CS 391L Machine Learning HW 5: Backpropagation

Yuege Xie, EID: yx4256, Email: yuege@ices.utexas.edu*

1 Introduction

Handwritten digits recognition is important and widely used in our daily life, such as recognizing zip codes for postal mail sorting and processing the amount of bank check. MNIST dataset developed by LeCun contains $50,000$ training and $10,000$ test images of hand written digits of numbers $0-9$. Each image is a 28×28 gray-scale $(0-255)$ image with label $0-9$. Figure 1 shows some examples of training and test images with their true labels. The project aims at training a neural network as a classifier to classify unseen handwritten digits. We evaluate the classifier by its accuracy on test data.

To train neural networks, we apply gradient based methods like Adam to optimize loss functions using back propagation to compute the gradients. The back propagation computes the gradient of the loss function with respect to each weight by the chain rule by computing the gradient one layer at a time and iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule (from Wikipedia).

In this project, I used Torch package in Python to do auto differentiation to train one-hidden-layer neural networks with different hidden layer sizes $(1000, 2000, 5000)$ and different normalization methods such as batch normalization (Ioffe and Szegedy, 2015) and weight normalization (Salimans and Kingma, 2016). I compared the loss (Figure 3 and 5) and accuracy (Figure 4 and 6) during training and test periods of different models and list the best test accuracy of each model in Table 1.

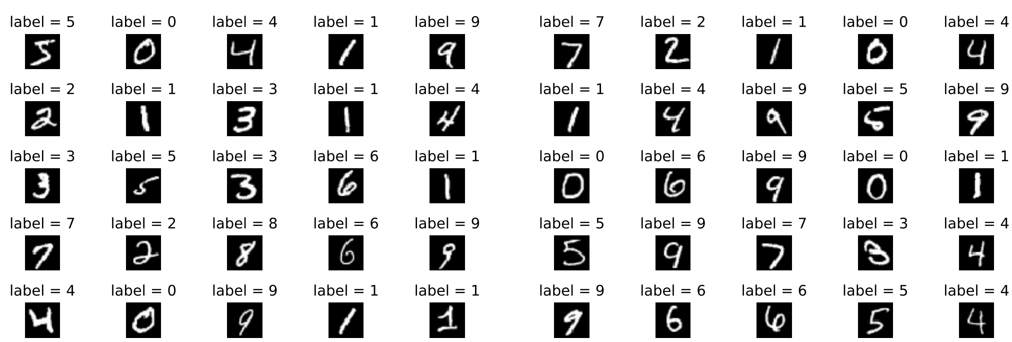

Figure 1: Left: Examples of training images; Right: Examples of test digits

2 Method

I put the codes, results and plots in the backprop.zip. The main models and codes are in utils.py and experiment.py; to run the models, please run run-models.ipynb , you can use different options and see the tensor board results by running Ipython code inside; to see the reproduce plots, you can run plot-records.ipynb. I train these models using Google Cloud and make it run on Google Colab. The main steps are as follows.

1. **Build data loaders:** I download the data in the "data" file folder (with "download=True"). I use data loder with Normalize the data with mean 0.1307 and std 0.3081. I use 100 as training data batch size, which means 500 iterations per epoch.

*Oden Institute for Computational Engineering and Sciences, UT Austin.

2. **Build models:**

   - **One Hidden Layer Neural Network:** I use one-hidden layer neural network with ReLU activation. First layer (fc1): 784 as input size and hidden layer size $(1000, 2000, 5000)$ as output size; Activation (relu); Second layer(fc2): hidden layer size $(1000, 2000, 5000)$ as input size and number of classes 10 as output size. I denote this model as "nmodel".
   - **Models with normalization layer:** I use batch normalization ("bmodel") and weight normalization ("wmodel"), the results are in Figure 3 and 4.

3. **Train and test models:**

   - **Backpropagation:** The back propagation using PyTorch is in Figure 2. (optimizer.zero_grad(), loss.backward(), optimizer.step())

```
for epoch in pbar(range(params.num_epochs)):
    for phase in ['train', 'test']:
        logs = {'Loss': 0.0, 'Accuracy': 0.0}
        # Set the model to the correct phase
        model.train() if phase == 'train' else model.eval()

        for images, labels in getattr(params, phase + '_loader'):
            # Move tensors to the configured device
            images = images.reshape(-1, 28 * 28).to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            with torch.set_grad_enabled(phase == 'train'):

                # Forward pass
                outputs = model(images)
                loss = criterion(outputs, labels)
                accuracy = torch.sum(torch.max(outputs, 1)[1] == labels.data).item()

                # Update log
                logs['Loss'] += images.shape[0] * loss.detach().item()
                logs['Accuracy'] += accuracy

                # Backward pass
                if phase == 'train':
                    loss.backward()
                    optimizer.step()
```

Figure 2: Back propagation using PyTorch.

