02_Convolutional_Neural_Network

October 18, 2017

1 TensorFlow Tutorial #02

2 Convolutional Neural Network

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube

2.1 Introduction

The previous tutorial showed that a simple linear model had about 91% classification accuracy for recognizing hand-written digits in the MNIST data-set.

In this tutorial we will implement a simple Convolutional Neural Network in TensorFlow which has a classification accuracy of about 99%, or more if you make some of the suggested exercises.

Convolutional Networks work by moving small filters across the input image. This means the filters are re-used for recognizing patterns throughout the entire input image. This makes the Convolutional Networks much more powerful than Fully-Connected networks with the same number of variables. This in turn makes the Convolutional Networks faster to train.

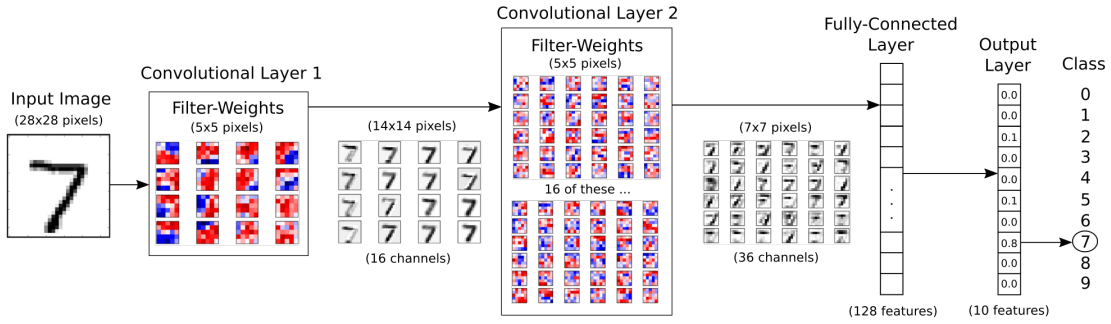
You should be familiar with basic linear algebra, Python and the Jupyter Notebook editor. Beginners to TensorFlow may also want to study the first tutorial before proceeding to this one.

2.2 Flowchart

The following chart shows roughly how the data flows in the Convolutional Neural Network that is implemented below.

```
In [1]: from IPython.display import Image
        Image('images/02_network_flowchart.png')
```

Out[1]:



The input image is processed in the first convolutional layer using the filter-weights. This results in 16 new images, one for each filter in the convolutional layer. The images are also down-sampled so the image resolution is decreased from 28x28 to 14x14.

These 16 smaller images are then processed in the second convolutional layer. We need filter-weights for each of these 16 channels, and we need filter-weights for each output channel of this layer. There are 36 output channels so there are a total of 16 x 36 = 576 filters in the second convolutional layer. The resulting images are down-sampled again to 7x7 pixels.

The output of the second convolutional layer is 36 images of 7x7 pixels each. These are then flattened to a single vector of length 7 x 7 x 36 = 1764, which is used as the input to a fully-connected layer with 128 neurons (or elements). This feeds into another fully-connected layer with 10 neurons, one for each of the classes, which is used to determine the class of the image, that is, which number is depicted in the image.

The convolutional filters are initially chosen at random, so the classification is done randomly. The error between the predicted and true class of the input image is measured as the so-called cross-entropy. The optimizer then automatically propagates this error back through the Convolutional Network using the chain-rule of differentiation and updates the filter-weights so as to improve the classification error. This is done iteratively thousands of times until the classification error is sufficiently low.

These particular filter-weights and intermediate images are the results of one optimization run and may look different if you re-run this Notebook.

Note that the computation in TensorFlow is actually done on a batch of images instead of a single image, which makes the computation more efficient. This means the flowchart actually has one more data-dimension when implemented in TensorFlow.

2.3 Convolutional Layer

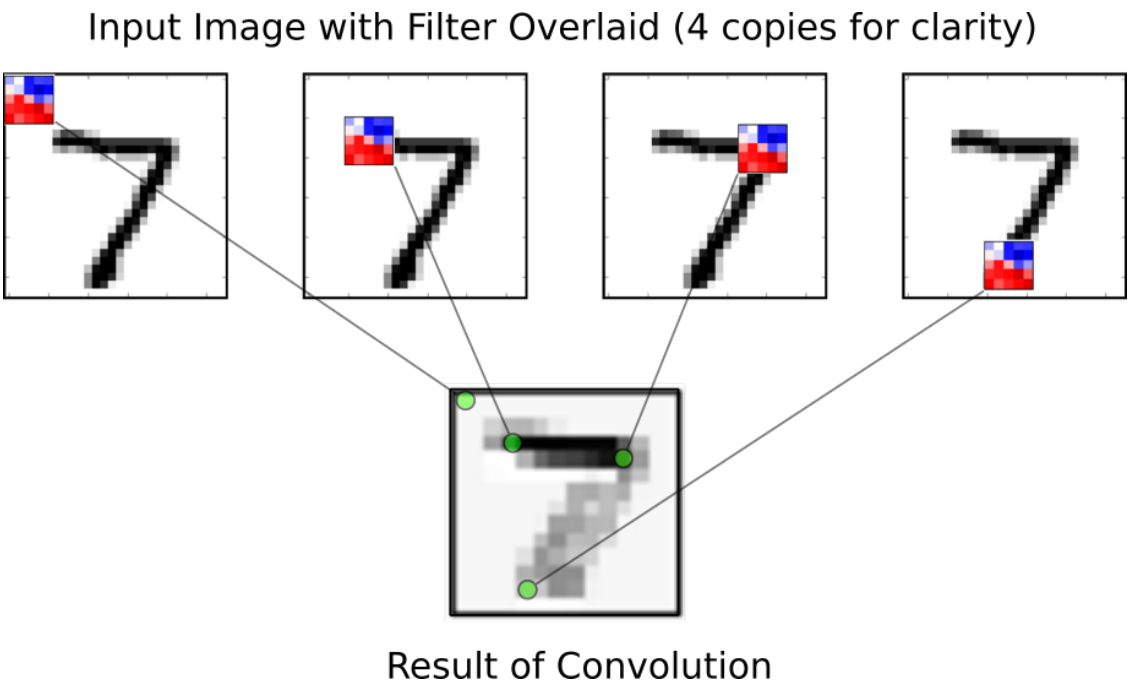
The following chart shows the basic idea of processing an image in the first convolutional layer. The input image depicts the number 7 and four copies of the image are shown here, so we can see more clearly how the filter is being moved to different positions of the image. For each position of the filter, the dot-product is being calculated between the filter and the image pixels under the filter, which results in a single pixel in the output image. So moving the filter across the entire input image results in a new image being generated.

The red filter-weights means that the filter has a positive reaction to black pixels in the input image, while blue pixels means the filter has a negative reaction to black pixels.

In this case it appears that the filter recognizes the horizontal line of the 7-digit, as can be seen from its stronger reaction to that line in the output image.

```
In [2]: Image('images/02_convolution.png')
```

Out[2]:



The step-size for moving the filter across the input is called the stride. There is a stride for moving the filter horizontally (x-axis) and another stride for moving vertically (y-axis).

In the source-code below, the stride is set to 1 in both directions, which means the filter starts in the upper left corner of the input image and is being moved 1 pixel to the right in each step. When the filter reaches the end of the image to the right, then the filter is moved back to the left side and 1 pixel down the image. This continues until the filter has reached the lower right corner of the input image and the entire output image has been generated.

When the filter reaches the end of the right-side as well as the bottom of the input image, then it can be padded with zeroes (white pixels). This causes the output image to be of the exact same dimension as the input image.

Furthermore, the output of the convolution may be passed through a so-called Rectified Linear Unit (ReLU), which merely ensures that the output is positive because negative values are set to zero. The output may also be down-sampled by so-called max-pooling, which considers small windows of 2x2 pixels and only keeps the largest of those pixels. This halves the resolution of the input image e.g. from 28x28 to 14x14 pixels.

Note that the second convolutional layer is more complicated because it takes 16 input channels. We want a separate filter for each input channel, so we need 16 filters instead of just one. Furthermore, we want 36 output channels from the second convolutional layer, so in total we need 16 x 36 = 576 filters for the second convolutional layer. It can be a bit challenging to understand how this works.

