ARTIFICIAL INTELLIGENCE 2

Multilayer Perceptrons

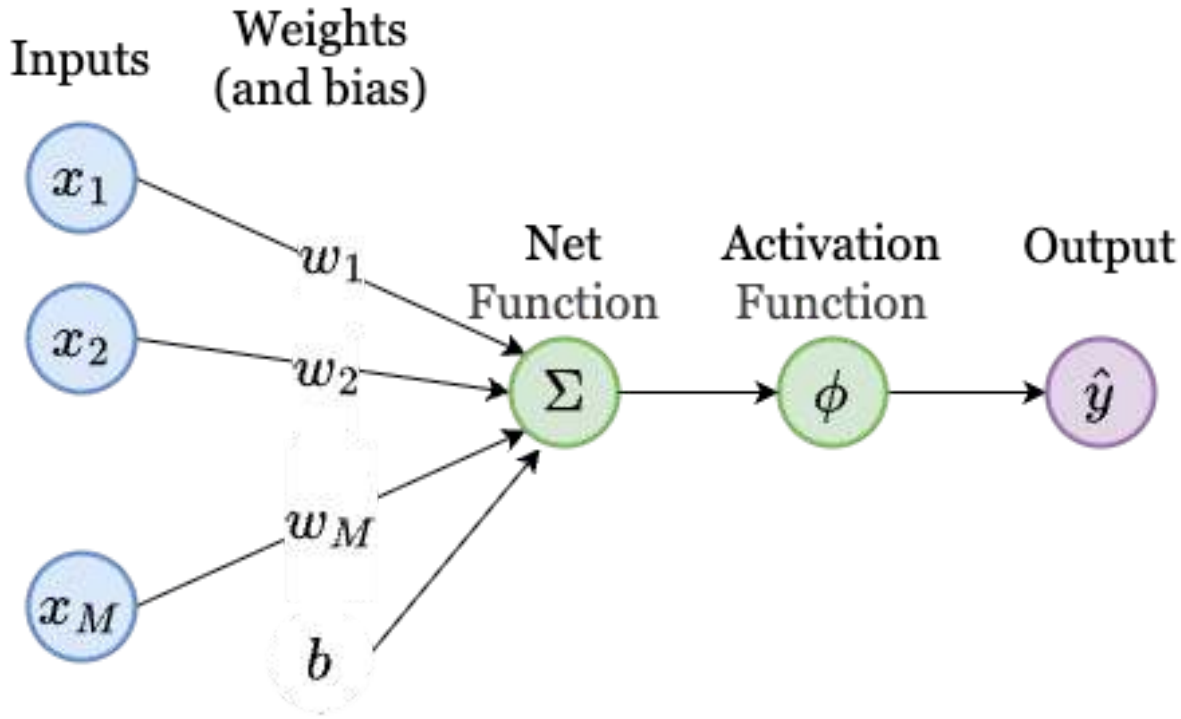
Francesco Spinnato

* francesco.spinnato@unipi.it
University of Pisa

Perceptron

The limitations of the linear perceptron are:

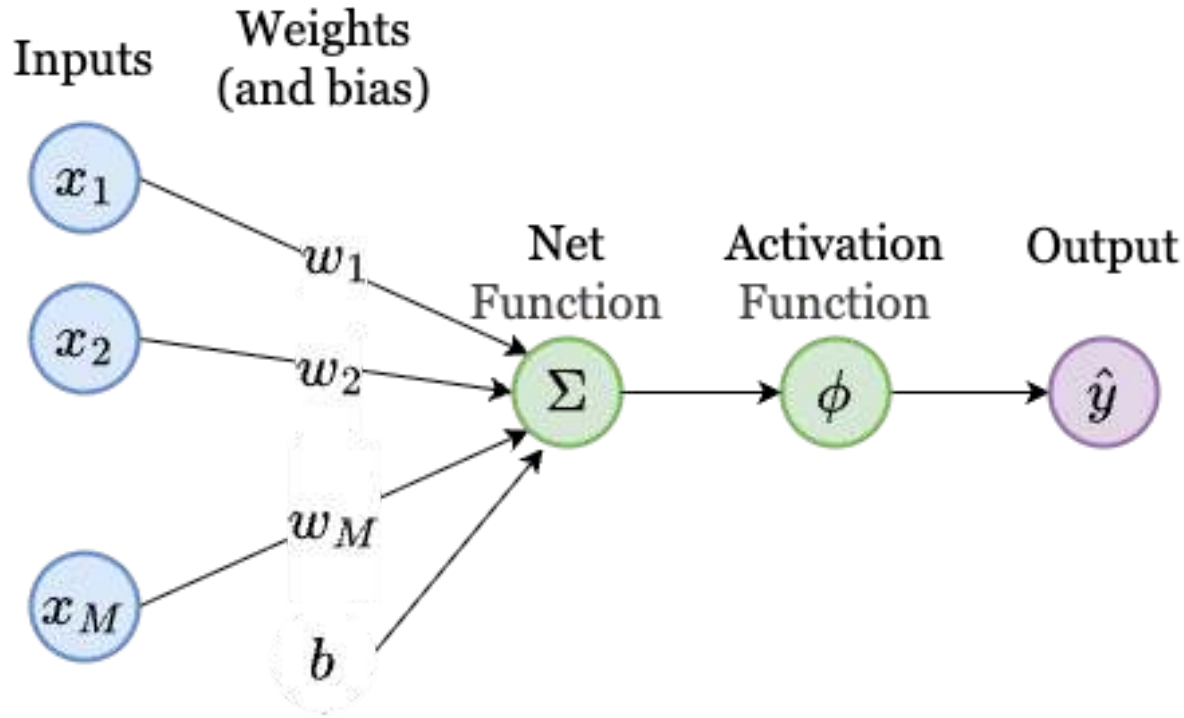
- It can only learn **linearly separable** decision boundaries;
- It does not converge if the data are not linearly separable;
- In its basic form, it performs only **binary classification**.



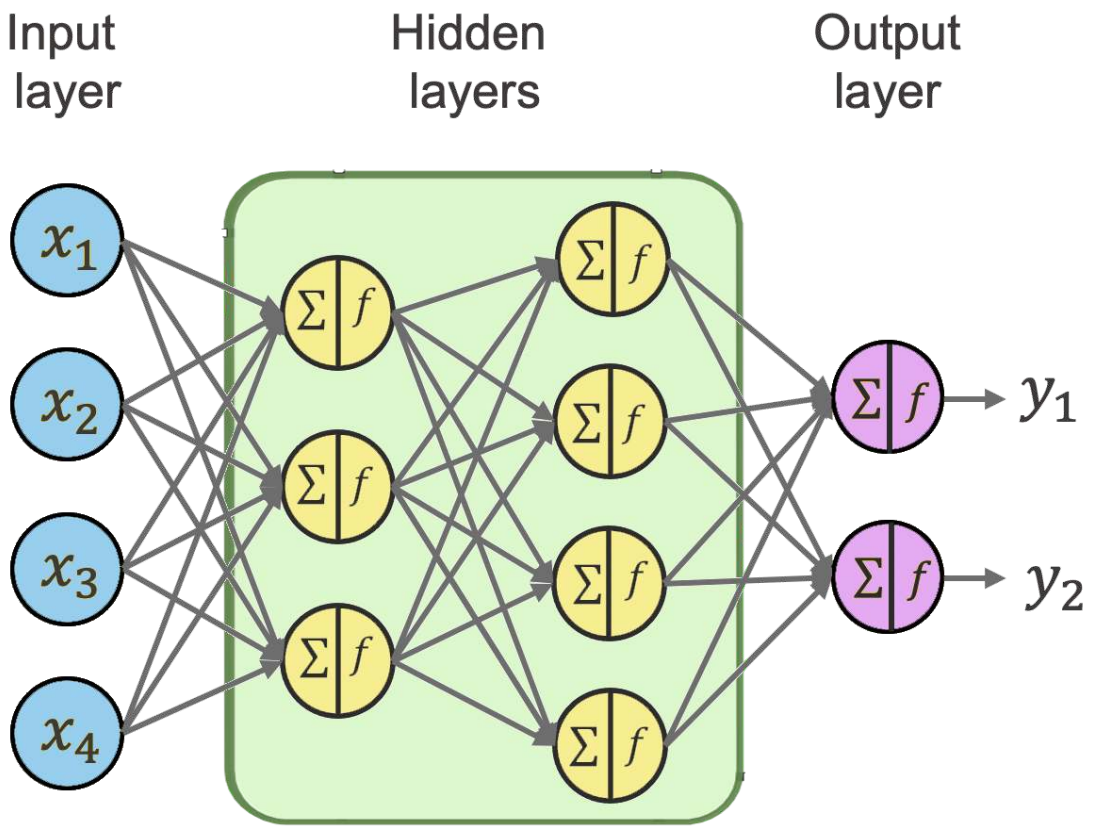
Perceptron

We want:

- A **more robust learning rule**;
- Combination of multiple neurons and layers to **learn more complex, non-linear, decision boundaries**;
- Ability to solve multiclass classification and multivariate regression.



Multilayer Perceptron (MLP)



The expressiveness of the perceptron can be increased by:

- adding intermediate **hidden** layers between the input and output;
- having **multiple non-linear** activation functions;
- allowing for **multiple outputs**.

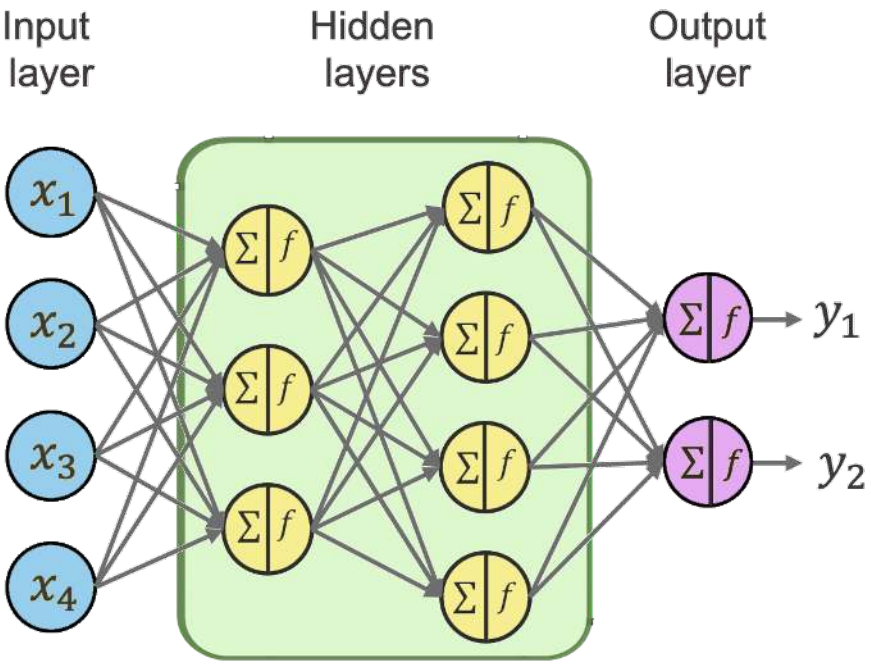
Without nonlinear activation functions, stacking multiple layers would still produce a single linear transformation!

