Advanced Machine Learning (Semester 2 2025)

Neural Network Architectures (II)

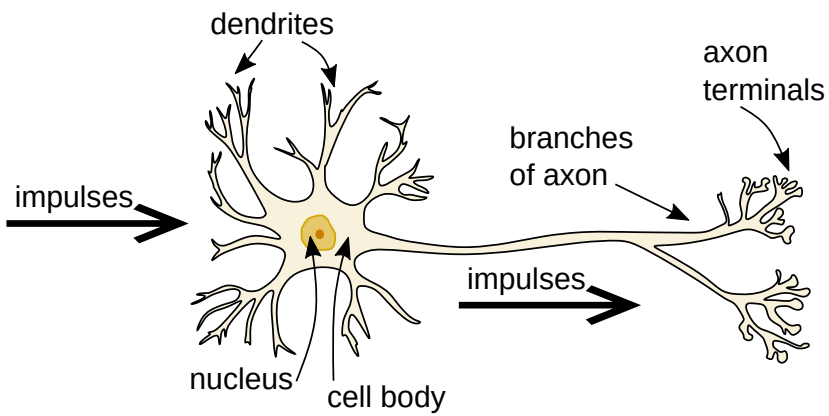
Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

September 2, 2025

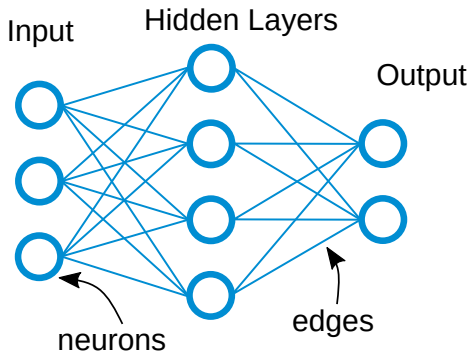
Recap

Artificial neural networks (ANNs), usually simply called neural networks (NNs), are mimicing processes in the human brain to **solve complex data-driven problems**.
The input data is processed through different **layers of artificial neurons** producing the desired output.

biological neural network



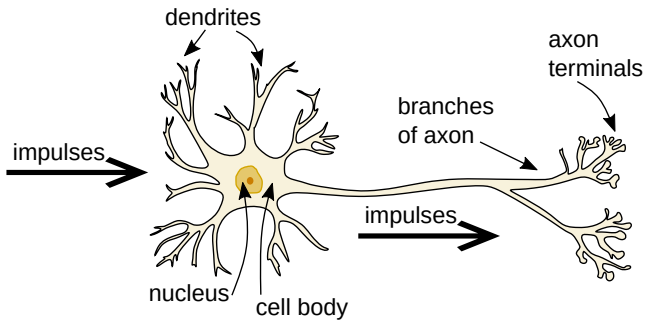
artificial neural network



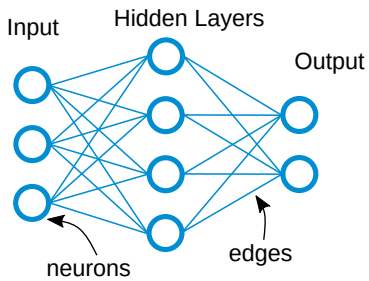
Recap

Artificial neural networks (ANNs), usually simply called neural networks (NNs), are mimicing processes in the human brain to **solve complex data-driven problems**.
The input data is processed through different **layers of artificial neurons** producing the desired output.

biological neural network



artificial neural network

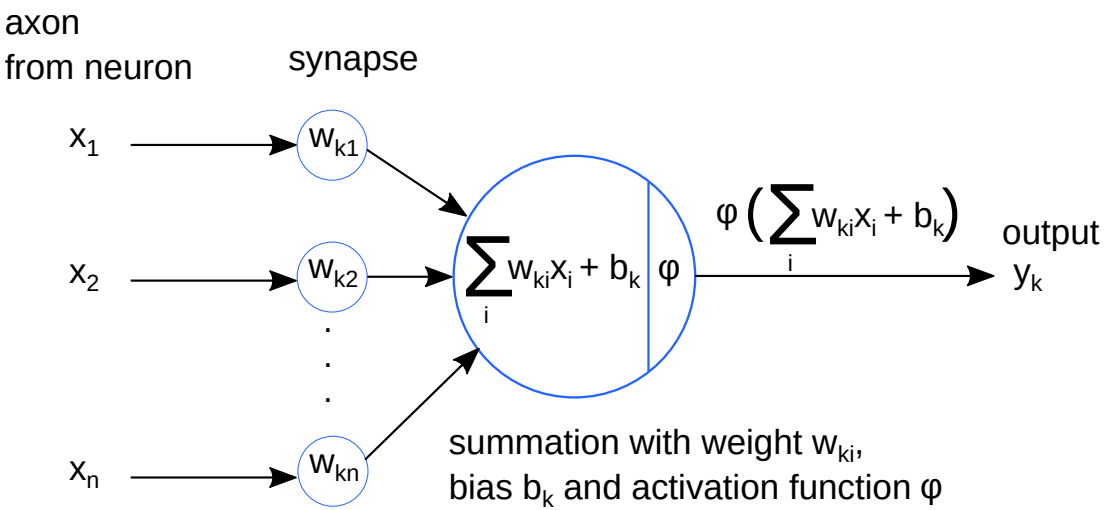


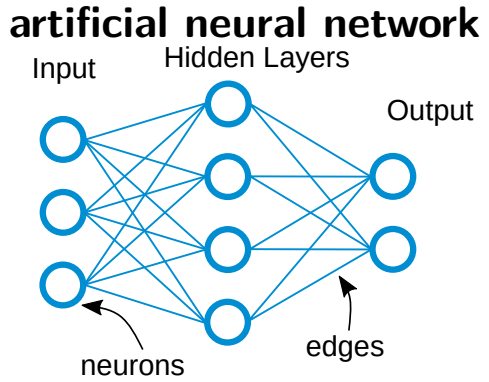
applications:

- pattern recognition
- predictive modeling
- robotics

Recap

simple mathematical model of a **neuron** k:



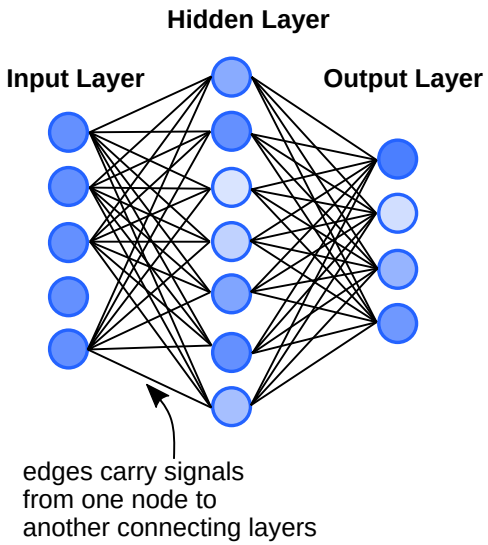


A neuron k receives an n-dimensional input $\mathbf{x}$, which is weighted with the weight vector $\mathbf{w}$. The neuron fires a signal with an intensity given by the activation function φ which controls the amplitude of the output to be within $[0, 1]$.

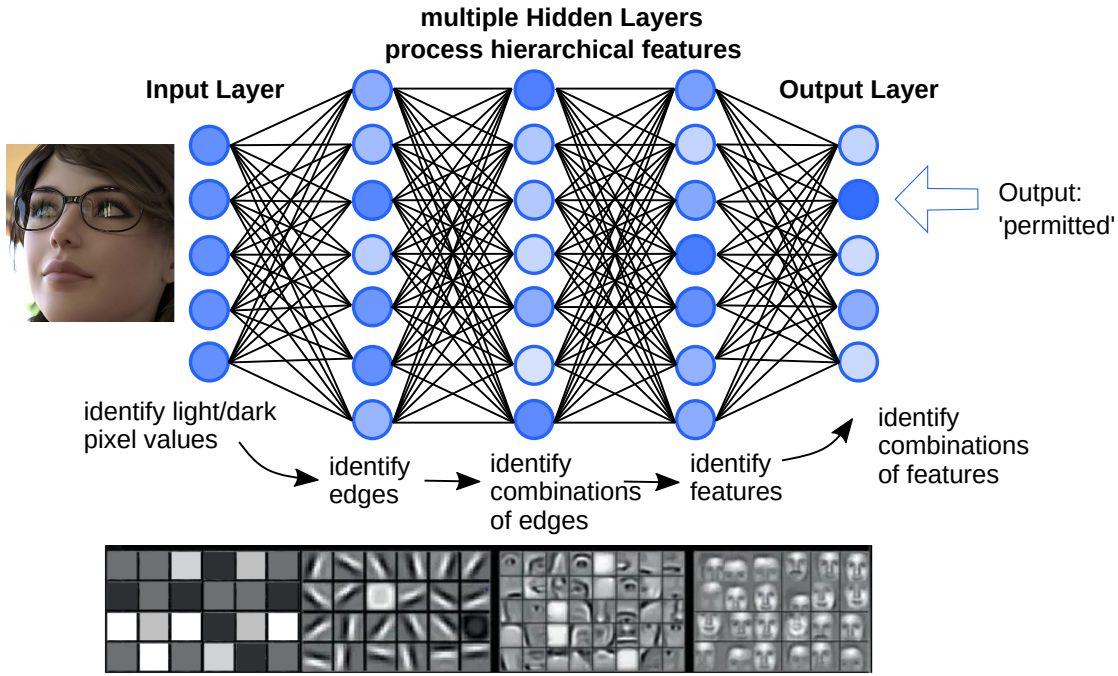
Recap

In contrast to **shallow neural networks** like the Perceptron (input, output, at most one hidden layer), modern **Deep-Learning Networks** can have dozens to hundreds of layers. Each layer trains on a distinct set of features based on the previous layer's output with increasing complexity since they aggregate and recombine features from the previous layer.

1980s-Era Neural Network



Deep Learning Network

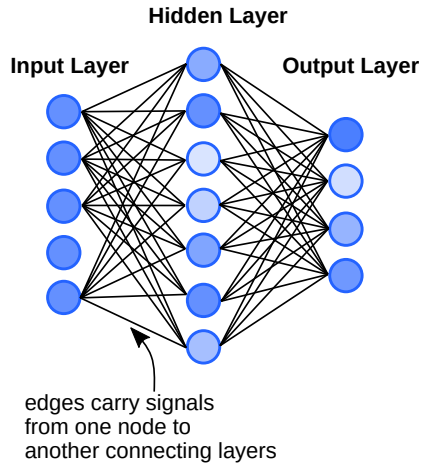


Recap

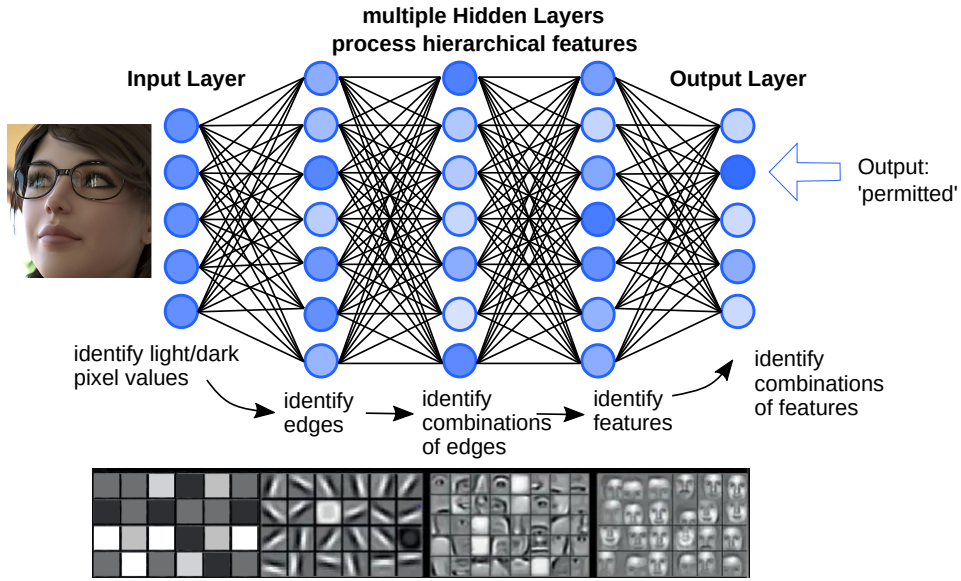
In contrast to **shallow neural networks** like the Perceptron (input, output, at most one hidden layer), modern **Deep-Learning Networks** can have dozens to hundreds of layers. Each layer trains on a distinct set of features based on the previous layer's output with increasing complexity since they aggregate and recombine features from the previous layer.

