5/10/2017

Eye for the blind: GPU & CNN

ECEN 5593 Final Project Report

Rahul Yamasani & Mounika Reddy Edula
UNIVERSITY OF COLORADO, BOULDER

INDEX

1 TABLE OF CONTENTS

2 VISION

According to World Health Organization (WHO), 285 million people are visually impaired worldwide; 39 million of them are blind. Thanks to Braille, which helps in visualizing the world. However, there are limitations on the same. How about reading newspaper? What about recognizing numbers that are everywhere?

Can't we design something that can help them to visualize by some other means. This motivated us to design an "EYE FOR THE BLIND".

3 INTRODUCTION

Deep Learning powered computer vision has the potential to make our vision to reality. Imagine a visual impaired person can read a newspaper by placing it in front of camera. The cameras are executing computer vision algorithms for extraction and recognition powered by trained convolutional neural networks.

We used NVIDIA Jetson TX1 as the hardware accelerator for better performance. Remote access of the system is through socket communication and gives response in an interactive human speech.

In the initial sections, we discussed on Machine learning and convolutional neural networks. Then we went on to explain Why Caffe Framework for our application. This is followed by Hardware Block Diagram and Software Architecture. The later sections describe the performance analysis of GPU vs CPU. The final section is conclusion followed by Future scope of our project

4 MACHINE LEARNING:

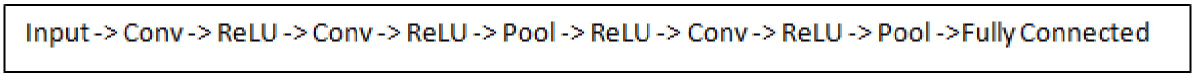
Machine learning is a type of artificial intelligence that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can change when exposed to new data.

Machine learning uses that data to detect patterns in data and adjust program actions accordingly. Machine learning algorithms are often categorized as being supervised or unsupervised. Supervised algorithms can apply what has been learned in the past to new data. Unsupervised algorithms can draw inferences from datasets.

4.1 CNN:

CNNs do take a biological inspiration from the visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. For example, some neurons fired when exposed to vertical edges and some when shown horizontal or diagonal edges when they cross the threshold.

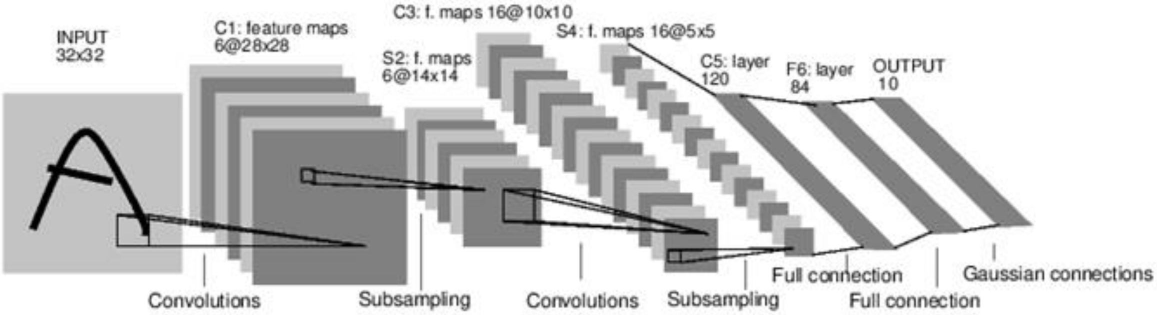
Computer considers image as an array of pixels. The input for CNN is a 32 x 32 x 3 array of pixel values. Now, the best way to explain a conv layer is to imagine a flashlight that is shining over the top left of the image. Let's say that the light this flashlight shines covers a 5 x 5 area. And now, let's imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a **filter**(or sometimes referred to as a **neuron** or a **kernel**) and the region that it is shining over is called the **receptive field**. Now this filter is also an array of numbers (the numbers are called **weights** or **parameters**). let's take the first position the filter is in for example. It would be the top left corner. As the filter is sliding, or **convolving**, around the input image, it is multiplying the values in the filter with the original pixel values of the image (aka computing **element wise multiplications**). These multiplications are all summed up (mathematically speaking, this would be 75 multiplications in total). So now you have a single number. Remember, this number is just representative of when the filter is at the top left of the image. Now, we repeat this process for every location on the input volume. (Next step would be moving the filter to the right by 1 unit, then right again by 1, and so on). Every unique location on the input volume produces a number. After sliding the filter over all the locations, you will find out that what you're left with is a 28 x 28 x 1 array of numbers, which we call an **activation map** or **feature map**. The reason you get a 28 x 28 array is that there are 784 different locations that a 5 x 5 filter can fit on a 32 x 32 input image. These 784 numbers are mapped to a 28 x 28 array.



Input -> Conv -> ReLU -> Conv -> ReLU -> Pool -> ReLU -> Conv -> ReLU -> Pool ->Fully Connected

The final result should be a 1D vector.But the matrix multiplication does not lead to a 1D vector..Pooling is a converting a matrix (2x2) to 1 pixel based on maximum value/average in the matrix. This is given as input to the next layers in the network.

Now that we can detect these high level features, the icing on the cake is attaching a **fully connected layer** to the end of the network. This layer basically takes an input volume (whatever the output is of the conv or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program has to choose from. For example, if you wanted a digit classification program, N would be 10 since there are 10 digits. Each number in this N dimensional vector represents the probability of a certain class. For example, if the resulting vector for a digit classification program is [0 .1 .1 .75 0 0 0 0 0 .05], then this represents a 10% probability that the image is a 1, a 10% probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9.



