Machine Learning using GPGPU

Logistic Regression, Multi-Layer Perceptron, Convolutional Neural-Net

Final Project report for

Advanced Computer Architecture (ECEN 5593)

By

Surjith Bhagavath Singh

Table of Contents

Introduction

In 1959 Arthur Samuel defined machine learning as a field of study that gives computers the ability to learn without being explicitly programmed[1].

Machine Learning has evolved over the years and the accuracy rate started surpassing the human error rate. It's all because of the evolution in computational power of the devices and open sourced libraries. Without knowing we are using machine learning and artificial intelligence applications in our day to day life. Google uses machine learning algorithms to search content according the user. Facebook uses its algorithms to show customized ads on our wall according to user preference.

Machine learning became so popular because of its capability to learn things from the wide range of datasets. In simple words this is like teaching something to a kid from the scratch. More the training more the accuracy is. Since it is handling gigabytes of data to train the model it needs lot of computing power to train it fast. Typically, it takes days to train a convolutional neural network in CPUs to get the accuracy to surpass the human error rate. Machine learning is nothing but a set of mathematical equations running on a dataset and updating the weights according to the labelled and predicted output. It can be generalized as a matrix operation.

The scope of this project is to implement the hand written code recognition using different machine learning algorithms such as Logistic Regression, Multi-Layer Perceptron and Convolutional Neural-net on different GPUs and CPUs. GPUs are efficient in doing Matrix (throughput) operations and CPUs are efficient in doing things fast(Latency). This project is all about the basic machine learning algorithms, implementation of algorithms in CPU/GPU and comparing the time taken for CPU/GPU to train a dataset.

History

In 1950 – Alan Turing creates "Turing test" to determine if a computer has intelligence to fool a human into believing it as a human. In 1952 – Arthur Samuel wrote the first computer learning program for IBM to play the game of checkers. It improved the game by 0.37% for every game it played by studying the winning strategies and incorporating the strategies into its program. In 1957 – Frank Rosenblatt designed the first neural network(Perceptron) inspired by human brain. In 1967 – The "nearest neighbor" algorithm was written, this allows computer to recognize very basic pattern recognition. In 1979 – Stanford university students invented "Stanford Cart" which can navigate obstacles in a room on its own. In 1981 – Gerald Dejong introduces Explanation based Learning (EBL), analyses training data and creates general rule. In 1985 – Terry Sejnowski invents NetTalk, program which learns to pronounce words the same way baby does. In 1990-Knowledge driven approach to Data driven approach to draw conclusions. In 1997 – IBM's Deep Blue beats the world champion at chess. In 2006 – Geoffrey Hinton coins the term "Deep Learning". In 2010 – Microsoft Kinect can track human features at a rate of 30 times per second. In 2011 – IBM's Watson beats its human competitors at jeopardy, Google Brain was developed. In 2012 – Google's X lab develops an algorithm which automatically browse and detects cat videos in the youtube. In 2014 – Facebook develops Deepface, able to recognize individuals on photos. In 2015 – Over 3000 AI and Robotics researchers endorsed by Stephen Hawking, Elon Musk and Steve Wozniak Sign an open letter of the

[1] https://en.wikipedia.org/wiki/Machine_learning

danger of autonomous weapons which select and engage targets without human intervention. In 2016 – Google's AlphaGo algorithm defeated the champion of Chinese board game Go.[2]

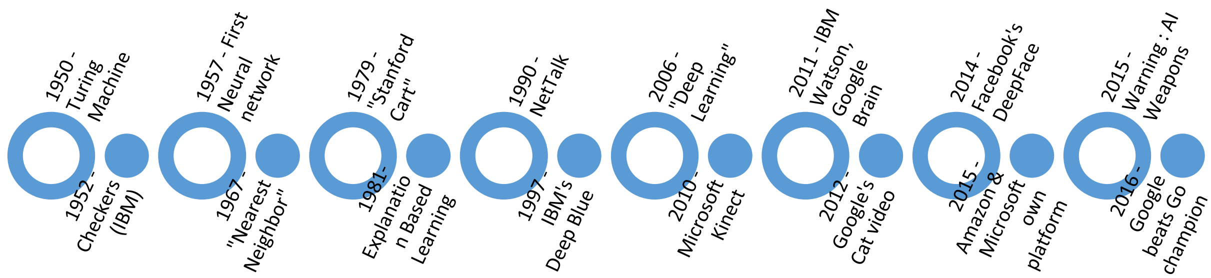

Figure 1: History of Machine Learning

Analysis – Open Source Development platforms

The reason behind the progress in machine learning is because, most of the famous algorithms from big companies like Google and Facebook are open sourced. The tools that are used for machine learning (i.e) Libraries, Toolkits are open sourced. Google Open-Sourced its platform TensorFlow. Lot of libraries contributed by University of Berkeley(Caffe), University of Montreal(Theano), New York University(Torch). Since most of the algorithms are open sourced, it attracts the open source contributors to learn and contribute to the code. It is easy for the researchers to have progress in their research rather than keeping it proprietary.

This project is implemented using Theano (Python Library by University of Montreal). Theano has good documentation compared to other platforms.

Machine Learning Techniques used in the scope of this project

Softmax

This soft max equation helps normalizing the data. This softmax function acts as the activation link for the classifier. Each pixel data from the image is multiplied with its corresponding weights and the result is being computed using the softmax function.

$$\begin{aligned} P(Y=i|x,W,b) &= softmax_i(Wx+b) \\ &= \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}} \end{aligned}$$

```
self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
```