Multilayer Perceptron

We can compute the forward pass of an MLP using matrix multiplication. Each layer computes a weighted sum of its inputs followed by a nonlinearity.
For a dataset $\mathbf{X} \in \mathbb{R}^{n \times 4}$, with n samples and 4 features:

First hidden layer:

$$\mathbf{H}_1 = f(\mathbf{X}\mathbf{W}_1) \in \mathbb{R}^{n \times 3}, \quad \mathbf{W}_1 \in \mathbb{R}^{4 \times 3}$$



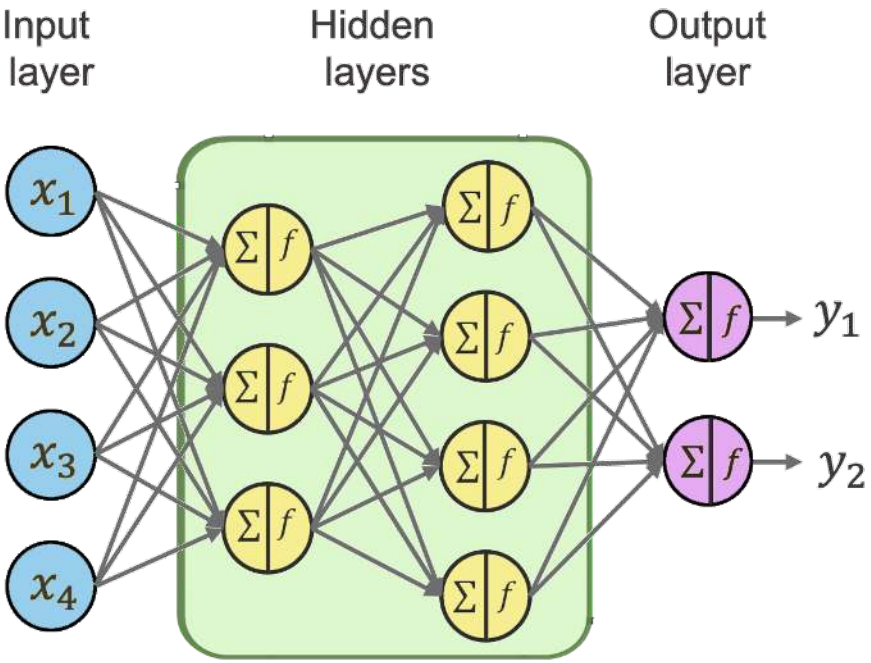
From now on, we omit bias vectors for simplicity

Multilayer Perceptron

We can compute the forward pass of an MLP using matrix multiplication. Each layer computes a weighted sum of its inputs followed by a nonlinearity.
For a dataset $\mathbf{X} \in \mathbb{R}^{n \times 4}$, with n samples and 4 features:

Second hidden layer:

$$\mathbf{H}_2 = f(\mathbf{H}_1 \mathbf{W}_2) \in \mathbb{R}^{n \times 4}, \quad \mathbf{W}_2 \in \mathbb{R}^{3 \times 4}$$

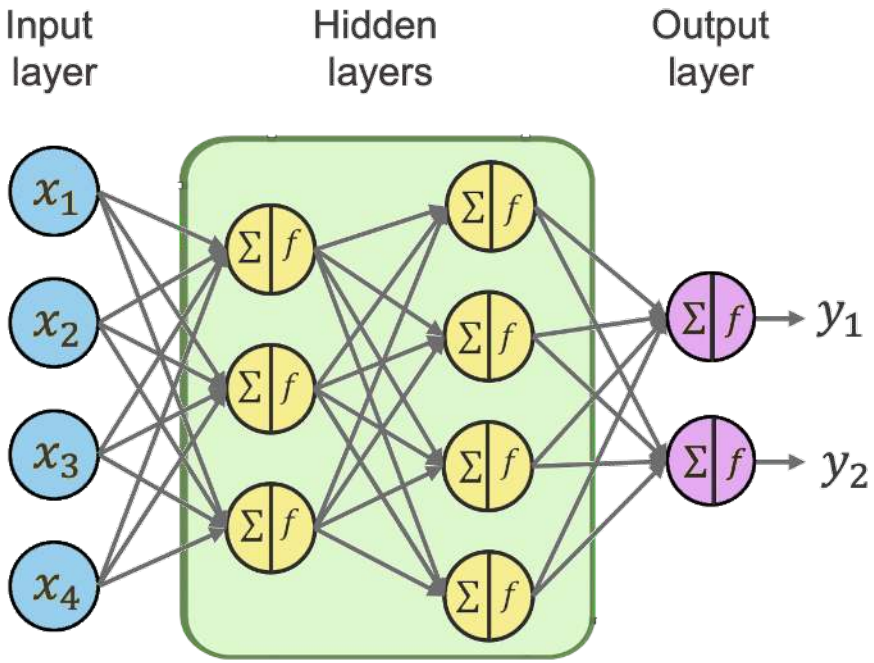


Multilayer Perceptron

We can compute the forward pass of an MLP using matrix multiplication. Each layer computes a weighted sum of its inputs followed by a nonlinearity.
For a dataset $\mathbf{X} \in \mathbb{R}^{n \times 4}$, with n samples and 4 features:

Output layer:

$$\hat{\mathbf{Y}} = f(\mathbf{H}_2\mathbf{W}_3) \in \mathbb{R}^{n \times 2}, \quad \mathbf{W}_3 \in \mathbb{R}^{4 \times 2}$$

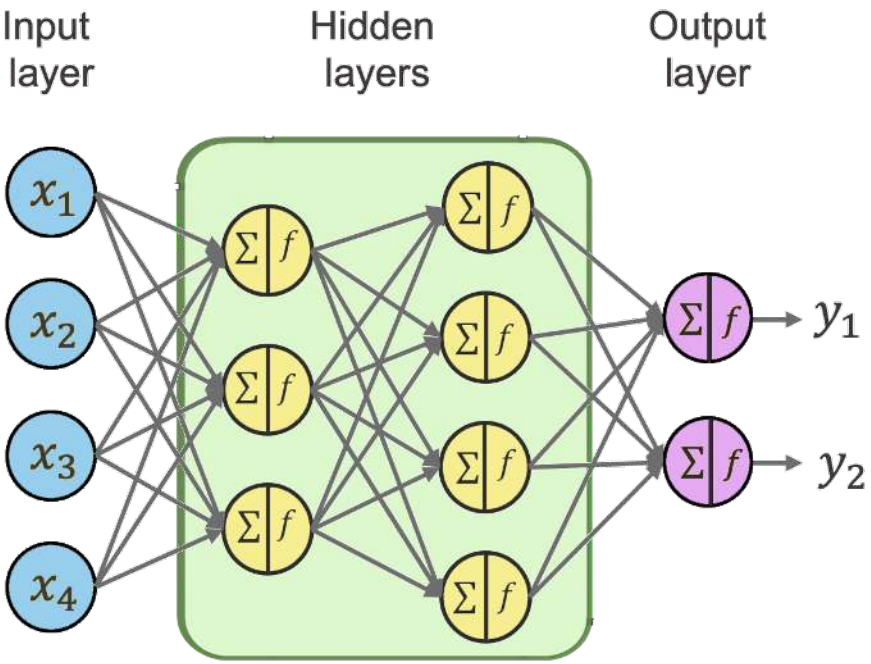


Multilayer Perceptron

We can compute the forward pass of an MLP using matrix multiplication. Each layer computes a weighted sum of its inputs followed by a nonlinearity.
For a dataset $\mathbf{X} \in \mathbb{R}^{n \times 4}$, with n samples and 4 features:

So it's like nesting operations between layers:

$$\hat{\mathbf{Y}} = f(f(f(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3)$$



Multilayer Perceptron

Intuitively, each layer:

- takes the representation produced by the previous layer,
- computes a weighted sum of its inputs via matrix multiplication (i.e., a linear transformation),
- and then applies a non-linear function.

As we go deeper into the network, the data is transformed into increasingly abstract representations, until the final layer produces the output.

$$\hat{\mathbf{Y}} = f(f(\cdots f(f(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3 \cdots)\mathbf{W}_h)$$

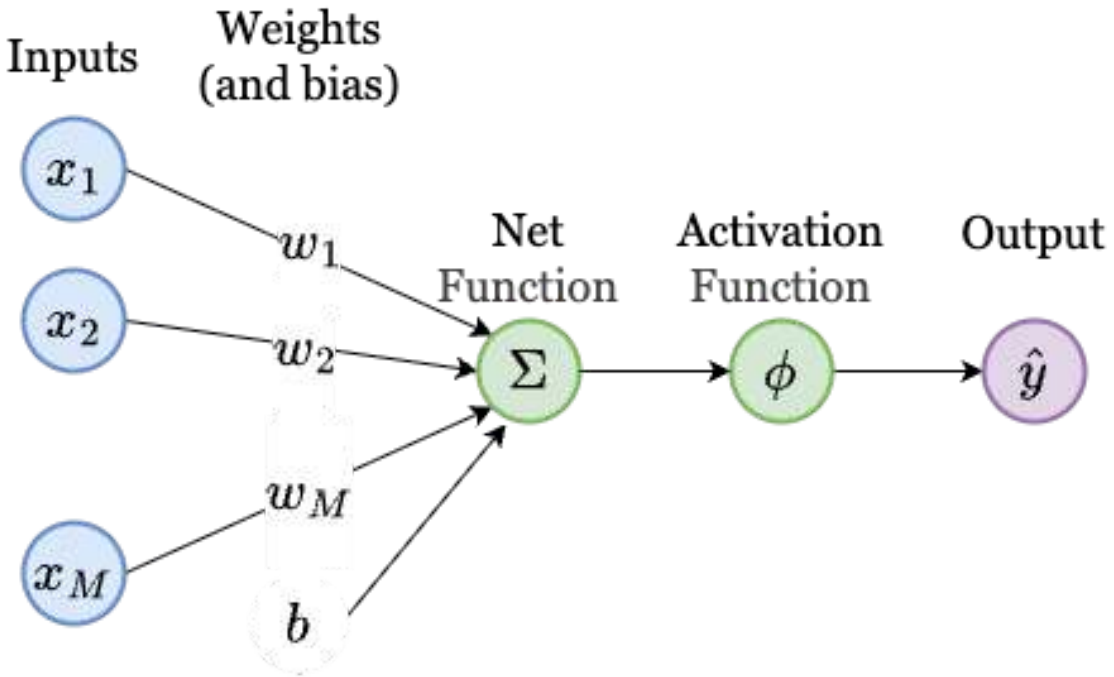
Note that activation functions can be different from layer to layer!
For example, the output layer may use a different activation (e.g., softmax or identity) than the hidden layers.

