Projet R Reseaux de Neurones

In this paper we will try to explain how a neural network works in the case of a classification. We have a wide range of clothes' pictures that we want to predict the category(=classes).

Let's have a look at our data Fashion-MNIST, it is a dataset of Zalando's article images, there Each image are represented by 28 pixels in height and 28 pixels in width, for a total of 784 pixels. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels, and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.

Here are the figures of the dataset :

- 70000 pictures, where 48000 are used for training, 12000 for validation and 10000 for testing,
- 10 classes : T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag and Ankle boot.

1 Neural network definition

Let $x_j \in \mathbb{R}^{n_j}$ be the vector of neurons of the j-th layer, for all j=$1,\dots,J$, and let x_0 be the input layer,

Let $W_j \in M_{n_j,n_{j-1}}$ and $b_j \in \mathbb{R}^{n_j}$ be respectively the Matrix of weights and the biais of the j-th layer.

Each row of the matrix W_j corresponds to the weights of a neuron of the $j-1$-th layer.

Then we have for all j,

$$(*) \quad x_j = \nu_j(W_j x_{j-1} + b_j)$$

where $\nu_j : \mathbb{R}^{n_j} \to \mathbb{R}^{n_j}$ denote the activation function of the j-th layer.

Tipically we can use $Relu$ function defined as $f : \mathbb{R} \to \mathbb{R}, x \to max(0,x)$.

With this function any output neuron of the hidden layer is x if $< w,x > + b$ is positive, O otherwise, in other words the output is x if $x \in H : \{x : \ < w,x > +b > 0 \}$ the hyperplan constructed with the weight and the biais, the neural network approximate the optimal hyperplan.

We see here that in order to obtain one layer given the previous one, we just need to apply a **linear transformation** (matrix mulitplication and adding a biais) following by applying a **non-linear activation function**.

The hyperparameters that we have to determine are all the weights and all the biais. In total we have $\sum_{j=1}^{J} n_j n_{j-1} + n_j$ hyperparameters.

2 Loss function

2.1 General definition

In the general case, the loss is the prediction error we are making given θ parameter and can be estimated with our training dataset $(x_i, y_i)_{i=1,\dots,n}$ by : $L(\theta)$ = $\frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i) \approx E(l(\theta,X,Y))$

2.2 Classification case

As our aim is to predict a bellonging to a class, the **J-th** layer need to be of length 10, we will then apply a **softmax** function to interpret the output as the probability that the image bellongs to a class.

For the last layer x_J = $(x_J(1), \dots, x_J(10))$ we apply the **softmax** as follow,

$\hat{y}$ = **softmax**(x_J) = ($\frac{e^{x_J(1)}}{\sum_{k'=1}^{10} e^{x_J(k')}}, \dots, \frac{e^{x_J(10)}}{\sum_{k'=1}^{10} e^{x_J(k')}}$) this vector can be seen as a probability distribution.

2.3 Loss function as Cross entropy

Then we want to compare these probability to our response **y** = (0,...,0,1,0,...,0) a vector of zeros expect for the class (denoted by k) for which the image belongs.

The loss function needs to compare two probability distribution, we naturally use the cross entropy.

For any data (x_i, y_i) from the training set, the loss function measure the error we have made between our prediction and the true response :

$$\begin{aligned} (**) \quad l(\theta, \hat{y}_i, y_i) &= -\sum_{k'=1}^{10} y_{k'} log(softmax(\hat{y_{k'}})) \\ &= -1 * log(softmax(\hat{y}_k)) \\ &= log(\sum_{k'=1}^{10} e^{\hat{y_{k'}}}) - \hat{y}_k \end{aligned}$$

where θ = $(W_j, b_j)_{j=1,\dots,J}$ is the parameter to be estimated by the network.

3 Gradient Descent

3.1 Minimization problem

We want to minimize the average loss (which is an estimation of the true loss) with respect to θ parameter.

$$(P) \quad \min_\theta L(\theta) = \min_\theta \frac{1}{n} \sum_{i=1}^{n} l(\theta, x_i, y_i)$$

3.2 Batch Gradient Descent

To solve (P) we use a gradient descent, we go to the direction for which the derivative is minimal so that the loss function decreases as much as possible.

The optimal direction of norm 1 is the opposite of the gradient, indeed, for any function f and direction d, $\frac{\partial f}{\partial d}$ = $< \nabla f, d > \leq \|\nabla f\| \ \|d\|$ = $\|\nabla f\|$ by Cauchy-Schwartz and the supremum is reached for d = $\frac{\nabla f}{\|\nabla f\|}$. So by taking - d the function decreases as much as possible.

$$Batch \ Gradient \qquad \theta_{n+1} = \theta_n - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta l(\theta_n, x_i, y_i)$$

"Batch" means that we use the whole dataset to compute the gradient of the loss.

Therefore for each iteration we would need to compute n gradient, but n can be a very large number (million) and it would take too much time to converge to the optimal θ.

To estimate the error of prediction one can also decide to take less than n observation in order to reduce gradients' time computation.

3.3 m-Batch Gradient Descent

We then **randomly** choose without remplacement a batch of m observations for the descent. Tipically m ranges from 50 to 200.

Note that the so called **Stochastic Gradient Descent** is a particular case where m=1.

$$m - Batch \ Gradient \qquad \theta_{n+1} = \theta_n - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta l(\theta_n, x_i, y_i)$$

The number of iteration can be chosen by a tolerance with respect to the the gradient norm and by a number of 'epochs'.

Definition of epoch : One epoch is when an entire dataset is passed through the neural network only once, i.e when n gradients have been computed.

So in our algorithm, we visit nb_epoch time the whole data.

