EXPLORATION OF PROBABILISTIC NEURAL NETWORKS

TECHNICAL REPORT

Aditya Kompella
Department of Computer Science
Cornell University

May 19, 2021

ABSTRACT

Utilizing the torch.distributions package I leverage distributions in this experiment to use in neural network models. This is possible due to something called the reparameterization trick which allows you to differentiate the sampling operation in these PDFs with respect to the parameters. This paper focuses on training "probabilistic neural networks" and regular deterministic to see how network performance differs between them on image classification tasks. I created probabilistic fully connected layers using random sampling through various probability distribution and trained models using these layers on the MNIST dataset and observed the results. I develop a technique to try to improve performance of these models called Multi Forward Pass Inference(MFPI) and tested this to see how this improvement affected the performance. I also looked into an improvement to the classic Cross Entropy Loss function in these models called Standard Deviation Loss that can mitigate the negative affects of over fitting to the data. I compiled all the results of training networks with these layers and compared the results of accuracy and Validation Loss to see if there is any merit in using these Probabilistic Neural Networks for image classification tasks.

1 Data

The data I chose was the MNIST Data set which is a large database of handwritten digits from 0-9. I chose this data set as it is a classic image classification data set that would be a good data set to test out the effectiveness of Probabilistic Neural Networks while being small in resolution making the training time faster so that I could iterate over my experiments much faster than large image resolution data sets. These images are prepossessed to be a black and white grey scale image. The images are of size 28*28 pixels. I unrolled each of the 28*28 images into a 784-dimensional vector and batched them into a batch of 1028 vectors so I could train them using my Probabilistic Fully Connected Networks. The goal of the network is to predict what digit the given image is.

2 Methods

2.1 Background: Reparameterization trick

There are many different probability distributions that can be sampled from that are offered by the Pytorch library. They can be differentiated through and be used in Deep Learning and neural networks due to the reparameterization trick[1]. For example, the Normal Distribution has two parameters, μ and σ. To be able to differentiate sampling a distribution through both of those parameters we have to use this trick.
For example say:
$Output = \mathcal{N}(\mu, \sigma)$
then this can be written as:
$Output = \mu * \mathcal{N}(0, 1) + \sigma$
Now that both parameters of the distribution μ and σ have been taken out of the sampling operation of the Normal Distribution we can treat sampling from the Standard Normal Distribution as just a constant and compute the gradient:

$\frac{\partial Output}{\partial \mu} = \mathcal{N}(0,1)$
$\frac{\partial Output}{\partial \sigma} = \vec{1}$
This same technique can be used to differentiate through sampling many different distributions by "separating" the parameters of the distribution from the direct sampling process. Now that the gradient can be computed, this operation of sampling through probability distributions can be utilized in neural networks with backpropogation algorithm to use in gradient descent.

2.2 Probabilistic Fully Connected Layers

Probabilistic Fully Connected Layers can be constructed by using regular fully connected layers to project the input into the parameters and then feeding those parameters as input to the distribution. Ex: Normal Linear Layer
Pseudo Code:
Normal Linear Layer:
$\mu = Linear(input)$
$\sigma = Linear(input)$
$output = Normal(\mu, \sigma).sample()$
The use of these type of layers is common in Bayesian Deep Learning[2] where distributions are used instead in neural networks instead of deterministic layers. However, the method of training is Bayesian Deep Networks is slightly different and the weights are sampled from a PDF and not the outputs themselves like in this experiment. Our Networks made up of these layers is also trained exclusively end to end using Gradient descent.
The different Distributions I used to construct the Layers are:
1. Normal
2. Laplace
3. Beta
4. Continuous Bernoulli
5. Dirichlet
6. Exponential
7. Gamma

I chose these distributions as these are most of the distributions offered by pytorch that support differentiating through them to be used in Neural Networks.

2.3 Multi-Pass Forward Inference(MPFI)

I propose a method called multi-forward-pass Inference(MFPI). In Normal Deterministic Neural Networks, a common technique to improve performance is to have multiple different models trained on the same data make predictions on the data and then to average their logits.. This can improve performance because even if some of the models predict a certain image incorrectly, if the majority of the models agree on the correct label, since we are averaging the logits, we will predict the correct class. This cannot be done with a single deterministic neural network because every time you do a forward pass through a network you will always get the same result. This can be done with probabilistic neural networks however as every time you do a forward pass through the network, since there is randomness involved, our outputs will be different. That is why at inference time, you can do multiple forward passes and average their logits before making a final prediction.

