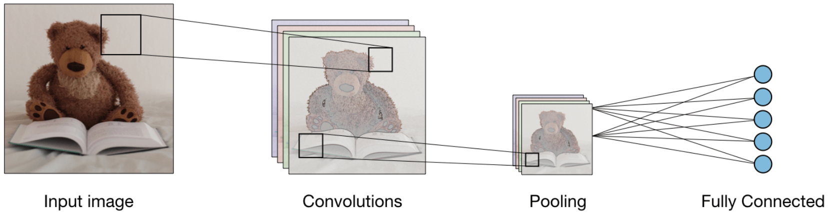
VIP Cheatsheet: Convolutional Neural Networks

Afshine Amidi and Shervine Amidi

November 26, 2018

Overview

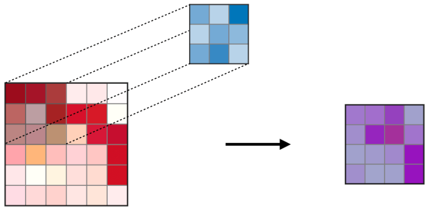
❒ **Architecture of a traditional CNN** – Convolutional neural networks, also known as CNNs, are a specific type of neural networks that are generally composed of the following layers:



The convolution layer and the pooling layer can be fine-tuned with respect to hyperparameters that are described in the next sections.

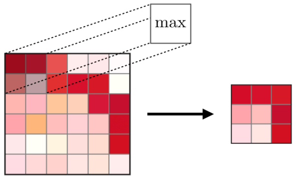
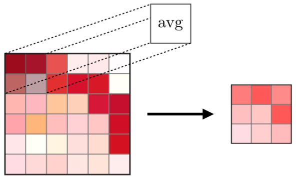
Types of layer

❒ **Convolutional layer (CONV)** – The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input I with respect to its dimensions. Its hyperparameters include the filter size F and stride S. The resulting output O is called *feature map* or *activation map*.

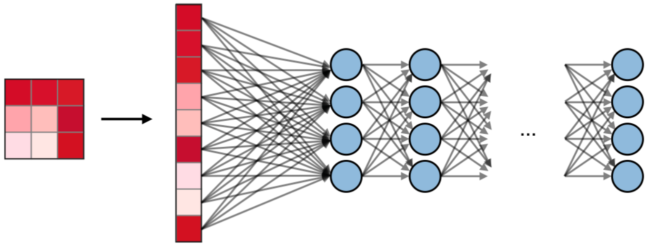


Remark: the convolution step can be generalized to the 1D and 3D cases as well.

❒ **Pooling (POOL)** – The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.

	Max pooling	**Average pooling**
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	- Preserves detected features - Most commonly used	- Downsamples feature map - Used in LeNet

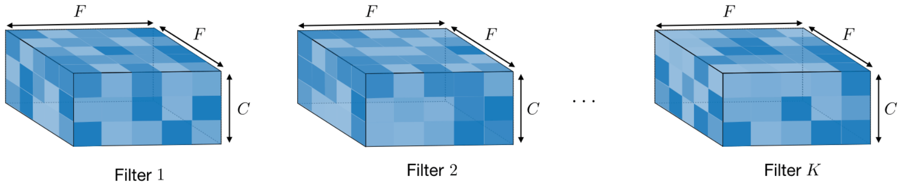
❒ **Fully Connected (FC)** – The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons. If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



Filter hyperparameters

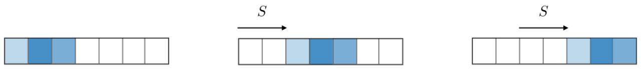
The convolution layer contains filters for which it is important to know the meaning behind its hyperparameters.

❒ **Dimensions of a filter** – A filter of size $F \times F$ applied to an input containing C channels is a $F \times F \times C$ volume that performs convolutions on an input of size $I \times I \times C$ and produces an output feature map (also called activation map) of size $O \times O \times 1$.

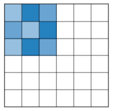
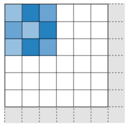
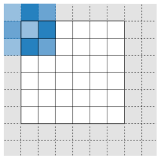


Remark: the application of K filters of size $F \times F$ results in an output feature map of size $O \times O \times K$.

