CS-618
Deep Learning
Spring 2023

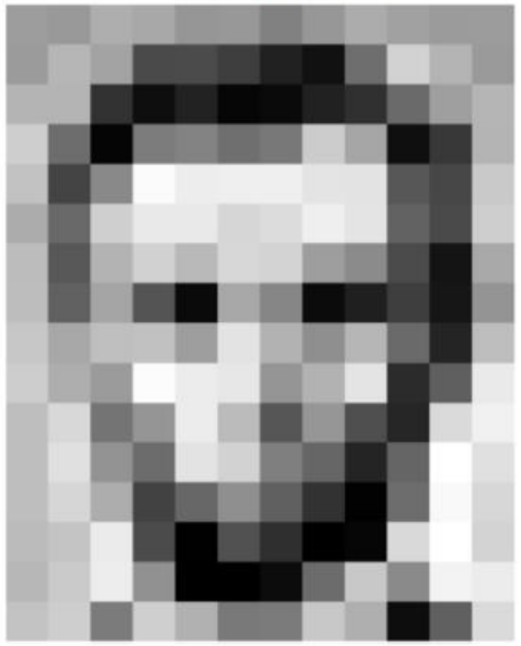
Lecture 4

Agenda

- The building blocks of CNNs
    - Understanding CNNs and feature hierarchies
    - Performing discrete convolutions
        - Discrete convolutions in one dimension
        - Padding inputs to control the size of the output feature maps
        - Determining the size of the convolution output
        - Performing a discrete convolution in 2D
    - Subsampling layers
    - Putting everything together – implementing a CNN
        - Working with multiple input or color channels
        - Regularizing an NN with dropout
    - Loss functions for classification
- Implementing a deep CNN using TensorFlow
    - The multilayer CNN architecture
    - Loading and preprocessing the data
    - Implementing a CNN using the TensorFlow Keras API
        - Configuring CNN layers in Keras
        - Constructing a CNN in Keras
    - Gender classification from face images using a CNN
        - Loading the CelebA dataset
        - Image transformation and data augmentation
        - Training a CNN gender classifier
- Summary

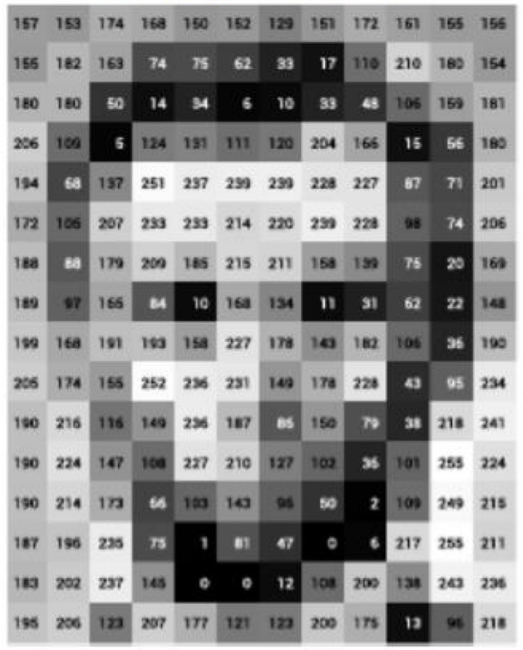
What computers see: Images as Numbers

What you see



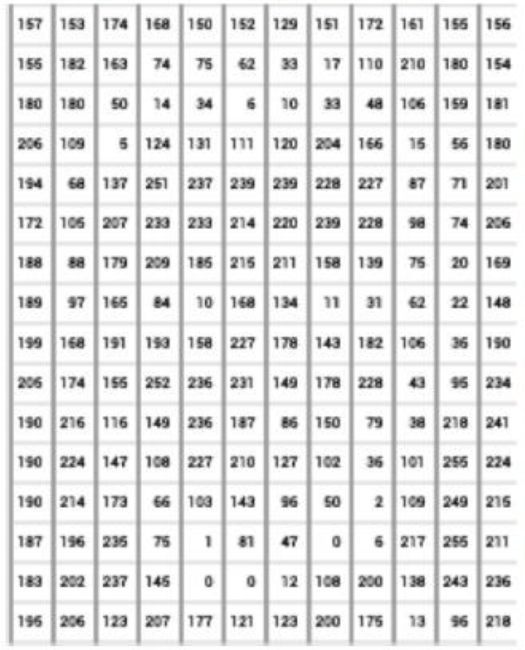
Input Image

What you both see



Input Image + values

What the computer "sees"



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Levin *Image Processing & Computer Vision*

Pixel intensity values
(“pix-el”=picture-element)

How can computers recognize objects?

Ad Sign
Traffic Sign
Road Sign
Tram Train
Person
Bus
Person
Car
Car
Bicycle
Bicycle

Convolutional Neural Networks (CNN)

- Convolution operations in one and two dimensions
- The building blocks of CNN architectures
- Implementing deep CNNs in TensorFlow
- Data augmentation techniques for improving the generalization performance
- Implementing a face image-based CNN classifier for predicting the gender of a person

CNN

Convolutional Neural Networks (CNNs) are a type of neural network that are commonly used for image recognition, classification, and other computer vision tasks. They are inspired by the structure and function of the visual cortex in the brain, which processes visual information by detecting patterns and edges in the input.

What does CNN consist of?

At a high level, a CNN consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. The input to the network is an image, which is represented as a matrix of pixel values. The first layer of the network is typically a convolutional layer, which applies a set of learnable filters to the input image to extract features such as edges, corners, and other visual patterns.

CNN process (1of 3)

The output of the convolutional layer is then passed through a pooling layer, which reduces the spatial size of the feature maps by taking the maximum, average, or other function of small groups of adjacent pixels. This helps to make the representation more robust to small shifts and variations in the input.

CNN process (2of 3)

The process of applying convolutional and pooling layers is typically repeated multiple times, with the number of filters and the size of the pooling window increasing in each layer to capture more complex and abstract features.

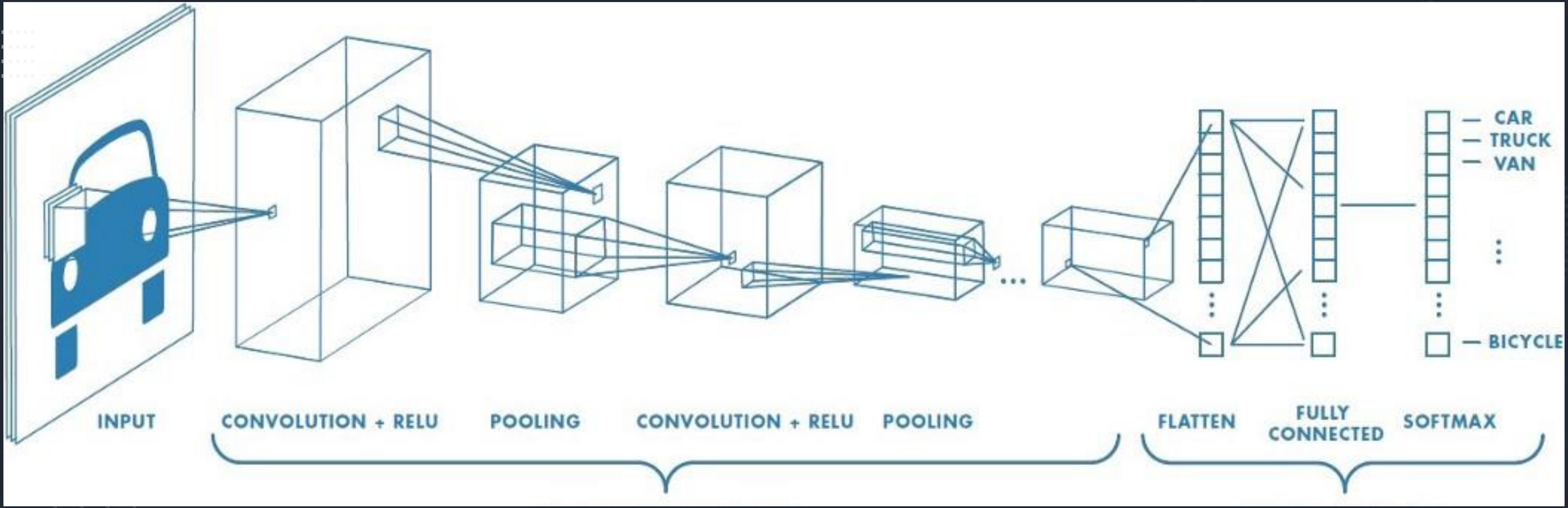
CNN process (3of 3)

The final layer of the network is typically a fully connected layer, which takes the flattened output of the last convolutional/pooling layer and uses it to make a prediction about the input image. This can involve classifying the image into one of several categories, or assigning a score to each category based on how likely the image is to belong to that category.

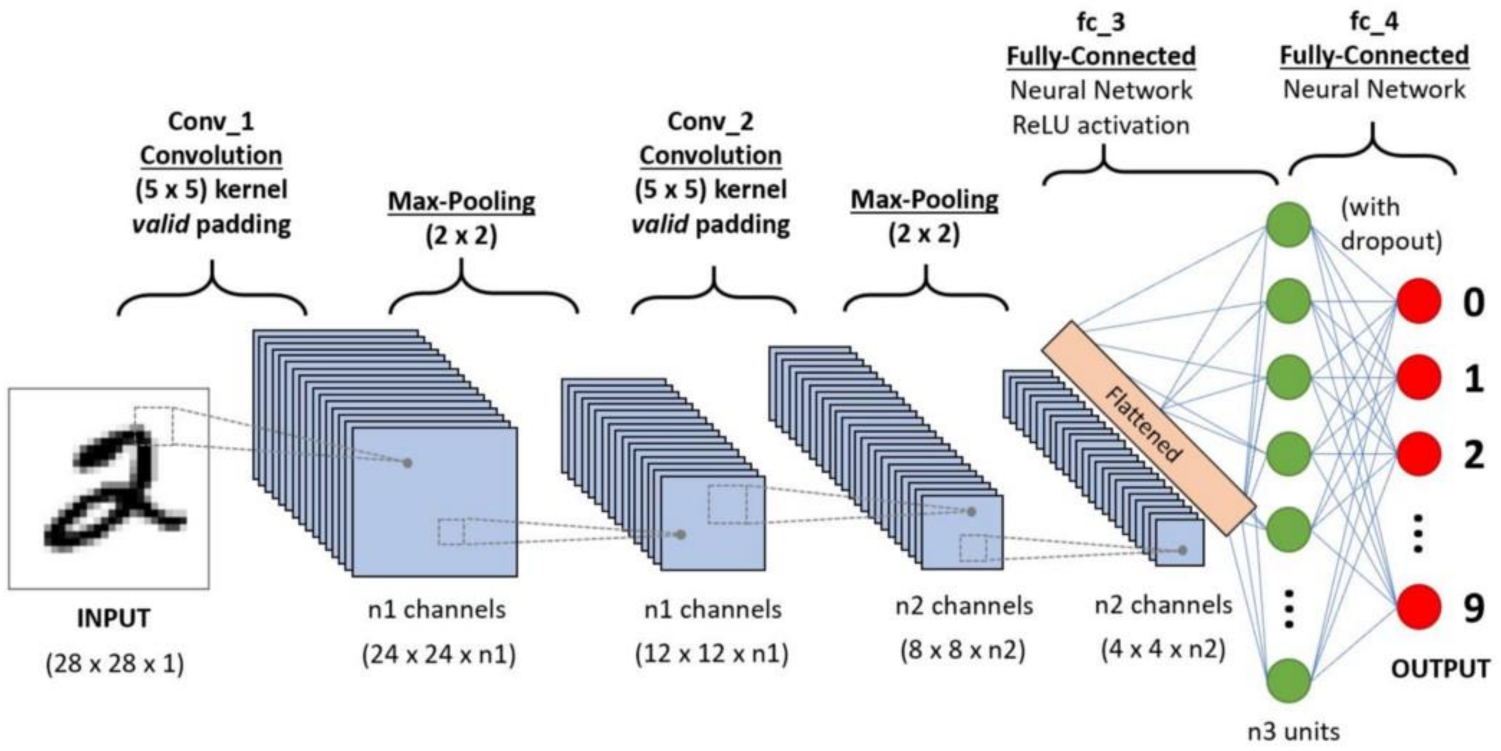
CNNs are trained using backpropagation, where the weights of the filters and fully connected layers are adjusted to minimize the difference between the predicted output and the actual output. This is typically done using a loss function such as cross-entropy or mean squared error, and an optimization algorithm such as stochastic gradient descent.

