IFT 6135 **Assignment 1 (Practical)**

Kaggle team name: *Équipe de l'humour*

Team members:

- Clément Jumel
- Vincent Mallet
- Basile Dura
- Julien Horwood - 1198276

The code for this report can be found here.

1 Problem 1

In order to use the MNIST dataset, we had to make a first step of preprocessing, which consists in running the `data_generator.py` file.

1.1 Building the Model

1. Using $h^1 = 700$ and $h^2 = 300$, and considering the biases as well, we get a total number of parameters equal to:

$$\begin{aligned}(h^0 + 1) \times h^1 + (h^1 + 1) \times h^2 + (h^2 + 1) \times h^3 &= 785 \times 700 + 701 \times 300 + 301 \times 10 \\ &= 762\ 810\end{aligned}$$

2. See the code for this question.
3. See the code for this question. Note that during the training, we monitor the accuracy on the validation set and we return only the network that achieves the best performance on this quantity.

1.2 Initialization

In this part, we use as parameters the values specified in table 1. Unless stated otherwise, we will use these parameters in the following. These standard parameters lead to an accuracy on the test set of 0.9426 (trained for 10 epochs, with "Glorot" initialization). Note that these parameters are not those who lead to the best results (as we will see later), however they have been chosen arbitrarily and are very satisfying.

1. See the code for this question.
2. Note that initializing both bias and weights to zero means that the network cannot learn anything since the gradient of the ReLU function is 0 whenever $x \leq 0$, hence the flat curve in the case of the "Zero" initialization. This is confirmed by the norm of the weights remaining always 0.

Parameter	Value
Activation	ReLU
h^1	700
h^2	300
Batch size	16
Learning rate (α)	0.01
Regularization (λ)	0.1

Table 1: Standard parameters

The "Normal" and "Glorot" initializations, however, seem to perform similarly, when monitoring the training loss. This idea is confirmed when we check the accuracy of these networks on the test set: the "Zero" initialization has an accuracy of 0.101, which is basically chance, whereas the "Normal" and "Glorot" initializations lead to an accuracy of 0.9423 and 0.9426 respectively, which are very satisfying.

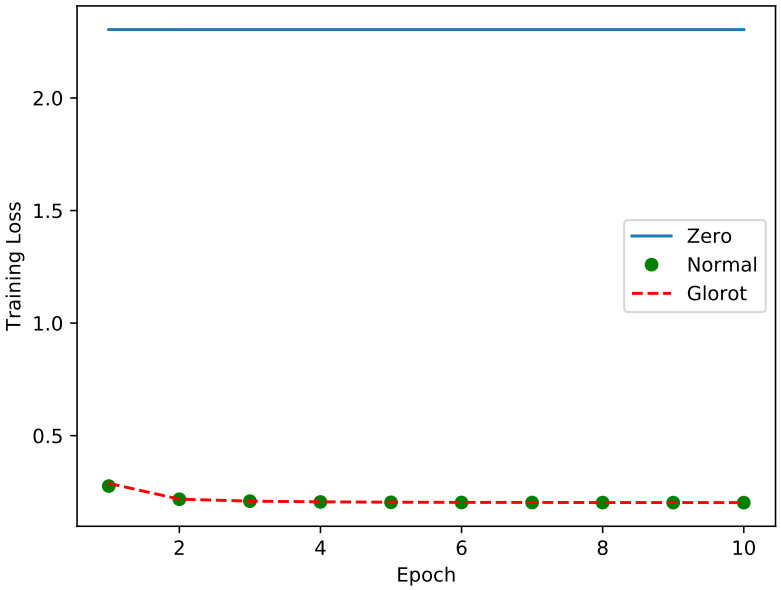


Figure 1: Comparison of the different initializations

1.3 Hyperparameter search

1. To find parameters that achieve good performances, we performed a random search (which is more efficient and computationally less expensive than a grid search). To find a good model, we ran the search for four hours and tested several dozens of models. See the `code` to see the exact parameters of the search (default parameters of the corresponding function).
2. The parameters we found that performed best are presented in table 2, and achieve an accuracy on the test set of 0.9734.