The gradient descent algorithm (GD) can be expressed by the following pseudo-code :

```
theta = theta_0
for i from 1 to nb_epochs:
data = shuffle(data)
batches = get_batches(data, batch_size=m)
for batch in batches:
gradient = gradient_computation(loss_function, batch, theta)
theta = theta - learning_rate * gradient
```

Note that **get_batches** is a function separating the data into groups of batches of size m.

Why do we use more than one epoch ? Gradient Descent is an iterative process, and we are using a limited dataset to optimize the learning. So, updating the weights with single pass or one epoch is not enough. In this case one iteration of the GD algorithm is when **theta** is updated once, so in total we get :

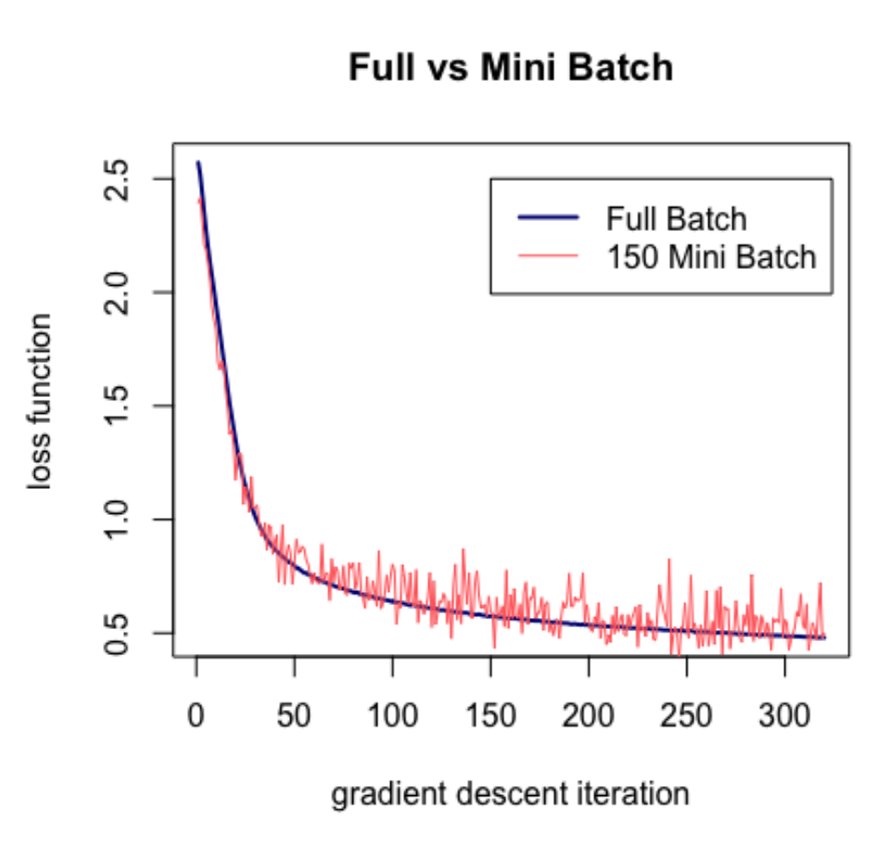
$$\begin{aligned} Number \ of \ iteration &= epochs * \frac{n}{m} \\ Gradient \ computation \ by \ iteration &= m \end{aligned}$$

where,

- $epochs$ denote the number of epoch
- n is the length of the dataset
- m is the size of the batch.

3.4 Full Batch vs m-Batch Gradient Descent convergence

Although we are always computing n gradients by epoch, the number of iteration depends of the size of the batch.



Although we are always computing n gradients by epoch, the number of iteration by epoch depends of the size of the batch.

Indeed in the graph below, we are comparing the full batch gradient descent (i.e batch size = n = 60000) with the mini-batch gradient descent (with batch size = m = 150).

At the 100-th iteration :

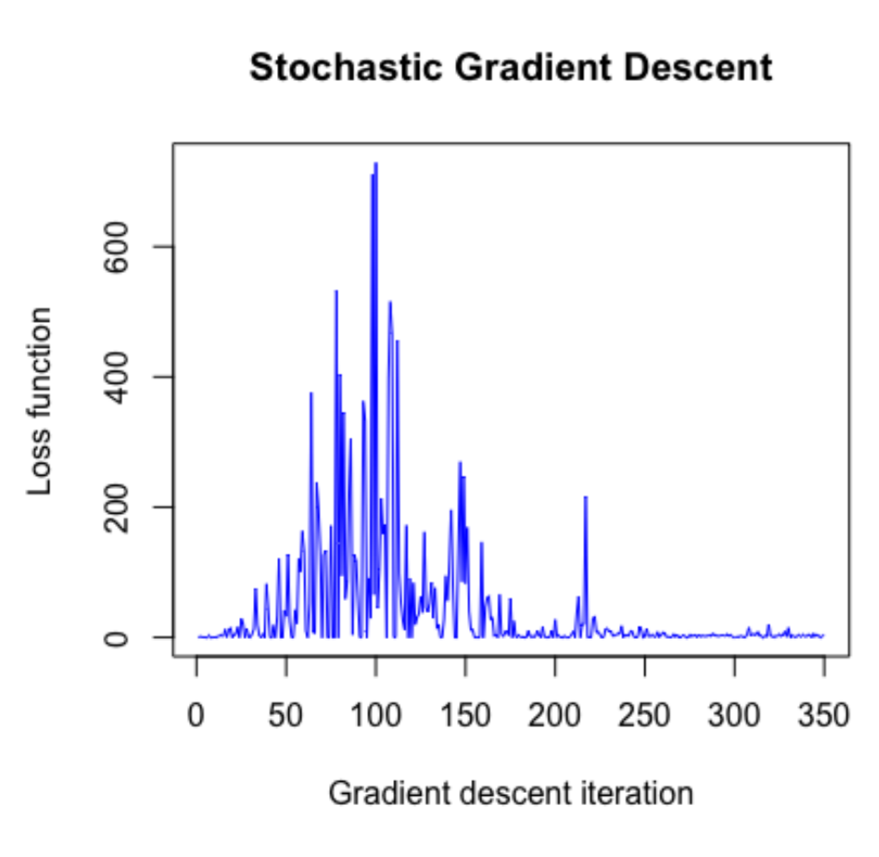
- 100* 60000 = 6 * 10^6 have been computed for the full batch algorithm,
- 100* 150 = 1500 have been computed for the mini batch algorithm.

