Ablation study of Twin Networks: Matching the Future For Sequence Generation

Mehrzad Mortazavi
Mehrzad.Mortazavi@mail.mcgill.ca

Mandana Samiei
Mandana.Samiei@mail.mcgill.ca

***Abstract*—The teacher forcing algorithm trains recurrent networks by feeding the input sequence during training and using the network's one-step ahead predictions to do multi-step sampling. In this paper, we experiment the effectiveness of "backward" recurrent network in sequence generation, named "TwinNets". We considered an unconditioned generation task by using sequential MNIST dataset and achieved 80.52 nat score by tweaking the model.**

I. Introduction

Recurrent Neural Networks (RNNs) are usually trained by teacher forcing at each point in a given sequence, that means the RNN is optimized to predict the next token given all preceding tokens. This corresponds to optimizing one-step ahead prediction and the model may prefer to focus on the most recent tokens instead of capturing subtle long-term dependencies that could contribute to global coherence. As a result, samples from RNNs tends to exhibit local coherence but lack meaningful global structure. This difficulty in capturing long-term dependencies has been discussed in several works (Bengio et al., 1994; Pascanu et al., 2013). In Twin Networks paper, the authors mitigate this issue by running a backward network to anticipate future states. However, their concentration was mainly on the conditioned sequence generation problems such as Speech Recognition and Image Captioning. TwinNets achieved 9% improvement through these 2 tasks. In case of sequential MNIST, they had a lower performance with respect to the baselines. PixelRNN with 7 layers (Oord et.al, 2016b) and PixelVAE (Gulrajani et al., 2016) perform 79.20 and 79.02 over NLL while the baseline LSTM with added 3-layer TwinNets gained 79.35. This indicates TwinNets needs still to get some improvement to outperform these baselines, however, there is a merit to use TwinNets, since the backward network is discarded during sampling, the inference process has the exact same computation steps as the baseline. This makes the architecture more applicable.

The sequential MNIST data set always forces the model not to see the whole image at once, but only one pixel at a time. We are using the standard binarized sampled version of MNIST dataset proposed by Serdyuk et al. (2018). It includes 50000 samples for train, 10000 for validation and 10000 for test. However, due to time and computational resources limitations, we used minimum samples that covers the most features of the whole dataset. To find the best this ratio, We did some experiments which are explained in following sections.

II. Potential Challenges

The original paper we are doing ablation study is experimented on 4 applications of Speech Recognition, Image Captioning, Sequential Image Generation and Language Modeling. We decided to start with sequential Image Generation task by using MNIST dataset, since it was more interesting to the group and also to have the opportunity of improving the performance; since TwinNets has not outperformed the state of the art in Sequential MNIST problem, we decided to focus on this task in depth.

Another challenge we confronted was poor quality of the provided code on the Github repository. We had to go over the code and compare the files multiple times to understand the meaning of each variable, class, or function and their rule in finding the best performance of the model that was very time consuming.

Furthermore, the training phase was computationally expensive since we required to run a backward network with the same size of the forward one and it doubles the computations with respect to the other baselines training time.P However, this is not the case in the inference time since the backward network is discarded by the time of sampling or inference task. Also, this was the reason of that we asked for more computational resources to run the experiments.

III. Algorithms

A. Recurrent Neural Networks (RNN)

RNN is a neural network where connections between units of one data set form a directed graph along a sequence. The idea behind RNNs is make use of sequential information because of its ability to exhibit dynamic temporal behavior for a time sequence. Different from feed forward neural networks, RNNs can use their internal state (memory) to process sequences of inputs which the output being depended on the previous computations. This makes them applicable to tasks such as Speech Recognition, Image Captioning and sequential MNIST.

For the Speech Recognition task, how RNN operate over text is shown as the following:

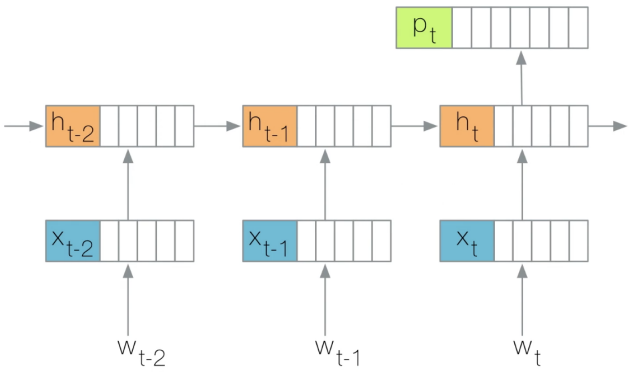


Fig. 1: a RNN operation over text

We assume that given a text corpus represented as sequence of words over time.

