# NEURAL NETWORKS

# Terminology

If $\mathbf{w}^T\mathbf{x} + w_0 > 0$ assign $\mathbf{x}$ to $\omega_1$

If $\mathbf{w}^T\mathbf{x} + w_0 < 0$ assign $\mathbf{x}$ to $\omega_2$

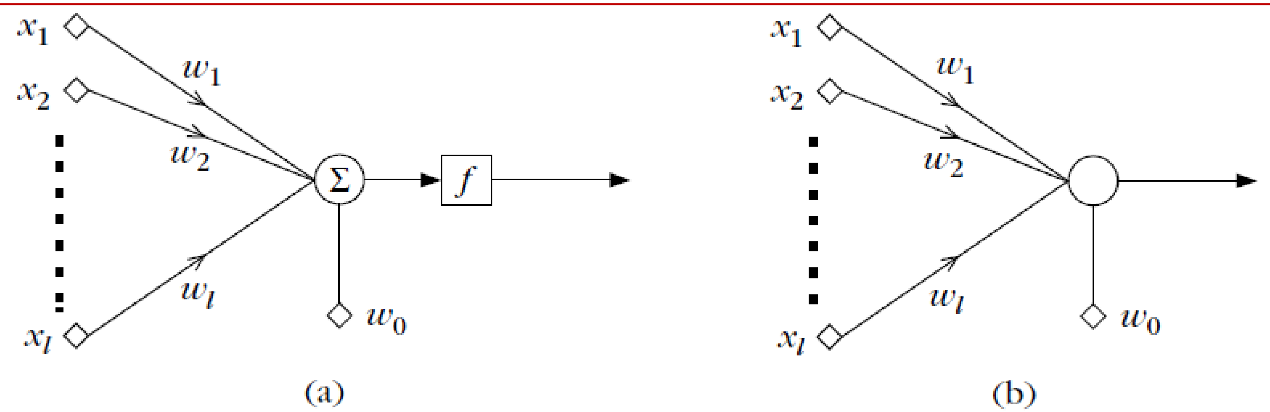


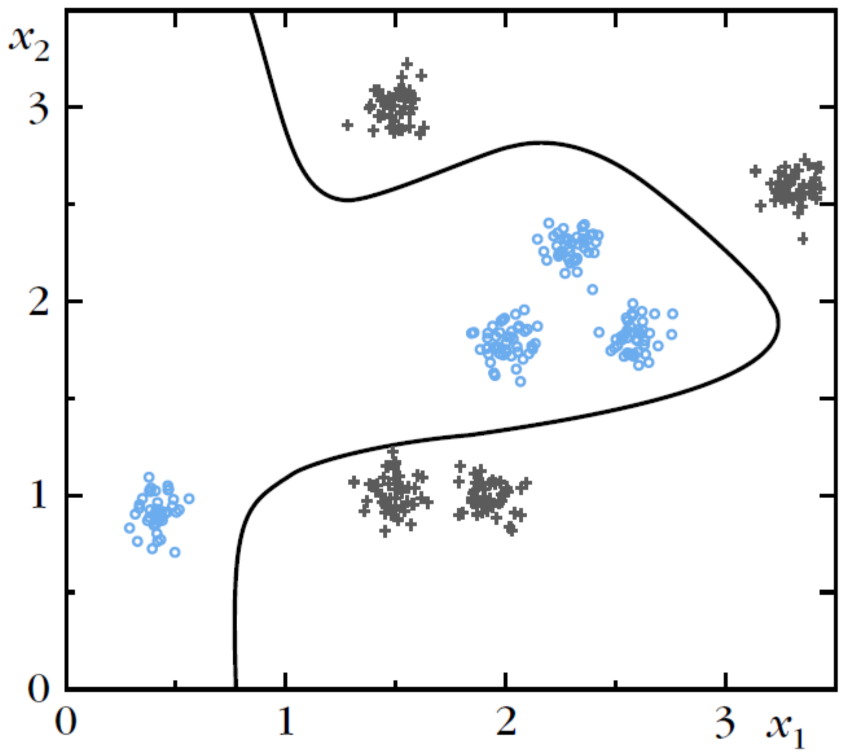

**FIGURE 3.5**

The basic perceptron model. (a) A linear combiner is followed by the activation function. (b) The combiner and the activation function are merged together.

- **Perceptron** or **neuron**
- **Synaptic weights** or **synapses**
- **Activation function**: *e.g.* $f(x) = s(x)$ (step function)

# Nonlinear Classifiers

We deal with problems that are not linearly separable



# ONE! TWO! THREE!

# One-Layer Perceptron

- XOR problem is not linearly separable

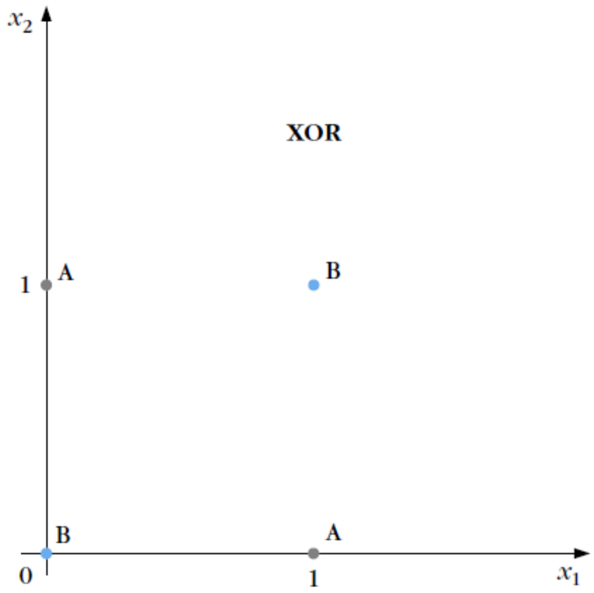
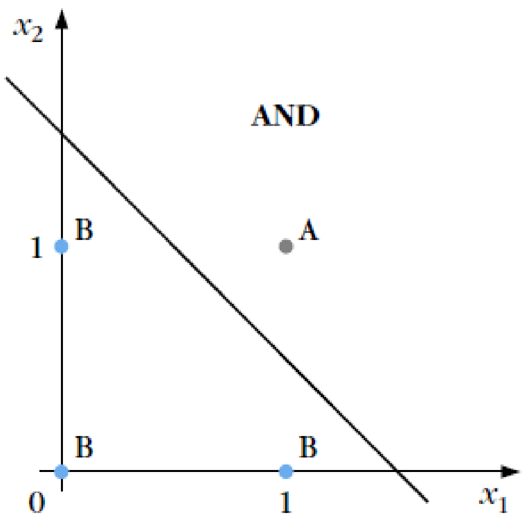



**Table 4.1** Truth Table for the XOR Problem

| $x_1$ | $x_2$ | XOR | Class |
|---|---|---|---|
| 0 | 0 | 0 | B |
| 0 | 1 | 1 | A |
| 1 | 0 | 1 | A |
| 1 | 1 | 0 | B |

# One-Layer Perceptron

- AND and OR problems are linearly separable



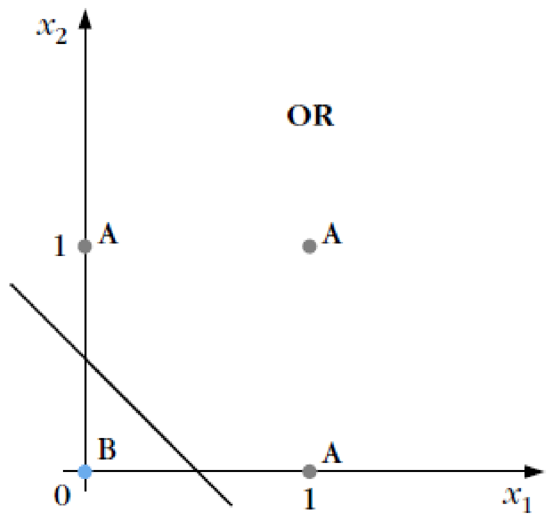
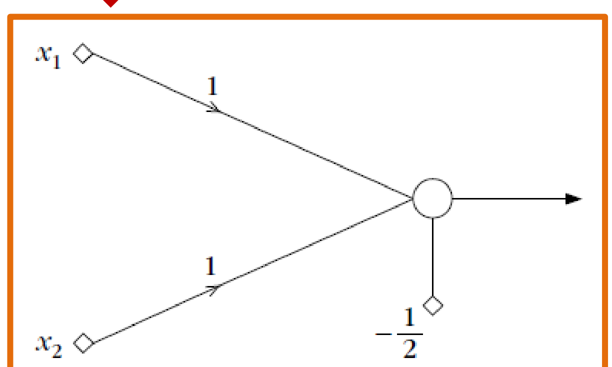




**Table 4.2** Truth Table for AND and OR Problems

| $x_1$ | $x_2$ | AND | Class | OR | Class |
|---|---|---|---|---|---|
| 0 | 0 | 0 | B | 0 | B |
| 0 | 1 | 0 | B | 1 | A |
| 1 | 0 | 0 | B | 1 | A |
| 1 | 1 | 1 | A | 1 | A |

1-layer perceptron implementation



# Two-Layer Perceptron

- XOR problem: solve it in two successive phases
  - $1^{st}$ phase (or layer) uses two lines

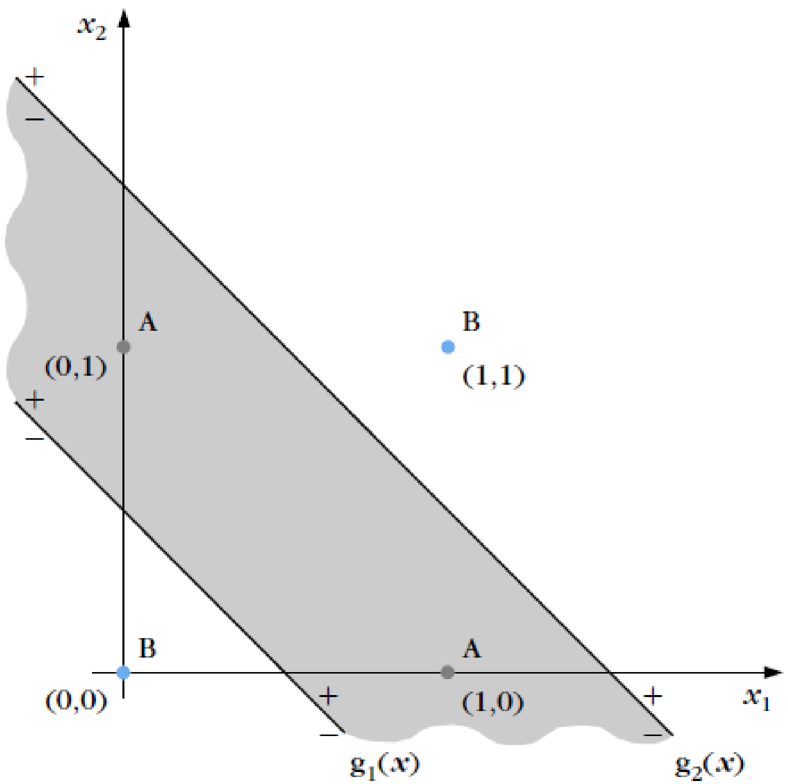



**Table 4.3** Truth Table for the Two Computation Phases of the XOR Problem

| 1st Phase | | | | |
|---|---|---|---|---|
| $x_1$ | $x_2$ | $y_1$ | $y_2$ | 2nd Phase |
| 0 | 0 | 0 (−) | 0 (−) | B (0) |
| 0 | 1 | 1 (+) | 0 (−) | A (1) |
| 1 | 0 | 1 (+) | 0 (−) | A (1) |
| 1 | 1 | 1 (+) | 1 (+) | B (0) |

# Two-Layer Perceptron

- XOR problem: solve it in two successive phases
  - $2^{nd}$ phase

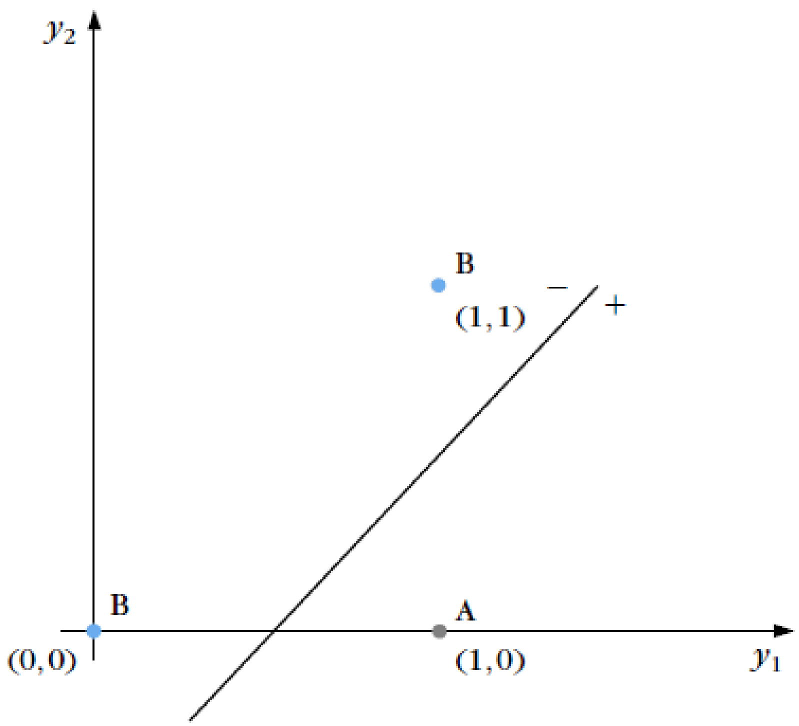
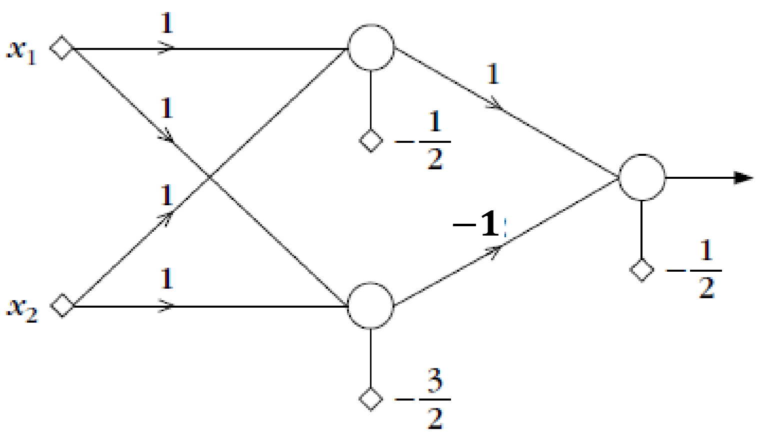



