Project-2 of "Neural Network and Deep Learning"

Haowen Shen

May 16, 2023

Abstract

First part, I build a simple Conv2d network and implement some beneficial modifications based on it. Also I conduct some investigation on the network architecture and training process, ultimately my best model achieves a 96.68% test accuracy and CrossEntropy test loss reduced to 0.0012. Second, by researching into the BatchNorm algorithm, I verify its benefits on faster training, reducing the loss variance and allowing a wider range of learning rate without compromising the training convergence.

1 Train a Network on CIFAR-10

CIFAR-10 is a widely used dataset for visual recognition task. The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32×32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks (as shown in Figure 1). There are 6,000 images of each class. Since the images in CIFAR-10 are low-resolution (32×32), this dataset can allow us to quickly try our models to see whether it works.
Here are the classes in the dataset, as well as 10 random images from each:

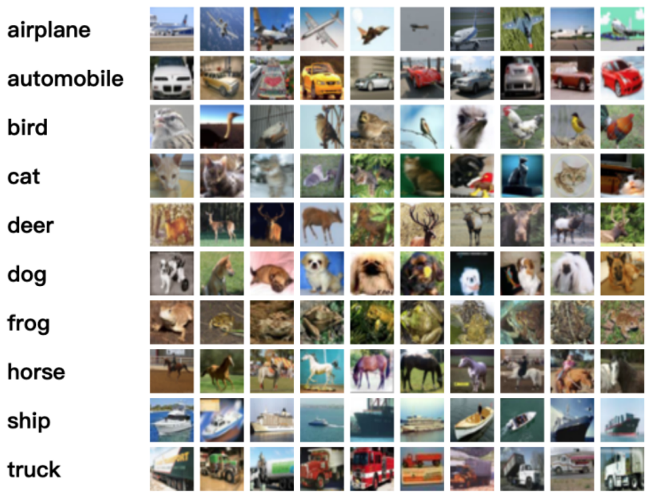


Figure 1: CIFAR-10 Dataset

1.1 Getting started

To start with, I first build a basic Conv2d network from scratch. This network contains the primitive structure of Conv2d layer, Maxpool2d layer, RELU and three Fully-connected layers in the end of the network. The visualization of network structure using netron is displayed in the appendix. Detailed code is attached to the cifar10 raw.py file.
For this basic Conv2d network, it reached an accuracy of 71% with 6 epochs. Hyperparameters in detail are shown below.

Hyperparameters	Value
epoch	42
batch size	4
learning rate	0.001
weight decay	0
activation function	RELU
optimizer	SGD(with momentum=0.9)
data augmentation	no data augmentation
dropout probability	no dropout

Table 1: Detailed Hyperparameters

1.2 Increasing batch size

First, I tried to change the batch size to test the speed of training process and the accuracy under different batch size. It can be seen from the table below that with the increase of batch size, the training time of one epoch declines to a certain level and then equalize. Also I found that more epochs are needed for large batch size. Under 6 epochs, with 128 batch size the test accuracy was only 64%. After about 20 epochs, the network converged and reached an accuracy of 73%. For the batch size of 256, it is harder for the network to converge, as a result of which we should either increase the number of epochs or increase the learning rate when increasing the batch size.

Batch size	training time	Best Accuracy
4	68.856s	71%(under 6 epochs)
128	32.288s	73%(under 20 epochs)
256	30.678s	73%(under 50 epochs)

Table 2: Different batch size

1.3 Implementing dropout

I implemented dropout after every linear layer of the basic Conv2d network. However, the speed of loss decrease slowed down on the contrary. This may be due to excessive regularization, which leads to underfitting of the model. Since the basic Conv2d model doesn't have huge amount of parameters, regularization might affect the network's ability to converge. I will give it a try later in the VGG network.

1.4 Implementing Residual Connection

Residual connection is very useful when dealing with deep neural networks. It can help solve the problem of deterioration of learning ability when the network is too deep. In this project, I tried to use the structure of Resnet in dealing with CIFAR-10. Code of Resnet is added in the attachment RESNET.py and CIFAR-10.py files.
After 20 epochs, ResNet achieved the best test accuracy of 83.31% and test lost of 0.0024. I conducted a 200 round epochs as well, the best test accuracy reached 96.68% and the test lost reduced to 0.0012, which are pretty promising.

