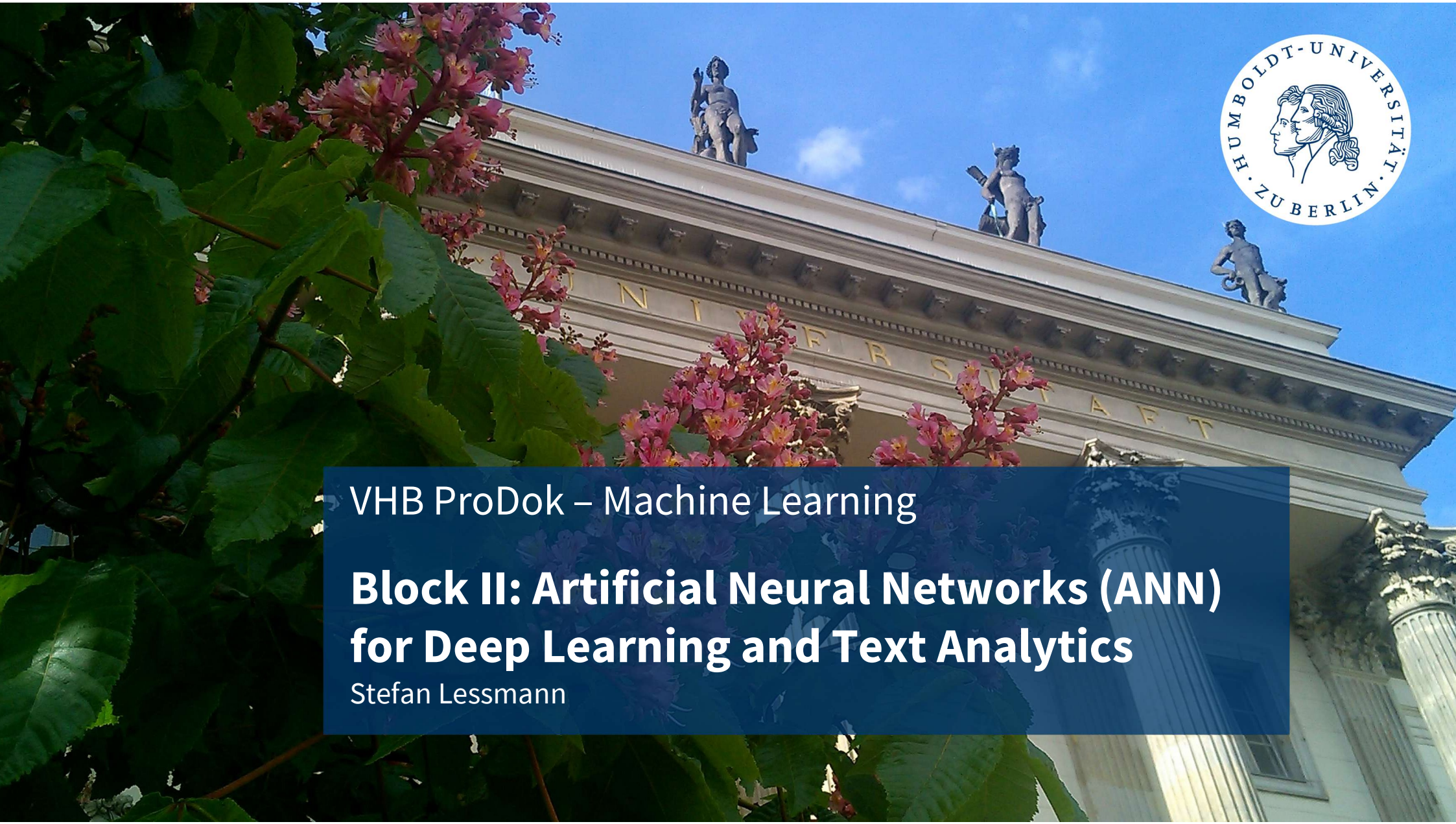
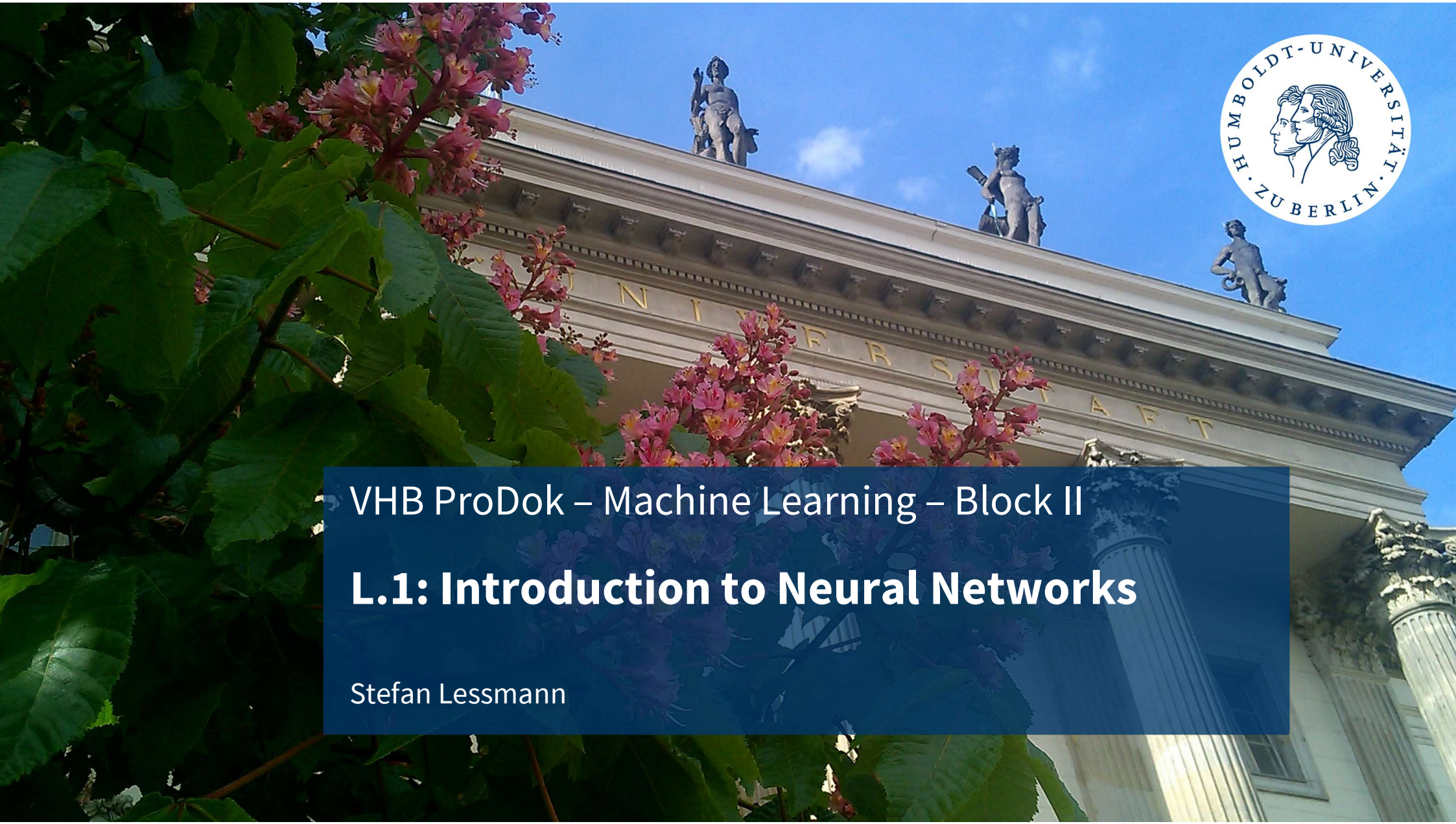


VHB ProDok – Machine Learning

Block II: Artificial Neural Networks (ANN) for Deep Learning and Text Analytics

Stefan Lessmann



VHB ProDok – Machine Learning – Block II

L.1: Introduction to Neural Networks

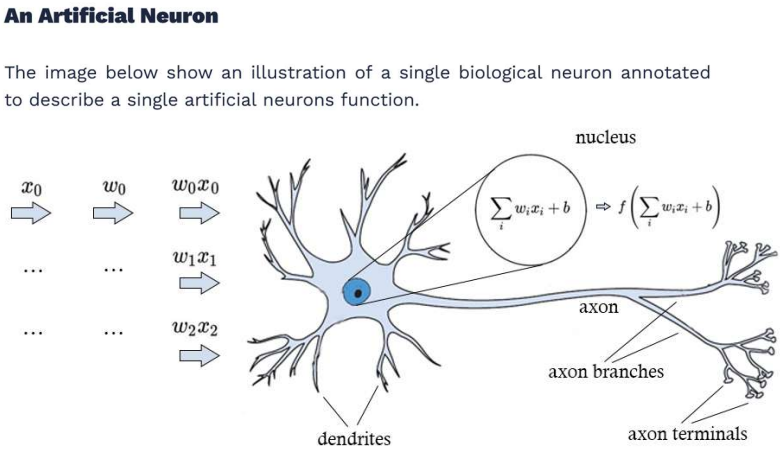
Stefan Lessmann

Artificial Neural Networks (ANN)

The backbone of deep learning and modern AI

- **Family of algorithms that draw inspiration from the functioning of the human brain**

Source: https://lilamed.de/was-sind-die-funktionen-von-neuronen/



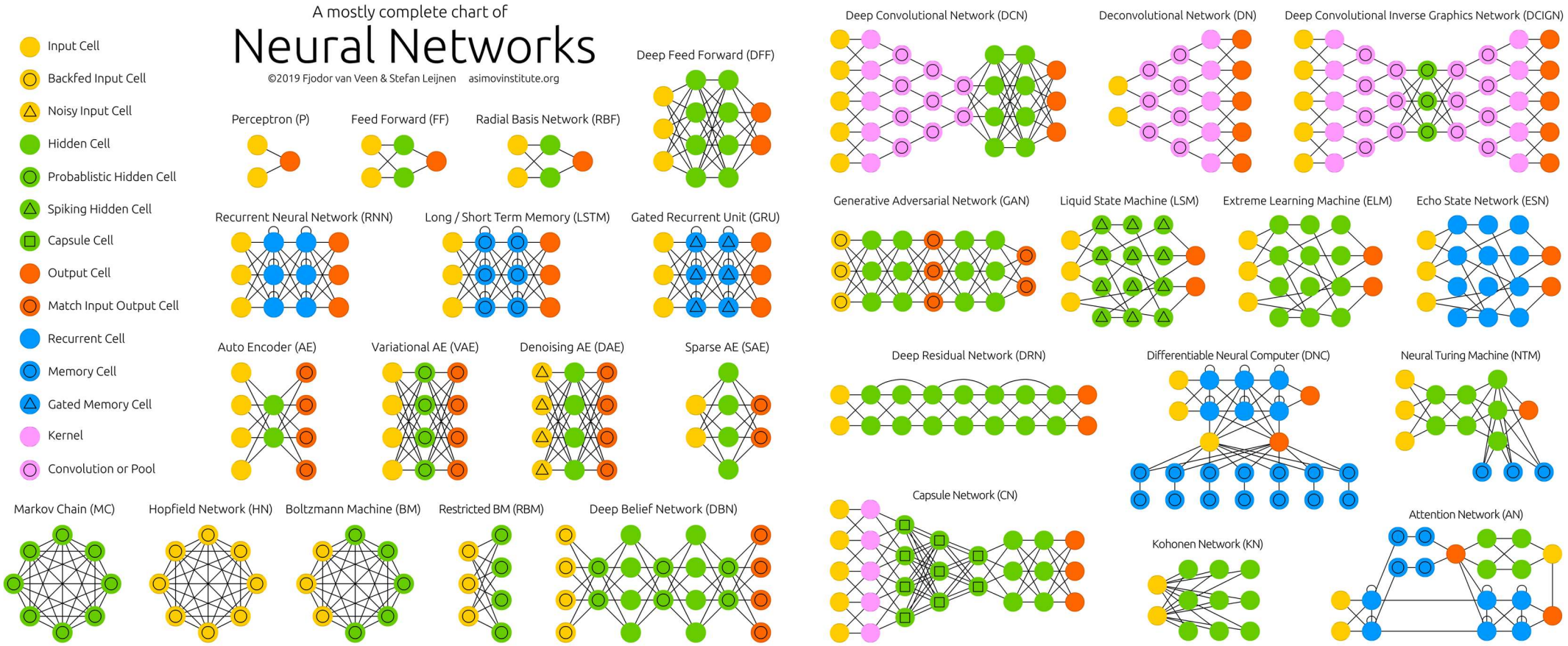
A biological neuron receives input signals from its dendrites from other neurons and sends output signals along its axon, which branches out and connects to other neurons. In the illustration above, the input signal is represented by x_0, as this signal 'travels' it is multiplied ($w_0\ x_0$) based on the a weight variable (w_0). The weight variables are learnable and the weights strength and polarity (positive or negative) control the influence of the signal. The influence is determined by summing the signal input and weight $\left(\sum_i w_j x_j + b\right)$ which is then calculated by the activation function f, if it is above a certain threshold the neuron fires.

Source: https://krisbolton.com/a-quick-introduction-to-artificial-neural-networks-part-1

- **Or, more formally, a semi-parametric model for nonlinear regression and related data analytics problems**

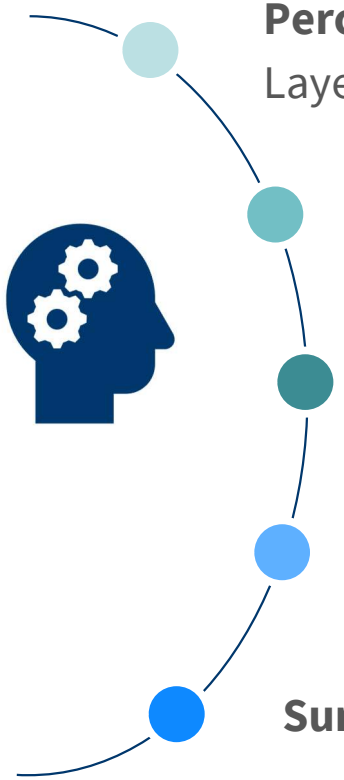
Deep Learning is More Than Adding Layers to Feedforward Networks

By F. Van Veen and S. Leijnen from The Asimov Institute



Source: Van Veen, F. & Leijnen, S. (2019). The Neural Network Zoo. Retrieved from https://www.asimovinstitute.org/neural-network-zoo

Agenda



Perceptrons and neural networks
Layered architectures, nonlinearity, and deep networks

Neural network training
Backpropagation, gradient descent, and training parameters

Applications of Artificial Neural Networks
Interplay between modeling task & network architecture

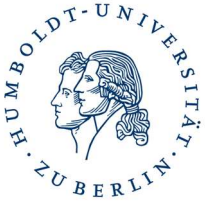
Regularization and model selection
Penalties, early stopping, dropout, and random search

Summary

Perceptrons and Neural Networks

Layered architecture, nonlinearity and deep networks

Data for Training Supervised ML Models for Regression



Abstraction

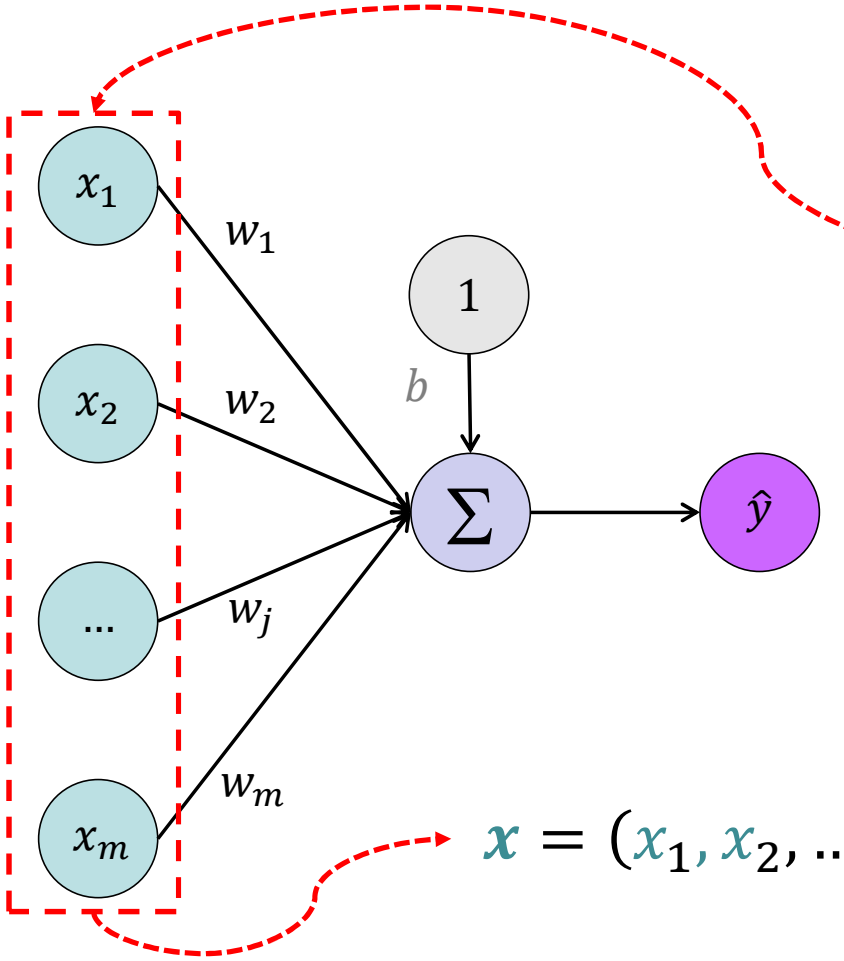
LIST PRICE [$]	AGE [month]	PRODUCT	…	**RESALE PRICE [$]**
2,500	36	Dell XPS 15’	…	347
2,500	24	Dell XPS 15’	…	416
3,000	36	Dell XPS 17’	…	538
1,300	24	HP Envy 17’	…	121
1,900	36	HP EliteBook 850	…	172
799	12	Lenovo Yoga 11’	…	88
1,100	12	Lenovo Yoga 13’	…	266
…	…	…	…	…

TRAINING DATA					
i	X_1	X_2	…	X_m	Y
1	x_{11}	x_{12}		x_{1m}	y_1
2	x_{21}	x_{22}		x_{2m}	y_2
…	…	…	…	…	…
…	…	…	…	…	…
…	…	…	…	…	…
n	x_{n1}	x_{n2}		x_{nm}	y_n

$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ an arbitrary data instance

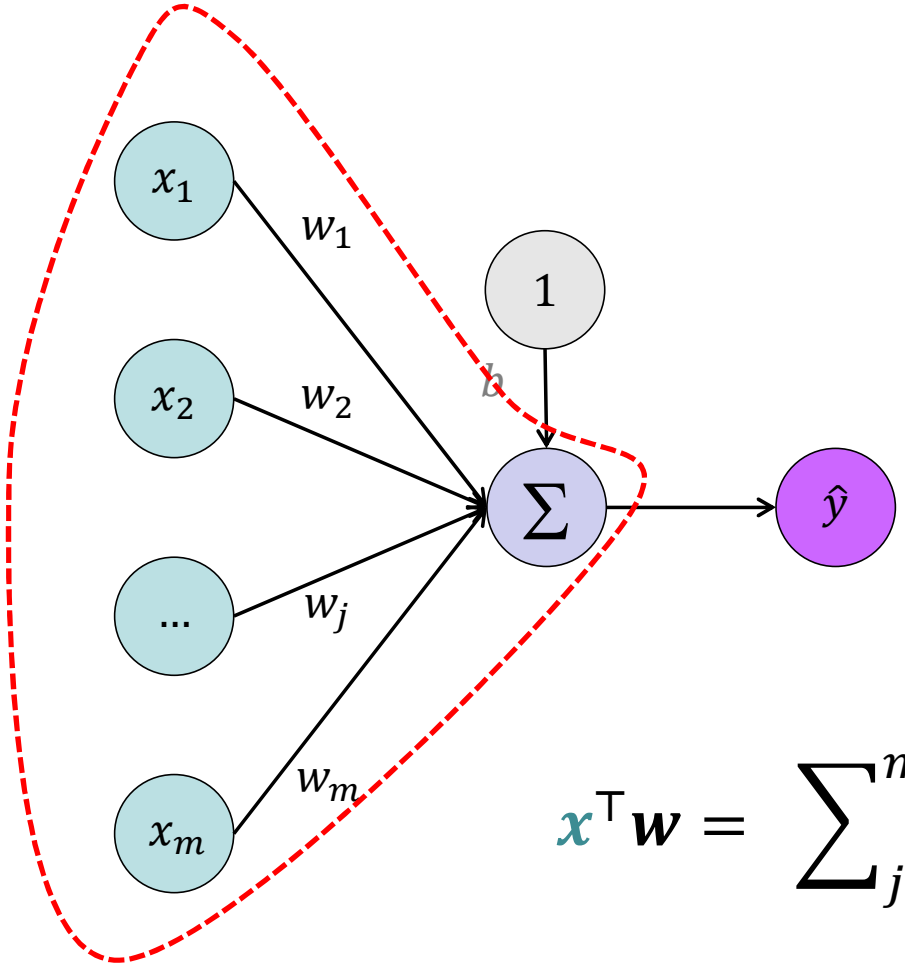
$\boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im})$ a specific data instance

Linear Regression as Directed Graph



TRAINING DATA						
i	X_1	X_2	...	X_m	Y	$\hat{Y}$
1	x_{11}	x_{12}	...	x_{1m}	y_1	$\hat{y}_1$
2	x_{21}	x_{22}	...	x_{2m}	y_2	$\hat{y}_2$
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
n	x_{n1}	x_{n2}	...	x_{nm}	y_n	$\hat{y}_n$

