The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

487/819– Computer Vision and Image Processing

Assignment 6

Date Due: December 6, 2018

Total Marks: 18

1 Submission Instructions

- Assignments are to be submitted via the Moodle class page.
- Programs must be written in Python 3.
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Background

The purpose of this assignment is to gain some experience with convolutional neural networks, design a convolutional neural network from scratch, and use them for prediction on realistic datasets.

We will be using the Python module `keras` (which is a high-level wrapper around Tensorflow) in this assignment. These packages should be installed on the Spinks lab machines. If you want to use them on your own computer, they can be installed with anaconda in the usual way using terminal commands:

```
conda install tensorflow
conda install Keras
```

2.1 Assignment Synposis

In Question 1 you'll design a convolutional neural network model that will be trained with the provided dataset of dog and cat images. Your network should then be able to predict whether previously unseen images are cat or dog.

You will design your network in five steps:

1. Design the network.
2. Prepare the training and validation images.
3. Run your CNN on training data to train the network.
4. Save the model.
5. Predict dog or cat from a test set of previously unseen images using the trained model and determine the correct classification rate.

We now elaborate with some additional background for each step.

2.2 Designing A Network

Recalling the lecture slides, we discussed how typical CNN architectures consist of four parts:

- Convolution layers
- Max-pooling layers
- Flattening
- Fully connected layers.

Here we discuss somewhat generally how to create a CNN of this type of architecture using Keras.

Initialize the CNN object

To create a CNN using Keras, we must first import and initialize a Keras object. Note that in the provided notebook, all the objects and modules you need are already imported in the first code block.

Call the `Sequential()` method from the `keras.models` module to create a new CNN object (it's already imported for you) and assign the result to a variable called `model`. This will be our CNN object.

Add Convolutional and Max-pooling Layers

The first thing to do is to add a convolution layer. This can be done with the `add()` method of the CNN object. The convolution layer is itself an object that must be created using the `Conv2D()` method from the `keras.layers` module. Thus, adding a convolution layer to your network looks something like this:

```
from keras.layers import Conv2D  # already done for you

model.add( Conv2D(num_feature_maps, map_dimension,
      input_shape = <tuple>, activation = <string>) )
```

where `num_feature_maps` is the number of feature maps (use 32), and `map_dimension` is a tuple defining size of each feature map (use `(3,3)`). The `input_shape` is a tuple defining the shape of the input image. Use `(64,64,3)` for a 64×64 RGB image. We can use larger input shapes but this will increase the computational cost significantly. The `activation` parameter selects the activation function to use; use `'relu'` to select the Rectified Linear Unit activation function.

Next we will add a max-pooling layer. To create such a layer, we must call the `MaxPooling2D` function from the `keras.layers` module. This will look something like this:

```
from keras.layers import MaxPooling2D. # already done for you

model.add( MaxPooling2D( pool_size = <tuple>) )
```

where `pool_size` is a tuple defining the size of neighbourhood to be pooled. Use `(2,2)`. We are using max-pooling because we want to capture strong responses to the convolution filter used in the convolution layer.

We can add as many pairs of convolution layers and max-pooling layers in our network as we want. **Add three more pairs** of convolution and max pooling layers to your network now! For these subsequent layers, we don't need to specify the `input_shape` because this can be inferred from the output of the earlier layers. So you can just cut-and-paste the previous two lines of code and remove `input_shape` parameter.

Adding a Flattening Layer

This is pretty straightforward. Call the `Flatten()` method from the `keras.layers` module and use the `add()` method of the CNN to add it:

```
from keras.layers import Flatten   # already done for you

model.add(Flatten())
```

Adding Fully Connected Layer(s)

We can add as many fully connected network layers as we want. First we need to create a suitable layer object by calling the `Dense()` method from the `keras.layers` module, then just again add it to our network using the `add()` method.

```
   from keras.layers import Dense     # already done for you

   model.add(Dense(units = <integer>, activation = <string>,
      kernel_regularizer=<string>))
```

The `units` parameter defines the number of outputs for the layer. The `activation` parameter selects the activation function for the layer. The `kernel_regularizer` parameter selects which regularizer to use. If you have more than one fully connected layer, the hidden layers (i.e. everything but the last output layer) should use `activation='relu'` and however many outputs you like. The final layer, the output layer, should use `activation='sigmoid'` for a two-class problem and `activation='softmax'` for more than two classes. Layer outputs are binary (either 0 or 1) so for a two-class problem, the final layer should have one output — the output is 0 or 1 depending on the class. For more than two classes, there should be one output per class; tensorflow allows only one of the outputs to be non-zero and this is the predicted class.

For our purposes we want to add a f connected later with 128 units, `'relu'` as the activation function, and an L2 regularization kernel. The latter can be done by specifying the parameter:

`kernel_regularizer=regularizers.l2(0.01)`

The `regularizers.l2` object comes from the `regularizers` module imported in the first block of the notebook.

