Sapienza University of Rome

Master in Artificial Intelligence and Robotics
Master in Engineering in Computer Science

Machine Learning

A.Y. 2020/2021

Prof. L. Iocchi, F. Patrizi

Sapienza University of Rome, Italy - Machine Learning (2020/2021)

SECOND PART OF THE COURSE

11. Artificial Neural Networks

L. Iocchi, F. Patrizi

we can apply NN both for classification and regression

so far we have considered only linear model while NN are provided for non linear model (X and Y are non linear)
In NN we don't need to specify the kernel but the network is able to learn which is the best

with contributions from Valsamis Ntouskos

Overview

- Feedforward networks
- Architecture design
- Cost functions
- Activation functions
- Gradient computation (back-propagation)
- Learning (stochastic gradient descent)
- Regularization

References

Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning - Chapters 6, 7, 8. `http://www.deeplearningbook.org`

Artificial Neural Networks (ANN)

Alternative names:

- Neural Networks - (NN)
- Feedforward Neural Networks - (FNN)
- Multilayer Perceptrons - (MLP)

initially we will use these as synonims

Function approximator using a parametric model.

Suitable for tasks described as associating a vector to another vector.

Artificial Neural Networks (ANN)

Goal:

Estimate some function $f : X \to Y$, with $Y = \{C_1, \ldots, C_k\}$ or $Y = \Re$

Data:

$D = \{(\mathbf{x}_n, t_n)_{n=1}^N\}$ such that $t_n \approx f(\mathbf{x}_n)$

Framework:

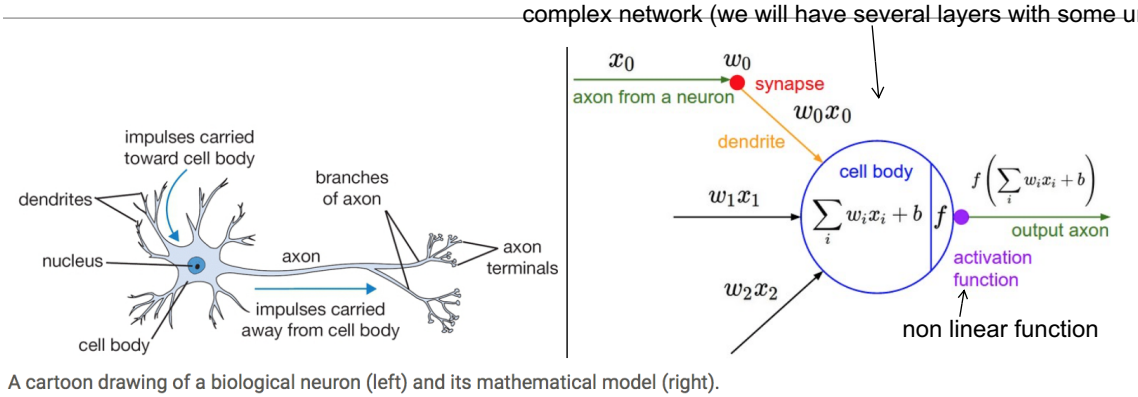
Define $y = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ and learn parameters $\boldsymbol{\theta}$ so that $\hat{f}$ approximates f.

the task is to define a parametric model

Feedforward Networks

Draw inspiration from brain structures

this is a processing unit but we can combine several of these obtaining a complex network (we will have several layers with some units in each)



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Image from Isaac Changhau `https://isaacchanghau.github.io`

Hidden layer output can be seen as an array of **unit** (neuron) activations based on the connections with the previous units

Note: Only use some insights, they are not a model of the brain!

Feedforward Networks - Terminology

the process procede from input to output in only one direction (---->)
These models are called feedforward because information flows through the function being evaluated from x , through the intermediate computations used to define f , and finally to the output y . There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks

Feedforward information flows from input to output without any loop

Networks f is a composition of elementary functions in an acyclic graph

Example:

$$f(\mathbf{x};\boldsymbol{\theta}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x};\boldsymbol{\theta}^{(1)});\boldsymbol{\theta}^{(2)});\boldsymbol{\theta}^{(3)})$$

function that transforms its input

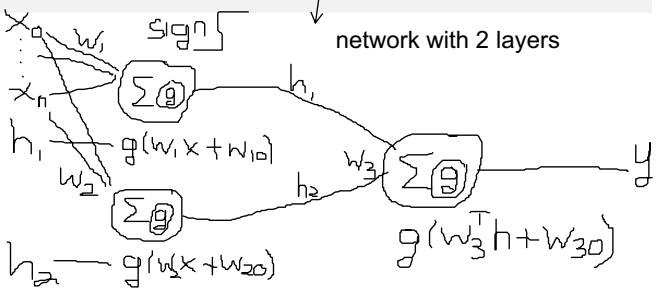
where:

$f^{(m)}$ the m-th layer of the network

and

$\boldsymbol{\theta}^{(m)}$ the corresponding parameters

Feedforward Networks - Terminology



FNNs are **chain structures**

The length of the chain is the **depth** of the network

the dimensionality of these hidden layers determines the width of the model

Final layer also called **output layer**

Deep learning follows from the use of networks with a large number of layers (large depth)

Feedforward Networks

Why FNNs?

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they can be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

Linear models cannot model interaction between input variables

To extend linear models to represent nonlinear functions of x , we can apply the linear model not to x itself but to a transformed inputϕ(x), where ϕ is a non linear ransformation. equivalently we can apply the kernel trick to obtain a non linear algorithm

Kernel methods require the choice of suitable kernels

- use generic kernels e.g. RBF, polynomial, etc. (convex problem)
- use hand-crafted kernels - application specific (convex problem)

FNN leaning:
complex combination of many parametric functions (non-convex problem)

The XOR function ("exclusive or") is an operation on two binary values, x1 and x2 . When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0

XOR Example - Linear model

Learning the XOR function - 2D input and 1D output

if we use a linear kernel will not have a good solution because the dataset is not linear separable

Dataset: $D = \{((0,0)^T, 0), ((0,1)^T, 1), ((1,0)^T, 1), ((1,1)^T, 0)\}$

Using linear regression with Mean Squared Error (MSE)

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(t_n - y(\mathbf{x}_n))^2$$

We can treat this problem as a regression problem and use a mean squared error loss function. We have chosen this loss function to simplify the math for this example as much as possible. In practical applications, MSE is usually not an appropriate cost function for modeling binary data

with $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$

Optimal solution:

We can minimize J(θ) in closed form with respect to w and b using the normal equations. After solving the normal equations

$$\mathbf{w} = 0 \text{ and } w_0 = \frac{1}{2}, \text{ hence } y = 0.5 \text{ everywhere!}$$

