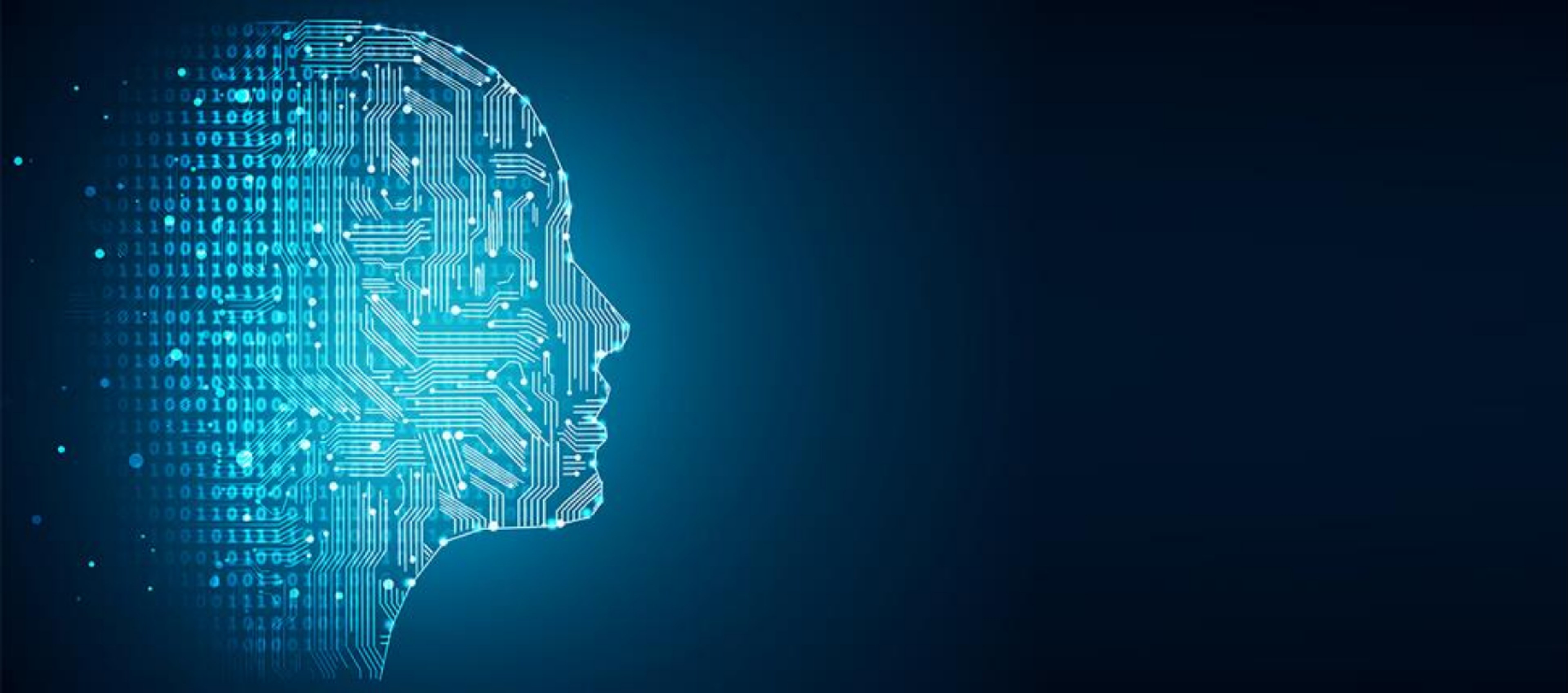


Concepts in Artificial Intelligence & Machine Learning Technologies

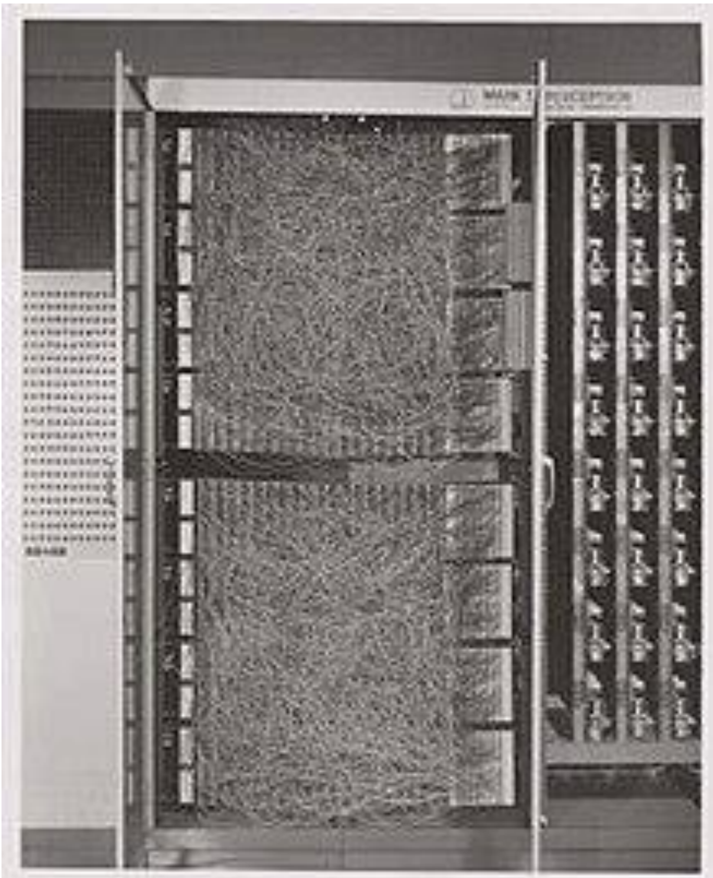
Deep Learning Basis

By Dr. Hu Wang, Dr. Wei Zhang

Deep Learning

Perceptron Learning Algorithm

- Perceptron



The perceptron algorithm was invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt

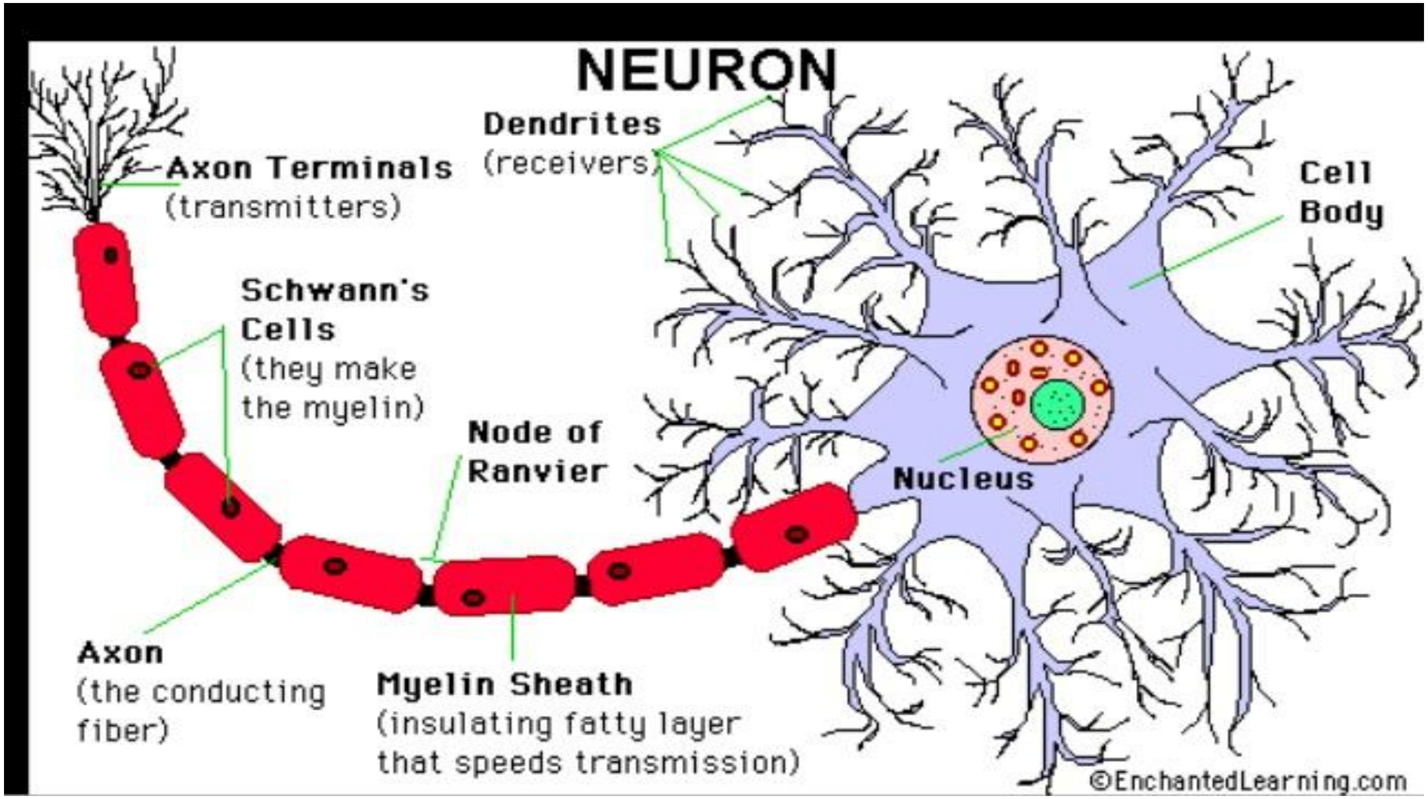
The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704.

This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.

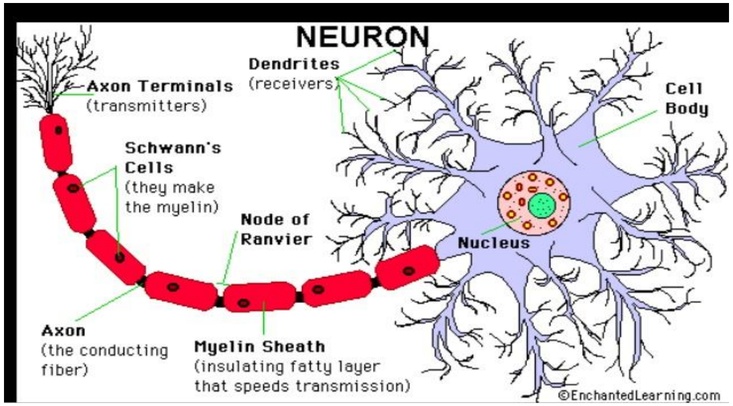
The New York Times reported the perceptron to be "the embryo of an electronic computer that expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."

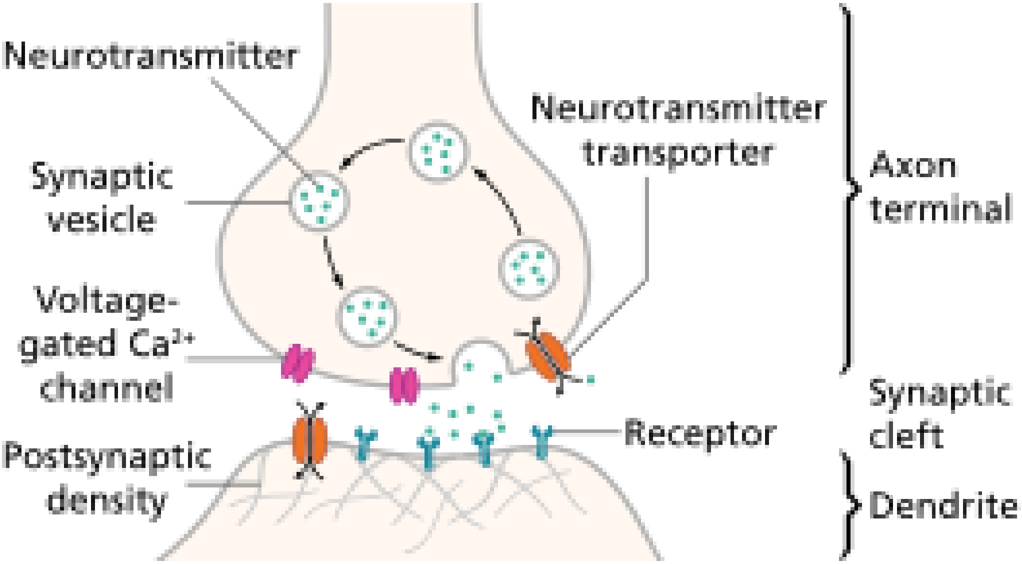
https://en.wikipedia.org/wiki/Perceptron