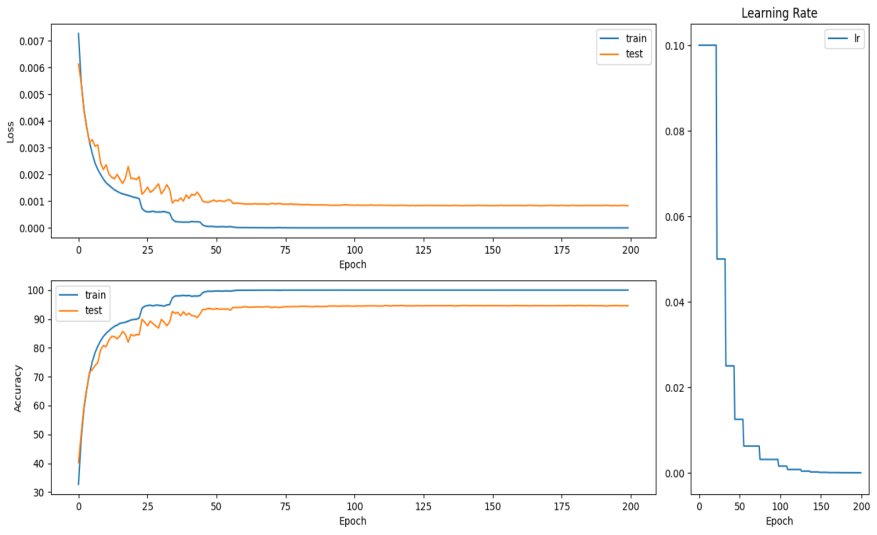


Figure 2: Performance of ResNet

1.5 Try different number of neurons/filters

Here I compared the performance and processing speed of different ResNet structure, including ResNet18, ResNet50, ResNet101. The detailed code is attached to the multi_resnet.py file.

number of layers	number of parameters	processing time for one epoch
ResNet 18	11.174M	23.025s
ResNet 34	21.282M	73.043s
ResNet 50	23.521M	84.950s
ResNet 101	42.513M	135.262s
ResNet 152	58.157M	248.403s

Table 3: Parameters and running time of different ResNet

I also draw the train test loss and accuracy of different ResNet. It can be seen from the image that ResNet18 has a faster speed of convergence in both train and test dataset. However, ResNet50 seemed to have a poor performance in the test dataset where big fluctuations occur.

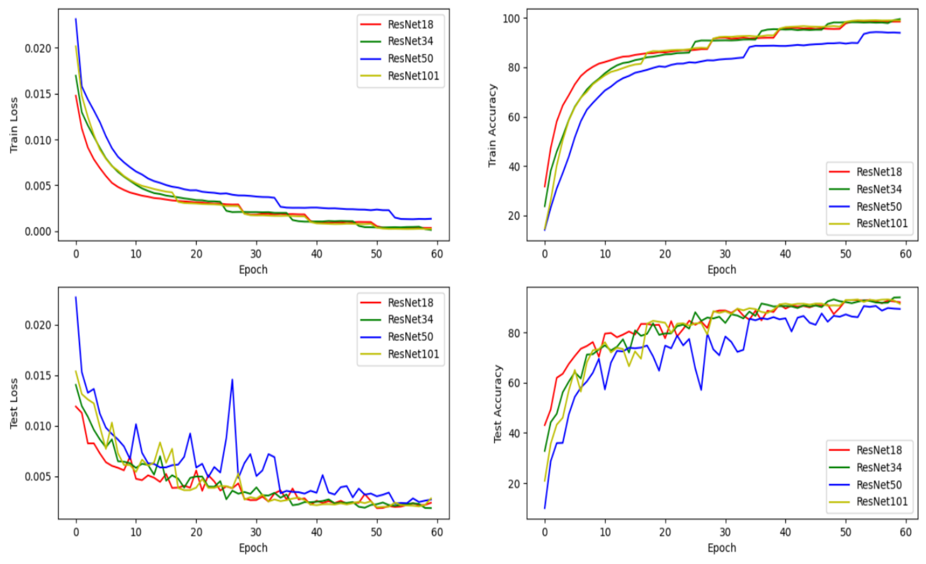


Figure 3: Performance of different ResNet

1.6 Try different loss functions

Here I implemented loss functions of "MSELoss", "CrossEntropyLoss", "SmoothL1Loss", "BCEWithLogitsLoss" and printed the movement of loss and accuracy. We can get from the figure that except for the CrossEntropyLoss, the loss of other loss functions all converged close to zero. However, CrossEntropyLoss outperforms other loss functions in both train and test accuracy. Detailed code is in the multi_loss.py file

I also found that when using big learning rate like 0.1, both train and test loss would become NAN. I suppose exploding gradients is the cause of the occurrence of NAN. The image displaying the change of loss and accuracy is shown below.

