2ID90 Deep learning assignment
template report

Group 98: Auguste Lefevre, Lucas Giordano

September 27, 2020

Contents

1 Introduction

Neural network are now used everywhere in all areas of industry especially for deep learning. Mostly used for pattern recognition (analysis of photos to recognize people or faces, recognition of fish swarms in sonar readings, recognition and classification of military vehicles in radar scans, or any number of other applications) Neural Networks can also be trained to recognize spoken language and hand written text or to control self driving cars etc. During this assignment our objective is to gain practical experience with basic aspects of *Deep Learning*, especially with deep artificial neural network (DNN). Artificial neural networks can be defined as a set of algorithms, modeled loosely after the human brain. They help us ro cluster and classify different types of data set. We will experiment different type of works we can do with DNN and also see with what kind methods and design structures can be used to improve results. DNN requires 3 main characteristics. First of all the choice of the model depending on the data representation. Secondly the learning algorithm and finally the robustness of the DNN. To obtain better results on our different experiments we will make vary hyperparameters (learningRate,batchSize and epochs) but we will also pre-processing the data using different methods and also try different optimisations as the Stochastic Gradient Descent method that could be used for finding a local minimum of a differentiable function.

1.1 Hyperparameters

During the all assignment we will speak a lot about 3 hypperparameters : learning rate, batch size and epochs. We decided to do a little introduction on those hyperparameters in order to not repeat ourselves in each experiment on what are the hyperparameters and what are their impact.

- Learning rate : Each time the model weights are update, the learning rate controls how much to change the model in response to estimated error. It could be very challenging to find the correct value for a specific experience. If you choose a value too small it may result in a long training process that could eventually be stuck at a certain type. If you choose a value too large it could result in learning a sub-optimal set of weights too fast or an unstable training process. So it is important to well choose the learning rate. In order to do our experiments we had to adapt our learning rate; we were able to investigate the effects of the learning rate on model performance and to build an intuition about the dynamics of the learning rate on model behavior using an article published by Machine Learning Mastery [2].

- Batch size : The batch size is a hyperparameter of gradient descent that controls the number of training samples to work through before the modelâs internal parameters are updated. We can see the batch as a for loop iterating over samples and making predictions. At the end of the batch (or the for loop) we compare the predictions to the expected output variables and error is then calculated. There are different names for the learning algorithms depending of the batchSize and the size of the training set :

  - Stochastic Gradient Descent: when batch Size = 1
  - Batch Gradient Descent: when batch Size = Size of Training Set
  - Mini-Batch Gradient Descent: when 1 < Batch Size < Size of Training Set

  In our case we are using the Mini-batch gradient descent method, and some popular batch size value to test are 16, 32, 64, 128. The batch size can have an big impact on the stability of the training of the neural networks, it also controls the accuracy of the estimate of the error gradient when training neural network. In order to do our experiments we used this article [1] to know the exact impact of the batch size on the neural network training. Another very interesting article on the subject that we use can be find here [8] . It helped us a lot in understanding the effect of the batch size on training dynamics

- Epochs : We can consider the epochs value as the number of rounds of optimization that are applied while doing the training of the model. To be precise this is a measure of the number of times all of the training vectors are used once to update the weights. Obviously, with a large value, the error on training data will reduce further and further but computation time will increase further and further. More over, if the epochs value is too large the model will over-fit the training set and will then loose in performance. But if you choose an epochs value too small then you will not train enough you model and the result will be an under-fitting model. There is no specific method to find the values of epochs, it depend of your satisfaction criterias (accuracy, error etc..). So you have to adjust the value in order to obtain the optimal one without over(under)-fitting the training set.

2 Regression

2.1 Implementation

In this part of the assignment we have to design a network that fits the data, choosing appropriate values for the hyperparameters *epochs, batchSize and learning rate*. Given this network we then have to experiment with

different combinations of *epochs and batchSize*.

The first question we will answer is how we did that. The idea was to first design different architectures by adding layers and by vary the number of neurons on layers. Then for each of these architectures we will run a triple for loop that try different combination of *bacthSize, epochs and learningRate* so we can compare results and find the best combination.

- Layer : We tested 6 different structures. One with 0 hidden layer, one with 1 hidden layer and so on until we have 5 hidden layers.
- Number of neuron : For each structure we decided to test different number of neuron on our layers. We tested with 5, 10 and 15 neurons on each layer.
- Hyperparameters : This was maybe the longest part. First of all we have done some research to find what are the 'common' values of our 3 hyper parameters in our type of regression. This is what we found :
  - BatchSize : [16,32,64,128,256]
  - epochs : [5,8,10,15,50,100]
  - learningRate : [0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]

  Using these 3 lists of values we decided to run the FunctionExperiement with all possible combinations, using a triple for loop algorithm (1) and saving values of the best solution:

Algorithm 1 Test all the hyperparameter combination and store the best one, Regression

```
// Run all combinatins of hyperparameters and save the best one
bestValidation := 0;
for each learningRate do
    for each batchSize do
        for each epochs do
            run FunctionExperiement(); //run the experiment with 1 combination
            if (currentValidation < bestValidation) then do //check if the current combination is the better one
                bestValidation := currentValidation; //refresh the best result
                store learningRate, batchSize and epochs of the current combination; //save parameter's values
}
```

2.2 Results

By testing our different structures and different combinations we obtained interesting result that we will now discuss. Remembering that we are here in the case of a regression validation using sqrt of mean squared - it calculates how accurate the prediction is for regression using root mean squared error, so in this case *lower is better*. As asked before we have to find the best design but also the best values for the hyperparameters. As described before we tested different types of architectures and iterate over different values for the hyperparamaters so it tests all possible combinations. After doing this we found that the best configuration is the following one :

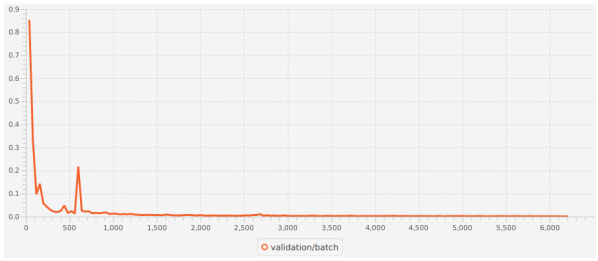
- bacthSize : 16
- epochs : 100
- learningRate : 0.05
- additional hidden layers : 5
- neurons per hidden layers : 10

and with this configuration we obtained a validation ratio equal to 2.101E-4 which is very low as wanted.

