3116 – Lab Distributed Data Analytics – Group 2

Exercise Sheet 5

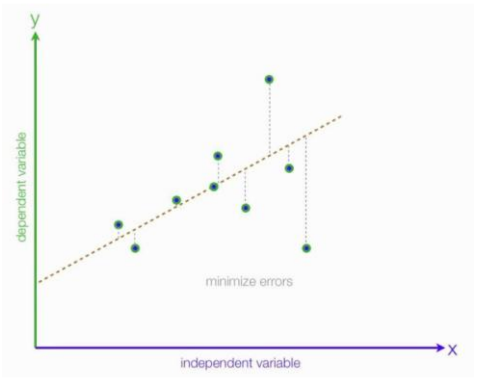
Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

Exercise 1: Linear Regression

There are 2 parts to this question, but both will follow the same linear regression methodology as previous lab. Both parts follow the similar code algorithm as outlined below. There are some differences in Part (a) and Part(b) as outlined below:

Univariate linear regression: The regression algorithm will model the relationship between a single feature (explanatory variable x) and a continuous valued response (target variable y). The relationship is modelled by setting an arbitrary line and computing the distance from this line to the data points. This distance are the residuals or prediction's errors. The regression algorithm will keep moving the line through each iteration, trying to find the best-fit line. This is the line with the minimum error.



For only one feature, the equation becomes:

$$Y = X.W + b$$

Where Weight, 'W' and bias, 'b' are scalars.

Multivariate linear regression: Now we have a set of input features $X = \{x_1, x_2, x_3 \dots x_n\}$ and the weights associated with it, $W = \{w_1, w_2, w_3 \dots w_n\}$. Thus, the equation with bias, 'b' becomes:

$$Y = \sum_{i=1}^{n} (X_i . W_i) + b$$

After the predicted values are calculated, the weights are iteratively calculated by loss functions. In part(a) – Mean Squared Error (MSE); and in part(b) – Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE).

MAE: The average of the absolute difference between actual data points and the predicted outcome. Each step of the gradient descent reduces the error.

$$MAE = \frac{1}{N} \sum |\hat{y} - y|$$

MSE: The average of the squared difference between actual data points and the predicted outcome. Each step of the gradient descent reduces the MSE. The preferred method to compute the best-fitting line, and it is also called Ordinary Least Squares (OLS).

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

RMSE: Simply the square root of MSE

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$$

Gradient Descent: Since both the datasets are relatively small, we are using Batch Gradient Descent. Parameters are updated after computing the gradient of error with respect to the entire training dataset. One important point to note here, for both parts the learning rate is made to be extremely small – since linear regression is a closed form solution, the small learning rate prevents a 'vanishing' or 'exploding' gradient descent problem. TensorFlow provides the built-in gradient descent optimizer.

Exercise 1A: Univariate Linear Regression

Step 1: Data generation

- Toy dataset generated based on the formula $y = 0.5x + 2 + \lambda$; where λ is Gaussian noise.
- X generated via numpy.random.uniform[size=1000]
- λ generated via numpy.random.normal[mean=0,standard deviation=50]
- Dataset split into testing/training data via sklearn.test_train_split.
- Scatter plot of ground truth to be plotted later on.

Step 2: Linear regression

- Define function "linear_regression" to calculate predicted value, $\hat{y}$ and MSE loss.

```
def LinearRegression():
    y_pred = tf.add(tf.multiply(X, W), b)
    loss = tf.reduce_mean(tf.square(tf.subtract(y_pred, Y)))
    return loss
```

Step 3: TensorFlow parameters

- Initialize W and b as TensorFlow get_variable() with zeros_initializers. If this is set to random, the model either converges fasters or slower, depending on the random values generated. Putting to zero (or any fixed value) gives a very consistent and predictable model performance when running the program many times.
- Initialize X and Y as TensorFlow placeholders.

