Reinforcement Learning (RL)

Chapter 8:

Introduction to ANN and Deep Learning

Saeed Saeedvand, Ph.D.

Outline

- Linear classification
- Cost functions
- Optimization
- Backpropagation Algorithm
- Artificial Neural Networks
- Deep learning

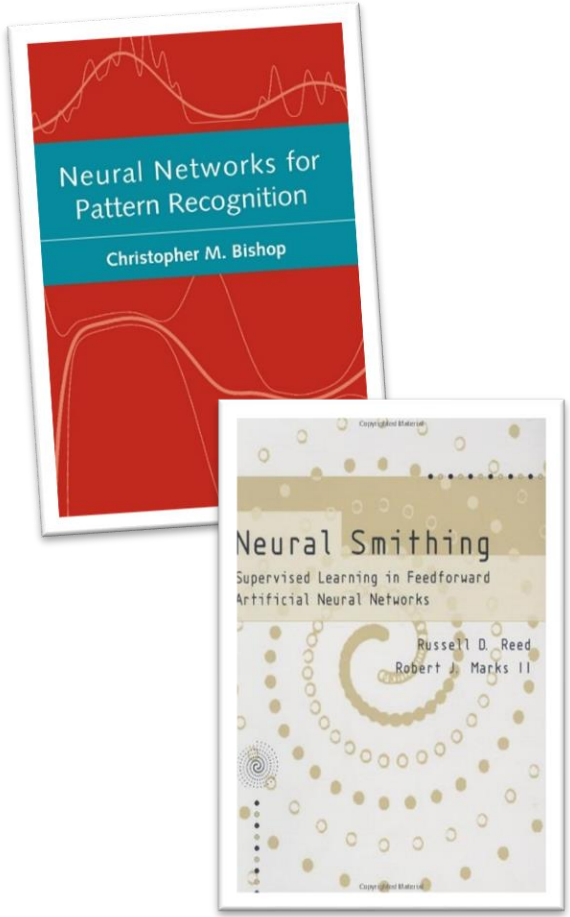
Resources

- **Neural Networks for Pattern Recognition**

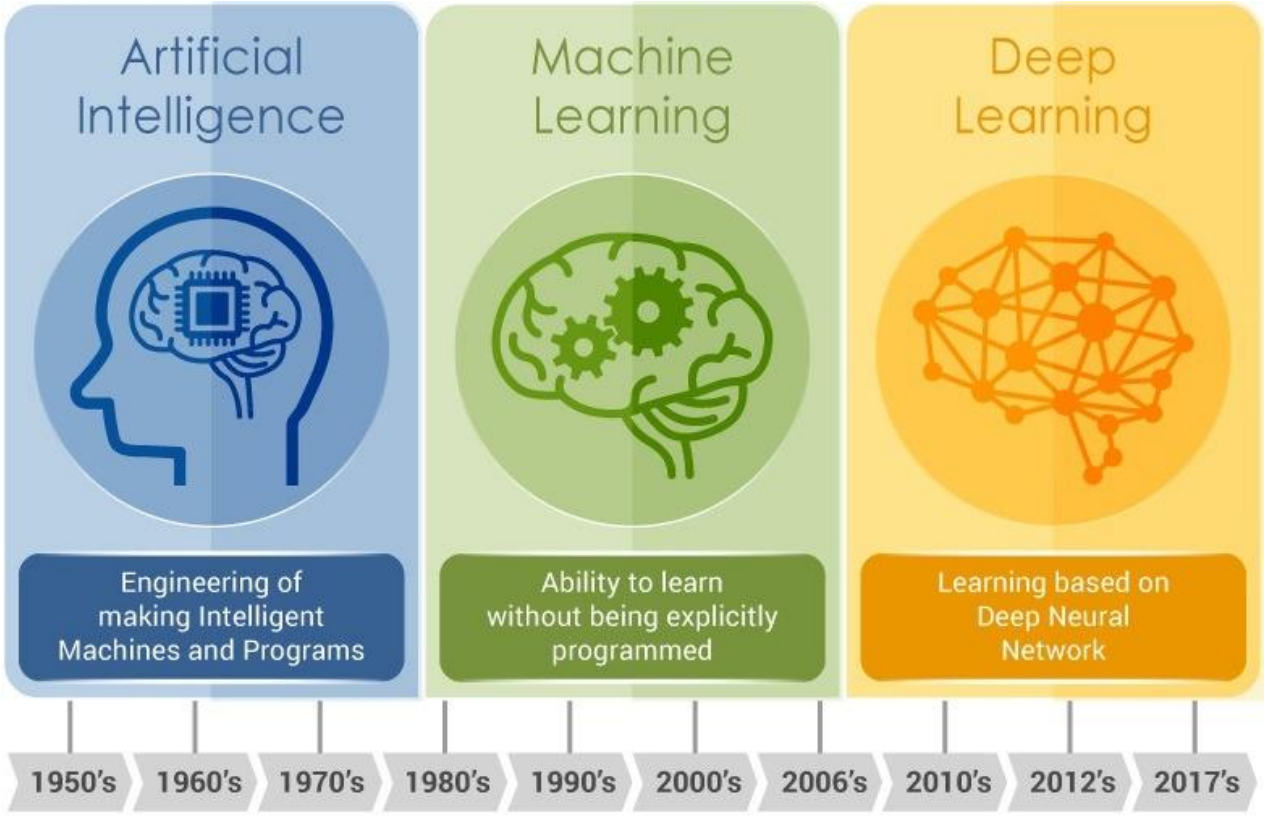
  [Bishop]

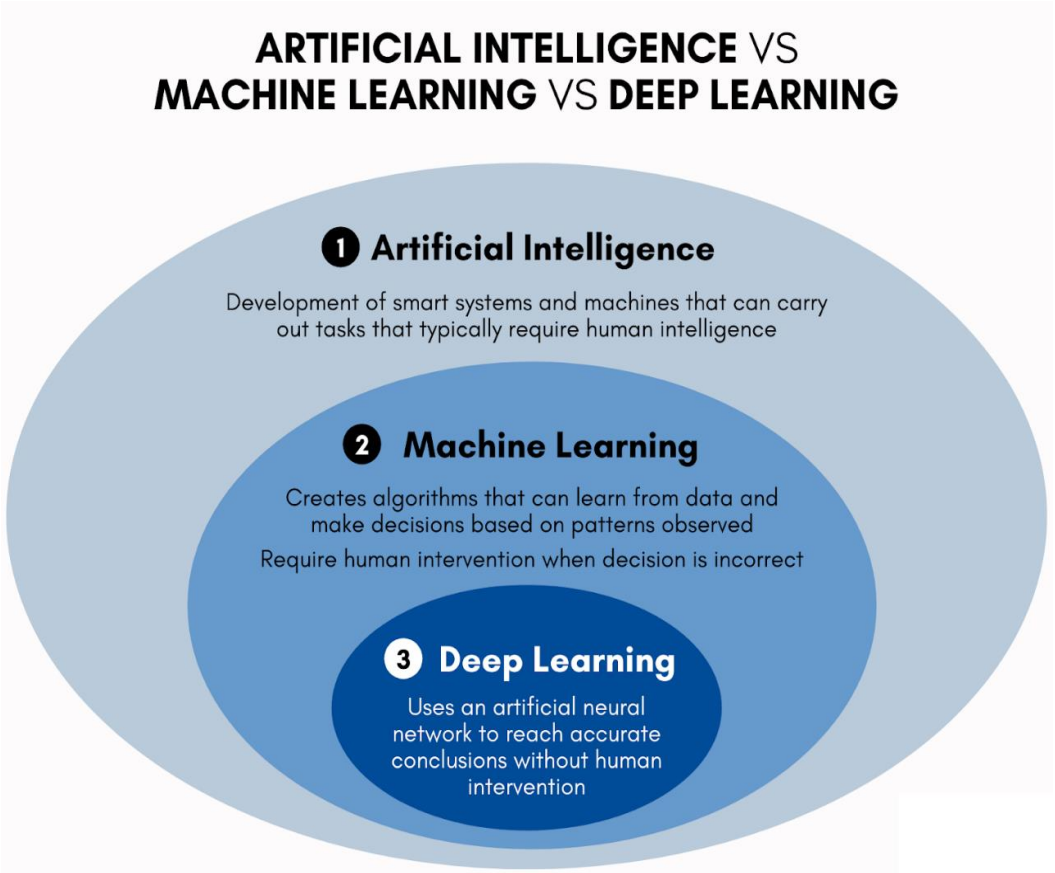
- **Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks**

  [Russrell]



What is Deep learning?





Deep Learning Challenges

✓ Although Neural Networks date back decades they became popular again mainly because of 3 reasons:

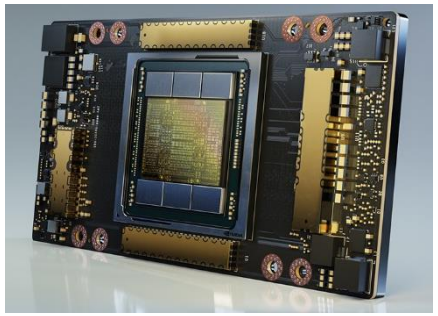
1. Big Data

- Larger Datasets
- Easier Collection and Storage

2. Hardware

- Graphics Processing Units (GPU)

NVIDIA H100 Tensor Core GPU

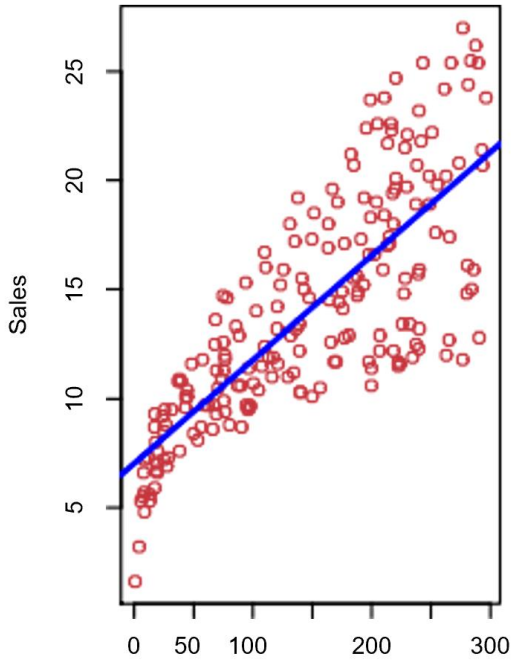


3. Software

- New Models and Techniques
- New Toolboxes

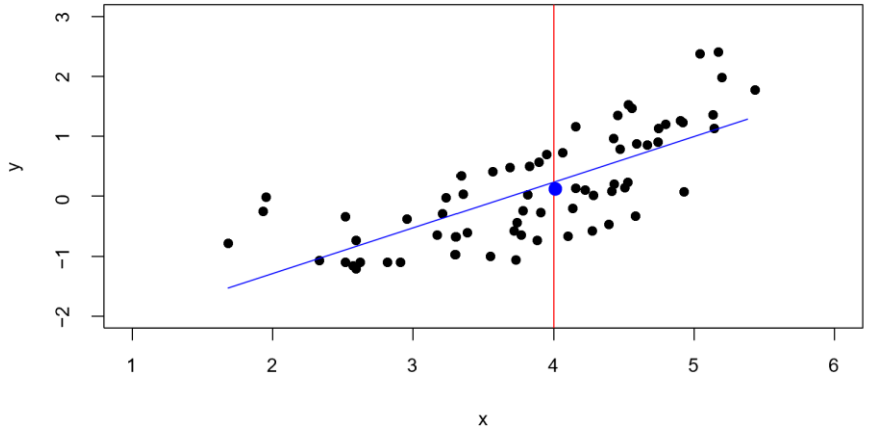
linear-regression

Advertisement on TV data



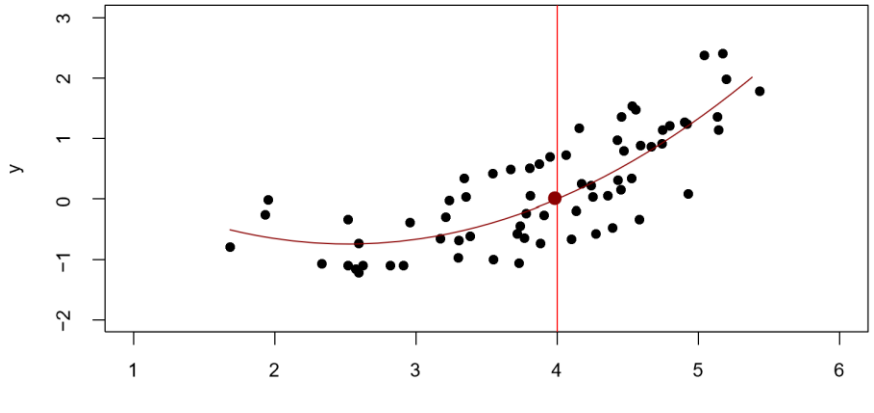
A linear model with a reasonable fit here

$$\hat{f}(X) = \hat{\beta}_0 + \hat{\beta}_1 X$$



A non-linear model with better fit here (quadratic model)

$$\hat{f}(X) = \hat{\beta}_0 + \hat{\beta}_1 + \hat{\beta}_2 X^2$$



Why we need Deep Learning

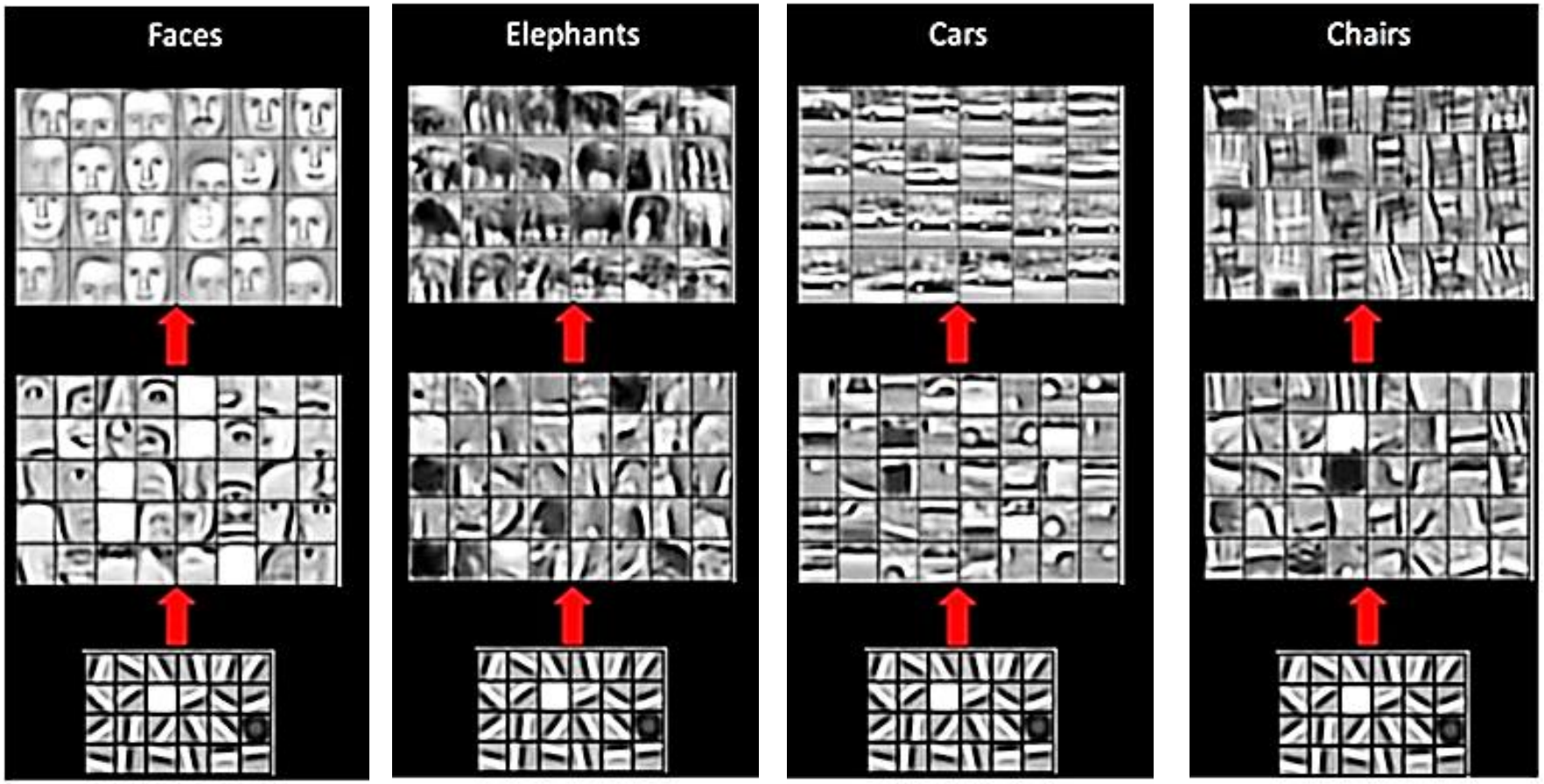
- ✓ Finding manual features are hard and time consuming
- ✓ Even if do it usually it not scalable in practice

Is it possible to learn the underlying features directly from data?

High level features

Mid level features

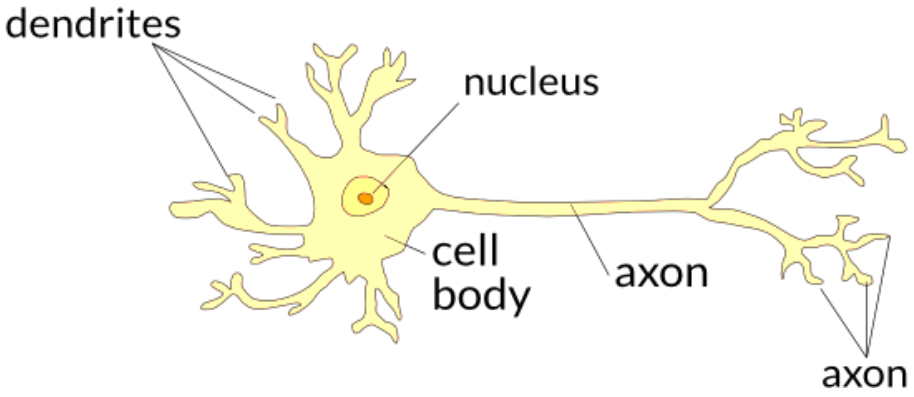
Low level features



Neural Network (A neuron or Perceptron)

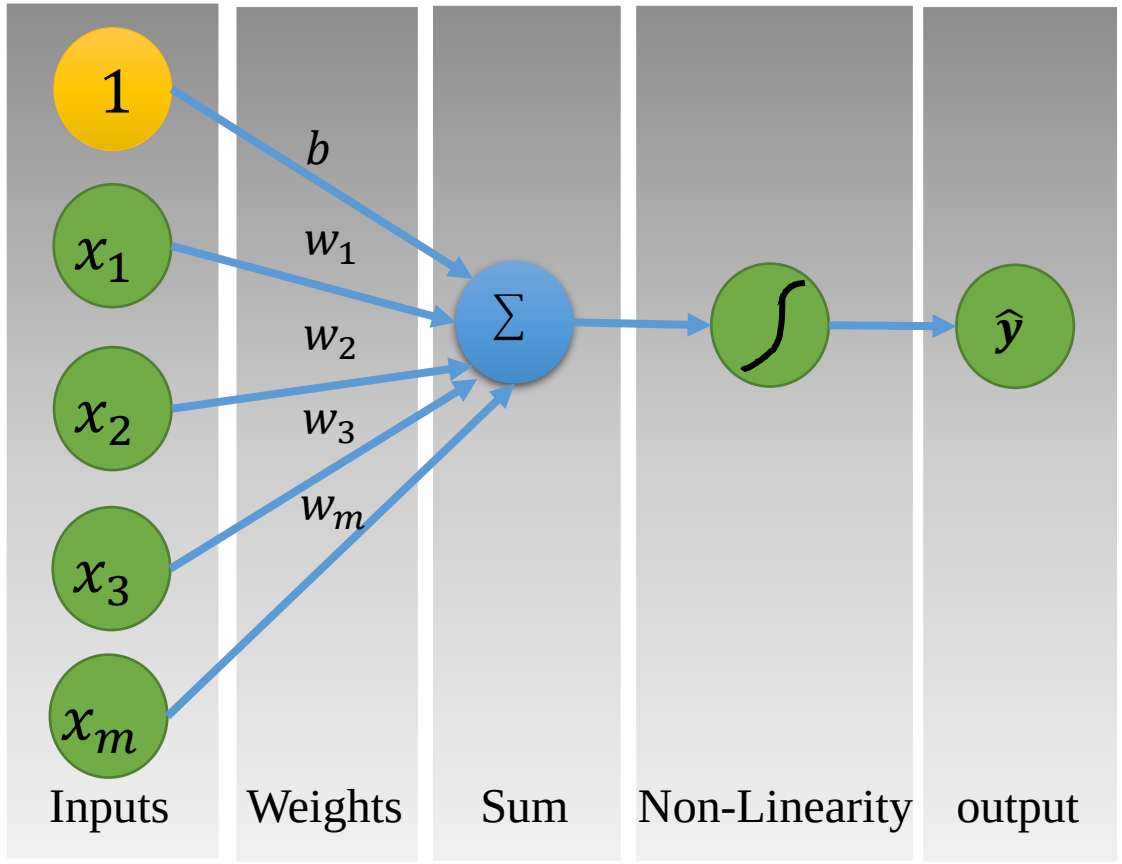
Artificial neurons

- ✓ Artificial neurons also known as Perceptron, or Nodes
- ✓ They are the simplest elements of a neural network.
- ✓ Artificial neurons are inspired by biological neurons that are found in the human brain.



