IFT 6135 - Homework 1

Antoine Chehire, Philippe Marchandise, Adel Nabli

14/02/2019

Link of the Github where the code used is stored: `https://github.com/achehire/Deep_1_AC_AN_PW`

1 Problem 1

We want to build a MLP with two hidden layers with h^1 and h^2 hidden units. The data is given in the form of a design matrix $X \in \mathbb{R}^{n \times 784}$ and $\forall i \in [\![1, n]\!], y^i \in \mathbb{R}^{10}$ is the one-hot encoding of the label. The output layer is parametrized by a $softmax$ function $s(z)_c = \frac{e^{z_c}}{\sum_c e^{z_c}}$.

- We choose the sigmoid function as a non linearity for both layers. We have: $\sigma(z) = \frac{1}{1+e^{-z}}$ and $\sigma'(z) = (1 - \sigma(z))\sigma(z)$.
- Thus, our NN is computing, for each example $x \in \mathbb{R}^{784}$:

$$A_1 = \sigma(W_1 x + b_1) \; with \; W_1 \in \mathbb{R}^{h_1 \times 784} \; and \; b_1 \in \mathbb{R}^{h_1}$$
$$A_2 = \sigma(W_2 A_1 + b_2) \; with \; W_2 \in \mathbb{R}^{h_2 \times h_1} \; and \; b_2 \in \mathbb{R}^{h_2}$$
$$O = s(W_3 A_2 + b_3) \; with \; W_3 \in \mathbb{R}^{10 \times h_2} \; and \; b_3 \in \mathbb{R}^{10}$$

- The **cross entropy loss** is defined by:

$$l(y, O) = -\sum_{c=1}^{10} y_c \log(o_c) \quad with \; y_c = 1_{label=c}$$

  which leads to the following **regularized empirical risk** if we use an L^2 regularization:

$$\hat{R} = \frac{-1}{n} \sum_{i=1}^{n} \sum_{c=1}^{10} y_c^i \log(o_c^i) + \lambda \sum_{j=1}^{3} ||W_j||_2^2$$

- Our purpose is to learn each parameter $\theta \in \{W_1, b_1, W_2, b_2, W_3, b_3\}$ by **mini batch** gradient descent incrementally:

$$\theta^{t+1} = \theta^t - \alpha_t \big(\frac{1}{|I_t|} \sum_{i_t \in I_t} \frac{\partial l}{\partial \theta}(x_{i_t}, y_{i_t}) + 2\lambda\theta^t \big) \quad for \; \theta \in \{W_1, W_2, W_3\}$$
$$\theta^{t+1} = \theta^t - \frac{\alpha_t}{|I_t|} \sum_{i_t \in I_t} \frac{\partial l}{\partial \theta}(x_{i_t}, y_{i_t}) \quad otherwise$$

  with $\alpha_t, I_t, x_{I_t}, y_{I_t}$ being respectively the learning rate, the mini-batch, the training examples and the corresponding labels at time t. We use the formula $\alpha_t = \frac{\alpha_0}{1+\delta t}$ for the learning rate, α_0 and δ being hyperparameters we will have to tune.

- For that, using the chain-rule and noting $\odot$ the Hadamard product, we derive:

$$\begin{aligned}
\frac{\partial l}{\partial b_3} &= O - y \\
\frac{\partial l}{\partial W_3} &= (O - y)A_2^T \\
\frac{\partial l}{\partial b_2} &= \frac{\partial l}{\partial A_2} \odot (1 - A_2) \odot A_2 \; with \; \frac{\partial l}{\partial A_2} = W_3^T \frac{\partial l}{\partial b_3} \\
\frac{\partial l}{\partial W_2} &= \frac{\partial l}{\partial b_2} A_1^T \\
\frac{\partial l}{\partial b_1} &= \frac{\partial l}{\partial A_1} \odot (1 - A_1) \odot A_1 \; with \; \frac{\partial l}{\partial A_1} = W_2^T \frac{\partial l}{\partial b_2} \\
\frac{\partial l}{\partial W_1} &= \frac{\partial l}{\partial b_1} x^T
\end{aligned}$$

- To initialize our parameters W_l with $l \in [\![1, 3]\!]$, we have three options:
  - **Zero**: we initialize with zeros
  - **Normal**: we initialize with $\mathcal{N}(0, 1)$
  - **Glorot**: we initialize with $\mathcal{U}(-d^l, d^l)$ where $d^l = \sqrt{\frac{6}{h^{l-1} + h^l}}$

1.1 Building the model

We decided to use the *sigmoid* function as a non linearity for both layers. We found that having $h^1 = 400$ and $h^2 = 700$ leads good results. In total, we thus have:

$$784 \times 400 + 400 + 700 \times 400 + 700 + 700 \times 10 + 10 = 601710$$

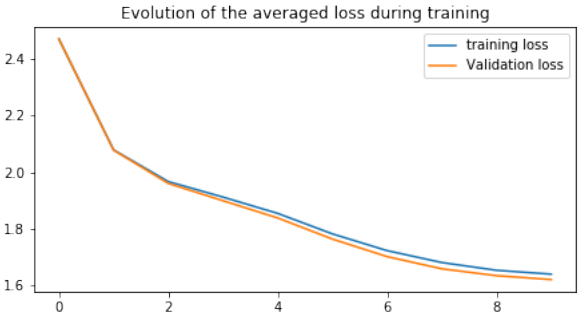
parameters to train, which is in the range $[\![0.5M, 1M]\!]$.