❒ **Stride** – For a convolutional or a pooling operation, the stride S denotes the number of pixels by which the window moves after each operation.



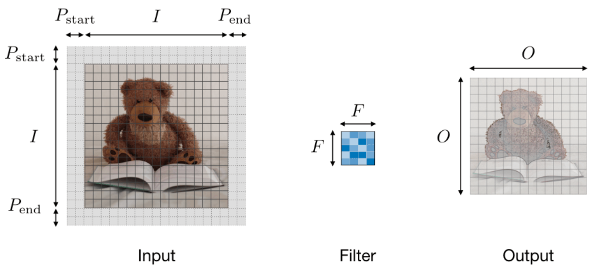
❒ **Zero-padding** – Zero-padding denotes the process of adding P zeroes to each side of the boundaries of the input. This value can either be manually specified or automatically set through one of the three modes detailed below:

	Valid	**Same**	**Full**
Value	$P = 0$	$P_{\textrm{start}} = \left\lfloor\frac{S\lceil\frac{I}{S}\rceil - I + F - S}{2}\right\rfloor$ $P_{\textrm{end}} = \left\lceil\frac{S\lceil\frac{I}{S}\rceil - I + F - S}{2}\right\rceil$	$P_{\textrm{start}} \in [\![0,F-1]\!]$ $P_{\textrm{end}} = F - 1$
Illustration			
Purpose	- No padding - Drops last convolution if dimensions do not match	- Padding such that feature map size has size $\left\lceil\frac{I}{S}\right\rceil$ - Output size is mathematically convenient - Also called 'half' padding	- Maximum padding such that end convolutions are applied on the limits of the input - Filter 'sees' the input end-to-end

Tuning hyperparameters

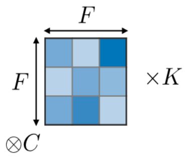
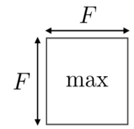
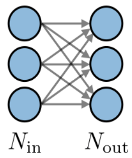
❒ **Parameter compatibility in convolution layer** – By noting I the length of the input volume size, F the length of the filter, P the amount of zero padding, S the stride, then the output size O of the feature map along that dimension is given by:

$$\boxed{O = \frac{I - F + P_{\textrm{start}} + P_{\textrm{end}}}{S} + 1}$$



Remark: often times, $P_{start} = P_{end} \triangleq P$, in which case we can replace $P_{start} + P_{end}$ by $2P$ in the formula above.

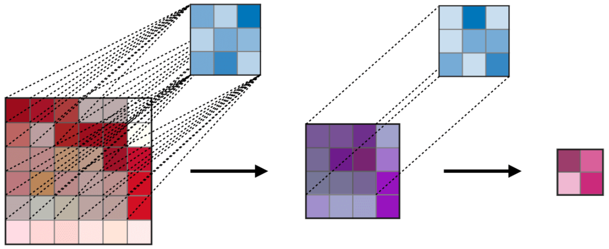
❒ **Understanding the complexity of the model** – In order to assess the complexity of a model, it is often useful to determine the number of parameters that its architecture will have. In a given layer of a convolutional neural network, it is done as follows:

	CONV	**POOL**	**FC**
Illustration			
Input size	$I \times I \times C$	$I \times I \times C$	$N_{\textrm{in}}$
Output size	$O \times O \times K$	$O \times O \times C$	$N_{\textrm{out}}$
Number of parameters	$(F \times F \times C + 1) \cdot K$	0	$(N_{\textrm{in}} + 1) \times N_{\textrm{out}}$
Remarks	- One bias parameter per filter - In most cases, $S < F$ - A common choice for K is $2C$	- Pooling operation done channel-wise - In most cases, $S = F$	- Input is flattened - One bias parameter per neuron - The number of FC neurons is free of structural constraints

❒ **Receptive field** – The receptive field at layer k is the area denoted $R_k \times R_k$ of the input that each pixel of the k-th activation map can 'see'. By calling F_j the filter size of layer j and S_i the stride value of layer i and with the convention $S_0 = 1$, the receptive field at layer k can be computed with the formula:

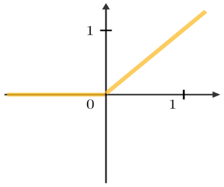
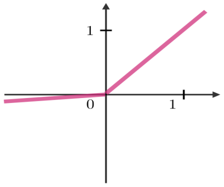
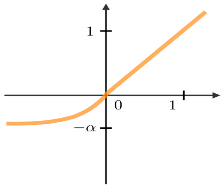
$$\boxed{R_k = 1 + \sum_{j=1}^{k}(F_j - 1)\prod_{i=0}^{j-1} S_i}$$

In the example below, we have $F_1 = F_2 = 3$ and $S_1 = S_2 = 1$, which gives $R_2 = 1+2 \cdot 1+2 \cdot 1 = 5$.



Commonly used activation functions

❒ **Rectified Linear Unit** – The rectified linear unit layer (ReLU) is an activation function g that is used on all elements of the volume. It aims at introducing non-linearities to the network. Its variants are summarized in the table below:

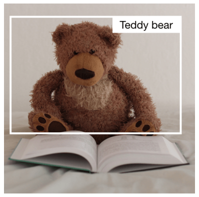
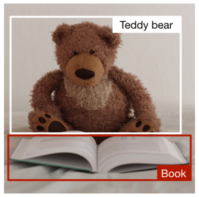
ReLU	Leaky ReLU	ELU
$g(z)=\max(0,z)$	$g(z)=\max(\epsilon z,z)$ with $\epsilon\ll 1$	$g(z)=\max(\alpha(e^z-1),z)$ with $\alpha\ll 1$
		
Non-linearity complexities biologically interpretable	Addresses dying ReLU issue for negative values	Differentiable everywhere

❒ **Softmax** – The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $x\in\mathbb{R}^n$ and outputs a vector of output probability $p\in\mathbb{R}^n$ through a softmax function at the end of the architecture. It is defined as follows:

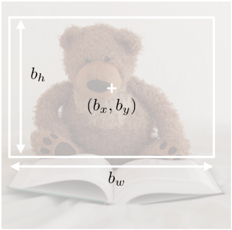
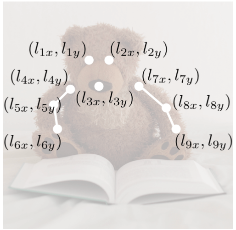
$$\boxed{p=\begin{pmatrix}p_1\\\vdots\\p_n\end{pmatrix}}\quad\text{where}\quad\boxed{p_i=\frac{e^{x_i}}{\displaystyle\sum_{j=1}^n e^{x_j}}}$$

Object detection

❒ **Types of models** – There are 3 main types of object recognition algorithms, for which the nature of what is predicted is different. They are described in the table below:

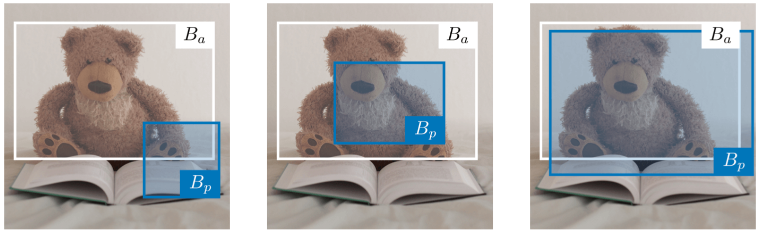
Image classification	Classification w. localization	Detection
		
- Classifies a picture - Predicts probability of object	- Detects object in a picture - Predicts probability of object and where it is located	- Detects up to several objects in a picture - Predicts probabilities of objects and where they are located
Traditional CNN	Simplified YOLO, R-CNN	YOLO, R-CNN

❒ **Detection** – In the context of object detection, different methods are used depending on whether we just want to locate the object or detect a more complex shape in the image. The two main ones are summed up in the table below:

