Performance and User-friendliness of low-level and higher-level deep neural network frameworks

Team: J^3
Members: Jack Gu, Joseph Gozum, Justin Lam

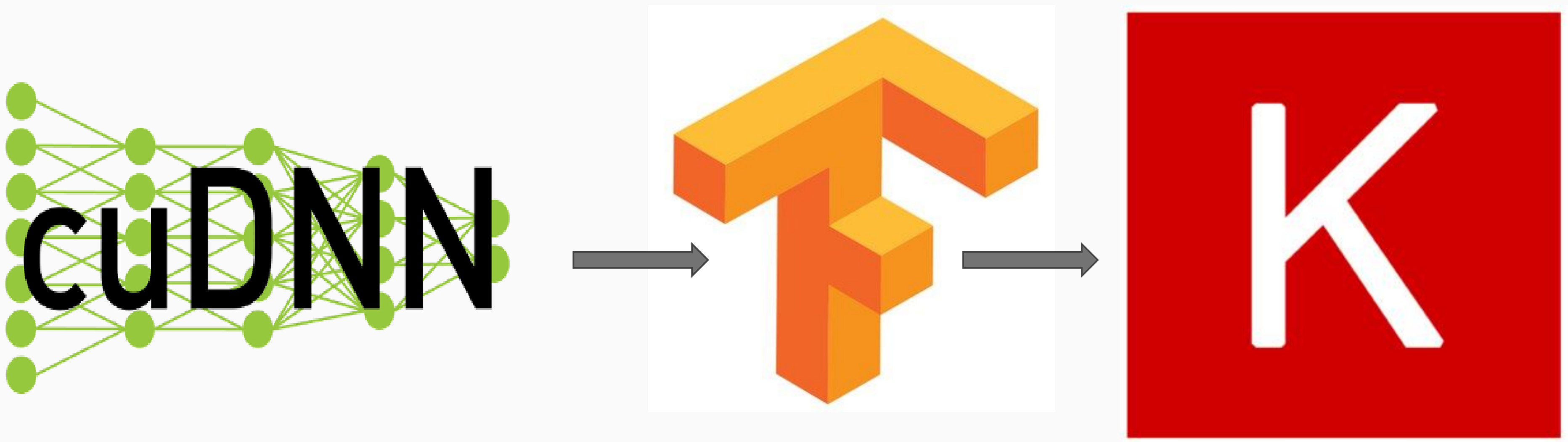
Deep Learning

An ever growing field with a multitude of different programming frameworks that provide high-level programming interface (e.g. Python, C++, etc)

Caffe2 Microsoft CNTK TensorFlow theano

Chainer PYTORCH dmlc mxnet K

API Levels



Goal of Project

From the lowest-level (cuDNN) to increasingly higher-level frameworks (i.e. Tensorflow, Keras), we want to observe a possible increase in overhead and any other trade-offs as the level of abstraction rises while we implement the LeNet CNN

¯_(ツ)_/¯

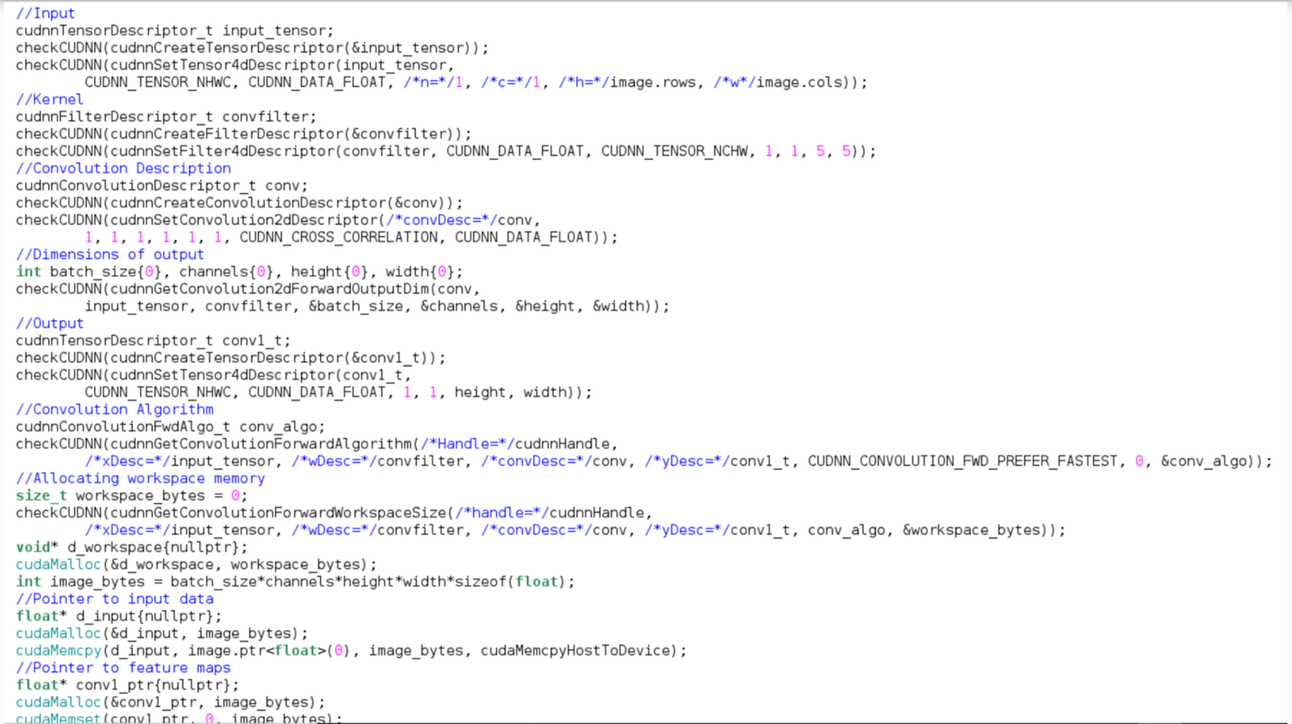
Observations?

Our metric focuses on the time to classification/inference, measured in milliseconds.

We'll then discuss optimizations or lack thereof that may have been taken in each framework or in our code.

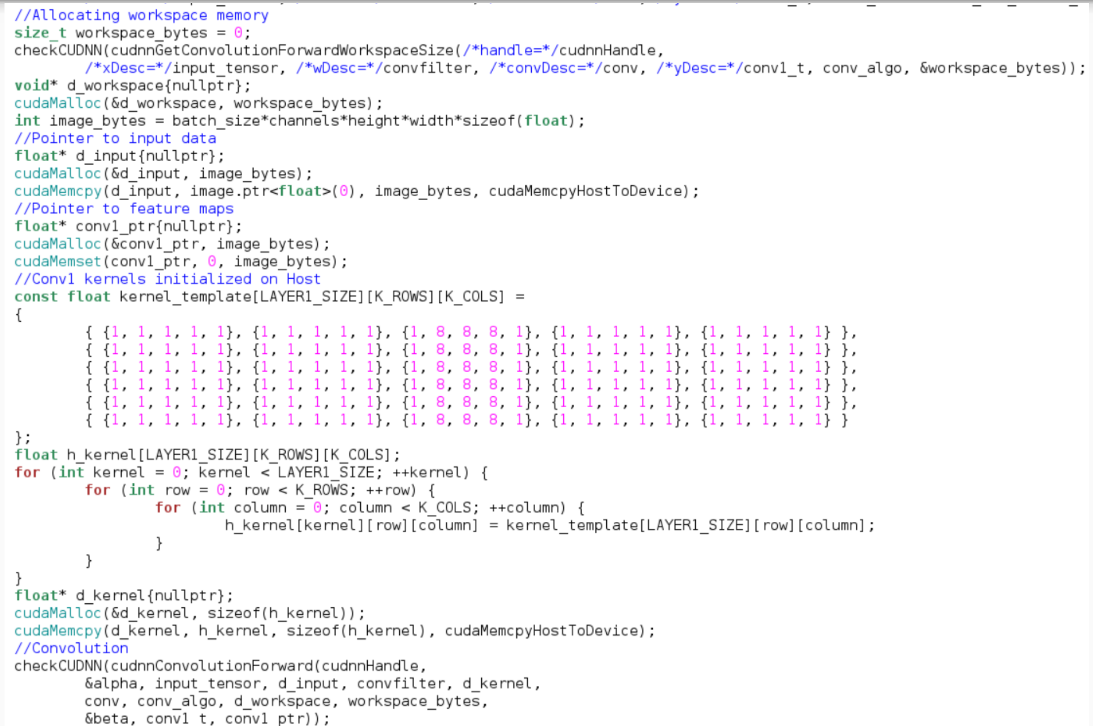
A discussion on the tradeoff between frameworks.