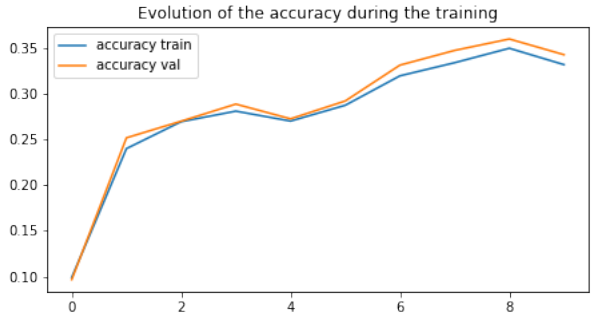
1.2 Initialization

In this section, we trained our model using the data from `http://deeplearning.net/data/mnist/` with the following hyperparameters:

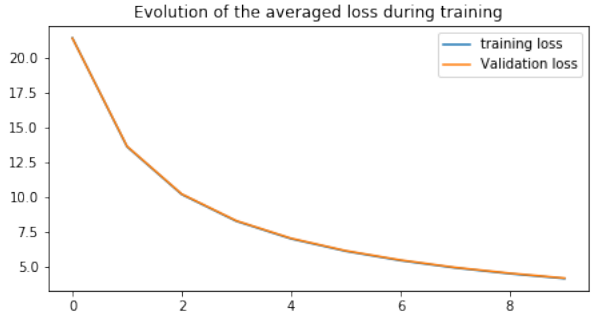
non linearity: sigmoid **h¹**: 400 **h²**: 700 α_0: 4 δ: 2e-3 **mini-batch size**: 32 λ: 0.0001

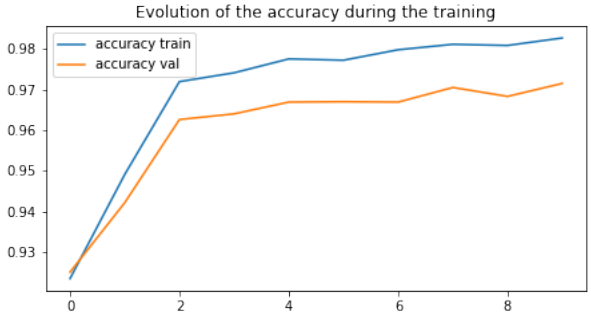
The results we get after training for 10 epochs with the 3 different types of initialization are as follows:



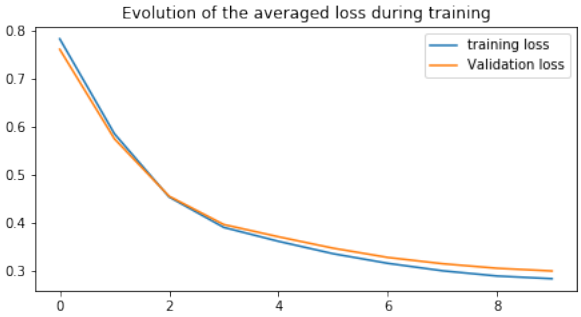


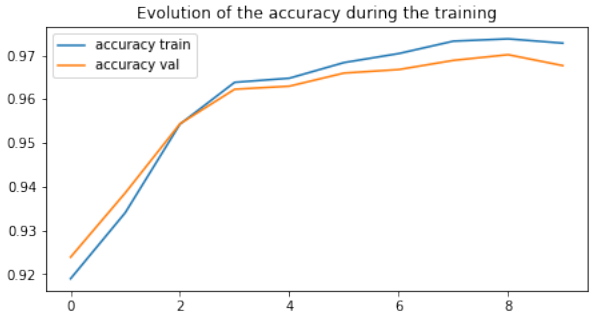
Results after initializing with **zero**





Results after initializing with **normal**





Results after initializing with **Glorot**

Seeing those results, we can deduce that the **fastest rate of learning with respect to the loss** is obtained using the **Glorot** initialization *(we obtain a loss smaller than 1 after the first epoch whereas the two other methods don't reach that after 10)*. Compared to the loss using the normal initialization, the loss with the zero initialization is smaller, but the accuracy shows us that the model actually perform way better with the normal initialization. This contrast may be due to the fact that the loss is computed with penalization and that we can expect having a stronger penalization on the weights of the normal initialization than the ones starting at zero. But as expected, looking at the accuracy, the model initialized with zeros doesn't seem to learn well *(accuracy stuck under 35%)* as the symmetry of the network isn't broken. The two other initialization create two good performing models *(accuracy over 96% after 10 epochs)*.