**Table 4.3** Truth Table for the Two Computation Phases of the XOR Problem

| 1st Phase | | | | 2nd Phase |
|---|---|---|---|---|
| $x_1$ | $x_2$ | $y_1$ | $y_2$ | |
| 0 | 0 | 0 (−) | 0 (−) | B (0) |
| 0 | 1 | 1 (+) | 0 (−) | A (1) |
| 1 | 0 | 1 (+) | 0 (−) | A (1) |
| 1 | 1 | 1 (+) | 1 (+) | B (0) |

# Two-Layer Perceptron

- XOR problem: solve it in two successive phases
  - 2-layer perceptron (or 2-layer feedforward neural network)



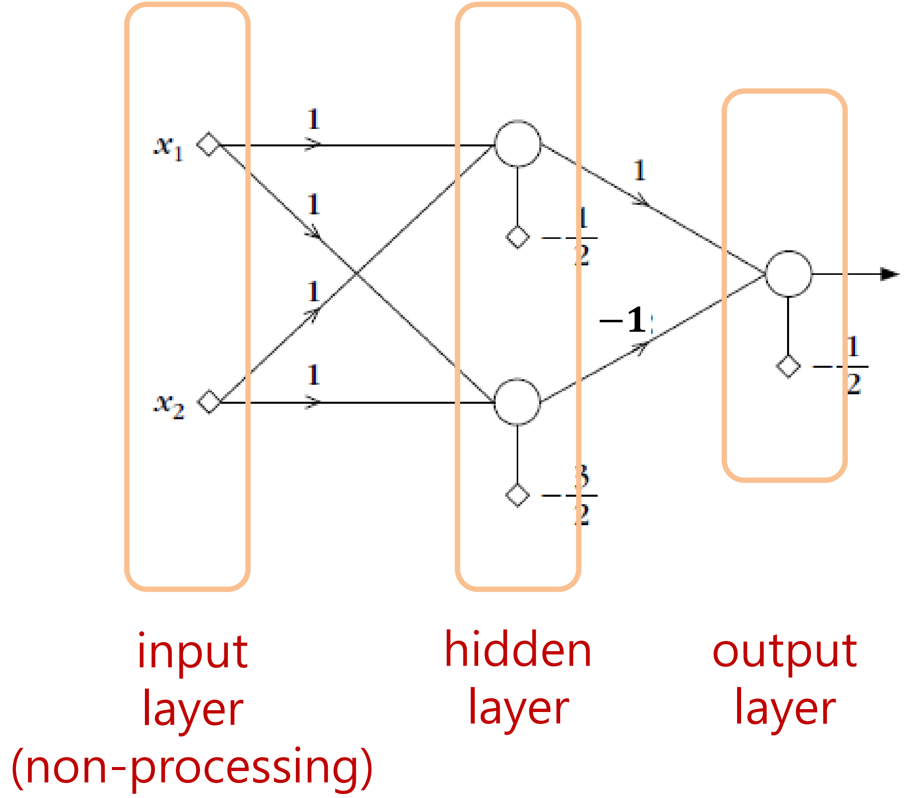

- $g_1(\mathbf{x}) = x_1 + x_2 - \frac{1}{2} = 0$
- $g_2(\mathbf{x}) = x_1 + x_2 - \frac{3}{2} = 0$
- $g(\mathbf{y}) = y_1 - y_2 - \frac{1}{2} = 0$

**Table 4.3** Truth Table for the Two Computation Phases of the XOR Problem

| 1st Phase | | | | |
|---|---|---|---|---|
| $x_1$ | $x_2$ | $y_1$ | $y_2$ | 2nd Phase |
| 0 | 0 | 0 (−) | 0 (−) | B (0) |
| 0 | 1 | 1 (+) | 0 (−) | A (1) |
| 1 | 0 | 1 (+) | 0 (−) | A (1) |
| 1 | 1 | 1 (+) | 1 (+) | B (0) |

# Two-Layer Perceptron

- Terminology
  - 2-layer perceptron (or 2-layer feedforward neural network)

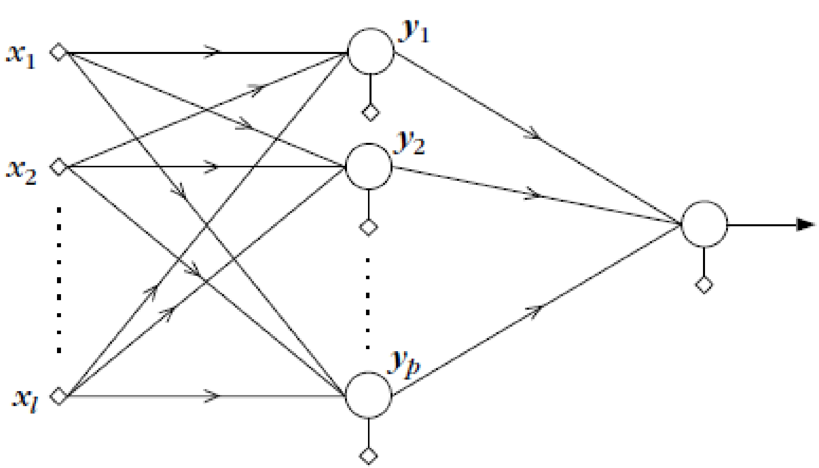


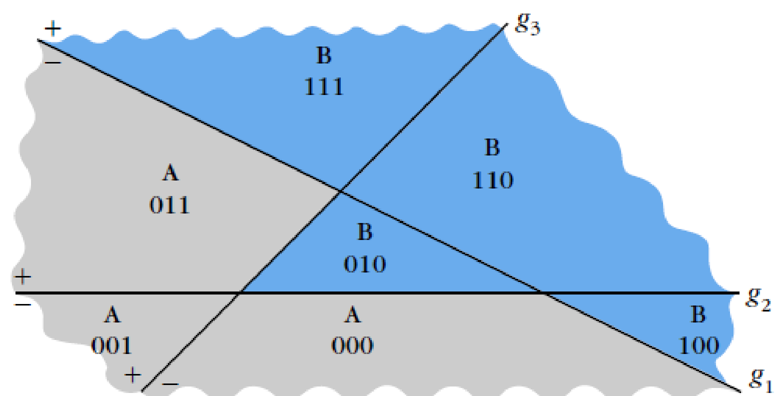
# Two-Layer Perceptron

- Classification capabilities of two-layer perceptron
  - 1st layer maps input to vertices of the unit hypercube

$$H_p = \left\{ \left[ y_1, \ldots, y_p \right]^T \in \mathbb{R}^p \colon \; y_i \in [0,1] \;\text{ for }\; 1 \le i \le p \right\}$$

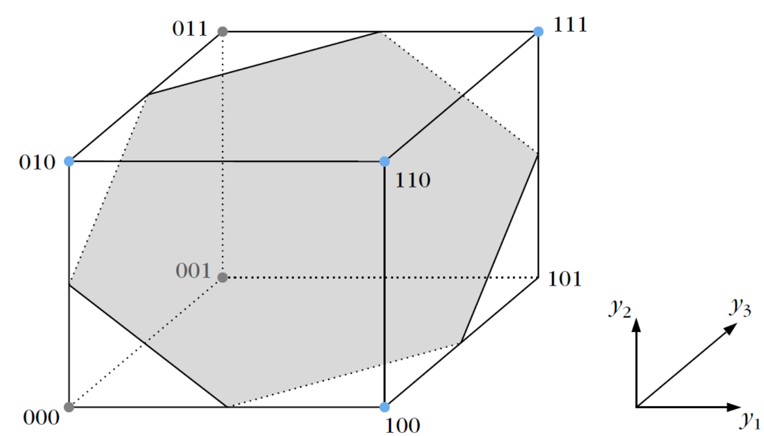
  - An output of 1st layer corresponds to a polyhedron

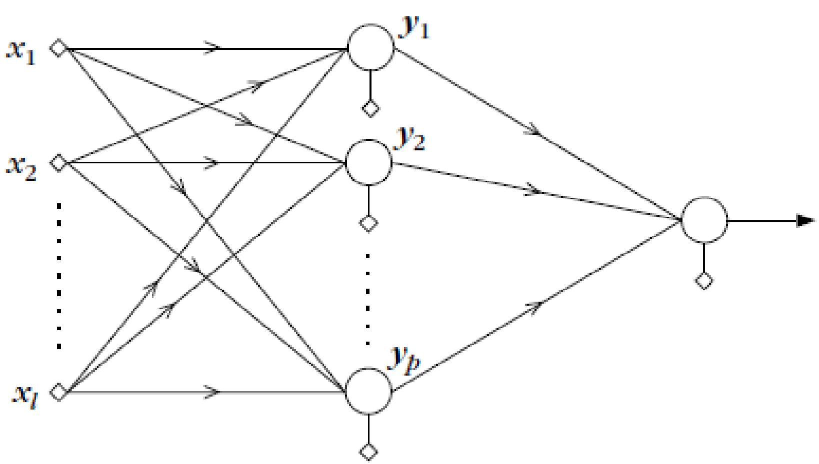


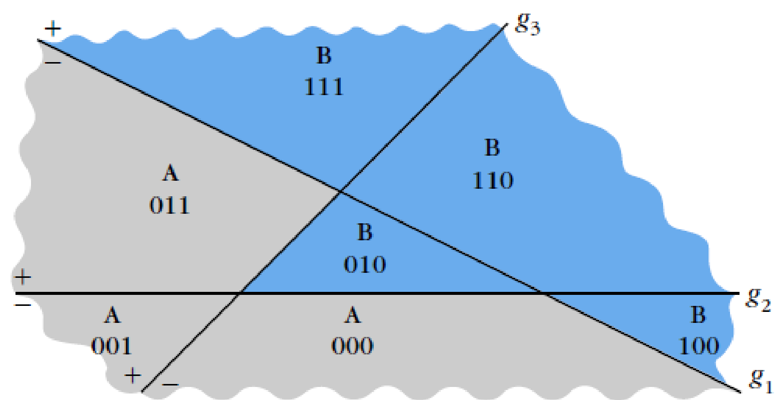



# Two-Layer Perceptron

- Classification capabilities of two-layer perceptron
  - $2^{nd}$ layer detects a union of selected polyhedron



