Machine learning and statistical learning

Chapter 5. Deep learning

Ewen Gallic
ewen.gallic@gmail.com

MASTER in Economics - Track EBDS - 2nd Year

2018-2019

1. Introduction

Introduction

In the previous chapters, we have looked at models that rely on **past observations to learn from experience**, *i.e.*, whe have looked at **machine learning** methods.

We have distinguished between regression and classification problems, in a context of a **supervised learning** approach.

In both situations, the goal is to learn a mapping from inputs $\mathbf{x}$ to **labeled outputs** y, where y can be either **quantitative** (regression problems) or **qualitative** (classification problems).

In this chapter, we will also try to learn a mapping from inputs to outputs.

We will allow data to have a complex architecture and envisage combining different non-linear transformations.

Principle in a nutshell

Up to know, what we have done can be viewed as the use of a simple **two-layer architecture**:

- first layer: the input data
- second layer: the output data

This can be viewed as a single neural network.

The idea behind deep learning is to :

- extract **linear combinations of the inputs** as derived predictors
- use a **combination of simple non-linear functions** on these predictors to predict the output.

Instead of a two-layer architecture, deep learning allow for more layers.

Deep learning is built on a **combination of neural networks**, which, when combined, form a **deep neural network**.

Different architectures

Different types of architecture exist for neural networks, among which:

- the multilayer perceptrons (MLP)
  - oldest and simplest one
  - the input data goes through the nodes and exit on the output nodes
- the convolutional neural networks (CNN)
  - based on using convolutions in place of general matrix multiplication in at least one of the layers
  - used to process data that has a grid-like topology (*e.g.*, time series in 1D or images in 2D)
- the recurrent neural networks (RNN)
  - the output of a layer is saved and fed back to the input
  - at each step, each neuron remembers some information from the previous step
  - this is useful for sequential data (*e.g.*, text or time series)

Useful References

- Ambroise and Donnet (2018), *Deep learning pour et par les nuls*.
- Desboulets (2018), *Artificial Neural Networks in Regression Analysis*
- Goodfellow et al. (2016), *Deep learning*. MIT press Cambridge.
- Hastie et al. (2001), Neural networks. In *The Elements of Statistical Learning*, chapter 11, pages 389–416. Springer New York.
- Ng and Katanforoosh (2018), *Deep Learning, CS229 Lecture Notes*
- Wikistat (2016), *Neural Networks and Introduction to Deep Learning — wikistat* (by Philippe Besse)
- Multi-Layer Neural Network. In *Deep Learning Tutorial*, Stanford University.
- 3Blue1Brown. (2017). *Deep Learning*, Youtube. (chapters 1 to 4).
- LeCun, Y. (2016) L'apprentissage profond : une révolution en intelligence artificielle. Leçon inaugurale, Collège de France. (In French)

2. Artificial Neural networks

A quick recap of Pierre Michel's lesson.

We will begin with a quick recap of Pierre Michel's lesson on Artificial Neural Network.

Artificial Neural networks

We consider an output y and some input $\mathbf{x}$ data from a training set: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$.

An **Artificial Neural Network** is an application non linear with respect to in its parameters $\theta = (W, b)$ which associates to a vector of inputs $\mathbf{x}$ an output y that we will denote:

$$y = h_{W,b}(\mathbf{x}), \tag{1}$$

where W and b are the parameters we want to estimate using the training data.

To estimate these parameters, we seek to minimize a non convex cost function. It can be done by computing the gradient of the neural network (this is known as **backpropagation of the gradient**).

