Neural Networks

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Neural networks

Neural Networks (NN) provide a method to approximate a non–linear function ϕ

A Neural Network can be interpreted as a non–linear parametric function $\phi(\boldsymbol{x}, \boldsymbol{\Pi})$

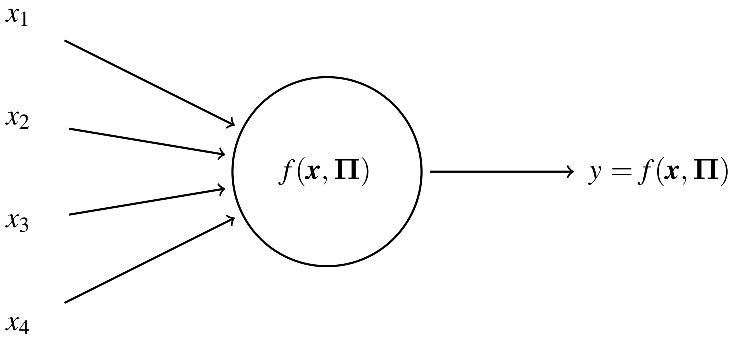
The parameters of the non–linear transformation are learned from the data

The function is represented by means of a directed graph

Each node is associated to a function that operates on the input nodes and provides the node output

Neural networks

The basic unit of a neural network is a computation node


x_1
x_2
x_3
x_4
$f(\boldsymbol{x}, \boldsymbol{\Pi})$
$y = f(\boldsymbol{x}, \boldsymbol{\Pi})$

The node computes a parametric function of its inputs $f(\boldsymbol{x}, \boldsymbol{\Pi})$

Neural networks

Typically, the non-linear function is expressed in terms of a parametric affine combination of the node inputs and a scalar, non-linear, non-parametric transformation

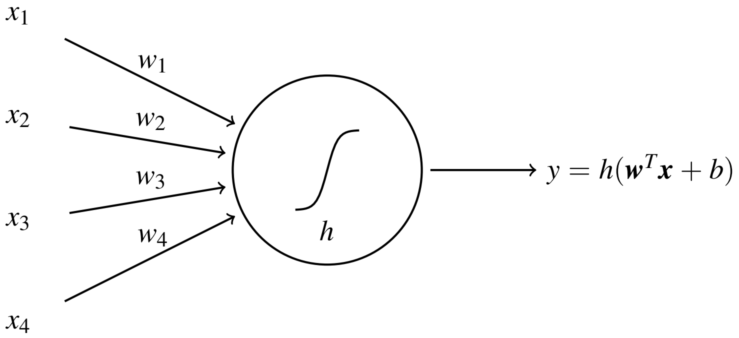
$$f(\boldsymbol{x}, \boldsymbol{\Pi}) = h(\boldsymbol{w}^T \boldsymbol{x} + b)$$

$\boldsymbol{\Pi} = (\boldsymbol{w}, b)$ are the function parameters: $\boldsymbol{w}$ is a vector containing the weights of the affine combination and b is a bias (scalar)

In this case, we can graphically associate the weights parameters with the node input arcs

Neural networks

Let $\boldsymbol{w} = \left[w_1 \dots w_d\right]$, where d is the dimensionality of the node input. The node representation becomes



The weights can be interpreted as the “strength” of the connection — $w_i = 0$ implies that the corresponding input x_i does not influence the final result

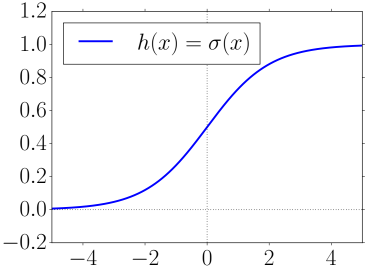
Neural networks

Several possible functions have been proposed for the non–linearity function h

- Sigmoid function $h(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent $h(x) = \tanh(x)$
- Rectified linear $h(x) = max(0, x)$

Neural networks

Sigmoid function $h(x) = \frac{1}{1+e^{-x}}$

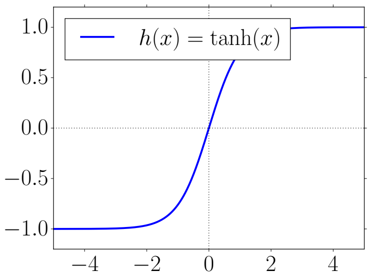


A sigmoid function-activated node can be interpreted as a binary logistic regression model for the input data $\boldsymbol{x}$

$$h(\boldsymbol{w}^T\boldsymbol{x} + b) = \frac{1}{1 + e^{-\boldsymbol{w}^T\boldsymbol{x}+b}}$$

Neural networks

Hyperbolic tangent $h(x) = \tanh(x)$



It can be interpreted as a symmetric version of the sigmoid

$$\sigma(x) = \frac{\tanh\left(\frac{x}{2}\right) + 1}{2}, \quad \tanh(x) = 2\sigma(2x) - 1$$

Neural networks

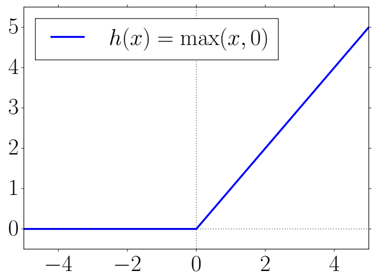
For small inputs both sigmoid and hyperbolic tangent behave almost linearly

For large inputs the functions saturate (non-linear behavior)

Computationally expensive, may pose issues during training

Neural networks

Rectified linear $h(x) = max(0, x)$



More computationally efficient, less prone to training issues due to vanishing gradients, but not differentiable everywhere

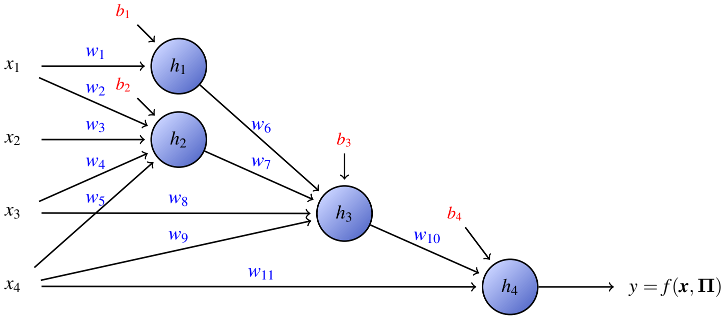
Widely adopted in practice

Neural networks

A neural network is a directed graph of nodes

We will consider only graphs without loops (acyclic)

The graph defines a sequence of operations



$$f(\boldsymbol{x}, \boldsymbol{\Pi}) = h_4(w_{10}h_3(w_6h_1(w_1x_1+b_1)+w_7h_2(w_2x_1+w_3x_2+w_4x_3+w_5x_4+b_2) \\ + w_8x_3 + w_9x_4 + b_3) + w_{11}x_4 + b_4)$$

Neural networks

An acyclic graph is also known as feed forward network

Information “flows” from the input to the output nodes

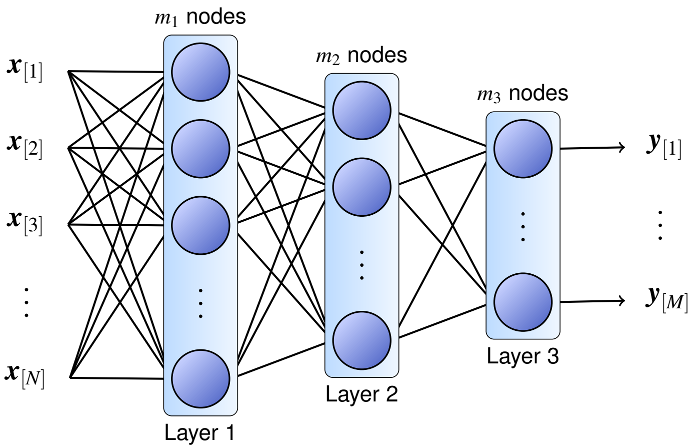
It’s useful to organize nodes in layers

Layer: group of nodes that share the same inputs

Typically, a layer receives as input the output of a previous layer

Neural networks

Multi-layer network: units are organized in layers



Connections are defined between layers (no loops)

Neural networks

The layers are identified with a progressive index

We can represent the input $\boldsymbol{n}_j$ of the layer as the vector of the inputs of the nodes associated to that layer

We can represent the output $\boldsymbol{x}_j$ of the layer as the vector of outputs of its nodes

