Preparation of the project environment

For the course 1DT109 Acceleration System with Programmable Logic Components, 2022.
Department IT, Uppsala University

This documentation will walk you through

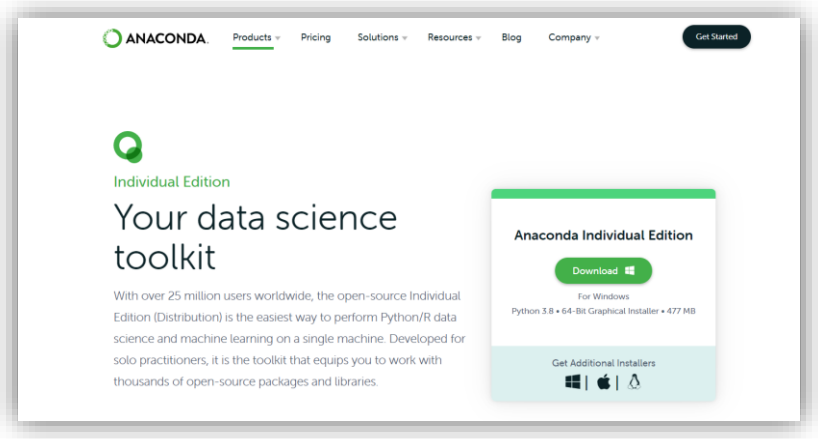
1. Installation of Anaconda, a virtual environment manager
2. Import the premade Anaconda environment file (hdr-nn.yml) into your system. This script will setup everything for you so that you do not need to maneuver yourself.
3. Run the default hdr-nn code and dump trained weights/bias for a given network.
4. Examine the default network structure

(The document is written with Windows 10 Version 21H1. However, if you work with Linux the whole setup should also work. If you indeed encounter problems, please feel free to contact with us at yuan.yao@it.uu.se; shiming.li@it.uu.se.)

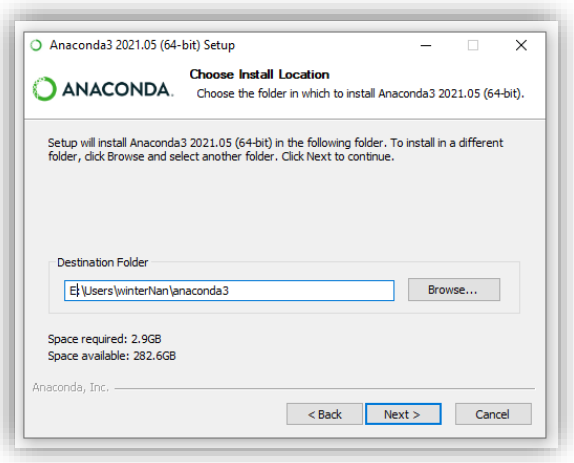
Throughout the documentation, you will find the **@TODO** labels. Such are concrete ideas/practices that you need to tackle with in your project.

1, Download Anaconda, the virtual environment manager

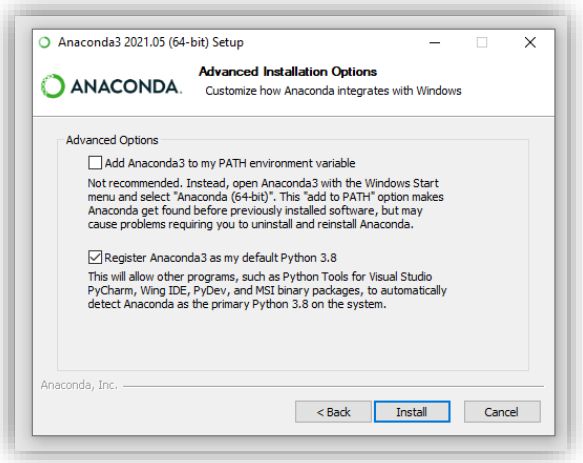
Download form the official website: https://www.anaconda.com/products/individual



Install it to your system. You can choose the installation path as you want. All the environment supporting files we will download later will go to this path, too.



Follow the following choices and proceed to Install.