[2] http://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#5a1539ea323f

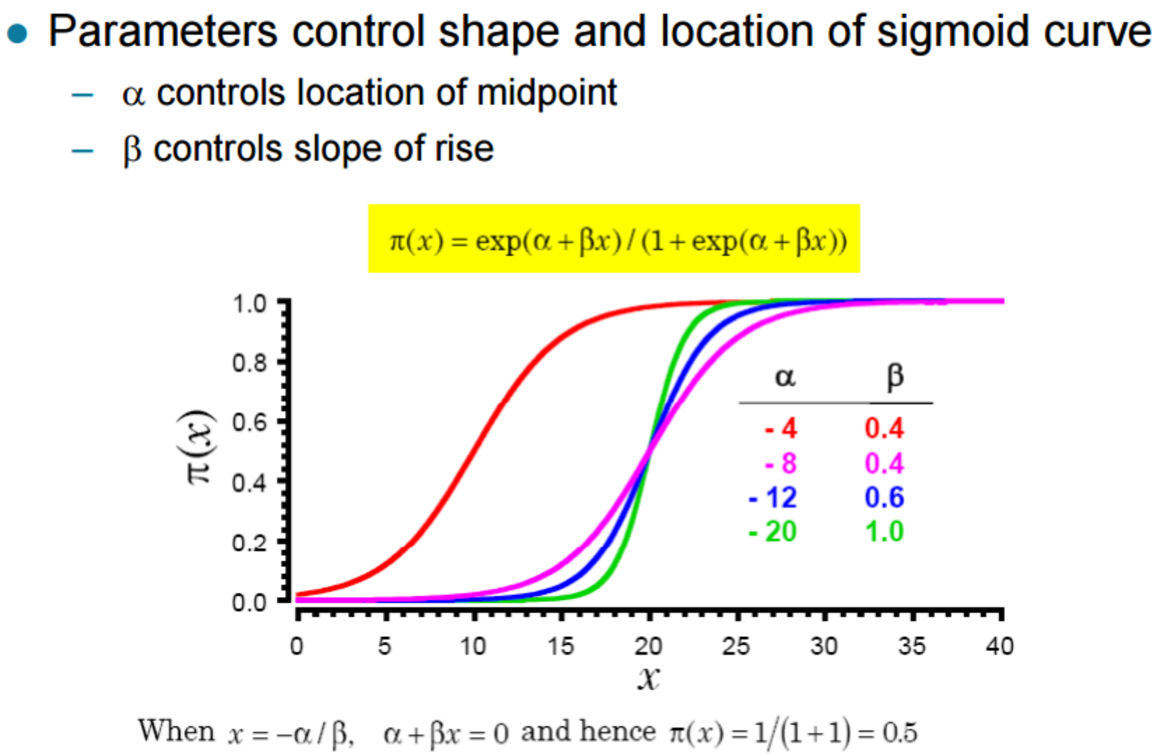

Figure 2: Softmax activation function

Each pixel has a corresponding weight associated with it. In mNIST Data, an image(dataset) looks like the figure shown below.

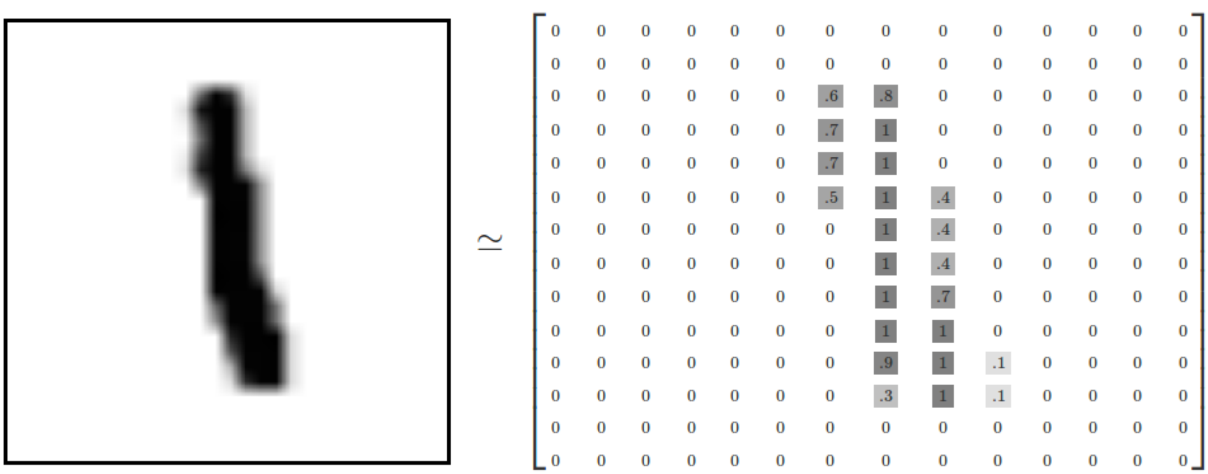

Figure 3: mNIST Dataset - sample dataset[3]

This entire 2 dimensional dataset is converted into one single vector for making the computation easier. We can flatten this array into a vector of 28x28 = 784 numbers (x_i). Our primary aim is to classify the hand written number codes which has numbers from 0 to 9. So it has 10 weight matrices(w_{ij}) with the

[3] https://www.tensorflow.org/versions/r0.7/images/MNIST-Matrix.png

dimension of the image set. Where i= 784 and j=10. Bias elements play an important role in designing the model. Even bias elements will be updated during the training.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

Cost function/ Loss Function

In order to train the model, some parameter has to be defined to check how good is the model. in machine learning it has been defined what it means for a model to be bad, called the cost or loss, and then try to minimize how bad it is. But the two are equivalent.

One very common, very nice cost function is "cross-entropy." Surprisingly, cross-entropy arises from thinking about information compressing codes in information theory but it winds up being an important idea in lots of areas, negative log likelihood,

Since the zero-one loss is not differentiable, optimizing it for large models (thousands or millions of parameters) is prohibitively expensive (computationally). We thus maximize the log-likelihood of our classifier given all the labels in a training set.

The likelihood of the correct class is not the same as the number of right predictions, but from the point of view of a randomly initialized classifier they are pretty similar. Remember that likelihood and zero-one loss are different objectives; you should see that they are correlated on the validation set but sometimes one will rise while the other falls, or vice-versa. Since we usually speak in terms of minimizing a loss function, learning will thus attempt to minimize the negative log-likelihood (NLL), defined as:[4]

$$NLL(\theta, \mathcal{D}) = -\sum_{i=0}^{|\mathcal{D}|} \log P(Y = y^{(i)} | x^{(i)}, \theta)$$

This is the parameter which decides whether to update the weight or not. This parameter tells how good/bad the model is. The code snippet for this function in Theano is.

```
NLL = -T.sum(T.log(p_y_given_x)[T.arange(y.shape[0]), y])
```

How to Train the model?