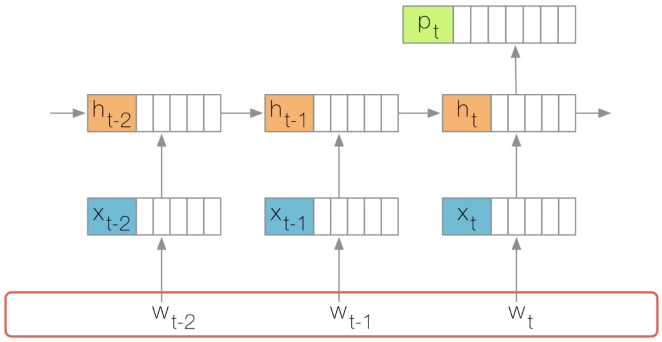


Fig. 2: a sequence of words

each word w is represented by a numerical vector x

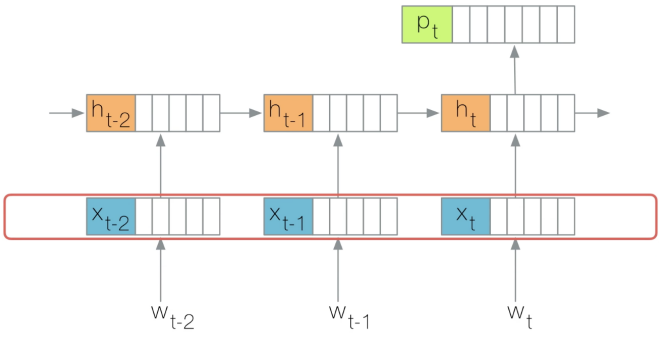


Fig. 3: a sequence of numerical vector x

the word vector is the input to the current network which transform the words into a hidden vector h, the hidden state vector h_t is a function of the word vector x_t and the previous state vector $h_t - 1$.

These hidden state vectors form an abstract representation of the text corpus, this representation can be used in many

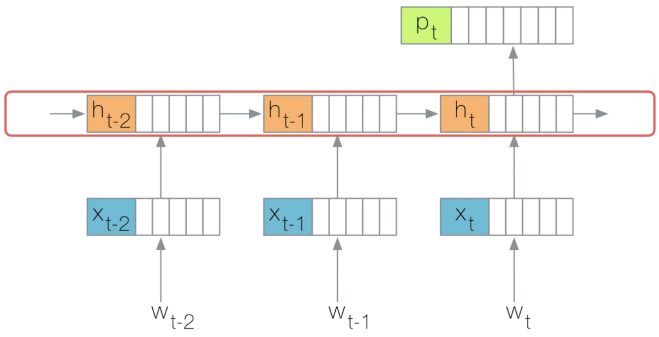


Fig. 4: a sequence of hidden vector h

different task such as predicting the next word in the sequence, which is known as language modelling.

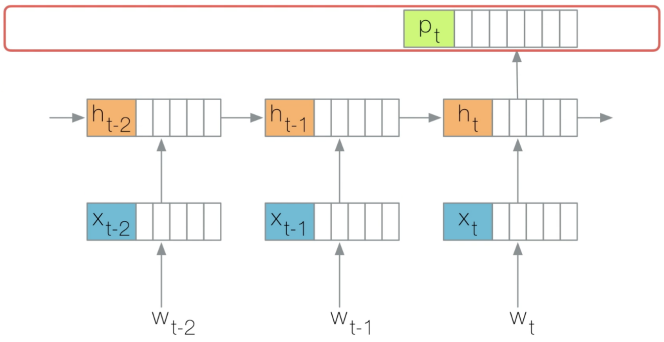


Fig. 5: prediction

B. Fully Connected Feed Forward Neural Network

A feed forward network implies absence of feedback or recurrent connections, it is essentially the opposite of a RNN. The path is only forward facing, no backward feed connections between neurons are present. A fully connected feed forward neural network can be described as a fully connected layer in a neural network where one such that every input neuron is connected to every neuron in the next layer. This, for example, contrasts with convolutional layers, where each output neuron depends on a subset of the input neurons. It is important to know that in a feed forward network the gradient is clearly defined and computable through backpropagation (i.e. chain rule), and on the other hand, for a RNN the gradient computation requires, most likely, an infinite number of operations, thus one usually have to limit the training of a RNN to a fixed number of steps, and it is also more expensive in any case.

C. Long Short-Term Memory Units (LSTMs)

A LSTM is a variation of RNNs which help preserve the error that can be backpropagated through time and layers also known as the long term dependency problem, LSTMs were developed to deal with the exploding and vanishing gradient

problem when training traditional RNNs. It is composed of a cell, an input gate, an output gate and a forget gate, there are connections between these gates and the cell. The cell is responsible for memorizing values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates compute an activation by using an activation function of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". By maintaining a more constant error, LSTMs allow RNNs to continue to learn over many time steps, thereby opening a channel to link causes and effects remotely. A LSTM is good for classify, process and predict time series given time lags of unknown size and duration between important events.

D. LSTM with an Embedded Layer

The Embedding Layer is used to create word vectors for incoming words. It sits between the input and the LSTM layer, which means the output of the Embedding layer is the input to the LSTM layer. The idea came from if one think of how to provide a word as an input to LSTM, one will realize it is simply a network of matrix multiplications and additions and floating functions, which require some integer or float based representation of words to feed into your network. The most logical representation would be to represent each word by an integer and since vocabulary of your data set is finite, you would have a finite set of integers representing the words. In order to have a representation where similar words have smaller euclidean distance than the ones which are not similar at all. The similarity is based upon what conditions or surroundings is the word used in. This representation is called word embedding. During, the forward pass of training (or test), a lookup of the words is done from this embedding layer to get a particular word embedding. During backward pass, the gradients of the loss function also flow through the embedding layer, thus learning the word embeddings most suitable for the task at hand.

