

APPLICATIONS



OF DATA SCIENCE

Convolutional Neural Networks

Applications of Data Science - Class 17

Giora Simchoni

gsimchoni@gmail.com and add #dsapps in subject

Stat. and OR Department, TAU

2020-06-12

APPLICATIONS



OF DATA SCIENCE

Understanding Images

APPLICATIONS

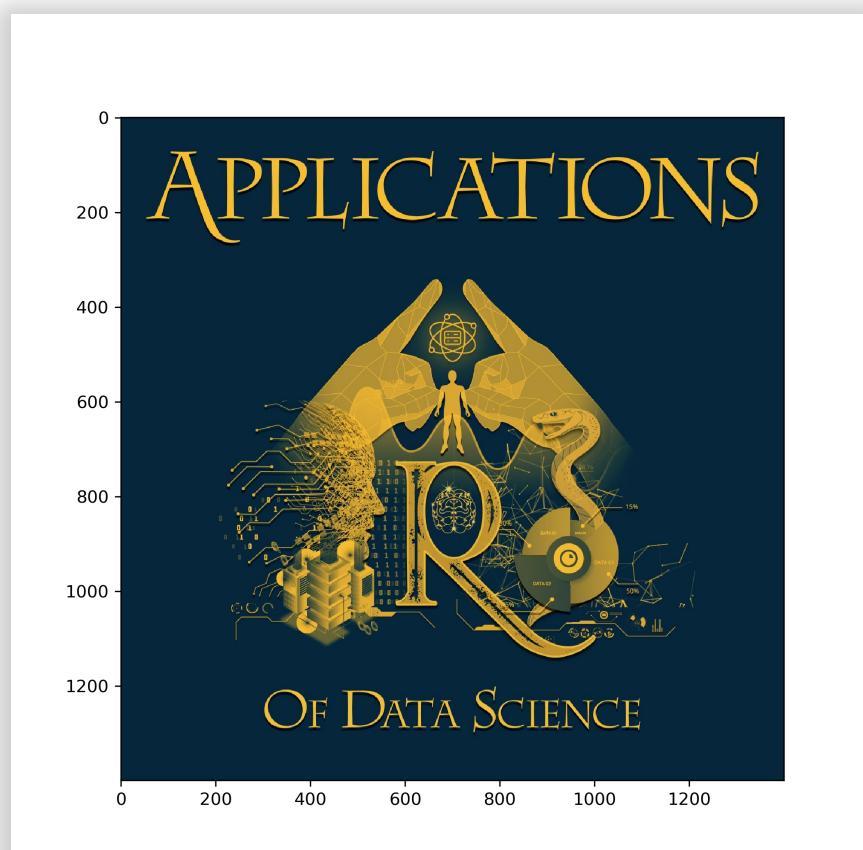


OF DATA SCIENCE

```
import matplotlib.pyplot as plt
from matplotlib.image import imread

logo = imread('../DSApps_logo.jpg')

plt.imshow(logo)
plt.show()
```



```
print(type(logo))

## <class 'numpy.ndarray'>

print(logo.shape)

## (1400, 1400, 3)

print(logo[:4, :4, 0])

## [[6 6 6 6]
##  [6 6 6 6]
##  [6 6 6 6]
##  [6 6 6 6]]

print(logo.min(), logo.max())

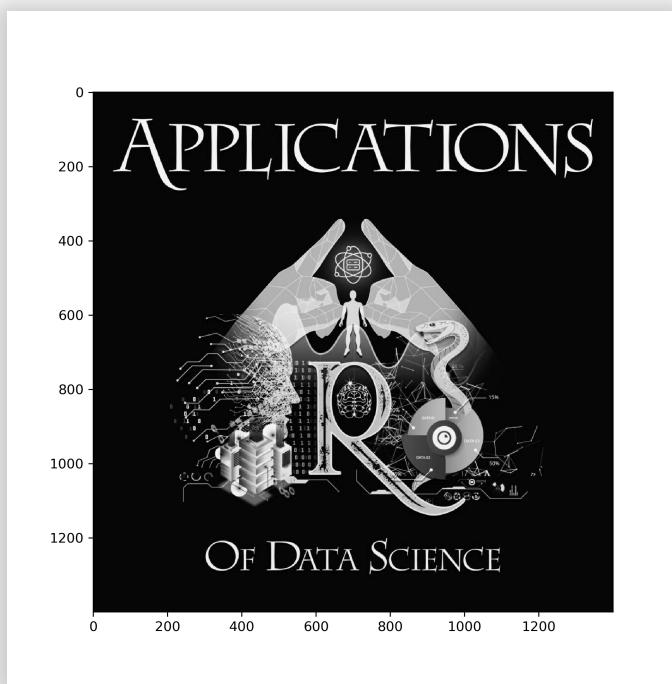
## 0 255

print(logo.dtype, logo.size * logo.itemsize)

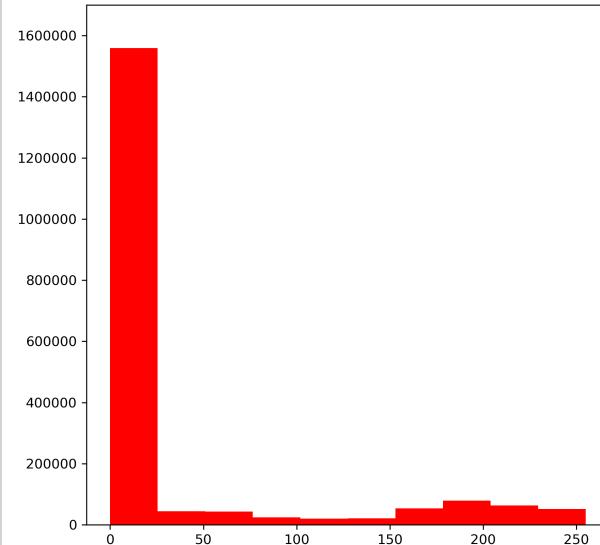
## uint8 5880000
```

Red channel

```
red = logo[:, :, 0]
plt.imshow(red,
           cmap='gray')
plt.show()
```

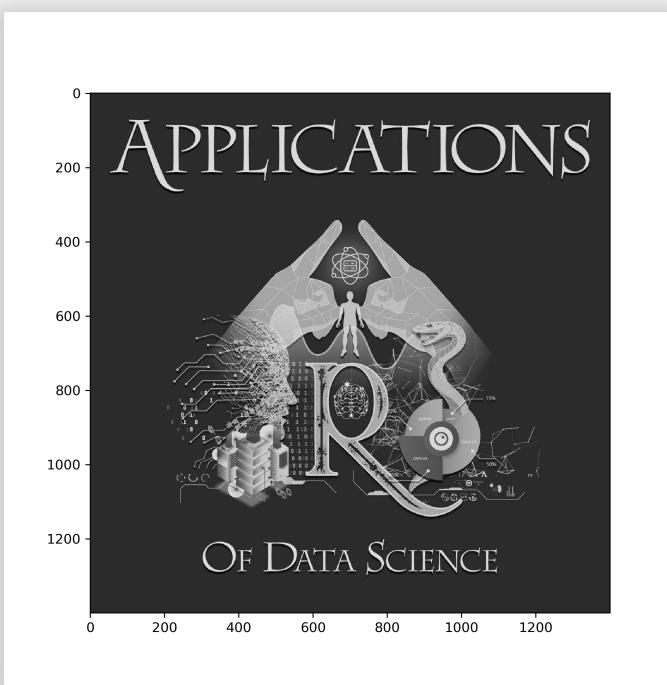


```
fig = plt.hist(red.flatten(),
               color = 'r')
fig = plt.ylim(0, 1700000)
plt.show()
```

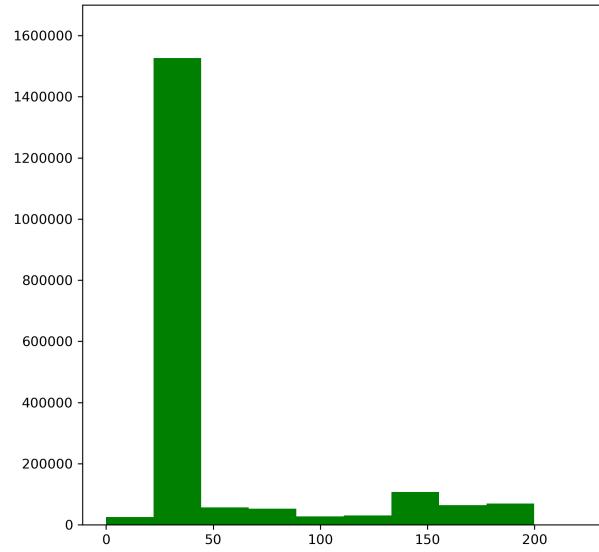


Green channel

```
green = logo[:, :, 1]
plt.imshow(green,
           cmap='gray')
plt.show()
```

