# Deep Learning and Temporal Data Processing

1 - Deep Neural Networks

Andrea Palazzi

June 21th, 2017

University of Modena and Reggio Emilia

# Agenda

# Introduction

# Deep Neural Networks: overview

[7]

# Linear classifiers

For the purpose of this lecture, we'll stick to the task of image classification.
Let's assume we have a training dataset of $N$ images

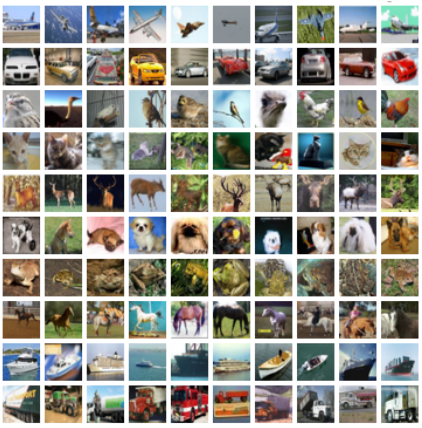
$$x_i \in \mathbb{R}^D, i = 1, \ldots, N$$

that we want to classify into $K$ distinct classes.
Thus, training set is made by couples:

$$(x_i, y_i), \ \textit{where}\ y_i \in \{1, \ldots, K\}$$

Our goal is to define a function $f : \mathbb{R}^D \mapsto \mathbb{R}^K$ that maps images to class scores.

Making a real-world example: let's take the `CIFAR-10` dataset, which consists of $N = 60000$ 32x32 RGB images belonging to 10 different classes.



Each image is 32x32x3, thus it can be thought as a column vector $x_i \in \mathbb{R}^{3072}$.
Now we can define a **linear mapping**:

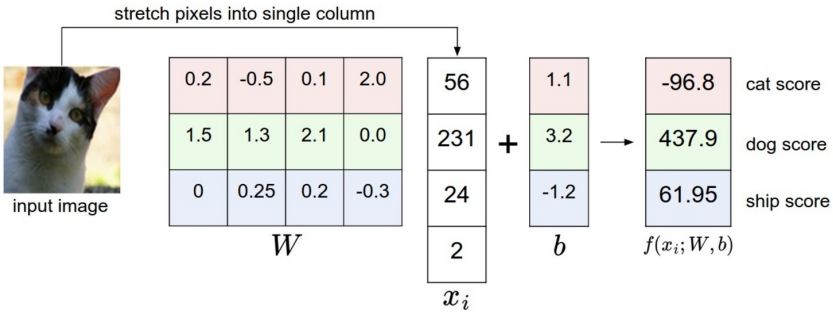
$$f(x_i, W, b) = Wx_i + b$$

where the parameters are:

- the weight matrix $W \in \mathbb{R}^{10x3072}$
- the bias vector $b \in \mathbb{R}^{10}$.

Intuitively, our goal is to learn the parameters from the training set *s.t.* when a new test image $x_i^{test}$ is given as input, the score of the correct class is higher that the scores of other classes.

# Linear Classifier




Example of mapping an image to a score. For the sake of visualization, here the image is assumed to have only 4 grayscale pixels.

# Logistic Regression

First let's introduce the **softmax function**:

$$softmax_j(\mathbf{z}) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

It takes a vector of arbitrary real-valued scores $\mathbf{z}$ and squashes it to a vector of values between zero and one that sum to one.

*e.g.*

$$\mathbf{z} = \begin{bmatrix} 1.2 \\ 5.1 \\ 2.7 \end{bmatrix} \quad softmax(\mathbf{z}) = \begin{bmatrix} 0.018 \\ 0.90 \\ 0.08 \end{bmatrix}$$

**Softmax Classifier** generalizes Logistic Regression classifier to multi-class classification.

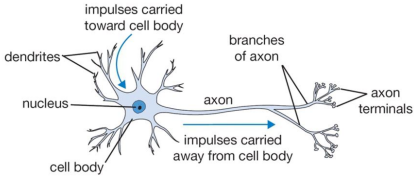
In the Softmax classifier the scores of linear function mapping $f(x_i, W) = Wx_i$ are interpreted as unnormalized log probabilities and we use the **cross-entropy loss**:

$$L_i = -log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

Prove yourself that this loss function makes sense.

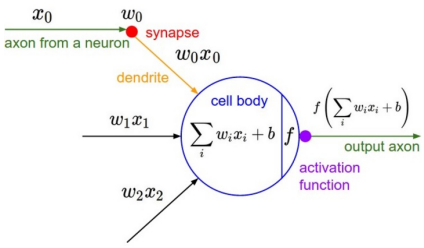
# Modeling a Neuron

Neural Networks are a mathematical model **coarsely** inspired by the way our own brain works.