2.4 Imports

```
In [3]: %matplotlib inline
        import matplotlib.pyplot as plt
        import tensorflow as tf
```

```
import numpy as np
from sklearn.metrics import confusion_matrix
import time
from datetime import timedelta
import math
```

This was developed using Python 3.5.2 (Anaconda) and TensorFlow version:

```
In [4]: tf.__version__
```

```
Out[4]: '1.1.0'
```

2.5 Configuration of Neural Network

The configuration of the Convolutional Neural Network is defined here for convenience, so you can easily find and change these numbers and re-run the Notebook.

```
In [5]: # Convolutional Layer 1.
        filter_size1 = 5          # Convolution filters are 5 x 5 pixels.
        num_filters1 = 16         # There are 16 of these filters.

        # Convolutional Layer 2.
        filter_size2 = 5          # Convolution filters are 5 x 5 pixels.
        num_filters2 = 36         # There are 36 of these filters.

        # Fully-connected layer.
        fc_size = 128             # Number of neurons in fully-connected layer.
```

2.6 Load Data

The MNIST data-set is about 12 MB and will be downloaded automatically if it is not located in the given path.

```
In [6]: from tensorflow.examples.tutorials.mnist import input_data
        data = input_data.read_data_sets('data/MNIST/', one_hot=True)
```

```
Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
```

The MNIST data-set has now been loaded and consists of 70,000 images and associated labels (i.e. classifications of the images). The data-set is split into 3 mutually exclusive sub-sets. We will only use the training and test-sets in this tutorial.

```
In [7]: print("Size of:")
        print("- Training-set:\t\t{}".format(len(data.train.labels)))
        print("- Test-set:\t\t{}".format(len(data.test.labels)))
        print("- Validation-set:\t{}".format(len(data.validation.labels)))
```

```
Size of:
- Training-set:                55000
- Test-set:                    10000
- Validation-set:              5000
```

The class-labels are One-Hot encoded, which means that each label is a vector with 10 elements, all of which are zero except for one element. The index of this one element is the class-number, that is, the digit shown in the associated image. We also need the class-numbers as integers for the test-set, so we calculate it now.

```
In [8]: data.test.cls = np.argmax(data.test.labels, axis=1)
```

2.7 Data Dimensions

The data dimensions are used in several places in the source-code below. They are defined once so we can use these variables instead of numbers throughout the source-code below.

```
In [9]: # We know that MNIST images are 28 pixels in each dimension.
        img_size = 28

        # Images are stored in one-dimensional arrays of this length.
        img_size_flat = img_size * img_size

        # Tuple with height and width of images used to reshape arrays.
        img_shape = (img_size, img_size)

        # Number of colour channels for the images: 1 channel for gray-scale.
        num_channels = 1

        # Number of classes, one class for each of 10 digits.
        num_classes = 10
```

2.7.1 Helper-function for plotting images

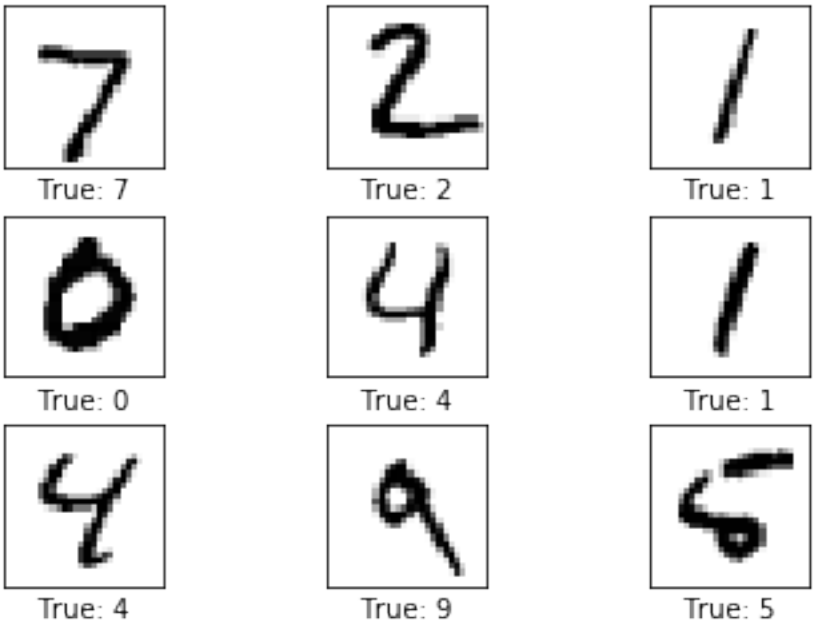
Function used to plot 9 images in a 3x3 grid, and writing the true and predicted classes below each image.

```
In [10]: def plot_images(images, cls_true, cls_pred=None):
             assert len(images) == len(cls_true) == 9

             # Create figure with 3x3 sub-plots.
             fig, axes = plt.subplots(3, 3)
             fig.subplots_adjust(hspace=0.3, wspace=0.3)

             for i, ax in enumerate(axes.flat):
                 # Plot image.
                 ax.imshow(images[i].reshape(img_shape), cmap='binary')
```

```
            # Show true and predicted classes.
            if cls_pred is None:
                xlabel = "True: {0}".format(cls_true[i])
            else:
                xlabel = "True: {0}, Pred: {1}".format(cls_true[i], cls_pred[i])

            # Show the classes as the label on the x-axis.
            ax.set_xlabel(xlabel)

            # Remove ticks from the plot.
            ax.set_xticks([])
            ax.set_yticks([])

        # Ensure the plot is shown correctly with multiple plots
        # in a single Notebook cell.
        plt.show()
```

2.7.2 Plot a few images to see if data is correct

```
In [11]: # Get the first images from the test-set.
         images = data.test.images[0:9]

         # Get the true classes for those images.
         cls_true = data.test.cls[0:9]

         # Plot the images and labels using our helper-function above.
         plot_images(images=images, cls_true=cls_true)
```



2.8 TensorFlow Graph

The entire purpose of TensorFlow is to have a so-called computational graph that can be executed much more efficiently than if the same calculations were to be performed directly in Python. TensorFlow can be more efficient than NumPy because TensorFlow knows the entire computation graph that must be executed, while NumPy only knows the computation of a single mathematical operation at a time.

TensorFlow can also automatically calculate the gradients that are needed to optimize the variables of the graph so as to make the model perform better. This is because the graph is a combination of simple mathematical expressions so the gradient of the entire graph can be calculated using the chain-rule for derivatives.

TensorFlow can also take advantage of multi-core CPUs as well as GPUs - and Google has even built special chips just for TensorFlow which are called TPUs (Tensor Processing Units) and are even faster than GPUs.

A TensorFlow graph consists of the following parts which will be detailed below:

- Placeholder variables used for inputting data to the graph.
- Variables that are going to be optimized so as to make the convolutional network perform better.
- The mathematical formulas for the convolutional network.
- A cost measure that can be used to guide the optimization of the variables.
- An optimization method which updates the variables.

In addition, the TensorFlow graph may also contain various debugging statements e.g. for logging data to be displayed using TensorBoard, which is not covered in this tutorial.

2.8.1 Helper-functions for creating new variables

Functions for creating new TensorFlow variables in the given shape and initializing them with random values. Note that the initialization is not actually done at this point, it is merely being defined in the TensorFlow graph.

```
In [12]: def new_weights(shape):
             return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
```

```
In [13]: def new_biases(length):
             return tf.Variable(tf.constant(0.05, shape=[length]))
```

2.8.2 Helper-function for creating a new Convolutional Layer

This function creates a new convolutional layer in the computational graph for TensorFlow. Nothing is actually calculated here, we are just adding the mathematical formulas to the TensorFlow graph.

It is assumed that the input is a 4-dim tensor with the following dimensions:

1. Image number.
2. Y-axis of each image.

3. X-axis of each image.
4. Channels of each image.

Note that the input channels may either be colour-channels, or it may be filter-channels if the input is produced from a previous convolutional layer.

The output is another 4-dim tensor with the following dimensions:

1. Image number, same as input.
2. Y-axis of each image. If 2x2 pooling is used, then the height and width of the input images is divided by 2.
3. X-axis of each image. Ditto.
4. Channels produced by the convolutional filters.

```
In [14]: def new_conv_layer(input,              # The previous layer.
                            num_input_channels, # Num. channels in prev. layer.
                            filter_size,        # Width and height of each filter.
                            num_filters,        # Number of filters.
                            use_pooling=True):  # Use 2x2 max-pooling.

             # Shape of the filter-weights for the convolution.
             # This format is determined by the TensorFlow API.
             shape = [filter_size, filter_size, num_input_channels, num_filters]

             # Create new weights aka. filters with the given shape.
             weights = new_weights(shape=shape)

             # Create new biases, one for each filter.
             biases = new_biases(length=num_filters)

             # Create the TensorFlow operation for convolution.
             # Note the strides are set to 1 in all dimensions.
             # The first and last stride must always be 1,
             # because the first is for the image-number and
             # the last is for the input-channel.
             # But e.g. strides=[1, 2, 2, 1] would mean that the filter
             # is moved 2 pixels across the x- and y-axis of the image.
             # The padding is set to 'SAME' which means the input image
             # is padded with zeroes so the size of the output is the same.
             layer = tf.nn.conv2d(input=input,
                                  filter=weights,
                                  strides=[1, 1, 1, 1],
                                  padding='SAME')

             # Add the biases to the results of the convolution.
             # A bias-value is added to each filter-channel.
             layer += biases

             # Use pooling to down-sample the image resolution?
```

```
    if use_pooling:
        # This is 2x2 max-pooling, which means that we
        # consider 2x2 windows and select the largest value
        # in each window. Then we move 2 pixels to the next window.
        layer = tf.nn.max_pool(value=layer,
                               ksize=[1, 2, 2, 1],
                               strides=[1, 2, 2, 1],
                               padding='SAME')

    # Rectified Linear Unit (ReLU).
    # It calculates max(x, 0) for each input pixel x.
    # This adds some non-linearity to the formula and allows us
    # to learn more complicated functions.
    layer = tf.nn.relu(layer)

    # Note that ReLU is normally executed before the pooling,
    # but since relu(max_pool(x)) == max_pool(relu(x)) we can
    # save 75% of the relu-operations by max-pooling first.

    # We return both the resulting layer and the filter-weights
    # because we will plot the weights later.
    return layer, weights
```

