RnD Project Report

Chidaksh Ravuru

January 2023

Contents

1 Image Classification using Convolutional Neural Networks 3

2 The Six Jargons 4
- 2.1 Task . . . 4
- 2.2 Dataset . . . 4
  - 2.2.1 CIFAR-10 . . . 4
  - 2.2.2 IMAGENET . . . 4
- 2.3 Models and Device Used . . . 4
- 2.4 LENET Architectures . . . 4
  - 2.4.1 LeNet . . . 4
  - 2.4.2 LeNetBN Architecture . . . 5
  - 2.4.3 LeNetDp Architecture . . . 7
  - 2.4.4 LeNet Architecture with L2-weight decay . . . 8
  - 2.4.5 LeNetBNDp Architecture . . . 9
  - 2.4.6 SuperLeNet Architecture . . . 11
- 2.5 RESNET Architectures . . . 13
  - 2.5.1 ResNet . . . 13
- 2.6 Loss function . . . 15
  - 2.6.1 Cross Entropy Loss . . . 15
- 2.7 Learning Algorithm . . . 16
  - 2.7.1 Gradient Descent . . . 16
  - 2.7.2 SPSA . . . 16
- 2.8 Optimizers on top of Learning Algorithm . . . 17
  - 2.8.1 Adam . . . 17
- 2.9 Evaluation . . . 18

3 Optimizations for improving performance and reducing computational cost while Training the Model 18
- 3.1 Intialization of Weights and biases of a Neural Network . . . 18
  - 3.1.1 Xavier Initialization . . . 19
  - 3.1.2 Kaiming/He Initialization . . . 19

4 Experiments and Results **20**
4.1 LeNet Architectures on CIFAR-10 dataset 20
4.2 LeNet Architectures on TinyImageNet 24
4.3 ResNet Architectures on CIFAR-10 28
4.4 ResNet on TinyImageNet Dataset 30
4.5 Importance of Proper Intialization of weights and biases of a NeuralNet 32
4.5.1 Effect of Weight Initialization on CIFAR-10 33
4.5.2 Effect of Weight Initialization on TinyImageNet 37
4.6 Effect of Batch Size 42

5 Language Modelling using Recurrent Neural Networks **44**

6 The Six Jargons **44**
6.1 Tasks 44
6.2 Models and Device Used 45
6.3 Simple RNN Architecture 45
6.4 LSTM Architecture 46
6.5 GRU Architecture 47
6.6 Learning Algorithm 48
6.7 Optimizers on top of Learning Algorithm 48
6.8 Task1: Charecter Level Language Model 48
6.8.1 Dataset 48
6.8.2 Loss Function 48
6.8.3 Model Architecture: 48
6.8.4 Plots: 49
6.8.5 Name Genrated Before Training and After Training: . . . 49
6.9 Task2: Word Level Language Model 50
6.9.1 Dataset 50
6.9.2 Loss Function 50
6.10 Model Architectures: 50
6.10.1 Plots: 51
6.10.2 Comparing Inferences: 52
6.10.3 Important Things Observed 52
6.11 Task3: Classifying movie reviews into Positive/negative 53
6.11.1 Dataset: 53
6.11.2 Model Architecture: 54
6.11.3 Loss Function 54
6.11.4 Evaluation 54
6.11.5 Plots and Tables: 55
6.12 Task4: Language Translation from French to English 56
6.12.1 Dataset 56
6.12.2 Loss Function 56
6.13 Model Architecture: 56
6.13.1 Model Outputs vs Actual Outputs 57
6.14 Plots: 58

1 Image Classification using Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are widely used in computer vision applications. CNNs are designed to handle inputs with a grid-like topology, such as images, by exploiting two key properties of image data: sparse connectivity and weight sharing.

Sparse connectivity refers to the fact that each neuron in a layer is connected to only a small subset of the neurons in the previous layer. This is achieved through convolutional filters, which are small matrices applied to the input's local regions. By using a small filter size and spreading it across the entire input, we can achieve sparse connectivity without losing the spatial structure of the information. Let's compare it with Fully Connected Feed Forward Neural Networks. Each neuron in the current network is fully connected to all other neurons in the previous layer, which increases the number of parameters of your Neural Networks.

Weight sharing refers to the same set of weights used for each location in the input. This is based on the assumption that the features that detect a particular object or pattern in one part of the image are also helpful in seeing that object or practice in other regions. This dramatically reduces the number of parameters in the network, making it more efficient to train and less prone to overfitting.

Sparse connectivity and **weight sharing** make CNNs powerful in image recognition tasks. By exploiting the spatial structure of the input data and sharing weights across the entire input, CNNs can learn features that are robust to changes in the information, such as changes in position, scale, or orientation. This makes them highly effective in image classification, object detection, and segmentation tasks.

In addition to sparse connectivity and weight sharing, CNNs also typically use pooling layers, which are used to downsample the input and reduce the spatial dimensionality of the data. This helps to make the network more computationally efficient while also making it more robust to variations in the information.

Overall, CNNs are a highly effective tool for image recognition tasks primarily due to their ability to exploit the spatial structure of the input data through sparse connectivity and weight sharing.

2 The Six Jargons

2.1 Task

The main task is training DL models for **ImageClassification** from scratch.

2.2 Dataset

2.2.1 CIFAR-10

CIFAR-10 has a total of 60000 images. I split the dataset into train dataset and test dataset. We later split training dataset into training and validation dataset. Final Datasets used to make dataloaders are as follows,

Training Data: 45000 Images
Validation Data: 5000 Images
Testing Data: 10000 Images

2.2.2 IMAGENET

The full ImageNet dataset has about 14 million images and 22,000 categories but it is very noisy, imbalanced, and has a number of other issues.

So due to the above issues, I have used Tiny ImageNet Dataset [1] which has 200 classes and 500 training images of each class. Validation data consists of 50 images of each class.

Training Data: 100000 Images
Validation Data: 10000 Images

2.3 Models and Device Used

I tried different Model Architectures, which can be divided into two classes LeNet Architectures and ResNet Architectures which are inspired from the actual LeNet[1] and ResNet Models. The later sections contain detail description of the Model Architectures. All the experiments were ran on **NVIDIA RTX A6000 GPU** with a GPU Memory of size **47.99 GB**.

2.4 LENET Architectures

2.4.1 LeNet

I took the basic LeNet Architecture [2] which consists of different convolutional layers with tanh as the activation. The architecture of LeNet along with Number of Parameters is attached in the following Image.

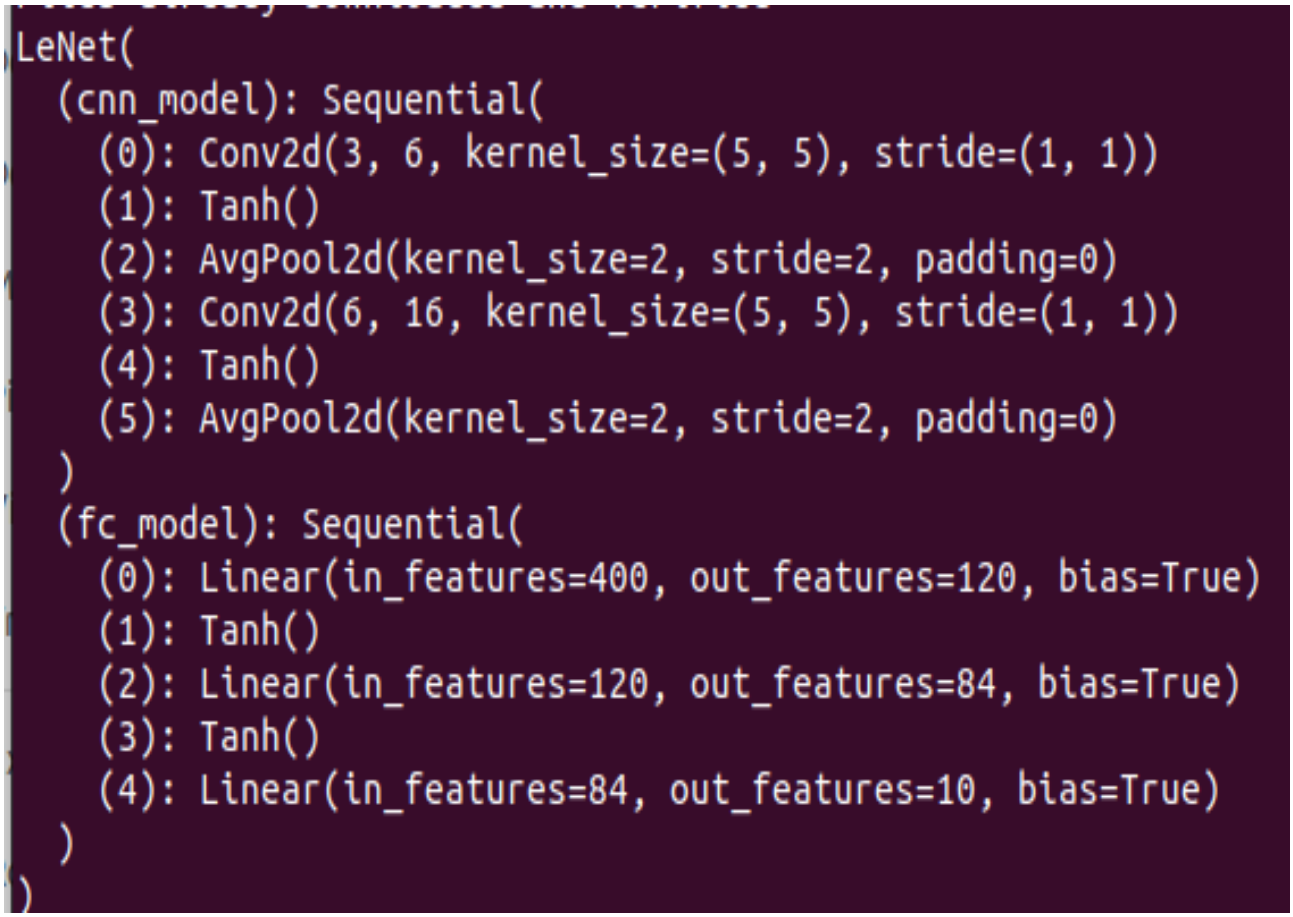


Figure 1: LeNet Model Summary

LeNet was one of early proposed architectures of CNN which has minimal Conv2D Layers with tanh activation and Max Pooling layers. So to study the effect of each technique like Batch Normalization, DropOut, Regularization, Optimizer, Back Propagation Algorithm, etc I chose LeNet architecture as it is itself doesn't provide much features in the architecture, so we can see the effect of each of these techniques clearly.

2.4.2 LeNetBN Architecture

In LeNetBN architecture, I added Batch Normalization units after each layer. Normalizing the inputs is crucial in ensuring learning happens smoothly in the neural network. If my inputs are not standardized, my weights will also be improper, as some weights would be huge and some would be small, resulting in my learning algorithm oscillating to and fro before reaching minima.Normalization helps us in smoothening the loss landscape so that my gradients converge to

zero by reaching the point of minima. My gradients also would be biased towards the dimension with larger weights, so my smaller weights will need to be appropriately updated. So normalizing inputs in most cases is profitable. Here we can treat the output of the current layer as the input of the next layer, so we add a normalization layer before every layer of the CNN to normalize its inputs. Below Image shows the Layers of the LeNetBN architecture using BatchNorm layers.

```
LeNetBN(
  (cnn_model): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): Tanh()
    (2): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (4): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (5): Tanh()
    (6): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (fc_model): Sequential(
    (0): Linear(in_features=400, out_features=120, bias=True)
    (1): Tanh()
    (2): BatchNorm1d(120, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Linear(in_features=120, out_features=84, bias=True)
    (4): Tanh()
    (5): BatchNorm1d(84, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): Linear(in_features=84, out_features=10, bias=True)
  )
)
```

Figure 2: LeNetBN Model Summary

2.4.3 LeNetDp Architecture

Dropouts give us a way of creating an ensemble of neural network architectures without affecting my complexity much. Dropout uses two tricks to do this, which are the following:

- **Sampling Neural Networks**: Sample a different Neural Network while Training.
- **Weight Sharing**: All the neural nets share the weights, but the weights update happens to those neurons active in the current neural network after applying dropouts.

With the help of these two tricks, we can create an ensemble of neural networks, ensuring that weights are properly updated. Dropouts are one of the common regularization technique to use in the case of Neural Nets, as it forces the model to learn with fewer parameters.

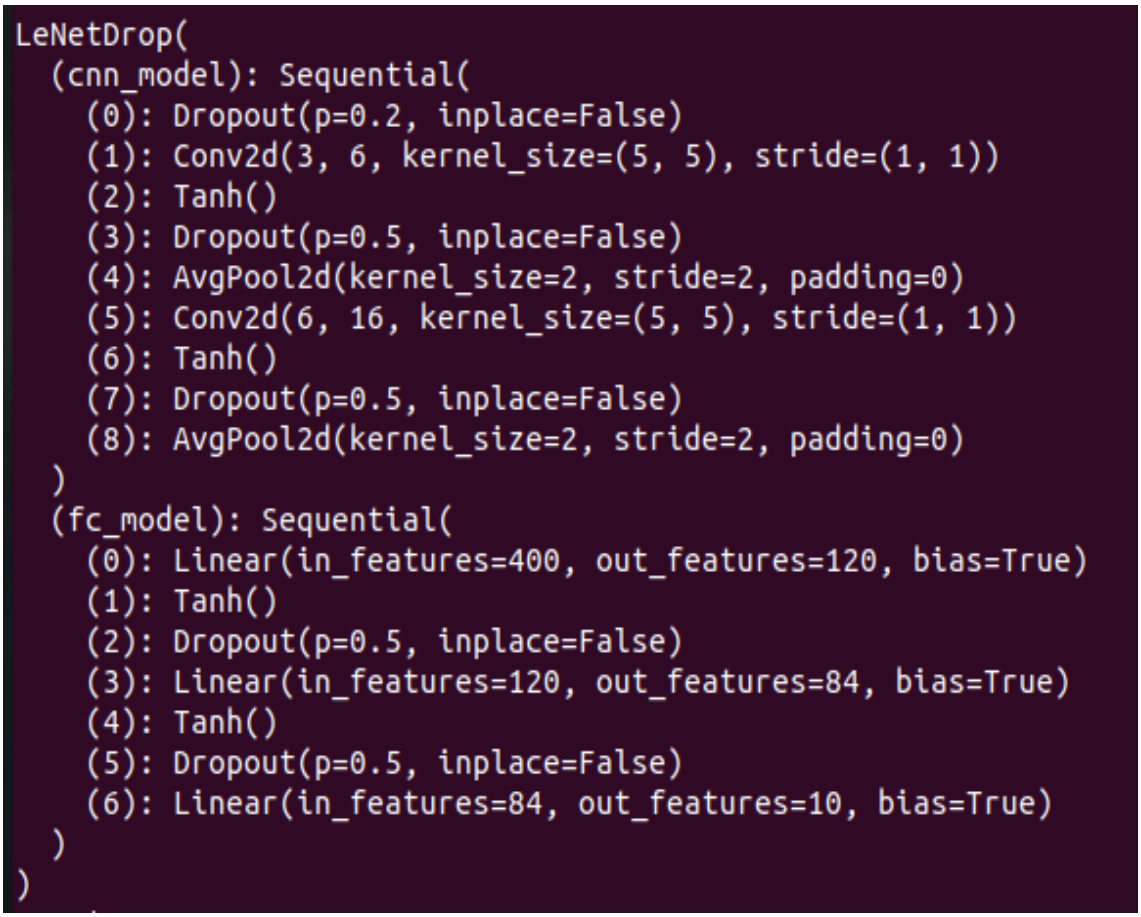


Figure 3: LeNetDp Model Summary

2.4.4 LeNet Architecture with L2-weight decay

L2 regularization, also known as weight decay, is a commonly used technique in CNNs to prevent overfitting of the model. The basic idea behind L2 regularization is to add a penalty term to the loss function that encourages the model to have smaller weight values. This penalty term is proportional to the square of the L2 norm of the weight matrix.

$$\mathcal{L}total = \mathcal{L}data + \lambda \sum_i w_i^2$$

$$= \mathcal{L}_{data} + \frac{\lambda}{2}|\mathbf{w}|_2^2$$

In the above equations, $\mathcal{L}total$ represents the total loss function of the CNN, which includes both the data loss $\mathcal{L}data$ and the L2 regularization term. The

regularization term penalizes the magnitudes of the weights w_i, with the regularization strength controlled by the hyperparameter λ. I used pytorch weight decay to implement L2 regularizer in my architecture.

