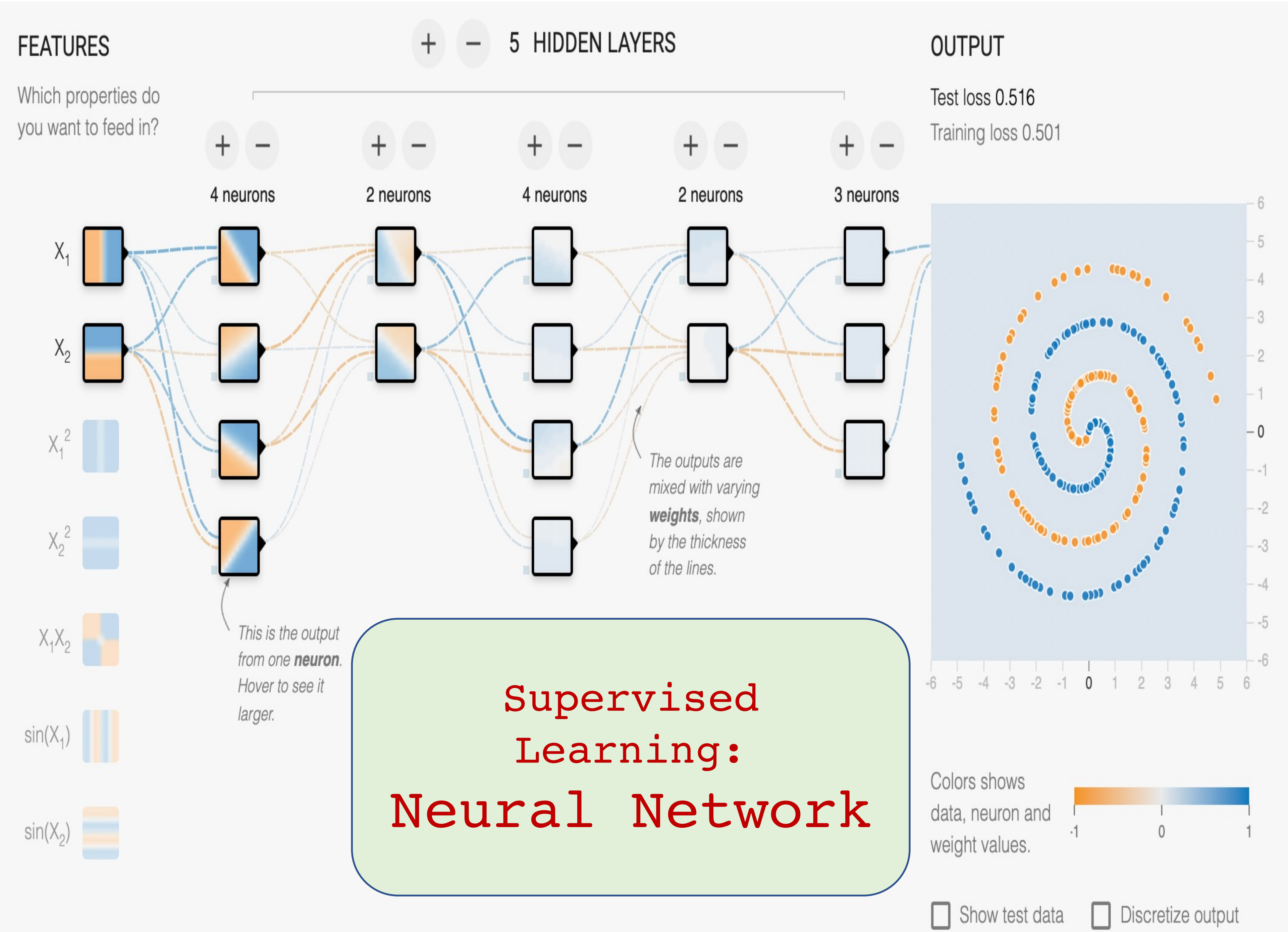


Supervised Learning: Neural Network

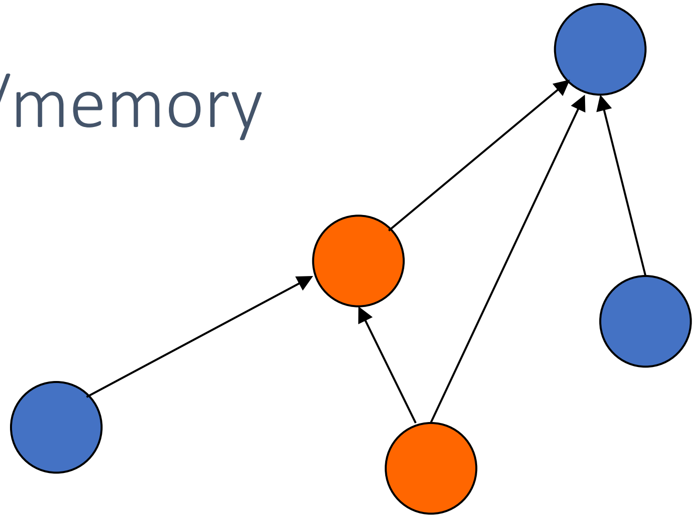
Image captured from playground.tensorflow.org: [A NN Demo]

Roadmap

- Overview of neural networks
- Single-layer NN: Perceptron
  - Training, delta rule, etc.
  - Limitation
- Multi-layered NN: Multilayered Perceptron (MLP)
  - Chain rule
  - Back-propagation algorithm
  - Other issues
- Take-home messages

Multilayered Perceptron (MLP): A Very First 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



Understanding the Brain

- Levels of analysis (Marr, 1982)
  1. Computational theory
  2. Representation and algorithm
  3. Hardware implementation
- Reverse engineering: From hardware to theory
- Parallel processing:

  SIMD (Single instruction, multiple data) vs

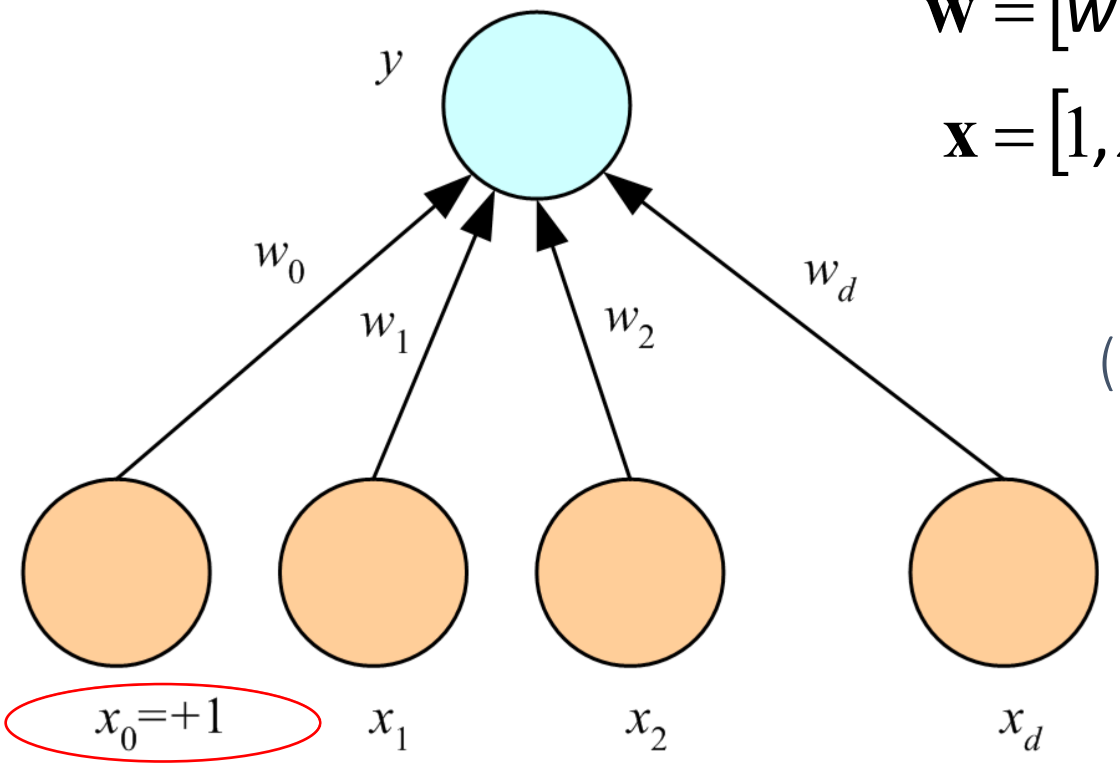
  MIMD (Multiple instruction, multiple data)
- Neural net: SIMD with modifiable local memory
- Learning: Update by training/experience (*E*)