E. TwinNet

The idea behind the TwinNet is to regularize an RNN that promotes modeling the aspects of the past that are predictive of the long-term future. To do so, we run a backward recurrent neural network (without parameter sharing) that predicts the sequence in reverse in parallel to the standard forward recurrent neural network. During the training process, we encourage the hidden state of the forward RNN to be as close as possible to the backward network used to predict the same token. This should force the forward network to focus on the past information that is useful to predicting a specific token, which is also present in and useful to the backward network coming from the future.

We can visualize this method by looking at the following figure, the forward and the backward RNN predict the sequence $s = x1, ..., x4$ individually. Then the penalty matches the two parametric function of the forward and the backward hidden states, which led to the forward RNN receiving the gradient signal from the log-likelihood objective as well as L_t between states that predict the same token. For the backward RNN, it is only trained in order to maximize the data log-likelihood. The colored with orange part of the figure is discarded during the evaluation. The cost L_t comes out to be either a Euclidean distance or a learned metric $||g(h_t^t) - h_t^b||_2$ with an affinity g. (For further explanation on the model and the math behind it, refer to Serdyuk, D., et al. 2017)

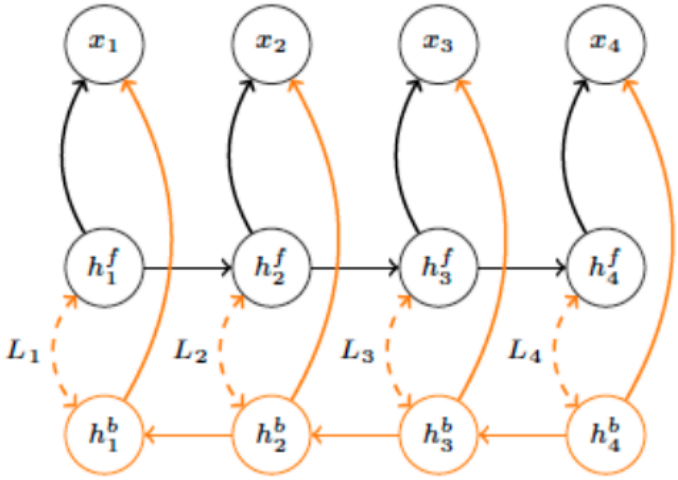


Fig. 6: Twin Network Architecture

IV. Methodology

In the following sections, we have explained our methodology to find the best hyperparameters of our models using Negative Log-Likelihood (NLL) of the validation and test sets. As a general rule, we started by getting some intuitions of behaviour of the model and based on the observations, we proposed some assumptions, trained a new model to test our hypothesis, and evaluating and validating our assumptions to decide for the next experiment. Sometimes results of the experiments were not clearly preferable over the others, since their final NLL were close. In these cases, we analyzed the results and learning curve to come up with the best explanation to select the best hyperparameter configuration. We tried to give each model maximum possible time to evolve, but if in the last epochs the model stopped improving, we considered that as the final performance of the model and terminated the experiment at that point. Typically, a few percentage difference meant that the hyperparameter was not essential for improving the performance and the network was not modified. In this section we have categorized the experiments into architecture modification, hyper-parameter tuning, dataset exploration. As seen in the following sections, we first examined the model by tweaking ratio, second, we experimented hyperparameters tuning who was the most effective ones of the model, we called them first-order hyperparameters, afterwards, we investigated optimizer and twin hyperparameters who are of the architecture. As the last experiment, we tried second-order hyperparameters who have the lower priority. In all of our experiments, validation and test NLL was almost same and

A. First Round Experiments: Dataset Exploration

In the first stage, we planned to figure out how much data do we need to get the same result as using whole data set for training. There is a variable called "ratio" determining the ratio of the whole training set to be used. We trained the TwinNet model and evaluated it on the validation and test sets. During all these experiments learning rate was 0.0001 and other settings kept constant, only the value of the ratio was changed. We allowed the model to evolve and values of weights was updating until negative log-likelihood converged and the difference of NLL values between the last two epochs was less than 0.01. Table 1 shows the result after each experiment. According to this experiment, we chose 0.5 as the value of ratio for our next round experiments to be able to train and evaluate more models in this limited time and shortage of computational resources; furthermore, in comparison with ratio equals to 0.1, it has less variance and covers as enough as possible features of the whole dataset. As a result, we are using 25000 samples for training, 10000 samples for validation and 10000 for test.

