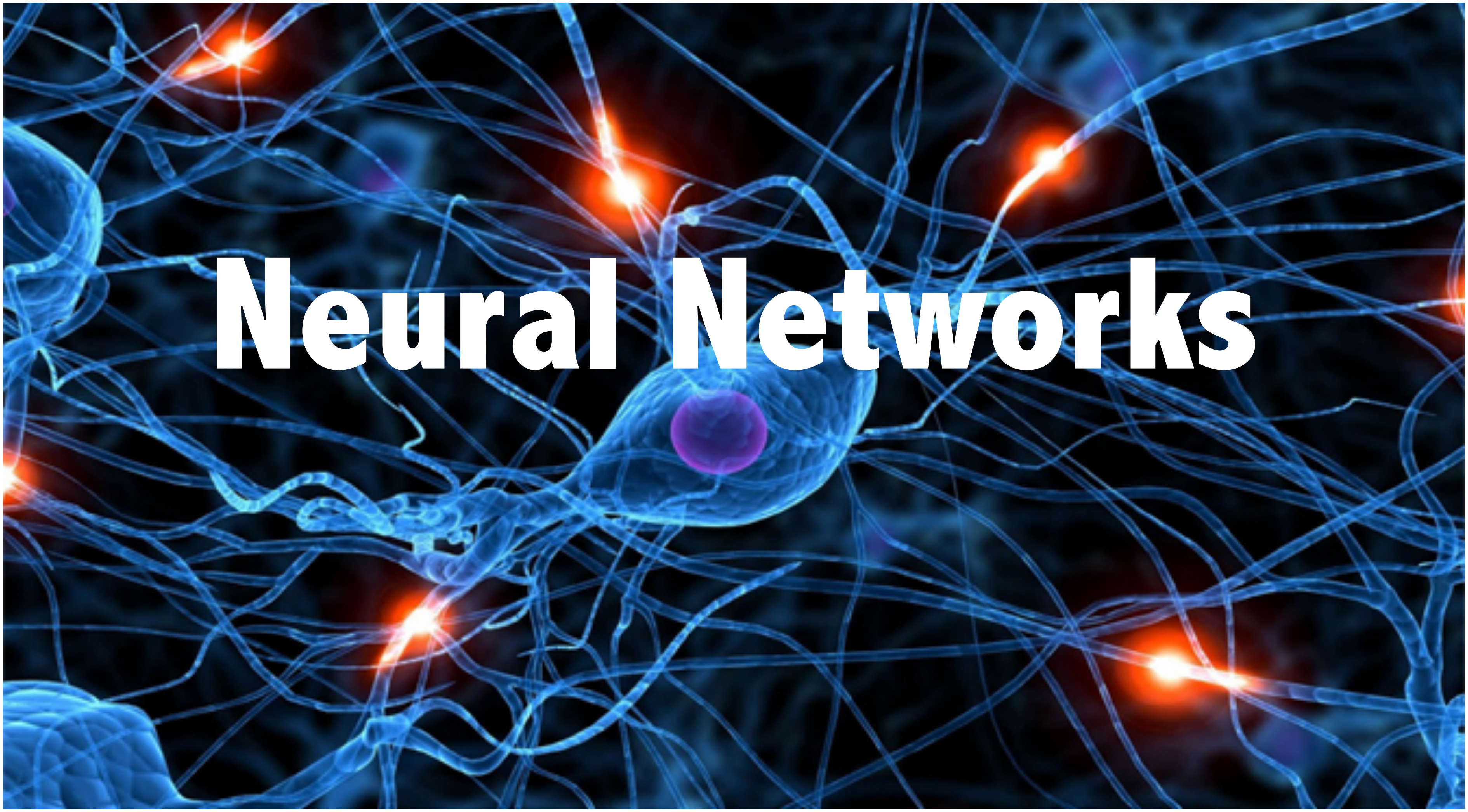

Neural Networks

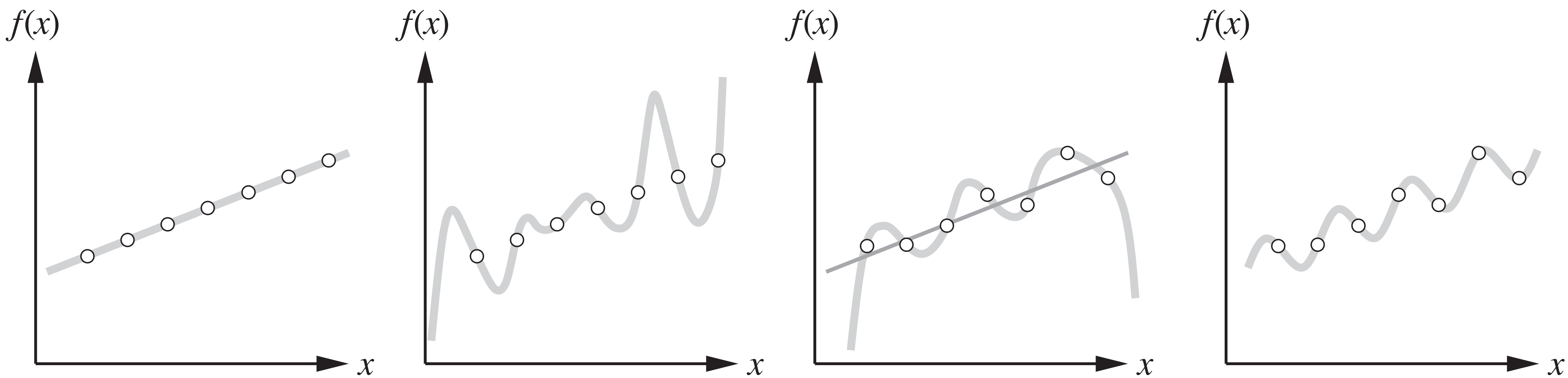
Review – Machine Learning

- Three forms:
- Supervised learning
  - The agent is given some input-output pairs and it learns a function that maps the input to the output.
  - Example: training a naïve Bayes classifier.
- Unsupervised learning
  - The agent learns patterns in the input even though no explicit output or feedback is given.
  - Example: clustering
- Reinforcement learning
  - The agent is given feedback (rewards) during the steps of a task and the agent learns a function from states to predicted rewards.