A Full Convolutional Neural Network (LeNet)

So backpropagation can be separated into 4 distinct sections, the forward pass, the loss function, the backward pass, and the weight update. During the **forward pass**, you take a training image which as we remember is a 32 x 32 x 3 array of numbers and pass it through the whole network. On our first training example, since all of the weights or filter values were randomly initialized, the output will probably be something like [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1], basically an output that doesn't give preference to any number in particular. The network, with its current weights, isn't able to look for those low level features or thus isn't able to make any reasonable conclusion about what the classification might be. This goes to the **loss function** part of backpropagation. Remember that what we are using right now is training data. This data has both an image and a label. Let's say for example that the first training image inputted was a 3. The label for the image would be [0 0 0 1 0 0 0 0 0 0]. A loss function can be defined in many different ways but a common one is MSE (mean squared error), which is ½ times (actual - predicted) squared.

As you can imagine, the loss will be extremely high for the first couple of training images. We want to get to a point where the predicted label (output of the ConvNet) is the same as the training label.Now what we want to do is perform a backward pass which is determining the weights which are responsible for the loss and updating the weights. The weights update depends on the learning rate that is chosen by the programmer. Higher learning rate the model takes less time to converge.

5 Why Caffe

There are numerous machine learning open source Frameworks available in the market. The dominant in the market are Tensor Flow, Caffe, Torch. The framework we choose has to support the MNIST dataset. Keeping this is mind we narrowed down to TensorFlow and Caffe. In addition to this we also have a constraint in the physical memory as we are using Nvidia Jetson TX1.

5.1 TensorFlow

The memory available in Tx1 is only 5GB after installation of CUDA and CuDNN. Tensorflow[] has some other additional requirements such as to Java runtime environment, bazel for building tools, datasets for training and testing. To install Tensor flow after all the dependencies are installed we need atleast 8GB to complete the installation part. We can use a SD card for swapping the memory but the projects main idea is about measuring the performance. If swap file is used for supporting the additional memory requirements, then the performance of GPU will be reduced. Realizing the fact, we choose Caffe over Tensorflow.

5.2 Caffe

Caffe[] is a well-known and widely used machine-vision library that ported Matlab's implementation of fast convolutional nets to C and C++. Caffe is not intended for other deep-learning applications such as text, sound or time series data. Caffe does not require any writing of code for training data. Caffe is what our application requires where our primary focus is on increasing the performance of GPU while extracting the numbers from a Sudoku puzzle image and forwarding that image to a trained model for number recognition. The requirements of Caffe on GPU are CUDA and protobuf which can be installed without the need of any swap memory.

All these extensive features has encouraged us to choose Caffe over Tensorflow.

5.3 Why OpenCV4Tegra

OpenCV4Tegra is a CPU and GPU accelerated version of the standard OpenCV library.

There are three versions of OpenCV that you can run on the Jetson:

1. "Regular" OpenCV
2. OpenCV with GPU enhancements
3. OpenCV4Tegra with both CPU and GPU enhancements

There are two proprietary patented algorithms, **SIFT** and **SURF**, which exist in opencv-nonfree. Because these are patented, NVIDIA does not include them in their distribution of OpenCV4Tegra. Therefore, if your code does not use SIFT or SURF, then you can use OpenCV4Tegra and get the best performance.

6 CAFFE ARCHITECTURE

Modules present in Caffe are

1)Blobs

2)Layers

3)Nets

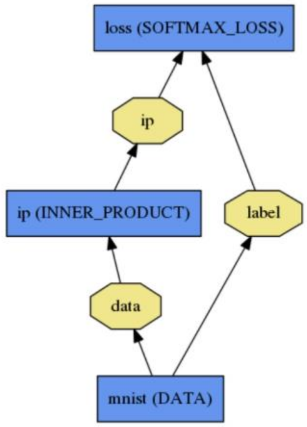
4)Solvers

5)Loss

6.1 BLOBS:

Deep networks are inter connected layers that works on chunks of data. The network defines the entire model bottom-to-top from input data to loss. Caffe stores communicates and manipulates the information as blobs, N dimensional array stored in a Contiguous fashion, standard array and unified memory interface for the framework. Blob under the hood provides synchronization capability between the CPU and GPU. Blobs conceal the computational and mental overhead of mixed CPU/GPU operation by synchronizing from the CPU host to the GPU device as needed. Memory on the host and device is allocated on the demand for efficient memory usage. Blob stores 2 chunks of memories, data and difference. The former is the normal data that we passalong and the latter is the gradient computed by network. The synchronization between CPU and GPU memories is due to this architecture where data is transferred when there is change not all the time.

6.2 LAYERS:



Layers convolve filters,pools which take blob i.e data and give to the next layer. Layers have two key responsibilities for the operation of the network as a whole: a forward pass that takes the inputs and produces the outputs, and a backward pass that takes the gradient with respect to the output, and computes the gradients with respect to the parameters and to the inputs, which are in turn back-propagated to earlier layers. These passes are simply the composition of each layer's forward and backward.

6.3 NETS:

The net is a set of layers connected in a computation graph. Net is a set of layers and their connections in a plaintext modeling language. Model initialization does two things creating the blobs and layers and also validation for the correctness of the overall network architecture. After construction of the network it can run on either CPU or GPU.Layers come with corresponding CPU and GPU routines that produce identical results.

The models are defined in a plaintext protocol buffer schema.

The forward and backward passes are the essential computations of a net. The forward pass computes the output given the input for inference.In forward caffe composes the computation of each layer to compute the “function” represented by the model. The pass goes from bottom to top. The backward pass computes the gradient given the loss for learning.In backward caffe reverse-composes the gradient for each layer to compute the gradient of the whole model by automatic differentiation. This is backpropagation.This pass goes from top to bottom.

6.4 LOSS:

In machine learning learning is driven by the loss function. A loss function specifies the goal of learning by mapping parameter settings to a scalar value specifying the badness of these parameter settings. The goal is to find a setting of the weights that minimizes the loss function. The loss in caffe is computed by the Forward pass of the network. Each layer takes a set of input(bottom) blobs and produces a set of output(top) blobs. In a Softmax With Loss function the top blob is a scalar which averages the loss over the entire mini-batch.