TABLE I: Ratio Tuning

Ratio	Validation/Test Negative Log Likelihood (NLL)
1.0	81.657
0.5	82.615
0.1	92.682

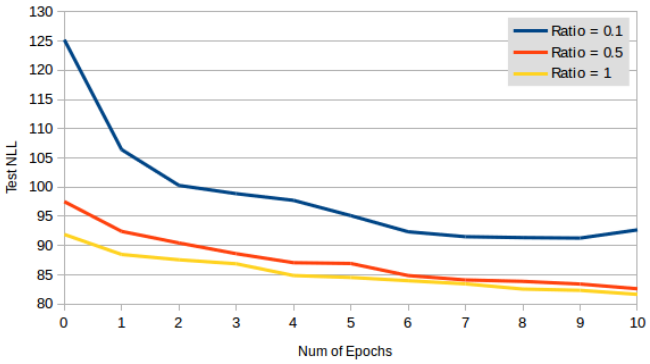


Fig. 7: Ratio Experiment

B. Second Round Experiments: Tuning hyperparameters of the model

Given a large number of hyperparameters in most neural language models, the process of tuning models for new datasets can be laborious and expensive. In this section, we attempt to determine the importance of each hyperparameter by training and experimenting 14 models. Each model has trained 25 epochs, and we evaluated the models by looking at the Negative Log-Likelihood on the validation and test set. At first, we tried different values of learning rate since according to our experiments, it has more effect on the result than the other hyperparameters. If we didn't find the appropriate learning rate, the model couldn't learn the data well, as a result, the model may prune to under-fitting. For example, we tried learning rate equals to 0.01 when the optimizer is Adam and observed that the NLL starts from a very large value equals to 172.748 and converges to 84.78 after 25 epochs, at this point we terminated the experiment since we inferred the model in under fitted and probably needs more time to reach the optimal minimum. After training 3 models, we chose 0.001 as the constant learning rate through the next experiments, although LR equals to 0.0001 gives a lower negative log-likelihood (NLL), we observed if we start with a bigger learning rate and decrease it through the time, we achieve a better performance due to the possibility of jumping out from local minimum/maximum. After choosing learning rate, we experimented dropout in the range of 0, 0.05, 0.10, 0.25, to define the range we started from 0 and increase the drop out by 0.05, as we increased the dropout,we observed adding dropout gives almost 0.4 improvement; however, the improvement decreases by approaching values higher than 0.10 since the capacity of the model decreases by adding more dropout, NLL increases consequently. According to the NLL values, we selected dropout equals to 0.05 as the constant one to the next experiment. Here, the dropout corresponds to the hidden layer. Afterward, we tried 2 values for the number of layers, we chose the range of values according to Merity et.all, we observed the number of layers equals to 3 gives a lower error but on the other hand the convergence time was very long so we selected number of layer equals to 2 for the next examinations. The overall trend of NLL variations over learning rate, dropout and the number of layers can be seen in Figures 8, 9, 10

TABLE II: First-order Hyperparameter Tuning

Hyper Parameter	Tested values	NLL	Selected Value
Lr	0.01, 0.001, 0.0001	81.787, 81.214, 83.189	0.001
Dropout	0, 0.05	81.698 , 81.595	0.05
	0.10, 0.25	81.698, 81.586	
Num.of Lr.	2, 3	81.15, 81.79	2

