Initialization

In [1]: %load_ext autoreload
%autoreload 2
%matplotlib inline

In [2]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

In [3]: import os
from tensorflow.keras.preprocessing.image import load_img, img_to_array, array_to_img,

from tensorflow.keras.preprocessing.image import Iterator

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout

from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import RMSprop

Acknowledgement

- This lecture, including the code, is based closely on section 5.2 of: François Chollet: Deep Learning with Python, Manning Publications, 2018

Warning

- Training the networks in this notebook might each take an hour
- The only exception is when we use early stopping, in which case training will take several minutes
It's Raining Cats and Dogs

- A dataset, supplied by Microsoft researchers, for a Kaggle competition: https://www.kaggle.com/c/dogs-vs-cats
  - 12,500 medium-resolution JPEGs depicting cats and 12,500 depicting dogs
- We use a subset of the full dataset:
  - training set: 1000 cats and 1000 dogs
  - validation set: 500 cats and 500 dogs
  - test set: 500 cats and 500 dogs

In [4]:
base_dir = "datasets/dataset_cats_and_dogs"
train_dir = os.path.join(base_dir, "train")
val_dir = os.path.join(base_dir, "validation")
test_dir = os.path.join(base_dir, "test")

- Let's look at one of the dogs:

In [5]:
train_dogs_dir = os.path.join(train_dir, "dogs")
filenames = [os.path.join(train_dogs_dir, filename) for filename in os.listdir(
idx = 303  # Change this if you want to look at a different dog
some_example = load_img(filenames[idx], target_size=(150, 150))
nplt.imshow(some_example)

Out[5]:

Data Preprocessing

- We must
  - Decode JPEGs to uncompressed grids of RGB pixels
  - Convert the RGB values (integers in $[0, 255]$) to floats in $[0,1]$)
  - Resize images so they're all the same size, in our case $150 \times 150$
  - Encode the class labels appropriately
- It's also better not to read the whole dataset into main memory
  - We can process it in mini-batches and should read and convert batches on an as-needed basis
  - We'll use mini-batches of size 20
- In Keras, the ImageDataGenerator class does all this!
  - It's flow_from_directory function yields batches indefinitely, in an infinite loop
  - (It uses a Python generator)
Creating the ConvNet

- Since the images are bigger than the MNIST ones, we use a network with more layers

```python
train_data_generator = ImageDataGenerator(rescale=1./255)
train_generator = train_data_generator.flow_from_directory(
    train_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

val_data_generator = ImageDataGenerator(rescale=1./255)
val_generator = val_data_generator.flow_from_directory(
    val_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

test_data_generator = ImageDataGenerator(rescale=1./255)
test_generator = test_data_generator.flow_from_directory(
    test_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

def build_convnet():
    network = Sequential()
    network.add(Conv2D(32, (3, 3), activation="relu", input_shape=(150, 150, 3))
    network.add(MaxPooling2D((2, 2)))
    network.add(Conv2D(64, (3, 3), activation="relu")
    network.add(MaxPooling2D((2, 2)))
    network.add(Conv2D(128, (3, 3), activation="relu")
    network.add(MaxPooling2D((2, 2)))
    network.add(Conv2D(128, (3, 3), activation="relu")
    network.add(MaxPooling2D((2, 2)))
    network.add(Flatten())
    network.add(Dense(512, activation="relu")
    network.add(Dense(1, activation="sigmoid")
    network.compile(optimizer=RMSprop(lr=0.0003), loss="binary_crossentropy",
    return network

network = build_convnet()
```
In [9]:

```python
network.summary()
```

```
Model: "sequential"

Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 32)      896
max_pooling2d (MaxPooling2D) (None, 74, 74, 32)        0
conv2d_1 (Conv2D)            (None, 72, 72, 64)        18496
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
conv2d_2 (Conv2D)            (None, 34, 34, 128)       73856
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 128)       0
conv2d_3 (Conv2D)            (None, 15, 15, 128)       147584
max_pooling2d_3 (MaxPooling2 (None, 7, 7, 128)         0
flatten (Flatten)            (None, 6272)              0
dense (Dense)                (None, 512)               3211776
dense_1 (Dense)              (None, 1)                 513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Training and Testing

- When using generators, we train using `fit_generator`, instead of `fit`
- Since the generator draws mini-batches indefinitely, we must specify arguments to make it terminate:
  - `steps_per_epoch`: since we asked for mini-batches of size 20, an epoch that contains 100 steps will cover all 2000 training examples
  - `epochs`: this is how many epochs — in our case, we guess at 30

In [10]:

```python
network.fit_generator(train_generator, steps_per_epoch=100, epochs=30, verbose=0)
```

Out[10]:

```
<tensorflow.python.keras.callbacks.History at 0x7f59306e3a90>
```

- And now we can evaluate on the validation set (but not the test set — not until we’ve finished with all fine-tuning!)
- We use `evaluate_generator` instead of `evaluate` and specify `steps` — with mini-batch size 20 again, 50 steps will cover all 1000 validation examples

In [11]:

```python
val_loss, val_acc = network.evaluate_generator(val_generator, steps=50)
```

Out[11]:

```
val_acc
```

0.732
We can even combine these:
- We can arrange for `fit_generator` to test on the validation data after each epoch
- We can return a `History` object, which records the values of the loss metric and the other metrics we specified (accuracy) on both the training set and the validation set for each epoch
- From the `History` object, we can write code to plot loss and accuracy against epochs

```
In [12]:
    history = network.fit_generator(
             train_generator, steps_per_epoch=100, epochs=30,
             validation_data=val_generator, validation_steps=50,
             verbose=0)

In [13]:
    x_vals = range(1, 31)

In [14]:
    train_loss = history.history["loss"]
    val_loss = history.history["val_loss"]

    fig = plt.figure()
    plt.title("Loss")
    plt.plot(x_vals, train_loss, label="Training loss", color = "purple")
    plt.plot(x_vals, val_loss, label="Validation loss", color = "orange")
    plt.legend()
    plt.show()

In [15]:
    train_acc = history.history["accuracy"]
    val_acc = history.history["val_accuracy"]

    fig = plt.figure()
    plt.title("Accuracy")
    plt.plot(x_vals, train_acc, label="Training accuracy", color = "purple")
    plt.plot(x_vals, val_acc, label="Validation accuracy", color = "orange")
    plt.legend()
    plt.show()
```
Since this all takes so long, we can use early stopping:
- During training, monitor a metric, such as loss on the validation set
- Interrupt training when that metric has stopped improving for a certain number of epochs
- In Keras, this is done using the EarlyStopping callback
  - E.g. patience = 2 interrupts training when the metric has stopped improving for more than two epochs
  - Then, restore_best_weights=True restores weights from the epoch before they stopped improving

```python
In [16]: history = network.fit_generator(
                   train_generator, steps_per_epoch=100, epochs=30,
                   validation_data=val_generator, validation_steps=50,
                   callbacks=[EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True)]
```

Out[16]:

```
{'loss': [0.01568174646220788, 0.01868000215170998, 0.006528019693770766],
 'accuracy': [0.996, 0.996, 0.9985],
 'val_loss': [2.339303926229477, 2.4560345017910006, 2.435882581472397],
 'val_accuracy': [0.726, 0.698, 0.715]}
```

- In this case, it interrupts training after a handful of epochs
- This takes much less time than running for the full 30 epochs
- It's not worth showing a graph

**Data Augmentation**

- With so few examples, overfitting is a major concern
- We know all sorts of things we can try in order to reduce the overfitting
  - We'll add a dropout layer below
- Another way to reduce overfitting is to get more examples but here we do something similar: data augmentation
- We augment the training set with examples that we synthesize from the existing training set
- It's relatively easy when the dataset consists of images to use transformations on existing images to synthesize believable-looking new images
- Be aware that this is not as good as additional real examples
  - These synthesized examples are correlated with each other and the originals from which they were generated
- In Keras, the `ImageDataGenerator` can be configured to do this for us, and we will use the following ones here:
  - `rotation_range`: degree range for random rotations
  - `width_shift_range`: range for random horizontal shifts
  - `height_shift_range`: range for random vertical shifts
  - `shear_range`: Shear angle in counter-clockwise direction as radians
  - `zoom_range`: range for random zoom
  - `horizontal_flip`: randomly, whether to flip inputs horizontally
  - `fill_mode`: the strategy for filling-in newly created pixels that appear after some of the other transformations
- We only augment the training data generator, not the validation or test data generators
Out of curiosity, let's take the example image from earlier and see some of its transformations.