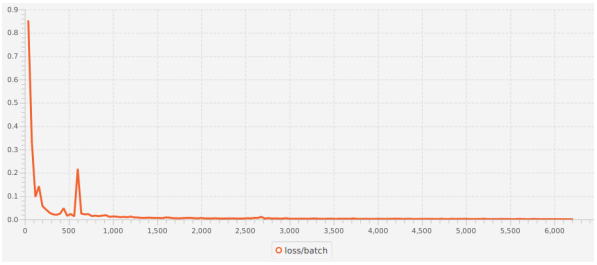
```
.#########################################################################################
BEST VALUES : batch : 16, epoch : 100, learningRate : 0.05 and validation : 2.1011101E-4
```

Figure 1: Result of the 3-for loop algo (1) to determined the best combination of hyperparameters

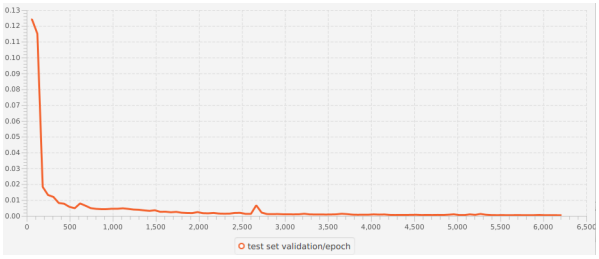
And in this configuration these are the graphs we obtained [figure 2] :



(a) **validation/batch**



(b) **loss/batch**



(c) **test set validation/epoch**

Figure 2: Result of the 1st experiment in the optimal configuration

As wanted, these curves are decreasing until reaching a really small value en then stay approximately constant around y = 0 (no oscillation). Finally for more information, here is the final status of the training [figure 3]:

epoch	#batches	#samples	#samples/sec	smooth loss
100	6200	9.920000e+04	1.336112e+04	5.481953e-04

Figure 3: Final status of the 1st experiment with nbr of epoch, nbr of processed batches, nbr of processed samples, nbr of sample/second and the smooth loss

In short, and in conclusion of this experiment, we managed to design a network architecture, to find the best optimization method and to tune our hyperparamters in order to get high validation/batch, low loss/batch and high test set validation/epochs. You can find the final configuration on table 1.

Parameter	**Value**
BatchSize	16
Epochs	100
LearningRate	0.05
Optimizer	Gradient Descent
Design	Fully Connected with 5 hidden layers, MSE (loss function)
Nbr of Neurons per layer	10

Table 1: Final configuration for the Function Experiment

3 Classification: fashion-MNIST

3.1 Implementation

In this experiment we are going to look at the fashion-MNIST dataset which contains labaled gray-scale images of 28x28 pixels. There are 60,000 training images and 10,000 validation images. Each image can be label by one of these 10 possibilities: 0: T-shirt/top, 1: Trouser, 2: Pullover, 3: Dress, 4: Coat, 5: Sandal, 6: Shirt, 7: Sneaker, 8: Bag, and 9: Ankle boot. By following the .pdf we directly obtained a functional experiment. We only had to find the TensorShape but it was obviously 28,28,1 since we are using 28x28 images with depth 1 (gray scale). For this experiment the .pdf also told us to use exclusively an epochs value of 5. But to obtain better results, 3 steps have been done and should be describes. First we optimize the hyperparameters *learning Rate and batchSize*. Then we also try to improve our result by pre-processing the data using the *Mean Subtraction* method. Finally we also used a variation of the gradient descent, the *Gradient Descent with Momentum*. Most of our research come from the lecture note of the CS231 Stanford University course [4].

- Optimization of the hyperparameters : Here we used exactly the same idea than for the 1st experiment. We developed a double for loop algorithm (2) to test each combinations of our hyperparameters *batchSize and learning Rate* with a fixed value of epochs (=5). Conversely to the 1st experiment we decided to test the following values only (since this experiment needs a lot of computation power) :
  - batchSize : [16,32,64]
  - learningRate : [0.1, 0.05, 0.01, 0.005, 0.001]

Algorithm 2 Testing all hyperparameter combinations and store the best one.

```
// Run all combinatins of hyperparameters and save the best one
bestValidation := 0;
for each learningRate do
    for each batchSize do
        run ZalandoExperiment(); //run the experiment with 1 combination
        if (currentValidation > bestValidation) then do //check if the current combination is the better one
            bestValidation := currentValidation; //refresh the best result
            store learningRate and batchSize  of the current combination; //save parameter's values
}
```

- Mean Subtraction : As described before, one way to improve our result is to pre-processing the data set. To do so one method is use : the mean subtraction method. Pre-processing is one of the most important step in machine learning project. It allow us to prepare and filter the data in order to analyse the data set in an easy way. Mean subtraction method involves subtracting the mean across every individual feature on the data, and has a geometric interpretation of centering the cloud of data around the origin along every dimension. The algorithm is then divide into two parts, first we compute the mean of the data set [algorithm 3], then we subtract this mean to each individual feature on the data set [algorithm 5].