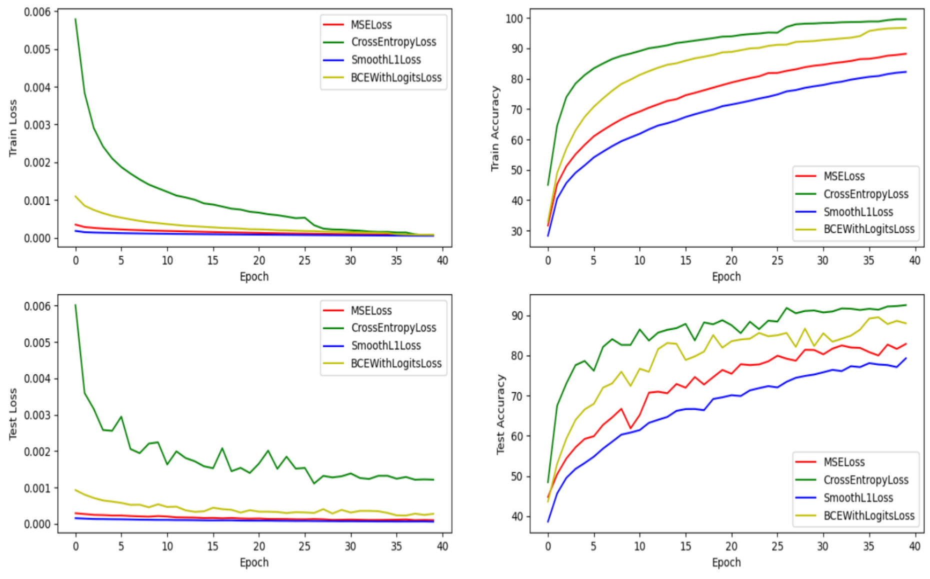


Figure 4: Performance of different loss functions

1.7 Try different activations

It is common to use RELU as activation. Here I compared "ReLU", "Sigmoid", 'LeakyReLU', 'Tanh', 'Softplus' and ploted the change of their loss and accuracy releatively in 50 epochs. It can be seen from the image that sigmoid and softplus activation have a great oscillation in the test loss and accuracy. Among the relu family, relu and leakyrelu have similar performance, and overtakes other activations in the form of convergence and stability. Detailed code is in the Activation_figure.py file.

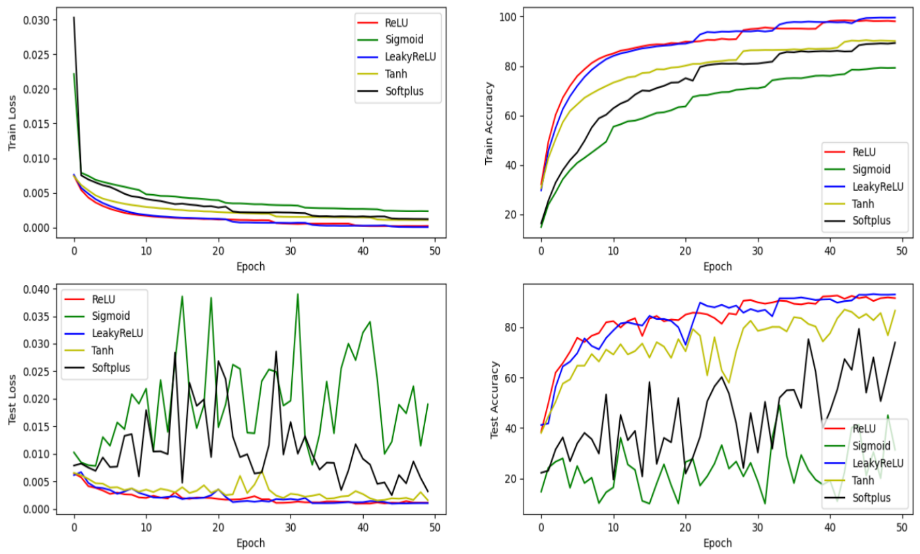


Figure 5: Performance of different activations

1.8 Try different optimizers using torch.optim

Here I tried different optimizers, including SGD, Adagrad, Adadelta, Adam. Adadelta has a good performance especially in the training dataset. It outperforms other optimizers in both train loss and accuracy. Meanwhile, SGD and Adam have a more stable performance in the test dataset. Detailed code is in the multi_optim.py file.