Architecture of CNN

- A typical CNN has 4 layers
    - Input layer
    - Convolution layer
    - Pooling layer
    - Fully connected layer



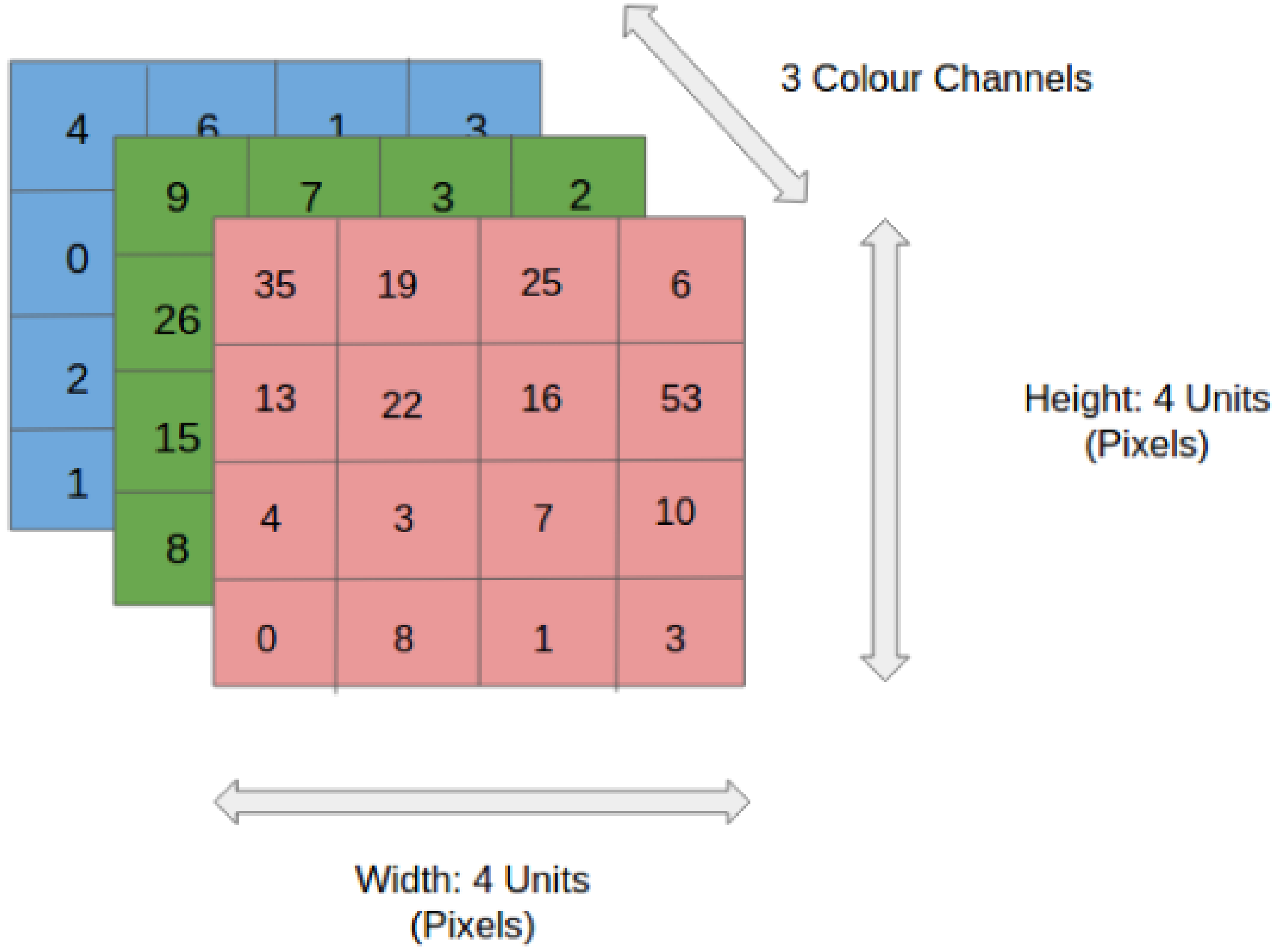
Introduction



A **Convolutional Neural Network (ConvNet/CNN)**

- is a Deep Learning algorithm that can take in an input image,
- assign importance (learnable weights and biases) to various aspects/objects in the image
- be able to differentiate one from the other.
- The pre-processing required in a ConvNet is much lower as compared to other classification algorithms.
- While in primitive methods filters are hand-engineered, with enough training
- ConvNets have the ability to learn these filters/characteristics.

Input Image



Source: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Convolution Layer — The Kernel

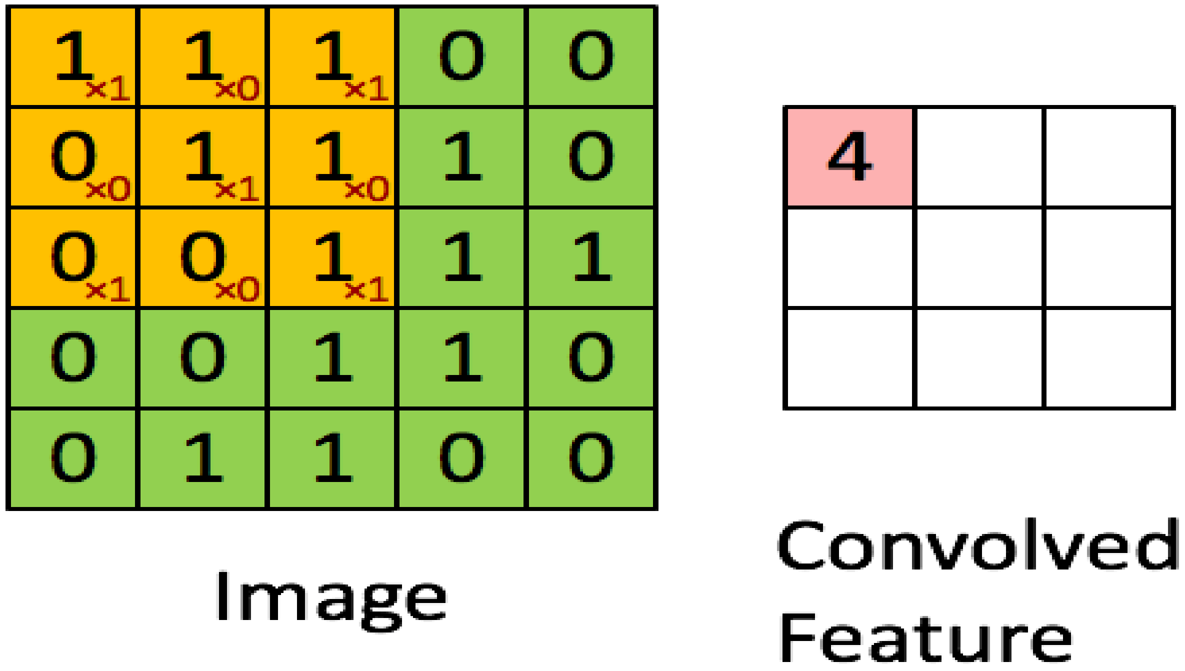
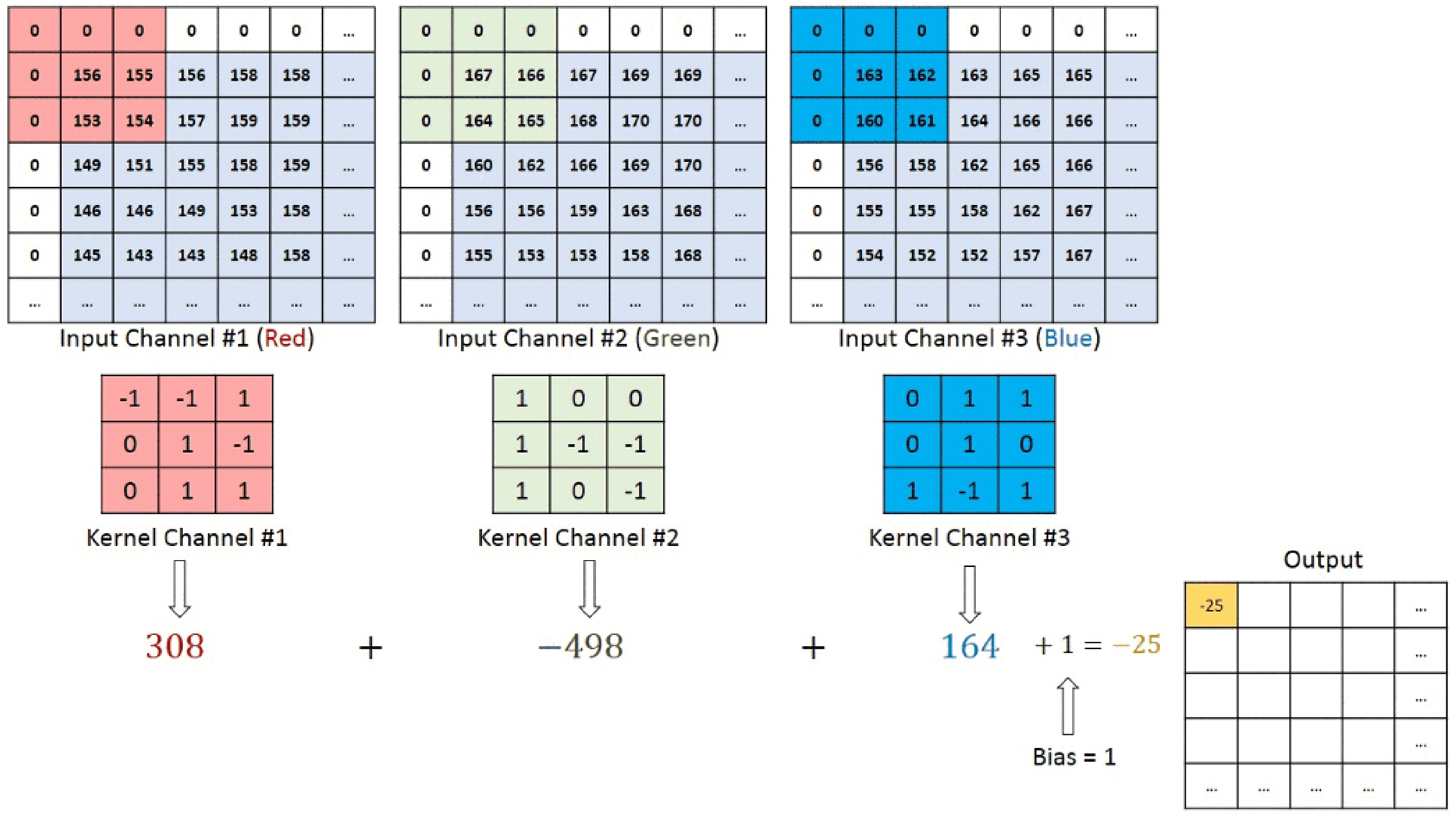


Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB)
In the above demonstration, the green section resembles our **5x5x1 input image, I**. The element involved in the convolution operation in the first part of a Convolutional Layer is called the **Kernel/Filter, K**, represented in color yellow. We have selected **K as a 3x3x1 matrix.**


Input Channel #1 (Red)

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Kernel Channel #1

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

Kernel Channel #2

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

Kernel Channel #3

0	1	1
0	1	0
1	-1	1

$$308 + -498 + 164 + 1 = -25$$

Bias = 1

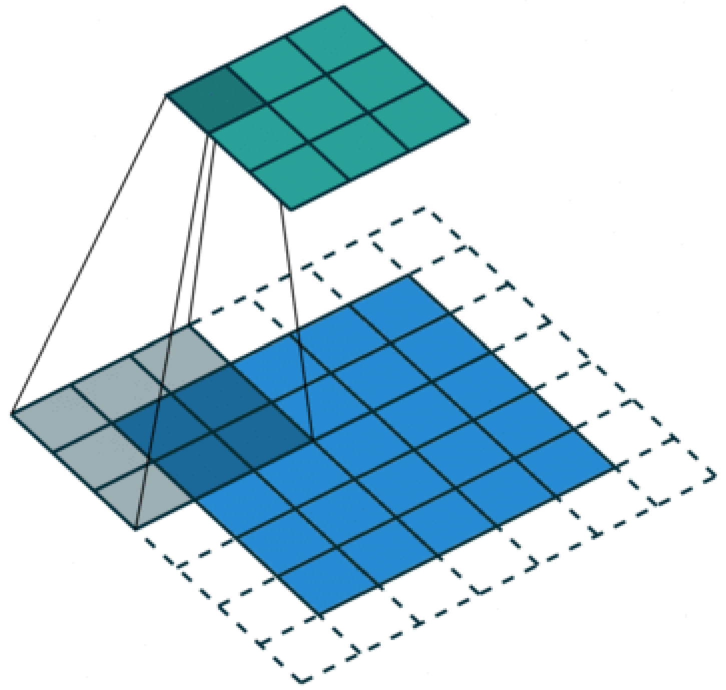
Output

-25				...
				...
				...
				...
...	...	...	...	...