Parameter	Value
Activation	tanh
h^1	228
h^2	213
Batch size	32
Learning rate (α)	0.01
Regularization (λ)	0.005

Table 2: Best parameters found during the random search

1.4 Gradient validation using finite differences

1. See the code for this question.
2. Figure 2 shows the difference between the finite difference and the true gradient. It shows that our computation of the gradient is correct. Note that the difference rapidly falls to 0 when ε goes to 0, which indicates that for small ε, the approximation works well. However, for larger ε, the finite approximation is no longer valid.

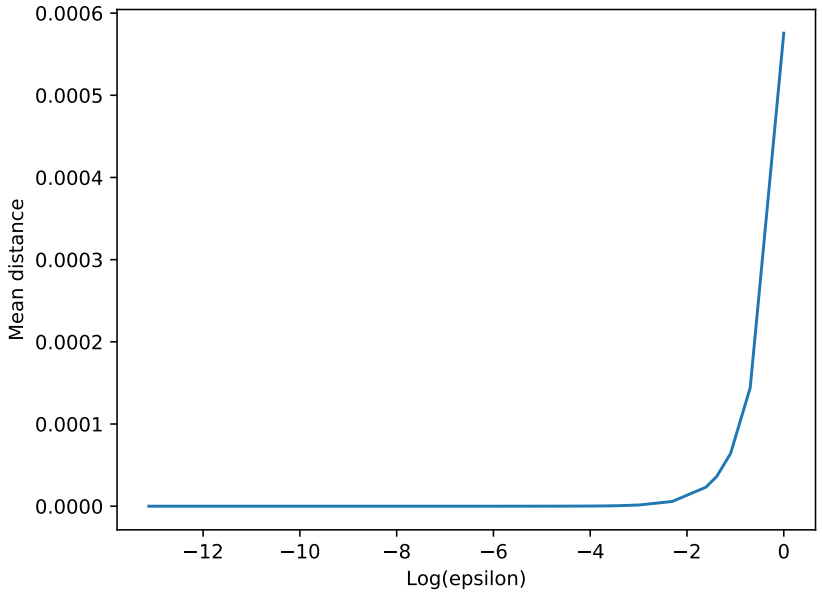


Figure 2: Maximum difference between the gradient and the finite difference as a function of ε

2 Problem 2

1. The architecture used for the CNN is explained in table 3. You can also look at its Pytorch definition in the code.

   This network uses four convolution layers with 3×3 kernels and same padding. The number of channel increases with the depth of each layer, such that the last convolution

layer has an output containing 128 channels.

Layer	Kernel	Padding	Channels	Input Size	Output Size
1st Convolution	3×3	1	16	$1 \times 28 \times 28$	$16 \times 28 \times 28$
2nd Convolution	3×3	1	32	$16 \times 28 \times 28$	$32 \times 28 \times 28$
3rd Convolution	3×3	1	64	$32 \times 28 \times 28$	$64 \times 28 \times 28$
Max Pool	2×2	—	—	$64 \times 28 \times 28$	$64 \times 14 \times 14$
4th Convolution	3×3	1	128	$64 \times 14 \times 14$	$128 \times 14 \times 14$
Max Pool	2×2	—	—	$128 \times 14 \times 14$	$128 \times 7 \times 7$
Dense Layer	—	—	—	$128 \cdot 7 \cdot 7$	10

Table 3: Architecture of the CNN

Along the forward pass, we use max pooling twice in order to reduce the dimension of the signal.

The final output is given by a dense layer, connecting the last convolution layer ($128 \times 7 \times 7$ neurons) to a softmax classification layer of ten neurons.

The number of parameters in this network is:

$$\begin{aligned} n &= 1 \cdot 3 \times 3 \cdot 16 + 16 \cdot 3 \times 3 \cdot 32 + 32 \cdot 3 \times 3 \cdot 64 + 64 \cdot 3 \times 3 \cdot 128 + 128 \cdot 7 \cdot 7 \times 10 \\ &= 159632 \end{aligned}$$

Which is in the same order of magnitude as for our multi-layer perceptron (MLP) model.

