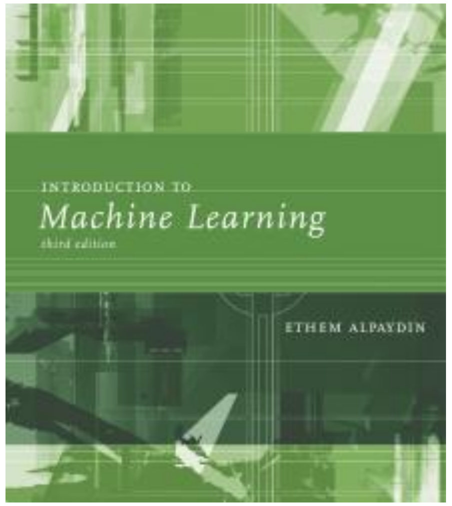


Neural Networks

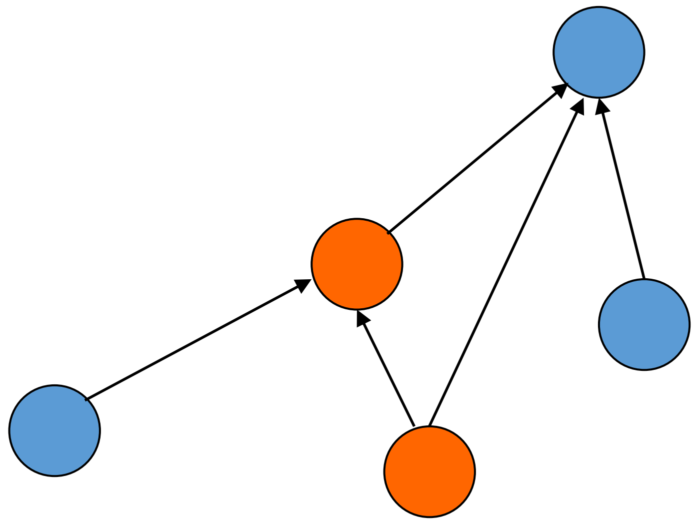
Lecture notes by Ethem Alpaydın
Introduction to Machine Learning (Boğaziçi Üniversitesi)

Lecture notes by Kevyn Collins-Thompson
Applied Machine Learning (Coursera)

Lecture notes by Andrew NG
Machine Learning by Stanford University (Coursera)

Neural Networks

- Networks of processing units (neurons) with connections (synapses) between them
- Large number of neurons: 10^{10}
- Large connectitivity: 10^5
- Parallel processing
- Distributed computation/memory
- Robust to noise, failures



Neural Networks

- Origins: Algorithms that try to mimic the brain.
- Was very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications

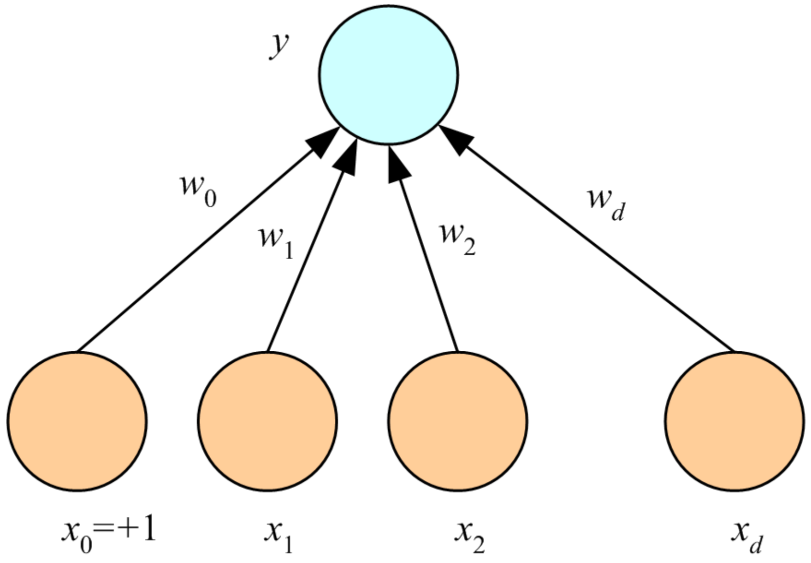
Perceptron

- The *perceptron* is the basic processing element.
- It has inputs that may come from the environment or may be the outputs of other perceptrons

Perceptron

- Associated with each input, $x_j \in \mathcal{R}, j = 1, \dots, d$ is a *connection weight*, or *synaptic weight* $w_j \in \mathcal{R}$, and the output, y, in the simplest case is a weighted sum of the inputs

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$



Perceptron

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$

w_0 is the intercept value to make the model more general; it is generally bias unit modeled as the weight coming from an extra *bias unit*, x_0, which is always +1.

Then, we can write $y = w^T X$

Perceptron

$$y = w^T X$$

where $w = [w_0, w_1, \dots, w_d]^T$ and $X = [1, x_1, \dots, x_d]^T$ are *augmented* vectors to include also the bias weight and input.

Perceptron

- During testing, with given weights, w, for input X, we compute the output y.
- To implement a given task, we need to *learn* the weights w, the parameters of the system, such that correct outputs are generated given the inputs.

What a Perceptron Does

- Regression
- Classification

What a Perceptron Does

Regression

When *d* = 1 and *x* is fed from the environment through an input unit, we have

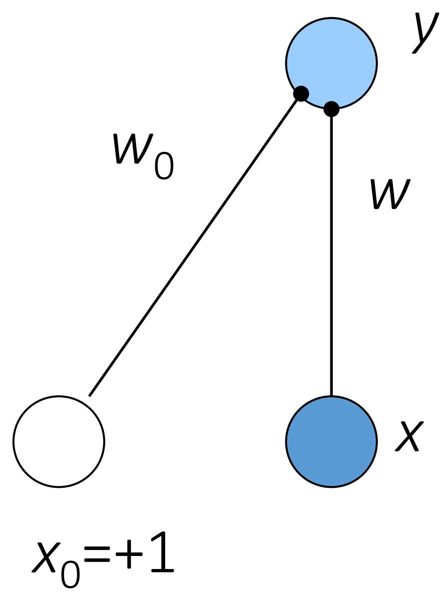
$$y = wx + w_0$$

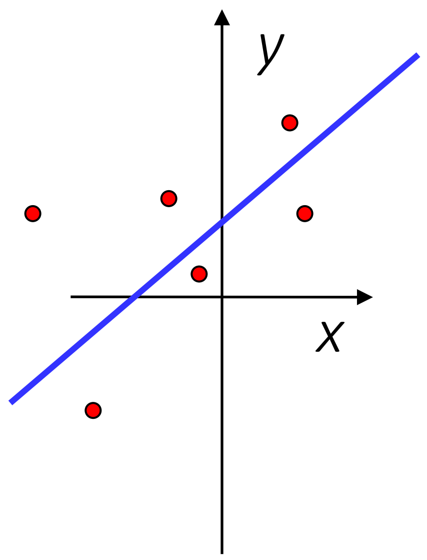
which is the equation of a line with w as the slope and w_0 as the intercept.

What a Perceptron Does

Regression

Thus this perceptron with one input and one output can be used to implement a linear fit.





What a Perceptron Does

Regression

- With more than one input, the line becomes a (hyper)plane, and the perceptron with more than one input can be used to implement multivariate linear fit.
- Given a sample, the parameters w_j can be found by regression

What a Perceptron Does

Classification

The perceptron defines a hyperplane and as such can be used to divide the input space into two: the half-space where it is positive and the half-space where it is negative.

By using it to implement a linear discriminant function (logistic regression), the perceptron can separate two classes by checking the sign of the output.

What a Perceptron Does

Classification

If we define *s(·)* as the *threshold function*

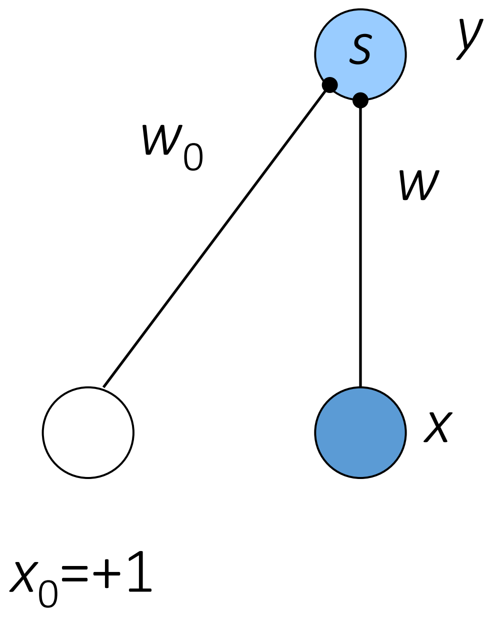
$$s(a) = \begin{cases} 1 & if\ a > 0 \\ 0 & otherwise \end{cases}$$

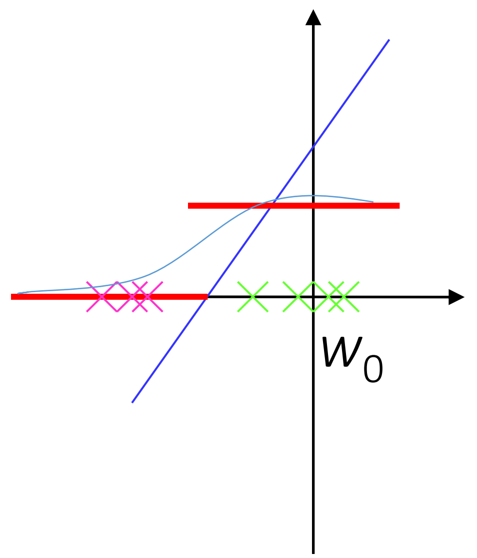
Then we can

choose $\begin{cases} C_1 & if\ s(w^T X) > 0 \\ C_2 & otherwise \end{cases}$

What a Perceptron Does

Classification

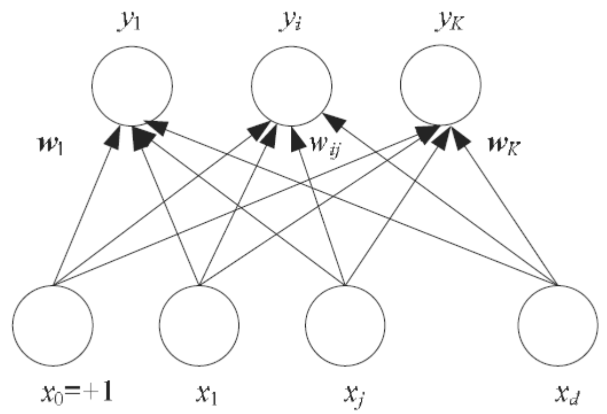




$$y = \text{sigmoid}(o) = \frac{1}{1 + \exp\left[-\mathbf{w}^T \mathbf{x}\right]}$$

K Outputs

When there are *K* > 2 outputs, there are *K* perceptrons, each of which has a weight vector w_i



$$y_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} = \mathbf{w}_i^T \mathbf{x}$$

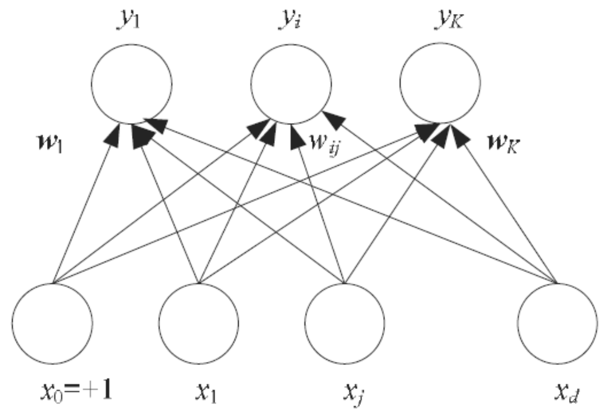
$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

where w_{ij} is the weight from input x_j to output y_i.