2.1 Artificial neuron

Artificial neuron

The kth **artificial neuron** is a function h_k of

- the input $\mathbf{x}_k$
- weighted by a vector of connection weights $W_k = (w_{k1}, \ldots, w_{km})$ between neuron k and the m inputs
- and an intercept parameters b_k (also known as a **neuron bias**)
- associated to an **activation function** f_k

That is, the kth artificial neuron writes:

$$y_k = h_i(\mathbf{x}_k) = f_k\left(\sum_{i=1}^{m} W_k \mathbf{x}_k + b_k\right) \tag{2}$$

Activation functions

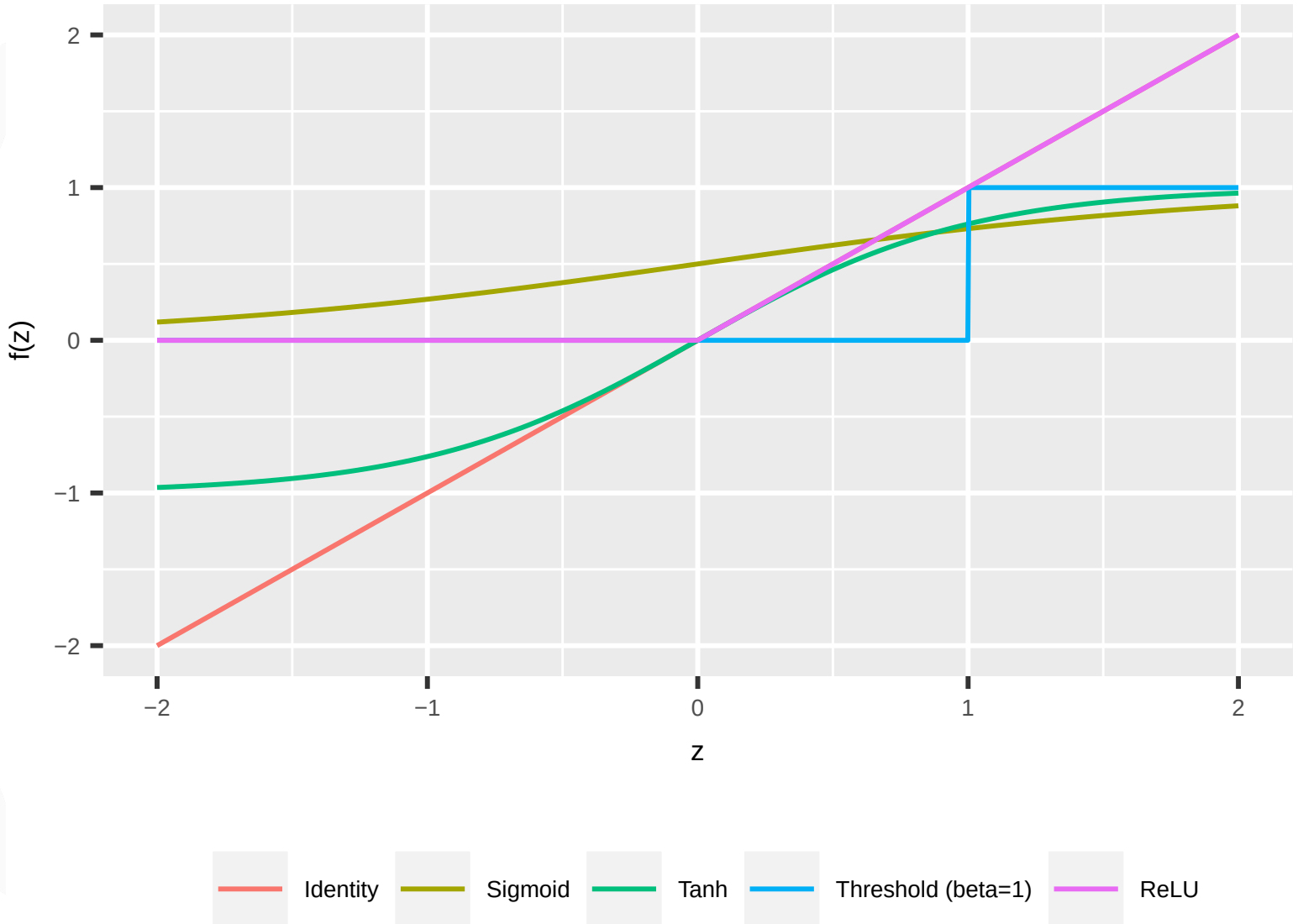
Different activation functions can be chosen:

- the **identify function**: $f(z) = z$
- the **sigmoid/logistic function**: $f(z) = \frac{1}{1+\exp(-z)}$
  - $\mathbb{R} \to [0, 1]$
- the **softmax function**: $f(z)_j = \left(\frac{\exp(z_j)}{\sum_{k=1}^{K} \exp z_k} \right)$, for $j \in \{1, \dots, K\}$, where K is the number of elements of the input vector
  - Generalization of the logistic function
  - $\mathbb{R} \to [0, 1]^K$

Activation functions

- the **hyperbolic tangent function**:
  $f(z) = \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} = \frac{\exp(2z)-1}{\exp(2z)+1}$
  - $\mathbb{R} \to [-1, 1]$
- the **hard threshold function**: $f_\beta(z) = \mathbb{1}_{(x \geq \beta)}$
  - $\mathbb{R} \to [0, 1]$
- the **Re**ctified **L**inear **U**nit function (ReLU): $f(z) = \max(0, z)$
  - $\mathbb{R} \to [0, max(z)]$, allows to predict positive values. The ReLu function is continuous but not derivable.

Activation functions



2.2 Multilayer perceptron

Multilayer perceptron

The **Multilayer perceptron** is a structure composed of several **hidden layers** of neurons.

We will use l to denote the lth layer among the L layers of the neural network.

The output of the ith neuron of the lth layer becomes the input of the $l+1$ layer.

Note: In the case of recurrent neural networks, the output of the lth layer can also be the input of the same layer or previous layers.

The **activation functions** are the same in each layer, **except for the last layer**, in which a different activation function can be chosen depending on the objective of the estimation (regression or classification).

Parameters

The **parameters** of a neural network are:

- the number $L-2$ of hidden layers
- the number s_ℓ of neuron in each hidden layer $\ell = 1, \ldots, L-1$
- the activation function f for the the hidden layers
- the activation function ψ for the output layer.

The **choice of the activation function** ψ for the output layer depends on the problem:

- in a **regression problem**: identity or linear function
- in a **classification problem**: logistic (if binary response variable) softmax (if multiclass response variable)

But, as usual, there is no free-lunch in the choice of the activation function...

An example

For convenience, let us consider an example.

Let us say we want to train an ANN to classify the numbers from the MNIST dataset.

Recall that each training example consists in values of greyscale levels (ranging from 0 to 255) for each of the pixels of a 28×28 grid.

There are therefore $28 \times 28 = 784$ neurons in the input layer.

In the output layer, we want 10 neurons, one for each of the classes (0, 1, 2, . . . , 9).

Let us furthur suppose that we want two hidden layers, with 4 neurons in each layers.

Weights

In the first hidden layer, let us suppose that one of the neurons will be trained to specifically detect a given pattern in the input data.

To each point of the 28×28 grid that serves as an input data, a weight can be associated.

Now, let us denote a_1, a_2, ..., a_n the n inputs (in our example, $n = 784$). Let us denote w_1, w_2, ..., w_n the associated weights, *i.e.*, the weights from the neuron of interest and the neurons from the first layer.

The weighted sum of the activation from the first layer writes:

$$w_1 a_1 + w_2 a_2 + \ldots + w_n a_n \tag{3}$$

These weights could, for example, be set according to the following rule:

- a positive weight for brighter points in the image
- a negative weight for darker ones

Weights and bias

The resulting value of the weighted sum can be any number in $\mathbb{R}$.

By applying an **activation function** to this value, we can end-up with values comprised between a certain range of values:

- for example, if we use the logistic function, we can end-up with values comprised between 0 and 1.

Once again, the objective of applying an activation function is to **obtain a signal on whether or not the neuron is activated**.

$$f(w_1 a_1 + w_2 a_2 + \ldots + w_n a_n)$$

We may want the neuron to be activated only when the weighted sum amounts a certain value. To achieve this, **we can add a bias** prior to the application of the activation function:

$$f(w_1 a_1 + w_2 a_2 + \ldots + w_n a_n + b)$$

Weights and bias

The question here is: **what should the weights and bias be so that the ANN can be able to correctly classify the numbers provided as inputs?**

In our example, we need to estimate the value of the $784\times4+4\times4+4\times10=3192$ weights of the ANN and the value of the $4+4+10=18$ biases.

