Introduction to Deep Learning with Keras

Acknowledgment

This tutorial was adapted from The Ultimate Beginner's Guide to Deep Learning in Python (https://elitedatascience.com/keras-tutorial-deep-learning-in-python) from EliteDataScience. It is a hands-on "walk through" describing how to build a deep Convolutional Neural Network (CNN) and use it to perform handwritten digit recognition.

The tutorial has been rewritten and modified to:

- use Python 3 instead of Python 2
- use the Tensorflow backend instead of Theano
- use a GPU device instead of the CPU.

Background

Keras

Keras is a high-level deep learning modeling library. It's simple, extensible, and Pythonic. Keras provides building blocks implemented in several lower-level frameworks (i.e. backends) for quicker development of deep learning models. At this time, Keras supports three backend engines: Tensorflow, Theano, and CNTK.

- Tensorflow (http://www.tensorflow.org/) is an open-source symbolic tensor manipulation framework developed by Google.
- Theano (http://deeplearning.net/software/theano/) is an open-source symbolic tensor manipulation framework developed at Université de Montréal.
- CNTK (https://www.microsoft.com/en-us/cognitive-toolkit/) is an open-source toolkit for deep learning developed by Microsoft.

See https://keras.io (https://keras.io) for further details on code found in this tutorial.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are deep, multi-layer neural networks well-suited for processing image data. CNNs often exhibit excellent classification performance, while significantly reducing the number of parameters that need to be tuned for learning. They are designed to handle the high dimensionality of raw image/video data.

Seven Steps to Deep Learning

1. Setup the environment
2. Load libraries, modules, and data

3. Pre-process input data and class labels
4. Define the neural network architecture
5. Configure the learning process (Compile)
6. Train the model (Fit)
7. Test the model (Evaluate)

Note: in order to take advantage of Tensorflow's GPU capabilities, this lab tutorial is designed to be run on **ghost** (the CIS machine learning server in the DEN). However, the instructions should be applicable to any Ubuntu-based system with adequate hardware.

1. Setup the environment

Software used in this tutorial:

- Python 3.6.7
- Scipy 1.2.1
- Numpy 1.16.2
- Graphviz 2.40.1-2
- Pydot 1.4.1
- Tensorflow 1.12.0 (with package `tensorflow-gpu`)

This tutorial assumes that you have already completed the virtual environment Setup.

1.1 Validate/Configure Keras

Confirm that a `.keras` directory is present in your **ghost** home directory. It contains the datasets used in this tutorial and the `keras.json` configuration file. The config file defines datatypes, constants, and specifies which lower-level tensor manipulation library to use. It should look like this:

```
$ cat $HOME/.keras/keras.json
{
 "floatx": "float32",
 "epsilon": 1e-07,
 "backend": "tensorflow",
 "image_data_format": "channels_last"
}
```

1.2 The virtual environment

In order to facilitate running the tutorial, a virtual environment should have been created in your home directory on **ghost** (as a result of completing the Setup step). Virtual environments help manage the various software dependencies that different projects may require.

If correctly configured, the virtual environment should contain all packages needed to run this tutorial. The environment was created using `virtualenv`, a tool for managing local python package installations for individual users without impacting the system-wide python installation or any other users.

To use the virtual environment, *activate* it with the `source` command. Change into the `dltut` directory and enter the following command:

```
<user>@ghost:~/dltut $ source bin/activate
(dltut) <user>@ghost:~/dltut $
```

You should see output similar to that shown above. The `(dltut)` in parentheses in front of the prompt indicates that the virtual environment (named 'dltut') is active. Note that your original file system is still in place; the virtual environment only controls the context in which your programs execute.

To exit the virtual environment use `deactivate`:

```
(dltut) <user>@ghost:~/dltut $ deactivate
<user>@ghost $
```

The parentheses and name of the virtual environment disappear.

