Till now in MLP

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()
# Imports a Flatten layer to convert the image matrix into a vector
# Defines the neural network architecture
model.add( Flatten(input_shape = (28,28) ))
model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
model.summary()
```

- Params after the flatten layer = 0, because this layer only flattens the image to a vector for feeding into the input layer. The weights haven't been added yet.
- Params after layer 1 = (784 nodes in input layer) × (512 in hidden layer 1) + (512 connections to biases) = 401,920.
- Params after layer 2 = (512 nodes in hidden layer 1) × (512 in hidden layer 2) + (512 connections to biases) = 262,656.
- Params after layer 3= (512 nodes in hidden layer 2) × (10 in output layer) + (10 connections to biases) = 5,130.
- Total params in the network = 401,920 + 262,656 + 5,130 = 669,706.

Why Convolutions?

SPATIAL INVARIANCE or LOSS IN FEATURES

The spatial features of a 2D image are lost when it is flattened to a 1D vector input. Before feeding an image to the hidden layers of an MLP, we must flatten the image matrix to a 1D vector, as we saw in the mini project. This implies that all of the image's 2D information is discarded.

```
from google.colab.patches import cv2_imshow
import os, json, cv2, random

!wget http://images.cocodataset.org/val2017/000000439715.jpg -q -O input.jpg
im = cv2.imread("./input.jpg")
cv2_imshow(im)
```

Image Matrix

```
im
```

```
array([[[140, 160, 118],
        [128, 156, 120],
        [123, 138, 111],
        ...,
        [ 67, 109,  92],
        [ 62,  88,  52],
        [ 78, 100,  72]],

       [[125, 173, 125],
        [122, 149, 109],
        [102, 133,  96],
        ...,
        [ 94, 104,  98],
        [ 59,  95,  71],
        [ 78, 114, 100]],

       [[108, 157, 119],
        [110, 157, 119],
        [ 84, 151, 112],
        ...,
        [ 69, 124,  97],
        [ 78, 110,  79],
```

```
 [ 88, 104,  80]],

...,

[[ 21,  19,  19],
 [ 21,  19,  19],
 [ 17,  18,  16],
 ...,
 [155, 156, 177],
 [155, 153, 175],
 [158, 155, 177]],

[[ 17,  16,  18],
 [ 22,  22,  22],
 [ 16,  18,  18],
 ...,
 [175, 173, 195],
 [174, 168, 191],
 [162, 154, 177]],

