Neural Networks

Machine Learning 2022-23
UML book chapter 20
Slides: F. Chiariotti, P. Zanuttigh, F. Vandin
Artificial Neural Networks

- Model of computation inspired by the structure of neural networks in the brain
- Large number of basic computing devices (neurons) connected to each other
- Neural Networks (NN) are represented with directed graphs where the nodes are the neurons and the edges correspond to the links between the neurons
- Firstly proposed in 1940-50
- First practical applications in the 80-90s but practical results were lower than SVM and other techniques
- From 2010 on deep architectures with impressive performances
From Simple Algorithms to Deep Learning

- **Data**
- **Prior**
  - expert system
  - use prior knowledge to construct $\phi(x)$ and learn $\langle w, \phi(x) \rangle$
  - less prior knowledge, more data
- **No Free Lunch**
- **Deep Networks**
Feedforward network: the graph representing the network has no cycles (data flows only in one direction)

The network is typically organized into layers: each neuron takes in input only the output of neurons of the previous layer

Notation (NN): Graph $G=(V,E)$ and function $w: E \rightarrow \mathbb{R}$

- $V$: neurons ($|V|$ is the size of the network)
- $E$: connections between neurons (directed edges)
- $w: E \rightarrow \mathbb{R}$ weight function over the edges (the weights $w$ are the parameters to be learned)

Each neuron:
1. Takes in input the sum of the outputs of the connected neurons from previous layer weighted by the edge weights ($w$)
2. Applies to the result a simple scalar function (activation function, $\sigma$)
Represent a network as the union of a set of (disjoint) layers: $V = \bigcup_{t=0}^{T} V_t$

- $V_t$, $t = 0, \ldots, T$: $t$-th layer,
- $d_t + 1$ number of nodes of layer $t$
  - "+1": constant neuron (avoid bias, incorporate as in homogenous coord.)
- $V_0$: input layer, $V_T$: output layer, $V_1, \ldots, V_{T-1}$ inner (hidden) layers
- $T$: depth of the network
  - $T=2$ in "classic" NN, $T >> 2$ in deep networks
• \( v_{t,i} \): \( i \)-th neuron in the \( t \)-th layer
• \( \mathbf{v}^{(t)} = (1, v_{t,1}, \ldots, v_{t,d_t})^T \): all neurons of layer \( t \)
• Weights \( w_{rj}^{(t+1)} = w(v_{t,r}, v_{t+1,j}) \): weight of arc from neuron \( r \) of layer \( t \) to neuron \( j \) of layer \( t+1 \)
• \( \mathbf{w}_j^{(t)} = \left( w_{0j}^{(t)}, \ldots, w_{d^{(t-1)}j}^{(t)} \right)^T \): all weights of arcs in input to neuron \( j \) of layer \( t \) (notice: from layer \( t-1 \) to \( t \))
• \( \mathbf{w}^{(t)} \): matrix of weights of all arcs incoming to layer \( t \)

\[
\mathbf{w}^{(t)} = \begin{bmatrix}
    w_{01}^{(t)} & w_{02}^{(t)} & \cdots & w_{0d_t}^{(t)} \\
    w_{11}^{(t)} & w_{12}^{(t)} & \cdots & w_{1d_t}^{(t)} \\
    \vdots & \vdots & \ddots & \vdots \\
    w_{d^{(t-1)}1}^{(t)} & w_{d^{(t-1)}2}^{(t)} & \cdots & w_{d^{(t-1)}d(t)}^{(t)}
\end{bmatrix}
\]
Compute output $o_{t,i}(x)$ of the $i$-th neuron in the $t$-th layer when $x$ is fed to the network:

- Compact notation: use $v_{t,i}$ also to represent the output of the neuron.