cuDNN: Convolution



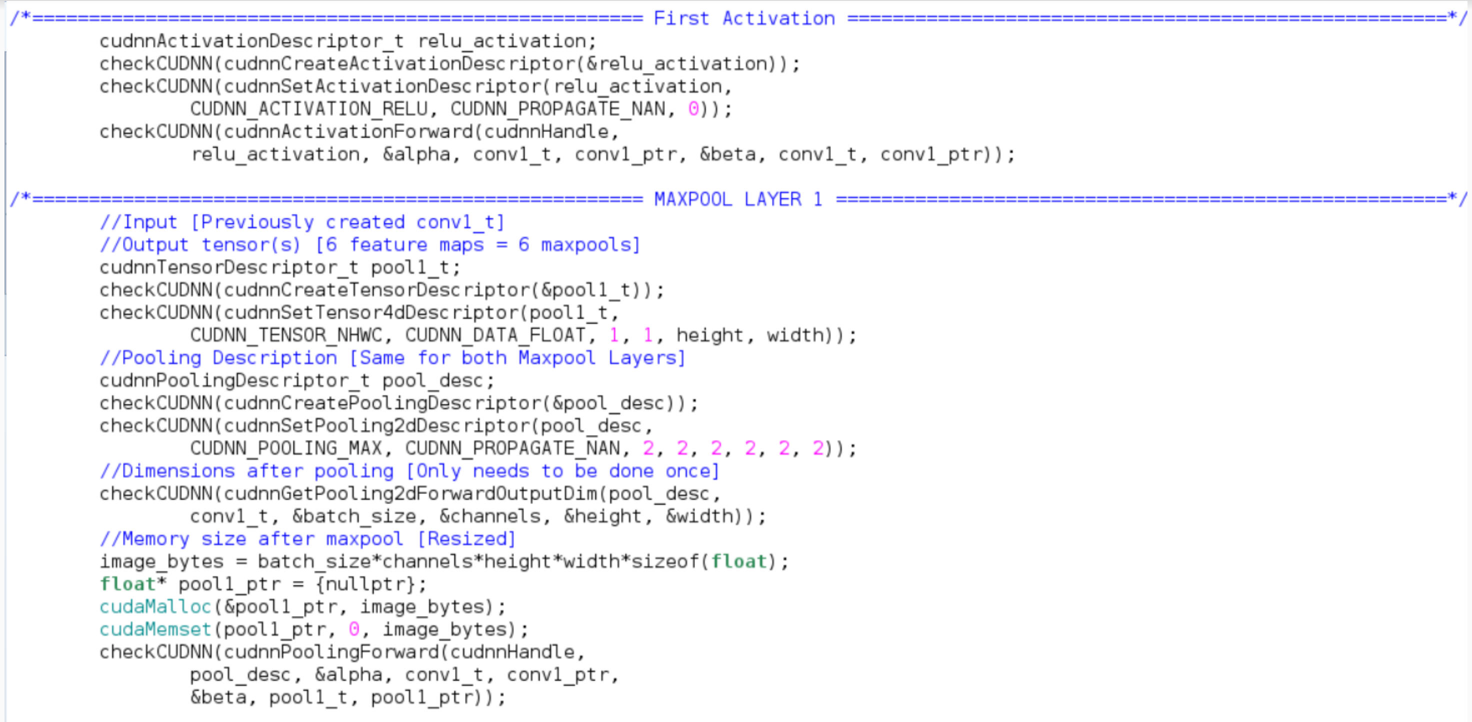
```
//Input
cudnnTensorDescriptor_t input_tensor;
checkCUDNN(cudnnCreateTensorDescriptor(&input_tensor));
checkCUDNN(cudnnSetTensor4dDescriptor(input_tensor,
        CUDNN_TENSOR_NHWC, CUDNN_DATA_FLOAT, /*n=*/1, /*c=*/1, /*h=*/image.rows, /*w*/image.cols));
//Kernel
cudnnFilterDescriptor_t convfilter;
checkCUDNN(cudnnCreateFilterDescriptor(&convfilter));
checkCUDNN(cudnnSetFilter4dDescriptor(convfilter, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 1, 1, 5, 5));
//Convolution Description
cudnnConvolutionDescriptor_t conv;
checkCUDNN(cudnnCreateConvolutionDescriptor(&conv));
checkCUDNN(cudnnSetConvolution2dDescriptor(/*convDesc=*/conv,
        1, 1, 1, 1, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
//Dimensions of output
int batch_size{0}, channels{0}, height{0}, width{0};
checkCUDNN(cudnnGetConvolution2dForwardOutputDim(conv,
        input_tensor, convfilter, &batch_size, &channels, &height, &width));
//Output
cudnnTensorDescriptor_t conv1_t;
checkCUDNN(cudnnCreateTensorDescriptor(&conv1_t));
checkCUDNN(cudnnSetTensor4dDescriptor(conv1_t,
        CUDNN_TENSOR_NHWC, CUDNN_DATA_FLOAT, 1, 1, height, width));
//Convolution Algorithm
cudnnConvolutionFwdAlgo_t conv_algo;
checkCUDNN(cudnnGetConvolutionForwardAlgorithm(/*Handle=*/cudnnHandle,
        /*xDesc=*/input_tensor, /*wDesc=*/convfilter, /*convDesc=*/conv, /*yDesc=*/conv1_t, CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &conv_algo));
//Allocating workspace memory
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(/*handle=*/cudnnHandle,
        /*xDesc=*/input_tensor, /*wDesc=*/convfilter, /*convDesc=*/conv, /*yDesc=*/conv1_t, conv_algo, &workspace_bytes));
void* d_workspace{nullptr};
cudaMalloc(&d_workspace, workspace_bytes);
int image_bytes = batch_size*channels*height*width*sizeof(float);
//Pointer to input data
float* d_input{nullptr};
cudaMalloc(&d_input, image_bytes);
cudaMemcpy(d_input, image.ptr<float>(0), image_bytes, cudaMemcpyHostToDevice);
//Pointer to feature maps
float* conv1_ptr{nullptr};
cudaMalloc(&conv1_ptr, image_bytes);
cudaMemset(conv1_ptr, 0, image_bytes);
```

cuDNN: Convolution cont.



