Deep Learning Practical Work 2-c and 2-d
Convolutional Neural Networks

Nicolas Thome Guillaume Couairon Mustafa Sukhor Asya Grechka
Matthieu Cord

Homework

- For all practical works, contact the supervisor of your group : nicolas.thome@sorbonne-universite.fr, mustafa.shukor@isir.upmc.fr and guillaume.couairon@gmail.com
- Email object must be [RDFIA] [Homework-2].
- After doing 2-d, you will need to finish by yourself at home the whole homework (2-a, 2-b, 2-c and 2-d). You will send by email this homework. You are encouraged to do it by group of two (please write both names in the email !)
  - You need to provide answers to all questions in one single PDF file. You can include inside the PDF code snippets (not a whole file !), graphs, or images that you deemed necessary.
  - Don't write long paragraphs of bullshit, we'll just get tired of it and be more severe.
  - Questions with a "Bonus" mark are optional. Questions with a ★ are worth double points.

Data, code, and PDF version of this file are available at
`https://rdfia.github.io`

Goals

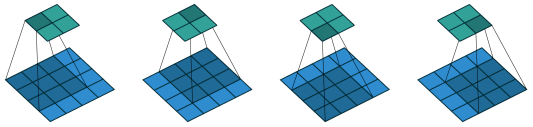
The objective of this lab is to become familiar with convolutional neural networks, that are very suitable to deal with images. To do so, we will study the classical layers of this type of network and we will set up a first network trained on the standard CIFAR-10 dataset. Once the first network is tested, we will discuss different techniques to improve the network learning process : normalization, data augmentation, variants of SGD, dropout and batch normalization.

Partie 1 – Introduction to convolutional networks (1 hour)

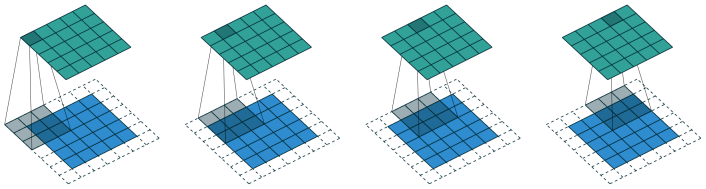
Convolutional Neural Networks (CNNs) have become the state-of-the-art architectures in almost all *machine learning* tasks applied to images. These networks differ from more traditional networks by two basic layers used in their architecture : convolution and pooling layers.

1.1 Convolutional layers

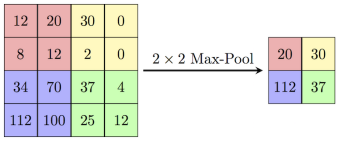
Input These layers take as input a set of D feature maps, each feature map being a matrix of size $n_x \times n_y$. We therefore have an 3-d tensor in entry of size $n_x \times n_y \times D$. More generally, a tensor is the n-dimensional equivalent of a matrix.



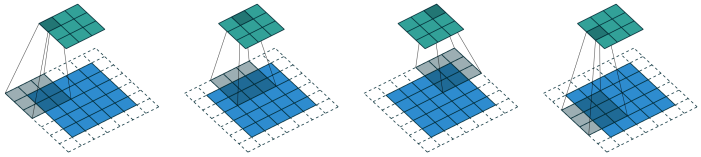
(a) Simple convolution, no padding, stride of 1, kernel of size 3.



(b) Convolution with padding of 1, stride of 1, kernel of size 3.



(c) Example of max pooling, no padding, stride of 2, kernel of size 2.



(d) Convolution with padding of 1, stride of 2, kernel of size 3.

FIGURE 1 – Illustrations of convolutions and pooling. For convolutions, input feature maps at the bottom (only 1 here, normally several), output feature map at the top.

Output On this input, we apply C convolutions, each with a convolution filter of size $w \times h \times D$, we generally have $w = h$ that we call *kernel size* k. These C filters constitute the parameters we need to learn. Each convolution with a filter produces an output feature map, of size $n'_x \times n'_y \times C$, where n'_x and n'_y depend on the hyperparameter of the convolution.

