Parallelized Convolutional Neural Network

Hong-Ye Hu and Leon Nguyen

July 20, 2017

Contents

I Abstract 1

II Introduction to Convolutional Neural Network (CNN) 1

II.1 Background ......................................................... 1

II.2 Architecture overview ........................................... 3

II.3 Supervised training on CNN ...................................... 6

III Single thread CNN performance profile 10

IV Multi-threads CNN performance profile 12
IV.1 Parallel method ................................................................. 12
IV.2 Implement parallel computing using numba CPU multi-thread ........ 13
IV.3 MNIST performance profile ................................................. 17

V Conclusion ........................................................................... 19

VI Appendix I: Creating a Singularity image to run Numba on Comet ... 21

VII Appendix II: Backpropagation ............................................. 23

VII Appendix III: Implement parallel computing using Numba-CUDA ... 24

IX Appendix IV: Code ................................................................. 25
I   Abstract

Neural networks are biologically inspired algorithms for machine learning. Nowadays, they have been successfully used in a wide range of fields from computer vision to speech recognition. In this paper, we implemented the convolutional neural network (CNN) to classify the MNIST data set. We implemented a single and multi-thread version of the CNN using Python with the Numba and threading packages. We profiled our single and multi-thread implementation of the CNN and we found that although a parallel implementation can improve training time by 6 to 7 times, there is a slowdown exhibited with higher number of threads.

II   Introduction to Convolutional Neural Network (CNN)

II.1   Background

The convolutional neural network (CNN) is one of the successful examples of neural networks. In 2012, a group of researchers in Geoffrey Hinton’s lab at the University of Toronto won the ImageNet challenge\cite{1} using a CNN. They trained a large, deep convolutional neural network to classify the 1.3 million high-resolution images in the LSVRC-2010 ImageNet training set into the 1000 different classes. On the test data, they achieved top-1 and top-5 error rates of 39.7% and 18.9% which is considerably better than the previous state-of-the-art results. In this paper, we create a CNN from
scratch using Python and NumPy to train with the MNIST\textsuperscript{1} dataset. The MNIST dataset contains examples of handwritten digits, each in a 28 by 28 pixel grayscale image. There are 10 classes of images for each of the 10 digits from ‘0’ to ‘9’. The training set includes 50,000 images, and the test set includes 10,000.

The basic element of any neural network are the neurons and the connections between them. Each neuron can be modeled using linear-nonlinear model\textsuperscript{2}. The input of each neuron first goes through a linear filter, and then the filtered result is put into a nonlinear function, which determines its spiking frequency. The CNN is inspired by the study of retinal ganglion cells. Today, one of the most popular theories about retinal ganglion cells is that each cell has a receptive field, or filter. When it receives an image (input), the receptive field (filter) will compare the similarity between the filter and the image. In the CNN, this is modeled by the convolution between filters and images, which is a linear operation. Then the filtered output goes into a nonlinear function, such as \textit{softmax}. Unlike a real neuron, the \textit{neuron} in neural network produces a real number output, instead of a spike.

\textsuperscript{1}http://yann.lecun.com/exdb/mnist/
II.2 Architecture overview

Figure II.1: An example architecture of CNN. The main types of layers are convolutional layer, pooling layer, and fully-connected layer (sometimes people separate RELU/activation layer from convolutional layer).

Fig. II.1 shows an example of the typical CNN architecture. The main layers of CNN are the convolutional layer (CONV), pooling (sub-sampling) layer (POOL), and fully-connected layer (FC). The inputs and outputs between each layer are called ‘activations’.

A typical CNN starts with a series of convolutional and pooling layers before ending with several fully-connected layers. The purpose of the convolutional layer is to filter the input activation with a filter bank that represents important image features, such as lines and patterns. The pooling layer then subsamples the output activation from the convolutional layer to compress the representation of the original input activation.

The fully-connected layer unravels its input activation into a one-dimensional array and it implements a standard neural network, which is a matrix $w \in \mathbb{R}^{m \times n}$ of weights that transforms the input activation from $\mathbb{R}^m$ to $\mathbb{R}^n$. The final output layer is a fully-connected layer that maps its input activations into a number of outputs that represents the number of classes in the dataset. Typically, this final layer uses a softmax nonlinear
operation on the output to transform the values to the range \([0, 1]\), giving the output a probabilistic interpretation as to which class the original input image belongs. Because of this, the final fully-connected layer is typically called the softmax layer. For MNIST, there are 10 classes, so the final layer will have 10 outputs.