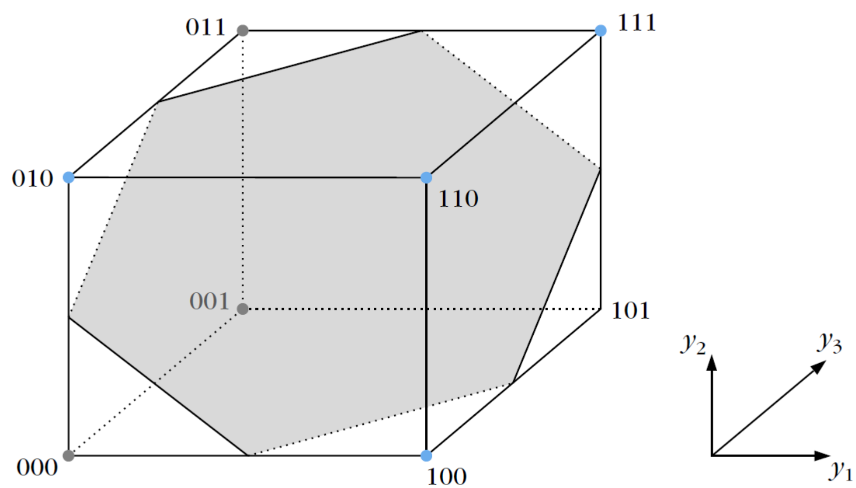



# Two-Layer Perceptron

- Classification capabilities of two-layer perceptron

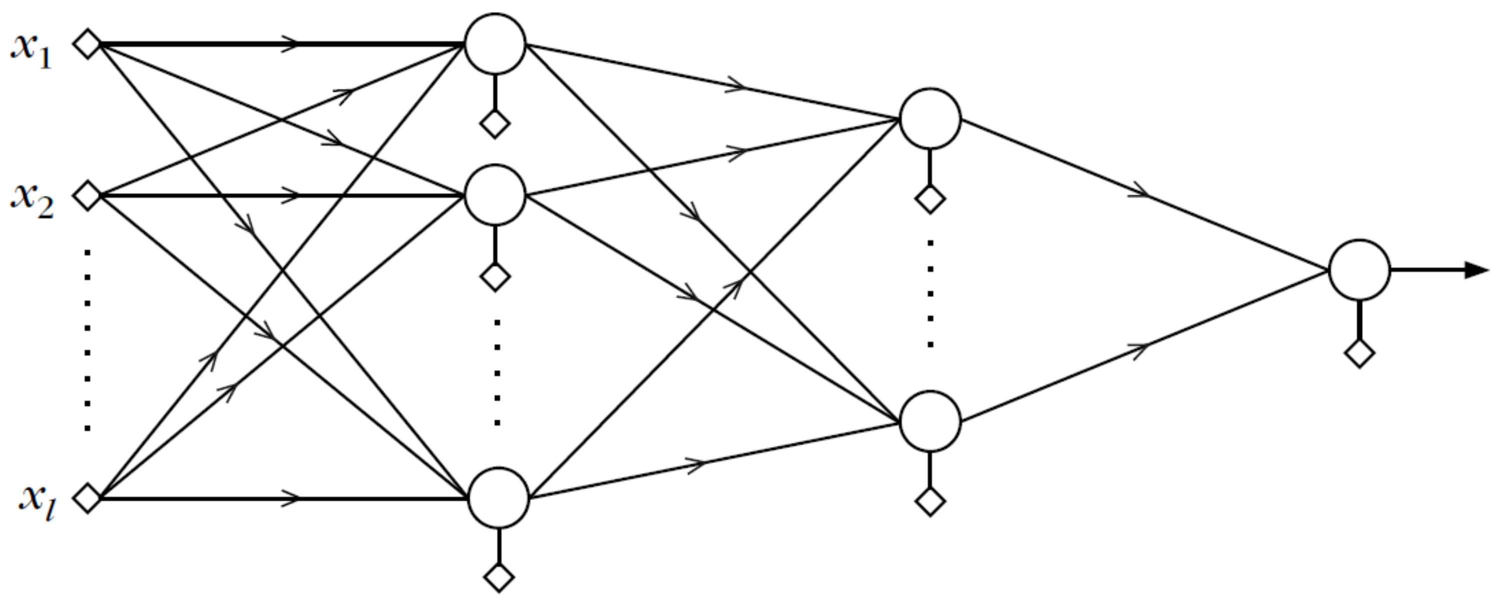
  Two-layer perceptron can detect a class, which consists of a union of polyhedral regions, but not any union of such regions



# Three-Layer Perceptron

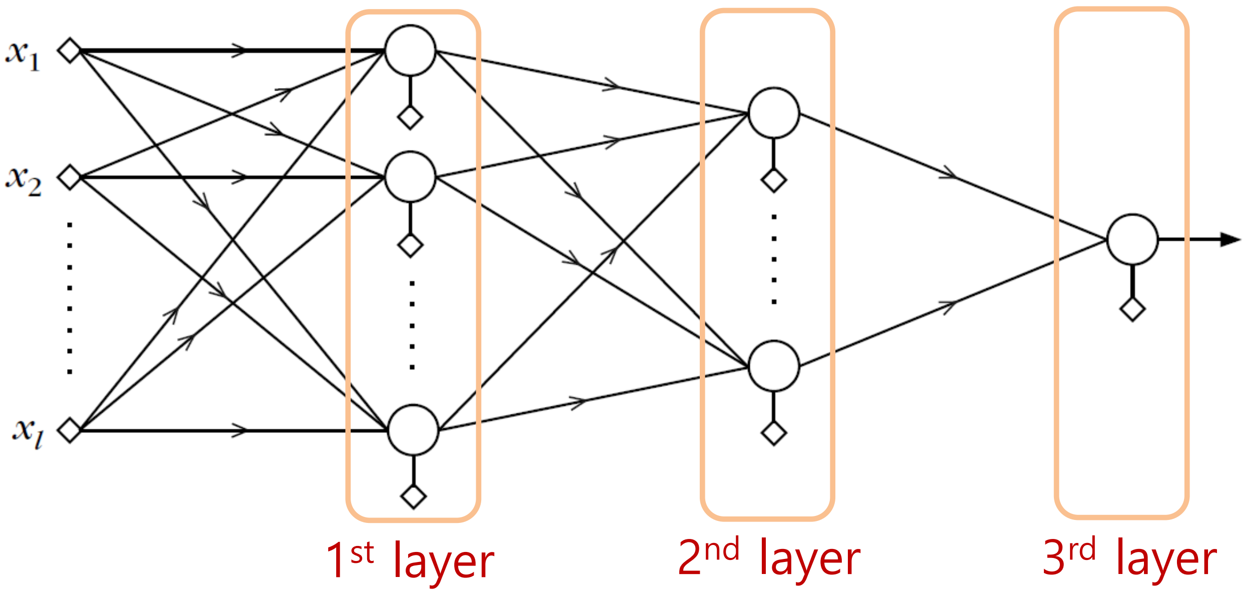
- Classification capabilities of three-layer perceptron

  Three-layer perceptron can detect a class, which consists of **any** union of polyhedral regions



# Three-Layer Perceptron

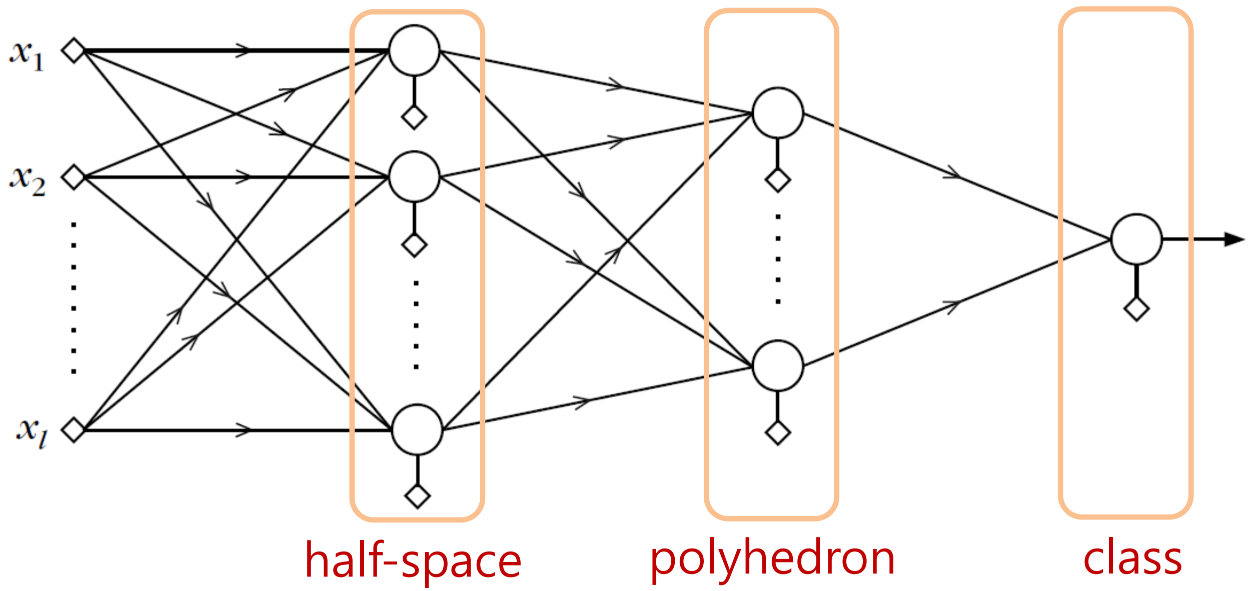
- Classification capabilities of three-layer perceptron




  - In 2nd layer, for each neuron, the synaptic weights are chosen so that the realized hyperplane leaves only one of the $H_p$ vertices on one side and all the rest on the other
  - 3rd layer implements OR gate

# Three-Layer Perceptron

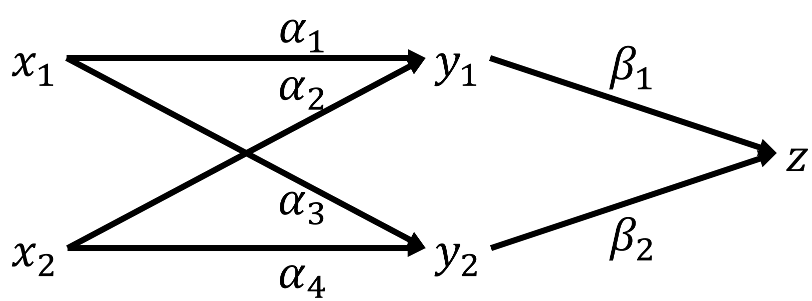
- Classification capabilities of three-layer perceptron



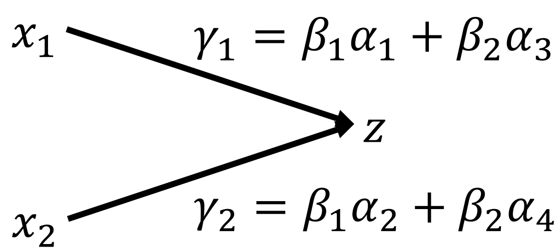

  - 1st layer detects half-spaces
  - 2nd layer detects polyhedra
  - 3rd layer detects a class, which is any union of polyhedra

# Nonlinearity

- A succession of two linear layers
  - $z = \beta_1 y_1 + \beta_2 y_2$
    $= \beta_1(\alpha_1 x_1 + \alpha_2 x_2) + \beta_2(\alpha_3 x_1 + \alpha_4 x_2)$




  - Simplify it to one linear layer



# Nonlinearity

- Activation function

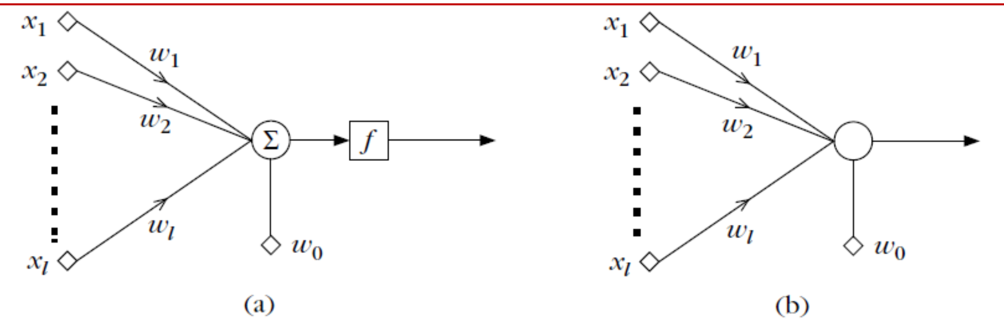
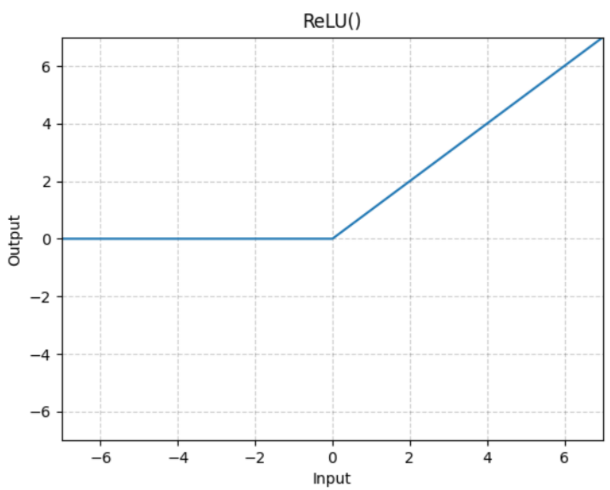
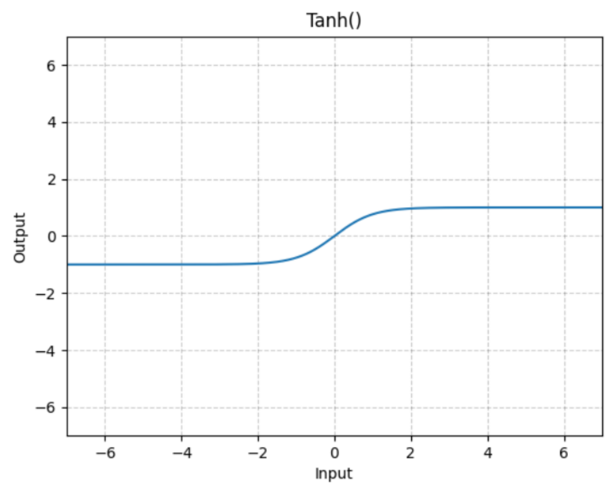
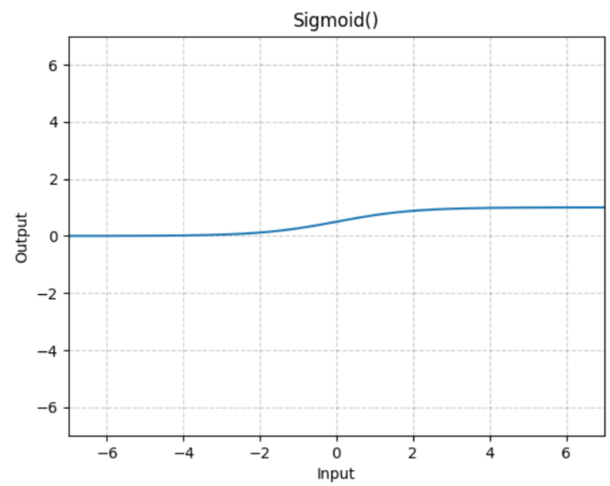



**FIGURE 3.5**

The basic perceptron model. (a) A linear combiner is followed by the activation function. (b) The combiner and the activation function are merged together.



