RDFIA: Homework 2-a,b,c,d

GALMICHE Nathan PIRIOU Victor

November 17, 2022

Abstract

Since the ImageNet 2012 challenge, deep learning is the go-to technology for numerous and various prediction or estimation problems. When used right, deep learning-based methods are often more effective than the traditional ones that have been developed for decades to reach their efficiency (*e.g.* the SVM+BoW approach studied in the previous practical works).

However useful or pleasant to use they can be, understanding how these technologies work is quite challenging.

Deep Learning is a subset of Machine Learning and is based on artificial neural networks. The goal of such networks is to learn some parameters that best approximate a mapping between an input data and a desired output. These networks are modelled as acyclic weighted oriented graphs composed of simplified models of neurons aggregated into successive *layers* that are connected by learnable weights. Each layer allows the learnt function to be potentially more non-linear and then more complex. The architecture of the network depends on the data we want to process and the results we are looking for. For example, in this practical work we learned that convolutional neural networks are very well-suited to classify images.

Interestingly, the *Universal Approximation Theorem* states that any continuous function can be modelled by a sufficiently large single-layer perceptron. There are, however, a few difficulties with the use of an extremely wide and shallow network. The main issue is that these very wide and shallow networks are excellent at memorization, but not so good at generalization. Unlike deep neural networks that are good at generalization and able to capture the hierarchical nature of the world.

For the function to be learned, the network needs to be trained to understand the data. During the training, the possible error an Artificial Neural Network makes, when predicting the expected outcome, is measured by a *loss function* that needs to be differentiable. This function allows the problem of learning the weights of these networks to be cast as an optimization problem that seeks to be solved by iterative gradient descent methods.

It's important to mention that despite myriad of efforts, the ability of the learning to converge to a good local minima, of the possibly non-convex loss function, has never been proven.

1 Theoretical foundation

1. ★ **What are the train, val and test sets used for?**
   The train set is the part of the dataset we use to compute the optimal weights of the neural network.

   The evaluation set is used to evaluate different configurations of hyperparameters (number of layers, number of neurons, etc). It also allows to watch the model's efficiency on less biased data.

   The test set is especially dedicated to obtain an unbiased evaluation of the final model, that is done on unseen data. It allows to evaluate our final model.

2. **What is the influence of the number of examples N ?**
   The bigger N is, the better the capacity of the model to generalize will be. This means that with few examples, the model will be trained to stay close to these and won't be effective on new examples. If N is very big, the model will be trained to recognize all of them and it is more likely that new examples will be close to the ones already used for training.

3. **Why is it important to add activation functions between linear transformations?**
   An artificial neural network composed of L layers that can been seen as a combination of non-linear function (applied element-wise) compositions and matrix multiplications:

   $$f(x) := g^L(W^L g^{L-1}(W^{L-1}...g^1(W^1 x)...)),$$

   where $W^L = \{w^l_{i,j}\}$ is the weight between the j-th node in layer $l-1$ and the i-th node in layer l.

   Without such non-linear functions, f would be a succession of linear transformations, *i.e.* a linear transformation.

4. ★ **What are the sizes n_x, n_h, n_y in the figure 1?**
   In figure 1:

   - n_x is the number of neurons of the first layer which is 2.
   - n_h is the number of neurons of the hidden layer which is 4.
   - n_y is the number of neurons of the output layer which is 2. Generally, the way we have to interpret this number isn't always the same. In our case, it represents the number of class since we use the SoftMax function.

   In practice, how are these sizes chosen? In practice, n_x and n_y are respectively equal to the input data size and target size. In order to determine the hyper-parameter n_h, we need to test different values and see which one works the best. The optimal value depends on the data and the problem we want to solve.

5. **What do the vectors $\hat{y}$ and y represent ? What is the difference between these two quantities?**
   y is a one-hot encoding of the groundtruth whereas $\hat{y}$ is the output vector of the network and it represents the probability distribution of the class. The goal is to make $\hat{y}$ as close as y as possible.

6. **Why use a SoftMax function as the output activation function?**
   We use it to make the sum of our outputs equal to one. This is a condition that needs to be satisfied for the output to be a probability distribution.

7. **Write the mathematical equations allowing to perform the *forward* pass of the neural network, i.e. allowing to successively produce $\tilde{h}, h, \tilde{y}$ and $\hat{y}$ starting at x.**

   $$\begin{aligned}
   \tilde{h} &= W_h x + b_h \\
   h &= \tanh(\tilde{h}) \\
   \tilde{y} &= W_y h + b_y \\
   \hat{y} &= \text{SoftMax}(\tilde{y})
   \end{aligned}$$

There are many ways to do the forward. Here, we chose one that is different than that of *PyTorch*. That's why in questions 16/17 we don't obtain exactly the same results as the ones written in the backpropagation cheatsheet that uses the same convention as *PyTorch*.

8. **During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function $\mathcal{L}$?**
Mathematically, if $\mathcal{L}$ is the cross entropy then we have:

$$\frac{\partial l(y,\hat{y})}{\partial \hat{y}_i} = \frac{\partial(-y_i \cdot \log(\hat{y}_i) - (1-y_i)\cdot\log(1-\hat{y}_i))}{\partial \hat{y}_i} = -y_i \cdot \frac{\partial \log(\hat{y}_i)}{\partial \hat{y}_i} - (1-y_i)\cdot\frac{\partial \log(1-\hat{y}_i)}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i},$$

and, if it's the mean squared error we have:

$$\frac{\partial l(y,\hat{y})}{\partial \hat{y}_i} = \frac{\partial(\sum_i (y_i-\hat{y}_i)^2)}{\partial \hat{y}_i} = \frac{\partial (y_i-\hat{y}_i)^2}{\partial \hat{y}_i} = -2\cdot(y_i-\hat{y}_i).$$

In both cases, in order to decrease the global loss function $\mathcal{L}$, $\hat{y}_i^{(t)}$ must vary such that it moves toward $\hat{y}_i^{(t+1)}$ defined by:

$$\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} - \gamma \cdot \frac{\partial l(y,\hat{y})}{\partial \hat{y}_i},$$

where γ is the learning rate.

Finally:

- if $\hat{y}_i > y_i$ then $\frac{\partial l(y,\hat{y})}{\partial \hat{y}_i} > 0$ and $y_i < \hat{y}_i^{(t+1)} < \hat{y}_i^{(t)}$;
- else $\hat{y}_i < y_i$ then $\frac{\partial l(y,\hat{y})}{\partial \hat{y}_i} < 0$ and $y_i > \hat{y}_i^{(t+1)} > \hat{y}_i^{(t)}$;

so $\hat{y}_i$ varies towards y_i.

9. **How are these functions better suited to classification or regression tasks?**

Cross entropy loss is better for classification. Minimizing the cross entropy loss is equivalent to maximum likelihood estimation (MLE), under the label independence assumption. In MLE,

$$\begin{aligned}
f^* &= \arg\max_f \prod_{i=1}^{n_x} \Pr\left(\hat{y}_i = y_i \mid f, x_i\right) \\
&= \arg\max_f \sum_{i=1}^{n_x} \log\left[\Pr\left(\hat{y}_i = y_i \mid f, x_i\right)\right] \\
&= \arg\max_f \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} \mathbb{1}_{y_i=y_j} \log\left[\Pr\left(\hat{y}_i = y_i \mid f, x_i\right)\right] \\
&= \arg\max_f \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} \Pr\left(y_i = y_j \mid f, x_i\right) \log\left[\Pr\left(\hat{y}_i = y_i\right)\right] \\
&= \arg\max_f \sum_{i=1}^{n_x} -\mathrm{H}\left(Pr\left(y_i \mid f, x_i\right), \Pr\left(\hat{y}_i = y_i \mid f, x_i\right)\right) \\
&= \arg\min_f \sum_{i=1}^{n_x} \mathrm{H}\left(\Pr\left(y_i \mid f, x_i\right), \Pr\left(\hat{y}_i = y_i \mid f, x_i\right)\right) \text{ where } \mathrm{H}(y_i,\hat{y}_i) = -\sum y_i \log \hat{y}_i \text{ is the binary cross-entropy.}
\end{aligned}$$

Mean Squared Error loss is better for regression. Using this loss amounts to suppose that the samples are independent and normally distributed, which are good assumptions for regression.

$$
\begin{aligned}
f^* &= \arg\max_f \prod_{i=1}^{n_x} \Pr\left(\hat{y}_i = y_i \mid f, x_i\right) \\
&= \arg\max_f \sum_{i=1}^{n_x} \log\left[\Pr\left(\hat{y}_i = y_i \mid f, x_i\right)\right] \\
&= \arg\max_f \sum_{i=1}^{n_x} \log\left[\mathcal{N}\left(y_i \mid f\left(x_i\right), \sigma^2\right)\right] \\
&= \arg\max_f \sum_{i=1}^{n_x} \log\left[\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\left(y_i - f\left(x_i\right)\right)^2}{2\sigma^2}}\right] \\
&= \arg\max_f \sum_{i=1}^{n_x} \left[\log\frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{2\sigma^2}\left(y_i - f\left(x_i\right)\right)^2\right]^2 \\
&= \arg\max_f n_x \log\frac{1}{\sigma\sqrt{2\pi}} - \sum_{i=1}^{n_x} \frac{1}{2\sigma^2}\left(y_i - f\left(x_i\right)\right)^2 \\
&= \arg\max_f -\sum_{i=1}^{n_x}\left(y_i - f\left(x_i\right)\right)^2 \\
&= \arg\min_f \sum_{i=1}^{n_x}\left(y_i - f\left(x_i\right)\right)^2 \text{ where } f\left(x_i\right) = \hat{y}_i.
\end{aligned}
$$

10. **What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions? Which one seems the most reasonable to use in the general case?**

    Classic gradient descent It's the more stable variation but it's the one that needs the most epochs for the convergence of the learning.