```
In [19]:
    augmented_train_data_generator = ImageDataGenerator(
        rescale=1./255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode="nearest")
    augmented_train_generator = augmented_train_data_generator.flow_from_directory(
        train_dir, target_size=(150, 150), batch_size=20, class_mode="binary")

Found 2000 images belonging to 2 classes.
```

- Out of curiosity, let's take the example image from earlier and see some of its transformations.
In [20]:
x = img_to_array(some_example)
x = x.reshape((1,) + x.shape)

i = 0
for batch in augmented_train_data_generator.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()
Creating, Training and Testing

```python
In [21]:
def build_convnet_with_dropout():
    network = Sequential()
    network.add(Conv2D(32, (3, 3), activation="relu", input_shape=(150, 150, 3))
    network.add(Conv2D(64, (3, 3), activation="relu"))
    network.add(MaxPooling2D((2, 2)))
    network.add(Conv2D(128, (3, 3), activation="relu"))
    network.add(MaxPooling2D((2, 2)))
    network.add(Conv2D(128, (3, 3), activation="relu"))
    network.add(MaxPooling2D((2, 2)))
    network.add(Flatten())
    network.add(Dropout(0.5))
    network.add(Dense(512, activation="relu"))
    network.add(Dense(1, activation="sigmoid"))
    network.compile(optimizer=RMSprop(lr=0.0003), loss="binary_crossentropy",
    return network

In [22]:

In [23]:

In [24]:
```

```python
history = network_with_dropout.fit_generator(
    augmented_train_generator, steps_per_epoch=100, epochs=30,
    validation_data=val_generator, validation_steps=50,
    verbose=0)

train_loss = history.history["loss"]
val_loss = history.history["val_loss"]
```

```python
fig = plt.figure()
plt.title("Loss")
plt.plot(x_vals, train_loss, label="Training loss", color = "purple")
plt.plot(x_vals, val_loss, label="Validation loss", color = "orange")
plt.legend()
plt.show()
```
In [25]:

```python
train_acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]

fig = plt.figure()
plt.title("Accuracy")
plt.plot(x_vals, train_acc, label="Training accuracy", color = "purple")
plt.plot(x_vals, val_acc, label="Validation accuracy", color = "orange")
plt.legend()
plt.show()
```

- How did we do on the test set?

In [26]:

```python
test_loss, test_acc = network_with_dropout.evaluate_generator(test_generator, steps=test_acc)
```

Out[26]: 0.771

### Saving and Restoring Models

- Having learned the weights, we should save a model so that we don't have to learn them again!
- `save` is a method that saves a Keras model (network, weights, training configuration, state of the optimizer) into a HDF5 file
- `load_model`: reinstatates the model, including compiling the model using the saved training configuration

In [27]:

```python
network_with_dropout.save("models/my_model.h5")
```

In [28]:

```python
reloaded_network = load_model("models/my_model.h5")
```

### Example Predictions

- Let's read in some images, display them, turn them into a tensor and see what the model predicts for the images in this tensor
```python
In [29]:
path = "datasets/dataset_wikipedia_images"
pathnames = [os.path.join(path, filename) for filename in sorted(os.listdir(path))]
imgs = [load_img(img_path, target_size=(150, 150)) for img_path in pathnames]
for img in imgs:
    plt.figure()
    plt.imshow(img)
```

In [30]:
```
    img_tensor = np.array([img_to_array(img) for img in imgs])
    img_tensor /= 255
```

In [31]:
```
    reloaded_network.predict(img_tensor)
```

Out[31]:
```
array([[0.03515115],
       [0.99756575],
       [0.933441  ],
       [0.24576798]], dtype=float32)
```

In [ ]: