Introduction to Neural Networks

Danilo Pau
Advanced System Technology
Agrate Brianza
Learning XOR

• Not linearly separable

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \oplus x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Original $x$ space

Source https://vision.unipv.it/AI/AIRG.html
\( y = X \ast \vartheta \)

\[
\begin{bmatrix}
0 \\
1 \\
1 \\
0
\end{bmatrix}
\quad
\begin{bmatrix}
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{bmatrix}
\quad
\begin{bmatrix}
w_1 \\
w_2 \\
c
\end{bmatrix}
\]
Function Approximation: linear combination

• Linear Approximator as 1\textsuperscript{st} attempt

\[ \hat{y} = w \cdot x + c, \quad w \in \mathbb{R}^d, c \in \mathbb{R} \]

For XOR:
\[ \vartheta = (X^T X)^{-1} X^T y \]

\[ X^T X = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 4 \end{bmatrix} \quad \text{squared} \]
\[ (X^T X)^{-1} = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 1 & 0.5 \\ 0.5 & 0.5 & 0.75 \end{bmatrix} \quad \text{squared} \]

\[ (X^T X)^{-1} X^T y = \begin{bmatrix} 0 \\ 0 \\ 0.5 \end{bmatrix} \]

Hence approximator becomes \[ \hat{y} = 0.5 \]

Source https://vision.unipv.it/AI/AIRG.html
Function Approximation: linear combination

\[ \hat{y} = 0.5 \]

Source https://vision.unipv.it/AI/AIRG.html
Solving XOR

- (shallow) feed-forward neural network

$$\tilde{y} = w \cdot \text{ReLU}(W x + c) + c$$
Universal approximation theorem
(Cybenko, 1989, Hornik, 1991)

For any target function
\[ y = f^*(x), \quad x \in \mathbb{R}^d \] (which is continuous and Borel measurable)

and any \( \varepsilon > 0 \) there exists parameters
\[ h \in \mathbb{Z}^+, \quad W \in \mathbb{R}^{h \times d}, \quad w, c \in \mathbb{R}^h, \quad c \in \mathbb{R} \]

\( h \) is the dimension of the hidden layer; it is a parameter in the theorem

such that the (shallow) feed-forward neural network
\[ \tilde{y} = w \cdot g(Wx + c) + c \]

approximates the target function by less than \( \varepsilon \)
\[ |f^*(x) - w \cdot g(Wx + c) + c| < \varepsilon \]

Source https://vision.unipv.it/AI/AIRG.html (on a compact subset of \( \mathbb{R}^d \))
Artificial Neural Network

State of the art

Core of all deep learning applications

Adaptable
What are Neural Networks?

- Also referred to as Artificial Neural Networks.
- Inspired by human neural system.
- Human neuron has three main components
  - Dendrites
    - Takes inputs from other neurons in terms of electrical pulses.
  - Cell body
    - Makes the inferences and decides the actions to take.
  - Axon terminals
    - Sends the outputs to other neurons in terms of electrical pulses.
- Synapse
  - Interface between Axons and Dendrites
Michael Jordan: There are no spikes in deep-learning systems. There are no dendrites. And they have bidirectional signals that the brain doesn’t have.

We don’t know how neurons learn. Is it actually just a small change in the synaptic weight that’s responsible for learning? That’s what these artificial neural networks are doing. In the brain, we have precious little idea how learning is actually taking place.

Spectrum: I read all the time about engineers describing their new chip designs in what seems to me to be an incredible abuse of language. They talk about the “neurons” or the “synapses” on their chips. But that can’t possibly be the case; a neuron is a living, breathing cell of unbelievable complexity! Aren’t engineers appropriating the language of biology to describe structures that have nothing remotely close to the complexity of biological systems?

Michael Jordan: Well, I want to be a little careful here. I think it’s important to distinguish two areas where the word neural is currently being used.

