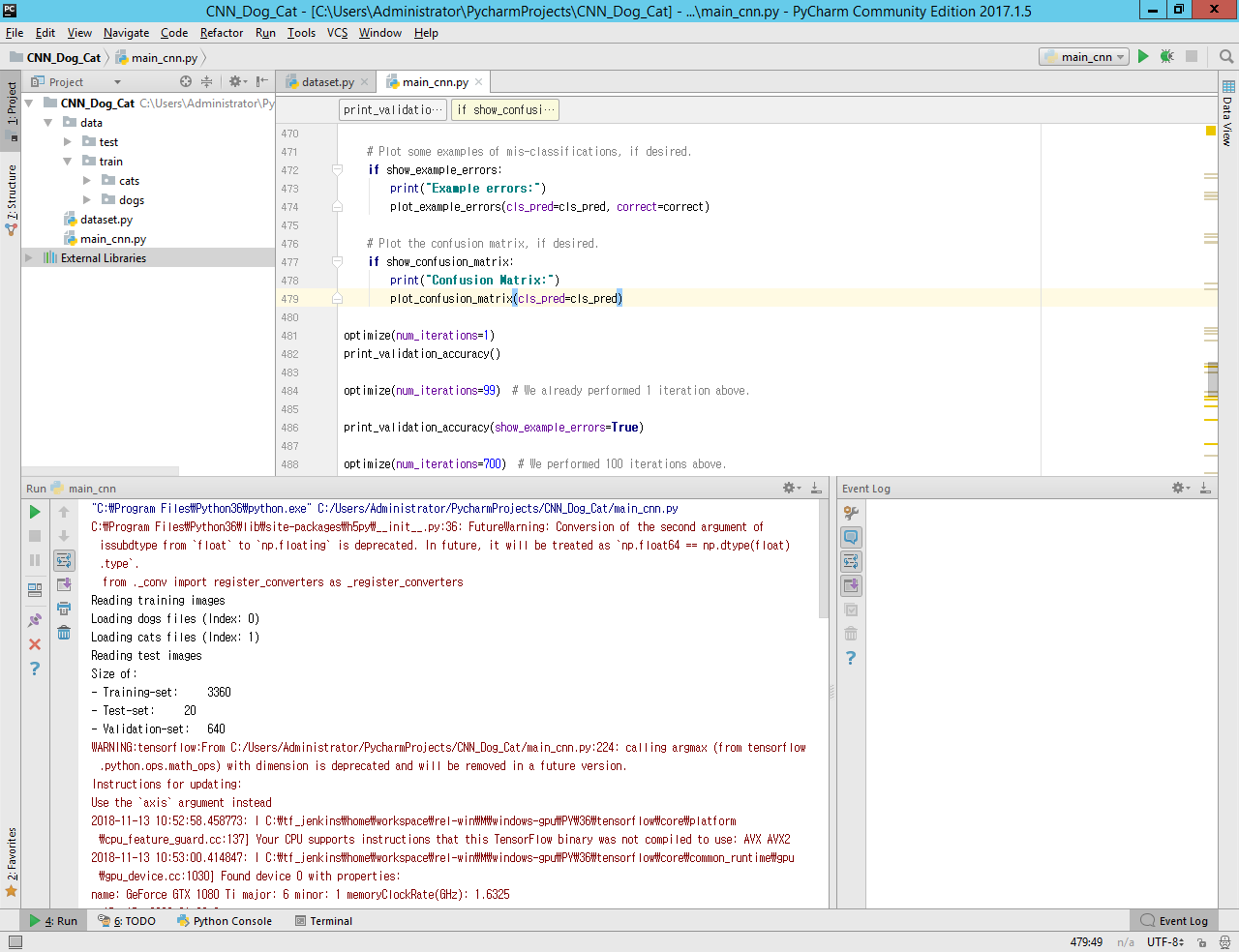
**Dog and Cat Classification using CNN**

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.

