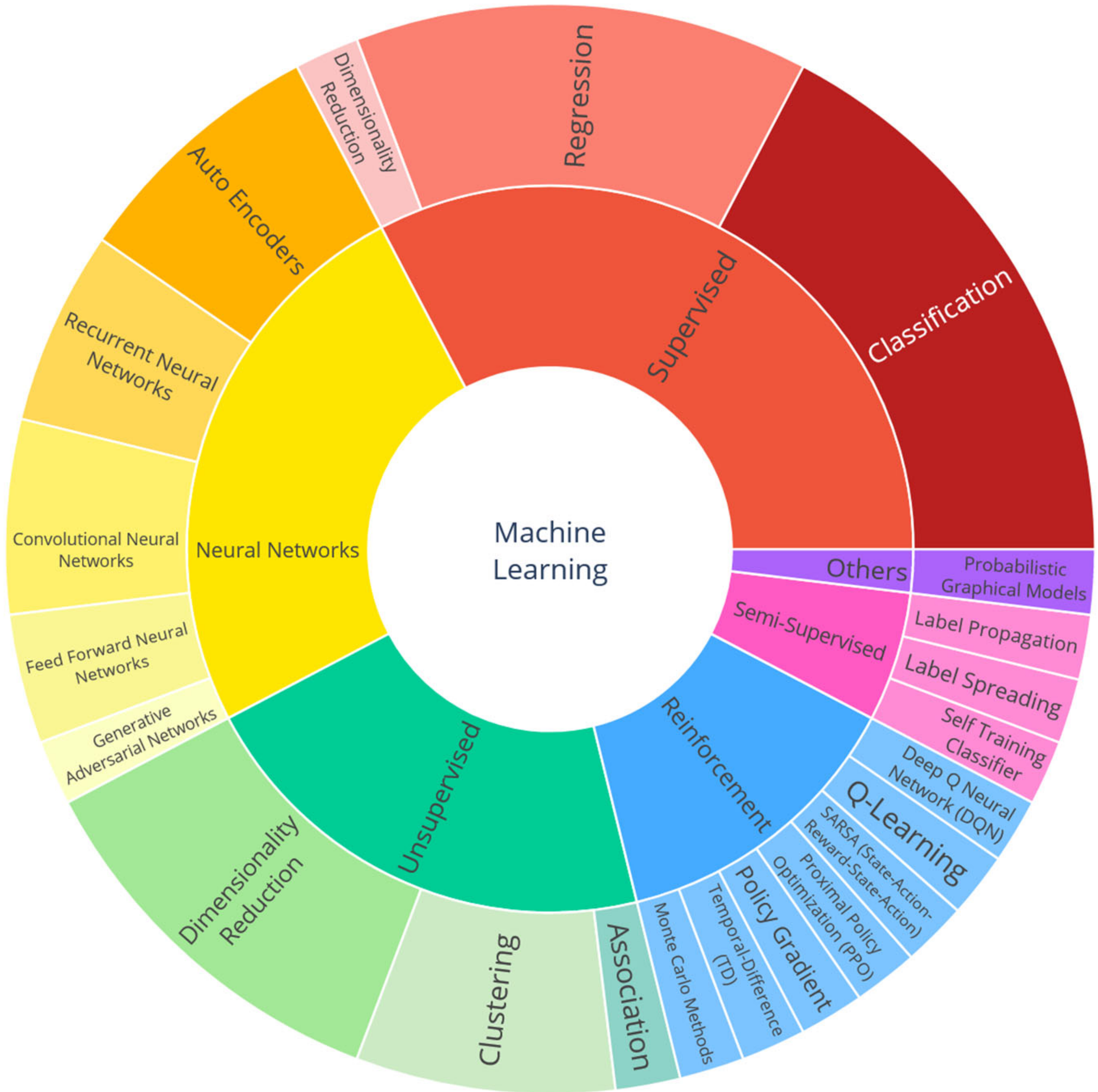
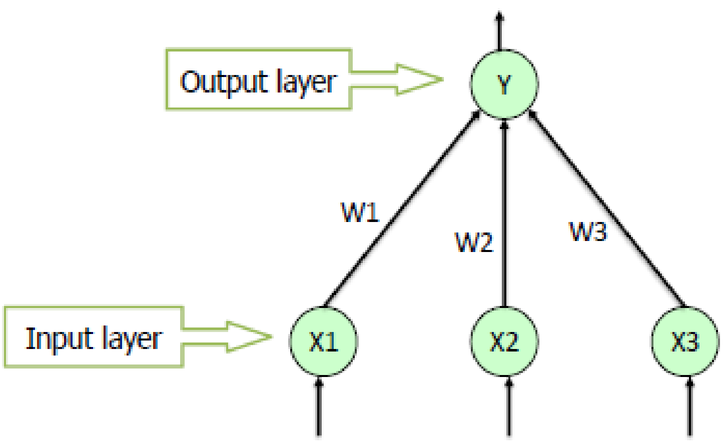
Introduction to Machine Learning

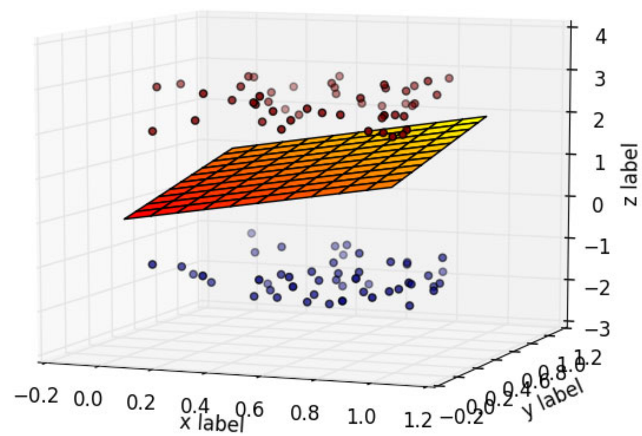
Neural Networks


Machine Learning
Supervised
Regression
Classification
Dimensionality Reduction
Neural Networks
Auto Encoders
Recurrent Neural Networks
Convolutional Neural Networks
Feed Forward Neural Networks
Generative Adversarial Networks
Unsupervised
Dimensionality Reduction
Clustering
Association
Reinforcement
Monte Carlo Methods
Temporal-Difference (TD)
Policy Gradient
Proximal Policy Optimization (PPO)
SARSA (State-Action-Reward-State-Action)
Q-Learning
Deep Q Neural Network (DQN)
Semi-Supervised
Label Propagation
Label Spreading
Self Training Classifier
Others
Probabilistic Graphical Models