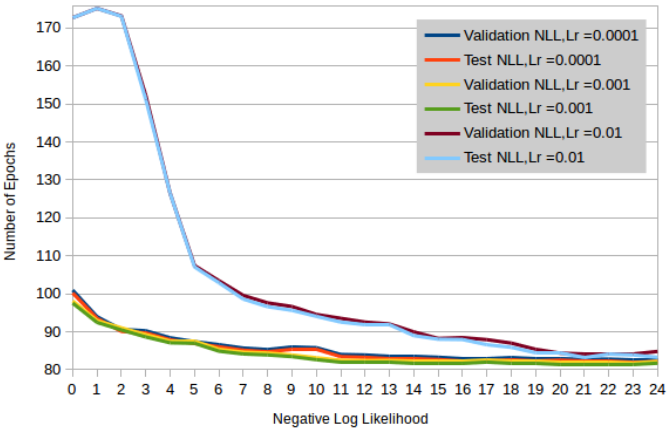


Fig. 8: Learning Rate Experiment

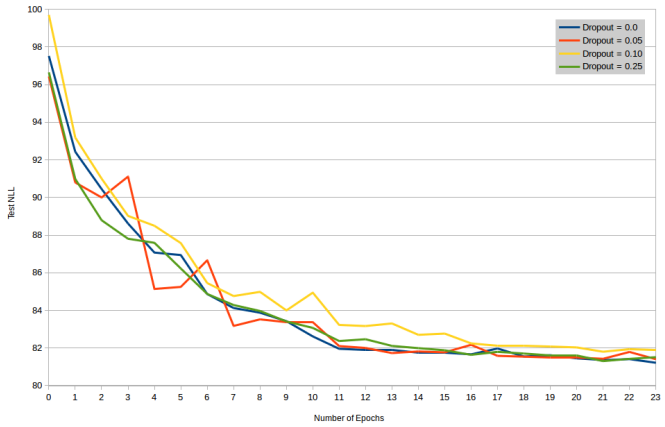


Fig. 9: Dropout Experiment

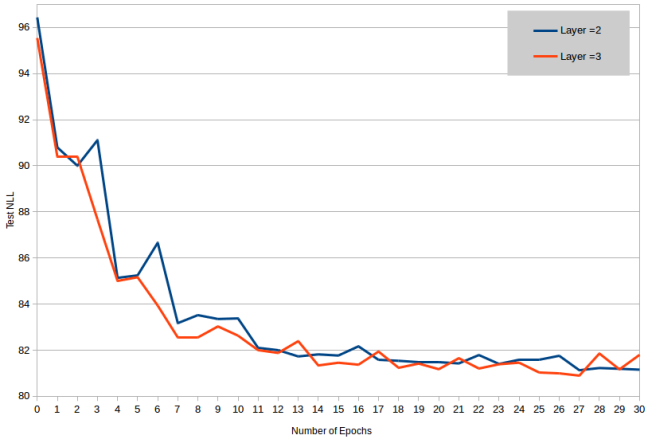


Fig. 10: Number of Layer Experiment

C. Third Round Experiments: Architecture Modification

1) Optimizer: Third experiment was done by taking the TwinNet model and hand tuning hyper-parameters until an optimal result was found. The results were done by taking the training set and performing an experiment on Adam and SGD with Momentum optimizer. The evaluation of each variable was performed with 30 epochs, which looked at Negative Log-Likelihood on the validation and test sets. The overal trend of NLL variations can be seen in Figure 11. According to the figure, we chose Adam optimizer as the best one to keep it constant in the next stage of our investigations.

TABLE III: Optimizer Tuning

Hyper Parameter	Tested values	NLL	Selected Value
Optimizer	Adam, SGD with Momentum	81.426, 84.394	Adam

2) Twin Regularization Parameter: In this section, we tried different values of Twin hyperparameter to see how the model evolve by keeping all other variables fixed. Twin hyperparameter is the TwinNet regularization cost that regularizes the hidden states of the network to anticipate future states. We compared the performance of the model by adding twin hyper-

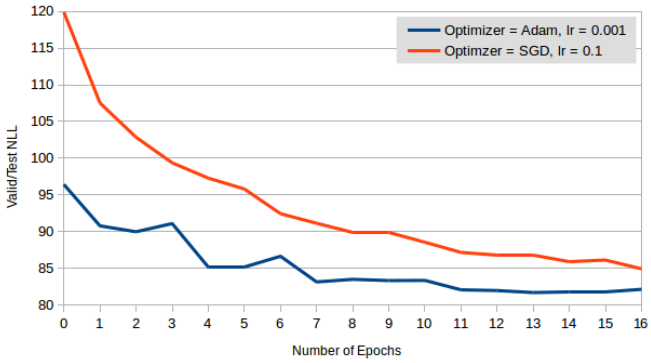


Fig. 11: Optimizer Experiments

parameter and we observed the NLL decreased by increasing twin value which proves the claim of the paper. This parameter penalizing the distance between forward and backward hidden states leading to same prediction and trying to minimize this difference. And the following is what we have discovered:

$$L_t(s) = ||g(h_t^f) - h_t^b||$$

TABLE IV: Twin Regularization Parameter Tuning

Hyper Parameter	Tested values	NLL	Selected Value
Twin	0, 0.1, 0.2, 1	81.2, 81.5, 80, 81	0.1

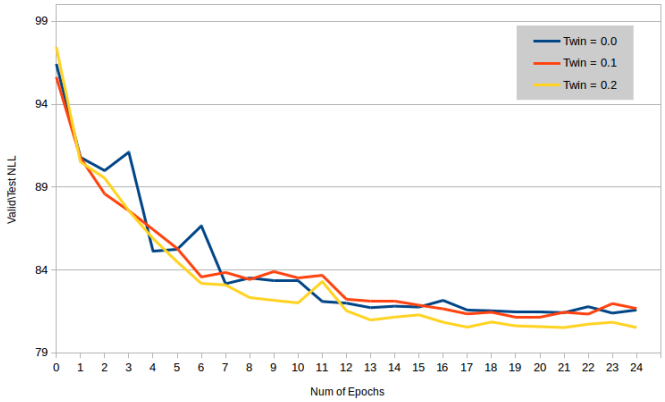


Fig. 12: Twin Regularization Term Experiments

3) Embedding Layer: To continue our experiments on LSTM with an Embedding Layer, the weights for the Embedding Layer can either be initialized with random values or more often they are initialized with third-party word embeddings pre-trained models such as word2Vec, Doc2Vec or GloVe, and these weights can optionally be fine-tuned during training. We investigated the effects of increasing or decreasing the parameters of embedding function in the training phase which result in the different permutation of inner trainable weights, i.e. the change effects the distances among the learned representation which determine the closest pixel associated with a specific pixel. Although, due to lacking of time and

computational resources we couldn't observe the effect of embedding layer, we did research about the effectiveness of increasing/decreasing embedding layer in the network.

D. Fourth Round Experiment: Second-order Hyperparameters Tuning

After choosing Adam as our optimizer and Twin equals to 0.2, we experimented second order hyper-parameters including Batch Size and Number of Hidden Nodes. we called it second order since their effect is less than the learning rate, dropout and number of layers, also they are less computationally expensive as we can get a result as good as changing the number of hidden nodes which takes a very long time in comparison with the one equals to 1024. However, even we expected to get a better result with hidden nodes equals to 1024 as the capacity of the model increases, but it wasn't the case.

