Introduction to Neural Networks

Jean-Michel Marin

Faculty of Sciences, Montpellier

HAX912X - 2026/2027

Perceptron

- Simplest type of artificial neuron
- Input vector $x = [x_1, x_2, \ldots, x_n]$
- Weight vector $w = [w_1, w_2, \ldots, w_n]$
- Activation: $y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$
- Step function for binary classification

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

Limitations

Perceptron can only solve linearly separable problems ; for more complex problems, we need multilayer networks

Multilayer Networks and Activation Functions

- Multilayer Perceptron (MLP): A network of neurons organized in layers
- Hidden layers allow for the learning of complex patterns
- Activation functions introduce non-linearity

Common Activation Functions:

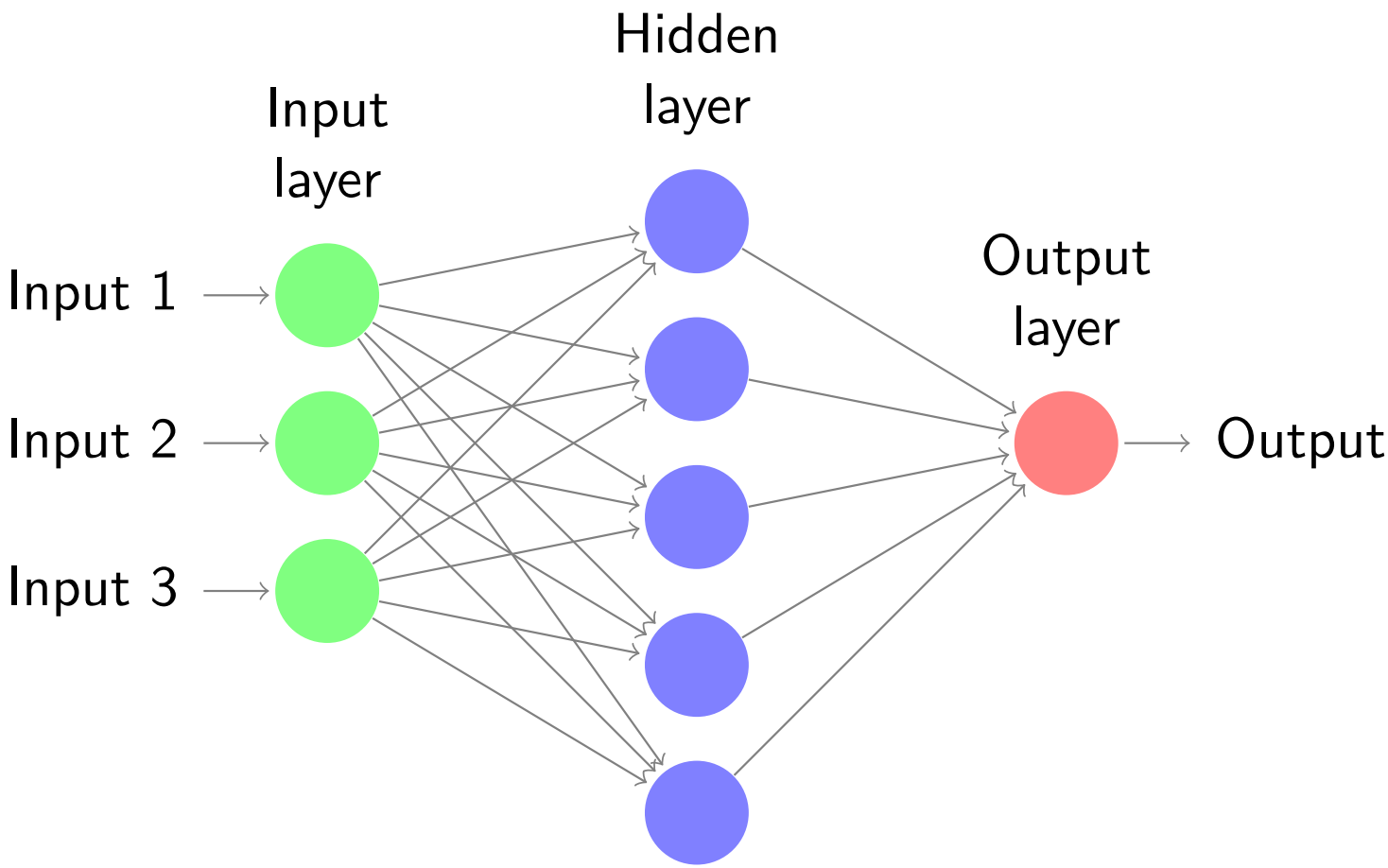
- Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic Tangent (tanh): $f(z) = \frac{2}{1+e^{-2z}} - 1$
- Rectified Linear Unit (ReLU): $f(z) = \max(0, z)$

Why Non-linear Activation?

