TP3 Deep Learning with PyTorch: CIFAR10 object classification

Link to the associated github : link

I. Intro

As part of the class ROB313 at ENSTA, about Computer Vision, we followed an introduction course to Pytorch and deep convolutional neural networks. The project aims at implementing a deep neural network model for predicting the class of an object in a 32x32 pixels image, among 10 different classes.

The set of images, on which we will train the models and infer classes is the dataset called CIFAR-10. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 5000 test images.

We worked on the two following aspect of the project :

- Reaching a good accuracy by using a personal, simple and quite naïve model. This has brought us into considering different aspects of a CNN architecture, such as regularization and data augmentation.
- We also aimed at implementing and explaining some recent architectures which achieved top score in benchmarks.

We will explain the different concepts in deep learning that were used, how they were implemented and optimized, then we will describe the results that we obtained.

We will focus on two architectures: one we built by ourselves, and one state of the art architecture: Shake Shake.

- github to our collab architecture: https://github.com/LMaxence/Cifar10_Classification
- github to the shake shake architecture: https://github.com/FlavienGelineau/shake-shake_pytorch

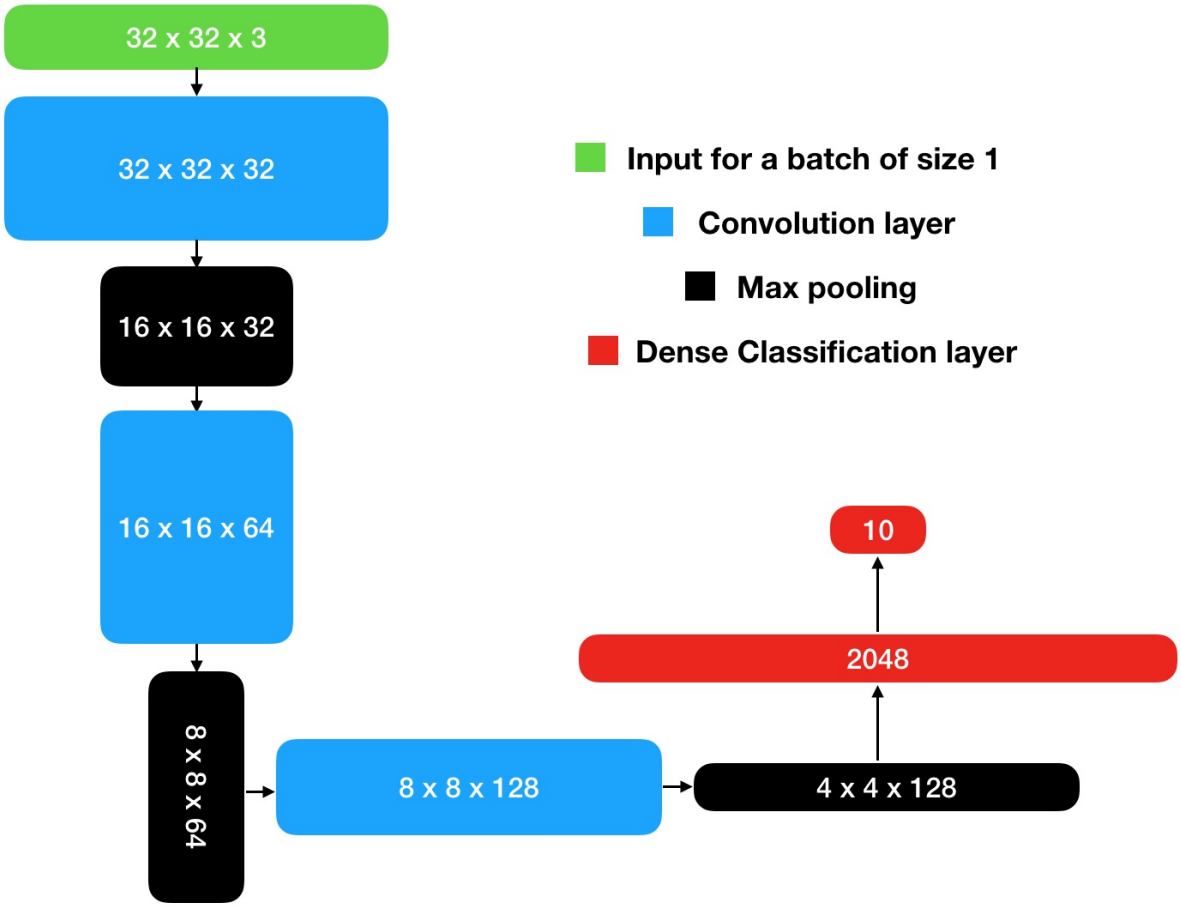
II. Initial basic architecture

Our network

Our architecture is based on the stack of 3 convolution layers, followed by a dense classification layer.

The initial structure of our custom network is as follows:

The goal of the multiple convolution layers is to create features and reduce the dimensionality of the inputs (which is $batch_size\times32\times32\times3$ initially). The process can be presented as follows :



Convolution layer

The idea behind convolution layer in neural networks is to apply kernels to the inputs (which is a 32×32 pixels image in this case), rather than applying a linear function as it is in basic neural network layers. The main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision. For example, given many pictures of cats and dogs it learns distinctive features for each class by itself.

How a convolution layer is applied to the $\it{32\times 32 \times 3}$ image ?

The input image is a actually a volume, meaning here that it has 3 features for each pixels, for the red, green and blue channels. Let's say that we use a convolution layer bringing the number of channels to 32, for instance one that is defined as follows in Pytorch :

```
self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
```

We apply 32 convolutions of size 3 to the image. Each convolution is applied on every channels of the input (which form $32\times 32 \times 1$ inputs), resulting in 3 $32\times 32 \times 1$ outputs for each convolution. For each pixels, a linear combination of the 3 outputs of that pixel form the final output of the kernel for that layer, this operation applied to every pixel resulting in the $32\times 32 \times 1$ feature map of this specific kernel. This operation applied to every kernel builds the final $32\times 32 \times 32$ volume output of this layer.