Neural Network (A neuron or Perceptron)

Forward Propagation



Weights that we want to find

Bias (shift to left or right)

$$\hat{y} = h(b + \sum_{i=0}^{m} x_i w_i)$$

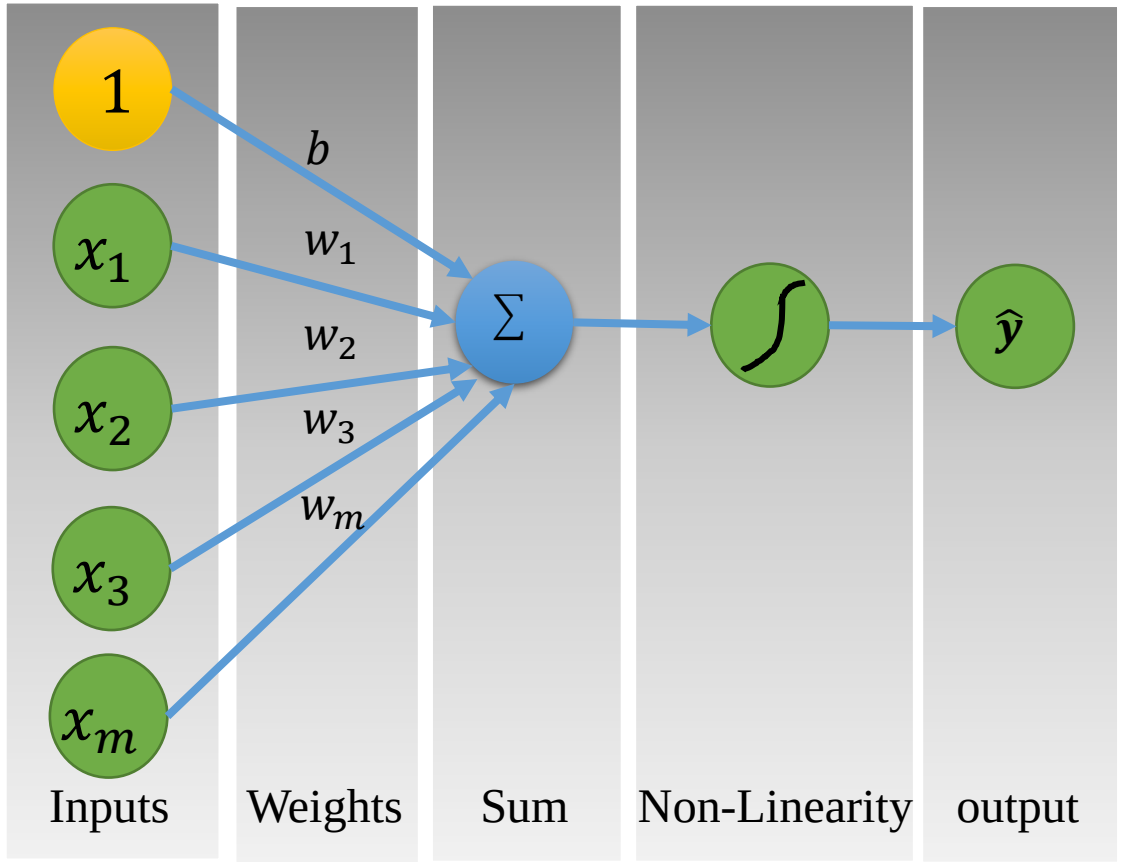
output

Non-Linear activation function to produce output

Inputs' linear combination of inputs

Neural Network (A neuron or Perceptron)

Forward Propagation



$$\hat{y} = h(b + \sum_{i=0}^{m} x_i w_i)$$

In form of Linear Algebra

$$\hat{y} = h(b + X^T W)$$

$$X = \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix} \qquad W = \begin{bmatrix} w_1 \\ \cdot \\ \cdot \\ \cdot \\ w_m \end{bmatrix}$$

Activation Functions

- ✓ The Purpose of activation functions is to **introduce non-linearities** into the network
  - Linear activation functions **results linear decision**
  - Non-linear activation functions results in **approximating arbitrarily complex function**

Linear function

Non-linear function

Activation Functions

- ✓ Deciding **what goes to the next neuron**
- ✓ Millions of parameters after calculating **weighed sum can go out of limit** (restrict output of the neuron to a certain limit)

Non-Linear activation function

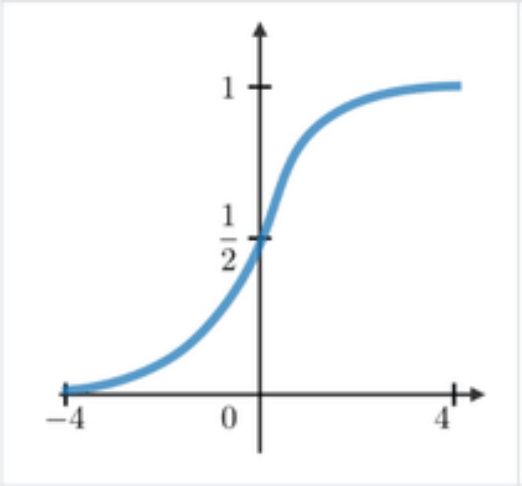
$$\hat{y} = h(b + X^T W)$$

All activation functions are non-linear

Activation Functions

$$\hat{y} = h(b + X^T W)$$

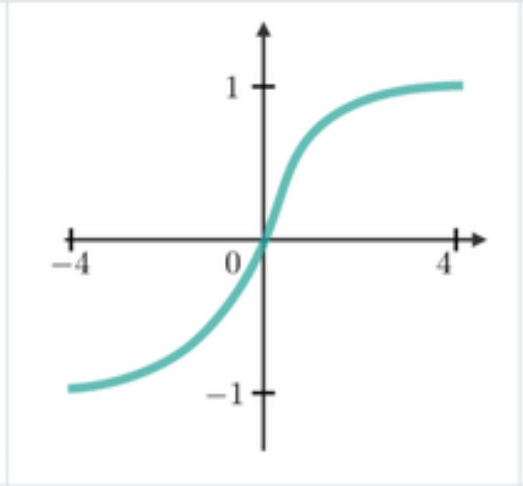
Sigmoid



$$h(z) = \frac{1}{1 + e^{-z}}$$

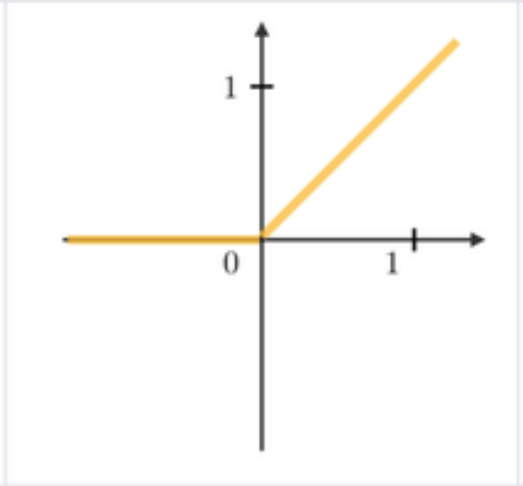
Popular and suitable for probability because [0, 1]

Hyperbolic/Tanh



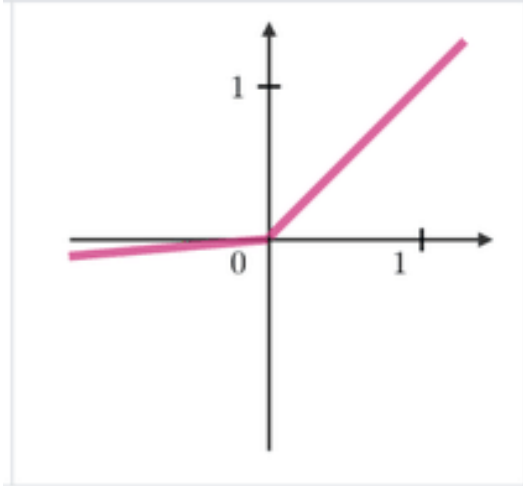
$$h(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

ReLU



$$h(z) = \max(0, z)$$

Leaky ReLU



$$h(z) = \max(\epsilon z, z)$$
With $\epsilon \ll 1$

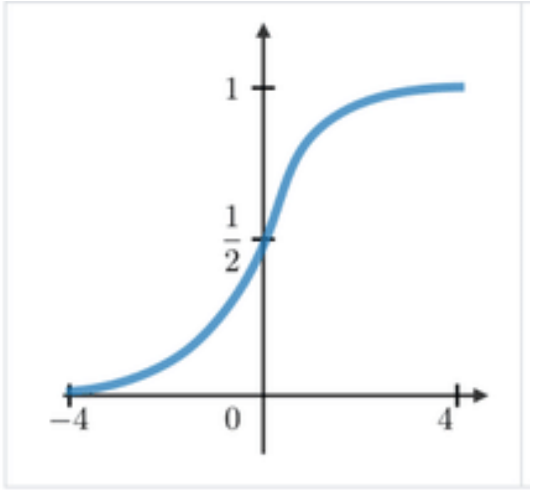
Activation Functions

Gradient vanishing Problem

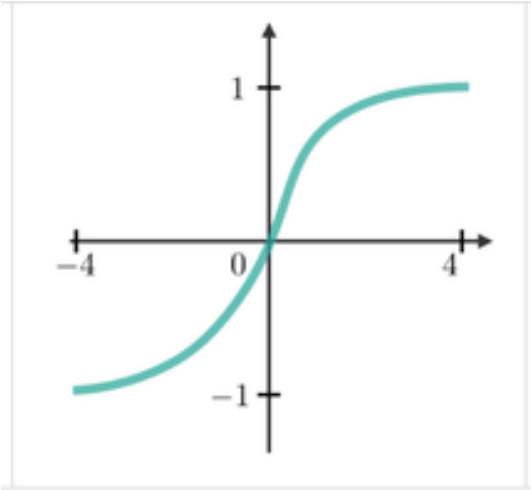
- ✓ Using **some activation function**; after each perceptron may can **reduce numbers every time and this can be problematic**

- ✓ Especially with normalized data as input between [0,1]

Sigmoid



Hyperbolic/Tanh



Activation Functions

Gradient vanishing Problem

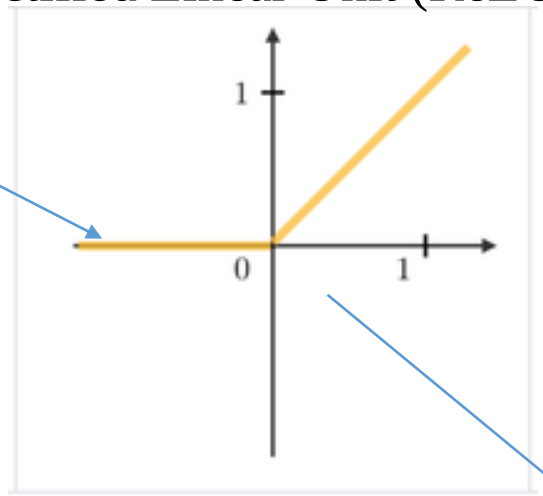
Solution

✓ Solution is using some activation functions that don't have this problem in big Neural Networks.

A common activation function in Deep Learning

Rectified Linear Unit (ReLU)

Ignore negative or bad outputs



$h(z) = \max(0, z)$

Pass patterns linearly

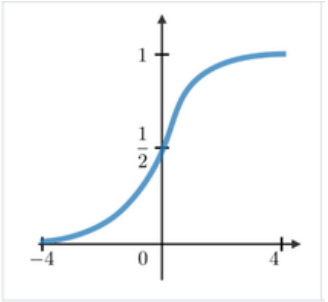
Is ReLU linear?

- ➢ No because ReLU 's output is not a straight line!

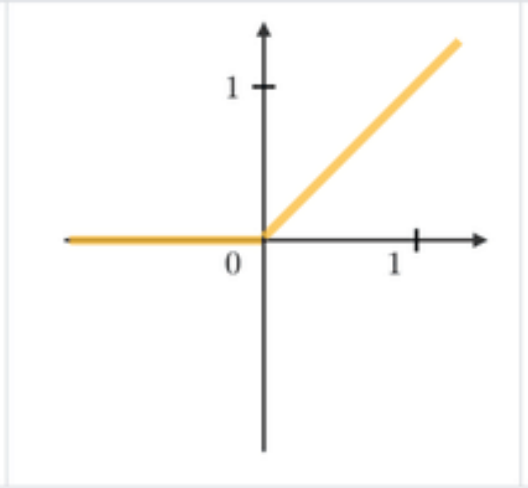
Activation Functions

Disadvantage of ReLU

- ✓ There is no limitation for the output of the Relu, but e.g. Sigmoid have it.



Rectified Linear Unit (ReLU)



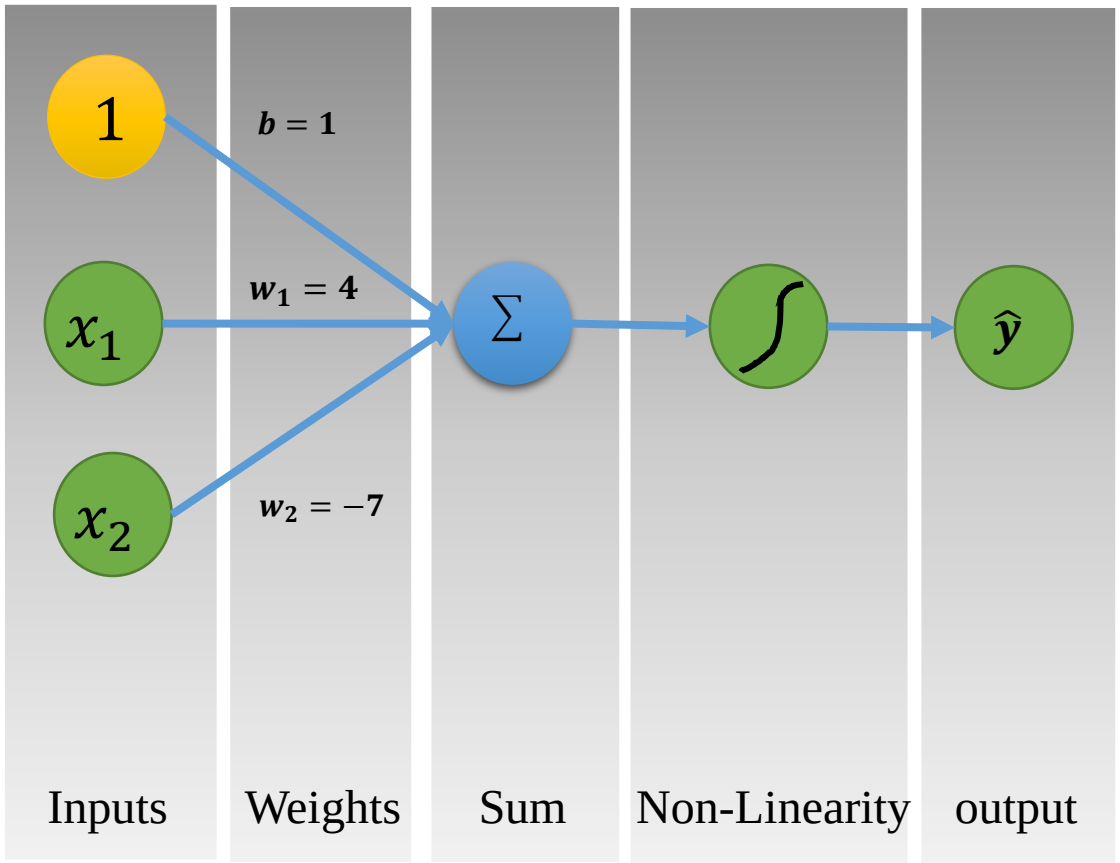
$h(z) = \max(0, z)$

Solution

- ✓ Batch normalization:
  - Normalize the output of the previous layers
  - Maintain the distribution of the data
  - It can be used at several points in between the layers (before or after the activation function)
  - Normalizing the outputs using mean=0, standard dev=1 (μ=0,σ=1).

Neural Network (Forward Propagation)

Example 1



For given weights:

$$b = 1$$

$$w = \begin{bmatrix} 4 \\ -7 \end{bmatrix}$$

$$\hat{y} = h(b + X^T W)$$

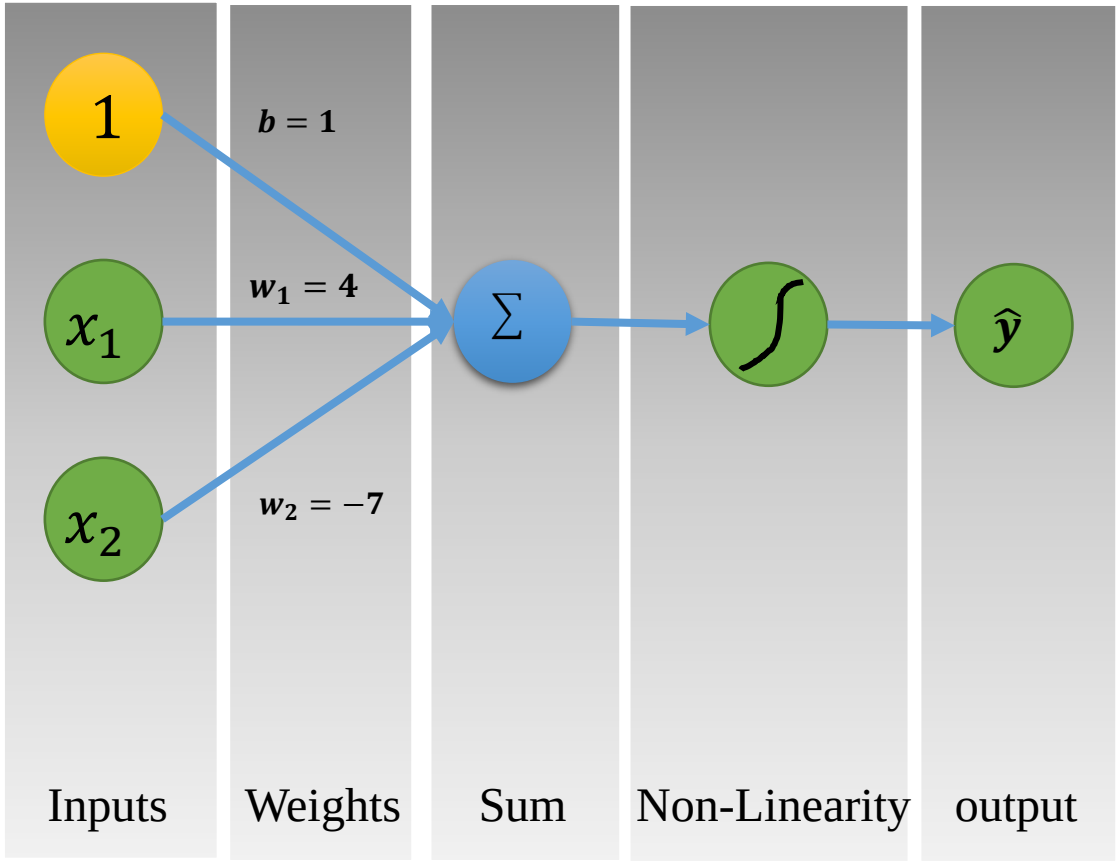
$$= h(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 4 \\ -7 \end{bmatrix})$$

$$\hat{y} = h(\underbrace{1 + 4x_1 - 7x_2}_{})$$

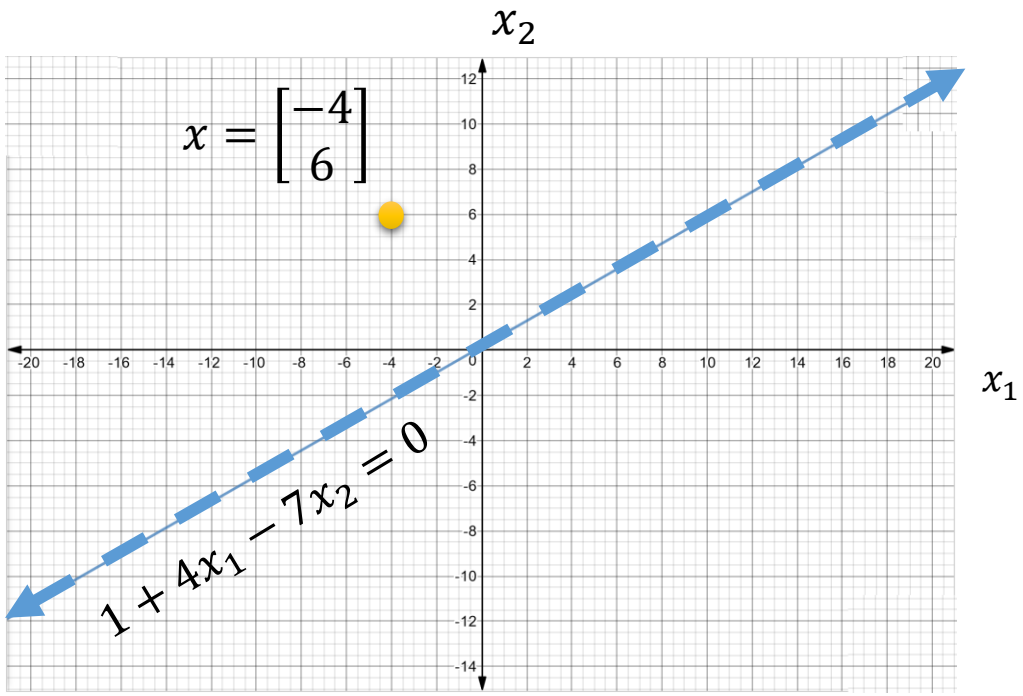
It's line in 2D

The Propagation: Example

Example 1



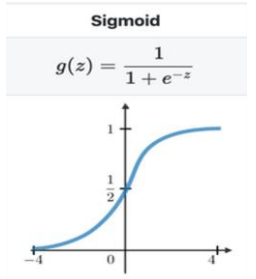
$$\hat{y} = h(1 + 4x_1 - 7x_2)$$



If input: $x = \begin{bmatrix} -4 \\ 6 \end{bmatrix}$

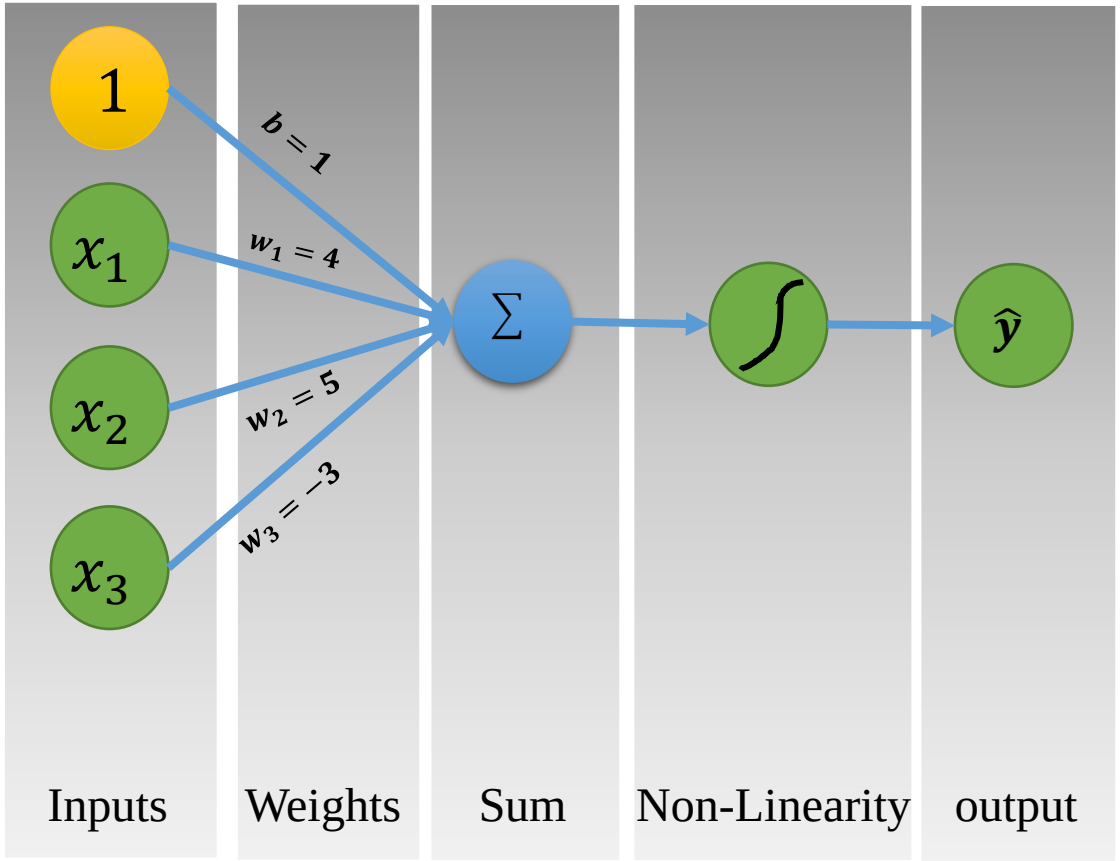
$$\hat{y} = h(1 + (4 * -4) - (7 * 6))$$
$$= h(-25) \approx 1.38$$

$sigmoid$



Neural Network (Forward Propagation)

Example 2



For given weights:

$$b = 1$$

$$w = \begin{bmatrix} 4 \\ 5 \\ -3 \end{bmatrix}$$

$$\hat{y} = h(w_0 + X^T W)$$

$$= h(1 + \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}^T \begin{bmatrix} 4 \\ 5 \\ -3 \\ 2 \end{bmatrix})$$

