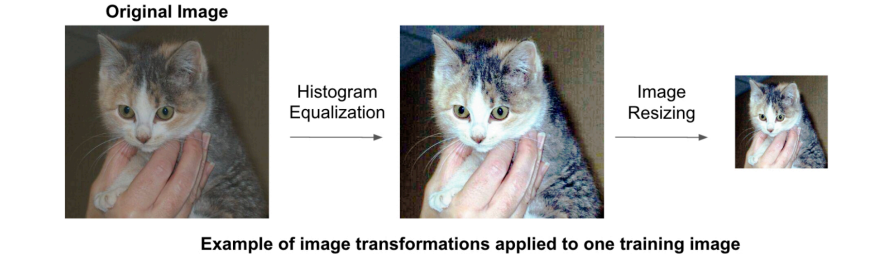
A Gentle Introduction to Convolutional Neural Networks

- Shashank Mujumdar, Abhinav Jain
IBM Research, India

IBM

Building a Deep Neural Network for Image Classification with Keras

Implementing a Neural Network



Example of image transformations applied to one training image

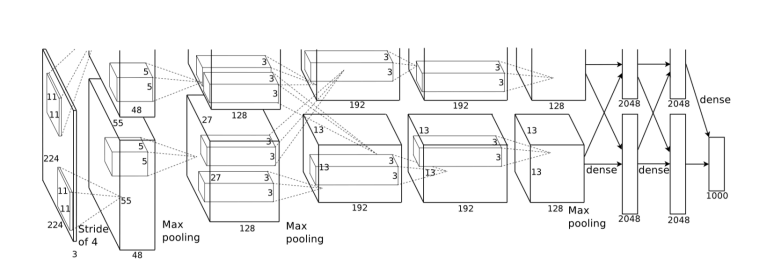


Fig. A sample Convolutional Neural Network for Classification

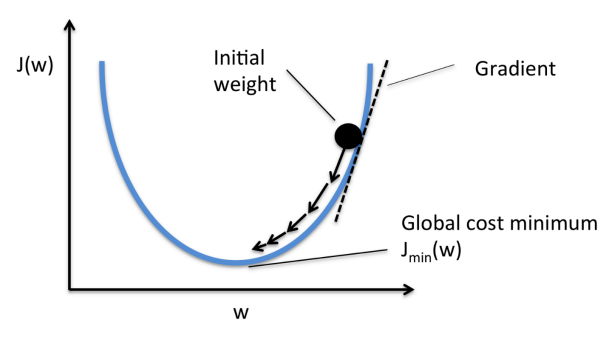
1. Data Preparation

- Image Resizing
- Image Normalization
- Random data transformation or pre-processing required by the model for specific purposes

2. Model Architecture

- Define layers – convolutional, max-pooling, fully-connected etc.
- Configure each layer – specify parameters like num of filters, stride, activation function etc.

Implementing a Neural Network



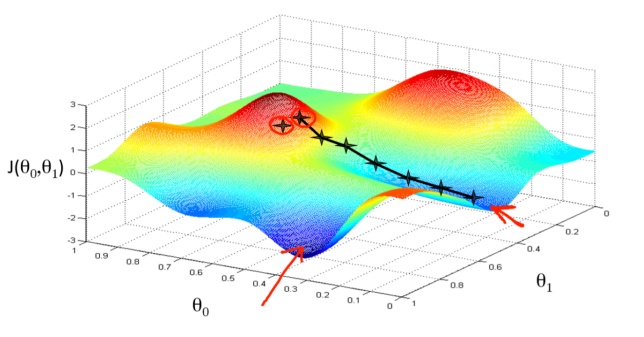


Fig. Minimizing loss function using Gradient Descent at training time. (Illustration) 2-Dimensional (left) v/s 3-Dimensional (right)

3. Model Configuration

- Specify the optimizer – SGD, Adam, RMSProp
- Configure the learning process with batch size, iterations, number of epochs, learning rate etc.
- Specify the Loss function – Cross Entropy, MSE, Hinge etc.

