# Assignment 2

**Parallel Deep Neural Networks and Data Augmentation**

Due Date: 6.1.2021 23:59
TA in charge of exercise: Sagi.
Questions:

- Questions regarding this assignment should only be asked on Piazza.
- Problems with connecting to servers should only be sent to Amit.
- Postponements can only be authorized by the TA in charge Amit.

# Part 1

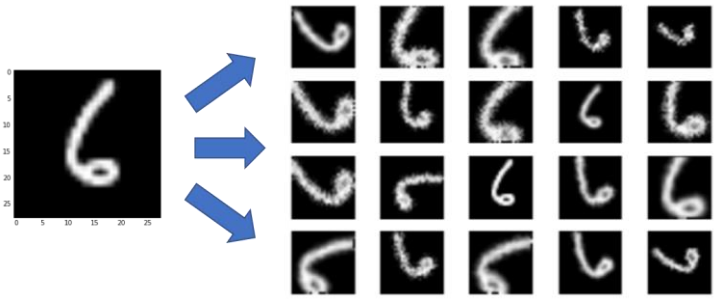
## Brief Background

One of the biggest issues concerning DNNs to this day, is our lack of understanding (prior to the training procedure) whether we have enough data in order to train a model for a given task. The problem gets even more complicated when we add the fact that every additional data collection is associated with some cost. This cost can be in terms of an expense, human effort, computational resources and off course - time. This leads us to the understanding we must always attempt to maximize the performance while minimizing the cost in the process.

### Data Augmentation

Data Augmentation is a well known technique that can be used to artificially expand the size of a given training dataset, by creating modified versions of instances in the dataset. The major advantage of this technique is that it does not require additional data but it creates the effect of enriching our dataset by just utilizing better the data we already have.

There are many ways to augment data. In images, you can rotate the original image, change lighting conditions, crop it differently, so with one image you can generate many different sub-samples.



In the example above we show that using the Data Augmentation technique with images we are able to generate different altered results of a single image.

---

## Our Goal

In this part of the assignment we will train a DNN with images being generated by the Data Augmentation process over the original MNIST dataset, Inorder to reach higher accuracies. https://en.wikipedia.org/wiki/MNIST_database

## Implementation Overview

As a continuation of the last exercise (Ex 1 - part 1), you will implement two classes:

***Worker* (under *preprocessor.py*)**

This class inherits from “Process” (imported from the multiprocessing library in python) and contains multiple image augmentation operators (implemented as separate functions). The class must include the **job** queue member - containing the jobs needed to process, and a **result** queue member - containing the finished processed jobs. Additionally this class contains two functions:

1. ***process_image*** - applying all augmentation operations sequentially over an input image and returning the result.

2. ***run*** - process images from the jobs queue and adding their augmented variation to the result queue.

***IPNeuralNetwork* (under *ip_network.py*)**

This class Inherits from “NeuralNetwork” and overrides two methods:

1. **fit** - Originally this method encapsulates the training procedure of the DNN via Stochastic Gradient Descent. You are required to add the current logic, the creation and destruction of Preprocessor Workers responsible for producing augmented batches.

2. **create_batches** - This method receives an original dataset batch and returns an augmented variation of the batch, created by the Workers mentioned in the above method.

---

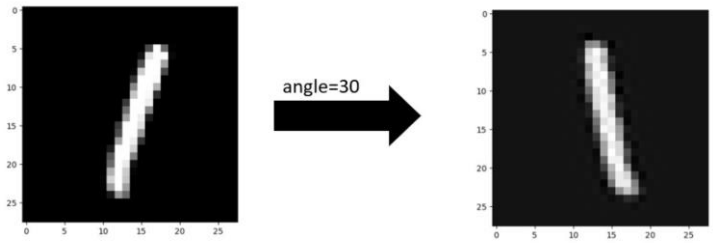
## Implementation Internals

**Worker methods under preprocess.py:**

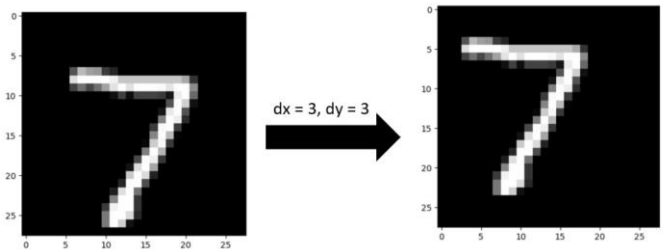
**def __init__ (self, jobs, result):**
Initializes the class and it’s members as you think will help you in this exercise.
Note you may add more arguments to the function.