    Online stochastic gradient descent The most unstable but it's the one that needs the smallest number of epochs to converge.

    Mini-batch stochastic gradient descent It's a trade-off between the classic and the online stochastic gradient descent. It's the one that is the most reasonable to use in the general case.

11. ★ **What is the influence of the learning rate η on learning?**
    If η is lower than the optimal learning rate, it will take more epochs for the learning to converge. If it's higher, it will take less epochs at the cost of a risk of divergence as the point of convergence can be skipped by a step being too large. In short, η influences the learning speed but it has to be set cautiously.

12. ★ **Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the *loss* with respect to the parameters, using the naive approach and the *backprop* algorithm.**
    The naive approach consists of directly computing the gradient with respect to each weight individually so its complexity is $\mathcal{O}(n!)$ where n is the number of layers.

    The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule so its complexity is $\mathcal{O}(n)$ which is much smaller than that of the naive approach.

13. **What criteria must the network architecture meet to allow such an optimization procedure ?**
    The network must be modelled as acyclic oriented graphs where each of its function are derivable.

14. **The function SoftMax and the *loss* of *cross-entropy* are often used together and their gradient is very simple. Show that the *loss* can be simplified by:**

$$\ell = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right).$$

Let's $\tilde{y}_i$ be the output of the network:

$$\hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_{i=1}^{n_y} \exp(\tilde{y}_i)},$$

hence the cross-entropy loss is defined by:

$$\begin{aligned}
\ell(y, \hat{y}) &= -\sum_i y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}\right) \\
&= -\sum_i y_i \left(\log\left(e^{\tilde{y}_i}\right) - \log\left(\sum_j e^{\tilde{y}_j}\right)\right) \\
&= -\sum_i y_i \left(\tilde{y}_i - \log\left(\sum_j e^{\tilde{y}_j}\right)\right) \\
&= -\sum_i y_i \tilde{y}_i + \sum_i y_i \log\left(\sum_j e^{\tilde{y}_j}\right)
\end{aligned}$$

Because the y_i are probabilities we have $\sum_i y_i = 1$ so we finally obtain:

$$\ell(y, \hat{y}) = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right).$$

15. **Write the gradient of the *loss (cross-entropy)* relative to the intermediate output $\tilde{y}$.**

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{y}_i} &= \frac{\partial\left(-\sum_j y_j \tilde{y}_j + \log\left(\sum_k e^{\tilde{y}_k}\right)\right)}{\partial \tilde{y}_i} \\
&= -\frac{\partial\left(\sum_j y_j \tilde{y}_j\right)}{\partial \tilde{y}_i} + \frac{\partial\left(\sum_k e^{\tilde{y}_k}\right)}{\partial \tilde{y}_i} \cdot \frac{1}{\sum_k e^{y_k}} \\
&= -y_i + \frac{e^{\tilde{y}_i}}{\sum_k e^{\tilde{y}_k}} \\
&= \text{SoftMax}\left(\tilde{y}_i\right) - y_i \\
&= \hat{y}_i - y_i
\end{aligned}$$

hence :

$$\nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \begin{bmatrix} \widehat{y}_1 - y_1 \\ \vdots \\ \widehat{y}_{n_y} - y_{n_y} \end{bmatrix} = \widehat{y} - y.$$

16. **Using the *backpropagation*, write the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} \ell$. Note that writing this gradient uses $\nabla_{\tilde{y}} \ell$. Do the same for $\nabla_{b_y} \ell$.**

Gradient of the loss with respect to the weights of the output layer $\nabla_{W_y}\ell$:

$$\nabla_{W_y}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y1}} & \cdots & \frac{\partial \ell}{W_{y,n_yn_h}} \end{bmatrix} = \begin{bmatrix} \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,i,j}}\Big|_{i,j=1,1} & \cdots & \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,i,j}}\Big|_{i,j=1,n_h} \\ \vdots & \ddots & \vdots \\ \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,i,j}}\Big|_{i,j=n_y,1} & \cdots & \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,i,j}}\Big|_{i,j=n_y,n_h} \end{bmatrix},$$

where :

$$\frac{\partial \tilde{y}_k}{\partial W_{y,i,j}} = \frac{\partial\left(\sum_{\ell=1}^{n_h} W_{y,k,\ell}\cdot h_l + b_{y,k}\right)}{\partial W_{y,i,j}} = \frac{\partial(W_{y,i,j}\cdot h_j + b_{y,k})}{\partial W_{y,i,j}} = h_j.$$

Finally, we get :

$$\nabla_{W_y}\ell = \begin{bmatrix} \frac{\partial l}{\partial \tilde{y}_1}\cdot h_1 & \cdots & \frac{\partial l}{\partial \tilde{y}_1}\cdot h_{n_h} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}}\cdot h_1 & \cdots & \frac{\partial l}{\partial \tilde{y}_{n_y}}\cdot h_{n_h} \end{bmatrix} = \nabla_{\tilde{y}}\ell \cdot h^T.$$

Gradient of the loss with respect to the weights of the output layer $\nabla_{b_y}\ell$: The same way, we get :

$$\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{\partial\left(\sum_{\ell=1}^{n_h} W_{y,k,\ell}\cdot h_l + b_{y,k}\right)}{\partial b_{y,i}} = \frac{\partial\left(\sum_{\ell=1}^{n_h} W_{y,i,j}\cdot h_l + b_{y,k}\right)}{\partial b_{y,i}} = \frac{\partial(b_{y,i})}{\partial b_{y,i}} = 1.$$

It follows that :

$$\frac{\partial \ell}{\partial b_{y,i}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \tilde{y}_i}.$$

Finally :

$$\nabla_{b_y}\ell = \nabla_{\tilde{y}}\ell.$$

17. ★ **Compute other gradients: $\nabla_{\tilde{h}}\ell, \nabla_{W_h}\ell, \nabla_{b_h}\ell$.**
Gradient$\nabla_{\tilde{h}}\ell$

$$\nabla_{\tilde{h}}\boldsymbol{\ell} = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{h}_1} \\ \vdots \\ \frac{\partial l}{\partial \tilde{h}_{n_h}} \end{bmatrix} = \begin{bmatrix} \sum_k \frac{\partial \ell}{\partial h_k}\frac{\partial h_k}{\partial \tilde{h}_i}\Big|_{i=1} \\ \vdots \\ \sum_k \frac{\partial l}{\partial h_k}\frac{\partial h_k}{\partial \tilde{h}_i}\Big|_{i=n_h} \end{bmatrix},$$

where:

- $\frac{\partial h_k}{\partial \tilde{h}_i} = \frac{\partial(\tanh(\tilde{h}_k))}{\partial \tilde{h}_i} = \frac{\partial(\tanh(\tilde{h}_k))}{\partial \tilde{h}_i} = \begin{cases} 1-\tanh^2\left(\tilde{h}_k\right) = 1-\tanh^2\left(\tanh^{-1}(h_k)\right) = 1-h_i^2 & \text{if } k=i \\ 0 & \text{otherwise} \end{cases}$,
- $\frac{\partial \ell}{\partial h_i} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial h_i} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial\left(\sum_{\ell=1}^{n_h} W_{y,k,\ell}\cdot h_\ell + b_{y,k}\right)}{\partial h_i} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial(W_{y,k,i}\cdot h_i + b_{y,k})}{\partial h_i} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}W_{y,k,i}.$

Finally, we get:

$$\nabla_{\tilde{h}}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{h}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{h}_{n_h}} \end{bmatrix} = \begin{bmatrix} \left(1-h_1^2\right)\sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\cdot W_{y,k,1} \\ \vdots \\ \left(1-h_{n_h}{}^2\right)\sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\cdot W_{y,k,n_h} \end{bmatrix} = W_y^T\cdot\nabla_{\tilde{y}}\ell \odot \left(1-h^2\right).$$

Gradient $\nabla_{W_h}\ell$

$$\nabla_{W_h}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{h,1,1}} & \cdots & \frac{\partial \ell}{\partial W_{h,1,n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{h,n_h,1}} & \cdots & \frac{\partial \ell}{W_{h,n_h,n_x}} \end{bmatrix} = \begin{bmatrix} \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial W_{h,i,j}}\Big|_{i,j=1,1} & \cdots & \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial W_{h,i,j}}\Big|_{i,j=1,n_x} \\ \vdots & \ddots & \vdots \\ \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial W_{h,i,j}}\Big|_{i,j=n_h,1} & \cdots & \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial W_{h,i,j}}\Big|_{i,j=n_h,n_x} \end{bmatrix}$$

where $\sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial W_{h,i,j}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial\left(\sum_{\ell=1}^{n_x} W_{h,k,\ell}\cdot x_\ell + b_{h,k}\right)}{\partial W_{h,i,j}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial(W_{h,k,j}\cdot x_j + b_{h,k})}{\partial W_{h,i,j}} = \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial(W_{h,i,j}\cdot x_j + b_{h,i})}{\partial W_{h,i,j}} = \frac{\partial \ell}{\partial \tilde{h}_i}\cdot x_j.$