- Perceptron --- prototype

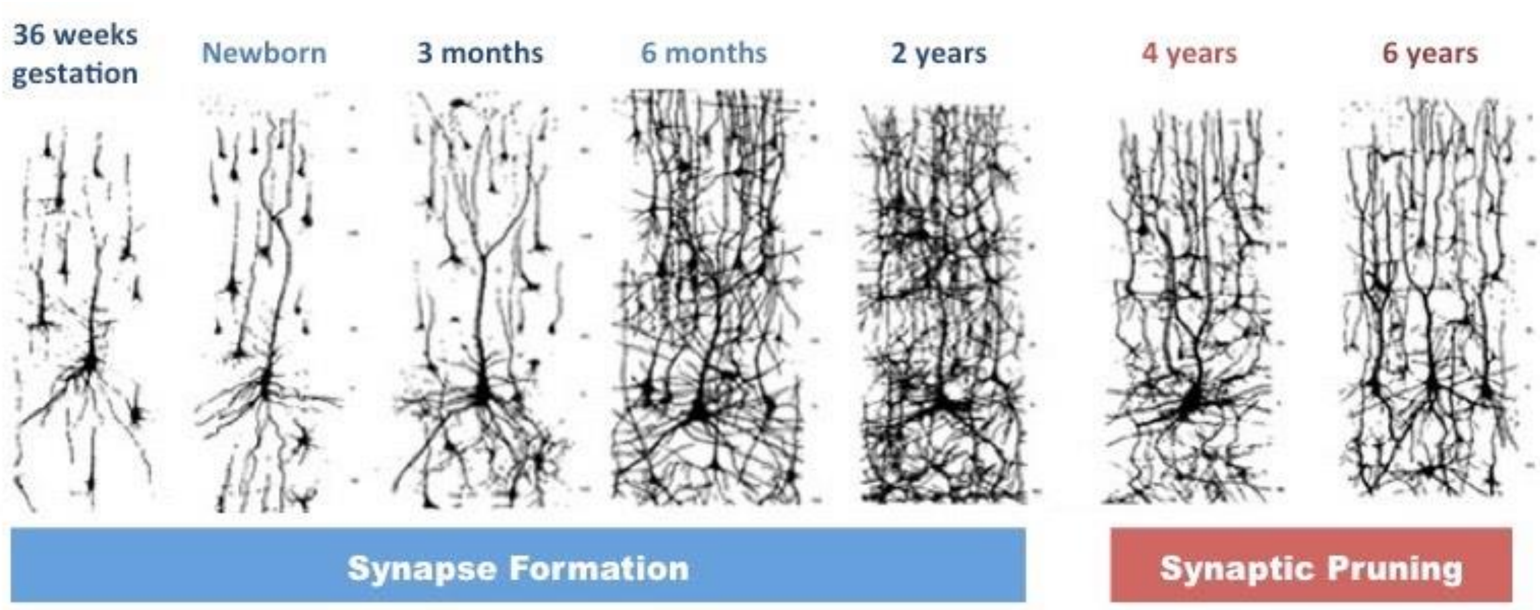


- Perceptron --- prototype

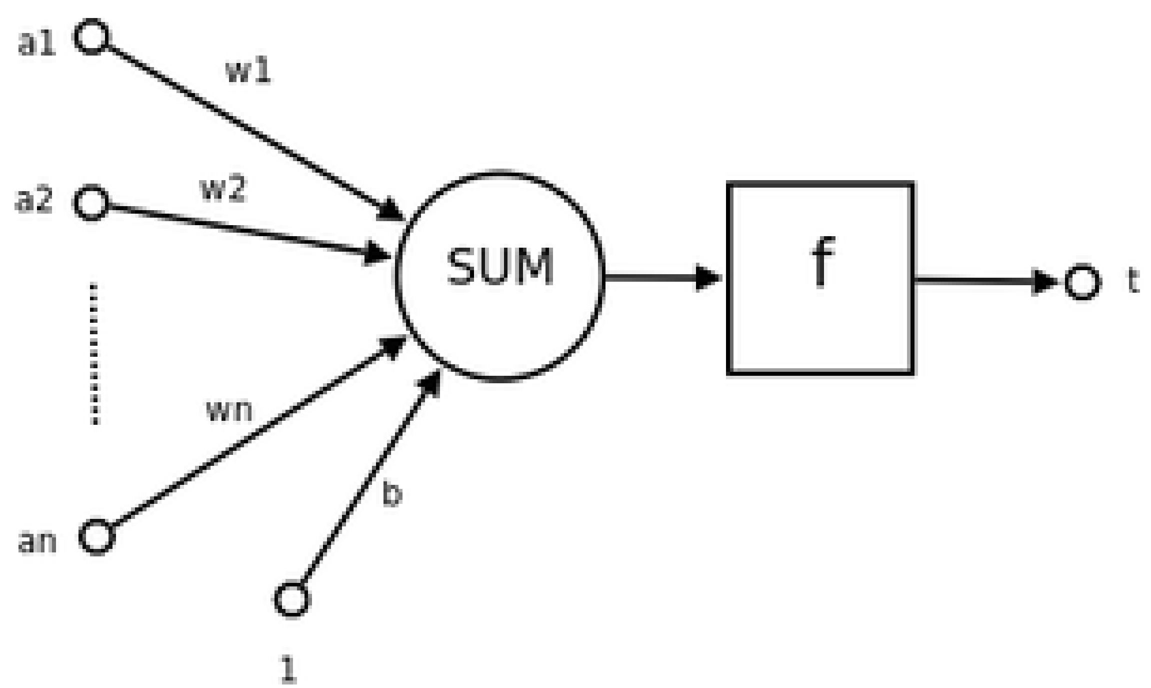


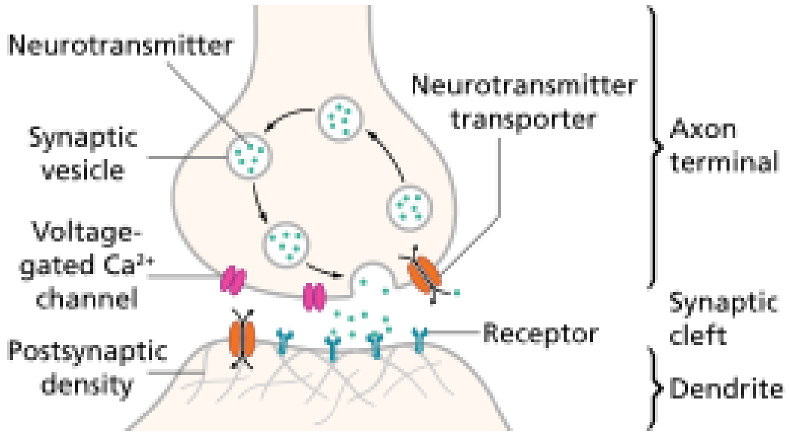


- Perceptron --- prototype

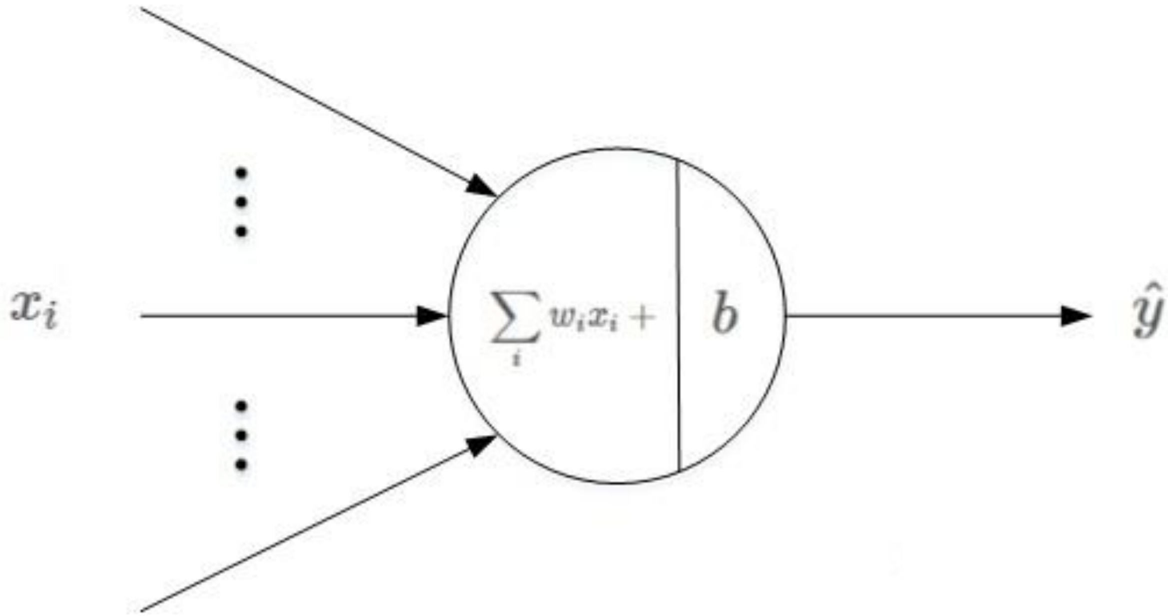


- Perceptron --- prototype





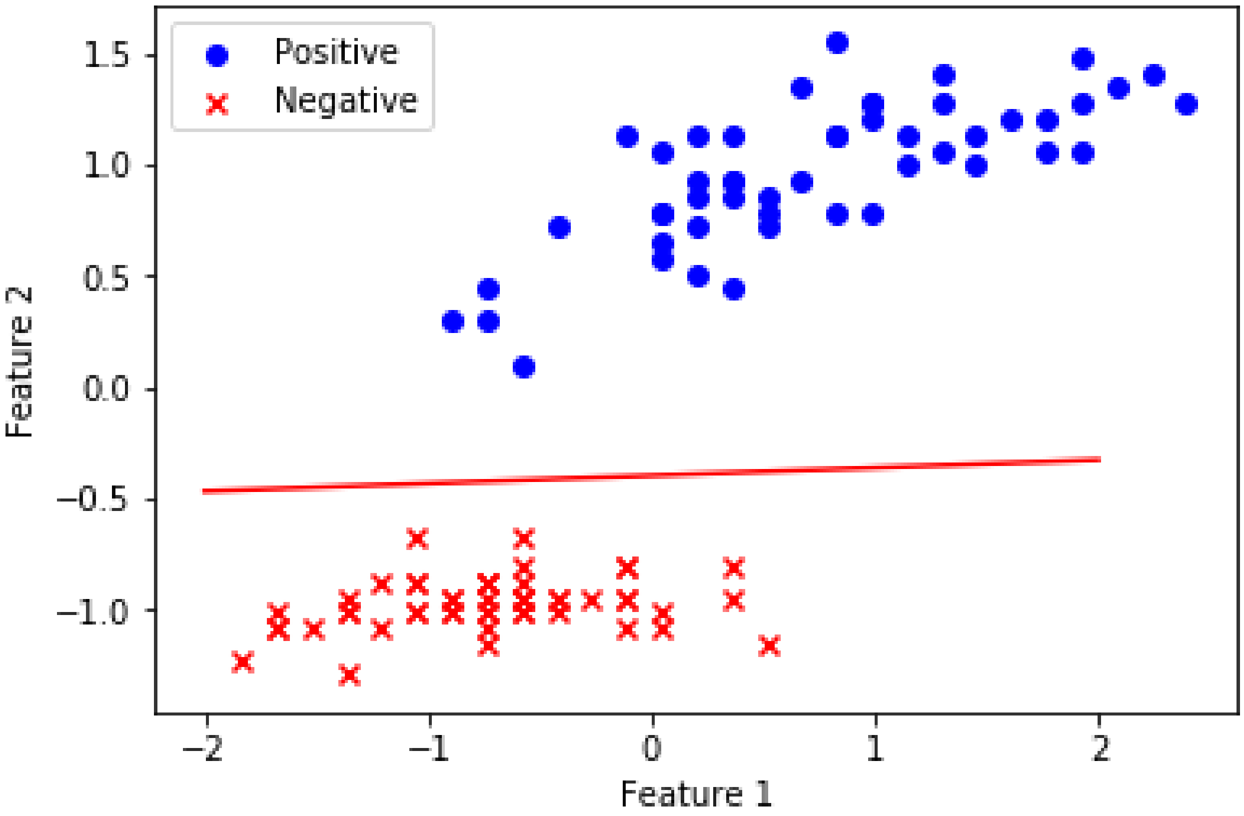
- Perceptron Learning Algorithm



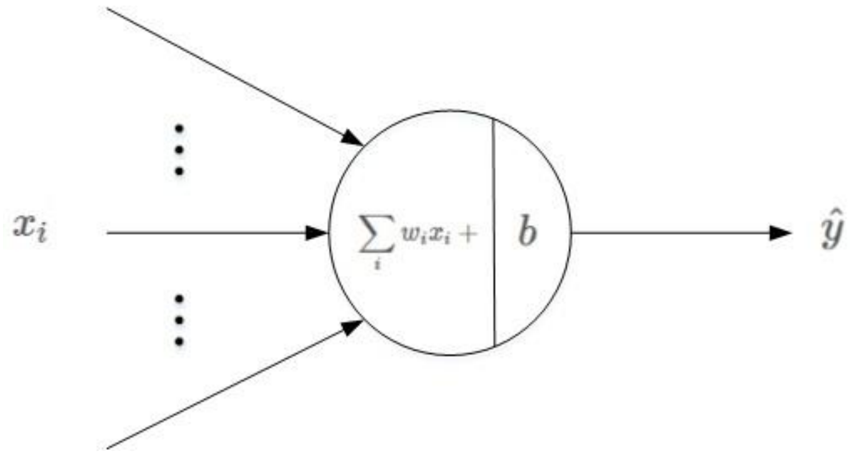
where x_i is the input, w_i is the weights, and b is the bias

$$scores = \sum_i^N w_i x_i + b$$

- Perceptron Learning Algorithm --- example



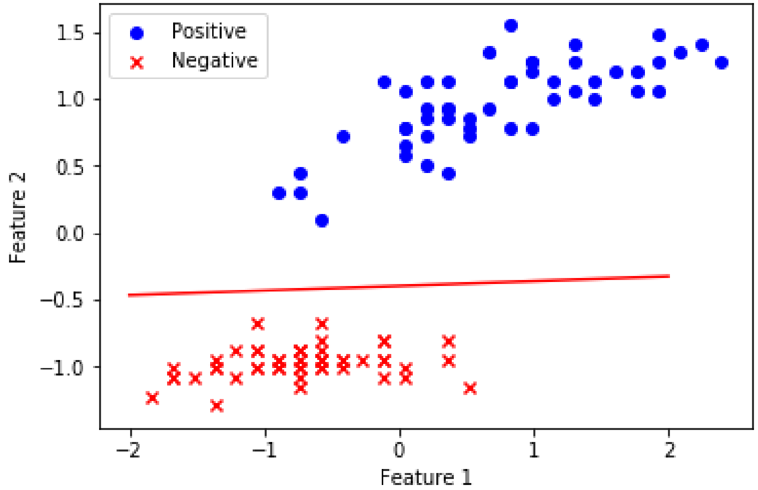
- Perceptron Learning Algorithm



$$scores = \sum_{i}^{N} w_i x_i + b$$

When scores >= 0, y_pred = 1
When scores < 0, y_pred = -1

- ## Perceptron Learning Algorithm



$$scores = \sum_{i}^{N} w_i x_i + b$$

For binary classification problems, the PLA model can be used. The basic principle of PLA is to correct point by point.