W is the $K * (d+1)$ weight matrix of w_{ij} whose rows are the weight vectors of the *K* perceptrons.

K Outputs

W is the $K * (d+1)$ weight matrix of w_{ij} whose rows are the weight vectors of the K perceptrons.



$$\mathrm{W} = \begin{bmatrix} w_{10} & \cdots & w_{1d} \\ \vdots & \ddots & \vdots \\ w_{K0} & \cdots & w_{Kd} \end{bmatrix}$$

$$\mathrm{y} = \mathrm{Wx} = \begin{bmatrix} w_{10} & \cdots & w_{1d} \\ \vdots & \ddots & \vdots \\ w_{K0} & \cdots & w_{Kd} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_d \end{bmatrix}$$

K Outputs

When used for classification, during testing, we

choose $C_i \ \ if\ y_i = \max_k y_k$

K Outputs

use the softmax if we need the posterior probabilities

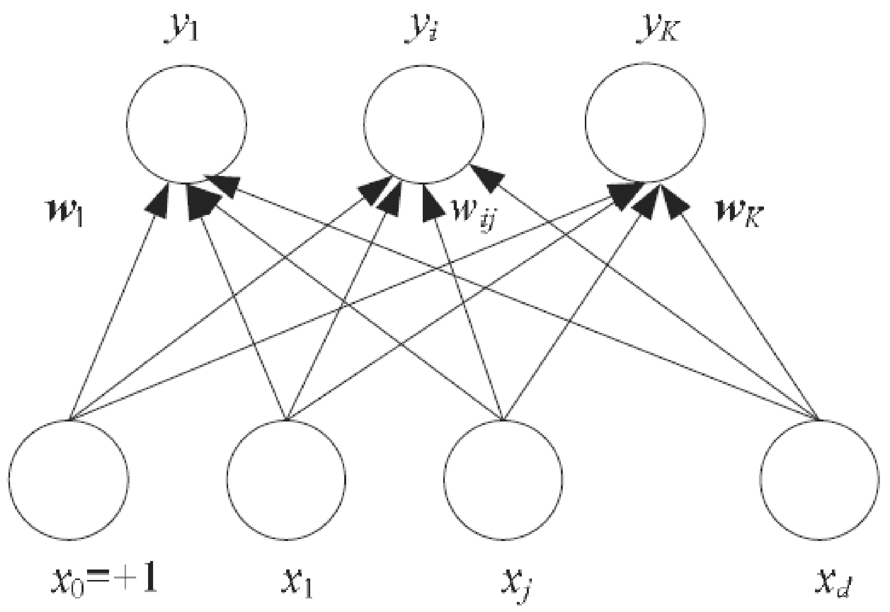
- calculate the weighted sums
- calculate the softmax values

$$o_i = \mathbf{w}_i^T \mathbf{x}$$

$$y_i = \frac{\exp o_i}{\sum_k \exp o_k}$$

choose C_i

if $y_i = \max_k y_k$



Training

In training neural networks, we generally use online learning where we are *not given the whole sample*, but we are given instances *one by one* and would like the network to update its parameters after each instance adapting itself slowly in time.

Training

- Online (instances seen one by one) vs batch (whole sample) learning:
  - No need to store the whole sample
  - Problem may change in time
  - Wear and degradation in system components

Training

- In *online learning*, we do not write the error function over the whole sample but on individual instances.
- Starting from random initial weights, at each iteration we adjust the parameters a little bit to minimize the error, without forgetting what we have previously learned.
- If this error function is differentiable, we can use gradient descent.

Training - Regression

The error on the single instance pair with index $t, (x^t, r^t)$ is $(j = 1, \dots, d)$

$$E^t\left(\mathbf{w} \mid \mathbf{x}^t, r^t\right) = \frac{1}{2}\left(r^t - y^t\right)^2 = \frac{1}{2}\left[r^t - \left(\mathbf{w}^T \mathbf{x}^t\right)\right]^2$$

$$\Delta w_j^t = \eta\left(r^t - y^t\right)x_j^t$$

where η is the learning rate.

This is known as ***stochastic gradient descent***

Training - Classification

Update rules can be derived for classification problems using logistic discrimination where updates are done after each pattern, instead of summing them and doing the update after a complete pass over the training set

Training - Classification

- Single sigmoid output

$$y^t = \text{sigmoid}\left(\mathbf{w}^T \mathbf{x}^t\right)$$
$$E^t\left(\mathbf{w} \mid \mathbf{x}^t, \mathbf{r}^t\right) = -r^t \log y^t - \left(1 - r^t\right) \log\left(1 - y^t\right)$$

Cross-entropy

$$\Delta w_j^t = \eta\left(r^t - y^t\right) x_j^t$$

- *K*>2 softmax outputs

$$y^t = \frac{\exp \mathbf{w}_i^T \mathbf{x}^t}{\sum_k \exp \mathbf{w}_k^T \mathbf{x}^t} \qquad E^t\left(\{\mathbf{w}_i\}_i \mid \mathbf{x}^t, \mathbf{r}^t\right) = -\sum_i r_i^t \log y_i^t$$
$$\Delta w_{ij}^t = \eta\left(r_i^t - y_i^t\right) x_j^t$$

Training

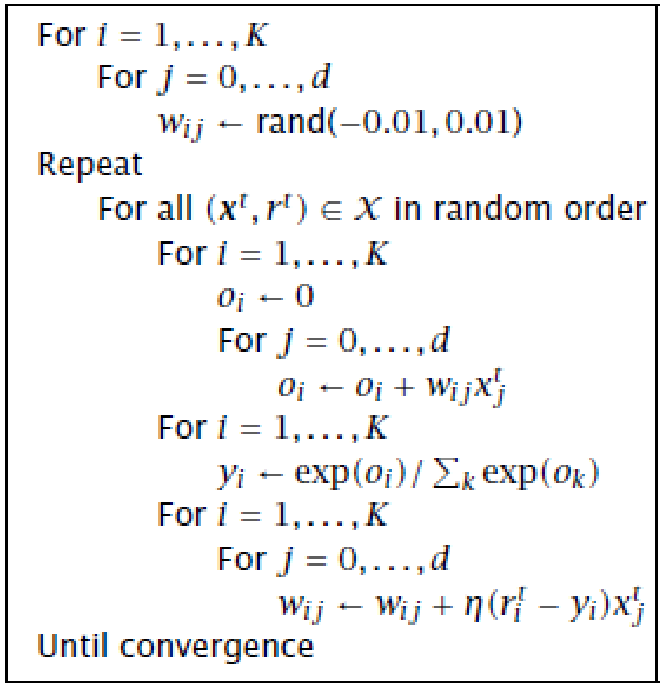
- Stochastic gradient-descent: Update after a single pattern
- Generic update rule (LMS rule):

$$\Delta w_{ij}^t = \eta \left(r_i^t - y_i^t \right) x_j^t$$

$$\text{Update} = \text{LearningFactor} \cdot (\text{DesiredOutput} - \text{ActualOutput}) \cdot \text{Input}$$

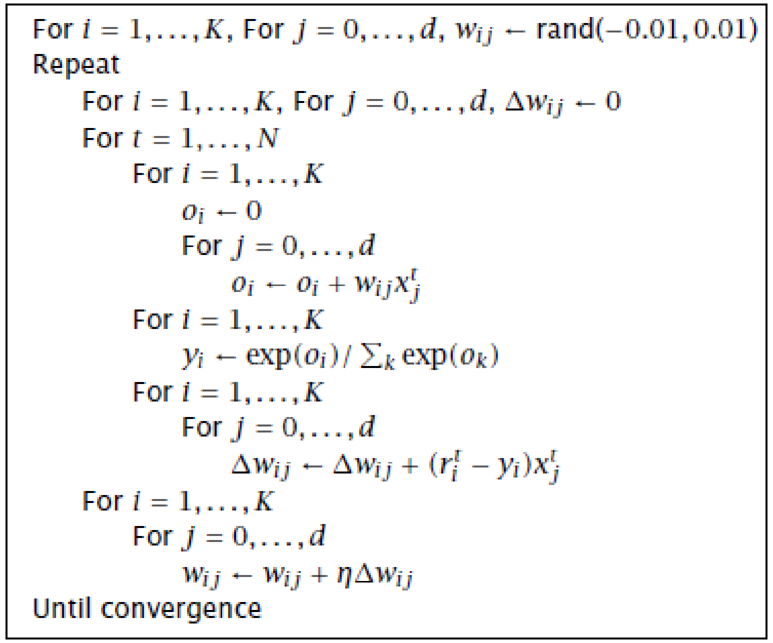
Online vs batch learning:

Stochastic Gradient Descent (Online)



```
For $i = 1, \ldots, K$
    For $j = 0, \ldots, d$
        $w_{ij} \leftarrow \text{rand}(-0.01, 0.01)$
Repeat
    For all $(x^t, r^t) \in \mathcal{X}$ in random order
        For $i = 1, \ldots, K$
            $o_i \leftarrow 0$
            For $j = 0, \ldots, d$
                $o_i \leftarrow o_i + w_{ij} x_j^t$
        For $i = 1, \ldots, K$
            $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$
        For $i = 1, \ldots, K$
            For $j = 0, \ldots, d$
                $w_{ij} \leftarrow w_{ij} + \eta (r_i^t - y_i) x_j^t$
Until convergence
```

Gradient Descent (Batch)