All the trick is there in the weight matrix, How do we initialize the weight matrix? Here in my implementation I have initialized it to random numbers. So for the first dataset it checks for the y1 value and it compares with the label. If that matches it would not update the weights for it. If it doesn’t it will

[4] http://deeplearning.net/tutorial/deeplearning.pdf

increment the weight matrix by a small factor, so during the training it can reach an optimal point where it can converge and give accurate results. Here comes the cost function.

Stochastic Gradient Descent[5]

It is a simple algorithm in which we repeatedly make small steps downward on an error surface defined by a loss function of some parameters. For the purpose of ordinary gradient descent we consider that the training data is rolled into the loss function. Then the pseudocode of this algorithm can be described as :

```
# GRADIENT DESCENT

while True:

      loss = f(params)

      d_loss_wrt_params = ... # compute gradient

      params -= learning_rate * d_loss_wrt_params

      if <stopping condition is met>:

            return params
```

Stochastic gradient descent (SGD) works according to the same principles as ordinary gradient descent, but proceeds more quickly by estimating the gradient from just a few examples at a time instead of the entire training set. In its purest form, we estimate the gradient from just a single example at a time:

```
#STOCHASTIC GRADIENT DESCENT
for (x_batch,y_batch) in train_batches:
      # imagine an infinite generator
      # that may repeat examples
      loss = f(params, x_batch, y_batch)
      d_loss_wrt_params = ... # compute gradient using theano
      params -= learning_rate * d_loss_wrt_params
      if <stopping condition is met>:
            return params
```

[5] This section is taken from http://deeplearning.net/tutorial/deeplearning.pdf

There is a tradeoff in the choice of the minibatch size B. The reduction of variance and use of SIMD instructions helps most when increasing B from 1 to 2, but the marginal improvement fades rapidly to nothing. With large B, time is wasted in reducing the variance of the gradient estimator, that time would be better spent on additional gradient steps. An optimal B is model-, dataset-, and hardware-dependent, and can be anywhere from 1 to maybe several hundreds. In the tutorial we set it to 20, but this choice is almost arbitrary.

Loading the Data set

This is the code snippet for loading the modified NIST Data for hand written code recognition.

```
import cPickle, gzip, numpy

 # Load the dataset

f = gzip.open(’mnist.pkl.gz’, ’rb’)

train_set, valid_set, test_set = cPickle.load(f)

f.close()
```

CUDA Shared Variable

In order to make the memory access fast, the data has to be there in the shared memory with faster access. Else the entire time will be wasted in copying data back and forth from GPU to CPU. This “theano.shared” function helps theano optimize the code for GPU implementation.

```
shared_dataset(data_xy, borrow=True):

        data_x, data_y = data_xy

        shared_x = theano.shared( numpy.asarray (data_x, dtype = 
theano.config.floatX), borrow=borrow)

        shared_y = theano.shared(  numpy.asarray(data_y,  dtype = 
theano.config.floatX), borrow=borrow)

        return shared_x, T.cast(shared_y, 'int32')
```

How to not over-fit the model with training dataset?

When we have to stop training the model? Early stopping stops the model from over-fitting to the training data set. It validates the dataset from the validation set, which is not involved in training. It doesn’t involve in any gradient descent. There is a possibility for improvement as well as decline in the performance. It properly handles when to increase geometrically the patience to train the dataset.

How to load/save the pre trained model?

It takes hours to train the big model like convolutional neural net. There is a simple way to load/save the weights in python. There is a library called cPickle in python, which makes this task simpler. Here is a code snippet for this operation

```
#To save the weights w
import cPickle
save_file = open('path', 'wb')
cPickle.dump(w.get_value(borrow=True), save_file, -1)

#To load the weights w
import cPickle
save_file = open('path')
w.set_value(cPickle.load(save_file), borrow=True)
```

Machine Learning Algorithms Implemented

Logistic Regression

This is the straight forward probabilistic mathematical model, it has an input layer and an output layer. No intermediate hidden layers. This makes the model simpler and it can achieve an accuracy of 94% over 75 epochs.

Entire image is converted into a single vector an acts as an input layer for this model. Weight matrix of order 784*10 , is multiplied with the input vector of order 1*784 . The results are fed into the activation function (Softmax) to give a binary result.

The weights are updated by stochastic gradient descent method mentioned above. Detailed diagram can be found below.

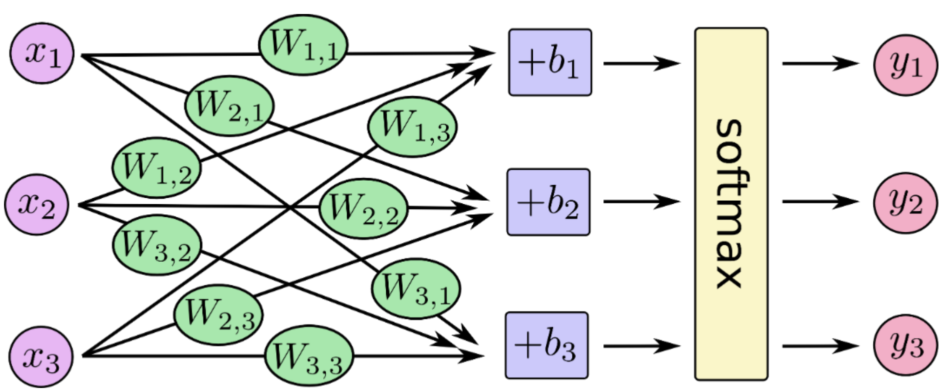

Figure 4: Logistic Regression Model[6]

Multi-Layer Perceptron (MLP)

This model is similar to logistic regression model but it has few hidden layers, which makes it improve the accuracy of the results. As the number of layer increases, accuracy increases. Because it can store lot more weights and the model becomes intelligent enough to store more details.

[6] https://www.tensorflow.org/versions/r0.7/tutorials/mnist/beginners/index.html

This has an input layer, output layer and few hidden layers. Training time increases as the number of layer increases. Weights are updated according to stochastic gradient descent (SGD). Detailed diagram can be found below.