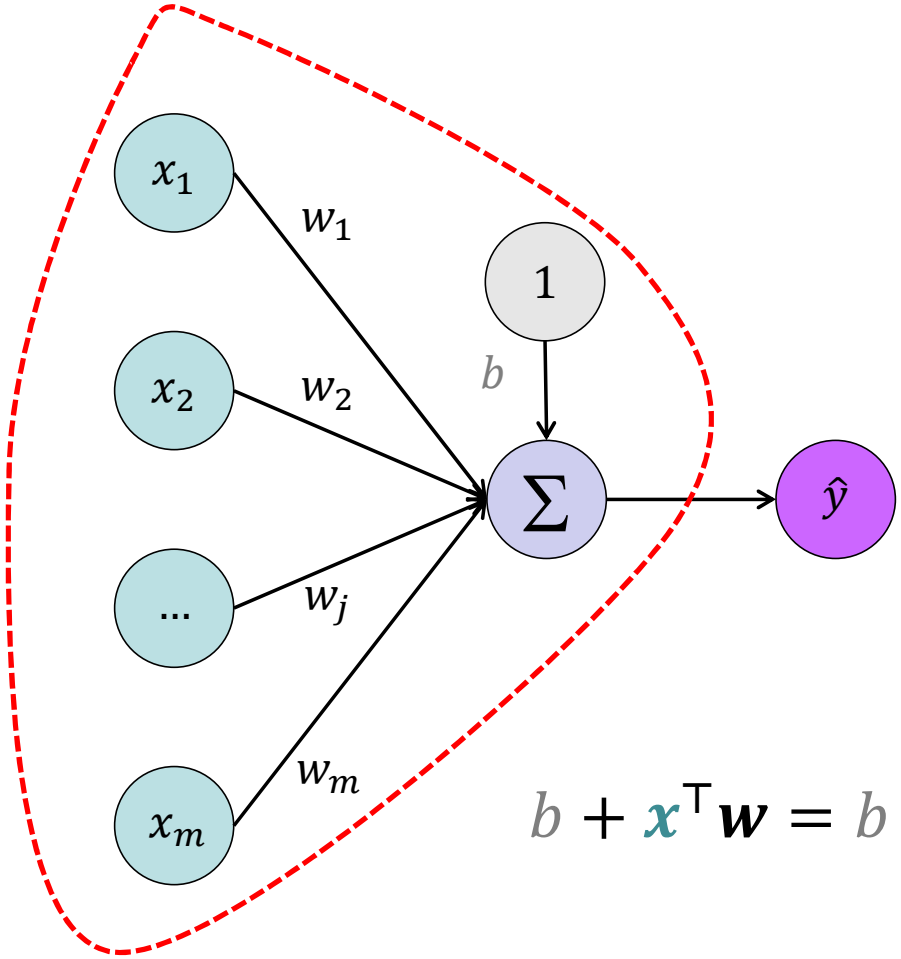
Linear Regression as Directed Graph



$$\boldsymbol{x}^\top \boldsymbol{w} = \sum_{j=1}^{m} w_j x_j$$

TRAINING DATA						
i	X_1	X_2	...	X_m	Y	$\hat{Y}$
1	x_{11}	x_{12}	...	x_{1m}	y_1	$\hat{y}_1$
2	x_{21}	x_{22}	...	x_{2m}	y_2	$\hat{y}_2$
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
n	x_{n1}	x_{n2}	...	x_{nm}	y_n	$\hat{y}_n$

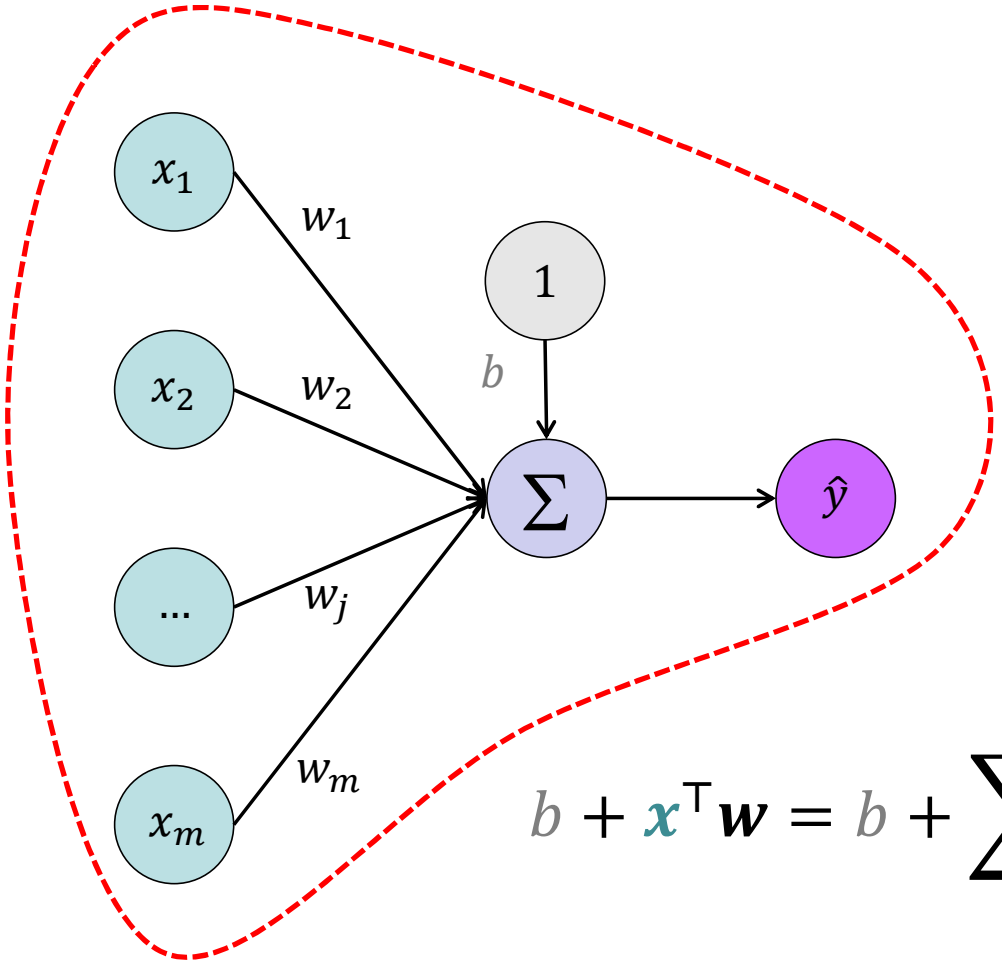
Linear Regression as Directed Graph



$$b + \boldsymbol{x}^\top \boldsymbol{w} = b + \sum_{j=1}^{m} w_j x_j$$

TRAINING DATA						
i	X_1	X_2	...	X_m	Y	$\hat{Y}$
1	x_{11}	x_{12}	...	x_{1m}	y_1	$\hat{y}_1$
2	x_{21}	x_{22}	...	x_{2m}	y_2	$\hat{y}_2$
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
n	x_{n1}	x_{n2}	...	x_{nm}	y_n	$\hat{y}_n$

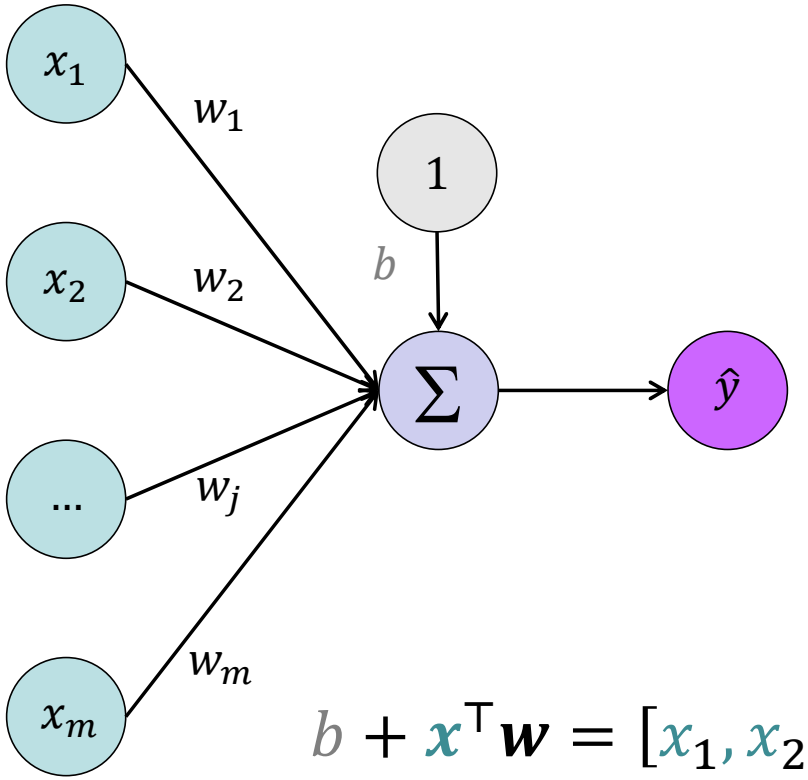
Linear Regression as Directed Graph



TRAINING DATA						
i	X_1	X_2	...	X_m	Y	$\hat{Y}$
1	x_{11}	x_{12}	...	x_{1m}	y_1	$\hat{y}_1$
2	x_{21}	x_{22}	...	x_{2m}	y_2	$\hat{y}_2$
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
n	x_{n1}	x_{n2}	...	x_{nm}	y_n	$\hat{y}_n$

$$b + \boldsymbol{x}^\top \boldsymbol{w} = b + \sum_{j=1}^{m} w_j x_j = \hat{y}$$

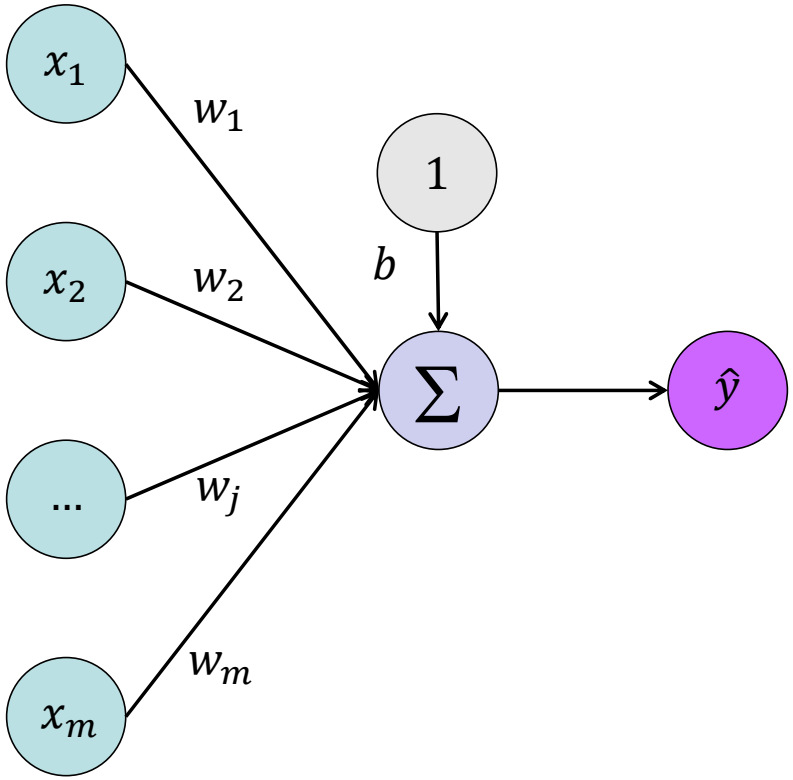
Linear Regression as Directed Graph in Matrix Notation



TRAINING DATA						
i	X_1	X_2	...	X_m	Y	$\hat{Y}$
1	x_{11}	x_{12}	...	x_{1m}	y_1	$\hat{y}_1$
2	x_{21}	x_{22}	...	x_{2m}	y_2	$\hat{y}_2$
...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
n	x_{n1}	x_{n2}	...	x_{nm}	y_n	$\hat{y}_n$

$$b + \boldsymbol{x}^\top \boldsymbol{w} = [x_1, x_2, \ldots, x_m] \times \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} + b = \hat{y}$$

Linear Regression as Directed Graph



Weighted sum over the inputs

Output

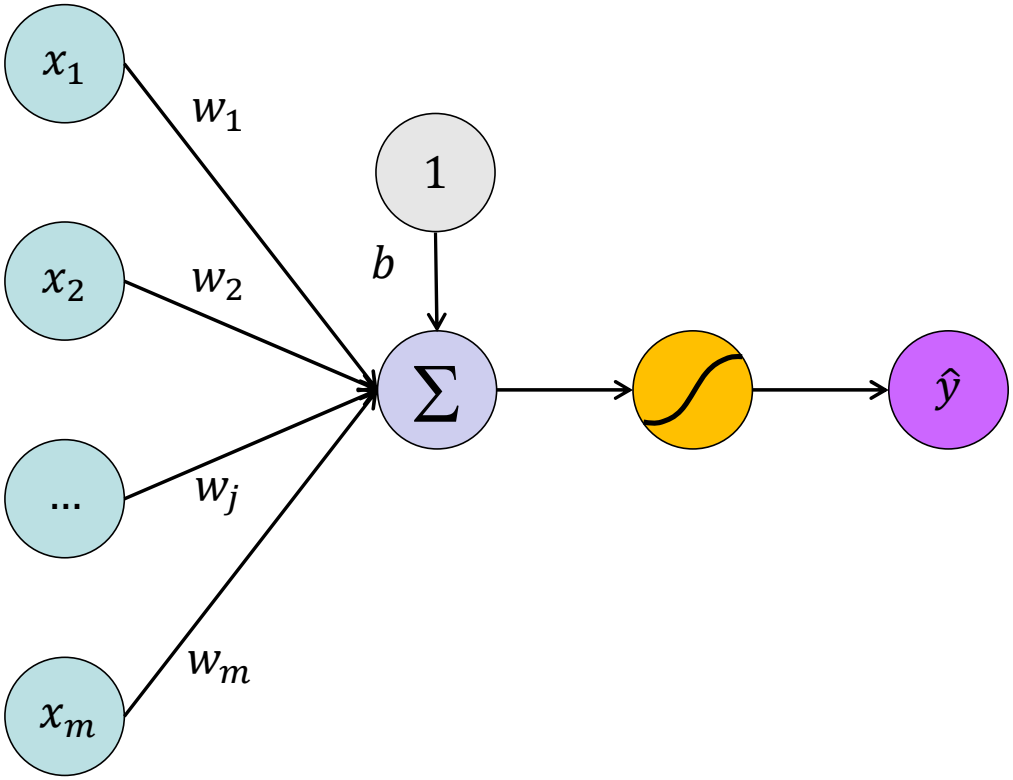
$$\hat{y} = b + \sum_{j=1}^{m} w_j x_j$$

Bias

Linear combination

A Graphical Model of Generalized Linear Models (GLM)

This model is also known as Percenptron (Rosenblatt 1958)



Output

Bias

Weighted sum over the inputs

$$\hat{y} = g\left(b + \sum_{j=1}^{m} w_j x_j\right)$$

Nonlinear transfer or activation function

Linear combination

Common Choices for the Transfer / Activation Function

Nonlinear transformation of neuron input to neuron output

$$\hat{y} = g\left(b + \sum_{j=1}^{m} w_j x_j\right)$$

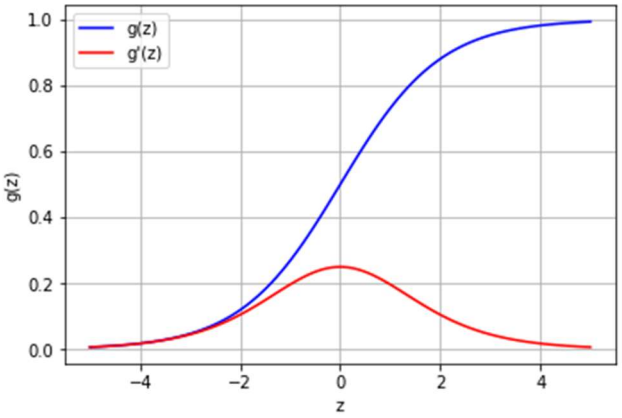
Output: $\hat{y}$

Bias: b

Nonlinear transfer or activation function: g

Linear combination: $\sum_{j=1}^{m} w_j x_j$

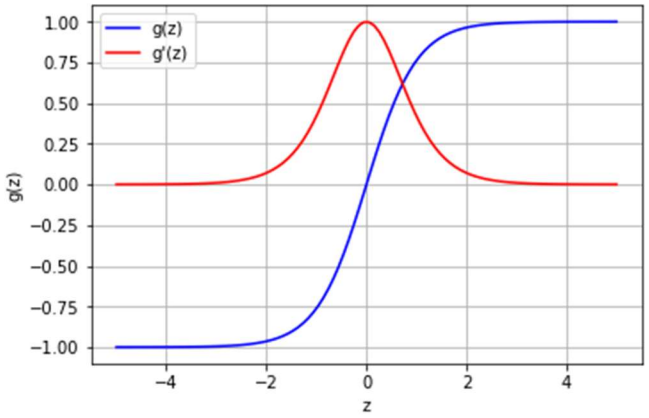
Sigmoid



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

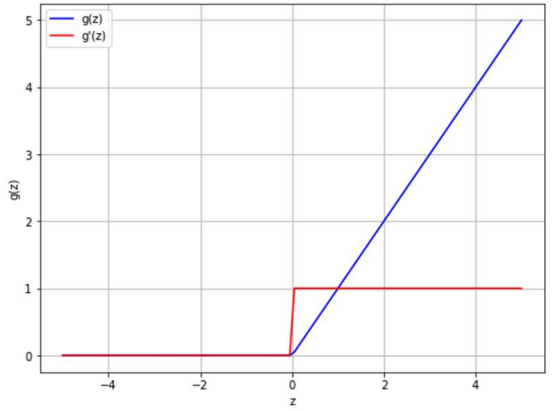
Hyperbolic tangent (tanH)



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

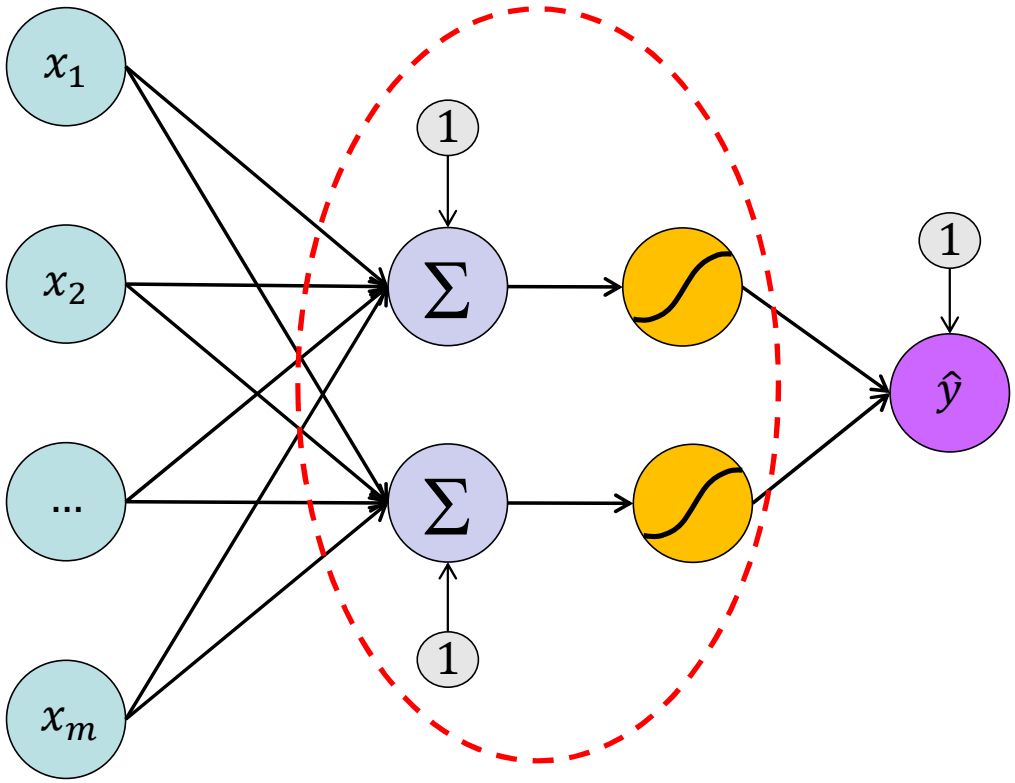
Rectified linear unit (RELU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Neural Network (NN) with one Hidden Layer

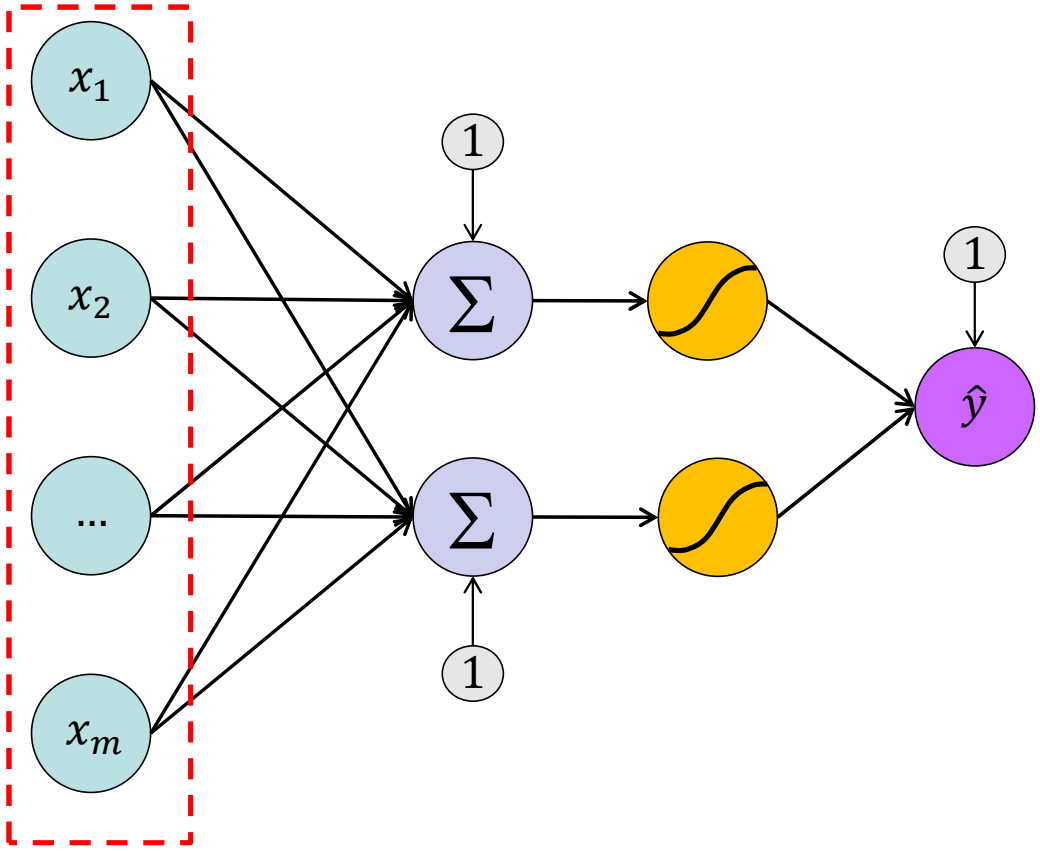


Hidden layer with $d = 2$ nodes (aka neurons)

The hidden layer (s) in a NN incorporate(s) computational units (aka hidden nodes). These hidden nodes function exactly as the perceptron introduced above.
Specifically, they:

- receive input signals (i.e., data)
- aggregate these signals by a weighted sum
- put the resulting aggregate to a transfer function
- and pass the result on to subsequent nodes and eventually the output node(s)

