Diogo Ribeiro

ESMAD - Escola Superior de Média Arte e Design
Lead Data Scientist, Mysense.ai

dfr@esmad.ipp.pt 0009-0001-2022-7072

November 16, 2025

Outline

What is Deep Learning?

Deep Learning

A class of machine learning algorithms that use **multiple layers** of non-linear transformations to learn hierarchical representations from data.

Key characteristics:

- **Depth:** Many layers (hence "deep")
- **End-to-end learning:** Learn features automatically
- **Hierarchical representations:** Low-level $\rightarrow$ High-level
- **Scalability:** Performance improves with more data

Why now?

- Big Data: Massive datasets available
- Compute: GPUs, TPUs enable fast training
- Algorithms: Better architectures and training techniques

The Biological Inspiration

Artificial neurons inspired by biological neurons:

Biological Neuron:

- Dendrites: Receive signals
- Cell body: Processes signals
- Axon: Sends output
- Synapses: Connect neurons

Artificial Neuron:

- Inputs: $x_1, \ldots, x_n$
- Weights: $w_1, \ldots, w_n$
- Aggregation: $\sum w_i x_i + b$
- Activation: $f(\cdot)$

Important

Modern deep learning has moved beyond simple biological analogy!

- Backpropagation doesn't occur in brain
- Architectures are engineering solutions
- Biological plausibility not the goal

The Perceptron

Perceptron (Rosenblatt, 1958)

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f(\mathbf{w}^T \mathbf{x} + b) \tag{1}$$

where f is a step function: $f(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$

Learning rule:

$$w_i \leftarrow w_i + \eta(y_{\text{true}} - y_{\text{pred}})x_i \tag{2}$$

where η is the learning rate.

Limitations (Minsky & Papert, 1969):

- Can only learn linearly separable functions

Multi-Layer Perceptron (MLP)

Solution: Add hidden layers!

Feed-Forward Neural Network

Layer l:

$$\boldsymbol{z}^{(l)} = \mathbf{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{3}$$

$$\boldsymbol{a}^{(l)} = f(\boldsymbol{z}^{(l)}) \tag{4}$$

where:

- $\boldsymbol{a}^{(l)}$: Activations (outputs) of layer l
- $\mathbf{W}^{(l)}$: Weight matrix
- $\boldsymbol{b}^{(l)}$: Bias vector
- f: Activation function

Activation Functions

Non-linearity is crucial! Without it, network is just linear regression.

1. **Sigmoid:**

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

- Output: $(0, 1)$
- Smooth gradient
- Problem: Vanishing gradients

2. **Tanh:**

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{6}$$

- Output: $(-1, 1)$

3. **ReLU:**

$$\text{ReLU}(z) = \max(0, z) \tag{7}$$

- Most popular!
- No vanishing gradients
- Fast computation
- Problem: "Dead neurons"

4. **Variants:**
- **Leaky ReLU:** $\max(0.01z, z)$
- **ELU:** Smooth for $z < 0$

The Learning Problem

Goal: Minimize loss function $\mathcal{L}(\theta)$ over parameters θ.

Common loss functions:

- **Mean Squared Error (Regression):**

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{8}$$

- **Cross-Entropy (Classification):**

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \tag{9}$$

- **Binary Cross-Entropy:**

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}[y_i\log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{10}$$

Backpropagation

Efficiently compute gradients using chain rule!

Chain Rule

For composite function $f(g(x))$:

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx} \tag{11}$$

For neural networks:

Forward pass: Compute activations layer by layer

$$\boldsymbol{a}^{(l)} = f(\mathbf{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}) \tag{12}$$

Backward pass: Compute gradients layer by layer

$$\delta^{(l)} = (\mathbf{W}^{(l+1)})^T \delta^{(l+1)} \odot f'(\boldsymbol{z}^{(l)}) \tag{13}$$

$$\partial\mathcal{L} \qquad \delta^{(l)}(\boldsymbol{a}^{(l-1)})^T \tag{14}$$

Gradient Descent

Update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{15}$$

Variants:

- **Batch Gradient Descent:** Use full dataset
  - Accurate gradients
  - Slow for large datasets
- **Stochastic Gradient Descent (SGD):** One sample at a time
  - Fast updates
  - Noisy gradients
- **Mini-Batch SGD:** Batches of B samples
  - Balance between accuracy and speed
  - Most commonly used
  - Typical batch sizes: 32, 64, 128, 256

Advanced Optimizers

Momentum (Polyak, 1964):

$$\boldsymbol{v}_t = \beta \boldsymbol{v}_{t-1} + \nabla_\theta \mathcal{L}(\theta_t) \tag{16}$$

$$\theta_{t+1} = \theta_t - \eta \boldsymbol{v}_t \tag{17}$$

Accumulates velocity, smooths updates.

Adam (Kingma & Ba, 2015): Most popular!

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L} \tag{18}$$

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L})^2 \tag{19}$$

$$\hat{\boldsymbol{m}}_t = \boldsymbol{m}_t/(1 - \beta_1^t), \quad \hat{\boldsymbol{v}}_t = \boldsymbol{v}_t/(1 - \beta_2^t) \tag{20}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \tag{21}$$

- Adaptive learning rates per parameter
- Default: $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

Why Regularization?

The Overfitting Problem

Deep networks have millions of parameters $\rightarrow$ Can memorize training data!

Consequences:

- Perfect training accuracy
- Poor generalization to test data
- Unreliable predictions

Regularization techniques:

1. L1/L2 weight penalties
2. Dropout
3. Batch normalization
4. Data augmentation
5. Early stopping

L1 and L2 Regularization

Add penalty term to loss function:

L2 (Ridge):

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_i w_i^2 \tag{22}$$

- Shrinks weights towards zero
- Smooth, differentiable
- Most common in deep learning

L1 (Lasso):

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_i |w_i| \tag{23}$$

- Encourages sparsity
- Some weights $\rightarrow$ exactly zero
- Acts as feature selection

Dropout

Dropout (Srivastava et al., 2014)

During training, randomly set a fraction p of neurons to zero.
Training:

$$\boldsymbol{a}^{(l)} = f(\boldsymbol{z}^{(l)}) \odot \boldsymbol{m}, \quad \boldsymbol{m} \sim \text{Bernoulli}(1-p) \tag{24}$$

Testing: Use all neurons, scale by $(1-p)$

Why does it work?

- Ensemble effect: Training many "thinned" networks
- Prevents co-adaptation of neurons
- Forces redundant representations

Typical values: $p = 0.5$ for hidden layers, $p = 0.1$ for input

Batch Normalization

Batch Normalization (Ioffe & Szegedy, 2015)

Normalize activations across mini-batch:

$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} \boldsymbol{z}_i \tag{25}$$

$$\sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B} (\boldsymbol{z}_i - \mu_B)^2 \tag{26}$$

$$\hat{\boldsymbol{z}}_i = \frac{\boldsymbol{z}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{27}$$

$$\boldsymbol{y}_i = \gamma \hat{\boldsymbol{z}}_i + \beta \tag{28}$$

where γ, β are learned parameters.

Why CNNs for Images?

Fully-connected networks don't scale:

- 224×224 RGB image: 150,528 inputs
- 1000 hidden units: 150M parameters (just first layer!)
- Ignores spatial structure

Key principles of CNNs:

1. **Local connectivity:** Neurons connect to small regions
2. **Parameter sharing:** Same weights applied everywhere
3. **Translation equivariance:** Shift input $\rightarrow$ Shift output

Biological Inspiration

Based on Hubel & Wiesel's (1962) work on cat visual cortex:

- Simple cells: Detect edges
- Complex cells: Invariant to position

Convolution Operation

2D Convolution

$$(I * K)[i,j] = \sum_m \sum_n I[i+m, j+n] \cdot K[m,n] \tag{29}$$

where I is input image, K is kernel (filter).