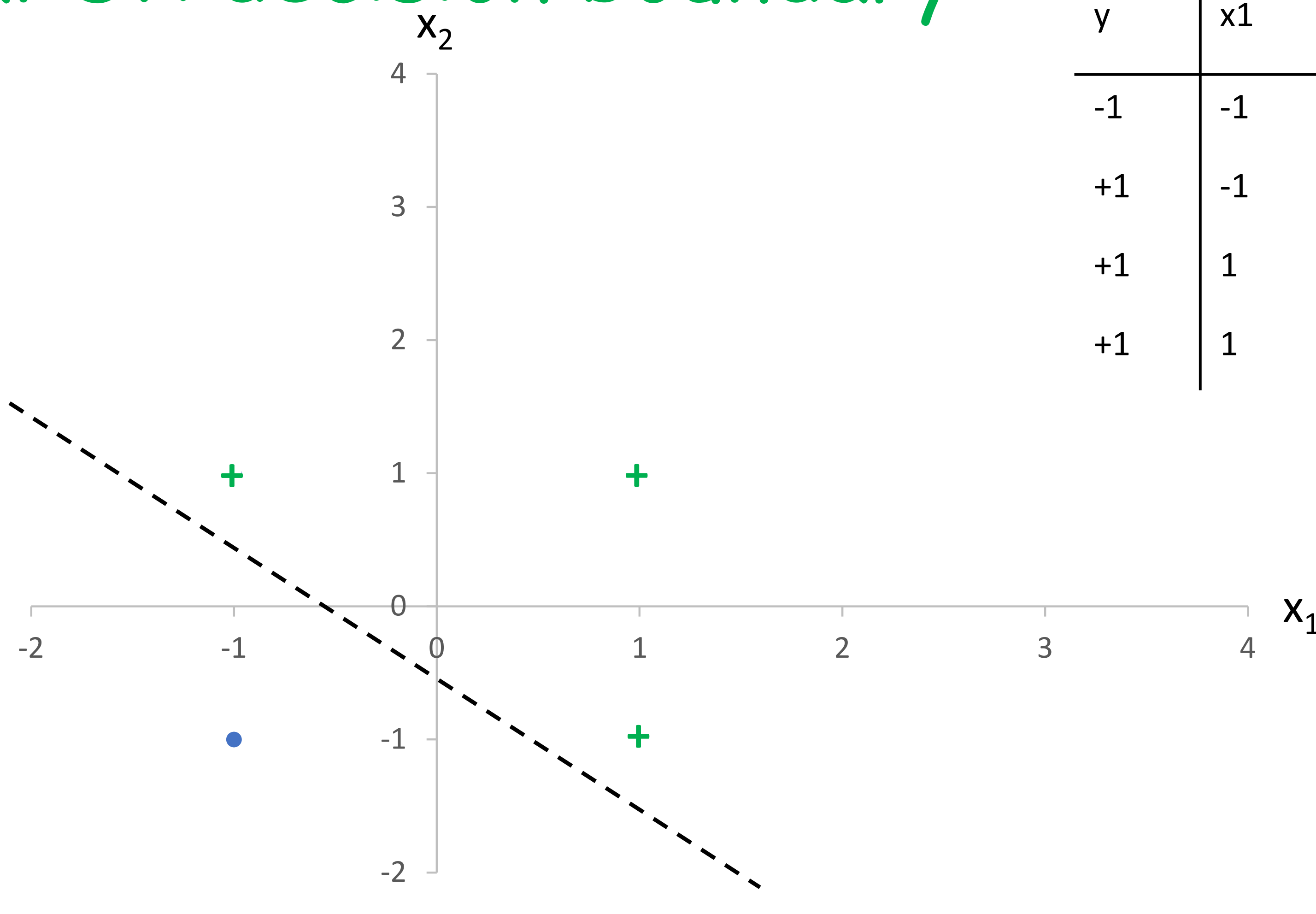
Linearly separable

- If the classes can be separated by a hyperplane, then they are linearly separable