TABLE V: Second-order Hyperparameters Tuning

Hyper Parameter	Tested values	NLL	Selected Value
Batch size	20, 40	80.43, 80.52	20
Hidden nodes	512, 1024	81.52, 80.78	512

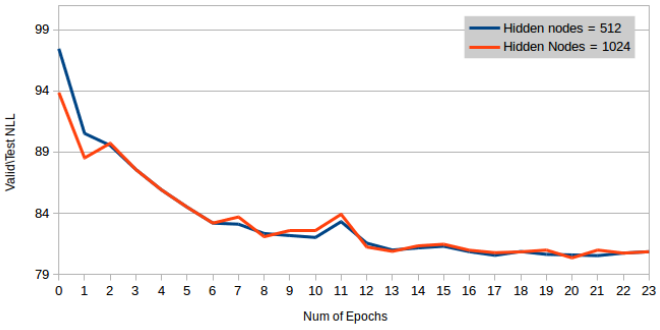


Fig. 13: Hidden Nodes Exp

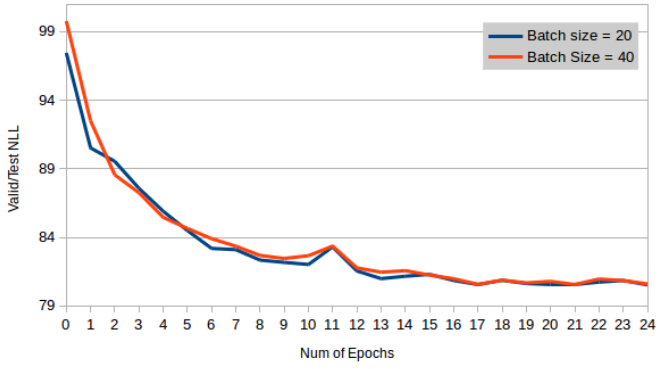


Fig. 14: batchsize Exp

V. Results

After training the model on the dataset, 80.52 nat was achieved on Sequential MNIST. This achieved by applying all of the optimal values found using architecture experimentation, hyper-parameter tuning, and data exploration. The final results can be seen in Table VI. We were able to reach to 80.52 NLL using 0.001 as learning rate, Adam optimizer, 0.05 as dropout value, twin regularization equals to 0.2 after 23 epochs. We were interested on how different hyperparameters would perform and what is the effect of changing each on the performance of the model. The following is what we have discovered: We were able to get better results after same number of epoch in models with 3 LSTM instead of 2 layers and using 1024 nodes versus 512, but these changes increased the parameters of the model and we did not have time and resources; thus, we decided to use 2 LSTM layers with 512 nodes. As we can see in the table, the effect of batch size is very low, so we kept it as default equals to 20 and also for the number of hidden nodes we observed 512 gives a slightly better result which proves the setting of twin net in the paper.

TABLE VI: Results of Experiments

Hyperparameter	Best Value	Best NLL	Improvment in NLL
Learning Rate	0.001	81.214	2.24
Dropout	0.05	81.306	0.43
Number of Layer	3	80.896	0.24
Optimizer	Adam	81.426	2.96
Twin Regularization Term	0.2	80.528	0.61
Hidden Nodes	512	80.528	0.26
Batch Size	20	80.528	0.03

VI. Discussion

One area that more experiments could be done is trying different number of layers of hidden layer and number of nodes layer, adding dropout to the input and output layer instead hidden nodes. Another approach would be to use L1 loss instead of L2.

Finally, Using the whole dataset instead of half of that, will probably enhance the results. Since we used half of the dataset in our experiment due to lacking in time and computational resources.

VII. Conclusion

In this project, we did ablation study on Twin Networks in which a backward recurrent neural network is added to the feed forward RNN, as a regularization method to force the RNN to consider long-term dependencies of the data.
In our experiments, the computed negative log likelihood over validation and test sets were relatively equal; we considered this as a sign that we do not have overfitting and these metrics are good indicators of the performance and capacity of the model. Choosing the initial learning rate played the most important role in reaching out to the best model in a feasible time; we could even have an underfitted model by choosing an inappropriate learning rates. By adding twin regularization cost to the network, we decreased NLL by 0.61 which proved that the proposed approach enhances the RNNs performance. Choosing correct optimizer also is very significant in finding

the best model. We tried both Adam and SGD with momentum. Adam was easier to tune and had a more stable behavior, but in case of SGD, we tried multiple learning rates, but we were not able to get the promising results as the Adam. Also, SGD had a more unpredictable behavior since loss and NLL of the model increased and decreased multiple times. Based on this, we can say that because of more complex intuition of the Adam, it works better than simple SGD with momentum. The unfortunate aspect of this method was doubling time and computational complexity, due to the backward RNN, but in comparison to another models in the RNN state of the art, it is relatively reasonable. In other words, conventional RNNs need refinements in order to handle sequential data better, and Twin Networks are the solution to this problem.

VIII. Acknowledgments

Authors would like to thank instructors of the Applied Machine Learning course at McGill University, including Ryan Lowe, Sarath Chandar, and Herke Van Hoof for their useful instructions and insights. Also we would like to thank developers of the PyTorch toolkit for their well-implemented platform and also Google Compute Cloud for providing us access to their computational resources.

IX. References

[1] Dmitriy Serdyuk, Rosemary Nan Ke, Alessandro Sordoni, Chris Pal, and Yoshua Bengio. Twin networks: Using the future as a regularizer. arXiv preprint arXiv:1708.06742, 2017.