Bounding box detection	Landmark detection
Detects the part of the image where the object is located	- Detects a shape or characteristics of an object (e.g. eyes) - More granular
	 (l_{1x},l_{1y}) (l_{2x},l_{2y}) (l_{4x},l_{4y}) (l_{7x},l_{7y}) (l_{5x},l_{5y}) (l_{3x},l_{3y}) (l_{8x},l_{8y}) (l_{6x},l_{6y}) (l_{9x},l_{9y})
Box of center (b_x,b_y), height b_h and width b_w	Reference points (l_{1x},l_{1y}), ...,(l_{nx},l_{ny})

❒ **Intersection over Union** – Intersection over Union, also known as IoU, is a function that quantifies how correctly positioned a predicted bounding box B_p is over the actual bounding box B_a. It is defined as:

$$\boxed{\textrm{IoU}(B_p,B_a)=\frac{B_p\cap B_a}{B_p\cup B_a}}$$



$\textrm{IoU}(B_p,B_a)=0.1$ $\textrm{IoU}(B_p,B_a)=0.5$ $\textrm{IoU}(B_p,B_a)=0.9$

Remark: we always have $IoU\in[0,1]$. By convention, a predicted bounding box B_p is considered as being reasonably good if $IoU(B_p,B_a)\geqslant 0.5$.

❒ **Anchor boxes** – Anchor boxing is a technique used to predict overlapping bounding boxes. In practice, the network is allowed to predict more than one box simultaneously, where each box prediction is constrained to have a given set of geometrical properties. For instance, the first prediction can potentially be a rectangular box of a given form, while the second will be another rectangular box of a different geometrical form.

❒ **Non-max suppression** – The non-max suppression technique aims at removing duplicate overlapping bounding boxes of a same object by selecting the most representative ones. After having removed all boxes having a probability prediction lower than 0.6, the following steps are repeated while there are boxes remaining:

- Step 1: Pick the box with the largest prediction probability.
- Step 2: Discard any box having an IoU $\geqslant 0.5$ with the previous box.

1 Convolutional Neural Networks

1.0.1 Introduction

CNNs have certain features that identify patterns in images because of "convolution operation" including:

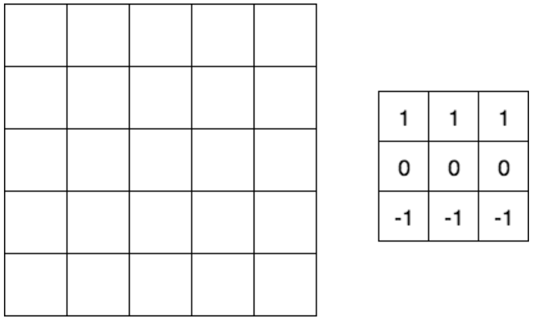
- Dense layers learn global patterns in their input feature space
- Convolution layers learn local patterns, and this leads to the following interesting features:
    - Unlike with densely connected networks, when a convolutional neural network recognizes a patterns let's say, in the upper-right corner of a picture, it can recognize it anywhere else in a picture.
    - Deeper convolutional neural networks can learn spatial hierarchies. A first layer will learn small local patterns, a second layer will learn larger patterns using features of the first layer patterns, etc.

Because of these properties, CNNs are great for tasks like:

- Image classification
- Object detection in images
- Picture neural style transfer

1.0.2 The Convolution Operation

The idea behind the convolutional operation is to detect complex building blocks, or features, that can aid in the larger task such as image recognition. For example, we'll detect vertical or horizontal edges present in the image. Below is an example of a horizontal edge detection.



1	1	1
0	0	0
-1	-1	-1

This is a simplified 5 x 5 pixel image (greyscale!). You use a so-called "filter" (denoted on the right) to perform a convolution operation. This particular filter operation will detect horizontal edges. The matrix in the left should have number in it (from 1-255, or let's assume we rescaled it to number 1-10). The output is a 3 x 3 matrix. (*This example is for computational clarity, no clear edges*)

1.0.3 Padding

There are some issues with using filters on images including:

- The image shrinks with each convolution layer: you're throwing away information in each layer! For example:
    - Starting from a 5 x 5 matrix, and using a 3 x 3 matrix, you end up with a 3 x 3 image.
    - Starting from a 10 x 10 matrix, and using a 3 x 3 matrix, you end up with a 8 x 8 image.
    - etc.
- The pixels around the edges are used much less in the outputs due to the filter.