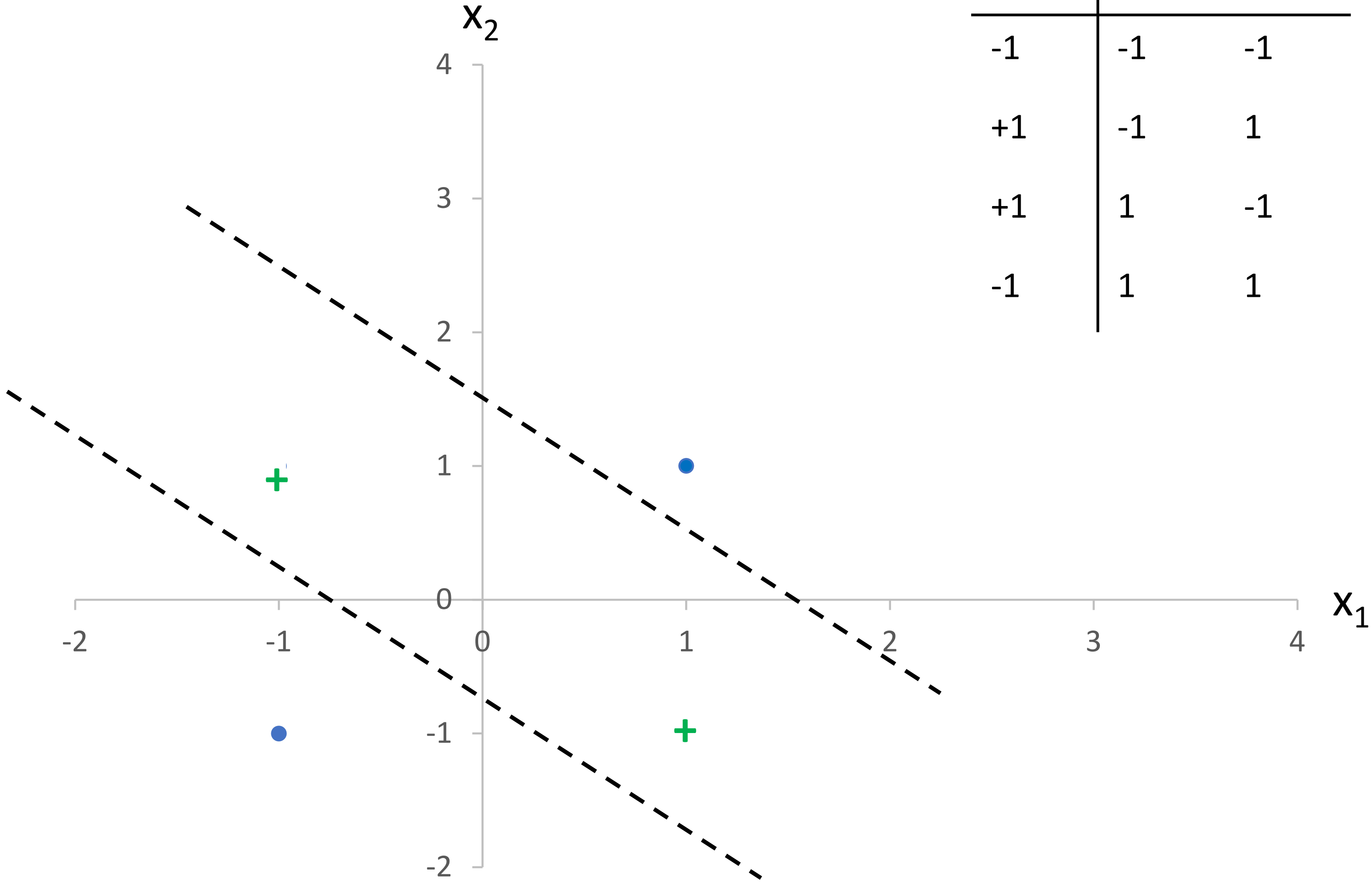


Logical OR decision boundary



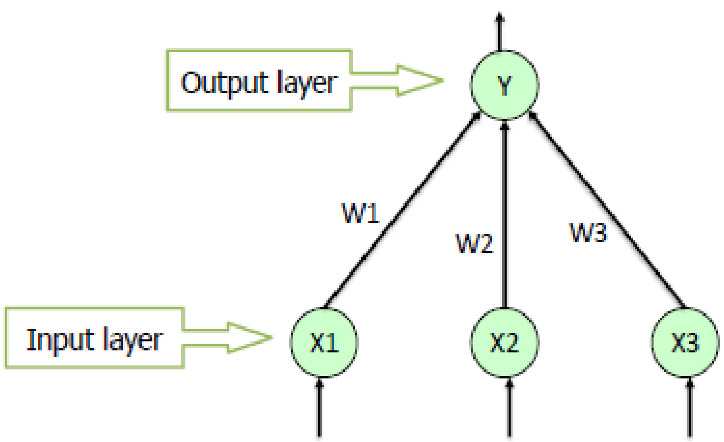
y	x1	X2
-1	-1	-1
+1	-1	1
+1	1	-1
+1	1	1

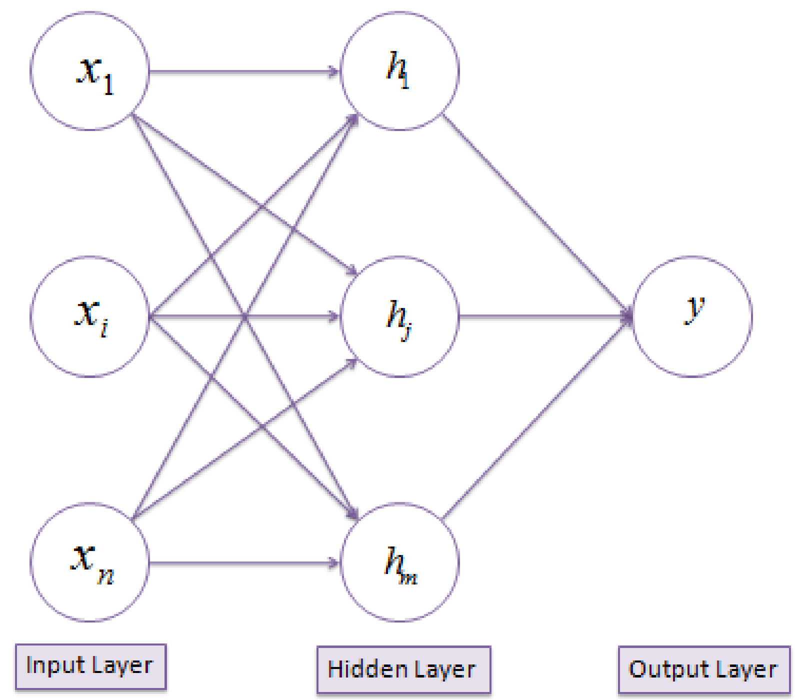
XOR decision boundary



y	x_1	X_2
-1	-1	-1
+1	-1	1
+1	1	-1
-1	1	1

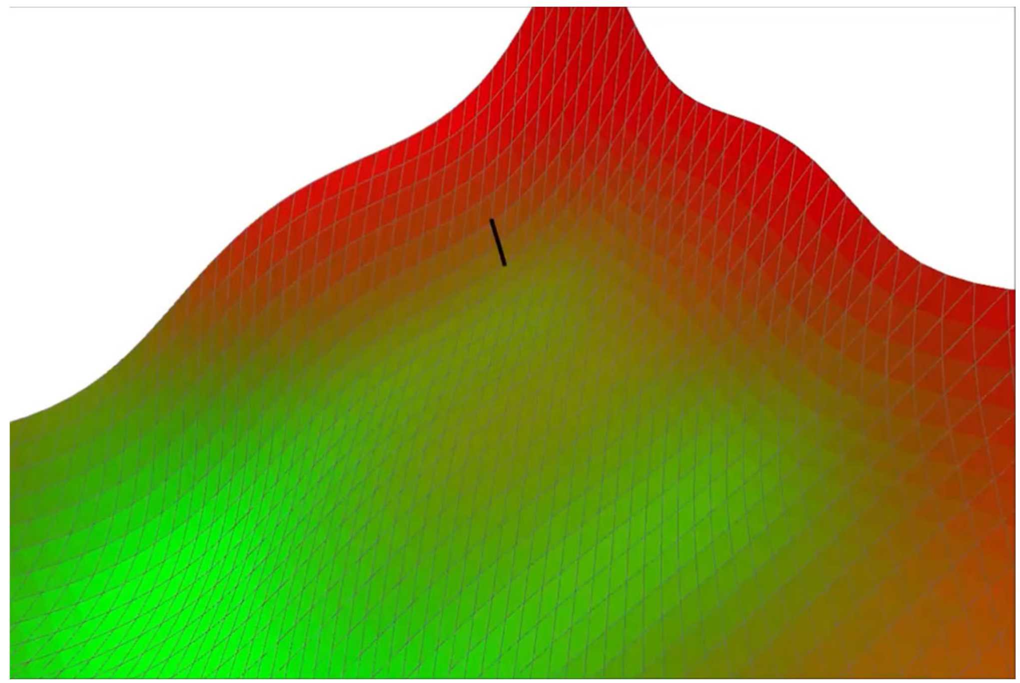
Adding layers





Activation function and gradient descent

$$h_i = f\left(w_i \cdot x\right)$$



Algorithm 21 $\textsc{GradientDescent}(\mathcal{F}, K, \eta_1, \dots)$

$z^{(0)} \leftarrow \langle 0, 0, \dots, 0 \rangle$ // initialize variable we are optimizing

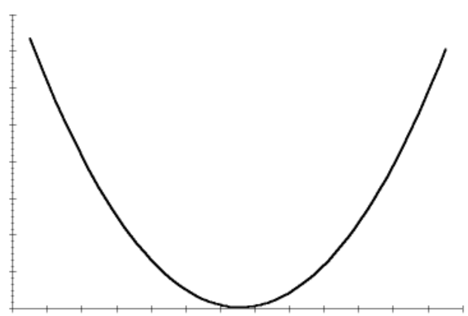
for $k = 1 \dots K$ **do**

  $g^{(k)} \leftarrow \nabla_z \mathcal{F}|_{z^{(k-1)}}$ // compute gradient at current location