Sometimes in the literature one may see an additional layer called the ReLU or activation layer. These names refer to a layer that performs a nonlinear operation on an activation. The purpose of doing this is to perturb the activations in between layers so that layers cannot be trivially combined by linearity, since convolution and matrix products are linear operations. In our implementation, we include the nonlinear activation at the end of the convolutional and fully-connected layers. The output activation from the pooling layer does not need to undergo a nonlinear operation because the pooling operation is already nonlinear.

In the CNN that we created, the user can define the architecture of the CNN with an arbitrary amount of layers, specifying the type of each layer. As an example of how to create a CNN, we use the CIFAR10 dataset, which has RGB images of dimension 32 by 32 and 10 different classes. We will have 1 convolutional layer, 1 pooling layer, 1 fully-connected layer, and 1 softmax (output) layer. For the convolutional layer, we have 12 filters each of dimension 3 by 3. The pooling layer will use the ‘max pooling’ technique with a downsampling factor of 2, which will downsample the activation by 2 by keeping the max value of each 2 by 2 block. The first fully connected layer will have 100 outputs, and the output fully-connected layer will have 10 outputs. The nonlinear
operation that we use is called the ReLU\(^2\) which is defined as \(\max(0, x)\). We can declare such a CNN using the following code.

```python
x = CNN(inputSize = (3, 32, 32), layers = ['C', 'P', 'F', 'S'],
        convFilters = [(3, 12, 1)], downsample = [2], fcSize = [100, 10])
```

This declares the CNN object to the variable \(x\), which keeps tracks of important parameters for each layer during the training of the CNN. Such parameters include the weights and biases for each convolutional and fully-connected layer.

- **inputSize** indicates a 3-channel image of dimension \([32 \times 32]\). For the MNIST dataset, this parameter would be \((1, 28, 28)\) because the images are grayscale.

- **layers** is a list that specifies the layers in the CNN. In this case, we have a convolutional (C), then a pooling (P), then a fully-connected (F), and then a softmax (S) layer.

- **convFilters** is a list of tuples that specifies the square dimension of the filter, the number of filters, and the stride (how many pixels the filter slides while convolution). We have 12 filters of dimension \([3 \times 3]\), and they move in steps of 1 during convolution.

- **downsample** is a list specifying the downsampling rate for each pooling layer. Here, we specify a rate of 2 for our only pooling layer.

- **fcSize** is a list specifying the output dimension after each fully-connected layer.

\(^2\)ReLU stands for rectified linear unit
The first fully-connected layer maps to 100 values, while second maps to 10.

The convolutional layer computes the 2D convolution between the input activation and each filter. As a reminder, the 2D convolution on an input matrix $A$, filter matrix $B$, and output $C$ is defined as

$$
C[m, n] = A[m, n] * B[m, n] = \sum_{u, v=-\infty}^{\infty} A[u, v]B[m - u, n - v].
$$

(II.1)

Because the input to the convolutional layer can be 3 dimensional, as is the case for an RGB image, each filter needs to match the dimensionality of the input. Per our example, if we have 12 filters, each filter will be a $[3 \times 3 \times 3]$ cube. Therefore, for a single filter, each step in the convolution will be computing the dot product of the filter with a $[3 \times 3 \times 3]$ block of the input. Because there are 12 filters, the output will be $[12 \times 32 \times 32]$. Our CNN class automatically computes the amount of zero-padding necessary for the input activation so that the output will have the same square dimensions. For more details, see Fig. II.2.

II.3 Supervised training on CNN

So far, we introduced the forward propagation of the CNN. In order for the CNN to learn the weights for the convolutional layer filters and the fully-connected layers, we need to train the CNN. Initially, these weights are drawn from a random distribution. When an image is first forward propagated through the CNN, the output that classifies
Figure II.2: An example of a convolutional layer. The input activation (blue) dimension is $[3 \times 5 \times 5]$, and we pad with 1 layer of zeros around the image. The filter (red) dimension is $[3 \times 3 \times 3]$, and number of filters are 2. The output (green) dimension will be $[2 \times 3 \times 3]$.\[5]
Figure II.3: The same example as in Fig. II.2, but now the stride is 2, which means we move the filters by 2 pixels during convolution. Because of this, we need an extra layer of zero-padding to maintain the square dimension of the input activation.[5]

