To check out the whole project: code, report, images etc. Click CIFAR-10 & BN to the Github repository.

1 Train a Network on CIFAR-10

CIFAR-10 is a widely used benchmark dataset in the field of computer vision and machine learning. It stands for the "Canadian Institute for Advanced Research 10änd was created by researchers at the University of Toronto. The dataset consists of 60,000 color images in a 32x32 pixel format, categorized into 10 different classes. Each class contains 6,000 images.
The 10 classes in CIFAR-10 represent common objects and animals, including airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is balanced, meaning each class has an equal number of images.
CIFAR-10 is often used as a benchmark for training and evaluating machine learning models, particularly for image classification tasks. Researchers and practitioners use this dataset to develop and compare different algorithms, architectures, and techniques for image recognition. The small image size and diverse class categories make CIFAR-10 challenging yet manageable for experimenting with various models and approaches. Due to its popularity, several state-of-the-art models and techniques have been developed using CIFAR-10 as a testbed. The dataset serves as a standard reference for assessing the performance and generalization capabilities of machine learning algorithms in the context of image classification.

1.1 Network Architecture

The general network architecture in our experiments roughly covers everything that the assignment requires. The family of our models can be found in the model module.

1.1.1 Comparison

In Table 1, we listed the different models' scale of parameters and their performances in terms of test accuracy. We run the models for an epoch number of 100 to save some time. But we should keep in mind that 100 epochs is not enough to guarantee convergence with the help of data augmentation, without which the model can easily overfit within 50 epochs.

Model	#Parameters	Train Accuracy (%)	Test Accuracy (%)
ResNet18	11173962	95.32	92.06
ResNeXt50_32x4d	22992714	98.70	**94.85**
DPN26	11574842	97.07	93.26
DLA	16291386	98.22	94.65
DenseNet121	6956298	**98.77**	94.69

Table 1: Model scale and Accuracy

1.1.2 Convergence History

In this part, we show the convergence history (train accuracy, train loss, test accuracy, test loss) for the networks in Table 1. The results are in the following Figure 1, Figure 2, Figure 3, Figure 4.

1.2 Best Result

Due to the limited time and computational resources, we achieve a test error of **5.11%** using the model settings in Table 2. We did **NOT** the combinations of different hyperparameters enough to get the best results. That is to say, although certain hyperparameters may seem well according to general experiments, it may not have an outstanding performance when colloborating with each other.
The best model turns out to be `DenseNet121`. It not only shows great performance, but it also has a small number of parameters. For more details besides Table 2, check out the source code in the project repository.

