Neural Networks - Part 1

Dr. Daniele Cattaneo, Prof. Dr. Josif Grabocka

Machine Learning Course
Winter Semester 2023/2024

Albert-Ludwigs-Universität Freiburg

cattaneo@informatik.uni-freiburg.de, grabocka@informatik.uni-freiburg.de

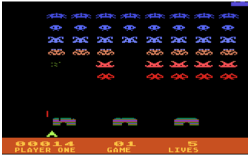
November 20, 2023

Table of Contents

Motivation: Deep Learning

- Excellent empirical results, e.g., in reasoning in games

  - Superhuman performance in playing Atari games [Mnih et al, Nature 2015]



  - Beating the world's best Go player [Silver et al, Nature 2016]

- **Deep Learning** is an umbrella term for:
  - Neural Network architectures
  - Regularization approaches for Neural Networks
  - Optimization Techniques for Neural Networks
  - Large-scale training of Neural Networks, etc.

What is a Deep Forward Network (DFN)?

- **Feedforward networks**, **feedforward neural networks** or **multilayer perceptrons**

- Given a function $y = f^*(x)$ that maps input x to category y

- A DFN defines a parametric mapping $\hat{y} = f(x; w)$ with parameters w

- Aim is to learn w such as $f(x; w)$ *best* approximates $f^*(x)$!

Neural Networks are Composite Functions

- Each k-th neuron is a simple function:

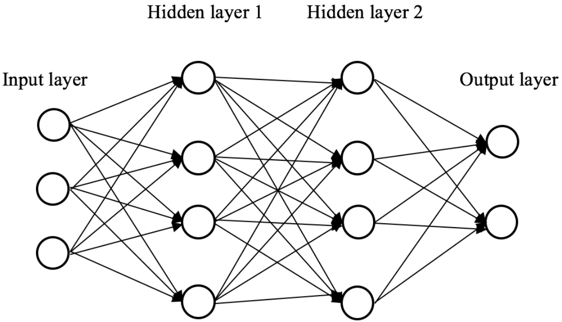
$$h^{(k)}(x_1, \dots, x_M) = g\left(w_0^{(k)} + \sum_{i=1}^{M} w_i^{(k)} x_i\right)$$

- Inputs to neurons can be the outputs of predecessor neurons
  - The indices of the M predecessor neurons to the k-th neuron are defined as the sequence $P^{(k)} = \{P_1^{(k)}, \cdots, P_M^{(k)}\} \in \mathbb{N}^M$

$$h^{(k)}\left(h^{\left(P_1^{(k)}\right)}, \dots, h^{\left(P_M^{(k)}\right)}\right) = g\left(w_0^{(k)} + \sum_{i=1}^{M} w_i^{(k)} h^{\left(P_i^{(k)}\right)}\right)$$

- **Input Neurons:** For a set of neurons the predecessors are the input features x
- **Output Neurons:** The output of a set of neurons estimates the target $\hat{y}$

Neural Networks are Composite Functions



DFN are "functions of functions of ... of functions of x":

$$\hat{y} := g^{(k)}\left(g^{(l)}\left(\ldots\left(g^{(n)}(x, w^{(n)})\ldots\right), w^{(l)}\right), w^{(k)}\right), \quad k > l > n$$

Why *Feedforward*?

- Given a Feedforward Network $\hat{y} = f(x; w)$
  - Input x, then pass through a chain of steps before outputting $\hat{y}$

- No feedback exists between the chains of steps
  - Feedback connections yield the **Recurrent Neural Network**

- Example $f^{(1)}(x)$, $f^{(2)}(x)$ and $f^{(3)}(x)$ can be chained as:
  - $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$
    - $f^{(1)}$ is the first layer, or the **input** layer
    - $f^{(2)}$ is the second layer, or a **hidden** layer
    - $f^{(3)}$ is the last layer, or the **output** layer
- Number of hidden layers define the **depth** of the network
- Dimensionality of the hidden layers defines the **width** of the network

Why *Neural*?