the image will most likely be incorrect. The goal is to update the weights so that the output correctly classifies the input image. This learning process is called supervised machine learning because the CNN has access to the true labels for each image. In neural networks, we make iterative updates to the weights based off of the error in the output. This error is defined by a loss function. The loss function we use is the cross-entropy loss:

$$E = \sum_i -y^i \log h(x^{(i)}; \theta),$$

where $h(x^{(i)}; \theta)$ is output given input $x^{(i)}$ and network parameters $\theta$, and $y^i$ is the true label of the input. To minimize this loss, we can iterate through the training images.
and update all of the parameters $\theta$. In our case, $\theta$ constitutes the weights in the CNN. After training an image, we can update the weights by the following learning rule:

$$
\theta_i = \theta_i - \alpha \Delta \theta_i, \quad (II.3)
$$

where $\alpha$ is the learning rate to specify how much the weights update. Typically this is set to a small value, such as 0.01. However, more sophisticated implementations use an adaptive learning rate.

The trick is how to use the output error $E$ to update our weights $\theta$. There is an algorithm called ‘backpropagation’, which is shorthand for backward propagation. The idea is that after forward propagating the input image $x^i$ through the network, we arrive at the error $E$. We then need to backward propagate $E$ through the CNN to find the error of our loss function with respect to the weights. We can use this value as an estimate for $\Delta \theta_i$. This process of updating parameters by error gradients is called stochastic gradient descent (SGD). For details on the backpropagation algorithm, please see the appendix Section VII.

After training the CNN, the filters will appear to resemble important features, as shown in Fig. II.4, which are the final filters from I. Some of the filters search for edges, and some others search for colors.
Figure II.4: An example of filters after the training. After the training, we can clearly see the filters are looking for edges and curves, just like what retinal ganglion cell does. Results are from [1].

III Single thread CNN performance profile

Figure III.1: A pie chart showing the percentage of time used by the three main layers (convolutional, pooling, fully-connected) in the CNN while training the MNIST dataset with a single thread. The total time spent training was 4650.09 seconds. The ‘other’ category indicates operations outside of the three main layers, such as the update of the network parameters during SGD.
Throughout training, we used a CNN with 1 convolutional layer, 1 pooling layer, and 3 fully-connected layers. The number of convolution filters in the convolutional layer varied throughout our experiments. In the single thread implementation of the CNN, which is just the serial code, we used 6 filters. We trained on the MNIST dataset, which has 50,000 training images. We trained 1 SGD epoch with 400 minibatches, each with 125 training images, to have the CNN see each training image once. On the Comet shared node, we used the cProfile package in Python to clock the training time at 4650.09 seconds (almost 1.5 hours). Testing on the test set of 10,000 images, our trained CNN achieved a 81.3% classification accuracy. Although the state-of-the-art CNNs achieve above 99% accuracy, they have more sophisticated learning rules, use deeper networks, and train for multiple epochs. For example, the network in [1] trained for two weeks before testing. We did not focus on fine-tuning our CNN to achieve high accuracy because our focus is on parallelization. We notice in Figure III.1 that 82% of the computation is spent in the convolutional layer. Therefore, we want to focus on parallelizing the convolution operations that are used in the forward and backward propagation.
IV Multi-threads CNN performance profile

IV.1 Parallel method

From Chapter [III] we notice that the bottleneck of the program is the convolution operation, so we decided to parallelize our convolution function. As shown in Fig. [IV.1] we see that each dot product in the convolution is independent from one another. Therefore, we can parallelize by having each thread perform a different section of dot products. The pseudo-code is below:

```python
1 def CONV(output_layer, input_layer, filters):
2   create n threads
3   for each thread:
4     do convolution(filters[thread_id], input_layer)
5     post_layer[thread_id] <-- result
6 end
```

Although matrix multiplication is not the bottleneck of the program, it can be also parallelized. The pseudo-code is below:

```python
1 def Matrix_multiplication(C, A, B):
2 #calculate C = A * B
```
row_num = row(C)
col_num = column(C)
create (row_num*col_num) threads:
for each thread:
i = get_grid_id_x, j = get_grid_id_y
C[i, j] = sum(A[i, k]*B[k, j])
return

We used the Python multithreading package to implement the convolution operation, and Numba CUDA package to implement the matrix multiplication. In the following two subsections, we are going to discuss the parallelization in details.