Inputs: Model, Images, Num Forward Passes
Result: Inference on Image
logits = zeroes();
for *iteration in num passes* **do**
  logits += model(Images);
end
logits = logits/(num passes);
return logits;

Algorithm 1: Multi Pass Forward Inference(MPFI) Algorithm Psuedo Code

2.4 Standard Deviation Loss

In these probabilistic Neural Networks, whenever we have a distribution that has a mean and standard deviation parameter (ex:Laplace, Normal), the standard deviation vectors that the network tends to learn tend to a magnitude of zero. This is because for the lowest loss, the network wants to learn a standard deviation of zero so that is 100% sure of

its output of every layer. This is undesirable though as at that point the probabilistic neural network tends to become just a regular deterministic neural network. Therefore, I also tested adding a term to the loss function I call Standard Deviation Loss which averaged the magnitude of the standard deviation vectors at every layer. Subtracting this term from the original loss function causes the network to learn larger standard deviation vectors during training. This now added a new parameter lambda to weight this loss term which during experiments showed best results when it equaled 0.1.
New Loss Equation: L(y, f(θ)) $= CrossEntropyLoss(y, f(\theta)) - \lambda * SD_{Loss}(f(\theta))$
For example if the neural network($f(\theta)$) has 5 probabilistic layers with each layers standard deviation vectors being called $\text{sd}_{1..5}$:
$\text{SD}_{Loss}(f(\theta)) = E[sd_1.sum() + sd_2.sum() + sd_3.sum() + sd_4.sum() + sd_5.sum()]$
I tried this loss addition with both the Laplace and Normal Distributions and tracked the training and final performance for each.

2.5 Network Architecture

For the network architecture, I chose a simple fully connected network that has a hidden dimension of 1,000. It has 5 fully connected/ probabilistic layers and an activation function in between each one. The activation function I chose is Mish which is a state of the art activation function currently[3]. This allowed me to make use of the 784 dimension MNIST image vectors that I trained on. Probabilistic Neural Networks and the way I implemented them work better with fully connected layers than Convolutional layers which is why I picked this model. I picked a small network because the MNIST dataset is an easy problem for a neural network to solve. The last fully connected layer projects from a hidden dimension of a 1,000 to 10 because there are 10 classes in the MNIST data set. I had a regular network with only fully connected layers, a 100% probabilistic fully connected model, 60% probabilistic fully connected model, 20% probabilistic fully connected with different ratios of regular and probabilistic layers respectively. I decided to make different amounts of the network probabilistic to see how this would affect performance. I also used the Cross Entropy Loss that already has a soft-max operation included in it because it was a multi class classification problem.

Table 1: Fully Connected Neural Network Architecture

Layer Name	Output Size	Layer Type
Input	(batch_size,784)	Input
fc_1	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_2	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_3	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_4	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_5	(batch_size, 10)	Fully Connected

Table 2: 100 % Probabilistic Fully Connected Neural Network Architecture

Layer Name	Output Size	Layer Type
Input	(batch_size,784)	Input
Probabilistic_fc_1	(batch_size, 1000)	Probabilistic Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_2	(batch_size, 1000)	Probabilistic Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_3	(batch_size, 1000)	Probabilistic Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_4	(batch_size, 1000)	Probabilistic Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_5	(batch_size, 10)	Probabilistic Fully Connected

Table 3: 60 % Probabilistic Fully Connected Neural Network Architecture

Layer Name	Output Size	Layer Type
Input	(batch_size,784)	Input
fc_1	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_2	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_3	(batch_size, 1000)	Probabilistic Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_4	(batch_size, 1000)	Probabilistic Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_5	(batch_size, 10)	Probabilistic Fully Connected

Table 4: 20 % Probabilistic Fully Connected Neural Network Architecture

Layer Name	Output Size	Layer Type
Input	(batch_size,784)	Input
fc_1	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_2	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_3	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
fc_4	(batch_size, 1000)	Fully Connected
Mish	(batch_size, 1000)	Activation
Probabilistic_fc_5	(batch_size, 10)	Probabilistic Fully Connected

3 Results

3.1 Probabilistic Models MNIST Performance

All models trained with Adam Optimizer with lr = 0.001
For each of the distributions I trained them as long as it took to train to convergence and until the valid loss and accuracy stabilized. Since training these probabilistic models take longer than their deterministic counterparts, I had to train for different epochs for each distribution.

