ECE421 Introduction of Machine Learning

Assignment 1: Logistic Regression

Zetong Zhao

1 Logistic Regression with Numpy

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

$z = \mathbf{w}^T\mathbf{x} + b$ called the logit, and $\sigma(z) = 1/(1 + \exp(-z))$ is sigmoid function.

The total loss function is:

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

where λ is the regularization parameter.

1.1 Loss Function and Gradient

According to the formula above, Python implementation of the total loss function is shown below. Grad_loss is calculated by the partial derivative of w and b.

$$\frac{d\mathcal{L}_{CE}}{d\hat{y}(z)} = \frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\frac{d\hat{y}}{dz} = \hat{y}^2(1-\frac{1}{\hat{y}}) = \hat{y}(1-\hat{y})$$

$$\frac{dz}{dw} = \boldsymbol{x}, \frac{dz}{db} = 1$$

$$\frac{\partial \mathcal{L}_w}{\partial w} = \sum_{n=1}^{N}\lambda w_n$$

To get the partial derivative of w and b, combining the derivative of cross-entropy with respect to $\hat{y}$, the derivative of $\hat{y}$ with respect to z, and the derivative of z with respect to w and b.

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{N}[\mathbf{x}^T(\hat{y}-y)] + \sum_{n=1}^{N}\lambda w_n, \quad \frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N}\frac{1}{N}(\hat{y}-y)$$

Then according to the formula, we can implement the loss and grad_loss using numpy as following.

```
def loss(W, b, x, y, reg):
        z = np.matmul(x,W)+b
        y_ = 1/(1+np.exp(-z))
        loss_CE = np.sum(-(y*np.log(y_)) - (1-y)*np.log(1-y_))
        loss_CE = loss_CE/(np.shape(y)[0])
        loss_w = reg/2*np.sum(W*W)
        loss = loss_CE+loss_w

        return loss
```

Figure1 total loss function by numpy

```
def grad_loss(W, b, x, y, reg):
    dLw = reg*W
    z = np.matmul(x,W)+b
    y_ = 1/(1+np.exp(-z))
    dLCE = np.matmul(np.transpose(x),y_-y)
    grad_W = dLCE/(np.shape(y)[0])+dLw
    grad_b = np.sum(y_ - y)/(np.shape(y)[0])
    return grad_W,grad_b
```

Figure2 gradient of total loss function by numpy

1.2 Gradient Descent Implementation

Gradient Descent is implemented by iterating a number of epochs. Gradient of loss indicates the direction of improvement, and learning rate indicates how much will the step approaching a better value. During each iteration, weight (w) and bias (b) are updated according to learning rate (alpha) times gradient losses. Iteration can be stopped early if the error is within tolerance.

