# Optimization for deep learning

Toulouse School of Economics

Ludovic De Matteis

# Course Overview

## Planning

| | Day | Time | Subject |
|---|---|---|---|
| 1 | 13/10/2025 | 9:30 - 12:30 | Basic definitions, Gradient Descent and Newton's method |
| 2 | 27/10/2025 | 9:30 - 12:30 | Practical - Gradient Descent and Newton's method |
| 3 | 03/11/2025 | 9:30 - 12:30 | Neural networks and stockastic gradient descent |
| 4 | 10/11/2025 | 9:30 - 12:30 | Practical - Neural Networks and digit recognition |
| 5 | 17/11/2025 | 9:30 - 12:30 | Alternative Neural Structures |
| 6 | 24/11/2025 | 9:30 - 12:30 | Practical - Alternative Neural structure, Adversial networks |

# Lecture 2 -

## Optimizing neural networks

# Summary

# From perceptrons to multi-layer perceptrons

# From perceptrons to multi-layer perceptrons

## Motivations

- Neural networks are inspired by the structure of the human brain where neurons are interconnected to process information and learning occurs by adjusting the strength of these connections
- The perceptron is a fundamental building block of neural networks, representing a simplified model of a biological neuron
- In machine learning, we look for a parametrized model capable of approximating a function $f : \mathbb{R}^d \mapsto \mathbb{R}^n$ from examples

$$f_\theta \approx (x_i)y_i$$

# From perceptrons to multi-layer perceptrons

## History

- **1943**: McCulloch and Pitts propose a mathematical model of a neuron based on logical gates
- **1958**: Frank Rosenblatt invents the perceptron, an early neural network model capable of binary classification
- **1969**: Minsky and Papert publish "Perceptrons", highlighting limitations of single-layer perceptrons, leading to a decline in neural network research
- **1986**: Rumelhart, Hinton, and Williams popularize the backpropagation algorithm, enabling training of multi-layer neural networks

# From perceptrons to multi-layer perceptrons

## Definition of a perceptron

- A perceptron is a simple computational unit that takes multiple input signals, applies weights to them, sums them up, and passes the result through an activation function to produce an output
- Mathematically, a perceptron can be represented as:

$$\text{output} = \Phi\Big(\sum(\text{weights} * \text{inputs}) + \text{bias}\Big)$$

or

$$y = \Phi(w^T x + b)$$

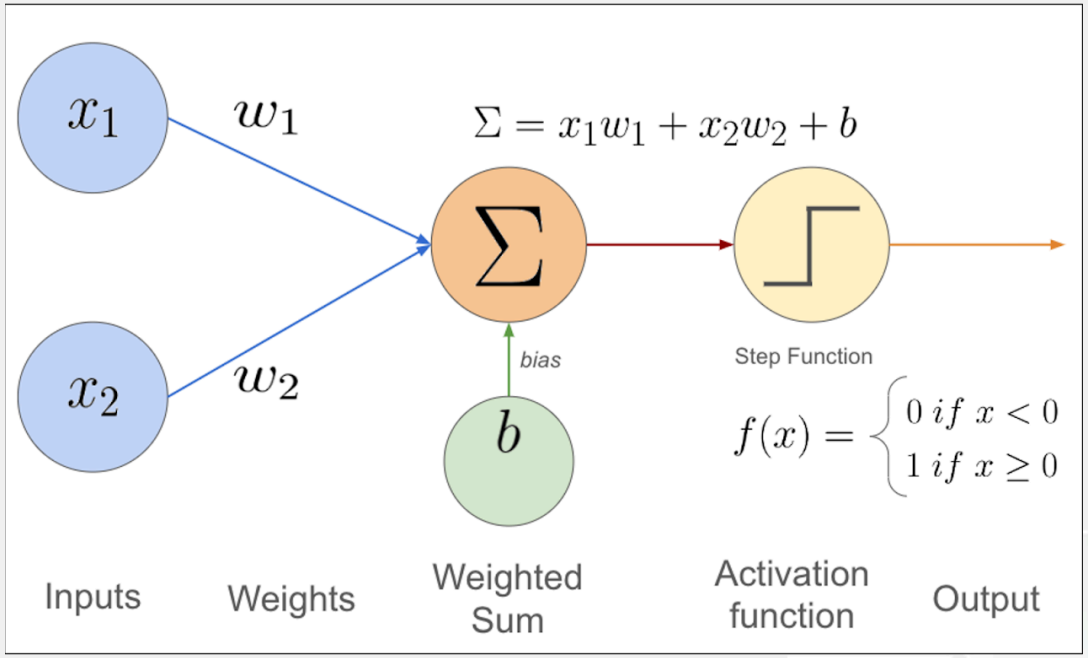
where:

- $x$ are the **input** features
- $w$ are the **weights**, the parameters learned during training
- $b$ is the bias, an additional parameter to shift the activation function
- $\Phi$ is the **activation function** introducing non-linearity (e.g., step function, sigmoid, ReLU)

# From perceptrons to multi-layer perceptrons

## Definition of a perceptron

- The classical representation of a perceptron is as follows:



# From perceptrons to multi-layer perceptrons

## Linear classifier

- If we use a step activation function, the perceptron can be used as a linear classifier

$$\Phi(z) = 1 \text{ if } z \geq 0 \text{ else } 0$$

- The decision boundary is defined by the equation:

$$w^T x + b = 0$$

- In the first practical session, we implemented the perceptron using a sigmoid activation function (closer to the step function)

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

# From perceptrons to multi-layer perceptrons

## Optimizing a perceptron

- We look for the optimal values of $w$ and $b$
- We can use gradient descent to minimize a loss function, such as the mean squared error or cross-entropy loss (if everything if differentiable)
- Or we can use the following algorithm:
  1. Compute the output of the perceptron for each training example

$$\hat{y}_i = \Phi(w^T x_i + b)$$

  2. Update the weights and bias based on the prediction error

$$w := w + \eta(y_i - \hat{y}_i)x_i$$

$$b := b + \eta(y_i - \hat{y}_i)$$

  where $\eta$ is the learning rate, controlling the step size of the updates

# From perceptrons to multi-layer perceptrons

## Limits of single-layer perceptrons

- Single-layer perceptrons can only learn linearly separable functions
- They cannot solve problems that require non-linear decision boundaries, such as the XOR problem
- To overcome these limitations, we extend the perceptron to multi-layer architectures, allowing for more complex representations and decision boundaries

# Multi-layer perceptron and forward propagation

# Multi-layer perceptron and forward propagation

## Motivations

- To obtain non linear functions, we combine multiple linear functions with non-linear activation functions
- The output will therefore be defined for a composition of $L$ functions as:

$$y = \Phi_L(W_L.\Phi_{L-1}(W_{L-1}....\Phi_1(W_1x + b_1)... + b_{L-1}) + b_L)$$

# Multi-layer perceptron and forward propagation

## Architecture of a multi-layer perceptron

- A multi-layer perceptron (MLP) consists of multiple layers of interconnected perceptrons (neurons)
- It typically includes an input layer, one or more hidden layers, and an output layer
- Each layer applies a linear transformation followed by a non-linear activation function (that can be different for each layer)
- The architecture allows the network to learn complex patterns and representations from the data
- The function computed by each layer $l$ can be expressed as:

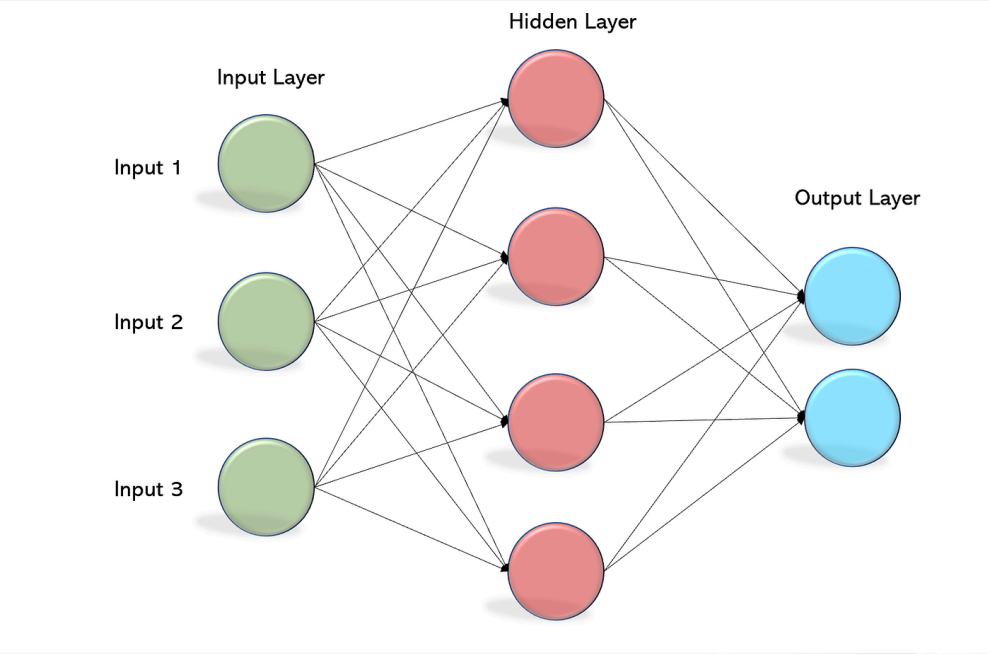
$$a^l = \Phi_l\left(W_l a^{l-1} + b_l\right)$$