[2] Alex M Lamb, Anirudh Goyal, Ying Zhang, Saizheng Zhang, Aaron C Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks. In NIPS, 2016.

[3] Zhongliang Li and Raymond Kulhanek and Shaojun Wang and Yunxin Zhao and Shuang Wu. Slim Embedding Layers for Recurrent Neural Language Models. In NIPS, CoRR: abs/1711.09873, 2017.

[4] Stephen Merity, Nitish Shirish Keskar, Richard Socher. An Analysis of Neural Language Modeling at Multiple Scales. arXiv:1803.08240v1, 2018.

[5] Goldberg, Yoav, and Omer Levy. "word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method." arXiv preprint arXiv:1402.3722 (2014).

[6] Mao, J., Xu, W., Yang, Y., Wang, J., Yuille, A. L. (2014). Explain images with multimodal recurrent neural networks. arXiv preprint arXiv:1410.1090.

X. Appendix(optional)

Here we included additional results, more detail of the methods and the effect of each hyperparameter on the performance of the model. We also attached the pictures in our submission.

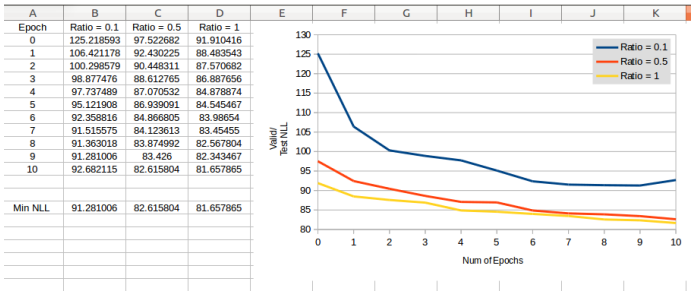


Fig. 15: Data Ratio Experiment

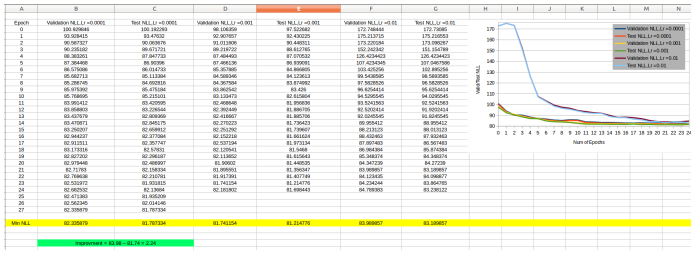


Fig. 16: Learning Rate Experiments

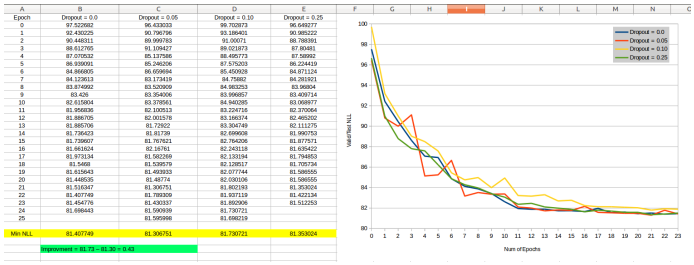


Fig. 17: Dropout Experiment

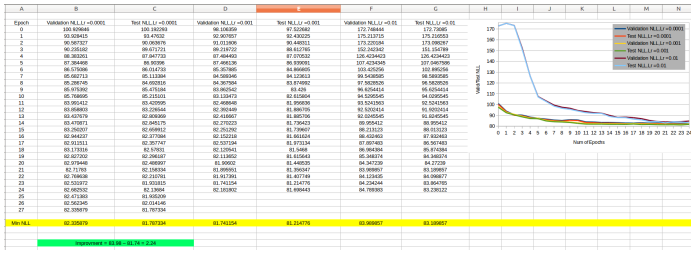


Fig. 18: Number of Layer Experiment

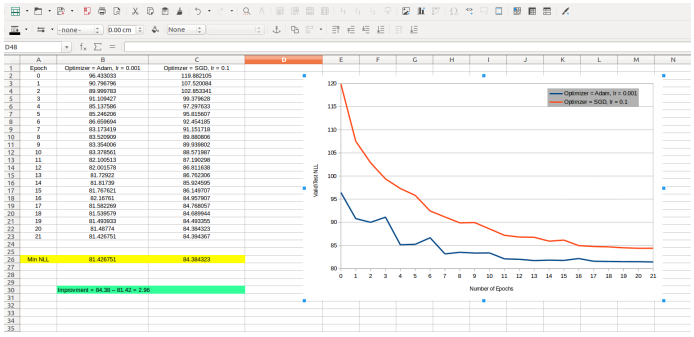
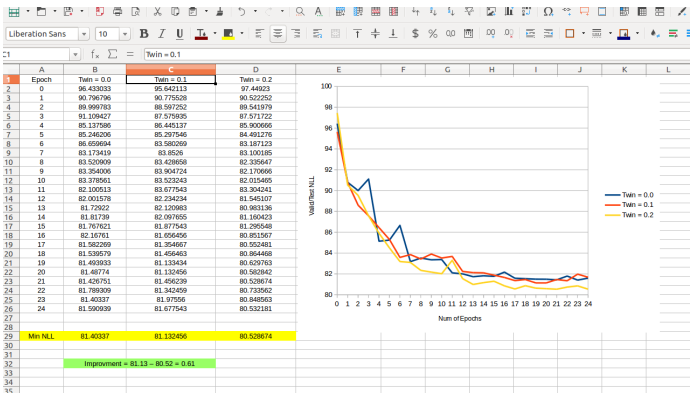
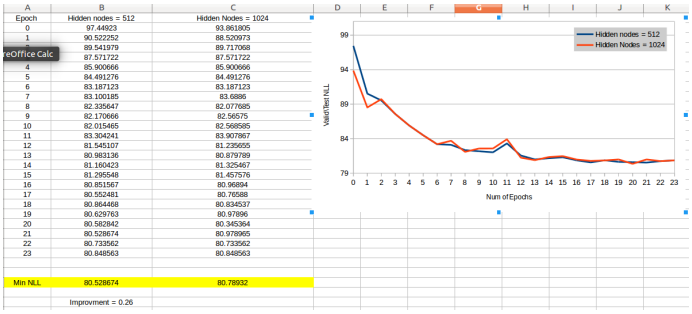


