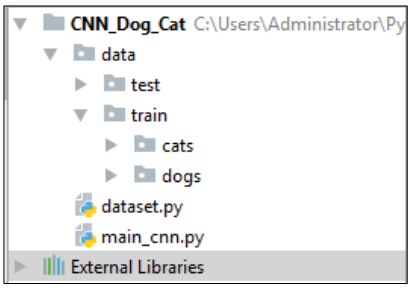
Dog and Cat Classification using CNN

Create a directory hierarchy same as follow



Test folder contains test image, while train folder contains subfolder cats which containing cat images and dogs folder containing dog images for training.

A. Datasets, save the following code to *dataset.py*

1. Importing some packages

```
import os
import glob
import numpy as np
import cv2
from sklearn.utils import shuffle
```

2. Load training data

```
def load_train(train_path, image_size, classes):
    images = []
    labels = []
    ids = []
    cls = []

    print('Reading training images')
    for fld in classes:   # assuming data directory has a separate folder for each class, and
that each folder is named after the class
        index = classes.index(fld)
        print('Loading {} files (Index: {})'.format(fld, index))
        path = os.path.join(train_path, fld, '*g')
        files = glob.glob(path)
        for fl in files:
            image = cv2.imread(fl)
            image = cv2.resize(image, (image_size, image_size), cv2.INTER_LINEAR)
            images.append(image)
            label = np.zeros(len(classes))
            label[index] = 1.0
            labels.append(label)
            flbase = os.path.basename(fl)
            ids.append(flbase)
```

```
            cls.append(fld)
    images = np.array(images)
    labels = np.array(labels)
    ids = np.array(ids)
    cls = np.array(cls)

    return images, labels, ids, cls
```

3. Load testing data

```
def load_test(test_path, image_size):
  path = os.path.join(test_path, '*g')
  files = sorted(glob.glob(path))

  X_test = []
  X_test_id = []
  print("Reading test images")
  for fl in files:
      flbase = os.path.basename(fl)
      img = cv2.imread(fl)
      img = cv2.resize(img, (image_size, image_size), cv2.INTER_LINEAR)
      X_test.append(img)
      X_test_id.append(flbase)

  ### because we're not creating a DataSet object for the test images, normalization happens
here
  X_test = np.array(X_test, dtype=np.uint8)
  X_test = X_test.astype('float32')
  X_test = X_test / 255

  return X_test, X_test_id
```

4. Class data set

```
class DataSet(object):

  def __init__(self, images, labels, ids, cls):
    """Construct a DataSet. one_hot arg is used only if fake_data is true."""

    self._num_examples = images.shape[0]


    # Convert shape from [num examples, rows, columns, depth]
    # to [num examples, rows*columns] (assuming depth == 1)
    # Convert from [0, 255] -> [0.0, 1.0].

    images = images.astype(np.float32)
    images = np.multiply(images, 1.0 / 255.0)

    self._images = images
    self._labels = labels
    self._ids = ids
    self._cls = cls
    self._epochs_completed = 0
```

```
    self._index_in_epoch = 0

  @property
  def images(self):
    return self._images

  @property
  def labels(self):
    return self._labels

  @property
  def ids(self):
    return self._ids

  @property
  def cls(self):
    return self._cls

  @property
  def num_examples(self):
    return self._num_examples

  @property
  def epochs_completed(self):
    return self._epochs_completed
```

```
def next_batch(self, batch_size):
  """Return the next `batch_size` examples from this data set."""
  start = self._index_in_epoch
  self._index_in_epoch += batch_size

  if self._index_in_epoch > self._num_examples:
    # Finished epoch
    self._epochs_completed += 1

    # # Shuffle the data (maybe)
    # perm = np.arange(self._num_examples)
    # np.random.shuffle(perm)
    # self._images = self._images[perm]
    # self._labels = self._labels[perm]
    # Start next epoch

    start = 0
    self._index_in_epoch = batch_size
    assert batch_size <= self._num_examples
  end = self._index_in_epoch

  return self._images[start:end], self._labels[start:end], self._ids[start:end],
self._cls[start:end]
```