1.3 Hyperparameter search

With the hyperparameters listed above and the **normal** initialization we obtain an accuracy of 97,15% on the validation set after 10 epochs *(using the Glorot initialization, we pass 97% after the* 11^{th} *epoch)*.

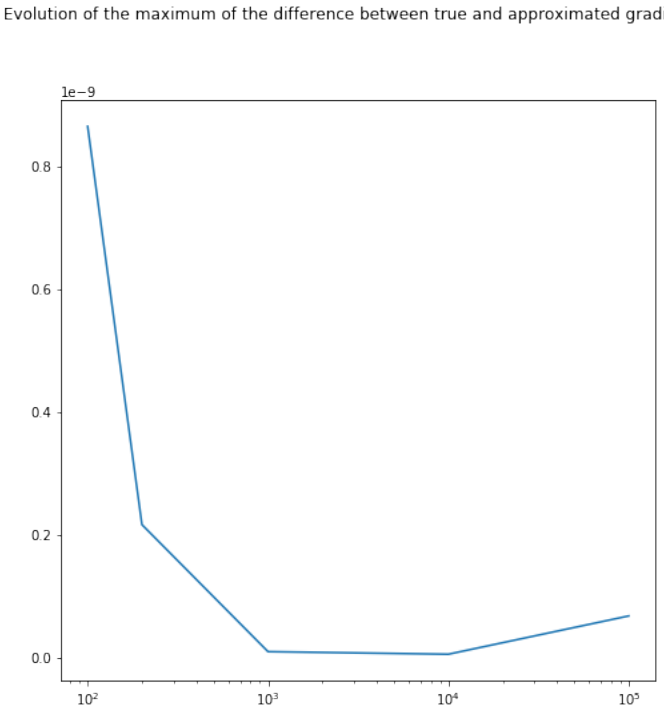
non linearity	$\mathbf{h^1}$	$\mathbf{h^2}$	**initialization**	**mini batch size**	α_0	δ	λ	**accuracy**
sigmoid	400	700	glorot	64	4	2e-3	0	97.17%
sigmoid	400	700	normal	32	4	2e-3	0.0001	97.15%
sigmoid	400	700	glorot	32	3	2e-3	0.0001	96.78%
sigmoid	400	700	glorot	32	4	2e-3	0.0001	96.77%
sigmoid	400	700	glorot	64	4	2e-3	0.0001	96.61%
sigmoid	400	700	zero	32	4	2e-3	0.0001	34.26%
sigmoid	400	700	glorot	64	4	2e-1	0.01	26.42%

A few of the hyperparameters tested. Accuracy on the validation set reported after 10 epochs.

1.4 Validate Gradients using Finite Difference

We consider the 10 first weights of the first row of W_2 and use the finite difference formula ∇_i^N to approximate the gradient of the loss with respect to those parameters $\frac{\partial L}{\partial \theta_i}$. We compare the approximated and the true gradient for different values of $\epsilon = \frac{1}{N}$ for $N \in \{100, 200, 1000, 10000, 100000\}$ and plot the graph of $\max_{1 \leq i \leq 10} |\nabla_i^N - \frac{\partial L}{\partial \theta_i}|$ for every value of N.

We computed the graph using the **first sample of the training set** and using the best performing model we had during the above hyperparameter search after 10 epochs *(first line of the table)*.