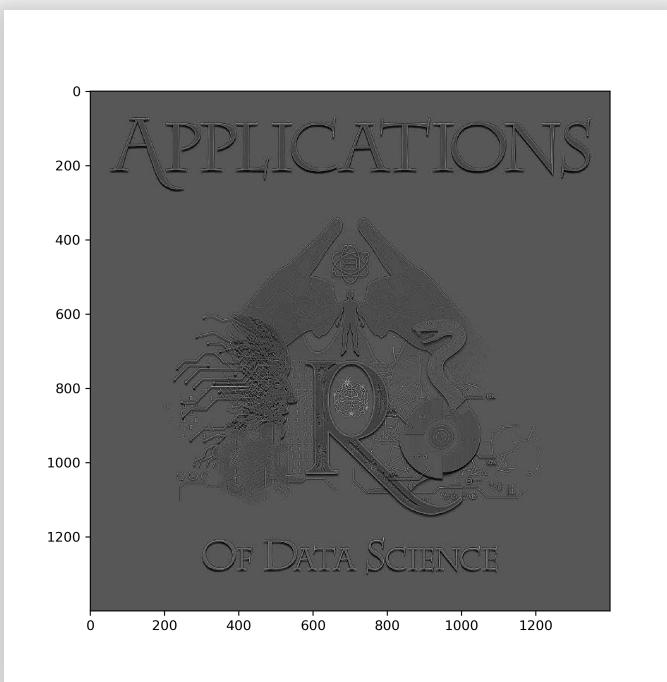


```
fig = plt.hist(green.flatten(),
               color = 'g')
fig = plt.ylim(0, 1700000)
plt.show()
```

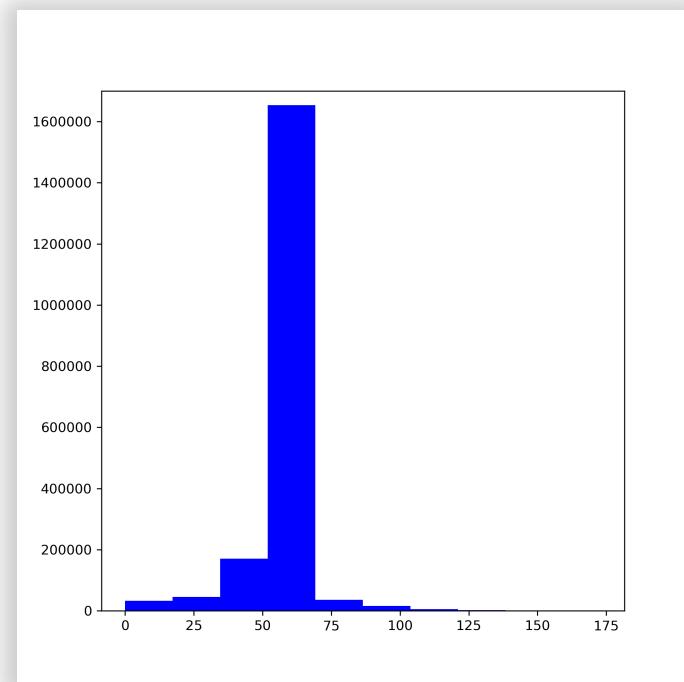


Blue channel

```
blue = logo[:, :, 2]
plt.imshow(blue,
           cmap='gray')
plt.show()
```



```
fig = plt.hist(blue.flatten(),
               color = 'b')
fig = plt.ylim(0, 1700000)
plt.show()
```



Until Convolutional Networks

💡 So how many features is that? 😊 😊 😊

- In 1998 LeCun et. al. published [LeNet-5](#) for digit recognition
- But it wasn't until 2012 when Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton published [AlexNet](#) when the Deep Learning mania really took off.

Until then:

1. Treat it as a regular high-dimensional ML problem
2. Image feature engineering

Convolutional Layer (2D)

APPLICATIONS



OF DATA SCIENCE

What is Convolution?

Convolution

From Wikipedia, the free encyclopedia

For the usage in formal language theory, see [Convolution \(computer science\)](#). For other uses, see [Convolute](#).

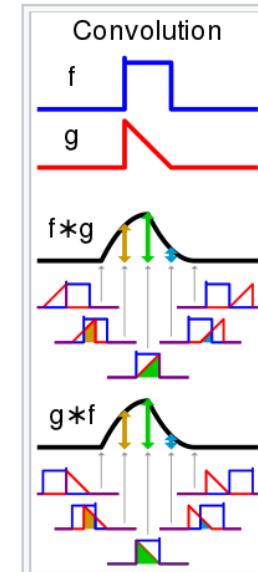
In mathematics (in particular, [functional analysis](#)) **convolution** is a [mathematical operation](#) on two functions (f and g) that produces a third function expressing how the shape of one is modified by the other. The term *convolution* refers to both the result function and to the process of computing it. It is defined as the [integral](#) of the product of the two functions after one is reversed and shifted. And the integral is evaluated for all values of shift, producing the convolution function.

Some features of convolution are similar to [cross-correlation](#): for real-valued functions, of a continuous or discrete variable, it differs from cross-correlation only in that either $f(x)$ or $g(x)$ is reflected about the y -axis; thus it is a cross-correlation of $f(x)$ and $g(-x)$, or $f(-x)$ and $g(x)$.^[A] For continuous functions, the cross-correlation operator is the [adjoint](#) of the convolution operator.

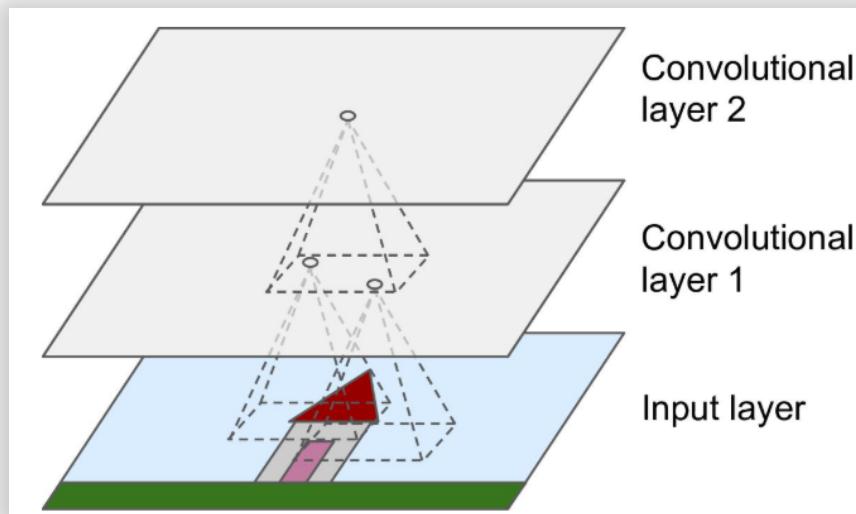
Convolution has applications that include [probability](#), [statistics](#), [computer vision](#), [natural language processing](#), [image](#) and [signal processing](#), [engineering](#), and [differential equations](#).^[citation needed]

The convolution can be defined for functions on [Euclidean space](#), and other [groups](#).^[citation needed]

For example, [periodic functions](#), such as the [discrete-time Fourier transform](#), can be defined on a