It is a huge gain of gradient computation, however we pay the price of having less gradient to compute by a largest fluctuation due to highest variance as mini batch is a noisy approximation of the loss.

Reference (5) Keras code to obtain loss value per iteration

3.5 Stochastic Gradient Descent in Keras

Note that the so called **Stochastic Gradient Descent** is a particular case where m=1, using the later having a lot of noise but it reduces a lot time computation. In keras we implement **optimizer_sgd()** function to use it in our model. This function has several paramaters to optimize the precision and the speed, such as $momentum$ and $nesterov$. Below is a graph of convergence using these parameters.



3.5.1 Momentum

In **Stochastic Gradient Descent** we are easily get in wrong direction by processing the gradient descent algorithm because we are just taking one observation instead of all data in our gradient, so it becomes only one gradient. By this way, we tend to oscillate a lot to find the minimum, so there is a method to solve this problem, it is called momentum, this algorithm is defined this way (we pick (x_i, y_i) uniformly over the entire dataset for each iteration) :

$$\begin{aligned} v_{n+1} &= \gamma v_n + \eta \nabla_\theta l(\theta_n, x_i, y_i) \\ \theta_{n+1} &= \theta_n - v_n \end{aligned}$$

where v_n is an auxiliar sequence to calculate θ_n, γ is the momentum term usually set to 0.9 and η is the step of gradient descent.

3.5.2 Nesterov

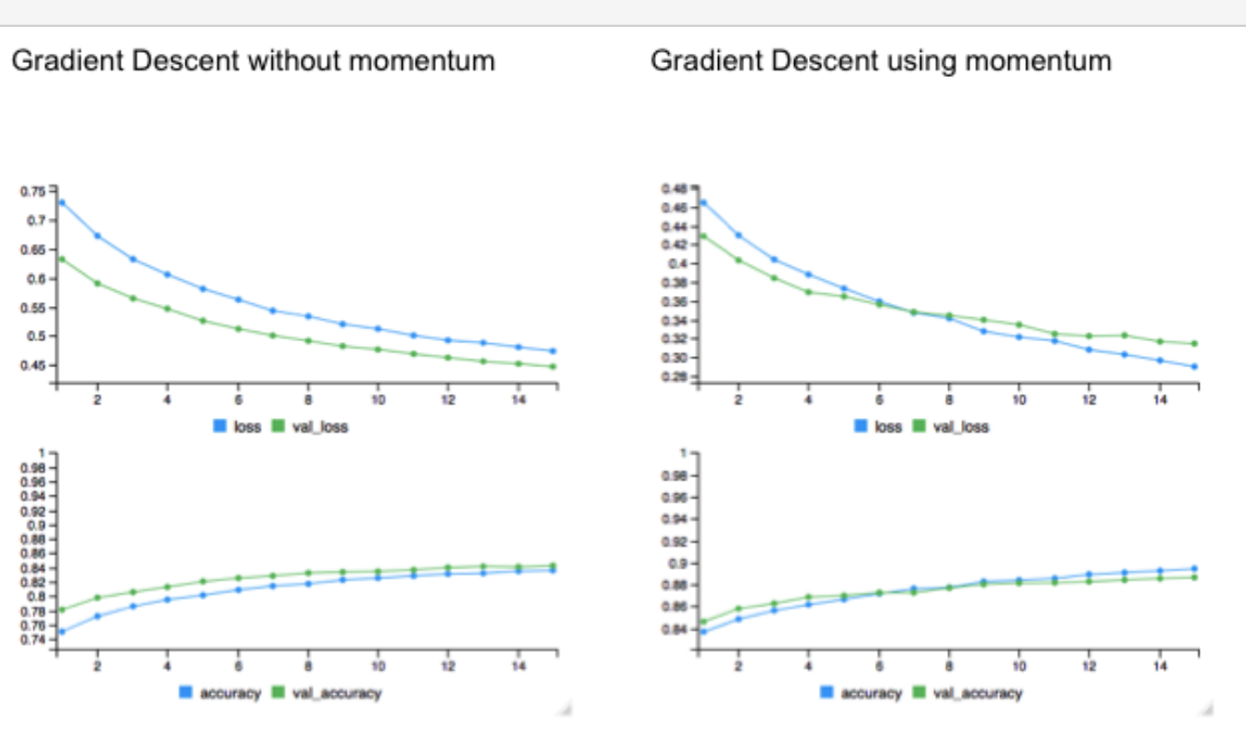
γv_n is our momentum term for updating the parameter. Nesterov accelerated gradient (NAG) is a way to correct our path while updating the new θ by anticipating approximatively the next step with θ_n - γv_n in the gradient. So before updating our θ_n we correct the anticipated term θ_n - γv_n in order to prevent a strong acceleration in wrong direction.

$$\begin{aligned} v_{n+1} &= \gamma v_n + \eta \nabla_\theta l(\theta_n - \gamma v_n, x_i, y_i) \\ \theta_{n+1} &= \theta_n - v_n \end{aligned}$$

Reference (2) From section '2.Gradient Descent Variant' of the article

Exploration with and without momentum and nesterov arguments.

```
p1 <- ggdraw() + draw_image("Desktop/without_momentum.png", scale = 0.9) + draw_figure_label('Gradient Descent wi
thout momentum')
p2 <- ggdraw() + draw_image("Desktop/with_momentum.png", scale = 0.9) +
draw_figure_label('Gradient Descent using momentum')
plot_grid(p1, p2)
```



We clearly seen a significant drop of the loss using momentum algorithm.