Supervised learning

- Given a ***training set*** of *N* example input-output pairs:
  - $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$
- Each y is generated by an unknown function y = f(x).
- Goal: discover a function h that approximates the true function f.
- h is called a ***hypothesis***.
- Machine learning algorithms conduct searches for the "best" f.
- We can measure the accuracy of a hypothesis on a ***test set*** of examples that are distinct from the training set.
- A hypothesis ***generalizes well*** if it correctly predicts examples from the test set (even though it has never seen them before).

Supervised learning



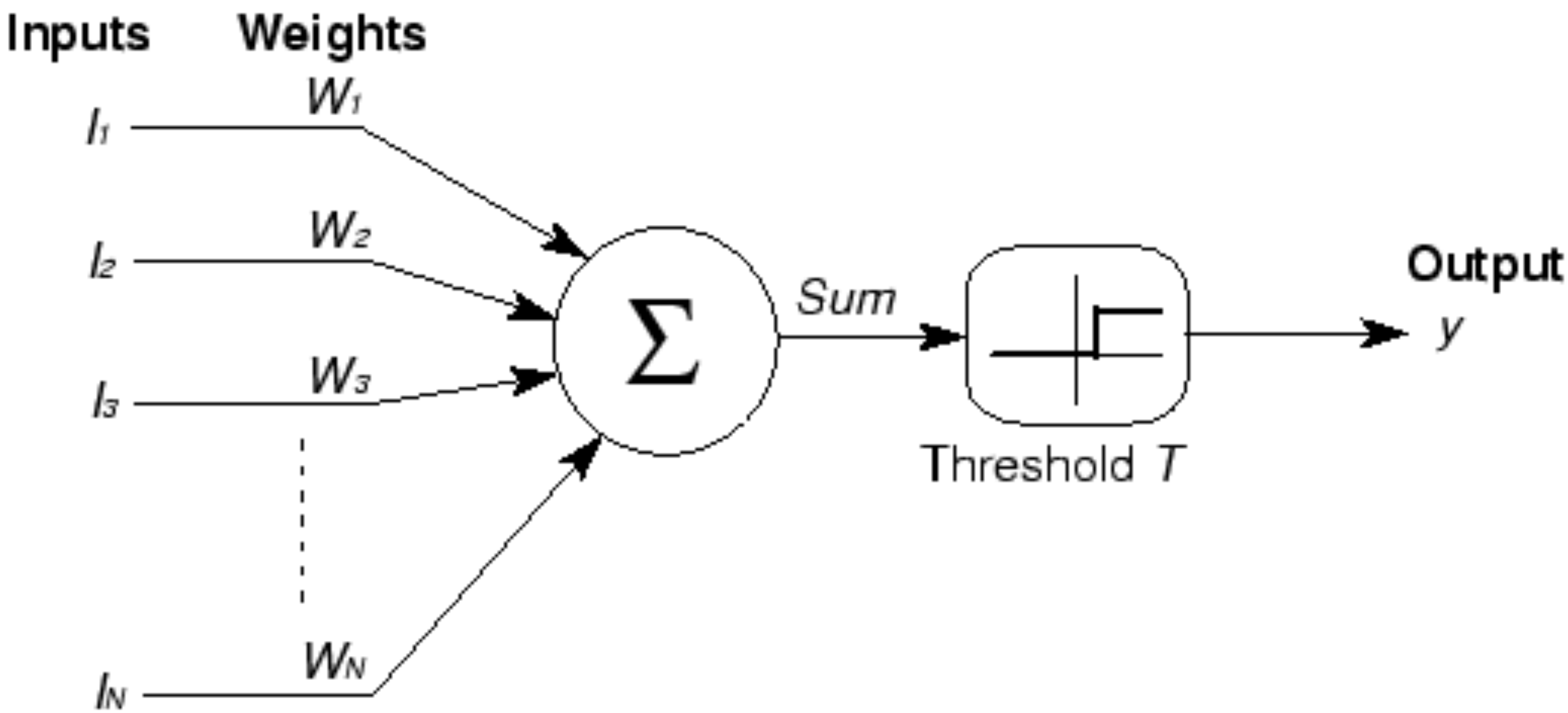
(a) (b) (c) (d)

Supervised learning

- Poor generalization is sometimes caused by overfitting: our hypothesis has learned the training set very well, but it has poor accuracy on the test set.
    - Analogous to "memorizing" the training set.
- When the output y is one of a finite set of values (e.g., sunny/cloudy/rainy or true/false), the learning problem is called ***classification***.
- When the output is a number, the problem is called ***regression***.
    - Yes, linear regression is a machine learning algorithm!

