DATA130011.01 Neural Network and Deep Learning Project 2

吴嘉骜 21307130203

2024 年 6 月 20 日

Abstract

This report presents the completion of two tasks for Project 2. Task one involves training neural network models on CIFAR-10 to optimize performance. We use ResNet as the base network and experiment with different hyperparameters and various settings. The best test accuracy we achieve on CIFAR-10 is 95.50% by training ResNet-34 for 200 epochs. Task two involves testing the effectiveness of Batch Normalization. We experiment with VGG-A with and without Batch Normalization on CIFAR-10 and explore its impact on optimization. The report details the training specifics, performance metrics, and visualization results for both tasks.

Project description

The objective of this project is to

1) train neural network models on CIFAR-10 to optimize performance,

2) and analyze how BN help optimization through experiments on VGG-A with and without BN.

Experiment environment

GPU: Tesla V100-PCIE-32GB * 1

CPU: 6 vCPU Intel(R) Xeon(R) Gold 6130

PyTorch 2.0.0 Python 3.8 (ubuntu20.04) Cuda 11.8

1 Task 1: Train a Network on CIFAR-10

1.1 Introduction

In this task, we mainly focus on training **ResNet** models and its variants on the CIFAR-10 dataset. We first introduce the dataset and the model architecture, then discuss the optimization process and hyperparameter tuning. Finally we present the training and visualization results.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. A sketch of the dataset is shown in Figure 1.

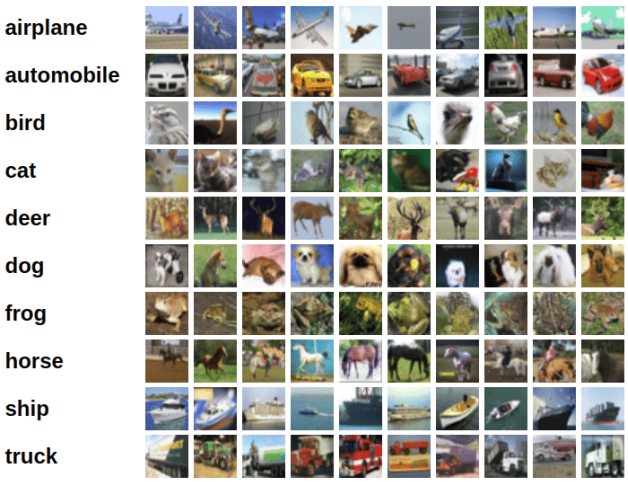


Figure 1: CIFAR-10 Dataset

ResNet, short for Residual Network, is a deep neural network architecture renowned for its capability to train very deep networks effectively. ResNet's innovation lies in its residual connections, which enable the training of significantly deeper networks by mitigating the degradation problem encountered in traditional deep networks. The network incorporates several fundamental components:

- **Fully-Connected Layer**: Typically utilized at the end of the network for classification tasks.
- **2D Convolutional Layer**: Essential for feature extraction from input data using convolutions.
- **2D Pooling Layer**: Used for downsampling feature maps to reduce dimensionality and cost.
- **Activations**: Applied after convolutional and fully-connected layers to introduce non-linearity.

Additionally, ResNet include some of the following components to enhance performance:

- **Batch-Norm Layer**: Normalizes activations within a mini-batch, accelerating convergence.
- **Residual Connection**: Introduces skip connections that bypass one or more layers, facilitating the flow of gradients and alleviating the vanishing gradient problem.

In this project, we mainly use ResNet-18 as the base network, and its architecture is shown in Figure 2. It has 18 layers, including convolutional layers, four residual blocks, and fully connected layers. It is worthnoting that we change the initial 7×7 kernel to 3×3 and delete the max pooling in the first layer, since the CIFAR-10 dataset has a much smaller image size (32×32) compared to ImageNet.

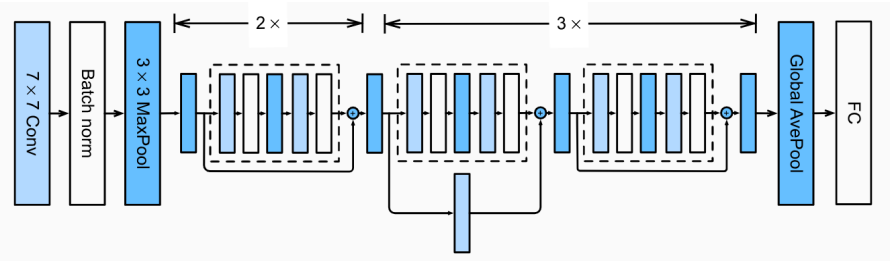


Figure 2: ResNet-18 Architecture, from Dive into Deep Learning

1.2 Optimize the Network

This section details the optimization process for training the ResNet-18 model on the CIFAR-10 dataset. We set the initial learning rate as 0.01, batch size as 128 and learning rate schedule as ReduceLROnPlateau, and train each model for 25 epochs in the following searchings.