- Loosely inspired by neuroscience, hence Artificial Neural Network
- Each hidden layer node resembles a neuron
- Input to a neuron are the synaptic connections from the previous attached neuron
- Output of a neuron is an aggregation of the input vector
- Signal propagates forward in a chain of "Neuron"-to-"Neuron" transmissions
- However, modern Deep Learning research is steered mainly by mathematical and engineering principles!

Why *Network*?

- A feed-forward network is an **acyclic directed graph**, but
  - Graph nodes are structured in layers
  - Directed links between nodes are **parameters/weights**
  - Each node is a computational functions
  - No inter-layer and intra-layer connections (but possible)
  - Input to the first layer is given (the features x)
  - Output is the computation of the last layer (the target $\hat{y}$)

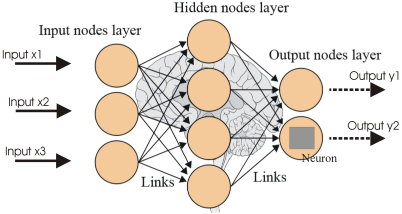


Figure 1: DFN, Source `www.analyticsvidhya.com`

Nonlinear Mapping

- We can easily solve linear regression, but not every problem is linear.
- Can the function $f(x) = (x+1)^2$ be approximated through a linear function?

Nonlinear Mapping

- We can easily solve linear regression, but not every problem is linear.
- Can the function $f(x) = (x+1)^2$ be approximated through a linear function?
- Yes, but only if we **map** the feature x into a new space:

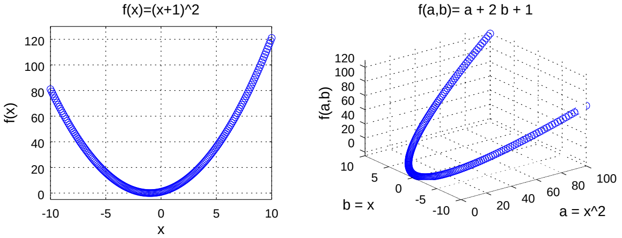


Figure 2: Mapping feature x into a new dimensionality $x \rightarrow \phi(x) = (a, b)$

Nonlinear Mapping (II)

- Which mapping $\phi(x)$ is the best?

There are various ways of designing $\phi(x)$:

1. Hand-craft (manually engineered) $\phi(\mathrm{x})$
2. Use a very generic $\phi(\mathrm{x})$, RBF or polynomial expansion
3. Parametrize and learn the mapping $\mathrm{f}(\mathrm{x}; \theta, \mathrm{w}) := \phi(\mathrm{x}, \theta)^{\mathsf{T}}\mathrm{w}$

Deep Forward Networks follow the third approach, where:

- the hidden layers (weights θ) learn the mapping $\phi(x, \theta)^T$
- the output layer (weights w) learns the function $f(x; \theta, w)$

Layered DFN

A DFN with L hidden layers:

$$
\begin{aligned}
h^{(1)} &= g^{(1)}({w^{(1)}}^T x + w_0^{(1)}) \\
h^{(2)} &= g^{(2)}({w^{(2)}}^T h^{(1)} + w_0^{(2)}) \\
&\cdots \\
h^{(L)} &= g^{(L)}({w^{(L)}}^T h^{(L-1)} + w_0^{(L)}) \\
\hat{y} &= h^{(L)}
\end{aligned}
$$

Different layers can have different activation functions $g^{(i)}$.

Layered DFN - Forward Step

Let $M^{(\ell)}$ be the number of neurons at the ℓ-th layer:

$$\begin{aligned} w^{(\ell)} &\in \mathbb{R}^{M^{(\ell-1)} \times M^{(\ell)}} \\ w_0^{(\ell)} &\in \mathbb{R}^{M^{(\ell)}} \\ h^{(\ell)} &\in \mathbb{R}^{M^{(\ell)}} \end{aligned}$$

The activation of the j-th neuron of the ℓ-th layer when inputted the n-th data point x_n is:

$$h_{n,j}^{(\ell)} = g^{(\ell)} \left(w_{0,j}^{(\ell)} + \sum_{i=1}^{M^{(\ell-1)}} {w^{(\ell)}}^T_{j,i} \, h_{n,i}^{(\ell-1)} \right)$$

$$\forall j \in \left\{1, \ldots, M^{(\ell)}\right\}, \ \forall n \in \{1, \ldots, N\}$$

$$\text{where } h_n^{(0)} := x_n \in \mathcal{X}, \ M^{(0)} = |\mathcal{X}|, \ \hat{y}_n = h_n^{(L)}$$