Each node j in layer i has an associated bias $b_{i,j}$ and a vector of weights

$$\boldsymbol{w}_{i,j} = \begin{bmatrix} \boldsymbol{w}_{i,j,1} \\ \dots \\ \boldsymbol{w}_{i,j,m_{i-1}} \end{bmatrix}$$

where m_i denoted the number of nodes in a layer

Neural networks

We can arrange the weight vectors in a layer matrix, and the biases in an array

$$\boldsymbol{W}_j = \begin{bmatrix} \boldsymbol{w}_{j,1} \dots \boldsymbol{w}_{j,m_j} \end{bmatrix} \;, \quad \boldsymbol{b}_j = \begin{bmatrix} \boldsymbol{b}_{j,1} \\ \dots \\ \boldsymbol{b}_{j,m_j} \end{bmatrix}$$

The input $\boldsymbol{n}_j$ of layer j can be expressed in terms of the output $\boldsymbol{x}_{j-1}$ of layer $j-1$ as

$$\boldsymbol{n}_j = \boldsymbol{W}_j^T \boldsymbol{x}_{j-1} + \boldsymbol{b}_j = \begin{bmatrix} \boldsymbol{w}_{j,1}^T \boldsymbol{x}_{j-1} + \boldsymbol{b}_{j,1} \\ \vdots \\ \boldsymbol{w}_{j,m_j}^T \boldsymbol{x}_{j-1} + \boldsymbol{b}_{j,m_j} \end{bmatrix}$$

with $\boldsymbol{x}_0 = \boldsymbol{x}$ representing the network input

Neural networks

The layer output is computed from the layer input by applying element-wise the nodes non-linearity

Assuming that all nodes share the same non-linear activation function h:

$$\boldsymbol{x}_j = \begin{bmatrix} h(\boldsymbol{n}_{j,1}) \\ \vdots \\ h(\boldsymbol{n}_{j,m_j}) \end{bmatrix} = \boldsymbol{h}(\boldsymbol{n}_j)$$

Note that we use the same letter, but in bold $\boldsymbol{h}$, to indicate the vector-valued function that applies the scalar function h to all its inputs

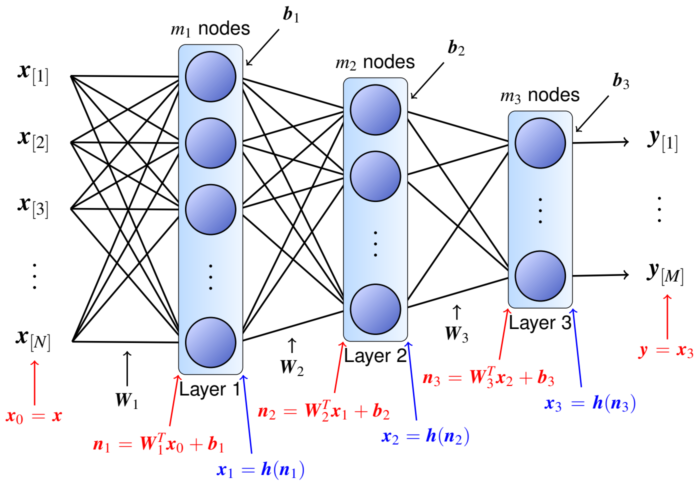
Each layer of the network computes a parametric linear transformation, followed by an element-wise non-parametric non-linear transformation[1]

$$\boldsymbol{f}_j(\boldsymbol{x}_{j-1}, \boldsymbol{W}_j, \boldsymbol{b}_j) = \boldsymbol{h}(\boldsymbol{W}_j^T \boldsymbol{x}_{j-1} + \boldsymbol{b}_j)$$

[1] These networks can be extended to include parametric non-linearities

Neural networks

Multi-layer feed-forward network:



m_1 nodes
$\boldsymbol{b}_1$
m_2 nodes
$\boldsymbol{b}_2$
m_3 nodes
$\boldsymbol{b}_3$
$\boldsymbol{x}_{[1]}$
$\boldsymbol{x}_{[2]}$
$\boldsymbol{x}_{[3]}$
$\boldsymbol{x}_{[N]}$
$\boldsymbol{y}_{[1]}$
$\boldsymbol{y}_{[M]}$
$\boldsymbol{W}_1$
$\boldsymbol{W}_2$
$\boldsymbol{W}_3$
Layer 1
Layer 2
Layer 3
$\boldsymbol{x}_0 = \boldsymbol{x}$
$\boldsymbol{n}_1 = \boldsymbol{W}_1^T \boldsymbol{x}_0 + \boldsymbol{b}_1$
$\boldsymbol{x}_1 = \boldsymbol{h}(\boldsymbol{n}_1)$
$\boldsymbol{n}_2 = \boldsymbol{W}_2^T \boldsymbol{x}_1 + \boldsymbol{b}_2$
$\boldsymbol{x}_2 = \boldsymbol{h}(\boldsymbol{n}_2)$
$\boldsymbol{n}_3 = \boldsymbol{W}_3^T \boldsymbol{x}_2 + \boldsymbol{b}_3$
$\boldsymbol{x}_3 = \boldsymbol{h}(\boldsymbol{n}_3)$
$\boldsymbol{y} = \boldsymbol{x}_3$

Neural networks

To compute the network function we "forward" the input vector through the different layers. For a network with m layers:

$$
\begin{aligned}
\boldsymbol{x}_0 &= \boldsymbol{x} \\
\boldsymbol{x}_1 &= \boldsymbol{f}_1(\boldsymbol{x}_0, \boldsymbol{W}_1, \boldsymbol{b}_1) = \boldsymbol{h}(\boldsymbol{W}_1^T \boldsymbol{x}_0 + \boldsymbol{b}_1) \\
\boldsymbol{x}_2 &= \boldsymbol{f}_2(\boldsymbol{x}_1, \boldsymbol{W}_2, \boldsymbol{b}_2) = \boldsymbol{h}(\boldsymbol{W}_2^T \boldsymbol{x}_1 + \boldsymbol{b}_2) \\
&\vdots \\
\boldsymbol{x}_{m-1} &= \boldsymbol{f}_{m-1}(\boldsymbol{x}_{m-2}, \boldsymbol{W}_{m-1}, \boldsymbol{b}_{m-1}) = \boldsymbol{h}(\boldsymbol{W}_{m-1}^T \boldsymbol{x}_{m-2} + \boldsymbol{b}_{m-1}) \\
\boldsymbol{x}_m &= \boldsymbol{f}_m(\boldsymbol{x}_{m-1}, \boldsymbol{W}_m, \boldsymbol{b}_m) = \boldsymbol{h}(\boldsymbol{W}_m^T \boldsymbol{x}_{m-1} + \boldsymbol{b}_m) \\
\boldsymbol{y} &= \boldsymbol{x}_m = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi})
\end{aligned}
$$

$f(\boldsymbol{x}, \boldsymbol{\Pi})$ is the function that corresponds to the network, and $\boldsymbol{\Pi}$ is the set of layer weights $\boldsymbol{W}_i$ and bias vectors $\boldsymbol{b}_i$

$\boldsymbol{f}_j(\boldsymbol{x}_{j-1}, \boldsymbol{W}_j, \boldsymbol{b}_j)$ is the function of the j-th network layer

Neural networks

The layers between the input and output layer are called hidden layers

A feed-forward neural network with hidden layers is also called multilayer perceptron (MLP)

It can be shown that any continuous function can be approximated up to a desired degree by a MLP of sufficient size

Neural networks

We can employ MLPs to represent a non-linear, parametric transformation of out input data

We can then classify our samples with a linear model in the transformed feature space induced by the MLP function

In a similar way, we can employ neural networks to compute non linear mappings to and from a lower dimensional space to achieve a form of non-linear dimensionality reduction

Neural networks

Binary problem: combine a binary logistic regression model with the MLP transformation

$$\log \frac{P(C=1|\boldsymbol{X}=\boldsymbol{x})}{P(C=0|\boldsymbol{X}=\boldsymbol{x})} = \boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + b$$

The class posterior probability can be expressed as

$$P(C=1|\boldsymbol{X}=\boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + b)$$