   - **Optimizer:** The loss is "CrossEntropyLoss". I use Adam with 0.01 as initial learning rate, momentum = 0.9 to train the models. I use learning rate scheduler to decay learning rate by 0.1 for every 20 epochs.
   - **Train and Test model:** Please run run-models.ipynb to train models, you can choose different models ("nmodel", "bmodel" and "wmodel") with different optimizers and learning rates. I train the models with 100 epochs to make it enough to get 100% training accuracy with almost 0 training error so that it is fair to compare test accuracy. I use $10,000$ test data to test models and the criteria is test accuracy.

3 Results

The results are in "runs" and "record", and the plots are in "plots". I show and analyze the plots as follows. I compared the loss (Figure 3 and 5) and accuracy (Figure 4 and 6) during training and test periods of different models and list the best test accuracy of each model in Table 1.

Model	hs5000-step20	hs2000-step20	hs1000-step20
nmodel	97.35	97.22	97.31
bmodel	98.73	98.73	98.73
wmodel	98.65	98.42	98.50

Table 1: Best Accuracy of Different Models and different hidden sizes

3.1 Accuracy of different models (normalization)

I compare "nmodel" (plain NN), "bmodel" (NN with batch normalization) and "wmodel" (with weight normalization) in 3 (loss) and 4 (accuracy). I train the models with 100 epochs to make it enough to get 100% training accuracy with almost 0 training error so that it is fair to compare test accuracy. As it shows in Figure 4, the "nmodel" can only achieve less than 98% (worst), but "bmodel" can achieve round 98.73% accuracy (best). The performance of "wmodel" is between the two.

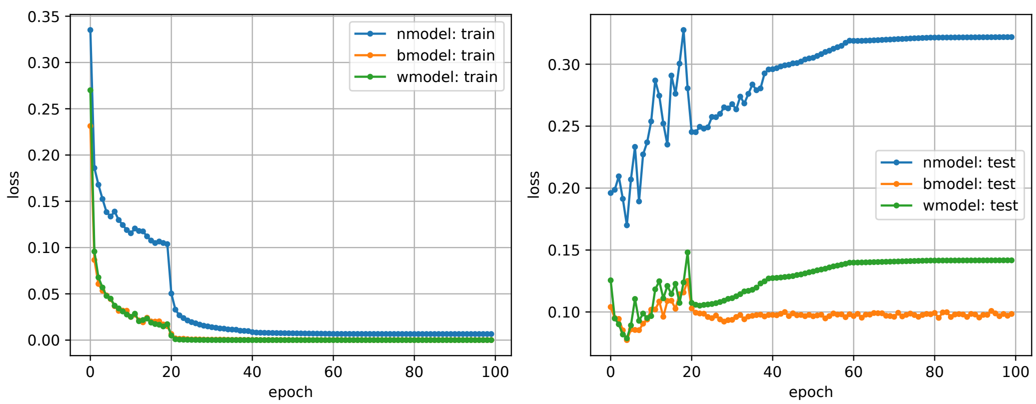

Figure 3: Training (left) and test (right) losses of different models.

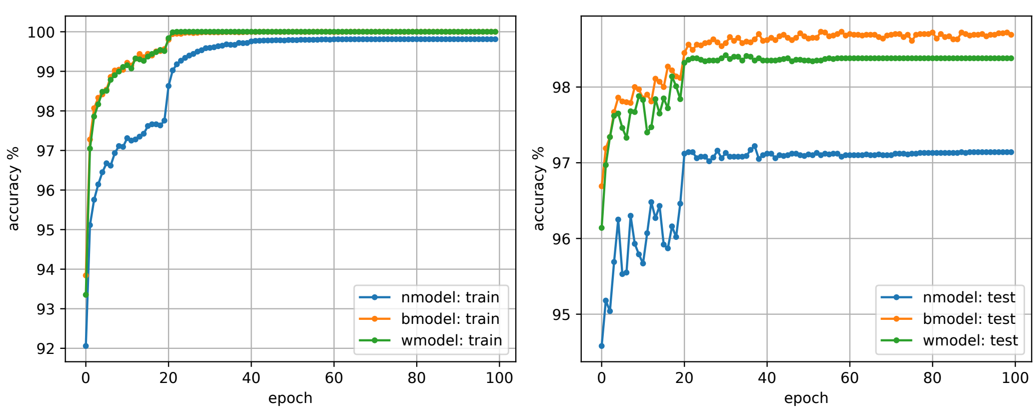

Figure 4: Training (left) and test (right) accuracy of different models.