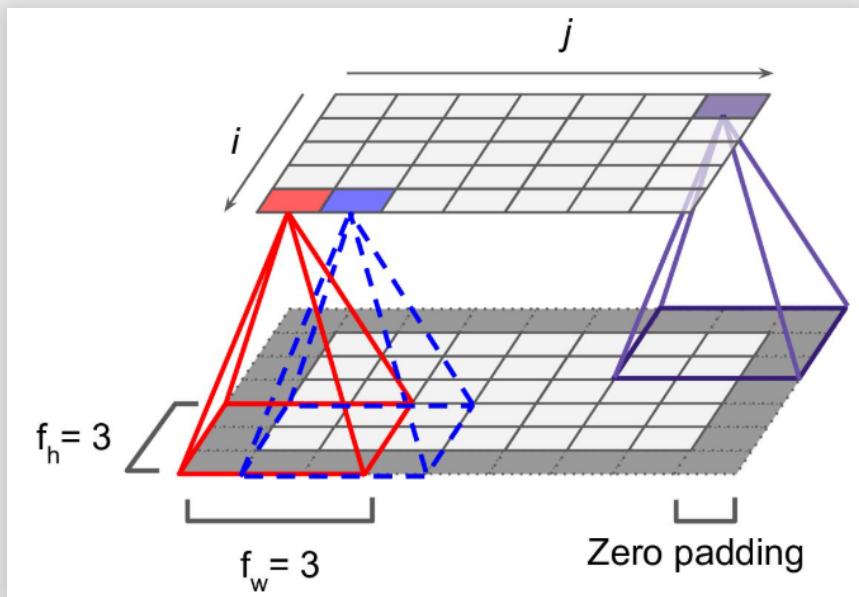
The Convolutional Layer



Source: [Geron 2019](#)

- First layer: every neuron has a "receptive field", it is focused on a specific rectangle of the image, usually 2×2 , 3×3
- Second layer: every neuron has a receptive field in the first layer
- Etc.

More specifically



Source: [Geron 2019](#)

- The $[i, j]$ neuron looks at the rectangle at rows i to $i + f_h - 1$, columns j to $j + f_w - 1$
- Zero Padding: adds 0s around the image to make the next layer the same dimensions

Filters/Features/Kernels

But what does the neuron actually *do*?

All neurons in a layer learn a single *filter* the size of their receptive field, the f_h, f_w rectangle.

Suppose $f_h = f_w = 3$ and the first layer learned the W_{3x3} filter:

```
W = np.array(  
    [  
        [0, 1, 0],  
        [0, 1, 0],  
        [0, 1, 0]  
    ]  
)
```

X is the 5x5 image, suppose it has a single color channel (i.e. grayscale), sort of a smiley:

```
x = np.array(  
    [  
        [0, 1, 0, 1, 0],  
        [0, 1, 0, 1, 0],  
        [0, 0, 0, 0, 0],  
        [1, 0, 0, 0, 1],  
        [0, 1, 1, 1, 0]  
    ]  
)
```

Each neuron in the new layer Z would be the sum of elementwise multiplication of all 3x3 pixels/neurons in its receptive field with W_{3x3} :

$$Z_{i,j} = b + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} X_{i+u,j+v} W_{u,v}$$

Good God, Excel?

| M1 | <code>=A1*\$I\$1+B1*\$J\$1+C1*\$K\$1+A2*\$I\$2+B2*\$J\$2+C2*\$K\$2+A3*\$I\$3+B3*\$J\$3+C3*\$K\$3</code> | | | | | | | | | | | | | | | | | | | | |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | | 0 | 2 | 0 | 2 | 0 | | | | |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | 0 | 1 | 0 | | 0 | 2 | 0 | 2 | 0 | | | | |
| 3 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | 0 | 1 | 0 | | 1 | 1 | 0 | 1 | 1 | | | | |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | 1 | 1 | 1 | 1 | 1 | | | | |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | | | | | | 1 | 1 | 1 | 1 | 1 | | | | |
| 6 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | | | | | | | | | | | | | | |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | | | | | | | |
| 9 | | | X | | | | | | W | | | | Z | | | | | | | | |
| 10 | | | | | | | | | | | | | | | | | | | | | |

💡 What low-level feature did this layer learn to look for? What pattern will make its neurons most positive (i.e. will "turn them on")?

Another filter

| 04 | =C4*\$I\$1+D4*\$J\$1+E4*\$K\$1+C5*\$I\$2+D5*\$J\$2+E5*\$K\$2+C6*\$I\$3+D6*\$J\$3+E6*\$K\$3 | | | | | | | | | | | |
|----|--|---|---|---|---|---|--|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 | 1 | 0 | | 1 | 1 | 1 | 1 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | 1 | 2 |
| 5 | 0 | 1 | 0 | 0 | 0 | 1 | | | | | 0 | 0 |
| 6 | 0 | 0 | 1 | 1 | 1 | 0 | | | | | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | 0 | 0 |
| 8 | | | | | | | | | | | | |
| 9 | X | | | | W | | | Z | | | | |
| 10 | | | | | | | | | | | | |

💡 What low-level feature did this layer learn to look for? What pattern will make its neurons most positive (i.e. will "turn them on")?

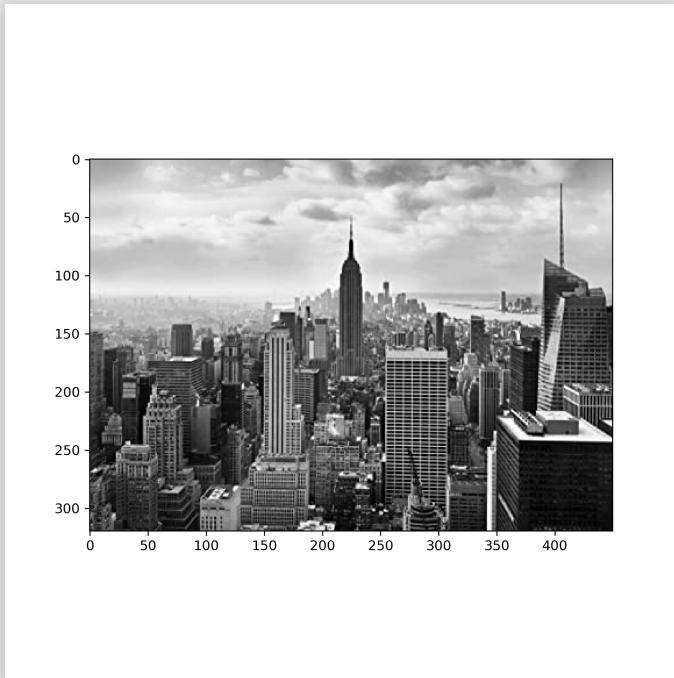
Convolving with Tensorflow

```
import tensorflow as tf

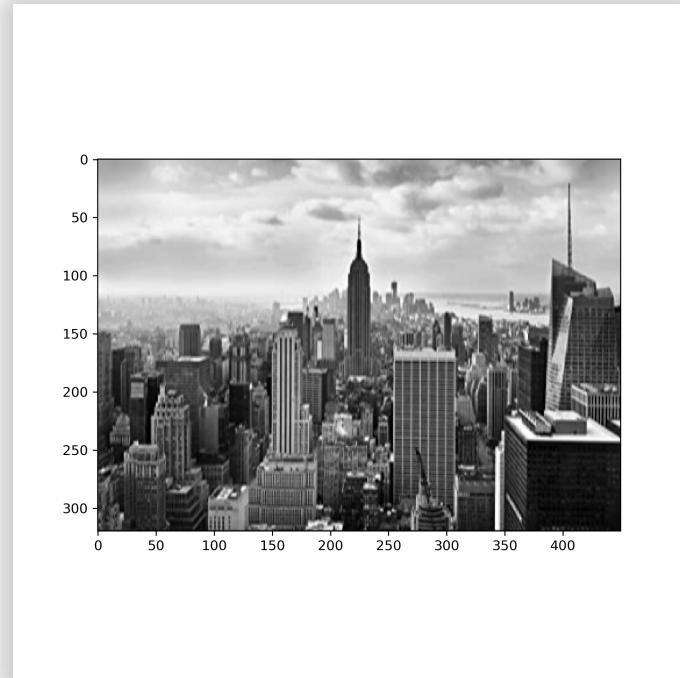
ny = imread('images/new_york.jpg').mean(axis=2)
ny4D = np.array([ny.reshape(ny.shape[0], ny.shape[1], 1)])
W4d = W.reshape(W.shape[0], W.shape[0], 1, 1)

ny_convolved = tf.nn.conv2d(ny4D, W4d, strides=1, padding='SAME')
```

```
plt.imshow(  
    ny,  
    cmap = 'gray')  
plt.show()
```