2. Figure 3 shows how the performance of the convolutional network evolves during training.

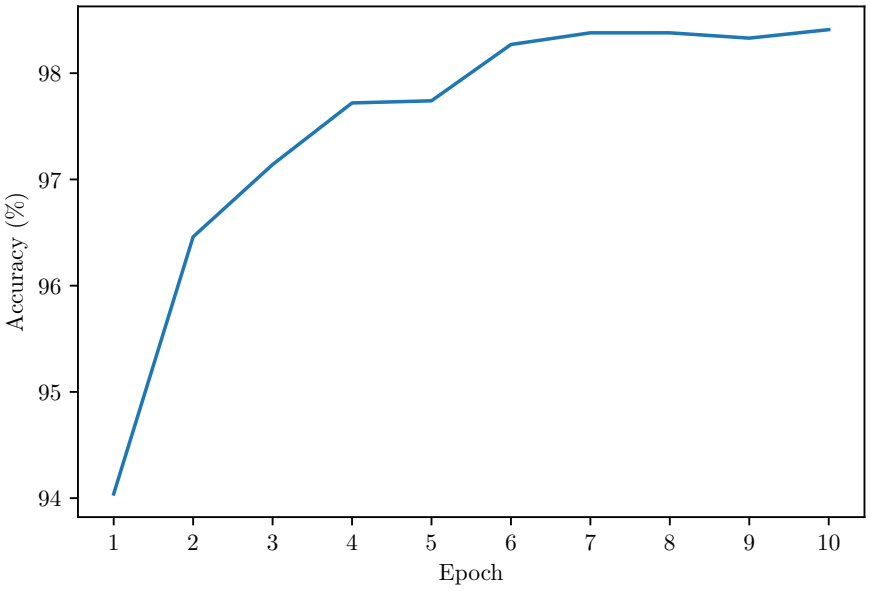


Figure 3: Validation accuracy of the CNN during training

The accuracy reaches 98.41% after 10 passes over the whole dataset.

Note that without any hyperparameter tuning, and with much less parameters (although within the same order of magnitude), this simple CNN model beats our best MLP quite

easily. This stems from the fact that, unlike the MLP, the CNN network is able to preserve and leverage the two-dimensional structure of the input image, which leads to better results.

3 Problem 3: Kaggle competition

The code for the Kaggle competition can be found here.

3.1 Model architecture

The architecture used for our final CNN model is presented in table 4. Note that the input and output sizes do not take into account the batch size.

Layer	Kernel	Padding	Channels	Input Size	Output Size
1st Convolution	3×3	1	16	$3 \times 64 \times 64$	$64 \times 64 \times 64$
2nd Convolution	3×3	1	128	$64 \times 64 \times 64$	$128 \times 64 \times 64$
3rd Convolution	3×3	1	256	$128 \times 64 \times 64$	$256 \times 64 \times 64$
Max Pool	2×2	—	—	$256 \times 64 \times 64$	$256 \times 32 \times 32$
4th Convolution	3×3	1	128	$256 \times 32 \times 32$	$128 \times 32 \times 32$
1st Dense Layer	—	—	—	$128 \cdot 32 \cdot 32$	500
2nd Dense Layer	—	—	—	500	100
3rd Dense Layer	—	—	—	100	2