  $z^{(k)} \leftarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$ // take a step down the gradient

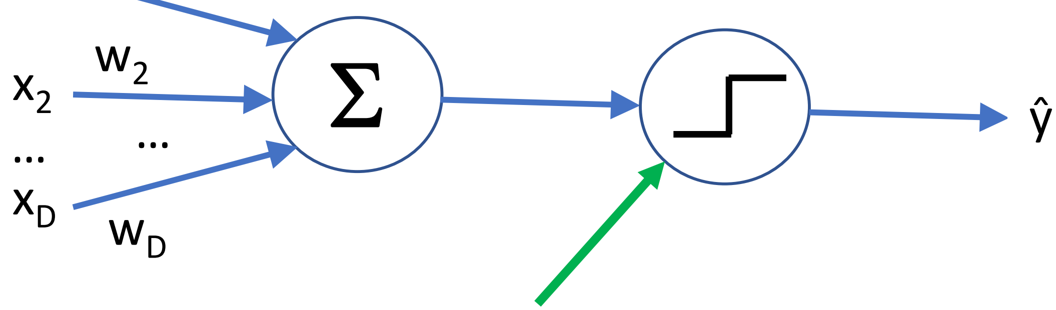
end for

return $z^{(K)}$



Perceptron neuron

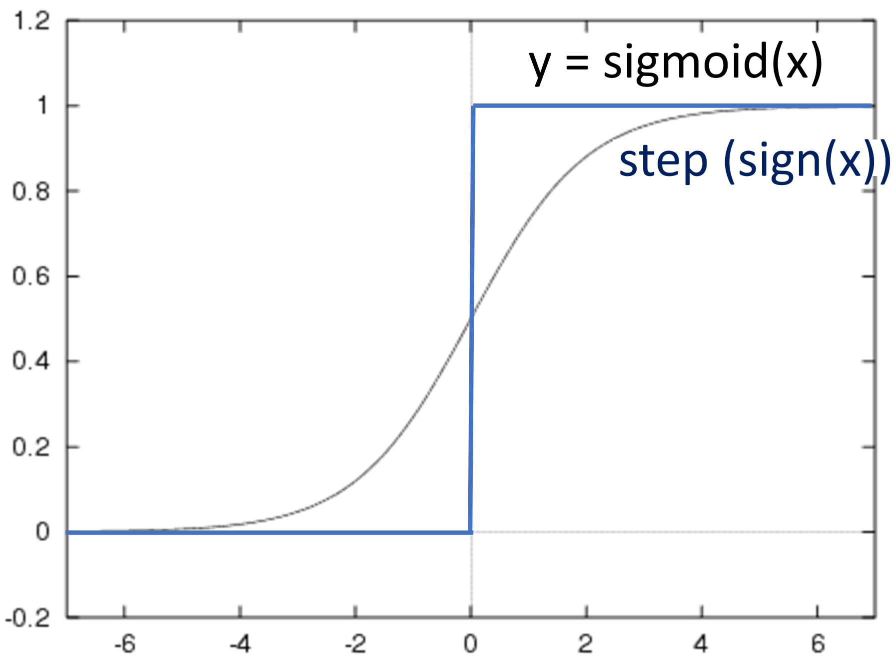
$$a = \sum_{d=1}^{D} w_d \, x_d$$



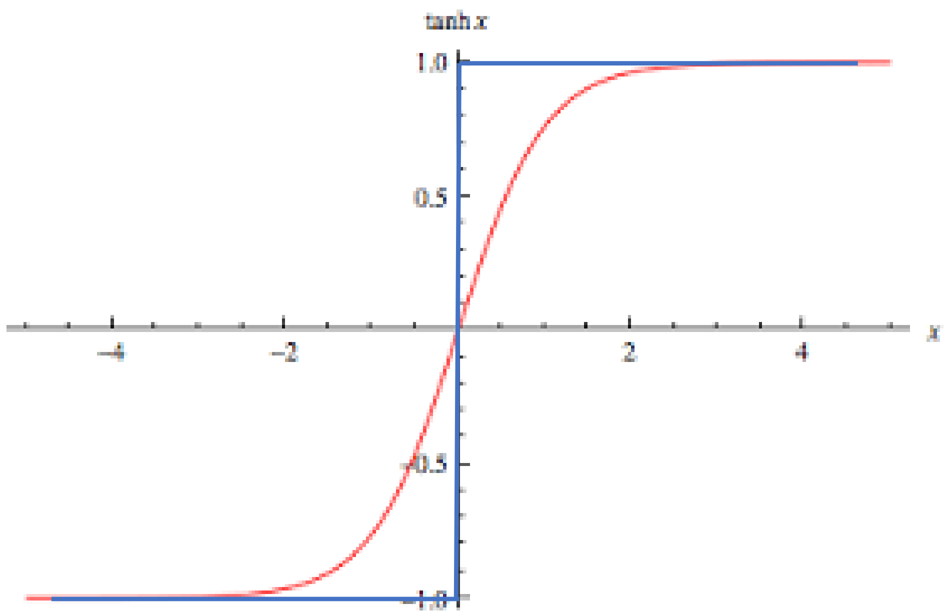
Not differentiable!

derivative = infinity here

Alternatives that are differentiable



We will use this one, tanh(x)



Derivative is $1 - tanh^2$(x)

Algorithm 25 TwoLayerNetworkPredict($\mathbf{W}$, $\boldsymbol{v}$, $\hat{\boldsymbol{x}}$)

for $i = 1$ **to** *number of hidden units* **do**

    $h_i \leftarrow \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}})$ // compute activation of hidden unit i

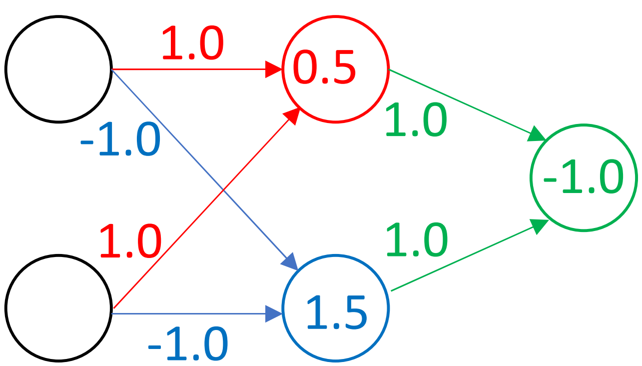
end for

return $\boldsymbol{v} \cdot \boldsymbol{h}$ // compute output unit

- W is matrix of weights from input nodes to hidden nodes
- v is vector of weights from hidden nodes to output node
- Note the different way the hidden value is computed in comparison with the output value

XOR

y	x1	x2
-1	-1	-1
+1	-1	1
+1	1	-1
-1	1	1



Top hidden node computes "or", weights=1.0, bias=0.5.
Associate bias with each node (handled various ways, here listed in circle).