2.8.3 Helper-function for flattening a layer

A convolutional layer produces an output tensor with 4 dimensions. We will add fully-connected layers after the convolution layers, so we need to reduce the 4-dim tensor to 2-dim which can be used as input to the fully-connected layer.

```
In [15]: def flatten_layer(layer):
    # Get the shape of the input layer.
    layer_shape = layer.get_shape()

    # The shape of the input layer is assumed to be:
    # layer_shape == [num_images, img_height, img_width, num_channels]

    # The number of features is: img_height * img_width * num_channels
    # We can use a function from TensorFlow to calculate this.
    num_features = layer_shape[1:4].num_elements()

    # Reshape the layer to [num_images, num_features].
    # Note that we just set the size of the second dimension
    # to num_features and the size of the first dimension to -1
    # which means the size in that dimension is calculated
    # so the total size of the tensor is unchanged from the reshaping.
    layer_flat = tf.reshape(layer, [-1, num_features])

    # The shape of the flattened layer is now:
```

```
    # [num_images, img_height * img_width * num_channels]

    # Return both the flattened layer and the number of features.
    return layer_flat, num_features
```

2.8.4 Helper-function for creating a new Fully-Connected Layer

This function creates a new fully-connected layer in the computational graph for TensorFlow. Nothing is actually calculated here, we are just adding the mathematical formulas to the TensorFlow graph.

It is assumed that the input is a 2-dim tensor of shape `[num_images, num_inputs]`. The output is a 2-dim tensor of shape `[num_images, num_outputs]`.

```
In [16]: def new_fc_layer(input,          # The previous layer.
                          num_inputs,     # Num. inputs from prev. layer.
                          num_outputs,    # Num. outputs.
                          use_relu=True): # Use Rectified Linear Unit (ReLU)?

    # Create new weights and biases.
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)

    # Calculate the layer as the matrix multiplication of
    # the input and weights, and then add the bias-values.
    layer = tf.matmul(input, weights) + biases

    # Use ReLU?
    if use_relu:
        layer = tf.nn.relu(layer)

    return layer
```

2.8.5 Placeholder variables

Placeholder variables serve as the input to the TensorFlow computational graph that we may change each time we execute the graph. We call this feeding the placeholder variables and it is demonstrated further below.

First we define the placeholder variable for the input images. This allows us to change the images that are input to the TensorFlow graph. This is a so-called tensor, which just means that it is a multi-dimensional vector or matrix. The data-type is set to `float32` and the shape is set to `[None, img_size_flat]`, where `None` means that the tensor may hold an arbitrary number of images with each image being a vector of length `img_size_flat`.

```
In [17]: x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='x')
```

The convolutional layers expect `x` to be encoded as a 4-dim tensor so we have to reshape it so its shape is instead `[num_images, img_height, img_width, num_channels]`. Note that `img_height == img_width == img_size` and `num_images` can be inferred automatically by using -1 for the size of the first dimension. So the reshape operation is:

```
In [18]: x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
```

Next we have the placeholder variable for the true labels associated with the images that were input in the placeholder variable `x`. The shape of this placeholder variable is `[None, num_classes]` which means it may hold an arbitrary number of labels and each label is a vector of length `num_classes` which is 10 in this case.

```
In [19]: y_true = tf.placeholder(tf.float32, shape=[None, num_classes], name='y_true')
```

We could also have a placeholder variable for the class-number, but we will instead calculate it using argmax. Note that this is a TensorFlow operator so nothing is calculated at this point.

```
In [20]: y_true_cls = tf.argmax(y_true, dimension=1)
```

2.8.6 Convolutional Layer 1

Create the first convolutional layer. It takes `x_image` as input and creates `num_filters1` different filters, each having width and height equal to `filter_size1`. Finally we wish to down-sample the image so it is half the size by using 2x2 max-pooling.

```
In [21]: layer_conv1, weights_conv1 = \
             new_conv_layer(input=x_image,
                            num_input_channels=num_channels,
                            filter_size=filter_size1,
                            num_filters=num_filters1,
                            use_pooling=True)
```

Check the shape of the tensor that will be output by the convolutional layer. It is (?, 14, 14, 16) which means that there is an arbitrary number of images (this is the ?), each image is 14 pixels wide and 14 pixels high, and there are 16 different channels, one channel for each of the filters.

```
In [22]: layer_conv1
```

```
Out[22]: <tf.Tensor 'Relu:0' shape=(?, 14, 14, 16) dtype=float32>
```

2.8.7 Convolutional Layer 2

Create the second convolutional layer, which takes as input the output from the first convolutional layer. The number of input channels corresponds to the number of filters in the first convolutional layer.

```
In [23]: layer_conv2, weights_conv2 = \
             new_conv_layer(input=layer_conv1,
                            num_input_channels=num_filters1,
                            filter_size=filter_size2,
                            num_filters=num_filters2,
                            use_pooling=True)
```

Check the shape of the tensor that will be output from this convolutional layer. The shape is (?, 7, 7, 36) where the ? again means that there is an arbitrary number of images, with each image having width and height of 7 pixels, and there are 36 channels, one for each filter.

```
In [24]: layer_conv2
```

```
Out[24]: <tf.Tensor 'Relu_1:0' shape=(?, 7, 7, 36) dtype=float32>
```

2.8.8 Flatten Layer

The convolutional layers output 4-dim tensors. We now wish to use these as input in a fully-connected network, which requires for the tensors to be reshaped or flattened to 2-dim tensors.

```
In [25]: layer_flat, num_features = flatten_layer(layer_conv2)
```

Check that the tensors now have shape (?, 1764) which means there's an arbitrary number of images which have been flattened to vectors of length 1764 each. Note that 1764 = 7 x 7 x 36.

```
In [26]: layer_flat
```

```
Out[26]: <tf.Tensor 'Reshape_1:0' shape=(?, 1764) dtype=float32>
```

```
In [27]: num_features
```

```
Out[27]: 1764
```

2.8.9 Fully-Connected Layer 1

Add a fully-connected layer to the network. The input is the flattened layer from the previous convolution. The number of neurons or nodes in the fully-connected layer is `fc_size`. ReLU is used so we can learn non-linear relations.

```
In [28]: layer_fc1 = new_fc_layer(input=layer_flat,
                                  num_inputs=num_features,
                                  num_outputs=fc_size,
                                  use_relu=True)
```

Check that the output of the fully-connected layer is a tensor with shape (?, 128) where the ? means there is an arbitrary number of images and `fc_size` == 128.

```
In [29]: layer_fc1
```

```
Out[29]: <tf.Tensor 'Relu_2:0' shape=(?, 128) dtype=float32>
```

2.8.10 Fully-Connected Layer 2

Add another fully-connected layer that outputs vectors of length 10 for determining which of the 10 classes the input image belongs to. Note that ReLU is not used in this layer.

```
In [30]: layer_fc2 = new_fc_layer(input=layer_fc1,
                                  num_inputs=fc_size,
                                  num_outputs=num_classes,
                                  use_relu=False)
```

```
In [31]: layer_fc2
```

```
Out[31]: <tf.Tensor 'add_3:0' shape=(?, 10) dtype=float32>
```

2.8.11 Predicted Class

The second fully-connected layer estimates how likely it is that the input image belongs to each of the 10 classes. However, these estimates are a bit rough and difficult to interpret because the numbers may be very small or large, so we want to normalize them so that each element is limited between zero and one and the 10 elements sum to one. This is calculated using the so-called softmax function and the result is stored in `y_pred`.

```
In [32]: y_pred = tf.nn.softmax(layer_fc2)
```

The class-number is the index of the largest element.

```
In [33]: y_pred_cls = tf.argmax(y_pred, dimension=1)
```

2.8.12 Cost-function to be optimized

To make the model better at classifying the input images, we must somehow change the variables for all the network layers. To do this we first need to know how well the model currently performs by comparing the predicted output of the model `y_pred` to the desired output `y_true`.

The cross-entropy is a performance measure used in classification. The cross-entropy is a continuous function that is always positive and if the predicted output of the model exactly matches the desired output then the cross-entropy equals zero. The goal of optimization is therefore to minimize the cross-entropy so it gets as close to zero as possible by changing the variables of the network layers.