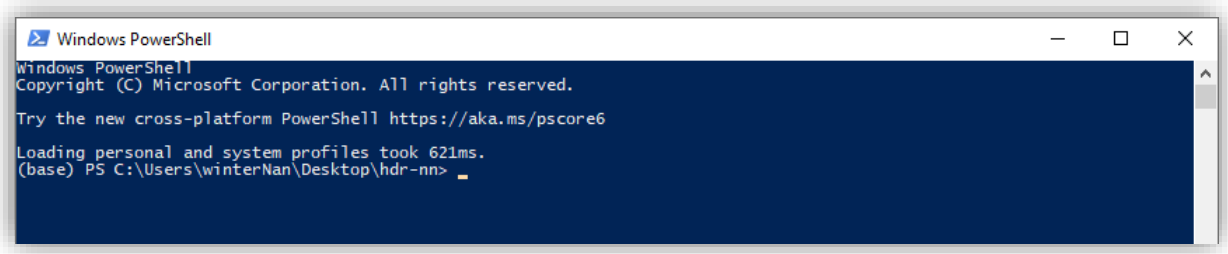
Done!

2, Import the premade Anaconda environment file hdr-nn.yml

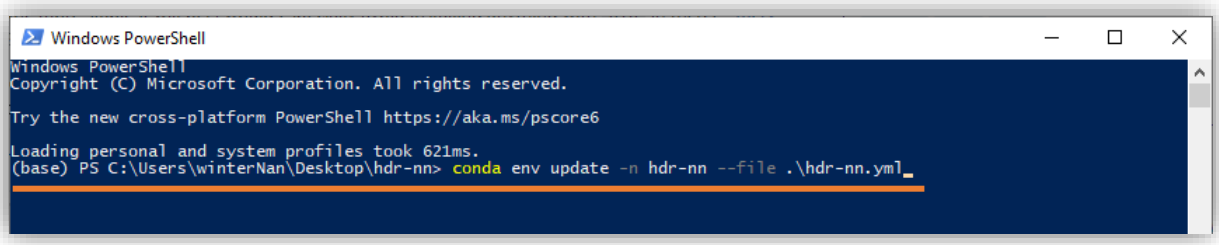
Download the zip file from studium (hdr-nn.zip). You will find a file named hdr-nn.yml inside the root. This file lists all the necessary packages/libraries that you need to run hdr-nn. You can open the file up using any program that can display ASCII. Take a look at it to get a overview of the packets and their versions that you will install.

Now we are ready to install our first virtual environment (hdr-nn) in Anaconda.

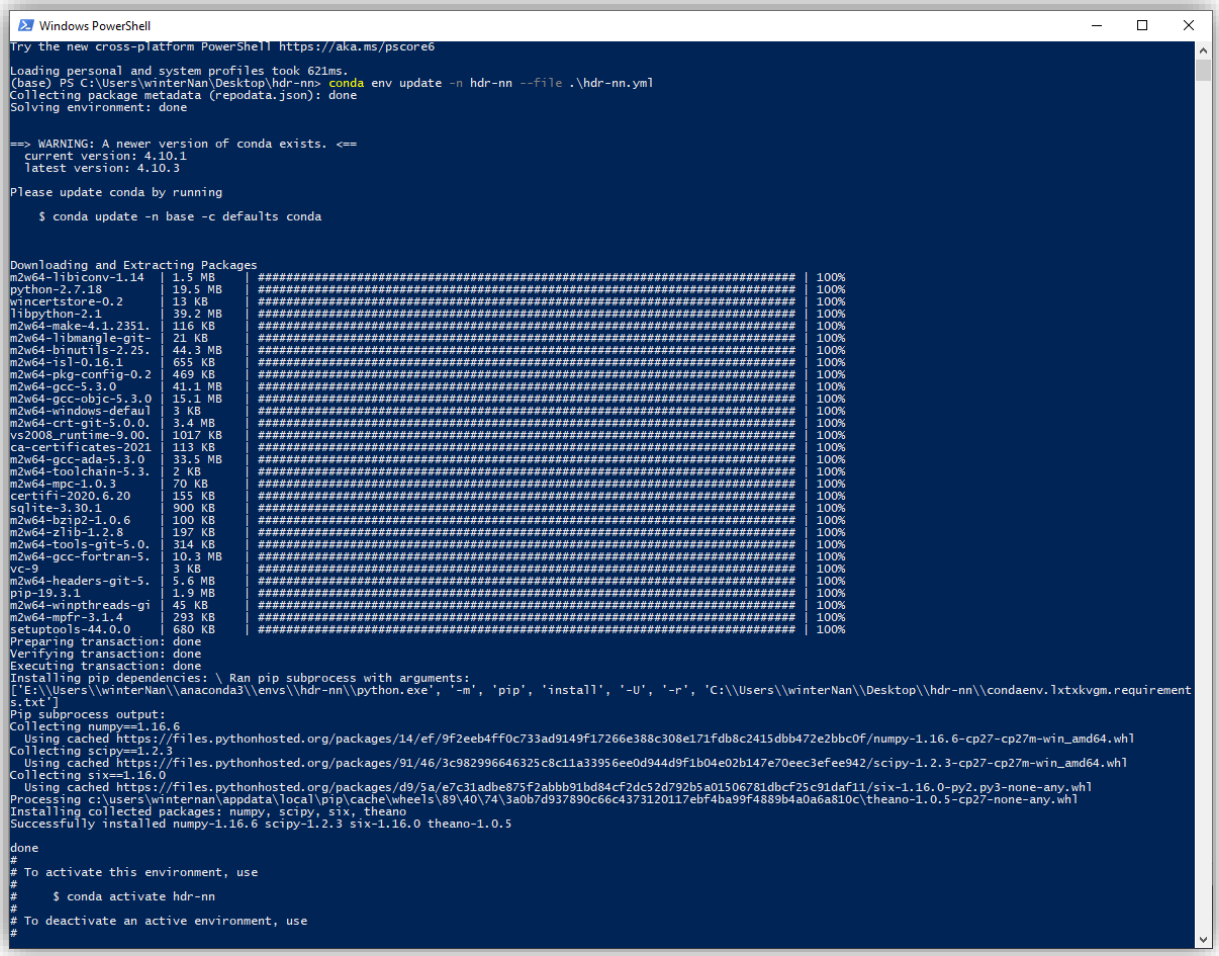
Open Windows power shell inside the root of hdr-nn.