A Perceptron

$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$

$$\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$$

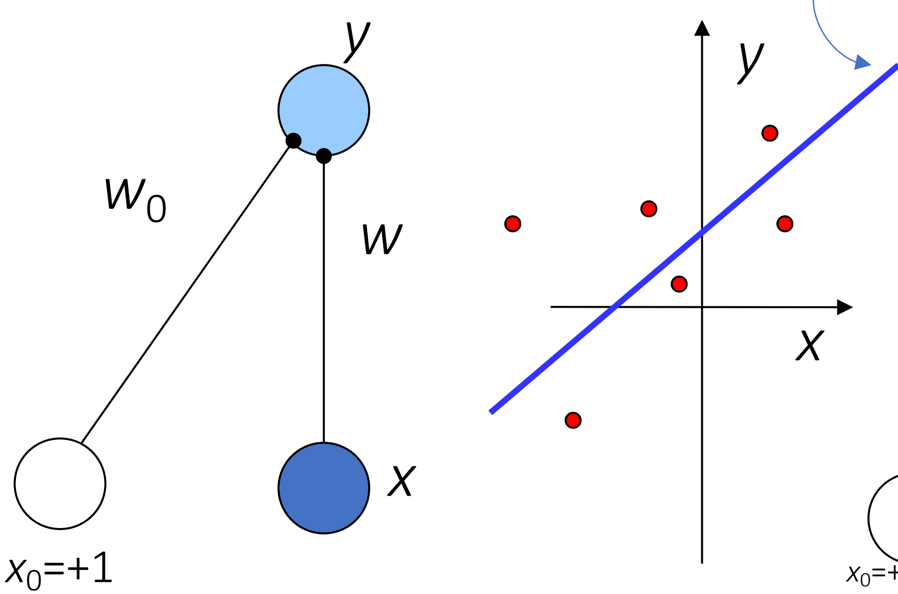
$$\mathbf{x} = [1, x_1, \ldots, x_d]^T$$

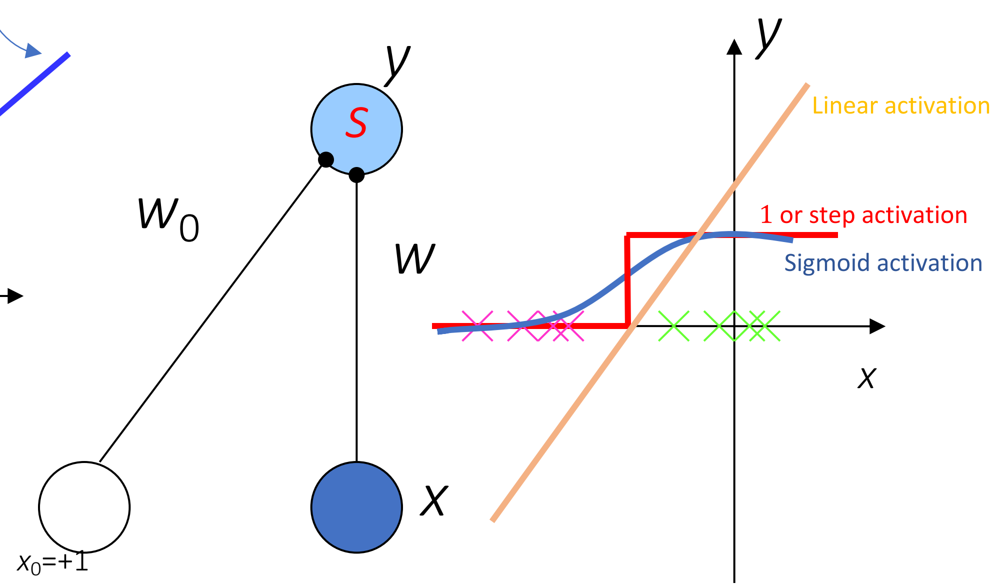


(Rosenblatt, 1962)

What a **Perceptron** Does?

- Regression: $y=wx+w_0$
- Classification: $y=1(wx+w_0>0)$





Activation Functions:

1 or step activation: $y = 1(o) = \begin{cases} 1 & if\ o > 0 \\ 0 & otherwise \end{cases}$

Linear activation: $y = o = \boldsymbol{w}^T\boldsymbol{x}$

Sigmoid activation: $y = sigmoid(o) = \dfrac{1}{1+e^{-\boldsymbol{W}^T\boldsymbol{x}}}$

K **Perceptrons** for K Outputs

Classification:

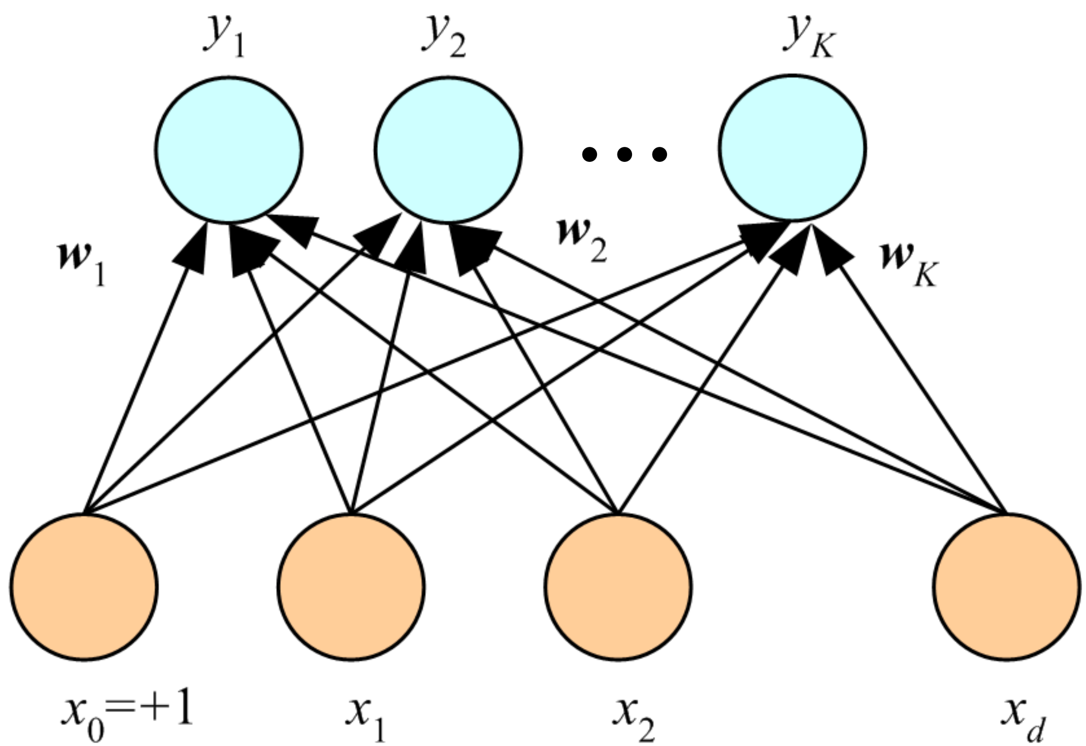
$$o_i = \mathbf{w}_i^T \mathbf{x}$$

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

choose C_i

if $y_i = \max_k y_k$

Softmax function



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

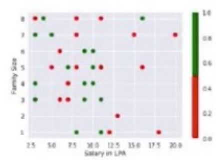
$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

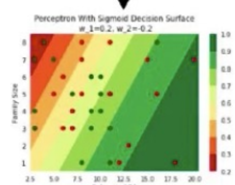
Supervised Learning with NN

Objective function

Real inputs $\in \mathbb{R}$

Classification /Regression





$$Loss = \sum_{i=1}^{n}(y-\hat{y})^2$$

$$w = w + \Delta w$$
$$b = b + \Delta b$$

OR

$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$

$$RMSE$$

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Learning/Training

- Online (instances seen one by one) learning
  - No need to store the whole sample as in batch learning
  - Problem may change in time
- Stochastic gradient-descent (**SGD**): Update after a single instance
- Generic update rule (delta rule):

t denotes current training instance

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

Weight_update = learning_rate x (desired_output - predicted_output) x input

Training a **Perceptron**: Regression Task

- Regression (Linear output): Linear activation: $y = o = \boldsymbol{w}^T \boldsymbol{x}$

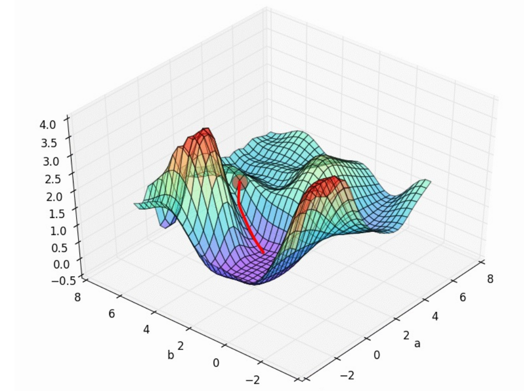
Half squared error/loss

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

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

$$\boxed{\Delta w_j^t = -\eta \frac{\partial E^t}{\partial w_j^t} \quad \text{<---- Gradient descent}}$$

Gradient Descent



- E($\boldsymbol{w}$|X) is error with parameters $\boldsymbol{w}$ on sample X

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} E(\boldsymbol{w} \mid X)$$

- Gradient

Partial derivatives

$$\nabla_w E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_d}\right]^T$$

Gradient in n-D space, i.e., vector of Partial derivatives; del or nabla operator