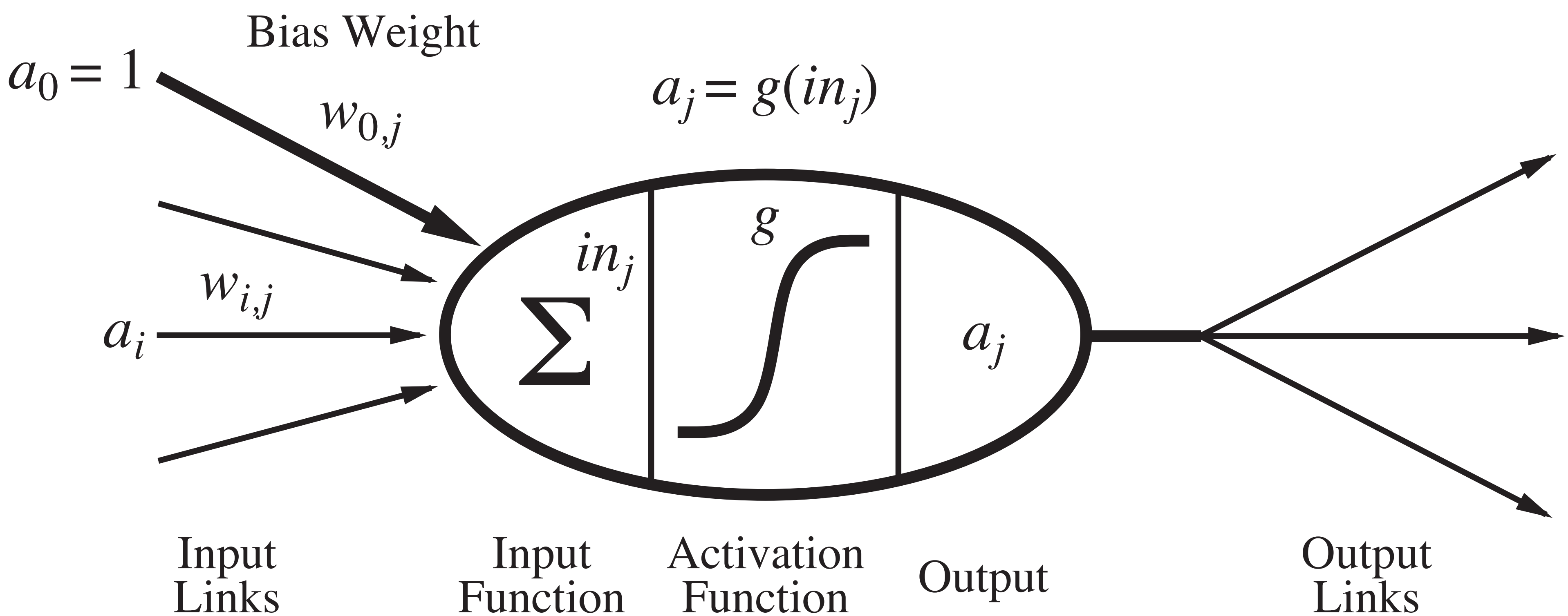
McCullough-Pitts neuron

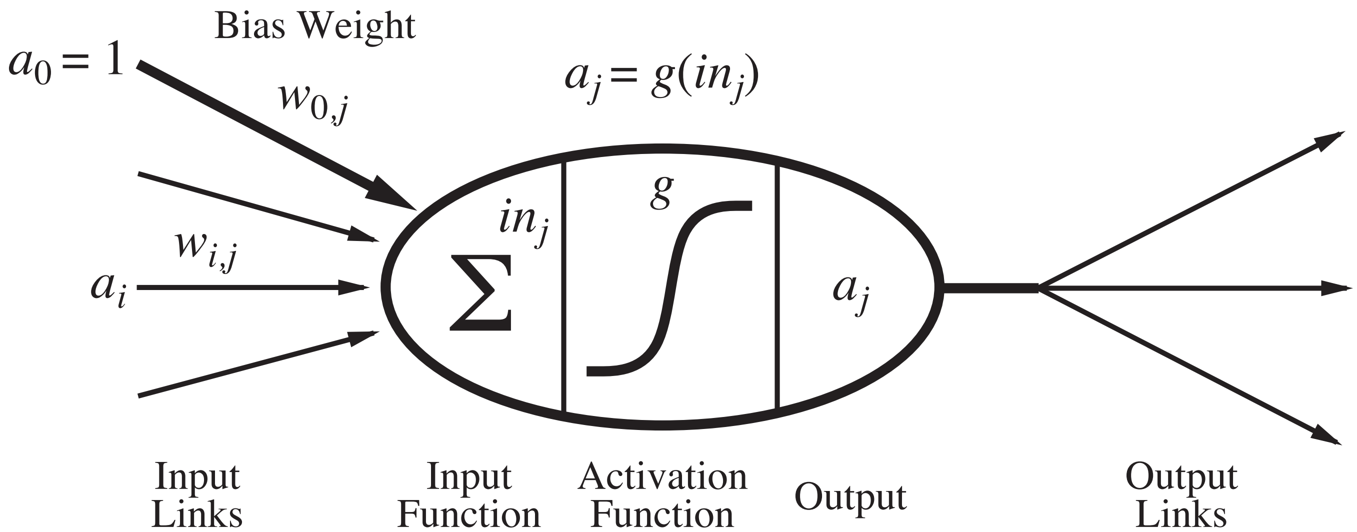
- 1943: Warren McCullough and Walter Pitts, two electrical engineers, develop the first model of an ***artificial neuron***, called threshold logical units.