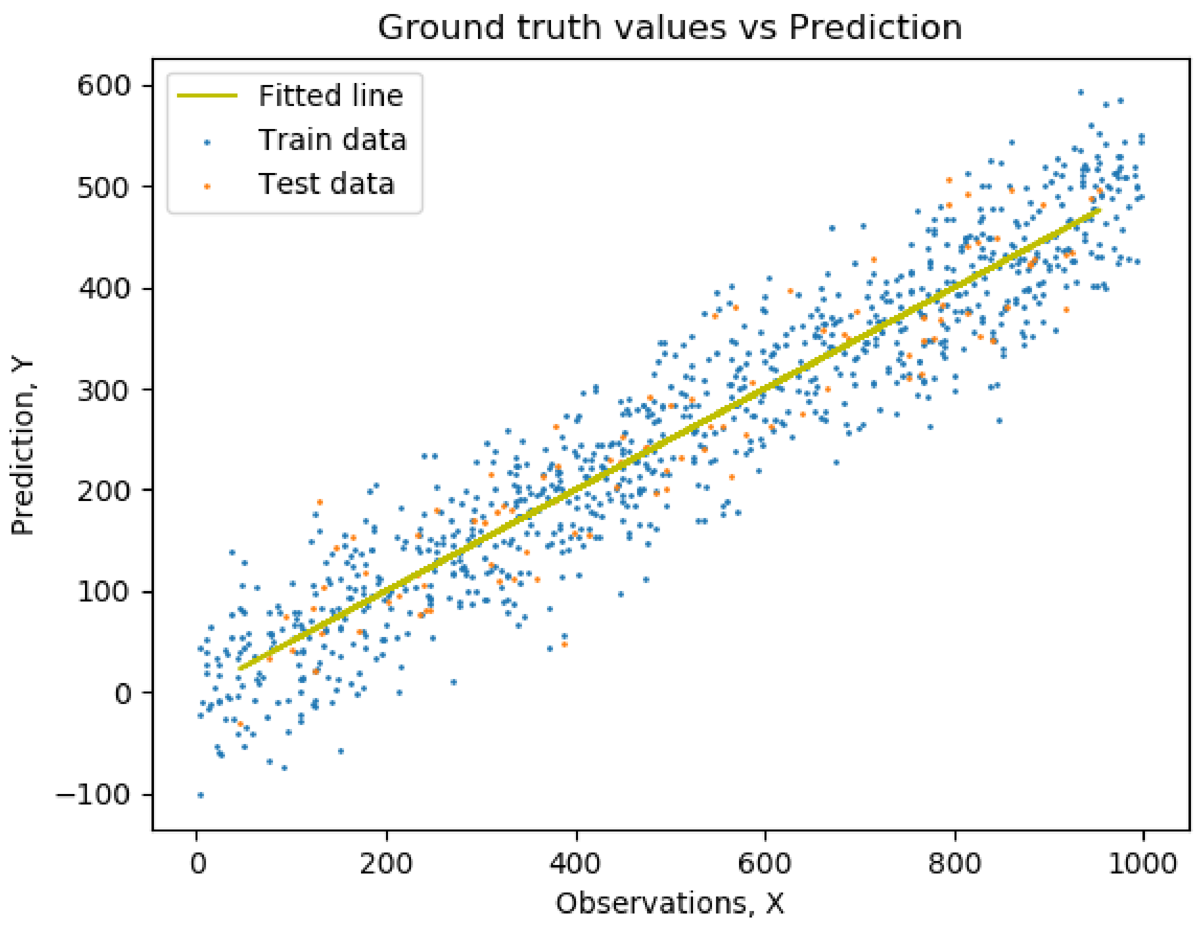
- Optimizer used is the 'GradientDescentOptimizer'. As mentioned above, the learning rate is set to 0.0000001, and the function will minimize the loss.

Step 4: TensorFlow session

- For the range of epochs, the program calculates and updates the W and B parameters, while converging the cost function to a minima.
- Number of epochs is set to 100, this is enough to fully converge.
- The model is then tested with the test dataset, and the line of best-fit (based on test data) is plotted against the ground truth values.

Results:

Full program output is in Appendix A; showing the line is reaching of Weight, W ≈ 0.5 with the respective bias, b. The test/train loss value is relatively large; however, this is expected due to the very large standard deviation. Lowering this value gives a lower loss accordingly. The predictions versus observations are shown as below:



Exercise 1B: Multivariate Linear Regression

Step 1: Data pre-processing

This is the most crucial step, as the bulk of the algorithm is same as Part 1A, and proper data pre-processing is necessary to properly build and verify the model.

- Data imported via pandas.read_csv. Column names are specified and passed as the parameter.
- Name column is dropped, since car names are duplicated and these kind of ID columns should not be part of the model anyway.
- NaN and unknown values are dropped.
- “Origin” categorical column are converted to numerical columns via one-hot encoding.
- X observations dataframe created, target column ‘mpg’ is dropped from X.
- Y features dataframe created, with target column ‘mpg’.
- X and Y are normalized, via sklearn.preprocessing.
- X and Y are split into training/testing dataset via sklearn.test_train_split.

Step 2: Multivariate linear regression function

- Same as part 1(a), except now W is a matrix and b is a vector.
- Formula for three loss functions are put here, with one active function at a time by uncommenting code.

```
def LinearRegression():
    y_pred = tf.add(tf.matmul(X, W), b)
    loss = tf.reduce_sum(tf.abs(y_pred - Y)) / nExamples  # Mean Absolute Error
    #loss = tf.reduce_mean(tf.square(tf.subtract(y_pred, Y))) # Mean Squared Error
    #loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(y_pred, Y)))) # Root Mean Squared Error
    return loss
```

Step 3: TensorFlow parameters

- Initialize W and b as TensorFlow get_variable() with zeros_initializers. If this is set to random, the model either converges fasters or slower, depending on the random values generated. Putting to zero (or any fixed value) gives a very consistent and predictable model performance when running the program many times.
- Initialize X and Y as TensorFlow placeholders.
- Optimizer used is the ‘GradientDescentOptimizer’. The learning rate is set to 0.00005, and the function will minimize the loss.