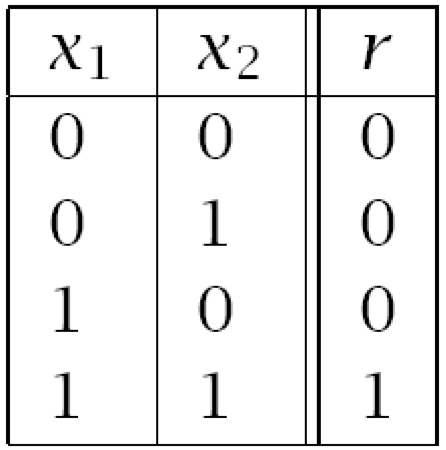
```
For $i = 1, \ldots, K$, For $j = 0, \ldots, d$, $w_{ij} \leftarrow \text{rand}(-0.01, 0.01)$
Repeat
    For $i = 1, \ldots, K$, For $j = 0, \ldots, d$, $\Delta w_{ij} \leftarrow 0$
    For $t = 1, \ldots, N$
        For $i = 1, \ldots, K$
            $o_i \leftarrow 0$
            For $j = 0, \ldots, d$
                $o_i \leftarrow o_i + w_{ij} x_j^t$
        For $i = 1, \ldots, K$
            $y_i \leftarrow \exp(o_i) / \sum_k \exp(o_k)$
        For $i = 1, \ldots, K$
            For $j = 0, \ldots, d$
                $\Delta w_{ij} \leftarrow \Delta w_{ij} + (r_i^t - y_i) x_j^t$
    For $i = 1, \ldots, K$
        For $j = 0, \ldots, d$
            $w_{ij} \leftarrow w_{ij} + \eta \Delta w_{ij}$
Until convergence
```

Learning Boolean Functions

- In a Boolean function, the inputs are binary and the output is 1 if the corresponding function value is true and 0 otherwise.
- Therefore, it can be seen as a two-class classification problem.

Learning Boolean AND

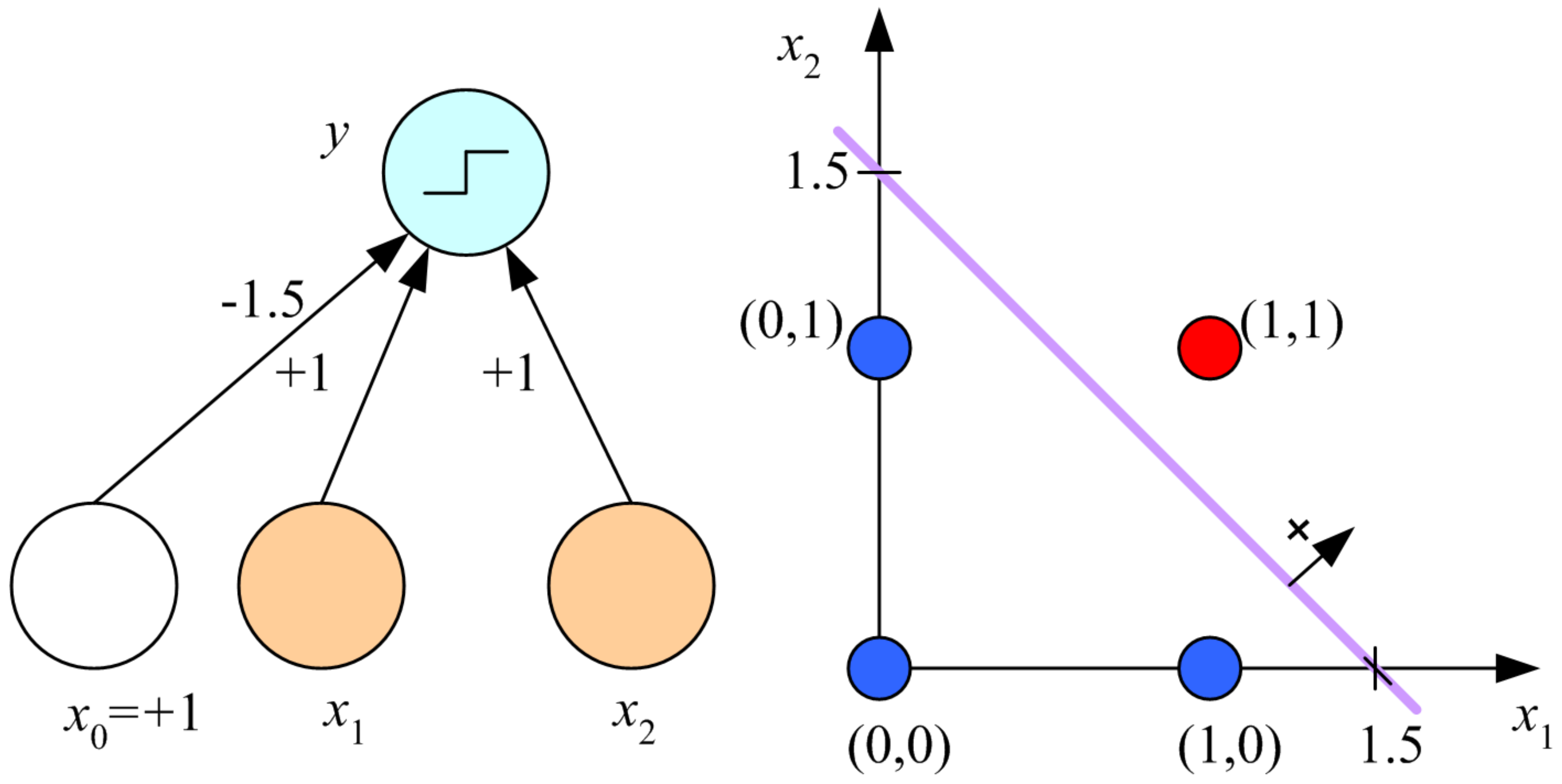
For learning to AND two inputs, the table of inputs and required outputs is



x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

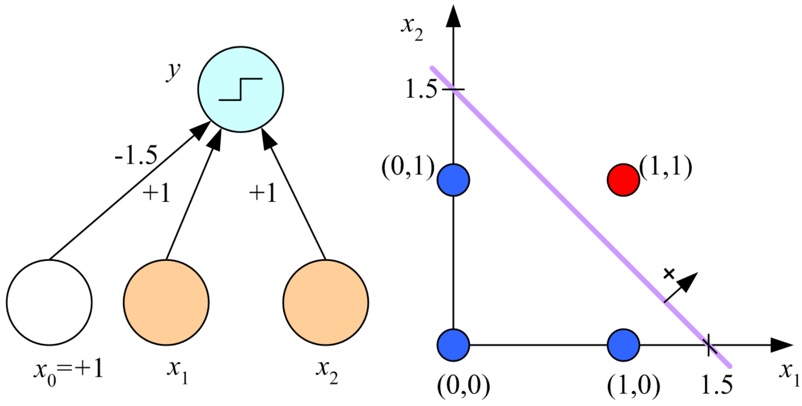
Learning Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1



The perceptron that implements AND and its geometric interpretation.

Learning Boolean AND



- The discriminant is

$$y = s(x_1 + x_2 - 1.5)$$

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1

For example,

$$x_1 = 0, x_2 = 0 \ \rightarrow y = s(0 + 0 - 1.5) = 0$$
$$x_1 = 1, x_2 = 0 \ \rightarrow y = s(1 + 0 - 1.5) = 0$$
$$x_1 = 0, x_2 = 1 \ \rightarrow y = s(0 + 1 - 1.5) = 0$$
$$x_1 = 1, x_2 = 1 \ \rightarrow y = s(1 + 1 - 1.5) = 1$$

$$s(a) = \begin{cases} 1 & if\ a > 0 \\ 0 & otherwise \end{cases}$$

Learning Boolean NOT

- The discriminant is

$$y = s(-x + 0.5)$$

For example,

$$x = 0 \rightarrow y = s(0 + 0.5) = 1$$
$$x = 1 \rightarrow y = s(-1 + 0.5) = 0$$

Learning Boolean OR

- The discriminant is

$$y = s(x_1 + x_2 - 0.5)$$

For example,

$$x_1 = 0, x_2 = 0 \ \rightarrow y = s(0 + 0 - 0.5) = 0$$
$$x_1 = 1, x_2 = 0 \ \rightarrow y = s(1 + 0 - 0.5) = 1$$
$$x_1 = 0, x_2 = 1 \ \rightarrow y = s(0 + 1 - 0.5) = 1$$
$$x_1 = 1, x_2 = 1 \ \rightarrow y = s(1 + 1 - 0.5) = 1$$

Learning Boolean Functions

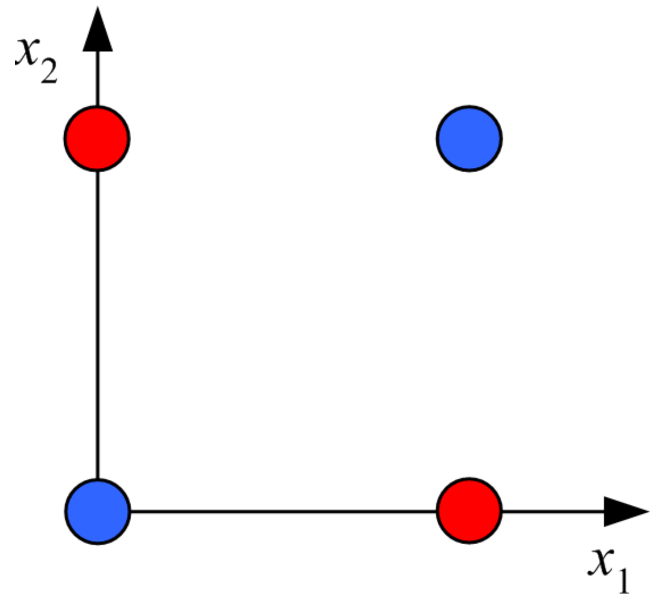
Though Boolean functions like AND and OR are linearly separable and are solvable using the perceptron, certain functions like XOR are not.

XOR

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0

- No w_0, w_1, w_2 satisfy:

$$\begin{aligned} w_0 &\le 0 \\ w_2 + w_0 &> 0 \\ w_1 + w_0 &> 0 \\ w_1 + w_2 + w_0 &\le 0 \end{aligned}$$



XOR problem is not linearly separable. We cannot draw a line where the empty circles are on one side and the filled circles on the other side.

(Minsky and Papert, 1969)

Multilayer Perceptrons

- A perceptron that has a single layer of weights can only approximate linear functions of the input and cannot solve problems like the XOR, where the discriminant to be estimated is nonlinear.
- Similarly, a perceptron cannot be used for nonlinear regression.
- This limitation does not apply to feedforward networks with intermediate or *hidden layers* between the multilayer input and the output layers.