6.5 SOLVER:

Solver optimizes a model by first calling forward to yield the output and loss,then calling backward to generate the gradient of the model, and then in uses the gradient in updating the weight that attempts to minimize the loss.

Well known available Solvers are :

Stochastic Gradient Descent (type: "SGD"),

AdaDelta (type: "AdaDelta"),

Adaptive Gradient (type: "AdaGrad"),

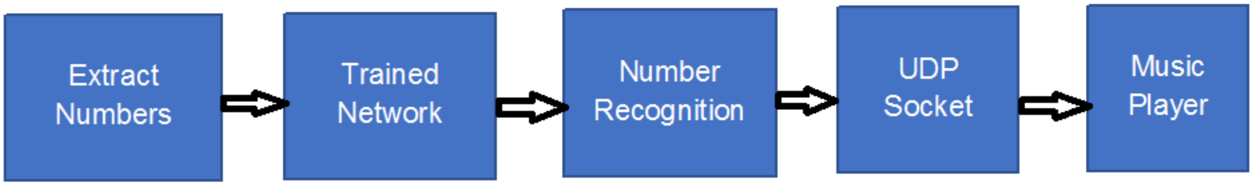
Adam (type: "Adam"),

Nesterov’s Accelerated Gradient (type: "Nesterov") and

RMSprop (type: "RMSProp")

The actual weight update is made by the solver then applied to the net parameters. First final gradient with respect to network weight is calculated, which are scaled by the learning rate and the update to subtract is stored in each parameter Blob’s difference field. Finally, diff parameter is updated in the blobs.

7 BLOCK DIAGRAM:



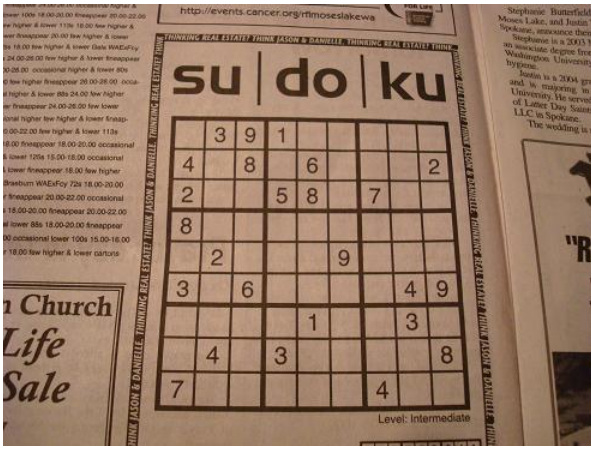
For number recognition, we should detect the numbers and feed them into a Trained neural network which will predict the numbers as the application is for blind people we need a voice feedback. But, we were able to play audio using HDMI but we faced difficulties in playing audio through USB sound card. So, we used UDP socket for sending the message to the computer and the digits recognized are converted into speech using gTTS (google text to speech) API.

8 SOFTWARE ARCHITECTURE:

8.1 STEP 1: FEATURE EXTRACTION (EXTRACT NUMBERS)

Our application needs to extract the numbers present in the sudoku puzzle and feed the images into the neural network.

The first thing is to identify the puzzle. In practical scenario, a picture of Sudoku from our daily newspaper wouldn't be perfect square as shown in the figure below.



Finally, we should be able to divide the Sudoku grid to 9*9. To simplify the problem, the first task we choose is to identify the border square of the given input image.

8.1.1 Segmenting the Sudoku

Read the image using opencv. We used adaptive Threshold to extract the edge of the images. This function accepts a gray scale image with the function.

1. Convert the image to grayscale with cvtColor. The adaptive threshold takes as first argument the input image,
2. Argument returns the binary image,
3. Argument is the non-zero value assigned to the pixel for which the condition is satisfied.
4. Next is the adaptive thresholding algorithm,
5. Final argument is the threshold type, the next argument is size of the pixel neighborhood and finally the last argument is a constant that is subtracted from the mean or weighted mean.

8.1.2 Code Description:

We have to find the connected contours in the image (finding Contours). These function returns a vector with the corners of the contours. We'll find the sudoku in this contours.

The assumption is that the sudoku puzzle has 4 sides and it's convex. Checking the number of the contour is equal to four and using the function isConvex to check if the square is convex. We obtain the possible candidates.

Then, we need to filter the candidates using the assumption: the sudoku puzzle is the biggest square in the image. We calculate the area of the possible contours (contourArea). The biggest square is the sudoku puzzle.

Now we have the sudoku puzzle segmented. We have got the corner points of the puzzle. It's currently not really usable for much. The sudoku puzzle is a bit distorted. It's necessary to correct the skewed perspective of the image. We need a way to mapping from the puzzle in the original picture back into a square. Where each corner of the sudoku puzzle corresponds to a corner on the a new image.

We use a transformation that will map one arbitrary 2D quadrilateral into another. We can use a perspective transformation

This perspective transformation maps a point (x,y)(x,y) in one quadrilateral into a new (X,Y)(X,Y) in another quadrilateral. These two equations contain 8 unknowns, but we have 8 values. (the corners xx and yy coordinates of the puzzle). Solving these equations gives us the a,b,c,d,e,f,g,ha,b,c,d,e,f,g,h which provide us with a mapping to get our puzzle out nice and straight.

The OpenCV function getperspectivetransform resolved the perspective transformation. Calculates a perspective transform from four pairs of the corresponding points, where the first parameter are the coordinates of quadrangle vertices in the source image and the second parameter are the coordinates of the corresponding quadrangle vertices in the destination image. To applies a perspective transformation to an image we used the function warpPerspective. This function transforms the source image using the specified matrix. We obtains the image below.

We need to sort the corner of the sudoku puzzle and then associate each point with the new image dimension. But, The dataset images have 16x16 pixels. The size of the new image will be 144x144, because we have to divide each row and col by 9 and we have to return the same size of the images.

