CSC 470: Introduction to Neural Networks

Programming Project #3

Cats vs. Dogs: A Simple Dataset for Image Classification Using a Convolutional Neural Network

Background

Computer vision is a subdiscipline of artificial intelligence that studies and applies the use of machine learning to enable computers to analyze and make sense of image data. It is one of the most popular applications of neural networks, especially convolutional neural networks (CNNs), since CNNs are designed specifically for extracting and classifying features from images. Common problems solved by CNNs include object detection and identification, classification of images into categories, and facial recognition.

As we have seen from our brief introduction to CNNs, they tend to be computationally heavy, with their complexity increasing dramatically as the dimensions of the input images increase. One or more hidden convolutional layers are used to generate sets of feature maps from the image itself (for the first convolutional layer) or from the output of a previous layer. Each convolutional layer is followed by a subsampling layer that reduces the amount of information flowing forward through the network. One of the major principles of CNNs is this reduction in complexity as the data pass forward through the hidden layers. Between the convolutional/subsampling layer pairs and the output layer there are one or more fully connected layers that combine the extracted features in a non-linear way to enable the CNN to make predictions about the images.

Due to the heavy computations needed for large images, it is usually necessary to use systems with higher end graphics cards installed to enable the CNNs to operate in a practical length of time. The GPUs of high end graphics cards are much faster than the CPUs on a computer's motherboard, and the cards also tend to have higher memory capacities with faster memory. Very small images can still be used reasonably well on a standard desktop computer or laptop, however.

For this project, the goal will be to build a relatively simple CNN to classify images from a dataset as either "dog", if the image contains a dog, or "cat", if the image contains a cat. The training set contains 25,000 JPG images of varying dimensions. The images are all in color, so they are **3-channel** images to account for the standard RGB color scheme. As with the previous project, I will present step by step information about the building of the model. Your task will be to build this model, then try to tweak various aspects of it to try to improve its performance.

As always, please do not hesitate to email me if you run into any problems!

Installing Needed Third-Party Libraries

The third-party libraries you will need for this project are:

- `NumPy`
- `matplotlib`
- `keras`
- `piexif`

You should already have the first three (`NumPy`, `matplotlib`, and `keras`) installed since they were dependencies for Programming Project #2. The only additional dependency you need is `piexif`, which is a library for handling image metadata. The information for `piexif` is shown below.

piexif (library for handling image metadata)	Version: 1.1.3 URL: https://pypi.org/project/piexif/ API: https://piexif.readthedocs.io/en/latest/ Install command: `pip install piexif`

You may find that when you run the install command you will see a message stating the `piexif` library is already installed. If so, you're good to go.

Scripts Provided:

I am providing you with the following scripts:

`dogs_vs_cats_CNN.py`

This is the main script that calls the helper function to partition the dataset, then builds, trains and tests the CNN.

`dataset_utilities.py`

This script contains the train_test_split helper function that partitions the dataset into a training set and a testing set.

The Dataset

The dataset we will use for this project comes from an image classification contest that was hosted on Kaggle a number of years ago (the contest ended in 2014). Microsoft is now hosting a copy of the dataset, and you will need to use Microsoft's version of it for this project. To download it, visit the following URL (you do not need a login to access the dataset):

https://www.microsoft.com/en-us/download/details.aspx?id=54765

When you visit this page you should see a red "Download" button; use this button to download the dataset. It is a .zip file approximately 805 megabytes in size containing an archive of 25,000 images, 12,500 cat images and 12,500 dog images. The folder structure of the archive's contents should be as follows:

```
PetImages
      |__Cat
      |__Dog
```

Do not alter this folder structure, since the code I will be presenting expects the folder structure as it appears above.

Exploratory Analysis of the Dataset

In the previous project we performed several operations to analyze the dataset prior to using it, just so we could get an idea of what we were dealing with. An exploratory analysis is always advisable, since it can save a lot of potential headaches down the line. For this project the dataset consists of images, not numerical data. We can glean a lot of information about the data just from viewing it in a file explorer window. Some of the things we want to look out for with image data are the following:

1. **File formats.** We want all the images to have te same file format if possible. For this project all the images are JPG images.

