CMSC 636 Neural Nets and Deep Learning
Spring 2022, Instructor: Dr. Milos Manic, *http://www.people.vcu.edu/~mmanic*
Homework 3

Student certification:

Team member 1:
Print Name: Samah Ahmed *Date: 14 April 2022*
I have contributed by doing the following: 3.1, 3.2
Signed: Samah

Team member 2:
Print Name: Md Touhiduzzaman *Date: 15 April 2022*
I have contributed by doing the following: 3.3
Signed: Touhid

Team member 2:
Print Name: Maher Al Islam *Date: 15 April 2022*
I have contributed by doing the following: 3.4
Signed: Maher

3.0 Intro to TensorFlow (0 pts)

Complete the previously posted *Intro to TensorFlow* exercise.

3.1 TensorFlow (3 pts)

Please answer the following questions:

- What are Variables in TensorFlow?
- What are Tensors in TensorFlow?
- What is a gradient tape?

Report: Answers to questions.

Answer to the question 3.1:

TensorFlow Variable:

A TensorFlow **variable** is the recommended way to represent a shared, persistent state your program manipulates. This guide covers how to create, update, and manage instances of tf.Variable in TensorFlow.

Variables are created and tracked via the tf.Variable class. A tf.Variable represents a tensor whose value can be changed by running ops on it. Specific ops allow you to read and modify the values of this tensor. Higher level libraries like tf.keras use tf.Variable to store model parameters.

TensorFlow variables can be used to represent **model trainable parameters** like weights and biases of a neural network. For example:

```
a = tf.Variable([2.0, 3.0])
a.assign([3, 4])
a
```

```
<tf.Variable 'Variable:0' shape=(2,) dtype=float32, numpy=array([3., 4.], dtype=float32)>
```

Tensors in TensorFlow:

Tensors are multi-dimensional arrays with a uniform type (called a dtype). It can be used to represent **the data being fed into the model** and also the intermediate representations of the data as it passes through the model like output of activation functions.

For example:

```
import tensorflow as tf
rank_2_tensor = tf.constant([[1, 2],
                             [3, 4],
                             [5, 6]], dtype=tf.float16)
rank_2_tensor
```

```
<tf.Tensor: shape=(3, 2), dtype=float16, numpy=
array([[1., 2.],
       [3., 4.],
       [5., 6.]], dtype=float16)>
```

Gradient Tape:

TensorFlow provides the tf.GradientTape API for automatic differentiation; that is, computing the gradient of a computation with respect to some inputs, usually tf.Variables. TensorFlow "records" relevant operations executed inside the context of a tf.GradientTape onto a "tape". TensorFlow then uses that tape to compute the gradients of a "recorded" computation using reverse mode differentiation.

Example:

```
x = tf.Variable([2.0, 3.0])
with tf.GradientTape() as tape:
  y = x**2

dy_dx = tape.gradient(y, x) # dy = 2*x * dx => dy/dx = 2*x
dy_dx.numpy()
```

```
array([4., 6.], dtype=float32)
```

3.2 Image classification using Multilayer Perceptron (6 pts)

Train a Multilayer Perceptron Neural Network to classify numbers on the MNIST dataset http://yann.lecun.com/exdb/mnist/. In the attached script “mlp_mnist.html”, you will find a starter code that downloads the MNIST dataset and loads it into this Python script.

Please complete this script (where indicated) to train a multilayer perceptron with one hidden layer of 100 units, using ReLU activation function.

The classification accuracy on both training and testing dataset should be above 90%.

Note: do NOT use the tf.keras library.

Deliverables:

- Jupyter notebook including the generated output (following *Deliverables* directions above).
- Report on the accuracy on training and testing datasets (as part of the single pdf report, following *Deliverables* directions above).

Answer to the question no. 3.2:

Report on the accuracy on training and testing datasets:

- Train accuracy: 99.595%
- Test accuracy: 97.4%

The provided script is updated & attached as “mnist_mlp.ipynb” file. A screenshot of the results of the last epochs is:

```
8000 , train: 99.2  | test: 97.8  | loss: 0.05240652631036937
8200 , train: 99.435  | test: 98.0  | loss: 0.047275986969470975
8400 , train: 99.405  | test: 98.0  | loss: 0.047518889531493184
8600 , train: 99.265  | test: 96.9  | loss: 0.05266458410769701
8800 , train: 99.49  | test: 97.4  | loss: 0.04634139774367213
9000 , train: 99.485  | test: 98.1  | loss: 0.045844548922032116
9200 , train: 99.38  | test: 98.2  | loss: 0.04827182383276522
9400 , train: 99.63  | test: 97.3  | loss: 0.04236996789462864
9600 , train: 99.5  | test: 98.5  | loss: 0.0448344667442143
9800 , train: 99.405  | test: 97.3  | loss: 0.04710992259904742
10000 , train: 99.595  | test: 97.4  | loss: 0.04208581139333546
```