Neural Network (NN) with one Hidden Layer – More Formally

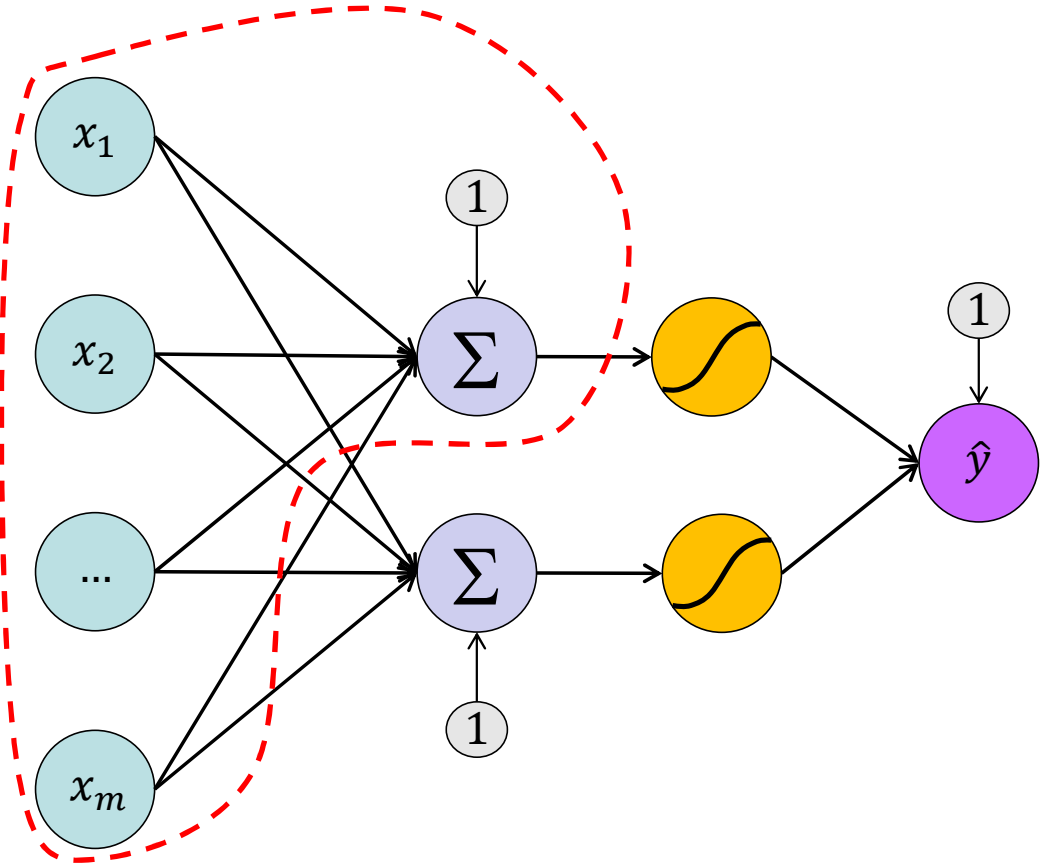


$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$$

$$\boldsymbol{x} = (2500, 36, \text{Dell XPS } 15' \ldots)$$

LIST PRICE [$]	AGE [month]	PRODUCT	…	**RESALE PRICE [$]**
2,500	36	Dell XPS 15’	…	347
2,500	24	Dell XPS 15’	…	416
3,000	36	Dell XPS 17’	…	538
1,300	24	HP Envy 17’	…	121
1,900	36	HP EliteBook 850	…	172
799	12	Lenovo Yoga 11’	…	88
1,100	12	Lenovo Yoga 13’	…	266
…	…	…	…	…

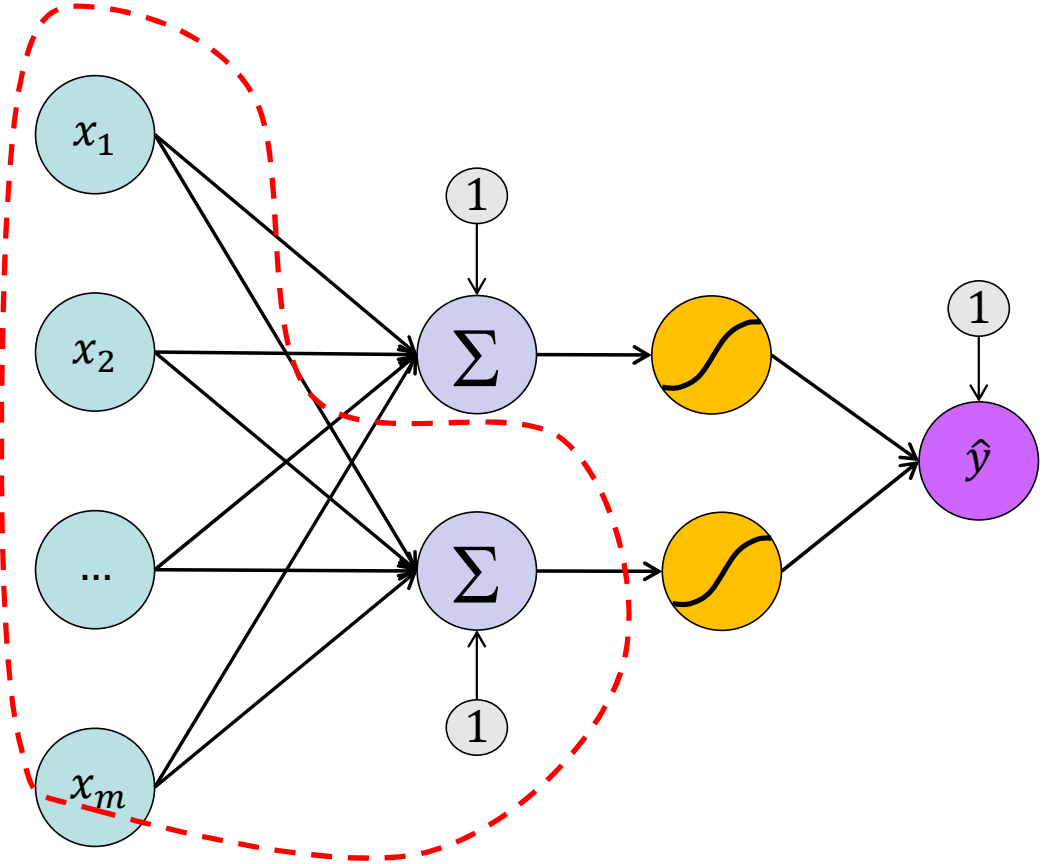
Neural Network (NN) with one Hidden Layer – More Formally



$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$$

$$z = b + \sum_{j=1}^{m} w_j x_j$$

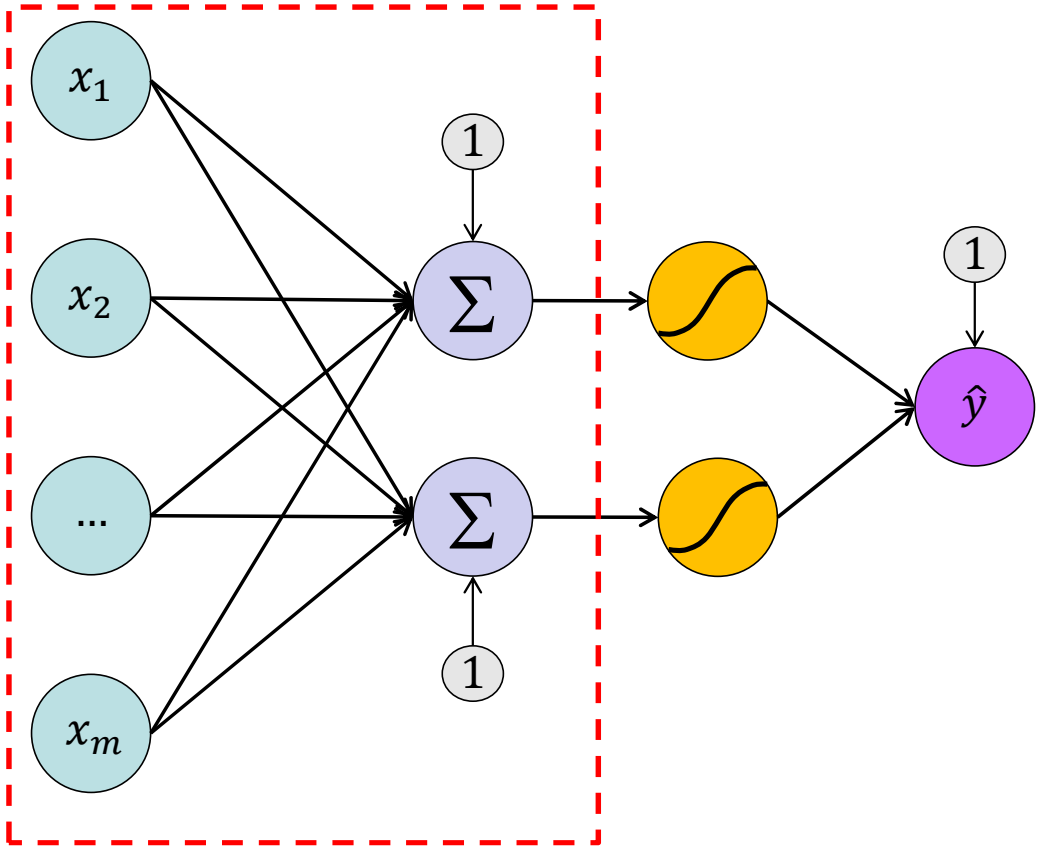
Neural Network (NN) with one Hidden Layer – More Formally



$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$$

$$z = b + \sum_{j=1}^{m} w_j x_j$$

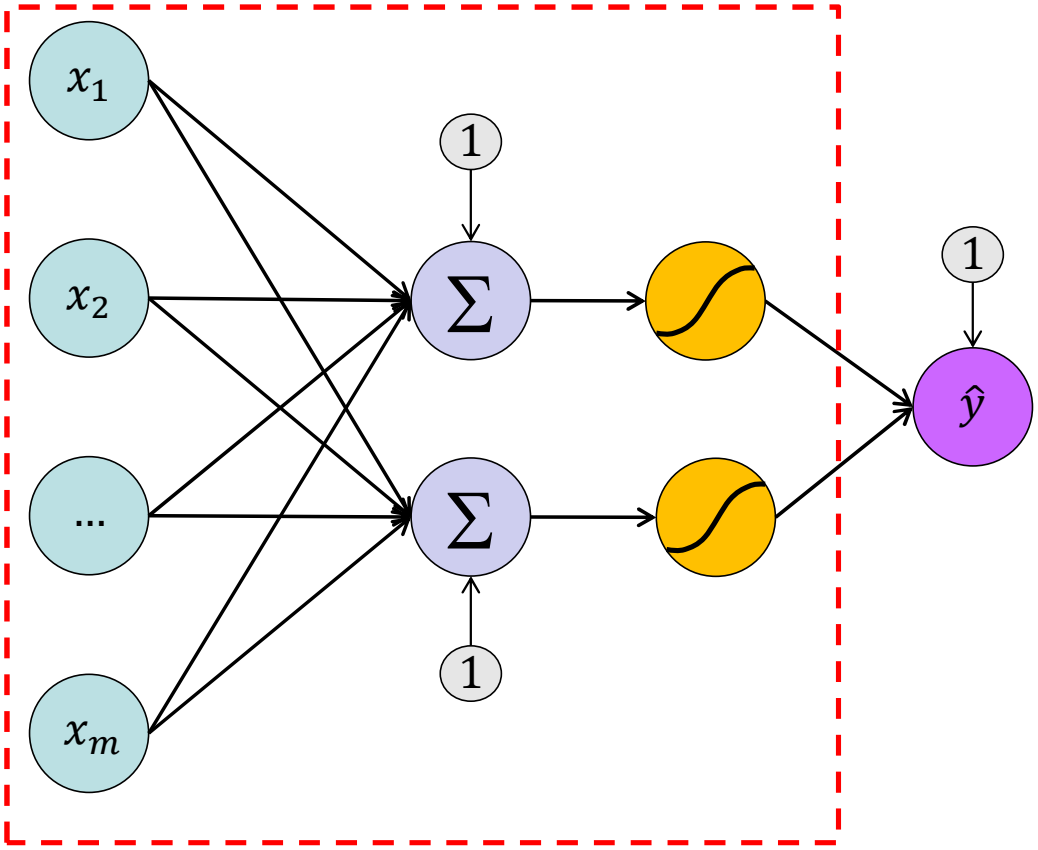
Neural Network (NN) with one Hidden Layer – More Formally



$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$$

$$z_k = b_k + \sum_{j=1}^{m} w_{kj} x_j$$

Neural Network (NN) with one Hidden Layer – More Formally

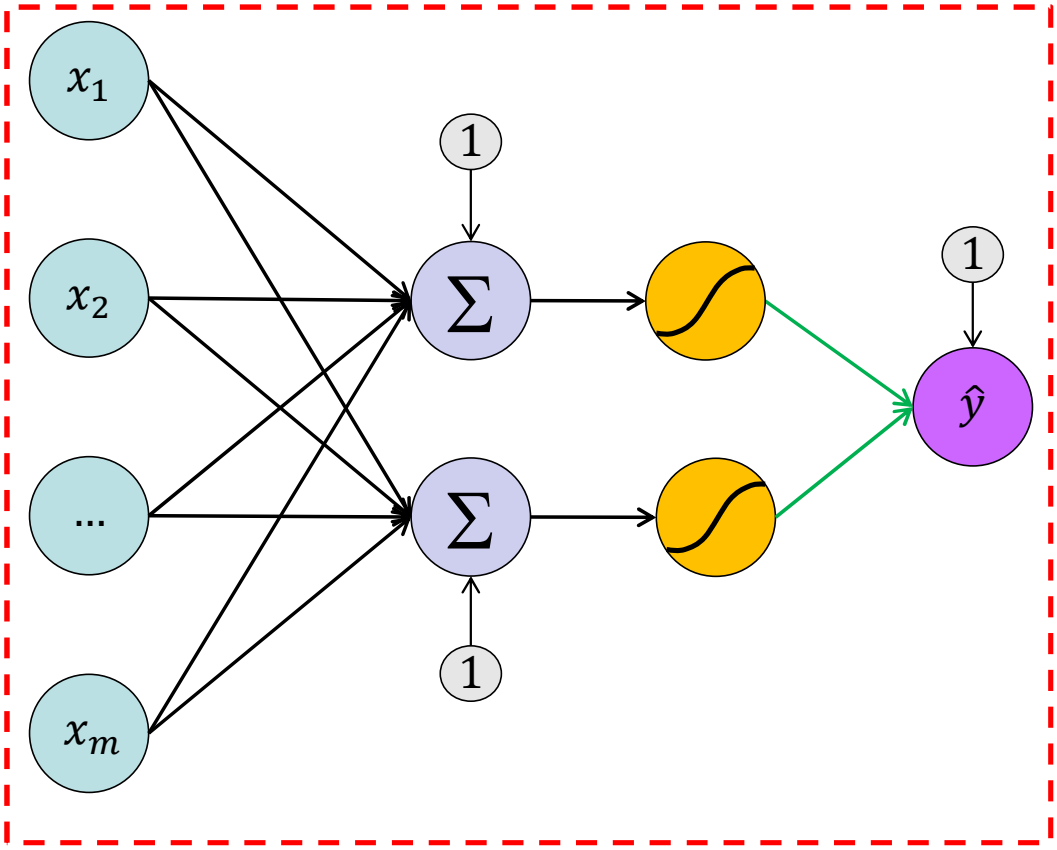


$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$$

$$z_k = b_k + \sum_{j=1}^{m} w_{kj} x_j$$

$$h_k = g(z_k)$$

Neural Network (NN) with one Hidden Layer – More Formally



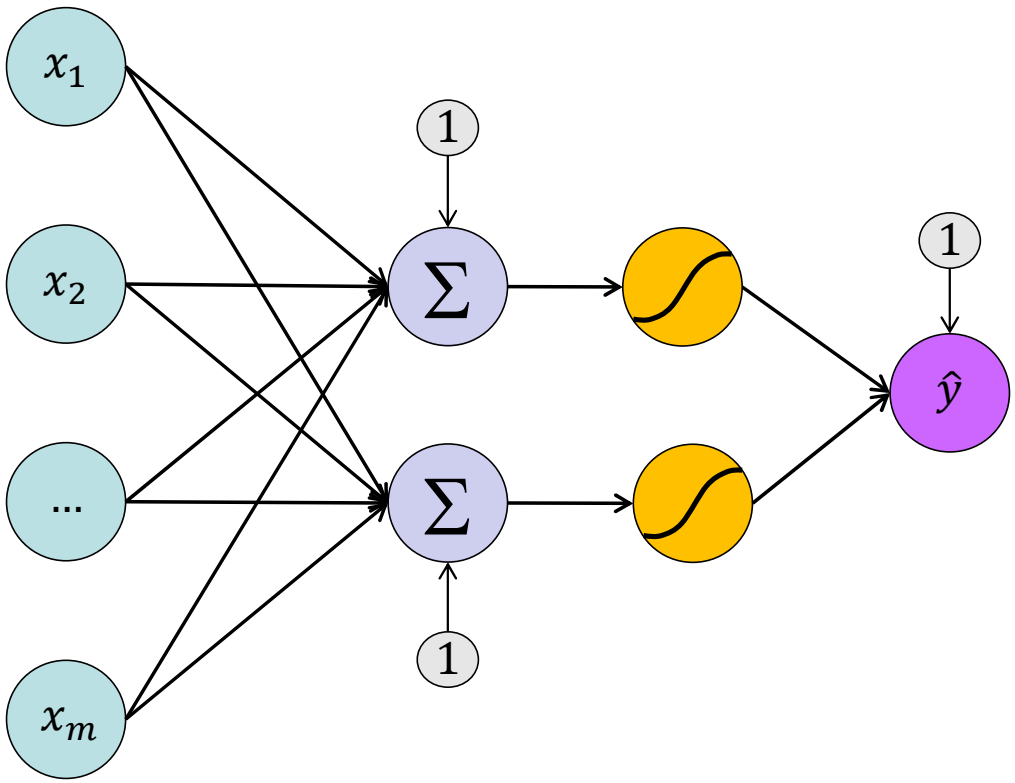
$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$$

$$z_k = b_k + \sum_{j=1}^{m} w_{kj} x_j$$

$$h_k = g(z_k)$$

$$\hat{y} = b + \sum_{k=1}^{2} v_k h_k$$

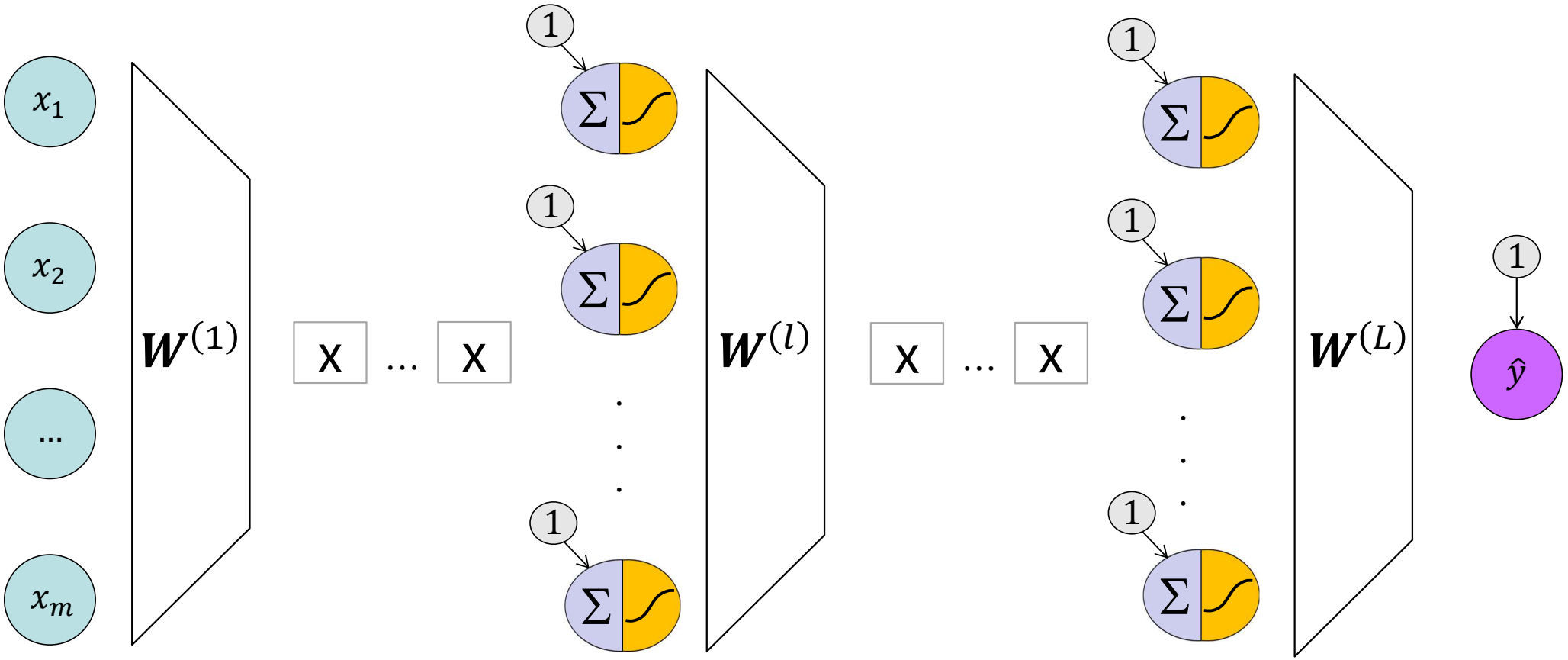
Neural Network (NN) with one Hidden Layer – More Formally




```
weighted sum[
   nonlinearity(
      weighted sum[]
   )
]
```

$$\hat{y} = b + \sum_{k=1}^{2} v_k g\left(b_k + \sum_{j=1}^{m} w_{kj} x_j\right)$$

Generalization to Deep Neural Networks

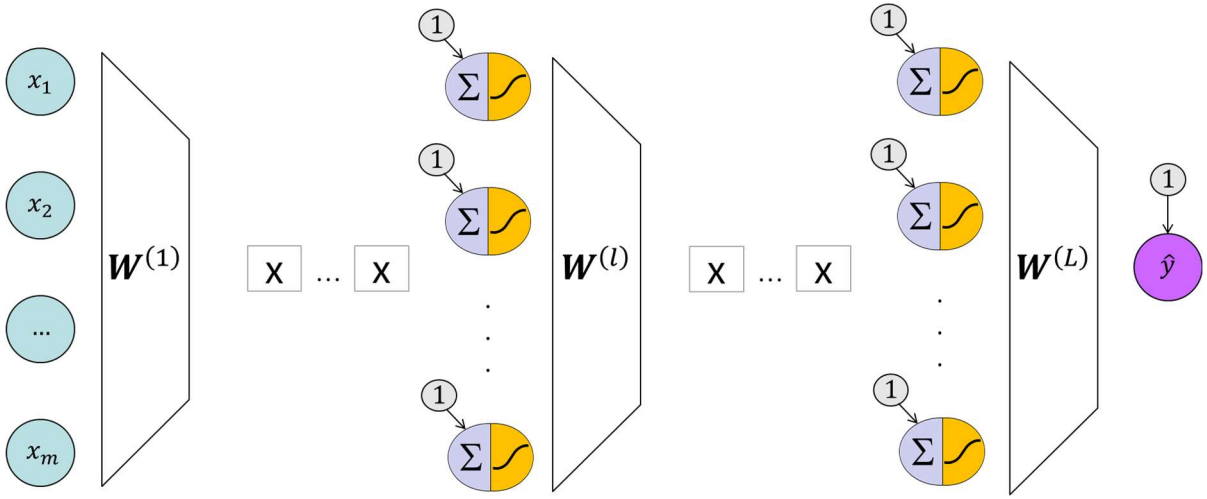


Input layer

$L-1$ hidden layers

Output layer

Formalization of the Deep Neural Network



Data point: $X = \{x_1, x_2, \dots, x_m\} \in R^m$

Network with $L-1$ hidden layers of d_l nodes l

Node input: $z_k^{(l)}$ with $k = 1, 2, \dots, d_l$

Node output : $h_k^{(l)}$ with $k = 1, 2, \dots, d_l$

Node bias: $b_k^{(l)}$ with $k = 1, 2, \dots, d_l$

In- and output of hidden layer l of dimension $d_l \times 1$: $Z^{(l)}$ respectively $H^{(l)}$

