrent Neural Networks (RNNs)

A. Dakshina Dataset: Language Transliteration

The Dakshina Dataset by Google contains pairs of words where the first element is a word in the native script and the second word is the corresponding transliteration in the Latin script. If such a pair is denoted (x_i, y_i), then given many such pairs, I am training an RNN to model the $\hat{y}_i = f(x_i)$. 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.

I have chosen Hindi (following the Devanagari script) as the output language that is transliterated into English.

B. Architecture: Version 1

1) An input *Embedding* Layer that generates character embeddings
2) Encoder RNN - sequentially encode the input English letters
3) Decoder RNN - takes the last state of the encoder as input and produces one output character at a time. The decoder obeys the following equation:

$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$s_o = h_T$$

I've taken the following hyperparameters for training the RNN:

- Optimizer: Adam (learning rate = 0.01)
- Embedding size: 128
- Hidden size: 256
- No. of layers: 4

C. Performance of RNN_v1 on the Dataset

The loss function used to evaluate this dataset is the *Negative Log Likelihood* loss. This loss when applied to a vector of probabilities, essentially forms the Cross-Entropy Loss. To evaluate the model's performance, measured 2 kinds of accuracy: word-to-word accuracy for predictions matching the actual word, and a Levenshtein (edit-distance) based accuracy. The visual representation of these results can be seen in Fig. 13.

This model was trained by passing 200,000 pairs of English-Hindi words. This would indicate roughly 6 passes over the training dataset. It's performance on certain metrics are as follows:

- Training Loss = 1.367
- Accuracy = 8% (train) and 6% (val)
- Levenshtein accuracy = 56.6%

D. Architecture: Version 2 & its performance

The same problem is tackled again with a slightly different Decoder. This Decodes takes the final hidden state of the Encoder as an input at every decoding state. It's governing equations are given by:

$$s_t = RNN(s_{t-1}, [e(\hat{y}_{t-1}), h_T])$$
$$s_o = h_T$$

With the same parameters, in Fig. **??** this RNN outperforms the previous architecture by achieving the following metrics:

- Training Loss = 1.106
- Accuracy = 15% (train) and 11% (val)
- Levenshtein accuracy = 64.8%

This improved performance implies that the encoder's output, when fed continuously to the decoder, is able to guide the prediction better.

Some interesting predictions on the dataset by both versions are visible in Fig. 15

E. Main takeaways from RNNs

1) a simple RNN is able to capture some sequential dependency given than it's able to predict 400 out of 4000 words correctly
2) the RNN can be improved by potentially making it bidirectional
3) More complex RNN architectures like LSTMs, GRUs may drastically improve performance. Additionally, using attention maps can help us engineer better architectures for the data.

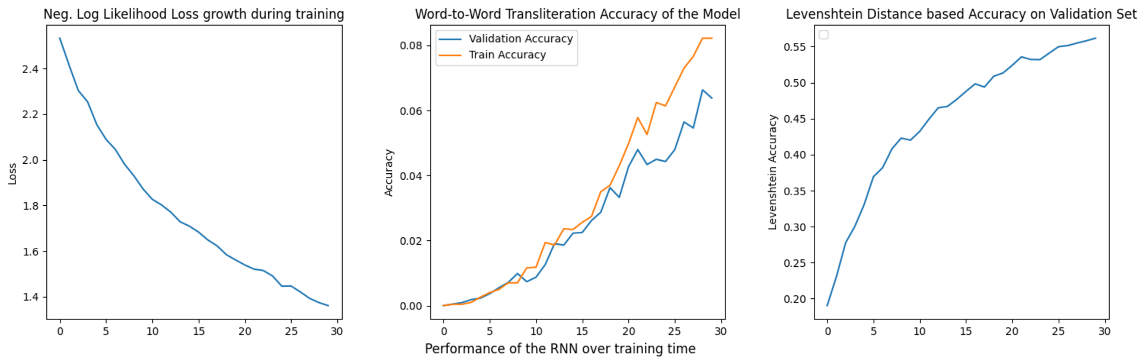


Figure 13. RNN_v1 performance

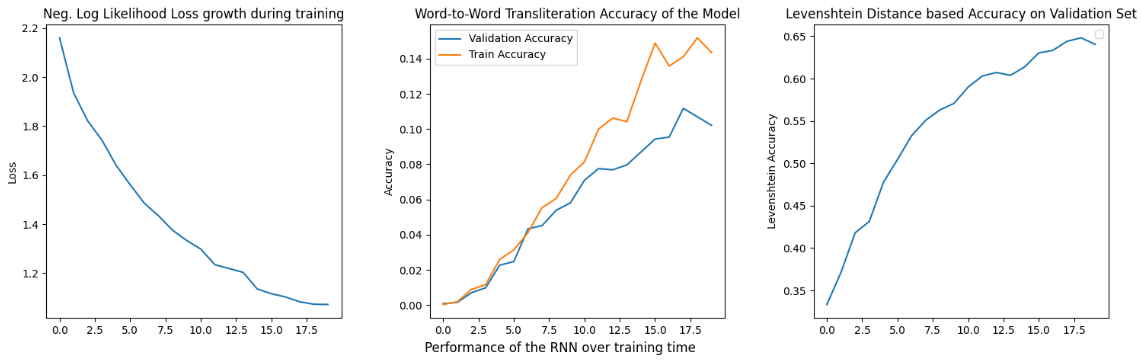
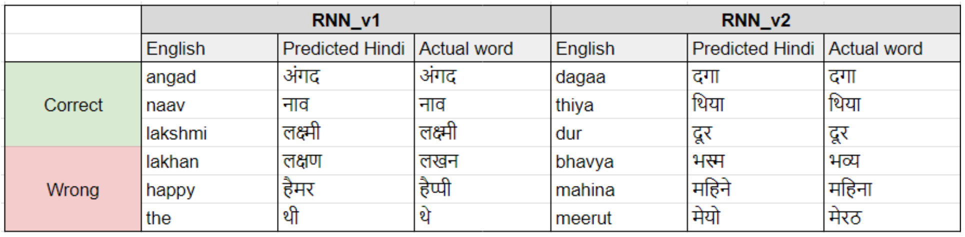


Figure 14. RNN_v2 performance



	RNN_v1			**RNN_v2**		
	English	Predicted Hindi	Actual word	English	Predicted Hindi	Actual word
Correct	angad	अंगद	अंगद	dagaa	दगा	दगा
	naav	नाव	नाव	thiya	थिया	थिया
	lakshmi	लक्ष्मी	लक्ष्मी	dur	दूर	दूर
Wrong	lakhan	लक्षण	लखन	bhavya	भस्म	भव्य
	happy	हैमर	हैप्पी	mahina	महिने	महिना
	the	थी	थे	meerut	मेयो	मेरठ

Figure 15. Some predictions on the test dataset