Multilayer Perceptrons

Activation Functions

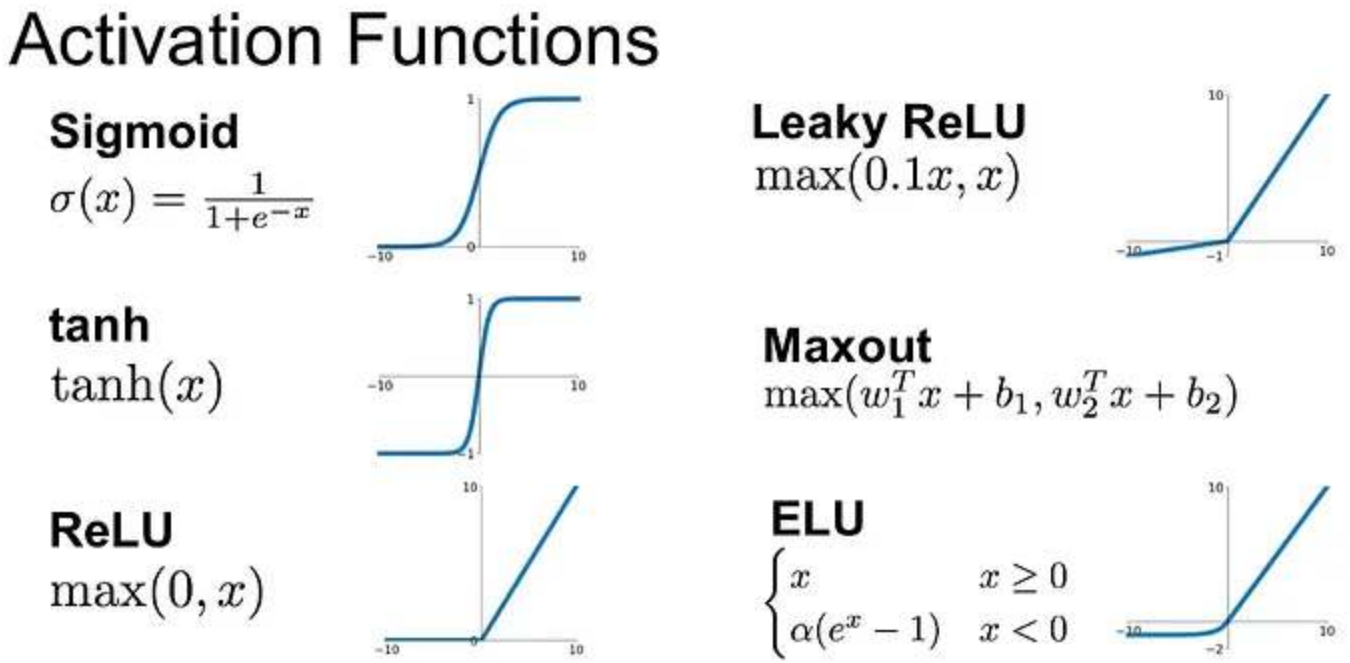
Activation Functions

The linear perceptron uses the $\phi = \text{sign}$ activation function...



Activation Functions

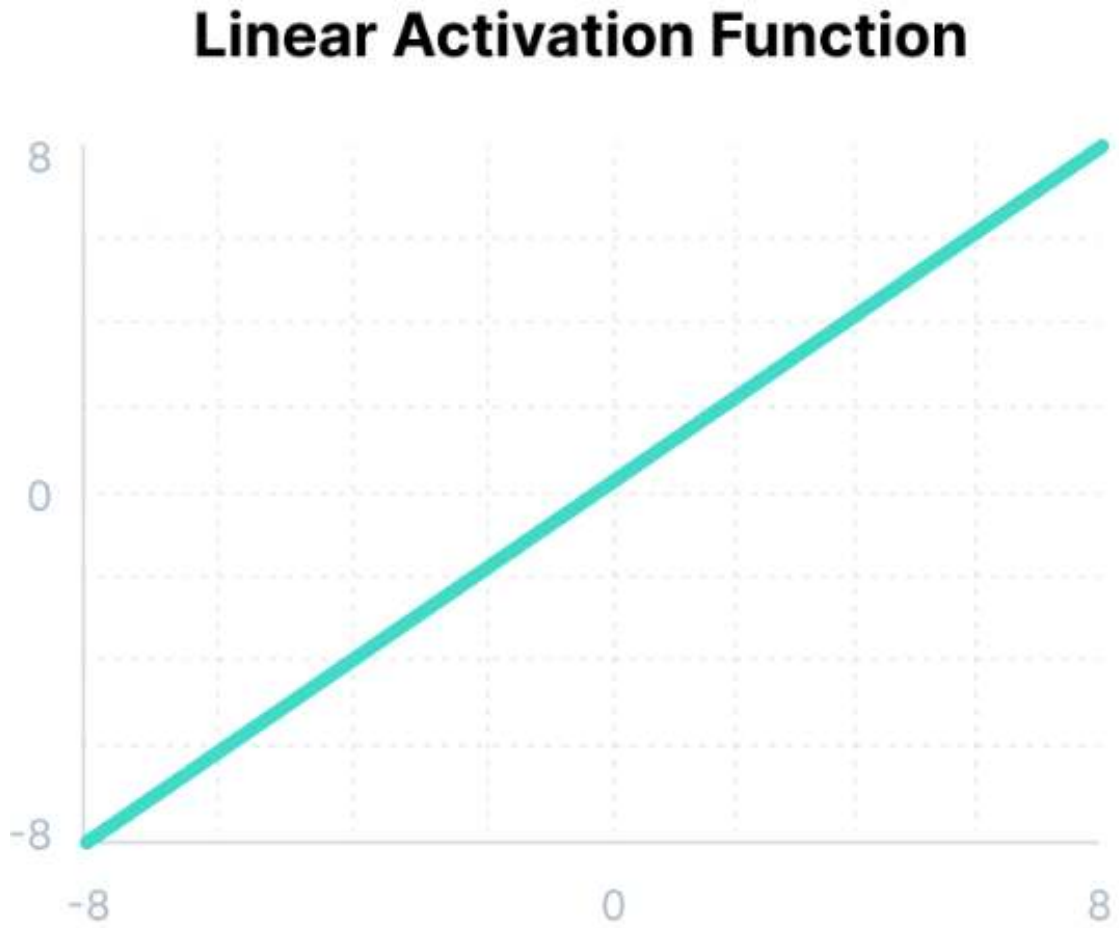
... however there are several alternatives that can be applied for each layer!



Linear

For regression problems, using no activation function (i.e., the identity function) often works well.

$$f(x) = x$$



Sigmoid / Logistic

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

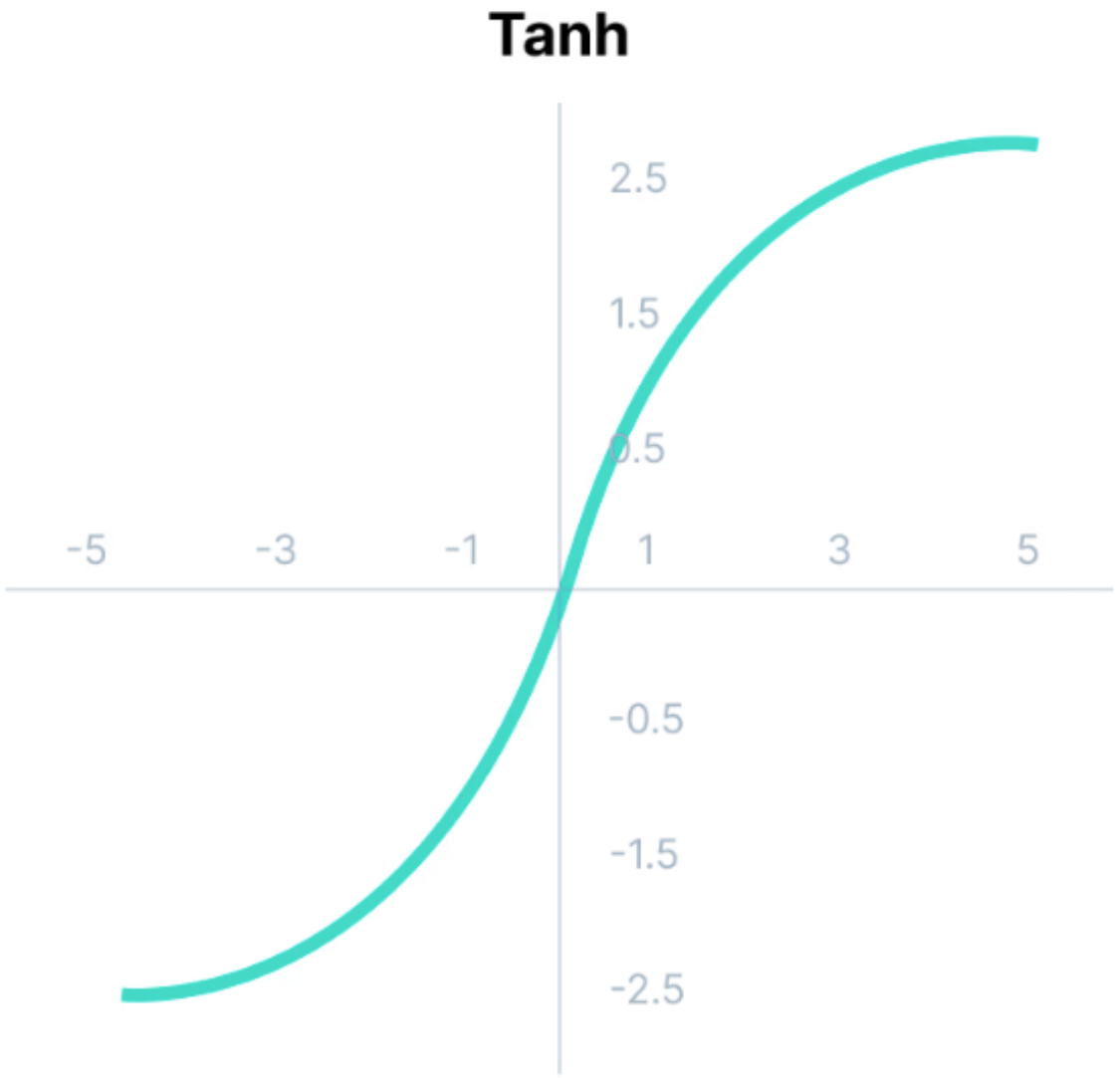
- Takes any real value as input and outputs values in the range of 0 to 1.
- The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0.
- It is so common that it has its own symbol, σ.

Hyperbolic Tangent

Tanh is closely related to the sigmoid: it is a scaled and shifted version:

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

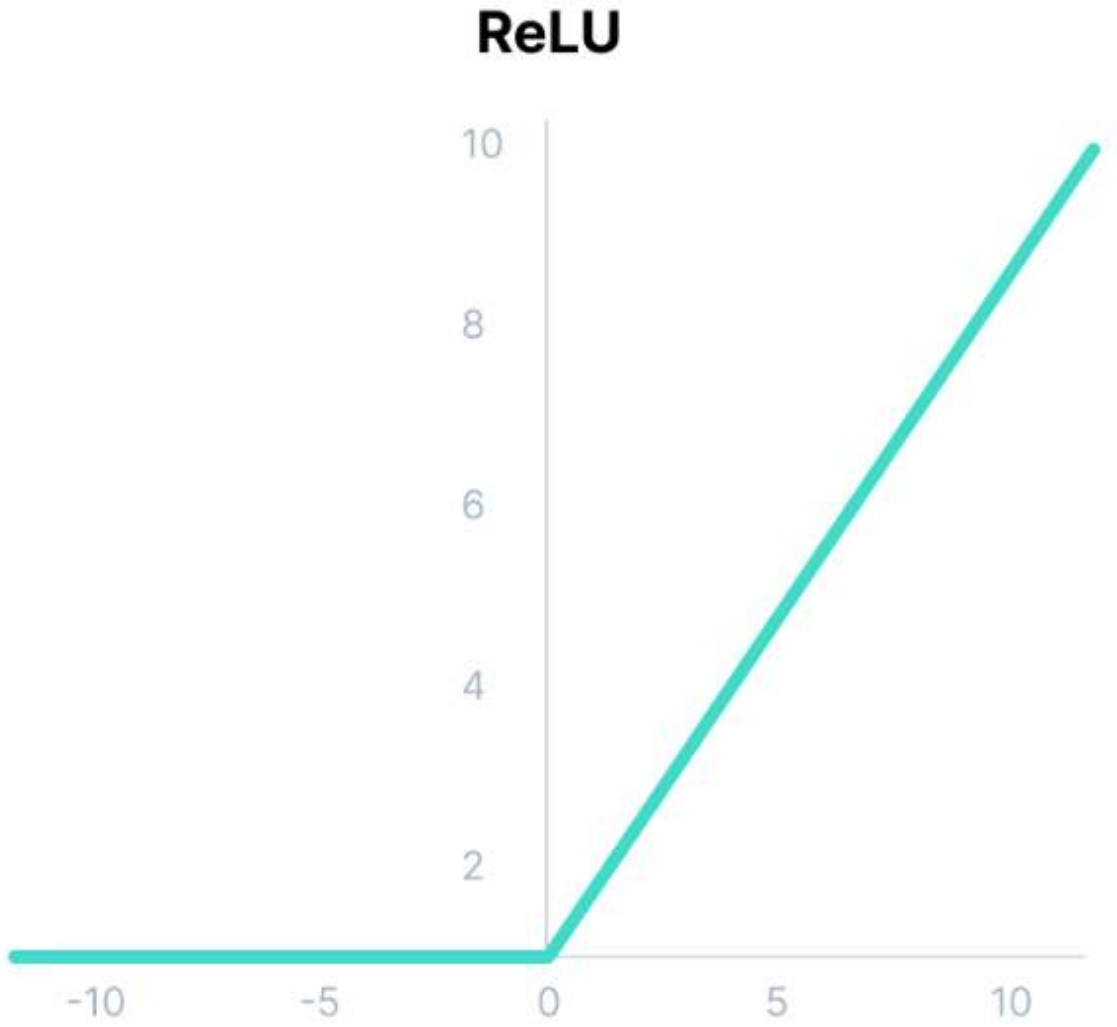
- The output is zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.
- Outputs lie between -1 to 1;
- It helps in centering the data in 0 and makes learning for the next layer much easier.



