Assignment 1: Logistic Regression

By Ahoo Saeifar

ECE421-Winter 2021

February 8, 2021

Part1 Logistic Regression with Numpy

1.1 LOSS FUNCTION AND GRADIENT

$$L = L_{CE} \ + \ L_w$$
$$= \frac{1}{N}\sum_{n=1}^{N}[-y^{(n)}log(\hat{y}(\mathbf{x}^{(n)})) - (1-y^{(n)})log(1-\hat{y}(\mathbf{x}^{(n)}))] \ + \ \frac{\lambda}{2} \parallel \mathbf{w} \parallel_2^2$$

Where $\hat{y}(x) = \sigma(\mathbf{w}^T\mathbf{x} + b) = \sigma(z)$ and $\sigma(z) = \dfrac{1}{1+e^{-z}}$

We need to select b and **w** to minimize the loss therefore in taking the gradient, it should be taken once with respect to w and once with respect to b.
The gradients can be calculated using chain rule:

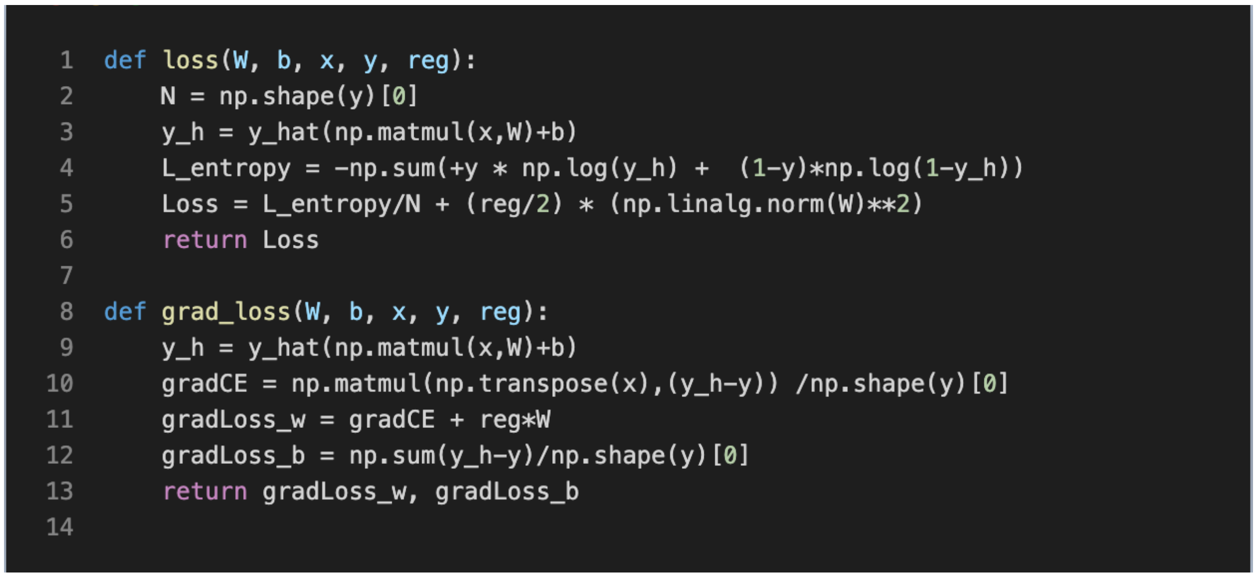
$$\frac{L_{CE}}{dw_j} = \frac{d}{dw_j}[-ylog(\sigma(\mathbf{w}^T\mathbf{x}+b)) - (1-y)log(1-\sigma(\mathbf{w}^T\mathbf{x}+b))]$$
$$= -\frac{y}{\sigma(\mathbf{w}^T\mathbf{x}+b)}\frac{d}{dw_j}\sigma(\mathbf{w}^T\mathbf{x}+b) - \frac{1-y}{\sigma(\mathbf{w}^T\mathbf{x}+b)}\frac{d}{dw_j}(1-\sigma(\mathbf{w}^T\mathbf{x}+b))$$

By plugging $\dfrac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$ in the equation above and continuing the chain rule we get:

$\dfrac{dL_{CE}}{dw_j} = [\sigma(\mathbf{w}^\mathbf{T}\mathbf{x}+b) - y]x_j = [\hat{y}(x) - y]x_j$ so overall $\dfrac{dL}{dw_j} = [\hat{y}(x_j) - y]x_j + \lambda w$

The following shows the derivation of gradient with respect to b: $\dfrac{dL}{db} = \dfrac{dL}{dz}\dfrac{dz}{db} = \dfrac{1}{N}\sum_{n=1}^{N}(\hat{y}(x) - y)$

The python code snippet below finds the loss and the gradient of loss with the help of Numpy library.