4 Gradient back propagation

Motivation : Once we know how to update θ, the last thing to do is to compute the gradient of one given observation $\nabla_\theta l(\theta, x_i, y_i)$. To do so we use the back propagation principle, the goal is to compute $\nabla_\theta l$ = $(\nabla_{W_1} l, \dots, \nabla_{W_J} l)^T$, we include biais in the matrix of weights.

Chain Rules : Recall the chain rules for composition function, $z(x)$ = $g(y(x)) \implies \nabla_x z$ = $(\frac{\partial y}{\partial x})^T \ \nabla_y z$, where $z : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^m \to \mathbb{R}$, $y : \mathbb{R}^n \to \mathbb{R}^m$ and $(\frac{\partial y}{\partial x})$ denote the Jacobian Matrix of y.

4.1 Back propagation principle

Then using $(*)$ equation and applying chain rules to $l(\nu_j(W_j x_{j-1} + b_j))$ for a given $j \in [1, \dots, J]$, we get the following equations :

$$\nabla_{x_{j-1}} l = (\frac{\partial \nu_j}{\partial x_{j-1}})^T \nabla_{x_j} l$$

$$\nabla_{W_j} l = (\frac{\partial \nu_j}{\partial W_j})^T \nabla_{x_j} l$$

From the two equation below, given $\nabla_{x_j} l$, we see that in order to obtain the gradient of the loss with respect to x_{j-1} and W_j we just need to multiply them by the Jacobian Matrix. Althought we are interested in computing $\nabla_{x_{j-1}} l$ we need it to obtain $\nabla_{W_{j-2}} l$ so that we can propagate the gradient. Let's compute these Jacobian Matrix :

$$\begin{aligned} \frac{\partial \nu_j}{\partial x_{j-1}} &= (\frac{\partial W_j x_{j-1}}{\partial x_{j-1}})^T . \ \frac{\partial \nu_j}{\partial W_j x_{j-1}} \\ &= W_j^T . \ Diag(\nu_j'(W_j x_{j-1} + b_j)) \end{aligned}$$

$$\begin{aligned} \frac{\partial \nu_j}{\partial W_j} &= (\frac{\partial W_j x_{j-1}}{\partial W_j})^T . \ \frac{\partial \nu_j}{\partial W_j x_{j-1}} \\ &= x_{j-1} . \ Diag(\nu_j'(W_j x_{j-1} + b_j)) \end{aligned}$$

where $Diag(\nu_j'(W_j x_{j-1} + b_j)$ is a diagonal matrix (Jacobian of ν).

Finally, from $(**)$ we get, $\nabla_{x_J} l$ = softmax(x_J) - y_i, hence we can compute $\nabla_{W_J} l$ and $\nabla_{x_{J-1}} l$ and so on for all j until $\nabla_{W_1} l$.

Reference (4) "Descente de gradient et rétro-propagation du gradient" course

5 Model testing

5.1 Training vs validation test

Every epoch, Keras compute the loss and accuracy over the training set, however we test our model on the data we used to build it, hence this could lead to overfitting : we predict well our training data but we are doing poorly on new data. To solve this issue and in order to decide when to stop training the model, thanks to validation_split (ranges from 0 to 1) argument Keras uses a pourcentage of the data only for testing the model and the other for training.

example with **validation_split** = 0.2

```
for i from 1 to nb_epoch:
shuffle(data)
training set = 80% of data
validation set = 20% of data
train the model on training set
compute loss and accuracy on training set and validation set
```

One can look for the number of epoch for which the model is doing better with the training set rather than with the validation test.