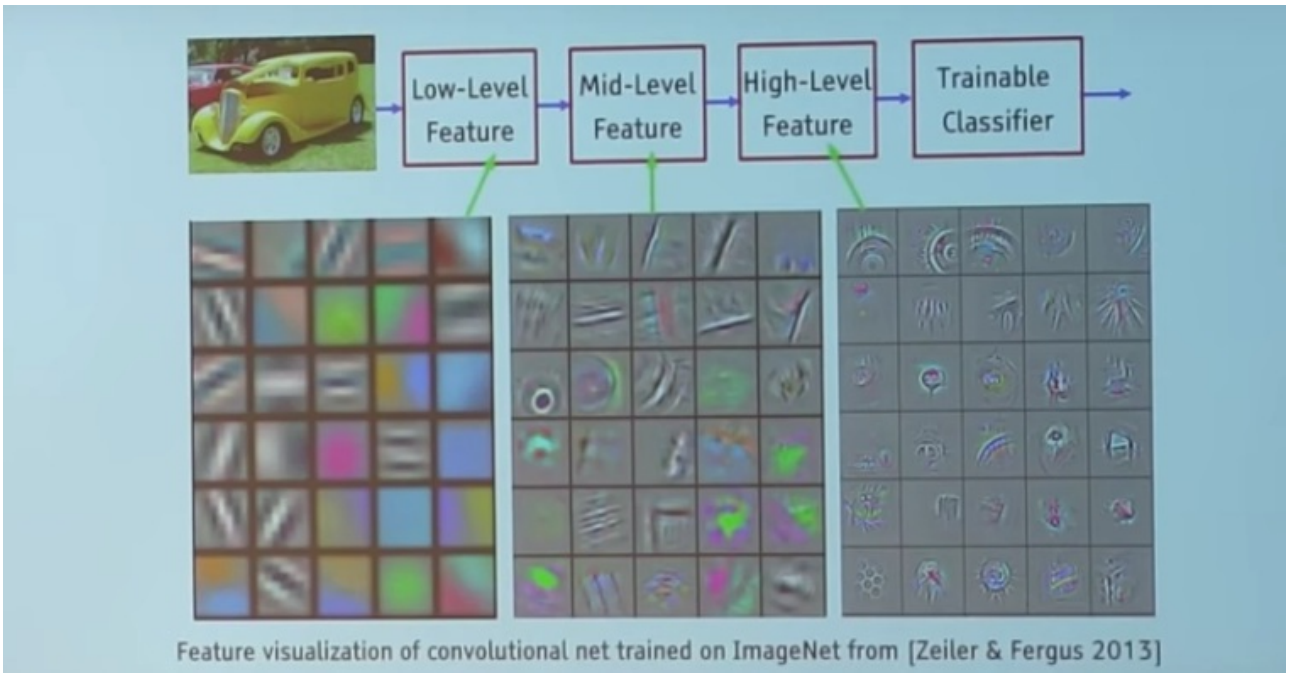
Max-pooling layer

A Max-Pooling layer defines a window (which size is an argument) that goes through the feature maps and selects the highest feature on each window. The result of this operation is a reduction of the dimensionality of the feature map. For instance the following layer will bring the volume from our first convolution layer to a feature map of size $16\times 16 \times 32$.

```
self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

Other pooling layers can be experimented, such as Average-Pooling but we only made usage of Max-Pooling in this work.

The succession of convolution layers and max pooling layers leads to a final 2048×1 tensor which is the final features that will be interpreted by the classification layers. Each one of these outputs may describe an specific aspect of the image, like a corner or a circular-shaped item, as it can be seen in the next picture :



Complete architecture

```
MyConvolutionalNetwork_basic(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=2048, out_features=10, bias=True)
)
Parameters : 307498
```

Firstly, we can observe that we duplicated the convolution layers. This is meant to increase the level of abstraction of the network. Finally, there are 300k trainable parameters in this model.

Optimizer and Criterion loss

One of the key features of our algorithm, which is supposed to train with the experiment, is it's evaluation function and it's optimizer.

Here we use a softmax cross-entropy loss function because it has proven to have good results on multiclass classification problems. As optimizer, we use Adam, which the state of the art for convolutional neural networks.

The optimizer has an important parameter, the learning rate. This element will be discussed later, but we chose to use a constant learning rate of 0.001 in the first place.

Classification Layer

The classification layer used is nothing more than a Dense layer, we crush the final output from the conv layers (of size $4 \times 4 \times 128$) into one big vector of length 2048. The dense layer, combined with the softmax non-linearity produces a final output vector of size 10. This vector represents the **probability that an image belongs to each one of the 10 classes**

Activation function

Activation function is a wrapper of the weight in the neurons. Activation functions are used between each convolution layer and before the final output layer.
Standard activation function is relu for middle neurons and softmax for the last one, because we are in a multiclass classification problem.

First non-optimized results

Training hyperparameters
The training phase is realized with a training set of size **40000** images and **10000** test images on each epoch. There are **20 epochs**. We also use **32 images** per mini-batch.