Connection weight (parameter) from node j in layer $l-1$ to node k in layer l: w_{kj}

Matrix of connection weights in layer l with dimension $d_l \times d_{l-1}$: $\boldsymbol{W}^{(l)}$

$$z_k^{(l)} = b_k^{(l)} + \sum_{j=1}^{d_{l-1}} w_{kj}^{(l)} h_j^{(l-1)}$$

$$Z^{(l)} = \begin{bmatrix} z_1^{(l)} \\ \vdots \\ z_{d_l}^{(l)} \end{bmatrix} = B^{(l)} + \boldsymbol{W}^{(l)} H^{(l-1)}$$

$$\boldsymbol{W}^{(l)} = \begin{bmatrix} w_{11} & \cdots & w_{1d_{l-1}} \\ \vdots & \ddots & \vdots \\ w_{d_l 1} & \cdots & w_{d_l d_{l-1}} \end{bmatrix}$$

$$h_k^{(l)} = g\left(z_k^{(l)}\right)$$

$$H^{(l)} = \begin{bmatrix} h_1^{(l)} \\ \vdots \\ h_{d_l}^{(l)} \end{bmatrix} = g\left(Z^{(l)}\right) = \begin{bmatrix} g\left(z_1^{(l)}\right) \\ \vdots \\ g\left(z_{d_l}^{(l)}\right) \end{bmatrix}$$

$$B^{(l)} = \begin{bmatrix} b_1^{(l)} \\ \vdots \\ b_{d_l}^{(l)} \end{bmatrix}$$

Calculation of the Network Output aka the Forward Path

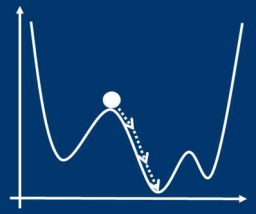
Note: While already looking complicated, our notation illustrates the calculation of the network output for a single data point

$$Z^{(1)} = B^{(1)} + \boldsymbol{W}^{(1)} H^{(0)} \qquad H^{(1)} = g\left(Z^{(1)}\right) \qquad Z^{(2)} = B^{(2)} + \boldsymbol{W}^{(2)} H^{(1)} \qquad H^{(2)} = g\left(Z^{(2)}\right)$$

$$H^{(0)} \equiv \begin{bmatrix} x_1^{(1)} \\ \vdots \\ x_m^{(1)} \end{bmatrix} \qquad Z^{(1)} = \begin{bmatrix} z_1^{(1)} \\ \vdots \\ z_{d_l}^{(l)} \end{bmatrix} \qquad H^{(1)} = \begin{bmatrix} g\left(z_1^{(1)}\right) \\ \vdots \\ g\left(z_{d_l}^{(1)}\right) \end{bmatrix} \qquad Z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ \vdots \\ z_{d_l}^{(2)} \end{bmatrix} \qquad H^{(2)} = \begin{bmatrix} g\left(z_1^{(2)}\right) \\ \vdots \\ g\left(z_{d_l}^{(2)}\right) \end{bmatrix} \quad \cdots$$

$$\cdots \qquad Z^{(L-1)} = \begin{bmatrix} z_1^{(L-1)} \\ \vdots \\ z_{d_l}^{(L-1)} \end{bmatrix} \qquad H^{(L-1)} = \begin{bmatrix} g\left(z_1^{(L-1)}\right) \\ \vdots \\ g\left(z_{d_{L-1}}^{(L-1)}\right) \end{bmatrix} \qquad Z^{(L)} = \left[z_1^{(L)}\right] \equiv \hat{y}$$

$$H^{(L-1)} = g\left(Z^{(L-1)}\right) \qquad Z^{(L)} = B_{1\times 1}^{(L)} + \boldsymbol{W}_{1\times d_{l-1}}^{(L)} H^{(L-1)}$$

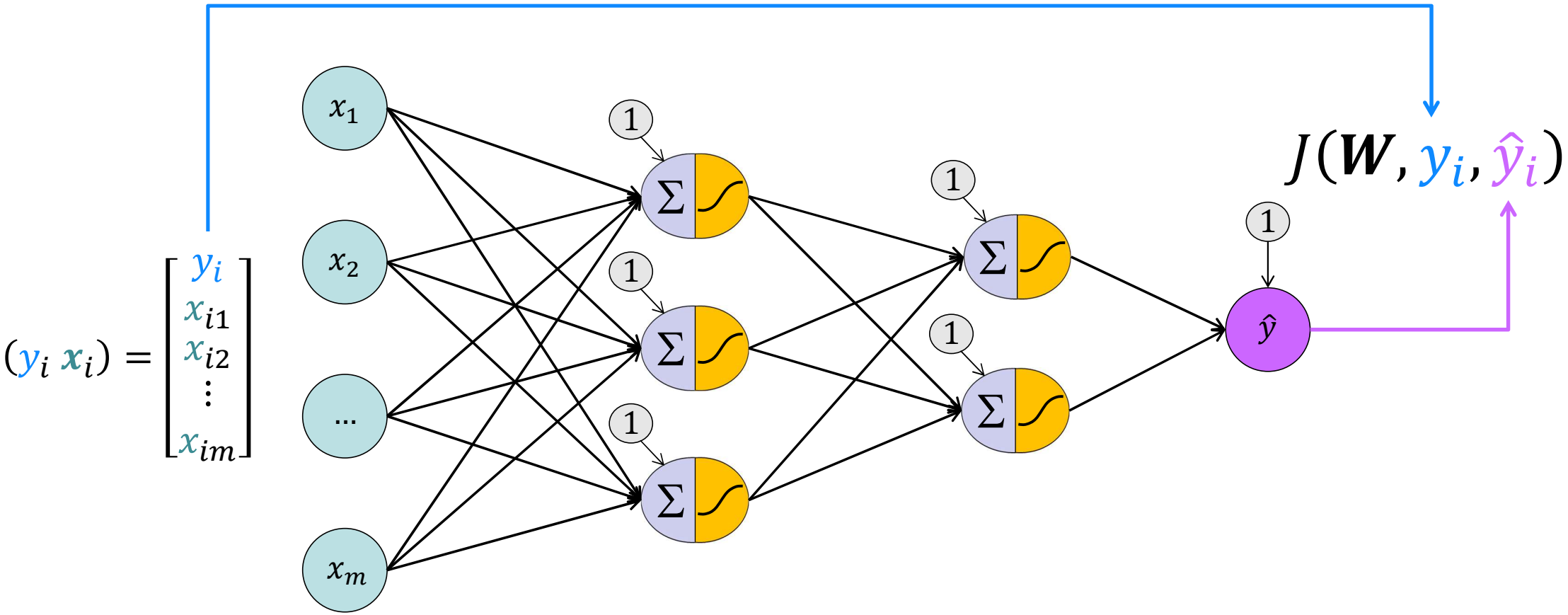


Neural Network Training

Loss function, gradient descent, and backpropagation

Neural Network Training

Loss function measures deviation between network output true target



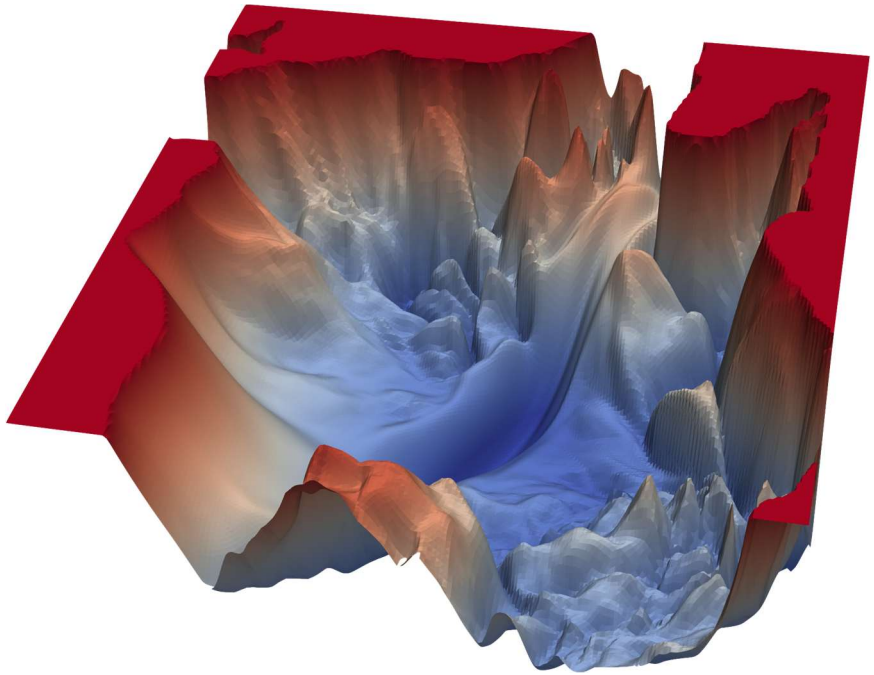
Neural Network Training

- **Minimize loss function over free parameter (connection weights)**

$$\widehat{\boldsymbol{W}} = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W}, y, \hat{y})$$

- **Common loss functions**
  - Least-square loss for regression
  - Cross-entropy for classification (see below)
- **Nonlinear, non-convex problem**
- **No analytical solution**
- **Training based on iterative, numerical algorithms**

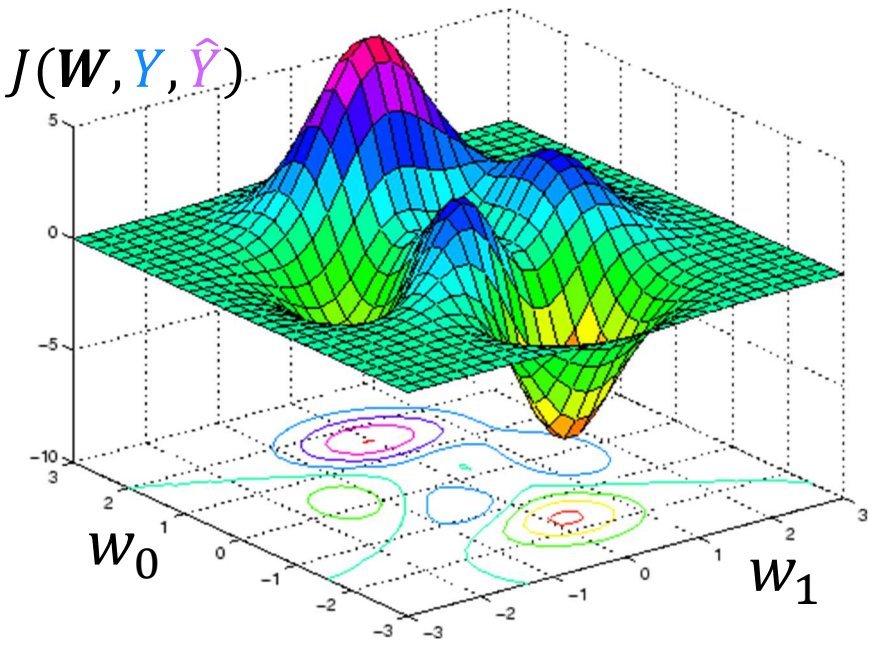
2D Projection of the loss surface in deep NNs for computer vision



Source: https://www.cs.umd.edu/~tomg/projects/landscapes/
3D visualization tool: https://www.telesens.co/loss-landscape-viz/viewer.html
Paper: Hao et al., NeurIPS, 2018.

ANN Training by (Stochastic) Gradient Descent

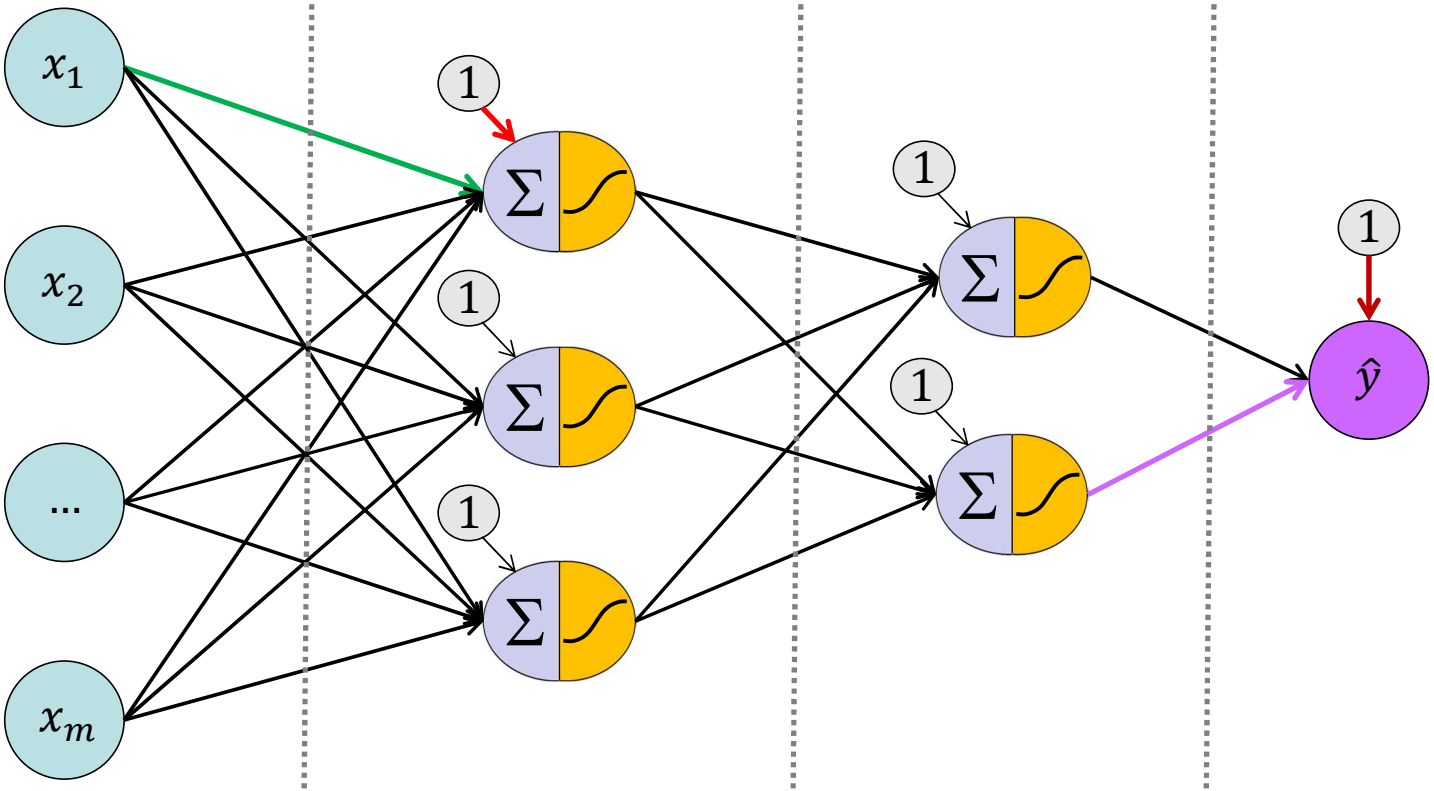
- **Minimization problem** $\widehat{\boldsymbol{W}} = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W}, Y, \hat{Y})$
- **Initialize weights** $\boldsymbol{W}_0 \sim N(0, \sigma^2)$
- **Repeat**
  - Compute network output (forward path) $\hat{Y} = f_{\boldsymbol{W}_t}(\boldsymbol{X})$
  - Compute loss gradient $\frac{\partial J(\boldsymbol{W}_t)}{\partial \boldsymbol{W}_t}$
  - Update weights $\boldsymbol{W}_{t+1} \leftarrow \boldsymbol{W}_t - \eta \frac{\partial J(\boldsymbol{W}_t)}{\partial \boldsymbol{W}_t}$
  - Check stopping criterion
    - Max number of iterations
    - Change of weights



Learning rate determines how much we adjust connection weights per iteration.

The Backpropagation Algorithm

Efficient calculation of the partial derivates of the loss function



Input layer $l = 1$
$d_1 = m$ nodes

Hidden layer $l = j$
$d_j = 3$ nodes

Hidden layer $l = k$
$d_k = 2$ nodes

Output layer L
$d_L = 1$ node

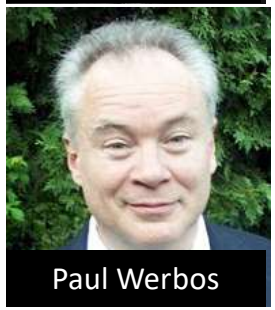
$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial J}{\partial b_1^{(1)}} \\ \frac{\partial J}{\partial w_{11}^{(1)}} \\ \vdots \\ \frac{\partial J}{\partial b_k^{(l)}} \\ \frac{\partial J}{\partial w_{kj}^{(l)}} \\ \vdots \\ \frac{\partial J}{\partial b_{d_L}^{(L)}} \\ \frac{\partial J}{\partial w_{d_L d_{L-1}}^{(L)}} \end{bmatrix}$$

Neural Network History and Backpropagation

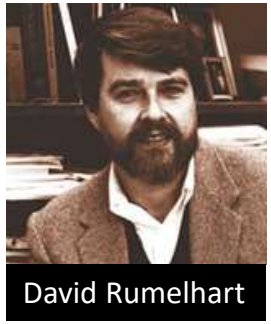
- **Early work starting in 1943**
- **First break through: Rosenblatt's perceptron (1958)**
  - Neural network with one neuron in the hidden layer
  - So actually logistic regression (yet with an online learning rule)
- **Second break through: Backpropagation**
  - Training neural networks with more than one hidden neuron
  - Werbos (1974), Rumelhart et al. (1986)
- **Decline**
  - Lost popularity in late nineties
  - Advent of support vector machines
- **Renaissance**
  - Several advancements since the early 2000 and after
  - Deep networks for computer vision, NLP, etc.


Frank Rosenblatt


Yoshua Bengio


Paul Werbos

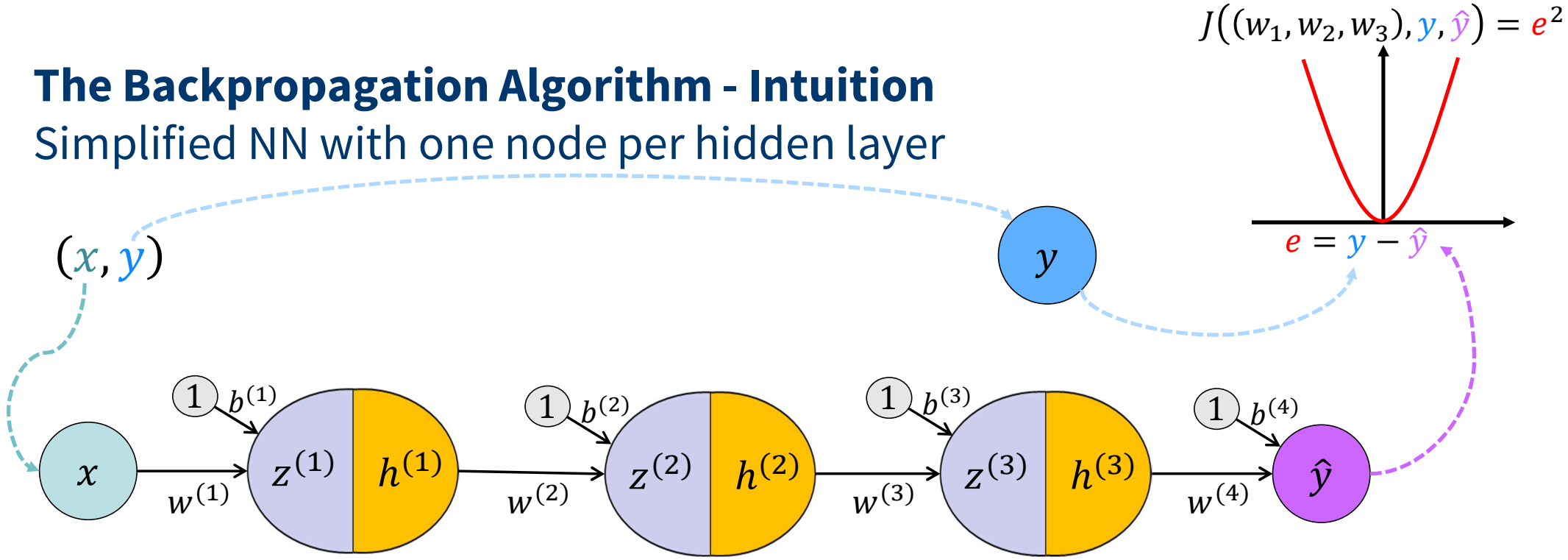

Geoffrey Hinton


David Rumelhart


Yann LeCun

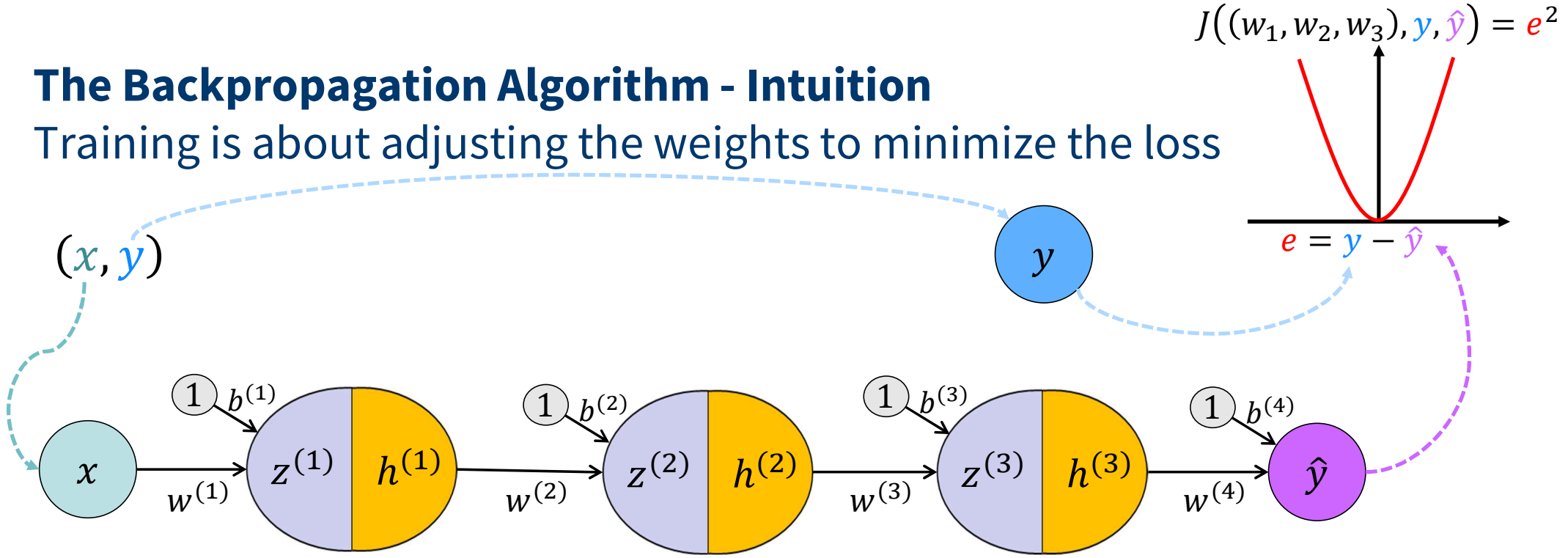
The Backpropagation Algorithm - Intuition