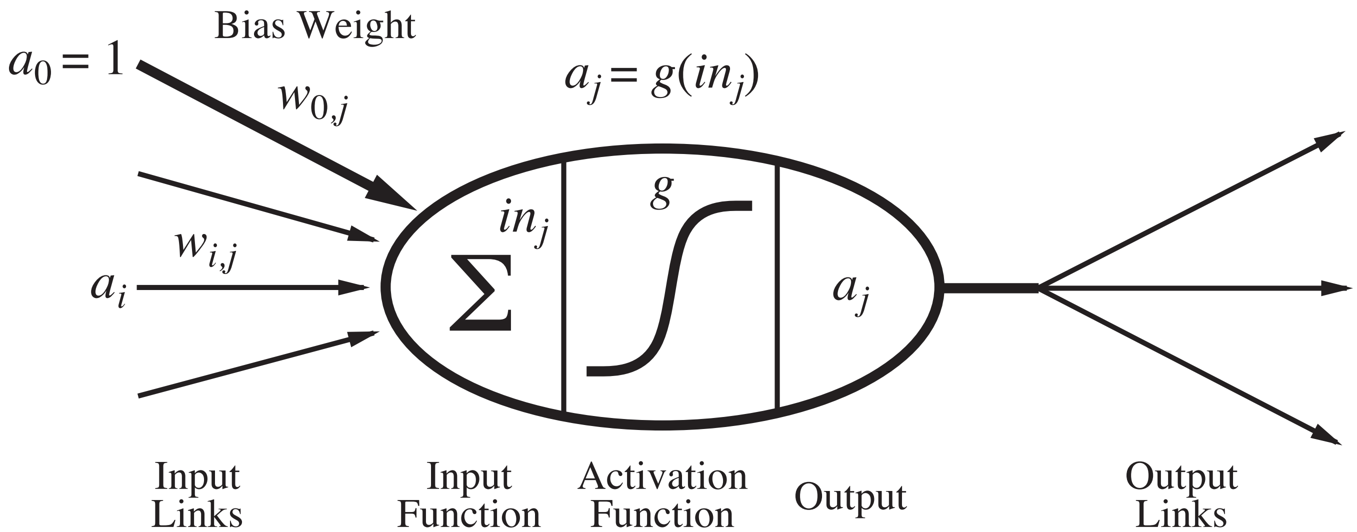
Perceptron

- 1958: Frank Rosenblatt refined the McCullough-Pitts neuron into the ***perceptron***.





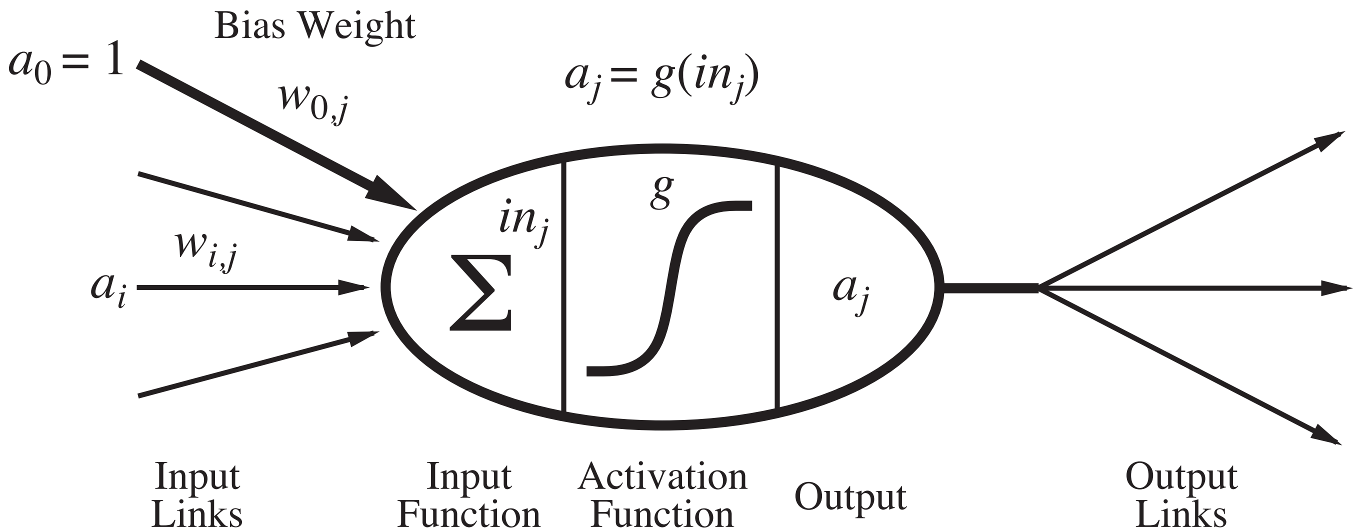
- NNs are composed of nodes or units connected by directed links (a graph structure).
- Each unit receives a collection of numeral inputs (a_0, a_1, ...) and produces a numeral output (a_j).
- A link from unit *i* to unit *j* has a weight w_{ij} associated with it.
- Each unit has a dummy input (a_0) that is always set to 1.



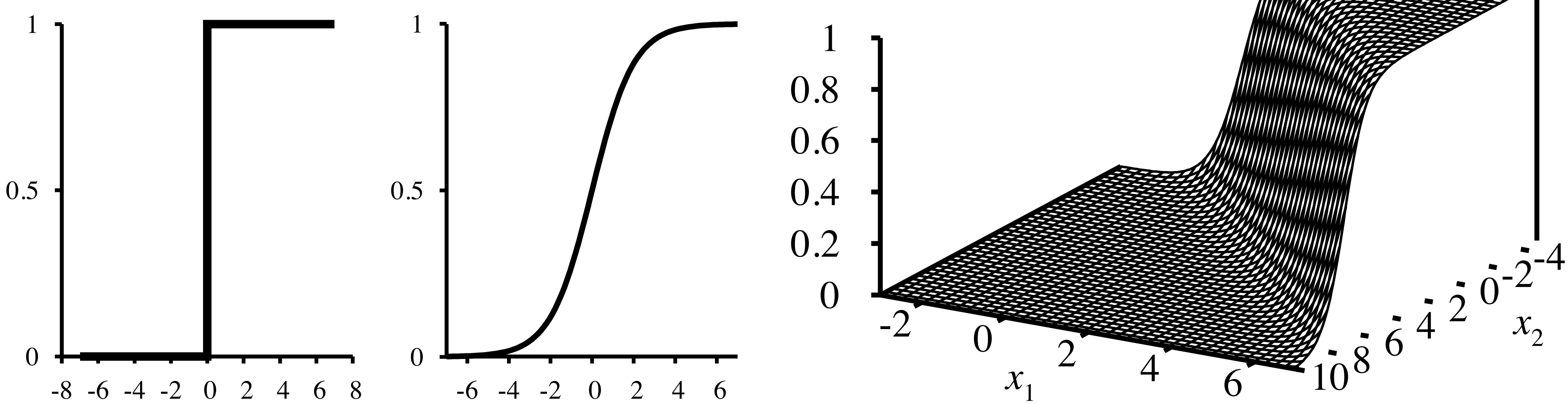
- Each unit j first computes a weighted sum of its inputs:

$$in_j = \sum_{i=0}^{n} w_{i,j} \cdot a_j$$

- Then it applies an activation function g to this sum to produce the output: $a_j = g(in_j)$



- The function g is typically either a hard threshold function or the logistic function: $\dfrac{1}{1+e^{-x}}$

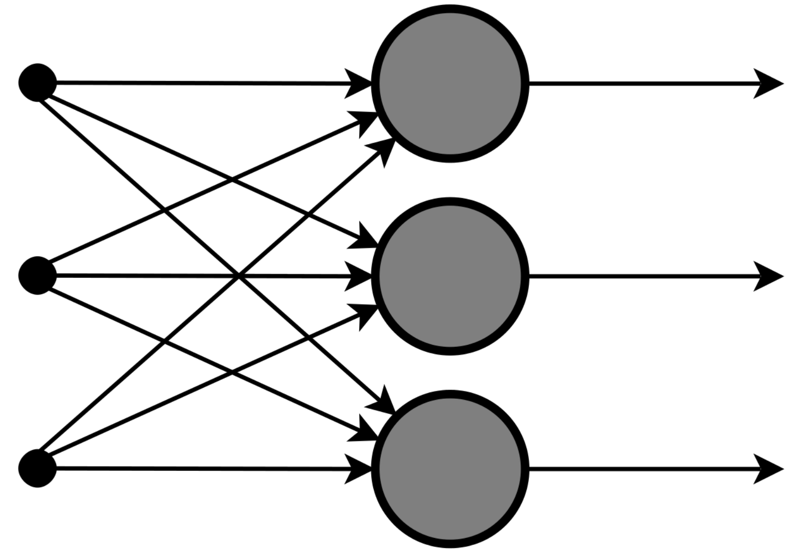


Neural networks

- Two basic types of networks.
  - Feed-forward: Links are only in one direction (DAG).
  - Recurrent: Allows outputs to feed back into inputs.
    - System may reach a steady state or may exhibit oscillations or chaotic behavior.
- Feed-forward networks are usually arranged in layers, where each layer only receives input from the previous layer.
  - Single layer – all inputs connected directly to outputs
  - Multi-layer - one or more ***hidden layers*** of units in between input and output.

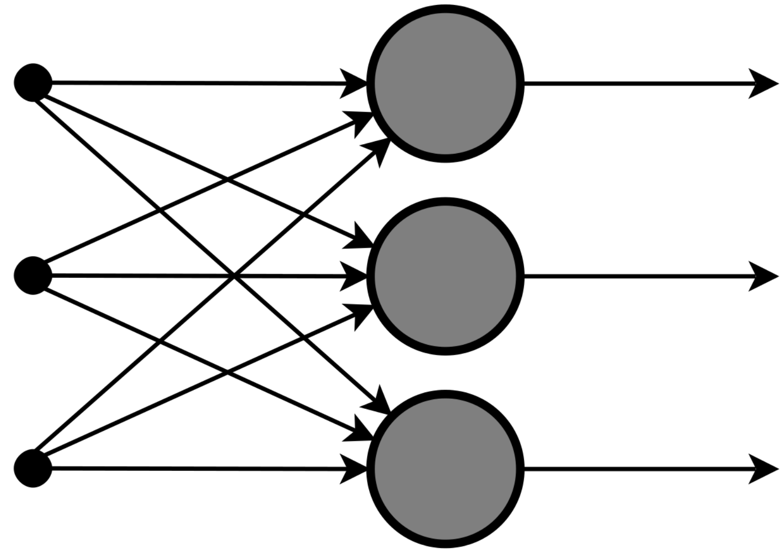
Single layer feed forward networks

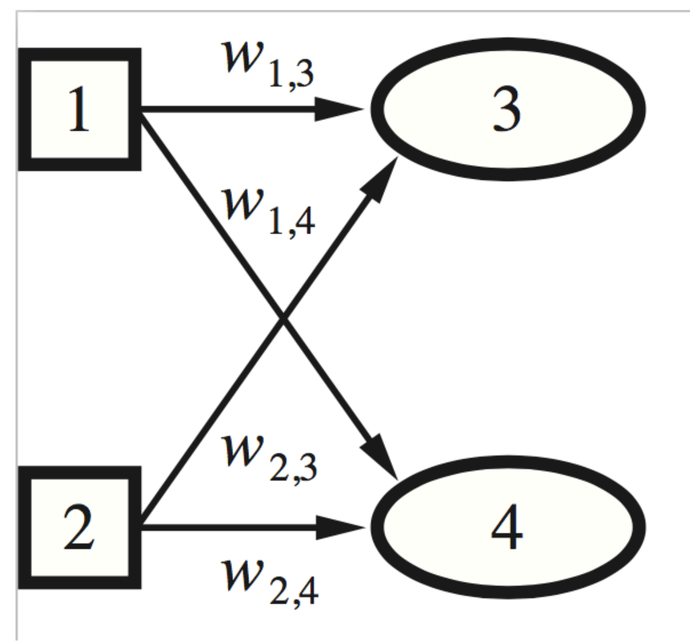
- One input layer (which is just the raw inputs).
- One output layer (of perceptron units).
- Example.



