Report Practical assignment 1

TheMachinists team:

Fares Ben Slimane
Parviz Haggi
Mohammed Loukili
Jorge A. Gutierrez Ortega

February 17, 2019

Abstract

This report explains our approaches to solving the problems of the practical assignment 1, the experiments we performed, the results and conclusion of our work. The code we developped is uploaded to our Github repository [3]

1 Problem 1 (MLP)

1.1 Building model

1. Using the python script mlp.py under the folder problem 1 [3], we built an MLP with two hidden layers h1 (512 hidden units), and h2 (256 hidden units).The total number of the parameters of this network is:

$$784 * 512 + 512 + 512 * 256 + 256 + 256 * 10 + 10 = 535,818 \approx 0.5M$$

2. We implemented the forward and backward propagation for a flexible network that can include any number of layers. This was done with the numpy package follwing the provided class structure and without the use of deep learning framework. (See python script mlp.py under problem 1 folder).

3. We trained the MLP using cross entropy as the training criterion (See loss method in the NN class in the mlp.py script) which is then minimized in order to find the optimal model parameters. To this end we used stochastic gradient descent (See the update method in the NN class in the mlp.py script).

1.2 Initialization

We consider a model architecture of two hidden layers h1 with 512 hidden units and h2 with 256 hidden units, and a total number of 535,818 parameters. We chose RELU as an activation function, a learning rate of 0.01 and a mini-batch size of 64.

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.

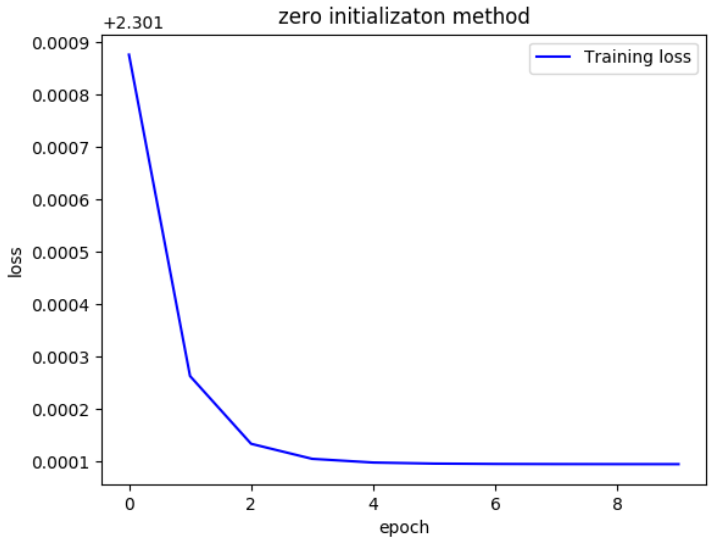


Figure 1: average loss against the training time (epoch) using zero initialization method.

- Zero initialization(Figure 1): 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3
- Normal initialization(Figure 2): inf, inf, 4.37, 2.90, 2.15, 1.67, 1.35, 1.11, 0.94, 0.80
- Glorot initialization(Figure 3): 0.85, 0.36, 0.30, 0.27, 0.24, 0.22, 0.21, 0.19, 0.18, 0.17

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 slowly. An explanation for this is that by initializing all the weights to zero, all the hidden nodes will have the same value and the network will learn just one function. This is called the symmetry problem, which is dealt with by initializing the weights randomly (as in the cases of Glorot and normal initializations)[1].

1.3 Hyperparameter Search

1. The combination of hyper-parameters that we found in which the average accuracy rate on the validation set reach 97.3% accuracy: Since we have the freedom to examine a variation of different architectures and hyperparameters, we chose a network with 2 hidden layers h1 (512 hidden units) and h2 (256 hidden units) with a total number of 535,818 parameters. We chose Relu activation, a learning rate of 0.01, a mini batch size of 64, the number of epochs as 30 and the glorot initialization. (See Figure 4).

2. We tried different hyper-parameters for learning rate, different network architectures, batch sizes and activation functions (Tanh, Relu, Sigmoid).

1.3.1 Changing the Learning rate

- learning rate of 0.01: validation accuracy = 97.3% (See Figure 4).

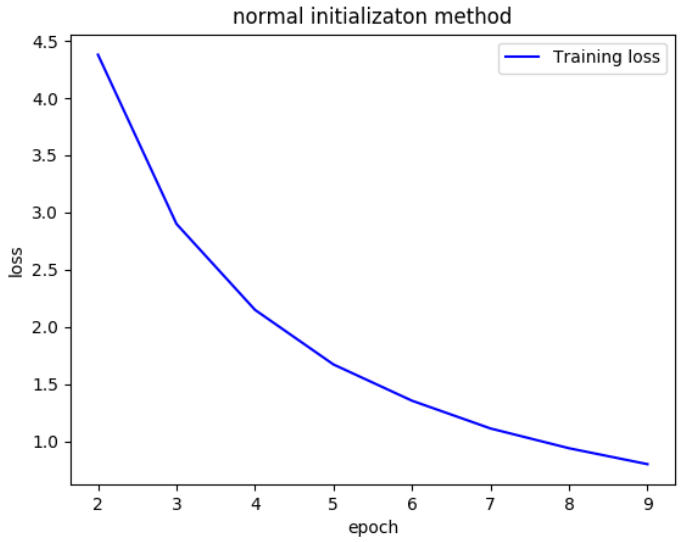


Figure 2: average loss against the training time (epoch) using normal initialization method.