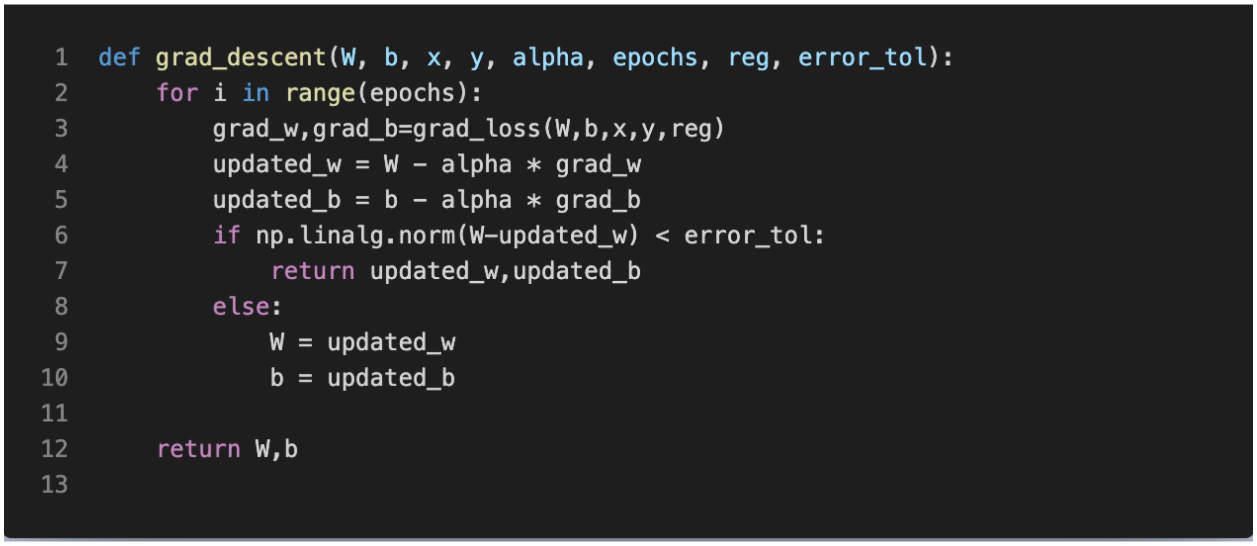


```
def loss(W, b, x, y, reg):
    N = np.shape(y)[0]
    y_h = y_hat(np.matmul(x,W)+b)
    L_entropy = -np.sum(+y * np.log(y_h) +  (1-y)*np.log(1-y_h))
    Loss = L_entropy/N + (reg/2) * (np.linalg.norm(W)**2)
    return Loss

def grad_loss(W, b, x, y, reg):
    y_h = y_hat(np.matmul(x,W)+b)
    gradCE = np.matmul(np.transpose(x),(y_h-y)) /np.shape(y)[0]
    gradLoss_w = gradCE + reg*W
    gradLoss_b = np.sum(y_h-y)/np.shape(y)[0]
    return gradLoss_w, gradLoss_b

```

1.2 GRADIENT DESCENT IMPLEMENTATION

In order to have our loss quickly converge or in other words to decrease the loss function quickly the parameters w and b must be updated in the opposite direction of their gradient. The update is as follows: $w_{updated} \leftarrow w - \alpha \nabla_w L \quad and \quad b_{updated} \leftarrow b - \alpha \nabla_b L$ where α is the learning rate and the larger it is the larger step the update takes.

The following python code snippet makes use of the grad_loss function defined in part 1.1 to update w and b values for a selected number of epochs or until the difference of the updated weight and the old weight is smaller than the error tolerance. This function was used for testing and plotting, the more extensive code snippet is available in Appendix A.



```python
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
    for i in range(epochs):
        grad_w,grad_b=grad_loss(W,b,x,y,reg)
        updated_w = W - alpha * grad_w
        updated_b = b - alpha * grad_b
        if np.linalg.norm(W-updated_w) < error_tol:
            return updated_w,updated_b
        else:
            W = updated_w
            b = updated_b

    return W,b

```

1.3 Tuning the Learning Rate

In this section the implementation of gradient descent was tested with 3 different learning rate values, $\alpha = \{0.005, 0.001, 0.0001\}$, 5000 epochs and $\lambda=0$. Figure 1 displays the 6 plots that were generated for the training/validation loss and accuracy. By analyzing the loss plots, it can be deduced that the larger learning rate makes the loss drop faster while training and as we reduce the learning rate it takes longer to achieve the same result over the same number of epochs. This observation can also be reached by comparing the testing accuracies given in table 1. Therefore a value of 0.005 for our learning rate is reasonable and it gives an accuracy of 97.24%.
This finding can further be justified by analyzing the update equations and by intuition we can see that the smaller learning rates require result in smaller changes made to the weights each update, whereas larger learning rates result in larger changes so they may require fewer training epochs to converge.