For example, if we apply 3x3 filters to a 5x5 image, our original 5x5 image contains 25 pixels, but tiling our 3x3 filter only has 9 possible locations.

1.0.4 Stride Convolutions

Another method to change the output of your convolutions is to change the stride. The stride is how the convolution filter is moved over the original image. In our above example, we moved the filter one pixel to the right starting from the upper left hand corner, and then began to do this again after moving the filter one pixel down. Alternatively, by changing the stride, we could move our filter by 2 pixels each time, resulting in a smaller number of possible locations for the filter.

Strided convolutions are rarely used in practice but a good feature to be aware of for some models.

1.0.5 Pooling Layer

The last element in a CNN architecture (before fully connected layers as we have previously discussed in other neural networks) is the pooling layer. This layer is meant to substantially downsample the previous convolutional layers. The idea behind this is that the previous convolutional layers will find patterns such as edges or other basic shapes present in the pictures. From there, pooling layers such as Max pooling (the most common) will take a summary of the convolutions from a larger section. In practice Max pooling (taking the max of all convolutions from a larger area of the original image) works better then average pooling as we are typically looking to detect whether a feature is present in that region. Downsampling is essential in order to produce viable execution times in the model training.

Max pooling has some important hyperparameters:

- f (filter size)
- S (stride)

Common hyperparameters include: f=2, s=2 and f=3, s=2, this shrinks the size of the representations. If a feature is detected anywhere in the quadrants, a high number will appear. so max pooling preserves this feature.

1.0.6 Fully Connected Layers

Once you have addded a number of convolutional layers and pooling layers, you add fully connected (dense) layers as we did before in previous neural network models. This now allows the network to learn a final decision function based on these transformed informative inputs generating from the convolutional and pooling layers.

1.0.7 Libraries For Directory Organization

In [ ]:

```
# used for creating folders and moving files
import os, shutil
```

1.0.7.1 Creating Folders

In [ ]:

```
new_dir = 'split/'
os.mkdir(new_dir)
train_folder = os.path.join(new_dir, 'train')
train_santa = os.path.join(train_folder, 'santa')
train_not_santa = os.path.join(train_folder, 'not_santa')
```

1.0.7.2 Moving Files

In [ ]:

```
# train santa
imgs = imgs_santa[:271]
for img in imgs:
    origin = os.path.join(data_santa_dir, img)
    destination = os.path.join(train_santa, img)
    shutil.copyfile(origin, destination)
```

1.0.8 Libraries for Keras / Densely Connected Network

In [ ]:

```
import matplotlib.pyplot as plt
import scipy
import numpy as np
from PIL import Image
from scipy import ndimage
from keras.preprocessing.image import ImageDataGenerator, array_to_img
from keras.preprocessing.image import img_to_array, load_img
from keras import models
from keras import layers
```

1.0.8.1 Example Image Reshape

In [ ]:

```
# CONVERT PIXEL VALUES FROM (0,255) TO (0,1) BY DIVIDING BY 255
test_generator = ImageDataGenerator(rescale=1./255).flow_from_directory(
        test_folder,
        target_size=(64, 64), batch_size = 180)
```

1.0.8.2 Creating Data Sets (via ImageDataGenerator)

In [ ]:

```
# create the data sets
train_images, train_labels = next(train_generator)
```

1.0.8.3 Reshape Images & Labels

In [ ]:

```
train_img = train_images.reshape(train_images.shape[0], -1)
# train_img.shape => (542,12288)

# uses shape of train_img (length of inputs - 542)
train_y = np.reshape(train_labels[:,0], (542,1))
```

1.0.8.4 Model Initialization, Compiling, & Fitting

In [ ]:

```
model = models.Sequential()
model.add(layers.Dense(20, activation='relu', input_shape=(12288,))) #2 hic
model.add(layers.Dense(7, activation='relu'))
model.add(layers.Dense(5, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

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

histoire = model.fit(train_img,
                    train_y,
                    epochs=50,
                    batch_size=32,
                    validation_data=(val_img, val_y))
```