1. **Datasets, save the following code to *dataset.py***
2. Importing some packages

|  |
| --- |
| **import** os **import** glob **import** numpy **as** np **import** cv2 **from** sklearn.utils **import** shuffle |

1. 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 |

1. 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 |

1. 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] |

1. 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 |

1. Read test dataset

|  |
| --- |
| **def** read\_test\_set(test\_path, image\_size):  images, ids = load\_test(test\_path, image\_size)  **return** images, ids |

1. **Main class, save the following codes to *main\_cnn.py***
2. 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 |

1. 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/"** |

1. 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:\t\t{}"**.format(len(data.train.labels))) print(**"- Test-set:\t\t{}"**.format(len(test\_images))) print(**"- Validation-set:\t{}"**.format(len(data.valid.labels))) |

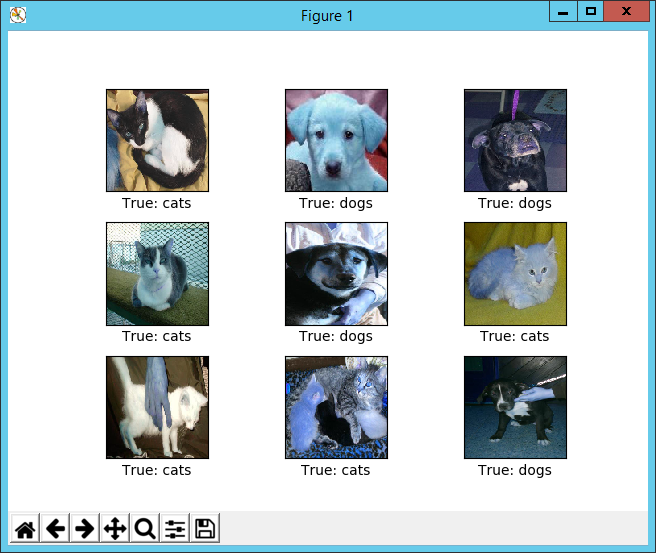
1. Function for plotting image

|  |
| --- |
| **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() |

1. 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



1. 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])) |

1. 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 |

1. 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 |

1. 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 |

1. 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) |

1. Create Convolution layer 1

|  |
| --- |
| layer\_conv1, weights\_conv1 = \  new\_conv\_layer(input=x\_image,  num\_input\_channels=num\_channels,  filter\_size=filter\_size1,  num\_filters=num\_filters1,  use\_pooling=**True**) |

1. Create Convolution layer 2

|  |
| --- |
| layer\_conv2, weights\_conv2 = \  new\_conv\_layer(input=layer\_conv1,  num\_input\_channels=num\_filters1,  filter\_size=filter\_size2,  num\_filters=num\_filters2,  use\_pooling=**True**) |

1. Create Convolution layer 3

|  |
| --- |
| layer\_conv3, weights\_conv3 = \  new\_conv\_layer(input=layer\_conv2,  num\_input\_channels=num\_filters2,  filter\_size=filter\_size3,  num\_filters=num\_filters3,  use\_pooling=**True**) |

1. Create flattened layer

|  |
| --- |
| layer\_flat, num\_features = flatten\_layer(layer\_conv3) |

1. Create fully connected layer 1

|  |
| --- |
| layer\_fc1 = new\_fc\_layer(input=layer\_flat,  num\_inputs=num\_features,  num\_outputs=fc\_size,  use\_relu=**True**) |

1. Create fully connected layer 2

|  |
| --- |
| layer\_fc2 = new\_fc\_layer(input=layer\_fc1,  num\_inputs=fc\_size,  num\_outputs=num\_classes,  use\_relu=**False**) |

1. 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 |

1. 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) |

1. 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)) |

1. Tensorflow Run

|  |
| --- |
| session = tf.Session()  session.run(tf.initialize\_all\_variables())  train\_batch\_size = batch\_size |