Convolution operation on a MxNx3 image matrix with a 3x3x3 Kernel

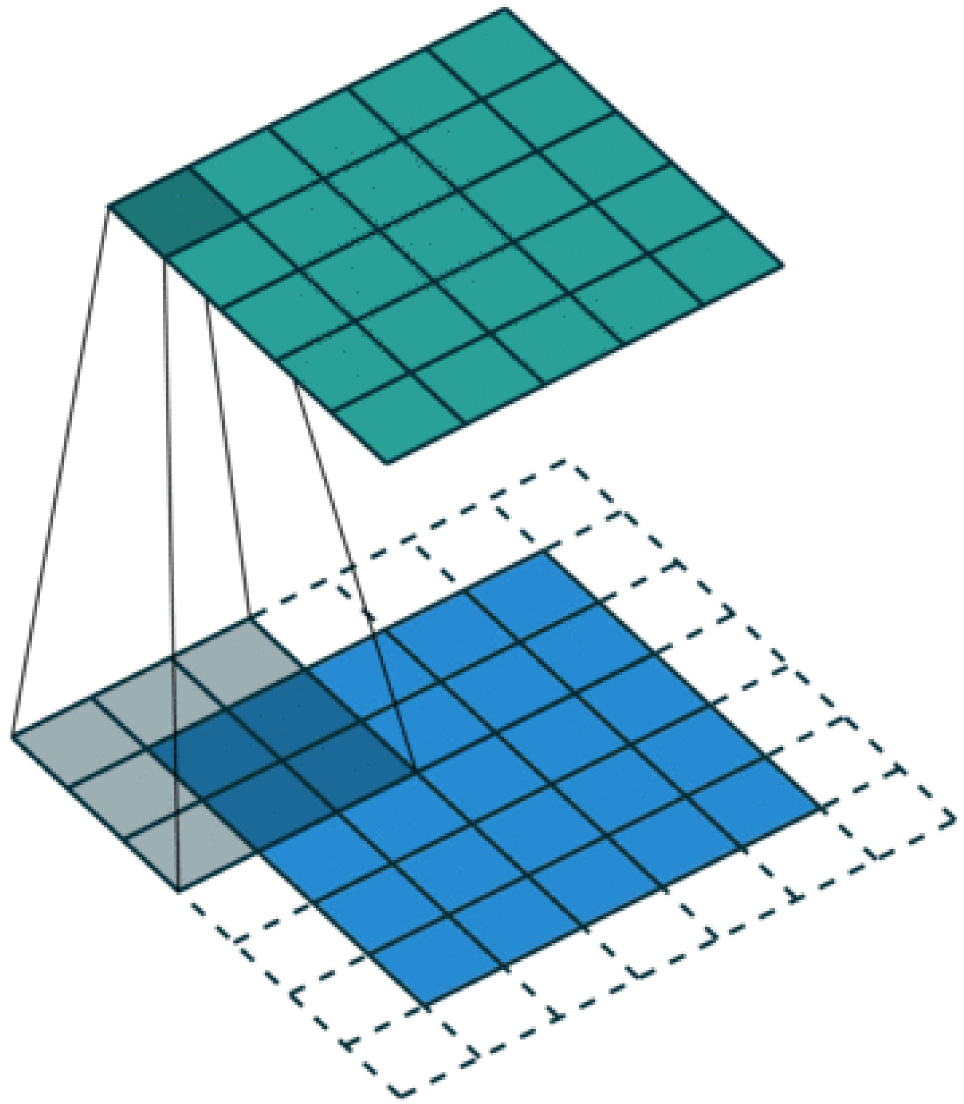
Convolution Operation with Stride Length = 2

the Kernel has the same depth as that of the input image. Matrix Multiplication is performed between Kn and In stack ([K1, I1]; [K2, I2]; [K3, I3]) and all the results are summed with the bias to give us a squashed one-depth channel Convoluted Feature Output.



Padding

- There are two types of results to the operation
  - one in which the convolved feature is reduced in dimensionality as compared to the input.
  - the other in which the dimensionality is either increased or remains the same.
  - This is done by applying **Valid Padding** in the case of the former, or **Same Padding** in the case of the latter.



Objective

- The objective of the Convolution Operation is to **extract the high-level features** such as edges, from the input image.
- ConvNets need not be limited to only one Convolutional Layer.
- Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc.
- With added layers, the architecture adapts to the High-Level features as well, giving us a network that has a wholesome understanding of images in the dataset, similar to how we would.

Input layer

- Example input a 28 pixel by 28 pixel grayscale image
  - Input is presented to network in 2D as 28 x 28 matrix
  - This makes capturing spatial relationships easier

Convolution layer

- Composed of multiple filters (kernels)
- Filters for 2D image are also 2D
- Suppose we have a 3 by 3 filter (9 values in total)
  - Values are randomly set to 0 or 1
- Convolution: placing 3 by 3 filter on the top left corner of image
  - Multiply filter values by pixel values, add the results
  - Move filter to right one pixel at a time, and repeat this process
  - When at top right corner, move filter down one pixel and repeat process
  - Process ends when we get to bottom right corner of image

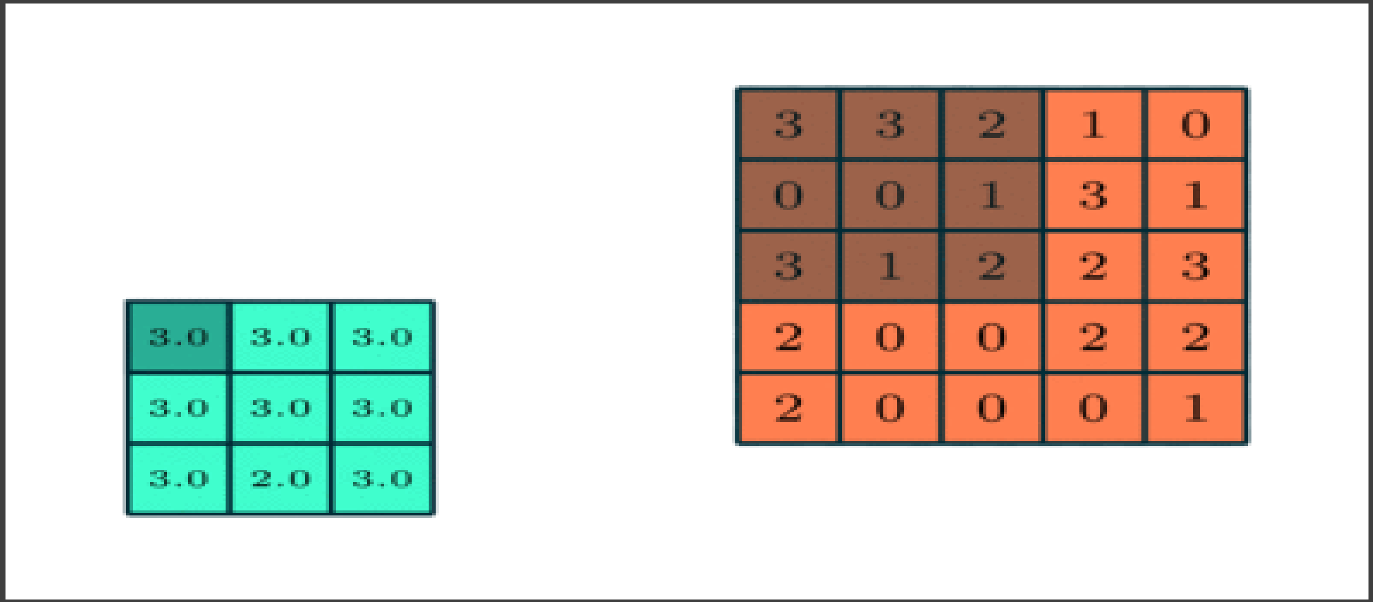
Pooling layer

- Pooling layer performs down sampling to reduce spatial dimensionality of input
- This decreases number of parameters
  - Reduces learning time/computation
  - Reduces likelihood of overfitting
- Most popular type is *max* pooling
  - Usually a 2 x 2 filter with a stride of 2
  - Returns maximum value as it slides over input data

Fully connected layer

- Last layer in a CNN
- Connect all nodes from previous layer to this fully connected layer
  - Which is responsible for classification of the image

Pooling Layer



3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

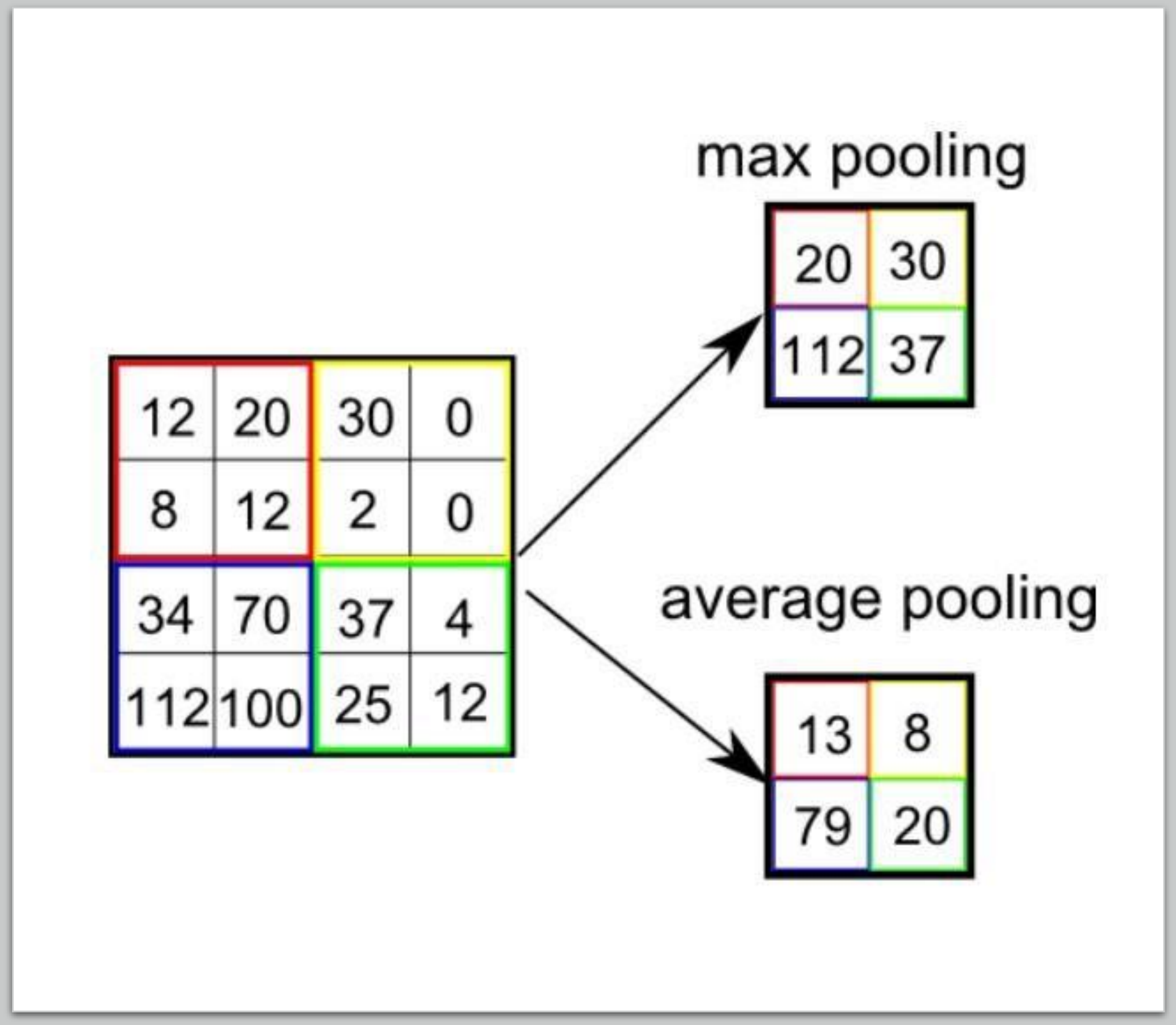
The Pooling layer is responsible for reducing the spatial size of the Convolved Feature.

This is to **decrease the computational power required to process the data** through dimensionality reduction.

it is useful for **extracting dominant features** which are rotational and positional invariant, thus maintaining the process of effectively training the model.