2. **Image dimensions.** The image dimensions are important because the size of an image is directly relevant to how long it will take to train a CNN. Training time increases as the image size increases. We also want to look out for images that are very small. It may be too difficult to extract features from images that are very tiny, depending on what we are looking for, and the presence of very tiny images could throw off the performance of a CNN. One final important thing to note is that the images in a dataset are not necessarily required to have the same dimensions. In our case, there is a lot of variation in the image dimensions.

3. **Image quality.** The ability of a CNN to extract features from an image will also depend very heavily on the image quality. Ideally we would like for all the images to be crisp and appropriately lit. Blurry images, dark images, and noisy images will all present problems for a CNN. Of course, we may have no choice but to accept poor quality images, but we need to know what sorts of problems there are so we can take the appropriate pre-processing steps.

4. **Number and distribution of images.** Learning from image data tends to require significantly more training samples than does learning from numerical data. In the previous two projects the datasets were relatively small. The sonar dataset only had 208 samples, while the diabetes dataset only had 768 samples — in total. The dogs vs. cats dataset contains 25,000 images in total. In addition to ensuring we have enough samples, we also want to make sure each class in our classification

scheme will have roughly the same number of samples, to avoid skewing the learning process towards one class or another. In the dataset for this project the distribution is exactly 50% cats and 50% dogs.

5. **Corrupt or non-image files.** We also want to see if there happen to be any image files that are either corrupt; i.e., they are image files, but they cannot be opened, or if there are any files in the dataset that are not image files. These files will either need to be removed or bypassed when creating the training and testing sets.

6. **Image metadata.** When a CNN attempts to extract features from an image, it is only interested in the pixel data that make up the image itself. However, image files contain information other than just the pixel data. For example, the date and time the images was taken (if it came from a camera), information about the camera used to take the image, descriptors and copyright information, and thumbnails are all examples of metadata that can be contained in an image file. This image metadata is more or less standardized by the Exif (Exchangeable image file format) standard, but there can be deviations from the standard. This is where the `piexif` library comes in. Although the library only has 5 functions, those functions can help eliminate some of the consistencies found in image metadata.

Loading Image Data Using Keras

Because a CNN will usually need to process thousands of images during the learning process, it won't be possible to store all the images in memory at the same time. To deal with this problem, `keras` provides a function, `flow_from_directory`, that generates small batches of images that can be loaded into memory an processed. This function also supports pre-processing and image augmentation, freeing the CNN (and us) from having to worry about memory management issues.

One vital stipulation of the function is that we must abide by a defined folder structure. Inside the base folder containing the entire dataset, there must be separate subfolders for the training samples and testing samples. Inside each of these folders there must be separate folders for each class of image. You have some freedom in how you name these folders, but the hierarchy must be maintained or you will end up with a training set or testing set that is empty, and you will get an error. Below is the hierarchy that will be assumed for this project:

```
Dataset
    |__PetImages
           |__Cat
           |
           |__Dog
           |
           |__Test
           |    |__Cat
           |    |__Dog
           |
           |__Train
                |__Cat
                |__Dog
```

The raw samples will be in the `Cat` and `Dog` folders directly inside the `PetImages` folder. When the dataset is split into a training set and a testing set, the training samples will be in the two folders inside the `Train` folder, while the testing samples will be in the two folders inside the `Test` folder. The code I will present will take care of creating the Test and Train subfolders, as well as the `Cat` and `Dog` folders inside each of them, and the raw data is already split into

separate `Cat` and `Dog` folders, so all you will need to do is copy the `PetImages` folder to your dataset folder.

One advantage of using a folder hierarchy like this is that the CNN will use the folder names to infer the output classes.

Splitting the Dataset into the Training and Testing Sets

Before creating the CNN we need to partition the dataset into a training set and a testing set. For the sake of clarity we'll write the code for this in a separate script from the script where we will define the CNN. The `Dataset_Utilities.py` script I have provided contains a function, `train_test_split`, that will take care of partitioning the dataset. For convenience, the code for the function is also shown on the next page. I'll run through the code piece by piece to explain how it works. First, here are the import statements for the dependencies we will need:

```
import os          # For functions that depend on the operating system
import shutil      # Allows high-level file operations such as removal of
                   # files an folder hierarchical trees
import random      # For random numbers
import piexif      # For dealing with image metadata
```

The first three dependencies are packaged with Python. The last one, `piexif`, is the one I mentioned earlier. We will use the `remove` function from this library to remove any corrupted exif metadata from the image files.

