Assignment 1
Introduction to parallel deep neural networks

Due Date: 10/12/2018 23:30

Questions:
- Questions regarding this assignment should only be sent to Qasem
- Problems with connecting to servers should only be sent to Or
- Postponements can only be authorized by the TA in charge Ido.
Part 1

**Brief Background:**

Neural networks are made up of building blocks known as Sigmoid Neurons. These are named so because their output follows Sigmoid Function.

A neural network is made up by stacking layers of neurons, and is defined by the weights of connections and biases of neurons. Activations are a result dependent on a certain input.
Naming and Indexing Convention:

Layers

Input layer is the $0_{th}$ layer, and output layer is the $L_{th}$ layer. Number of layers: $N_L = L + 1$.

In the above example: sizes = [2, 3, 1]

Weights

Weights in this neural network implementation are a list of matrices (numpy.ndarrays). weights[l] is a matrix of weights entering the lth layer of the network (Denoted as $w_l$).
An element of this matrix is denoted as $w_{lj}$k. It is a part of jth row, which is a collection of all weights entering jth neuron, from all neurons (0 to k) of (l-1)th layer.
No weights enter the input layer, hence weights[0] is redundant, and further it follows as weights[1] being the collection of weights entering layer 1 and so on.

In the above example: weights = 

In the above example: 
\[
\text{biases} = \begin{array}{c}
[0], [0] \\
[1], \\
[2],
\end{array}
\]

'Z's

For input vector x to a layer l, z is defined as: 
\[z_l = w_l \cdot x + b_l\]

Input layer provides x vector as input to layer 1, and itself has no input, weight or bias, hence zs[0] is redundant.

Dimensions of zs will be same as biases.

Activations

Activations of \(l\)th layer are outputs from neurons of \((l-1)\)th which serve as input to \((l+1)\)th layer. The dimensions of biases, zs and activations are similar.

Input layer provides x vector as input to layer 1, hence activations[0] can be related to x - the input training example.

Implementation:

First, you have to implement the functions in utils.py:

\[
\text{def sigmoid(x):}
\]

Calculates the above sigmoid function with \(\beta=1\). This function outputs \(f(x)\).

\[
\text{def sigmoid_prime(x):}
\]

Calculates the derivative function of sigmoid \(f'\) with input x and is given by \(f(x) \cdot (1 - f(x))\).

\[
\text{def random_weights(sizes):}
\]

Calculates and returns a list of random xavier initialized np arrays of shapes \((sizes[i],sizes[i+1])\) for \(i\) from 0 to the length of \(sizes -2\).

\[
\text{def zeros_weights(sizes):}
\]

Calculates and returns a list of zeros np arrays of shapes \((sizes[i],sizes[i+1])\) for \(i\) from 0 to the length of \(sizes -2\).

\[
\text{def zeros_biases(sizes):}
\]

Calculates and returns a list of zeros np arrays of size \(sizes[i] (rank 1)\) for \(i\) from 0 to the length of \(sizes -1\).

\[
\text{def create_batches(data, labels, batch_size):}
\]

Creates batches of training data, returns a list of batches from the training_data in order, each batch is of batch_size size. If batch_size does’t divide the length of data/labels then the last batch is the remaining items. Assume that data and labels are of the same size.
def add_elementwise(list1, list2):
Returns a list of the sum of each two elements with the same index, in order from 0 to length -1.
Assume that the two lists are of the same size.

Note: each function of the above can be implemented in one python line.

Second, go over the places in network.py where these function are used and make sure you implemented them correctly, then run main.py and make sure the neural network is training as supposed to be and the final accuracy is > 90%.

Note: you shouldn’t change anything in network.py.

Part 2

in this part we will see an example of problems that can be speeded up a lot using GPU, you will implement a function that calculates a histogram of a very large array, you are given an array of size 1000*1024 of values [0,256), the function should return a histogram of size 256 for the values in the given array.

Implement the functions in hist_functions.py:

1. def hist_cpu(A):
   Calculates the histogram in the cpu and returns it.
2. def hist_numba(A):
   Uses the njit to speedup the calculation of histogram in the cpu and returns it.
3. def hist_gpu(A):
   Calculates the histogram in the gpu by invoking the hist_kernel with 1000 blocks with 1024 threads each.

Then run the hist_functions.py on 1 core (flag -c1) to see time comparison and make sure that the gpu and numba calculations are correct, include a screenshot and an explanation of the gpu kernel in the below report.

*You must get a speedup of at least 20 with the gpu. (regarding hist_cpu)*

*hist_cpu function should be implemented straight forward, don’t insert trivial delays.

Note: you can use cuda atomic add and cuda syncthreads.
Part 3

In this part you will implement a CUDA kernel for calculating the `matmul` between two matrices.