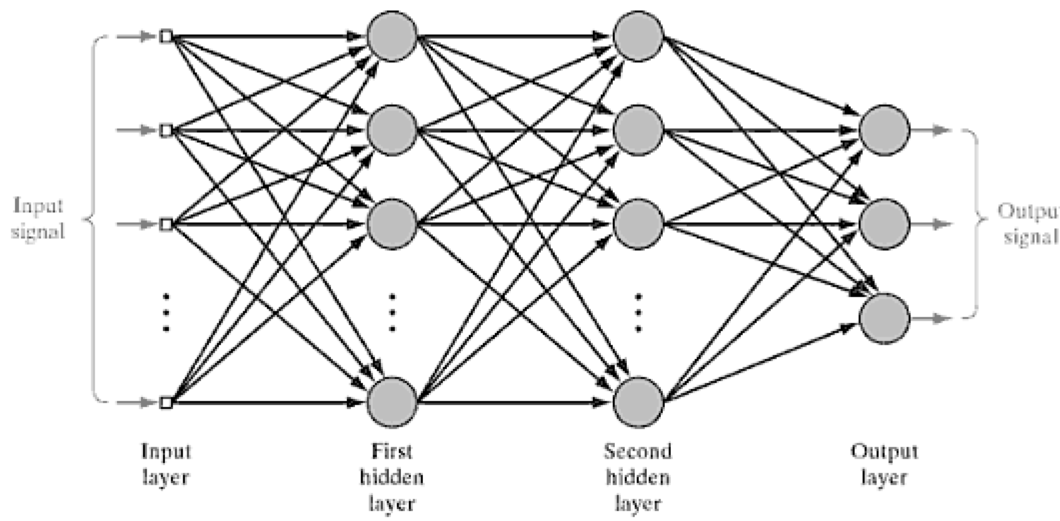

Figure 5: Multi-layer perceptron model (MLP)[7]

Convolutional Neural Network[8]

This model is the combination of the above mentioned models and a little bit of modification. Instead of calculating every pixel, this does the feature extraction using Convolution. It has five layers in it. Input layer, Convolution Layer, Sub sample Layer, Convolution Layer, Sub Sample Layer and Fully Connected Layer.

INPUT [28*28] will hold the raw pixel values of the image, in this case an image of width 28, height 28.

CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and the region they are connected to in the input volume. Then the layer is sub sampled .

CONV layer will compute the output of neurons that are connected to local regions in the first convolutional layer, each computing a dot product between their weights and the region they are connected to in the previous layers volume. .This has been sub sampled and the sub sample results in (10*10).

FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of MNIST. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

---

[7] https://elogeel.files.wordpress.com/2010/05/050510_1627_multilayerp1.png
[8] http://cs231n.github.io/convolutional-networks/#case

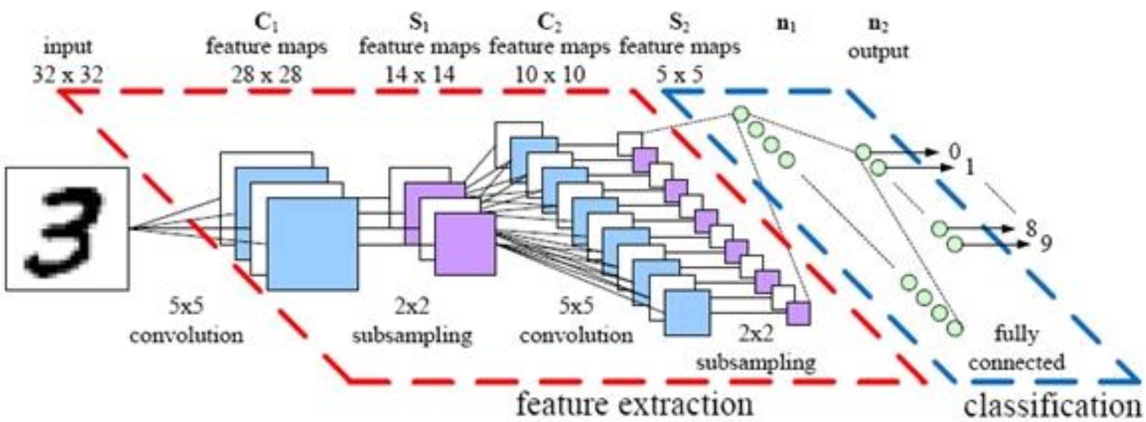

Figure 6: Convolutional Neural Network – Sample Model[9]

Implementation

These machine learning algorithms are implemented in two platforms CPU and GPGPU in Server node(Janus) and Embedded platform(NVidia Jetson – TK1). Both uses the same code. In this implementation Theano libraries are used for doing mathematical operations. Theano uses numpy libraries for matrix operations, which is the most optimized library. GPUs are good at doing floating point matrix operations. Theano has backend support of CUDA and it generates CUDA code for GPU operations. The implementation and platform details are mentioned below.

Platform

Two different platforms are chosen for the implementation of these models. Mobile-Embedded platform (NVidia Jetson) and Super Computer node (Janus). Reason for choosing two extreme platforms is to explore and compare their performances for these complex machine learning algorithms.

NVidia Jetson TK1 (Tegra) [10]

This development kit is a famous SoC mobile embedded platform with an ARM Cortex A15 Quadcore CPU with 192 core Kepler GPU with a clock of 2.3GHz. This core is used in Google Nexus 7 tablet. It is a powerful and power efficient SoC suitable for mobile phone chips. The capacity of this machine is 300 GFLOPS(Maximum)[11] according to its datasheet specifications. It has on 16 GB 4.51 eMMC Memory. It has a 2 GB RAM Memory with 64-bit Width. Theano flags for CPU/GPU

[9] http://parse.ele.tue.nl/cluster/2/CNNArchitecture.jpg
[10] http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html
[11] FLOPS_CPU = (CPU speed in Hz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node)

```
THEANO_FLAGS=mode=FAST_RUN,device=cpu,floatX=float32 python mlp.py
THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python mlp.py
```

Janus[12] – GPU Node[13]

Janus is a super computer available for all CU Students and it is one of the powerful computers in Central America. It boasts the capabilities of having a 12 core Intel Xeon X5660 State of the art CPU(For each node) and a State of the art 448 core NVIDIA Tesla GPU with a clock of 2.8GHz. The capacity of the node is 649 GFLOPS. [14] It has 1368 Compute nodes, It's overall capacity is 184 TFLOPS. It has 24GB RAM per node.

