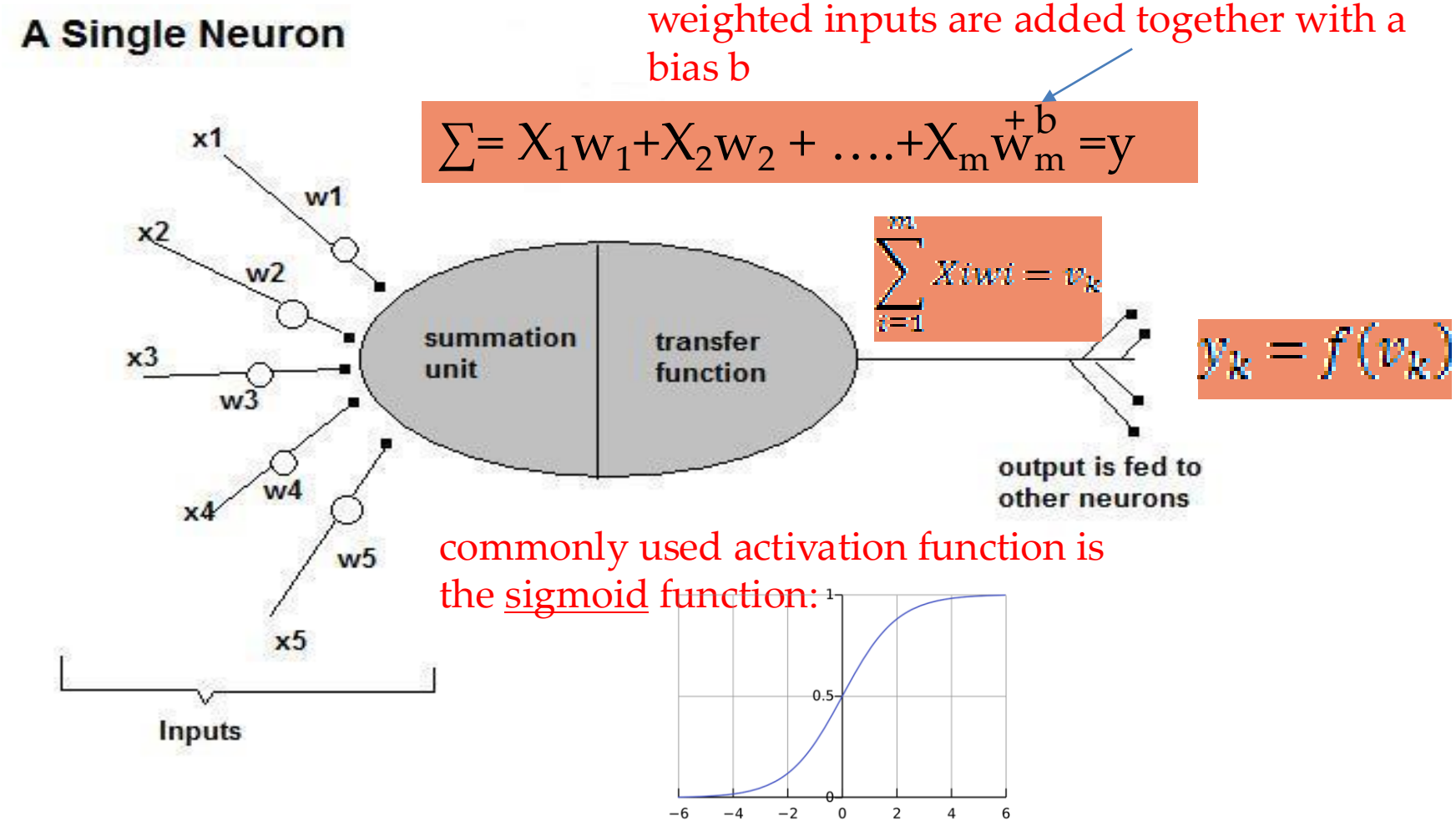
Embedded ML

Convolution Networks

Prof. Venki Muthukumar

How do ANNs work?

- model of an artificial neuron.



Process of ANN architecture design

1. Take inputs
2. Add bias (if required)
3. Assign random weights to input features
4. Run the code for training.
5. Find the error in prediction.
6. Update the weight by gradient descent algorithm.
7. Repeat the training phase with updated weights.
8. Make predictions.

Design Issues

- Initial weights (small random values ∈[-1,1])
- Transfer/Activation function (How the inputs and the weights are combined to produce output?)
- Error estimation
- Weights adjusting - Gradient Descent (Optimization)
- Number of neurons
- Data representation
- Size of training set

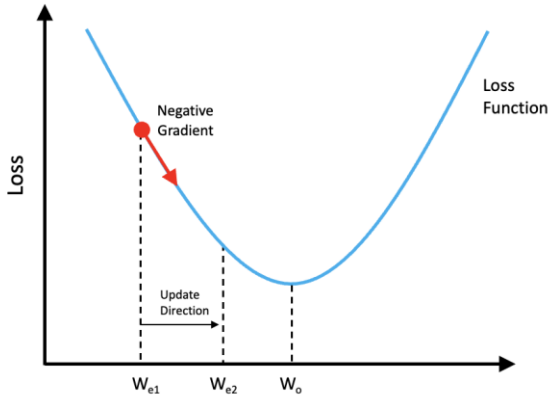
Weights Adjusting

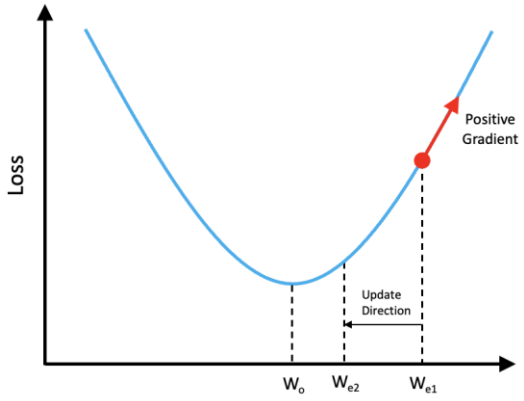
- After each iteration, weights should be adjusted to minimize the error.
  - All possible weights
  - Back propagation

