Report of Assignment 2

COMP 576: A Practical Introduction to Deep Machine Learning - Fall 2022

Eddy Huang (eh58@rice.edu)

Department of Computer Science

Rice University

1 Visualizing a CNN with CIFAR10

a) CIFAR10 Dataset

Please refer to the source code in the zip files.

b) Train LeNet5 on CIFAR10

Analysis: with train = 1000, we first change the optimizer with different learning rates 1) learning rate: $1e-2$, $1e-3$, $1e-4$; and 2) optimizer: AdamOptimizer, GradientDescentOptimizer, and MomentumOptimizer. Then, we changed the momentum values for different optimizers with the best learning rate to observe the performance differences if applicable. Thus, we generated 15 pairs of test accuracy and loss figures. Among all the test results, the best performance is Test 2: AdamOptimizer with learning rate $1e-3$ and default momentum values of 0.9. After doing some research, I learned that in general momentum values of 0.9 gives the best performance [1]. See below for the attachments,

Test 1: optimizer = AdamOptimizer, learning rate = $1e-2$, learning rate = 0.9

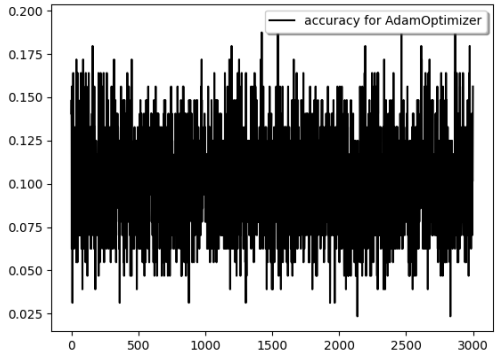

Figure 1.1

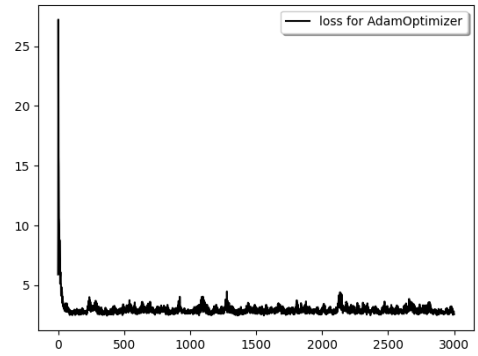

Figure 1.2

step 0, loss 5.89619, training accuracy 0.140625

step 100, loss 2.57446, training accuracy 0.0859375

…

step 2800, loss 3.02012, training accuracy 0.132812

step 2900, loss 2.76285, training accuracy 0.0625

test accuracy 0.1

Test 2: optimizer = AdamOptimizer, learning rate = $1e-3$, learning rate = 0.9

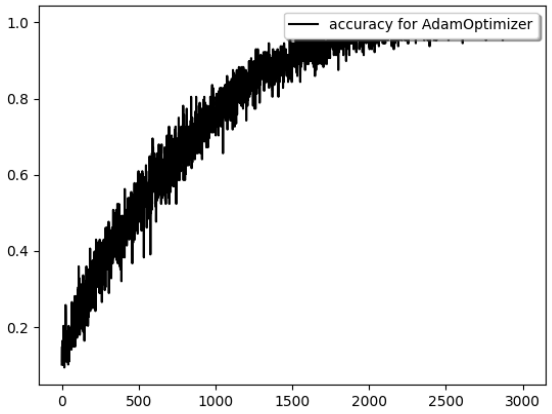

Figure 2.1

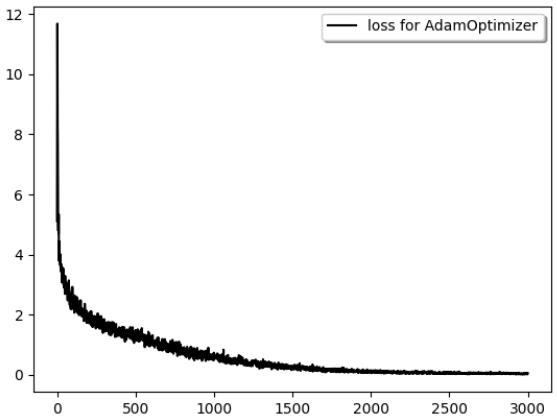

Figure 2.2

step 0, loss 5.11542, training accuracy 0.0703125

step 100, loss 2.35517, training accuracy 0.265625

…

step 2800, loss 0.053781, training accuracy 0.984375

step 2900, loss 0.0501603, training accuracy 0.992188

test accuracy 0.54

Test 3: optimizer = AdamOptimizer, learning rate = $1e-4$, learning rate = 0.9

