DMIF, University of Udine

An Introduction to Neural Networks

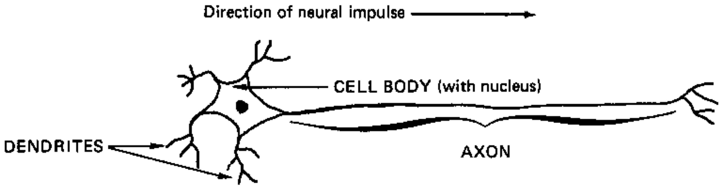
Andrea Brunello, Ph.D. Student
andrea.brunello@uniud.it

November 22, 2018

Brain neurons

A brain neuron can be seen as an organic switch:

- it receives multiple signals through *synapses*, located on the *dendrites*
- when the received signals are strong enough, the neuron is *activated* and emits a signal through the *axon*, which might then activate other neurons



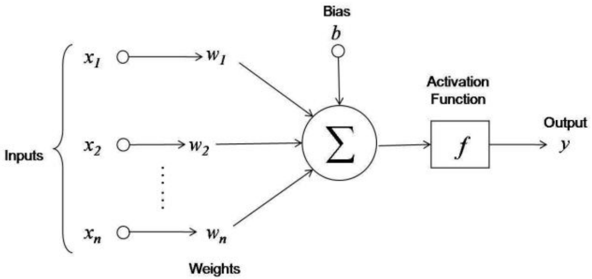
Artificial neurons

An artificial neuron can be seen as an *abstraction* of a brain neuron:

- it computes a *weighted sum* of the inputs:

$$b + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

- the result is then passed through a nonlinear *activation function*



Neural network, single output

The neurons are organized in layers ($\rightsquigarrow$ abstraction hierarchies).
Input: $x_1, \ldots, x_n$ attribute values of the specific instance

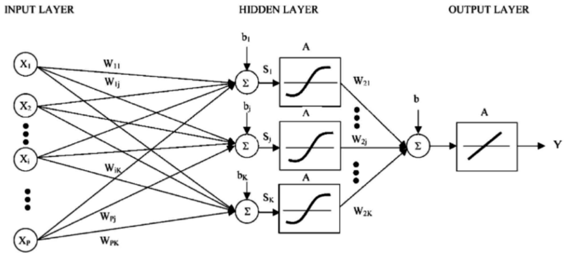
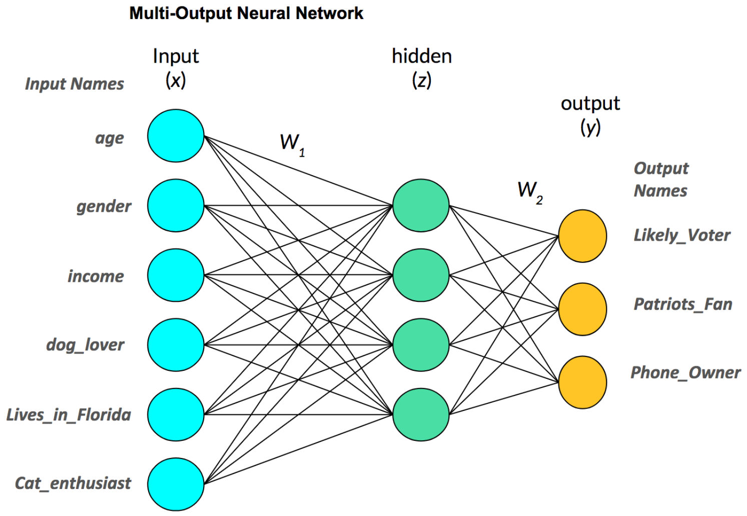


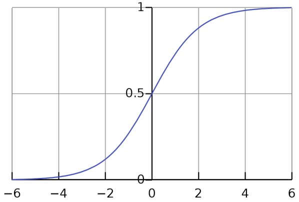
Figure: A 2-layer, fully-connected, feed-forward neural network.