3.2 Accuracy of different hidden layer sizes

I compare "nmodel" with different hidden layer size $(1000, 2000, 5000)$ in 5 (loss) and 6 (accuracy). I train the models with 100 epochs to make it enough to get 100% training accuracy with almost 0 training error so that it is fair to compare test accuracy. As it shows in Figure 6, $hs = 5000$ has best test accuracy, $hs = 2000$ performs worst, and $hs = 1000$ is similar to $hs = 5000$. From Table 1, we can see the best accuracy does not change much with hidden layer sizes, especially for model with batch normalization ("bmodel").

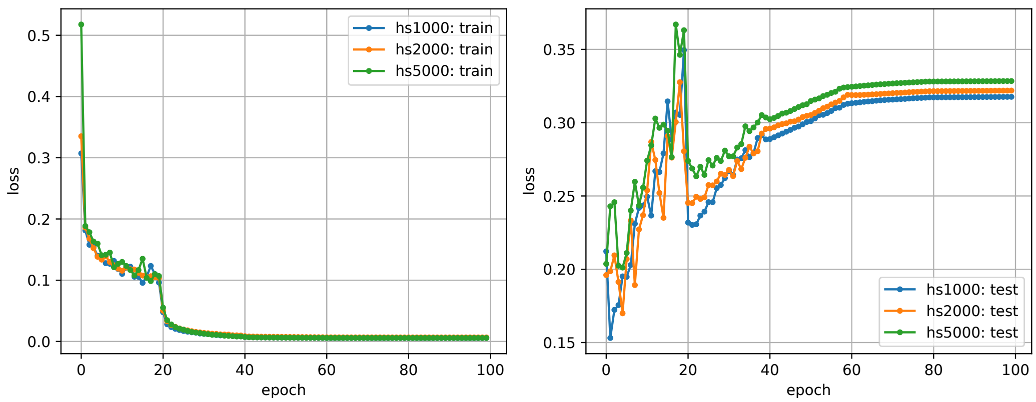

Figure 5: Training (left) and test (right) losses of different hidden layer sizes using nmodel.

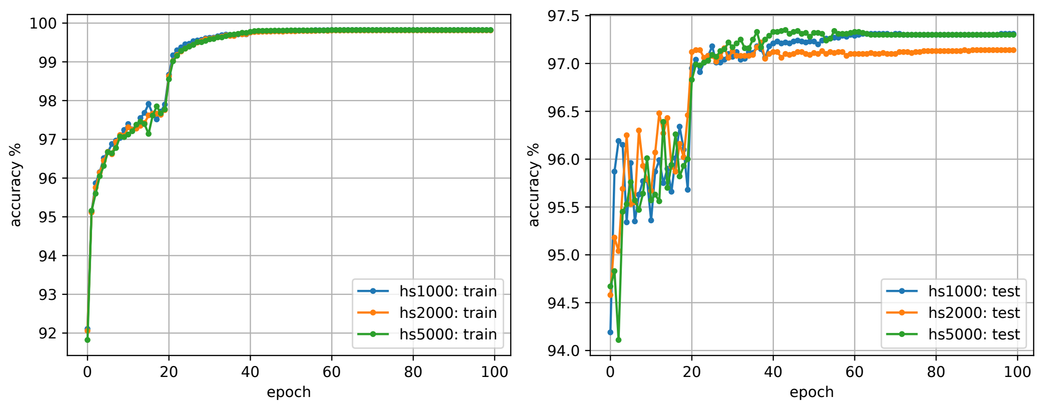

Figure 6: Training (left) and test (right) accuracy of different hidden layer sizes using nmodel.

4 Summary

The one-hidden layer neural network model, model training details, and methods to run my models are in "Method". The code using PyTorch to do back propagation is in Figure 2. I train and test models with different normalization methods and different hidden layer sizes. This project uses the accuracy (%) on test dataset to evaluate NN classifiers. The model with batch normalization ("bmodel") has best accuracy than plain NN ("nmodel") and with weight normalization ("wmodel"), and it is stable as hidden layer size varies. With different hidden layer size, $hs = 5000$ has best performance. However, as long as the model is overparameterized (the number of parameters are larger than the number of data) and it is trained long enough, it can achieve global minimum (100% training accuracy or 0 loss) (Allen-Zhu et al., 2018). Hence, the best test accuracy does not change too much, especially for "bmodel".

References

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in neural information processing systems*, pages 901–909, 2016.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *arXiv preprint arXiv:1811.03962*, 2018.