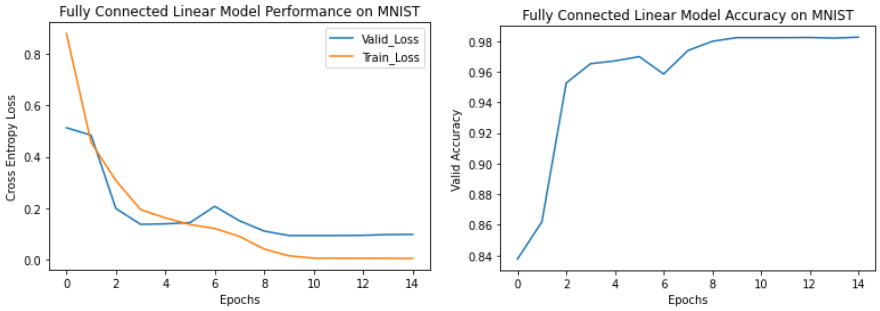

Figure 1: Regular Fully Connected Model Results (Final Accuracy: 0.9826)

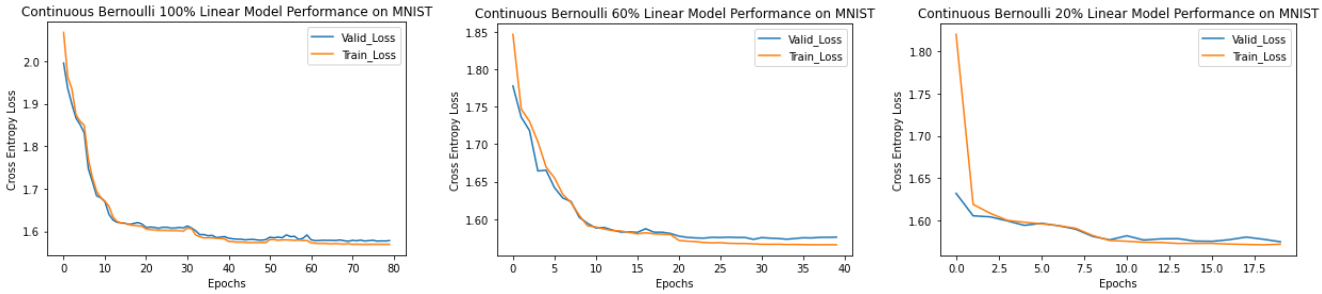

Figure 2: Continuous Bernoulli Distribution Train/Validation Loss

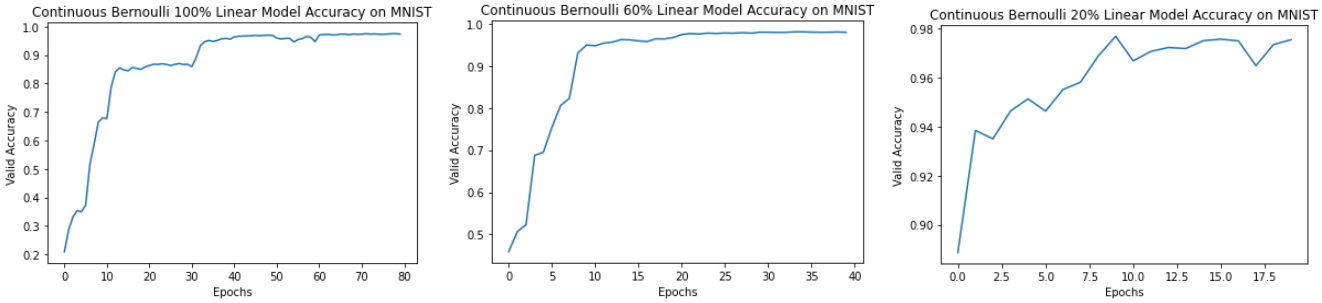

Figure 3: Continuous Bernoulli Distribution Accuracy , (Final Accuracy: (0.9731,0.9797,0.9754)

