TP 2 (a), (b), (c) et (d) : Reconnaissance des formes pour l'analyse et l'interprétation d'images

Réalisé par :

Gaël Marec et Léo Hein

Table des matières

The goal of this practical work is to build a simple neural network and learn how to train it with backpropagation of the gradient. To do so, we will begin by theoretically studying a perceptron with a hidden layer and its learning procedure. We will then implement this network with the PyTorch library on a toy problem to check that it is working correctly, then on the MNIST dataset. Next, in order to become more familiar with convolutional neural networks, we will study their classical layers and 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.

1 Theoretical foundation

To apply a neural network to a machine learning problem, we need 4 elements adapted to the problem : • A supervised dataset • A network architecture • A loss function that we will optimize • An optimization algorithm to minimize the loss function. Each following subsection will deal with one of these elements.

1.1 Supervised dataset

(1) The training and testing sets are drawn from the same distribution. The training set is used to train our model while the testing set is only used to test our model, meaning no information of the testing set is used during the learning process. This aims at avoiding overfitting (our model learns too much the specificity of the data, which is irrelevant when and applied to other data). The validation set is generally used to set up the right hyperparameters. Indeed, all hyper parameters that control model capacity will automatically choose the model of highest capacity, resulting in overfitting. That's why we need a set that neither the training nor the testing set observe.

(2) The more the model is given examples, say a number N, the better the model will learn because it can better draw the relations between the features of the examples and their labels, which is true only if the examples are diversified. In addition, the more example the model has to learn, the longer is the computation time.

1.2 Neural architecture

(3) Activation functions present several benefits. First, it weights the input of the next layer. It also adds non linearity in the process, which is mathematically necessary to approximate non-linear functions, by changing the space of representation.

(4) n_x is the number of features that x contains, n_y is the amount of in our classification problem, n_h the number of neurons in the hidden layer, also called the hidden layer width. On the figure 1 we have, $n_x = 2$, $n_h = 4$, $n_y = 2$. In practice, n_x and n_y are given because they depend on the data set. n_h is an hyperparameter we need to wisely fix. Indeed, a too large width can cause overfitting while a too small one might

may not be able to learn the dataset.

(5) The vector $\hat{y}$ is the estimation of y made by the neural network, y is the true value of the label associate with x. To make the model better we want to minimize in a certain way the difference between y and $\hat{y}$.

(6) The *SoftMax* function is differentiable so we will be able to calculate the gradient descent easily. Moreover the *SoftMax* function is particularly efficient for classification problems.

(7) First, we have $\boxed{\tilde{h} = W_h X + b_h}$ with $X \in \mathcal{M}_{n_x,1}(\mathbb{R})$ the vector of the features. Then, $\boxed{h = \tanh(\tilde{h})}$. Then, $\boxed{\tilde{y} = W_y h + b_y}$. Finally we get $\boxed{\hat{y} = \text{SoftMax}(\tilde{y})}$

1.3 Loss Function

(8) $\forall i$, the vector $y^{(i)}$ is one-hot encoded, so for the cross-entropy loss we can write : $\mathcal{L}(X,Y) = \frac{1}{N}\sum_i l(y^{(i)}, f(x^{(i)})) = -\frac{1}{N}\sum_i \log(\hat{y}_{k_i}^{(i)})$. It is then obvious that we want $\hat{y}_{k_i}^{(i)}$ as close to 1 as possible ($\forall i$) to minimize the global loss. For the same reason, we have for the MSE : $\mathcal{L}(X,Y) = \frac{1}{N}\sum_i l(y^{(i)}, f(x^{(i)})) = \sum_i (||\hat{y}^{(i)}||^2 + 1 - 2\hat{y}_{k_i}^{(i)})$. Thus we also want in that case to maximise $\hat{y}_{k_i}^{(i)}$ to minimise the loss, i.e we want to get $\hat{y}_{k_i}^{(i)}$ as close to 1 as possible.

(9) When deriving the cost function from the aspect of probability and distribution, we can observe that MSE happens when we assume the error follows a normal distribution of our data and cross entropy happens when we assume a binomial distribution. It means that implicitly when we use MSE, we are doing regression (estimation) and when we use cross-entropy, we are doing classification. Both can be seen as a maximum likelihood estimators, simply with different assumptions about the dependent variable, some assumptions corresponding to regression problems and others to classification ones.

1.4 Optimization algorithm

(10) The classic descent is exhaustive and theoretically perfect, but it can be a very slow algorithm as it calculates every differentials. On the other hand, the online stochastic version is by far the fastest version but has more trouble converging. Finally, the mini-batch stochastic gradient represents the trade-off between these two first versions, i.e a trade-off between convergence and computation time. As a matter of fact, the stochastic gradient descent by mini-batch with a mini-batch size adapted to the problem usually is the best method to use among those three.

(11) The learning rate η influences the speed at which the new weights replace the new, so it represents the speed at which the model learns. If η is too small, the gradient descent can converge slowly or make the weights stuck around a local minimum. If η

is too big, the descent can go too far and never converge.

(12) The complexity of calculating the gradients of the loss with respect to the parameters using the backprop algorithm grows linearly with the number of edges in the neural networks while it is a lot larger with the naive approach. For example, using the earlier one hidden layer neural network example, we have $n_x n_h + n_h n_y = 16$ gradients to compute with the backprop approach and $2n_x n_h n_y = 32$ gradients using the naive approach, with each gradient being computed two times during the process, which is a waste of computation time. Now let's take a neural network with a total of d layers, each of width n. The number of gradients to compute in this architecture with the backprop algorithm would be $(d-1) * n^2$ while it would be $(d-1) * n^d$ with the naive approach. As a matter of fact, the computation complexities would respectively be $O(n^2 * L)$ and $O(n^d * L)$ with L the computation time of one gradient.