⇒ **feature hierarchy**

1980s-Era Neural Network



Deep Learning Network

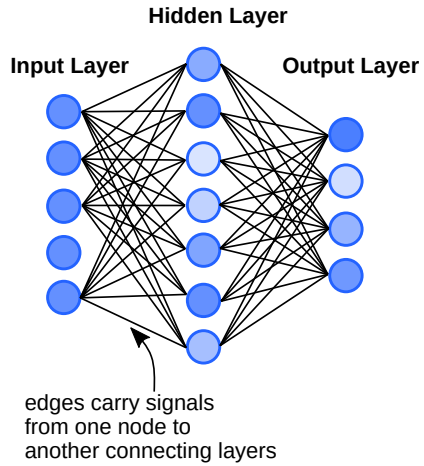


Recap

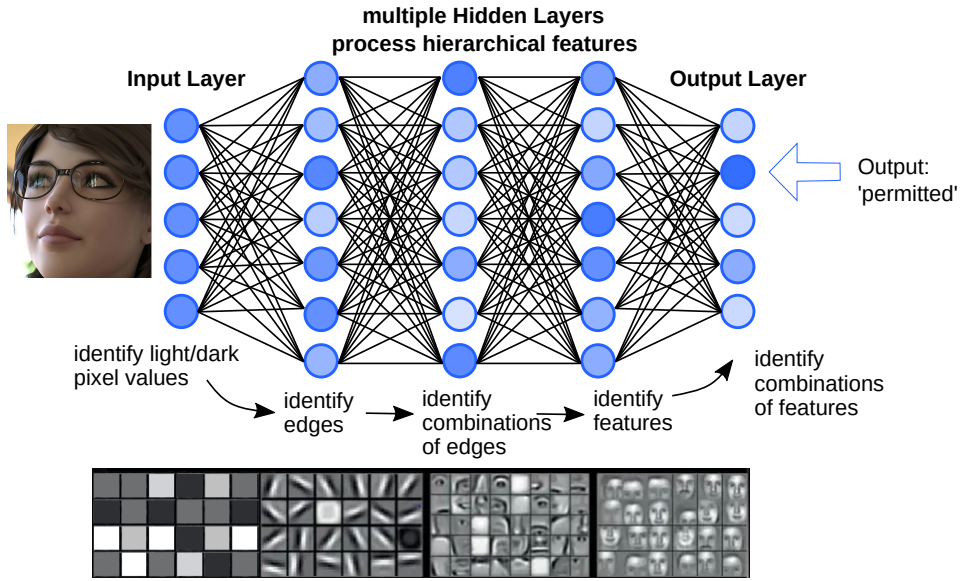
Unlike most traditional machine-learning algorithms, deep-learning networks perform **automatic feature extraction** without human intervention.

⇒ ideal to structure large data sets with hard to define features (e.g.: images, videos, sound)

1980s-Era Neural Network



Deep Learning Network



Recap

Key Terms:

Neuron: A building block of ANN. It is responsible for accepting input data, performing calculations, and producing output.

Artificial Neural Network (ANN): A computational system inspired by the way biological neural networks in the human brain process information.

Deep Neural Network: An ANN with many hidden layers placed between the input layer and the output layer.

Weights: The strength of the connection between two neurons. Weights determine what impact the input will have on the output.

Bias: An additional parameter used along with the sum of the product of weights and inputs to produce an output.

Activation Function: Determines the output of a neural network.

Motivation

How does the neural network **learn**?

Motivation

How does the neural network **learn**?

Was does learning technically mean?

Motivation

How does the neural network **learn**?

Was does learning technically mean?

Recall that the **components** that make up a perceptron:
The output depends on the inputs, weights, bias, and the activation function. The inputs (data) are fixed.
The activation functions are parameterless functions.

The weights increase or decrease the strength of the signal at a connection.

Motivation

How does the neural network **learn**?

Was does learning technically mean?

Recall that the **components** that make up a perceptron:
The output depends on the inputs, weights, bias, and the activation function. The inputs (data) are fixed.
The activation functions are parameterless functions.

The weights increase or decrease the strength of the signal at a connection.

⇒ to train a perceptron, **adjust the weights**

Motivation

in a (multi-layer) perceptron weights are first assigned at random and then updated iteratively by supervised learning until the best set of weights is found

? What is the "best" set of weights?

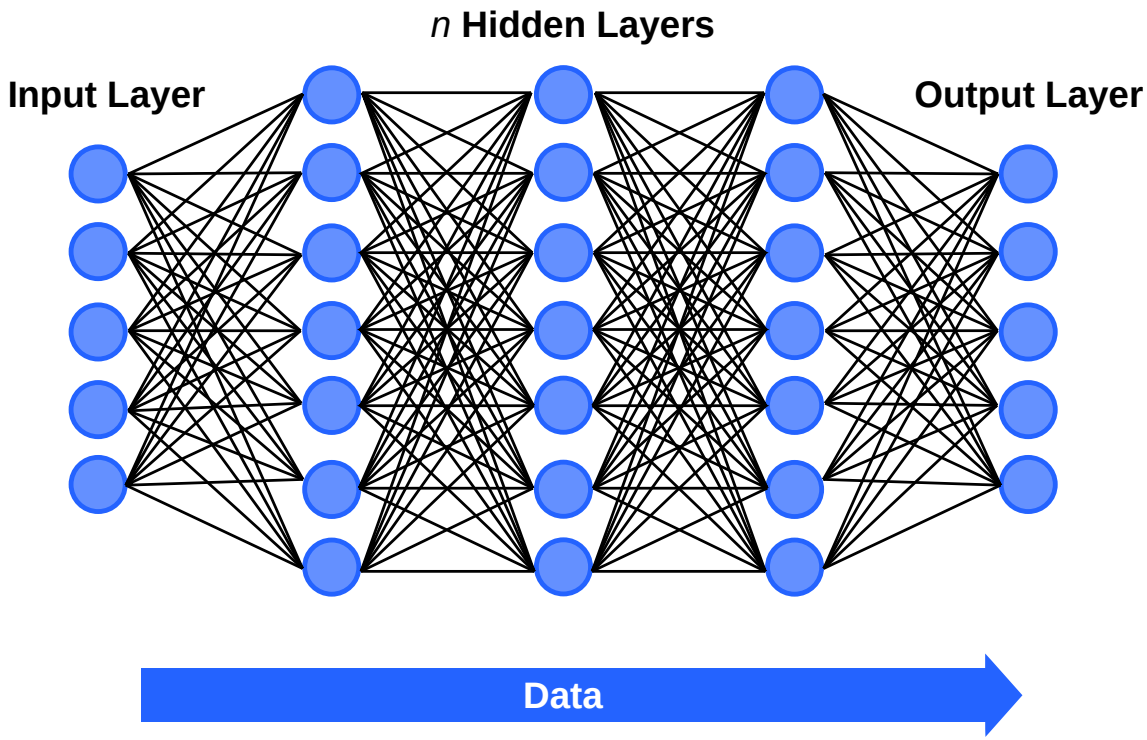
Feed-Forward Neural Networks

we've seen:
A perceptron represents a single neuron. Perceptrons stacked in a row and piled in different layers build a (deep learning) neural network.

Feed-Forward Neural Networks

we've seen:
A perceptron represents a single neuron. Perceptrons stacked in a row and piled in different layers build a (deep learning) neural network.

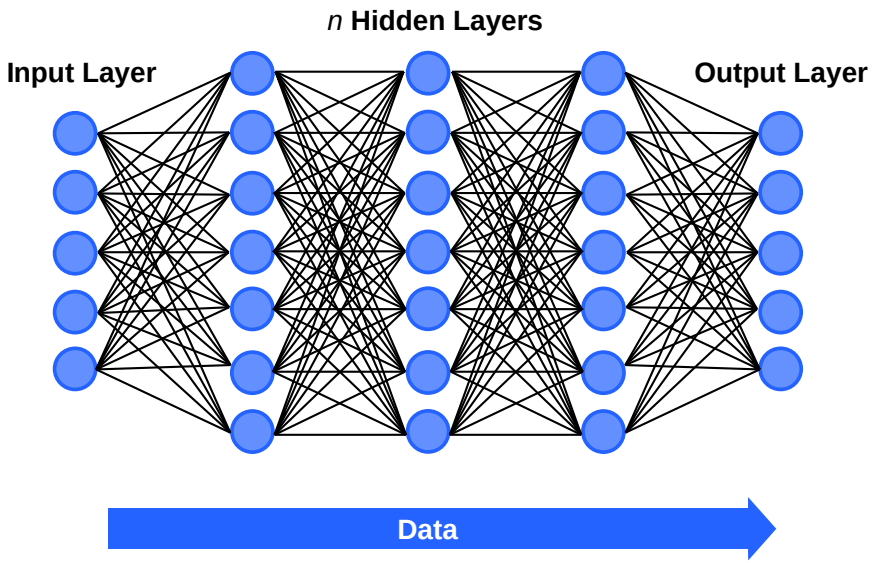
A **feed-forward neural network** (FNN) is a multi-layer neural network where the data moves in a single direction from the input nodes, to the hidden layers where calculations happen, and leaves at the output layer.



Feed-Forward Neural Networks

we've seen:
A perceptron represents a single neuron. Perceptrons stacked in a row and piled in different layers build a (deep learning) neural network.

A **feed-forward neural network** (FNN) is a multi-layer neural network where the data moves in a single direction from the input nodes, to the hidden layers where calculations happen, and leaves at the output layer.



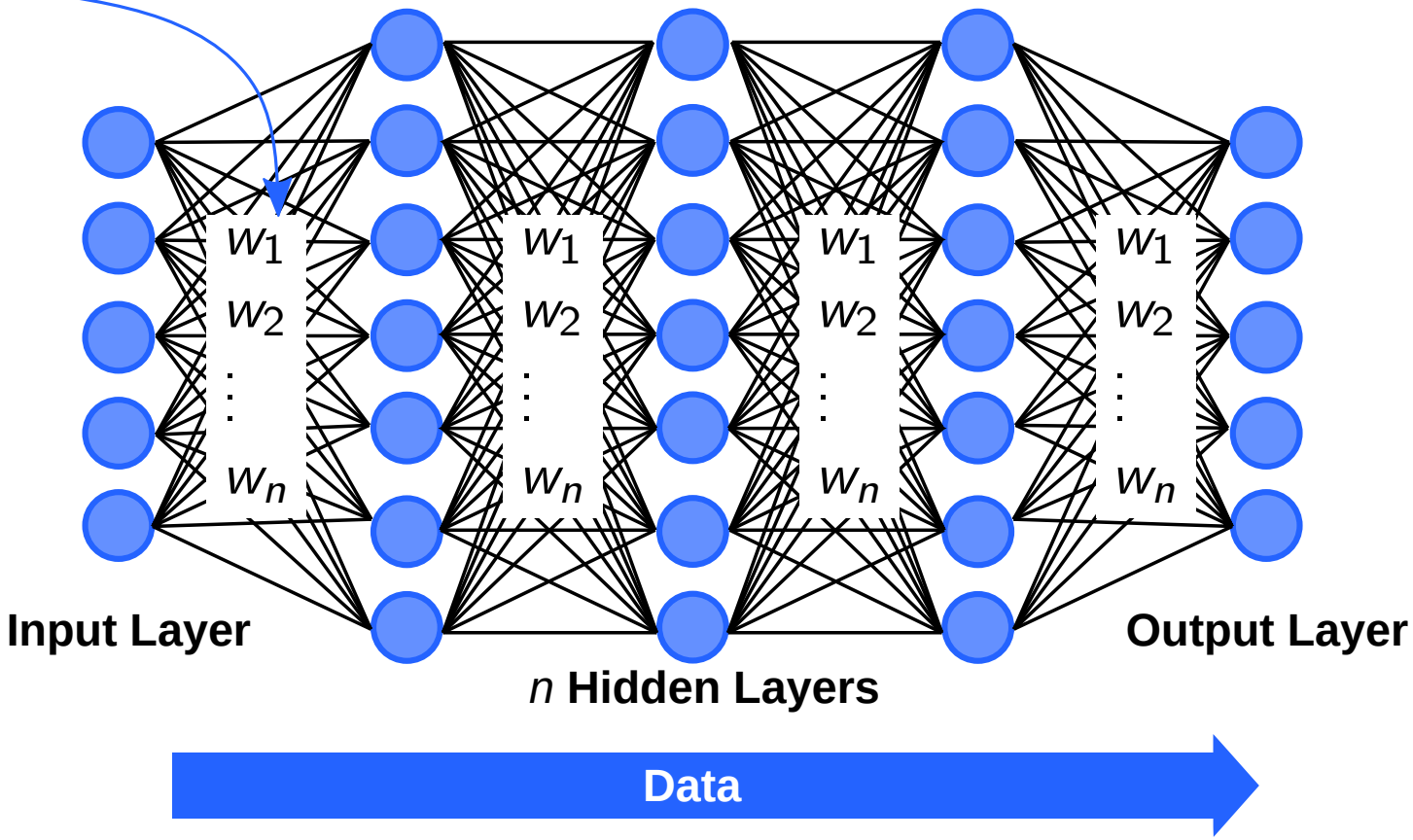
Layers give no feedback to the previous layers (different from e.g. a recurrent neural network where connections form cycles*).

