

ELEC576 – Fall 2022

Assignment2

Student's Name: Hsuan-You (Shaun) Lin / Net ID: hl116

Student ID: S01435165

1. Visualizing a CNN with CIFAR10:

(a) CIFAR10 Dataset

There are 10 classes in the CIFAR10 dataset, which are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The size of each image is 28*28. There are 1000 images per class in the training dataset, and for the test dataset, there are 100 images per class. here I choose 50 as my batch size.

```
84
85     ntrain = 1000 # per class
86     ntest = 100 # per class
87     nclass = 10 # number of classes
88     imsize = 28 # image size 28 * 28
89     nchannels = 1
90     batchsize = 50
```

Figure 1. Python code for CIFAR10 dataset

(b.1) Train LeNet5 on CIFAR10:

In Assignment 1, we learned how to define the functions `weight_variable()`, `bias_variable()`, `conv2d()` and `max_pool_2x2()`, we also learned that ReLU is the best activation function for deep learning and that the best performance can be obtained with the Adam optimizer. Here I not only used Adam optimizer, I also checked Momentum (with 0.5 momentum) and Adagrad optimization methods, and I tested different learning rates for each method, *Figure 2* shows my python code for implementing LeNet5.

```

120 # -----
121 # model
122
123 # First Convolutional layer with kernel 5 x 5 and 32 filter maps followed by ReLU
124 W_conv1 = weight_variable([5, 5, 1, 32])
125 b_conv1 = bias_variable([32])
126 h_conv1 = tf.nn.relu(conv2d(tf_data, W_conv1) + b_conv1)
127 h_pool1 = max_pool_2x2(h_conv1) # Max Pooling layer subsampling by 2
128
129 # Second Convolutional layer with kernel 5 x 5 and 64 filter maps followed by ReLU
130 W_conv2 = weight_variable([5, 5, 32, 64])
131 b_conv2 = bias_variable([64])
132 h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
133 h_pool2 = max_pool_2x2(h_conv2) # Max Pooling layer subsampling by 2
134
135 # First Fully Connected layer that has input 7 * 7 * 64 and output 1024
136 W_fc1 = weight_variable([7 * 7 * 64, 1024])
137 b_fc1 = bias_variable([1024])
138 h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
139 h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
140
141 # Dropout
142 keep_prob = tf.placeholder(tf.float32)
143 h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
144
145 # Second Fully Connected layer that has input 1024 and output 10 (for the classes)
146 # Softmax layer (Softmax Regression + Softmax Non-linearity)
147 W_fc2 = weight_variable([1024, 10])
148 b_fc2 = bias_variable([10])
149 h_fc2 = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
150 # -----
151 # loss
152 # set up the loss, optimization, evaluation, and accuracy
153 cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels = tf_labels, logits = h_fc2))
154
155 # Momentum_optimizer = tf.train.MomentumOptimizer(learning_rate = 1e-3, 0.5).minimize(cross_entropy)
156 # Adagrad_optimizer = tf.train.AdagradOptimizer(learning_rate = 1e-3, 0.5).minimize(cross_entropy)
157 Adam_optimizer = tf.train.AdamOptimizer(learning_rate = 1e-3).minimize(cross_entropy)
158 correct_prediction = tf.equal(tf.argmax(h_fc2, 1), tf.argmax(tf_labels, 1))
159 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
160
161 # -----
162 # optimization
163 sess.run(tf.initialize_all_variables())
164 batch_xs = np.zeros((batchsize, imsize, imsize, nchannels))
165 batch_ys = np.zeros((batchsize, nclass))
166
167 losses_list = []
168 accs_list = []
169 for i in range(3500): # original 5500, try a small iteration size once it works then continue
170     perm = np.arange(ntrain * nclass)
171     np.random.shuffle(perm)
172     for j in range(batchsize):
173         batch_xs[j, :, :, :] = Train[perm[j], :, :, :]
174         batch_ys[j, :] = LTrain[perm[j], :]
175     loss = cross_entropy.eval(feed_dict = {tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5})
176     acc = accuracy.eval(feed_dict = {tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5})
177
178     losses_list.append(loss)
179     accs_list.append(acc)
180     if i % 100 == 0:
181         # calculate train accuracy and print it
182         print('step %d, loss %g, training accuracy %g' % (i, loss, acc))
183     # Momentum_optimizer.run(feed_dict = {tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5}) # dropout only during training
184     # Adagrad_optimizer.run(feed_dict = {tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5}) # dropout only during training
185     Adam_optimizer.run(feed_dict = {tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5}) # dropout only during training
186