Finally, we get:

$$\nabla_{W_h}\ell = \nabla_{\tilde{h}}\ell \cdot x^T.$$

Gradient $\nabla_{b_h}\ell$

$$\nabla_{b_h}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial b_{h,1}} \\ \vdots \\ \frac{\partial \ell}{\partial b_{h,n_h}} \end{bmatrix} = \begin{bmatrix} \sum_k \frac{\partial l}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial b_{h,i}}\Big|_{i=1} \\ \vdots \\ \sum_k \frac{\partial l}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial b_{h,i}}\Big|_{i=n_h} \end{bmatrix}$$

where $\sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k}\frac{\partial\left(\sum_{\ell=1}^{n_x} W_{h,k,\ell}\cdot x_\ell + b_{h,k}\right)}{\partial b_{h,i}} = \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial\left(\sum_{\ell=1}^{n_x} W_{h,i,\ell}\cdot x_\ell + b_{h,i}\right)}{\partial b_{h,i}} = \frac{\partial \ell}{\partial \tilde{h}_i}\frac{\partial\left(\sum_{\ell=1}^{n_x} W_{h,i,\ell}\cdot x_\ell + b_{h,i}\right)}{\partial b_{h,i}} = \frac{\partial \ell}{\partial \tilde{h}_i}.$

Finally, we get:

$$\nabla_{b_h}\ell = \nabla_{\tilde{h}}\ell.$$

2 Implementation

2.1 Neural networks

1. **Discuss and analyze your experiments following the implementation. Provide pertinent figures showing the evolution of the loss; effects of different batch size / learning rate, etc.**

We experimented on the neural networks by training on a dataset made of two classes, with the first encircling the second. We trained four different networks, each made differently. They should be very similar as the architectures are the same and the only difference is the way we implemented them. We manually implemented everything in the first one, we used PyTorch's autograd for the backward pass in the second one, we replaced the forward pass by PyTorch's torch.nn in the third one and finally in the fourth one we simplified the SGD with PyTorch's optimizer. Here is how they compare.
We can see that they all behave a little differently during training. However it is important to first note that the results are very similar as all four networks have a final accuracy of around 94%, which was expected since the architectures are the same. The aspect of the curves slightly differs from one implementation to the other because of the random initialization of the weights.
In the experiments above we used a learning rate of 0.07. This number can be increased a little bit to make our models converge faster. However, if we set it too high there will be a risk of divergence. As we can see in the figure

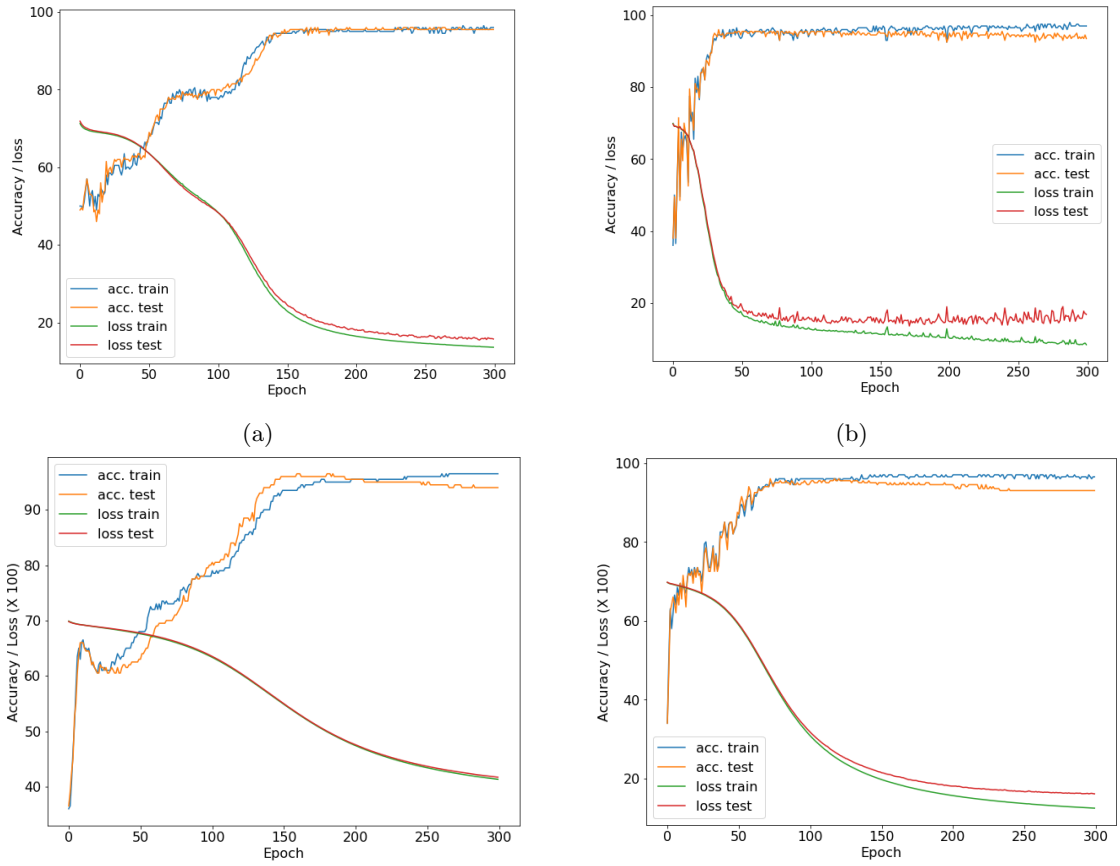


Figure (1) Graphs illustrating the evolution of the loss and the accuracy during the training. (a) was a fully implemented neural network, (b) was a network using autograd, (c) used autograd and torch.nn and (d) used autograd, torch.nn and optimizer.

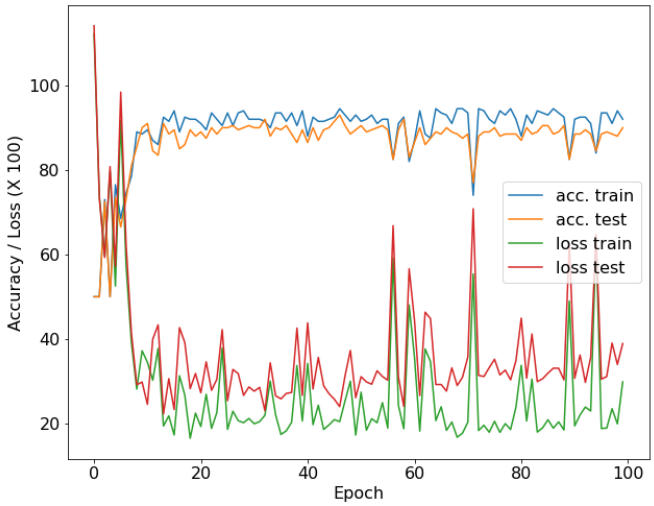
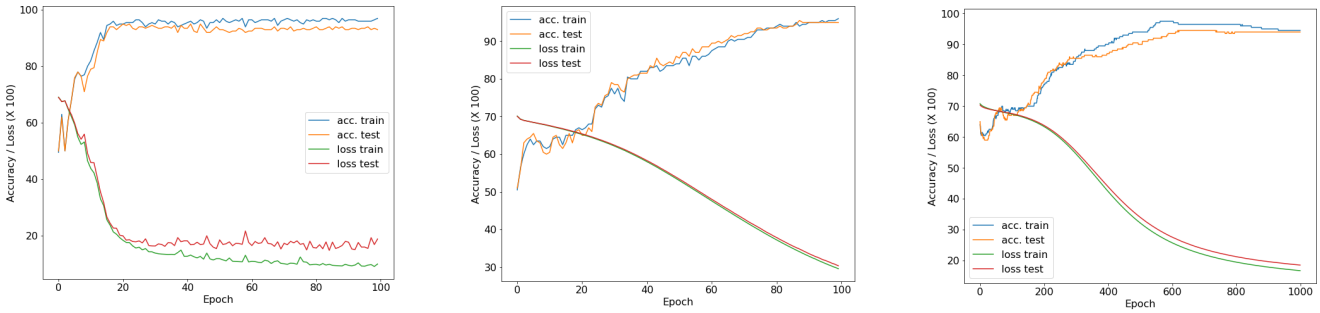


Figure (2) Evolution of the loss and accuracy of a neural network trained with a learning rate of 2.

2, even if the network reaches pretty good scores on the test set, we can not know if it is optimal as it is very unstable.

Finally, we experimented on the batch size. It allows to choose the number of data the networks look at before

updating its weights. On figure 3, we can see the results we obtained when using a small batch size, a reasonable one (which we used for the previous trainings) and a large one.