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
    for i in range(epochs):
        grad_lossW, grad_lossb = grad_loss(W,b,x,y,reg)
        W_ = W-alpha*grad_lossW
        b_ = b-alpha*grad_lossb
        diff = np.linalg.norm(W_-W)
        if (diff < error_tol):
            return W_,b_
        else:
            W = W_
            b = b_
    return W_,b_
```

Figure3 gradient descent of total loss function by numpy

1.3 Tuning the Learning Rate

As shown on the graph, with larger learning rate (alpha), the train loss and valid loss reach 0.5 within smaller number of epochs. With larger learning rate (alpha), the train accuracy and valid accuracy reach 0.8 within smaller number of epochs. Smaller learning rate means a slower rate to approach the best solution. It will take more epochs to reach the same accuracy with smaller learning rate. Therefore, the alpha=0.005 is best model in the three choices.

	Alpha = 0.005	Alpha = 0.001	Alpha = 0.0001
Best train accuracy	0.962	0.948	0.85
Best valid accuracy	0.96	0.95	0.8
Lowest train loss	0.127	0.172	0.48
Lowest valid loss	0.133	0.108	0.606

Table1 The best train/valid accuracy and lowest train/valid loss for different alpha

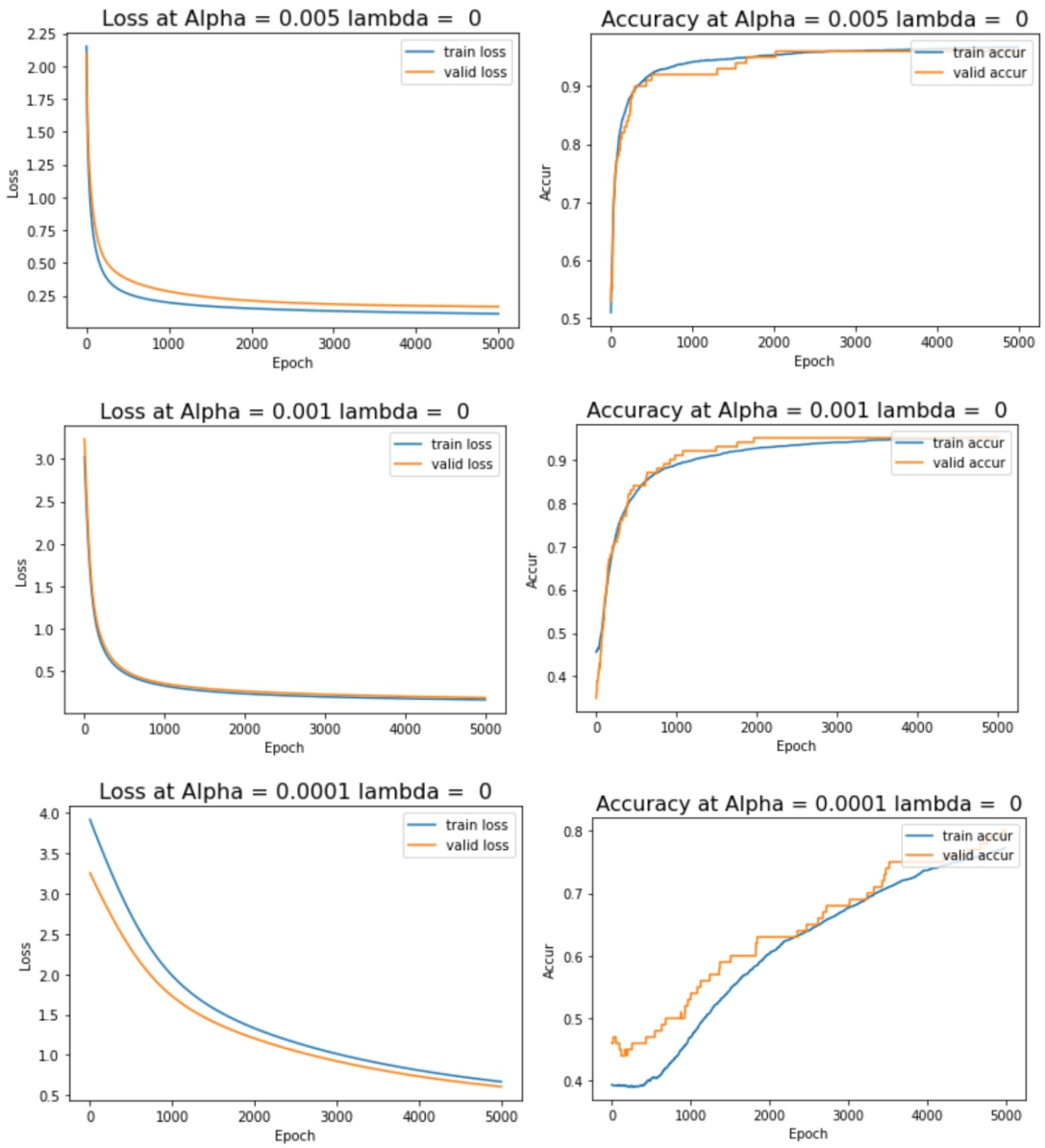


Figure4 training/validation loss/accuracy for different learning rate