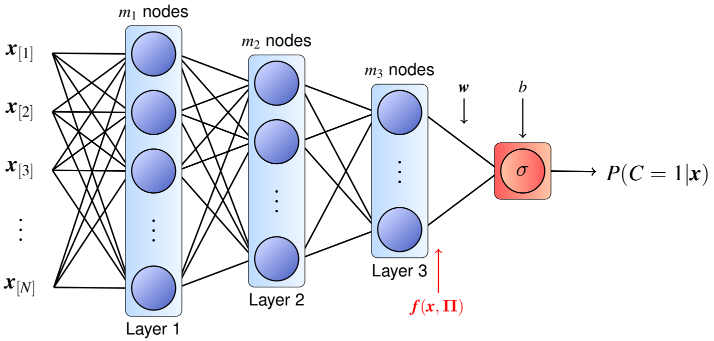
The network $f(\cdot, \boldsymbol{\Pi})$ represents the non-linear transformation of our data

$(\boldsymbol{w}, b)$ are the parameters of a linear logistic regression classifier

Neural networks

The map from the network output $f(\boldsymbol{x}, \boldsymbol{\Pi})$ to the class posterior probability $P(C = 1|\boldsymbol{x})$ can be represented as a sigmoid-activated network layer with a single node:

$$P(C=1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + b) = \widehat{f}(\boldsymbol{x}, \widehat{\boldsymbol{\Pi}}) \ , \quad \widehat{\boldsymbol{\Pi}} = (\boldsymbol{\Pi}, \boldsymbol{w}, b)$$



Neural networks

To estimate the network parameters we can optimize the logistic loss (also referred to as binary cross-entropy loss)

$$\begin{aligned}\boldsymbol{\Pi}^*, \boldsymbol{w}^*, b^* &= \arg\min_{\boldsymbol{\Pi}, \boldsymbol{w}, b} \frac{1}{N} \sum_{i=1}^{N} \log\left(1 + e^{-z_i(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\Pi}) + b)}\right) \\ &= \arg\min_{\boldsymbol{\Pi}, \boldsymbol{w}, b} -\frac{1}{N} \sum_{i=1}^{N} \left[c_i \log \sigma(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\Pi}) + b)\right. \\ &\qquad \left. + (1 - c_i) \log(1 - \sigma(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\Pi}) + b))\right]\end{aligned}$$

where $\boldsymbol{x}_i$ is the i-th sample in the dataset (pay attention not to confuse $\boldsymbol{x}_i$ in the loss expression with the output of the i-th network layer for sample $\boldsymbol{x}$)

Neural networks

In terms of the network $\widehat{f}$ we can express the optimization as

$$\widehat{\mathbf{\Pi}}^* = \arg\min_{\widehat{\mathbf{\Pi}}} -\frac{1}{N}\sum_{i=1}^{N}\left[c_i \log \widehat{f}(\boldsymbol{x}_i, \widehat{\mathbf{\Pi}}) + (1-c_i)\log(1-\widehat{f}(\boldsymbol{x}_i, \widehat{\mathbf{\Pi}}))\right]$$

As for linear logistic regression models, we cannot directly optimize the objective function

Again, we can rely on numerical optimization

Typically, numerical solvers require that we are able to compute both the objective function value (forward run), and its gradient

Neural networks

We thus turn our attention to the computation of the gradient of a loss function that depends on the outputs of a neural network

Let $\mathcal{L}(\boldsymbol{W}_1, \boldsymbol{b}_1, \ldots \boldsymbol{W}_m, \boldsymbol{b}_m)$ be the objective function that we want to minimize

For binary classification, $\mathcal{L}$ is the average logistic loss computed from the network $\widehat{f}$ (i.e., the extended network that includes $m-1$ feature transformation layers and, as last layer, the single-node, sigmoidal linear classification layer)

$$\begin{aligned}\mathcal{L}(\boldsymbol{W}_1, \boldsymbol{b}_1, \ldots \boldsymbol{W}_m, \boldsymbol{b}_m) = \\ \arg\min_{\boldsymbol{W}_1, \boldsymbol{b}_1, \ldots \boldsymbol{W}_m, \boldsymbol{b}_m} -\frac{1}{N} \sum_{i=1}^{N} \Big[c_i \log \widehat{f}(\boldsymbol{x}_i, \boldsymbol{W}_1, \boldsymbol{b}_1, \ldots \boldsymbol{W}_m, \boldsymbol{b}_m) \\ + (1 - c_i) \log(1 - \widehat{f}(\boldsymbol{x}_i, \boldsymbol{W}_1, \boldsymbol{b}_1, \ldots \boldsymbol{W}_m, \boldsymbol{b}_m)) \Big]\end{aligned}$$

Neural networks

In terms of the network layer, the loss can be represented as

$$\mathcal{L}(\boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_m, \boldsymbol{b}_m) = \frac{1}{N} \sum_{i=1}^{N} \ell(\widehat{f}(\boldsymbol{x}_i, \boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_m, \boldsymbol{b}_m), c_i)$$

where ℓ is a loss function, e.g., the logistic loss

$$\ell(y, c) = -\left[c \log y + (1-c) \log(1-y)\right]$$

The gradient of $\mathcal{L}$ requires computing the partial derivatives of $\mathcal{L}$ with respect to the network parameters $\boldsymbol{W}_{i,jk}$ and $\boldsymbol{b}_{i,j}$

The derivatives can be computed using the standard chain rule

Neural networks

Let's consider the derivatives with respect to the parameters of the last layer first

The parameters are $\boldsymbol{W}_m, \boldsymbol{b}_m$

We want to compute the partial derivatives $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{m,ij}}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_{m,i}}$

We have

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{m,ij}} = \frac{1}{N} \sum_{i=1} \frac{\partial \ell}{\partial \boldsymbol{W}_{m,jk}}$$

We thus need to compute the term $\frac{\partial \ell}{\partial \boldsymbol{W}_{m,jk}}$, i.e., the derivative of the loss function ℓ

Neural networks

We recall that, as a function of $\boldsymbol{W}_m$, the loss can be expressed as

$$\ell(f(\boldsymbol{x}, \boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_m, \boldsymbol{b}_m), c) = \ell(\boldsymbol{x}_m, c) = \ell(\boldsymbol{h}(\boldsymbol{W}_m^T \boldsymbol{x}_{m-1} + \boldsymbol{b}_m), c_i)$$

where $\boldsymbol{x}_m = \boldsymbol{h}(\boldsymbol{W}_m^T \boldsymbol{x}_{m-1} + \boldsymbol{b}_m)$ is the output of the last layer of the network, which depends directly on $\boldsymbol{W}_m$ and the output of the $(m-1)$-th layer $\boldsymbol{x}_{m-1}$

Pay attention that $\boldsymbol{x}_m$ is the output of the last layer for input $\boldsymbol{x}$, it's not the m-th sample of the dataset

In the following we will consider a single sample $\boldsymbol{x}$ to avoid having too many indices For the same reason, we will not consider single elements of $\boldsymbol{W}_{m,jk}$ but derive an expression for the gradient of ℓ

Neural networks

The gradient of ℓ is defined as the (row) vector of partial derivatives of ℓ with respect to all the parameters

In the following, we consider the gradient components that correspond to the terms of $\boldsymbol{W}_m$ (and $\boldsymbol{b}_m$):

$$\nabla_{\boldsymbol{W}_m}\ell = \left[\frac{\partial \ell}{\partial \boldsymbol{W}_{m,11}} \dots \frac{\partial \ell}{\partial \boldsymbol{W}_{m,1d_m}}, \frac{\partial \ell}{\partial \boldsymbol{W}_{m,21}} \dots \frac{\partial \ell}{\partial \boldsymbol{W}_{m,2d_m}} \dots \frac{\partial \ell}{\partial \boldsymbol{W}_{m,d_{m-1}1}} \dots \frac{\partial \ell}{\partial \boldsymbol{W}_{m,d_{m-1}d_m}}\right]$$

$$\nabla_{\boldsymbol{b}_m}\ell = \left[\frac{\partial \ell}{\partial \boldsymbol{b}_{m,1}} \dots \frac{\partial \ell}{\partial \boldsymbol{b}_{m,d_m}}\right]$$

where d_m and d_{m-1} are the size of layers m and $m-1$

Neural networks

Let's consider a scalar function $g : \mathbb{R}^q \rightarrow \mathbb{R}$, a vector function $\boldsymbol{h} : \mathbb{R}^p \rightarrow \mathbb{R}^q$, with components $\boldsymbol{h}(z) = \begin{bmatrix} \boldsymbol{h}_1(z) \\ \vdots \\ \boldsymbol{h}_p(z) \end{bmatrix}$