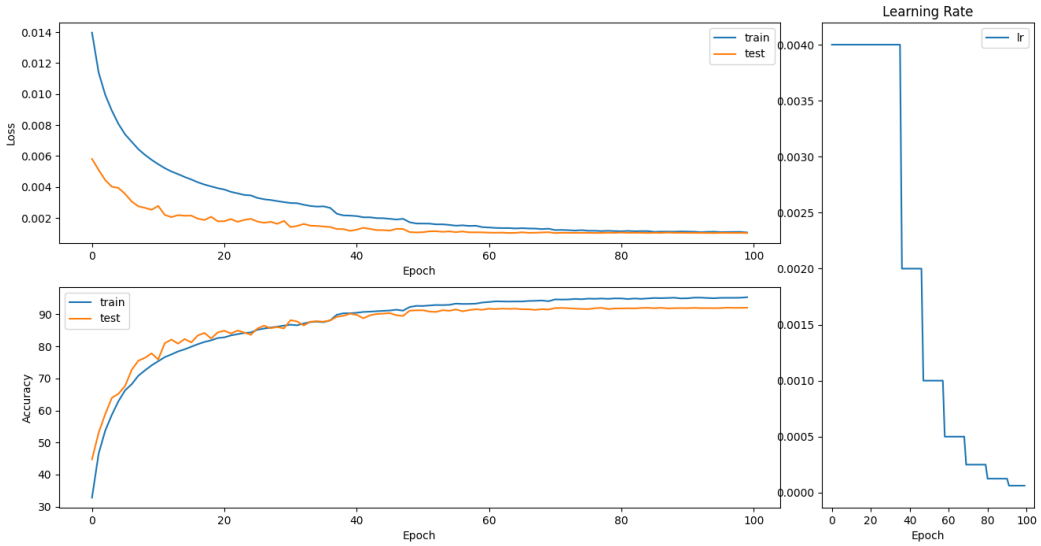

Figure 1: ResNet18

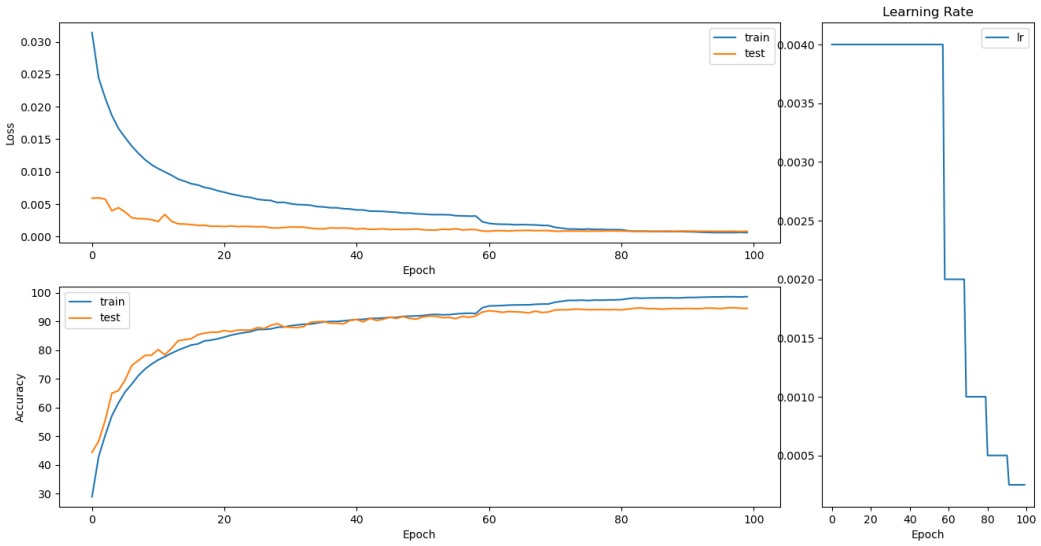

Figure 2: ResNeXt50_32x4d

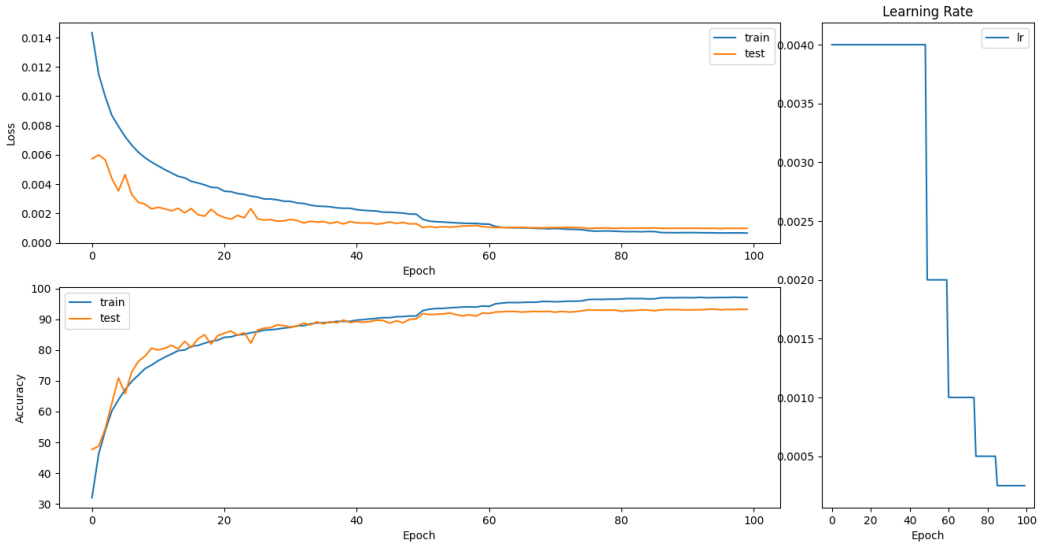

Figure 3: DPN26

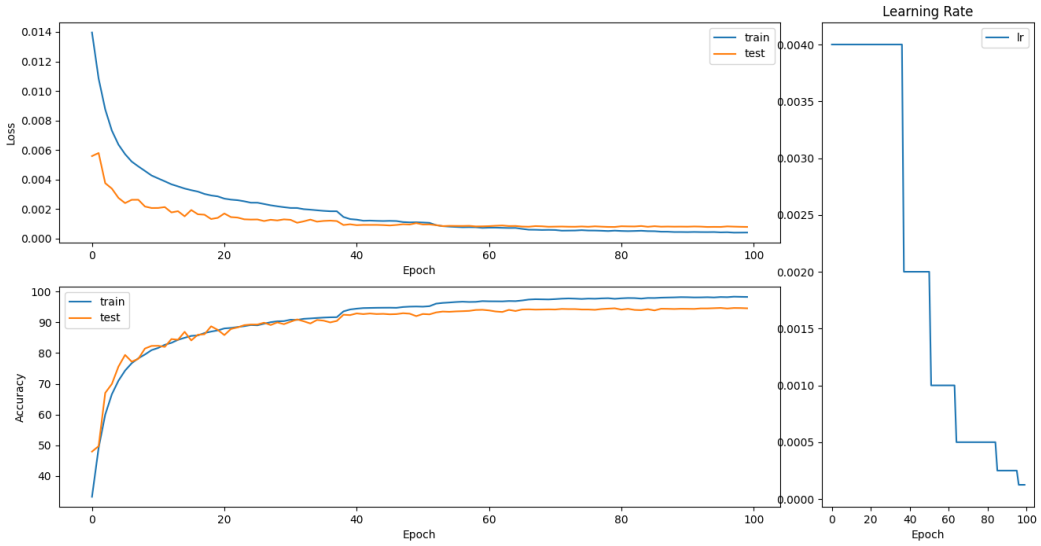

Figure 4: DLA