The output of a neuron is a non-linear (activation) function applied to the linear combination of the inputs coming from the previous layer:

- $\sigma$: non-linear activation function.
- $a_{t+1,j} = \langle w_j^{(t+1)}, v(t) \rangle$ = output of neuron before the activation function.

$$o_{t+1,j}(x) = \sigma \left( \sum_{r: (v_{t,r}, v_{t+1,j}) \in E} w(v_{t,r}, v_{t+1,j}) o_{t,r}(x) \right) = \sigma \left( a_{t+1,j}(x) \right)$$

In vector notation:

$$v_{t+1,j} = \sigma \left( \langle w_j^{(t+1)}, v(t) \rangle \right) = \sigma(a_{t+1,j})$$
Various activation functions $\sigma(a)$ can be exploited:

1. Sign function
2. Threshold function
3. Sigmoid function
4. Hyperbolic Tangent
5. Rectified Linear Unit
Activation: Sign and Threshold

\[
\sigma(a) = \text{sign}(a)
\]
\[\mathbb{R}^n \to [-1; 1]\]

+ Simple/fast
+ Nice interpretation as the firing rate of a neuron
  - -1 = not firing
  - 1 = firing

- Output is not smooth/continuous
- saturate and kill gradients, thus NN will barely learn

Threshold function: similar behaviour
Activation: Sigmoid

Takes a real-valued number and "squashes" it into range between 0 and 1

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]

\[ \sigma'(a) = \sigma(a)[1 - \sigma(a)] \]

\[ \mathbb{R}^n \rightarrow [0, 1] \]

+ Smooth output

+ Nice interpretation as the firing rate of a neuron
  - 0 = not firing at all
  - 1 = fully firing

- Sigmoid neurons saturate and kill gradients, thus NN will have issues in learning
  - when the neuron’s activation are 0 or 1 (saturate)
    😞 gradient at these regions almost zero
    😞 almost no signal will flow to its weights
    😞 if initial weights are too large then most neurons would saturate
Activation: Tanh

Takes a real-valued number and “squashes” it into a range between -1 and 1.

\[\sigma(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} = \frac{e^{2a} - 1}{e^{2a} + 1}\]

\[\sigma'(a) = 1 - [\tanh(a)^2]\]

\[\mathbb{R}^n \rightarrow [0,1]\]

Tanh is a **scaled and shifted sigmoid**: \(\tanh(x) = 2\text{sigmoid}(2x) - 1\)

+ Like sigmoid, tanh neurons **saturate**
- Unlike sigmoid, output is **zero-centered**
Activation: ReLU

Takes a real-valued number and thresholds it at zero

$$\sigma(a) = \max(0, a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$$

$$\sigma'(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$$

Most Deep Networks use ReLU nowadays

+ Trains much faster
  - accelerates the convergence of SGD
  - due to linear, non-saturating form
+ Less expensive operations
  - compared to \textit{sigmoid}/\textit{tanh} (exponentials etc.)
  - implemented by simply thresholding a matrix at zero
+ More expressive
+ Prevents the gradient vanishing problem
Take an input sample and compute the output of the network.

Start from the input (layer 0)…

….compute the output of layer 1, send to layer 2 and get output….

…. through all the layers up to the output layer.

$$v^{(0)} \rightarrow v^{(1)} \rightarrow v^{(2)} \rightarrow \ldots \rightarrow v^{(T)}$$
Neural Network (NN): \((V,E,\sigma,\nu)\)

- Corresponds to a function \(h_{V,E,\sigma,\nu}: \mathbb{R}^{|V_0-1|} \rightarrow \mathbb{R}^{|V_T|}\)
- The hypothesis class of a network is defined by fixing its architecture:

\[
\mathcal{H}_{V,E,\sigma} = \{ h_{V,E,\sigma,\nu} : \nu \text{ is a mapping from } E \text{ to } \mathbb{R} \}
\]

- \(V,E,\sigma\) defines the architecture of the network
- \(\nu\) contains the parameters that are going to be learned
- Training of the NN: finding the optimal set of weights \(\nu\)
### Proposition

For every \( d \), there exists a graph \((V, E)\) of depth 2 such that \(\mathcal{H}_{V, E, \text{sign}}\) contains all functions from \(\{-1, 1\}^d\) to \(\{-1, 1\}\)

**NN can implement every boolean function!**

Unfortunately the graph \((V, E)\) is very big...