Rectified Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

- The neurons will only be deactivated if the output of the linear transformation is less than 0.
- Enables efficient training due to its simple computation
- Has some nice math properties



Exercise

Exercise 1 (6 points)

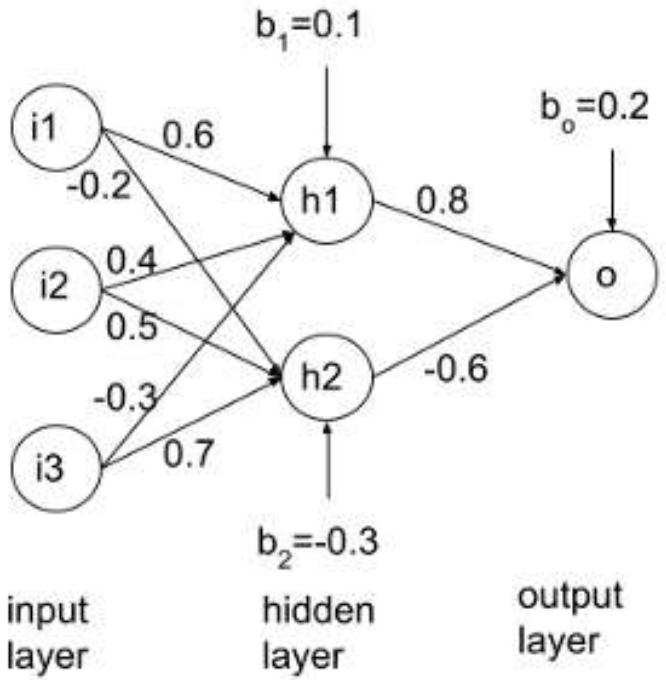
Given the following trained neural network with assigned weights and two different activation functions:

1)f(S) = ReLU(S) for the hidden layer

2)f(S) = sign(S) for the output layer (where sign(x) is equal to +1 when x >=0, is equal to -1 otherwise).

Apply it to the test set provided. Then, compute accuracy, precision and recall for the positive class.

i1	i2	i3	Real label
1	0	-1	1
0	1	1	1
1	1	1	-1
1	0	1	1
1	-1	1	1



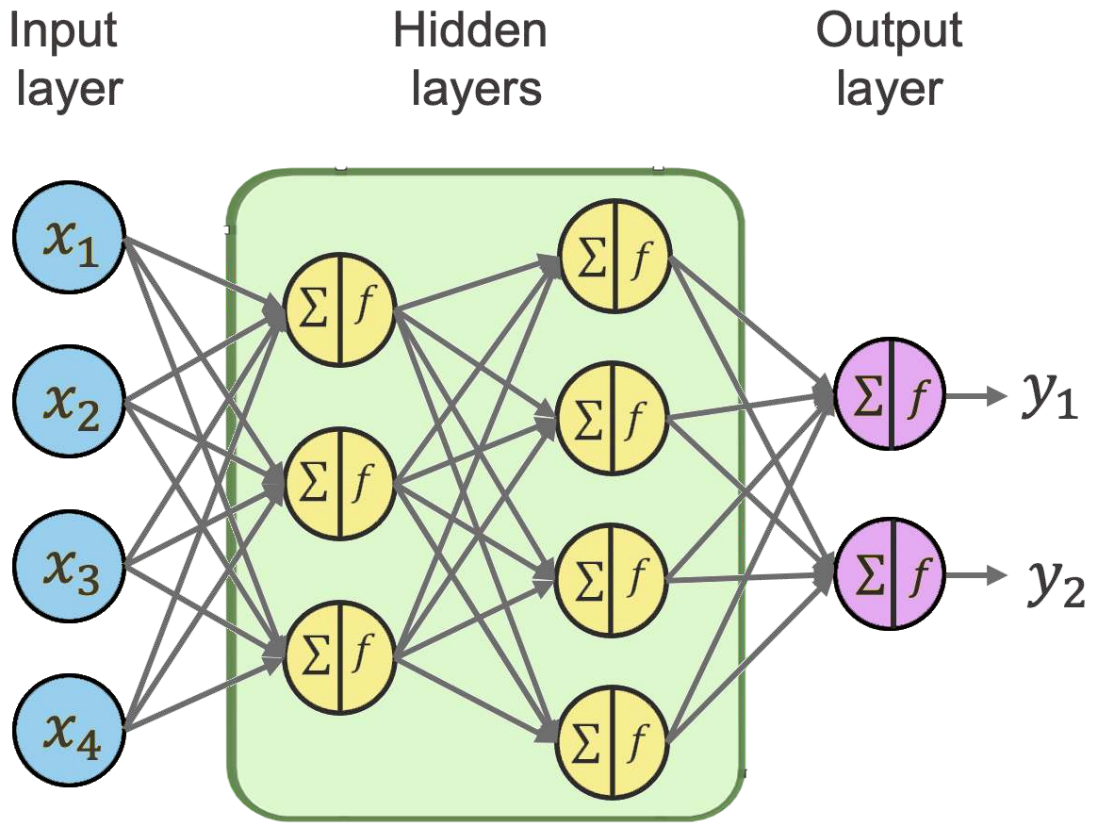
Multilayer Perceptrons

Gradient Descent

Training an MLP

Similar to the linear perceptron, training in an MLP has the objective of:

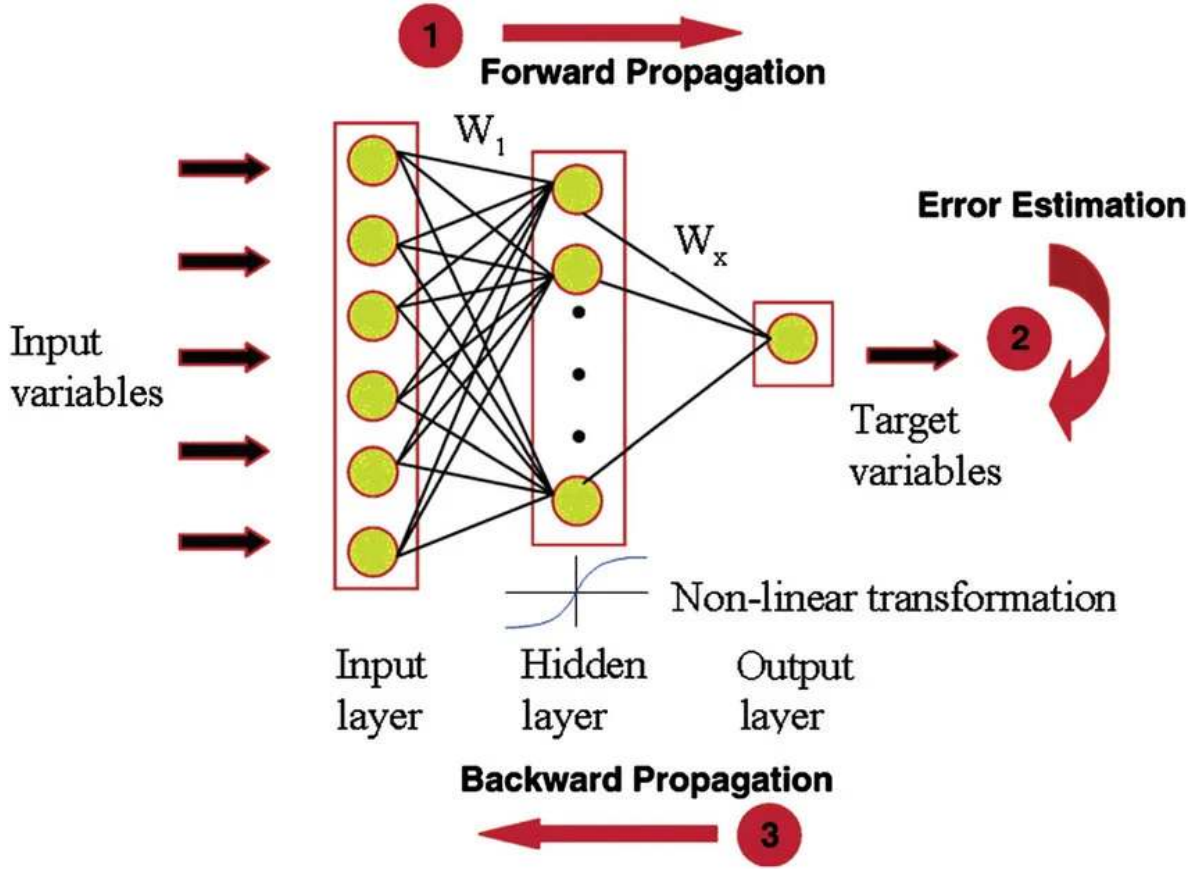
- adjust the weights of the network
- to move its output in the direction
- that minimizes the loss (error) on training data



Phases

Training always happens in two phases:

1. **Forward pass:**
   - Compute the output of the network
   - Compute the loss (error) to quantify how wrong the prediction is
2. **Backward pass (backpropagation):**
   - Propagate the error **backwards**
   - Compute how much **each weight contributed** to the error



Why Not the Perceptron Rule?