1.0.8.5 Inspecting History Dictionary

In [ ]:

```
results_train = model.evaluate(train_img, train_y)
results_test = model.evaluate(test_img, test_y)
```

1.0.8.6 View Results

In [ ]:

```
results_train, results_test
# [0.18314, 0.9742],  [0.3234, 0.8424]
```

1.0.9 Convnet

1.0.9.1 Model Initialization, Compiling, & Fitting

In [ ]:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(64 ,64,  3)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(32, (4, 4), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer="sgd",
              metrics=['acc'])

history = model.fit(train_images,
                    train_y,
                    epochs=30,
                    batch_size=32,
                    validation_data=(val_images, val_y))
```

1.0.10 Data Augmentation

Arguments Documentation: https://keras.io/preprocessing/image/ (https://keras.io/preprocessing/image/)

- `rotation_range` (degree, 0-
- `width_shift_range` (fraction if < 1, pixels if > = 1)
- `height_shift_range` (fraction if < 1, pixels if > = 1)
- `shear_range` (float)
- `zoom_range` ((float)
- `horizontal_flip` (boolean)
- `fill_mode` (string)

1.0.10.1 Example using `.flow_from_directory(directory)`

In [ ]:

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        'data/train',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        'data/validation',
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')

history = model.fit_generator(
        train_generator,
        steps_per_epoch=2000,
        epochs=50,
        validation_data=validation_generator,
        validation_steps=800)
```

1.0.10.2 Example Accuracy Plot

In [ ]:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(acc))
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

1.0.10.3 Model Evaluation

In [ ]:

```
test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

1.0.11 Visualizing Intermediate Activations

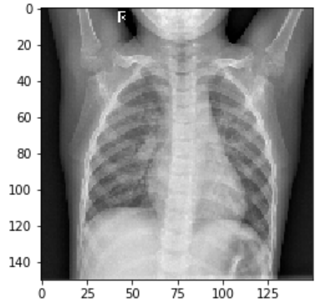
1.0.11.1 Loading Model & Loading Image From Training Set

In [ ]:

```
from keras.models import load_model

model = load_model('chest_xray_all_data.h5')
model.summary()  # As a reminder.
```

In [ ]:

```
from keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

img_path = 'IM-0115-0001.jpeg'


img = image.load_img(img_path, target_size=(150, 150))
img_tensor = image.img_to_array(img)
img_tensor = np.expand_dims(img_tensor, axis=0)

#Follow the Original Model Preprocessing
img_tensor /= 255.

#Check tensor shape
print(img_tensor.shape)

#Preview an image
plt.imshow(img_tensor[0])
plt.show()
```



1.0.11.2 Visualizing A Layer

In [ ]:

```
from keras import models
# Extract model layer outputs
layer_outputs = [layer.output for layer in model.layers[:8]]

# Rather then a model with a single output
# we are going to make a model to display the feature maps
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

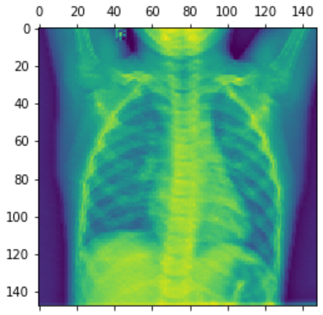
In [ ]:

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856     
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584    
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776   
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 513       
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

The initial two layers output feature maps that have 32 channels each. You can visualize each of these channels individually by slicing the tensor along that axis. Subsequentially, the next two layers have 64 channels each and the 5th through 8th layers have 128 channels each. Recall that this allows the CNN to detect successively more abstract patterns. Here's what slicing one of these feature maps and visualizing an individual channel looks like in practice:

In [ ]:
```
# Returns an array for each activation layer
activations = activation_model.predict(img_tensor)

first_layer_activation = activations[0]
print(first_layer_activation.shape)

#We slice the third channel and preview the results
plt.matshow(first_layer_activation[0, :, :, 3], cmap='viridis')
plt.show()
```



1.0.11.3 Example Visualizing Multiple Layers

In [ ]:
```
fig, axes = plt.subplots(8, 4, figsize=(12,24))
for i in range(32):
    row = i//4
    column = i%4
    ax = axes[row, column]
    first_layer_activation = activations[0]
    ax.matshow(first_layer_activation[0, :, :, i], cmap='viridis')
```

1.0.12 Using Pretrained Networks

1.0.12.1 Examples

Keras has several pretrained models available. Here is a list of pretrained image classification models. All these models are available in `keras.applications` and were pretrained on the ImageNet dataset, a data set with 1.4 Million labeled images and 1,000 different classes.