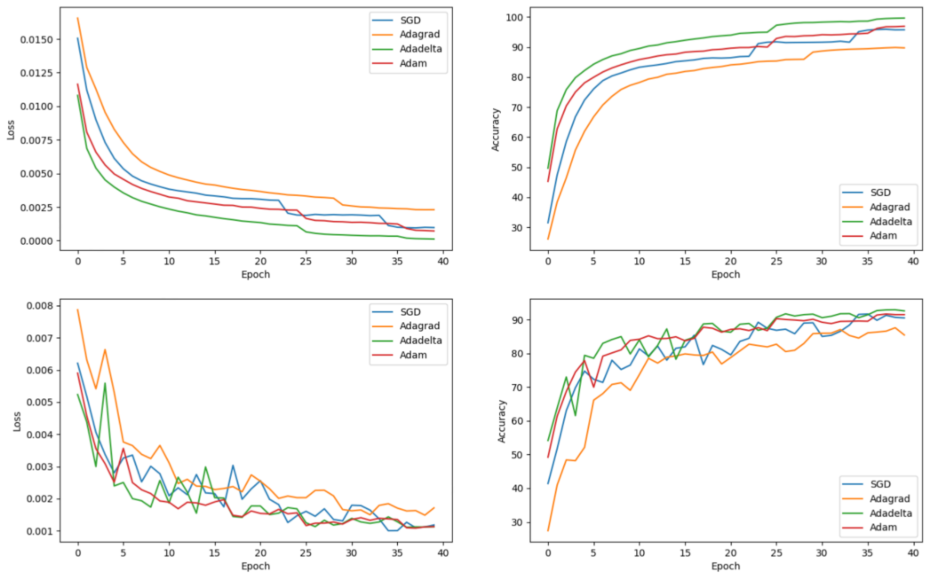
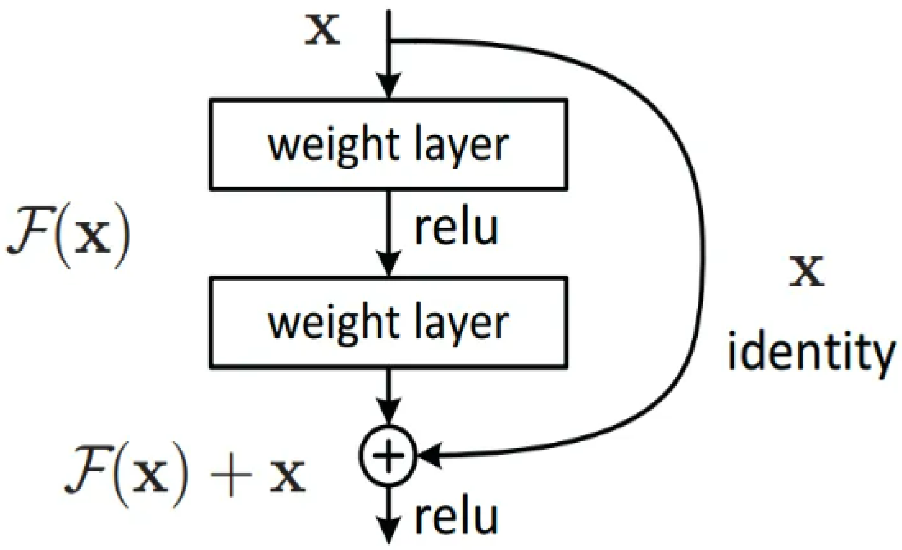


