ECE421 - Fall 2019
Assignment 1:
Linear and Logistic Regression

Due date: Wednesday, October 9, 2019
Submission: Please hand in a hard copy of the report in the class.
The code should be submitted electronically on Quercus.

Objectives:

In this assignment, you will be investigating the classification performance of linear and logistic regression. In this assignment, you will be implementing the batch Gradient Descent optimization algorithm for a linear regression classifier and logistic regression classifer. After, you will compare the performance of your algorithm against a state-of-the-art optimization technique, ADAM using Stochastic Gradient Descent. For both linear and logistic regression, implementations will be done in Numpy. For comparison with ADAM, you will be allowed to use a Tensorflow implementation. You are encouraged to look up TensorFlow APIs for useful utility functions, at: https://www.tensorflow.org/api_docs/python/.

General Note:

- Full points are given for complete solutions, including justifying the choices or assumptions you made to solve each question. A written report should be included in the final submission.
- Homework assignments are to be solved in groups of two. You are encouraged to discuss the assignment with other students, but you must solve it within your own group. Make sure to be closely involved in all aspects of the assignment. Please indicate the contribution percentage from each group member at the beginning of your report.

1 Linear Regression [18 points]

Linear regression can also be used for classification in which the training targets are either 0 or 1. Once the model is trained, we can determine an input's class label by thresholding the model's prediction. We will consider the following Mean Squared Error (MSE) loss function $\mathcal{L}_{\mathcal{D}}$ and weight

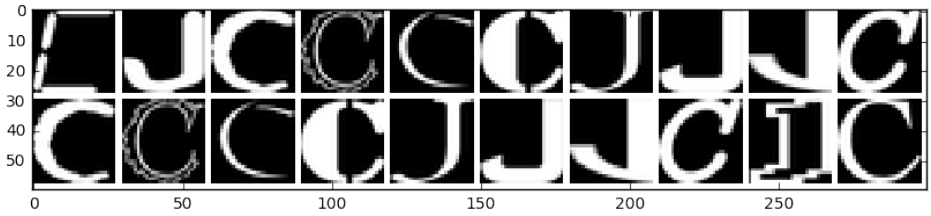
decay loss $\mathcal{L}_W$ for training a linear regression model, in which the goal is to find the best total loss,

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\mathcal{D}} + \mathcal{L}_W \\ =& \sum_{n=1}^{N} \frac{1}{N} \|W^T \mathbf{x}^{(n)} + b - y^{(n)}\|_2^2 + \frac{\lambda}{2} \|W\|_2^2\end{aligned}$$

You will train linear regression model for classification on the *two-class notMNIST* dataset (described in the next section) by implementing the batch gradient descent algorithm. You will be using the validation set to tune the hyper-parameter of the weight decay regularizer.

Two-class notMNIST dataset

We use the following script to generate a smaller dataset that only contains the images from two letter classes: "C"(the positive class) and "J"(the negative class). This smaller subset of the data contains 3500 training images, 100 validation images and 145 test images.




```
with np.load('notMNIST.npz') as data :
    Data, Target = data ['images'], data['labels']
    posClass = 2
    negClass = 9
    dataIndx = (Target==posClass) + (Target==negClass)
    Data = Data[dataIndx]/255.
    Target = Target[dataIndx].reshape(-1, 1)
    Target[Target==posClass] = 1
    Target[Target==negClass] = 0
    np.random.seed(521)
    randIndx = np.arange(len(Data))
    np.random.shuffle(randIndx)
    Data, Target = Data[randIndx], Target[randIndx]
    trainData, trainTarget = Data[:3500], Target[:3500]
    validData, validTarget = Data[3500:3600], Target[3500:3600]
    testData, testTarget = Data[3600:], Target[3600:]
```

1. **Loss Function and Gradient [4 pts]:**
   Implement two *vectorized* Numpy functions to compute the loss and gradient respectively. Both functions should accept 5 arguments - the weight vector, the bias, the data matrix, the

labels, and λ, the regularization parameter. The loss function return a number (indicating total loss). The gradient function should be an analytical expression of the loss function and return the gradient with respect to the weights and the gradient with respect to the biases. Both function headers are below. Include both the analytical expression for the gradient and the Python code snippet in your report.

```
def MSE(W, b, x, y, reg):
#Your implementation here#

def grad_MSE(W, b, x, y, reg):
#Your implementation here#
```