Let $f(z) = g(\boldsymbol{h}(z))$. The chain rule allows computing the gradient of f w.r.t. z from the gradient of $g(\boldsymbol{h})$ w.r.t. its input $\boldsymbol{h}$

$$\nabla_{\boldsymbol{h}} \boldsymbol{g} = \begin{bmatrix} \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}_1} & \cdots & \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{h}_q} \end{bmatrix}$$

and the matrix of partial derivatives (Jacobian matrix) of $\boldsymbol{h}(z)$

$$\frac{\partial \boldsymbol{h}}{\partial z} = \begin{bmatrix} \frac{\partial \boldsymbol{h}_1}{\partial z_1} & \cdots & \frac{\partial \boldsymbol{h}_1}{\partial z_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial \boldsymbol{h}_q}{\partial z_1} & \cdots & \frac{\partial \boldsymbol{h}_q}{\partial z_p} \end{bmatrix}$$

The gradient of $\boldsymbol{f}$ is obtained as

$$\nabla_z f = \nabla_{\boldsymbol{h}} g \cdot \frac{\partial \boldsymbol{h}}{\partial z}$$

Neural networks

We can apply the chain rule to the derivation of the partial gradient of the loss function w.r.t. the terms $\boldsymbol{W}_m$. We recall that

$$\boldsymbol{x}_m = \boldsymbol{h}(\boldsymbol{n}_m) \ , \quad \boldsymbol{n}_m = \boldsymbol{W}_m^T \boldsymbol{x}_{m-1} + \boldsymbol{b}_m$$

Thus

$$\nabla_{\boldsymbol{W}_m} \ell = \nabla_{\boldsymbol{x}_m} \ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m} \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{W}_m}$$

The left-most term is the loss function gradient. For example, for binary logistic regression $\boldsymbol{x}_m$ is a scalar, and the gradient is thus a 1-element vector:

$$\ell(\boldsymbol{x}_m, c) = c \log \boldsymbol{x}_m + (1-c) \log(1 - \boldsymbol{x}_m)$$
$$\nabla_{\boldsymbol{x}_m} \ell = \left[\tfrac{c}{\boldsymbol{x}_m} - \tfrac{1-c}{1-\boldsymbol{x}_m} \right]$$

Neural networks

The term $\frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m}$ is the matrix of partial derivatives of the nodes non-linearity $\boldsymbol{h}$

Since function $\boldsymbol{h}$ consists of element-wise non-linearities, we can compute the matrix of partial derivatives as

$$\frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m} = \begin{bmatrix} \left.\frac{dh(x)}{dx}\right|_{\boldsymbol{n}_{m,1}} & 0 & \dots & 0 \\ 0 & \left.\frac{dh(x)}{dx}\right|_{\boldsymbol{n}_{m,2}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \left.\frac{dh(x)}{dx}\right|_{\boldsymbol{n}_{m,d_m}} \end{bmatrix}$$

$\left.\frac{dh(x)}{dx}\right|_{y}$ is the derivative of the non-linear function $h(x)$ evaluated at y.

For example: $h(x) = \sigma(x) \rightarrow \frac{dh}{dx}(x) = \sigma(x)(1 - \sigma(x))$
$h(x) = \tanh(x) \rightarrow \frac{dh}{dx}(x) = 1 - \tanh^2(x)$

Neural networks

We observe that, given the gradient of ℓ with respect to the network outputs, to compute the gradient w.r.t. $\boldsymbol{W}_m$ we must first compute the product

$$\nabla_{\boldsymbol{x}_m}\ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m}$$

Since $\frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m}$ is diagonal, we can compute efficiently the result by multiplying each element of $\nabla_{\boldsymbol{x}_m}\ell$ by the corresponding element of the diagonal of $\frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m}$

Let

$$\boldsymbol{v}_m = \nabla_{\boldsymbol{x}_m}\ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m}$$

We further need to compute

$$\nabla_{\boldsymbol{W}_m}\ell = \boldsymbol{v}_m \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{W}_m}$$

i.e. the product of the vector $\boldsymbol{v}_m$ and the matrix of partial derivatives of input of the network last layer with respect to the layer weights

Neural networks

Since $\boldsymbol{W}_m$ is a matrix, it's useful to represent in matrix form the subset of the gradient that corresponds to derivatives w.r.t. $\boldsymbol{W}_{m,jk}$:

$$\nabla_{\boldsymbol{W}_m}\ell = \begin{bmatrix} \nabla_{\boldsymbol{w}_1}\ell \\ \vdots \\ \nabla_{\boldsymbol{w}_{d_m}}\ell \end{bmatrix} = \begin{bmatrix} \frac{\partial \ell}{\partial \boldsymbol{W}_{m,11}} & \cdots & \frac{\partial \ell}{\partial \boldsymbol{W}_{m,d_{m-1}1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \boldsymbol{W}_{m,1d_m}} & \cdots & \frac{\partial \ell}{\partial \boldsymbol{W}_{m,d_{m-1}d_m}} \end{bmatrix}$$

Note: a row of $\nabla_{\boldsymbol{W}_m}\ell$ contains the derivatives with respect to the elements of a column of $\boldsymbol{W}_m = \begin{bmatrix} \boldsymbol{w}_1 & \dots & \boldsymbol{w}_{d_m} \end{bmatrix}$

The gradient $\nabla_{\boldsymbol{W}_m}\ell$ can be computed in matrix form as

$$\nabla_{\boldsymbol{W}_m}\ell = \boldsymbol{v}_m \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{W}_m} = \boldsymbol{x}_{m-1}\boldsymbol{v}_m$$

Similarly, we can show that

$$\nabla_{\boldsymbol{b}_m}\ell = \boldsymbol{v}_m$$

Neural networks

Let's now consider the set of derivatives with respect to the previous-from-last layer parameters $\boldsymbol{W}_{m-1}, \boldsymbol{b}_{m-1}$

As before, we can express the dependency of the loss on these parameters through $\boldsymbol{x}_m$:

$$\begin{aligned} \boldsymbol{x}_m &= \boldsymbol{h}(\boldsymbol{n}_m) \\ \boldsymbol{n}_m &= \boldsymbol{W}_m^T \boldsymbol{x}_{m-1} + \boldsymbol{b}_m \\ \boldsymbol{x}_{m-1} &= \boldsymbol{h}(\boldsymbol{n}_{m-1}) \\ \boldsymbol{n}_{m-1} &= \boldsymbol{W}_{m-1}^T \boldsymbol{x}_{m-2} + \boldsymbol{b}_{m-1} \end{aligned}$$

Applying the chain rule, we can write

$$\nabla_{\boldsymbol{W}_m} \ell = \nabla_{\boldsymbol{x}_m} \ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m} \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{x}_{m-1}} \cdot \frac{\partial \boldsymbol{x}_{m-1}}{\partial \boldsymbol{n}_{m-1}} \cdot \frac{\partial \boldsymbol{n}_{m-1}}{\partial \boldsymbol{W}_{m-1}}$$

Neural networks

We can observe that the product of the first three terms corresponds to the set of partial derivatives of the loss w.r.t. the $(m-1)$-th layer outputs

$$\nabla_{\boldsymbol{x}_{m-1}} \ell = \nabla_{\boldsymbol{x}_m} \ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m} \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{x}_{m-1}}$$

$$\nabla_{\boldsymbol{W}_{m-1}} \ell = \nabla_{\boldsymbol{x}_{m-1}} \ell \cdot \frac{\partial \boldsymbol{x}_{m-1}}{\partial \boldsymbol{n}_{m-1}} \cdot \frac{\partial \boldsymbol{n}_{m-1}}{\partial \boldsymbol{W}_{m-1}}$$

We can compare with the derivatives w.r.t $\boldsymbol{W}_m$:

$$\nabla_{\boldsymbol{W}_m} \ell = \nabla_{\boldsymbol{x}_m} \ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m} \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{W}_m}$$

Neural networks

We observe that in both cases we can express the gradient as a vector-matrix multiplication

The vector term represents the gradient of the loss w.r.t. the layer outputs

The matrix terms are partial derivatives of the layer function with respect to the layer parameters, computed at the previous layer outputs