### Proposition

For every \( d \), let \( s(d) \) be the minimal integer such that there exists a graph \((V, E)\) with \(|V| = s(d)\) such that \(\mathcal{H}_{V, E, \text{sign}}\) contains all functions from \(\{-1, 1\}^d\) to \(\{-1, 1\}\). Then \( s(d) \) is an exponential function of \( d \).

**Note:** similar result for \( \sigma = \text{sigmoid} \)
Expressive Power of NN (demonstration)

Consider sign activation $H_{V,E,\text{sign}}$

1. Use this 3 layers NN:
   \[
   \begin{align*}
   \text{INPUT: } |V_0| &= n + 1 \\
   \text{HIDDEN: } |V_1| &= 2^n + 1 \\
   \text{OUTPUT: } |V_2| &= 1
   \end{align*}
   \]

2. Define: $\mathbf{u}_1, \ldots, \mathbf{u}_k \in \{\pm 1\}^n$: all input vectors leading to an output of 1

3. Notice: $\langle \mathbf{x}, \mathbf{u}_i \rangle = \begin{cases} n & \text{if } \mathbf{x} = \mathbf{u}_i \\ \leq n - 2 & \text{if } \mathbf{x} \neq \mathbf{u}_i \end{cases}$ (all bits match)

4. Define $g_i = \text{sign}(\langle \mathbf{x}, \mathbf{u}_i \rangle - n + 1) = \begin{cases} 1 & \text{if } \mathbf{x} = \mathbf{u}_i \\ -1 & \text{otherwise} \end{cases}$

5. Adapt weights $\mathbf{w}$ to get $g_i$ in the hidden layer
   \rightarrow each hidden layer neuron looks if the input is $\mathbf{u}_i$

6. Output layer: $f(\mathbf{x}) = \text{sign}(\sum_{i=1}^{k} g_i(\mathbf{x}) + k - 1)$ (sign is 1 if at least one is true)

Notice: network exponentially large, works but «brute-force» solution probably leading to overfitting
Expressive Power of NN (real valued functions)

**Proposition**

For every fixed $\varepsilon > 0$ and every Lipschitz function $f : [-1, 1]^d \to [-1, 1]$ it is possible to construct a neural network such that for every input $\mathbf{x} \in [-1, 1]^d$ the output of the neural network is in $[f(\mathbf{x}) - \varepsilon, f(\mathbf{x}) + \varepsilon]$.

**Note:** first result proved by Cybenko (1989) for sigmoid activation function, requires only 1 hidden layer!

NNs are **universal approximators**!

But again...

**Proposition**

Fix some $\varepsilon \in (0, 1)$. For every $d$, let $s(d)$ be the minimal integer such that there exists a graph $(V, E)$ with $|V| = s(d)$ such that $\mathcal{H}_{V, E, \sigma}$, with $\sigma =$ sigmoid, can approximate, with precision $\varepsilon$, every 1-Lipschitz function $f : [-1, 1]^d \to [-1, 1]$. Then $s(d)$ is exponential in $d$. 

Not part of the course
Implement Conjunction and Disjunction with NN

- NN can implement boolean AND / OR
- Consider sign activation and $k$ inputs with values $\pm 1$

Conjunction (AND)

$$f(x) = \text{sign} \left( 1 - k + \sum_{i=1}^{k} x_i \right)$$

(positive if all positive, AND)

Disjunction (OR)

$$f(x) = \text{sign} \left( k - 1 + \sum_{i=1}^{k} x_i \right)$$

(positive if at least one positive, OR)
Expressive Power of NN (example)

- Input in $\mathbb{R}^2$, 2-layer NN
- $k$ neurons, sign activation
- Each neuron: an halfspace
- Intersection of halfspaces
- Convex polytopes with $k-1$ faces

- Input in $\mathbb{R}^2$, 3-layer NN
- $k$ neurons, sign activation
- Each neuron: an halfspace
- Intersection and unions of halfspaces
- Union of polytopes
VC dimension of NN

• With *sign* activation
  VC dimension of $\mathcal{H}_{V,E,sign} = O(|E| \log |E|)$  (no demonstration)

• With *Sigmoid* ($\sigma$) activation
  VC dimension of $\mathcal{H}_{V,E,\sigma} = O(|V|^2 |E|^2)$  (no demonstration)

→ Large NNs require a lot of data!