Parameter space

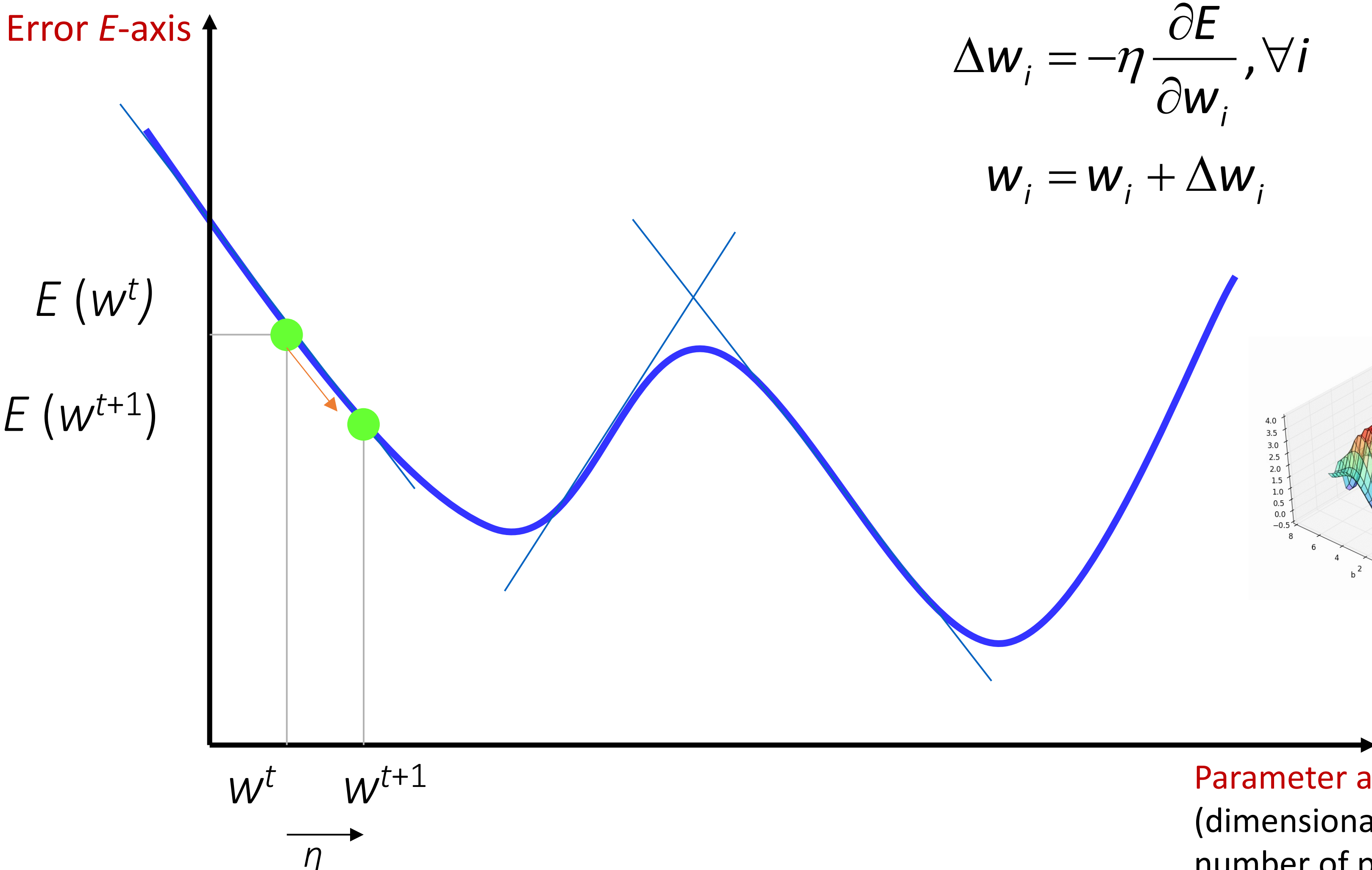
- Gradient-descent:

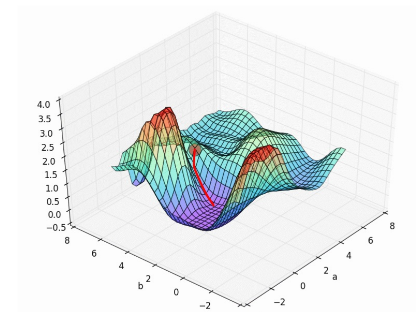
Starts from random $\boldsymbol{w}$ and updates $\boldsymbol{w}$ iteratively in the negative direction of gradient. (Note that gradient is a vector pointing at the greatest increase of a function, negative gradient is a vector pointing at the greatest decrease of a function!)

Gradient Descent

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \forall i$$

$$w_i = w_i + \Delta w_i$$





Training a **Perceptron**: Classification Task

- K=1 Single sigmoid output

Sigmoid activation: $y = sigmoid(o) = \dfrac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}}$

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

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

Logistic loss

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

Weight_update = learning_rate x (desired_output - predicted_output) x input

- $K \geq 2$ Softmax outputs

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

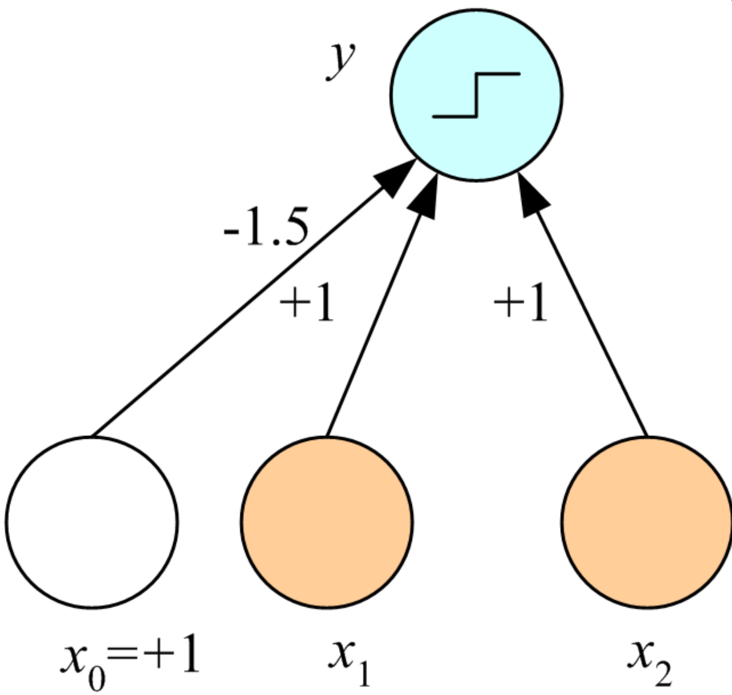
$$\Delta w_{ij}^t = \eta\left(r_i^t - y_i^t\right)x_j^t$$

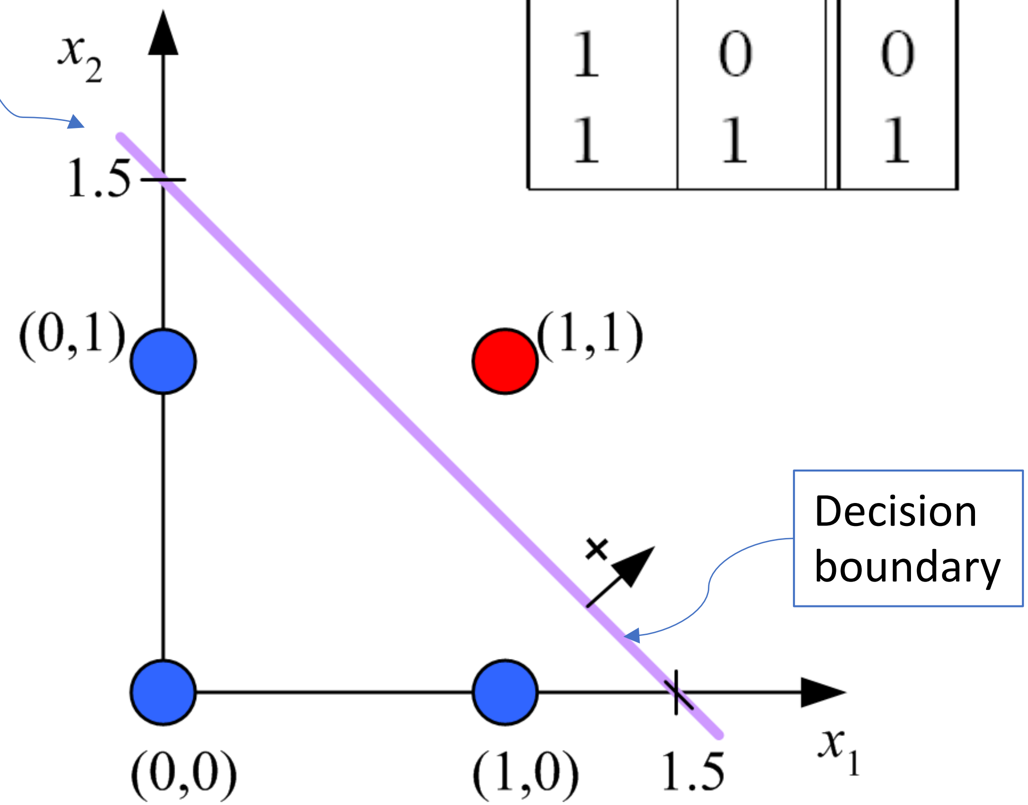
Weight_update = learning_rate x (desired_output - predicted_output) x input

Learning Boolean AND

Linear separable!