Table 4: Architecture of the CNN

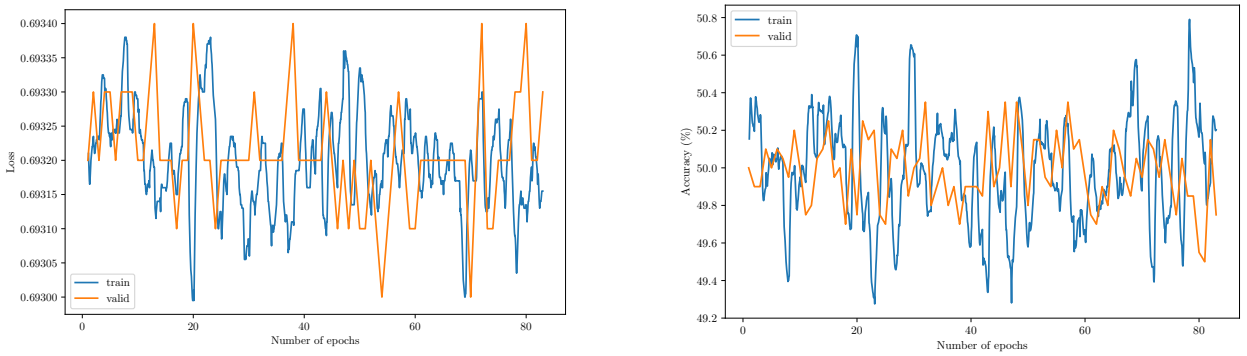
This medium-sized convolutional network contains four convolutional layers, which increase the number of channels to a maximum of 256. These layers are interspersed with a Max Pool layer and followed by three dense layers, ending with a softmax activation that can be interpreted as a logistic regression on the features extracted by the network. The total number of parameters resulting from this architecture is 66252658.

In order to obtain this architecture, we performed multiple tests on various both simpler and more complex architectures. This, in conjuntion with the various training parameters we tested, comprised our model selection and hyperparameter search process.

3.2 Model Selection

We initially had great difficulty in training models with any significant depth. Indeed, our baseline model of two convolutional layers and a total of 3282190 trainable parameters was the only model which made progress under our initial training regime, and yet was only able to achieve an accuracy marginally above 60% on the validation set. Bigger networks were then performing extremely well with modern optimizers such as Adam, but training them with vanilla SGD proved extremely tricky.

Our first breakthrough in circumventing these issues was normalizing our data. We did so by standardizing images across channels. This improved performance accross the board, and allowed our models to train. However, the baseline still outperformed larger models which were training extremely slowly. Indeed, under a normalized data regime, our baseline achieved 78% accuracy on the validation set, while other much larger models struggled to break above 70% after 200 epochs. We can intuitively justify this improvement by the fact that normalization makes our data space better behaved, with less variability across the image set. This in turn may make the optimization process easier and less likely to fall in various local minima and saddle points.



(a) Cross-entropy loss

(b) Classification error

Figure 4: One under-performing network...
(2-layer CNN, no normalization, batch size of 100)

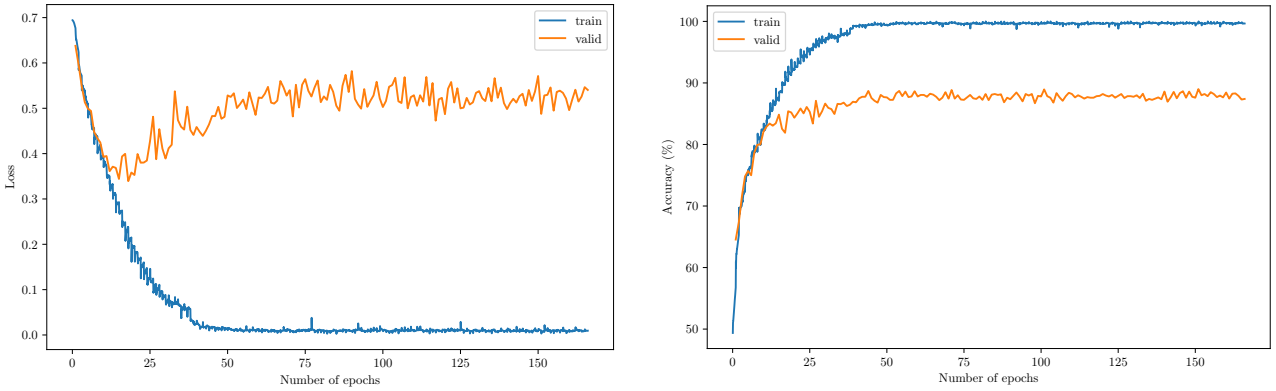
The second significant improvement, which allowed us to train deeper models effectively, came from tuning the batch size. Indeed, under our initial batch size regime of 128, deeper models learned extremely slowly. By reducing this hyperparameter to 10, our final archictecture (described above) reached 89% accuracy on the validation set. We hypothesize this to be because a smaller batch size encourages more exploration by performing a larger number of updates based on higher variance estimates of the loss during training. We summarize the results of our tests in the table below. The large CNN corresponds to `meganet.py` in the code repository. We show the best results achieved for each hyperparameter setting across all epochs. We used an initial learning rate of 10^{-2} to encourage early exploration coupled with a scheduler to diminish the learning rate over time.

