

Report Practical assignment 1

Machinists:

Fares Ben Slimane Parviz Haggi Mohammed Loukili

February 12, 2019

Abstract

This report explain our approaches to solving the problems of the practical assignment 1. the experiments we performed, results and conclusion of our work.

1 Problem 1 (MLP)

1.1 Building model

1. (using the python script `mlp.py` under the folder `problem1`), we build an MLP with two hidden layers `h1`(24 hidden units) and `h2`(12 hidden units).The total number of parameters of the network = $784 * 512 + 512 + 512 * 1024 + 1024 + 1024 * 10 + 10 = 937,482 \approx 0.9M$.
2. We Implemented the forward and backward propagation in a generalized way (can work in any number of layers) of the MLP in numpy without the use of a deep learning framework and using the provided class structure. (See python script `mlp.py` under `problem 1` folder).
3. We trained the MLP using the probability loss (cross entropy) as training criterion (See loss method in the NN class) and minimize this criterion to optimize the model parameters using stochastic gradient descent (See the update method in the NN class). (See python script `mlp.py` under `problem 1` folder).

1.2 Initialization

We consider a model architecture of two hidden layers `h1` = 24 hidden units and `h2` = 12 hidden units and a total number of parameters of 19270. We chose RELU as an activation function, a learning rate of 0.01 and a mini-batch size of 1000.

1. We trained the model for 10 epochs using the 3 initialization methods (Zero, normal and glorot) and we recorded the average loss measured for each method.
 - Zero initialization: 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3
 - Normal initialization: 3.41, 2.25, 2.20, 2.18, 2.16, 2.15, 2.13, 2.11, 2.08, 2.03

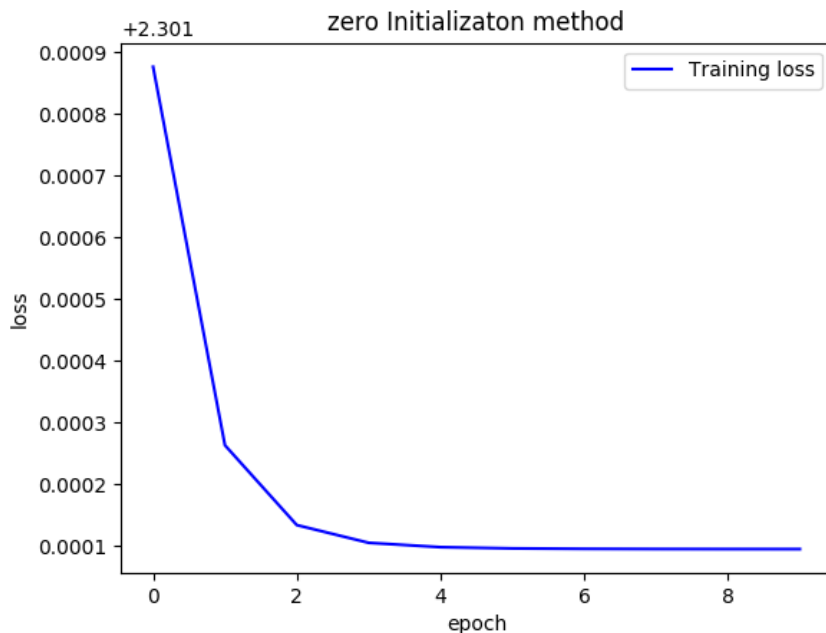


Figure 1: average loss against the training time (epoch) using zero initialization method.

- Glorot initialization: 1.55, 0.55, 0.40, 0.35, 0.32, 0.30, 0.29, 0.27, 0.26, 0.25

2. We plot the losses against the training time (epoch) using each initialization method (Figures 1, 2 and 3). We conclude from the plots that the glorot initialization is the best among the methods in which the loss decreases rapidly at each epoch whereas, for the zero initialization, the loss decreases very very slowly. An explanation for this is that by initializing all the weights to zero. All the hidden nodes will end up with the same value and therefore we end up learning just one function. This is called the symmetry problem. We break the symmetry problem by initializing the weights randomly (like we did in the glorot and normal initializations).

1.3 Hyperparameter Search

1. The combination of hyper-parameters that we found in which the average accuracy rate on the validation set reach 97.2% accuracy: A network with 2 hidden layers $h1 = 64$ hidden units and $h2 = 32$ hidden units with a total number of parameters of 52650. A Relu activation function, a learning rate of 0.01, a mini batch size of 64, a number of epochs of 50. (See Figure ?? plotting the training/validation accuracy over epochs).
2. We tried different hyper-parameters for learning rate, number of epochs
Number of epochs is not an hyperparameter, see slack channel tp1 01/29,

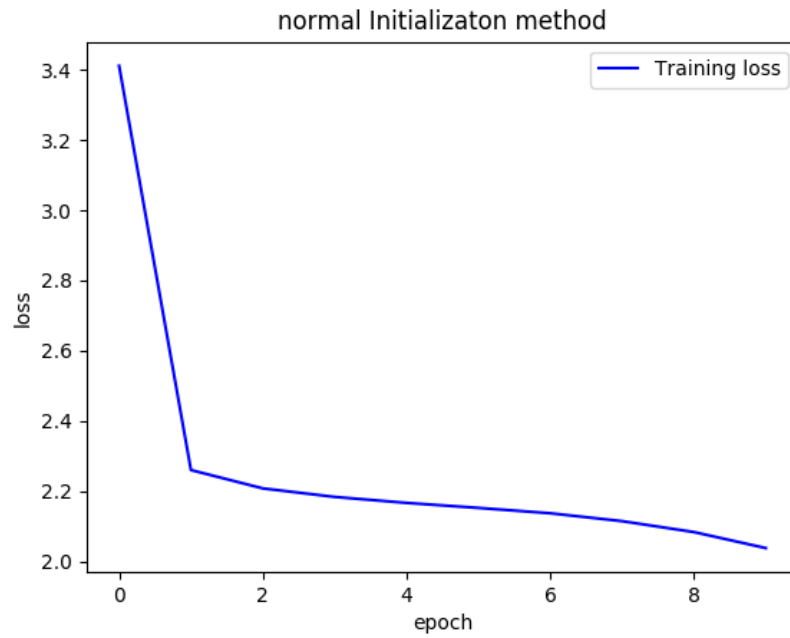


Figure 2: average loss against the training time (epoch) using normal initialization method.

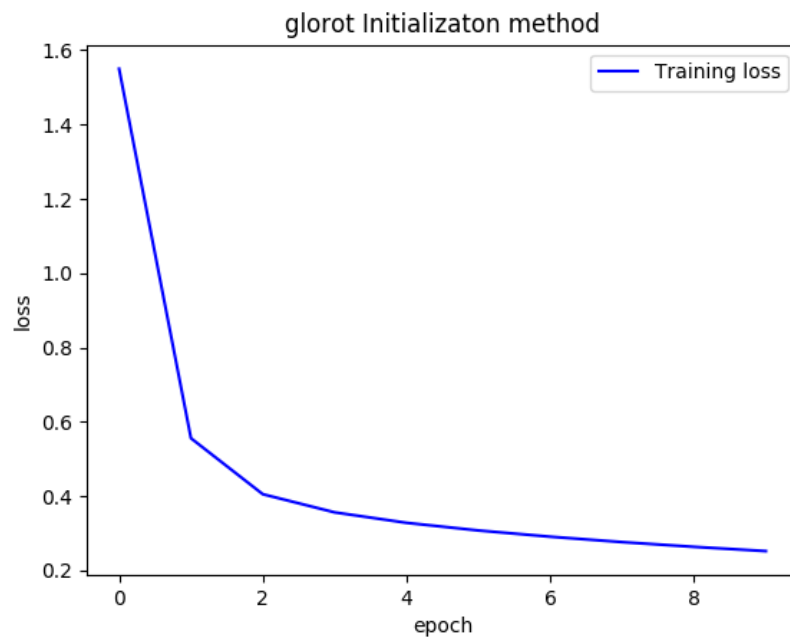


Figure 3: average loss against the training time (epoch) using glorot initialization method.

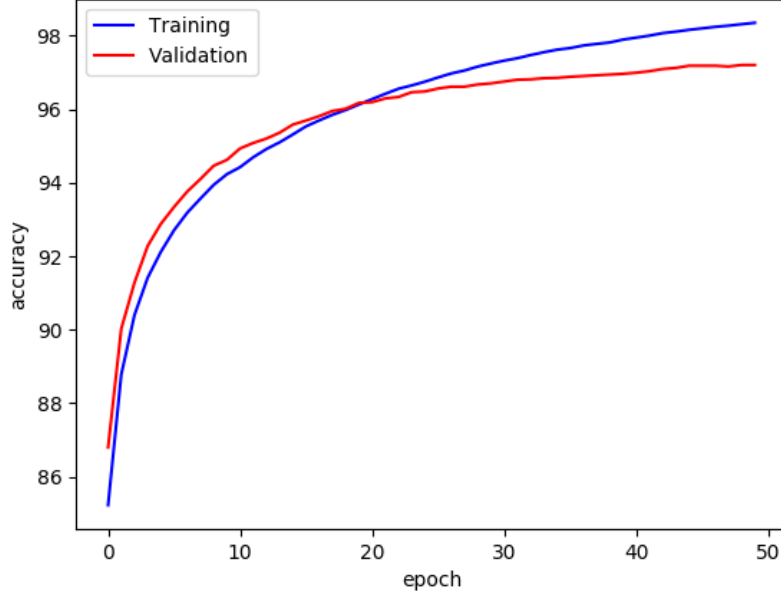


Figure 4: Validation/training accuracy against the training time (epoch) using the chosen hyper-parameters.

network architecture and batch size. We consider the same model architecture.

1.3.1 Learning rate

We keep the same settings and we only change the learning rate.

- learning rate of 0.01: validation accuracy = 97.2% (See Figure ??).
- learning rate of 0.1: validation accuracy = 97.6% (See Figure 5).
- learning rate of 0.001: validation accuracy = 93% (See Figure 6).

1.3.2 Number of epochs

We keep the same settings and we only change the number of epochs.

- number of epochs of 50: validation accuracy = 97.2% (See Figure 4).
- number of epochs of 5: validation accuracy = 92.8% (See Figure 7).
- number of epochs of 150: validation accuracy = 97.3% (See Figure 8).

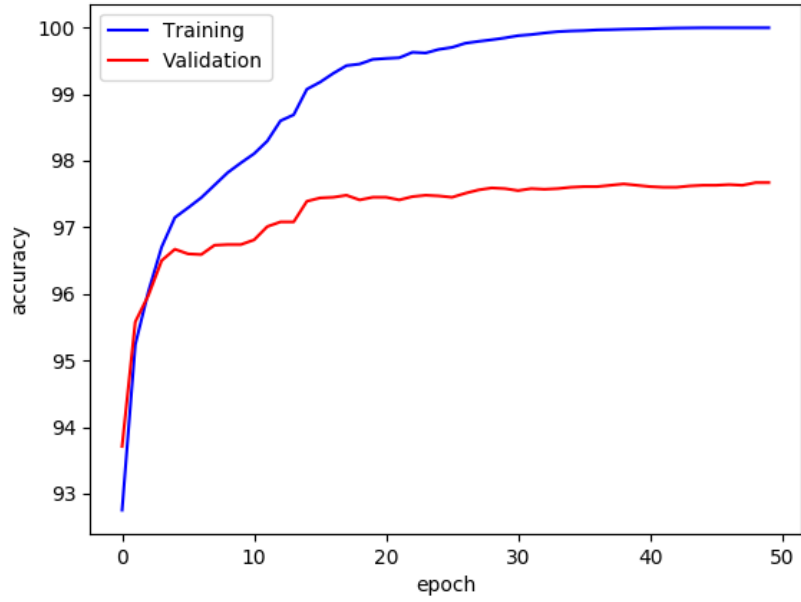


Figure 5: Validation accuracy against the training time (epoch) using a learning rate of 0.1.

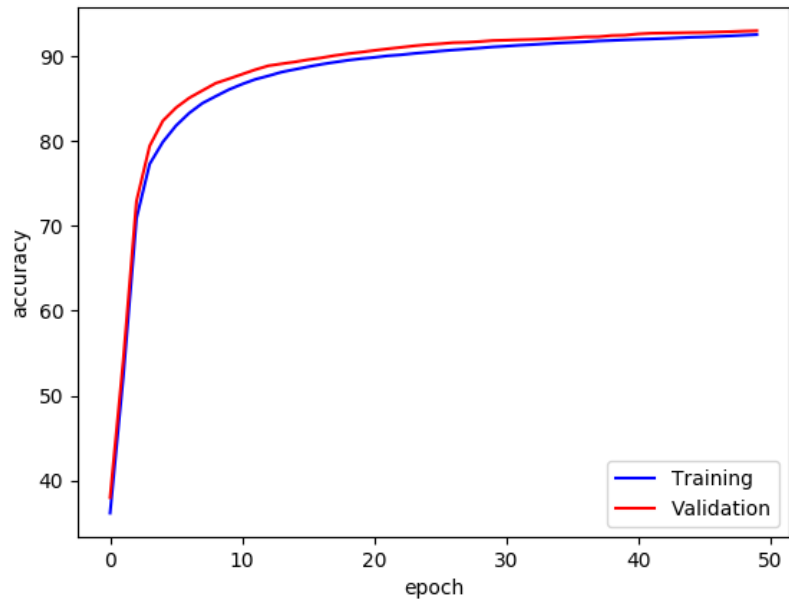


Figure 6: Validation accuracy against the training time (epoch) using a learning rate of 0.001.

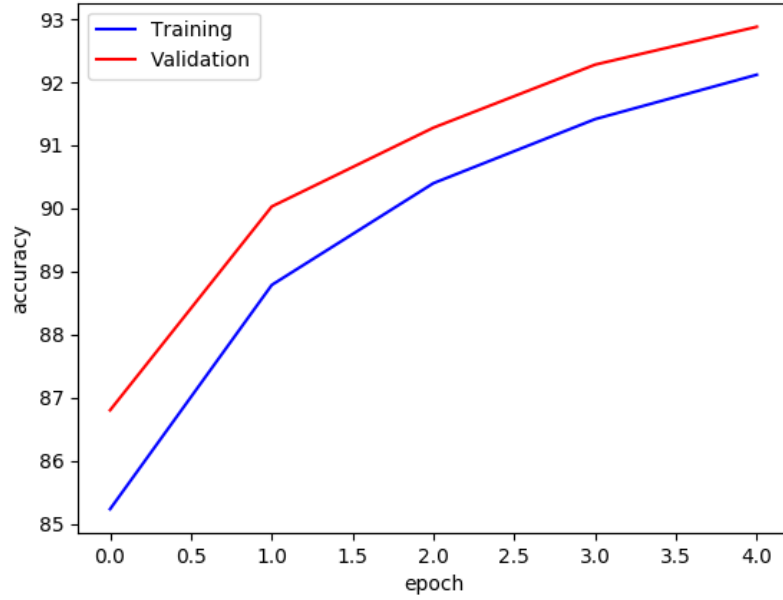


Figure 7: Validation/training accuracy against the training time (epoch) using a number of epochs of 5..

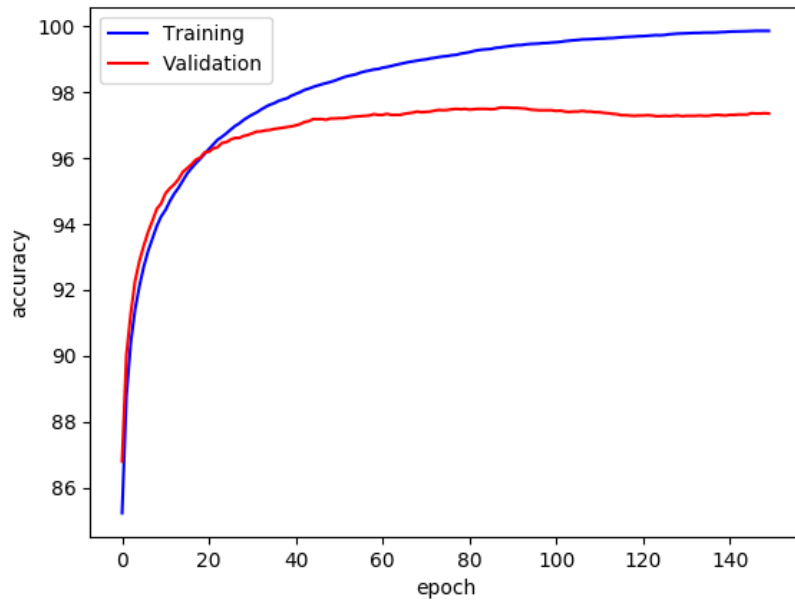


Figure 8: Validation/training accuracy against the training time (epoch) using a number of epochs of 150.

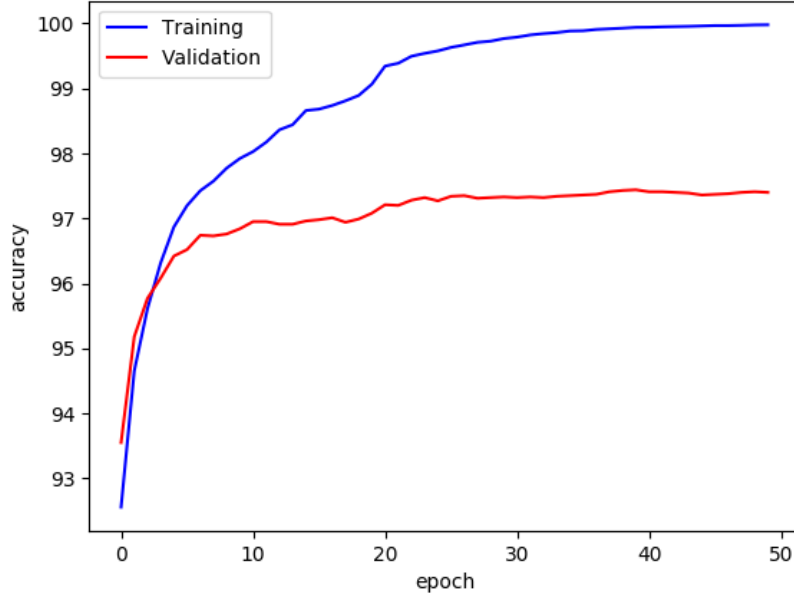


Figure 9: Validation/training accuracy against the training time (epoch) using a number of mini-batches of 8.

1.3.3 Number of mini-batches

We keep the same settings and we only change the number of mini-batches.

- number of mini-batches of 64: validation accuracy = 97.2% (See Figure 4).
- number of mini-batches of 8: validation accuracy = 97.4% (See Figure 9).
- number of mini-batches of 512: validation accuracy = 93.4% (See Figure 10).

1.3.4 Non-linearity activation function

We keep the same settings and we only change the activation function.

- Relu activation function: validation accuracy = 97.2% (See Figure 4).
- Sigmoid activation function: validation accuracy = 92.2% (See Figure 11).
- tanh activation function: validation accuracy = 93.9% (See Figure 12).

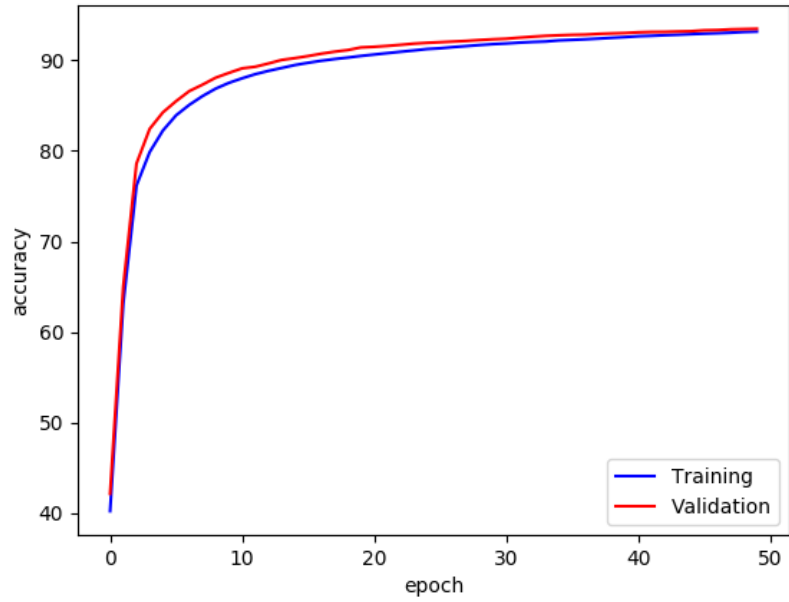


Figure 10: Validation/training accuracy against the training time (epoch) using a number of mini-batches of 512.

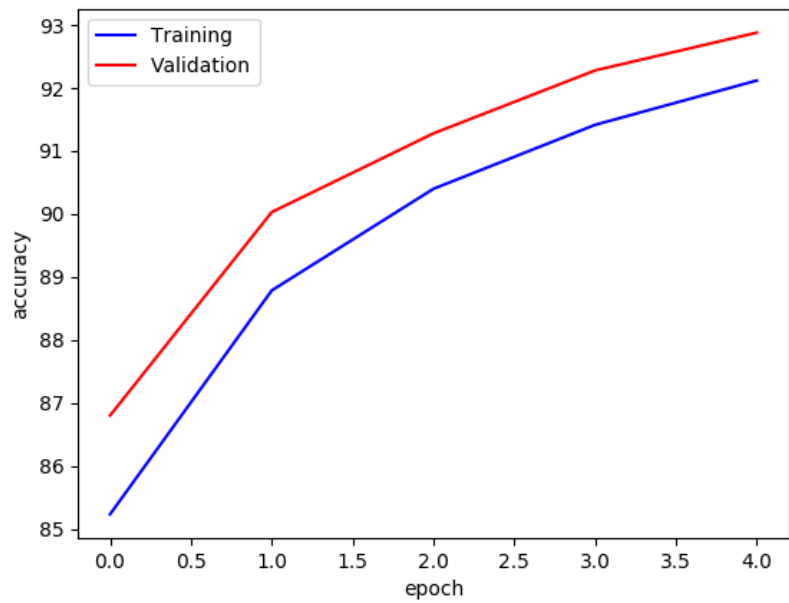


Figure 11: Validation/training accuracy against the training time (epoch) using a sigmoid activation function.

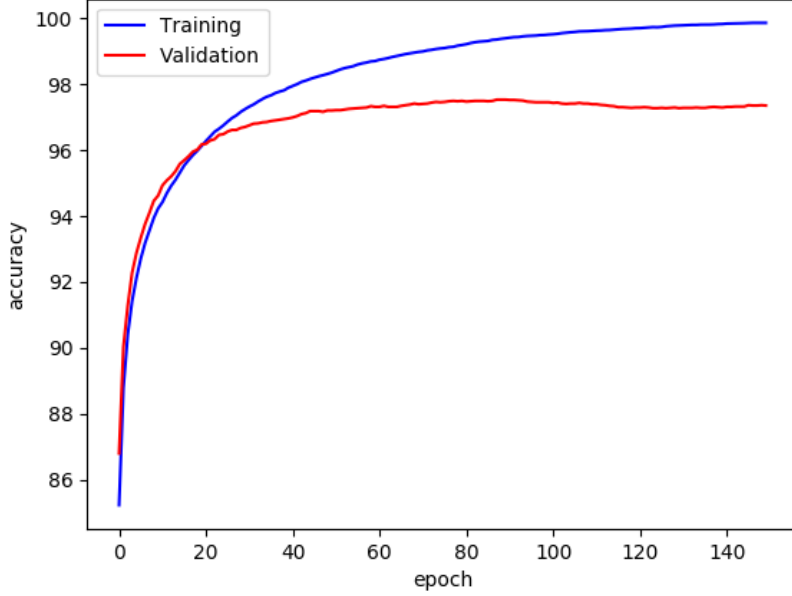


Figure 12: Validation/training accuracy against the training time (epoch) using a tanh activation function.

1.3.5 Network architecture (number of hidden units)

We keep the same settings and we only change the hidden layers dimensions.

- Network with 2 hidden layers $h1 = 64$, $h2 = 32$: validation accuracy = 97.2% (See Figure 4).
- Sigmoid activation function: validation accuracy = 92.2% (See Figure 5).
- tanh activation function: validation accuracy = 93.9% (See Figure 6).

1.4 Gradient validation using finite difference approximation

We implemented a gradient validation function *grad_check* which returns the maximum difference between the true gradient and the finite difference gradient approximation for a given number of elements of a network parameter and a given precision ϵ .

We validated the gradients for the first $p = \min(10, m)$ elements of the second layer weights ($W2$) with m number of elements. Using $\epsilon = \frac{1}{N}$ Figure 1.4 shows the maximum difference as a function of the N .

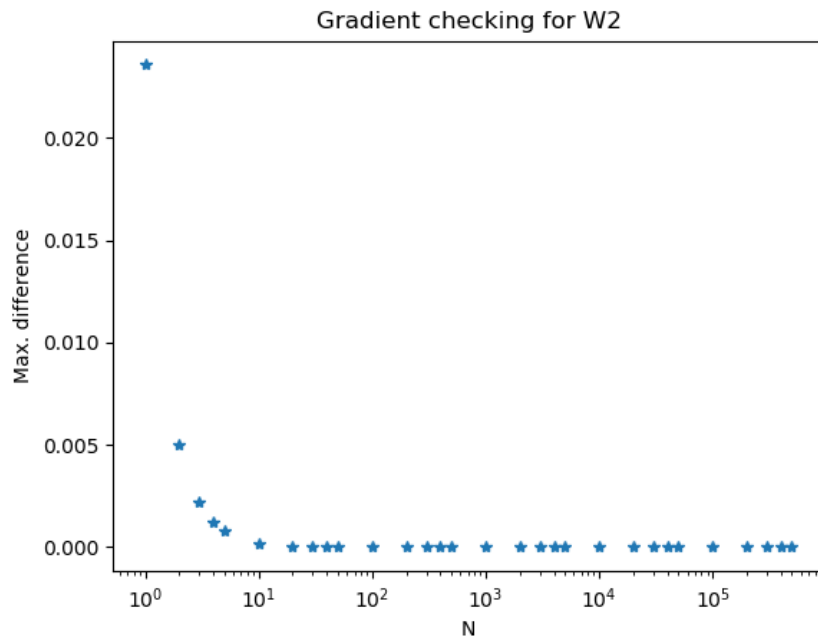


Figure 13: Maximum difference between the analytic gradient and the finite difference approximation for some elements of the weight matrix at the second layer as a function of the precision $N = \frac{1}{\epsilon}$

The approximation of the gradient for each element gets closer to the real partial derivative as ϵ gets smaller, this is consistent with theory since in the definition of the derivative the epsilon tends to zero.

2 Problem 2 Convolutional Networks

2.1 Architecture

As it is a common practice, we decided to implement our convolutional network using layers that sequentially apply a convolution followed by a ReLU followed by pooling. Given the size of the images in the dataset MNIST (28×28 pixels) we decided to use four layers where the convolutions are padded in order to keep the size and the pooling kernels have a spatial size 2×2 and stride of 2 to obtain a single spatial dimension at the end. In order to fix some other parameters we chose the convolution kernels to have the spatial size of 3×3 and the number of channels at to double at each convolution layer. With these settings the only parameter that controls the size of the network is the number of output channels at the first convolutional layer.

In order to obtain a similar number of parameters than our mlp of Problem 1 ($\sim 50K$ parameters) we set that value to 12, so the number of channels at the layers of our network are 1, 12, 24, 48, 96.

2.2 Performance

Figure 2.2 shows the training and validation accuracy of the model at each epoch. After training the model for 10 epoch we obtain a validation accuracy of 97%

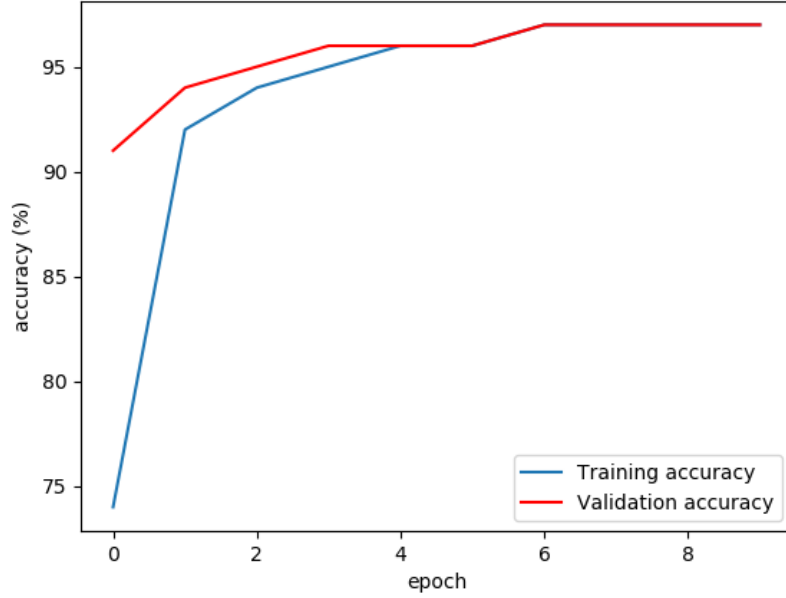


Figure 14: Training and validation accuracy of the convolutional model of Section 2

3 Problem 3 (Kaggle challenge)

3.1 Architecture of the model

The model's architecture that we have used is inspired from VGG model, which has the following components (see Figure 15) :

- 13 convolutions layers, with 0 padding at each layer, and a kernel of size (3,3) and stride of 1.
- 5 max pooling with kernel of size (2,2) and stride of 2.
- 2 fully connected layers with 4096 hidden units
- 1 output layer

The total parameters of this model is 23,111,490, which is a medium size in comparison with the most recent deep learning models.

Our experimentation has shown that a model with more layers performs better than a model with fewer ones. In fact, for this competition, we have found that the VGG model above have performed better than AlexNet-based architecture.

Layer (type)	Output Shape	Param #
zero_padding2d_1 (ZeroPaddin	(None, 66, 66, 3)	0
conv2d_1 (Conv2D)	(None, 64, 64, 64)	1792
zero_padding2d_2 (ZeroPaddin	(None, 66, 66, 64)	0
conv2d_2 (Conv2D)	(None, 64, 64, 64)	36928
max_pooling2d_1 (MaxPooling2	(None, 32, 32, 64)	0
zero_padding2d_3 (ZeroPaddin	(None, 34, 34, 64)	0
conv2d_3 (Conv2D)	(None, 32, 32, 128)	73856
zero_padding2d_4 (ZeroPaddin	(None, 34, 34, 128)	0
conv2d_4 (Conv2D)	(None, 32, 32, 128)	147584
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 128)	0
zero_padding2d_5 (ZeroPaddin	(None, 18, 18, 128)	0
conv2d_5 (Conv2D)	(None, 16, 16, 256)	295168
zero_padding2d_6 (ZeroPaddin	(None, 18, 18, 256)	0
conv2d_6 (Conv2D)	(None, 16, 16, 256)	590080
zero_padding2d_7 (ZeroPaddin	(None, 18, 18, 256)	0
conv2d_7 (Conv2D)	(None, 16, 16, 256)	590080
max_pooling2d_3 (MaxPooling2	(None, 8, 8, 256)	0
zero_padding2d_8 (ZeroPaddin	(None, 10, 10, 256)	0
conv2d_8 (Conv2D)	(None, 8, 8, 512)	1180160
zero_padding2d_9 (ZeroPaddin	(None, 10, 10, 512)	0
conv2d_9 (Conv2D)	(None, 8, 8, 512)	2359808
zero_padding2d_10 (ZeroPaddi	(None, 10, 10, 512)	0
conv2d_10 (Conv2D)	(None, 8, 8, 512)	2359808
max_pooling2d_4 (MaxPooling2	(None, 4, 4, 512)	0
zero_padding2d_11 (ZeroPaddi	(None, 6, 6, 512)	0
conv2d_11 (Conv2D)	(None, 4, 4, 512)	2359808
zero_padding2d_12 (ZeroPaddi	(None, 6, 6, 512)	0
conv2d_12 (Conv2D)	(None, 4, 4, 512)	2359808
zero_padding2d_13 (ZeroPaddi	(None, 6, 6, 512)	0
conv2d_13 (Conv2D)	(None, 4, 4, 512)	2359808
max_pooling2d_5 (MaxPooling2	(None, 2, 2, 512)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 2048)	4196352
dense_2 (Dense)	(None, 2048)	4196352
dense_3 (Dense)	(None, 2)	4098
Total params: 23,111,490		
Trainable params: 23,111,490		
Non-trainable params: 0		

Figure 15: Model architecture

3.2 Learning curves

The following figures (16,17) show respectively the training and validation loss and accuracy per epochs:

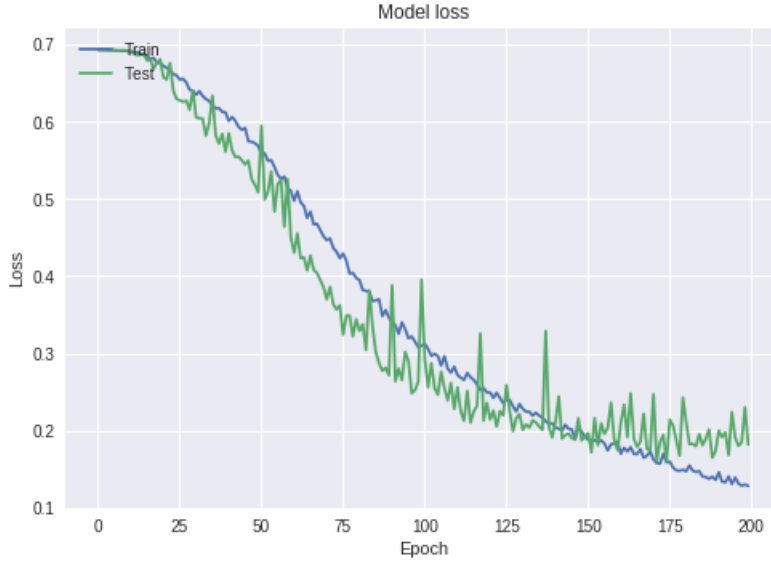


Figure 16: Loss of the model by epoch



Figure 17: Accuracy of the model by epoch

Before starting training the model, we have split the input images randomly to training and validation sets with a ratio of 80% – 20% respectively. (see the code on appendix).

We have also used data augmentation strategy to improve the variance of the model.

The learning curves above show that the model reaches its optimal value around epoch 150. After that point, the model starts overfitting as the loss on

validation set starts growing.

We believe that a regularization strategy as *Weight decay* could potentially get a better result, as it will push the model to learn beyond 150 epochs without overfitting.

Notice that the accuracy on the testset is better than the accuracy on the validation set, which is a proof that the model generalize well, and the data augmentation strategy was a good one to improve the model's performance on unseen data.

3.3 Hyperparameters settings

We have selected different hyperparameters to tune on the validation set in order to get the best model configuration, which are: - Number of hidden layers - Size of the kernels - Learning rate - Batch size

The following table gives a comparison of the model's performance while for each parameter while keeping the others fixed.

The table below (1) show the accuracy of the model for given different model architecture while keeping fixed the other parameters : kernel size, number of epoch, ...

Nbr of layers	Accuracy	Loss	Total nb of parameters
6	0.3109	0.8574	2,736,066
10	0.2713	0.8841	5,147,458
13	0.2068	0.9078	17,076,546
16 lite	0.1984	0.9158	23,111,490
16	?	?	39,896,898

Table 1: Number of layers tuning

We notice that a model has a better performance when he has more hidden layers. This is normal because more layers means more capacity for the model.

The next table (2) gives a comparison for different kernel size. This hyperparameter influence drastically the architecture of the model, that is a bigger kernel shorten the model and it could'nt be deeper unless we add 0 padding which gives a poorer feature map. We notice that in general a smaller kernel size gives a better result!

Kernel size	Accuracy	Loss
2	0.5832	0.6919
3	0.5761	0.6912
4	0.5444	0.6894
5	0.5418	0.6930

Table 2: Kernel size tuning

The next table (3) gives a comparison of the model performance as a function of the batch size. We notice that a small batch size push the model to make more iterations which make the SGD converge better than with bigger batch size.

Batch size	Accuracy	Loss
16	0.5320	0.6892
32	0.5015	0.6919
64	0.4859	0.6935
128	0.5000	0.6929

Table 3: Kernel size tuning

And finally, table (4) gives a comparison of the model performance as a function of the learning rate. We notice, that the model convergence is very slow with a small learning rate and faster with a relatively bigger one like 0.03.

Learning rate	Accuracy	Loss
0.03	0.8935	0.2417
0.01	0.8595	0.3279
0.003	0.6930	0.5981
0.001	0.5825	0.6709






Table 4: Kernel size tuning

Our final model has those values as hyperparameters: - Number of hidden layers: 13 convolutions and 3 FC - Size of the kernels: (3,3) - Learning rate: 0.03 - Batch size: 16

This model give an accuracy of : 94% on validation set and 93.7% on the public leaderboard. The model was trained for 200 but get his best parameters around 150 epochs.

As we haven't used any regularization strategies, on the training phase, we have used the early stopping strategy the use the best model's parameters before he starts overfitting.

The model get a decent performance (not expected though) of 91% on validation set and 92% (see Figure ?? on the public leader board (see Figure 18) which is a very good result.

#	Team Name	Kernel	Team Members	Score 🏆	Entries	Last
1	shotgun pocket = noob?			0.97519	3	22d
2	Max Schwarzer			0.95278	6	2d
3	Jakil			0.94037	4	1h
4	Ryuk			0.93837	1	5d
5	TheMachinists			0.93717	5	now

Your Best Entry ⬆

You advanced 4 places on the leaderboard!

Your submission scored 0.93717, which is an improvement of your previous score of 0.92036. Great job!


 [Tweet this!](#)

Figure 18: Rank at the public leaderboard

3.3.1 Feature map visualization

The feature map visualization give in some sens an intuition on how a CNN works to learn from images. Across the layers, the feature map go from high level pattern recognition to low level as we go deeper in the model. The following image shows for each layers the feature map after each activation layer

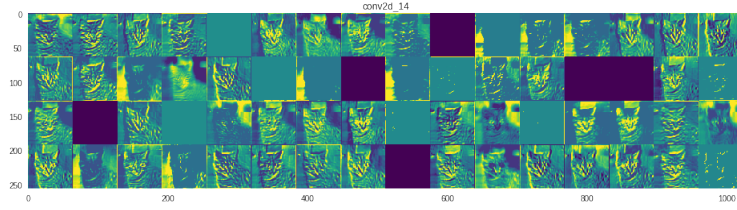


Figure 19: Feature map after conv layer 1

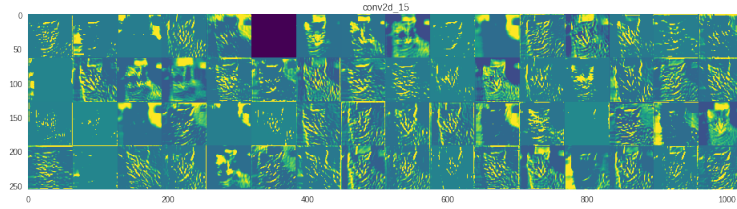


Figure 20: Feature map after conv layer 2

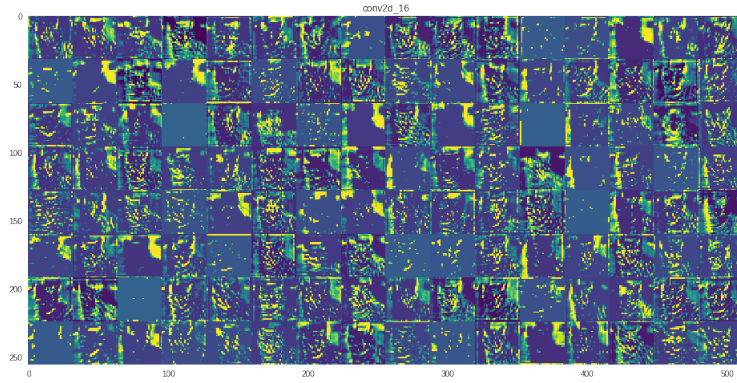


Figure 21: Feature map after conv layer 3

3.3.2 Performance of the model

The model get nearly 94% accuracy on test set (public leaderboard) which is not bad given that we haven't used any fancy techniques like regularization. It is interesting to investigate which images the model has misclassified, in order to identify ways to improve the model's performance.

Our model has misclassified nearly 5% of the validation set images, as shown in the confusion matrix (see Figure 22). We notice that the model has misclassified more Dogs than Cats.

```
[36] confusion_matrix(y_true2[:,1].astype(int), y_pred)
Out[36]: array([[1821, 179],
               [ 92, 1908]])
```

Figure 22: Confusion matrix of the model

To understand why the model have done such misclassification, we will display a sample of two kind of mislcassifications:

- (a) The model clearly misclassified images:

In the Figure 23, we displayed a sample of images that the model has misclassified with probability ≥ 0.6 . We notice that these images are noisy and doesn't show clearly the animal's figure or they contains other objects like human or fences.

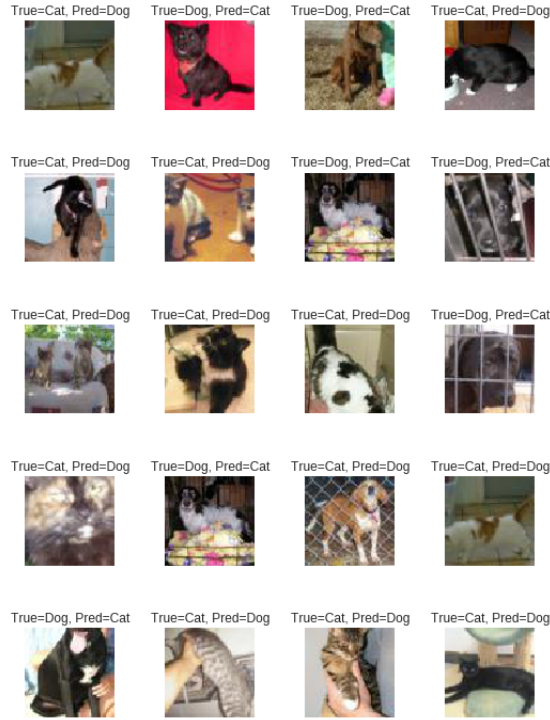


Figure 23: 100% miiclassification

- (b) The model predicts around 50% on both classes:

In the Figure 24 we can see some images that the model have bearily misclassified them. We notice the same pattern as the item (a) where the

animals took different position in the picture with other objects or half captured.

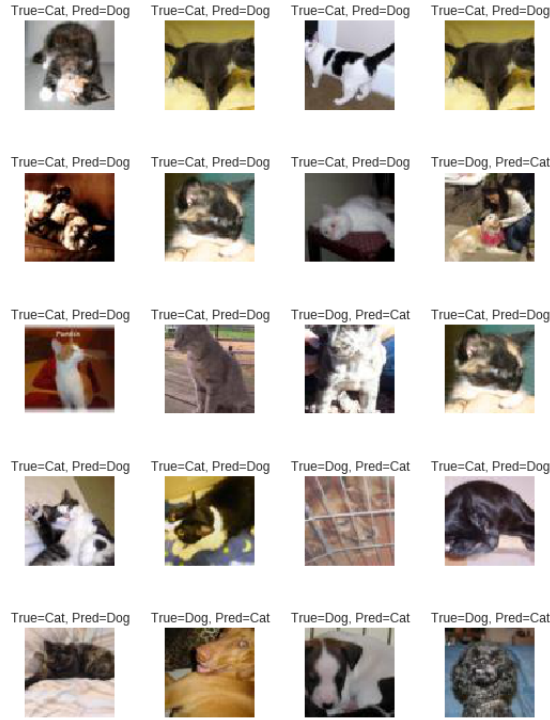


Figure 24: 50% misclassification

As a conclusion, the classifier that we have build was able to classify correctly 94% of images which is a good result without even using any regularization or fancy optimization algorithm. However, the model couldn't classify some images because of it's complexity, like having other objects around or taking just a part of the animal. In that case, we may suggest to add more data by sampling over those kind of misclassified images and give train the model on them, or use the K-Fold cross validation in order to train the model on all kind of images.

References

online, author = Thayumanav Jayadevan, title = Why don't we initialize the weights of a neural network to zero?, year = 2018, url = <https://www.quora.com/Why-dont-we-initialize-the-weights-of-a-neural-network-to-zero>, urldate = 2019-02-05