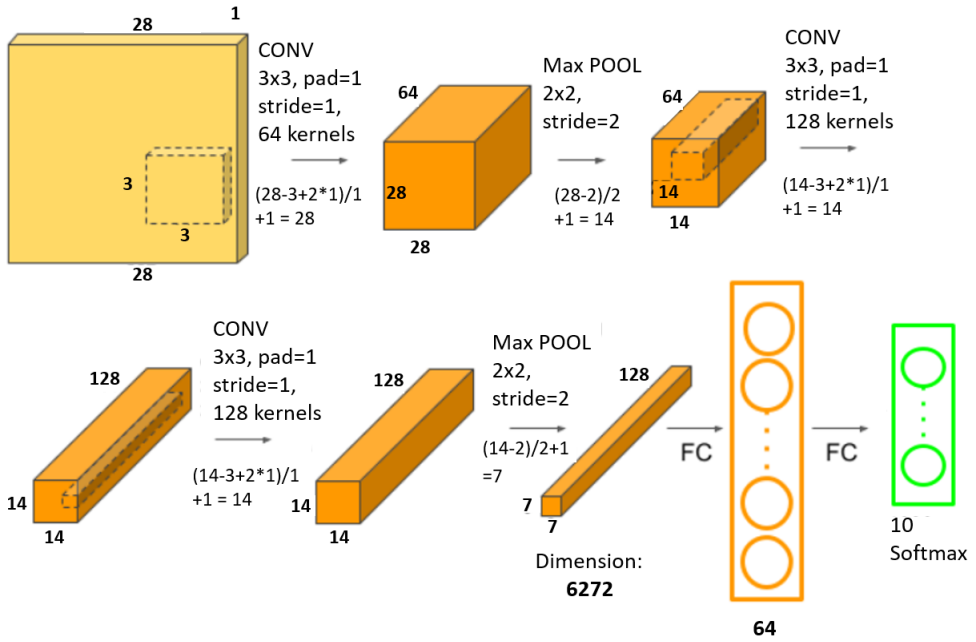


Thus, we can observe that the global tendency is that the difference between the true gradient and the finite difference one gets smaller as ϵ gets smaller, which is what we could have expected as the chord resembles more and more to the tangent as ϵ gets small.

2 Problem 2

We want to build a convolutional neural network to classify the hand drawn digits from the MNIST data set. In order to compare the results with those obtained in the first part, we will only train a model with similar number of parameters for 10 epochs.

2.1 The network's architecture



Detailed architecture of the CNN model

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 28, 28, 64)	640
max_pooling2d_3 (MaxPooling2	(None, 14, 14, 64)	0
conv2d_4 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_5 (Conv2D)	(None, 14, 14, 128)	147584
max_pooling2d_4 (MaxPooling2	(None, 7, 7, 128)	0
flatten_2 (Flatten)	(None, 6272)	0
dense_2 (Dense)	(None, 64)	401472
dense_3 (Dense)	(None, 10)	650

Total params: 624,202
Trainable params: 624,202
Non-trainable params: 0

Summary of the CNN model

2.2 Performance of the model

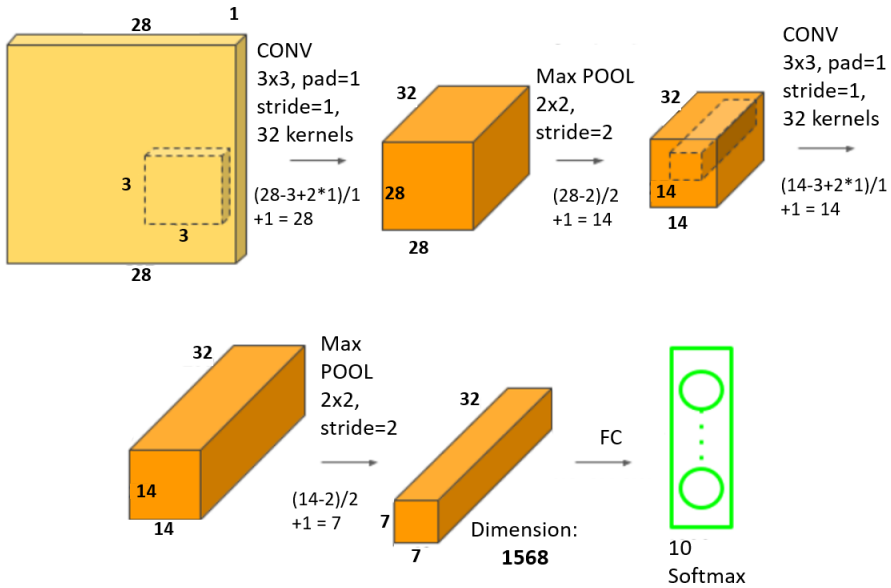
Our network has 3 convolutional layers and one fully connected layer. The detailed architecture can be found above. The model was initialized with Glorot weights. It was then trained with a fixed learning rate of 0.01. We report below the evolution of the training and validation accuracy and losses during training:

Accuracy and loss during training

Our CNN model reached the 97% accuracy on validation data on the second epoch. We can see here the power of convolutional neural networks when it comes to dealing with images. We built this model with 624202 parameters in order to have a similar number of parameters as in the previous part. However, we also trained a lighter model with only 25000 parameters (24.7 times fewer parameters than this model). The results were stunning. We reported the details of this lighter model in the next subsection.

2.3 Bonus: A lighter model

What we found most impressive was not the performances of the above model, but the performances of this model. Indeed, although it has only 25258 parameters, this model still reaches a validation accuracy of 97% in three epochs. Moreover, after 10 epochs, both this model and the one described in the previous subsection reach very similar performance. We found that this model describes best how powerful convolutional neural networks are when dealing with images. This is why we added the details about of this model in this section even though it was not required.