5. Read train dataset

```
def read_train_sets(train_path, image_size, classes, validation_size=0):
  class DataSets(object):
```

```
    pass
  data_sets = DataSets()

  images, labels, ids, cls = load_train(train_path, image_size, classes)
  images, labels, ids, cls = shuffle(images, labels, ids, cls)  # shuffle the data

  if isinstance(validation_size, float):
    validation_size = int(validation_size * images.shape[0])

  validation_images = images[:validation_size]
  validation_labels = labels[:validation_size]
  validation_ids = ids[:validation_size]
  validation_cls = cls[:validation_size]

  train_images = images[validation_size:]
  train_labels = labels[validation_size:]
  train_ids = ids[validation_size:]
  train_cls = cls[validation_size:]

  data_sets.train = DataSet(train_images, train_labels, train_ids, train_cls)
  data_sets.valid = DataSet(validation_images, validation_labels, validation_ids,
validation_cls)

  return data_sets
```

6. Read test dataset

```
def read_test_set(test_path, image_size):
  images, ids  = load_test(test_path, image_size)
  return images, ids
```

B. Main class, save the following codes to *main_cnn.py*

1. Importing some packages

```
import time
import math
import random

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import dataset
import cv2

from sklearn.metrics import confusion_matrix
from datetime import timedelta
```

2. Configuration hyper parameters

```
# Convolutional Layer 1.
filter_size1 = 3
num_filters1 = 32

# Convolutional Layer 2.
filter_size2 = 3
num_filters2 = 32

# Convolutional Layer 3.
filter_size3 = 3
num_filters3 = 64

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

# Number of color channels for the images: 1 channel for gray-scale.
num_channels = 3

# image dimensions (only squares for now)
img_size = 128

# Size of image when flattened to a single dimension
img_size_flat = img_size * img_size * num_channels

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

# class info
classes = ['dogs', 'cats']
num_classes = len(classes)

# batch size
batch_size = 16

# validation split
validation_size = .16

# how long to wait after validation loss stops improving before terminating training
early_stopping = None  # use None if you don't want to implement early stoping

train_path = 'data/train/'
test_path = 'data/test/'
checkpoint_dir = "models/"
```

3. Load data and print data

```
data = dataset.read_train_sets(train_path, img_size, classes, validation_size=validation_size)
test_images, test_ids = dataset.read_test_set(test_path, img_size)

print("Size of:")
print("- Training-set:WtWt{}".format(len(data.train.labels)))
print("- Test-set:WtWt{}".format(len(test_images)))
print("- Validation-set:Wt{}".format(len(data.valid.labels)))
```