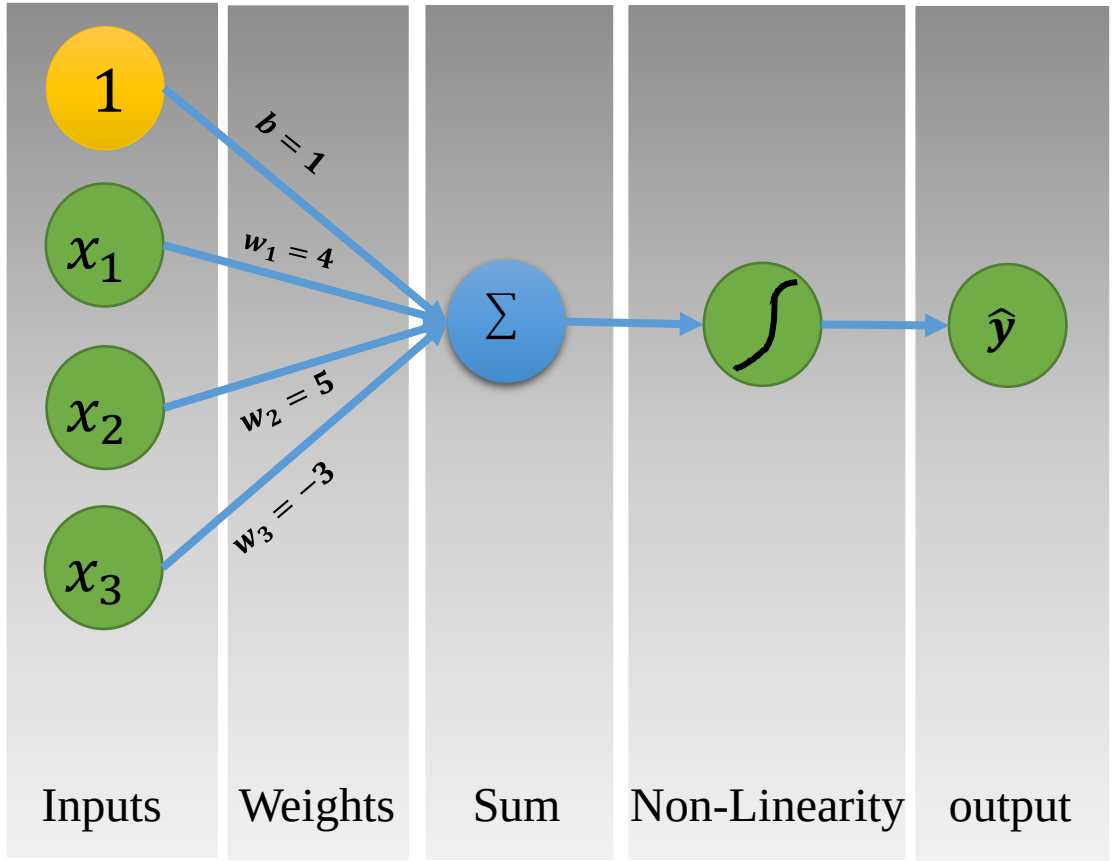
$$\hat{y} = h(\underbrace{1 + 4x_1 + 5x_2 - 3x_3}) $$

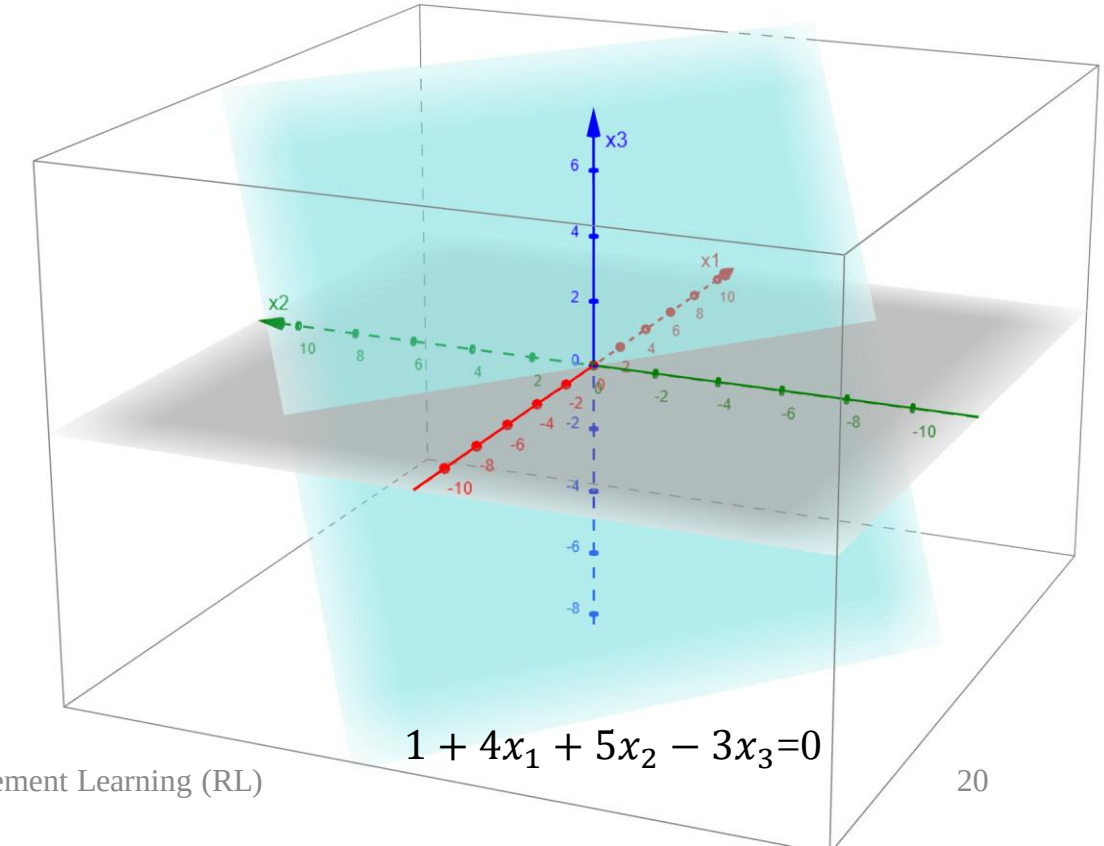
A plane in 3D!

Neural Network (Forward Propagation)

Example 2

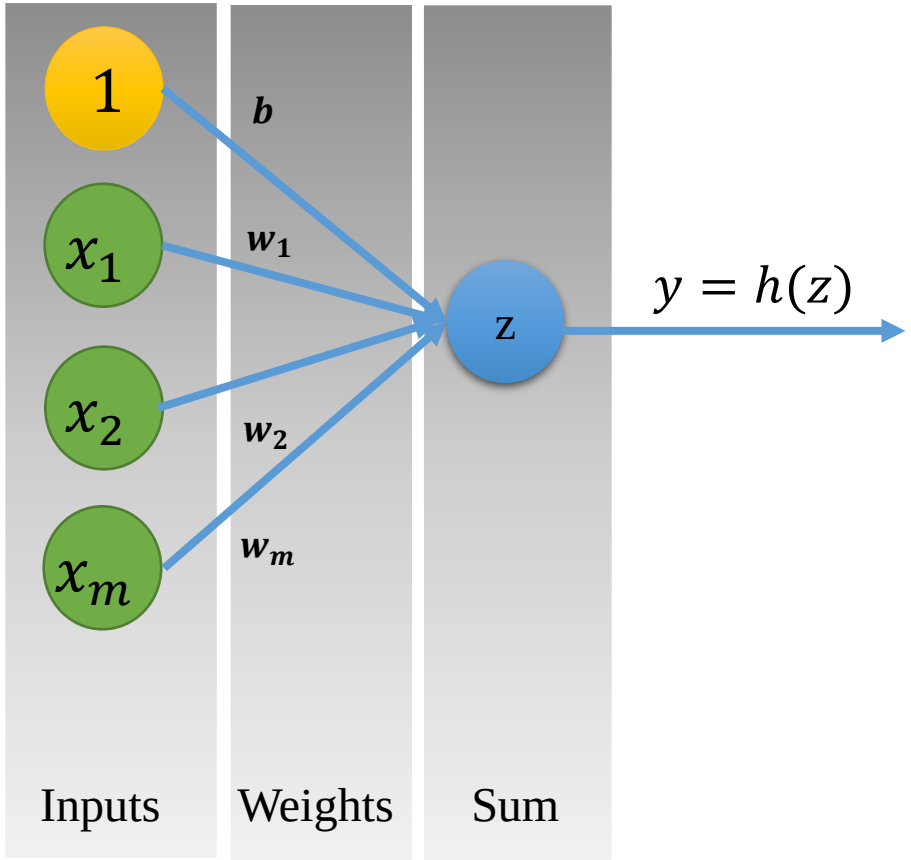
$$\hat{y} = h(1 + 4x_1 + 5x_2 - 3x_3)$$





The Propagation: Simplified

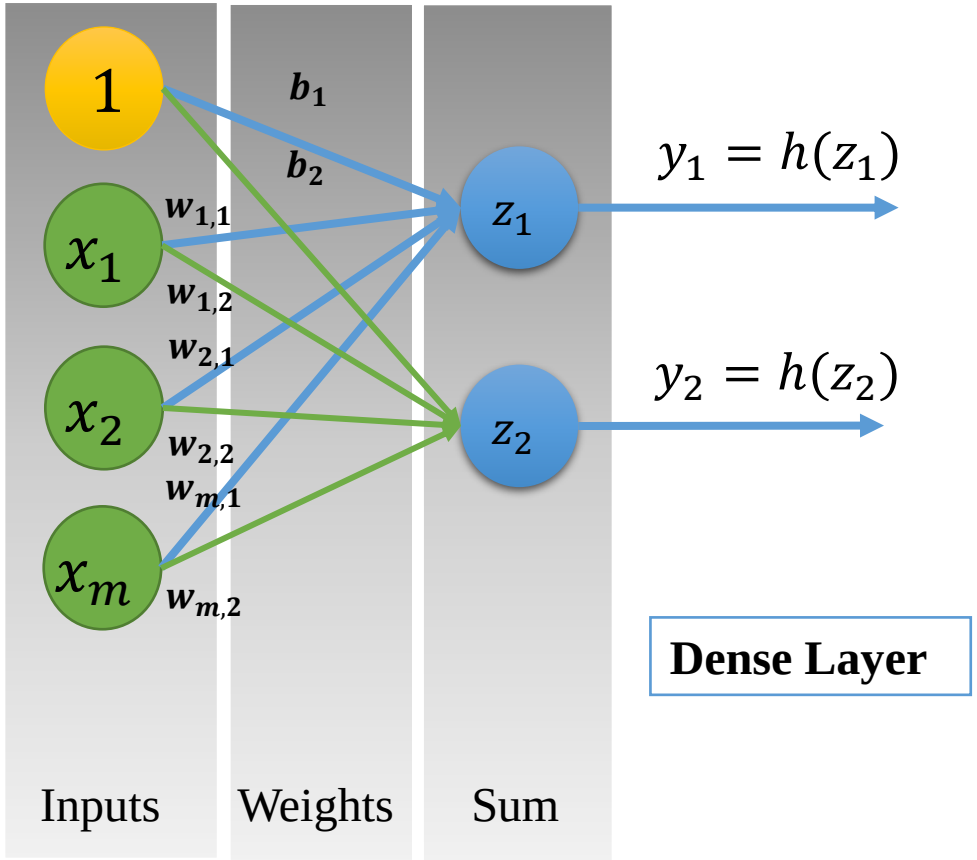
Neural Networks structure with Perceptron



$$z = b + \sum_{j=1}^{m} x_j w_j$$

The Propagation: Multi output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layer



Dense Layer

$$z_i = b_i + \sum_{j=1}^{m} x_j w_{j,i}$$

```
import torch.nn as nn
layer1 = nn.Linear(3, 2, bias=True)
```

Neural Network function approximation

✓ In practice Neural Network with minimum **one hidden layer** is able to approximate any function closely.

- We need to have proper amount of neurons
- We need to have proper activation function

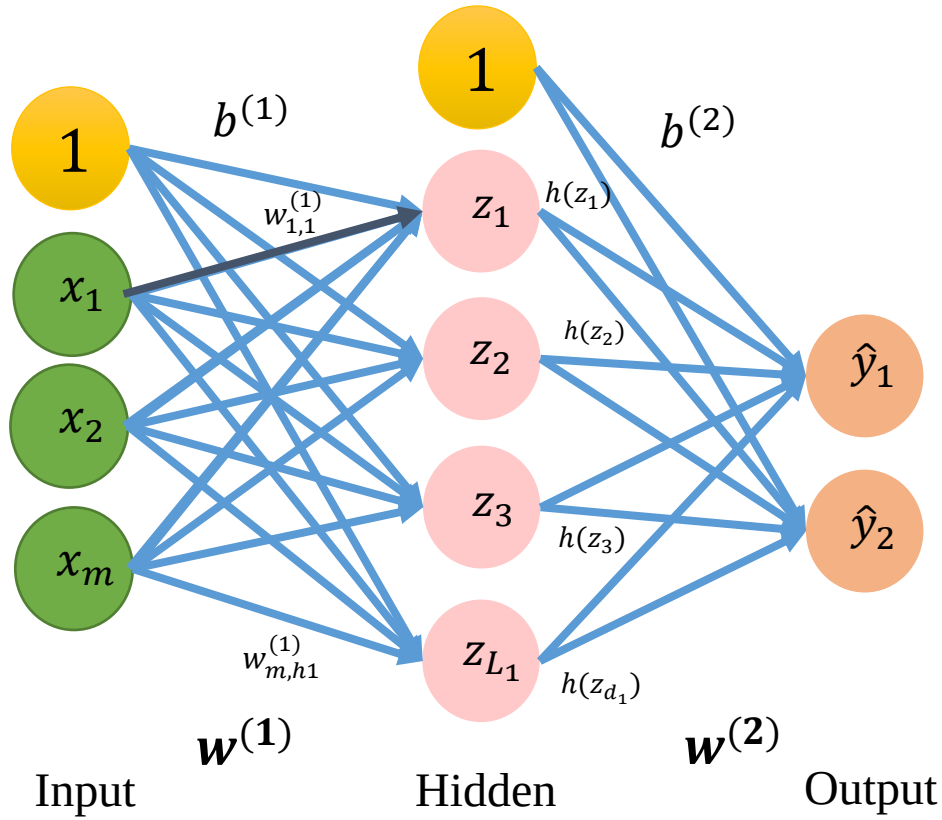
Neural Network training

A typical training procedure for a neural network:

- ✓ We define the neural network with learnable weights (parameters)
- ✓ Feed input through the network and iterate
- ✓ Compute the loss (compare output with ground truth or labels)
- ✓ Propagate gradients back into the network (calculate parameters)
- ✓ Update the weights of the network, e.g.:

$$\text{Weight} = \text{Weight} - \text{learningRate} * \text{Gradient}$$

The Propagation: Single Layer Neural Network



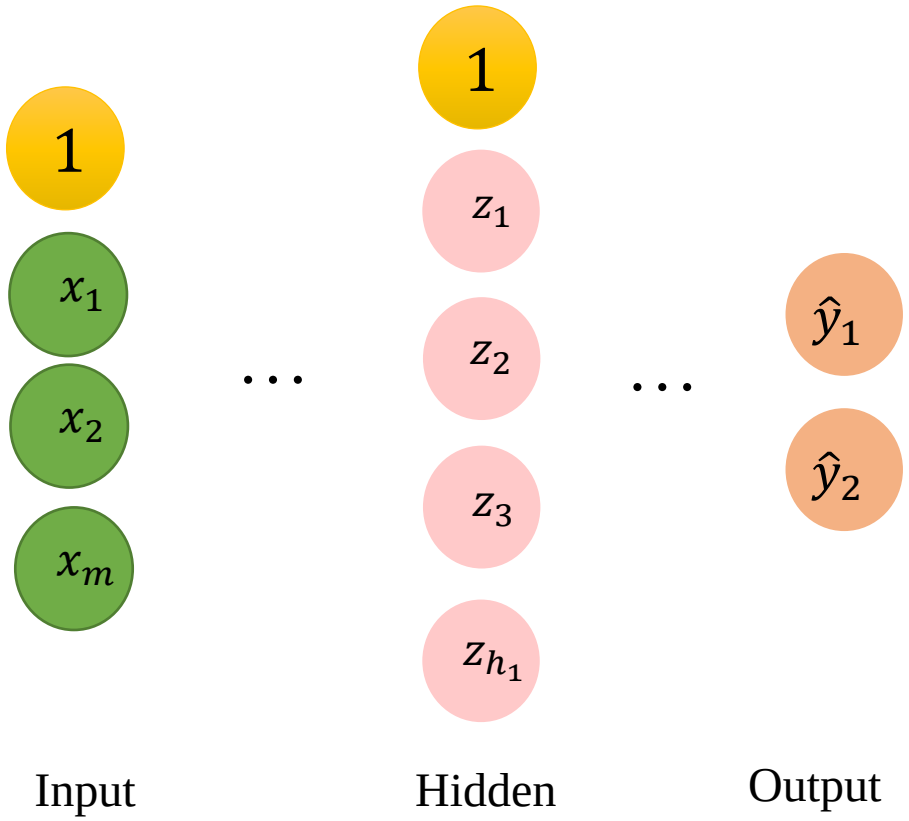
$$z_i = b^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}$$

$$\hat{y}_i = h_2(b^{(2)} + \sum_{j=1}^{L_1} h_1(z_j) w_{j,i}^{(2)})$$

Example:

$$z_1 = b^{(1)} + x_1 w_{1,1}^{(1)} + x_2 w_{2,1}^{(1)} + x_m w_{m,1}^{(1)}$$

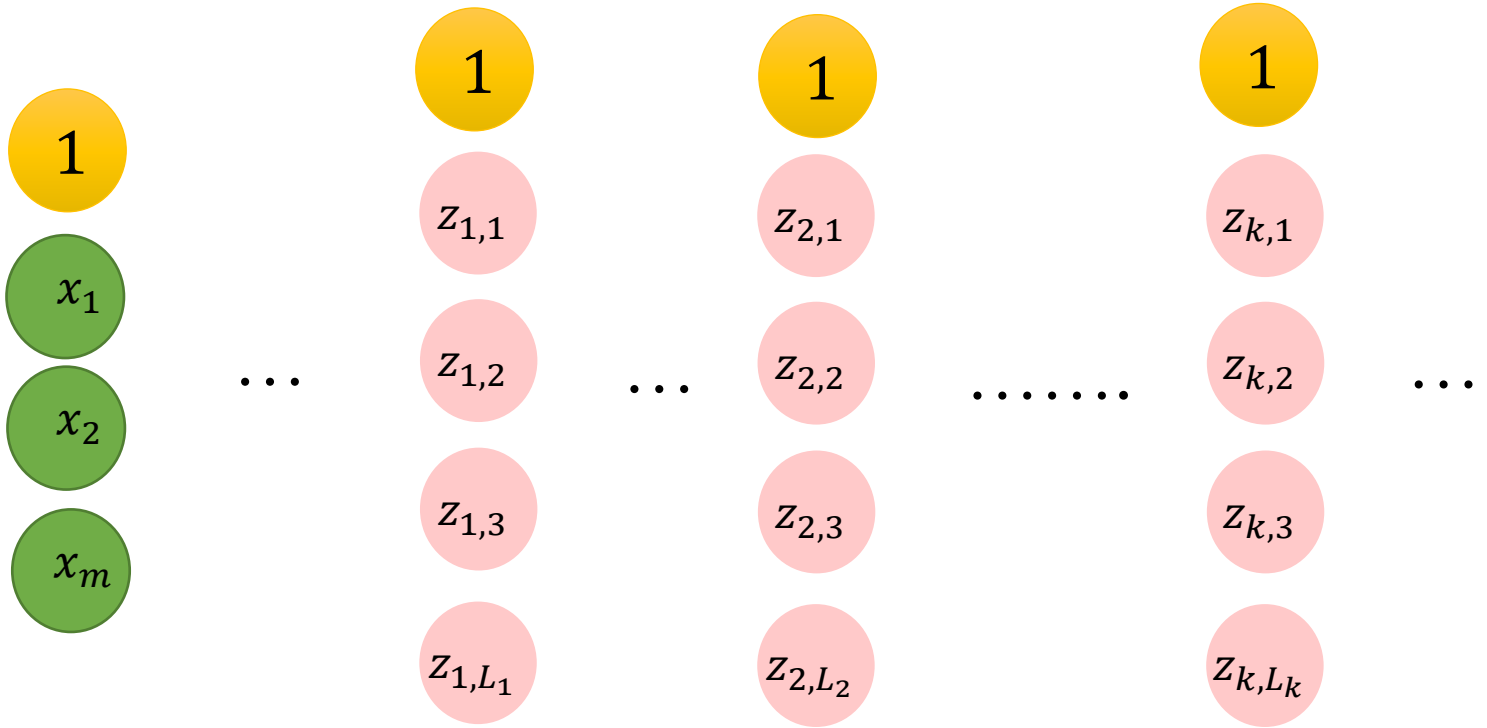
The Propagation: Single Layer Neural Network



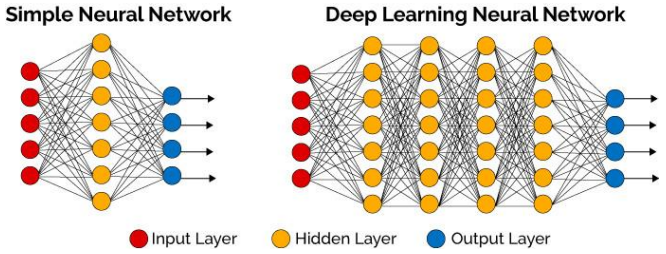

```
import torch.nn as nn
layer1 = nn.Linear(3, 4, bias=True)
layer2 = nn.Linear(4, 2, bias=True)
```

Dense layer

Deep Neural Network




```
import torch.nn as nn
layer1 = nn.Linear(m, h1)
layer2 = nn.Linear(h1, h2)
layer3 = nn.Linear(h2, hk)
layer4 = nn.Linear(hk, 2)
```



$$z_{k,i} = b^{(i)} + \sum_{j=1}^{L_{k-1}} h(z_{k-1}, j) w_{j,i}$$

Example Problem

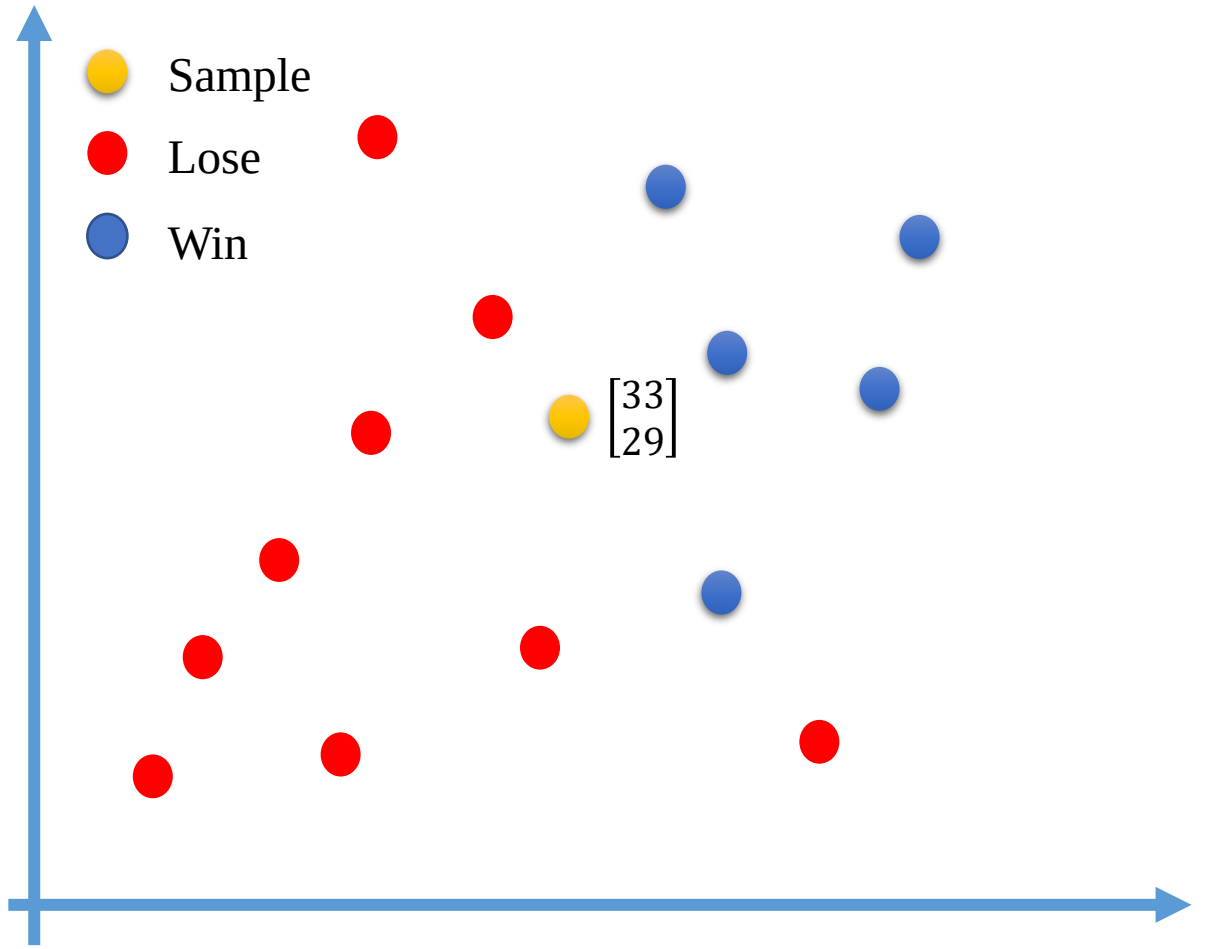
A simple two feature model

Success in swimming competition?

x_1 = Number of hours of the swimming practices

x_2 = Number of hours of the dryland workouts

Example Problem: Wining swimming competition?