An example - XOR

- XOR is a function:

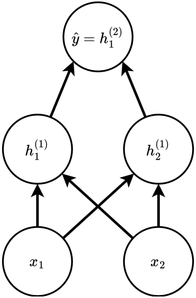
x_1	x_2	$y = f^*(x)$
0	0	0
0	1	1
1	0	1
1	1	0

- Can we learn a DFN $\hat{\mathrm{y}} = \mathrm{f}(\mathrm{x}; \mathrm{w})$ such that f resembles f^*?
- Our dataset $\mathcal{X} = \{[0,0]^T, [1,0]^T, [0,1]^T, [1,1]^T\}$
- Leading to the optimization:

$$\begin{aligned} \underset{w}{\operatorname{argmin}} \quad & J(\theta) \\ J(\theta) \quad = \quad & \frac{1}{4} \sum_{x \in \mathcal{X}} (f^*(x) - f(x; w))^2 \end{aligned}$$

An example - XOR (2)

- We will learn a simple DFN with one hidden layer:



- Chained $h^{(1)} = f^{(1)}(x; w^{(1)})$ and $h^{(2)} = \hat{y} = f^{(2)}(h^{(1)}; w^{(2)})$
  - Hidden-layer: $h_{n,j}^{(1)} = g^{(1)}\left({w_{j,:}^{(1)}}^T x_n + w_{0,j}^{(1)}\right), \forall\, j \in \{1, 2\}$
  - Output layer: $\hat{y}_n = h_{n,1}^{(2)} = {w_{1,:}^{(2)}}^T h_n^{(1)} + w_{0,1}^{(2)}$
  - $w^{(1)} \in \mathbb{R}^{2\times 2}, w_0^{(1)} \in \mathbb{R}^{2\times 1}, w^{(2)} \in \mathbb{R}^{2\times 1}, w_0^{(2)} \in \mathbb{R}$

Rectified Linear Unit

The rectified linear unit (ReLU) is defined by the activation function $g(z) = \max\{0, z\}$, i.e.:

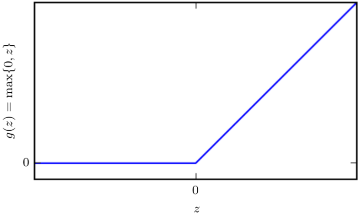


Figure 3: The ReLU activation, Source: Goodfellow et al., 2016

Yielding the overall function:

$$\hat{y} = w^{(2)^T} \max\left\{0, w^{(1)^T} x + w_0^{(1)}\right\} + w_0^{(2)}$$

"Deus ex machina" solution?

Suppose I magically found out that:

$$w^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},\ w_0^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},\ w^{(2)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},\ w_0^{(2)} = 0$$

We would later on see an optimization technique called Stochastic Gradient Descent with Backpropagation to learn the network parameters.

XOR Solution - Hidden Layer Computations

$$
\begin{aligned}
h^{(1)}_{1,1} &= g\left({w^{(1)}}^T_{1,:} x_1 + w^{(1)}_{0,1}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + 0\right) = g\left(0\right) = 0 \\
h^{(1)}_{1,2} &= g\left({w^{(1)}}^T_{2,:} x_1 + w^{(1)}_{0,2}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} - 1\right) = g\left(-1\right) = 0 \\
h^{(1)}_{2,1} &= g\left({w^{(1)}}^T_{1,:} x_2 + w^{(1)}_{0,1}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} + 0\right) = g\left(1\right) = 1 \\
h^{(1)}_{2,2} &= g\left({w^{(1)}}^T_{2,:} x_2 + w^{(1)}_{0,2}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} - 1\right) = g\left(0\right) = 0 \\
h^{(1)}_{3,1} &= g\left({w^{(1)}}^T_{1,:} x_3 + w^{(1)}_{0,1}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + 0\right) = g\left(1\right) = 1 \\
h^{(1)}_{3,2} &= g\left({w^{(1)}}^T_{2,:} x_3 + w^{(1)}_{0,2}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} - 1\right) = g\left(0\right) = 0 \\
h^{(1)}_{4,1} &= g\left({w^{(1)}}^T_{1,:} x_4 + w^{(1)}_{0,1}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + 0\right) = g\left(2\right) = 2 \\
h^{(1)}_{4,2} &= g\left({w^{(1)}}^T_{2,:} x_4 + w^{(1)}_{0,2}\right) = g\left(\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} - 1\right) = g\left(1\right) = 1
\end{aligned}
$$