Algorithm 3 Compute the mean of the data.

```
int computeMean(Data) {
    mean := 0;
    for each individual feature A on Data do
        mean := mean + mean(A);
    mean := mean / data.size;
    return mean;
}
```

Algorithm 4 Subtract the mean to every individual features on the data.

```
void subtractMean(Data) {
    mean := algorithm\ref{alg:meancompute}(Data) ;
    for each individual feature A on Data do
        A := A - mean;
}
```

- Gradient Descent with Momentum : As said before there are several variation of gradient descent method. Gradient descent with momentum is one of them. The momentum update is another approach that usually leads to a better converge rate on deep networks. The name momentum come from an analogy to momentum in physics. The weight vector w, thought of as a particle traveling through parameter space, incurs acceleration from the gradient of the loss. Unlike in classical stochastic gradient descent, it tends to keep traveling in the same direction, preventing oscillation. From the lecture note of CS231n we can obtained more information on the gradient descent momentum variation. Momentum update :
  - integrate velocity :
$$v = mu \times v - learningRate \times dx \tag{1}$$
  - integrate position :
$$x = x + v \tag{2}$$

  In this version v should be initialize to 0 and an additional hyperparameter is used : *mu*. This parameter usually take a value from this list : [0.5, 0.9, 0.95, 0.99]. Here is the pseudo code of our update function for the gradient descent with momentum :

Algorithm 5 Implementation of the Gradient Descet with Momentum method.

```
mu := 0.9;
INDArray toUpdate;

void update(array, isBias, learningRate, batchSize, gradient) {
    factor := -(learningRate/batchsize); // compute the factor
    if (toUpdate == null) do update = 0; // we initialize our array to 0
    toUpdate := toUpdate * mu + gradient * factor; //velocity
    array := array + toUpdate; //position
}
```

3.2 Results

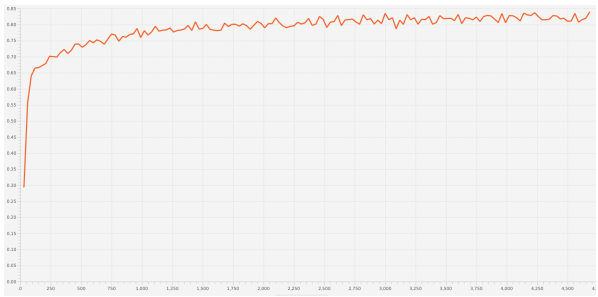
Now that we described how we implement our different method and optimization we will observe the results. Remembering that in this case we are doing a classification so we want loss/bach low, validation/bacth high and test set validation/epoch high. First of all we were asked to determined the optimal value for 2 of our 3 hyperparameter : *learning Rate and batchSize*. As described before we run the 2-for loop algorithm (2) to find the best combination and we obtained this configuration :

- batchsize : 64
- epochs (fixed) : 5
- learning rate : 0.01

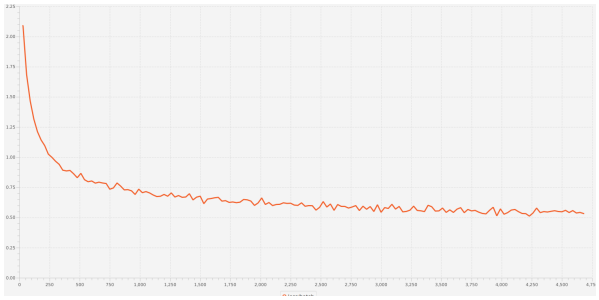
```
.############################################
BEST VALUES : batch : 64, learningRate : 0.01
```

Figure 4: Result of the 2-for loop (2) to determined the best combination of hyperparameters on Zalando experiment

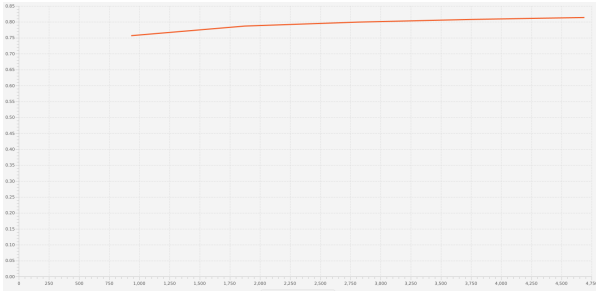
Using this configuration without other optimization give us an accuracy of *81 %*, these are the exact results obtained [figure 5]:



(a) **validation/batch**



(b) **loss/batch**

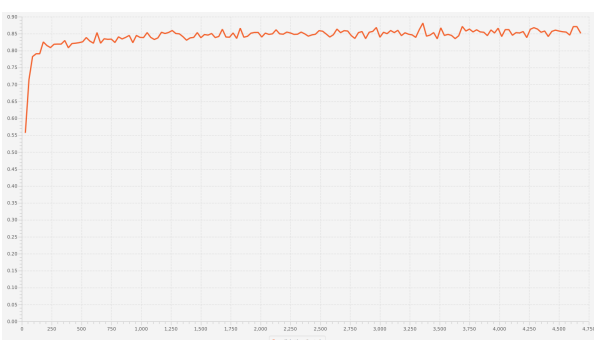


(c) **test set validation/epoch**

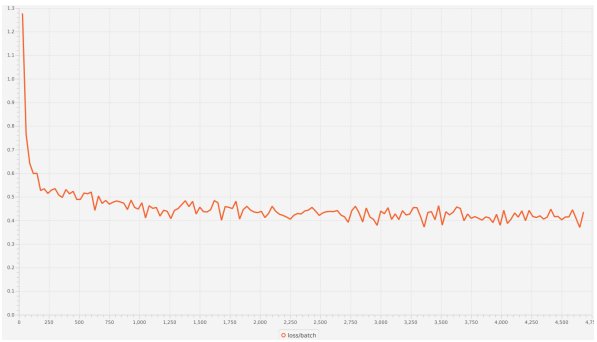
Figure 5: Result of the 2nd experiment in the optimal configuration without optimization