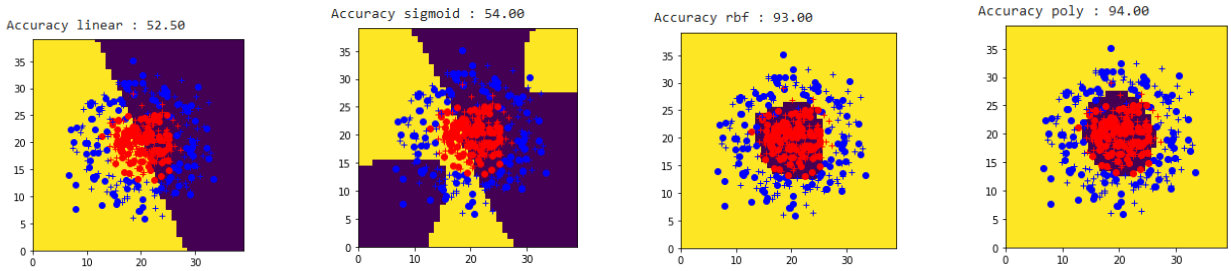
(a) (b) (c)

Figure (3) Evolution of the loss and accuracy of a neural network trained with a batch size of 1 (a), 10 (b) and 50 (c)

We see that using a batch of 50 takes very long to converge and using a batch of 1 is very fast. This was expected because with 50 data per batch, the weights are updated only a few times per epoch. A lot of epochs are thus required in order to update the weights enough time for it to converge.
On the other hand, we see that using a batch size of 1 results in pretty unstable losses and accuracies. This is also expected because the weights are updated with every data which means that the updates are very poorly generalized. The update fits only the data seen and not the rest. Even though using a low batch size can make the training faster as we said, it is also possible that computing the gradient on only one example does not give a good estimation of the gradient thus resulting in divergence or a slower convergence.
In the end, using a batch size of 10 seems like a good trade-off in order for the model to generalize well while still converging fast enough.

2.2 SVM

We wanted to predict the classes of the data from the test set of the circles dataset. We tried different kernel types and the first thing we note is that the linear and sigmoid kernels are not able to predict the classes. This was expected since one class circles the other. It is thus impossible to separate the classes linearly or with a sigmoid function. Also the polynomial kernel is only effective when its degree is even. Otherwise, the results are similar to the sigmoid function as it is not able to circle data.



(a) (b) (c) (d)

Figure (4) Separation of the data using different kernels. (a) linear kernel with C=0.01, (b) sigmoïd kernel with C=0.1, (c) RBF kernel with C=0.01 and (d) polynomial kernel of degree 2 and with C=100.

We will focus on both the RBF kernel and the polynomial kernel with a degree of 2.

```
[0.001, 0.01, 0.1, 1, 10, 100]
scores rbf : [0.95, 0.95, 0.95, 0.94, 0.935, 0.93]
scores polynomial : [0.535, 0.745, 0.955, 0.955, 0.945, 0.94]
```

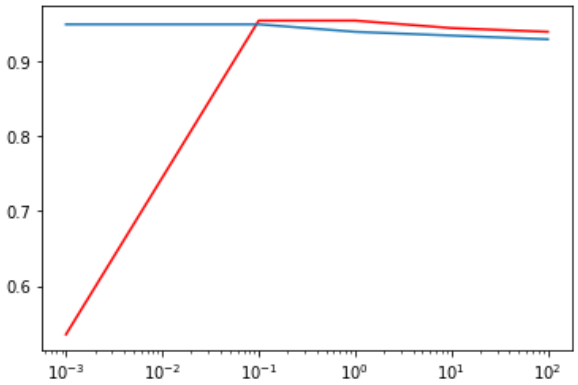


Figure (5) Scores obtained with different values of C. red=poly, blue=rbf.

We can see that both kernel types are very effective and reach a precision up to 95.5% for the polynomial kernel and 95% for the RBF kernel.
As the data are already pretty well separated, the influence of C is lesser. We can see it with the RBF that stays effective whatever the value of C. The polynomial kernel still catches some of the outlying values when the value of C is lower and is less effective.

3 Introduction to convolutional networks

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?**
   The output size will be $x' \times y' \times z'$ where:

   - $x' = \left\lfloor \frac{x-(k-1)-1+2p}{s} \right\rfloor + 1 = \left\lfloor \frac{x-k+2p}{s} \right\rfloor + 1$
   - $y' = \left\lfloor \frac{y-(k-1)-1+2p}{s} \right\rfloor + 1 = \left\lfloor \frac{y-k+2p}{s} \right\rfloor + 1$

   How much weight is there to learn ?
   There will be $(k \times k) \times z$ weights 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 ?
   If a fully-connected layer were to produce an output of the same size, it would have needed $(x \times y \times z) \times 2 \times (x' \times y' \times z)$ because there are two learnable parameters per connection.

2. ★ **What are the advantages of convolution over fully-connected layers ?**
   Using fully-connected layers to process an image wouldn't be a good idea because fully-connected layers ignore the topology of the input data so they are not robust and invariant to small distortions of the image. In addition, the number of weights grows largely with the size of the input image.

   Whereas convolution layers don't have these problems. Indeed, they use shared weights, which allow them to be translation equivariant (*i.e.* translating the input to a convolutional layer will result in translating the output) and to require a dramatically reduced number of parameters. In other words, by outputting 2D feature maps that keep spatial information, they provide a more stable representation of the images.

   What is its main limit ?
   The attention process is done locally, requiring a big number of layers in order to obtain a global assessment of the image.

3. ★ **Why do we use spatial pooling?**
   After each convolutional layer we use spatial pooling in order to add a small amount of translational invariance. This is interesting because in multiple fields such as object recognition where an object could be distorted or more or less big (making its features farther apart), spatial pooling allows to account for multiple locations at the same time making the model resistant to such changes. It's also useful to reduce the size of the feature maps without needing any parameter.

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 ?**
   The convolutional as well as the pooling layers can be applied on any size. On the contrary, the input size of a fully-connected layers depends on its number of connexions so it's fixed. Consequently, this larger image can only be processed by the convolutional network until the last layer preceding the first fully-connected layer.

5. **Show that we can analyze fully-connected layers as particular convolutions.**
   The output of a neuron in a fully-connected layer is simply the sum of a linear combination of the elements of the receptive field and a bias. Therefore, the linear combination can be replaced by the result of a convolution applied on the receptive field with a filter whose size is equal to the number of element of the receptive field.

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 ?**
   It still is not possible to use all of the layers of the network on this image. Indeed, the SoftMax function takes as input as many variables as there are classes. However, since the input image is larger than the initially planned size, instead of just having variables, we still have feature maps, of size $(r+1) \times (c+1)$ where a and b are respectively the number of rows and columns added. The interest is that we can apply a Global Average Pooling (GAP) on these feature maps in order to reduce each of these maps to its average value. Once we do that, we are able to process images larger than the initially planned size. This is a much smarter approach than the approach that consists of sliding a window on the whole image to detect the object we are interested in. Moreover, this technique using GAP can be used to do localize an object in an image in a weakly supervised manner.

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 ?**
   The size of the receptive fields of the neurons of the first convolutional layer is k_1 where k_1 is the kernel size of the first layer. The size of the second one is $(k_1 + (k_2 - 1) \times s_1)^2$ where k_2 is the kernel size of the second layer, s_1 the stride of the first layer and s_2 the stride of the second one. If $k_1 = k_2 = 3$ and $s_1 = 1$ then the size of the first receptive field is 9 and that of the second one is 25.

   Can you imagine what happens to the deeper layers ? How to interpret it ?
   The deeper the layer is, the larger its receptive field is and the more global and then abstract features it's able to detect. For instance, elementary local features detected in the first layers can be lines whereas the more global and abstract features detected in the last layers can be faces or cars.

4 Training from *scratch* of the model

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 ?**
   We have to solve $\left\lfloor \frac{x-k+2p}{s} \right\rfloor + 1 = x$ and $\left\lfloor \frac{y-k+2p}{s} \right\rfloor + 1 = y$ for p and s. Since we don't want to reduce the spatial dimensions, we can choose the smallest possible stride size, which is 1. Now, given the fact that $x - k + 2p$ and $y - k + 2p$ are integers, these equations are easily solvable and we get $p = \frac{k-1}{2}$.

9. **For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed ?**
   We want to reduce the spatial dimensions by a factor of 2 while minimizing the loss of information so we don't

want any padding. Let's choose it equal to 0. Now, we have to solve $\left\lfloor \frac{x-k}{s} \right\rfloor + 1 = \frac{x}{2}$ and $\left\lfloor \frac{y-k}{s} \right\rfloor + 1 = \frac{y}{2}$. The solutions are $s = k = 2$.

10. ★ **For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.**
The input size of the dataset is $32 \times 32 \times 3$. Besides, it's assumed that:

    - every filter/neuron has a bias to learn;
    - the spatial dimensions of the output are the same as that of the input.

layer	**description**	**output size**	**number of weights to learn**
conv1	32 convolutions 5×5, followed by ReLU	$32 \times 32 \times 32$	$32 \times [(5 \times 5 \times 3) + 1] = 2,432$
pool1	max-pooling 2×2	$16 \times 16 \times 32$	0
conv2	64 convolutions 5×5, followed by ReLU	$16 \times 16 \times 64$	$64 \times [(5 \times 5 \times 32) + 1] = 51,264$
pool2	max-pooling 2×2	$8 \times 8 \times 64$	0
conv3	64 convolutions 5×5, followed by ReLU	$8 \times 8 \times 64$	$64 \times [(5 \times 5 \times 64) + 1] = 102,464$
pool3	max-pooling 2×2	$4 \times 4 \times 64$	0
fc4	fully-connected, 1000 output neurons, followed by ReLU	1000	$1000 \times [(4 \times 4 \times 64) + 1] = 1,025,000$
fc5	fully-connected, 10 output neurons, followed by SoftMax	10	$10 \times (1000 + 1) = 10,010$

11. **What is the total number of weights to learn ? Compare that to the number of examples.**
We have 50,000 examples and a model with 1,191,170 parameters. The fact that the number of weights to learn is much bigger than the number of training examples is positively correlated to the risk of overfitting.