*more on this later

Feed-Forward Neural Networks

calculate weighted sum and add bias:

$$\sum_i w_i x_i + b$$



picks up signals and passes them to the next layer

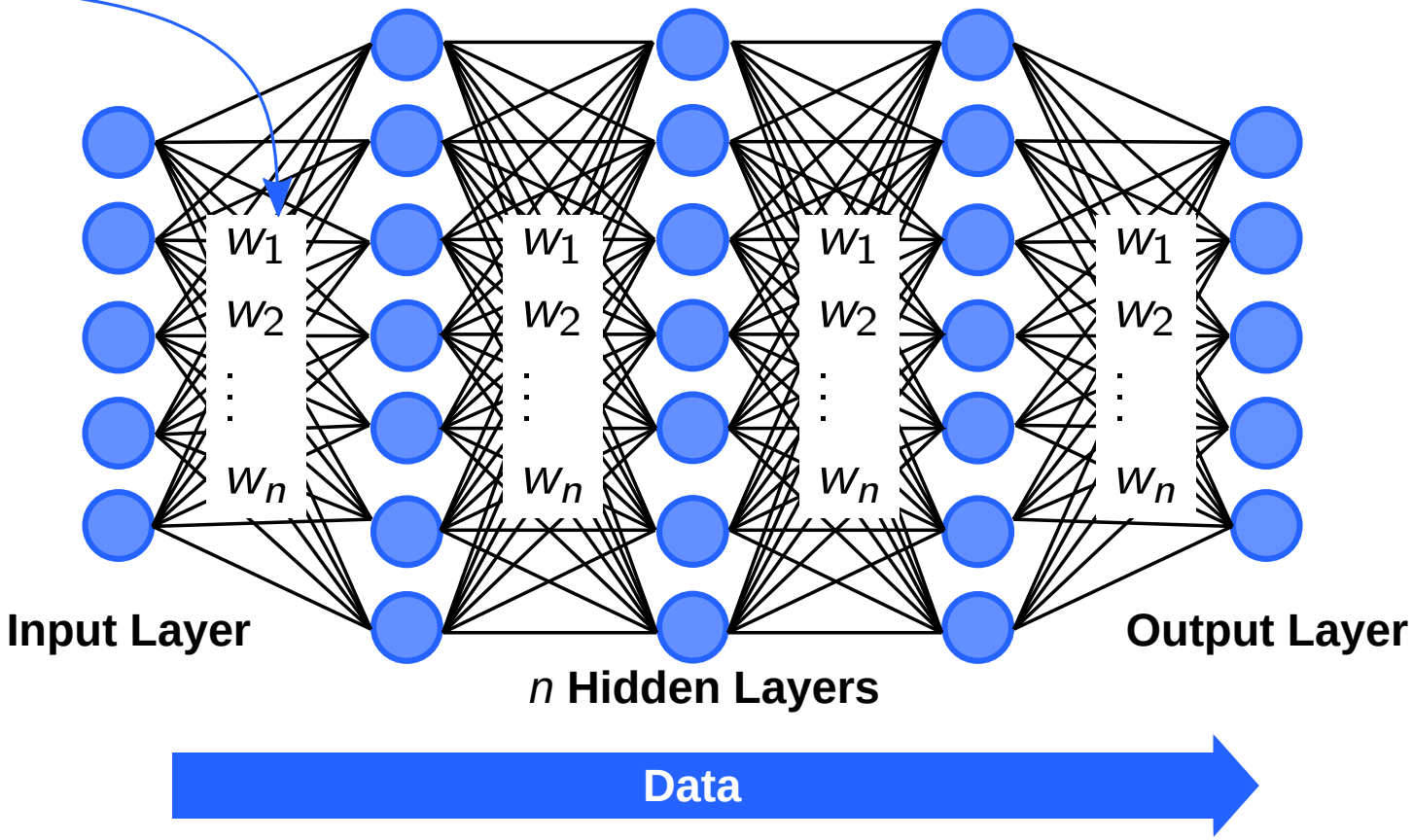
calculations

delivers the final result

Feed-Forward Neural Networks

activation function φ
controls output amplitude:

$$\varphi\left(\sum_i w_i x_i + b\right)$$



picks up signals and passes them to the next layer

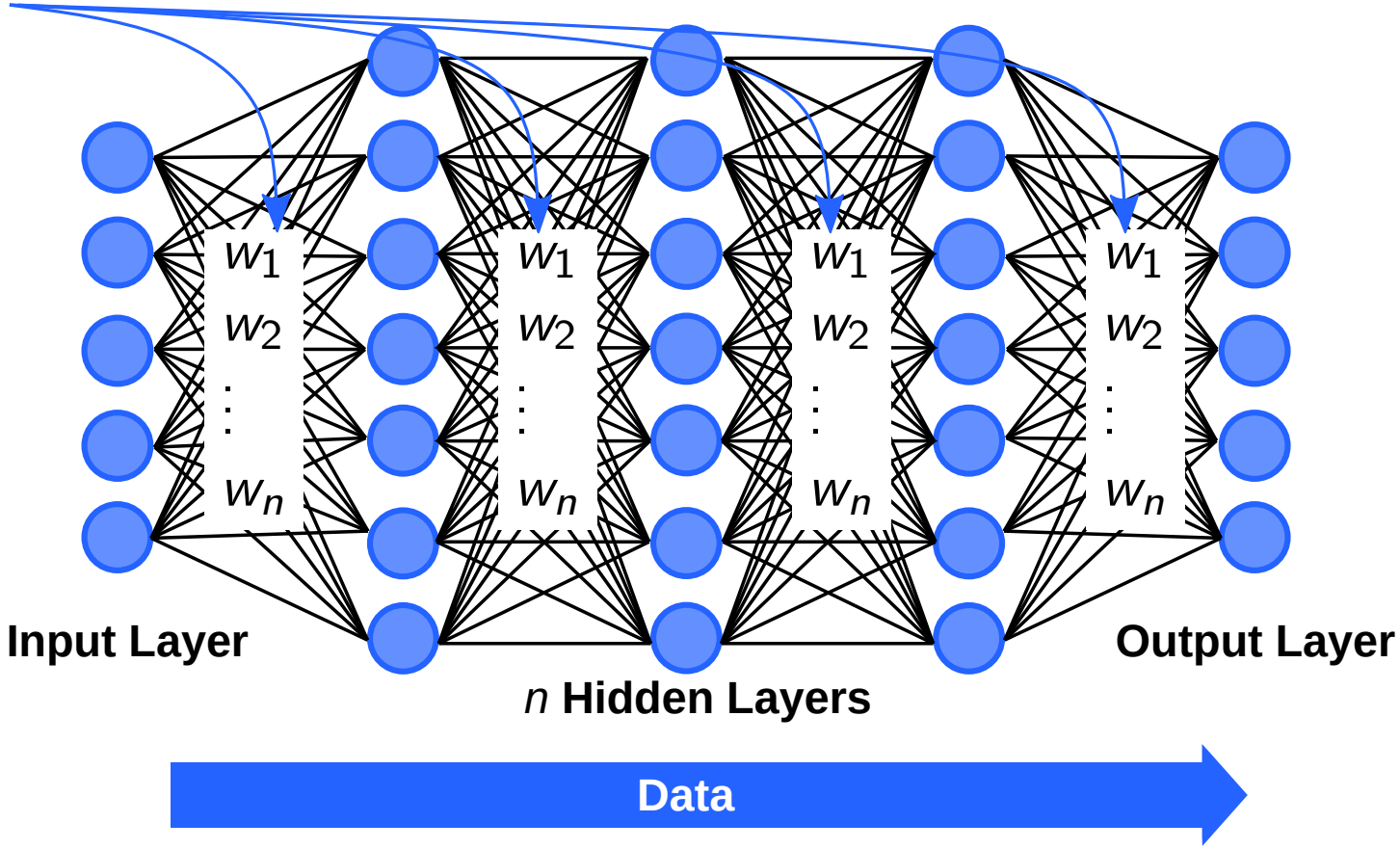
calculations

delivers the final result

Feed-Forward Neural Networks

activation function φ
controls output amplitude:

$$\varphi\left(\sum_i w_i x_i + b\right)$$



picks up signals and passes them to the next layer

calculations

delivers the final result

Feed-Forward Networks

We have to find a **mathematical notation** for that.

Feed-Forward Networks

We have to find a **mathematical notation** for that.

We've seen:

The Perceptron is mathematically represented as:

$$y = \varphi(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b) = \varphi(\mathbf{w} \cdot \mathbf{x} + b)$$

with

$$\varphi(\mathbf{w} \cdot \mathbf{x} + b) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

where $\mathbf{w}$ is a vector of real-valued weights, $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_{i=1}^{n} w_i x_i$, where n is the input vector dimension, and b is the bias.

Feed-Forward Networks

We have to find a **mathematical notation** for that.

We've seen:

The Perceptron is mathematically represented as:

$$y = \varphi(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b) = \varphi(\mathbf{w} \cdot \mathbf{x} + b)$$

with

$$\varphi(\mathbf{w} \cdot \mathbf{x} + b) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

where $\mathbf{w}$ is a vector of real-valued weights, $\mathbf{w} \cdot \mathbf{x}$ is the dot product $\sum_{i=1}^{n} w_i x_i$, where n is the input vector dimension, and b is the bias.

For a particular choice of $\mathbf{w}$ and b, the output y only depends on the input vector $\mathbf{x}$.

Feed-Forward Networks - Mathematical Notation

The Perceptron is mathematically represented as:

$$y = \varphi(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b) = \varphi(\mathbf{w} \cdot \mathbf{x} + b)$$

first an **example**: Going swimming or not (Yes: 1, No: 0). The decision is our predicted outcome, or y. Let's assume the decision is based on three factors:

Is the water warm? (Yes: 1, No: 0)
Is the beach empty? (Yes: 1, No: 0)
Was there a recent shark attack? (Yes: 0, No: 1)

Feed-Forward Networks - Mathematical Notation

The Perceptron is mathematically represented as:

$$y = \varphi(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b) = \varphi(\mathbf{w} \cdot \mathbf{x} + b)$$

first an **example**: Going swimming or not (Yes: 1, No: 0). The decision is our predicted outcome, or y. Let's assume the decision is based on three factors:

Is the water warm? (Yes: 1, No: 0)	$x_1 = 1$
Is the beach empty? (Yes: 1, No: 0)	$x_2 = 0$
Was there a recent shark attack? (Yes: 0, No: 1)	$x_3 = 1$

Feed-Forward Networks - Mathematical Notation

The Perceptron is mathematically represented as:

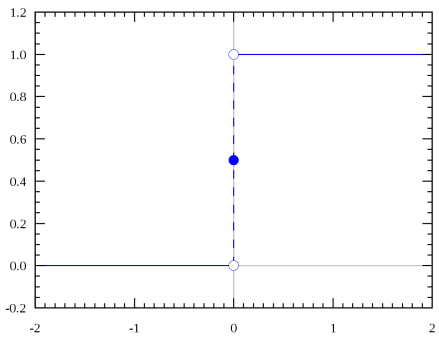
$$y = \varphi(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b) = \varphi(\mathbf{w} \cdot \mathbf{x} + b)$$

first an **example**: Going swimming or not (Yes: 1, No: 0). The decision is our predicted outcome, or y. Let's assume the decision is based on three factors:

Is the water warm? (Yes: 1, No: 0)	$x_1 = 1$
Is the beach empty? (Yes: 1, No: 0)	$x_2 = 0$
Was there a recent shark attack? (Yes: 0, No: 1)	$x_3 = 1$

Also, we set for the weights:

$w_1 = 4$, since the water isn't often warm
$w_2 = 2$, since you're used to the crowds
$w_3 = 4$, since you have a fear of sharks

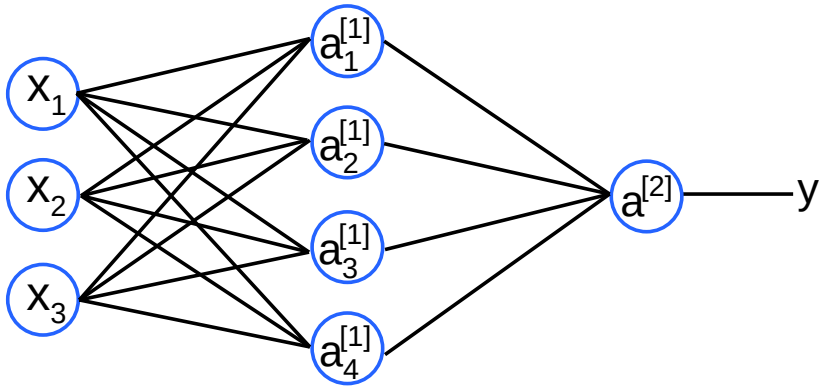


With a threshold value of 3 (b = -3), we finally get

$y = \varphi(1 \times 4 + 0 \times 2 + 1 \times 4 - 3) = \varphi(5) = 1$ (decision: go!)