1.3 Validate Tensorflow

Activate your virtual environment. Confirm that tensorflow_gpu is operational by executing the following python command:

```
$ python -c "import tensorflow as tf; tf.enable_eager_execution(); print(tf.reduce_sum(tf.random_normal([1000, 10
00])))"
```

You'll see a lot of output, but the last few lines should look something like this:

```
2019-03-20 17:59:19.692930: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/
job:localhost/replica:0/task:0/device:GPU:0 with 10976 MB memory) -> physical GPU (device: 0, name: TITAN V, pci
bus id: 0000:0b:00.0, compute capability: 7.0)

2019-03-20 17:59:19.693389: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/
job:localhost/replica:0/task:0/device:GPU:1 with 22729 MB memory) -> physical GPU (device: 1, name: Quadro P6000,
pci bus id: 0000:41:00.0, compute capability: 6.1)

tf.Tensor(1599.2335, shape=(), dtype=float32)
```

This output indicates that tensorflow is installed and operating correctly. It also indicates that two Nvidia GPUs are available:

- device 0, a TITAN V (https://www.nvidia.com/en-us/titan/titan-v/)
- device 1, a Quadro P6000 (https://www.nvidia.com/object/quadro-graphics-with-pascal.html)

GPUs are massively-parallel systems (the Titan has 5120 processing cores, the Quadro has 3840 cores). The large number of cores are exploited to execute floating-point operations in parallel, greatly reducing the execution time required for training a neural network.

1.4 Target a GPU

To specify that tensorflow should use a specific GPU, adjust the `CUDA_VISIBLE_DEVICES` environment variable. This can be done from the shell (i.e. `bash`) or from within your Python 3 session. The tutorial has been verified to run on either GPU; but the Titan (device 0) is the default.

```
# from the shell (set one of these before starting Python)
$ export CUDA_VISIBLE_DEVICES=0 # TITAN V
$ export CUDA_VISIBLE_DEVICES=1 # Quadro P6000

# from within a Python session or script (set one of these early in the session/script)
import os
os.environ["CUDA_VISIBLE_DEVICES"]="0"
os.environ["CUDA_VISIBLE_DEVICES"]="1"
```

2. Load libraries, modules, and data

What follows is a line-by-line walk-through of the python code needed to perform the digit recognition task. My suggestion is that you activate your virtual environment in a console session, startup Python, specify a GPU, and copy and paste each of the following lines of code individually at the prompt. Then read along, observing (and understanding) each step in the process.

Note: for later use, the complete code listing can be found at the end of this tutorial in Appendix A.

Libraries and modules

First, import `numpy` and set a seed value for the pseudo-random number generator.

```
import numpy as np
np.random.seed(42)
```

Next, import the Sequential model type from Keras.
This will allow you to linearly "stack" multiple neural network layers to easily build a feed-forward CNN.

```
from keras.models import Sequential
# Using TensorFlow backend.
```

Now import the core Keras layers.
These are commonly-found layers used in most neural network models.

```
from keras.layers import Dense, Dropout, Activation, Flatten
```

Then import the Convolutional layers used for this application.

```
from keras.layers import Conv2D, MaxPooling2D
```

Finally, import some utilities which will be useful for transforming the data.

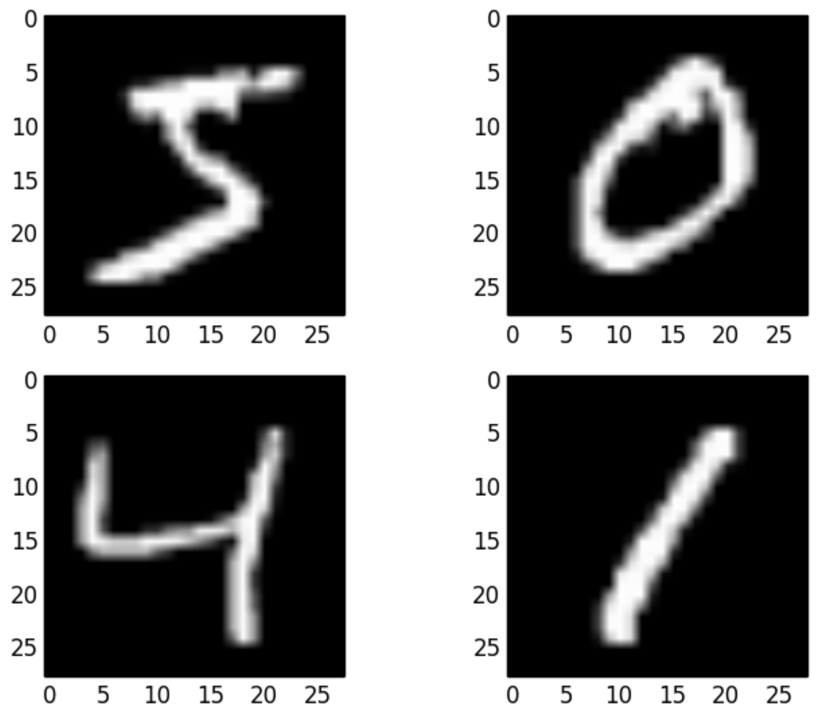
```
from keras.utils import np_utils
```

Data

Now that the libraries and modules have been made available, it's time to load the data.