Firstly, randomly select a classification surface on the hyperplane and count the error points; then randomly correct a certain error point, that is, change the position of the straight line, so that the error point can be corrected; Then randomly select an error point to correct ...

The classification surface keeps changing until all the points are classified correctly, and the best classification surface is obtained.

- # Perceptron Learning Algorithm --- case 1

$$scores = \sum_{i}^{N} w_i x_i + b$$

Incorrectly classify a positive sample (y=1) as a negative sample (y=-1)

- When scores >= 0, y_pred = 1
- When scores < 0,   y_pred = -1

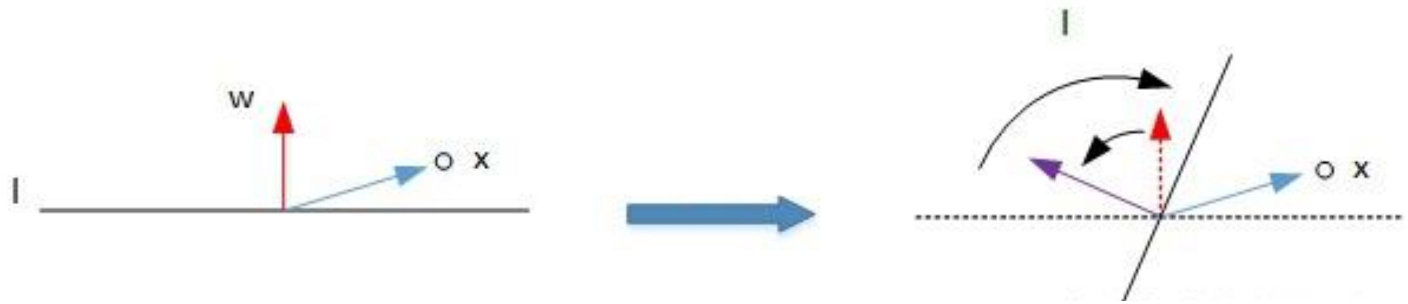
At this time **wx**<0 (it should be >=0), that is, the angle between **w** and **x** is greater than 90 degrees. They are on two sides of the classification boundary.

The correction method is to make the included angle smaller and correct the **w** value so that the two are on the same side of the straight line

$$\mathbf{w} := \mathbf{w} + \mathbf{x} = \mathbf{w} + y\mathbf{x}$$

- ## Perceptron Learning Algorithm --- case 2

$$scores = \sum_{i}^{N} w_i x_i + b$$



Incorrectly classify a negative sample (y=-1) as a positive sample (y=1)

- When scores >= 0, y_pred = 1
- When scores < 0,   y_pred = -1

At this time **wx**>0, that is, the angle between **w** and **x** is smaller than 90 degrees. They are on the same side of the classification boundary.

The correction method is to make the included angle larger.

$$\mathbf{w} := \mathbf{w} - \mathbf{x} = \mathbf{w} + y\mathbf{x}$$

Perceptron Learning Algorithm --- Unite

$$w := w + x = w + yx$$

Incorrectly classify a positive sample (y=1) as a negative sample (y=-1)

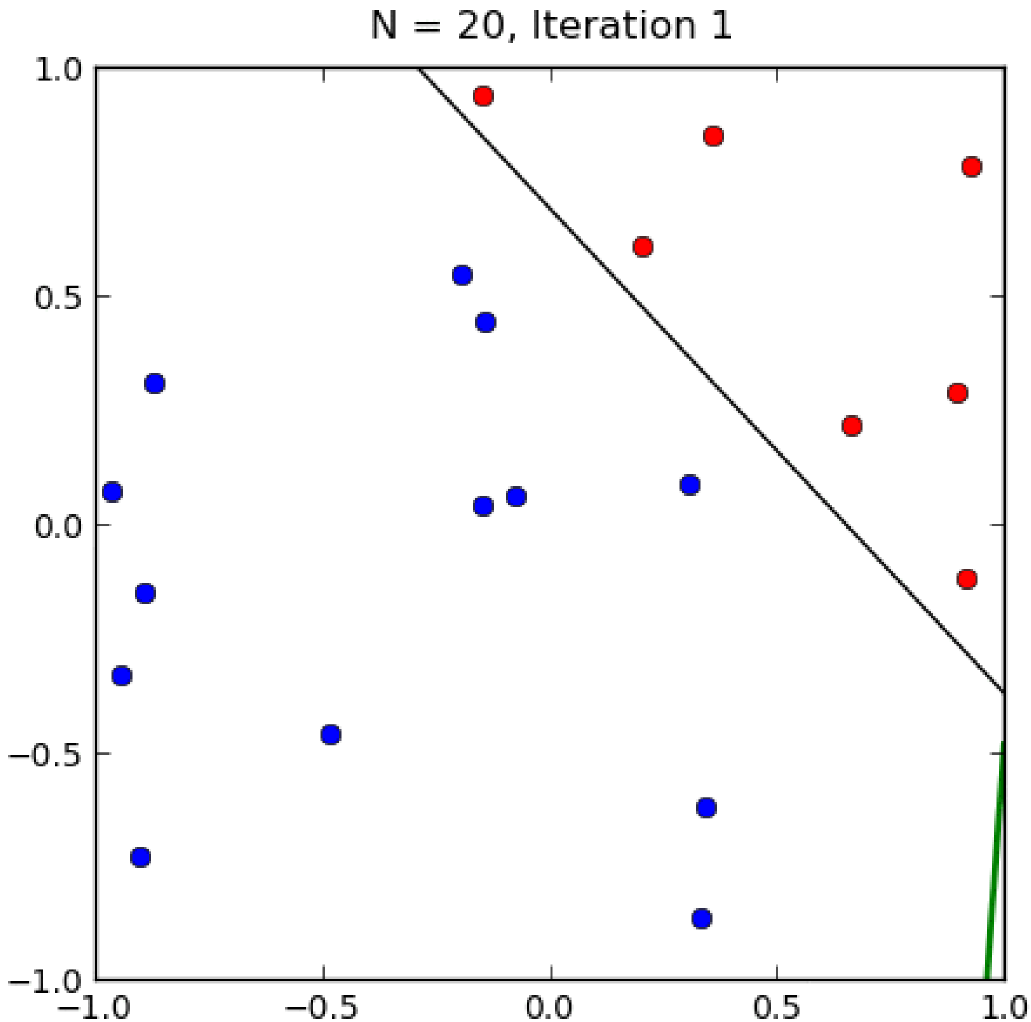
$$w := w - x = w + yx$$

Incorrectly classify a negative sample (y=-1) as a positive sample (y=1)

After analyzing two cases, we found that the w update expression of PLA is the same every time.

$$w := w + yx$$

- Perceptron Learning Algorithm



Programming Example

Demo

Input data

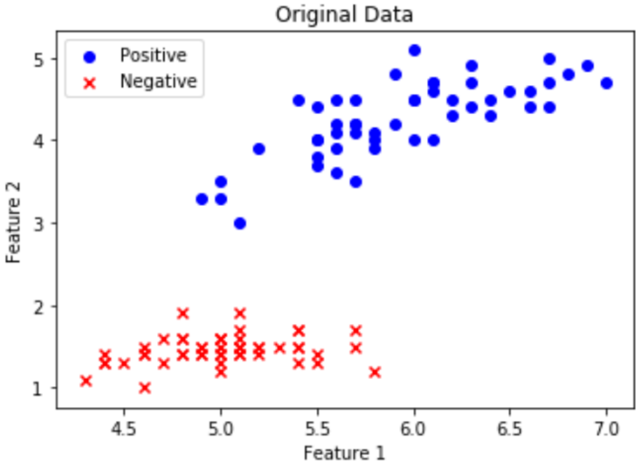
In [1]:
```
import numpy as np
import pandas as pd

data = pd.read_csv('./data1.csv', header=None)
# input samples, dim (100, 2)
X = data.iloc[:,:2].values
# output samples, dim (100, )
y = data.iloc[:,2].values
```

Data visualization

In [3]:
```
import matplotlib.pyplot as plt

plt.scatter(X[:50, 0], X[:50, 1], color='blue', marker='o', label='Positive')
plt.scatter(X[50:, 0], X[50:, 1], color='red', marker='x', label='Negative')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc = 'upper left')
plt.title('Original Data')
plt.show()
```



PLA algorithm

Feature normalization

First, normalize the two features separately

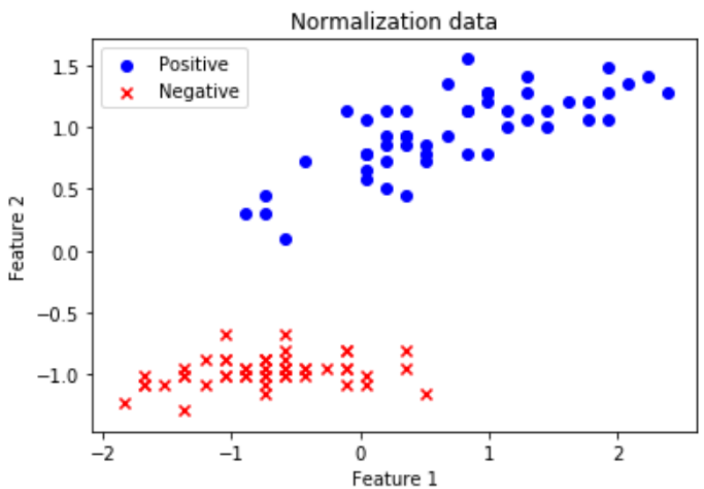
$$X = \frac{X - \mu}{\sigma}$$

Among them, μ is the feature mean, and σ is the feature standard deviation.

In [4]:
```
# Mean
u = np.mean(X, axis=0)
# standard deviation
v = np.std(X, axis=0)

X = (X - u) / v

# draw
plt.scatter(X[:50, 0], X[:50, 1], color='blue', marker='o', label='Positive')
plt.scatter(X[50:, 0], X[50:, 1], color='red', marker='x', label='Negative')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc = 'upper left')
plt.title('Normalization data')
plt.show()
```

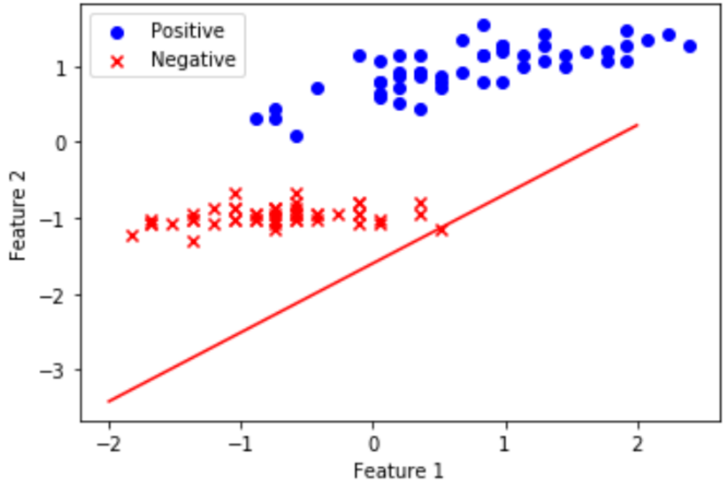


Classification Boundary init

In [5]:
```
# X + offset
X = np.hstack((np.ones((X.shape[0],1)), X))
# weight init
w = np.random.randn(3,1)
```

Display initial line position:

In [6]:
```
# First coordinate (x1, y1)
x1 = -2
y1 = -1 / w[2] * (w[0] * 1 + w[1] * x1)
# Second coordinate (x2, y2)
x2 = 2
y2 = -1 / w[2] * (w[0] * 1 + w[1] * x2)
# draw
plt.scatter(X[:50, 1], X[:50, 2], color='blue', marker='o', label='Positive')
plt.scatter(X[50:, 1], X[50:, 2], color='red', marker='x', label='Negative')
plt.plot([x1,x2], [y1,y2],'r')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc = 'upper left')
plt.show()
```



Calculate scores, update weights

In [7]:
```
s = np.dot(X, w)
y_pred = np.ones_like(y)    # predict the output
loc_n = np.where(s < 0)[0]
y_pred[loc_n] = -1
```

Next, select one of the misclassified samples and use PLA to update the weight coefficient w.