TensorFlow has a built-in function for calculating the cross-entropy. Note that the function calculates the softmax internally so we must use the output of `layer_fc2` directly rather than `y_pred` which has already had the softmax applied.

```
In [34]: cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2,
                                                                 labels=y_true)
```

We have now calculated the cross-entropy for each of the image classifications so we have a measure of how well the model performs on each image individually. But in order to use the cross-entropy to guide the optimization of the model's variables we need a single scalar value, so we simply take the average of the cross-entropy for all the image classifications.

```
In [35]: cost = tf.reduce_mean(cross_entropy)
```

2.8.13 Optimization Method

Now that we have a cost measure that must be minimized, we can then create an optimizer. In this case it is the `AdamOptimizer` which is an advanced form of Gradient Descent.

Note that optimization is not performed at this point. In fact, nothing is calculated at all, we just add the optimizer-object to the TensorFlow graph for later execution.

```
In [36]: optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost)
```

2.8.14 Performance Measures

We need a few more performance measures to display the progress to the user.

This is a vector of booleans whether the predicted class equals the true class of each image.

```
In [37]: correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

This calculates the classification accuracy by first type-casting the vector of booleans to floats, so that False becomes 0 and True becomes 1, and then calculating the average of these numbers.

```
In [38]: accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

2.9 TensorFlow Run

2.9.1 Create TensorFlow session

Once the TensorFlow graph has been created, we have to create a TensorFlow session which is used to execute the graph.

```
In [39]: session = tf.Session()
```

2.9.2 Initialize variables

The variables for `weights` and `biases` must be initialized before we start optimizing them.

```
In [40]: session.run(tf.global_variables_initializer())
```

2.9.3 Helper-function to perform optimization iterations

There are 55,000 images in the training-set. It takes a long time to calculate the gradient of the model using all these images. We therefore only use a small batch of images in each iteration of the optimizer.

If your computer crashes or becomes very slow because you run out of RAM, then you may try and lower this number, but you may then need to perform more optimization iterations.

```
In [41]: train_batch_size = 64
```

Function for performing a number of optimization iterations so as to gradually improve the variables of the network layers. In each iteration, a new batch of data is selected from the training-set and then TensorFlow executes the optimizer using those training samples. The progress is printed every 100 iterations.

```
In [42]: # Counter for total number of iterations performed so far.
         total_iterations = 0

         def optimize(num_iterations):
             # Ensure we update the global variable rather than a local copy.
             global total_iterations

             # Start-time used for printing time-usage below.
             start_time = time.time()
```

```
    for i in range(total_iterations,
                   total_iterations + num_iterations):

        # Get a batch of training examples.
        # x_batch now holds a batch of images and
        # y_true_batch are the true labels for those images.
        x_batch, y_true_batch = data.train.next_batch(train_batch_size)

        # Put the batch into a dict with the proper names
        # for placeholder variables in the TensorFlow graph.
        feed_dict_train = {x: x_batch,
                           y_true: y_true_batch}

        # Run the optimizer using this batch of training data.
        # TensorFlow assigns the variables in feed_dict_train
        # to the placeholder variables and then runs the optimizer.
        session.run(optimizer, feed_dict=feed_dict_train)

        # Print status every 100 iterations.
        if i % 100 == 0:
            # Calculate the accuracy on the training-set.
            acc = session.run(accuracy, feed_dict=feed_dict_train)

            # Message for printing.
            msg = "Optimization Iteration: {0:>6}, Training Accuracy: {1:>6.1%}"

            # Print it.
            print(msg.format(i + 1, acc))

    # Update the total number of iterations performed.
    total_iterations += num_iterations

    # Ending time.
    end_time = time.time()

    # Difference between start and end-times.
    time_dif = end_time - start_time

    # Print the time-usage.
    print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))
```

2.9.4 Helper-function to plot example errors

Function for plotting examples of images from the test-set that have been mis-classified.

```
In [43]: def plot_example_errors(cls_pred, correct):
             # This function is called from print_test_accuracy() below.
```

```
    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # correct is a boolean array whether the predicted class
    # is equal to the true class for each image in the test-set.

    # Negate the boolean array.
    incorrect = (correct == False)

    # Get the images from the test-set that have been
    # incorrectly classified.
    images = data.test.images[incorrect]

    # Get the predicted classes for those images.
    cls_pred = cls_pred[incorrect]

    # Get the true classes for those images.
    cls_true = data.test.cls[incorrect]

    # Plot the first 9 images.
    plot_images(images=images[0:9],
                cls_true=cls_true[0:9],
                cls_pred=cls_pred[0:9])
```

2.9.5 Helper-function to plot confusion matrix

```
In [44]: def plot_confusion_matrix(cls_pred):
    # This is called from print_test_accuracy() below.

    # cls_pred is an array of the predicted class-number for
    # all images in the test-set.

    # Get the true classifications for the test-set.
    cls_true = data.test.cls

    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_true,
                          y_pred=cls_pred)

    # Print the confusion matrix as text.
    print(cm)

    # Plot the confusion matrix as an image.
    plt.matshow(cm)

    # Make various adjustments to the plot.
    plt.colorbar()
```

```
    tick_marks = np.arange(num_classes)
    plt.xticks(tick_marks, range(num_classes))
    plt.yticks(tick_marks, range(num_classes))
    plt.xlabel('Predicted')
    plt.ylabel('True')

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

2.9.6 Helper-function for showing the performance

Function for printing the classification accuracy on the test-set.

It takes a while to compute the classification for all the images in the test-set, that's why the results are re-used by calling the above functions directly from this function, so the classifications don't have to be recalculated by each function.

Note that this function can use a lot of computer memory, which is why the test-set is split into smaller batches. If you have little RAM in your computer and it crashes, then you can try and lower the batch-size.

```
In [45]: # Split the test-set into smaller batches of this size.
         test_batch_size = 256

         def print_test_accuracy(show_example_errors=False,
                                 show_confusion_matrix=False):

             # Number of images in the test-set.
             num_test = len(data.test.images)

             # Allocate an array for the predicted classes which
             # will be calculated in batches and filled into this array.
             cls_pred = np.zeros(shape=num_test, dtype=np.int)

             # Now calculate the predicted classes for the batches.
             # We will just iterate through all the batches.
             # There might be a more clever and Pythonic way of doing this.

             # The starting index for the next batch is denoted i.
             i = 0

             while i < num_test:
                 # The ending index for the next batch is denoted j.
                 j = min(i + test_batch_size, num_test)

                 # Get the images from the test-set between index i and j.
                 images = data.test.images[i:j, :]

                 # Get the associated labels.
```

```
        labels = data.test.labels[i:j, :]

        # Create a feed-dict with these images and labels.
        feed_dict = {x: images,
                     y_true: labels}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Convenience variable for the true class-numbers of the test-set.
    cls_true = data.test.cls

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means 1.
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classified
    # images divided by the total number of images in the test-set.
    acc = float(correct_sum) / num_test

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

2.10 Performance before any optimization

The accuracy on the test-set is very low because the model variables have only been initialized and not optimized at all, so it just classifies the images randomly.

```
In [46]: print_test_accuracy()
```

```
Accuracy on Test-Set: 10.1% (1009 / 10000)
```

2.11 Performance after 1 optimization iteration

The classification accuracy does not improve much from just 1 optimization iteration, because the learning-rate for the optimizer is set very low.

```
In [47]: optimize(num_iterations=1)
```

```
Optimization Iteration:      1, Training Accuracy:   7.8%
Time usage: 0:00:00
```

```
In [48]: print_test_accuracy()
```

```
Accuracy on Test-Set: 8.3% (834 / 10000)
```

2.12 Performance after 100 optimization iterations

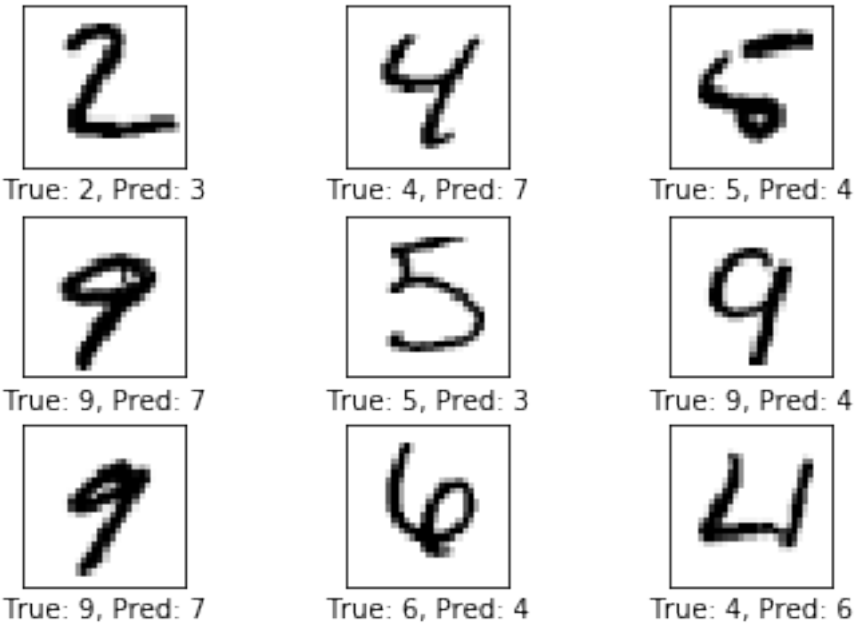
After 100 optimization iterations, the model has significantly improved its classification accuracy.

```
In [49]: optimize(num_iterations=99) # We already performed 1 iteration above.
```

```
Time usage: 0:00:00
```

```
In [50]: print_test_accuracy(show_example_errors=True)
```

```
Accuracy on Test-Set: 70.2% (7019 / 10000)
Example errors:
```