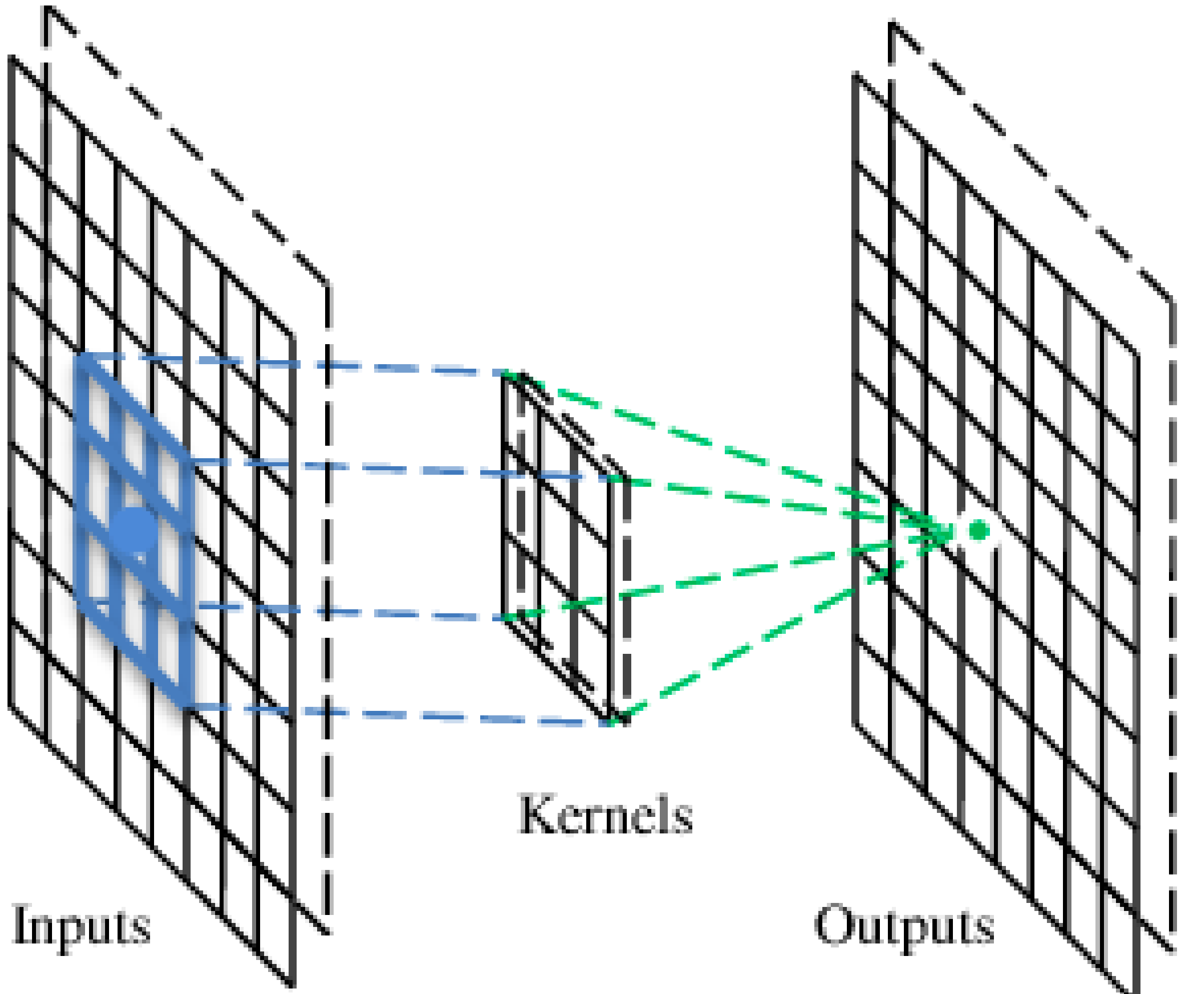
Types of Pooling

- There are two types of Pooling: Max Pooling and Average Pooling.
- **Max Pooling** returns the **maximum value** from the portion of the image covered by the Kernel.
- On the other hand, **Average Pooling** returns the **average of all the values** from the portion of the image covered by the Kernel.



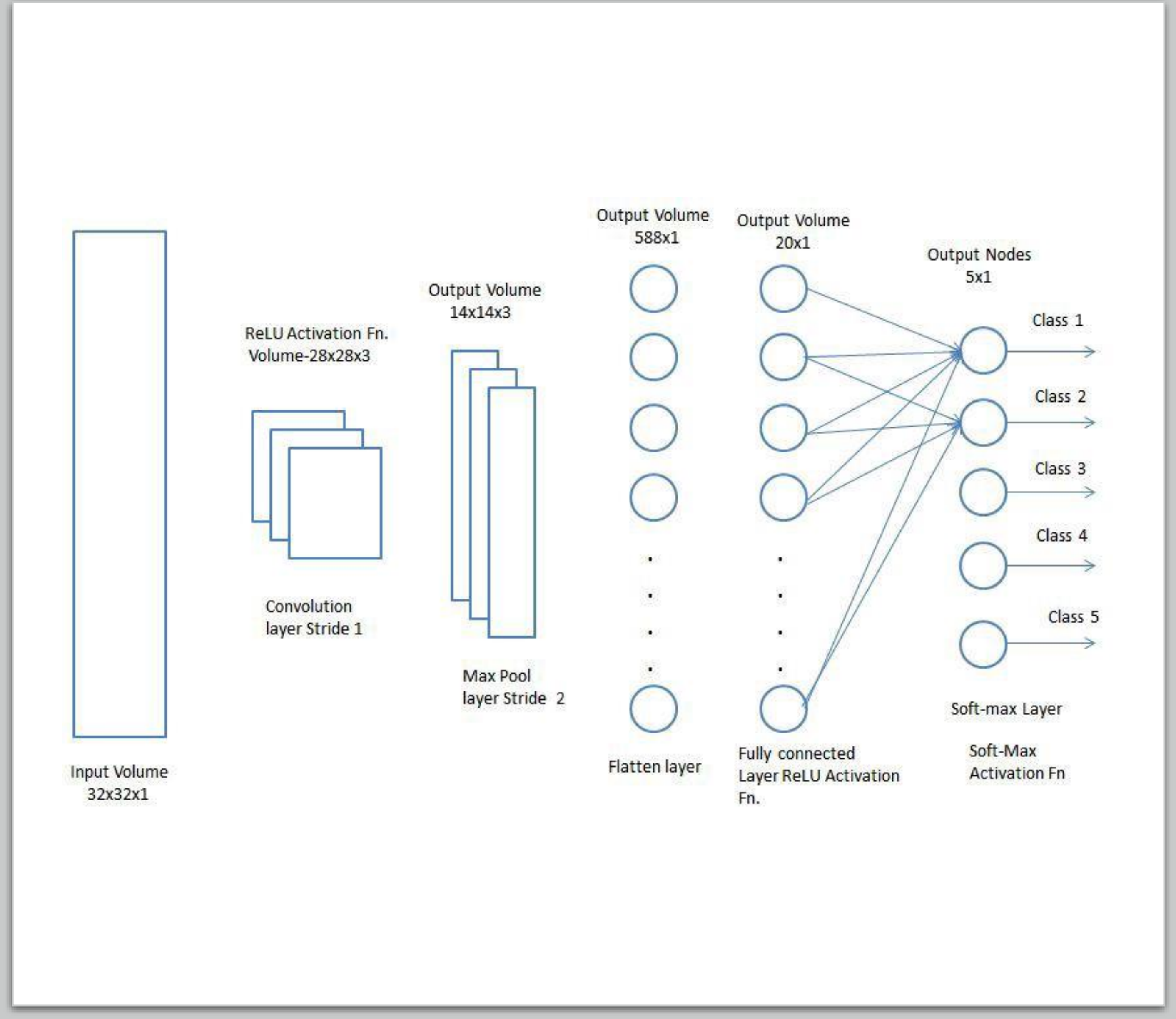
A convolutional layer

- A CNN is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that does convolutional operation.

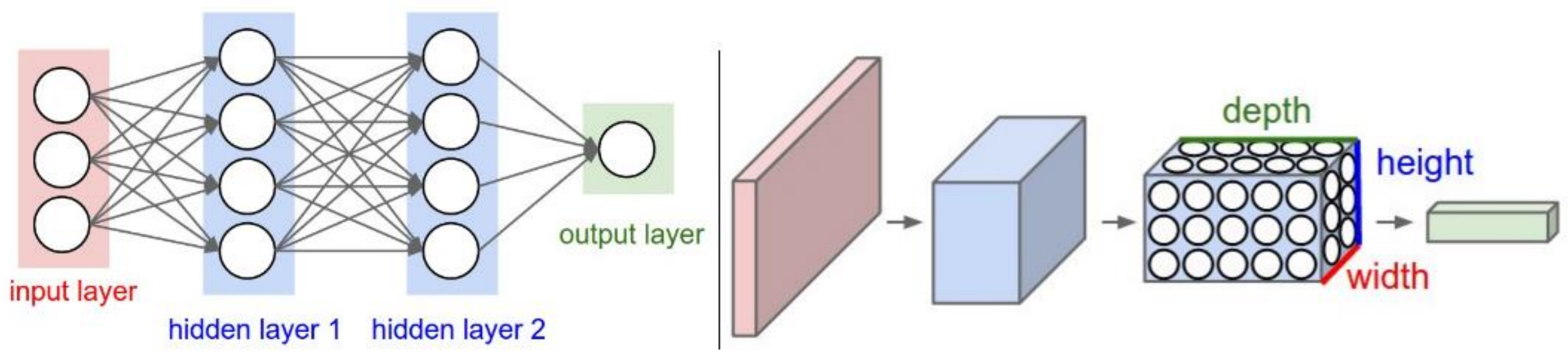


Classification — Fully Connected Layer (FC Layer)

- input image converted a suitable form for our Multi-Level Perceptron
- we flatten the image into a column vector.
- The flattened output is fed to a feed-forward neural network and backpropagation is applied to every iteration of training.
- Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the **Softmax Classification** technique.



Dense neural network and Convolutional neural network

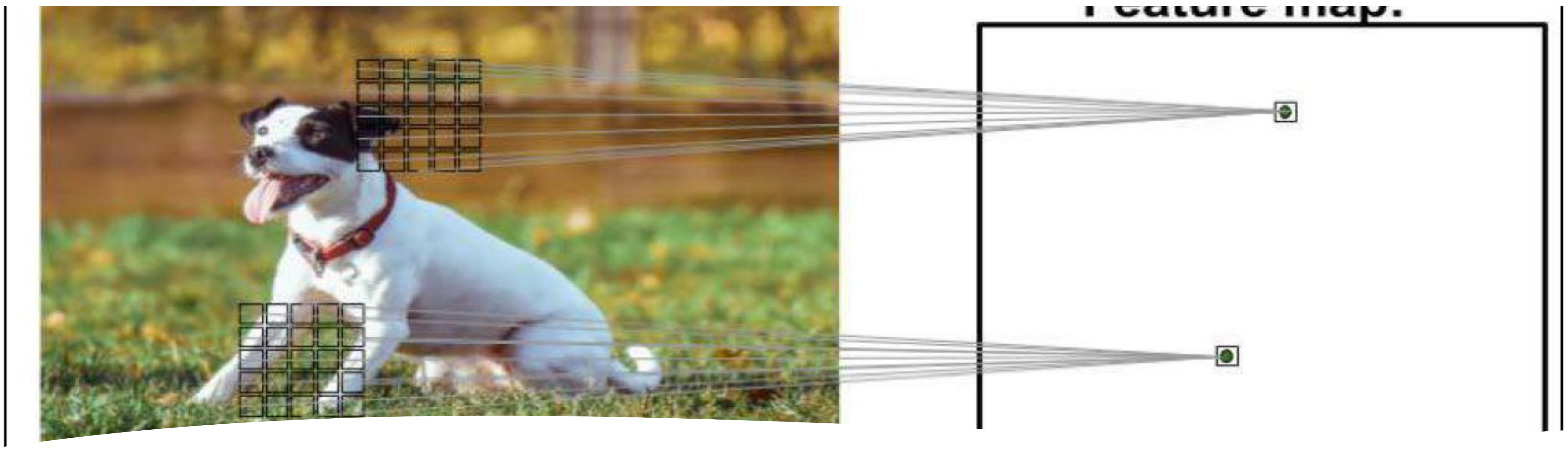


Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

building blocks of CNNs

CNNs are a family of models that were originally inspired by how the visual cortex of the human brain works when recognizing objects. The development of CNNs goes back to the 1990s, when Yann LeCun and his colleagues proposed a novel NN architecture for classifying handwritten digits from images (*Handwritten Digit Recognition with a Back-Propagation Network*, *Y*. *LeCun*, and others, *1989*,published at the *Neural Information Processing Systems (NeurIPS)* conference).