x_1=-1, x_2=-1, tanh(1*-1 + 1*-1 – 0.5) = -0.905 (-1)
x_1=-1, x_2=1, tanh(1*-1 + 1*1 – 0.5) = +0.462 (1)
x_1=1, x_2=-1, tanh(1*1 + 1*-1 – 0.5) = +0.462 (1)
x_1=1, x_2=1, tanh(1*1 + 1*1 – 0.5) = +0.987 (1)

Bottom hidden node computes "nand", weights=-1.0, bias=1.5.
x_1=-1, x_2=-1, tanh(-1*-1 + -1*-1 - 1.5) = +0.998 (1)
x_1=-1, x_2=1, tanh(-1*-1 + -1*1 - 1.5) = +0.905 (1)
x_1=1, x_2=-1, tanh(-1*1 + -1*-1 - 1.5) = +0.905 (1)
x_1=1, x_2=1, tanh(-1*1 + -1*1 - 1.5) = -0.462 (-1)

Output computes "and(or,nand)", weights=1.0, bias=1.0.
x_1=-1, x_2=-1, h = (-.905, .998) = -.907 (-1)
x_1=-1, x_2=1, h = (.462, .905) = .367 (1)
x_1=1, x_2=-1, h = (.462, .905) = .367 (1)
x_1=1, x_2=1, h = (.987, -.462) = -.475 (-1)

How big of a network do I need?

Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate *F* arbitrarily well. Namely, for all **x** in the domain of *F*, $|F(\mathrm{x}) - \hat{F}(\mathrm{x})| < \varepsilon$.

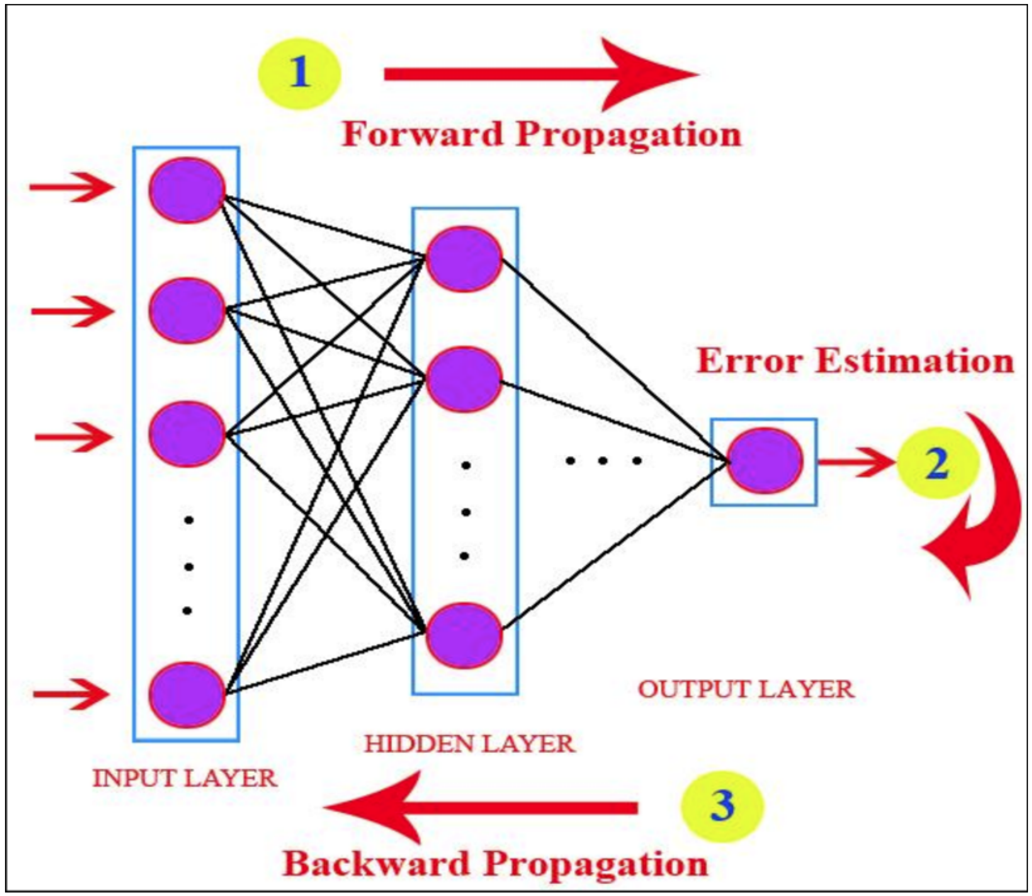
How many hidden units?

If D dimensions, K hidden units, then (D+2)K parameters

1 parameter for bias, 1 for weight to the output node, thus N = (D+2)K, N/(D+2)=K

If you want 1-2 examples for each parameter you are learning, then use

#hidden nodes = K = $\left\lfloor \frac{N}{D} \right\rfloor$


1
Forward Propagation
Error Estimation
2
INPUT LAYER
HIDDEN LAYER
OUTPUT LAYER
3
Backward Propagation

Backpropagation

- We know how to compute output
- How do we learn weights?

backpropagation = gradient descent + chain rule

Review: Linear Regression

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \bullet \frac{1}{2n} \sum_{i=1}^{n} (h(xi) - yi)^2$$

$$= \frac{\partial}{\partial \theta_j} \bullet \frac{1}{2n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^i - y^i)^2$$

Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \qquad \text{(for every } j\text{)}$$

}

Optimization function

- Based on loss function: Squared error

$$\min_{\mathbf{W},v} \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)\right)^2$$

Adjusting weights to output node

- Differentiate objective with respect to v

$$\min_{\mathbf{W},v} \quad \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)\right)^2$$

$$\nabla_v = -\sum_n e_n \boldsymbol{h}_n$$

Adjusting weights to hidden node

- Only compensate for *portion* of error for that hidden node
- Not trying to produce specific output value

Adjusting weights to hidden node

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(w_i \cdot x)\right)^2$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial w_i}$$

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(w_i \cdot x)\right) v_i = -e v_i$$

$$\frac{\partial f_i}{\partial w_i} = f'(w_i \cdot x)x$$

This is the gradient with respect to w_i

- (network error) * (weight from this hidden node to output node) * (derivative of activation function) * (feature value for this data point x)

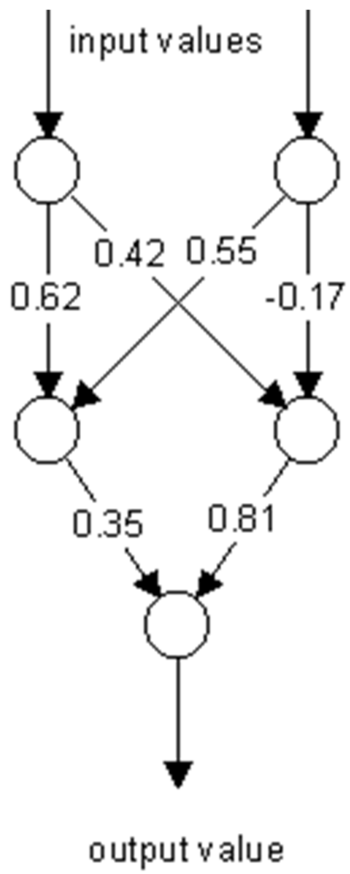
$$\nabla_{w_i} = -e v_i f'(w_i \cdot x)x$$