Type the following command.



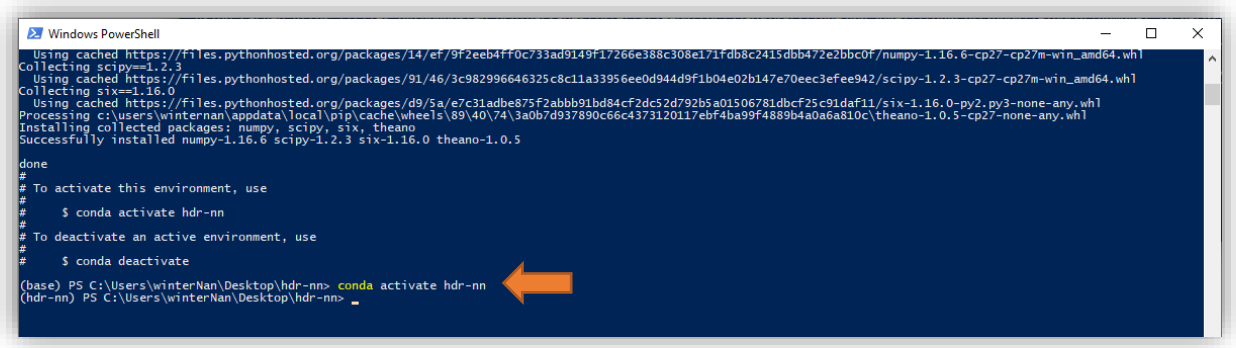
Note that there is a “(base)” indicator at the very front of the prompt. This shows that we are now inside the default environment in Anaconda, which is exactly the environment of your system before Anaconda is installed. Inside the (base) environment, all the software/binary/library versions that you’ve installed before Anaconda are used.



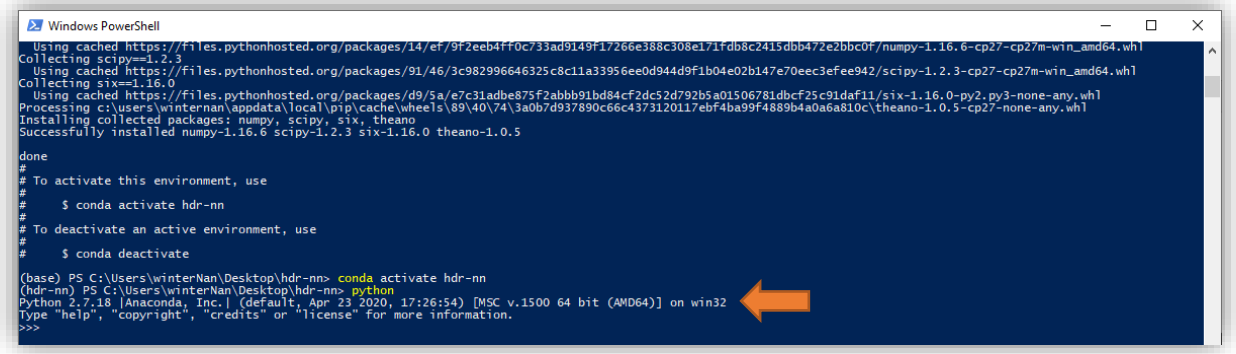
Import of the virtual environment is now finished. You can check what packages are installed/configured from the standard output dump (your screen). If your system has no python/whatever that is needed installed, now you have them all.