In [8]:
```
# The first error point
t = np.where(y != y_pred)[0][0]
# update weights w
w += y[t] * X[t, :].reshape((3,1))
```

Iterative update training

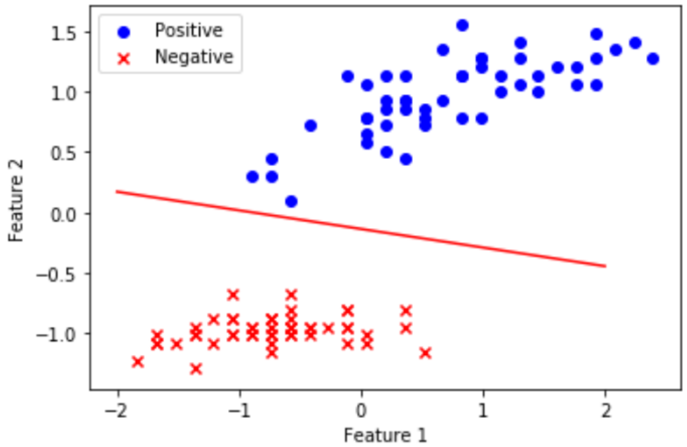
Updating the weight w is an iterative process. As long as there are misclassified samples, it will continue to update until all samples are classified correctly. (Note that the premise is that the positive and negative samples are completely separable)

In [9]:
```
for i in range(100):
    s = np.dot(X, w)
    y_pred = np.ones_like(y)
    loc_n = np.where(s < 0)[0]
    y_pred[loc_n] = -1
    num_fault = len(np.where(y != y_pred)[0])
    print('Update time %2d, error points: %2d' % (i, num_fault))
    if num_fault == 0:
        break
    else:
        t = np.where(y != y_pred)[0][0]
        w += y[t] * X[t, :].reshape((3,1))
```

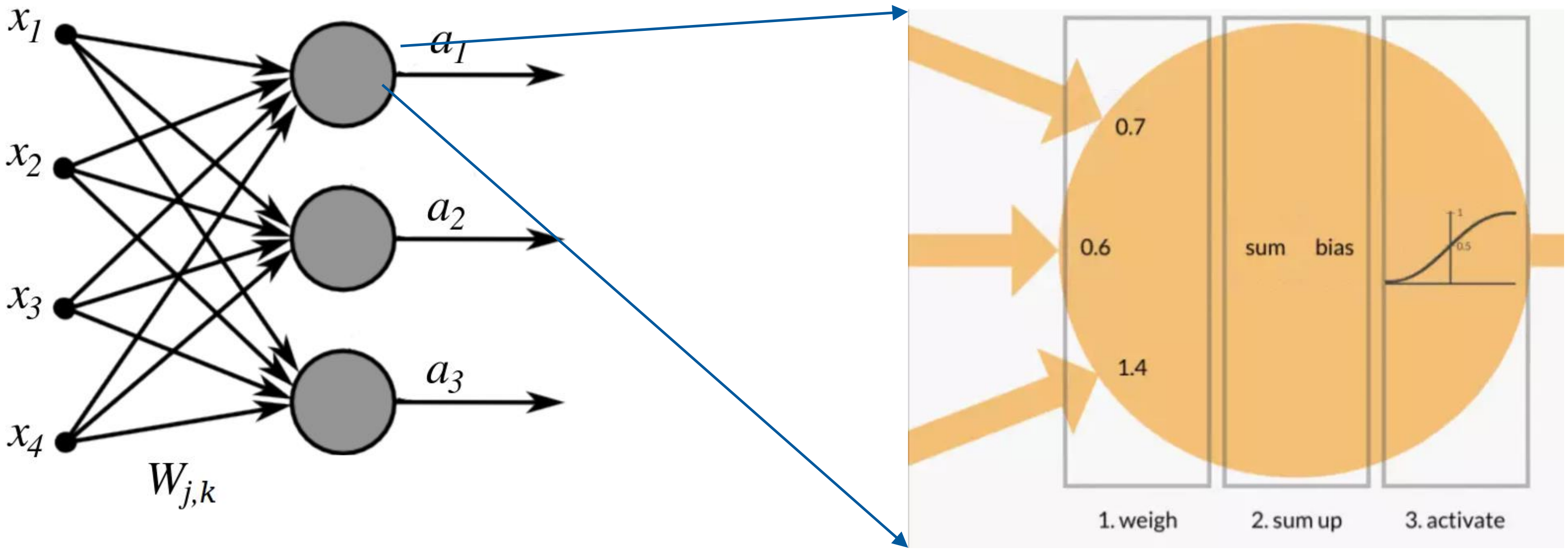
```
Update time  0, error points: 11
Update time  1, error points:  0
```

Draw result

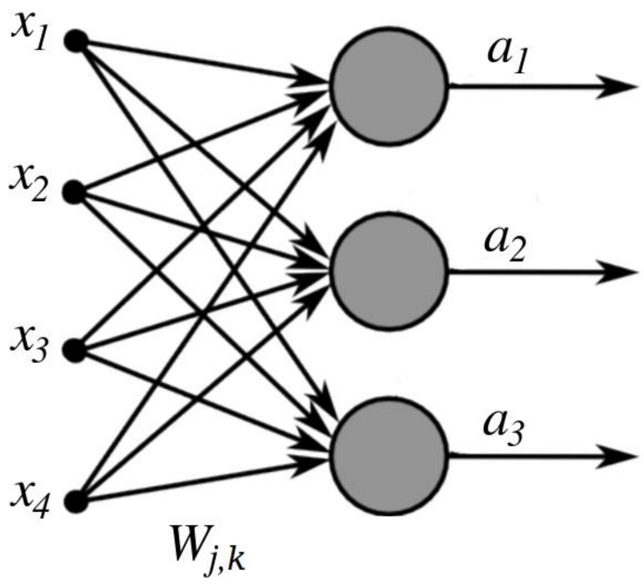
In [10]:
```
# First coordinate (x1, y1)
x1 = -2
y1 = -1 / w[2] * (w[0] * 1 + w[1] * x1)
# Second coordinate (x2, y2)
x2 = 2
y2 = -1 / w[2] * (w[0] * 1 + w[1] * x2)
# draw
plt.scatter(X[:50, 1], X[:50, 2], color='blue', marker='o', label='Positive')
plt.scatter(X[50:, 1], X[50:, 2], color='red', marker='x', label='Negative')
plt.plot([x1,x2], [y1,y2],'r')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(loc = 'upper left')
plt.show()
```

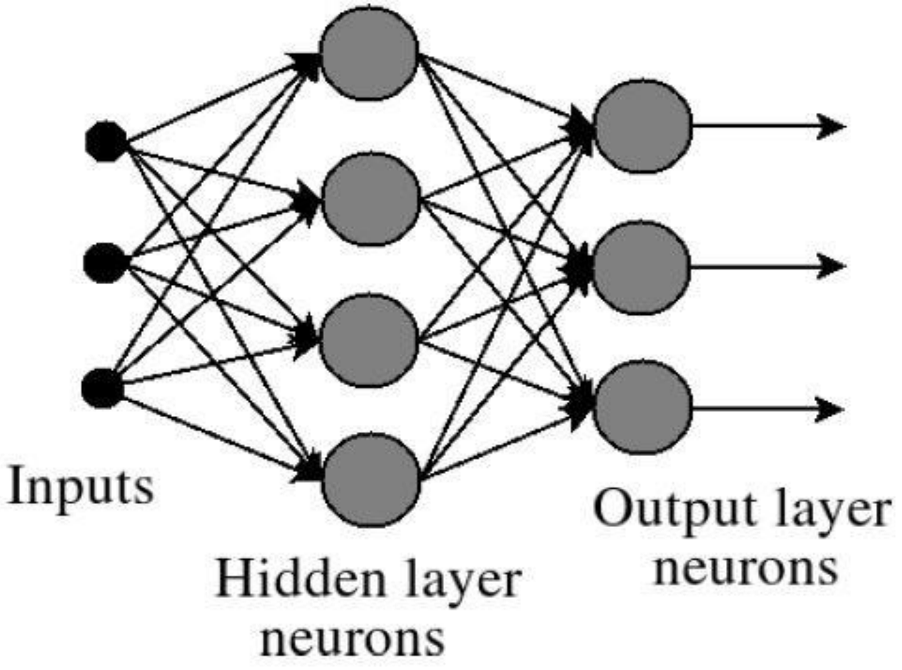


Perceptron -> Multi-layer Perceptron



Multi-layer Perceptron - A Non-linear Classifier

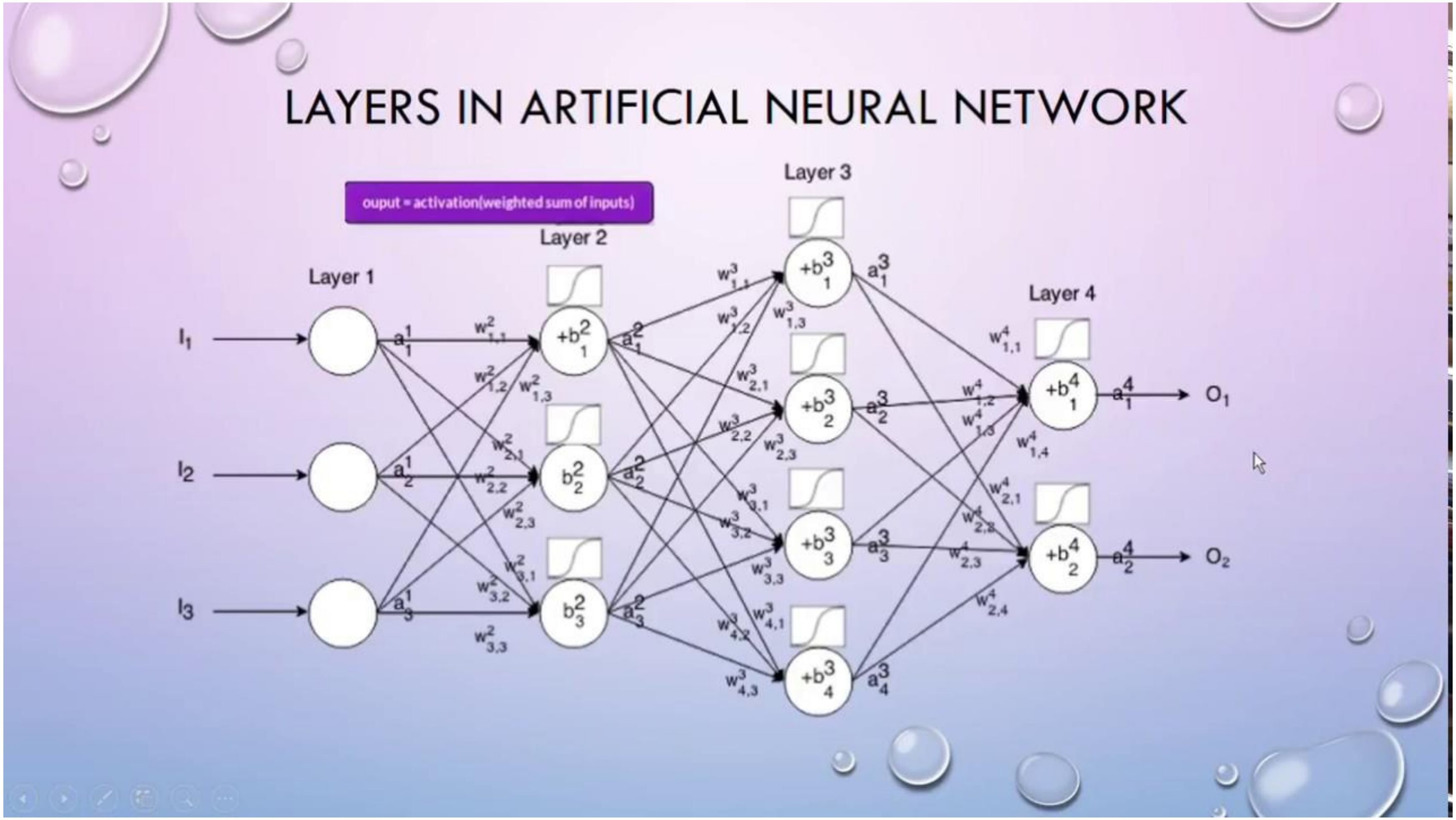




Perceptrons **MLP**

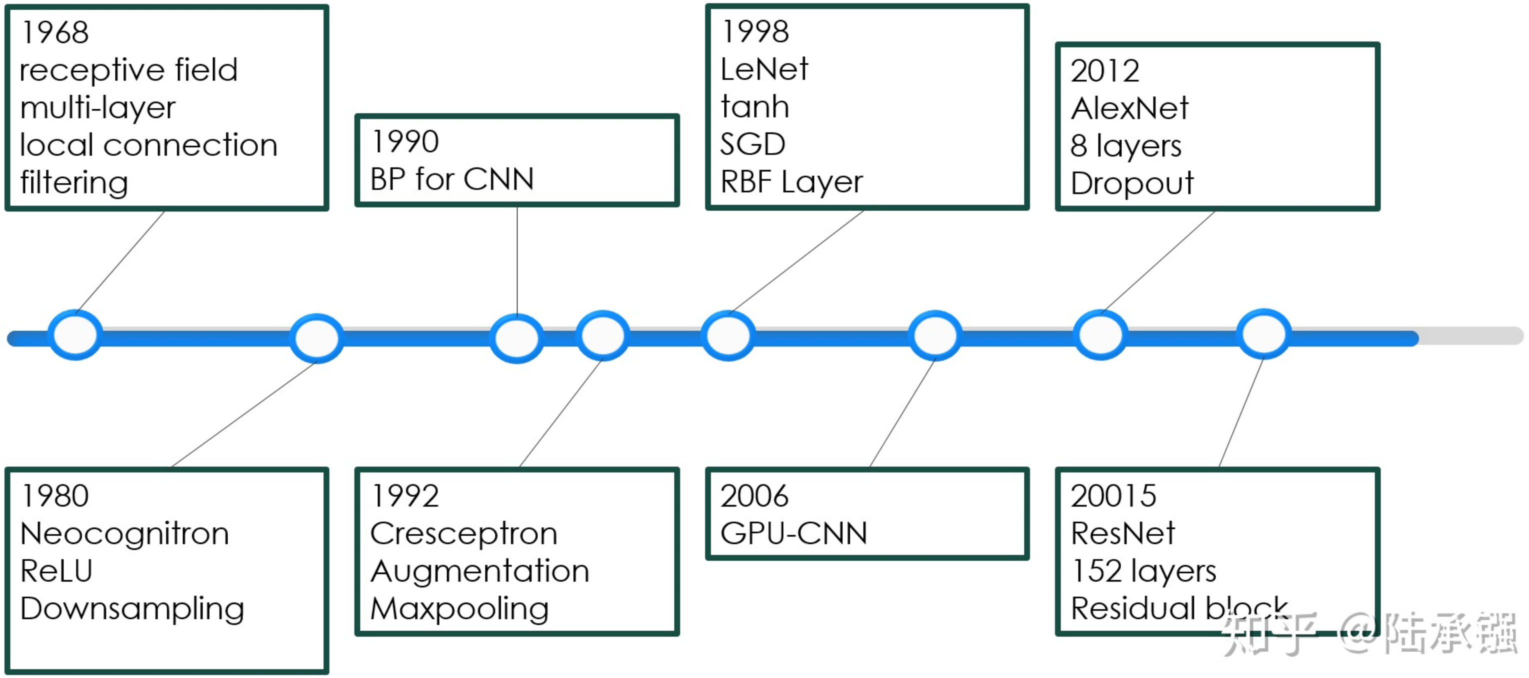
MLPs are more expressive than Perceptrons since they can learn highly non-linear class boundaries.