$= -e v_i(1 - \tanh^2(a_i))x$

Algorithm 26 TwoLayerNetworkTrain($\mathbf{D}$, η, K, $MaxIter$)

$\mathbf{W} \leftarrow D{\times}K$ matrix of small random values // initialize input layer weights
$v \leftarrow K$-vector of small random values // initialize output layer weights
for $iter = 1 \ldots MaxIter$ **do**
  $\mathbf{G} \leftarrow D{\times}K$ matrix of zeros // initialize input layer gradient
  $g \leftarrow K$-vector of zeros // initialize output layer gradient
  for all $(x,y) \in \mathbf{D}$ **do**
    for $i = 1$ **to** K **do**
      $a_i \leftarrow w_i \cdot \hat{x}$
      $h_i \leftarrow \tanh(a_i)$ // compute activation of hidden unit i
    end for
    $\hat{y} \leftarrow v \cdot h$ // compute output unit
    $e \leftarrow y - \hat{y}$ // compute error
    $g \leftarrow g - eh$ // update gradient for output layer
    for $i = 1$ **to** K **do**
      $\mathbf{G}_i \leftarrow \mathbf{G}_i - ev_i(1 - \tanh^2(a_i))x$ // update gradient for input layer
    end for
  end for
  $\mathbf{W} \leftarrow \mathbf{W} - \eta\mathbf{G}$ // update input layer weights
  $v \leftarrow v - \eta g$ // update output layer weights
end for
return $\mathbf{W}$, v

Example



Example Input $\rightarrow$ Target: 0 1 $\rightarrow$ 0; Assume learning rate=1.0, ignore bias

Forward propagation:
Dot product for left hidden node = 0*0.62 + 1*0.55 = 0.55
Dot product for right hidden node = 0*0.42 + 1*-0.17 = -0.17
h(left) = tanh(.55) = .5005;   h(right) = tanh(-.17) = -.1684
output = 0.35*.5005 + 0.81*-.1684 = 0.039;   error = 0 – 0.039 = -0.039

Backward propagation:
g -= error * h, g(left) -= -0.039*.5005 =.0195; v(left)-=0.35*.0195 = 0.33
g(right) -= -0.039 *-.1684 = -.007; v(right)-=0.81+.007 = 0.82
W(left_to_left) +=-.039*.35*(1-tanh^2(.55))0 = .62+0 (W = 0.62, *the same*)
W(left_to_right) +=-.039*.81*(1-tanh^2(-.17))0=.42+0 (W = 0.42, *the same*)
W(right_to_left)+=-.039*.35*(1-tanh^2(.55))1 = 0.55 + (0.39*.35*.75*1) = .52;
W(right_to_right)+=-.039*.81*(1-tanh^2(-.17))1 = -.17 +(0.39*.81*.97*1)= -.20

Let's try this out

Practical issues

- Initializing weights

- Why not initialize to 0? Can get stuck in local optimum.

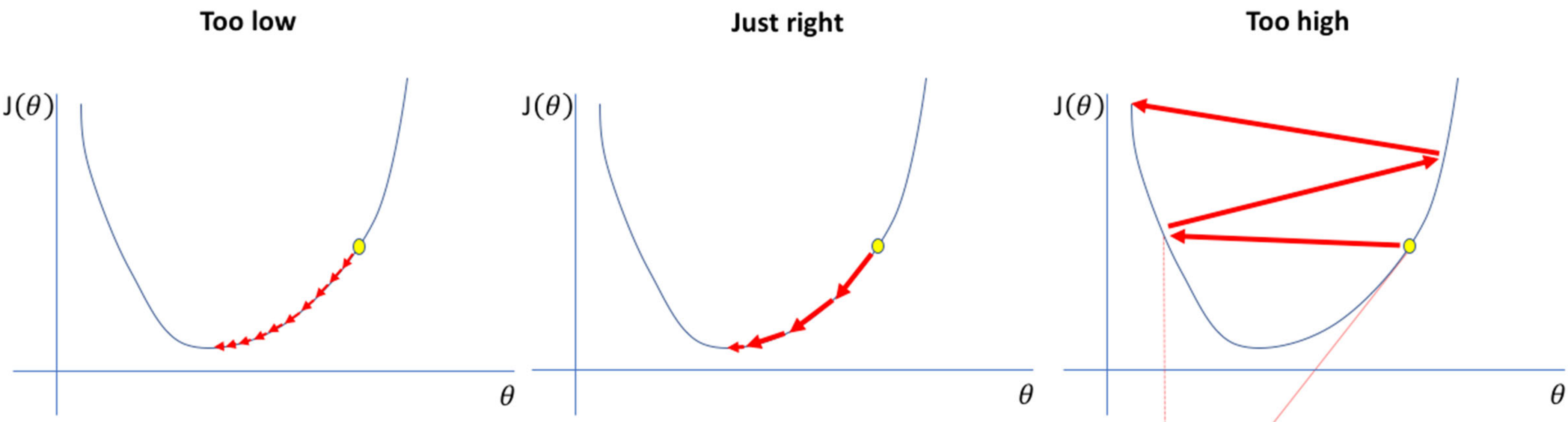
- If W=0 and v=0 then the activation h_i of hidden units will all be 0 (because W=0). Because h will be 0 e*h will be 0 so the weights v will not change.

- Note if sigmoid is used instead of tanh then activation will be non-zero so weights will change but they will all change identically and hidden unit values will always be the same. The model will eventually converge but hidden nodes are ignored.