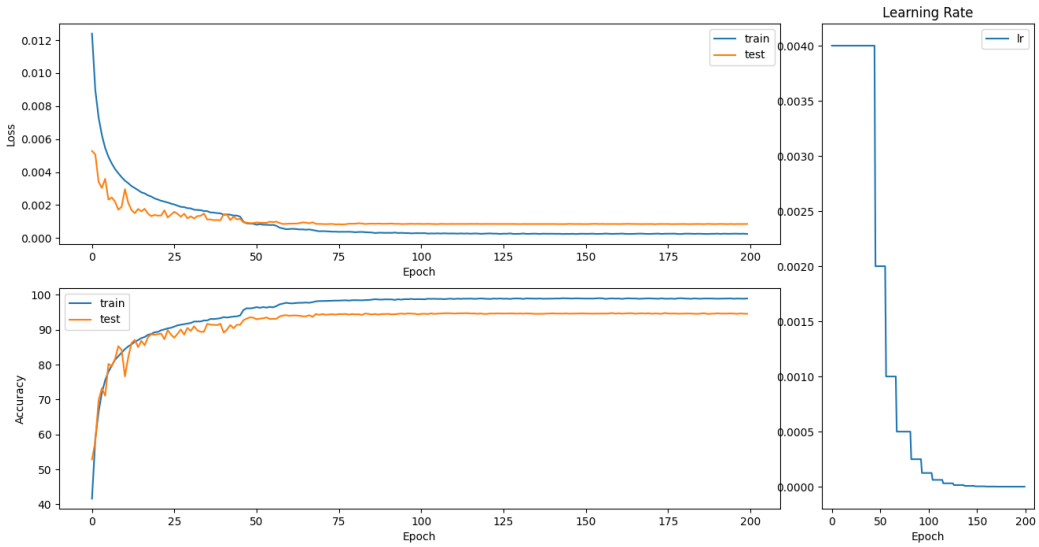

Figure 5: DenseNet121

The convergence history of our best result shown in Figure 5. For a small model like Table 2, achieve a train error of 1% or so along with a test error of 5% or so could be satisfactory.