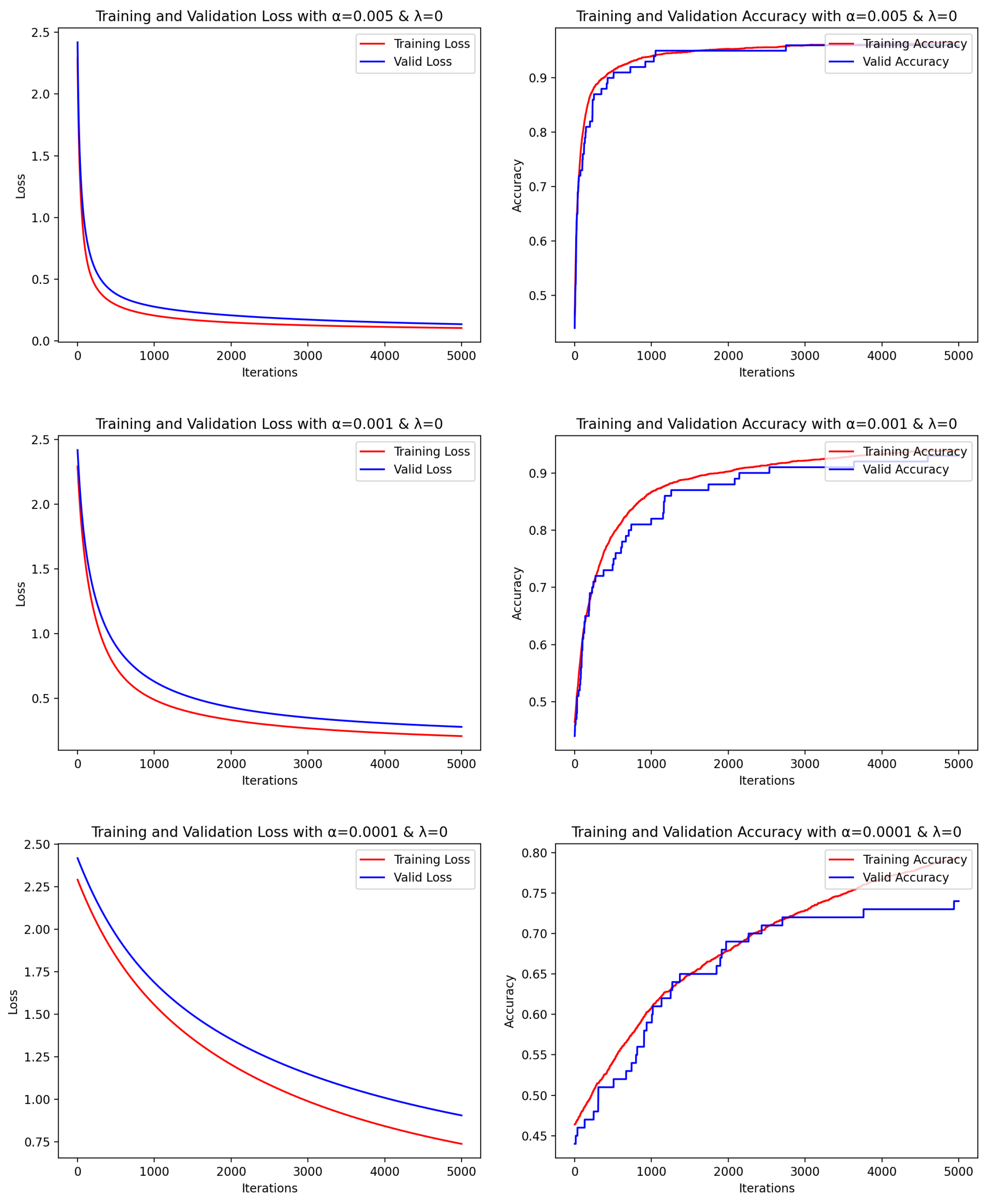


Figure 1. Training/Validation loss and accuracy over varying learning rates

	α=0.005	α=0.001	α=0.0001
Training Accuracy	97%	94%	79%
Valid Accuracy	96%	93%	74%
Testing Accuracy	97.24%	96.55%	80%

Table 1. Training, Valid and Testing Accuracies for varying learning rate

1.4 Generalization

In general regularization is added to prevent overfitting which happens when the model's performance on the training set improves while its' performance on the testing set starts to decline after a point. In this section the implementation of gradient descent was tested with 3 different regularization parameter values, λ = {0.001, 0.1, 0.5}, 5000 epochs and α=0.005. Figure 2 displays the 6 plots that were generated for the training/validation loss and accuracy. It can be observed from the plots that as the parameter is increased, the larger parameter results in a larger loss. λ=0.1 results in the best testing accuracy (98%) and prevents overfitting.