8.1.3 Segmenting the numbers

extract_number(x,y):

After remapping, we can divide the puzzle into a 9×9 grid. Each cell in the grid will correspond (approximately) to a cell on the puzzle. The correspondence might not be perfect, but it should be good enough.

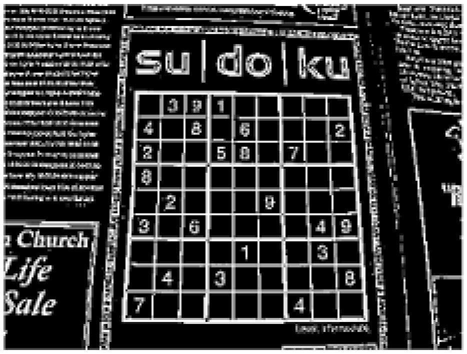
Now, we know the size of the bounding box. We also know the size of the image. We can easily center the image. We create a new image that's the same size as the original.

Sending images directly Neural Network is not a good idea. A little bit of work on the image can increase accuracy. Here, we'll center the actual digit in the image. The dataset has been made in such a way, all digits are centered by their bounding box.To remove the noise around the number (lines in general) we delete all the pixels outside the center of the image from a radius.

find_biggest_bounding_box():

Sometimes after remove the noise of the images can appear other particles close to the number. Because of this, we find the biggest bounding box in the small squared. We make the bounding box a little more bigger to improve the matching with the dataset.

Now we have to separate all the numbers from the grid. First we extract the numbers. If we have more active pixels than a threshold, we find the biggest bounding box in the image if the size of this bounding box is bigger than 0, we store this number in a matrix. In another case we store a matrix of zeros.



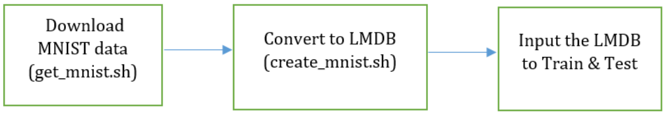
8.1.4 Inverting the image

The neural network is trained with MNIST data. To support the MNIST data it is necessary to invert the images.

8.2 STEP 2: TRAINING THE NEURAL NETWORK

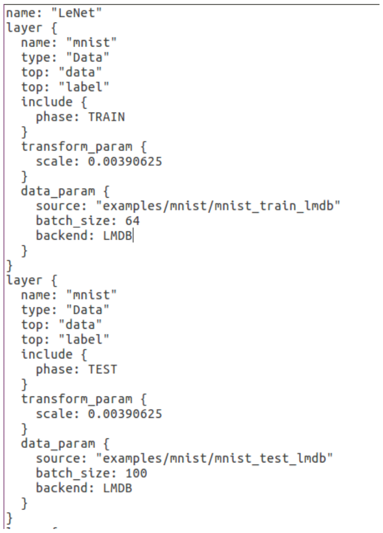
8.2.1 Dataset to train:

Caffe model uses google .prototxt. The input dataset required for training is obtained from MNIST which will be in the format of ubyte and is generated using get_mnist.sh. But, the data should be converted LMDB database (create_mnist.sh). This generates the LMDB database for Test and Train dataset.


Download MNIST data (get_mnist.sh)
Convert to LMDB (create_mnist.sh)
Input the LMDB to Train & Test

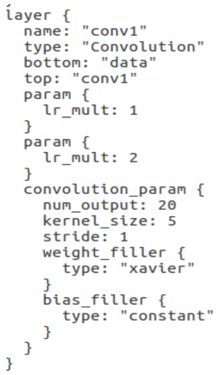
8.2.2 Layers

Input Layer → Train,Test



```
name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
```

We used a batch size of 64, and scale the incoming pixels so that they are in the range [0,1). Why 0.00390625? It is 1 divided by 256. And finally, this layer produces two blobs, one is the data blob, and one is the label blob. This is the input to the Convolution Layer

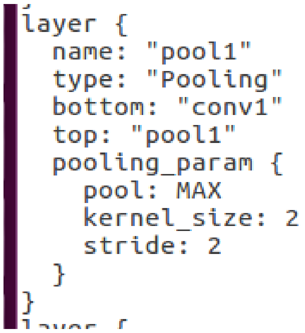


```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

The fillers allow us to randomly initialize the value of the weights and bias. For the weight filler, we will use the xavier algorithm that automatically determines the scale of initialization based on the number of input and output neurons. For the bias filler, we will simply initialize it as constant, with the default filling value 0.

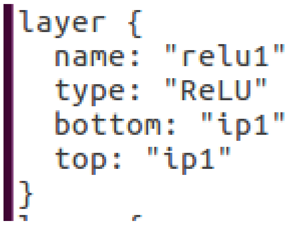
lr_mults are the learning rate adjustments for the layer's learnable parameters. In this case, we will set the weight learning rate to be the same as the learning rate given by the solver during runtime, and the bias learning rate to be twice as large as that - this usually leads to better convergence rates.

The input will be given as bottom and the output will be top because of caffe architecture. This will be fed to the pooling to reduce the image to 1D.



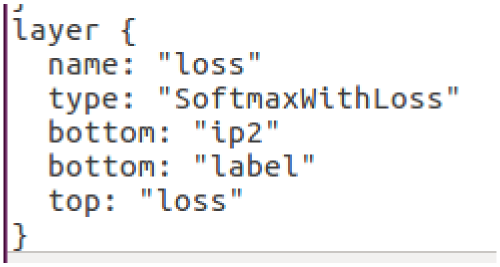
```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