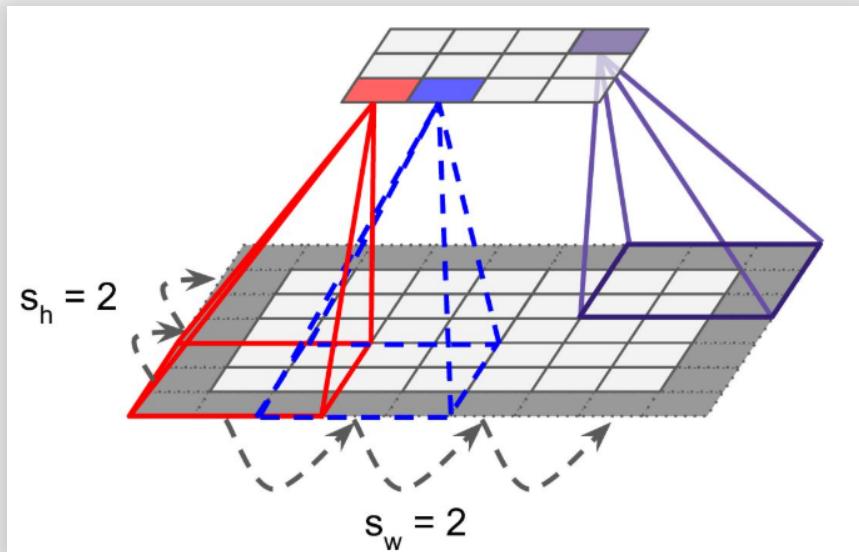


```
plt.imshow(  
    ny_convolved[0, :, :, 0],  
    cmap = 'gray')  
plt.show()
```



Strides

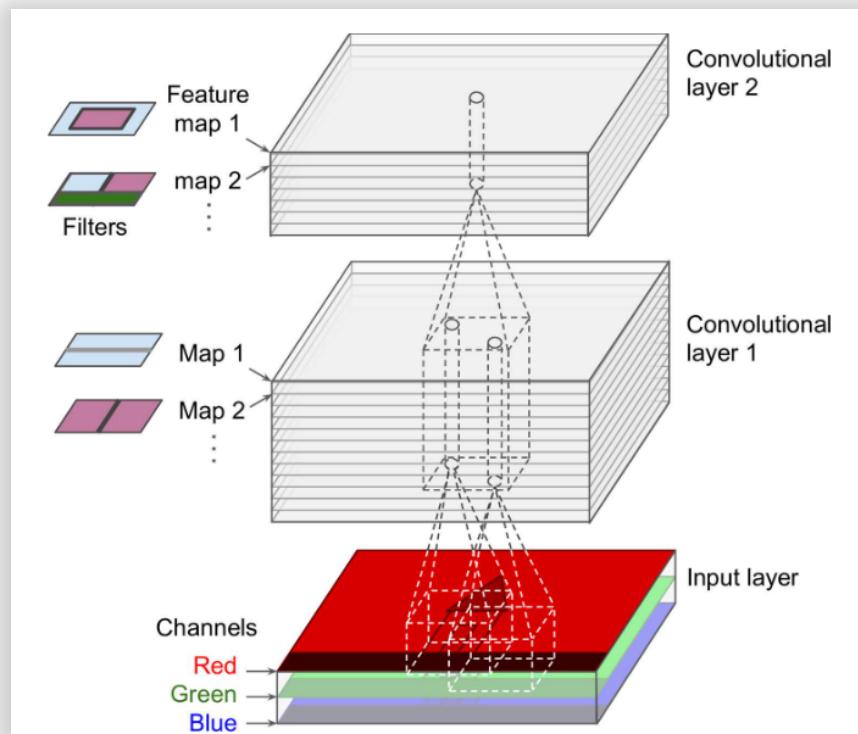
In many CNN architectures layers tend to get smaller and smaller using *strides*:



- The $[i, j]$ neuron looks at the rectangle at rows $i \cdot s_h$ to $i \cdot s_h + f_h - 1$, columns $j \cdot s_w$ to $j \cdot s_w + f_w - 1$

Stacking Feature Maps

A convolutional layer is actually a 3D stack of a few of the 2D layers we described (a.k.a *Feature Maps*), each learns a single filter.



- Feature map 1 learns horizontal lines
- Feature map 2 learns vertical lines
- Feature map 3 learns diagonal lines
- Etc.

And each such feature map takes as inputs all feature maps (or color channels) in the previous layer, and sums:

$$Z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_n-1} X_{i \cdot s_h + u, j \cdot s_w + v, k'} \cdot W_{u,v,k',k}$$

Where f_n is the number of feature maps (or color channels) in the previous layer.

- Feature map 1 with $f_h \cdot f_w$ filter $\cdot f_n$ color channels + 1 bias term (it's a filter *cube*!)
- Feature map 2 with $f_h \cdot f_w$ filter $\cdot f_n$ color channels + 1 bias term
- Etc.

And how does the network *learn* these filters?

Too early to rejoice

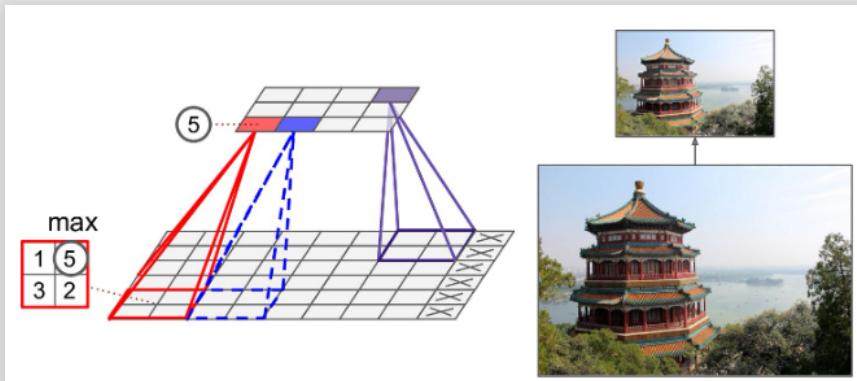
That's still quite a lot.

- 100 x 100 RGB image
- 100 feature maps in first layer of filter 3x3
- $(3 \times 3 \times 3 + 1) \times 100 = 2800$ params, not too bad
- But 100 x 100 numbers in each feature map ($\times 100$) = 1M numbers, each say takes 4B, that's 4MB for 1 image for 1 layer
- Each number is a weighted sum of $3 \times 3 \times 3 = 27$ numbers, so that's $1M \times 27 = 27M$ multiplications for 1 layer...

Pooling Layers

A pooling layer "sums up" a convolutional layer, by taking the max or mean of the receptive field.

No params!



Source: [Geron 2019](#)

Usually with strides $s_w = s_h = f_w = f_h$ and no padding (a.k.a VALID):

| S1 | =MAX(M1:N2) | | | | | | | | | | | | | | | | | | | |
|----|-------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | | 1 | 1 | 2 | 1 | 1 | | 1 | 2 |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | | 2 | 3 |
| 3 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | | 1 | 1 | 1 | | 1 | 1 | 0 | 1 | 1 | | | |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | 1 | 2 | 3 | 2 | 1 | | | |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | | | | | | 0 | 0 | 0 | 0 | 0 | | | |
| 6 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | | | | | | | | | | | | | |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | | | | | | |
| 8 | | | | | | | | | | | | | | | | | | | | |
| 9 | | X | | | | | | | W | | | | Z | | | | | | | |
| 10 | | | | | | | | | | | | | | | | | | | | |

Clearly we're losing info.

💡 A 2x2 max pooling layer would reduce the size of the previous layer by how many percents?

But maybe sometimes it's a *good* thing to ignore some neurons?
Where did we get this crazy idea from?

Image Data Augmentation

It's very hard to "augment" data about people, products, clicks.

Images are different.

- Translation
- Rotation
- Rescaling
- Flipping
- Stretching



Are there some images or applications you wouldn't want augmentations or at least should go about it very carefully?