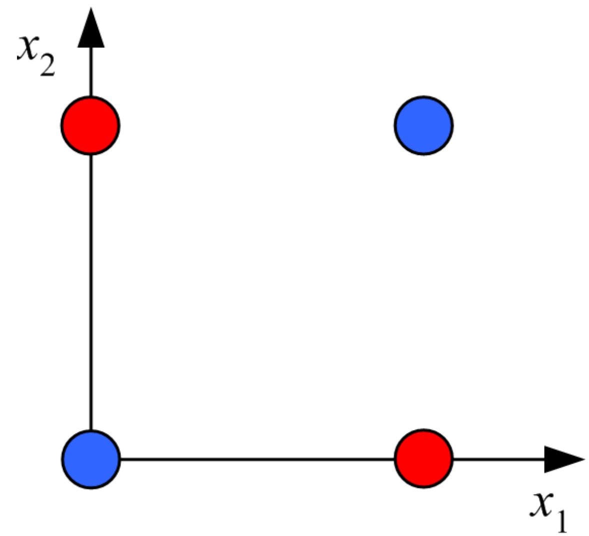
x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1





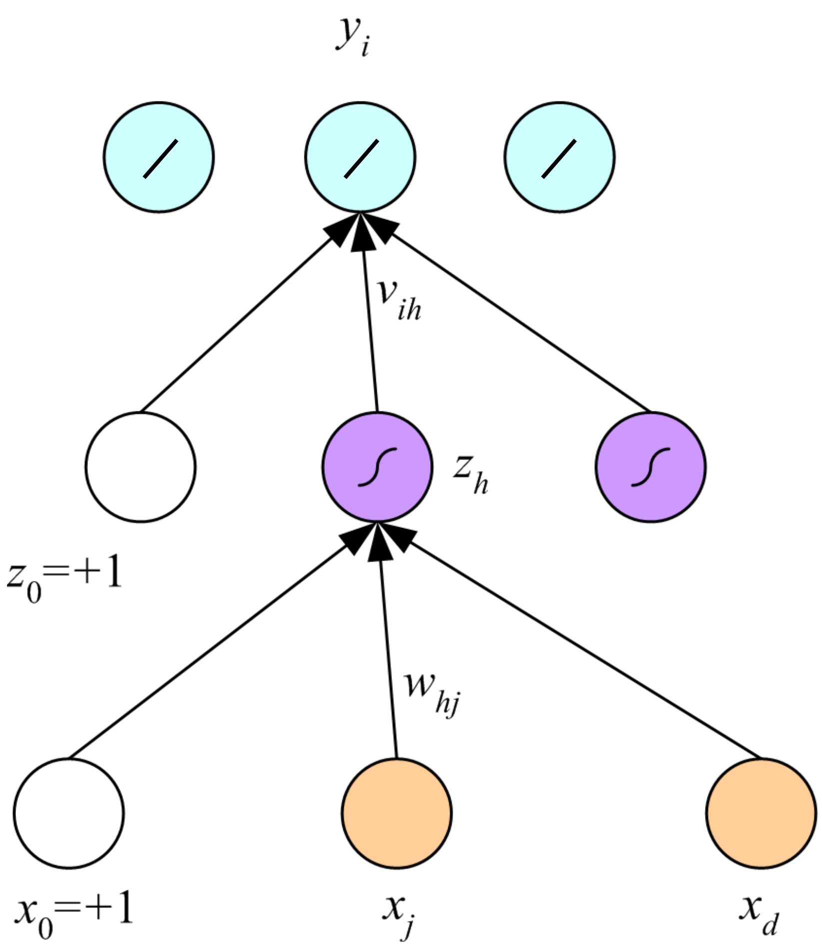
How about learning $\underbrace{\text{XOR}}_{\text{Linear non-separable!}}$?

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



Any straight line (one line only) to perfectly separate the four pts?

Empowering perceptron with more layers: Multilayer Perceptrons (MLP)

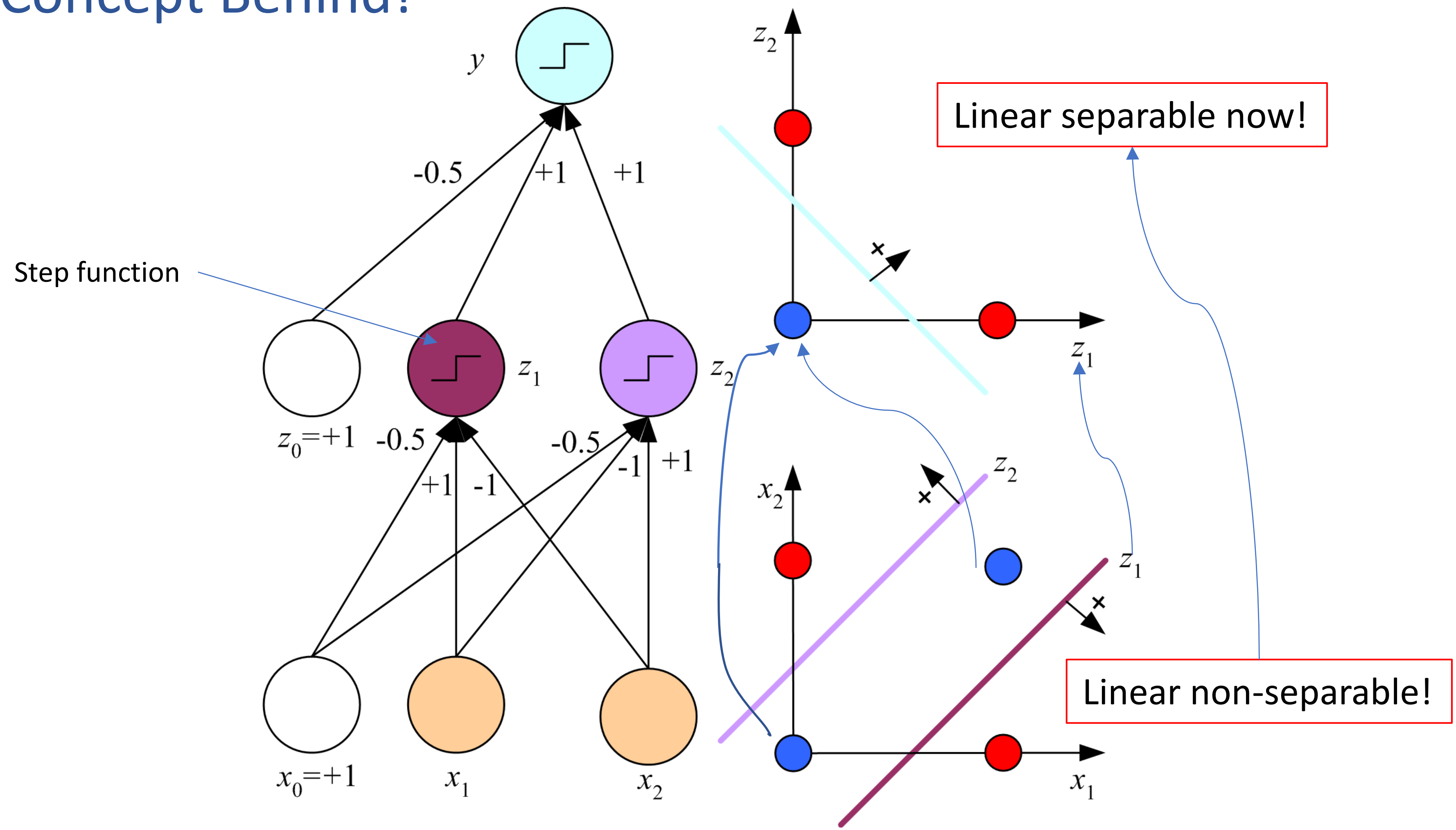


$$y_i = \mathbf{v}_i^T \mathbf{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0}$$

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

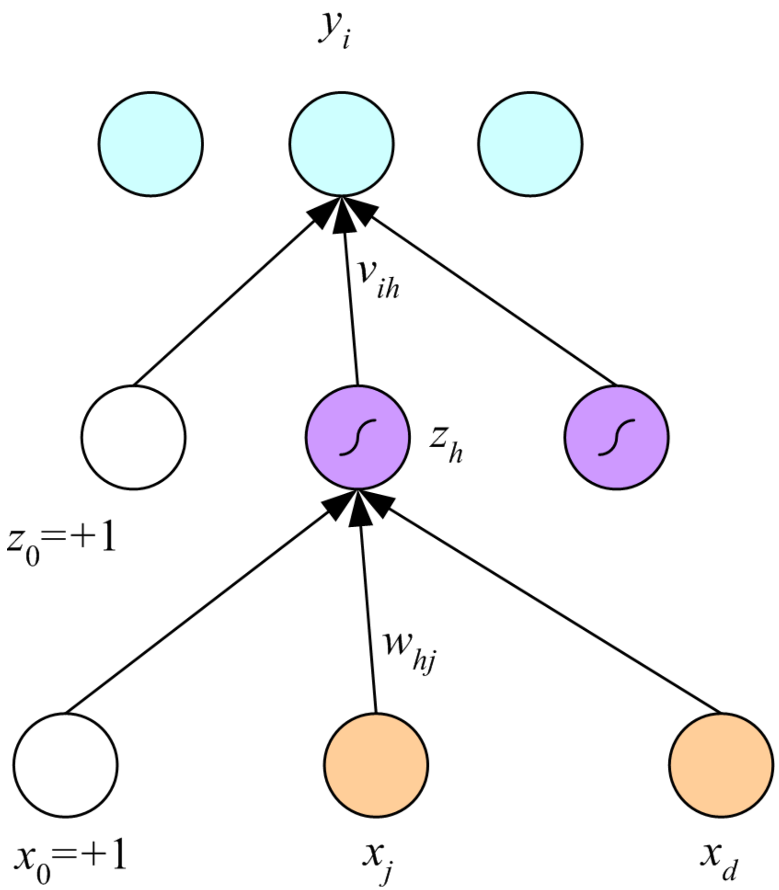
(Rumelhart et al., 1986)

Concept Behind!