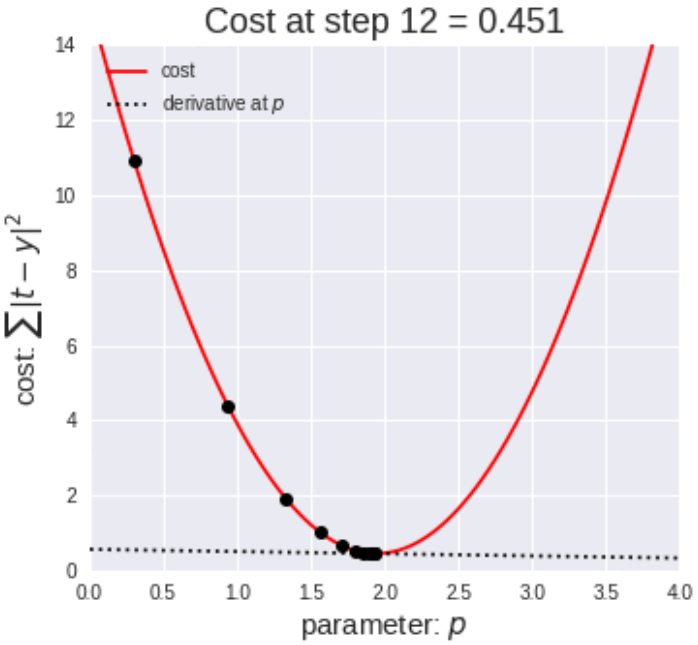
$$W_{e2} = W_{e1} - \text{Gradient} * \text{LearningRate}$$

Direction to move

Tuning Parameter that determines how big a step to take

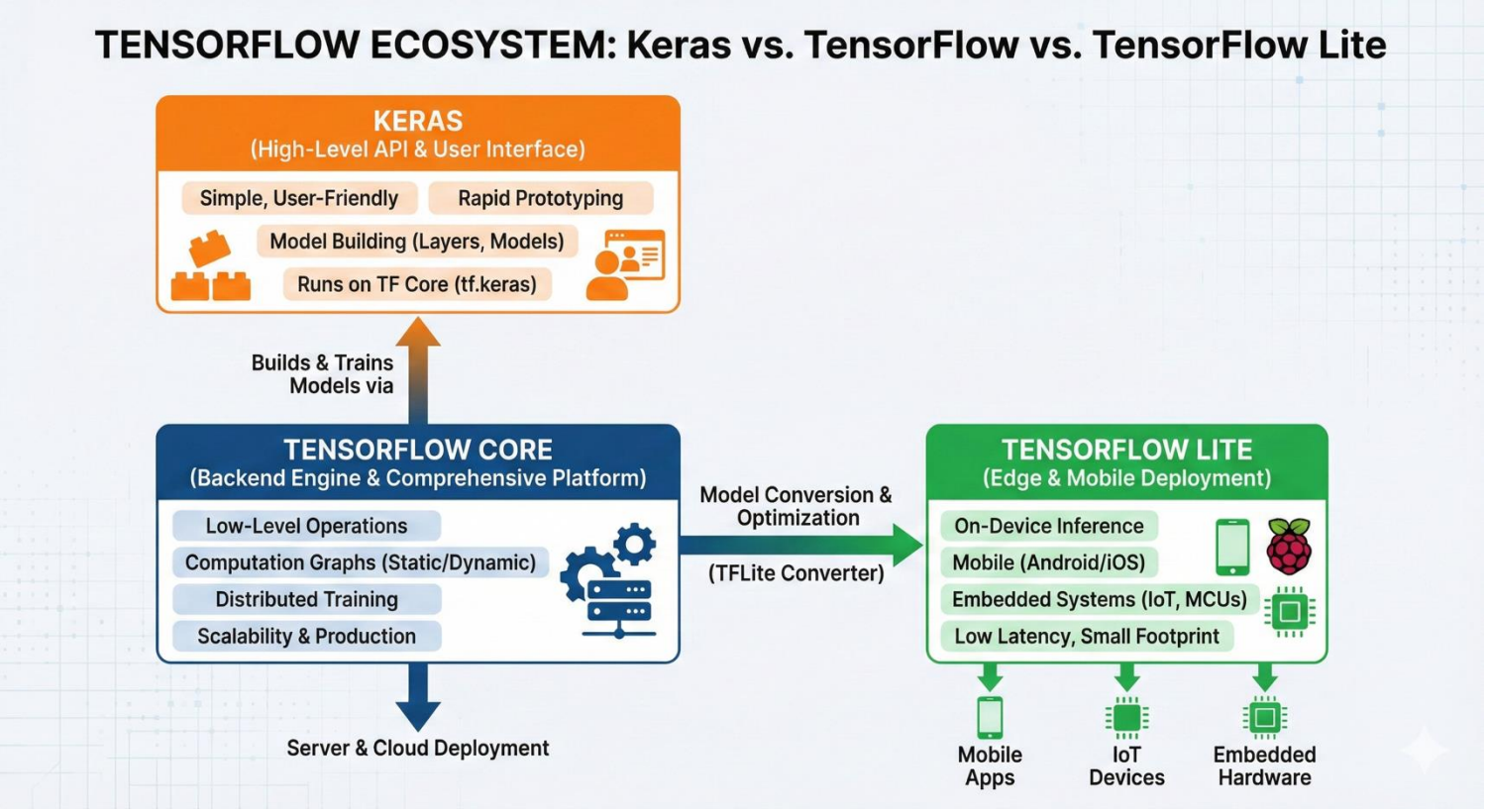






Tensorflow & Keras

- TensorFlow is the **engine that does the heavy math** behind machine learning.
- **Keras** is a **high-level API** built on top of TensorFlow.
- **TensorFlow Lite (TFLite)** is a **lightweight version of TensorFlow** designed for **resource-constrained devices**.



First Neural Network

- So, consider if I give you a set of numbers like this:
- X: -1, 0, 1, 2, 3, 4
- And then I give you another set of numbers like this:
- Y: -3, -1, 1, 3, 5, 7
- Can you figure out the relationship between the two sets? There's a function that converts -1 to -3, 0 to -1, 1 to 1, 2 to 3, 3 to 5 and 4 to 7. Can you figure out the relationship.

First Neural Network

Embedded ML

```
import numpy as np
from tensorflow import keras

model = keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
```

```
[[17.276058]]
```

```
Epoch 1/50
1/1 [==============================] - 0s 272ms/step - loss: 5.7897
Epoch 2/50
1/1 [==============================] - 0s 10ms/step - loss: 4.7362
Epoch 3/50
1/1 [==============================] - 0s 10ms/step - loss: 3.9037
Epoch 4/50
1/1 [==============================] - 0s 9ms/step - loss: 3.2450
Epoch 5/50
1/1 [==============================] - 0s 8ms/step - loss: 2.7233
Epoch 6/50
```

Keras Training Options

- “model.compile” method in Keras allows you to configure various aspects of the training process for your neural network model.
- Parameters:
  - optimizer, loss, and metrics
- Optimizer
  - 'sgd': Stochastic Gradient Descent.
  - 'adam': Adam optimizer, an adaptive learning rate optimization algorithm.
  - 'rmsprop': Root Mean Square Propagation
- Loss
  - 'binary_crossentropy': suitable for binary classification problems.
  - 'categorical_crossentropy': suitable for multi-class classification
- Metrics (evaluate model)
  - 'accuracy': Accuracy metric, most common.
  - 'mean_squared_error': suitable for regression problems.