Simplified NN with one node per hidden layer



The Backpropagation Algorithm - Intuition

Training is about adjusting the weights to minimize the loss



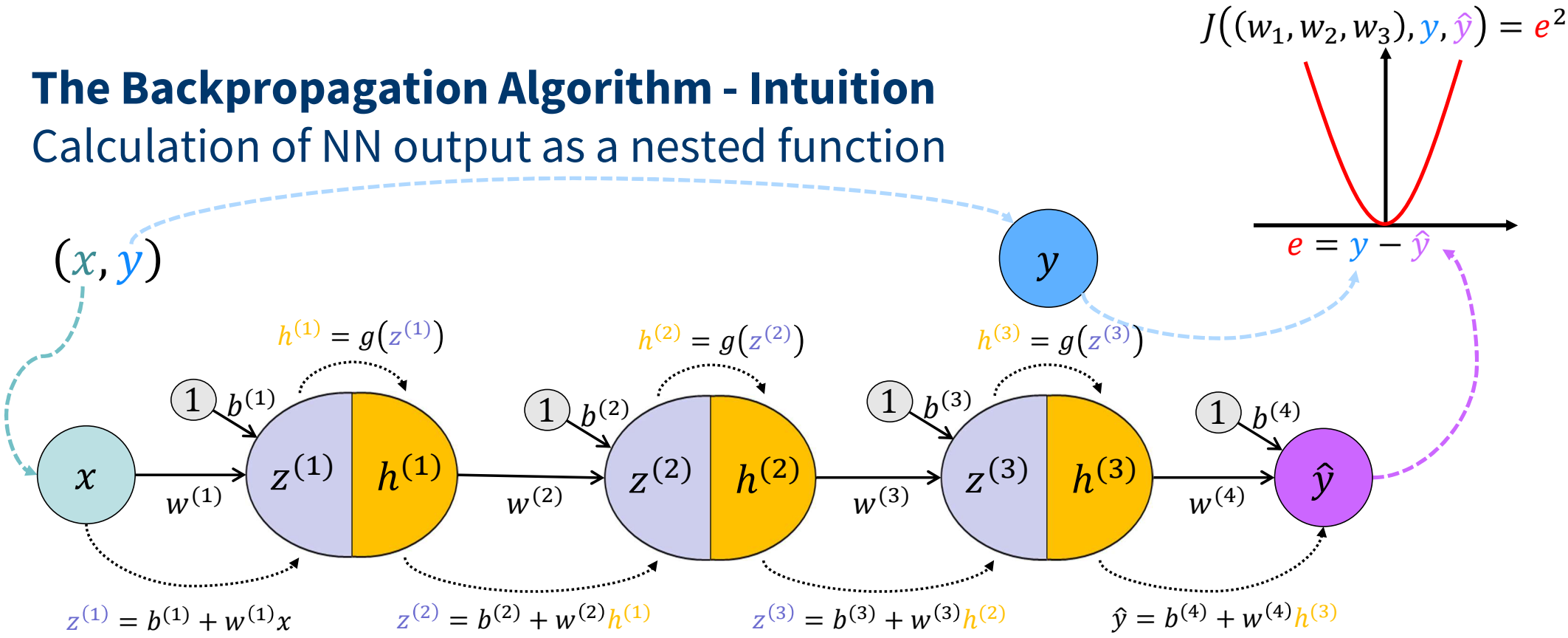
- **Loss J varies with residual e**
- **Residual e varies with NN output $\hat{y}$**
- **NN output $\hat{y}$ varies with**
  - Network weight $w^{(4)}$
  - Output of hidden unit $h^{(3)}$

- **How to adjust $w^{(4)}$ to reduce J?**
- **Examine how J changes locally with $w^{(4)}$**
  - Formally: partial derivative of J with respect to $w^{(4)}$: $\frac{\partial J}{\partial w^{(4)}}$
  - To compute this derivative we need:

The chain rule

The Backpropagation Algorithm - Intuition

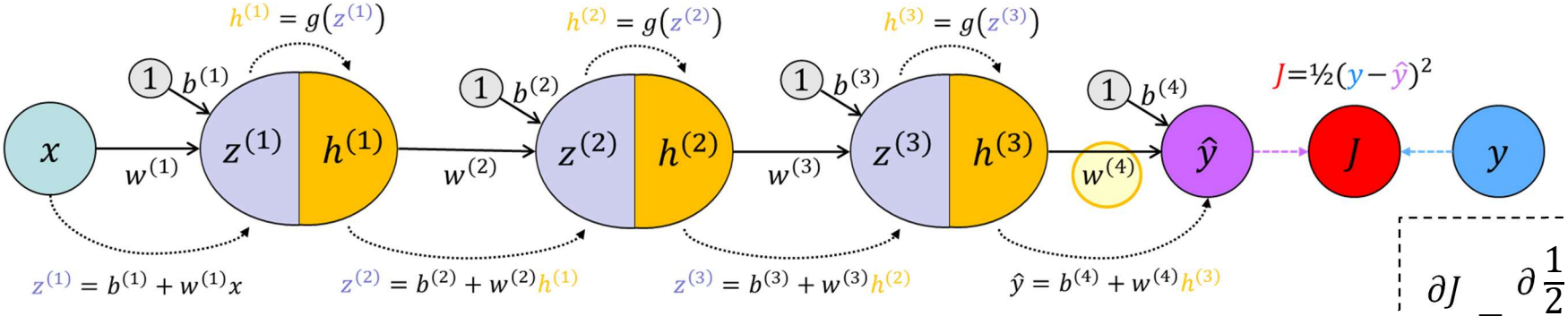
Calculation of NN output as a nested function



The Backpropagation Algorithm

Calculation of the partial derivatives in layer 4

$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \vdots \\ \frac{\partial J}{\partial w^{(4)}} \\ \vdots \end{bmatrix}$$



$$\frac{\partial J}{\partial w^{(4)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w^{(4)}} = (\hat{y} - y)h^{(3)} = \delta^{(4)}h^{(3)}$$

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial \frac{1}{2}(y - \hat{y})^2}{\partial \hat{y}} = -(y - \hat{y}) = (\hat{y} - y)$$

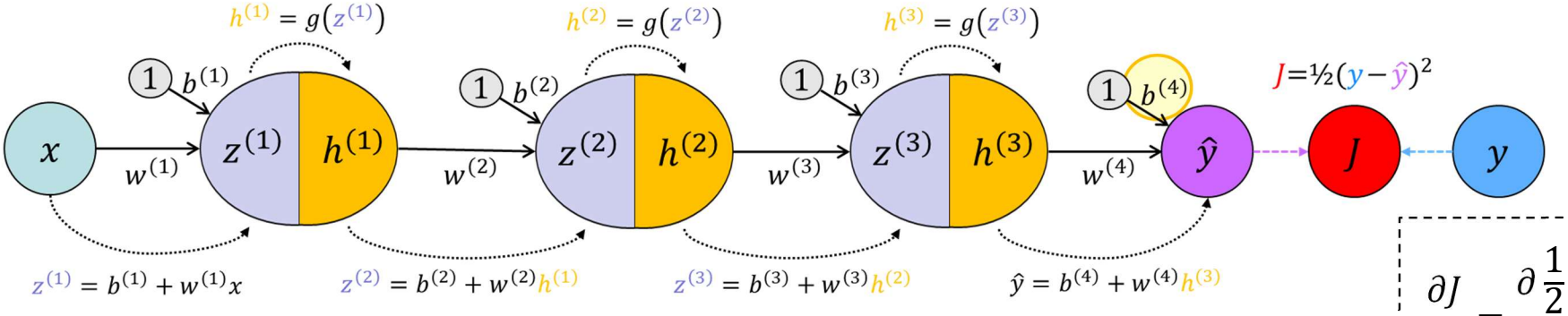
$$\frac{\partial \hat{y}}{\partial w^{(4)}} = \frac{\partial \left(b^{(4)} + w^{(4)}h^{(3)}\right)}{\partial w^{(4)}} = h^{(3)}$$

$$\delta^{(4)} \equiv (\hat{y} - y)$$

The Backpropagation Algorithm

Calculation of the partial derivatives in layer 4

$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \vdots \\ \frac{\partial J}{\partial b^{(4)}} \\ \vdots \end{bmatrix}$$



$$\frac{\partial J}{\partial w^{(4)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w^{(4)}} = (\hat{y} - y)h^{(3)} = \delta^{(4)}h^{(3)}$$

$$\frac{\partial J}{\partial b^{(4)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial b^{(4)}} = (\hat{y} - y)1 = \delta^{(4)}$$

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial \frac{1}{2}(y - \hat{y})^2}{\partial \hat{y}} = -(y - \hat{y}) = (\hat{y} - y)$$

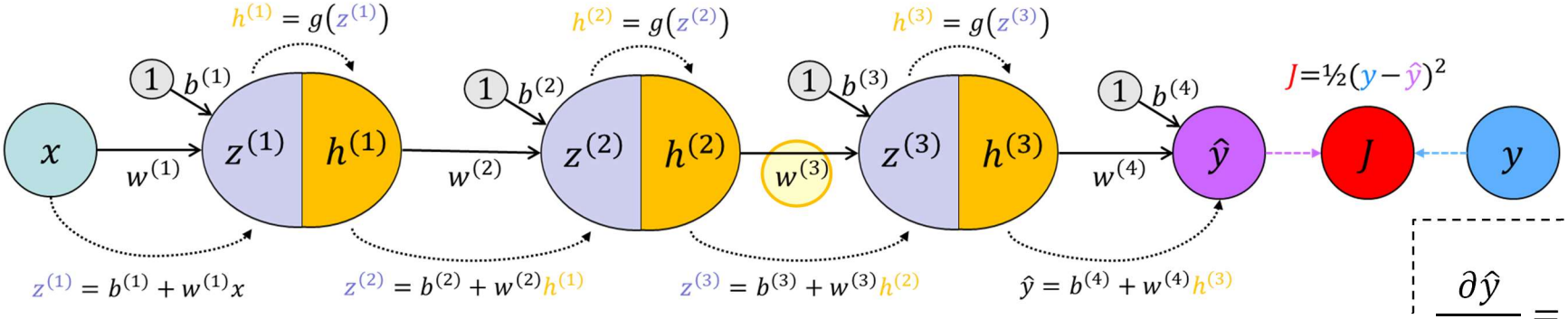
$$\frac{\partial \hat{y}}{\partial b^{(4)}} = \frac{\partial \left(b^{(4)} + w^{(4)}h^{(3)}\right)}{\partial b^{(4)}} = 1$$

$$\delta^{(4)} \equiv (\hat{y} - y)$$

The Backpropagation Algorithm

Calculation of the partial derivatives in layer 3

$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \vdots \\ \dfrac{\partial J}{\partial w^{(3)}} \\ \vdots \end{bmatrix}$$



$$\frac{\partial J}{\partial w^{(4)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w^{(4)}} = (\hat{y} - y)h^{(3)} = \delta^{(4)}h^{(3)}$$

$$\frac{\partial J}{\partial w^{(3)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h^{(3)}} \times \frac{\partial h^{(3)}}{\partial z^{(3)}} \times \frac{\partial z^{(3)}}{\partial w^{(3)}} = (\hat{y} - y)w^{(4)}g'\left(z^{(3)}\right)h^{(2)} = \delta^{(3)}h^{(2)}$$

$$\frac{\partial \hat{y}}{\partial h^{(3)}} = \frac{\partial\left(b^{(4)} + w^{(4)}h^{(3)}\right)}{\partial h^{(3)}} = w^{(4)}$$

$$\frac{\partial h^{(3)}}{\partial z^{(3)}} = \frac{\partial\left(g\left(z^{(3)}\right)\right)}{\partial z^{(3)}} = g'\left(z^{(3)}\right)$$

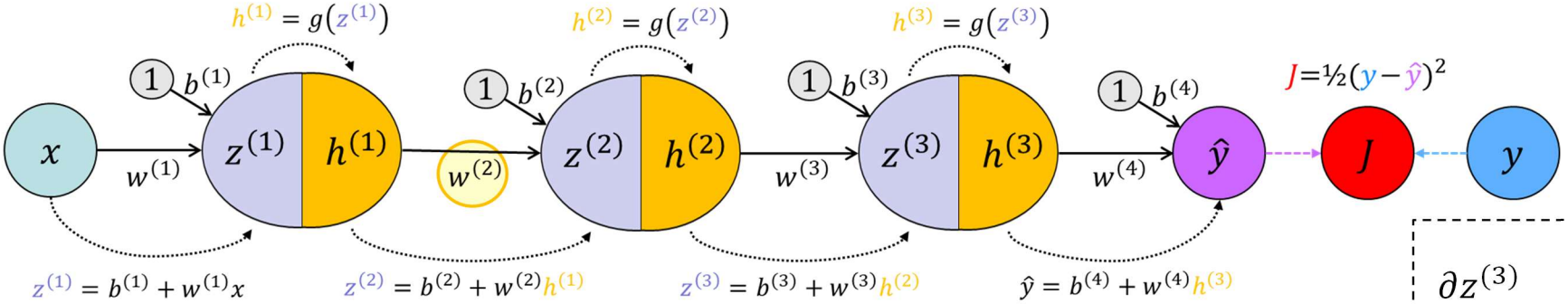
$$\frac{\partial z^{(3)}}{\partial w^{(3)}} = \frac{\partial\left(b^{(4)} + w^{(3)}h^{(2)}\right)}{\partial w^{(3)}} = h^{(2)}$$

$$\delta^{(3)} \equiv \delta^{(4)}w^{(4)}g'\left(z^{(3)}\right)$$

The Backpropagation Algorithm

Calculation of the partial derivatives in layer 2

$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \vdots \\ \frac{\partial J}{\partial w^{(2)}} \\ \vdots \end{bmatrix}$$



$$\frac{\partial J}{\partial w^{(4)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w^{(4)}} = \delta^{(4)} h^{(3)}$$

$$\frac{\partial J}{\partial w^{(3)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h^{(3)}} \times \frac{\partial h^{(3)}}{\partial z^{(3)}} \times \frac{\partial z^{(3)}}{\partial w^{(3)}} = \delta^{(3)} h^{(2)}$$

$$\frac{\partial J}{\partial w^{(2)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h^{(3)}} \times \frac{\partial h^{(3)}}{\partial z^{(3)}} \times \frac{\partial z^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial w^{(2)}} = \cdots = \delta^{(2)} h^{(1)}$$

$$\frac{\partial z^{(3)}}{\partial h^{(2)}} = \frac{\partial \left(b^{(3)} + w^{(3)} h^{(2)}\right)}{\partial z^{(3)}} = w^{(3)}$$

$$\frac{\partial h^{(2)}}{\partial z^{(2)}} = \frac{\partial \left(g\left(z^{(2)}\right)\right)}{\partial z^{(2)}} = g'\left(z^{(2)}\right)$$

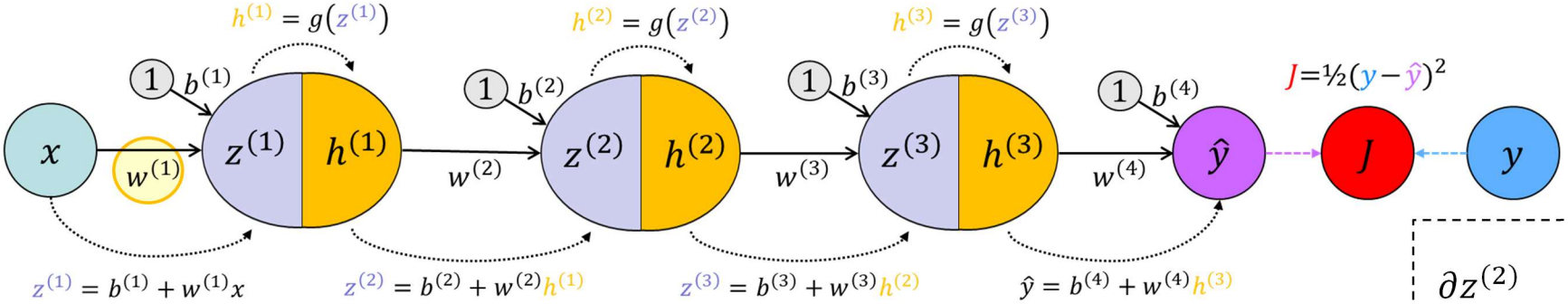
$$\frac{\partial z^{(2)}}{\partial w^{(2)}} = \frac{\partial \left(b^{(2)} + w^{(2)} h^{(1)}\right)}{\partial w^{(2)}} = h^{(1)}$$

$$\delta^{(2)} \equiv \delta^{(3)} w^{(3)} g'\left(z^{(2)}\right)$$

The Backpropagation Algorithm

Calculation of the partial derivatives in layer 1

$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \vdots \\ \frac{\partial J}{\partial w^{(1)}} \\ \vdots \end{bmatrix}$$



$$\frac{\partial J}{\partial w^{(4)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w^{(4)}} = \delta^{(4)} h^{(3)}$$

$$\frac{\partial J}{\partial w^{(3)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h^{(3)}} \times \frac{\partial h^{(3)}}{\partial z^{(3)}} \times \frac{\partial z^{(3)}}{\partial w^{(3)}} = \delta^{(3)} h^{(2)}$$

$$\frac{\partial J}{\partial w^{(2)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h^{(3)}} \times \frac{\partial h^{(3)}}{\partial z^{(3)}} \times \frac{\partial z^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial w^{(2)}} = \cdots = \delta^{(2)} h^{(1)}$$

$$\frac{\partial J}{\partial w^{(1)}} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h^{(3)}} \times \frac{\partial h^{(3)}}{\partial z^{(3)}} \times \frac{\partial z^{(3)}}{\partial h^{(2)}} \times \frac{\partial h^{(2)}}{\partial z^{(2)}} \times \frac{\partial z^{(2)}}{\partial h^{(1)}} \times \frac{\partial h^{(1)}}{\partial z^{(1)}} \times \frac{\partial z^{(1)}}{\partial w^{(1)}} = \delta^{(1)} x$$

$$\frac{\partial z^{(2)}}{\partial h^{(1)}} = \frac{\partial \left(b^{(2)} + w^{(2)} h^{(1)}\right)}{\partial z^{(2)}} = w^{(2)}$$

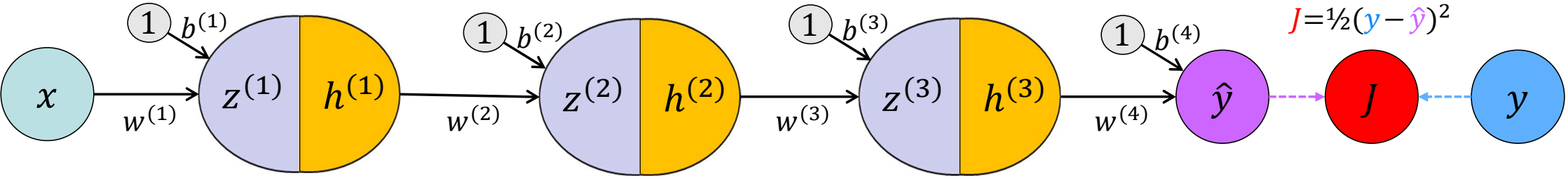
$$\frac{\partial h^{(1)}}{\partial z^{(1)}} = \frac{\partial \left(g\left(z^{(1)}\right)\right)}{\partial z^{(1)}} = g'\left(z^{(1)}\right)$$

$$\frac{\partial z^{(1)}}{\partial w^{(1)}} = \frac{\partial \left(b^{(1)} + w^{(1)} x\right)}{\partial w^{(1)}} = x$$

$$\delta^{(1)} \equiv \delta^{(2)} w^{(2)} g'\left(z^{(1)}\right)$$

The Backpropagation Algorithm – Delta Rule

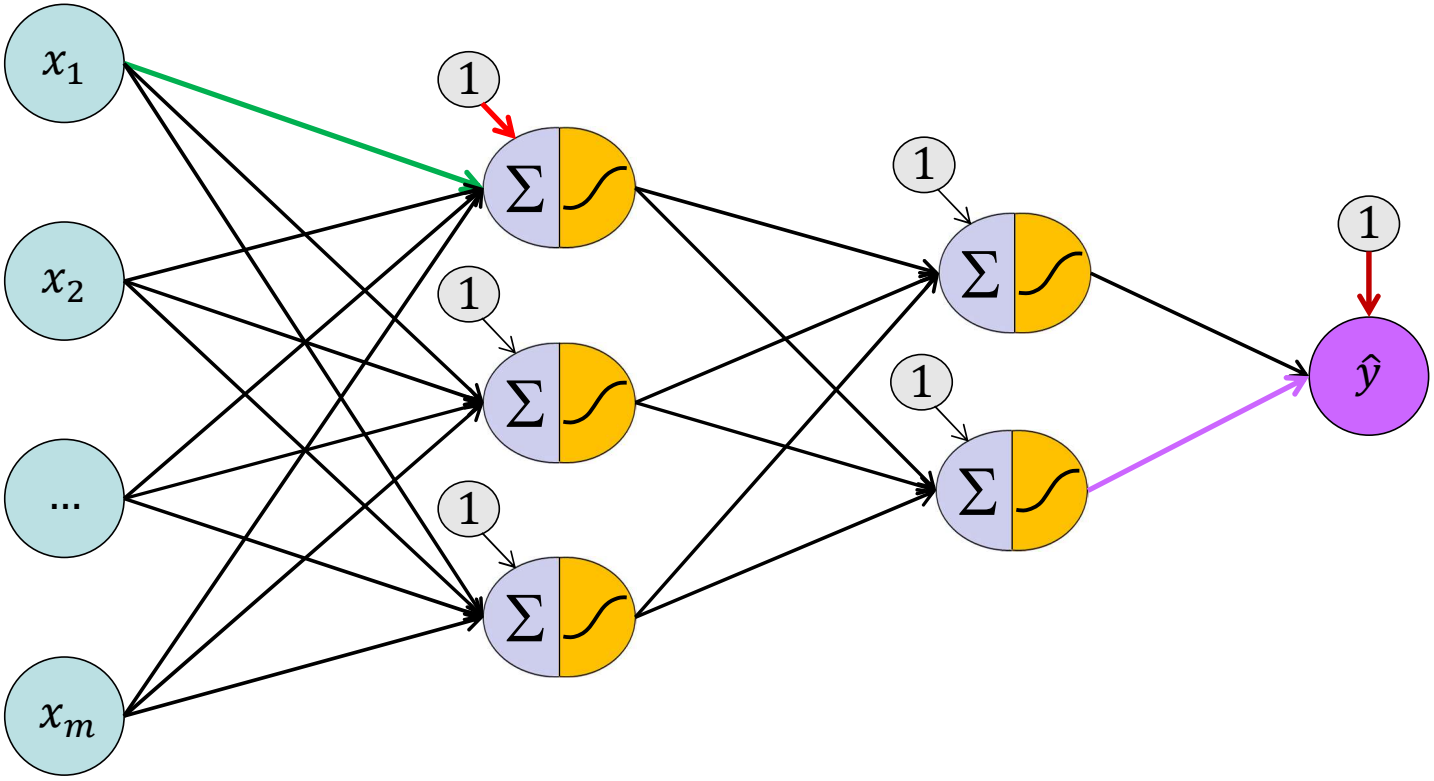
Calculation of the partial derivatives in anylayer l



