Assignment 2:
Linear and Logistic Regression
Solution

March 23, 2018

1 Linear Regression

One possible implementation for linear regression

```
def buildGraph():
    # Variable creation
    W = tf.Variable(tf.truncated_normal(shape=[28*28,1], stddev=0.5),
                                                        name='weights')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float32, [None, 28*28], name='input_x')
    y_target = tf.placeholder(tf.float32, [None,1], name='target_y')
    Lambda = tf.placeholder("float32", name='Lambda')

    # Graph definition
    y_predicted = tf.matmul(X, W) + b

    # Error definition
    meanSquaredError = tf.reduce_mean(tf.square(y_predicted - y_target),
                                      name='mean_squared_error')
    weight_loss = tf.reduce_sum(W*W) * Lambda * 0.5
    loss = meanSquaredError + weight_loss

    # Training mechanism
    optimizer = tf.train.GradientDescentOptimizer(learning_rate = 0.005)
    train = optimizer.minimize(loss=loss)
    return W, b, X, y_target, y_predicted, meanSquaredError, train, Lambda
```

One possible implementation for SGD

```
B = 500
max_iter = 20000
wd_lambda = 0.1


wList = []
trainLoss_list = []
trainAcc_list = []
validLoss_list = []
validAcc_list = []
testLoss_list = []
testAcc_list = []
numBatches = np.floor(len(trainData)/B)

for step in range(0, max_iter + 1):
    if step % numBatches == 0:
        ## sample minibatch without replacement
        randIdx = np.arange(len(trainData))
        np.random.shuffle(randIdx)
        trainData = trainData[randIdx]
        trainTarget = trainTarget[randIdx]
        i = 0         # cyclic index for mini-batch

        # storing MSE and Acc for the three datasets every epoch
        err = meanSquaredError.eval(feed_dict = {X: trainData, y_target: trainTarget})
        acc = np.mean((y_predicted.eval(feed_dict={X: trainData}) > 0.5) == trainTarget)
        trainLoss_list.append(err)
        trainAcc_list.append(acc)

        err = meanSquaredError.eval(feed_dict = {X: validData, y_target: validTarget})
        acc = np.mean((y_predicted.eval(feed_dict={X: validData}) > 0.5) == validTarget)
        validLoss_list.append(err)
        validAcc_list.append(acc)

        err = meanSquaredError.eval(feed_dict = {X: testData, y_target: testTarget})
        acc = np.mean((y_predicted.eval(feed_dict={X: testData}) > 0.5) == testTarget)
        testLoss_list.append(err)
        testAcc_list.append(acc)

    # slicing a mini-batch from the whole training dataset
    feeddict = {X: trainData[i*B:(i+1)*B], y_target: trainTarget[i*B:(i+1)*B],
                            Lambda: wd_lambda}
    ## Update model parameters
    _, err, currentW, currentb, yhat = sess.run([train, meanSquaredError,
                                                 W, b, y_predicted],
```

```
                                                    feed_dict = feeddict)
    # storing weights every iteration
    wList.append(currentW)
    i += 1

    # displaying training MSE error every 100 iterations
    if not (step % 100):
        print("Iter: %3d, MSE-train: %4.2f"%(step, err))
```

1. **Tuning the learning rate**:
   Figure 1 shows the learning curves for $\eta = \{0.005, 0.001, 0.0001\}$. Using $\eta = 0.005$ gives the fastest convergence. Increasing η to 0.01 makes the gradient descent algorithm diverges.

2. **Effect of the mini-batch size**:
   Using the three mini-batches for training and the same number of iterations, the model converges approximately to the same training MSE (Table 1), so, in terms of computation time, $B = 500$ gives fastest convergence.

3. **Generalization**:
   From Table 2, the best λ is 0.1 and the test accuracy for this value is 97.2%