```

Figure 2. Python code for implement a LeNet5

(b.2) Plot train/test accuracy and train loss:

1. Training with Momentum (with 0.5 momentum) optimizer:

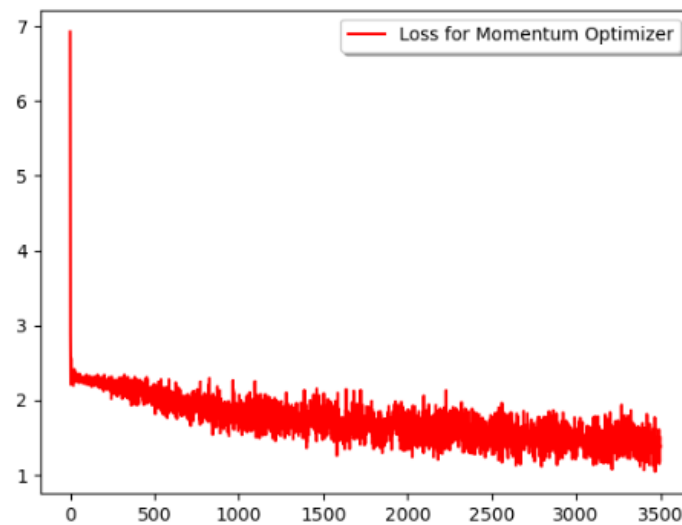


Figure 3. Plot training loss iteration over 3500 steps (learning rate = $1e-2$)

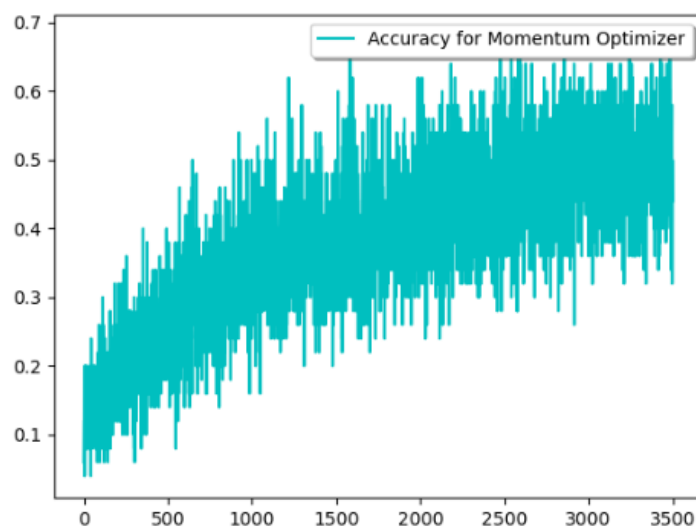


Figure 4. Plot training accuracy iteration over 3500 steps (learning rate = $1e-2$)

```
step 3300, loss 1.30091, training accuracy 0.6  
step 3400, loss 1.45852, training accuracy 0.5  
test accuracy 0.483
```

Figure 5. Test accuracy (learning rate = $1e-2$)

2. Training with Adagrad optimizer

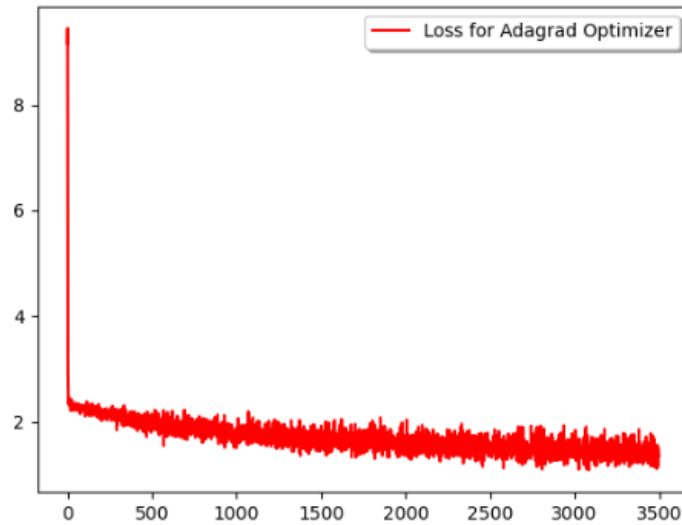


Figure 6. Plot training loss iteration over 3500 steps (learning rate = $1e-2$)