1.4 Generation

Lambda is added to the total loss function in order to avoid overfitting due to noisy data or not enough data. If lambda is too small, it cannot reduce overfitting problem much. If lambda is too large, it will produce more noise to the result.

With the same learning rate, smaller lambda leads to smaller loss from 0-1000 epochs. Validation loss is getting closer to the train loss with larger lambda. From the graphs, it trivial that when lambda=0.5 the validation accuracy is stable with about 1000 epochs. When lambda equals 0.1, train accuracy is larger. However, validation loss and accuracy is the most important to the performance of the model. Lambda=0.1 with 5000 epochs at learning rate=0.005 will result in a better model, since it has the smallest validation loss.

	Lambda = 0.001	Lambda = 0.1	Lambda = 0.5
Best train accuracy	0.968	0.974	0.968
Best valid accuracy	0.96	0.97	0.97
Lowest train loss	0.19436	0.177	0.1986
Lowest valid loss	0.26	0.1688	0.198

Table2 The best train/valid accuracy and lowest train/valid loss for different lambda

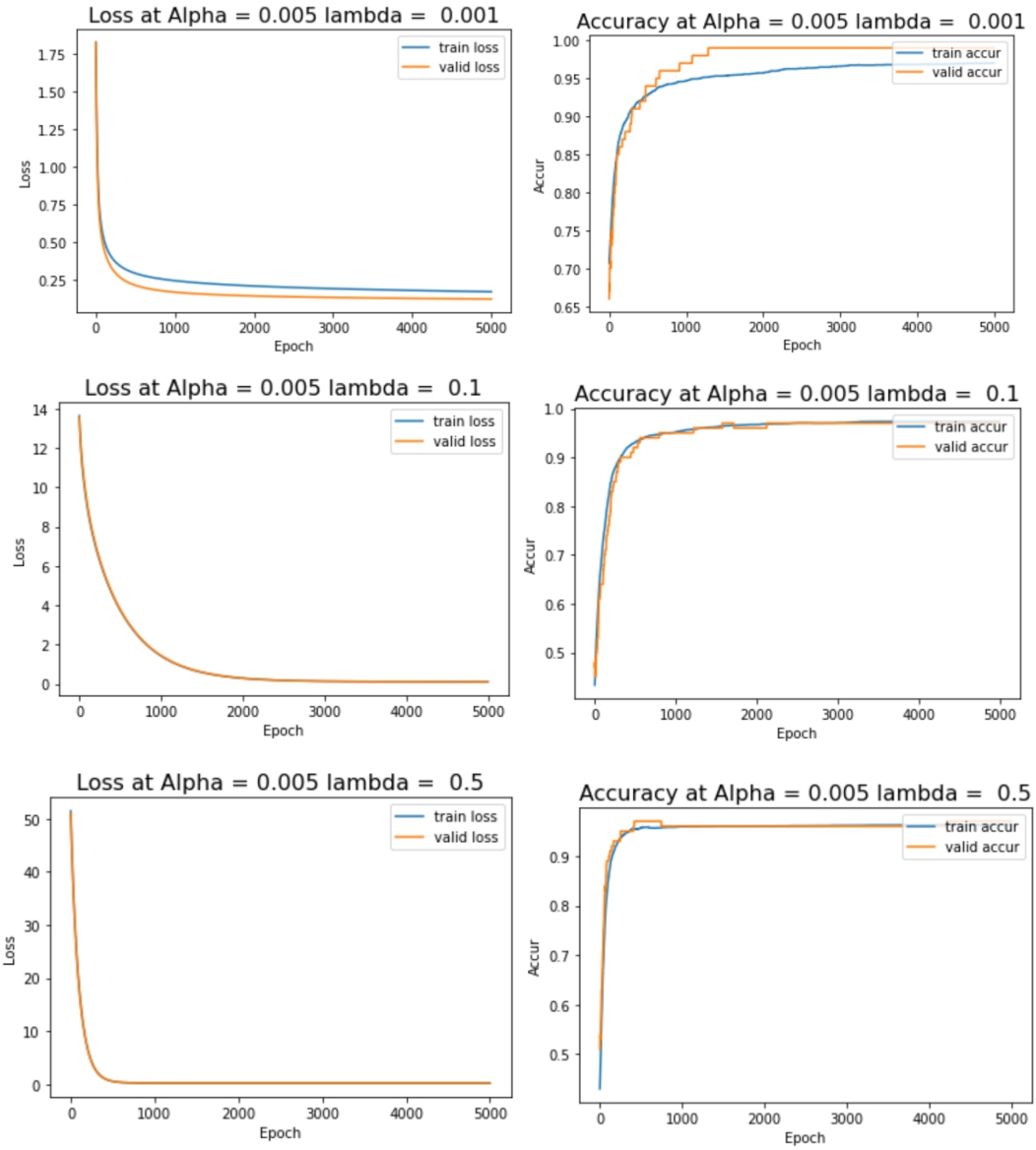


Figure5 training/validation loss/accuracy for different lambda

2 Logistic Regression in TensorFlow

2.1 Building the Computational Graph