1.2.1 Try different number of neurons/filters

To experiment with varying the number of neurons or filters in the ResNet-18 architecture, we introduce a parameter called `filter_multiplier`. This parameter scales the number of filters used in each convolutional layer throughout the network. By adjusting `filter_multiplier`, we increase or decrease the number of filters in all convolutional layers uniformly. Here we set `filter_multiplier` to 2, namely doubling the number of filters in each layer.

We train the larger network for 200 epochs, and the training curves are shown in Figure 3. The results are shown in Table 5 for comparison. Note that the overfitting issue is obvious.

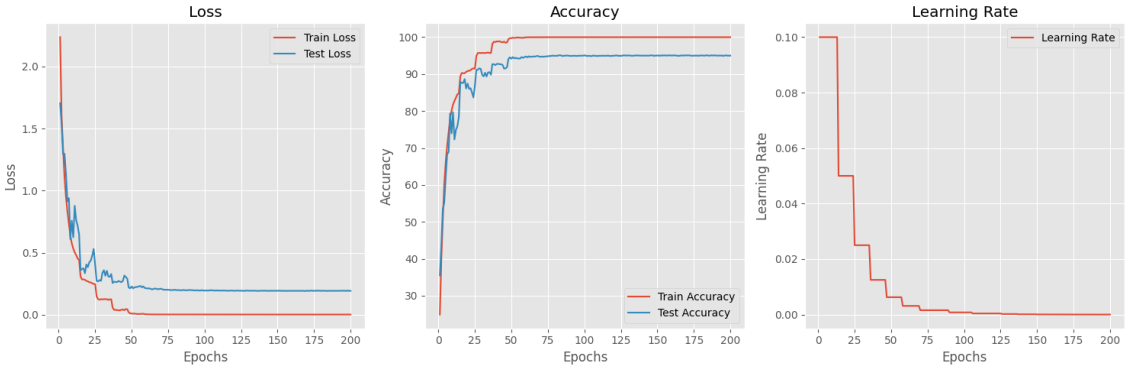


Figure 3: Double Filters Network Training Curves

1.2.2 Try different loss functions

Loss functions are crucial for training neural networks, as they quantify the difference between predicted and actual values, guiding the optimization process.

We experiment with different loss functions to optimize the model's performance. The loss functions we consider include: Cross-Entropy Loss, BCE Loss, MSE Loss, Huber Loss, and SoftMargin Loss. The training curves are shown in Figure 4, and the results are summarized in Table 1.

Table 1: Training 25 epochs with Different Loss Functions

Loss Function	**Train Accuracy**	**Best Test Accuracy**
CrossEntropy	0.9499	**0.8911**
BCEWithLogits	0.9294	0.8799
MSE	0.8765	0.8517
Huber	0.8355	0.8039
SoftMargin	0.9293	0.8734

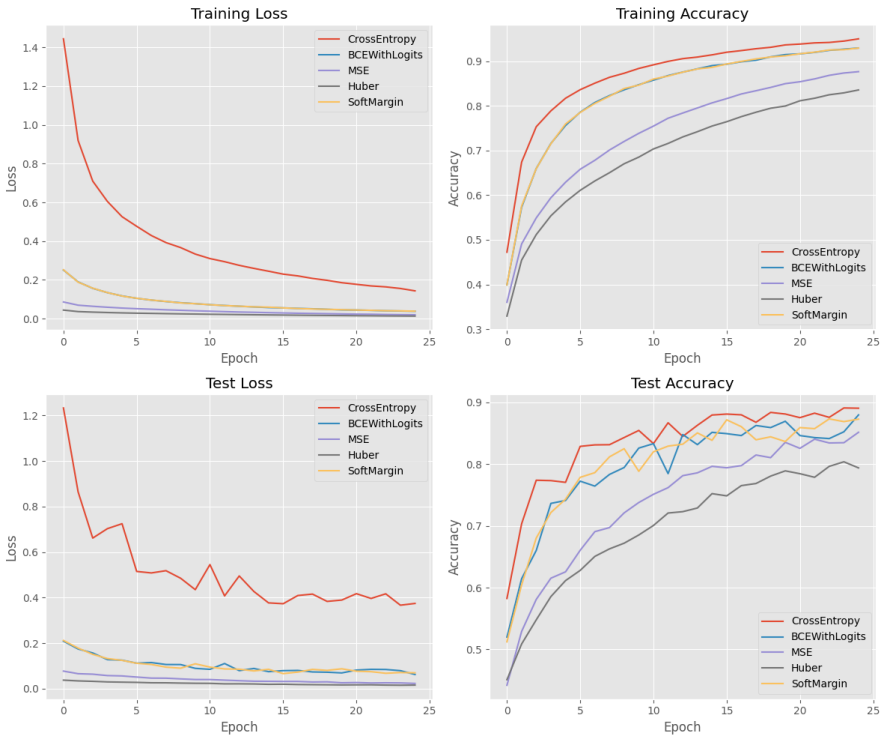


Figure 4: Loss Functions Comparison

We find that the Cross-Entropy Loss function achieves the best performance, with a test accuracy of 89.11%. Note that the loss values do not make much sense here due to different metrics. We continue to choose CrossEntropy Loss in the following experiments.