Neural network, multiple outputs



Activation function

- In classification tasks, to simulate a biological neuron, the activation func. should have a "switch on" characteristic
- *Sigmoid* was commonly used: $S(x) = 1/(1 + e^{-x})$
- In this case, a single neuron is similar to a logistic classifier
- *Vanishing gradient* problem: the function saturates, i.e., in some regions, even a large change in the input will produce a small change in the output (gradient is small)

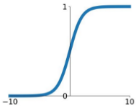


Other activation functions

The choice of the most appropriate activation function depends on the specific task at hand

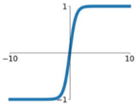
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



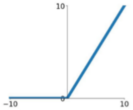
tanh

$\tanh(x)$



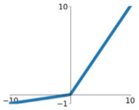
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

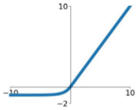


Maxout **(Layer)**

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

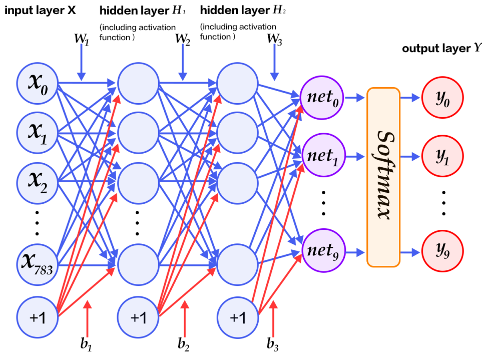


Typically, layers are homogeneous w.r.t the activation function

Output layer and activation function

For the output layer, typically:

- **identity**: regression tasks
- **sigmoid**: multiple classification, e.g., image object recognition
- **softmax (layer)**: classification, it generates probabilities



Properties of activation functions

Among the others:

- **continuously differentiable**: to allow for the training of the network (ReLU?)
- **nonlinear**: then, a two-layer (1 HL) neural network can be proven to be a *universal function approximator*: a FF network with 1 HL can represent any continuous function that maps intervals of $\mathbb{R}$ to intervals of $\mathbb{R}$, but the layer may be infeasibly large and may fail to learn and generalize
- **monotonic**: in such case, the error surface associated with a single-layer model is guaranteed to be convex

Full list: `https://en.wikipedia.org/wiki/Activation_function`

Training of the net

- The weights and the biases in the network are initialized with *small random values*
- Then, they are *iteratively adjusted* considering the output of the net, through the following training loop:
    1. draw a batch of training samples x with corresponding targets y
    2. run the network on x (forward pass) to obtain $\hat{y}$
    3. compute a measure of the mismatch between y and $\hat{y}$
    4. update all weights in the net in a way that slightly reduces the loss on this batch

Training of the net: problem definition

- We want to find the best weights w and biases b so that the output a_i from the network approximates the label $y(x_i)$ for every input $x_1, x_2, \ldots, x_n$
- To quantify how well we are achieving this goal, we define a *cost function*:

$$C(\vec{w}, \vec{b}) = \frac{1}{2n} \sum_{i=1}^{n} (y(x_i) - a_i)^2$$

- The objective is to minimize such cost function (in the biggest neural networks it may depend on billions of variables!)

Training of the net: a simple strategy

How can we adjust the weights based on the result of the cost function on a batch?

- freeze all weights except for the one being considered, say $w_k = 0.5$
- calculate the cost function for $w_k = 0.75$ and $w_k = 0.25$
- update the value of w setting it to the one that reduces the cost most
- repeat for all coefficients in the network (thousands or millions)

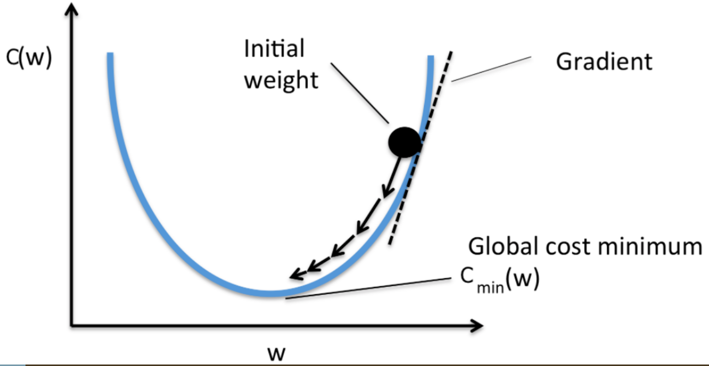
$\rightsquigarrow$ Horribly inefficient!