As you noticed the neural network uses the `np.matmul` function to do the matrix multiplications, we are interested to check alternative implementations of functions for matrix multiplications, you have to implement the functions in `matmul_functions.py`:

1. `def matmul_trivial(X, Y):
   This function calculates matrix multiplication in the most trivial way of 3 nested for loops, implement it that way.
2. `def matmul_numba(X, Y):
   Uses njit numba compiler to speed the matmul_trivial function.
3. `def matmul_gpu(X, Y):
   This function calculates matrix multiplication in the gpu, you should implement the below `matmul_kernel` and call for this kernel to calculate the matrix multiplication, **you should utilize the gpu to get a speedup over numpy matmul**. (don’t expect a much better performance)
   Matmul_kernel should always be called with 1 thread block with 1024 threads, your implementation should take this into consideration.

Note: running `matmul_functions.py` will print matmul comparison of your implementations.
Make sure you run with `-c1` meaning that the serial part runs on one core.

After you finished implementing, fill the below report.

**Report:**

1. Give a detailed explanation of your `hist_kernel` implementation, include screenshot and calculate the speedup between `hist_gpu/hist_numba` and `hist_cpu`.
2. Give a detailed explanation of your `matmul_gpu` implementation.
3. Run `Matmul_functions.py`, include a screenshot and explanation of the results.
4. Run `main.py` with comparison of all 3 implementations, include a screenshot and explanation of the results.
5. Now run `main.py` with 32 cores (flag `-c32`), include a screenshot and explanation of the results.

Note: you are not expected to understand how auto numba njit works, so just explain what you think is going on according to ideas learned in class.
Notes and Tips

1. You can add variables and prints as you need, but your code must be clear and organized.
2. Don’t remove prints or comments already in the code, adhere to instruction comments.
4. It’s recommended that you work with PyCharm, but performance should only be measured in the course server, you can simulate a gpu by setting the environment variable `NUMBA_ENABLE_CUDASIM` to 1, but take into consideration that it will be very very slow.

Server

Connecting to the course server:

1. ssh to rishon server using your user name: `ssh -X user@rishon.cs.technion.ac.il`
2. ask for a bash shell and then start working: `srun -p 236370 -c1 --gres=gpu:1 --pty bash`

For more info: https://hpc.cswp.cs.technion.ac.il/2018/04/09/%D7%9E%D7%93%D7%A8%D7%99%D7%9A-%D7%9C%D7%9E%D7%A9%D7%AA%D7%9E%D7%A9-2366056-%D7%90%D7%91%D7%99%D7%91-2018/

Install the necessary packages before you start working (through bash shell, only once):
`pip3 install --user --upgrade pip`
`pip3 install --user numpy numba`

Submission

Submit a hw1.zip with the following files only:

1. `util.py` with your implementation.
2. `hist_functions.py` with your implementations.
3. `matmul_functions.py` with your implementations.
4. `main.py` with your code.
Report example:

1. Give a detailed explanation of your hist_kernel implementation (flag -c1):
   ..... 
   ..... 
   ..... 

```
[sqasem@rishon1:~/hw1_cdp]$ python3 hist_functions.py
CPU: 3.11247264991843
Numba: 0.015951482113450766
CUDA: 0.06999670388177037
```

Gpu Speedup = ....
numba Speedup = ....

2. Give a detailed explanation of your matmul_kernel implementation (flag -c1):
   ..... 
   ..... 
   ..... 

3. Matmul_functions.py comparison:

```
[sqasem@rishon1:~/hw1_cdp]$ python3 matmul_functions.py
Numpy: 0.31894025206565857
Numba: 0.611919716000557
CUDA: 0.2618924794587027
```

The gpu matmul is better because ... we didn't get alot of performance boost because ...
   ..... 
   ..... 

4. main.py comparison:

```
[sqasem@rishon1:~/hw1_cdp]$ python3 main.py
Epoch 1, accuracy 92.54 %.
Epoch 2, accuracy 95.87 %.
Epoch 3, accuracy 96.01 %.
Epoch 4, accuracy 96.59 %.
Epoch 5, accuracy 96.59 %.
Time matmul_nb: 599.0606455339661
Epoch 1, accuracy 92.34 %.
Epoch 2, accuracy 94.8 %.
Epoch 3, accuracy 95.84 %.
Epoch 4, accuracy 96.77 %.
Epoch 5, accuracy 96.97 %.
Time matmul_nb: 116.32704997062683
Epoch 1, accuracy 91.71 %.
Epoch 2, accuracy 94.74 %.
Epoch 3, accuracy 95.62 %.
Epoch 4, accuracy 96.55 %.
Epoch 5, accuracy 97.07 %.
Time matmul_gpu: 205.008626301765442
Test Accuracy: 96.79% ...
```

Explanation:
   ..... 
   .....
5. main.py comparison with 32 cores (flag -c32):
   [Screenshot]
   Explanation:
   .....  
   .....  
   .....  

Note: this is just an example, in your report give the results you got and explain them, there’s no one right answer.