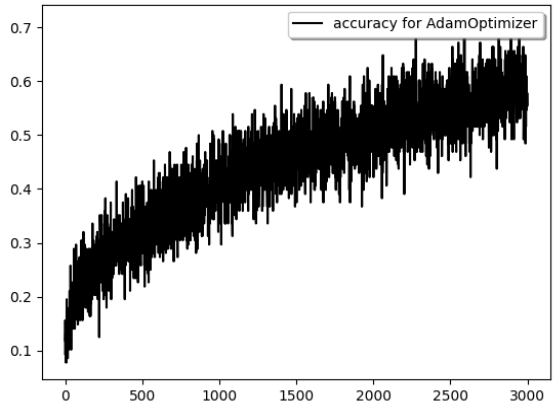

Figure 3.1

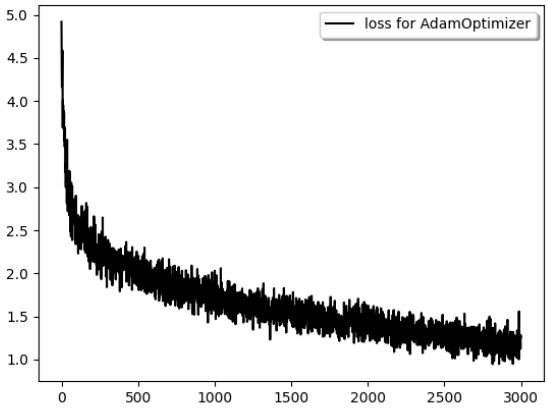

Figure 3.2

step 0, loss 4.91851, training accuracy 0.09375

step 100, loss 2.58724, training accuracy 0.234375

…

step 2800, loss 1.14305, training accuracy 0.609375

step 2900, loss 1.13062, training accuracy 0.625

test accuracy 0.489

Test 4: optimizer = AdamOptimizer, learning rate = $1e-3$, learning rate = 0.1

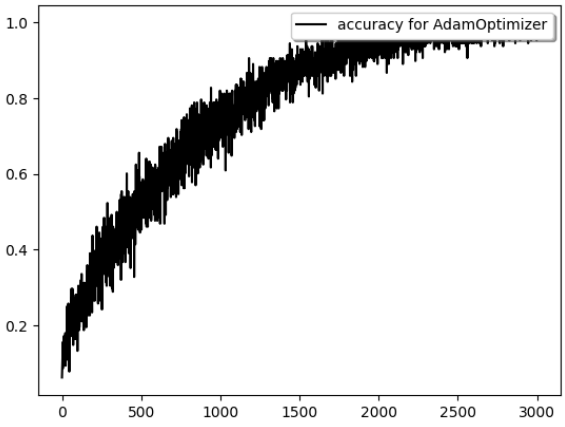

Figure 4.1

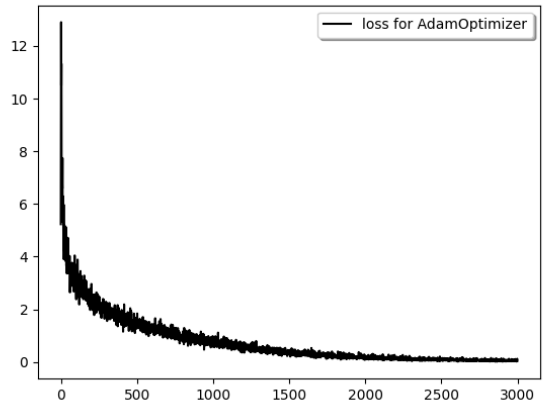

Figure 4.2

step 0, loss 5.23125, training accuracy 0.0625

step 100, loss 2.65417, training accuracy 0.242188

…

step 2800, loss 0.0405572, training accuracy 0.976562

step 2900, loss 0.0650353, training accuracy 0.960938

test accuracy 0.53

Test 5: optimizer = AdamOptimizer, learning rate = $1e-3$, learning rate = 0.3

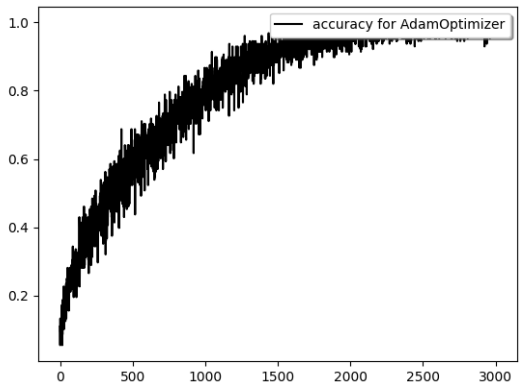

Figure 5.1

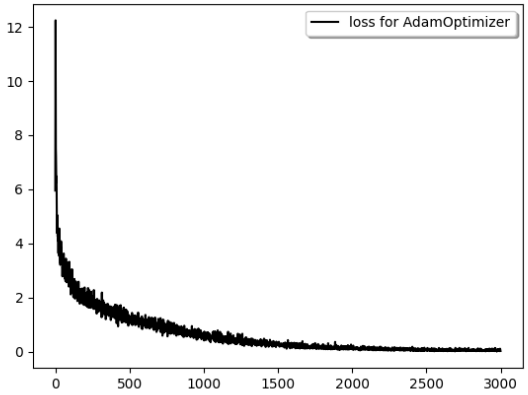

Figure 5.2

step 0, loss 5.95257, training accuracy 0.109375

step 100, loss 2.85569, training accuracy 0.242188

step 200, loss 2.04566, training accuracy 0.367188

…

step 2800, loss 0.0277834, training accuracy 0.992188

step 2900, loss 0.0713669, training accuracy 0.976562

test accuracy 0.532

Test 6: optimizer = AdamOptimizer, learning rate = $1e-3$, learning rate = 0.6

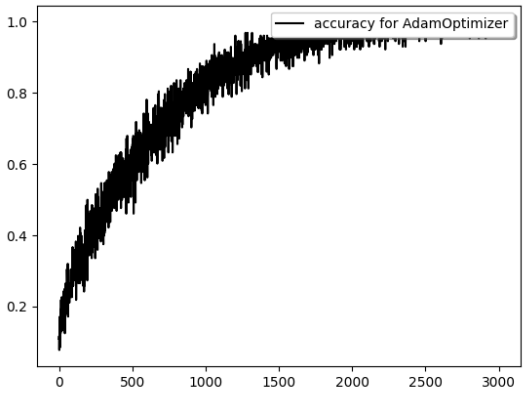

Figure 6.1

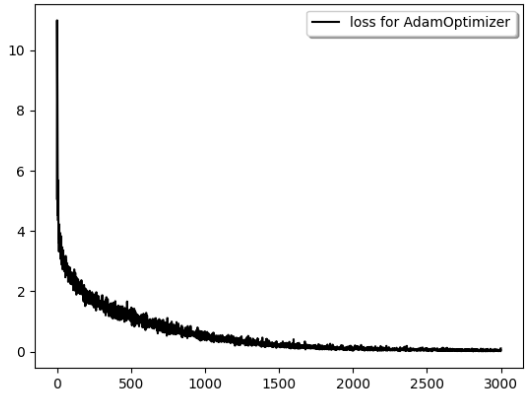

Figure 6.2

step 0, loss 5.07153, training accuracy 0.109375

step 100, loss 2.2398, training accuracy 0.359375

…

step 2800, loss 0.043352, training accuracy 0.984375

step 2900, loss 0.0719968, training accuracy 0.976562

test accuracy 0.539

Test 7: optimizer = GradientDescentOptimizer, learning rate = $1e-2$

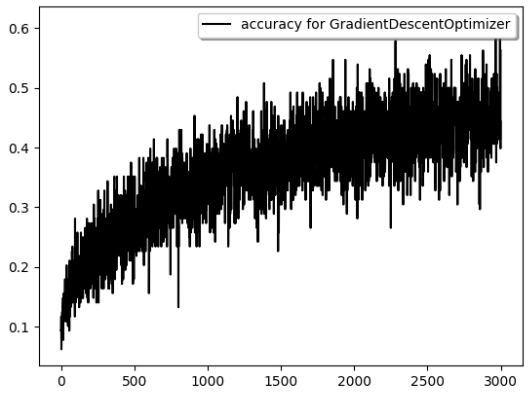

Figure 7.1

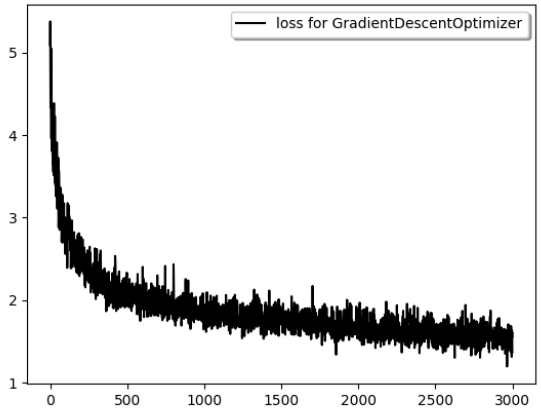

Figure 7.2

step 0, loss 5.08732, training accuracy 0.09375