Multilayer Perceptrons

- If used for classification, such *multilayer perceptrons* (MLP) can implement nonlinear discriminants
- If used for regression, it can approximate nonlinear functions of the input

Multilayer Perceptrons

- Input ***x*** is fed to the input layer (including the bias), the “activation” propagates in the forward direction, and the values of the hidden units z_h are calculated.
- Each hidden unit is a perceptron by itself and applies the nonlinear sigmoid function to its weighted sum

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

Multilayer Perceptrons

- The output y_i are perceptrons in the second layer taking the hidden units as their inputs

$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

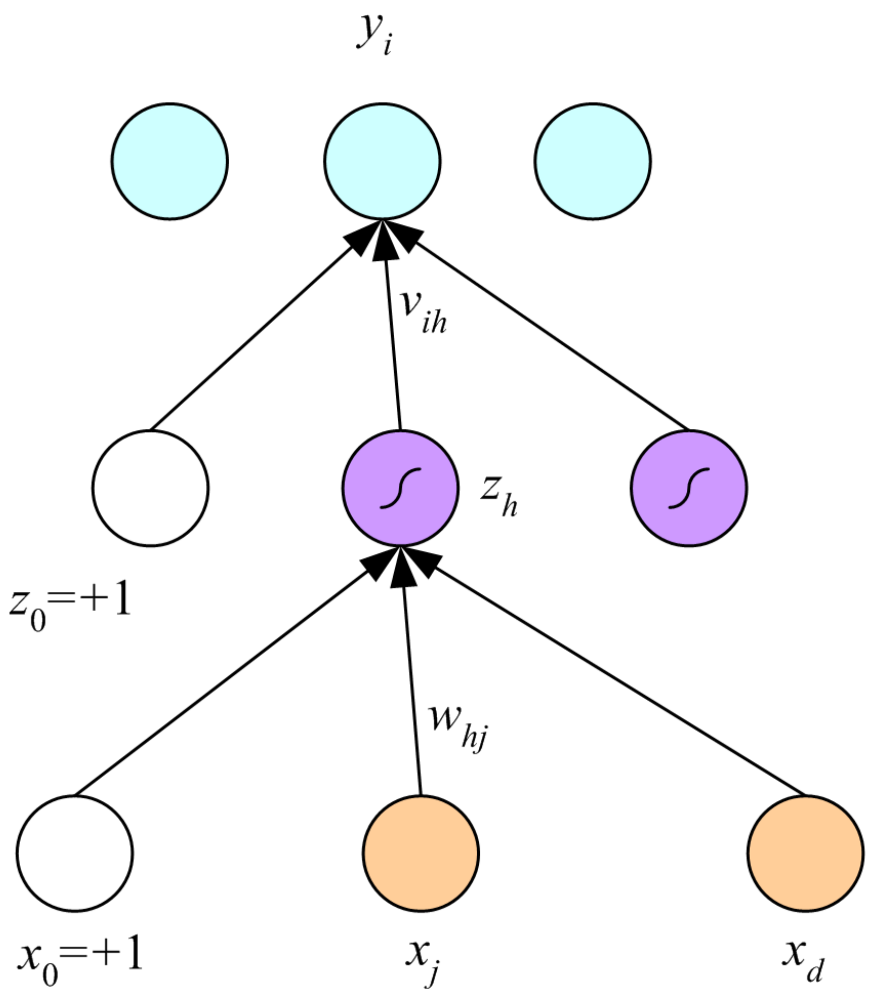
where there is also a bias unit in the hidden layer, which we denote by z_0, and v_{i0} are the bias weights.

Multilayer Perceptrons

- As usual, in a regression problem, there is no nonlinearity in the output layer in calculating y.

- In a two-class discrimination task, there is one sigmoid output unit and when there are $K > 2$ classes, there are K outputs with softmax as the output nonlinearity

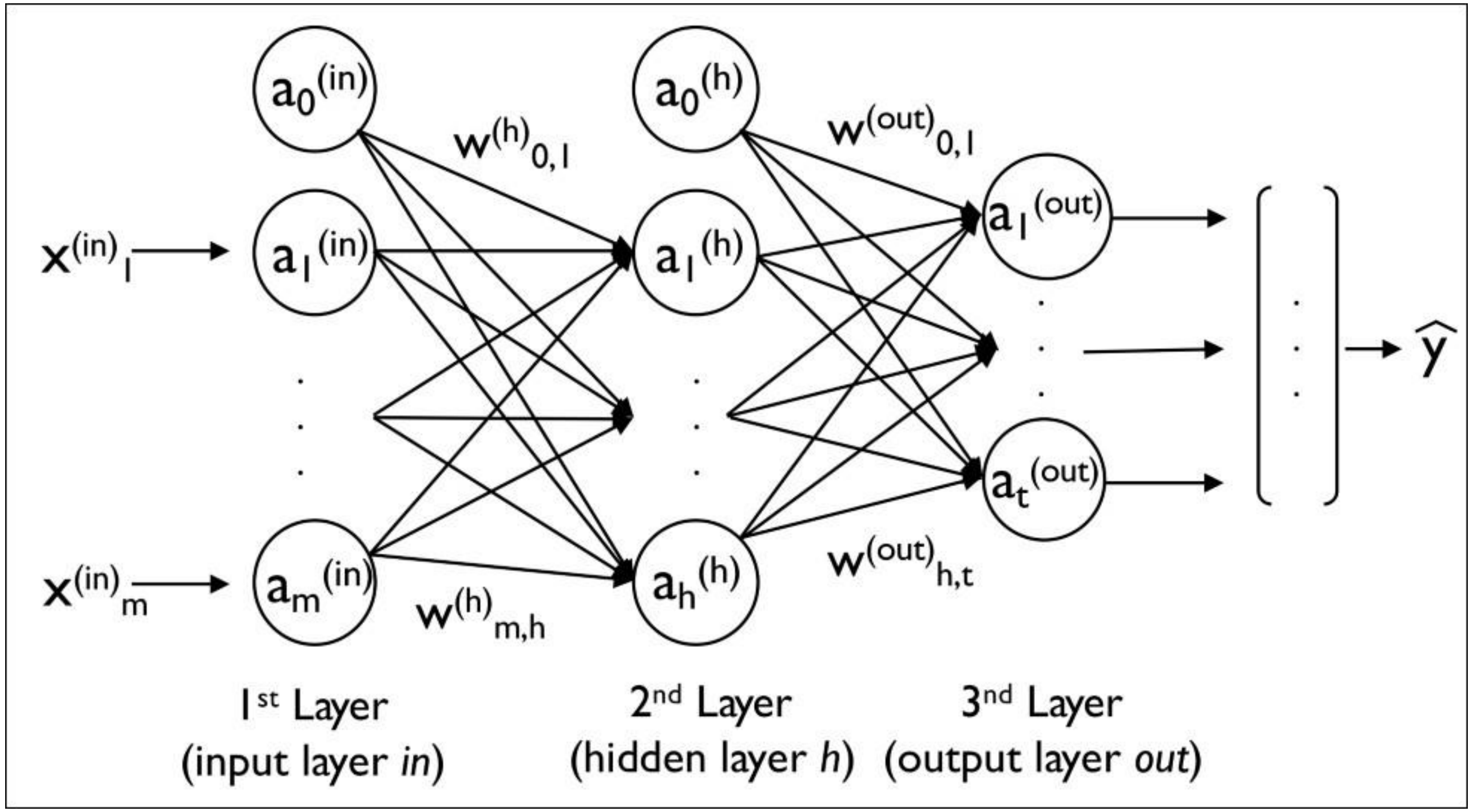
Multilayer Perceptrons



$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$
$$= \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

Multilayer Perceptrons



https://towardsdatascience.com/introduction-to-multilayer-neural-networks-with-tensorflows-keras-api-abf4f813959

XOR

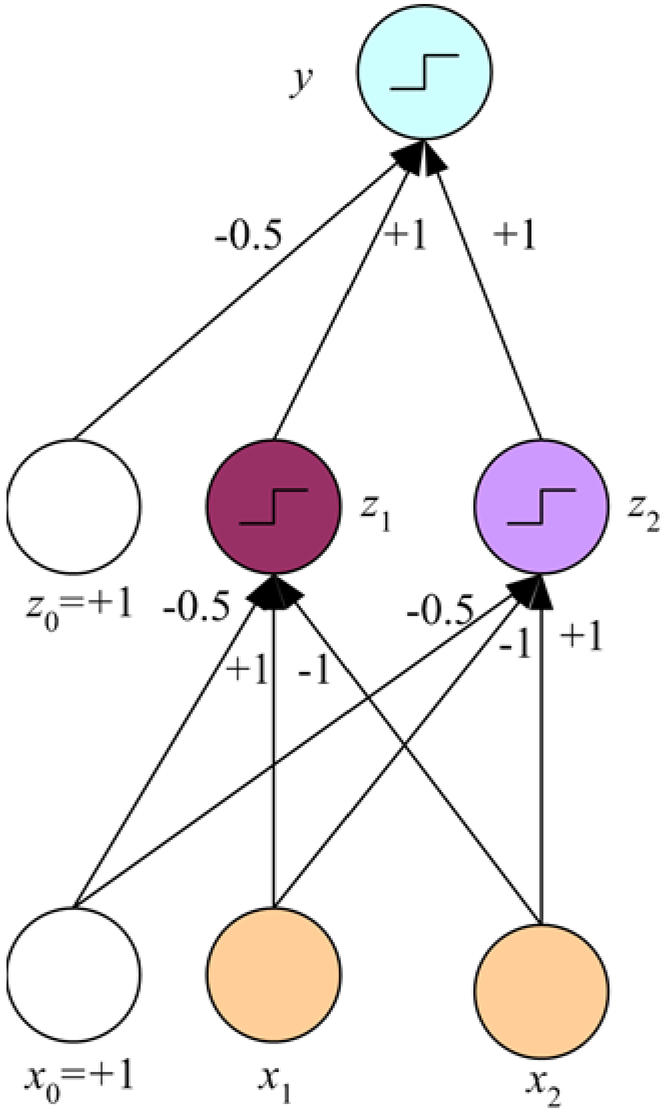
- We can represent any Boolean function as a disjunction of conjunctions, and such a Boolean expression can be implemented by a multilayer perceptron with one hidden layer
- Each conjunction is implemented by one hidden unit and the disjunction by the output unit.

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } {\sim}x_2) \text{ OR } ({\sim}x_1 \text{ AND } x_2)$$

XOR

- Two perceptrons can in parallel implement the two AND, and another perceptron on top can OR them together.