Neural networks

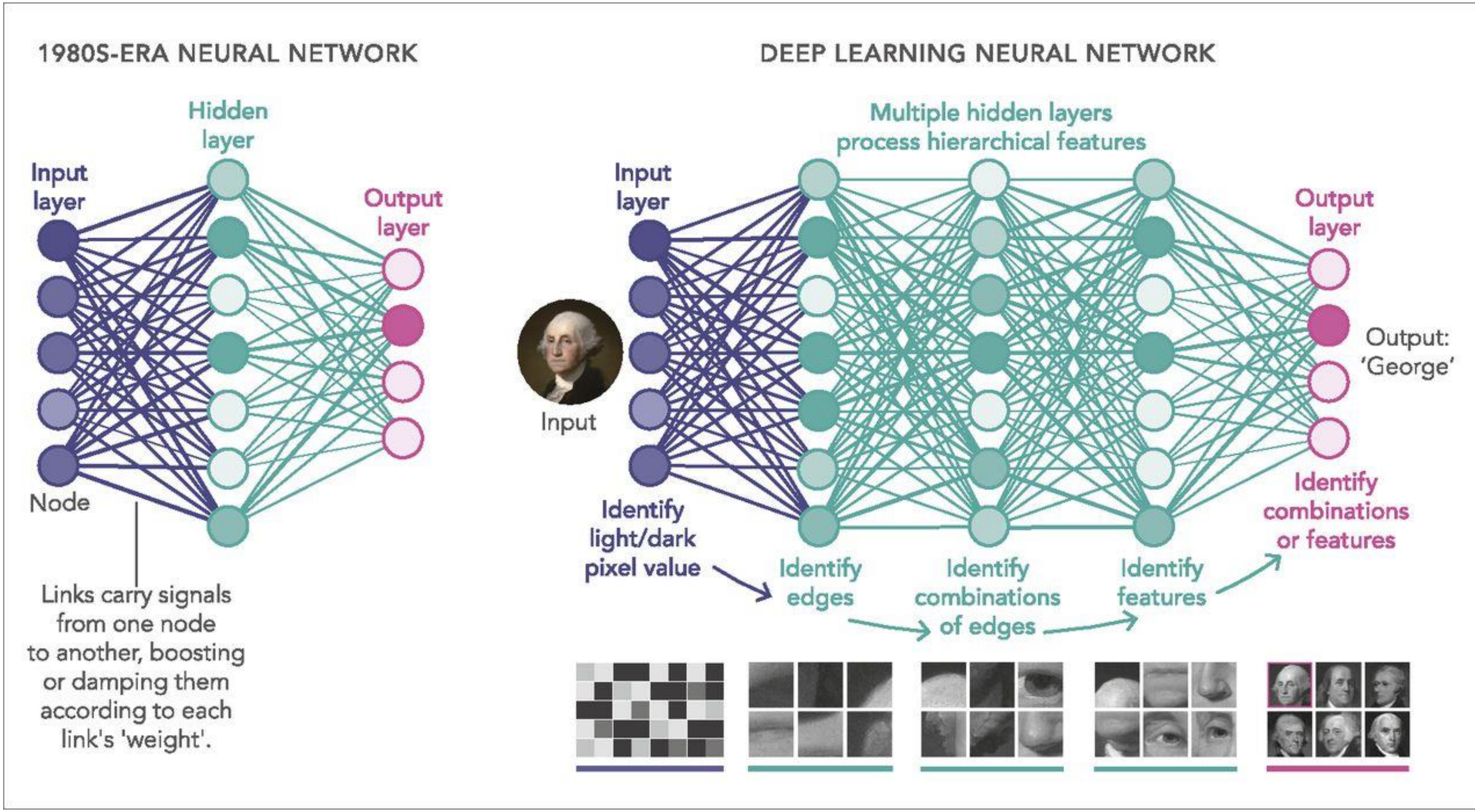


https://www.youtube.com/watch?v=O-bCDtHVPtA

Deep learning

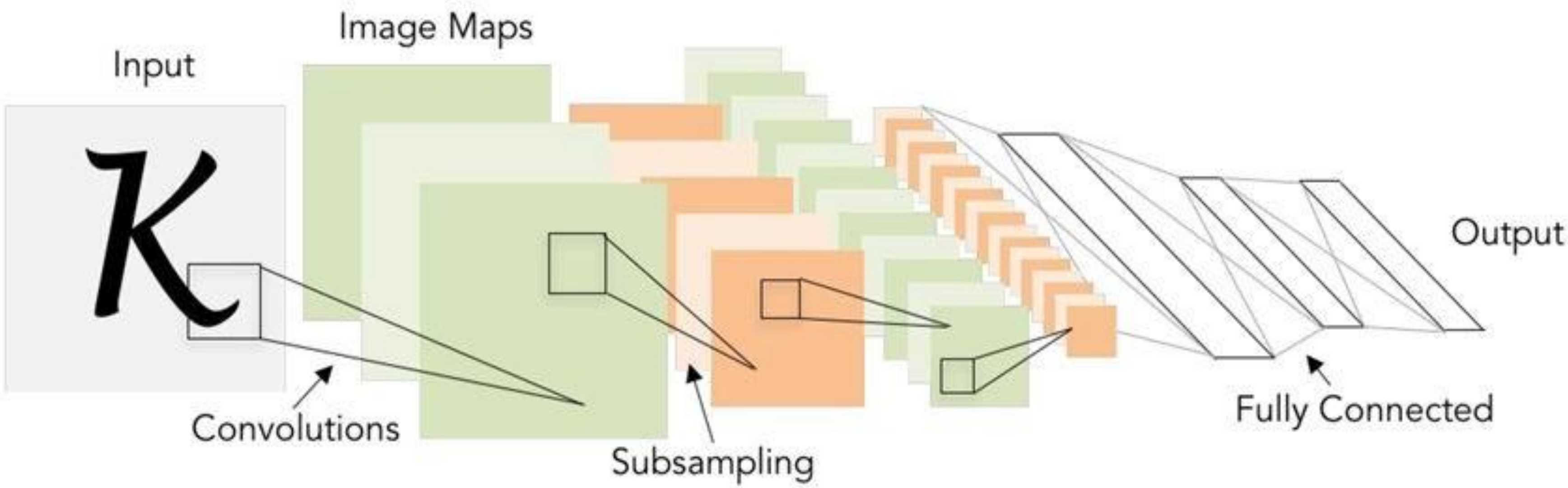


Neural Networks & Deep Learning



Convolutional neural network

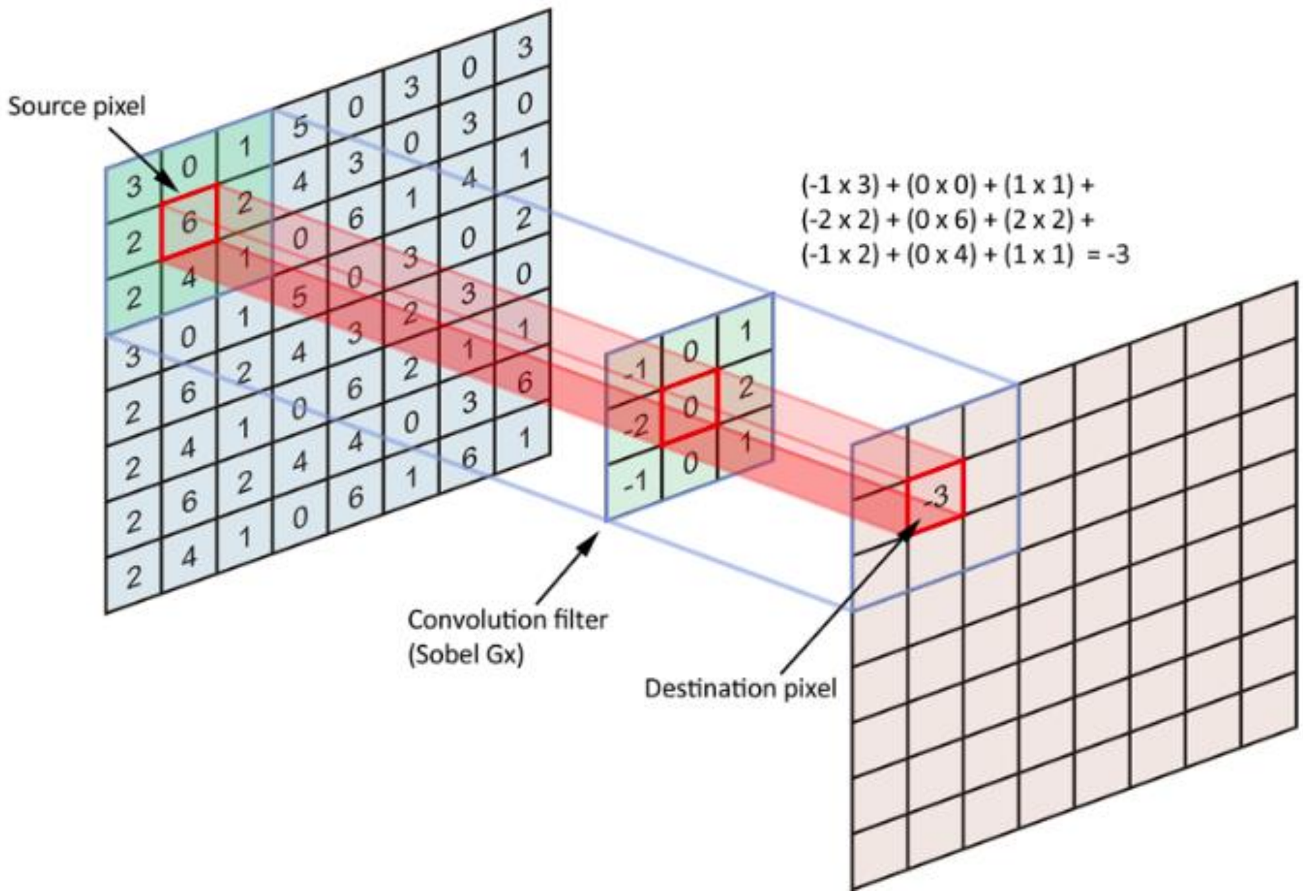
Building Blocks of Deep CNNs



LeNet-5

- Convolution layers - replaces many fully connected layers.
- Subsampling layers - max pooling, average pooling…
- Fully connected layers
- Activations - mostly Rectified Linear Units (ReLu) these days.

