ECE421 - Winter 2022
Assignment 2:
Neural Networks

Due date: Monday, Feb 28, 2022

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

Objectives:

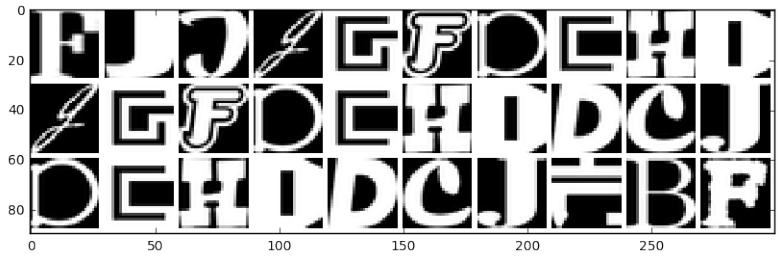
The purpose of this assignment is to investigate the classification performance of neural networks. You will be implementing a neural network model using Numpy, followed by an implementation in Tensorflow. You are encouraged to look up TensorFlow APIs for useful utility functions, at: `https://www.tensorflow.org/api_docs/python/`.

Guidelines:

- 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. Do not dump all your codes and outputs in your report. Keep it short, readable, and well-organized.
- 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 `pa2`.

notMNIST Dataset

The dataset that we will use in this assignment is a permuted version of notMNIST[1], which contains 28-by-28 images of 10 letters (A to J) in different fonts. This dataset has 18720 instances, which can be divided into different sets for training, validation and testing. The provided file is in **.npz** format which is for Python. You can load this file as follows.




```
def load_data():
    with np.load("notMNIST.npz") as data:
        data, targets = data["images"], data["labels"]

        np.random.seed(521)
        rand_idx = np.arange(len(data))
        np.random.shuffle(rand_idx)

        data = data[rand_idx] / 255.0
        targets = targets[rand_idx].astype(int)

        train_data, train_target = data[:10000], targets[:10000]
        valid_data, valid_target = data[10000:16000], targets[10000:16000]
        test_data, test_target = data[16000:], targets[16000:]
    return train_data, valid_data, test_data, train_target, valid_target, test_target
```

Since you will be investigating multi-class classification, you will need to convert the data into a one-hot encoding format. The code snippet below will help you with that.

```
def convert_onehot(train_target, valid_target, test_target):
    new_train = np.zeros((train_target.shape[0], 10))
    new_valid = np.zeros((valid_target.shape[0], 10))
    new_test = np.zeros((test_target.shape[0], 10))
    for item in range(0, train_target.shape[0]):
        new_train[item][train_target[item]] = 1
    for item in range(0, valid_target.shape[0]):
        new_valid[item][valid_target[item]] = 1
    for item in range(0, test_target.shape[0]):
        new_test[item][test_target[item]] = 1
    return new_train, new_valid, new_test
```

[1] http://yaroslavvb.blogspot.ca/2011/09/notmnist-dataset.html

1 Neural Networks using Numpy [20 pts.]

In this part, you will be tasked with implementing and training a neural network to classify the letters using Numpy and Gradient Descent with Momentum. The network you will be implementing has the following structure:

- 3 layers - 1 input, 1 hidden with ReLU activation and 1 output with Softmax:
  - Input Layer: $\mathbf{x}$ (F units, here: $F = 784$)
  - Hidden Layer: $\mathbf{h} = \text{ReLU}(\mathbf{W}_h\,\mathbf{x} + \mathbf{b}_h)$ (H units)
  - Output Layer: $\mathbf{p} = \text{softmax}(\mathbf{o})$, where $\mathbf{o} = \mathbf{W}_o\,\mathbf{h} + \mathbf{b}_o$ (K units, here $K = 10$)
- Cross Entropy Loss: $\mathcal{L} = -\sum_{k=1}^{K} y_k \log(p_k)$,
  where $\mathbf{y} = [y_1, y_2, \cdots, y_K]^\top$ is the one-hot coded vector of the label.

During the training process, it may be beneficial to save weights to a file during the training process - the function `numpy.savetext` may be useful. As an estimate of the running time, training the Numpy implementation should not take longer than an hour (tested on an Intel i7 3770K at 3.40 GHz and 16 GB of RAM). For this part only, **Tensorflow implementations will not be accepted**.

1.1 Helper Functions [6 pt.]

To implement the neural network described earlier, you will need to implement the following *vectorized* (i.e. no for loops, they must rely on matrix/vector operations) helper functions. Include the snippets of your Python code in the report.

1. `relu()`: This function will accept one argument and return Numpy array with the ReLU activation and the equation is given below. [0.5 pt]

$$\text{ReLU}(x) = \max(x, 0)$$

2. `softmax()`: This function will accept one argument and return a Numpy array with the softmax activations of each of the inputs and the equation is shown below. [1.5 pt]

$$\sigma(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}, \quad j = 1, \cdots, K \text{ for K classes.}$$

   Important Hint: In order to prevent overflow while computing exponentials, you should first subtract the maximum value of $\mathbf{z}$ from all its elements.