Figure 6: Performance of different optimizers

1.9 Network interpretation

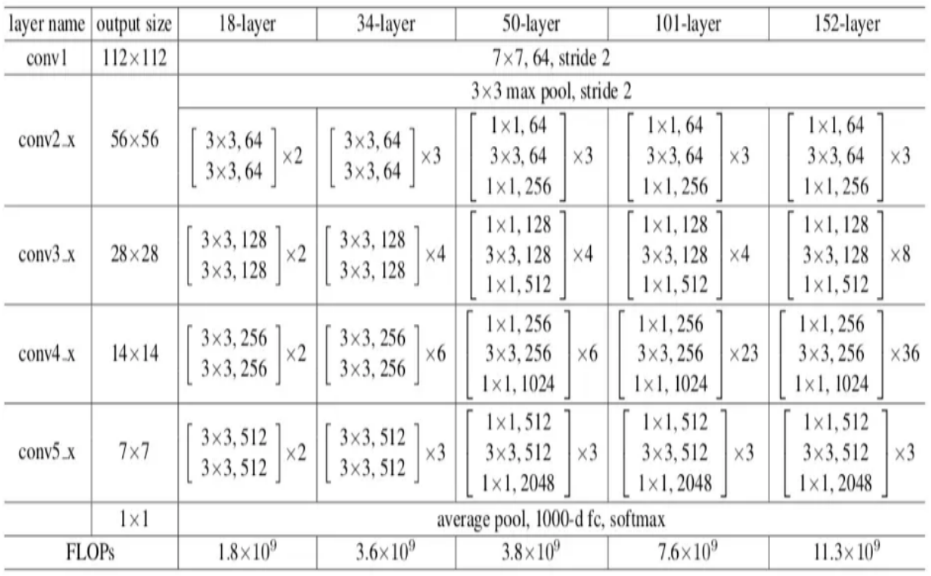
It is famous of ResNets for solving the problems of vanishing gradient. When the network is too deep, the gradients from where the loss function is calculated easily shrink to zero after several applications of the chain rule. This result on the weights never updating its values and therefore, no learning is being performed.
With ResNets, the gradients can flow directly through the skip connections backwards from later layers to initial filters.



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 7: Residual Connection

In the table, there is a summary of the output size at every layer and the dimension of the convolutional kernels at every point in the structure.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 8: Sizes of outputs and convolutional kernels for different ResNet

Now we visualize the BasicBlock in the ResNeT. We are using 2 [3x3, 64] kernel and both the input and output size are [56x56]. Here we visualize the filter as Figure 5 illustrates.

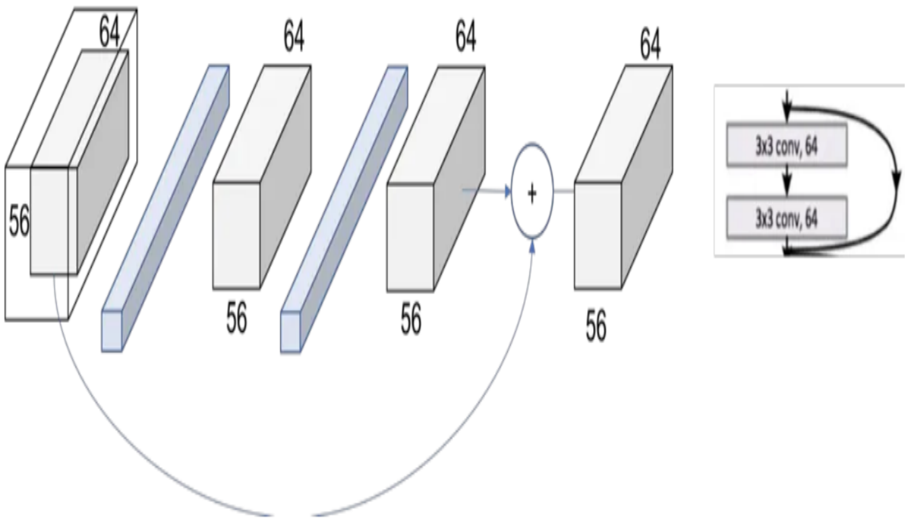


Figure 9: 2 [3x3, 64] kernel and residual connection

The same procedure can be expanded to the entire layer. Now, we can completely replicate ResNet34 at Conv2_x layer as follows. Also, the visualization of Resnet18 structure using netron is displayed in the appendix.