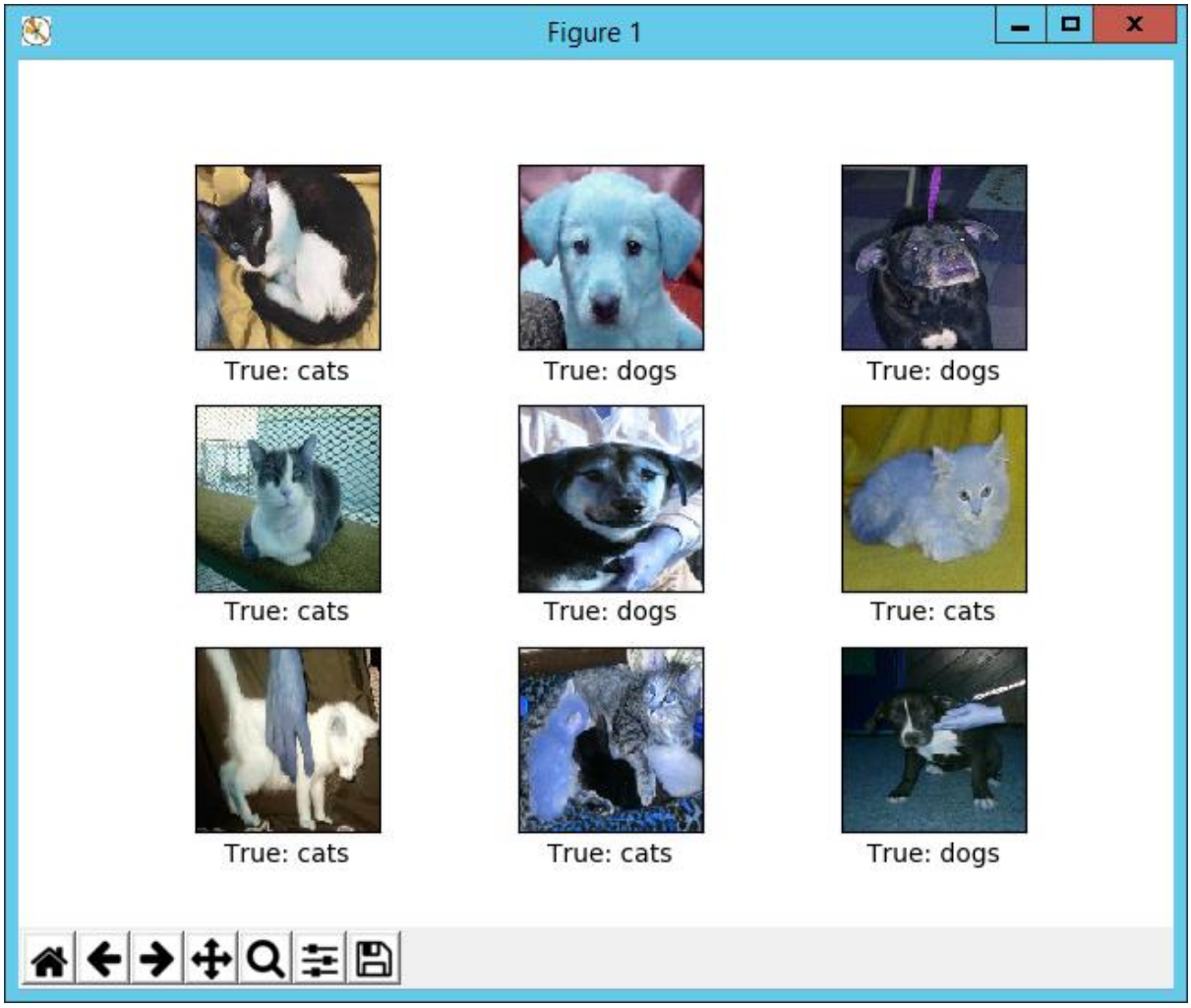
4. Function for plotting image

```
def plot_images(images, cls_true, cls_pred=None):
    if len(images) == 0:
        print("no images to show")
        return
    else:
        random_indices = random.sample(range(len(images)), min(len(images), 9))

    images, cls_true = zip(*[(images[i], cls_true[i]) for i in random_indices])

    # 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_size, img_size, num_channels))

        # 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()
```

5. Plot a few images to see if data is correct or not

```
# Get some random images and their labels from the train set.

images, cls_true  = data.train.images, data.train.cls

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

Output



6. Function for creating new variables weights and bias

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

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

7. Function for creating a new convolutional layer

```
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
```

8. Function flattened layer

```
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
```

9. Function for fully connected layer

```
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
```

10. Place holder variables

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

x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])

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

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

11. Create Convolution layer 1

```
layer_conv1, weights_conv1 = W
    new_conv_layer(input=x_image,
```

```
                    num_input_channels=num_channels,
                    filter_size=filter_size1,
                    num_filters=num_filters1,
                    use_pooling=True)
```

12. Create Convolution layer 2

```
layer_conv2, weights_conv2 = W
    new_conv_layer(input=layer_conv1,
                    num_input_channels=num_filters1,
                    filter_size=filter_size2,
                    num_filters=num_filters2,
                    use_pooling=True)
```

13. Create Convolution layer 3

```
layer_conv3, weights_conv3 = W
    new_conv_layer(input=layer_conv2,
                    num_input_channels=num_filters2,
                    filter_size=filter_size3,
                    num_filters=num_filters3,
                    use_pooling=True)
```

14. Create flattened layer