Optimize learning rate

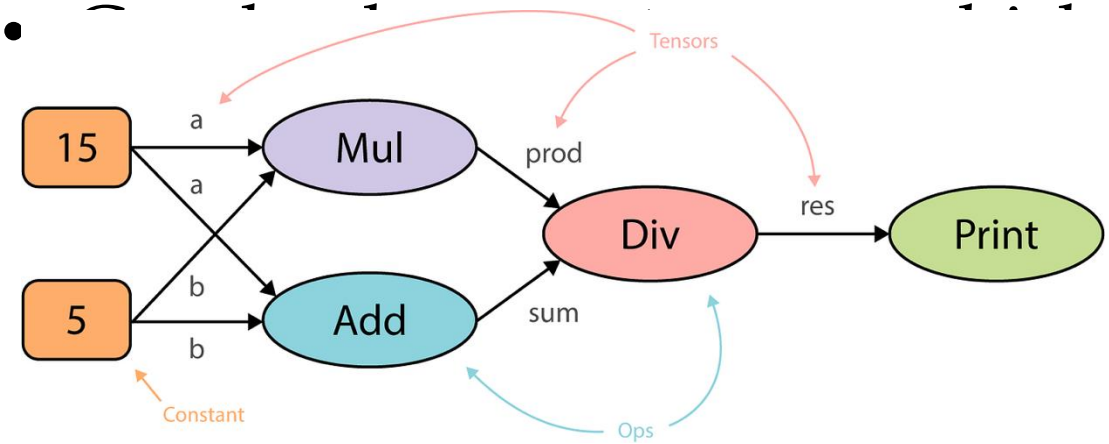
- For SGD or Adam,
  - configure the learning rate to control the step size during weight updates.
  - custom_optimizer = SGD(learning_rate=0.01)
  - model.compile(optimizer=custom_optimizer, loss='binary_crossentropy', metrics=['accuracy'])
- Multiple metrics
  - model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy', 'precision', 'recall'])
- Callbacks – saving the best model
  - EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, etc.
  - early_stopping = EarlyStopping(monitor='val_loss', patience=3)
  - model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'], callbacks=[early_stopping])

What is TensorFlow

- **Key idea**: express a numeric computation as a **graph**

- Graph nodes are **operations** with any number of inputs and outputs

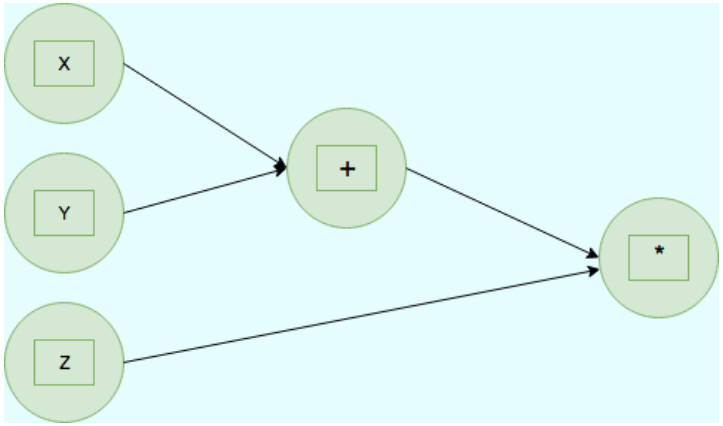
- flow between

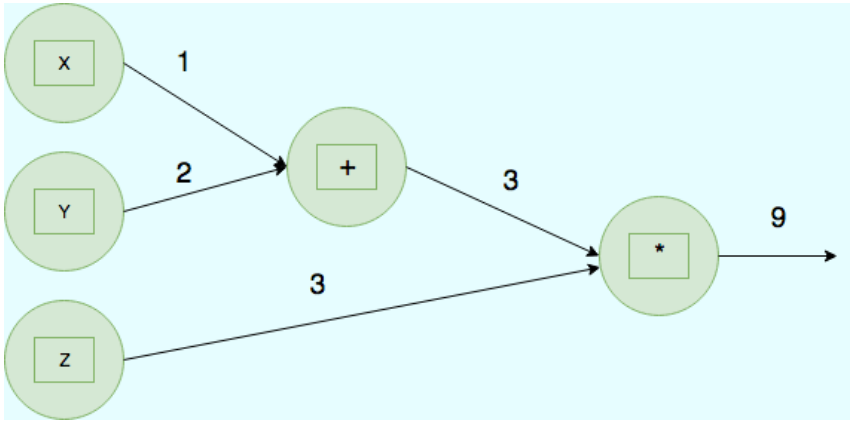


Neural Networks As Computational Graphs

- consider the relatively simple expression: f(x, y, z) = (x + y) * z.