[[ 14,  13,  15],
 [ 22,  24,  25],
 [ 17,  19,  19],
 ...,
 [165, 162, 184],
 [166, 156, 179],
 [160, 147, 171]]], dtype=uint8)
```

Sample Image

0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

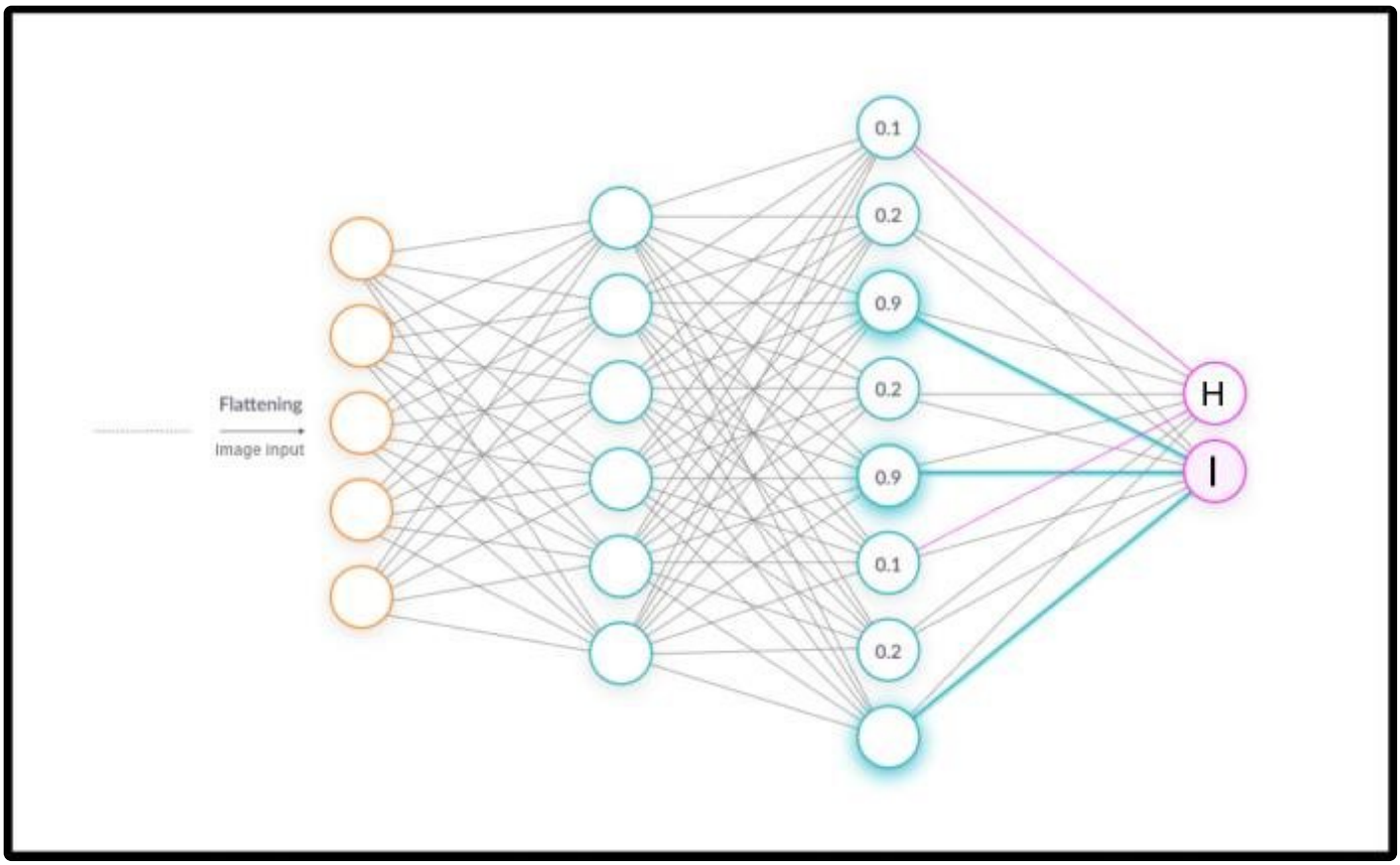
Increase in Parameter Issue

While increase in Parameter Issue is not a big problem for the
MNIST dataset because the images are really small in size (28 × 28), what happens when we try to process larger images?

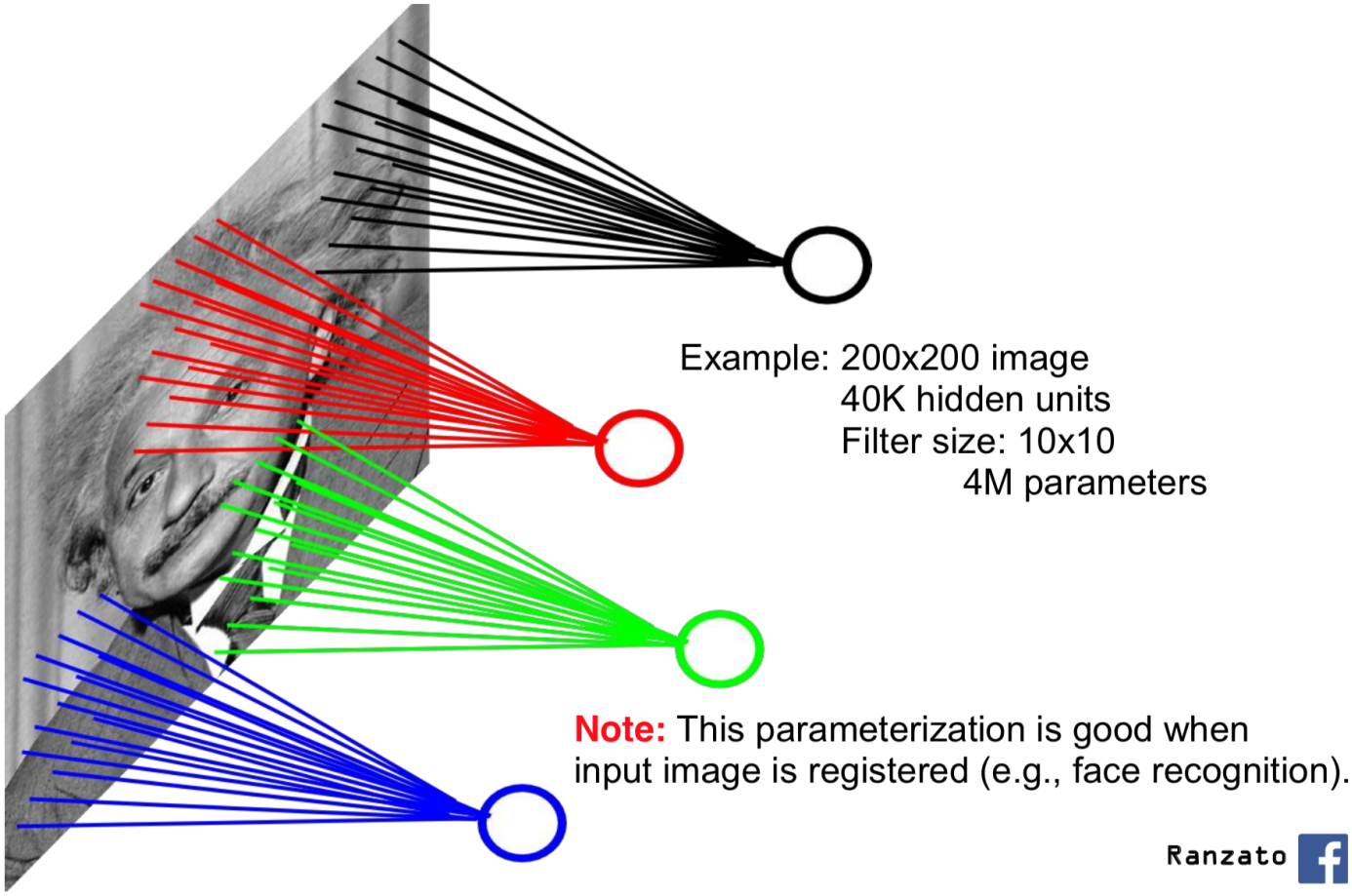
For example, if we have an image with dimensions 1,000 × 1,000, it will yield 1 million parameters for each node in the first hidden layer.

- So if the first hidden layer has 1,000 neurons, this will yield 1 billion parameters even in such a small network. You can imagine the computational complexity of optimizing 1 billion parameters after only the first layer.

Fully Connected Neural Net



Local Connected Neural Net

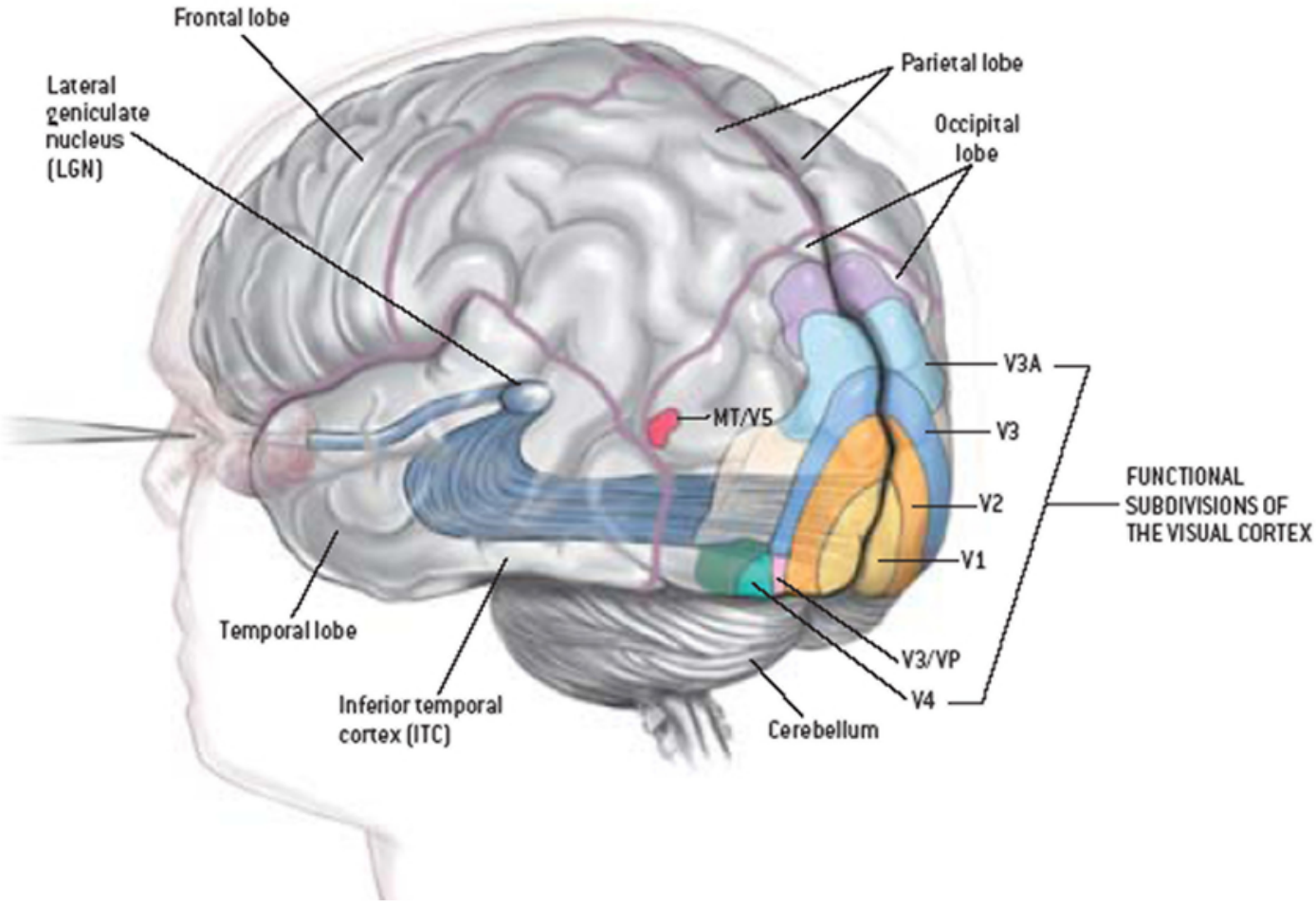


Source

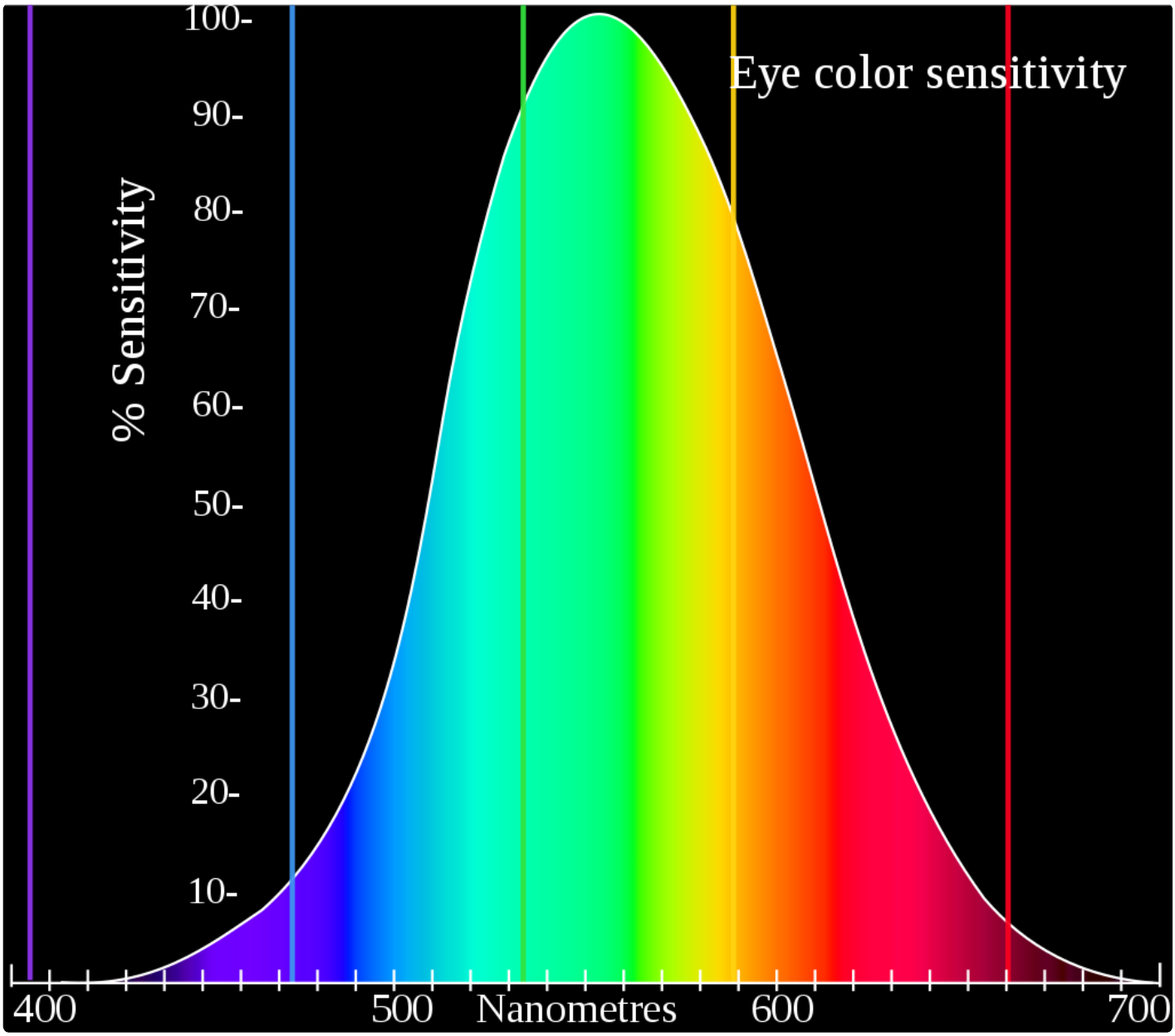
Guide for design of a neural network architecture suitable for computer vision

- In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called translation invariance.
- The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the locality principle. Eventually, these local representations can be aggregated to make predictions at the whole image level.

Human Brain Visual Cortex processing



Human Eye Colour Sensitivity



Source

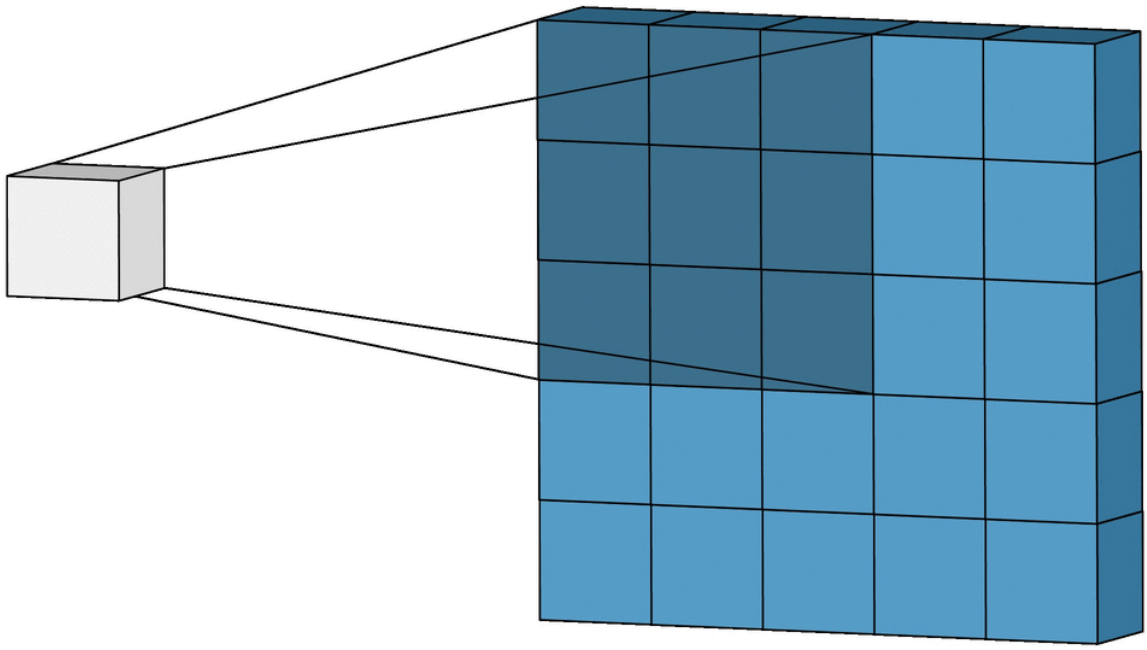
What are Convolutional Neural Networks?

Convolutional Neural Networks (ConvNets or CNNs) are a category of Neural Networks that have proven very effective in areas such as image recognition and classification. ConvNets have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self driving cars.

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. In this article we will discuss the architecture of a CNN and the back propagation algorithm to compute the gradient with respect to the parameters of the model in order to use gradient based optimization.

Visualizing the Process

Simple Convolution

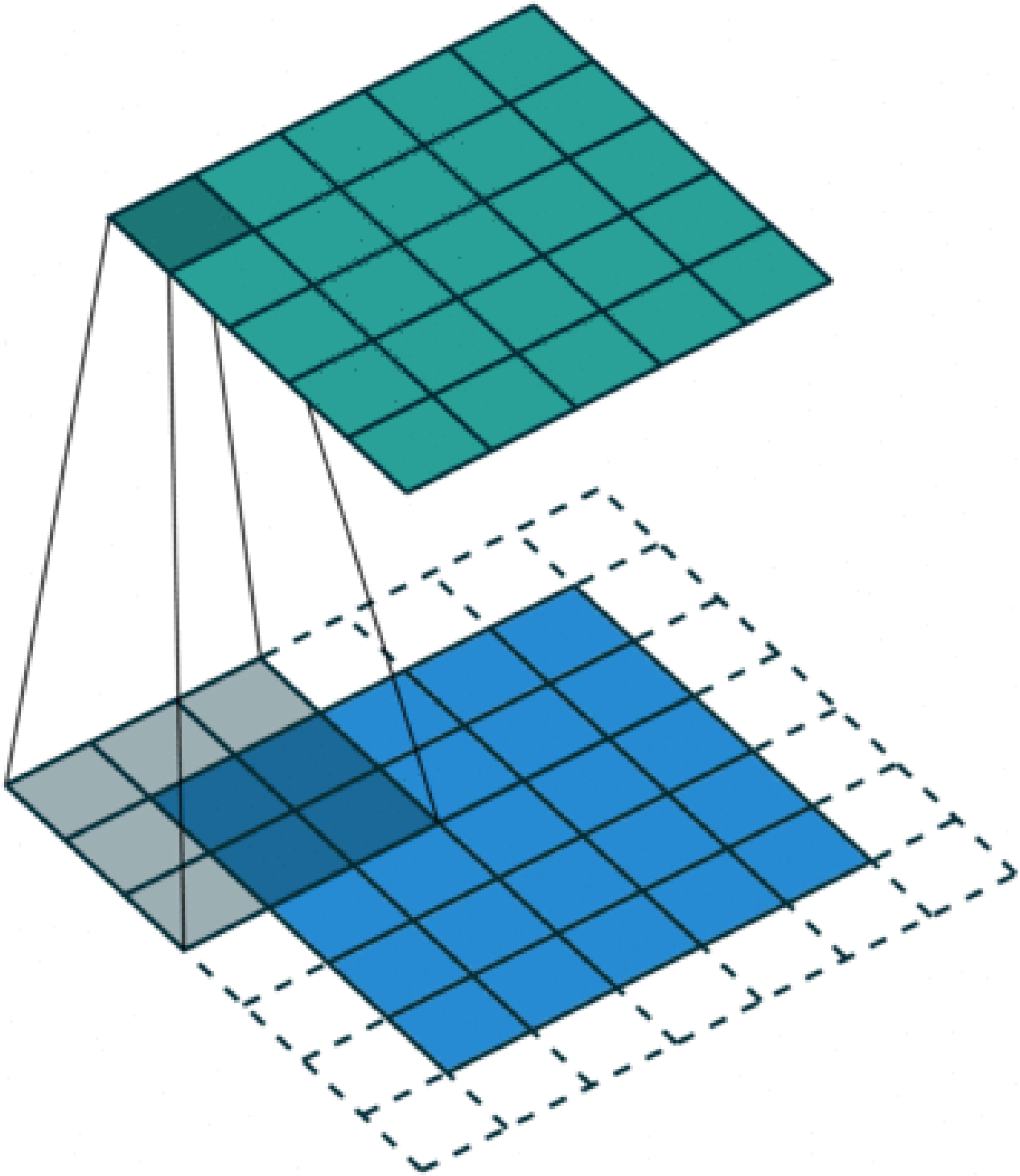


Matrix Calculation

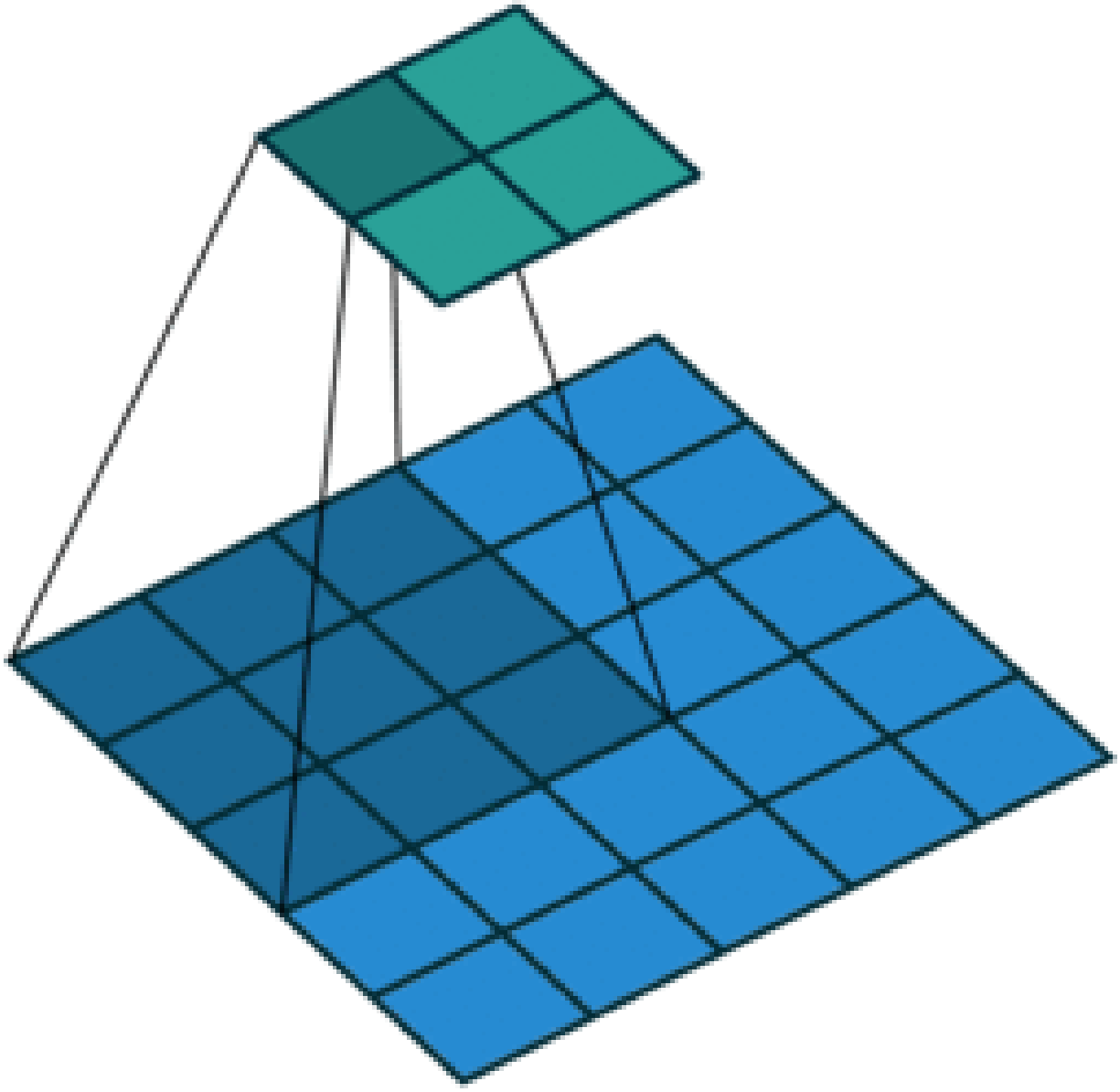
3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

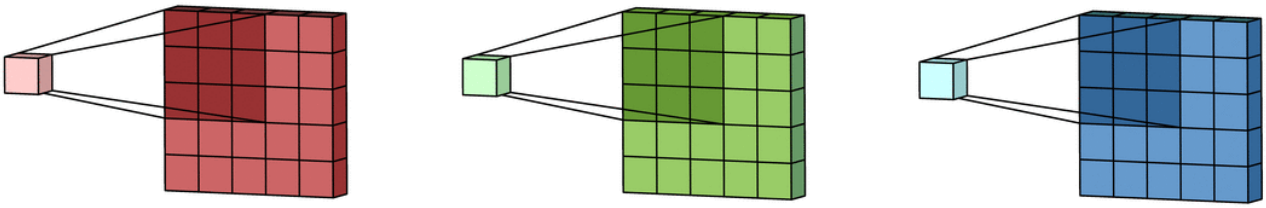
Padding Concept



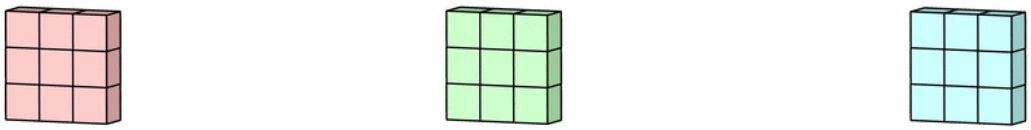
Stride Concept



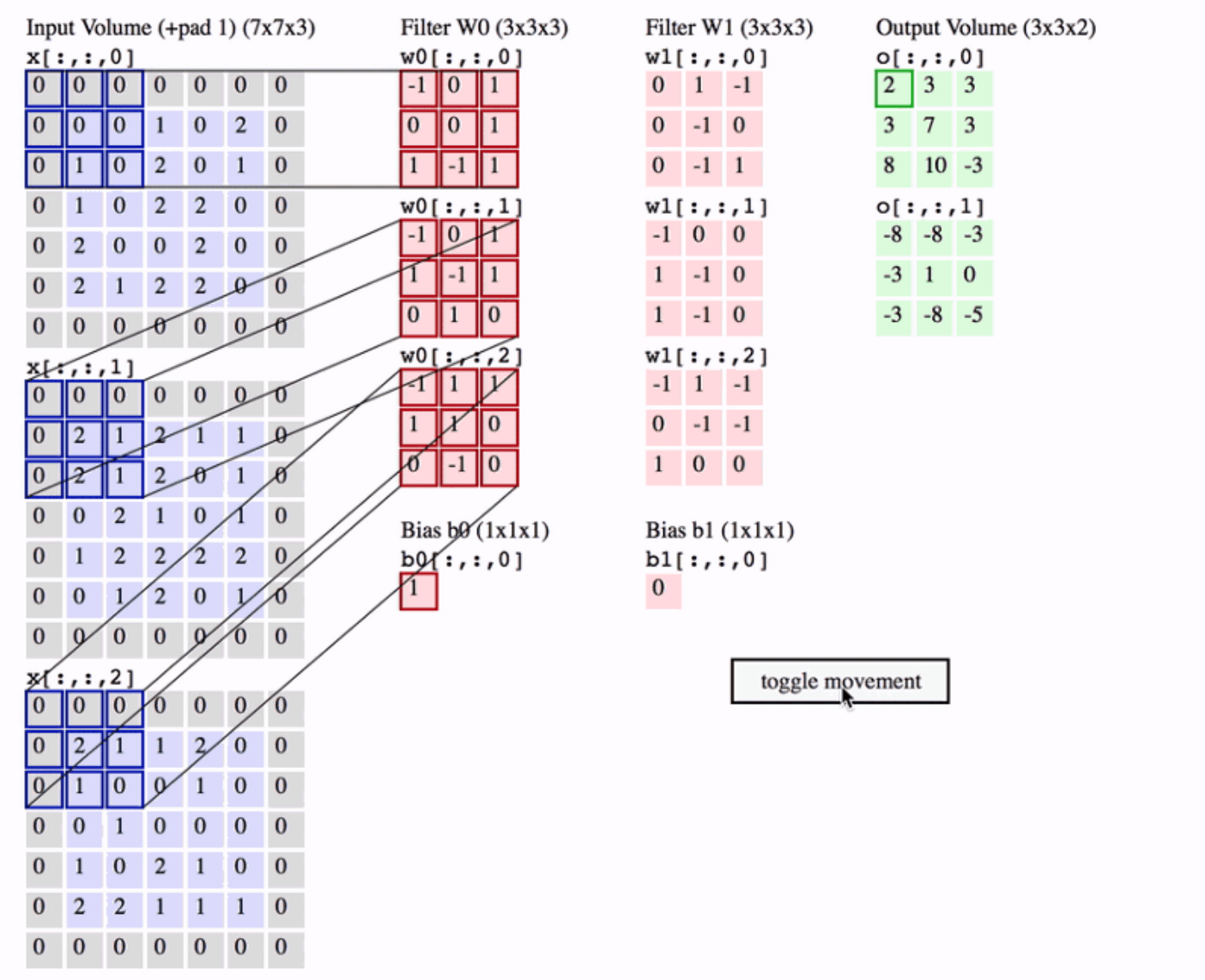
Feature Accumulation



Feature Aggregation



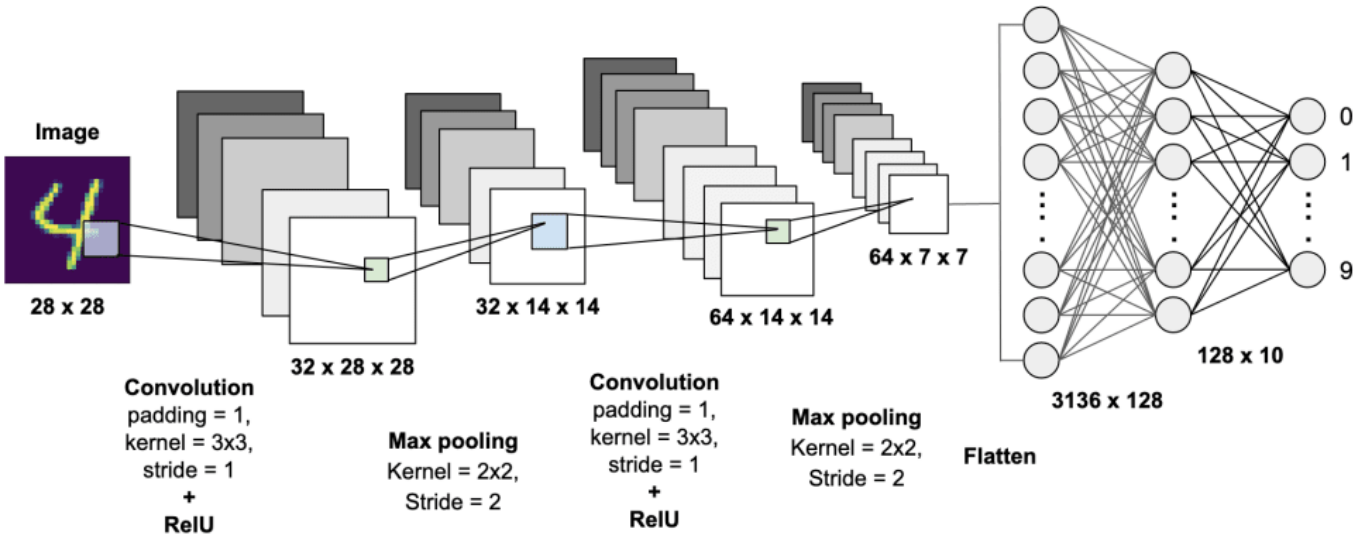
Convolution Operation



Source

Source

The CNN Complete Network Overview