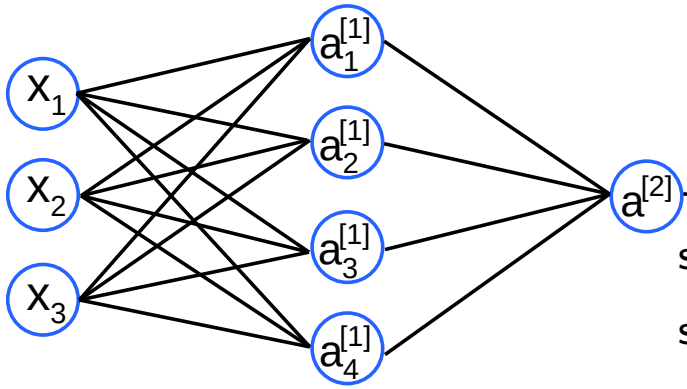
Feed-Forward Networks - Mathematical Notation

based on the perceptron: forward propagation process for a **2-layer perceptron** with a single data set with only 3 features x_1, x_2, x_3



Feed-Forward Networks - Mathematical Notation

based on the perceptron: forward propagation process for a **2-layer perceptron** with a single data set with only 3 features x_1, x_2, x_3



superscript bracket denotes the i-th layer

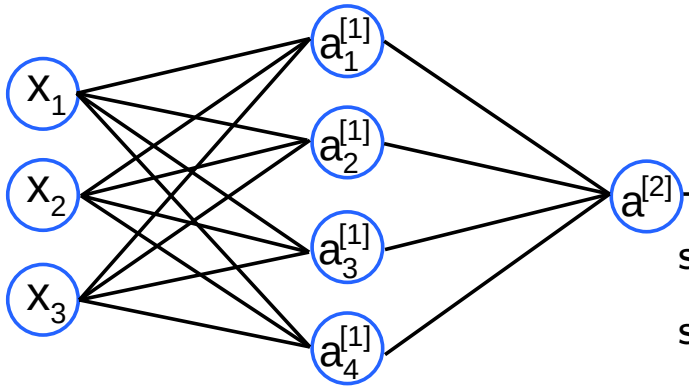
subscript denotes the j-th node in a layer

$$\mathbf{z}^{[1]} = \mathbf{w}^{[1]} \cdot \mathbf{x} + b^{[1]}$$

$$\left.\begin{aligned}
z_1^{[1]} &= \mathbf{w}_1^{[1]}\mathbf{x} + b_1^{[1]} \Rightarrow a_1^{[1]} = \varphi(z_1^{[1]}) \\
z_2^{[1]} &= \mathbf{w}_2^{[1]}\mathbf{x} + b_2^{[1]} \Rightarrow a_2^{[1]} = \varphi(z_2^{[1]}) \\
z_3^{[1]} &= \mathbf{w}_3^{[1]}\mathbf{x} + b_3^{[1]} \Rightarrow a_3^{[1]} = \varphi(z_3^{[1]}) \\
z_4^{[1]} &= \mathbf{w}_4^{[1]}\mathbf{x} + b_4^{[1]} \Rightarrow a_4^{[1]} = \varphi(z_4^{[1]})
\end{aligned}\right\} \quad z^{[2]} = \mathbf{w}^{[2]}\mathbf{a}^{[1]} + b^{[2]}$$

Feed-Forward Networks - Mathematical Notation

based on the perceptron: forward propagation process for a **2-layer perceptron** with a single data set with only 3 features x_1, x_2, x_3



superscript bracket denotes the i-th layer

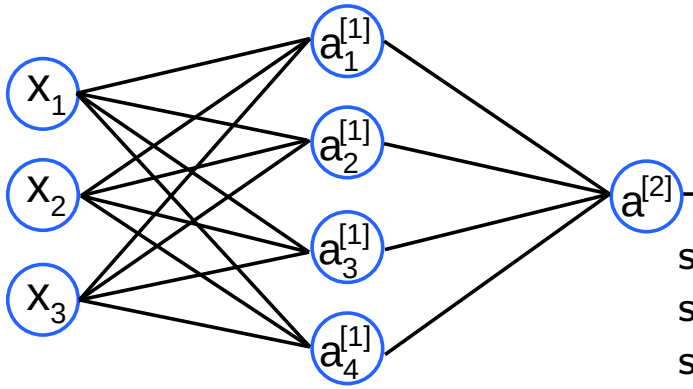
subscript denotes the j-th node in a layer

$$\mathbf{z}^{[1]} = \mathbf{w}^{[1]} \cdot \mathbf{x} + b^{[1]}$$

$$\left.\begin{aligned}
z_1^{[1]} &= \mathbf{w}_1^{[1]}\mathbf{x} + b_1^{[1]} \Rightarrow a_1^{[1]} = \varphi(z_1^{[1]}) \\
z_2^{[1]} &= \mathbf{w}_2^{[1]}\mathbf{x} + b_2^{[1]} \Rightarrow a_2^{[1]} = \varphi(z_2^{[1]}) \\
z_3^{[1]} &= \mathbf{w}_3^{[1]}\mathbf{x} + b_3^{[1]} \Rightarrow a_3^{[1]} = \varphi(z_3^{[1]}) \\
z_4^{[1]} &= \mathbf{w}_4^{[1]}\mathbf{x} + b_4^{[1]} \Rightarrow a_4^{[1]} = \varphi(z_4^{[1]})
\end{aligned}\right\}
\begin{aligned}
z^{[2]} &= \mathbf{w}^{[2]}\mathbf{a}^{[1]} + b^{[2]} \\
\Rightarrow y &= a^{[2]} = \varphi(z^{[2]})
\end{aligned}$$

Feed-Forward Networks - Mathematical Notation

now we **generalize** this: vectorized forward propagation for m training examples for the same neural network



superscript bracket denotes the i-th layer
subscript denotes the j-th node in a layer
superscript round bracket denotes the k-th training example

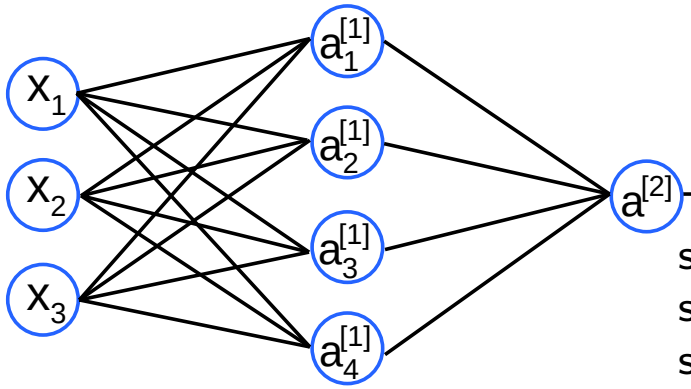
$$\mathbf{z}^{[1]} = \mathbf{w}^{[1]} \cdot \mathbf{x} + b^{[1]} = \begin{bmatrix} \mathbf{w}_1^{[1]} \\ \vdots \\ \mathbf{w}_4^{[1]} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ \vdots & \vdots & \ddots & \\ x_3^{(1)} & x_3^{(2)} & \cdots & x_3^{(m)} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ \vdots \\ b_4^{[1]} \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} \mathbf{w}_1^{[1]}\mathbf{x}^{(1)} + b_1^{[1]} & \mathbf{w}_1^{[1]}\mathbf{x}^{(2)} + b_1^{[1]} & \cdots & \mathbf{w}_1^{[1]}\mathbf{x}^{(m)} + b_1^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_4^{[1]}\mathbf{x}^{(1)} + b_4^{[1]} & \cdots & \cdots & \mathbf{w}_4^{[1]}\mathbf{x}^{(m)} + b_4^{[1]} \end{bmatrix}}_{m \text{ training examples}} \} \text{4 nodes in layer 1}$$

$$\mathbf{a}^{[1]} = \varphi(\mathbf{z}^{[1]})$$

Feed-Forward Networks - Mathematical Notation

now we **generalize** this: vectorized forward propagation for m training examples for the same neural network



superscript bracket denotes the i-th layer
subscript denotes the j-th node in a layer
superscript round bracket denotes the k-th training example

$$\mathbf{z}^{[2]} = \mathbf{w}^{[2]} \begin{bmatrix} \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \dots & \mathbf{a}^{[1](m)} \end{bmatrix} + b^{[2]}$$
$$= \begin{bmatrix} \mathbf{w}^{[2]}\mathbf{a}^{[1](1)} + b_1^{[2]} & \dots & \mathbf{w}^{[2]}\mathbf{a}^{[1](m)} + b^{[2]} \end{bmatrix}$$

$$\mathbf{a}^{[2]} = \varphi(\mathbf{z}^{[2]})$$

Feed-Forward Networks - Mathematical Notation

thought:

Every layer computes a linear regression **if** activation functions φ are absent.

Could this be simplified?

Feed-Forward Networks - Mathematical Notation

thought:

Every layer computes a linear regression **if** activation functions φ are absent.

Could this be simplified?

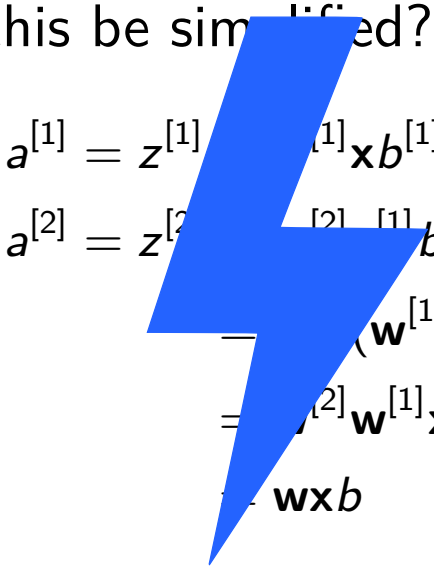
$$
\begin{aligned}
a^{[1]} = z^{[1]} &= \mathbf{w}^{[1]}\mathbf{x}b^{[1]} \\
a^{[2]} = z^{[2]} &= \mathbf{w}^{[2]}a^{[1]}b^{[2]} \\
&= \mathbf{w}^{[2]}(\mathbf{w}^{[1]}\mathbf{x} + b^{[1]}) + b^{[2]} \\
&= \mathbf{w}^{[2]}\mathbf{w}^{[1]}\mathbf{x} + (\mathbf{w}^{[2]}b^{[1]}) + b^{[2]}) \\
&= \mathbf{w}\mathbf{x}b
\end{aligned}
$$

Feed-Forward Networks - Mathematical Notation

thought:

Every layer computes a linear regression **if** activation functions φ are absent.

Could this be simplified?



$$a^{[1]} = z^{[1]} \quad [1]\mathbf{x}b^{[1]}$$

$$a^{[2]} = z^{[2]} \quad [2]\ [1]\ b^{[2]}$$

$$= (\mathbf{w}^{[1]}\mathbf{x} + b^{[1]}) + b^{[2]}$$

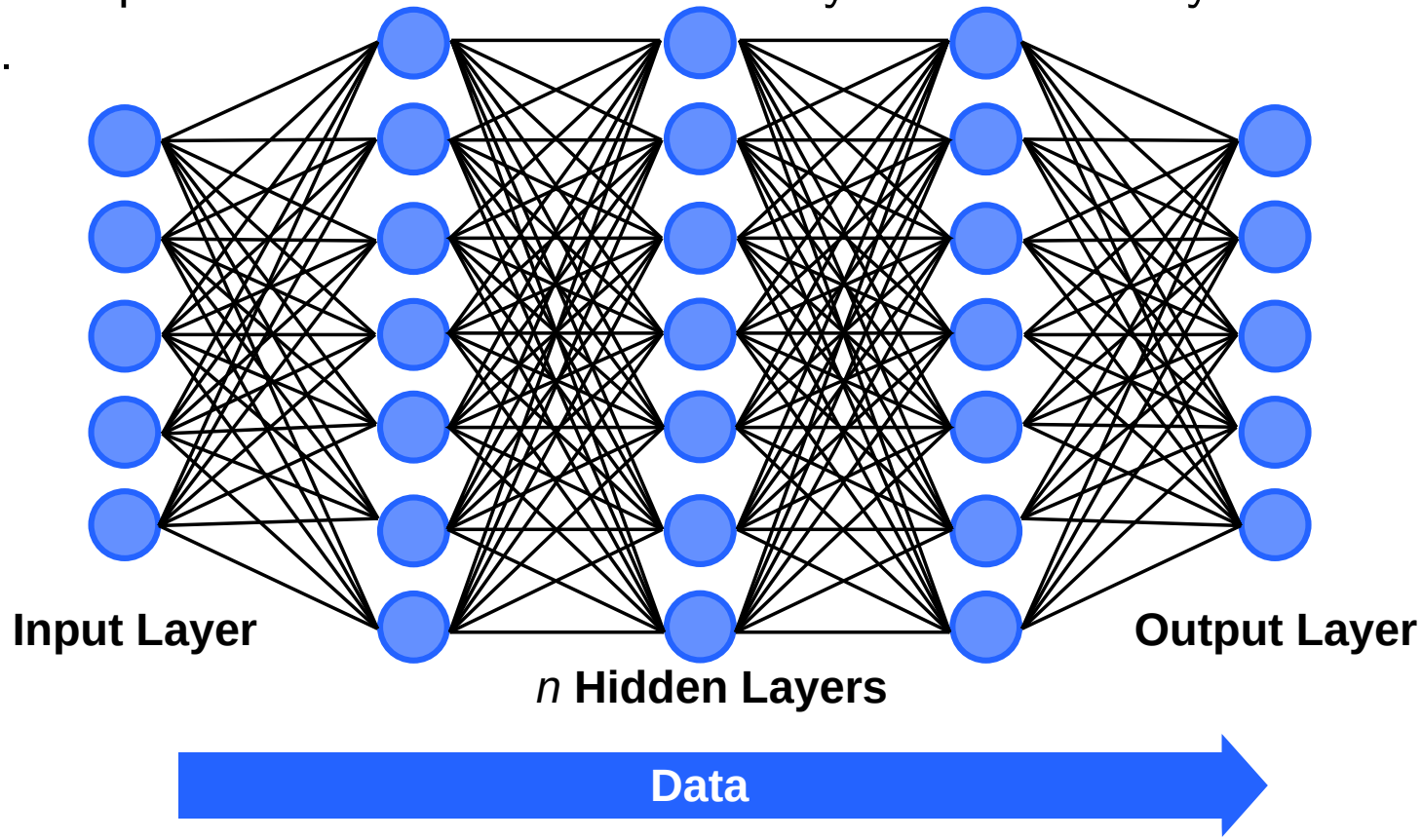
$$= [2]\mathbf{w}^{[1]}\mathbf{x} + (\mathbf{w}^{[2]}b^{[1]}) + b^{[2]})$$