1.2.3 Try different regularization weights

Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function, discouraging overly complex models.

As there is no special considerations for the sparsity of the model, we only experiment with L2 regularization. We train the model with different L2 regularization weights, and the results are shown in Table 2 and Figure 5.

Table 2: Training 25 epochs with Different Weight Decay Values

Weight Decay	**Train Accuracy**	**Best Test Accuracy**
0.01	0.8379	0.7999
0.001	0.9400	0.8958
0.005	0.8839	0.8535
0.0001	0.9536	0.8983
0.0005	0.9505	**0.8984**

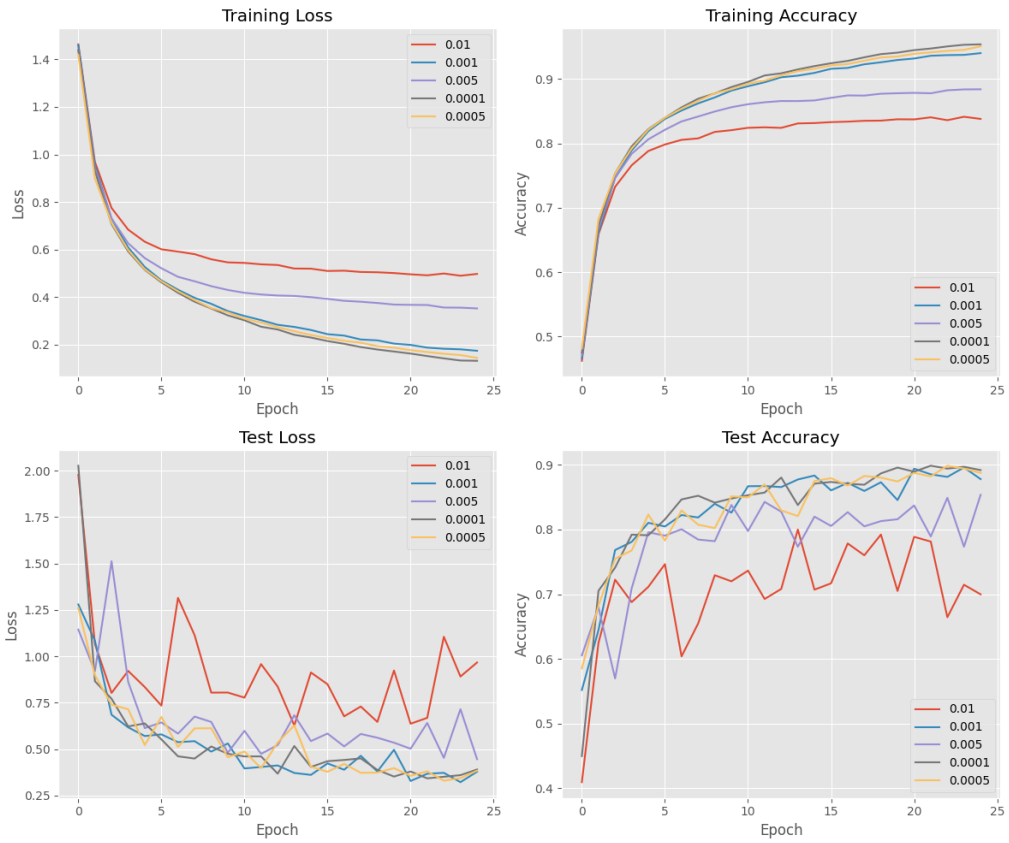


Figure 5: Weight Decay Comparison

We find that a weight decay of 0.0005 achieves the best test accuracy of 89.84%. The larger weight decay values (0.01 and 0.001) result in underfitting and the training process is not stable. 0.0001's performance is close to 0.0005, but the latter is slightly better due to the smaller gap between training and test accuracy. We choose 0.0005 as the weight decay value for the following experiments.

1.2.4 Try different activation functions

Activation functions introduce non-linearity to the model, enabling it to learn complex patterns and relationships in the data.

We experiment with different activation functions, including ReLU, Leaky ReLU, ELU, and GELU. The training results are shown in Table 3 Figure 6.