In Keras you can augment "on the fly" as you train your model, here I'm just showing you this works:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

idg = ImageDataGenerator(
    rotation_range=30,
    zoom_range=0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.15,
    horizontal_flip=True,
    fill_mode='nearest'
)

ny_generator = idg.flow(ny4D, batch_size=1)

nys = [ny_generator.next() for i in range(10)]
```

See the [ImageDataGenerator](#) for more.



Why do CNN work?

- No longer a long 1D column of neurons, but 2D, taking into account spatial relations between pixels/neurons
- First layer learns very low-level features, second layer learns higher level features, etc.
- Shared weights --> learn feature in one area of the image, generalize it to the entire image
- Less weights --> smaller size, more feasible model, less prone to overfitting
- Less overfitting by Pooling
- Invariance by Max Pooling
- Hardware/Software advancements enable processing of huge amounts of images

Back to Malaria!

APPLICATIONS



OF DATA SCIENCE

```
import tensorflow_datasets as tfds
from skimage.transform import resize
from sklearn.model_selection import train_test_split

malaria, info = tfds.load('malaria', split='train', with_info=True)

images = []
labels = []
for example in tfds.as_numpy(malaria):
    images.append(resize(example['image'], (100, 100)).astype(np.float32))
    labels.append(example['label'])
    if len(images) == 2500:
        break

X = np.array(images)
y = np.array(labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

print(X_train.shape)
```

```
## (2000, 100, 100, 3)
```

```
print(X_test.shape)
```

```
## (500, 100, 100, 3)
```

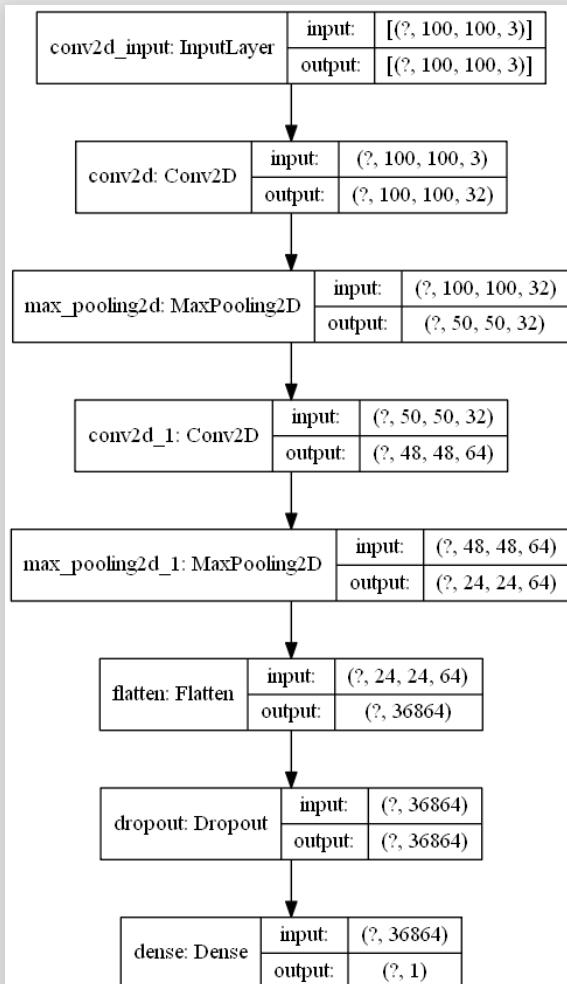
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, ...
from tensorflow.keras.callbacks import EarlyStopping

model = Sequential()
model.add(Conv2D(filters=32, input_shape=(X_train.shape[1:]),
    kernel_size=(3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
    optimizer='adam', metrics=['accuracy'])
```

```
model.summary()
```

```
## Model: "sequential"
##
##             Layer (type)      Output Shape       Param #
## =====
## conv2d (Conv2D)        (None, 100, 100, 32)    896
## max_pooling2d (MaxPooling2D) (None, 50, 50, 32)    0
## conv2d_1 (Conv2D)       (None, 48, 48, 64)    18496
## max_pooling2d_1 (MaxPooling2D) (None, 24, 24, 64)    0
## flatten (Flatten)      (None, 36864)        0
## dropout (Dropout)      (None, 36864)        0
## dense (Dense)          (None, 1)            36865
## =====
## Total params: 56,257
## Trainable params: 56,257
## Non-trainable params: 0
##
```

```
from tensorflow.keras import utils  
  
utils.plot_model(model, 'images/malaria_cnn.png', show_shapes=True)
```

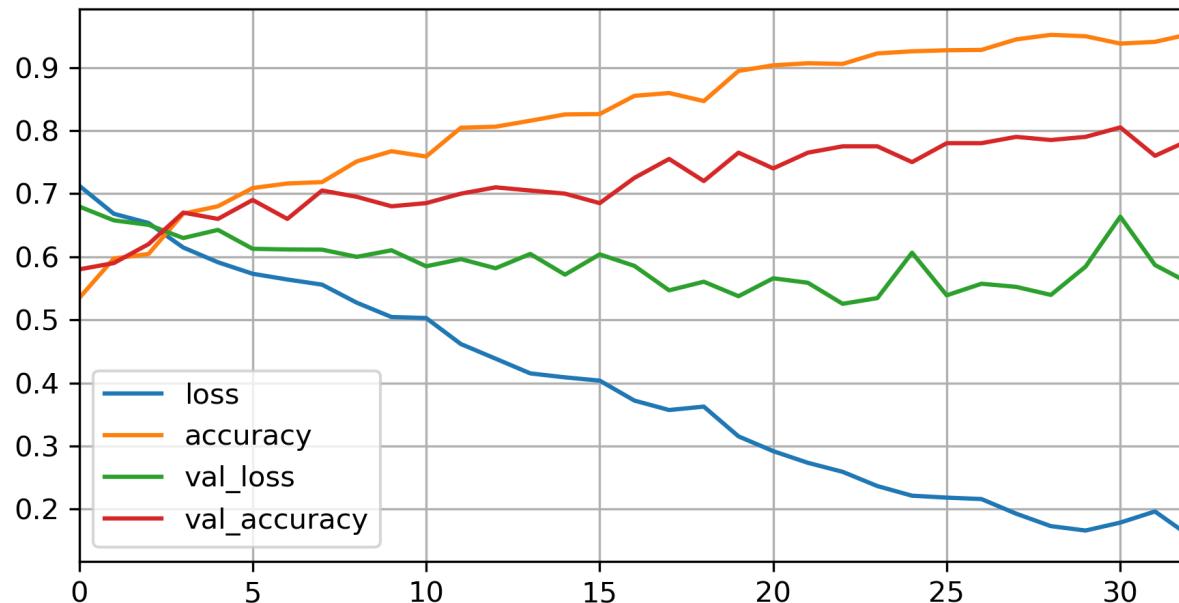


```
callbacks = [EarlyStopping(monitor='val_loss', patience=5)]
history = model.fit(X_train, y_train, batch_size=100, epochs=50,
    validation_split=0.1, callbacks=callbacks)

# Epoch 1/50
#
# 1/18 [>.....] - ETA: 0s - loss: 0.6919
# 2/18 [==>.....] - ETA: 8s - loss: 0.6895
# 3/18 [====>.....] - ETA: 11s - loss: 0.7002
# 4/18 [=====>.....] - ETA: 11s - loss: 0.7009
# 5/18 [=====>.....] - ETA: 11s - loss: 0.6954
# 6/18 [=====>.....] - ETA: 10s - loss: 0.6929
# 7/18 [=====>.....] - ETA: 9s - loss: 0.6882
```

```
import pandas as pd

pd.read_csv(history.history).plot()
plt.grid(True)
plt.show()
```



Last time we got to 66% accuracy after tuning...

```
from sklearn.metrics import confusion_matrix

y_pred = (model.predict(X_test) > 0.5).astype(int).reshape(y_test.

np.mean(y_pred == y_test)

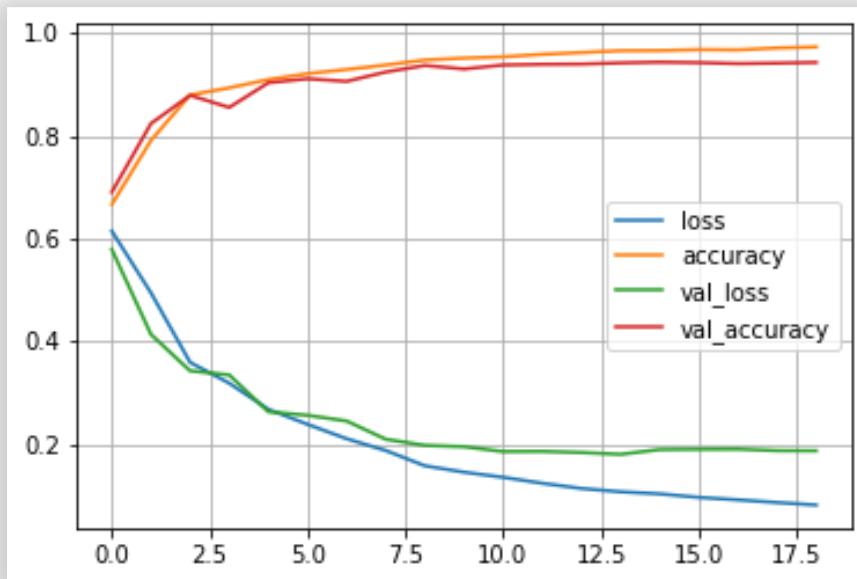
## 0.766

pd.DataFrame(
    confusion_matrix(y_test, y_pred),
    index=['true:yes', 'true:no'],
    columns=['pred:yes', 'pred:no']
)

##           pred:yes  pred:no
## true:yes      191      71
## true:no       46     192
```

And this is just 10% of the data.