3. `compute()`: This function will accept 3 arguments: a weight matrix, an input vector, and a bias vector and return the product between the weights and input, plus the biases (i.e. a prediction for a given layer). [1 pt]

4. `average_ce()`: This function will accept two arguments, the targets (e.g. labels) and predictions - both are matrices of the same size. It will return a number, average the cross entropy loss for the dataset (i.e. training, validation, or test). For K classes, the formula is shown below. [1 pt]

$$\text{Average CE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{(n)} \log\left(p_k^{(n)}\right)$$

   Here, $y_k^{(n)}$ is the true one-hot label for sample n, p_k is the predicted class probability (i.e. softmax output for the k^{th} class) of sample n, and N is the number of examples.

5. `grad_ce()`: This function will accept two arguments, the targets (i.e. labels $\mathbf{y}$) and the input to the softmax function (i.e. $\mathbf{o}$). It will return the gradient of the cross entropy loss with respect to the inputs to the softmax function: $\partial\mathcal{L}/\partial\mathbf{o}$. **Show the analytical expression in your report.** [2 pt.]

1.2 Backpropagation Derivation [8 pts.]

To train the neural network, you will need to implement the backpropagation algorithm. For the neural network architecture outlined in the assignment description, derive the following analytical expressions and include them in your report:

1. $\frac{\partial\mathcal{L}}{\partial\mathbf{W}_o}$, the gradient of the loss with respect to the output layer weights. [2 pt.]
   - Shape: $(H \times 10)$, with H units
2. $\frac{\partial\mathcal{L}}{\partial\mathbf{b}_o}$, the gradient of the loss with respect to the output layer biases. [2 pt.]
   - Shape: (1×10)
3. $\frac{\partial\mathcal{L}}{\partial\mathbf{W}_h}$, the gradient of the loss with respect to the hidden layer weights. [2 pt.]
   - Shape: $(F \times H)$, with F features, H units
4. $\frac{\partial\mathcal{L}}{\partial\mathbf{b}_h}$, the gradient of the loss with respect to the hidden layer biases. [2 pt.]
   - Shape: $(1 \times H)$, with H units.

Hints: The labels $\mathbf{y}$ have been one hot encoded. You will also need the derivative of the ReLU() function in order to backpropagate the gradient through the activation.

1.3 Learning [6 pts.]

Construct the neural network and train it for 200 epochs with a hidden unit size of $H = 1000$. First, initialize your weight matrices following the Xaiver initialization scheme (zero-mean Gaussians with variance $\frac{2}{\text{units in}+\text{units out}}$) and your bias vectors to zero, each with the shapes as outlined in section 1.2. Using these matrices, compute a forward pass of the training data and then, using the gradients derived in section 1.2, implement the backpropagation algorithm to update all of the network's weights and biases. The optimization technique to be used for backpropagation will be Gradient Descent with momentum and the equation is shown below.

$$\boldsymbol{\nu}_{\text{new}} \leftarrow \gamma \boldsymbol{\nu}_{\text{old}} + \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{\nu}_{\text{new}}$$

For the $\boldsymbol{\nu}$ matrices, initialize them to the same size as the hidden and output layer weight matrix sizes, with a very small value (e.g. 10^{-5}). Additionally, initialize your γ values to values slightly less than 1 (e.g. 0.9 or 0.99) and set $\alpha = 0.1$ for the average loss. (Note that you need to scale the learning rate if you are using the total loss).

Plot the training and validation loss in one figure, and the training and validation accuracy curves in a second figure and include them in your report. For the accuracy metric, the `np.argmax()` function will be helpful.

2 Neural Networks in Tensorflow [optional]

In this part, you will be implementing a Convolutional Neural Network, the most popular technique for image recognition, using Tensorflow. It is recommended that you train the neural network using a GPU (although this is not required.) The neural network architecture that you will be implementing is as follows:

1. Input Layer
2. A 3×3 convolutional layer, with 32 filters, using vertical and horizontal strides of 1.
3. ReLU activation
4. A batch normalization layer
5. A 2×2 max pooling layer
6. Flatten layer
7. Fully connected layer (with 784 output units, i.e. corresponding to each pixel)
8. ReLU activation
9. Fully connected layer (with 10 output units, i.e. corresponding to each class)
10. Softmax output
11. Cross Entropy loss

2.1 Model Implementation

Implement the described neural network architecture described at the beginning of this section. You will find the `tf.nn.conv2d`, `tf.nn.relu`, `tf.nn.batch_normalization` and `tf.nn.max_pool` utility functions useful. For the convolutional layer, initialize each filter with the Xaiver scheme. Initialize your weight and biases for the other layers like you did in Part 1 (but with Tensorflow tensors). For the `padding` parameter, set it to the `SAME` method. For the batch normalization layer, the `tf.nn.moments` function will be useful for obtaining the mean and variance.
You are allowed to use the built-in cross-entropy loss function. Include your Python code snippets in your report.

2.2 Model Training

Train your implemented model using SGD for a batch size of 32, for 50 epochs and the Adam optimizer for learning rate of $\alpha = 1 \times 10^{-4}$, making sure to shuffle your training data after each epoch. Your objective function will be to minimize the cross entropy loss. Plot the training and validation loss/accuracy curves and include them in your report.

2.3 Dropout

A popular method to control overfitting in very deep neural networks is to apply dropout to certain layers in the model. Add a dropout layer after step 7, described in section 2 and test it with keeping probabilities (1− dropout rate) $p = [0.9, 0.75, 0.5]$ while holding all other parameters constant as in section 2.2 with no regularization and plot the training and validation accuracy/loss for 50 epochs and include them in your report.