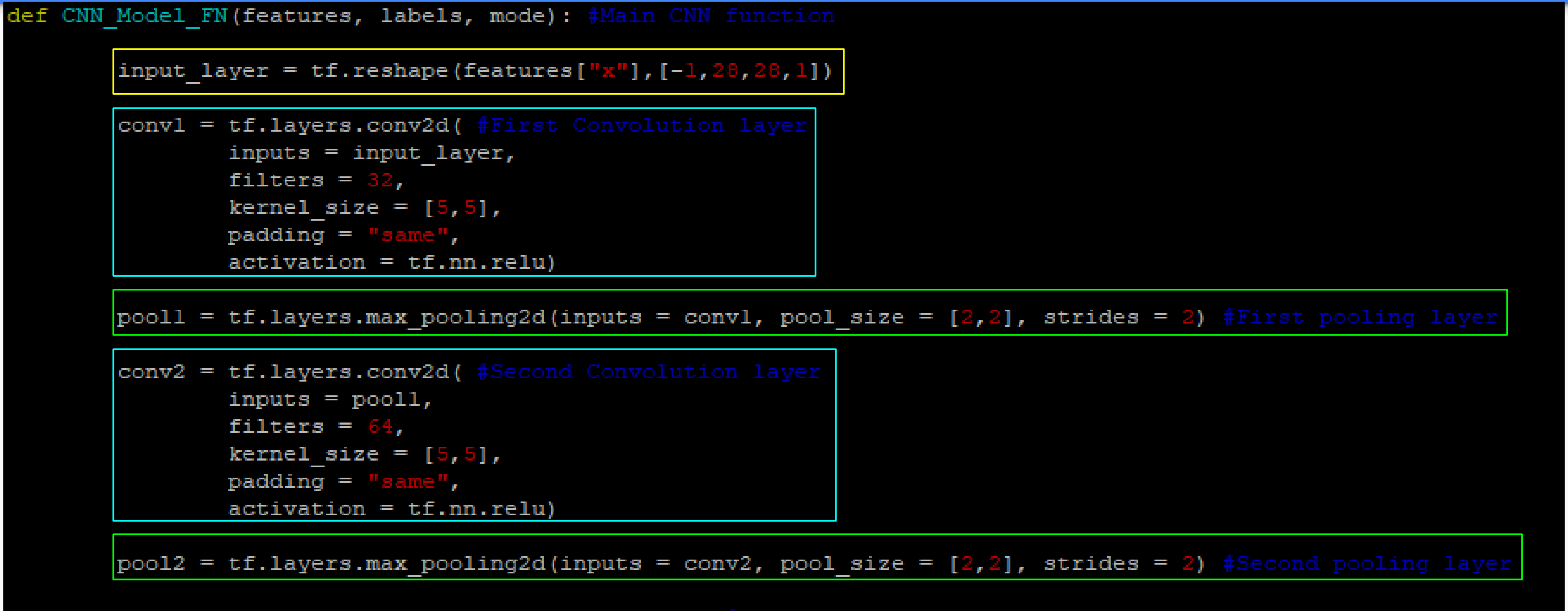
```
//Allocating workspace memory
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(/*handle=*/cudnnHandle,
        /*xDesc=*/input_tensor, /*wDesc=*/convfilter, /*convDesc=*/conv, /*yDesc=*/conv1_t, conv_algo, &workspace_bytes));
void* d_workspace{nullptr};
cudaMalloc(&d_workspace, workspace_bytes);
int image_bytes = batch_size*channels*height*width*sizeof(float);
//Pointer to input data
float* d_input{nullptr};
cudaMalloc(&d_input, image_bytes);
cudaMemcpy(d_input, image.ptr<float>(0), image_bytes, cudaMemcpyHostToDevice);
//Pointer to feature maps
float* conv1_ptr{nullptr};
cudaMalloc(&conv1_ptr, image_bytes);
cudaMemset(conv1_ptr, 0, image_bytes);
//Conv1 kernels initialized on Host
const float kernel_template[LAYER1_SIZE][K_ROWS][K_COLS] =
{
        { {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1}, {1, 8, 8, 8, 1}, {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1} },
        { {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1}, {1, 8, 8, 8, 1}, {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1} },
        { {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1}, {1, 8, 8, 8, 1}, {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1} },
        { {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1}, {1, 8, 8, 8, 1}, {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1} },
        { {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1}, {1, 8, 8, 8, 1}, {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1} },
        { {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1}, {1, 8, 8, 8, 1}, {1, 1, 1, 1, 1}, {1, 1, 1, 1, 1} }
};
float h_kernel[LAYER1_SIZE][K_ROWS][K_COLS];
for (int kernel = 0; kernel < LAYER1_SIZE; ++kernel) {
        for (int row = 0; row < K_ROWS; ++row) {
                for (int column = 0; column < K_COLS; ++column) {
                        h_kernel[kernel][row][column] = kernel_template[LAYER1_SIZE][row][column];
                }
        }
}
float* d_kernel{nullptr};
cudaMalloc(&d_kernel, sizeof(h_kernel));
cudaMemcpy(d_kernel, h_kernel, sizeof(h_kernel), cudaMemcpyHostToDevice);
//Convolution
checkCUDNN(cudnnConvolutionForward(cudnnHandle,
        &alpha, input_tensor, d_input, convfilter, d_kernel,
        conv, conv_algo, d_workspace, workspace_bytes,
        &beta, conv1_t, conv1_ptr));
```

cuDNN: Activation and Pooling



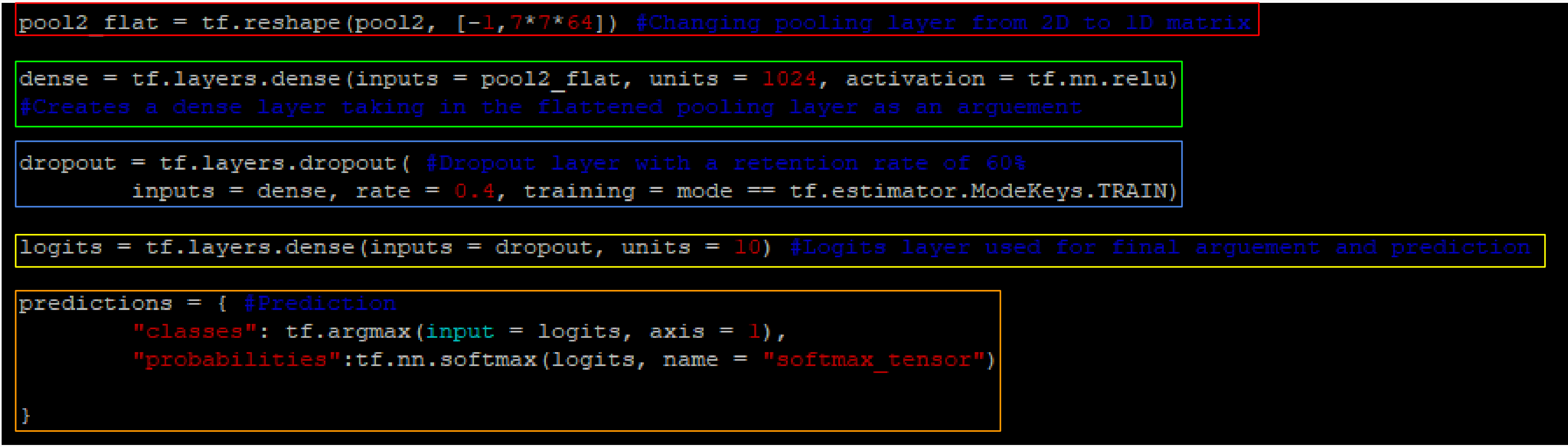
```
/*======================================================== First Activation ========================================================*/
        cudnnActivationDescriptor_t relu_activation;
        checkCUDNN(cudnnCreateActivationDescriptor(&relu_activation));
        checkCUDNN(cudnnSetActivationDescriptor(relu_activation,
                CUDNN_ACTIVATION_RELU, CUDNN_PROPAGATE_NAN, 0));
        checkCUDNN(cudnnActivationForward(cudnnHandle,
                relu_activation, &alpha, conv1_t, conv1_ptr, &beta, conv1_t, conv1_ptr));

/*======================================================== MAXPOOL LAYER 1 ========================================================*/
        //Input [Previously created conv1_t]
        //Output tensor(s) [6 feature maps = 6 maxpools]
        cudnnTensorDescriptor_t pool1_t;
        checkCUDNN(cudnnCreateTensorDescriptor(&pool1_t));
        checkCUDNN(cudnnSetTensor4dDescriptor(pool1_t,
                CUDNN_TENSOR_NHWC, CUDNN_DATA_FLOAT, 1, 1, height, width));
        //Pooling Description [Same for both Maxpool Layers]
        cudnnPoolingDescriptor_t pool_desc;
        checkCUDNN(cudnnCreatePoolingDescriptor(&pool_desc));
        checkCUDNN(cudnnSetPooling2dDescriptor(pool_desc,
                CUDNN_POOLING_MAX, CUDNN_PROPAGATE_NAN, 2, 2, 2, 2, 2, 2));
        //Dimensions after pooling [Only needs to be done once]
        checkCUDNN(cudnnGetPooling2dForwardOutputDim(pool_desc,
                conv1_t, &batch_size, &channels, &height, &width));
        //Memory size after maxpool [Resized]
        image_bytes = batch_size*channels*height*width*sizeof(float);
        float* pool1_ptr = {nullptr};
        cudaMalloc(&pool1_ptr, image_bytes);
        cudaMemset(pool1_ptr, 0, image_bytes);
        checkCUDNN(cudnnPoolingForward(cudnnHandle,
                pool_desc, &alpha, conv1_t, conv1_ptr,
                &beta, pool1_t, pool1_ptr));
```