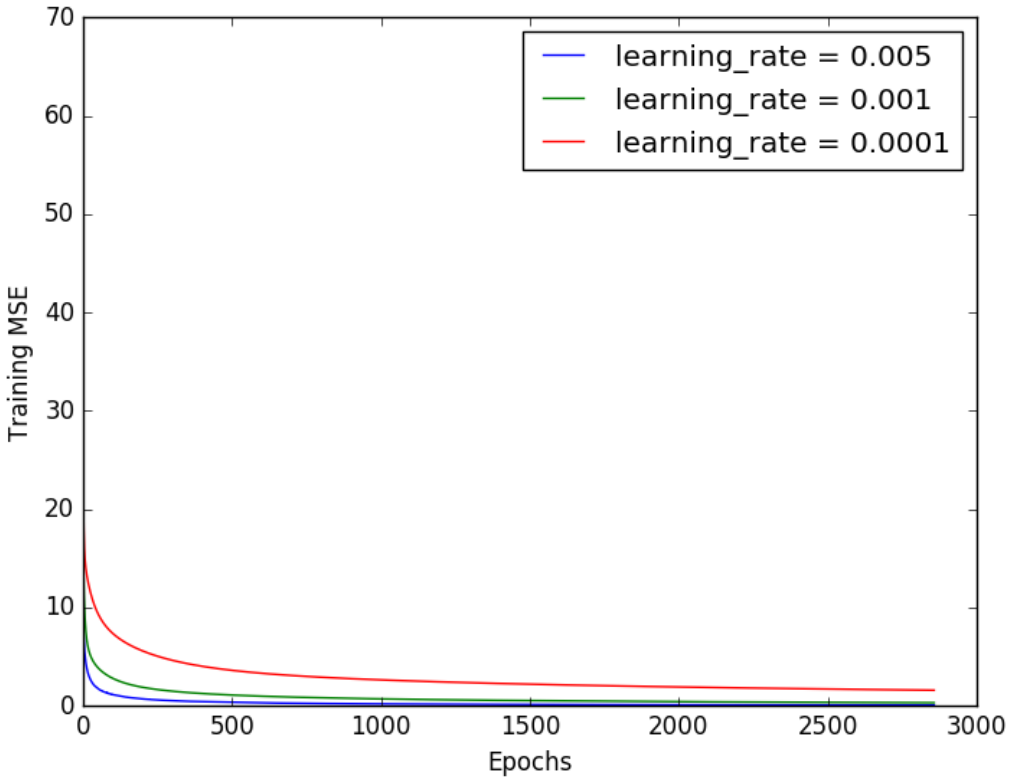


Figure 1: Learning curves for different learning rates

Table 1: Training MSE vs mini-batch size

B	Training MSE
500	0.08
1500	0.08
3500	0.09

Table 2: Validation accuracy vs λ

λ	Validation Accuracy (%)
0	90
0.01	95
0.1	98
1	97

4. **Comparing SGD with normal equation**:
Using normal equation, the final training MSE achieved is 0.018, while, using SGD with mini-batch 500, 0.005 learning rate and 40000 iterations, the final training MSE achieved was 0.04. Table 3 provides performance comparison between SGD and normal equation. Normal equation is much faster than SGD in terms of computation time, however, SGD is practical in case of high dimensional data (features $>$ 10000) and in case of no closed form optimum solution e.g. logistic regression.

2 Logistic Regression

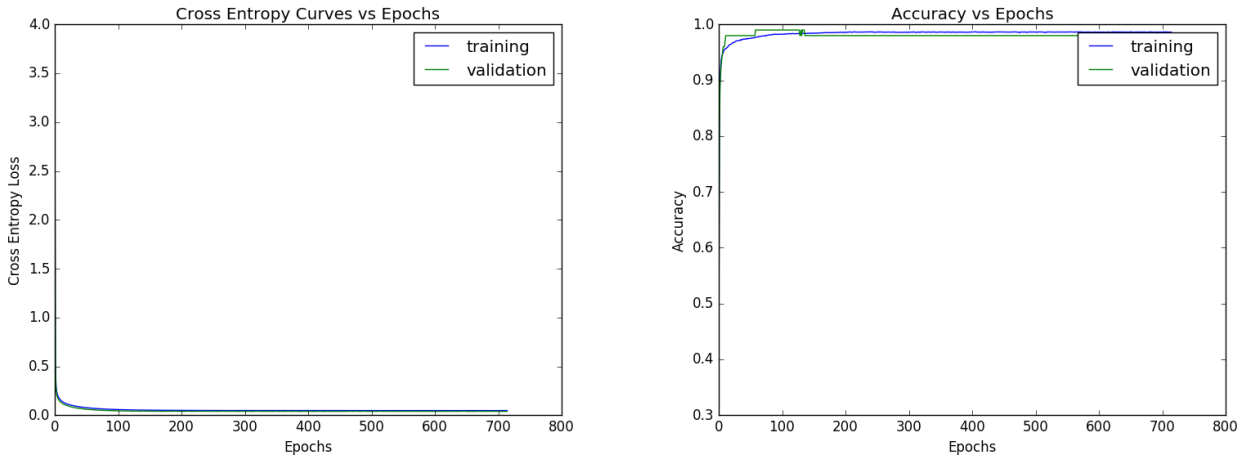
2.1 Binary cross-entropy loss

One possible implementation

```
def buildGraph():
    # Variable creation
    W = tf.Variable(tf.truncated_normal(shape=[28*28,1], stddev=0.5), name='weights')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float32, [None, 28, 28], name='input_x')
```