Reason: No linear separator can explain the non-linear XOR function

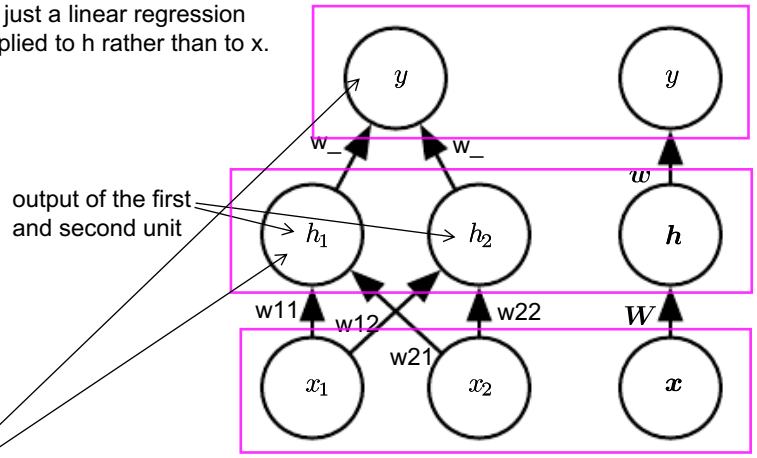
linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution

XOR Example - FNN

these are the same network, in the left is specified each unit in the right no

Specify a two layers network:

The output layer is still just a linear regression model, but now it is applied to h rather than to x.



output layer->layer that generates the output (2 nodes)

hidden layer (2 nodes)

input layer->node that are in the dataset (2 nodes)

activation function->in the same layer is always the same but in general is different from that used in the output layer (in this case we use ReLU)

W describes the mapping from x to h, and a vector w describes the mapping from h to y.

XOR Example - FNN

Hidden units:

biases

weights of a linear transformation

w10
w20

$$\mathbf{h} = g(\mathbf{W}^T\mathbf{x} + \mathbf{c})$$

w11 w12 x1
w21 w22 x2

with $g(\alpha) = \max(0, \alpha)$

Output:

$$y = \mathbf{w}^T\mathbf{h} + b$$

Full model:

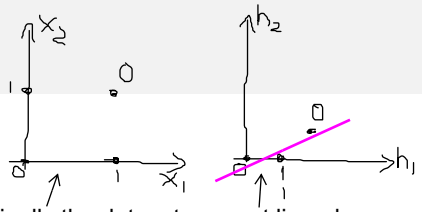
$$y(\mathbf{x}) = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{w}^T \max(0, \mathbf{W}^T\mathbf{x} + \mathbf{c}) + b$$

with $\boldsymbol{\theta} = \langle \mathbf{W}, \mathbf{c}, \mathbf{w}, b \rangle$

Note: non-linear model in $\boldsymbol{\theta}$

XOR Example - FNN

$$\bar{h} = \mathrm{ReLU}(\bar{x}W + \bar{c}) = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \end{pmatrix}$$



originally the dataset was not linearly separable
now it is transformed in a linearly separable set

Model:

$$y(\mathbf{x}) = f(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{w}^T \max(0, \mathbf{W}^T \mathbf{x} + \mathbf{c}) + b$$

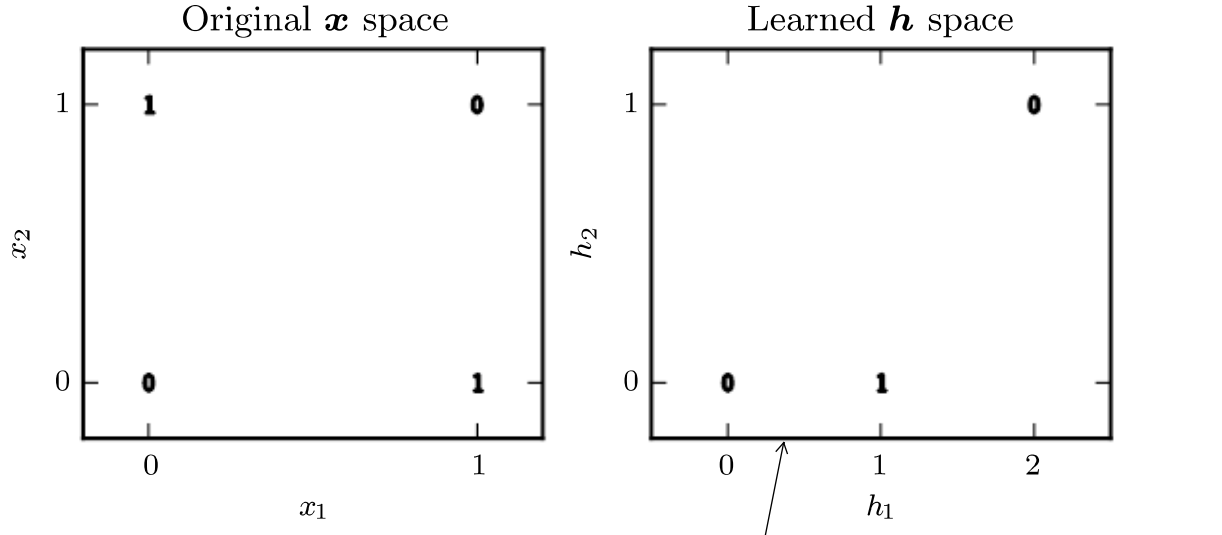
Mean Squared Error (MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(\mathbf{x}_n))^2$$

Solution:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

XOR Example - FNN



In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem.

Architecture design

Overall structure of the network

How many hidden layers? **Depth**

How many units in each layer? **Width**

Which kind of units? **Activation functions** it is very important in the output layer

Which kind of cost function? **Loss function**

Architecture design

How many hidden layers? **Depth**

Universal approximation theorem: a FFN with a linear output layer and at least one hidden layer with any "squashing" activation function (e.g., sigmoid) can approximate any Borel measurable function with any desired amount of error, provided that enough hidden units are used.

It works also for other activation functions (e.g., ReLU)

Architecture design

How many units in each layer? **Width**

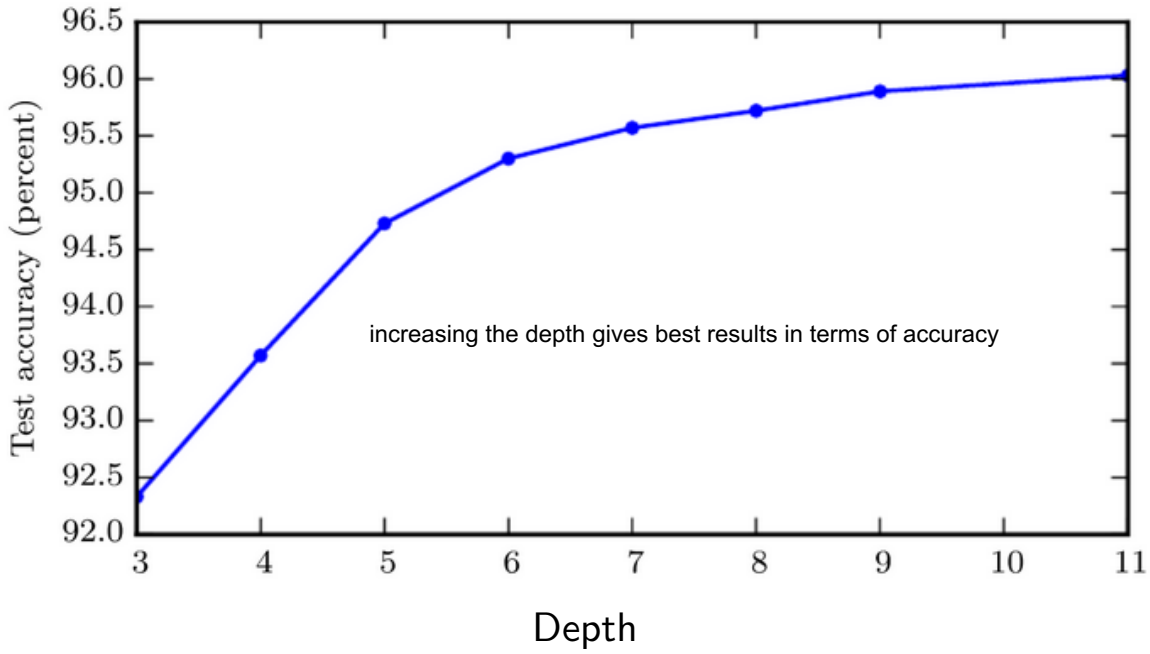
Universal approximation theorem does not say how many units.