XOR Solution - Output Layer Computations

$$
\begin{aligned}
\hat{y}_1 &= h^{(2)}_{1,1} = w^{(2)\,T} h^{(1)}_{1,:} + w^{(2)}_{0,1} = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + 0 = 0 \\
\hat{y}_2 &= h^{(2)}_{2,1} = w^{(2)\,T} h^{(1)}_{2,:} + w^{(2)}_{0,1} = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 = 1 \\
\hat{y}_3 &= h^{(2)}_{3,1} = w^{(2)\,T} h^{(1)}_{3,:} + w^{(2)}_{0,1} = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 = 1 \\
\hat{y}_4 &= h^{(2)}_{4,1} = w^{(2)\,T} h^{(1)}_{4,:} + w^{(2)}_{0,1} = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} + 0 = 0
\end{aligned}
$$

The computations of the final layer match exactly those of the XOR function.

Types of Hidden Units

- Question: Can we use a linear activation $h = w^T x + b$?

Types of Hidden Units

- Question: Can we use a linear activation $h = w^T x + b$?
- Remember the most used hidden layer is ReLU:

$$h = g(w^T x + b) = \max(0, w^T x + b)$$

- Alternatively, the sigmoid function:

$$h = \sigma(z) = \frac{e^z}{e^z + 1}$$

- or, the hyperbolic tangent:

$$h = \tanh(z) = 2\sigma(2z) - 1$$

Table of Contents

Continuous Target - Regression

- Output layer is an affine transformation with no nonlinearity
  - Given features h, produces $\hat{y} = w^T h + b$
- Used to produce the mean of a conditional Gaussian distribution
  - $p(y \mid x) = \mathcal{N}(y; \hat{y}, I)$

Regression Loss

The loss/cost can be expressed in probabilistic terms as

$$J(\theta) = -\mathbb{E}_{(x,y)\sim\hat{p}_{data}} \log p_{\text{model}}(y \mid x)$$

Assuming normality $p_{\text{model}}(y \mid x) = \mathcal{N}(y; f(x;\theta), I)$:

$$J(\theta) = \frac{1}{2}\mathbb{E}_{(x,y)\sim\hat{p}_{data}} ||y - f(x;\theta)||^2 + \text{const}$$

Solving for the optimal DFN parameters:

$$\theta^{\text{opt}} =: \underset{\theta}{\text{argmin}}\, \mathbb{E}_{(x,y)\sim\hat{p}_{data}} ||y - f(x;\theta)||^2$$

Yields an estimation: $f(x, \theta^{\text{opt}}) = \mathbb{E}_{x,y\sim\hat{p}_{data}(y|x)}[y]$

Binary Classification Target

- Binary target variables follow a Bernoulli distribution
  $P(y = 1) = p, \; P(y = 0) = 1 - p$
- Train a DFN such that $\hat{y} = f(x; w) \in [0, 1]$
- Naive Option: Clip a linear output layer:
  - $P(y = 1 \mid x) = \max\left\{0, \min\left\{1, w^T h + b\right\}\right\}$
- What is the problem with the clipped linear output layer?

