Concurrent and Distributed Training for Deep Learning Methods

Assignment 1
Introduction to Parallel Deep Neural Networks

Due Date: 23/12/24

T.A in charge: Guy Sudai, guy.sudai@campus.technion.ac.il

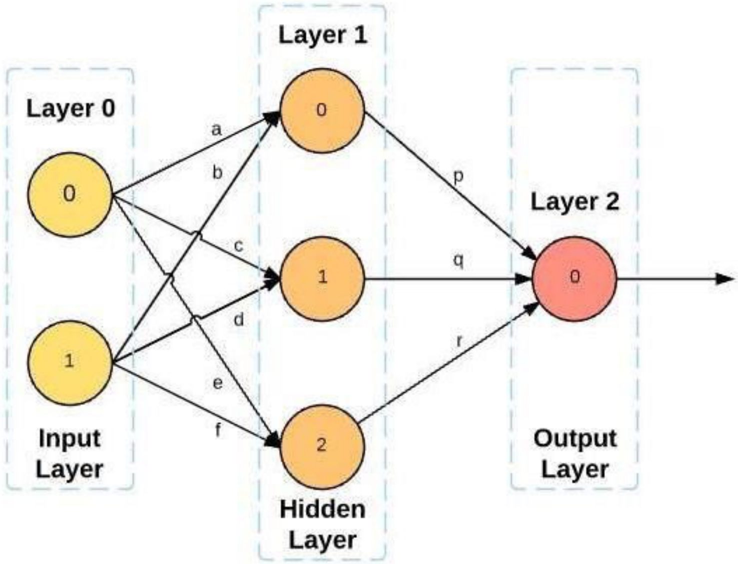
Part 1

Brief Background

Please refer to lecture 3, Tutorial 3 and the mandatory reading part of HW0.

Conventions

We are going to implement a simple neuronal network. For this, we will use in our implantation the following conventions:



<u>Layers</u>

The input layer is the 0_{th} layer, and the output later is the L_{th} layer.

The number of layers is $N_L = L + 1$.

The size-layer vector is a vector of length $L + 1$ where element i represents the number of neurons in layer i. In the example above, the size-layer vector is [2,3,1].

We will note $size(l_i)$ as the number of neurons in layer l_i.

<u>Weights</u>

Weights in this neural network implementation are a list of numpy matrices.

Given a neural network with $N_L = L + 1$ layers, and a size-layer vector $[size(l_0), \dots, size(l_L)]$:

The weight list is of list of length L, denoted as $[w_{01}, w_{12}, \dots, w_{(l-1)l}]$,

where w_{ij} is a matrix of shape $\left(size(l_i), size(l_j)\right)$, which corresponds to the weight matrix between layers i, j in the network.

In the example above, the weight list is: $[w_{01}, w_{12}]$, where $w_{01} = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$ and $w_{12} = [p \quad q \quad r]$.

Biases

Biases in this neural network implementation is a list of one-dimensional vectors.

Given a neural network with $N_L = L + 1$ layers, and a size-layer vector $[size(l_0), \dots, size(l_L)]$:

Bias list is of length L, denoted as $\left[b_{01}, b_{12}, \dots, b_{(l-1)l}\right]$,

where b_{ij} is a one-dimensional vector of size $size(l_j)$, where entry k represents the bias of neuron number k in the j_{th} layer.

In the example above, the bias list is: $[b_{01}, b_{12}]$, where $b_{01} = [0, 1, 2]$ and $w_{12} = [0]$.

Z

For input vector x to layer l_{th}, the output z is defined as follows:

$$z = w_{(l-1)l}^{T} \cdot x + b_{(l-1)l}$$

Activations

Activations of the l_{th} layer is the operation of activation function on the z output of the same layer. The result from the above calculation is used as the input for the $(l+1)_{th}$ layer.

Implementation:

You will implement server basic components of the described neural network.

In utils.py file, implement the following:

def sigmoid(x): Calculates the standard sigmoid function. This function outputs $f(x)$.

- Sigmoid is a standard activation function, where $f(x) = \frac{1}{1+e^{-x}}$.

def sigmoid_prime(x): Calculates the derivative function of sigmoid with input x.

def random_weights(sizes): Calculates and returns a list of random xavier initialized numpy arrays of shapes $(size[i], size[i+1])$ for $0 \leq i < N_L$.

- Look at the end of utils.py for xavier initialization implementation.

def zeros_weights(sizes): Calculates and returns a list of zeros numpy arrays of shapes $(size[l_i], size[l_{i+1}])$ for $0 \leq i < N_L$.

def zeros_biases(list): Calculates and returns a list of zeros numpy arrays of size $size(l_i)$ for $0 < i \leq N_L$.

def create_batches (data, labels, batch_size): Creates batches of training data.
Returns a list of batches from the training data, where each batch is of batch_size size.
If the length of dataset is not dividable by the batch_size, then the last batch will get the remaining samples.

- Assume that data and labels are of the same size.

def add_elementwise(list1, list2): Returns list3 which is an elementwise of list1, list2.