```
layer_flat, num_features = flatten_layer(layer_conv3)
```

15. Create fully connected layer 1

```
layer_fc1 = new_fc_layer(input=layer_flat,
                         num_inputs=num_features,
                         num_outputs=fc_size,
                         use_relu=True)
```

16. Create fully connected layer 2

```
layer_fc2 = new_fc_layer(input=layer_fc1,
                         num_inputs=fc_size,
                         num_outputs=num_classes,
                         use_relu=False)
```

17. Predicted class

```
y_pred = tf.nn.softmax(layer_fc2)

y_pred_cls = tf.argmax(y_pred, dimension=1) #the class number is the index of largest element
```

18. Cost function to be optimized

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=layer_fc2,
                                                        labels=y_true)
cost = tf.reduce_mean(cross_entropy)
```

19. Optimization method

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost)

correct_prediction = tf.equal(y_pred_cls, y_true_cls)

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

20. Tensorflow Run

```
session = tf.Session()

session.run(tf.initialize_all_variables())

train_batch_size = batch_size
```

21. Function to perform optimization iterations

```
def print_progress(epoch, feed_dict_train, feed_dict_validate, val_loss):
    # Calculate the accuracy on the training-set.
    acc = session.run(accuracy, feed_dict=feed_dict_train)
    val_acc = session.run(accuracy, feed_dict=feed_dict_validate)
    msg = "Epoch {0} --- Training Accuracy: {1:>6.1%}, Validation Accuracy: {2:>6.1%}, Validation
Loss: {3:.3f}"
    print(msg.format(epoch + 1, acc, val_acc, val_loss))
```

22. 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()

    best_val_loss = float("inf")
    patience = 0

    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, _, cls_batch = data.train.next_batch(train_batch_size)
        x_valid_batch, y_valid_batch, _, valid_cls_batch =
data.valid.next_batch(train_batch_size)

        # Convert shape from [num examples, rows, columns, depth]
        # to [num examples, flattened image shape]

        x_batch = x_batch.reshape(train_batch_size, img_size_flat)
        x_valid_batch = x_valid_batch.reshape(train_batch_size, img_size_flat)

        # 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}

        feed_dict_validate = {x: x_valid_batch,
                              y_true: y_valid_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 at end of each epoch (defined as full pass through training dataset).
        if i % int(data.train.num_examples / batch_size) == 0:
            val_loss = session.run(cost, feed_dict=feed_dict_validate)
            epoch = int(i / int(data.train.num_examples / batch_size))

            print_progress(epoch, feed_dict_train, feed_dict_validate, val_loss)

            if early_stopping:
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    patience = 0
                else:
                    patience += 1

                if patience == early_stopping:
                    break

    # 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 elapsed: " + str(timedelta(seconds=int(round(time_dif)))))
```

23. Function to plot example errors

```
def plot_example_errors(cls_pred, correct):
    # 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.valid.images[incorrect]

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

    # Get the true classes for those images.
    cls_true = data.valid.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])
```

24. Function to plot confusion matrix

