Project-2 of “Neural Network and Deep Learning”

Yanwei Fu

May 20, 2021

Abstract

(1) This is the second project of our course. The deadline is 5:00pm, May 30, 2021. Please upload the report via elearning.

(2) The goal of your write-up is to document the experiments you’ve done and your main findings. So be sure to explain the results. The report can be written by Word or Latex. Generate a single pdf file of your mini-projects and turned in along with your code. package your code and a copy of the write-up pdf document into a zip or tar.gz file and named as Project2-*your-student-id*_your_name.[zip|tar.gz]. Only include functions and scripts that you modified. Also put the names and Student ID in your paper.

(3) About the deadline and penalty. In general, you should submit the paper according to the deadline of each mini-project. The late submission is also acceptable; however, you will be penalized 10% of scores for each week’s delay.

(4) It is not required to answer the questions from the extra bonus sections. We do count the scores of extra bonus in the final scores of this course.

(5) We will allocate GPUs.

1 Train a Network on CIFAR-10 (60%)

CIFAR-10 [7] 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.

In this project, you will train neural network models on CIFAR-10 to optimize performance. Report the best test error you are able to achieve on this dataset, and report the structure you constructed to achieve this.

1.1 Getting Started

1. You may download the dataset from the official website [2] or use the torchvision package. Here is a demo provided by PyTorch [1].

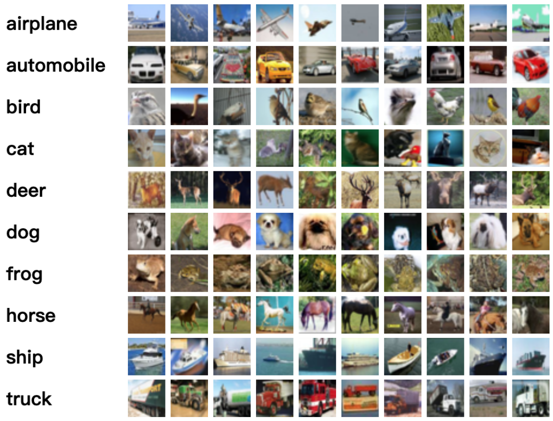


Figure 1: CIFAR-10 Dataset

```
1  import torch
2  import torchvision
3  import torchvision.transforms as transforms
4  transform = transforms.Compose( [transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
       0.5))])
5  trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
6  trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)
7  testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
8  testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=False, num_workers=2)
```

When setting *download=True* in line 5, it will download the dataset to the defined *root* path automatically. For the construction, training and testing the neural network model, you may read the tutorial [1] as a start.

2. Your network will have all the following components:(16%)
   (a) Fully-Connected layer;
   (b) 2D convolutional layer;
   (c) 2D pooling layer;
   (d) Activations;
3. Your network may have some of (at least one) the following components:(8%)
   (a) Batch-Norm layer;
   (b) Drop out;
   (c) Residual Connection;
   (d) Others;
4. To optimize your network, you will try all the following strategies:(8%)
   (a) Try different number of neurons/filters;

   (b) Try different loss functions (with different regularization);
   (c) Try different activations;

5. To optimize your network, you may select one of the following strategies: (8%)
   (a) Try different optimizers using *torch.optim*;
   (b) Implement an optimizer for a network including 2(a)-(d), and use *torch.optim* to optimize your full model;
   (c) Implement an optimizer for your full model;

6. Reveal the insights of your network, say, visualization of filters, loss landscape, network interpretation. (8%)

1.2 Scores of this task

Yes, you are totally free to use any component in torch, pytorch or other deep learning toolbox. Then how do we score your project in this task? We care about

(1) (12%)The classification performance of your network, in term of total parameters of network, network structure, and training speed. For the projects of similar results, we will check the total parameters, and network structure, or whether any new optimization algorithms are used to train the network.

(2) Any insightful results and interesting visualization of the learned model, and training process, or anything like this.

(3) You can report the results from more than one network; but if you directly utilize the public available models without anything changes, the scores of your projects would be slightly penalized.

2 Batch Normalization (30%)

Batch Normalization (BN) is a widely adopted technique that enables faster and more stable training of deep neural networks (DNNs). The tendency to improve accuracy and speed up training have established BN as a favorite technique in deep learning. At a high level, BN is a technique that aims to improve the training of neural networks by stabilizing the distributions of layer inputs. This is achieved by introducing additional network layers that control the first two moments (mean and variance) of these distributions.