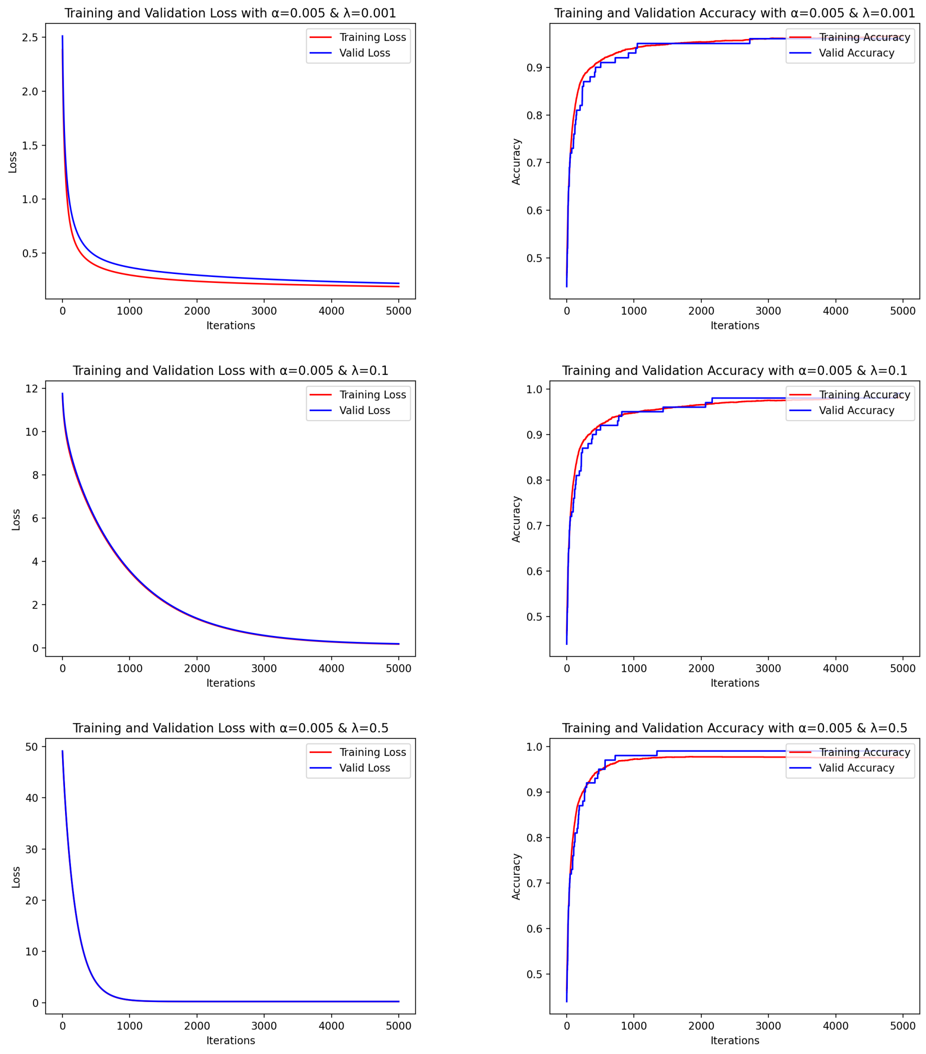


Figure 2. Training/Validation loss and accuracy over varying regularization parameter

	λ=0.001	λ=0.1	λ =0.5
Training Accuracy	96%	98%	98%
Valid Accuracy	96%	99%	99%
Testing Accuracy	97%	98%	97%

Table 2. Training, Valid and Testing Accuracies for varying regularization parameters

Part 2 Logistic Regression in TensorFlow

2.1 Building the Computational Graph

*helper functions in Appendix B

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

  minibatch_size = 1750
  alpha = 0.001
  W = tf.Variable(tf.random.truncated_normal(shape=(784, 1), mean=0.0, stddev=0.5, dtype=tf.float32,seed= None, name="W"))
  b = tf.Variable(tf.zeros(1),name="b")
  reg = 0
  x = tf.placeholder(tf.float32, (None, 784),name = "x")
  y = tf.placeholder(tf.float32, (None, 1),name = "y")


  valid_data = tf.placeholder(tf.float32, shape=(100, 784), name = "valid_data")
  valid_target = tf.placeholder(tf.int8, shape=(100, 1), name = "valid_target")