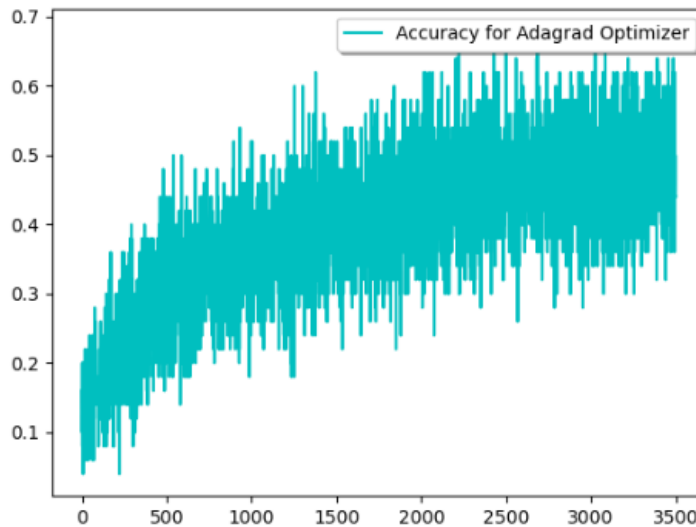


Figure 7. Plot training accuracy iteration over 3500 steps (learning rate = $1e-2$)

```
step 3300, loss 1.40588, training accuracy 0.52  
step 3400, loss 1.13355, training accuracy 0.66  
test accuracy 0.471
```

Figure 8. Test accuracy (learning rate = $1e-2$)

3. Training with Adam optimizer

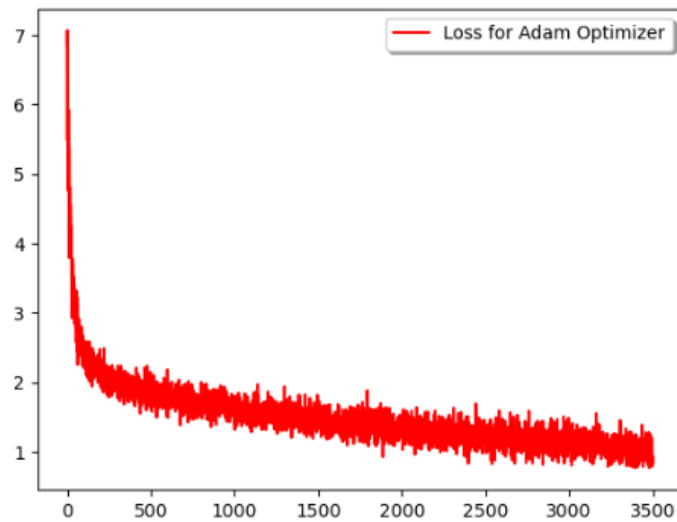


Figure 9. Plot training loss iteration over 3500 steps (learning rate = $1e-4$)

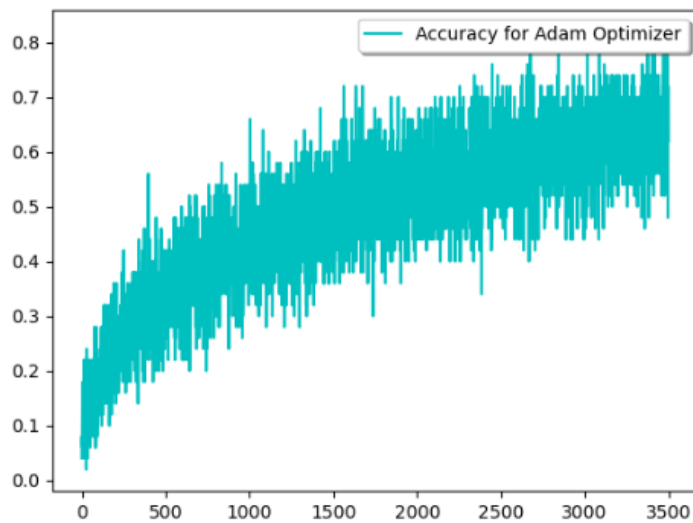


Figure 10. Plot training accuracy iteration over 3500 steps (learning rate = $1e-4$)

```
step 3300, loss 1.11728, training accuracy 0.62  
step 3400, loss 1.25015, training accuracy 0.56  
test accuracy 0.529
```