```
def buildGraph(shape_x, alpha, beta1=None, beta2=None, epsilon=None):


    W = tf.Variable(tf.random.truncated_normal((shape_x, 1), mean=0, stddev=0.5))
    b = tf.Variable(0, dtype=tf.float32)
    x = tf.placeholder(tf.float32, shape=(None, shape_x))
    y = tf.placeholder(tf.float32, shape=(None, 1))
    reg = tf.placeholder(tf.float32, shape=None)

    z = tf.matmul(x,W)+b

    loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=z, labels=y)) + reg*tf.nn.l2_loss(W)
    optimizer = tf.train.AdamOptimizer(alpha).minimize(loss)
    train_prediction = tf.nn.sigmoid(z)

    return W, b, x, y, train_prediction, loss, optimizer, reg
```

Figure6 Initialize a TensorFlow computational graph

2.2 Implementing Stochastic Gradient Descent

```
def SGD(epochs, batch_size, alpha):
    lam = 0
    train_loss = []
    valid_loss = []
    train_accur = []
    valid_accur = []
    test_accur = []
    trainData, trainTarget, validData, validTarget, testData, testTarget = loader()
    trainData = trainData.reshape((trainData.shape[0], trainData.shape[1]*trainData.shape[2]))
    validData = validData.reshape((-1,validData.shape[1]*validData.shape[2]))
    testData = testData.reshape((-1,testData.shape[1]*testData.shape[2]))
    shape_x = trainData.shape[1]
    W, b, x, y, train_prediction, loss, optimizer, reg = buildGraph(shape_x,alpha)
    num_batch = int(len(trainTarget)/batch_size)
    with tf.Session() as session:
        session.run(tf.global_variables_initializer())
        for i in range(epochs):
            idx = np.random.permutation(len(trainTarget))
            trainData, trainTarget = trainData[idx], trainTarget[idx]
            loss_train=0
            accur_train=0
            for n in range(num_batch):
                batch_data = trainData[n*batch_size: (n+1)*batch_size]
                batch_label = trainTarget[n*batch_size: (n+1)*batch_size]
                train_p, train_l, label, op = session.run([train_prediction, loss, y, optimizer],
                                                          feed_dict={x: batch_data, y: batch_label, reg: lam})
                train_a = np.sum((train_p>0.5)==label)/batch_size#accuracy(train_p,trainTarget)
                loss_train+=train_l
                accur_train+=train_a
            train_loss.append(loss_train/num_batch)
            train_accur.append(accur_train/num_batch)

            p, valid_l, label = session.run([train_prediction, loss, y],
                                            feed_dict={x: validData, y: validTarget, reg: lam})
            valid_loss.append(valid_l)
            valid_accur.append(np.sum((p>0.5)==label)/label.shape[0])

            p, test_l, labelt = session.run([train_prediction, loss, y],
                                            feed_dict={x: testData, y: testTarget, reg: lam})
            test_accur.append(np.sum((p>0.5)==labelt)/len(labelt))