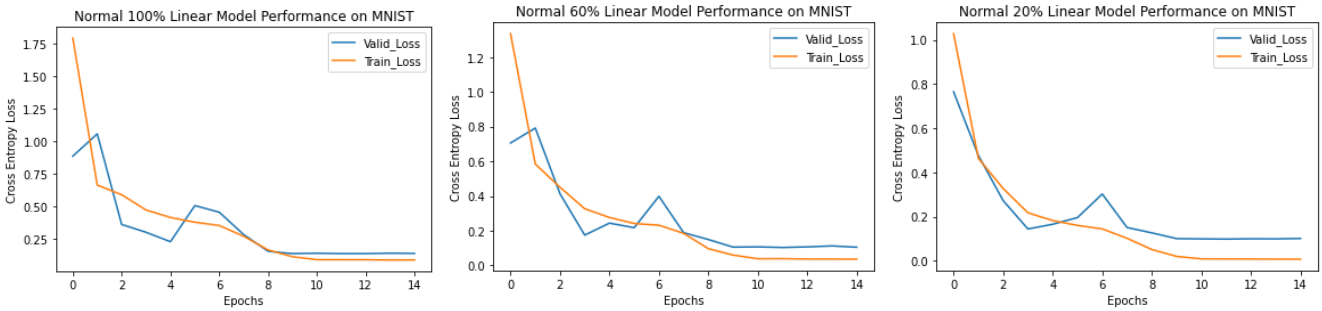

Figure 4: Normal Distribution Train/Validation Loss

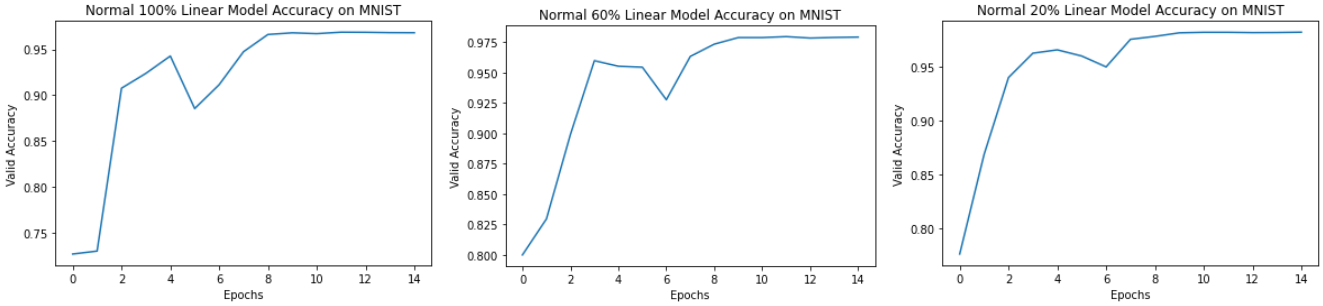

Figure 5: Normal Distribution Accuracy, (Final Accuracy:(0.9681,0.9793,0.9818)

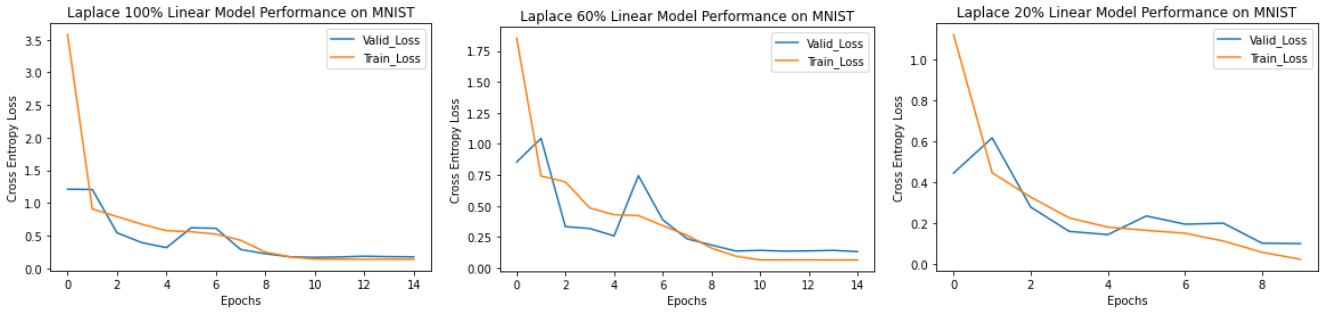

Figure 6: Laplace Distribution Train/Validation Loss

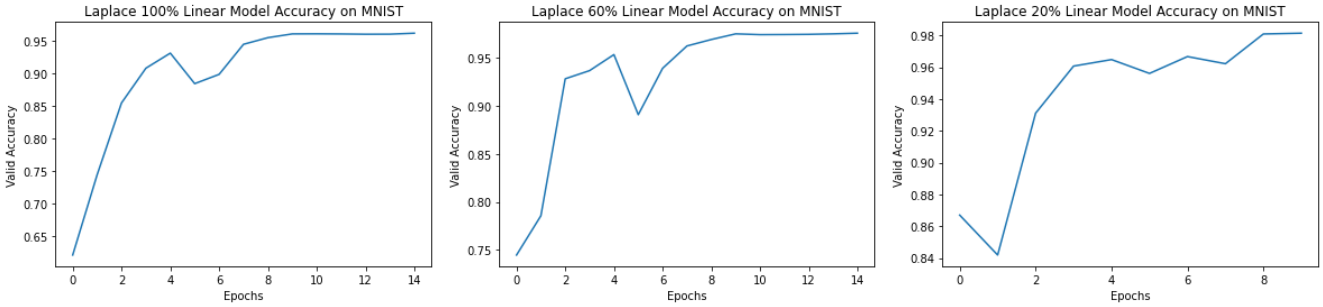

Figure 7: Laplace Distribution Accuracy, (Final Accuracy:(0.9622,0.9758,0.9814)