One of them is in deep learning. And there, each “neuron” is really a cartoon. It’s a linear-weighted sum that’s passed through a nonlinearity. Anyone in electrical engineering would recognize those kinds of nonlinear systems. Calling that a neuron is clearly, at best, a shorthand. It’s really a cartoon. There is a procedure called logistic regression in statistics that dates from the 1950s, which had nothing to do with neurons but which is exactly the same little piece of architecture.
Artificial Neuron

The heart of a neural network

\[ \sum_{i=1}^{m} (w_i x_i) + \text{bias} \]

Activation function:
\[ f(x) = \begin{cases} 1 & \text{if } \sum wx + b \geq 0 \\ 0 & \text{if } \sum wx + b < 0 \end{cases} \]
Artificial Neuron

The heart of a neural network

Inputs: \( x_1, x_2, \ldots, x_m \)

Weights: \( w_1, w_2, \ldots, w_m \)

Cell body:

\[
\sum_{i=1}^{m} (w_i x_i) + bias
\]

Activation function:

\[
f(x) = \begin{cases} 
1 & \text{if } \sum wx + b \geq 0 \\
0 & \text{if } \sum wx + b < 0 
\end{cases}
\]

Output: \( \hat{y} \)
Artificial Neuron

The heart of a neural network

Inputs → Weights → Summation and Bias → Activation → Output

Cell body

\[ \sum_{i=1}^{m} (w_i x_i) + bias \]

Activation function

\[ f(x) = \begin{cases} 1 & \text{if } \sum w x + b \geq 0 \\ 0 & \text{if } \sum w x + b < 0 \end{cases} \]
Artificial Neuron

The heart of a neural network

\[
\sum_{i=1}^{m} (w_i x_i) + bias
\]

Activation function:

\[
f(x) = \begin{cases} 
1 & \text{if } \sum wx + b \geq 0 \\
0 & \text{if } \sum wx + b < 0 
\end{cases}
\]

Inputs, Weights, Summation and Bias, Activation, Output
Artificial Neuron

The heart of a neural network

\[
\sum_{i=1}^{m} (w_i x_i) + \text{bias}
\]

Activation function

\[
f(x) = \begin{cases} 
1 & \text{if } \sum w_i x_i + b \geq 0 \\
0 & \text{if } \sum w_i x_i + b < 0 
\end{cases}
\]
Artificial Neuron

The heart of a neural network

$$\sum_{i=1}^{m} (w_i x_i) + \text{bias}$$

Activation function:

$$f(x) = \begin{cases} 
1 & \text{if } \sum w_x + b \geq 0 \\
0 & \text{if } \sum w_x + b < 0
\end{cases}$$
Artificial Neuron

The heart of a neural network

\[ \sum_{i=1}^{m} (w_i x_i) + bias \]

Activation function

\[ f(x) = \begin{cases} 
1 & \text{if } \sum wx + b \geq 0 \\
0 & \text{if } \sum wx + b < 0 
\end{cases} \]

Approximation of the expected output
Artificial Neuron

Perceptron: The heart of a neural network

\[ y \hat{=} \sum_{i=1}^{m} w_i x_i \]
Convolutions

- **Kernel Size**: the field of view of the convolution
- **Stride**: the step size of the kernel when traversing the image.
- **Padding**: defines how the border of a sample is handled.
- **Input & Output Channels**: A convolutional layer takes a certain number of input channels and calculates a specific number of output channels
2D Convolutions

Input

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

2D kernel

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

2D convolution process:
- Multiply the 2D kernel with the corresponding input values.
- Add the products together, including the bias term.
- The result is the convolved feature.

- Output values:
  - Input values: 1, 4, 0, 3
  - Kernel values: 2, 1, 1, 2
  - Bias = 2

- Convolved feature value: 11
**Convolution operation**

- A **convolution filter** is a square (or cubic) matrix
  - It is first centered on a pixel of the input image
  - It produces a scalar value: the dot product between the filter and the image region around the pixel
  - By mapping the same procedure on all pixels of the input image,
  - a new image is produced (i.e. a *feature map*)
3D Convolutions

- Convolution operations (on images)
  - A convolution filter is a square (or cubic) matrix
  - In symbols: $Y_i := W_i \ast X$
  - where:

Input image (e.g. RGB)

Convolution filters

Feature Maps

[Image from http://cs231n.github.io/convolutional-networks/]