```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

%matplotlib inline
```

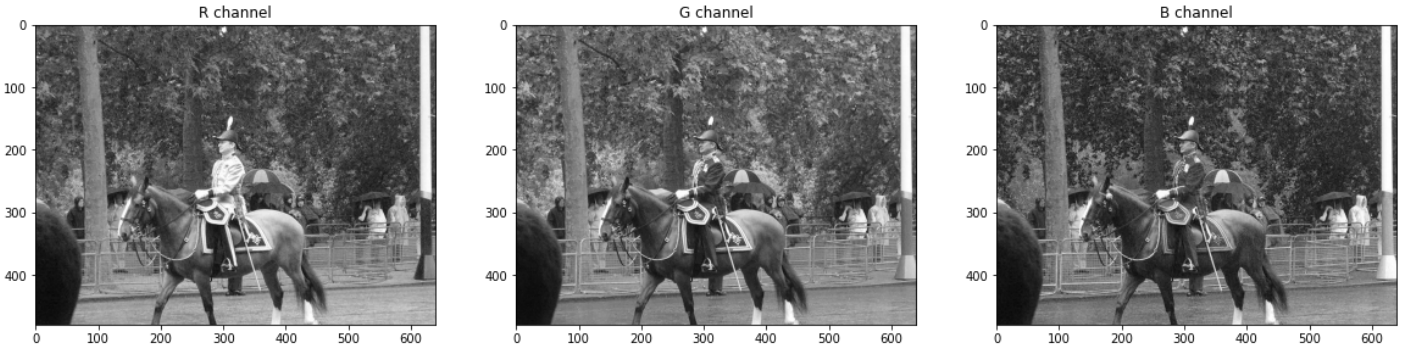
```
# Read in the image
image = mpimg.imread('input.jpg')

plt.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7f6b5c292bd0>
```


```
# Isolate RGB channels
r = image[:,:,0]
g = image[:,:,1]
b = image[:,:,2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
```

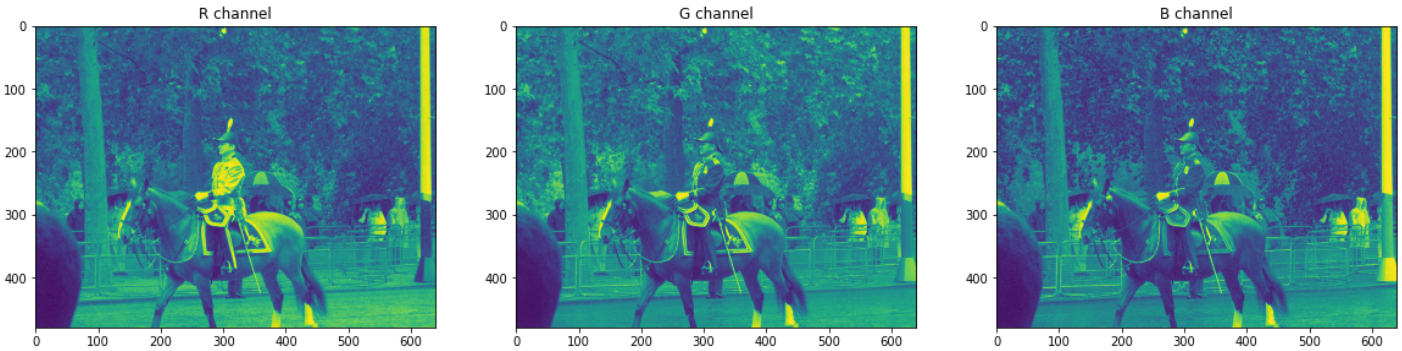
```
ax1.set_title('R channel')
ax1.imshow(r, cmap='gray')
ax2.set_title('G channel')
ax2.imshow(g, cmap='gray')
ax3.set_title('B channel')
ax3.imshow(b, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f6b5bd00910>
```




```
# Isolate RGB channels
r = image[:,:,0]
g = image[:,:,1]
b = image[:,:,2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.set_title('R channel')
ax1.imshow(r)
ax2.set_title('G channel')
ax2.imshow(g)
ax3.set_title('B channel')
ax3.imshow(b)
```

```
<matplotlib.image.AxesImage at 0x7f6b5bb9f1d0>
```

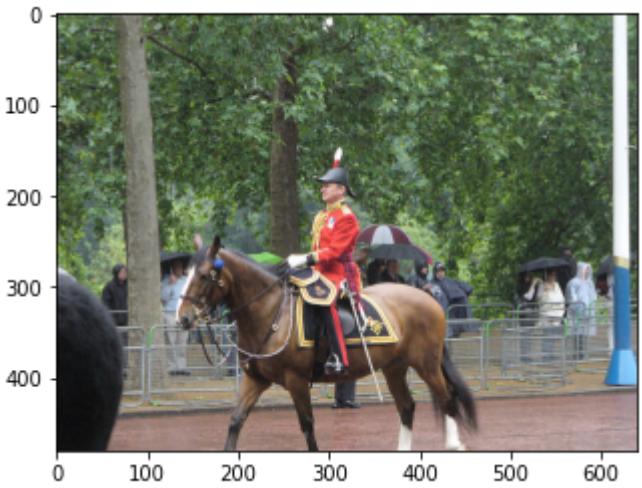


Focusing on Filters

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import cv2
import numpy as np
```

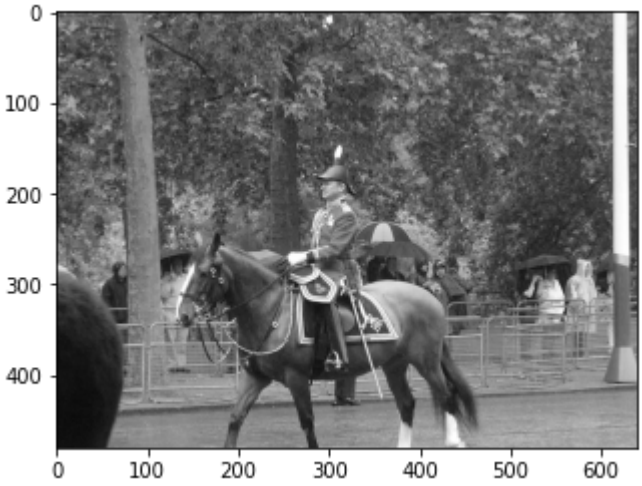
```
%matplotlib inline

# Read in the image
image = mpimg.imread('input.jpg')

plt.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7f6b5bb01710>
```



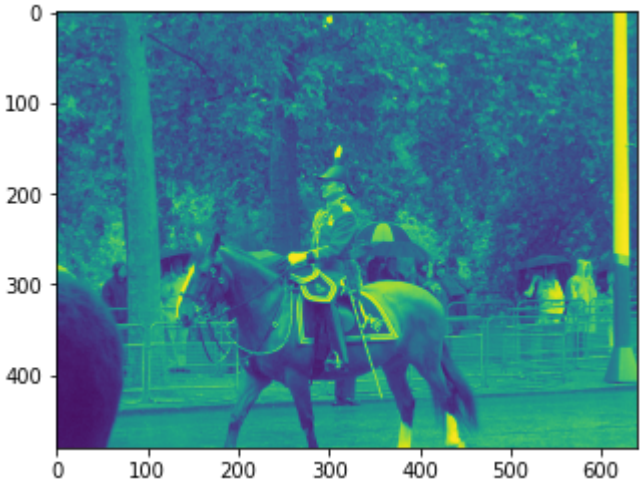

```
# Convert to grayscale for filtering
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

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

```
<matplotlib.image.AxesImage at 0x7f6b5ba63a50>
```



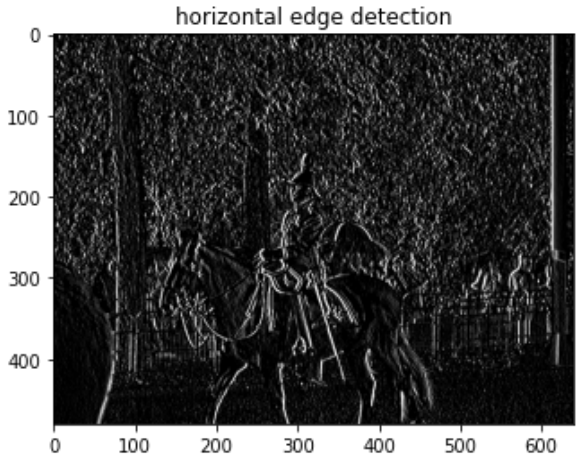

```
# Convert to grayscale for filtering
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

plt.imshow(gray)
```

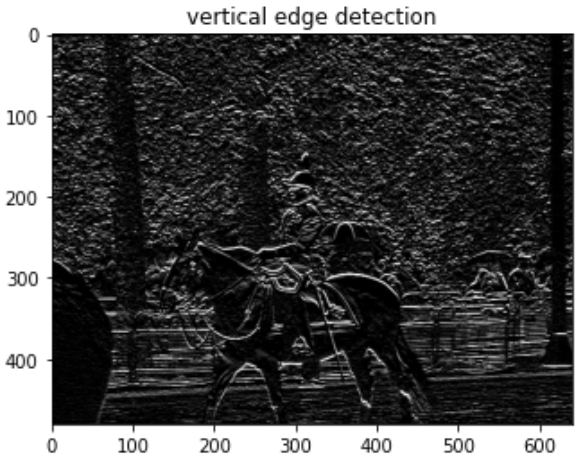
```
<matplotlib.image.AxesImage at 0x7f6b5b9ccf90>
```