- DenseNet
- InceptionResNetV2
- InceptionV3
- MobileNet
- NASNet
- ResNet50
- VGG16
- VGG19
- Xception

You can find an overview here too: https://keras.io/applications/ (https://keras.io/applications/)

For each of these pretrained models, you can look at their structure. You can simply import the desired pretrained model, and use it as a function with 2 arguments: `weights` and `include_top`. Use `"imagenet"` in weights in order to use the weights that were obtained when training on the ImageNet data set. You can chose to iclude the top of the model (whether or not to include the fully-connected layer at the top of the network), through the argument `include_top` . Here, we'll have a look at the structure of the MobileNet neural network.

In [ ]:

```
from keras.applications import MobileNet
conv_base = MobileNet(weights='imagenet',
                  include_top = True)
```

1.0.12.2 Feature Extraction & Fine Tuning

- **Feature extraction**: here, you use the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.
- **Fine-tuning**: when finetuning, you'll "unfreeze" a few top layers from the model and train them again together with the densely connected classifier. Note that you are changing the parts of the convolutional layers here that were used to detect the more abstract features. By doing this, you can make your model more relevant for the classification problem at hand.

1.0.13 VGG19: Feature Extraction (Pretrained Networks)

In [ ]:

```
from keras.preprocessing.image import ImageDataGenerator, array_to_img
from keras.applications import VGG19
cnn_base = VGG19(weights='imagenet',
                  include_top=False,
                  input_shape=(64, 64, 3))
```

1.0.13.1 Extract Features Method

In [ ]:

```
def extract_features(directory, sample_amount):
    features = np.zeros(shape=(sample_amount, 2, 2, 512))
    labels = np.zeros(shape=(sample_amount))
    generator = datagen.flow_from_directory(
        directory, target_size=(64, 64),
        batch_size = 10,
        class_mode='binary')
    i=0
    for inputs_batch, labels_batch in generator:
        features_batch = cnn_base.predict(inputs_batch)
        features[i * batch_size : (i + 1) * batch_size] = features_batch
        labels[i * batch_size : (i + 1) * batch_size] = labels_batch
        i = i + 1
        if i * batch_size >= sample_amount:
            break
    return features, labels
```

In [ ]:

```
# you should be able to divide sample_amount by batch_size!!
train_features, train_labels = extract_features(train_folder, 540)
validation_features, validation_labels = extract_features(val_folder, 200)
test_features, test_labels = extract_features(test_folder, 180)

train_features = np.reshape(train_features, (540, 2 * 2 * 512))
validation_features = np.reshape(validation_features, (200, 2 * 2 * 512))
test_features = np.reshape(test_features, (180, 2 * 2 * 512))
```

1.0.13.2 Model Initialization, Compiling, & Fitting

In [ ]:

```
from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=2 * 2 * 512))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(train_features, train_labels,
                    epochs=20,
                    batch_size=10,
                    validation_data=(validation_features, validation_labels

results_test = model.evaluate(test_features, test_labels)
results_test

# ACCURACY PLOT
train_acc = history.history['acc']
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
epch = range(1, len(train_acc) + 1)
plt.plot(epch, train_acc, 'g.', label='Training Accuracy')
plt.plot(epch, val_acc, 'g', label='Validation acc')
plt.title('Accuracy')
plt.legend()
plt.figure()
plt.plot(epch, train_loss, 'r.', label='Training loss')
plt.plot(epch, val_loss, 'r', label='Validation loss')
plt.title('Loss')
plt.legend()
plt.show()
```