Neurons are the basic computational unit of our brain. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately $10^{14}$ - $10^{15}$ **synapses**. Each neuron receives input signals from its **dendrites** and produces output signals along its (single) **axon**. The axon eventually branches out and connects via synapses to dendrites of other neurons.

More formally, we can model a single **neuron** as follows:




Each neuron can have multiple inputs. The neuron's output is the dot product between the inputs and its weights, plus the bias: then, a non-linearity is applied.

It's easy to see that a single neuron can be used to implement a binary classifier.

Indeed, when **cross-entropy loss** is applied to neuron's output, we can optimize a **binary Softmax classifier** (*a.k.a.* Logistic regression).

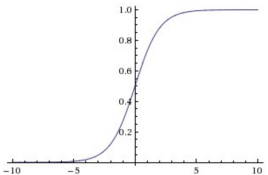
**Activation functions** are non-linear functions computed on the output of each neuron. There are a number of different activation functions you could use. In practice, the three most widely used functions have been *sigmoid*, *tanh* and *ReLu*. Nonetheless, more complex activation functions exist (*e.g.* [4, 3]).

**Why do we bother using activations?**

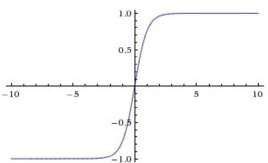
**Activation functions** are non-linear functions computed on the output of each neuron. There are a number of different activation functions you could use. In practice, the three most widely used functions have been *sigmoid*, *tanh* and *ReLu*. Nonetheless, more complex activation functions exist (*e.g.* [4, 3]).

**Why do we bother using activations?**

Composition of linear functions is a linear function. Without nonlinearities, neural networks would reduce to 1 layer logistic regression.




Sigmoid activation

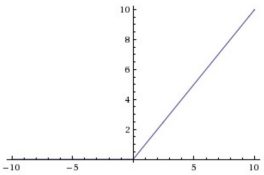



Tanh activation

**Sigmoid** nonlinearity has form $\sigma(x) = 1/(1 + e^{-x})$. Sigmoid function squashes any real-valued input into range $[0, 1]$. It has seen frequent use historically, yet now it's rarely used. It has two major drawbacks:

- saturates and kill the gradient
- output is not zero centered

**Tanh** is a scaled sigmoid neuron: $tanh(x) = 2\sigma(x) - 1$. Differently to sigmoid, input is squashed in range [-1, 1], so the output is 0-centered. However, activation can still saturate and kill the gradient.

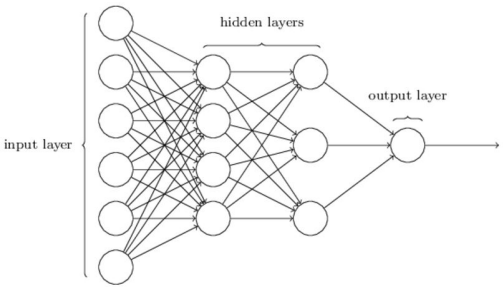



ReLu activation

**ReLu** stands for Rectified Linear Unit and computes the function $f(x) = max(0, x)$. ReLu activation was found to greatly accelerate the convergence of SGD compared to sigmoid/tanh functions [5]. Furthermore, ReLu can be implemented by a simple threshold, *w.r.t.* other activations which require complex operations.

# Neural Networks

When we connect an ensemble of neurons in an acyclic graph is when the magic happens and we get an actual **neural network**.




Neural networks are arranged in **layers**, with one *input layer*, one *output layer* and *N hidden layers* in the middle.

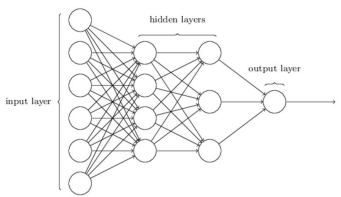
*The network depicted here has a total of* 47 *learnable parameters. Does this make sense to you?*

The 4-layer network previously depicted can be simply expressed as:

$$out = \phi(\mathbf{W_3}\phi(\mathbf{W_2}\phi(\mathbf{W_1}\mathbf{x}))) \tag{1}$$

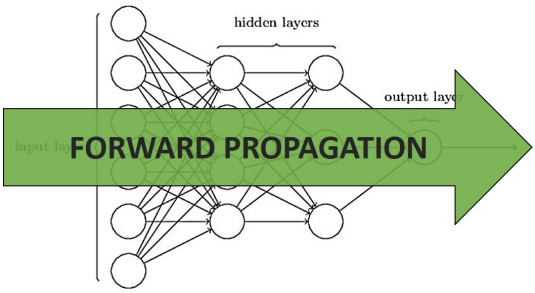
where:

- $\phi$ is the activation function
- $\mathbf{x} \in \mathbb{R}^6$ is the input
- $\mathbf{W_1} \in \mathbb{R}^{4\times6}$ are the weights of first layer
- $\mathbf{W_2} \in \mathbb{R}^{3\times4}$ are the weights of second layer
- $\mathbf{W_3} \in \mathbb{R}^{1\times3}$ are the weights of third layer




Notice that to ease the notation biases have been incorporated into weight matrices W.