```
sobel_y = np.array([[ -1, -2, -1],
                    [ 0, 0, 0],
                    [ 1, 2, 1]])
# vertical edge detection
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])
# filter the image using filter2D(grayscale image, bit-depth, kernel)
filtered_image1 = cv2.filter2D(gray, -1, sobel_x)
filtered_image2 = cv2.filter2D(gray, -1, sobel_y)
f, ax = plt.subplots(1, 2, figsize=(15, 4))
ax[0].set_title('horizontal edge detection')
ax[0].imshow(filtered_image1, cmap='gray')
ax[1].set_title('vertical edge detection')
ax[1].imshow(filtered_image2, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f6b5b8b04d0>
```



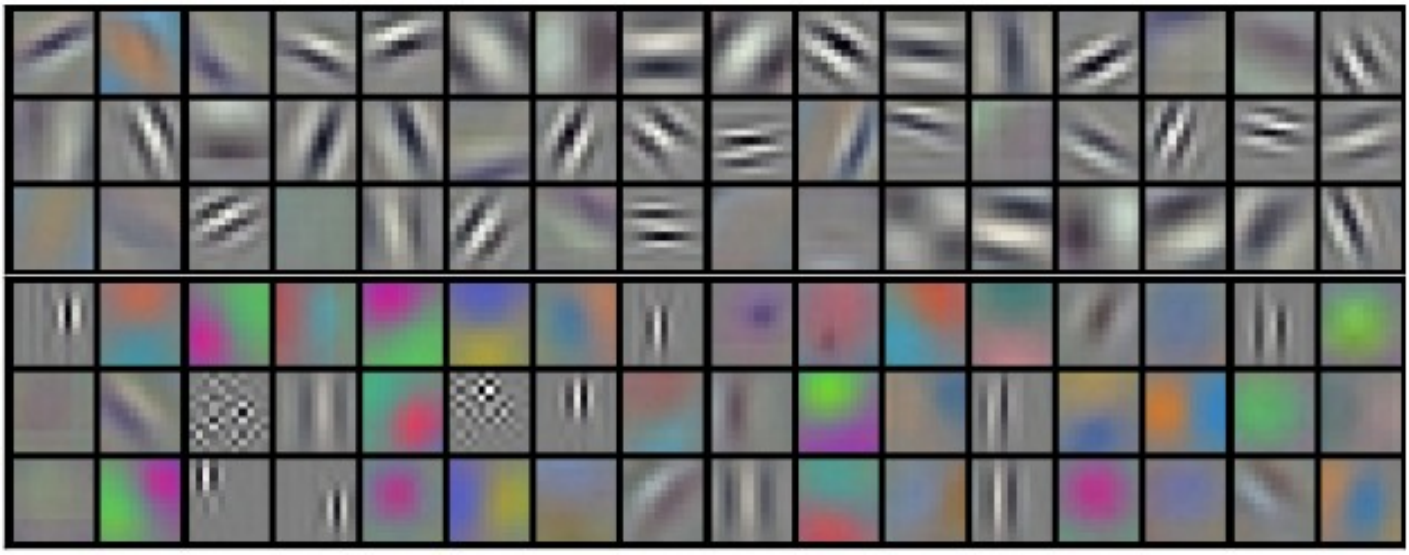


Best Place to Explore Kernels

Kernels

Kernels as Edge Detector

Features extracted by Kernels



Features > Patterns > Parts of Object

Source

Intuition

Let's develop better intuition for how Convolutional Neural Networks (CNN) work. We'll examine how humans classify images, and then see how CNNs use similar approaches.

Let’s say we wanted to classify the following image of a dog as a Golden Retriever:

As humans, how do we do this?

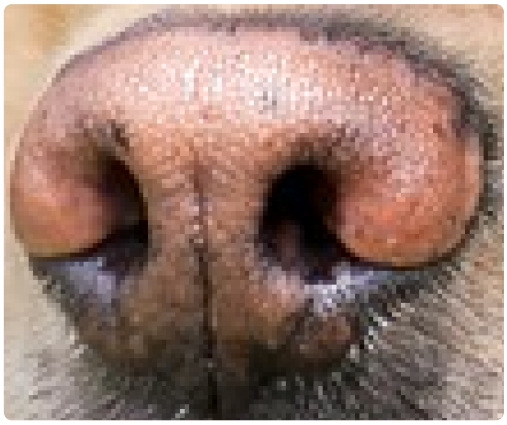
One thing we do is that we identify certain parts of the dog, such as the nose, the eyes, and the fur. We essentially break up the image into smaller pieces, recognize the smaller pieces, and then combine those pieces to get an idea of the overall dog.

In this case, we might break down the image into a combination of the following:

- A nose
- Two eyes
- Golden fur

These pieces can be seen below:

The eye of the dog.



The nose of the dog.

The fur of the dog.

Going One Step Further

But let's take this one step further. How do we determine what exactly a nose is? A Golden Retriever nose can be seen as an oval with two black holes inside it. Thus, one way of classifying a Retriever's nose is to to break it up into smaller pieces and look for black holes (nostrils) and curves that define an oval as shown below:

A curve that we can use to determine a nose

A nostril that we can use to classify the nose of the dog

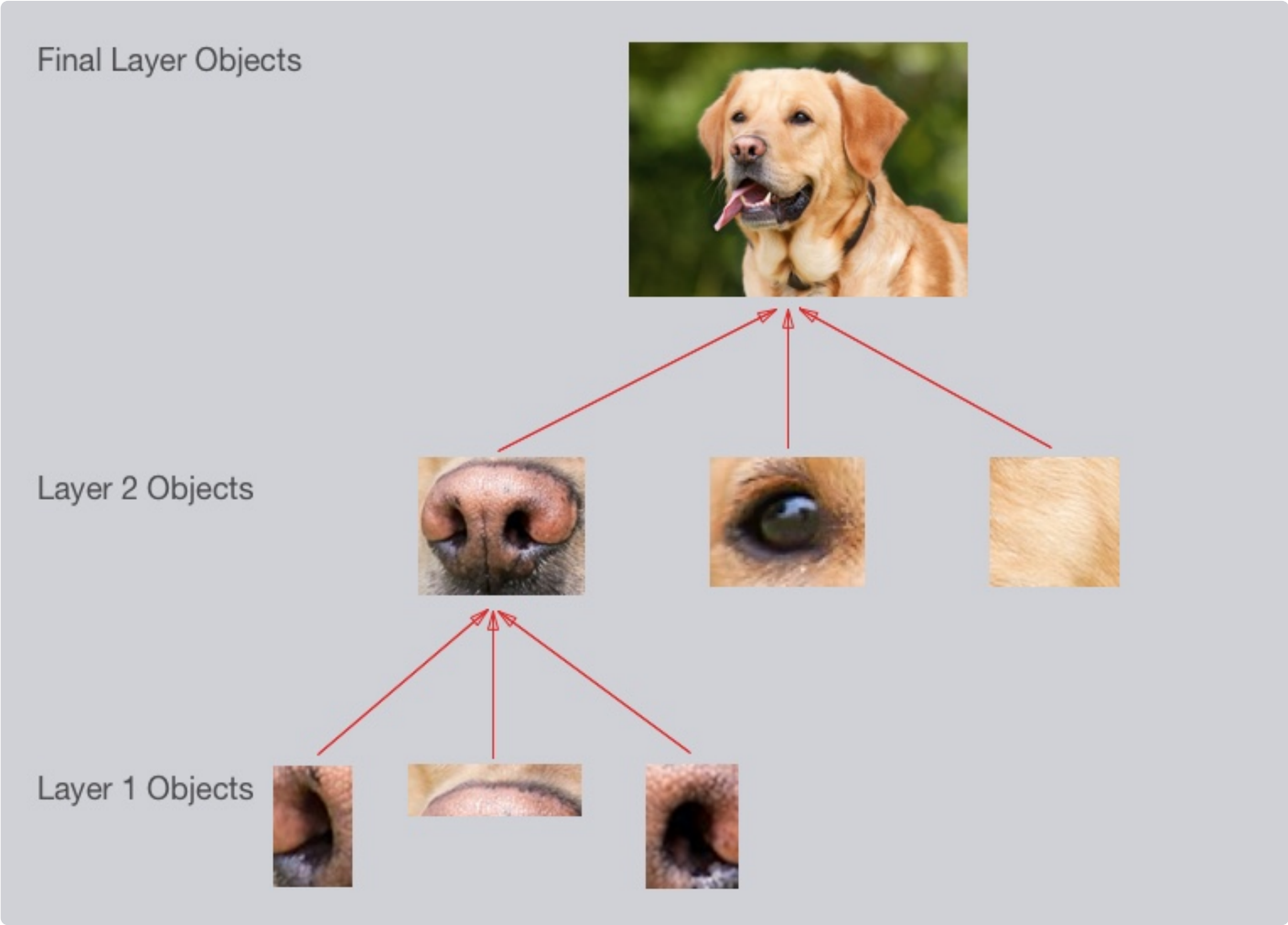
Broadly speaking, this is what a CNN learns to do. It learns to recognize basic lines and curves, then shapes and blobs, and then increasingly complex objects within the image. Finally, the CNN classifies the image by combining the larger, more complex objects.

In our case, the levels in the hierarchy are:

- Simple shapes, like ovals and dark circles
- Complex objects (combinations of simple shapes), like eyes, nose, and fur
- The dog as a whole (a combination of complex objects)

With deep learning, we don't actually program the CNN to recognize these specific features. Rather, the CNN learns on its own to recognize such objects through forward propagation and backpropagation!

It's amazing how well a CNN can learn to classify images, even though we never program the CNN with information about specific features to look for.



An example of what each layer in a CNN might recognize when classifying a picture of a dog

A CNN might have several layers, and each layer might capture a different level in the hierarchy of objects. The first layer is the lowest level in the hierarchy, where the CNN generally classifies small parts of the image into simple shapes like horizontal and vertical lines and simple blobs of colors. The subsequent layers tend to be higher levels in the hierarchy and generally classify more complex ideas like shapes (combinations of lines), and eventually full objects like dogs.

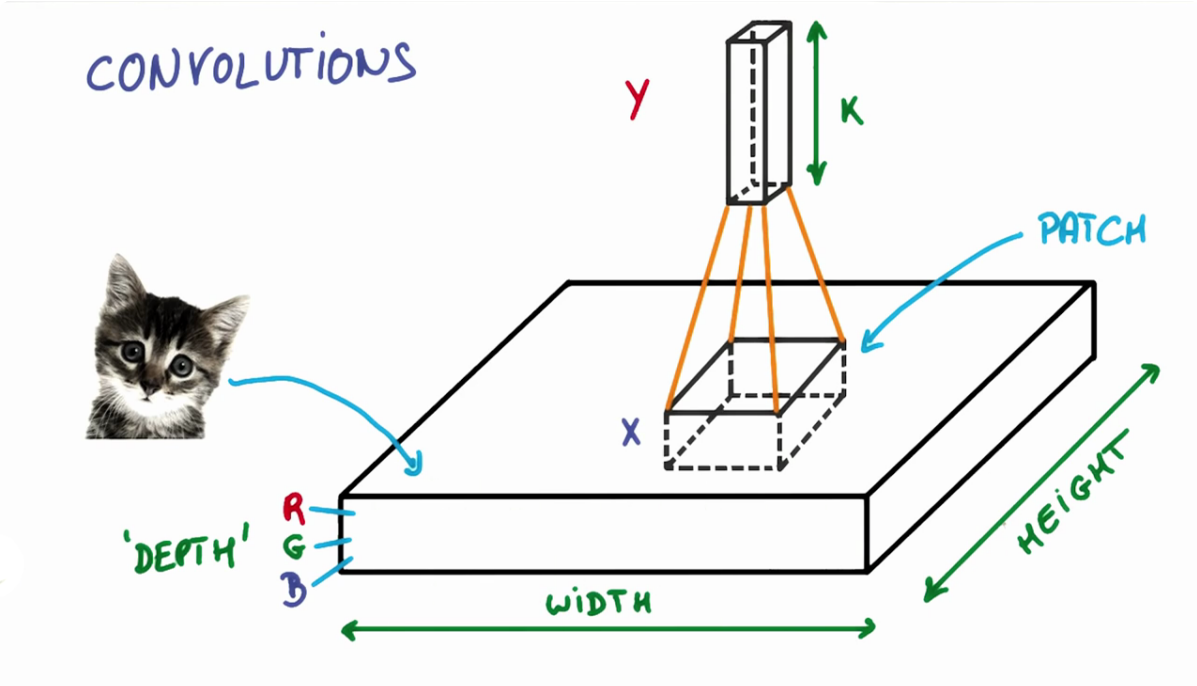
