ECE421 Assignment1 Report

Shutong Zhang(1005722498)

February 7, 2022

1 Logistic Regression With Numpy

The cross-entropy loss function can be written in the form:

$$L = \frac{1}{N}\sum_{n=1}^{N}[-y^{(n)}log\hat{y}(x^{(n)}) - (1-y)log(1-\hat{y}(x^{(n)}))] + \frac{\lambda}{2}\|w\|_2^2$$

Where $\hat{y}(x) = \sigma(w^T x + b)$.

1.1 Loss Function and Gradient

The implementation of function *loss* is intuitive by following the equation listing above.

```
def loss(w, b, x, y, reg):
    y_hat = 1.0 / (1.0 + np.exp(-np.matmul(x, w) - b))
    error = -1 * y * np.log(y_hat) - (1 - y) * np.log(1 - y_hat)
    CEloss = np.sum(error) / np.shape(x)[0] + reg * np.sum(w*w) / 2
    return CEloss
```

Then, in order to find the gradient of the weights and the bias, I employed the chain rule to take the derivative. For a single set of data (x, y), let $z = w^T x + b$, $s = \sigma(z)$, then we have:

$$L_{CEsingle} = -y^{(n)}logs - (1-y)log(1-s)$$

Apply chain rule we have:

$$\frac{\partial L_{CEsingle}}{\partial w} = \frac{\partial L}{\partial s}\frac{\partial s}{\partial z}\frac{\partial z}{\partial w} = (-\frac{y}{s} + \frac{1-y}{1-s})s(1-s)x = (s-y)x$$

$$and$$

$$\frac{\partial L_{CEsingle}}{\partial b} = \frac{\partial L}{\partial s}\frac{\partial s}{\partial z}\frac{\partial z}{\partial b} = (-\frac{y}{s} + \frac{1-y}{1-s})s(1-s) = (s-y)$$

For the regularization loss, we have:

$$\frac{\partial L_w}{\partial w} = \lambda w \quad and \quad \frac{\partial L_w}{\partial b} = 0$$

Finally, add all those terms together we have:

$$\frac{\partial L}{\partial w} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_{CEsingle}}{\partial w} + \frac{\partial L_w}{\partial w} \quad and \quad \frac{\partial L}{\partial w} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial L_{CEsingle}}{\partial b} + \frac{\partial L_w}{\partial b}$$

Following the equation above, the implementation of the function *grad_loss* is shown below:

```
def grad_loss(w, b, x, y, reg):        #Note that x is already reshaped before it passed in
    s = 1 / (1 + np.exp(-np.matmul(x, w) - b))    #Or we add x = x.reshape(x.shape[0], -1)
    grad_w = (np.matmul(np.transpose(x), s - y)) / np.shape(x)[0] + reg * w
    grad_b = np.sum(s - y) / np.shape(x)[0]
    return grad_w, grad_b
```

1.2 Gradient Descent Implementation

Gradient decent algorithm is a common algorithm we used to optimize our model by finding the weight and bias with the smallest loss. Gradient is the steepest direction of the current location, we calculate the gradient and then use the following the rules to update the weight and bias.

$$w_{new} = w_{old} - \alpha \frac{\partial L}{\partial w} \quad and \quad b_{new} = b_{old} - \alpha \frac{\partial L}{\partial b}$$

We end the process after *epochs* iterations or the difference between w_{new} and w_{old} is smaller than the *error_tolerance*. The implementation of the gradient descent algorithm is shown below:

```
def grad_descent(w, b, x, y, x1, y1, alpha, epochs, reg, error_tol):
    #x1 and y1 are for validation data
    #store loss and accuracy
    train_loss, train_acc, validation_loss, validation_acc = [], [], [], []
    for epochs_num in range(epochs):
        #calculate loss and accuracy(see implementation of get_accuracy in code file)
        train = loss(w, b, x, y, reg)
        train_a = get_accuracy(w, b, x, y)
        val = loss(w, b, x1, y1, reg)
        val_a = get_accuracy(w, b, x1, y1)
        train_loss.append(train)
        validation_loss.append(val)
        train_acc.append(train_a)
        validation_acc.append(val_a)
        #calculate gradient for w and b
        grad_w, grad_b = grad_loss(w, b, x, y, reg)
        new_w = w - alpha * grad_w
        new_b = b - alpha * grad_b
        #check difference between w and w_new
        diff = np.linalg.norm(w - new_w)
        if diff < error_tol:
            return w, b, train_loss, validation_loss, train_acc, validation_acc
        #update w and b
        w = new_w
        b = new_b
    return w, b, train_loss, validation_loss, train_acc, validation_acc
```