Keras includes a copy of the MNIST hand-written digits dataset (http://yann.lecun.com/exdb/mnist/). It consists of a large set (70000 samples) of 28x28 pixel images of handwritten numbers used for classification testing.

This is what a few MNIST samples look like when loaded into an image viewer.



As can be surmised from the images, each sample consists of (28x28 = 784) pixels, plus the class label (e.g. '4').

The next step is to load the (pre-shuffled) image set and partition the data into Training (60000) and Test (10000) sets.

Note: The variable `X` (uppercase) refers to the input data. The variable `y` (lowercase) is used to hold the class labels (i.e. supervisor output). We will be adjusting `y` later.

```
from keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Examine the shape of the Training set. There are 60,000 samples, each with dimension 28x28.

```
print(X_train.shape)
# (60000, 28, 28)
```

3. Pre-process input data and class labels

Input data

Tensorflow requires an explicit declaration of a dimension for the color depth (channel) of the input image. Full-color images with 3 RGB channels have a depth of 3. The black and white MNIST images only have a depth of 1. This is communicated to tensorflow by transforming the shape of the dataset from `(60000, 28, 28)` to shape `(60000, 28, 28, 1)`.

When using Tensorflow, the channel depth dimension is specified last; this corresponds to the `image_data_format` configuration parameter defined in the `keras.json` file referenced in step 1.1 above. Reshape the data.

```
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
```

Confirm the new shape by re-printing the dimensions of the Training/Test sets.

```
print(X_train.shape)
# (60000, 28, 28, 1)
```

The input data consists of pixel intensities, represented as integer values ranging from 0 (black) to 255 (white). As mentioned in class, it's important to normalize (i.e. scale) the input data presented to a neural network (in order to avoid saturating the perceptrons). The final data transformation is to convert the data to **`float32`**, then normalize the values onto the range [0, 1].

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

Class labels

Recall that the variable `y` holds the class labels. Examine the shape of the class label data.

```
print(y_train.shape)
# (60000,)
```

This indicates a 1-dimensional array of class labels. But what we want is *k*=10 different class labels (one per digit). Inspecting (printing) the first few samples reveals the problem: the output data consists of a single class label that ranges from 0..9.

```
print(y_train[:5])
# [5 0 4 1 9]
```

Ultimately, our network will be designed to have *k*=10 output neurons, one per class (digit). The neuron whose output is maximized represents our predicted digit. But the way the class labels are organized suggests a single output neuron, whose value ranges from 0..9.

The desired transform can be easily performed by applying the function `to_categorical()` from `np_utils`, which converts an integer class array into a binary class matrix. The effect is to convert a single decimal class label into a binary one-hot vector.

Apply this transform to both the Training/Test class labels.

Note the use of upper-case `Y` for the new class label variables.

```
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

Re-examine the shape (remembering to use upper-case `Y`):

```
print(Y_train.shape)
# (60000, 10)
```

Re-inspect the first few samples (and compare to the listing above). The labels [5 0 4 1 9] have been transformed into:

```
print(Y_train[:5])
#[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

Notice that each sample consists of *k*=10 binary class labels (the desired output of our model).

The input data and class labels are now properly formatted and ready for model training.

4. Define the neural network architecture

Start by creating a sequential model.

```
model = Sequential()
```

The first layer to add to the model is always the input layer. Since the input data consists of 2-D spatial images, we begin by adding a 2-D Convolutional layer, comprised of 32 filters, each of which is 3x3 in size. We specify the activation function to be a Rectified Linear Unit (ReLU). Normally, Keras can deduce the required connections between adjacent layers based on their type and size (automatic shape inference).

However, for the first layer we have to specify the shape of the initial input -- a batch of 28x28x1 samples.

```
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', input_shape=(28,28,1)))
```

You can use `summary()` at any time to inspect your model.

```
model.summary()

_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_1 (Conv2D) (None, 26, 26, 32) 320
=================================================================
Total params: 320
Trainable params: 320
Non-trainable params: 0
_________________________________________________________________
```

The `None` in the first dimension of the Output Shape indicates this number is not defined. That's because it's the (variable) batch size. If you convolutionally apply a 3x3 filter to a 28x28 image, you get a 26x26 output. Since there are 32 filters the shape of the output for this layer is (26x26x32).

There are 3x3 inputs for each of the 32 Convolutional units, each with an additional bias input, which means there are (3 * 3 * 32 + 32 = 320) Parameters (i.e. weights) to learn in this layer.

It's simple to stack (add) additional hidden layers:

- add another 2-D Convolutional layer, to help detect higher-level features
- add a 2-D MaxPooling layer, to perform downsampling
- add a Dropout layer, to protect against overfitting

Keras provides the ability to quickly combine layers into custom model architectures, leaving the low-level implementation details to the backend framework (in this case, Tensorflow). Deep learning is as much art as it is science, and the ability to quickly iterate through different ideas is the Keras library’s key benefit.