Training and Validation loss

We can observe here that the training loss is decreasing until the end of the training, but the validation loss start increasing again very quickly. The phenomenon obesrved here is the **overfitting**. Our algorithm performs on it's training data but is really bad at inferring the class of new entries. The algorithm finally inferred rules that might be significantly observed on the training set, but that are not verified in reality.

Our first improvements will focus on reducing the overfitting.

Regularization methods

There are multiple regularization methods. We firstly added new regularization layers, then we used data augmentation in order to use diversified training data.

Batch Normalization

Batch normalization was proposed by Sergey Ioffe and Christian Szegedy in the article *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* : https://arxiv.org/abs/1502.03167. The idea is to perform normalization inside the architecture, after the layers for example. This is a strong help against overfitting and allowed to push further the state of the art accuracy.

To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. There are a lot of papers and article dealing with how batch normalization improves the stability of the neural network, but in a nutshell :

- It makes each layer a bit more independant from one another, and thus the model is less sensitive to noisy images, and variations in the distribution of the inputs.
- After the normalization, there are no output of the layer that has gone really high or really low, and therefore the learning rate can be increased without fearing a gradient divergence.

Dropout

Dropout was proposed by Hinton in *Dropout: A Simple Way to Prevent Neural Networks fromOverfitting* : http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com&utm_campaign=buffe
The idea is to randomly set the activation function of a certain percentage of a layer to the null function during the training phase. By doing this, training is more difficult because the layer can't "collaborate" as well with the next one. It learns then to generalize more.

The second method that we used is the Dropout. It consists in not updating each parameters of a layer when the optimizer moves one step forward. With a probability p, a neuron will remain inactive for the training minibatch. We offer more details in the section regarding state-of-the-art techniques.

Optimization strategy

At each epoch, it will get the validation loss on the validation set. It will save only best model weight which was obtained, so prediction is not sensible to the possible overfitting.
To speed the training, if the val loss has not improved during n epochs, we stop early the training, we consider that overfitting point have been reached.

New architecture