For a specific neuron k in the ℓth layer, the output value of this neurons writes:

$$a_k^{(\ell)} = f\left(w_{k,1}^{(\ell)}a_1^{(\ell-1)} + w_{k,2}^{(\ell)}a_2^{(\ell-1)} + \ldots + w_{k,s_{\ell-1}}^{(\ell)}a_{s_{\ell-1}}^{(\ell-1)} + b_k^{(\ell)}\right) \quad (4)$$

Vectorial notation

The neurons from the ℓth layer can be represented using a vectorial notation:

$$a^{(\ell)} = f\left(W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}\right),$$

where

$$W^{(\ell)} = \begin{bmatrix} w_{1,1}^{(\ell)} & w_{1,2}^{(\ell)} & \dots & w_{1,s_{\ell-1}}^{(\ell)} \\ w_{2,1}^{(\ell)} & w_{2,2}^{(\ell)} & \dots & w_{2,s_{\ell-1}}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,1}^{(\ell)} & w_{k,2}^{(\ell)} & \dots & w_{k,s_{\ell-1}}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{s_\ell,1}^{(\ell)} & w_{s_\ell,2}^{(\ell)} & \dots & w_{s_\ell,s_{\ell-1}}^{(\ell)} \end{bmatrix}, \quad a^{(\ell-1)} = \begin{bmatrix} a_1^{(\ell-1)} \\ a_2^{(\ell-1)} \\ \dots \\ a_k^{(\ell)} \\ \dots \\ a_{s_{\ell-1}}^{(\ell-1)} \end{bmatrix}, \text{ and } b^{(\ell)} = \begin{bmatrix} b_1^{(\ell)} \\ b_2^{(\ell)} \\ \vdots \\ b_k^{(\ell)} \\ \vdots \\ b_{s_{\ell-1}}^{(\ell)} \end{bmatrix}$$

This writing is convenient: we can see that it is possible to apply f to each row of the vector obtained by computing $W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}$.

2.2.1 Training a neural network

In a nutshell

The question we asked still holds: how can we estimate the weights and biases? In other words, how does the neural network learns to obtain good performances?

As usual, in a nutshell, the idea is to split the data into two parts: (i) a **training sample** and (ii) a **test sample**. Then:

1. first **initialize** the weights and biases by picking **random values** according to a pseudo-random number generator
2. set-up a **cost function** to measure how wrong we are when predicting given some values for the weights and bias
3. minimize that cost function using an **optimization algorithm** with respect to the weights and bias and given some training examples.

Cost function

The natural thing to consider to know how good we are at predicting given some values for the weights and bias, is to define a cost function.

Let us consider the **cost of a single training example first**. The latter can be defined as the square of the errors between the value of activation and the value we desire, summed for all neurons in the output layer.

Then, to measure the cost of **all training examples**, we can simply compute the average of the cost computed for each training example.

The aim is to **find the parameters that minimize this cost**.

To minimize the cost function, we can, once again, rely on a gradient descent algorithm.

Minimizing the cost function

The details of the algorithm are provided in Pierre Michel's course.

Let us just quickly recall the main steps of the algorithm here.

Given a starting point for the parameters, we want to:

1. find the direction of the steepest ascent
   - that corresponds to compute the gradient of the cost function: ∇J
2. take a small step in the opposite of that direction: $-\nabla J$
3. repeat these first two steps until convergence to a local minimum.

Minimizing the cost function

In our example of the 4-layer ANN, recall that there are 3192 parameters to estimate. Consider that they are put into a vector:

$$\overrightarrow{W} = \begin{bmatrix} w_{1,1}^{(\ell)} \\ w_{1,2}^{(\ell)} \\ \vdots \\ b_{s_{\ell-1}}^{(\ell)} \end{bmatrix}$$

The negative gradient vector $-\nabla J(\overrightarrow{W})$ will contain at each step, a value indicating the step for each parameter that should be taken to decrease the cost function.

Minimizing the cost function

Intuitively, if the value of the kth element of the negative gradient vector of the cost function is negative (positive), the value of the corresponding parameter should decrease (increase).