Table 3: Training 25 epochs with Different Activation Functions

Activation Function	**Train Accuracy**	**Best Test Accuracy**
ReLU	0.9469	0.8945
Leaky ReLU	0.9482	0.8932
ELU	0.8925	0.8750
GELU	0.9536	**0.9019**

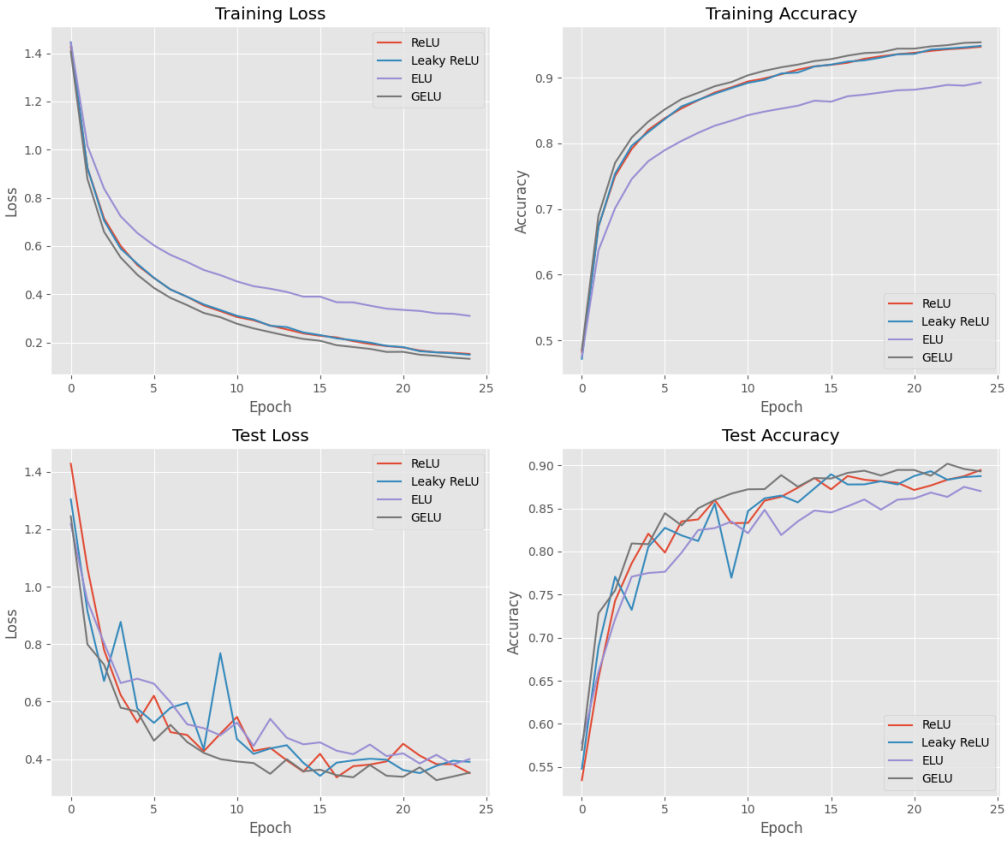


Figure 6: Activation Functions Comparison

We find that the GELU activation function achieves the best performance, with a test accuracy of 90.19%. Since GELU is a variant of ReLU, it introduces non-linearity to the model while maintaining sparsity, leading to better performance. But we also notice that the difference between GELU and ReLU is not significant, and ReLU is still a good choice for activation functions, so we choose ReLU for the following experiments.

1.2.5 Try different optimizers

Optimizers are algorithms used to update the model's parameters during training, minimizing the loss function.

We experiment with different optimizers, including SGD, Adagrad, Adadelta, Adam, and Adamax, and list results in Table 4 and plot the training curves in Figure 7.

We observe that Adagrad achieves the best test accuracy of 89.44%, followed by SGD with 89.20%. The Adam optimizer performs the worst, and its testing process is not stable at all. Since SGD is a simple and effective optimizer, we choose it for the following experiments.

Table 4: Training 25 epochs with Different Optimizers

Optimizer	Train Accuracy	Best Test Accuracy
SGD	0.9491	0.8920
Adagrad	0.9457	**0.8944**
Adadelta	0.8723	0.8201
Adam	0.7224	0.6963
Adamax	0.8321	0.8085