We define the `train_test_split` function to accept two arguments, 1) the path to the source folder for the images, and 2) the fraction of the dataset to use for training, expressed as a decimal. The relative base path for the images will be `Dataset/PetImages`, to reflect the folder structure of the downloaded dataset. We also specify a default value of 0.8 for the size of the training set, meaning we will use 80% of the dataset for training and 20% for testing. This is similar to what we have seen in the past. The value for this can be overidden, if desired.

The first thing the function does is remove any previously created `Train` and `Test` folders, along with their contents, and recreate them, so we can start over from scratch. This is for convenience, so we don't have to remember to do it manually. It is important to start with a blank slate, so to speak, each time, since when we randomly choose images for the training set we don't want to have any unchosen images hanging around from a previous run.

Next, we get to the main purpose of the function, which is to determine the number of each class of image to use for training, and partition the dataset by randomly selecting images for training, leaving any leftover images for the testing set. Since we are going to be traversing thousands of images in the process, we will also incorporate a little preprocessing by removing corrupt image files and non-image files.

First, we take advantage of the `os.walk` function to traverse the folder containing the cat images, storing all the filenames of the images in a list. The syntax for doing this is a bit unusual:

```
_, _, cat_images = next(os.walk(src_folder + 'Cat/'))
```

The `os.walk` function returns a 3-tuple rather than a single value. The first return value is the name of the base folder (referred to as `root`). The second return value is a list of all the subfolder names (referred to as `dirs`). The last return value is a list of all the file names (referred to as `files`). We only need the file names in this case, which we store in the variable,

```
def train_test_split(src_folder, train_size = 0.8):

        # Remove existing training and testing subfolders
        shutil.rmtree(src_folder + 'Train/Cat/', ignore_errors=True)
        shutil.rmtree(src_folder + 'Train/Dog/', ignore_errors=True)
        shutil.rmtree(src_folder + 'Test/Cat/', ignore_errors=True)
        shutil.rmtree(src_folder + 'Test/Dog/', ignore_errors=True)

        # Create new empty train and test folders
        os.makedirs(src_folder + 'Train/Cat/')
        os.makedirs(src_folder + 'Train/Dog/')
        os.makedirs(src_folder + 'Test/Cat/')
        os.makedirs(src_folder + 'Test/Dog/')

        # Retrieve the cat images
        _, _, cat_images = next(os.walk(src_folder + 'Cat/'))

        # These files are corrupt or they are non-image files
        files_to_be_removed = ['Thumbs.db', '666.jpg', '835.jpg']

        # Remove unwanted files
        for file in files_to_be_removed:
                cat_images.remove(file)

        # Calculate number of cat images left, and determine the number
        # of cat images to use for the training and testing sets
        num_cat_images = len(cat_images)
        num_cat_images_train = int(train_size * num_cat_images)
        num_cat_images_test = num_cat_images - num_cat_images_train

        # Now retrieve the dog images
        _, _, dog_images = next(os.walk(src_folder+'Dog/'))

        # These files are corrupt or they are non-image files
        files_to_be_removed = ['Thumbs.db', '11702.jpg']

        # Remove unwanted files
        for file in files_to_be_removed:
                dog_images.remove(file)

        # Calculate number of dog images left, and determine the number
        # of dog images to use for the training and testing sets
        num_dog_images = len(dog_images)
        num_dog_images_train = int(train_size * num_dog_images)
        num_dog_images_test = num_dog_images - num_dog_images_train

        # Randomly assign cat images to the training and testing sets
        cat_train_images = random.sample(cat_images, num_cat_images_train)

        for img in cat_train_images:
                shutil.copy(src=src_folder + 'Cat/' + img, dst=src_folder + 'Train/Cat/')

        cat_test_images  = [img for img in cat_images if img not in cat_train_images]

        for img in cat_test_images:
                shutil.copy(src=src_folder + 'Cat/' + img, dst=src_folder + 'Test/Cat/')

        # Randomly assign dog images to the training and testing sets
        dog_train_images = random.sample(dog_images, num_dog_images_train)

        for img in dog_train_images:
                shutil.copy(src=src_folder + 'Dog/' + img, dst=src_folder + 'Train/Dog/')

        dog_test_images  = [img for img in dog_images if img not in dog_train_images]

        for img in dog_test_images:
                shutil.copy(src=src_folder + 'Dog/' + img, dst=src_folder + 'Test/Dog/')

        # Remove corrupted exif data from the dataset
        remove_exif_data(src_folder + 'Train/')
        remove_exif_data(src_folder + 'Test/')
```

`cat_images`. We don't really care about storing the name of the root folder or the names of the subfolders, so we use Python's '_' identifier to store these values. This identifier is designed to be used as a placeholder for "throwaway" variables.