We can see in the table how the reduction of batch size coupled with normalized data allowed us to train much more powerful models, and eventually surpass our previously superior baseline model. However, we were yet unable to fully take advantage of very deep architectures, showing the limitations of a simple SGD optimizer. Despite this, we significantly improved our validation set accuracy to a very reasonable level. Furthermore, our results in the competition submission indicate that the validation set accuracy was a very good estimate of our true accuracy, considering our 88.475 % accuracy on the test subset.

Figure 5 shows the training curves of both classification error and cross-entropy loss (the quantity being minimized) for our final architecture. The overfitting regime is exemplified here: both training error and loss go to zero as the number of epochs grows, while on the validation set, these quantities quickly stagnate, and even go up in the case of the validation loss.

Hyperparameter	Model	Validation Accuracy (%)
Batch size = 128, No normalization	Baseline	50.35
	Final architecture	51.00
	Large CNN	48.95
With Normalization	Baseline	78.3
	Final architecture	63.55
	Large CNN	48.95
Normalization, Batch size = 10	Baseline	83.15
	Final architecture	89.00
	Large CNN	75.6

Table 5: Hyperparameter tuning results



(a) Cross-entropy loss

(b) Classification error

Figure 5: Evolution of training and validation error metrics during training

In order to curb this phenomenon, we could have used regularization techniques such as dropout or batch normalization. These would very likely have helped generalize better and improve our model's validation set accuracy to a certain extent. Furthermore, as mentioned previously, use of an optimizer with momentum, such as Adam, would have allowed our deeper architectures to learn much more efficiently.

3.3 Visual Analysis

To gain some insight into the behaviour of our model, it can be useful to visualize where it goes wrong. In order to do so, we show examples of where our model misclassifies images with a high degree of confidence.

Figure 6: Images incorrectly classified with more than 95 % probability

Although explaining these errors is mostly speculative, one can hypothesize that our model

may have learned "pointy ears" as a key feature for classifying cats. The image of a dog on the right with pointed ears may thus have confused our classifier. Other aspects such as image truncation (left), noisy items (flowers) or varying scale can also provide some hints as to the source of error. In order to curb some of these effects, we attempted augmenting data via random cropping and resizing of our images, random rotations and flipping. However it may be difficult for the model to generalize these notions very well beyond the training set without regularization.

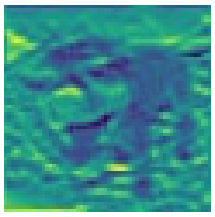
Similarly, Figure 6 provides examples of high uncertainty for our model.

(a) 62% probability of dog

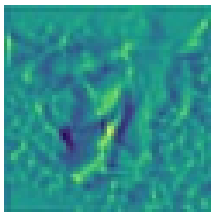
(b) 40% probability of dog

Figure 7: Images classified with around 50 % probability

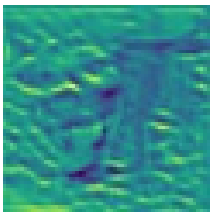
As a final means of visualizing our network , we plot the feature maps obtained for a given image to gain some insights into the behaviour at various points in the forward pass of our trained weights. Specifically, we show the first kernel at each convolutional layer for a given image.



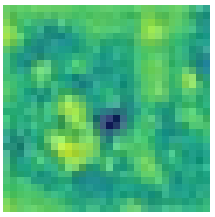
(a) Layer 1



(b) Layer 2



(c) Layer 3



(d) Layer 4

Figure 8: Feature maps for a given image through layers of the network

We notice that the feature map appears much more interpretable in the first layers. Indeed, this can be understood by the fact that these kernels are much closer to the raw image pixels. As the data flows through the computation graph, these kernels become increasingly abstract. In the final layer, the feature map seems to be highlighting patches rather than providing any humanly discernible shapes.