Here the output of convolutional layer L1 as input(bottom) and the output will be pool1 which is fed into another convolutional layer. This says we will perform max pooling with a pool kernel size 2 and a stride of 2 (so no overlapping between neighboring pooling regions).



```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

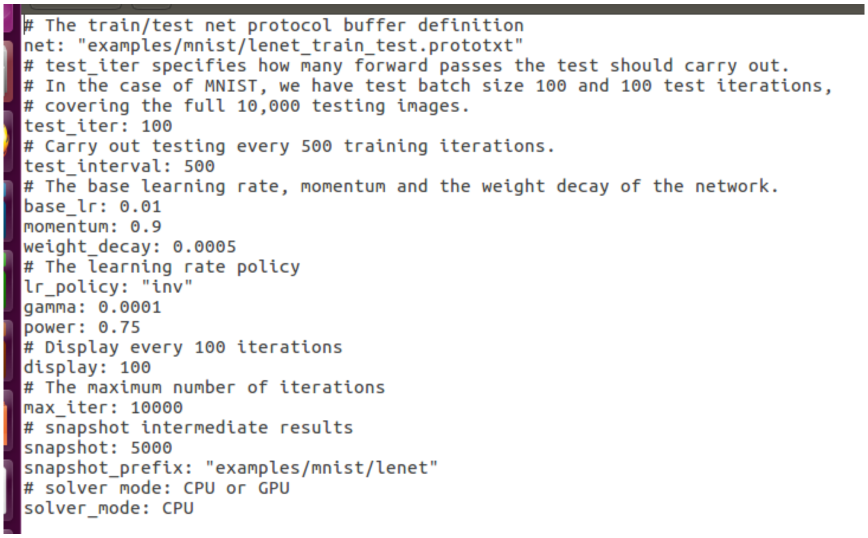
Since Relu is an element-wise operation we do in-place operations to save some memory. So both output and input are given same.



```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

The softmax_loss layer implements both the softmax and the multinomial logistic loss (that saves time and improves numerical stability). It takes two blobs, the first one being the prediction and the second one being the label provided by the data layer.. It does not produce any outputs - all it does is to compute the loss function value, report it when backpropagation starts, and initiates the gradient with respect to ip2.

8.2.3 Solver:



```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: CPU
```

Here we decided what is the number of iterations for training, testing, CPU/GPU, learning rate.

8.3 STEP:3 DEPLOY TRAINED MODEL (NUMBER RECOGNITION)

8.3.1 Problems faced and Incremental approach to victory:

8.3.1.1 Stage 1 – 0% Accuracy:

Initially we fed the images to our trained model and the results were disappointing as the accuracy was **0%**. The reason for this was the images were having the text in black color. As the MNIST data was trained for the handwritten images with the text being in white color. After realizing the images are inverted before feeding into the trained model. This increased the accuracy to **40%.**

8.3.1.2 Stage 2 – 40% Accuracy:

Even though the modificatoins in the above stage improved accuracy it was not quite enough for our model. The problem was digits like 8,0,3,7 were predicted in the same manner. This made us realize that the model can be trained in better way.

8.3.1.3 Stage 3 – 80% Accuracy:

There are many solvers available which helps in reducing the loss rate and we started with adam solver**.** The model was trained and deployed again with the modifications incorporated. The accuracy increased to **80%**. The model wasn't able to recognize the digits 3,7.

8.3.1.4 Stage 3 – 99.01% Accuracy:

After regress research to increase the accuracy of the model it is found that if the the mean of the dataset is subtracted from the images while training increases the accuracy. Mean for dataset is calculated using **compute_image_mean.cpp** which generated the mean of the dataset .binaryproto and is fed into the training model input layers(Train,Test).

```
mean_file: "/home/ubuntu/cats-dogs-tutorial/input/mean.binaryproto"
```

We trained the model again and deployed with the extracted images which increased the accuracy to **99.01%.** Now the model was able to predict all the digits perfectly.

8.4 STEP 4: UDP TRANSMISSION

As explained in the previous sections, the final aim is to play out the digits recognized. One option was to use the HDMI ouput but we choose to use UDP communication to transfer the digits recognized to remote desktop. This will enable the person to listen to digits without any connectivity with the GPU/CPU. On the receiving side we used gTTS (Google Text to Speech) library to speak out the digits received.

8.5 STEP 5: PLAY THE SOUND (DEMO)

For the purpose of demo we have extracted the digits 1 to 9 in order from Sudoku and transmitted to the remote desktop.

9 PERFORMANCE ANALYSIS:

The most interesting part of the project was Performance analysis. We made several comparisons to determine the performance of GPU over CPU.