Step 4: TensorFlow session

- For the range of epochs, the program calculates and updates the W and B parameters, while converging the cost function to a minima.
- Number of epochs is set to 50000, this is enough to fully converge.
- The model is then tested with the test dataset. A scatter plot is impossible here since there are N-features, unless each feature is to be plotted independently.

MAE

Epoch	Loss value
5000	0.500225365
10000	0.000221096
15000	0.000060971
20000	0.000060323
25000	0.000059905
30000	0.000059579
35000	0.000059274
40000	0.000058972
45000	0.000058697
50000	0.000058476

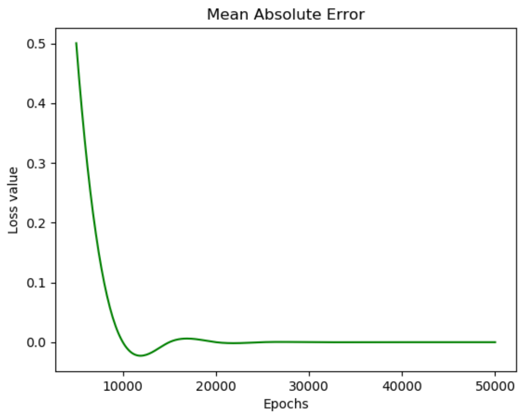
MSE

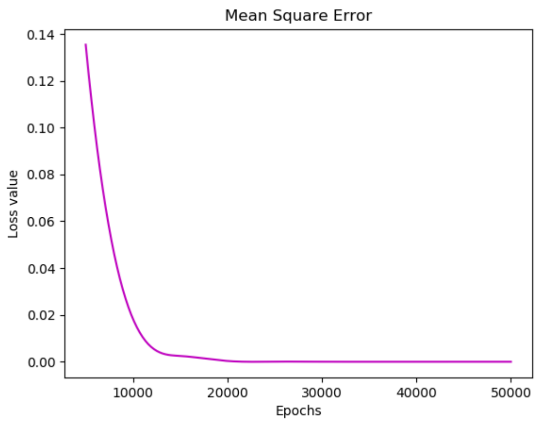
Epoch	Loss value
5000	0.135419831
10000	0.018331185
15000	0.002481434
20000	0.000335913
25000	0.000045486
30000	0.000006180
35000	0.000000867
40000	0.000000147
45000	0.000000042
50000	0.000000042

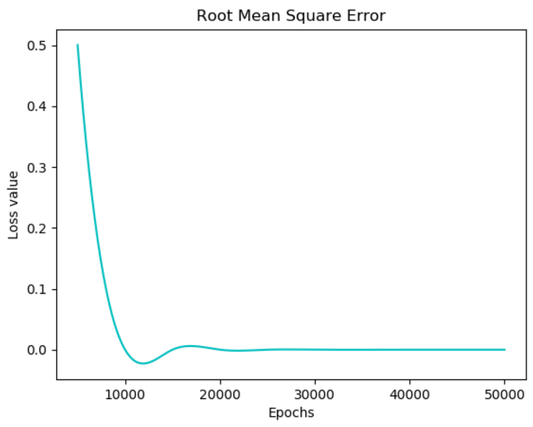
RMSE

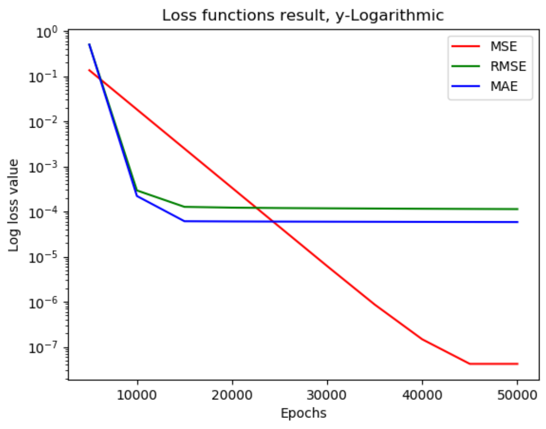
Epoch	Loss value
5000	0.500225306
10000	0.000293739
15000	0.000126985
20000	0.000121944
25000	0.000119484
30000	0.000117791
35000	0.000116400
40000	0.000115198
45000	0.000114148
50000	0.000113228

The plots for each three functions for training loss are shown below. Note that the dip in values is not a 'real' datapoint, but it is due to the line smoothing function used. The idea here is to show the curve, instead jagged points. Beside this, a y-logarithmic plot for all three training losses are included, since the order of magnitudes of the results are quite different. This makes it easier to visualise the performance. As observed from the table and curve plots, MSE loss function converges faster, and to a smaller minimum. However, the MAE and RMSE are not too far off. Bear in mind this is for 50,000 epochs and the dataset is quite small, so the differences here are not really significant. But still, it shows why MSE is the preferred cost computation for linear regression.