The virtual environment we’ve just imported is called “hdr-nn”, which locates inside the path: $INSTALL/anaconda3/envs/hdr-nn. $INSTALL if the path that you choose to where you install Anaconda.

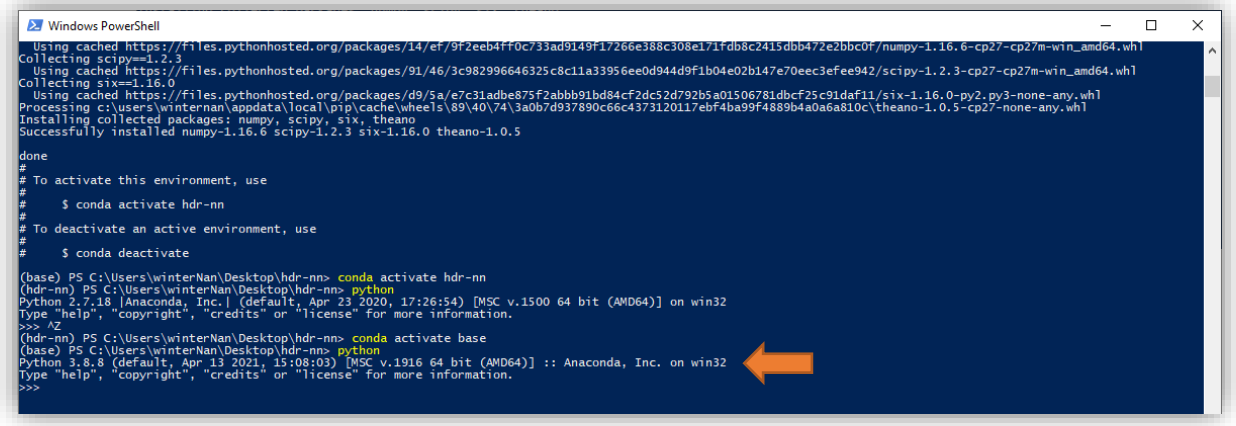
To active the virtual environment, type the following command:



If you have successfully active the virtual environment, the env indicator will change from (base) to (hdr-nn). Inside this env, you can access all the libraries with the version number specified in hdr-nn.yml. For example, type “python” and you will see python version 2.7.18:



This is because the python version specified in hdr-nn.yml is indeed 2.7.18. Otherwise, if you type “python” in (base), you will see python version 3.8.8:



This is because the default python version in my system is 3.8.8, before Anaconda is installed.

You can also see that you can switch to different virtual envs anytime you want by typing “conda active ENV-NAME”, where ENV-NAME is the name for the target environment.

Now everything is done with import the environment.

Important message.
Notice that all the libraries installed for env hdr-nn are kept inside the path "$INSTALL/anaconda3/envs/hdr-nn". That is to say, Anaconda is nothing magical at all but downloads and keeps all the binaries/libraries for each virtual environment in the path "$INSTALL/anaconda3/envs" so that you do not need to maintain your library version manually. Note that the more virtual environments you've created, the more disk you need to keep the corresponding files.

3, Run the default hdr-nn code and dump trained weights/bias for a given network.

You will find the code for hdr-nn inside the folder you've downloaded from sdudium. Put it to your favorite working space/path so that you can open it with your best development tool.

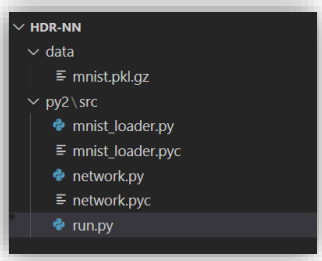
In my case, I just put it to my Desktop. 😊

And I will open the project inside VScode. You can open the project up with whatever IDE you want.