```
MyConvolutionalNetwork(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (drop1): Dropout(p=0.2)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (drop2): Dropout(p=0.3)
  (conv5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (drop3): Dropout(p=0.4)
  (fc1): Linear(in_features=2048, out_features=10, bias=True)
)
Parameters : 308394
```

New results after adding regularization layers


```
Accuracy of the network on the 40000 train images: 81.37 %
Accuracy of the network on the 10000 validation images: 80.69 %
Accuracy of the network on the 10000 test images: 73.52 %
Class      Accuracy (%)
plane        74.00
car          86.80
bird         67.70
cat          59.30
deer         75.20
dog          66.10
frog         73.10
horse        78.80
ship         81.00
truck        84.90
```

This model shows slightly better results, however we would like to have the validation loss to decrease during more than 3 or 5 epochs, in order to reach a better final validation loss.

Data Augmentation

Concept

Data augmentation is the idea that some classes are invariant by some transformations.

For example, let's imagine a cat image. If the image is rotated, the image will still reasonably be considered as a cat. Thus, we have created a second image, different from the first one, which is a cat.

With this, we can increase the size of our data set, and train the future model with much more images. It will then be more robust to the potential noise it will encounter on the test set.

Caution: if we do data augmentation too much, we might have a potential overfitting on the train set. (give links / examples).

Classic data augmentation techniques are

- random cropping of the image : torchvision.transforms.RandomCrop(32, padding=4),
- horizontal flipping : transforms.RandomHorizontalFlip()
- rotation of the image: transforms.RandomRotation(5)

State of the art data-augmentation methods will be described in the following section.

In the first place, we use simple transformation to create our augmented dataset :

```
transform_augmentation = transforms.Compose(
    [transforms.RandomCrop(32, padding=4),
     transforms.RandomHorizontalFlip(),
     transforms.RandomRotation(15),
     transforms.ToTensor(),
     transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
    ])
```

State of the art in Data augmentation

Data augmentation is a very powerfull tool widely used in deep learning on vision problem. Two state of the art techniques in data augmentation were proposed, and deeply improved state of the art accuracy in deep learning vision : Autoaugment and Cutout.

Autoaugment was proposed in the *AutoAugment: Learning Augmentation Policies from Data* : https://arxiv.org/pdf/1805.09501.pdf.

Here they considered an ensemble of data augmentation techniques policies. On it, for a given dataset, they consider each policy as composed by many subpolicies, where each subpolicy consists in two operations(translation, rotation, or shearing, and their associated probabilities of being applied). A search algorithm to find the best policy such that the neural network yields the highest validation accuracy on a target dataset is proposed.

Computing this data augmentation technique would cost a very long time, but fortunately the following technique proposed was open-sourced, and is named CIFAR10Policy. We have taken an implementation to boost our data augmentation techniques on the following github: https://github.com/DeepVoltaire/AutoAugment

Cutout

Cutout is a simple yet very powerfull data augmentation technique, and was proposed in the article *Improved Regularization of Convolutional Neural Networks with Cutout* : https://arxiv.org/abs/1708.04552.

The idea is to randomly masking out square regions of input during training. It is a crucial point for convolutional nets because they spot patterns, and ocluding spatial patterns force neural networks to spot multiple patterns who could have first not be used. This can be used to improve the robustness and overall performance of convolutional neural networks. We stacked this method in our data augmentation methods to be improve the robustess of our model. An implementation can be found here:
https://github.com/uoguelph-mlrg/Cutout

Results after data augmentation

We decided to firstly launch a new training phase on the network trained after the introduction of regularization methods, but using our augmented dataset this time.

There are still 20 epochs and a learning rate of 0.001, these parameters haven't changed yet.

The validation error is higher than before the data augmentation, but this is because the augmented dataset is much more harder to train on than the original one. However we can see that the plateau from before is not anymore, and that the validation loss is still decreasing after 20 epochs.