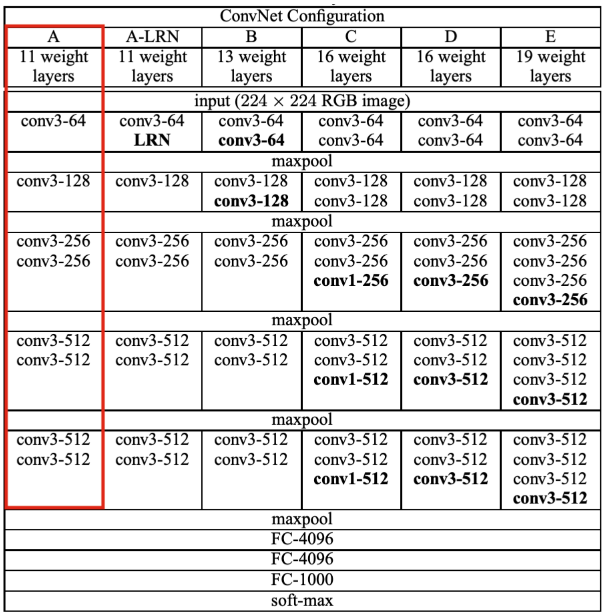
In this project, you will first test the effectiveness of BN in the training process, and then explore how does BN help optimization. The sample codes are provided by Python.

2.1 The Batch Normalization Algorithm

Here we primarily consider BN for convolutional neural networks. Both the input and output of a BN layer are four dimensional tensors, which we refer to as $I_{b,c,x,y}$ and $O_{b,c,x,y}$, respectively. The dimensions corresponding to examples within a batch b, channel c, and two spatial dimensions x, y respectively. For input images the channels correspond to the RGB channels. BN applies the same normalization for all activations in a given channel,

$$O_{b,c,x,y} \leftarrow \gamma_c \frac{I_{b,c,x,y} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} + \beta_c \qquad \forall b, c, x, y.$$

Here, BN subtracts the mean activation $\mu_c = \frac{1}{|\mathcal{B}|} \sum_{b,x,y} I_{b,c,x,y}$from all input activations in channel c, where $\mathcal{B}$ contains all activations in channel c across all features b in the entire mini-batch and all spatial x, y, locations. Subsequently, BN divides the centered activation by the standard deviation σ_c (plus ϵ for numerical stability) which



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2: ConvNet configurations (shown in columns) of VGG model

is calculated analogously. During testing, running averages of the mean and variances are used. Normalization is followed by a channel-wise affine transformation parametrized through γ_c, β_c, which are learned during training.

Dataset. To investigate batch normalization we will use following experimental setup: image classification on CIFAR-10 with a network has the same architecture as VGG-A except the size of Linear layers is smaller since input assumed to be $32 \times 32 \times 3$, instead of $224 \times 224 \times 3$. And all sample codes are implemented based on *Pytorch*.

You can run *loaders.py* to download CIFAR-10 dataset, and output some examples to familiarize yourself with the data storage format. Note that if you are using a remote server, you need to use *matplotlib.pyplot.savefig()* function to save the plot results, and then download to local to view.

2.2 VGG-A with and without BN (15%)

In this section, you will compare the performance and characteristics of VGG-A with and without BN. We encourage you to extend and modify the provided code for clearer and more convincing experimental results. Note that you should understand the code first instead of using it as a black box.

If you want to use partial dataset to train for faster results, set *n_ items* to meet your wish. The basic VGG-A network is implemented in *VGG.py*, you can train this network first to understand the overall network architecture and view the training results. Then write a class *VGG_ BatchNorm* to add the BN layers to the original network, and finally visualize the training results of the two for comparison. Sample code for training and visualizing has been included in VGG_Loss_Landscape.py.

2.3 How does BN help optimization? (15%)

This part is the focus of the project. It is not enough to just use BN. We should understand why it can play a positive role in our optimization process so as to have a more comprehensive understanding of the optimization process of deep learning and select the appropriate network structure according to the actual situation in the future works. You may want to check some papers, e.g., [3].

So what stands behind BN? After all, in order to understand how BN affects the training performance it would be logical to examine the effect that BN has on the corresponding optimization landscape. To this end, recall that our training is performed using gradient descent method and this method draws on the first-order optimization paradigm. In this paradigm, we use the local linear approximation of the loss around the current solution to identify the best update step to take. Consequently, the performance of these algorithms is largely determined by how predictive of the nearby loss landscape this local approximation is.

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.

2.3.1 Loss Landscape

To test the impact of BN on the stability of the loss itself, *i.e.*, its Lipschitzness, for each given step in the training process, you should 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. You can do as following for a simple implementation:

1. 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]);

2. Save the training loss of all models for each step;

3. 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*;

4. Plot the results of the two lists, and use *matplotlib.pyplot.fill_ between* method to fill the area between the two lines.