```
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(rate=0.25))
```

[Note: you may get a few warnings; tensorflow is constantly upgrading.]

As used here, MaxPooling reduces the number of parameters in the model by sliding a 2x2 filter across the output from previous layer, at each point taking the maximum of the 4 values in the filter. The Dropout layer specifies that each neuron has a 25% chance of being "turned off" for a particular time step; preventing the network from using higher-order approximations to fit too tightly to the data.

Last, add a Dense (fully-connected) layer and a final Dense (output) layer.

```
model.add(Flatten())
model.add(Dense(units=128, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(units=10, activation='softmax'))
```

The output from the earlier 2-dimensional Convolutional layers must be "flattened" (made 1-dimensional) before passing it on to a Dense layer. Note that the final fully-connected layer has an output size of 10, corresponding to the 10 classes of digits. Note also that it uses the SoftMax activation function.

Examine the final configuration of the network.

```
model.summary()

_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_1 (Conv2D) (None, 26, 26, 32) 320

_________________________________________________________________
conv2d_2 (Conv2D) (None, 24, 24, 32) 9248

_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32) 0

_________________________________________________________________
dropout_1 (Dropout) (None, 12, 12, 32) 0

_________________________________________________________________
flatten_1 (Flatten) (None, 4608) 0

_________________________________________________________________
dense_1 (Dense) (None, 128) 589952

_________________________________________________________________
dropout_2 (Dropout) (None, 128) 0

_________________________________________________________________
dense_2 (Dense) (None, 10) 1290
=================================================================
Total params: 600,810
Trainable params: 600,810
Non-trainable params: 0