```
def plot_confusion_matrix(cls_pred):
    # 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.valid.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()
```

25. Function for showing the performance

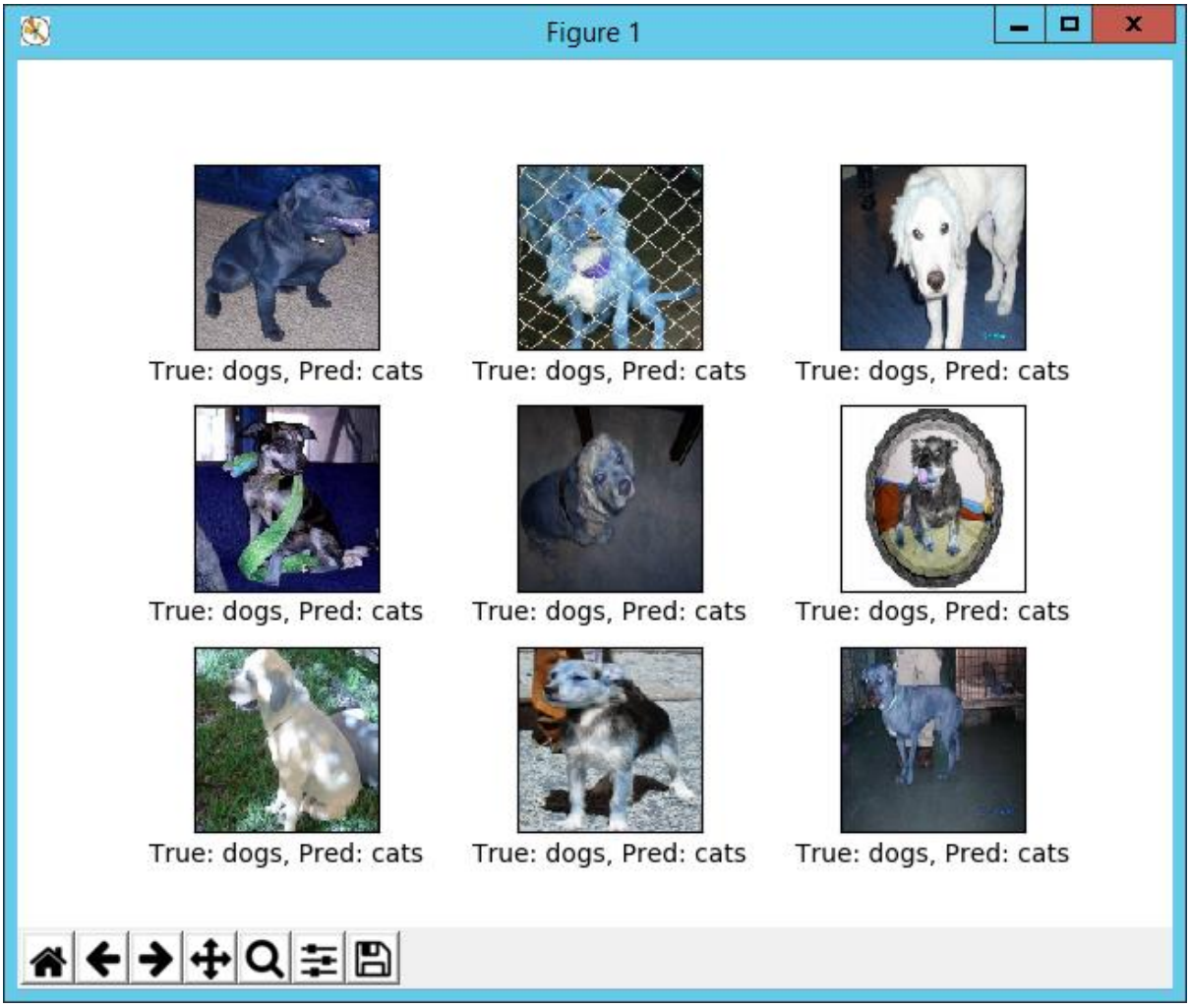
```
def print_validation_accuracy(show_example_errors=False,
                              show_confusion_matrix=False):
    # Number of images in the test-set.
    num_test = len(data.valid.images)

    # 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 + batch_size, num_test)

        # Get the images from the test-set between index i and j.
        images = data.valid.images[i:j, :].reshape(batch_size, img_size_flat)

        # Get the associated labels.
        labels = data.valid.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
```

```
    cls_true = np.array(data.valid.cls)
    cls_pred = np.array([classes[x] for x in cls_pred])