step 100, loss 2.92382, training accuracy 0.140625

…

step 2800, loss 1.47761, training accuracy 0.460938

step 2900, loss 1.73298, training accuracy 0.421875

test accuracy 0.444

Test 8: optimizer = GradientDescentOptimizer, learning rate = $1e-3$

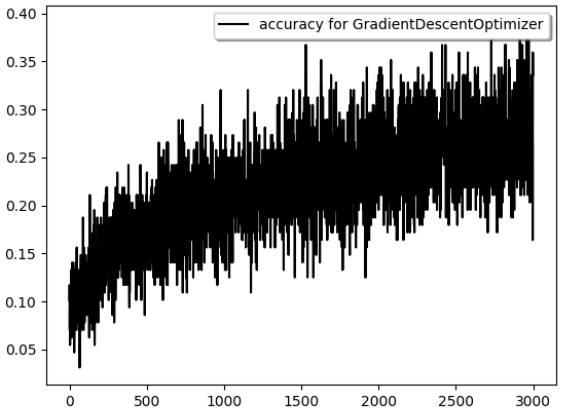

Figure 8.1

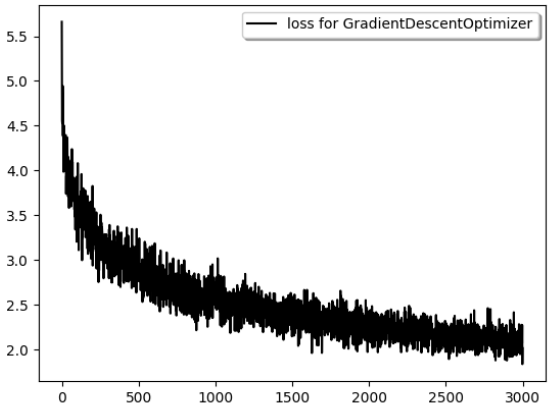

Figure 8.2

step 0, loss 5.6605, training accuracy 0.101562

step 100, loss 3.78212, training accuracy 0.132812

…

step 2800, loss 2.15316, training accuracy 0.296875

step 2900, loss 2.10632, training accuracy 0.242188

test accuracy 0.335

Test 9: optimizer = GradientDescentOptimizer, learning rate = $1e-4$

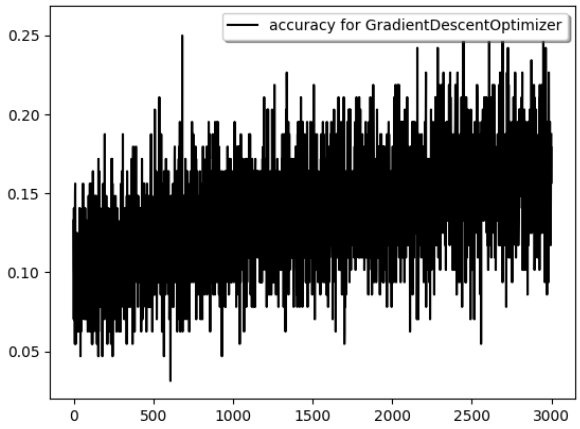

Figure 9.1

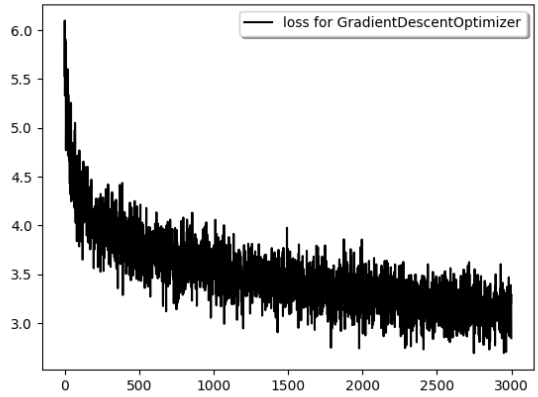

Figure 9.2

step 0, loss 5.53316, training accuracy 0.132812

step 100, loss 4.60991, training accuracy 0.109375

…

step 2800, loss 3.21105, training accuracy 0.203125

step 2900, loss 3.31617, training accuracy 0.140625

test accuracy 0.235

Test 10: optimizer = MomentumOptimizer, learning rate = $1e-2$, momentum values = 0.9

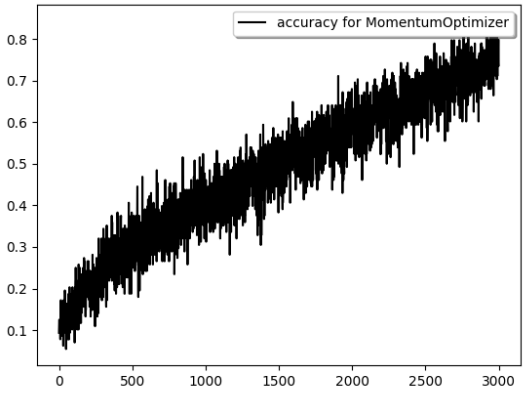

Figure 10.1

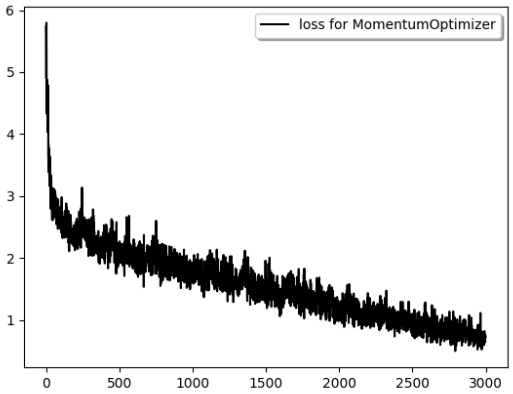

Figure 10.2

step 0, loss 5.73375, training accuracy 0.09375

step 100, loss 2.6112, training accuracy 0.179688

…

step 2800, loss 0.74255, training accuracy 0.742188

step 2900, loss 0.703186, training accuracy 0.75

test accuracy 0.494

Test 11: optimizer = MomentumOptimizer, learning rate = $1e-3$, momentum values = 0.9

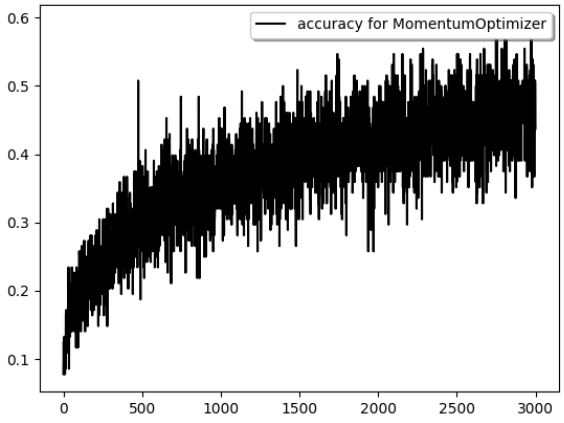

Figure 11.1

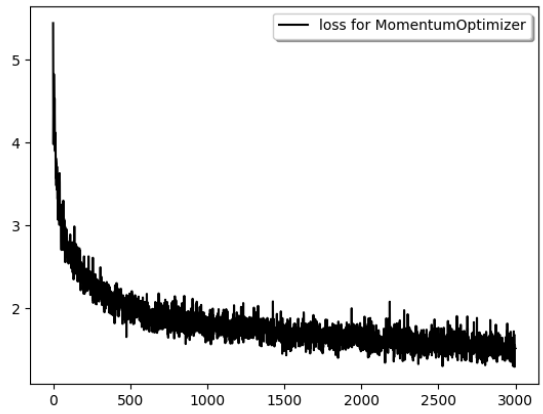

Figure 11.2

step 0, loss 5.44405, training accuracy 0.078125

step 100, loss 2.54106, training accuracy 0.179688

…

step 2800, loss 1.64176, training accuracy 0.40625

step 2900, loss 1.49669, training accuracy 0.476562

test accuracy 0.446

Test 12: optimizer = MomentumOptimizer, learning rate = $1e-4$, momentum values = 0.9