f(x, y, z) = h(g(x, y) , z)
g(i, j) = i + j
h(p, q) = p*q

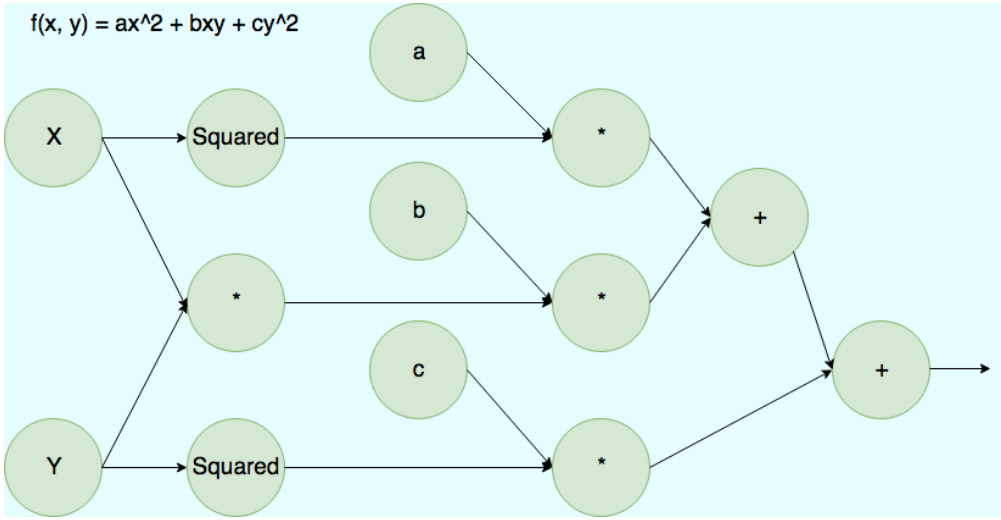


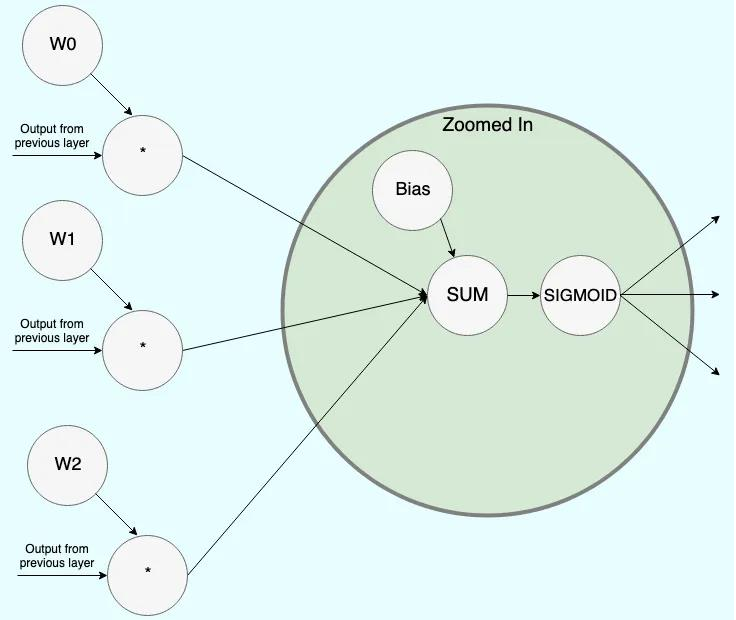


example for computing f(1, 2, 3).
f(1, 2, 3) = h(g(1, 2), 3)
g(1, 2) = 1 + 2 = 3
f(1, 2, 3) = h(3, 3)
h(3, 3) = 3*3 = 9
f(1, 2, 3) = 9

Neural Networks As Computational Graphs

- Consider this function as an example:
- $f(x, y) = ax^2 + bxy + cy^2$; where a, b, and c are scalars

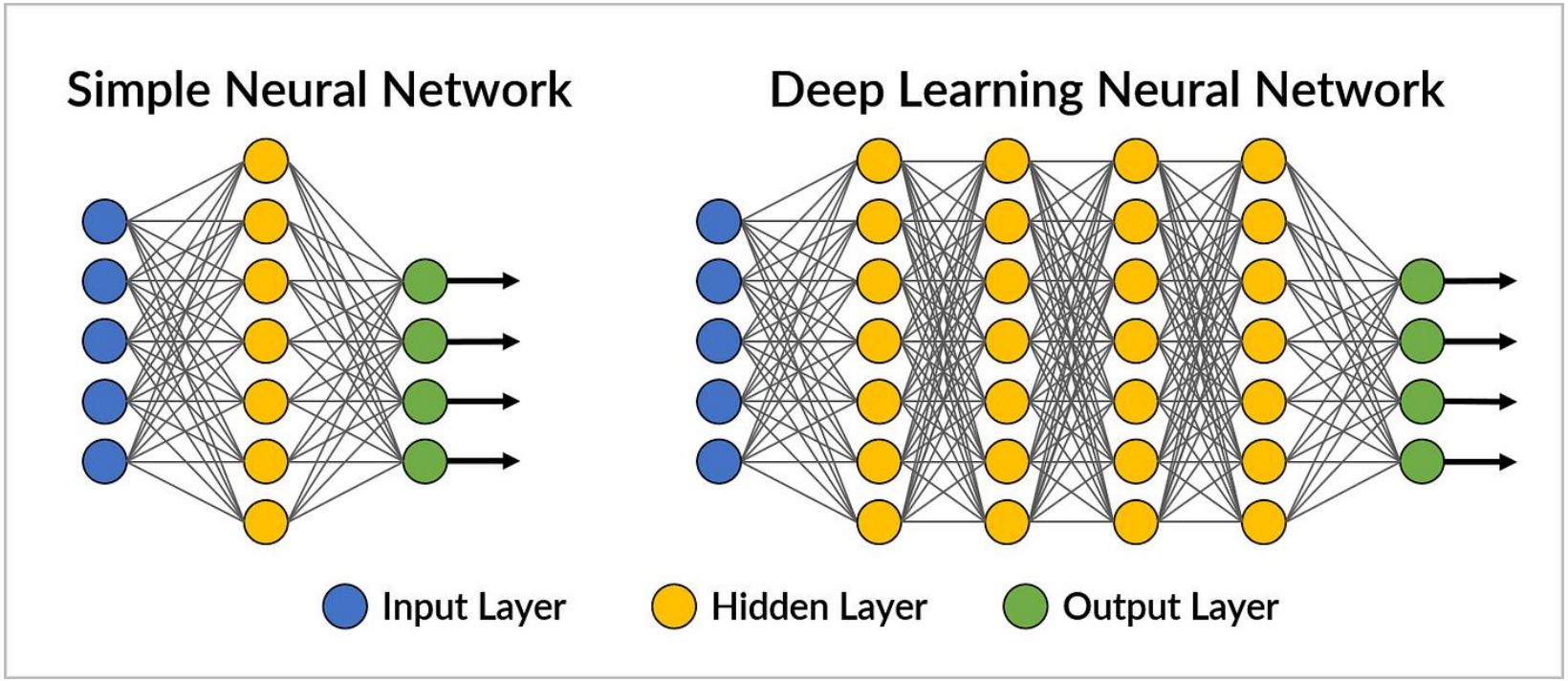




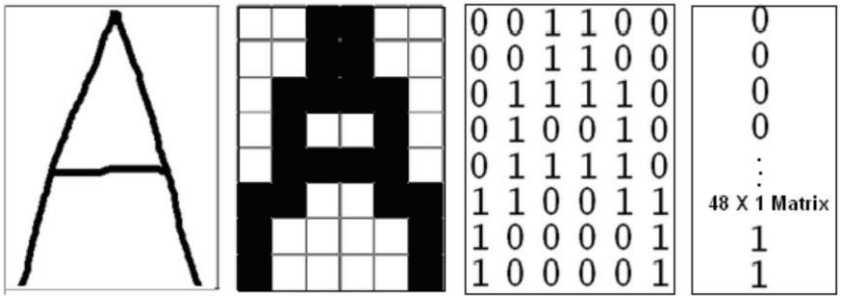
REF: http://rb.gy/nbhtx5

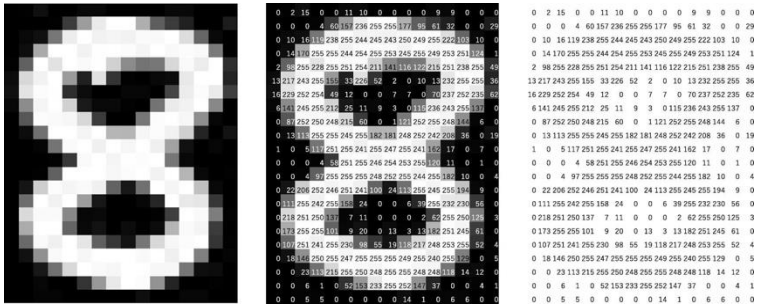
Deep Neural Network (DNN)

- **Deep Neural Network (DNN)** is a neural network with **multiple hidden layers** between the input and output.
- **NN**: Few layers → learns simple patterns
- **DNN**: Many layers → learns **complex, hierarchical patterns**



Neural Network Input: Digital Image

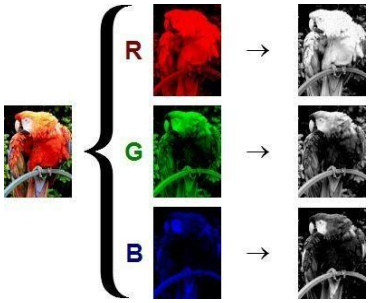


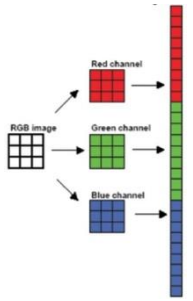


Digital image - 2D set of pixels.