Source: python-machine-learning-and-deep-learning-with-python-scikit-learn-and-tensorflow-2



Understanding CNNs and feature hierarchies

- it's common to consider CNN layers as feature extractors
- the early layers (those right after the input layer) extract **low-level features** from raw data, and the later layers (often **fully connected layers** like in a multilayer perceptron (MLP)) use these features to predict a continuous target value or class label.
- As you can see in the following image, a CNN computes **feature maps** from an input image, where each element comes from a local patch of pixels in the input image:

- Due to the outstanding performance of CNNs for image classification tasks, this particular type of feedforward NN gained a lot of attention and led to tremendous improvements in machine learning for computer vision.
- This local patch of pixels is referred to as the **local receptive field**.
- CNNs will usually perform very well on image-related tasks, and that's largely due to two important ideas:
- **Sparse connectivity**: A single element in the feature map is connected to only a small patch of pixels.
- **Parameter-sharing**: The same weights are used for different patches of the input image.
- CNNs are composed of several **convolutional** and subsampling layers that are followed by one or more fully connected layers at the end. The fully connected layers are essentially an MLP, where every input unit, i, is connected to every output unit, j, with weight w
- subsampling layers, commonly known as **pooling layers**, do not have any learnable parameters

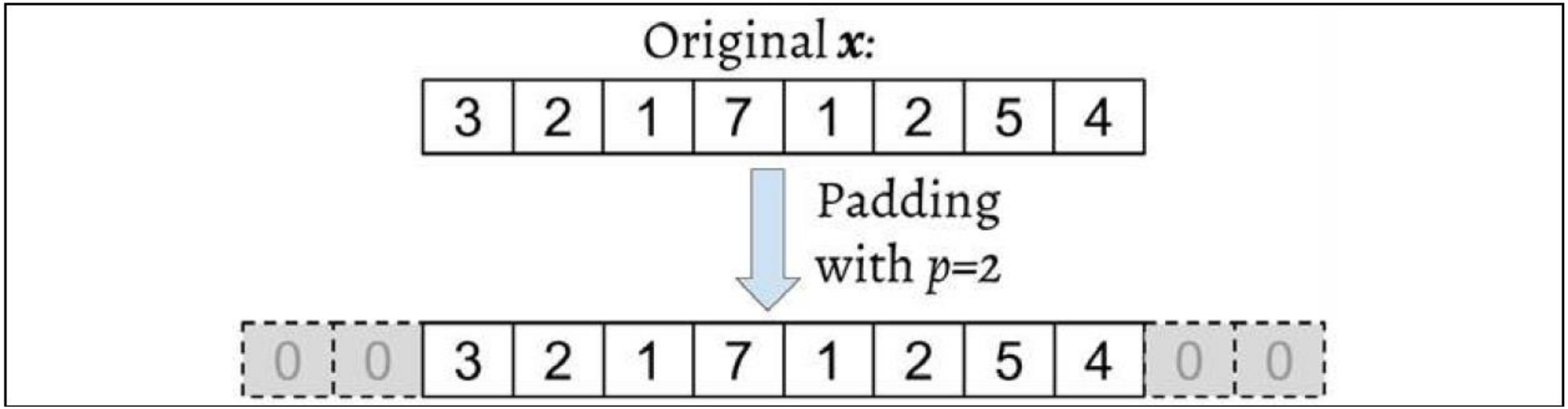
Performing discrete convolutions

- is a fundamental operation in a CNN.
- discuss some of the **naive** algorithms to compute convolutions of one-dimensional tensors (vectors) and two-dimensional tensors (matrices).
- A discrete convolution for two vectors, $\boldsymbol{x}$ and $\boldsymbol{w}$, is denoted by $\boldsymbol{yy} = \boldsymbol{xx} * \boldsymbol{ww}$, in which vector $\boldsymbol{x}$ is our input (sometimes called **signal**) and $\boldsymbol{w}$ is called the **filter** or **kernel**.
- A discrete convolution is mathematically defined as follows:
- Therefore, to correctly compute the summation, it is assumed that $\boldsymbol{x}$ and $\boldsymbol{w}$ are filled with zeros.
- Since this is not useful in practical situations, $\boldsymbol{x}$ is padded only with a finite number of zeros.

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]\, w[k]$$

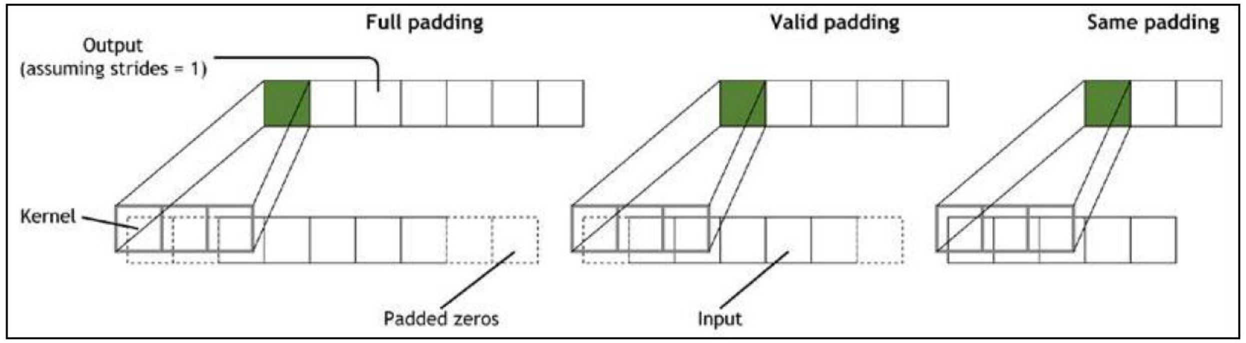
zero-padding

- the number of zeros padded on each side is denoted by *p*.



Modes of padding

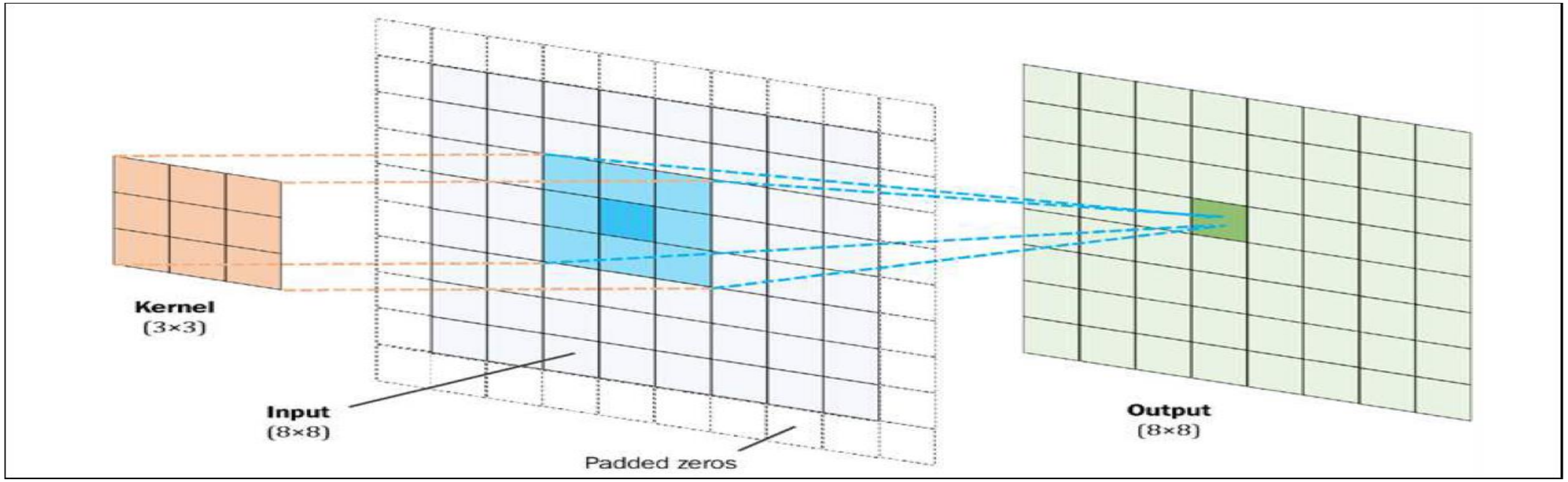
- In full mode, the padding parameter, p, is set to $p = m - 1$. Full padding increases the dimensions of the output; thus, it is rarely used in CNN architectures.
- Same padding is usually used to ensure that the output vector has the same size as the input vector, $\boldsymbol{x}$. In this case, the padding parameter, p, is computed according to the filter size, along with the requirement that the input size and output size are the same.
- Finally, computing a convolution in the valid mode refers to the case where $p = 0$ (no padding)
- SAME: Expects padding to be such that the input and output is of same size (provided stride=1) (pad value = kernel size)
- VALID: Padding value is set to 0



Performing a discrete convolution in 2D

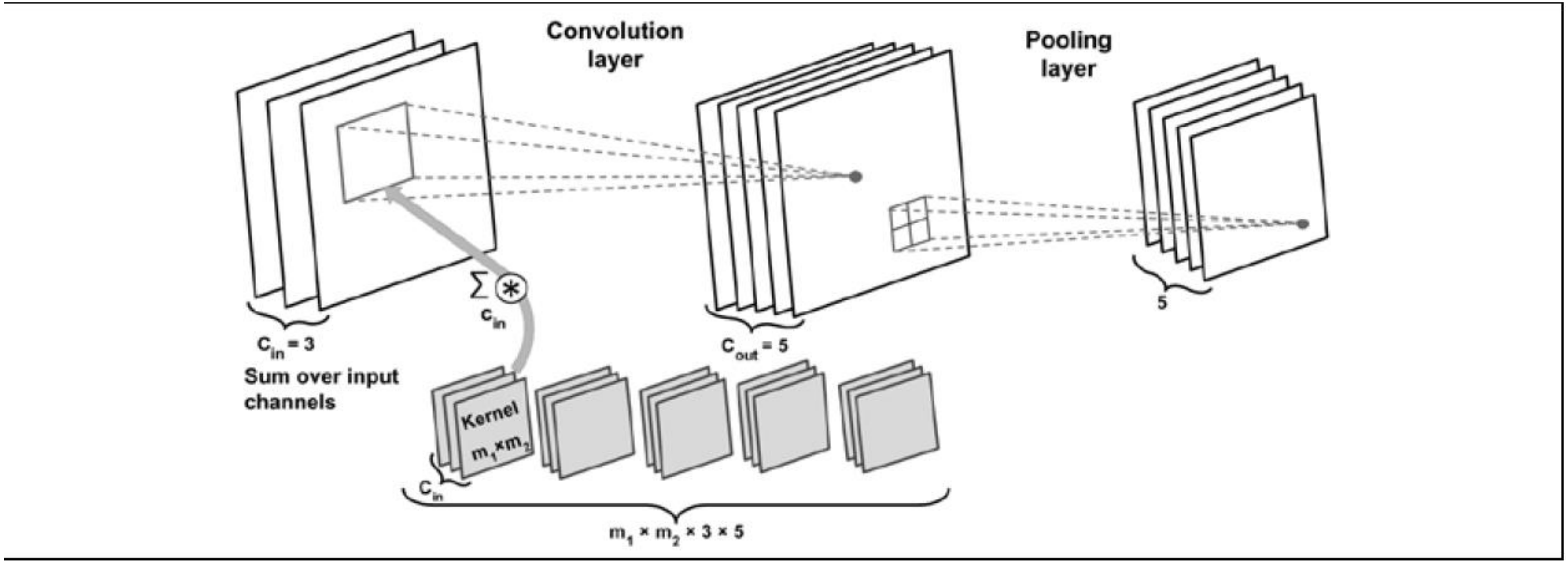
- This is defined mathematically as follows:

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W} \;\rightarrow\; Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2]\, W[k_1,k_2]$$



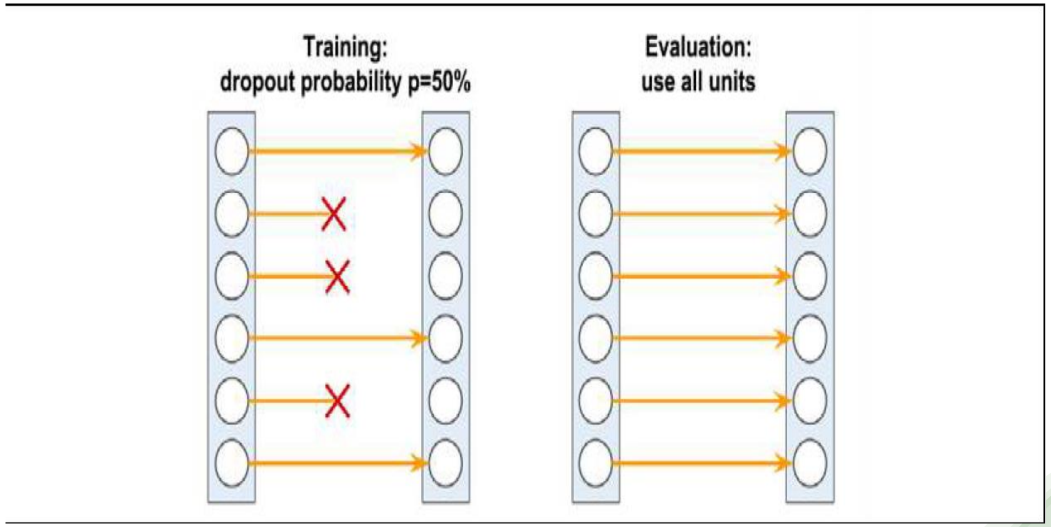
Working with multiple input or color channels

- An input to a convolutional layer may contain one or more 2D arrays or matrices with dimensions $N1 \times N2$
- These $N1 \times N2$ matrices are called *channels*.
- three-dimensional array $\boldsymbol{X}N1{\times}N2{\times}Ci$ where Ci is the number of input channels.
- If the image is colored and uses the RGB color mode, then Ci = 3
- if the image is in grayscale, then we have Ci = 1


Convolution layer
Pooling layer
$\sum_{c_{in}} \circledast$
$C_{in} = 3$
Sum over input channels
$C_{out} = 5$
5
Kernel $m_1 \times m_2$
C_{in}
$m_1 \times m_2 \times 3 \times 5$