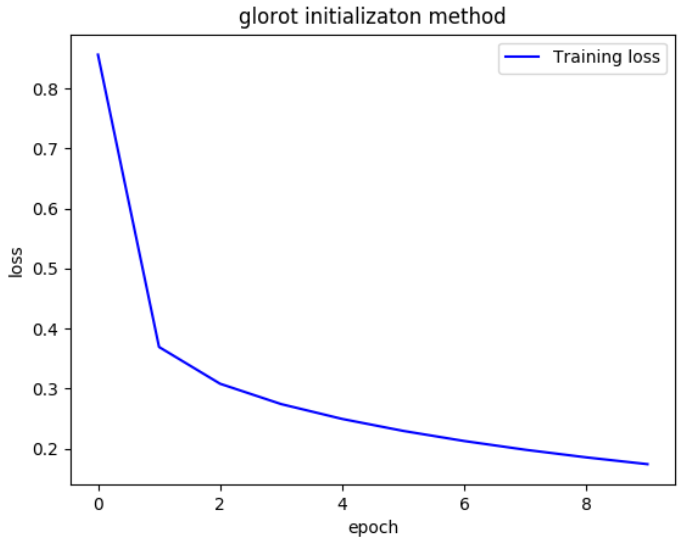


Figure 3: average loss against the training time (epoch) using glorot initialization method.

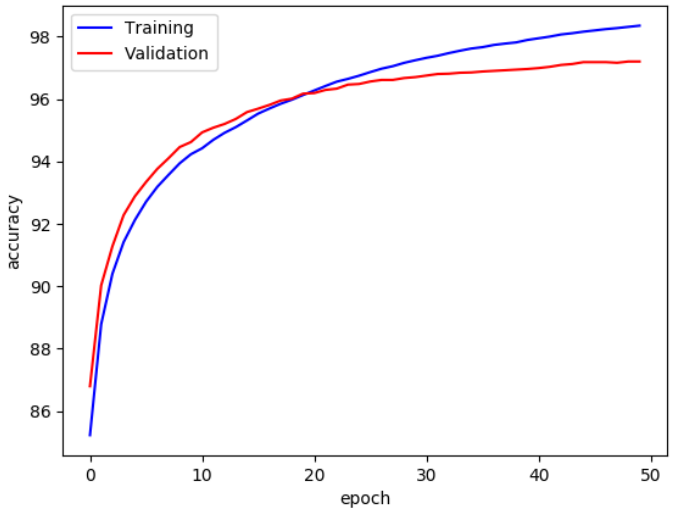


Figure 4: Validation/training accuracy against the training time (epoch) using the chosen hyper-parameters.

- learning rate of 0.1: validation accuracy = 98.1% (See Figure 5).
- learning rate of 0.001: validation accuracy = 92.6% (See Figure 6).
- learning rate of 0.7: validation accuracy = 98.4% (See Figure **??**).

We observe that when the learning rate is low, training is more reliable, however, it takes much more time to converge. On the other hand, if the learning rate is high, then training may not converge and we observe unstable learning curve (since the weight changes is big and overshoot the local min).

1.3.2 Changing the Number of mini-batches

- number of mini-batches of 64: validation accuracy = 97.3% (See Figure 4).
- number of mini-batches of 8: validation accuracy = 97.4% (See Figure 7).
- number of mini-batches of 512: validation accuracy = 93.1% (See Figure 8).

We observe that the model performs better when the batch size is large.

1.3.3 Non-linearity activation function

- Relu activation function: validation accuracy = 97.2% (See Figure 4).
- Sigmoid activation function: validation accuracy = 92.2% (See Figure 9).
- tanh activation function: validation accuracy = 93.9% (See Figure 10).

Obviously Relu and tanh perform better in terms of accuracy.

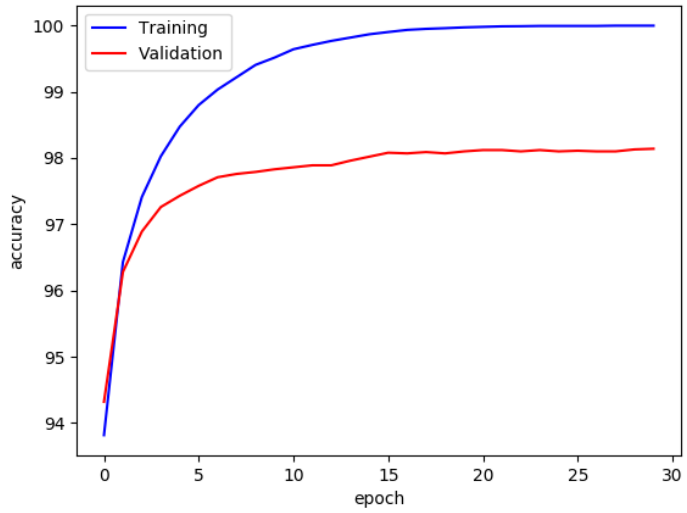


Figure 5: Validation accuracy against the training time (epoch) using a learning rate of 0.1.