Fortunately, we can take advantage of the fact that all operations used in the network are *differentiable*

Training of the net: gradient descent

Gradient descent is used to determine how to vary the weights and biases, in order to minimize the cost

- the gradient (i.e., derivative) of the cost function w.r.t. the weights and biases is considered to find a (local) minimum
- single-weight case: $w_{new} = w_{old} - \eta * \nabla C, \ \nabla C = \frac{\partial C}{\partial w}$



Training of the net: gradient descent (2)

In general, the gradient is a vector of partial derivatives. For each weight w^i and bias b^j of the net, we have:

$$w^i_{new} = w^i_{old} - \eta \frac{\partial C}{\partial w^i_{old}}$$
$$b^j_{new} = b^j_{old} - \eta \frac{\partial C}{\partial b^j_{old}}$$

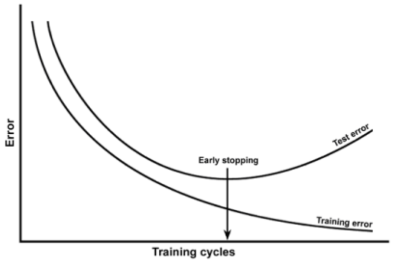
How may we calculate the gradient of the cost function, i.e., the partial derivatives with respect to any weight and bias in the network?

- *Backpropagation* algorithm (chanining rule)

Overfitting

As all other machine learning techniques, also neural networks are prone to overfitting:

- it happens when the model is too complex w.r.t. the amount of training data available
- the performance of the model on validation data peaks after a few epochs, then begins to degrade



Overfitting: possible solutions

How to counter overfitting?

- increase the amount of training data
- decrease the model complexity (number and size of layers)
- reduce the number of training iterations (epochs)
- specific strategies:
  - *regularization*
  - *dropout*

Overfitting: regularization

Idea: a *simple mode* is a model where the distribution of parameter values is “smooth”:

- a possible way to achieve this is by forcing the weights to take only small values
- this typically makes the distribution of weight values more regular (weight regularization)
- simply add to the loss function of the network a cost associated to having large weights
- two main flavours:
  - *L1_regularization*: the cost added is proportional to the absolute value of the weight coefficients
  - *L2_regularization*: the cost added is proportional to the square of the value of the weight coefficients

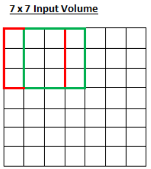
Overfitting: dropout

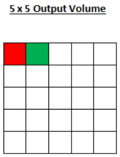
One of the most effective and commonly used regularization techniques for neural networks:

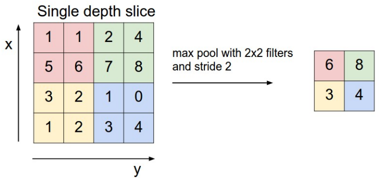
- during *training*, a fraction of output features of the layer (typically, 0.2-0.5) are randomly dropped out (set to zero)
- at *test* time, no units are dropped. Instead, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time
- the core idea is that introducing noise in the output values of a layer can break up patterns that are not significant, which the network would start memorizing if no noise is present

Convolutional Neural Networks

- classical, fully-connected layers learn *global patterns* in their input space
- sometimes, it is useful to learn *local patterns*
- for example, when dealing with image classification, we may extract patterns referred to small 2D patches of the input image
- typical operations: *convolution* + *pooling* (downsampling)



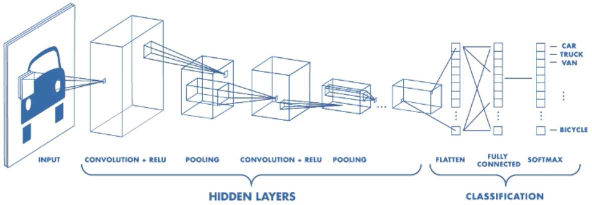




Note: there are also 1D convs, typically applied to seq. data

Convolutional Neural Networks

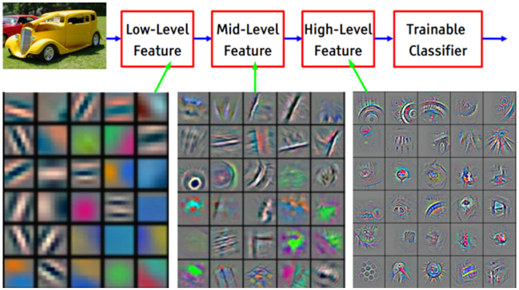
- 2D convolutions typically operate over 3D data (height, width, color depth of an image)
- the output is still 3D, with an arbitrary depth
- intuitively, each output channel in the depth axis works as a *filter*, encoding specific aspects of the input data
- the convolutional layers are then followed by a series of classic, fully-connected layers



Convolutional Neural Networks

CNNs have two interesting properties:

- the patterns they learn are *translation invariant*: a pattern learnt in the lower-right corner of a picture can be recognized everywhere
- they learn *spatial hierarchies* of patterns



Recurrent Neural Networks

All neural networks described so far *have no memory* (FF):

