# Assignment 1

## Introduction to parallel deep neural networks

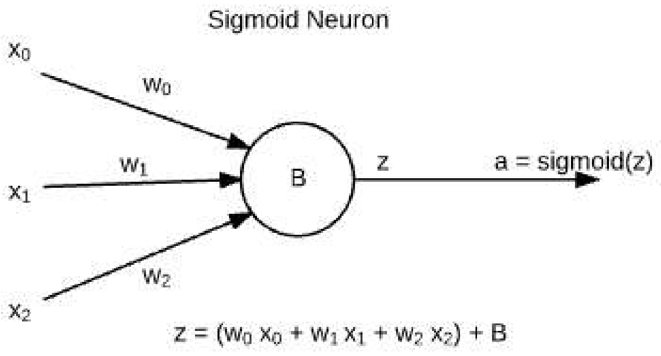
Due Date: Wednesday 9/12/2019 23:59

Questions:

- Questions regarding this assignment should only be asked on Piazza.
- Problems with connecting to servers should only be sent to amiteisinger@campus.technion.ac.il
- Postponements can only be authorized by the TA in charge amiteisinger@campus.technion.ac.il
  (remember the late submission policy published on the course website)

# Part 1

## Brief Background:



Neural networks are made up of building blocks called Neurons.

A Neuron is a mathematical function defined by:

$f(Z) = x$ where $f : R \to R;\ Z : W \times X \times B \to R;\ W, X \in R^n;\ B \in R$

Z is defined by:

$$Z = \vec{X} \cdot \vec{W} + B$$

f is some non-linear function. In our case this function is a simple kind of logistic function called Sigmoid and therefore a Neuron is called a Sigmoid Neurons.

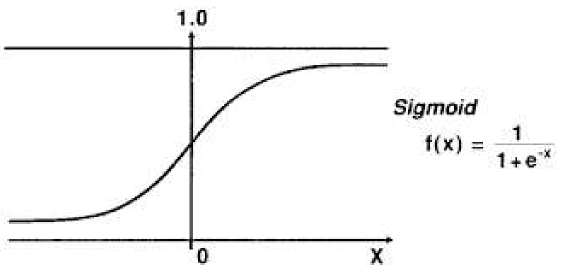
Logistic function equation: $f(x) = \dfrac{L}{1 + e^{-k(x - x_0)}}$

Where:

- L is the maximum value of f
- k is the logistic growth rate
- $x_0$ is x value where $f(x) = \dfrac{L}{2}$

Standard logistic Sigmoid function:

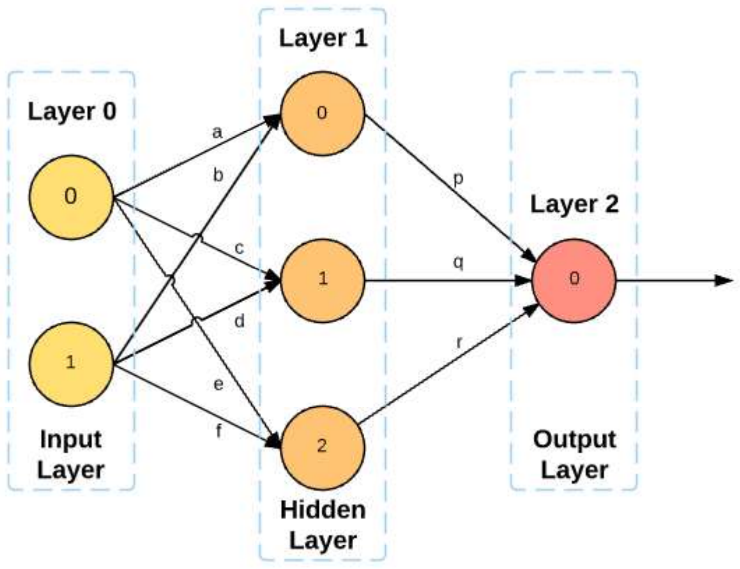
$$L = 1, k = 1, x_0 = 0$$



X is a vector of input values, W is a vector of weights where $W[i]$ multiplies $X[i]$. B is the bias of the Neuron which is added to Z. the output of a Neuron is also called activation.