**def rotate (image, angle):**
Rotate the image by the given angle.
(Hint: You should look at scipy library)

Example for using the rotate function




**def shift (image, dx, dy):**
Shift the given image by dx cells to the left and dy cells upwards.
If the coordinates are out of range replace it with black (0).
(Hint: You should look at NumPy library)
Example for using the shift function:




**def step_func (image, steps):**
Step function is a function that increases or decreases abruptly from one value to another. You should manipulate every pixel according to the step function that matches the given argument. The argument “steps” acts as the number of steps in the range [0-1] increasing in a steady interval. In the general case:

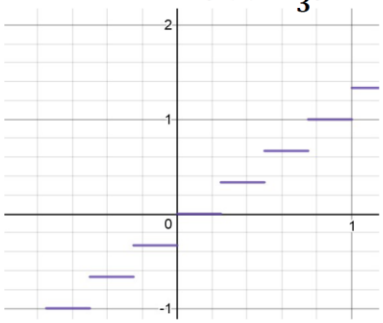
$$f(x) = \frac{1}{steps-1} floor(steps \cdot x)$$

You may only use numpy library in the function.

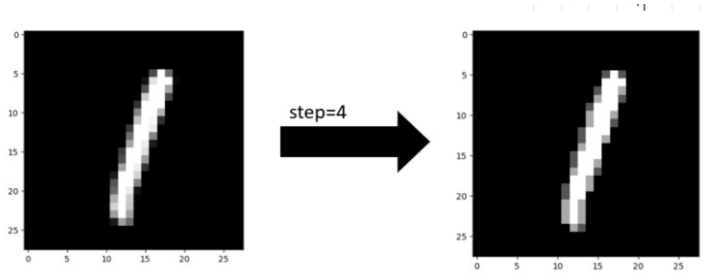
Example: if $steps = 4$ then the function should hold that:

$$f(x) = \begin{cases} \frac{0}{3} & 0 \le x < 0.25 \\ \frac{1}{3} & 0.25 \le x < 0.5 \\ \frac{2}{3} & 0.5 \le x < 0.75 \\ \frac{3}{3} & 0.75 \le x < 1 \end{cases}$$

$$f(x) = \frac{1}{3} floor(4x)$$




An example of the step_func result:




**def skew (image, tilt):**

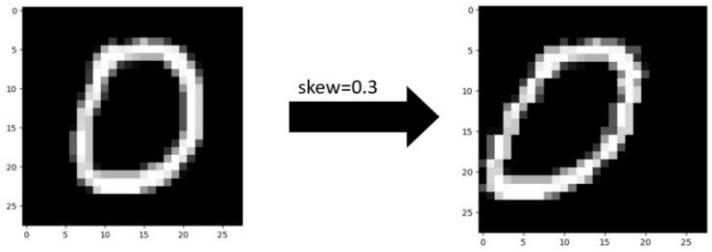
Manipulate image by given tilt. For your convenience, we give you a skew formula to use:

$$SkewedImage[y][x] = Image[y][x + y * tilt]$$

If the coordinates are out of range replace it with black (0).

You may only use numpy library in the function.

Example:




**def process_image (self, image):**

Run the previous functions sequentially to manipulate the given image. You should use the functions you implemented above in any order you would like.

Use python's random to decide the arguments for the functions (experiment with the bounds to see which gives good results).

Try to get better accuracies when using the augmented data.

**def run (self):**
The function should produce augmented images as stated above.

---

**IPNeuralNetwork Implementation ip_network.py:**

**def create_batches (self, data, labels, batch_size):**
You should override this function, so it will create batches of augmented images from workers rather than using the original batches.