Exercise 2: Logistic Regression

While Linear Regression is useful to predict outcome based on some given features, Logistic Regression is useful to help classify an input given the input's features. We can adapt linear regression's $Y = W.X + b$ to work for logistic regression by merely transforming:

- feature vector, X
- prediction / outcome vector, Y / Y'
- cost function, H

The term "Logistic" is taken from the Logit function that is used in this method of classification. It allows categorizing data into discrete classes by learning the relationship from a given set of labelled data. It learns a linear relationship from the given dataset and then introduces non-linearity in the form of the Sigmoid function, which takes any real value between zero and one.

However, in this case I am using 'softmax regression' because this is multinomial logistic regression and we need to handle multiple class labels. Softmax function is just a generalization of the sigmoid function.

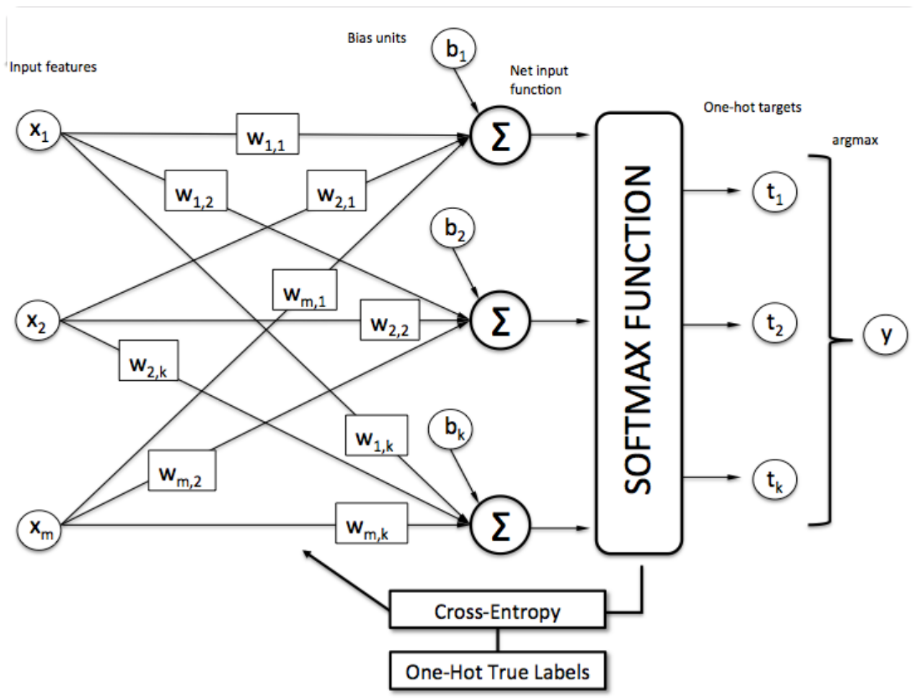


Figure 1: Softmax Regression[9]

Now we can apply cross-entropy (H) between the predicted vector score probability distribution (y') and the actual vector score probability distribution (y). This is what we want to minimize:

$$H_{y'}(y) = -\sum_{i} y'_i \log((softmax(y_i))$$

Step 1: Data pre-processing

This dataset contains a set of face images taken between April 1992 and April 1994 at AT&T Laboratories Cambridge. As described via the website: “There are ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement)”. The “target” for this database is an integer from 0 to 39 indicating the identity of the person pictured. The version used here consists of 64x64 images, so unfolding all the pixels into features creates a dataset made of 400 cases and 4,096 variables

- Read dataset from sklearn.datasets
- Test/Train split via sklean.train_test_split
- No normalization needed here
- Get number of features, number of labels

Step 2: Logistic Regression

- Logits found using same formula as predicted value of Y in linear regression
- Apply TensorFlow softmax function
- Initialize the accuracy parameter for epoch-ing

```
def logistic_regression():
    # https://www.geeksforgeeks.org/ml-logistic-regression-using-tensorflow/
    logits = tf.add(tf.matmul(X, W),b)
    y_pred = tf.nn.softmax(logits)
    prediction = tf.equal(tf.argmax(y_pred, 1), Y)
    accuracy = tf.reduce_mean(tf.cast(prediction, tf.float32))
    return logits,accuracy
```