2.13 Performance after 1000 optimization iterations

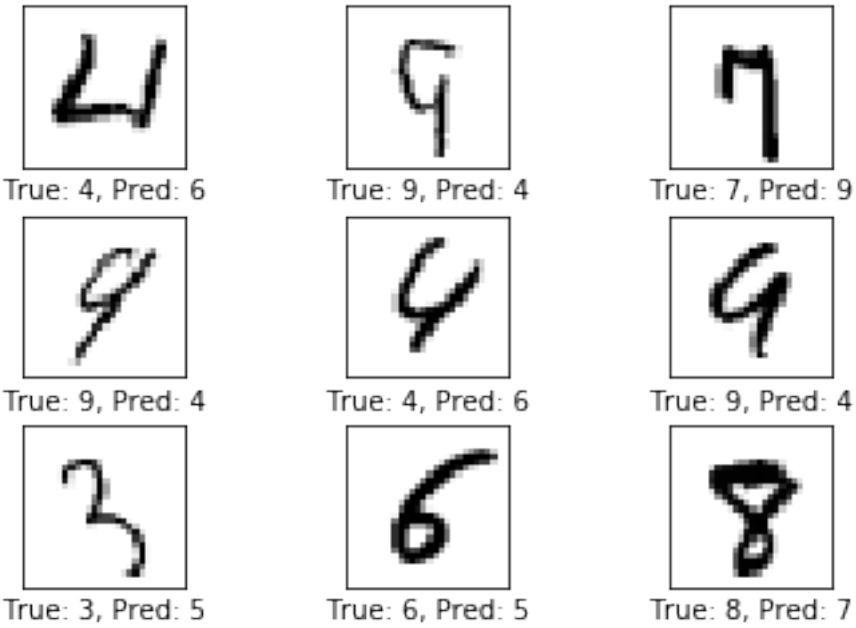
After 1000 optimization iterations, the model has greatly increased its accuracy on the test-set to more than 90%.

```
In [51]: optimize(num_iterations=900) # We performed 100 iterations above.
```

```
Optimization Iteration:    101, Training Accuracy:  76.6%
Optimization Iteration:    201, Training Accuracy:  81.2%
Optimization Iteration:    301, Training Accuracy:  82.8%
Optimization Iteration:    401, Training Accuracy:  87.5%
Optimization Iteration:    501, Training Accuracy:  89.1%
Optimization Iteration:    601, Training Accuracy:  84.4%
Optimization Iteration:    701, Training Accuracy:  90.6%
Optimization Iteration:    801, Training Accuracy:  98.4%
Optimization Iteration:    901, Training Accuracy:  89.1%
Time usage: 0:00:03
```

```
In [52]: print_test_accuracy(show_example_errors=True)
```

```
Accuracy on Test-Set: 93.0% (9303 / 10000)
Example errors:
```



2.14 Performance after 10,000 optimization iterations

After 10,000 optimization iterations, the model has a classification accuracy on the test-set of about 99%.

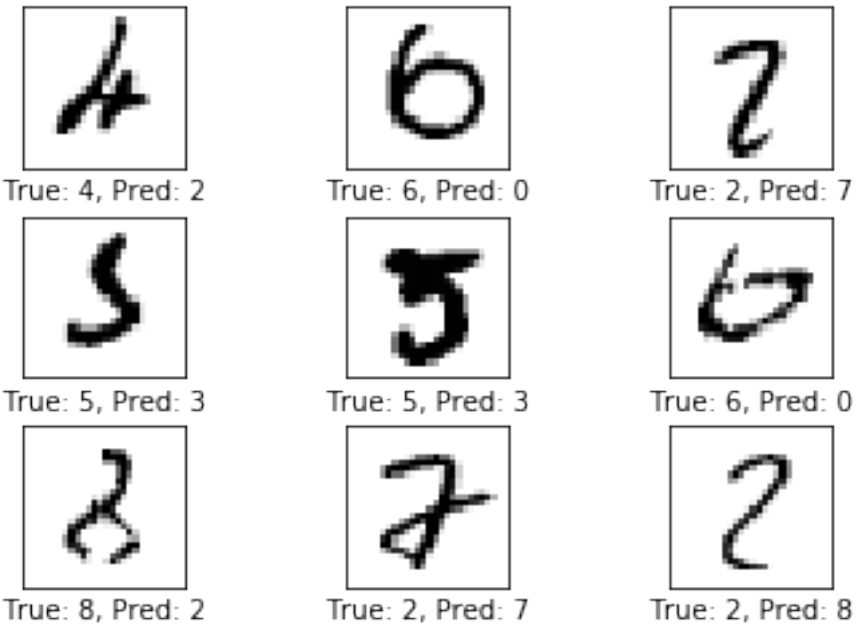
```
In [53]: optimize(num_iterations=9000) # We performed 1000 iterations above.
```

```
Optimization Iteration:   1001, Training Accuracy:  93.8%
Optimization Iteration:   1101, Training Accuracy:  92.2%
Optimization Iteration:   1201, Training Accuracy:  93.8%
Optimization Iteration:   1301, Training Accuracy:  92.2%
Optimization Iteration:   1401, Training Accuracy:  96.9%
Optimization Iteration:   1501, Training Accuracy:  93.8%
Optimization Iteration:   1601, Training Accuracy:  96.9%
Optimization Iteration:   1701, Training Accuracy: 100.0%
Optimization Iteration:   1801, Training Accuracy:  93.8%
Optimization Iteration:   1901, Training Accuracy:  96.9%
Optimization Iteration:   2001, Training Accuracy:  95.3%
Optimization Iteration:   2101, Training Accuracy: 100.0%
Optimization Iteration:   2201, Training Accuracy:  98.4%
Optimization Iteration:   2301, Training Accuracy:  98.4%
Optimization Iteration:   2401, Training Accuracy:  92.2%
Optimization Iteration:   2501, Training Accuracy: 100.0%
Optimization Iteration:   2601, Training Accuracy:  95.3%
Optimization Iteration:   2701, Training Accuracy: 100.0%
Optimization Iteration:   2801, Training Accuracy:  95.3%
Optimization Iteration:   2901, Training Accuracy:  98.4%
Optimization Iteration:   3001, Training Accuracy: 100.0%
Optimization Iteration:   3101, Training Accuracy: 100.0%
Optimization Iteration:   3201, Training Accuracy:  98.4%
Optimization Iteration:   3301, Training Accuracy:  98.4%
Optimization Iteration:   3401, Training Accuracy:  95.3%
Optimization Iteration:   3501, Training Accuracy:  98.4%
Optimization Iteration:   3601, Training Accuracy:  96.9%
Optimization Iteration:   3701, Training Accuracy: 100.0%
Optimization Iteration:   3801, Training Accuracy: 100.0%
Optimization Iteration:   3901, Training Accuracy: 100.0%
Optimization Iteration:   4001, Training Accuracy: 100.0%
Optimization Iteration:   4101, Training Accuracy:  98.4%
Optimization Iteration:   4201, Training Accuracy:  98.4%
Optimization Iteration:   4301, Training Accuracy:  98.4%
Optimization Iteration:   4401, Training Accuracy:  96.9%
Optimization Iteration:   4501, Training Accuracy:  98.4%
Optimization Iteration:   4601, Training Accuracy:  95.3%
Optimization Iteration:   4701, Training Accuracy:  96.9%
Optimization Iteration:   4801, Training Accuracy: 100.0%
Optimization Iteration:   4901, Training Accuracy:  98.4%
Optimization Iteration:   5001, Training Accuracy:  96.9%
Optimization Iteration:   5101, Training Accuracy:  96.9%
```