Each pixel has numeric value(s) associated to it:

- Monochrome: 0, 1
- Grayscale: 0 (black) - 255 (white)
- Color: 0 - 255 (in each channel: R, G, B)

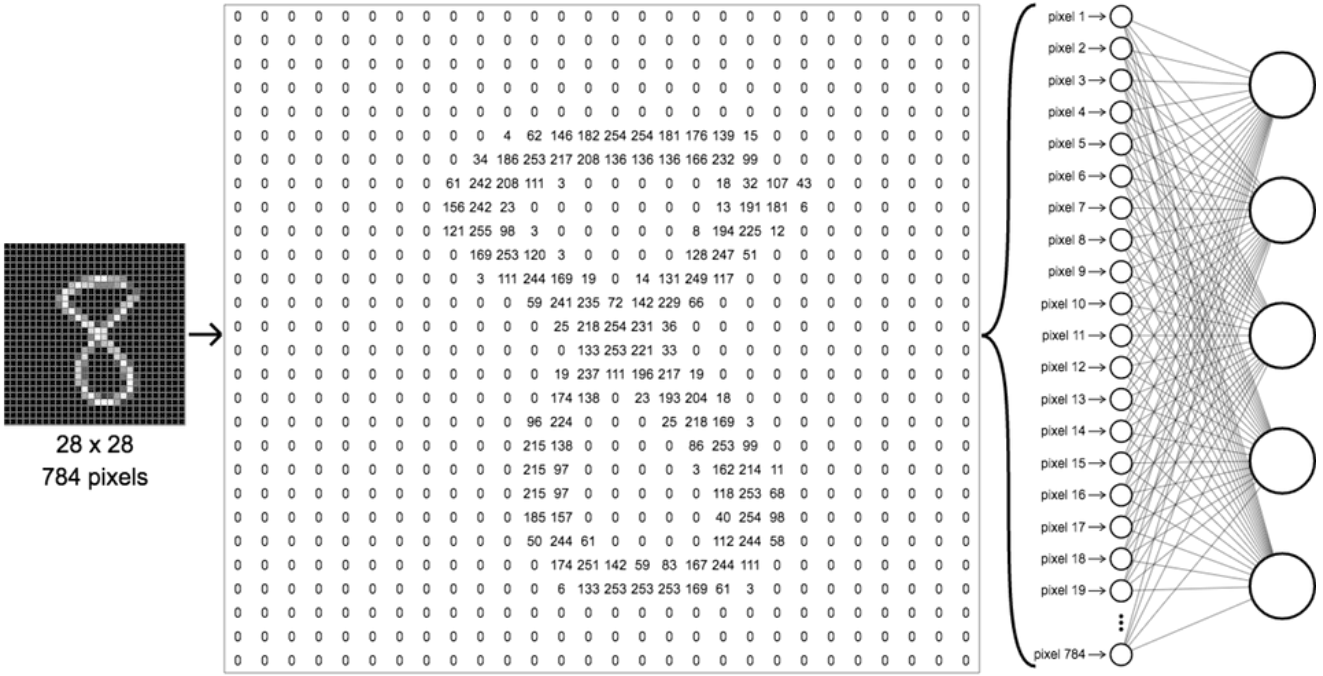




Computer sees an array of numbers.

Problem with fully-connected input layer

- Explosion of parameters
- Spatial information lost



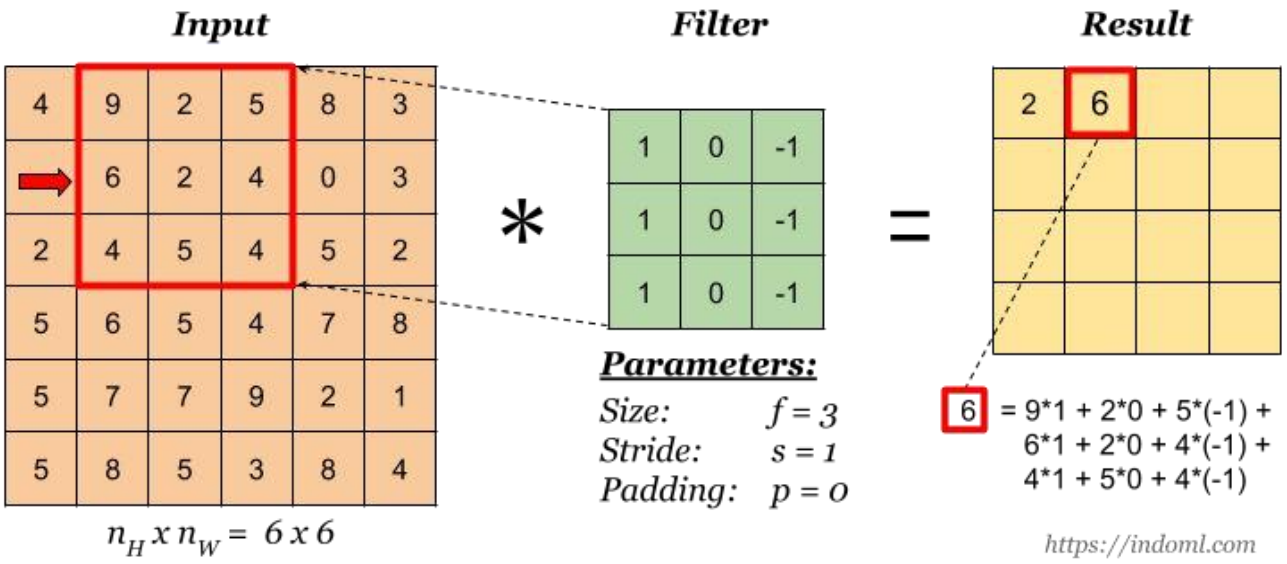
Solution: Convolution

How to preserve that valuable spatial structure of the input?

