P6: Dog v. Cat Classification with Deep Learning

Deep learning has the potential to profoundly impact game design, development, and play. It offers a general method of learning complex functions from data, it is easy to use without a great deal of specialized knowledge (thanks to a growing repertoire of tools and libraries), and, in principle, it is vastly cheaper than the alternative of hand coding game features and solution algorithms for specific tasks. Within deep learning, classifiers acquire a mapping that assigns inputs to one of a finite set of categories. A surprising range of problems can be expressed in this form, including situation interpretation, action selection, and user preference modeling as might be encountered in games.

This assignment asks you to employ deep learning for image classification, specifically to recognize dogs vs cats from supervised data (images with labels as to their true identity). We add the wrinkle that the data is in short supply, which is a common problem that can be addressed with several technical tricks. You will build an initial classifier using a Convolutional Neural Net (CNN) for feature extraction, coupled to a fully connected neural net (also called densely connected layers) for final classification. Then, you will employ data augmentation, drop out layers, and transfer learning to improve the accuracy of your classifier.

The assignment includes an extra credit option to adapt your trained network to solve an entirely different classification task (sandwiches vs sushi) with a small amount of additional training.

We ask you to employ Keras for this assignment. Keras is deep learning middleware with an associated library, implemented on top of TensorFlow, which is in turn implemented in python.

The following sections describe the assignment in terms of the work flow you should follow.

1. Load and Install Keras

You can download Keras from https://keras.io/getting_started/

2. Gather the Data

The dog-v-cat dataset was made available by Kaggle as part of a computer vision competition in 2013. You will need to create a Kaggle account if you don't already have one. The pictures are medium resolution JPEGs of different sizes, taken in natural, mostly indoor settings. The winners in 2013 achieved 95% accuracy using convolutional networks. You will hopefully get similar results while training on 10% of the data they employed.

You can download the entire Dog v Cat dataset from www.kaggle.com/c/dogs-vs-cats/data

We have provided code to extract the first 2000 elements of the dog and cat data, and separate them into training, validation, and test sets. You will need to specialize this code to store the images in appropriate local directories on your machine. **It is important for this assignment that you use *only* the first 2000 elements of this dataset.**

Use the supplied code (`train_test_split.py`) to create training, validation and test sets for the cat and dog images.

3. Preprocess the Data

All deep learning tasks require a certain amount of data preprocessing. Here, you will need to:

1. Read in the image files
2. Preprocess the original jpeg data into RGB grids of pixels
3. Convert those into floating-point tensors (multi-dimensional arrays)
4. Rescale the values to the 0-1 range.

Keras has a module with image processing tools, located at `keras.preprocessing.image`. The class `ImageDataGenerator` lets you set up python generators that turn image files into batches of preprocessed tensors.

A python generator is an object that acts as an iterator (you can use it with the *for … in* operator). They are useful in deep learning applications for multiple reasons. Here, they avoid the need to keep all training, validation, and test data in memory by producing a data sample on each call. In addition, the generator can be modified to manipulate each piece of data as it enters the learning pipeline (you will utilize this capability later, in the data augmentation part of this assignment). Generators are built using the *yield* operator:

```
/* create a generator that yields integers */
def generator():
 i = 0
   while True:
         i += 1
         yield i

for item in generator():
   print(item)
   if item > 4:
         break

Output:
   1
   2
   3
   4
   5
```

Use the `ImageDataGenerator` class to initialize two generators (e.g, `train_datagen` and `test_datagen`) that scale the image pixel values to the (0,1) range, then build those generators using the function `flow_from_directory`. Set the image target size to 150 x 150, the batch size to 20, and specify `class_mode = 'binary'`, meaning your labels have binary values in a two-class problem. Implement this generator in the provided file `preprocess.py`

Note that the generators will produce data endlessly (looping over the images in the target folder), so you will need to insert a break statement whenever you enumerate values. The output of your generator for training data should have the following shape:

```
>>>  for data__batch, label_batch in train_generator:  /* your variable name here*/
>>>    print('data batch shape:', data_batch.shape)
>>>    print('labels batch shape:' labels_batch.shape)
>>>    break
```

```
data batch shape: (20, 150, 150, 3)
labels batch shape:  (20,)
```

4. Build an Initial Network

Compose a neural net for the dog v cat classification problem. It must have the following layers (in order):

- a CNN using some number of convolutional and maxpooling layers that gradually decrease the 150 x 150 input image to a 7x7 layer
- a single flatten layer
- a "hidden" densely connected layer
- a final densely connected layer that outputs a single number

Use 'relu' activation in the convolutional layers and the "hidden" dense layer, and sigmoid activation in the final layer.

Define this initial model in `models/basic_model.py`

5. Train your Network

Training a model in Keras only requires a couple of function calls. First, you need to configure the model for training.

```
from keras import optimizers
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])
```

The `loss` is the error measure that learning strives to reduce. Here, binary_crossentropy is the measure appropriate to 2-class classification problems. RMSprop is one of several available optimizers, while the parameter *lr* refers to a learning rate (the amount to adjust weights in the neural network in the direction that decreases the loss function). The parameter `metrics` is the function used to track progress during learning, through comparison of predicted vs actual values from the validation set. Here, 'acc' refers to mean squared error.