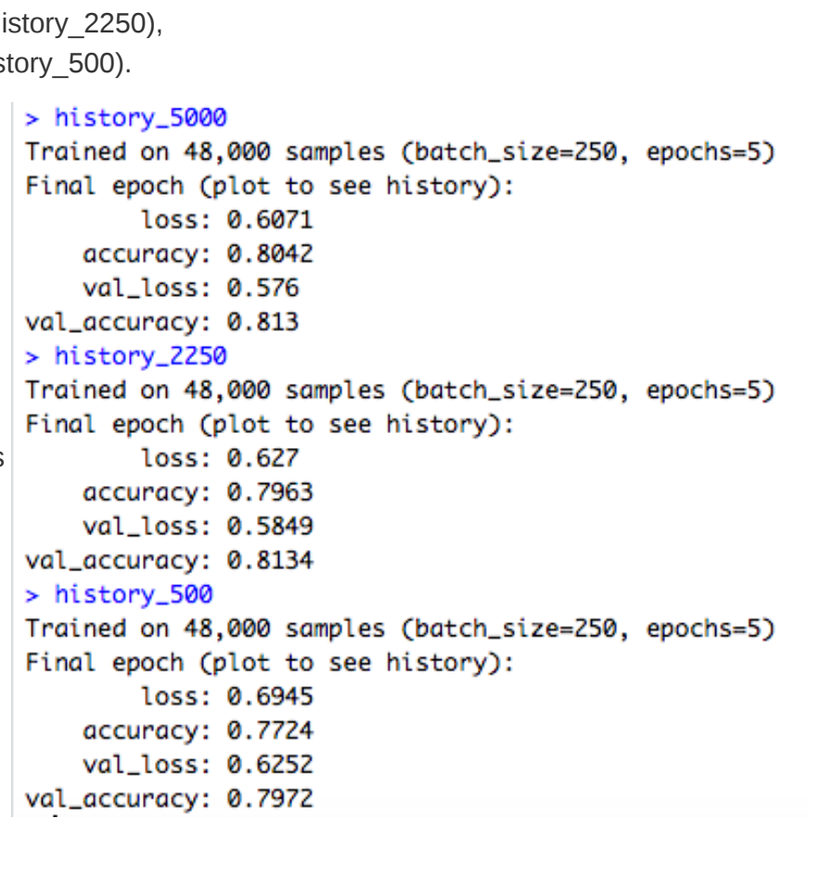
5.2 Effect of adding neurons on a layer

We can analyze the effect of adding neurons on a single layer, the model we use for testing is a single hidden layer with a dropout rate of 40%.

We test 3 models :

- Hidden layer with 5000 neurons (model name : history_5000),
- Hidden layer with 2250 neurons (model name : history_2250),
- Hidden layer with 500 neurons (model name : history_500).

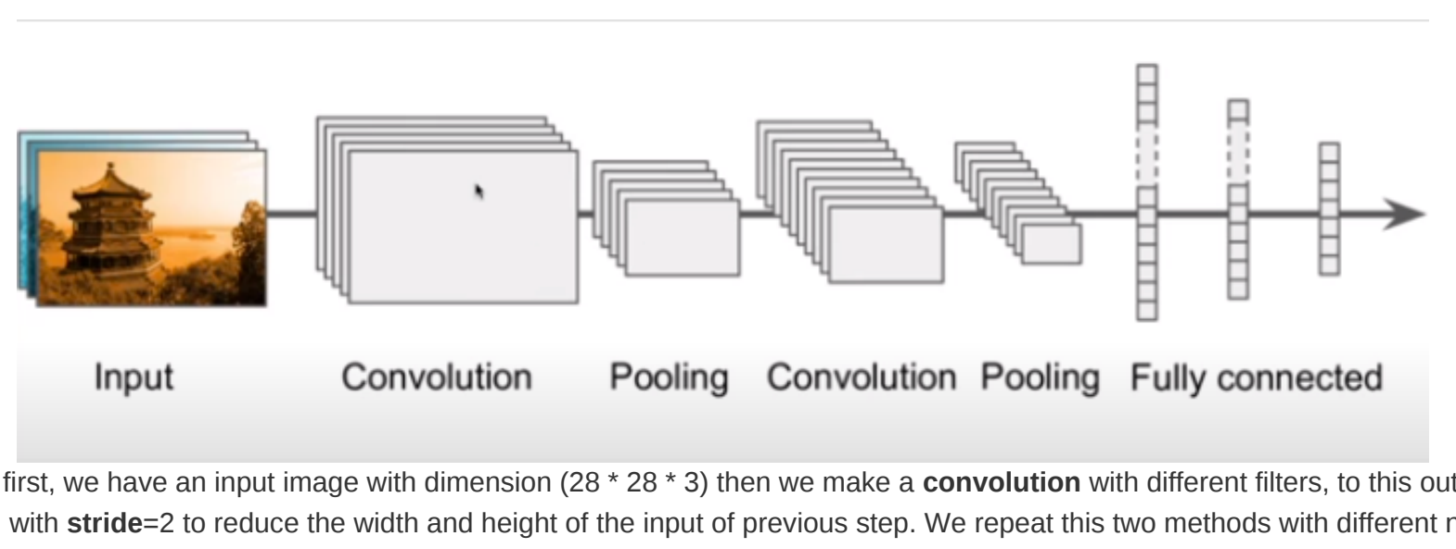
Performances over 3 models



```
> history_5000
Trained on 48,000 samples (batch_size=250, epochs=5)
Final epoch (plot to see history):
        loss: 0.6071
    accuracy: 0.8042
    val_loss: 0.576
val_accuracy: 0.813
> history_2250
Trained on 48,000 samples (batch_size=250, epochs=5)
Final epoch (plot to see history):
        loss: 0.627
    accuracy: 0.7963
    val_loss: 0.5849
val_accuracy: 0.8134
> history_500
Trained on 48,000 samples (batch_size=250, epochs=5)
Final epoch (plot to see history):
        loss: 0.6945
    accuracy: 0.7724
    val_loss: 0.6252
val_accuracy: 0.7972
```

6 Convolutional Neural Network (CNN)

Convolutional Neural Network is a method to classify the images into different classes by using several filters in order to find different significant characteristics in our image, the follow piture show us how a CNN works:



As we can see first, we have an input image with dimension (28 * 28 * 3) then we make a **convolution** with different filters, to this output we make a **max pooling** with **stride**=2 to reduce the width and height of the input of previous step. We repeat this two methods with different numbers of layers. At the end, we flat the final data in order to apply a classical neural network to it.

6.1 CNN framework

Convolution : In CNN, a convolution is a transformation from an input picture (of size nxn) to an output picture (of size mxm, m$ < n$) using a filter which is a square subpicture.