Step 3: TensorFlow parameters

- Initialize W and b as TensorFlow get_variable() with zeros_initializers.
- Initialize X and Y as TensorFlow placeholders. Y type is [int64] due to the optimizer used.
- Three different optimizers are coded, only one will run at the time depending on the choosing.
    I. Batch gradient descent, learning rate = 0.01
    II. Adam, learning rate = 0.0001
    III. Adagrad, learning rate = 0.01

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(loss)
#optimizer = tf.train.AdagradOptimizer(learning_rate=0.01).minimize(loss)
#optimizer = tf.train.AdamOptimizer(learning_rate=0.0001).minimize(loss)
```

Step 4: TensorFlow session

- For the range of epochs, the program calculates and updates the W and B parameters, while converging the cost function to a minima, depending on the selected optimizer
- Number of epochs is set to 100, this is enough to fully converge.
- The model is then tested with the test dataset.
- Each selected optimizer has two plots, training loss, and train + test accuracy

Results:

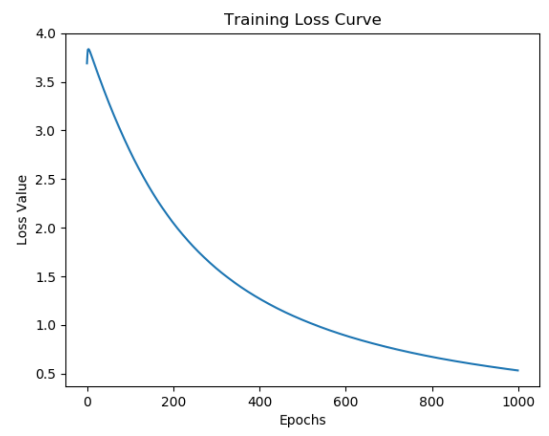
The program is run using all three different optimizers, and the training loss and test/train accuracy is plotted. Full program output is shown in Appendix C.

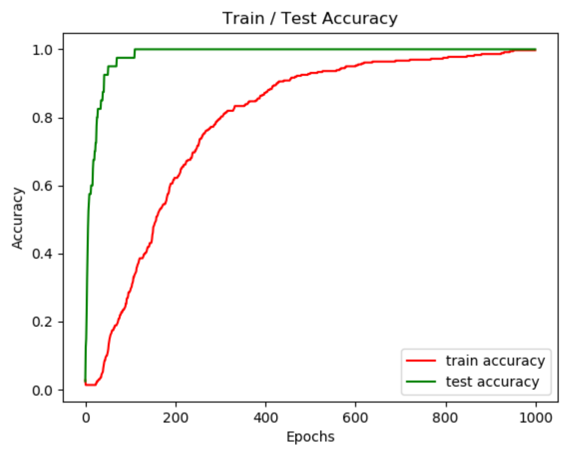
	Epoch	Loss value	Accuracy
Gradient Descent	200	2.055481911	0.62222
	400	1.273522139	0.87222
	600	0.892999947	0.95000
	800	0.673991978	0.97500
	1000	0.533832788	0.99722

	Epoch	Loss value	Accuracy
Adam	200	1.243102551	0.82500
	400	0.536206126	0.98611
	600	0.280819565	1.00000
	800	0.167816013	1.00000
	1000	0.109477267	1.00000

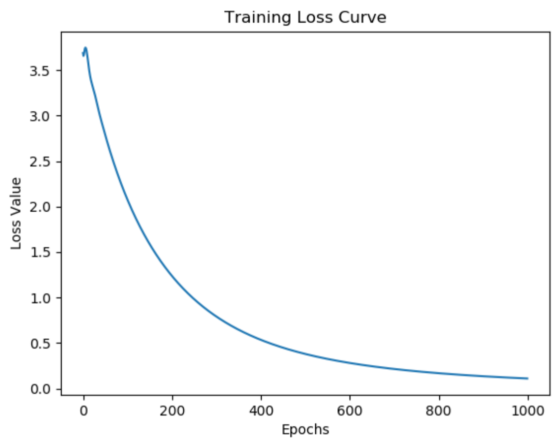
	Epoch	Loss value	Accuracy
Adagrad	200	0.911152442	0.93056
	400	0.448804259	0.99722
	600	0.284191638	1.00000
	800	0.203741878	1.00000
	1000	0.157232955	1.00000

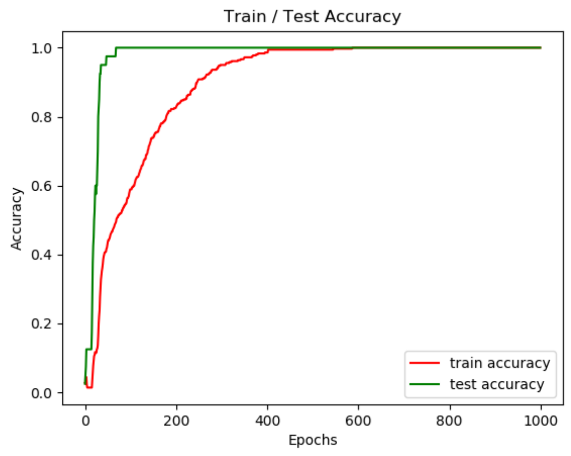
Gradient Descent Optimizer:



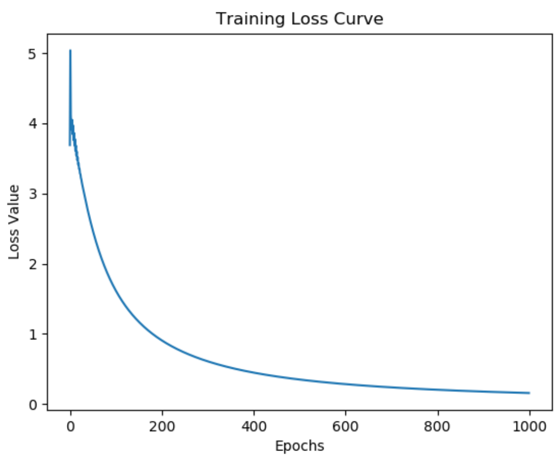


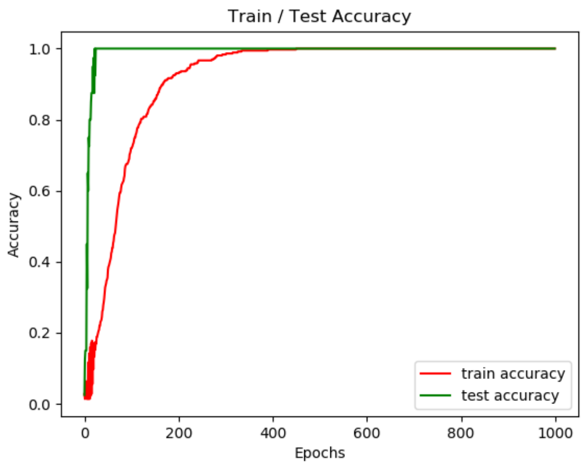
Adam Optimizer





Adagrad Optimizer





Observations:
Adam coverges the fastest, even with a much lower learning rate. Setting the learning rate to be equal as the other two makes it converge by 200 epochs. Adam is more stable than the other optimizers, it doesn't suffer any major decreases in accuracy. Overall, Adam is the best choice of optimizers. However, the other two are not fare of. In fact, Adagrad is almost similar in terms of performances, just falling short by a little bit. AdaGrad or adaptive gradient