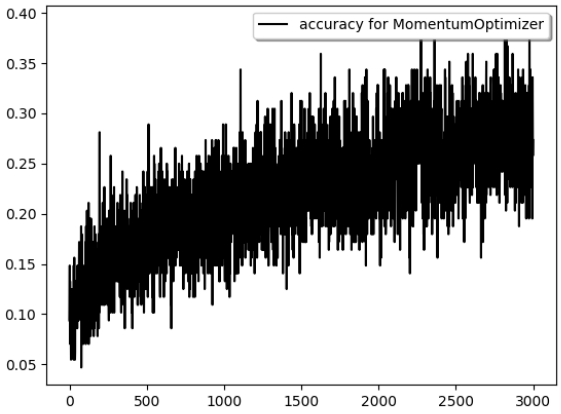

Figure 12.1

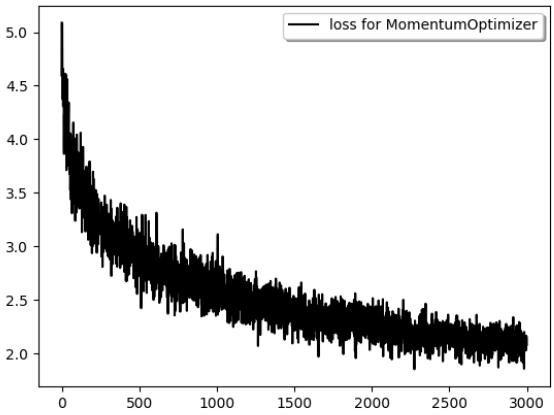

Figure 12.2

step 0, loss 5.08805, training accuracy 0.09375

step 100, loss 3.64254, training accuracy 0.125

…

step 2800, loss 2.12834, training accuracy 0.28125

step 2900, loss 2.08686, training accuracy 0.25

test accuracy 0.318

Test 13: optimizer = MomentumOptimizer, learning rate = $1e-3$, momentum values = 0.1

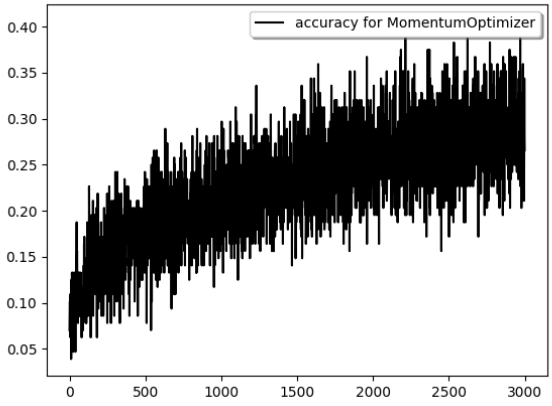

Figure 13.1

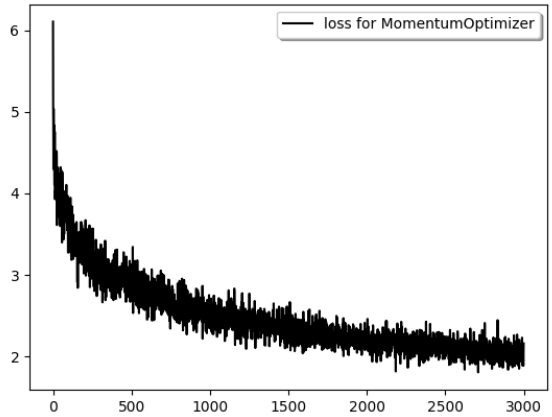

Figure 13.2

step 0, loss 6.10557, training accuracy 0.0703125

step 100, loss 3.27781, training accuracy 0.125

…

step 2800, loss 2.03405, training accuracy 0.234375

step 2900, loss 1.83842, training accuracy 0.375

test accuracy 0.351

Test 14: optimizer = MomentumOptimizer, learning rate = $1e-3$, momentum values = 0.3

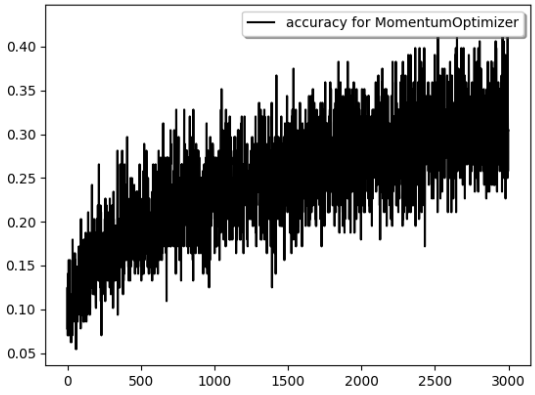

Figure 14.1

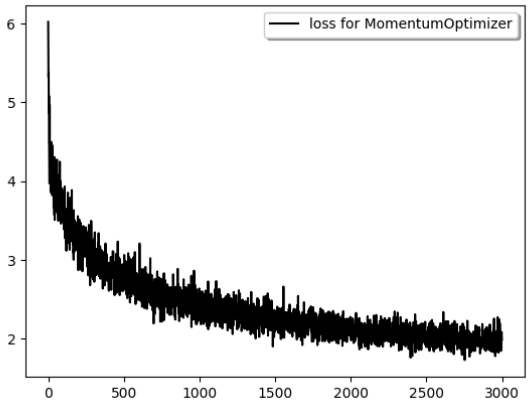

Figure 14.2

step 0, loss 6.02052, training accuracy 0.078125

step 100, loss 3.55658, training accuracy 0.164062

…

step 2800, loss 2.05859, training accuracy 0.28125

step 2900, loss 1.98615, training accuracy 0.351562

test accuracy 0.35

Test 15: optimizer = MomentumOptimizer, learning rate = $1e-3$, momentum values = 0.6

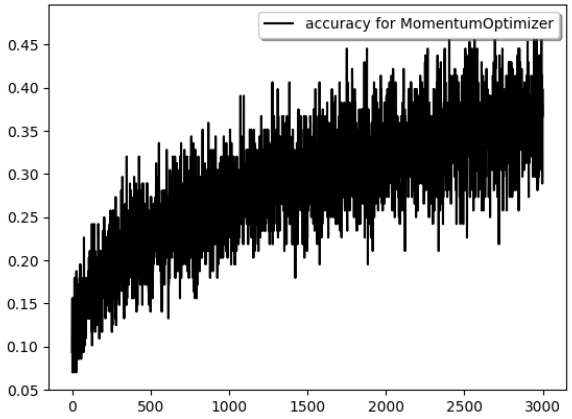

Figure 15.1

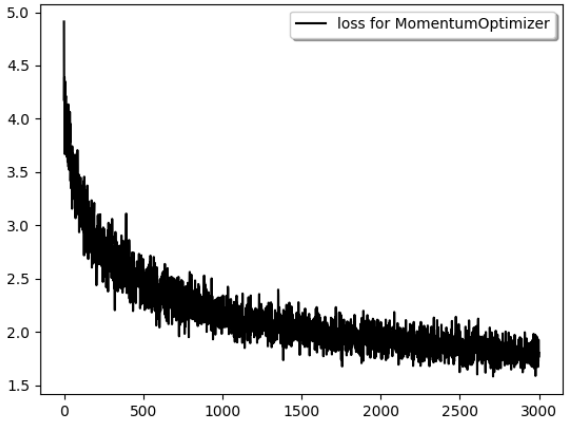

Figure 15.2

step 0, loss 4.9121, training accuracy 0.09375

step 100, loss 3.29391, training accuracy 0.15625

…

step 2800, loss 1.86872, training accuracy 0.328125

step 2900, loss 1.80693, training accuracy 0.367188

test accuracy 0.397

c) Visualize the Trained Network

Analysis: from the visualized figure we can see that the 32 filter maps focus on different parts of the original images which can help the CNN to capture the structures of the images. The change of the activations between the two layers indicates that the second activations may have more sparser but focused mappings.