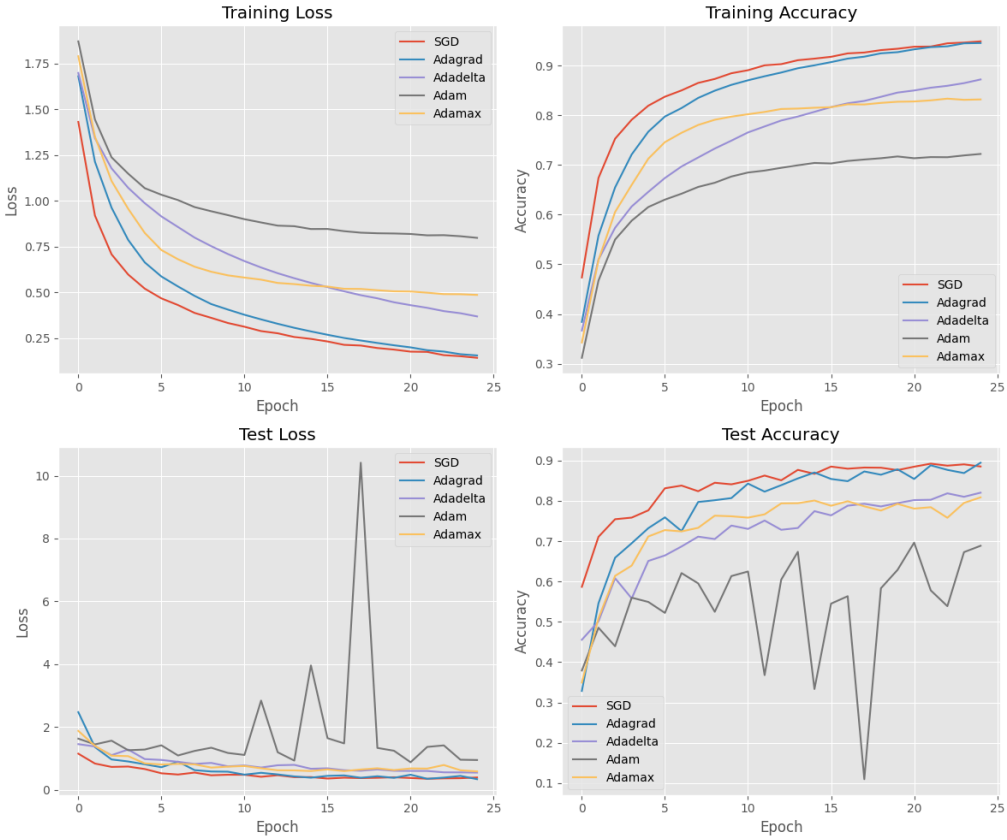


Figure 7: Optimizers Comparison

1.2.6 Try adding dropout layers

To enhance the generalization ability of ResNet-18, we incorporate Dropout layers into the network. Dropout is a regularization technique that aids in preventing overfitting by randomly dropping units (along with their connections) from the neural network during training.

In our implementation, Dropout layers were inserted after each fully connected layer in the classifier section of ResNet-18. This approach involves setting a specific dropout probability (`p`) for each dropout layer, controlling the proportion of units to drop.

We experiment with dropout probability 0.5 for 200 epochs, and the training curves are shown in Figure 8. The results are shown in Table 5 for comparison. Note that the overfitting issue is reduced but the

test accuracy is lower than the original ResNet-18 model.

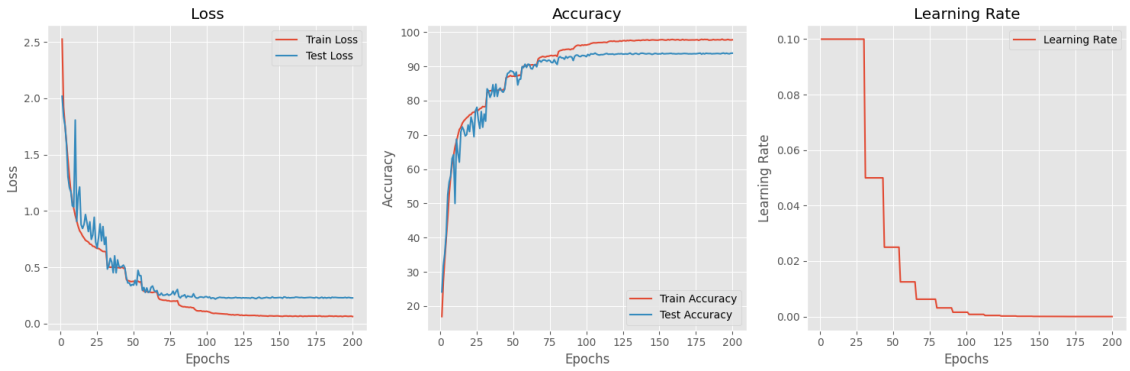


Figure 8: ResNet-18 with Dropout Training Curves

1.3 Implementation and Training Results

The model definitions and main training framework are in courtesy of [1]. We adjust the codes to fit our experiments and record the training results in more detail.
Our final model configurations are as follows:

- Learning rate: 0.01.
- Learning rate scheduler: ReduceLROnPlateau with patience=5, min_lr=1e-6.
- Batch size: 128.
- Number of epochs: 200.
- Loss function: Cross-entropy loss.
- Optimizer: Stochastic Gradient Descent (SGD) with momentum of 0.9.
- Regularization: L2 regularization with weight decay of 0.0005.
- Data augmentation: Random resized cropping and horizontal flipping, normalization.

And we train the following models for 200 epochs, as shown in Table 5.