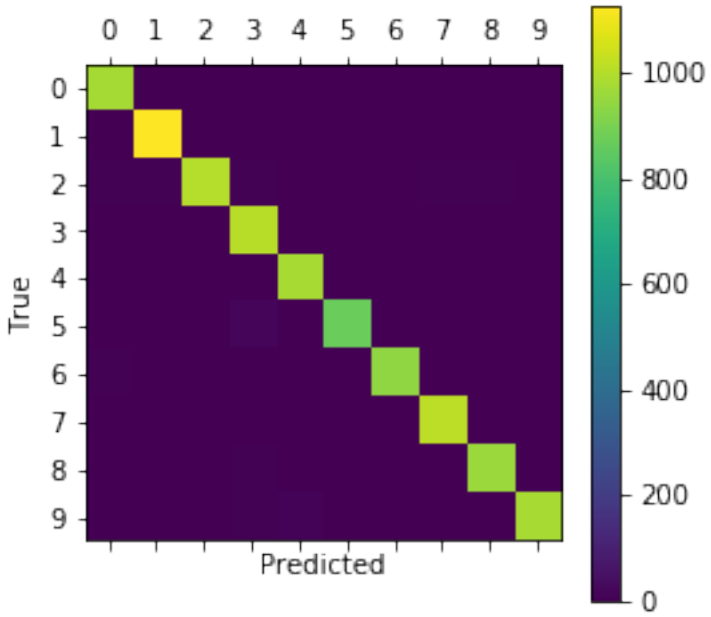
```
Optimization Iteration:   5201, Training Accuracy:  98.4%
Optimization Iteration:   5301, Training Accuracy:  96.9%
Optimization Iteration:   5401, Training Accuracy:  95.3%
Optimization Iteration:   5501, Training Accuracy: 100.0%
Optimization Iteration:   5601, Training Accuracy:  96.9%
Optimization Iteration:   5701, Training Accuracy: 100.0%
Optimization Iteration:   5801, Training Accuracy:  95.3%
Optimization Iteration:   5901, Training Accuracy: 100.0%
Optimization Iteration:   6001, Training Accuracy:  98.4%
Optimization Iteration:   6101, Training Accuracy:  96.9%
Optimization Iteration:   6201, Training Accuracy:  98.4%
Optimization Iteration:   6301, Training Accuracy:  96.9%
Optimization Iteration:   6401, Training Accuracy:  98.4%
Optimization Iteration:   6501, Training Accuracy:  98.4%
Optimization Iteration:   6601, Training Accuracy: 100.0%
Optimization Iteration:   6701, Training Accuracy: 100.0%
Optimization Iteration:   6801, Training Accuracy: 100.0%
Optimization Iteration:   6901, Training Accuracy: 100.0%
Optimization Iteration:   7001, Training Accuracy: 100.0%
Optimization Iteration:   7101, Training Accuracy:  98.4%
Optimization Iteration:   7201, Training Accuracy: 100.0%
Optimization Iteration:   7301, Training Accuracy:  96.9%
Optimization Iteration:   7401, Training Accuracy:  98.4%
Optimization Iteration:   7501, Training Accuracy:  98.4%
Optimization Iteration:   7601, Training Accuracy:  95.3%
Optimization Iteration:   7701, Training Accuracy:  96.9%
Optimization Iteration:   7801, Training Accuracy: 100.0%
Optimization Iteration:   7901, Training Accuracy:  96.9%
Optimization Iteration:   8001, Training Accuracy:  98.4%
Optimization Iteration:   8101, Training Accuracy: 100.0%
Optimization Iteration:   8201, Training Accuracy: 100.0%
Optimization Iteration:   8301, Training Accuracy: 100.0%
Optimization Iteration:   8401, Training Accuracy:  98.4%
Optimization Iteration:   8501, Training Accuracy:  96.9%
Optimization Iteration:   8601, Training Accuracy:  98.4%
Optimization Iteration:   8701, Training Accuracy: 100.0%
Optimization Iteration:   8801, Training Accuracy:  98.4%
Optimization Iteration:   8901, Training Accuracy: 100.0%
Optimization Iteration:   9001, Training Accuracy:  98.4%
Optimization Iteration:   9101, Training Accuracy:  95.3%
Optimization Iteration:   9201, Training Accuracy: 100.0%
Optimization Iteration:   9301, Training Accuracy:  98.4%
Optimization Iteration:   9401, Training Accuracy:  98.4%
Optimization Iteration:   9501, Training Accuracy: 100.0%
Optimization Iteration:   9601, Training Accuracy: 100.0%
Optimization Iteration:   9701, Training Accuracy:  96.9%
Optimization Iteration:   9801, Training Accuracy: 100.0%
Optimization Iteration:   9901, Training Accuracy:  96.9%
```

```
Time usage: 0:00:25
```

```
In [54]: print_test_accuracy(show_example_errors=True,
                             show_confusion_matrix=True)

Accuracy on Test-Set: 98.6% (9859 / 10000)
Example errors:
```




```
Confusion Matrix:
[[ 975    0    0    0    0    1    1    1    2    0]
 [   0 1128    1    2    1    0    1    1    1    0]
 [   5    5 1002    7    1    0    0    6    5    1]
 [   1    0    0 1008    0    0    0    0    1    0]
 [   0    0    1    0  981    0    0    0    0    0]
 [   1    0    0   17    0  870    1    1    1    1]
 [   7    2    0    0    4    4  940    0    1    0]
 [   1    2    3    3    0    0    0 1017    1    1]
 [   3    1    1    6    1    1    0    2  957    2]
 [   2    4    0    5    9    2    0    4    2  981]]
```



2.15 Visualization of Weights and Layers

In trying to understand why the convolutional neural network can recognize handwritten digits, we will now visualize the weights of the convolutional filters and the resulting output images.

2.15.1 Helper-function for plotting convolutional weights

```
In [55]: def plot_conv_weights(weights, input_channel=0):
             # Assume weights are TensorFlow ops for 4-dim variables
             # e.g. weights_conv1 or weights_conv2.

             # Retrieve the values of the weight-variables from TensorFlow.
             # A feed-dict is not necessary because nothing is calculated.
             w = session.run(weights)

             # Get the lowest and highest values for the weights.
             # This is used to correct the colour intensity across
             # the images so they can be compared with each other.
             w_min = np.min(w)
             w_max = np.max(w)

             # Number of filters used in the conv. layer.
             num_filters = w.shape[3]

             # Number of grids to plot.
             # Rounded-up, square-root of the number of filters.
```

```
    num_grids = math.ceil(math.sqrt(num_filters))

    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot all the filter-weights.
    for i, ax in enumerate(axes.flat):
        # Only plot the valid filter-weights.
        if i<num_filters:
            # Get the weights for the i'th filter of the input channel.
            # See new_conv_layer() for details on the format
            # of this 4-dim tensor.
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=w_min, vmax=w_max,
                      interpolation='nearest', cmap='seismic')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

2.15.2 Helper-function for plotting the output of a convolutional layer

```
In [56]: def plot_conv_layer(layer, image):
    # Assume layer is a TensorFlow op that outputs a 4-dim tensor
    # which is the output of a convolutional layer,
    # e.g. layer_conv1 or layer_conv2.

    # Create a feed-dict containing just one image.
    # Note that we don't need to feed y_true because it is
    # not used in this calculation.
    feed_dict = {x: [image]}

    # Calculate and retrieve the output values of the layer
    # when inputting that image.
    values = session.run(layer, feed_dict=feed_dict)

    # Number of filters used in the conv. layer.
    num_filters = values.shape[3]

    # Number of grids to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_filters))
```

```
    # Create figure with a grid of sub-plots.
    fig, axes = plt.subplots(num_grids, num_grids)

    # Plot the output images of all the filters.
    for i, ax in enumerate(axes.flat):
        # Only plot the images for valid filters.
        if i<num_filters:
            # Get the output image of using the i'th filter.
            # See new_conv_layer() for details on the format
            # of this 4-dim tensor.
            img = values[0, :, :, i]

            # Plot image.
            ax.imshow(img, interpolation='nearest', cmap='binary')

        # Remove ticks from the plot.
        ax.set_xticks([])
        ax.set_yticks([])

    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

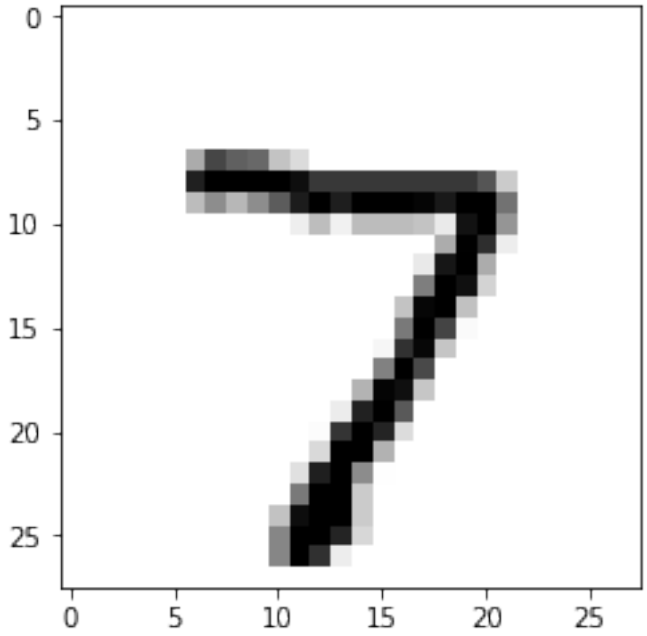
2.15.3 Input Images

Helper-function for plotting an image.

```
In [57]: def plot_image(image):
             plt.imshow(image.reshape(img_shape),
                        interpolation='nearest',
                        cmap='binary')

             plt.show()
```