1.0.14 Feature Extraction (Freezing)

```
Here's an overview of the process:
* Add the pretrained model as the first layer
* Add some dense layers for a classifier on top
* Freeze the convolutional base
* Train the model

Layer freezing means that all of the weights associated with that layer(s)
will remain unchanged through the optimization process. Freezing the base
is important as we wish to preserve the features encoded in this CNN base.
```

1.0.14.1 Initial Model (CNN Base)

In [ ]:

```
from keras.applications import VGG19
cnn_base = VGG19(weights='imagenet',
                  include_top=False,
                  input_shape=(64, 64, 3))

model = models.Sequential()
model.add(cnn_base)
model.add(layers.Flatten())
model.add(layers.Dense(132, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

1.0.14.2 Inspecting If Layer Is Frozen/Trainable

In [ ]:

```
#You can check whether a layer is trainable (or alter its setting) through
for layer in model.layers:
    print(layer.name, layer.trainable)

#Similarly, we can check how many trainable weights are in the model:
print(len(model.trainable_weights))
```

1.0.14.3 Freeze CNN Base Layer

In [ ]:

```
cnn_base.trainable = False
```

1.0.14.4 Train, Compile, & Fit Model (Same As Previous Examples)

In [ ]:

```
# get all the data in the directory split/train (542 images), and reshape t
train_datagen = 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')

train_generator = train_datagen.flow_from_directory(
        train_folder,
        target_size=(64, 64),
        batch_size= 20,
        class_mode= 'binary')

# get all the data in the directory split/validation (200 images), and resh
val_generator = ImageDataGenerator(rescale=1./255).flow_from_directory(
        val_folder,
        target_size=(64, 64),
        batch_size = 20,
        class_mode= 'binary')

# get all the data in the directory split/test (180 images), and reshape th
test_generator = ImageDataGenerator(rescale=1./255).flow_from_directory(
        test_folder,
        target_size=(64, 64),
        batch_size = 180,
        class_mode= 'binary')

test_images, test_labels = next(test_generator)

## MODEL COMPILE
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])

## MODEL FIT
history = model.fit_generator(
              train_generator,
              steps_per_epoch= 27,
              epochs = 10,
              validation_data = val_generator,
              validation_steps = 10)

## ACCURACY PLOT
train_acc = history.history['acc']
val_acc = history.history['val_acc']
train_loss = history.history['loss']
val_loss = history.history['val_loss']
epch = range(1, len(train_acc) + 1)
plt.plot(epch, train_acc, 'g.', label='Training Accuracy')
plt.plot(epch, val_acc, 'g', label='Validation acc')
plt.title('Accuracy')
plt.legend()
plt.figure()
plt.plot(epch, train_loss, 'r.', label='Training loss')
plt.plot(epch, val_loss, 'r', label='Validation loss')
plt.title('Loss')
plt.legend()
plt.show()
```

1.0.15 Fine Tuning

Fine tuning starts with the same procedure that we have demonstrated for feature extraction. From there, we further fine-tune the weights of the most abstract layers of the convolutional base.

When fine-tuning these layers from the convolutional base, it is essential that you first freeze the entire convolutional base and train a classifier as we discussed with the feature engineering technique above. Without this, when gradient descent is initialized to optimize our loss function, we would be apt to loose any significant patterns learned by the original classifier that we are adapting to our current situation. As a result, we must first tune the fully-connected classifier that sits on top of the pretrained convolutional base. From there, our model should have a relatively strong accuracy and we can fine tune the weights of the last few layers of the convolutional base. Unfreezing initial layers of the convolutional base is not apt to produce substantial gains as these early layers typically learn simple representations such as colors and edges which are typically useful in all forms of image recognition, regardless of application.

1.0.15.1 Unfreeze Base Layer

In [ ]:

```
cnn_base.trainable = True
```

1.0.15.2 Refreeze Layers (Unfreezing Final Block Of Layers)

In [ ]:

```
cnn_base.trainable = True
set_trainable = False
for layer in cnn_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
```

1.0.15.3 Model Compile & Fit as Before (see above)

1.0.15.4 Evaluation (Test Set)

In [ ]:

```
# test_generator = test_datagen.flow_from_directory(
#         test_dir,
#         target_size=(150, 150),
#         batch_size=20,
#         class_mode='binary')
test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```