Neural networks

Iterating this process, we can verify that

$$\nabla_{\boldsymbol{W}_k} \ell = \boldsymbol{v}_k \cdot \frac{\partial \boldsymbol{n}_k}{\partial \boldsymbol{W}_k}$$
$$\nabla_{\boldsymbol{b}_k} \ell = \boldsymbol{v}_k \cdot \frac{\partial \boldsymbol{n}_k}{\partial \boldsymbol{b}_k}$$

with

$$\boldsymbol{v}_k = \nabla_{\boldsymbol{x}_k} \ell \cdot \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{n}_k}$$

In matrix form we can express the gradients as

$$\nabla_{\boldsymbol{W}_k} \ell(\boldsymbol{x}_m, c) = \boldsymbol{x}_{k-1} \boldsymbol{v}_k \ , \quad \nabla_{\boldsymbol{b}_k} \ell(\boldsymbol{x}_m, c) = \boldsymbol{v}_k$$

We can thus compute the gradient by computing the terms $\boldsymbol{v}_k$ for each layer

Neural networks

In general, we can verify that, for layer k, we have

$$\nabla_{\boldsymbol{x}_k}\ell = \nabla_{\boldsymbol{x}_{k+1}}\ell \cdot \frac{\partial \boldsymbol{x}_{k+1}}{\partial \boldsymbol{n}_{k+1}} \cdot \frac{\partial \boldsymbol{n}_{k+1}}{\partial \boldsymbol{x}_k}$$

$$\boldsymbol{v}_k = \boldsymbol{v}_{k+1} \cdot \frac{\partial \boldsymbol{n}_{k+1}}{\partial \boldsymbol{x}_k} \cdot \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{n}_k}$$

Furthermore, we can show that

$$\nabla_{\boldsymbol{x}_k}\ell = \boldsymbol{v}_{k+1} \cdot \frac{\partial \boldsymbol{n}_{k+1}}{\partial \boldsymbol{x}_k}$$

$$\boldsymbol{v}_k = \nabla_{\boldsymbol{x}_k}\ell \cdot \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{n}_k}$$

Neural networks

Each term $\boldsymbol{v}_k$ corresponds to the gradient of the loss with respect to the k-th layer inputs

$$\boldsymbol{v}_k = \nabla_{\boldsymbol{n}_k} \ell$$

Starting from the last layer, we can efficiently compute $\nabla_{\boldsymbol{x}_k} \ell$ and $\boldsymbol{v}_k = \nabla_{\boldsymbol{n}_k} \ell$ through an iterative procedure that starts from $\nabla_{\boldsymbol{x}_m} \ell$:

$$\begin{tikzcd}
\nabla_{\boldsymbol{x}_m} \ell \arrow[d] & \nabla_{\boldsymbol{x}_{m-1}} \ell = \boldsymbol{v}_m \cdot \frac{\partial \boldsymbol{n}_m}{\partial \boldsymbol{x}_{m-1}} \arrow[d] & \nabla_{\boldsymbol{x}_{m-2}} \ell = \boldsymbol{v}_m \cdot \frac{\partial \boldsymbol{n}_{m-1}}{\partial \boldsymbol{x}_{m-2}} \arrow[d] \\
\boldsymbol{v}_m = \nabla_{\boldsymbol{x}_m} \ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m} \arrow[ur] & \boldsymbol{v}_{m-1} = \nabla_{\boldsymbol{x}_{m-1}} \ell \cdot \frac{\partial \boldsymbol{x}_{m-1}}{\partial \boldsymbol{n}_{m-1}} \arrow[ur] & \boldsymbol{v}_{m-2} = \nabla_{\boldsymbol{x}_{m-2}} \ell \cdot \frac{\partial \boldsymbol{x}_{m-2}}{\partial \boldsymbol{n}_{m-2}}
\end{tikzcd}$$

The procedure is called back-propagation

At each step, we are propagating the loss gradient backwards through the layers

Neural networks

We already showed how to compute the terms $\frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{n}_k}$

It remains to see how to compute the terms $\frac{\partial \boldsymbol{n}_k}{\partial \boldsymbol{x}_{k-1}}$

Since $\boldsymbol{n}_k = \boldsymbol{W}_k^T \boldsymbol{x}_{k-1} + \boldsymbol{b}_k$ we can show that

$$\frac{\partial \boldsymbol{n}_k}{\partial \boldsymbol{x}_{k-1}} = \boldsymbol{W}_k$$

and the iterative procedure becomes

$$\boldsymbol{v}_k = \nabla_{\boldsymbol{x}_k} \ell \cdot \frac{\partial \boldsymbol{x}_k}{\partial \boldsymbol{n}_k}$$
$$\nabla_{\boldsymbol{x}_{k-1}} \ell = \boldsymbol{v}_k \boldsymbol{W}_k$$

Neural networks

Forward run (compute the network outputs):

$$\boldsymbol{n}_1 = \boldsymbol{W}_1^T \boldsymbol{x}_0 + \boldsymbol{b}_1$$
$$\boldsymbol{x}_1 = \boldsymbol{h}(\boldsymbol{n}_1)$$
$$\boldsymbol{n}_2 = \boldsymbol{W}_2^T \boldsymbol{x}_1 + \boldsymbol{b}_2$$
$$\boldsymbol{x}_2 = \boldsymbol{h}(\boldsymbol{n}_2)$$
$$\dots$$
$$\boldsymbol{n}_m = \boldsymbol{W}_m^T \boldsymbol{x}_{m-1}$$
$$\boldsymbol{x}_m = \boldsymbol{h}(\boldsymbol{n}_m)$$

(Column vectors)

Backward run (compute the terms required for gradient computation):

$$\boldsymbol{v}_m = \nabla_{\boldsymbol{x}_m} \ell \cdot \frac{\partial \boldsymbol{x}_m}{\partial \boldsymbol{n}_m}$$
$$\nabla_{\boldsymbol{x}_{m-1}} \ell = \boldsymbol{v}_m \boldsymbol{W}_m$$
$$\boldsymbol{v}_{m-1} = \nabla_{\boldsymbol{x}_{m-1}} \ell \cdot \frac{\partial \boldsymbol{x}_{m-1}}{\partial \boldsymbol{n}_{m-1}}$$
$$\nabla_{\boldsymbol{x}_{m-2}} \ell = \boldsymbol{v}_{m-1} \boldsymbol{W}_{m-1}$$
$$\dots$$
$$\boldsymbol{v}_1 = \nabla_{\boldsymbol{x}_2} \ell \cdot \frac{\partial \boldsymbol{x}_2}{\partial \boldsymbol{n}_1}$$
$$\nabla_{\boldsymbol{x}_0} \ell = \boldsymbol{v}_1 \boldsymbol{W}_1$$

(Row vectors)

Neural networks

Once we are able to compute gradients, we can apply a numerical solver to find a local minimum of our objective

While for small datasets methods like L-BFGS may provide fast and robust results, for larger datasets these methods may become expensive

Faster approaches based on Gradient Descent (GD) are typically employed

GD: starting from an initial value for the network weights $\boldsymbol{W}_1^0, \boldsymbol{b}_1^0 \ldots \boldsymbol{W}_m^0, \boldsymbol{b}_m^0$ the weights are iteratively updated according to

$$\boldsymbol{W}_k^t = \boldsymbol{W}_k^{t-1} - \alpha_t \nabla^T_{\boldsymbol{W}_k^{t-1}} \mathcal{L} \ , \quad \boldsymbol{b}_k^t = \boldsymbol{b}_k^{t-1} - \alpha_t \nabla^T_{\boldsymbol{b}_k^{t-1}} \mathcal{L}$$

Neural networks

The coefficient α_t is called learning rate, and controls the strength of the weights update

GD convergence is guaranteed if

$$\sum_t \alpha_t = \infty \qquad \sum_t \alpha_t^2 < \infty$$

However, in practice the number of iterations can be heavily influenced by the learning rate schedule

Since neural networks typically require large training sets, the standard GD approach is not practical, as it has the same drawback of L-BFGS: it requires a full iteration over the whole dataset to compute the loss gradient

$$\nabla \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell(\boldsymbol{x}_i, c_i)$$

Neural networks

Toa ddress this issue, training is usually performed using Stochastic Gradient Descent (SGD) over batches

A batch is a set of randomly selected samples

We approximate the gradient

$$\nabla\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\nabla\ell(\boldsymbol{x}_i, c_i)$$

with

$$\nabla\mathcal{L} \approx \frac{1}{K}\sum_{\boldsymbol{x}_i\in B}\nabla\ell(\boldsymbol{x}_i, c_i)$$

where B is a set of K samples

Typical batch sizes range in the tens to few hundreds

Neural networks

Training is also usually organized in epochs. At each epoch:

1. Randomly sample a batch using samples that have been not employed during the current epoch yet
2. Compute the batch gradient and update the weights using the approximated gradient
3. Update the learning rate
4. Repeat from 1. until the whole dataset has been used

A limitation of SGD is that its performance relies heavily on the selection of a good learning rate schedule

Large values of α may overshoot the local minimum, but small values may make slow progress

Neural networks

We can extend SGD by incorporating a momentum term

The momentum term performs an exponential smoothing of the gradient

At iteration t we compute the update

$$\begin{aligned}
\Delta^t_{\boldsymbol{W}_k} &= \eta \Delta^{t-1}_{\boldsymbol{W}_k} - \alpha_t \nabla^T_{\boldsymbol{W}^{t-1}_k} \mathcal{L} \\
\Delta^t_{\boldsymbol{b}_k} &= \eta \Delta^{t-1}_{\boldsymbol{b}_k} - \alpha_t \nabla^T_{\boldsymbol{b}^{t-1}_k} \mathcal{L} \\
\boldsymbol{W}^t_k &= \boldsymbol{W}^{t-1}_k + \Delta^t_{\boldsymbol{W}_k} \\
\boldsymbol{b}^t_k &= \boldsymbol{b}^{t-1}_k + \Delta^t_{\boldsymbol{b}_k}
\end{aligned}$$

where η is a constant factor

More sophisticated approaches have been recently introduced (RMSProp, Adam) to improve the convergence rate of SGD

Neural networks

We can extend neural networks to multiclass classification

As for the binary case, we can assume that a neural network $\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi})$ computes a non-linear feature transformation

We can pair the network output with a multiclass logistic regression model

The model defines the class posterior probabilities as

$$P(C = k|\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{\Pi}, \boldsymbol{x}) = \frac{e^{\boldsymbol{w}_k^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + \boldsymbol{b}_k}}{\sum_{j=1}^K e^{\boldsymbol{w}_j^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + \boldsymbol{b}_j}}$$

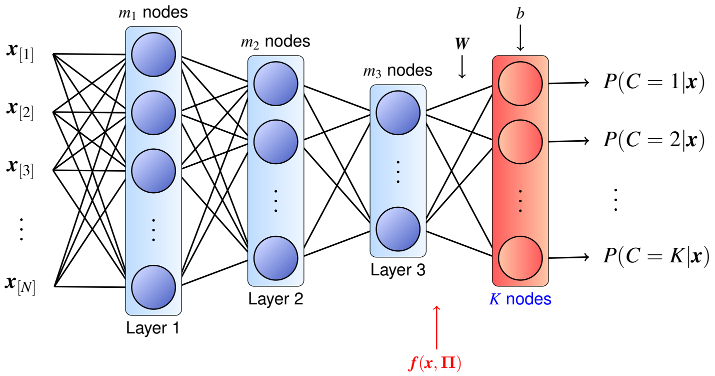
where $\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1 & \dots & \boldsymbol{w}_k \end{bmatrix}$ and K is the number of classes

We can estimate the model parameters by minimizing the cross-entropy

$$\arg\min_{\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{\Pi}} -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K z_{ik} \log \frac{e^{\boldsymbol{w}_k^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + b_k}}{\sum_{j=1}^K e^{\boldsymbol{w}_j^T \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\Pi}) + b_j}}$$

Neural networks

As for binary logistic regression, we can cast the whole model as a single neural network $\widehat{f}$ that incorporates also the logistic regression parameters $\boldsymbol{W}, \boldsymbol{b}$



Neural networks

The last layer activation function is called softmax activation

The output of each node is computed by applying the softmax function to its inputs

$$\boldsymbol{x}_m = s(\boldsymbol{n}_m) = \begin{bmatrix} \frac{e^{\boldsymbol{n}_{m,1}}}{\sum_{i=1}^{K} e^{\boldsymbol{n}_{m,i}}} \\ \vdots \\ \frac{e^{\boldsymbol{n}_{m,K}}}{\sum_{i=1}^{K} e^{\boldsymbol{n}_{m,i}}} \end{bmatrix}$$

Note that the layer does not strictly follow the feed-forward network topology we defined earlier, as the output values of the nodes of the last layer depend on the inputs of the other nodes of the layer

However, if we abstract the network at the layers level this does not introduce practical differences

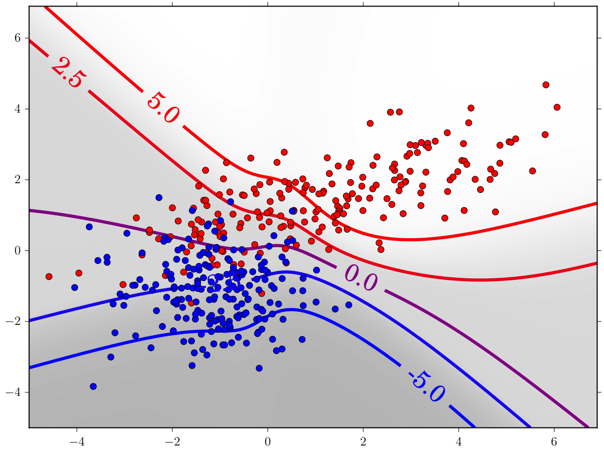
Neural networks

As in the binary case, we can employ back-propagation to compute the gradient of the loss with respect to the model parameters

Stochastic gradient descent or similar optimizers can then be employed to iteratively train the network weight and bias terms

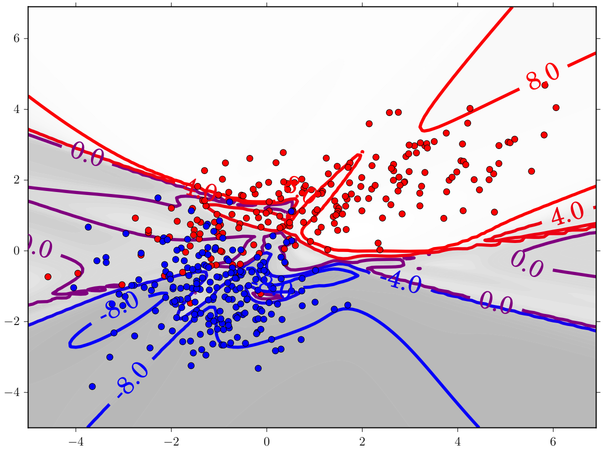
Neural networks

A binary 2D example



Neural networks

Overfitting can be much more dramatic than for linear logistic regression



Neural networks

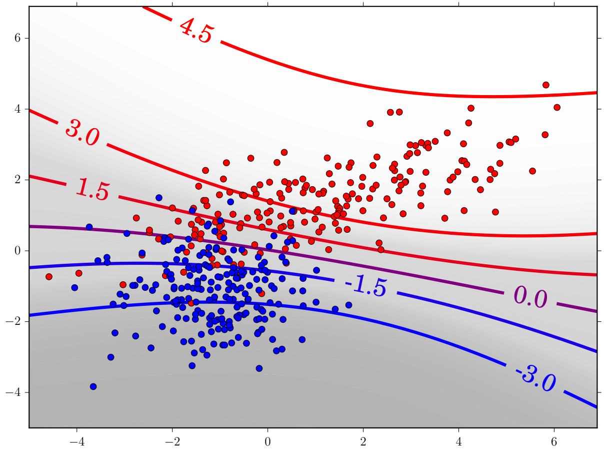
Different regularization strategies can be adopted

- L2 weights regularization
- Dropout
- Early stopping (computing error on validation set)

Neural networks

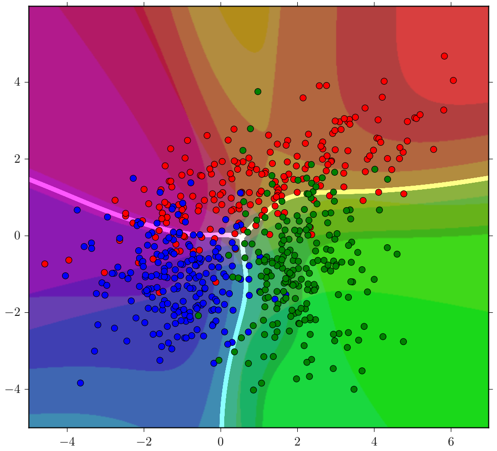
L2 regularization: penalize the squared norm of the weights

$$\arg\min_{\boldsymbol{W}_1,\boldsymbol{b}_1\ldots\boldsymbol{W}_m,\boldsymbol{b}_m} \mathcal{L}(\boldsymbol{W}_1,\boldsymbol{b}_1\ldots\boldsymbol{W}_m,\boldsymbol{b}_m) + \frac{\lambda}{2}\sum_{i=1}^{m}\|\boldsymbol{W}_m\|^2$$