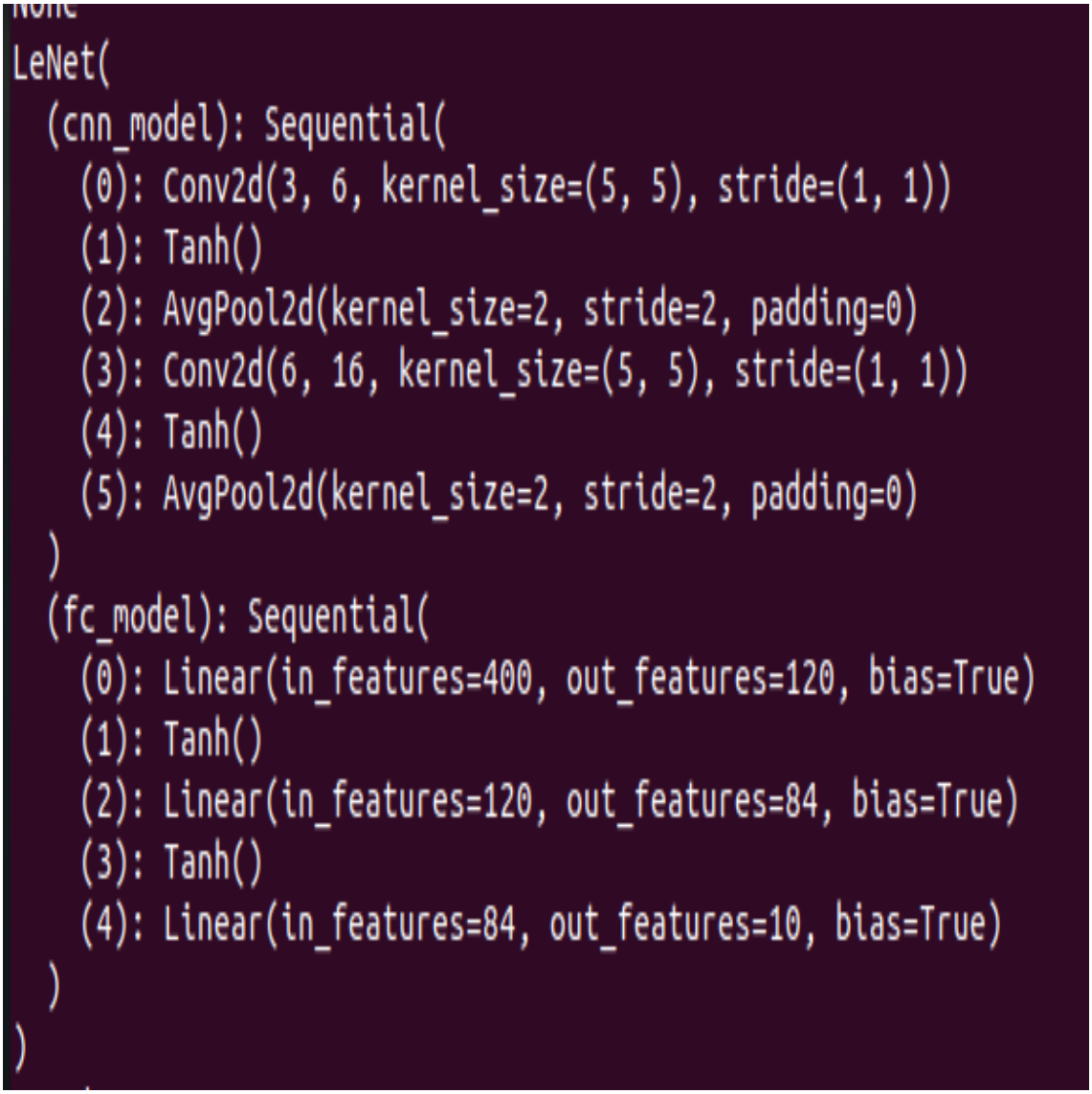


Figure 4: LeNetL2Reg Model Summary

2.4.5 LeNetBNDp Architecture

In this architecture, I'm using both batch normalization and dropouts together

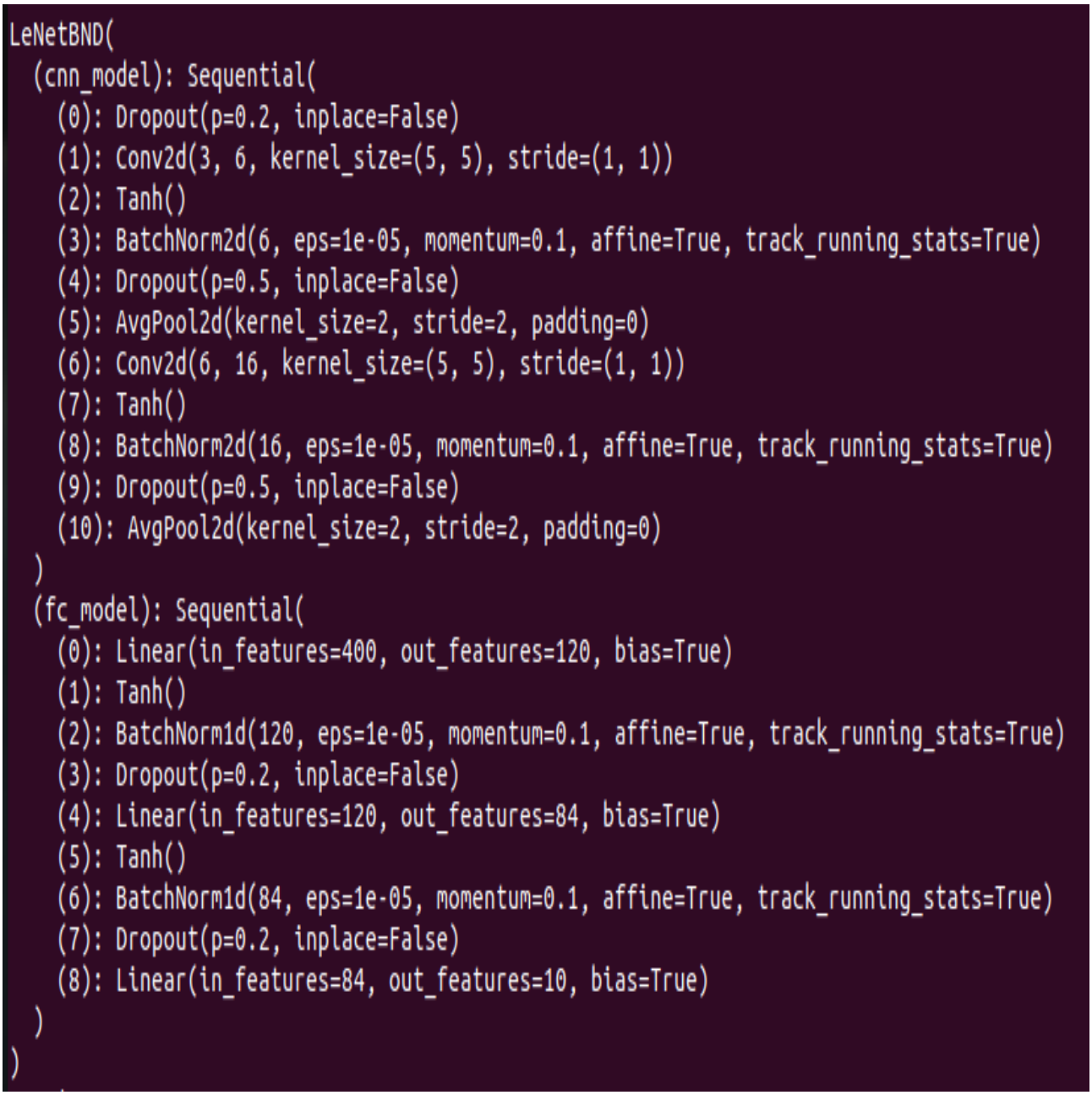


Figure 5: LeNetBNDp Model Summary

2.4.6 SuperLeNet Architecture

In SuperLeNet Architecture we bring all nice techniques and optimizations together which help us in increasing the performance not only on training set but also on validation and test datasets.

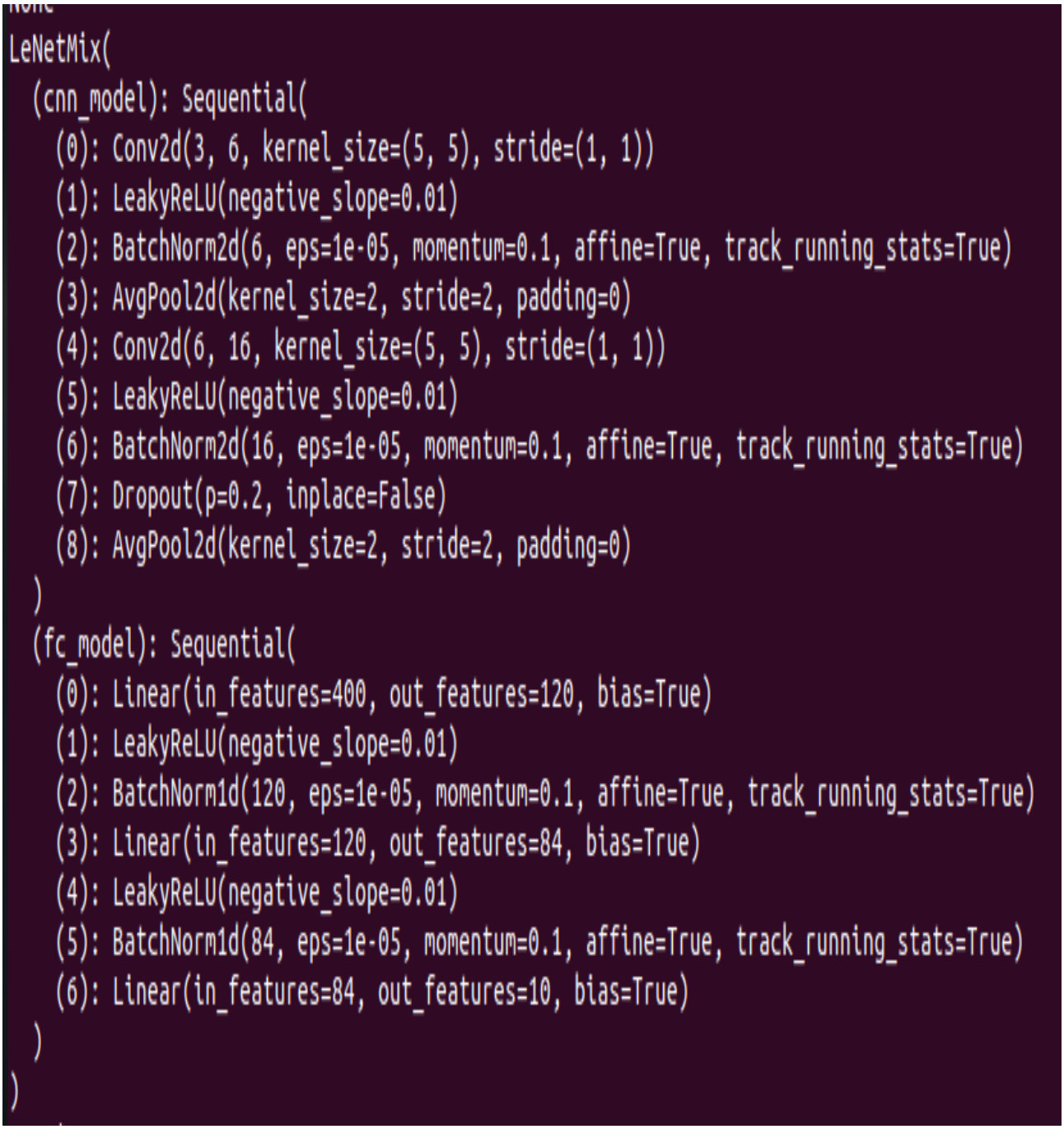


Figure 6: SuperLeNet Model Summary

2.5 RESNET Architectures

2.5.1 ResNet

ResNet (Residual Network) is a deep neural network architecture that was introduced in 2015. ResNet addresses the problem of vanishing gradients in deep neural networks by introducing a residual connection, or so called "skip connection," that allows the gradient to flow more easily through the network during backpropagation. The residual connection involves adding the output of a previous layer to the output of a subsequent layer, rather than directly passing the output through the subsequent layer. This also makes sure that we don't deviate a lot from the input during the process of training due to the large depth of the network.

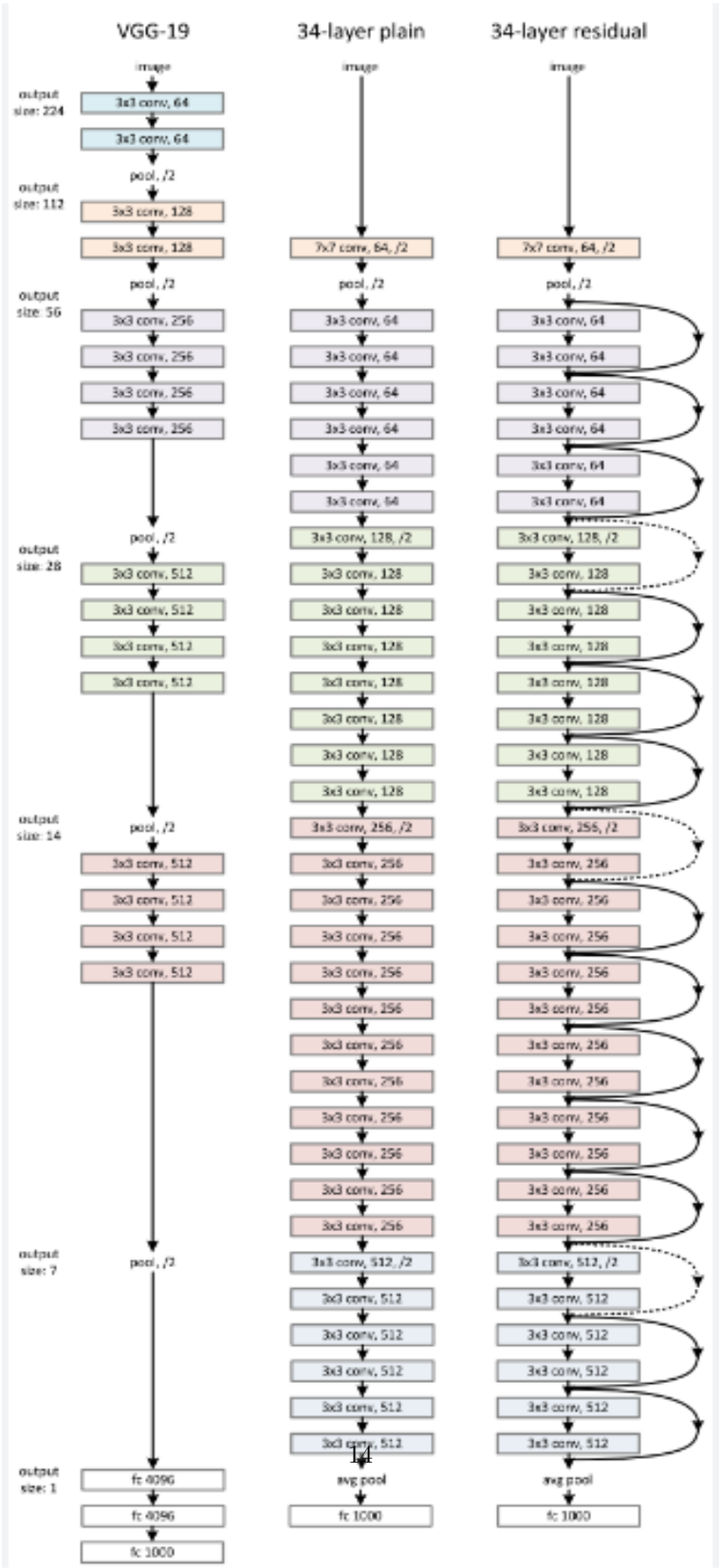


Figure 7: Skip Connections in ResNet

ResNet comes in different sizes, denoted by the number of layers in the architecture (e.g., ResNet18, Resnet34, ResNet50, ResNet101, ResNet152). The larger models have more layers and more complex structures, and are generally more accurate but also more computationally expensive. For Running ResNet on TinyImagenet as a sweetspot model we chose ResNet34.