$Y_0$

$Y_1$
3D Convolutions

Input image → Convolutional Filters → Feature Maps

- Input Volume (+pad 1) (7x7x3)
- Filter W0 (3x3x3)
  - Filter W1 (3x3x3)
  - Output Volume (3x3x2)

```
Input Volume (+pad 1) (7x7x3)
x[:,:,0] =
0 0 0 0 0 0 0
0 0 1 0 2 0 0
0 1 0 2 0 1 0
0 1 0 2 2 0 0
0 2 0 0 2 0 0
0 2 1 2 2 0 0
0 2 1 2 0 1 0

Filter W0 (3x3x3)
w0[:,:,0] =
-1 0 1
0 0 1
1 -1 1
-1 0 -1
0 -1 1
0 -1 1
1 -1 1

Filter W1 (3x3x3)
w1[:,:,0] =
0 1 -1
0 1 0
0 -1 1
0 1 0
0 -1 1

Output Volume (3x3x2)
\sigma[:,:,0] =
2 3 3
3 7 3
8 10 -3
8 -8 -3
-3 1 0
-3 -8 -5
```

Bias b0 (1x1x1)
b0[:,:,0] =
1

Bias b1 (1x1x1)
b1[:,:,0] =
0
Transposed Convolutions

• Also known as Fractionally Strided Convolutions (e.g. 1/2, 1/4 etc)

• Perform some fancy padding on the input

• Not able to numerically reverse convolution followed by down sampling
Sparse connections due to small convolution kernel

Dense connections
Sparse connections due to small convolution kernel

Dense connections
Dilated Convolutions

Dilation = 0

\[ s_1, s_2, s_3, s_4, s_5 \]

\[ x_1, x_2, x_3, x_4, x_5 \]

Dilation = 1

\[ s_1, s_2, s_3, s_4, s_5 \]

\[ x_{-1}, x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 \]
Convolutions with Stride

\[ s_1 \overset{2}{\rightarrow} s_3 \overset{1}{\rightarrow} s_2 \overset{3}{\rightarrow} s_4 \overset{1}{\rightarrow} s_5 \]

0 = zero padding

\[ s_1 \overset{0}{\rightarrow} x_1 \overset{1}{\rightarrow} x_2 \overset{1}{\rightarrow} x_3 \overset{1}{\rightarrow} x_4 \overset{0}{\rightarrow} x_5 \overset{0}{\rightarrow} \]

Stride 2

\[ s_1 \overset{0}{\rightarrow} s_3 \overset{0}{\rightarrow} s_5 \overset{0}{\rightarrow} \]

Stride 1
Receptive field or field of view of the filter (filter size, e.g. 5x5x3)

Each filter has $(5 \times 5 \times 3 + 1) = 76 \text{ weights}$ and 75 local (in space= connections to a neuron)

$W = 32 \text{ #input size}$
$F = 5 \text{ #filter size}$
$P = 2 \text{ # padding}$
$S = 1 \text{ # stride}$

$\text{width} = \text{height} = \frac{W-F+2P}{S} + 1 = 32$
Complexity of 3D Convolutional Layer

Number of filters $K$, Filter spatial extension $F_x, F_y, F_z$, The stride $S$, The amount of zero padding $P$.

$$W_2 = \frac{W_1 - F_x + 2P}{S} + 1$$

$$H_2 = \frac{H_1 - F_y + 2P}{S} + 1$$

$$D_2 = K$$

$$F_x * F_y * F_z * K * H_2 * W_2$$

$$((F_x * F_y * F_z - 1) + 1) * K * H_2 * W_2$$

Output feature map

Input feature map
Netscope CNN Analyzer

- [https://dgschwend.github.io/netscope/#/preset/vgg-16](https://dgschwend.github.io/netscope/#/preset/vgg-16)