Hyperparameters In addition to the number of filters and the kernel size, there are two common hyperparameters for convolution layers, the **padding** and the **stride**. The padding indicates how many rows of 0 to add around the entry. The stride indicates how many steps we move between two convolution calculations. The " classic " convolution has a smaller output than the input because of the size of the filter that must be placed inside the input feature map, see figure 1a. Adding padding allows for example to recover the initial size, see figure 1b. Adding a stride allows to skip some values, this corresponds to sub-sampling the output, see figure 1d.

References A demonstration of convolution can be viewed at `http://cs231n.github.io/assets/conv-demo/index.html`, and the article by Dumoulin & Visin (2016) from which the figure 1 is taken is very interesting for understanding convolutions and some variants not presented here.

1.2 Pooling layers

Principle Pooling is a function of spatial downsampling. It also takes feature maps of size $n_x \times n_y \times D$ as input, and reduces the first two spatial dimensions. The calculation is carried out according to the same principle as a convolution, but applies to each feature map independently. We therefore go through each feature map with a sliding window, but instead of producing a convolution product between the window and a filter, we perform a pooling operation on the input window to produce a value. We therefore obtain an output of size $n'_x \times n'_y \times D$ with n'_x and n'_y significantly smaller than n_x and n_y (often by a factor of 2).

Hyperparameters A pooling layer has a **kernel size** (defining the window size), a **stride** and **padding** which behave as for convolution, and a **pooling operation**, the most common being *max pooling* (we keep the maximum value in the window) and *average pooling* (we take the average value of the window). A very common pooling is a max pooling with kernel of size 2, stride of size 2 and without padding, as shown in figure 1c. The pooling layer does not have parameters to learn.

1.3 Common convolutional architectures

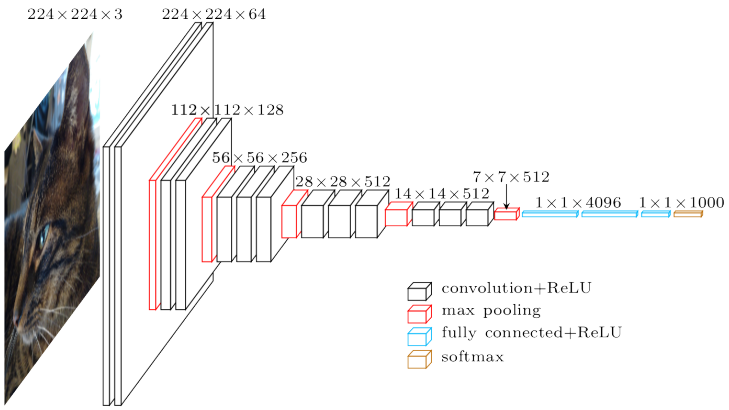


FIGURE 2 – VGG16 network.

Classical convolutional neural networks are generally composed of a succession of layers of convolutions (with ReLU) with more and more filters, and whose spatial dimension is progressively reduced by layers of max pooling possibly until total aggregation of the spatial dimensions. At the end of the network, there is therefore only the "depth" corresponding to the number of filters applied by the last convolution ($1 \times 1 \times C$). Finally, we generally add one or a few linear layers (called *fully-connected*). An example of this type of architecture is the VGG16 network Simonyan & Zisserman (2014) as shown in Figure 2.

Since then, more complex architectures have been developed, in particular Inception or ResNet architectures, but the idea remains the same.

1.4 Questions

1. Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size $x \times y \times z$ what will be the output size ?
   How much weight is there to learn ?
   How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size ?
2. ★ What are the advantages of convolution over fully-connected layers ? What is its main limit ?
3. ★ Why do we use spatial pooling ?
4. ★ Suppose we try to compute the output of a classical convolutional network (for example the one in Figure 2) for an input image larger than the initially planned size (224 × 224 in the example). Can we (without modifying the image) use all or part of the layers of the network on this image ?
5. Show that we can analyze fully-connected layers as particular convolutions.
6. Suppose that we therefore replace fully-connected by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest ?
7. We call the *receptive field* of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers ? Can you imagine what happens to the deeper layers ? How to interpret it ?

Partie 2 – Training *from scratch* of the model (1 hour)

We will now implement our first convolutional network that we will apply to the CIFAR-10 database (Krizhevsky, 2009, *i.e.*, Figure 3). This dataset of 32 × 32 pixels RGB images has 10 classes, 50k images in *train* and 10k images in *test*.