A neural network is made up by stacking layers of neurons (typically all Neurons in a layer are of the same kind i.e. the same non-linear output function).

## 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 $l^{th}$ layer of the network (Denoted as $wl$ ).
An element of this matrix is denoted as $wl_{jk}$. It is a part of the $j^{th}$ row, which is a collection of all weights entering the $j^{th}$ neuron, from all neurons (0 to k) of $(l-1)^{th}$ layer.
No weights enter the input layer, hence weights[0] is redundant, weights[1] enters layer 1 and so on.

```
In the above example: weights = |¯  [[]],   [[a, c, e],   [[p, q, r]]  ¯|
                                |            [b, d, f]]                 |
                                |_                                     _|
```

### Biases

Biases in this neural network implementation is a list of one-dimensional vectors (`numpy.ndarrays`). `biases[l]` is a vector of biases of neurons in the $l^{th}$ layer of network. Input layer has no biases, hence `biases[0]` is redundant, `biases[1]` matches layer 1 and so on.

```
In the above example: biases =   |‾  [],   [0, 1, 2],   [0]  ‾|
                                 |_                          _|
```

### 'Z's

For input vector x to a layer l, z is defined as: $z_l = w_l . x + b_l$
Input layer provides x vector as input to layer 1. It has no input nor weight and bias i.e. $zs[0]$ is redundant.

The dimensions of $zs[0]$ are the same dimensions as biases.

### Activations

Activations of the $l^{th}$ layer are outputs from neurons this layer and are used as input for the $(l-1)^{th}$ 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:

In this part you will implement several basic components of a neural network

First, you need implement the functions in utils.py:

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

**def** sigmoid_prime(x):
Calculates the derivative function of sigmoid $f$ with input x and is given by $f(x) * (1 - f(x))$.

**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.

**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.

**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.

**def** create_batches(data, labels, batch_size):
Creates batches of training data, returns a list of batches from the training_data in order, where each batch is of batch_size size, if batch_size doesn'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.

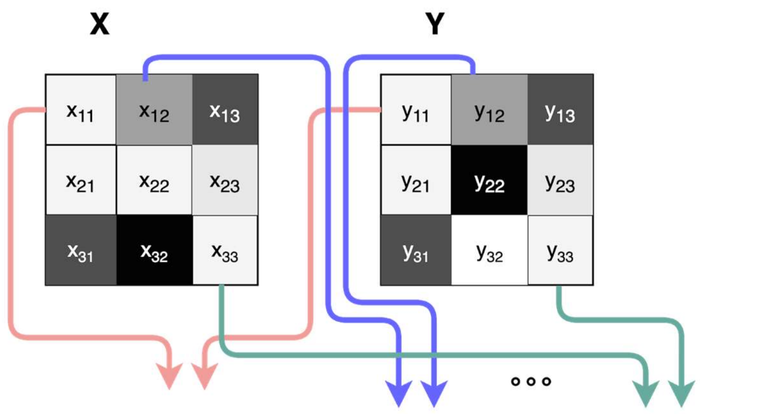
**def** add_elementwise(list1,list2):
Returns a list of the sum of each two elements with the same index, in order from 0 to length -1. Assume that the two lists are of the same size.

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

Second, go over the places in network.py where these functions 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



$$||X-Y||_p = \sqrt[p]{|x_{1,1}-y_{1,1}|^p + |x_{1,2}-y_{1,2}|^p + \ldots + |x_{3,3}-y_{3,3}|^p}$$

In this part we will see an example of problems that can achieve a significant speed-up using GPU. We will implement a function that calculates the p-norm induced distance between two large scale arrays. Given two arrays of size 1000*1000 with values in the range [0,256) and p (the order of the norm which induces the distance), the function should return the scalar result of the p-norm induced distance between the inputs..
Implement the functions in norm_functions.py:

1. **def** dist_cpu(A, B, p):
   Calculates the p-norm induced distance on the CPU and returns it.
2. **def** dist_numba(A, B, p):
   Uses the njit to speed-up the calculation of p-norm induced distance on the CPU and returns it.
3. **def** dist_gpu(A, B, p):
   Calculates the p-norm induced distance on the GPU by invoking the dist_kernel with 1000 blocks with 1000 threads each.