4. Model training and testing

- Train the model using training data
- Optimize for model parameters using validation data
- Evaluate the model using test data

USE-CASE

CLASSIFYING DOG-CAT IMAGES FROM KAGGLE

Building an Image Classifier using Conv Nets

System Requirements

- ✓ Python 3.0+
- ✓ Keras 2.0.0+
- ✓ Numpy
- ✓ Scipy
- ✓ PIL

Outline

We will go over the following training regimes:

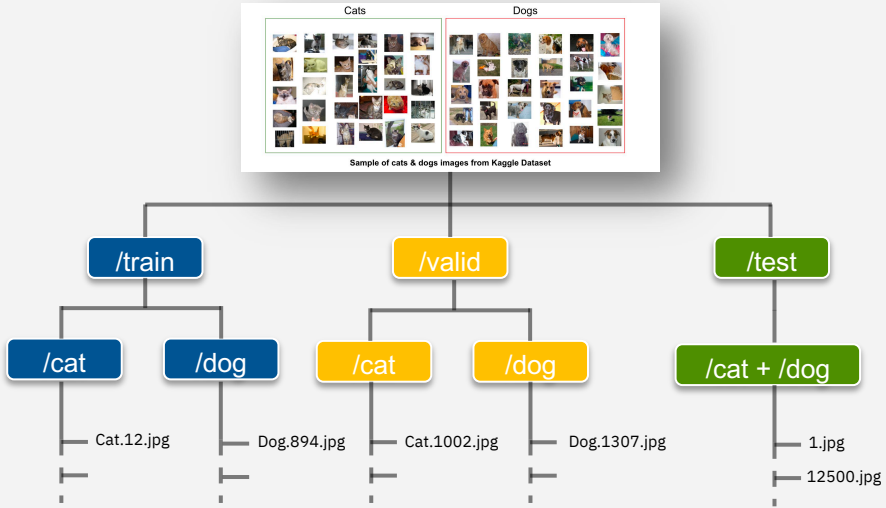
- Training a neural network from scratch (for baseline)
- Using the bottleneck features of pre-trained VGG16
- Fine-tuning the top layers of pre-trained VGG16

We will learn about following features of Keras

- Sequential and Functional API for model definition
- Fit_generator and fir methods for model training
- ImageDataGenerator for real-time data augmentation
- Layer-freezing and model fine-tuning

Training the neural network from scratch – Data Preprocessing

Dataset



- Kaggle Cats and Dogs Dataset for Image Classification
- Number of Samples per class (original) are 12,500
- For our use-case: 1000 samples for training and 200 for validation per class

Data Augmentation and Generation

- For a small dataset, we augment images with random transformations so that model does not see an image twice. This helps the model generalize better
- We will use Keras provided method for data generation (Simple and easy to use)

Fig. Data augmentation example: random transformations of a sample image

Training the neural network from scratch – Data Preprocessing (Code)