Hyperparameters:

- **Kernel size:** Usually 3×3 or 5×5
- **Stride:** Step size (usually 1 or 2)
- **Padding:** "same" or "valid"
- **Number of filters:** Determines output depth

Output size:

$$O = \left\lfloor \frac{I + 2P - K}{S} \right\rfloor + 1 \tag{30}$$

CNN Architecture Components

Typical CNN structure:

1. **Convolutional Layers:**
   - Apply learned filters
   - Extract local features
   - Multiple filters $\rightarrow$ Multiple feature maps
2. **Pooling Layers:**
   - Reduce spatial dimensions
   - **Max pooling:** Take maximum in each region
   - **Average pooling:** Take average
   - Provides translation invariance
3. **Fully-Connected Layers:**
   - At the end of network
   - Combine features for final prediction

Example: `Conv → ReLU → Pool → Conv → ReLU → Pool → FC → Softmax`

Classic CNN Architectures

1. **LeNet-5 (LeCun, 1998):**
   - First successful CNN
   - Digit recognition (MNIST)
   - 7 layers, 60K parameters
2. **AlexNet (Krizhevsky et al., 2012):**
   - ImageNet breakthrough (top-5 error: 15.3%)
   - 8 layers, 60M parameters
   - ReLU, dropout, data augmentation
   - Trained on GPUs
3. **VGGNet (Simonyan & Zisserman, 2014):**
   - Very deep (16-19 layers)
   - Small 3×3 filters throughout
   - 138M parameters
4. **ResNet (He et al., 2015):**
   - Residual connections enable very deep networks (50-152 layers)
   - Skip connections: $F(x) + x$

Sequential Data

Many problems involve sequences:

- Text: Words in sentences
- Speech: Audio over time
- Time series: Stock prices, weather
- Video: Frames over time

Challenge: Variable-length inputs and outputs

RNN solution: Maintain hidden state that captures history

$$\boldsymbol{h}_t = f(\mathbf{W}_{hh}\boldsymbol{h}_{t-1} + \mathbf{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h) \tag{31}$$

Key idea: Share parameters across time steps!

RNN Architectures

Different input/output configurations:

1. **One-to-One:** Standard neural network
2. **One-to-Many:** Image captioning (image $\rightarrow$ sequence)
3. **Many-to-One:** Sentiment analysis (sequence $\rightarrow$ label)
4. **Many-to-Many (same length):** Video classification
5. **Many-to-Many (different length):** Machine translation

Challenges:

- **Vanishing gradients:** Hard to learn long-term dependencies
- **Exploding gradients:** Unstable training
- **Solution:** LSTM and GRU cells

LSTM (Long Short-Term Memory)

Hochreiter & Schmidhuber (1997) - Addresses vanishing gradients

Key components:

- **Cell state $\boldsymbol{c}_t$:** Long-term memory highway
- **Gates:** Control information flow

LSTM equations:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\mathbf{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) \quad \text{(forget gate)} && (32)\\
\boldsymbol{i}_t &= \sigma(\mathbf{W}_i[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) \quad \text{(input gate)} && (33)\\
\tilde{\boldsymbol{c}}_t &= \tanh(\mathbf{W}_c[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c) \quad \text{(candidate)} && (34)\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \quad \text{(cell state)} && (35)\\
\boldsymbol{o}_t &= \sigma(\mathbf{W}_o[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) \quad \text{(output gate)} && (36)\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) \quad \text{(hidden state)} && (37)
\end{aligned}$$

Attention is All You Need

Vaswani et al. (2017) - Revolutionary architecture

Problems with RNNs:

- Sequential computation (can't parallelize)
- Long-range dependencies still challenging
- Slow training

Transformer solution:

- **Self-attention:** Every position attends to all positions
- **Parallelizable:** No sequential dependency
- **Positional encoding:** Add position information

Impact

Transformers now dominate NLP (BERT, GPT) and expanding to vision (ViT), audio, and more!