In general it is exponential in the size of the input.

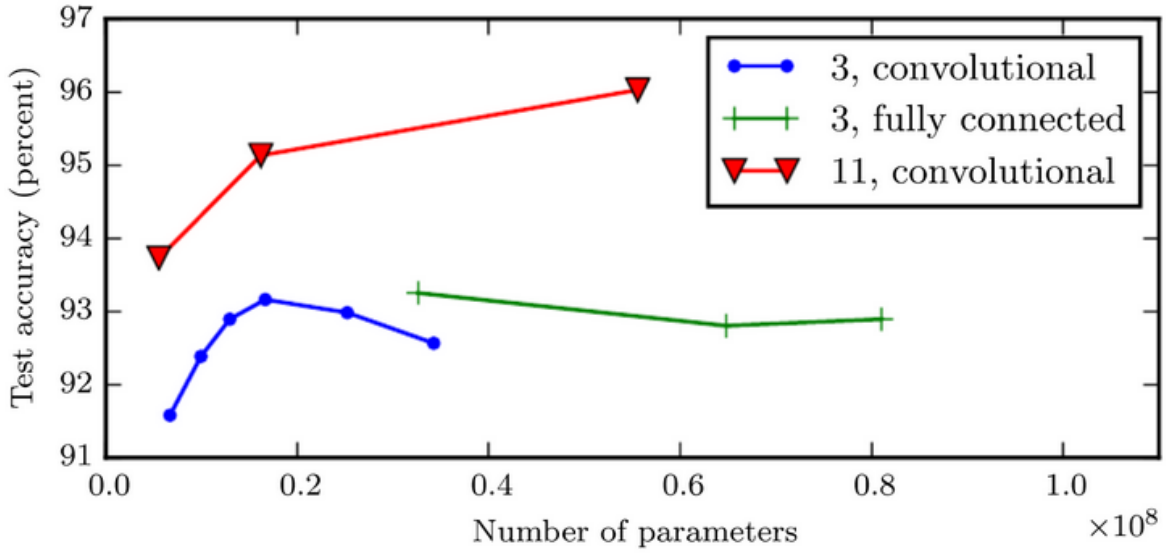
In theory, a short and very (ampia) wide network can approximate any function.

In practice, a deep and narrow network is easier to train and provides better results in generalization.

Architecture design

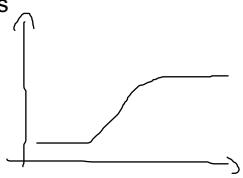


Architecture design

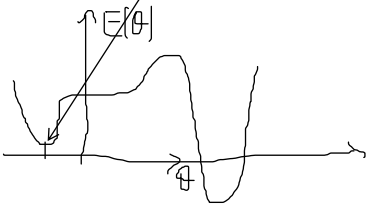


Architecture design

The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become nonconvex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.



thistype of function has a big problem on the flat region because we have a problem when we compute the gradient, there is also a problem with saturation of the gradient (so when it is close to zero)

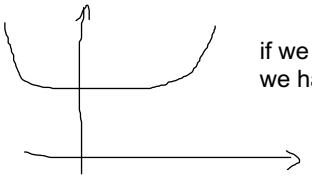


Which kind of units? **Activation functions**

Which kind of cost function? **Loss function**

Gradient-based learning remarks

- Unit saturation can hinder learning
- When units saturate gradient becomes very small
- Suitable cost functions and unit nonlinearities help to avoid saturation



if we have a function of this type we have almost the optimal

For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values.

to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model.

Cost function

Model implicitly defines a conditional distribution $p(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta})$

Cost function: Maximum likelihood principle (**cross-entropy**)

This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution.

$$J(\boldsymbol{\theta}) = E_{\mathbf{x},\mathbf{t}\sim\mathcal{D}} \left[-\ln(p(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta})) \right]$$

the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.

Example:

Assuming additive Gaussian noise we have

$$p(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{t}|f(\mathbf{x}; \boldsymbol{\theta}), \beta^{-1} I)$$

we want to minimize according totheta but there are many components that not depends on theta, so I can not consider these

and hence

$$J(\boldsymbol{\theta}) = E_{\mathbf{x},\mathbf{t}\sim\mathcal{D}} \left[\frac{1}{2} ||\mathbf{t} - f(\mathbf{x}; \boldsymbol{\theta})||^2 \right]$$

full transformation of the full network

Maximum likelihood estimation with Gaussian noise corresponds to mean squared error minimization.

Output units activation functions

like in SLIDE 11

Let $\mathbf{h} = f(\mathbf{x}; \boldsymbol{\theta}^{(n-1)})$ the output of the hidden layers,

which output model $y = f^{(n)}(\mathbf{h}; \boldsymbol{\theta}^{(n)})$?

function that control the output layer

which cost function $J(\boldsymbol{\theta})$?

Choice of network output units and cost function are related.

- Regression
- Binary classification
- Multi-classes classification

depending on the kind of problem, we can hhave these cases

Output units activation functions

Regression it's convenient to consider a linear model as last layer, in which the function is an identity function

Linear units: Identity activation function

$$y = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

Use a Gaussian distribution noise model

$$p(t|\mathbf{x}) = \mathcal{N}(t|y, \beta^{-1})$$

Cost function: maximum likelihood (cross-entropy) that is equivalent to minimizing **mean squared error**.

it's an important combnation becuase guaranties that the gradient not saturate

Note: linear units do not saturate

Output units activation functions

Many tasks require predicting the value of a binary variable y. Classification problems with two classes can be cast in this form.

