Introduction to Neural Networks

Jean-Michel Marin

Faculty of Sciences, Montpellier

October 2024

Course Overview

- Perceptron and its limitations
- Multilayer Networks
- Backpropagation Algorithm
- Key Applications

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:

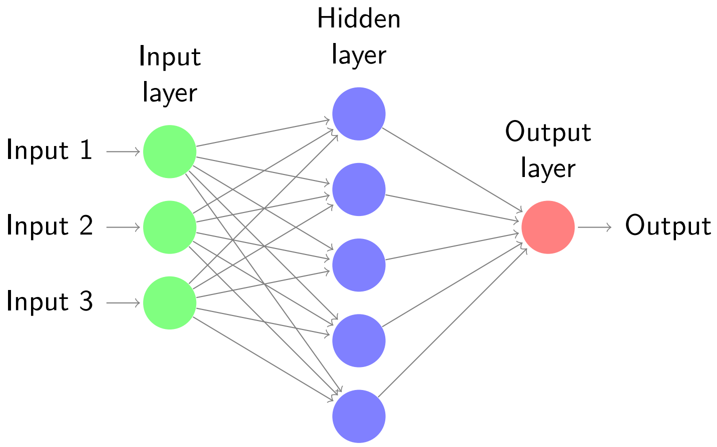
- Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic Tangent (tanh): $f(z) = \frac{2}{1+e^{-2z}} - 1$
- Rectified Linear Unit (ReLU): $f(z) = \max(0, z)$

Why Non-linear Activation?

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

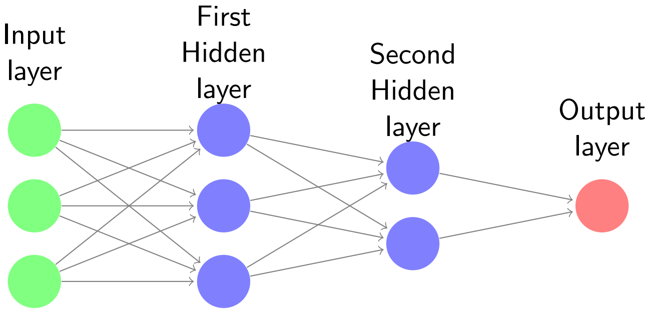
Neural Network Structure Example

- 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

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

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** (by Meta AI)
    - **TensorFlow** (by Google AI)

PyTorch Code Example

```python
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

```python
# 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

```python
# 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

# Display loss every 100 epochs
for epoch in range(100, epochs+1, 100):
    loss = model.evaluate(data, labels, verbose=0)
    print(f'Epoch [{epoch}/{epochs}], Loss: {loss:.4f}')
```

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)
```

Code Comparison: PyTorch vs TensorFlow

- **PyTorch:**
  - More Pythonic, natural control flow
  - Dynamic graph allows flexibility
  - Easy to debug with standard Python tools (e.g., pdb).
- **TensorFlow:**
  - Eager execution available, but TensorFlow traditionally uses static graphs
  - Strong deployment tools for production use (e.g., TensorFlow Serving, TensorFlow Lite)
  - Integrated with Keras for simplified model building

When to Use PyTorch

- Ideal for research and experimentation
- Preferred for projects that require fast prototyping
- Dynamic networks and tasks with high flexibility

When to Use TensorFlow

- Ideal for production environments
- Preferred for mobile and embedded applications (TensorFlow Lite)
- Suitable for large-scale models and distributed computing

The need of GPU

A GPU (Graphics Processing Unit) is a specialized processor designed primarily to accelerate the rendering of images, videos, and animations.

Originally developed for graphics tasks in video games, GPUs have evolved to handle parallel processing tasks, making them ideal for workloads like machine learning, AI, and scientific simulations.

Unlike CPUs, which are optimized for sequential tasks, GPUs excel at processing large amounts of data simultaneously. Modern GPUs are crucial for deep learning, data science, and high-performance computing applications.

The need of GPU

To train a network like ChatGPT, which is based on transformer models and deep learning, you would need an extensive setup of high-performance GPUs. The number of GPUs required depends on the model size, dataset, and training time.

For example, GPT-3 used around 285,000 GPU hours, and systems like this often leverage clusters of thousands of GPUs (e.g., NVIDIA A100s).

A realistic estimation for such large models could be hundreds to thousands of GPUs depending on the training goals and available infrastructure.

The need of GPU

The cost to train models like GPT-3 is extremely high due to the massive computational resources required.

Estimates place the training cost for GPT-3 at around 4.6 million for the complete training process of the model, from start to finish. This means the entire process of feeding data into the model and adjusting its parameters (weights and biases) to minimize the error and improve accuracy.

This includes the use of thousands of GPUs for weeks or months, massive electricity consumption, and the need for a robust infrastructure to handle both storage and computation. The final cost can vary depending on factors like hardware efficiency, energy prices, and research optimizations.

The need of GPU

The NVIDIA A100 is a high-performance graphics card designed specifically for data centers and AI workloads.

It's built on NVIDIA's Ampere architecture and offers up to 80 GB of high-bandwidth memory (HBM2e), enabling it to handle massive datasets.

It excels in tasks such as AI model training, deep learning, and high-performance computing (HPC). The A100 is often used in large-scale AI research, including training models like GPT, due to its ability to scale efficiently across multiple GPUs in a cluster.

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

In the context of Apple's Metal API, MPS (Metal Performance Shaders) is a framework designed to optimize performance for GPU-accelerated compute and machine learning tasks.

The synchronize() function ensures that all operations on the GPU are completed before the next step of the code proceeds.

In frameworks like PyTorch, GPU operations are asynchronous by default to improve performance, meaning they are queued and executed without blocking the CPU.

When you want to measure the execution time of GPU operations accurately or need to ensure that computations are completed before moving forward, synchronize() forces the program to wait until all GPU tasks are finished.

The need of GPU

```python
# 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}")
```

Some applications of Neural Networks

- Fraud detection
- Customer service
- Financial services
- Natural language processing (chatbots, language translators...)
- Facial recognition
- Self-driving vehicles
- Predictive analytics (forecasting revenue, product development, decision-making, manufacturing...)
- Recommender systems (streaming services, e-commerce, social media...)
- Health care
- Industrial