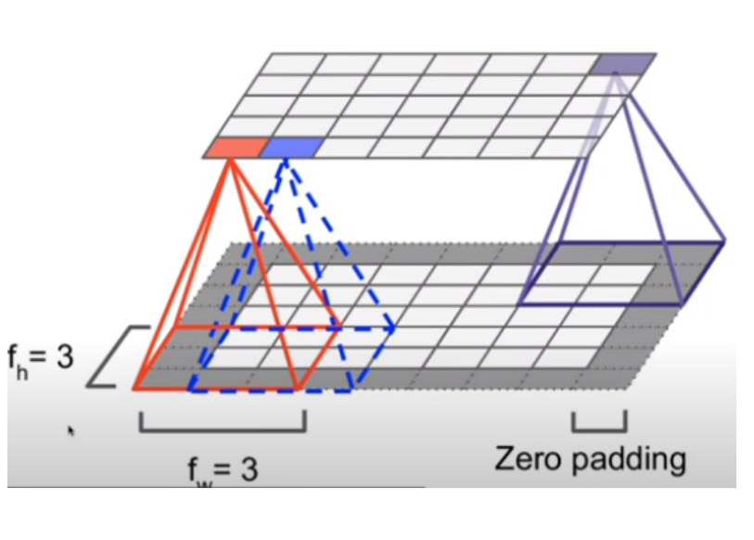
Filter : Subpicture of the input picture, each pixel of the filter corresponds to a weight. Applying a filter to the input picture consists in computing a linear transformation to all subpicture of the input picture, i.e $w^T x$ where w in the filter weights and x the vector of pixel of the subpicture.

Stride : Stride is the step we choose to shift the filter in order to select subpictures.If stride=2 we will divide the input dimension by 4 (divide by 2 in height and by 2 in width).

Padding : Add a contour to the input image in order to obtain an input and output image with the same dimension as when filtering an image we are loosing the contour of the picture.

Max pooling : Special filter with no weight, instead of calculating a linear combination of the subpicture we just take the max value.

In the picture below a convolution is shown, using **filter of size** 3x3, a **stride of** 1 and **zero padding**. By convention we decide that the output value $w^T x$ is the center of the filter.



Reference (6) Keras documention function for convolutional layer

Reference (7) Aurelien Geron GitHub

6.2 Convolution processing

In the following formula, we will introduce how we process a convolution with a 3 * 3 filter and without padding (so we lose the contour of the input picture). If we take an input picture of size n * n, the output will be of size $n-2$ * $n-2$.

The convolution can be seen as a mathematical convolotion G = $f * g$ where :

- $G : [|0,25|]^2 \to [0,1]$, $(i,j) \to G(i,j)$ represents the intensity of the pixel (i,j) of the output picture,
- $g : [|0,27|]^2 \to [0,1]$, $(i,j) \to g(i,j)$ represents the intensity of the pixel (i,j) of the input picture,
- $f : [|0,2|]^2 \to \mathbb{R}$, $(k,l) \to f(k,l)$ represents the weight of the filter.

$$\forall \ (i,j), \quad G(i,j) \ = (f * g)(i,j) = \sum_{k=0}^{2} \sum_{l=0}^{2} f(k,l) g(i+k, j+l)$$

In following formula, we will introduce how we process a convolution with a filter 3 * 3 using a padding.

$$\begin{aligned} \forall \ (i,j), \quad G(i,j) = (f * g)(i,j) \ &= \sum_{k=0}^{2} \sum_{l=0}^{2} f(k,l) g'(i+k, j+l) \\ where \ g'(i,j) &= \begin{cases} 0 & if \ i \in \{0, n-1\} \ or \ j \in \{0, n-1\} \\ g(i-1, j-1) & else \end{cases} \end{aligned}$$

g' represents the new input picture with the zero padding. One can also apply a non-linear function to the output G, for example for each pixel we apply a $Relu$ function.

6.3 CNN Hyperparameters

Although filter size is a choosen parameters by the user, weights and biais will be learned by gradient descent. Convolution are nothing but a classic neural network with less weights to be learned.

Convolutional layer : Batch of convolution using the same filter.

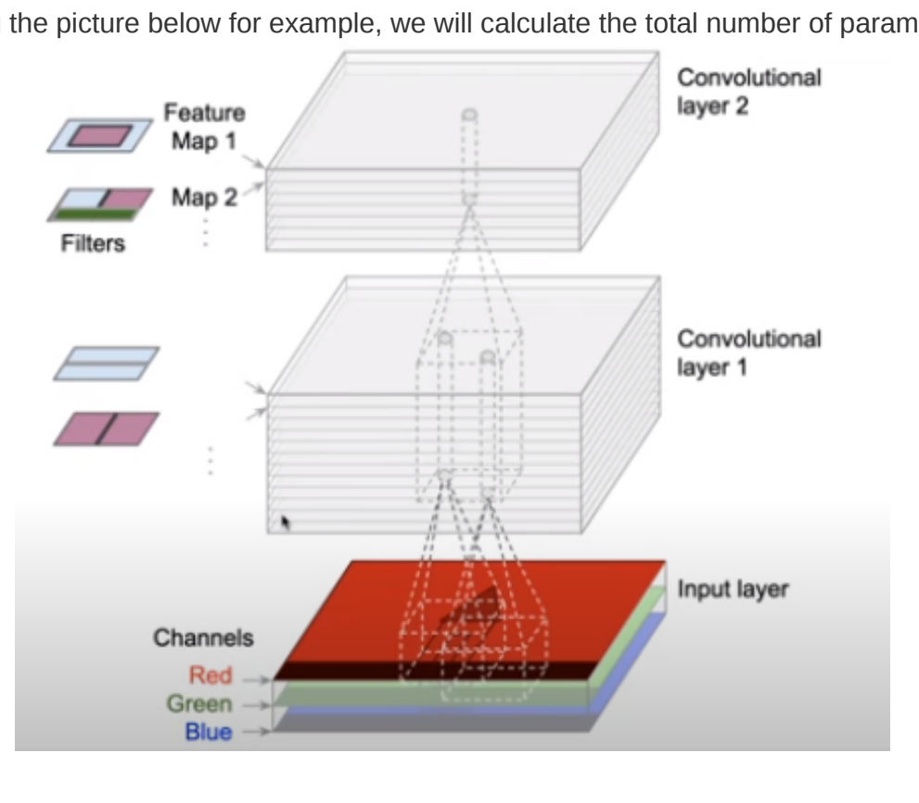
The total number of parameters to be learned by the cnn is given by :

$$\sum_{j=1}^{J} N_j * (n * n * N_{j-1} + 1)$$

where,

- j = $1, \dots, J$ denote the j-th convolution layer,
- N_j is the number of convolution used in the j-th layer (= number of filter), with N_0 is the depth of the input picture (e.g: 1 for a black and white picture and 3 for a RGB one),
- n x n is the filter size.