A set of packages has to be loaded in order to run the program.

```
ml intel
ml CUDA/7.0.28
ml python/2.7.3
ml numpy
ml scipy
ml slurm
```

These packages are needed for the super computer node to run the program file. The shell script to run in the GPU/ CPU has been attached below.

```
#!/bin/bash
#SBATCH --nodes 1
#SBATCH --output hello-world.out
#SBATCH --qos gpu
#SBATCH --partition crc-gpu
export
PYTHONPATH=$PYTHONPATH:/projects/subh6068/python_packages/lib/python2.7/site-
packages
export CUDA_ROOT=$CUDA_ROOT:/curc/tools/x86_64/rh6/software/cuda/7.0.28
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/curc/tools/x86_64/rh6/software/cuda/7.0.28/
lib
THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python mlp.py
```

[12] https://www.rc.colorado.edu/resources/compute/janus
[13] https://www.rc.colorado.edu/resources/compute/gpunodes
[14] FLOPS_GPU= no of cores * no of SIMD units * ((no of mul-add units*2) + (no of mul units)) * clock speed in Hz

```bash
#!/bin/bash

#SBATCH --nodes 1

#SBATCH --output mlp-cpu.out

#SBATCH --qos janus-compile

export
PYTHONPATH=$PYTHONPATH:/projects/subh6068/python_packages/lib/python2.7/site-packages

export CUDA_ROOT=$CUDA_ROOT:/curc/tools/x86_64/rh6/software/cuda/7.0.28

export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/curc/tools/x86_64/rh6/software/cuda/7.0.28/lib

THEANO_FLAGS=mode=FAST_RUN,device=cpu,floatX=float32 python mlp.py
```

Results

Quad core ARM Cortex A15 CPU

The logistic regression algorithm has been implemented in this processor, using CPU only flag in Jetson and the results are pretty slow compared to other platforms. It took so long to train the other models. It took 128 Seconds to train the Logistic regression model and it can be able to achieve 92.5% accuracy in validation. Capacity of this CPU – 27 GFLOPs

Quad core ARM Cortex A15 CPU + Kepler GPU

This combination is powerful and it can be compared with the super computer nodes. All three algorithms are trained using this combination using GPU flags in Theano. It took 74 seconds to train the Logistic Regression model, 390 minutes for Multi-Layer Perceptron and 183.19 minutes for Convolutional neural net. It can be able to achieve 92.5% accuracy in logistic Regression, 98.3% in Multi-Layer Perceptron and 99.1 % accuracy in Convolutional Neural net model. Capacity of this SoC – 300 GFLOPs

12 Core Intel Xeon 5660 CPU

This processor is superfast compared to the ARM processor. All three algorithms are trained using this CPU using CPU flags in Theano. It took 12.2 seconds to train logistic Regression model, 75.21 minutes for Multi-layer perceptron and 148.93 minutes for convolutional neural net. It can be able to achieve 92.5% accuracy in logistic Regression, 98.3% in Multi-Layer Perceptron and 99.1 % accuracy in Convolutional Neural net model. Capacity of this Supercomputer node –134 GFLOPs

12 Core Intel Xeon 5660 CPU + NVIDIA Tesla M2070 GPU

This is the high speed combo among all four. All three algorithms are trained using GPU flags in Theano. It took 10 seconds to train logistic Regression model, 85.7 minutes for Multi-layer perceptron and 38.89 minutes for convolutional neural net. It can be able to achieve 92.5% accuracy in logistic Regression, 98.3% in Multi-Layer Perceptron and 99.1 % accuracy in Convolutional Neural net model. Capacity of this Supercomputer node + GPU – 649 GFLOPs.

Analysis

Comparative results of the entire implementation is tabulated and attached below. The reason for the fast training of CPU in Multi Layer perceptron model is because of not using shared variables, which means it makes the GPU performance poor. Processing time is less than data transferring time. CGMA is poor for that case. From the results it is clear that CPU+GPU Combination can achieve faster goal. Embedded platforms are capable enough and cost efficient.

	ARM® Cortex™-A15 CPU – Quadcore (Jetson SoC)	**ARM® A15 + 192 NVIDIA Kepler CUDA Cores (Jetson SoC)**	**Intel® Xeon® X5660 – 12 core (Janus Node)**	**Intel® Xeon® X5660 + 448 NVIDIA Tesla CUDA Cores (Janus Node)**
Operating Capacity	27 GFLOPS	300 GFLOPS	134 GFLOPS	649 GFLOPS
Cores	4 CPU cores	4 CPU + 192 GPU	12 CPU cores	12 CPU + 448 GPU
Logistic Regression(75 epochs)	128 seconds 92.5% accurate	74 seconds 92.5% accurate	12.2 seconds 92.5% accurate	10 seconds 92.5%accurate
Multi Layer Perceptron(1000 Epoch)	No Patience to do this	390.22 minutes 98.3% accurate	75.21 minutes 98.3% accurate	85.77 minutes 98.3% accurate
Convolutional Neural Net (200 epochs)	No Patience to do this	183.19 minutes 99.09% accurate	148.93 minutes 99.09% accurate	38.89 minutes 99.09% accurate

Table 1: Results

Future Development

Implementing the same algorithms for different computer vision problem (Saliency) using machine learning, ultimate aim is to contribute for the research I am currently doing under Prof. Sam Siewert. I have taken this course as an opportunity to learn about the computer architecture and machine learning. The objective is achieved. The code has been uploaded in github[15] webpage for future development.