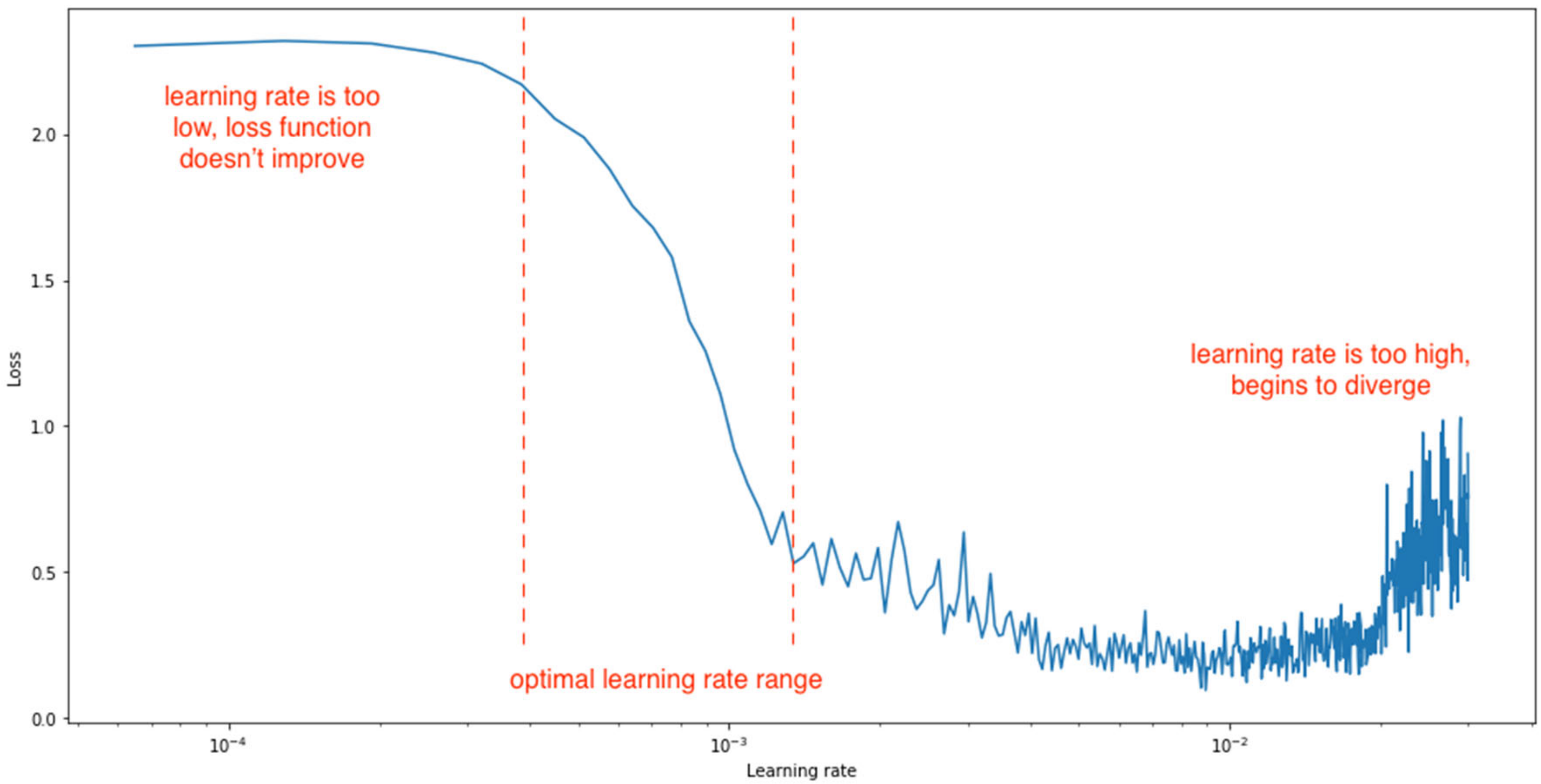
- Moral: neural networks are sensitive to their initialization. Random (small) initialization is most effective.

$\mathbf{W} \leftarrow D{\times}K$ matrix of small random values
$v \leftarrow K$-vector of small random values
for $iter = 1 \ldots MaxIter$ **do**
    $\mathbf{G} \leftarrow D{\times}K$ matrix of zeros
    $g \leftarrow K$-vector of zeros
    for all $(x,y) \in \mathbf{D}$ **do**
        for $i = 1$ **to** K **do**
            $a_i \leftarrow w_i \cdot \hat{x}$
            $h_i \leftarrow \tanh(a_i)$
        end for
        $\hat{y} \leftarrow v \cdot h$
        $e \leftarrow y - \hat{y}$
        $g \leftarrow g - eh$
        for $i = 1$ **to** K **do**
            $\mathbf{G}_i \leftarrow \mathbf{G}_i - ev_i(1 - \tanh^2(a_i))x$
        end for
    end for
    $\mathbf{W} \leftarrow \mathbf{W} - \eta\mathbf{G}$
    $v \leftarrow v - \eta g$
end for
return $\mathbf{W}, v$