Setting	Value
Model	DenseNet121
#Parameters	6956298
Data Augmentation	Cutout
Epoch	200
Activation	LeakyReLu
Loss	CrossEntropyLoss
Optimizer	Adam
Train Accuracy (%)	98.96
Test Accuracy (%)	**94.89**

Table 2: Best Model Configuration

1.3 Experiments

In this part, we conducted three major experiments on our choice of different activations, losses and optimizers. In each experiment, we guarantee that the other hyperparameters remains the same. Then we use the same model with the same configurations to run a given number of epochs. Loss and accuracy on both the training set and test set are reported and presented as a figure.
The whole process of the experiments are in `cifar_{quota}.py` in the root directory, where `quota` $\in$ {`activation`, `loss`, `optimizer`}. For more details, check `utils.runner` to see the constructure of the runner. Moreover, for each experiment, we equip it with suitable configurations in `config_{quota}.yaml` in `config` directory.

1.3.1 Activation

The activation functions we select are part of 'ReLu family', including but not limited to

- ReLu,
- ReLu6,
- LeakyReLu,
- ELU,

- SELU,
- PReLU.

Other favorable choices are

- Softplus,
- Sigmoid,
- Softmax
- Tanh,
- Hardtanh.

Our implementation for different activations can be found in `utils.runner.activation`. Our choices are on the top 4 of the former list. The main insight behind this to explore as much functions as we can and save computational resources at the same time. As a matter of fact, the 'ReLu family' is friendly to us. The results are shown in Figure 6. We can conclude that ReLu and LeakyRelu is slightly better than ReLu6 and ELU and LeakyReLu has a tiny edge against ReLu. Although the former one adds a little computation complexity, it effectively avoids the phenomenon of dead neurons. Consequently, we will choose LeakyReLu to zoom the network's capability.

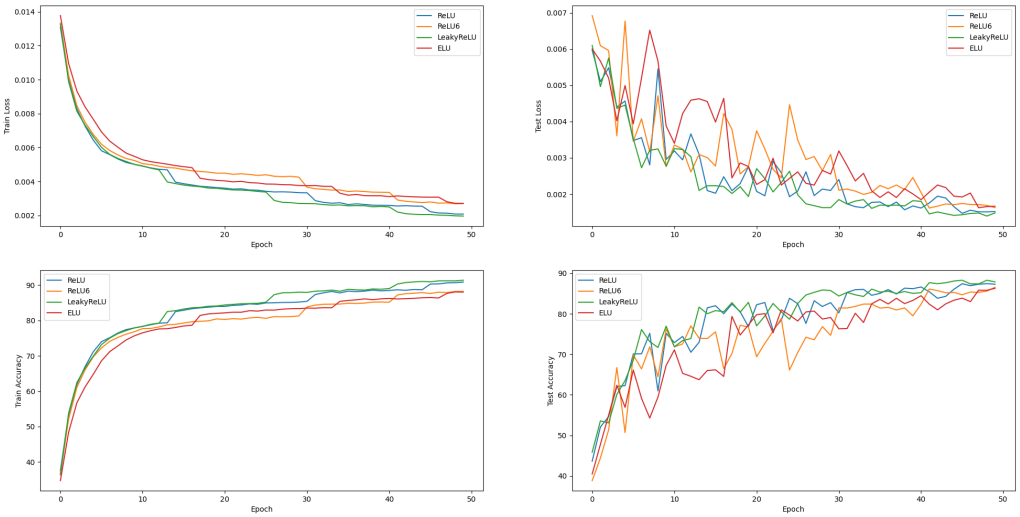

Figure 6: Choice on Activations

1.3.2 Loss

Our implementation for different losses can be found in `utils.runner.loss`. The losses are

1. CrossEntropyLoss,
2. MSELoss,
3. BCEWithLogitsLoss,
4. MultiLabelSoftMarginLoss.

Note that **NOT** all losses are appropriate. Comparing with some bad loss function will also help us gain some insights of the task. Also, we can use different regularizations together with different losses. For simplicity, we will **NOT** show the results here. You can tune and choose different regularizers by passing different parameters to the net initialization process.
The final results are shown in Figure 7. We can conclude that CrossEntropyLoss is the best. The rest is similar to each other in terms of test accuracy. MultiLabelSoftMarginLoss is strongly **NOT** recommended.