**def fit (self, training_data, validation_data=None):**
You should override this function, so it will create Preprocessor Workers depending on the number of CPUs used before running the super function and destroying them after. (use os.environ['SLURM_CPUS_PER_TASK']).

Trivial solution with only one Worker or a Worker for every single job will not get full points.

**Part 1 Important Notices**

1. You must add **utils.py** from the previous HW submission to your working directory in order to run NeuralNetwork, but do not submit it again!

2. You were provided an implementation of NeuralNetwork with some changes to the structure of the class. The changes are listed below:
    a. The class NeuralNetwork can now be supplied with number_of_batches.
    b. The function create_batches(...) is a class function (Instead of being in Utils.py)
    c. create_batches(...) implementation in NerualNetwork is returning random batches according to the number_of_batches.

3. Throughout the augmentation functions we advise using numpy and scipy library built-in functions rather than implementing them yourself.

# Part 2

In this part, you will implement a simple queue and replace the result queue member in the IPNeuralNetwork class with this implementation (you don't need to replace jobs queue!).

You will have to use **Pipe** and **Lock** which you have seen in class. The class needs to work on multiple writers and one reader (meaning – you can assume that in any given moment only one process may try to read from the Queue. However, any number of processes may try to write). Determine where we must synchronize and where synchronization is redundant.

In **my_queue.py** implement the following methods of the **MyQueue** class:

**def __init__ (self):**
Initialize the queue and it's members.

**def put (self, msg):**
Send a msg through a pipe.

**def get (self):**
Read a msg from a pipe.

Hint: this is a very easy and short part. Don't write a complicated implementation (2-3 lines of code maximum for each function).

# Part 3

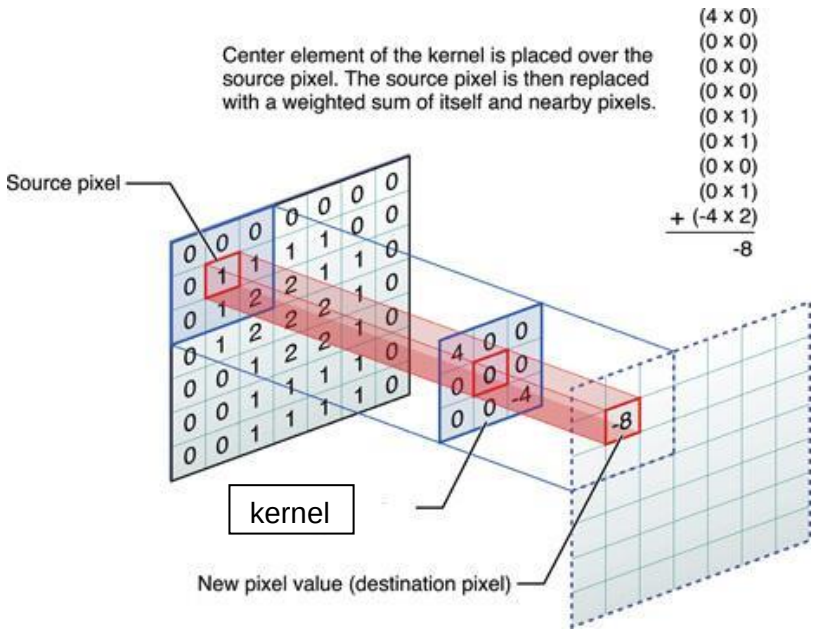
### Correlation

Correlation is a mathematical operation including a combination between two input signals to form a third signal. The correlation is commonly used in various fields and applications. In image processing it is used for blurring, sharpening, embossing, edge detection and much more.

### Image Correlation

The correlation result between a given image and a kernel (In image processing, a kernel is a small matrix) could be thought of as replacing every pixel in the image with a weighted summation of its neighbors where the weights are configured by the kernel.

**Example 1** - Let us examine the case of a 3 by 3 kernel of ones. In this case, the correlation result is exactly the summation of all joint neighbors.
**Example 2** – as explained in the following image:




Note: some kernel elements can be out of bounds for near-edge pixels. In these cases, “pad” the image with zeros (meaning - all values outside the image are treated as zeros).