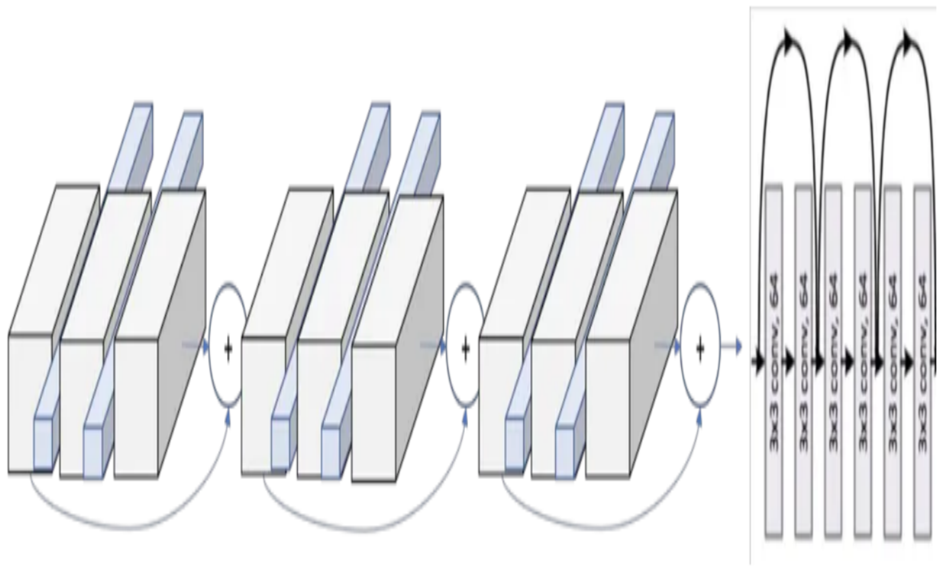


Figure 10: Visualization of Conv2_x layer in ResNet34

2 Batch Normalization

Batch-Normalization (BN) is an algorithmic method which makes the training of Deep Neural Networks (DNN) faster and more stable.
It consists of normalizing activation vectors from hidden layers using the first and the second statistical moments (mean and variance) of the current batch. This normalization step is applied right before (or right after) the nonlinear function.

2.1 VGG-A with and without BN

Here I compared different VGG-A network structures. the VGG-A Light network contains only two stages compared to five stages in VGG-A. Also I implemented dropout and batch normalization relatively to test their performance. It can be seen from the image that since VGG-A Light have much fewer parameters and less depth than VGG-A, its final performance is relatively poor compared to others. Also, after applying Dropout, we can conclude from the test loss figure that when the number of epochs increase, VGG-A test loss begin to raise while VGG-A with Dropout remains stable. Implementing Dropout does have a benefit against overfitting. Also, I found that when using SGD optimizer, VGG-A loss did not decrease and the fit did not converge, while VGG-A Light and VGG-A with BN could achieve a good performance. So I changed the optimizer as Adam and fit of VGG-A then started to converge. I guess Adam have a better ability in choosing the down trend and controlling the degree of gradient descent. Detailed code is attached in the VGG.py and main.py files.

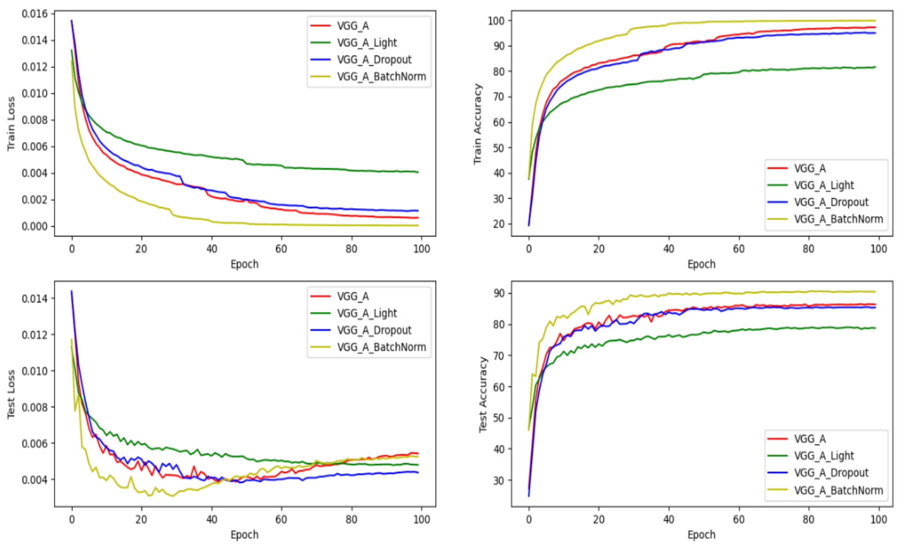


Figure 11: Performance of different VGG-A

With batchnorm, we can see that the speed of both train and test convergence increase impressively. VGG-A with BN outperforms others without doubt. However, we also find that when epochs increase, overfitting occurs in the test loss of VGG-A with BN. So I planed to implement dropout before FC layer in the classifier block of VGG-A with BN. Detailed code of VGG models is attached to VGG.py. The following image displays the performance of VGG-A with BN and VGG-A with BN and dropout. With Dropout, test loss does decrease when epochs raise. The accuracy of both models tends to become the same when fitting converged.