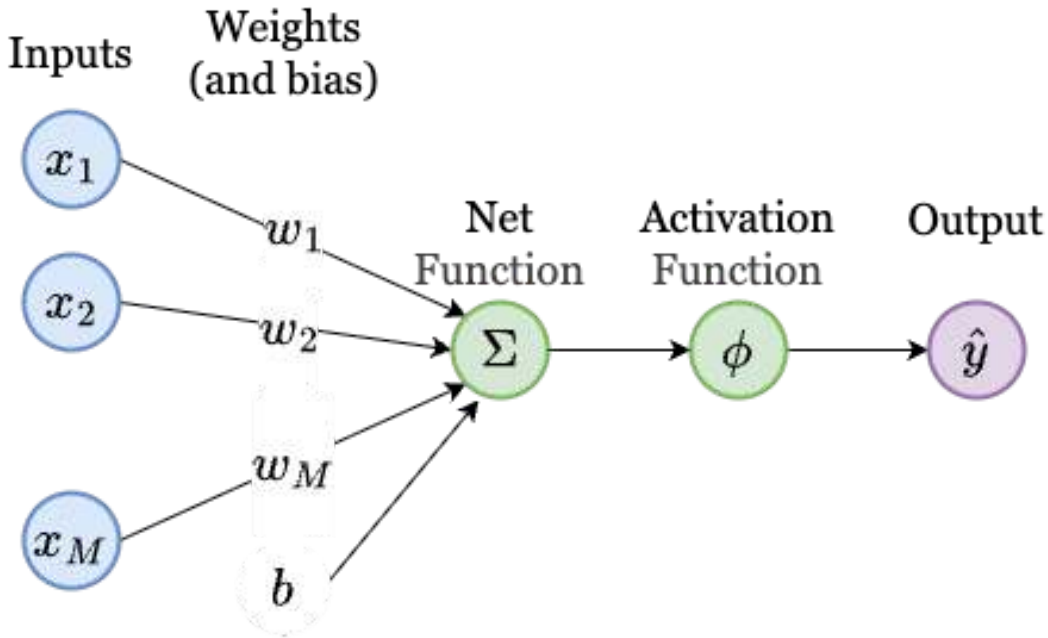
The Perceptron Update Rule is:

$$\mathbf{w}_{\text{new}} \leftarrow \mathbf{w}_{\text{old}} + \lambda(y - \hat{y})\mathbf{x}$$

This works because:

- There is a **single layer** (0 hidden layers)
- Each weight **directly affects the output**
- The error $(y - \hat{y})$ can be assigned **immediately**

Each weight "knows" how to update itself from the final error: $(y - \hat{y})$ tells you whether to increase or decrease the output, while $\mathbf{x}$ gives the direction that directly changes $\mathbf{w} \cdot \mathbf{x}$.



What Changes in MLPs

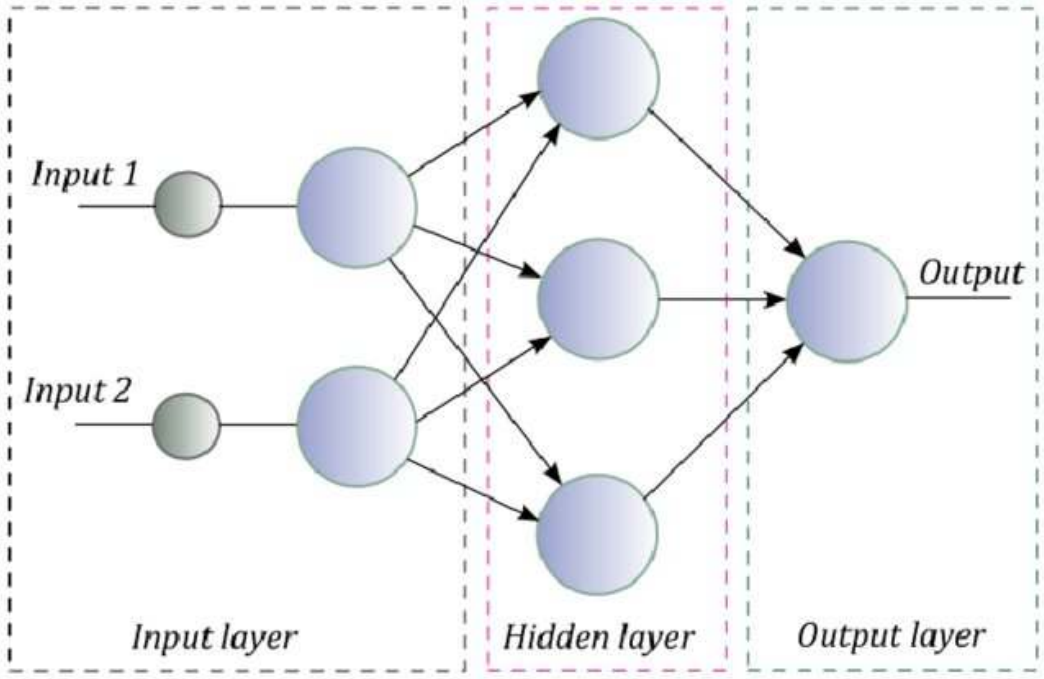
Let's take for example an MLP with one hidden layer:

$$\hat{\mathbf{y}} = f(f(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)$$

Now:

- Early weights (e.g. $\mathbf{W}_1$) **do not directly affect the output**
- Their effect passes through **multiple layers and nonlinearities**

The relationship between weights and output is no longer direct, so **we cannot directly use the perceptron update rule**!
We no longer know how each weight contributed to the error.



The Credit Assignment Problem

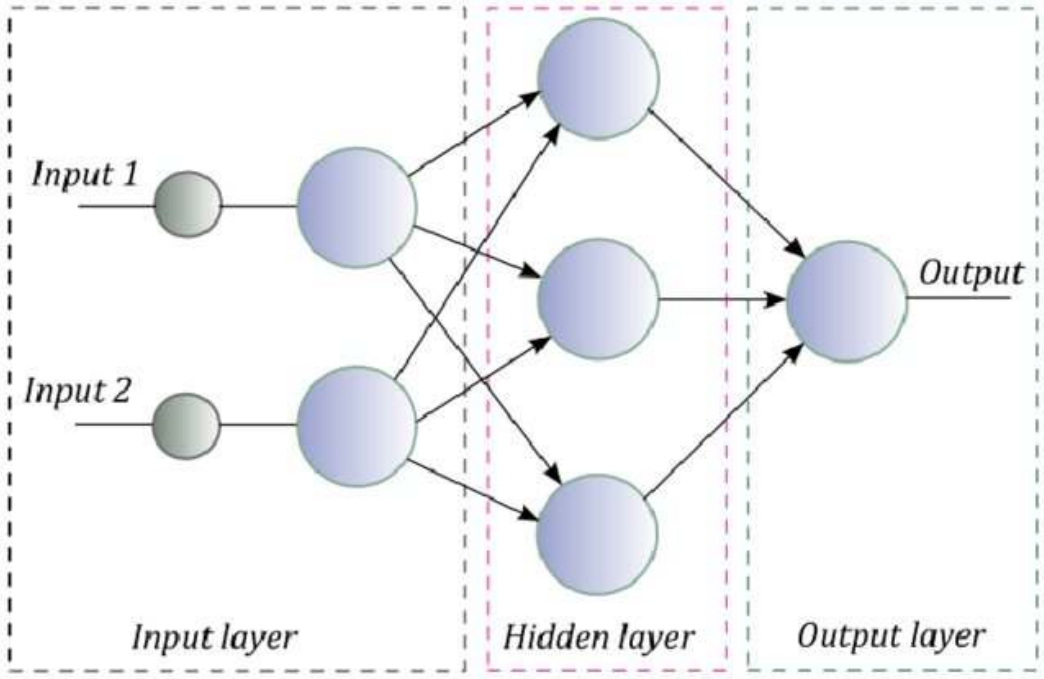
We observe an error at the output:

$$\mathcal{L}(y, \hat{y})$$

But:

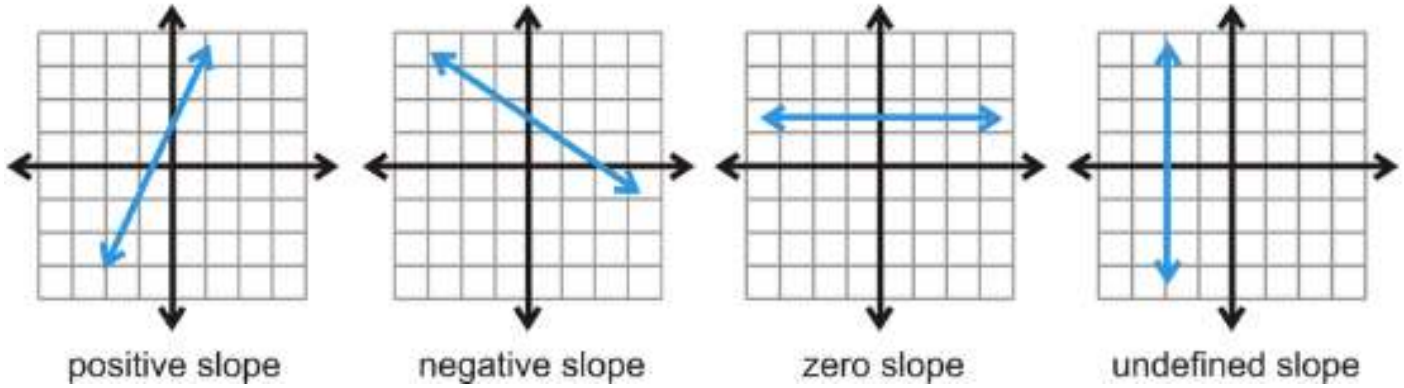
- Which layer caused the error?
- Which neuron contributed the most?
- How much should each weight change?

We need to assign **responsibility** for the error. How can we get the "direction" in which to then change the weights?



Slope

The slope is the steepness of a line. A line can have positive, negative, zero (horizontal), or undefined (vertical) slope.