allows the learning rate to adapt based on parameters. It performs larger updates for infrequent parameters and smaller updates for frequent one. Because of this it is well suited for sparse data (image recognition). Adam or adaptive momentum is an algorithm similar to AdaDelta. But in addition to storing learning rates for each of the parameters it also stores momentum changes for each of them separately.

References:

[1] Zero initialiser for biases using get_variable in tensorflow. (4, 2). Retrieved from https://stackoverflow.com/questions/41821502/zero-initialiser-for-biases-using-get-variable-in-tensorflow

[2] Saha, S. (2018, December 9). An Introduction to TensorFlow and implementing a simple Linear Regression Model. Retrieved from https://medium.com/datadriveninvestor/an-introduction-to-tensorflow-and-implementing-a-simple-linear-regression-model-d900dd2e9963

[3] python machine_learning auto mpg dataset. (2016, December 19). Retrieved from https://fatihsarigoz.com/tag/python-machine_learning-auto-mpg-dataset.html

[4] Roman, V. (2019, January 15). Supervised Learning: Basics of Linear Regression. Retrieved from https://towardsdatascience.com/supervised-learning-basics-of-linear-regression-1cbab48d0eba

[5] AKM, A. (2018, November 5). Basics of linear regression. Retrieved from https://medium.com/datadriveninvestor/basics-of-linear-regression-9b529aeaa0a5

[6] Soon Hin Khor, Ph.D. (2017, February 5). Gentlest Intro to Tensorflow #4: Logistic Regression. Retrieved from https://medium.com/all-of-us-are-belong-to-machines/gentlest-intro-to-tensorflow-4-logistic-regression-2afd0cabc54

[7] 5.6.1. The Olivetti faces dataset — scikit-learn 0.19.2 documentation. (n.d.). Retrieved June 2, 2019, from https://scikit-learn.org/0.19/datasets/olivetti_faces.html

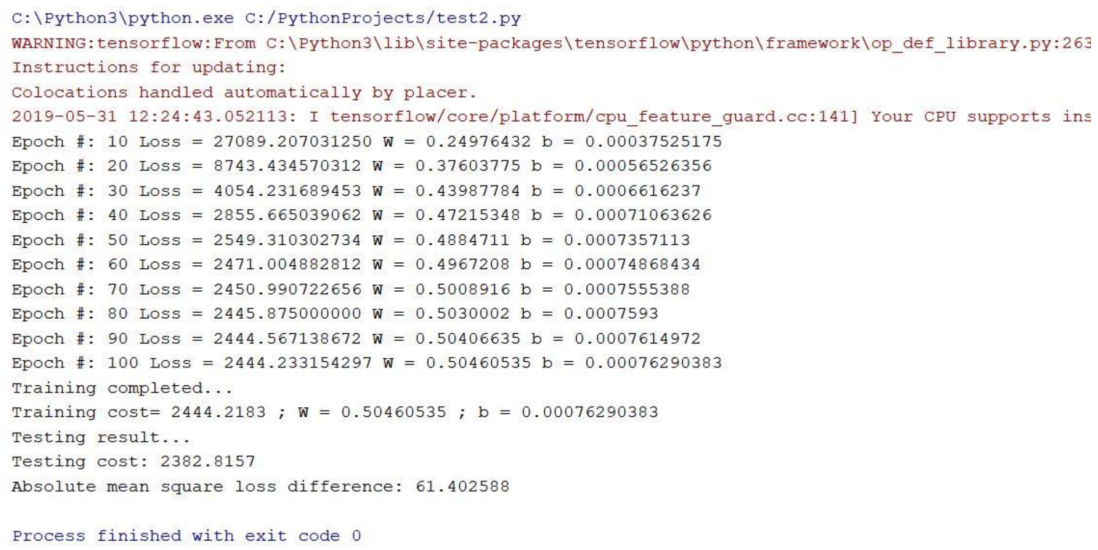
[8] What is Softmax Regression and How is it Related to Logistic Regression? (n.d.). Retrieved from https://www.kdnuggets.com/2016/07/softmax-regression-related-logistic-regression.html

[9] Logits representation in TensorFlow's sparse_softmax_cross_entropy. (1, 1). Retrieved from https://stackoverflow.com/questions/49883631/logits-representation-in-tensorflow-s-sparse-softmax-cross-entropy

[10]Gradient Descent vs Adagrad vs Momentum in TensorFlow. (2, 3). Retrieved from https://stackoverflow.com/questions/36162180/gradient-descent-vs-adagrad-vs-momentum-in-tensorflow

Appendix A

Sample output for Exercise 1 A:

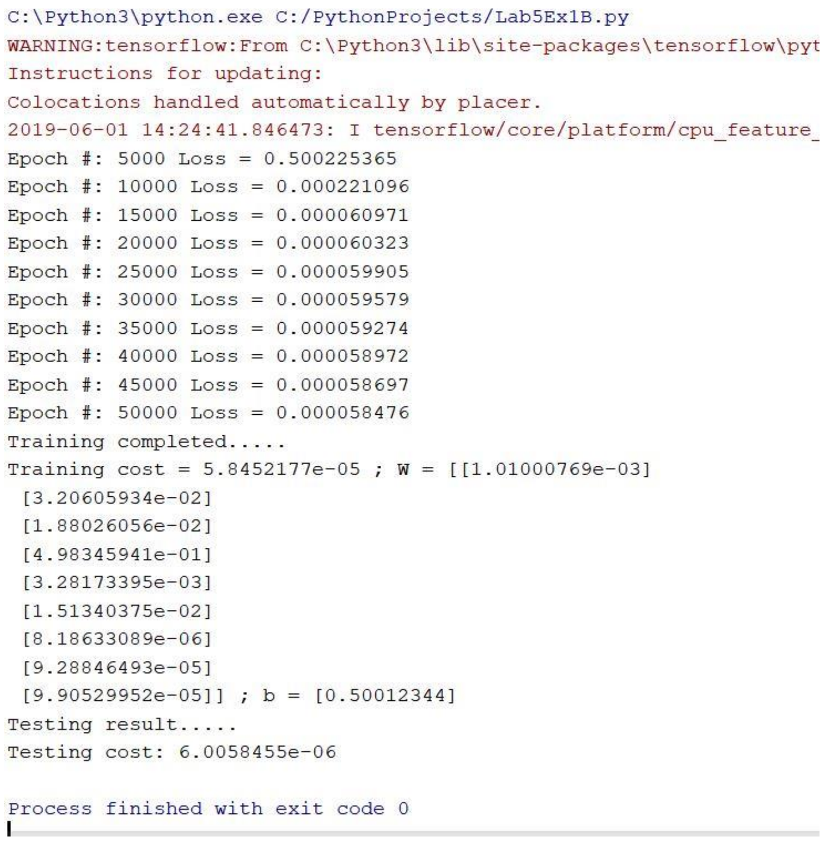