1) **Visualize the first convolutional layer's weights:**

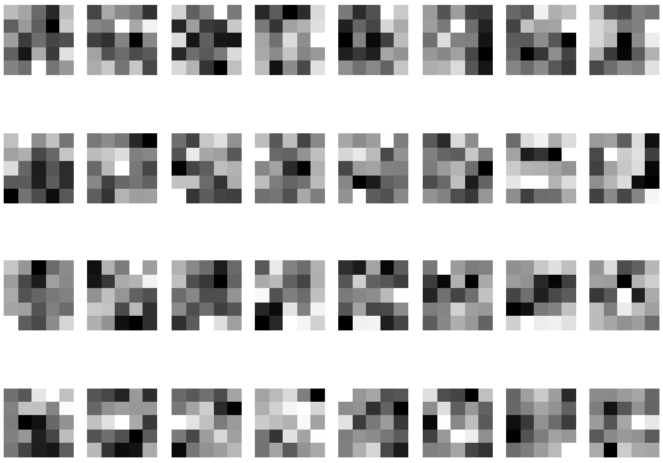

Figure 7

2) **Statistics of activations:**

   activation1: mean -0.0137963, variance 0.0160433

   activation2: mean -0.231102, variance 0.120792

2 Visualizing and Understanding Convolutional Networks

Motivation: due to several factors, Convolutional Networks (convnets) have gained much interests: (i) the availability of much larger training sets, with millions of labeled examples; (ii) powerful GPU implementations, making the training of very large models practical and (iii) better

model regularization strategies, such as Dropout (Hinton et al., 2012). However, there is still little insight into the internal operation and behavior of these complex models, or how they achieve such good performance. Thus, the authors develop a novel way to visualize the Convnets, and how to use the visualization to remedy the model.

Overview: the author presents a non-parametric view of invariance, showing top-down projections that reveal structures within each patch that stimulate a particular map back to the input pixel space.

Approach: the authors use standard fully supervised convnet models as defined by (LeCun et al., 1989) and (Krizhevsky et al., 2012).

- Deconvnet
    - Deconvolutional Network (deconvnet): map these activities back to the input pixel space, showing what input pattern originally caused a given activation in the feature maps. A deconvnet is attached to each of its layers to provide a continuous path back to image pixels.
        - Input image is presented to the convnet and features computed throughout the layers
        - To examine a given convnet activation, all other activations in the layer were set to zero and pass the feature maps as input to the attached deconvnet layer.
    - Unpooling:
        - obtain an approximate inverse by recording the locations of the maxima within each pooling region in a set of switch variables.
        - in the deconvnet, the unpooling operation uses these switches to place the reconstructions from the layer above into appropriate locations, preserving the structure of the stimulus.
    - Rectification: pass the reconstructed signal through a relu non-linearity.
    - Filtering: the deconvnet uses transposed versions of the same filters to flipp each filter vertically and horizontally.
    - Projecting down from higher layers uses the switch settings generated by the max pooling in the convnet on the way up. the reconstruction obtained from a single activation thus resembles a small piece of the original input image, with structures weighted according to their contribution toward to the feature activation.
- Convnet
    - Feature Visualization: Projecting each separately down to pixel space reveals the different structures that excite a given feature map, hence showing its invariance to input deformations. Alongside these visualizations we show the corresponding image patches.
    - Feature Evolution during Training: visualizes the progression during training of the strongest activation within a given feature map projected back to pixel space, capturing the source of raw image that results in the jump.

    - Feature Invariance: sample images being translated, rotated and scaled by varying degrees while looking at the changes in the feature vectors from the top and bottom layers of the model, relative to the untransformed feature.
- Architecture Selection: also assist with selecting good architectures in the first place.
- Occlusion Sensitivity: to answer if the model is truly identifying the location of the object in the image, or just using the surrounding context, it systematically occluding different portions of the input image with a gray square, and monitoring the output of the classifier. The visualization genuinely corresponds to the image structure that stimulates that feature map.
- Correspondence Analysis:
    - an intriguing possibility is that deep models might be implicitly computing them;
    - To explore this feature, the authors investigate the differences between feature vectors before and after occluding the same part of face in each image. The results show that the model does establish some degree of correspondence.

Experiments: the authors present experiments on ImageNet 2012. They explore the architecture of the model and the ability of features extraction layers to generalize to other datasets. And analyzed the discriminative features in each layer of Imagenet-pretrained model.

Conclusion: this work explored large convolutional neural network models, trained for image classification, in a number ways.

1. The authors presented a novel way to visualize the activity within the model. It reveals the features to be far from random, uninterpretable patterns;
2. showed how these visualization can be used to debug problems with the model to obtain better results;
3. demonstrated through a series of occlusion experiments that the model, while trained for classification, is highly sensitive to local structure in the image and is not just using broad scene context;
4. showed how the ImageNet trained model can generalize well to other datasets.

3 Build and Train an RNN on MNIST

a) Setup an RNN

- Number of nodes in the hidden layer: `nHidden = 128` for baseline experiments
- Learning rate: `learningRate = 1e-3`
- Number of iterations: `trainingIters = 50000`
- Cost (hint use softmax cross entropy with logits): cross-entropy
- Optimizer: AdamOptimizer (for baseline experiments) and GradientDescentOptimizer

b) How about using an LSTM or GRU

Analysis: 1) First, I run the experiments with setup showing in section 3a) and combine with RNN(original), LSTM, and GRU. Please refer to Test 1 to Test 3. From the results we can see the test accuracy: RNN(original) < LSTM < GRU. 2) Then, I tested GRU with GradientDescentOptimizer and found out the performance with AdamOptimizer is better. Please refer to Test 4. 3) Last, I tested GRU under AdamOptimizer with different numbers of nHidden. Please refer to Test 5 to Test 9. We can see from the final test results that more layers help to result better performance until a certain limit. According to our test results, the best performance happened at Test 8: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 512. When the hidden layers are too much, we will encounter data overfitting issues.