Even if these results are already satisfying, it is asked to try to improve our results using the 2 methods described before : *mean subtracting and gradient descent with momemtum*. We expect to improve our result using these 2 methods so we run again the experiment but this time we pre-processed the data using *mean subtracting* and modify the classic gradient descent method to use momentum instead. After running the experiment again we obtained a validation rate of 84 % which is 3 % higher than without these optimization. We can say that we

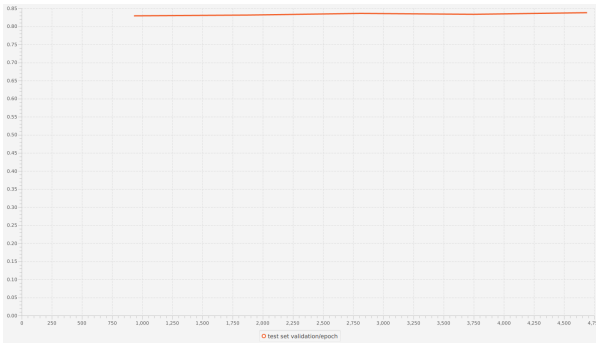
improved our result and that theses two methods are efficient on this experiment. These are the final results [figure 6 and 9]:



(a) **validation/batch**



(b) **loss/batch**



(c) **test set validation/epoch**

Figure 6: Result of the 2nd experiment in the optimal configuration and optimization (Mean Subtraction and Gradient Descent with Momentum)

epoch	#batches	#samples	#samples/sec	smooth loss
5	4685	2.998400e+05	2.115851e+04	4.133310e-01

Figure 7: Final status of the 2nd experiment with nbr of epoch, nbr of processed batches, nbr of processed samples, nbr of sample/second and the smooth loss

In short, and in conclusion of this experiment, we managed to design a network architecture, to find the best optimization method and to tune our hyperparamters in order to get high validation/batch, low loss/batch and high test set validation/epochs. You can find the final configuration on table 2.

Parameter	**Value**
BatchSize	64
Epochs	5
LearningRate	0.01
Optimizer	Gradient Descent With Momentum
Pre-processing	Mean Subtraction
Design	Fully Connected with Flatten layer, Cross entropy (loss function)

Table 2: Final configuration for the Zalando Experiment

4 Classification: Square, Circle, Triangle

4.1 Implementation

In this part of the assignment we will also try to classify our data as in the 2nd experiment but this time we have a data set of 28x28 gray scale image containing either a square, a circle or a triangle. Our goal is of course to classify these images. We will divided the work in 4 step. First we will run the ZalandoExperiment model on

it and we will try to tune the hyperparameters to obtained the best result possible. Secondly we will improve our performance by designing a new network architecture using convolutional layers possibly combined with pooling layers. Thirdly we will implement the *L2 weight decay* as described in the CS231n course note [4] to make a custom *UpdateFunction* that combine gradient descent update with L2 weight decay. Finally we will implement another gradient descent alternative, the *Adadelta* method. To implement this last method we will use the paper [9] of Matthew D. Zeiler. Before presenting the result we will explain how did we implements those different step.

- Apply Zalando Network and tune parameter : Using the network of the Zalando Network the idea is to apply the new data set and tunes the parameters to determine the best combination. To do that we re-run the 2-for loop algorithm (2) on the ZalandoExperiment Class but this time on the SCT data set. There was nothing to change as image are also 28x28 and with depth 1, so we wont go deeper in the explanation and we will discuss result later. We decided to test the following values only (since this experiment needs a lot of computation power) :
  - batchSize : [16,32,64]
  - learningRate : [0.1, 0.05, 0.01, 0.005, 0.001]
- Design a network : The second step of this experiment is to design a network that fit better the data set. The ides is to use convolutional layers combined with pooling layer. It is said that our convolutional layers have stride 2, and zero-padding such that width and height of output images equal those of the input images. It limits the choice of the kernel size. For the pooling layer the stride has to be a divisor of the image size, so 2 is a good stride value. To implement this step we used the network of the ZalandoExperiment and we added our new layers and change the classic output layer into a *OutputSoftmax* layer. When using a Convolutional Neural Network (CNN) we have to use new hyperparameters : kernel and stride. Most of the information we used to find the correct TensorShape and the correct design are described in the CS231 lecture note on CNN [3]. This is the design we implemented :

```
############################  MODEL ##############################
(1) InputLayer      : {name=In, input shape=(1,1,28,28), output shape=(1,1,28,28)}
(2) Convolution2D   : {name=conv1, input shape=(1,1,28,28), output shape=(1,10,28,28), activation=RELU, kernel size=3, kernels=10}
(3) PoolMax2D       : {name=poolMax1, input shape=(1,10,28,28), output shape=(1,10,14,14), stride=2}
(4) Flatten         : {name=Flatten, input shape=(1,10,14,14), output shape=(1,1960)}
(5) OutputSoftmax   : {name=Out, input shape=(1,1960), output shape=(1,3), activation=Softmax}
```

Figure 8: Design of the Convolutional Neural Network *(system.out.println(model))*

- L2 Weight Decay : The L2 weight decay, also called L2 regularization, is one of the most form of regularization. The main objective of a regularization is to controlling the capacity of Neural Network to prevent over-fitting. The idea behind this method is to penalizing the squared magnitude of all parameters directly in the objective. This type of regularization usually prefer diffuse weight vectors over peaky weigh vectors. In our case we have to combined gradient descent with the L2 weight decay method. By reading the lecture note of the course CS231 [4], we can find that a way to combined those methods is to create an update function that update the weight (every weight is decayed lineraly) as follow :

$$W = W - lambda * W \tag{3}$$

where lambda is the regularization strength, sometimes call decay and usually equal to 0.001 but it can be 10, 100 times lower. Then after this we can simply call a classical gradient descent update function. We decided to implement this equation in our project and so create a new *UpdateFunction : L2Decay*. Here is the pseudo code for the update method of this *UpdateFunction* :