1.3 Tuning the Learning Rate

In this section, I trained the model with a $error\_tol = 1 \times 10^{-7}$ and $\alpha = 0$ for 5000 epochs with 0.005, 0.001 and 0.0001 learning rate. The table below shows the training/validation loss/accuracy with different learning rate.

learning rate	training loss	validation loss	training accuracy	validation accuracy
0.005	0.106	0.125	0.968	0.950
0.001	0.199	0.399	0.948	0.900
0.0001	0.505	0.491	0.847	0.850

Table 1: training/validation loss/accuracy with different learning rate.

The figures below shows the changing of loss and accuracy with respect to different learning rate.

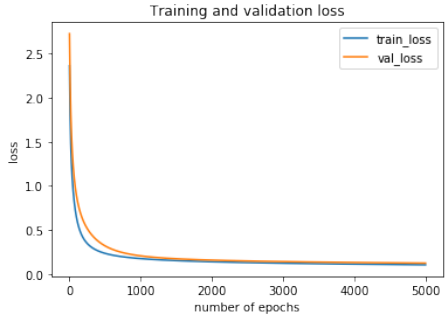

Figure 1: Loss when learning rate is 0.005

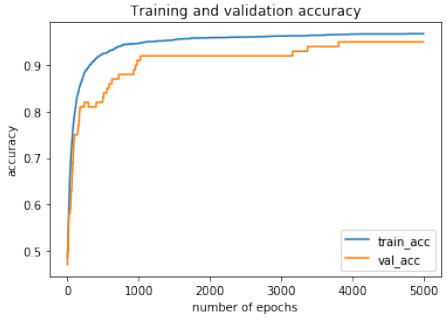

Figure 2: Accuracy when learning rate is 0.005

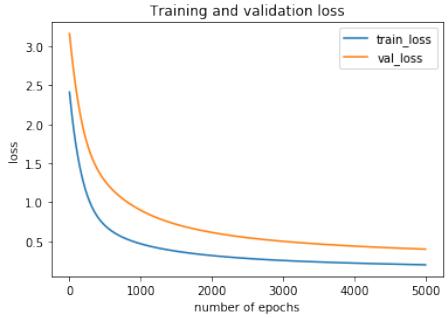

Figure 3: Loss when learning rate is 0.001

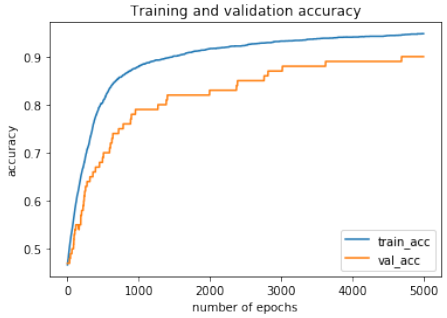

Figure 4: Accuracy when learning rate is 0.001

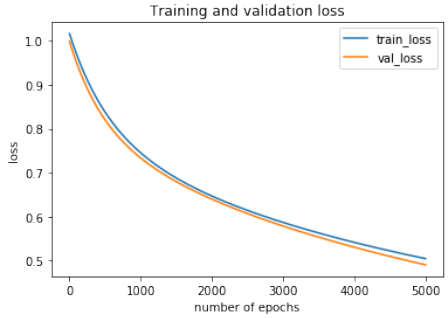

Figure 5: Loss when learning rate is 0.0001

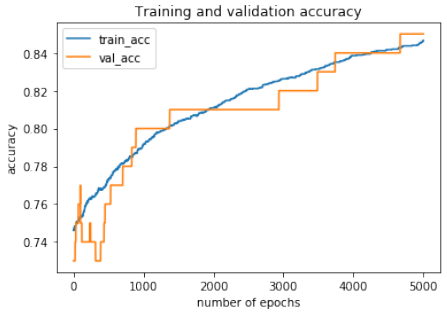

Figure 6: Accuracy when learning rate is 0.0001

According to the result during the training process, I choose 0.005 as the best learning rate. This is because when learning rate is 0.005, we have the lowest loss and the highest accuracy(both for training and validation data). And it seems that when learning rate is 0.001 or 0.0001, the model is still in the under-fit stage. Then I use the weight and bias generated when learning rate is 0.005 on the test data(I only test once because ideally we should only use the test data at very end of the training process).Finally, I got a training accuracy of 0.96. Then I test my choice with weight and bias of all learning rate, the result proves that I am correct.

learning rate	0.005	0.001	0.0001
test accuracy	0.959	0.938	0.869

Table 2: Test accuracy with different learning rate.

1.4 Generalization

In this section, I trained the model with a $error\_tol = 1 \times 10^{-7}$ and $learningrate = 0.005$ for 5000 epochs with 0.001, 0.1 and 0.5 regularization parameter λ. The table below shows the training/validation loss/accuracy with different λ.

regularization parameter	training loss	validation loss	training accuracy	validation accuracy
0.001	0.189	0.219	0.966	0.960
0.1	0.170	0.180	0.980	0.970
0.5	0.198	0.215	0.975	0.990

Table 3: training/validation loss/accuracy with different regularization parameter.

The figures below shows the changing of loss and accuracy with respect to different learning rate.