TensorFlow: Filtering and Pooling



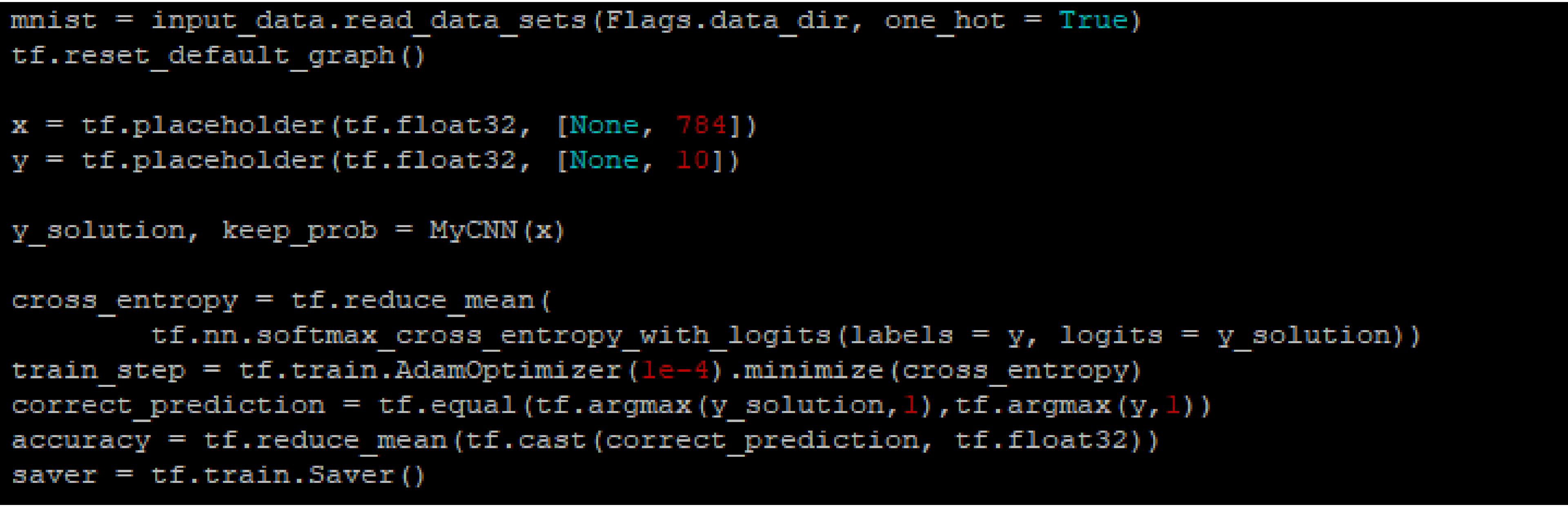
```
def CNN_Model_FN(features, labels, mode): #Main CNN function

        input_layer = tf.reshape(features["x"],[-1,28,28,1])

        conv1 = tf.layers.conv2d( #First Convolution layer
                inputs = input_layer,
                filters = 32,
                kernel_size = [5,5],
                padding = "same",
                activation = tf.nn.relu)

        pool1 = tf.layers.max_pooling2d(inputs = conv1, pool_size = [2,2], strides = 2) #First pooling layer

        conv2 = tf.layers.conv2d( #Second Convolution layer
                inputs = pool1,
                filters = 64,
                kernel_size = [5,5],
                padding = "same",
                activation = tf.nn.relu)

        pool2 = tf.layers.max_pooling2d(inputs = conv2, pool_size = [2,2], strides = 2) #Second pooling layer
```

TensorFlow: Flattening and Predicting



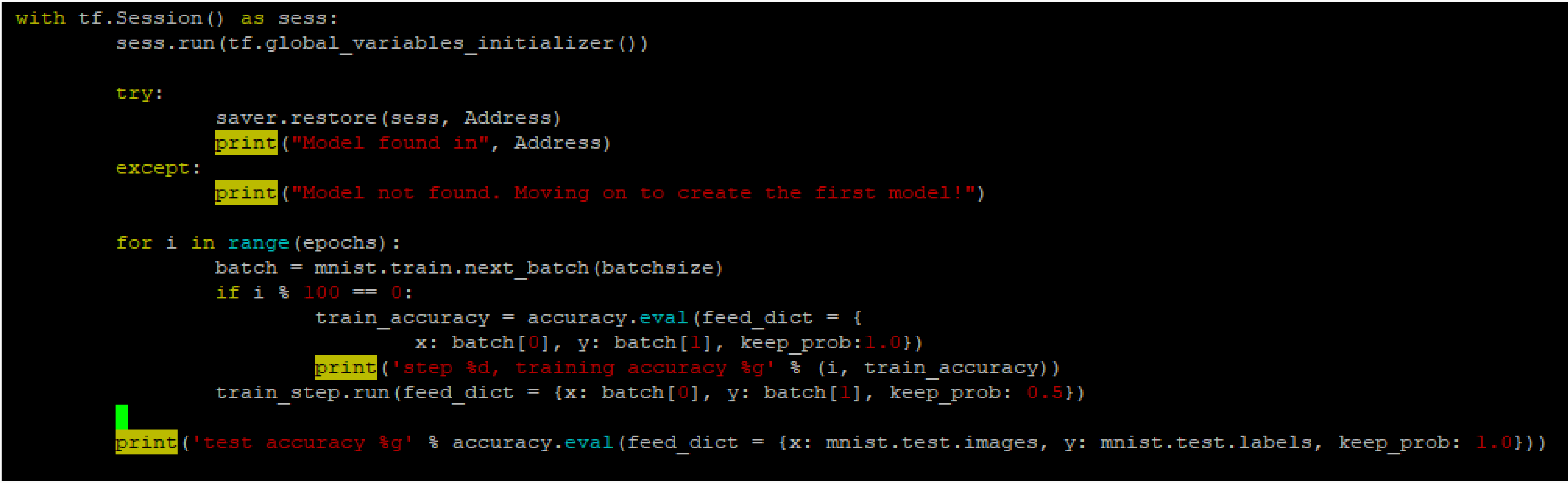
```
pool2_flat = tf.reshape(pool2, [-1,7*7*64]) #Changing pooling layer from 2D to 1D matrix

dense = tf.layers.dense(inputs = pool2_flat, units = 1024, activation = tf.nn.relu)
#Creates a dense layer taking in the flattened pooling layer as an arguement

dropout = tf.layers.dropout( #Dropout layer with a retention rate of 60%
        inputs = dense, rate = 0.4, training = mode == tf.estimator.ModeKeys.TRAIN)

logits = tf.layers.dense(inputs = dropout, units = 10) #Logits layer used for final arguement and prediction

predictions = { #Prediction
        "classes": tf.argmax(input = logits, axis = 1),
        "probabilities":tf.nn.softmax(logits, name = "softmax_tensor")

}
```