Self-Attention Mechanism

Scaled Dot-Product Attention

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{38}$$

where:

- $\mathbf{Q}$: Query matrix (what we're looking for)
- $\mathbf{K}$: Key matrix (what's available)
- $\mathbf{V}$: Value matrix (actual content)
- d_k: Dimension of keys (for scaling)

Multi-Head Attention:

- Run attention mechanism multiple times in parallel
- Each "head" learns different patterns
- Concatenate and project outputs

Transformer Architecture

Encoder-Decoder structure:

Encoder (left side):

1. Multi-head self-attention
2. Add & Normalize
3. Feed-forward network
4. Add & Normalize
5. (Repeat N times)

Decoder (right side):

1. Masked multi-head self-attention
2. Add & Normalize
3. Cross-attention to encoder
4. Add & Normalize
5. Feed-forward network
6. Add & Normalize
7. (Repeat N times)

Transformer Variants

Major models based on Transformers:

BERT (Devlin et al., 2018):

- Encoder-only
- Bidirectional context
- Pre-training: Masked language modeling
- Fine-tuning for downstream tasks

GPT (Radford et al., 2018-2023):

- Decoder-only
- Autoregressive generation
- Scaling to 175B+ parameters (GPT-3)
- In-context learning

Vision Transformer (Dosovitskiy et al., 2020):

- Apply Transformers to images
- Split image into patches
- Competitive with CNNs on large datasets

Training Best Practices

1. **Data Preprocessing:**
   - Normalize inputs (zero mean, unit variance)
   - Data augmentation (images: flip, crop, rotate)
   - Handle class imbalance
2. **Initialization:**
   - **Xavier/Glorot:** For tanh/sigmoid
   - **He initialization:** For ReLU
   - Never initialize all weights to zero!
3. **Learning Rate:**
   - Start with default values (0.001 for Adam)
   - Learning rate schedules (decay, cyclical)
   - Warm-up for large batches
4. **Batch Size:**
   - Larger batches: Faster, more stable
   - Smaller batches: Better generalization
   - Trade-off with GPU memory

Debugging Neural Networks

Common issues and solutions:

1. **Loss not decreasing:**
   - Check learning rate (too high or too low)
   - Verify gradient flow (print gradient norms)
   - Check data preprocessing
2. **Loss exploding:**
   - Reduce learning rate
   - Gradient clipping
   - Check for bugs in loss computation
3. **Overfitting:**
   - Add regularization (dropout, L2)
   - More data or data augmentation
   - Reduce model capacity
4. **Underfitting:**
   - Increase model capacity
   - Train longer
   - Reduce regularization

Summary

Key concepts covered:

- **Foundations:** Neurons, activation functions, backpropagation
- **Optimization:** SGD, Adam, learning rate schedules
- **Regularization:** Dropout, batch normalization, weight decay
- **CNNs:** Convolution, pooling, ResNet
- **RNNs:** LSTM, sequence modeling
- **Transformers:** Attention, BERT, GPT

The Deep Learning Revolution

- Transforming AI and industry
- Rapid progress in architectures and applications
- Democratization through frameworks (PyTorch, TensorFlow)
- Still many open questions!

Further Reading

Books:

- Goodfellow, Bengio, Courville (2016). *Deep Learning*
- Zhang et al. (2023). *Dive into Deep Learning*

Courses:

- Stanford CS231n (CNNs for Visual Recognition)
- Stanford CS224n (NLP with Deep Learning)
- fast.ai Practical Deep Learning

Key Papers:

- Krizhevsky et al. (2012). "ImageNet Classification with Deep CNNs" (AlexNet)
- He et al. (2015). "Deep Residual Learning" (ResNet)
- Vaswani et al. (2017). "Attention Is All You Need" (Transformer)

Acknowledgments

- ESMAD for institutional support
- Mysense.ai for deep learning applications
- Deep learning research community

Contact Information

Diogo Ribeiro

ESMAD - Escola Superior de Média Arte e Design
Lead Data Scientist, Mysense.ai

dfr@esmad.ipp.pt

0009-0001-2022-7072

github.com/diogoribeiro7

linkedin.com/in/diogoribeiro7