Applied Machine Learning in Biomedicine

Enrico Grisan
enrico.grisan@dei.unipd.it
Neuron basics

- **Impulses carried toward cell body**
- **Dendrites**
- **Nucleus**
- **Cell body**
- **Impulses carried away from cell body**
- **Branches of axon**
- **Axon**
- **Axon terminals**
Neuron: real and simulated
A bit of history

Widrow and Hoff, ~1960: Adaline
From biology to models
Biological models?

Careful with brain analogies:

Many different types of neurons

Dendrites can perform complex non-linear computations

Synapses are not single weights but complex dynamical dynamical system

Rate code may not be adequate

[Dendritic Computation. London and Hausser]
Single neuron classifier
Neuron and logistic classifier

\[ o(x) = \sigma(w^T x) = \frac{1}{1 + e^{-(w_0 + \sum_i w_i x_i)}} \]
How a change in the output (loss) affects the weights?

$$L(w, y) \approx (y - \hat{y})^2$$
Linking output to input

How a change in the output (loss) affects the weights?

Backward flow
Activation function

\[
\frac{1}{1 + e^{-y(w^T x)}} = \sigma(w^T x) = \sigma(z)
\]

**Ups**
1) Easy analytical derivatives
2) Squashes numbers to range [0,1]
3) Biological interpretation as saturating «firing rate» of a neuron

**Downs**
1) Saturated neurons kill the gradients
2) Sigmoid output are not zero-centered
Sigmoid backpropagation

Assume the input of a neuron is always positive. What about the gradient on \( w \)?

\[
f \left( \sum_i w_i x_i + b \right)
\]

\[
w^{t+1} = w^t + \nabla_w f
\]

Gradient is all positive or all negative!
Improving activation function

Ups
1) Still analytical derivatives
2) Squashes numbers to range $[-1,1]$
3) Zero-centered!

Downs
1) Saturated neurons kill the gradients

\[ \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \]

\[ \frac{d \tanh(x)}{dx} = 1 - \tanh^2(x) \]
activation function 2

rectifying linear unit: ReLU

ups
1) Does not saturate
2) Computationally efficient
3) Converges faster in practice

downs
1) What happens for x<0?

\[ f(x) = \max(0, x) \]

\[ \frac{df}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases} \]
ReLU neuron killing

DATA CLOUD

active ReLU

dead ReLU will never activate => never update
Activation function 3

Leaky ReLU

**Ups**
1) Does not saturate
2) Computationally efficient
3) Converges faster in practice
4) Keep neurons alive!

\[ f(x) = H(-x)\alpha x + H(x)x \]

\[
\frac{df}{dx} = \begin{cases} 
1 & x > 0 \\
\alpha & x < 0 
\end{cases}
\]
Maxout

\[ f(x) = \max(w_1^T x, w_2^T x) \]

**Ups**
1) Does not saturate
2) Computationally efficient
3) Linear regime
4) Keeps neurons alive!
5) Generalizes ReLU and leaky ReLU

**Downs**
1) Is not a dot product
2) Doubles the parameters
Neural Networks: architecture

“Fully-connected” layers

input layer
hidden layer
output layer

input layer
hidden layer 1
hidden layer 2
output layer
Neural Networks: architecture

2-layers Neural Network
1-hidden layer Neural Network

3-layers Neural Network
2-hidden layers Neural Network
Neural Networks: architecture

Number of neurons?
Number of weights?
Number of parameters?
Neural Networks: architecture

Number of neurons: 4+2=6
Number of weights: 4\times3+2\times4=20
Number of parameters: 20+6
Neural Networks: architecture

Number of neurons: \(4+2=6\)
Number of weights: \(4 \times 3 + 2 \times 4 = 20\)
Number of parameters: \(20 + 6\)

Number of neurons: \(4+4+1=9\)
Number of weights: \(4 \times 3 \times 4 + 4 \times 4 + 1 \times 4 = 32\)
Number of parameters: \(32 + 9\)
Neural Networks: architecture

