TensorFlow Tutorial #06

CIFAR-10

by Magnus Erik Hvass Pedersen (http://www.hvass-labs.org/) / GitHub (https://github.com/Hvass-Labs/TensorFlow-Tutorials) / Videos on YouTube (https://www.youtube.com/playlist?list=PL9Hr9sNUjfsmEu1ZniY0XpHSzl5uihcXZ)

Introduction

This tutorial shows how to make a Convolutional Neural Network for classifying images in the CIFAR-10 data-set. It also shows how to use different networks during training and testing.

This builds on the previous tutorials, so you should have a basic understanding of TensorFlow and the add-on package Pretty Tensor. A lot of the source-code and text in this tutorial is similar to the previous tutorials and may be read quickly if you have recently read the previous tutorials.

Flowchart

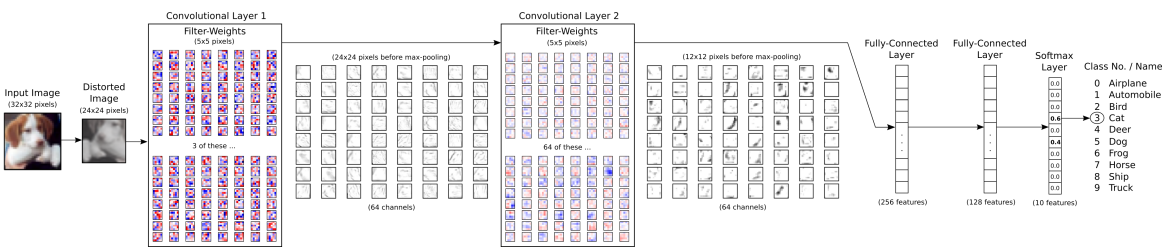
The following chart shows roughly how the data flows in the Convolutional Neural Network that is implemented below. First the network has a pre-processing layer which distorts the input images so as to artificially inflate the training-set. Then the network has two convolutional layers, two fully-connected layers and finally a softmax classification layer. The weights and outputs of the convolutional layers are shown in more detail in the larger plots further below, and Tutorial #02 goes into more detail on how convolution works.

In this case the image is mis-classified. The image actually shows a dog but the neural network is confused whether it is a dog or a cat and thinks the image is most likely a cat.

In [1]:

```
from IPython.display import Image
Image('images/06_network_flowchart.png')
```

Out[1]:



Imports

In [2]:

```
%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
import os

# Use PrettyTensor to simplify Neural Network construction.
import prettytensor as pt
```

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

In [3]:

```
tf.__version__
```

Out[3]:

```
'0.12.1'
```

PrettyTensor version:

In [4]:

```
pt.__version__
```

Out[4]:

```
'0.7.1'
```

Load Data

In [5]:

```
import cifar10
```

Set the path for storing the data-set on your computer.

In [6]:

```
cifar10.data_path = "data/CIFAR-10/"
```

The CIFAR-10 data-set is about 163 MB and will be downloaded automatically if it is not located in the given path.

In [7]:

```
cifar10.maybe_download_and_extract()
```

```
Data has apparently already been downloaded and unpacked.
```

Load the class-names.

In [8]:

```
class_names = cifar10.load_class_names()
class_names
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/batches.meta
```

Out[8]:

```
['not-cat', 'cat']
```

Load the training-set. This returns the images, the class-numbers as integers, and the class-numbers as One-Hot encoded arrays called labels.

In [9]:

```
images_train, cls_train, labels_train = cifar10.load_training_data()
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_1
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_2
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_3
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_4
Loading data: data/CIFAR-10/cifar-10-batches-py/data_batch_5
(50000,)
5000 50000
(10000,)
```

Load the test-set.

In [10]:

```
images_test, cls_test, labels_test = cifar10.load_test_data()
```

```
Loading data: data/CIFAR-10/cifar-10-batches-py/test_batch
1000 10000
```

The CIFAR-10 data-set has now been loaded and consists of 60,000 images and associated labels (i.e. classifications of the images). The data-set is split into 2 mutually exclusive sub-sets, the training-set and the test-set.

In [11]:

```
print("Size of:")
print("- Training-set:\t\t{}".format(len(images_train)))
print("- Test-set:\t\t{}".format(len(images_test)))
```

```
Size of:
- Training-set:         10000
- Test-set:             2000
```

Initial (10 classes) Size of:

- Training-set: 50000
- Test-set: 10000

Data Dimensions

The data dimensions are used in several places in the source-code below. They have already been defined in the cifar10 module, so we just need to import them.

In [12]:

```
from cifar10 import img_size, num_channels, num_classes
```

The images are 32 x 32 pixels, but we will crop the images to 24 x 24 pixels.

In [13]:

```
img_size_cropped = 24
```

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 [14]:

```
def plot_images(images, cls_true, cls_pred=None, smooth=True):

    assert len(images) == len(cls_true) == 9

    # Create figure with sub-plots.
    fig, axes = plt.subplots(3, 3)

    # Adjust vertical spacing if we need to print ensemble and best-net.
    if cls_pred is None:
        hspace = 0.3
    else:
        hspace = 0.6
    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    for i, ax in enumerate(axes.flat):
        # Interpolation type.
        if smooth:
            interpolation = 'spline16'
        else:
            interpolation = 'nearest'

        # Plot image.
        ax.imshow(images[i, :, :, :],
                  interpolation=interpolation)

        # Name of the true class.
        cls_true_name = class_names[cls_true[i]]

        # Show true and predicted classes.
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true_name)
        else:
            # Name of the predicted class.
            cls_pred_name = class_names[cls_pred[i]]

            xlabel = "True: {0}\nPred: {1}".format(cls_true_name, cls_pred_name)

        # 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()
```

Plot a few images to see if data is correct

In [15]:

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

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

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

The pixelated images above are what the neural network will get as input. The images might be a bit easier for the human eye to recognize if we smoothen the pixels.

In [16]:

```
plot_images(images=images, cls_true=cls_true, smooth=True)
```

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 loss 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.

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 array. The data-type is set to `float32` and the shape is set to `[None, img_size, img_size, num_channels]`, where `None` means that the tensor may hold an arbitrary number of images with each image being `img_size` pixels high and `img_size` pixels wide and with `num_channels` colour channels.

In [17]:

```
x = tf.placeholder(tf.float32, shape=[None, img_size, img_size, num_channels], name='x')
```

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 [18]:

```
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 [19]:

```
y_true_cls = tf.argmax(y_true, dimension=1)
```

Helper-function for creating Pre-Processing

The following helper-functions create the part of the TensorFlow computational graph that pre-processes the input images. Nothing is actually calculated at this point, the function merely adds nodes to the computational graph for TensorFlow.

The pre-processing is different for training and testing of the neural network:

- For training, the input images are randomly cropped, randomly flipped horizontally, and the hue, contrast and saturation is adjusted with random values. This artificially inflates the size of the training-set by creating random variations of the original input images. Examples of distorted images are shown further below.
- For testing, the input images are cropped around the centre and nothing else is adjusted.

In [20]:

```
def pre_process_image(image, training):
    # This function takes a single image as input,
    # and a boolean whether to build the training or testing graph.

    if training:
        # For training, add the following to the TensorFlow graph.

        # Randomly crop the input image.
        image = tf.random_crop(image, size=[img_size_cropped, img_size_cropped,
num_channels])

        # Randomly flip the image horizontally.
        image = tf.image.random_flip_left_right(image)

        # Randomly adjust hue, contrast and saturation.
        image = tf.image.random_hue(image, max_delta=0.05)
        image = tf.image.random_contrast(image, lower=0.3, upper=1.0)
        image = tf.image.random_brightness(image, max_delta=0.2)
        image = tf.image.random_saturation(image, lower=0.0, upper=2.0)

        # Some of these functions may overflow and result in pixel
        # values beyond the [0, 1] range. It is unclear from the
        # documentation of TensorFlow 0.10.0rc0 whether this is
        # intended. A simple solution is to limit the range.

        # Limit the image pixels between [0, 1] in case of overflow.
        image = tf.minimum(image, 1.0)
        image = tf.maximum(image, 0.0)
    else:
        # For training, add the following to the TensorFlow graph.