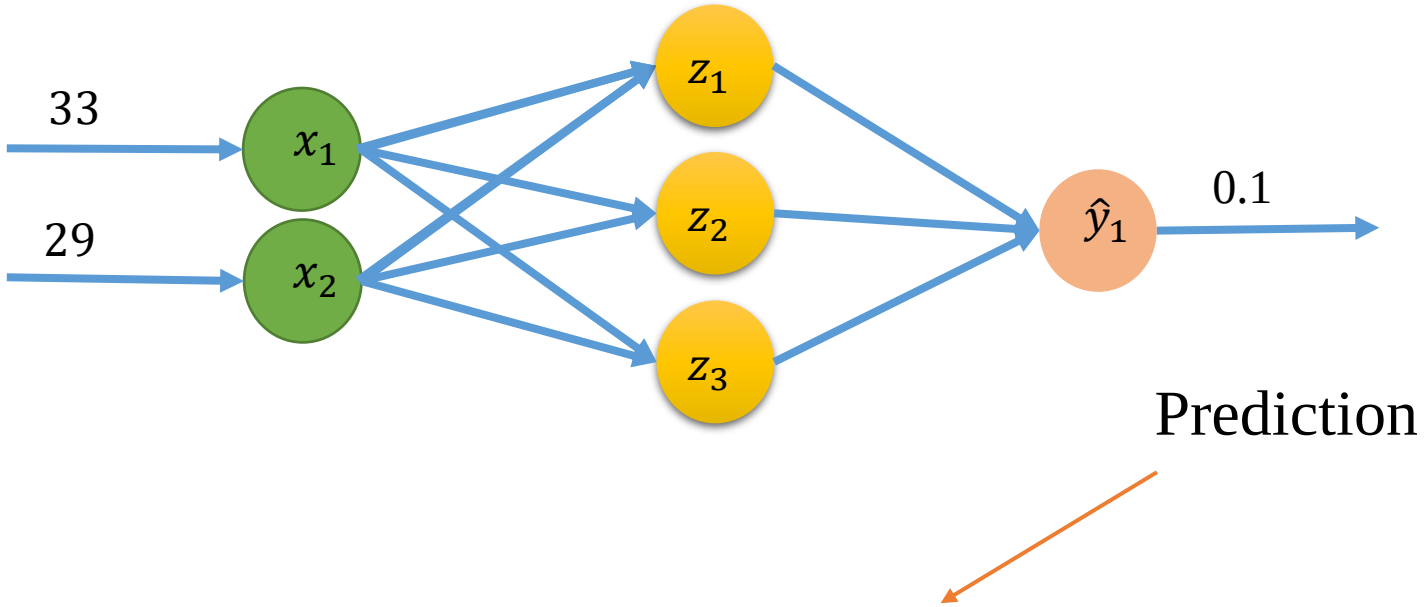


x_2 = Number of hours of the dryland workouts

Example Problem

Untrained Network

$x^{(1)} = [33, 29]$



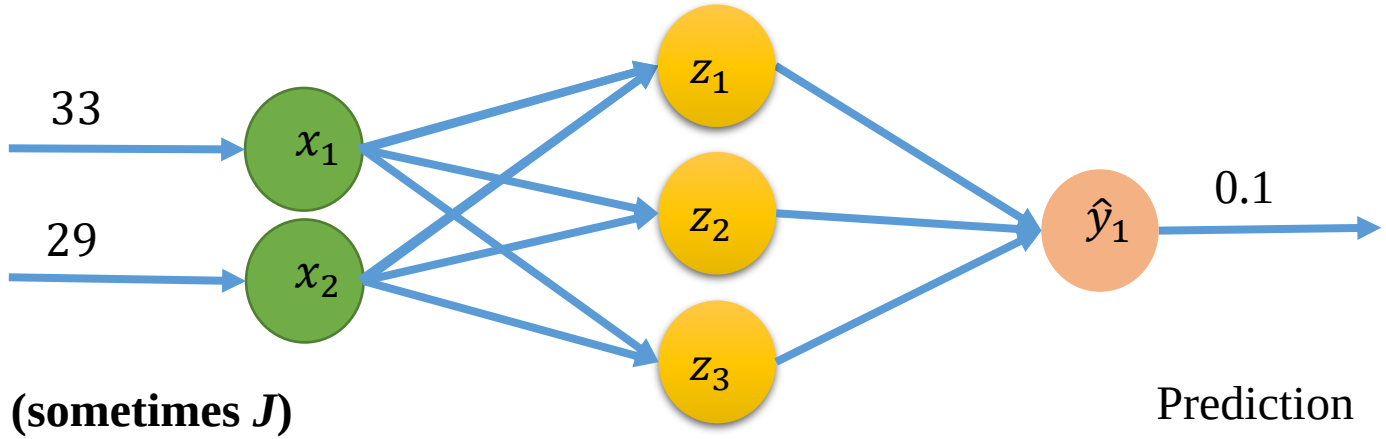
After training we suppose to see 1 as output (pass)

Loss function

The **Loss** or error of the network:

✓ Measures the cost of the incorrect predictions

$x^{(1)} = [33, 29]$



Prediction

Objective or cost function (sometimes *J*)

$$E(W) = \mathcal{L}(f(x^{(i)}, W), y^{(i)})$$

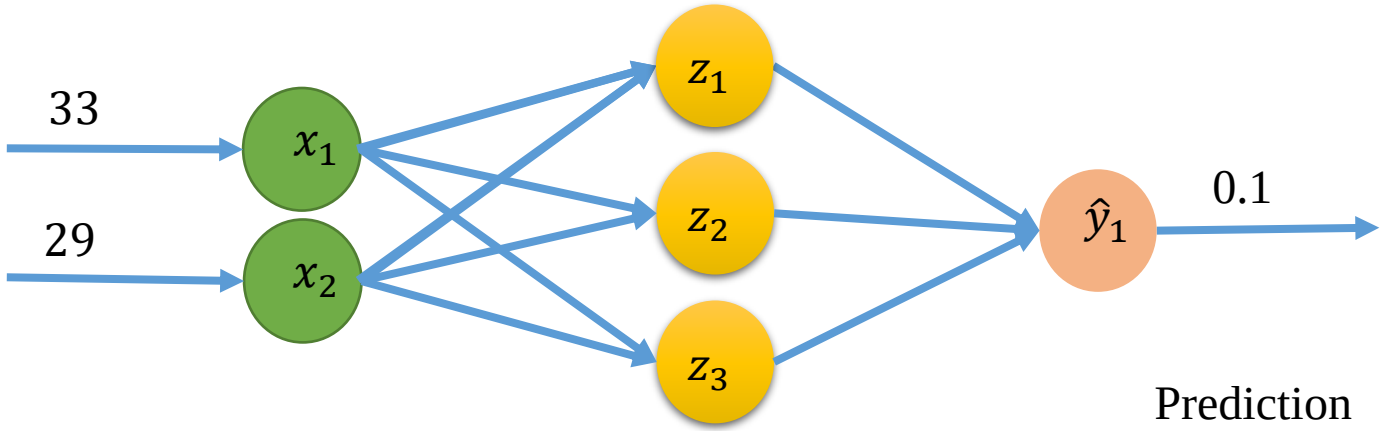
Prediction

Label or rue value or ground truth

Empirical Loss

✓ Calculating the total loss over our entire dataset

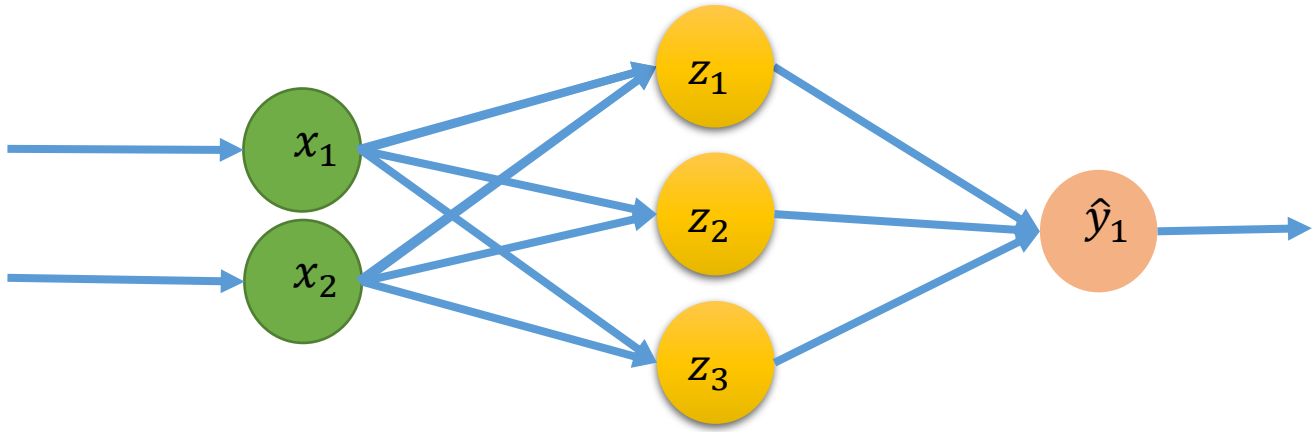
$$x^{(1)} = \begin{bmatrix} 33, 29 \\ 28, 32 \\ 38, 15 \\ \dots \end{bmatrix}$$



$$E(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}, W), y^{(i)})$$

Mean Squared Error (MSE) Loss

$$x^{(1)} = \begin{bmatrix} 33,29 \\ 28,32 \\ 38,15 \\ \dots \end{bmatrix}$$



$f(x)$

$$\begin{bmatrix} 90 \\ 78 \\ 55 \\ \dots \end{bmatrix}$$

y

$$\begin{bmatrix} 95 \\ 80 \\ 72 \\ \dots \end{bmatrix}$$

Prediction

$$E(W) = \frac{1}{n}\sum_{i=1}^{n} \left|\left| f\left(x^{(i)}, W\right) - y^{(i)} \right|\right|^2$$

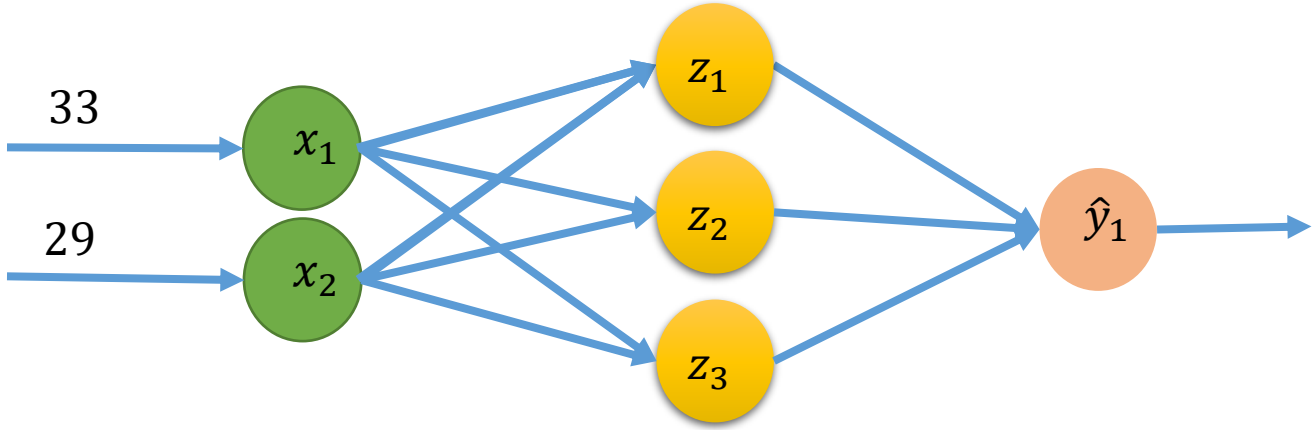
Neural Network function

```
loss = loss = nn.MSELoss()
output = loss(y, prediction)
```

Binary Cross Entropy Loss

- ✓ We can use Cross entropy loss for output a probability between 0 and 1
- ✓ compares each of the predicted probabilities to actual class output which can be either 0 or 1

$$x^{(1)} = \begin{bmatrix} 33,29 \\ 28,32 \\ 38,15 \\ \dots \end{bmatrix}$$



$f(x)$

$$\begin{bmatrix} 0.2 \\ 0.9 \\ 0.5 \\ \dots \end{bmatrix}$$

y

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ \dots \end{bmatrix}$$

Prediction

$$E(W) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log(\, f(x^{(i)}, W)) + (1 - y^{(i)}) \log(\, 1 - f(x^{(i)}, W))$$

label

```
loss = nn.CrossEntropyLoss()
output = loss(y, prediction)
```

Training Neural Networks - Loss Optimization

✓ The objective is to find a network weights with minimum loss

$$W^* = \text{argmin} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

$$\text{where } W = \{w^{(0)}, w^{(1)}, \ldots.\}$$

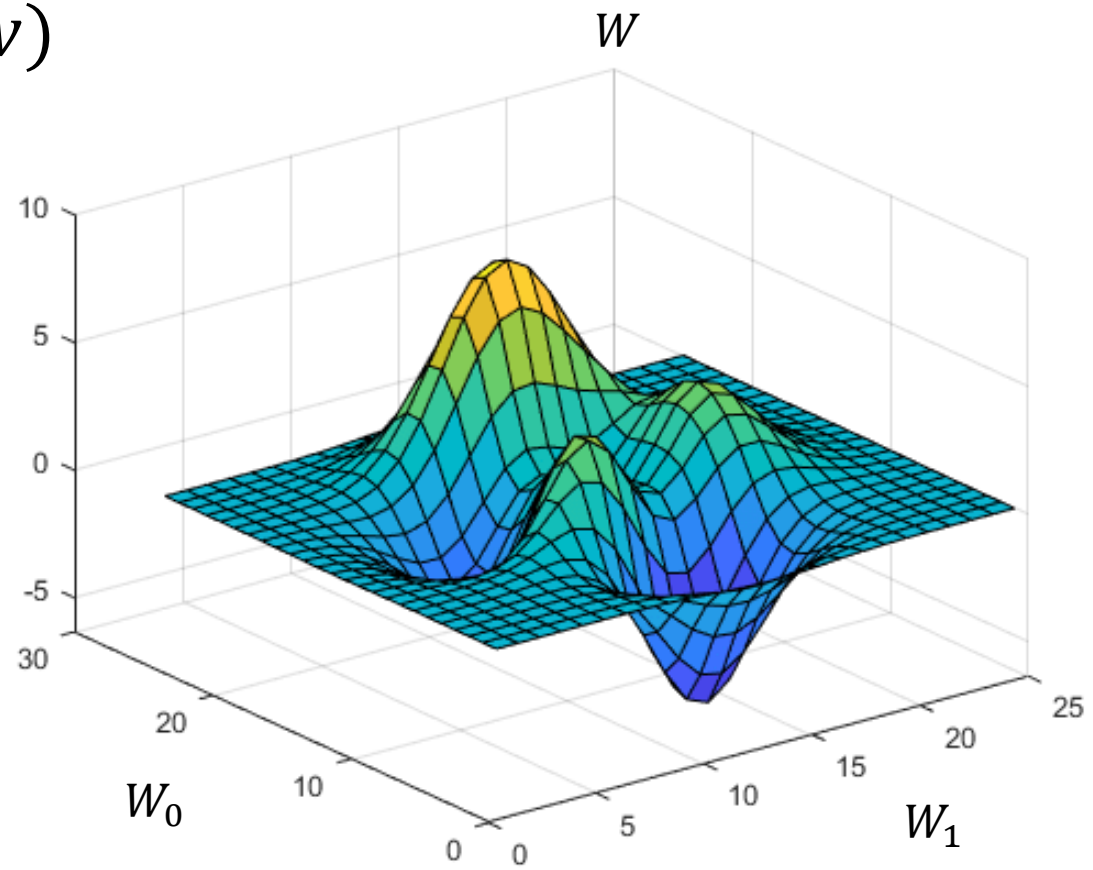
$$W^* = \underset{W}{\text{argmin}}\, E(w)$$

Loss Optimization

✓ The loss function in fact the network's weights

$$W^* = \operatorname{argmin} E(w)$$

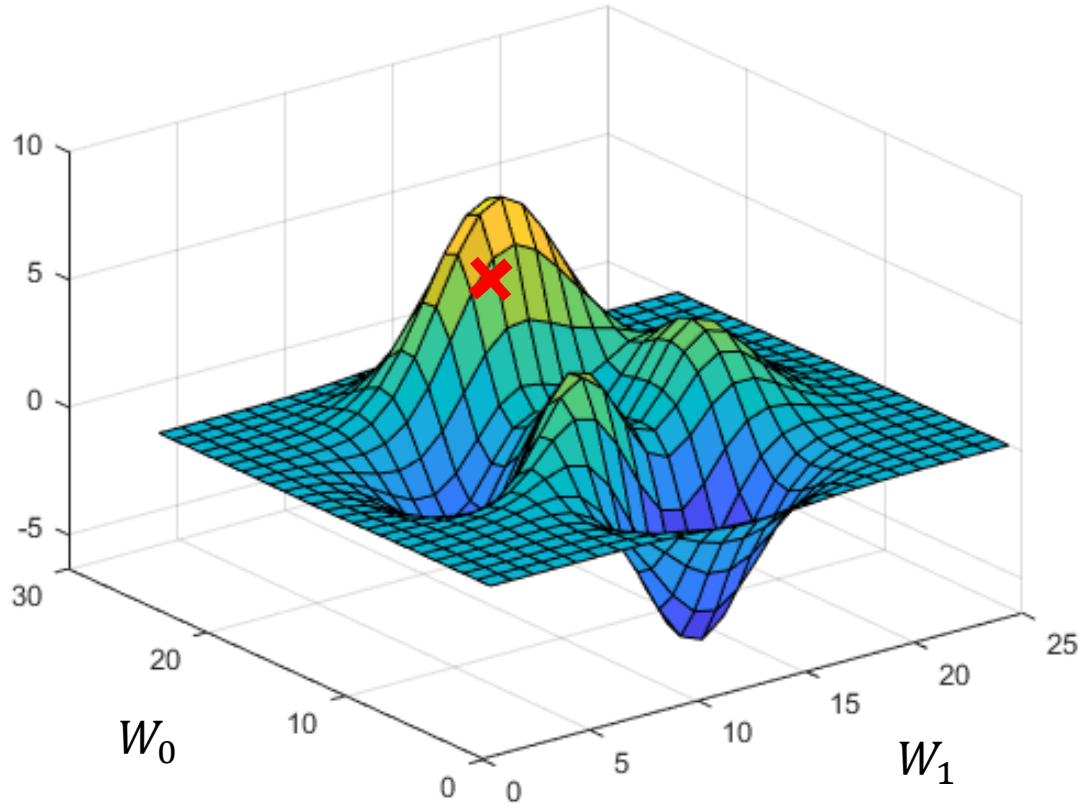
$J(w_0, w_1)$



Loss Optimization

✓ The idea is Randomly pick an initial weigh if we have two (w_0, w_1)

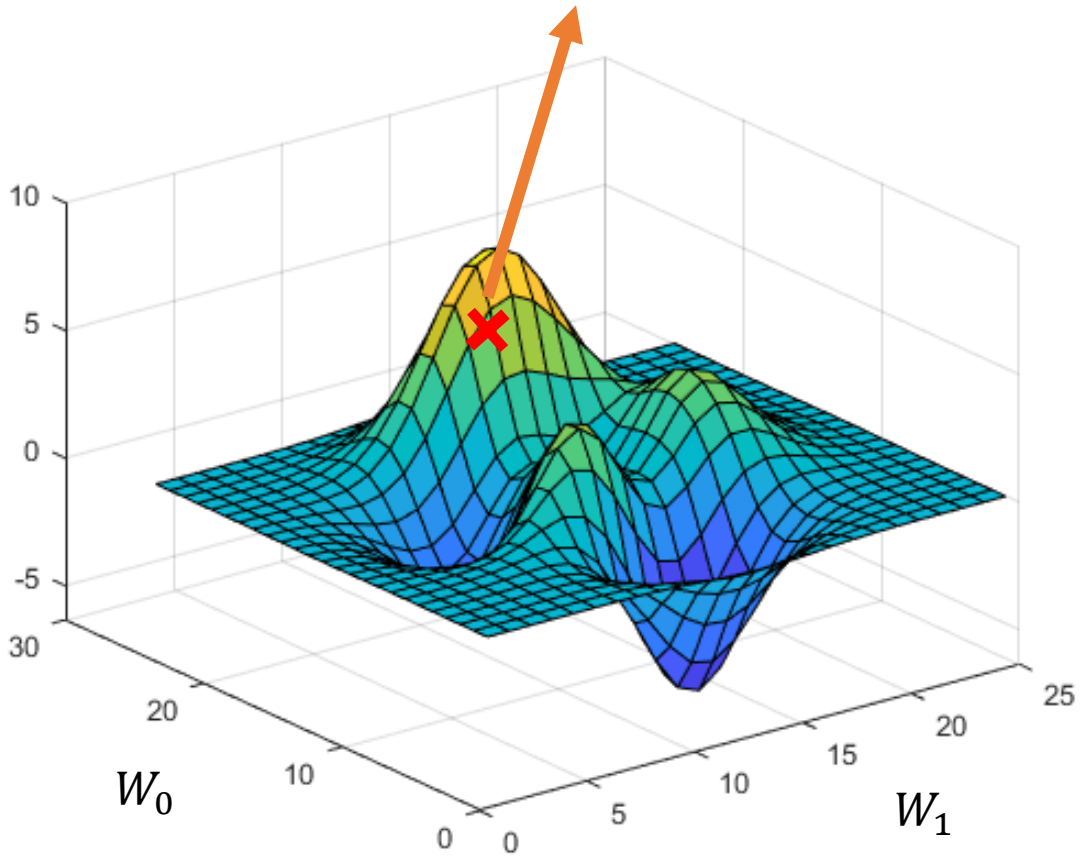
$E(w_0, w_1)$



Loss Optimization

✓ Compute its gradient as: $\frac{\partial E(W)}{\partial W}$

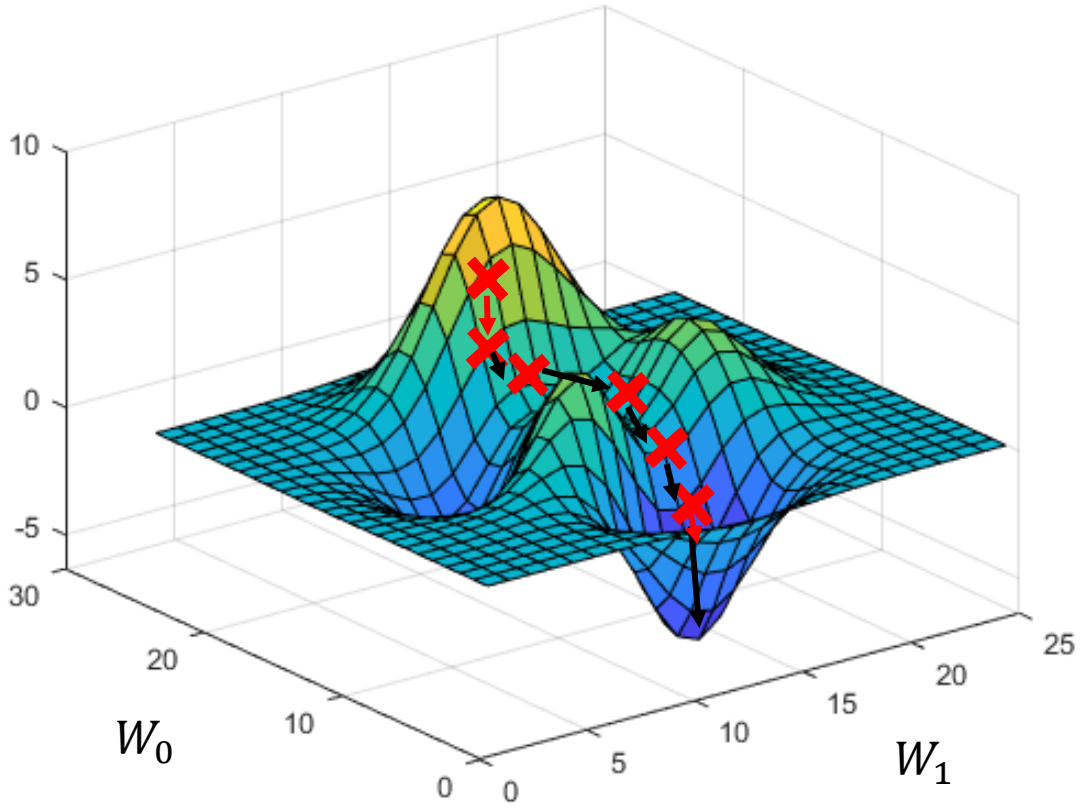
$E(w_0, w_1)$



Loss Optimization

- ✓ In opposite direction of gradient take steps based on our defined step size and repeat until convergence.