Keras module for image preprocessing

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
```

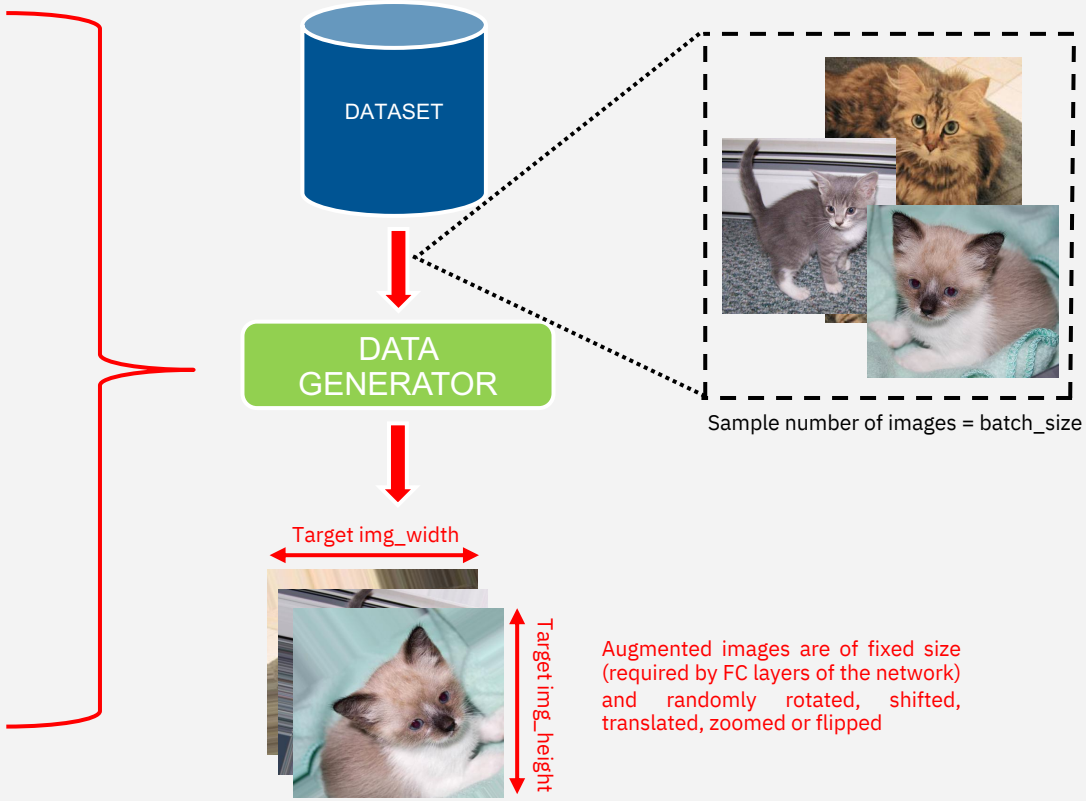
```
datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')
```

Configure random transformations

Src: https://keras.io/preprocessing/image/

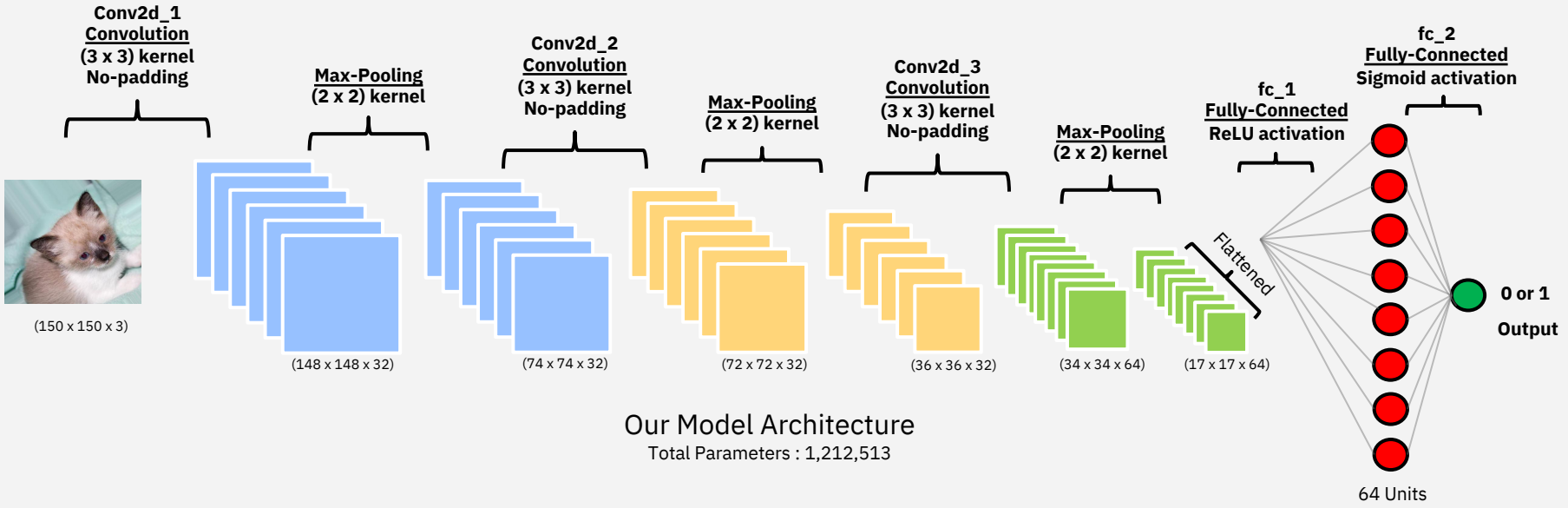
```
train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(img_width, img_height),
        batch_size=batch_size,
        class_mode='binary')
```

Use 'flow_from_directory' method of 'ImageDataGenerator' to instantiate a data generator that reads pictures from sub-folders of 'data/train' and generates batches of augmented image data



Augmented images are of fixed size (required by FC layers of the network) and randomly rotated, shifted, translated, zoomed or flipped

Training the neural network from scratch – Defining Model



Our Model Architecture

Total Parameters : 1,212,513

Key features of our model:

- It is a small convNet consisting of only 3 convolutional layers.
- **Overfitting**: Occurs when model is exposed to too few examples and and learns patterns that do not generalize to new data
- Data augmentation as one of the strategies to prevent overfitting but it might not work if augmented samples are highly correlated.
- Use of dropout to prevent overfitting by preventing a layer (in our case, fc_1) from seeing twice the exact same pattern

Other ways of controlling our model's entropic capacity:

- Other way is to optimize for number of layers and size of each layer in our network
- We can also use L1 or L2 regularization to force model weights to take smaller values

Training the neural network from scratch – Defining Model (Code)

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

1

2

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

3

1. We import the Sequential API of Keras to construct our model
2. We then import the following layers:-
   - Conv2D(num_of_filters, kernel_size): Convolutional Layer
   - MaxPooling2D(pool_size): Max Pooling Layer
   - Activation('act_fn'): To define activation function for the preceding layer
   - Flatten(): To collapse the preceding 17 x 17 x 64 feature map into one-dimensional 18496 x 1 feature vector required by the subsequent fully-connected layers.
   - Dense(num_units): Fully-connected layer
   - Dropout(drop_prob): To randomly drop activations of the preceding layer
3. We use sigmoid function which is perfect for binary classification. It evaluates to a probabilistic value that determines how close model prediction is to the ground truth label.

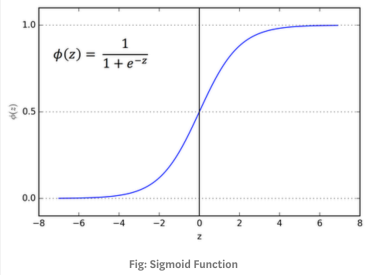


Fig: Sigmoid Function

Training the neural network from scratch – Model Optimization and Training

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

1

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

2

$$-(y \log(p) + (1 - y) \log(1 - p))$$

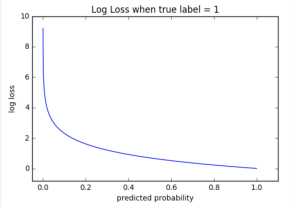


Fig. Cross-entropy Loss equation (left) and plot (right)