If we have enough data, what about the computation time?
Applying the ERM rule to a NN \((V,E,\sigma,w)\) is computationally difficult, even for relatively small NN...

**Theorem:**

*Hypothesis:* Let \(k \geq 3\). For every \(d\), let \((V,E)\) be a layered graph with \(d\) input nodes, \(k+1\) nodes at the (only) hidden layer (where one of them is the constant neuron), and a single output node.

*Thesis:* It is **NP-hard** to implement the ERM rule with respect to \(H_{V,E,\text{sign}}\)

(no demonstration)

Even approximations of ERM rule are infeasible
Also by changing the activation things do not get better

Need a different strategy....

SGD and backpropagation algorithm!
Target of ERM: given training data \((x_1, y_1), ..., (x_m, y_m)\) find the weights that minimize the training error:

\[
L_s(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h, (x_i, y_i))
\]

- The problem is challenging!
- Idea:
  1. Forward propagate the training data and compute the loss
  2. Consider the loss as a function of the weights and compute the gradient of the loss w.r.t. the weights
  3. Update the weights with SGD

Good Idea! But we need the gradient of the loss w.r.t. the weights
**SGD for Neural Networks**

**parameters:**
Number of iterations $\tau$
Step size sequence $\eta_1, \eta_2, \ldots, \eta_\tau$
Regularization parameter $\lambda > 0$

**Input:**
Network: layered graph $G=(V,E)$
differentiable activation function $\sigma: \mathbb{R} \to \mathbb{R}$

**Algorithm:**
chose $w^{[1]} \in \mathbb{R}^{|E|}$ at random
(from a distribution s.t. $w^{[1]}$ is close enough to 0)
for $s = 1, 2, \ldots, \tau$

- sample $(x, y) \sim D$
- calculate gradient $v_s = \text{backpropagation}(x, y, w, (V, E), \sigma)$
- update $w^{[s+1]} = w^{[s]} - \eta_s (v_s + \lambda w^{[s]})$

**output:**
$\bar{w}$ is the best performing $w^{[s]}$ on a validation set

- adaptive learning rate
- regularization
SGD for NN:

How to Compute the Gradient

Update Rule (baseline version)

\( w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta_s \frac{\partial L_S}{\partial w_{ij}^{(t)[s]}} \)

\[ \frac{\partial L_S}{\partial w} = \frac{\partial}{\partial w} \left( \frac{1}{m} \sum_{i=1}^{m} \ell(h, (x_i, y_i)) \right) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L(h, (x_i, y_i))}{\partial w} \]

- We need the gradient w.r.t. each single weight in the network
- But we can compute the loss only on the output (i.e., after the last layer)
- Recall that each neuron contains only the non-linear activation function

\[ \delta^{(t)} = \frac{\partial L}{\partial a^{(t)}} = \begin{bmatrix} \frac{\partial L}{\partial a_{t,1}} \\ \vdots \\ \frac{\partial L}{\partial a_{t,d_t}} \end{bmatrix} \]

\( \delta^{(t)} \): change in error w.r.t. to the weighted average before the non-linear transformation
BackPropagation (1)

 Decompose the gradient with the chain rule

 Recall $a_{t,j} = \sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k}$

 Each weight $w_{ij}^{(t)}$ impacts only on $a_{t,j}$

 We need $\delta^{(t)} = \frac{\partial L}{\partial a^{(t)}}$ to compute the gradient

 $\sigma'$ depends on the selected activation function

$$\frac{\partial L}{\partial w_{ij}^{(t)}} = \frac{\partial L}{\partial a_{t,j}} \frac{\partial a_{t,j}}{\partial w_{ij}^{(t)}} = \delta_j^{(t)} \frac{\partial}{\partial w_{ij}^{(t)}} \left( \sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k} \right) = \delta_j^{(t)} v_{t-1,i}$$

$$\delta_j^{(t)} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \sigma'(a_{t,j})$$

remains only $k=i$ term
BackPropagation (2)