Binary classification we assume to encode the 2 classes with 0 and 1

Sigmoid units: Sigmoid activation function $y = \sigma(\mathbf{w}^T \mathbf{h} + b)$

it can be seen as the posterior probability of one of the class (that identificaed by 1)

The likelihood corresponds to a Bernoulli distribution

$$J(\boldsymbol{\theta}) = E_{\mathbf{x},t\sim\mathcal{D}} \left[-\ln p(t|\mathbf{x}) \right]$$

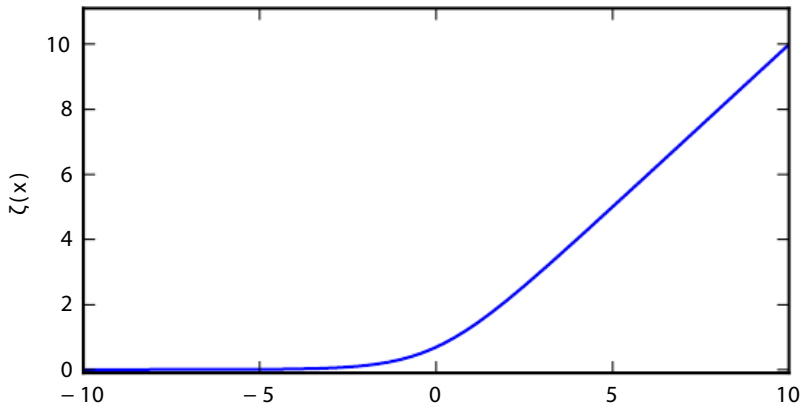
linear combination coming from the previous layer of the network

$$\begin{aligned} -\ln p(t|\mathbf{x}) &= -\ln \sigma(\alpha)^t (1 - \sigma(\alpha))^{1-t} \\ &= -\ln \sigma((2t-1)\alpha) \\ &= \texttt{softplus}((1-2t)\alpha) \end{aligned}$$

with $\alpha = \mathbf{w}^T \mathbf{h} + b$.

Note: Unit saturates only when it gives the correct answer.

Output units activation functions



The $\mathrm{softplus}$ function

Output units activation functions

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function, which was used to represent a probability distribution over a binary variable.

Multi-class classification More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

Softmax units: Softmax activation function

$$y_i = \mathtt{softmax}(\alpha^{(i)}) = \frac{\exp(\alpha^{(i)})}{\sum_j \exp(\alpha_j)}$$

if we have K classes it will be a vector with K components

Likelihood corresponds to a Multinomial distribution

$$J_i(\boldsymbol{\theta}) = E_{\mathbf{x},\mathbf{t}\sim\mathcal{D}}\left[-\ln \mathtt{softmax}(\alpha^{(i)})\right]$$

with $\alpha^{(i)} = \mathbf{w}_i^T\mathbf{h} + b_i$.

Note: Unit saturates only when there are minimal errors. so when we are close to solution

Output units activation functions

Summary

there are no proves that these are the best choices, they are empirical evidence

Regression: **linear** output unit, **mean squared error** loss function

Binary classification: **sigmoid** output unit, **binary cross-entropy**

Multi-class classification: **softmax** output unit, **categorical cross-entropy**

Note: on-going research on different loss functions.

Hidden units activation functions

all the layers except the last

Many choices, some intuitions, no theoretical principles.
Predicting which activation function will work best is usually impossible.

Rectified Linear Units (ReLU):

$$g(\alpha) = \max(0, \alpha).$$

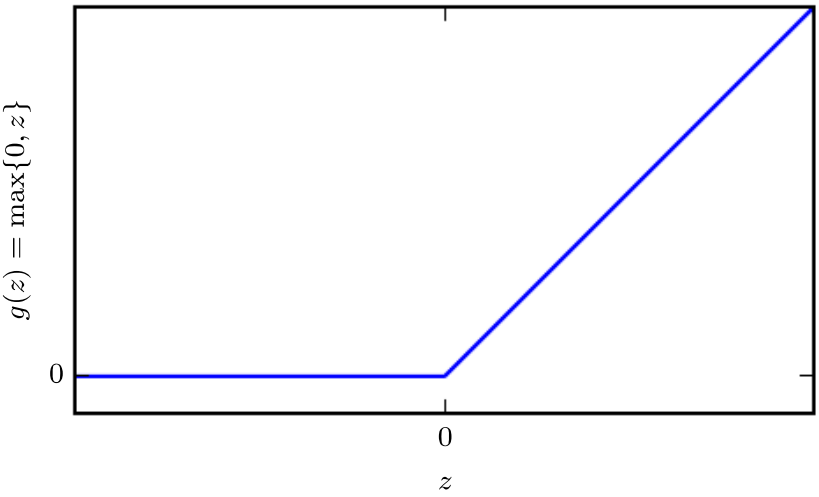
have no flat region in the positive part

- Easy to optimize - because similar to linear units
- Not differentiable at 0 - does not cause problems in practice

at Alpha=0

N.B: other generalization of ReLU are leaky ReLU, parametric ReLU , or PReLU

Hidden unit activation functions



Hidden unit activation functions

Sigmod and hyperbolic tangent:

$$g(\alpha) = \sigma(\alpha)$$

they can saturate

and

$$g(\alpha) = \tanh(\alpha)$$

Closely related as $\tanh(\alpha) = 2\sigma(2\alpha) - 1$.

Remarks: When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid It resembles the identity function more closely, in the sense that tanh(0)=0 while sigma(0)=1/2

- No logarithm at the output, the units saturate easily.
- Gradient based learning is very slow.
- Hyperbolic tangent gives larger gradients with respect to the sigmoid.
- Useful in other contexts (e.g., recurrent networks, autoencoders).

Activation functions overview

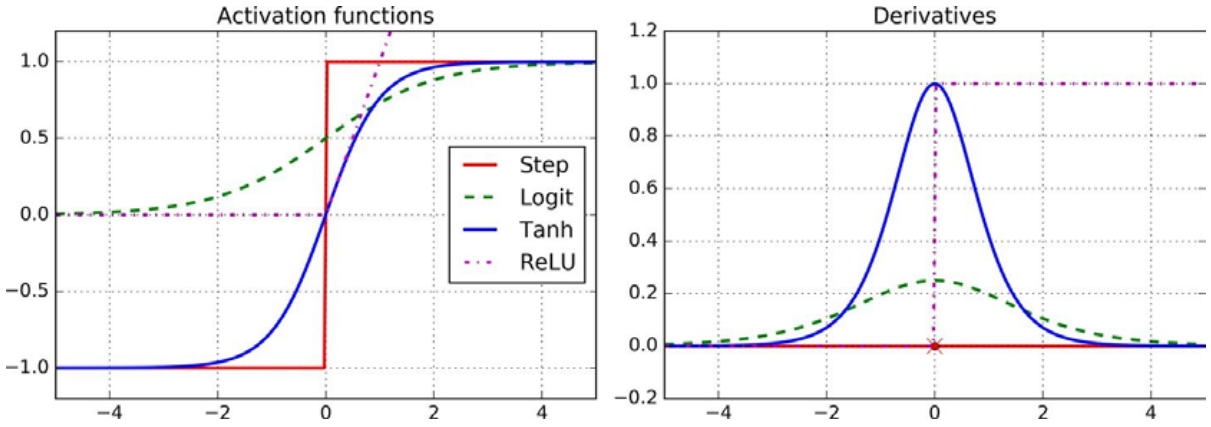


Image from Geron A. "Hands-On Machine Learning with Scikit-Learn and TensorFlow", O'Reilly 2017

Gradient Computation

The input x provides the initial information that then propagates up to the hidden units at each layer and finally produces y^. This is called forward propagation.During training, forward propagation can continue onward until it produces a scalar cost J(θ).The back-propagation algorithm, often simply called backprop, allows the information from the cost to then flow backward through the network in order to compute the gradient. (permette alle informazioni del costo di fluire all'indietro attraverso la rete per calcolare il gradiente.)