There were many issues in running ResNet34 on TinyImagenet Dataset. ResNet34 is a shallow model compared to ResNet50, ResNet101, ResNet152 but still since TinyImageNet is also a smaller dataset, overfitting is bound to happen in this case. We have used many techniques to overcome the overfitting caused due to the above reasons. Some of them are:

DropOuts: Spatial DropOuts were added after BatchNormalization layers in the ResNet34 architecture
Regularizer: L2 Regularizer was used with a weight decay of 0.01
Image Augmentation: During training instead of passing the same image techniques like RandomHorizontalFlip, RandomCrop, Adding random noise to the channels of the image (Colour Jittering) were used. For better training all the images were finally resized into a fixed shape and were normalized.

2.6 Loss function

2.6.1 Cross Entropy Loss

The Cross-Entropy loss function is commonly used in classification tasks, particularly when the output of the model is a probability distribution over classes. The Cross-Entropy loss measures the difference between the predicted probability distribution and the true probability distribution of the classes.

Given N examples with C classes, let $y_{i,j}$ denote the true label for example i and class j, where $y_{i,j}$ is 1 if the true label for example i is class j, and 0 otherwise. Let $p_{i,j}$ denote the predicted probability of class j for example i. Then, the Cross-Entropy loss function is given by:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log(p_{i,j}) \tag{1}$$

where log is the natural logarithm.

The Cross-Entropy loss can be interpreted as the negative log-likelihood of the true class labels, given the predicted probability distribution. The loss is small when the predicted probabilities for the true class labels are high, and large when the predicted probabilities for the true class labels are low.

In pytorch implementation of Cross-Entropy Loss, the loss is often used in combination with softmax activation function in the output layer of a neural network. The softmax function converts the output of the neural network into a probability distribution over classes, which can then be used to compute the Cross-Entropy loss. The Cross-Entropy loss is commonly used in training neural networks for classification tasks, such as image recognition or natural language processing, where the goal is to predict the correct class label for a given input example.

2.7 Learning Algorithm

2.7.1 Gradient Descent

The Gradient Descent Method is a commonly used optimization technique in machine learning and statistics. It is a first-order optimization algorithm that can be used to minimize the given function. Gradient Descent or Steepest Descent is a line search method where the decent direction along which we move is opposite to the gradient at that point. The Algorithm of Gradient Descent is Mentioned below.

Algorithm 1 Gradient Descent

Require: Initial guess $x^{(0)}$, step size $\alpha > 0$, stopping criterion $\nabla f(x^{(k)})_2 < \epsilon$, for some $\epsilon > 0$

1: $k \leftarrow 0$
2: **while** $\nabla f(x^{(k)})_2 > \epsilon$ **do**
3: Compute gradient $\nabla f(x^{(k)})$
4: Update: $x^{(k+1)} \leftarrow x^{(k)} - \alpha \nabla f(x^{(k)})$
5: $k \leftarrow k + 1$
6: **end while**
7: **return** $x^{(k)}$

2.7.2 SPSA

Simultaneous Perturbation Stochastic Approximation (SPSA) is a stochastic optimization algorithm used for optimizing noisy, black-box functions. It is particularly useful in situations where gradient information is unavailable or difficult to compute. SPSA is commonly used in machine learning and optimization applications, such as reinforcement learning, online learning, and deep learning.

The basic idea of SPSA is to estimate the gradient of the objective function by perturbing the parameters in a randomized way. The gradient estimate is then used to update the parameters in a descent direction. Unlike other optimization algorithms, SPSA does not require the computation of gradients at each iteration. Instead, it estimates the gradient using a two-point finite differ-

ence formula.

$$\theta_{n+1} = \theta_n - a_n \widetilde{\nabla L}(\theta_n), \tag{2}$$

where θ_n denotes the parameter vector at iteration n, a_n is the step size at iteration n, and $\widetilde{\nabla L}(\theta_n)$ is the gradient estimate at iteration n.

The gradient estimate is computed using a two-point finite difference formula, which involves perturbing each component of the parameter vector in a random direction. Specifically, the perturbation Δ_n is sampled from a distribution D. The distribution D should be selected such that the elements in perturbation vector Δ_n are independent and symmetrically distributed about 0 with finite inverse moments. A popular choice for D is Rademacher distribution (Bernoulli distribution with equal probabilities of $+1$ and -1).The gradient estimate is then computed as follows:

$$\widetilde{\nabla L}(\theta_n) = \frac{L(\theta_n + c_n\Delta_n) - L(\theta_n - c_n\Delta_n)}{2c_n\Delta_n}, \tag{3}$$

where c_n is the perturbation size at iteration n,

Algorithm 2 SPSA

Require: Parameters: $a > 0$, $A \geq 0$, $c > 0$, $\alpha \in (0, 1]$, $\gamma \in (1/6, 1/2]$, and a distribution D.

1: **for** $k = 1, 2, 3, ...$ **do**
2: Sample a vector $\Delta \sim D_a$
3: $a_k \leftarrow \frac{a}{(k+A)^\alpha}$
4: $c_k \leftarrow \frac{c}{k^\gamma}$
5: $\hat{g} \leftarrow \frac{J(\theta + c_k\Delta) - J(\theta - c_k\Delta)}{2c_k\Delta_k}$
6: $\theta \leftarrow \theta - a_k\hat{g}$
7: **end for**
8: **Output**: Optimized parameter θ

I used gradient decent as my learning algorithm for training my neural network

2.8 Optimizers on top of Learning Algorithm

2.8.1 Adam

Adam (short for Adaptive Moment Estimation) is an optimization algorithm that combines the benefits of both momentum-based optimization and gradient scaling. It adapts the learning rate for each parameter based on the historical gradients, while also incorporating a momentum-like term that keeps track of the exponential moving average of the gradient and its square.

The update rule for Adam is given by:

$$m_t = \beta_1 v_{t-1} + (1 - \beta_1)\frac{\delta L}{\delta w_t} \tag{4}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\frac{\delta L}{\delta w_t})^2 \tag{5}$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} m_t \tag{6}$$

$$m_t = \frac{m_t}{1 - {\beta_1}^t} \tag{7}$$

$$v_t = \frac{v_t}{1 - {\beta_2}^t} \tag{8}$$

where $\theta_{t,i}$ is the i-th component of the parameter vector at iteration t, $\nabla_{\theta_i} J(\theta_{t-1})$ is the gradient of the loss function with respect to the i-th parameter at iteration $t-1$, α is the learning rate, β_1 and β_2 are decay rates that control the contribution of past gradients and their squares respectively, and ϵ is a small constant added to the denominator to prevent division by zero. The terms $m_{t,i}$ and $v_{t,i}$ are the first and second moments of the gradient, respectively.

By using the exponential moving average of the gradient and its square, Adam is able to adapt the learning rate for each parameter based on both the current and past gradients. Additionally, the momentum-like term in the update rule allows Adam to handle noisy gradients and converge more quickly in directions with consistent gradients.

In practice, Adam is widely used as a default optimization algorithm due to its strong performance and adaptability to a wide range of problem types.

2.9 Evaluation

Since the above Tasks involved were classification tasks we used normal accuracy as our evaluation metric.

$$Accuracy = \frac{Number\ of\ Correct\ Instances}{Total\ Number\ of\ Instances} \tag{9}$$

3 Optimizations for improving performance and reducing computational cost while Training the Model

3.1 Intialization of Weights and biases of a Neural Network

Initialization of weights and biases can play an important role in Neural Nets. Initializing weights and biases wrongly can have many effects starting from increasing computational cost to saturating all neurons in the model due to which

your model stops learning.

3.1.1 Xavier Initialization

Xavier initialization, also known as Glorot initialization, is a widely used method for weight initialization in deep neural networks. It aims to set the initial weights of the network in a way that helps improve the convergence and performance of the network during training.

The Xavier initialization method computes the initial weights using a Gaussian distribution with zero mean and a variance that depends on the size of the input and output layers of a particular layer. The variance for the Gaussian distribution is given by the following equation:

$$\text{variance} = gain^2 \times \frac{2}{\text{fan\_in} + \text{fan\_out}} \tag{10}$$

where fan_in is the number of input units in the layer, and fan_out is the number of output units in the layer. *gain* is a numerical constant associated with the activation function you are using. For example, RELU has the *gain* as $sqrt(2)$.

The Xavier initialization method can be applied to different activation functions, including sigmoid, tanh, and ReLU. For example, for a layer with a sigmoid activation function, the weights can be initialized as:

$$\text{weight} \sim \mathcal{N}\left(0, varience\right) \tag{11}$$

Xavier initialization helps in reducing the issue of vanishing or exploding gradients during training, which can result in slow convergence or poor performance of neural networks. It is a widely used initialization method in practice and has been shown to be effective in improving the training dynamics and performance of deep neural networks.

3.1.2 Kaiming/He Initialization

The He or Kaiming initialization is a weight initialization technique designed specifically for deep neural networks that use the ReLU activation function. It helps to prevent vanishing and exploding gradients by initializing weights in a way that the variance of the outputs of each layer is equal to the variance of the inputs.

Assuming a layer has n_{in} inputs and n_{out} outputs, the He initialization sets the weights of that layer using a Gaussian distribution with a mean of 0 and a standard deviation of:

$$\text{variance} = \frac{gain}{\sqrt{fan\_mode}} \tag{12}$$

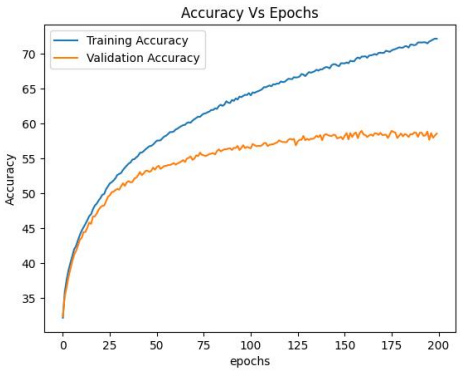
where fan_mode can be fan_in or fan_out. *gain* is a numerical constant associated with the activation function you are using

For example, for a layer with a sigmoid activation function, the weights can be initialized as:

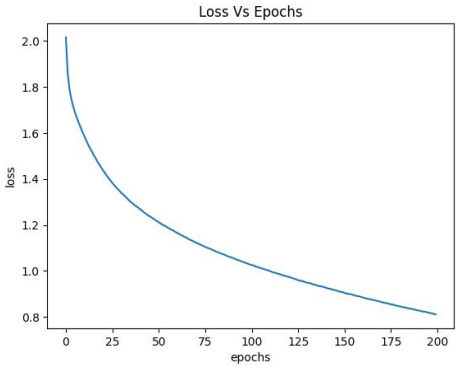
$$\text{weight} \sim \mathcal{N}(0, varience) \tag{13}$$

4 Experiments and Results

4.1 LeNet Architectures on CIFAR-10 dataset

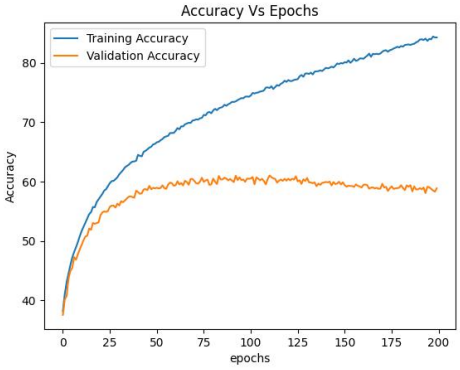


(a) LeNet Accuracy plot for 200 epochs

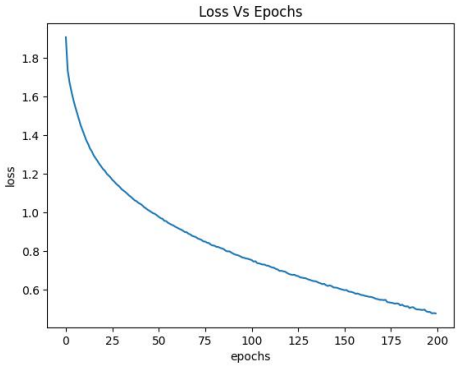


(b) LeNet Average Loss plot for 200 epochs

Figure 8: LeNet plots

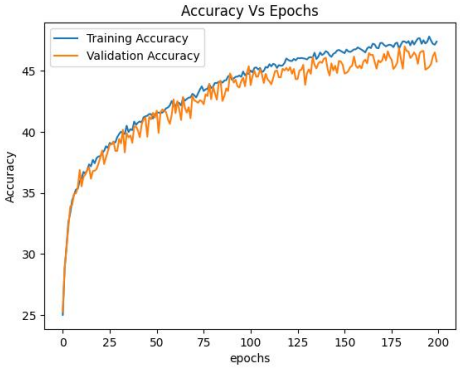


(a) LeNet with Normalization Accuracy plot for 200 epochs

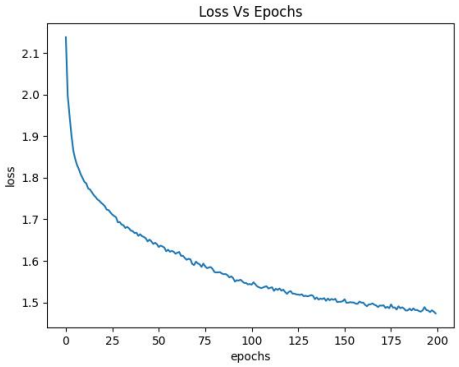


(b) LeNet with Normalization Average Loss plot for 200 epochs

Figure 9: LeNetBN plots

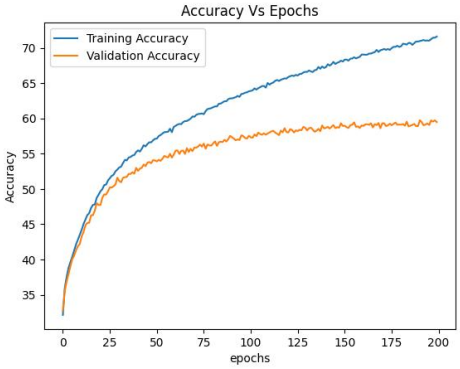


(a) LeNet with Dropouts Accuracy plot for 200 epochs

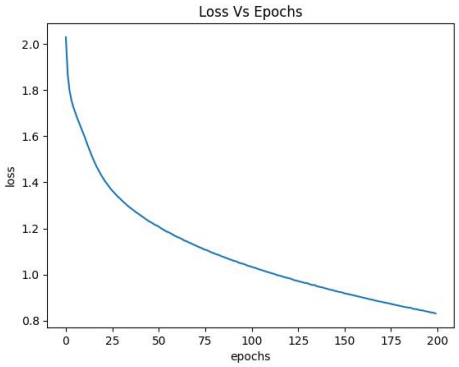


(b) LeNet with Dropouts Average Loss plot for 200 epochs

Figure 10: LeNetDp plots

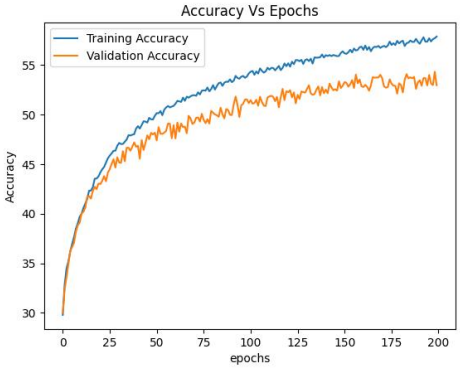


(a) LeNet with L2 Reg Accuracy plot for 200 epochs

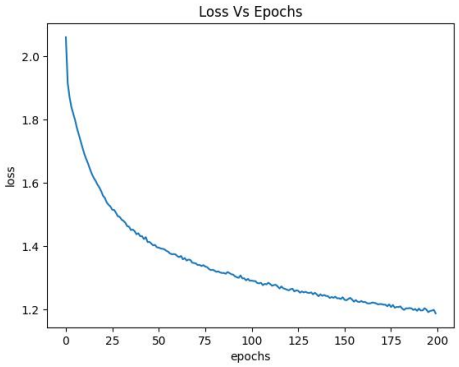


(b) LeNet with L2 Reg Average Loss plot for 200 epochs

Figure 11: LeNetL2Reg plots

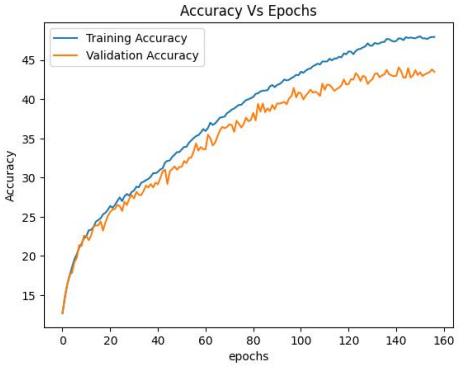


(a) LeNet with both Batch Normalization and DropOuts Accuracy plot for 200 epoch

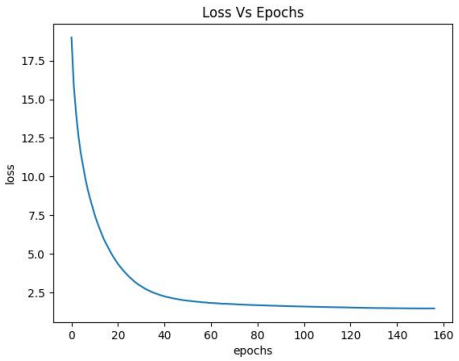


(b) LeNet with both Batch Normalization and DropOuts Average Loss plot for 200 epochs