1. 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)) |

1. 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))))) |

1. 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]) |

1. 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() |

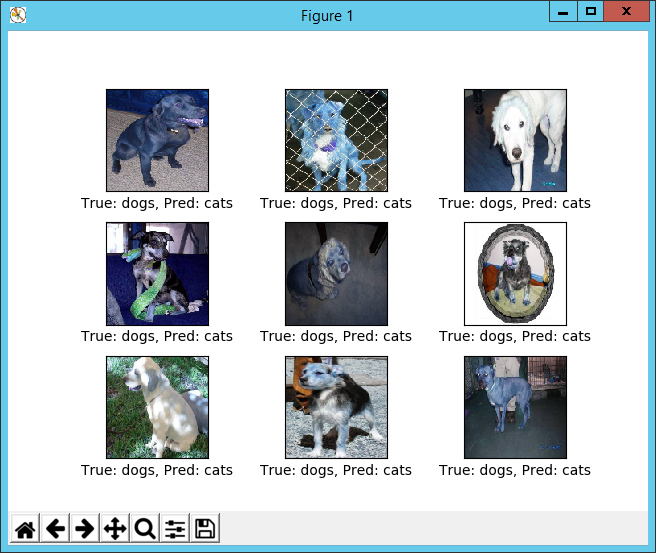
1. Function for showing the performance

|  |
| --- |
| **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) |

1. Performance after 1 optimization iteration

|  |
| --- |
| optimize(num\_iterations=1) print\_validation\_accuracy() |

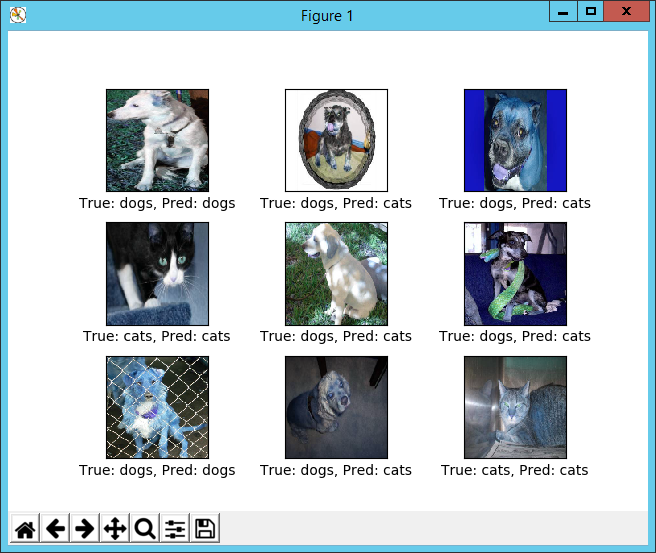
Output



1. Performance after 100 optimization iterations

|  |
| --- |
| optimize(num\_iterations=100) *# We already performed 1 iteration above.* print\_validation\_accuracy(show\_example\_errors=**True**) |

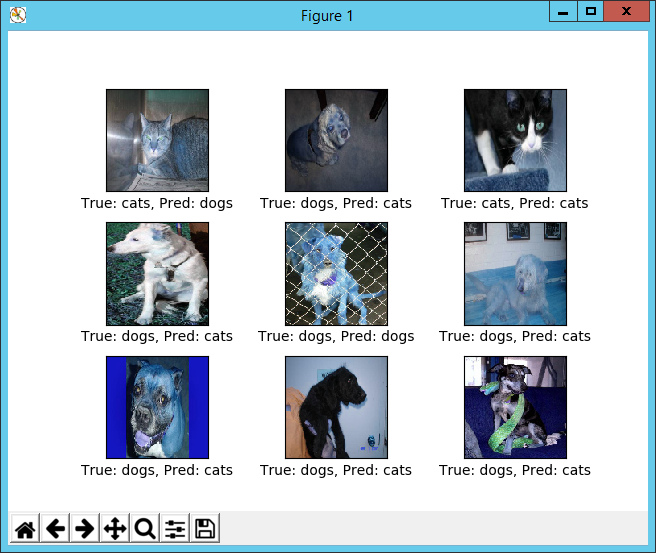
Output



1. Performance after 500 optimization iterations

|  |
| --- |
| optimize(num\_iterations=500) *# We already performed 1 iteration above.* print\_validation\_accuracy(show\_example\_errors=**True**) |

Output



1. 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**) |

1. 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))) |

1. 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() |

1. 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() |

1. 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) |

1. 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**