**Implementations in filters.py:**

In this part you will implement the correlation, once using **njit** to speedup calculations and another time using **GPU**.

1. def **correlation_numba (kernel, image):**
   Implement the function using njit to speed up the calculation

2. def **correlation_gpu (kernel, image):**
   Implement the function using cuda.jit

Make sure that the results of gpu and numba calculations with **kernel** and **image** are equal to scipy.signal convolve2d (**flipped_kernel**, **image**) where **flipped_kernel** is **kernel** after flipping the rows and columns in order (i.e essentially rotating the matrix by 180 degrees).

# Part 4

**Sobel Operator**

In this part we shall use the correlation function implemented in filters to see one of its applications. Under **sobel_operator(...)** (located in filters.py) load **"data/image.jpg"** and calculate the operations below. Use the numba correlation implementation from Part 3.

$$filter = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

$$G_x = correlation\,(filter, Image)$$
$$G_y = correlation\,(transpose\,(filter)\,, Image)$$

$$Image - sobel_{ij} = \sqrt{{G_x}_{ij}^{\,2} + {G_y}_{ij}^{\,2}}$$

Show the resulting image (image-sobel) in the report.
Make sure that you get the same image when using the cpu correlation of sklearn (convolve2d with the flipped kernel as before)
To view the image, run test_filters() on your local computer with the numba_correlation implementation.

For further reading on the sobel operator and it's uses:
https://www.tutorialspoint.com/dip/sobel_operator.htm

**Guidelines:**

1. use **numpy** for:
    a. transpose(...) - transposes matrix
    b. sqrt(...)
    c. pow(...)
2. use **imageio** for
    a. imread(...) - loads image
3. use **matplotlib.pyplot** for:
    a. imshow(image, cmap='gray') - displaying image

# Report

**Part 1**

1. Run main.py with [8, 16, 32] cores (flag -c<core_number>)

2. Compare runtime of IPNerualNetwork between different core numbers. Include a screenshot and a short explanation about what number of cores gave the best performance and for what reason.

3. Run main.py again. Compare the accuracy percentage for different epochs between NeuralNetwork and IPNeuralNetwork. Include a comparison table and an explanation.

4. Answer: Why are we using processes and not threads?

5. Answer: Give two ideas how we could accelerate even more the training phase.

---

**Part 2**

6. Explain your implementation in part 2 and the reason you decided to implement it that way.

---

**Part 3**

7. Give a detailed explanation of your correlation_numba and correlation_gpu implementation.
8. Run filters_test.py on 1 core (flag -c1) to see time comparison.
9. Include screenshot and calculate the speedup between them and the scipy's convolve2d.
10. Answer: what will happen if we use a larger kernel?

---

**Part 4**

11. Show the result of the sobel operator using matplotlib.pyplot
    To view the image, run test_filters() on your local computer with the numba_correlation implementation.

**Notes and Tips**

1. Notice that in part 1, the image is a numpy array of size 784 while you should manipulate it as a matrix of size 28X28.
2. Notice that in part 1, rotate, shift, step_func and skew are static functions.
3. In filters_test.py there is a function called show_image(image). It will help you understand what your filters are actually doing.
4. You can add variables and prints as you need, but your code must be clear and organized.
5. Don't remove prints or comments already in the code, adhere to instruction comments.
6. Document your code thoroughly.
7. 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 to 1, but take into NUMBA_ENABLE_CUDASIM consideration that it will be very very slow.
8. Don't forget to install imageio on the server.

Server: all server instructions and installations are detailed in pip.pdf in exercise 1
(and here: https://hpc.cswp.cs.technion.ac.il/2020/08/31/lambda-computational-cluster/).

---

**Submission**

Submit a hw2.zip with the following files only:

1. preprocessor.py with your implementation.
2. ip_network.py with your implementation.
3. my_queue.py with your implementation.
4. filters.py with your implementations.
5. A hw2.pdf report of performance analysis of maximum 3 pages, make it concise.