  test_data = tf.placeholder(tf.float32, shape=(145, 784), name = "test_data")
  test_target = tf.placeholder(tf.int8, shape=(145, 1), name="test_target")

  z = tf.matmul(x,W) + b
  y_hat = tf.sigmoid(z)
  loss = CELoss(x,y,W,b,reg)

  z_valid = tf.matmul(valid_data,W) + b
  y_hat_valid = tf.sigmoid(z_valid)
  validLoss = CELoss(valid_data,valid_target,W,b,reg)

  z_test = tf.matmul(test_data,W) + b
  y_hat_test = tf.sigmoid(z_test)
  testLoss = CELoss(test_data,test_target,W,b,reg)

  optimizer = tf.train.AdamOptimizer(learning_rate=alpha).minimize(loss)
```

2.2 Implementing Stochastic Gradient Descent

*helper functions in Appendix B

```
    # SGD implementation
    epochs = 700
    N = trainData.shape[0]

    # total number of batches required
    batchRange = int(N/minibatch_size)

    for step in range(epochs):
      #shuffling data
      newInd = np.arange(len(trainData))
      np.random.shuffle(newInd)
      trainData, trainTarget = trainData[newInd], trainTarget[newInd]
      for j in range(batchRange):
        #sampling
        XBatch = trainData[j*minibatch_size:(j+1)*minibatch_size]
        YBatch = trainTarget[j*minibatch_size:(j+1)*minibatch_size]

        my_dict = { x: XBatch, y: YBatch, valid_data: validData, valid_target: validTarget, test_data: testData,test_target: testTarget}
        opt, updated_w, updated_b, train_loss, pred_y, valid_loss, valid_pred, test_loss, test_pred = session.run([optimizer, W, b, loss,y_hat, validLoss,
y_hat_valid, testLoss, y_hat_test], feed_dict=my_dict)

      trainLossArr.append(train_loss)
      trainAccuracy.append(accuracy(updated_w,trainData,updated_b,trainTarget))

      validLossArr.append(valid_loss)
      validAccuracy.append(accuracy(updated_w,validData,updated_b,validTarget))

      testLossArr.append(test_loss)
      testAccuracy.append(accuracy(updated_w,testData,updated_b,testTarget))