$E(w_0, w_1)$



Gradient Descent

Gradient Descent Algorithm

Initialize weights (random)

Loop until convergence:

Compute gradient, $\frac{\partial E(W)}{\partial W}$

learning rate

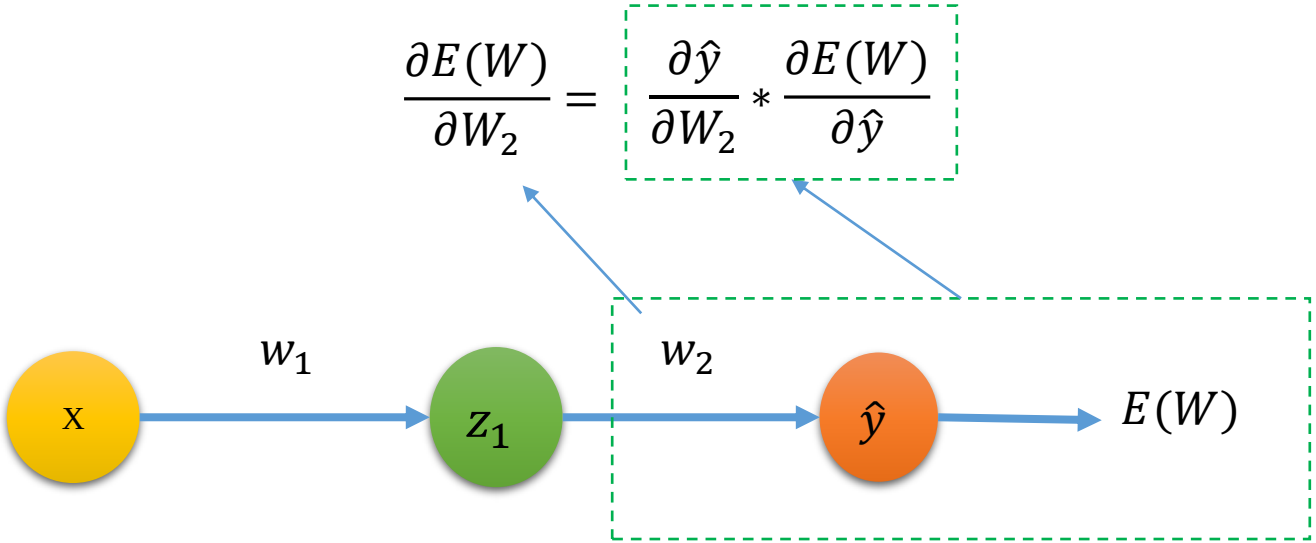
Update weights, $W \longleftarrow W - \eta \frac{\partial E(W)}{\partial W}$

Return weights

```
optim = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.92)
optim.step() #gradient descent
```

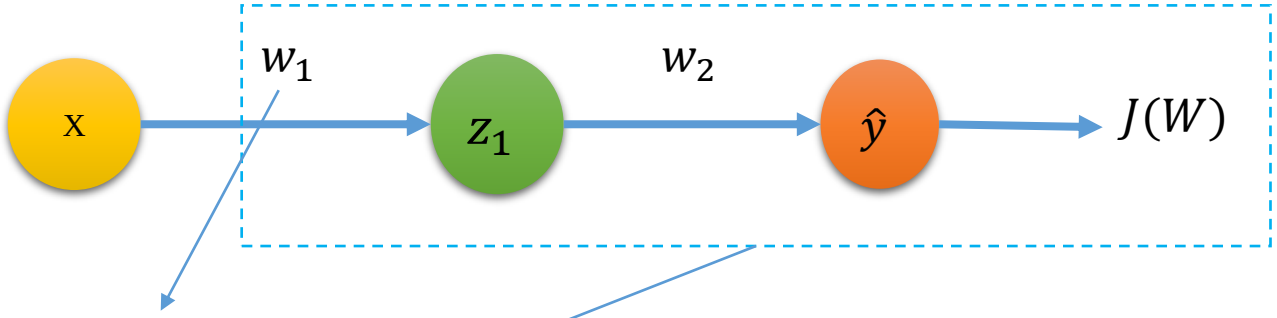
Computing Gradients: Backpropagation

✓ Backpropagation is changing in one weight will update the final loss $E(W)$



Computing Gradients: Backpropagation

$$\frac{\partial E(W)}{\partial W_2} = \frac{\partial \hat{y}}{\partial W_2} * \frac{\partial E(W)}{\partial \hat{y}}$$



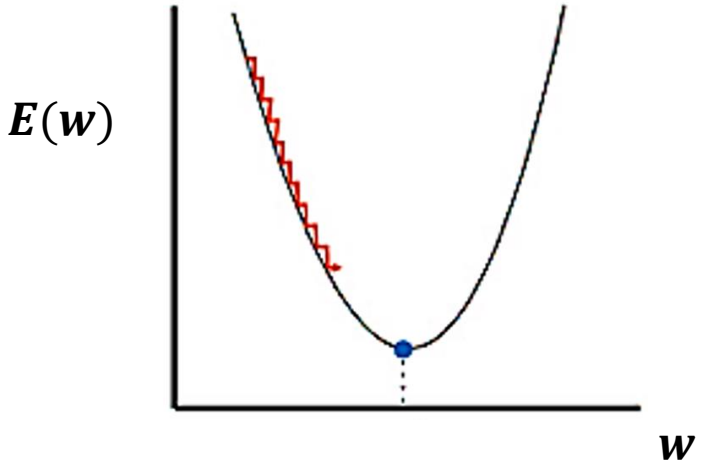
$$\frac{\partial E(W)}{\partial w_1} = \frac{\partial z_1}{\partial w_1} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial E(W)}{\partial \hat{y}}$$

Note: In Pytorch the implementation of back-propagation and gradient update is automatic

Learning Rate Challenges

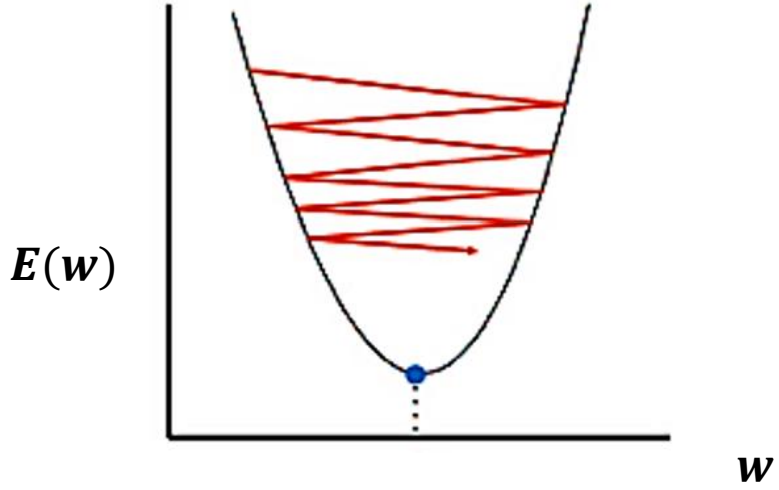
The Small learning rate:

Converges slowly and can stuck in local minima



Large Learning Rate:

overshoot, become unstable and diverge



Stable learning rates: can avoid local minima and converges smoothly

Learning Rate

How to solve learning rate challenge?

- ✓ Try different learning rates and choose best
- ✓ Use an **adaptive learning** rate that is able to adopt

Learning Rate

Adaptive Learning Rates

- ✓ Adaptive learning rate methods are an optimization of gradient descent methods
- ✓ Goal of minimizing the objective function
- ✓ Learning rates can be dynamic depending on:
  - ➢ Gradient size
  - ➢ Momentum of the learning
  - ➢ Particular weights size
  - ➢ etc …

Different Gradient Descent Algorithms

Some example famous Algorithms

SGD

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

Adam

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
```

Adagrad

```
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.1)
```

RMSProp

```
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.1, momentum=0.8)
```

A method useful to adjust for to faster converging

Gradient Descent

Algorithm

Initialize weights (random)

Loop until convergence:

Compute gradient $\frac{\partial E(W)}{\partial W}$

computationally expensive (calculate for all points)

Update weights, $W \longleftarrow W - \eta \frac{\partial E(W)}{\partial W}$

Return weights

Stochastic Gradient Descent (SGD)

Algorithm

Initialize weights (random)

Loop until convergence:

Pick single data Point i

Compute gradient $\frac{\partial E_i(W)}{\partial W}$

Noisy because only one sample

Update weights, $W \longleftarrow W - \eta \frac{\partial E(W)}{\partial W}$

Return weights

Stochastic Gradient Descent (SGD) - Mini-batches

Algorithm

Initialize weights (random)

Loop until convergence:

Pick N Mini-batches of data points

Compute gradient $\frac{\partial E(W)}{\partial W} = \frac{1}{N}\sum_{k=1}^{N}\frac{\partial E_k(W)}{\partial W}$

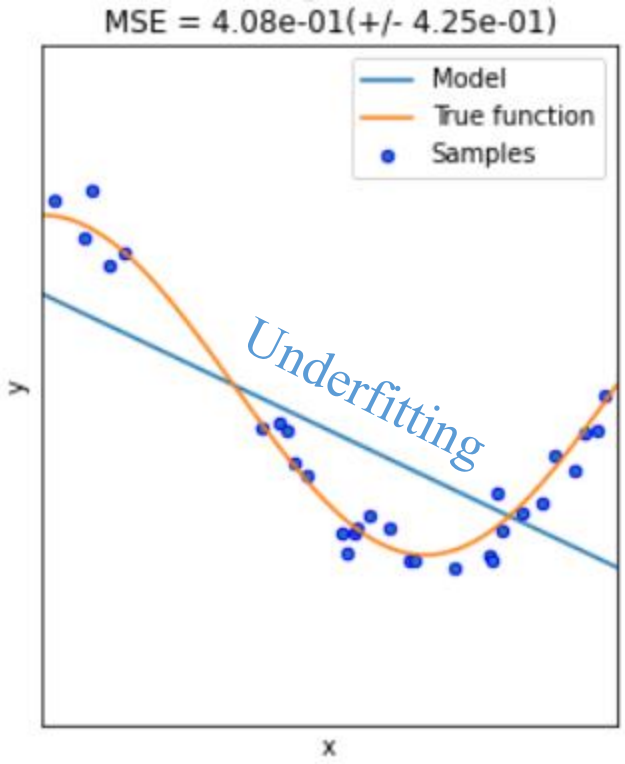
Update weights, $W \longleftarrow W - \eta \frac{\partial E(W)}{\partial W}$

Return weights

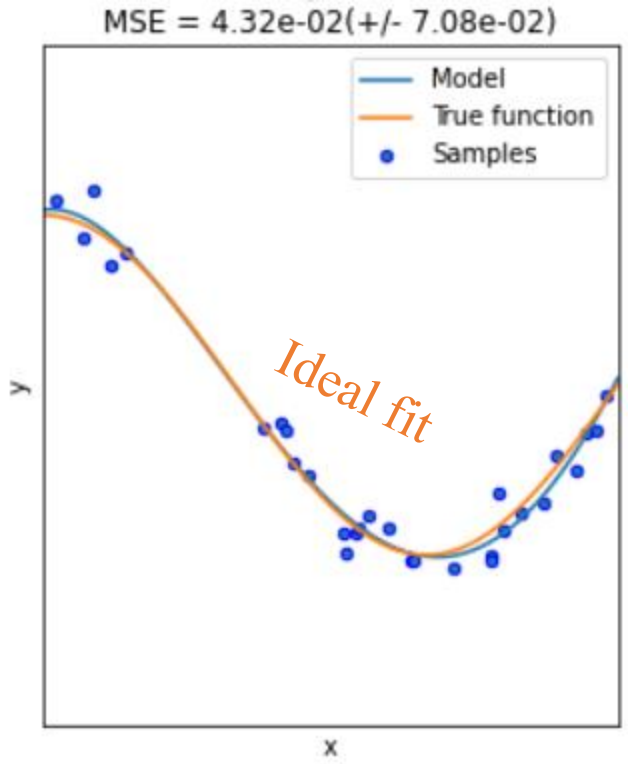
Computationally less expensive

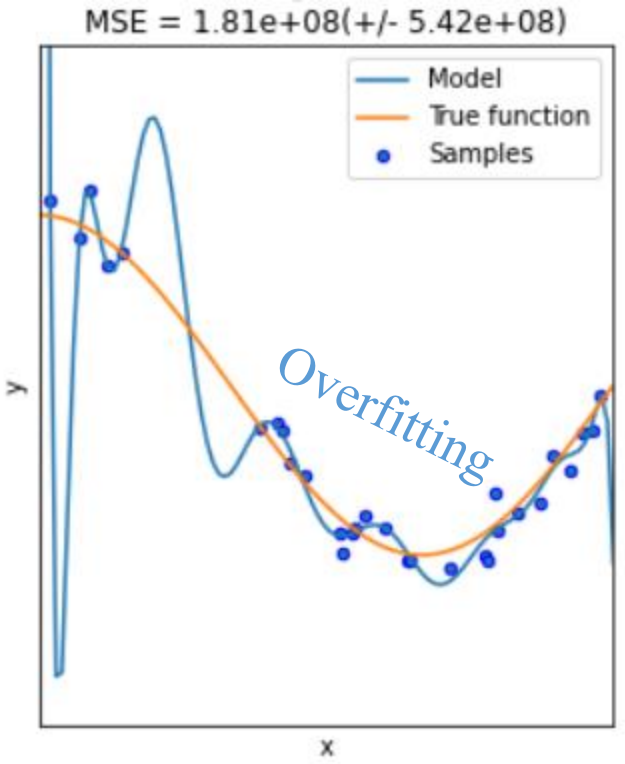
- ✓ Smooth convergence
- ✓ Higher accuracy for estimating Gradient
- ✓ We can increase learning rate
- ✓ Parallelized computation

Overfitting problem



Model cannot fully learn the data



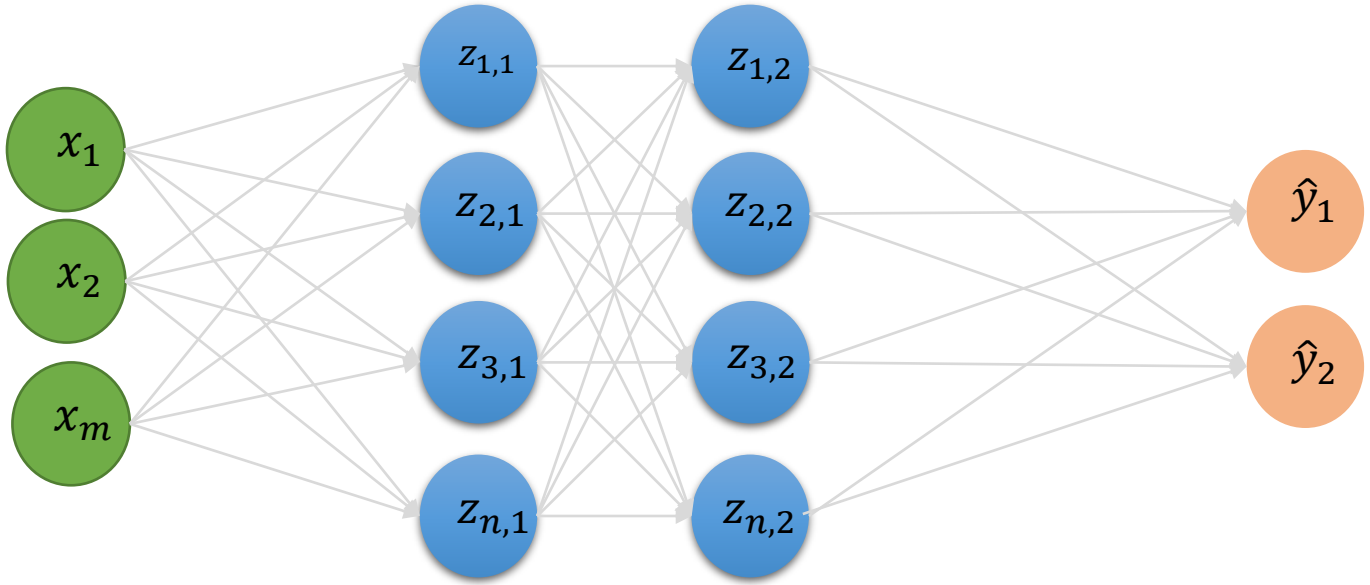


Fit too much so it is not generalized

Overfitting

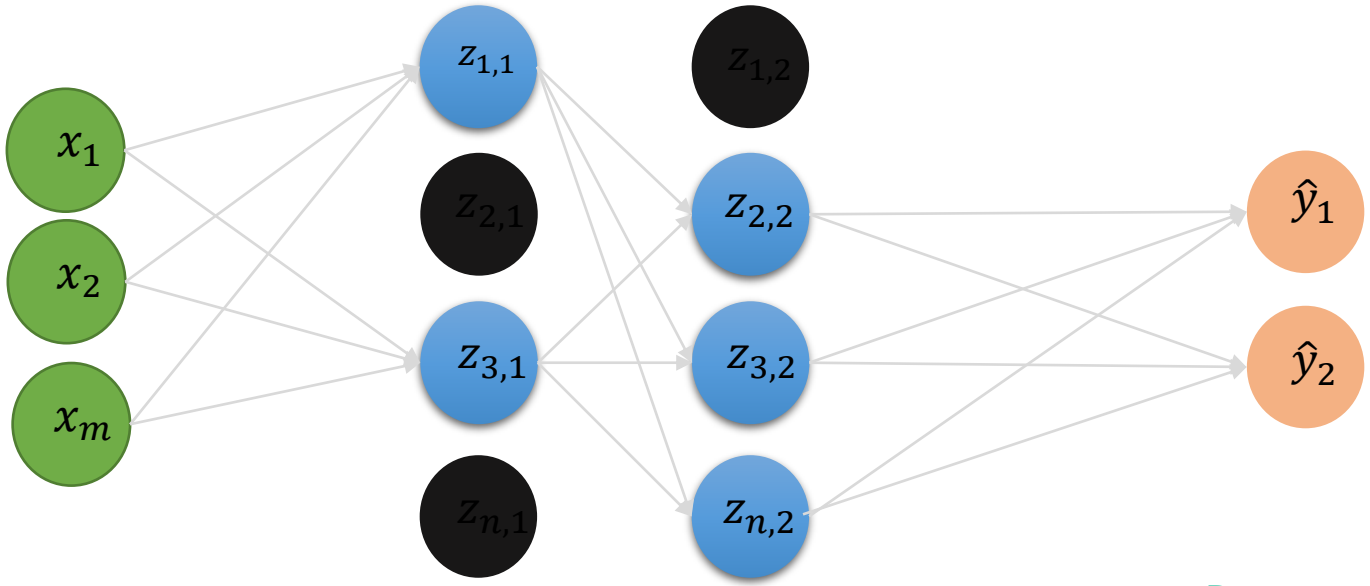
Regularization

- ✓ A technique to prevent overfitting
- ✓ Improve generalization of the model on unseen data
- ✓ Force the network to not relay on any node



Regularization approach: Dropout

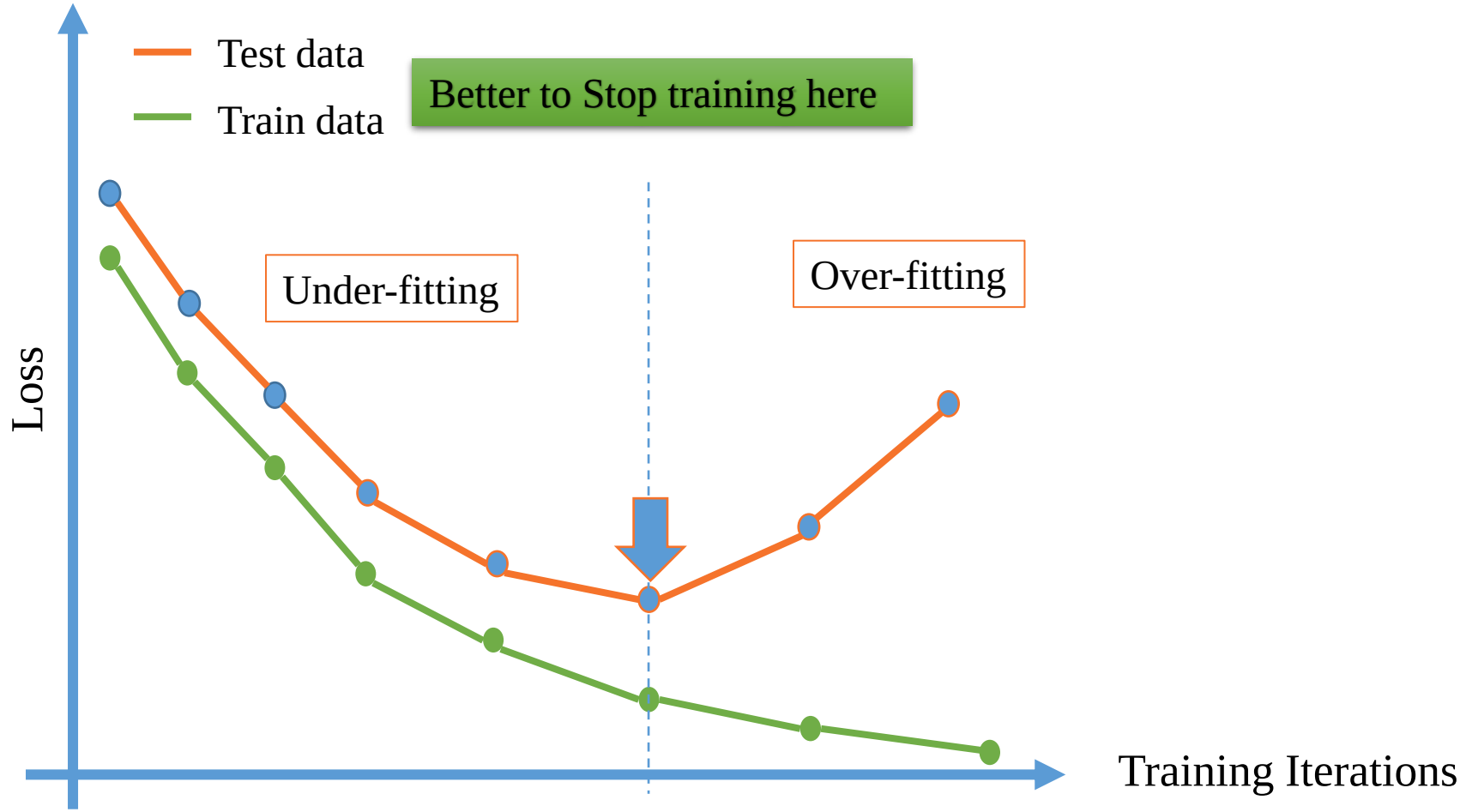
✓ Randomly set some of the activations to 0 every time (e.g. 30%)