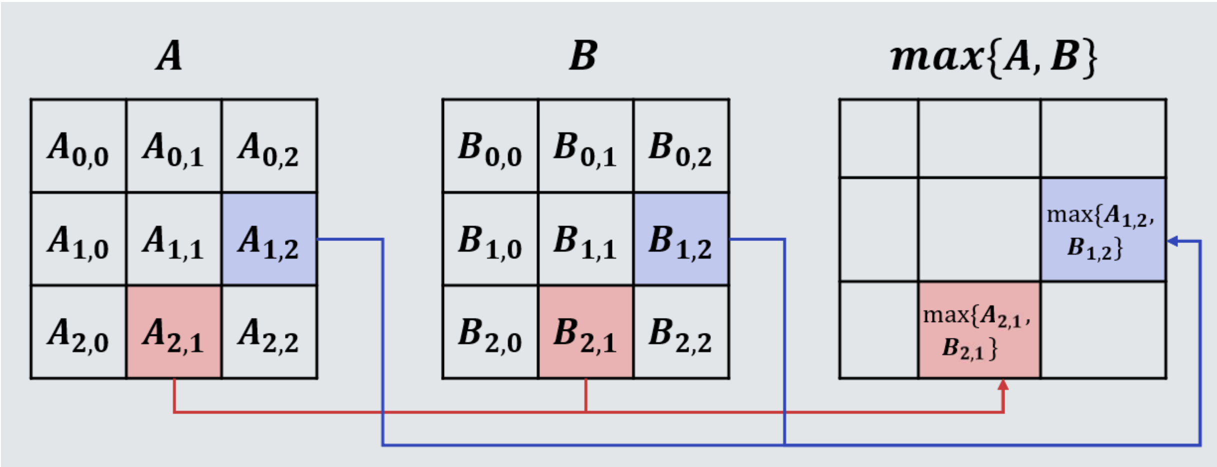
- Assume list1, list2 is of the same size.

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

Now check the following:
Run main.py, and make sure the neural network is training as supposed.
Make sure the final accuracy is above 95%.

Note: You can adjust the learning rate in main.py.

Part 2



In this part you will see how we can achieve a significant speed-up using the GPU.

You will implement a function that calculates the element-wise maximum between two larger scale matrixes.

Given matrixes A, B of size $(1000, 1000)$ with integer values of range $[0,255]$, the function should return a matrix C of the same size, where $C_{ij} = \max\{A_{ij}, B_{ij}\}$.

Implement the following functions in the file max_functions.py:

def max_cpu (A, B): Calculates the element-wise maximum on the CPU and returns it.

- Do not use numpy vectorize operations.

def max_numba (A, B): Use the NJIT to speed up the above calculation.

def max_gpu (A, B): Calculates the element-wise maximum between A, B on the GPU, by invoking the max_kernel function with 1000 block, where each block contains 1000 threads.

- max_kernel is defined in the same file.

Now do the following:

Run max_functions.py on 1 core (flag -c 1) to see time comparisons.

<u>Make sure that the NUMBA and GPU calculations are correct.</u>

Include a screenshot of the time comparison between the three methods, and an explanation about the GPU implementation, in the report to be followed.

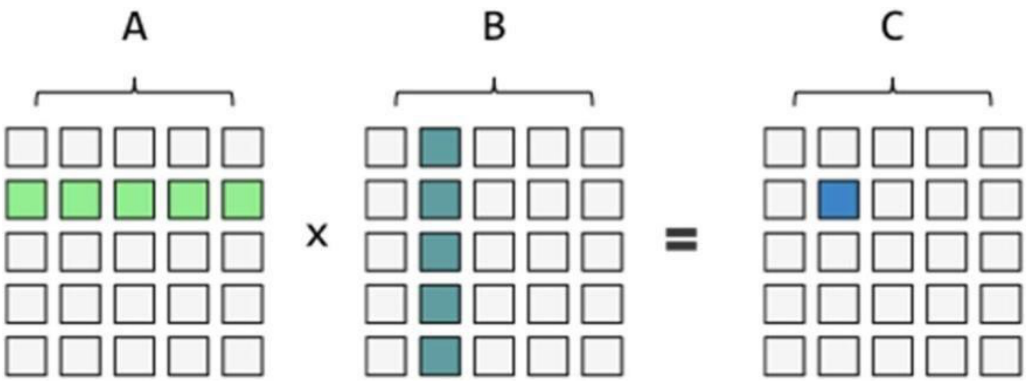
In addition, run the max_functions.py with 2, 4, and 8 cores, and explain the difference.

Notes:

1. You must get a speed-up of at least 40, between the CPU run-time to the GPU run-time.
2. The max_cpu function should be implemented in a trivial way. Do not insert trivial delays.
3. You must implement the functions yourself – don't include any existing functions from numpy or any other library.
4. You may use cuda atomic add, cuda atomic max and cuda syncthreads.

Part 3

In this part you will implement a matrix calculation between two matrixes.



Specifically, we are interested in a function that given matrix X will calculate $X \cdot X^T$ efficiently.

Implement the following functions, in matmul_functions.py:

def matmul_transpose_trivial(X): Calculates $X \cdot X^T$ in the most trivial way – using 3 nested for loops.

def matmul_transpose_numba(X): Use NJIT to speed up the function from above.

def matmul_transpose_gpu(X): Calculates $X \cdot X^T$ on the GPU.

- You should implement matmul_kernel and use it.
- matmul_kernel should always be called with 1 thread block which contains 1024 threads.

Run matmul_functions.py, which will generate comparisons of the run-time of the functions above.

Notes

Report

You must include a report.

1. Provide a detailed explanation of your max_kernel implementation, include screenshot, and calculate the speedup between max_gpu/max_numba and max_gpu/max_cpu, and explanation of the results.
2. Provide a detailed explanation of your matmul_kernel implementation, include a screenshot and explanation of the results.

Notes and Tips

- You can add variables and prints as you need, but your code must be clear and organized.
- Don't remove prints or comments already in the code, adhere to instruction comments.
- Document your code thoroughly.

Server

Full explanation can be found in the Jupyter notebook at the course website (in HW1 section).

Submission

Submit a hw1.zip with the following files only:

- utils.py with your implementation.
- max_functions.py with your implementations.
- matmul_functions.py with your implementations.
- hw1.pdf report of performance analysis.