Figure 12: LeNetBNDp plots

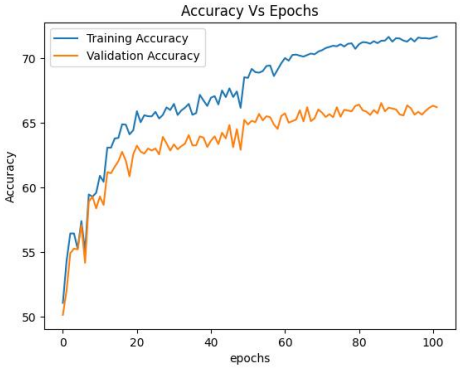


(a) Mixed LeNet wrong weights Accuracy plot for 200 epochs

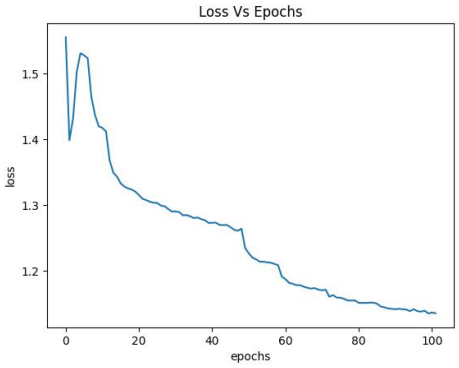


(b) Mixed LeNet wrong weights Average Loss plot for 200 epochs

Figure 13: SuperLeNetNoInit plots



(a) Mixed LeNet wrong weights Accuracy plot for 200 epochs



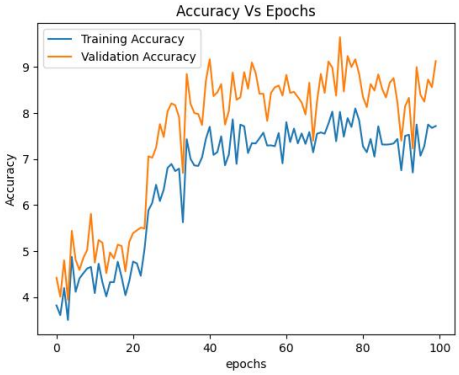
(b) Mixed LeNet wrong weights Average Loss plot for 200 epochs

Figure 14: SuperLeNet plots

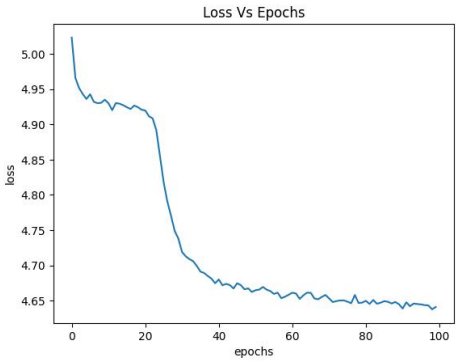
Model	**Train Accuracy**	**Validation Accuracy**	**Test Accuracy**	**GPU Time (in sec)**	**GPU Utilization**	**GPU Memory Usage in GB**	**Number of Parameters**
LeNet	69.60	58.96	58.98	3181.56	6%	1.70	62,006
LeNetBN	**75.77**	60.42	60.77	3142.94	9%	1.78	62,458
LeNetDp	47.28	45.92	46.50	3131.83	7%	1.70	62,006
LeNetBNDp	57.68	54.00	54.23	3198.16	9%	1.78	62,458
LeNet with L2 weight decay	70.96	59.74	59.43	3071.10	6%	1.70	62,006
SuperLeNet with Wrong Weights Init	47.49	43.22	44.47	2425.49	10%	1.79	62,458
SuperLeNet with Proper Weight Init	71.38	**66.32**	**66.00**	**1592.60**	**10%**	**1.79**	62,458

Table 1: Comparision between different LeNet Models on CIFAR-10

4.2 LeNet Architectures on TinyImageNet

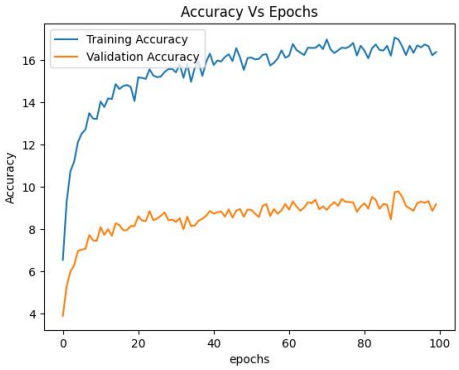


(a) LeNet Accuracy plot for 100 epochs

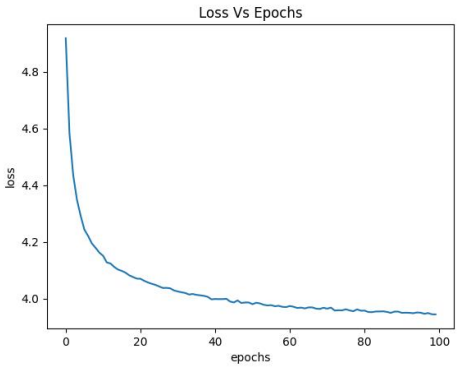


(b) LeNet Average Loss plot for 100 epochs

Figure 15: LeNetImage plots

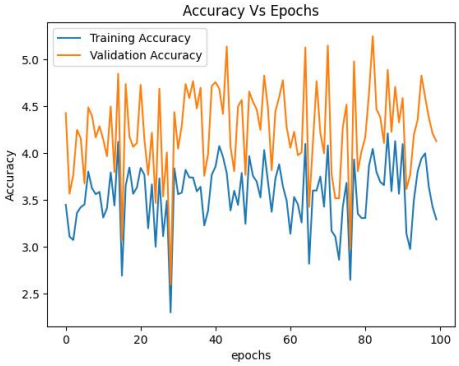


(a) LeNet with Normalization Accuracy plot for 100 epochs

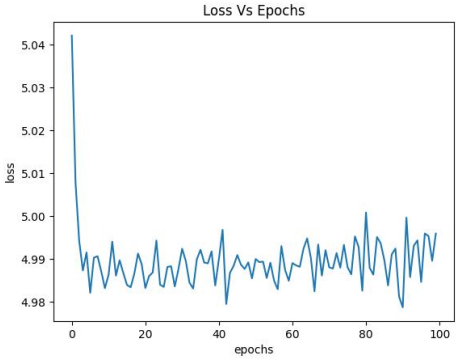


(b) LeNet with Normalization Average Loss plot for 100 epochs

Figure 16: LeNetBNImage plots

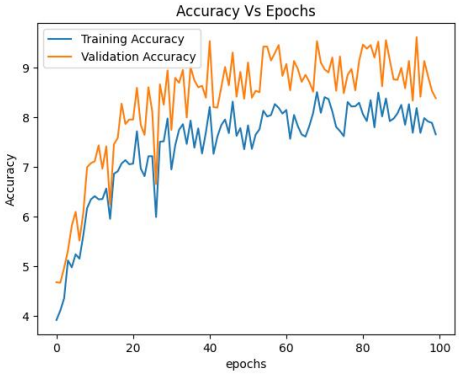


(a) LeNet with Dropouts Accuracy plot for 100 epochs

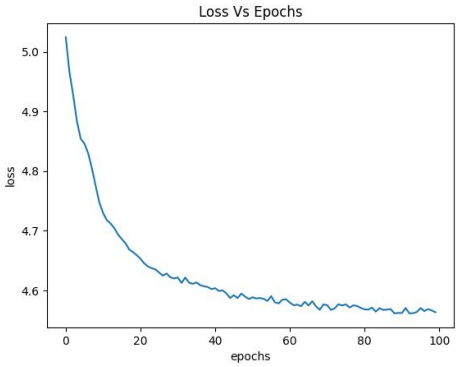


(b) LeNet with Dropouts Average Loss plot for 100 epochs

Figure 17: LeNetDpImage plots

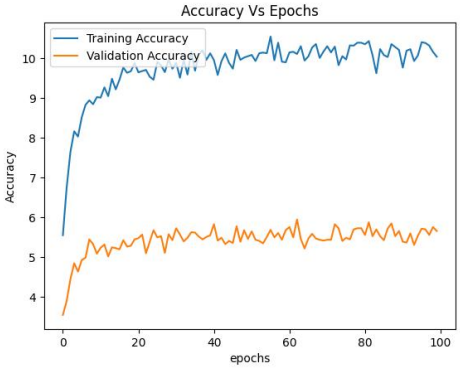


(a) LeNet with L2 Reg Accuracy plot for 100 epochs

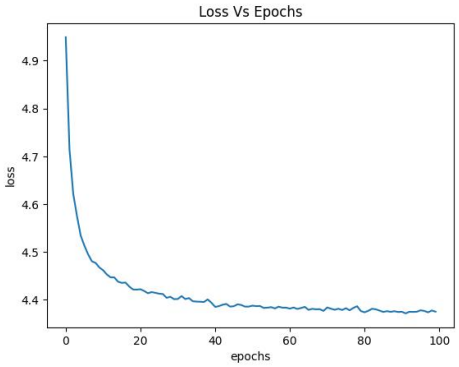


(b) LeNet with L2 Reg Average Loss plot for 100 epochs

Figure 18: LeNetL2RegImage plots

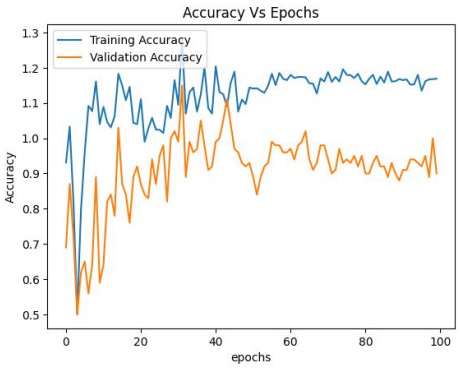


(a) LeNet with both Batch Normalization and DropOuts Accuracy plot for 100 epoch

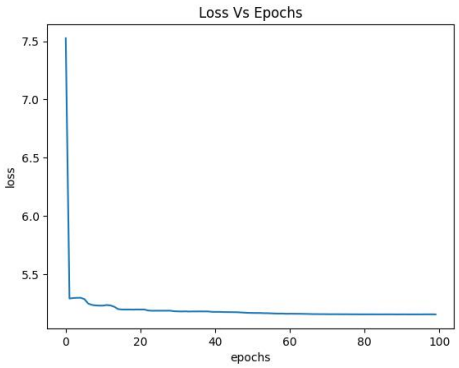


(b) LeNet with both Batch Normalization and DropOuts Average Loss plot for 100 epochs

Figure 19: LeNetBNDpImage plots

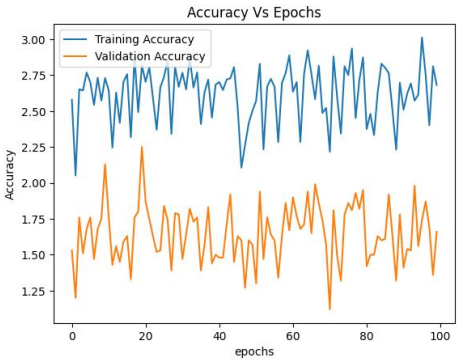


(a) Mixed LeNet wrong weights Accuracy plot for 100 epochs

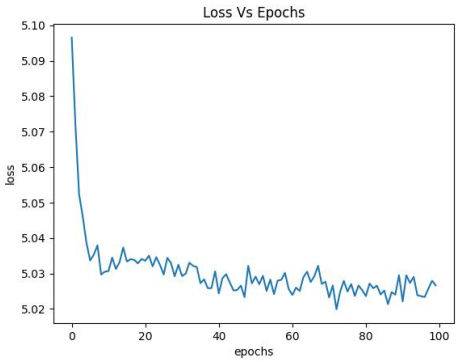


(b) Mixed LeNet wrong weights Average Loss plot for 100 epochs

Figure 20: SuperLeNetNoInitImage plots



(a) Mixed LeNet wrong weights Accuracy plot for 100 epochs



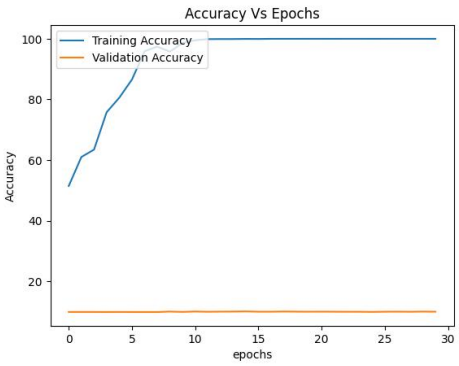
(b) Mixed LeNet wrong weights Average Loss plot for 100 epochs

Figure 21: SuperLeNet plots

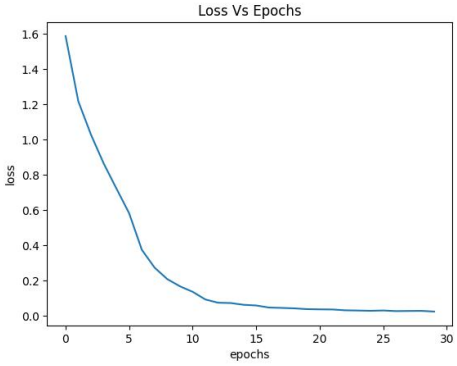
Model	**Train Accuracy**	**Validation Accuracy**	**GPU Time (in sec)**	**GPU Utilization**	**GPU Memory Usage in GB**	**Number of Parameters**
LeNet	8.0	9.65	4507.04	2%	1.76	83,072
LeNetBN	17.00	9.78	4518.99	3%	1.79	83,116
LeNetDp	4.01	4.90	4504.29	3%	1.76	83,072
LeNetBNDp	10.12	5.42	4531.46	3%	1.79	83,116
LeNet with L2 weight decay	8.13	9.62	4505.04	3%	1.76	83,072
SuperLeNet with Wrong Weights Init	**17.24**	**8.36**	3973.02	**5%**	**1.79**	83,116
SuperLeNet with Proper Weight Init	11.82	5.89	**3966.93**	5%	1.79	83,116