```
model.fit_generator(
        train_generator,
        steps_per_epoch=nb_train_samples // batch_size,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=nb_validation_samples // batch_size)
```

3

1. We use Keras in-built optimizers like 'Stochastic Gradient Descent, *sgd*', 'Root Mean-Square Propagation, *rmsprop*', *'adagrad'* etc.
2. Before training the model, we have to compile (or compile) the model.
   - *Loss*: Objective function that needs to be optimizer. Here, we minimize the binary cross entropy loss. It measures the performance of a classification model whose output is a probability between 0 and 1
   - *Optimizer*: Each optimizer can be declared separately along with its parameters. But for now, we use default values and pass the optimizer as an argument to *compile* method.
   - *Metrics*: List of metrics to be evaluated by the model during training and testing. In our case, we want to measure the *accuracy* of the model while predicting labels (*0 for cat or 1 for dog*).
3. Finally, we use *fit_generator* method to train the model on data generated batch-by-batch. It accepts following arguments.
   - *Training data generator:* The generator instance that we created using *'ImageDataGenerator'* and *'flow_from_directory'*
   - *Num of epochs:* Number of iterations where each iteration runs over entire data.
   - *Steps per epoch:* In how many batches do we want to cover the entire training data. This is set to number of samples divided by batch size.
   - *Validation data:* Generator instance for validation data
   - *Validation Steps:* In how many batches do we want to cover the entire validation data.

Training the neural network from scratch – Model Optimization and Training

1

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

2

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

$$-(y \log(p) + (1 - y) \log(1 - p))$$

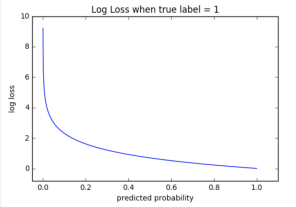


Fig. Cross-entropy Loss equation (left) and plot (right)

3