        print("train_accuracy is %d", np.max(train_accur))
        print("valid_accuracy is %d", np.max(valid_accur))
        print("train_loss is %d", np.min(train_loss))
        print("valid_loss is %d", np.min(valid_loss))
        plot1(train_loss, valid_loss, alpha, batch_size, "loss")
        plot1(train_accur, valid_accur, alpha, batch_size, "accur")
    return train_loss, valid_loss, train_accur, valid_accur
```

Figure7 Implement SGD using TensorFlow

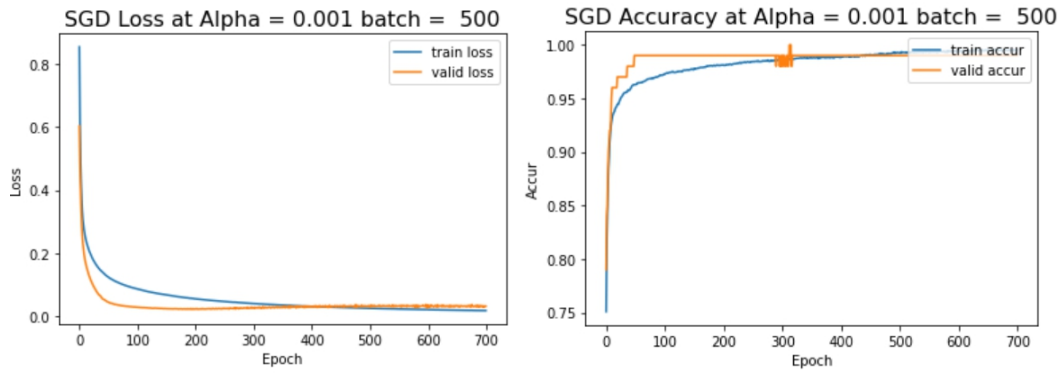


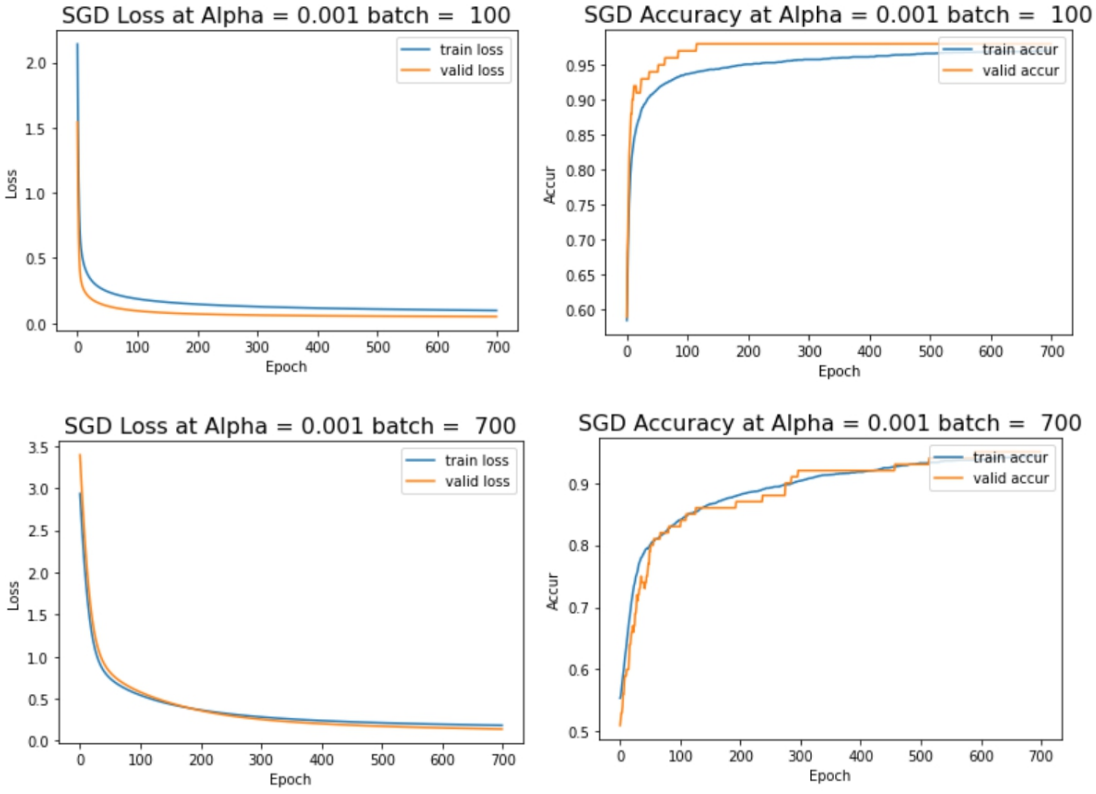
Figure8 SGD loss and accuracy

2.3 Batch Size Investigation

Batch size refers to the number of training examples used for each iteration. Train accuracy and valid accuracy are very high for the three different batch sizes. From the graphs, it's trivial that the rate of increase for accuracy is slower with larger batch size. Validation accuracy is the highest with batch size=100.

	Batch =100	Batch = 700	Batch = 1750
Best train accuracy	0.9994	0.99	0.975
Best valid accuracy	0.99	0.99	0.98
Lowest train loss	0.0065	0.02856	0.08757
Lowest valid loss	0.026	0.0297	0.0581

Table3 The best train/valid accuracy and lowest train/valid loss for different batch sizes



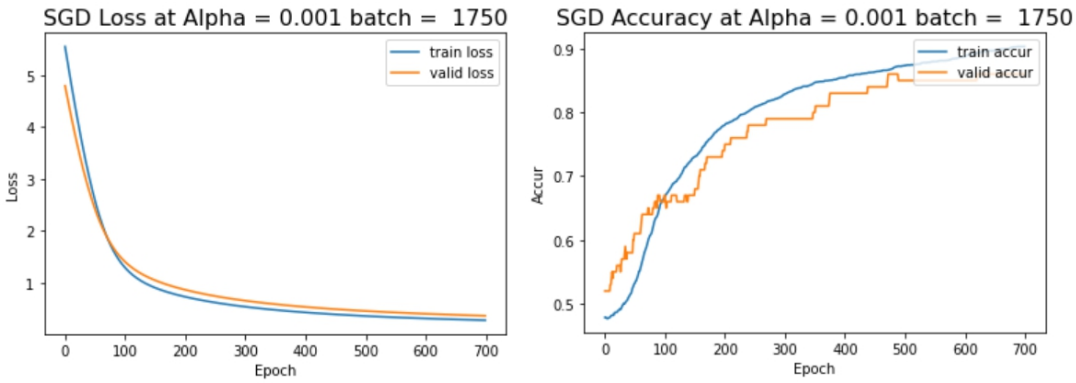


Figure9 SGD loss/accuracy for different batch sizes

2.4 Hyperparameter Investigation

Adam algorithm is a SGD algorithm. The estimation of 1st and 2nd moment gradient is based on hyperparameters (beta1, beta2, epsilon).