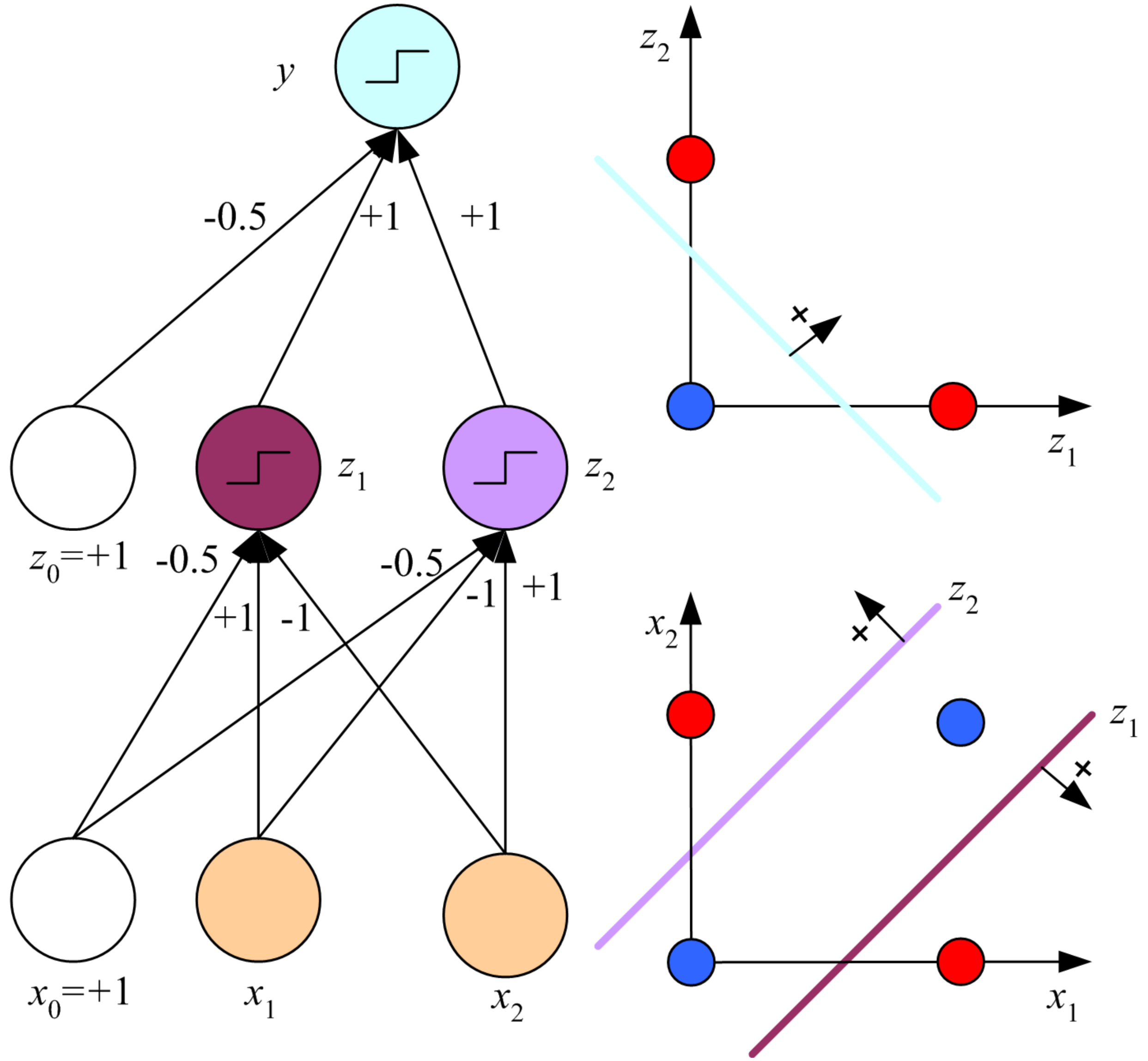
x_1 XOR x_2 = (x_1 AND ~x_2) OR (~x_1 AND x_2)



x_1 XOR x_2 = (x_1 AND ~x_2) OR (~x_1 AND x_2)

We see that the first layer maps inputs from the *(x_1, x_2)* to the *(z_1, z_2)* space defined by the first-layer perceptrons.

Note that both inputs, (0,0) and (1,1), are mapped to (0,0) in the *(z_1, z_2)* space, allowing linear separability in this second space.



Training

- Training a multilayer perceptron is the same as training a perceptron; the only difference is that now the output is a nonlinear function of the input thanks to the nonlinear basis function in the hidden units.

- Considering the hidden units as inputs, the second layer is a perceptron and we already know how to update the parameters, v_{ij}, in this case, given the inputs z_h.

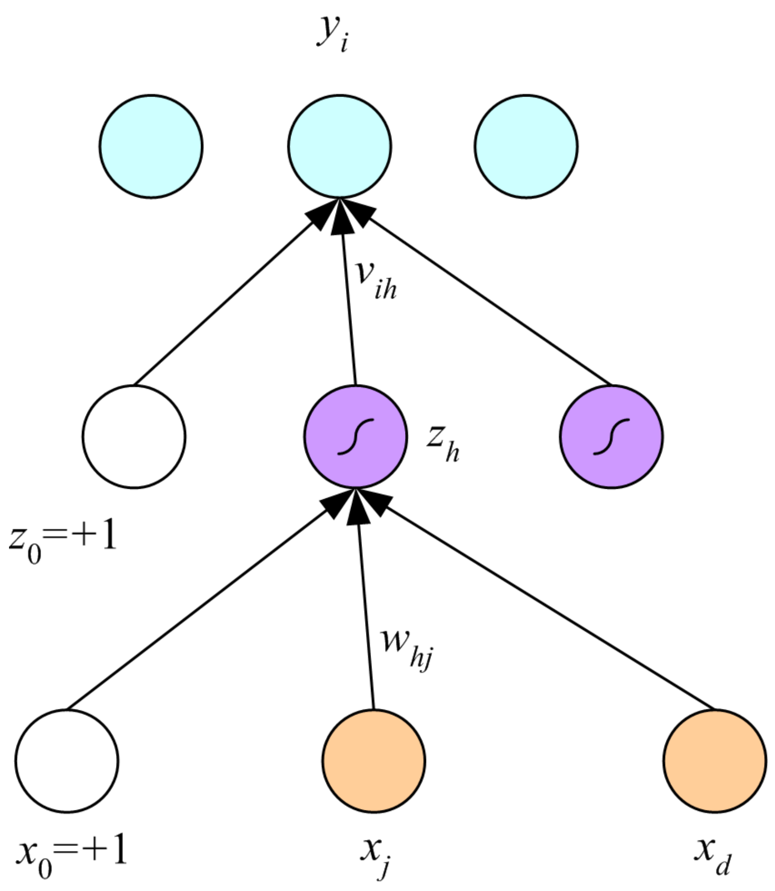
Training

For the first-layer weights, w_{hj} , we use the chain rule to calculate the gradient:

$$\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}$$

It is as if the error propagates from the output *y* back to the inputs and hence the name *backpropagation* was coined.

Backpropagation



$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

$$= \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}\right)\right]}$$

$$\boxed{\frac{\partial E}{\partial w_{hj}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}}$$

Backpropagation-
Regression with one output

$$E(\mathbf{W},\mathbf{v}\mid\mathcal{X})=\frac{1}{2}\sum_t\left(r^t-y^t\right)^2$$

$$y^t=\sum_{h=1}^{H}v_h z_h^t+v_0$$

$$\boxed{\Delta v_h=\sum_t\left(r^t-y^t\right)z_h^t}$$

Backward

$$\boxed{\begin{aligned}\Delta w_{hj}&=-\eta\frac{\partial E}{\partial w_{hj}}\\&=-\eta\sum_t\frac{\partial E}{\partial y^t}\frac{\partial y^t}{\partial z_h^t}\frac{\partial z_h^t}{\partial w_{hj}}\\&=-\eta\sum_t-\left(r^t-y^t\right)v_h\boxed{z_h^t\left(1-z_h^t\right)}x_j^t\\&=\eta\sum_t\left(r^t-y^t\right)v_h z_h^t\left(1-z_h^t\right)x_j^t\end{aligned}}$$

Forward

$$z_h=\boxed{\text{sigmoid}}\left(\mathbf{w}_h^T\mathbf{x}\right)$$

$$x$$

Backpropagation – Regression with one output

The second layer is a perceptron with hidden units as the inputs, and we use the least-squares rule to update the second-layer weights:

$$\Delta v_h = -\eta \frac{\partial E}{\partial v_h} = -\eta \sum_t \boxed{\frac{\partial E}{\partial y^t}} \boxed{\frac{\partial y^t}{\partial v_h}} = -\eta \sum_t \boxed{-(r^t - y^t)} . \boxed{z_h^t}$$

$$\Delta v_h = \eta \sum_t (r^t - y^t) . z_h^t \, , z_0^t = 1$$

$$E(\mathbf{W}, \mathbf{v} \mid \mathcal{X}) = \frac{1}{2} \sum_t \left(r^t - y^t\right)^2$$

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

Backpropagation – Regression with one output

The chain rule

$$
\begin{aligned}
\Delta w_{hj} &= -\eta \frac{\partial E}{\partial w_{hj}} \\
&= -\eta \sum_t \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial z_h^t} \frac{\partial z_h^t}{\partial w_{hj}} \\
&= -\eta \sum_t \underbrace{-(r^t - y^t)}_{\partial E^t / \partial y^t} \underbrace{v_h}_{\partial y^t / \partial z_h^t} \underbrace{z_h^t (1 - z_h^t) x_j^t}_{\partial z_h^t / \partial w_{hj}} \\
&= \eta \sum_t (r^t - y^t) v_h z_h^t (1 - z_h^t) x_j^t
\end{aligned}
$$

$$E(\mathbf{W}, \mathbf{v} \mid \mathcal{X}) = \frac{1}{2} \sum_t \left(r^t - y^t\right)^2$$

$$y^t = \sum_{h=1}^{H} v_h z_h^t + v_0$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