TensorFlow: Loading MNIST and training



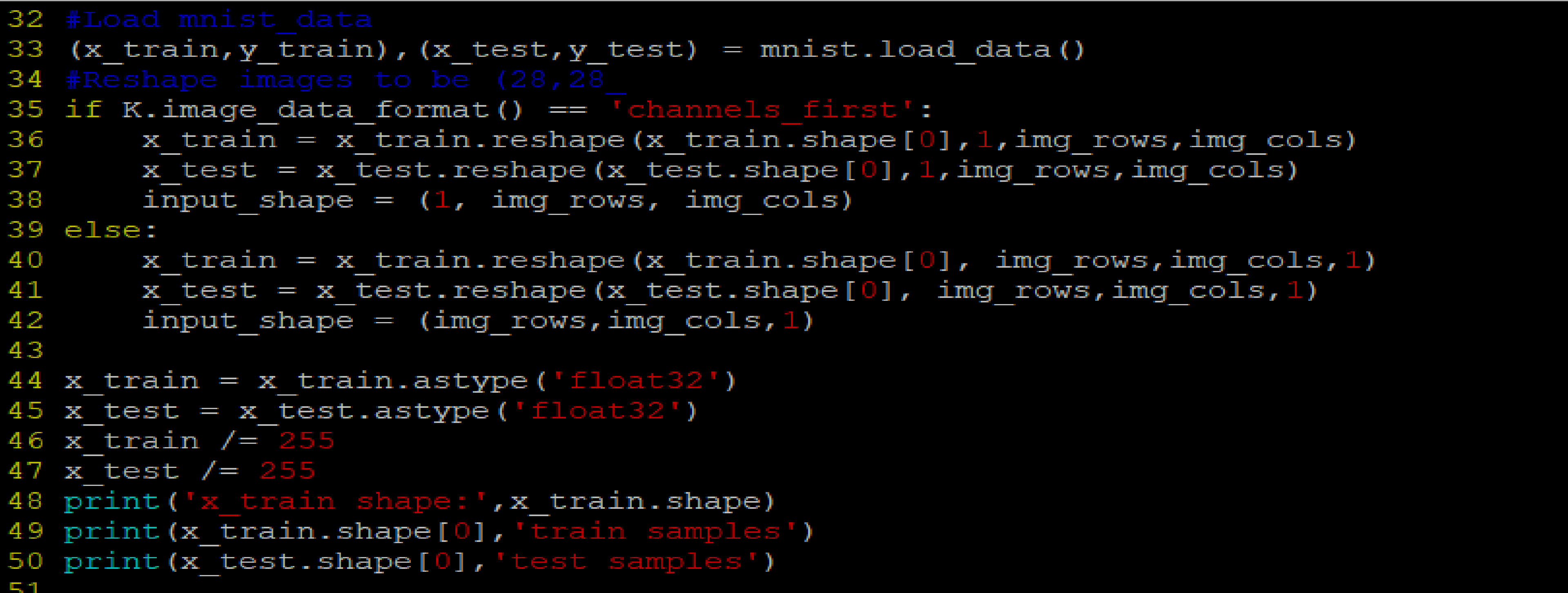
```
mnist = input_data.read_data_sets(Flags.data_dir, one_hot = True)
tf.reset_default_graph()

x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])

y_solution, keep_prob = MyCNN(x)

cross_entropy = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits(labels = y, logits = y_solution))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_solution,1),tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
saver = tf.train.Saver()
```

TensorFlow: Training Program



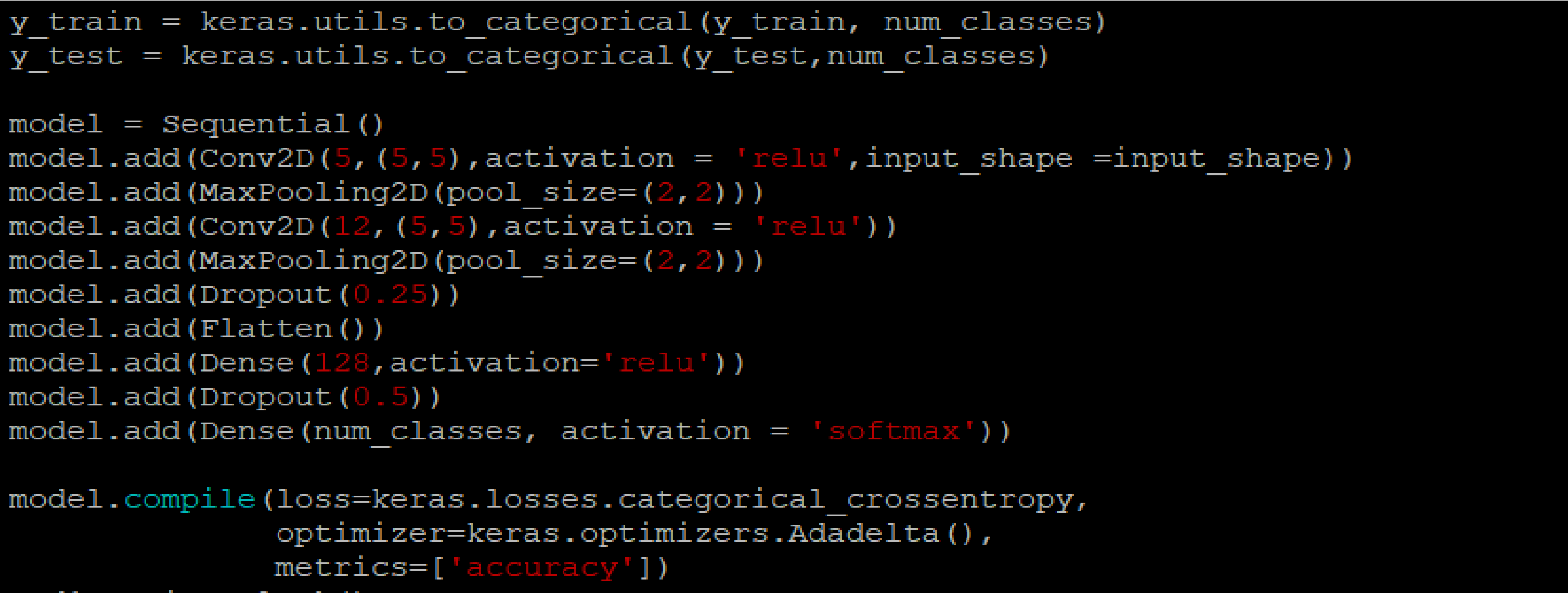
```
with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        try:
                saver.restore(sess, Address)
                print("Model found in", Address)
        except:
                print("Model not found. Moving on to create the first model!")

        for i in range(epochs):
                batch = mnist.train.next_batch(batchsize)
                if i % 100 == 0:
                        train_accuracy = accuracy.eval(feed_dict = {
                                x: batch[0], y: batch[1], keep_prob:1.0})
                        print('step %d, training accuracy %g' % (i, train_accuracy))
                train_step.run(feed_dict = {x: batch[0], y: batch[1], keep_prob: 0.5})

        print('test accuracy %g' % accuracy.eval(feed_dict = {x: mnist.test.images, y: mnist.test.labels, keep_prob: 1.0}))
```