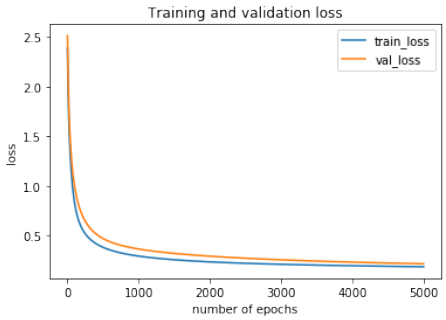

Figure 7: Loss when λ is 0.001

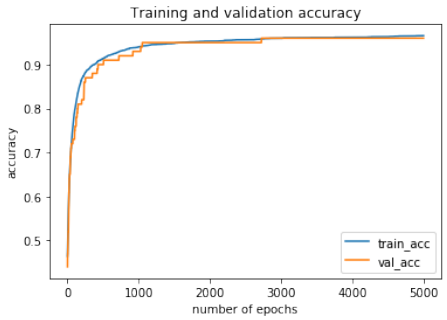

Figure 8: Accuracy when λ is 0.001

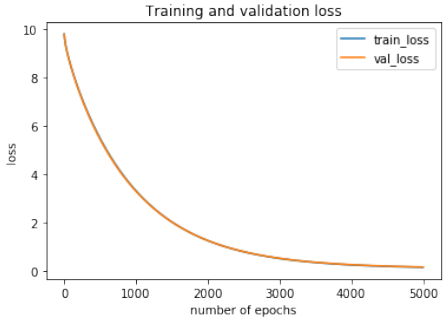

Figure 9: Loss when λ is 0.1

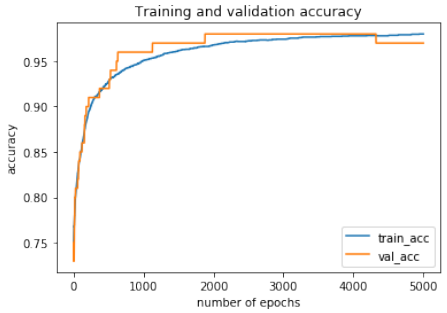

Figure 10: Accuracy when λ is 0.1

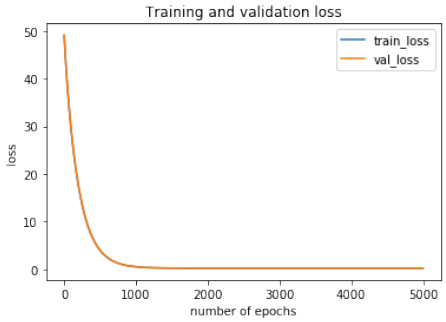

Figure 11: Loss when λ is 0.5

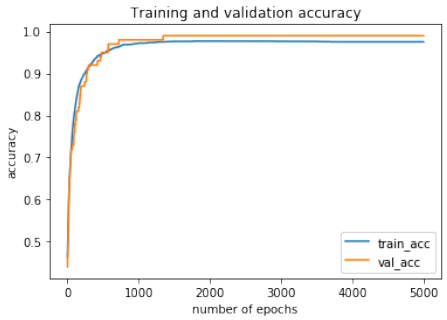

Figure 12: Accuracy when λ is 0.5

According to the result during the training process, I choose 0.5 as the best regularization parameter. This is because when regularization is 0.5, we have the highest validation accuracy(although it has a higher loss, may need us to train more epochs). Then I use the weight and bias generated when regularization parameter is 0.5 on the test data(I only test once because ideally we should only use the test data at very end of the training process).Finally, I got a training accuracy of 0.98. Then I test my choice with weight and bias of all learning rate, the result proves that I am correct(Actually three results are pretty close, this may because we only have a small number of test data).

regularization parameter	0.001	0.1	0.5
test accuracy	0.972	0.979	0.979

Table 4: Test accuracy with different learning rate.

2 Logistic Regression in TensorFlow

In this section, I implemented the SGD algorithm with Adam optimizer using Tersorflow.

2.1 Building the Computational Graph

The implementation of the *buildGraph*() function is shown below. The function takes learning rate as input(for easier changing), and returns the Tensorflow object of weight, bias, predicted labels, real labels, the loss, and the optimizer.

```
def buildGraph(learning_rate):
    #w and b are set as variables
    w = tf.Variable(tf.truncated_normal(shape=(784, 1), mean=0.0, stddev=0.5,
        dtype=tf.float32))
    b = tf.Variable(0, dtype=tf.float32)
    #train data/target and lambda are set as placeholders for input
    train_data = tf.placeholder(tf.float32)
    train_label = tf.placeholder(tf.float32)
    lam = tf.placeholder(tf.float32)
    #calculate loss
    logits = tf.matmul(train_data, w) + b
    loss = tf.losses.sigmoid_cross_entropy(train_label, logits)
    regularization = tf.nn.l2_loss(w)
    CEloss = loss + lam * regularization
    #Using Adam optimizer
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    train = optimizer.minimize(CEloss)
    #return weight, bias, predicted labels, real labels, the loss, and the optimizer
    return w, b, train_data, train_label, logits, CEloss, train, lam
```

2.2 Implementing Stochastic Gradient Descent