Watch me reach ~94% accuracy with 100% of the data in [Google Colab](#).



Think about the researchers seeing this for the first time in 2012...

Visualizing filters/kernels/features/weights

```
W, b = model.get_layer('conv2d').get_weights()
```

```
W.shape
```

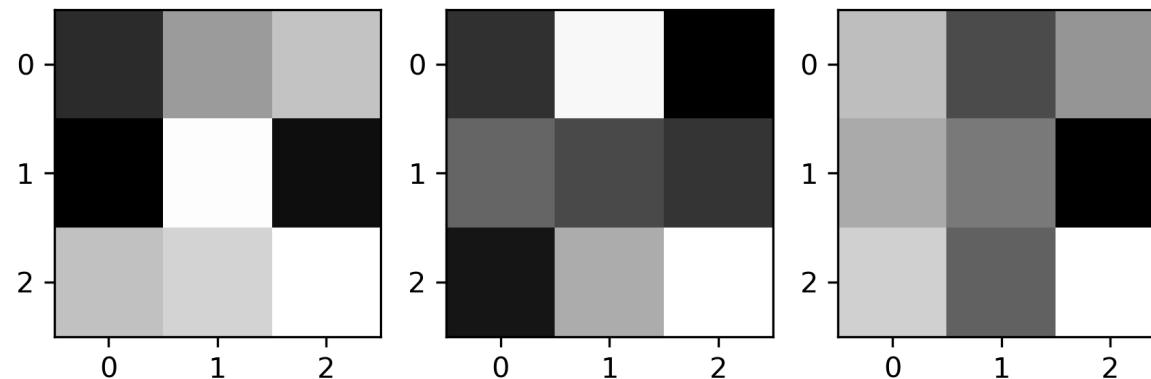
```
## (3, 3, 3, 32)
```

```
W[:, :, 0, 0]
```

```
## array([[-0.05327014,  0.05076471,  0.0883847 ],
##         [-0.0933094 ,  0.14134812, -0.08010183],
##         [ 0.08545554,  0.10305361,  0.14380825]], dtype=float32)
```

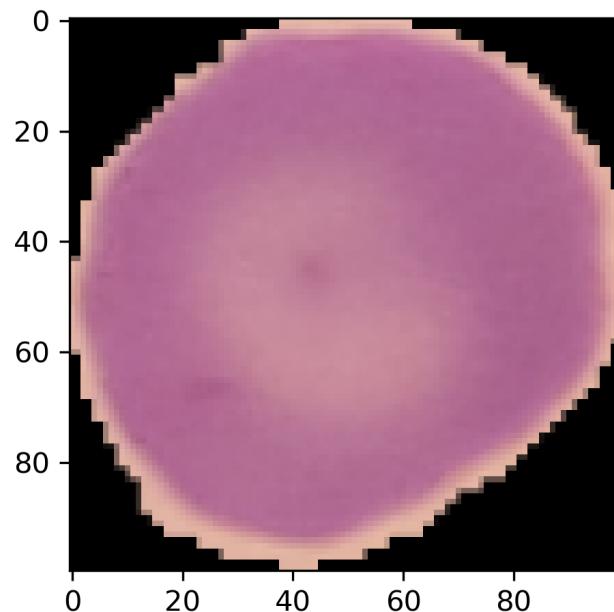
First filter of 32:

```
plt.subplot(1, 3, 1)
plt.imshow(W[:, :, 0, 0], cmap='gray')
plt.subplot(1, 3, 2)
plt.imshow(W[:, :, 1, 0], cmap='gray')
plt.subplot(1, 3, 3)
plt.imshow(W[:, :, 2, 0], cmap='gray')
plt.show()
```



Visualizing Feature Maps

```
cell = X_test[0, :, :, :].reshape(1, 100, 100, 3)  
  
plt.imshow(cell[0])  
plt.show()
```



```

from tensorflow.keras import Model

model_1layer = Model(inputs=model.inputs,
                      outputs=model.layers[0].output)

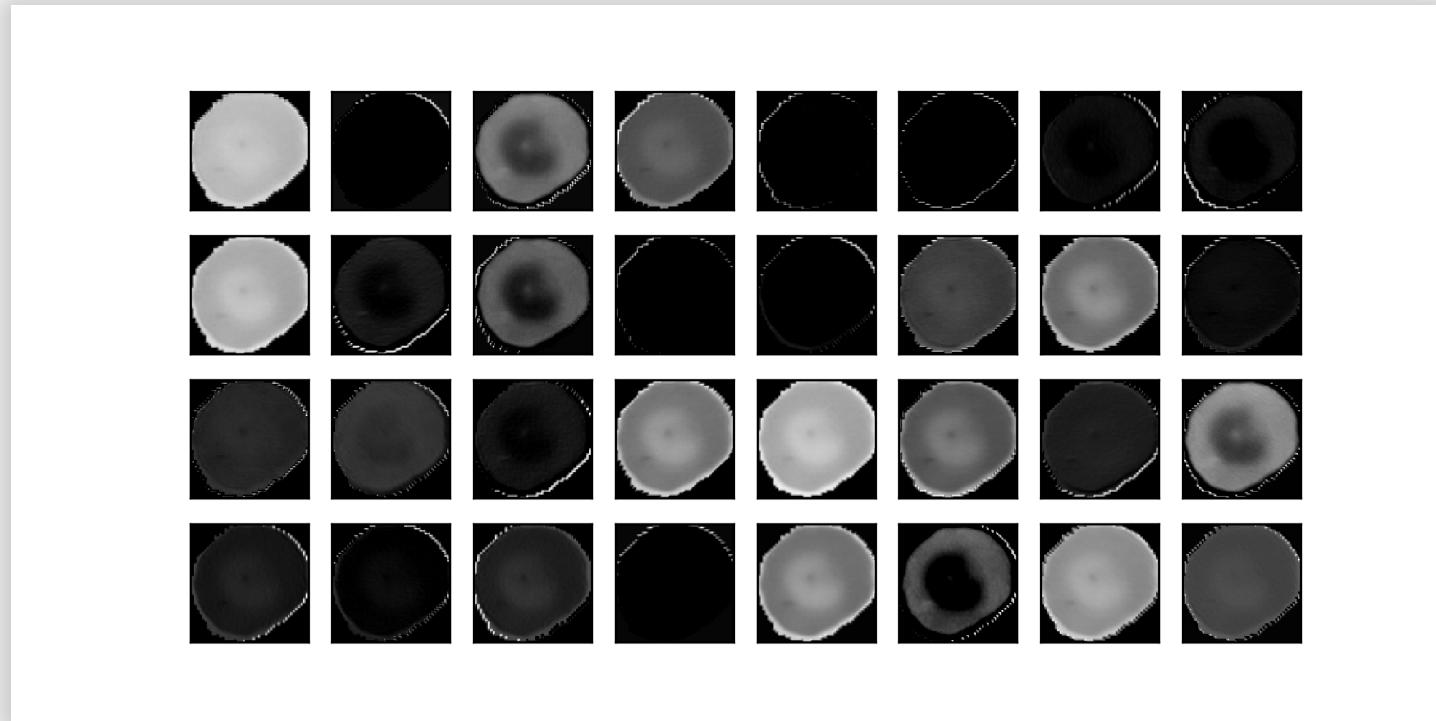
feature_maps = model_1layer.predict(cell)

feature_maps.shape

## (1, 100, 100, 32)

# source: https://machinelearningmastery.com/how-to-visualize-filters-in-a-convolutional-neural-network/
width = 8
height = 4
ix = 1
for _ in range(height):
    for _ in range(width):
        ax = plt.subplot(height, width, ix)
        _ = ax.set_xticks([])
        _ = ax.set_yticks([])
        _ = plt.imshow(feature_maps[0, :, :, ix-1], cmap='gray')
        ix += 1
plt.show()