	Input layer $l = 1$	**Hidden layers** $1 < l < L$	**Output layer** $l = L$
Calculation of δ	$\delta^{(l)} \equiv \delta^{(l+1)} w^{(l+1)} g'\left(z^{(l)}\right)$		$\delta^{(L)} = \frac{\partial J}{\partial \hat{y}}$
Partial derivative of the loss with respect to w	$\frac{\partial J}{\partial w^{(l)}} = \delta^{(l)} \times h^{(l-1)}$		
Partial derivative of the loss with respect to b	$\frac{\partial J}{\partial b^{(l)}} = \delta^{(l)}$		

The Backpropagation Algorithm – Delta Rule

Computing the gradient by backpropagating the error signal



Input layer
$l = 1$
of $d_1 = m$ nodes

Hidden layer
$l = j$
of $d_j = 3$ nodes

Hidden layer
$l = k$
of $d_k = 2$ nodes

Output layer
L
of $d_L = 1$ nodes

$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial J}{\partial b_1^{(1)}} = \delta^{(1)} \\ \frac{\partial J}{\partial w_{11}^{(1)}} = \delta^{(1)} x \\ \vdots \\ \frac{\partial J}{\partial b_k^{(l)}} = \delta^{(l)} \\ \frac{\partial J}{\partial w_{kj}^{(l)}} = \delta^{(l)} h^{(l-1)} \\ \vdots \\ \frac{\partial J}{\partial b_{d_L}^{(L)}} = \delta^{(L)} \\ \frac{\partial J}{\partial w_{d_L d_{L-1}}^{(L)}} = \delta^{(L)} h^{(L-1)} \end{bmatrix}$$

NN Training by Backpropagation & Gradient Descent

Interim Summary

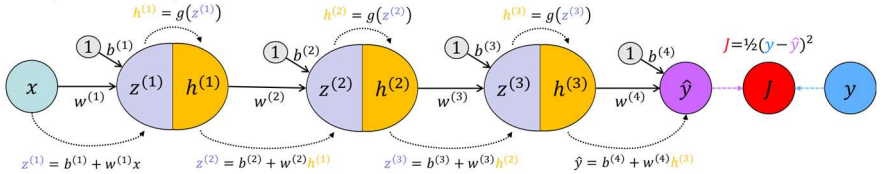
- **Initialize weights randomly*, e.g., $W \sim N(0, \sigma^2)$**
- **Loop until convergence**

1. Compute forward path and loss
2. Compute loss and loss gradient
3. Update network weights
4. Check stopping criterion

- **Return weights**

*There are better ways to initialize the weights (see Glorot & Bengio 2010)

1



2

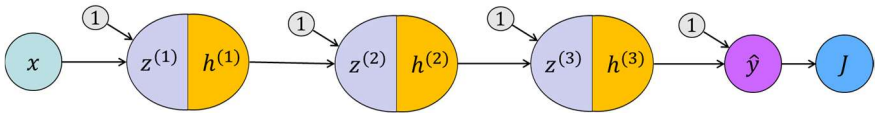
	Input $l = 1$	Hidden $1 < l < L$	Output $l = L$
Calculation of δ	$\delta^{(l)} \equiv \delta^{(l+1)} w^{(l+1)} g'(z^{(1)})$		$\delta^{(L)} = \frac{\partial J}{\partial \hat{y}} \times \cdots$
Partiel derivative of w	$\frac{\partial J}{\partial w^{(l)}} = \delta^{(l)} \times h^{(l-1)}$		
Partial derivative of b	$\frac{\partial J}{\partial b^{(l)}} = \delta^{(l)}$		

3

$$W_{t+1} \leftarrow \begin{bmatrix} w_{11}^{(1)} \\ \vdots \\ w_{kj}^{(l)} \\ \vdots \\ w_{d_L d_{L-1}}^{(L)} \end{bmatrix} - \eta \begin{bmatrix} \delta^{(1)} x \\ \vdots \\ \delta^{(l)} h^{(l-1)} \\ \vdots \\ \delta^{(L)} h^{(L-1)} \end{bmatrix}$$

Extension: Hidden Layers Do Not Have One But Many Nodes

- **We started from here**

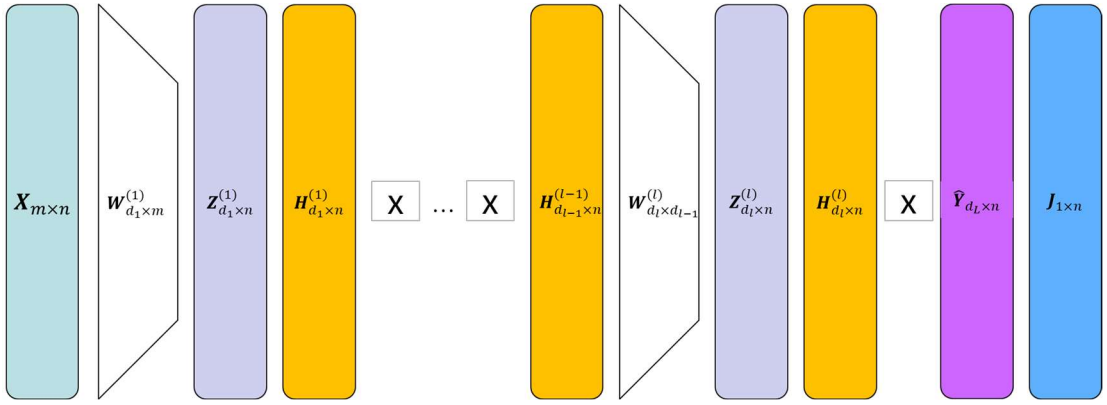


- **And found this**

$$\begin{cases} \dfrac{\partial J}{\partial w^{(l)}} = \delta^{(l)} \times h^{(l-1)} \\ \dfrac{\partial J}{\partial b^{(l)}} = \delta^{(l)} \end{cases}$$

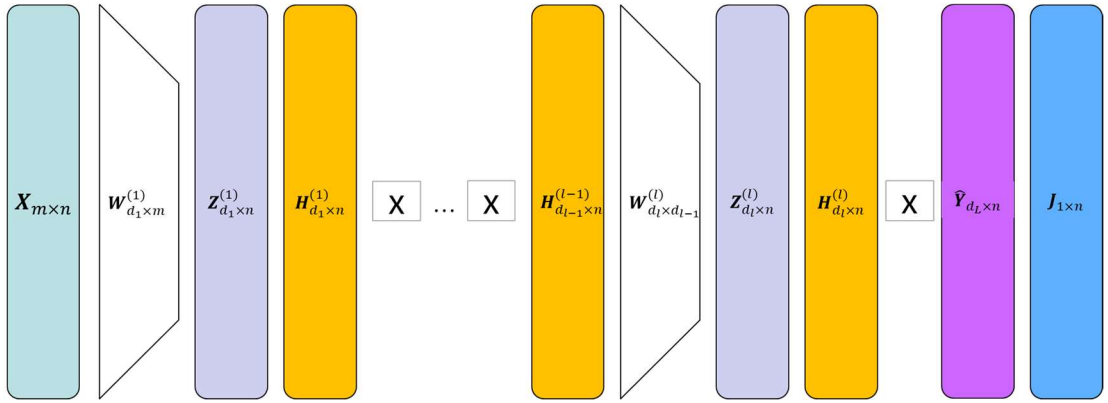
with $\delta^{(l)} \equiv \delta^{(l+1)} w^{(l+1)} g'\left(z^{(1)}\right)$

- **In practice, we have this**



Extension: Hidden Layers Do Not Have One But Many Nodes

How to generalize NN training ?



- **Same principle as before**
  - Compute forward path and loss
  - Compute gradient
  - Update weights
- **Need to adjust calculation of δ**
  - Our $\delta^{(l)}$ from above were scalars
  - Now we work with vectors and matrices

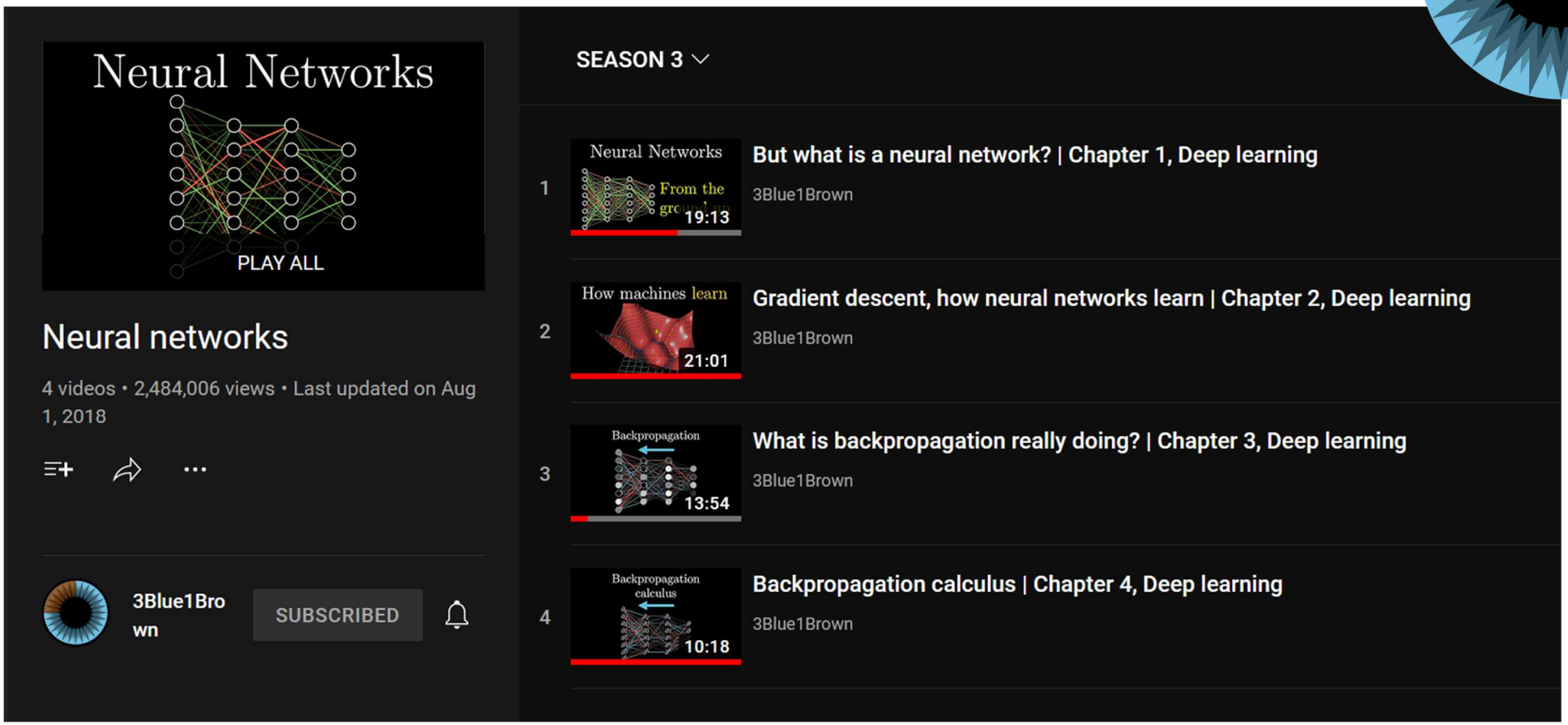
$$\boldsymbol{\delta}^{(l)} \equiv \left(\left(\boldsymbol{W}^{(l+1)}\right)^{\mathsf{T}}\boldsymbol{\delta}^{(l+1)}\right)\odot g'\left(\boldsymbol{Z}^{(l)}\right)$$

- **With the generalized $\delta^{(l)}$, we can compute the gradient just as before**
- **And update the weights just as before**
- **Calculations get somewhat tedious (see Part 3 of Salloum (2019)) if interested**

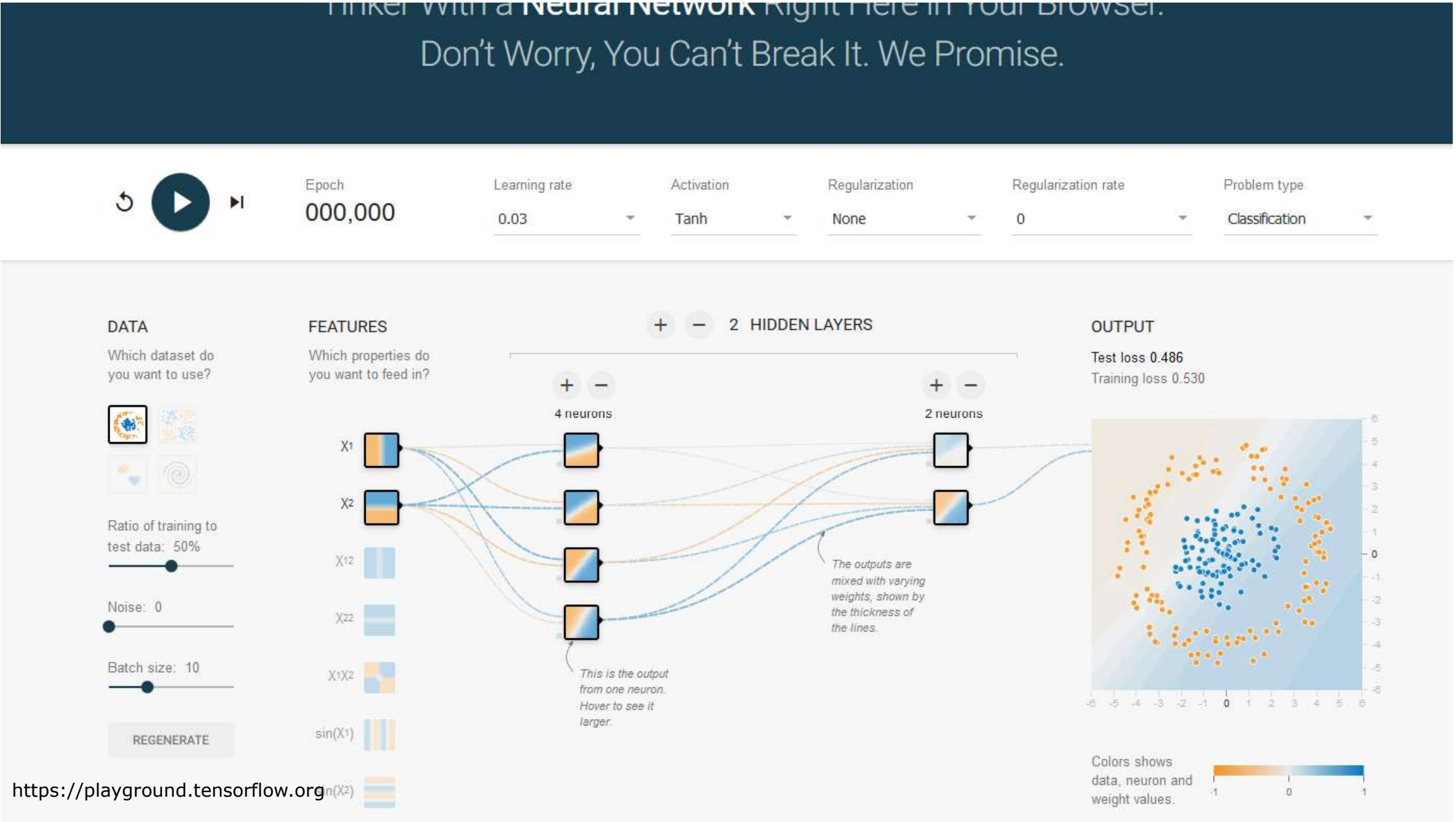
Salloum, Z. (2019) Back Propagation, the Easy Way. 3 Part Blog Series. https://towardsdatascience.com/back-propagation-the-easy-way-part-1-6a8cde653f65

Further Resources on Backpropagation

Neural network series on 3Blue1Brown



Source: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi



https://playground.tensorflow.org

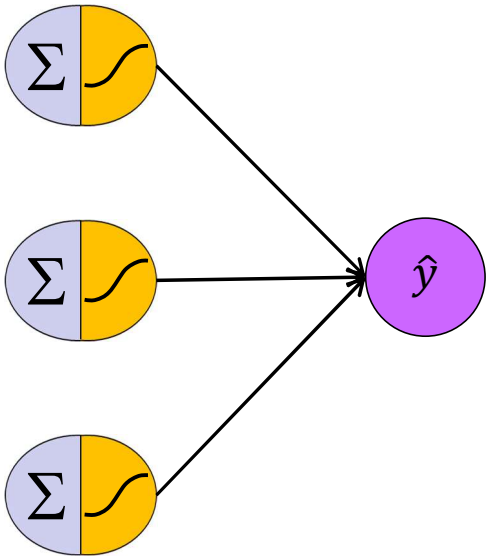
Applications of Artificial Neural Networks

Interplay between modeling task & network architecture

Applications of Neural Networks

What we achieved thus far

- **Output layer with a single node facilitates modeling of one target variable**
- **Squared error for network training**
- **Examples and use cases**
  - Sales forecasting
    - Y: Product sales in period t
    - $\boldsymbol{X}$: Price, Last period‘s sales, competitor price, …
  - Customer lifetime value (CLV)
    - Y: Discounted cash flow of customer-level profit
    - $\boldsymbol{X}$: (Socio-)demographic, transactional, relational, … data
  - Real estate pricing
    - Y: Offer or sales prices of real estate
    - $\boldsymbol{X}$: Apartment size, no. rooms, floor, equipment, location data, …



Applications of Neural Networks

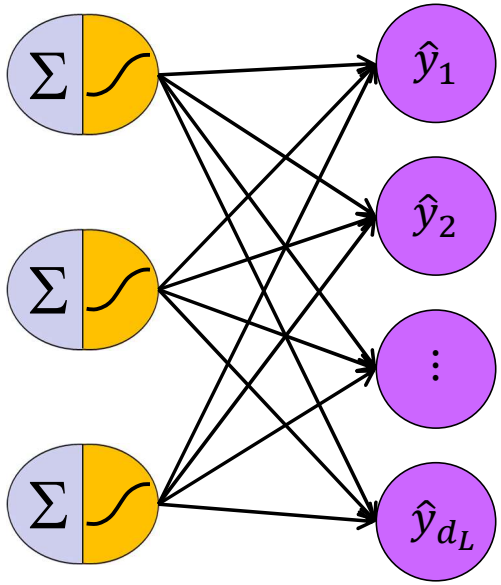
Extensions to other types of (prediction) problems

- **Neural networks provide much flexibility**
  - All forms of supervised learning problems are supported
  - Simple regression, multi-output-regression, binary/multi-class classification, survival analysis, …
- **Adjustment of network architecture to adapt network for given use case**
  - Structure of the input data (e.g., time series forecasting)
  - Configuration of the output layer (e.g., multi-output regression)
  - Loss function for network training (e.g., classification)
- **Hyperparameter tuning also required**
  - Number and dimension of hidden layers
  - Activation function
  - Learning rate
  - …

Applications of Neural Networks

Network architecture for multi-output regression

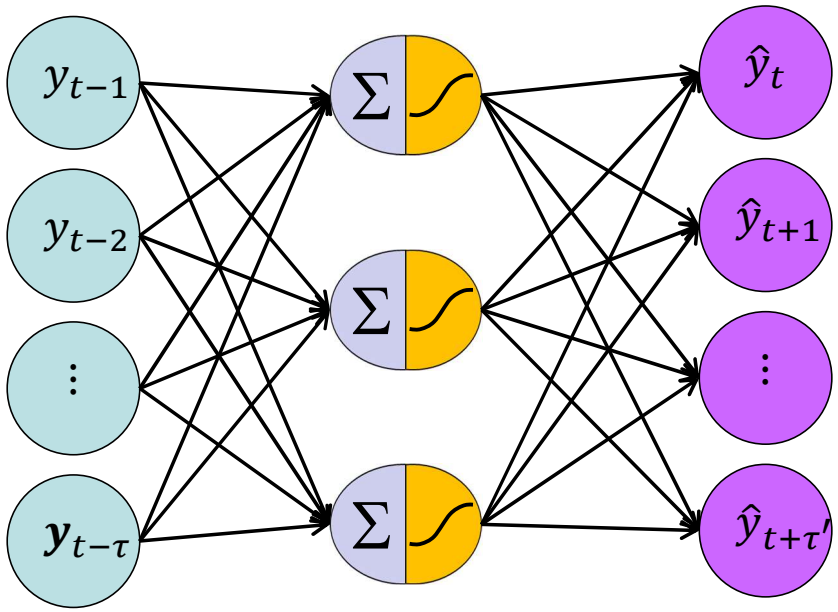
- **Output layer with $d_L > 1$ nodes, each representing the forecast for one target**
- **Train using aggregated (across targets) squared error loss**
- **Exemplary use case: financial market modeling**
  - $\boldsymbol{Y}$: Closing prices of all stocks in the S&P500 on day t
  - $\boldsymbol{X}$: Measurements of S&P500 stocks
    - Past (lagged) prices from past day $t-1, t-2, \ldots, t-\tau$
    - Trading volume from past days
    - Technical indicators (MACD, RSI, Bollinger bands, …)
    - Other stock indices
    - Commodity prices and other macro-economic data



Applications of Neural Networks

Network architecture for time series forecasting

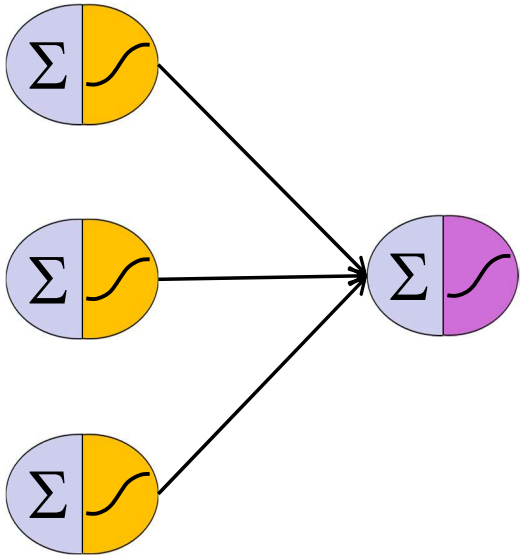
- **One target variable with past realizations**
- **Output layer with $d_L \geq 1$ nodes**
  - Each node gives one forecast horizon (multi-step ahead forecast)
  - $\hat{y}_t, \hat{y}_{t+1}, \dots, \hat{y}_{t+\tau'}$
- **Train using squared error loss**
  - Compute mean error across horizons
  - Perhaps include recency-based weighting
- **Input data**
  - Lagged realizations of the target variable
  - $y_{t-1}, y_{t-2}, \dots, y_{t-\tau}$