with $a^0 = x$

# Multi-layer perceptron and forward propagation

## Architecture of a multi-layer perceptron

- Example of a simple MLP with one hidden layer (we often omit the bias nodes in diagrams for clarity):



# Multi-layer perceptron and forward propagation

## Forward propagation

- Forward propagation is the process of computing the output of the MLP given an input
- It involves passing the input through each layer, applying the linear transformation and activation function at each step
- The steps for forward propagation are as follows:
  1. Initialize the input layer with the input features $x$
  2. For each layer $l$ from 1 to $L$:
     - Compute the linear transformation:

$$z^l = W_l a^{l-1} + b_l$$

     - Apply the activation function:

$$a^l = \Phi_l(z^l)$$

  3. The final output is obtained from the output layer:

$$\hat{y} = a^L$$

# Multi-layer perceptron and forward propagation

## Forward propagation algorithm

- It can be simply implemented in Python as follows:

```
def forward_propagation(X, weights, biases):
    a = X
    for W, b, Phi in zip(weights, biases):
        z = W @ a + b
        a = phi(z)
    return a
```

# Multi-layer perceptron and forward propagation

## XOR problem

- The XOR problem is a classic example that illustrates the limitations of single-layer perceptrons and the capabilities of multi-layer perceptrons
- It can be solved using a MLP with a single hidden layer of two neurons
- It is possible to write by hand the formulation of the output

# Multi-layer perceptron and forward propagation

## Activation functions

- The choice of activation function can significantly impact the performance and convergence of the network during training
- Common activation functions used in MLPs include:
  - Sigmoid:

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

    Bounded, differentiable, but can suffer from vanishing gradients
  - ReLU (Rectified Linear Unit):

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

    Simple and efficient, helps mitigate vanishing gradients, but can suffer from dying ReLU problem

# Multi-layer perceptron and forward propagation

## Activation functions

- Tanh:

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

  Bounded between $-1$ and 1, zero-centered, but can also suffer from vanishing gradients
- Leaky ReLU:

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

  Similar to ReLU but allows a small gradient when the unit is not active
- Softmax:

$$\Phi(z_i) = \frac{e^{z_i}}{\sum(e^{z_j})}$$

  Used in the output layer for multi-class classification problems

# Multi-layer perceptron and forward propagation

## Loss functions

- The loss function quantifies the difference between the predicted output and the true output
- Common loss functions used in training MLPs include:
  - Mean Squared Error (MSE):

$$L(y, \hat{y}) = \frac{1}{2} \sum \left( (y_i - \hat{y}_i)^2 \right)$$

    Used for regression tasks
  - Binary Cross-Entropy Loss:

$$L(y, \hat{y}) = -\sum (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

    Used for binary classification tasks
  - Cross-Entropy Loss:

$$L(y, \hat{y}) = -\sum (y_i \log(\hat{y}_i))$$

    Used for classification tasks, especially with softmax outputs

# Multi-layer perceptron and forward propagation

## Feature detection in layers

- In MLPs, each layer can be thought of as learning to detect different features of the input data
- Early layers may learn to detect simple features (e.g., edges in images), while deeper layers can learn more complex patterns (e.g., shapes, objects)
- This hierarchical feature learning allows MLPs to effectively model complex relationships in the data

# Multi-layer perceptron and forward propagation

## Exercise

- Given

$x = [0.5, -1.0]^T, \quad W^{(1)} = \begin{pmatrix} 1 & 2 \\ -1 & 0.5 \end{pmatrix}, \quad b^{(1)} = [0, 1]^T, \quad W^{(2)} = [1, -2], \quad b^{(2)} = [0]$