- Feed a small patch of the input image to a single neuron in a hidden layer
- Each neuron in that hidden layer is seeing only a patch in the input image

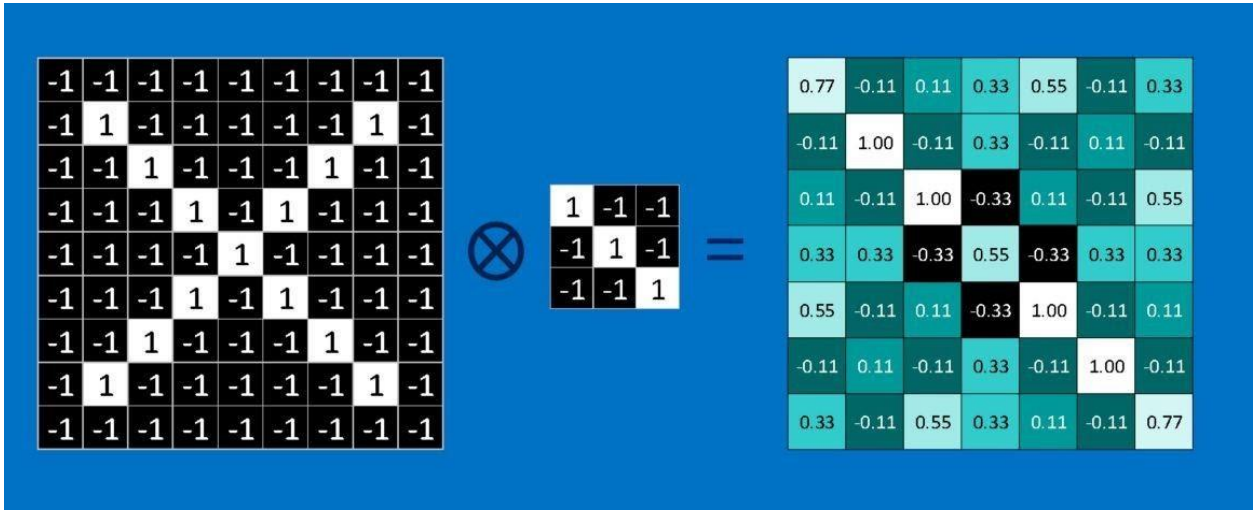
How to extract features? (fundamental task of DNN)

- Use convolution as a mapping function of a patch into a neuron



Convolution

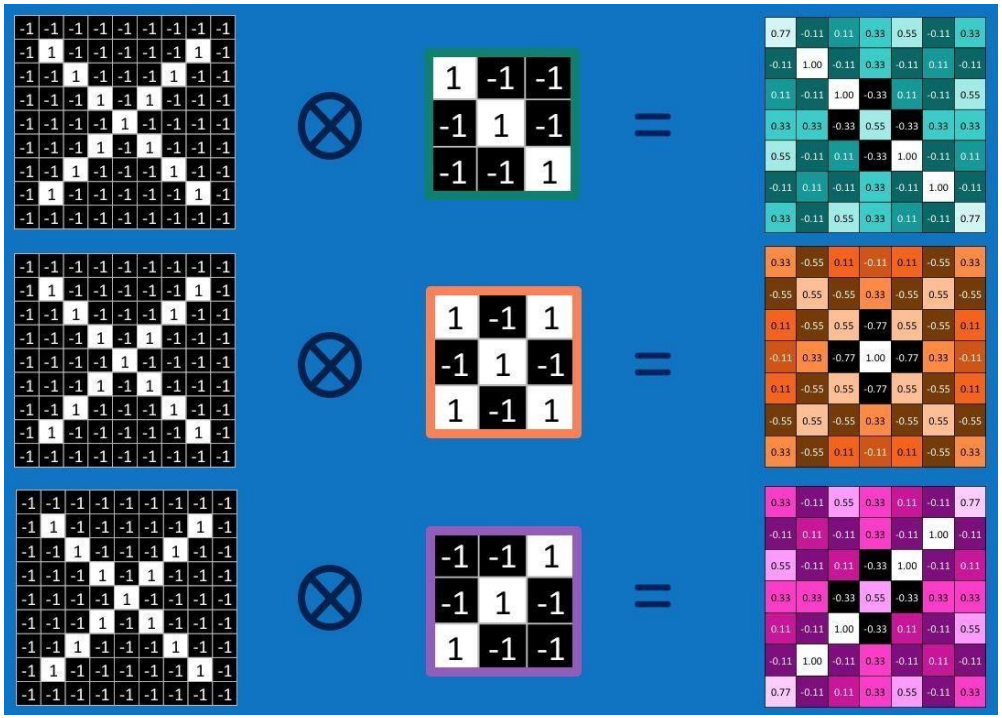
- Measure of similarity between a filter (kernel, small feature) and currently observed region in the input image.
- Detects that image contains given feature and creates an output matrix - a feature map. Feature map tells where in the image is the activation for this particular filter
- Depth of a filter must match the depth of the input for the filter (e.g. the number of channels).



Feature Maps

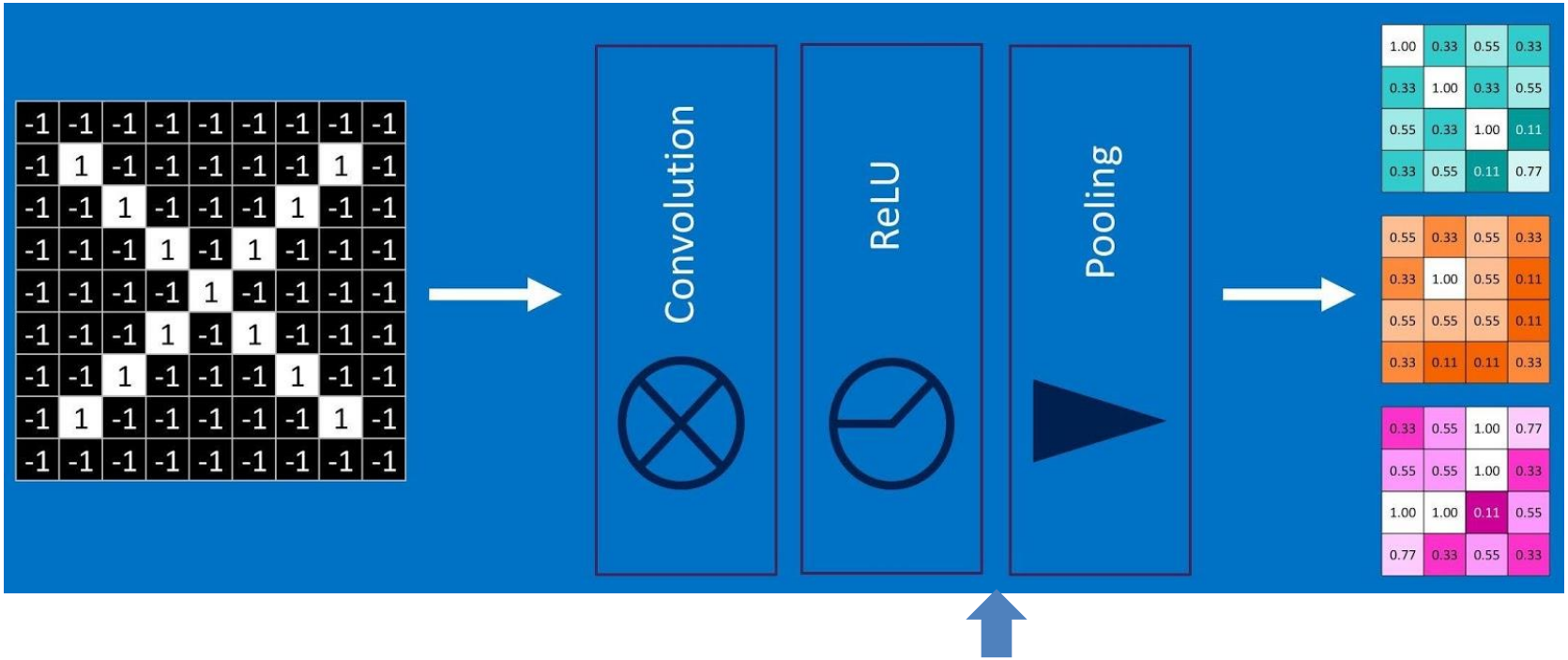
Convolution layer creates a feature map for each filter.