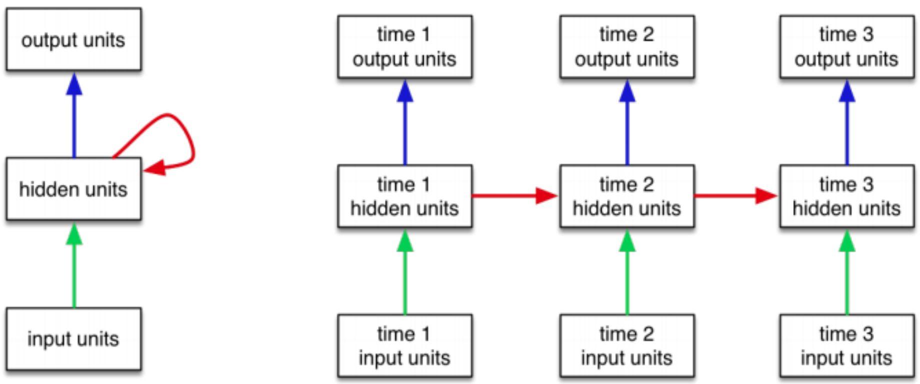
- **CNN**

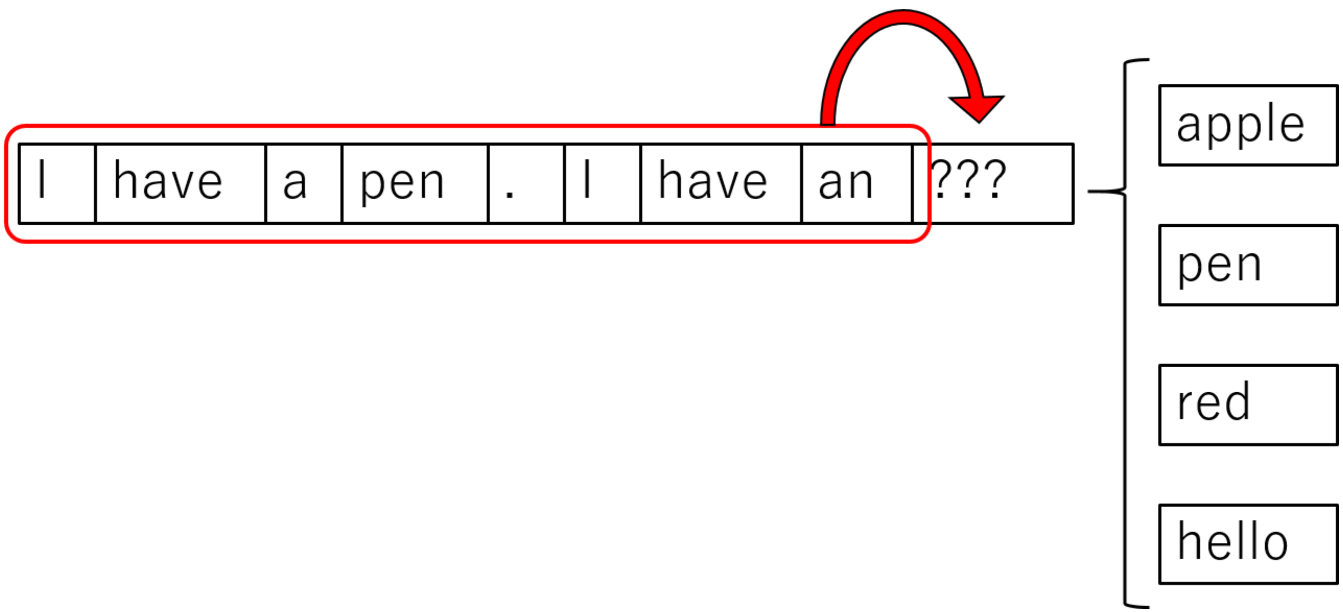

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

What AlexNet Learns?

Recurrent neural network

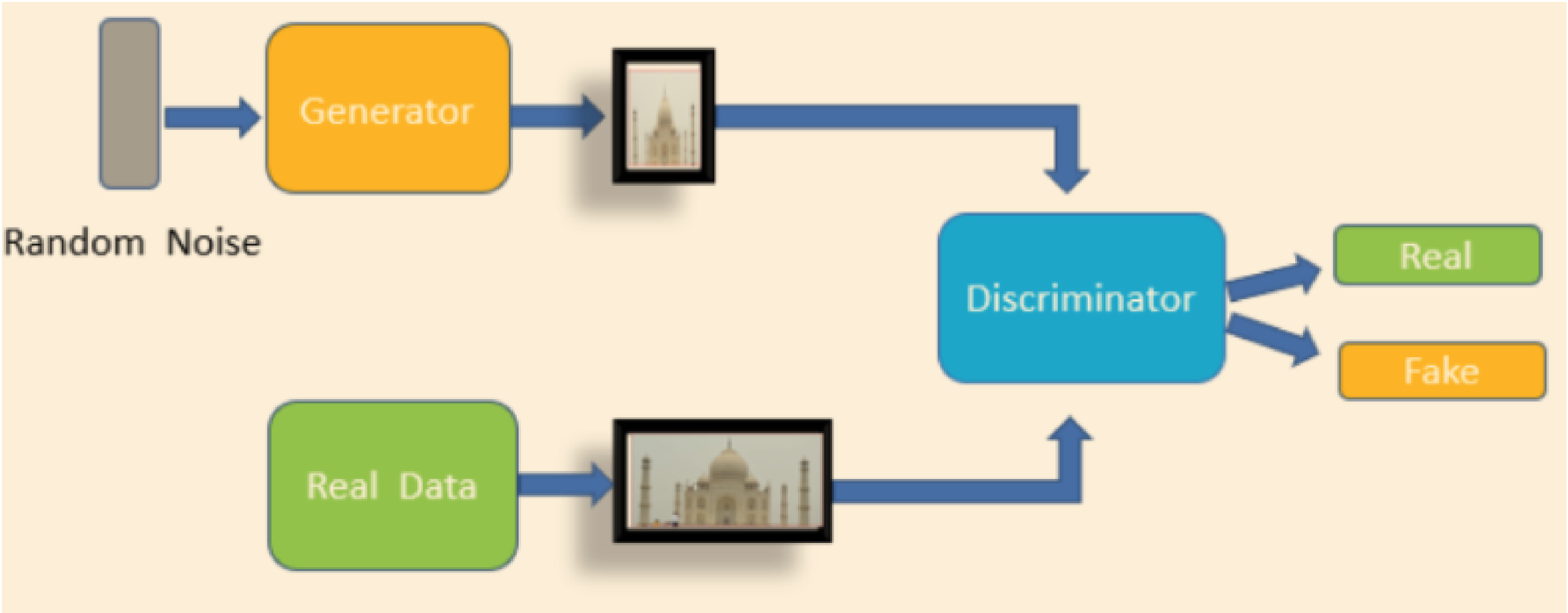
Recurrent neural network



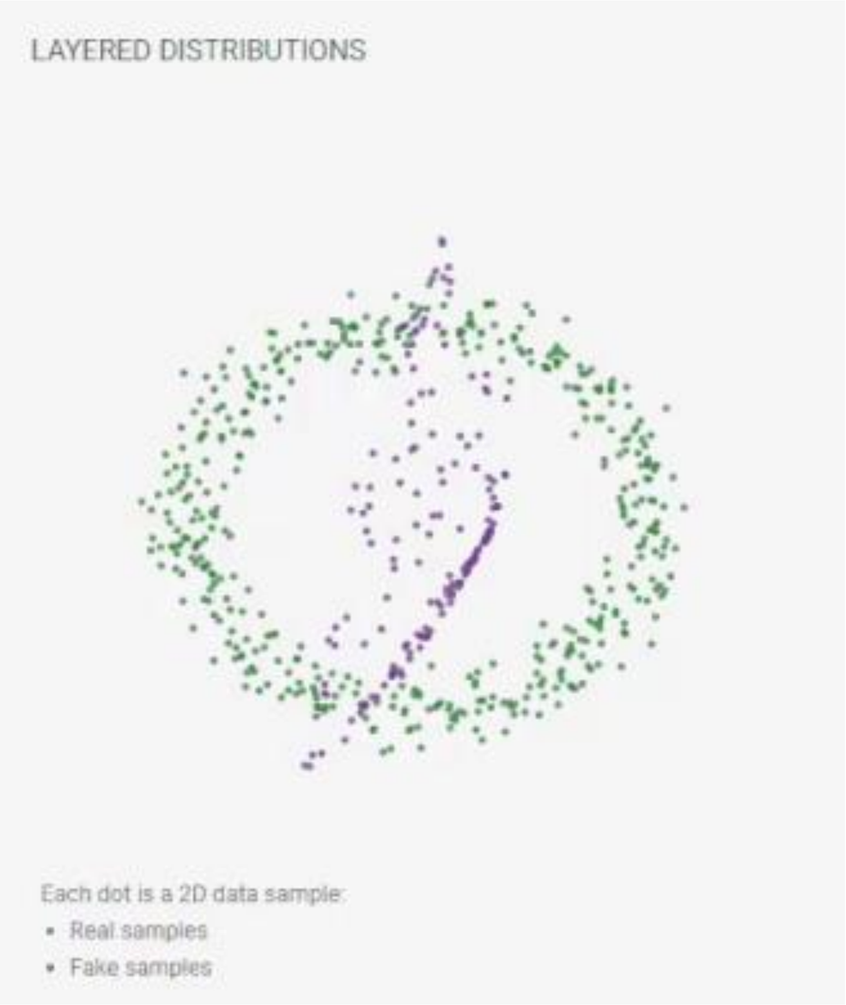


Generative Adversarial Network

- Generative adversarial network

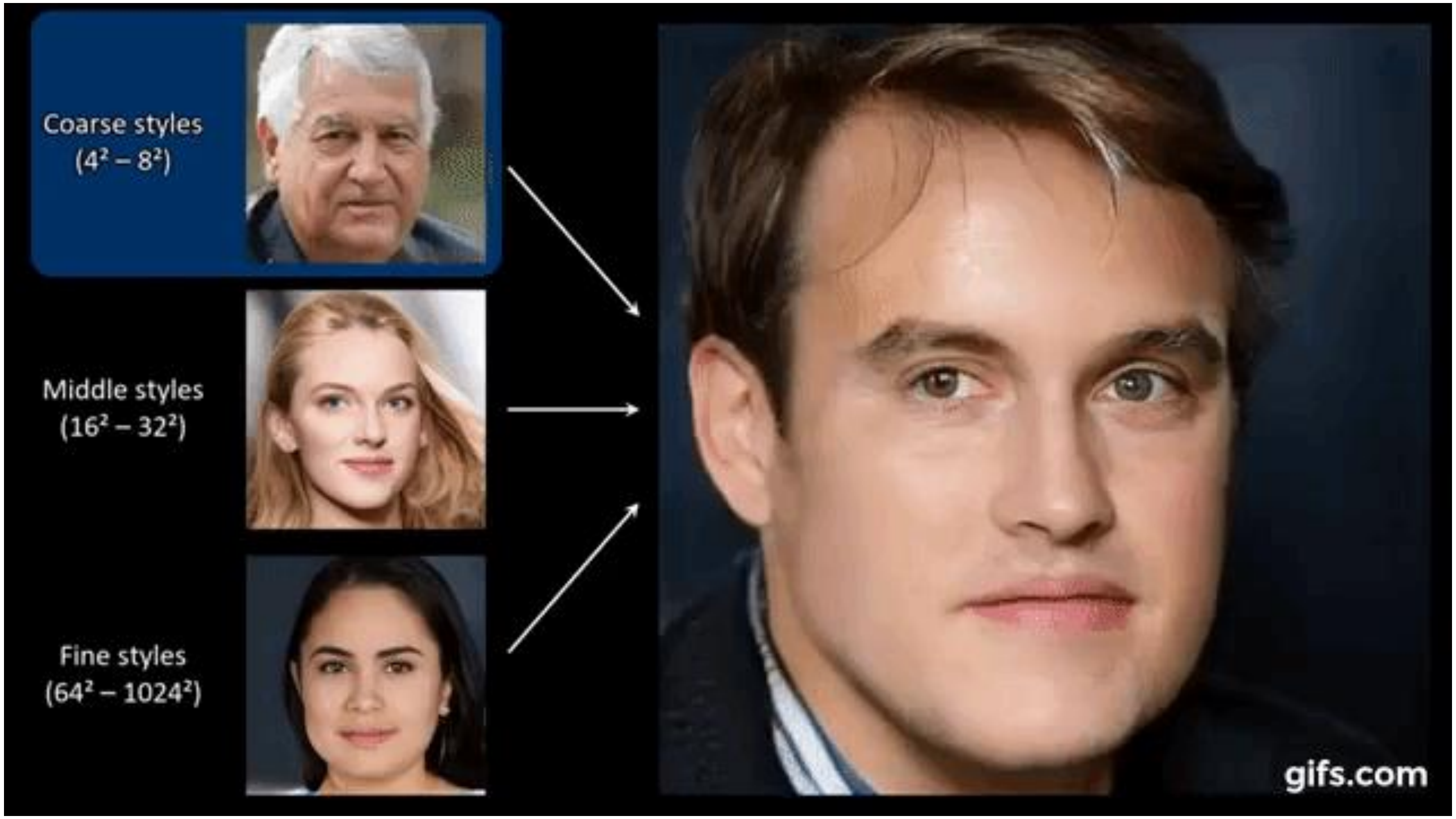


- Generative adversarial network



https://poloclub.github.io/ganlab/

- # Generative adversarial network

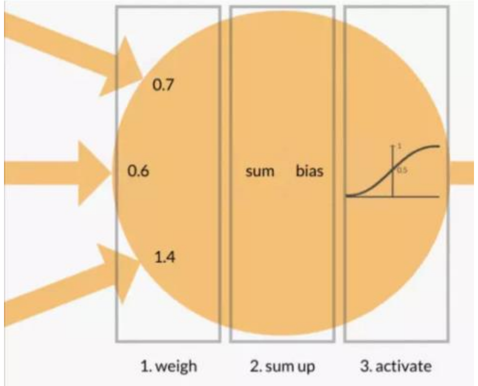


https://www.kdnuggets.com/2020/03/generate-realistic-human-face-using-gan.html

Activation

Activation Function: Sigmoid

- Non-linearity based on sigmoid.