  return trainLossArr,validLossArr,testLossArr,trainAccuracy,validAccuracy,testAccuracy
```

2.3 Batch Size Investigation
Figure 3 displays the training/validation accuracy and loss curves with learning rate of 0.001 and regularization constant of 0 over 3 different batch sizes. It can be observed that the accuracy goes down as a result of increased batch size. This observation is justified as a smaller batch size will make more optimizations leading to an increase in the accuracy overall. The final testing accuracies are very close for this model however a mini batch size of around 700 is adequate as it will reach the same accuracy as 100 mini-batch with less optimization.

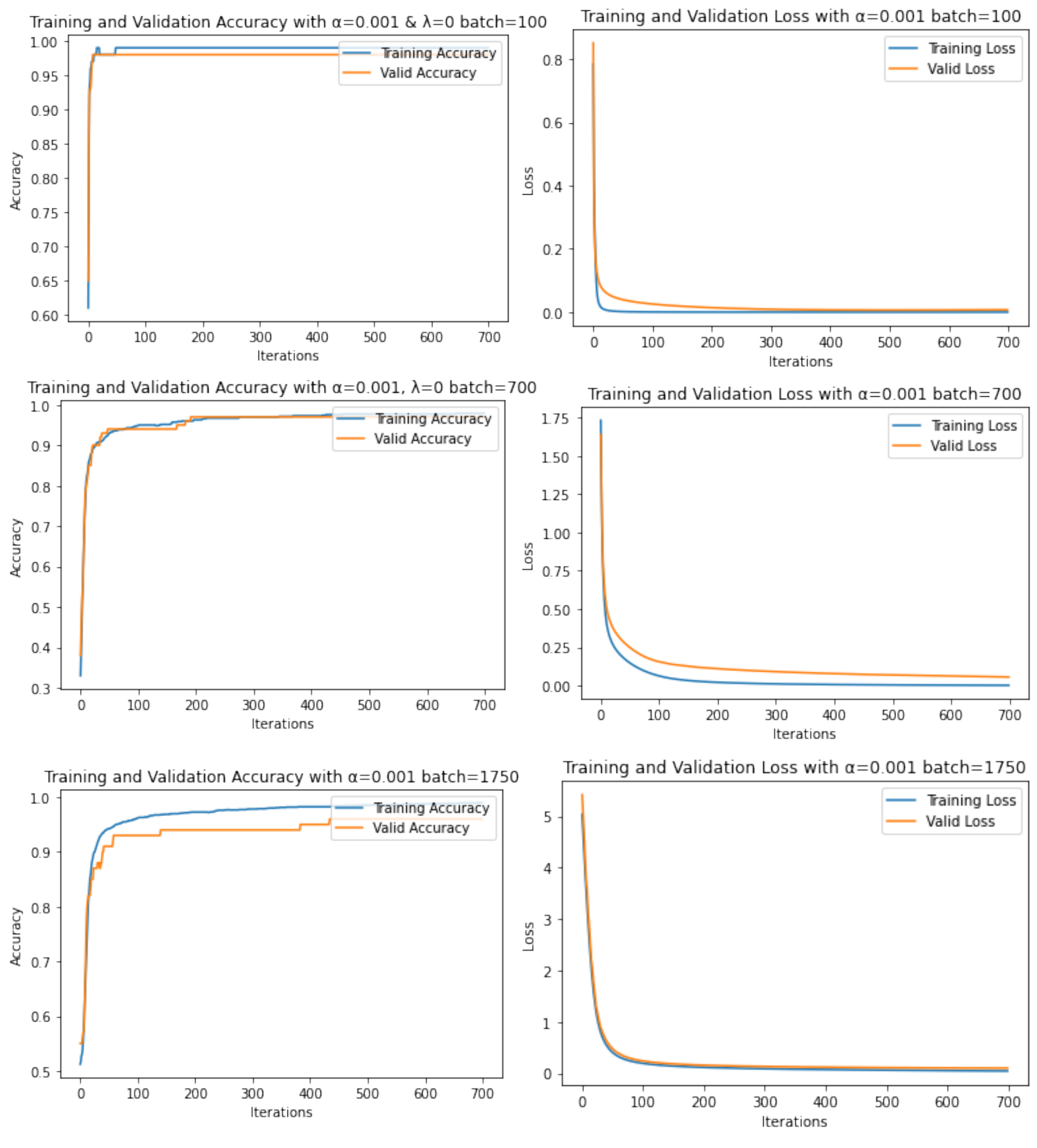


Figure 3. Training/Validation loss and accuracy for varying mini-batch size

Batch size	100	700	1750
Testing Accuracy	97%	97%	95%

Table 3. Testing Accuracies for varying mini-batch size

2.3 HYPERPARAMETER INVESTIGATION

We used Adam to optimize our model, Adam uses Momentum and Adaptive Learning Rates to converge faster. For choosing the best hyperparameters, the validation set needs to be analyzed therefore for the following section accuracy will refer to valid accuracy.

- $\beta 1$ is the first momentum term and in default it has a value of 0.9, this parameter speeds up the gradient descent resulting in a faster convergence. The better beta1 is 0.99 because as it was increased from 0.95 to 0.99 the accuracy improved very slightly.
- $\beta 2$ is the 2nd momentum term with default value of 0.999. It behaves similar to beta1 in the sense that it speeds up the convergence. In this case as well the smaller beta2 results in a higher accuracy, however again the difference is minimal.
- ε is a very small number that prevents division by zero in the implementation of the optimizer therefore it should not have a significant impact on the accuracy however the smaller epsilon resulted in a very slightly better accuracy.