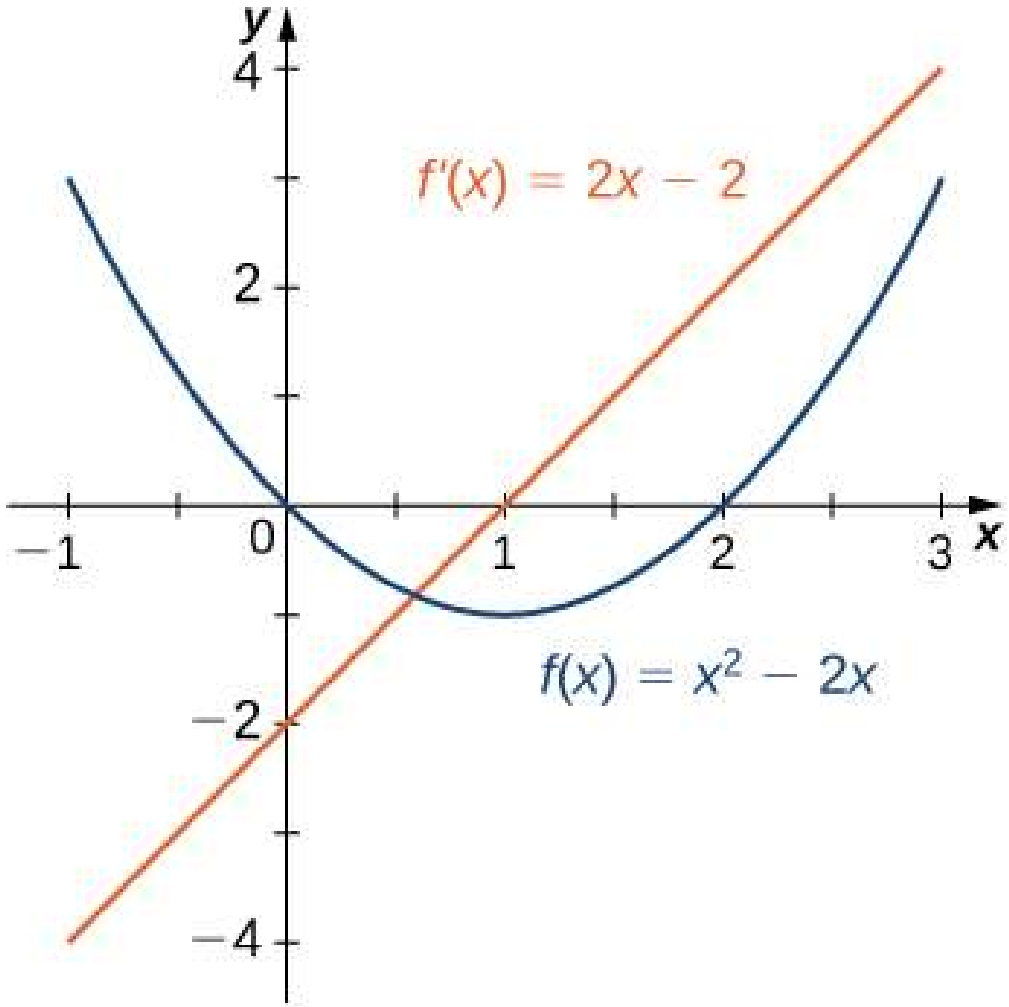


Derivative

The derivative of a function $f(x)$ with respect to x is a function

$$f'(x) = \frac{df}{dx}$$

that returns the slope of f at a given point x

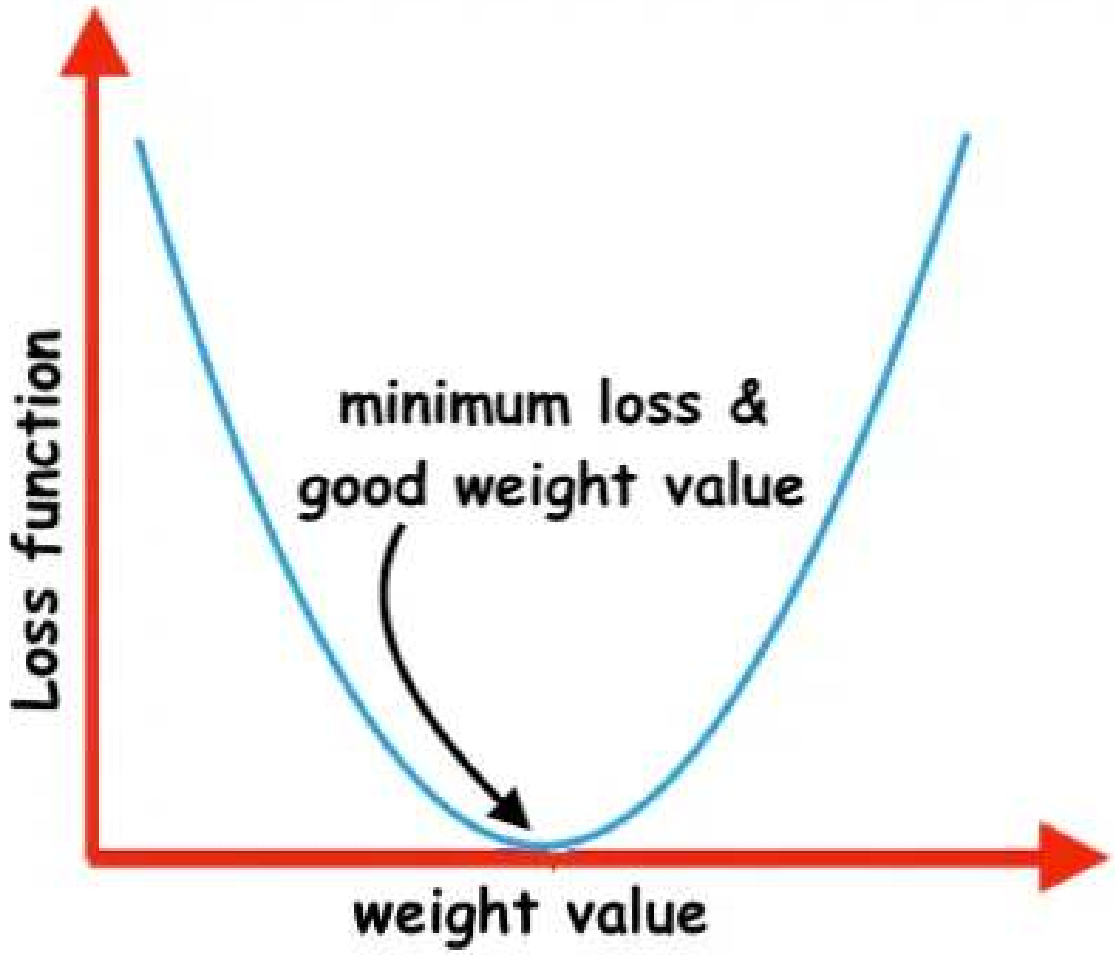


Derivative

It quantifies the sensitivity to change of a function's output with respect to its input.

In MLPs we want to understand how the **loss function** changes depending on the weight(s)

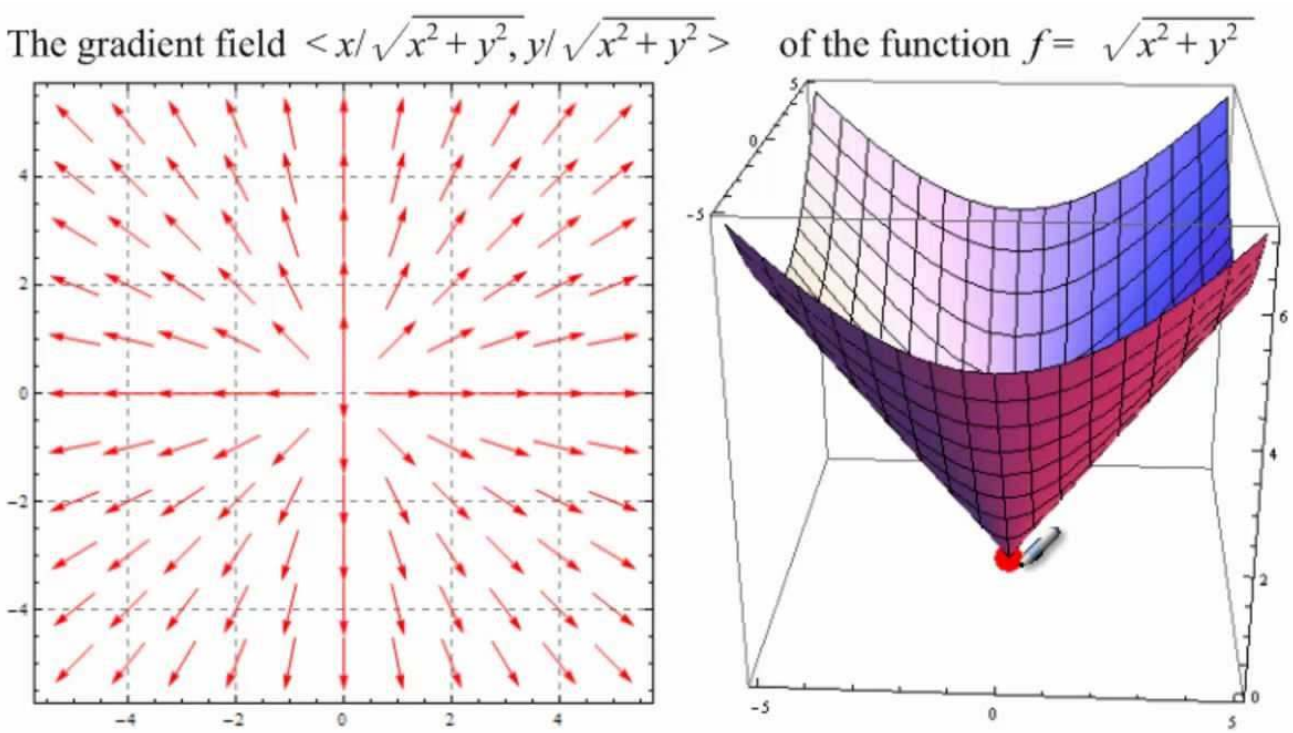
$$\frac{\partial \mathcal{L}}{\partial w}$$



Gradients

Given that in an MLP we have multiple weights we need **gradients**: partial derivatives of functions of multiple variables.

- The gradient indicates the direction of steepest increase of the loss.
- To reduce the loss we move in the opposite direction.



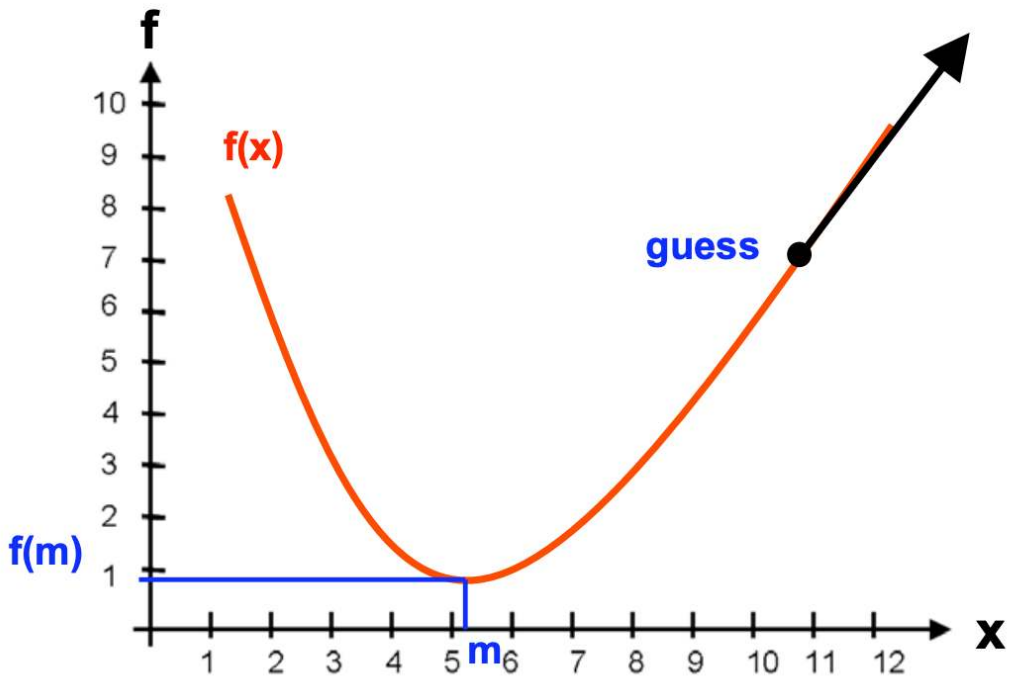
Weight Update Rule

Given weights $\mathbf{w}$, loss $\mathcal{L}(y, \hat{y})$, we compute **the gradient vector of the loss with respect to the weights**, $\nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y})$, that is the vector of the slopes of $\mathcal{L}$ with respect to each weight.