The code referenced in this section is split between `models/basic_model.py` and `train.py`. Run `python train.py` to run the model you define in `basic_model.py`

After configuring the model for training, you will need to fit the model using the fit.generator operator. The structure of the call is:

```
history = model.fit_generator (
  generator
  steps_per_epoch= 100
  epochs = 30
  validation_data = validation_generator
  validation_steps = 50)
```

Here, an epoch represents learning from 1 complete pass through the training data. The generator you defined earlier can produce an infinite sequence of batches (each containing 20 supervised data pairs) by recycling its data, so Keras needs a method of defining an epoch by the quantity of data the generator has produced. This is the role of the parameter `steps_per_epoch`; we compute the steps_per_epoch value as the total number of training data points (2000) divided by the batch size (20), giving a value of 100. In each step, Keras learns (performs one gradient descent pass through the model) on every supervised data pair in a batch. The parameter `validation_steps` plays the analogous role for controlling the amount of data produced for validation purposes from your `validation_generator`. You will need to alter the number of epochs to avoid overfitting as you train the various models in this assignment. Overfitting is the condition where *additional* training *decreases* accuracy of the learned mapping.

It is always good practice to save the model after learning, via a call like the following:

```
model.save('cats_and_dogs_small_1.h5')
```

This call saves both the architecture of the trained model (the layers and their connectivity) and the learned weights (collectively, the performance system). A saved model can be reloaded with an appropriate `load()` command.
We ask you to **submit the following:**

- **a printout showing the shape of your network, as generated by `model.summary()`.**
- **a plot showing training and validation accuracy as a function of epoch,**
- **a plot showing training and validation loss as function of epoch**

- **the accuracy and loss of your best learned model (obtained as the model in effect when overfitting begins) when measured against the held-back *test* set**
- **the model as an .h5 file**

We provide the code for generating these plots (see the file train.py). If all goes well, this classifier (a CNN with a densely connected back end) should achieve ~70% accuracy on the validation set.

6. Use Data Augmentation and Dropout

Data augmentation is a means of compensating for sparse training data. The idea is to modify each piece of training data in multiple ways that preserve its label but present new information to the learning system. In the case of image data, it is common to rotate, stretch, zoom, and flip each image (among other manipulations).

Keras provides a simple method of configuring a data generator to augment each training image. Read the documentation for `ImageDataGenerator` and configure it to perform data augmentations of your choice. (Remember not to augment the validation data - that test needs to be pure.) Experiment with the type and quantity of data augmentation and observe the impact on learning by plotting accuracy and loss during training. Overall, data augmentation is about remixing information present in the training data, and is not a substitute for employing additional training data if it is available.

While data augmentation will diminish overfitting when training on the same number of epochs, it's useful to employ other defenses against overfitting at the same time. In particular, a *dropout* layer randomly sets the inputs of its neurons to zero with some frequency at each gradient descent path through the network. There are many explanations for why dropout layers reduce overfitting. My favorite is that it forces the network to represent knowledge more diffusely, which inhibits its tendency to memorize training data. Memorizing training data is bad, as it causes poorer performance on held back data that demands generalization.

For this portion of the assignment, insert a dropout layer just after the flatten layer and before the "hidden" dense layer. Experiment with the dropout rate, and with the type and quantity of data augmentation. You may also experiment with the number of layers in the network. Then, pick your best performing configuration, and **submit the following**:

- **a printout showing the shape of your network, as generated by `model.summary()`.**
- **a plot showing training and validation accuracy as a function of epoch,**
- **a plot showing training and validation loss as function of epoch**

- **the accuracy and loss of your best learned model (obtained as the model in effect when overfitting begins) when measured against the held-back *test* set**

If all goes well, your classifier with data augmentation and dropout should achieve > 80% accuracy on the validation set.

Update the data generators in `preprocess.py` to add data augmentation. You will define your dropout model in `dropout_model.py`. Change the import line in `train.py` to choose which model is trained when you run `python train.py`

7. Use Feature Extraction

One way to bolster classification accuracy is to provide the classifier with inputs that encode features of the data useful for classification. Here, you will apply a pretrained image classifier to your data, extract a vector of features from it, and use those features to train a fresh fully-connected network to produce a final dog-v-cat classification. In essence, you are swapping out your convolutional base for the one in a large and very well-trained image classifier, performing knowledge transfer from it into the dog-v-cat classification task.

Keras provides a number of pre-trained image classifiers (e.g., Xception, Inception V3, ResNet50, VGG16, VGG19, Mobile Net). You will use VGG16 for this section, as it has a similar structure to the network you have already created. You can import VGG16 from the *keras.applications* module:

```
from keras.applications import VGG16
conv_base = VGG16(weights='imagenet',
                  include _top=False,
                  input_shape=(150, 150, 3))
```

Here:

- `weights='imagenet'` tells the system to use the weights obtained from training VGG16 on the ~1.3M images in the imagenet training set)
- `include_top = False` keeps discards VGG16's final fully connected layers that generate 1000 classes (and keeps the rest)
- `input_shape` is an optional parameter that specifies the shape of the image tensors you will feed to `conv_base`.