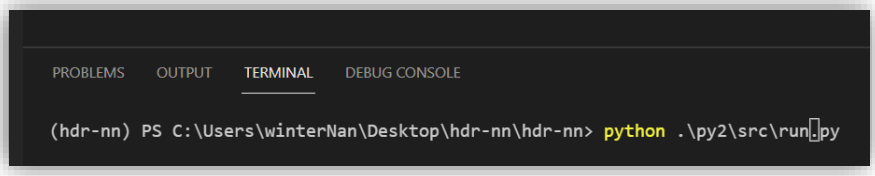
Now, switch power shell to the root of hdr-nn, that is, "./Desktop/hdr-nn". This is important since you need to run the python file there in order to load the training set correctly.

Important message.
The training images are kept as a compression filed named "mnist.pkl.gz" inside "hdr-nn/data". The loader we use to load these images are called "mnist_loader.py" which locates inside "hdr-nn/py2/src". "mnist_loader.py" will load "mnist.pkl.gz" from the relative path "./data/mnist.pkl.gz". Thus, to successfully run the code, your CWD need to be inside "../Desktop/hdr-nn".

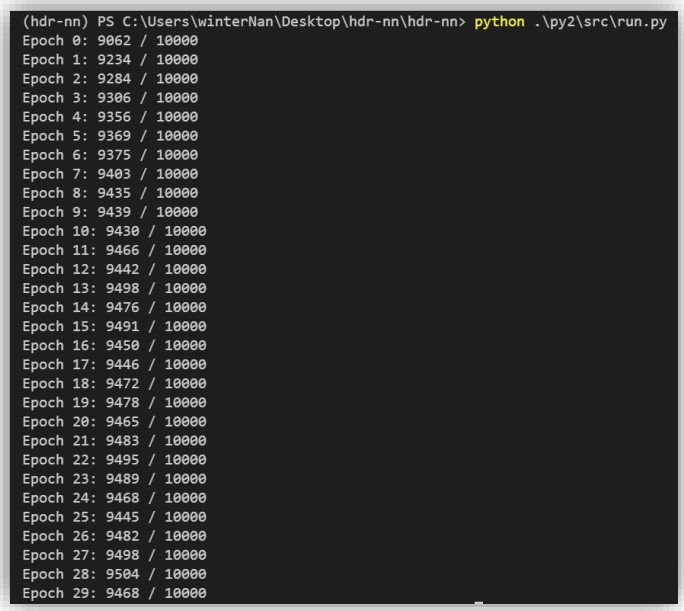
There are three files inside the project folder "./Desktop/hdr-nn/py2/src", as shown below (don't mind the .pyc files, they are auto-generated from the python interpreter). Let us run the code and then go through each file separately.



To run the code, type the following in the power shell (remember that you need to activate env "hdr-nn" beforehand).



Press "return" and you will get the following output from the command line. The total run time is ~3.5 min on my machine featured with Intel i7-9700K with 16GB DDR4. The output looks like this:



By default, the network is trained with 30 epochs, with mini-batch size of 10 and learning rate of 3.0. After each epoch of training, the network is tested with another set of test images (10,000 images) and report and accuracy. At initialization, all the weights/biases of the network are randomly picked up. We can see that with the training going on, we indeed get more accuracy, which in-turn goes up and down because of various reasons. (**@TODO.** Can you tell why is that?)

To dump the trained weights/biases, go to network.py. Uncomment the codes on line 74-76 to output the weights/biases AFTER training to the screen. You can also uncomment the codes on line 40-42 to output the weights/biases BEFORE training to the screen. Compare then to see what are changed and unchanged.

Because the weights on the first layer are pretty of a *huge* number (784), thus, you will observe a lot of dots (…) on the output. To suppress python auto-truncation of outputting large array, uncomment code on line 19-20.

@TODO. In the project, you may want to dump the weights/biases to a file instead of outputting them to the screen.

Now let us examine the three files.