In this section, I implemented the Stochastic Gradient Descent algorithm. The function takes eight input: batch size, epoch numbers, learning rate, training data, training target, validation data, validation target and regularization parameter. The function first call *buildGraph*() to load all the Tensorflow objects, then start the session. The outer loop traverse the number of epochs and shuffle training data/target while the inner loop traverse the number of iterations each epochs. Inside the inner loop, it first use the training data and target to compute the training loss and update the weight and bias. Then it use the validation data and target to compute the validation loss without updating the weight and bias. After each epoch, it collect the training/validation loss/accuracy and finally plot them after all epochs.

```
def SGD(batch_size, epoch_num, learning_rate, traindata, traintarget, validata, validtarget,
    reg):
    w, b, train_data, train_label, logits, CEloss, train, lam = buildGraph(learning_rate)
    #loss/accuracy collector
    train_loss, train_acc, val_loss, val_acc = [], [], [], []
    #initialize and run new session
    init_op = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init_op)
        for epochs in range(epoch_num):
            #shuffle training data/target according to indices
            indices = np.arange(traindata.shape[0])
            np.random.shuffle(indices)
            traindata = traindata[indices]
            traintarget = traintarget[indices]
            #calculate iteration numbers: iterations = data number/batch size
            iteration_number = int(traindata.shape[0]/batch_size)
            for i in range(iteration_number):
                train_d = trainData[i*batch_size:(i+1)*batch_size]
                train_t = trainTarget[i*batch_size:(i+1)*batch_size]
                #calculate training loss and update w, b
                train_w, train_b, t_loss, t_label, train_optimizer = sess.run([w, b, CEloss,
                    logits, train],{
                        train_data: train_d,
                        train_label: train_t,
                        lam: reg
                    })
                #calculate validation loss without update w, b
                v_loss = sess.run(CEloss, {
                        train_data: validata,
                        train_label: validtarget,
                        lam: reg
                    })
            #store train/validation loss/accuracy
            train_loss.append(t_loss)
            train_acc.append(get_accuracy(train_w, train_b, traindata, traintarget))
            val_loss.append(v_loss)
            val_acc.append(get_accuracy(train_w, train_b, validata, validtarget))
    #plot and print result
    number = range(len(train_loss))
    plt.plot(number, train_loss)
    plt.plot(number, val_loss)
    plt.legend(['train_loss', 'val_loss'])
    plt.xlabel("epoch numbers")
    plt.ylabel("loss")
    plt.title("Training and validation loss")
    plt.show()
    plt.plot(number, train_acc)
    plt.plot(number, val_acc)
    plt.legend(['train_acc', 'val_acc'])
    plt.xlabel("epoch numbers")
    plt.ylabel("accuracy")
    plt.title("Training and validation accuracy")
    plt.show()
    print("Final training loss:", train_loss[-1], ", validation loss:", val_loss[-1])
    print("Final training accuracy:", train_acc[-1], ", validation accuracy:", val_acc[-1])
    return train_w, train_b
```