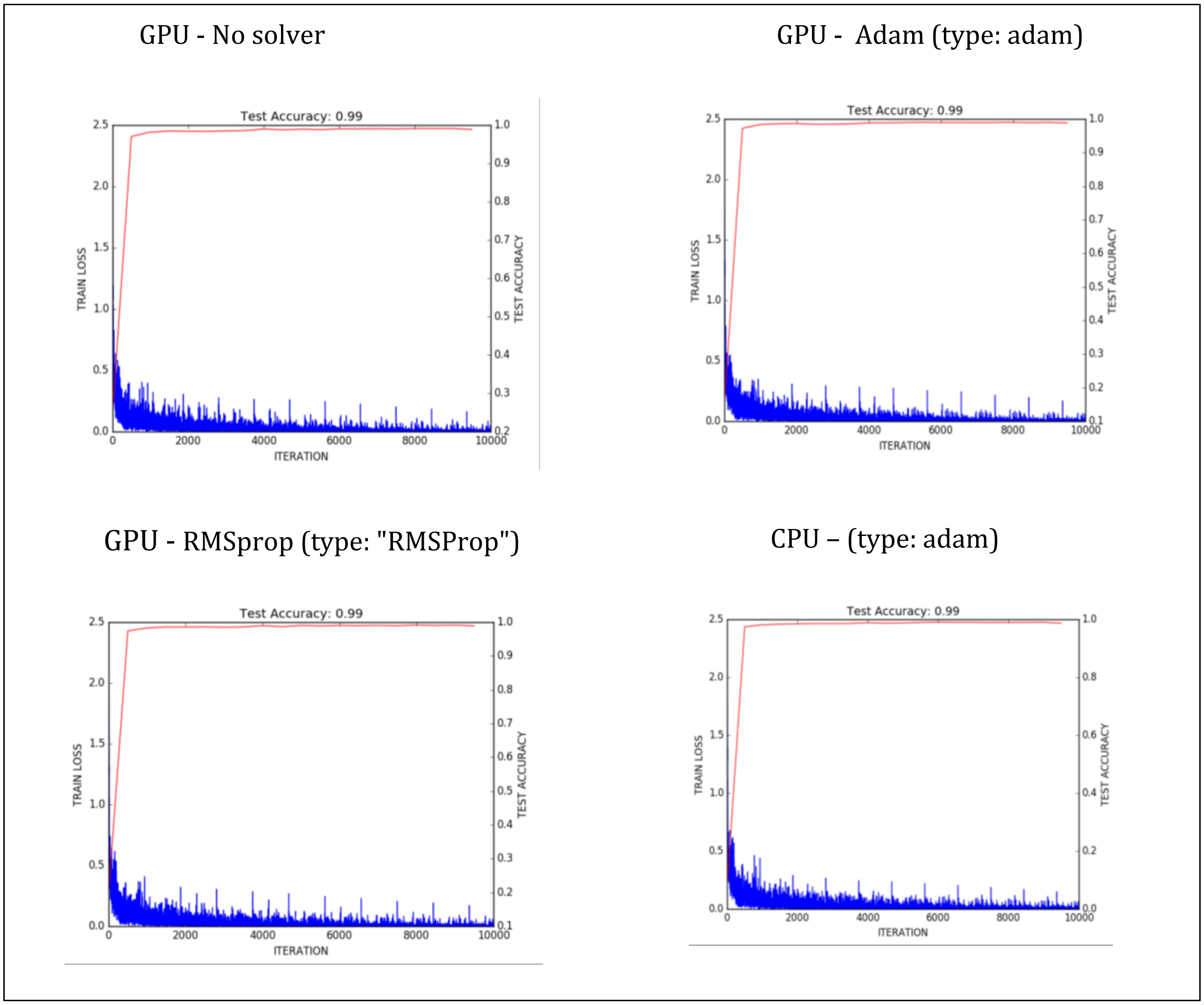
The first was comparison was for Test Accuracy and Train Loss with Iteration.

9.1 TEST ACCURACY/ TRAIN LOSS VS ITERATION

9.1.1 Test case:

Number of iterations for training: 10,000.
The batch size :100
Iterations for test: 500 iterations.

9.1.2 Graphs


GPU - No solver

GPU - Adam (type: adam)

GPU - RMSprop (type: "RMSProp")

CPU – (type: adam)

9.1.3 Observations:

Accuracy of the test data increases and training loss decreases with iterations. The maximum accuracy reached with or without different solvers is 99%. As explained in the previous section we made analysis using several algorithms namely SGD, ADAM, RMSPROP

One more interesting observation was the accuracy increased when the extracted digits are classified using the trained set derived from ADAM solver.

10 TIMING ANALYSIS

The next factor for performance analysis is time, how fast the model reaches the highest accuracy. In general notion, the solvers help in optimizing the loss and increasing the accuracy. So, when a solver is used then the time it will take to reach the maximum accuracy will be less when compared without using as solver. So we went onto perform the timing analysis for Test accuracy and Train loss with time.

10.1 TIME VS TEST ACCURACY

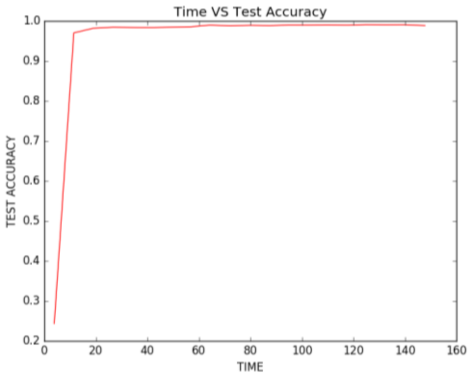
The following case illustrates comparisons made for Test Accuracy and Time for various algorithms on (GPU and CPU)

10.1.1 Test case:

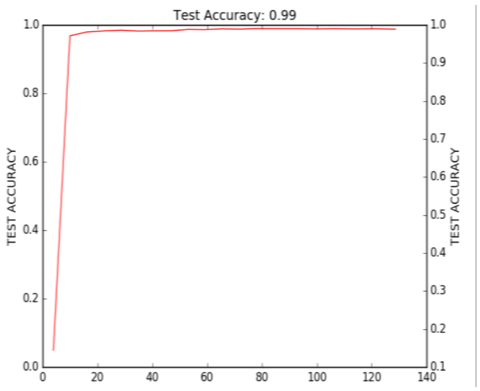
Number of iterations for training: 10,000.
The batch size :100
Iterations for test: 500 iterations.

10.1.2 Graphs

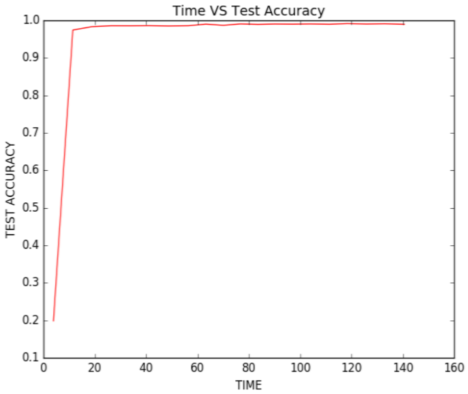
GPU - No solver



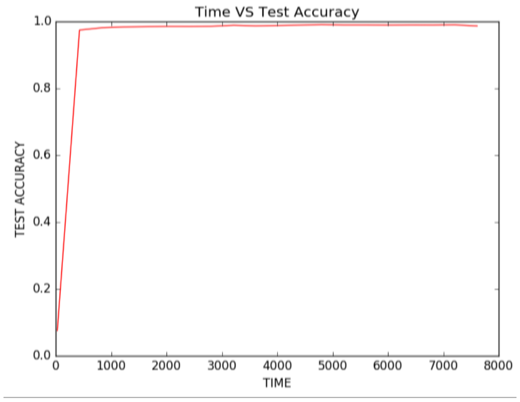
GPU - Adam (type: adam)



GPU - RMSprop (type: "RMSProp")



CPU – (type: adam)



10.1.3 Tabular Results

Algorithm	GPU (sec)	CPU (sec)
SGD	152.1	8162.54
ADAM	134.63	7945.34
RMSPROP	141.21	8044.67

10.1.4 Observations

From the above table it can be observed that time taken for CPU was very high when compared its GPU contemporaries. It can also be observed from the above table that ADAM solver takes least time to reach the maximum accuracy for the given test case.