Detailed architecture of the CNN model

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 32)	9248
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 32)	0
flatten_1 (Flatten)	(None, 1568)	0
dense_1 (Dense)	(None, 10)	15690

Total params: 25,258
Trainable params: 25,258
Non-trainable params: 0

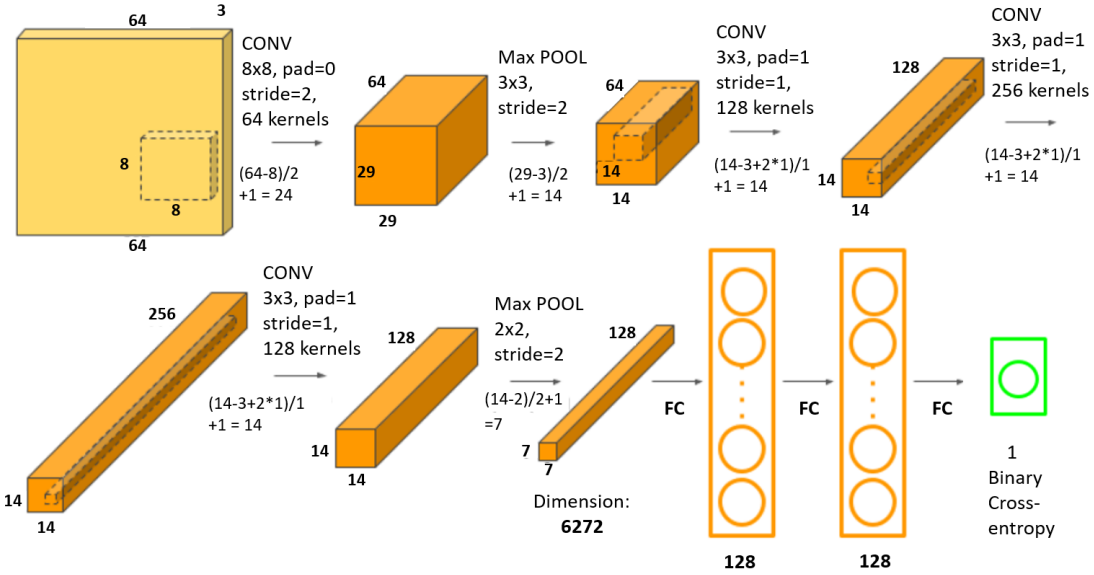
Summary of the CNN model

Accuracy and loss during training

3 Problem 3

This problem is about binary classification on images to determine if they're representing a cat or a dog. Since the inputs are images, we decided to use convolutional layers in our architecture as part 2 demonstrated how powerful they are.

3.1 The network's architecture



Detailed architecture of the CNN model

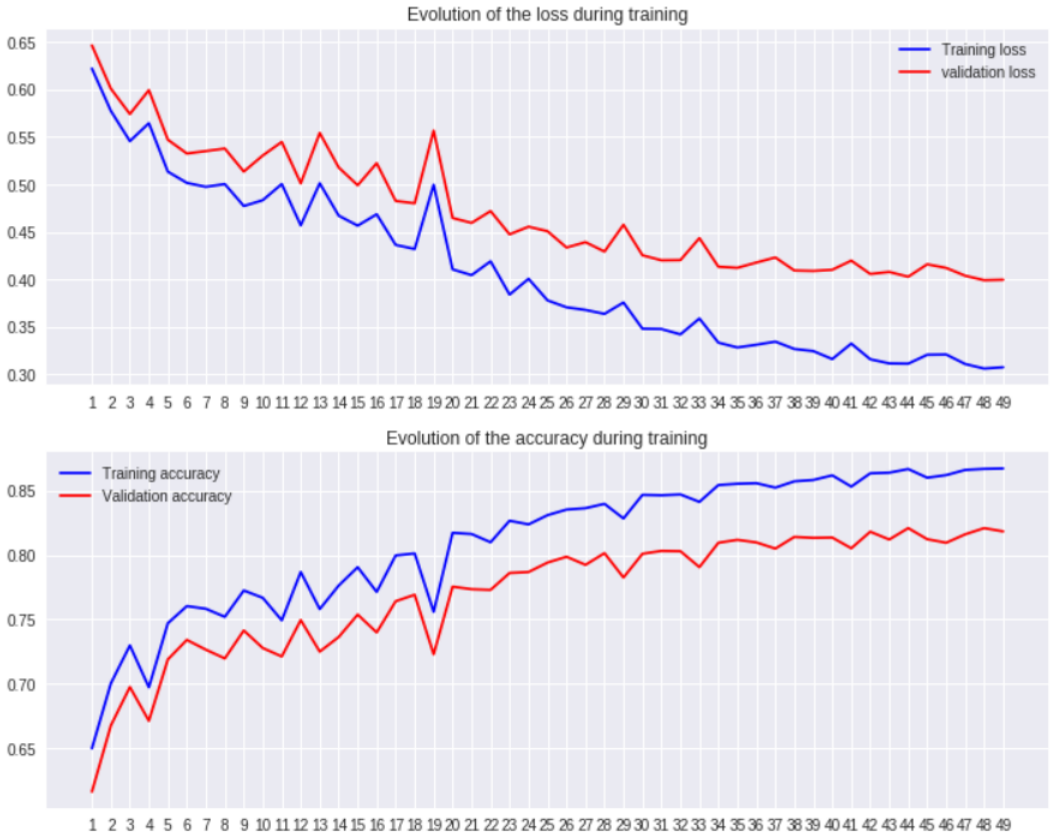
Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 29, 29, 64)	12352
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 64)	0
conv2d_2 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_3 (Conv2D)	(None, 14, 14, 256)	295168
conv2d_4 (Conv2D)	(None, 14, 14, 128)	295040
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 128)	0
flatten_1 (Flatten)	(None, 6272)	0
dense_1 (Dense)	(None, 128)	802944
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 1)	129