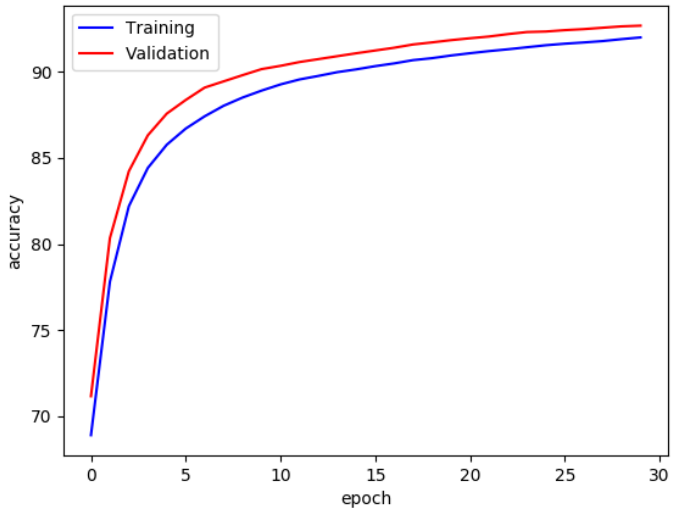


Figure 6: Validation accuracy against the training time (epoch) using a learning rate of 0.001.

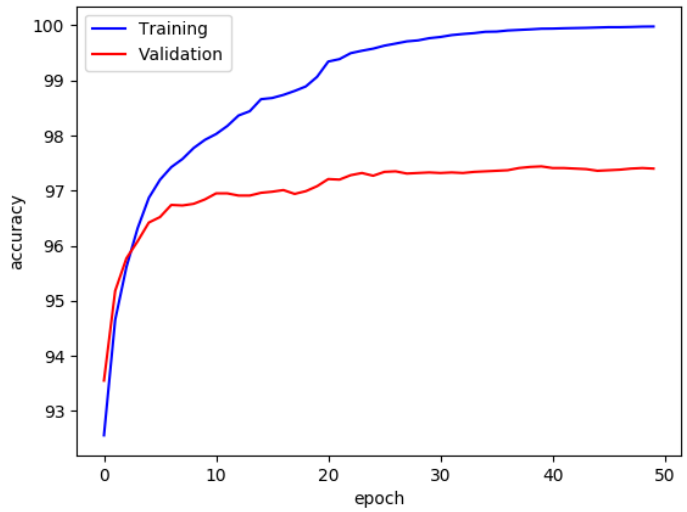


Figure 7: Validation/training accuracy against the training time (epoch) using a number of mini-batches of 8.

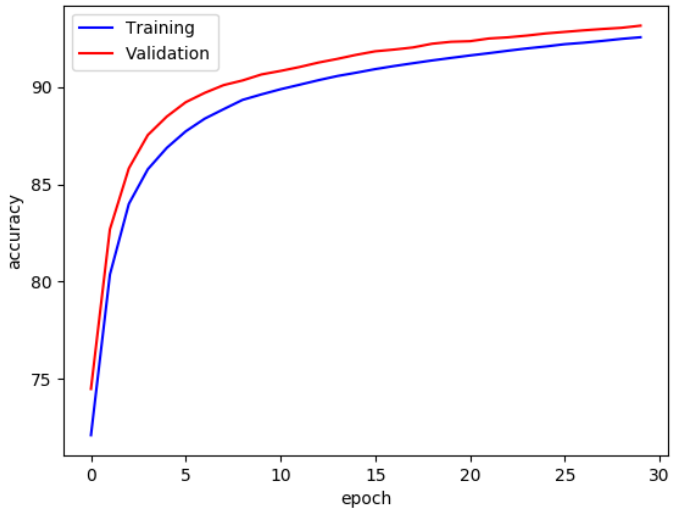


Figure 8: Validation/training accuracy against the training time (epoch) using a number of mini-batches of 512.

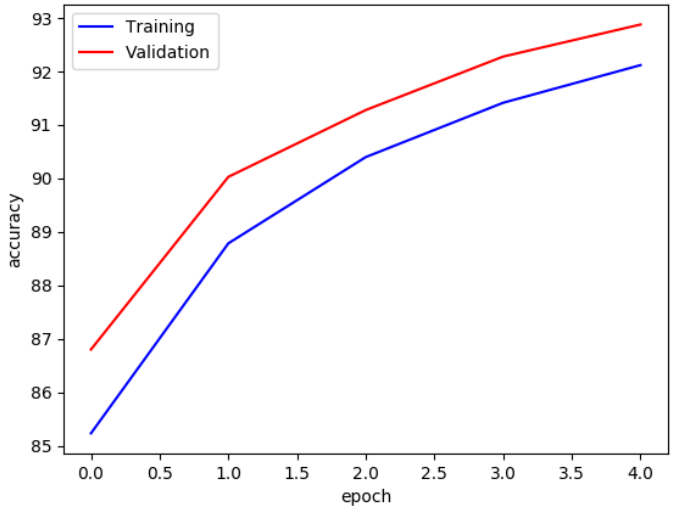


Figure 9: Validation/training accuracy against the training time (epoch) using a sigmoid activation function.