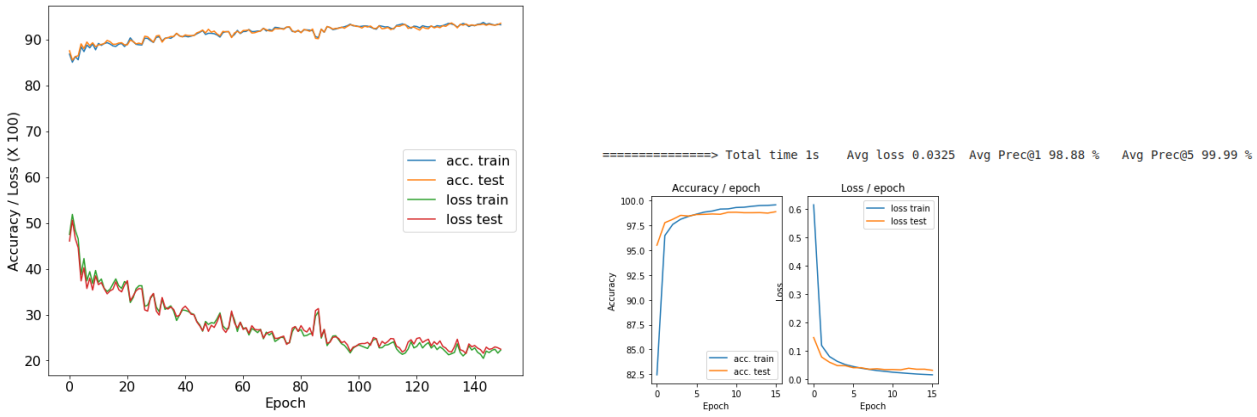
12. **Compare the number of parameters to learn with that of the BoW and SVM approach.**
The BoW+SVM approach requires one SVM for each of the 10 classes we have and each each of these SVM has a number of learnable parameters equal to the number of words from which the input data are encoded. This number was equal to 1000 in the last practical work so in total this approach mainly requires 10000 parameters which is more than 1000 times smaller than the total number of parameter of the convolutional network approach. However, the latter has the advantages to allow an end-to-end learning (of features) whereas the Bow+SVM approach relies on handcrafted features and each of its steps (extraction of features, clustering, SVM classification) need to be processed separately, which is tedious.

13. **Read and test the code provided.**
It takes only a few epochs for the convolutional neural network to stabilize and reach much higher scores than the network from part 1. The fully connected layers network reached scores up to 94.9% while the convolutional network reached 98.9%.

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 difference in data) ?**
The same functions are called to calculate the loss and accuracy in train and in test. The main difference is the moment at which they are called. In fact, the loss and accuracy in train are generated before test because during the training the weights are modified. The loss in train is used to calculate the gradients and optimize the network's weights while the loss in test is simply used to evaluate the model. We thus generate the loss in test after we generate loss in train because the loss in test allows to evaluate how effectively were the weights modified.

15. **Modify the code to use the CIFAR-10 dataset and implement the architecture requested above.**

    We can see that our convolutional model is not very effective as it reaches a maximum accuracy of around



(a) (b)

Figure (6) Scores reached by the fully connected layers network (a) and the convolutional network (b).

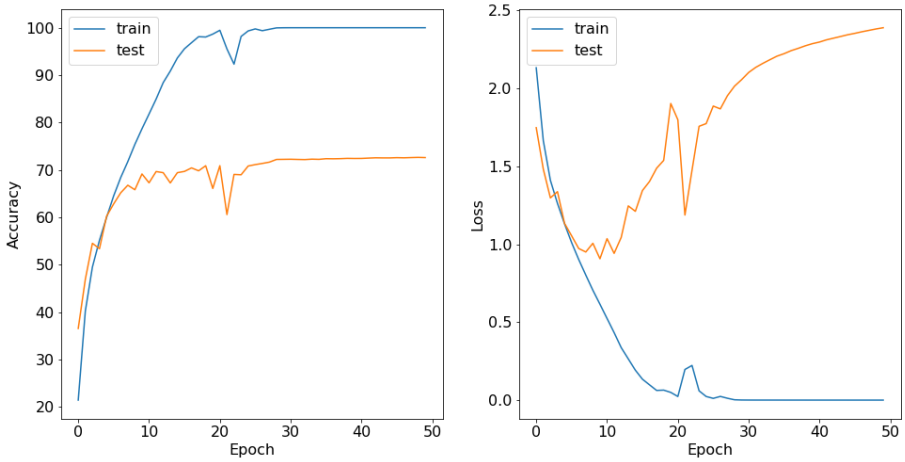


Figure (7) Evolution of the loss and the accuracy during the training of the convolutional network on the CIFAR10 dataset.

70%. The accuracy does not improve because the network manages to be perfect on the training set and thus can't learn anymore. There are two reasons that could explain why the network manages to be so perfect on the training set and not on the test set. The first one is that the data might be too "easy" in the sense the objects are too identifiable on the training images and they are too different from one class to the other. The second reason is simply that there is not enough data in the training set. The consequence is that the model will have few examples for each class and it will not be able to generalize when seeing the data from the test set.

16. ★ **What are the effects of the *learning rate* and of the *batch-size* ?**
The smaller the batch-size, the more we can parallelize! When we lower the batch size, we reduce even more the network's ability to generalize. As a result, the loss is even more unstable and the accuracy stays pretty bad. Even on the training's loss we see that the network struggles to generalize and we see the loss increase. When we set it too high, We see that the model takes way too long to converge. Moreover, it generalizes too much and accuracies reached are not really interesting.
As for the learning rate, and as we discussed earlier in this report, a big learning rate keeps the network from converging so we need to not set it too high in order for the training to be fast enough while still converging.

17. **What is the error at the start of the first epoch, in train and test ? How can you interpret this ?**
At the start of the first epoch, the accuracy is around 10%. This was expected since there are 10 classes so if they are picked randomly there is one chance out of 10 that it will be correct which is 10%.

18. ★ **Interpret the results. What's wrong ? What is this phenomenon ?**

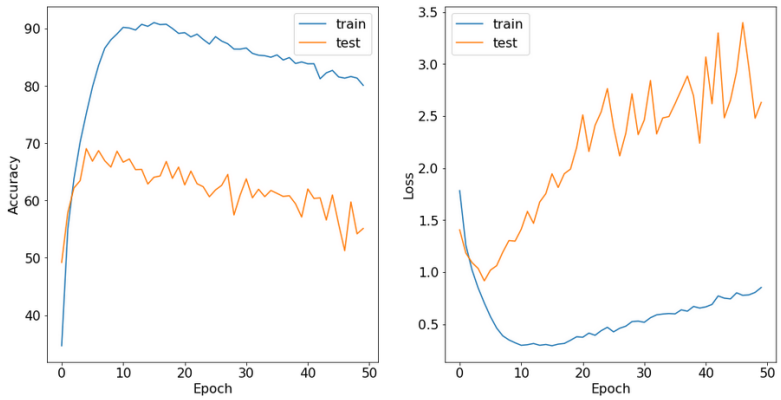


Figure (8) Evolution of the loss and accuracy during a training with a batch size set to 32.

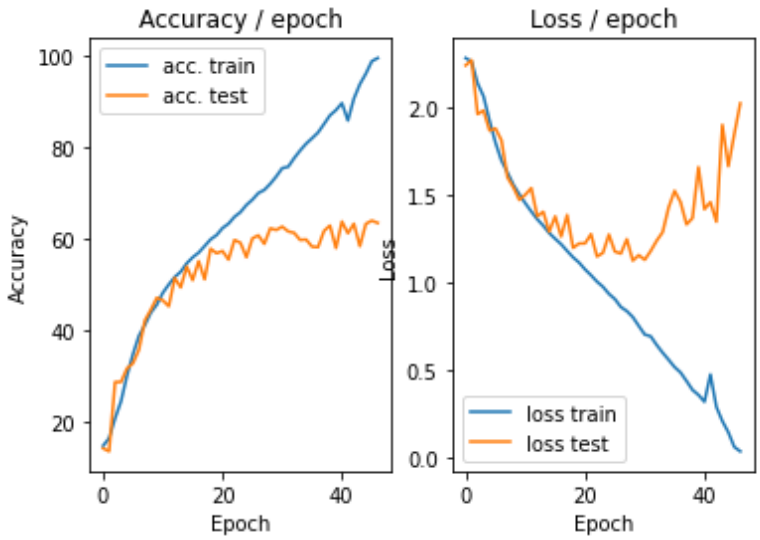


Figure (9) Evolution of the loss and accuracy during a training with a batch size set to 512.