$$= \mathbf{w}\mathbf{x}b$$

From the above mathematical justification, it turns out that every layer would have been computing a linear regression graph, which is essentially meaningless for the Perceptron to add more hidden layers for introducing greater non-linearity to its computation. Hence, the activation functions are critical for deep learning architecture.

Fully Connected Networks

Most neural networks are fully connected: each hidden unit and each output unit is connected to every unit in the layers either side.



Training Neural Networks

As we start to think about more practical use cases for neural networks, like image recognition or classification, we'll leverage supervised learning, or labeled datasets, to train the algorithm (to adjust the weights).

Learning

Learning (or training) is the **adaptation of the neural network** to better handle a task **by considering a training set with known results** $\hat{y}$.

Learning

Learning (or training) is the **adaptation of the neural network** to better handle a task **by considering a training set with known results** $\hat{y}$.

Learning involves adjusting the weights (and optional thresholds) of the network to improve the accuracy of the result y. This is done by minimizing the observed errors (as results are known for the training set). Learning is complete when examining additional observations (enhancing the training set) or further iterations are not reducing the error rate.

Even after learning, the error rate typically does not reach 0.

Learning

How to set the weights $\mathbf{w}$ (and bias $\mathbf{b}$)?

Learning

How to set the weights **w** (and bias **b**)?

We've seen:
Weights are numeric values which are multiplied with inputs.

Learning

How to set the weights **w** (and bias **b**)?

We've seen:
Weights are numeric values which are multiplied with inputs.

In **backpropagation**, they are modified to reduce a loss function:

Learning

How to set the weights **w** (and bias **b**)?

We've seen:
Weights are numeric values which are multiplied with inputs.

In **backpropagation**, they are modified to reduce a loss function:

Training starts with all of its weights (and thresholds) initialized with random values. Training data is processed by the neural network. During training, the weights and thresholds are continually adjusted to **reduce a loss function** (a function of the difference between expected outputs $\hat{y}$ vs. training outputs y) until training data with the same labels consistently yield **similar and correct outputs** $y \sim \hat{y}$ so it **converges**.

Learning

How to set the weights **w** (and bias **b**)?

We've seen:
Weights are numeric values which are multiplied with inputs.

In **backpropagation**, they are modified to reduce a loss function:

Training starts with all of its weights (and thresholds) initialized with random values. Training data is processed by the neural network. During training, the weights and thresholds are continually adjusted to **reduce a loss function** (a function of the difference between expected outputs $\hat{y}$ vs. training outputs y) until training data with the same labels consistently yield **similar and correct outputs** $y \sim \hat{y}$ so it **converges**.

Once the neural network is applied to new data, weights are machine-learnt values.

Backpropagation

The general **implementation** of the backpropagation algorithm is as follows:

0. The interconnections are assigned weights $\mathbf{w}$ at random.

Iteratively:

1. Perform a **forward pass** (also called *inference*) through the neural network using $\mathbf{w}$ from previous step to obtain output $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$.

2. Calculate new weights from loss function E

$$\mathbf{w}^* = \arg\min_w \sum_{n=1}^{N} E(\mathbf{y}^{(n)}, \hat{\mathbf{y}}^{(n)})$$

where $\mathbf{y} = f(\mathbf{x}, \mathbf{w})$ is the output of a neural network with input vector $\mathbf{x}$, weight vector $\mathbf{w}$, and $\hat{\mathbf{y}}$ is the vector of expected outputs

Backpropagation - Training the Perceptron

We're first looking at an **individual Perceptron** j.

1. **Initialize the weights and calculate the output**
   - weights are initialized with random values
   - a forward pass is executed, result y_j

2. **Calculating the Error**

The loss function has to be dependent on the weights and it needs to relate actual output y_j of a perceptron j to desired output $\hat{y}_j$. We define the error as:

$$e_j = \hat{y}_j - y_j$$

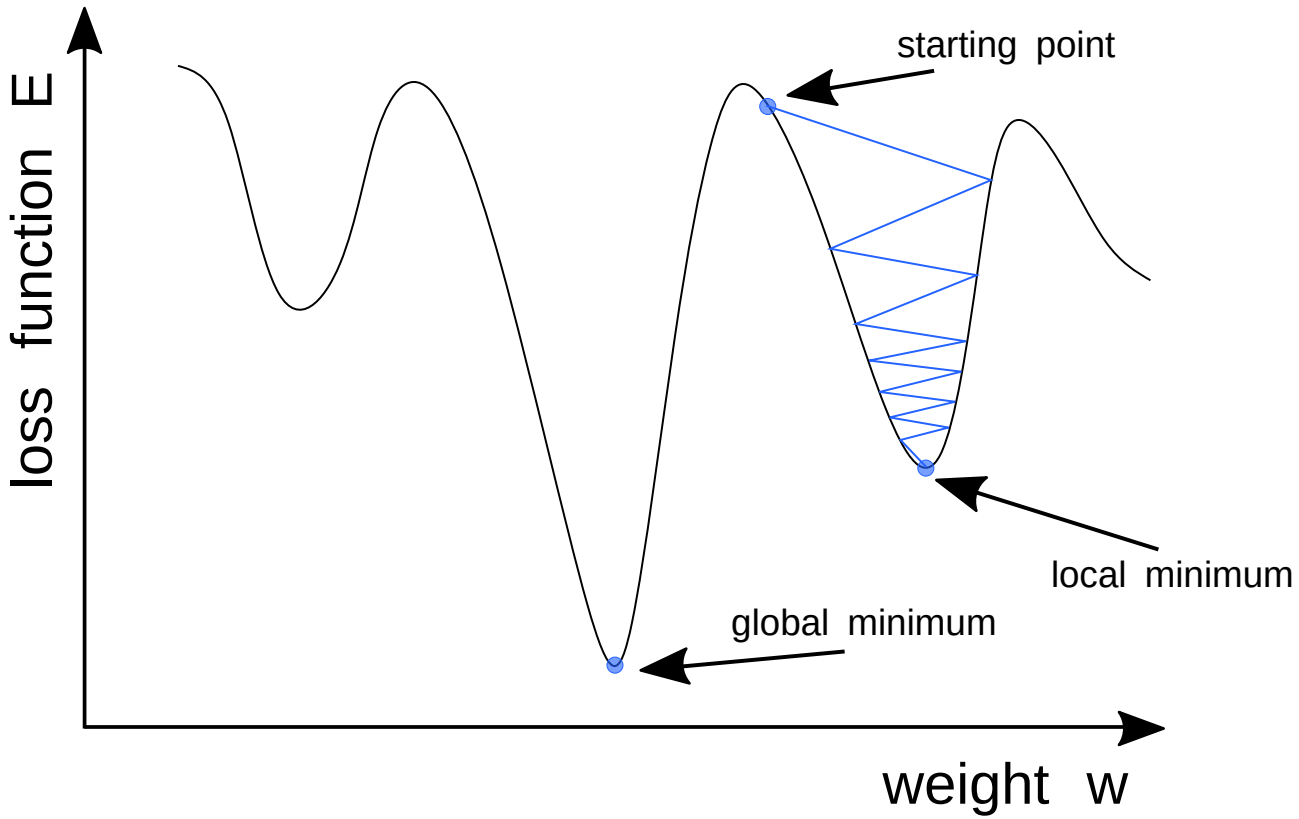
This function e_j does relate $\hat{y}_j$ to y_j and is in fact dependent on the weights because we know the y_j term itself is dependent on the weights. As we want to minimize e_j, we square it to make sure e_j is always positive, which gives us the **loss function**:

$$E_j = \frac{1}{2} e_j^2 = \frac{1}{2} (\hat{y}_j - y_j)^2$$

Training Neural Networks

training neural networks is a **non-convex optimization problem**

this means we can run into many local minima during training



Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (I)

Now that we have a loss function $E_j = \frac{1}{2} e_j^2 = \frac{1}{2}(\hat{y}_j - y_j)^2$, we can use it to determine how to update w_{ij} to reduce the error.

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (I)

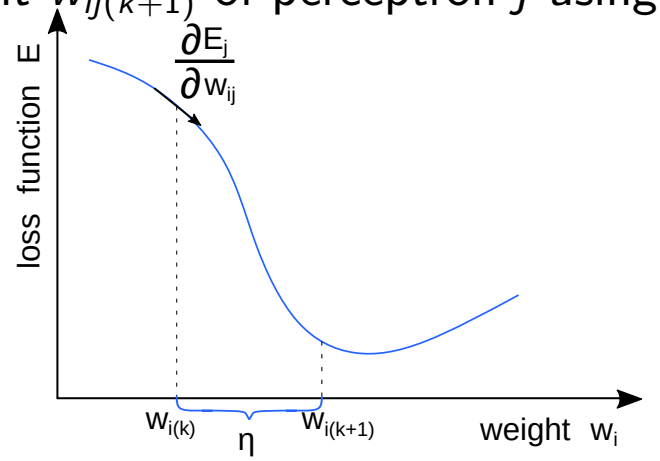
Now that we have a loss function $E_j = \frac{1}{2}e_j^2 = \frac{1}{2}(\hat{y}_j - y_j)^2$, we can use it to determine how to update w_{ij} to reduce the error.

The equation

$$w_{ij(k+1)} = w_{ij(k)} - \eta \frac{\partial E_j}{\partial w_{ij}}$$

is an expression for finding an updated weight $w_{ij(k+1)}$ of perceptron j using

- the current weight $w_{ij(k)}$
- a step size (**learning rate**) η
- the derivative of the loss function E_j with respect to w_{ij}, $\frac{\partial E_j}{\partial w_{ij}}$



This is the equation of **gradient descent**, an iterative method of finding the minimum of a function.

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (II)

Given an initial starting point $w_{ij(k)}$ we want to move in a **direction of steepest descent** (derivative of the loss function scaled by some amount η) to arrive at some new point $w_{ij(k+1)}$ which corresponds to the point $w_{ij(k)}$ either reduced or increased by the quantity $\eta \frac{\partial E_j}{\partial w_{ij}}$.

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (II)

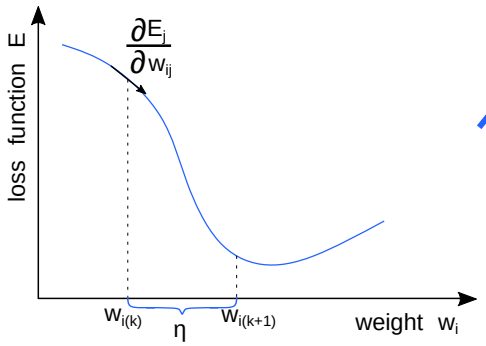
Given an initial starting point $w_{ij(k)}$ we want to move in a **direction of steepest descent** (derivative of the loss function scaled by some amount η) to arrive at some new point $w_{ij(k+1)}$ which corresponds to the point $w_{ij(k)}$ either reduced or increased by the quantity $\eta \frac{\partial E_j}{\partial w_{ij}}$.