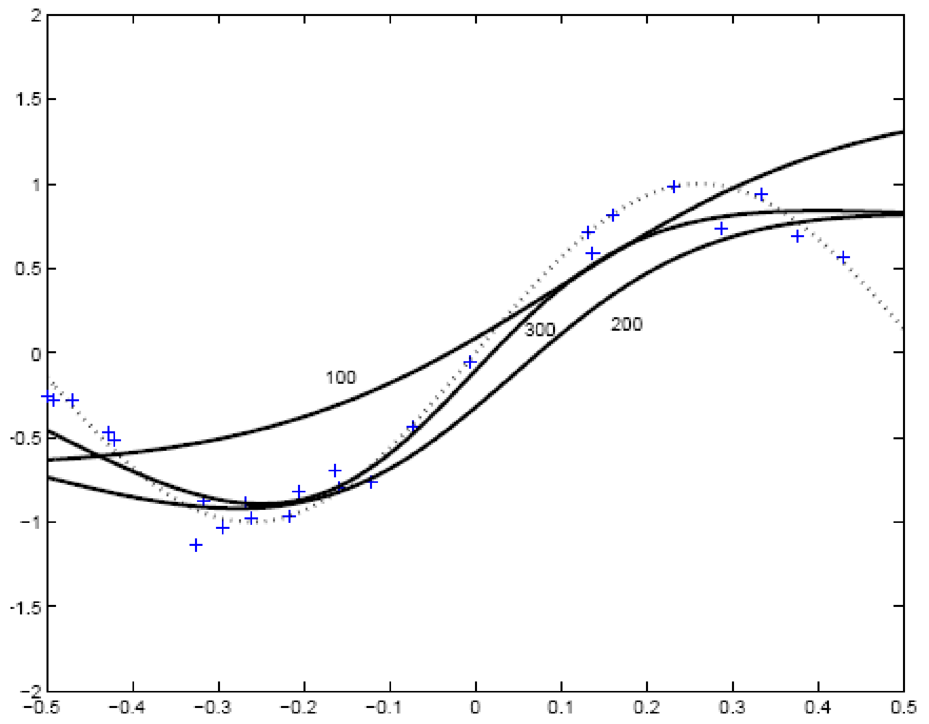
Batch Learning

- With the learning equations given here, for each pattern, we compute the direction in which each parameter needs be changed and the magnitude of this change.
- In *batch learning*, we accumulate these changes over all patterns and make the change once after a complete pass over the whole training set is made.

Online Learning

- It is also possible to have online learning, by updating the weights after each pattern, thereby implementing stochastic gradient descent.
- A complete pass over all the patterns in the training set is called an *epoch*.
- The learning factor, η, should be chosen smaller in this case and patterns should be scanned in a random order.
- Online learning converges faster because there may be similar patterns in the dataset, and the stochasticity has an effect like adding noise and may help escape local minima.

An example of training a multilayer perceptron for regression



The evolution of the fit of an MLP with two hidden units after 100, 200, and 300 epochs is drawn.

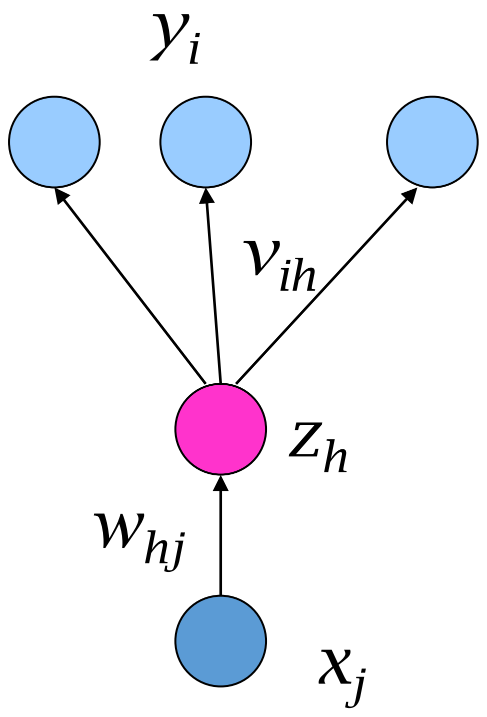
Regression with Multiple Outputs

$$E(\mathbf{W},\mathbf{V}\mid\mathcal{X})=\frac{1}{2}\sum_t\sum_i\left(r_i^t-y_i^t\right)^2$$

$$y_i^t=\sum_{h=1}^{H}v_{ih}z_h^t+v_{i0}$$

$$\Delta v_{ih}=\eta\sum_t\left(r_i^t-y_i^t\right)z_h^t$$

$$\Delta w_{hj}=\eta\sum_t\left[\sum_i\left(r_i^t-y_i^t\right)v_{ih}\right]z_h^t\left(1-z_h^t\right)x_j^t$$



Two-Class Discrimination

- When there are two classes, one output unit suffices:

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right)$$

- The error function is

$$E(\mathbf{W}, \mathbf{v} \mid \mathcal{X}) = -\sum_t r^t \log y^t + \left(1 - r^t\right) \log\left(1 - y^t\right)$$

Two-Class Discrimination

$$E(\mathbf{W},\mathbf{v}|\mathcal{X}) = -\sum_t r^t \log y^t + (1-r^t)\log(1-y^t)$$

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right)$$

$$a = \sum_{h=1}^{H} v_h z_h^t + v_0$$

$$\Delta v_h = -\eta \frac{\partial E}{\partial v_h} = -\eta \sum_t \frac{\partial E}{\partial y^t} \frac{dy^t}{da} \frac{\partial a}{\partial v_h}$$

$$= \eta \sum_t \left(\frac{r^t}{y^t} - \frac{1-r^t}{y^t}\right) y^t(1-y^t).\, z_h^t$$

$$\Delta v_h = \eta \sum_t (r^t - y^t).\, z_h^t$$

Two-Class Discrimination

$$E(\mathbf{W},\mathbf{v}\mid\mathcal{X}) = -\sum_t r^t \log y^t + (1-r^t)\log(1-y^t)$$

$$y^t = \text{sigmoid}\left(\sum_{h=1}^{H} v_h z_h^t + v_0\right)$$

$$a = \sum_{h=1}^{H} v_h z_h^t + v_0$$

$$z_h = \text{sigmoid}\left(\mathbf{w}_h^T \mathbf{x}\right)$$

$$\Delta w_{hj} = -\eta \frac{\partial E}{\partial w_{hj}} = -\eta \sum_t \frac{\partial E}{\partial y^t} \frac{dy^t}{da} \frac{\partial a}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}$$

$$= \eta \sum_t \left(\frac{r^t}{y^t} - \frac{1-r^t}{y^t}\right) y^t(1-y^t) v_h z_h^t (1-z_h^t) x_j^t$$

$$\Delta w_{hj} = \eta \sum\nolimits_t (r^t - y^t) v_h\, z_h^t (1-z_h^t) x_j^t$$

Multiclass Discrimination (*K*>2 Classes)

$$o_i^t = \sum_{h=1}^{H} v_{ih} z_h^t + v_{i0}$$

$$y_i^t = \frac{\exp o_i^t}{\sum_k \exp o_k^t} \equiv P\left(C_i \mid \mathbf{x}^t\right)$$

$\boxed{\text{Softmax}}$

$$E\left(\mathbf{W}, \mathbf{v} \mid \mathcal{X}\right) = -\sum_t \sum_i r_i^t \log y_i^t$$

$$\Delta v_{ih} = \eta \sum_t \left(r_i^t - y_i^t\right) z_h^t$$

$$\Delta w_{hj} = \eta \sum_t \left[\sum_i \left(r_i^t - y_i^t\right) v_{ih} \right] z_h^t \left(1 - z_h^t\right) x_j^t$$

Multiple Hidden Layers

- MLP with one hidden layer is a universal approximator (Hornik et al., 1989), but using multiple layers may lead to simpler networks

$$z_{1h} = \text{sigmoid}\left(\mathbf{w}_{1h}^T \mathbf{x}\right) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right), h = 1, \ldots, H_1$$

$$z_{2l} = \text{sigmoid}\left(\mathbf{w}_{2l}^T \mathbf{z}_1\right) = \text{sigmoid}\left(\sum_{h=1}^{H_1} w_{2lh} z_{1h} + w_{2l0}\right), l = 1, \ldots, H_2$$

$$y = \mathbf{v}^T \mathbf{z}_2 = \sum_{l=1}^{H_2} v_l z_{2l} + v_0$$

Neural Networks: Pros and Cons

Pros

- They form the basis of state-of-the-art models and can be formed into advanced architectures that effectively capture complex features given enough data and computation.

Cons

- Larger, more complex models require significant training time, data, and customization.
- Careful preprocessing of the data is needed.
- A good choice when the features are of similar types, but less so when features of very different types.

Training Procedures

Improving Convergence

- Gradient descent has various advantages.
- It is simple.
- It is local; namely, the change in a weight uses only the values of the presynaptic and postsynaptic units and the error (suitably backpropagated).
- When online training is used, it does not need to store the training set and can adapt as the task to be learned changes.
- However, there are two frequently used simple techniques that improve the performance of the gradient descent considerably, making gradient-based methods feasible in real applications.

Training Procedures

Improving Convergence

1. Momentum
2. Adaptive Learning Rate

Training Procedures

Improving Convergence

Momentum

$$\Delta w_i^t = -\eta \frac{\partial E}{\partial w_i} + \alpha \Delta w_i^{t-1} \quad \alpha \in [0.5, 1.0]$$

- This approach is especially useful when online learning is used, where as a result we get an effect of averaging and smooth the trajectory during convergence.
- The disadvantage is that the past Δw_i^{t-1} values should be stored in extra memory

Training Procedures

Improving Convergence

Adaptive Learning Rate:

It can be made adaptive for faster convergence, where it is kept large when learning takes place and is decreased when learning slows down:

$$\Delta\eta = \begin{cases} +a & if\, E^{t+\tau} < E^t \\ -b\eta & otherwise \end{cases}$$

Thus we increase η by a constant amount if the error on the training set decreases and decrease it geometrically if it increases.

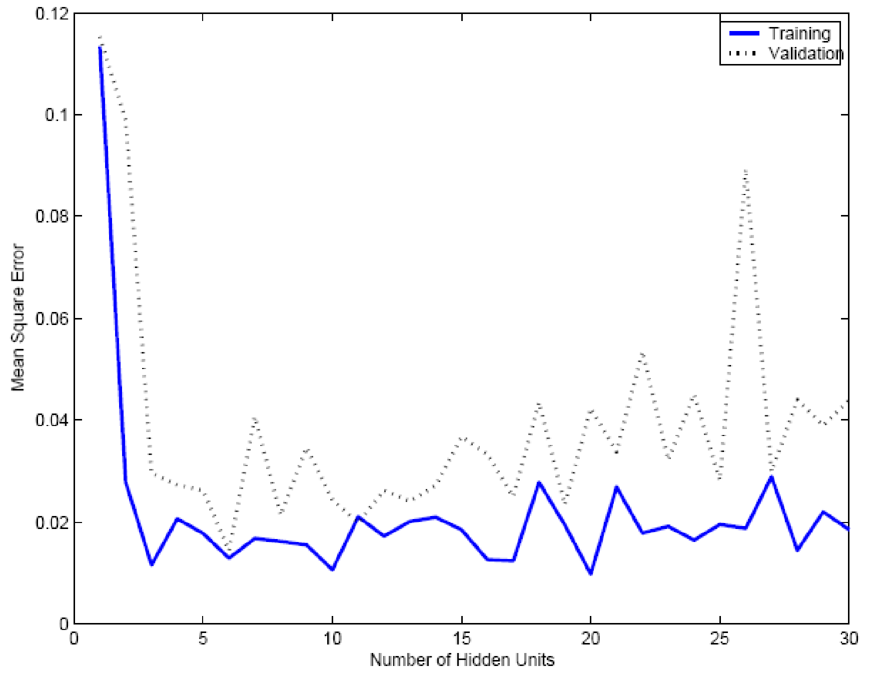
Overtraining

- A multilayer perceptron with d inputs, H hidden units, and K outputs has $H(d+1)$ weights in the first layer and $K(H+1)$ weights in the second layer.
- Number of weights: $H\,(d+1)+(H+1)K$
- Both the space and time complexity of an MLP is $O(H \cdot (K + d))$.
- When e denotes the number of training epochs, training time complexity is $O(e \cdot H \cdot (K + d))$.