The update rule is:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y})$$

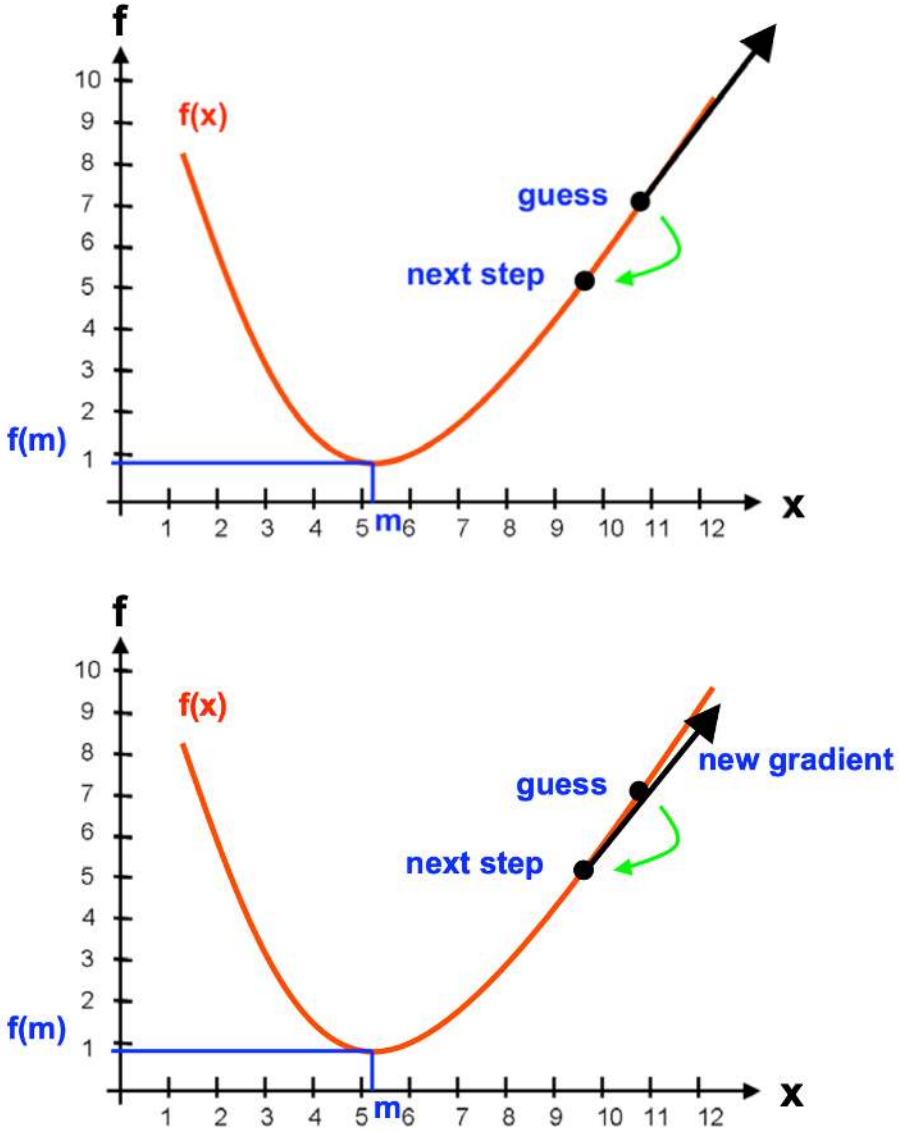
- η is the learning rate
- $-\nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y})$ is the direction of steepest decrease (of the loss)



Stochastic Gradient Descent (SGD) - Online

A classical way of training an MLP is with SGD:

1. Initialize the weights $\mathbf{w}$.
2. For every training epoch:
   - for each training example $\mathbf{x}^{[i]}$, with label $y^{[i]}$:
     a. Inference: $\hat{y} = f_{\mathbf{w}}(\mathbf{x}^{[i]})$
     b. Compute the gradients: $\nabla_{\mathbf{w}} \mathcal{L}(y^{[i]}, \hat{y}^{[i]})$
     c. Update weights: $\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(y^{[i]}, \hat{y}^{[i]})$

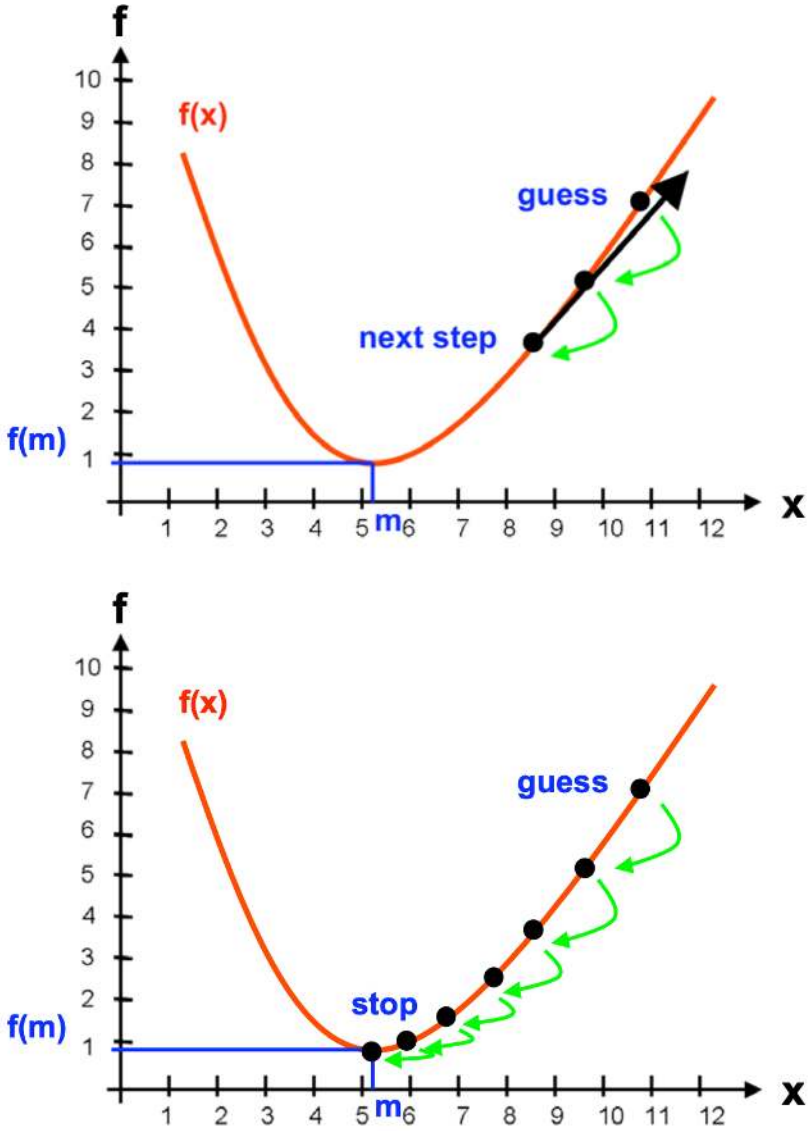


Stochastic Gradient Descent (SGD) - Online

A classical way of training an MLP is with SGD:

1. Initialize the weights $\mathbf{w}$.
2. For every training epoch:
   - for each training example $\mathbf{x}^{[i]}$, with label $y^{[i]}$:
     a. Inference: $\hat{y} = f_{\mathbf{w}}(\mathbf{x}^{[i]})$
     b. Compute the gradients: $\nabla_{\mathbf{w}} \mathcal{L}(y^{[i]}, \hat{y}^{[i]})$
     c. Update weights: $\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(y^{[i]}, \hat{y}^{[i]})$

Gradients tend to become smaller as training approaches the minimum, because the loss surface flattens.



Backpropagation

Gradients $\nabla_{\mathbf{w}} \mathcal{L}(y, \hat{y})$ are computed via **backpropagation.**

Backpropagation:

- Starts from the **final error**
- Propagates it **backwards through the network**
- Each layer receives:
  - how much it contributed to the error
  - how its weights should change

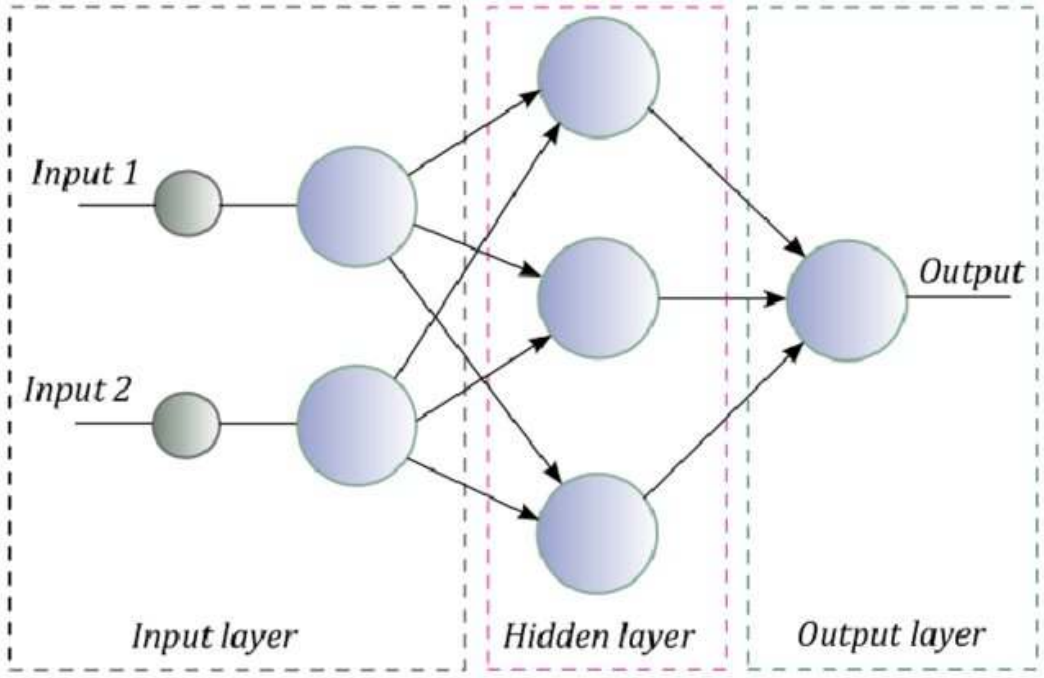
"Who is responsible for the error, and by how much?"

$$\text{Chain Rule:} \quad z = f(g(x)), \quad \text{let } u = g(x) \quad \Rightarrow \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial u} \cdot \frac{\partial u}{\partial x}$$

Chain Rule Example

For an MLP with one hidden layer:

$$
\begin{aligned}
\mathbf{Z}_1 &= \mathbf{X}\mathbf{W}_1, \\
\mathbf{H}_1 &= f(\mathbf{Z}_1), \\
\mathbf{Z}_2 &= \mathbf{H}_1\mathbf{W}_2, \\
\hat{\mathbf{y}} &= g(\mathbf{Z}_2), \\
\mathcal{L} &= \mathcal{L}(\hat{\mathbf{y}})
\end{aligned}
$$