Neural networks

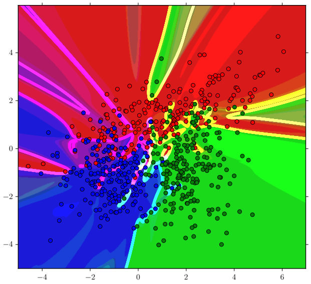
Multiclass (simple network)



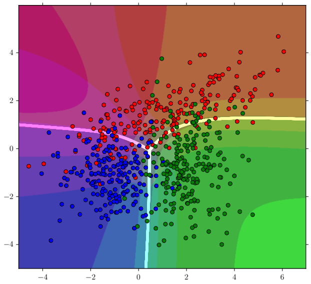
Neural networks

Multiclass (deep network)

No regularization



L2 regularization



Neural networks

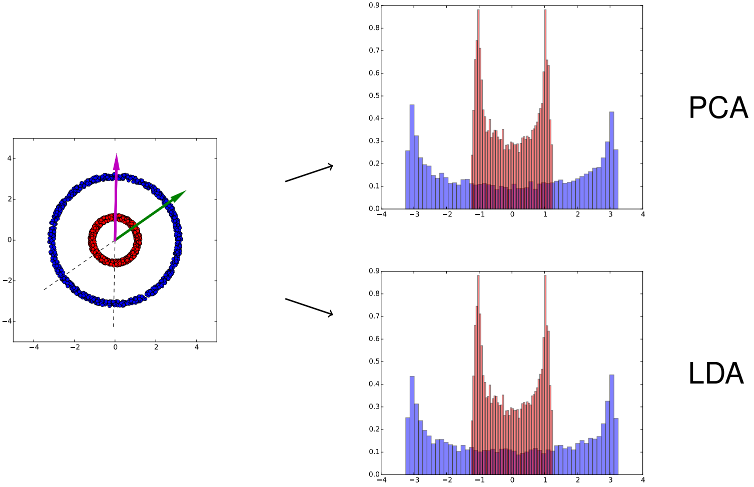
MNIST — Error rates for Neural Networks[2]

	No Reg.	L2 ($\lambda = 1e^{-5}$)	Dropout ($p = 0.5$)
MLP (Tanh) 512–512–512	1.9% [1.8%]	2.0% [1.7%]	1.5% [1.5%]
MLP (ReLU) 512–512–512	1.6% [1.5%]	1.7% [1.6%]	1.7% [1.4%]
MLP (ReLU) 1024–1024–1024–1024	1.6% [1.5%]	1.6% [1.4%]	$\boxed{\text{1.4\% [1.4\%]}}$

[2]Training set was split into development (90% of the data) and validation (10% of the data) sets to select the best performing model. The performance of the model with lowest error rate on the test set is shown in brackets.

Neural networks

Linear transformations are not always suited for our data



Neural networks

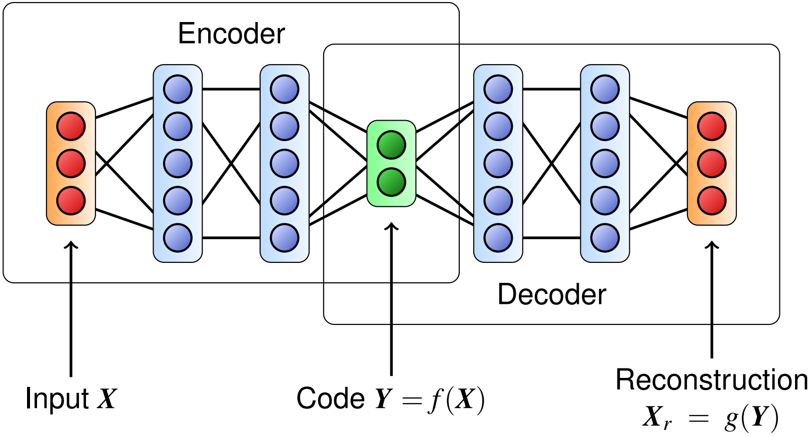
Neural networks can be used to discover the underlying data structure

Autoencoders are similar to PCA

- Unsupervised training (no guarantee that the resulting low–dimensional embedding is useful for classification)
- Minimize reconstruction error
- Used mainly to remove noise from samples (denoising autoencoders)
- Can be used also for dimensionality reduction

Neural networks

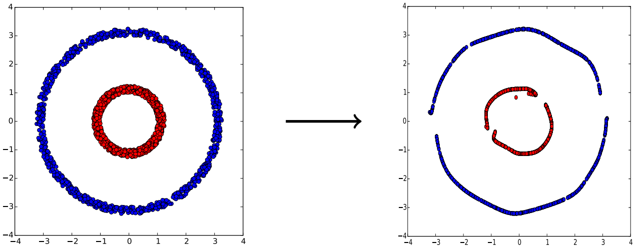
Autoencoder structure:


Encoder
Decoder
Input $\boldsymbol{X}$
Code $\boldsymbol{Y} = f(\boldsymbol{X})$
Reconstruction
$\boldsymbol{X}_r = g(\boldsymbol{Y})$

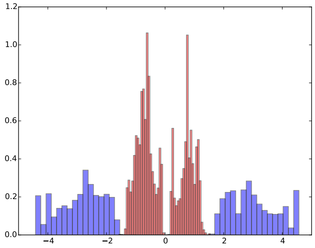
Neural networks

- The input layer has n nodes
- The central hidden layer has $m \ll n$ nodes (bottleneck)
- The central hidden layer acts as a compact representation of the input
- We optimize the network in order to minimize the reconstruction error $\|\boldsymbol{X} - \boldsymbol{X}_r\|^2$
- Denoising autoencoder: provide as input a noisy version of the samples and minimize the reconstruction error with respect to the clean sample

Neural networks

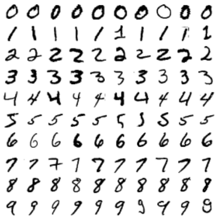


BN layer:



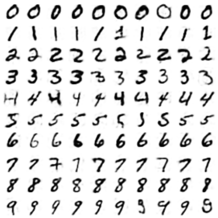
Neural networks

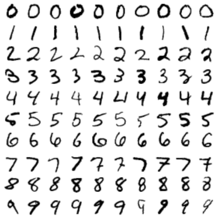
Autoencoder on MNIST — 50 bottleneck nodes



Train Set

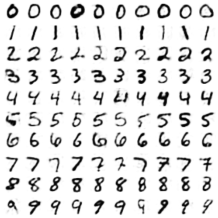
$\longrightarrow$





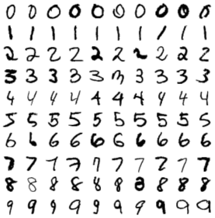
Test Set

$\longrightarrow$



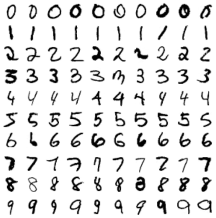
Neural networks

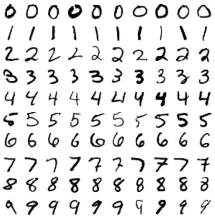
Autoencoder on MNIST — 50 bottleneck nodes — sub–sampled



Train Set

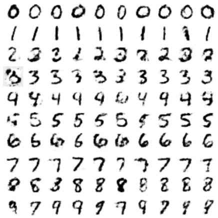
$\longrightarrow$





Test Set

$\longrightarrow$



Neural networks

Overfitting: A low reconstruction error on the training set does not guarantee a low reconstruction error on a different set

The problem is more evident with complex models.