Table 3: SGD vs normal equation

	Linear Regression (Normal Eq)	Linear Regression (40000 iterations)
Training	99.34%	96.8%
Valid	97%	94%
Test	95.8%	94.5%



(a) (b)

Figure 2: Learning curves for logistic regression

```
X_flatten = tf.reshape(X, [-1, 28*28])
y_target = tf.placeholder(tf.float32, [None,1], name='target_y')
Lambda = tf.placeholder("float32", name='Lambda')

# Graph definition
y_logit = tf.matmul(X_flatten, W) + b
y_predicted = tf.nn.sigmoid(y_logit)

# Error definition
crossEntropyError = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels = y_t
                                    logits = y_logit),
                                    name='mean_cross_entropy')

weight_loss = tf.reduce_sum(W*W) * Lambda * 0.5

loss = crossEntropyError + weight_loss

# Training mechanism
optimizer = tf.train.GradientDescentOptimizer(learning_rate = 0.005)
#optimizer = tf.train.AdamOptimizer(learning_rate = 0.001)
train = optimizer.minimize(loss=loss)

return W, b, X, y_target, y_predicted, crossEntropyError, train, Lambda
```

1. **Learning**:
   Figure 2 shows the learning and accuracy curves for training and validation datasets for learning rate = 0.1. The best test accuracy achieved is 97.9%.

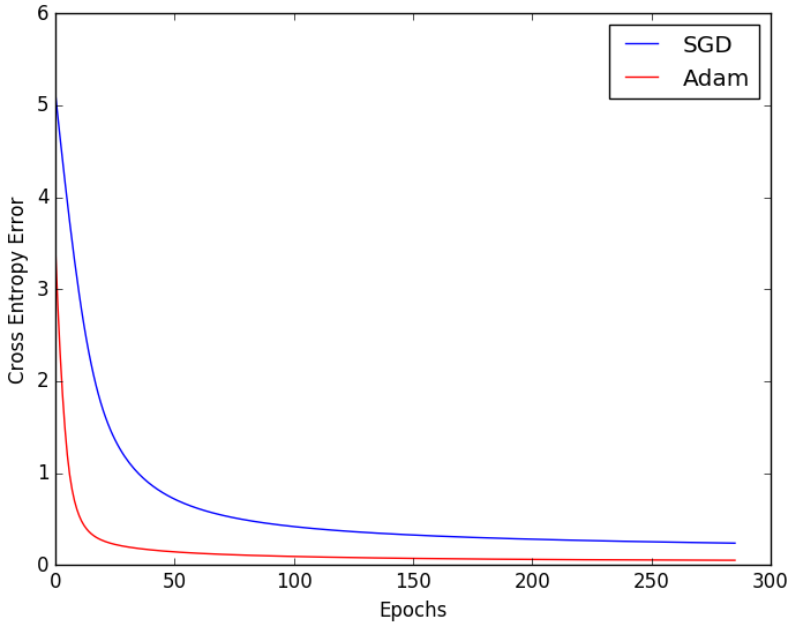


Figure 3: Learning curve of Adam vs SGD

Table 4: Linear and logistic regression comparison

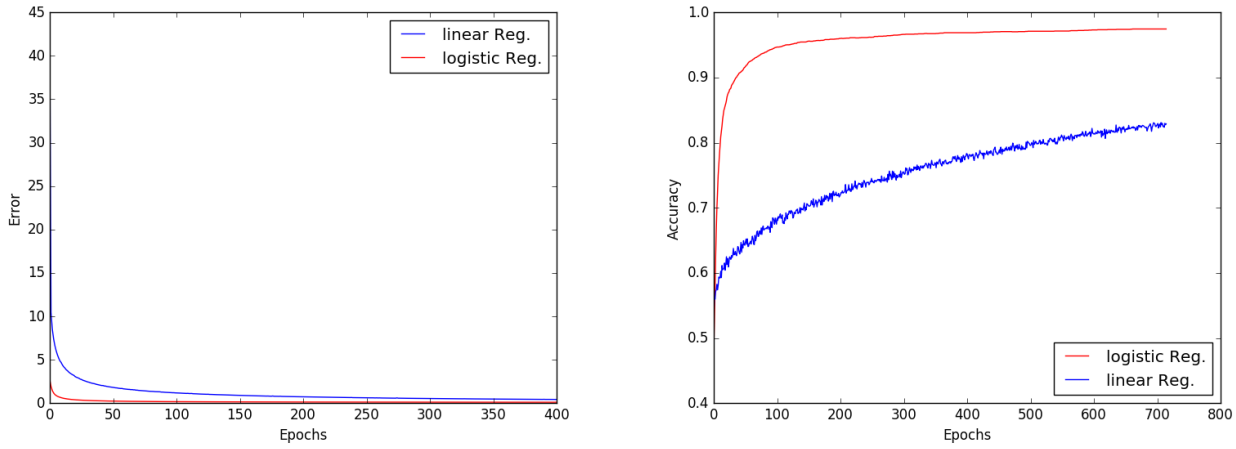
	Linear Regression (Normal Eq)	Logistic Regression
Training	99.34%	98.8%
Valid	97%	99%
Test	95.8%	97.9%