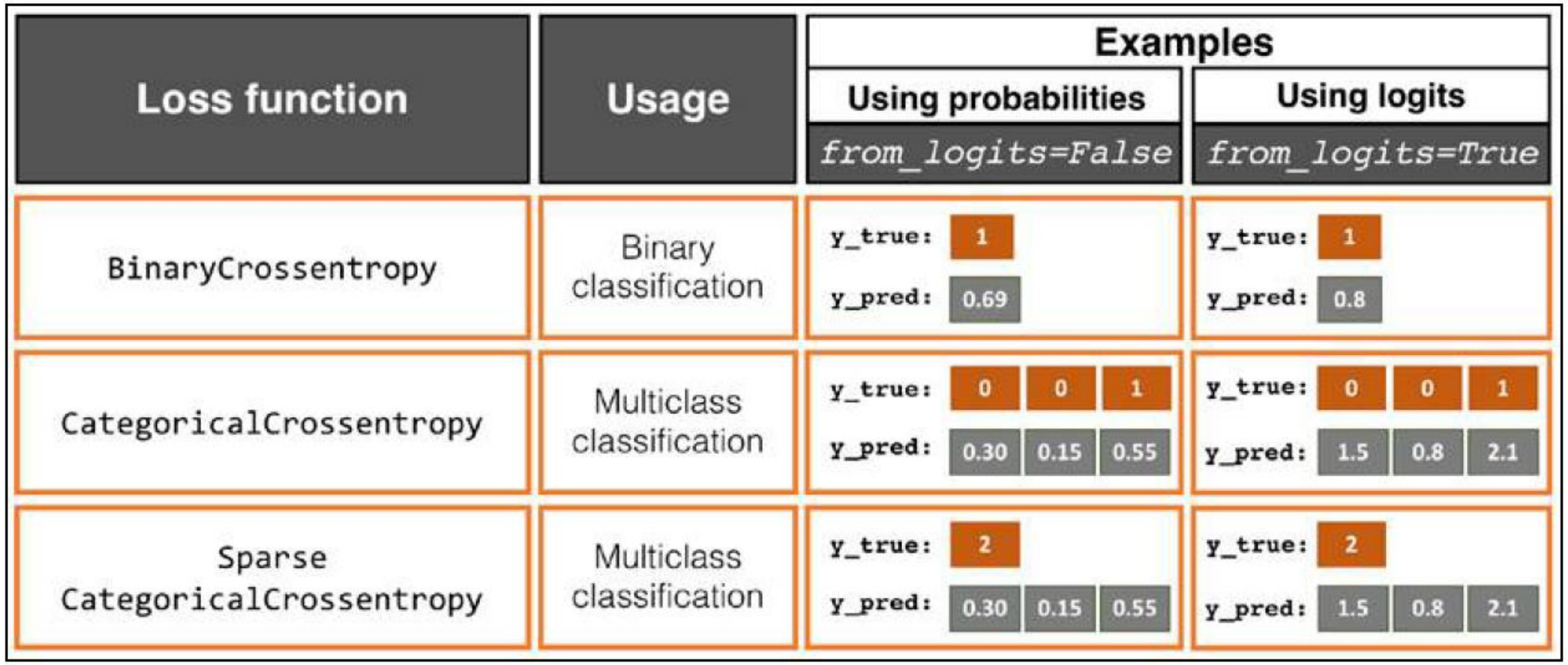
Regularizing an NN with dropout

- **dropout** has emerged as a popular technique for regularizing (deep) NNs to avoid overfitting
- Dropout is usually applied to the hidden units of higher layers and works as follow:
- during the training phase of an NN, a fraction of the hidden units is randomly dropped at every iteration with probability $p$drop

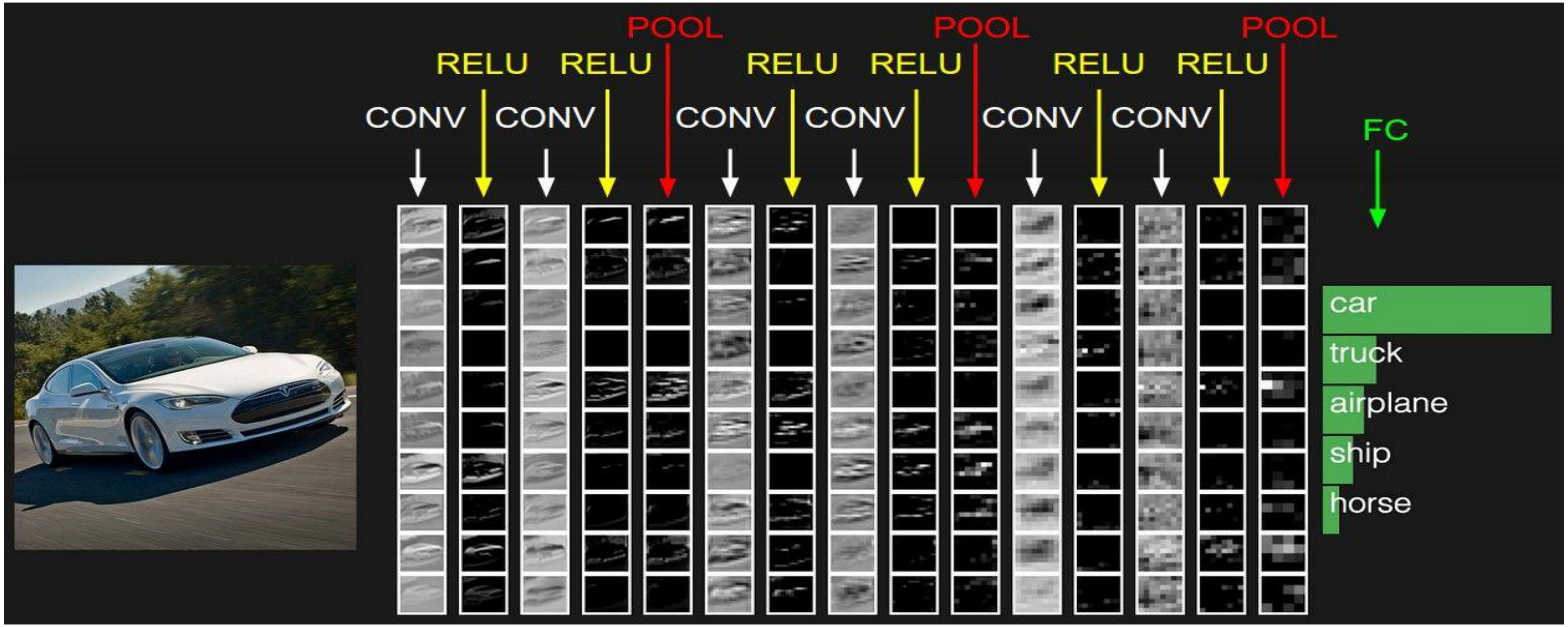


Loss functions for classification

binary classification, multiclass with one-hot encoded ground truth labels, and multiclass with integer (sparse) labels.



Loss function	Usage	Examples	
		Using probabilities	Using logits
		from_logits=False	*from_logits=True*
`BinaryCrossentropy`	Binary classification	**y_true:** 1 **y_pred:** 0.69	**y_true:** 1 **y_pred:** 0.8
`CategoricalCrossentropy`	Multiclass classification	**y_true:** 0 0 1 **y_pred:** 0.30 0.15 0.55	**y_true:** 0 0 1 **y_pred:** 1.5 0.8 2.1
`Sparse CategoricalCrossentropy`	Multiclass classification	**y_true:** 2 **y_pred:** 0.30 0.15 0.55	**y_true:** 2 **y_pred:** 1.5 0.8 2.1

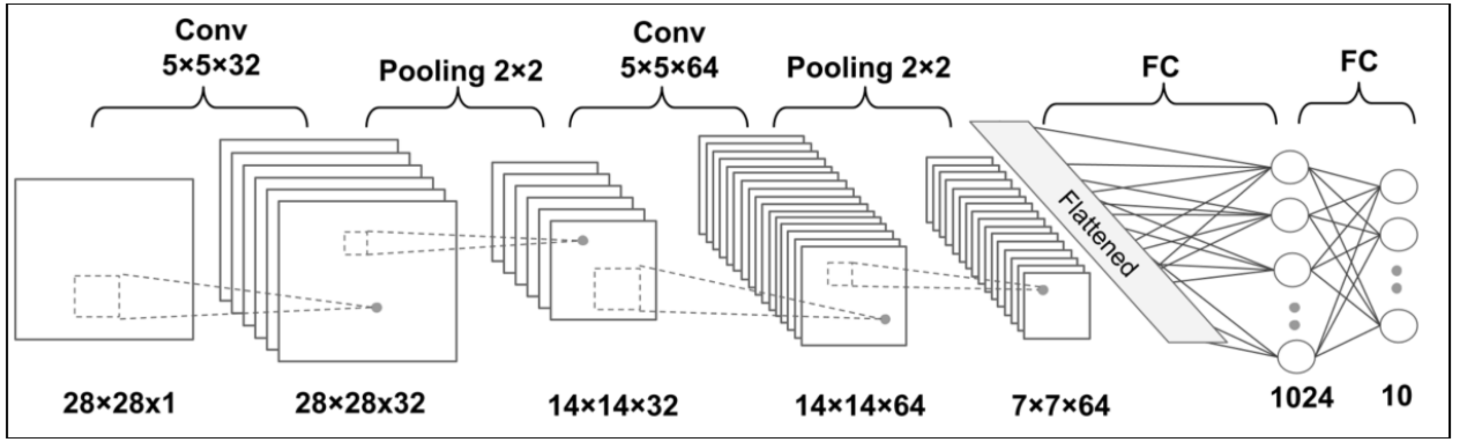


CONV: Convolutional kernel layer
RELU: Activation function
POOL: Dimension reduction layer
FC: Fully connection layer

Source: https://cs.stanford.edu/

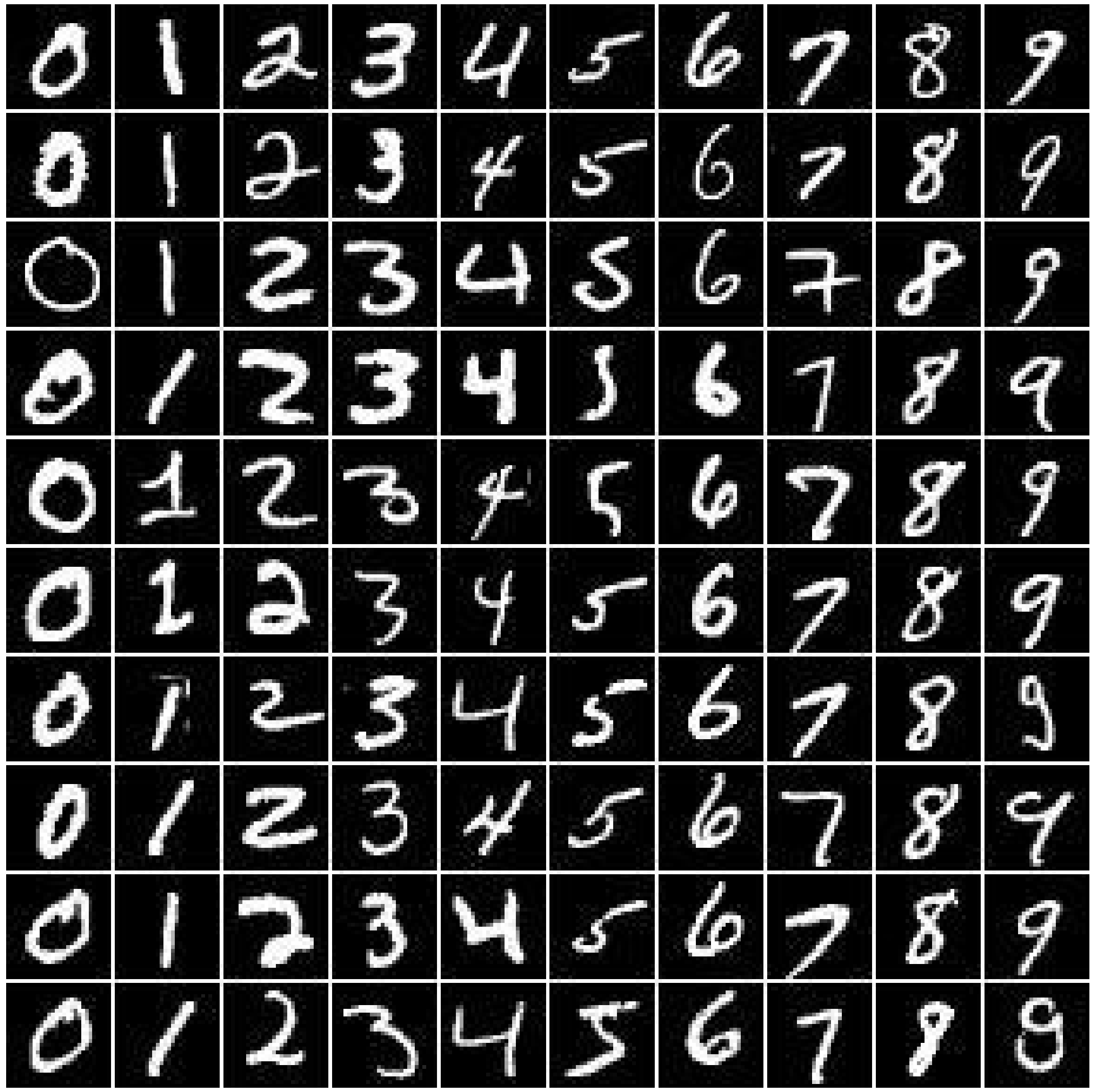
Implementing a deep CNN using TensorFlow

- The inputs are 28 × 28 grayscale images
- The input data goes through two convolutional layers that have a kernel size of 5 × 5 .
- The first convolution has 32 output feature maps, and the second one has 64 output feature maps.
- Each convolution layer is followed by a subsampling layer in the form of a max-pooling operation, PP2×2.
- Then a fully connected layer passes the output to a second fully connected layer, which acts as the final *softmax* output layer.