Once again, the CNN **learns all of this on its own**. We don't ever have to tell the CNN to go looking for lines or curves or noses or fur. The CNN just learns from the training set and discovers which characteristics of a Golden Retriever are worth looking for.

Filters

Breaking up an Image

The first step for a CNN is to break up the image into smaller pieces. We do this by selecting a width and height that defines a filter.

The filter looks at small pieces, or patches, of the image. These patches are the same size as the filter.

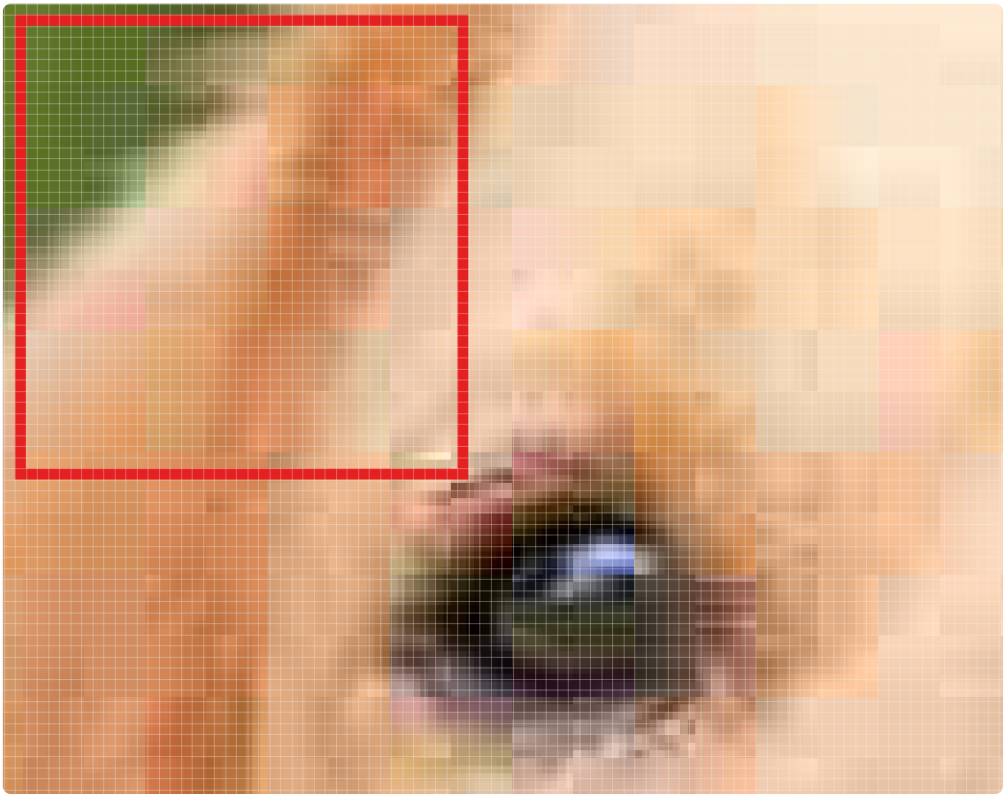


A CNN uses filters to split an image into smaller patches. The size of these patches matches the filter size.

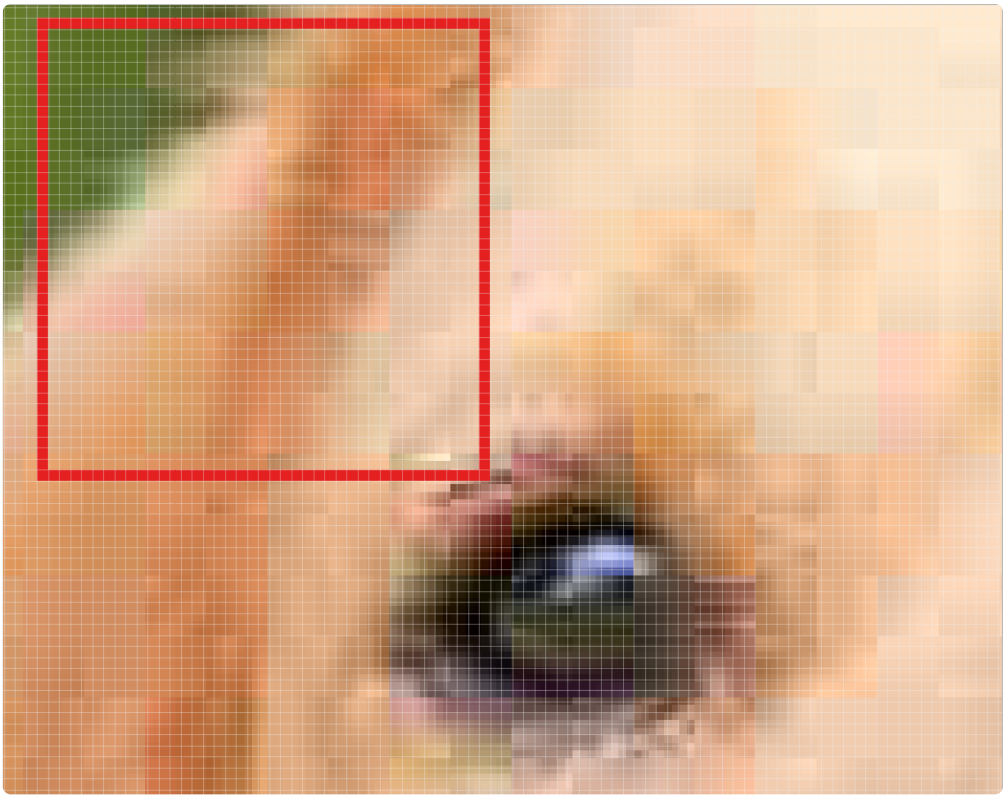
We then simply slide this filter horizontally or vertically to focus on a different piece of the image.

The amount by which the filter slides is referred to as the **'stride'**. The stride is a hyperparameter which the engineer can tune. Increasing the stride reduces the size of your model by reducing the number of total patches each layer observes. However, this usually comes with a reduction in accuracy.

Let's look at an example. In this zoomed in image of the dog, we first start with the patch outlined in red. The width and height of our filter define the size of this square.



We then move the square over to the right by a given stride (2 in this case) to get another patch.



We move our square to the right by two pixels to create another patch.

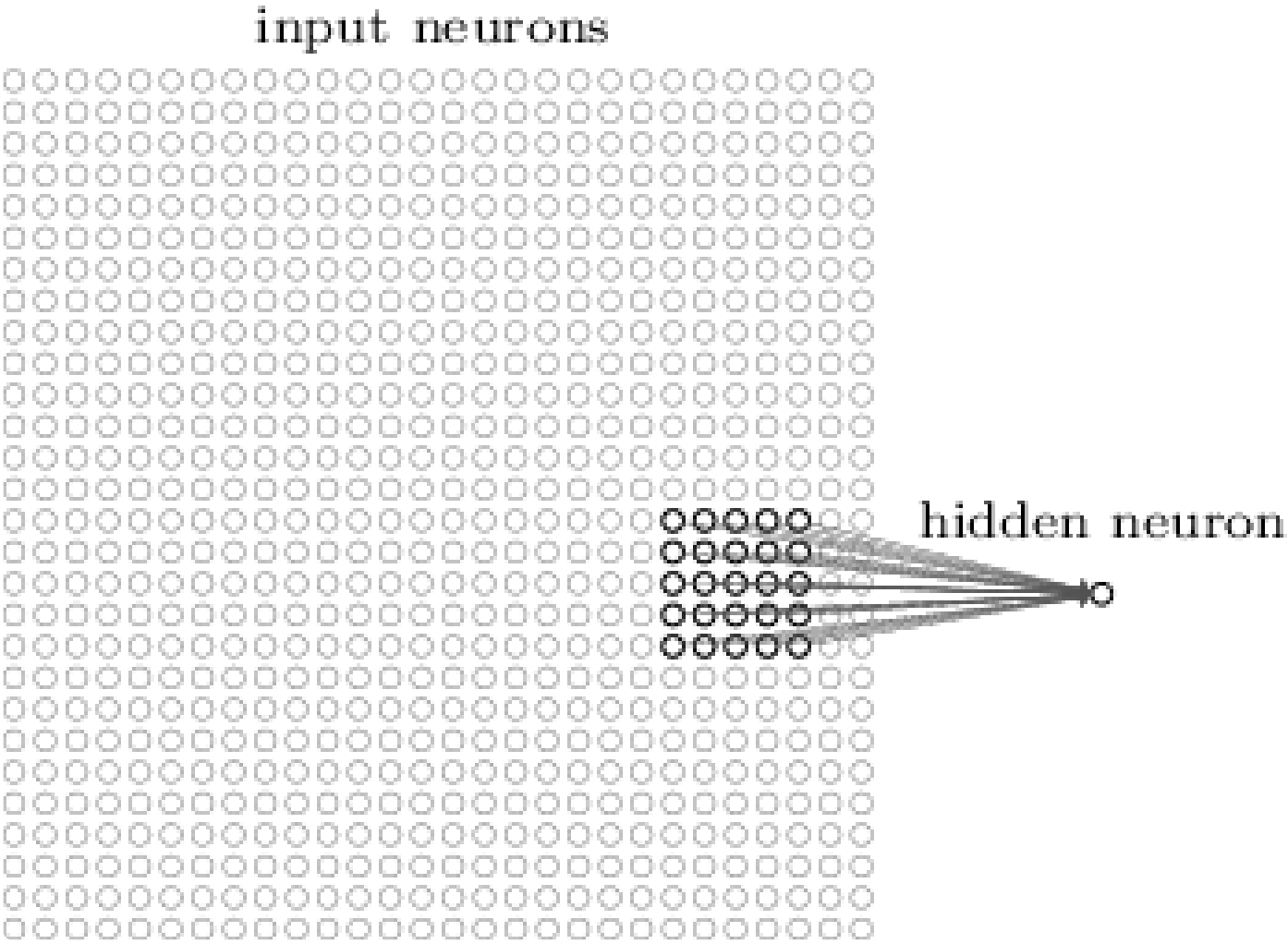
What's important here is that we are **grouping together adjacent pixels** and treating them as a collective.

In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason and have special meaning.

By taking advantage of this local structure, our CNN learns to classify local patterns, like shapes and objects, in an image.

Filter Depth

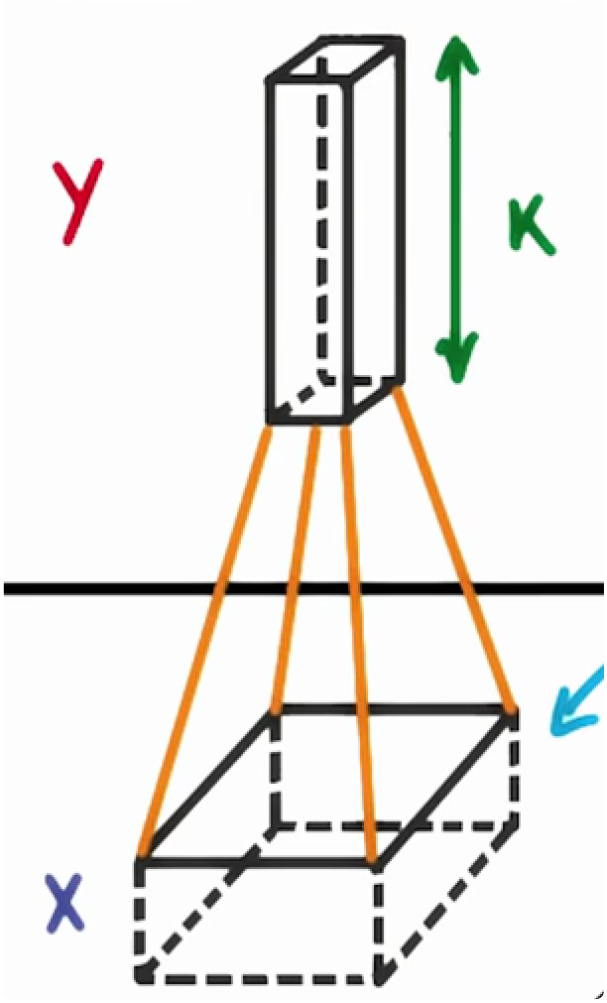
It's common to have more than one filter. Different filters pick up different qualities of a patch. For example, one filter might look for a particular color, while another might look for a kind of object of a specific shape. The amount of filters in a convolutional layer is called the **filter depth**.



In the above example, a patch is connected to a neuron in the next layer. Source: MIchael Neilsen.

How many neurons does each patch connect to?

That's dependent on our filter depth. If we have a depth of k, we connect each patch of pixels to k neurons in the next layer. This gives us the height of k in the next layer, as shown below. In practice, k is a hyperparameter we tune, and most CNNs tend to pick the same starting values.



Choosing a filter depth of k connects each path to k neurons in the next layer

But why connect a single patch to multiple neurons in the next layer? Isn't one neuron good enough?

Multiple neurons can be useful because a patch can have multiple interesting characteristics that we want to capture.

For example, one patch might include some white teeth, some blonde whiskers, and part of a red tongue. In that case, we might want a filter depth of at least three - one for each of teeth, whiskers, and tongue.

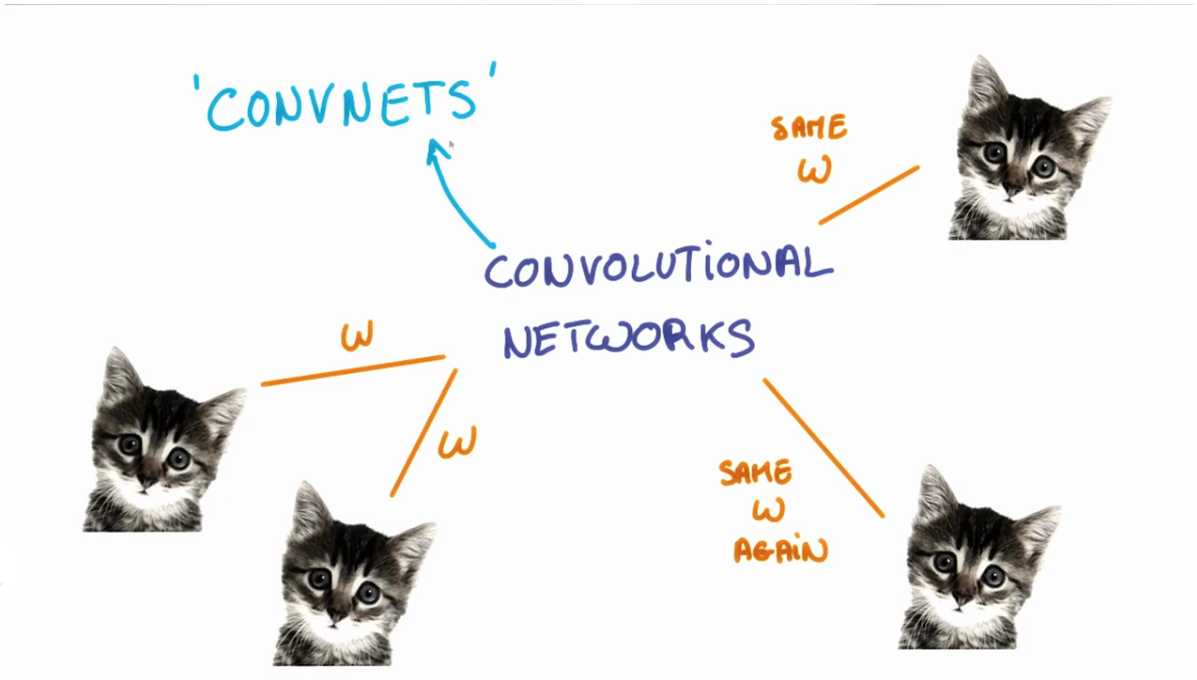
This patch of the dog has many interesting features we may want to capture. These include the presence of teeth, the presence of whiskers, and the pink color of the tongue.

Having multiple neurons for a given patch ensures that our CNN can learn to capture whatever characteristics the CNN learns are important.

Remember that the CNN isn't "programmed" to look for certain characteristics. Rather, it **learns on its own** which characteristics to notice.

Parameters

Parameter Sharing