Non-linear activation functions allow the model to capture more complex relationships in the data

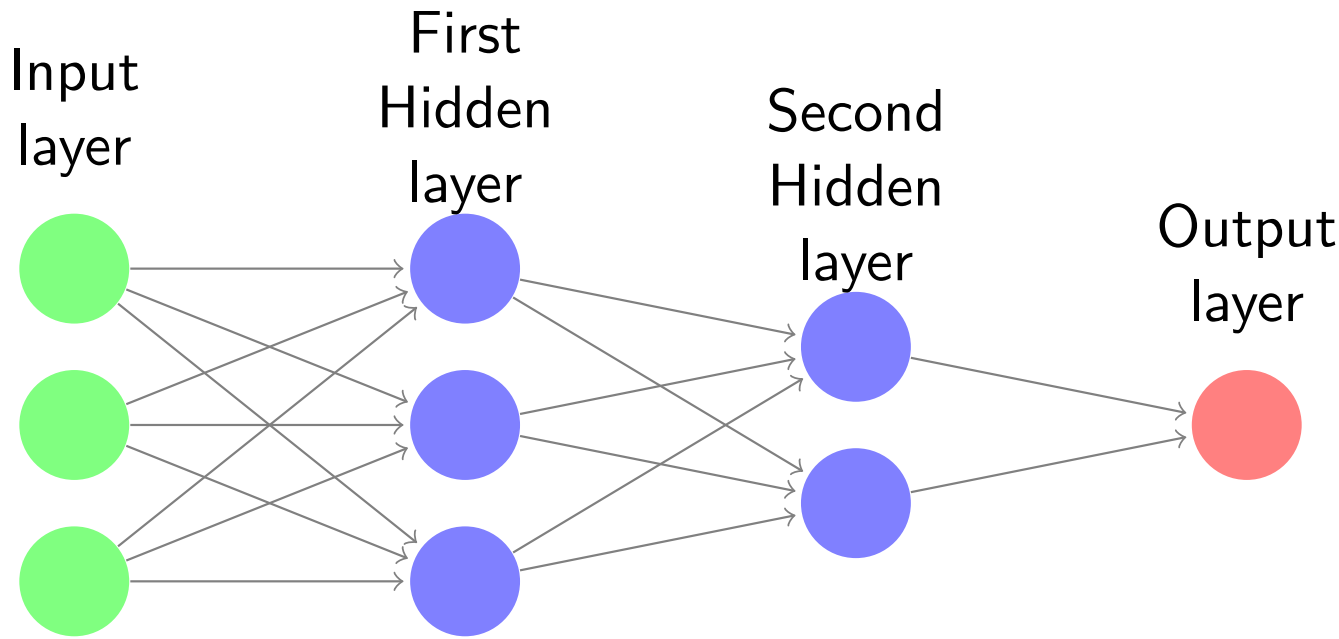
Neural Network Structure Example

- Example of a fully connected neural network with one hidden layer



Example with 2 hidden layers

- Example of a fully connected neural network with two hidden layers



Example with 2 hidden layers

$y \in \mathbb{R}$ response dimension 1, $x_1, x_2, x_3 \in \mathbb{R}$ 3 predictors
2 hidden layers: first layer with 3 neurons,
second layers with 2 neurons,
g, h, m 3 activation functions

$$n_{1,k} = g\left[\alpha_{0,k} + \sum_{l=1}^{3} \alpha_{l,k} x_l\right], \quad k = 1, 2, 3$$

$$n_{2,j} = h\left[\beta_{0,j} + \sum_{l=1}^{3} \beta_{l,k} n_{1,j}\right], \quad j = 1, 2$$

$$\hat{y} = m\left[\gamma_0 + \sum_{j=1}^{2} \gamma_j n_{2,j}\right]$$

Forward Propagation

- Inputs are passed through the network layer by layer
- At each layer, the input is multiplied by weights, and the activation function is applied
- Output layer produces final predictions

Example with 2 hidden layers

$\boldsymbol{\alpha}$ is a 4×3 matrix (weights matrix associated with the first hidden layer including the bias terms)
$\boldsymbol{\beta}$ is a 4×2 matrix (weights matrix associated with the second hidden layer including the bias terms)
$\boldsymbol{\gamma}$ is a 3×1 matrix (weights vector associated with the output layer including the bias term)

