CS229 Final Project - Handwriting Recognition

Bowen Zheng
SID:861130900, CS229-Machine Learning, Dec 9, 2016

I. Problem Formulation

We first mathematically define the handwriting recognition problem. The input vector is x with length 128, corresponding to the pixels of a 16x8 image. Each input vector x belongs to a Letter Class i, which encoded as a number from 0 to 25. We turn the class in to a *hot vector* with length 26, and only one position of the vector would be 1. The corresponding index would be the class number. If it is parametrized by a weight matrix W and bias b, the problem can be formulated as below. Here we follow the problem formulation in Google's tensorflow tutorial on the MNIST digits recognition problem [3], [1].

$$\begin{aligned} P(Y=i) &= softmax_i(Wx+b) \\ &= \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}} \end{aligned} \tag{1}$$

To predict the classes y_{pred}, we should get the index which has the maximal probability, thus,

$$y_{pred} = argmax_i P(Y=i) \tag{2}$$

The loss function is defined as Equation (3), similarly as in [3], [1] for the MNIST digits recognition problem.

$$L = -\sum_i y_i log(softmax(fx_i)) \tag{3}$$

II. Intitial Approach - Two Hidden Layer Neural Network

A. Approach

In my initial approach, I build a neural network with 2-hidden layers. The topology is showing as Figure. 1. We have 128 nodes for the input layer, and two hidden layers each with 30 nodes, and finally the output layer with 26 nodes (as a hot vector).

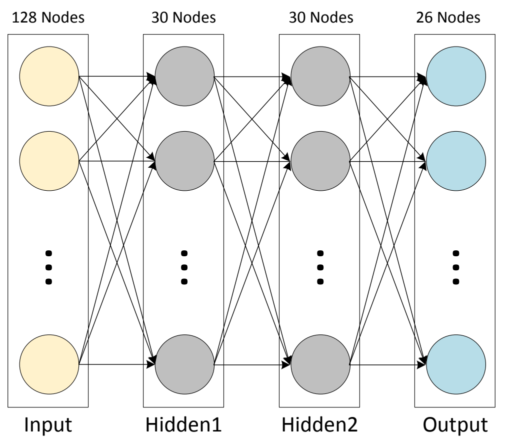

Fig. 1: The neural network topology for the initial approach.

The implementation is described in the rest of the section using Google's tensorflow interfaced with Python.

We first build the neural network topology, and the following code is used to create hidden layer 1. Here $imageLength = 128$, and the first hidden layer has 30 units. So for the weights of the first layer $W_{hidden1}$, its size is 128x30. The length of $b_{hidden1}$ is 30. We apply the none-linear function $tf.nn.relu()$ to the results of $W_{hidden1} \cdot x + b_{hidden1}$.

CODE 1: Hidden Layer 1

```
x = tf.placeholder(tf.float32, shape=[None, imageLength])
y = tf.placeholder(tf.float32, shape=[None, numClasses])

#hidden_layer_1
hidden_units_1 = 30
W_hidden1 = weight_variable([imageLength, hidden_units_1])
b_hidden1 = bias_variable([hidden_units_1])

h_hidden1 = tf.nn.relu(tf.matmul(x, W_hidden1) + b_hidden1
     )
```

Similarly, we can build the second hidden layer with 30 nodes and the output layer.

CODE 2: Hidden Layer 2

```
#hidden_layer_2
hidden_units_2 = 30
W_hidden2 = weight_variable([hidden_units_1,
     hidden_units_2])
b_hidden2 = bias_variable([hidden_units_2])

h_hidden2 = tf.nn.relu(tf.matmul(h_hidden1, W_hidden2) +
     b_hidden2)
```

CODE 3: Output Layer

```
#output_layer
W = weight_variable([hidden_units_2, numClasses])
b = bias_variable([numClasses])

fx = tf.matmul(h_hidden2, W) + b
```

Then we specify the loss function according to Equation (3). The optimizer $tf.train.AdamOptimizer()$ is based on Gradient Descent with an initial learning rate. It can adaptively change the learning rate according to the difference between current and previous loss function value.

CODE 4: Loss Function and Optimization

```
# cost function
fx_0_1 = tf.nn.softmax(fx)
cross_entropy = tf.reduce_mean(-y * tf.log(fx_0_1))

correct_prediction = tf.equal(tf.argmax(fx_0_1,1), tf.
     argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.
     float32))

# optimisation function
```

```
train_step = tf.train.AdamOptimizer(LEARNING_RATE).
     minimize(cross_entropy)
```

Then we start training the neural network. We use batches to train the data set. Here one batch contains 100 data. After we use all the data from the data set, we call it one *epoch*, and we shuffle the data set for the next round.