Single layer feed forward networks

- One input layer (which is just the raw inputs).
- One output layer (of perceptron units).
- Let's design a network to add two bits together.
- Needs two inputs (x_1, x_2), and two outputs (y_3, y_4).





Single layer feed forward networks

- There is an algorithm to change the weights of a single-layer network to make the network learn any function...
- Initialize starting weights randomly
- Do until you want to stop (*typically when accuracy is good enough or weights stop changing*):
  - for each training example (x, y):
    - use NN to get prediction of h(x)
    - if h(x) differs from y, update all weights:
    - w[i] = w[i] + (y – h(x)) * x[i]
  - compute accuracy over entire training data = (# predicted correctly)/(# of training examples)

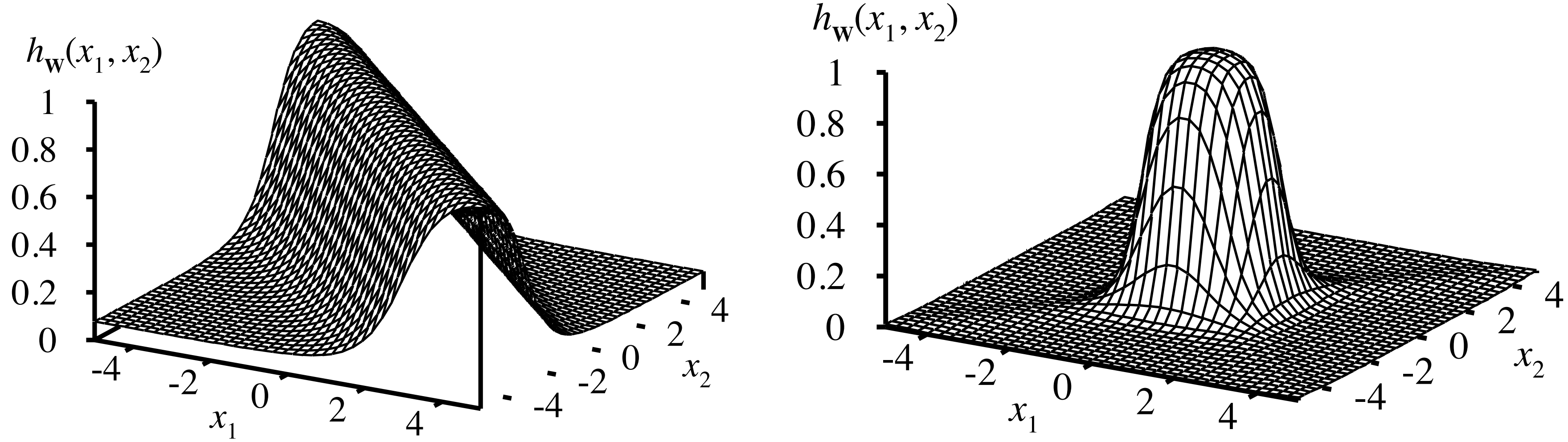
Single layer feed forward networks

- There is an algorithm to change the weights of a single-layer network to make the network learn any function...
- as long as it is linearly-separable!

Multi-layer feed forward networks

- McCullough, Pitts, and Rosenblatt were all aware of the linear separability problem.
- If we add another layer of units between the input and output layers, we can learn any function!
- http://playground.tensorflow.org/

Multi-layer feed forward networks



Multi-layer feed forward networks

- Learning is done through the backpropagation algorithm (*backprop*).
- Derived through calculus (we will skip).

Review perceptron learning

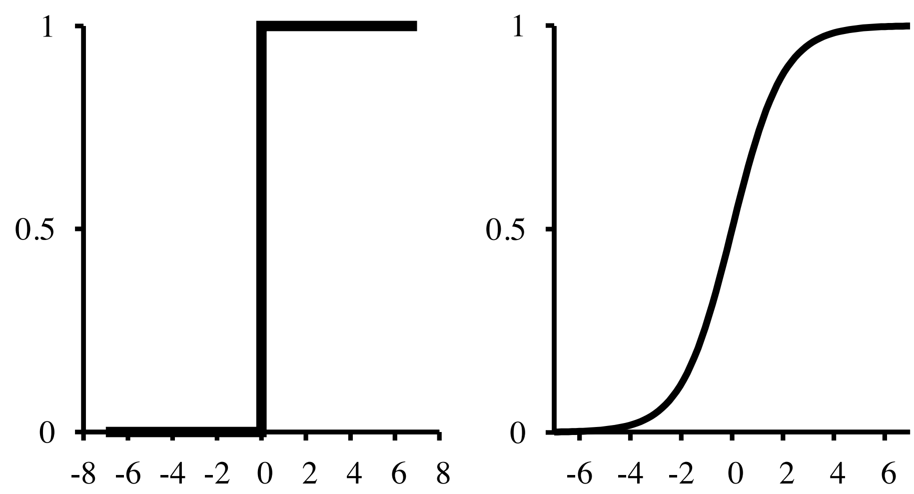
- Initialize starting weights randomly
- Do until you want to stop (*typically when accuracy is good enough or weights stop changing*):
    - for each training example (x, y):
        - use NN to get prediction of h(x)
        - if h(x) differs from y, update all weights:
        - w[i] = w[i] + (y – h(x)) * x[i]
    - compute accuracy over entire training data = (# predicted correctly)/(# of training examples)

Review perceptron learning

- In the perceptron learning algorithm, where did the update equation come from?
- w[i] = w[i] + (y – h(x)) * x[i]
- Recall h(x) = w[0] * x[0] + w[1] * x[1] + ...
- If y = 1, but h(x) = 0, then h(x) is too small.
  - How do we increase h(x)?
  - Increase the weights w[0], w[1], ...
  - By how much?
  - Proportionally to their corresponding input x[i] value.