```
nn.Dropout(p=0.3)
```

Regularization approach: Early Stopping

✓ We can Stop training before we have a chance to overfit



Different flexibility of training model

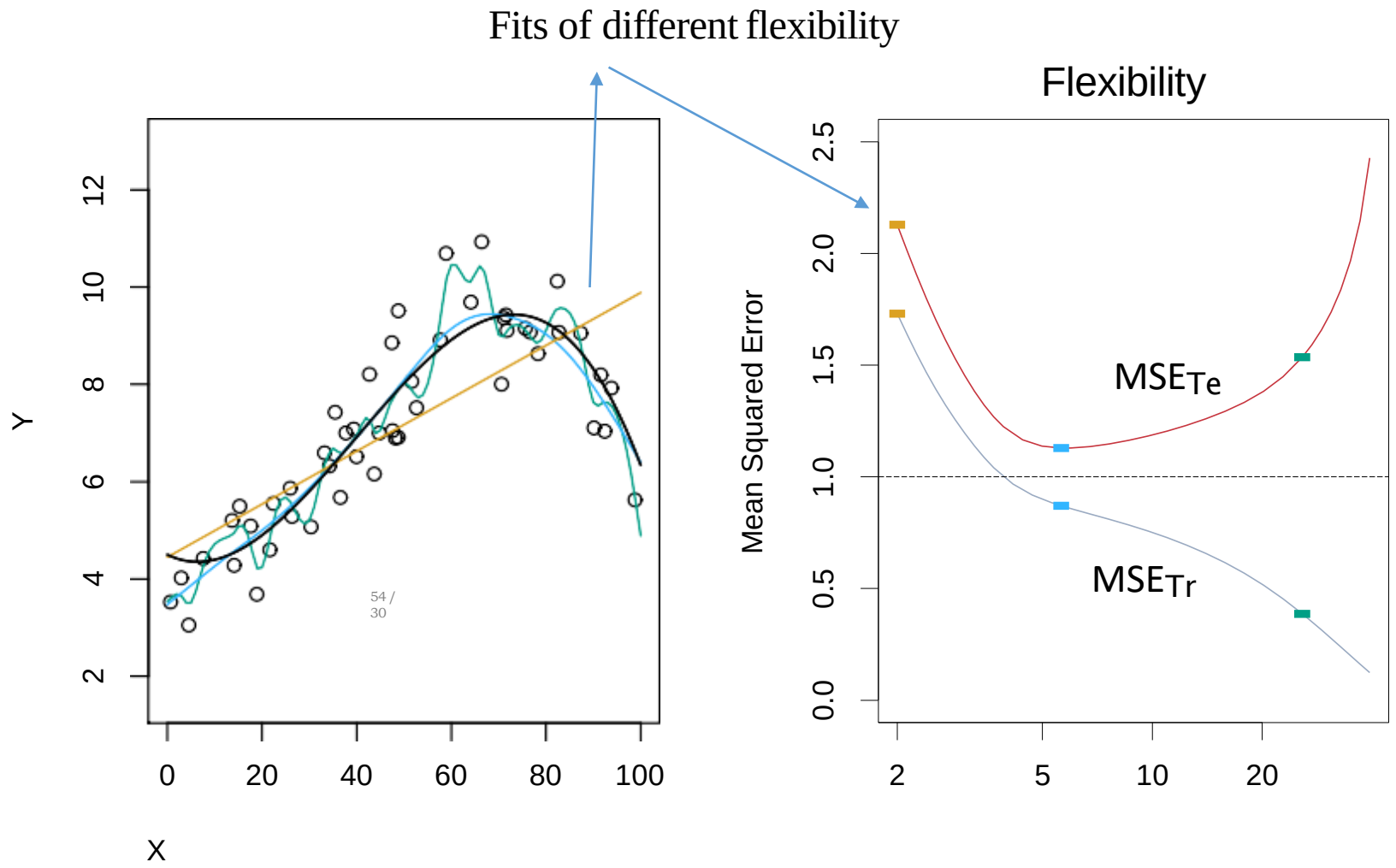
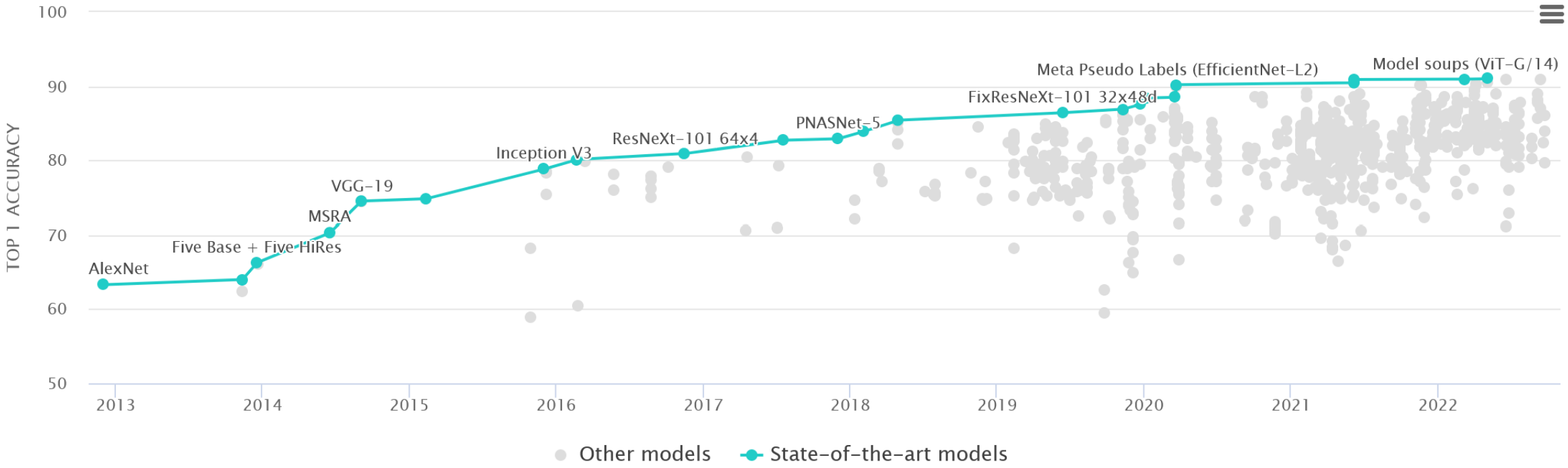


Image Classification on ImageNet (Deep learning)

ImageNet

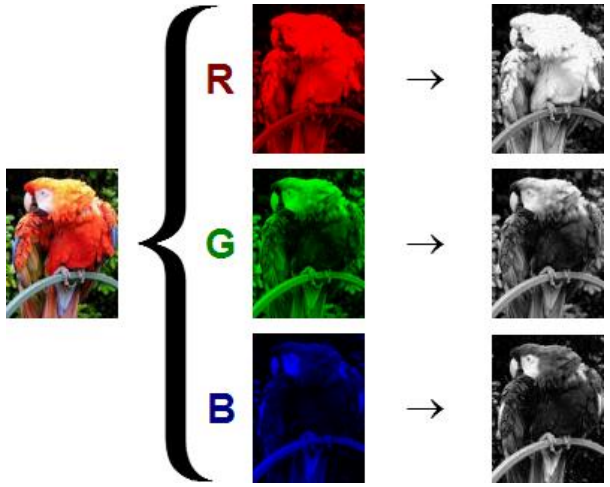
- ✓ It is large visual database designed for use in visual object recognition
- ✓ More than 14 million images have been hand-annotated
- ✓ One million of the images includes bounding boxes



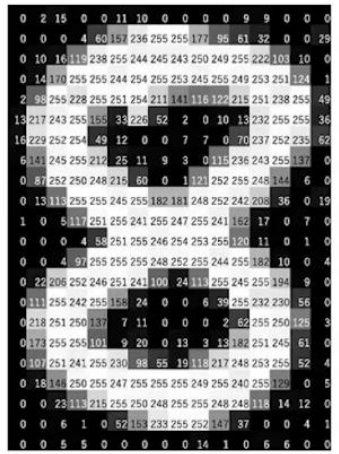
https://paperswithcode.com/sota/image-classification-on-imagenet

Images and visual information

- ✓ In visual data feature extraction is very important (indicates state of the environment)
- ✓ Visual information can be represented as arrays of numbers
- ✓ For example images can be shown as matrix of numbers in range of [0,255]:



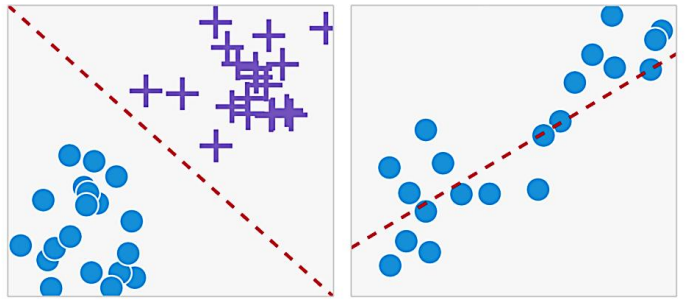
$400 \times 800 \times 3 \times 255$



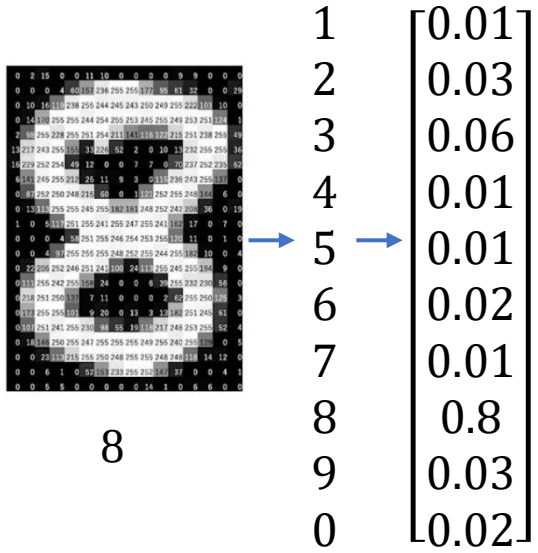
$16 \times 22 \times 1 \times 255$

Computer vision tasks

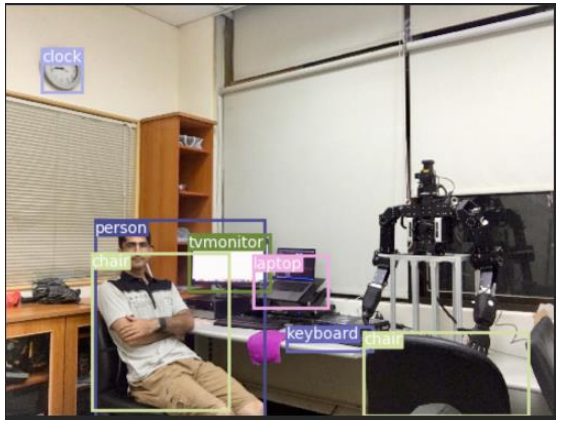
- ✓ Classification task (labeling)
- ✓ Regression Task (model learns to predict numeric scores based on previous history – continues value)
- ✓ Localization task



Classification Regression



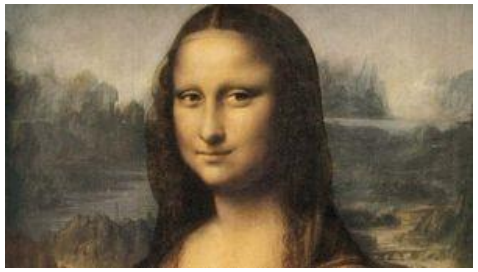
8

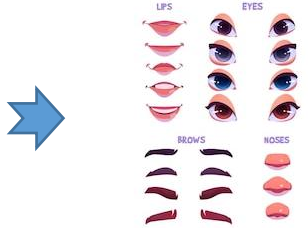


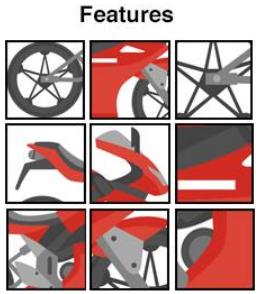
Localize and classify

High level feature extraction

✓ Identifying key feature in images can be very useful to do the detection







Wheels
Doors
Window frame
Headlight

High level feature extraction

Manual feature detection:

- ✓ Needs knowledge of each domain
- ✓ Feature definition for each domain
- ✓ Feature detection and classification

viewpoint

Main Problem

- ✓ Needs a lot of effort
- ✓ Image scale, deformation viewpoint, noise, illumination etc. can make problem very hard.

Illumination

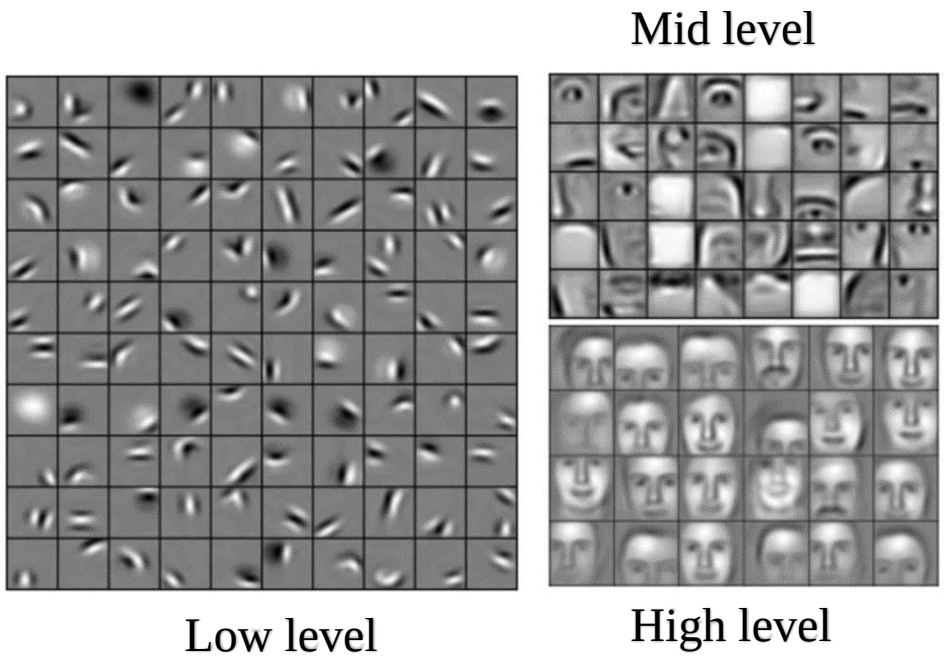
Deformation

High level feature extraction

Learning feature detection:

✓ The idea is to learn features in hierarchy form from low level o high level

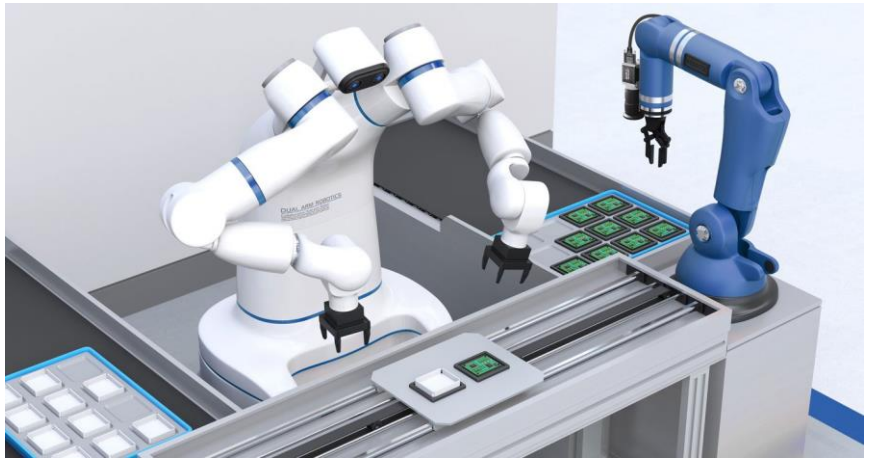
✓ We are not limited to images



High level feature extraction

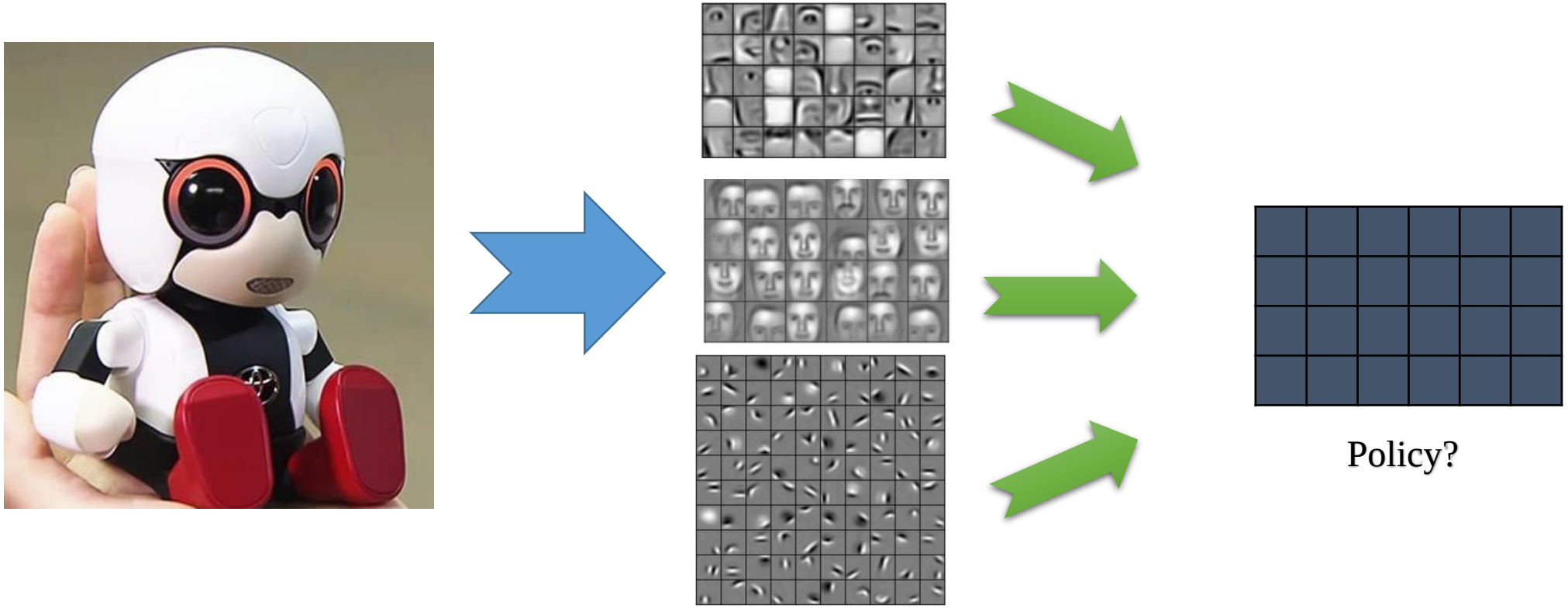
Why feature extraction is important for reinforcement learning?

- ✓ Feature extractions in large state-space can be helpful to detect environment state and agent can learn policy
- ✓ Each state can be a frame of an image (or visual data from environment)



High level feature extraction

Why feature extraction is important for reinforcement learning?



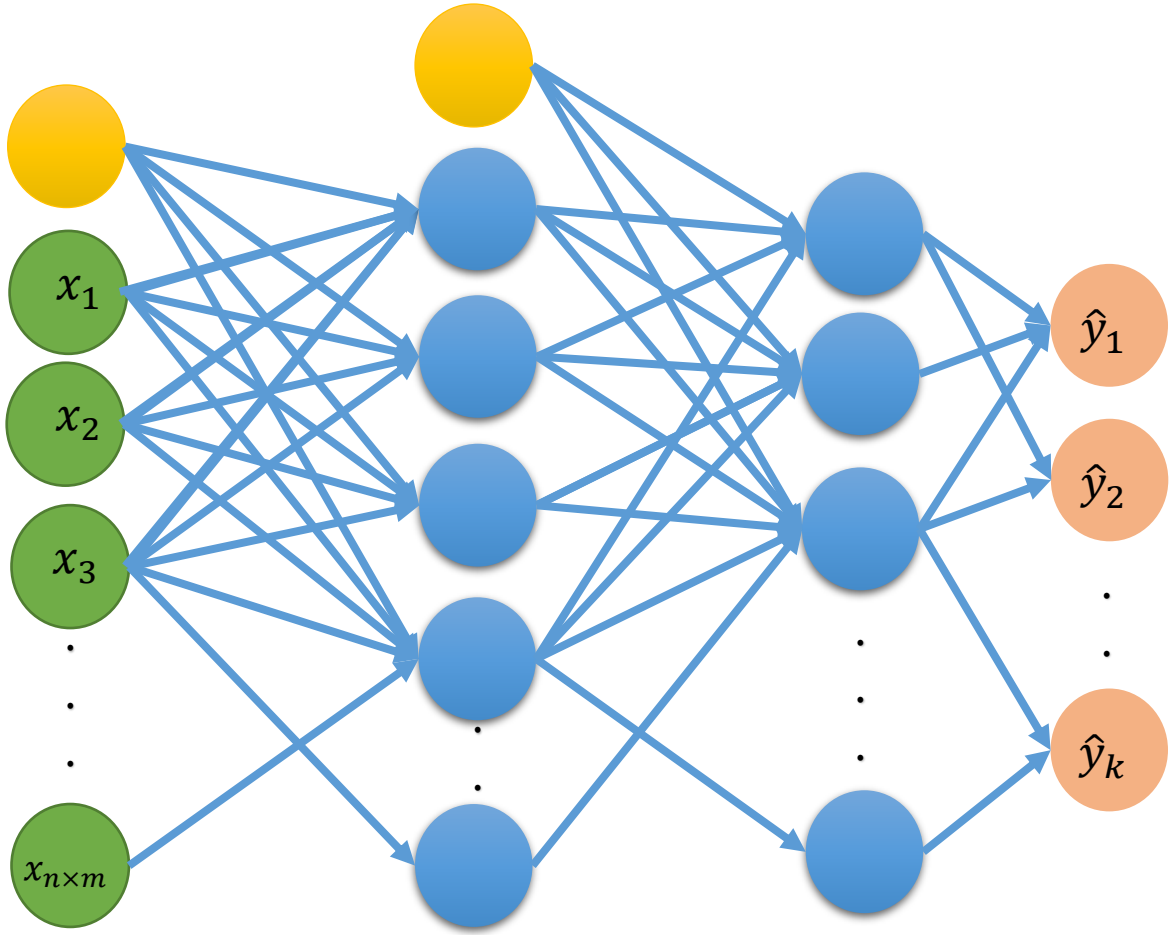
Which level to use as input?

Fully Connected Deep Neural Network

n

m

Vector of flatten pictures can be as input $x_{n\times m}$

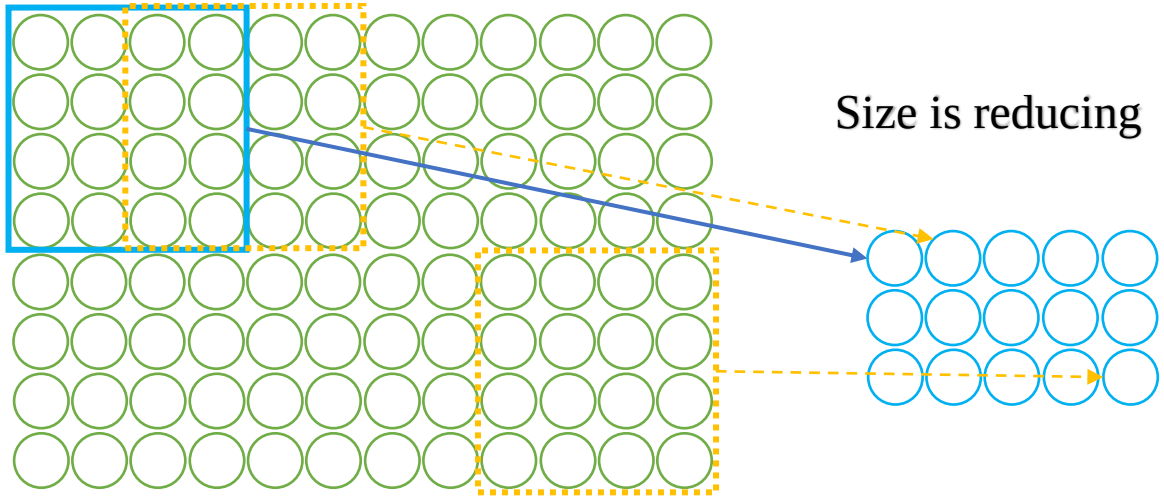


Problem?

We don't use any special structure to use important features only

Sliding Window - Convolutional layers

- ✓ We can connect path of input at each layer to single neuron
- ✓ This is called sliding window

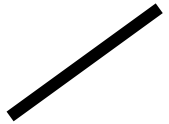


How to weight the patch to new neuron?

Convolutional layers - Filters

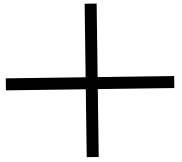
How to weight the patch to new neuron?

- ✓ We can use **different filters** to weigh and extract **local features in** image
- ✓ Filters need to share parameters for cover important complexity

Diagonal $\begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \end{bmatrix}$ 

Diagonal $\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$ 

Multiply $\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix}$ 

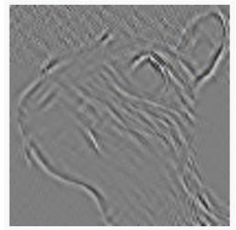
Sum $\begin{bmatrix} -1 & 1 & -1 \\ 1 & 1 & 1 \\ -1 & 1 & -1 \end{bmatrix}$ 

Edge $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & 1 & -1 \end{bmatrix}$ 

Identity $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ 

Blurring $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ 

Sharpen $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ 

Ridge detection $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ 

Convolutional layers – Filters example

✓ Filters can be used to detect features

Sum $\begin{bmatrix} -1 & 1 & -1 \\ 1 & 1 & 1 \\ -1 & 1 & -1 \end{bmatrix}$ +

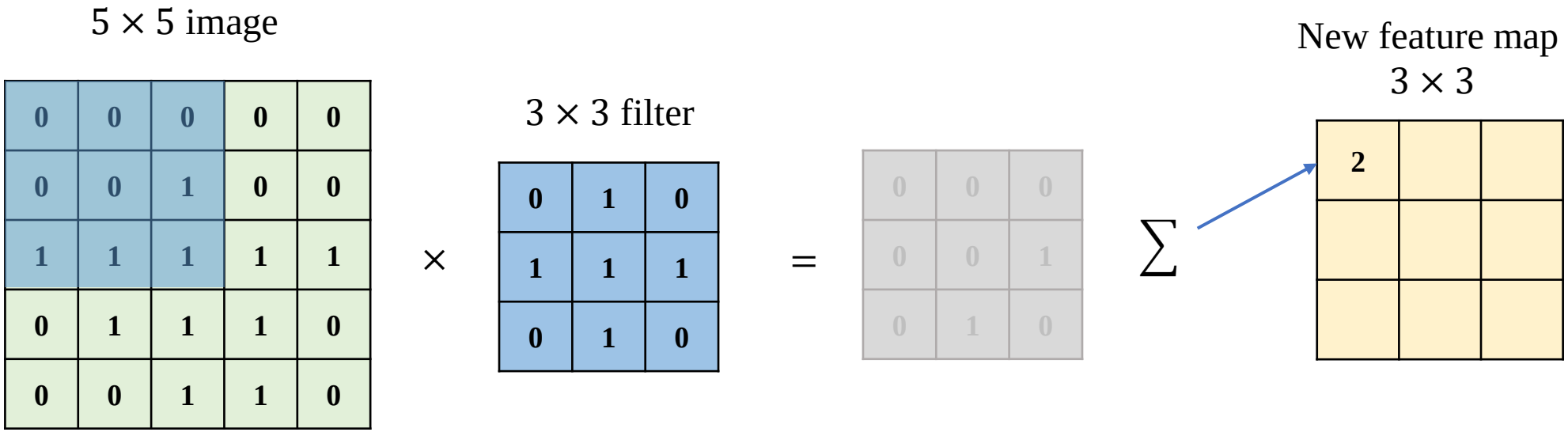
1	1	1	1	1	1	1	1	1
1	1	-1	1	1	1	-1	1	1
1	1	-1	1	1	1	-1	1	1
1	1	-1	1	1	1	-1	1	1
1	1	-1	-1	-1	-1	-1	1	1
1	1	-1	1	1	1	-1	1	1
1	1	-1	1	1	1	-1	1	1
1	1	-1	1	1	1	-1	1	1
1	1	1	1	1	1	1	1	1

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 1 & -1 \\ 1 & 1 & 1 \\ -1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & -1 & -1 \\ 1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix} = -7$$

Element wise multiplication **Sum**

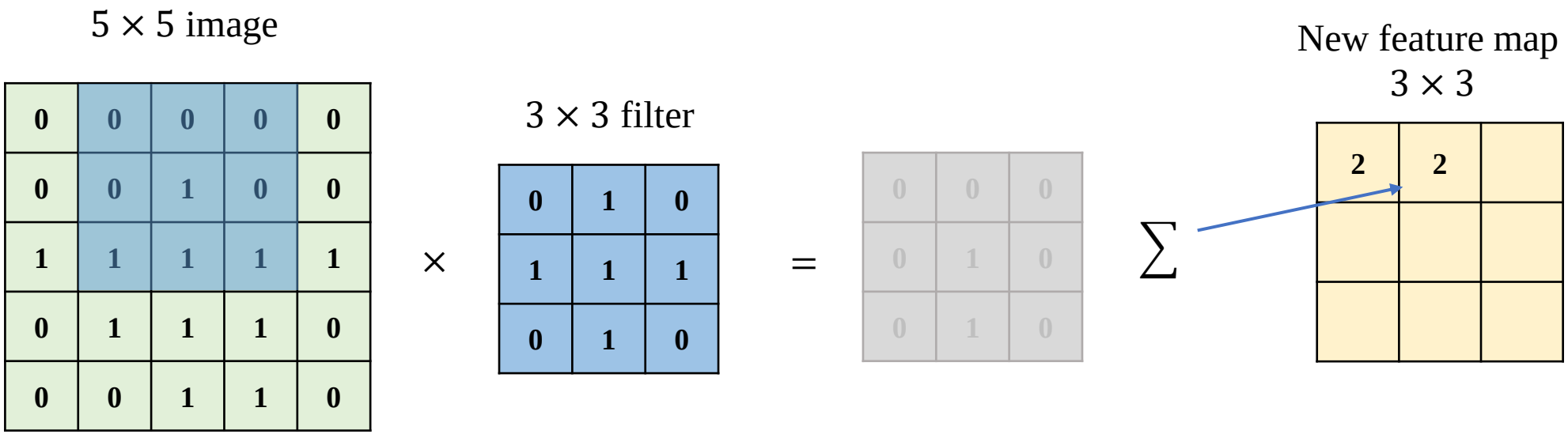
Convolutional layers – Filters example

✓ If we have an image size 5×5 and if we run convolution operation with 3×3 filter



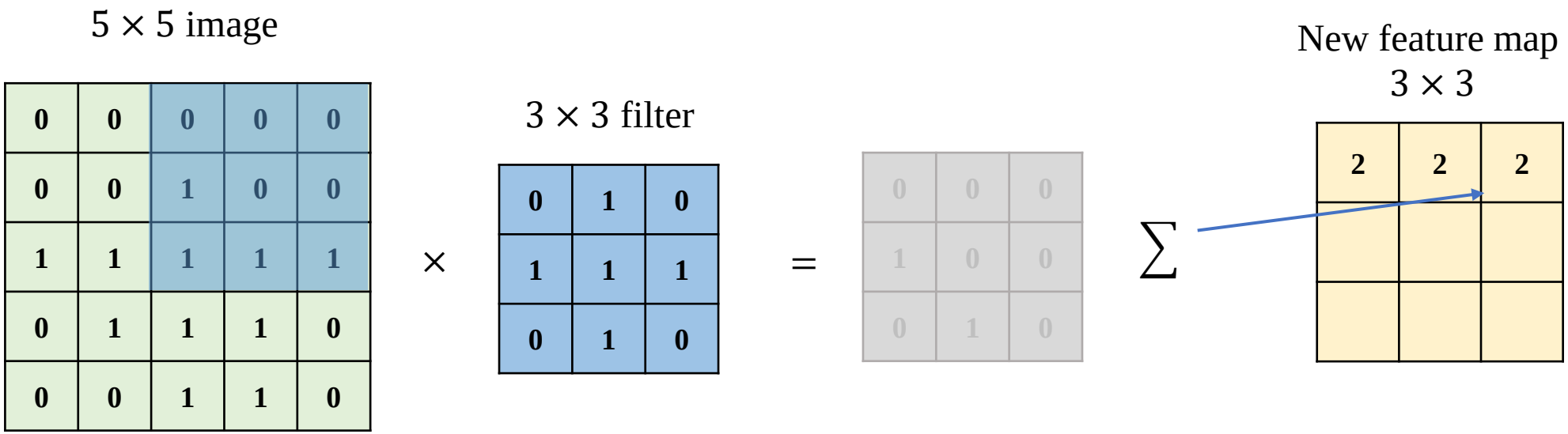
Convolutional layers – Filters example

✓ Slide with stride 1



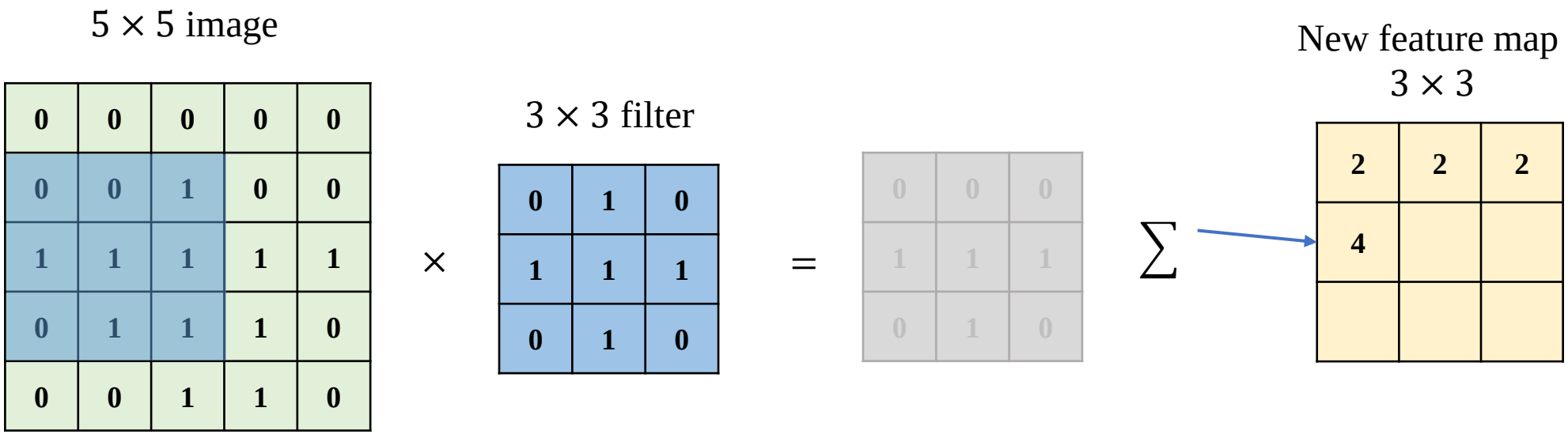
Convolutional layers – Filters example

✓ Slide with stride 1



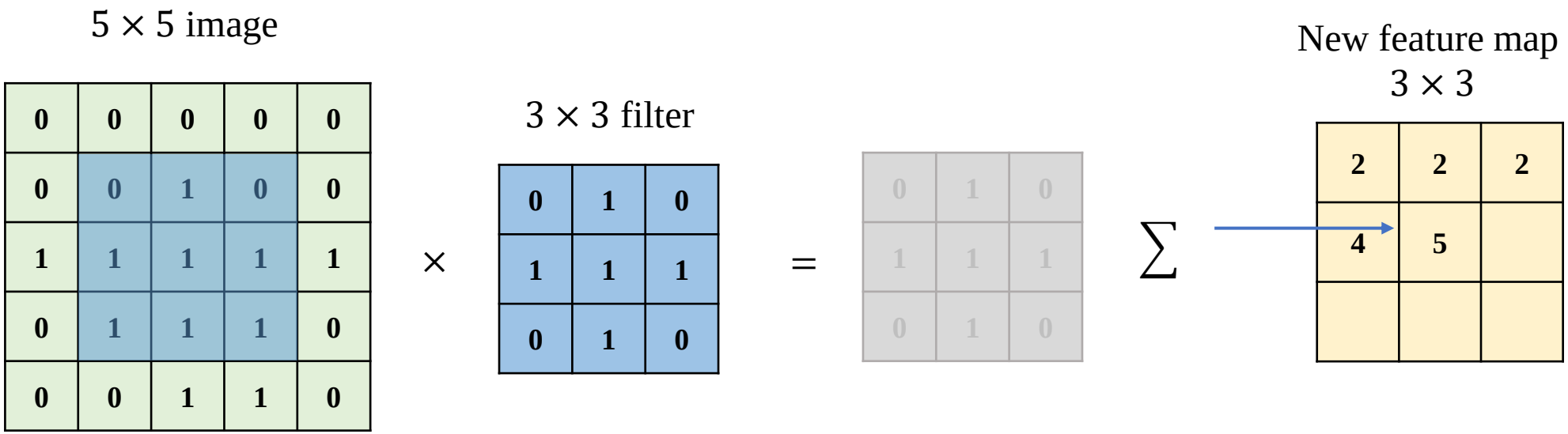
Convolutional layers – Filters example

✓ Slide with stride 1



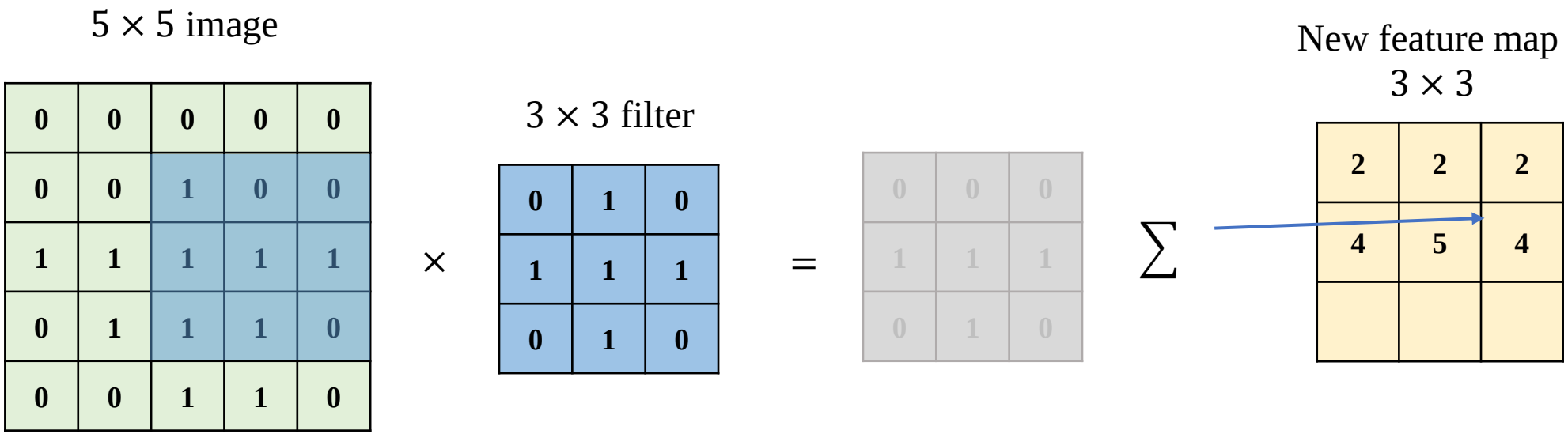
Convolutional layers – Filters example

✓ Slide with stride 1



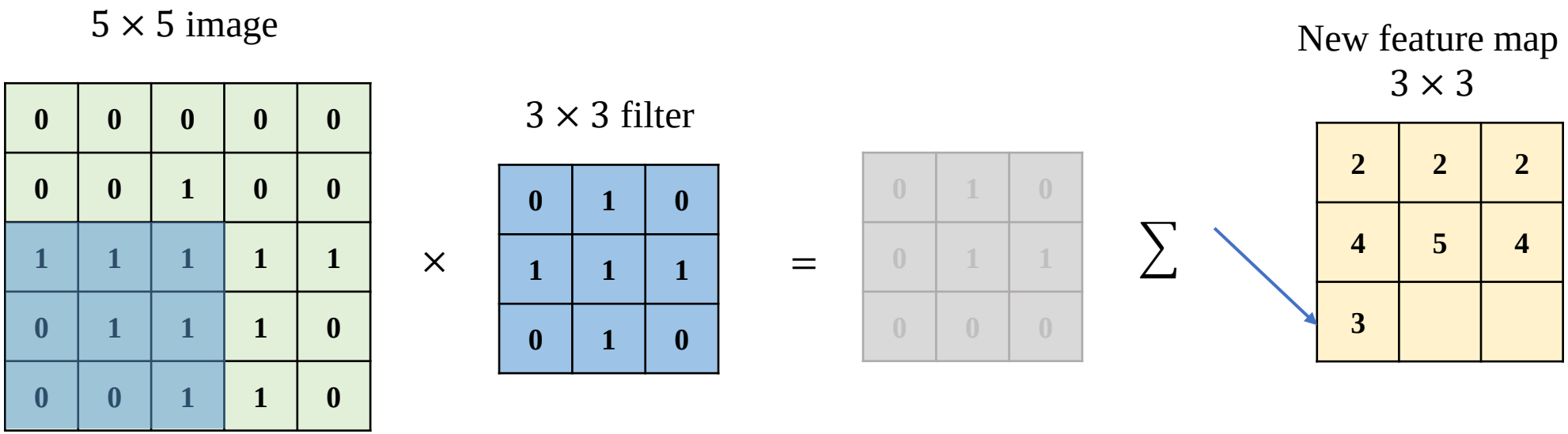
Convolutional layers – Filters example

✓ Slide with stride 1



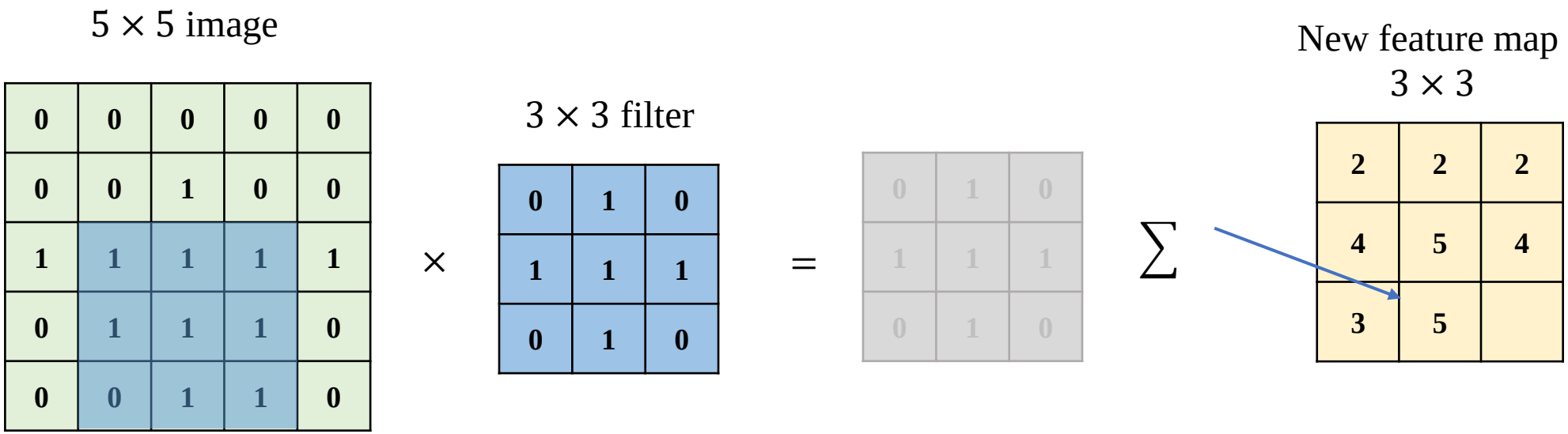
Convolutional layers – Filters example

✓ Slide with stride 1



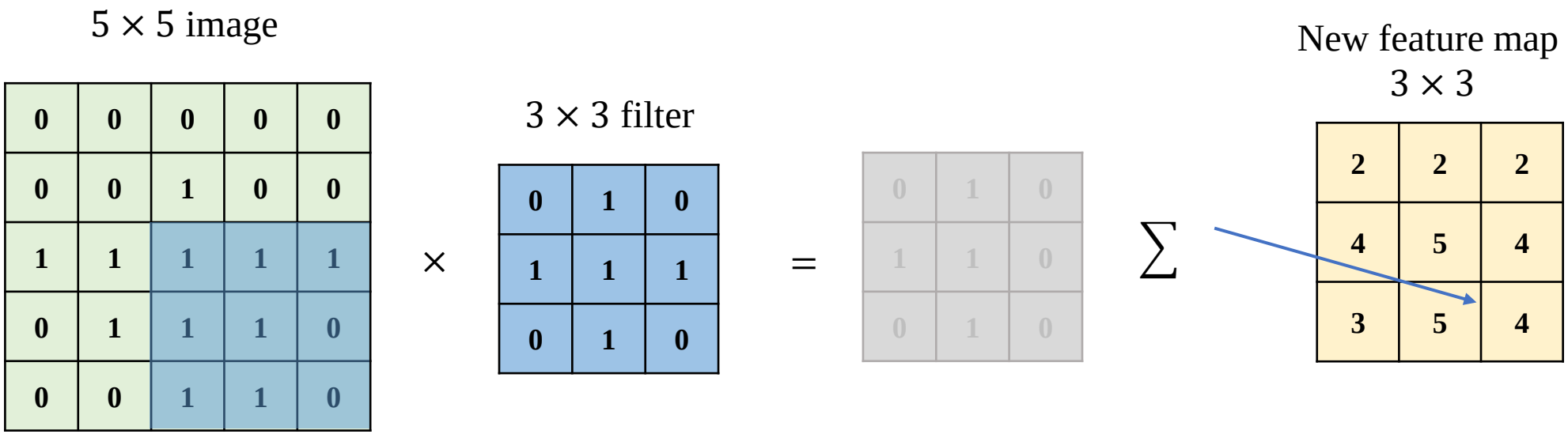
Convolutional layers – Filters example

✓ Slide with stride 1



Convolutional layers – Filters example

✓ Slide with stride 1



0	0	0	0	0
0	0	1	0	0
1	1	1	1	1
0	1	1	1	0
0	0	1	1	0

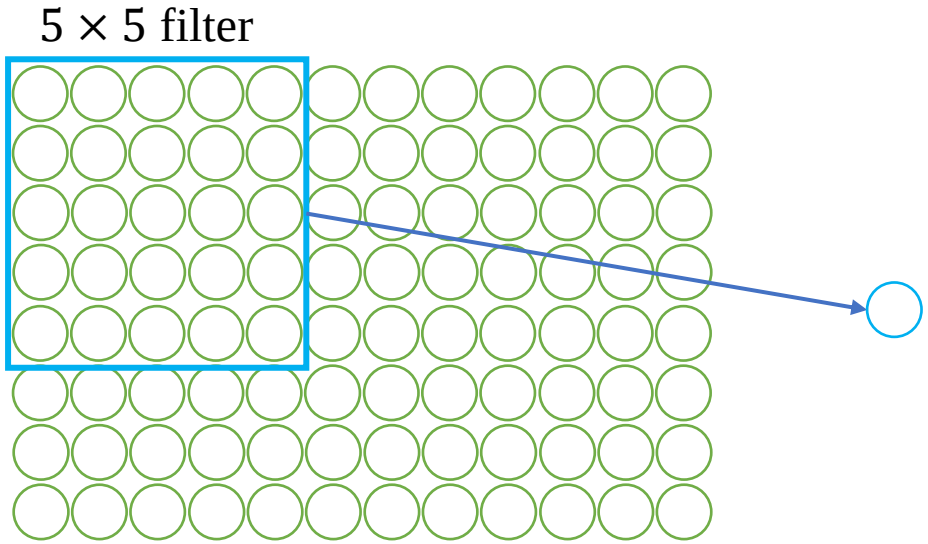
0	1	0
1	1	1
0	1	0

0	1	0
1	1	0
0	1	0

2	2	2
4	5	4
3	5	4

Sliding Window - Convolutional layers

✓ We have matrix or window of weights w_{ij} that linear combination of them after non-linear activation gives us a new neuron

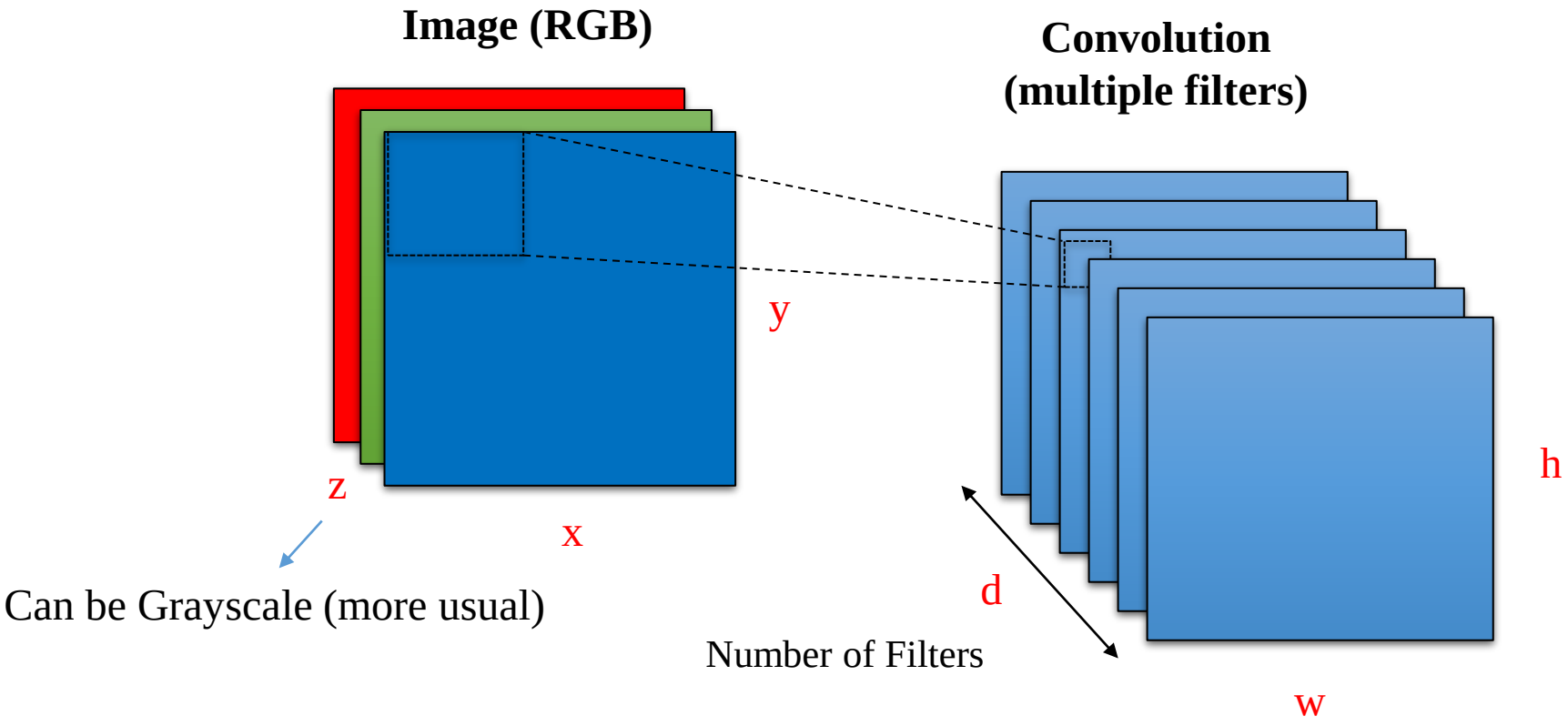


$$\sum_{i=1}^{5}\sum_{j=1}^{5} w_{ij}x_{i+n,j+m} + b$$

Computing weighed sum of patch for neuron (n, m) and add bias