**all the experiments are run with tensorflow v1.14.0 on google colab*

b1)

Test 1: Optimizer = AdamOptimizer, RNN version = original, nHidden = 128

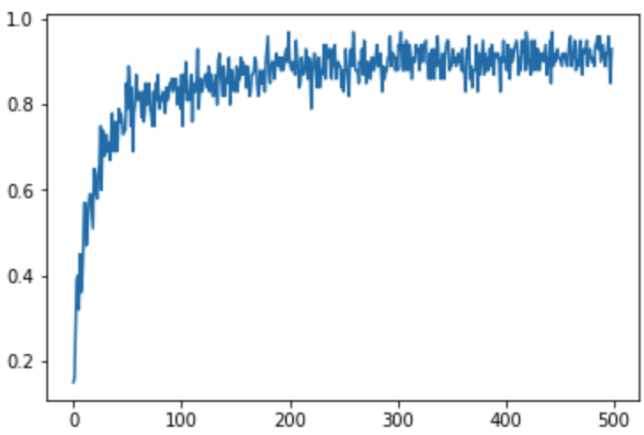

Figure 16.1: Test 1 accuracy

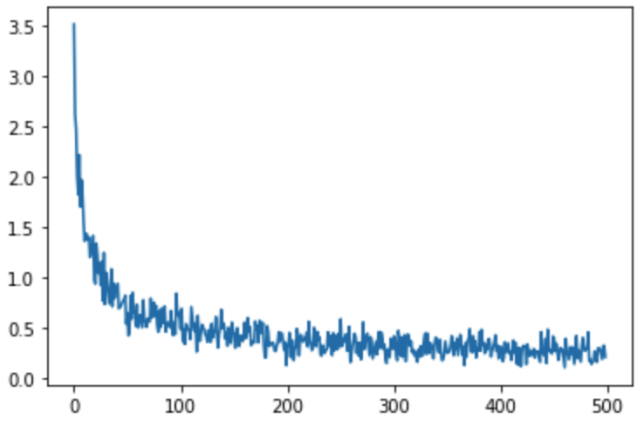

Figure 16.2: Test 1 Loss

Iter 1000, Minibatch Loss= 1.557570, Training Accuracy= 0.46000

Iter 2000, Minibatch Loss= 0.982637, Training Accuracy= 0.65000

…

Iter 48000, Minibatch Loss= 0.275612, Training Accuracy= 0.91000

Iter 49000, Minibatch Loss= 0.204110, Training Accuracy= 0.92000

Testing Accuracy: 0.9132

Test 2: Optimizer = AdamOptimizer, RNN version = with LSTM, nHidden = 128

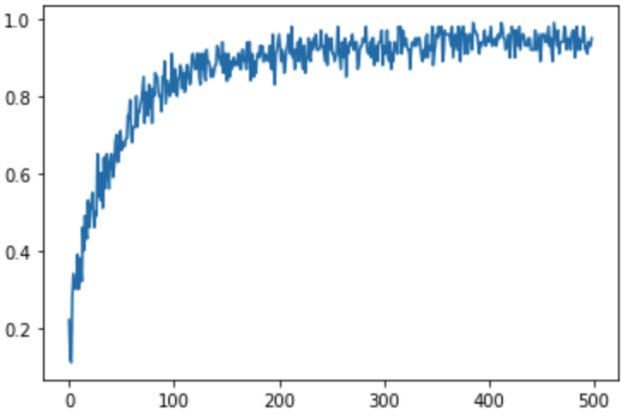

Figure 17.1: Test 2 accuracy

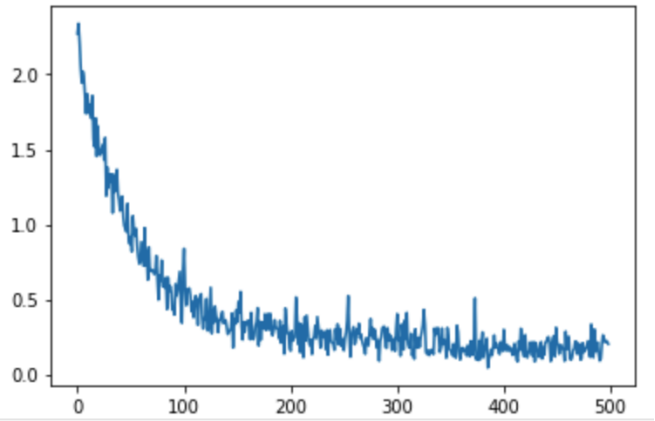

Figure 17.2: Test 2 loss

Iter 1000, Minibatch Loss= 1.869787, Training Accuracy= 0.30000

Iter 2000, Minibatch Loss= 1.656205, Training Accuracy= 0.46000

…

Iter 48000, Minibatch Loss= 0.222113, Training Accuracy= 0.96000

Iter 49000, Minibatch Loss= 0.139053, Training Accuracy= 0.94000

Testing Accuracy: 0.9333

Test 3: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 128

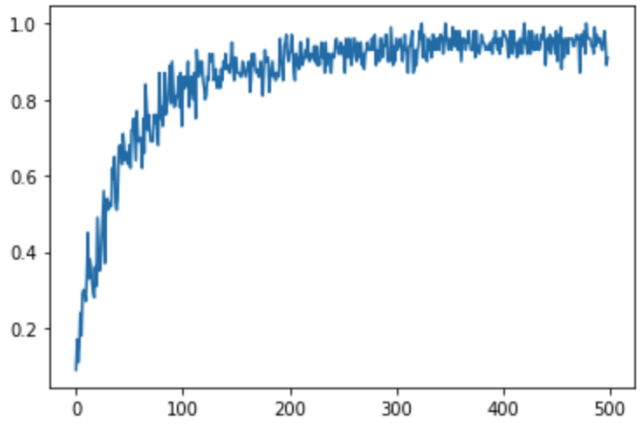

Figure 18.1: Test 3 accuracy

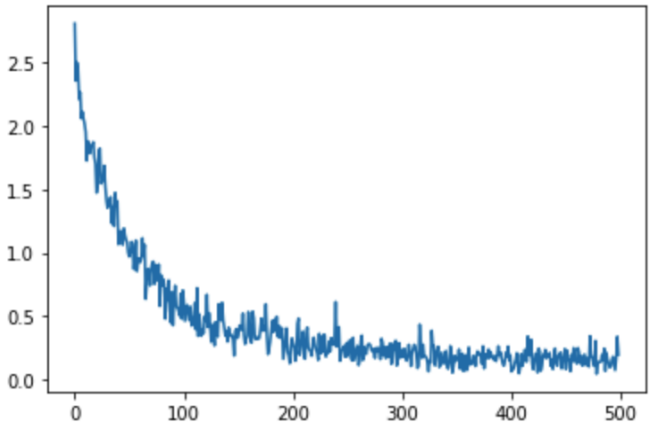

Figure 18.2: Test 3 loss

Iter 1000, Minibatch Loss= 2.015705, Training Accuracy= 0.27000

Iter 2000, Minibatch Loss= 1.699272, Training Accuracy= 0.31000

…

Iter 48000, Minibatch Loss= 0.140012, Training Accuracy= 0.97000

Iter 49000, Minibatch Loss= 0.130273, Training Accuracy= 0.96000

Testing Accuracy: 0.9556

b2)

Test 4: Optimizer = GradientDescentOptimizer, RNN version = with GRU, nHidden = 128