23 parameters

$$\hat{y}_x(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}) = m\left[\gamma_0 + \sum_{j=1}^{2} \gamma_j h\left[\beta_{0,j} + \sum_{k=1}^{3} \beta_{k,j} g\left[\alpha_{0,k} + \sum_{l=1}^{3} \alpha_{l,k} x_l\right]\right]\right]$$

Backpropagation Algorithm

- Objective: Minimize the error between predicted and actual outputs by adjusting weights
- Backpropagation is the algorithm used to compute the gradient of the loss function with respect to weights
- Gradient Descent is then used to update the weights

Example with 2 hidden layers

Loss function

$$l(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}; y, x) = [\hat{y}_x(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}) - y]^2$$

We observe an N-sample

$$(y, x)^{(1)}, \ldots, (y, x)^{(N)}$$

Objective: find

$$(\hat{\boldsymbol{\alpha}}, \hat{\boldsymbol{\beta}}, \hat{\boldsymbol{\gamma}}) \in \underset{(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma})}{\arg\min} \sum_{i=1}^{N} l(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}; x^{(i)}, y^{(i)})$$

Key Steps in Backpropagation

1. **Forward Pass**: Compute the predicted output through the network
2. **Compute the Loss**: Calculate the error or loss (e.g., Mean Squared Error for regression, Cross-Entropy for classification)
3. **Backward Pass**: Propagate the error backward through the network, calculating gradients for each weight
4. **Weight Update**: Use Gradient Descent to adjust weights

Intuition Behind Backpropagation

- The error at the output is backpropagated layer by layer
- Each neuron's contribution to the final error is computed, allowing targeted weight adjustments
- This allows the network to "learn" the optimal weights that reduce overall error

Chain Rule of Calculus

The partial derivatives in backpropagation rely on the chain rule to propagate errors backward through the layers

Overfitting and Regularization

- **Overfitting**: When a model learns not only the underlying patterns but also the noise in the training data
- **Regularization**: Techniques used to prevent overfitting by penalizing complex models

Common Regularization Techniques:

- L2 Regularization (Ridge): Adds a penalty proportional to the square of the weights
- Dropout: Randomly sets a fraction of the neurons to zero during training to prevent co-adaptation
- Early Stopping: Stop training when the validation error starts increasing

Tools

- Deep Learning: A subset of machine learning involving neural networks
- Two of the most popular frameworks:
  - **PyTorch** was originally developed by Facebook AI Research (FAIR), now part of Meta AI
    PyTorch is governed by the PyTorch Foundation, under the Linux Foundation (by Meta AI)
  - **TensorFlow** was developed by the Google Brain team and released by Google

PyTorch Code Example

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the model using nn.Sequential
model = nn.Sequential(
    nn.Linear(2, 5),      # First layer: 2 input features, 5
        neurons
    nn.ReLU(),            # ReLU activation after first layer
    nn.Linear(5, 1),      # Second layer: 5 neurons to 1
        output
    nn.Sigmoid()          # Sigmoid activation for binary
        classification
```

PyTorch Code Example

```
# Initialize the model, loss function and optimizer
criterion = nn.BCELoss()   # Binary cross entropy for binary
    classification
optimizer = optim.SGD(model.parameters(), lr=0.01)

data = torch.tensor([
    [25, 120],  # Age 25, Blood pressure 120
    [45, 140],  # Age 45, Blood pressure 140
    [35, 130],  # Age 35, Blood pressure 130
    [50, 160],  # Age 50, Blood pressure 160
    [60, 170],  # Age 60, Blood pressure 170
    [30, 115],  # Age 30, Blood pressure 115
    [55, 155],  # Age 55, Blood pressure 155
    [40, 135],  # Age 40, Blood pressure 135
    [65, 180],  # Age 65, Blood pressure 180
    [70, 190],  # Age 70, Blood pressure 190
    [32, 125],  # Age 32, Blood pressure 125
    [48, 150],  # Age 48, Blood pressure 150
    [58, 165],  # Age 58, Blood pressure 165
    [42, 145],  # Age 42, Blood pressure 145
    [38, 132]   # Age 38, Blood pressure 132
], dtype=torch.float32)
```

PyTorch Code Example

```
# Labels: 0 (no disease), 1 (disease)
labels = torch.tensor([
    [0], [1], [0], [1], [1],
    [0], [1], [0], [1], [1],
    [0], [1], [1], [1], [0]
], dtype=torch.float32)