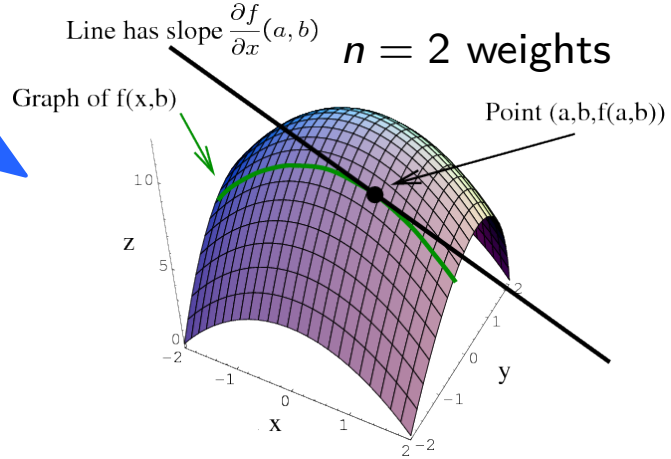
Why the Partial Derivative?

3. Gradient Descent - Updating the Weights to Reduce Error (II)

Given an initial starting point $w_{ij(k)}$ we want to move in a **direction of steepest descent** (derivative of the loss function scaled by some amount η) to arrive at some new point $w_{ij(k+1)}$ which corresponds to the point $w_{ij(k)}$ either reduced or increased by the quantity $\eta \frac{\partial E_j}{\partial w_{ij}}$.

Why the Partial Derivative?





graph of the error as a function of a **single** weight
more realistic cases: multiple weights
generally: for n weights the loss function is a n-dimensional hypersurface in the $n+1$ dimensional space and its derivative is a n-dimensional gradient vector

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (II)

Gradient Descent - given an initial starting point $w_{ij(k)}$ we want to move in a direction of steepest descent which can be calculated by the derivative of the loss function scaled by some amount η - thus arriving to some new point $w_{ij(k+1)}$ which corresponds to the point $w_{ij(k)}$ either reduced or increased by some amount that is equivalent to the quantity $\eta \frac{\partial E_j}{\partial w_{ij}}$.

We continue with an **expression for an updated weight** $w_{ij(k+1)}$:

$$w_{ij(k+1)} = w_{ij(k)} - \eta \frac{\partial E_j}{\partial w_{ij}}$$

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (II)

Gradient Descent - given an initial starting point $w_{ij(k)}$ we want to move in a direction of steepest descent which can be calculated by the derivative of the loss function scaled by some amount η - thus arriving to some new point $w_{ij(k+1)}$ which corresponds to the point $w_{ij(k)}$ either reduced or increased by some amount that is equivalent to the quantity $\eta \frac{\partial E_j}{\partial w_{ij}}$.

We continue with an **expression for an updated weight** $w_{ij(k+1)}$:

$w_{ij(k+1)} = w_{ij(k)} - \eta \frac{\partial E_j}{\partial w_{ij}}$

From this we can derive the quantity Δw_{ij} by which we need to change the current weight during the next iteration in order to further reduce E_j:

$$\begin{aligned} \Delta w_{ij} &= w_{ij(k+1)} - w_{ij(k)} \\ &= -\eta \frac{\partial E_j}{\partial w_{ij}} \end{aligned}$$

Δw_{ij} is the change in the current weight that defines the step size and direction to move from the current weight to the updated weight.
This is an application of the gradient descent in the context of the perceptron to modify the weights to reduce the value of the loss function.

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (III)

To calculate this derivative we have to acknowledge that the loss function E_j is a composite function:

$E_j = \frac{1}{2} e_j^2 \Rightarrow E_j$ is a function of e_j

$e_j = (\hat{y}_j - y_j)^2 \Rightarrow e_j$ is a function of y_j

$y_j = \frac{1}{\varphi(A_j)} \Rightarrow y_j$ is a function of A_j

$A_j = \sum_{i=1}^{n} w_{ij} x_i + b \Rightarrow A_j$ is a function of w_{ij}

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (III)

To calculate this derivative we have to acknowledge that the loss function E_j is a composite function:

$$\left.\begin{aligned} &E_j = \tfrac{1}{2} e_j^2 \Rightarrow E_j \text{ is a function of } e_j \\ &e_j = (\hat{y}_j - y_j)^2 \Rightarrow e_j \text{ is a function of } y_j \\ &y_j = \tfrac{1}{1+\exp(-A_j)} \Rightarrow y_j \text{ is a function of } A_j \\ &A_j = \sum_{i=1}^{n} w_{ij} x_i + b \Rightarrow A_j \text{ is a function of } w_{ij} \end{aligned}\right\} \Rightarrow \text{use the chain rule:}$$

$$\frac{\partial E_j}{\partial w_{ij}} = \frac{\partial E_j}{\partial e_j} \times \frac{\partial e_j}{\partial y_j} \times \frac{\partial y_j}{\partial A_j} \times \frac{\partial A_j}{\partial w_{ij}}$$

After computing each of the four partial derivative terms, we can plug them back into the chain rule equation for the derivative of E_j and get the following expression for calculating the gradient vector of the loss function:

$$\frac{\partial E_j}{\partial w_{ij}} = -e_j(1 - y_j)y_j x_i$$

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (IV)

From the derivative we can derive the perceptron δ function which quantifies the change that needs to be added to or subtracted from the current weight in order to arrive at the updated weight that further reduces the error.
Let $-e_j[1 - y_j]y_j = \delta_j$, then

$$\frac{\partial E_j}{\partial w_{ij}} = \delta_j x_i$$

With plugging this back into the expression for Δw_{ij} we get the **Perceptron δ Function**:

$$\Delta w_{ij} = \eta \delta_j x_i$$

The perceptron δ function quantifies by how much a current weight $w_{ij(k)}$ needs to be changed in order to obtain the updated weight $w_{ij(k+1)}$ which will contribute to the further reduction of the error.

Backpropagation - Training the Perceptron

3. Gradient Descent - Updating the Weights to Reduce Error (IV)

From the derivative we can derive the perceptron δ function which quantifies the change that needs to be added to or subtracted from the current weight in order to arrive at the updated weight that further reduces the error.
Let $-e_j[1 - y_j]y_j = \delta_j$, then

$$\frac{\partial E_j}{\partial w_{ij}} = \delta_j x_i$$

With plugging this back into the expression for Δw_{ij} we get the **Perceptron δ Function**:

$$\Delta w_{ij} = \eta \delta_j x_i$$

The perceptron δ function quantifies by how much a current weight $w_{ij(k)}$ needs to be changed in order to obtain the updated weight $w_{ij(k+1)}$ which will contribute to the further reduction of the error.

⇨ Iteratively repeating this process until the error is minimized is what happens when training the perceptron.

Backpropagation - Training the Perceptron

Why Gradient Descent?

Why not simply minimize the loss function by solving the problem analytically (setting its derivative to zero)?

Backpropagation - Training the Perceptron

Why Gradient Descent?

Why not simply minimize the loss function by solving the problem analytically (setting its derivative to zero)?

With increasing number of weights, the error function quickly becomes very complex: Each additional weight adds another dimension to this graph. In addition, we might want to minimize the error function not only for a single input data but for an entire training data set.
Such functions typically:

- have many minima
- are too complex to be expressed mathematically in a neat equation
- so complex that calculating the derivative of such complex functions may not even be feasible with a neat mathematical equation

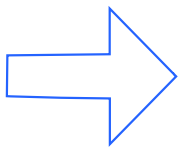
Backpropagation - Training the Perceptron

Why Gradient Descent?

Why not simply minimize the loss function by solving the problem analytically (setting its derivative to zero)?

With increasing number of weights, the error function quickly becomes very complex: Each additional weight adds another dimension to this graph. In addition, we might want to minimize the error function not only for a single input data but for an entire training data set.
Such functions typically:

- have many minima
- are too complex to be expressed mathematically in a neat equation
- so complex that calculating the derivative of such complex functions may not even be feasible with a neat mathematical equation



finding the arguments (the optimal weights) that correspond to the global minima is not realistic

instead: use an iterative optimizer, such as Gradient Descent

Backpropagation - Training the Perceptron

summary so far:

Training a perceptron with backpropagation is an **optimization problem** which involves iteratively updating the weights to minimize a loss function.

Backpropagation - Training the Perceptron

summary so far:

Training a perceptron with backpropagation is an **optimization problem** which involves iteratively updating the weights to minimize a loss function.

Key Idea of Learning in Neural Networks:

find new weights using a loss function E and the previous output, $\mathbf{y}$:

$$\mathbf{w}^* = \arg\min_w \sum_{n=1}^{N} E(\mathbf{y}^{(n)}, \hat{\mathbf{y}}^{(n)})$$

define a loss function E, e.g. the squared loss:

$$\sum_k \frac{1}{2}(\hat{\mathbf{y}}_k^{(n)} - y_k^{(n)})^2$$

minimize the loss function by using the gradient descent:

$$w_{ij(k+1)} = w_{ij(k)} - \eta \frac{\partial E_j}{\partial w_{ij}}$$

This is carried out iteratively until additional observations (enhancing the training set) or further iterations are not reducing the error rate.

Activation Functions

Activation functions play a key role in neural networks, so it is **essential** to understand the advantages and disadvantages of different choices to achieve better performance.

The **purpose** of an activation function is to add **non-linearity** to the neural network.

Activation Functions

Activation functions play a key role in neural networks, so it is **essential** to understand the advantages and disadvantages of different choices to achieve better performance.

The **purpose** of an activation function is to add **non-linearity** to the neural network.

What would happen to a neural network without activation functions?

Activation Functions

Activation functions play a key role in neural networks, so it is **essential** to understand the advantages and disadvantages of different choices to achieve better performance.

The **purpose** of an activation function is to add **non-linearity** to the neural network.

What would happen to a neural network without activation functions?

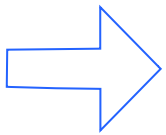
In that case, every neuron will only be performing a linear transformation on the inputs using the weights and biases. No matter how many hidden layers we add to the neural network; all layers will behave in the same way because the composition of two linear functions is a linear function itself.

Activation Functions

Activation functions play a key role in neural networks, so it is **essential** to understand the advantages and disadvantages of different choices to achieve better performance.

The **purpose** of an activation function is to add **non-linearity** to the neural network.

What would happen to a neural network without activation functions?

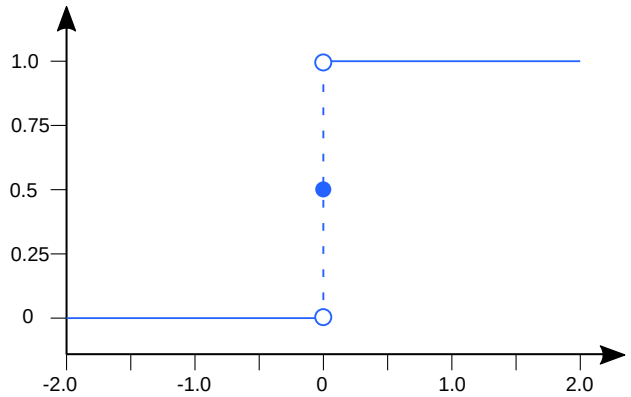


Despite the seemingly complexity, learning any complex task is impossible, and our model would be just a linear regression model.

Activation Functions

Binary Step Function

The binary step function depends on a threshold value that decides whether a neuron should be activated or not. If the input to the activation function is greater than the threshold, the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next layer.



$$\varphi(z) = \begin{cases} 0, & \text{if } z < 0. \\ 1, & \text{if } z \geq 0. \end{cases}$$

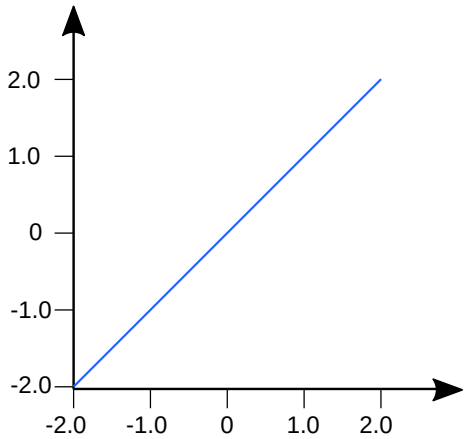
limitations:

- It cannot provide multi-value outputs, for example, it cannot be used for multi-class classification problems.
- The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

Activation Functions

Linear Activation Function

The linear activation function, also known as *no activation* or *identity function*, is where the activation is proportional to the input.



$$\varphi(z) = z$$

limitations:

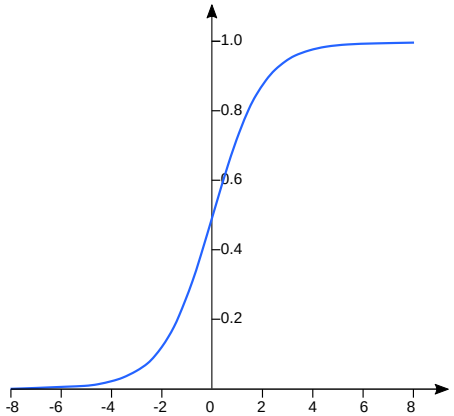
- Backpropagation cannot be applied as the derivative is a constant and thus has no relation to the input.
- A linear activation function turns the neural network into just one layer, no matter the actual number of layers in the neural network.