2. **Gradient Descent Implementation [6 pts]:**
Using the gradient computed from Part A, implement the batch Gradient Descent algorithm to classify the two classes in the *notMNIST* dataset. The function should accept 8 arguments - the weight matrix, the bias, the data matrix, the labels, the learning rate, the number of epochs[1], λ and an error tolerance (set to 1×10^{-7}). The error tolerance will be used to compute the difference between the old and updated weights per iteration. The function should return the optimized weight and bias matrices. The function header is below.

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
#Your implementation here#
```

Your function must return the trained weights and biases. You may also wish to print and store the training, validation and test losses/accuracies in this function for plotting. (In this case, you can add two more inputs for validation and test data to `grad_descent()`).

3. **Tuning the Learning Rate[3 pts]:**
Test your implementation of Gradient Descent with 5000 epochs and $\lambda = 0$. Investigate the impact of learning rate, $\alpha = 0.005, 0.001, 0.0001$ on the performance of your classifier. Plot the training, validation and test losses.

4. **Generalization [3 pts]:**
Investigate impact by modifying the regularization parameter, $\lambda = \{0.001, 0.1, 0.5\}$. Plot the training, validation and test loss for $\alpha = 0.005$ and report the final training, validation and test accuracy of your classifier.

5. **Comparing Batch GD with normal equation [2 pts]:**
For linear regression, you can find the optimum weights using the closed form equation for the derivative of the means square error (normal equation). For zero weight decay, Write a Numpy script to find the optimal linear regression weights on the *two-class notMNIST* dataset using the "normal equation" of the least squares formula. Compare in terms of final training MSE, accuracy and computation time between Batch GD and normal equation.

[1]Epoch is defined as a complete pass of the training data. By definition, batch gradient descent operates on the entire training dataset

2 Logistic Regression [10 points]

2.1 Binary cross-entropy loss

The MSE loss function works well for typical regression tasks in which the model outputs are real values. Despite its simplicity, MSE can be overly sensitive to mislabelled training examples and to outliers. A more suitable loss function for the classification task is the cross-entropy loss, which compares the log-odds of the data belonging to either of the classes. Because we only care about the probability of a data point belonging to one class, the real-valued linear prediction is first fed into a sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ that "squashes" the real-valued output to fall between zero and one. The model output is therefore $\hat{y}(\mathbf{x}) = \sigma(W^T\mathbf{x} + b)$. The cross-entropy loss is defined as:

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\mathcal{D}} + \mathcal{L}_W \\ =& \sum_{n=1}^{N} \frac{1}{N}\Big[- y^{(n)} \log \hat{y}(\mathbf{x}^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}(\mathbf{x}^{(n)}))\Big] + \frac{\lambda}{2}\|W\|_2^2\end{aligned}$$

The sigmoid function is often called the logistic function and hence a linear model with the cross-entropy loss is named "logistic regression".

1. **Loss Function and Gradient [4 pts]:**
   Implement two *vectorized* Numpy functions to compute the Binary Cross Entropy Error and its gradient respectively. Similar to Part 1.1, both functions should accept 5 arguments - the weight vector, the bias, the data matrix, the labels, and the regularization parameter. They should return the loss and gradients with respect to weights and biases respectively. Both function headers are below. Include the analytical expressions in your report as well as a snippet of your Python code.

```
def crossEntropyLoss(W, b, x, y, reg):
#Your implementation

def gradCE(W, b, x, y, reg):
#Your implementation
```

2. **Learning [4 pts]:**
   Modify the function from Part 1.2 to include a flag, specifying the type of loss/gradient to use in the classifier. Modify your function to update weights and biases using the Binary Cross Entropy loss and report on the performance of the Logistic Regression model by setting $\lambda = 0.1$ and 5000 epochs. Plot the loss and accuracy curves for training, validation, and test data set.

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, lossType="MSE"):
```

3. **Comparison to Linear Regression [2 pts]:**
   For zero weight decay, learning rate of 0.005 and 5000 epochs , plot the training cross entropy

loss and MSE loss for logistic regression and linear regression respectively. Comment on the effect of cross-entropy loss convergence behaviour.