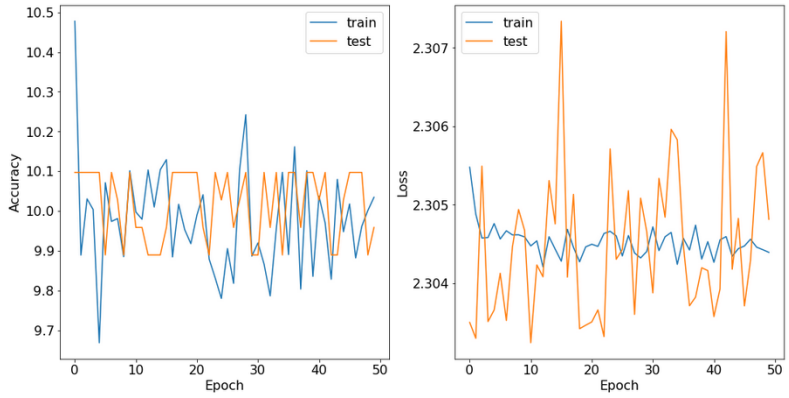


Figure (10) Evolution of the loss and accuracy during a training with a learning rate of 1.

As we said above, the results are not great. We mentioned that the network achieves perfect accuracy on the training set. Also the loss on the test set starts to increase around the tenth epoch. This means that our network overfits on the training set. It is too precise on the the images from this set and it is not able to generalize on the other images such as the ones in the test set.

5 Results improvements

★ 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.

19. **Describe your experimental results.**
The first effect of normalization we notice is that the network converges faster. We can also see an improvement in the accuracies as the network now reaches an accuracy of 75.8% compared to the previous 70%. These improvements can be explained because we 'evened' the data. In fact, the learning could be disturbed by data with extreme values in its calculations. Normalizing allows to smooth these values for more meaningful processing.
However the data sets stay similar and the learning follows a similar path as before, resulting again in overfitting.

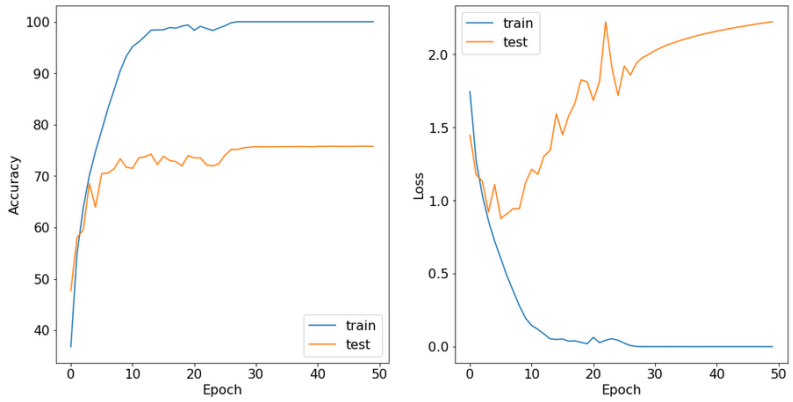


Figure (11) Performance of the network when we normalize the data.

20. **Why only calculate the average image on the training examples and normalize the validation examples with the same image ?**
First, it's important to note that the normalization of the validation examples is needed because they need to be in the same condition that the normalized examples on which the network is trained.

Computing the average image from the test or validation set would result in a bias so we use the average image of the training set. It makes sense to do that because we assumed that the distribution of the training examples is representative of that of the test set.

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.**
Among the other existing normalization schemes, we can mention Principal Component Analysis (PCA) and Zero-phase Component Analysis (ZCA).

PCA In order to normalize images with PCA, we first need to pre-process the dataset such that each pixel has a zero mean and unit variance. However, in natural images a pixel is far from being independent from its nearby pixels. Consequently, decorrelating the pixels by projecting them into the eigenbasis and keeping only the top few eigenvectors would completely destroy the natural spatial-correlation.

ZCA ZCA is a kind of whitening transformation, *i.e.* a linear transformation that transforms a vector of random variables with a known covariance matrix into a set of new variables whose covariance is the identity matrix. Therefore, if we pre-process the images with ZCA, only the linear dependencies between pixels will be removed. This implies that the pre-processed images will still be efficiently recognizable by the convolutional network since it uses local attention in the first layers.

22. **Describe your experimental results and compare them to previous results.**
Again, our results are improving. Adding crops and horizontal symmetries allows to reach an accuracy of

about 80%. The main reason for this improvement is the adding of new data. The more a network is given data to be trained on, the better it will be. Here we add data by modifying the one we already have which proves to be effective. Of course, having more data to train on means that the training will take longer and we in fact see that the model converges later than before (around the twentieth epoch against the tenth fro the previous training). It is not as obvious as before but the model overfits again starting around the twentieth epoch. This is because the network does not manage to be perfect on the training set, simply because there is more data which might even be harder for the model since some of the added data is flipped.

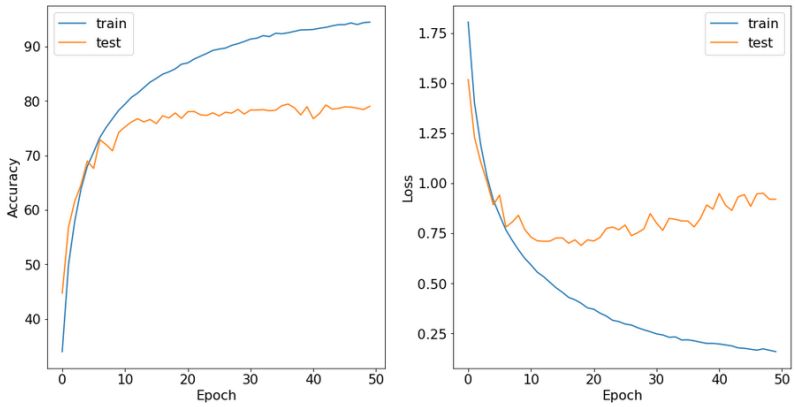


Figure (12) Network's perfromance when trained with cropping and flipping augmentations.

23. **Does this horizontal symmetry approach seems usable on all types of images ?**
No because this horizontal flipping can change the meaning of the images.

In what cases can it be or not be ?
For instance, if we want to train a network to detect all biological cells in an image then this approach is relevant because a flipped cell still looks like a cell.

On the contrary, if we want to train a network to classify the letter present in an image, into one of the 26 classes of the possible letters, using this horizontal symmetry approach would worsen the result. For example, a flipped image containing the letter "M" is an image that seems to contain a "W".

24. **What limits do you see in this type of data increase by transformation of the dataset ?**
Identifying an optimal data augmentation strategy is difficult.

Some more sophisticated data augmentation strategies exist but they are tricky to implement because for instance they require the use of Generative Adversarial Networks.

If the dataset is already large, the learning of the augmented one will require additional time and power.

25. **Bonus : Other data augmentation methods are possible. Find out which ones and test some.**
Among the numerous data augmentation methods available in PyTorch, we can mention:

- **ColorJitter** Randomly change the brightness, contrast, saturation and hue of an image; The problem with changing an image's hue is that it changes objects' color which is one of the aspects that allow to identify them. That is what happened in our experiment, explaining the poor performance.
The intensity and the contrast usually are very interesting to change in order to augment our data. In fact, an object captured in different configurations usually appears in different intensities and contrasts on the images. In our case, we suspect that the images are too small and changes in intensity and contrast are too powerful on the few pixels of the images. As a result, the network performs better than when we modify hue but much worse than when we only crop and flip.
- **RandomAffine** Random affine transformation of the image keeping center invariant; Much like modifying the hue of an image, affine transformations can be dangerous for object recognition. An object can be identifiable depending on its orientation. It is thus no surprise to see our model perform pretty badly with this augmentation.

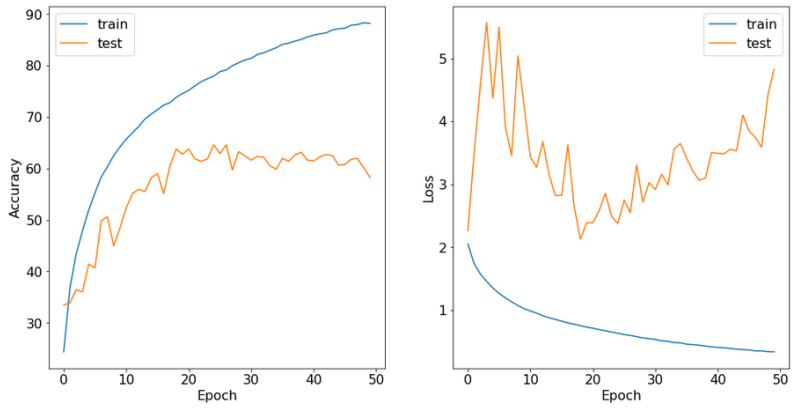


Figure (13) Network's performance when trained with a modification of intensity and hue.

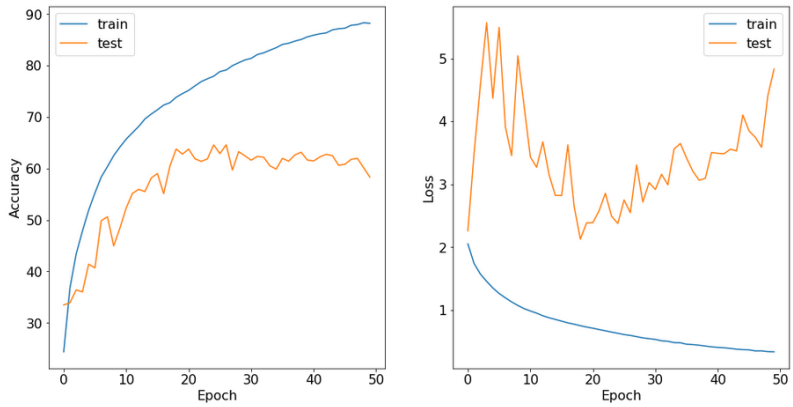


Figure (14) Network's performance when trained with a modification of intensity and contrast.