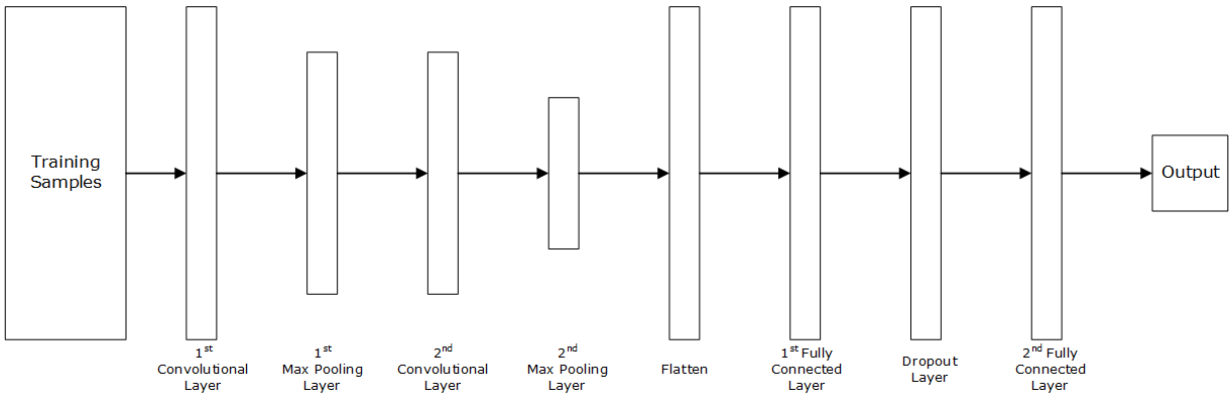
Next, we remove three files that are unwanted. The Thumbs.db file is peculiar to the Windows operating system. It is not an image file, but it stores the thumbnails of the smaller images that are displayed in Windows Explorer when the folder is viewed in a mode that shows icons for the files in addition to their names. The other two files are image files that are corrupt. We repeat this process for the dog images. In this case only two files need to be removed. As you probably noticed, the files needed to be specified by name; the script does not programmtically determine whether a file contains a corrupt image or non-image data. We could have manually removed these files prior to running the script, but there is actually precedent for not doing so. Even though these files do not represent valid cat images, at least as far as we know, they technically are still part of the dataset. As a matter of principle, it is inadvisable to alter the raw data manually, even if doing so seems to prove beneficial to us. If we remove the files from the raw dataset we also remove all traces they were ever in the dataset in the first place. This information could prove useful to us in the future, so we leave the original, raw dataset untainted.

After the unwanted files have been removed, it is a simple matter to use the `len()` function to find the total number of cat images, and then calculate the numbers of images to use for the training set and the testing set. The process of actually partitioning the dataset is straightforward. We use the `random.sample()` function to randomly select the proper number of images from the cat dataset, store those file names in the variable, `cat_train_images`, then use the `shutil.copy()` function to physically copy those files to the designated subfolder for cat training samples. Finally, we store the file names of all the cat images that were *not* selected for the training set into the variable, `cat_test_images`, and copy the files to the designated subfolder for cat testing samples. We then repeat the process to partition the dog images.

The last two statements of the function invoke a helper function to remove any corrupt exif metadata that may be present in the partitioned images. I won't show the code for that helper function here, as there is nothing remarkable about it. It simply walks the training and testing subfolders for both the cat and dog images, and invokes the `piexif remove()` function to remove the corrupt metadata.

Building the CNN

Now that we have partitioned the dataset into training samples and testing samples, we are ready to build the convolutional neural network. I've provided the basic script for building and running the CNN in `dogs_vs_cats.py`, but I'll go through each part of the script to explain how it works. The diagram below shows a rough overview of the architecture we will use for the CNN.