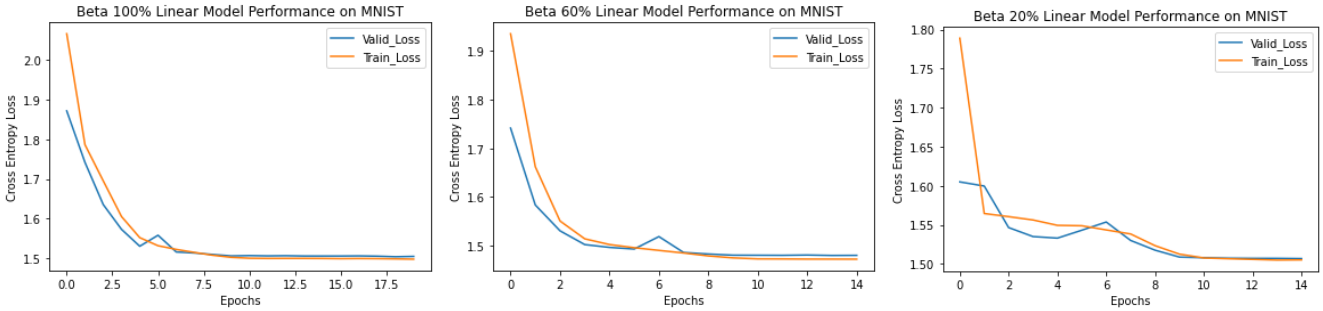

Figure 8: Beta Distribution Train/Validation Loss

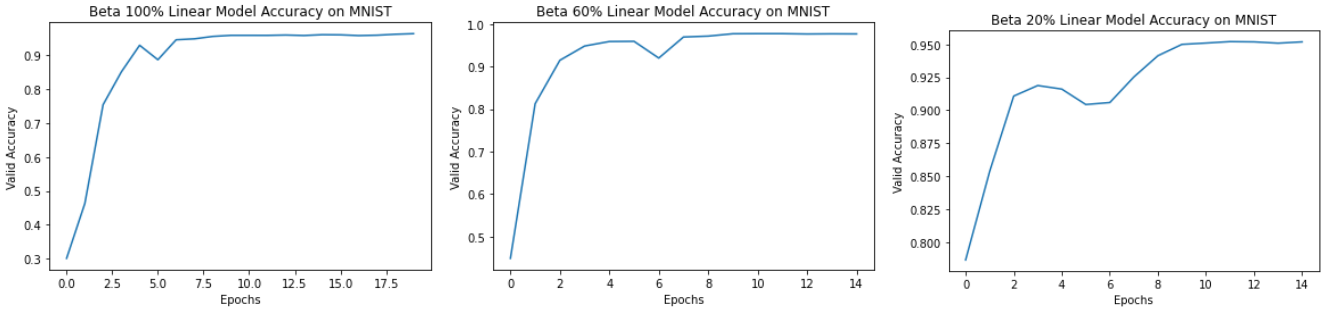

Figure 9: Beta Distribution Accuracy, (Final Accuracy:(0.963,0.9768,0.9519)