CODE 5: Training

```
batch_size = 100
data_size = train_images.shape[0]
for epoch in range(epoch_len):
    for batch_i in range(data_size/batch_size):
        batch_images = train_images[batch_i*batch_size : (
             batch_i+1)*batch_size]
        batch_labels = train_labels[batch_i*batch_size : (
             batch_i+1)*batch_size]
        sess.run(train_step, feed_dict = {x: batch_images,
              y: batch_labels})
    #shuffle the data
    perm = np.arange(data_size)
    np.random.shuffle(perm)
    train_images = shuffle(train_images, perm)
    train_labels = shuffle(train_labels, perm)
```

Then we can evaluate our neural network with test data. We define the *accuracy* as the ratio of correctly predicted data on total data. We feed the computation of accuracy with test data to obtain the value.

CODE 6: Evaluating

```
correct_prediction = tf.equal(tf.argmax(fx,1), tf.argmax(y
     ,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.
     float32))

prediction = tf.argmax(fx,1)

predicted_labels = prediction.eval(feed_dict = {x:
     test_images})
accuracy = accuracy.eval(feed_dict={x: test_images,y:
     test_labels})
```

B. Cross Validation

In order to validate our approach, I conducted **10-fold Cross Validation** to evaluate the approach. Here I divide the data set into 10 sets. Each time, I pick one set as the validation data and the other 9 sets as the training set. So there are 10 different situations in total. We compute the mean of the training accuracy and testing accuracy.

C. Results and Analysis

First, we want to find out the relationship between bias and variance. I increase the size of the data set, then evaluate the training accuracy and testing accuracy through 10-fold cross validation. The results are shown in Figure. 2. We can tell two things from the figure.

- **The bias is high** as our target accuracy is at least over 90%, and there is still a gap.
- **The variance is not large**, as the gap between training error and testing error is small. So, bias is the main problem.

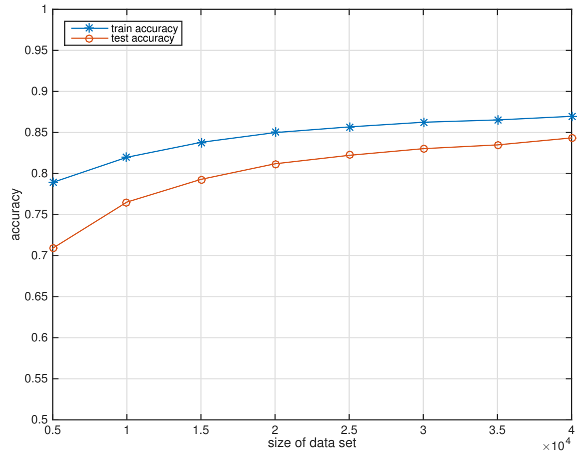

Fig. 2: The training accuracy and testing accuracy with increase of the size of data set

Then we also want to know whether the time for the optimization is long enough. So we conducted another experiment to increase the training epoch and see the change of the accuracy, as shown in Figure. 3. From this figure we can tell that current training iterations more or less make the optimization converge. **Therefore, This is NOT the problem of optimization or short optimization time.**

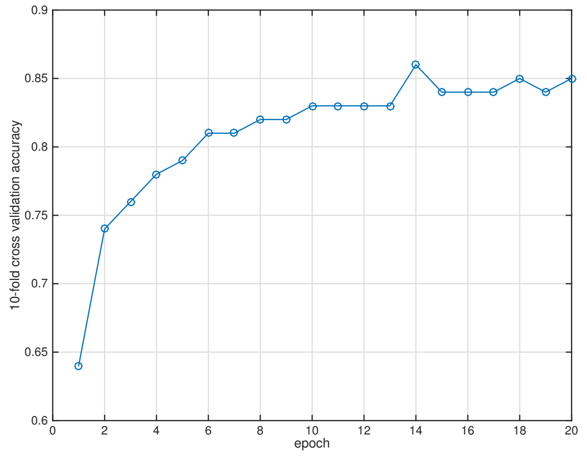

Fig. 3: The validation accuracy with increase of training epoch

D. How to improve?

- **How to reduce bias?** One possible solution is to increase or change the features. As the features currently are just the pixels, if there are some methods to turn those pixels to something more meaningful, it might reduce the bias. After searching on the web, doing convolution is probably a good way to naturally abstract some features from the image, and the *Convolutional Neural Network* is discussed to be useful for digit recognition [2].

- **How to reduce variance?** One possible solution is to drop out some of the nodes in the neural network. I will try this in the next section and this is not our main concern.