Table 5: Comparison of Different Model Configurations

Model Configuration	**Parameters (M)**	**Test Accuracy (%)**	**Average Epoch Time (s)**
ResNet18	11.17	94.93	23.61
ResNet18_dropout	11.17	93.90	23.63
ResNet18_filtermul	44.60	95.12	40.69
ResNet34	21.28	**95.50**	33.66

We find that the ResNet-34 model achieves the best test accuracy of 95.50% among all models. Its training curves are shown in Figure 9.

The ResNet-18 model with doubled filters also performs well, achieving a test accuracy of 95.12%, but the training time is significantly longer due to the increased number of parameters. This indicates that increasing the number of filters may not be as effective as using a deeper network.

The ResNet-18 model with dropout layers achieves a test accuracy of 93.90%, which is lower than the original ResNet-18 model. This suggests that dropout layers may not be necessary for this task, or the dropout probability may need to be adjusted. Since CIFAR-10 is a relatively simple dataset, the original ResNet-18 model without dropout layers already achieves good performance.

The original ResNet-18 model achieves a test accuracy of 94.93%, which is slightly lower than the ResNet-34 model. This indicates that increasing the depth of the network can improve performance, but the improvement may not be significant due to the simplicity of the dataset.

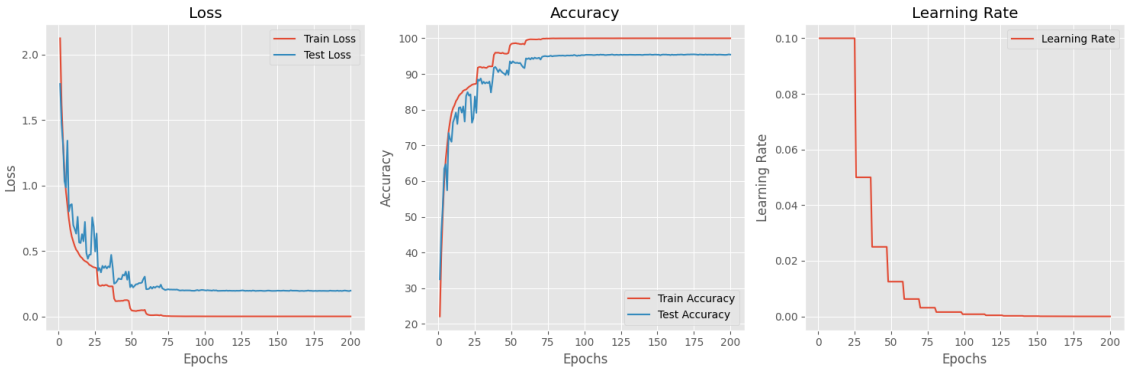
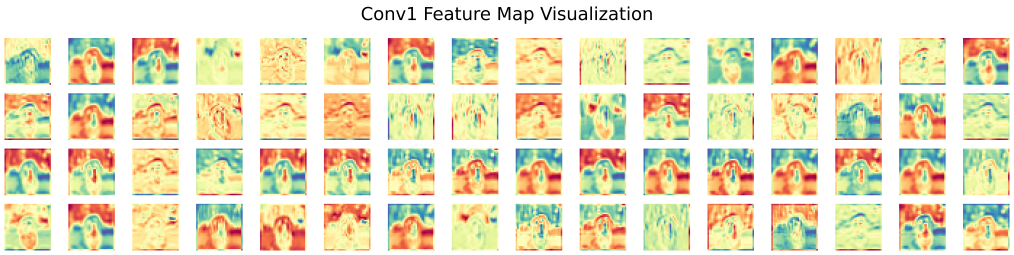


Figure 9: ResNet-34 Training Curves

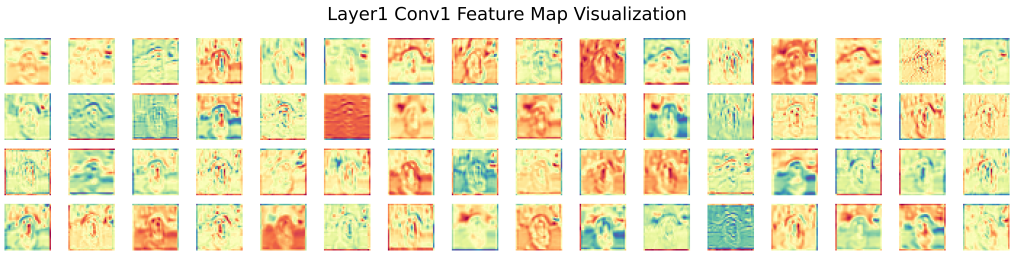
1.4 Visualization

We visualize the first two convolution layers' feature maps of the ResNet-18 model. Our case image is shown in Figure 10 [4], and the feature maps are shown in Figure 11.

Figure 10: Case Image



(a) Conv1 Feature Maps



(b) Layer1 Conv1 Feature Maps

Figure 11: Feature Maps Visualization

In the first convolutional layer, ResNet-18 learnt more explanatory features, and the outline of the dog was portrayed very clearly; in the second one, the features were a bit fuzzy compared to the first one, but it was still easy to observe certain local features, which shows that the model's explanatory power is still good.