I set the regularization parameter $\lambda = 0$ and learning rate $\alpha = 0.001$, with a minibatch size of 500 over 700 epochs. After training, I got the following result: $trainingloss = 0.041$, $validationloss = 0.079$, $trainingaccuracy = 0.988$ and $validationaccuracy = 0.970$. Thee final testing accuracy is 0.979. The plot of training/validation loss/accuracy are shown below.

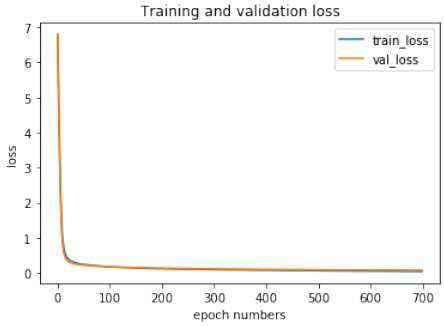

Figure 13: Loss(500, 700)

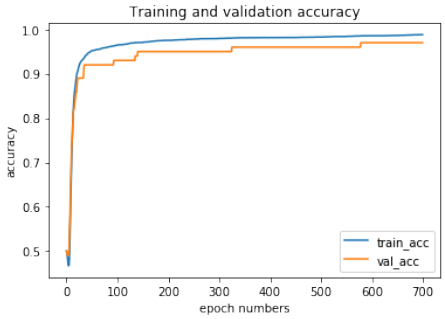

Figure 14: Accuracy(500, 700)

2.3 Batch Size Investigation

In this section, we studied the batch size behavior on the Adam optimizer. We set regularization parameter $\lambda = 0$ and learning rate $\alpha = 0.001$, then train the model with three different batch size 100, 700 and 1750. The training results(train/validation loss/accuracy) are shown in the table below.

batch size	training loss	validation loss	training accuracy	validation accuracy
100	0.003	0.112	0.999	0.970
700	0.041	0.038	0.987	0.990
1750	0.076	0.090	0.977	0.960

Table 5: training/validation loss/accuracy with different batch size.

The figures below shows the changing of loss and accuracy with respect to different learning rate.