The same pixel in each feature map is connected to the same patch of the input image.



Stack

multiple CNN layers placed one after another, where the output of one layer becomes the input to the next.

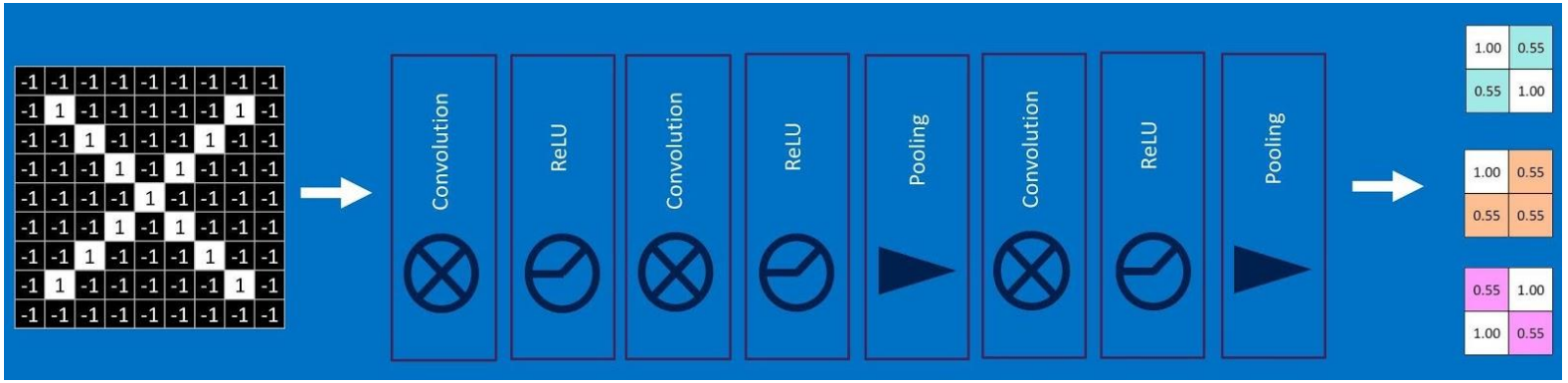


Batch Normalization (BatchNorm) is a technique that **normalizes the outputs of a layer during training** so that the network trains **faster, more stably, and more accurately**. [Mean, Std. Dev, Normalize, Scale, Shift)

Deep Stacking

With each new convolutional layer:

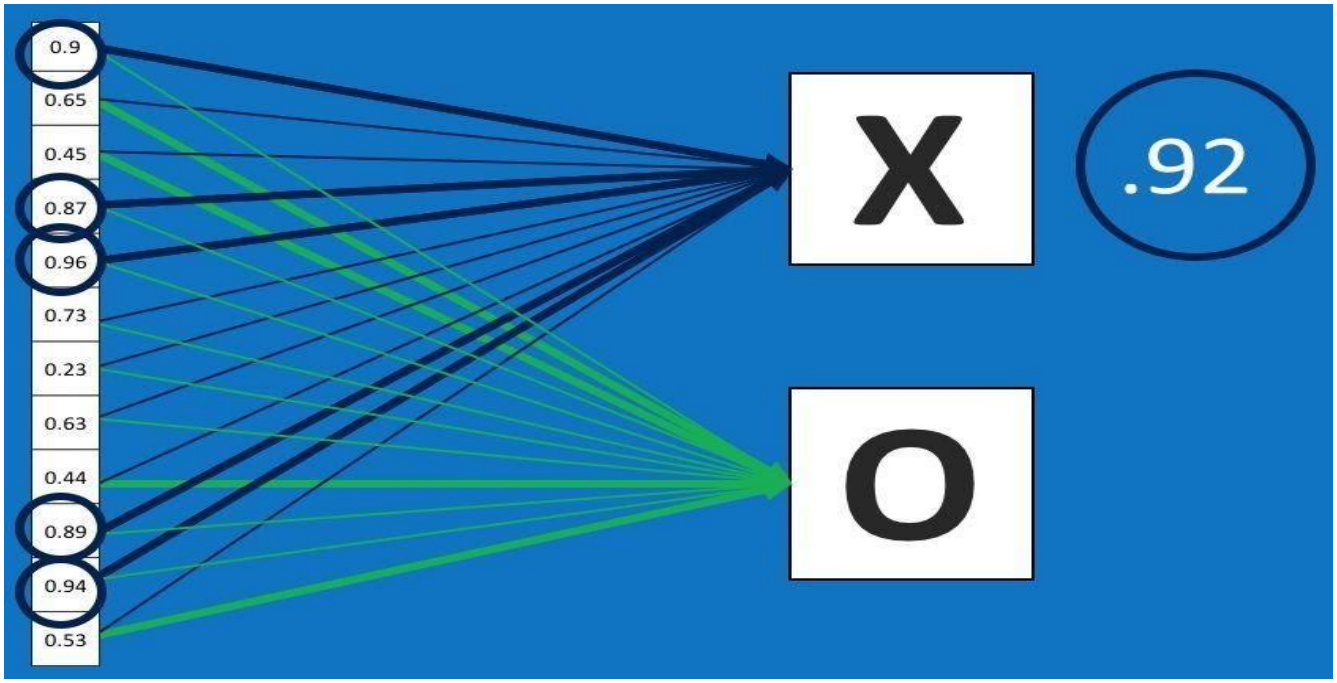
- filters become larger and learn larger and more complex features
- feature maps are shrinking and become smaller