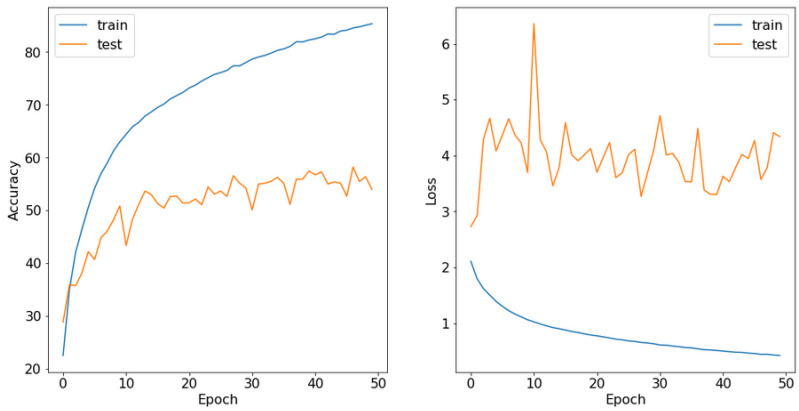


Figure (15) Network's performance when trained with a affine transformation augmentation.

- **RandomPerspective** Performs a random perspective transformation of the given image with a given probability. This augmentation is interesting because objects often look different when we look at them from another perspective. Simulating this by modifying the original images can help the network generalize better. In fact this augmentation is actually the best one we tested in this question. The best scores on the test set are similar to what we obtained with only the crop and flip augmentations.
- **RandomRotation** Rotate the image by angle; It can already be done in the RandomAffine augmentation mentionned above.
- **RandomVerticalFlip** Vertically flip the given image randomly with a given probability; It is similar to the horizontal symmetry we already performed.

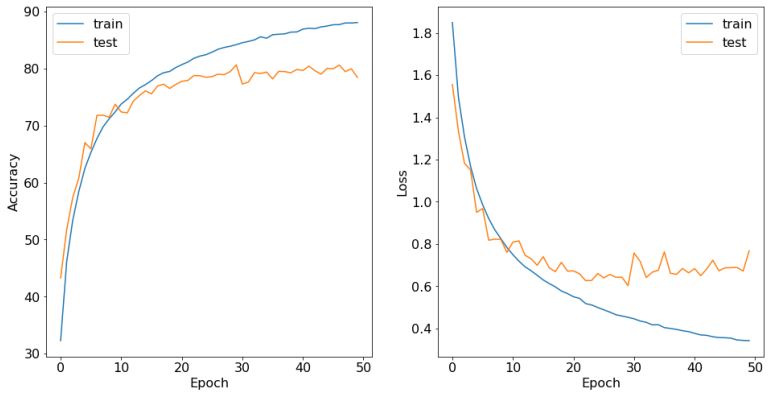


Figure (16) Network's performance when trained with a perspective trnaqformation augmentation.

- **GaussianBlur** Blurs image with randomly chosen Gaussian blur.
  This augmentation could be pretty effective. In fact, whatever the way we acquire the image, there is always a chance that the quality will be poor. Often, this degradation in quality takes the form of a blur on the image. Thus, applying this augmentation makes the network more robust to poor quality images. However in our case, the quality of the image is probably pretty good and similar in both the training and test set. This means that adding blurring makes the network generalize too much and it is less effective than without blurring (around 74% max accuracy).

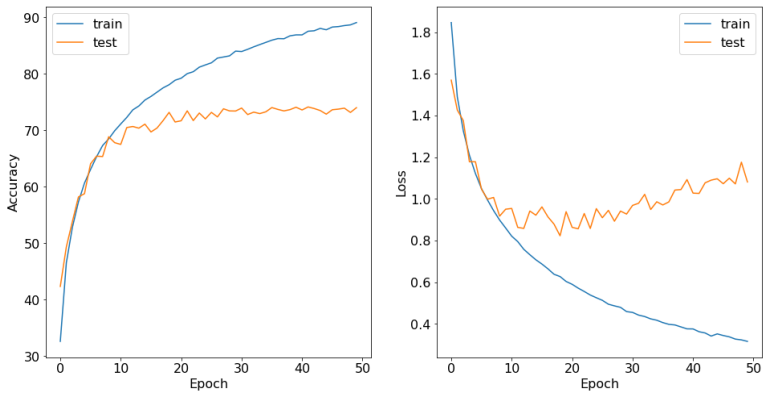


Figure (17) Network's performance when trained with a blurring augmentation.

The relevance of each of these transformation depends not only on the data but also on the intended application. In our case, we won't use any of them as they do not really improve our results and they consequently increase the training time.

26. **Describe your experimental results and compare them to previous results, including learning stability.**
    The learning rate scheduler slightly improves our results. Refining the learning rate over time allowed to gain 1% in accuracy on the test set. The other thing we notice after adding it is that the loss and accuracy curves are smoother. In figure 12, the loss and accuracy curves on the training set are not as smooth as in figure 18. For the test set, we can see more peaks in figure 12 than in figure 18.

27. **Why does this method improve learning ?**
    When we use a constant learning rate with this network, the algorithm tends toward a good minimum. However, it keeps wandering around a minima without being able to completely reach it. Actually, it's the fact that the learning rate is too big to avoid divergence that prevents it from going deeper in the landscape of the loss function. Besides, it allows to have an initially large learning rate to accelerate training and escaping spurious local minima that we may encounter at the beginning of the learning.

    An initially large learning rate may also help the network from memorizing noisy data while decaying the

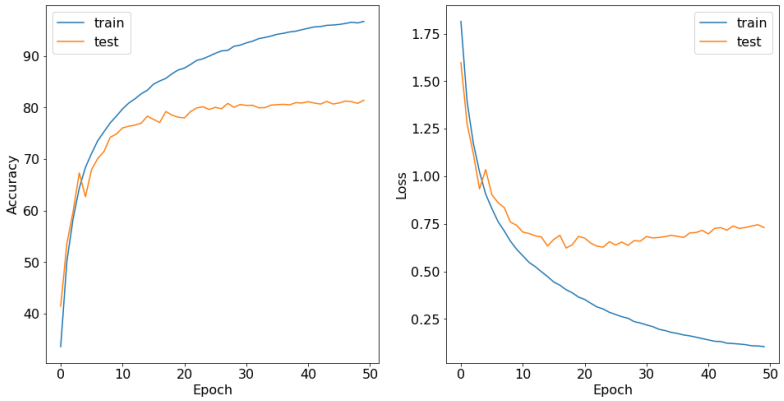


Figure (18) Network's performance with a learning rate scheduler

learning rate improves the learning of complex patterns (*cf.* the paper "How does learning rate decay help modern neural networks?" by Kaichao You *et al.*, published in 2019).

28. **BONUS : Many other variants of SGD exist and many *learning rate* planning strategies exist. Which ones ? Test some of them.**

Given the fact that the update of the SGD algorithm is based on an approximation of the true gradient, there can be a lot of oscillations that can cause the convergence to be very slow. To attenuate this problem, instead of using the vanilla SGD:

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}(x,y)}{\partial w},$$

we can use SGD with momentum:

$$v_{\text{mom}}^{t+1} = \beta_1 \cdot v_{\text{mom}}^{t} + (1-\beta_1) \cdot \frac{\partial \mathcal{L}(x,y)}{\partial w},$$

$$w \leftarrow w - \eta \cdot v_{\text{mom}}^{t+1},$$

where $v_0 = 0$ and β is typically equal to 0.9.

Among the other common existing learning rate planning strategies, we can mention the:

- **step decay** $\eta_t = \eta_0 \times r^{\frac{t}{t_u}}$;
- **inverse (time-based) decay** $\eta_t = \frac{\eta_0}{1+r \cdot t}$;

where η_0, t_u, r are hyper-parameters and t is the iteration number.

There also exist methods like Root Mean Square propagation optimzer (RMSprop) or Adam that can adaptively the learning rates, and even do so per parameter.
We experimented with the step decay which proved to decrease too fast when we set the learning rate to 0.1 and the step size to 0.002. Exponential decay proves to be way more adapted to the way the learning evolves as it adapts to the fast evolution at the start and slow at the end.

RMSProp The update of this optimizer is:

$$v_{i,rms}^{t+1} = \beta_2 \cdot v_i^t + (1-\beta_2) \cdot \left(\frac{\partial \mathcal{L}(x,y)}{\partial w} \odot \frac{\partial \mathcal{L}(x,y)}{\partial w}\right),$$

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{v_i^{t+1}}} \cdot \frac{\partial \mathcal{L}(x,y)}{\partial w_i}.$$

The more a parameter makes the estimate of the cost function oscillate, the more its update is penalized. In our experiments, we find that this optimizer is less efficient than the SGD.

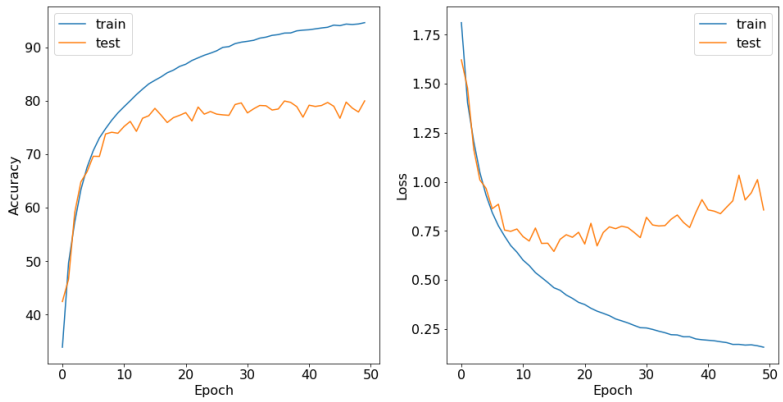


Figure (19) Network's performance when scheduling the learning rate with a step decay.

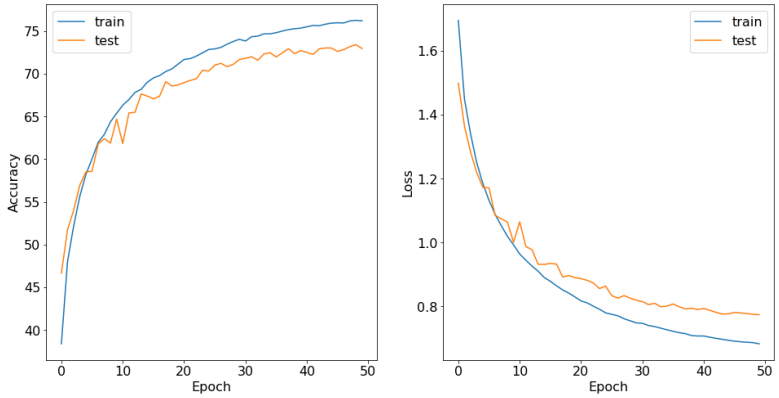


Figure (20) Network's performance when optimizing with the RMSprop method.

Adam The update of this optimize is given by:

$$v^{t+1}_{i,\text{corrected mom}} = \frac{v^{t+1}_{i,mom}}{1-\beta_1^t},\ v^{t+1}_{i,\text{corrected rms}} = \frac{v^{t+1}_{i,rms}}{1-\beta_2^t}$$

$$w_i \leftarrow w_i - \eta \cdot \frac{v^{t+1}_{i,\text{corrected mom}}}{\sqrt{v^{t+1}_{i,\text{corrected rms}}}+\varepsilon} \cdot \frac{\partial \mathcal{L}(x,y)}{\partial w_i}.$$

where ε is a small number to avoid division by zero.

We can see that it's a kind of a fusion of the momentum and RMSProp optimizers. The bias corrections are just here to adjust for a slow startup when estimating momentum and a second moment because they are initialized to zero.
In our experiments, Adam achieves very similar results to the SGD with an accuracy on the test set almost reaching 81%.

However, SGD, SGD with momemtum, RMSProp and Adam have a common drawback. Indeed, when there mutltiple dimensions, the loss can reach a minimum along one but not the others. This means that the gradient will be equal to 0 and the descent will stop. A type of optimisation based on Newton's method solves this problem by using the following update :

$$w \leftarrow w - [H_{\mathcal{L}}(X,Y)]^{-1} \cdot \frac{\partial \mathcal{L}(X,Y)}{\partial w},$$

where HL(X, Y) is the Hessian matrix (of the loss function) which intuitively describes the local curvature of the loss function. The update is more efficient and it does not require any hyper parameter such as the learning rate. The problem of this method is that it very complex to compute the Hessian matrix and its

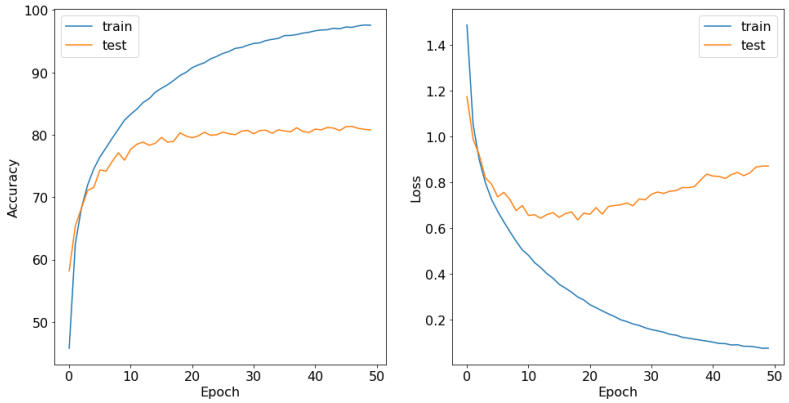


Figure (21) Network's performance when optimizing with the Adam method.

inverse so it is hardly usable in our context of deep learning. However, some methods have been developed to get close to copy this one in a less costly fashion, such as L-BFGS.

29. **Describe your experimental results and compare them to previous results.** Adding the dropout

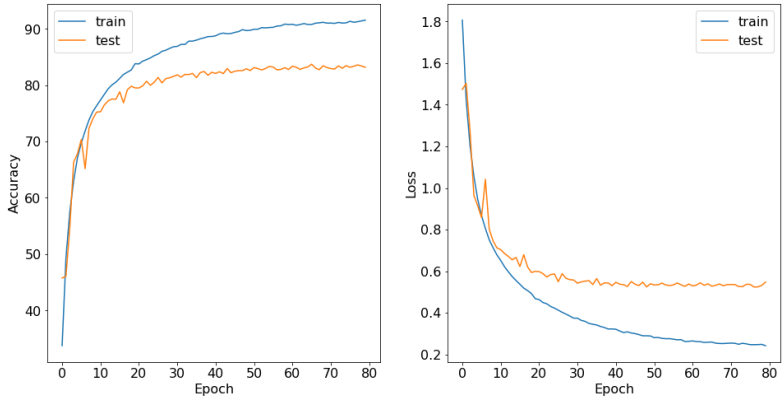


Figure (22) Network's performance with a dropout layer added.

allows us to improve even more the accuracy on the test set. In fact we reach a score of around 82%. We also find that overfitting is not as high as it was before, both the losses and accuracies converge now.

30. **What is regularization in general ?**
Regularization is to make results more simply explainable. Generally we use for two different purposes:

    (a) Limiting the complexity of a machine learning model in order to prevent overfitting and increase its ability to generalize. This limitation is done by removing the least impactful variables (*e.g.* dropout).

    (b) Finding solutions to ill-posed problems (*e.g.* to impose sparsity in compressed sensing).

    In both cases, these constraints are often defined by the addition of a penalization term.

31. **Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it ?**
By preventing feature co-adaptation (feature only helpful when other specific features present), the Dropout technique significantly improves generalization and therefore reduces overtifitting (especially when there are huge fully connected layers). One can interpret a neural network with dropout as an average of many neural networks.

    We will also understand in a future class of RDFIA that:

    - a neural network with dropout can be precisely interpreted as a variational Bayesian approximation "with particular prior and approximate posterior";

- this interpretation offers an explanation to some of dropout's key properties, such as its robustness to over-fitting.

32. **What is the influence of the hyperparameter of this layer ?**
This hyper-parameter corresponds to the probability of a neuron of this layer to be disabled during a pass. If it's too big then the network will underfit the data and will be unstable. If it's too small the potential of dropout to correct overfitting will not be exploited enough.

33. **What is the difference in behavior of the dropout layer between training and test ?**
During the training neurons are randomly disabled whereas when we want to test the network we have to use all the neurons. Furthermore, in PyTorch the outputs are scaled by a factor of $\frac{1}{1-p}$. This allows us to make sure the outputted numbers of the layers in train and in test have approximatively the same scale even if the number of neurons from which they depend is not the same.

34. **Describe your experimental results and compare them to previous results.**
Finally, adding batch normalization brings the score to a maximum of 85%. This is effective because similarly

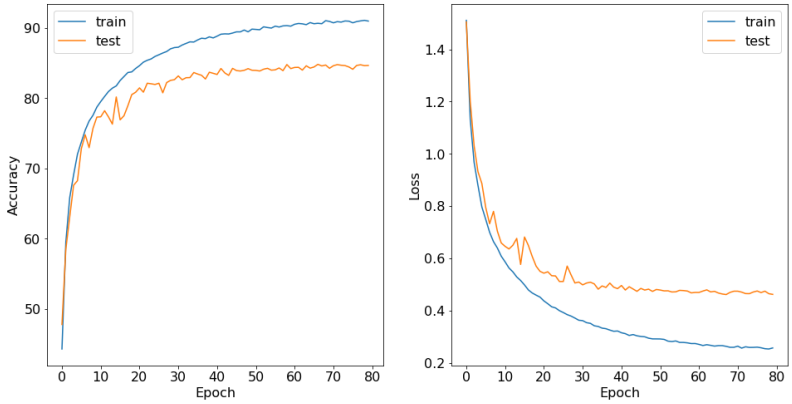


Figure (23) Network's performance with a batch normalization added.

to the impactful normalization we did earlier in this work, outputs of the layers are not renormalized and it is possible to find extreme values among those.
Also, batch normalization is interesting because the training is then less sensible to poor parameter tuning. It also acts as a regularization technique as the extreme values are attenuated by the normalization.