        # Crop the input image around the centre so it is the same
        # size as images that are randomly cropped during training.
        image = tf.image.resize_image_with_crop_or_pad(image,
                                                       target_height=img_size_cr
opped,
                                                       target_width=img_size_cro
pped)

    return image
```

The function above is called for each image in the input batch using the following function.

In [21]:

```
def pre_process(images, training):
    # Use TensorFlow to loop over all the input images and call
    # the function above which takes a single image as input.
    images = tf.map_fn(lambda image: pre_process_image(image, training), images)

    return images
```

In order to plot the distorted images, we create the pre-processing graph for TensorFlow, so we may execute it later.

In [22]:

```
distorted_images = pre_process(images=x, training=True)
```

Helper-function for creating Main Processing

The following helper-function creates the main part of the convolutional neural network. It uses Pretty Tensor which was described in the previous tutorials.

In [23]:

```
def main_network(images, training):
    # Wrap the input images as a Pretty Tensor object.
    x_pretty = pt.wrap(images)

    # Pretty Tensor uses special numbers to distinguish between
    # the training and testing phases.
    if training:
        phase = pt.Phase.train
    else:
        phase = pt.Phase.infer

    # Create the convolutional neural network using Pretty Tensor.
    # It is very similar to the previous tutorials, except
    # the use of so-called batch-normalization in the first layer.
    with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase):
        y_pred, loss = x_pretty.\
            conv2d(kernel=5, depth=64, name='layer_conv1',
batch_normalize=True).\
            max_pool(kernel=2, stride=2).\
            conv2d(kernel=5, depth=64, name='layer_conv2').\
            max_pool(kernel=2, stride=2).\
            flatten().\
            fully_connected(size=256, name='layer_fc1').\
            fully_connected(size=128, name='layer_fc2').\
            softmax_classifier(num_classes=num_classes, labels=y_true)

    return y_pred, loss
```

Helper-function for creating Neural Network

The following helper-function creates the full neural network, which consists of the pre-processing and main-processing defined above.

Note that the neural network is enclosed in the variable-scope named 'network'. This is because we are actually creating two neural networks in the TensorFlow graph. By assigning a variable-scope like this, we can re-use the variables for the two neural networks, so the variables that are optimized for the training-network are re-used for the other network that is used for testing.

In [24]:

```
def create_network(training):
    # Wrap the neural network in the scope named 'network'.
    # Create new variables during training, and re-use during testing.
    with tf.variable_scope('network', reuse=not training):
        # Just rename the input placeholder variable for convenience.
        images = x

        # Create TensorFlow graph for pre-processing.
        images = pre_process(images=images, training=training)

        # Create TensorFlow graph for the main processing.
        y_pred, loss = main_network(images=images, training=training)

    return y_pred, loss
```

Create Neural Network for Training Phase

First create a TensorFlow variable that keeps track of the number of optimization iterations performed so far. In the previous tutorials this was a Python variable, but in this tutorial we want to save this variable with all the other TensorFlow variables in the checkpoints.

Note that `trainable=False` which means that TensorFlow will not try to optimize this variable.

In [25]:

```
global_step = tf.Variable(initial_value=0,
                          name='global_step', trainable=False)
```

Create the neural network to be used for training. The `create_network()` function returns both `y_pred` and `loss`, but we only need the `loss`-function during training.

In [26]:

```
_, loss = create_network(training=True)
```

Create an optimizer which will minimize the `loss`-function. Also pass the `global_step` variable to the optimizer so it will be increased by one after each iteration.

In [27]:

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss, global_ste
p=global_step)
```

Create Neural Network for Test Phase / Inference

Now create the neural network for the test-phase. Once again the `create_network()` function returns the predicted class-labels `y_pred` for the input images, as well as the `loss`-function to be used during optimization. During testing we only need `y_pred`.

In [28]:

```
y_pred, _ = create_network(training=False)
```

We then calculate the predicted class number as an integer. The output of the network `y_pred` is an array with 10 elements. The class number is the index of the largest element in the array.

In [29]:

```
y_pred_cls = tf.argmax(y_pred, dimension=1)
```

Then we create a vector of booleans telling us whether the predicted class equals the true class of each image.

In [30]:

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
```

The classification accuracy is calculated by first type-casting the vector of booleans to floats, so that False becomes 0 and True becomes 1, and then taking the average of these numbers.

In [31]:

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Saver

In order to save the variables of the neural network, so they can be reloaded quickly without having to train the network again, we now create a so-called Saver-object which is used for storing and retrieving all the variables of the TensorFlow graph. Nothing is actually saved at this point, which will be done further below.

In [32]:

```
saver = tf.train.Saver()
```

Getting the Weights

Further below, we want to plot the weights of the neural network. When the network is constructed using Pretty Tensor, all the variables of the layers are created indirectly by Pretty Tensor. We therefore have to retrieve the variables from TensorFlow.

We used the names `layer_conv1` and `layer_conv2` for the two convolutional layers. These are also called variable scopes. Pretty Tensor automatically gives names to the variables it creates for each layer, so we can retrieve the weights for a layer using the layer's scope-name and the variable-name.

The implementation is somewhat awkward because we have to use the TensorFlow function `get_variable()` which was designed for another purpose; either creating a new variable or re-using an existing variable. The easiest thing is to make the following helper-function.

In [33]:

```
def get_weights_variable(layer_name):
    # Retrieve an existing variable named 'weights' in the scope
    # with the given layer_name.
    # This is awkward because the TensorFlow function was
    # really intended for another purpose.

    with tf.variable_scope("network/" + layer_name, reuse=True):
        variable = tf.get_variable('weights')

    return variable
```

Using this helper-function we can retrieve the variables. These are TensorFlow objects. In order to get the contents of the variables, you must do something like: `contents = session.run(weights_conv1)` as demonstrated further below.

In [34]:

```
weights_conv1 = get_weights_variable(layer_name='layer_conv1')
weights_conv2 = get_weights_variable(layer_name='layer_conv2')
```

Getting the Layer Outputs

Similarly we also need to retrieve the outputs of the convolutional layers. The function for doing this is slightly different than the function above for getting the weights. Here we instead retrieve the last tensor that is output by the convolutional layer.

In [35]:

```
def get_layer_output(layer_name):
    # The name of the last operation of the convolutional layer.
    # This assumes you are using Relu as the activation-function.
    tensor_name = "network/" + layer_name + "/Relu:0"

    # Get the tensor with this name.
    tensor = tf.get_default_graph().get_tensor_by_name(tensor_name)

    return tensor
```

Get the output of the convoluational layers so we can plot them later.

In [36]:

```
output_conv1 = get_layer_output(layer_name='layer_conv1')
output_conv2 = get_layer_output(layer_name='layer_conv2')
```

TensorFlow Run

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 [37]:

```
session = tf.Session()
```

Restore or initialize variables

Training this neural network may take a long time, especially if you do not have a GPU. We therefore save checkpoints during training so we can continue training at another time (e.g. during the night), and also for performing analysis later without having to train the neural network every time we want to use it.

If you want to restart the training of the neural network, you have to delete the checkpoints first.

This is the directory used for the checkpoints.

In [38]:

```
save_dir = 'checkpoints/'
```

Create the directory if it does not exist.

In [39]:

```
if not os.path.exists(save_dir):
    os.makedirs(save_dir)
```

This is the base-filename for the checkpoints, TensorFlow will append the iteration number, etc.

In [40]:

```
save_path = os.path.join(save_dir, 'cifar10_cnn')
```

First try to restore the latest checkpoint. This may fail and raise an exception e.g. if such a checkpoint does not exist, or if you have changed the TensorFlow graph.

In [41]:

```
try:
    print("Trying to restore last checkpoint ...")

    # Use TensorFlow to find the latest checkpoint - if any.
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=save_dir)

    # Try and load the data in the checkpoint.
    saver.restore(session, save_path=last_chk_path)

    # If we get to this point, the checkpoint was successfully loaded.
    print("Restored checkpoint from:", last_chk_path)
except:
    # If the above failed for some reason, simply
    # initialize all the variables for the TensorFlow graph.
    print("Failed to restore checkpoint. Initializing variables instead.")
    session.run(tf.global_variables_initializer())
```

```
Trying to restore last checkpoint ...
Restored checkpoint from: checkpoints/cifar10_cnn-50000
```

Helper-function to get a random training-batch

There are 50,000 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 [42]:

```
train_batch_size = 64
```

Function for selecting a random batch of images from the training-set.

In [43]:

```
def random_batch():
    # Number of images in the training-set.
    num_images = len(images_train)

    # Create a random index.
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)

    # Use the random index to select random images and labels.
    x_batch = images_train[idx, :, :, :]
    y_batch = labels_train[idx, :]

    return x_batch, y_batch
```

Helper-function to perform optimization

This function performs 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. A checkpoint is saved every 1000 iterations and also after the last iteration.

In [44]:

```
def optimize(num_iterations):
    # Start-time used for printing time-usage below.
    start_time = time.time()

    for i in range(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 = random_batch()

        # 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.
        # We also want to retrieve the global_step counter.
        i_global, _ = session.run([global_step, optimizer],
                                  feed_dict=feed_dict_train)

        # Print status to screen every 100 iterations (and last).
        if (i_global % 100 == 0) or (i == num_iterations - 1):
            # Calculate the accuracy on the training-batch.
            batch_acc = session.run(accuracy,
                                    feed_dict=feed_dict_train)

            # Print status.
            msg = "Global Step: {0:>6}, Training Batch Accuracy: {1:>6.1%}"
            print(msg.format(i_global, batch_acc))

        # Save a checkpoint to disk every 1000 iterations (and last).
        if (i_global % 1000 == 0) or (i == num_iterations - 1):
            # Save all variables of the TensorFlow graph to a
            # checkpoint. Append the global_step counter
            # to the filename so we save the last several checkpoints.
            saver.save(session,
                       save_path=save_path,
                       global_step=global_step)

            print("Saved checkpoint.")