_________________________________________________________________
```

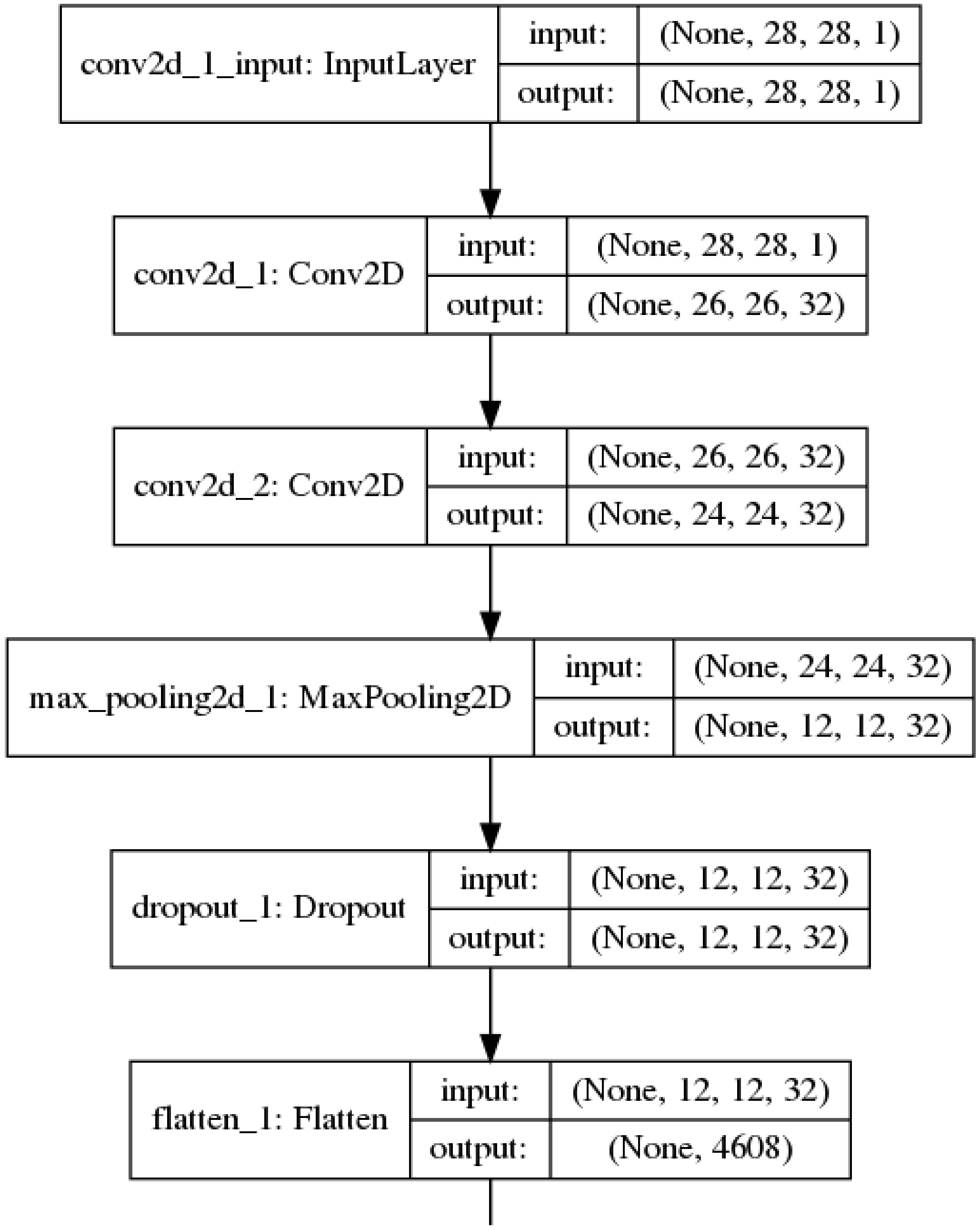
The second Convolutional layer, with 3x3 filters, again reduces the output shape -- to 24x24 (with 32 filters). The MaxPooling layer, using 2x2 filters, further reduces the output shape to 12x12 (with 32 filters). Flattening the data gives a 1-dimensionally shaped output of (12x12x32 = 4608). This is forwarded into Dense and Dropout layers with 128 units each, and finally to the output layer, with *k*=10 class units. To summarize, this network inputs an image of size 28x28 pixel intensities, and outputs 10 probability values which together represent its predicted classification.

Each of the 32 units in the second Convolutional layer receives 32 3x3 inputs, and an additional 32 bias inputs, for a total of (32 * 3 * 3 * 32 + 32 = 9248) parameters. The first Dense (fully-connected) layer inputs 4608 values into each of the 128 units + 128 bias inputs for a total of (4608 * 128 + 128 = 589952) parameters. The second Dense layer inputs 128 values into each of the 10 output units + 10 bias inputs for a total of (128 * 10 + 10 = 1290) parameters. Considering all of the hidden layers and bias units, the network will train 600,810 parameters (i.e. it will adjust that many edge weights and kernel parameters).

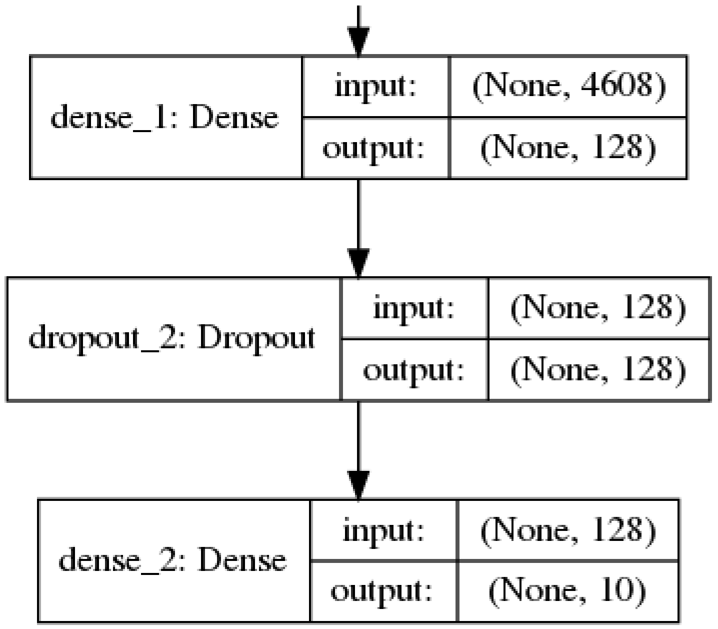
It's also possible to get a graphical representation of the model. The following utility functions will produce a graph, and output it to a file, which can then be downloaded for viewing.