MNIST dataset

- The MNIST database of handwritten digits,
- available from this page,
- has a training set of 60,000 examples, and a test set of 10,000 examples.
- It is a subset of a larger set available from NIST.
- The digits have been size-normalized and centered in a fixed-size image.



The dimensions of the tensors in each layer are as follows:

- Input: [batchsize × 28 × 28 × 1]
- Conv_1: [batchsize × 28 × 28 × 32]
- Pooling_1: [batchsize × 14 × 14 × 32]
- Conv_2: [batchsize × 14 × 14 × 64]
- Pooling_2: [batchsize × 7 × 7 × 64]
- FC_1: [batchsize × 1024]
- FC_2 and softmax layer: [batchsize × 10]

Loading and preprocessing the data

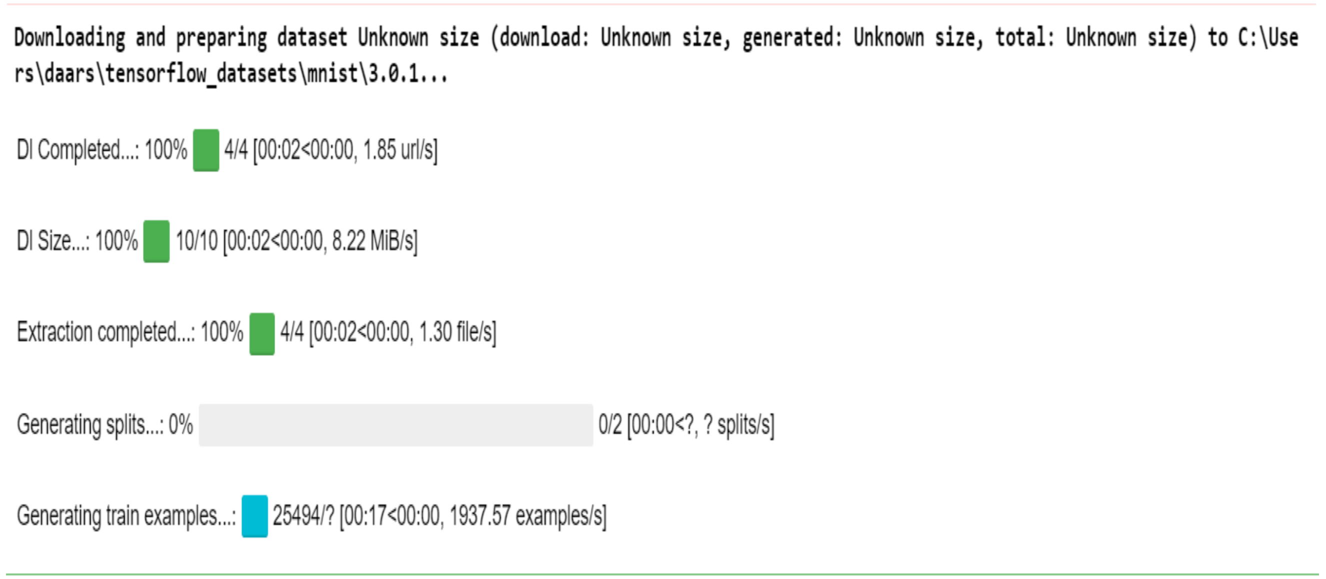
- pip install tensorflow_datasets
- import tensorflow_datasets as tfds
- import pandas as pd

- import matplotlib.pyplot as plt
- %matplotlib inline

Loading the dataset

```
## MNIST dataset

mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
datasets = mnist_bldr.as_dataset(shuffle_files=False)
print(datasets.keys())
mnist_train_orig, mnist_test_orig = datasets['train'], datasets['test']
```

The MNIST dataset comes with a pre-specified training and test dataset partitioning scheme



```
Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unknown size, total: Unknown size) to C:\Use
rs\daars\tensorflow_datasets\mnist\3.0.1...

Dl Completed...: 100%  4/4 [00:02<00:00, 1.85 url/s]

Dl Size...: 100%  10/10 [00:02<00:00, 8.22 MiB/s]

Extraction completed...: 100%  4/4 [00:02<00:00, 1.30 file/s]

Generating splits...: 0%  0/2 [00:00<?, ? splits/s]

Generating train examples...:  25494/? [00:17<00:00, 1937.57 examples/s]
```

Define parameters

- An example of this behavior where the number of labels in the validation datasets changes due to reshuffling of the train/validation splits.
- This can cause *false* performance estimation of the model, since the train/validation datasets are indeed mixed.)
- We can split the train/validation datasets as follows:

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64
NUM_EPOCHS = 20
```

Train the model

- mnist_train = mnist_train_orig.map(
-     lambda item: (tf.cast(item['image'], tf.float32)/255.0,
-                   tf.cast(item['label'], tf.int32)))

- mnist_test = mnist_test_orig.map(
-     lambda item: (tf.cast(item['image'], tf.float32)/255.0,
-                   tf.cast(item['label'], tf.int32)))

- tf.random.set_seed(1)

- mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
-                                   reshuffle_each_iteration=False)

- mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
- mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)

Implementing a CNN using the TensorFlow Keras API

- Configuring CNN layers in Keras
    - Conv2D: tf.keras.layers.Conv2D
        - filters
        - kernel_size
        - strides
        - padding
    - MaxPool2D: tf.keras.layers.MaxPool2D
        - pool_size
        - strides
        - padding
    - Dropout tf.keras.layers.Dropout2D
        - rate

Constructing a CNN in Keras

- model = tf.keras.Sequential()

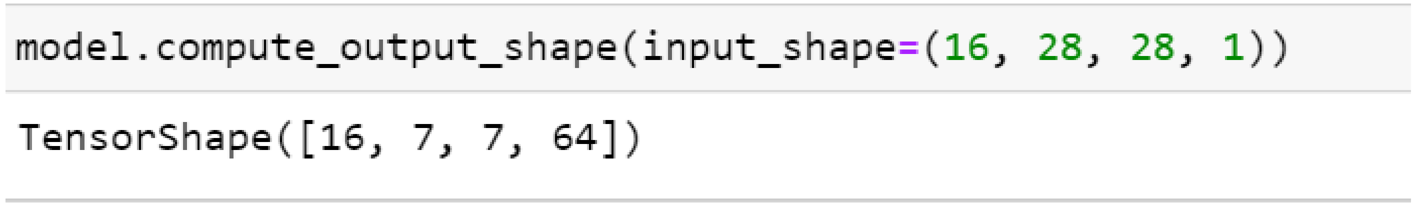
- model.add(tf.keras.layers.Conv2D(
-     filters=32, kernel_size=(5, 5),
-     strides=(1, 1), padding='same',
-     data_format='channels_last',
-     name='conv_1', activation='relu'))

- model.add(tf.keras.layers.MaxPool2D(
-     pool_size=(2, 2), name='pool_1'))
-
- model.add(tf.keras.layers.Conv2D(
-     filters=64, kernel_size=(5, 5),
-     strides=(1, 1), padding='same',
-     name='conv_2', activation='relu'))

- model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2), name='pool_2'))

Model shaping

```
model.compute_output_shape(input_shape=(16, 28, 28, 1))
```

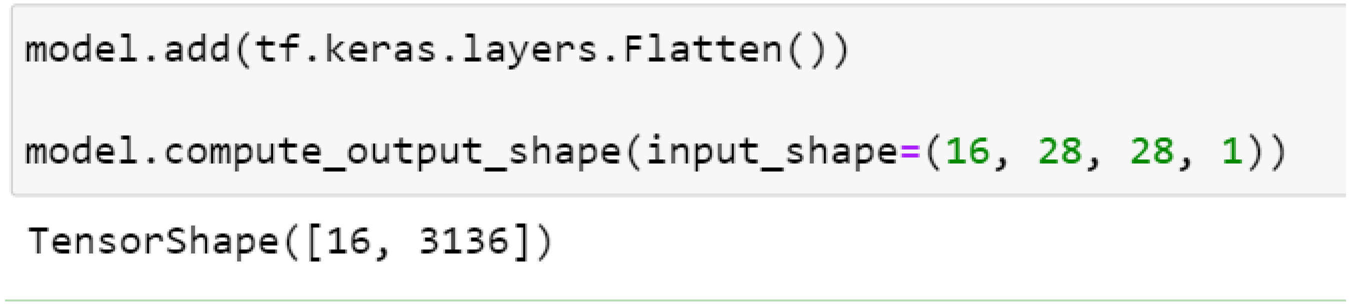


```
model.compute_output_shape(input_shape=(16, 28, 28, 1))
```

```
TensorShape([16, 7, 7, 64])
```

Model add

```
model.add(tf.keras.layers.Flatten())
model.compute_output_shape(input_shape=(16, 28, 28, 1))
```



```
model.add(tf.keras.layers.Flatten())

model.compute_output_shape(input_shape=(16, 28, 28, 1))
```

```
TensorShape([16, 3136])
```

dropout

- model.add(tf.keras.layers.Dense(
-     units=1024, name='fc_1',
-     activation='relu'))

- model.add(tf.keras.layers.Dropout(
-     rate=0.5))
-
- model.add(tf.keras.layers.Dense(
-     units=10, name='fc_2',
-     activation='softmax'))

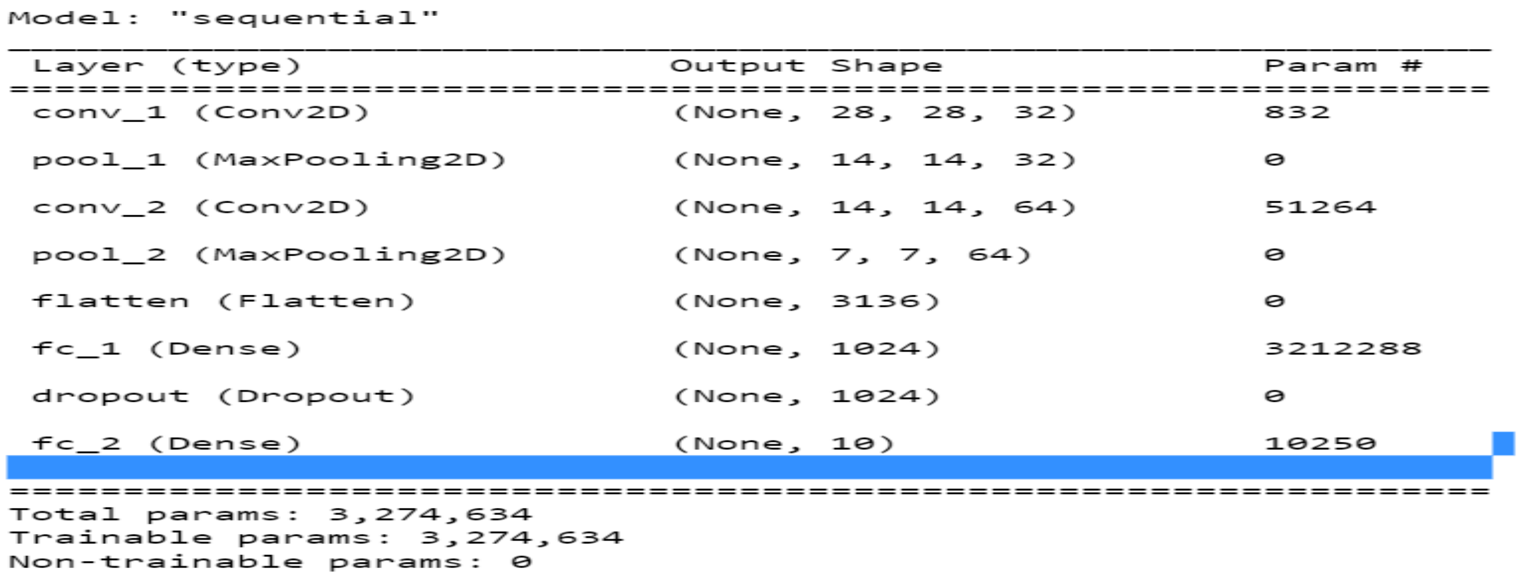
Build the model

- tf.random.set_seed(1)
- model.build(input_shape=(None, 28, 28, 1))

- model.compute_output_shape(input_shape=(16, 28, 28, 1))

Model summary

- model.summary()



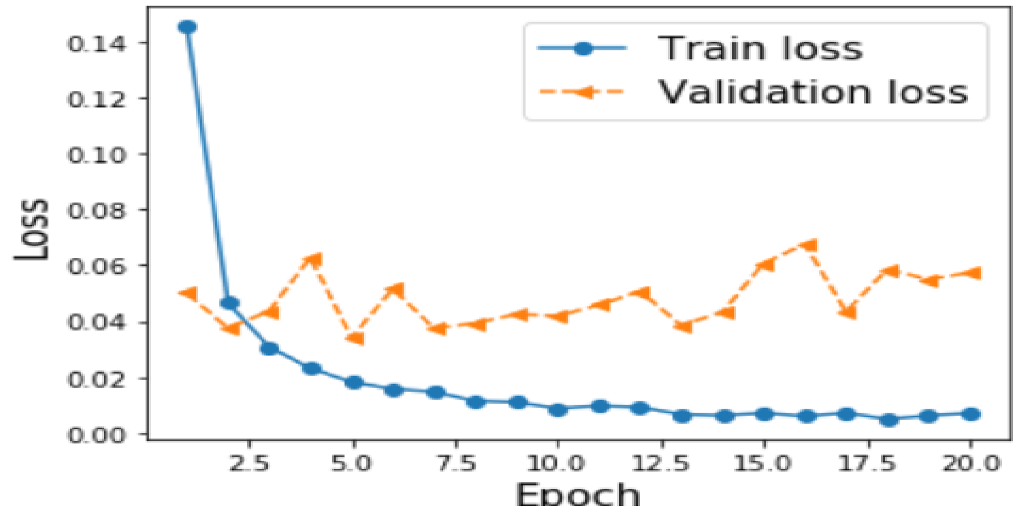
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (Conv2D)             (None, 28, 28, 32)        832

 pool_1 (MaxPooling2D)       (None, 14, 14, 32)        0

 conv_2 (Conv2D)             (None, 14, 14, 64)        51264

 pool_2 (MaxPooling2D)       (None, 7, 7, 64)          0

 flatten (Flatten)           (None, 3136)              0

 fc_1 (Dense)                (None, 1024)              3212288

 dropout (Dropout)           (None, 1024)              0

 fc_2 (Dense)                (None, 10)                10250

=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
```