```



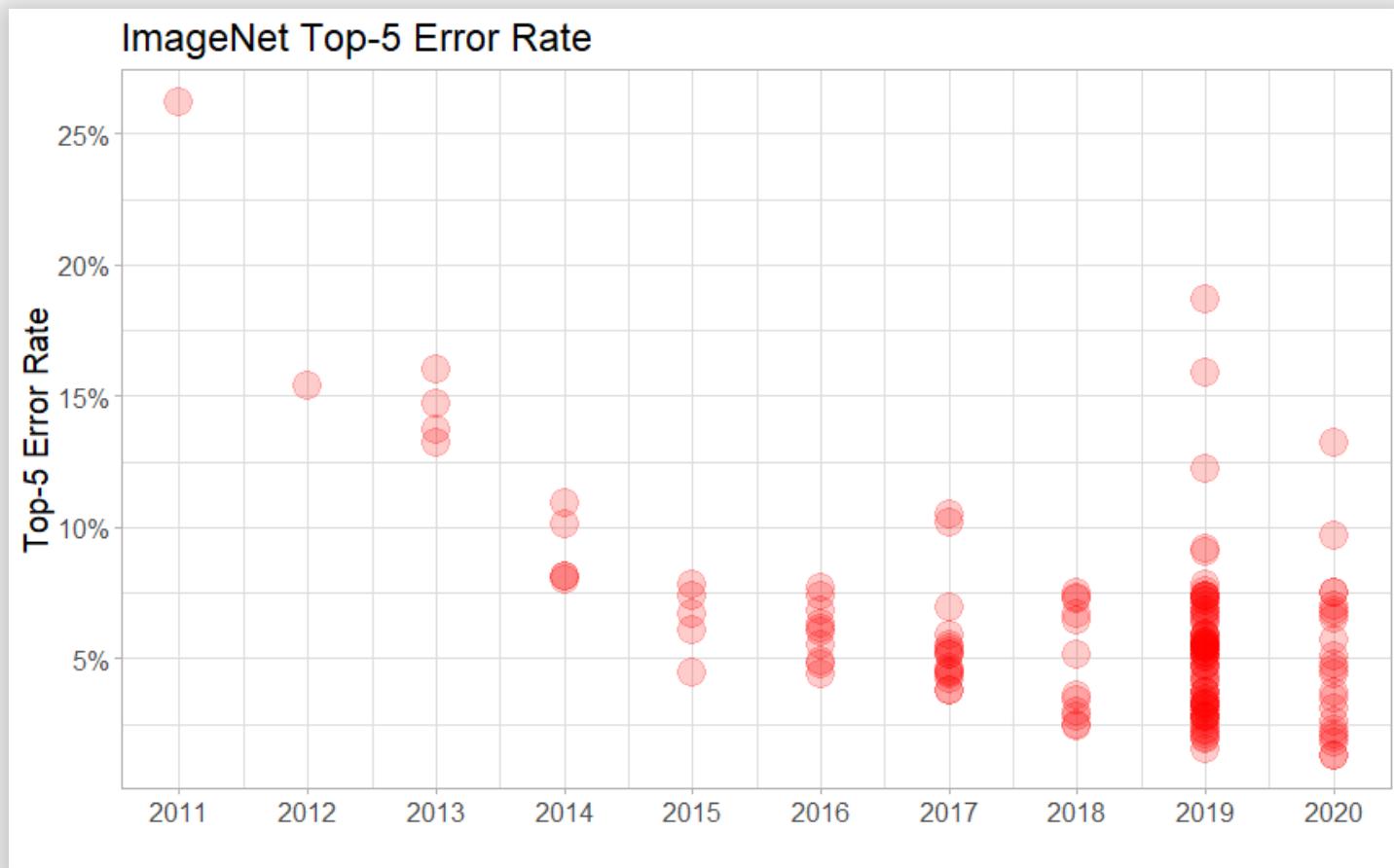
CNN Architectures

APPLICATIONS

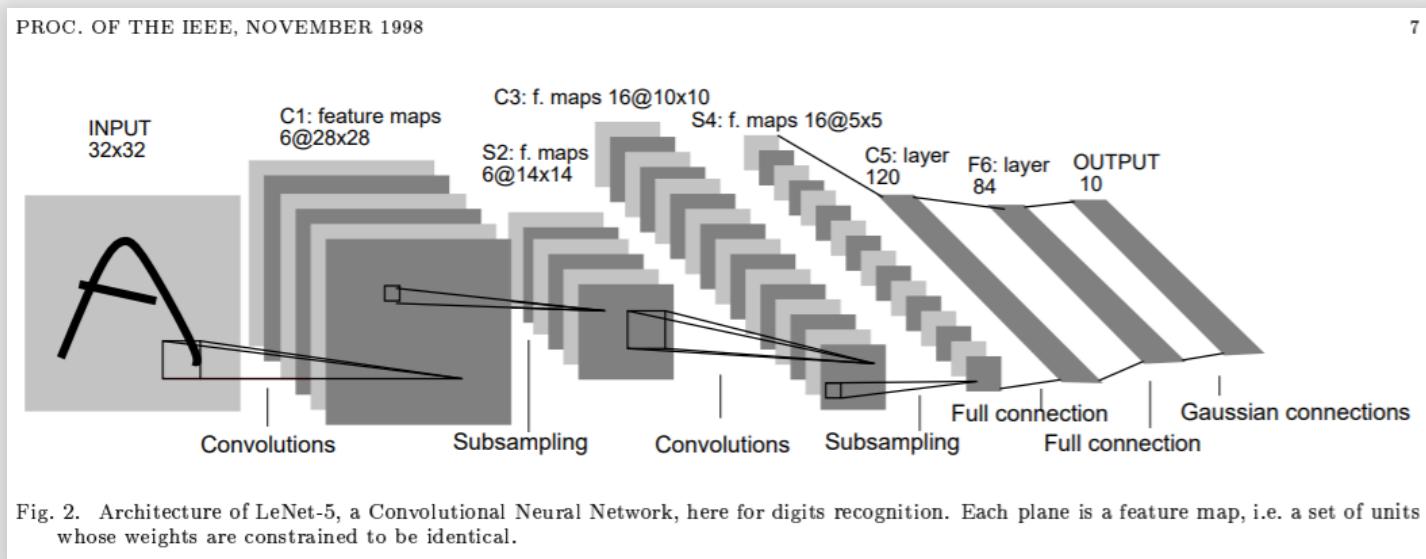


OF DATA SCIENCE

ImageNet



LeNet-5 (1998)