The relative magnitude also indicates how far we should move. It provides information on how more sensitive the cost function to a change in the value of a parameter relatively to a change in another parameter.

Effect of a single example on the cost

Let us consider a single training example, for instance, a 9.

We want to know what is the effect of that example on how the weights and bias should be adjusted.

At the beginning of the iteration of the optimization algorithm, the activation in the output layer probably gives terrible predictions with the starting values provided for the parameters.

Even if we can only change the values of the weights and biases, we know more information we can use: **we know what adjustment we whish should take place**.

- If the activation neuron for class 9 in the output layer is too low, we want it to increase
- If the activation neuron for any other class in the output layer is too high, we want it to decrease
  - the size of the decrease should be proportional to how far away the current value is from its target value.

Effect of a single example on the cost

The value of the output neuron for class 9 is obtained with:

$$a_{10}^{(L)} = \psi\left(w_{10,1}^{(L)} a_1^{(L-1)} + w_{10,2}^{(L)} a_2^{(L-1)} + \ldots + w_{10,s_L}^{(L)} a_{s_L}^{(L-1)} + b_{10}^{(\ell)}\right) \quad (5)$$

There are therefore three ways of changing the value:

1. changing the weights
2. changing the bias
3. changing the activation from the previous layer.

Effect of a single example on the cost

$$a_{10}^{(L)} = \psi\left(w_{10,1}^{(L)} a_1^{(L-1)} + w_{10,2}^{(L)} a_2^{(L-1)} + \ldots + w_{10,s_L}^{(L)} a_{s_L}^{(L-1)} + b_{10}^{(\ell)}\right)$$

In that formula, we we can clearly see that the **weights have different levels of influence on the activation of the neuron**:

- the connections with neurons from the previous layer with higher values ($a_k^{(L-1)}$) have relatively larger effects on the activation
  - this is due to the fact that the weights $w_{10,k}^{(L)}$ are multiplied by larger values $a_k^{(L-1)}$

Hence, for the current training example, increasing these weights will have more impact on the output value than increasing the weights associated with neurons from the previous layer with smaller values.

To sum up: **the increase in $w_{10,k}$ is proportional to $a_{10,k}$**.

Effect of a single example on the cost

$$a_{10}^{(L)} = \psi\left(w_{10,1}^{(L)} a_1^{(L-1)} + w_{10,2}^{(L)} a_2^{(L-1)} + \ldots + w_{10,s_L}^{(L)} a_{s_L}^{(L-1)} + b_{10}^{(\ell)}\right)$$

What if, instead of changing the value of the weights $w_{10,k}^{(L)}$ or the bias $b_{10}^{(\ell)}$ we want to **influence the value of the activation** of the neuron by **changing the activation value from the previous layer**, *i.e.*, changing $a_k^{(L-1)}$?

As we wish $a_{10}^{(L)}$ to increase here with our training example:

- if $a_k^{(L-1)}$ is connected to the neuron with a positive weight: we want $a_k^{(L-1)}$ it to increase
- if $a_k^{(L-1)}$ is connected to the neuron with a negative weight: we want $a_k^{(L-1)}$ it to decrease

Hence, the change in $a_k^{(\ell)}$ should be proportional to the size of $w_{10,k}^{(\ell)}$.

Effect of a single example on the cost

Unfortunately, it is not possible to directly change the value of a_k^{L-1} directly.

But it is possible to **keep a track on the desired change**.

Backward propagation

So far, with our training example, we have looked at the neuron for the class 9 in the ouput layer.

We want the value for the activation of this neuron to increase.

But in the mean time, we also want the values of the other neurons in the output layer to decrease.

The same kind of reasonning as that of the 10th neuron applies. **We want to keep track of all desired changes** (direction and relative magnitude) of the weights, the bias and the activation values for each neuron from the previous layer.

So, for our training example, we add all the desired effects we want to obtain on the activation value for each neuron in the layer $L-1\ldots$

and we apply **backward** the same process to the weights and biases that are used to determine their values (in the previous layer), and so on.

Backward propagation

At the end of the backward iteration process, we can get a value of how much the weights and bias should be changed for this single training example.