    # 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)
```

26. Performance after 1 optimization iteration

```
optimize(num_iterations=1)
print_validation_accuracy()
```

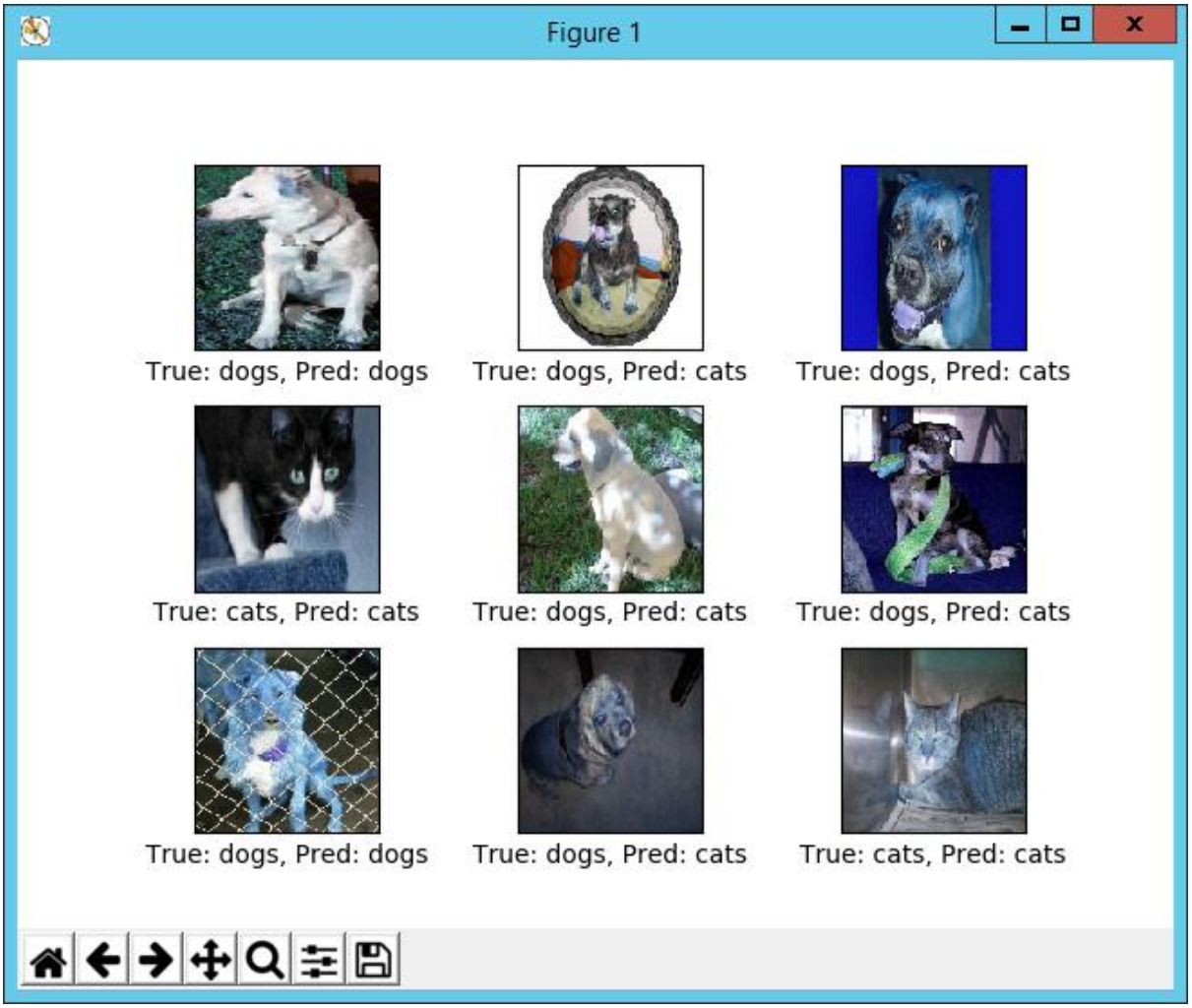
Output



27. Performance after 100 optimization iterations

```
optimize(num_iterations=100)  # We already performed 1 iteration above.