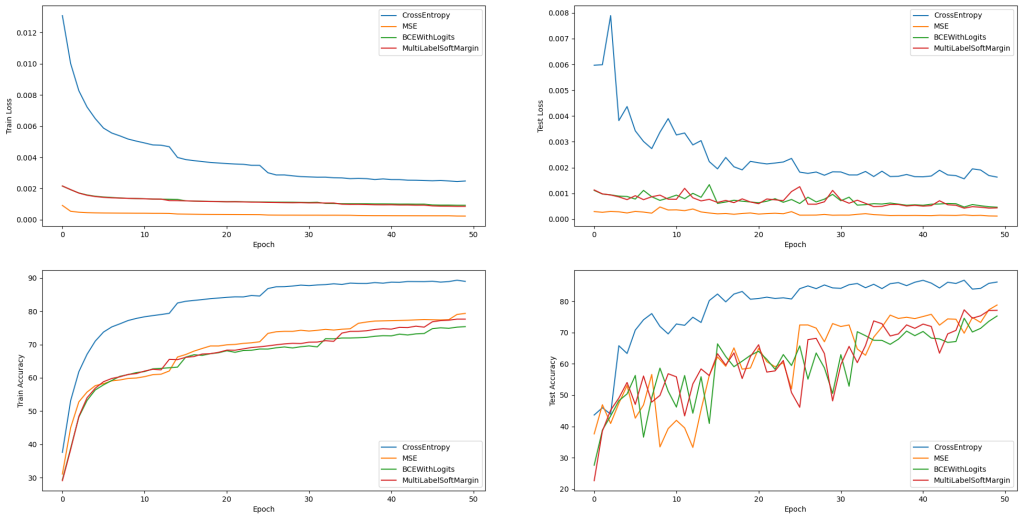

Figure 7: Choice on Losses

1.3.3 Optimizer

Our implementation for different optimizers can be found in `utils.runner.optimizer`. Our choice of the optimizers are

1. Adam,
2. SGD,
3. RMSprop,
4. Adadelta.

To be compatible with our schedulers, second order algorithms like LBFGS will not be used, though it should be interesting to compare its bahavior with some widely used first order optimization methods. The outcomes are in Figure 8. We can conclude that SGD is better than Adam. RMSprop and Adadelta is not a favorable choice for the sake of their bad performance both on training set and test set. Whereas, the number of epochs may not be enough to draw the above conclusions. The network has not converged yet. Moreover, Adam is more pervasive than SGD because it converges faster on large networks like BERT. SGD is more prune to the choice of its parameters[1].

1.4 Other Requirements

So what's more about the network we implemented? Apart from some main functionalities shown in the previous parts, we add some necessary score points of this assignment, which is the components of the network, the strategies used to optimize the network and my insights of the network (visualization of filters, loss landscape, network interpretation).

1.4.1 Components

Full-Connected Layer Of course we have it. Otherwise we will **NOT** be able to finish this task.

2D Convolutional Layer 2D convolution is a basic operation in ResNet and other common image feature extractors. You can find it everywhere in module `model`.

2D Pooling Layer 2D pooling is also a necessary and common operation about image processing. It is used to reduce the spatial dimensions (height and width) of the input image while keeping the depth (number of channels) constant. In the forward process of the basic class `ResNet` we implemented, we did use it.

[1] $\sum_{i=1}^{\infty} \alpha_i = \infty, \quad \sum_{i=1}^{\infty} \alpha_i^2 < \infty$