    # 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)))))
```

Helper-function to plot example errors

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

In [45]:

```
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 = images_test[incorrect]

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

    # Get the true classes for those images.
    cls_true = cls_test[incorrect]

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

Helper-function to plot confusion matrix

In [46]:

```
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 confusion matrix using sklearn.
    cm = confusion_matrix(y_true=cls_test,  # True class for test-set.
                          y_pred=cls_pred)  # Predicted class.

    # Print the confusion matrix as text.
    for i in range(num_classes):
        # Append the class-name to each line.
        class_name = "({}) {}".format(i, class_names[i])
        print(cm[i, :], class_name)

    # Print the class-numbers for easy reference.
    class_numbers = [" ({0})".format(i) for i in range(num_classes)]
    print("".join(class_numbers))
```

Helper-functions for calculating classifications

This function calculates the predicted classes of images and also returns a boolean array whether the classification of each image is correct.

The calculation is done in batches because it might use too much RAM otherwise. If your computer crashes then you can try and lower the batch-size.

In [47]:

```
# Split the data-set in batches of this size to limit RAM usage.
batch_size = 128

def predict_cls(images, labels, cls_true):
    # Number of images.
    num_images = len(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_images, 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_images:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_images)

        # Create a feed-dict with the images and labels
        # between index i and j.
        feed_dict = {x: images[i:j, :],
                     y_true: labels[i:j, :]}

        # 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

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)
    print ('LALALA2',cls_true,cls_pred, len(cls_true) ,len(cls_pred) , correct)

    return correct, cls_pred
```

Calculate the predicted class for the test-set.

In [48]:

```
def predict_cls_test():
    return predict_cls(images = images_test,
                       labels = labels_test,
                       cls_true = cls_test)
```

Helper-functions for the classification accuracy

This function calculates the classification accuracy given a boolean array whether each image was correctly classified. E.g. `classification_accuracy([True, True, False, False, False]) = 2/5 = 0.4`. The function also returns the number of correct classifications.

In [49]:

```
def classification_accuracy(correct):
    # When averaging a boolean array, False means 0 and True means 1.
    # So we are calculating: number of True / len(correct) which is
    # the same as the classification accuracy.

    # Return the classification accuracy
    # and the number of correct classifications.
    return correct.mean(), correct.sum()
```

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.

In [50]:

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

    # For all the images in the test-set,
    # calculate the predicted classes and whether they are correct.
    correct, cls_pred = predict_cls_test()

    # Classification accuracy and the number of correct classifications.
    acc, num_correct = classification_accuracy(correct)

    # Number of images being classified.
    num_images = len(correct)

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

    # 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)
```

Helper-function for plotting convolutional weights

In [51]:

```
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)

    # Print statistics for the weights.
    print("Min:  {0:.5f}, Max:   {1:.5f}".format(w.min(), w.max()))
    print("Mean: {0:.5f}, Stdev: {1:.5f}".format(w.mean(), w.std()))

    # 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)
    abs_max = max(abs(w_min), abs(w_max))

    # 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.
            # The format of this 4-dim tensor is determined by the
            # TensorFlow API. See Tutorial #02 for more details.
            img = w[:, :, input_channel, i]

            # Plot image.
            ax.imshow(img, vmin=-abs_max, vmax=abs_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()
```

Helper-function for plotting the output of convolutional layers

In [52]:

```
def plot_layer_output(layer_output, image):
    # Assume layer_output is a 4-dim tensor
    # e.g. output_conv1 or output_conv2.

    # Create a feed-dict which holds the single input image.
    # Note that TensorFlow needs a list of images,
    # so we just create a list with this one image.
    feed_dict = {x: [image]}

    # Retrieve the output of the layer after inputting this image.
    values = session.run(layer_output, feed_dict=feed_dict)

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

    # Number of image channels output by the conv. layer.
    num_images = values.shape[3]

    # Number of grid-cells to plot.
    # Rounded-up, square-root of the number of filters.
    num_grids = math.ceil(math.sqrt(num_images))

    # 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 image-channels.
        if i<num_images:
            # Get the images for the i'th output channel.
            img = values[0, :, :, i]

            # Plot image.
            ax.imshow(img, vmin=values_min, vmax=values_max,
                      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()
```

Examples of distorted input images

In order to artificially inflate the number of images available for training, the neural network uses pre-processing with random distortions of the input images. This should hopefully make the neural network more flexible at recognizing and classifying images.

This is a helper-function for plotting distorted input images.

In [53]:

```
def plot_distorted_image(image, cls_true):
    # Repeat the input image 9 times.
    image_duplicates = np.repeat(image[np.newaxis, :, :, :], 9, axis=0)

    # Create a feed-dict for TensorFlow.
    feed_dict = {x: image_duplicates}

    # Calculate only the pre-processing of the TensorFlow graph
    # which distorts the images in the feed-dict.
    result = session.run(distorted_images, feed_dict=feed_dict)

    # Plot the images.
    plot_images(images=result, cls_true=np.repeat(cls_true, 9))
```

Helper-function for getting an image and its class-number from the test-set.

In [54]:

```
def get_test_image(i):
    return images_test[i, :, :, :], cls_test[i]
```

Get an image and its true class from the test-set.

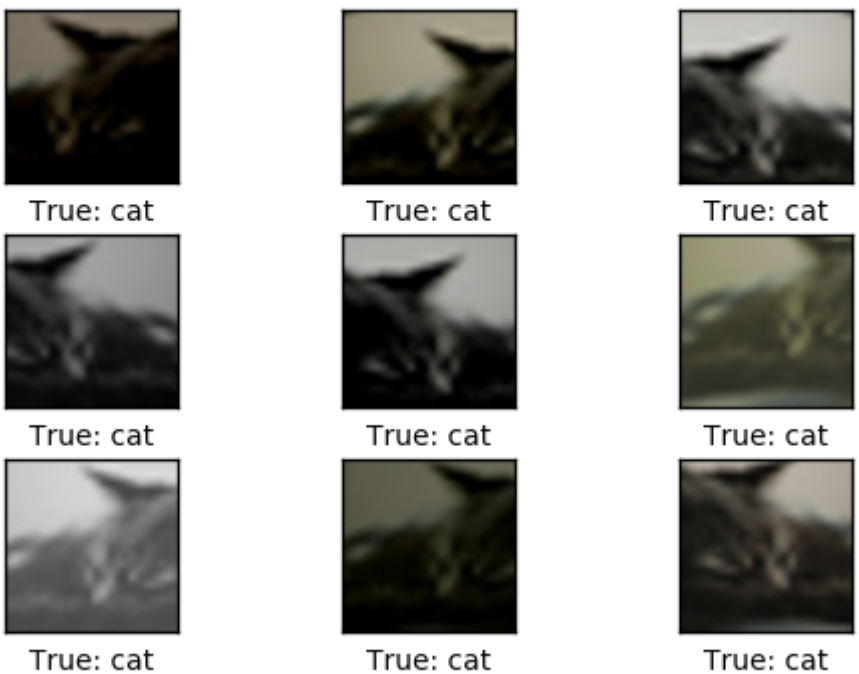
In [55]:

```
img, cls = get_test_image(16)
```

Plot 9 random distortions of the image. If you re-run this code you will get slightly different results.

In [56]:

```
plot_distorted_image(img, cls)
```



Perform optimization

My laptop computer is a Quad-Core with 2 GHz per core. It has a GPU but it is not fast enough for TensorFlow so it only uses the CPU. It takes about 1 hour to perform 10,000 optimization iterations using the CPU on this PC. For this tutorial I performed 150,000 optimization iterations so that took about 15 hours. I let it run during the night and at various points during the day.

Because we are saving the checkpoints during optimization, and because we are restoring the latest checkpoint when restarting the code, we can stop and continue the optimization later.

In [58]:

```
if False:
    optimize(num_iterations=50000)
```

```
Global Step:    100, Training Batch Accuracy:  71.9%
Global Step:    200, Training Batch Accuracy:  67.2%
Global Step:    300, Training Batch Accuracy:  78.1%
Global Step:    400, Training Batch Accuracy:  76.6%
Global Step:    500, Training Batch Accuracy:  73.4%
Global Step:    600, Training Batch Accuracy:  71.9%
Global Step:    700, Training Batch Accuracy:  76.6%
Global Step:    800, Training Batch Accuracy:  78.1%
Global Step:    900, Training Batch Accuracy:  75.0%
Global Step:   1000, Training Batch Accuracy:  59.4%
Saved checkpoint.
Global Step:   1100, Training Batch Accuracy:  70.3%
Global Step:   1200, Training Batch Accuracy:  65.6%
Global Step:   1300, Training Batch Accuracy:  78.1%
Global Step:   1400, Training Batch Accuracy:  73.4%
Global Step:   1500, Training Batch Accuracy:  81.2%
Global Step:   1600, Training Batch Accuracy:  75.0%
Global Step:   1700, Training Batch Accuracy:  67.2%
Global Step:   1800, Training Batch Accuracy:  62.5%
Global Step:   1900, Training Batch Accuracy:  79.7%
Global Step:   2000, Training Batch Accuracy:  76.6%
Saved checkpoint.
Global Step:   2100, Training Batch Accuracy:  76.6%
Global Step:   2200, Training Batch Accuracy:  79.7%
Global Step:   2300, Training Batch Accuracy:  73.4%
Global Step:   2400, Training Batch Accuracy:  82.8%
Global Step:   2500, Training Batch Accuracy:  81.2%
Global Step:   2600, Training Batch Accuracy:  76.6%
Global Step:   2700, Training Batch Accuracy:  73.4%
Global Step:   2800, Training Batch Accuracy:  81.2%
Global Step:   2900, Training Batch Accuracy:  82.8%
Global Step:   3000, Training Batch Accuracy:  79.7%
Saved checkpoint.
Global Step:   3100, Training Batch Accuracy:  76.6%
Global Step:   3200, Training Batch Accuracy:  68.8%
Global Step:   3300, Training Batch Accuracy:  78.1%
Global Step:   3400, Training Batch Accuracy:  82.8%
Global Step:   3500, Training Batch Accuracy:  81.2%
Global Step:   3600, Training Batch Accuracy:  79.7%
Global Step:   3700, Training Batch Accuracy:  71.9%
Global Step:   3800, Training Batch Accuracy:  75.0%
Global Step:   3900, Training Batch Accuracy:  78.1%
Global Step:   4000, Training Batch Accuracy:  82.8%
Saved checkpoint.
Global Step:   4100, Training Batch Accuracy:  82.8%
Global Step:   4200, Training Batch Accuracy:  73.4%
Global Step:   4300, Training Batch Accuracy:  85.9%
Global Step:   4400, Training Batch Accuracy:  79.7%
Global Step:   4500, Training Batch Accuracy:  71.9%
Global Step:   4600, Training Batch Accuracy:  85.9%
Global Step:   4700, Training Batch Accuracy:  87.5%
Global Step:   4800, Training Batch Accuracy:  81.2%
Global Step:   4900, Training Batch Accuracy:  76.6%
Global Step:   5000, Training Batch Accuracy:  81.2%
Saved checkpoint.
Global Step:   5100, Training Batch Accuracy:  73.4%
Global Step:   5200, Training Batch Accuracy:  73.4%
Global Step:   5300, Training Batch Accuracy:  81.2%
Global Step:   5400, Training Batch Accuracy:  79.7%
Global Step:   5500, Training Batch Accuracy:  82.8%
Global Step:   5600, Training Batch Accuracy:  78.1%
```

```
Global Step:   5700, Training Batch Accuracy:  85.9%
Global Step:   5800, Training Batch Accuracy:  78.1%
Global Step:   5900, Training Batch Accuracy:  79.7%
Global Step:   6000, Training Batch Accuracy:  89.1%
Saved checkpoint.
Global Step:   6100, Training Batch Accuracy:  79.7%
Global Step:   6200, Training Batch Accuracy:  79.7%
Global Step:   6300, Training Batch Accuracy:  84.4%
Global Step:   6400, Training Batch Accuracy:  79.7%
Global Step:   6500, Training Batch Accuracy:  71.9%
Global Step:   6600, Training Batch Accuracy:  85.9%
Global Step:   6700, Training Batch Accuracy:  85.9%
Global Step:   6800, Training Batch Accuracy:  82.8%
Global Step:   6900, Training Batch Accuracy:  82.8%
Global Step:   7000, Training Batch Accuracy:  73.4%
Saved checkpoint.
Global Step:   7100, Training Batch Accuracy:  81.2%
Global Step:   7200, Training Batch Accuracy:  84.4%
Global Step:   7300, Training Batch Accuracy:  90.6%
Global Step:   7400, Training Batch Accuracy:  71.9%
Global Step:   7500, Training Batch Accuracy:  84.4%
Global Step:   7600, Training Batch Accuracy:  81.2%
Global Step:   7700, Training Batch Accuracy:  78.1%
Global Step:   7800, Training Batch Accuracy:  82.8%
Global Step:   7900, Training Batch Accuracy:  84.4%
Global Step:   8000, Training Batch Accuracy:  82.8%
Saved checkpoint.
Global Step:   8100, Training Batch Accuracy:  81.2%
Global Step:   8200, Training Batch Accuracy:  85.9%
Global Step:   8300, Training Batch Accuracy:  76.6%
Global Step:   8400, Training Batch Accuracy:  81.2%
Global Step:   8500, Training Batch Accuracy:  78.1%
Global Step:   8600, Training Batch Accuracy:  82.8%
Global Step:   8700, Training Batch Accuracy:  85.9%
Global Step:   8800, Training Batch Accuracy:  78.1%
Global Step:   8900, Training Batch Accuracy:  81.2%
Global Step:   9000, Training Batch Accuracy:  81.2%
Saved checkpoint.
Global Step:   9100, Training Batch Accuracy:  78.1%
Global Step:   9200, Training Batch Accuracy:  89.1%
Global Step:   9300, Training Batch Accuracy:  76.6%
Global Step:   9400, Training Batch Accuracy:  87.5%
Global Step:   9500, Training Batch Accuracy:  81.2%
Global Step:   9600, Training Batch Accuracy:  82.8%
Global Step:   9700, Training Batch Accuracy:  84.4%
Global Step:   9800, Training Batch Accuracy:  85.9%
Global Step:   9900, Training Batch Accuracy:  82.8%
Global Step:  10000, Training Batch Accuracy:  82.8%
Saved checkpoint.
Global Step:  10100, Training Batch Accuracy:  85.9%
Global Step:  10200, Training Batch Accuracy:  82.8%
Global Step:  10300, Training Batch Accuracy:  84.4%
Global Step:  10400, Training Batch Accuracy:  82.8%
Global Step:  10500, Training Batch Accuracy:  87.5%
Global Step:  10600, Training Batch Accuracy:  82.8%
Global Step:  10700, Training Batch Accuracy:  82.8%
Global Step:  10800, Training Batch Accuracy:  79.7%
Global Step:  10900, Training Batch Accuracy:  82.8%
Global Step:  11000, Training Batch Accuracy:  87.5%
Saved checkpoint.
Global Step:  11100, Training Batch Accuracy:  82.8%
```

```
Global Step:  11200, Training Batch Accuracy:  82.8%
Global Step:  11300, Training Batch Accuracy:  79.7%
Global Step:  11400, Training Batch Accuracy:  89.1%
Global Step:  11500, Training Batch Accuracy:  70.3%
Global Step:  11600, Training Batch Accuracy:  79.7%
Global Step:  11700, Training Batch Accuracy:  75.0%
Global Step:  11800, Training Batch Accuracy:  89.1%
Global Step:  11900, Training Batch Accuracy:  84.4%
Global Step:  12000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  12100, Training Batch Accuracy:  89.1%
Global Step:  12200, Training Batch Accuracy:  84.4%
Global Step:  12300, Training Batch Accuracy:  85.9%
Global Step:  12400, Training Batch Accuracy:  85.9%
Global Step:  12500, Training Batch Accuracy:  90.6%
Global Step:  12600, Training Batch Accuracy:  82.8%
Global Step:  12700, Training Batch Accuracy:  87.5%
Global Step:  12800, Training Batch Accuracy:  93.8%
Global Step:  12900, Training Batch Accuracy:  87.5%
Global Step:  13000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  13100, Training Batch Accuracy:  87.5%
Global Step:  13200, Training Batch Accuracy:  82.8%
Global Step:  13300, Training Batch Accuracy:  85.9%
Global Step:  13400, Training Batch Accuracy:  84.4%
Global Step:  13500, Training Batch Accuracy:  75.0%
Global Step:  13600, Training Batch Accuracy:  85.9%
Global Step:  13700, Training Batch Accuracy:  82.8%
Global Step:  13800, Training Batch Accuracy:  75.0%
Global Step:  13900, Training Batch Accuracy:  78.1%
Global Step:  14000, Training Batch Accuracy:  78.1%
Saved checkpoint.
Global Step:  14100, Training Batch Accuracy:  89.1%
Global Step:  14200, Training Batch Accuracy:  75.0%
Global Step:  14300, Training Batch Accuracy:  79.7%
Global Step:  14400, Training Batch Accuracy:  93.8%
Global Step:  14500, Training Batch Accuracy:  78.1%
Global Step:  14600, Training Batch Accuracy:  87.5%
Global Step:  14700, Training Batch Accuracy:  90.6%
Global Step:  14800, Training Batch Accuracy:  85.9%
Global Step:  14900, Training Batch Accuracy:  84.4%
Global Step:  15000, Training Batch Accuracy:  81.2%
Saved checkpoint.
Global Step:  15100, Training Batch Accuracy:  87.5%
Global Step:  15200, Training Batch Accuracy:  89.1%
Global Step:  15300, Training Batch Accuracy:  87.5%
Global Step:  15400, Training Batch Accuracy:  89.1%
Global Step:  15500, Training Batch Accuracy:  81.2%
Global Step:  15600, Training Batch Accuracy:  82.8%
Global Step:  15700, Training Batch Accuracy:  76.6%
Global Step:  15800, Training Batch Accuracy:  87.5%
Global Step:  15900, Training Batch Accuracy:  82.8%
Global Step:  16000, Training Batch Accuracy:  89.1%
Saved checkpoint.
Global Step:  16100, Training Batch Accuracy:  81.2%
Global Step:  16200, Training Batch Accuracy:  89.1%
Global Step:  16300, Training Batch Accuracy:  81.2%
Global Step:  16400, Training Batch Accuracy:  90.6%
Global Step:  16500, Training Batch Accuracy:  84.4%
Global Step:  16600, Training Batch Accuracy:  79.7%
Global Step:  16700, Training Batch Accuracy:  93.8%
```