- VGG ILSVRC 16 layers
Parameter sharing

- To limit number of parameters in Convolutional Layers.
- Using the example before
  - a volume of size [32x32x5] has 5 depth slices, each of size [32x32]
  - there are $32 \times 32 \times 5$ (slide 20) = 5,120 neurons in the Conv Layer
  - each neuron has a own $5 \times 5 \times 3$ (filter) = 75 weights and 1 bias.
  - This adds up to $5,120 \times 76 = 389,120$ parameters for Conv layer itself.
- **How to reduce it?**
  - **Spatial correlation assumption**: if one feature is useful to compute at some spatial position $(x,y)$, then it should also be useful to compute at a different position $(x_2, y_2)$.
  - **Solution**: To constrain the neurons in each depth slice to use the same weights and bias.
  - Only 5 unique set of weights, one for each depth slice, for a total of 5 slices*(5*5*3 weights per slice) = 375 unique weights, (+5 biases).
Summary of Convolutional Layer

With parameter sharing, the layer requires \( F_x \times F_y \times F_z \times D_1 \) parameters per filter, for a total of \( F_x \times F_y \times F_z \times D_1 \times K + K \) biases.

Number of filters \( K \),
Filter spatial extension \( F_x, F_y, F_z \),
The stride \( S \),
The amount of zero padding \( P \).

\[
W_1 \times H_1 \times D_1
\]

\[
W_2 = \frac{W_1 - F_x + 2P}{S} + 1
\]

\[
H_2 = \frac{H_1 - F_y + 2P}{S} + 1
\]

\[
D_2 = K
\]

Usually \( F_z = D_1 \) and no padding is applied on \( z \) direction.
Depth wise separable convolution

- Consider $d_1 = 16$ channels, $w_f = h_f = 3$ each kernel (2D) $\rightarrow$ 16 feature maps.
- Traverse these 16 feature maps with $d_{out} = 32$, $w_2 = h_2 = 1$ convolutions each.
- This results in $656 = [16 \times (3 \times 3) + 16 \times (32 \times 1 \times 1)]$ parameters opposed to the $4608 = (16 \times 32 \times 3 \times 3)$ parameters from non depth separable filtering.
Pooling Features

Convolved Feature Map

Pooled Feature Map

Projection
Pooling features

Max Pooling, 3:1

Average Pooling, 3:1
Summary of Pooling Layer

Input Feature Map

Output Feature Map

Spatial extension $F$

The stride $S$

$W_2 = \frac{W_1 - F}{S} + 1$

$H_2 = \frac{H_1 - F}{S} + 1$

$D_2 = D_1$

$No$ parameters
### Activation Functions

\[ z = f(\sum_{i=1}^{m} w_i x_i + \text{bias}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Plot</th>
<th>Function</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Unit Step                   | ![Plot](image) | \( \phi(z) = \begin{cases} 
0, & z < 0, \\
0.5, & z = 0, \\
1, & z > 0, 
\end{cases} \) | Perceptron variant                     |
| Sign (Signum)               | ![Plot](image) | \( \phi(z) = \begin{cases} 
-1, & z < 0, \\
0, & z = 0, \\
1, & z > 0, 
\end{cases} \) | Perceptron variant                     |
| Linear                      | ![Plot](image) | \( \phi(z) = z \)          | Adaline, linear regression             |
| Piece-wise linear           | ![Plot](image) | \( \phi(z) = \begin{cases} 
1, & z \geq \frac{1}{2}, \\
\frac{1}{2} < z < \frac{1}{2}, \\
1, & z > 0, 
\end{cases} \) | Support vector machine                 |
| Logistic (sigmoid)          | ![Plot](image) | \( \phi(z) = \frac{1}{1 + e^{-z}} \) | Logistic regression, Multi layer-neural networks |
| Hyperbolic Tangent          | ![Plot](image) | \( \phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \) | Multi layer neural networks            |
| Rectified Linear Unit       | ![Plot](image) | \( \phi(z) = \begin{cases} 
z, & z > 0 \\
0, & z \leq 0 
\end{cases} \) | Regression, approximation, multi layer neural network |
Most widely used activations

- **Unit step**
  - Threshold

- **Sigmoid Function**
  - Like a step function but smoother
  - Best to predict probabilities

- **Tan hyperbolic**
  - Stretched out version of the sigmoid function

- **ReLU**
  - Computationally efficient

- Function choice depends on the characteristics of the data.
- For example Sigmoid Function works good for classification purposes, resulting in Faster training and convergence.
- ReLU is good for approximation. As it is simple so always start from this if you don’t know the data characteristics. Helps against gradient vanishing.
- We can also define custom activations.
Most widely used activations