Figure 11. Test accuracy (learning rate = $1e-4$)

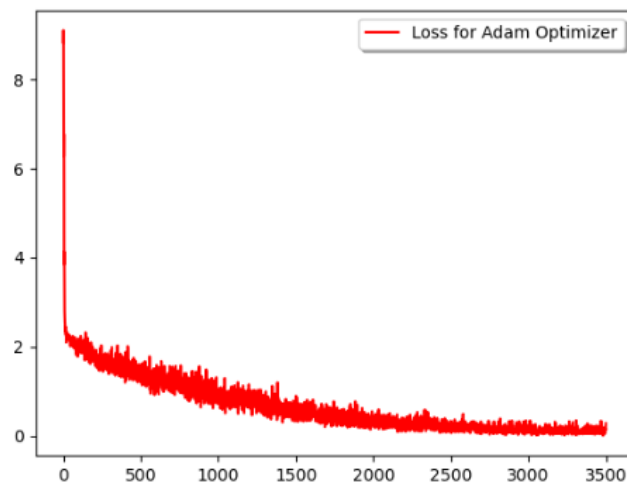


Figure 12. Plot training loss iteration over 3500 steps (learning rate = $1e-3$)

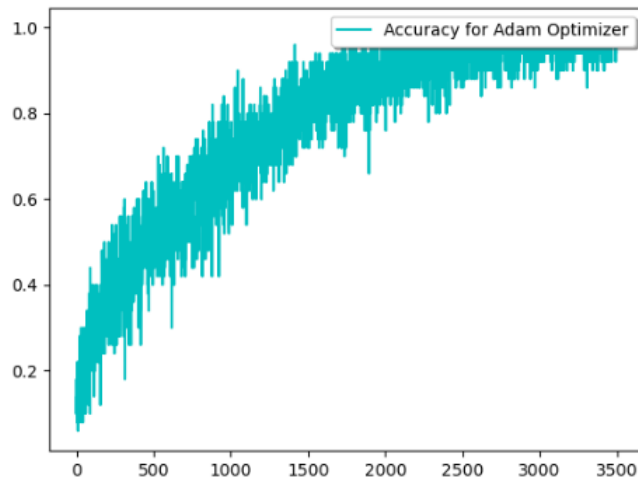


Figure 13. Plot training accuracy iteration over 3500 steps (learning rate = $1e-3$)

```
step 3300, loss 0.136642, training accuracy 0.9
step 3400, loss 0.0879621, training accuracy 0.98
test accuracy 0.57
```

Figure 14. Test accuracy (learning rate = $1e-3$)

My Observation:

From the above training results, we checked three types of optimization methods. Apparently, higher learning rate could make the iteration more faster, but if we chose $1e-4$ as learning rate, it will be too slow decrease loss and converge, and if we chose $1e-1$ as our learning rate, it will be too large to decrease the loss. Thus, we chose Adam optimization method, and set $1e-3$ as our learning rate for the final training, we obtained 0.57 test accuracy, which is the best than the other two optimization methods.

(c) Visualize the Trained Network:

Figure 15 shows my python code for visualize the trained network.

```
# first convolutional layer's weights
first_weight = W_conv1.eval()
# The statistics of the activations in the first convolutional layers on test images.
activation1 = h_conv1.eval(feed_dict={tf_data: Test, tf_labels: LTest, keep_prob: 1.0})
# The statistics of the activations in the second convolutional layers on test images.
activation2 = h_conv2.eval(feed_dict={tf_data: Test, tf_labels: LTest, keep_prob: 1.0})
mean1 = np.mean(np.array(activation1))
variance1 = np.var(np.array(activation1))
mean2 = np.mean(np.array(activation2))
variance2 = np.var(np.array(activation2))

# Visualize the first convolutional layer's weights
fig = plt.figure()
for i in range(32):
    ax = fig.add_subplot(4, 8, 1 + i)
    ax.imshow(first_weight[:, :, 0, i], cmap='gray')
    plt.axis('off')
plt.show()

# Calculate the statistics of the activations in the convolutional layers on test images.
print("Activation1: mean %g, variance %g" % (mean1, variance1))
print("Activation2: mean %g, variance %g" % (mean2, variance2))
```