print_validation_accuracy(show_example_errors=True)
```

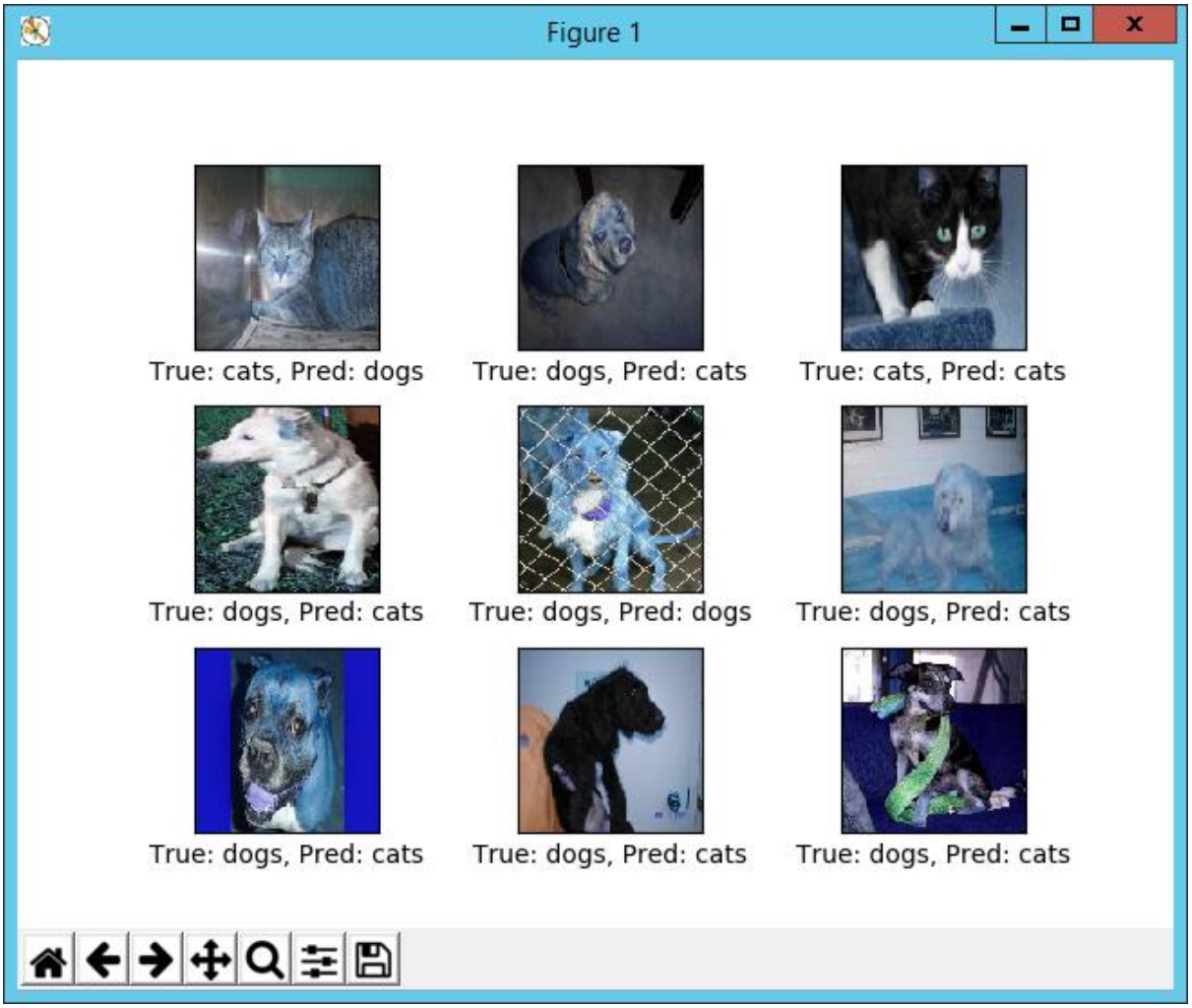
Output



28. Performance after 500 optimization iterations

```
optimize(num_iterations=500)  # We already performed 1 iteration above.

print_validation_accuracy(show_example_errors=True)
```

Output



29. Performance after 1000 optimization iterations

```
optimize(num_iterations=1000) # We performed 1000 iterations above.