Step 1: Partition the Dataset

The first step we need to take is to partition the dataset into a training set and testing set, as described previously. We call the `train_test_split` function contained in the `dataset_utilities.py` script to carry out the partitioning, but since the function is in a separate script, we need to import it before we can use it.

Step 2: Build and Compile the CNN

Next, we build and compile the CNN. We first define a number of hyperparameters that will constrain the CNN. The hyperparameters we will use are as follows:

```
FILTER_SIZE = 3
NUM_FILTERS = 32
INPUT_SIZE  = 32
MAXPOOL_SIZE = 2
BATCH_SIZE = 16
STEPS_PER_EPOCH = 20000//BATCH_SIZE
EPOCHS = 10
```

Recall that the filter is the small "window" that slides across the image in a left-to-right, top-to-bottom fashion. The area inside the filter is the local receptive field, and each position of the filter creates an activation that is used to form the feature map for the previous layer. For this project we are using a 3x3 pixel filter, which may seem too small, but in fact filters of this size are commonly used.

> *Note that we only specify one dimension for the parameters. When only one dimension is specified for a 2D shape, it is assumed the shape will be a square.*

The number of filters will directly affect the length of time the learning process takes, since the more filters you use, the more processing must occur. We will use 32 filters here, which offers a decent tradeoff between performance and speed.

The input size represents the dimensions of the images that will be processed. The dimensions of the raw images are not consistent, which is to be expected. We could process the images as they are, but to reduce the computational complexity we will compress all the images into 32x32 pixels. This will greatly reduce the processing time for the CNN, although there is some potential for information loss due to the compression.

We will be using max pooling for subsampling, and we need to set the dimensions for the filter that will slide across the feature maps produced by the convolutional layers. We will use 2x2 filters for this, which will result in subsampled matrices that are half the size of the feature maps.

Since we have thousands of training samples, it would be impractical to try to load all the samples into memory and run them all at once. Therefore, we will split the training samples into batches, and the processing of all the batches will constitute one epoch. For this project we will use a batch size of 16 images. To calculate the number of iterations, or steps, required for one epoch, we divide the number of training samples by the batch size. Not counting the images that we removed in preprocessing, the dataset contained 25,000 images. We specified 80%, or 20,000 images for the training set. We defined the batch size as 16, so to get the number of iterations (or steps), we simply divide the number of training samples by the batch size. We use the '`//`' operator, which is the "floor" division operator in Python. In the event the number of training samples is not evenly divided by the batch size, this operator truncates the decimal value. Finally, we will use 10 epochs for training.

Once the hyperparameters have been defined, we need to construct the CNN. We start with the basic `Sequential` class, since like the other neural networks we have seen, the CNN will be a sequence of layers. Next, we add the first convolutional layer:

```
model.add(Conv2D(NUM_FILTERS, (FILTER_SIZE, FILTER_SIZE), input_shape =
(INPUT_SIZE, INPUT_SIZE, 3), activation = 'relu'))
```

The `Conv2D` class from Keras represents a two-dimensional convolutional layer. In the above statement, we need to pass the number of filters, the filter dimensions, the input shape, and the activation function. Since we are using square filters we pass the filter size for both the width and the height. The input shape is also square, so we pass the input size for the width and height. However, we have 3-channel images, so the input shape has a third dimension equal to the number of channels. In keeping with convention for CNNs, we use the Rectified Linear Unit (ReLU) activation function.

Next, we add a subsampling layer to use max pooling to reduce the size of the feature map created from the first convolutional layer:

```
model.add(MaxPooling2D(pool_size = (MAXPOOL_SIZE, MAXPOOL_SIZE)))
```

All we need to specify here is the dimensions of the pool, which we said would be 2x2.

This gives us one convolutional/subsampling pair. We need another pair, since most CNNs use at least two convolutional layers.

```
model.add(Conv2D(NUM_FILTERS, (FILTER_SIZE, FILTER_SIZE), activation =
'relu'))

model.add(MaxPooling2D(pool_size = (MAXPOOL_SIZE, MAXPOOL_SIZE)))
```

The second convolutional layer is different from the first in that it does not directly read the image data. Instead, it takes as input the output of the first max pooling layer. The number of filters and their dimensions remain unchanged, though. The second subsampling layer is identical to the first one.