Information flows forward through the network when computing network output y from input $\mathbf{x}$

it has to be easily computable

To train the network we need to compute the gradients with respect to the network parameters $\boldsymbol{\theta}$

The **back-propagation** or **backprop** algorithm is used to propagate gradient computation from the cost through the whole network

the idea is to split the computation of the gradient in 2 parts

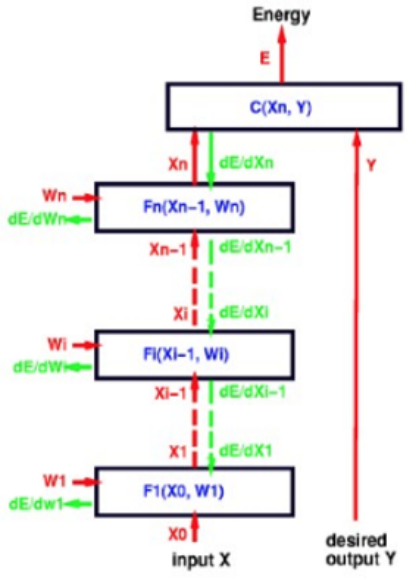


Image by Y. LeCun

back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.

Gradient Computation

Goal: Compute the gradient of the cost function w.r.t. the parameters

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Analytic computation of the gradient is straightforward

- simple application of the chain rule
- numerical evaluation can be expensive

Back-propagation is *simple* and *efficient*.

Remarks:

it's just estimating the gradient

- back-propagation is only used to compute the gradients
- back-propagation is **not** a training algorithm
- back-propagation is **not** specific to FNNs

Chain rule

Why compute a gradient in the NN is simple

Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

is used to compute the derivatives of functions formed by composing other functions whose derivatives are known.

Let: $y = g(x)$ and $z = f(g(x)) = f(y)$

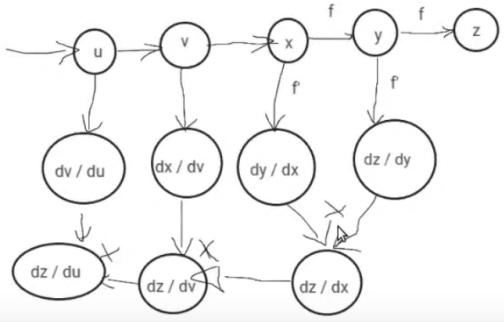
Applying the chain rule we have:

see model SLIDE 7->the derivative of this can be done with the partial derivatives

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

For vector functions, $g : \mathbb{R}^m \mapsto \mathbb{R}^n$ and $f : \mathbb{R}^n \mapsto \mathbb{R}$ we have:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i},$$



equivalently in vector notation:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z,$$

with $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ the $n \times m$ Jacobian matrix of g.

Back-propagation algorithm

algorithm for computing the gradient of the error/loss function

Forward step

Require: Network depth l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$ weight matrices
Require: $\mathbf{b}^{(i)}, i \in \{1, \ldots, l\}$ bias parameters
Require: $\mathbf{x}$ input value
Require: $\mathbf{t}$ target value

$h^{(0)} = \mathbf{x}$
for $k = 1, \ldots, l$ **do**
$\boldsymbol{\alpha}^{(k)} = \mathbf{b}^{(k)} + W^{(k)}\mathbf{h}^{(k-1)}$
$\mathbf{h}^{(k)} = f(\boldsymbol{\alpha}^{(k)})$
end for
$\mathbf{y} = \mathbf{h}^{(l)}$ prediction of the network for a given input x with the parameters above
$J = L(\mathbf{t}, \mathbf{y})$ error computing the loss function

it is composed by 2 parts:
FORWARD STEP: given the params of the networks, input, target value and the loss function, we compute the loss function applied to the output of the network (y) and the target value (t)
BACKWARD STEP: goes back from this error and compute the derivative of this error with respect to all the params of the network by going back from the output layer to the input layer

Back-propagation algorithm

it shows how to compute gradients of J with respect to parameters W and b. For simplicity, this demonstration uses only a single input example x. Practical applications should use a minibatch.

we start from the last layer and we compute the gradient from the next layer to the previous one

Backward step

After the forward computation, compute the gradient on the output layer

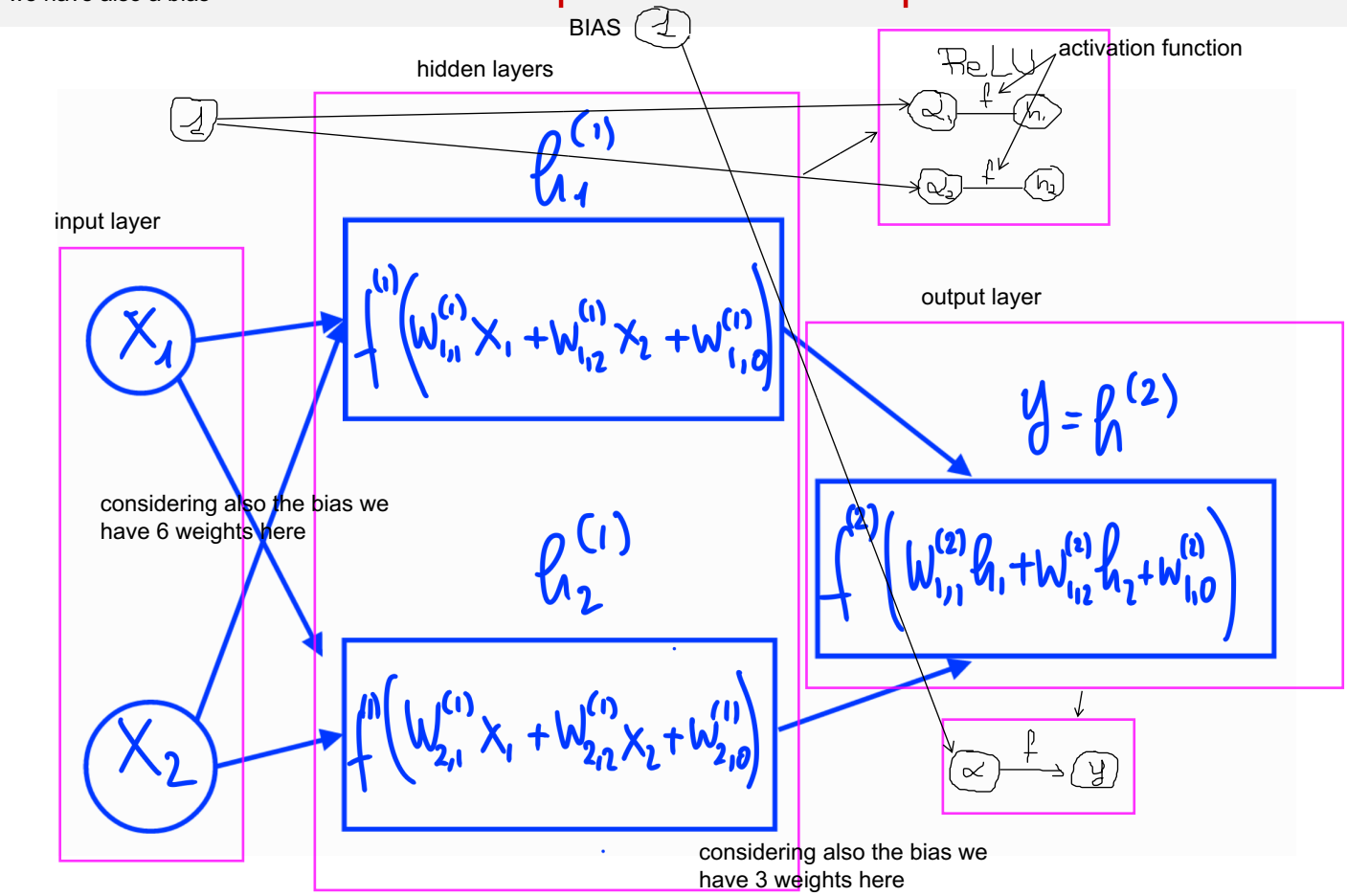
$\mathbf{g} \leftarrow \nabla_{\mathbf{y}} J = \nabla_{\mathbf{y}} L(\mathbf{t}, \mathbf{y})$
for $k = l, l-1, \ldots, 1$ **do** ← we procede backward
Propagate gradients to the pre-nonlinearity activations:
$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(k)}} J = \mathbf{g} \odot f'(\boldsymbol{\alpha}^{(k)})$ {$\odot$ denotes elementwise product}
$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g}$ ← ~~gradient of the previous layer~~ Compute gradients on weights and biases
$\nabla_{W^{(k)}} J = \mathbf{g}(\mathbf{h}^{(k-1)})^T$
Propagate gradients to the next lower-level hidden layer:
$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = (W^{(k)})^T \mathbf{g}$
end for