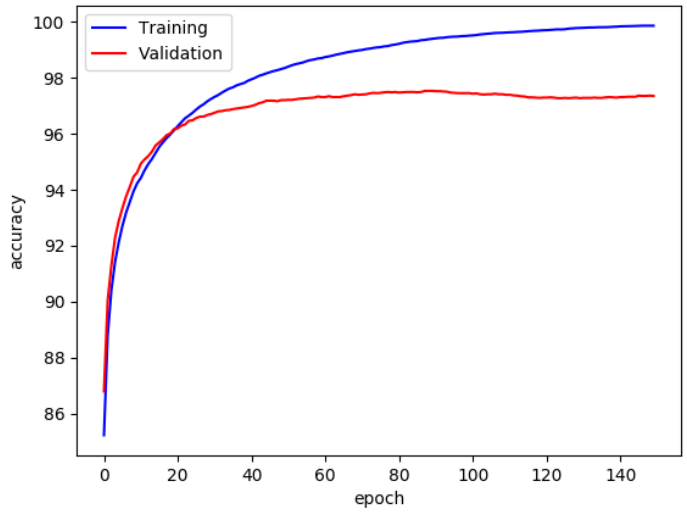


Figure 10: Validation/training accuracy against the training time (epoch) using a tanh activation function.

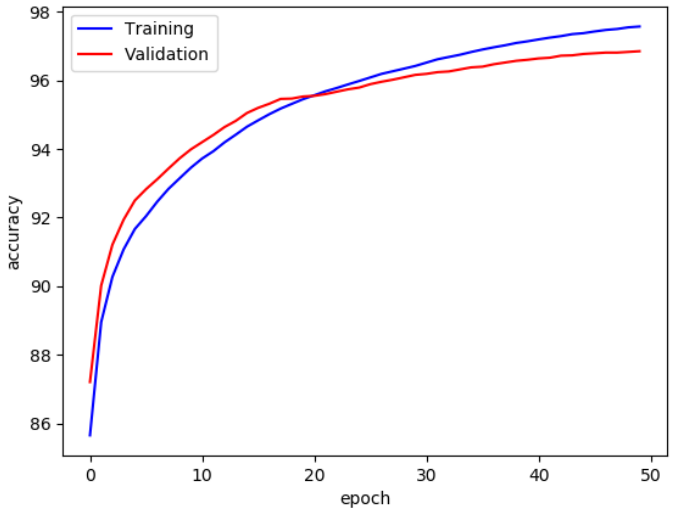


Figure 11: Validation/training accuracy against the training time (epoch) using a network architecture with h1=32 and h2=64

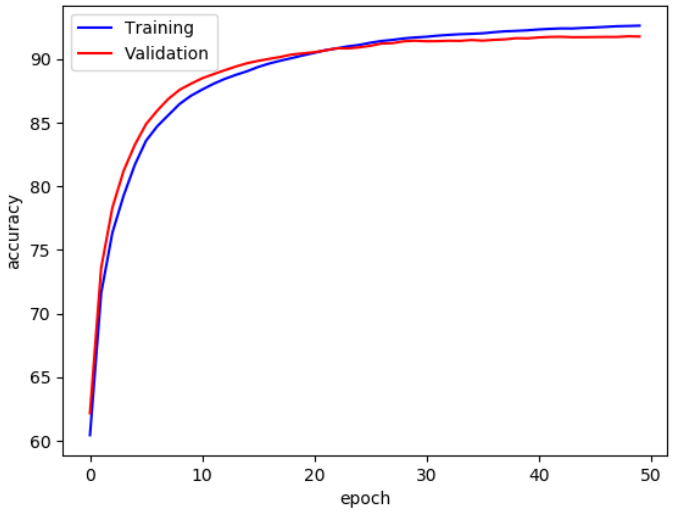


Figure 12: Validation/training accuracy against the training time (epoch) using a network architecture with h1=10 and h2=5

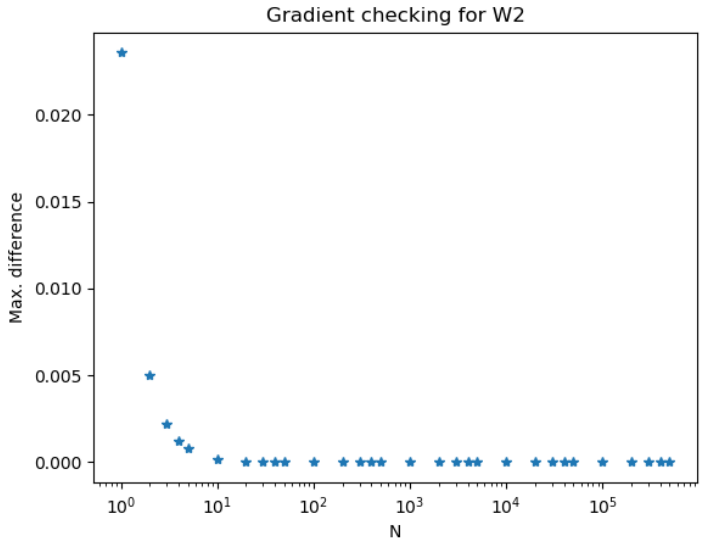


Figure 13: Maximum difference between the analytic gradient and the finite difference approximation for some elements of the weight matrix at the second layer as a function of the precision $N = \frac{1}{\epsilon}$

1.3.4 Network architecture (Chaning the number of hidden units)

- Network with 2 hidden layers h1 = 64, h2 = 32: validation accuracy = 97.2% (See Figure 4).
- Network with 2 hidden layers h1 = 32, h2 = 64: validation accuracy = 96.85% (See Figure 11).
- Network with 2 hidden layers h1 = 10, h2 = 5: validation accuracy = 91.77% (See Figure 12).

1.4 Gradient validation using finite difference approximation

We implemented the gradient validation method *grad_check* under the mlp.py script which returns the maximum difference between the true gradient and the finite difference gradient approximation for 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.

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 epochs we obtain a validation accuracy of 98%

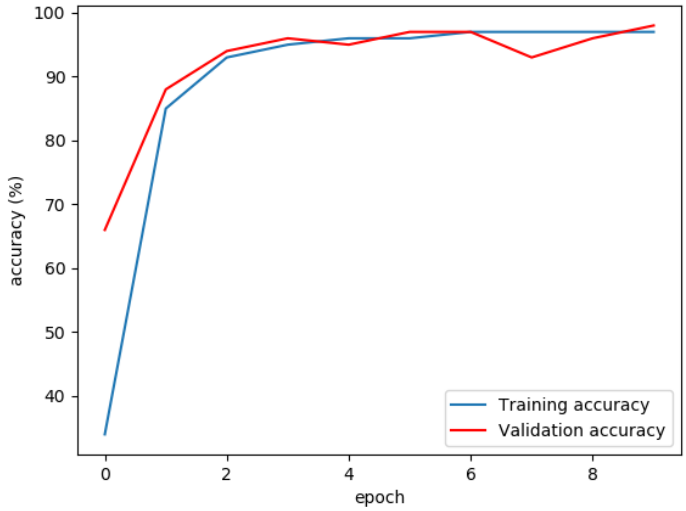


Figure 14: Training and validation accuracy of the convolutional model of Section 2

For similar number of parameters the CNN here performs better than the MLP in problem 1. Its accuracy is 98% compared to 97.6%. This is an important improvement given that we did not perform hyperparameter tuning as in the case of MLP. Also, this CNN architecture is simple, compact and was designed with little manipulation.

3 Problem 3 (Kaggle challenge)

3.1 Architecture of the model

The model's architecture that we have used is inspired from VGG model with 16 layers, 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.
- The number of filters goes from 64 to 640 accross the layers
- 5 max pooling with kernel of size (2,2) and stride of 2.
- 2 fully connected layers with 4096 hidden units
- 1 output layer

Layer (type)	Output Shape	Param #
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
zero_padding2d_12 (ZeroPaddi	(None, 6, 6, 640)	0
conv2d_12 (Conv2D)	(None, 4, 4, 640)	3687040
zero_padding2d_13 (ZeroPaddi	(None, 6, 6, 640)	0
conv2d_13 (Conv2D)	(None, 4, 4, 640)	3687040
max_pooling2d_5 (MaxPooling2	(None, 2, 2, 640)	0
flatten_1 (Flatten)	(None, 2560)	0
dense_1 (Dense)	(None, 4096)	10489856
dense_2 (Dense)	(None, 4096)	16781312
dense_3 (Dense)	(None, 2)	8194

Total params: 45,238,466
Trainable params: 45,238,466
Non-trainable params: 0

Figure 15: Model architecture

The total parameters of this model is $45,238,466$, 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.

3.2 Learning curves

Figures 16 and 17 show respectively the training and validation loss and accuracy per epoch. Before starting the training phase, we have split the input images randomly to training and validation sets with a ratio of $80\% - 20\%$ respectively. We have also used data augmentation strategy to improve the variance of the model. The transformation used were:

- rescale by up to 1/255
- width shift by up to 0.2
- height shift by up to 0.2
- zoom by up to 0.5
- shear by up to 0.2,
- horizontal flip

The learning curves above show that the model reaches its optimal value around epoch 250. After that point, the model starts slightly 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 250 epochs without overfitting. Additionnaly, if we have used a K-Fold crossvalidation we could have a better result as the model will be trained on all images given as input.

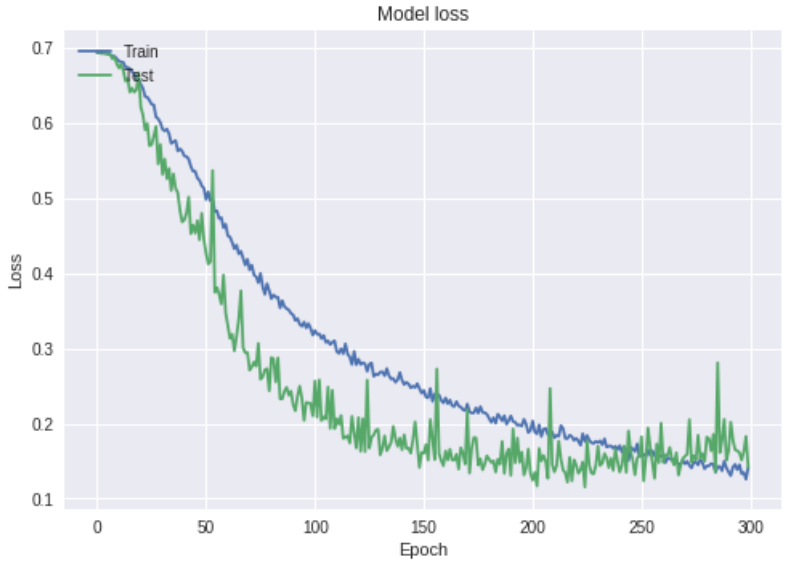


Figure 16: Loss of the model by epoch

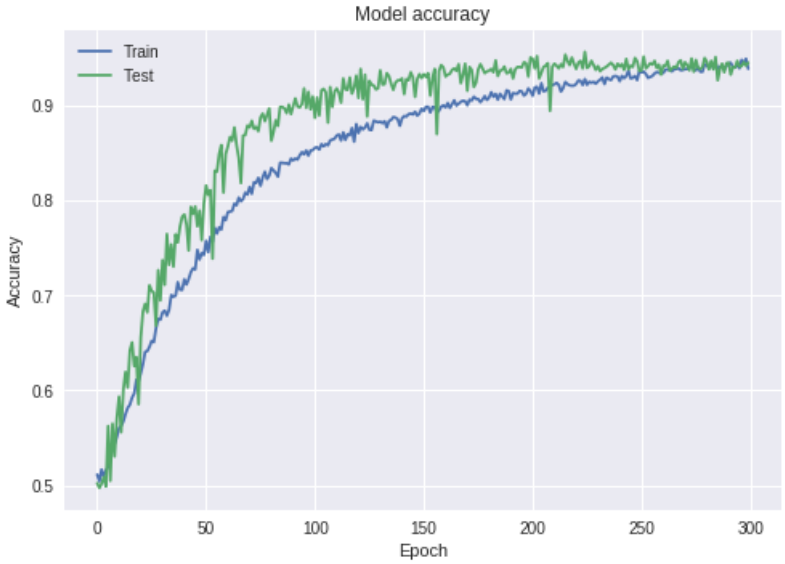


Figure 17: Accuracy of the model by epoch

3.3 Hyperparameters settings

In order to get a better performance, we have to choose the most relevent values for the hyperparameters during validation phase. That is, computing the accuracy and the loss of the model on validation set for multiple values of these hyperparameters. The hyperparameters that we have decided to tune are:

- Number of hidden layers
- Size of the kernels
- Learning rate
- Batch size

We will give for each of these hyperparameters the accuracy and the loss of the model while keeping the others fixed. The Table 1 shows the performance of the model with different number of convolutional layers. We notice that a model has generally a better performance when it

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	0.9445	0.1501	39,896,898
19	0.9304	0.1922	45,206,594

Table 1: Number of layers tuning

has more hidden layers. This is normal because more layers mean bigger capacity and then better result. However, this is not always true, that is a deeper model is more likely to overfit

especially when there is no regularization that controls it. In fact, the VGG-19 has overfitted the data in comparison with VGG-16, given the same amount of training data. The next Table 2 gives a comparison of the model's performance using different kernel sizes. This hyperparameter influence drastically the architecture of the model, that is a bigger kernel shorten the depth of the model and it could not be deeper unless we add 0 padding. 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

Table 3 gives a comparison of the model's performance as a function of the batch size. We notice that a small batch size push the model to make more iterations, and thus the SGD has a bigger chance to converge with enough epochs. Wheras, with a larger batch size, the SGD makes less iteration and may not get its minima within the same number of epochs.

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: Batch size tuning

And finally, Table 4 gives a comparison of the model's performance as a function of the learning rate. We notice, that the model converges slowly with a small learning rate and faster with a relatively bigger one like 0.03.

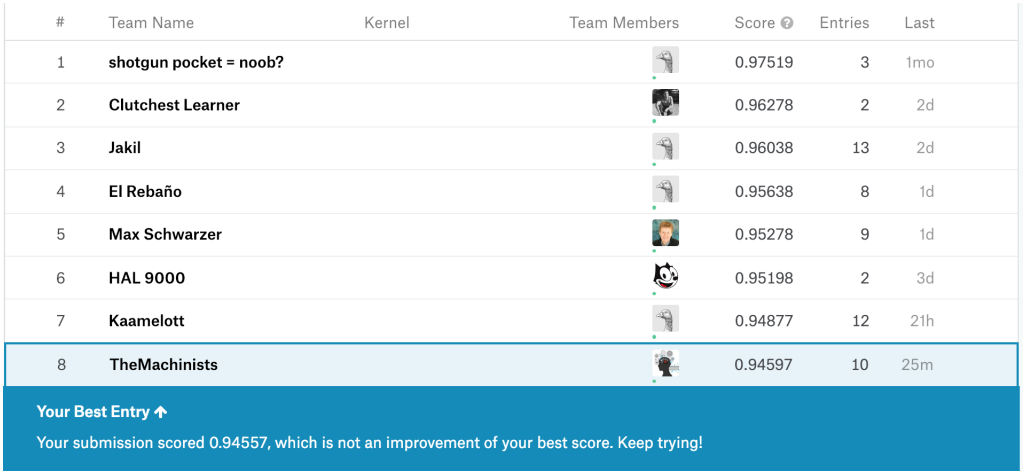
Learning rate	Accuracy	Loss
0.03	0.8935	0.2417
0.01	0.8595	0.3279
0.003	0.6930	0.5981
0.001	0.5825	0.6709

Table 4: Learning rate tuning

According to our hyperparameter search, we set the final model as follows:

- Number of hidden layers: 13 convolutions and 3 FC
- Size of the kernels: (3,3)
- Learning rate: 0.015
- Batch size: 16
- Number of epochs: 300

The accuracy on the public leaderboard was around 94% (see Figure 18), which is nearly the same on validation set 95%. However, the accuracy, on the private leaderboard, was a little lower 93% (see Figure 19). This result shows that the model had a good training and has not overfit on training set. Moreover, the data augmentation was a crucial action to improve the model's performance. As we haven't used any regularization strategies, on the training phase, we have used the early stopping strategy, that is taking the best model's parameters before it starts overfitting.

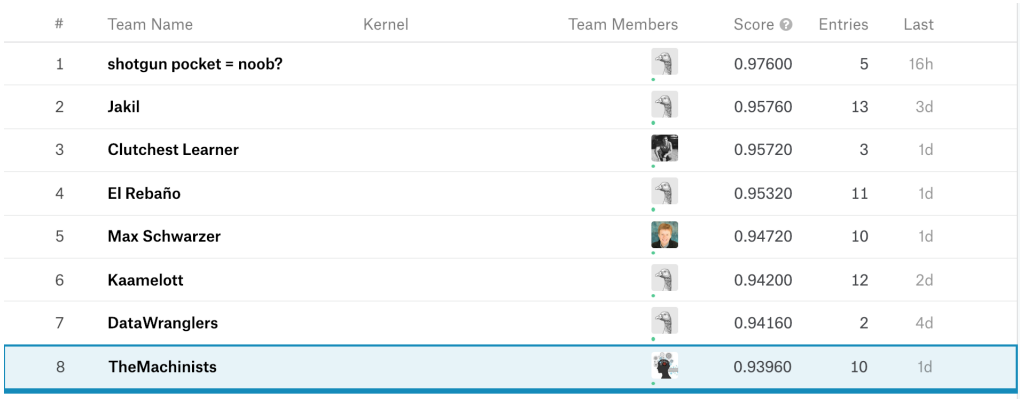


#	Team Name	Kernel	Team Members	Score	Entries	Last
1	**shotgun pocket = noob?**			0.97519	3	1mo
2	**Clutchest Learner**			0.96278	2	2d
3	**Jakil**			0.96038	13	2d
4	**El Rebaño**			0.95638	8	1d
5	**Max Schwarzer**			0.95278	9	1d
6	**HAL 9000**			0.95198	2	3d
7	**Kaamelott**			0.94877	12	21h
8	**TheMachinists**			0.94597	10	25m

Your Best Entry ↑

Your submission scored 0.94557, which is not an improvement of your best score. Keep trying!

Figure 18: Rank at the public leaderboard



#	Team Name	Kernel	Team Members	Score	Entries	Last
1	**shotgun pocket = noob?**			0.97600	5	16h
2	**Jakil**			0.95760	13	3d
3	**Clutchest Learner**			0.95720	3	1d
4	**El Rebaño**			0.95320	11	1d
5	**Max Schwarzer**			0.94720	10	1d
6	**Kaamelott**			0.94200	12	2d
7	**DataWranglers**			0.94160	2	4d
8	**TheMachinists**			0.93960	10	1d

Figure 19: Rank at the private leaderboard

3.3.1 Feature map visualization

The feature map visualization gives in some sens an intuition on how a CNN learns from images. Across the layers, feature map go from high level pattern to low level as we go deeper in the model. The following image shows for each layers the feature map after each activation layer. The code used to display these images was taken from the blog [2].