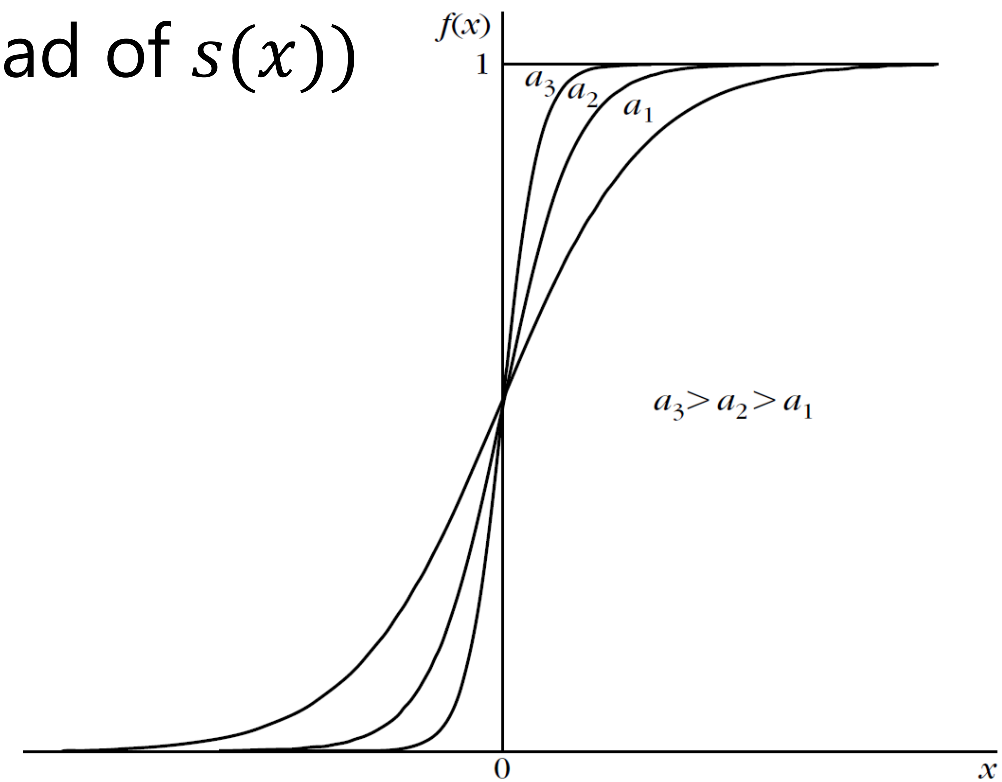



# BACKPROPAGATION ALGORITHM

# Multilayer Perceptron Design

- Design a multilayer perceptron
  - Fix an architecture, and optimize the synaptic weights
  - To use the gradient descent scheme, we need a continuous activation function
- Logistic function (instead of $s(x)$)
  - $f(x) = \frac{1}{1+\exp(-ax)}$



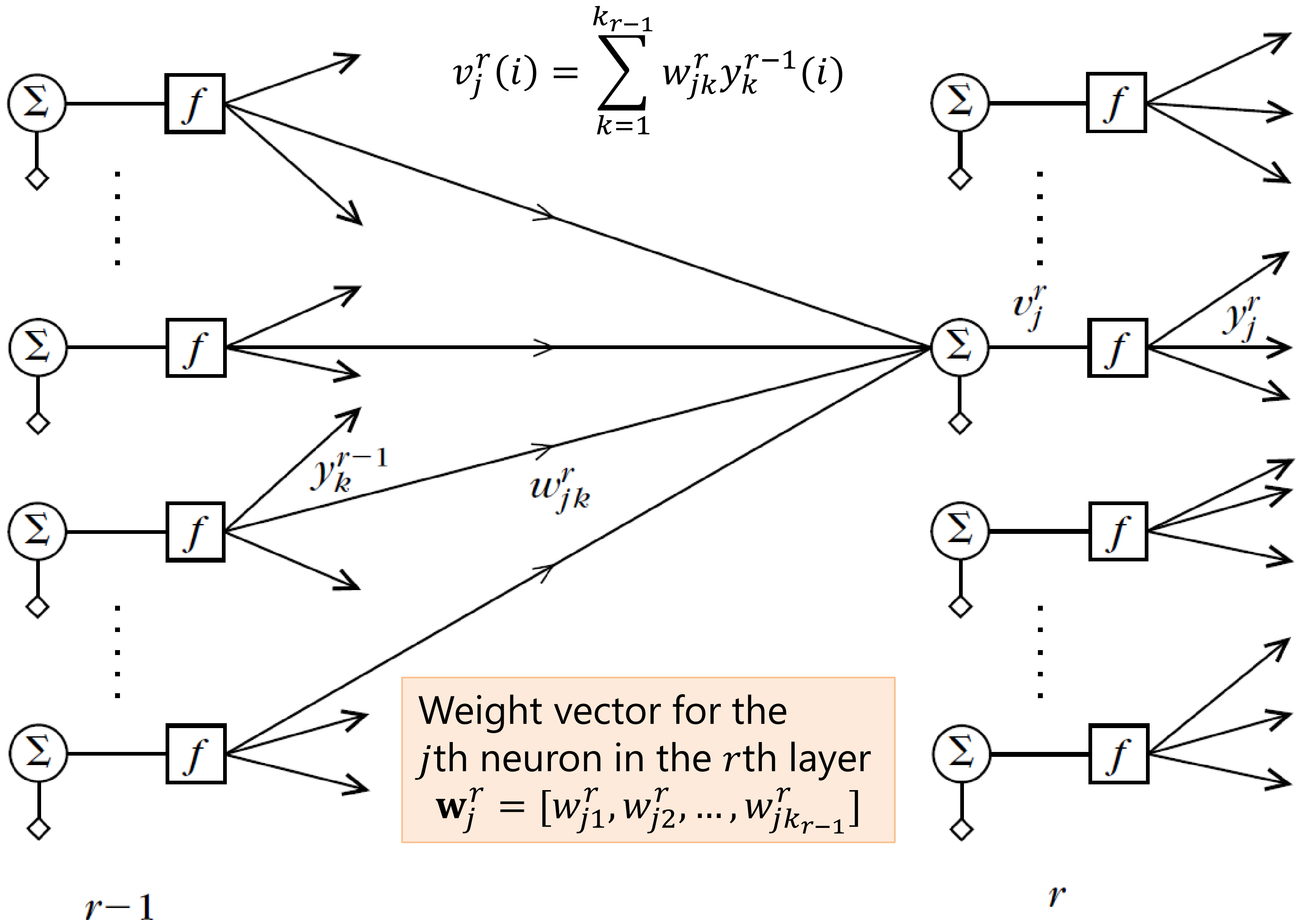
# Architecture and Formulation

- $L$ layers and $k_r$ neurons in the $r$th layer ($r = 1, \dots, L$)
  - $k_0 = l$ nodes in the input layer
  - $k_L$ output neurons
- $N$ training pairs, $(\mathbf{y}(i), \mathbf{x}(i))$, $i = 1, \dots, N$, are available
  - $\mathbf{y}(i) = \left[y_1(i), \dots, y_{k_L}(i)\right]^T$
  - $\mathbf{x}(i) = \left[x_1(i), \dots, x_{k_0}(i)\right]^T$
- During training, the actual output $\hat{\mathbf{y}}(i)$ is different from the desired one $\mathbf{y}(i)$
- Compute the synaptic weights to minimize

$$J = \frac{1}{N} \sum_{i=1}^{N} \mathcal{E}(i)$$

$$\mathcal{E}(i) = \frac{1}{2} \sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2} \sum_{m=1}^{k_L} \left(\hat{y}_m(i) - y_m(i)\right)^2$$

# Definition of Variables



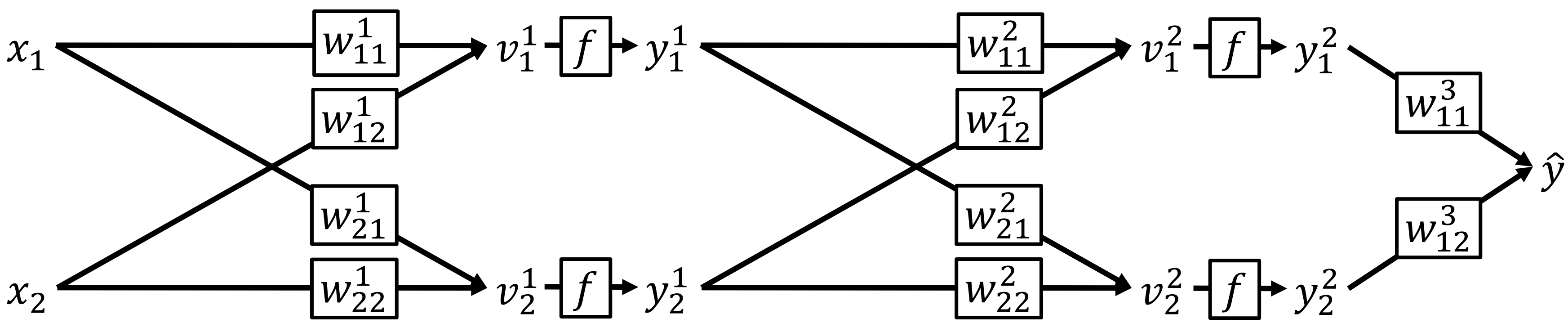
# Gradient Descent

$$\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \Delta\mathbf{w}_j^r$$

$$\Delta\mathbf{w}_j^r = -\eta\frac{\partial J}{\partial \mathbf{w}_j^r}$$

# Example

- Compute loss
  - Feedforward into three perceptron layers




  - Compute gradient and update its weight

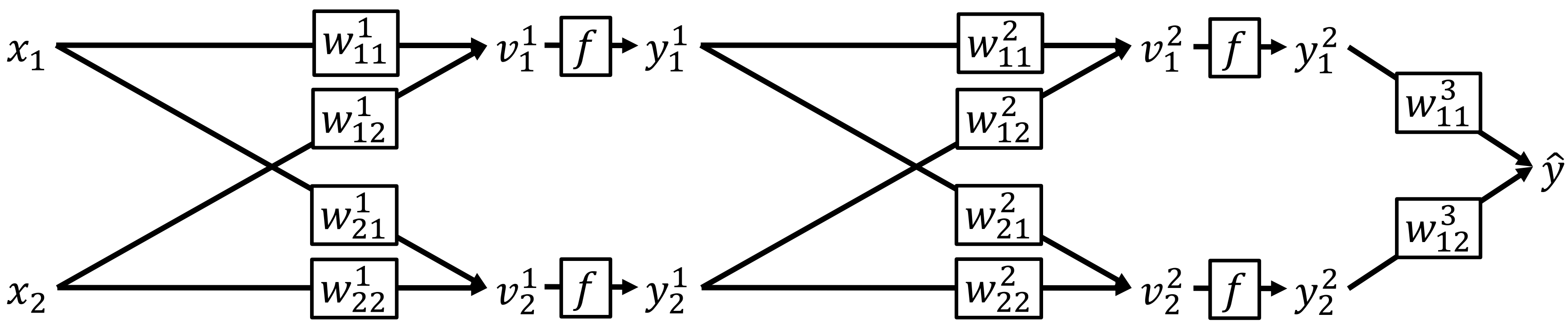
$$J = \frac{1}{N}\sum_{i=1}^{N} \mathcal{E}(i) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{2}(\hat{y}(i) - y(i))^2$$

$$\frac{\partial \mathcal{E}(i)}{\partial w_{11}^3} = \frac{1}{2}\frac{\left(w_{11}^3 y_1^2 - w_{12}^3 y_2^2\right)^2}{\partial W_{11}^3} = (\hat{y} - y) \times y_1^2$$

$$w_{11}^3(\text{new}) = w_{11}^3(\text{old}) - \eta \frac{\partial J}{\partial w_{11}^3}$$

# Example

- Compute loss
  - Feedforward into three perceptron layers




  - Compute gradient and update its weight

$$J = \frac{1}{N}\sum_{i=1}^{N} \mathcal{E}(i) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{2}\left(\hat{y}(i) - y(i)\right)^2$$

$$\frac{\partial \mathcal{E}(i)}{\partial w_{11}^2} = \frac{1}{2}\frac{\left(w_{11}^3\left(f\left(w_{11}^2 y_1^1 + w_{12}^2 y_2^1\right)\right) - w_{12}^3 y_2^2\right)^2}{\partial W_{12}^2} = (\hat{y} - y) \times w_{11}^3 \times \frac{\partial f\left(w_{11}^2 y_1^1 + w_{12}^2 y_2^1\right)}{\partial w_{12}^2}$$

$$w_{12}^2(\text{new}) = w_{12}^2(\text{old}) - \eta \frac{\partial J}{\partial w_{12}^2}$$