```
Global Step:  16800, Training Batch Accuracy:  87.5%
Global Step:  16900, Training Batch Accuracy:  90.6%
Global Step:  17000, Training Batch Accuracy:  84.4%
Saved checkpoint.
Global Step:  17100, Training Batch Accuracy:  82.8%
Global Step:  17200, Training Batch Accuracy:  85.9%
Global Step:  17300, Training Batch Accuracy:  89.1%
Global Step:  17400, Training Batch Accuracy:  82.8%
Global Step:  17500, Training Batch Accuracy:  90.6%
Global Step:  17600, Training Batch Accuracy:  85.9%
Global Step:  17700, Training Batch Accuracy:  78.1%
Global Step:  17800, Training Batch Accuracy:  89.1%
Global Step:  17900, Training Batch Accuracy:  85.9%
Global Step:  18000, Training Batch Accuracy:  85.9%
Saved checkpoint.
Global Step:  18100, Training Batch Accuracy:  84.4%
Global Step:  18200, Training Batch Accuracy:  90.6%
Global Step:  18300, Training Batch Accuracy:  92.2%
Global Step:  18400, Training Batch Accuracy:  95.3%
Global Step:  18500, Training Batch Accuracy:  89.1%
Global Step:  18600, Training Batch Accuracy:  84.4%
Global Step:  18700, Training Batch Accuracy:  92.2%
Global Step:  18800, Training Batch Accuracy:  87.5%
Global Step:  18900, Training Batch Accuracy:  84.4%
Global Step:  19000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  19100, Training Batch Accuracy:  89.1%
Global Step:  19200, Training Batch Accuracy:  90.6%
Global Step:  19300, Training Batch Accuracy:  78.1%
Global Step:  19400, Training Batch Accuracy:  85.9%
Global Step:  19500, Training Batch Accuracy:  89.1%
Global Step:  19600, Training Batch Accuracy:  93.8%
Global Step:  19700, Training Batch Accuracy:  87.5%
Global Step:  19800, Training Batch Accuracy:  87.5%
Global Step:  19900, Training Batch Accuracy:  95.3%
Global Step:  20000, Training Batch Accuracy:  87.5%
Saved checkpoint.
Global Step:  20100, Training Batch Accuracy:  85.9%
Global Step:  20200, Training Batch Accuracy:  85.9%
Global Step:  20300, Training Batch Accuracy:  93.8%
Global Step:  20400, Training Batch Accuracy:  92.2%
Global Step:  20500, Training Batch Accuracy:  84.4%
Global Step:  20600, Training Batch Accuracy:  98.4%
Global Step:  20700, Training Batch Accuracy:  89.1%
Global Step:  20800, Training Batch Accuracy:  84.4%
Global Step:  20900, Training Batch Accuracy:  89.1%
Global Step:  21000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  21100, Training Batch Accuracy:  90.6%
Global Step:  21200, Training Batch Accuracy:  90.6%
Global Step:  21300, Training Batch Accuracy:  90.6%
Global Step:  21400, Training Batch Accuracy:  92.2%
Global Step:  21500, Training Batch Accuracy:  78.1%
Global Step:  21600, Training Batch Accuracy:  85.9%
Global Step:  21700, Training Batch Accuracy:  89.1%
Global Step:  21800, Training Batch Accuracy:  95.3%
Global Step:  21900, Training Batch Accuracy:  90.6%
Global Step:  22000, Training Batch Accuracy:  82.8%
Saved checkpoint.
Global Step:  22100, Training Batch Accuracy:  87.5%
Global Step:  22200, Training Batch Accuracy:  92.2%
```

```
Global Step:  22300, Training Batch Accuracy:  90.6%
Global Step:  22400, Training Batch Accuracy:  92.2%
Global Step:  22500, Training Batch Accuracy:  95.3%
Global Step:  22600, Training Batch Accuracy:  95.3%
Global Step:  22700, Training Batch Accuracy:  92.2%
Global Step:  22800, Training Batch Accuracy:  87.5%
Global Step:  22900, Training Batch Accuracy:  93.8%
Global Step:  23000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  23100, Training Batch Accuracy:  89.1%
Global Step:  23200, Training Batch Accuracy:  93.8%
Global Step:  23300, Training Batch Accuracy:  93.8%
Global Step:  23400, Training Batch Accuracy:  93.8%
Global Step:  23500, Training Batch Accuracy:  93.8%
Global Step:  23600, Training Batch Accuracy:  87.5%
Global Step:  23700, Training Batch Accuracy:  82.8%
Global Step:  23800, Training Batch Accuracy:  84.4%
Global Step:  23900, Training Batch Accuracy:  89.1%
Global Step:  24000, Training Batch Accuracy:  82.8%
Saved checkpoint.
Global Step:  24100, Training Batch Accuracy:  82.8%
Global Step:  24200, Training Batch Accuracy:  90.6%
Global Step:  24300, Training Batch Accuracy:  82.8%
Global Step:  24400, Training Batch Accuracy:  90.6%
Global Step:  24500, Training Batch Accuracy:  84.4%
Global Step:  24600, Training Batch Accuracy:  95.3%
Global Step:  24700, Training Batch Accuracy:  90.6%
Global Step:  24800, Training Batch Accuracy:  85.9%
Global Step:  24900, Training Batch Accuracy:  93.8%
Global Step:  25000, Training Batch Accuracy:  93.8%
Saved checkpoint.
Global Step:  25100, Training Batch Accuracy:  90.6%
Global Step:  25200, Training Batch Accuracy:  87.5%
Global Step:  25300, Training Batch Accuracy:  87.5%
Global Step:  25400, Training Batch Accuracy:  79.7%
Global Step:  25500, Training Batch Accuracy:  89.1%
Global Step:  25600, Training Batch Accuracy:  90.6%
Global Step:  25700, Training Batch Accuracy:  93.8%
Global Step:  25800, Training Batch Accuracy:  92.2%
Global Step:  25900, Training Batch Accuracy:  87.5%
Global Step:  26000, Training Batch Accuracy:  81.2%
Saved checkpoint.
Global Step:  26100, Training Batch Accuracy:  92.2%
Global Step:  26200, Training Batch Accuracy:  92.2%
Global Step:  26300, Training Batch Accuracy:  90.6%
Global Step:  26400, Training Batch Accuracy:  81.2%
Global Step:  26500, Training Batch Accuracy:  92.2%
Global Step:  26600, Training Batch Accuracy:  84.4%
Global Step:  26700, Training Batch Accuracy:  90.6%
Global Step:  26800, Training Batch Accuracy: 100.0%
Global Step:  26900, Training Batch Accuracy:  82.8%
Global Step:  27000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  27100, Training Batch Accuracy:  89.1%
Global Step:  27200, Training Batch Accuracy:  85.9%
Global Step:  27300, Training Batch Accuracy:  93.8%
Global Step:  27400, Training Batch Accuracy:  87.5%
Global Step:  27500, Training Batch Accuracy:  87.5%
Global Step:  27600, Training Batch Accuracy:  90.6%
Global Step:  27700, Training Batch Accuracy:  96.9%
Global Step:  27800, Training Batch Accuracy:  92.2%
```