- Understand how the loss changes w.r.t. $v_{t,j}$
- Change in layer $t$ affects only neurons in layer $t+1$ (and then each following layer up to the loss at the end)
- Each neuron can affect all the neurons in next layer
- Need to sum contributions to all the neurons in layer $t+1$
- To compute $\delta^{(t)}$ we need $\delta^{(t+1)}$ (solution of the next layer)

\[
\frac{\partial L}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} \frac{\partial L}{\partial a_{t+1,k}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}
\]

\[
\delta_j^{(t)} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \sigma'(a_{t,j}) \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}
\]
The solution for each layer need the solution of the following one
1. Start from the last layer ($\delta^{(L)}$ can be computed from the loss on the output)
2. Backpropagate the gradients through all the layers up to the first
Input: data point \((x_i, y_i)\), NN (with weights \(w_{ij}^{(t)}\), for \(1 \leq t \leq T\))

Output: \(\delta^{(t)}\) for \(t = 1, \ldots, T\)
compute \(a^{(t)}\) and \(v^{(t)}\) for \(t = 1, \ldots, T\);
\[
\delta^{(L)} \leftarrow \frac{\partial L}{\partial a^{(L)}};
\]
for \(t = T - 1\) downto 1 do
\[
\delta_j^{(\ell)} \leftarrow \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)} \text{ for all } j = 1, \ldots, d^{(t)};
\]
return \(\delta^{(1)}, \ldots, \delta^{(T)}\);

\[\nu^{(0)} \leftarrow \nu^{(1)} \leftarrow \nu^{(2)} \leftarrow \ldots \leftarrow \nu^{(T)}\]
**NN Training: complete algorithm**

*BackPropagation algorithm with SGD*

*Input:* training data \((x_1, y_1), \ldots, (x_m, y_m)\)

*Output:* NN weights \(w_{ij}^{(t)}\)

Initialize \(w_{ij}^{(t)}\) \(\forall i,j,t;\)

**for** \(s \leftarrow 0,1,2, \ldots \) **do**

- pick \((x_k, y_k)\) at random from training data;  // SGD: 1 sample at random
- compute \(v_{t,j}\) \(\forall j,t;\)  // forward propagation
- compute \(\delta_j^{(t)}\) \(\forall j,t;\)  // backward propagation

\[ w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta v_{t-1,i} \delta_j^{(t)} \] \(\forall i,j,t;\)  // update weights

**if converged then return** \(w_{ij}^{(t)[s]}\) \(\forall i,j,t;\)
NN Training Details: Pre-processing and Initialization

Pre-processing:
- Typically all inputs are normalized and centred around 0
- Both local or global normalization strategies

Initialization of the weights
- All to 0 does not work
- Random values around 0 (regime where model is roughly linear)
- Uniform or normal (Gaussian) distribution can be used
- Sometimes multiple initializations and trainings, then select best result (smallest training error)
- In deep NN "Glorot" initialization: normal distribution with variance inversely proportional to the sum of the number of incoming and outcoming connections of the neuron
NN Training details: When to Stop

When to stop?

- Small training error
- Small marginal improvement in error at each step
- Upper bound on number of iterations

Loss function usually has multiple local minima

- With highly dimensional spaces the risk is smaller than in low dimensional ones, but no guarantee
- Run stochastic gradient descent (SGD) from different (random) initial weights
Regularization

- Minimize weighted sum of the loss with the sum of all the weights
- Avoid too large weights and make optimization more stable
- Regularization parameter $\lambda$
- **L1** or **L2** regularization can be used

$$L_s(h) + \frac{\lambda}{m} \sum_{i,j,t} |w_{ij}^{(t)}|$$

$$L_s(h) + \frac{\lambda}{m} \sum_{i,j,t} (w_{ij}^{(t)})^2$$