According to the algorithm shown in figure10, the hyperparameters will impact the update of parameters. Beta1 is the rate of exponential decay for the first moment. It adjusted the direction and step size during convergence. The adjustment can help the algorithm escape the saddle point, and thus lead to a faster convergence. Beta2 the exponential decay rate for the second-moment estimates. Same as beta1, it will change the direction and step size in a small range related to gradient descent. Since the second moment estimate will take square root, the range of change for beta2 should be smaller than beta1 to get a better control result. Epsilon is a mall number to avoid division of zero when updating parameters.

while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)

Figure10 Adam algorithm partial pseudo code[1]

Accuracy	β1=0.95	β1 =0.99	β2 = 0.99	β2 = 0.999	$\epsilon = 1e-9$	$\epsilon = 1e-4$
Train	0.992	0.994	0.997	0.993	0.995	0.995
Validation	0.97	0.99	1.0	0.98	0.98	0.99
Test	0.986	0.979	0.986	0.979	0.986	0.986

Table4 The train/validation/test accuracy for different hyperparameters

a) Although validation accuracy is essential for picking parameters, test accuracy shows the performance of an algorithm during actual usage. Test accuracy when beta1 = 0.95 is higher, so it is a

better choice. Beta1 updates based on accumulation of previous movements of gradient descent. Better performance for lower beta1 may because it takes less consideration of history values.