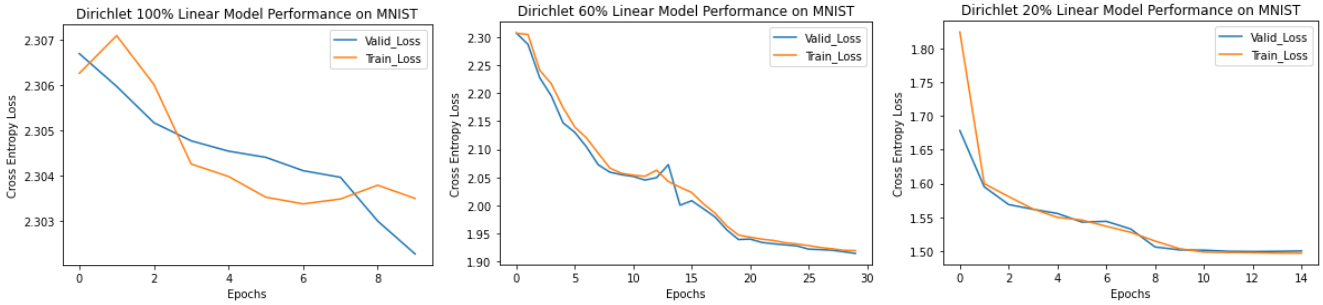

Figure 10: Dirichlet Distribution Train/Validation Loss

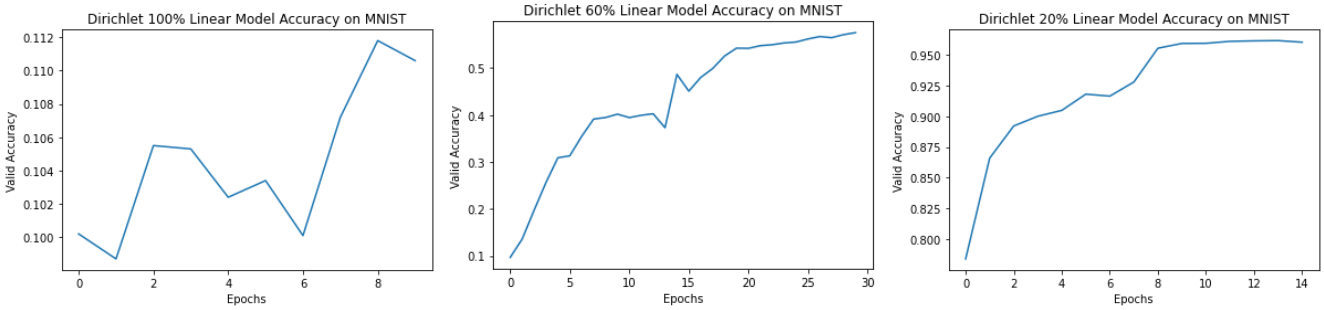

Figure 11: Dirichlet Distribution Accuracy, (Final Accuracy:(0.1106,0.5757,0.9603)

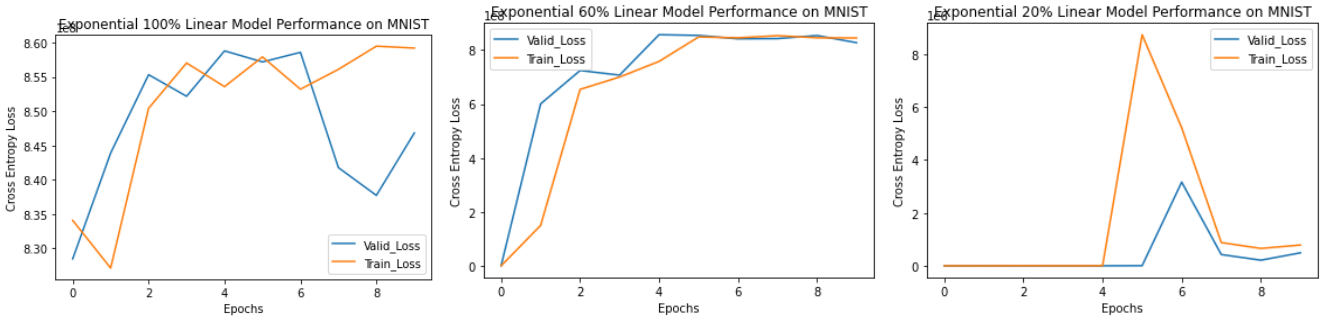

Figure 12: Exponential Distribution Train/Validation Loss

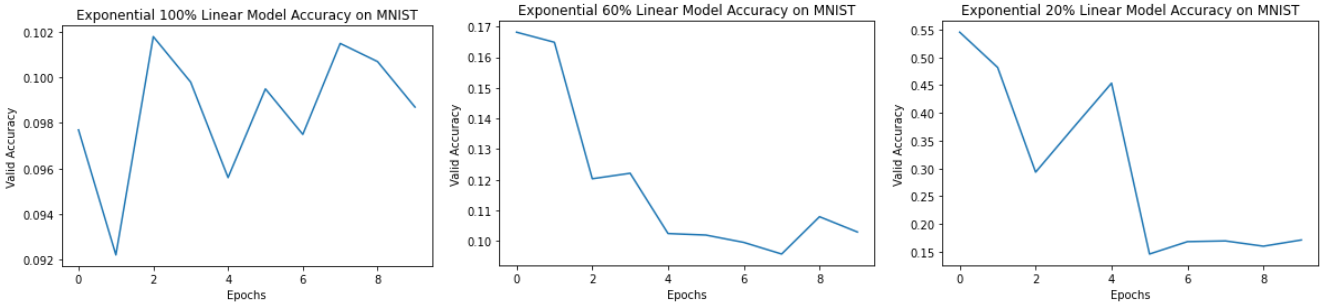

Figure 13: Exponential Distribution Accuracy, (Final Accuracy:(0.0987,0.1029,0.1713)