III. Improvement - Convolutional Neural Network

A. Implementation

As we discussed above, we will use convolutional neural network to naturally abstract more features to reduce bias and conduct dropout to reduce variance. The topology is to change the two hidden layers to two convolution layers, and conduct dropout, as shown in Figure. 4. The implementation change is

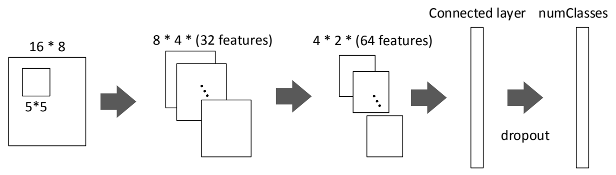

Fig. 4: The topology for convolutional neural network

the convolution layers and dropout. The others are the same as the previous code.

CODE 7: Implementation of convolution layers

```
#conv_layer_1
feature1 = 32
W_hidden1 = weight_variable([5, 5, 1, feature1])
b_hidden1 = bias_variable([feature1])

x_image = tf.reshape(x, [-1, imageWidth, imageHeight, 1])

h_hidden1 = tf.nn.relu(conv2d(x_image, W_hidden1) +
     b_hidden1)
h_pool1 = max_pool_2x2(h_hidden1)

#conv_layer_2
feature2 = 64
W_hidden2 = weight_variable([5, 5, feature1, feature2])
b_hidden2 = bias_variable([feature2])

h_hidden2 = tf.nn.relu(conv2d(h_pool1, W_hidden2) +
     b_hidden2)
h_pool2 = max_pool_2x2(h_hidden2)

#densely connected layer
h_pool2_flat = tf.reshape(h_pool2, [-1, 4 * 2 * feature2])
W_fc1 = weight_variable([4 * 2 * feature2, imageLength])
b_fc1 = bias_variable([imageLength])

h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

#dropout
keep_prob = tf.placeholder('float')
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

#out_layer
W_fc2 = weight_variable([imageLength, numClasses])
b_fc2 = bias_variable([numClasses])
```

B. Analysis

I conduct the same analysis as for the previous approach. First the training accuracy and validation accuracy with regard to the size of the data set. The results are shown in Figure. 5.

We can tell from the figure that the bias is reduced (the validation accuracy is increased) and the variance does not decrease. That means, the dropout in this case does not help much with the variance.

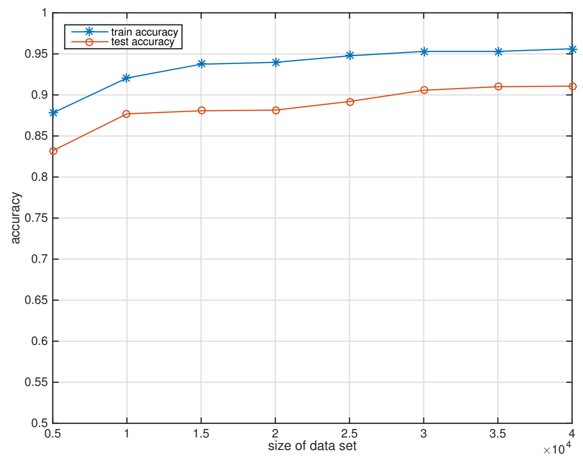

Fig. 5: The training accuracy and testing accuracy with increase of the size of data set (Convolutional Neural Network)

We also want to make sure the optimization is long enough to converge. We show the validation accuracy with the increase of epoch, compared with the previous neural network and the current convolutional neural network. We can find out that the accuracy is increased from 0.84 to 0.92, and the optimization is more or less converged.

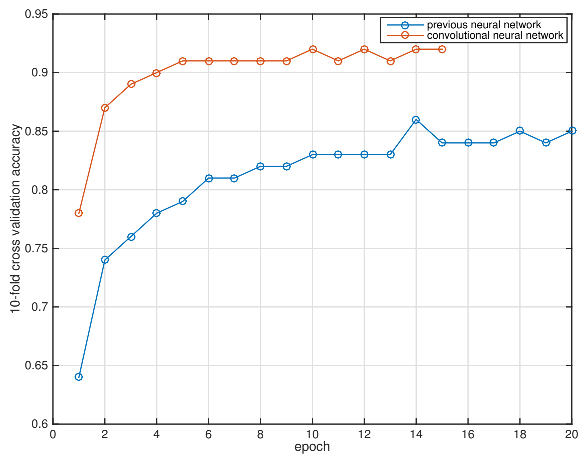

Fig. 6: The validation accuracy with increase of training epoch

References

[1] Classifying MNIST digits using Logistic Regression. http://deeplearning.net/tutorial/logreg.html.

[2] Deep Learning 05: Talk about Convolutional Neural NetworksCNN). https://ireneli.eu/2016/02/03/deep-learning-05-talk-about-convolutional-neural-network(/.

[3] MNIST For ML Beginners and Deep MNIST Experts. https://www.tensorflow.org/versions/r0.12/tutorials/mnist/beginners/ index.html#mnist-for-ml-beginners.