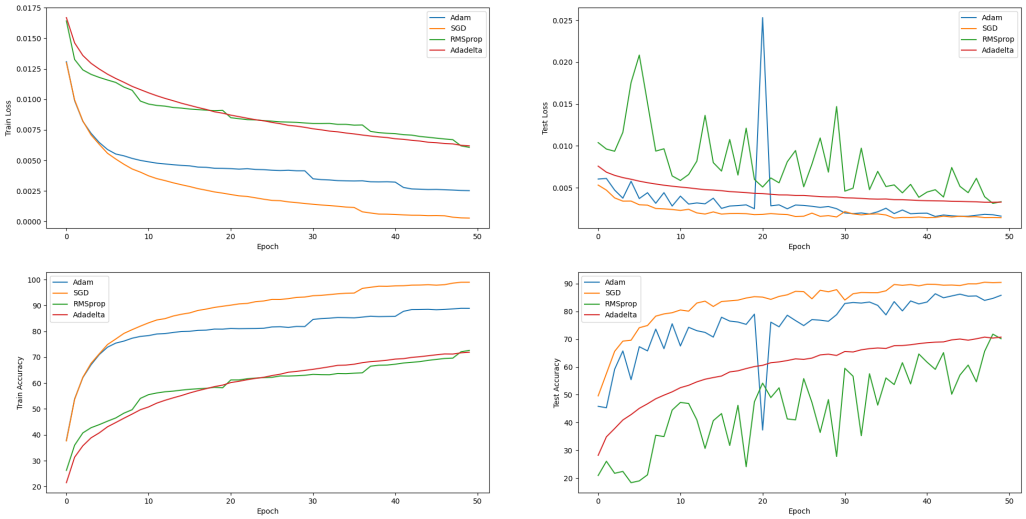

Figure 8: Choice on Optimizers

Activations We already did a major experiment on activations in the previous sections. You can alter the type of the activation by parsing the name of the activation function in our code.

Batch Norm Layer Batch normalization can be found at a number of places in our code. The basic model in ResNet family naturally includes the batch norm layer.

Dropout If you prefer a dropout mechanism to avoid overfitting, simply parse the dropout probability to the network. Note that this will work if and only if you specify the hidden layer structure by parsing a tuple to `hidden` in argparse. Default is an empty tuple.

Residual Connection Needless to say, since our basic model is ResNet, so we naturally include residual connection. It is the main reason of the capability of the model.

Others For example, you can adjust the sizes and numbers of hidden layers. In this way, you may get a better insight on why hidden layers are **NOT** recommended in our case. Other components we focus on are probably a variety of models we select, which covers some famous mainstream image classification network structures.

1.4.2 Strategies

Try different number of neurons/filters The configuration for this strategy not only has something to do with the parameter concerning hidden layers, but also lies in the design of different models. For example, ResNet18 and ResNet34 apparently has different settings in this respect.

Schedulers With the help of `torch.optim`, we are endowed with ease to design our own optimizers. What's more exciting is the fact the optimizers can be matched up with our handmade schedulers. The schedulers can be seen as an overall adjustment for the model in training, as it can alter the learning rate at your disposal during the whole training process.

1.4.3 Insights

Visualization of filters For our filters, it is necessary to know the different emphasis as for the different residual blocks at different depths. Further visualizations could be done to draw the conclusion that a shallower convolutional layer can capture the semantic details of the image object, while a deeper convolutional layer concentrates more on the general shape and abstract information of the image.

Loss landscape If the network facility can make the loss landscape smoother, then the parameters will become easy to learn. Therefore, if we visualize the loss landscape, we can find that the smoother network will have a better performance on the test set, and this type of network usually has multiple batch norm layers.

Network interpretation We can use the gradient of any target class, flowing into the last convolutional layer to produce a coarse localization map highlighting important regions in the original image for predicting the class. In this way we can find out that from an interpretive point of view, networks with a better performance, i.e. a strong capability to generalize between classes, can capture the important regions more precisely.

2 Batch Normalization

Batch Normalization (BatchNorm) is a technique that has been widely adopted in the field of deep neural networks (DNNs). Its application enables faster and more stable training of DNNs. However, there is still a lack of understanding regarding the exact reasons for BatchNorm's effectiveness. While it is commonly believed that BatchNorm controls the change of the layer's input distributions during training, thereby reducing ïnternal covariate shift,"this work demonstrates that such distributional stability of layer inputs has little to do with the success of BatchNorm. Instead, it is revealed that BatchNorm has a more fundamental impact on the training process: it makes the optimization landscape significantly smoother. This smoothness induces a more predictive and stable behavior of the gradients, which ultimately allows for faster training processes.
In this part, we will reproduce the experiments done in NeurIPS 2018 (1805.11604), then prove the above conclusion. Since the requirements only includes the loss landscape or variation of the value of the loss, we will only illustrate that. As for gradient predictiveness or the change of the loss gradient and maximum difference in gradient over the distance, both are included in our code but for simplicity, we won't show the results here. The visualization figures can be found in log directory.
To test the impact of BatchNorm on the stability of the loss itself, i.e., its Lipschitzness, for each given step in the training process, we compute the gradient of the loss at that step and measure how the loss changes as we move in that direction. That is, at a particular training step, measure the variation in loss. We did the following procedure for a simple implementation:

- Select a list of learning rates to represent different step sizes to train and save the model (i.e. [1e-3, 2e-3, 1e-4, 5e-4]);
- Save the training loss of all models for each step;
- Maintain two lists: `max_curve` and `min_curve`, 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`;
- Plot the results of the two lists, and use `matplotlib.pyplot.fill_between` method to fill the area between the two lines.

We conducted the above procedure for VGG-A model with and without BN, and get the visualization result in Figure 9. In the step-loss Figure 9, we plot nearly 10000 steps in total. It's quite straightforward to make a conclusion that BN substantially reduce the loss variance, thus rendering it smoother and more stable. This transform makes the model easier to learn and interpolate.

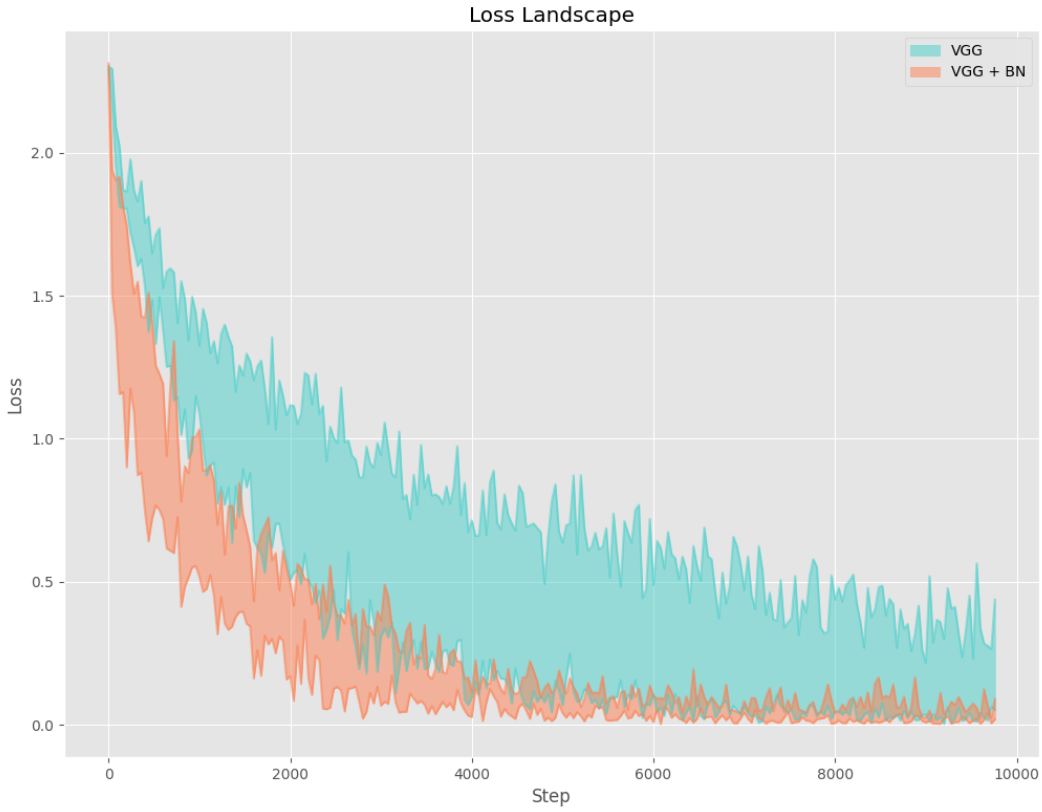

Figure 9: Loss landscape for VGG and VGG_BN