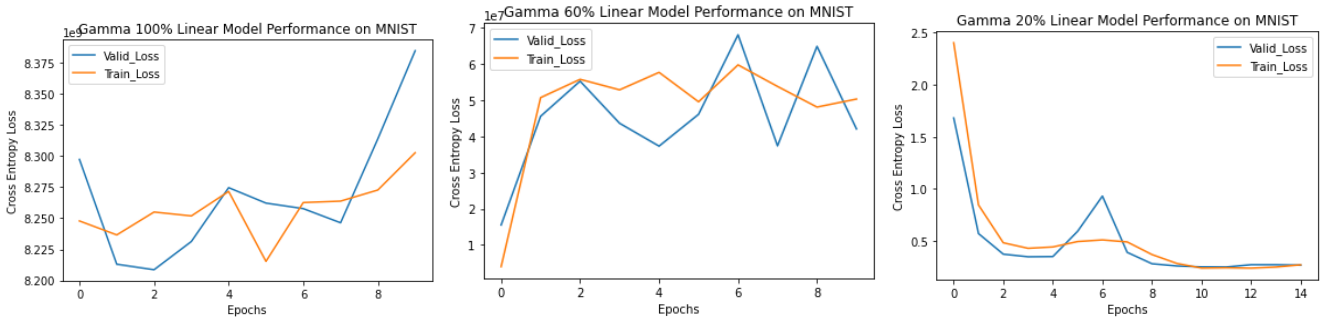

Figure 14: Gamma Distribution Train/Validation Loss

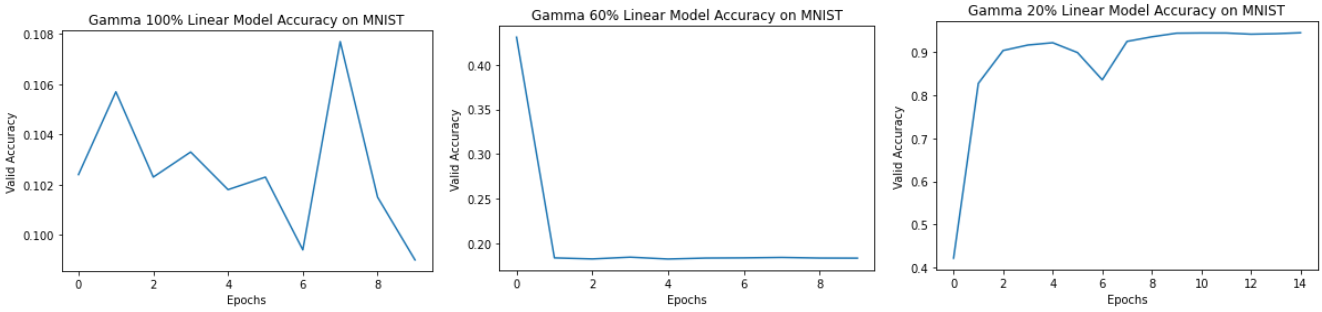

Figure 15: Gamma Distribution Accuracy, (Final Accuracy:(0.099,0.1831,0.9442)

We see that the lower the probabilistic percentage of the network, the more the network over fits on the data. This shows the regularization affect of probabilistic layers. We see that the models with the best performance are actually the 60% probabilistic model that is a blend of the performance that deterministic models give along with the regularization affect of probabilistic models. Also the training for models without standard deviation parameters train much more smoothly with less oscillations in loss during training than the ones that do. We see that certain distributions,(Usually ones that have a hard time being symmetric) do not converge and have very bad accuracy and loss like the Dirichlet, Gamma or Exponential Distribution which didn't perform well when trained with fully probabilistic network. It seems that the best models arise from training the Normal Distribution and the Continuous Bernoulli with a 60% probabilistic architecture.

3.2 Multi-Pass Forward Inference(MPFI) for all models

Results were obtained by taking each of the Probabilistic models and running 10 forward pass Inference on them and obtaining accuracy.

Network	Regular Inference Accuracy	10 pass MPFI Accuracy
Fully Connected	**0.9826**	0.9826
Continuous Bernoulli 100% FC	0.9731	0.9772
Continuous Bernoulli 60% FC	0.9797	**0.9846**
Continuous Bernoulli 20% FC	0.9754	0.978
Normal 100% FC	0.9681	0.974
Normal 60% FC	0.9793	0.978
Normal 20% FC	0.9818	0.9784
Laplace 100% FC	0.9622	0.9683
Laplace 60% FC	0.9758	0.9698
Laplace 20% FC	0.9814	0.9759
Beta 100% FC	0.963	0.967
Beta 60% FC	0.9768	0.9778
Beta 20% FC	0.9519	0.954
Dirichlet 100% FC	0.1106	0.1150
Dirichlet 60% FC	0.5757	0.5935
Dirichlet 20% FC	0.9603	0.9621
Exponential 100% FC	0.0987	0.112
Exponential 60% FC	0.1029	0.178
Exponential 20% FC	0.1713	0.184
Gamma 100% FC	0.099	0.116
Gamma 60% FC	0.1831	0.196
Gamma 20% FC	0.9442	0.9466

We see that Multi Pass Forward Inference Benefits models that don't have standard deviation parameters much more than the ones that do because the network will learn to zero out the standard deviation vectors during training making MPFI not as effective. It seems that this techniques causes the Continuous Bernoulli Distribution 60% network have better accuracy than the Regular Fully Connected Network. Although the Regular inference accuracy for the Regular Fully Connected Network is the Highest, the probabilistic networks can obtain higher accuracy with this technique.