```
Accuracy of the network on the 40000 train images: 74.92 %
Accuracy of the network on the 10000 validation images: 80.63 %
Accuracy of the network on the 10000 test images: 72.92 %
Class      Accuracy (%)
plane        71.40
car          84.20
bird         65.70
cat          56.40
deer         68.90
dog          63.80
frog         70.80
horse        81.60
ship         81.40
truck        82.60
```

We improved the accuracy.

We decide then to use more epochs for the training phase, and to do so, we have to configure a learning rate scheduler. This will aim at preventing overfitting as the number of trained epochs increases.

```
lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=
[75,100], gamma=0.5)
```

On epoch 75 and 100, the learning rate will be multiplied by 0.5. Results for this final train are presented in section **4) Conclusion**.

III) Conclusion and selected model's parameters

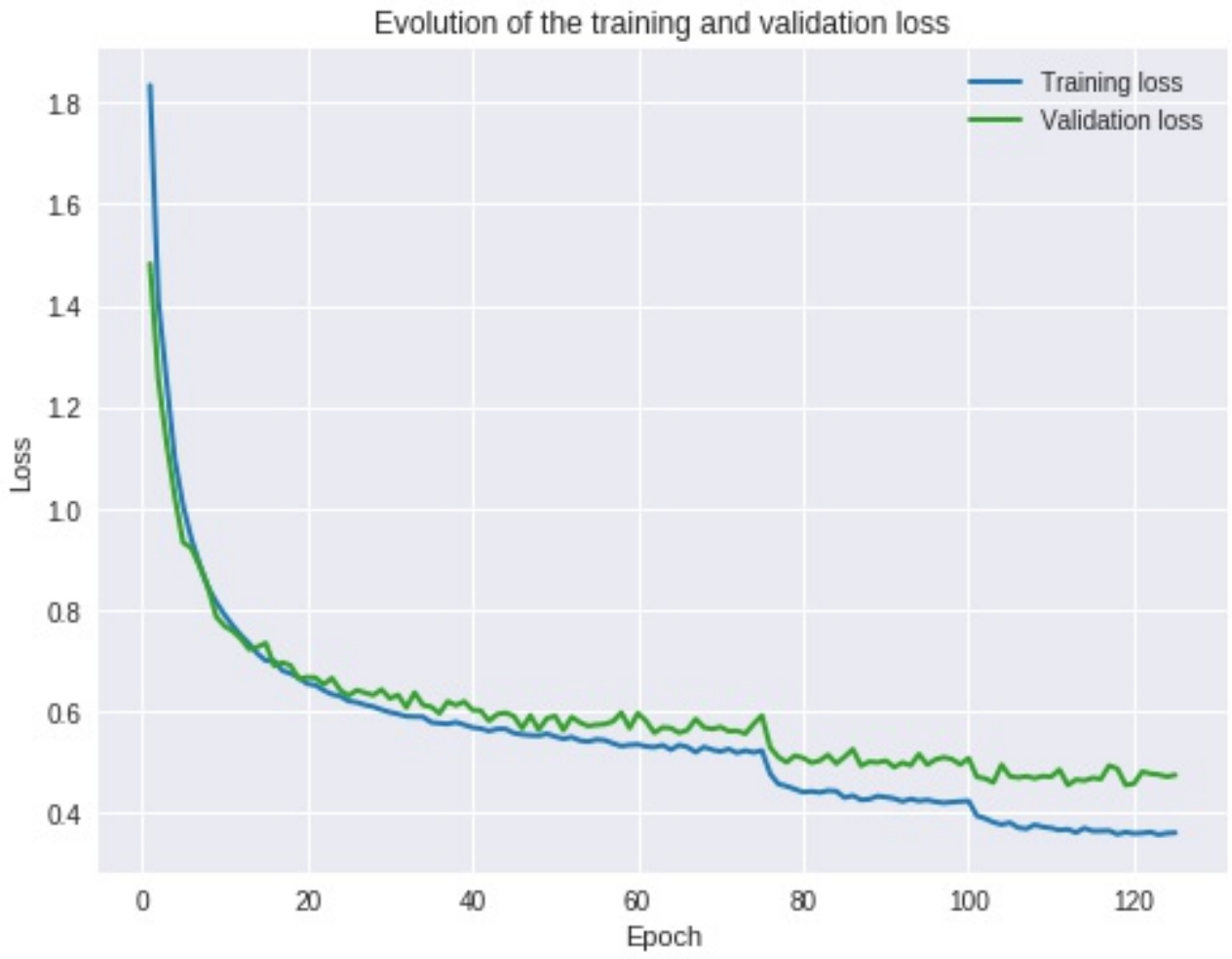
Personnal model

The final training phase has been realized with all the former improvements (batch normalization, dropout, data augmentation, full dataset, use of a learning scheduler).

Training parameters are the following

- **Minibatch** : 32 images
- **Epochs** : 125
- **Learning rate** : 0.001 initially, 0.0005 after 75 epochs, 0.0001 after 100 epochs

Results



We managed to continuously decrease the validation loss over the 125 epochs. The minimal validation loss is 0.45. A very interesting observation is the drop in the loss after iteration 75 and 100, which are the epochs where we manually fixed a smaller learning rate.