Total params: 1,496,001
Trainable params: 1,496,001
Non-trainable params: 0

Summary of the CNN model

3.2 Performance of the model

Our network has 3 convolutional layers and 2 fully connected layer. The detailed architecture can be found above. The model was initialized with Glorot weights. It was then trained with a decreasing learning rate on plateau when the validation loss increased. The initial learning rate was 0.0003, which is pretty low as we chose not to re-scale the images. Below are the evolution of the training and validation accuracy and losses during training:



Accuracy and loss during training

3.2.1 Comment on the curves

Please note that a better image is being created...

First of all, we notice that the training loss does not always decrease. At some epochs, the training loss even increases. There are two main reasons:

- We generate random images during training. Therefore, the images we actually train our model are different from epoch to epoch.
- As it was easier to implement, we chose to decrease the learning rate non continuously and make it a plateau function. This however, means that we recompile the Keras model every time we decrease the learning rate. Although recompiling the model does not change the weights, it does reset the optimizer. This means that the optimizer starts from scratch. As a result, for one or two epochs after decreasing the learning rate, the loss is actually higher than it was before.

We chose to keep the model that yielded the best results on the accuracy. We thus chose to keep the model with 48 epochs for our submission.

3.2.2 Preprocessing

In this sub section, we will detail our choices during preprocessing.

First of all, We intentionally removed 9 completely white images from the training data set as we found it could make the training harder. In order to obtain better results, we had to use data augmentation.

For each training image, we randomly generated 7 other images. We generated those images by slightly transforming the training images. Basically, for each training image, we applied:

- A random rotation between 0 and 15 degrees
- A random shearing between 0 and 10 degrees
- A random zoom between 0 and 20 percent
- A random horizontal flip
- A random height adjustment between 0 and 10 percent
- A random width adjustment between 0 and 10 percent

By doing so, we trained our model on a data set artificially 8 times as big. Our model is thus more robust and predicts more accurately the image it had yet to see.

Without this preprocessing, our model stopped improving after reaching approximately 75% accuracy. Thanks to the preprocessing, we were thus able to improve the performances of our model by 10% (relative improvement, not absolute).

3.2.3 How to improve the model

As we can see on the curves, there is an obvious gap between the training accuracy and the validation accuracy. In order to improve the validation accuracy, we could use some of the following techniques:

- Use dropout during training
- Apply some regularization on the loss function (None in our model)
- Generate random images that are even less close to the real training images
- Apply some more modifications for the data augmentation (like lightness filters or salt pepper noise)
- Use a continuous decreasing learning rate function that does not require to recompile the model
- Use a different optimizer (Adam)

3.2.4 Test results (on Kaggle)

Our full test accuracy is: 81.8% (82.9% on only half the test set). It is very similar to the validation accuracy of our model (82.13% for the 48^{th} epoch). Therefore, we can conclude that we did not overfit our model on the validation set.

3.3 Deeper analysis of the model

3.3.1 Hyperparameters tuning

For our hyperparameters, we mostly focused on 2 points:

- Learning rate: different initial values
- Model architecture

For the learning rate, we don't have that many results to report because we only kept the best one. When the validation accuracy decreased significantly (for a few epochs), we reloaded the weights we had a few epochs ago when the validation accuracy was higher and we trained it with a lower learning rate. Thus we don't have any real history of our experiments with the learning rate.

We tried many different architecture. In the Kaggle challenge, we submitted the results of one of our lighter model with data augmentation. The model had less than 280000 parameters but reached 76% on test accuracy. This result was only possible through data augmentation. Without data augmentation, the model struggled around 73% validation accuracy.

3.3.2 Analysis of feature maps

In order to analyze our feature maps, we passed an image to our network and we monitored the output of each layer. The results are shown in the notebook. Below, we chose to only focus on the results of the first layer as it is the easiest to interpret and as it would not be practical to represent them all. Moreover, we only focused on the red channel as they all yield very similar results.

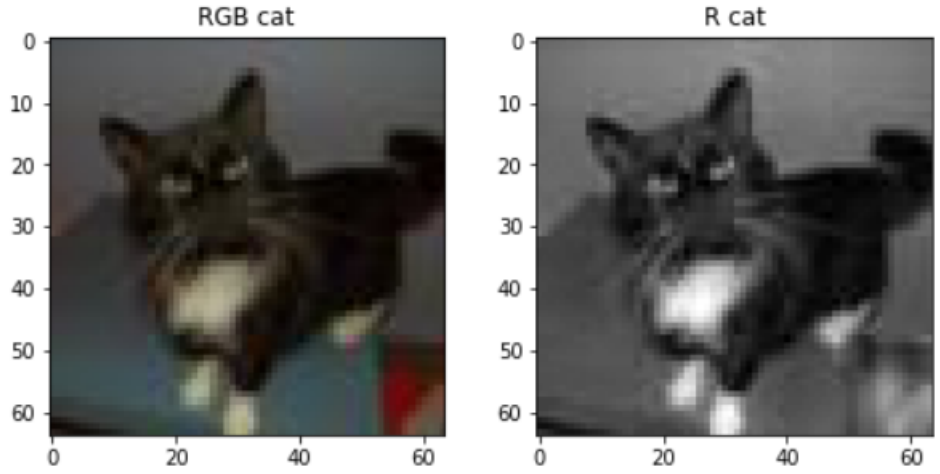
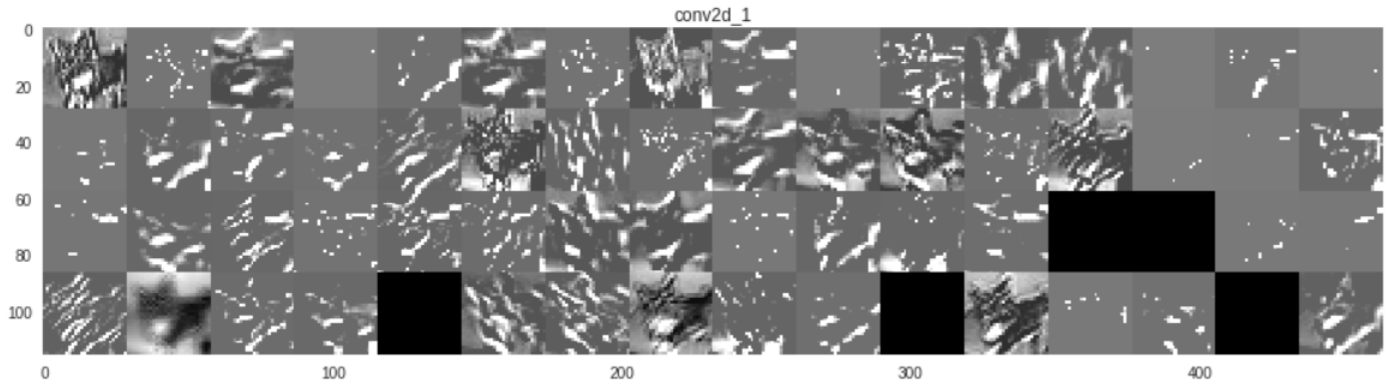