Table 2: Comparision between different LeNet Models on TinyImageNet

4.3 ResNet Architectures on CIFAR-10

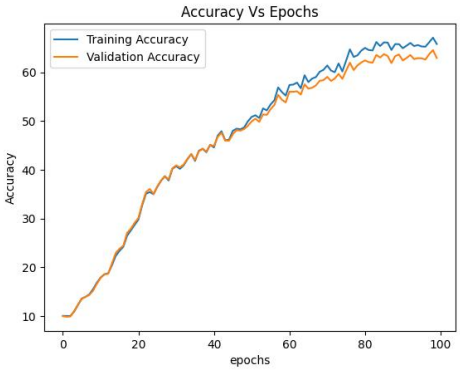


(a) Accuracy of ResNet on CIFAR-10 with no DropOuts

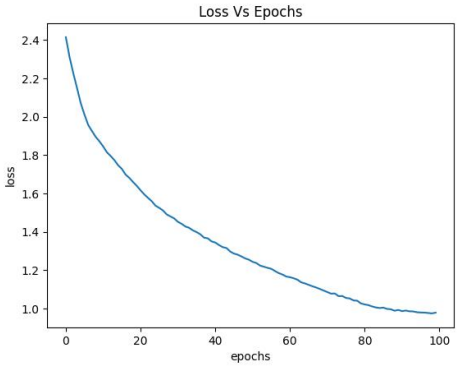


(b) Loss of ResNet on CIFAR-10 with no DropOuts

Figure 22: Plane ResNet plots

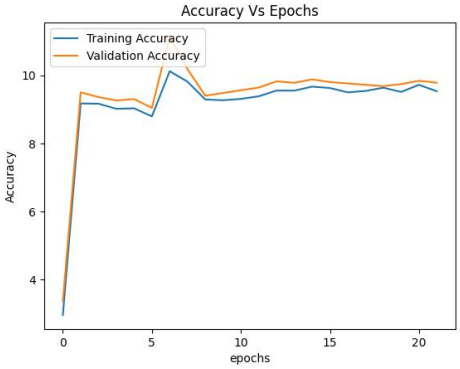


(a) Accuracy of ResNet on CIFAR-10 with DropOuts

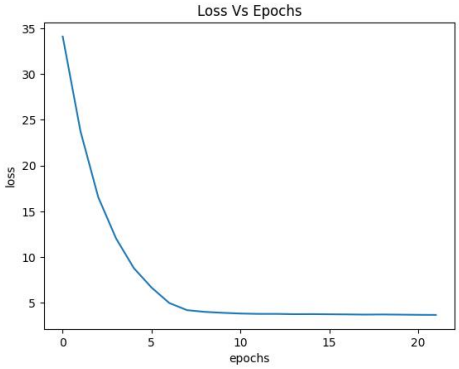


(b) Loss of ResNet on CIFAR-10 with DropOuts

Figure 23: ResNet with Dropout plots

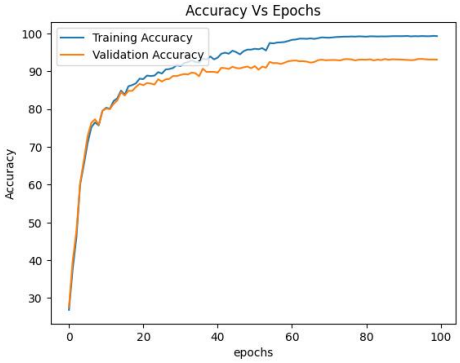


(a) Accuracy of SuperResNetWrongInit on CIFAR-10

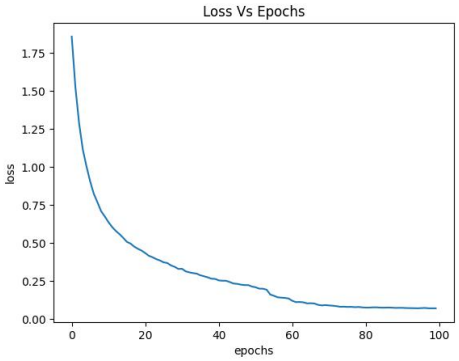


(b) Loss of SuperResNetWrongInit on CIFAR-10

Figure 24: SuperResNetWrongInit plots



(a) Accuracy of SuperResNet on CIFAR-10



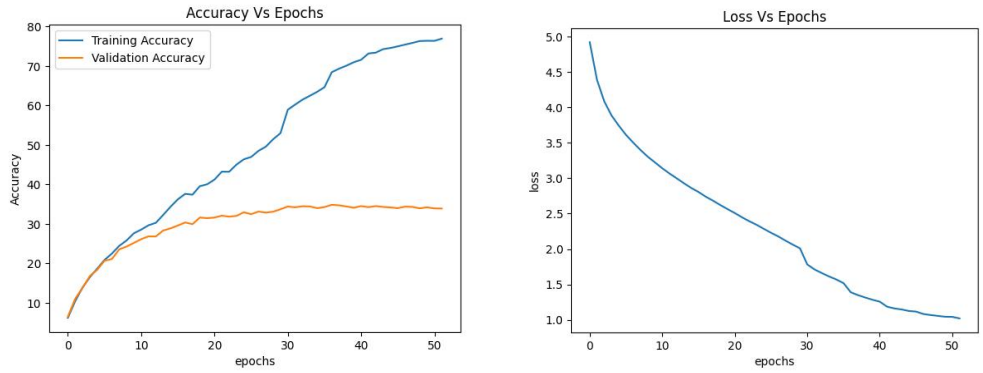
(b) Loss of SuperResNet on CIFAR-10

Figure 25: SuperResNet plots

Model	Train Accuracy	Validation Accuracy	Test Accuracy	GPU Time (in sec)	GPU Utilization	GPU Memory Usage (in GB)	Number of Parameters
ResNet without any dropouts	98.58	88.78	86.10	3299.12	90%	7.25	21,286,346
ResNet with dropouts	91.41	87.34	86.27	13638.76	91%	7.32	21,286,346
SuperResNet with Improper Initialization	9.69	9.78	9.83	3091.24	92%	7.34	21,286,346
SuperResNet with Proper weight Initialization	**99.30**	**93.12**	**93.03**	13853.55	**93%**	**7.34**	21,286,346

Table 3: Comparision between different ResNet Models on CIFAR-10 Datset

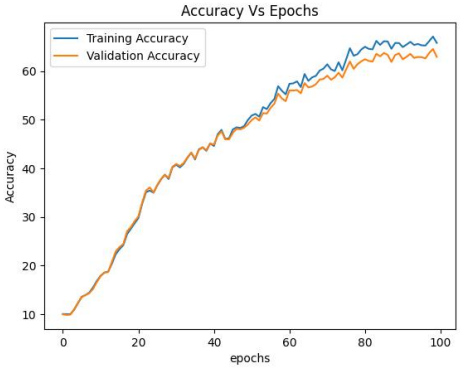
4.4 ResNet on TinyImageNet Dataset



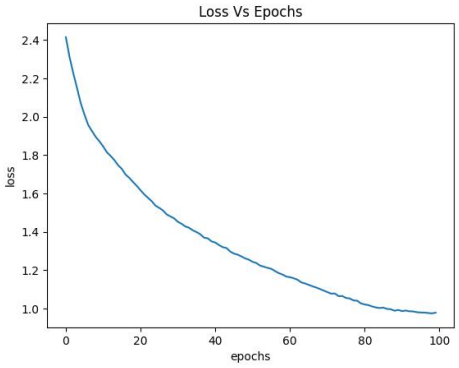
(a) Accuracy of ResNet on TinyImageNet with no DropOuts

(b) Loss of ResNet on TinyImageNet with no DropOuts

Figure 26: Plane ResNet plots

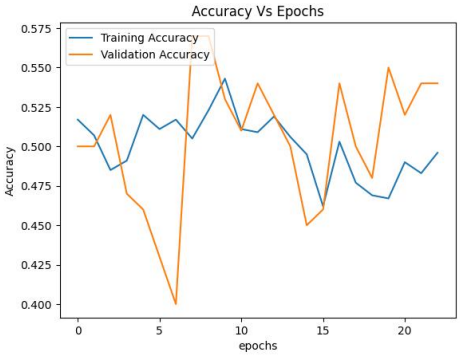


(a) Accuracy of ResNet on TinyImageNet with DropOuts

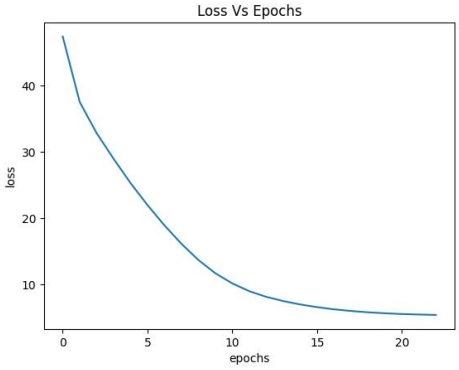


(b) Loss of ResNet on TinyImageNet with DropOuts

Figure 27: ResNet with Dropouts

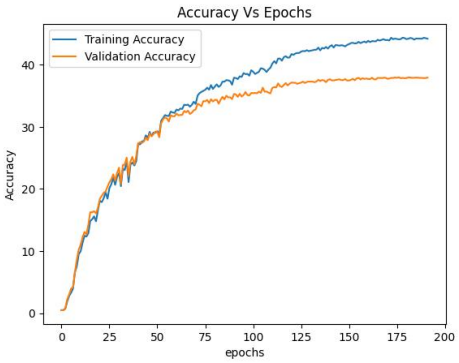


(a) Accuracy of SuperResNetWrongInit on TinyImageNet

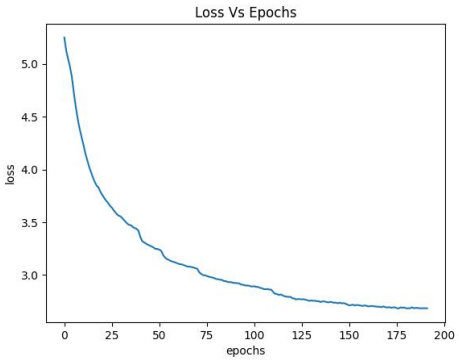


(b) Loss of SuperResNetWrongInit on TinyImageNet

Figure 28: SuperResNetWrongInit plots



(a) Accuracy of SuperResNet on TinyImageNet



(b) Loss of SuperResNet on TinyImageNet

Figure 29: SuperResNet plots

Model	**Train Accuracy**	**Validation Accuracy**	**GPU Time (in sec)**	**GPU Utilization**	**GPU Memory Usage (in GB)**	**Number of Parameters**
ResNet without any dropouts	**76.86**	33.89	3318.45	25%	2.32	21,383,816
ResNet with dropouts	45.98	35.66	5299.38	28%	2.33	21,383,816
SuperResNet with Improper Initialization	0.50	0.54	1371.86	29%	2.39	21,383,816
SuperResNet with Proper weight Initialization	45.35	**40.11**	12715.88	**30%**	**2.40**	21,383,816

Table 4: Comparision between different ResNet Models on TinyImageNet Dataset

4.5 Importance of Proper Intialization of weights and biases of a NeuralNet

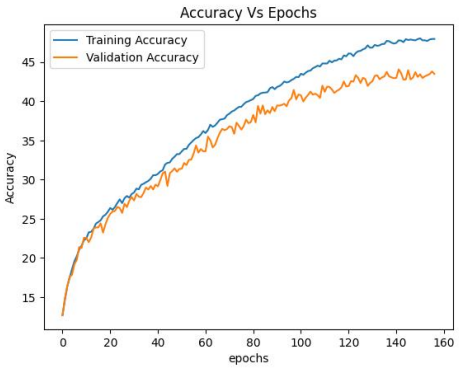
We took our final LeNet model (SuperLeNet) and did two experiments one with correct initialization off weights where weights are initialized according to kaiming initialization and biases are initialized to 0.01 instead of zero to give some randomness in learning. In the second case weights are initialized to large guassian random values and biases to 0.1 to show that high weights and biases at initialization increase the length of the hockey stick performance observed. Since we have 200 classes, even if the model predicts all images as a single class

the cross entropy loss in this case will be $-log(1/10)$ which is roughly (2.302). So if our initial loss is less than the above loss, we are starting from a good random point. For TinyImagenet since we have 200 classes, with good initialization we can start from a loss of $-log(1/500)$ which is roughly (5.3).

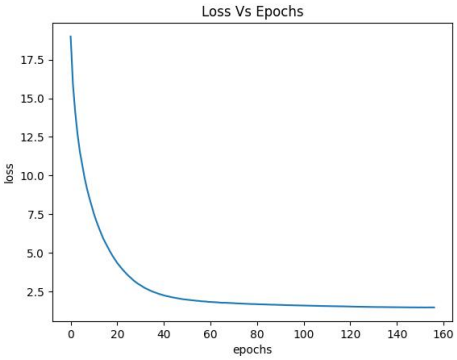
As we can see below, the losses are indeed affected by the initialization of weights and biases

4.5.1 Effect of Weight Initialization on CIFAR-10

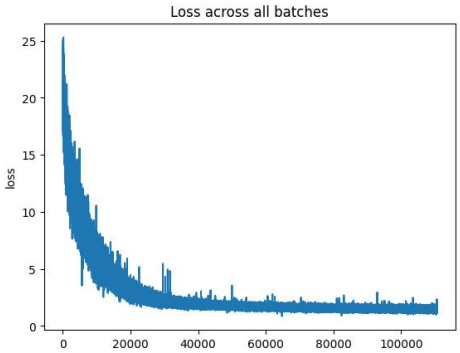
Best Case loss with Proper initialization = 2.29
Worst Case Loss with large weights and biases = 18.4524



(a) SuperLeNet Accuracy with large weights for 200 epochs

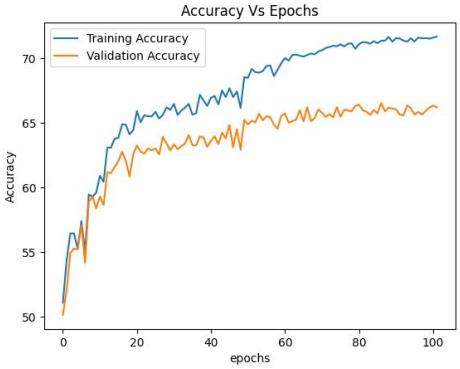


(b) SuperLeNet Loss with large weights for 200 epochs

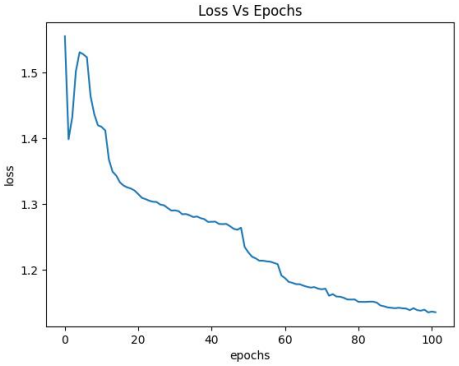


(c) SuperLeNet Hockey stick with large weights for 200 epochs

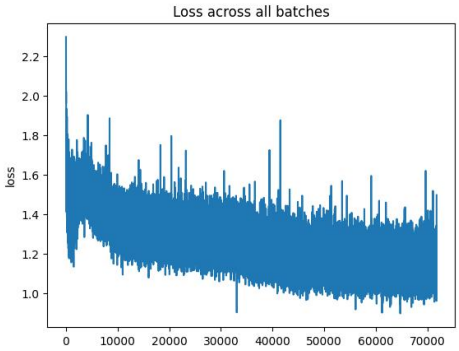
Figure 30: SuperLeNet with Improper Weight Initialization