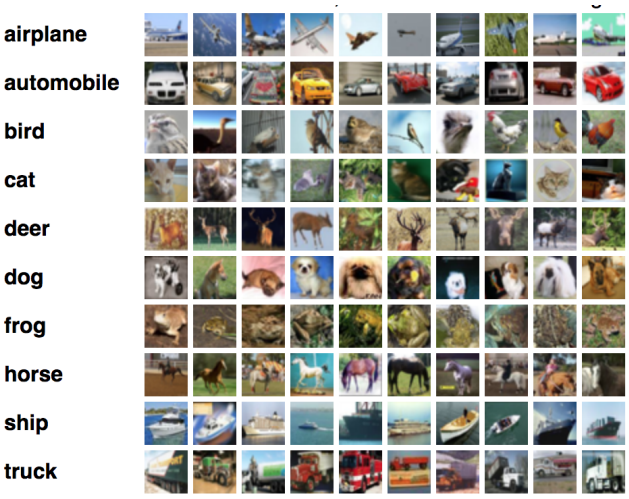


FIGURE 3 – CIFAR-10 database.

2.1 Network architecture

The network that we are going to implement has a style close to the AlexNet architecture of Krizhevsky et al. (2012) adapted to the CIFAR-10 base whose images are smaller. It will be composed of the following layers :

- `conv1` : 32 convolutions 5×5, followed by ReLU
- `pool1` : max-pooling 2×2
- `conv2` : 64 convolutions 5×5, followed by ReLU
- `pool2` : max-pooling 2×2
- `conv3` : 64 convolutions 5×5, followed by ReLU
- `pool3` : max-pooling 2×2
- `fc4` : fully-connected, 1000 output neurons, followed by ReLU
- `fc5` : fully-connected, 10 neurons output, followed by softmax

Questions

8. For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed ?
9. For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed ?
10. ★ For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.
11. What is the total number of weights to learn ? Compare that to the number of examples.
12. Compare the number of parameters to learn with that of the BoW and SVM approach.

2.2 Network learning

To learn the network, start from the `base.py` file provided. This file learns a small historical convolutional network, called LeNet5, based on the MNIST database. Start by reading and experimenting with this code before modifying it to do what you want.

Questions

13. Read and test the code provided. You can start the training with this command : `main (batch_size, lr, epochs, cuda = True)`
14. ★ In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the the difference in data) ?
15. Modify the code to use the CIFAR-10 dataset and implement the architecture requested above. (the class is `datasets.CIFAR10`). Be careful to make enough epochs so that the model has finished converging.
16. ★ What are the effects of the *learning rate* and of the *batch-size* ?
17. What is the error at the start of the first epoch, in train and test ? How can you interpret this ?
18. ★ Interpret the results. What's wrong ? What is this phenomenon ?

Partie 3 – Results improvements (2 hours)

★ For this part, in the end of session report, indicate the methods you have successfully implemented and for each one explain in one sentence its principle and why it improves learning.

We will now see several classic techniques to improve the performance of our model.

3.1 Standardization of examples

A common technique in *machine learning* is to standardize the examples in order to better condition the learning. When learning CNN, the most common technique is to calculate the mean value and the standard deviation of each RGB channel over the whole train. We therefore obtain 6 values. Each image is then normalized by subtracting from each pixel the mean value corresponding to its channel and dividing it by the corresponding standard deviation.

For CIFAR-10, the values are $\mu = [0.491, 0.482, 0.447]$ and $\sigma = [0.202, 0.199, 0.201]$.

Implémentation Add normalization in the data pre-processing.
This should be done when calling `datasets.CIFAR10` :

```
train_dataset = datasets.CIFAR10(path, train=True, download=True,
transform=transforms.Compose([
        # Set of transformations to apply, we have as input a PIL image (Python
        ↪ Image Library). Refer to the PyTorch documentation in the
        ↪ torchvision.transforms package
        transforms.ToTensor () # Transform the PIL image to a torch.Tensor
        # Add normalization with values defined above
]))
```

Questions

19. Describe your experimental results.
20. Why only calculate the average image on the training examples and normalize the validation examples with the same image ?
21. **Bonus :** There are other normalization schemes that can be more efficient like ZCA normalization. Try other methods, explain the differences and compare them to the one requested.