```
model.fit_generator(
        train_generator,
        steps_per_epoch=nb_train_samples // batch_size,
        epochs=epochs,
        validation_data=validation_generator,
        validation_steps=nb_validation_samples // batch_size)
```

Baseline Results:-
Validation Accuracy of 74-78% after 50 epochs

Can we achieve better accuracy in shorter time?

Using the bottleneck Features of a Pre-Trained Network - Overview

- In computer vision applications, we can achieve much ***better accuracy*** by leveraging a pre-trained network on a larger dataset.
- In the following part, we will use the VGG16, a network pre-trained on ***1000 classes*** of ***ImageNet Dataset*** (includes multiple 'cat' and 'dog' classes)

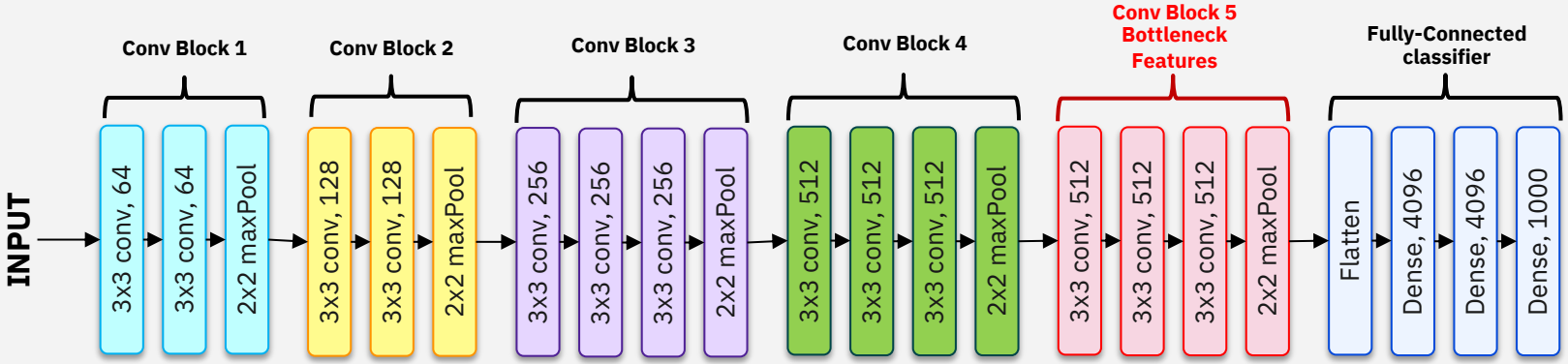


Fig. VGG16 architecture for 1000 class image classification

How do we adapt a pre-trained network for our use case?

- We will use the bottleneck features, which are features from the last activation maps (Conv Block 5) before the fully-connected layers
- Record the features for both training and validation data
- Train a fully-connected model on top of the stored features

Using the bottleneck Features of a Pre-Trained Network - Code

1
```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

2
```
# build the VGG16 network
model = applications.VGG16(include_top=False, weights='imagenet')
```

3
```
generator = datagen.flow_from_directory(
train_data_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode=None,
shuffle=False)

bottleneck_features_train = model.predict_generator(generator, nb_train_samples // batch_size)

np.save('bottleneck_features_train.npy', bottleneck_features_train)
```

4
```
train_data = np.load('bottleneck_features_train.npy')
train_labels = np.array([0] * (nb_train_samples // 2) + [1] * (nb_train_samples // 2))
```

5
```
model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

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

model.fit(train_data, train_labels,
          epochs=epochs,
          batch_size=batch_size,
          validation_data=(validation_data, validation_labels))
```

1. Import Keras pre-defined models using '*applications*' module
2. Import in particular, the VGG16 model trained on *'ImageNet'* weights. Make sure to import the model without its top fully-connected layers, set '*include_top*' as **False.** (Note: Weights are downloaded automatically)
3. Instantiate a generator for training and validation data. Use ***'predict_generator'*** to predict Conv_5 features for each sample. Lastly, save the features into a NumPy array
4. Load the bottleneck features back into a NumPy array and set the class labels for each sample.
5. Define the fully-connected classifier (Using Sequential Model API) consisting of two dense layers (256 units and 1 unit respectively). Compile the model and train it (Conv Layers : frozen, FC Layers : trainable) using *'fit'* method.

Note: The 'fit' method is different from 'fit_generator' method. Former requires complete dataset for training while later requires a generator that provides samples batch-by-batch.