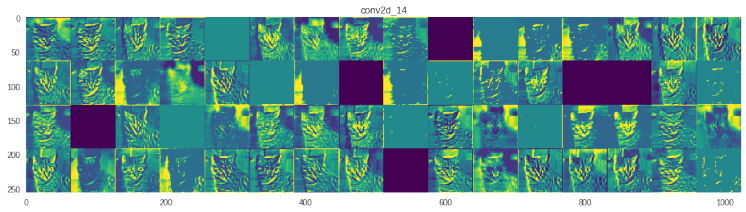


Figure 20: Feature map after conv layer 1

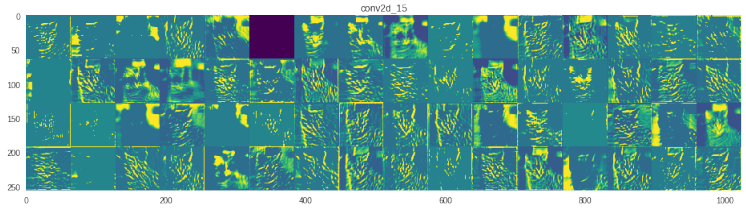


Figure 21: Feature map after conv layer 2

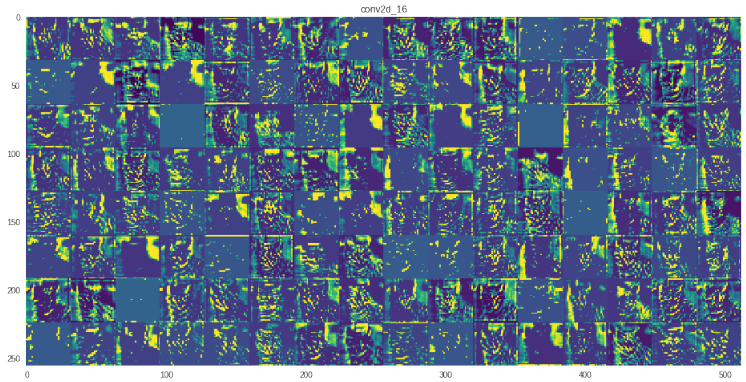


Figure 22: Feature map after conv layer 3

3.3.2 Performance of the model

The model gets 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.

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

(a) The model clearly misclassified images:
In the Figure 23, we can see 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.

(b) The model predicts, around 50% on both classes:
In the Figure 24 we can see some images that the model have bearly misclassified them.

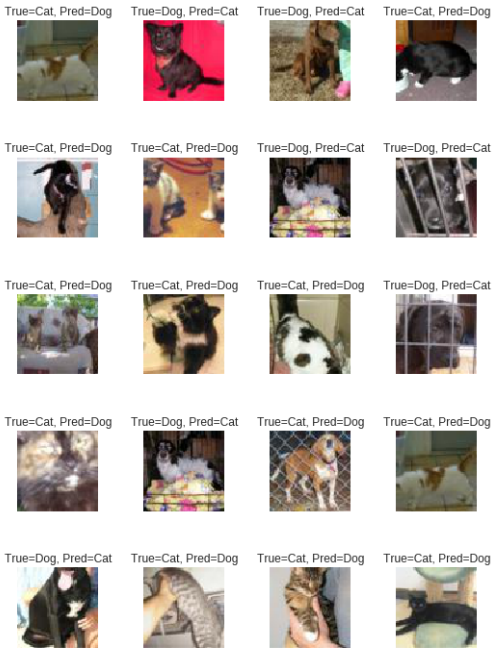


Figure 23: 100% miclassification

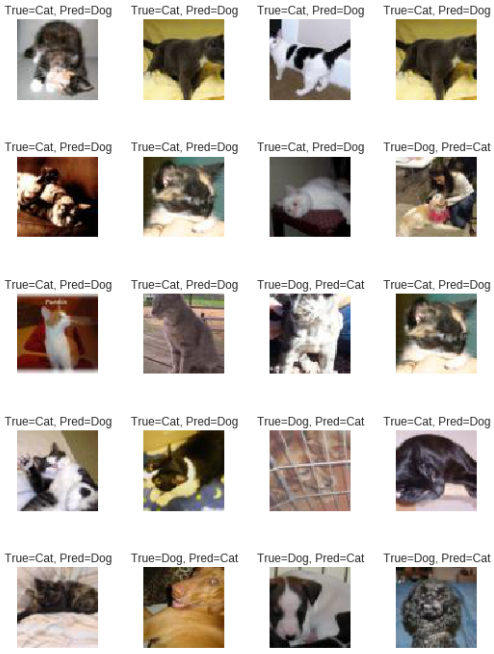


Figure 24: Around 50% miclassification

We notice the same pattern as the item (a) where the animals took different position in the picture with other objects or half captured.

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

[1] Jayadevan Thayumanav: Why don't we initialize the weights of a neural network to zero?, 2018.
https://www.quora.com/Why-dont-we-initialize-the-weights-of-a-neural-network-to-zero

[2] Applied Deep Learning - Part 4: Convolutional Neural Networks
https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584b

[3] TheMachinists github repository: Representation-Learning
https://github.com/faresbs/Representation-Learning.git