```
self.conv1 = nn.Conv2d(5, 5, kernel_size=3, stride=1, padding=0)
```

Convolutional Neural Network (CNNs)



Problem?

The Size is increasing!
(maybe, why maybe?)

Pooling - Maxpooling

Solution

- ✓ Using pooling layer
- ✓ Maxpooling is common down sampling technique

6 × 6 image

3 × 3 filters

3	0	0	0	1	1
2	3	1	2	2	0
1	3	1	1	1	1
0	5	1	0	1	0
1	5	4	1	1	0
3	0	2	0	0	1

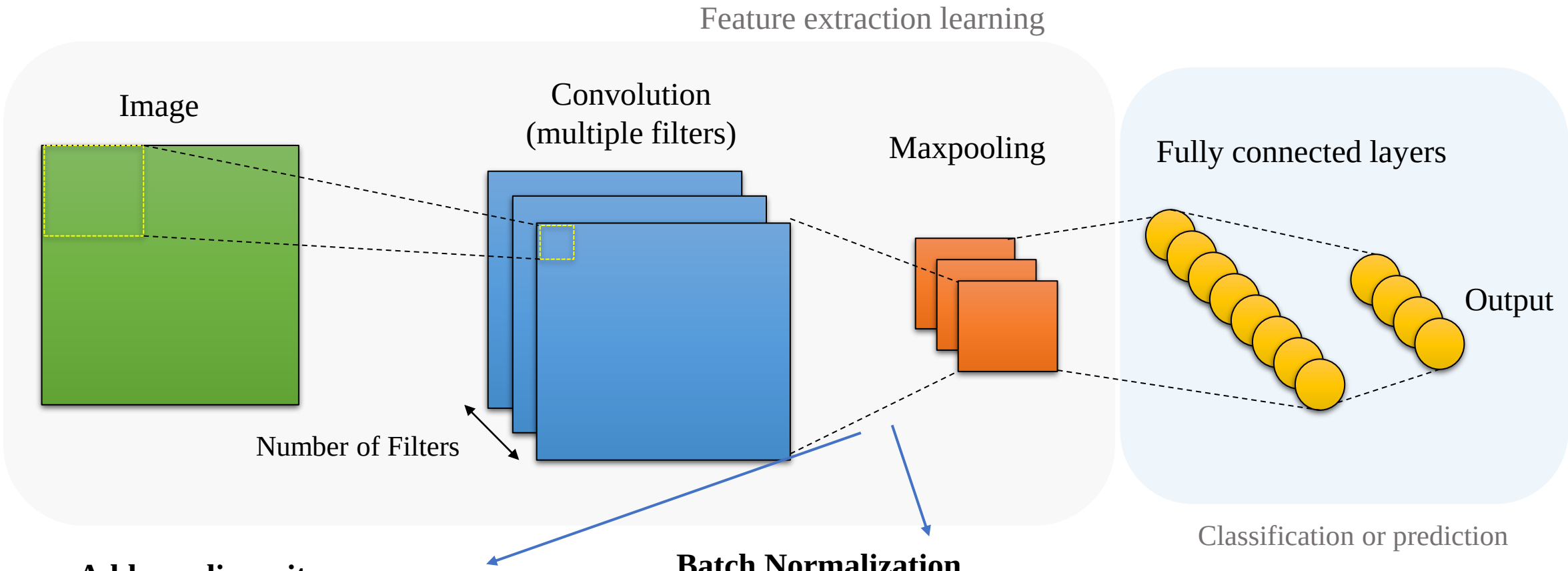
3 × 3 filter for max pooling with stride 3

Down sampled

3	2
5	1