Using the bottleneck Features of a Pre-Trained Network - Code

1

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential, Model
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense, Input
from keras import backend as K
from keras import applications, optimizers
import numpy as np
```

2

```
# build the VGG16 network
model = applications.VGG16(include_top=False, weights='imagenet')
```

3

```
generator = datagen.flow_from_directory(
train_data_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode=None,
shuffle=False)

bottleneck_features_train = model.predict_generator(generator, nb_train_samples // batch_size)

np.save('bottleneck_features_train.npy', bottleneck_features_train)
```

4

```
train_data = np.load('bottleneck_features_train.npy')
train_labels = np.array([0] * (nb_train_samples // 2) + [1] * (nb_train_samples // 2))
```

5

```
model = Sequential()
model.add(Flatten(input_shape=train_data.shape[1:]))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

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

model.fit(train_data, train_labels,
          epochs=epochs,
          batch_size=batch_size,
          validation_data=(validation_data, validation_labels))
```

Validation Accuracy of ~90% after 50 epochs

Taking a step further – Fine-tuning (Using Functional API of Keras)

With fine-tuning, we can further adapt VGG16 model to the defined task. Fine-tuning retrains an already trained network on a new dataset using 'small' weight updates. For our use-case, we follow following steps:

- Instantiate the convolutional base of VGG16 and load its weights
- Add our previously-defined FC classifier and load its weights
- Freeze the layers of the model up to last conv block; we only fine-tune Conv5 Block of our model

Import the VGG16 model without its top layers and pre-set weights (1)

```
# build the VGG16 network
model_vgg16_conv = applications.VGG16(include_top=False, weights='imagenet', input_shape=input_shape)
```

Build the classifier model and load its weights. (2)

```
input_FC = Input(shape=model_vgg16_conv.output_shape[1:])
x = Flatten()(input_FC)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)
FC_model = Model(input=input_FC, output=x)
FC_model.load_weights(top_model_weights_path)
```

Note that architecture of the defined model should match with the model whose weights we are loading.

(3) Add the FC classifier on top of the convolutional base.

```
input_model = Input(shape=input_shape)
output_vgg16_conv = model_vgg16_conv(input_model)
prediction = FC_model(output_vgg16_conv)
model = Model(input=input_model, output=prediction)
```

(4) Fine-tuning: Freeze the first 25 layers i.e. set them as untrainable

```
for layer in model.layers[:25]:
    layer.trainable = False
```

(5) Compile the model using a non-adaptive optimizer like SGD with a small learning rate.

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

Taking a step further – Fine-tuning (Using Functional API of Keras)

Validation Accuracy of ~95% after 50 epochs at the cost of higher computation time

Import the VGG16 model without its top layers and pre-set weights (1)

```
# build the VGG16 network
model_vgg16_conv = applications.VGG16(include_top=False, weights='imagenet', input_shape=input_shape)
```

Build the classifier model and load its weights. (2)

```
input_FC = Input(shape=model_vgg16_conv.output_shape[1:])
x = Flatten()(input_FC)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(1, activation='sigmoid')(x)
FC_model = Model(input=input_FC, output=x)
FC_model.load_weights(top_model_weights_path)
```

Note that architecture of the defined model should match with the model whose weights we are loading.

(3) Add the FC classifier on top of the convolutional base.

```
input_model = Input(shape=input_shape)
output_vgg16_conv = model_vgg16_conv(input_model)
prediction = FC_model(output_vgg16_conv)
model = Model(input=input_model, output=prediction)
```

(4) Fine-tuning: Freeze the first 25 layers i.e. set them as untrainable

```
for layer in model.layers[:25]:
    layer.trainable = False
```

(5) Compile the model using a non-adaptive optimizer like SGD with a small learning rate.

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

Taking a step further – Fine-tuning (Precautions)

In order to not wreck the previously learned features of a pre-trained model, here are some of the common practices followed during fine-tuning:

- **Fine-tuning is to be done for the network with properly trained weights**. We cannot put randomly initialized FC layers on top of a trained Convolutional base because large weight updates for the FC layers will spoil the weights of the Conv Base. That's why we first trained the FC layers using bottleneck features of the Conv base and then only fine-tuned it.

- We keep the first few Conv blocks frozen (learning general features) and only fine-tune the last ones (learning specialized features) in a deep network. This is done to prevent overfitting since a large network has high model entropic capacity and thus a strong tendency to overfit.

- Fine-tuning should be done with a very slow learning rate so as to keep the weight updates small. We use non-adaptive learning rate optimizer like SGD instead of RMSProp.

Exercise

Try getting accuracy above 95% (HINTS!!):

- Try more aggressive dropout. (Try increasing dropout probability)
- Use L1 or L2 regularization. (Regularization in Keras can be done per-layer basis. For details, visit https://keras.io/regularizers/)
- Fine-tune one more convolutional block.

Thank You