$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } \underbrace{\sim x_2}_{\text{Not } x_2}) \text{ OR } (\underbrace{\sim x_1}_{\text{Not } x_1} \text{ AND } x_2)$$

Backpropagation (**BP**) Algorithm
for Training **MLP**



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

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

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

Very famous “Chain rule” in calculus!

Regression

with (K=)1 Output

Error (to backpropagate):

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

Output layer

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

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

Backward

Feed-Forward

$$\Delta w_{hj}=-\eta\frac{\partial E}{\partial w_{hj}}$$

$$=-\eta\sum_t\boxed{\frac{\partial E}{\partial y^t}}\boxed{\frac{\partial y^t}{\partial z_h^t}}\boxed{\frac{\partial z_h^t}{\partial w_{hj}}}$$

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

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

Hidden layer

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

Feed-Forward

Input layer

$$x$$

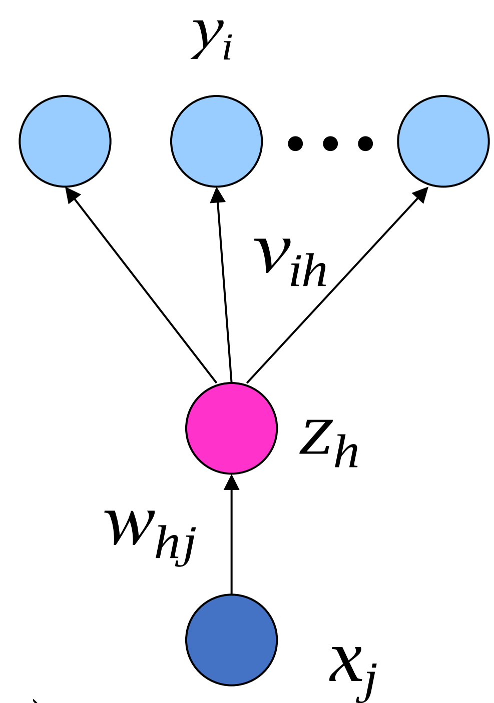
Regression with Multiple ($K \geq 2$) Outputs

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

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

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

$$\Delta w_{hj} = \eta \sum_t \underbrace{\left[\sum_i \left(r_i^t - y_i^t\right) v_{ih}\right]}_{\text{Aggregated errors to backpropagate!}} z_h^t \left(1 - z_h^t\right) x_j^t$$



Back Propagation Algorithm

Initialize all v_{ih} and w_{hj} to rand$(-0.01, 0.01)$
Repeat
  For all $(\boldsymbol{x}^t, r^t) \in \mathcal{X}$ in random order
    For $h = 1, \ldots, H$ /* for all hidden neurons */
      $z_h \leftarrow \mathrm{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}^t)$
    For $i = 1, \ldots, K$ /* for all output neurons */
      $y_i = \boldsymbol{v}_i^T \boldsymbol{z}$
    For $i = 1, \ldots, K$ /* Compute updates for all output neurons */
      $\Delta \boldsymbol{v}_i = \eta(r_i^t - y_i^t)\boldsymbol{z}$
    For $h = 1, \ldots, H$ /* Compute updates for all hidden neurons */
      $\Delta \boldsymbol{w}_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) \boldsymbol{x}^t$
    For $i = 1, \ldots, K$ /* Update parameters for all output neurons */
      $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta \boldsymbol{v}_i$
    For $h = 1, \ldots, H$ /* Update parameters for all hidden neurons */
      $\boldsymbol{w}_h \leftarrow \boldsymbol{w}_h + \Delta \boldsymbol{w}_h$
Until convergence

Feedforward

Feedback (Back-propagation)

Learning Curve and Performance

Learning curve…error decreasing

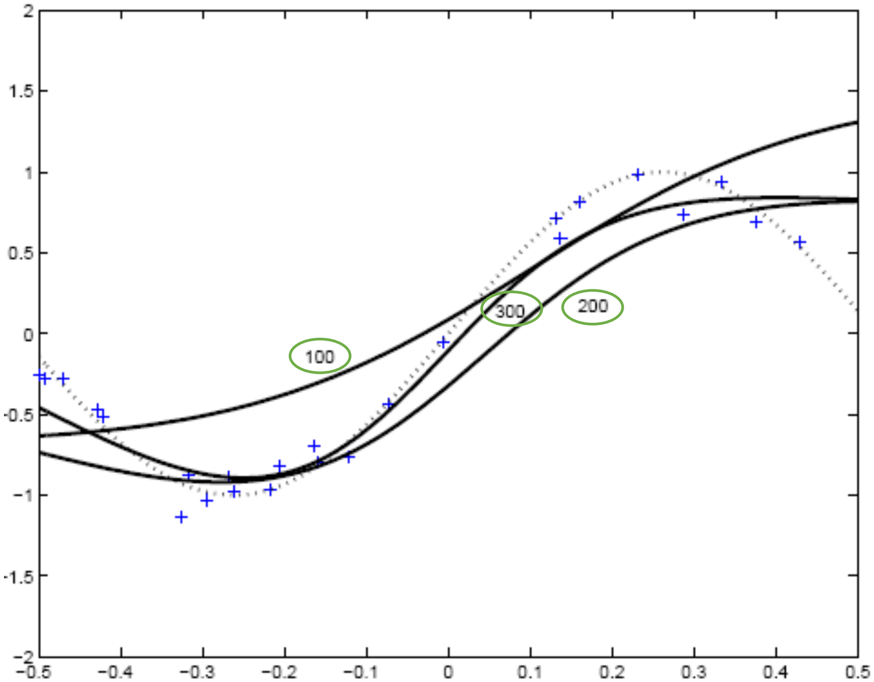


Figure 11.8 Sample training data shown as '+', where $x^t \sim U(-0.5, 0.5)$, and $y^t = f(x^t) + \mathcal{N}(0, 0.1)$. $f(x) = \sin(6x)$ is shown by a dashed line. The evolution of the fit of an MLP with two hidden units after 100, 200, and 300 epochs is drawn.

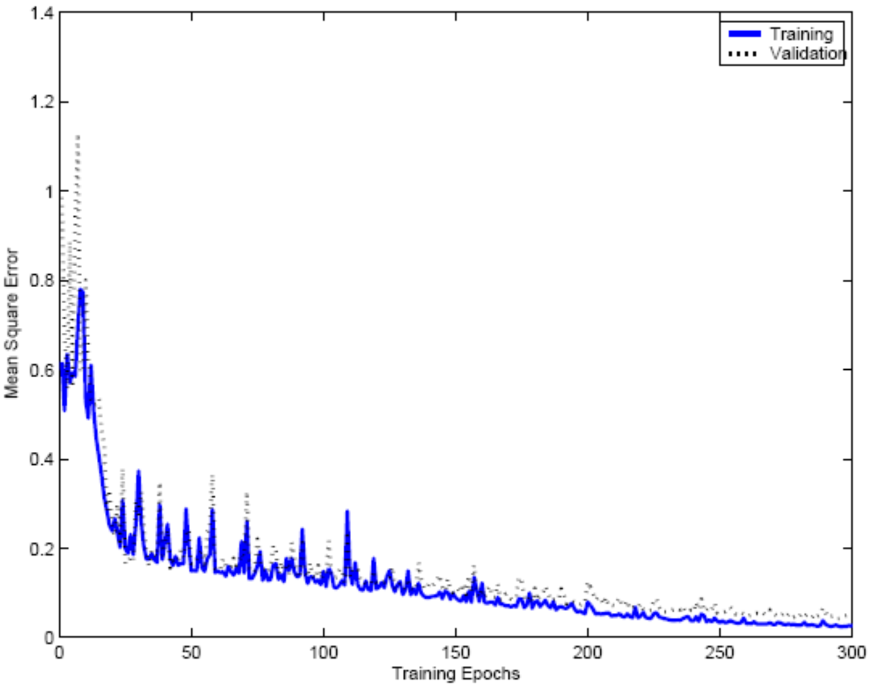


Figure 11.9 The mean square error on training and validation sets as a function of training epochs.