The weights, w, are shared across patches for a given layer in a CNN to detect the cat above regardless of where in the image it is located.

When we are trying to classify a picture of a cat, we don't care where in the image a cat is. If it's in the top left or the bottom right, it's still a cat in our eyes. We would like our CNNs to also possess this ability known as translation invariance. How can we achieve this?

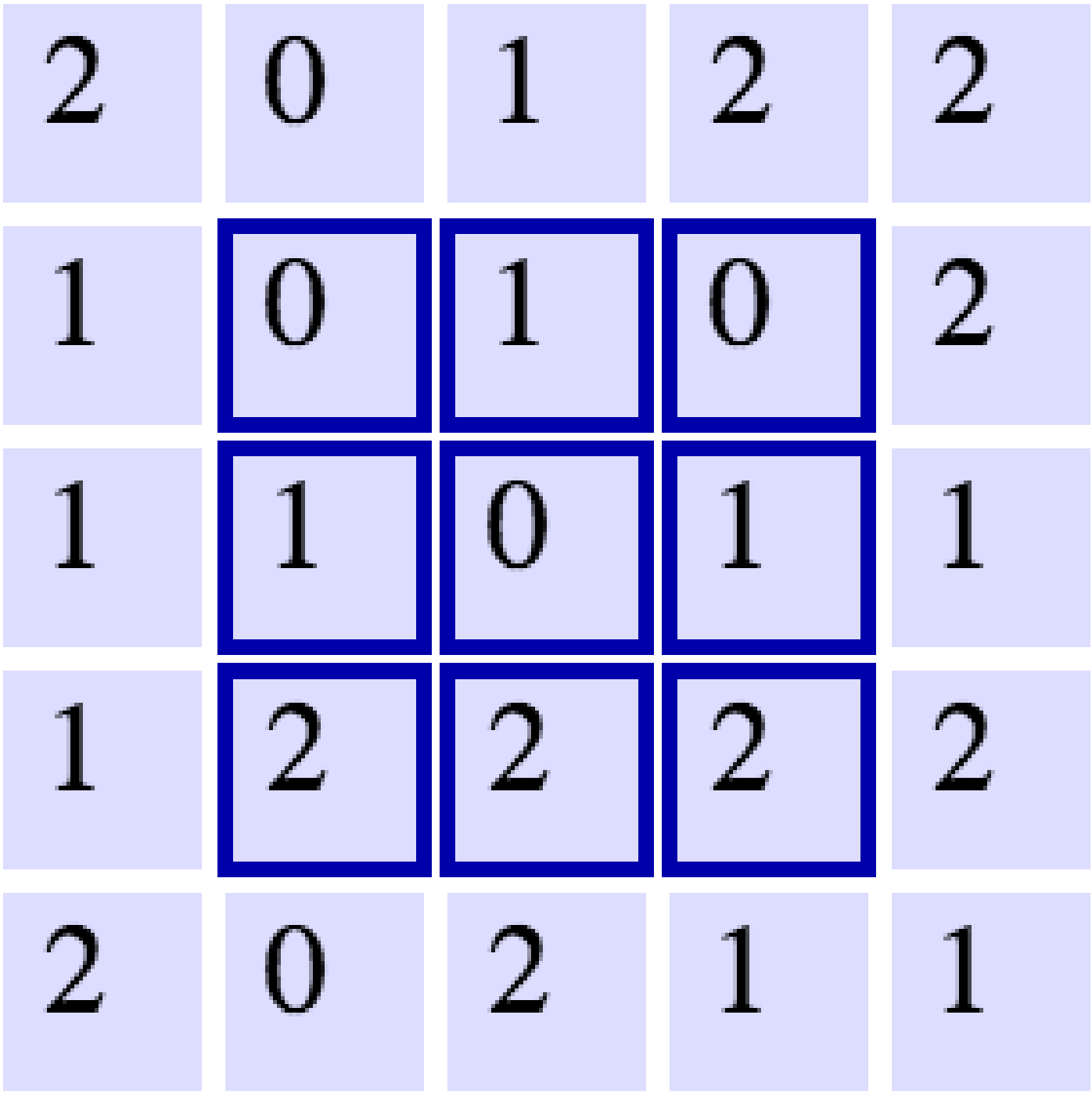
As we saw earlier, the classification of a given patch in an image is determined by the weights and biases corresponding to that patch.

If we want a cat that's in the top left patch to be classified in the same way as a cat in the bottom right patch, we need the weights and biases corresponding to those patches to be the same, so that they are classified the same way.

This is exactly what we do in CNNs. The weights and biases we learn for a given output layer are shared across all patches in a given input layer. Note that as we increase the depth of our filter, the number of weights and biases we have to learn still increases, as the weights aren't shared across the output channels.

There's an additional benefit to sharing our parameters. If we did not reuse the same weights across all patches, we would have to learn new parameters for every single patch and hidden layer neuron pair. This does not scale well, especially for higher fidelity images. Thus, sharing parameters not only helps us with translation invariance, but also gives us a smaller, more scalable model.

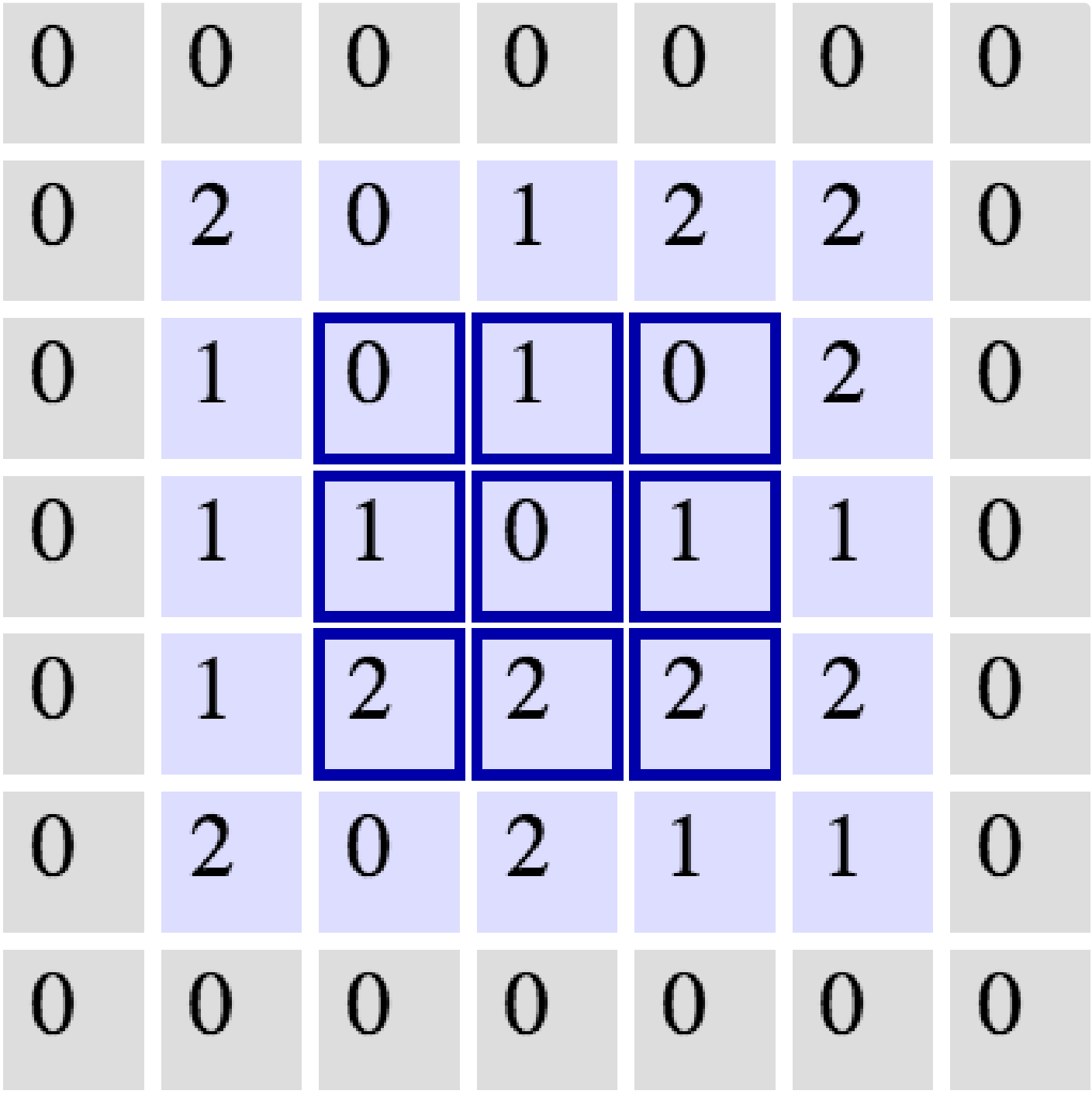
Padding



A 5x5 grid with a 3x3 filter. Source: Andrej Karpathy.

Let's say we have a `5x5` grid (as shown above) and a filter of size `3x3` with a stride of `1`. What's the width and height of the next layer? We see that we can fit at most three patches in each direction, giving us a dimension of `3x3` in our next layer. As we can see, the width and height of each subsequent layer decreases in such a scheme.

In an ideal world, we'd be able to maintain the same width and height across layers so that we can continue to add layers without worrying about the dimensionality shrinking and so that we have consistency. How might we achieve this? One way is to simple add a border of `0`s to our original `5x5` image. You can see what this looks like in the below image:



The same grid with 0 padding. Source: Andrej Karpathy.

This would expand our original image to a $7x7$. With this, we now see how our next layer's size is again a $5x5$, keeping our dimensionality consistent.

Visualizing CNNs

Let's look at an example CNN to see how it works in action.

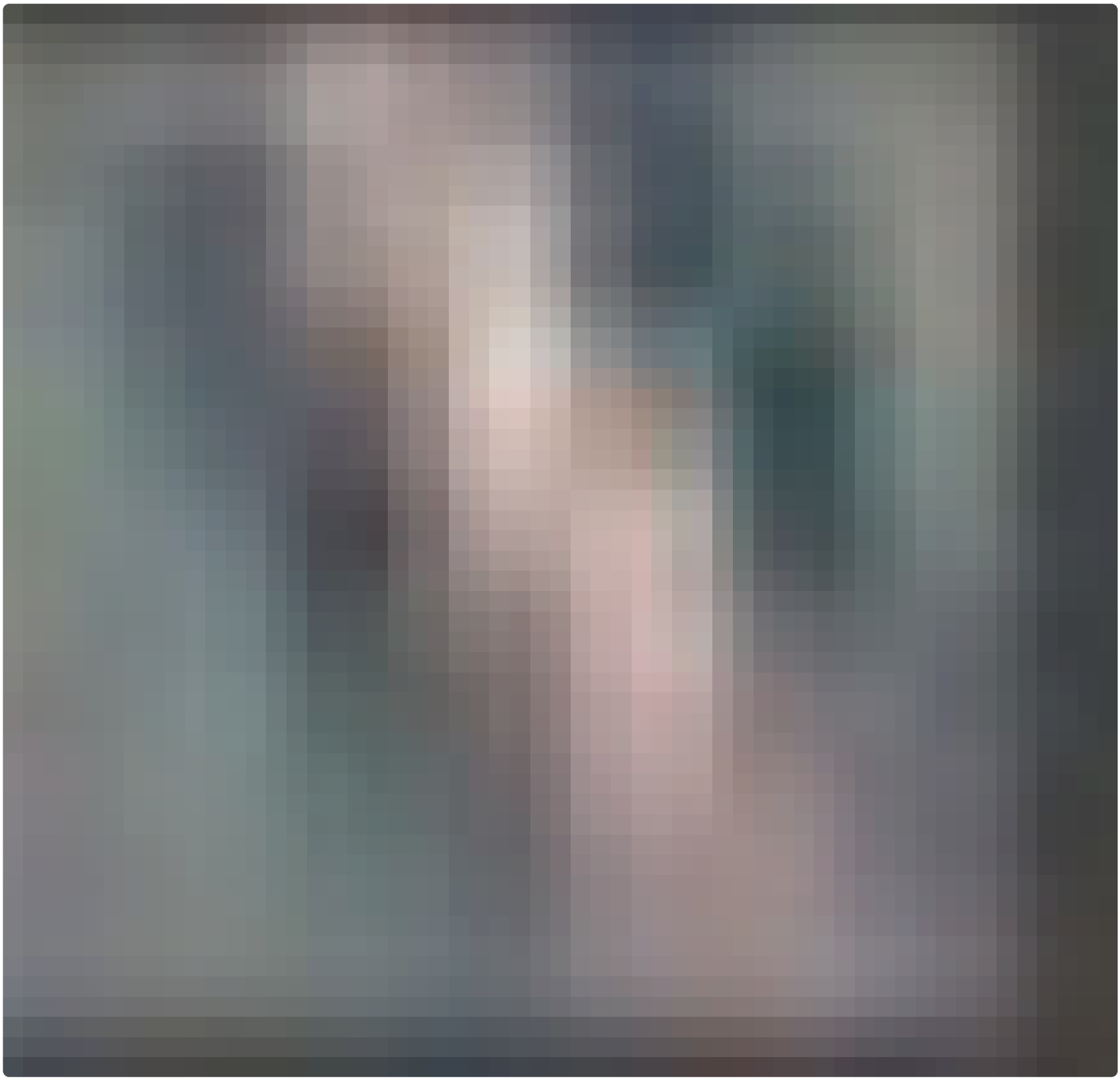
The CNN we will look at is trained on ImageNet as described in this paper by Zeiler and Fergus. In the images below (from the same paper), we'll see what each layer in this network detects and see *how* each layer detects more and more complex ideas.

Layer 1

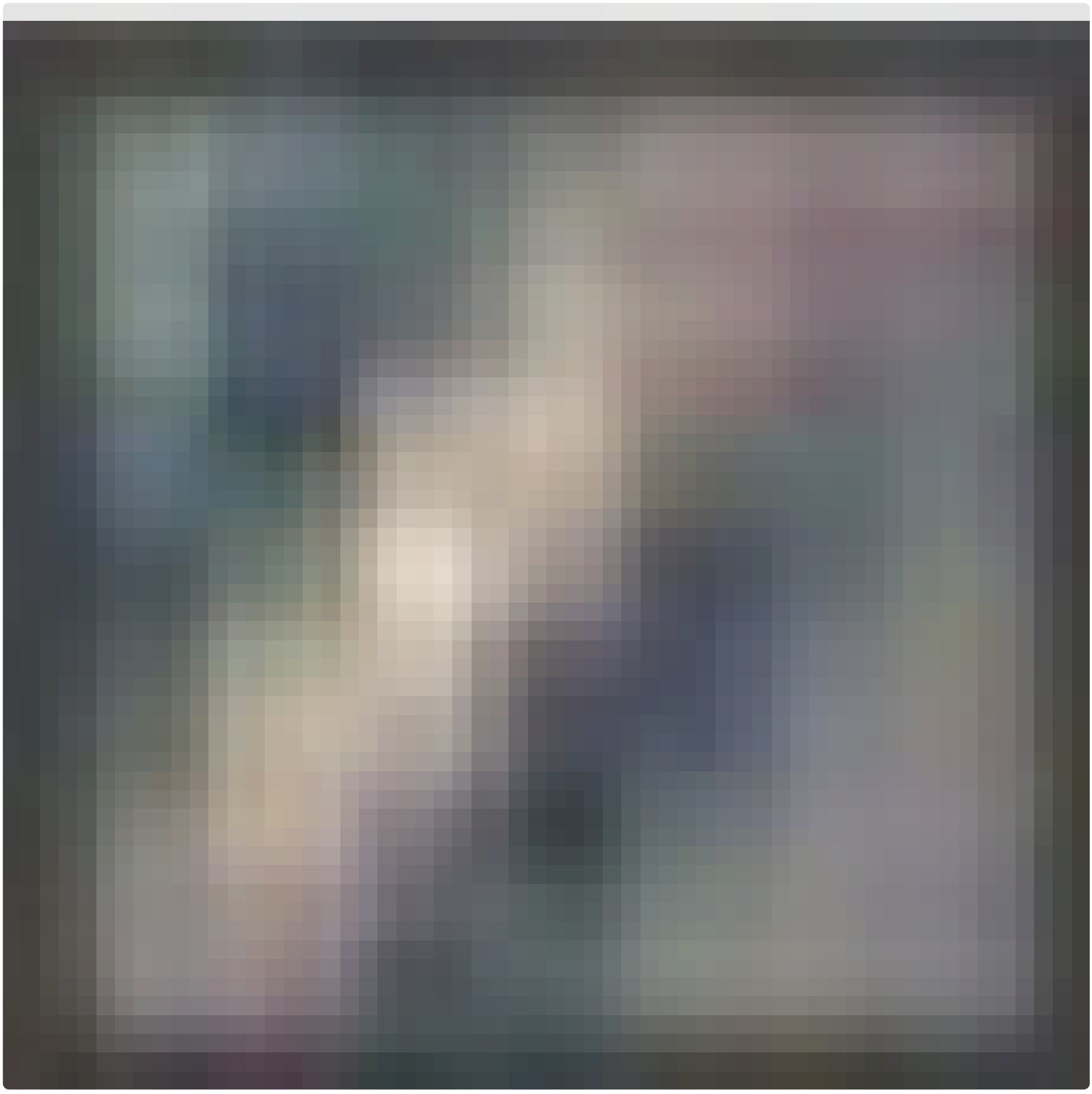
Example patterns that cause activations in the first layer of the network. These range from simple diagonal lines (top left) to green blobs (bottom middle).

The images above are from Matthew Zeiler and Rob Fergus' deep visualization toolbox, which lets us visualize what each layer in a CNN focuses on.

Each image in the above grid represents a pattern that causes the neurons in the first layer to activate - in other words, they are patterns that the first layer recognizes. The top left image shows a -45 degree line, while the middle top square shows a +45 degree line. These squares are shown below again for reference:

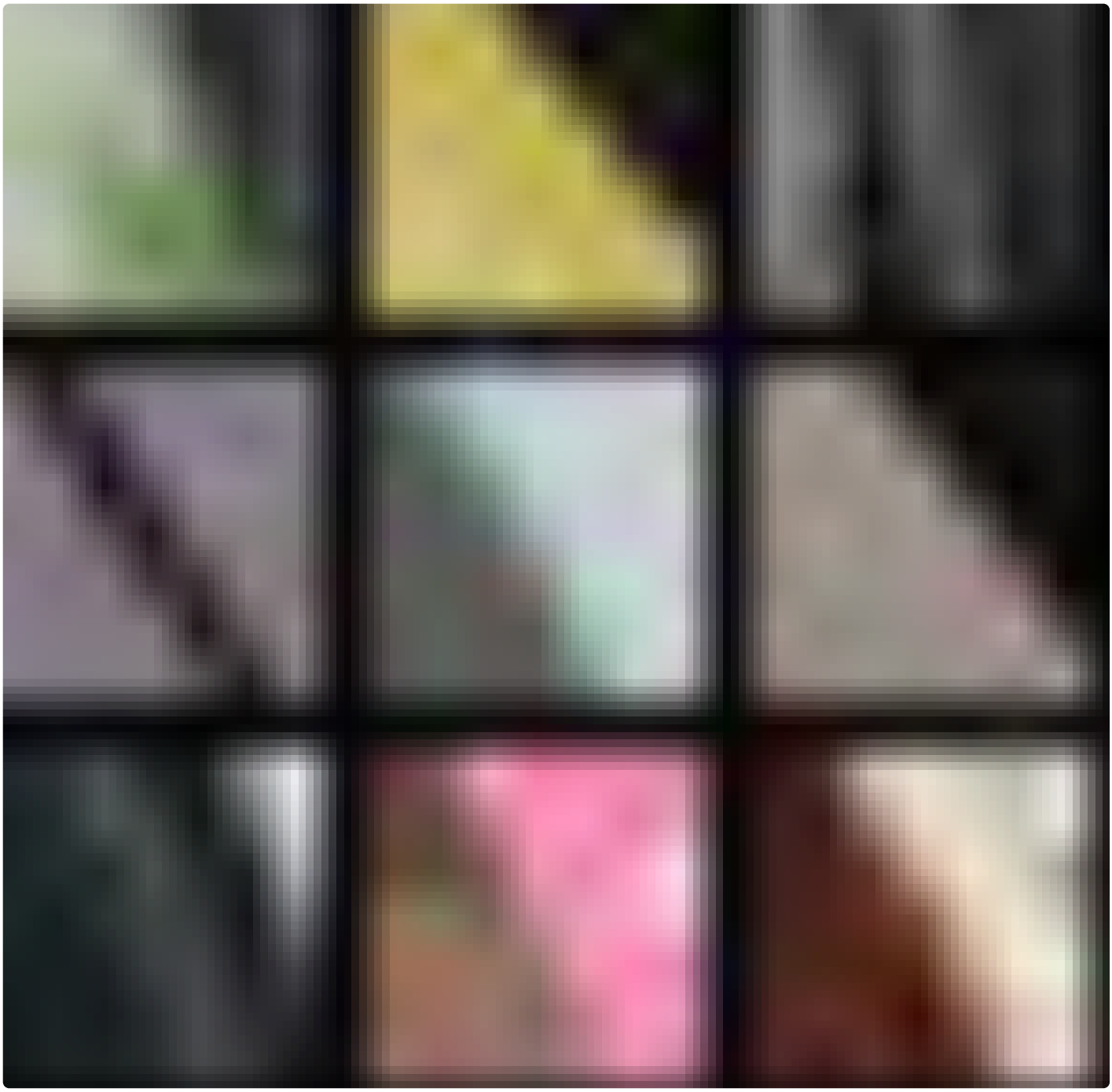


As visualized here, the first layer of the CNN can recognize -45 degree lines.