We won't use any additional convolutional or subsampling layers in this CNN, so the nest step is to add the fully connected layer. However, first we must flatten the output of the second subsampling layer into a one-dimensional vector. For example, if we have a multidimensional matrix of dimensions (32, 32, 3), flattening this vector would produce a vector of dimensions (32 x 32 x 3) = (3072). The code for performing the flattening is:

```
model.add(Flatten())
```

With the flattening accomplished, we can now add the fully connected layer, which is a dense layer; i.e., the weights from every neuron in the previous layer are connected to all the neurons in the current layer. Keras's `Dense` class will give us a fully connected layer:

```
model.add(Dense(units = 128, activation = 'relu'))
```

The layer will have 128 neurons, and like the convolutional layers it will use the ReLU activation function.

To improve the accuracy of the CNN we will use a second fully connected layer, but to help prevent overfitting we will sandwich what is known as a **dropout layer** between the two fully connected layers. A dropout layer is a regularization technique that randomly sets a specified fraction of the synaptic weights to 0 in order to minimize the chances that a small number of weights will unduly influence the CNN's performance.

Keras provides the `Dropout` class especially for this purpose. The following code will create and add a dropout layer to the model that will set randomly 50% of the weights to 0 at each step:

```
model.add(Dropout(0.5))
```

Then we add the second and final fully connected layer. This layer will produce the output of the CNN, so we only use one neuron, and instead of using the ReLU activation function we switch to a non-linear sigmoid activation function.

```
model.add(Dense(units = 1, activation = 'sigmoid'))
```

This completes the construction of the CNN. It is now ready to be compiled, as follows:

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

We will use the adam optimizer to optimize the gradient descent as the synaptic weights are updated. This optimizer is very commonly used in training CNNs. Since we only have two output classes, which amounts to a binary result, we will use a binary cross-entropy loss function to help compute the updates to the weights. If we had more than two output classes we would use a categorical cross-entropy loss function instead. We also soecify accuracy as a performance metric, since we are interested in seeing what percent of the samples are correctly predicted.

Step 3: Train the CNN

The CNN is now ready to train. We first need to create an `ImageDataGenerator` to handle batch loading the images into memory, since we can't load them all into memory at once. Next, we generate the training set using the `flow_from_directory` function to load and train one batch of images at a time taken from the specified folder. The target size is compressed to 32x32 pixels, as we mentioned before. We also need to specify the batch size, as well as the fact we're only doing a binary classification. Finally, we call the model's `fit` function to carry out the actual training. We need to specify the training set to use, the number of epochs, and the number of steps per epoch, which we defined earlier. The verbose setting will force the display of the information for each epoch as training takes place. The code for doing these three tasks is shown below. Training this network will take significantly longer than it took to train the networks in the previous two assignments, but it should not be an impractical length of time. It took my laptop between 5-10 minutes to run through all 10 epochs. Expect an additional 10 minutes or so for partitioning the dataset.

```
# Create ImageDataGenerator to batch load images into memory
training_data_generator = ImageDataGenerator(rescale = 1./255)

# Create batch-loaded training set
training_set = training_data_generator.flow_from_directory(
               src + 'Train/',
               target_size = (INPUT_SIZE, INPUT_SIZE),
               batch_size = BATCH_SIZE,
               class_mode = 'binary')

# Train the model
model.fit_generator(training_set,
                    steps_per_epoch = STEPS_PER_EPOCH,
                    epochs = EPOCHS,
                    verbose=1)
```