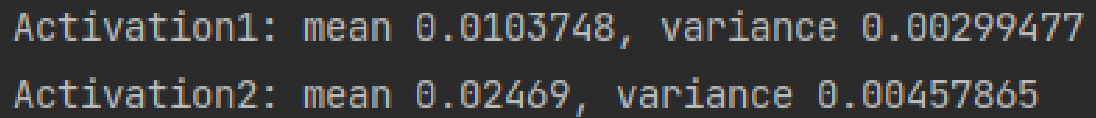
Figure 15. Python code for first convolutional layer's weights and the statistics of the activations for first/second convolutional layers

The weights of first convolutional layer are visualized as below.



Figure 16. First convolutional layer's weights

The statistics of the activations (mean and variance) in the convolutional layers as shown in *Figure 17*.



```
Activation1: mean 0.0103748, variance 0.00299477
Activation2: mean 0.02469, variance 0.00457865
```

Figure 17. The statistics of the activations in the first/second convolutional layers on test images.

My Observation:

From the Figure 16, we have 32 filter maps that could make the CNN capture the better structures of images. For activation on the test images, we get the mean: 0.02469 and variance: 0.00457865 for second layer, which is higher than first layer, as we go deeper in the layers, the activations become increasingly abstract and less visually interpretable.

2. Visualizing and Understanding Convolutional Networks:

My Observation:

This paper explored the issues of why large convolutional networks models could perform better, and how to improved them.

First of all, authors present a novel visualization method to visualize the activity within the model, which shows the function of intermediate feature layers and the operation of the classifier. They propose to use a multi-layered deconvolutional network to map these activities back to the input pixel space, which namely Deconvnet.

Secondly, Deconvnet will attached to each of its layers, then input image in convnet model send to Deconvnet model to recontruct the small region of original input, it is like a small piece of the original input image, its structure is weighted according to its contribution to feature activation, which means parts of the input image are discriminative. The activity in each layer, Unpooling, Rectification, and Filtering are repeated used until input pixel space is reached.

In addition, the authors proposed the problem of occlusion sensitivity. It systematically blocked different parts of the input image and monitors the classification results . The results show that if the object is blocked in the image the output results will perform poorly. Also an experiment was presented in this paper for ImageNet 2012, he explores the architecture of the model and how the ImageNet trained model can generalize well to other datasets.

To summarize, this paper uses Deconvnet to visualize feature activation and is useful for Feature Visualization, Feature Evolution during Training, Feature Invariance, and Architecture Selection. He also demonstrated how models can be remediated by visualization. He presents experiments on Occlusion Sensitivity and ImageNet 2012, where the author suggests that deep models may be implicitly computing relationships between specific object parts in different images, which can be useful for Correspondence Analysis.

3. Build and Train an RNN on MNIST:

(a) Setup an RNN:

In this section, due to the previous experience, I set the learning rate to $1e-3$, which has better performance results, and I use 128 for the number of nodes in each hidden layer, also the training iteration is set to be 100000, I choose the softmax cross entropy with logits in the cost, which is loss. In the end, I still used three optimization methods to compare results, which is MomentumOptimizer, AdagradOptimizer and AdamOptimizer, those setting as shown in the *Figure 18*, and the training result as shown in *Figure 19~21*.

```
10 # Load MNIST dataset,
11 # tensorflow 2.10.0 doesn't have examples.tutorials.mnist,
12 # we need to download it from earlier version.
13 from tensorflow.examples.tutorials.mnist import input_data
14 mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
15
16 learningRate = 1e-3
17 trainingIters = 100000
18 batchSize = 10
19 displayStep = 100
20
21 nInput = 28 #we want the input to take the 28 pixels
22 nSteps = 28 #every 28
23 nHidden = 128 #number of neurons for the RNN
24 nClasses = 10
25
26
27 def RNN(x, weights, biases):
28     x = tf.transpose(x, [1, 0, 2])
29     x = tf.reshape(x, [-1, nInput])
30     x = tf.split(value=x, num_or_size_splits=nSteps, axis=0)
31
32     # uncomment, if you want to use RNN
33     rnnCell = rnn_cell.BasicRNNCell(nHidden)
34     outputs, states = tf.nn.nn.static_rnn(rnnCell, x, dtype=tf.float32)
35
36
37
38
39
40
41
42
43
44
45
46
47
48 #optimization
49 #create the cost, optimization, evaluation, and accuracy
50 #for the cost softmax_cross_entropy_with_logits seems really good
51 cost = tf.losses.softmax_cross_entropy(logits=pred, onehot_labels=y)
52
53 optimizer = tf.train.MomentumOptimizer(learning_rate=learningRate, momentum=0.5).minimize(cost)
54 # optimizer = tf.train.AdagradOptimizer(learning_rate=learningRate).minimize(cost)
55 # optimizer = tf.train.AdamOptimizer(learning_rate=learningRate).minimize(cost)
56
57 correctPred = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
58 accuracy = tf.reduce_mean(tf.cast(correctPred, tf.float32))
```