The first layer of the CNN is also able to recognize +45 degree lines, like the one above.

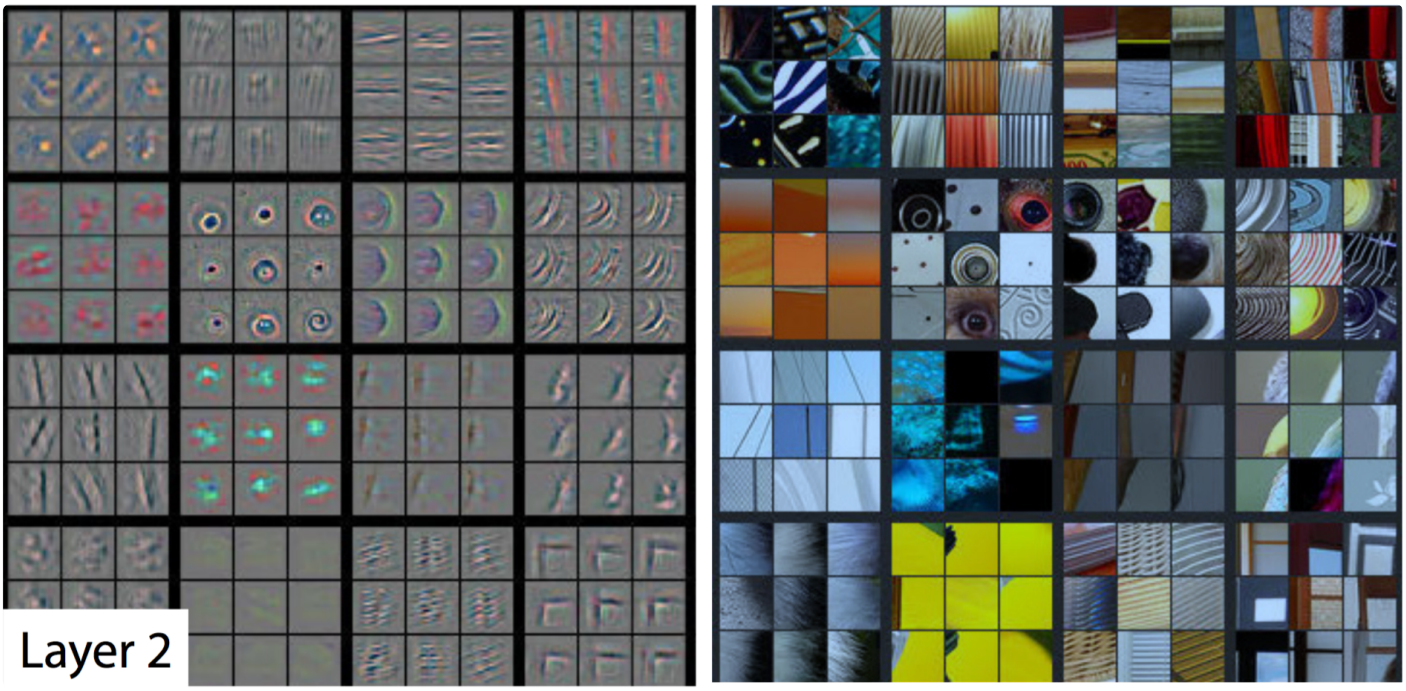
Let's now see some example images that cause such activations. The below grid of images all activated the -45 degree line. Notice how they are all selected despite the fact that they have different colors, gradients, and patterns.



Example patches that activate the -45 degree line detector in the first layer.

So, the first layer of our CNN clearly picks out very simple shapes and patterns like lines and blobs.

Layer 2



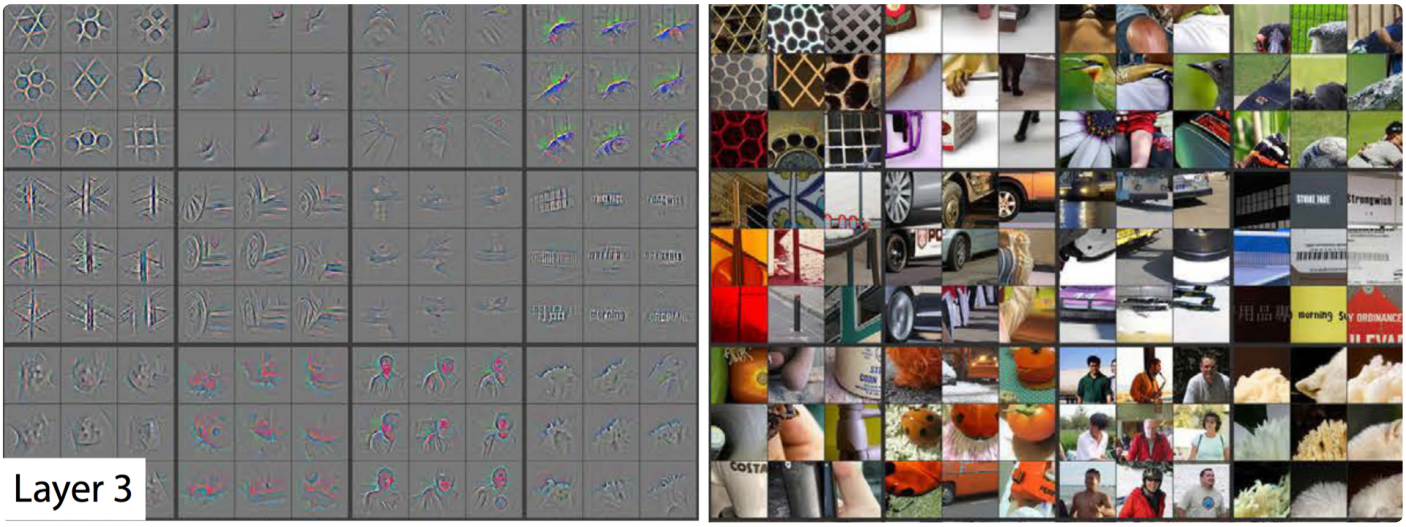
A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The second layer of the CNN captures complex ideas.

As you see in the image above, the second layer of the CNN recognizes circles (second row, second column), stripes (first row, second column), and rectangles (bottom right).

The CNN learns to do this on its own. There is no special instruction for the CNN to focus on more complex objects in deeper layers. That's just how it normally works out when you feed training data into a CNN.

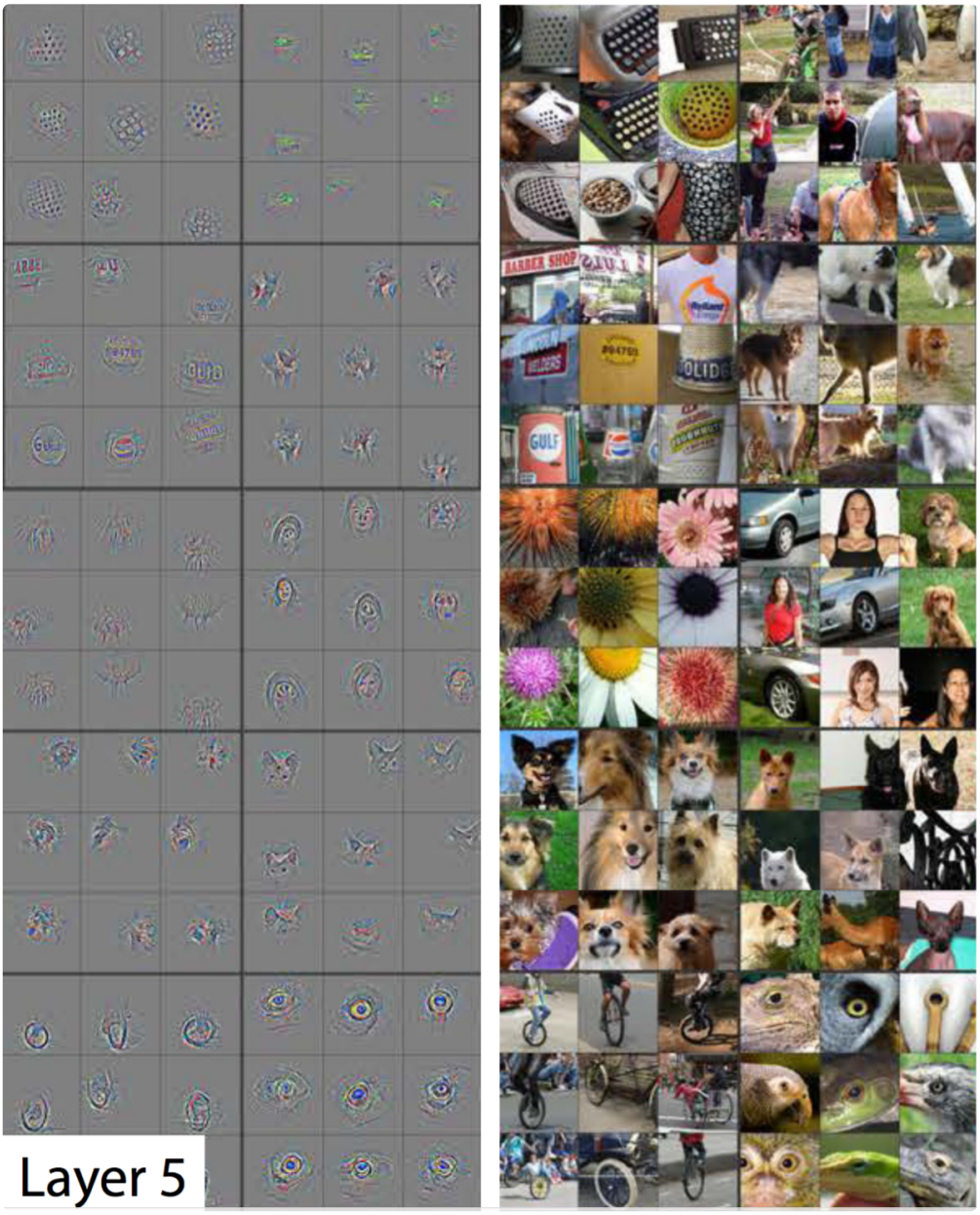
Layer 3



A visualization of the third layer in the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The third layer picks out complex combinations of features from the second layer. These include things like grids, and honeycombs (top left), wheels (second row, second column), and even faces (third row, third column).

Layer 5

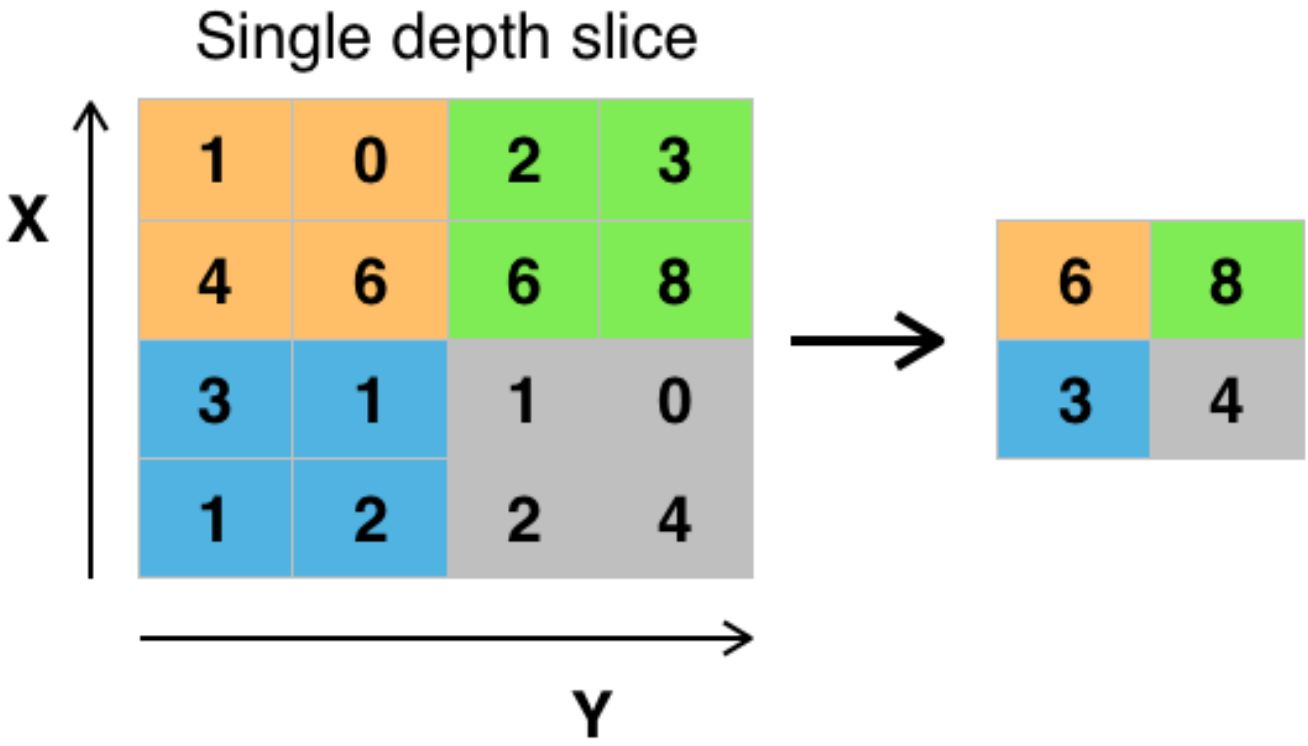


A visualization of the fifth and final layer of the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

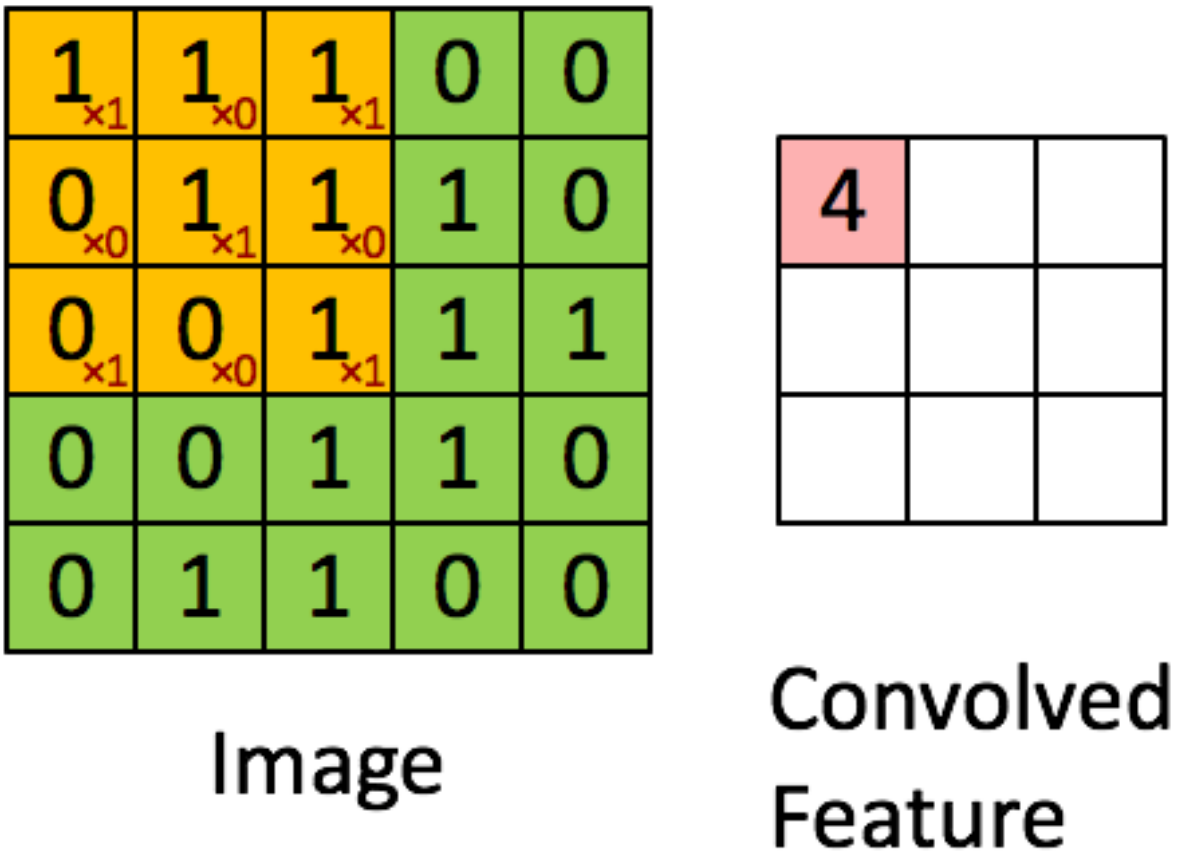
We'll skip layer 4, which continues this progression, and jump right to the fifth and final layer of this CNN.

The last layer picks out the highest order ideas that we care about for classification, like dog faces, bird faces, and bicycles.

Max Pooling



Convolutions



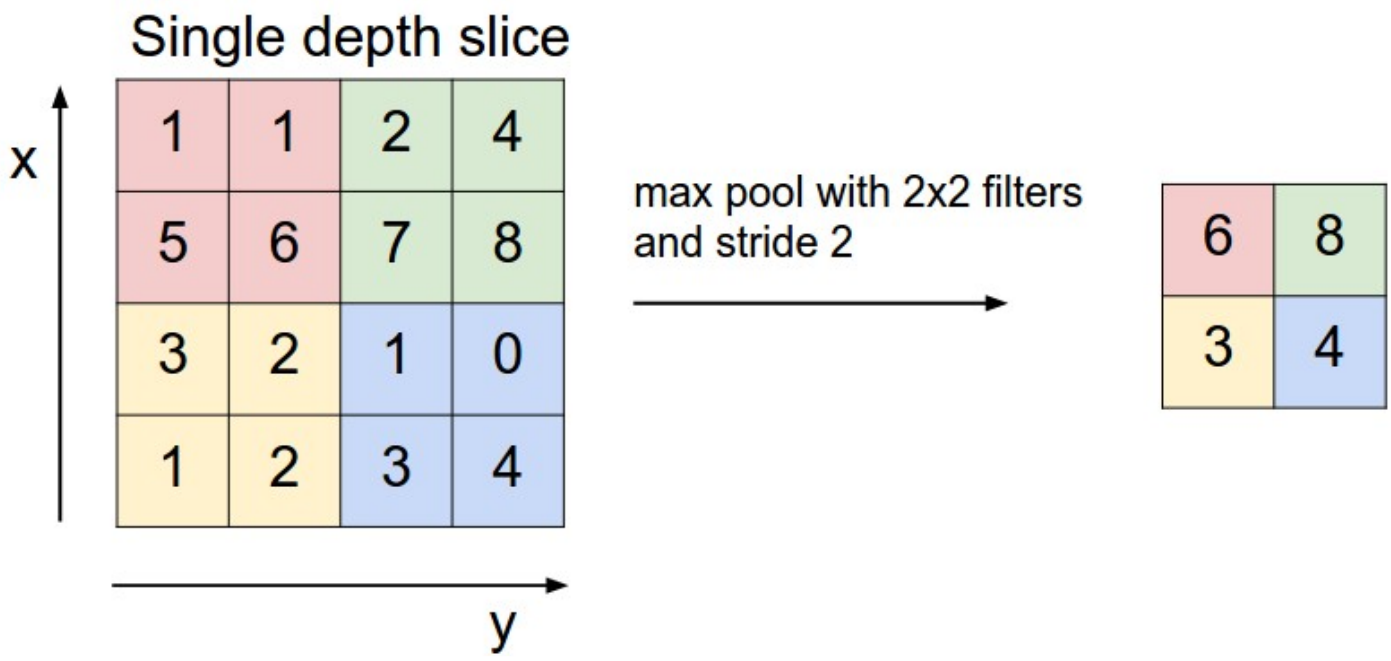
Convolution with 3×3 Filter. Source:

http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

The above is an example of a convolution with a 3x3 filter and a stride of 1 being applied to data with a range of 0 to 1. The convolution for each 3x3 section is calculated against the weight, `[[1, 0, 1], [0, 1, 0], [1, 0, 1]]`, then a bias is added to create the convolved feature on the right. In this case, the bias is zero.

Stride is an array of 4 elements; the first element in the stride array indicates the stride for batch and last element indicates stride for features. It's good practice to remove the batches or features you want to skip from the data set rather than use stride to skip them. You can always set the first and last element to 1 in stride in order to use all batches and features.

Max Pooling



Max Pooling with 2x2 filter and stride of 2. Source: http://cs231n.github.io/convolutional-networks/

The above is an example of max pooling with a 2x2 filter and stride of 2. The left square is the input and the right square is the output. The four 2x2 colors in input represents each time the filter was applied to create the max on the right side. For example, `[[1, 1], [5, 6]]` becomes `6` and `[[3, 2], [1, 2]]` becomes `3`.

Practical Intuition with Code Explanation

In this notebook, we train a Convolutional Neural Network to classify images from the MNIST database.

1. Load MNIST Database

MNIST is one of the most famous datasets in the field of machine learning.

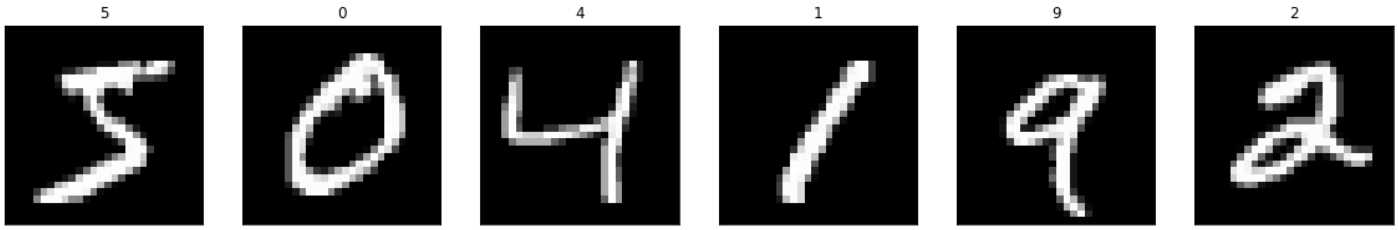
- It has 70,000 images of hand-written digits
- Very straight forward to download
- Images dimensions are 28x28
- Grayscale images