IV.2 Implement parallel computing using numba CPU multithread

Python contains a default package named threading, which allows for CPU parallel computing. Although it supports multithreading calculations, Python is not designed for this and it has global interpreter lock (GIL), which prevents multiple threads from accessing multiple cores. To truly implement parallel computing in Python, we used the Numba package to unlock the GIL via the nogil parameter. Numba can also support nopython mode for certain operations, which compiles the code to be faster. However, in nopython mode, only basic NumPy array operations are supported. To activate the nogil and nopython mode, we need to add the decorator before the definition of the function. For example:

```python
from numba import jit
@jit(nogil = True, nopython = True)
def f(x, y):
```
Because Comet does not include the Numba package in its Python distribution, we use Singularity to create an image that contains the Numba package. For details about how to build an image using singularity and run it on Comet, please go to Chapter VI.

First we define `nthreads`, the number of threads that we want to use in parallel computing. The output `postlayer` of the convolutional layer will be a three dimensional array. One element in the output can be accessed as `postlayer[k, i, j]`. We need to separate all of the elements into `n` chunks, and we need to create chunk lists that contains the elements in the output for each thread to compute. For example, suppose the dimension of the output is `[1, 2, 3]`, and we are using 3 threads. The lists of coordinates are:

- `i_list = [[0, 1], [0, 1], [0, 1]]`
- `j_list = [[0, 0], [1, 1], [2, 2]]`
- `k_list = [[0, 0], [0, 0], [0, 0]]`

The p-th thread will receive the parameters `i_list[p], j_list[p],` and `k_list[p]`. We then use the `threading` package to make a list of jobs for each thread.

```python
for p in range(nthreads):
    thread = threading.Thread(target = function_name, args = *args)
    threads.append(thread)
```