# Example

- Python code

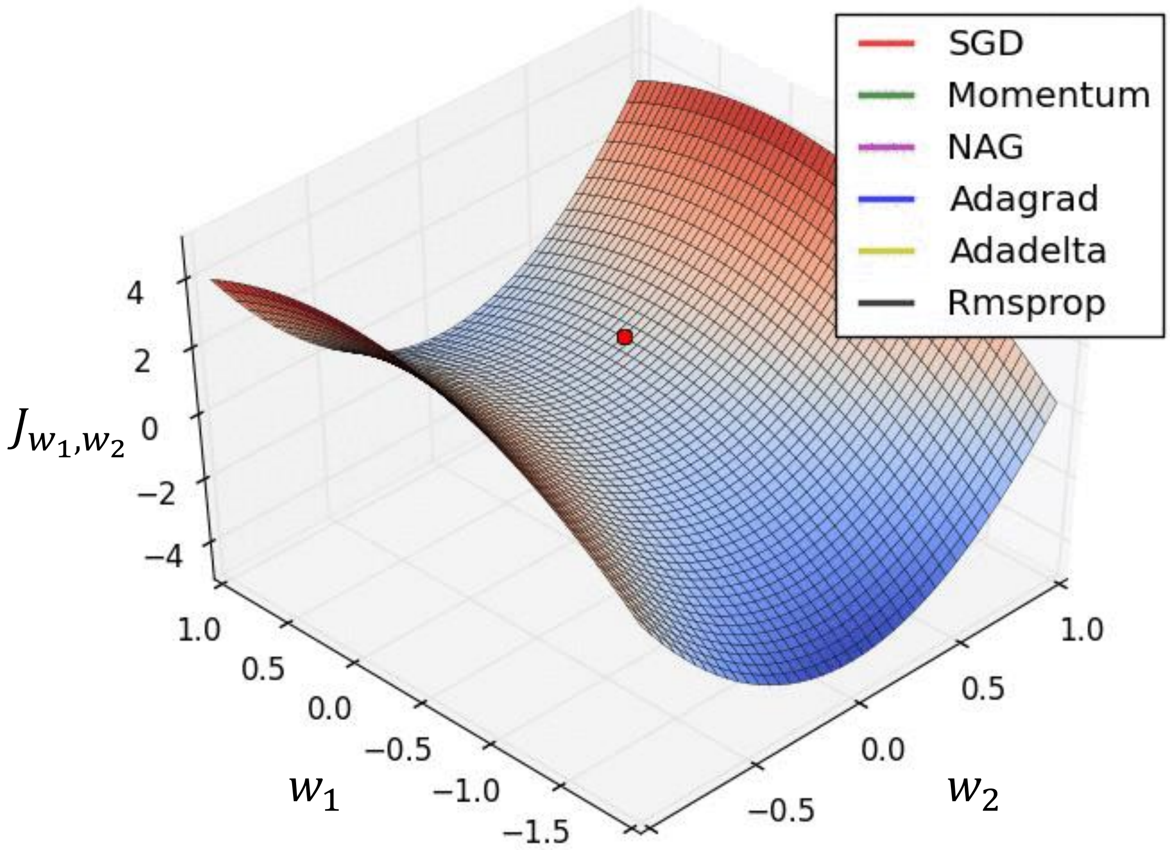
```python
import numpy as np

# Create random input and output data
x = np.array([0.05, 0.10])
y = np.array([0.01, 0.99])

# Randomly initialize weights
w1 = np.array([[0.15, 0.20], [0.25, 0.30]])
w2 = np.array([[0.40, 0.45], [0.50, 0.55]])
b1 = [0.35, 0.35]
b2 = [0.60, 0.60]

learning_rate = 0.01
```

```python
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1) + b1
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2) + b2

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
    b2 -= learning_rate * grad_y_pred
    b1 -= learning_rate * grad_h
```

# Update weights

- How to update the weights effectively

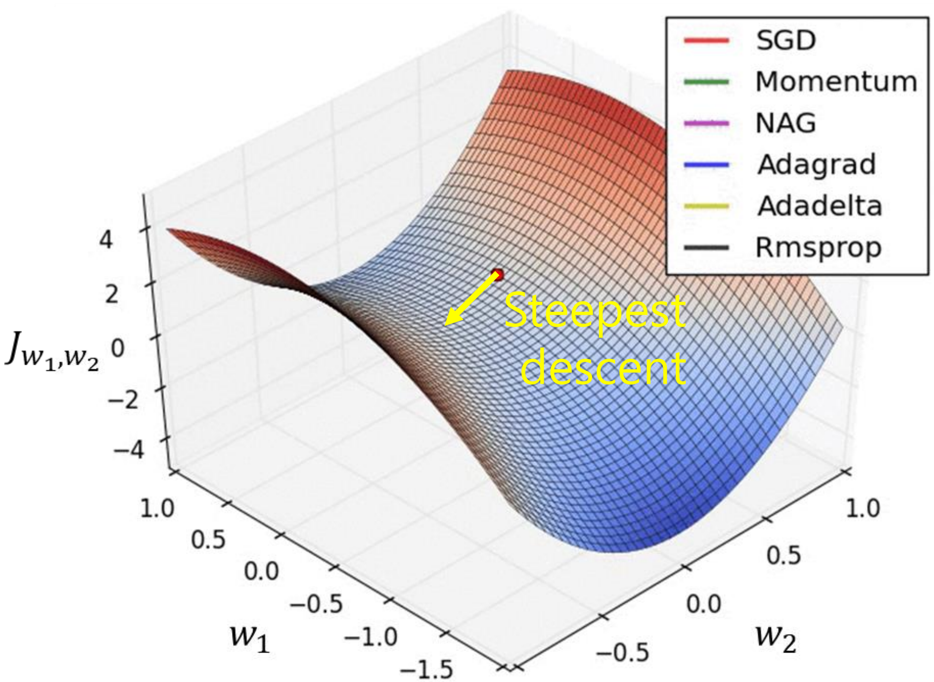
Contour of a loss function

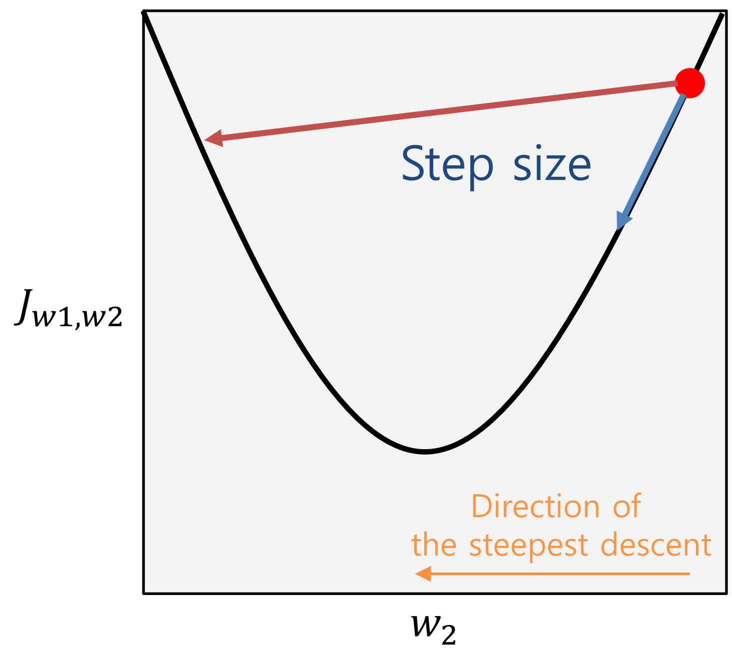


# Update weights

- How to update the weights effectively
  - Direction
  - Step size (learning rate)

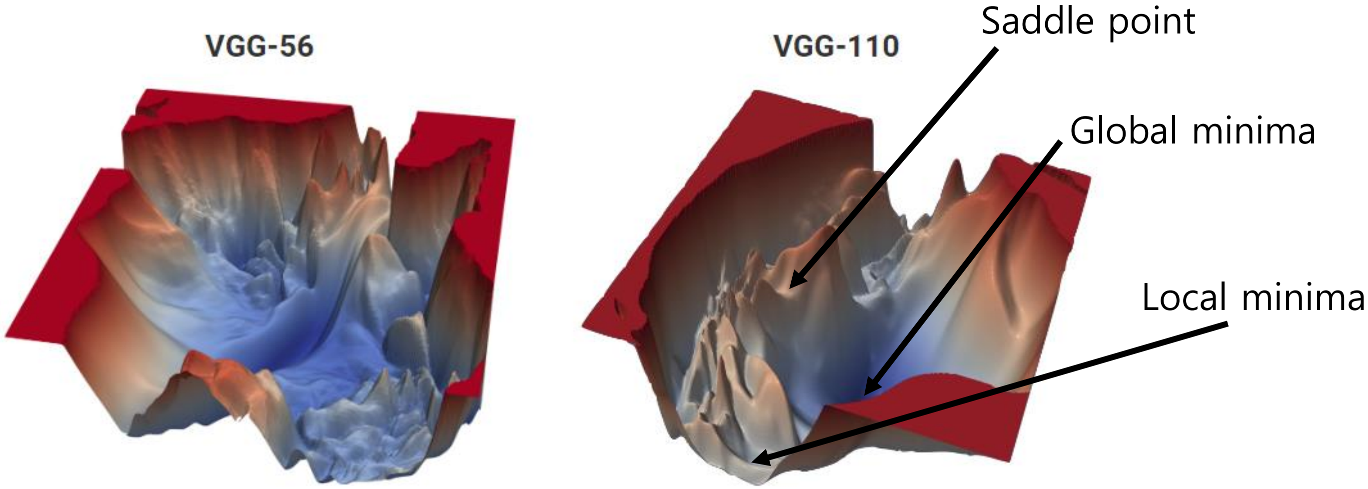
Contour of a loss function



# Gradient Descent

- Limitations of the gradient descent
  - Compute the gradient with an average of all training pairs
    - $J = \frac{1}{N}\sum_{i=1}^{N} \mathcal{E}(i)$
    - $\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \eta \frac{\partial J}{\partial \mathbf{w}_j^r}$
  - Take a long time when # of training pairs is large
  - Local minima & saddle point




Contours of a loss function for CIFAR-10 [1]

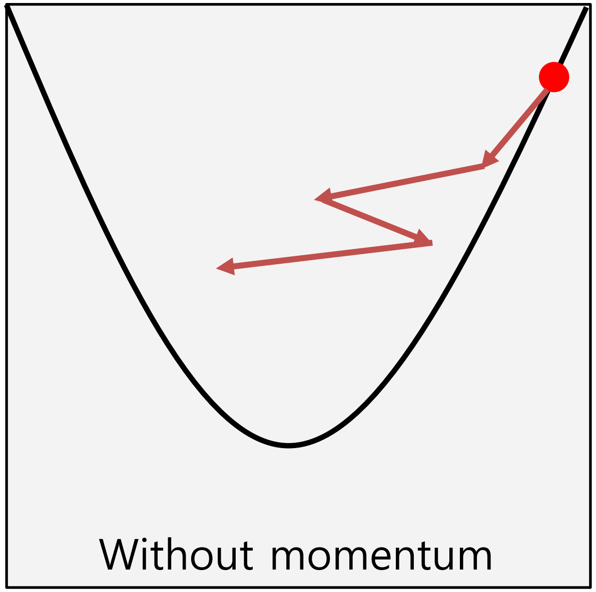
[1] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural Nets," in NIPS 2018.

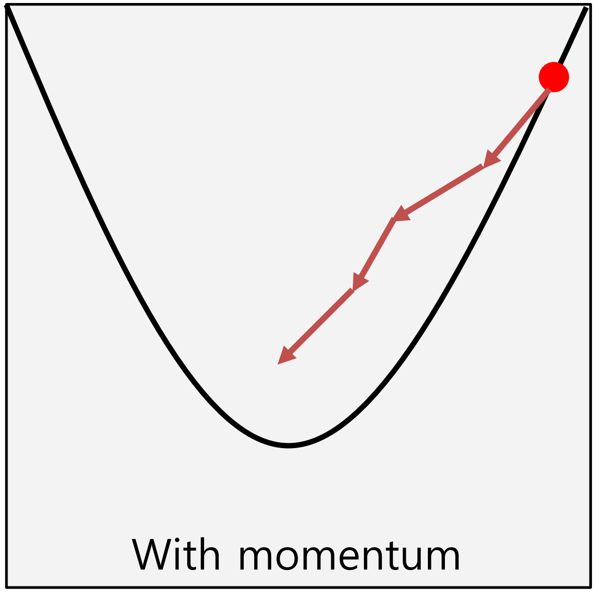
# Stochastic gradient descent

- Use randomness
  - Compute the gradient with an average of randomly sampled pairs
    - $J_{X_k} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{E}(X_k(i))$
    - $\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \eta \frac{\partial J_{X_k}}{\partial \mathbf{w}_j^r}$
    - Repeat the above process until convergence
      - $\frac{1}{N}\sum_{i=1}^{N} \mathcal{E}(i) \approx \frac{1}{m}\sum_{k=1}^{m} J_{X_k}$
  - Speed up learning
  - May overcome local minima & saddle point
    - Direction slightly different to the gradient from all pairs