2. **Beyond plain SGD**:
   Figure 3 shows the cross entropy curves of SGD vs Adam Optimizer. For the same learning rate, Adam Optimizer shows faster convergence than SGD.

3. **Comparison with linear regression**:
   Logistic regression outperforms linear regression in terms of accuracy for validation and testing datasets as provided in Table 4. In terms of convergence (Figure 4), logistic regression provides faster convergence and this is explained in Figure 5. Compared with L_2 loss, the cross-entropy loss penalizes incorrect predictions more heavily. The slope (gradient) of the cross entropy loss increases exponentially as the prediction moves far away from the true target, however, for L_2 loss the slope is approximately constant, this explains the faster convergence of cross entropy for classification.

2.2 Multi-class classification

One possible implementation



(a) (b)

Figure 4: Comparing convergence of linear and logistic regression

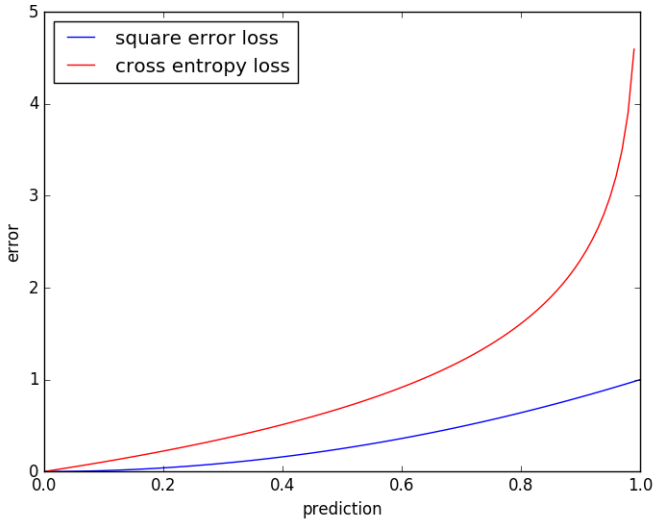


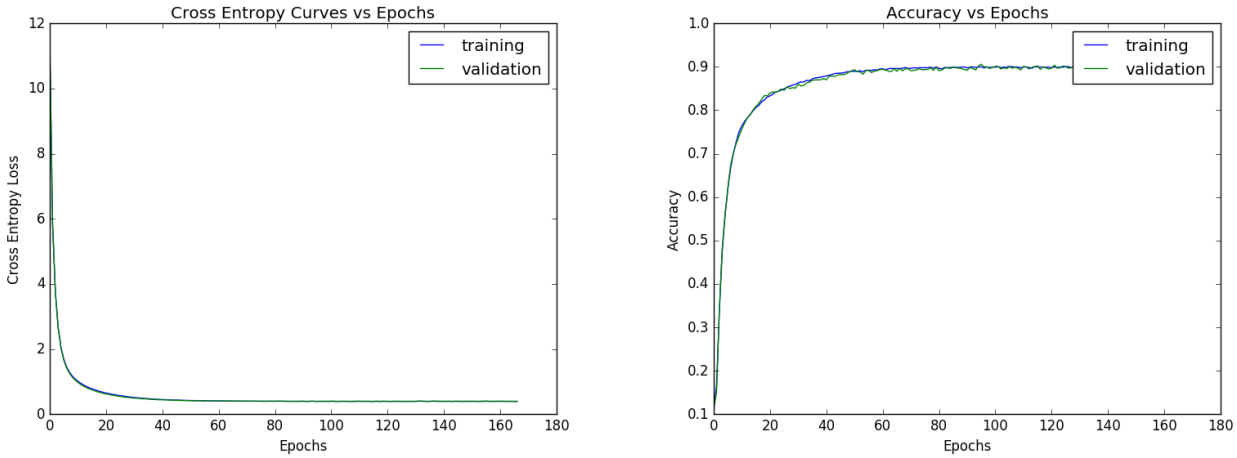
Figure 5: Square error loss vs cross entropy loss

```
def buildGraph():
    # Variable creation
    W = tf.Variable(tf.truncated_normal(shape=[28*28, 10], stddev=0.5), name='weights')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float32, [None, 28, 28], name='input_x')
    X_flatten = tf.reshape(X, [-1, 28*28])
    y_target = tf.placeholder(tf.float32, name='target_y')
    y_onehot = tf.one_hot(tf.to_int32(y_target), 10, 1.0, 0.0, axis = -1)
    Lambda = tf.placeholder("float32", name='Lambda')
```

```
# Graph definition
y_logit = tf.matmul(X_flatten, W) + b