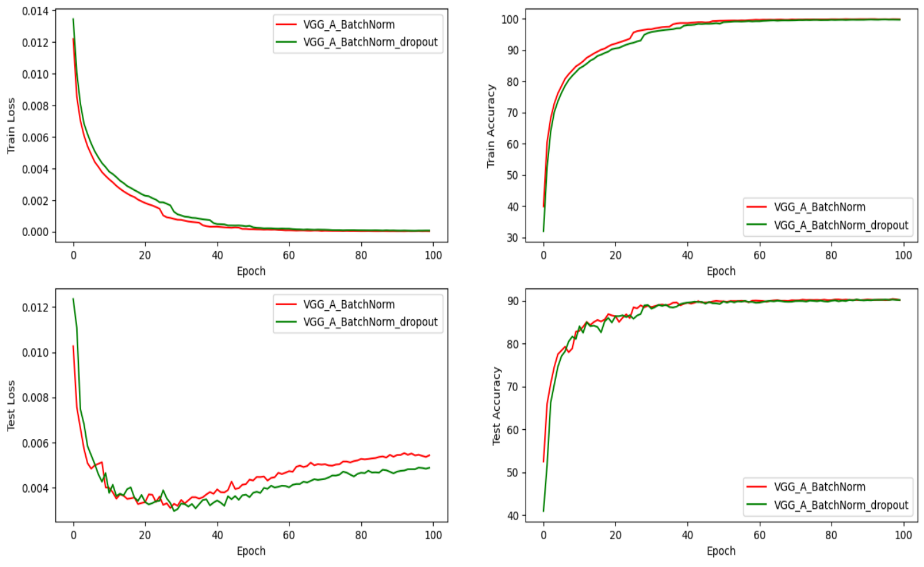


Figure 12: Performance of VGG-A-BN with and without Dropout

2.2 Loss Landscape

Here I set the learning rate to be [1e-3, 2e-3, 1e-4, 5e-4], and select the maximum value of loss in all models on the same step, add it to max-curve, and the minimum value to min-curve. After running 25 epochs I ploted the image below. Detailed code is in the loss landscape.py file.

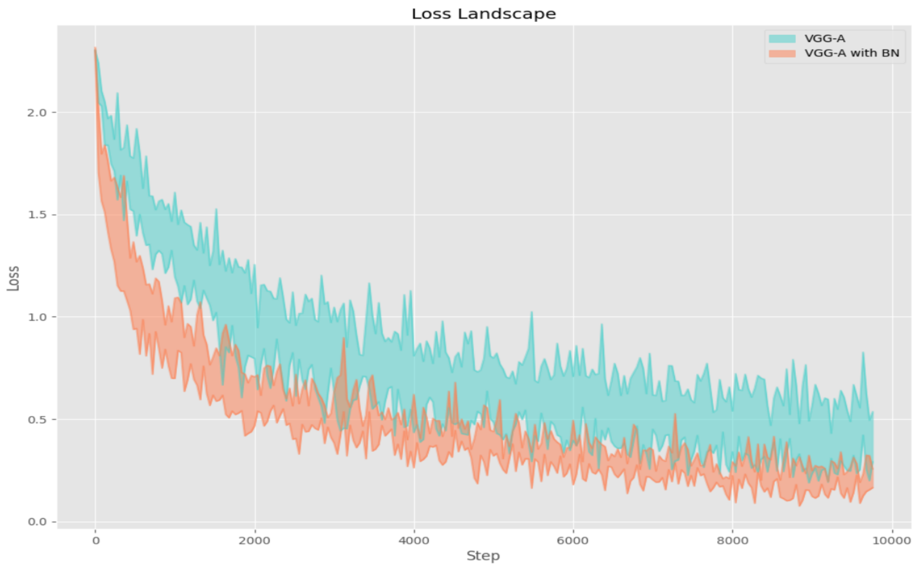


Figure 13: Loss landscape for VGG-A with and without BatchNorm

As we can see in the image, BN layers make the training faster, help reduce the loss variance and allow a wider range of learning rate without compromising the training convergence.