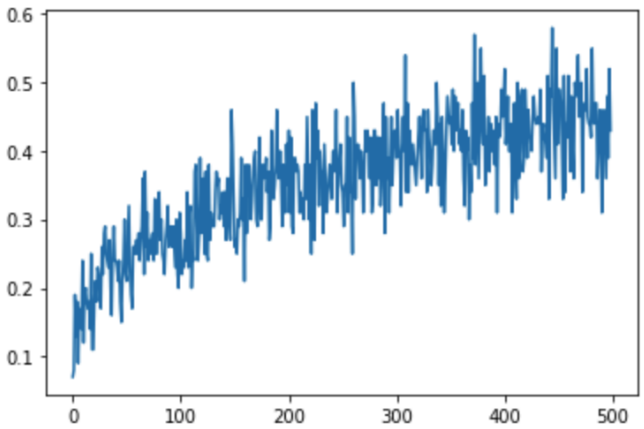

Figure 19.1: Test 4 accuracy

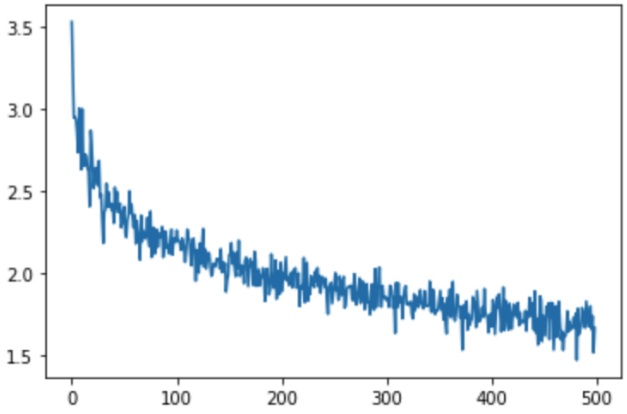

Figure 19.2: Test 4 loss

Iter 1000, Minibatch Loss= 2.634958, Training Accuracy= 0.24000

Iter 2000, Minibatch Loss= 2.777790, Training Accuracy= 0.11000

…

Iter 48000, Minibatch Loss= 1.668823, Training Accuracy= 0.44000

Iter 49000, Minibatch Loss= 1.670796, Training Accuracy= 0.46000

Testing Accuracy: 0.461

b3)