3.2 Increase in the number of training examples by *data increase*

Convolution networks often contain several million parameters and are learned on very large image databases. CIFAR-10 is a relatively small set of images relative to the number of model parameters. To counter this problem, we use *data augmentation* methods which seek to artificially increase the number of available examples, which can be crucial when the training set is small.

The principle consists in generating at each *epoch* a "variant" of each image, by applying random transformations to it. Here, we propose to test two most common transformations : a random crop of size 28×28 pixels (among the 32×32) and a random horizontal symmetry of the image (1 in 2 chance of applying it).

Note that these transformations are applied to the train images, but sometimes the evaluation phase must be adjusted to take this into account. For example here, we reduced the size of our learning images. One could think of several strategies to counter this, in particular : modifying the model to take smaller images and cropping centered on the test images or adding padding around the train images to return to a size of 32×32.

To keep the same number of parameters in the model (so that our results are more easily comparable), we propose to modify the `pool3` layer by adding the option `ceil_mode = True` so that the output spatial size is always 4×4.

Implementation

— Modify the call to `datasets.CIFAR10` to add in *train* a random crop size 28 and a random horizontal symmetry ; and in *test* a centered crop size 28.

— Modify the `pool3` layer to add the option `ceil_mode = True` .

Questions

22. Describe your experimental results and compare them to previous results.
23. Does this horizontal symmetry approach seems usable on all types of images ? In what cases can it be or not be ?
24. What limits do you see in this type of data increase by transformation of the dataset ?
25. **Bonus :** Other data augmentation methods are possible. Find out which ones and test some.

3.3 Variants on the optimization algorithm

Until now, a simple optimization algorithm, the SGD, has been used. There are many variations of this algorithm allowing faster or better convergence.

A first strategy consists in applying a *momentum* on the gradient, that is to say in applying a sliding average on the gradients in order not to completely forget the direction of the gradient provided by the preceding batch.

Another strategy consists in modifying the *learning rate* during the training (generally to decrease it), for example, one can apply an exponential decrease to η according to the formula $\eta_t = \eta \times m^t$ where t is the *epoch* and m the momentum.

```
# Import the package
import torch.optim.lr_scheduler
# [...]
# Create the scheduler
optimizer = ...
```

```
lr_sched = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
# [...]
# Modify the learning rate after each epoch
epoch(...)
lr_sched.step()
```

Implementation

— Modify the optimizer `SGD` to add momentum to it (see the documentation) using the standard value of 0.9.

— Add a *learning rate scheduler* and use an expoential decay with a coefficient of 0.95.

Questions

26. Describe your experimental results and compare them to previous results, including learning stability.
27. Why do these two methods improve learning ?
28. **Bonus :** Many other variants of SGD exist and many *learning rate* planning strategies exist. Which ones ? Test some of them.

3.4 Regularization of the network by *dropout*

We saw in the question 10 that the `fc4` layer of the network contains a very large number of weights, the risk of overfitting in this layer is therefore important. To remedy this problem, we are going to add a layer allowing a very efficient regularization for neural networks : the *dropout* layer. The general principle of this layer is to randomly deactivate at each pass a part of the neurons of the network to artificially reduce the number of parameters. All the necessary information about this layer can be found in the corresponding article of Srivastava et al. (2014).

Implementation Add a dropout layer between `fc4` and `fc5`.

Questions

29. Describe your experimental results and compare them to previous results.
30. What is regularization in general ?
31. Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it ?
32. What is the influence of the hyperparameter of this layer ?
33. What is the difference in behavior of the *dropout* layer between training and test ?

3.5 Use of *batch normalization*

A final learning strategy that we are going to test is the *batch normalization* of Ioffe & Szegedy (2015). This is a layer that learns to renormalize the outputs of the previous layer, allowing to stabilize the learning.

Implementation Add a 2D *batch normalization* layer immediately after each convolution layer.

Question

34. Describe your experimental results and compare them to previous results.

Références

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv*, 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 2012.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.