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

30. Test on sample image

```
plt.axis('off')

test_cat = cv2.imread('data/test/1.jpg')
test_cat = cv2.resize(test_cat, (img_size, img_size), cv2.INTER_LINEAR) / 255

preview_cat = plt.imshow(test_cat.reshape(img_size, img_size, num_channels))

test_dog = cv2.imread('data/test/12500.jpg')
test_dog = cv2.resize(test_dog, (img_size, img_size), cv2.INTER_LINEAR) / 255
```

```
preview_dog = plt.imshow(test_dog.reshape(img_size, img_size, num_channels))


def sample_prediction(test_im):
    feed_dict_test = {
        x: test_im.reshape(1, img_size_flat),
        y_true: np.array([[1, 0]])
    }

    test_pred = session.run(y_pred_cls, feed_dict=feed_dict_test)
    return classes[test_pred[0]]


print("Predicted class for test_cat: {}".format(sample_prediction(test_cat)))
print("Predicted class for test_dog: {}".format(sample_prediction(test_dog)))
```

31. Function for plotting convolution weight

```
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()
```

32. Function for plot convolutional layer

```
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.

    image = image.reshape(img_size_flat)

    # 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()
```

33. Test the image

```
def plot_image(image):
    plt.imshow(image.reshape(img_size, img_size, num_channels),
               interpolation='nearest')
    plt.show()
```

```
image1 = test_images[0]
plot_image(image1)

image2 = test_images[9]
plot_image(image2)

plot_conv_weights(weights=weights_conv1)

plot_conv_layer(layer=layer_conv1, image=image1)

plot_conv_layer(layer=layer_conv1, image=image2)

plot_conv_weights(weights=weights_conv2, input_channel=0)

plot_conv_weights(weights=weights_conv2, input_channel=1)

plot_conv_layer(layer=layer_conv2, image=image1)

plot_conv_layer(layer=layer_conv2, image=image2)
```

34. Predict all the test data and write the result in csv

```
def write_predictions(ims, ids):
    ims = ims.reshape(ims.shape[0], img_size_flat)
    preds = session.run(y_pred, feed_dict={x: ims})
    result = pd.DataFrame(preds, columns=classes)
    result.loc[:, 'id'] = pd.Series(ids, index=result.index)
    pred_file = 'predictions.csv'
    result.to_csv(pred_file, index=False)


write_predictions(test_images, test_ids)
```

Console ouput

```
Reading training images
Loading dogs files (Index: 0)
Loading cats files (Index: 1)
Reading test images
Size of:
- Training-set:     3360
- Test-set:     20
- Validation-set:   640

Epoch 1 --- Training Accuracy:  56.2%, Validation Accuracy:  37.5%, Validation Loss: 0.700
Time elapsed: 0:00:09
Accuracy on Test-Set: 52.0% (333 / 640)
Time elapsed: 0:00:02
Accuracy on Test-Set: 47.8% (306 / 640)
Example errors:
Epoch 2 --- Training Accuracy:  43.8%, Validation Accuracy:  37.5%, Validation Loss: 0.724
Epoch 3 --- Training Accuracy:  68.8%, Validation Accuracy:  68.8%, Validation Loss: 0.645
Epoch 4 --- Training Accuracy:  87.5%, Validation Accuracy:  68.8%, Validation Loss: 0.608
Time elapsed: 0:00:17
Accuracy on Test-Set: 65.3% (418 / 640)
Example errors:
Epoch 5 --- Training Accuracy:  93.8%, Validation Accuracy:  81.2%, Validation Loss: 0.534
Epoch 6 --- Training Accuracy:  93.8%, Validation Accuracy:  68.8%, Validation Loss: 0.591
Epoch 7 --- Training Accuracy:  93.8%, Validation Accuracy:  50.0%, Validation Loss: 0.680
Epoch 8 --- Training Accuracy:  93.8%, Validation Accuracy:  75.0%, Validation Loss: 0.612
Epoch 9 --- Training Accuracy:  93.8%, Validation Accuracy:  87.5%, Validation Loss: 0.415
Time elapsed: 0:00:25
Accuracy on Test-Set: 70.8% (453 / 640)
Example errors:
Confusion Matrix:
[[263  43]
 [144 190]]
Predicted class for test_cat: cats
Predicted class for test_dog: cats

Process finished with exit code 1
```