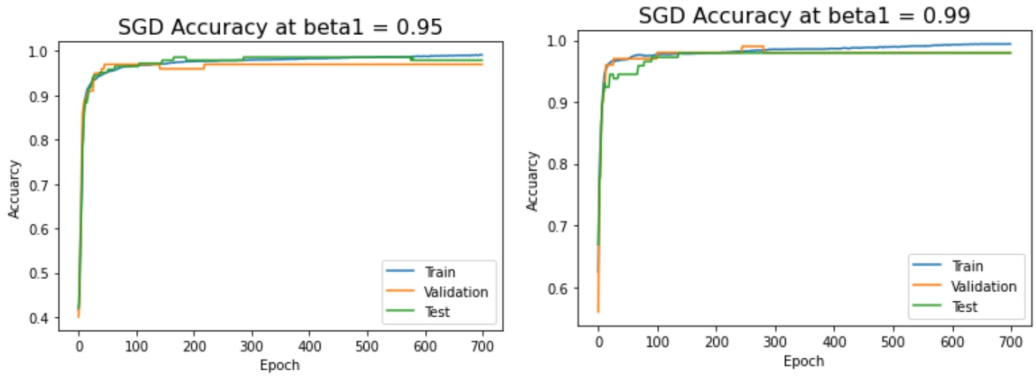


Figure11 SGD accuracy for different beta1

b) Beta2 = 0.99 validation accuracy is the highest. The validation and test accuracy are stable at about 200 epochs, which means a better performance than beta2=0.999. Beta2 also considers history values, so a lower beta2 will result in a more stable accuracy graph.

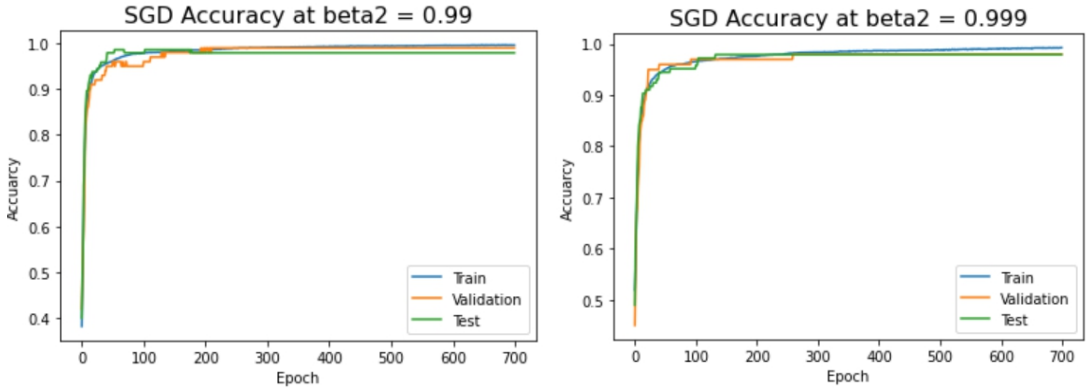


Figure12 SGD accuracy for different beta2

c) The performance of the two epsilon is very close. When epsilon=0.0001, the validation accuracy is slightly higher. The reason may be the larger effect of larger epsilon. If the epsilon is very small(e.g 1e-9), it may not be able to affect the parameters during update.

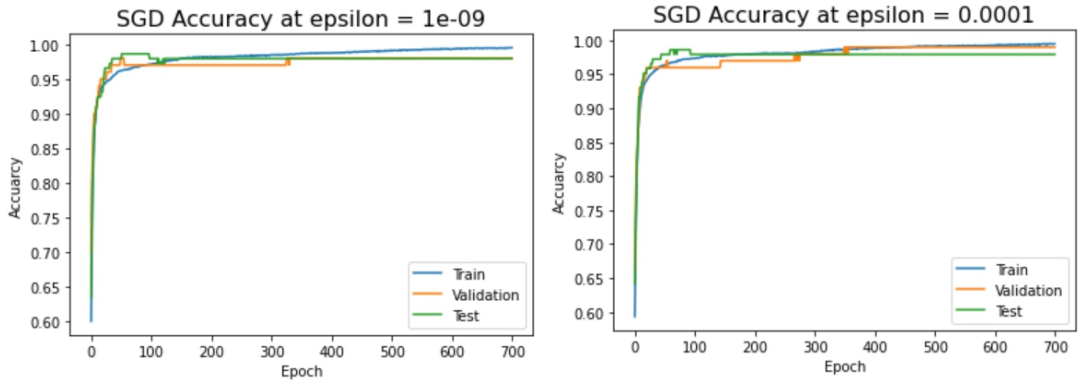


Figure13 SGD accuracy for different epsilon

2.5 Comparison against Batch GD

Batch gradient descent has validation accuracy outputs around 0.95, and the highest output is 0.96. Adam has validation accuracy all higher than 0.95 as result. Also, Adam is tested with 700 epochs, while batch gradient descent needs more than 1000 epochs to become stable. With the same learning rate and the same number of epoch, Adam has a better performance.

3 Reference

[1] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv e-prints, p. arXiv:1412.6980, Dec. 2014

4 Colab Link

https://colab.research.google.com/drive/1goIIcWC_w7gDJ9j5OwvpLyVXh-kjlVpu?usp=sharing