Keras: Loading Data



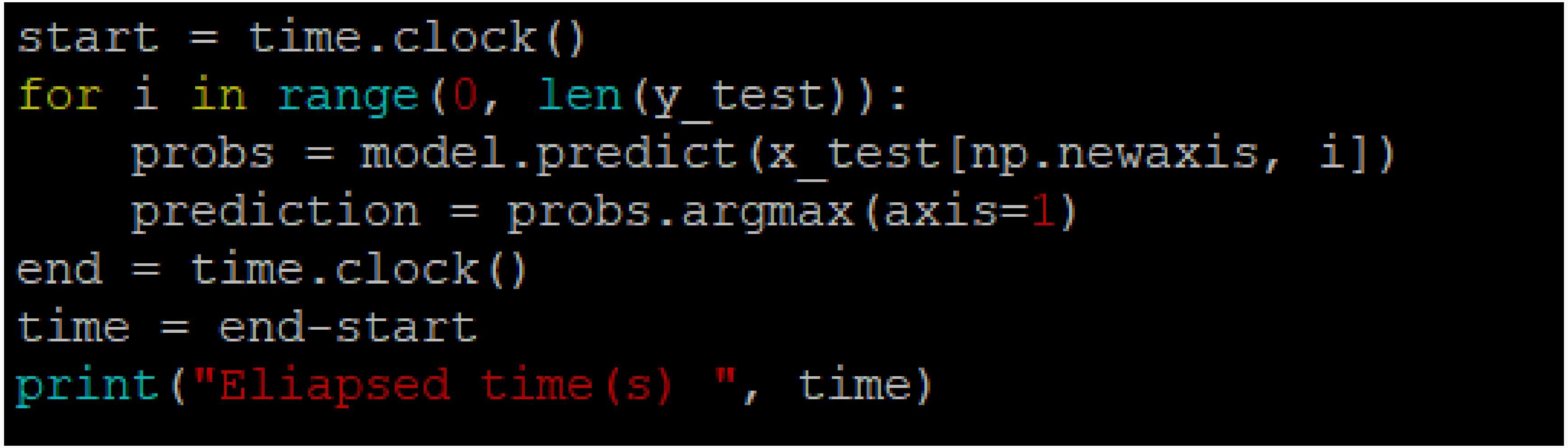
```python
#Load mnist data
(x_train,y_train),(x_test,y_test) = mnist.load_data()
#Reshape images to be (28,28
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0],1,img_rows,img_cols)
    x_test = x_test.reshape(x_test.shape[0],1,img_rows,img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows,img_cols,1)
    x_test = x_test.reshape(x_test.shape[0], img_rows,img_cols,1)
    input_shape = (img_rows,img_cols,1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:',x_train.shape)
print(x_train.shape[0],'train samples')
print(x_test.shape[0],'test samples')
```

Keras: Convolution and Pooling



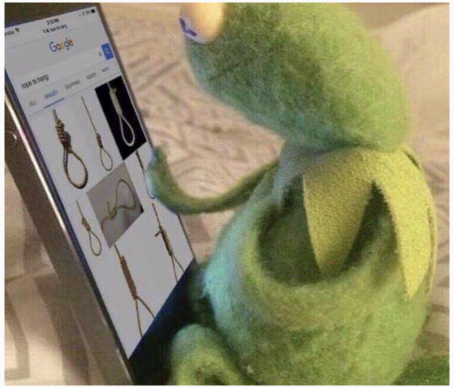
```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test,num_classes)

model = Sequential()
model.add(Conv2D(5,(5,5),activation = 'relu',input_shape =input_shape))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(12,(5,5),activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128,activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation = 'softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

Keras: Prediction

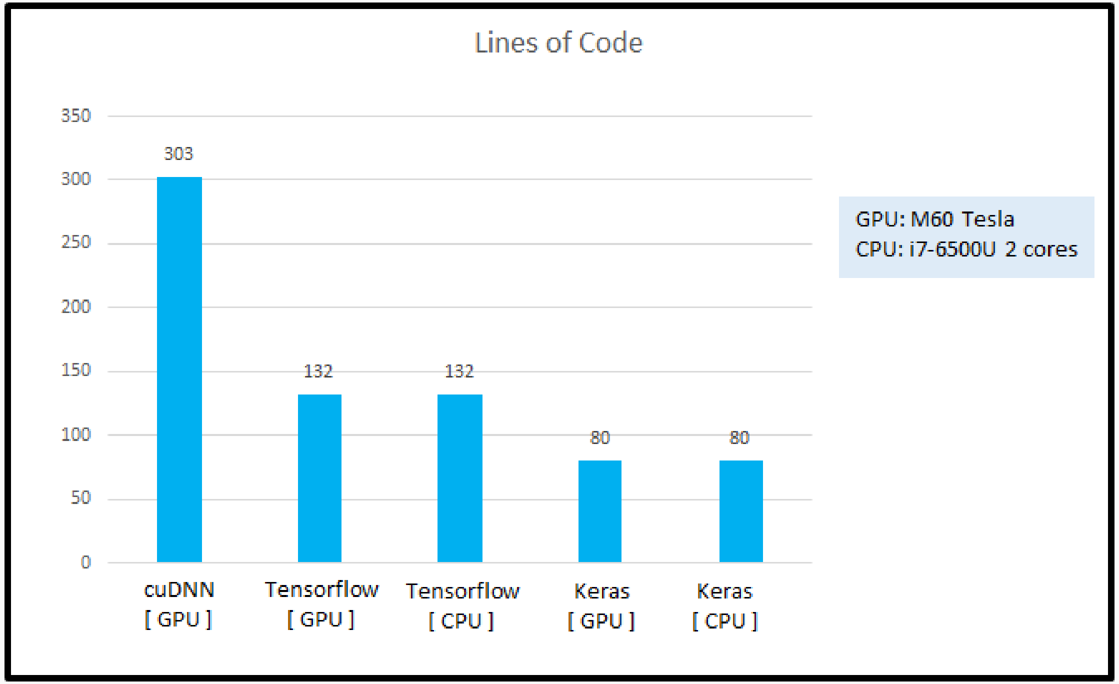