(a) SuperLeNet Acuracy with kaiming intialization for 200 epochs

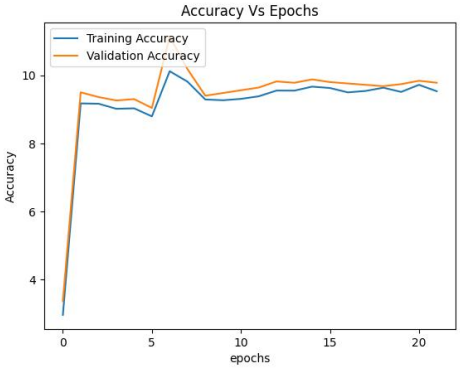


(b) SuperLeNet Loss with kaiming intialization for 200 epochs

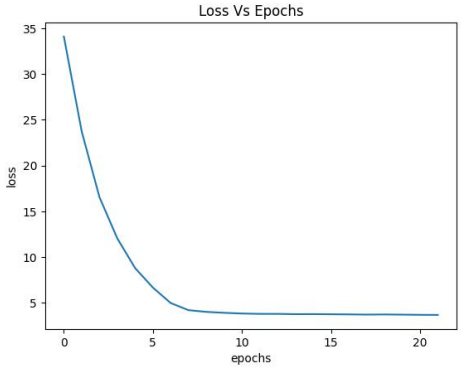


(c) SuperLeNet Hockey stick with kaiming intialization for 200 epochs

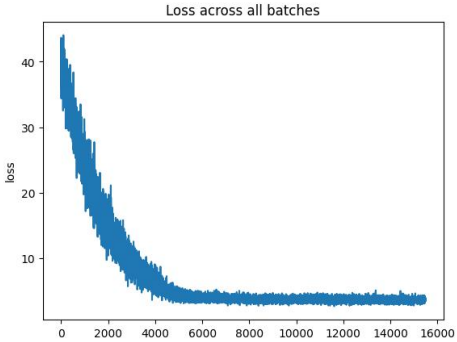
Figure 31: SuperLeNet with Proper Weight Initialization



(a) SuperResNet Accuracy with large weights for 200 epochs

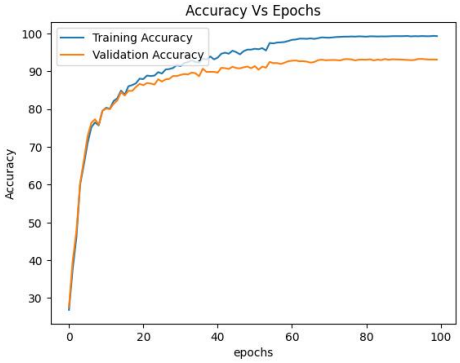


(b) SuperResNet Loss with large weights for 200 epochs

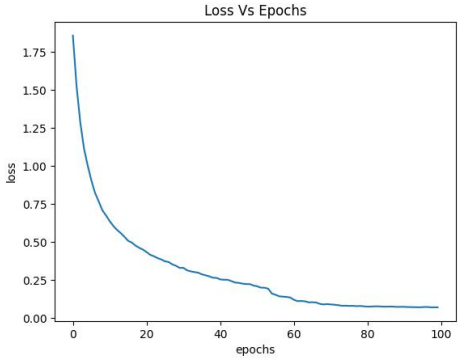


(c) SuperResNet Hockey stick with large weights for 200 epochs

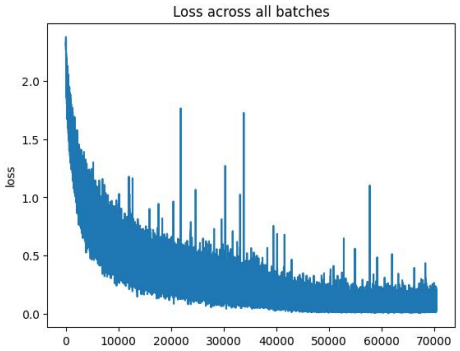
Figure 32: SuperResNet with Improper Weight Initialization



(a) SuperResNet Acuracy with kaiming intialization for 200 epochs



(b) SuperResNet Loss with kaiming intialization for 200 epochs



(c) SuperResNet Hockey stick with kaiming intialization for 200 epochs

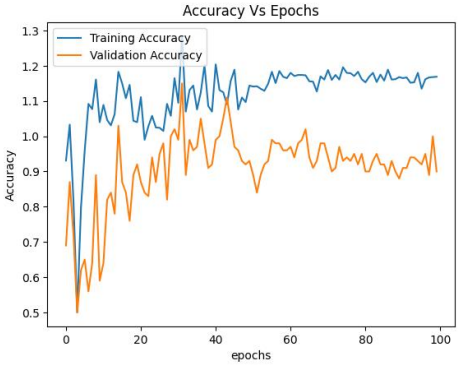
Figure 33: SuperResNet with Proper Weight Initialization

Model	Train Accuracy	Validation Accuracy	Test Accuracy	Initial Loss	GPU Execution Time (in sec)
SuperLeNet with kaiming Initialization	71.38	66.32	66.00	**2.2972**	**1592.60**
SuperLeNet with Improper Initialization	47.49	43.22	44.47	18.4523	2425.49
Super ResNet with kaiming Initialization	**99.30**	**93.12**	**93.03**	2.3045	13853.55
Super ResNet with Improper Initialization	9.69	9.78	9.83	40.1472	3091.24

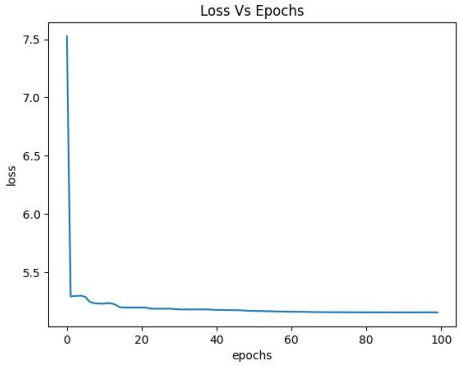
Table 5: Effect of Initialization on Training using CIFAR10

4.5.2 Effect of Weight Initialization on TinyImageNet

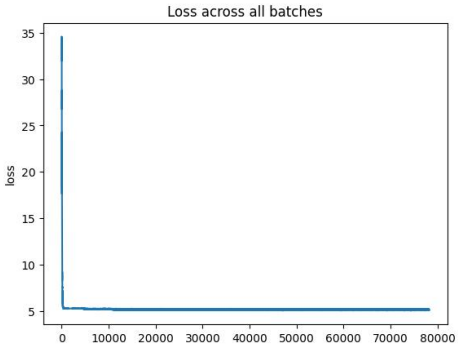
Best Case loss with Proper initialization = 5.321
Worst Case loss with large weights and biases = 56.81



(a) SuperLeNet Accuracy with large weights for 100 epochs

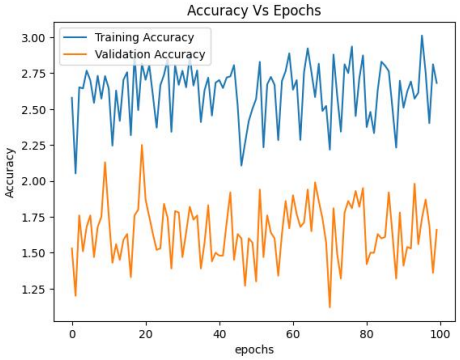


(b) SuperLeNet Loss with large weights for 100 epochs

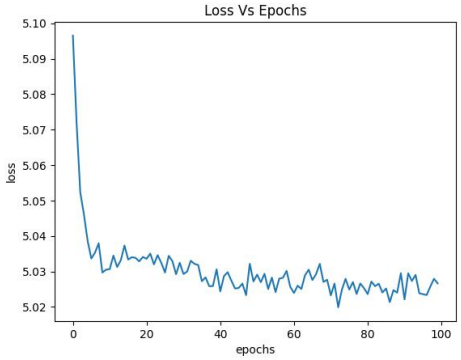


(c) SuperLeNet Hockey stick with large weights for 100 epochs

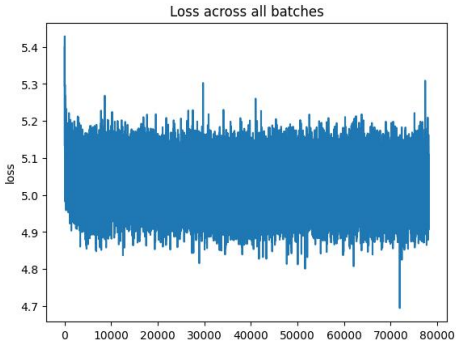
Figure 34: SuperLeNet with Improper Weight Initialization



(a) SuperLeNet Acuracy with kaiming intialization for 100 epochs

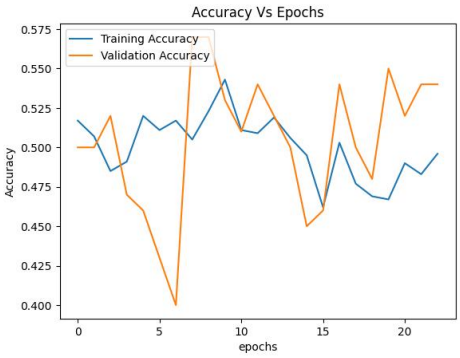


(b) SuperLeNet Loss with kaiming intialization for 100 epochs

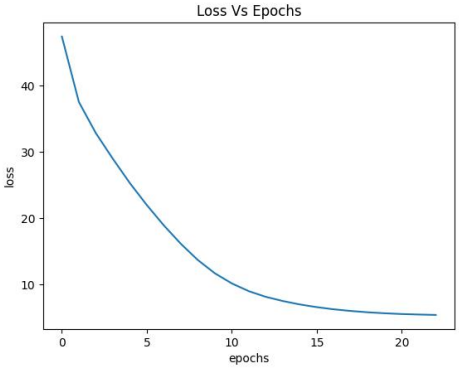


(c) SuperLeNet Hockey stick with kaiming intialization for 100 epochs

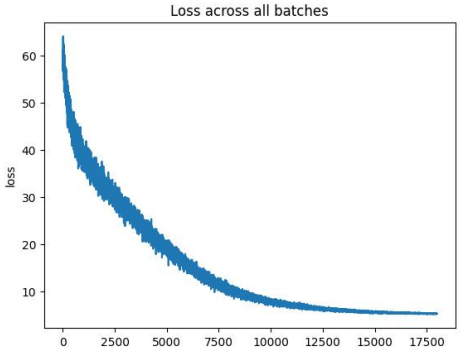
Figure 35: SuperLeNet with Proper Weight Initialization



(a) SuperResNet Accuracy with large weights for 200 epochs

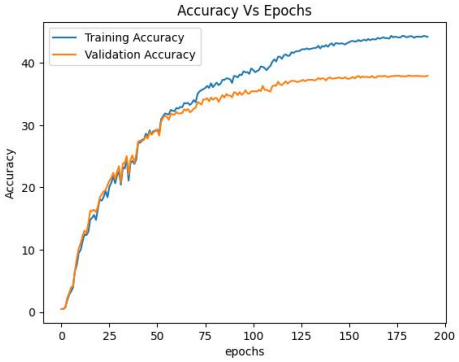


(b) SuperResNet Loss with large weights for 200 epochs

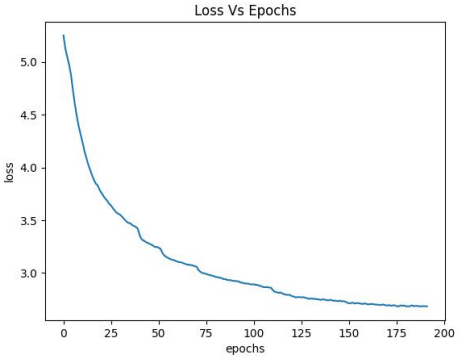


(c) SuperResNet Hockey stick with large weights for 200 epochs

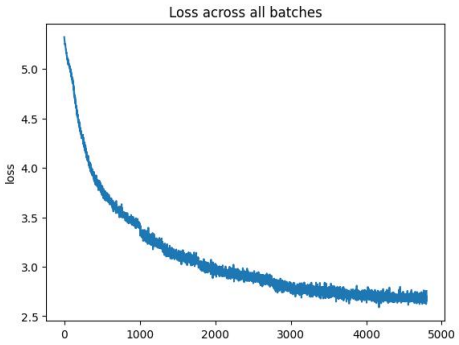
Figure 36: SuperLeNet with Improper Weight Initialization



(a) SuperResNet Acuracy with kaiming intialization for 200 epochs



(b) SuperResNet Loss with kaiming intialization for 200 epochs



(c) SuperResNet Hockey stick with kaiming intialization for 200 epochs

Figure 37: SuperLeNet with Proper Weight Initialization

Model	Train Accuracy	Validation Accuracy	Initial Loss	GPU Execution Time (in sec)
SuperLeNet with kaiming Initialization	11.82	5.89	**5.321**	3966.93
SuperLeNet with Improper Initialization	17.24	8.36	36.066	3973.02
Super ResNet with kaiming Initialization	**45.35**	**40.11**	5.324	7624.68
Super ResNet with Improper Initialization	0.50	0.54	56.81	**1371.86**

Table 6: Effect of Initialization on Training using TinyImageNet

4.6 Effect of Batch Size

To experiment how batch size affects the training time and performance, I tested SuperResNet model with proper weight Initialization on TinyImageNet Dataset with different batch sizes.

From the below Image and Results we can clearly see that increasing batch size helps us as the gradient updates are much smoother which helps us in converging more closer to the point of minima. But, due to this overfitting also can happen as we can see as the batch size increases, Training Accuracies and Validation Accuracies start diverging farther clearly showing us that the model is overfitting. So whenver we increase batch size we should also keep in mind that model is prone to overfit and we should use regularization to prevent overfitting.

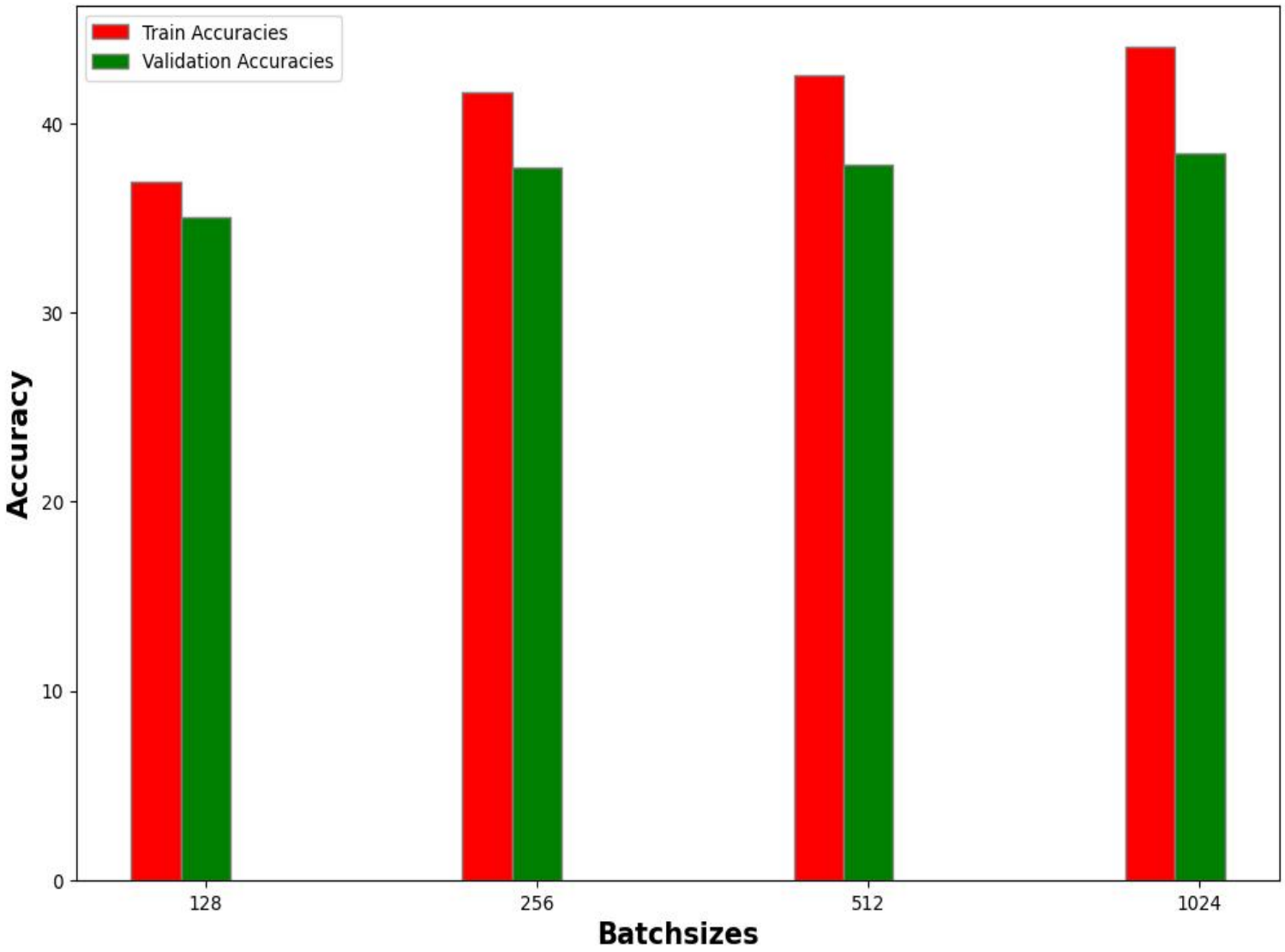


Figure 38: SuperResNet Training and Validation Accuracies vs batch size

Batch Size	Train Accuracy	Validation Accuracy	GPU Memory Usage (in GB)	GPU Execution Time (in sec)
128	36.92	35.02	2.03	5549.40
256	41.65	37.63	2.48	4380.53
512	42.55	37.79	3.11	**3839.92**
1024	**44.04**	**38.40**	4.50	3980.64
2048	43.83	37.48	7.20	3984.07
4096	39.57	35.06	12.53	4005.28
8192	34.92	32.11	23.08	4164.34
16384	—	—	~ 46	—

Table 7: Effect of Batch Size on Training SuperResNet on TinyImageNet for 100 epochs

5 Language Modelling using Recurrent Neural Networks

6 The Six Jargons

6.1 Tasks

General Tasks in RNN can be broadly classified into the following three tasks:

- **Sequence Classification:** In Sequence Classification we mainly want to map or classify the input sequence to a single output class or multiple output classes (Classification vs Multi-Class Classification). Here Output classes is a predefined deterministic set of classes.
  Example: Sentiment Analysis of a sentence.