Then we just need to run the jobs in the list `threads` and wait until every thread finishes the job to join into the serial code again. The code is listed below:

```python
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```
We tested this block of code on a Macbook Pro that has 8 cores. Our input is a 1 dimensional array with square dimensions $[1024 \times 1024]$, and we use 1 $[3 \times 3]$ filter. The table below shows our results. The first entry using 1 thread and a regular Python with NumPy implementation took 6600 ms. The next entry shows an improvement from just using the `nopython` compilation option of Numba. The rest of the entries show the result from implementing the multithreading strategy outlined above. Fig. IV.2 plots the results from the table.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>running time (ms)</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6600</td>
<td>python</td>
</tr>
<tr>
<td>1</td>
<td>220</td>
<td>@jit nopython = TRUE</td>
</tr>
<tr>
<td>1</td>
<td>215</td>
<td>multithread</td>
</tr>
<tr>
<td>4</td>
<td>198</td>
<td>multithread</td>
</tr>
<tr>
<td>6</td>
<td>167</td>
<td>multithread</td>
</tr>
<tr>
<td>8</td>
<td>151</td>
<td>multithread</td>
</tr>
</tbody>
</table>

Figure IV.2: Running time profile of CONV layer: The dimension of input image is 1024 by 1024. Only one 3 by 3 filter is tested.
The speedup of parallel computing is not guaranteed for any situation. If the size of the input image is very small, then the overhead time will overcome the speed up of multithread calculation. As an example, we also test when the input is a 1 dimensional array with square dimensions $[28 \times 28]$, and we still use 1 $[3 \times 3]$ filter. The table below shows the result. We can clearly see that as the number of threads go up, the running time also increases. This is because most of the time is spent on setting up the multithread environment and preparing the parameters for threads.

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>running time($\mu s$)</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4780</td>
<td>python</td>
</tr>
<tr>
<td>1</td>
<td>297</td>
<td>@jit nopython = TRUE</td>
</tr>
<tr>
<td>1</td>
<td>445</td>
<td>multithread</td>
</tr>
<tr>
<td>4</td>
<td>564</td>
<td>multithread</td>
</tr>
<tr>
<td>6</td>
<td>748</td>
<td>multithread</td>
</tr>
<tr>
<td>8</td>
<td>945</td>
<td>multithread</td>
</tr>
</tbody>
</table>

Figure IV.3: Running time profile of CONV layer: The dimension of input image is 28 by 28. Only one 3 by 3 filter is tested.
IV.3 MNIST performance profile

Figure IV.4: A plot showing the training time as the number of threads are varied. The blue line indicates the total time, and the green line indicates the time spent in the convolution layer.

Training the 50,000 MNIST images as we vary the number of threads, we find that although performance improves from the single-thread case, the time taken to train increases with the number of threads, as shown in Fig. IV.4. As shown in the previous section, this is because the dimension of the input images is not large enough to take advantage of parallel computation. The overhead in creating the threads does not justify the amount of computation to be done, so performance suffers as we increase the number of threads.

To see if we can justify the use of parallel computing, we upscaled our input images by a factor of 36 from $[28 \times 28]$ to $[1008 \times 1008]$. We did not retrain the whole training dataset.
Instead, we just trained on 20 images as a proof-of-concept. We see in Fig. IV.5 that we only see improvement with 4 threads before the running time grows. Many datasets do not have very large images, so we can conclude that a parallel implementation of the convolution operation might not be justifiable if the network is small. Perhaps a more reasonable parallelization is to parallelize the training of minibatches, which is done by most groups, including Hinton’s \[1\].

![Figure IV.5: A plot showing the training time as the number of threads are varied. We upscale the image from 28 by 28 to 1008 by 1008 to prove our reasoning. We see that for large training input we can gain speed up as up to 4 threads. As number of threads increases, the overhead preparing time overcome the speed gain.](image)

Another test is to maintain the same number of threads while increasing the size of the CNN. We used 2 threads while changing the number of convolutional filters from 2 to 14 in steps of 2, and we plot the results in IV.6. With values that fall off the increasing
trend at 8 and 12 number of filters, we believe that memory access problems on the shared node might have affected the runtime. Despite this, we see that the running time scales well as we increase the number of filters.

Figure IV.6: A plot showing the training time as the number of convolutional filters are varied. We fix the number of threads to 2, and the test was run on the shared node.

V Conclusion

In this project, we used NumPy to implement the serial convolutional neural network (CNN), and trained it using MNIST dataset. We profiled the serial code using cProfile to find that the bottleneck is the convolution operation, which takes around 80% of the running time. We used Numba and threading package in Python to parallelize the convolution operation. For that single parallelized block of code, we profiled the
performance. It shows great speed up when input image is large, such as 1024 by 1024 pixels. But for small image, like 28 by 28 pixels, as the number of threads increase, the running time also increases. This is because the overhead of setting up threads overcomes the performance gain for such a small input. For training parallelized CNN using MNIST dataset, we found that the images in common datasets are too small to take advantage of parallelization. A more trivial parallelization that is recommended is to parallelize the training of minibatches during SGD so that each thread is training a unique minibatch. As noted, most research groups in neural networks implement this paradigm of parallelization instead of trying to parallelize the convolution operation.

Despite the outcome of the project, this was a good opportunity to learn about the parallel tools in Numba, which we did not know about before this project. One of the downfalls of Python is GIL, which limits the otherwise powerful scripting language to only operate in a single-threaded mode. With Numba, it is easy to parallelize code on today’s multi-core personal computers. We also learned a lot about how to operate on an HPC, from basic aspects such as SLURM commands to more involved aspects such as setting up a Singularity image to run a custom environment.
VI Appendix I: Creating a Singularity image to run Numba on Comet

To use programs such as Numba that are not available on Comet, we need to configure a local environment that can run such programs and then use Singularity\(^3\) to access the environment on Comet. In this section, we detail the method to use Singularity for this purpose.

On macOS, install Vagrant\(^4\) and VirtualBox\(^5\). Then execute the following in Terminal to install Singularity in an Ubuntu VirtualBox. On Linux, just execute the lines within the `EOF` characters.