# Training loop
for epoch in range(1000):
    optimizer.zero_grad()  # Zero gradients
    outputs = model(data)  # Forward pass
    loss = criterion(outputs, labels)  # Compute loss
    loss.backward()  # Backward pass
    optimizer.step()  # Update weights

    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Loss: {loss.item()}')
```

PyTorch Code Example

```
# Testing the model
test_data = torch.tensor([
    [40, 145],  # New test data: Age 40, Blood pressure 145
    [30, 110],  # Age 30, Blood pressure 110
    [50, 170],  # Age 50, Blood pressure 170
    [60, 160],  # Age 60, Blood pressure 160
], dtype=torch.float32)

with torch.no_grad():
    predictions = model(test_data)

print("\nPredictions (probabilities):")
print(predictions)

# Convert probabilities to binary predictions (threshold at
    0.5)
binary_predictions = (predictions >= 0.5).float()
print("\nBinary Predictions (0: No disease, 1: Disease):")
print(binary_predictions)
```

TensorFlow Code Example

```
import tensorflow as tf
import numpy as np

# Define the model using tf.keras.Sequential
model = tf.keras.Sequential([
    tf.keras.Input(shape=(2,)),              # Explicit Input
        layer with input shape (2,)
    tf.keras.layers.Dense(5, activation='relu'),  # First
        hidden layer with 5 neurons
    tf.keras.layers.Dense(1, activation='sigmoid')  # Output
         layer with 1 neuron, sigmoid activation
])
```

TensorFlow Code Example

```
# Compile the model with binary cross-entropy loss and Adam
    optimizer
model.compile(optimizer=tf.keras.optimizers.Adam(
    learning_rate=0.01),
                loss='binary_crossentropy')
# Expanded simulated dataset: [age, blood pressure] -> label
     (disease or not)
data = np.array([
    [25, 120],  # Age 25, Blood pressure 120
    [45, 140],  # Age 45, Blood pressure 140
    [35, 130],  # Age 35, Blood pressure 130
    [50, 160],  # Age 50, Blood pressure 160
    [60, 170],  # Age 60, Blood pressure 170
    [30, 115],  # Age 30, Blood pressure 115
    [55, 155],  # Age 55, Blood pressure 155
    [40, 135],  # Age 40, Blood pressure 135
    [65, 180],  # Age 65, Blood pressure 180
    [70, 190],  # Age 70, Blood pressure 190
    [32, 125],  # Age 32, Blood pressure 125
    [48, 150],  # Age 48, Blood pressure 150
    [58, 165],  # Age 58, Blood pressure 165
    [42, 145],  # Age 42, Blood pressure 145
    [38, 132]   # Age 38, Blood pressure 132
], dtype=np.float32)
```

TensorFlow Code Example

```
# Labels: 0 (no disease), 1 (disease)
labels = np.array([
    [0], [1], [0], [1], [1],
    [0], [1], [0], [1], [1],
    [0], [1], [1], [1], [0]
], dtype=np.float32)

# Training the model
epochs = 1000
model.fit(data, labels, epochs=epochs, verbose=0)  # verbose
    =0 suppresses output during training
```

TensorFlow Code Example

```
# Testing the model
test_data = np.array([
    [40, 145],  # New test data: Age 40, Blood pressure 145
    [30, 110],  # Age 30, Blood pressure 110
    [50, 170],  # Age 50, Blood pressure 170
    [60, 160],  # Age 60, Blood pressure 160
], dtype=np.float32)