- **Sequence Labelling:** In Sequence Labelling we mainly want to map each input token to one output class. Unlike Sequence Classification where we map entire group of tokens to a single class, here we map each token to one output class.
  Example: Classifying each word in a sentence into it's corresponding parts of speech.

- **Sequence Translation:** In Sequence Translation we translate a given input sequence into an ouput sequence. They are popularly known **Seq2Seq**

models. Generally they are based on Encoder-Decoder Framework.
Example: Image Captioning, Audio to Text Conversion.

6.2 Models and Device Used

For each of the tasks all the three class of Models Simple RNNs, LSTMs and GRUs were used. All the experiments were ran on **NVIDIA 72C GPU** with a GPU Memory of size **4.0 GB**.

6.3 Simple RNN Architecture

RNNs are a class of neural networks that allows output of previous layer to be used as input to the next while having hidden states. The basic architecture of an RNN consists of a series of interconnected cells, where each cell takes as input the current data point and the output from the previous cell in the sequence. The output from the current cell is then passed as input to the next cell, and so on.

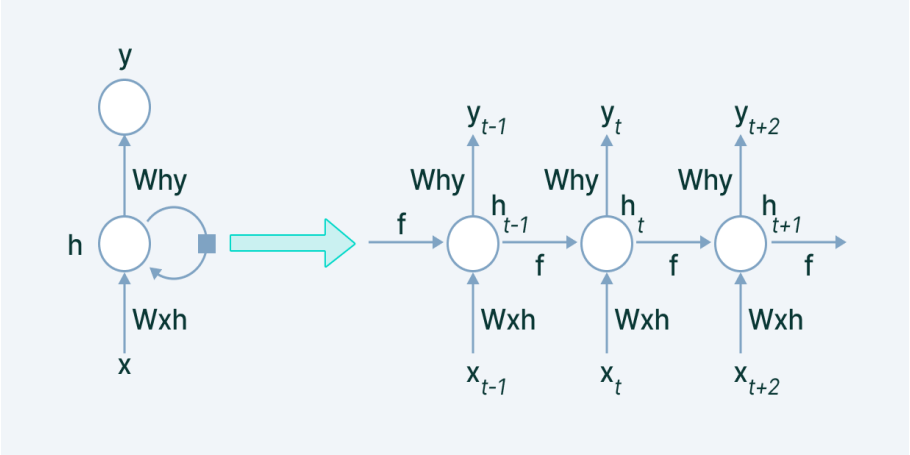


Figure 39: Simple RNN Architecture

Vanishing Gradient Problem is an important problem that occurs in RNNs. The gradients that are computed during backpropagation can become very small (i.e. they vanish) as they are propagated through many layers of the network. In RNN, the output of the earlier layers is used as the input for the further layers. Thus, the training for the time point t is based on inputs that are coming from the untrained layers. So, because of the vanishing gradient, the whole network is not being trained properly. In an RNN, the gradients must be propagated through time as well as through the layers of the network. This means that if the gradients are too small, they may not be able to effectively update the weights of the network, resulting in slow or even stalled learning.

6.4 LSTM Architecture

LSTM networks are an extension of RNNs mainly used wherever RNN fails. The basic difference between RNNs and LSTMs is that the hidden layer of LSTM is a gated cell. A cell, an input gate, an output gate and a forget gate make up a typical LSTM unit. The input gate controls how much of the current input should be added to the cell state, the forget gate controls how much of the previous cell state should be retained, and the output gate controls how much of the current cell state should be output. These gates are controlled by sigmoid activation functions, which allow them to selectively filter the information flowing through the network.

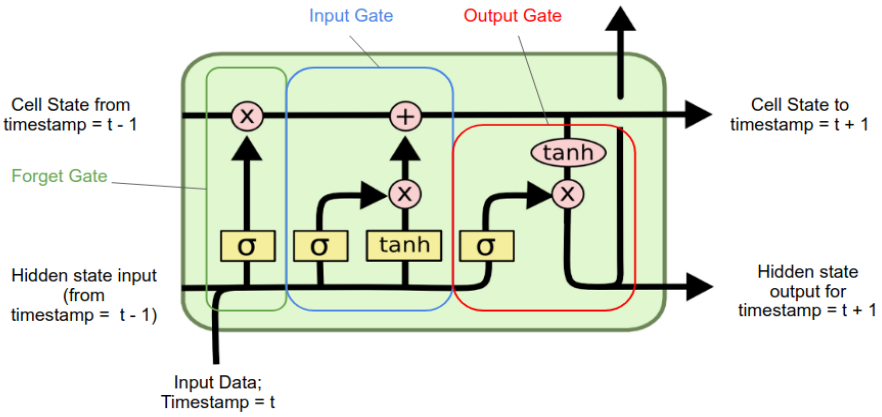


Figure 40: LSTM Cell

Assuming the input at time step t is denoted as x(t), the hidden state at time step t-1 is denoted as h(t-1), and the cell state at time step t-1 is denoted as C(t-1) and U, W are the weights, the equations for the LSTM model are as follows:

$$InputGate : i_t = \sigma(x_t.U^i + h_{t-1}.W^i) \tag{14}$$

$$ForgetGate : f_t = \sigma(x_t.U^f + h_{t-1}.W^f) \tag{15}$$

$$OutputGate : o_t = \sigma(x_o.U^o + h_{t-1}.W^o) \tag{16}$$

LSTMs were developed to deal with the vanishing gradient problem that was encountered when training traditional RNNs. LSTM solves this problem by introducing a set of gating mechanisms that regulate the flow of information through the network. These gates are designed to selectively remember or forget information from previous time steps, allowing the network to maintain

information over longer periods of time and prevent the gradients from vanishing. In this way, the LSTM can selectively retain or discard information over longer periods of time, allowing it to learn long-term dependencies and solving the vanishing gradient problem.

6.5 GRU Architecture

Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture that was designed to overcome the vanishing gradient problem that occurs in traditional RNNs, which can make it difficult to learn long-term dependencies in sequential data. The basic architecture of a GRU consists of a series of recurrent units, each of which contains a set of gates that control the flow of information through the unit. The gates are similar to those used in LSTM networks, but the GRU has fewer parameters, making it computationally less expensive.

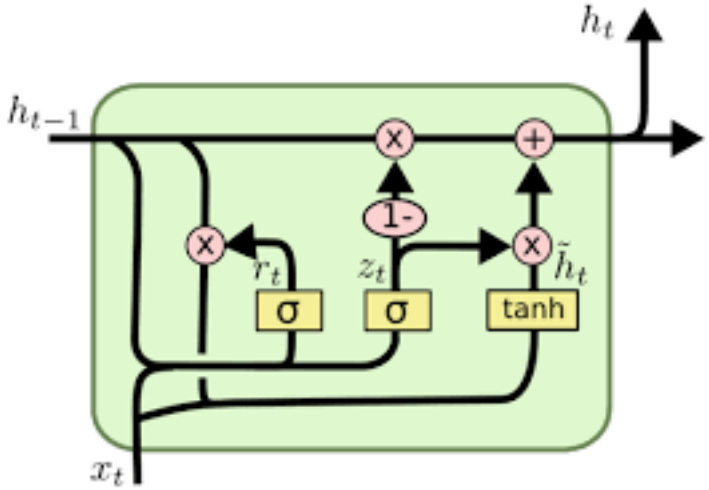


Figure 41: GRU Unit

$$ResetGate : r_t = \sigma(x_t.U^r + h_{t-1}.W^r) \tag{17}$$

$$UpdateGate : z_t = \sigma(x_t.U^z + h_{t-1}.W^z) \tag{18}$$

GRUs solve the vanishing gradient problem by using gating mechanisms that selectively allow information to flow through the network. Specifically, each recurrent unit in a GRU has two gates: a reset gate and an update gate. The reset gate allows the network to decide how much of the past information to forget, while the update gate allows the network to decide how much of the new information to incorporate into the current state. This gating mechanism

allows the network to selectively remember or forget information over time, which helps to prevent the gradients from becoming too small or too large during backpropagation. In addition, GRUs also use a hidden state vector to store and propagate information over time. The hidden state vector is updated at each time step based on the input and the previous hidden state, which allows the network to maintain a memory of the past inputs.

6.6 Learning Algorithm

Batch Gradient Decent was used as the Learning Algorithm

6.7 Optimizers on top of Learning Algorithm

ADAM Optimizer was used for updating the parameters.

6.8 Task1: Charecter Level Language Model

In this task I tried Sequence Labelling to predict the next character in the sequence.

6.8.1 Dataset

The Dataset used was a text file which had a collection of 50000 meaningful Names.

6.8.2 Loss Function

Loss Function used is NLLLoss.

6.8.3 Model Architecture:

```
RNNModel(
  (i2h): Linear(in_features=187, out_features=128, bias=True)
  (i2o): Linear(in_features=187, out_features=59, bias=True)
  (o2o): Linear(in_features=187, out_features=59, bias=True)
  (dropout): Dropout(p=0.1, inplace=False)
  (softmax): LogSoftmax(dim=1)
)
```

6.8.4 Plots:

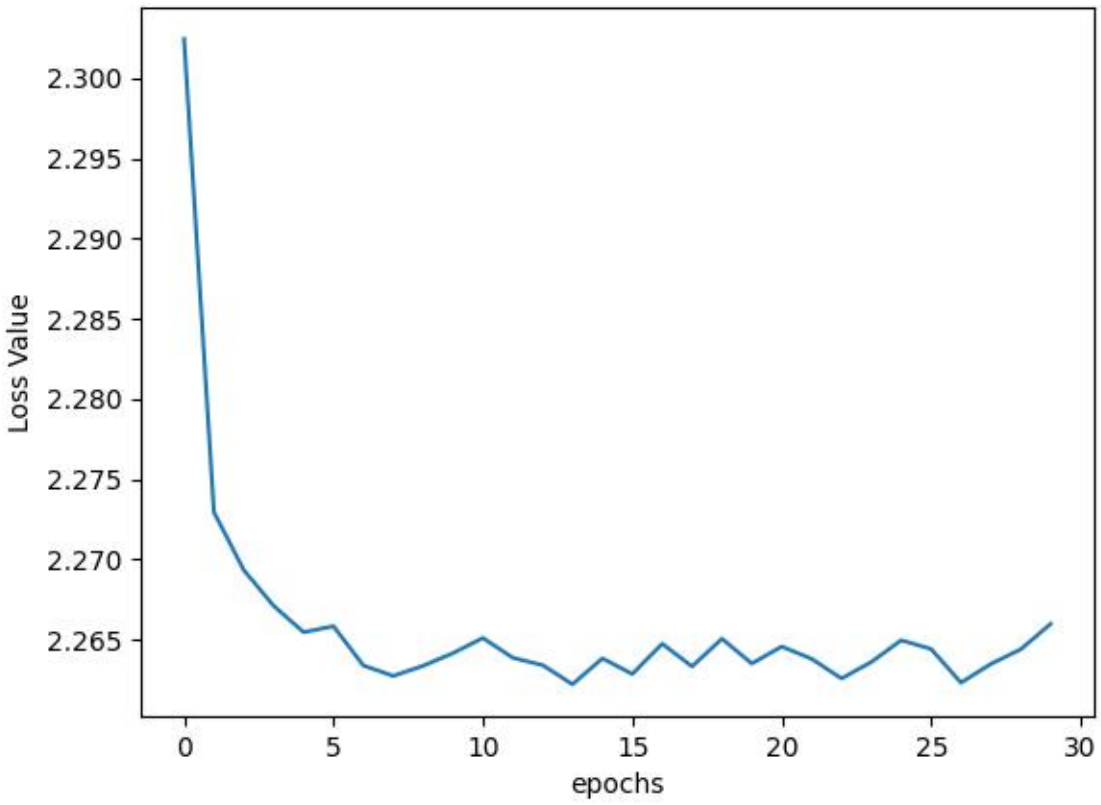


Figure 42: Loss Plot

6.8.5 Name Genrated Before Training and After Training:

Before Training:

- SI,I,I,I,I,I,I,,I,I,I
- AI,I,I,I,I,,ZII,I,,I,
- C,I,II,I,I,I,II,I,I,I

After Training:

- Sherter

- Panere
- Anter

As we can see the Model has started giving somewhat meaningful names.

6.9 Task2: Word Level Language Model

This task is an example of Sequence Classification.

6.9.1 Dataset

Two Datasets were used, one was Hugging face tiny_shakespeare Dataset and the other one was Hugging face wikitext Dataset.

6.9.2 Loss Function

Two Models were Implemented one using CrossEntropy Loss and the other one using NLLLoss.