Figure 18. Python code for MNIST dataset RNN training setup

1. RNN training with Momentum optimizer

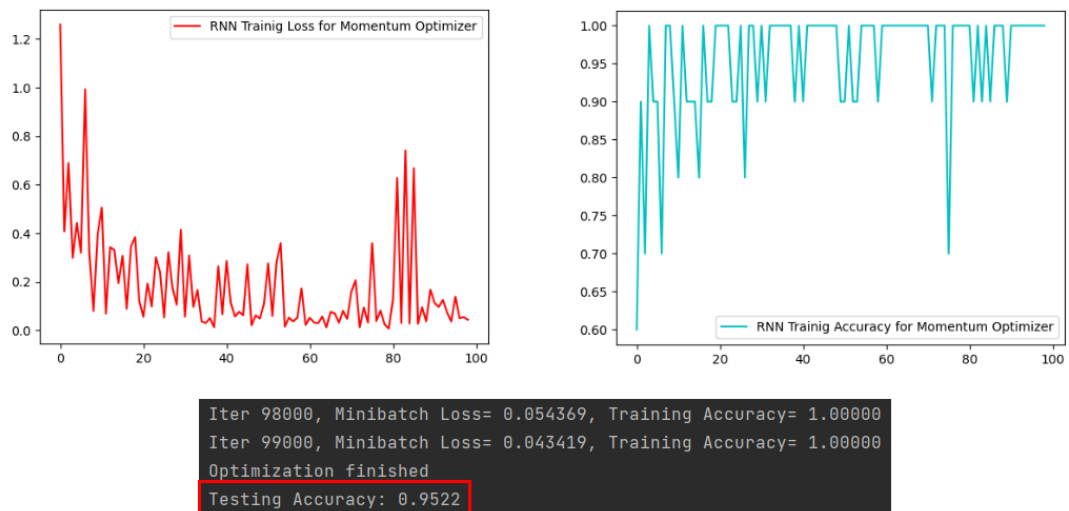


Figure 19. Training result for Momentum optimizer

2. RNN training with Adagrad optimizer

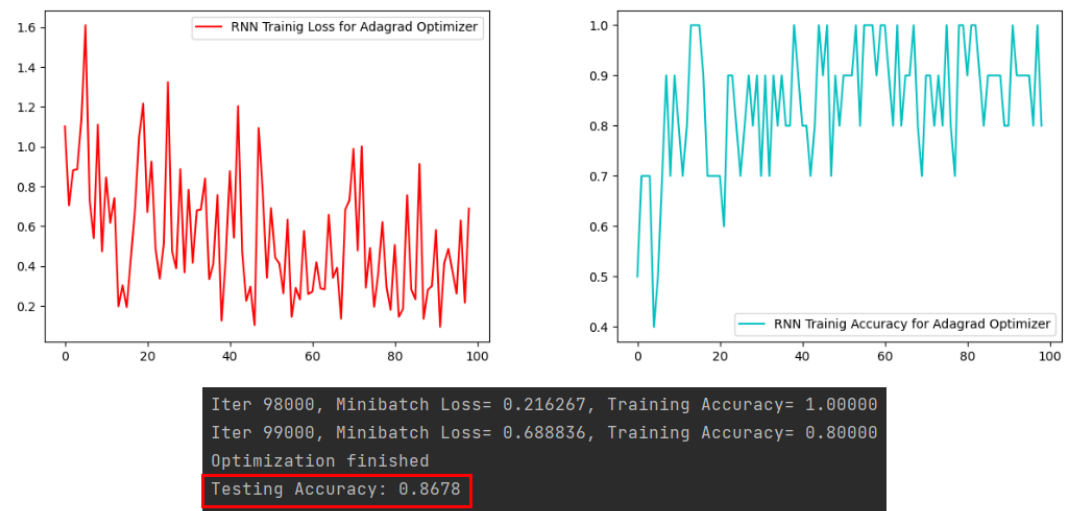


Figure 20. Training result for Adagrad optimizer

3. RNN training with Adam optimizer

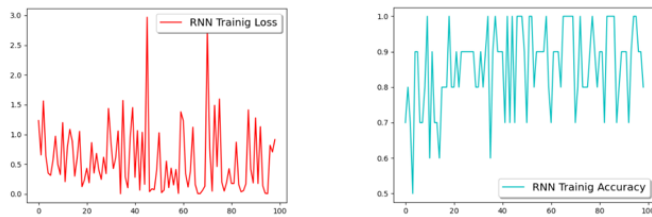


Figure 21. Training result for Adam optimizer

(b) How about using an LSTM or GRU:

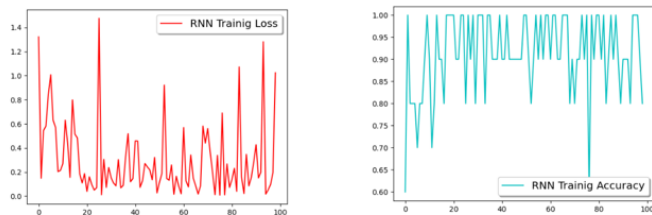
1. RNN (Recurrent neural network) - Default

Training result: RNN (with 256 hidden units)



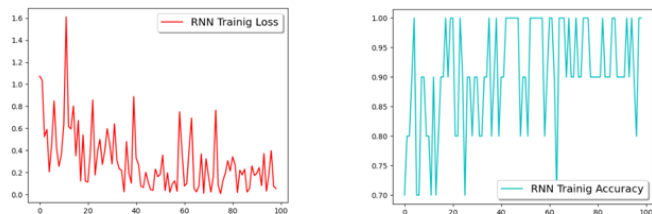
```
Iter 98000, Minibatch Loss= 0.118076, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.166437, Training Accuracy= 0.90000  
Optimization finished  
Testing Accuracy: 0.8938
```

Training result: RNN (with 128 hidden units)