4, Examine the default network structure

4.1, run.py

Let us start with run.py.

```
# ----------------------
# - read the input data:
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
training_data = list(training_data)
# ----------------------
# - run the network with SGD training
import network
```

```
net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Run.py is short and self-explanatory. On line 4-6, we call the mnist_loader to load three image vectors, that are, training_data to train the network, validation_data to validate our training (which is ignored in our simple network), and test_data which is used to test the network after each mini-batch training. For the scope of the project, you do not need to touch these codes.

In line 11, we import from the file Network.py with the actually network.

In line 12, we instantiate a network called "net", which serves as our baseline. The "net" is a simple structured network which contains 3 layers: an input layer, a hidden layer, and an output layer.

- The input layer contains 784 neurons, with each neuron getting input from a corresponding pixel from the 28*28 input image.
- The output layer contains 10 neurons, with each neuron output the possibility that the input image is one of the numbers of "0" ~ "9".
- The hidden layer contains 30 neurons. Each neuron receives all outputs from all neurons in the input layer, and each neuron outputs to all neurons on the output layer. That is, our simple network is a fully connected network without a convolution layer.

You can easily add more hidden layers to the network. For example, the following code will add one additional hidden layer to the network, which contains 60 neurons.

```
net = network.Network([784, 30, 60, 10])
```

All the default values of weights/biases are initialized with random numbers, which can be found on line 35-39 in Network.py.

After the network is instantiated, we will train the network using:

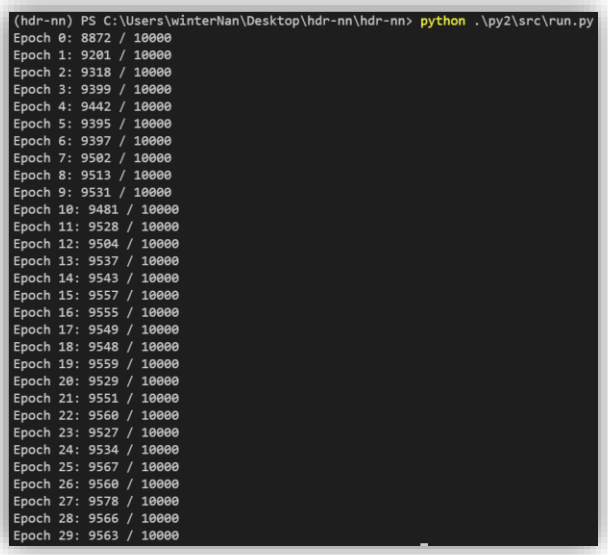
```
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

As the code shows, the training uses stochastic gradient descent (SGD), with training_data (50,000 images) as input picture pool, and test_data (another 10,000 images) as test image pool. Those numbers are specified by the mnist training set (line 68 of mnist_loader.py), rather than parameterized in the python code.

In the above training, we train the network with 30 epochs, with each epoch consists of 10 images. The default learning rate (η) is set to 3.0 empirically. We can apply the same methodology to train our new network which contains one additional hidden layer with 60 neurons.

```
net = network.Network([784, 30, 60, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Run run.py and the below are what I got from my system,



It seems that the ultimate accuracy is increased from ~94.8% to ~95.6%, which is to our expectation since we have added an additional layer. However, note that the size of the network is also increased dramatically which may render the accuracy harvest hard to achieve for FPGA.

@TODO. You can also experiment with increasing/adding more layers into the network and you will observe that the accuracy will drop instead of increasing!

4.2, network.py

The network.py implements the hdr-nn. It contains the following functions.

- __init__, which is the “constructor” of the network initializing all weights/biases. Note how __init__ creates arrays for weights and biases.
  @TODO. What are the dimensions of self.weights and self.biases?
- sigmoid(z), computes sigmoid function of input vector z. Note that input z is a vector or Numpy array, Numpy automatically applies the function sigmoid elementwise, that is, in vectorized form.
  @TODO. How to implement the sigmoid function in FPGA?
- feedforward(self, a), given an input a for the network, returns the corresponding output.
  @TODO. Figure out the dimensions of w, a, and b through out each loop iteration. Hint. You can use print x.shape to figure out the dimension of ndarray x.
- evaluate(self, test_data), computes the output vector x for input images in test_data, where y is the reference output read from test_data.
  @TODO. Figure out how evaluate calculates the output of the network.

The other functions such as SGD, update_mini_batch, etc, are used for training the network, which are not required to be implemented in the FPGA in the project. However, you are strongly recommended to read the code and understand the data flow even if you are not required to work with them in the project. **@TODO.** Understand the algorithm of the remaining functions.

4.3, mnist_loader.py

This is the loader we use to interpret test/training/invalidation images from the mnist data set. You are not required to work with them and please leave them untouched unless you know what to do.