# Momentum

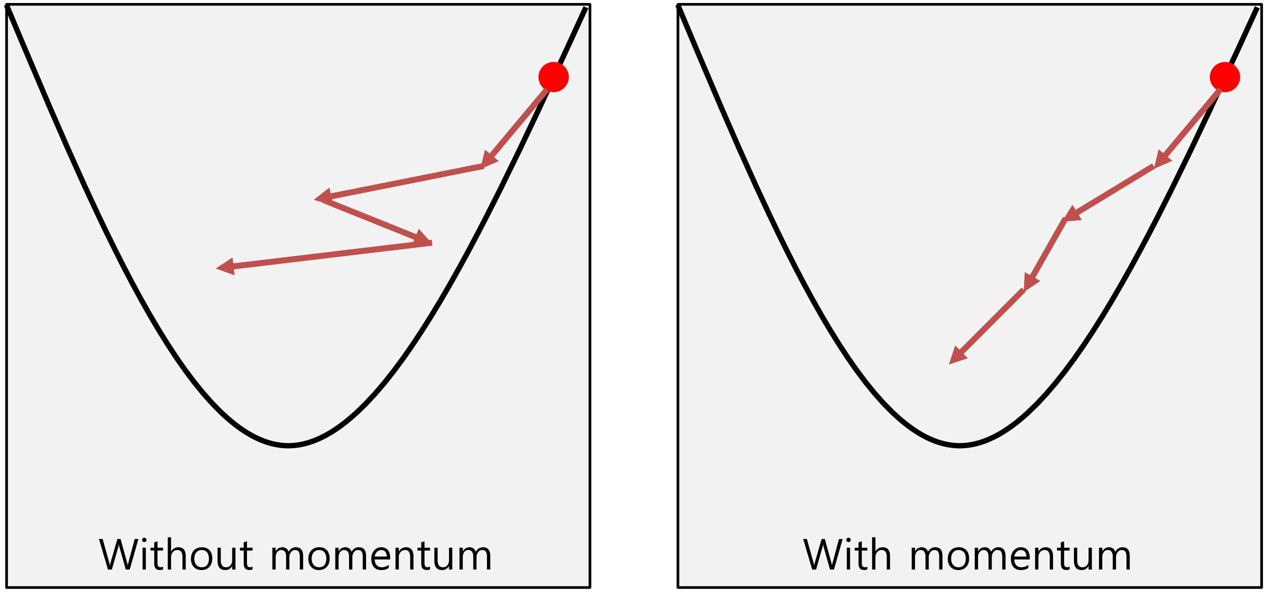
- Introduce velocity variables
  - Compute the gradient with an average of randomly sampled pairs
    - $J_{X_k} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{E}(X_k(i))$
    - $\mathbf{v}_j^r(\text{new}) = \mu \mathbf{v}_j^r(\text{old}) - \eta \frac{\partial J_{X_k}}{\partial \mathbf{w}_j^r}$
    - $\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) + \mathbf{v}_j^r(\text{new})$
    - Repeat the above process until convergence



# Momentum

- Introduce velocity variables
  - Effect over the iterations when $\mu = 0.9$
    - $\mathbf{v}_1 = -\boldsymbol{g}_1$
    - $\mathbf{v}_2 = -0.9\boldsymbol{g}_1 - \boldsymbol{g}_2$
    - $\mathbf{v}_3 = -0.9(0.9\boldsymbol{g}_1 - \boldsymbol{g}_2) - \boldsymbol{g}_3 = -0.81\boldsymbol{g}_1 - 0.9\boldsymbol{g}_2 - \boldsymbol{g}_3$



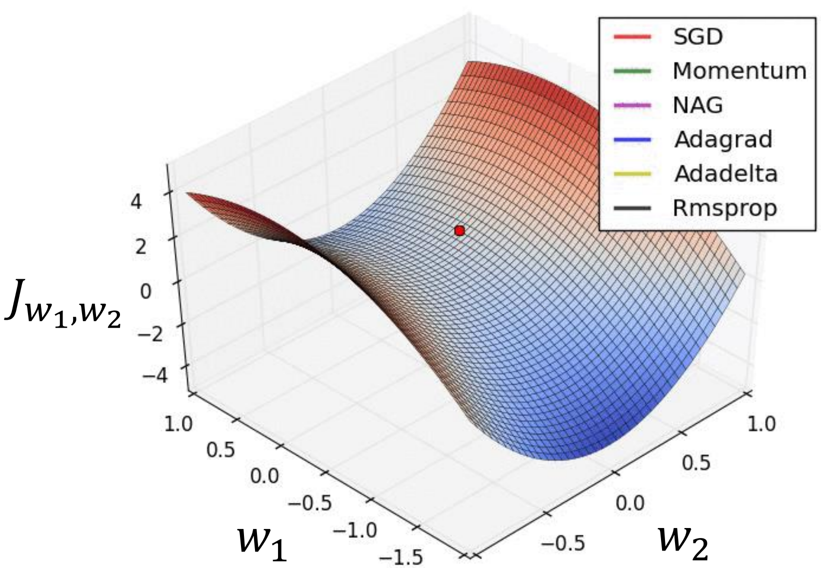
# Root Mean Square Propagation

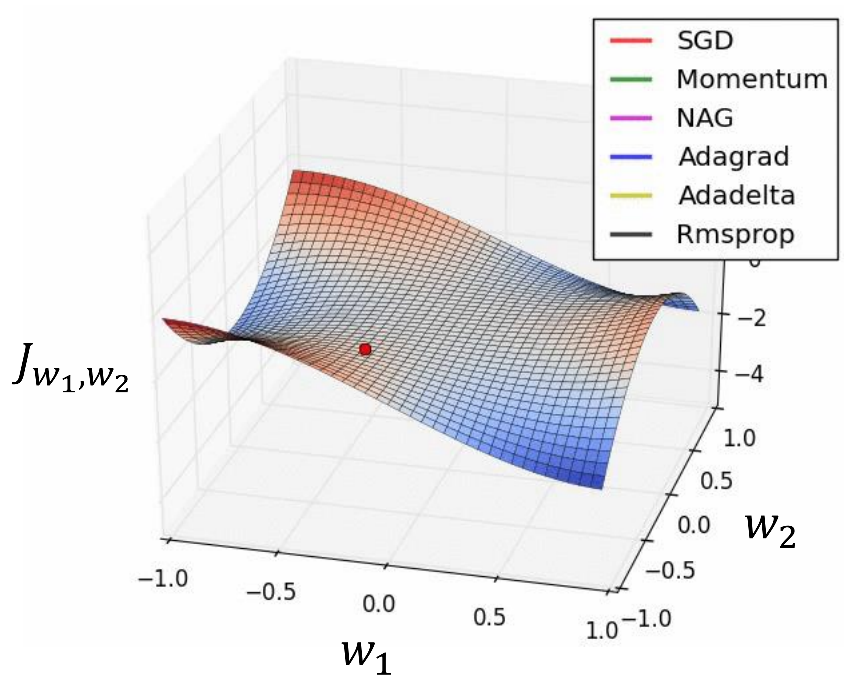
- Adaptive step size (learning rate) $\eta$
  - Chose a different step size $\eta$ for each weight
    - Increase $\eta$ when the accumulated magnitude of its gradients is small
    - Decrease $\eta$ when the accumulated magnitude of its gradient is large
  - Compute the gradient with an average of randomly sampled pairs
    - $J_{X_k} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{E}(X_k(i))$
    - $\mathbf{a}_j^r(\text{new}) = \alpha \mathbf{a}_j^r(\text{old}) - (1-\alpha)\left(\frac{\partial J_{X_k}}{\partial \mathbf{w}_j^r}\right)^2$
    - $\mathbf{w}_j^r(\text{new}) = \mathbf{w}_j^r(\text{old}) - \frac{\eta}{\sqrt{\mathbf{a}_j^r(\text{new})+\epsilon}} \frac{\partial J_{X_k}}{\partial \mathbf{w}_j^r}$

# Update weights

- Optimizer
  - Base : SGD
  - Direction : Momentum, NAG
  - Step size : Adagrad, RMSProp, AdaDelta
  - Both : Adam, etc.

Contour of a loss function



# CONVOLUTIONAL NEURAL NETWORKS

# Multi-layer perceptron (MLP)

- MLP for classifying the handwritten digit data
  - 784 input nodes for each image of $28 \times 28$ pixels

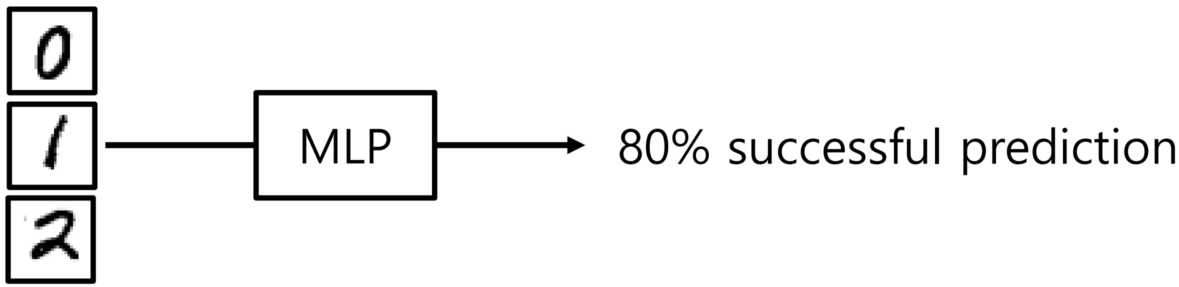
=

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 12, 0, 11, 39, 137, 37, 0, 152, 147, 84, 0, 0, 0, 0,
0, 1, 0, 0, 0, 41, 160, 250, 255, 235, 162, 255, 238, 206, 11, 13, 0, 0, 0, 0, 16, 9, 9, 150, 251, 45, 21, 184, 159, 154, 2
55, 233, 40, 0, 0, 10, 0, 0, 0, 0, 0, 145, 146, 3, 10, 0, 11, 124, 253, 255, 107, 0, 0, 0, 0, 3, 0, 4, 15, 236, 216, 0, 0,
38, 109, 247, 240, 169, 0, 11, 0, 1, 0, 2, 0, 0, 0, 253, 253, 23, 62, 224, 241, 255, 164, 0, 5, 0, 0, 6, 0, 0, 4, 0, 3, 252
, 250, 228, 255, 255, 234, 112, 28, 0, 2, 17, 0, 0, 2, 1, 4, 0, 21, 255, 253, 251, 255, 172, 31, 8, 0, 1, 0, 0, 0, 0, 0, 4,
 0, 163, 225, 251, 255, 229, 120, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 21, 162, 255, 255, 254, 255, 126, 6, 0, 10, 14, 6, 0, 0, 9
, 0, 3, 79, 242, 255, 141, 66, 255, 245, 189, 7, 8, 0, 0, 5, 0, 0, 0, 0, 26, 221, 237, 98, 0, 67, 251, 255, 144, 0, 8, 0, 0
, 7, 0, 0, 11, 0, 125, 255, 141, 0, 87, 244, 255, 208, 3, 0, 0, 13, 0, 1, 0, 1, 0, 0, 145, 248, 228, 116, 235, 255, 141, 34
, 0, 11, 0, 1, 0, 0, 0, 1, 3, 0, 85, 237, 253, 246, 255, 210, 21, 1, 0, 1, 0, 0, 6, 2, 4, 0, 0, 0, 6, 23, 112, 157, 114, 32
, 0, 0, 0, 0, 2, 0, 8, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

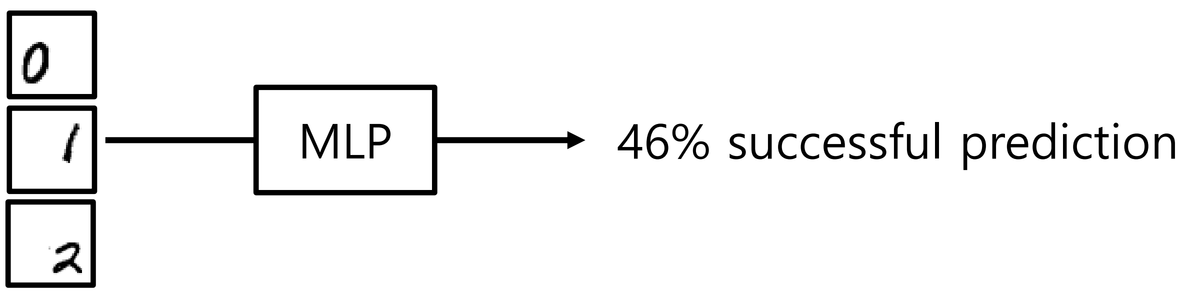
MNIST

# Multi-layer perceptron (MLP)

- MLP for classifying the handwritten digit data
  - 784 input nodes for each image of $28 \times 28$ pixels
  - Same as training set