[15] https://github.com/surjithbs17/machinelearning

Codes

Logistic Regression

```
"""
This code has been developed using the tutorial from deeplearning.net
Author: Surjith Bhagavath Singh
File Name: convolutional_mlp.py
"""


from __future__ import print_function

__docformat__ = 'restructedtext en'

import six.moves.cPickle as pickle
import gzip
import os
import sys
import timeit

import numpy

import theano
import theano.tensor as T

class LogisticRegression(object):

    def __init__(self, input, n_in, n_out):
        self.W = theano.shared(
            value=numpy.zeros(
                (n_in, n_out),
                dtype=theano.config.floatX
            ),
            name='W',
            borrow=True
        )
        self.b = theano.shared(
            value=numpy.zeros(
                (n_out,),
                dtype=theano.config.floatX
            ),
            name='b',
            borrow=True
        )

        self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)

        self.y_pred = T.argmax(self.p_y_given_x, axis=1)

        self.params = [self.W, self.b]

        self.input = input

    def negative_log_likelihood(self, y):

        return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
```

```
    def errors(self, y):

        if y.ndim != self.y_pred.ndim:
            raise TypeError(
                'y should have the same shape as self.y_pred',
                ('y', y.type, 'y_pred', self.y_pred.type)
            )
        if y.dtype.startswith('int'):
            return T.mean(T.neq(self.y_pred, y))
        else:
            raise NotImplementedError()


def load_data(dataset):
    data_dir, data_file = os.path.split(dataset)
    if data_dir == "" and not os.path.isfile(dataset):
        new_path = os.path.join(
            os.path.split(__file__)[0],
            "..",
            "data",
            dataset
        )
        if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
            dataset = new_path

    if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
        from six.moves import urllib
        origin = (
            'http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz'
        )
        print('Downloading data from %s' % origin)
        urllib.request.urlretrieve(origin, dataset)

    print('... loading data')

    # Load the dataset
    with gzip.open(dataset, 'rb') as f:
        try:
            train_set, valid_set, test_set = pickle.load(f,
encoding='latin1')
        except:
            train_set, valid_set, test_set = pickle.load(f)

    def shared_dataset(data_xy, borrow=True):
        data_x, data_y = data_xy
        shared_x =
theano.shared(numpy.asarray(data_x,dtype=theano.config.floatX),borrow=borrow)
        shared_y =
theano.shared(numpy.asarray(data_y,dtype=theano.config.floatX),borrow=borrow)
        return shared_x, T.cast(shared_y, 'int32')

    test_set_x, test_set_y = shared_dataset(test_set)
    valid_set_x, valid_set_y = shared_dataset(valid_set)
    train_set_x, train_set_y = shared_dataset(train_set)
```

```
    rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
            (test_set_x, test_set_y)]
    return rval


def sgd_optimization_mnist(learning_rate=0.13,
n_epochs=1000,dataset='mnist.pkl.gz',batch_size=600):
    datasets = load_data(dataset)
    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]

    n_train_batches = train_set_x.get_value(borrow=True).shape[0] //
batch_size
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] //
batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size

    print('... building the model')

    index = T.lscalar()  # index to a [mini]batch

    x = T.matrix('x')  # data, presented as rasterized images
    y = T.ivector('y')  # labels, presented as 1D vector of [int] labels

    classifier = LogisticRegression(input=x, n_in=28 * 28, n_out=10)

    cost = classifier.negative_log_likelihood(y)

    test_model = theano.function(
        inputs=[index],
        outputs=classifier.errors(y),
        givens={
            x: test_set_x[index * batch_size: (index + 1) * batch_size],
            y: test_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    validate_model = theano.function(
        inputs=[index],
        outputs=classifier.errors(y),
        givens={
            x: valid_set_x[index * batch_size: (index + 1) * batch_size],
            y: valid_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    g_W = T.grad(cost=cost, wrt=classifier.W)
    g_b = T.grad(cost=cost, wrt=classifier.b)

    updates = [(classifier.W, classifier.W - learning_rate * g_W),
               (classifier.b, classifier.b - learning_rate * g_b)]

    train_model = theano.function(
        inputs=[index],
        outputs=cost,
        updates=updates,
```

```
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
    print('... training the model')
    # early-stopping parameters
    patience = 5000  # look as this many examples regardless
    patience_increase = 2  # wait this much longer when a new best is
                              # found
    improvement_threshold = 0.995  # a relative improvement of this much is
                              # considered significant
    validation_frequency = min(n_train_batches, patience // 2)
                              # go through this many
                              # minibatche before checking the network
                              # on the validation set; in this case we
                              # check every epoch

    best_validation_loss = numpy.inf
    test_score = 0.
    start_time = timeit.default_timer()

    done_looping = False
    epoch = 0
    while (epoch < n_epochs) and (not done_looping):
        epoch = epoch + 1
        for minibatch_index in range(n_train_batches):

            minibatch_avg_cost = train_model(minibatch_index)
            iter = (epoch - 1) * n_train_batches + minibatch_index

            if (iter + 1) % validation_frequency == 0:
                validation_losses = [validate_model(i)
                                     for i in range(n_valid_batches)]
                this_validation_loss = numpy.mean(validation_losses)

                print(
                    'epoch %i, minibatch %i/%i, validation error %f %%' %
                    (
                        epoch,
                        minibatch_index + 1,
                        n_train_batches,
                        this_validation_loss * 100.
                    )
                )

                if this_validation_loss < best_validation_loss:
                    if this_validation_loss < best_validation_loss *  \
                       improvement_threshold:
                        patience = max(patience, iter * patience_increase)

                    best_validation_loss = this_validation_loss

                    test_losses = [test_model(i)
                                   for i in range(n_test_batches)]
                    test_score = numpy.mean(test_losses)
```

```
                    print(
                        (
                            '     epoch %i, minibatch %i/%i, test error of'
                            ' best model %f %%'
                        ) %
                        (
                            epoch,
                            minibatch_index + 1,
                            n_train_batches,
                            test_score * 100.
                        )
                    )

                    # save the best model
                    with open('best_model.pkl', 'wb') as f:
                        pickle.dump(classifier, f)

            if patience <= iter:
                done_looping = True
                break

    end_time = timeit.default_timer()
    print(
        (
            'Optimization complete with best validation score of %f %%,'
            'with test performance %f %%'
        )
        % (best_validation_loss * 100., test_score * 100.)
    )
    print('The code run for %d epochs, with %f epochs/sec' % (
        epoch, 1. * epoch / (end_time - start_time)))
    print(('The code for file ' +
           os.path.split(__file__)[1] +
           ' ran for %.1fs' % ((end_time - start_time))), file=sys.stderr)


def predict():
    # load the saved model
    classifier = pickle.load(open('best_model.pkl'))

    # compile a predictor function
    predict_model = theano.function(
        inputs=[classifier.input],
        outputs=classifier.y_pred)

    # some examples from test test
    dataset='mnist.pkl.gz'
    datasets = load_data(dataset)
    test_set_x, test_set_y = datasets[2]
    test_set_x = test_set_x.get_value()

    predicted_values = predict_model(test_set_x[:100])
    print("Predicted values for the first 100 examples in test set:")
    print(predicted_values)


if __name__ == '__main__':
```

```
    sgd_optimization_mnist()
    predict()
```