y_predicted = tf.nn.softmax(y_logit)

# Error and accuracy definition
crossEntropyError = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(\
                                    labels = y_onehot, logits = y_logit),
                                    name='mean_cross_entropy')
acc = tf.reduce_mean(tf.to_float(tf.equal(tf.argmax(y_predicted, -1),
                                          tf.to_int64(y_target))))

weight_loss = tf.reduce_sum(W*W) * Lambda * 0.5
loss = crossEntropyError + weight_loss

# Training mechanism
#optimizer = tf.train.GradientDescentOptimizer(learning_rate = 0.005)
optimizer = tf.train.AdamOptimizer(learning_rate = 0.001)
train = optimizer.minimize(loss=loss)

return W, b, X, y_target, y_predicted, crossEntropyError, train, Lambda, acc
```

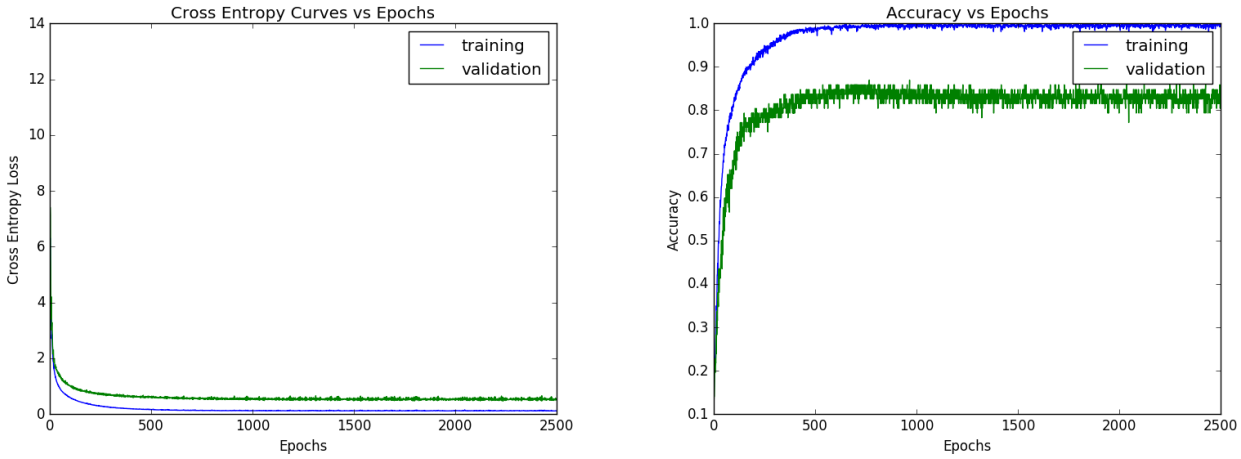
1. Figure 6 shows the learning curves for the multi-class classification task on *notMNIST* database using learning rate of 0.01. The best test accuracy achieved after convergence is 89.5%.

2. Figure 7 shows the learning curves for the multi-class classification task on *Facescrub* database using learning rate of 0.01 and weight decay of 0.001. The best test accuracy achieved after convergence is 82.8%. Also, Table 5 compares between logistic regression and k-NN.



(a) (b)

Figure 6: Learning curves for multi-class logistic regression on *notMNIST*



(a) (b)

Figure 7: Learning curves for multi-class logistic regression on *Facescrub*

Table 5: k-NN and logistic regression comparison

	k-NN ($k = 1$)	Logistic Regression
Training	100%	99.6%
Valid	66.3%	81.5%
Test	70.97%	82.8%