Once training commences following the partitioning of the dataset, you should see output similar to the what I show below (results will vary for each training run). You should see the accuracy increase, and the loss decrease, with each epoch.

```
Epoch 1/10
1250/1250 [==============================] - 29s 23ms/step - loss: 0.6389 - accuracy: 0.6272
Epoch 2/10
1250/1250 [==============================] - 29s 23ms/step - loss: 0.5505 - accuracy: 0.7170
Epoch 3/10
1250/1250 [==============================] - 30s 24ms/step - loss: 0.5052 - accuracy: 0.7506
Epoch 4/10
1250/1250 [==============================] - 29s 24ms/step - loss: 0.4787 - accuracy: 0.7673
Epoch 5/10
1250/1250 [==============================] - 28s 23ms/step - loss: 0.4544 - accuracy: 0.7827
Epoch 6/10
1250/1250 [==============================] - 30s 24ms/step - loss: 0.4335 - accuracy: 0.7985
Epoch 7/10
1250/1250 [==============================] - 32s 25ms/step - loss: 0.4115 - accuracy: 0.8082
Epoch 8/10
1250/1250 [==============================] - 31s 25ms/step - loss: 0.3933 - accuracy: 0.8195
Epoch 9/10
1250/1250 [==============================] - 30s 24ms/step - loss: 0.3752 - accuracy: 0.8290
Epoch 10/10
1250/1250 [==============================] - 29s 23ms/step - loss: 0.3559 - accuracy: 0.8397
```

Step 4: Test the CNN

Now that the CNN has been trained, we test it using only the samples in the testing set. The code for testing is very similar to the code for training, except that we use the testing set as input instead of the training set, and instead of calling the model's `fit` function, we call the `evaluate_generator` function. The steps parameter for this function specifies the number of batches of samples to use.

```
# Create ImageDataGenerator to batch load images into memory
testing_data_generator = ImageDataGenerator(rescale = 1./255)

# Create batch-loaded testing set
test_set = testing_data_generator.flow_from_directory(
           src + 'Test/',
           target_size = (INPUT_SIZE, INPUT_SIZE),
           batch_size = BATCH_SIZE,
           class_mode = 'binary')

# Test the model
score = model.evaluate_generator(test_set, steps=100)
```

Results will vary from run to run, but you should see output similar to this:

```
loss: 0.4466818869113922
accuracy: 0.8006250262260437
```

These results indicate the CNN was 80% accurate on the testing set, which is a very good result, considering this CNN is only very basic. There are additional things we could do to improve its performance. We may explore some of these options later in the course, as time permits.

What You Need to Do:

Like the previous assignment, I will be providing you with a base script you can use, and I'm asking you to play around with the hyperparameters to see what effect those changes have on the performance of the CNN.

1. Download the dataset and the scripts I've provided.

It is imperative that you download the dataset using the link I've provided in order to ensure you have the correct folder structure. Otherwise, the scripts will not work.

2. Train and test the CNN implemented as described in this assignment.

I've provided the script, `dogs_vs_cats_CNN.py`, for you, so you just need to makes sure you have the dataset installed, and run the script.

3. Create a copy of your script to test each of the following changes to the CNN:

a. Increase the value for INPUT_SIZE to reduce the amount of compression of the images, and determine the effect on performance. I would not recommend using completely uncompressed images since it would take a very long time to train the CNN. I would recommend something like 64x64 pixels (twice the size we used in the original implementation), but feel free to experiment with other values if you like.

b. Try adding a third convolutional layer and max pooling subsampling layer, immediately before the flatten layer, to see what effect adding additional feature extraction has on performance. (Training will obviously take longer, but how will it affect accuracy?) You do not need to leave this third set of layers in place for the other experiments, but you can if you wish.

c. Try decreasing the number of filters to 24, to see if you can achieve comparable accuracy using fewer filters. (Feel free to try other values if you wish.)

d. Try replacing the max pooling subsampling layers with average pooling layers instead. The Keras class for this is `AveragePooling2D`. Below is a link to the API:

https://keras.io/api/layers/pooling_layers/average_pooling2d/

Important!

You will be submitting a total of 5 Python scripts, 1 for the original CNN based on the code I've presented in the assignment, and 4 separate scripts for the 4 experimental items under #2 above. You also need to include a brief description of how the changes affected the performance of the CNN.

What You Need to Turn In

Code:

Your 5 Python scripts as described above. *You do **not** need to submit the dataset!*

Results:

Brief descriptions of what you observed when you made the specified changes to the CNN.

Point Distribution

Item	Points Possible	Points
Script #1 (the original CNN)	10	
Script #2 (change the value for INPUT_SIZE)	10	
Script #3 (adding a third convolutional & subsampling layer)	10	
Script #4 (decrease the number of filters to 24)	10	
Script #5 (try using `AveragePooling2D` subsampling layers instead of `MaxPooling2D`)	10	
Total:	**50**	