If you want to look at the structure of conv_base, use the command:

```
conv_base.summary()
```

For this section of the assignment, treat the conv_base as a separate performance system and apply it to generate a feature vector for every image in your 2000-member training set. You will store these extracted features in a Numpy array on disk. We have provided code for this purpose that you will need to modify to use the appropriate data source (see the file `feature_extraction.py`). Note the use of `conv_base.predict` to generate a feature vector from the pretrained VGG16 model, and the final aggregation of supervised data into training, validation, and test sets.

The extracted features are currently of the shape (samples, 4, 4, 512). You will need to feed them to a densely connected classifier. To do that, first flatten them into one-dimensional vectors:

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (2000, 4 * 4 * 512))
test_features = np.reshape(test_features, (2000, 4 * 4 * 512))
```

That done, define a new network to classify these features into the binary dog-v-cat distinction as before. This network should have at least 2 densely connected layers with a dropout layer in between. The hidden layer should use 'relu' activation, and the final layer should use 'sigmoid' activation, as before. Also use binary_crossentropy as the loss function, RMprop as the optimizer, and 'acc' as the metric in your instructions to model.compile.

Given the data is now explicit and on disk vs produced by a generator, use the `model.fit` function (vs model.fit_generator) to train the network (you can choose the epochs parameter):

```
history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20
                    validation_data=(validation_features, validation_labels)
```

Submit the following:

- **a printout showing the shape of your densely connected network, as generated by `model.summary()`**

- **a plot showing training and validation accuracy as a function of epoch**
- **a plot showing training and validation loss as function of epoch**
- **the accuracy and loss of your best learned model (obtained as the model in effect when overfitting begins) when measured against the held-back *test* set**

Note that we have omitted data augmentation in this section. In principle, you could reincorporate it by generating the suite of modified images, adding them (and the associated labels) to the training set, and storing that result on disk as an augmented training set for the densely connected network. You are not required to do that here. If all goes well, you will achieve ~90% accuracy with this model.

8. Extra Credit (3 points)

One of the interesting (and fun) things about deep learning models is that they provide leverage on other tasks that might not seem related. This section asks you to explore that effect by using your best trained model for the dog-v-cat classification task to address a sushi-vs-sandwich classification problem. The idea is (once again) to employ knowledge transfer. You will do this in two forms:

(1) Take the classifier (with all of its learned parameters) for the dog-v-cat task and retrain ***only*** the densely connected layer(s) at the very top against the supervised data for the new domain. That is, freeze all other layers.

(2) Take the classifier (with all of its learned parameters) for the dog-v-cat task and retrain all of its layers against the supervised data for the new domain.

For comparison, solve the same problem without knowledge transfer:

(3) Take the same classifier *without* its learned parameters and train it from scratch against the supervised data for the new domain.

You can download the sushi-vs-sandwich data from https://www.kaggle.com/brtknr/sushisandwich

Use the first 2000 elements of this dataset to train all three models, the **submit the following:**

- **a plot showing training and validation accuracy as a function of epoch,**
- **a plot showing training and validation loss as function of epoch**

- **the accuracy and loss of your best learned model (obtained as the model in effect when overfitting begins) when measured against the held-back *test* set**

Also **submit**:

- **a printout showing the shape of your network, as generated by `model.summary()`.**
- **A paragraph describing what you observed**. Which model required the least training to achieve its best performance (measured in epochs, and in elapsed time)? Which was the most accurate in the end? Why?

Supplied code

├── kaggle
- Extract the kaggle data here

├── models
│ ├── __init__.py
│ ├── feature_extraction_model.py
│ ├── basic_model.py
│ └── dropout_model.py
- These are where you define your keras models. They are imported and run via `python train.py`

├── feature_extraction.py
- For use in part 7 - extracts features using VGG16. Saves the features to `results/`

├── preprocess.py
- Scaffolding code for defining image data generators.
- `show_example_images` may be used to verify your solution is correctly iterating over the training and test data.

├── requirements.txt
- You can install all project dependencies by running `pip install -r requirements.txt` (or pip3)

├── train.py
- This is where you will fit the model. As provided, it imports from `models/basic_model.py`
- Included is a function to plot the model's training history with matplotlib.
- After training, it will save the model to an .h5 file. These can be reloaded with a call to the corresponding .load() method.

├── feature_extraction_train.py

- To be used similarly to `train.py` for part 7. Loads the feature vectors you created by running `feature_extraction.py` and uses them for training your `feature_extraction_model.py`

└── train_test_split.py

- Take a look at this first. After extracting the kaggle data to a `kaggle/` directory, run this script to move a small sample of the kaggle data to `cats_and_dogs_small/`