Overtraining

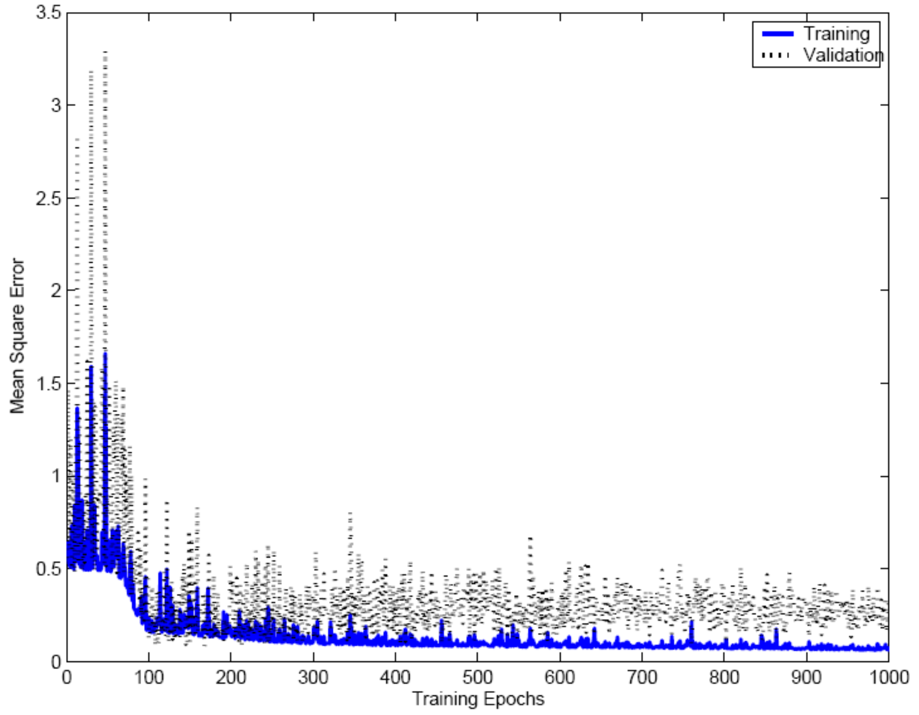
- In an application, *d* and *K* are predefined and *H* is the parameter that we play with to tune the complexity of the model.
- Similarly in an MLP, when the number of hidden units is large, the generalization accuracy deteriorates and the bias/variance dilemma also holds for the MLP

Overtraining



As complexity increases, training error is fixed but the validation error starts to increase and the network starts to overfit.

Overtraining



As training continues, the validation error starts to increase and the network starts to overfit.

Tuning the Network Size

- To find the optimal network size, the most common approach is to try many different architectures, train them all on the training set, and choose the one that generalizes best to the validation set.

Tuning the Network Size

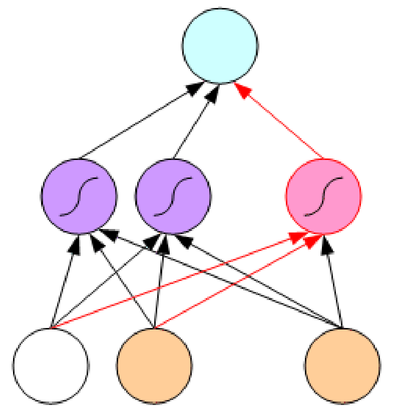
- Another approach is to incorporate structural this *structural adaptation* into the learning algorithm.
- There are two ways adaptation this can be done:
  - In the *destructive* approach, we start with a large network and gradually remove units and/or connections that are not necessary.
  - In the *constructive* approach, we start with a small network and gradually add units and/or connections to improve performance

Destructive

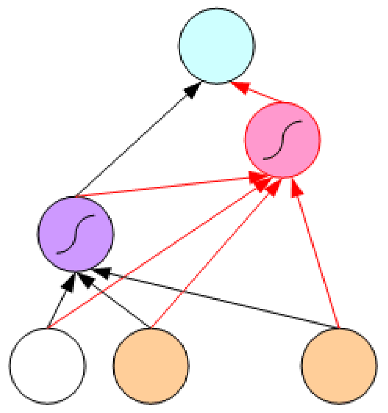
- One destructive method is *weight decay* where the idea is to remove unnecessary connections.
- Ideally to be able to determine whether a unit or connection is necessary, we need to train once with and once without and check the difference in error on a separate validation set.
- This is costly since it should be done for all combinations of such units/connections.

Constructive

- Dynamic node creation adds a unit to an existing layer.
- Cascade correlation adds each unit as a new hidden layer connected to all the previous layers.
- Read lines denote the newly added unit/connections.
- Bias units/weights are omitted for clarity.



Dynamic Node Creation



Cascade Correlation

Scikit learn

- MLPClassifier and MLPRegressor Important Parameters
  - hidden_layer_sizes: sets the number of hidden layers (number of elements in list), and number of hidden units per layer (each list element). *Default: (100).*
  - alpha: controls weight on the regularization penalty that shrinks weights to zero. *Default: alpha = 0.0001.*
  - activation: controls the nonlinear function used for the activation function, including: 'relu' (default), 'logistic', 'tanh'.

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

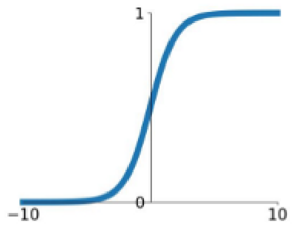
Activation Functions

- Sigmoid (logistic)
- tanh
- ReLU

Activation Functions

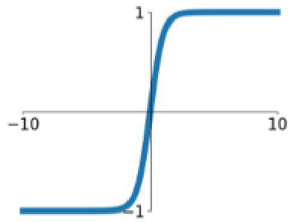
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



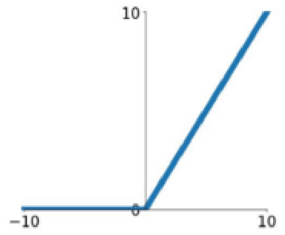
tanh

$\tanh(x)$



ReLU

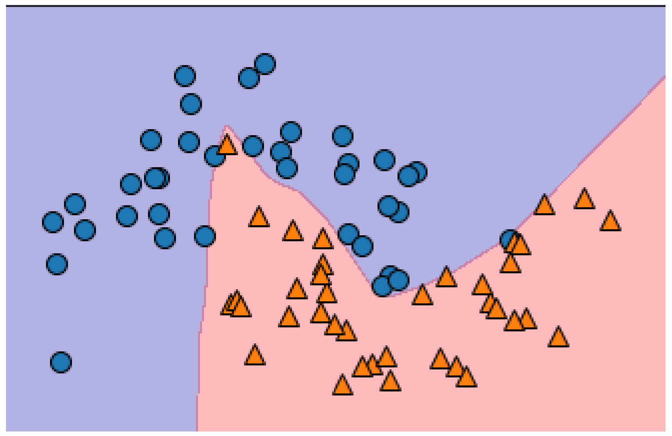
$\max(0, x)$



https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

Scikit learn

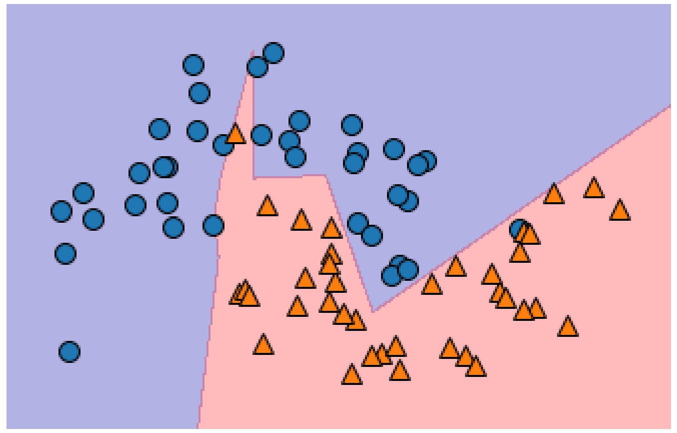
```
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)

mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```



https://github.com/amueller/introduction_to_ml_with_python/tree/master/mglearn

Scikit learn

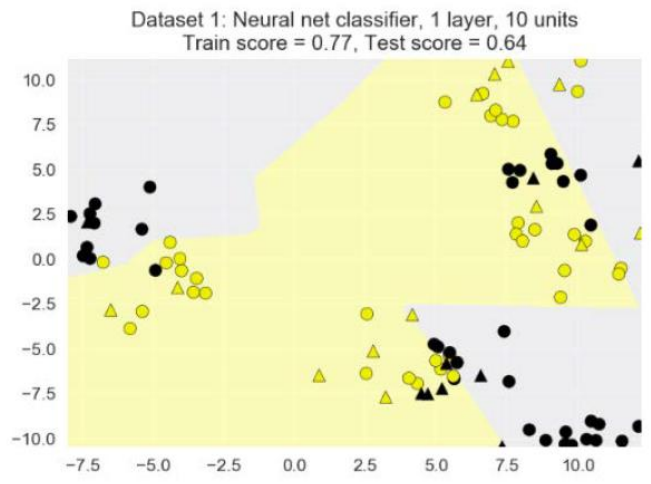
```
#By default, the MLP uses 100 hidden nodes
#We can reduce the number, 10 hidden units

mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```



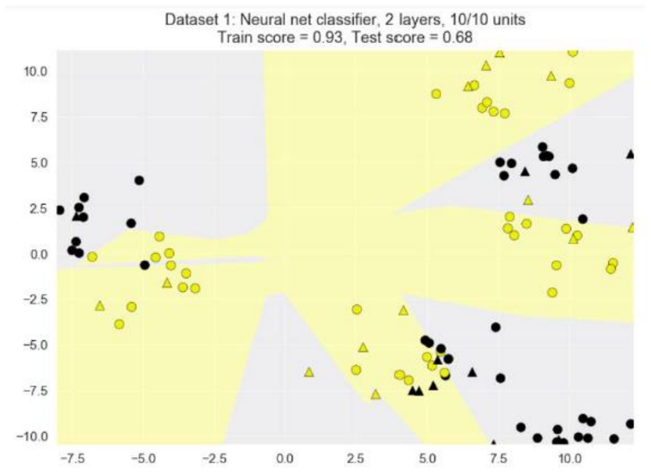
https://github.com/amueller/introduction_to_ml_with_python/tree/master/mglearn