**Forward propagation** is the process of computing the network output given its input.




The formula of forward propagation for our toy network above is the one in Eq. 1.

It has been shown (*e.g.* [1]) that any continuous function $f(x)$ can be approximated at arbitrary precision $\epsilon > 0$ by a neural network $g(x)$ with at least one hidden layer. That is:

$$\forall x, \quad |f(x) - g(x)| < \epsilon$$

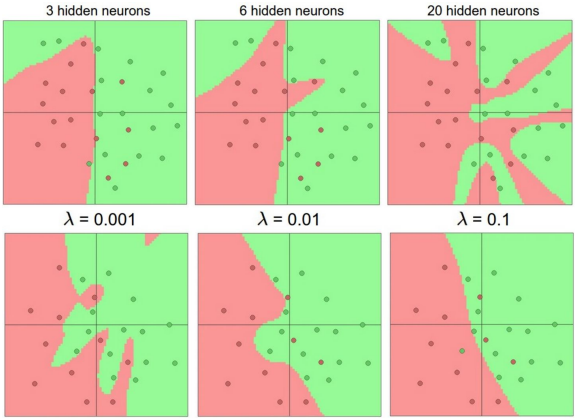
For this reason neural networks with at least one hidden layers are referred to as **universal approximators**. In practice however networks with more layers often outperform 2-layer nets, despite the fact that on paper their representational power is equal.

The number of layers in the network, as well as the number of neurons in each layers, are so called **hyperparameters** of the architecture.

Keep in mind that:

- the bigger the size and the number of layers, the more the **capacity** of the network increases, which is good because the space of representable functions grow.
- bigger networks without proper regularization are way more prone to **overfit** the training data, which is bad.

# Setting Hyperparameters




In practice, usually networks are made as big as computational budget allows.

Then overfitting is prevented through proper **regularization techniques** (*e.g.* dropout, L2 regularization, input noise).

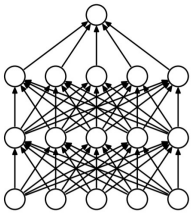
# Training a DNN

When initializing the weights in the neural network, turns out that **random initialization** is very important to **break the symmetry**.

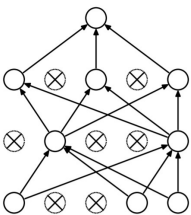
Indeed, if all weights were initialized to the same value (*e.g.* 0) then all neurons would compute the same output, the same gradient and would eventually undergo to the same update. Clearly, this is bad.

Common practice is to initialize all weights to small random numbers centered on zero (eventually scaled by neuron's *fan-in* [2, 4]) and all biases to zero.

# Dropout



(a) Standard Neural Net



(b) After applying dropout.

Besides traditional regularization techniques (*e.g.* $L_n$ weight regularization), neural networks feature a simple and extremely effective regularization which is **dropout** [6].

During the training process, each neuron is set to zero (dropped) with a *drop probability* $p$. This can be seen as sampling at each training step a different sub-network and updating only the corresponding portion of parameters.

At testing time, no dropout is applied. This can be interpreted as taking the average prediction of the ensemble of sub-networks.

# Backpropagation

# Credits

# Credits

These slides heavily borrow from the following Stanford course:

- http://cs231n.stanford.edu/

if you want to deepen these concepts, please start from here!

Also, nice convolution animations are taken from here:

- https://github.com/vdumoulin/conv_arithmetic

# References

[1] G. Cybenko.
**Approximation by superpositions of a sigmoidal function.**
*Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[2] X. Glorot and Y. Bengio.
**Understanding the difficulty of training deep feedforward neural networks.**
In *Aistats*, volume 9, pages 249–256, 2010.

[3] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio.
**Maxout networks.**
*arXiv preprint arXiv:1302.4389*, 2013.

[4] K. He, X. Zhang, S. Ren, and J. Sun.
**Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.**
In *Proceedings of the IEEE international conference on computer vision*, pages