```
C:\Python3\python.exe C:/PythonProjects/test2.py
WARNING:tensorflow:From C:\Python3\lib\site-packages\tensorflow\python\framework\op_def_library.py:263
Instructions for updating:
Colocations handled automatically by placer.
2019-05-31 12:24:43.052113: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports ins
Epoch #: 10 Loss = 27089.207031250 W = 0.24976432 b = 0.00037525175
Epoch #: 20 Loss = 8743.434570312 W = 0.37603775 b = 0.00056526356
Epoch #: 30 Loss = 4054.231689453 W = 0.43987784 b = 0.0006616237
Epoch #: 40 Loss = 2855.665039062 W = 0.47215348 b = 0.00071063626
Epoch #: 50 Loss = 2549.310302734 W = 0.4884711 b = 0.0007357113
Epoch #: 60 Loss = 2471.004882812 W = 0.4967208 b = 0.00074868434
Epoch #: 70 Loss = 2450.990722656 W = 0.5008916 b = 0.0007555388
Epoch #: 80 Loss = 2445.875000000 W = 0.5030002 b = 0.0007593
Epoch #: 90 Loss = 2444.567138672 W = 0.50406635 b = 0.0007614972
Epoch #: 100 Loss = 2444.233154297 W = 0.50460535 b = 0.00076290383
Training completed...
Training cost= 2444.2183 ; W = 0.50460535 ; b = 0.00076290383
Testing result...
Testing cost: 2382.8157
Absolute mean square loss difference: 61.402588