APPLIED MACHINE LEARNING IN PYTHON

One vs Two Hidden Layers

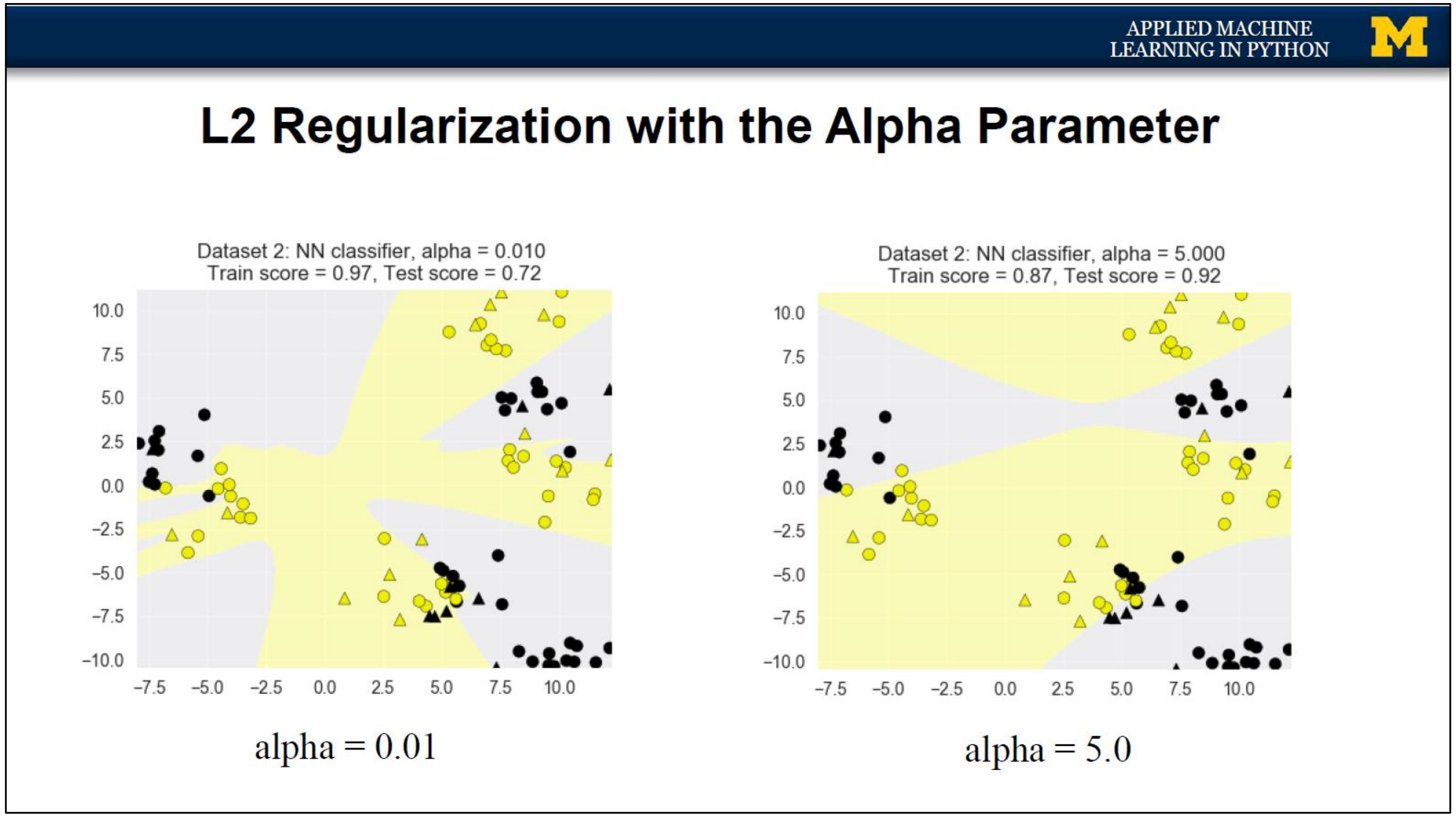


1 layer, 10 units



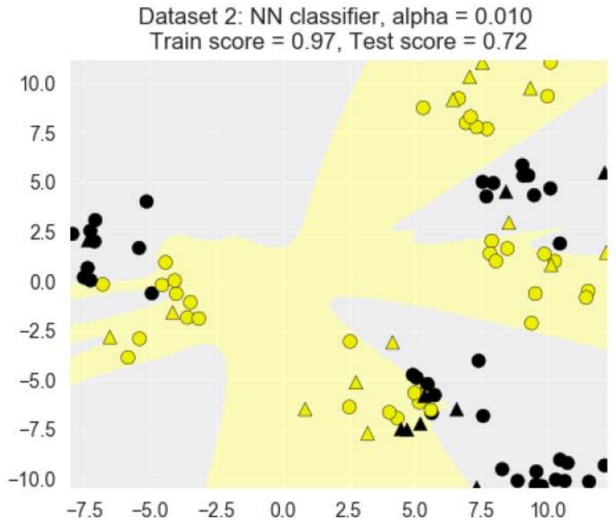
2 layers, (10, 10) units

Lecture notes by Kevyn Collins-Thompson
Applied Machine Learning (Coursera)

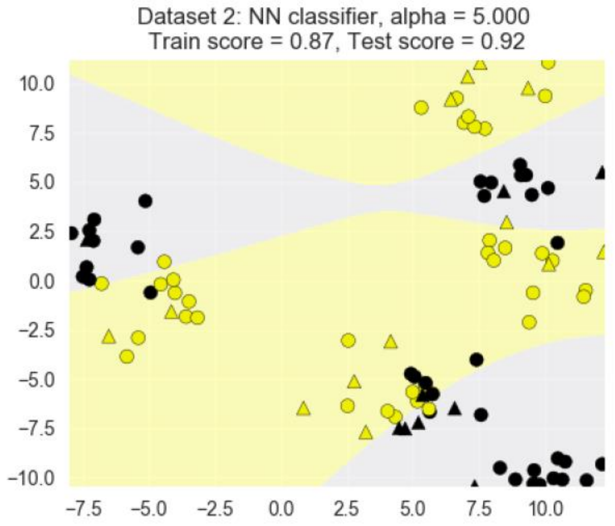


APPLIED MACHINE LEARNING IN PYTHON

L2 Regularization with the Alpha Parameter



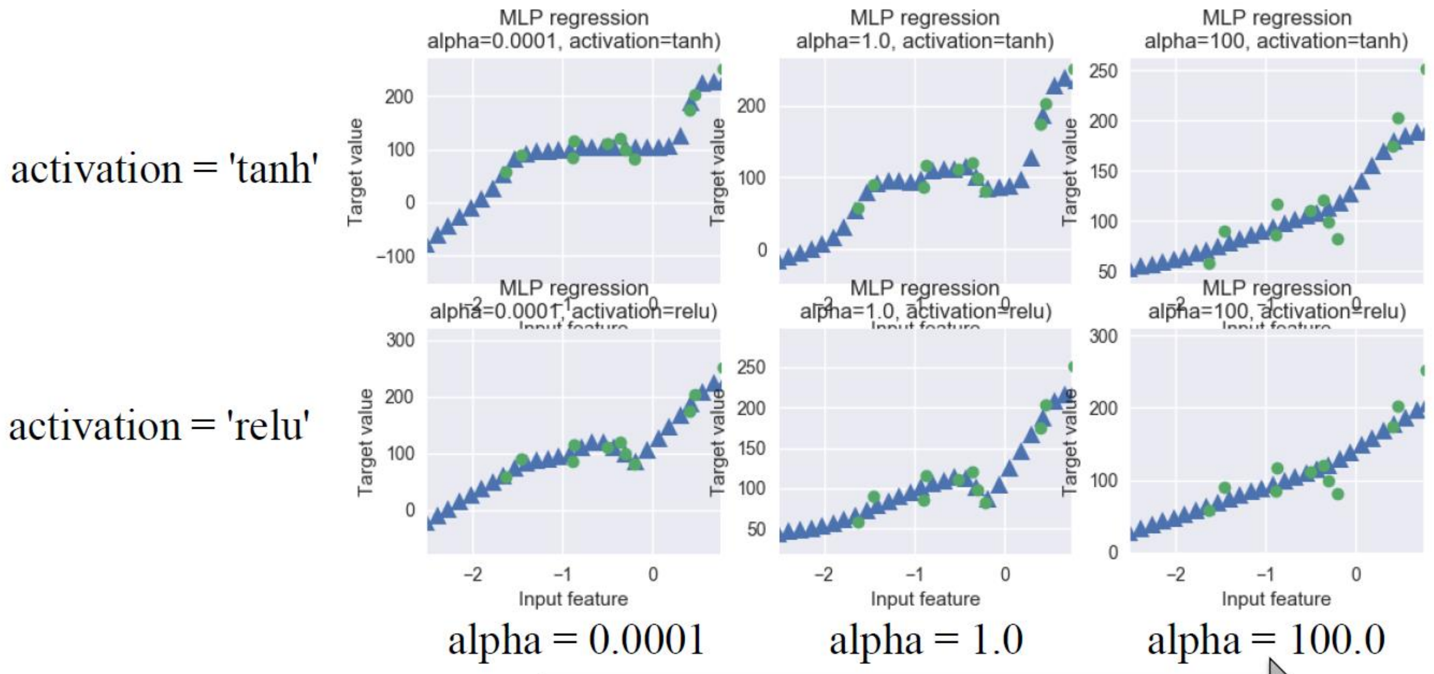
alpha = 0.01



alpha = 5.0

Lecture notes by Kevyn Collins-Thompson
Applied Machine Learning (Coursera)

Neural Network Regression with MLPRegressor



Increasing regularization

Lecture notes by Kevyn Collins-Thompson
Applied Machine Learning (Coursera)