2 Task 2: Batch Normalization

2.1 Introduction

Batch Normalization (BN) aims to mitigate internal covariate shift, thereby speeding up the training of deep neural networks. It achieves this by normalizing layer inputs to stabilize their means and variances. This process also enhances gradient flow by reducing sensitivity to parameter scales and initial values, enabling the use of higher learning rates without the risk of divergence. Additionally, Batch Normalization acts as a regularization technique, potentially reducing the need for Dropout.

Batch Normalization formula is as follows:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu^{(k)}}{\sqrt{\sigma^{2(k)} + \epsilon}},$$

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)},$$

where $x^{(k)}$ is the input to the layer (a minibatch), $\mu^{(k)}$ and $\sigma^{2(k)}$ are the mean and variance of $x^{(k)}$, $\gamma^{(k)}$ and $\beta^{(k)}$ are learnable parameters, and ϵ is a small constant to prevent division by zero.

Recent research results show that BN reparametrizes the underlying optimization problem to make its landscape significantly more smooth. So along this line, we are going to measure:

1. Loss landscape or variation of the value of the loss;
2. Gradient predictiveness or the change of the loss gradient;
3. Maximum difference in gradient over the distance.

2.2 VGG-A with and without BN

We experiment with VGG-A with and without Batch Normalization on CIFAR-10 to explore the impact of BN on optimization. The model configurations are as follows (mostly following [2]):

- Learning rate: 0.1.
- Batch size: 128.
- Number of epochs: 20.
- Loss function: Cross-entropy loss.
- Optimizer: Stochastic Gradient Descent (SGD).
- Duration: 30 epochs (to record the loss landscape).

The training curves are shown in Figure 12.

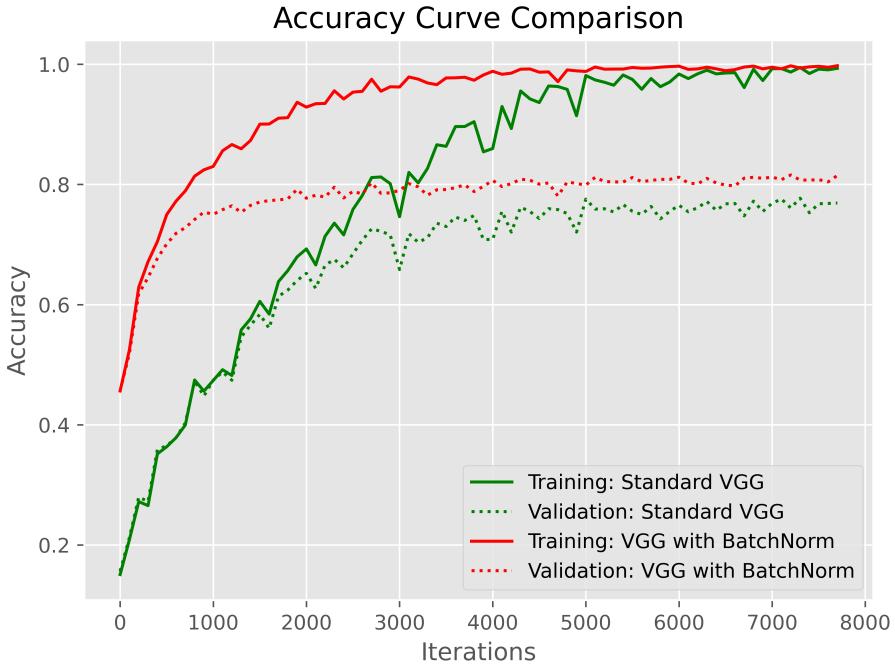


Figure 12: VGG-A with and without BN Training Curves

Obviously, the VGG-A model with Batch Normalization achieves better performance, in terms of convergence speed and final accuracy. The training process is more stable and the test accuracy is higher compared to the model without BN.

2.3 How does BN help optimization?

2.3.1 Loss Landscape

By setting the learning rate to 0.15, 0.1, 0.075 and 0.05 (larger learning rate to converge) and training each for 20 epochs, the fill-between plot of the loss landscape can be shown in Figure 13.