3.3 Image classification using Convolutional Neural Networks (6 pts)

Train a Convolutional Neural Network (CNN) on the MNIST dataset. In the script "convnet_mnist.html" you will find a starter code that downloads the MNIST dataset and loads it into this Python script. For this problem, you should do the following:

1. Update the script to setup the following architecture:
   - First convolutional layer
     - 32 filters, each one of size 5x5
     - ReLU activation function (Rectifier Linear Unit)
     - Max pooling layer of size 2x2, and stride of 2 in both x and y.
   - Second convolutional layer
     - 64 filters, each one of size 5x5
     - ReLU activation function
     - Max pooling layer of size 2x2, and stride of 2 in both x and y.
   - Reshape layer
     - Is responsible for transforming of 2D filtered maps to 1D vector, which is the input for the fully connected layer,
   - Fully connected layer
     - Linear layer with 256 units (fully connected layer),
     - ReLU activation function.
   - Final layer
     - Linear layer that maps the 256 units from the previous layer to the 10 output units (0 to 9).

2. Relative to tensors: what is the size in each dimension (shape) for the inputs and outputs of each layer?
   - for example, in case of a single gray image X.shape = (32, 32, 1)

3. Train the CNN with the MNIST dataset
   - Report accuracy and loss of training and test datasets.
   - Hint: Before going into a larger number of epochs, test the model on a single epoch. Correct errors, if any.

4. Compare the performance of the multilayer perceptron with the performance of the convolutional neural network.
   - Accuracy, training time (you can simply use your own watch).

Note: do NOT use the tf.keras library.

Deliverables:

- Jupyter notebook including the generated output (following *Deliverables* directions above).
- Report answers to questions 2, 3, and 4. Comment and discuss.

Answer to the question no. 3.3:

1. The script is updated as per the requirements and it is included with this submission as "mnist_convnet.ipynb" file.
2. We have 5 layers in total, each with the following specification os input & output dimensions (shapes):
   a. First convolutional layer
      i. Input: 1
      ii. Output: 32
   b. Second convolutional layer
      i. Input: 32
      ii. Output: 64
   c. Reshape layer: [-1, 4*4*64]
   d. Fully connected layer
      i. Input: 4*4*64
      ii. Output: 256

    e. Final layer
        i. Input: 256
        ii. Output: 10

3. The implemented CNN network is trained with the MNIST dataset in local machine & a GPU enabled VCU server.
   In a local machine without any use of GPU, we have got 99.845% training and 99.4% testing accuracy with 0.022966726017184556 as the final loss.
   In the Maple server (a GPU enabled VCU shared server), we trained the network with 10000 epochs and have got 99.97% training and 99.0% testing accuracy with 0.02 loss at the final epoch. A screenshot of that run given below:

```
1800 , train: 99.285  | test: 99.0  | loss: 0.04079201966058463
2000 , train: 99.54  | test: 99.1  | loss: 0.03087748760357499
2200 , train: 99.45  | test: 99.8  | loss: 0.034439907711930576
2400 , train: 99.405  | test: 99.5  | loss: 0.037064483468420804
2600 , train: 99.665  | test: 99.7  | loss: 0.027214896492660047
2800 , train: 99.575  | test: 98.6  | loss: 0.030846516047604383
3000 , train: 99.585  | test: 98.9  | loss: 0.03331029714550823
3200 , train: 99.78  | test: 98.8  | loss: 0.022624842189252377
3400 , train: 99.76  | test: 97.8  | loss: 0.025753682190552355
3600 , train: 99.655  | test: 99.4  | loss: 0.02958923962432891
3800 , train: 99.825  | test: 99.4  | loss: 0.02257872040849179
4000 , train: 99.855  | test: 99.0  | loss: 0.024868159983307124
4200 , train: 99.805  | test: 99.2  | loss: 0.025086730965413154
4400 , train: 99.935  | test: 99.2  | loss: 0.019280381104908883
4600 , train: 99.84  | test: 99.4  | loss: 0.024004313009791077
4800 , train: 99.755  | test: 99.4  | loss: 0.027942668991163374
5000 , train: 99.96  | test: 99.3  | loss: 0.018502453360706567
5200 , train: 99.905  | test: 98.9  | loss: 0.019707751083187758
5400 , train: 99.86  | test: 99.0  | loss: 0.02329894997179508
5600 , train: 99.945  | test: 98.9  | loss: 0.019035240225493907
5800 , train: 99.93  | test: 99.3  | loss: 0.021875131754204632
6000 , train: 99.945  | test: 99.2  | loss: 0.01925814157817513
6200 , train: 99.98  | test: 98.6  | loss: 0.016211010930128396
6400 , train: 99.915  | test: 99.8  | loss: 0.021377588184550406
6600 , train: 99.94  | test: 98.9  | loss: 0.020923264040611685
6800 , train: 99.93  | test: 99.3  | loss: 0.02021235373802483
7000 , train: 99.91  | test: 99.2  | loss: 0.019244929146952926
7200 , train: 99.935  | test: 99.3  | loss: 0.0207162307202816
7400 , train: 99.97  | test: 98.6  | loss: 0.018478872552514077
7600 , train: 99.925  | test: 99.5  | loss: 0.019884685734286905
7800 , train: 99.895  | test: 99.6  | loss: 0.021149501581676305
8000 , train: 99.98  | test: 99.1  | loss: 0.01571683948393911
8200 , train: 99.91  | test: 99.1  | loss: 0.022707769703119992
8400 , train: 99.94  | test: 99.3  | loss: 0.02006652444601059
8600 , train: 99.97  | test: 99.2  | loss: 0.01638954727444798
8800 , train: 99.955  | test: 99.5  | loss: 0.01605737549252808
9000 , train: 99.96  | test: 99.4  | loss: 0.01758362414781004
9200 , train: 99.94  | test: 98.8  | loss: 0.01849637017119676
9400 , train: 99.975  | test: 99.6  | loss: 0.016228764387778937
9600 , train: 99.995  | test: 99.1  | loss: 0.016424807249568404
9800 , train: 99.97  | test: 99.0  | loss: 0.020536542739719152
Number: 5
Prediction by the model: 5

real	4m49.798s
user	72m55.873s
sys	3m14.870s
(myenv) [touhiduzzamm@maple ~]$
```