3.3 Standard Deviation Loss Results

All results using SD_{Loss} with $\lambda = 0.1$
Tested the loss with Normal Distribution 100% network and Laplace Distribution 100% network.

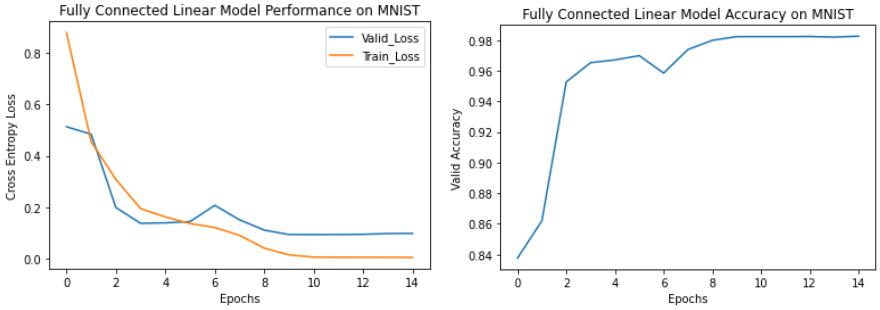

Figure 16: Regular Fully Connected Model Results (Final Accuracy: 0.9826)

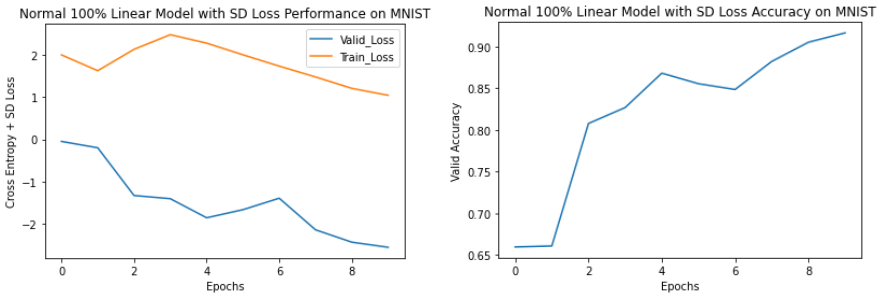

Figure 17: Normal 100% Linear Model with SD Loss Results (Final Accuracy: 0.9196)

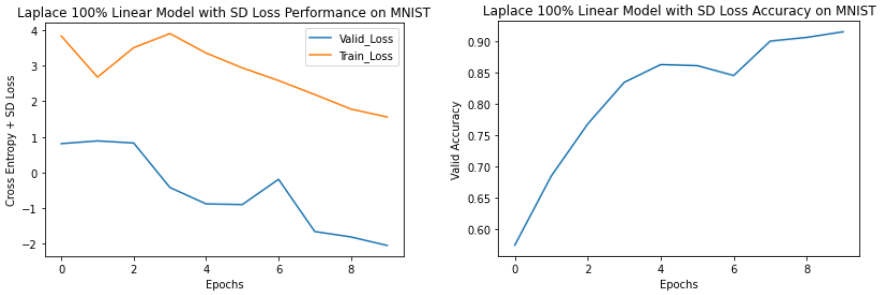

Figure 18: Laplace 100% Linear Model with SD Loss Results (Final Accuracy: 0.9151)

Although the accuracy of the models with SD Loss don't perform as well as the regular fully connected network, we can see that in both models with the loss, we see under fitting and that with the regular model we see over fitting. This shows how perhaps probabilistic models with this loss can train longer to obtain higher accuracy and learn to generalize better.

4 Conclusion

It seems that for some Distributions, probabilistic models can outperform their standard counterparts by a slight margin. In the realm of Deep Learning where even 0.5% accuracy increase is a big thing, these are quite promising results. The greatest performance seems to be obtained when we do multi-forward inference on a model that is 40% standard and 60% probabilistic. It seems that this balance in regular probabilistic layers lends best performance. We also see how Standard Deviation Loss can really mitigate the affects of over fitting and it seems to be a great regularization technique. We can see that the randomness baked into the model prevents too much over fitting and allows us to increase performance by averaging logits from multiple forward passes. This shows the effectiveness of this as a regularization method. In scenarios where models tend to over fit on the data, for example when there is very little data, this could be used as a way to increase validation accuracy. It seems that training time to reach convergence is much longer the more "probabilistic" your model is though. The results here show that there is promise in using Probabilistic Networks for their regularization benefits and for better performance in Deep Learning in the future.

References

[1] Diederik P. Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick, 2015.

[2] Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. Hands-on bayesian neural networks – a tutorial for deep learning users, 2020.

[3] Diganta Misra. Mish: A self regularized non-monotonic activation function, 2020.