Special test sets

Qualitative Description of Backpropagation Learning

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to **minimize the mean squared error** between the network's prediction and the actual target value
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**"
- Steps
  - Initialize weights to small random numbers, associated with biases
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)

Multiple Hidden Layers

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

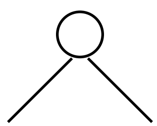
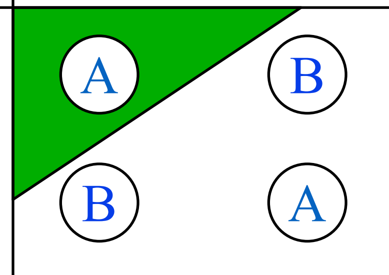
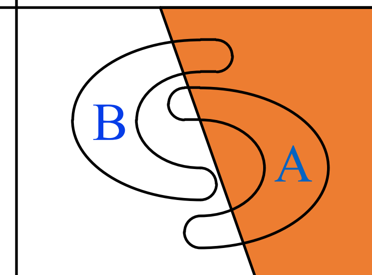
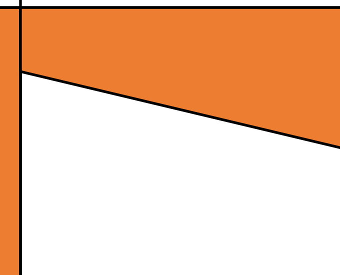
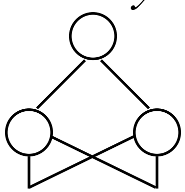
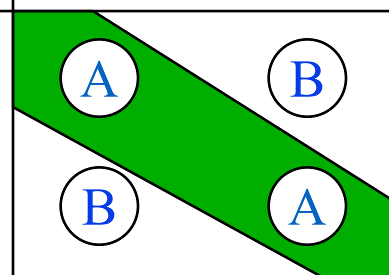
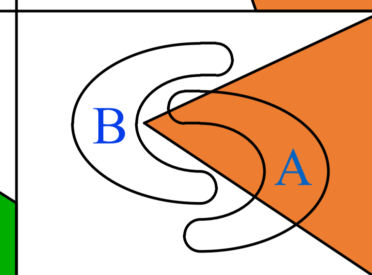
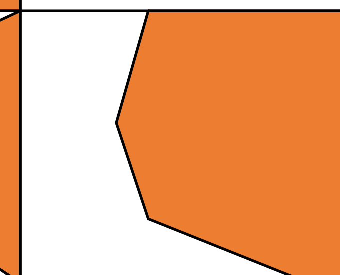
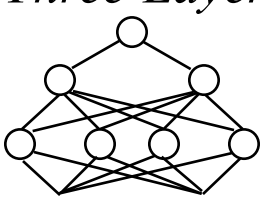
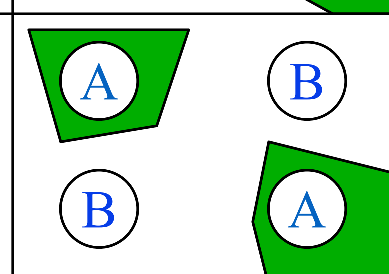
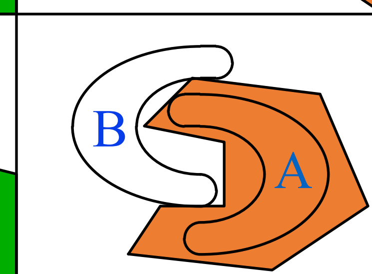
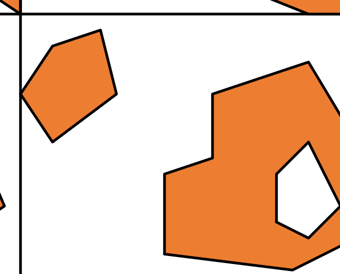
$$z_{1h} = \text{sigmoid}\left(\mathbf{w}_{1h}^T \mathbf{x}\right) = \text{sigmoid}\left(\sum_{j=1}^{d} w_{1hj} x_j + w_{1h0}\right), h = 1,\ldots,H_1$$

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

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

Different non-linearly separable problems

Neural Networks – An Introduction Dr. Andrew Hunter

Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
Single-Layer 	Half Plane Bounded By Hyperplane			
Two-Layer 	Convex Open Or Closed Regions			
Three-Layer 	Abitrary (Complexity Limited by No. of Nodes)			

A MLP Neural Network Demo

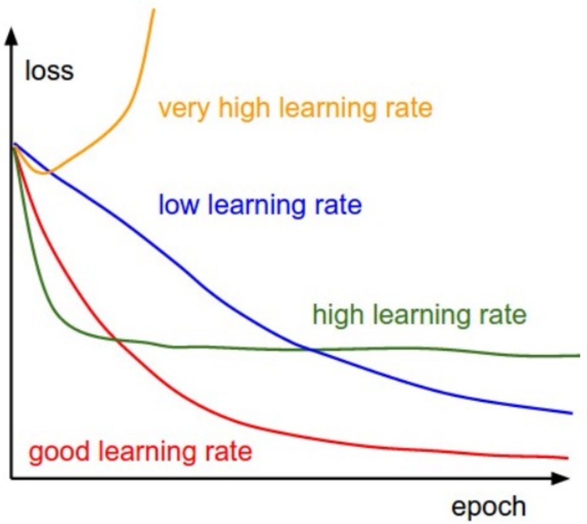
Improving Convergence

- Momentum

$$\Delta w_i^t = -\eta \frac{\partial E^t}{\partial w_i} + \alpha \Delta w_i^{t-1}$$

- Adaptive learning rate

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



Gradient descent with different learning rates. [Source]

Overfitting (with more complex NN)

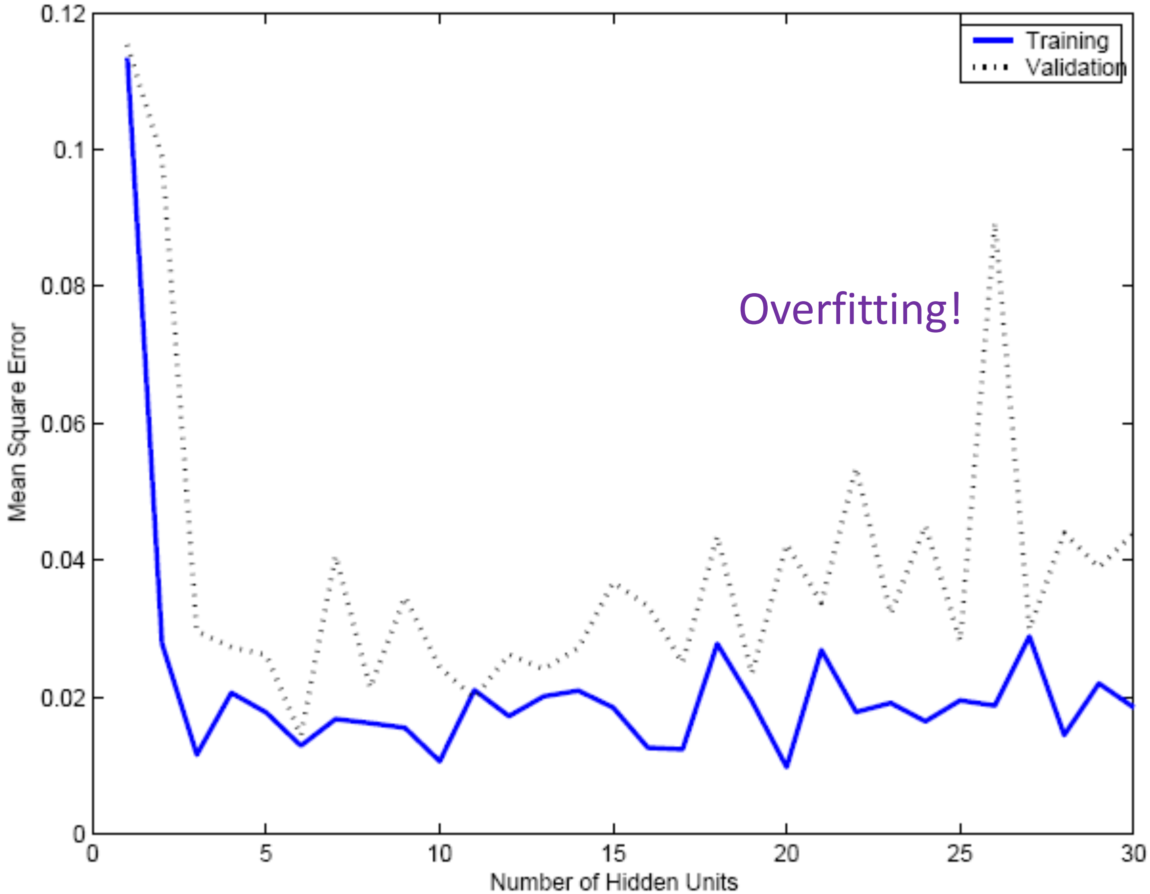


Figure 11.12 As complexity increases, training error is fixed but the validation error starts to increase and the network starts to overfit.