Use the same approach for VGG-A model with BN. Finally, try to visualize the results from VGG-A with BN and without BN on the same pic for more intuitive results.

For your better understanding, we provide sample code for visualizing the loss landscape (VGG_Loss_Landscape.py). You need to understand the code and train different models to reproduce the results of the Figure 2. Please feel free to modify and improve the sample code and report your choice of learning rates. Most importantly, show your final comparison results with the help of *matplotlib.pyplot.*

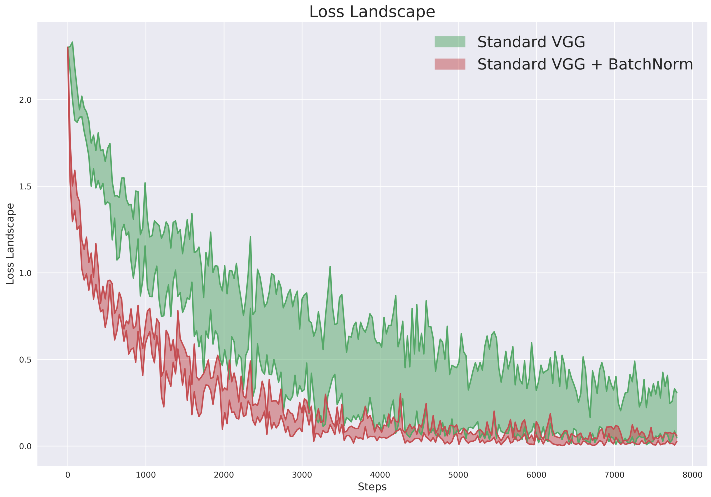


Figure 3: Loss landscape example

References

[1] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

[2] https://www.cs.toronto.edu/ kriz/cifar.html.

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

[4] Yanwei Fu, Chen Liu, Donghao Li, Xinwei Sun, Jinshan Zeng, and Yuan Yao. Dessilbi: Exploring structural sparsity of deep networks via differential inclusion paths. In *International Conference on Machine Learning*, pages 3315–3326. PMLR, 2020.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

2.4 Extra Bonus 1 (20%)

Following Sec. 2.3, there are some extra bonus, which is not required in this project.

2.4.1 Gradient Predictiveness (10%)

Similarly, to illustrate the increase in the stability and predictiveness of the gradients, you will make analogous measurements for the ℓ_2 *distance* between the loss gradient at a given point of the training and the gradients corresponding to different points along the original gradient direction. At this time you still need to choose different learning rates to train different models, but the difference is that this time the gradient value of the loss (back-propagated back to the output layer) is saved instead of the loss value, of course you can use the same model as Sec. 2.3.1. Then you should calculate the ℓ_2 distance between the loss gradient at *ith step* and *(i-1)th step* for the same model (you can try other step size and show your results and comments).

Similarly, please write a function *VGG_ Grad_ Pred()* to get clear comparison, and as before, you need to report the parameters you selected and the experimental results. If you can add your comments and thoughts, it will make the report more complete.

2.4.2 "Effective" β-Smoothness(10%)

To further demonstrate the effect of BN on the stability/Lipschitzness of the gradients of the loss, we should plot in the "effective" β-smoothness of the standard and BN networks throughout the training. The "effective" β-smoothness refers to the maximum difference (in ℓ_2-norm) in gradient over distance moved in that direction.

Similarly, please write a function *VGG_ Beta_ Smooth()* to get clear comparison, and as before, you need to report the parameters you selected and the experimental results. If you can add your comments and thoughts, it will make the report more complete.

2.5 Extra Bonus 2 (30%)

Nowadays the great success of deep learning relies on the over-parameterization. Though the over-parameterization can benefit the training process, it brings difficulties when using deep neural networks in our daily life due to a large quantity of parameters. As mentioned in our lecture, instead of using backward selection methods to get a compact model, we can use DessiLBI[4] to conduct a forward selection that can get the trained over-parameterized model and sparse structure simultaneously.

For DessiLBI, the important sparse structures are discovered by the augmented variables Γ. For shallow networks and simple datasets such as LeNet on MNIST, the vanilla DessiLBI can get a good structure as shown in Figure. 4. (Please refer to Figure 1 in [4].) The first step of this part is to find a good structure of LeNet on MNIST using DessiLBI. The example code is contained in https://github.com/DessiLBI2020/DessiLBI/tree/master/DessiLBI/. The setting of hyperparameters can be found in [4]. Example code for training LeNet on MNIST is also contained in this repo.