⇨ use **non-linear** activation functions

Activation Functions

Sigmoid / Logistic Activation Function

Historically popular: interpretation as a neuron's saturating 'firing rate'. This function takes any real value as input and outputs values in the range of 0 to 1. The more positive the input, the closer the output value will be to 1.0, whereas the more negative, the closer the output will be to 0.



$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

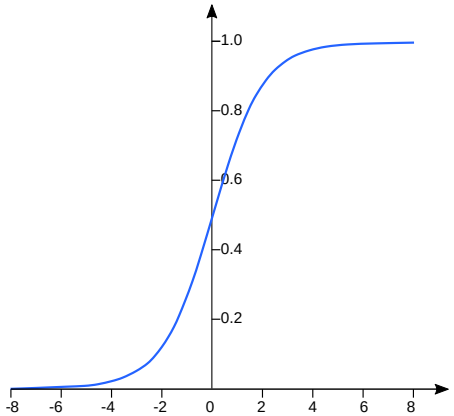
one of the most widely used activation functions:

- It is commonly used for models where we have to predict a probability. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values.

Activation Functions

Sigmoid / Logistic Activation Function

Historically popular: interpretation as a neuron's saturating 'firing rate'. This function takes any real value as input and outputs values in the range of 0 to 1. The more positive the input, the closer the output value will be to 1.0, whereas the more negative, the closer the output will be to 0.



$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

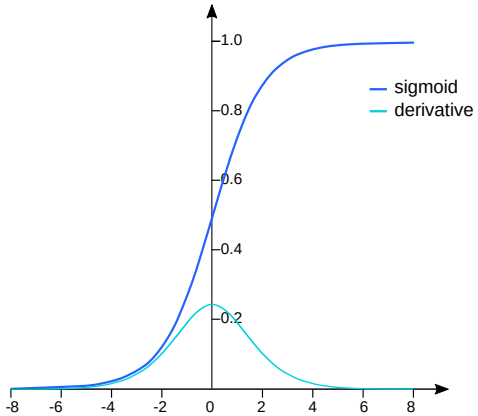
limitations:

- Sigmoid outputs are not zero-centered, so the output of all the neurons will be of the same sign. This makes the training of the neural network more difficult and unstable.
- Saturated neurons 'kill' the gradients

Activation Functions

Sigmoid / Logistic Activation Function

Historically popular: interpretation as a neuron's saturating 'firing rate'. This function takes any real value as input and outputs values in the range of 0 to 1. The more positive the input, the closer the output value will be to 1.0, whereas the more negative, the closer the output will be to 0.



$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

$$\varphi'(z) = \varphi(z)(1 - \varphi(z))$$

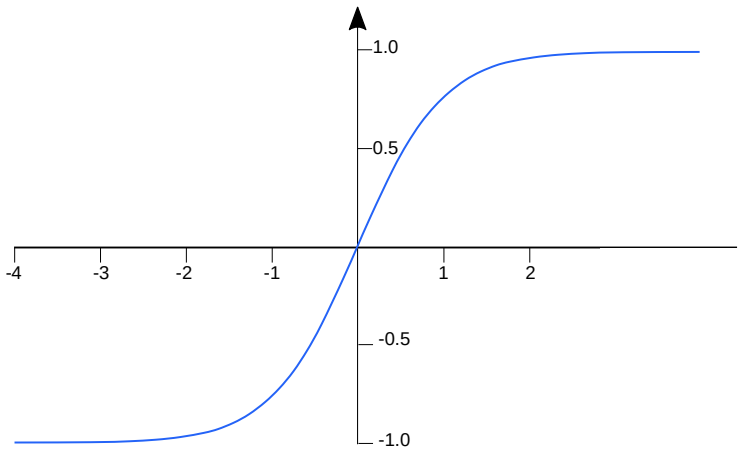
limitations:

- The derivative: The sigmoid's gradient values are only significant for range $\sim[-3, 3]$, and the graph gets much flatter in other regions. As the gradient value approaches zero, the network ceases to learn and suffers from the **Vanishing Gradient Problem**.

Activation Functions

Tanh Function (Hyperbolic Tangent)

The tanh function is very similar to the sigmoid/logistic activation function, but has a different output range of $[-1, 1]$.



$$\varphi(z) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

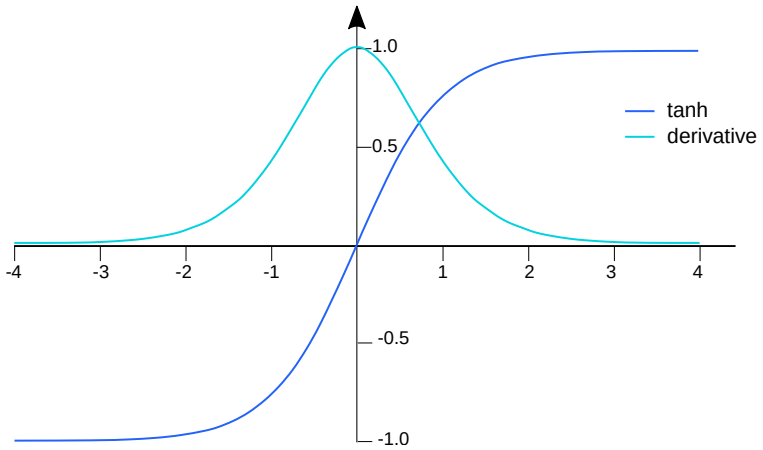
advantages:

- The output of the tanh activation function is 0-centered within $[-1, 1]$. This centers the data, making learning much easier for the next layer. Usually used in hidden layers.
- Other than for the sigmoid activation function no restriction of the gradient movement.

Activation Functions

Tanh Function (Hyperbolic Tangent)

The tanh function is very similar to the sigmoid/logistic activation function, but has a different output range of $[-1, 1]$.



$$\varphi(z) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

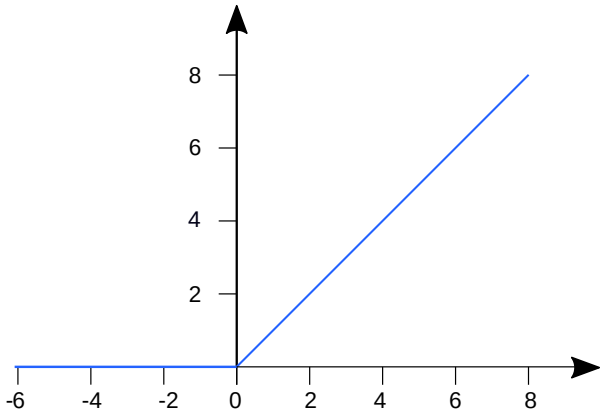
$$\varphi'(z) = 1 - \varphi^2(z)$$

limitations:

- vanishing gradients similar to the sigmoid activation function
- In addition, the gradient of the tanh function is much steeper than the gradient of the sigmoid function.

ReLU Function

ReLU stands for Rectified Linear Unit. Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.



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

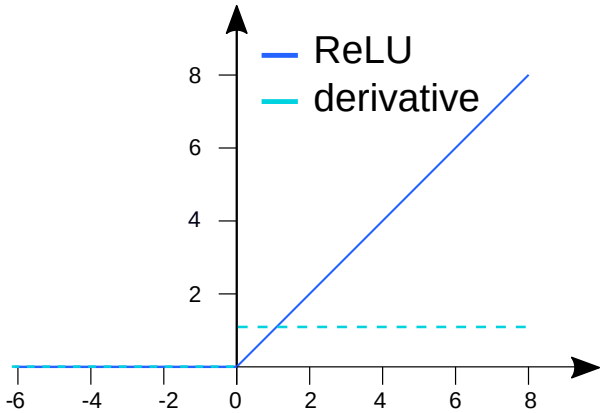
advantages:

- The neurons will be deactivated if the output of the linear part is < 0, thus not all neurons are active at the same time $\Rightarrow$ far more computationally efficient than sigmoid and tanh.
- ReLU activation converges much faster than sigmoid/tanh ($\sim \times 6$).
- ReLU accelerates the convergence of gradient descent towards the loss function's minimum due to its linear, non-saturating property.

Activation Functions

ReLU Function

ReLU stands for Rectified Linear Unit. Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.



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

$$\varphi'(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

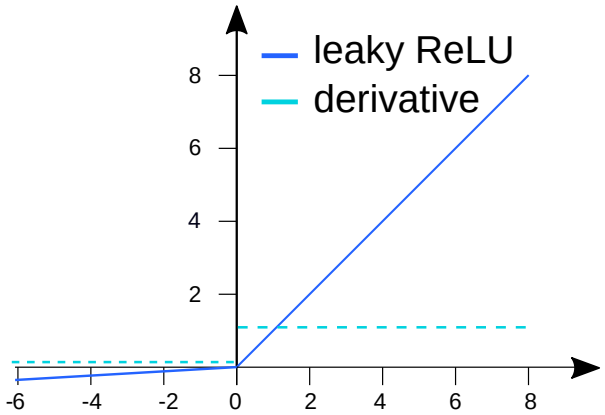
disadvantage: The **Dying ReLU Problem**

The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.
All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

Activation Functions

Leaky ReLU Function

Leaky ReLU is an improved version of ReLU function to solve the Dying ReLU problem as it has a small positive slope $\alpha = \text{const}$ (typically $\alpha = 0.01$) in the negative area.
By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value.



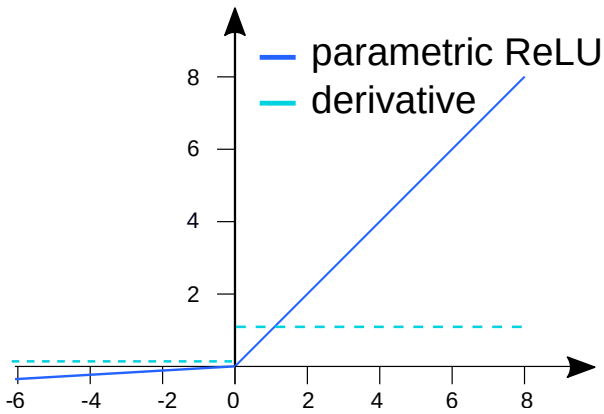
$$\varphi(z) = \max(\alpha z, z)$$

$$\varphi'(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ \alpha, & \text{otherwise.} \end{cases}$$

advantages:
The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.

Parametric ReLU Function

Parametric ReLU is another variant of ReLU that aims to solve the problem of gradient's becoming zero for the left half of the axis. The function looks like the Leaky ReLU, but the slope α is not longer a constant, but a variable. By performing backpropagation, the most appropriate value of α is learnt.



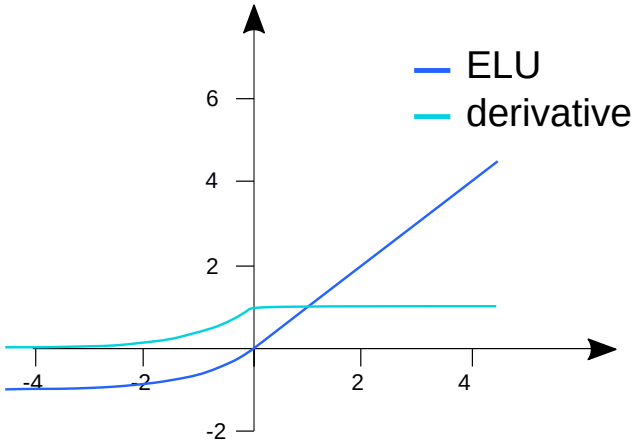
$$\varphi(z) = \max(\alpha z, z)$$

$$\varphi'(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ \alpha, & \text{otherwise.} \end{cases}$$

advantage:
The parametric ReLU function is used when the leaky ReLU function still fails at solving the problem of dead neurons, and the relevant information is not successfully passed to the next layer.

Exponential Linear Units (ELUs) Function

Exponential Linear Unit, or ELU for short, is also a variant of ReLU that modifies the slope of the negative part of the function. ELU uses a log curve to define the negativ values unlike the leaky ReLU and Parametric ReLU functions with a straight line.



$$\varphi(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha(e^z - 1), & \text{otherwise.} \end{cases}$$

$$\varphi'(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ \alpha e^z, & \text{otherwise.} \end{cases}$$

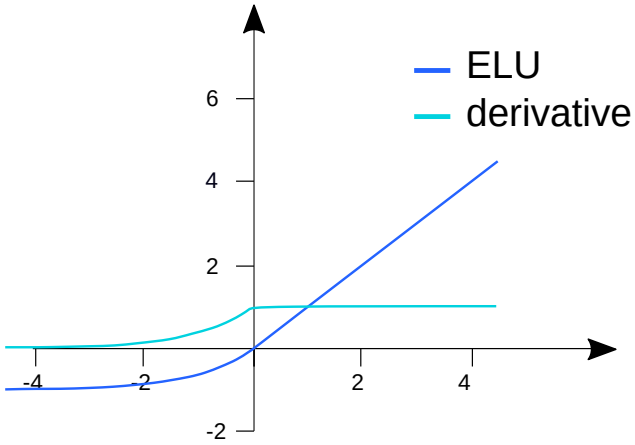
advantages:

- ELU becomes smooth slowly until its output equal to $-\alpha$ whereas RELU sharply smoothes.
- Avoids dead ReLU problem by introducing log curve for negative input values. It helps the network nudge weights and biases in the right direction.