3 Batch Gradient Descent vs. SGD and Adam [25 points]

3.1 SGD

In the exercises above, you implemented the Batch Gradient Descent Algorithm. For large datasets however, obtaining the gradient for *all* of the training data may be infeasible. Stochastic Gradient Descent, or Mini-batch gradient descent is aimed at solving this problem. You will be implementing the SGD algorithm and optimizing the training process using the Adaptive Moment Estimation technique (Adam), using Tensorflow.

1. **Building the Computational Graph [5 pts]:**

   Build a function, *buildGraph()* that accepts one argument, the loss type (either MSE or CE) and initializes the Tensorflow computational graph. To do so, you must initialize the following:

   (a) The weight and bias tensors (for the weight tensors, use the *tf.truncated_normal* command and set the standard deviation to 0.5.

   (b) Tensors to hold the "variables"; use the *tf.placeholder* command (e.g. tensors for the data, labels and λ.)

   (c) The loss tensor. Depending on the parameter passed to the *buildGraph()* function, you will need to either create the MSE or CE loss function with the regularization parameter. You may wish to investigate the Tensorflow API Documentation regarding losses and regularization on their website.

   (d) The optimizer. Be sure to set it to minimize the total loss. Set α to 0.001.

   The function should return the Tensorflow objects for weight, bias, predicted labels, real labels, the loss, and the optimizer. The function header is below.

```
def buildGraph(loss="MSE"):
   #Initialize weight and bias tensors
   tf.set_random_seed(421)
   if loss == "MSE":
   # Your implementation
   elif loss == "CE":
   #Your implementation here
```

2. **Implementing Stochastic Gradient Descent [5 pts.]** Implement the SGD algorithm for a minibatch size of 500 optimizing over 700 epochs [2], minimizing the MSE (you will repeat

[2] An epoch refers to a complete pass of the training data. SGD makes weight updates based on a *sample* of the training data.

this for the CE later). Calculate the total number of batches required by dividing the number of training instances by the minibatch size. After each epoch you will need to reshuffle the training data and start sampling from the beginning again. Initially, set $\lambda = 0$ and continue to use the same α value (i.e. 0.001). After each epoch, store the training, validation and test losses and accuracies. Use these to plot the loss and accuracy curves.

3. **Batch Size Investigation [2 pts.]** Study the effects of batch size on behaviour of the SGD algorithm optimized using Adam by optimizing the model using batch sizes of $B = \{100, 700, 1750\}$. For each batch size, plot training/validation/test MSE loss in one plot and training/validation/test accuracy in another plot. (you need to have a total of 6 plots for this section). What is the impact of batch size on the final classification accuracy for each of the 3 cases? What is the rationale for this?

4. **Hyperparameter Investigation [4 pts.]** Experiment with the following Adam hyperparameters and for each, report on the final training, validation and test accuracies:

   (a) $\beta 1 = \{0.95, 0.99\}$

   (b) $\beta 2 = \{0.99, 0.9999\}$

   (c) $\epsilon = \{1e-09, 1e-4\}$

   For this part, use a minibatch size $B = 500$ and a learning rate of $\alpha = 0.001$ and optimize over 700 epochs. For each of the three hyperparameters listed above, keep the other two as the default Tensorflow initialization. Note that in order to set $\beta 1$, $\beta 2$, and ϵ, you may wish to add these parameters to `build_graph()` inputs.
   For each, what is the hyperparameter impact on the final training, validation and test accuracy? Why is this happening?

5. **Cross Entropy Loss Investigation [6 pts.]** Repeat parts 3.1.2 to 3.1.4 by minimizing the binary cross entropy loss. How do the two models compare against each other in terms model performance (i.e. final classification accuracy)?

6. **Comparison against Batch GD [3 pts.]** Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier. Additionally, comment on the plots of the losses and accuracies of the SGD vs. batch gradient descent implementation. What do you notice about the curves? Why is this happening?