Number of weights:

H(*d*+1)+(*H*+1)*K* *for MLP with d inputs, H hidden units (1 hidden layer only), and K outputs*

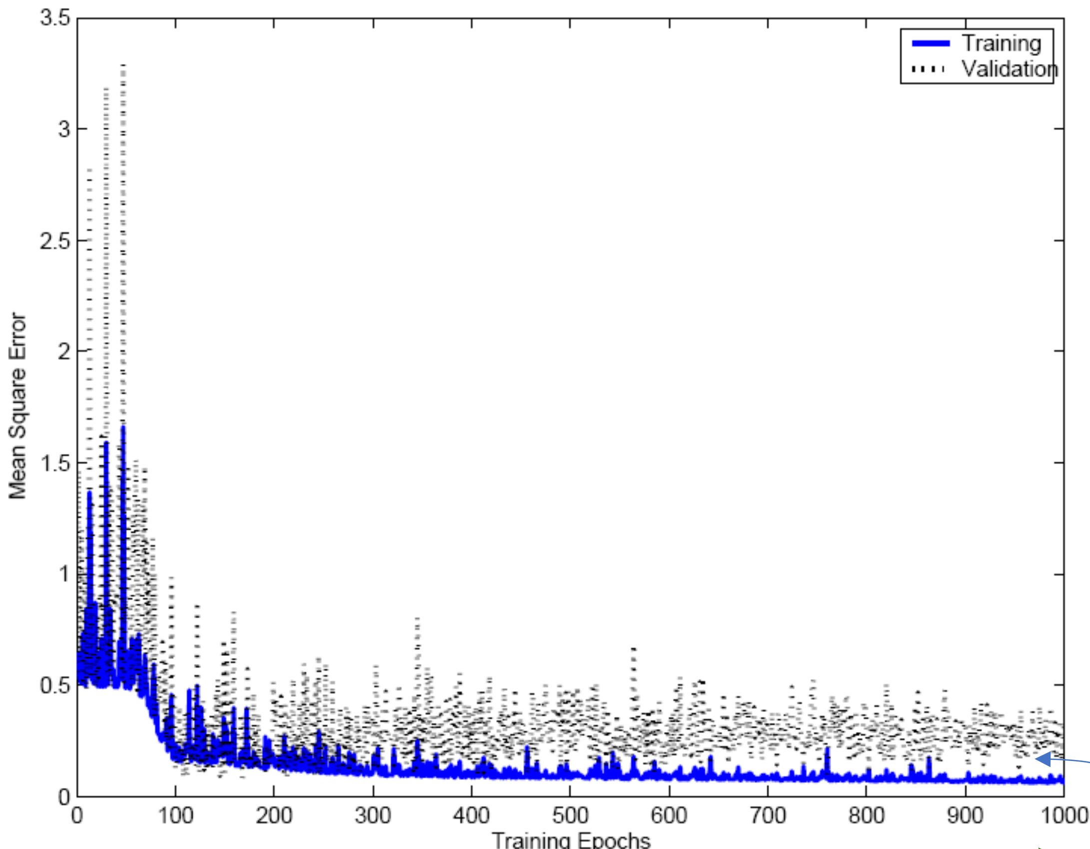
Space complexity:

O(*H*·(*d*+K)) *for MLP*

Time complexity:

O(*e* ·*H*·(*d*+K)) *where **e** is the number of training epochs*

Overfitting (with Overtraining)

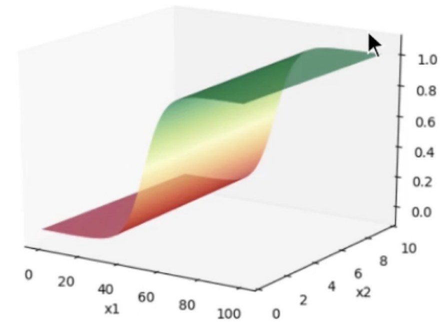


Discrepancy becomes larger!

More epochs (training)

Figure 11.13 As training continues, the validation error starts to increase and the network starts to overfit.

Loss function of sigmoid modified linear discriminant



- General loss function

$$loss = \sum_i (y_i - \hat{y}_i)^2$$

  i.e. the sum of the squared difference between the true output y_i and the predicted output $\hat{y}_i$

- An example:

x_1	x_2	y	$\hat{y}$
1	1	0.5	0.6
2	1	0.8	0.7
1	2	0.2	0.2
2	2	0.9	0.5

y could be binary, i.e., 0 or 1.

$$\hat{y} = \frac{1}{1 + e^{-(wx+b)}}$$

$$Loss = \sum_{i=1}^{4} (y - \hat{y})^2 = 0.18$$

Gradient Descent Learning

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient.
- In other words, move in a direction opposite to the gradient.

Parameter Update Rule

$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$

$$where\ \Delta w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t, b=b_t}, \Delta b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at w=w_t, b=b_t}$$

$$w = w + \Delta w$$
$$b = b + \Delta b$$

OR

$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$

Learning Algorithm

Initialise w, b

Iterate over data:

$compute\ \hat{y}$

$compute\ \mathscr{L}(w, b)$

$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$

till satisfied

Gradient Descent Optimization

Derivative of Sigmoid Function

$$\nabla w = \frac{\partial}{\partial w}\left[\frac{1}{2} * (f(x) - y)^2\right]$$

$$= \frac{1}{2} * \left[2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)\right]$$

$$= (f(x) - y) * \frac{\partial}{\partial w}(f(x))$$

$$= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right)$$

$$= (f(x) - y) * f(x) * (1 - f(x)) * x$$

$$\frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right)$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)}))$$

$$= \frac{-1}{(1+e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx+b)))$$

$$= \frac{-1}{(1+e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})} * (-x)$$

$$= \frac{1}{(1+e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})} * (x)$$

$$= f(x) * (1 - f(x)) * x$$

An Example

Emergency Room Visits	Narcotics	Pain	Total Visits	Medical Claims	PoorCare
0	2	6	11	53	1
1	1	4	25	40	0
0	0	5	10	28	0
1	3	5	7	20	1

Initialise $w_1, w_2, \ldots, w_5, b$

Iterate over data:

$$w_1 = w_1 - \eta \Delta w_1$$

$$w_2 = w_2 - \eta \Delta w_2$$

$$\vdots$$

$$w_5 = w_5 - \eta \Delta w_5$$

$$b = b - \eta \Delta b$$

till satisfied

$$z = w_1 * ER\_visits + w_2 * Narcotics + w_3 * Pain + w_4 * TotalVisits + w_5 * MedicalClaims + b$$

$$z = w_1 * x_{i1} + w_2 * x_{i2} + w_3 * x_{i3} + w_4 * x_{i4} + w_5 * x_{i5} + b$$

$$\hat{y} = \frac{1}{1+e^{-z}}$$

$$\hat{y} = \frac{1}{1+e^{-(w_1 * x_{i1} + w_2 * x_{i2} + w_3 * x_{i3} + w_4 * x_{i4} + w_5 * x_{i5} + b)}}$$

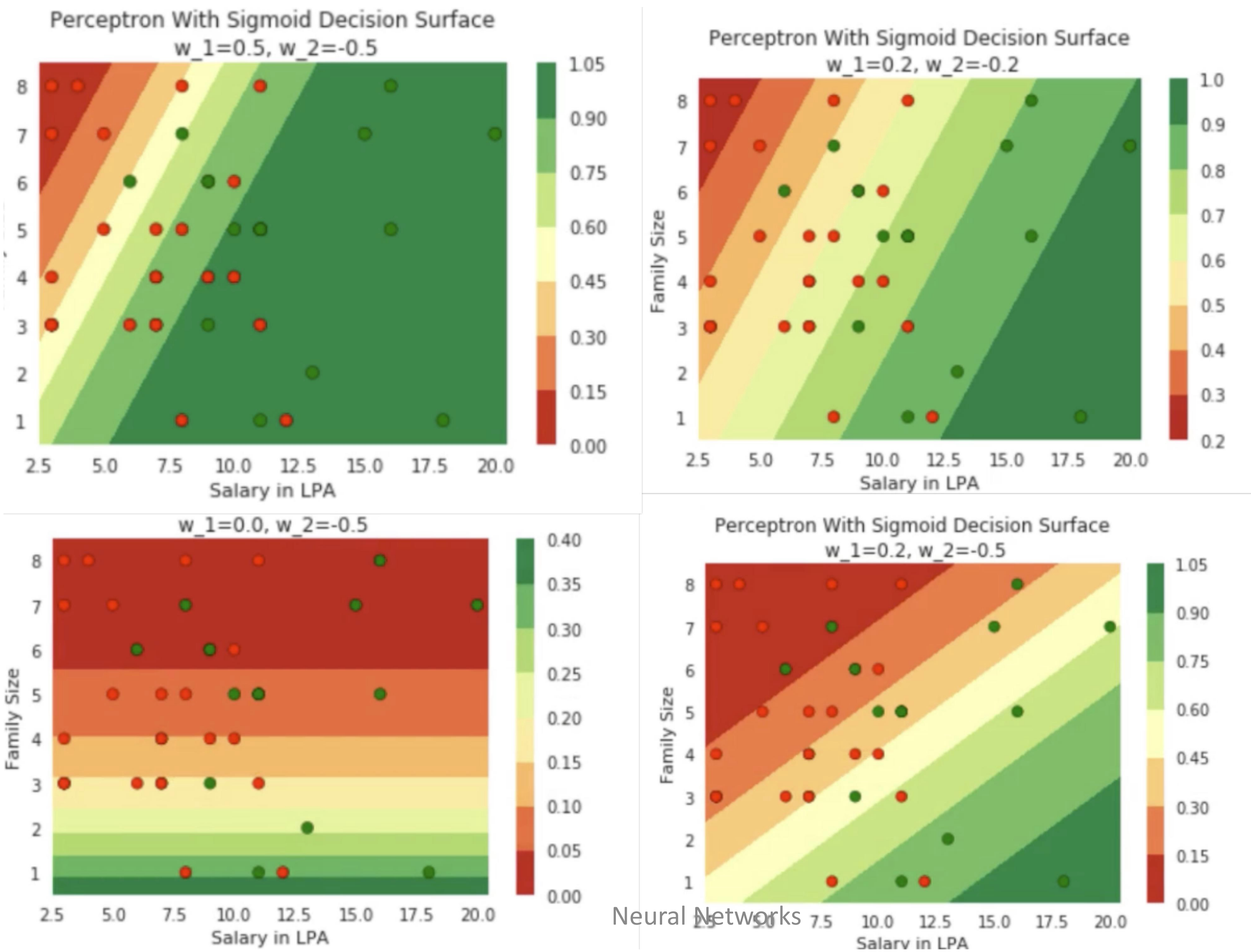
$$\Delta w = \sum_{i=1}^{m} (f(x) - y) * f(x) * (1 - f(x)) * x$$