2.3 Change of the loss gradient

Here I choose to record the change of gradient in the last linear layer of the classifier block. The image is shown below.

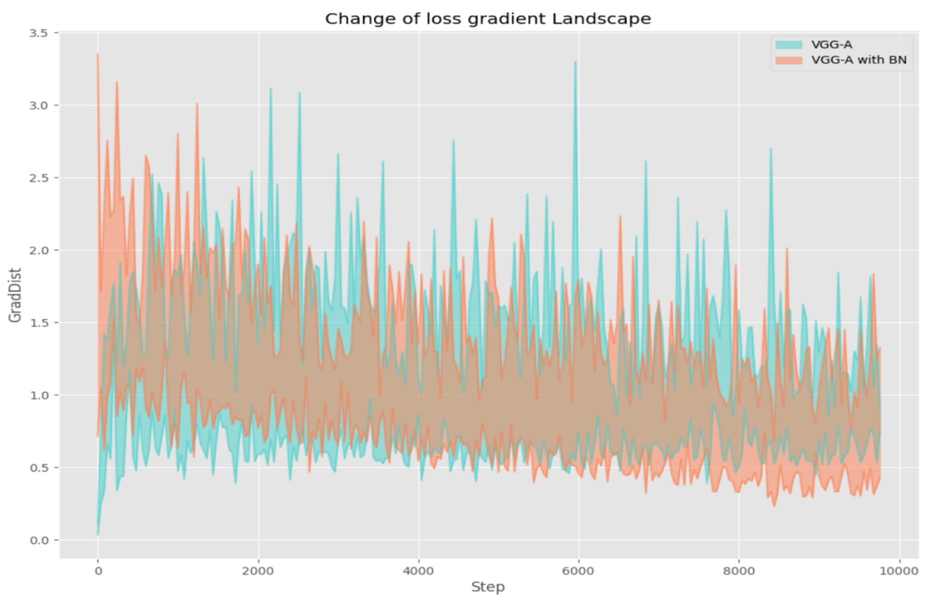


Figure 14: Change of loss gradient for VGG-A with and without BN

As we can see in the figure, the distance of the gradient difference shortens with the help of BN. Also, the training speed is faster.

2.4 Maximum difference in gradient over the distance

Here I test the β smooth of VGG-A with and without BN. β can be illustrate by the following formula:

$$\beta = \max \frac{\|\nabla f(\omega_t + 1) - \nabla f(\omega_t)\|}{\|\omega_t + 1 - \omega_t\|}$$

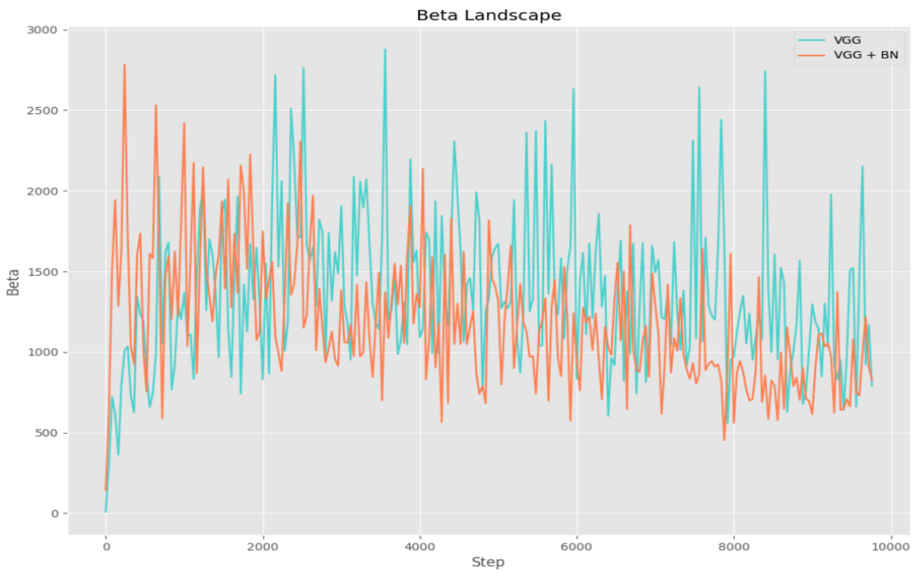


Figure 15: Beta Smoothness for VGG-A with and without BatchNorm

As we can see the maximum difference in gradient over the distance is greatly decreased after applying BN.