Next, add another fully connected layer with only one unit, and the `'sigmoid'` activation function. This will be our output layer.

Compiling the Model

Once the network has been defined, we have to compile it by calling the `compile` method of our CNN object:

```
   model.compile(optimizer=<string>, loss=<string>, metrics=['accuracy'])
```

The `optimizer` parameter determines the algorithm used to modify the network weights during training. This can be, for example back-propogation. For this assignment use `optimizer='adam'` and `loss='binary_crossentropy'` for the loss function since it is suitable for a two-class problem.

(The loss function `optimizer='categorical_crossentropy'` would be appropriate for a multi-class problem.)

Display the Model Summary

After compiling, to see that everything is set up right, you can call the `summary()` method of the CNN object to display it's configuration to the console.

2.3 Preparing the Training and Validation Sets

We will use an `ImageDataGenerator` object from the `keras.preprocessing.image` module to load the data sets. We will need a training set from which to learn the model weights, and a validation set to see how accurate the classifier is with its current model weights. Note that both of these are distinct from the test set which is used to test the accuracy of the final model.

First, create an object from using the `ImageDataGenerator()` function:

```
train_datagen = ImageDataGenerator( <parameters> )
```

The parameters are tricky and have to do with real-time data augmentation. We'll give you the parameters you need to use in the starter code. You'll need to make one data generator object for the training set, and one for the validation set.

Once we've defined these objects, we need to tell them where to find the images. This is done with the `flow_from_directory` method:

```
training_set = train_datagen.flow_from_directory( training_set_folder ,
    target_size = <tuple>, batch_size = <integer>, class_mode = <string>)
```

Parameter `training_set_folder` is a string specifying the path to the folder containing the training images (this should be the folder containing the `cats` and `dogs` subofolder). Parameter `target_size` is a pair indicating the size to which all images should be resized — this must be the same as the width and height used for the first convolutional layer in the CNN (i.e. (64, 64)) `batch_size` is the batch size, and defaults to 32 which is a good value for us. Use `class_mode='binary'` since we have a two-class problem. (We could use `class_mode='categorical'` for a multi-class problem and there are other options.)

You'll have to call `flow_from_directory()` on both your training generator object and validation generator object. The parameters will be the same except for the folder name.

You don't have to worry about telling Keras which images are which classes. Keras infers this from the filesystem folder structure.

2.4 Training the CNN

To train the CNN call its `fit_generator` function:

```
history = model.fit_generator( training_set , steps_per_epoch = <integer>,
      epochs = <integer>, validation_data = validation_set ,
      validation_steps = <integer>, verbose = <integer>)
```

`training_set` is the training set object you created in the previ step. `steps_per_epoch` is the number of batches of training images (`batch_size` from previous step) to use per epoch. `epochs` is the number of epochs to train for. `validation_data` is your validation set object from the previous step. `validation_steps` is the number of validation images batches to use to compute the model's current accuracy. `verbose` is the verbosity mode, and must be equal to 0, 1, or 2, depending on how much progress information you want printed to the console.

`fit_generator()` returns an history object containing the loss and accuracies after each epoch. Make sure you assign the return value of `fit_generator()` to a variable so that we can use the returned information later.

2.5 Saving the Model

Models are saved in a format called H5. Use the `save` method of the CNN object to save the model configuration. This saves all of the model's layers, parameters, and current weights.

```
    model.save('<filename>.h5')
```

2.6 Plotting the Training and Validation Loss/Accuracy

The object returned by `fit_generator()` has an attribute called `history` which is a dictionary containing the keys `'acc'` (training accuracy), `'val_acc'` (validation accuracy), `'loss'` (training loss function value), and `'val_loss'` (validation loss function accuracy). The value of each of these keys is a sequence that can be plotted. Use this data to produce two plots, one that plots the training and validation accuracy against the epoch number, d one that plots the training and validation loss against the epoch number (epoch number on the x-axis, loss and accuracy on the y-axis).

2.7 Predicting an Image's Class Using the Trained Model

In the final part of your assignment you'll classify all of the images in a test set and obtain the correct classification rate for the test set.

A test image can be loaded with `keras.preprocessing.image` module:

```
from keras.preprocessing import image

test_image = image.load_img(<filename>, target_size = <tuple>)
```

This loads the given image filename and rescales it to `target_size` which is a tuple containing height and width in pixels. The height and width must be the same as that for the first convolutional layer of your CNN.

The image must then be converted into a numpy array:

```
test_image = image.img_to_array(test_image)
```

This will give you an RGB image array of the `target_size` of shape `(M,N,3)` (where `(M,N)` is the `target_size` from `load_img()`). But the CNN expects an array of colour images, so we have to take the `(M,N,3)` array and make it `(1,M,N,3)`:

```
test_image = np.expand_dims(test_image, axis=0)
```

Now we're ready to ask the classifier to predict the class of `test_image` using our CNN's `predict` method:

```
result = model.predict(test_image)
```

Keep in mind that `model.predict()` returns a float which is not necessarily exactly equal to 0 or 1. You will need to threshold to get a binary decision. Because of the order of the folders in the filesystem 0 is the expected result for cat, 1 is the expected result for dog.