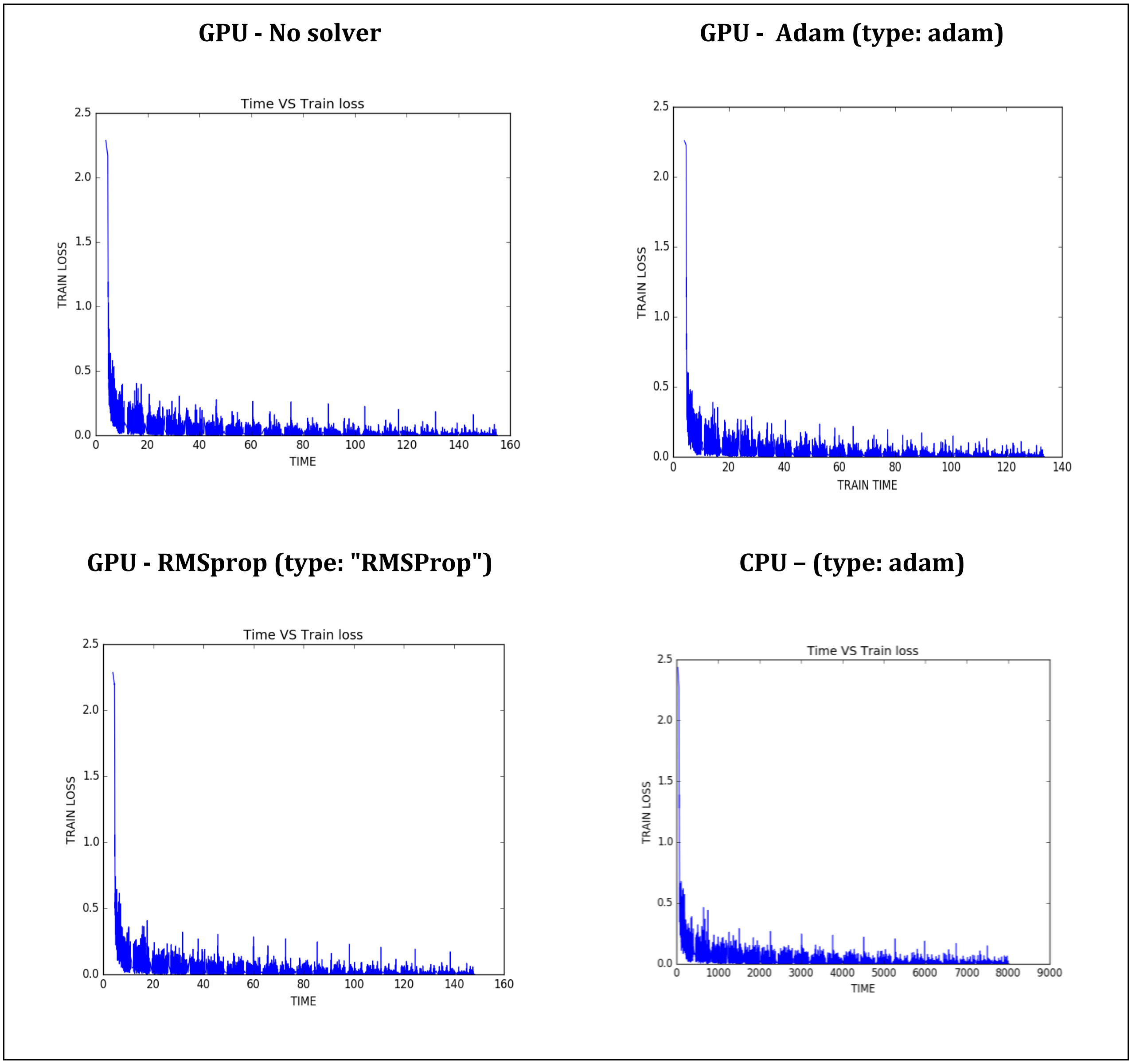
10.2 Time vs Train Loss

The following case illustrates comparisons made for Train loss and Time for various algorithms on (GPU and CPU)

10.2.1 Test case:

Number of iterations for training: 10,000.
The batch size :100
Iterations for test: 500 iterations.

10.2.2 Graphs



GPU - No solver

GPU - Adam (type: adam)

GPU - RMSprop (type: "RMSProp")

CPU – (type: adam)

The performance analysis for time can be considered for maximum accuracy or Train Loss. The train loss and test accuracy are inversely proportional the training loss will be decreasing with time and the accuracy will be increasing with time. In general notion, the solvers help in optimizing the loss and increasing the accuracy. So, when a solver is used then the time it takes to reach the minimum accuracy will be less when compared without using as solver.

10.2.3 Tabular Results

Algorithm	GPU (sec)	CPU (sec)
SGD	162.34	8192.54
ADAM	134.63	7934.34
RMSPROP	154.618	8044.67

10.2.4 Observations

From the above table it can be observed that time taken for CPU was very high when compared its GPU contemporaries. It can also be observed from the above table that ADAM solver takes least time to reach the maximum accuracy for the given test case.