Review perceptron learning

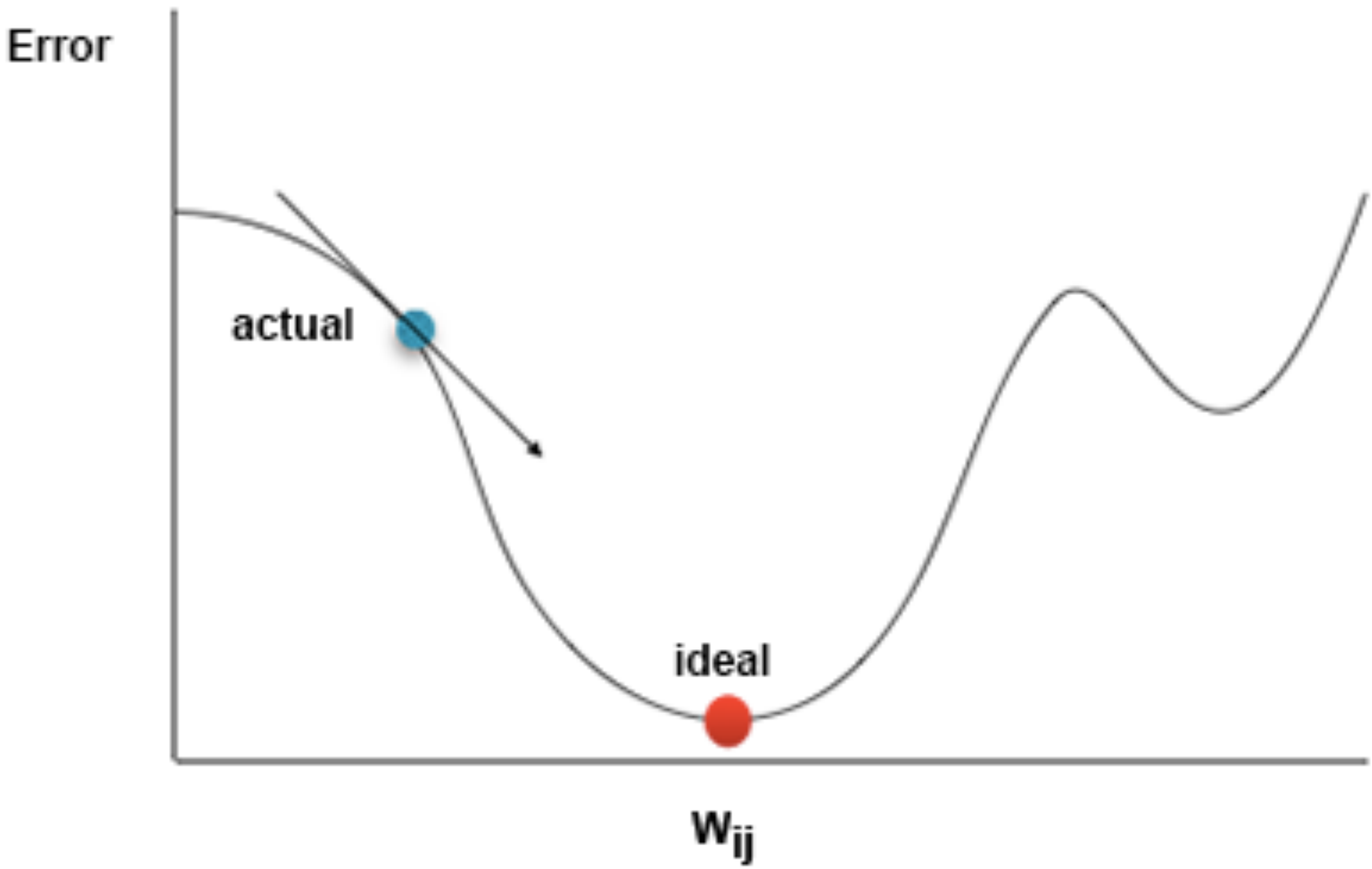
- Learning with a hard threshold function is rather erratic because of the binary nature of its output.
  - Threshold function is not differentiable (so what?)
  - Always outputs 0 or 1; doesn't give us gradations of predictions.
- Let's switch to the logistic/sigmoid function as our activation function.



Review perceptron learning

- How does the update equation change when our activation function is the logistic function rather than a hard threshold?
- Let's look at what errors look like under the hard threshold function vs the logistic function.

Gradient descent



Review perceptron learning

- Assuming a NN doesn't classify everything correctly, how do we measure how good or bad it is doing?
- Error functions:
  - L1-norm loss: measure $|y - h(x)|$
  - L2-norm loss: measure $(y - h(x))^2$
  - Clearly both loss functions penalize big differences more than small differences, but L1 penalizes proportionally, and L2 penalizes more harshly than L1

```
repeat
    for each weight w_{i,j} in network do
        w_{i,j} ← a small random number
    for each example (x, y) in examples do
        /* Propagate the inputs forward to compute the outputs */
        for each node i in the input layer do
            a_i ← x_i
        for ℓ = 2 to L do
            for each node j in layer ℓ do
                in_j ← Σ_i w_{i,j} a_i
                a_j ← g(in_j)
        /* Propagate deltas backward from output layer to input layer */
        for each node j in the output layer do
            Δ[j] ← g′(in_j) × (y_j − a_j)
        for ℓ = L − 1 to 1 do
            for each node i in layer ℓ do
                Δ[i] ← g′(in_i) Σ_j w_{i,j} Δ[j]
        /* Update every weight in network using deltas */
        for each weight w_{i,j} in network do
            w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
until some stopping criterion is satisfied
return network
```