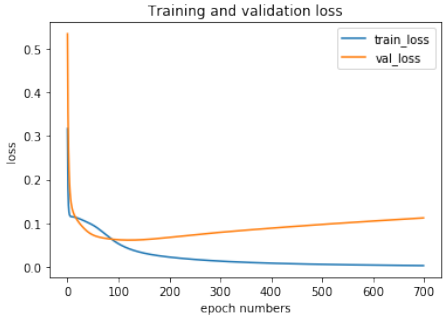

Figure 15: Loss when batch size is 100

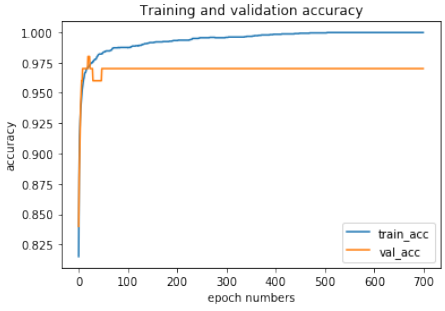

Figure 16: Accuracy when batch size is 100

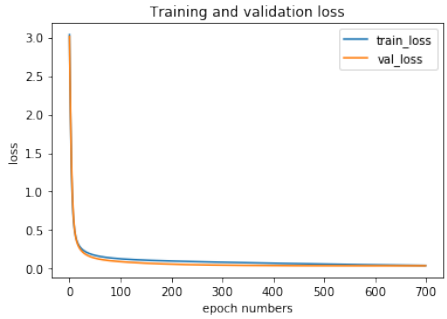

Figure 17: Loss when batch size is 700

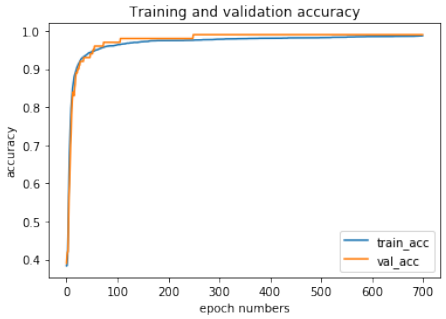

Figure 18: Accuracy when batch size is 700

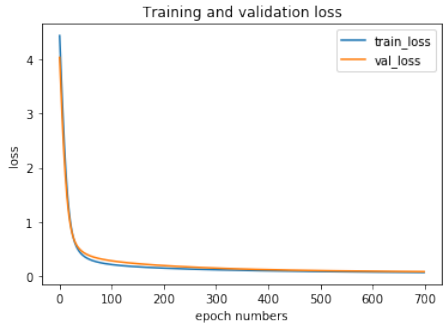

Figure 19: Loss when batch size is 1750

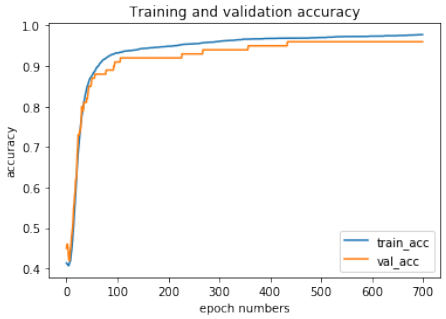

Figure 20: Accuracy when batch size is 1750

We can see from the figures that loss and validation plot are different with different batch size. We can also see from the table above that when batch size is 700, we have the lowest validation loss and the highest validation accuracy.

When batch size is equal to 100, we can see that with epoch numbers increase, the training loss keep decrease but the validation loss first decrease and then increase. This may happens because a small batch size will cause the inner loop to be called too many times, thus overfit to the training data. The training accuracy is highest among three groups, however, the validation accuracy is no the highest. This also happens due to the overfit.

When batch size is 700, we get the highest validation loss and accuracy. This is because with larger batch size, the inner loop is called fewer but enough time, which means there is no overfit and underfit happens. This is why the validation loss is lowest and accuracy is highest, even the training loss is not the lowest and the accuracy is not the highest.

When batch is 1750, we get the highest training loss and the lowest training accuracy. This is because the inner loop is called too few times. The validation loss becomes higher and the validation accuracy becomes lower, this is also because the model is not trained enough, which is in the underfit stage. Compare the slop of the curves in different figures, we can see that the slope of the curve decrease when batch size increase, this is because when batch size becomes larger, the inner loop is called fewer times in each epochs thus result in a slower update for weight and bias. The testing result also proved my observation.

batch size	100	700	1750
test accuracy	0.97	0.98	0.96

Table 6: Test accuracy with different batch size

2.4 Hyperparameter Investigation

In this part, I changed the prototype of buildGraph() and SGD() to allow $\beta 1$, $\beta 2$ and ϵ as input. The prototype of these function are shown below.

```
def buildGraph(batch_size, learning_rate, beta1, beta2, epsilon):
#same implementation as above
def SGD(batch_size, epoch_num, learning_rate, traindata, traintarget, validata, validtarget,
    reg, beta1, beta2, epsilon):
#same implementation as above
```