Algorithm 6 L2 Decay method combined with gradient descent.

```
1  double dacay := 0.001;
   UpdateFunction f; // from the supplier, given in the pdf
3
   void update(array, isBias, learningRate, batchSize, gradient) {
5      array := array - decay*array; \\ w += -decay*w
       f.update(array, isBias, learningRate, batchSize, gradient); // w += -grad(W)
7
   }
```

- Adadelta : The last optimization is a gradient descent alternative call *Adadelta*. The main advantage of this alternative is that it automatically chooses the learning rate. As said before, most of the info we got on this alternative come from the paper [9] of Matthew D. Zeiler. The adadelta method dynamicaly

adpats over time using first order information and as said before it does not need to manually tuning the learning rate. This method is supposed to show promising results when apply on the MNIST digit class. Instead of re-writing the full pseudo code we decided to add the pseudo code of the paper as we followed exactly this pseudo code to write our new *UpdateFunction* for the adadelta alternatives.

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ
Require: Initial parameter x_1
Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
for $t = 1 : T$ **do** %% Loop over # of updates
  Compute Gradient: g_t
  Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2$
  Compute Update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t$
  Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$
  Apply Update: $x_{t+1} = x_t + \Delta x_t$
end for

Figure 9: Pseudo-code for the Adadelta update function

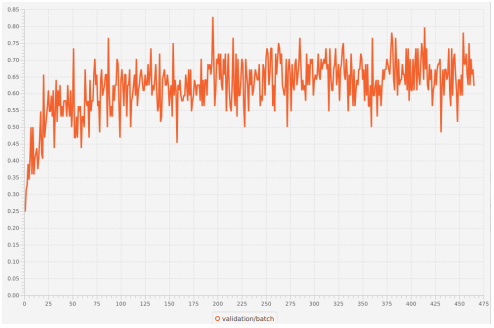
4.2 Results

From those 4 steps we got different results that we will now discuss. We will proceed step by step and observe the result. The first step is when we use the Zalando network and perform a 2-for loop algorithm (2)to find the best hyperparameters.

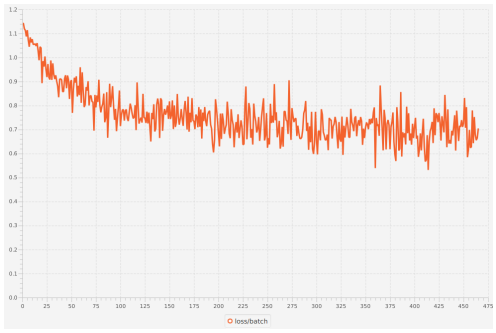
```
.##########################################################################################
BEST VALUES : batch : 64, learningRate : 0.01 and validation : 0.611
```

Figure 10: Result of the 2-for loop algorithm to determine the hyperparamater values for SCT Experiment

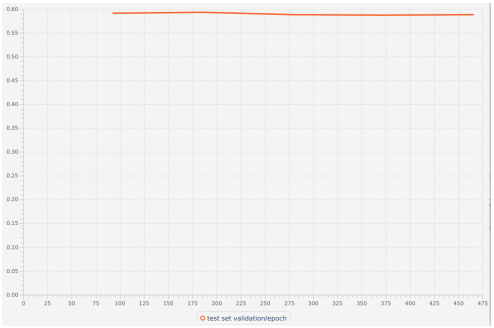
On figure 10 we can observe that the best values are : for the batchSize 64 and 0.01 for the learning rate (with a fixed number of epochs equal to 5). But even with those parameters, results are not satisfying as a the test set validation is around 60 % accuracy only and as you can see on the figure bellow the validation and loss per batch oscillate a lot and didn't converge to an optimal value. These are the exact results with the parameters cited before :



(a) **validation/batch**



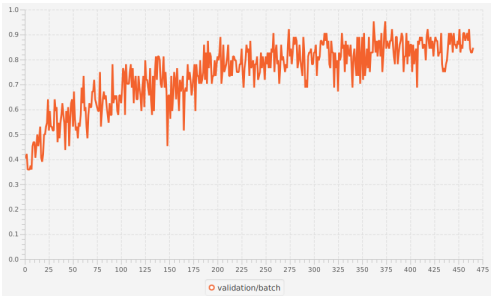
(b) **loss/batch**



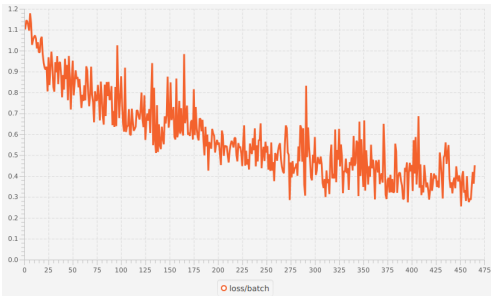
(c) **test set validation/epoch**

Figure 11: Result of the 3rd experiment apply on the Zalando network with optimal hyperparameters

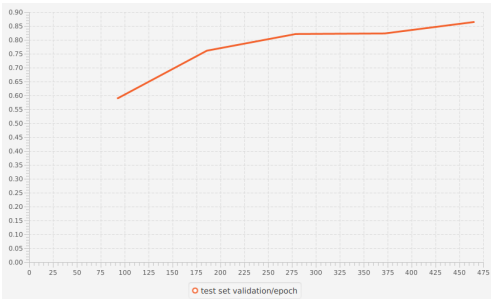
We will now change the network design to implement a CNN and apply the 3rd experiment on this network. We use the hyperparameter values found before. We expect to obtain a better accuracy but validation and loss function should probably stay unstable and result should not be satisfying enough since we didn't change the *UpdateFunction* yet (we still use the gradient descent with momentum and a pre-processing on the data using the mean subtraction method). These are the results obtained [figure 12]:



(a) **validation/batch**



(b) **loss/batch**

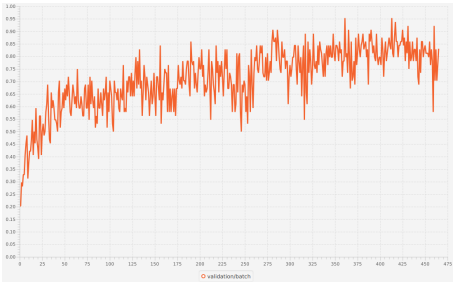


(c) **test set validation/epoch**

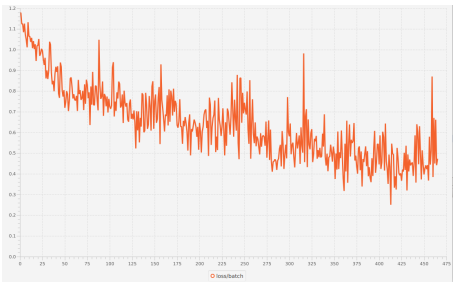
Figure 12: Result of the 3rd experiment apply on a CNN with optimal hyperparameters

By observing those result we can see that the CNN is way more efficient in this experiment because as expected the test set validation per epoch is way higher than with the Zalando network (around 81% against 60%). But the validation and loss per batch is still oscillating a lot a do not converge clearly.

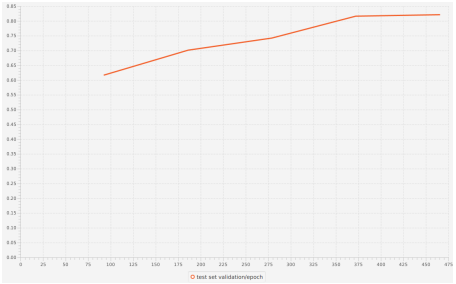
For the first step we will now re-use the CNN but this time we will use our new *UpdateFunction* which is a combination of the L2 weight decay method and the gradient descent one. This methods is suppose to prevent over fitting and then should improve our result or at least create robustness against over fitting. These are the result [figure 13 obtained with the new *UpdateFunction* :



(a) **validation/batch**



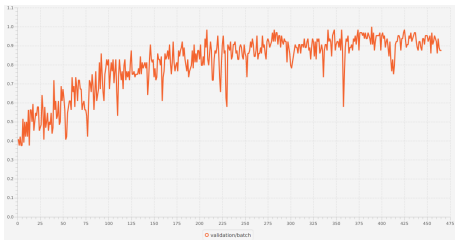
(b) **loss/batch**



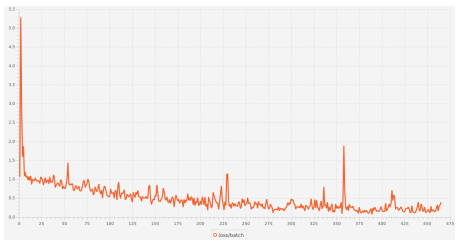
(c) **test set validation/epoch**

Figure 13: Result of the 3rd experiment apply on a CNN with optimal hyperparameters and L2Decay update function

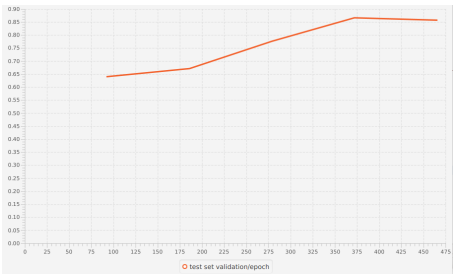
Finally the last step is also another *UpdateFunction*, but this time it is said that this is the best one for this type of data. So For this final test we use the optimal parameter found in step 1, the CNN use in step 2 and finally we change the classic gradient descent method by the Adadelta alternative. We now expect to obtain our best result and also to have loss and validation per batch with less oscillation and better result. These are the results [figure 14] :



(a) **validation/batch**



(b) **loss/batch**



(c) **test set validation/epoch**

Figure 14: Result of the 3rd experiment apply on a CNN with optimal hyperparameters and Adadelta update function

By looking closely at these results we can observe that we reached an accuracy on the test set validation/echos greater than 85 %, which is a good result. We can also observe that both validation and loss per batch oscillate less and converge better to their final value. In short, and in conclusion of this experiment, we managed to design a network architecture, to find the best optimization method and to tune our hyperparamters in order to get high validation/batch, low loss/batch and high test set validation/epochs. You can find the final configuration on table 3

Parameter	**Value**
BatchSize	64
Epochs	5
LearningRate	0.01
Optimizer	Adadelta
Pre-processing	Mean Subtraction
Design	CNN-1block
KernelSize, kernels and stride	3 , 10 and 2

Table 3: Final Configuration for the SCT Experiment

5 Convolution: CIFAR10

5.1 Implementation

The final experiment of this assignment is the CIFAR10 experiment. CIFAR10 data set contains 60,000 images (50000 training and 10,000 test images) of 32x32 pixel in RGB. There is normally 10 different labels for those images but for the exercise we will only use 5 different labels. Our goal is to classify correctly the different images depending of their labels. To to so the first step is to inspect the data and normalize it. Then we still have two major thing to do :

- 1- Design a network for the CIFAR10 data set
- 2- Tune : parameters layers, layer sizes, etcetera; targeting +70% accuracy

Optionally we can implement new type of layer (DropOut layer) or the Batch Normalization method. In order to complete this assignment we have to explain the implementation and the result of point 1 and 2. Our work can be divided in two step. First of all we decided to re-use all the design an parameter we already implement before and create a 5-for loop that iterate over 5 different parameter :