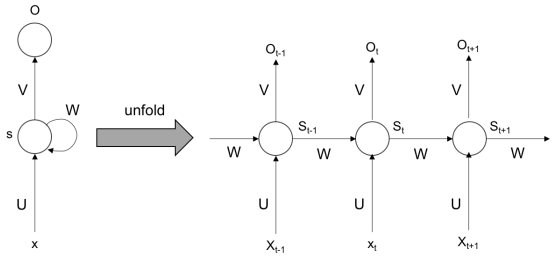
- with them, in order to process a sequence or a temporal series of data points, you have to show the entire sequence to the network at once, i.e., turn it into a single data point

A RNN processes sequences by iterating through the sequence elements:

- it keeps a *state* containing information on what it has seen so far in a sequence
- the state is reset between processing two different, independent sequences

Recurrent Neural Networks

In principle, a RNN consists of a *for loop* that reuses quantities computed during the previous iteration of the loop



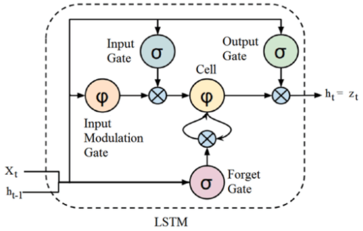
Although it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice such long-term dependecies are impossible to learn, due to the *vanishing gradient problem*

Recurrent Neural Networks

A solution to such a problem is given by specifically designed units, such as:

- Long Short-Term Memory (LSTM)
- Gated Recurrent Unit (GRU)

In short, a LSTM units allows the net to save information for later use, thus preventing older signals from gradually vanishing during processing



Some history: the first models

1943 Neurophysiologist S. McCulloch and mathematician W. Pitts develop the model of the first **artificial neuron**, subsequently implemented by a simple circuit

- input could be 0 or 1, and excitatory (+) or inhibitory (-)
- inputs are summed considering their sign
- if the sum surpassed a certain threshold, output was 1, otherwise it was set to 0
- no learning mechanism

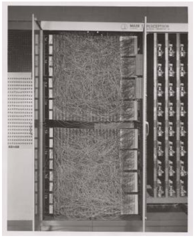
1949 D. Hebb proposes **connectionism** theory

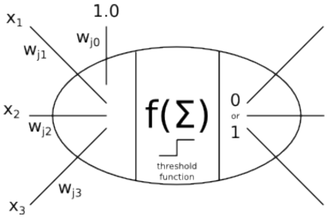
- neural pathways strengthen over each successive use,
- especially between neurons that tend to fire at the same time

Some history: the Perceptron

1957 F. Rosenblatt develops the **Perceptron**, which is the first "practical" artificial neural network

- single-layer network (in fact, a single neuron)
- linear classifier (e.g., can calculate AND and OR)
- weights are self-adjusted during the training phase
- "The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence [. . .] Dr. F. Rosenblatt said Perceptrons might be fired to the planets as mechanical space explorers" (NYT)





Some history: the Golden Age

1959 At Stanford, B. Widrow and M. Hoff develop **ADALINE**

- single-layer artificial neural network with multiple nodes
- each node accepts multiple inputs and generates an output
- improved training approach w.r.t. the Perceptron, that takes into account how much the error changes when each weight is changed (i.e., the derivative)

1962 **MADALINE** is the first neural network successfully applied to a real world problem

- two-layer, fully connected artificial neural network of ADALINE units
- removal of noise in phone lines

Some history: the AI Winter

1969 M. Minsky (founder of the MIT AI Lab) and S. Papert (director of the lab) publish the book **Perceptrons**

- they prove that basic Perceptrons are incapable of processing the XOR circuit
- they further argue that the Perceptron could not be translated into multi-layered neural networks

1970s The **AI Winter**

- neural network research slowdown
- public and private institutions abandon neural network approaches, focusing instead on *expert systems*
- in 1975, P. Werbos discovers the potential of *Backpropagation* in neural networks, and goes unnoticed

Some history: the rebirth

1980 K. Fukushima proposes the **Neocognitron**, which served as the inspiration for convolutional neural networks

1982 **Hopfield network** is one of the first forms of RNN

1986 D. Rumelhart, G. Hinton, and R. Williams publish **Learning representations by back-propagating errors**

- they clearly and concisely state the idea of Backpropagation, which finally becomes widely known
- also, they specifically address the problems discussed by Minsky in Perceptrons

1989 It is mathematically proven that multiple layers allow neural networks to act as **univeral approximators**

1989 AT&T Bell Labs publish the work **Backpropagation Applied to Handwritten Zip Code Recognition**

Some history: some problems emerge

1991 Hochreiter identifies the **vanishing and exploding gradient** problems of backpropagation, that make it difficult to train f.f. deep or recurrent neural networks