Modern CNNs: ~10 million artificial neurons
Human Visual Cortex: ~5 billion neurons
ANN representation

\[ \begin{bmatrix} 1 & x_{11} & \cdots & x_{13} \\ 1 & x_{21} & \cdots & x_{23} \\ 1 & x_{31} & \cdots & x_{33} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{N3} \end{bmatrix} = \begin{bmatrix} x_1^T \\ x_2^T \\ x_3^T \\ \vdots \\ x_N^T \end{bmatrix} \]

\[ w_{1,i} = \begin{bmatrix} w_{0,i} \\ w_{1,i} \\ w_{2,i} \\ w_{3,i} \end{bmatrix} \]

\[ w_{2,i} = \begin{bmatrix} w_{0,i} \\ w_{1,i} \\ w_{2,i} \\ w_{3,i} \end{bmatrix} \]
ANN representation

\[ X = \begin{bmatrix}
1 & x_{11} & \cdots & x_{13} \\
1 & x_{21} & \cdots & x_{23} \\
1 & x_{31} & \cdots & x_{33} \\
\vdots & \vdots & \ddots & \vdots \\
1 & x_{N1} & \cdots & x_{N3}
\end{bmatrix} = \begin{bmatrix}
x_1^T \\
x_2^T \\
x_3^T \\
\vdots \\
x_N^T
\end{bmatrix} \]

\[ W_1 = \begin{bmatrix}
w_{0,1} & w_{0,2} & w_{0,3} \\
w_{1,1} & w_{1,2} & w_{1,3} \\
w_{2,1} & w_{2,2} & w_{2,3} \\
w_{3,1} & w_{3,2} & w_{3,3}
\end{bmatrix} = \begin{bmatrix}
w_{1,1} & w_{1,2} & w_{1,3}
\end{bmatrix} \]

\[ W_2 = \begin{bmatrix}
w_{1,2} & w_{2,2}
\end{bmatrix} \]

\[ Z_1(X) = f_1 (W_1X) \]

\[ Y(X) = f_2 (W_2Z_1) = f_2 (W_2f_1 (W_1X)) \]
ANN becoming popular

To be more specific, then, let

$$E_p = \frac{1}{2} \sum_j (t_{pj} - o_{pj})^2$$

be our measure of the error on input/output pattern $p$ and let $E = \sum E_p$ be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in $E$ when the units are linear. We will proceed by simply showing that

$$-\frac{\delta E_p}{\delta w_{ji}} = \delta_{pj}i_{pi},$$

which is proportional to $\Delta_j w_{ji}$ as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.

$$\frac{\delta E_j}{\delta w_{ji}} = \frac{\delta E_j}{\delta o_{pj}} \frac{\delta o_{pj}}{\delta w_{ji}}.$$  (3)

The first part tells how the error changes with the output of the $j$th unit and the second part tells how much changing $w_{ji}$ changes that output. Now the derivatives are easy to compute. First, from Equation 2

$$\frac{\delta E_j}{\delta o_{pj}} = - (t_{pj} - o_{pj}) = - \delta_{pj}.$$  (4)

Not surprisingly, the contribution of unit $u_j$ to the error is simply proportional to $\delta_{pj}$. Moreover, since we have linear units,

$$o_{pj} = \sum_i w_{ji} i_{pi}.$$  (5)

from which we conclude that

$$\frac{\delta o_{pj}}{\delta w_{ji}} = i_{pi}$$

Thus, substituting back into Equation 3, we see that

$$-\frac{\delta E_p}{\delta w_{ji}} = \delta_{pj} i_{pi}.$$  (6)
Define a loss function $L(w^T; y)$

Each neuron computes: $a_j = \sum_i w_{ji} z_i$

And pass to the following layer: $z_j = f(a_j)$
ANN training: backward flow

Need to compute:
\[ \frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i \]

Considering that for the output neurons:
\[ \delta_k = \hat{y}_k - y_k \]
We get:
\[ \delta_j = f'(a_j) \sum_k w_{kj} \delta_k \]
ANN training: backpropagation

Updating all weights:

\[ w_{ji}^{t+1} = w_{ji}^t + \eta \delta_j \]
ANN training ex: forward

\[ f(a) = \tanh(a) \]

\[ L_n = \frac{1}{2} \sum_{k=1}^{K} (\hat{y}_k - y_k)^2 \]

\[ a_j = \sum_{i=0}^{D} w_{ij}^{(1)} x_i \]

\[ z_j = \tanh(a_j) \]

\[ \hat{y}_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j \]
ANN training ex: backward

\[ \delta_k = \hat{y}_k - y_k \]

\[ \delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k \]

\[ \frac{\partial L_n}{\partial w_{ji}^{(1)}} = \delta_j x_i \]

\[ \frac{\partial L_n}{\partial w_{kj}^{(2)}} = \delta_k z_j \]
What can ANN represent?
What can ANN classify?
Regularization

\[ \lambda = 0.001 \quad \lambda = 0.01 \quad \lambda = 0.1 \]