Test 5: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 64

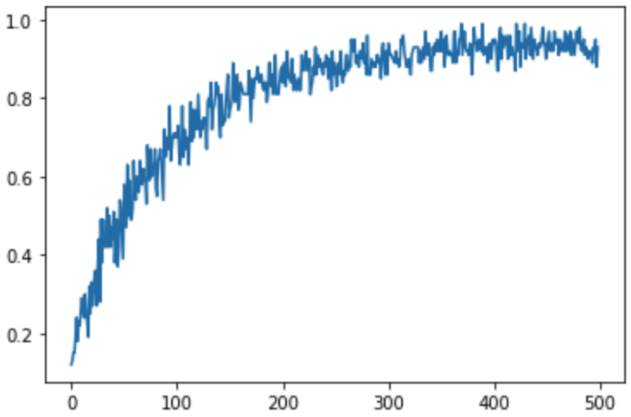

Figure 20.1: Test 5 accuracy

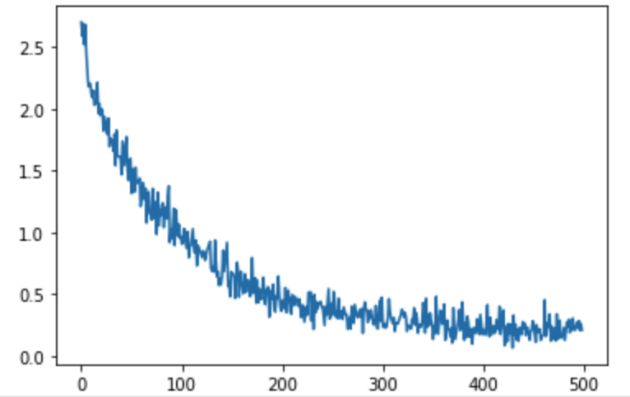

Figure 20.2: Test 5 loss

Iter 1000, Minibatch Loss= 2.204432, Training Accuracy= 0.26000

Iter 2000, Minibatch Loss= 1.947229, Training Accuracy= 0.33000

…

Iter 48000, Minibatch Loss= 0.167294, Training Accuracy= 0.95000

Iter 49000, Minibatch Loss= 0.306579, Training Accuracy= 0.90000

Testing Accuracy: 0.939

Test 6: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 32

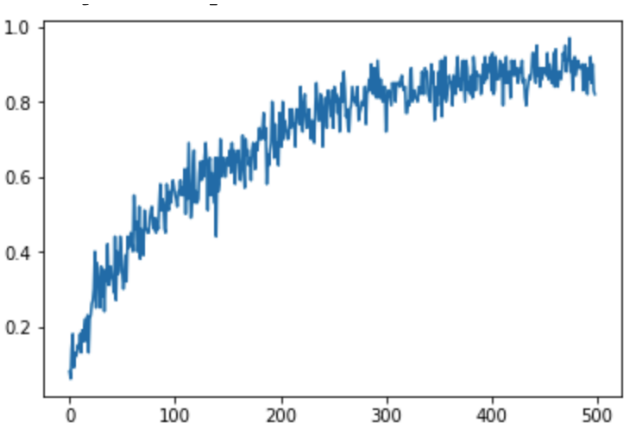

Figure 21.1: Test 6 accuracy

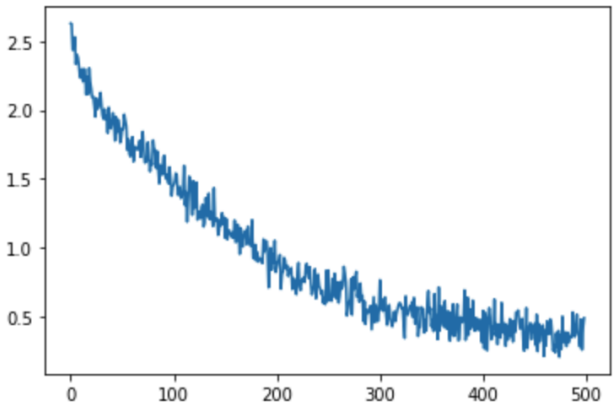

Figure 22.2: Test 6 loss

Iter 1000, Minibatch Loss= 2.238473, Training Accuracy= 0.14000

Iter 2000, Minibatch Loss= 2.205337, Training Accuracy= 0.22000

…

Iter 48000, Minibatch Loss= 0.314533, Training Accuracy= 0.92000

Iter 49000, Minibatch Loss= 0.349117, Training Accuracy= 0.90000

Testing Accuracy: 0.899

Test 7: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 256

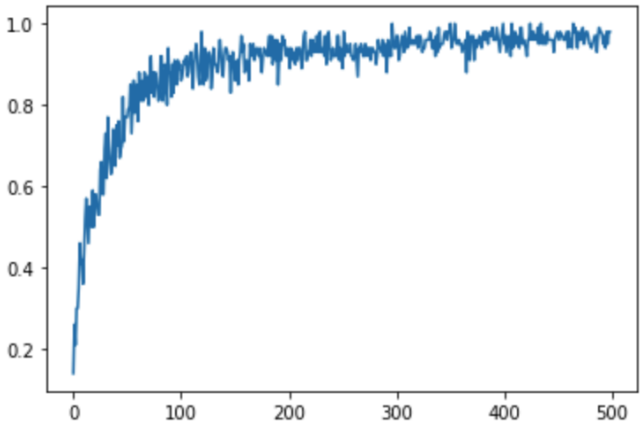

Figure 23.1: Test 7 accuracy

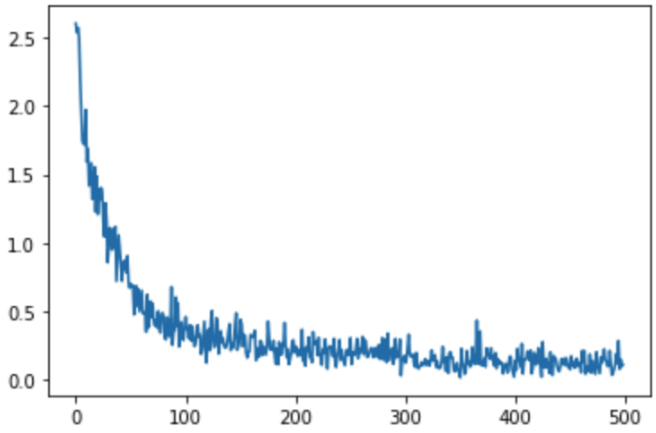

Figure 23.2: Test 7 loss

Iter 1000, Minibatch Loss= 1.972682, Training Accuracy= 0.36000

Iter 2000, Minibatch Loss= 1.486277, Training Accuracy= 0.50000

…

Iter 48000, Minibatch Loss= 0.124569, Training Accuracy= 0.95000

Iter 49000, Minibatch Loss= 0.047546, Training Accuracy= 0.98000

Testing Accuracy: 0.9569

Test 8: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 512

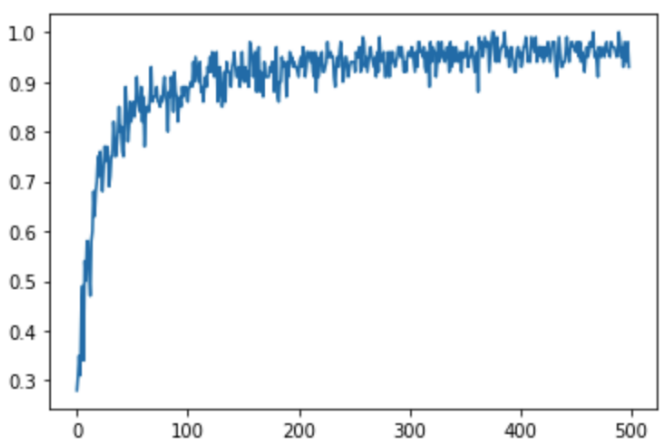

Figure 24.1: Test 8 accuracy

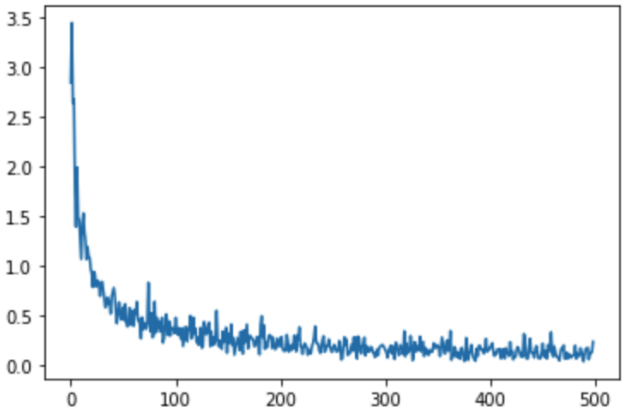

Figure 24.2: Test 8 loss

Iter 1000, Minibatch Loss= 1.260026, Training Accuracy= 0.58000

Iter 2000, Minibatch Loss= 0.969509, Training Accuracy= 0.75000

…

Iter 48000, Minibatch Loss= 0.094648, Training Accuracy= 0.96000

Iter 49000, Minibatch Loss= 0.033031, Training Accuracy= 1.00000

Testing Accuracy: 0.9621

Test 9: Optimizer = AdamOptimizer, RNN version = with GRU, nHidden = 1024

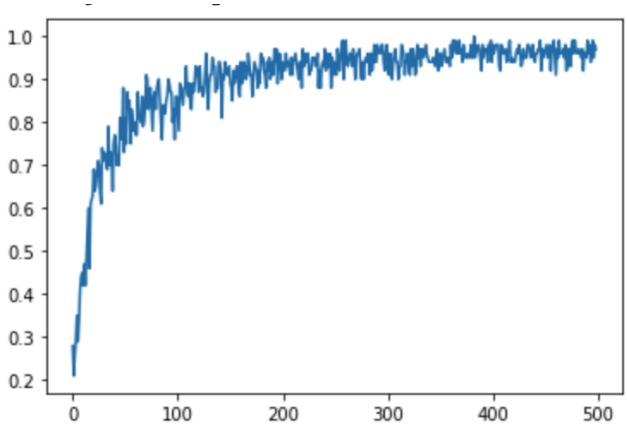

Figure 25.1: Test 9 accuracy

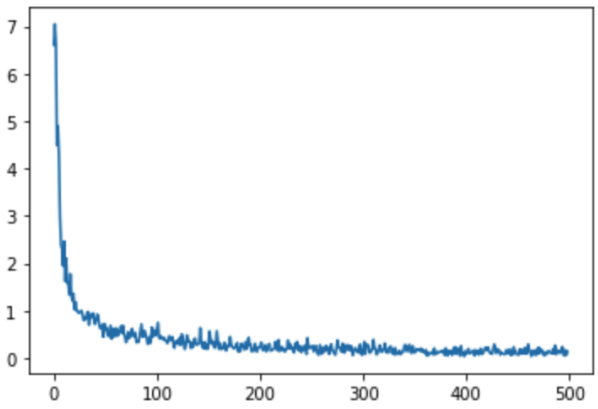

Figure 25.2: Test 9 loss

Iter 1000, Minibatch Loss= 1.945579, Training Accuracy= 0.45000

Iter 2000, Minibatch Loss= 1.361139, Training Accuracy= 0.63000

…

Iter 48000, Minibatch Loss= 0.137348, Training Accuracy= 0.97000

Iter 49000, Minibatch Loss= 0.165064, Training Accuracy= 0.95000

Testing Accuracy: 0.9588

c) Compare against the CNN

Similarities: both CNN and RNN are deep neural networks and can be used for image recognition tasks or some other similar tasks. Both can work with some classical optimizers and methods.

Differences: from our test results from assignment 1 and 2, we can see that for the image recognition task on MNIST dataset, CNN achieves a higher accuracy with much less training steps (iterations), while RNN can only achieve a lower accuracy after 50000 iterations. Thus, it indicates that CNN is more powerful than RNN for image recognition tasks. Based on some references [2], here are the breakdown comparisons of CNN and RNN:

CNN:

Advantages:

Very High accuracy in image recognition problems.

Automatically detects the important features without any human supervision.

Weight sharing.

Disadvantages:

CNN does not encode the position and orientation of objects.

Lack of ability to be spatially invariant to the input data.

Lots of training data is required.

RNN:

Advantages:

An RNN remembers each and every information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.

Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages:

Gradient vanishing and exploding problems.

Training an RNN is a very difficult task.

It cannot process very long sequences if using tanh or relu as an activation function.

References

[1] Why 0.9? Towards Better Momentum Strategies in Deep Learning. TowardsDataScience. 02/21
https://towardsdatascience.com/why-0-9-towards-better-momentum-strategies-in-deep-learning-827408503650

[2] Difference between ANN, CNN and RNN. GeeksforGeeks. 08/2022
https://www.geeksforgeeks.org/difference-between-ann-cnn-and-rnn/