Multi Layer Perceptron

```
"""
This code has been developed using the tutorial from deeplearning.net
Author: Surjith Bhagavath Singh
"""

from __future__ import print_function

__docformat__ = 'restructedtext en'


import os
import sys
import timeit

import numpy

import theano
import theano.tensor as T


from logistic_sgd import LogisticRegression, load_data


class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None,
                 activation=T.tanh):
        self.input = input
        if W is None:
            W_values = numpy.asarray(
                rng.uniform(
                    low=-numpy.sqrt(6. / (n_in + n_out)),
                    high=numpy.sqrt(6. / (n_in + n_out)),
                    size=(n_in, n_out)
                ),
                dtype=theano.config.floatX
            )
            if activation == theano.tensor.nnet.sigmoid:
                W_values *= 4

            W = theano.shared(value=W_values, name='W', borrow=True)

        if b is None:
            b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
            b = theano.shared(value=b_values, name='b', borrow=True)

        self.W = W
        self.b = b

        lin_output = T.dot(input, self.W) + self.b
```

```
        self.output = (
            lin_output if activation is None
            else activation(lin_output)
        )
        self.params = [self.W, self.b]


class MLP(object):

    def __init__(self, rng, input, n_in, n_hidden, n_out):
        self.hiddenLayer = HiddenLayer(
            rng=rng,
            input=input,
            n_in=n_in,
            n_out=n_hidden,
            activation=T.tanh
        )
        self.logRegressionLayer = LogisticRegression(
            input=self.hiddenLayer.output,
            n_in=n_hidden,
            n_out=n_out
        )
        self.L1 = (
            abs(self.hiddenLayer.W).sum()
            + abs(self.logRegressionLayer.W).sum()
        )

        self.L2_sqr = (
            (self.hiddenLayer.W ** 2).sum()
            + (self.logRegressionLayer.W ** 2).sum()
        )

        self.negative_log_likelihood = (
            self.logRegressionLayer.negative_log_likelihood
        )
        self.errors = self.logRegressionLayer.errors

        self.params = self.hiddenLayer.params +
self.logRegressionLayer.params

        self.input = input


def test_mlp(learning_rate=0.01, L1_reg=0.00, L2_reg=0.0001, n_epochs=1000,
             dataset='mnist.pkl.gz', batch_size=20, n_hidden=500):
    datasets = load_data(dataset)

    train_set_x, train_set_y = datasets[0]
    valid_set_x, valid_set_y = datasets[1]
    test_set_x, test_set_y = datasets[2]

    n_train_batches = train_set_x.get_value(borrow=True).shape[0] //
batch_size
    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] //
batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0] // batch_size
```

```
print('... building the model')

index = T.lscalar()  # index to a [mini]batch
x = T.matrix('x')  # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

rng = numpy.random.RandomState(1234)

classifier = MLP(
    rng=rng,
    input=x,
    n_in=28 * 28,
    n_hidden=n_hidden,
    n_out=10
)

cost = (
    classifier.negative_log_likelihood(y)
    + L1_reg * classifier.L1
    + L2_reg * classifier.L2_sqr
)
test_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: test_set_x[index * batch_size:(index + 1) * batch_size],
        y: test_set_y[index * batch_size:(index + 1) * batch_size]
    }
)

validate_model = theano.function(
    inputs=[index],
    outputs=classifier.errors(y),
    givens={
        x: valid_set_x[index * batch_size:(index + 1) * batch_size],
        y: valid_set_y[index * batch_size:(index + 1) * batch_size]
    }
)

gparams = [T.grad(cost, param) for param in classifier.params]

updates = [
    (param, param - learning_rate * gparam)
    for param, gparam in zip(classifier.params, gparams)
]

train_model = theano.function(
    inputs=[index],
    outputs=cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size],
        y: train_set_y[index * batch_size: (index + 1) * batch_size]
    }
)
print('... training')
```

```
patience = 10000  # look as this many examples regardless
patience_increase = 2  # wait this much longer when a new best is
                       # found
improvement_threshold = 0.995  # a relative improvement of this much is
                               # considered significant
validation_frequency = min(n_train_batches, patience // 2)
                              # go through this many
                              # minibatche before checking the network
                              # on the validation set; in this case we
                              # check every epoch

best_validation_loss = numpy.inf
best_iter = 0
test_score = 0.
start_time = timeit.default_timer()

epoch = 0
done_looping = False

while (epoch < n_epochs) and (not done_looping):
    epoch = epoch + 1
    for minibatch_index in range(n_train_batches):

        minibatch_avg_cost = train_model(minibatch_index)
        iter = (epoch - 1) * n_train_batches + minibatch_index

        if (iter + 1) % validation_frequency == 0:
            validation_losses = [validate_model(i) for i
                                 in range(n_valid_batches)]
            this_validation_loss = numpy.mean(validation_losses)

            print(
                'epoch %i, minibatch %i/%i, validation error %f %%' %
                (
                    epoch,
                    minibatch_index + 1,
                    n_train_batches,
                    this_validation_loss * 100.
                )
            )

            if this_validation_loss < best_validation_loss:
                if (
                    this_validation_loss < best_validation_loss *
                    improvement_threshold
                ):
                    patience = max(patience, iter * patience_increase)

                best_validation_loss = this_validation_loss
                best_iter = iter

                test_losses = [test_model(i) for i
                               in range(n_test_batches)]
                test_score = numpy.mean(test_losses)

                print(('     epoch %i, minibatch %i/%i, test error of '
```

```
                                   'best model %f %%') %
                                  (epoch, minibatch_index + 1, n_train_batches,
                                   test_score * 100.))

                if patience <= iter:
                    done_looping = True
                    break

    end_time = timeit.default_timer()
    print(('Optimization complete. Best validation score of %f %% '
           'obtained at iteration %i, with test performance %f %%') %
          (best_validation_loss * 100., best_iter + 1, test_score * 100.))
    print(('The code for file ' +
           os.path.split(__file__)[1] +
           ' ran for %.2fm' % ((end_time - start_time) / 60.)),
file=sys.stderr)


if __name__ == '__main__':
    test_mlp()
```