```
Iter 98000, Minibatch Loss= 0.196898, Training Accuracy= 0.90000  
Iter 99000, Minibatch Loss= 1.023709, Training Accuracy= 0.80000  
Optimization finished  
Testing Accuracy: 0.9177
```

Training result: RNN (with 64 hidden units)

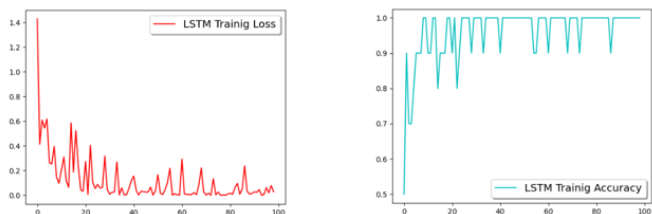


```
Iter 98000, Minibatch Loss= 0.075908, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.055422, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.9233
```

Figure 22. RNN Training result with Adam Optimizer and different hidden units

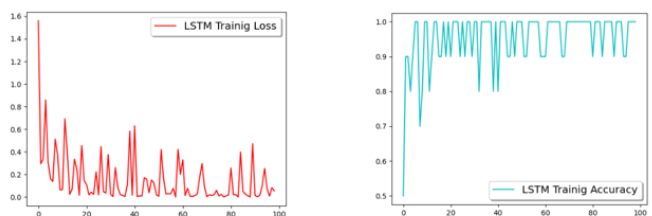
2. LSTM (Long Short-Term Memory)

Training result: LSTM (with 256 hidden units)



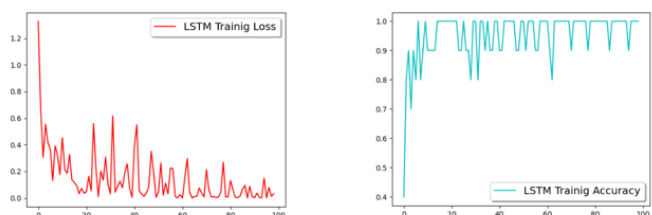
```
Iter 98000, Minibatch Loss= 0.079128, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.028933, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.9792
```