- Draw the architecture of the MLP
- Compute $z^{(1)}, a^{(1)}, z^{(2)}, a^{((2)}$
- Compute the output of the MLP using ReLU activation function

# Backpropagation and derivatives

# Backpropagation and derivatives

## Motivations

- The goal of the learning process is to minimize the loss function $L(y, \hat{y})$ by adjusting the parameters of the network $\theta = \left\{W^{(l)}, b^{(l)}\right\}$

$$\min_{\theta} L(y, \hat{y}(\theta))$$

- To achieve this, we need to compute the gradients of the loss function with respect to each parameter in the network

$$\nabla_{\theta} L = \frac{\partial L}{\partial \theta}$$

- The backpropagation algorithm efficiently computes these gradients for Multi-Layer Perceptrons

# Backpropagation and derivatives

## Chain rule

- A neural network performs function composition

$$f(x) = f_L(f_{L-1}(...f_1(x)...))$$

- To compute the derivative of $L$ with respect to a layer, we use the chain rule:

$$\frac{\partial L}{\partial a^l} = \frac{\partial L}{\partial a^{(l+1)}} \frac{\partial a^{(l+1)}}{\partial a^{(l)}}$$

- This is the basic principle behind backpropagation, we propagate the graidients **from the output layer back to the input layer**.

# Backpropagation and derivatives

## Backpropagation algorithm

- The backpropagation algorithm consists of the following steps:

**Step 1**

Perform a forward pass to compute the activations and outputs of each layer $z^{(l)}$ and $a^{(l)}$

**Step 2**

Compute the loss $L(y, \hat{y})$

**Step 3**

Compute the gradient of the loss with respect to the output layer

$$\delta^{(L)} = \frac{\partial L}{\partial z^{(L)}}$$

# Backpropagation and derivatives

## Backpropagation algorithm

### Step 4

For each layer $l$ from $L-1$ to 1:

- Compute the gradient of the loss with respect to the pre-activation $z^l$

$$\delta^{(l)} = \left(W^{(l+1)}\right)^T \delta^{(l+1)} * \Phi'\left(z^{(l)}\right)$$

- Compute the gradients with respect to the weights $W_l$ and biases $b_l$

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} \left(a^{(l-1)}\right)^T$$

$$\frac{\partial L}{\partial b^{(l)}} = \delta^{(l)}$$

We can then use these gradients to update the parameters using gradient descent

# Backpropagation and derivatives

## Example: MLP with one hidden layer

- Consider a MLP with one hidden layer and MSE loss function

### Step 1 - Forward pass

$$z^{(1)} = W_1 x + b_1, \quad a^{(1)} = \Phi\big(z^{(1)}\big)$$

$$z^{(2)} = W_2 a^{(1)} + b_2, \quad \hat{y} = a^{(2)} = \Phi\big(z^{(2)}\big)$$

### Step 2

$$L(y, \hat{y}) = \frac{1}{2} \parallel y - \hat{y} \parallel^2$$

### Step 3 - Output layer gradient

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = (\hat{y} - y) * \Phi'\big(z^{(2)}\big)$$

### Step 4 - Hidden layer gradient

$$\delta^{(1)} = \big(W_2\big)^T \delta^{(2)} * \Phi'\big(z^{(1)}\big)$$

# Backpropagation and derivatives

## Example: MLP with one hidden layer

### Final gradients

$$\frac{\partial L}{\partial W_2} = \delta^{(2)} \left(a^{(1)}\right)^T, \quad \frac{\partial L}{\partial b_2} = \delta^{(2)}$$

$$\frac{\partial L}{\partial W_1} = \delta^{(1)} x^T, \quad \frac{\partial L}{\partial b_1} = \delta^{(1)}$$

### Parameter update

$$W_l := W_l - \eta \frac{\partial L}{\partial W_l}$$

$$b_l := b_l - \eta \frac{\partial L}{\partial b_l}$$

# Backpropagation and derivatives

## Vanishing and exploding gradients

- During backpropagation, gradients can sometimes become very small (vanishing gradients) or very large (exploding gradients)
- Vanishing gradients make it difficult for the network to learn, especially in deep networks, as the updates to the weights become negligible
- Exploding gradients can lead to unstable training, causing the weights to oscillate or diverge
- Techniques to mitigate these issues include careful weight initialization, using activation functions like ReLU, and employing normalization techniques such as batch normalization