We then need to go through **all training examples**, and a average the value of these changes to obtain a vector proportional to the **negative gradient vector of the cost function**.

But... this is **computationally very expensive**!

To reduce these computantionnal costs: **mini-batch gradient descent**.

Mini-batch gradient descent

The mini-batch gradient descent consists in:

1. schuffle the training examples
2. group them by **mini-batches** (*e.g.*, by putting $n = 128$ training examples in each batch)
3. compute a **step of the gradient descent** according to a mini-batch :
   - as the gradient of the cost function is computed using a subset of the training example, the resulting value will not necessarily correspond to the most efficient step

The parameters are then updated for each mini-batch of n training examples, instead of being updated after each training example.

Backpropagation: more details

Now, let us turn to a more formal approach. For the sake of clarity, we will first consider a flat neural network with 4 layers, 1 neuron in each of those.

The cost function, in that minimalist example reponds to variations of 6 parameters: $w_1, b_1, w_2, b_2, w_3, b_3$.

We want to look at how sensitive the cost function is to these parameters, and what adjustments need to be made to decrease its value.

Backpropagation: more details

Let us denote:

- y the desired output of a training example
- a_L the activation of the last neurone
- a_{L-1} the activation of the neuron in the $L-1$ layer

The cost function for a given training example writes:

$$J_0(\dots) = \left(a^{(L)} - y\right)^2$$

Let $z^{(L)}$ be defined as:

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

The activation of the neuron in the Lth layer is computed as:

$$a^{(L)} = \psi\left(z^{(L)}\right)$$

Sensitivity of J_0 to the weights

How sensitive is the cost function to the weights?

Mathematically, this corresponds to compute the partial derivative of the cost function J_0 with respect to the weight $w^{(L)}$:

$$\frac{\partial J_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial J_0}{\partial a^{(L)}} \tag{6}$$

We then need to compute each of these partial derivatives.

Sensitivity of J_0 to the weights

$$\frac{\partial J_0}{\partial a^{(L)}} = 2\left(a^{(L)} - y\right) \tag{7}$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \psi'\left(z^{(L)}\right) \tag{8}$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} \tag{9}$$

Hence, for our training example:

$$\frac{\partial J_0}{\partial w^{(L)}} = a^{(L-1)}\psi'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right) \tag{10}$$

Sensitivity of J to the weights

Now, to get how sensitive the cost function is to the weight for all training examples, we simply need to average all the costs from each training example:

$$\frac{\partial J}{\partial w^L} = \frac{1}{n} \sum_{k=1}^{n} \frac{\partial J_k}{\partial w^{(L)}} \tag{11}$$

This corresponds to one value in the vector of the gradient of the cost function:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial w^{(1)}} \\ \frac{\partial J}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial J}{\partial w^{(L)}} \\ \frac{\partial J}{\partial b^{(L)}} \end{bmatrix}$$

Sensitivity of J_0 to the bias

Now let us turn to the sensitivity of the cost function to the bias.

For a given training example:

$$\frac{\partial J_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial J_0}{\partial a^{(L)}} \tag{12}$$

$$= \psi' \left(z^{(L)} 2 \left(a^{(L)} - y \right) \right), \tag{13}$$

since $\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1$.

Again, to get how sensitive the cost function is to this bias for all training examples, we simply need to average all the costs from each training example:

$$\frac{\partial J}{\partial b^{(L)}} = \frac{1}{n} \sum_{k=1}^{n} \frac{\partial J_k}{\partial b^{(L)}} \tag{14}$$

Sensitivity of J_0 to the activation of the previous layer

Now let us turn to the sensitivity of the cost function to the activation of the previous layer, for a given training example.

$$\frac{\partial J_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial J_0}{\partial a^{(L)}} \tag{15}$$

$$= w^{(L)} \psi' \left(z^{(L)} 2 \left(a^{(L)} - y \right) \right), \tag{16}$$

Indeed,

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)}$$

We can keep iterating this chain rule idea backwards...

Neural network with multiple neurons in each layer

The same idea applies with multiple neurons in each layer of the ANN.