Image of the cat we fed our network



Feature map of the first convolutional layer for the red channel of the cat

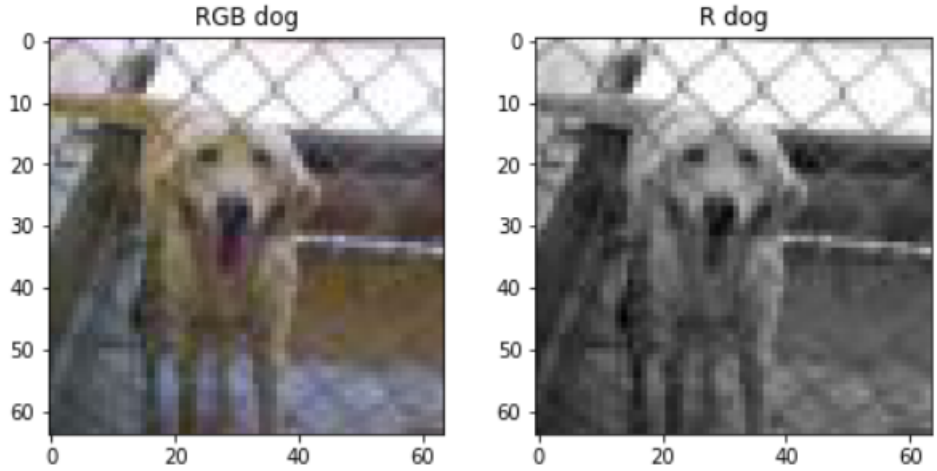
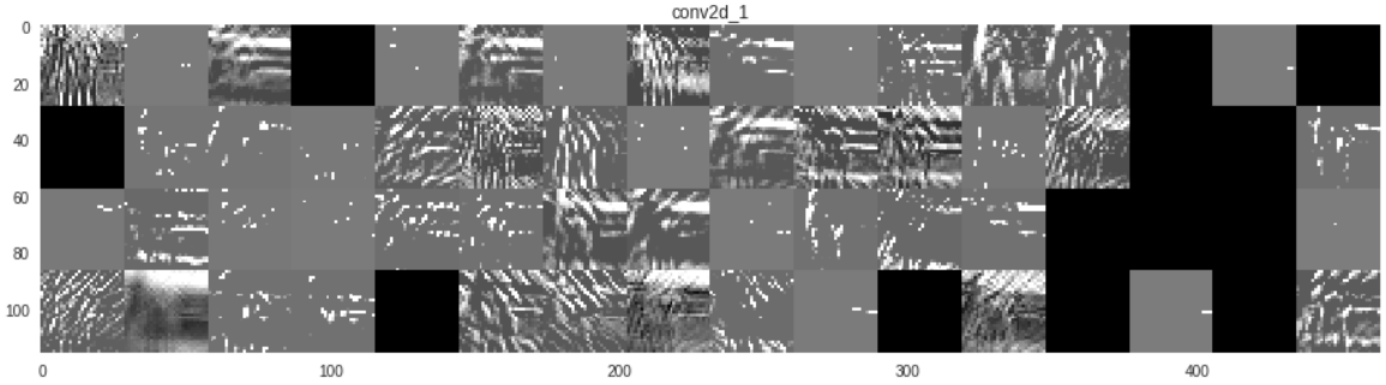


Image of the dog we fed our network



Feature map of the first convolutional layer for the red channel of the dog

As we can see, our model generates very different filters on the first layers. Some of them leave the picture nearly unchanged while others seem to only capture a direction.

We can also see that the output for the cat and the dog are very different. In particular some filters are completely blocked for the cat and not the dog. The opposite is also true.

3.3.3 Missclassified or hard to decide images

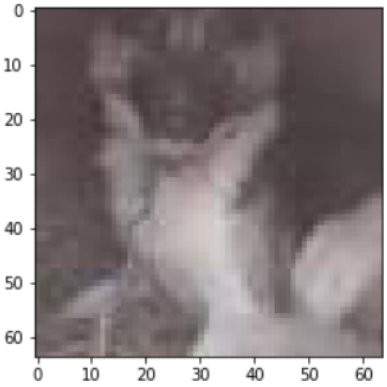
In this sub section we will report a few missclassified and hard to decide images.

In addition to the images that are difficult to classify from a human point of view, our model seems to struggle a bit on the darker images. We could improve the performances with a better preprocessing. We could for instance standardize the data and then map it to 0 - 255 to limit the impact of darker colors, or just apply a brightening/contrast filter.

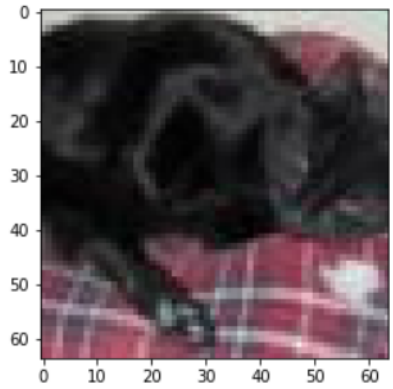
Also, since a lot of dog pictures and taken outside (and very few with cats), seeing grass can be interpreted as seeing a dog by our model.

Missclassified:

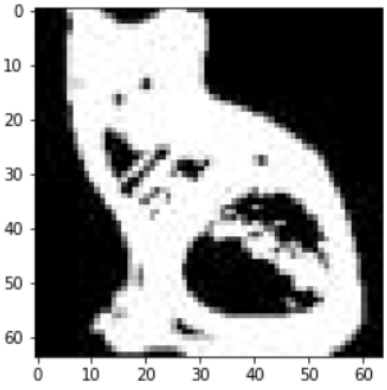
ID: 5959
Score 0.087
Real class: 1



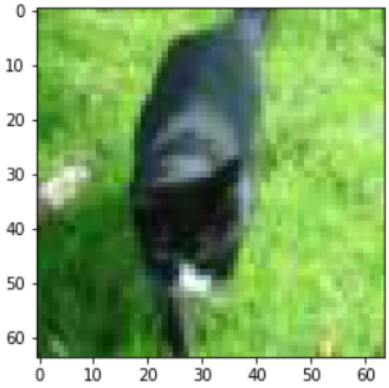
ID: 6876
Score 0.718
Real class: 0



ID: 4833
Score 0.766
Real class: 0



ID: 3605
Score 0.923
Real class: 0



Hard to decide:

ID: 225
Score 0.567
Real class: 1