# Make predictions
predictions = model.predict(test_data)
print("\nPredictions (probabilities):")
print(predictions)

# Convert probabilities to binary predictions (threshold at
    0.5)
binary_predictions = (predictions >= 0.5).astype(np.float32)
print("\nBinary Predictions (0: No disease, 1: Disease):")
print(binary_predictions)
```

PyTorch vs TensorFlow

- **PyTorch**
  - Pythonic and intuitive
  - Easy to experiment with and debug
  - Widely used in research and modern AI
  - Supports GPU acceleration (CUDA, MPS)

- **TensorFlow**
  - Mature deep learning framework
  - Strong tools for training and deployment
  - Supports CPU, GPU, and specialized accelerators (TPUs)
  - Closely associated with the Keras ecosystem

- **Today:** both are powerful frameworks for building, training, and deploying neural networks

The Need for GPUs

A **GPU (Graphics Processing Unit)** is a processor originally designed to accelerate graphics rendering, including images, videos, and animations

GPUs contain many processing units that can perform a large number of computations in parallel

CPUs are optimized for general-purpose computing and low-latency execution of complex tasks, whereas **GPUs** are optimized for high-throughput, massively parallel computations

As neural networks have grown larger and more computationally intensive, GPUs have become a key technology for training and running modern deep learning models

The Need for GPUs

Training modern **Large Language Models (LLMs)** requires enormous computational resources due to their large number of parameters and massive training datasets

A single GPU is generally not sufficient for training frontier-scale models. Training is therefore distributed across large clusters of **GPUs or other AI accelerators** working in parallel

Modern AI systems are trained on specialized **AI supercomputers**. For example, OpenAI has reported using Microsoft Azure AI supercomputing infrastructure to train its large-scale models

At this scale, efficient **distributed computing**, high-speed GPU interconnects, memory management, and energy consumption become as important as the individual GPUs themselves

The Cost of Training Large AI Models

Training modern **Large Language Models (LLMs)** requires enormous computational resources and can be extremely expensive

GPT-3 (2020) provides a useful historical reference

- 175 billion parameters ; 300 billion training tokens
- approximately 3.14×10^{23} **FLOPs** for training
- estimated compute cost: **several million dollars**

Since GPT-3, frontier models have become considerably more computationally demanding. Modern systems may combine **large-scale pre-training, post-training, reasoning, multimodal data, and reinforcement learning**

Recent frontier models are trained using large clusters of **GPUs or specialized AI accelerators**, together with high-speed networks, large-scale storage, and substantial power and cooling infrastructure

GPUs for Deep Learning

NVIDIA A100 (Ampere) was introduced in 2020 and became one of the most widely used GPUs for large-scale AI training and high-performance computing (HPC)

The A100 provides up to **80 GB of HBM2e memory** and includes specialized **Tensor Cores** designed to accelerate the matrix operations at the heart of deep learning

Since then, NVIDIA has introduced newer generations of AI accelerators:

- **H100/H200 – Hopper** (2022–2024)
- **B200/GB200 – Blackwell** (2024–2025)

Each generation provides substantially greater performance, memory capacity, and support for lower-precision numerical formats used in modern AI workloads

GPU Acceleration on Mac: MPS

Modern Macs with **Apple Silicon** (M1, M2, M3, M4, M5, ...) include an integrated GPU that can accelerate machine learning workloads

Apple provides the **Metal** API for GPU computing. **Metal Performance Shaders (MPS)** provides highly optimized GPU operations used by machine learning frameworks

PyTorch supports Apple GPUs through the `mps` device, providing an alternative to NVIDIA's `cuda` device

The need of GPU

```python
import torch
import time

# Choose device: "cuda" for NVIDIA, "mps" for Apple, or
    default to "cpu"
device = "cuda" if torch.cuda.is_available() else "mps" if
    torch.backends.mps.is_available() else "cpu"
print(f"Using device: {device}")

# GPU matrix multiplication
start_time = time.time()
a = torch.randn(10000, 10000, device=device)
b = torch.randn(10000, 10000, device=device)
torch.matmul(a, b)
torch.cuda.synchronize() if device == "cuda" else torch.mps.
    synchronize()
elapsed_time = time.time() - start_time
print(f"GPU ({device.upper()}) Time: {elapsed_time}")
```

The need of GPU

```
# CPU matrix multiplication for comparison
start_time = time.time()
a = torch.randn(10000, 10000)
b = torch.randn(10000, 10000)
torch.matmul(a, b)
elapsed_time = time.time() - start_time
print(f"CPU Time: {elapsed_time}")
```

Applications of Neural Networks

- **Generative AI:** text, images, audio, video, and code generation
- **Reasoning and Problem Solving:** mathematical reasoning, planning, coding, and multi-step decision-making
- **Natural Language Processing:** chatbots, translation, summarization, and search
- **Computer Vision:** image recognition, object detection, and medical imaging
- **Speech and Audio:** speech recognition, synthesis, and voice assistants
- **Recommender Systems:** e-commerce, streaming platforms, and social media
- **Finance:** fraud detection, credit risk, forecasting, and algorithmic trading

Applications of Neural Networks

- **Health Care:** diagnosis support, drug discovery, and personalized medicine
- **Autonomous Systems:** self-driving vehicles, robotics, and drones
- **Industry:** predictive maintenance, quality control, and process optimization
- **Science:** protein structure prediction, climate modeling, and scientific discovery