Chain Rule Example

Chain rule follows paths in the **computational graph:**

for $\mathbf{W}_2$

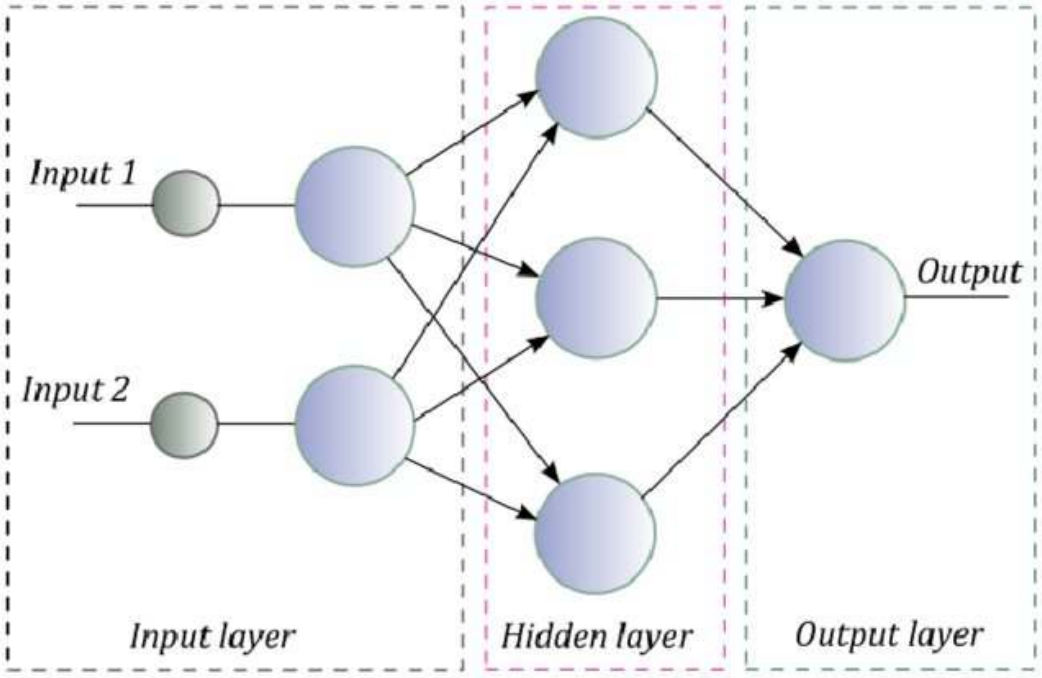
$$\mathbf{W}_2 \rightarrow \mathbf{Z}_2 \rightarrow \hat{\mathbf{y}} \rightarrow \mathcal{L}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{Z}_2} \cdot \frac{\partial \mathbf{Z}_2}{\partial \mathbf{W}_2}$$

for $\mathbf{W}_1$

$$\mathbf{W}_1 \rightarrow \mathbf{Z}_1 \rightarrow \mathbf{H}_1 \rightarrow \mathbf{Z}_2 \rightarrow \hat{\mathbf{y}} \rightarrow \mathcal{L}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{Z}_2} \cdot \frac{\partial \mathbf{Z}_2}{\partial \mathbf{H}_1} \cdot \frac{\partial \mathbf{H}_1}{\partial \mathbf{Z}_1} \cdot \frac{\partial \mathbf{Z}_1}{\partial \mathbf{W}_1}$$



Activation Function Derivatives

Earlier layers depend on **longer chains** of derivatives, which can make **gradient propagation more difficult**.

So certain activation functions are better because their derivatives preserve the signal during backpropagation. They should:

- avoid derivatives that are too small (which cause **vanishing gradients**)
- avoid derivatives that are too large (which cause **exploding gradients**)
- maintain useful gradients over a wide input range

Name	Function	Derivative
Sigmoid	$\phi(x) = \frac{1}{1+e^{-x}}$	$\phi'(x) = \phi(x)(1-\phi(x))$
TanH	$\phi(x) = \frac{2}{1+e^{-2x}} - 1$	$\phi'(x) = 1-\phi(x)^2$
ReLU	$\phi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$	$\phi'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$
Leaky ReLU	$\phi(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}$	$\phi'(x) = \begin{cases} \alpha & x \leq 0 \\ 1 & x > 0 \end{cases}$

Why It Is Efficient

A naive approach would recompute derivatives **many times**.

Backpropagation:

- Reuses intermediate results from the forward pass
- Reuses partial derivatives computed for later layers
- Computes **all gradients in one backward pass**

So the cost of computing **all gradients** is about the same as **one forward pass**.

Gradient Descent vs Backpropagation

- **Backpropagation is an algorithm to compute gradients.** It efficiently calculates the partial derivatives of the loss function with respect to each parameter in a neural network using the chain rule.
- **Gradient descent is an optimization method to update parameters.** It uses the gradients (often computed by backpropagation) to adjust the parameters in the direction that reduces the loss.

Batch vs Online Learning Principles

Online

1. Initialize the weights
2. For every training epoch:
    - for each training example $\mathbf{x}^{[i]}$, with label $y^{[i]}$:

      a) Inference: compute the output

      b) Calculate the error

      c) Update the weights

Batch

1. Initialize the weights
2. For every training epoch:
    - Initialize the weight update delta
    - For each training example $\mathbf{x}^{[i]}$, with label $y^{[i]}$:

      a) Inference: compute the output

      b) Calculate the error

      c) Update the weights delta
    - Update the weights

Batch vs Online Learning Principles

Online

Pros:

- more frequent updates;
- could converge faster
- small memory usage

Cons:

- more unstable

Batch

Pros:

- more stable
- parallel computation

Cons:

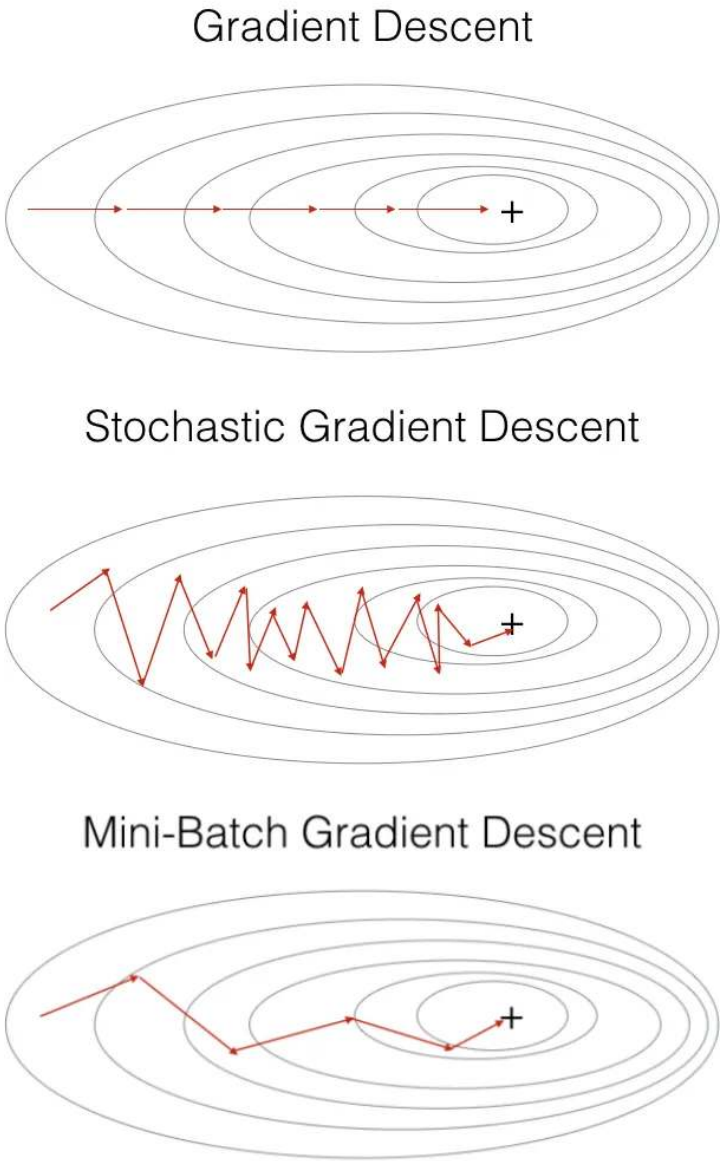
- could converge slower
- higher memory usage

Minibatch

1. Initialize the weights
2. For every training epoch:
   For every minibatch:
   - Initialize the weight update delta
   - For each training example in the minibatch:
     a) Inference: compute the output

     b) Calculate the error

     c) Update the weights delta
   - Update the weights



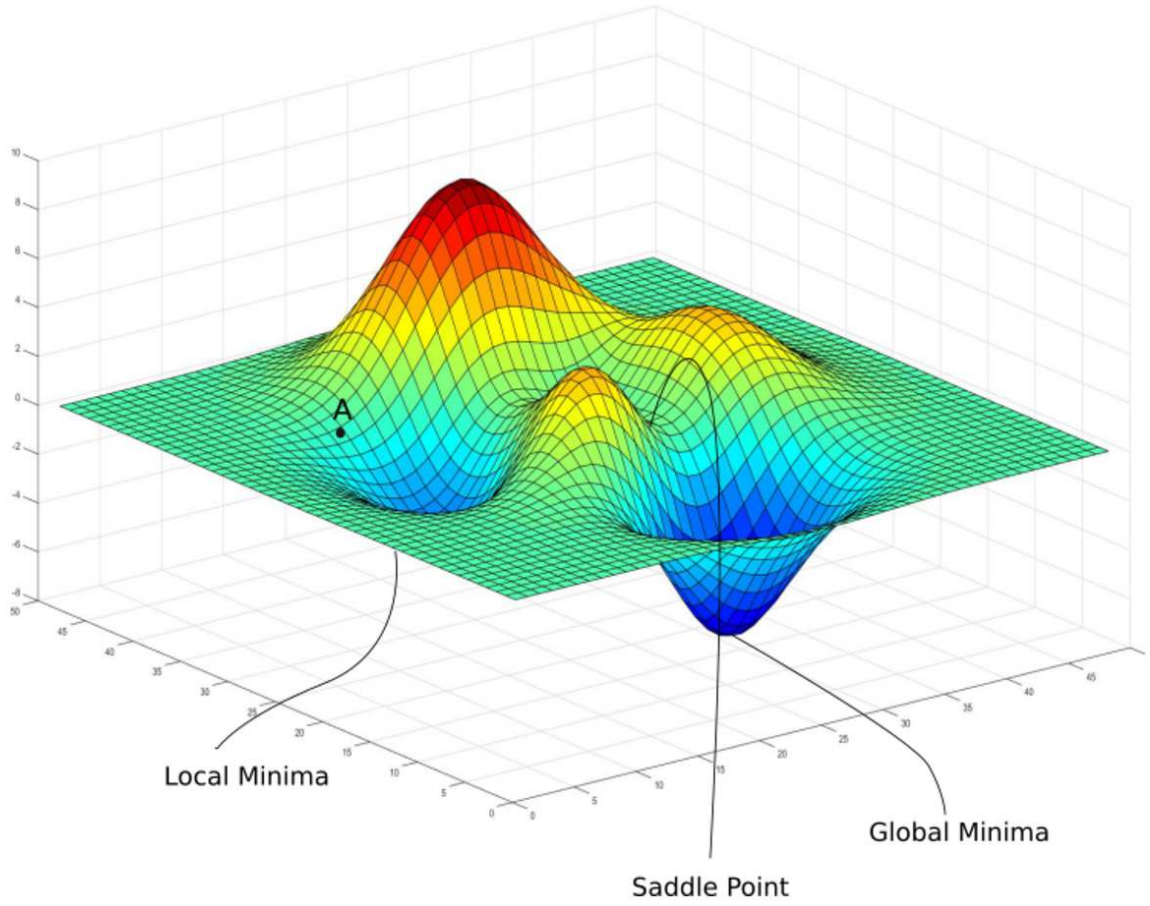
Convergence

MLPs with nonlinear activations produce a **non-convex loss function**, and non-convex landscapes contain many stationary points

Possible outcomes of gradient descent:

- Local minima
- Saddle points
- Flat regions (plateaus)

Therefore there is **no guarantee of convergence to the global minimum**, but convergence only to a local stationary point



See next lecture for tips to avoid these problems!

In Summary

Training a neural network is just:

1. Forward pass → compute predictions and loss
2. Backward pass → compute gradients (backpropagation)
3. Update weights → gradient descent
4. Repeat