```
from tensorflow.keras.datasets import mnist

# use Keras to import pre-shuffled MNIST database
(X_train, y_train), (X_test, y_test) = mnist.load_data()

print("The MNIST database has a training set of %d examples." % len(X_train))
print("The MNIST database has a test set of %d examples." % len(X_test))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
The MNIST database has a training set of 60000 examples.
The MNIST database has a test set of 10000 examples.
```

2. Visualize the First Six Training Images

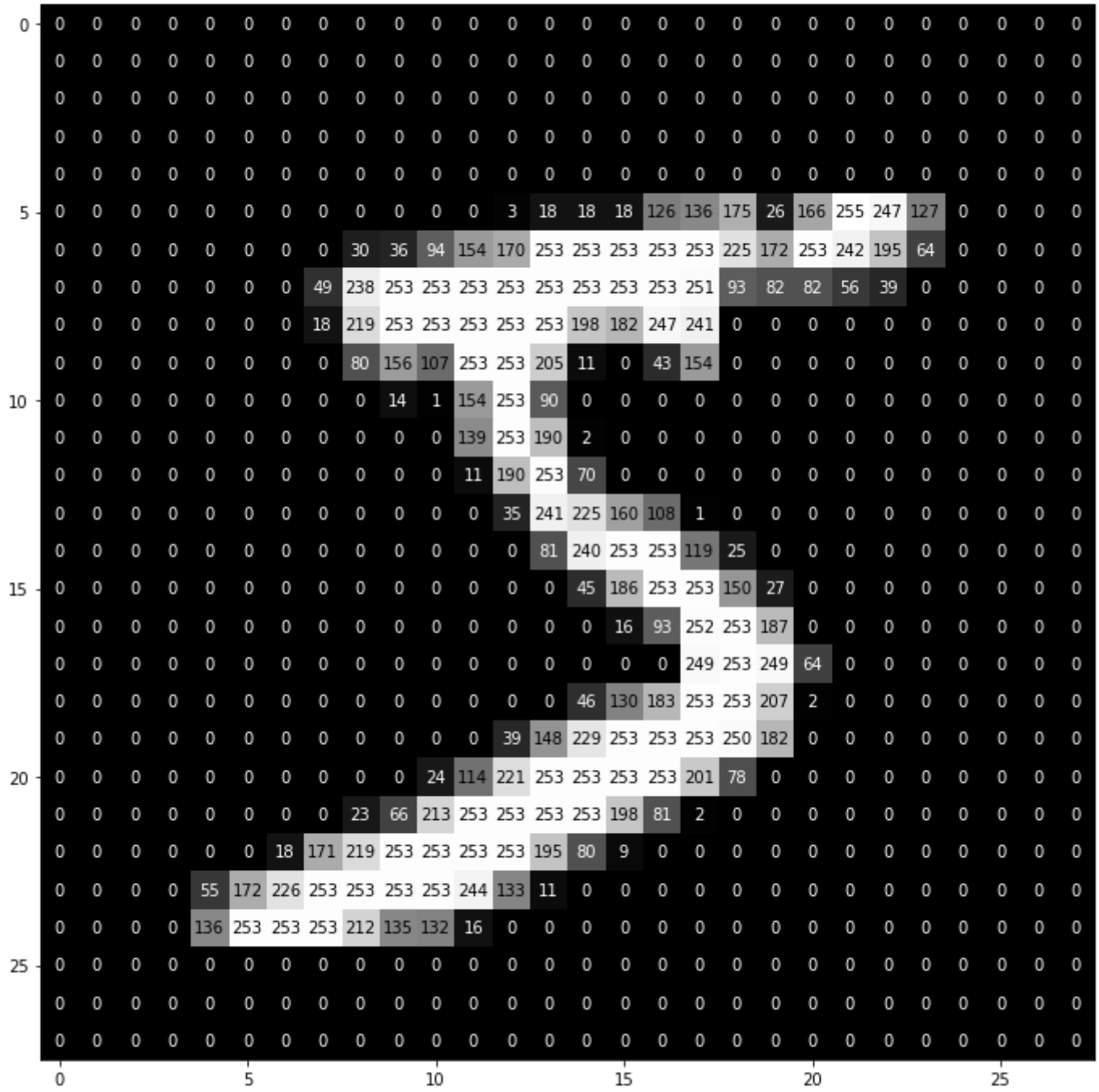
```
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib.cm as cm
import numpy as np

# plot first six training images
fig = plt.figure(figsize=(20,20))
for i in range(6):
    ax = fig.add_subplot(1, 6, i+1, xticks=[], yticks=[])
    ax.imshow(X_train[i], cmap='gray')
    ax.set_title(str(y_train[i]))
```



3. View an Image in More Detail

```
def visualize_input(img, ax):
    ax.imshow(img, cmap='gray')
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            ax.annotate(str(round(img[x][y],2)), xy=(y,x),
                        horizontalalignment='center',
                        verticalalignment='center',
                        color='white' if img[x][y]<thresh else 'black')

fig = plt.figure(figsize = (12,12))
```

```
ax = fig.add_subplot(111)
visualize_input(X_train[0], ax)
```



4. Preprocess input images: Rescale the Images by Dividing Every Pixel in Every Image by 255

```
# rescale to have values within 0 - 1 range [0,255] --> [0,1]
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255

print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
X_train shape: (60000, 28, 28)
60000 train samples
```

```
10000 test samples
```

5. Preprocess the labels: Encode Categorical Integer Labels Using a One-Hot Scheme

```
from keras.utils import np_utils

num_classes = 10
# print first ten (integer-valued) training labels
print('Integer-valued labels:')
print(y_train[:10])

# one-hot encode the labels
# convert class vectors to binary class matrices
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

# print first ten (one-hot) training labels
print('One-hot labels:')
print(y_train[:10])
```

```
Integer-valued labels:
[5 0 4 1 9 2 1 3 1 4]
One-hot labels:
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

6. Reshape data to fit our CNN (and input_shape)

```
# input image dimensions 28x28 pixel images.
img_rows, img_cols = 28, 28

X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)

print('input_shape: ', input_shape)
print('x_train shape:', X_train.shape)
```

```
input_shape:  (28, 28, 1)
```

```
x_train shape: (60000, 28, 28, 1)
```

7. Define the Model Architecture

You must pass the following arguments:

- filters - The number of filters.
- kernel_size - Number specifying both the height and width of the (square) convolution window.

There are some additional, optional arguments that you might like to tune:

- strides - The stride of the convolution. If you don't specify anything, strides is set to 1.
- padding - One of 'valid' or 'same'. If you don't specify anything, padding is set to 'valid'.
- activation - Typically 'relu'. If you don't specify anything, no activation is applied. You are strongly encouraged to add a ReLU activation function to every convolutional layer in your networks.

** Things to remember **

- Always add a ReLU activation function to the **Conv2D** layers in your CNN. With the exception of the final layer in the network, Dense layers should also have a ReLU activation function.
- When constructing a network for classification, the final layer in the network should be a **Dense** layer with a softmax activation function. The number of nodes in the final layer should equal the total number of classes in the dataset.

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

# build the model object
model = Sequential()

# CONV_1: add CONV layer with RELU activation and depth = 32 kernels
model.add(Conv2D(32, kernel_size=(3, 3), padding='same',activation='relu',input_shape=(
# POOL_1: downsample the image to choose the best features
model.add(MaxPooling2D(pool_size=(2, 2)))

# CONV_2: here we increase the depth to 64
model.add(Conv2D(64, (3, 3),padding='same', activation='relu'))
# POOL_2: more downsampling
model.add(MaxPooling2D(pool_size=(2, 2)))

# flatten since too many dimensions, we only want a classification output
model.add(Flatten())

# FC_1: fully connected to get all relevant data
model.add(Dense(64, activation='relu'))

# FC_2: output a softmax to squash the matrix into output probabilities for the 10 clas
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_32 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 7, 7, 64)          0
_________________________________________________________________
flatten_9 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_18 (Dense)             (None, 64)                200768
_________________________________________________________________
dense_19 (Dense)             (None, 10)                650
=================================================================
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
_________________________________________________________________
```

Things to notice:

- The network begins with a sequence of two convolutional layers, followed by max pooling layers.
- The final layer has one entry for each object class in the dataset, and has a softmax activation function, so that it returns probabilities.
- The Conv2D depth increases from the input layer of 1 to 32 to 64.
- We also want to decrease the height and width - This is where maxpooling comes in. Notice that the image dimensions decrease from 28 to 14 after the pooling layer.
- You can see that every output shape has **None** in place of the batch-size. This is so as to facilitate changing of batch size at runtime.
- Finally, we add one or more fully connected layers to determine what object is contained in the image. For instance, if wheels were found in the last max pooling layer, this FC layer will transform that information to predict that a car is present in the image with higher probability. If there were eyes, legs and a tails, then this could mean that there is a dog in the image.

8. Compile the Model

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

9. Train the Model

```
from tensorflow.keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
                               save_best_only=True)
hist = model.fit(X_train, y_train, batch_size=64, epochs=10,
          validation_data=(X_test, y_test), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

```
Epoch 1/10
938/938 - 35s - loss: 0.1542 - accuracy: 0.9514 - val_loss: 0.0729 - val_accuracy:
0.9753

Epoch 00001: val_loss improved from inf to 0.07286, saving model to
model.weights.best.hdf5
Epoch 2/10
938/938 - 3s - loss: 0.0456 - accuracy: 0.9859 - val_loss: 0.0345 - val_accuracy:
0.9894

Epoch 00002: val_loss improved from 0.07286 to 0.03452, saving model to
model.weights.best.hdf5
Epoch 3/10
938/938 - 3s - loss: 0.0319 - accuracy: 0.9899 - val_loss: 0.0286 - val_accuracy:
0.9910

Epoch 00003: val_loss improved from 0.03452 to 0.02864, saving model to
model.weights.best.hdf5
Epoch 4/10
938/938 - 3s - loss: 0.0227 - accuracy: 0.9929 - val_loss: 0.0275 - val_accuracy:
0.9906

Epoch 00004: val_loss improved from 0.02864 to 0.02747, saving model to
model.weights.best.hdf5
Epoch 5/10
938/938 - 3s - loss: 0.0181 - accuracy: 0.9945 - val_loss: 0.0297 - val_accuracy:
0.9914

Epoch 00005: val_loss did not improve from 0.02747
Epoch 6/10
938/938 - 3s - loss: 0.0144 - accuracy: 0.9958 - val_loss: 0.0346 - val_accuracy:
0.9903
```

```
Epoch 00006: val_loss did not improve from 0.02747
Epoch 7/10
938/938 - 3s - loss: 0.0117 - accuracy: 0.9965 - val_loss: 0.0281 - val_accuracy:
0.9928

Epoch 00007: val_loss did not improve from 0.02747
Epoch 8/10
938/938 - 3s - loss: 0.0086 - accuracy: 0.9974 - val_loss: 0.0398 - val_accuracy:
0.9897

Epoch 00008: val_loss did not improve from 0.02747
Epoch 9/10
938/938 - 3s - loss: 0.0078 - accuracy: 0.9978 - val_loss: 0.0334 - val_accuracy:
0.9908

Epoch 00009: val_loss did not improve from 0.02747
Epoch 10/10
938/938 - 3s - loss: 0.0061 - accuracy: 0.9983 - val_loss: 0.0355 - val_accuracy:
0.9927

Epoch 00010: val_loss did not improve from 0.02747
```

10. Load the Model with the Best Classification Accuracy on the Validation Set

```
# load the weights that yielded the best validation accuracy
model.load_weights('model.weights.best.hdf5')
```

11. Calculate the Classification Accuracy on the Test Set

```
# evaluate test accuracy
score = model.evaluate(X_test, y_test, verbose=0)
accuracy = 100*score[1]

# print test accuracy
print('Test accuracy: %.4f%%' % accuracy)
```

```
Test accuracy: 99.2000%
```

CNNs will truly OUTPERFORM MLPs.

Lets Work with RGB Images

Here we will train a CNN to classify images from the CIFAR-10 dataset.

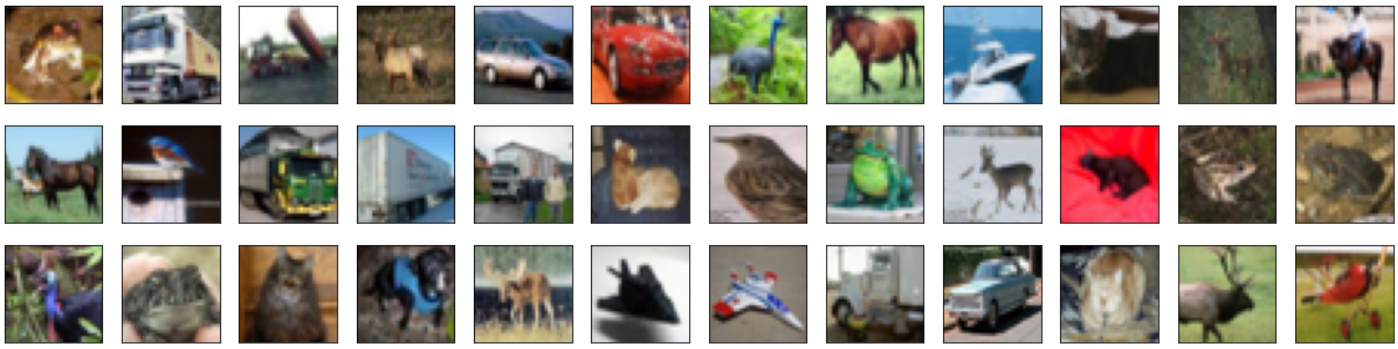
1. Load CIFAR-10 Database

```
import keras
from keras.datasets import cifar10

# load the pre-shuffled train and test data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 13s 0us/step
```

2. Visualize the First 24 Training Images

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```



3. Rescale the Images by Dividing Every Pixel in Every Image by 255

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure below shows Gradient Descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

** Tip: ** When using Gradient Descent, you should ensure that all features have a similar scale to speed up training or else it will take much longer to converge.

```
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

4. Break Dataset into Training, Testing, and Validation Sets

```
from keras.utils import np_utils
from tensorflow import keras
```

```
# one-hot encode the labels
num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

# print shape of training set
print('x_train shape:', x_train.shape)

# print number of training, validation, and test images
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```

```
x_train shape: (45000, 32, 32, 3)
45000 train samples
10000 test samples
5000 validation samples
```

5. Define the Model Architecture

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

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=3, padding='same', activation='relu',
                         input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d (Conv2D)              (None, 32, 32, 16)        448

 max_pooling2d (MaxPooling2D  (None, 16, 16, 16)       0
```

```
)

conv2d_1 (Conv2D)            (None, 16, 16, 32)        4640

max_pooling2d_1 (MaxPooling  (None, 8, 8, 32)          0
2D)

conv2d_2 (Conv2D)            (None, 8, 8, 64)          18496

max_pooling2d_2 (MaxPooling  (None, 4, 4, 64)          0
2D)

dropout (Dropout)            (None, 4, 4, 64)          0

flatten (Flatten)            (None, 1024)              0

dense (Dense)                (None, 500)               512500

dropout_1 (Dropout)          (None, 500)               0

dense_1 (Dense)              (None, 10)                5010

=================================================================
Total params: 541,094
Trainable params: 541,094
Non-trainable params: 0
_________________________________________________________________
```

6. Compile the Model

```
model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'
```

7. Train the Model

```
from keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1, save_best

hist = model.fit(x_train, y_train, batch_size=32, epochs=1,
          validation_data=(x_valid, y_valid), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

```
Epoch 1: val_loss improved from inf to 1.62405, saving model to model.weights.best.hdf5
```

```
1407/1407 - 17s - loss: 1.5797 - accuracy: 0.4261 - val_loss: 1.6241 - val_accuracy:
0.4320 - 17s/epoch - 12ms/step
```

8. Load the Model with the Best Validation Accuracy

```
model.load_weights('model.weights.best.hdf5')
```

10. Visualize Some Predictions

```
# get predictions on the test set
y_hat = model.predict(x_test)

# define text labels (source: https://www.cs.toronto.edu/~kriz/cifar.html)
cifar10_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'hors
```

```
# plot a random sample of test images, their predicted labels, and ground truth
fig = plt.figure(figsize=(20, 8))
for i, idx in enumerate(np.random.choice(x_test.shape[0], size=32, replace=False)):
    ax = fig.add_subplot(4, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_test[idx]))
    pred_idx = np.argmax(y_hat[idx])
    true_idx = np.argmax(y_test[idx])
    ax.set_title("{} ({})".format(cifar10_labels[pred_idx], cifar10_labels[true_idx]),
                 color=("blue" if pred_idx == true_idx else "red"))
```