1995 S. Thrun investigates the problem of learning a neural network capable of playing chess, getting fare worse results than those of the standard GNU-Chess software: **computers are still too weak**

1995 LeCun et al. find that **Support Vector Machines** worked better or the same as all but the best designed neural nets, for the task of handwritten digit recognition

Some history: another setback

1997 Schmidhuber and Hochreiter introduce **Long Short Term Memory** (LSTM), that essentially solves the problem of how to train recurrent neural networks

Late 90s Neural networks are still seen as a hassle to work with - the computers were not fast enough, and the algorithms were not smart enough

2001 **Random Forests** show remarkable performances, and are easier to work with than neural networks, which are once again disavowed by the machine learning community

Some history: the second rebirth

2006 Hinton, Osindero and Teh publish **A fast learning algorithm for deep belief nets**

- neural nets are rebranded as **Deep Learning**
- a new way of initializing the weights is presented, that allows to train neural networks with many layers well
- the key is having many layers of computing units so that good high-level representation of data could be learned, in contrast to hand-designing them

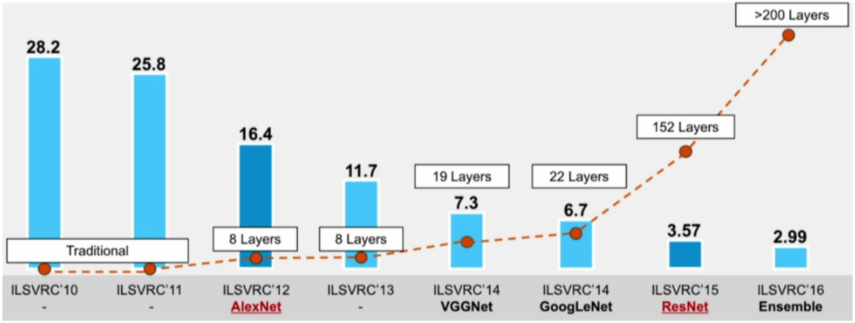
2009 Hinton et al. develop a neural network capable of superseding the existent models for phoneme recognition

2009 GPU-powered training

2012 **Microsoft, Google, IBM** and Hinton's lab publish "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups"

Some history: the final crowning

In **2012**, a convolutional neural network performs way better than previous approaches for image classification



The end (?)

So, why did purely supervised learning with backpropagation not work well in the past? Geoffrey Hinton summarized the findings up to today in these four points:

1. Our labeled datasets were thousands of times too small.
2. Our computers were millions of times too slow.
3. We initialized the weights in a stupid way.
4. We used the wrong type of non-linearity.

... well, probably not. What about the future?

Deep learning models have some *severe limitations*:

- the only real success so far has been the ability to map a space X to space Y using a continuous geometric transform, given huge amounts of annotated data
- i.e., there isn't any kind of reasoning, nor understanding behind: it is just a chain of simple, continuous geometric transformations mapping one vector space into another
- a deep learning model can be interpreted as a kind of program, but, inversely, most programs cannot be expressed as deep learning models
- e.g., even learning a sorting algorithm with a deep neural network is tremendously difficult

Key issues

In summary:

- deep learning models are severely limited in what they can represent
- given a task, even if an equivalent deep learning model does exists, it may be far too complex or simply not learnable, due to the current learning algorithms or lack of proper data
- we are still a long way from human-level AI

$\leadsto$ however, there's some light ahead…

Models as programs

Consider RNNs:

- they have slightly fewer limitations than FF networks, because they are a bit more than mere geometric transformations
- they are geometric transformations *repeatedly applied inside a for loop*, though a hardcoded one, and with severe limitations on what it does

We can build on such an idea:

- imagine a neural network augmented with programming primitives
- general for loops, if branches, while statements, variable creation, disk storage for long-term memory (NTUs), sorting operations, advanced data structures, ...

Models as programs

The space of programs that such a network could represent would be far broader, however:

- the model would be mostly no longer differentiable, even though some parts of it may still be (subroutines)
- new, complementary training strategies will be required, for example based on *evolutionary algorithms*

Once learned, subroutines may be stored in a kind of repository, so to reuse them when needed for another task, achieving higher generalization and abstraction than pretrained weights

▷ Contemporary RNNs can be seen as a prehistoric ancestor of such hybrid algorithmic-geometric models, that can be interpreted as an *artificial general intelligence* (AGI)