Some strategies can be employed to reduce overfitting issues:

- Early stopping
  - Monitor loss over an held–out validation set
  - Stop the optimization when the error over the validation set starts increasing
- Model regularization (L2, dropout, ...)
- Choose the simplest model that is suitable for the task (Occam's razor)

Neural networks

Ad-hoc architectures have been proposed for different tasks

For example, for image processing dense layers are not too effective, since image characteristics are typically local and may appear in different places in the image

Fully connected layers would require too many parameters

For image processing convolutional networks are typically used

Neural networks

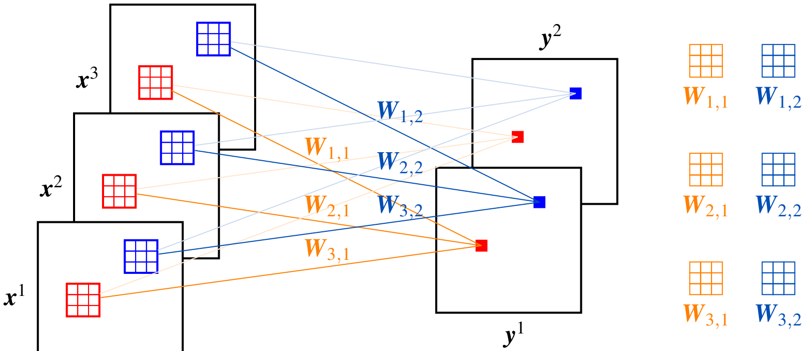
Convolutional networks behave like learned image filters

A convolutional layer receives as input a set of m 2-D channels (e.g., the image color channels) and outputs a set of n 2-D channels

The output is obtained by applying learnable convolutional filters on the different channels

Neural networks

A convolutional layer with 3x3 filters:



Neural networks

The input channels can be interpreted as 2-D maps

The output channels can also be interpreted as 2-D maps

For each combination input/output channel we have a convolutional filter (or, more precisely, a cross-correlation filter) - for example, a 3x3 filter can be

$$\boldsymbol{W}_{i,j} = \begin{bmatrix} \boldsymbol{W}_{i,j_{1,1}} & \boldsymbol{W}_{i,j_{1,2}} & \boldsymbol{W}_{i,j_{1,3}} \\ \boldsymbol{W}_{i,j_{2,1}} & \boldsymbol{W}_{i,j_{2,2}} & \boldsymbol{W}_{i,j_{2,3}} \\ \boldsymbol{W}_{i,j_{3,1}} & \boldsymbol{W}_{i,j_{3,2}} & \boldsymbol{W}_{i,j_{3,3}} \end{bmatrix}$$

We define the filter width as half the size of the filter matrix minus one, i.e. for a 3x3 filter the width is 1, for a 5x5 filter the width is 2 and so on

Neural networks

The output channel $\boldsymbol{y}^c$ is computed as

$$\boldsymbol{y}^c_{i,j} = h\left(\sum_{k=1}^{m} \left\langle \boldsymbol{x}^k_{[i-l,i+l],[j-l,j+l]}, \boldsymbol{W}_{k,c} \right\rangle + b_c\right) \quad , \quad c = 1 \ldots n$$

- l is the filter width
- $\boldsymbol{x}^k_{[i-l,i+l],[j-l,j+l]}$ is the sub-matrix of size $(2l+1) \times (2l+1)$ of the input channel $\boldsymbol{x}^k$ centered at (i,j)
- b_c is a bias term
- h is a non-linear function (e.g. ReLU or sigmoid)
- $\langle \cdot, \cdot \rangle$ is the inner product $\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \sum_{i,j} \boldsymbol{A}_{ij}\boldsymbol{B}_{ij}$
- m is the number of input channels, n is the number of output channels

Neural networks

The filters are also called kernels

Cross-correlation is not well defined for elements that are on the boundary of the input channels

To address this, either we compute cross-correlations only for valid inputs, or we add a padding, i.e., we employ default values for out-of-boundary elements (e.g. zeros)

We may also avoid computing cross-correlations for all input elements by computing correlations only for positions $(s \cdot i, s \cdot j)$, where s is a constant called stride. In practice, we compute correlations only for 1 in s input rows and columns

Neural networks

Convolutional neural networks typically also employ pooling layers, which aggregate information corresponding to different positions in the input channels

Pooling layers reduce the dimensionality of the input channels, thus increasing the receptive field (i.e. the components of the original input feature that may affect the output of the neuron) of a single neuron

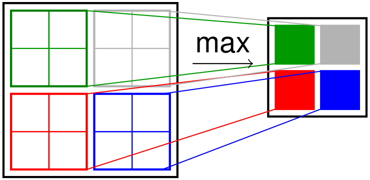
Neural networks

A commonly used pooling layer is the max-pooling layer

The max-pooling layer divides each input channel in $k \times k$ blocks and computes, as output, the maximum of the values of each block

For example, 2×2 pooling layer computes it outputs as

$$\boldsymbol{y}_{i,j}^{c} = max(\boldsymbol{x}_{2i,2j}^{c}, \boldsymbol{x}_{2i+1,2j}^{c}, \boldsymbol{x}_{2i,2j+1}^{c}, \boldsymbol{x}_{2i+1,2j+1}^{c})$$



Note that the output channels in this case have half the number of rows and columns of the input channels

Neural networks

MNIST — Error rates for Neural Networks[3]

	No Reg.	L2 ($\lambda = 1e^{-5}$)	Dropout ($p = 0.5$)
MLP (Tanh) 512–512–512	1.9% [1.8%]	2.0% [1.7%]	1.5% [1.5%]
MLP (ReLU) 512–512–512	1.6% [1.5%]	1.7% [1.6%]	1.7% [1.4%]
MLP (ReLU) 1024–1024–1024–1024	1.6% [1.5%]	1.6% [1.4%]	$\boxed{\text{1.4\% [1.4\%]}}$
ConvNet (ReLU)	1.1% [1.0%]	1.1% [1.0%]	$\boxed{\text{0.9\% [0.8\%]}}$

[3] Training set was split into development (90% of the data) and validation (10% of the data) sets to select the best performing model. The performance of the model with lowest error rate on the test set is shown in brackets.

Neural networks

Recent trends have seen networks becoming deeper rather than larger

When training deep networks gradient methods may incur in problems

Typically, back-propagation results in very small values (numerically zeros) for the gradient of the initial layers for networks that employ traditional sigmoid or hyperbolic tangent non linearities

Gradient methods may not be able to progress

Neural networks

Alongside using different activation functions such as ReLU, a typical approach is to introduce residual connection

Rather than computing the input-output transformation function, residual networks compute a residual function

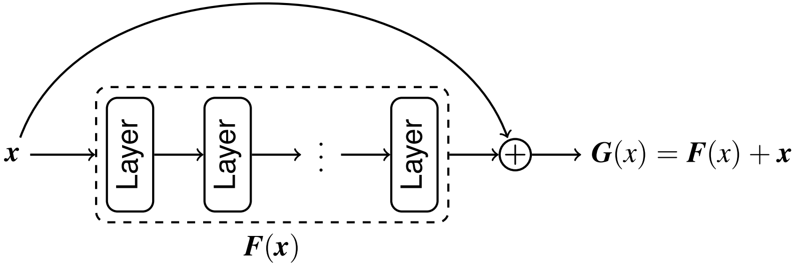
Let $\boldsymbol{G}(\boldsymbol{x})$ represent the function that we want to compute, with $\boldsymbol{G} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ (note that the functoin input and output have the same size, as typically happens for convolutional networks)

The residual blocks provide a model for $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{G}(\boldsymbol{x}) - \boldsymbol{x}$

Function $\boldsymbol{G}(\boldsymbol{x})$ can then be computed as $\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{x}) + \boldsymbol{x}$

Neural networks

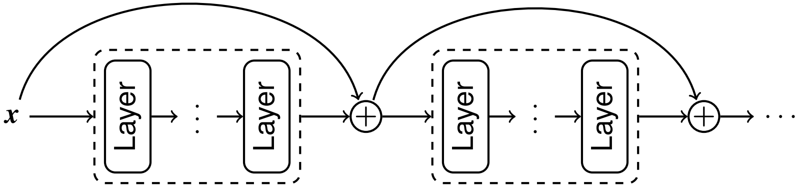
In practice, the effect can be obtained by introducing skip connections



$\boldsymbol{x}$
Layer
Layer
$\vdots$
Layer
$\boldsymbol{G}(x) = \boldsymbol{F}(x) + \boldsymbol{x}$
$\boldsymbol{F}(\boldsymbol{x})$

Neural networks

We can combine several residual blocks to obtain a deep residual network



Residual blocks allow for effective training of network with hundreds of layers