Back-propagation algorithm

Remarks:

- The previous version of backprop is specific for fully connected MLPs
- More general versions for acyclic graphs exist
- Dynamic programming is used to avoid doing the same computations multiple times (optimizations)
- Gradients can be computed either in symbolic or numerical form

Example of BackProp



Example of BackProp

the forward step is simply to apply these formulas

$$\alpha_i^{(1)} = w_{i,0}^{(1)} + w_{i,1}^{(1)} x_1 + w_{i,2}^{(1)} x_2 \quad i = 1, 2$$

$$h_i^{(1)} = f^{(1)}(\alpha_i^{(1)}) = \texttt{ReLU}(\alpha_i^{(1)}) \quad i = 1, 2$$

$$\alpha^{(2)} = w_{1,0}^{(2)} + w_{1,1}^{(1)} h_1^{(1)} + w_{1,2}^{(1)} h_2^{(1)}$$

$$h^{(2)} = f^{(2)}(\alpha^{(2)}) = \alpha^{(2)}$$

$$y = h^{(2)}$$

Loss function MSE

$$L(t, y) = \frac{1}{2}(t - y)^2$$

$$\boldsymbol{\theta} = \langle w_{1,0}^{(1)}, w_{1,1}^{(1)}, w_{1,2}^{(1)}, w_{2,0}^{(1)}, w_{2,1}^{(1)}, w_{2,2}^{(1)}, w_{1,0}^{(2)}, w_{1,1}^{(2)}, w_{1,2}^{(2)} \rangle$$

Example of BackProp

Forward step

Given $x_1, x_2, w_{i,j}^{(k)}, t$

compute $\alpha_1^{(1)}, \alpha_2^{(1)}, \alpha^{(2)}, h_1^{(1)}, h_2^{(1)}, h^{(2)}, y, J = L(t, y)$

Backward step starting from the output layer

Given $x_1, x_2, w_{i,j}^{(k)}, t, \alpha_1^{(1)}, \alpha_2^{(1)}, \alpha^{(2)}, h_1^{(1)}, h_2^{(1)}, h^{(2)}, y, J = L(t, y)$

compute $\frac{\partial J}{\partial w_{i,j}^{(k)}}$ derivative of the error with respect to the parameters of the network

Example of BackProp

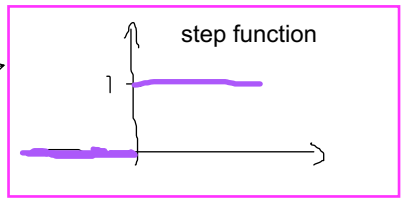
Gradient computation

$$\frac{\partial J(\boldsymbol{\theta})}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(t-y)^2 = y - t$$ derivative computed with respect to the output

$$\frac{\partial J(\boldsymbol{\theta})}{\partial w_{i,j}^{(2)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial y}\frac{\partial y}{\partial w_{i,j}^{(2)}} \quad \text{with} \quad \frac{\partial y}{\partial w_{1,0}^{(2)}} = 1, \quad \frac{\partial y}{\partial w_{1,1}^{(2)}} = h_1^{(1)} \quad \frac{\partial y}{\partial w_{1,2}^{(2)}} = h_2^{(1)}$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial h_i^{(1)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial y}\frac{\partial y}{\partial h_i^{(1)}} \quad \text{with} \quad \frac{\partial y}{\partial h_1^{(1)}} = w_{1,1}^{(2)} \quad \frac{\partial y}{\partial h_2^{(1)}} = w_{1,2}^{(2)}$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \alpha_i^{(1)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial h_i^{(1)}}\frac{\partial h_i^{(1)}}{\partial \alpha_i^{(1)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial h_i}\,\mathtt{step}(\alpha_i^{(1)})$$



$$\frac{\partial J(\boldsymbol{\theta})}{\partial w_{i,j}^{(1)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial \alpha_i^{(1)}}\frac{\partial \alpha_i^{(1)}}{\partial w_{i,j}^{(1)}} \quad \text{with} \quad \frac{\partial \alpha_i^{(1)}}{\partial w_{i,0}^{(1)}} = 1, \quad \frac{\partial \alpha_i^{(1)}}{\partial w_{i,1}^{(1)}} = x_1 \quad \frac{\partial y}{\partial w_{i,2}^{(1)}} = x_2$$

Example of BackProp (compact notation)

Gradient computation (in vector notation)

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} w_{1,2}^{(1)} \\ w_{2,1}^{(1)} w_{2,2}^{(1)} \end{bmatrix}, \ \mathbf{b}^{(1)} = \begin{bmatrix} w_{1,0}^{(1)} \\ w_{2,0}^{(1)} \end{bmatrix}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} w_{1,2}^{(2)} \end{bmatrix}, \ \mathbf{b}^{(2)} = \begin{bmatrix} w_{1,0}^{(2)} \end{bmatrix}$$

$$f^{(1)}(z) = \mathtt{ReLU}(z), \ f^{(2)}(z) = z$$

Example of BackProp (compact notation)

$$\mathbf{h}^{(0)} = \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\boldsymbol{\alpha}^{(1)} = \begin{bmatrix} \alpha_1^{(1)} \\ \alpha_2^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}$$

$$\mathbf{h}^{(1)} = \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \end{bmatrix} = f^{(1)}(\boldsymbol{\alpha}^{(1)}) = \begin{bmatrix} f^{(1)}(\alpha_1^{(1)}) \\ f^{(1)}(\alpha_2^{(2)}) \end{bmatrix} = \begin{bmatrix} \texttt{ReLu}(\alpha_1^{(1)}) \\ \texttt{ReLu}(\alpha_2^{(2)}) \end{bmatrix}$$

$$\boldsymbol{\alpha}^{(2)} = \left[\alpha^{(2)}\right] = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{h}^{(2)} = \left[h^{(2)}\right] = f^{(2)}(\boldsymbol{\alpha}^{(2)}) = \left[f^{(2)}(\alpha^{(2)})\right] = \left[\alpha^{(2)}\right] = \boldsymbol{\alpha}^{(2)}$$

$$\mathbf{y} = \mathbf{h}^{(2)} = \boldsymbol{\alpha}^{(2)}$$

Example of BackProp (compact notation)