- batchSize [16, 32, 64, 128]
- epochs [3,5,8,10]
- learningRate [0.09, 0.009, 0.0009, 0.00009]
- network design [Flatten layer (Zalando Experiment), CNN (SCT Experiment)]
- updateFunction [Gradient Descent with Momentum, L2Decay, Adadelta]

The pseudo code of the 5-for loop is the following :

Algorithm 7 Test all possible combinations and store the best one, 5-for loop, CIFAR10.

```
// Run all combinations possible
bestValidation := 0;
for each learningRate do
    for each batchSize do
        for each epochs
            for each design
                for each updateFunction
                    run CIFAR10(); //run the experiment with 1 combination
                    //check the accuracy of this solution compared to the best one found
                    if(currenValidation() > bestValidation) then do
                        //If this combination is the best, store this solution
                        bestValidation := currentValidation
                        store learngingRate, batchSize, epochs, design and updateFunction
}
```

For the first time of the assignment we are using RGB images, then we had to modify the mean subtraction method and adapt it to the CIFAR10 experiment. The idea is exactly the same than for a gray-scale image but this time we compute means of each color channel and subtract those means for each futures on the data. So basically we just repeat what we did for the classical mean subtraction method but this time we do it on the 3 color channel R,G and B. The algorithm is then divided into two parts, first we compute means of the data set [algorithm 8], then we subtract those mean to each individual feature on the data set [algorithm 9].

Algorithm 8 Compute the mean of the data for RGB images.

```
int[] computeMean(Data) {
    meanRed, meanGree, meanBlue := 0;
    for each individual feature A on Data do
        meanRed := meanRed + (A.red);
        meanGreen := meanGreen + (A.green);
        meanBlue := meanBlue + (A.blue);

    meanRed := meanRed / data.size;
    meanGreen := meanGreen / data.size;
    meanBlue := meanBlue / data.size;
    return [meanRed, meanGreen, meanBlue];
}
```

Algorithm 9 Subtract the mean to every individual features on the data for RGB images.

```
void subtractMean(Data) {
    mean[] := algorithm\ref{alg:meancomputeRGB}(Data) ;
    for each individual feature A on Data do
        A.red := A.red - mean[0]; //mean[0] = meanRed
        A.green := A.green - mean[1]; //mean[1] = meanGreen
        A.blue := A.blue - mean[2]; //mean[2] = meanBlue
}
```

Now that we have all what we need we can run the 5-for loop algorithm (7) to obtain our result. Unfortunately, this method was very long and the program run for more than 24h. Moreover, as you will see in the *Results* subsection, results obtained were not sufficient so we had to improve our model. We also used the excessive running time to do some research on the CIFAR10 experiment in order to find some clues of what type of networks is used, which update function is the best, which values for the hypperparamater etcetera. The CIFAR10 problem is a well known experiment and is very well documented through the web. By combining our 1st experiment and some research ([5], [7] and especially, [6]) we decided to change our network design. In fact

we already have tuned hyperparameters without enough success so we assume we had to modify our design. We decide to construct a 3 block CNN, as it seems to be the most popular network design use for CIFAR10 experiment. You can find the details of the design implementation on figure 15

```
################################  MODEL ###################################
(1) InputLayer      : {name=In, input shape=(1,3,32,32), output shape=(1,3,32,32)}
(2) Convolution2D   : {name=conv1, input shape=(1,3,32,32), output shape=(1,32,32,32), activation=RELU, kernel size=3, kernels=32}
(3) Convolution2D   : {name=conv2, input shape=(1,32,32,32), output shape=(1,32,32,32), activation=RELU, kernel size=3, kernels=32}
(4) PoolMax2D       : {name=poolMax1, input shape=(1,32,32,32), output shape=(1,32,16,16), stride=2}
(5) Convolution2D   : {name=conv3, input shape=(1,32,16,16), output shape=(1,64,16,16), activation=RELU, kernel size=3, kernels=64}
(6) Convolution2D   : {name=conv4, input shape=(1,64,16,16), output shape=(1,64,16,16), activation=RELU, kernel size=3, kernels=64}
(7) PoolMax2D       : {name=poolMax2, input shape=(1,64,16,16), output shape=(1,64,8,8), stride=2}
(8) Convolution2D   : {name=conv5, input shape=(1,64,8,8), output shape=(1,64,8,8), activation=RELU, kernel size=3, kernels=64}
(9) Convolution2D   : {name=conv6, input shape=(1,64,8,8), output shape=(1,64,8,8), activation=RELU, kernel size=3, kernels=64}
(10) PoolMax2D      : {name=poolMax3, input shape=(1,64,8,8), output shape=(1,64,4,4), stride=2}
(11) Flatten        : {name=Flatten, input shape=(1,64,4,4), output shape=(1,1024)}
(12) FullyConnected : {name=fc1, input shape=(1,1024), output shape=(1,64), activation=RELU}
(13) OutputSoftmax  : {name=Out, input shape=(1,64), output shape=(1,10), activation=Softmax}
```

Figure 15: Design of our CNN model for the CIFAR10 Experiment

So we have the design of our network and we found optimal values for the hyperparameter, it's now time to combined them together and run the CIFAR10 experiment once more.