4. MLP vs CNN in terms of Accuracy & Training Time:
   The accuracy is found to be better with CNN than in MLP with the same dataset and same number of epochs. Even, CNN yields higher accuracy with half number of (5000) epochs than the MLP (10000 epochs) experiment.
   But, CNN requires more time to get trained than the MLP implementation. More than 6 minutes and 32 seconds of runtime was required for the CNN algorithms 5000 epochs to complete, which is a lot higher than the 10000 epochs of MLP requiring 1 minute and 27 seconds in the local machine. We achieved slightly better runtime for CNN by running it in the GPU enabled Maple server, which is 4 minute and 49.79 seconds.

3.4 Extra credit (3 pts): Modify the script "mlp_mnist.ipy" for training of a multilayer perceptron network with two hidden layers, each with 100 units, using ReLU activation function.

- Does the training error improve?
- What about the testing error?
- Try using sigmoid activation function. Discuss the results.

Deliverables:

- Jupyter notebook including the generated output (following *Deliverables* directions above).
- Report answers to questions. Comment and discuss.

Deliverables/Report:

- Following instructions above, submit your file through Canvas.
- This assignment is worth 15 points.

<u>Answer to the question no. 3.4:</u>

The script is updated and attached with this submission as "mnist_mlp_34.ipynb".
We have noticed that the training accuracy of this updated version slightly improves to 99.795% than the previous 99.595%. Also, the testing accuracy improves a little to 97.5% from 97.4%.

```
8000 , train: 99.7  | test: 97.3  | loss: 0.031000382965430617
8200 , train: 99.55  | test: 97.3  | loss: 0.03744897070806474
8400 , train: 99.54  | test: 97.5  | loss: 0.037157550393603744
8600 , train: 99.765  | test: 97.4  | loss: 0.028271524873562156
8800 , train: 99.635  | test: 98.0  | loss: 0.03369258895050734
9000 , train: 99.56  | test: 97.3  | loss: 0.035886577139608564
9200 , train: 99.765  | test: 97.1  | loss: 0.02756453896407038
9400 , train: 99.72  | test: 98.0  | loss: 0.029209074317477643
9600 , train: 99.6  | test: 97.9  | loss: 0.037449182434938846
9800 , train: 99.83  | test: 96.9  | loss: 0.026858634077943862
10000 , train: 99.795  | test: 97.5  | loss: 0.02831031844485551
```

But using the sigmoid function does not change the accuracy that much than the ReLU function. Therefore, the ReLU activation function of the two hidden layers is a much better fit and the sigmoid function is an underfit in comparison to the MLP-2-hidden-layer.

~~~~~~ [The End] ~~~~~~
~~~~~~