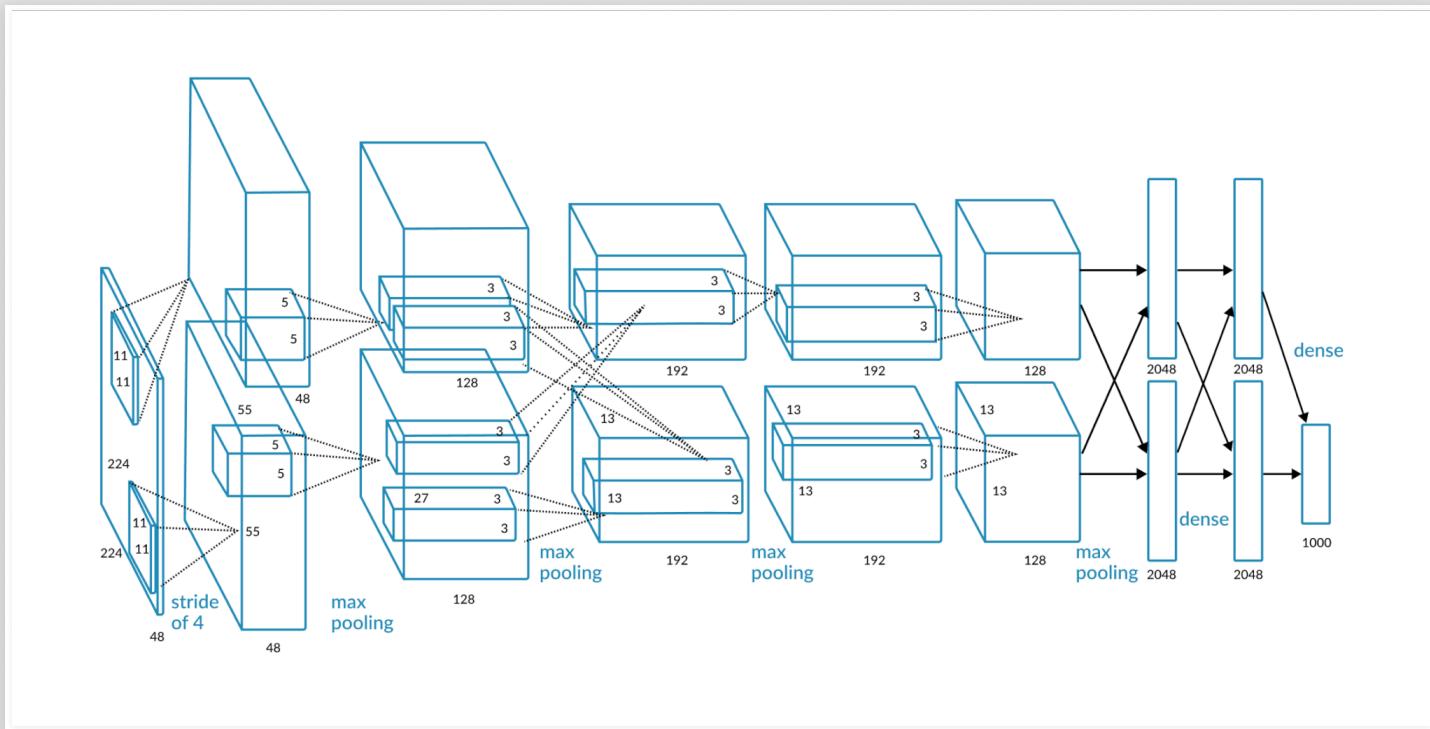


Source: [LeCun et. al. 1998](#)

You know you can implement this in just a few lines of Keras...

```
from tensorflow.keras.layers import Conv2D, AveragePooling2D,  
    Flatten, Dense  
from tensorflow.keras import Sequential  
  
model = keras.Sequential()  
  
model.add(Conv2D(filters=6, kernel_size=(3, 3),  
    activation='relu', input_shape=(32, 32, 1)))  
model.add(AveragePooling2D())  
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu'))  
model.add(AveragePooling2D())  
model.add(Flatten())  
model.add(Dense(120, activation='relu'))  
model.add(Dense(84, activation='relu'))  
model.add(Dense(10, activation = 'softmax'))
```

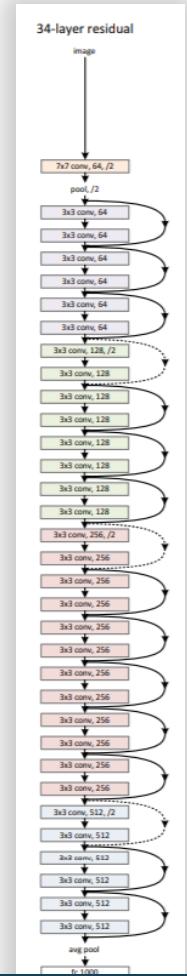
AlexNet (2012)



Source: [Krizhevsky et. al. 2012](#)

See a Keras implementation e.g. [here](#)

ResNet (2015)



- Source: [He et. al.](#)
- This is ResNet-34 (34 layers)
- ResNet-152 (152 layers!) achieved 4.5% To-5 error rate with single model

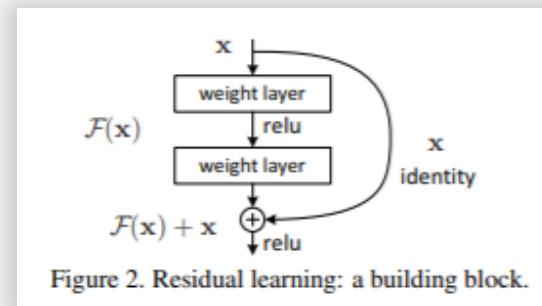


Figure 2. Residual learning: a building block.

Residual Learning

- "Is learning better networks as easy as stacking more layers?"
Yes, but:
- "When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated... and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error" (a.k.a the vanishing/exploding gradients problem)
- With *Residual Learning* the activation from a previous layer is being added to the activation of a deeper layer in the network
- The signal is allowed to propagate through the layers

 The model learning $H(X)$ will be forced to learn $H(X) - X$, hence *residual learning*.

Using Pre-trained Models

APPLICATIONS



OF DATA SCIENCE

Why are we working so hard?

Keras Applications

Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

Weights are downloaded automatically when instantiating a model. They are stored at `~/.keras/models/`.

Upon instantiation, the models will be built according to the image data format set in your Keras configuration file at `~/.keras/keras.json`. For instance, if you have set `image_data_format=channels_last`, then any model loaded from this repository will get built according to the TensorFlow data format convention, "Height-Width-Depth".

Available models

| Model | Size | Top-1 Accuracy | Top-5 Accuracy | Parameters | Depth |
|-------------------|--------|----------------|----------------|-------------|-------|
| Xception | 88 MB | 0.790 | 0.945 | 22,910,480 | 126 |
| VGG16 | 528 MB | 0.713 | 0.901 | 138,357,544 | 23 |
| VGG19 | 549 MB | 0.713 | 0.900 | 143,667,240 | 26 |
| ResNet50 | 98 MB | 0.749 | 0.921 | 25,636,712 | - |
| ResNet101 | 171 MB | 0.764 | 0.928 | 44,707,176 | - |
| ResNet152 | 232 MB | 0.766 | 0.931 | 60,419,944 | - |
| ResNet50V2 | 98 MB | 0.760 | 0.930 | 25,613,800 | - |
| ResNet101V2 | 171 MB | 0.772 | 0.938 | 44,675,560 | - |
| ResNet152V2 | 232 MB | 0.780 | 0.942 | 60,380,648 | - |
| InceptionV3 | 92 MB | 0.779 | 0.937 | 23,851,784 | 159 |
| InceptionResNetV2 | 215 MB | 0.803 | 0.963 | 55,073,726 | 572 |

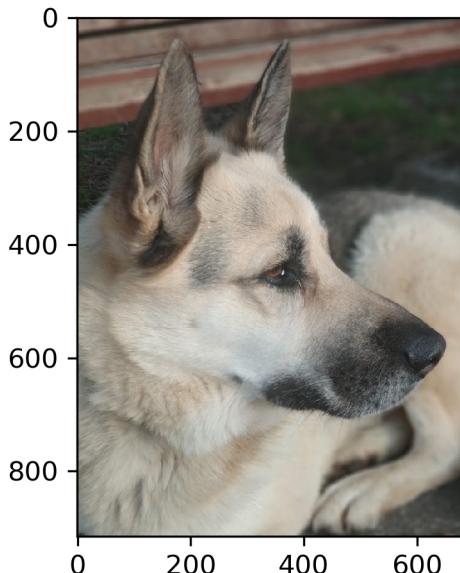
Look up keras.io/api/applications/

```
from tensorflow.keras.applications.resnet50 import ResNet50, prepro
from skimage.transform import resize

model = ResNet50(weights='imagenet')

johann = imread('images/johann.jpg')

fig = plt.imshow(johann)
plt.show()
```



```
johann = resize(johann, (224, 224))
johann = johann.reshape(1, 224, 224, 3) * 255
johann = preprocess_input(johann)

johann_pred = model.predict(johann)

print(johann_pred.shape)
```

```
## (1, 1000)
```

```
johann_breed = decode_predictions(johann_pred, top=5)

for _, breed, score in johann_breed[0]:
    print('%s: %.2f' % (breed, score))
```

```
## Norwegian_elkhound: 0.30
## Eskimo_dog: 0.28
## Siberian_husky: 0.17
## German_shepherd: 0.14
## malamute: 0.06
```

Transfer Learning

APPLICATIONS



OF DATA SCIENCE