# Optimization algorithms for training neural networks

# Optimization algorithm

## Motivations

- Now that we can compute the prediction of the neural network and the derivatives of the loss function with respect to the parameters of the network
- To optimize the parameters, we need to solve the following optimization problem:

$$\min_{\theta} L(y, \hat{y}(\theta))$$

- Due to the large number of parameters and the complexity of the loss landscape, efficient optimization algorithms are crucial for effective training

# Optimization algorithm

## Gradient descent

- We can use standard gradient descent methods to optimize the parameters of the neural network
- The update rule for gradient descent is given by:

$$\theta := \theta - \eta \nabla_\theta L$$

  where $\eta$ is the learning rate, controlling the step size of the updates
- In practice, computing the gradient over the entire dataset can be computationally expensive, leading to the use of stochastic and mini-batch gradient descent
- The complete gradient over the full dataset is given by:

$$\nabla_\theta L = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(y_i, \hat{y}_i(\theta))$$

  but can be costly to be computed for large datasets

# Optimization algorithm

## Challenges in optimization

- The loss landscape of neural networks is often highly non-convex, with many local minima, saddle points, and flat regions
- This complexity can make optimization challenging, as standard gradient descent methods may get stuck in poor local minima or take a long time to converge
- Additionally, the choice of learning rate is critical; too high a learning rate can lead to divergence, while too low a learning rate can result in slow convergence
- Various optimization algorithms have been developed to address these challenges, including stochastic gradient descent, mini-batch gradient descent, momentum-based methods, and adaptive learning rate methods like Adam

# Optimization algorithm

## Stochastic gradient descent

- Stochastic Gradient Descent (SGD) updates the parameters using the gradient computed from a batch of training examples

$$\nabla_\theta L \approx \frac{1}{n} \sum_{i \in \text{ batch}} \nabla_\theta L(y_i, \hat{y}_i(\theta))$$

- The update rule for SGD is given by:

$$\theta := \theta - \eta \nabla_\theta L(y_i, \hat{y}_i(\theta))$$

- Advantages
  - Introduces noise into the updates, which can help escape local minima and improve generalization
  - Less computationally expensive per iteration, allowing for faster updates
- Downsides
  - Can become unstable and oscillate around minima if the learning rate is not properly tune
  - Can be slower to converge if there is too much noise (i.e., batch size is too small)

# Optimization algorithm

## Momentum SGD

- Momentum SGD introduces a velocity term to the parameter updates, helping to smooth out the updates and accelerate convergence
- The update rule for Momentum SGD is given by:

$$v_t := \beta v_{t-1} + (1 - \beta) \nabla_\theta L(y_i, \hat{y}_i(\theta_t))$$

$$\theta_{t+1} := \theta_t - \eta v_t$$

  where $\beta$ is the momentum coefficient (typically between 0.9 and 0.99)
- Advantages
  - Helps accelerate convergence in the relevant direction and dampens oscillations
  - Can lead to faster training times compared to standard SGD
- Downsides
  - Requires tuning of the momentum coefficient in addition to the learning rate

# Optimization algorithm

## Nesterov Acceleration

- Nesterov Acceleration is a variant of momentum SGD that looks ahead at the future position of the parameters to compute the gradient
- The update rule for NAG is given by:

$$v_t := \beta v_{t-1} + (1-\beta)\nabla_\theta L(y_i, \hat{y}_i(\theta_t - \eta\beta v_{t-1}))$$

$$\theta_{t+1} := \theta_t - \eta v_t$$

- Advantages
  - Provides a more accurate estimate of the gradient, leading to improved convergence rates
  - Can help avoid overshooting minima compared to standard momentum SGD
- Downsides
  - Slightly more complex to implement and requires tuning of the momentum coefficient

# Optimization algorithm

## AdaGrad

- AdaGrad adapts the learning rate for each parameter based on the historical gradients, allowing for larger updates when required
- The update rule for AdaGrad is given by:

$$G_t := \sum_{\tau=1}^{t} \nabla_\theta L(y_i, \hat{y}_i(\theta_t))^2$$

$$\theta_{t+1} := \theta_t - \left( \frac{\eta}{\sqrt{G_t} + \varepsilon} \right) \nabla_\theta L(y_i, \hat{y}_i(\theta_t))$$