```
Global Step:  27900, Training Batch Accuracy:  90.6%
Global Step:  28000, Training Batch Accuracy:  85.9%
Saved checkpoint.
Global Step:  28100, Training Batch Accuracy:  85.9%
Global Step:  28200, Training Batch Accuracy:  85.9%
Global Step:  28300, Training Batch Accuracy:  85.9%
Global Step:  28400, Training Batch Accuracy:  89.1%
Global Step:  28500, Training Batch Accuracy:  93.8%
Global Step:  28600, Training Batch Accuracy:  89.1%
Global Step:  28700, Training Batch Accuracy:  90.6%
Global Step:  28800, Training Batch Accuracy:  87.5%
Global Step:  28900, Training Batch Accuracy:  93.8%
Global Step:  29000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  29100, Training Batch Accuracy:  93.8%
Global Step:  29200, Training Batch Accuracy:  84.4%
Global Step:  29300, Training Batch Accuracy:  95.3%
Global Step:  29400, Training Batch Accuracy: 100.0%
Global Step:  29500, Training Batch Accuracy:  93.8%
Global Step:  29600, Training Batch Accuracy:  93.8%
Global Step:  29700, Training Batch Accuracy:  96.9%
Global Step:  29800, Training Batch Accuracy:  89.1%
Global Step:  29900, Training Batch Accuracy:  93.8%
Global Step:  30000, Training Batch Accuracy:  95.3%
Saved checkpoint.
Global Step:  30100, Training Batch Accuracy:  85.9%
Global Step:  30200, Training Batch Accuracy:  98.4%
Global Step:  30300, Training Batch Accuracy:  90.6%
Global Step:  30400, Training Batch Accuracy:  90.6%
Global Step:  30500, Training Batch Accuracy:  87.5%
Global Step:  30600, Training Batch Accuracy:  92.2%
Global Step:  30700, Training Batch Accuracy:  89.1%
Global Step:  30800, Training Batch Accuracy:  90.6%
Global Step:  30900, Training Batch Accuracy:  84.4%
Global Step:  31000, Training Batch Accuracy:  89.1%
Saved checkpoint.
Global Step:  31100, Training Batch Accuracy:  95.3%
Global Step:  31200, Training Batch Accuracy:  95.3%
Global Step:  31300, Training Batch Accuracy:  92.2%
Global Step:  31400, Training Batch Accuracy:  96.9%
Global Step:  31500, Training Batch Accuracy:  85.9%
Global Step:  31600, Training Batch Accuracy:  85.9%
Global Step:  31700, Training Batch Accuracy:  90.6%
Global Step:  31800, Training Batch Accuracy:  87.5%
Global Step:  31900, Training Batch Accuracy:  95.3%
Global Step:  32000, Training Batch Accuracy: 100.0%
Saved checkpoint.
Global Step:  32100, Training Batch Accuracy:  96.9%
Global Step:  32200, Training Batch Accuracy:  93.8%
Global Step:  32300, Training Batch Accuracy:  92.2%
Global Step:  32400, Training Batch Accuracy:  84.4%
Global Step:  32500, Training Batch Accuracy:  84.4%
Global Step:  32600, Training Batch Accuracy:  95.3%
Global Step:  32700, Training Batch Accuracy:  95.3%
Global Step:  32800, Training Batch Accuracy:  89.1%
Global Step:  32900, Training Batch Accuracy:  85.9%
Global Step:  33000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  33100, Training Batch Accuracy:  92.2%
Global Step:  33200, Training Batch Accuracy:  95.3%
Global Step:  33300, Training Batch Accuracy:  82.8%
```

```
Global Step:  33400, Training Batch Accuracy:  92.2%
Global Step:  33500, Training Batch Accuracy:  87.5%
Global Step:  33600, Training Batch Accuracy:  95.3%
Global Step:  33700, Training Batch Accuracy:  95.3%
Global Step:  33800, Training Batch Accuracy:  92.2%
Global Step:  33900, Training Batch Accuracy:  89.1%
Global Step:  34000, Training Batch Accuracy:  89.1%
Saved checkpoint.
Global Step:  34100, Training Batch Accuracy:  92.2%
Global Step:  34200, Training Batch Accuracy:  93.8%
Global Step:  34300, Training Batch Accuracy:  92.2%
Global Step:  34400, Training Batch Accuracy:  92.2%
Global Step:  34500, Training Batch Accuracy:  90.6%
Global Step:  34600, Training Batch Accuracy:  95.3%
Global Step:  34700, Training Batch Accuracy:  87.5%
Global Step:  34800, Training Batch Accuracy:  95.3%
Global Step:  34900, Training Batch Accuracy:  92.2%
Global Step:  35000, Training Batch Accuracy:  93.8%
Saved checkpoint.
Global Step:  35100, Training Batch Accuracy:  82.8%
Global Step:  35200, Training Batch Accuracy:  90.6%
Global Step:  35300, Training Batch Accuracy:  92.2%
Global Step:  35400, Training Batch Accuracy:  93.8%
Global Step:  35500, Training Batch Accuracy:  93.8%
Global Step:  35600, Training Batch Accuracy:  95.3%
Global Step:  35700, Training Batch Accuracy:  92.2%
Global Step:  35800, Training Batch Accuracy:  93.8%
Global Step:  35900, Training Batch Accuracy:  89.1%
Global Step:  36000, Training Batch Accuracy: 100.0%
Saved checkpoint.
Global Step:  36100, Training Batch Accuracy:  90.6%
Global Step:  36200, Training Batch Accuracy:  90.6%
Global Step:  36300, Training Batch Accuracy:  93.8%
Global Step:  36400, Training Batch Accuracy:  89.1%
Global Step:  36500, Training Batch Accuracy:  93.8%
Global Step:  36600, Training Batch Accuracy:  92.2%
Global Step:  36700, Training Batch Accuracy:  90.6%
Global Step:  36800, Training Batch Accuracy:  85.9%
Global Step:  36900, Training Batch Accuracy:  95.3%
Global Step:  37000, Training Batch Accuracy:  93.8%
Saved checkpoint.
Global Step:  37100, Training Batch Accuracy:  92.2%
Global Step:  37200, Training Batch Accuracy:  95.3%
Global Step:  37300, Training Batch Accuracy:  92.2%
Global Step:  37400, Training Batch Accuracy:  90.6%
Global Step:  37500, Training Batch Accuracy:  93.8%
Global Step:  37600, Training Batch Accuracy:  93.8%
Global Step:  37700, Training Batch Accuracy:  93.8%
Global Step:  37800, Training Batch Accuracy:  92.2%
Global Step:  37900, Training Batch Accuracy:  92.2%
Global Step:  38000, Training Batch Accuracy:  84.4%
Saved checkpoint.
Global Step:  38100, Training Batch Accuracy:  95.3%
Global Step:  38200, Training Batch Accuracy:  92.2%
Global Step:  38300, Training Batch Accuracy:  92.2%
Global Step:  38400, Training Batch Accuracy:  93.8%
Global Step:  38500, Training Batch Accuracy:  95.3%
Global Step:  38600, Training Batch Accuracy:  92.2%
Global Step:  38700, Training Batch Accuracy:  93.8%
Global Step:  38800, Training Batch Accuracy:  95.3%
Global Step:  38900, Training Batch Accuracy:  90.6%
```

```
Global Step:  39000, Training Batch Accuracy:  92.2%
Saved checkpoint.
Global Step:  39100, Training Batch Accuracy:  93.8%
Global Step:  39200, Training Batch Accuracy:  95.3%
Global Step:  39300, Training Batch Accuracy:  92.2%
Global Step:  39400, Training Batch Accuracy:  93.8%
Global Step:  39500, Training Batch Accuracy:  92.2%
Global Step:  39600, Training Batch Accuracy:  95.3%
Global Step:  39700, Training Batch Accuracy:  93.8%
Global Step:  39800, Training Batch Accuracy:  90.6%
Global Step:  39900, Training Batch Accuracy:  90.6%
Global Step:  40000, Training Batch Accuracy:  90.6%
Saved checkpoint.
Global Step:  40100, Training Batch Accuracy:  90.6%
Global Step:  40200, Training Batch Accuracy:  90.6%
Global Step:  40300, Training Batch Accuracy:  96.9%
Global Step:  40400, Training Batch Accuracy:  92.2%
Global Step:  40500, Training Batch Accuracy:  90.6%
Global Step:  40600, Training Batch Accuracy:  92.2%
Global Step:  40700, Training Batch Accuracy:  89.1%
Global Step:  40800, Training Batch Accuracy:  93.8%
Global Step:  40900, Training Batch Accuracy:  90.6%
Global Step:  41000, Training Batch Accuracy:  92.2%
Saved checkpoint.
Global Step:  41100, Training Batch Accuracy:  96.9%
Global Step:  41200, Training Batch Accuracy:  93.8%
Global Step:  41300, Training Batch Accuracy:  93.8%
Global Step:  41400, Training Batch Accuracy:  84.4%
Global Step:  41500, Training Batch Accuracy:  89.1%
Global Step:  41600, Training Batch Accuracy:  98.4%
Global Step:  41700, Training Batch Accuracy:  93.8%
Global Step:  41800, Training Batch Accuracy:  92.2%
Global Step:  41900, Training Batch Accuracy:  93.8%
Global Step:  42000, Training Batch Accuracy: 100.0%
Saved checkpoint.
Global Step:  42100, Training Batch Accuracy:  89.1%
Global Step:  42200, Training Batch Accuracy:  95.3%
Global Step:  42300, Training Batch Accuracy:  89.1%
Global Step:  42400, Training Batch Accuracy:  90.6%
Global Step:  42500, Training Batch Accuracy:  85.9%
Global Step:  42600, Training Batch Accuracy:  90.6%
Global Step:  42700, Training Batch Accuracy:  95.3%
Global Step:  42800, Training Batch Accuracy:  96.9%
Global Step:  42900, Training Batch Accuracy:  93.8%
Global Step:  43000, Training Batch Accuracy:  96.9%
Saved checkpoint.
Global Step:  43100, Training Batch Accuracy:  92.2%
Global Step:  43200, Training Batch Accuracy:  92.2%
Global Step:  43300, Training Batch Accuracy:  87.5%
Global Step:  43400, Training Batch Accuracy:  89.1%
Global Step:  43500, Training Batch Accuracy:  93.8%
Global Step:  43600, Training Batch Accuracy:  96.9%
Global Step:  43700, Training Batch Accuracy:  90.6%
Global Step:  43800, Training Batch Accuracy:  93.8%
Global Step:  43900, Training Batch Accuracy:  96.9%
Global Step:  44000, Training Batch Accuracy:  93.8%
Saved checkpoint.
Global Step:  44100, Training Batch Accuracy:  93.8%
Global Step:  44200, Training Batch Accuracy:  92.2%
Global Step:  44300, Training Batch Accuracy:  90.6%
Global Step:  44400, Training Batch Accuracy:  89.1%
```