Training result: LSTM (with 128 hidden units)



```
Iter 98000, Minibatch Loss= 0.083752, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.054904, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.975
```

Training result: LSTM (with 64 hidden units)

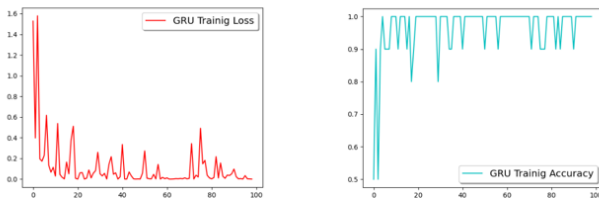


```
Iter 98000, Minibatch Loss= 0.013505, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.031870, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.9744
```

Figure 23. LSTM Training result with Adam Optimizer and different hidden units

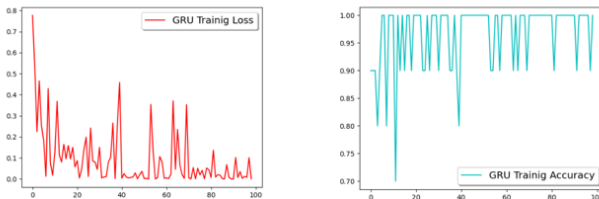
3. GRU (Gated Recurrent Units)

Training result: GRU (with 256 hidden units)



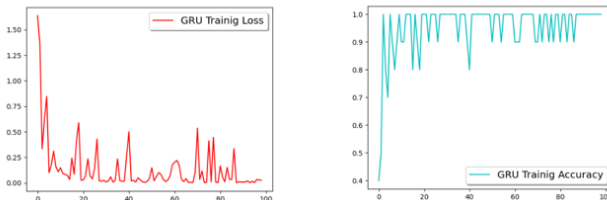
```
Iter 98000, Minibatch Loss= 0.101334, Training Accuracy= 0.90000  
Iter 99000, Minibatch Loss= 0.000421, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.9859
```

Training result: GRU (with 128 hidden units)



```
Iter 98000, Minibatch Loss= 0.001894, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.000359, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.9806
```

Training result: GRU (with 64 hidden units)



```
Iter 98000, Minibatch Loss= 0.026843, Training Accuracy= 1.00000  
Iter 99000, Minibatch Loss= 0.025774, Training Accuracy= 1.00000  
Optimization finished  
Testing Accuracy: 0.9783
```

Figure 24. GRU Training result with Adam Optimizer and different hidden units

My Observation:

In this section, I use different hidden units for RNN, LSTM, and GRU and choose Adam Optimizer for training. Despite that the best accuracy can be found in Figures 19 ~ 20 using the Momentum Optimizer for the RNN, I found that it is not applied to LSTM and GRU after testing, so I use Adam Optimizer in this case uniformly.

From the above training results, we can see that the accuracy of RNN decreases with the hidden units after training with Adam Optimizer, while on the contrary, LSTM and GRU both increase the accuracy with the increase of hidden units.

LSTM have forget gate to decides what is relevant to keep from prior steps, and the input gate decides what information is relevant to add from the current step, the cell state will calculate with those vectors and the previous cell state, also it has the output gate determines what the next hidden state should be. GRU is pretty similar to LSTM, but it's got rid of the cell state and used the hidden state to transfer information. GRU has fewer tensor operations, ideally GRU should be faster than LSTM.

In conclusion, the accuracy of GRU is slightly higher than LSTM as shown in Figure 23,24, but both of them are certainly better than RNN, especially GRU achieves the highest accuracy of 0.9859 with 256 hidden units, I also tried Momentum optimizer for GRU with 256 hidden units, but it only achieved 0.9125 accuracy, which is worse than RNN, so I didn't put the training result in this section.

(c) Compare against the CNN:

My Observation:

Both techniques could work on image recognition or other deep learning tasks, but RNN focus on dependency between lines of images, it's ideal for text translation, natural language processing, sentiment analysis and speech analysis, which means it's suitable for handling temporal or sequential data.

In addition, CNN is a type of feed-forward artificial neural network, it focus on capturing the same patterns on all the different subfields of image, which means CNN is more capable of processing and classifying data in a more hierarchical way, so usually CNN is ideal for images and video processing.