- Advantages
  - Automatically adjusts the learning rate, reducing the need for manual tuning
  - Particularly effective for sparse data and features
- Downsides
  - The learning rate can become excessively small over time, leading to slow convergence

# Optimization algorithm

## RMSProp

- RMSProp modifies AdaGrad by introducing a decay factor to the accumulated squared gradients, preventing the learning rate from becoming too small
- The update rule for RMSProp is given by:

$$E[g^2]_t := \beta E[g^2]_{t-1} + (1-\beta)\nabla_\theta L(y_i, \hat{y}_i(\theta_t))^2$$

$$\theta_{t+1} := \theta_t - \left(\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}}\right)\nabla_\theta L(y_i, \hat{y}_i(\theta_t))$$

- Advantages
  - Maintains a more stable learning rate over time compared to AdaGrad
  - Effective for non-stationary objectives and online learning scenarios
- Downsides
  - Requires tuning of the decay factor in addition to the learning rate

# Optimization algorithm

## Adam Optimizer

- Adam (Adaptive Moment Estimation) combines the benefits of Momentum SGD and RMSProp, maintaining both a running average of the gradients and their squared values
- The update rule for Adam is given by:

$$m_t := \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta L(y_i, \hat{y}_i(\theta_t))$$

$$v_t := \beta_2 v_{t-1} + (1 - \beta_2) \nabla_\theta L(y_i, \hat{y}_i(\theta_t))^2$$

$$\widehat{m}_t := \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t := \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} := \theta_t - \left( \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \right) \widehat{m}_t$$

# Optimization algorithm

## Adam Optimizer

- Advantages
  - Combines the benefits of momentum and adaptive learning rates
  - Generally performs well across a wide range of problems with minimal hyperparameter tuning
- Downsides
  - More complex to implement and requires tuning of multiple hyperparameters

# Optimization algorithm

## Summary of optimization algorithms

| Algorithm | Key idea | Advantages | Disadvantages |
|---|---|---|---|
| SGD | Mini-batch | Simple, better generalization | Sensitive to $\eta$ |
| Momentum SGD | Adds velocity | Accelerates convergence, reduces oscillations | Can oscillate, requires tuning of $\beta$ |
| Nesterov Acceleration | Looks ahead to compute gradients | Improved convergence rates, avoids overshooting | More complex implementation |
| AdaGrad | Adapts learning rates based on historical gradients | Automatic learning rate adjustment, effective for sparse data | Learning rate can become too small |
| RMSProp | Introduces decay factor to AdaGrad | Stable learning rates, effective for non-stationary objectives | Requires tuning of decay factor |
| Adam | Combines momentum and adaptive learning rates | Performs well across various problems | Complex implementation, multiple hyperparameters |

# Conclusion

# Conclusion

## Summary of key points

- Multi-layer perceptrons (MLPs) are powerful models capable of learning complex patterns through layers of interconnected neurons
- Forward propagation allows us to compute the output of the network given an input

$$\hat{y} = f_\theta(x)$$

- Backpropagation efficiently computes gradients for training

$$\nabla_\theta L(\theta) = \text{backpropagation}(y, \hat{y}(\theta))$$

- Various optimization algorithms, such as SGD, Momentum, RMSProp, and Adam, help in effectively training neural networks by addressing challenges in the loss landscape

$$\min_\theta L(y, \hat{y}(\theta))$$

# Conclusion

## Current challenges and future directions

- Despite the success of MLPs, several challenges remain in training deep neural networks
  1. Vanishing/exploding gradients: Developing techniques to mitigate these issues in very deep networks
  2. Overfitting: Developing better regularization techniques to improve generalization
  3. Scalability: Designing algorithms that can efficiently handle larger datasets and models
  4. Interpretability: Enhancing the transparency of neural networks to understand their decision-making processes
- Future research directions include exploring new architectures, optimization techniques, and applications across various domains

# Conclusion

## Next practical

- In the next class, we will explore practical implementations of the concepts covered today
- We will implement a multi-layer perceptron from scratch, including forward propagation, backpropagation, and optimization using different algorithms
- We will also apply these techniques to real-world datasets, such as digit recognition using the MNIST dataset

# Conclusion

## Next lecture

- In the next lecture, we will delve into different neural network architectures, focusing on Convolutional Neural Networks (CNNs)
- We will explore their architecture, how they differ from MLPs, and why they are particularly effective for image processing tasks
- We will also discuss advanced optimization techniques and regularization methods (Dropout, BatchNorm…)