Taking the picture below for example, we will calculate the total number of parameters



We take a matrix of 3 * 3 for each filter, so each filter in the convolutional layer 1 should occupy 3 previous input layers, so we get (3 * 3 * 3) + 1 (bias) parameters, and there are 32 filters, so in convolutional layer, we get 32 * (3 * 3 * 3 + 1). Similarily in convolutional layer 2, every filter should occupy 32 previous layers, so (3 * 3 * 32 + 1) parameters, and we have 64 filters.

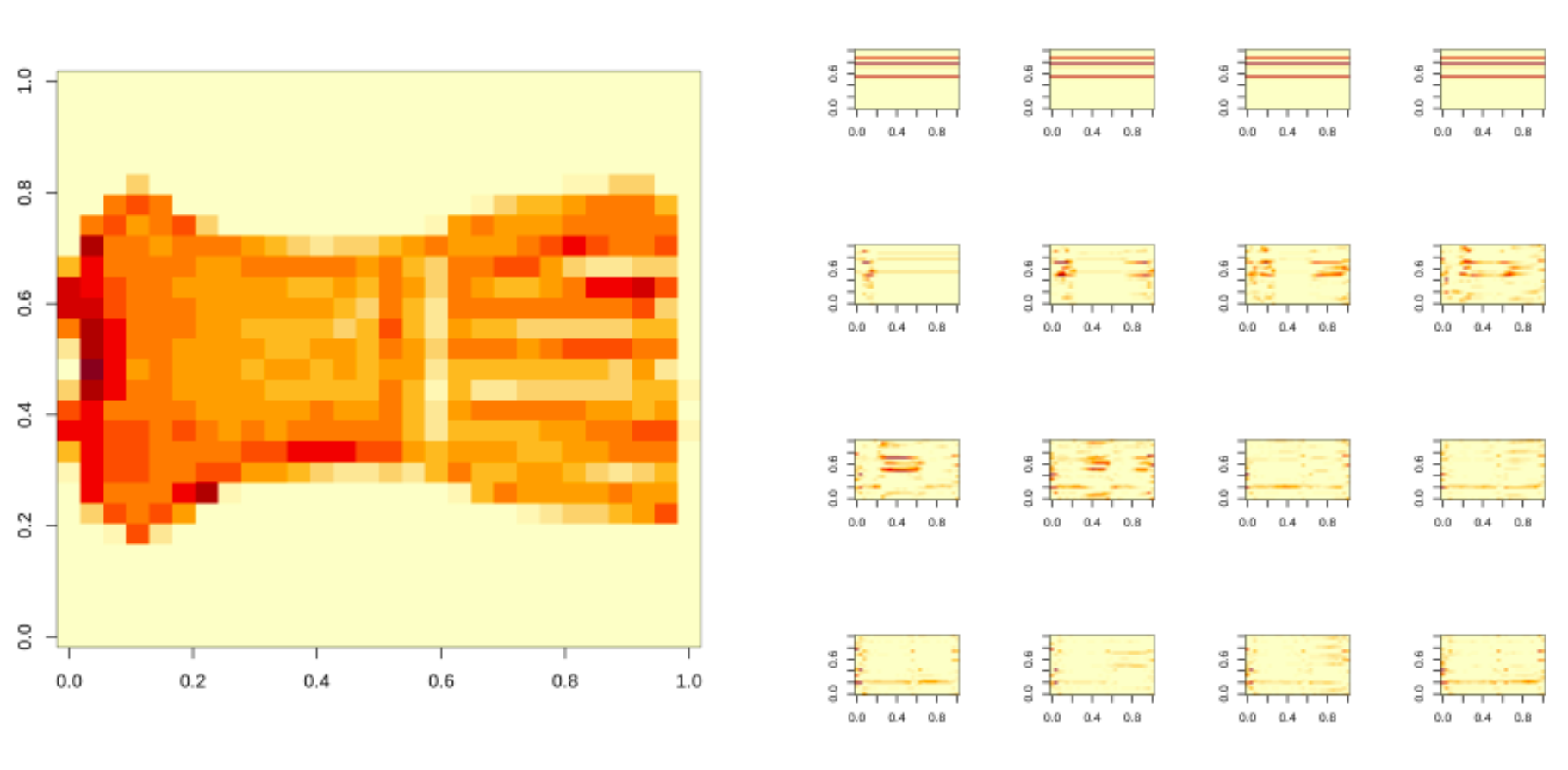
Finally,

Total number of parameters = 32 * (3 * 3 * 3 + 1) + 64 * (3 * 3 * 32 + 1) = 19392

6.4 Test case with a dress

Here we plot the first convolutional layer with 32 filters (we plot only 16 filters) of our model with a dress picture.