It should be noted that the plots were generated using shuffled data every time during training so the slight differences could also be due to that.

In the table 4 below it can be seen that the overall accuracies between each parameter do not vary much and that could be due to SDG reaching an optimal model within 700 epochs quickly. Therefore changing these hyper parameters do not change our accuracies by large.

	β1 =0.95	β1=0.99	β2=0.99	β2=0.9999	ε = exp(-9)	ε = exp(-4)
Training Accuracy	99%	99%	99%	98%	99%	97%
Valid Accuracy	97%	95%	98%	96%	98%	97%
Testing Accuracy	98%	98%	97%	97%	99%	97%

Table 4. Accuracies for different hyper-parameter values

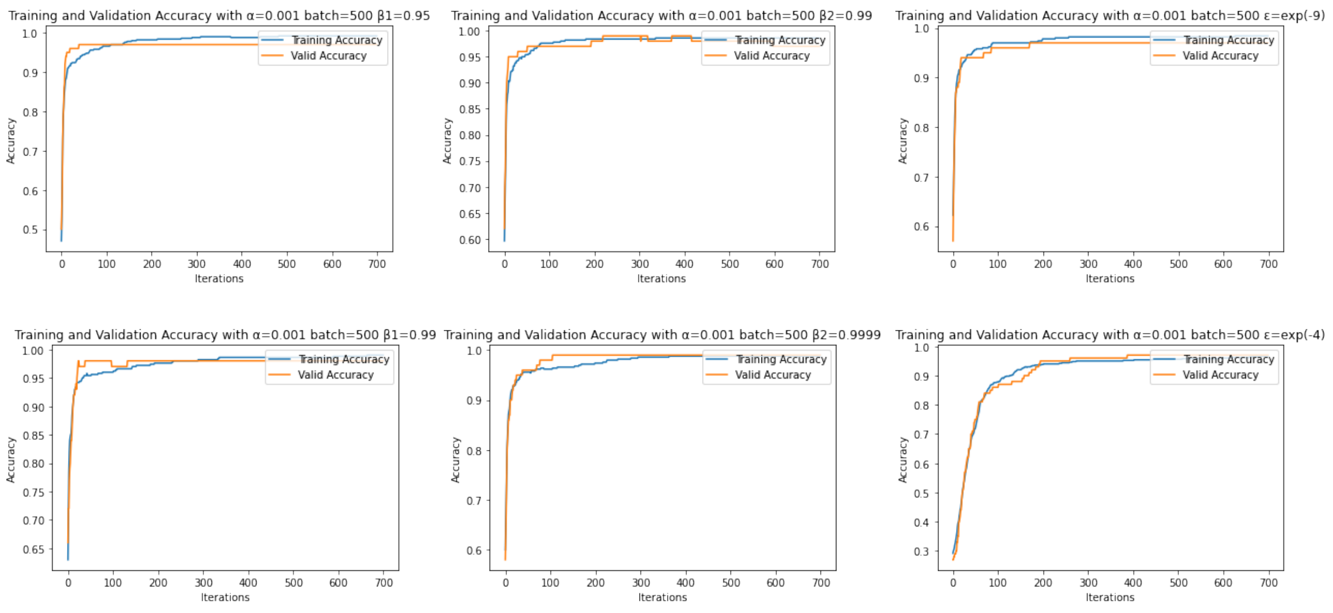


Figure 4. Training/Validation loss and accuracy for varying hyperparameters

2.5 Comparison against Batch GD

By analyzing the loss and accuracy plots that were generated in previous parts we can come to the following conclusion:

Loss:

SGD with Adam quickly gets to the minimum loss which is almost zero over 700 epochs whereas the batch GD needs 5000 epochs to achieve the same result. This is due to the fact that SGD with Adam, optimizes over more examples within the same computation time so it can converge faster than batch GD because it performs updates more frequently.

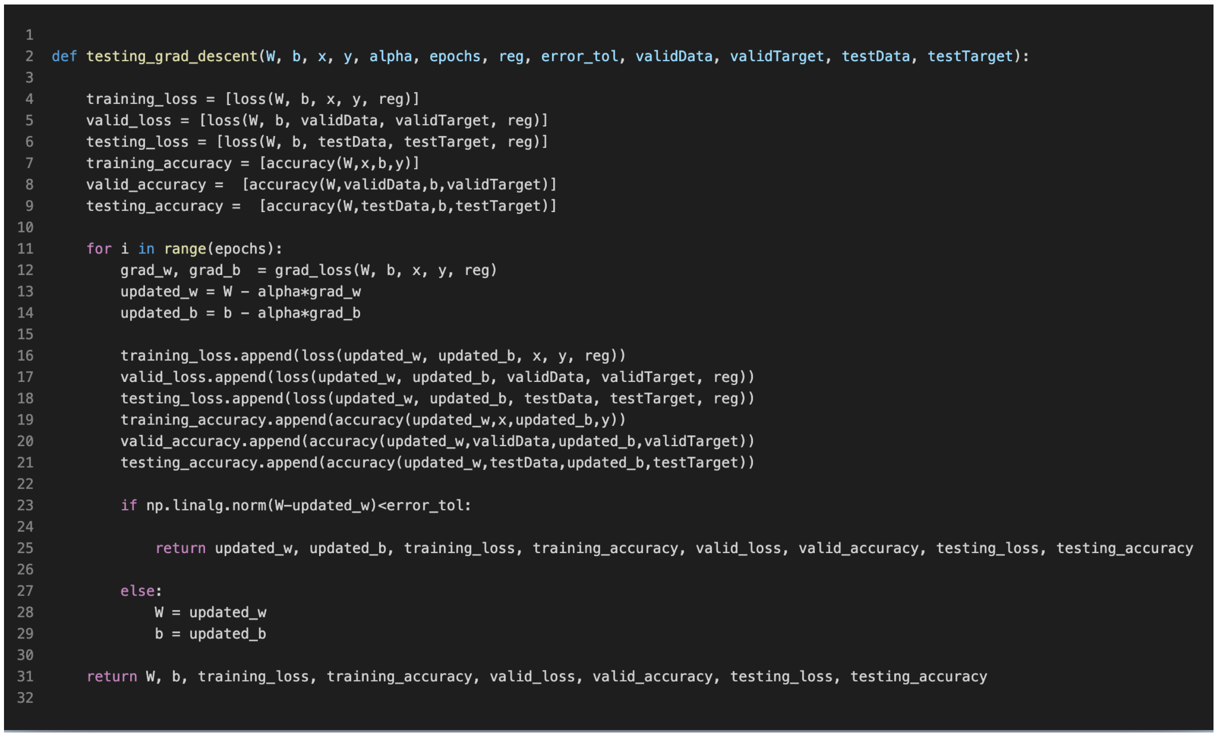
Accuracy:

The same observation as loss plots can be made for accuracy plots. The batch GD reaches a high accuracy over 5000 epochs while SGD with Adam reaches a high accuracy over 700. We can analyze further and see that SGD with Adam reaches approximately 90% accuracy within the first 100 epochs while it takes 1000 epochs for batch GD.

Finally, although they both resulted in the approximately same accuracy and loss, SGD with Adam is much more efficient as it quickly This observation is reasonable as batch GD moves directly downhill while SGD takes noisy steps but on average it goes in the right track.

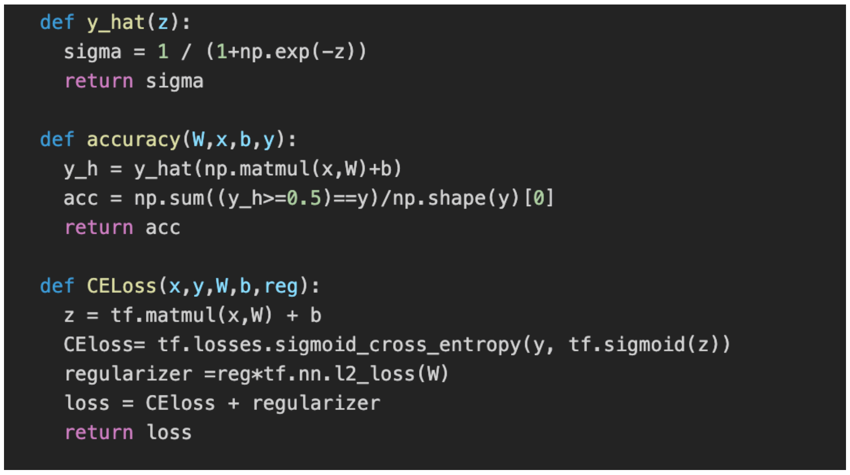
Appendix

A.