```
vagrant init ubuntu/trusty64
vagrant up

vagrant ssh -c /bin/sh <<EOF
    sudo apt-get update
    sudo apt-get -y install build-essential curl git sudo man vim autoconf libtool
git clone https://github.com/singularityware/singularity.git
cd singularity
    ./autogen.sh
```

\(^3\)http://singularity.lbl.gov/
\(^4\)https://www.vagrantup.com/
\(^5\)https://www.virtualbox.org/
./configure --prefix=/usr/local

make

sudo make install

EOF

On macOS, we can now log into the Ubuntu VirtualBox via ssh. On Linux, this is not necessary.

vagrant ssh

After logging in, create a Singularity image of up to 4 GB.

sudo singularity create -s 4096 /tmp/Ubuntu.img

Install the debootstrap program to install the tools necessary in your Singularity image via the singularity bootstrap command.

sudo apt-get -y install debootstrap

sudo singularity bootstrap /tmp/Ubuntu.img ubuntu.def

The ubuntu.def file installs basic programs into the Singularity image such as vim, build-essential, wget, and git. It also installs python along with the python-pip package manager, which we use to install Numba. Later on, we can create a new definition file and call singularity bootstrap in the same manner to install additional
programs into the Singularity image. Our `ubuntu.def` file is included in our code for reference on formatting.

The paradigm is that the maintainer of the Singularity image can edit their image on their personal computer with full admin rights. Once the maintainer distributes the image to Comet, users will be able to access the image and its amenities.

After bootstrapping the Singularity image, transfer it to the oasis filesystem via scp. The Singularity image will automatically create a binding to the `/home` directory on its host computer, so files on the Comet user filesystem can be accessed by the Singularity image.

When submitting a job on Comet via `sbatch`, the following can be placed in the `.sb` file to run Python code.

```bash
module load singularity
IMAGE=/oasis/scratch/comet/$USER/temp_project/Ubuntu.img
singularity exec $IMAGE python /home/$USER/your_code.py
```

### VII Appendix II: Backpropagation

In forward propagation, each layer $l$ has an output activation $a_l = \sigma(f(a_{l-1}, w_l) + b_l)$, where $\sigma(x)$ is the nonlinear operation, $f(x)$ is the linear operation such as a convolution or a matrix product, $w_l$ are the weights for layer $l$, and $b_l$ are the biases for layer $l$. We can imagine $a_l$ as a column vector, and we can define $z_l = f(a_{l-1}, w_l)$. 

For a fully-connected layer, the four backpropagation equations are:

\[
\begin{align*}
\delta_L &= \nabla_a L \odot \sigma'(z_L) \\
\delta_l &= w_{l+1}^T \delta_{l+1} \odot \sigma'(z_l) \\
\frac{\partial L}{\partial b_l} &= \delta_l \\
\frac{\partial L}{\partial w_l} &= a_{l-1} \delta_l^T e
\end{align*}
\] (VII.1)

Note that \(\odot\) indicates the Hadamard product. The first equation finds the error in the output layer \(L\). Using this error, we see in the second equation that we can evaluate the output error of the previous layer using the error in the output layer. This backpropagation of the error can allow for the calculation of the bias and weight error derivatives in the third and fourth equations, which are used in the SGD updates.

\section{Appendix III: Implement parallel computing using Numba-CUDA}

CUDA has a two dimensional physical thread arrangement so it would be very natural to parallelize matrix multiplication using CUDA. By definition, the matrix multiplication is

\[
C[i, j] = \sum_k A[i, k] * B[k, j]
\] (VIII.1)
The grid is a two dimensional object in the GPU, so we can use each thread in the grid to calculate the matrix element. The Numba package also provides a decorator \@cuda.jit to run parallel GPU code on NVIDIA GPUs. The code is listed below:

```python
@cuda.jit
def matmul(A, B, C):
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        temp = 0
        for k in range(A.shape[1]):
            temp += A[i, k] * B[k, j]
        C[i, j] = temp
```

We did not get a chance to implement this into our final code. In the future, we will work on profiling this block of code.

IX Appendix IV: Code

The file `CNN_FUNCTIONS.py` contains the CNN class along with the parallelized convolution codes.

The file `mnist.py` loads the MNIST dataset and trains a CNN on the entire MNIST training set and then tests the trained CNN on the MNIST testing set. At the end, it reports the accuracy and saves some parameters.

The file `mnist.pkl.gz` contains the MNIST dataset. It is read by `mnist.py`.

The file `mnistCNN.sb` contains the SLURM run file that opens the Singularity image and then runs `mnist.py`. It runs on a shared node with 4 cores.

The file `ubuntu.def` is the definition file used to bootstrap the Ubuntu image on Sin-
gularity. The Ubuntu image is located at:

/oasis/scratch/comet/leonn/temp_project/Ubuntu.img.

References