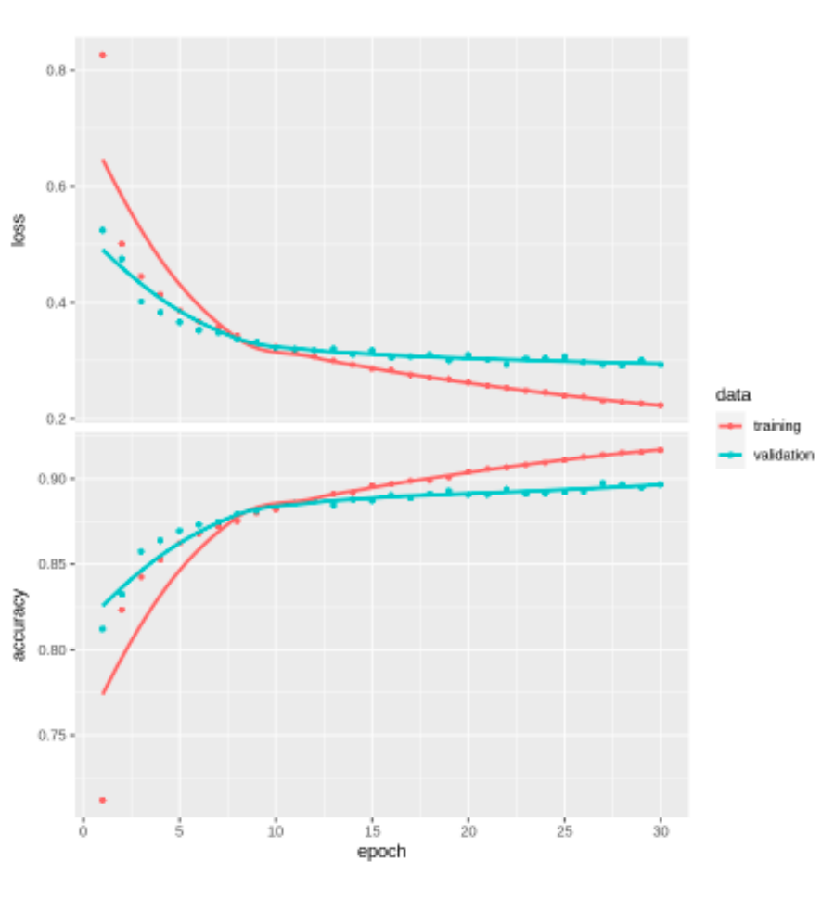
We can see than the first filters detect straight lines and the later detect details of the dress.



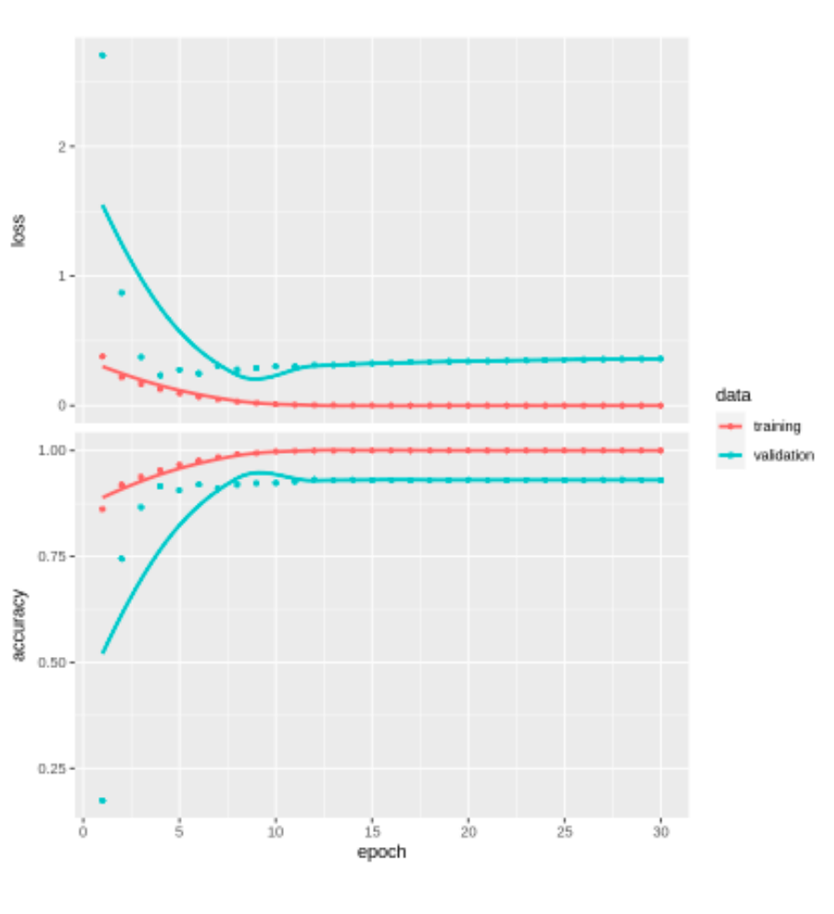
7 Comparaison between a Dense and Convolutional Neural Network

Please see the R Code to see the two different models used, there have been run under Google Colab notebook using GPU.

7.1 Dense Neural Network



7.2 Convolutional Neural Network



7.3 Loss and accuracy comparison

The first row are the result for the Dense network and the second for the convolutional.

loss:	0.31808140873909	accuracy:	0.890799999237061
loss:	0.39102429151535	accuracy:	0.925800025463104

8 References

(1) https://keras.rstudio.com/

(2) 'An overview of gradient descent optimization algorithms' https://arxiv.org/pdf/1609.04747.pdf

(3) https://www.college-de-france.fr/site/stephane-mallat/course-2019-03-20-11h15-video__1.htm

(4) https://www.college-de-france.fr/site/stephane-mallat/course-2019-03-20-09h30.htm

(5) https://keras.rstudio.com/articles/training_callbacks.html

(6) https://keras.rstudio.com/reference/layer_conv_2d.html

(7) https://github.com/amitanalyste/aurelienGeron

(8) https://github.com/ageron/handson-ml