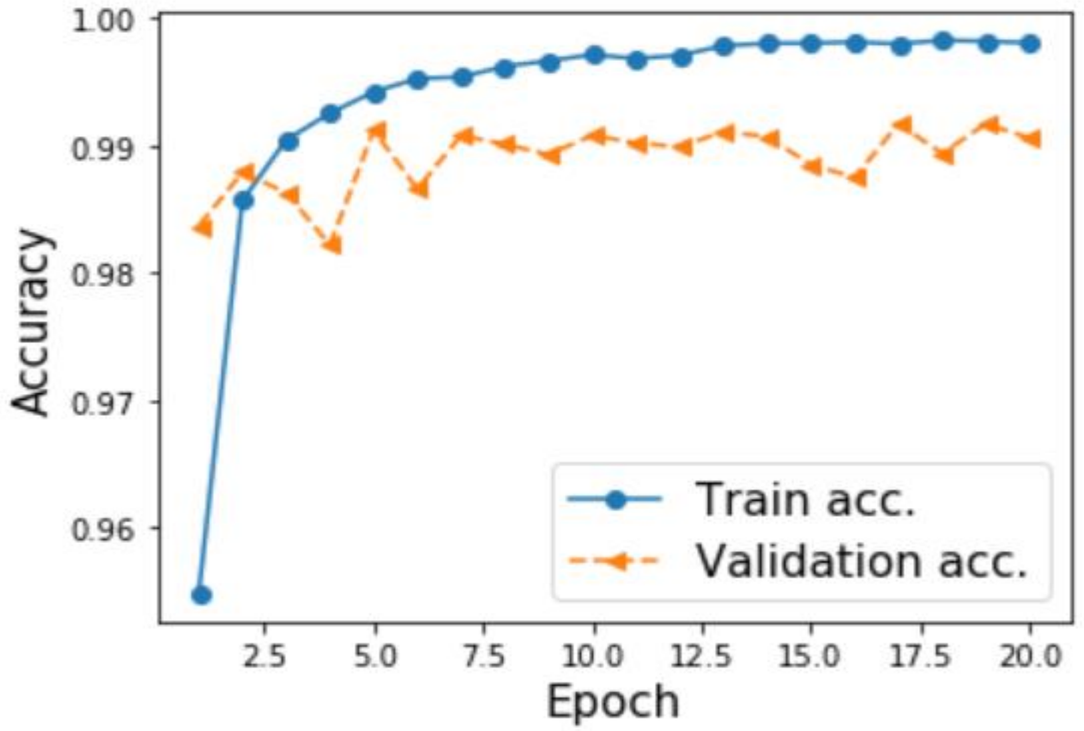
Compile the model

- model.compile(optimizer=tf.keras.optimizers.Adam(),
-               loss=tf.keras.losses.SparseCategoricalCrossentropy(),
-               metrics=['accuracy']) # same as `tf.keras.metrics.SparseCategoricalAccuracy(name='accuracy')`

- history = model.fit(mnist_train, epochs=NUM_EPOCHS,
-                     validation_data=mnist_valid,
-                     shuffle=True)

- hist = history.history
- x_arr = np.arange(len(hist['loss'])) + 1
- fig = plt.figure(figsize=(12, 4))
- ax = fig.add_subplot(1, 2, 1)
- ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
- ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
- ax.set_xlabel('Epoch', size=15)
- ax.set_ylabel('Loss', size=15)
- ax.legend(fontsize=15)
- ax = fig.add_subplot(1, 2, 2)
- ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
- ax.plot(x_arr, hist['val_accuracy'], '--<', label='Validation acc.')
- ax.legend(fontsize=15)
- ax.set_xlabel('Epoch', size=15)
- ax.set_ylabel('Accuracy', size=15)
- #plt.savefig('figures/15_12.png', dpi=300)
- plt.show()



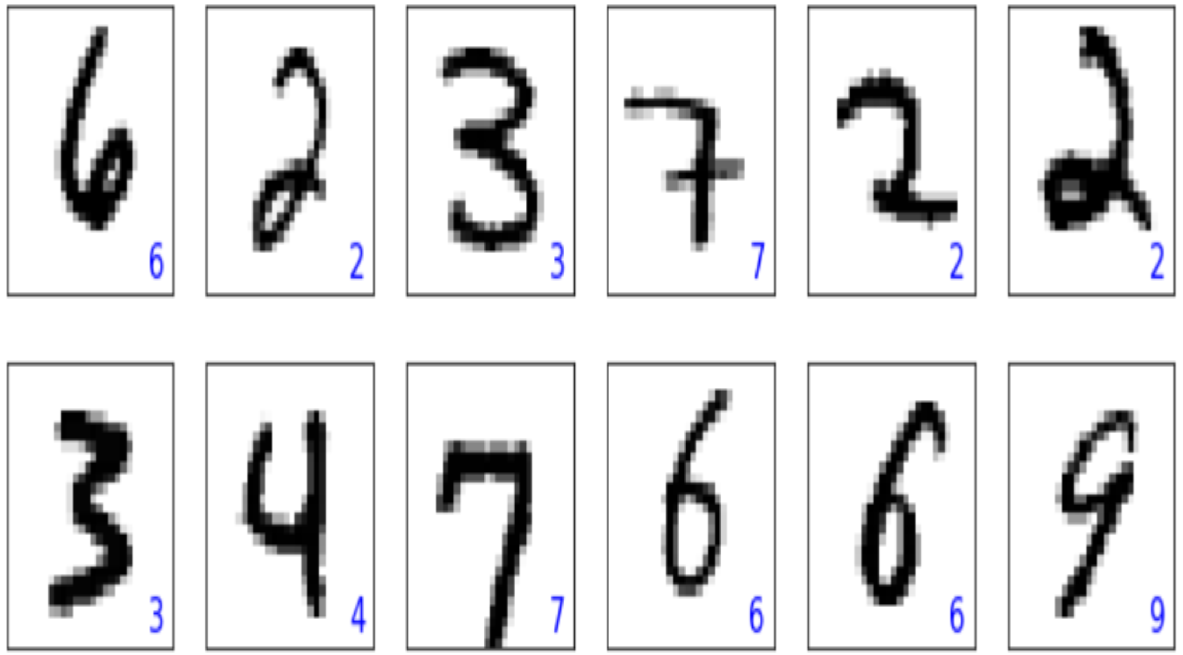


Testing

- test_results = model.evaluate(mnist_test.batch(20))
- print('\nTest Acc. {:.2f}%'.format(test_results[1]*100))

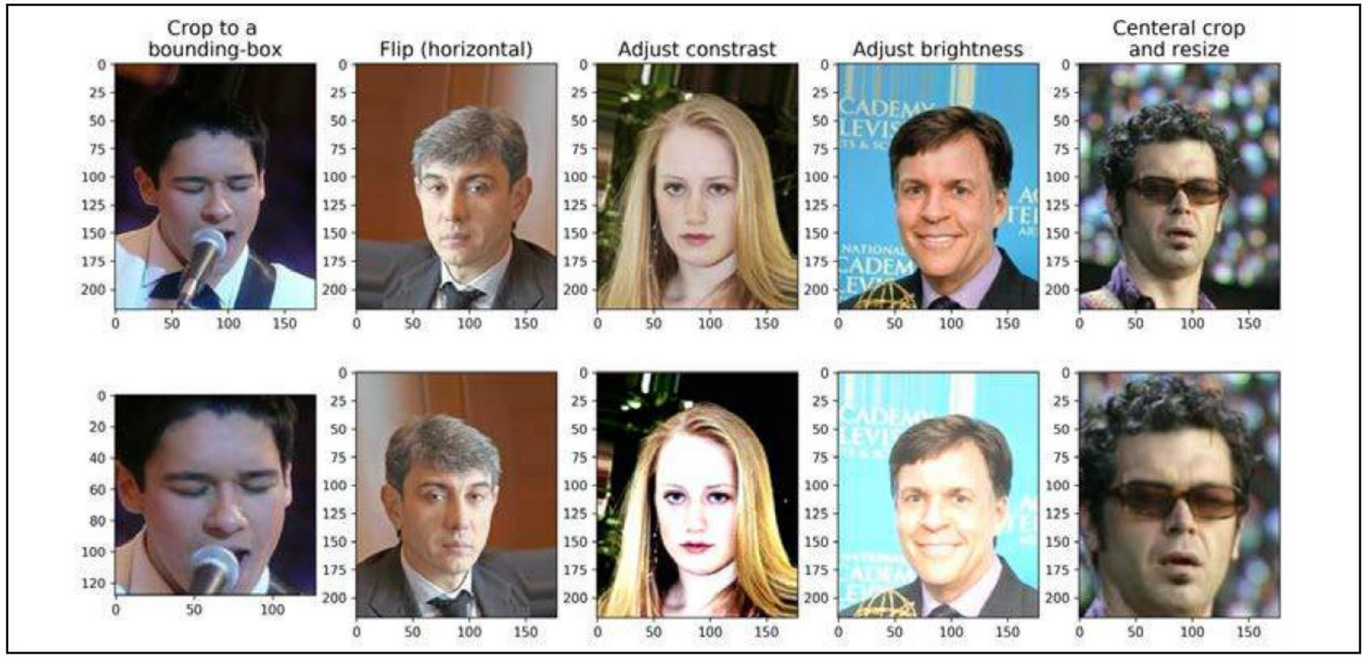
```
batch_test = next(iter(mnist_test.batch(12)))
preds = model(batch_test[0])
tf.print(preds.shape)
preds = tf.argmax(preds, axis=1)
print(preds)
fig = plt.figure(figsize=(12, 4))
for i in range(12):
   ax = fig.add_subplot(2, 6, i+1)
   ax.set_xticks([]); ax.set_yticks([])
   img = batch_test[0][i, :, :, 0]
   ax.imshow(img, cmap='gray_r')
   ax.text(0.9, 0.1, '{}'.format(preds[i]),
         size=15, color='blue',
         horizontalalignment='center',
         verticalalignment='center',
         transform=ax.transAxes)
   #plt.savefig('figures/15_13.png', dpi=300)
plt.show()
```

```
TensorShape([12, 10])
tf.Tensor([6 2 3 7 2 2 3 4 7 6 6 9], shape=(12,), dtype=int64)
```



Gender classification from face images using a CNN

- Loading the CelebA dataset



Thank You