11 CLASSIFIER TIMING ANALYSIS:

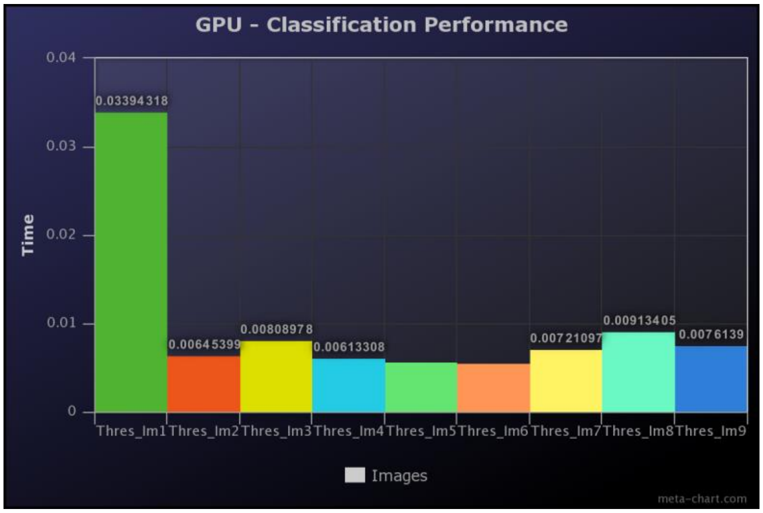
A final analysis was performed on time taken to classify each digit on GPU vs CPU. The following graphs illustrates the facts.

11.1 TEST CASE

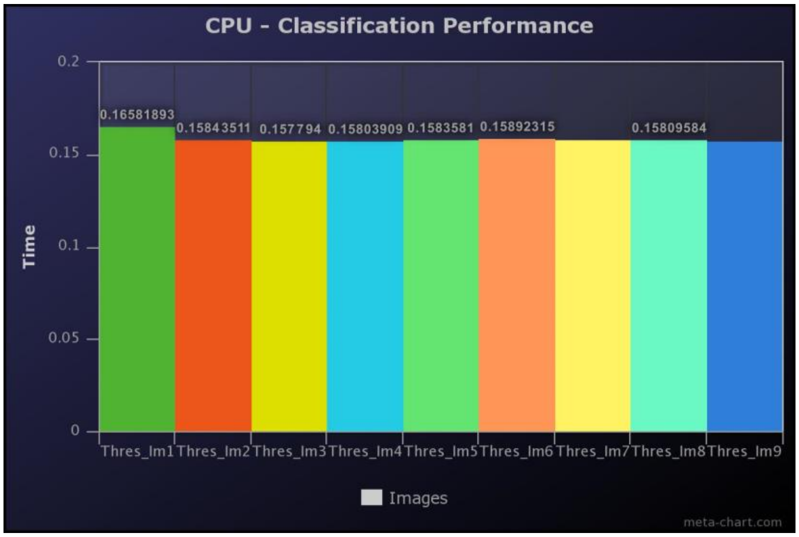
Images after Feature Extraction: 9
Trained model: AdamSolver.model

11.2 GRAPHS

11.2.1 GPU



11.2.2 CPU



11.3 Observations

GPU has better classifications timings that CPU for the same dataset.

12 Challenges Faced

Both of us are unfamiliar with the concepts of Machine leaning and Deep Neural Networks. Choosing a proper framework for the application was a difficult task. However, our research has greatly helped in choosing the right framework for the TX1 GPU architecture.

There have been numerous constraints on the memory as TX1 has only 16GB storage of which CUDA and its dependencies alone occupied 9 GB. Installing high end frameworks like Tensorflow was impractical as it would require a swap memory. However, choosing Caffe framework has solved the problem of memory.

The biggest challenge of our project was improving the accuracy of Trained neural networks. For this, we explored and tested various solver algorithms and determined ADAM solver to be the best. We further explored optimizing the training algorithm by subtracting the mean of the data set with every image while training.

13 Future scope

With the limited resources and time constraints we have only explored and implemented classification of numbers. This can be further developed into a system which can recognize alphabets. Recognizing alphabets would greatly help in simplifying the visual place recognition, transportation and many more day to day activities.

14 CONCLUSION

We have successfully implemented "EYE FOR THE BLIND" using GPU resources and Caffe framework. We used various solvers to optimize the timings and improve the accuracy. A part from performing analysis on the performance of GPU vs CPU, we also determined that ADAM solver is the best for the application.

15 ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to Mr. Dan Connors who gave us a wonderful opportunity and resources to do this dream project which also helped in doing a lot of research in Machine learning. We would also like to thank UC Berkeley on their wonderful documentation on open source library for machine learning.

16 REFERENCES

1. http://caffe.berkeleyvision.org/gathered/examples/mnist.html
2. https://github.com/9crk/caffe-mnist-test
3. http://adamsiembida.com/how-to-send-a-udp-packet-in-python/
4. https://pypi.python.org/pypi/gTTS
5. https://www.extremetech.com/g00/extreme/215170-artificial-neural-networks-are-changing-the-world-what-are-they?i10c.referrer=https%3A%2F%2Fwww.google.com%2F
6. https://www.tensorflow.org/
7. http://www.keywordsuggests.com/FXRr94b7c5fvnzvo7t2NwHJm7xsirVNDnhgU36dFlow/
8. http://caffe.berkeleyvision.org/
9. http://www.coldvision.io/2016/07/29/image-classification-deep-learning-cnn-caffe-opencv-3-x-cuda/
10. http://caffe.berkeleyvision.org/gathered/examples/mnist.html
11. http://nbviewer.jupyter.org/github/BVLC/caffe/blob/master/examples/01-learning-lenet.ipynb
12. https://github.com/NVIDIA/DIGITS
13. http://yann.lecun.com/exdb/mnist/:
14. http://yann.lecun.com/exdb/lenet/
15. https://www.tensorflow.org/get_started/mnist/beginners
16. https://www.tensorflow.org/get_started/mnist/beginners
17. https://github.com/NVIDIA/DIGITS/tree/master/examples/classification