```

def testing_grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, validData, validTarget, testData, testTarget):

    training_loss = [loss(W, b, x, y, reg)]
    valid_loss = [loss(W, b, validData, validTarget, reg)]
    testing_loss = [loss(W, b, testData, testTarget, reg)]
    training_accuracy = [accuracy(W,x,b,y)]
    valid_accuracy =  [accuracy(W,validData,b,validTarget)]
    testing_accuracy =  [accuracy(W,testData,b,testTarget)]

    for i in range(epochs):
        grad_w, grad_b  = grad_loss(W, b, x, y, reg)
        updated_w = W - alpha*grad_w
        updated_b = b - alpha*grad_b

        training_loss.append(loss(updated_w, updated_b, x, y, reg))
        valid_loss.append(loss(updated_w, updated_b, validData, validTarget, reg))
        testing_loss.append(loss(updated_w, updated_b, testData, testTarget, reg))
        training_accuracy.append(accuracy(updated_w,x,updated_b,y))
        valid_accuracy.append(accuracy(updated_w,validData,updated_b,validTarget))
        testing_accuracy.append(accuracy(updated_w,testData,updated_b,testTarget))

        if np.linalg.norm(W-updated_w)<error_tol:

            return updated_w, updated_b, training_loss, training_accuracy, valid_loss, valid_accuracy, testing_loss, testing_accuracy

        else:
            W = updated_w
            b = updated_b

    return W, b, training_loss, training_accuracy, valid_loss, valid_accuracy, testing_loss, testing_accuracy

```

B.



```
def y_hat(z):
  sigma = 1 / (1+np.exp(-z))
  return sigma

def accuracy(W,x,b,y):
  y_h = y_hat(np.matmul(x,W)+b)
  acc = np.sum((y_h>=0.5)==y)/np.shape(y)[0]
  return acc

def CELoss(x,y,W,b,reg):
  z = tf.matmul(x,W) + b
  CEloss= tf.losses.sigmoid_cross_entropy(y, tf.sigmoid(z))
  regularizer =reg*tf.nn.l2_loss(W)
  loss = CEloss + regularizer
  return loss
```