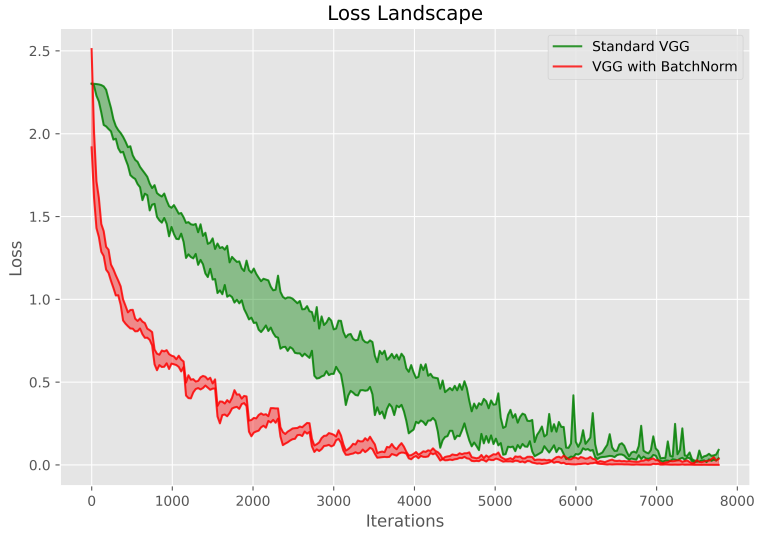


Figure 13: Loss Landscape with Different Learning Rates

We can see that the loss landscape of the model with Batch Normalization is smoother and more convex compared to the model without BN. This indicates that BN really helps to reduce the loss variance and thus boost the training speed.

2.3.2 Gradient Predictiveness

We record the change of the L2 loss gradient in the last linear layer of the model with and without BN. The results are shown in Figure 14.

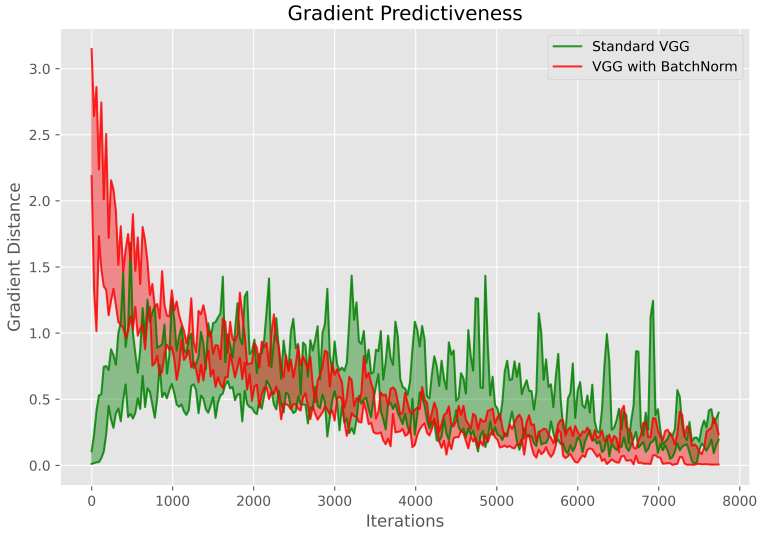


Figure 14: Gradient Predictiveness

We can see that the model with Batch Normalization has a more stable gradient, and the gradient change

is smaller compared to the model without BN.

2.3.3 Beta Smoothness

We say that the gradient is beta-smooth if the maximum difference in gradient over the distance is bounded by a constant beta. The formula is as follows:

$$\beta_t = \max \frac{\|\nabla f(w_{t+1}) - \nabla f(w_t)\|}{lr \cdot \|\nabla f(w_t)\|}$$

We plot the beta-smoothness of the model with and without BN in Figure 15.

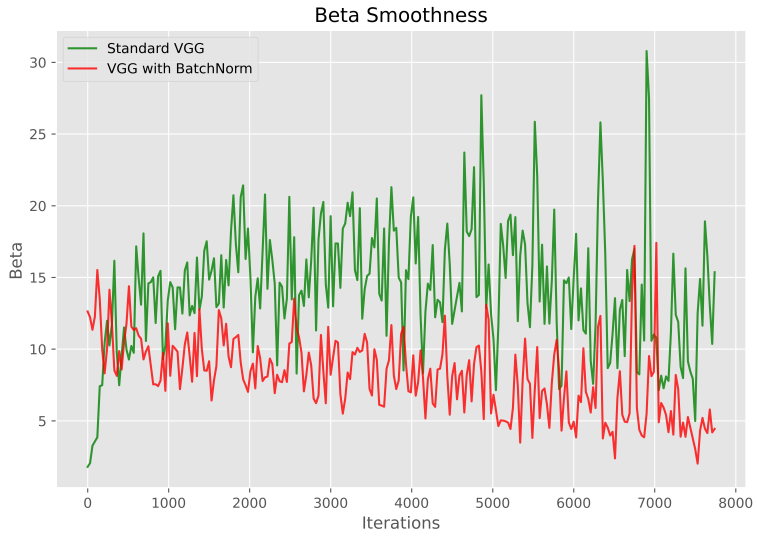


Figure 15: Beta Smoothness

This verifies that the model with Batch Normalization has a smoother gradient and is more beta-smooth compared to the model without BN.

References

[1] https://github.com/kuangliu/pytorch-cifar

[2] How does batch normalization help optimization? In NeurPIS, 2018.

[3] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[4] Pexels Svetozar Milashevich Image: https://www.pexels.com/zh-cn/photo/1490908/