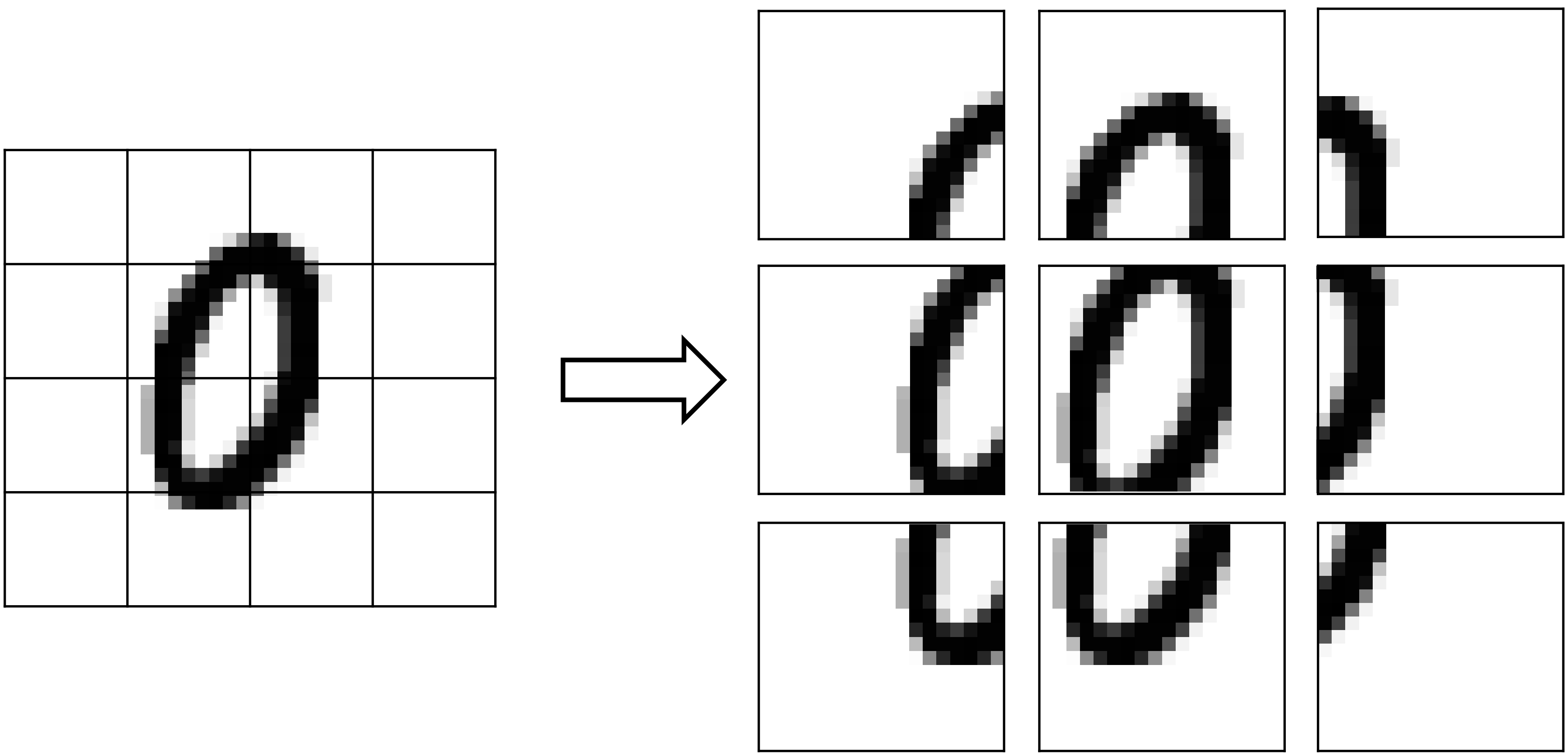



  - Slightly changes



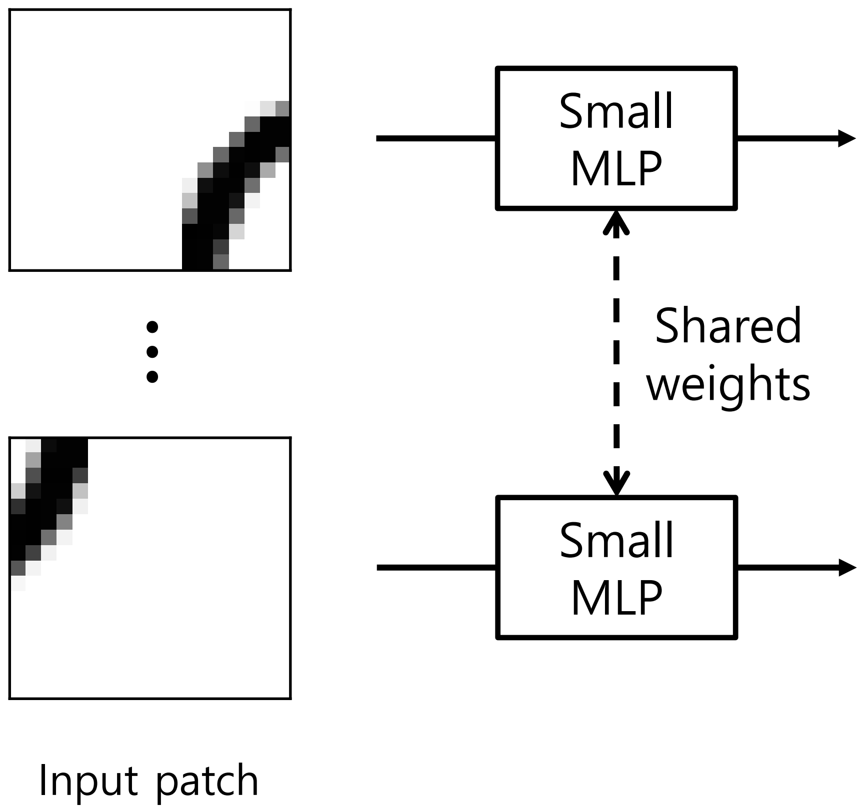
# Convolutional neural networks

- Design a network for translation invariance
  - Split into overlapping patches



# Convolutional neural networks

- Design a network for translation invariance
  - Feedforward each patch into a small MLP
  - Repeat for all patches

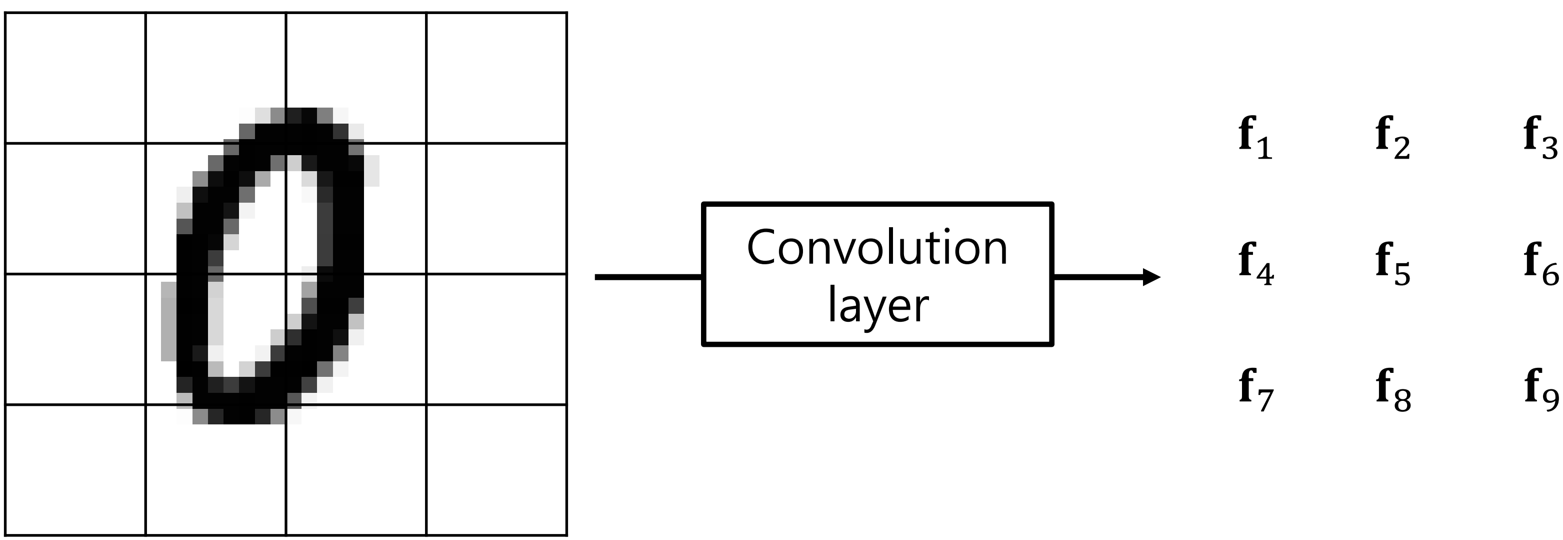



$\mathbf{f}_1 = [f_{11}, f_{12}, \dots, f_{1d}]^T$

$\mathbf{f}_9 = [f_{91}, f_{92}, \dots, f_{9d}]^T$

# Convolutional neural networks

- Design a network for translation invariance
  - ~~Feedforward each patch into a small MLP~~
  - ~~Repeat for all patches~~
  - Feedforward an image into convolution layer
  - Yield the hidden neurons (or feature map)




Input

Output
(feature map)

# Convolutional neural networks

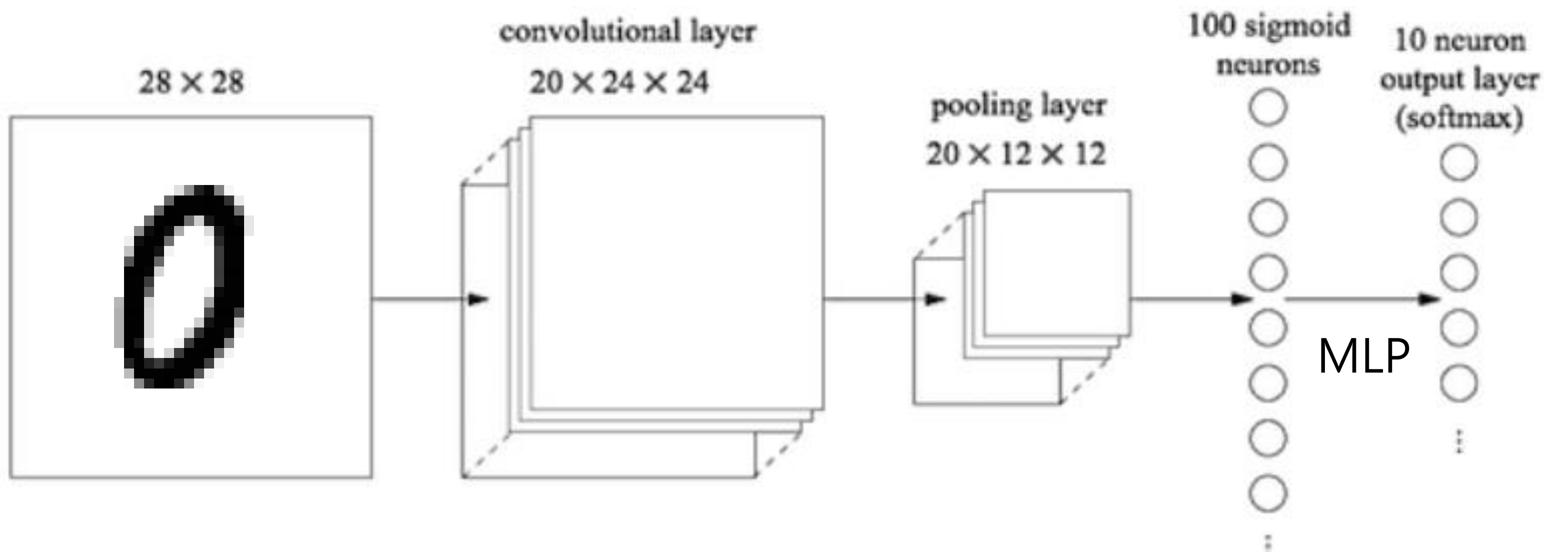
- Design a network for translation invariance
  - Simplify the information in the feature map
  - Yield the condensed feature map
  - Type of pooling
    - Max, average, *etc.*

$$\begin{matrix} \mathbf{f}_1 & \mathbf{f}_2 & \mathbf{f}_3 \\ \mathbf{f}_4 & \mathbf{f}_5 & \mathbf{f}_6 \\ \mathbf{f}_7 & \mathbf{f}_8 & \mathbf{f}_9 \end{matrix} \longrightarrow \boxed{\text{Max pooling layer}} \longrightarrow \begin{matrix} \max(\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_4, \mathbf{f}_5) & \max(\mathbf{f}_2, \mathbf{f}_3, \mathbf{f}_5, \mathbf{f}_6) \\ \max(\mathbf{f}_4, \mathbf{f}_5, \mathbf{f}_7, \mathbf{f}_8) & \max(\mathbf{f}_5, \mathbf{f}_6, \mathbf{f}_8, \mathbf{f}_9) \end{matrix}$$

Input (feature map) — Output

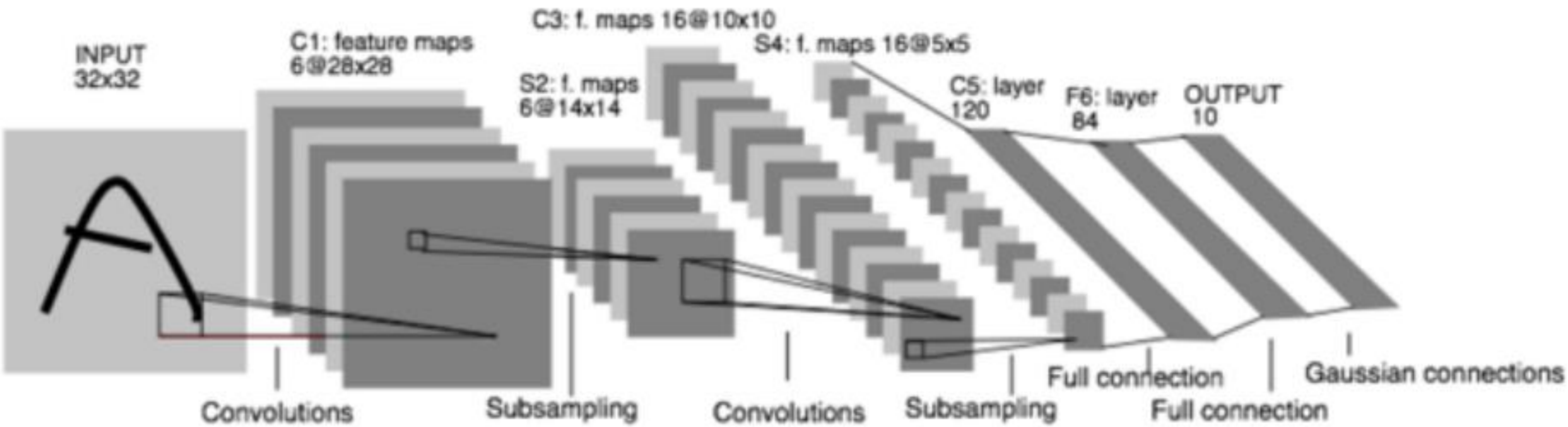
# Convolutional neural networks

- Design a network for translation invariance
  - Use the feature map for classification