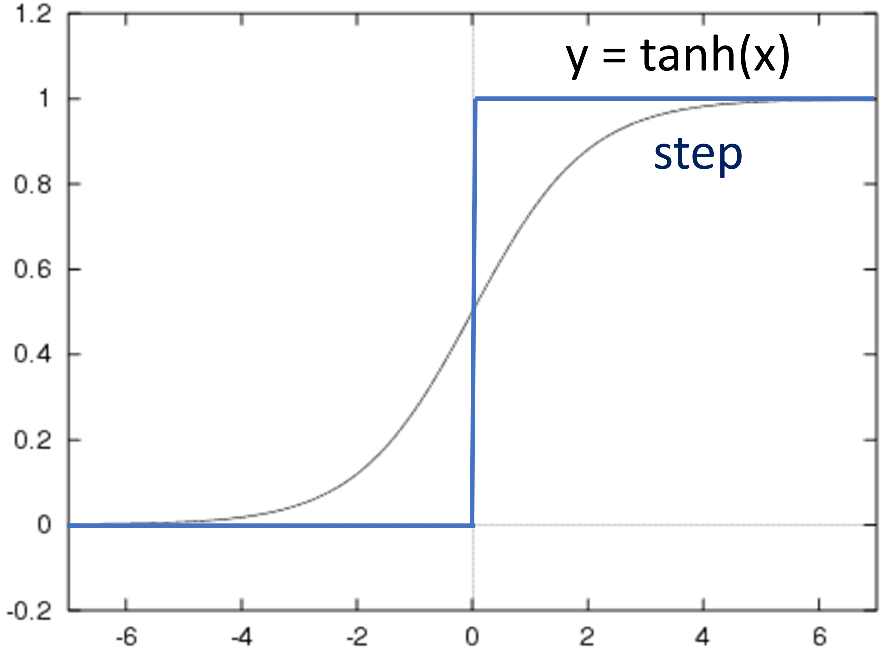
Learning rate



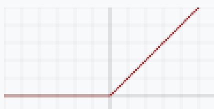
Learning rate

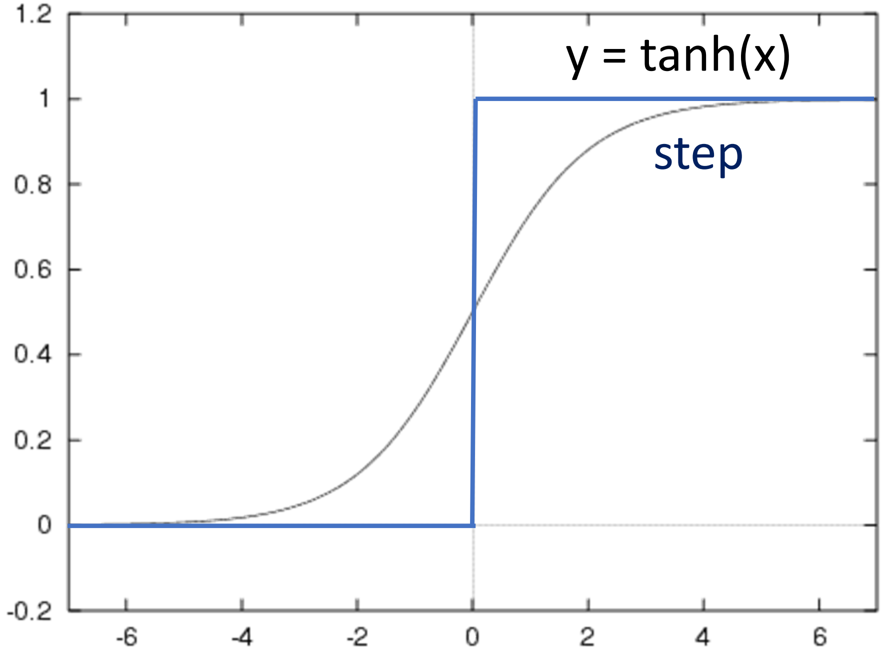


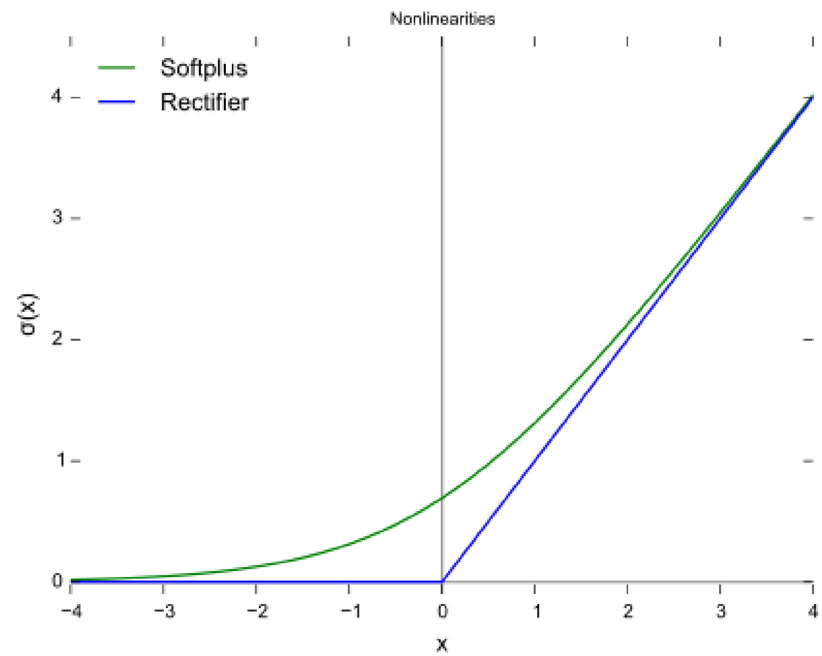
Activation function



ReLU

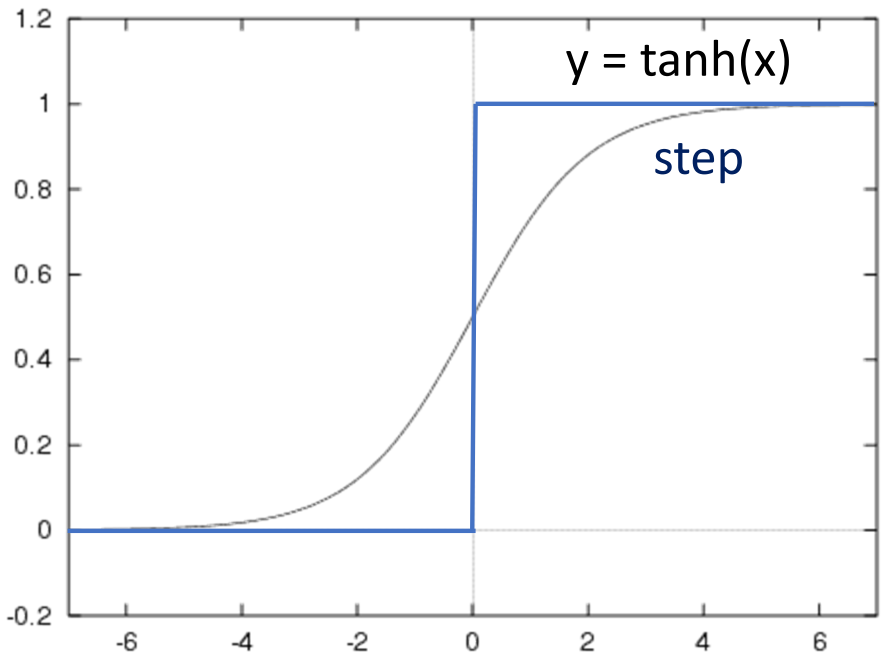
	$f(x) = \begin{cases} 0 & \text{for } x \le 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$	$f'(x) = \begin{cases} 0 & \text{for } x \le 0 \\ 1 & \text{for } x > 0 \end{cases}$	$[0, \infty)$

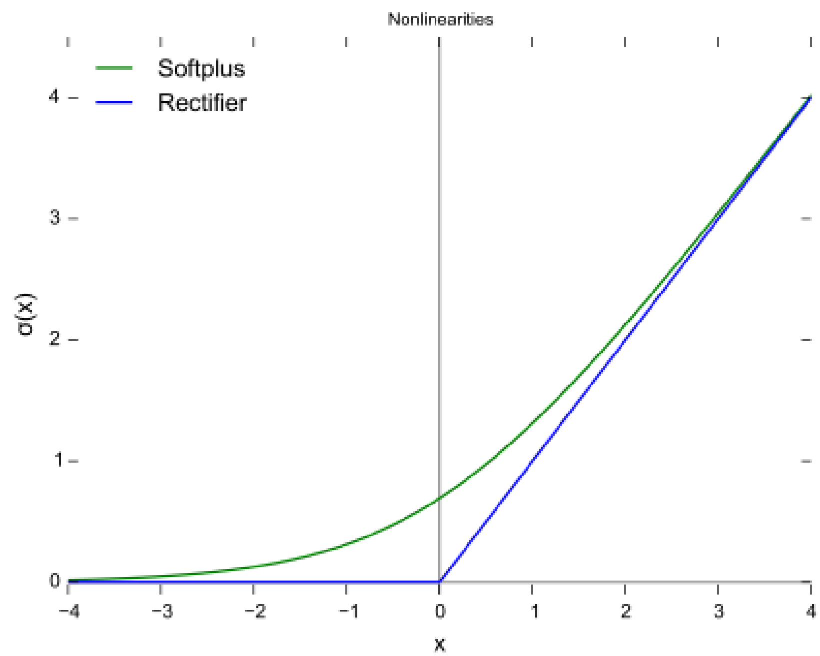




Softplus

$$f(x) = \log(1 + e^{x})$$





Hyperparameters

Practical issues

- Multiple classes
- Softmax

Class	Probability
apple	0.001
bear	0.040
candy	0.008
dog	0.950
egg	0.001