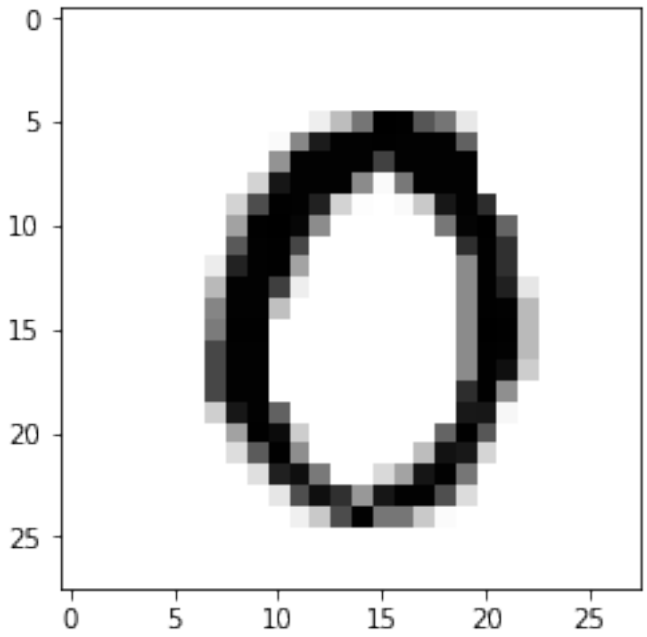
Plot an image from the test-set which will be used as an example below.

```
In [58]: image1 = data.test.images[0]
         plot_image(image1)
```



Plot another example image from the test-set.

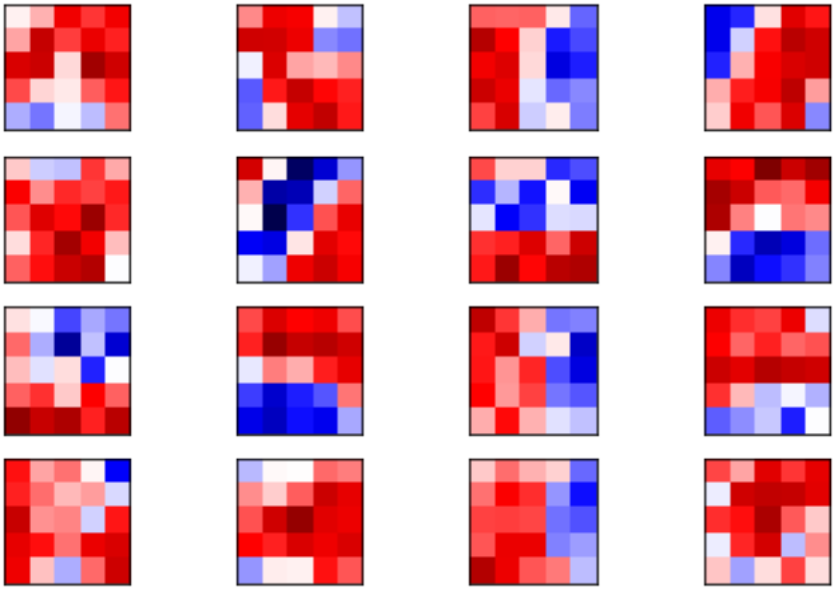
```
In [59]: image2 = data.test.images[13]
         plot_image(image2)
```



2.15.4 Convolution Layer 1

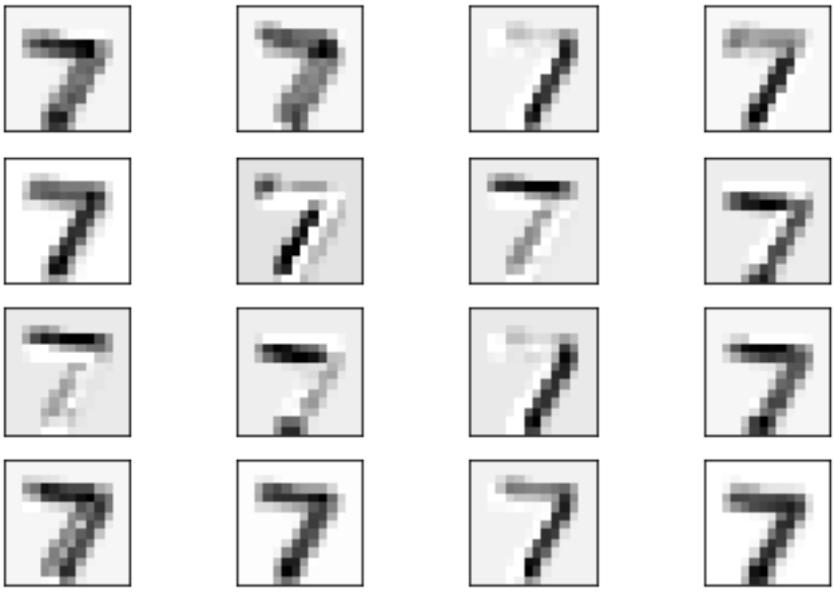
Now plot the filter-weights for the first convolutional layer.
Note that positive weights are red and negative weights are blue.

```
In [60]: plot_conv_weights(weights=weights_conv1)
```



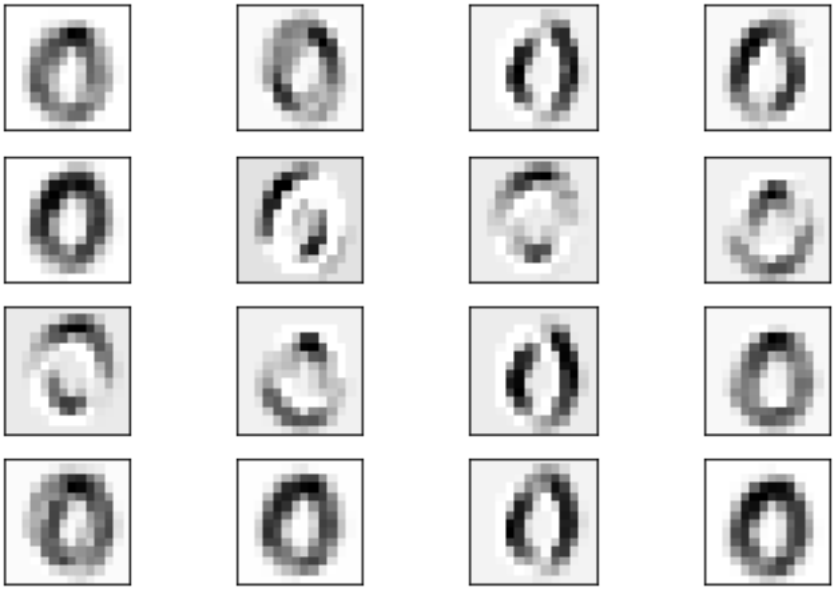
Applying each of these convolutional filters to the first input image gives the following output images, which are then used as input to the second convolutional layer. Note that these images are down-sampled to 14 x 14 pixels which is half the resolution of the original input image.

```
In [61]: plot_conv_layer(layer=layer_conv1, image=image1)
```



The following images are the results of applying the convolutional filters to the second image.

```
In [62]: plot_conv_layer(layer=layer_conv1, image=image2)
```



It is difficult to see from these images what the purpose of the convolutional filters might be. It appears that they have merely created several variations of the input image, as if light was shining from different angles and casting shadows in the image.

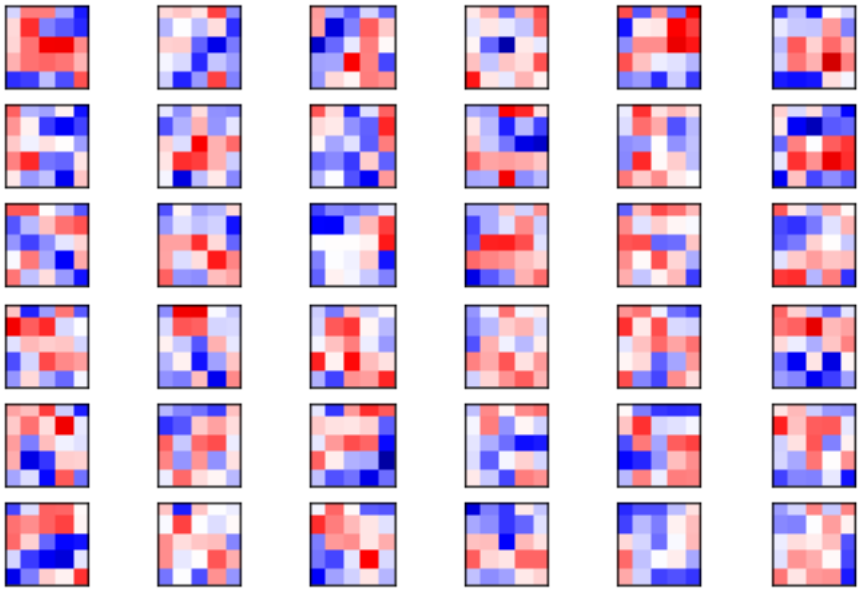
2.15.5 Convolution Layer 2

Now plot the filter-weights for the second convolutional layer.