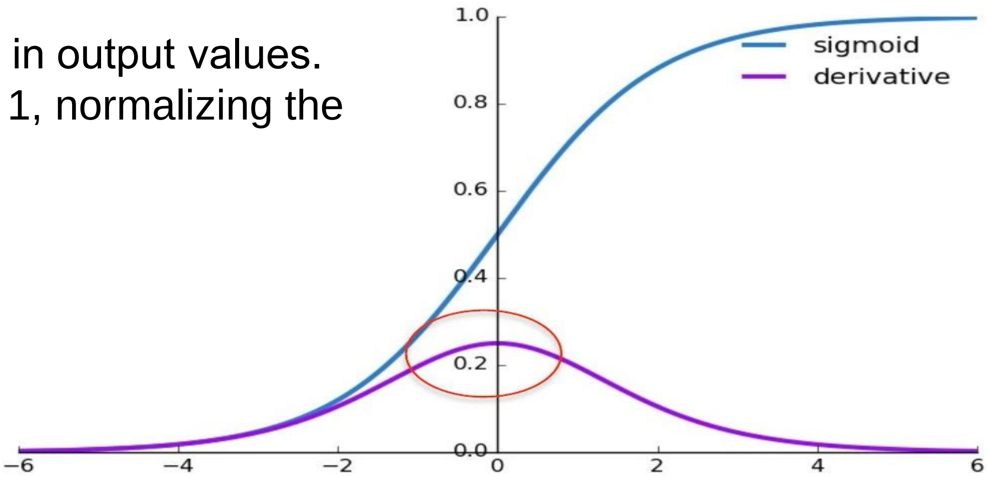


$$g_{sig}(in) = \frac{1}{1+e^{-in}}$$

$$g'_{sig}(in) = \frac{1}{(1+e^{-in})}\left(1 - \frac{1}{(1+e^{-in})}\right)$$

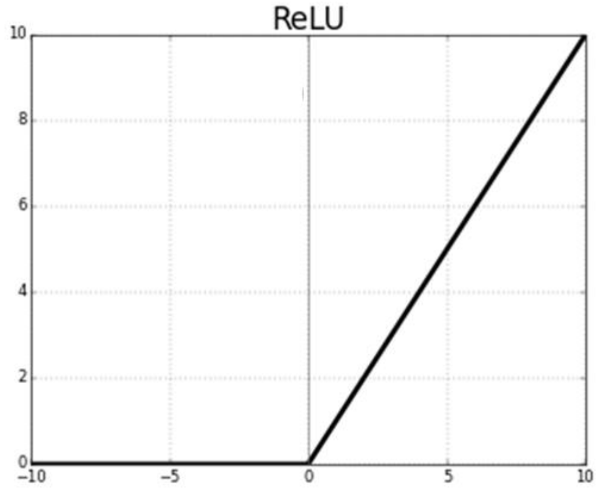
$$= g_{sig}(in)(1 - g_{sig}(in))$$

- Advantages
  - Smooth gradient, preventing “jumps” in output values.
  - Output values bound between 0 and 1, normalizing the output of each neuron.
- Disadvantages
  - Vanishing
  - Computationally expensive



Rectified Linear Units (ReLU)

- Maximum gradient magnitude is 1
- Still non-linear
- Gradient shape?

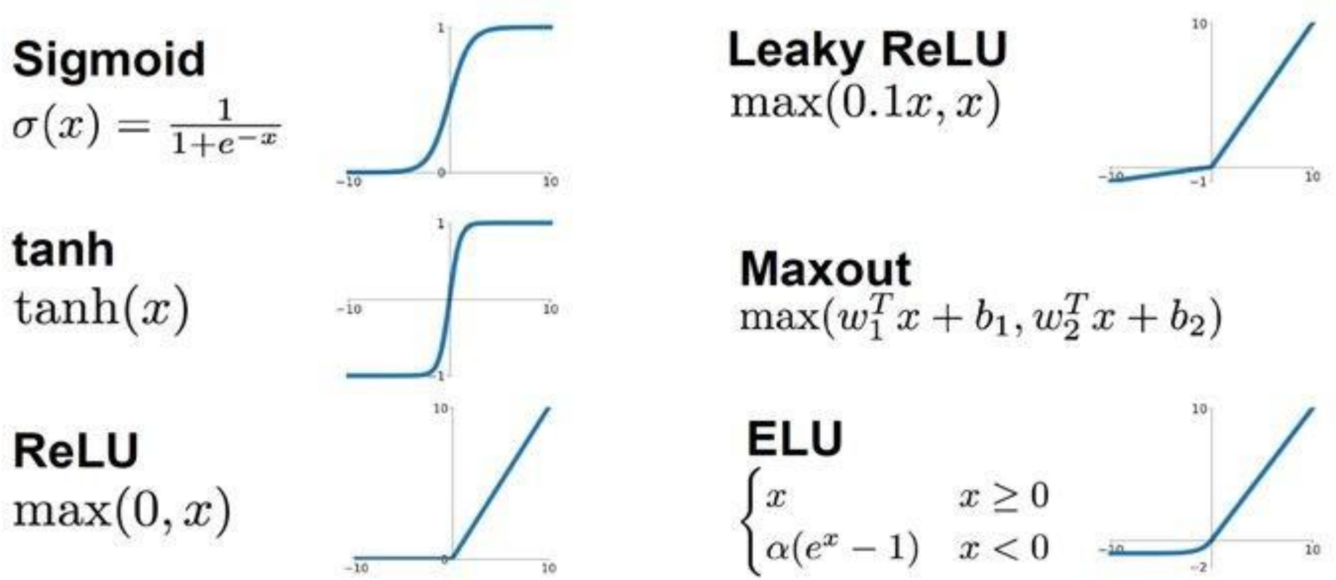


$$f(x) = \max(0, x).$$

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Advantages
  - Computationally efficient—allows the network to converge very quickly
  - Non-linear—although it looks like a linear function, ReLU has a derivative function and allows for backpropagation
- Disadvantages
  - The Dying ReLU problem—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

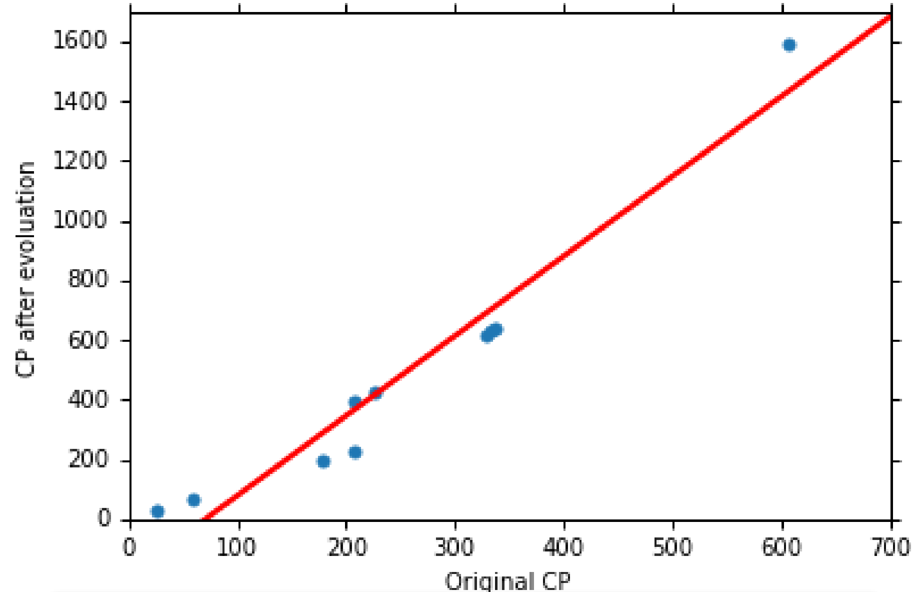
Rectified Linear Units (ReLU)



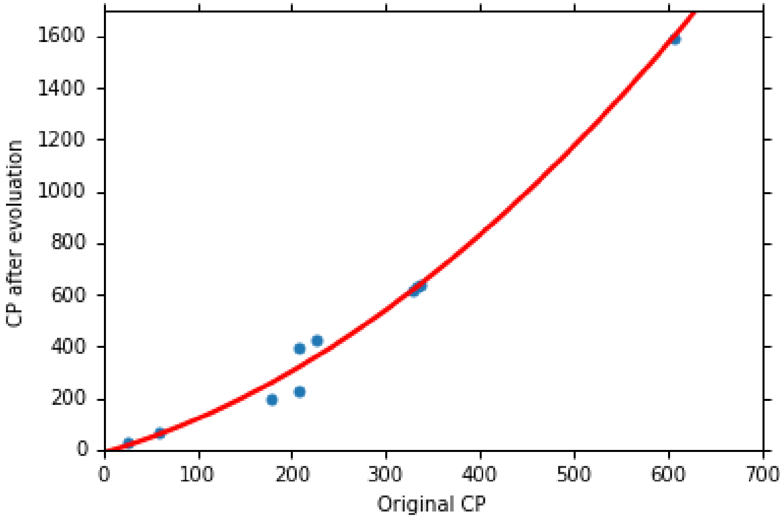
Bias & Variance

Bias VS Variance

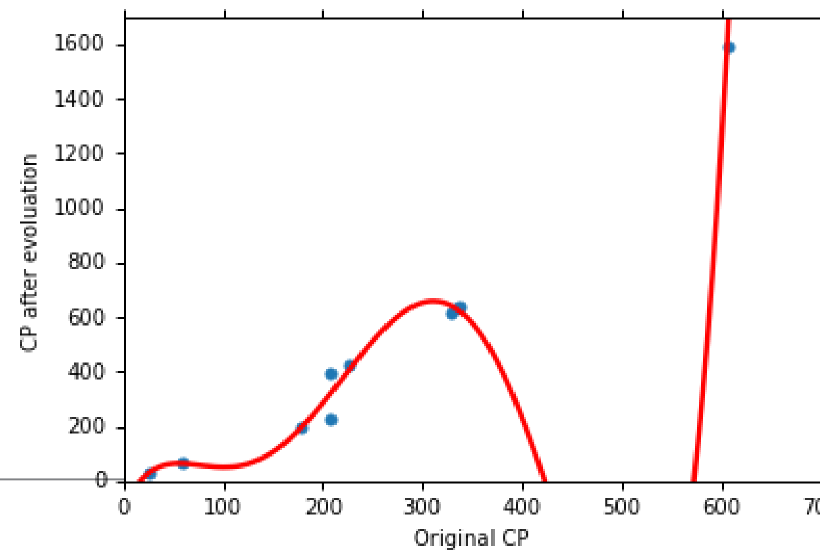
$$y = b + w \cdot x_{cp}$$



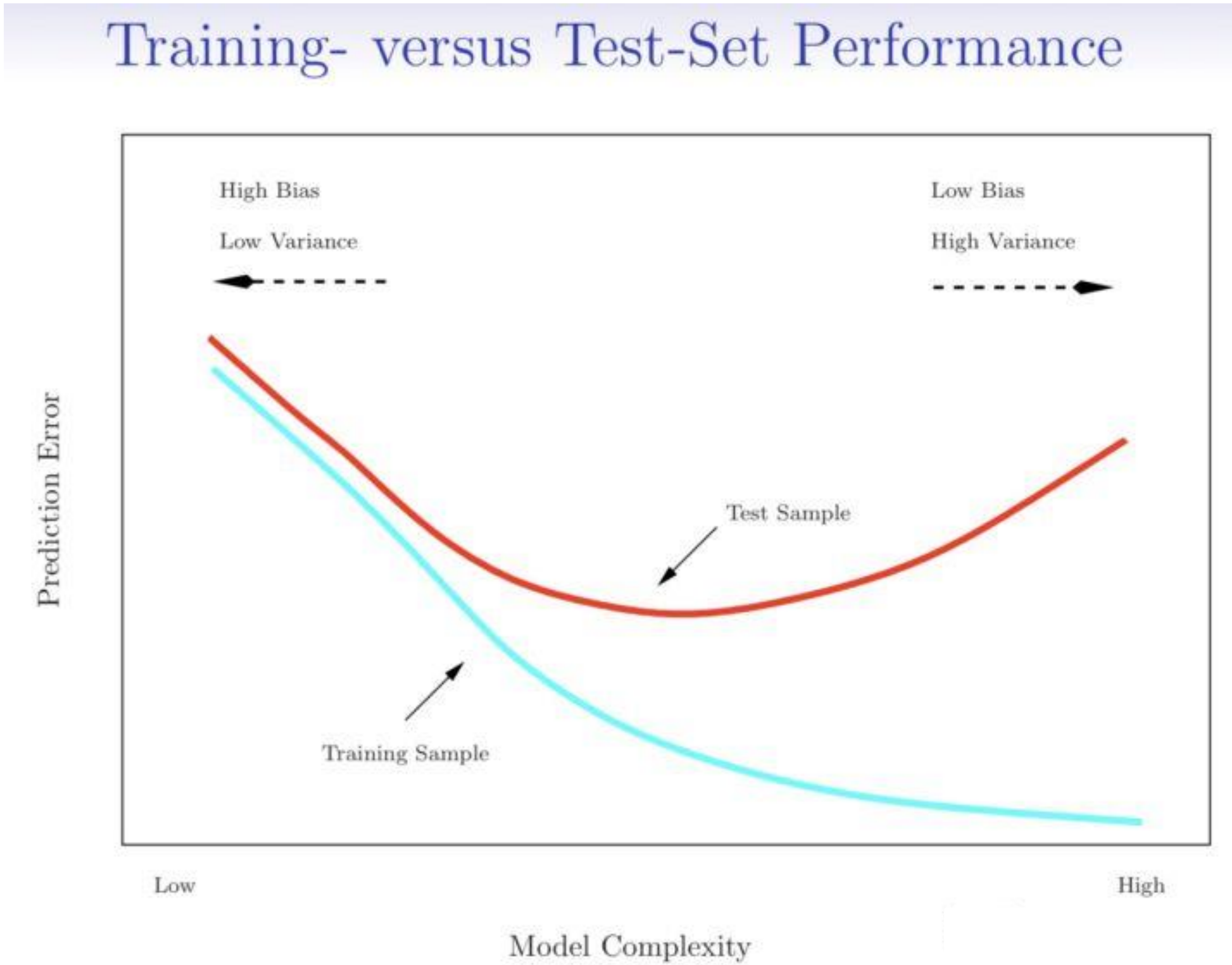
$$y = b + w_1 \cdot x_{cp} + w_2 \cdot (x_{cp})^2$$