```
self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
```

Convolutional Neural Network (CNNs)



Add non-linearity

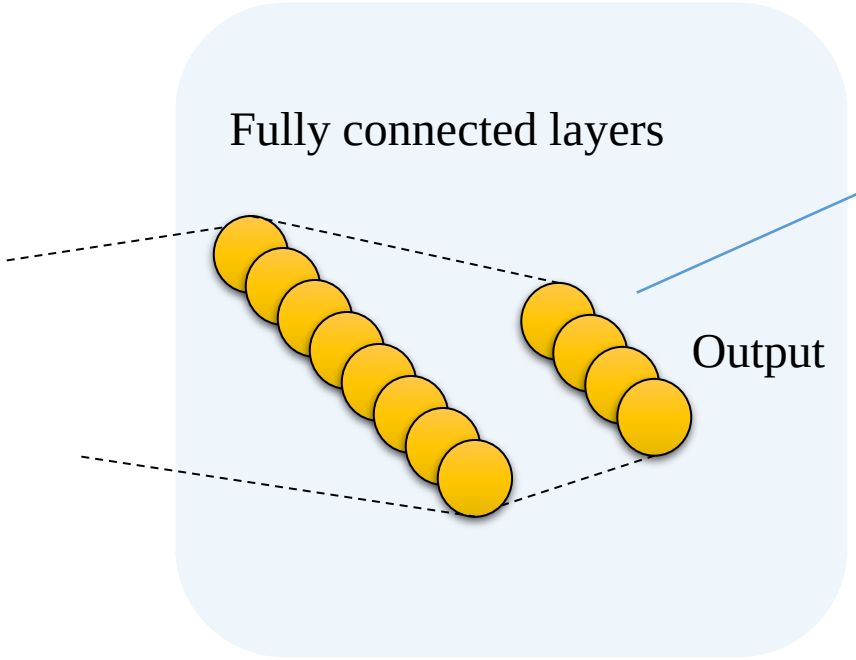
- ✓ Mostly ReLU in DL
- ✓ It removes negative values

Batch Normalization

- ✓ Normalization of the layers' inputs by re-centering and re-scaling
- ✓ faster and more stable learning

```
self.bn1 = nn.BatchNorm2d(16)
```

Convolutional Neural Network (CNNs)



Classification or prediction

If classification we usually have a SoftMax in last layer

SoftMax

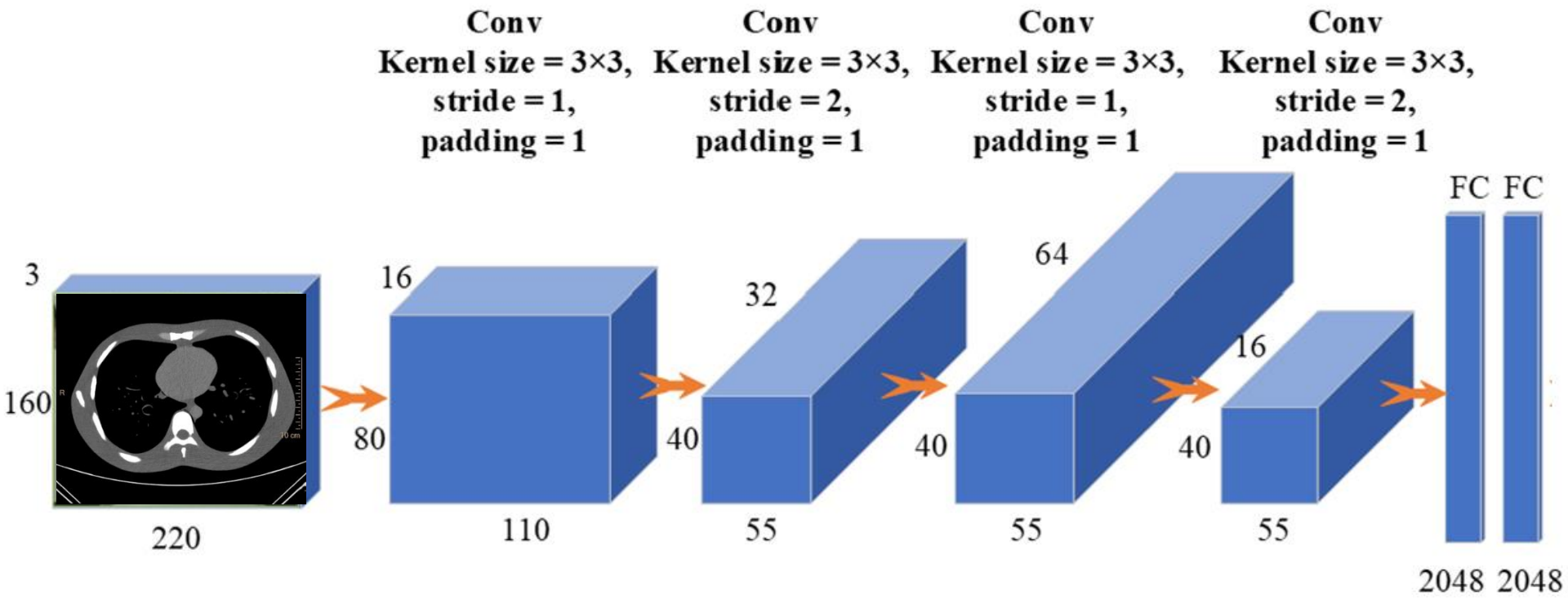
✓ Softmax is a mathematical function that converts a vector of numbers into a **vector of probabilities** (sum of all is 1)

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

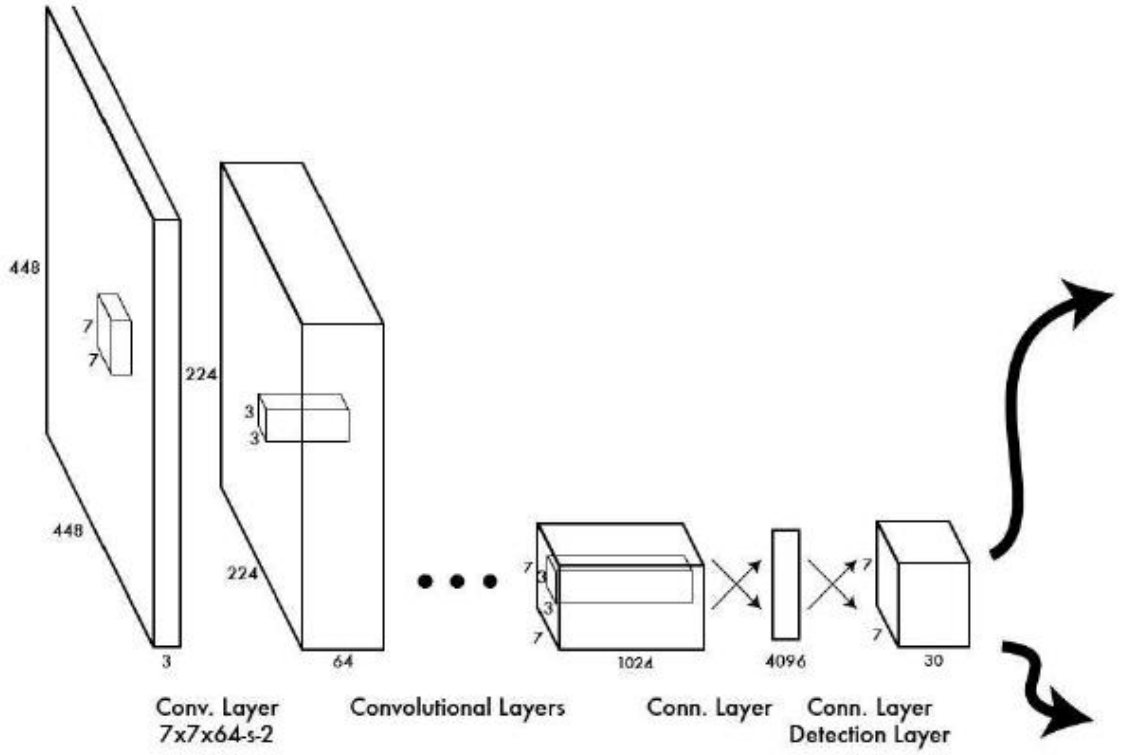
```
x = nn.Softmax(dim=1)
```

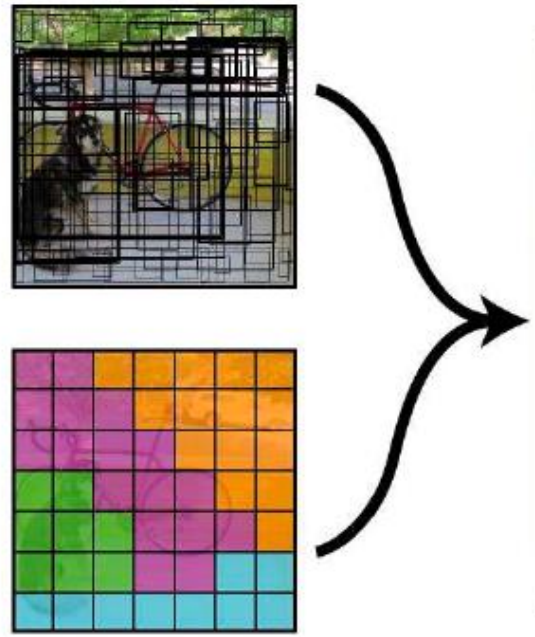
dim: normalizes values along wich axis

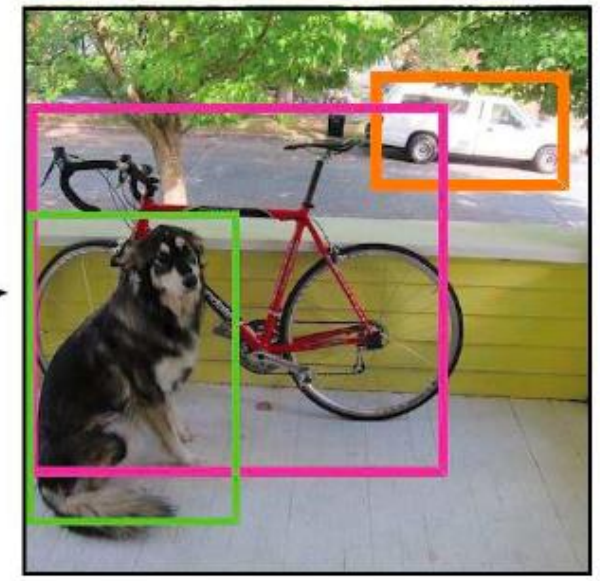
Multi-layer CNNs



YOLO (You Only Look Once)







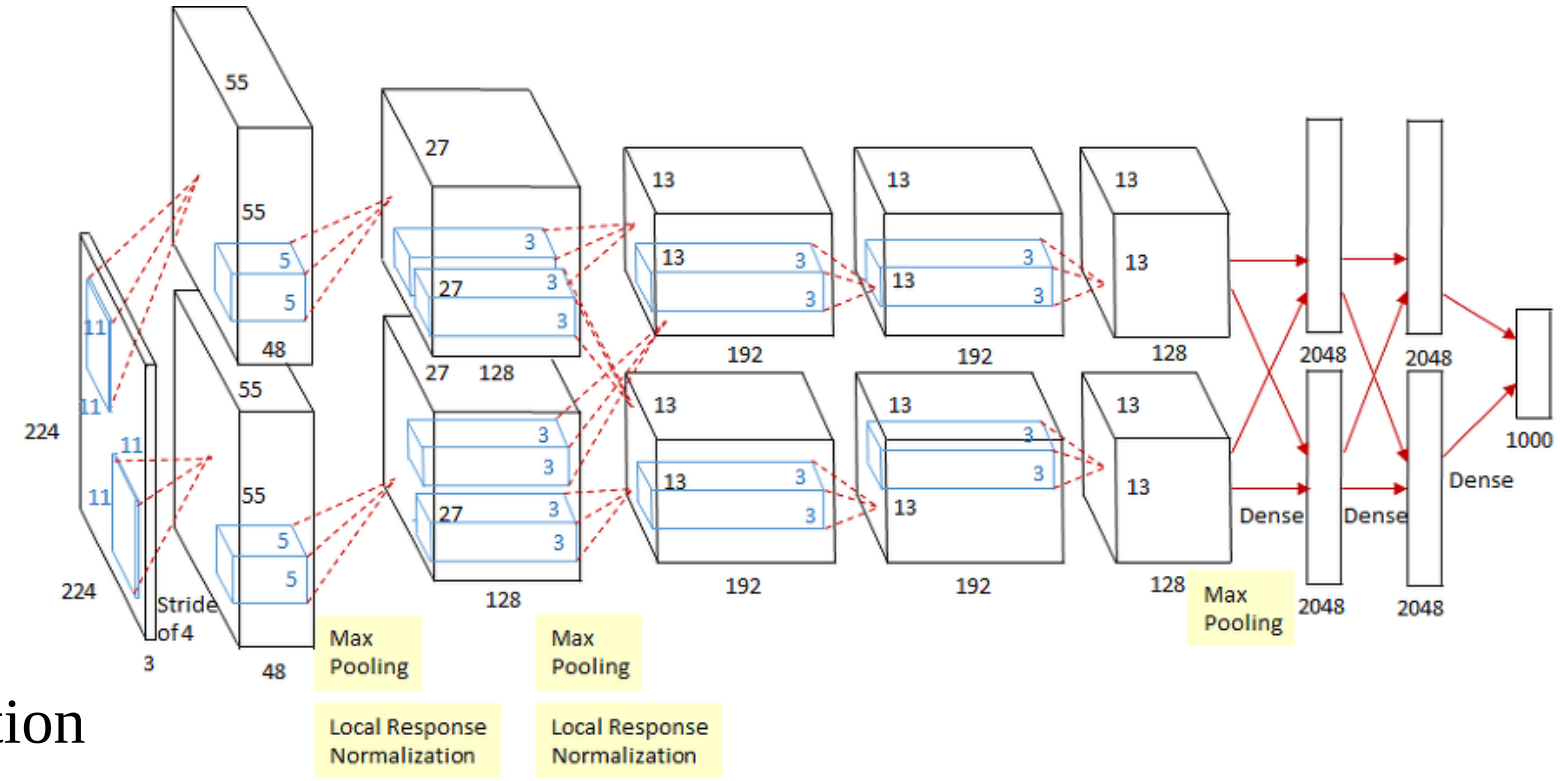
We have formal versions of (1-6) so far

https://dagshub.com/blog/yolov6/

Popular deep learning Architectures:

- ✓ LeNet5.
- ✓ Dan Ciresan Net
- ✓ AlexNet
- ✓ Overfeat
- ✓ VGG
- ✓ Network-in-network
- ✓ GoogLeNet and Inception
- ✓ Bottleneck Layer.

For instance: AlexNet Architecture



- ✓ It solves the image classification problem with 1000 different classes
- ✓ Output is a vector of 1000 numbers (probability of belonging)

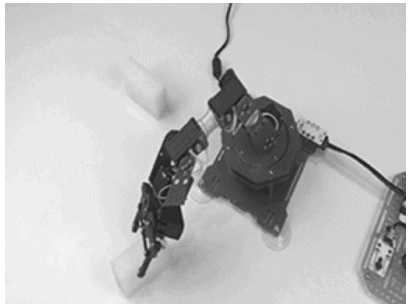
Popular deep learning algorithms:

- ✓ Convolutional Neural Networks (CNNs)
- ✓ Long Short Term Memory Networks (LSTMs)
- ✓ Recurrent Neural Networks (RNNs)
- ✓ Generative Adversarial Networks (GANs)
- ✓ Radial Basis Function Networks (RBFNs)
- ✓ Multilayer Perceptrons (MLPs)
- ✓ Self Organizing Maps (SOMs)

Sample environment

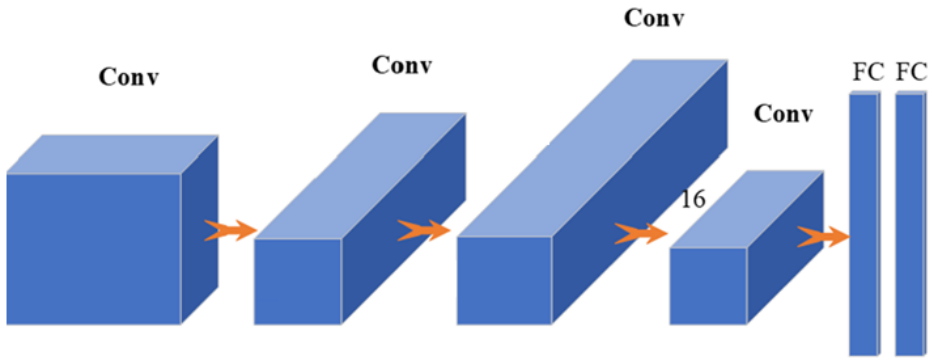
Image

n



m

State-space



Feature extraction

Action-value
State-value
…

Collecting all together

```
class DN(nn.Module):
    def __init__(self, h, w, outputs):
        super(DN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, stride=2, padding=0)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2, padding=0)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2, padding=0)
        self.bn3 = nn.BatchNorm2d(32)
        nn.MaxPool2d(2)

        def conv(input_size, kernel_size=5, stride=2):
            return ((input_size - (kernel_size - 1) - 1) // stride + 1)

        size_of_conv_w = conv(conv(conv(w)))
        size_of_conv_h = conv(conv(conv(h)))

        linear_input_size = size_of_conv_w * size_of_conv_h * 32
        self.head = nn.Linear(linear_input_size, linear_input_size)
        self.head = nn.Linear(linear_input_size, outputs)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))

        return self.head(x.view(x.size(0), -1))
```

Summery

In this chapter we discussed the concepts of following topics:

- ✓ The Perceptron
- ✓ Neural Networks
- ✓ Optimization
- ✓ backpropagation
- ✓ Adaptive learning
- ✓ Batching
- ✓ Regularization