6.10 Model Architectures:

WordModel1:

```
MyLM(
  (embeddings): Embedding(21949, 256)
  (backbone): LSTM(256, 256, num_layers=3)
)
```

WordModel2:

```
LSTM(
  (embedding): Embedding(29473, 1024)
  (lstm): LSTM(1024, 1024, num_layers=2, batch_first=True, dropout=0.65)
  (dropout): Dropout(p=0.65, inplace=False)
  (fc): Linear(in_features=1024, out_features=29473, bias=True)
)
```

6.10.1 Plots:

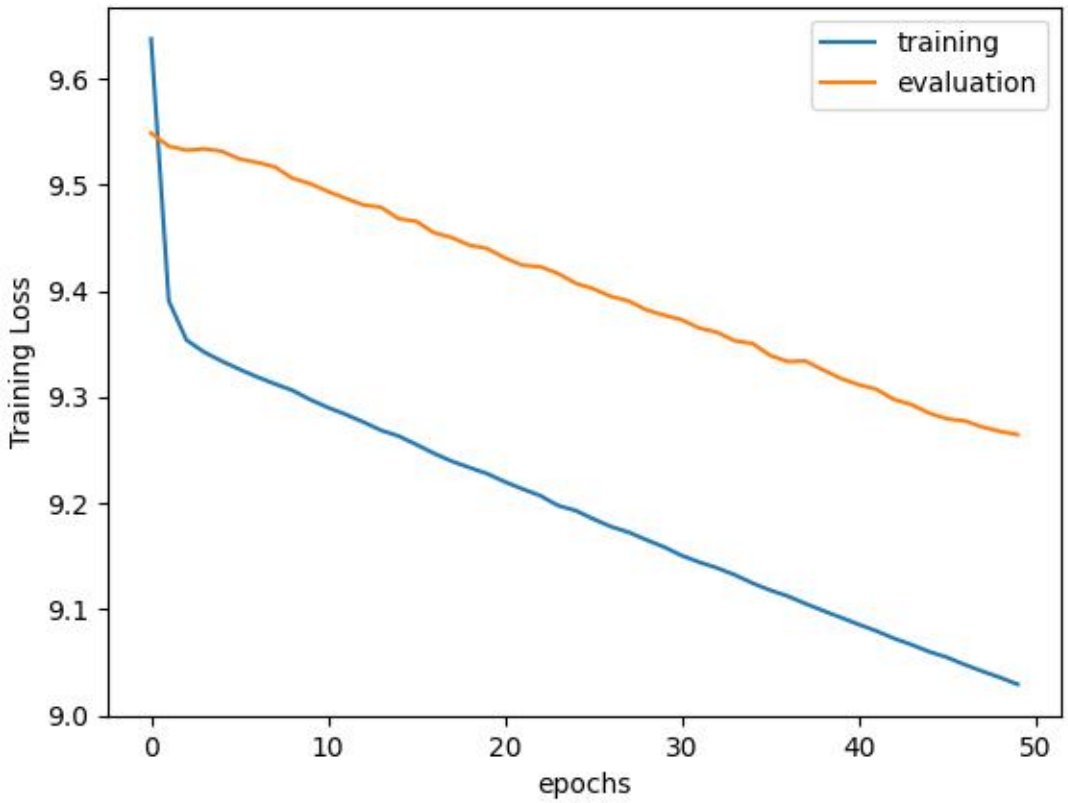


Figure 43: WordModel1 Loss Plot

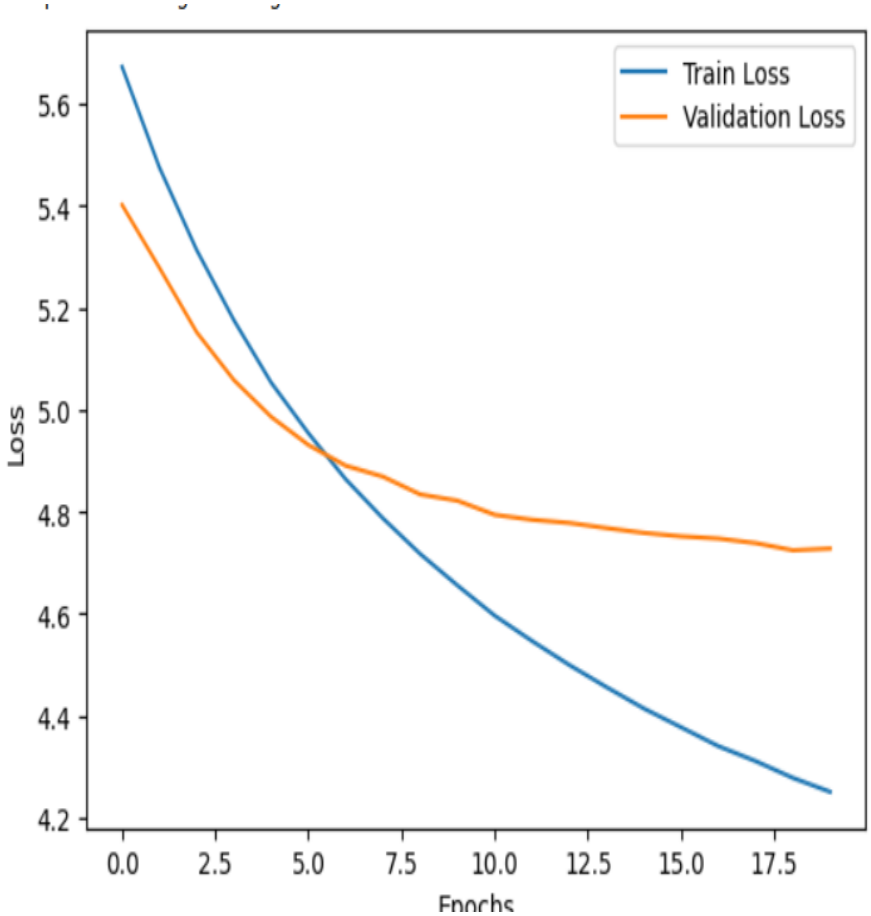


Figure 44: WordModel2 Loss Plot

6.10.2 Comparing Inferences:

Both the models were ran by giving the input statement "think about" and asked to finish the statement,

WordModel1: think about his friends , though his father lives it , a male with the friend , as the death of her own mother , is not a political . the following

WordModel2: think about his own life .

6.10.3 Important Things Observed

From the experiments done for other tasks mainly for this task, some major take aways are:

- Tockenizing, Removing of Unnesecary characters, deciding embedding layer size or latent space dimension plays an important role in the learning of the model. They are as important as other hyper parameters of the model.

6.11 Task3: Classifying movie reviews into Positive/negative

This task is an example of Sequence Classification Task.

6.11.1 Dataset:

The Dataset used for this task is IMDB Movie Review dataset. We process the data by removing stop words and some unnesecary charecters. The IMDB had almost equal number of Positive and Negative class Instances as shown in the below image.

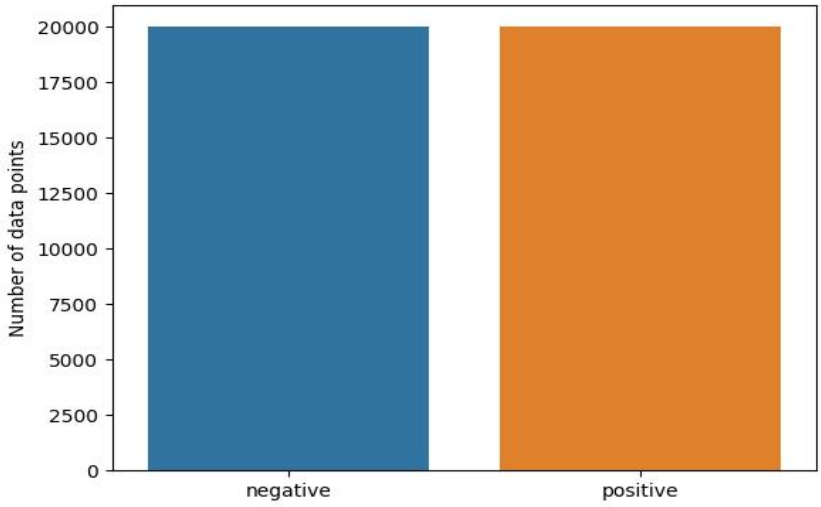


Figure 45: IMDB Movie Review Dataset

- Training Data: 40000
- Validation Data: 10000
- Vocabulary Size: 1000

A short description on how reviews look in the dataset,

- One of the other reviewers has mentioned that ... positive

- A wonderful little production.

The... positive
- I thought this was a wonderful way to spend ti... positive
- Basically there's a family where a little boy ... negative
- Petter Mattei's "Love in the Time of Money" is... positive

6.11.2 Model Architecture:

I have used LSTM Architecture to solve this task. The layered architecture of the LSTM model is described below,

```
SentimentRNN(
  (embedding): Embedding(1001, 64)
  (lstm): LSTM(64, 256, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (actv): Tanh()
  (sig): Sigmoid()
)
```

6.11.3 Loss Function

Loss Function Used for the above task is Binary Cross Entropy Loss.

6.11.4 Evaluation

Accuracy was used as the Evaluation Metric for the above task.

6.11.5 Plots and Tables:

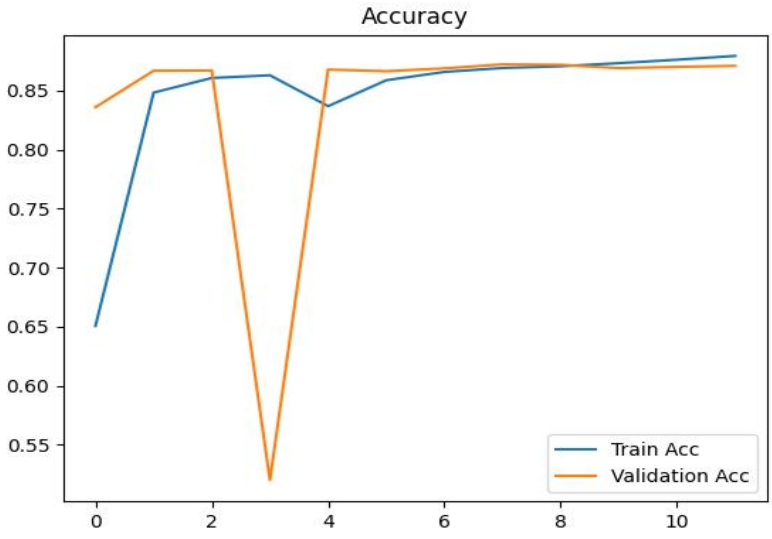


Figure 46: LSTM Model Acc

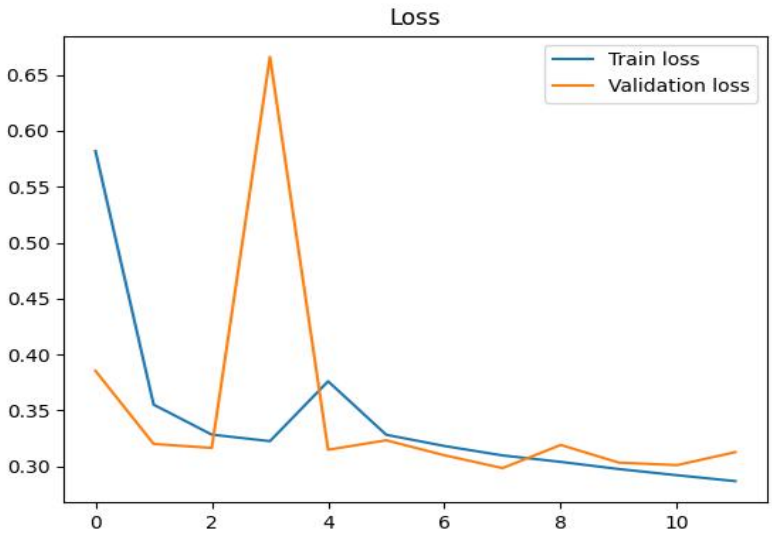


Figure 47: LSTM Model Loss

Test Accuracy	Validation Accuracy	Epochs
89.1125	86.2	13

Table 8: LSTM Model for Sentiment Analysis on IMDB dataset

6.12 Task4: Language Translation from French to English

This task is a classic example of Sequence Translation.

6.12.1 Dataset

The Dataset that was used for the particular task is a text file containing translations of different words and tiny sentences from English to French. each datapoint is a pair of a sentence in french and it's translation to English. For Example,

('ce sont deux choses absolument differentes .', 'they re two very different things .') is one translation pair.

We have trimmed the dataset from 135842 to 10599 sentence pairs, the vocabulary size of the languages French and English are 4345 and 2803 respectively.

6.12.2 Loss Function

Loss Function Used for this task is NLLLoss.

6.13 Model Architecture:

GRU:

```
EncoderRNN(
  (embedding): Embedding(4345, 256)
  (gru): GRU(256, 256)
)
DecoderRNN(
  (embedding): Embedding(2803, 256)
  (gru): GRU(256, 256)
  (out): Linear(in_features=256, out_features=2803, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

LSTM:

```
EncoderRNN(
  (embedding): Embedding(4345, 256)
  (lstm): LSTM(256, 256)
)
```

```
DecoderRNN(
  (embedding): Embedding(2803, 256)
  (lstm): LSTM(256, 256)
  (out): Linear(in_features=256, out_features=2803, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

6.13.1 Model Outputs vs Actual Outputs

For GRU Model,

- – French Sentence: je me rejouis de l entendre .
  – Actual Output: i m glad to hear that .
  – Predicted Output: i m glad to hear that . <EOS>
- – French Sentence: il est professeur de biologie a harvard .
  – Actual Output: he s a professor of biology at harvard .
  – Predicted Output: he s a bit of of at the . <EOS>
- – French Sentence: c est un puissant sorcier .
  – Actual Output: he s a powerful sorcerer .
  – Predicted Output: he s a powerful sorcerer . <EOS>

For LSTM Model,

- – French Sentence: ils bronzent autour de la piscine .
  – Actual Output: they re sunbathing around the pool .
  – Predicted Output: they re all who me . <EOS>
- – French Sentence: tu es tres genereux .
  – Actual Output: you re very generous .
  – Predicted Output: you re very . . <EOS>
- – French Sentence: vous etes inquietes n est ce pas ?
  – Actual Output: you re worried aren t you ?
  – Predicted Output: you re not supposed to as here . <EOS>

6.14 Plots:

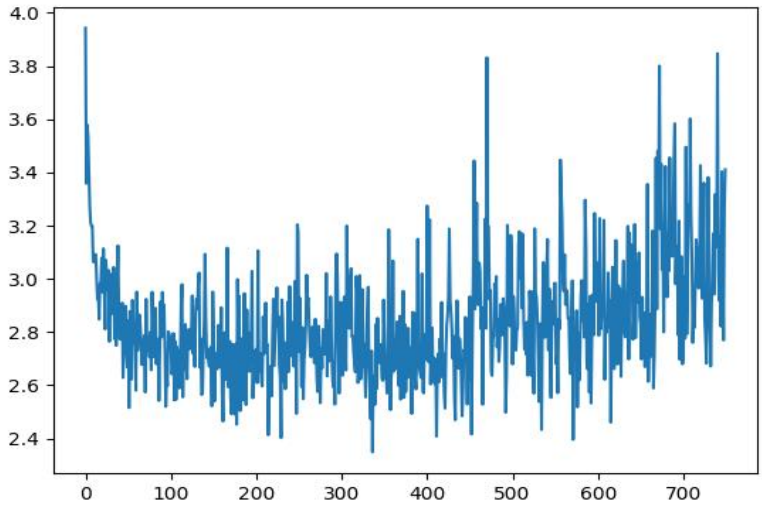


Figure 48: LSTM Model Loss

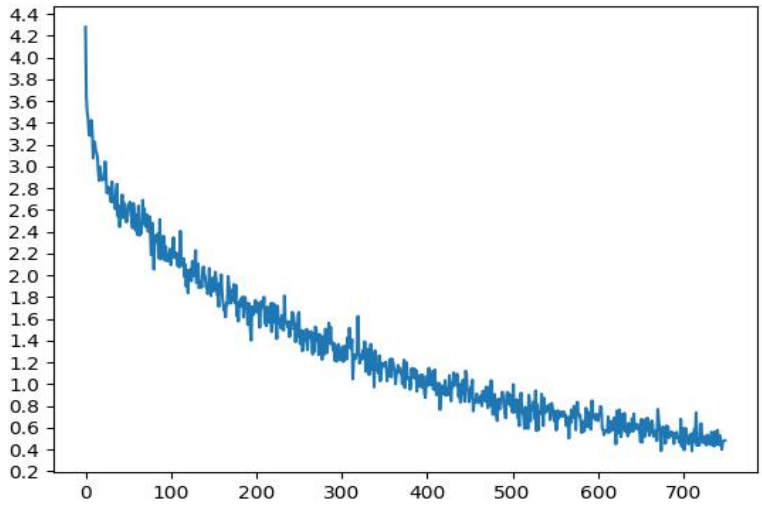


Figure 49: GRU Model Loss

References

[1] Ya Le and Xuan S. Yang. Tiny imagenet visual recognition challenge. 2015.

[2] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29:1–98, 06 2017.