Backprop highlights

repeat
    for each weight $w_{i,j}$ in *network* **do**
        $w_{i,j} \leftarrow$ a small random number
    for each example $(\mathbf{x}, \mathbf{y})$ **in** *examples* **do**
        /* *Propagate the inputs forward to compute the outputs* */
        for each node i in the input layer **do**
            $a_i \leftarrow x_i$
        for $\ell = 2$ **to** L **do**
            for each node j in layer ℓ **do**
                $in_j \leftarrow \sum_i w_{i,j} a_i$
                $a_j \leftarrow g(in_j)$

Backprop highlights

/ Propagate deltas backward from output layer to input layer */*

for each node j in the output layer **do**

  $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$

for $\ell = L - 1$ **to** 1 **do**

  for each node i in layer ℓ **do**

    $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$

/ Update every weight in network using deltas */*

for each weight $w_{i,j}$ in *network* **do**

  $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$

Compare

- w[i] = w[i] + (y – h(x)) * x[i]

$$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$$

$$\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$$

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$$

History

- 1943 – McCullough-Pitts neuron (can't be trained)
- 1958 – Rosenblatt's perceptron (can be trained)
- 1969 – Minsky and Papert publish *Perceptrons*, which explains the limits of single-layer NNs.
  - Ushers in first "AI Winter"
- 1982 – Backprop algorithm for NNs is published.
  - Was known in the 60s! AI Winter eliminated a lot of AI funding and people were discouraged from working on AI projects.
- 1980s – NNs rise again!
- 1989 – NNs are "universal approximators."

History

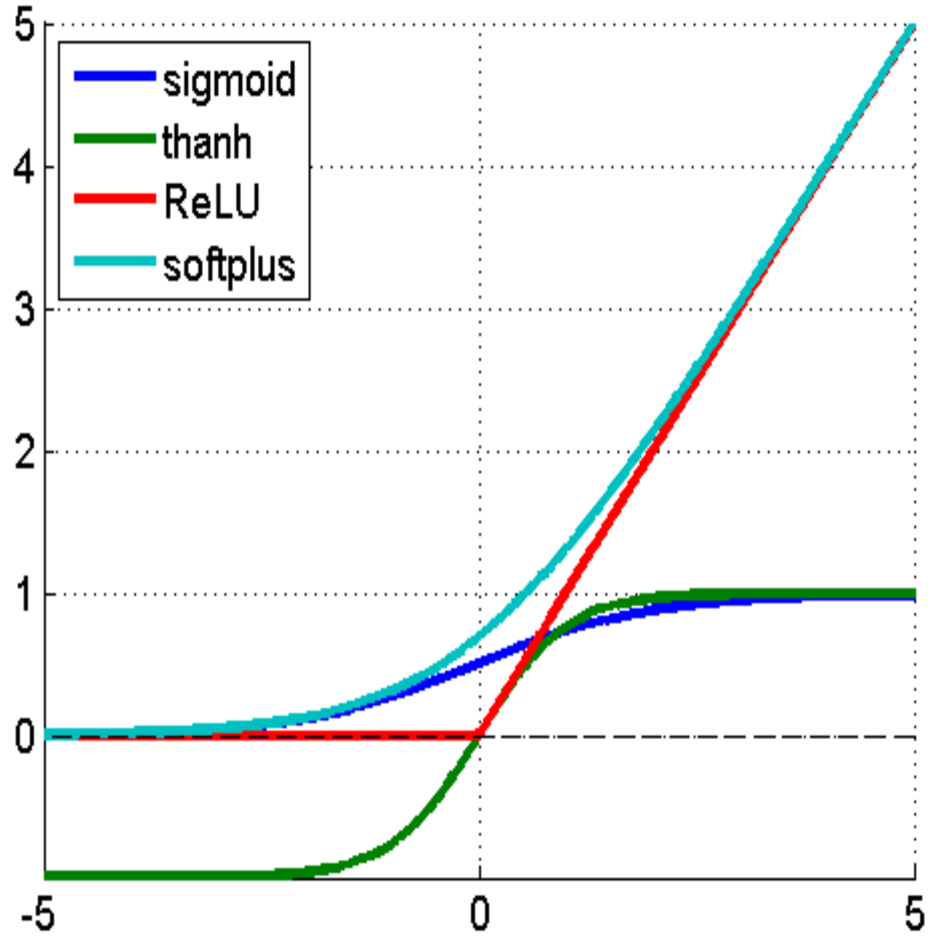
- 1989 – Convolutional NN used to do handwritten digit recognition for ZIP codes. (Yann LeCun)
- 1990s – NNs start to be seen as "painfully slow." Takes a long time to train them. At the same time, people start making more and more modifications to make NNs predict things better – adding more layers, making them recurrent etc.
- Mid 90s – 2nd AI Winter occurs when everything breaks down and the community loses faith in NNs (too slow, too hard to train with backprop, don't work well, nobody understands them anyway).
  - Move to other models, especially probabilistic.

History

- Winter continues through early 2000s, though some people continue working on NNs.
- 2006 paper: "A fast learning algorithm for deep belief nets"
  - Key idea – don't initialize weights randomly. Start off with a round of unsupervised learning to find reasonable initial values for the weights, then finish with regular supervised learning.
- 2nd key idea – pure computational power of GPUs.
  - Massively parallel! 70x faster than training on CPUs.
- 3rd key idea – huge data sets.

History

- 2010 – Turns out the activation function used makes a huge difference on training time and performance.



Lessons

- Our labeled datasets were thousands of times too small.
- Our computers were millions of times too slow.
- We initialized the weights in a stupid way.
- We used the wrong type of non-linearity.