Convolutional Neural Net

```
"""
This code has been developed using the tutorial from deeplearning.net
Author: Surjith Bhagavath Singh
File Name: convolutional_mlp.py
"""
from __future__ import print_function

import os
import sys
import timeit

import numpy

import theano
import theano.tensor as T
from theano.tensor.signal import downsample
from theano.tensor.nnet import conv2d

from logistic_sgd import LogisticRegression, load_data
from mlp import HiddenLayer


class LeNetConvPoolLayer(object):
    """Pool Layer of a convolutional network """

    def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2,
2)):

        assert image_shape[1] == filter_shape[1]
        self.input = input
```

```
        # there are "num input feature maps * filter height * filter width"
        # inputs to each hidden unit
        fan_in = numpy.prod(filter_shape[1:])
        # each unit in the lower layer receives a gradient from:
        # "num output feature maps * filter height * filter width" /
        #   pooling size
        fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:]) //
                   numpy.prod(poolsize))
        # initialize weights with random weights
        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                rng.uniform(low=-W_bound, high=W_bound, size=filter_shape),
                dtype=theano.config.floatX
            ),
            borrow=True
        )

        # the bias is a 1D tensor -- one bias per output feature map
        b_values = numpy.zeros((filter_shape[0],),
dtype=theano.config.floatX)
        self.b = theano.shared(value=b_values, borrow=True)

        # convolve input feature maps with filters
        conv_out = conv2d(
            input=input,
            filters=self.W,
            filter_shape=filter_shape,
            input_shape=image_shape
        )

        # downsample each feature map individually, using maxpooling
        pooled_out = downsample.max_pool_2d(
            input=conv_out,
            ds=poolsize,
            ignore_border=True
        )


        self.output = T.tanh(pooled_out + self.b.dimshuffle('x', 0, 'x',
'x'))

        # store parameters of this layer
        self.params = [self.W, self.b]

        # keep track of model input
        self.input = input


def evaluate_lenet5(learning_rate=0.1, n_epochs=200,
                    dataset='mnist.pkl.gz',
                    nkerns=[20, 50], batch_size=500):

    rng = numpy.random.RandomState(23455)

    datasets = load_data(dataset)
```

```
train_set_x, train_set_y = datasets[0]
valid_set_x, valid_set_y = datasets[1]
test_set_x, test_set_y = datasets[2]

n_train_batches = train_set_x.get_value(borrow=True).shape[0]
n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
n_test_batches = test_set_x.get_value(borrow=True).shape[0]
n_train_batches //= batch_size
n_valid_batches //= batch_size
n_test_batches //= batch_size

index = T.lscalar()  # index to a [mini]batch

x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

print('... building the model')

layer0_input = x.reshape((batch_size, 1, 28, 28))

layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 1, 28, 28),
    filter_shape=(nkerns[0], 1, 5, 5),
    poolsize=(2, 2)
)

layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 12, 12),
    filter_shape=(nkerns[1], nkerns[0], 5, 5),
    poolsize=(2, 2)
)

layer2_input = layer1.output.flatten(2)

layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 4 * 4,
    n_out=500,
    activation=T.tanh
)

layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)

cost = layer3.negative_log_likelihood(y)
test_model = theano.function(
    [index],
    layer3.errors(y),
    givens={
        x: test_set_x[index * batch_size: (index + 1) * batch_size],
        y: test_set_y[index * batch_size: (index + 1) * batch_size]
```

```
        }
    )

    validate_model = theano.function(
        [index],
        layer3.errors(y),
        givens={
            x: valid_set_x[index * batch_size: (index + 1) * batch_size],
            y: valid_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    params = layer3.params + layer2.params + layer1.params + layer0.params

    grads = T.grad(cost, params)

    updates = [
        (param_i, param_i - learning_rate * grad_i)
        for param_i, grad_i in zip(params, grads)
    ]

    train_model = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
    print('... training')
    # early-stopping parameters
    patience = 10000  # look as this many examples regardless
    patience_increase = 2  # wait this much longer when a new best is
                           # found
    improvement_threshold = 0.995  # a relative improvement of this much is
                                   # considered significant
    validation_frequency = min(n_train_batches, patience // 2)
                                  # go through this many
                                  # minibatche before checking the network
                                  # on the validation set; in this case we
                                  # check every epoch

    best_validation_loss = numpy.inf
    best_iter = 0
    test_score = 0.
    start_time = timeit.default_timer()

    epoch = 0
    done_looping = False

    while (epoch < n_epochs) and (not done_looping):
        epoch = epoch + 1
        for minibatch_index in range(n_train_batches):

            iter = (epoch - 1) * n_train_batches + minibatch_index
```

```
            if iter % 100 == 0:
                print('training @ iter = ', iter)
            cost_ij = train_model(minibatch_index)

            if (iter + 1) % validation_frequency == 0:

                validation_losses = [validate_model(i) for i
                                     in range(n_valid_batches)]
                this_validation_loss = numpy.mean(validation_losses)
                print('epoch %i, minibatch %i/%i, validation error %f %%' %
                      (epoch, minibatch_index + 1, n_train_batches,
                       this_validation_loss * 100.))

                if this_validation_loss < best_validation_loss:

                    if this_validation_loss < best_validation_loss *  \
                       improvement_threshold:
                        patience = max(patience, iter * patience_increase)

                    best_validation_loss = this_validation_loss
                    best_iter = iter

                    test_losses = [
                        test_model(i)
                        for i in range(n_test_batches)
                    ]
                    test_score = numpy.mean(test_losses)
                    print(('     epoch %i, minibatch %i/%i, test error of '
                           'best model %f %%') %
                          (epoch, minibatch_index + 1, n_train_batches,
                           test_score * 100.))

            if patience <= iter:
                done_looping = True
                break

    end_time = timeit.default_timer()
    print('Optimization complete.')
    print('Best validation score of %f %% obtained at iteration %i, '
          'with test performance %f %%' %
          (best_validation_loss * 100., best_iter + 1, test_score * 100.))
    print(('The code for file ' +
           os.path.split(__file__)[1] +
           ' ran for %.2fm' % ((end_time - start_time) / 60.)),
file=sys.stderr)

if __name__ == '__main__':
    evaluate_lenet5()


def experiment(state, channel):
    evaluate_lenet5(state.learning_rate, dataset=state.dataset)
```