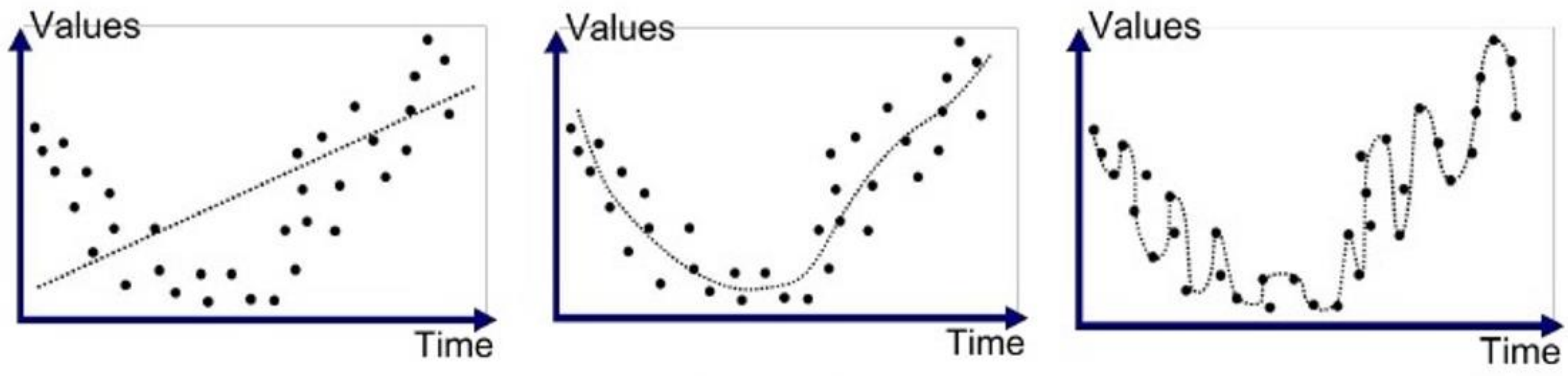
$$y = b + w_1 \cdot x_{cp} + w_2 \cdot (x_{cp})^2 + w_3 \cdot (x_{cp})^3 + w_4 \cdot (x_{cp})^4 + w_5 \cdot (x_{cp})^5$$



Bias & Variance



Underfit & Good fit & Overfit



Underfitted

Good Fit/Robust

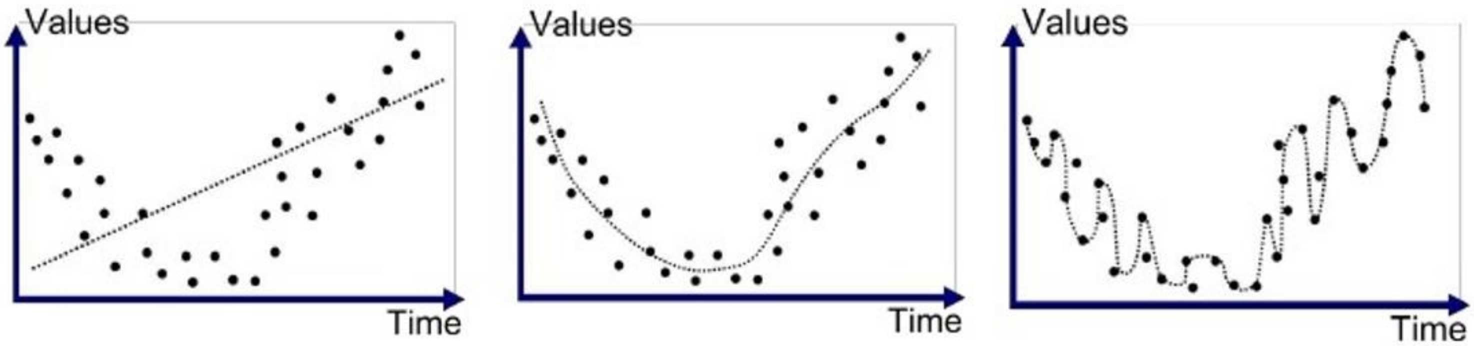
Overfitted

Regularization

Regularization

- Optimizing a loss function to learn parameters

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Fitting to data}} + \underbrace{\lambda R(W)}_{\text{Choose the simplest model}}$$



Underfitted   Good Fit/Robust   Overfitted

Regularization

- Commonly-used regularizers
  - L2-regularization (Lasso): $R_{L_2}(w) \triangleq ||W||_2^2$
  - L1-regularization (Ridge): $R_{L_1}(w) \triangleq \sum_{k=1}^{Q} ||W||_1$
  - Drop-out: it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.
  - Early stopping: keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model. This is known as early stopping.

L2-Norm

- L2-Norm: L2 regularization is also called *Weight decay*.

$$\|\mathrm{W}\|_2 \equiv \sqrt{\sum_{i=1}^{m} |w_i|^2}$$

$$L = L' + \frac{\lambda}{2n} \sum_w w^2$$

$$\frac{\partial L}{\partial w} = \frac{\partial L'}{\partial w} + \frac{\lambda}{n} w$$

$$w \rightarrow w - \eta \frac{\partial L'}{\partial w} - \frac{\eta \lambda}{n} w$$

$$= \underline{\left(1 - \frac{\eta \lambda}{n}\right)} w - \eta \frac{\partial L'}{\partial w}$$

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

L1-Norm

- L1-Norm:

$$\|\mathrm{W}\|_1 \equiv \sum_{i=1}^{m} |W_i|$$

$$L = L' + \frac{\lambda}{n} \sum_{w} |w|$$

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

$$\frac{\partial L}{\partial w} = \frac{\partial L'}{\partial w} + \frac{\lambda}{n} sgn(w)$$

$$w \rightarrow w - \frac{\eta\lambda}{n} sgn(w) - \eta \frac{\partial L'}{\partial w}$$

L1 vs L2

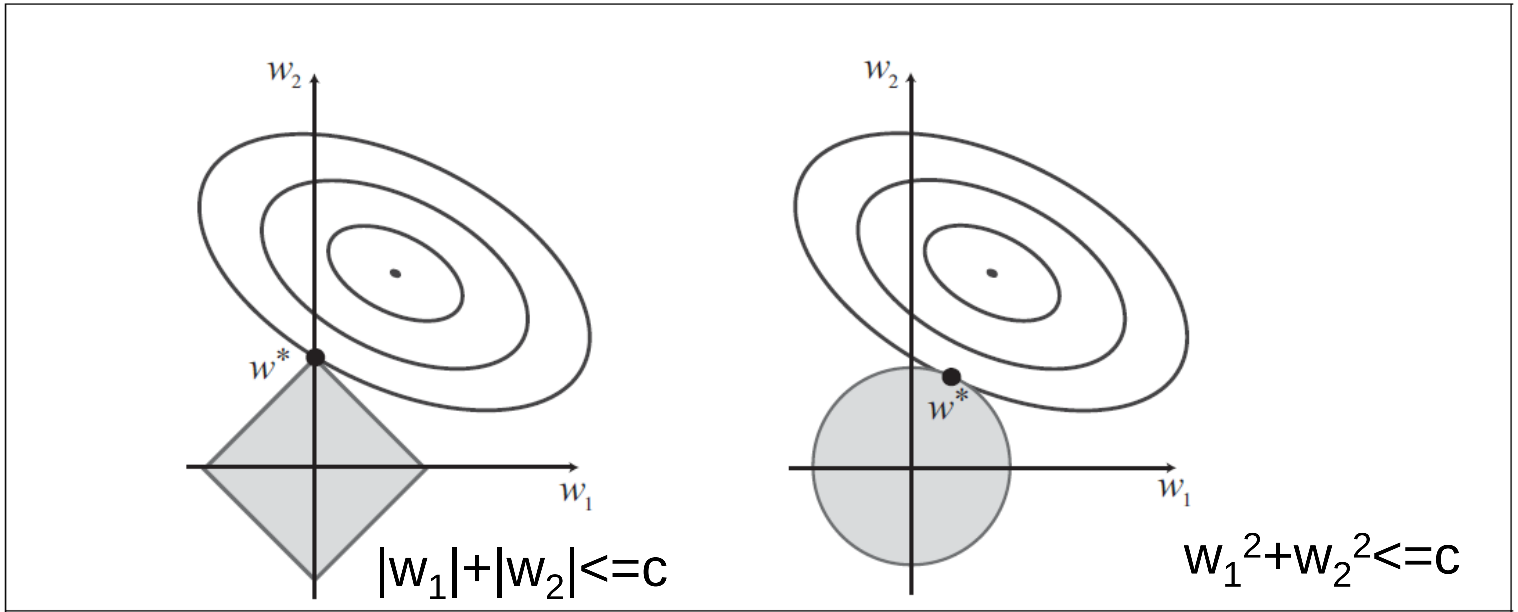
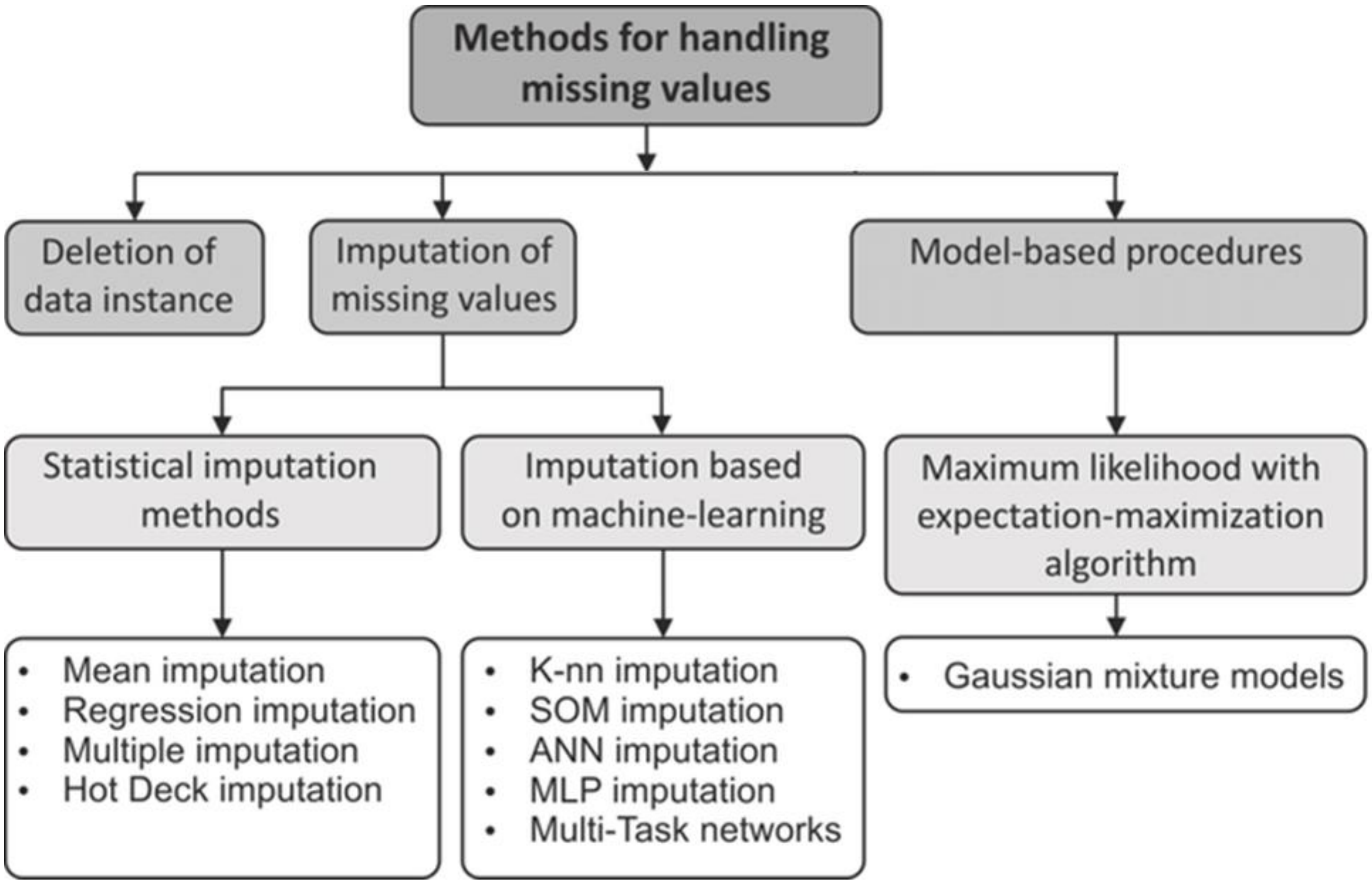


Figure 18.14 Why L_1 regularization tends to produce a sparse model. (a) With L_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. (b) With L_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

Handle Missing Value

Handle Missing Values

- Some models can handle missing data, e.g., XGBoost. Deep learning models.
- When models cannot handle missing values:
  - Too many missing values and the dataset is big, then delete the instance/feature
  - Categorical data: transform NaN as new category; Replace by most frequent value; Replace using an algorithm like KNN using the neighbours; Predict the observation using a multiclass predictor, etc.
  - Continuous data: NaN as 0; mean/medium/mode; replace with value before or after; interpolation; regression.



Source: Jaroslav Bendl, 2016

Reference

- Hungyi Lee Tutorial

  http://speech.ee.ntu.edu.tw/~tlkagk/courses_ML20.html