```
start = time.clock()
for i in range(0, len(y_test)):
    probs = model.predict(x_test[np.newaxis, i])
    prediction = probs.argmax(axis=1)
end = time.clock()
time = end-start
print("Eliapsed time(s) ", time)
```

Discussion: PROs and CONs of each API



Code Length



The higher level you go in APIs, the easier code is to pickup and learn.

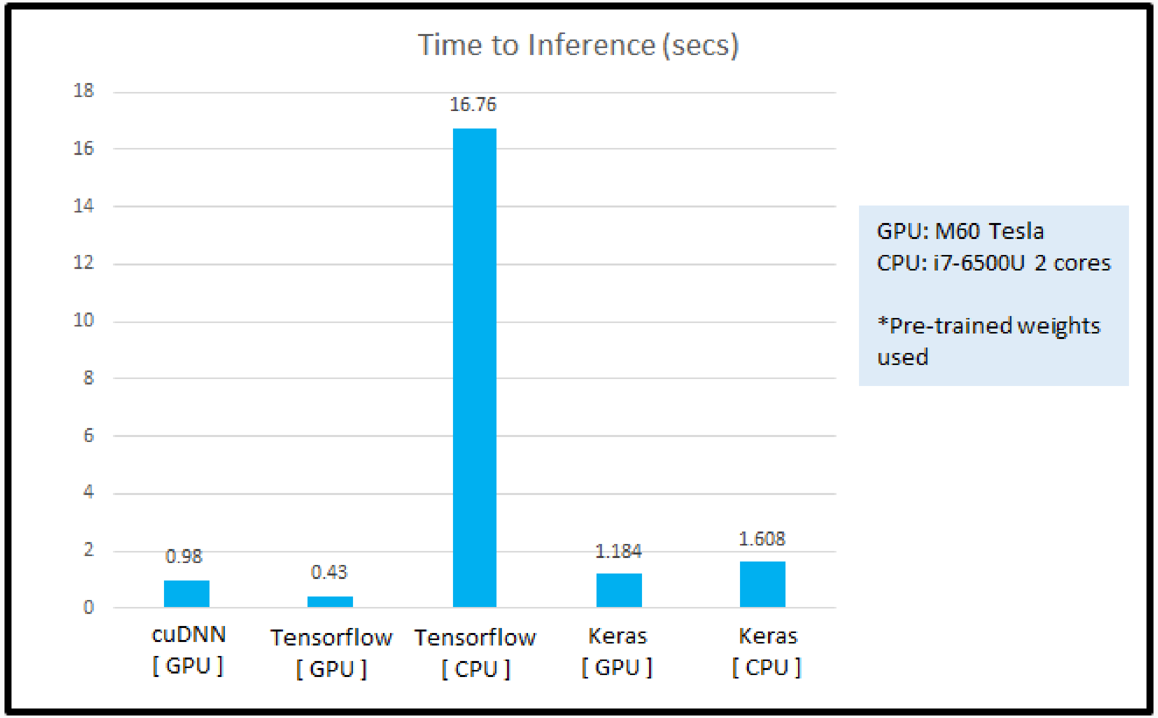
Overall code length is also shortened in higher-level implementations.

Time to Inference

An interesting result!

According to our data the highest-level API took the longest to classify amongst the code tested on a GPU.

Why might this be?
> Lack of optimization in Keras?
> Superb optimization in Tensorflow?
> Naive implementation on our part?
> All very confusing



GPU: M60 Tesla
CPU: i7-6500U 2 cores

*Pre-trained weights used

Problems and Challenges

Steep learning curve

Server side problems

<u>!!!Being an EE-major!!!</u>

Poor documentation/examples

Questions?

EE147: GPU Computing and Programming

PERFORMANCE COMPARISONS OF VARIOUS NEURAL NETWORK FRAMEWORKS

Jack Gu (SSID:861203821)
Joseph Gozum (SSID:861201282)
Justin Lam (SSID:861208924)

Contents

PROJECT IDEA/OVERVIEW

Background

Machine learning is a technique developed for computers to do what is natural to all animals, to learn from our experiences. This amazing process can be easily implemented with various APIs in wide variety of languages. APIs in recent years have become far more abstracted even leading to one API acting as a backend for another. But what benefits and costs come of using a higher API than the ones in use as backends? What do we gain from using Keras over Tensorflow over using cuDNN in terms of performance.

Project Overview

From the lowest-level (cuDNN) to increasingly higher-level frameworks such as TensorFlow and eventually Keras, we observe the possible increase in overhead as well as compare trade-offs between the aforementioned progamming APIs. The main metric observed betwen the three APIs will be of time to infer an input image. However, a look at difficulty in implementation as well as the tradeoff in control offered will also be discussed between the varying levels. We will mostly ignore the differences between compiled and interpreted languages but it will be noted.

IMPLEMENTATION DETAILS AND DOCUMENTATION

General specifications

For all three programs, we had the goal of training our neural network with the MNIST training data. This data set consists of images of handwritten single digit numbers, ranging from 0 to 9. For cuDNN, since there is no training algorithm implemented, we used the MNIST weights extracted from an online source. We also used these weights across the programs as they came in a standard [.hdf5] format.

cuDNN

Utilizing the cuDNN API was the most involved of the frameworks because of how low-level it is in comparison. While in the higher-level APIs, there were simple one line functions that initialized all the requirements and structures, coding at this level required individual initialization of all inputs, output, kernels, algorithms used, memory sizes, knowledge of data movement in non-unified memory, memory allocation. Any possible variable that could be manipulated needed to be stated before any convolution or pooling could occur.

TensorFlow

The implementation of Tensorflow was much easier than that of cuDNN. Instead of having to set up each individual filters, Tensorflow allocates the weights to each filter after training the neural network itself. Pooling and rectified linear operations (Creating rectified linear units AKA Relu) can also be done automatically with Tensorflow syntax. However, connecting the layers must still be done manually and individually between all layers. Extracting results are more complex, as the syntax to retrieve inference results as well as to even send an inference are complex in nature. As for the training and weights needed for a neural network, Tensorflow's syntax makes it possible for training to only be needed once, with syntax allowing the weights to be saved and automatically loaded in future activations of the program. Training is also customizable with the epochs (number of training iterations) customizable.

Keras

For Keras, everything was much simpler to set up with loading data taking one line and setting up multiple convolution layers in one line. Keras' high level made it extremely efficient in terms of code length. For training and testing the weights, once the code was all compiled for the model, we just had to optimize and let it run through the epochs and save the values we obtained.

EVALUATION/RESULTS

Performance

At the bottom of the paragraph (Fig. 2) is a histogram comparing computation time needed for a single inference in each of the APIs, as well as a comparison with it's CPU counterpart (Tensorflow and Keras are naturally optimized for GPU usage). Unsurprisingly, the Tensorflow and Keras programs on the GPU outperformed each of its respective CPU counterpart. In fact, the performance difference between the CPU and GPU version of Tensorflow was so vast that it just shows how Tensorflow was made for GPU usage. But, out of all three APIs, Keras actually performed the worst when it came down to running inferences. So our initial thought was that higher level APIs trade off performance for simplicity. However, cuDNN did not perform as well as Tensorflow. The poor performance on cuDNN most likely stems from a naive implementation and could be optimized. Further searching for an explanation seems to suggest that the reason for this is that Keras does not optimize Tensorflow that well as a backend API. Our initial theory before looking at the results of our inferences was that the API used did not matter as we had thought that each API was well optimized to the point where the only thing different between our programs would have been syntax/language difficulty.

Implementation

Implementation itself increased exponentially in difficulty the more you went down the API ladder. As shown in our histogram (Fig. 1), a lot more time and understanding of Convolutional Neural Networks was needed to implement the neural network the lower we went down the API rabbit hole, the more we needed to program. This brings up the argument of control vs execution difficulty. For example, in cuDNN, we have more control of the memory allocated as well as each individual filter. This makes it ideal for situations where we have hardware constraints such as memory. However, as you can ask Joseph, the tedious work makes it hard to execute. This brings up the argument of control vs simplicity of execution.

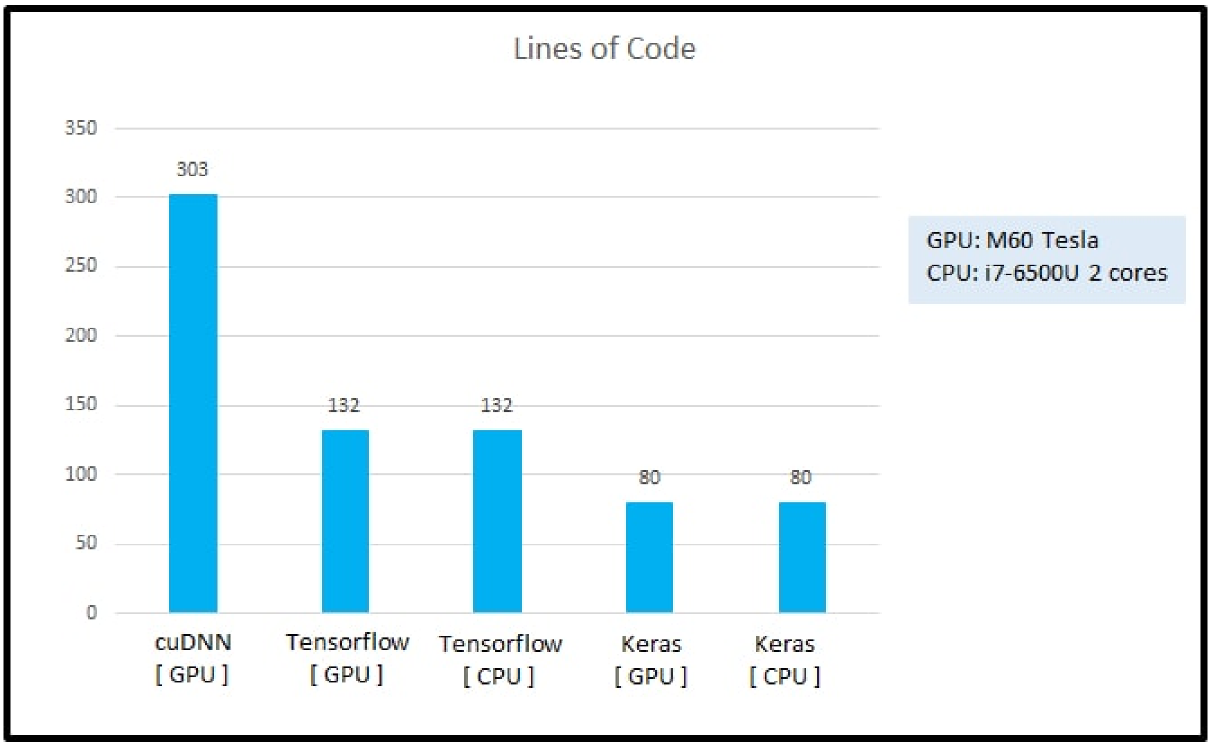


Figure 1: Lines of Code

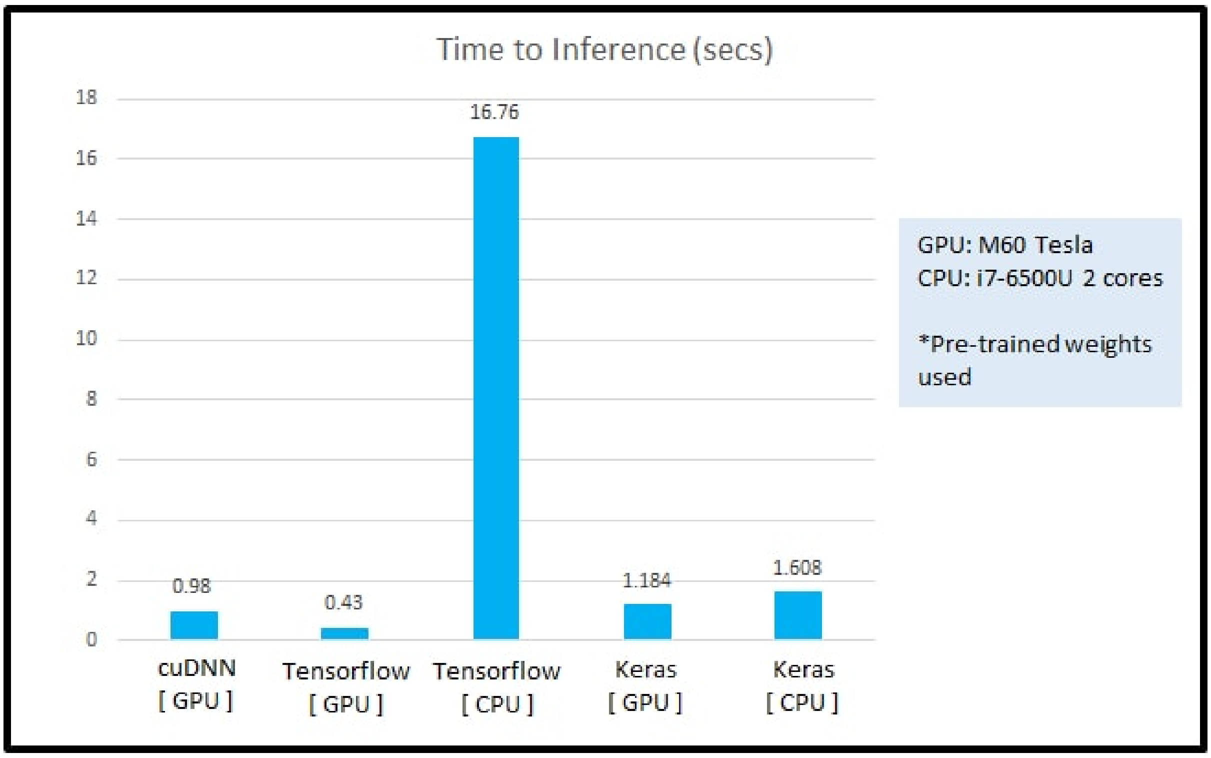


Figure 2: Processing Time

RUNNING THE CODE(S)

cuDNN