Process finished with exit code 0
```

Appendix B

Sample output for Exercise 1B, MAE loss:



```
C:\Python3\python.exe C:/PythonProjects/Lab5Ex1B.py
WARNING:tensorflow:From C:\Python3\lib\site-packages\tensorflow\pyt
Instructions for updating:
Colocations handled automatically by placer.
2019-06-01 14:24:41.846473: I tensorflow/core/platform/cpu_feature_
Epoch #: 5000 Loss = 0.500225365
Epoch #: 10000 Loss = 0.000221096
Epoch #: 15000 Loss = 0.000060971
Epoch #: 20000 Loss = 0.000060323
Epoch #: 25000 Loss = 0.000059905
Epoch #: 30000 Loss = 0.000059579
Epoch #: 35000 Loss = 0.000059274
Epoch #: 40000 Loss = 0.000058972
Epoch #: 45000 Loss = 0.000058697
Epoch #: 50000 Loss = 0.000058476
Training completed.....
Training cost = 5.8452177e-05 ; W = [[1.01000769e-03]
 [3.20605934e-02]
 [1.88026056e-02]
 [4.98345941e-01]
 [3.28173395e-03]
 [1.51340375e-02]
 [8.18633089e-06]
 [9.28846493e-05]
 [9.90529952e-05]] ; b = [0.50012344]
Testing result.....
Testing cost: 6.0058455e-06

Process finished with exit code 0
```

Appendix C

GradientDescentOptimizer, learning rate=0.01

```
Epoch #: 200 Training Loss = 2.055481911 Training accuracy= 0.62222
Epoch #: 400 Training Loss = 1.273522139 Training accuracy= 0.87222
Epoch #: 600 Training Loss = 0.892999947 Training accuracy= 0.95000
Epoch #: 800 Training Loss = 0.673991978 Training accuracy= 0.97500
Epoch #: 1000 Training Loss = 0.533832788 Training accuracy= 0.99722
Training completed.....
Final training loss: {} 0.5338327884674072
Train accuracy: 0.9972222447395325

Cost in test set: 0.0949879139661789
Test accuracy: 1.0
```

AdagradOptimizer, learning rate=0.01

```
Epoch #: 200 Training Loss = 0.911152422 Training accuracy= 0.93056
Epoch #: 400 Training Loss = 0.448804259 Training accuracy= 0.99722
Epoch #: 600 Training Loss = 0.284191638 Training accuracy= 1.00000
Epoch #: 800 Training Loss = 0.203741878 Training accuracy= 1.00000
Epoch #: 1000 Training Loss = 0.157232955 Training accuracy= 1.00000
Training completed.....

Final training loss: {} 0.15723295509815216
Train accuracy: 1.0

Cost in test set: 0.02857527695596218
Test accuracy: 1.0
```

AdamOptimizer

```
Epoch #: 200 Training Loss = 1.243102551 Training accuracy= 0.82500
Epoch #: 400 Training Loss = 0.536206126 Training accuracy= 0.98611
Epoch #: 600 Training Loss = 0.280819565 Training accuracy= 1.00000
Epoch #: 800 Training Loss = 0.167816013 Training accuracy= 1.00000
Epoch #: 1000 Training Loss = 0.109477267 Training accuracy= 1.00000
Training completed.....

Final training loss: {} 0.10947726666927338
Train accuracy: 1.0

Cost in test set: 0.030039221048355103
Test accuracy: 1.0
```