Then run the dist_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.

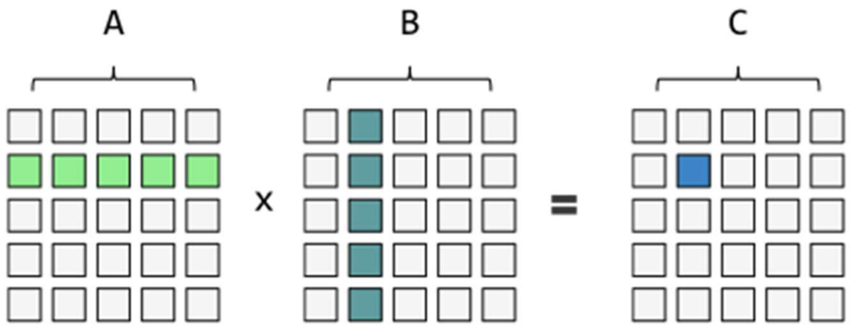
**Notes:**

**1. You must get a speedup of at least 40 with the GPU (regarding dist_cpu) on the server.**
**2. dist_cpu function should be implemented straight forward, don't insert trivial delays.**
**3. In the case of dist_gpu, the p-root should be implemented on the cpu after performing the other operation on the gpu.**
**4. You must implement your own functions – do not use existing implementations from numpy or any other library.**

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

## Part 3

in this part you will implement a CUDA kernel for calculating the matrix multiplication 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. We also interested of matrix transpose, so we will implement a function that given $X$ calculates $X \cdot X^t$.
Implement the functions in matmul_functions.py:

1. **def** matmul_transpose_trivial(X):
   This function calculates matrix multiplication in the most trivial way of 3 nested for loops, implement it that way. The result is $X \cdot X^t$
2. **def** matmul_transpose_numba(X):
   Uses njit numba compiler to speed the matmul_trivial function.
3. **def** matmul_transpose_gpu(X):
   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. Provide a detailed explanation of your dist_kernel implementation, include screenshot and calculate the speedup between dist_gpu/dist_numba and dist_cpu.
2. Provide a detailed explanation of your matmul_kernel implementation
3. Run Matmul_functions.py, include a screenshot and explanation of the results.
4. ~~You must implement your own functions – do not use existing implementations from numpy or any other library.~~

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.
3. Document your code thoroughly.
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 slow.

## Server

Full explanation regarding connection to the course server can be found at the course site.

**Important:** setup your environment using the following script (only once per user):

~~pip install numpy numba matplotlib sklearn scipy --user~~
~~pip install colorama==0.3.9 --user~~
See pip.pdf on the course website

For more info:
https://hpc.cswp.cs.technion.ac.il/2020/08/31/lambda-computational-cluster/

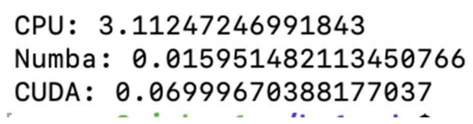
## Submission

Submit a hw1.zip with the following files only:

1. ~~util.py~~ with your implementation.
2. dist_functions.py with your implementations.
3. matmul_functions.py with your implementations.
4. A hw1.pdf report of performance analysis of **maximum 3 pages, make it concise.**

## Report example:

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

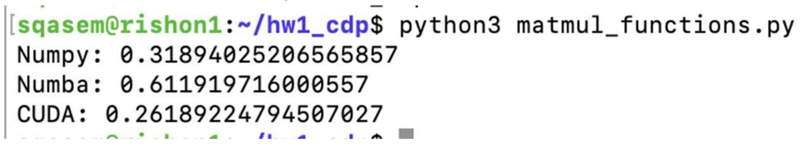


```
CPU: 3.11247246991843
Numba: 0.015951482113450766
CUDA: 0.06999670388177037
```

   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: 0.26189224794507027
```

   The gpu matmul is better because … we didn't get a significant performance boost because …
   …..
   …..
   …..

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