```
from keras.utils.vis_utils import plot_model
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```

This is the graph of the network assembled in this tutorial:



conv2d_1_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
conv2d_1: Conv2D
input: (None, 28, 28, 1)
output: (None, 26, 26, 32)
conv2d_2: Conv2D
input: (None, 26, 26, 32)
output: (None, 24, 24, 32)
max_pooling2d_1: MaxPooling2D
input: (None, 24, 24, 32)
output: (None, 12, 12, 32)
dropout_1: Dropout
input: (None, 12, 12, 32)
output: (None, 12, 12, 32)
flatten_1: Flatten
input: (None, 12, 12, 32)
output: (None, 4608)



The visualization graphically illustrates the "stacking", or layering, of the hidden layers. It also gives a good idea of how the output from one layer feeds into the next, and what those outputs/inputs look like.

5. Configure the learning process (Compile)

Critical to model learning is the choice of the loss function (for measuring error), the optimization function used for reducing error (i.e. learning), and the metric used to evaluate the model. These parameters are all set using the `compile` method.

```
model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
```

The error, the difference between the predicted and actual value, is calculated by using the cross-entropy (log loss) function. This measure is appropriate when there is a probability associated with each class (category). The Adam optimizer is an enhancement/extension of stochastic gradient descent that uses a separate, adaptive learning rate for each parameter (weight). The Accuracy metric is just the % of correctly predicted labels.

The fully-configured model is now ready to use.

6. Train the model (Fit)

Training the model is initiated using the `fit` method. This is the most time-consuming step in the process. Required parameters are: the training data, the (mini)-batch size, the number of epochs for which to train, and an optional verbose flag for monitoring progress.

```
model.fit(X_train, Y_train, batch_size=32, epochs=10, verbose=1)
# Epoch 1/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0789 - acc: 0.9757
# Epoch 2/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0596 - acc: 0.9815
# Epoch 3/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0515 - acc: 0.9847
# Epoch 4/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0432 - acc: 0.9867
# Epoch 5/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0407 - acc: 0.9871
# Epoch 6/10
# 60000/60000 [==============================] - 9s 143us/step - loss: 0.0352 - acc: 0.9886
# Epoch 7/10
# 60000/60000 [==============================] - 8s 142us/step - loss: 0.0310 - acc: 0.9898
# Epoch 8/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0298 - acc: 0.9905
# Epoch 9/10
# 60000/60000 [==============================] - 8s 142us/step - loss: 0.0269 - acc: 0.9913
# Epoch 10/10
# 60000/60000 [==============================] - 8s 141us/step - loss: 0.0240 - acc: 0.9922
```

Note 1: Fitting the model will take about 90 seconds on the Titan V GPU, and about 100 seconds on the Quadro P6000. It takes more than 33 minutes on **ghost**'s CPU (which is a pretty high-end processor).

Note 2: Tensorflow grabs a lot of GPU memory while executing the training step. If you receive a "CUDA out-of-memory" error that means somebody else is also using that GPU. Wait just a minute, then try again; or try configuring your code to use the other GPU. (You can examine the status of either GPU by running the ' `nvidia-smi` ' utility in another terminal window.)

7. Test the model (Evaluate)

The final step is to score/evaluate the trained model on the Test dataset formatted earlier. Testing the model is initiated using the `evaluate` method, which stores results into the variable `score` for subsequent display. The first value given is the Test data loss measure, and the second is the accuracy metric.

```
score = model.evaluate(X_test, Y_test, verbose=1)
# 10000/10000 [==============================] - 0s 44us/step
score
# [0.03102649506003163, 0.9914]
```

So how did our model do?

99.1%, not bad!

And that concludes our walk-through of the steps typically involved in applying a high-level framework to a Deep Learning problem.

Appendix A: full code listing

```python
#!/usr/bin/env python3
#
# dltut.py
#

# optionally target a GPU
#import os
#os.environ["CUDA_VISIBLE_DEVICES"]="0"

import numpy as np
np.random.seed(42)

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist

# Split pre-shuffled MNIST data into training and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Reshape for Tensorflow (image depth)
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)

# Change data type and normalize
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# Convert 1-dimensional class arrays to 10-dimensional binary class matrices
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)

# Define model architecture
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(rate=0.25))
model.add(Flatten())
model.add(Dense(units=128, activation='relu'))
model.add(Dropout(rate=0.5))
model.add(Dense(units=10, activation='softmax'))

# Choose loss function, optimzer, and metrics
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fit with training data
model.fit(X_train, Y_train, batch_size=32, epochs=10, verbose=1)

# Score with test data
score = model.evaluate(X_test, Y_test, verbose=1)
print(score)
```