Applications of Neural Networks

Network architecture for binary classification

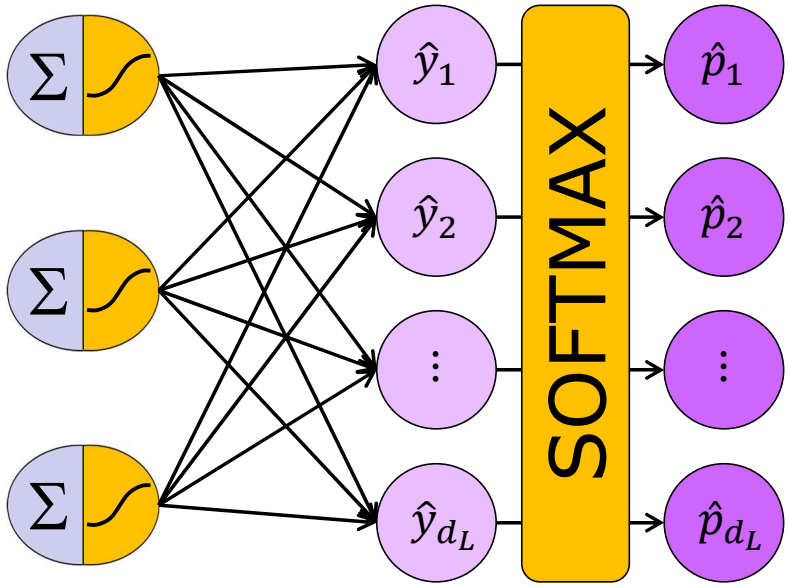
- **Output layer with one node**
  - Estimation of the entry probability of one of the two possible events (i.e., classes)
  - Add sigmoid activation to output layer to guarantee that $\hat{y} \in [0,1]$
- **Train network using log-loss function**
- **Examples**
  - Customer scoring (e.g., loan approval)
    - Y: Did client pay back the loan?
    - $\boldsymbol{X}$: Debt-to-income ratio, age, savings, employment type, …
  - Business failure prediction
    - Y: Did company file for insolvency?
    - $\boldsymbol{X}$: Balance sheet and other accounting data
  - Fraud detection (e.g., card transaction, tax statement, account login, …)
    - Y: Is transaction legitimate?
    - $\boldsymbol{X}$: Payment, purchased item, merchant,, …



Applications of Neural Networks

Network architecture for multi-class classification

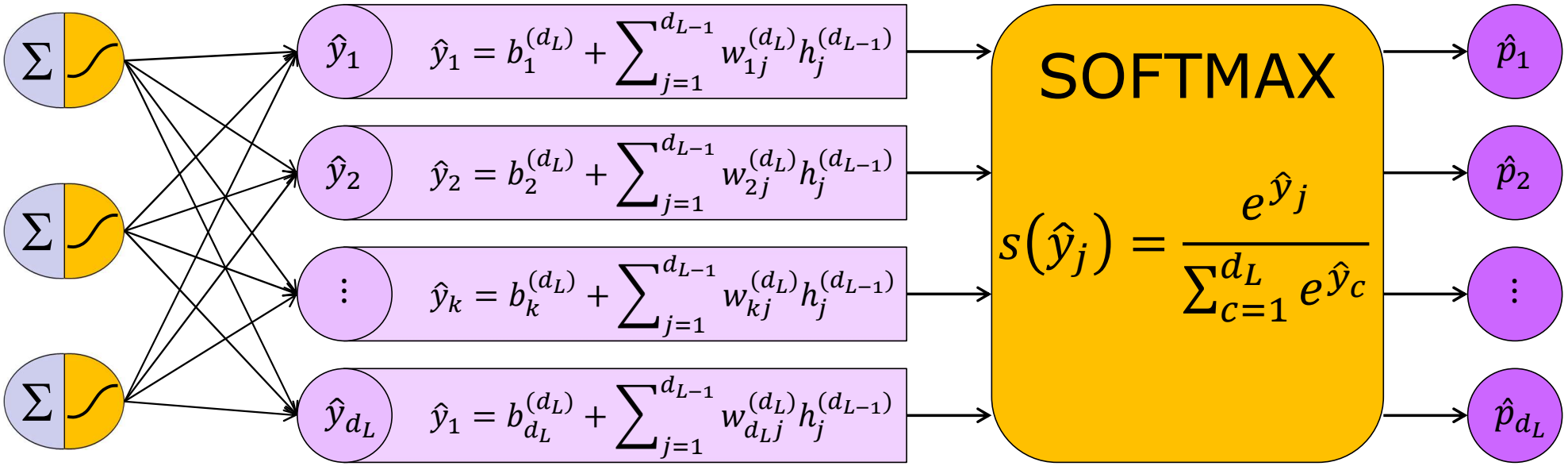
- **Application setting and NN architecture**
  - Multi-class classification problem with $d_L > 1$ mutually exclusive classes or states
  - Output layer with $d_L > 1$ nodes modeling the entry probability for one of the classes/states
- **Example: credit rating analysis**
  - Y: company rating in period t (e.g., AAA, AA, …D)
  - $\boldsymbol{X}$: balance sheet data, macro-economic factors, etc.
- **Network architecture and training**
  - Target classes are mutually exclusive, thus $\sum_{o=1}^{d_L} \hat{y}_o = 1$
  - Softmax layer normalizes the NN output
    - Like an activation function applied to output layer
    - Just that an activation function processes the input of one node whereas softmax processes all inputs to the output layer



The Softmax Function

Normalizing NN outputs for multi-class classification

$$\sum_{c=1}^{d_L} \hat{y}_c = 1$$



NN with d_L output nodes

Raw NN output scores (logits)

Normalized NN output probabilities

Cross-Entropy Loss Function for Multi-Class Classification Problems

- **Multi-class problem with $d_L > 1$ mutually exclusive classes or states**
- **Let $c = 1,2, \ldots, d_L$ index individual classes**
- **Cross entropy loss**

$$-\sum_{c=1}^{d_L} y_{ic} \log\left(f_{\boldsymbol{W}}(\boldsymbol{X}_i)\right)$$

$$\boldsymbol{Y} = \{y_{ic}\}_{\times d_L} = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1d_L} \\ y_{21} & y_{22} & \cdots & y_{2d_L} \\ \vdots & \vdots & \cdots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nd_L} \end{pmatrix}$$

- **Idea**
  - Calculate separate loss for each class
  - Sum over results (i.e., all classes)
- **One-hot encoding of class labels as (n, d_L) array**
- **Useful for vectorization**

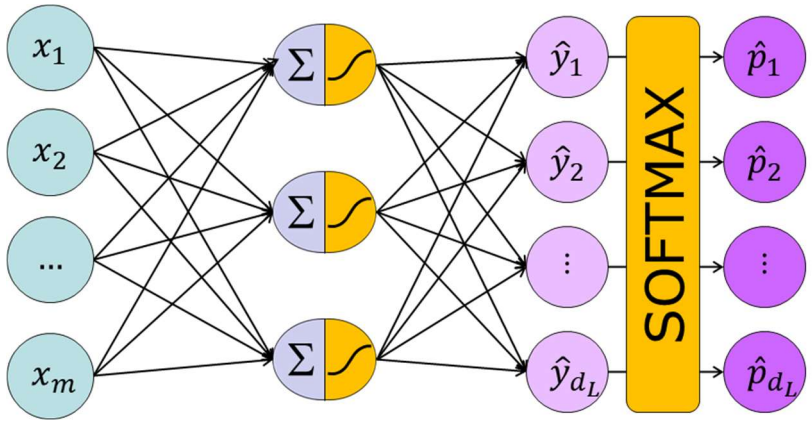
where $y_{ic} \in \{0,1\}$

$$\sum_c y_{ic} = 1 \; \forall i = 1, 2, \ldots n$$

Cross-Entropy Loss Function for Multi-Class Classification Problems

$$\boldsymbol{X} = \{x_{ij}\}_{n\times m} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \cdots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{pmatrix}$$

$$\boldsymbol{Y} = \{y_{ic}\}\ _{\times d_L} = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1d_L} \\ y_{21} & y_{22} & \cdots & y_{2d_L} \\ \vdots & \vdots & \cdots & \vdots \\ y_{n1} & y_{n2} & \cdots & y_{nd_L} \end{pmatrix}$$

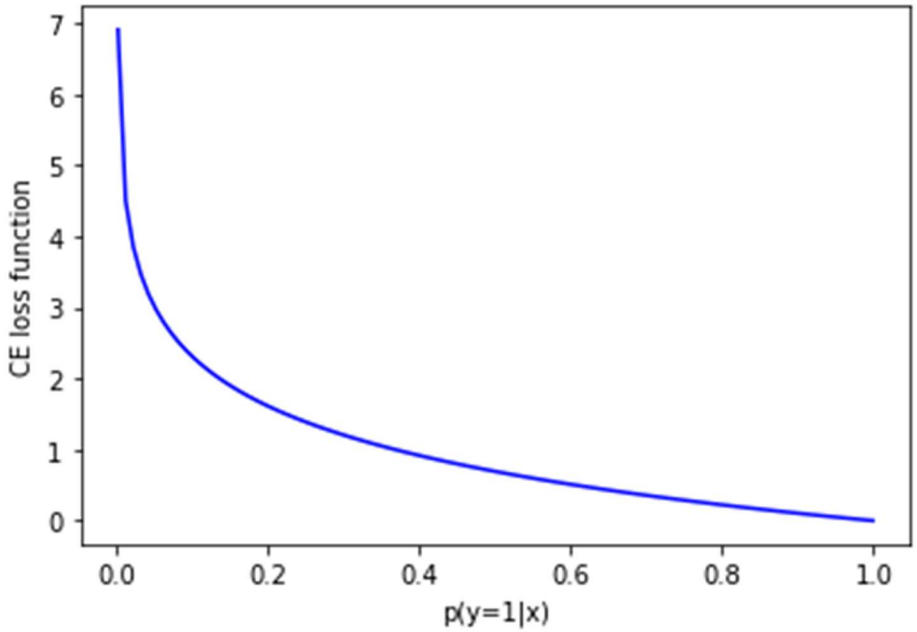


$$\hat{\boldsymbol{Y}} = \{\hat{p}_{ic}\}_{n\times d_L} = \begin{pmatrix} \hat{p}_{11} & \hat{p}_{12} & \cdots & \hat{p}_{1d_L} \\ \hat{p}_{21} & \hat{p}_{22} & \cdots & \hat{p}_{2d_L} \\ \vdots & \vdots & \cdots & \vdots \\ \hat{p}_{n1} & \hat{p}_{n2} & \cdots & \hat{p}_{nd_L} \end{pmatrix}$$

$$J(\boldsymbol{W}) = -\sum_{i=1}^{n}\sum_{c=1}^{d_L} y_{ic}\log\left(f_{\boldsymbol{W}}(\boldsymbol{X}_i)\right) = -\sum_{i=1}^{n} y_{i1}\log(\hat{p}_{11}) + y_{i2}\log(\hat{p}_{12}) + \cdots + y_{id_L}\log\left(\hat{p}_{1d_L}\right)$$

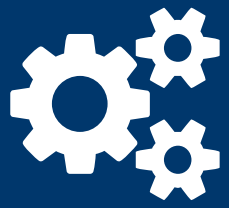
Binary Cross Entropy

- **Special case for two-class problems**
- **Loss Function for binary classification**
- **Equivalent to log-loss (aka negative log-likelihood)**



$$J(\boldsymbol{W}) = -\sum_{i=1}^{n}\sum_{c=1}^{d_L=2} y_{ic}\log\left(f_{\boldsymbol{W}}(\boldsymbol{X}_i)\right)$$

$$= -\sum_{i=1}^{n} y_{i1}\log(\hat{p}_{i1}) + (1-y_{i1})\log(1-\hat{p}_{i1})$$



Regularization and model selection

Penalties, early stopping, dropout, and random search

Early Stopping

- Holdback a subset of the training data for validation
- Trace loss on validation sample during network training
- Terminate training before network weights adjust too much to training data

Regularization

Introduce bias to reduce variance (and find a better balance between the two)

- **Add a term to the loss function that penalizes complexity**
- **Much of the complexities in NNs stems from the layered architecture with many free parameters (i.e., connection weights)**
- **To regularize a NN, penalize the weights**
  - Also called weight decay in NN learning
  - Approximate 'complexity' as the total sum over weights
  - Add meta-parameter to control the trade-off between low training error and low complexity

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - f_{\boldsymbol{W}}(\boldsymbol{X}_i)\right)^2$$

$$J'(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} J(\boldsymbol{W}) + \lambda\|\boldsymbol{W}\|$$

Regularization

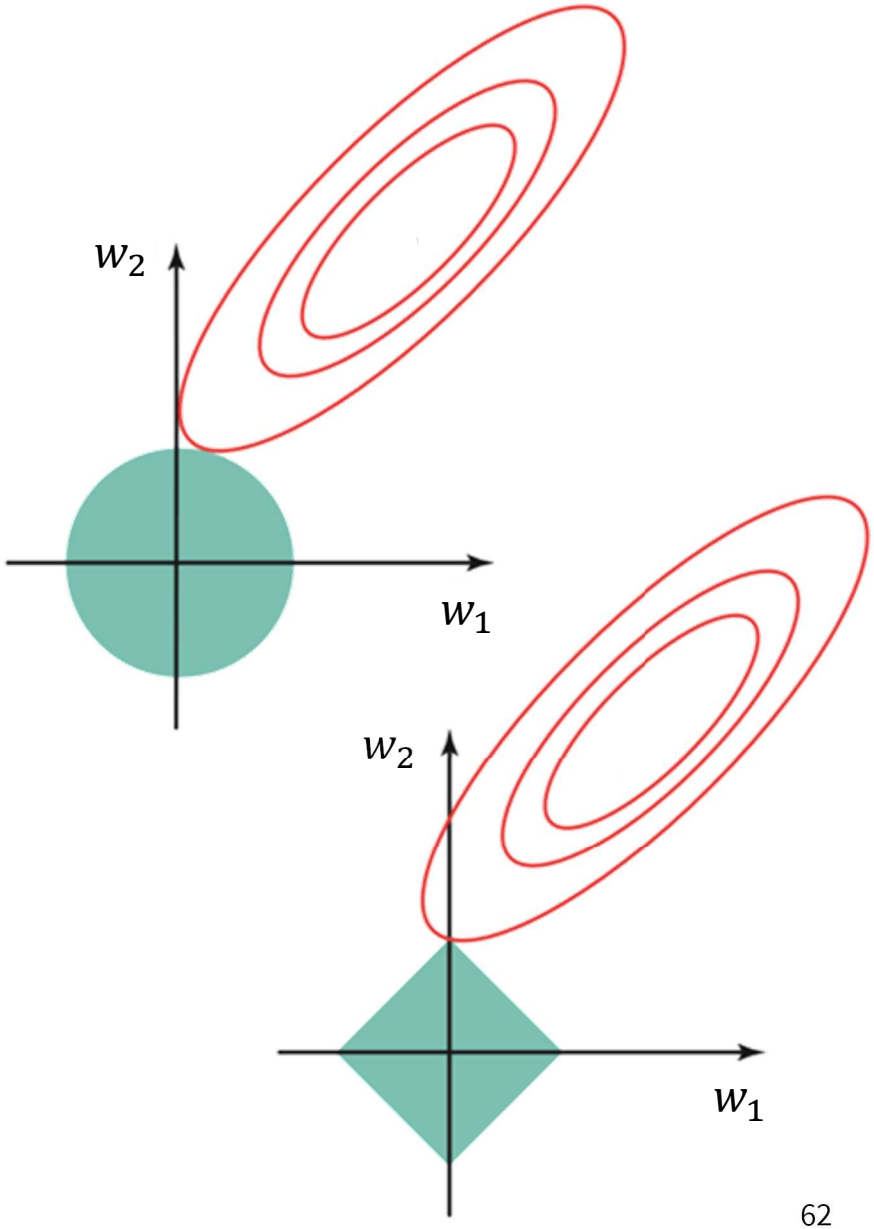
Common penalty functions

- **Ridge aka L2 penalty** $\|\boldsymbol{W}\|_2 = \lambda \sum_j w_j^2$

- **LASSO aka L1 penalty** $\|\boldsymbol{W}\|_1 = \lambda \sum_j |w_j|$

- **Elastic net penalty**

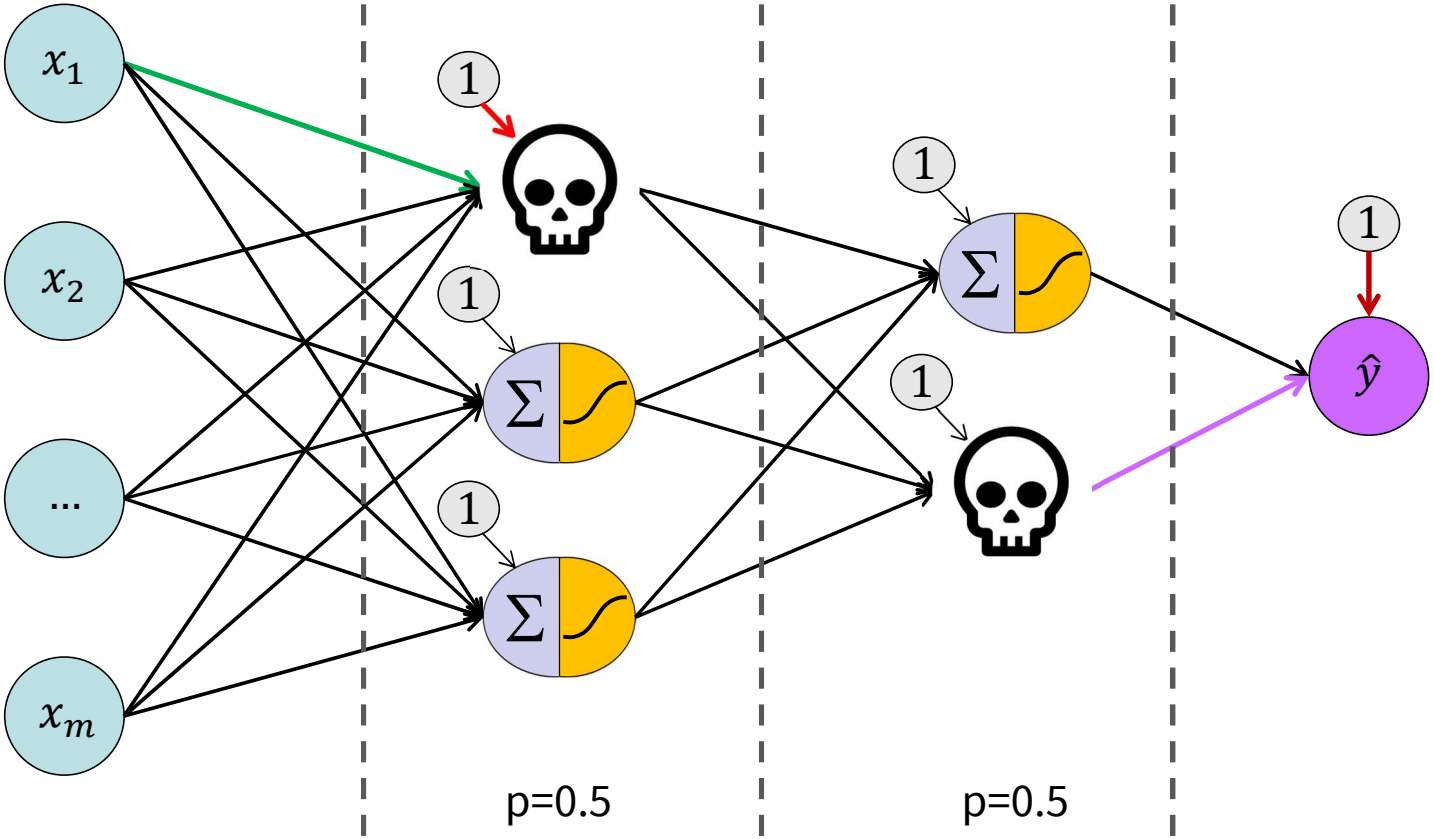
$$\|\boldsymbol{W}\|_{enet} = \lambda \left(\frac{1-\alpha}{2} \sum_j w_j^2 + \alpha \sum_j |w_j| \right)$$



https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a
https://www.datacamp.com/community/tutorials/tutorial-ridge-lasso-elastic-net

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

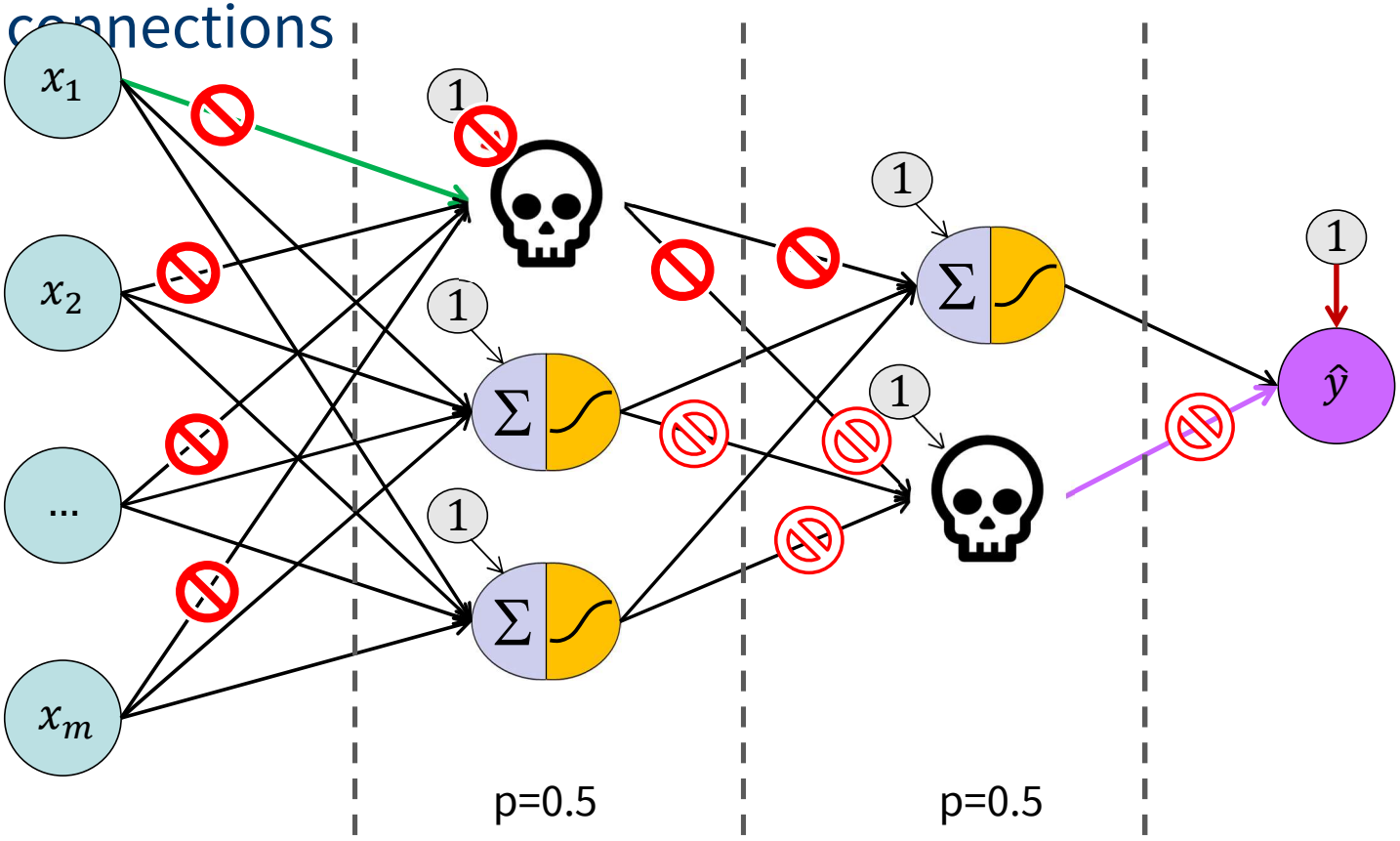
Stochastically deactivate some hidden units during training