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(2)}} J = \nabla_y J = \nabla_y \frac{1}{2}(t-y)^2 = \frac{\partial(\frac{1}{2}(t-y)^2)}{\partial y} = y - t$$

$$\mathbf{g} \leftarrow \nabla_{\alpha^{(2)}} J = \mathbf{g} \odot f^{(2)\prime}(\alpha^{(2)}) = \mathbf{g} \odot \frac{\partial \alpha^{(2)} - t}{\partial \alpha^{(2)}} = \mathbf{g} \odot 1 = \mathbf{g}$$

$$\nabla_{\mathbf{b}^{(2)}} J \leftarrow \frac{\partial J}{\partial w_{1,0}^{(2)}} = \mathbf{g}$$

$$\nabla_{\mathbf{W}^{(2)}} J \leftarrow \begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^{(2)}} & \frac{\partial J}{\partial w_{1,2}^{(2)}} \end{bmatrix} = \mathbf{g} \cdot (\mathbf{h}^{(1)})^T$$

Example of BackProp (compact notation)

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(1)}} J = \begin{bmatrix} \frac{\partial J}{\partial h_1^{(1)}} \\ \frac{\partial J}{\partial h_2^{(1)}} \end{bmatrix} = (\mathbf{W}^{(2)})^T \cdot \mathbf{g}$$

$$\mathbf{g} \leftarrow \nabla_{\boldsymbol{\alpha}^{(1)}} J = \mathbf{g} \odot f^{(1)\prime}(\boldsymbol{\alpha}^{(1)}) = \mathbf{g} \odot \begin{bmatrix} \frac{\partial \texttt{ReLU}(\alpha_1^{(1)})}{\partial \alpha_1^{(1)}} \\ \frac{\partial \texttt{ReLU}(\alpha_2^{(1)})}{\partial \alpha_2^{(1)}} \end{bmatrix} = \mathbf{g} \odot \begin{bmatrix} \texttt{step}(\alpha_1^{(1)}) \\ \texttt{step}(\alpha_2^{(1)}) \end{bmatrix}$$

$$\nabla_{\mathbf{b}^{(1)}} J \leftarrow \begin{bmatrix} \frac{\partial J}{\partial w_{1,0}^{(1)}} \\ \frac{\partial J}{\partial w_{2,0}^{(2)}} \end{bmatrix} = \mathbf{g}$$

$$\nabla_{\mathbf{W}^{(1)}} J \leftarrow \begin{bmatrix} \frac{\partial J}{\partial w_{1,1}^{(1)}} & \frac{\partial J}{\partial w_{1,2}^{(1)}} \\ \frac{\partial J}{\partial w_{2,1}^{(1)}} & \frac{\partial J}{\partial w_{2,2}^{(1)}} \end{bmatrix} = \mathbf{g} \cdot (\mathbf{h}^{(1)})^T$$

Learning algorithms

now that we have a method for estimating the gradient, we use it to find a solution of our Optimization problem that minimize the error function or the loss function

- Stochastic Gradient Descent (SGD)
- SGD with momentum
- Algorithms with adaptive learning rates

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep Learning in particular

Stochastic Gradient Descent

A crucial parameter for the SGD algorithm is the learning rate. it is necessary to gradually decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum.

Require: Learning rate $\eta \geq 0$
Require: Initial values of $\boldsymbol{\theta}^{(1)}$
$k \leftarrow 1$ the stop criterion can be the number of iterations or a specific value of a parameter
while stopping criterion not met **do**
  Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of m examples from the dataset D the best approach to use
  Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$ sum of all the loss on all the samples in the dataset
  Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - \eta \mathbf{g}$
  $k \leftarrow k + 1$
end while

Observe: $\nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{t})$ obtained with backprop

Stochastic Gradient Descent

far from the optimal it is convenient to have a large Learning rate and near a smaller one. the problem is that I don't know where I'm at the beginning

η usually changes according to some rule through the iterations

Until iteration τ $(k \leq \tau)$: it tau is too large I can reach diverging situations

$$\eta^{(k)} = \left(1 - \frac{k}{\tau}\right) \eta^{(k)} + \frac{k}{\tau} \eta^{(\tau)}$$

After iteration τ $(k > \tau)$:

we move quickly toward the local minimum

$$\eta^{(k)} = \eta^{(\tau)}$$

SGD with momentum

While stochastic gradient descent remains a popular optimization strategy, learning with it can sometimes be slow. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients

Momentum can accelerate learning The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction

Motivation: Stochastic gradient can largely vary through the iterations

2 params

Require: Learning rate $\eta \geq 0$

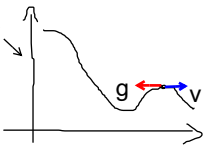
Require: Momentum $\mu \geq 0$ how much we want to move the solution even when the gradient is zero

Require: Initial values of $\boldsymbol{\theta}^{(1)}$

$k \leftarrow 1$

$\mathbf{v}^{(1)} \leftarrow 0$

theoretical situation that is impossible in practice. in this case g and v are both equal to zero, and that is an instable point of equilibrium, so we can move from that point either toward g or toeard v



while stopping criterion not met **do**

Sample a subset (minibatch) $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ of m examples from the dataset D

Compute gradient estimate: $\mathbf{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}^{(k)}), \mathbf{t}^{(i)})$

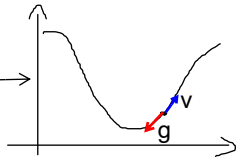
velocity

Compute velocity: $\mathbf{v}^{(k+1)} \leftarrow \mu \mathbf{v}^{(k)} - \eta \mathbf{g}$, with $\mu \in [0, 1)$

Apply update: $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} + \mathbf{v}^{(k+1)}$

$k \leftarrow k + 1$

we stop this climbing when v and g are the same



end while

SGD with momentum

Momentum μ might also increase according to some rule through the iterations.

SGD with Nesterov momentum

variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method

Nesterov momentum

The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum, the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum.

Momentum is applied before computing the gradient

$$\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}^{(k)} + \mu \mathbf{v}^{(k)}$$

$$\mathbf{g} = \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \mathbf{t}^{(i)})$$

Sometimes it improves convergence rate.

Algorithms with adaptive learning rates

the parameters of the SGD, are find automatically, and not fixed. For this aim we can use some optimizers

Based on analysis of the gradient of the loss function it is possible to determine, at any step of the algorithm, whether the learning rate should be increased or decreased.

Some examples:

- AdaGrad
- RMSProp
- Adam

(see Deep Learning Book, Section 8.5 for details)

Which optimization algorithm should I choose?

Empirical approach.

Regularization

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs.

As with other ML approaches, regularization is an important feature to reduce overfitting (generalization error).