Binary Classification Target

- Use a smooth sigmoid output unit:

$$
\begin{aligned}
\hat{y} &= \sigma(z) = \frac{e^z}{e^z + 1} \\
z &= w^T h + b
\end{aligned}
$$

- The loss for a DFN $f(x, \theta)$ with a sigmoid output is:

$$
J(w) = \sum_{n=1}^{N} -y_n \log(f(x_n, w)) - (1 - y_n) \log(1 - f(x_n, w))
$$

- Also called as Logistic Loss or the Cross-entropy

Multi-category Target

- For multi-category targets $\hat{y}_i = P(y = i|x), \; i \in \{1, \dots, C\}$
- Let the unnormalized log probability be defined as

$$
\begin{aligned}
z_i &= w_i^T h + b \\
z_i &= \log \; \tilde{P}(y = i|x)
\end{aligned}
$$

- Yielding the normalized probability estimation:

$$
P(y = i|x) \approx \mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}
$$

- Minimizing the log-likelihood loss:

$$
\begin{aligned}
J(w) &= \sum_{n=1}^{N} \sum_{i=1}^{C} -1_{y_n=i} \log P(y = i|x) \\
J(w) &= -\sum_{n=1}^{N} \sum_{i=1}^{C} 1_{y_n=i} \left(z_i - \log \sum_j e^{z_j} \right)
\end{aligned}
$$

How to train w for minimizing the loss?

Minimize $J(w)$ by updating w in the negative direction of $\frac{\partial J(w)}{\partial w}$:

$$w^{(\text{next})} \leftarrow w^{(\text{prev})} - \eta \frac{\partial J(w)}{\partial w^{(\text{prev})}}$$

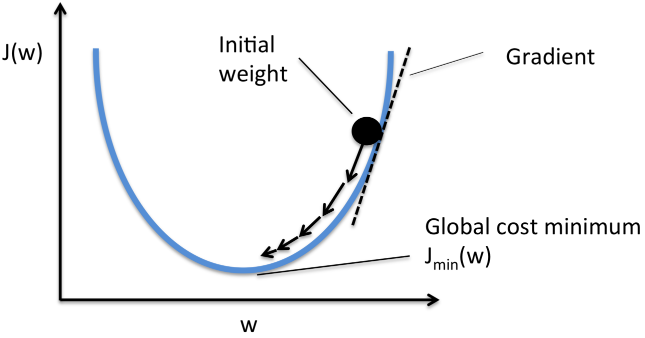


Figure 4: Source: `http://rasbt.github.io/`

Gradient Descent

Find the optimal parameters $w^* \in \mathbb{R}^K$ that minimize an objective function $J(w)$, given data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, i.e.:

$$w^* \;:=\; \underset{w}{\operatorname{argmin}} \; J(\mathcal{D}, w)$$

Algorithm 1: Gradient Descent Optimization

Require: Data $\mathcal{D}$, Learning rate $\eta \in \mathbb{R}^+$, Iterations $\mathcal{I} \in \mathbb{N}^+$

Ensure: $w \in \mathbb{R}^K$

1: $w \sim \mathcal{N}(0, \sigma^2)$

2: **for** $1, \ldots, \mathcal{I}$ **do**

3: $\quad w \leftarrow w - \eta \frac{\partial J(\mathcal{D}, w)}{\partial w}$

4: **return** w

Stochastic Gradient Descent

Divide the dataset into R partitions (mini-batches) as $\mathcal{D} = \bigcup_{r=1}^{R} \mathcal{D}_r$, yielding a decomposition of the loss:

$$J(\mathcal{D}, w) \;:=\; \sum_{r=1}^{R} J(\mathcal{D}_r, w) \;:=\; \sum_{r=1}^{R} J_r$$

Algorithm 2: Stochastic Gradient Descent Optimization

Require: Data $\mathcal{D} = \bigcup_{r=1}^{R} \mathcal{D}_r$, Learning rate $\eta \in \mathbb{R}^+$, Iters $\mathcal{I} \in \mathbb{N}^+$

Ensure: $w \in \mathbb{R}^K$

1: $w \sim \mathcal{N}(0, \sigma^2)$
2: **for** $1, \ldots, \mathcal{I}$ **do**
3:     **for each** $r \in \{1, \ldots, R\}$ *in random order* **do**
4:         $w \leftarrow w - \eta \frac{\partial J_r}{\partial w}$
5: **return** w

Next step

How to compute $\frac{\partial J(w)}{\partial w}$ for the all the weights of a DFN?

Backpropagation (next lecture)...