Let us denote :

- $a_k^{(L-1)}$ the activation of the kth neuron in layer $L-1$
- $a_j^{(L)}$ the jth neuron in layer L
- y_j the desired output in the jth neuron in the last layer L
- $w_{jk}^{(L)}$ the weight of the edge connecting the kth neuron from the layer $L-1$ to the jth neuron from the layer L

Using these notations, for neuron j in layer L:

$$J_0 = \sum_{j=1}^{s_L} \left(a_j^{(L)} - y_j \right)^2 \tag{17}$$

$$z_j^{(L)} = w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + \dots w_{js_L}^{(L)} a_{s_L}^{(L-1)} + b_j^{(L)} \tag{18}$$

$$a_j^L = \psi \left(z_j^{(L)} \right) \tag{19}$$

Neural network with multiple neurons in each layer

The derivative of the cost function for a given training example with respect to the weight in the current layer between the kth neuron from the previous layer and the jth neuron from the current layer writes:

$$\frac{\partial J_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial J_0}{\partial a_j^{(L)}} = a_k^{(L-1)} \psi' \left(z_j^L \right) 2 \left(a_j^{(L)} - y_j \right) \tag{20}$$

The derivative with respect to the bias in the jth neuron from the current layer writes:

$$\frac{\partial J_0}{\partial b_j^{(L)}} = \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial J_0}{\partial a_j^{(L)}} = \psi' \left(z_j^L \right) 2 \left(a_j^{(L)} - y_j \right) \tag{21}$$

Neural network with multiple neurons in each layer

The derivative of the cost function for a single training example, with respect to the activation from the kth neuron of the previous layer writes:

$$\frac{\partial J_0}{\partial a_k^{(L-1)}} = \sum_{j=1}^{s_L} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial J_0}{\partial a_j^{(L)}} = \sum_{j=1}^{s_L} w_{jk}^{(L)} \psi'\left(z_j^L\right) 2\left(a_j^{(L)} - y_j\right) \tag{22}$$

We need to sum over the kth neuron from layer $L-1$ influences the coest function through multiple paths:

- $a_k^{(L-1)}$ influences $a_1^{(L)}$
- $a_k^{(L-1)}$ influences $a_2^{(L)}$
- ...
- $a_k^{(L-1)}$ influences $a_{s_L}^{(L)}$

Neural network with multiple neurons in each layer

We know how sensitive the cost function is to the activation in this $L-1$ layer.

All we need to do is to repeat the same process for all weights and biases into that layer...

2.3 Universal approximation theorem

Universal approximation theorem

Hornik (1991) has shown that :

- a **feed-forward** 3-layers Neural Network
- with a finite number of neurons
- the same activation for each neuron in the hidden layer
- the identity activation function for the output layer

can approximate any bounded and regular function from $\mathbb{R}^p$ to $\mathbb{R}$

The **universal approximation theorem** therefore states that simple neural networks can represent a wide variety of functions.

Universal approximation theorem

But, as stated by Goodfellow et al. (2016) :

- "we are not guaranteed that the training algorithm will be able to learn that function" :
  - the optimization algorithm may not be able to find the value of the parameters that corresponds to the desired function
  - there is a risk of overfitting
- "the layer may be infeasibly large and may fail to learn and generalize correctly".

Universal approximation theorem

So, in practice, using deeper models:

- can reduce the number of units that are required to represent the function we want to approximate
- can reduce the amount of generalization error.

The strength of deep learning lies in the number of hidden layers.

Ambroise, C. and Donnet, S. (2018). Deep learning pour et par les nuls. [Tutorial given at the 7th R French meeting. Online; Page available on March 2019].

Desboulets, L. (2018). Artificial neural networks in regression analysis. [Seminar notes. Online; Page available on March 2019].

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Hastie, T., Friedman, J., and Tibshirani, R. (2001). Neural networks. In *The Elements of Statistical Learning*, chapter 11, pages 389–416. Springer New York.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

Ng, A. and Katanforoosh, K. (2018). Deep learning. [CS229 Lecture Notes. Online; Page available on March 2019].

Wikistat (2016). Neural networks and introduction to deep learning — wikistat. [By Philippe Besse. Online; Page available on March 2019].