```
./lenet <filepath-to-test-image>
```

Tensorflow

```
python lenet-tensorflow.py
```

Keras

```
python lenet-keras.py
```

PROBLEMS FACED

Execution Issues

There were several issues that we encountered along the way: This was Jack and Justin's first time learning how to code in Python and most of the time it consisted of using Google and stackoverflow. There was a steep learning curve as we gradually move down to the lower frameworks. When other people were also doing their projects within the same server, sometimes we would hit server issues where our code would get stuck and couldn't run.

For the cuDNN implementation, for the fully connected layer it's required to use either personally written code or as suggested by Nvidia use their proprietary cuBLAS library. There were problems getting the input tensors into the expected form for cuBLAS to work and as of right now it has been commented out. So the CNN is only implemented fully up to the pre-FC layer.

Library issues for cuDNN/cuBLAS

We did not install our necessary libraries or programs ourselves nor did we have administrative access to the GPU. The GPU was housed in a school server so we had no idea were libraries where they were installed or how they were configured.

Lack of documentation

This was a problem only for cuDNN, there is documentation on what the framework's functions output and what arguments they require but how you put them together is not initially clear. There exists very few examples of how to use the API.

Lack of background knowledge

For Jack and Justin, this was the first time that they had been exposed to CNNs and deep learning in general. This lead to a trial by fire where they had to develop a basic understanding very quickly in a short amount of time, at least enough to create a neural network using their designated frameworks in Python.

Inconveniences

Not exactly a problem, but because we did testing and everything on the server we do not have free access to download libraries that would allow us to use convenient functions to do things like download MNIST, showcase images from the terminal, functions that would allow us to focus more on the actual CNN creation and testing.