There are 16 output channels from the first conv-layer, which means there are 16 input channels to the second conv-layer. The second conv-layer has a set of filter-weights for each of its input channels. We start by plotting the filter-weigths for the first channel.

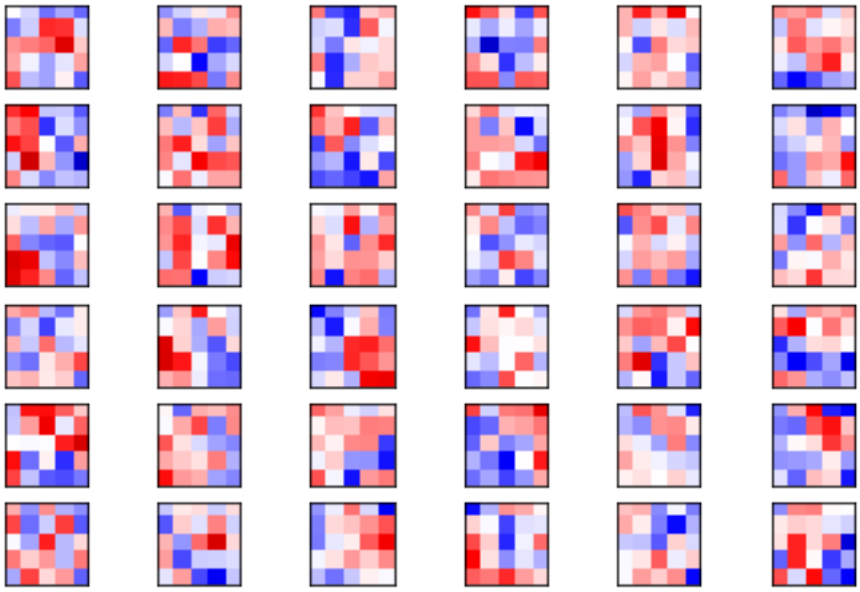
Note again that positive weights are red and negative weights are blue.

```
In [63]: plot_conv_weights(weights=weights_conv2, input_channel=0)
```



There are 16 input channels to the second convolutional layer, so we can make another 15 plots of filter-weights like this. We just make one more with the filter-weights for the second channel.

```
In [64]: plot_conv_weights(weights=weights_conv2, input_channel=1)
```

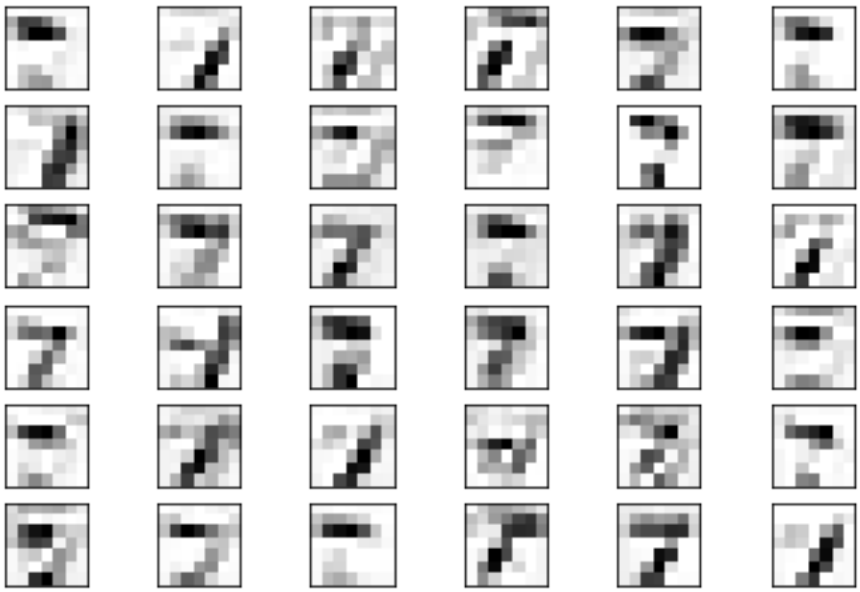


It can be difficult to understand and keep track of how these filters are applied because of the high dimensionality.

Applying these convolutional filters to the images that were ouput from the first conv-layer gives the following images.

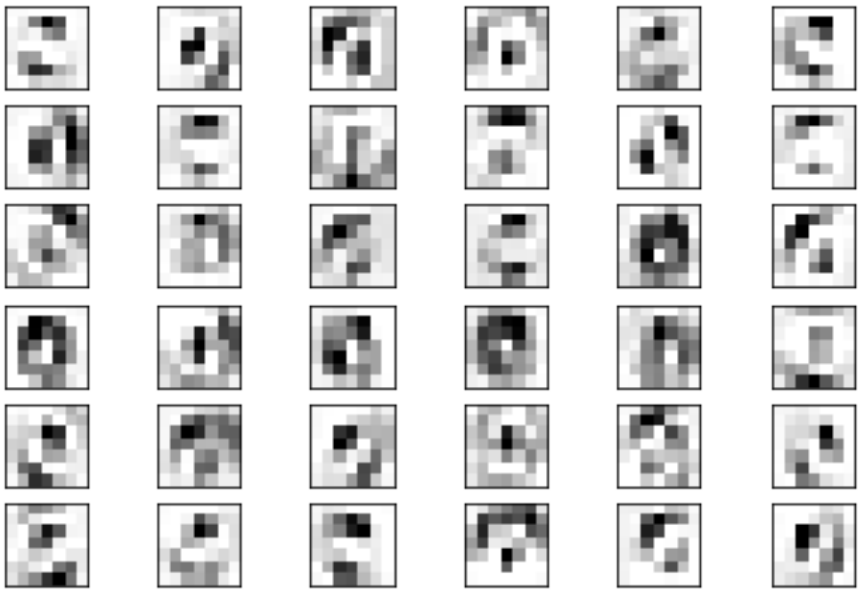
Note that these are down-sampled yet again to 7 x 7 pixels which is half the resolution of the images from the first conv-layer.

```
In [65]: plot_conv_layer(layer=layer_conv2, image=image1)
```



And these are the results of applying the filter-weights to the second image.

```
In [66]: plot_conv_layer(layer=layer_conv2, image=image2)
```



From these images, it looks like the second convolutional layer might detect lines and patterns in the input images, which are less sensitive to local variations in the original input images.

These images are then flattened and input to the fully-connected layer, but that is not shown here.

2.15.6 Close TensorFlow Session

We are now done using TensorFlow, so we close the session to release its resources.

```
In [67]: # This has been commented out in case you want to modify and experiment
         # with the Notebook without having to restart it.
         # session.close()
```

2.16 Conclusion

We have seen that a Convolutional Neural Network works much better at recognizing hand-written digits than the simple linear model in Tutorial #01. The Convolutional Network gets a classification accuracy of about 99%, or even more if you make some adjustments, compared to only 91% for the simple linear model.

However, the Convolutional Network is also much more complicated to implement, and it is not obvious from looking at the filter-weights why it works and why it sometimes fails.

So we would like an easier way to program Convolutional Neural Networks and we would also like a better way of visualizing their inner workings.

2.17 Exercises

These are a few suggestions for exercises that may help improve your skills with TensorFlow. It is important to get hands-on experience with TensorFlow in order to learn how to use it properly.

You may want to backup this Notebook before making any changes.

- Do you get the exact same results if you run the Notebook multiple times without changing any parameters? What are the sources of randomness?
- Run another 10,000 optimization iterations. Are the results better?
- Change the learning-rate for the optimizer.
- Change the configuration of the layers, such as the number of convolutional filters, the size of those filters, the number of neurons in the fully-connected layer, etc.
- Add a so-called drop-out layer after the fully-connected layer. Note that the drop-out probability should be zero when calculating the classification accuracy, so you will need a placeholder variable for this probability.
- Change the order of ReLU and max-pooling in the convolutional layer. Does it calculate the same thing? What is the fastest way of computing it? How many calculations are saved? Does it also work for Sigmoid-functions and average-pooling?
- Add one or more convolutional and fully-connected layers. Does it help performance?
- What is the smallest possible configuration that still gives good results?
- Try using ReLU in the last fully-connected layer. Does the performance change? Why?
- Try not using pooling in the convolutional layers. Does it change the classification accuracy and training time?
- Try using a 2x2 stride in the convolution instead of max-pooling? What is the difference?
- Remake the program yourself without looking too much at this source-code.
- Explain to a friend how the program works.

2.18 License (MIT)