Activation Functions

Exponential Linear Units (ELUs) Function

Exponential Linear Unit, or ELU for short, is also a variant of ReLU that modifies the slope of the negative part of the function. ELU uses a log curve to define the negativ values unlike the leaky ReLU and Parametric ReLU functions with a straight line.



$$\varphi(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha(e^z - 1), & \text{otherwise.} \end{cases}$$

$$\varphi'(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ \alpha e^z, & \text{otherwise.} \end{cases}$$

limitations:

- It increases the computational time because of the exponential operation included.
- No learning of an α parameter.
- exploding gradient problem

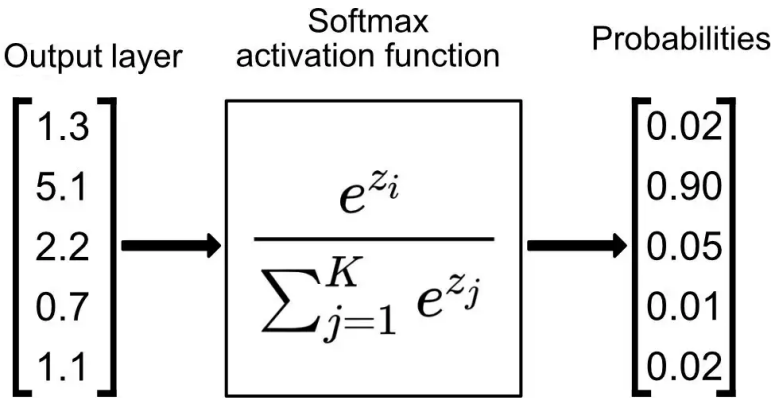
Activation Functions

Softmax Activation Function

Softmax is different from the rest of the activation function as it calculates the probabilities distribution of the event over different events, i.e. the probabilities of each target class over all possible target classes.
Softmax takes as input a vector $\mathbf{z}$ of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers.

$\Rightarrow$ After applying softmax, each component will be in $(0, 1)$, the components will add up to 1, and they can be interpreted as probabilities.

example:



1. Exponentiate every element of the output layer and sum the results (around 181.73 in this exemple).
2. Each element of the output layer is exponentiated and divided by the sum obtained in step 1 (exp(1.3) / 181.37 = 3.67 / 181.37 = 0.02)

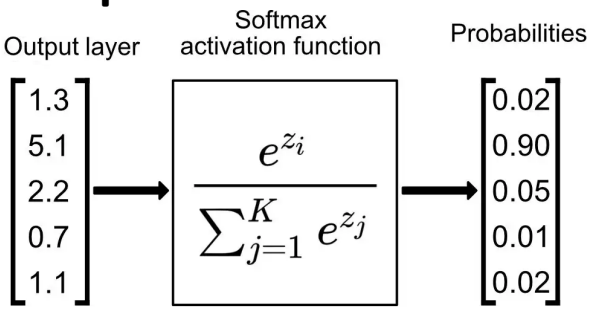
Activation Functions

Softmax Activation Function

Softmax is different from the rest of the activation function as it calculates the probabilities distribution of the event over different events, i.e. the probabilities of each target class over all possible target classes.
Softmax takes as input a vector $\mathbf{z}$ of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers.

$\Rightarrow$ After applying softmax, each component will be in $(0, 1)$, the components will add up to 1, and they can be interpreted as probabilities.

example:

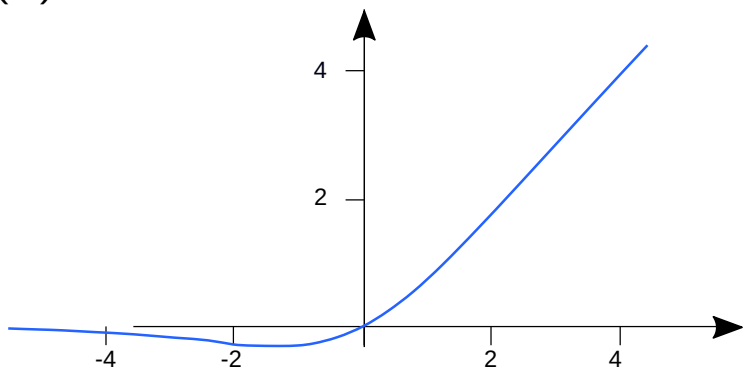


usage: Softmax is used for multi-classification, whereas Sigmoid is used for binary classification.

Activation Functions

Swish

Swish (developed by researchers at Google) matches or outperforms ReLU activation function on deep networks applied to various challenging domains such as image classification, machine translation etc. Swish is bounded below but unbounded above, i.e. $\varphi(z) \to \text{const}$ for $z \to -\infty$ but $\varphi(z) \to \infty$ for $z \to \infty$.



$$\varphi(z) = z \times \text{sigmoid}(z)$$
$$= \frac{z}{1 + \exp(-z)}$$

advantages over ReLU:

- Other than ReLU, Swish is a smooth function.
- Small negative values were zeroed out in ReLU activation function. However, those negative values may still be relevant for capturing patterns underlying the data.
- The swish function being non-monotonous enhances learning.

Activation Functions

How to **choose** the right Activation Function?

Activation Functions

How to **choose** the right Activation Function?

Some guidelines:

- ReLU activation function should only be used in the hidden layers.
- Sigmoid and Tanh functions should not be used in hidden layers as they make the model more susceptible to problems during training (due to vanishing gradients).
- Swish function is used in neural networks having more than 40 layers.

Activation Functions

How to **choose** the right Activation Function?

Some guidelines:

- ReLU activation function should only be used in the hidden layers.
- Sigmoid and Tanh functions should not be used in hidden layers as they make the model more susceptible to problems during training (due to vanishing gradients).
- Swish function is used in neural networks having more than 40 layers.

The activation function used in **hidden layers** is typically chosen based on the type of neural network architecture.

- Convolutional Neural Network (CNN)*: ReLU activation function.
- Recurrent Neural Network*: Tanh and/or Sigmoid activation function.

* more on this later on

Activation Functions

How to **choose** the right Activation Function?

You need to match your activation function for your **output layer** based on the type of prediction problem that you are solving - specifically, the type of predicted:

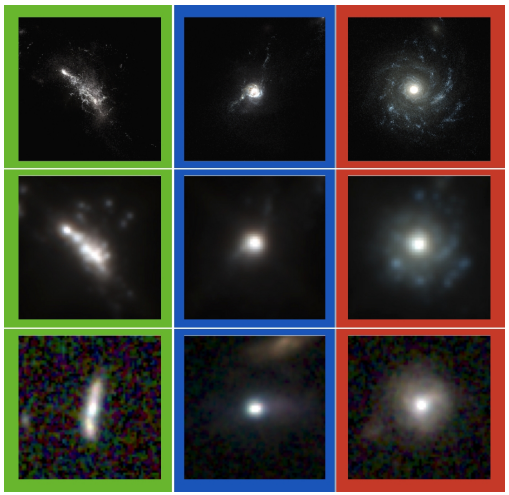
- Regression: Linear Activation Function
- Binary Classification: Sigmoid/Logistic Activation Function
- Multiclass Classification: Softmax
- Multilabel Classification: Sigmoid

An Outlook: Learning Paradigms & Applications

Supervised learning uses a training set of paired inputs and desired outputs. The learning task is to produce the desired output for each input.

In this case the loss function is related to eliminating incorrect deductions. A commonly used loss function is the mean-squared error, which tries to minimize the average squared error between the network's output and the desired output by providing continuous feedback on the quality of solutions.

Tasks suited for supervised learning with neural networks are pattern recognition (classification) and regression (function approximation), also applicable to sequential data (e.g., speech and gesture recognition).

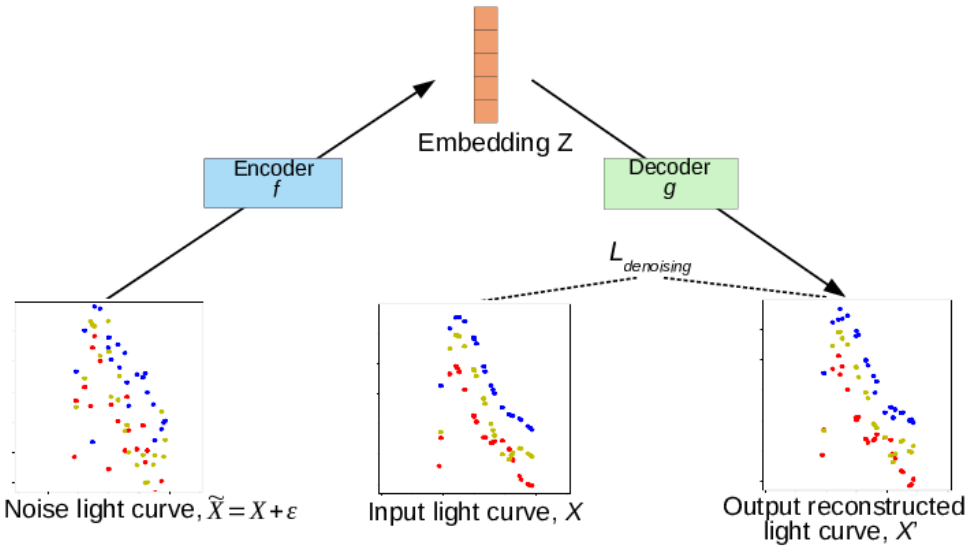


Simulations by Daniel Ceverino and Joel Primack; simulated images by Greg Snyder and Marc Huertas-Company; Hubble Space Telescope CANDELS.

Learning Paradigms

Unsupervised learning uses input data along with a loss function which depends on the task (the **model domain**) and prior assumptions (the implicit properties of the model, its parameters and the observed variables). From this, unsupervised learning tries to find hidden pattern in the data.

Tasks that fall within the paradigm of unsupervised learning are in general estimation problems; the applications include clustering, the estimation of statistical distributions, compression and filtering.

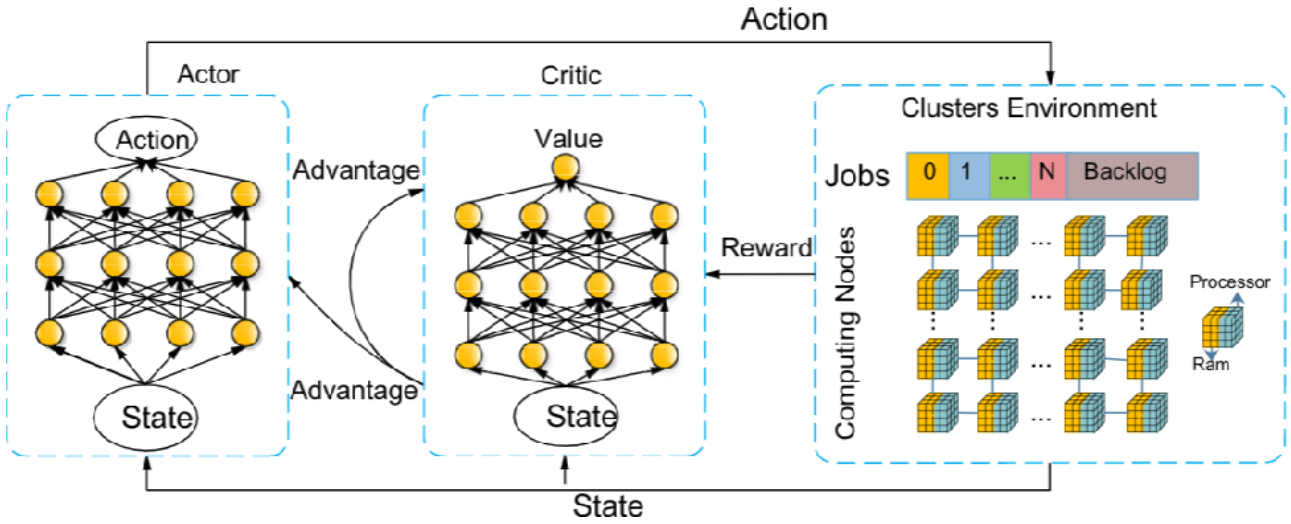


Denoising light curves by applying an autoencoder. Pasquet et al. (2019).

Learning Paradigms

Reinforcement learning aims to weight the network (devise a policy) to perform actions by an agent that minimize long-term (expected cumulative) loss by an environment. At any juncture, the agent decides whether to explore new actions to uncover their loss or to exploit prior learning to proceed more quickly.

Applications of reinforcement learning are such as robotics, observational schedules for telescopes, job scheduling for data centers.



Job Scheduling on Data Centers with Deep Reinforcement Learning. Liang & Yang (2019).