The default value of Adam optimizer is $\beta 1 = 0.9$, $\beta 2 = 0.999$ and $\epsilon = 1e-8$ [1]. I train the model with different $\beta 1$, $\beta 2$ and ϵ(change their value one at a time, other two remain default). The results are shown in the table below.

parameters	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 1e-4$	$\epsilon = 1e-9$
training loss	0.023	0.027	0.012	0.032	0.022	0.032
training accuracy	0.992	0.994	0.999	0.993	0.995	0.991
validation loss	0.053	0.104	0.094	0.092	0.074	0.073
validation accuracy	0.980	0.970	0.970	0.970	0.980	0.970
test accuracy	0.972	0.965	0.972	0.979	0.979	0.972

Table 7: training/validation loss/accuracy with different hyperparameters.

2.4.1 The effect of β_1

In Adam, the paremeter β_1 is used to calculate the first moment of the gradient. The calculation is shown as below:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \text{ [1]}$$

Where m_t is the moment of the current step, m_{t-1} is the moment from the previous step and g_t is the gradient of the current step. We can easily know from the equation that when $\beta 1$ becomes larger, the current moment will be affected more by the historical moment, thus may affect the result. As we can see from the table above, with $\beta 1$ increase, the validation loss increase, the validation accuracy decrease and the test accuracy also decrease. So I would pick $\beta_1 = 0.95$.

2.4.2 The effect of β_2

In Adam, the parameter β_2 is used to calculate the second moment of the gradient. The calculation is shown as below:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \text{ [1]}$$

Where v_t is the second moment of the current step, v_{t-1} is the second moment from the previous step and g_t is the gradient of the current step. It's pretty similar as $\beta 1$, the increase of $\beta 2$ will make v_2 depend more on the previous second moment. However, as shown in the table above, the change of $\beta 2$ dose not affect the result much. The validation loss/accuracy and testing accuracy are all almost the same. So I would pick β_2 with a higher validation accuracy which is 0.9999.

2.4.3 The effect of ϵ

The ϵ in the Adam optimizer is a number to avoid the equation divided by 0. The calculation is shown above:

$$w_t = w_{t-1} - \alpha \frac{m_t}{\sqrt{v_t} + \epsilon} \text{ [1]}$$

The ϵ is just for correction(avoid 0) and not for any other calculation, so the value of ϵ should be small. From the table above we can see that the training result is not affected by the value of ϵ. However, we should choose small value for ϵ(default 1e-8 or 1e-9) to avoid large ϵ affect small v_t. So I would pick $\epsilon = 1e-9$.

Using the parameter we choose above, which is $\beta_1 = 0.95$, $\beta_2 = 0.9999$ and $\epsilon = 1e - 9$. We run the optimizer again for 700 epochs and 500 batch size, the result are shown as follows.

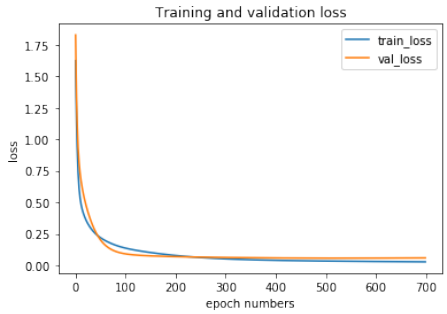

Figure 21: Loss with best parameter

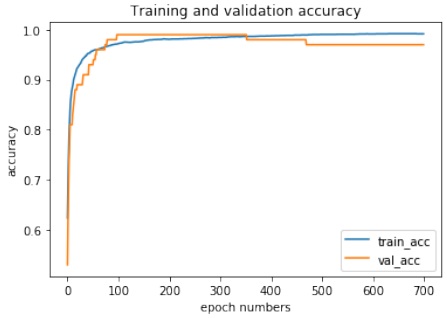

Figure 22: Accuracy with best parameter

validation loss	validation accuracy	test accuracy
0.060	0.970	0.986

Table 8: Result using best parameter.