```
Global Step:  44500, Training Batch Accuracy:  95.3%
Global Step:  44600, Training Batch Accuracy:  96.9%
Global Step:  44700, Training Batch Accuracy:  92.2%
Global Step:  44800, Training Batch Accuracy:  93.8%
Global Step:  44900, Training Batch Accuracy:  96.9%
Global Step:  45000, Training Batch Accuracy:  92.2%
Saved checkpoint.
Global Step:  45100, Training Batch Accuracy:  96.9%
Global Step:  45200, Training Batch Accuracy:  90.6%
Global Step:  45300, Training Batch Accuracy:  95.3%
Global Step:  45400, Training Batch Accuracy:  96.9%
Global Step:  45500, Training Batch Accuracy:  95.3%
Global Step:  45600, Training Batch Accuracy:  98.4%
Global Step:  45700, Training Batch Accuracy:  95.3%
Global Step:  45800, Training Batch Accuracy:  92.2%
Global Step:  45900, Training Batch Accuracy:  98.4%
Global Step:  46000, Training Batch Accuracy:  92.2%
Saved checkpoint.
Global Step:  46100, Training Batch Accuracy:  90.6%
Global Step:  46200, Training Batch Accuracy:  90.6%
Global Step:  46300, Training Batch Accuracy:  98.4%
Global Step:  46400, Training Batch Accuracy:  87.5%
Global Step:  46500, Training Batch Accuracy:  90.6%
Global Step:  46600, Training Batch Accuracy:  93.8%
Global Step:  46700, Training Batch Accuracy:  93.8%
Global Step:  46800, Training Batch Accuracy:  90.6%
Global Step:  46900, Training Batch Accuracy:  89.1%
Global Step:  47000, Training Batch Accuracy:  89.1%
Saved checkpoint.
Global Step:  47100, Training Batch Accuracy:  90.6%
Global Step:  47200, Training Batch Accuracy:  95.3%
Global Step:  47300, Training Batch Accuracy:  96.9%
Global Step:  47400, Training Batch Accuracy:  92.2%
Global Step:  47500, Training Batch Accuracy:  92.2%
Global Step:  47600, Training Batch Accuracy:  92.2%
Global Step:  47700, Training Batch Accuracy:  92.2%
Global Step:  47800, Training Batch Accuracy:  89.1%
Global Step:  47900, Training Batch Accuracy:  95.3%
Global Step:  48000, Training Batch Accuracy:  95.3%
Saved checkpoint.
Global Step:  48100, Training Batch Accuracy:  92.2%
Global Step:  48200, Training Batch Accuracy:  96.9%
Global Step:  48300, Training Batch Accuracy:  95.3%
Global Step:  48400, Training Batch Accuracy:  96.9%
Global Step:  48500, Training Batch Accuracy:  95.3%
Global Step:  48600, Training Batch Accuracy:  90.6%
Global Step:  48700, Training Batch Accuracy:  98.4%
Global Step:  48800, Training Batch Accuracy:  95.3%
Global Step:  48900, Training Batch Accuracy:  98.4%
Global Step:  49000, Training Batch Accuracy:  92.2%
Saved checkpoint.
Global Step:  49100, Training Batch Accuracy:  92.2%
Global Step:  49200, Training Batch Accuracy:  90.6%
Global Step:  49300, Training Batch Accuracy:  92.2%
Global Step:  49400, Training Batch Accuracy:  92.2%
Global Step:  49500, Training Batch Accuracy:  93.8%
Global Step:  49600, Training Batch Accuracy:  95.3%
Global Step:  49700, Training Batch Accuracy:  98.4%
Global Step:  49800, Training Batch Accuracy:  95.3%
Global Step:  49900, Training Batch Accuracy:  95.3%
Global Step:  50000, Training Batch Accuracy:  93.8%
```

```
Saved checkpoint.
Time usage: 5:39:58
```

Results

After 150,000 optimization iterations, the classification accuracy is about 79-80% on the test-set. Examples of mis-classifications are plotted below. Some of these are difficult to recognize even for humans and others are reasonable mistakes e.g. between a large car and a truck, or between a cat and a dog, while other mistakes seem a bit strange.

In [57]:

```
print_test_accuracy(show_example_errors=True,
                    show_confusion_matrix=True)
```

```
LALALA2 [0 1 0 ..., 1 1 0] [0 0 0 ..., 1 1 1] 2000 2000 [ True False
  True ...,  True  True False]
Accuracy on Test-Set: 82.2% (1644 / 2000)
Example errors:
```


```
Confusion Matrix:
[842 158] (0) not-cat
[198 802] (1) cat
 (0) (1)
```

Convolutional Weights

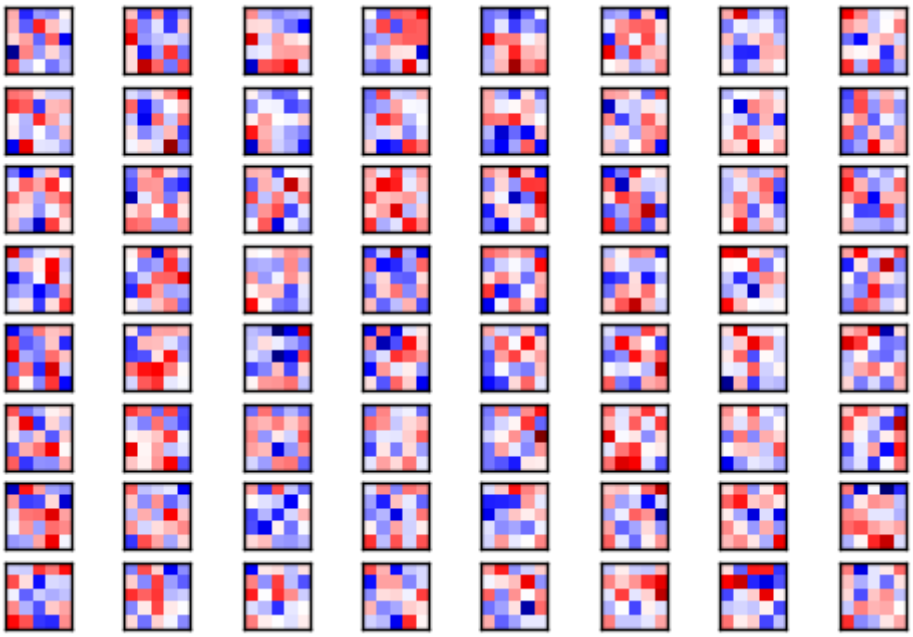
The following shows some of the weights (or filters) for the first convolutional layer. There are 3 input channels so there are 3 of these sets, which you may plot by changing the `input_channel`.

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

In [60]:

```
plot_conv_weights(weights=weights_conv1, input_channel=0)
```

```
Min:  -0.53483, Max:   0.58935
Mean: -0.00163, Stdev: 0.16052
```

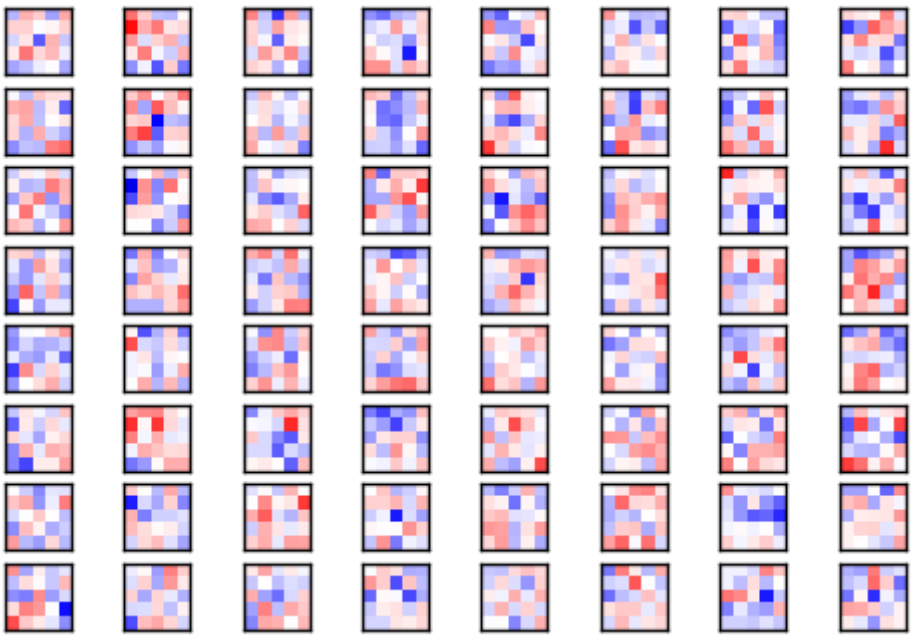


Plot some of the weights (or filters) for the second convolutional layer. These are apparently closer to zero than the weights for the first convolutional layers, see the lower standard deviation.

In [61]:

```
plot_conv_weights(weights=weights_conv2, input_channel=0)
```

```
Min:  -0.24218, Max:   0.18615
Mean: -0.00361, Stdev: 0.04329
```



Output of convolutional layers

Helper-function for plotting an image.

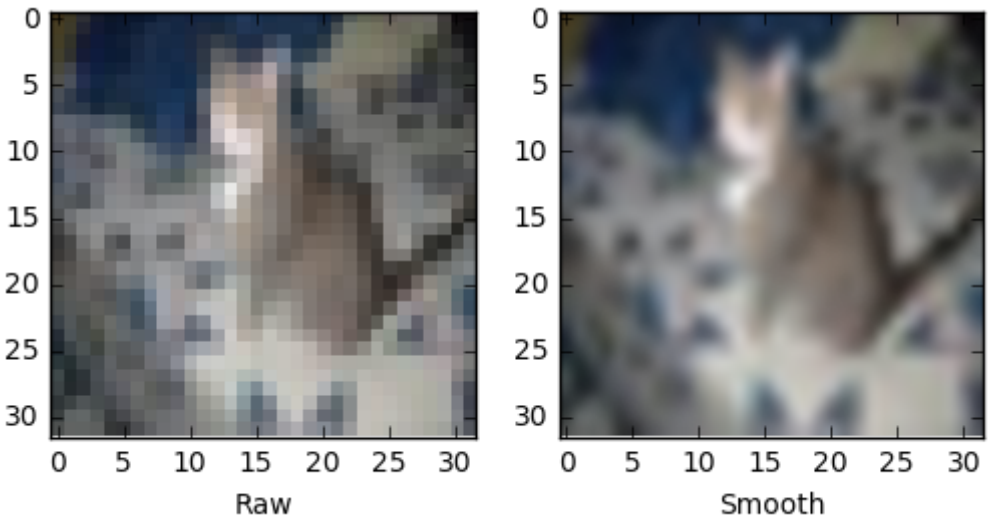
In [62]:

```
def plot_image(image):
    # Create figure with sub-plots.
    fig, axes = plt.subplots(1, 2)

    # References to the sub-plots.
    ax0 = axes.flat[0]
    ax1 = axes.flat[1]

    # Show raw and smoothened images in sub-plots.
    ax0.imshow(image, interpolation='nearest')
    ax1.imshow(image, interpolation='spline16')

    # Set labels.
    ax0.set_xlabel('Raw')
    ax1.set_xlabel('Smooth')

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

Plot an image from the test-set. The raw pixelated image is used as input to the neural network.

In [63]:

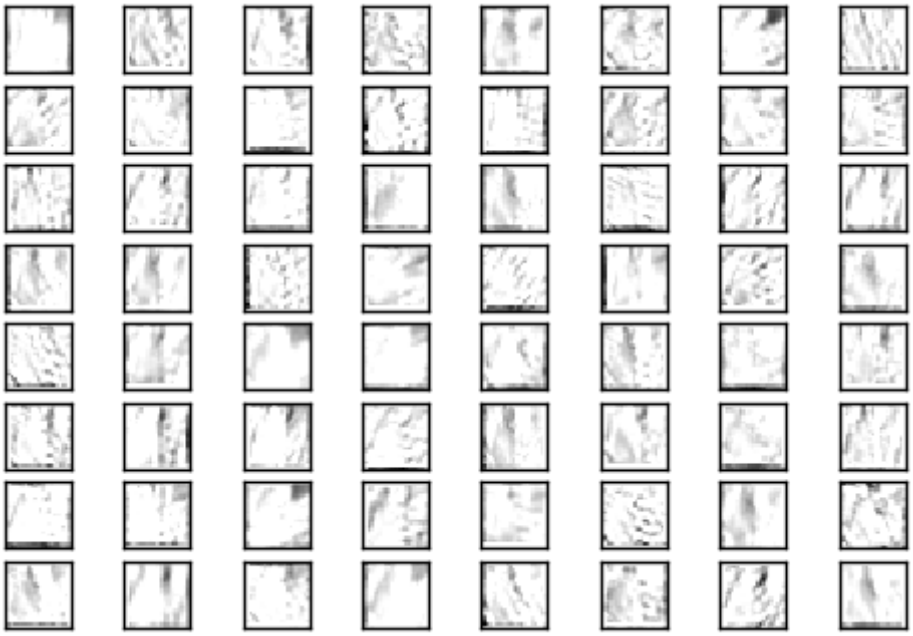
```
img, cls = get_test_image(16)
plot_image(img)
```



Use the raw image as input to the neural network and plot the output of the first convolutional layer.

In [64]:

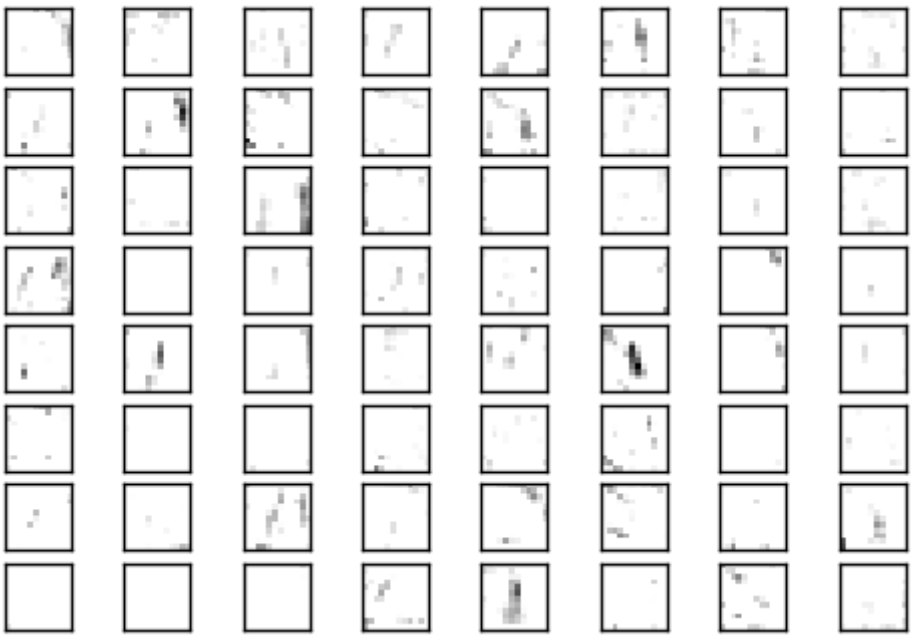
```
plot_layer_output(output_conv1, image=img)
```



Using the same image as input to the neural network, now plot the output of the second convolutional layer.

In [65]:

```
plot_layer_output(output_conv2, image=img)
```



Predicted class-labels

Get the predicted class-label and class-number for this image.

In [66]:

```
label_pred, cls_pred = session.run([y_pred, y_pred_cls],
                                   feed_dict={x: [img]})
```

Print the predicted class-label.

In [67]:

```
# Set the rounding options for numpy.
np.set_printoptions(precision=3, suppress=True)

# Print the predicted label.
print(label_pred[0])
```

```
[ 0.001  0.999]
```

The predicted class-label is an array of length 10, with each element indicating how confident the neural network is that the image is the given class.

In this case the element with index 3 has a value of 0.493, while the element with index 5 has a value of 0.490. This means the neural network believes the image either shows a class 3 or class 5, which is a cat or a dog, respectively.

In [68]:

```
class_names[1]
```

Out[68]:

```
'cat'
```

In [69]:

```
class_names[0]
```

Out[69]:

```
'not-cat'
```

Close TensorFlow Session

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

In [70]:

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

Conclusion

This tutorial showed how to make a Convolutional Neural Network for classifying images in the CIFAR-10 data-set. The classification accuracy was about 79-80% on the test-set.

The output of the convolutional layers was also plotted, but it was difficult to see how the neural network recognizes and classifies the input images. Better visualization techniques are needed.

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.

- Run the optimization for 10,000 iterations and see what the classification accuracy is. This will create a checkpoint that saves all the variables of the TensorFlow graph.
- Continue running the optimization for another 100,000 iterations and see if the classification accuracy has improved. Then try another 100,000 iterations. Does the accuracy improve and do you think it is worth the extra computational time?
- Try changing the image distortions in the pre-processing.
- Try changing the structure of the neural network. You can try making the neural network both smaller or bigger. How does it affect the training time and the classification accuracy? Note that the checkpoints cannot be reloaded when you change the structure of the neural network.
- Try using batch-normalization for the 2nd convolutional layer as well. Also try removing it from both layers.
- Research some of the better neural networks (http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html) for CIFAR-10 and try to implement them.
- Explain to a friend how the program works.

License (MIT)