Fully-connected layers

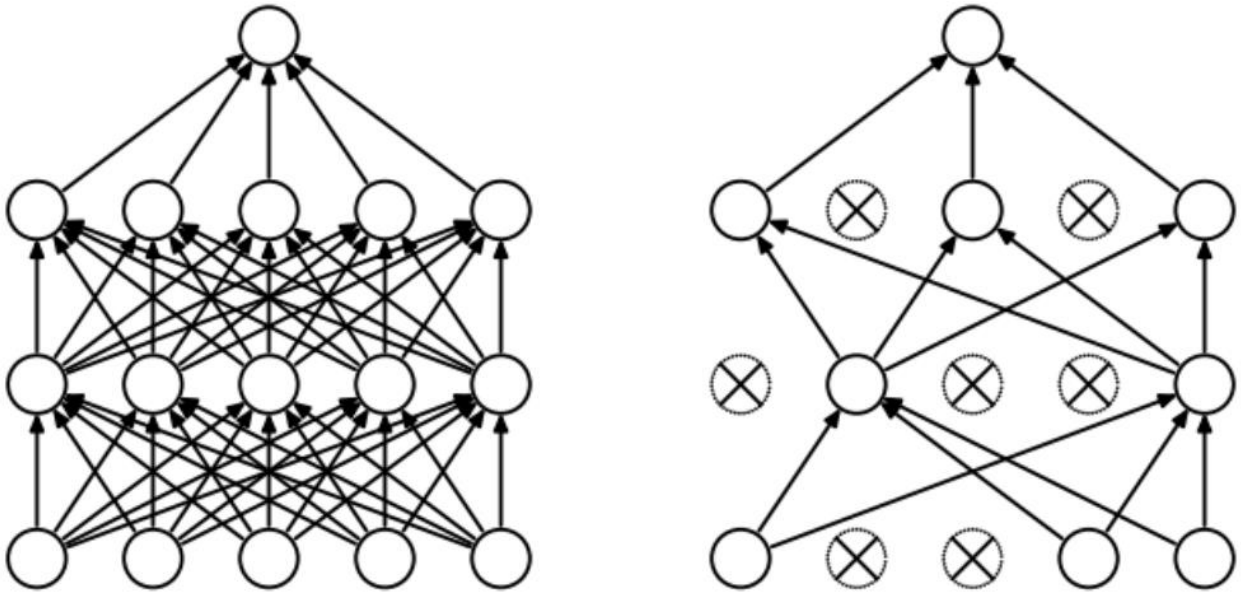
They learn non-linear combinations of the high-level features outputted by the convolutional layers.

Features are assigned weights which describe their contribution in recognizing particular object.



Dropout

- **Dropout** is a **regularization technique** where, during training, a **random subset of neurons is temporarily turned off** to prevent overfitting.
- DNN - Randomly drops individual neurons
  - Common dropout rates: **0.3 – 0.5**
- CNN - Drops entire feature maps (channels)
  - Common dropout rates: **0.1 – 0.3**

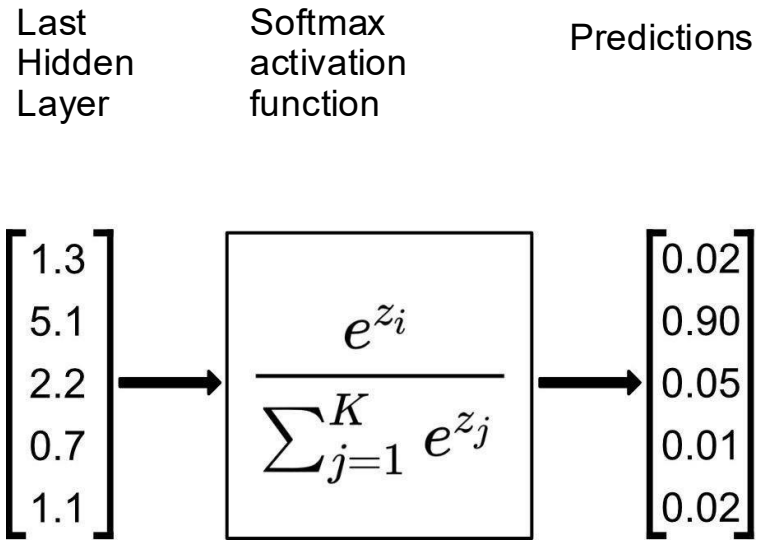


(a) Standard Neural Net

(b) After applying dropout.

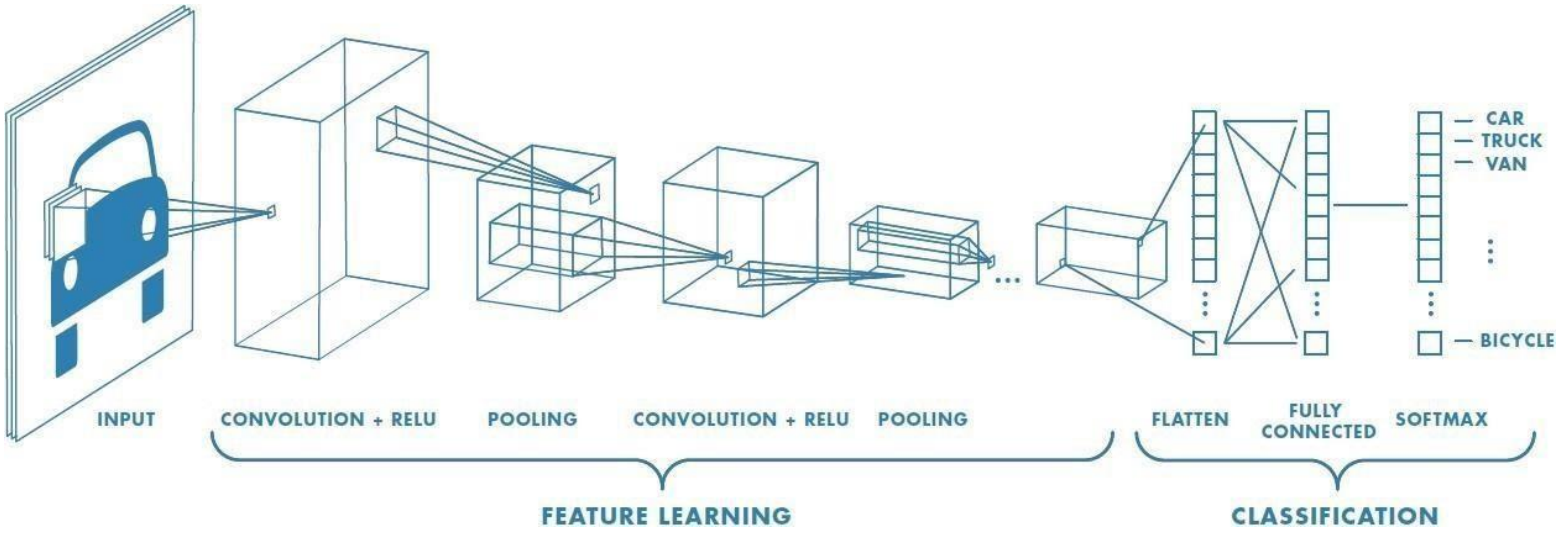
Softmax Layer

- Softmax is implemented through a neural network layer just before the output layer.
- Softmax layer must have the same number of nodes as the output layer.
- Softmax function is a mathematical function that takes a vector of real numbers as input and returns a probability distribution over the elements of the vector.



CNN Architectures

The first hidden layer is always a convolutional layer.



Hands-On