$$\Delta w_1 = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{i1}$$

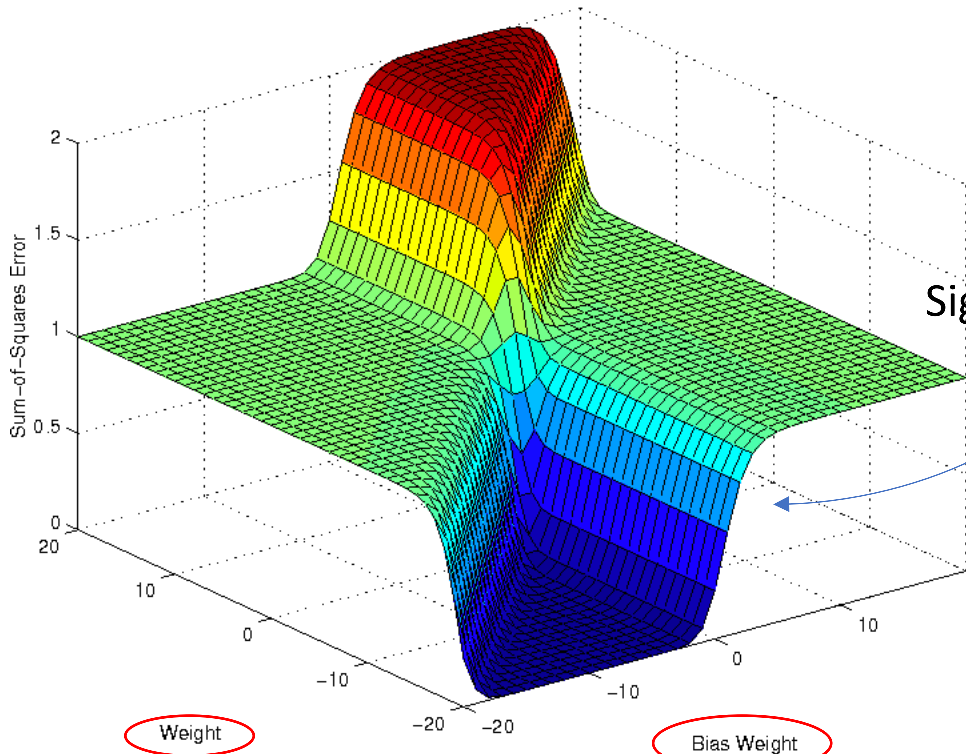
$$\Delta w_2 = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{i2}$$

$$\Delta w_j = \sum_{i=1}^{m} (\hat{y} - y) * \hat{y} * (1 - \hat{y}) * x_{ij}$$

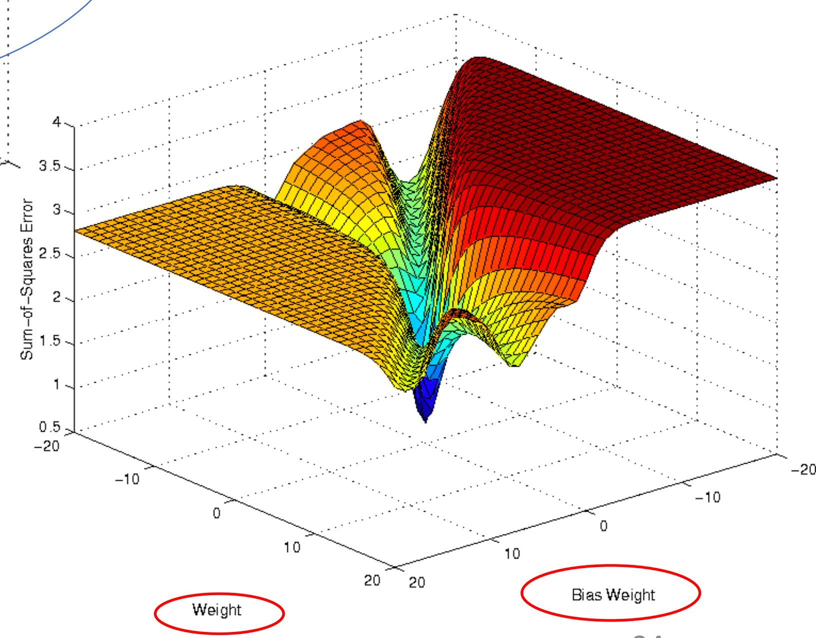
Changes in sigmoid decision surface



Decision surface -> optimization space (yet another space you need to understand!)

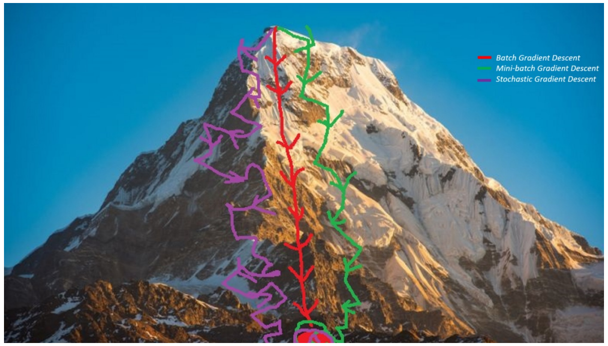


Sigmoid S shape



Here, only 2 parameters!

How to iterate over data?



- **Batch Gradient Descent**
  - uses the whole batch of training data at every step
  - calculates the error for each record and takes an average to determine the gradient
- **Stochastic Gradient Descent (SGD)**
  - just picks one instance from training set at every step
  - update gradient only based on that single record
- **Mini-Batch Gradient Descent**
  - Hybrid of Batch type and SGD
  - Pick 1<n<N instance to update

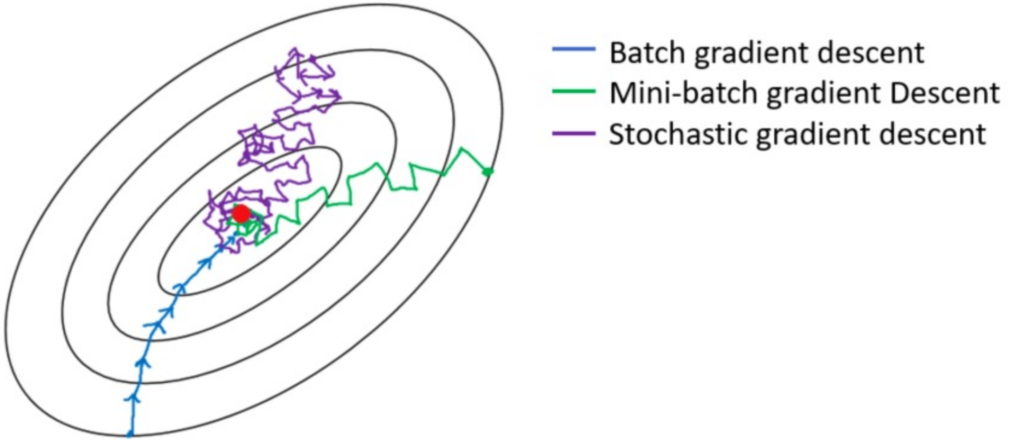


Image Source: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

till satisfied…how to measure?

- Simple ones

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y-\hat{y})^2}$$

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

Take-home messages

- The layered structure of NN makes it powerful!
- NN, originated from neuroscience, has very nice mathematical structure.
- Machine learning through optimization of error/loss function -> mathematical structure is important!
- Gradient descent is simple and well-fit for ML!

Acknowledgement

- Slides/Materials of
  - [1] E. Alpaydin, Introduction to Machine Learning. 2^{nd} Ed. MIT Press, 2010.
  - [2] https://medium.com/datadriveninvestor/simplified-sigmoid-neuron-a-building-block-of-deep-neural-network-5bfa75c8d8a9
- Photos from Internet