Dropout rate:
probability to deactivate a node in a given hidden layer of the NN

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

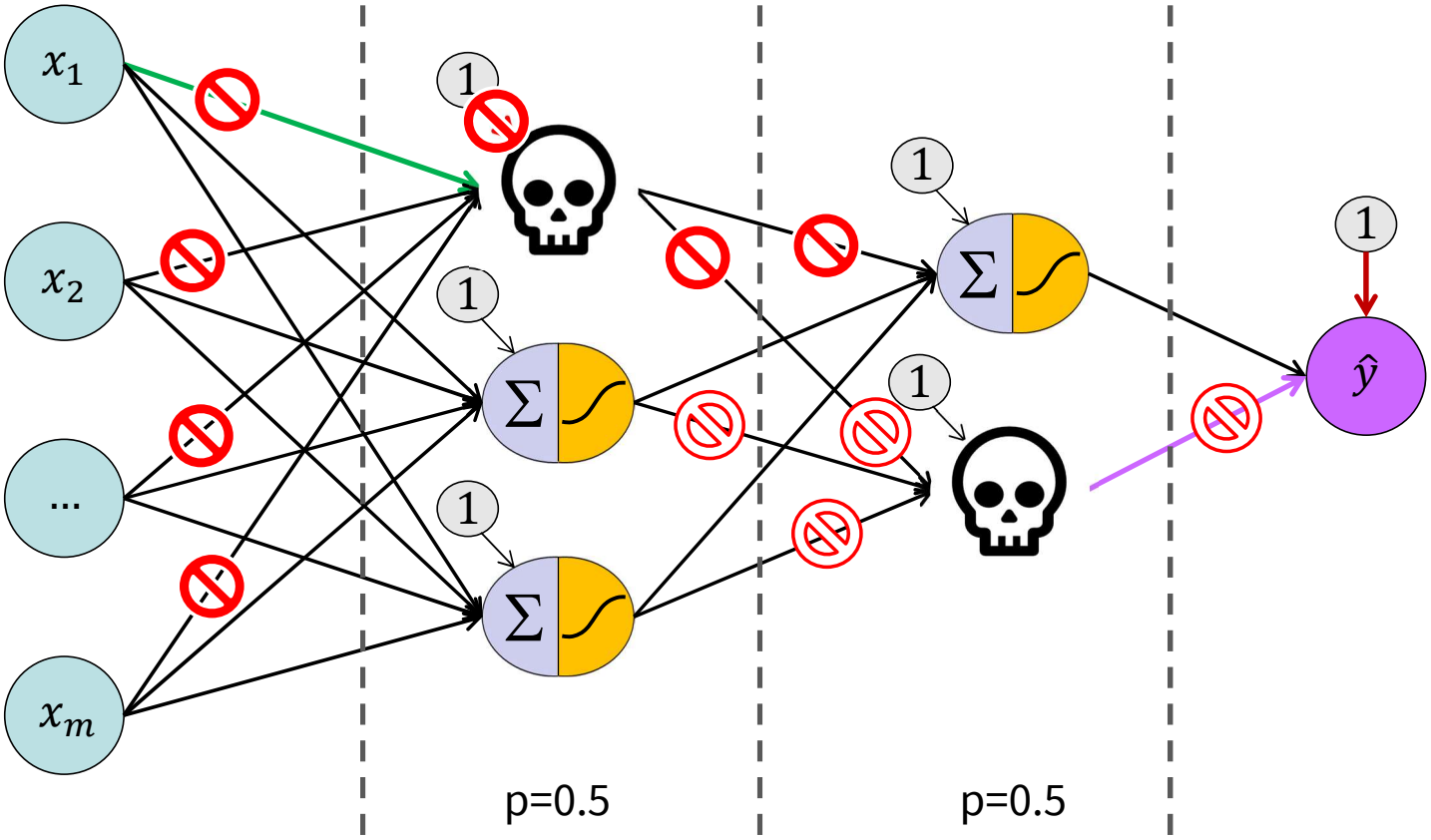
A dropped-out node blocks the flow of information over certain connections



Dropout rate:
probability to deactivate a node in a given hidden layer of the NN

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

The weights of those connections will not get trained

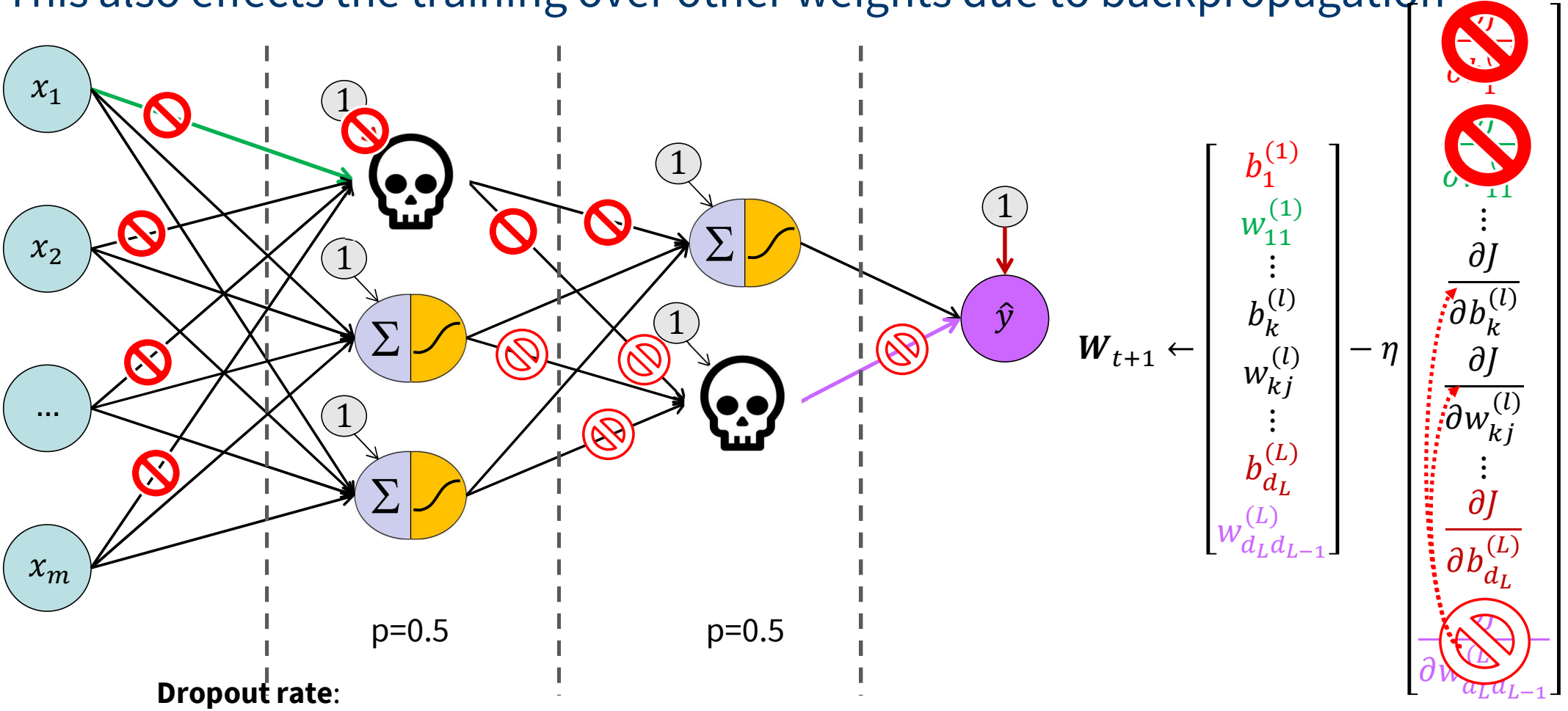


$$\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial J}{\partial b_1^{(l)}} \\ \frac{\partial J}{\partial b_2^{(l)}} \\ \vdots \\ \frac{\partial J}{\partial b_k^{(l)}} \\ \frac{\partial J}{\partial w_{kj}^{(l)}} \\ \vdots \\ \frac{\partial J}{\partial b_{d_L}^{(L)}} \\ \frac{\partial J}{\partial w_{d_L d_{L-1}}^{(L)}} \end{bmatrix}$$

Dropout rate:
probability to deactivate a node in a given hidden layer of the NN

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

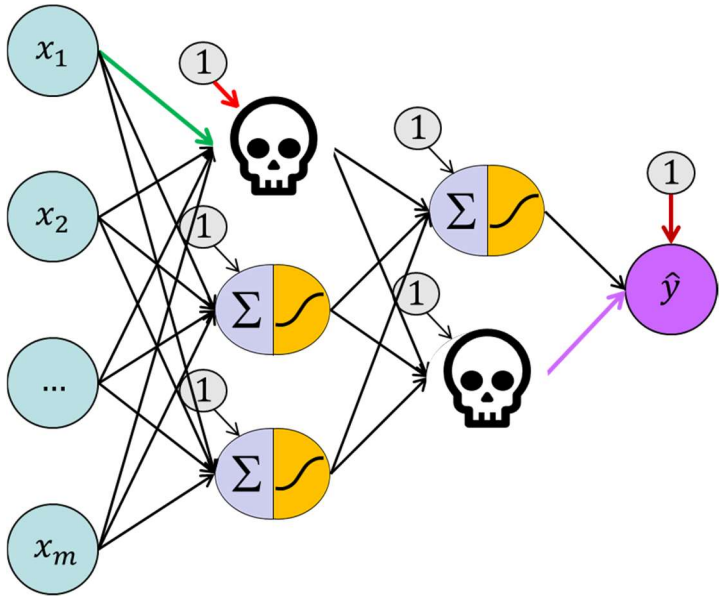
This also effects the training over other weights due to backpropagation



Dropout rate:
probability to deactivate a node in a given hidden layer of the NN

Dropout in a Nutshell

- **Each hidden unit is set to zero with probability p, often set to 50 percent**
  - Applied for each batch of examples or weight update
  - Gradient of that batch by-passes dropped-out neurons
    - Hidden nodes cannot co-adapt to other nodes
    - Hidden nodes must be more generally useful
  - Forces network to not rely on any one node
- **Effectively, dropout gives an ensemble**
  - Training is done across many NNs with different architecture
  - If a node is retained with probability *p* during training, the outgoing weights of that node are multiplied by *p* at test time
  - Like automatic model averaging with beneficial effect on variance

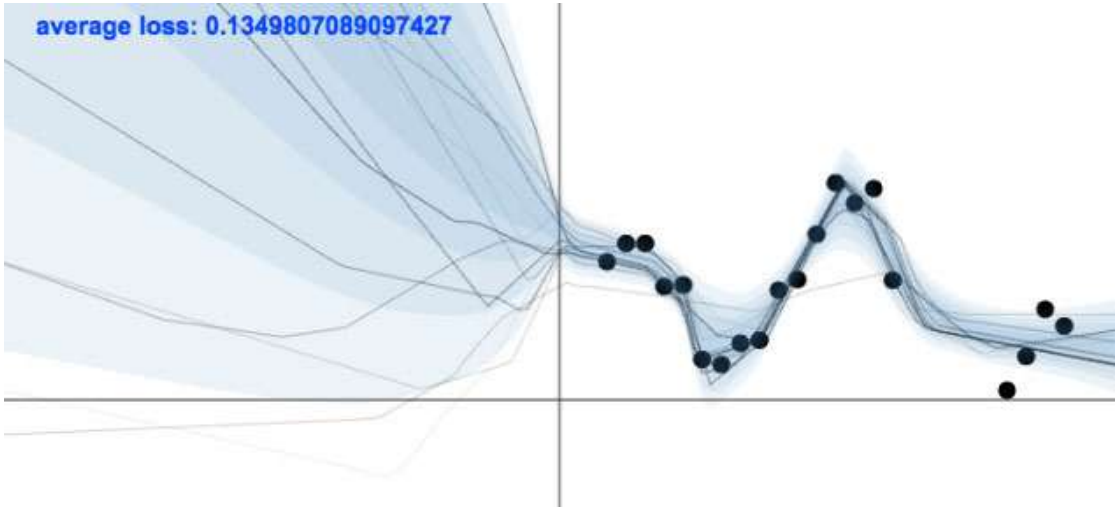


Srivastava et al. (2014), "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 15, 1929-1958

Dropout

Dropout as a Bayesian Approximation

- **Quantifying model uncertainty as key advantage of Bayesian deep learning**
- **Scalability (or lack of it) as key disadvantage of Bayesian deep learning**
- **Applying dropout not only at training but also at testing time as a cheap(er) way to obtain Bayesian uncertainty estimates**

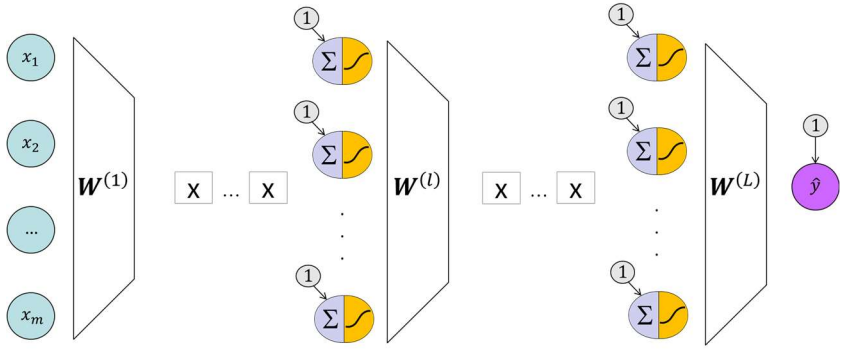


Gal, Y., & Ghahramani, Z. (2016). Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. Proceedings of the 33rd International Conference on Machine Learning (ICML)

Neural Network Model Selection

Tune hyperparameters to optimize performance in a given setting

- **Neural networks belong to the family of semi-parametric models**
- **Hyperparameters to be chosen by data analyst**
  - Number of hidden layers
  - Number of units per hidden layer
  - Hidden layer activation function
  - Regularization
    - Which penalty L1, L2, Enet?
    - Early stopping?
    - Dropout? Dropout rate?
    - Layer-wise decisions possible
  - Weight initialization
  - Batch normalization
  - Network training
    - Training algorithm
    - Number of epochs
    - Learning rate
    - Learning rate schedule
    - Momentum



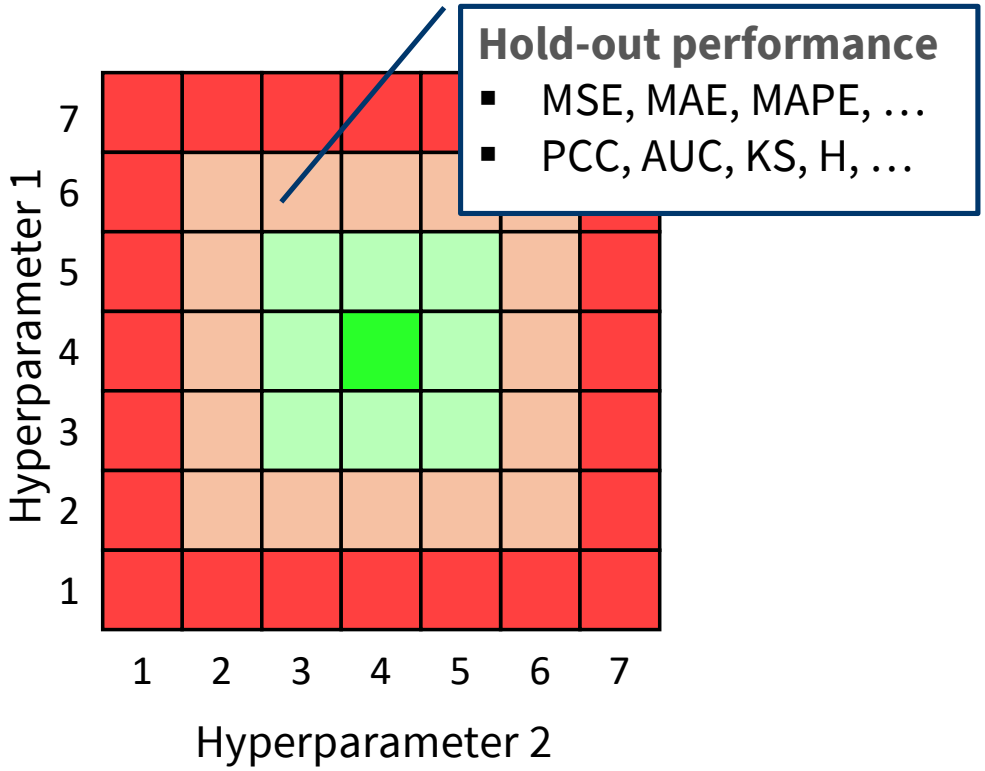
$$Z^{(l)} = \begin{bmatrix} z_1^{(l)} \\ \vdots \\ z_{d_l}^{(l)} \end{bmatrix} = B^{(l)} + \boldsymbol{W}^{(l)} H^{(l-1)}$$

$$H^{(l)} = \begin{bmatrix} h_1^{(l)} \\ \vdots \\ h_{d_l}^{(l)} \end{bmatrix} = g\left(Z^{(l)}\right) = \begin{bmatrix} g\left(z_1^{(l)}\right) \\ \vdots \\ g\left(z_{d_l}^{(l)}\right) \end{bmatrix}$$

Neural Network Model Selection

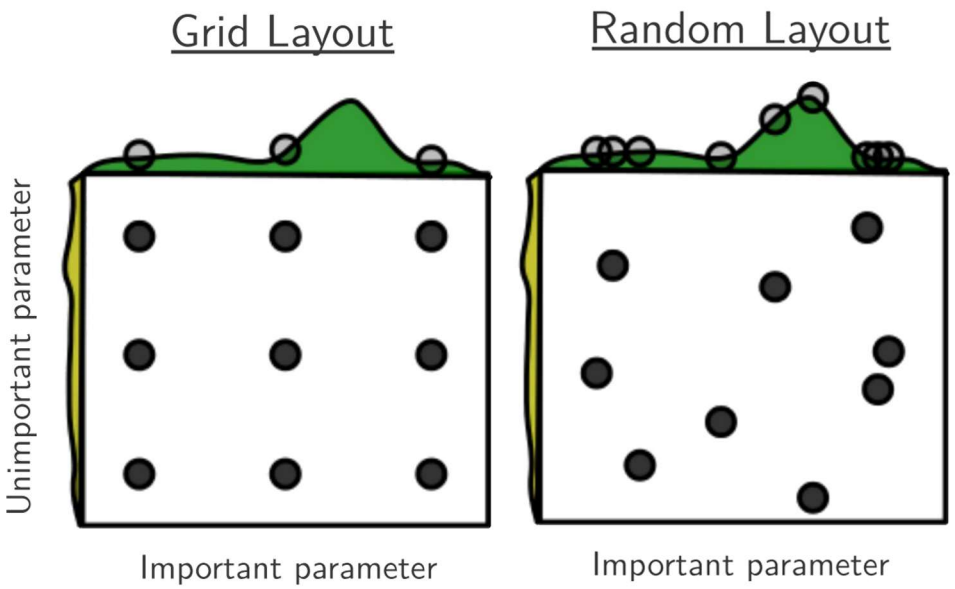
Grid Search

- **Fully enumerative search through all possible combinations of candidate hyperparameter settings**
- **Algorithm**
  - Define candidate range for each hyperparameter
  - Enumerate combinations of candidate values
  - Train model with given configuration
  - Assess model performance on hold-out data
  - Repeat with next configuration
- **Magnify grid resolution in promising regions of the search space**



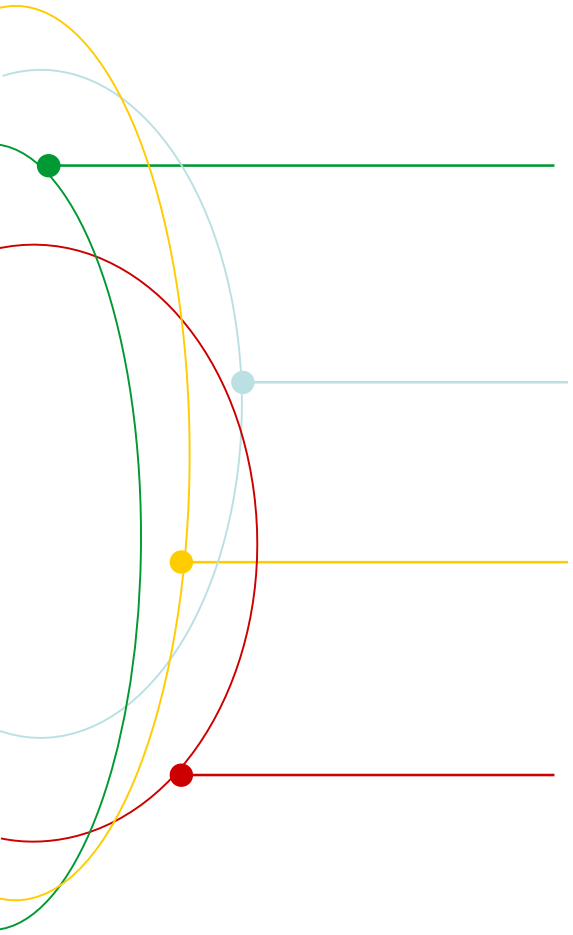
Neural Network Model Selection
Random Search

- **Grid Search is prohibitively expensive for (deep) neural networks**
  - To many distinct hyperparameters
  - To many candidate values per hyperparameter
- **Better use random search**
  - Sample hyperparameter from a given prior distribution
  - Bergstra & Bengio (2012) explore ways to set prior distributions. See also summary in Bengio (2012)
- **Algorithm**
  - Start from random initial solution
  - Sample new solution
  - Compare new to old and keep the better solution
  - Repeat



Summary

Summary



Learning goals

- Foundations of neural networks
- Training NNs using backpropagation

Findings

- Perceptron is equivalent to logistic regression
- NN = directed, multi-layered graph of perceptrons
- NN training finds weights with minimal loss
  - Training via (stochastic) gradient descent
  - Calculating partial derivatives via backprop.
  - Cross entropy loss and softmax for classification
- Address overfitting by regularization, early stopping and dropout
- Many hyperparameter, much need for tuning

What next

- Neural networks for sequential & text data
- Recurrent units and gated cells

Literature

- Bengio, Y. (2012), “Practical recommendations for gradient-based training of deep architectures”, in: Neural Networks: Tricks of the Trade. Springer, pp. 437–478.
- Bergstra, J., & Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization Journal of Machine Learning Research, 13, 281-305.
- Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research. https://proceedings.mlr.press/v9/glorot10a.html
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*: MIT Press.
- Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). *The Elements of Statistical Learning* (2nd ed.). New York: Springer.
- K. He et al. (2015), “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, Archive Pre-Print 1502.01852
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444.

Contact

Stefan Lessmann

Chair of Information Systems
School of Business and Economics
Humboldt-University of Berlin, Germany

Tel. +49.30.2093.5742
Fax. +49.30.2093.5741

stefan.lessmann@hu-berlin.de
http://bit.ly/lessmann

www.hu-berlin.de

Photo: Heike Zappe