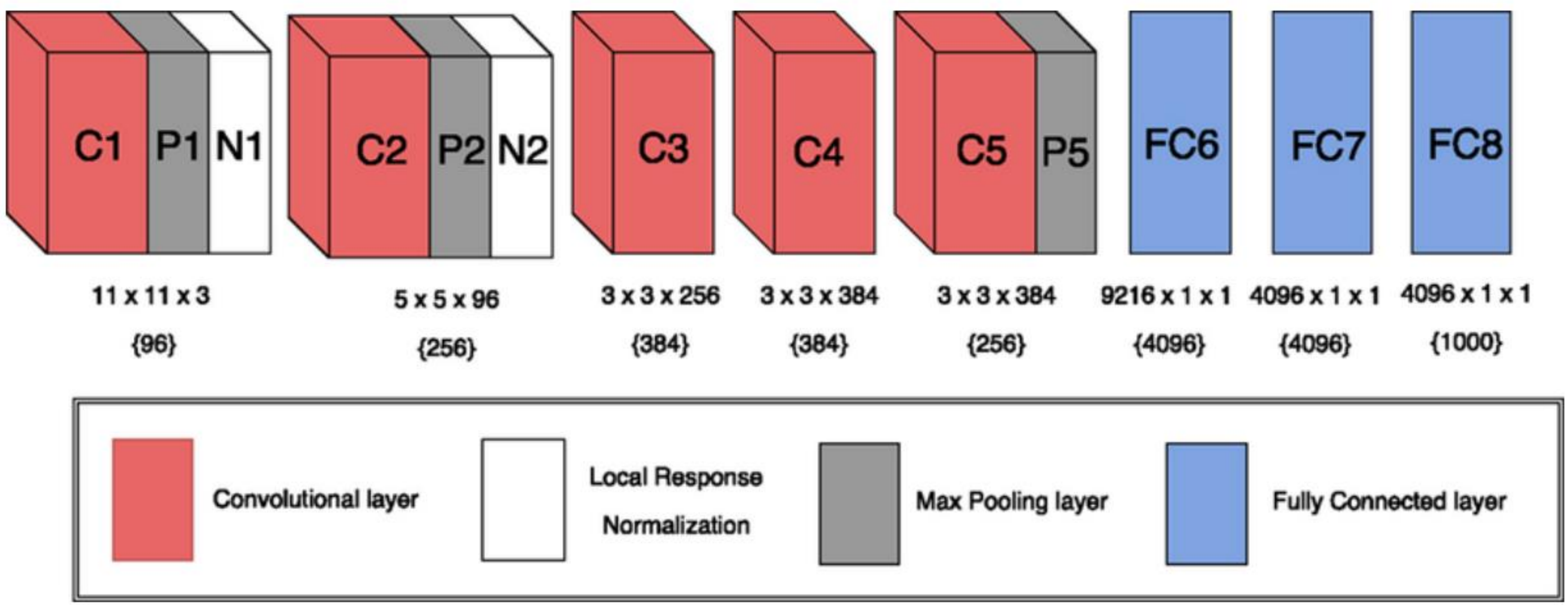
# Summary of CNN history

- LeNet
  - It was developed in 1990's
  - Use for digits
  - Outperformed many other existing algorithms



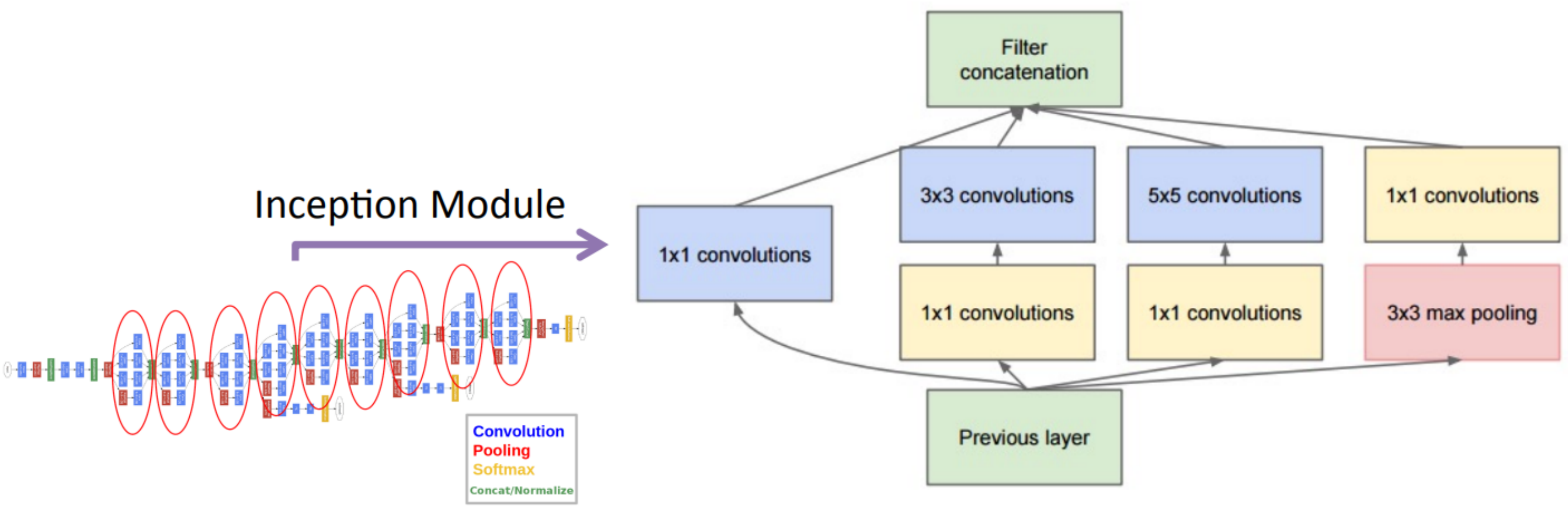
# Summary of CNN history

- AlexNet
  - It significantly outperformed
  - 2012 ImageNet challenge (ILSVRC) winner
  - Similar to LeNet, but deeper, and bigger



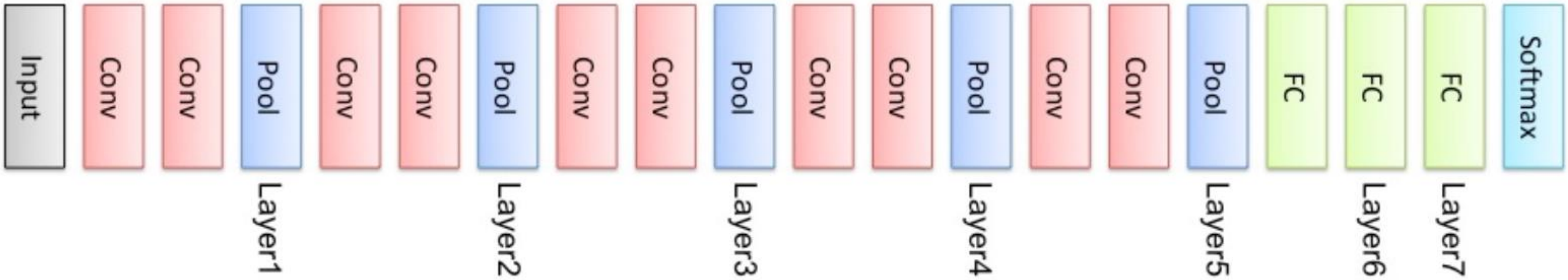
# Summary of CNN history

- GoogLeNet
  - Proposed the inception module
  - It reduced the number of parameters
    - GoogLeNet (4M, top-5 error rate of 6.67%)
    - AlexNet (60M, top-5 error rate of 15.3%)
  - Several follow-up versions were proposed (Inception-v4)



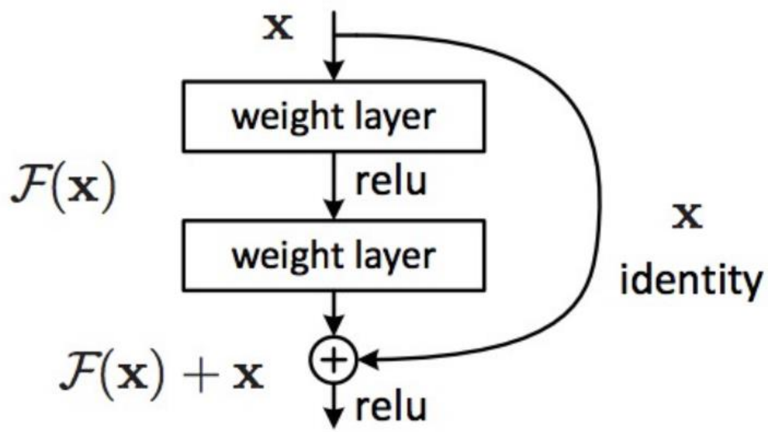
# Summary of CNN history

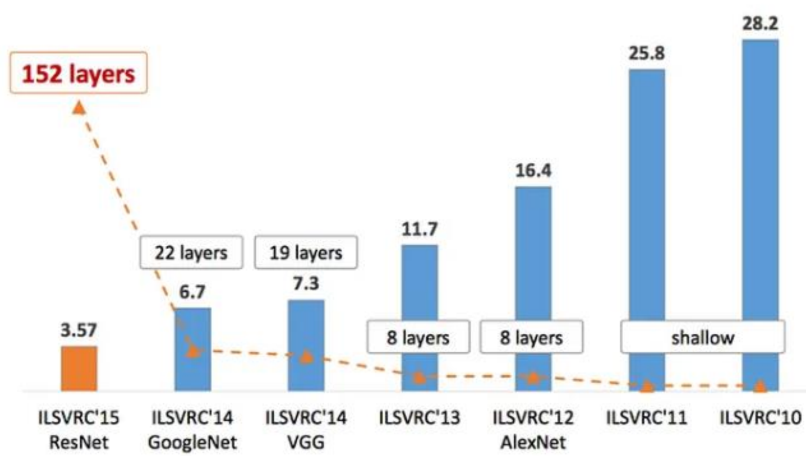
- VGGNet
  - Use only $3 \times 3$ convolutions and $2 \times 2$ pooling
    - Uniform architecture
    - Currently the most preferred structure
    - Use a baseline feature extractor
  - Showed that depth of the network is critical component



# Summary of CNN history

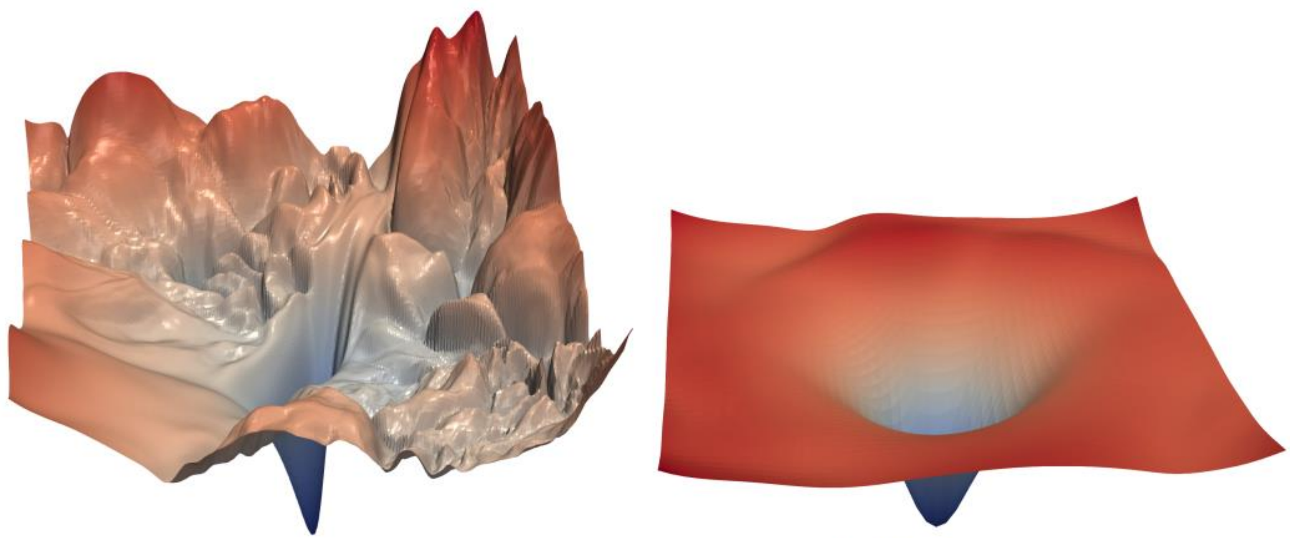
- ResNet
  - Proposed skip connections
  - Train a network with 152 layers successfully
  - Top-5 error rate of 3.57% at the ILSVRC 2015
    - Human-level performance



# Summary of CNN history

- ResNet
  - Proposed skip connections
  - Train a network with 152 layers successfully
  - Top-5 error rate of 3.57% at the ILSVRC 2015
    - Human-level performance



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.