ECE421 - Winter 2021
Assignment 1:
Logistic Regression

Due date: February 9, 2021

Submission: Submit both your report (a single PDF file) and all codes on Quercus.

Objectives:

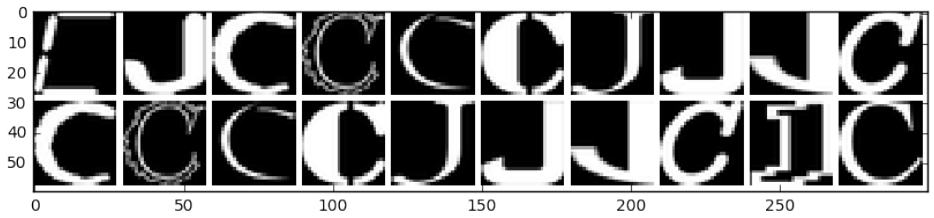
In this assignment, you will first implement a simple logistic regression classifier using Numpy and train your model by applying (Stochastic) Gradient Descent algorithm. Next, you will implement the same model, this time in TensorFlow and use Stochastic Gradient Descent and ADAM to train your model.
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.
- Programming assignments are to be solved and submitted **individually**. You are encouraged to discuss the assignment with other students, but you must solve it on your own.
- Please ask all questions related to this assignment on Piazza, using the tag `assignment1`.

Two-class notMNIST dataset

The notMNIST dataset is a image recognition dataset of font glyphs for the letters A through J useful with simple neural networks. It is quite similar to the classic MNIST dataset of handwritten digits 0 through 9. 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 Logistic Regression with Numpy[20 points]

Logistic regression is one the most widely used linear classification models in machine learning. In logistic regression, we model the probability of a sample $\mathbf{x}$ belonging to the positive class as

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

where $z = \mathbf{w}^\top \mathbf{x} + b$, also called *logit*, is basically the linear transformation of input vector $\mathbf{x}$ using weight vector $\mathbf{w}$ and bias scalar b, and $\sigma(z) = 1/(1 + \exp(-z))$ is the sigmoid or logistic function. The sigmoid function "squashes" the real-valued logits to fall between zero and one.

The cross-entropy loss $\mathcal{L}_{\text{CE}}$ and the regularization term $\mathcal{L}_{\text{w}}$ will form the total loss function as:

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{w}} \\ =& \frac{1}{N} \sum_{n=1}^{N} \left[- y^{(n)} \log \hat{y}\left(\mathbf{x}^{(n)}\right) - \left(1 - y^{(n)}\right) \log\left(1 - \hat{y}\left(\mathbf{x}^{(n)}\right)\right) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2\end{aligned}$$

Note that $y^{(n)} \in \{0, 1\}$ is the class label for the n-th training image and λ is the regularization parameter.

1. **Loss Function and Gradient [8 pts]:**
   Implement two *vectorized* Numpy functions (i.e. avoid using for loops by employing matrix products and broadcasting) to compute the loss function and its gradient. The `grad_loss` function should compute and return an analytical expression of the gradient of the loss with respect to both the weights and bias. Both function headers are below. Include the analytical expressions in your report as well as a snippet of your Python code.

```
def loss(w, b, x, y, reg):
#Your implementation

def grad_loss(w, b, x, y, reg):
#Your implementation
```

2. **Gradient Descent Implementation [6 pts]:**
   Using the gradient computed from part 1, implement the batch Gradient Descent algorithm to classify the two classes in the *notMNIST* dataset. The function should accept 8 arguments - the weight vector, 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 training should stop if the total number of epochs is reached, or the difference between the old and updated weights are smaller than the error tolerance. The function should return the optimized weight vector and bias. The function header is below.

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

   You may also wish to print and/or 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 and validation loss (on one figure) vs. number of passed epochs for each value of α. Repeat this for training and validation accuracy. You should submit a total of 6 figures in your report for this part. Also, explain how you choose the best learning rate, and what accuracy you report for the selected learning rate.

4. **Generalization [3 pts]:**
   Investigate the impact of regularization by modifying the regularization parameter, $\lambda = \{0.001, 0.1, 0.5\}$ for $\alpha = 0.005$. Plot the training/validation loss/accuracy vs. epochs figure, similar to the previous part. Also, explain how you choose the best parameter, and what accuracy you report for the selected model.

[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 in TensorFlow [20 points]

In the exercises above, you implemented the Batch Gradient Descent Algorithm. For large datasets however, obtaining the loss gradient using *all* the training data at each iteration 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]:**

   Define a function, `buildGraph()` that initializes the TensorFlow computational graph. To do so, you must initialize the following:

   (a) The weight and bias tensors: for the weight tensors, use `tf.truncated_normal` and set the standard deviation to 0.5. Initial the bias variable with zero.

   (b) Placeholders for data, labels and λ: use `tf.placeholder`.

   (c) The loss tensor: Calculates the CE loss function with the regularization term. You may wish to investigate the TensorFlow API Documentation regarding losses and regularization on their website.

   (d) The optimizer: use `tf.train.AdamOptimizer` to minimize the total loss. Set the learning rate α to 0.001.

   The function should return the TensorFlow objects for weight, bias, predicted labels, real labels, the loss, and the optimizer.

2. **Implementing Stochastic Gradient Descent [5 pts.]** For the training loop, implement the SGD algorithm using a minibatch size of 500 optimizing over 700 epochs [2]. 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 and validation losses and accuracies. Use these to plot the loss and accuracy curves.

3. **Batch Size Investigation [4 pts.]** Study the effects of batch size on behaviour of Adam by optimizing the model using batch sizes of $B = \{100, 700, 1750\}$. Also, set $\lambda = 0$ and continue to use $\alpha = 0.001$. For each batch size, plot training/validation loss in one plot and training/validation 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? Can you justify this observation?

4. **Hyperparameter Investigation [4 pts.]** Experiment with the following Adam hyperparameters and for each, report on the final training, validation and test accuracies. Explain which value you pick for hyperparameters in each part, and what accuracy you report.

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

(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 $B = 500$, a learning rate of $\alpha = 0.001$ with regularization, 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.

5. **Comparison against Batch GD [2 pts.]** Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier by comparing plots of the losses and accuracies of the Adam vs. batch gradient descent.