After finishing the first step, you can try to combine DessiLBI with other optimization algorithm such as Adam [6] and use the combined one to find sparse structure on MNIST and CIFAR-10. (Both MNIST and CIFAR-10 can be obtained by using torchvision, please refer to https://pytorch.org/vision/stable/datasets.html). You are required to report both the accuracy and the structure for these experiments.

For instance, we can combine Adam[6] with DessiLBI. You can view the implementation of Adam in Pytorch (https://pytorch.org/docs/1.2.0/_modules/torch/optim/adam.html#Adam), then you can write a similar update for DessiLBI. Here you are required to alter the update of W in DessiLBI and keep the update of Γ the same as the original one. Figure. 5 shows the core of Adam implementation in Pytorch. In this experiment, you can consider using Adam or other adaptive gradient methods to update the W of DessiLBI in the process of optimizing the deep networks. The possible choices of deep networks could be ResNet-56 [5], VGG-19, and other more advanced deep structures. You can use MNIST and CIFAR10 dataset.

Of course, it would be very nice to change the way of updating Γ by Adam or other adaptive gradient methods. This would be very interesting, but not the required in this project.

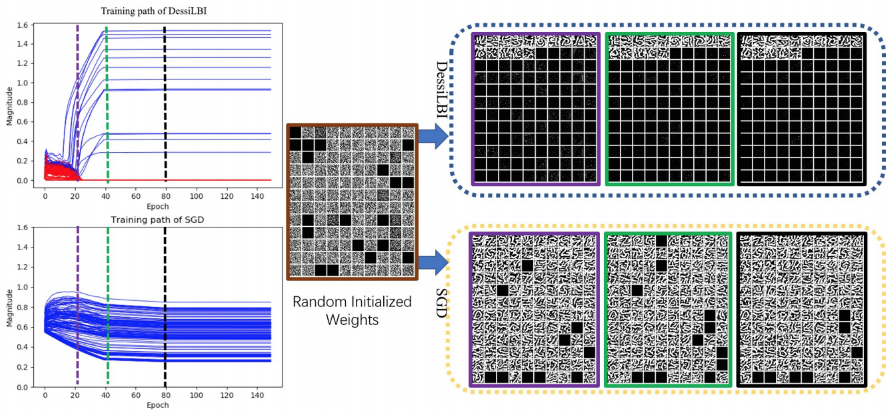
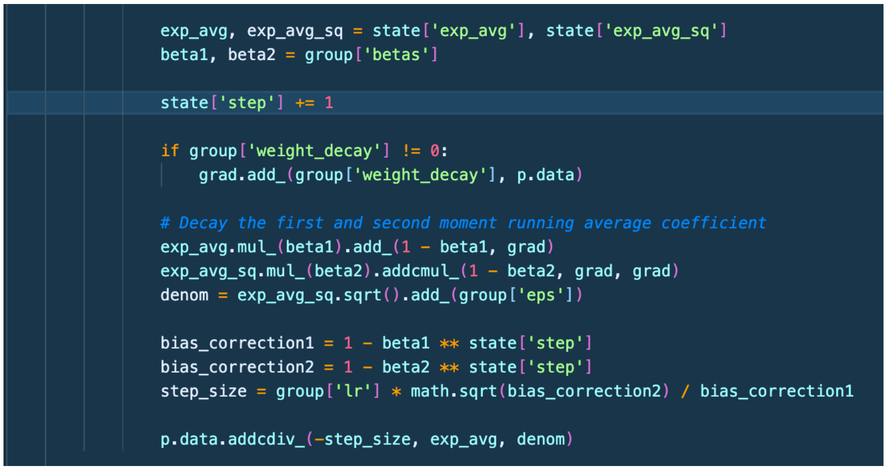


Figure 4: Illustration of training LeNet on MNIST.



```
exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
beta1, beta2 = group['betas']

state['step'] += 1

if group['weight_decay'] != 0:
    grad.add_(group['weight_decay'], p.data)

# Decay the first and second moment running average coefficient
exp_avg.mul_(beta1).add_(1 - beta1, grad)
exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)
denom = exp_avg_sq.sqrt().add_(group['eps'])

bias_correction1 = 1 - beta1 ** state['step']
bias_correction2 = 1 - beta2 ** state['step']
step_size = group['lr'] * math.sqrt(bias_correction2) / bias_correction1

p.data.addcdiv_(-step_size, exp_avg, denom)
```

Figure 5: Adam Implementation