This section has outlined how to classify a single image. You need to read all of the images in the test set, classify them, and determine the correct classification rate. You will know which images belong to which classes because they are in separate folders. You should expect the correct classification rate for the test set to be around 80–85%, give or take a few percent.

3 Problems

Question 1 (18 points):

Design and train a CNN for predicting the class of images of dogs and cats. Detailed instructions are provided in `asn6-q1.ipynb` and in section 2 above.

Expected output for step 1 (Designing the CNN architecture)

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 32)        896
-----------------------------------------------------------------
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 32)        0
-----------------------------------------------------------------
conv2d_2 (Conv2D)            (None, 29, 29, 32)        9248
-----------------------------------------------------------------
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 32)        0
-----------------------------------------------------------------
conv2d_3 (Conv2D)            (None, 12, 12, 32)        9248
-----------------------------------------------------------------
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 32)          0
-----------------------------------------------------------------
conv2d_4 (Conv2D)            (None, 4, 4, 32)          9248
-----------------------------------------------------------------
max_pooling2d_4 (MaxPooling2 (None, 2, 2, 32)          0
-----------------------------------------------------------------
flatten_1 (Flatten)          (None, 128)               0
-----------------------------------------------------------------
dense_1 (Dense)              (None, 128)               16512
-----------------------------------------------------------------
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 45,281
Trainable params: 45,281
Non-trainable params: 0
-----------------------------------------------------------------
```

Expected Output for Step 2 (loading the training and validation datasets)

```
Found 23000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
```

Sample output for Step 3 (training)

Your numbers won't match these exactly because of the random dataset augmentation.

```
Epoch 1/40
718/718 [==============================] - 223s - loss: 0.7679 - acc: 0.5479 - val_loss: 0.6451 - val_acc: 0.6400
Epoch 2/40
718/718 [==============================] - 225s - loss: 0.6234 - acc: 0.6600 - val_loss: 0.5689 - val_acc: 0.7120
Epoch 3/40
718/718 [==============================] - 236s - loss: 0.5537 - acc: 0.7221 - val_loss: 0.5529 - val_acc: 0.7160
Epoch 4/40
718/718 [==============================] - 218s - loss: 0.5095 - acc: 0.7555 - val_loss: 0.5118 - val_acc: 0.7445
Epoch 5/40
718/718 [==============================] - 244s - loss: 0.4733 - acc: 0.7830 - val_loss: 0.4416 - val_acc: 0.8005
```

```
Epoch 6/40
718/718 [==============================] - 238s - loss: 0.4465 - acc: 0.7979 - val_loss: 0.4133 - val_acc: 0.8135
Epoch 7/40
718/718 [==============================] - 234s - loss: 0.4248 - acc: 0.8079 - val_loss: 0.4190 - val_acc: 0.8210
Epoch 8/40
718/718 [==============================] - 226s - loss: 0.4066 - acc: 0.8203 - val_loss: 0.3848 - val_acc: 0.8265

[... many epochs omitted for brevity ...]

Epoch 36/40
718/718 [==============================] - 208s - loss: 0.2626 - acc: 0.8905 - val_loss: 0.2497 - val_acc: 0.8915
Epoch 37/40
718/718 [==============================] - 203s - loss: 0.2597 - acc: 0.8915 - val_loss: 0.2799 - val_acc: 0.8840
Epoch 38/40
718/718 [==============================] - 206s - loss: 0.2607 - acc: 0.8923 - val_loss: 0.2538 - val_acc: 0.8930
Epoch 39/40
718/718 [==============================] - 208s - loss: 0.2634 - acc: 0.8908 - val_loss: 0.2715 - val_acc: 0.8875
Epoch 40/40
718/718 [==============================] - 209s - loss: 0.2607 - acc: 0.8937 - val_loss: 0.2890 - val_acc: 0.8830
```

3.1 Important Note

If you are using Jupyter, be aware that if you re-run the training step without re-initializing your network from scratch from step 1, that you will be starting with a partly trained network already, and your results will get successively better and better. If you find that the training accuracy is starting at more than 50% then this is what's happened.

If you want to try different parameters, you must always run your notebook from the very beginning to accurately observe the effects.

4 Files Provided

`asn6-qX.ipynb`: These are iPython notebooks, one for each question, which includes instructions and in which you will do your assignment.

`ExampleCode.ipynb` Example of training a CNN to recognize the MNIST dataset.

`dataset_png_square` Folder of cat and dog image data containing subfolders for training set, validation set, and testing set respectively.

5 What to Hand In

Hand in your completed iPython notebooks, one for each question.

6 Appendix A — Grading Rubric

The grading rubric can be found on Moodle.