• The Softmax function, or normalized exponential function

• A generalization of the logistic function

• Squeeze the K-dimensional input vector of real values into values in the range $[0, 1]$.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \text{ for } j = 1, 2, 3, \ldots, K.$$
Local Response Normalization

- Compensating the tendency of ReLu to output large values

\[
Y_{ijl} = \frac{X_{ijl}}{(\alpha + \beta \sum_{k \in Nbr(l)} X_{ijl}^2)\gamma}
\]

\(\alpha, \beta, \gamma\) are hyper-parameters
Layers of a Neural Network

• Neural network has three types of layers
  • Input layer
    • Can be from other neurons or feature inputs
      • Age, height, weight, pixels in the images etc.
  • Hidden layers (one or more)
    • Real power lies here
    • Adding more neurons to the network
  • Output layer
    • Gives the output we want to predict
      • Probability of rain
      • Object class
      • Disease is fatal or not…
Neural Networks

• Notations

\[ x_n, n = \{1, 2, 3, \ldots \} \]
\[ h_p^m, m = \{1, 2, 3, \ldots \}, p = \{1, 2, 3, \ldots \} \]
Example of a Neural Network

• To predict if a person has to be hospitalized given
  • Age
  • Gender
  • Distance from hospital
  • Income
  • Number of General Physician (GP) visits

• Let us suppose to train neural network, which means to compute all the weights so that predictions are accurate.

• Consider to have
  • Age = 65
  • Gender = Female
  • Distance, income and GP visit high
Example of a Neural Network

- To predict if a person has to be hospitalized given
  - Age
  - Gender
  - Distance from hospital
  - Income
  - Number of General Physician (GP) visits

- Let us suppose to train neural network, which means to compute all the weights so that predictions are accurate.

- Consider to have
  - Age = 65
  - Gender = Female
  - Distance, income and GP visit high
Training neural networks

• In supervised learning an assumption is to have a relatively large labeled dataset.

• Feed all the samples as inputs to get an output. Called forward propagation or inference run outputs.

• At start the weights can be randomized or predefined depending on the applications scenario.

• The result $\hat{y}$ is compared with ground truth output $y$.

• The task is to make the output value $\hat{y}$ to be as close to $y$ as possible reducing the error expressed as Loss functions $L(\hat{y}, y)$.

$$L(\hat{y}, y) = (\mathbf{w} \cdot \text{ReLU}(\mathbf{W} \mathbf{x} + \mathbf{c}) + \mathbf{c} - y)^2 = \text{error}$$
Forward Propagation

- Let's do it *(in a graphical way)*

\[ L(\tilde{y}, y) = (\mathbf{w} \cdot \max(0, \mathbf{W} \mathbf{x} + \mathbf{c}) + c - y)^2 \]

*Element-wise loss, with ReLU as non-linearity*
Training neural networks

- Go back and adjust the weights slowly. Aim is $\text{Error}_T < \text{Error}_{T-1}$

- Repeat this process until the error we get is very small.

$$\lim_{\varepsilon \to 0} \text{Error}_T < \varepsilon$$
Backward Propagation

• Let's do it *(in a graphical way)*

\[
L(\tilde{y}, y) = (w \cdot \max(0, Wx + c) + c - y)^2
\]

*Element-wise loss, with ReLU as non-linearity*

\[
\frac{\partial}{\partial W}(w \cdot \text{ReLU}(Wx + c) + c - y)^2
\]
\[ error = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \]
Backpropagation

- Brute force
  - Try all the possible combination of weights.
  - Plot the cost function.
  - Use the weights which result in smallest error.
  - Sounds simple but will take too much time !!!

Cost function

\[ error = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \]
Backpropagation

• Enters the gradient descent

Gradient Descent

Learning rate compromise

• Big learning rate
  • Fast
  • May never converge.

• Small learning rate.
  • Lots of small steps
  • Will converge for sure
• Enters the gradient descent

Gradient Descent

Use adaptive learning rate!
Gradient descent in action

- Example: Finding best linear fit to a set of points.
https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html