For FNN, we have several options (can be applied together):

to make that the training error and the test error in the scope goes in parallel

- Parameter norm penalties
- Dataset augmentation
- Early stopping
- Parameter sharing
- Dropout

Parameter norm penalties

Add a regularization term E_{reg} to the cost function

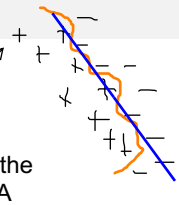
$$E_{\text{reg}}(\boldsymbol{\theta}) = \sum_j |\theta_j|^q.$$

Resulting cost function:

$$\bar{J}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda E_{\text{reg}}(\boldsymbol{\theta}).$$

Dataset augmentation

very often one problem of overfitting is given by the ability of the network to perfectly shape our dataset



if we have a model that perfectly shape the dataset (orange), sometimes is more convenient have a linear separation (purple line). so if we add random samples the system cannot more perfectly shape the dataset (DATA AUGMENTATION)

Generate additional data and include them in the dataset.

- Data transformations (e.g., image rotation, scaling, varying illumination conditions, ...)
- Adding noise

In Exercise 7, noisy XOR converges faster than XOR.

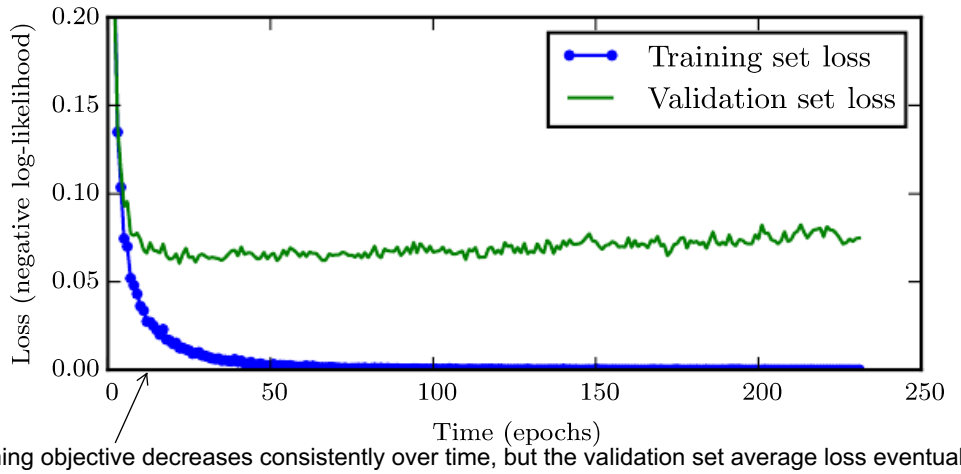
Early stopping

It is probably the most commonly used form of regularization in deep learning. Its popularity is due to both its effectiveness and its simplicity

Early stopping:

Stop iterations early to avoid overfitting to the training set of data

but we can't know if after we have done the early stopping, the training error will decrease with also the validation error



Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

When to stop? Use cross-validation to determine best values.

Parameter sharing

we need many parameters in each layer to have a working network. the XOR with only 9 parameters works only for that XOR problem

Parameter sharing: constraint on having subsets of model parameters to be equal.

Advantages also in memory storage (only the unique subset of parameters need to be stored).

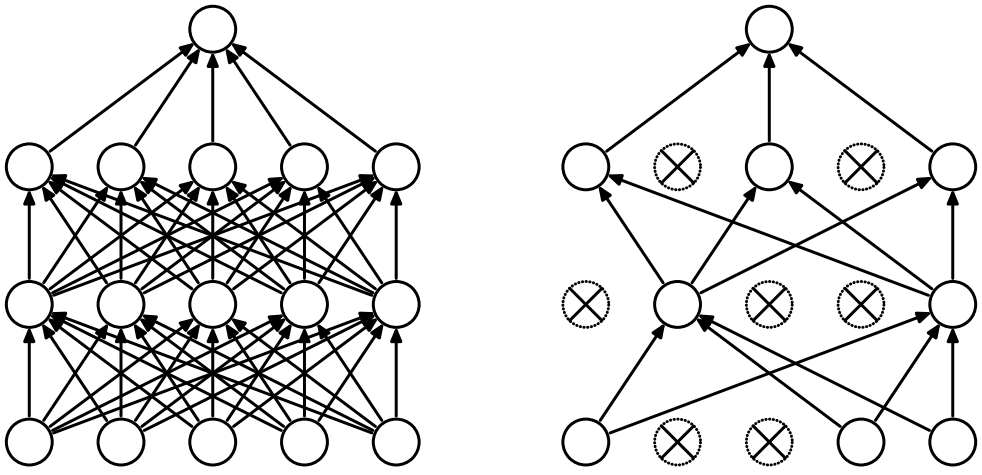
In Convolutional Neural Networks (CNNs) parameter sharing allows for invariance to translation.

we can constraint some parameters to be the same. for ex. of the first layer we have 100 connection but we add a constraint so that the number of training of all parameters is just 20. so we have 100 connections but only 20 trainable parameters. We apply this to all connections

Dropout

randomly not consider some connections

Dropout: Randomly remove network units with some probability α



(a) Standard Neural Net (b) After applying dropout.

Image from Srivastava *et al.*. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Dropout

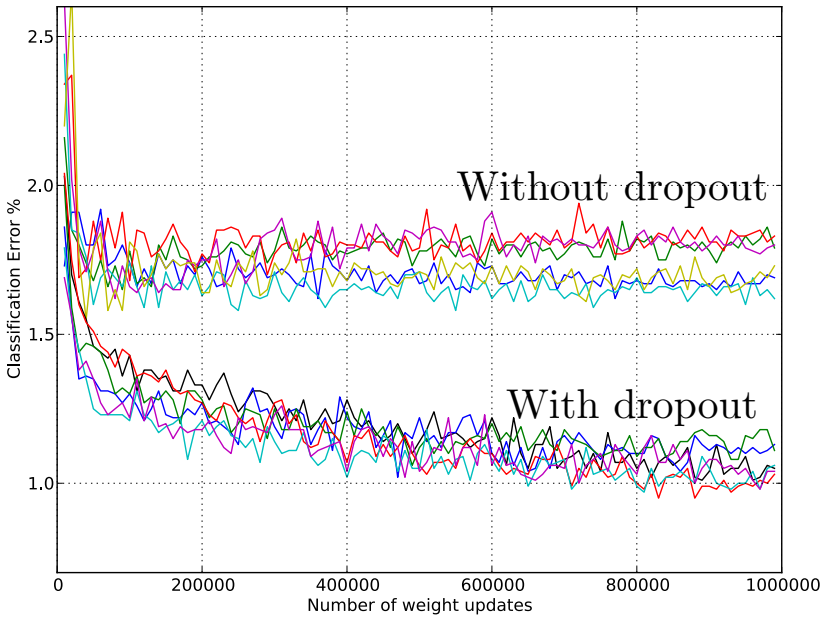


Image from Srivastava *et al.*. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting"

Summary

Feedforward neural networks (FNNs)

- parametric models with many combination of simple functions
- can effectively approximate any function (no need to guess kernel models)
- must be carefully designed (empirically)
- efficient ways to optimize the loss function
- deep architectures perform better
- optimization performance can be improved with momentum and adaptive learning rate
- generalization error can be reduced with regularization