5.2 Results

The first experiment for the CIFAR10 consisted into running a 5-for loop that iterates over different optimizer methods, different designs and different values for epochs, batchSize and learning rate. The goal of this first step was to :

- 1st : have an overview of which combination work and which one does not in order to determine which type of design and optimizer we should use
- 2nd : tune the hyperparameters values

We run the 5-for loop algorithm (7) and you can observe results on figure 16.

```
#############################################################################################
BEST VALUES : bacth : 128, epochs : 8, learningRate : 9.0E-4, design : 2, updateF : 2 and vaidation : 0.5366
```

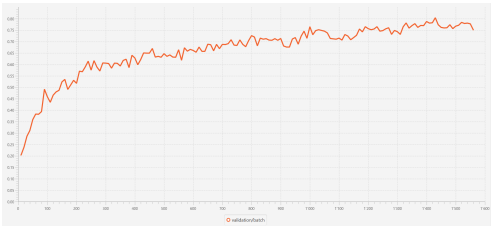
Figure 16: Result of the 5-for loop algorithm on CIFAR10

From this figure we can get some key information. First of all it gave us the optimal value for our 3 hyperparameters :

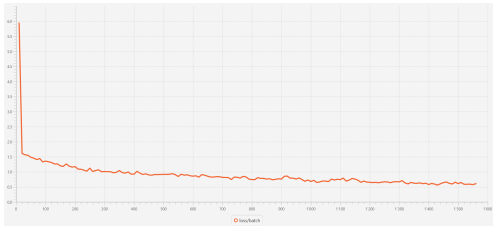
- Optimal bacthSize : 128
- Optimal epochs : 8
- Optimal learningRate : 0.0009

But more important, it gave us clues of which updateFunction and design should we use. *"Design : 2"* means that the optimal design is a CNN. That's why we decided to implement a better version of the CNN from SCT Experiment by adding layers and more (figure 15). *"UpdateFonction : 2"* means that the best update function for this experiment is the L2Decay already explained before in the report. Finally the last information we can get from this test us that the accuracy is around 54 %, so not enough yet.

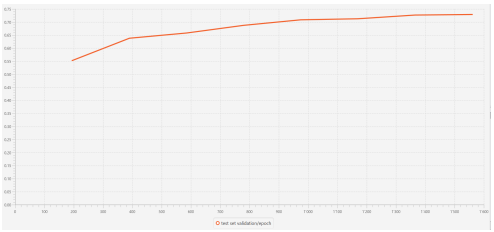
As described in the *Implementation* subsection, from those results we decided to improve our model in order to obtain a better accuracy. You can find the new model on figure 15. So we launched the CIFAR10 experiement but this time with the new model, the optimal combination of hyperparameter values and the correct optimizer method (L2Decay in this situation). With this specific combination we obtained interesting result that you can observe in figure 17 and 18. If you observe those results you can see that the validation per batch graph converge to 0.75 (75 %) and do not seem to oscillate too much. Moreover you can observe that the loss per batch graph also converge but only to 0.5 (50 %) so it could be an hint to know what we could change in order to improve our results. Finally, and most importantly, the test set validation per epoch accuracy converged 0.73 (73%) which is higher the 70% requested for the assignment. These result are very satisfying, it shows that we managed to find the correct hyperparameter value, the correct update function and finally to design a appropriate network for this experiment. You can find the final configuration on table 4.



(a) **validation/batch**



(b) **loss/batch**



(c) **test set validation/epoch**

Figure 17: Result of the CIFAR10 Experiment with the optimal combination of hyperparameter, optimizer and network design

epoch	#batches	#samples	#samples/sec	smooth loss
8	1560	1.996800e+05	1.004727e+02	6.145129e-01

Figure 18: Final status of the CIFAR10 experiment with nbr of epoch, nbr of processed batches, nbr of processed samples, nbr of sample/second and the smooth loss

Parameter	**Value**
BatchSize	128
Epochs	8
LearningRate	0.0009
Pre-processing	Mean Subtraction
Optimizer	L2Decay (decay = 0.0000001)
Design	CNN-3blocks
KernelSize, kernels and stride	3 , 32 and 2

Table 4: Final configuration for the CIFAR10 Experiment

6 Conclusions

In short, and in conclusion of this report, we would like to say that the goal of this assignment is achieved. In fact we gain practical experience with basic aspects of Deep Learning and especially with deep artificially neural network. We implemented 4 designs of DNN in order to succeed on the 4 experiments : Function (Regression), Zalando, SCT and CIFAR10. We learned about the different type of layers that composed a DNN and see how the design of the network can affect the result of our experiment. During this assignment we also implemented different interesting methods as the Mean Subtraction that allow us to pre-process the data but also optimization method as Gradient Descent with Momemtum, Adadelta or L2Decay. More over we also learned in which case to use them and alos how those methods work. Finally we also learned about the different hyperparameters used in DNN (batchSize, epochs, learning rate etcetera) and we saw how they can affect the results of our experiments and why this is very important to choose correctly those values in order to avoid over fitting or under fitting and to obtain optimal results. For this assignment we decided to use a 'brute force' method in order to determine most of our hyperparameters's values (after pre-selected some values which could fit our datas using different research done on the subject), but in the future it could be interesting to "redo" this assignment but this time using analytical methods or theoretical research only to determine the best configuration possible, without testing, to show if it is possible to predict our results mathematically.

7 Contributions

A statement on the contributions of each of the authors.

	implementation	**documentation**	**total #hours**
Auguste	55%	40%	30
Lucas	45%	60%	30

References

[1] Jason Brownlee. How to control the stability of training neural networks with the batch size. https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/.

[2] Jason Brownlee. Understand the impact of learning rate on neural network performance. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[3] Andrej Karpathy. Cs231 convolutional neural networks. http://cs231n.github.io/convolutional-networks/.

[4] Andrej Karpathy. Cs231 convolutional neural networks for visual recognition. https://cs231n.github.io/.

[5] Keras. https://keras.io/examples/cifar10$_cnn$/.

[6] Alex Krizhevsky. Convolutional deep belief networks on cifar-10. https://www.cs.toronto.edu/ kriz/conv-cifar10-aug2010.pdf.

[7] Machine Learning Mastery. https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/.

[8] Kevin Shen. Effect of batch size on training dynamics. https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e.

[9] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. 2012. https://arxiv.org/abs/1212.5701.