2.5 Comparison against Batch GD

It's obvious that the SGD algorithm with Adam optimizer is much better than the Batch GD algorithm. If we compare the same learning rate group, we can see that the accuracy of Adam optimizer with only 700 epochs is much better than the Batch GD algorithm with 5000 epochs. The figure below shows the comparison of their training/validation loss/accuracy.

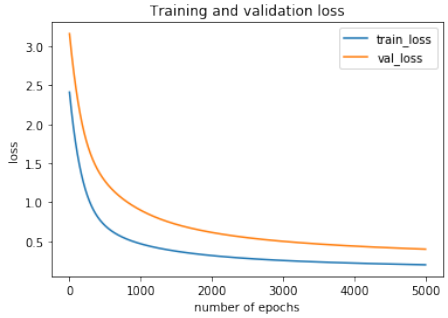

Figure 23: Loss of Batch GD for 5000 epochs

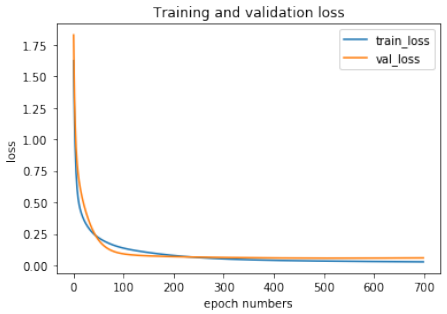

Figure 24: Loss of SGD with 700 epochs

From the loss curve we can see that the loss of Batch GD Algorithm decreased much slower than the SGD algorithm. And the final train/validation loss after 5000 epochs is also much higher than the SGD algorithm after 700 epochs. Even when we increased the learning rate to 0.005 and added regularization parameter of 0.5, the Batched GD algorithm still need 5000 epochs to have the similar final accuracy with the SGD algorithm.

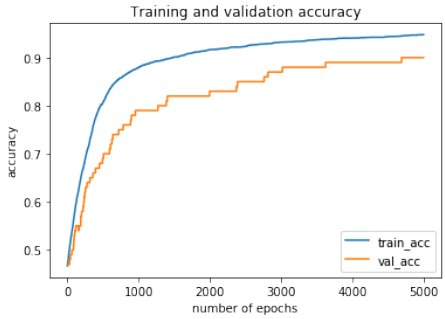

Figure 25: accuracy of Batch GD for 5000 epochs

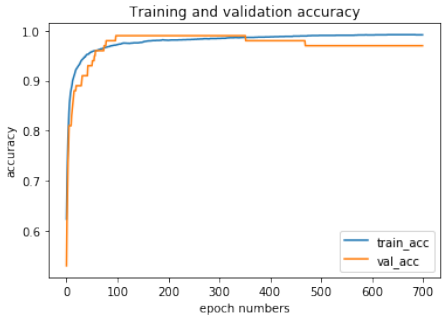

Figure 26: accuracy of SGD with 700 epochs

Then we observe the training/validation accuracy curve. we can also see that, similar to the loss curve, the accuracy increase of the Batch GD algorithm is much slower than the SGD algorithm. We can clearly see that the training process of the SGD algorithm is much faster than the Batched GD algorithm. The SGD with Adam optimizer also have a higher test accuracy than the batched GD algorithm.

The difference of the Batch GD algorithm between the SGD algorithm may happens for the following reasons:

1. The SGD algorithm break the whole training set into several smaller batch, so it takes shorter time to update the weight/bias than Batched GD algorithm. This is also the reason why for the SGD algorithm, the starting loss is lower and the starting accuracy is higher than the Batched GD algorithm(actually after one epoch).

2. The SGD algorithm make use of the Adam optimizer, the optimizer make use of the first and secondary moment of the gradient and also the past value, this will lead a faster convergence and speed up the training process.

3. The SGD algorithm with Adam optimizer can help to escape some local minimum/flat region using the momentum, and it also decrease the influence of noise [2].

Another benefit of using SGD algorithm is that when it break the whole data set in to several batches, it also decreased the memory required when training.

3 Reference

[1] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2014.

[2] R. Kleinberg, Y. Li, and Y. Yuan, "An Alternative View: When Does SGD Escape Local Minima?," 2018.