Fig. 19: Optimizer Experiment



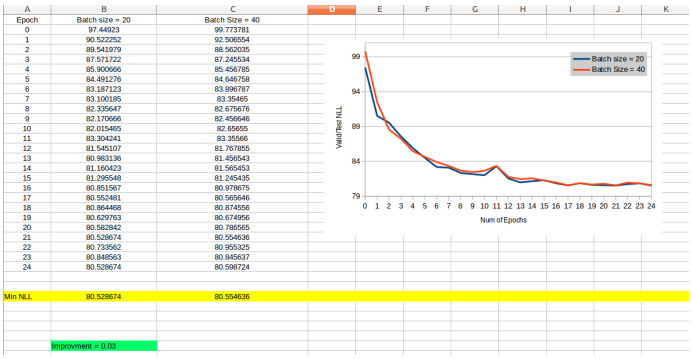
Epoch	Twin = 0.0	Twin = 0.1	Twin = 0.2
0	96.433033	95.642113	97.44923
1	90.796796	90.775526	90.522252
2	89.999783	88.597252	89.541979
3	91.109427	87.575935	87.571722
4	85.137585	86.445137	85.900666
5	85.246206	85.297546	84.491276
6	86.659694	83.580269	83.187123
7	83.173419	83.8526	83.100185
8	83.520909	83.428658	82.335647
9	83.354006	83.904724	82.170666
10	83.378561	83.523243	82.015465
11	82.105513	83.677543	83.304241
12	82.001578	82.234234	81.545107
13	81.72922	82.120983	80.983136
14	81.81739	82.097655	81.160423
15	81.767621	81.877543	81.295548
16	82.16761	81.656456	80.851567
17	81.582269	81.354667	80.552481
18	81.539579	81.456463	80.864468
19	81.493933	81.133434	80.629763
20	81.48774	81.132456	80.582842
21	81.426751	81.456239	80.528674
22	81.789309	81.342459	80.733562
23	81.40337	81.97556	80.848563
24	81.590939	81.677543	80.532181
Min NLL	81.40337	81.132456	80.528674
Improvment = 81.13 – 80.52 = 0.61			

Fig. 20: Twin Experiment



Epoch	Hidden nodes = 512	Hidden Nodes = 1024
0	97.44923	93.861805
1	90.522252	88.520973
2	89.541979	89.717068
3	87.571722	87.571722
4	85.900666	85.900666
5	84.491276	84.491276
6	83.187123	83.187123
7	83.100185	83.6886
8	82.335647	82.077685
9	82.170666	82.56575
10	82.015465	82.568585
11	83.304241	83.907867
12	81.545107	81.235655
13	80.983136	80.879789
14	81.160423	81.325467
15	81.295548	81.457576
16	80.851567	80.96894
17	80.552481	80.76588
18	80.864468	80.834537
19	80.629763	80.97896
20	80.582842	80.345364
21	80.528674	80.978965
22	80.733562	80.733562
23	80.848563	80.848563
Min NLL	80.528674	80.78932
Improvment = 0.26		

Fig. 21: Hidden Nodes Experiment



Epoch	Batch size = 20	Batch Size = 40
0	97.44923	99.773781
1	90.522252	92.506554
2	89.541979	88.562035
3	87.571722	87.245534
4	85.900666	85.456785
5	84.491276	84.646758
6	83.187123	83.896787
7	83.100185	83.35465
8	82.335647	82.675676
9	82.170666	82.456646
10	82.015465	82.65655
11	83.304241	83.35566
12	81.545107	81.767855
13	80.983136	81.456543
14	81.160423	81.565453
15	81.295548	81.245435
16	80.851567	80.978675
17	80.552481	80.565646
18	80.864468	80.874556
19	80.629763	80.674956
20	80.582842	80.786565
21	80.528674	80.554636
22	80.733562	80.955325
23	80.848563	80.845637
24	80.528674	80.598724
Min NLL	80.528674	80.554636
Improvment = 0.03		

Fig. 22: Batch Size Experiment