```
Accuracy of the network on the 40000 train images: 80.46 %
Accuracy of the network on the 10000 validation images: 85.09 %
Accuracy of the network on the 10000 test images: 77.05 %
Class      Accuracy (%)
plane        77.20
car          91.90
bird         67.80
cat          58.40
deer         70.20
dog          76.40
frog         74.30
horse        86.30
ship         85.80
truck        86.40
```

Shake Shake

https://arxiv.org/abs/1705.07485

Shake Shake is one of the best architectures presented on CIFAR10.
Shake Shake is based on Resnet blocks, which, stacked, make Resnet branches.
The idea is to replace, in the multi-branch network, the standard summation of parallel branches with a stochastic affine combination.
It is a good technique to work against overfitting and was first implemented in "Shake-Shake regularization".

In parallel we implemented a shake shake model with all state of the art methods presented before. Previous implementation was too big for the capacities of google collab so we had to tune it so that it wasn't too long to train.

Training parameters are the following

- **Minibatch** : 32 images
- **Epochs** : 125
- **Learning rate** : 0.01 initially, 0.005 after 10 epochs, 0.025 after 20 epochs 0.001 after 40 epochs.

We stopped too early because of google collabs limitations. With our pre stop, we obtained
Accuracy of the network on the 45000 train images: 79.52 %
Accuracy of the network on the 5000 validation images: 89.42 %
Accuracy of the network on the 5000 test images: 77.32 %

We recoded the model on our personnal machine, based of the following implementation: https://github.com/owruby/shake-shake_pytorch. We have a 1070 Nvidia and the code is parallelized on it so the computation is much faster.

We had the same structure of shake shake, with a netowrk depth of 26 and a network width of 64.

With the following hyperparameters

- **Minibatch** : 64 images
- **Epochs** : 1800
- **Learning rate** : 0.001

We obtained 96.03 accuracy on test. The code and the detailed results of the experiments are on the github link dedicated to the shake shake implementation.