(13) All functions used in the network architecture must be differentiable in order to compute the gradients. This only concerns the activation functions as linear functions are obviously differentiable.

(14) We have $l(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$. But we know that $\hat{y} = \text{SoftMax}(\tilde{y})$. So, by definition we get $\hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_i \exp(\tilde{y}_i)}$. Thus, $\boxed{l(y, \hat{y}) = -\sum_i y_i \tilde{y}_i + \log\Big(\sum_i \exp(\tilde{y}_i)\Big)}$.

(15) Deriving the formula above we simply get $\frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i$. So we can write the gradient, $\boxed{\nabla^T_{\tilde{y}} l = (\hat{y}_1 - y_1, \cdots, \hat{y}_{n_y} - y_{n_y})}$

(16) Using the results of the 7^{th} question, we get $\tilde{y}_k = \sum_{j \leq n_h} h_j W_{y,k,j}$. So, $\frac{\partial \tilde{y}_k}{\partial W_{y,i,j}} = \delta_{i,k} h_j$. We now can compute :

$$\frac{\partial l}{\partial W_{y,i,j}} = \sum_k \frac{\partial l}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_k}{\partial W_{y,i,j}} = \sum_k (\hat{y}_k - y_k)\delta_{i,k} h_j = \boxed{(\hat{y}_i - y_i) h_j}$$

So $\boxed{\nabla_{W_y} \ell = \nabla_{\tilde{y}} \ell . h^T}$

(17) First we compute $\frac{\partial \ell}{\partial \tilde{h_j}} = \sum_{i,k} \frac{\partial \ell}{\partial \tilde{y}_i} . \frac{\partial \tilde{y}_i}{\partial h_k} . \frac{\partial h_k}{\partial \tilde{h_j}}$ (double application of chain's rule).
And we have $\frac{\partial h_k}{\partial \tilde{h_j}} = \delta_{k,j}(1 - h^2)$ and $\frac{\partial \ell}{\partial \tilde{y}_i} = \hat{y}_i - y_i$ and $\frac{\partial \tilde{y}_i}{\partial h_k} = W_{y,i,k}$.
So $\frac{\partial \ell}{\partial \tilde{h_j}} = (1 - h_j^2) \sum_i (\hat{y}_i - y_i) W_{y,i,j}$ and finally

$$\boxed{\nabla_{\tilde{h_j}} \ell = (1 - h^2) \odot \nabla_{\hat{y}} \ell^T W_y}$$

Using again the chain's rule we have : $\frac{\partial \ell}{\partial W_{h,i,j}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} . \frac{\partial \tilde{h}_k}{\partial W_{h,i,j}}$.

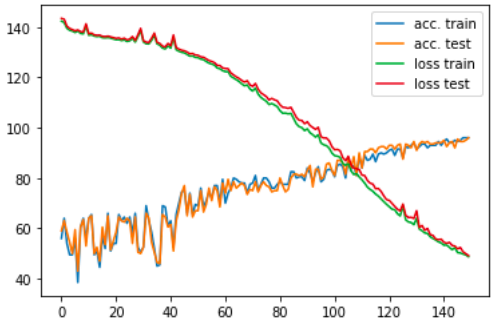
But we have $\frac{\tilde{h_k}}{\partial W_{h,i,j}} = \delta_{k,i} X_j$, it implies that $\boxed{\nabla W_h = \nabla_{\tilde{h}} \ell . X^T}$.

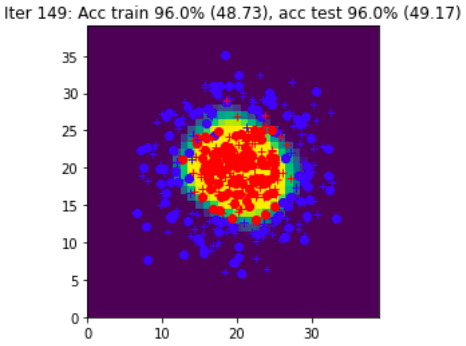
For the last gradient, $\frac{\partial \ell}{\partial b_{h,i}} = \sum_k \frac{\partial \ell}{\partial \tilde{h_k}} . \frac{\partial \tilde{h_k}}{\partial b_{h,i}} = \frac{\partial \ell}{\partial \tilde{h_i}}$. So $\boxed{\nabla_{b_h} \ell = \nabla_{\tilde{h}} \ell}$

2 Implementation

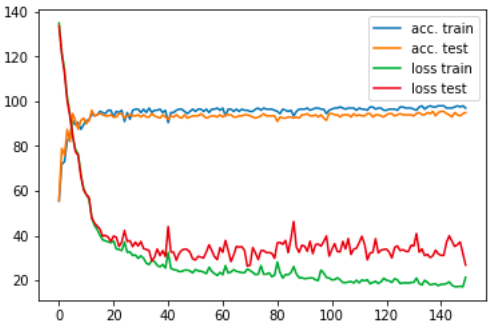
We now have all the equations allowing us to make predictions, to evaluate and to learn our model. In this section, we will implement the network using Pytorch based on the previous theoretical foundations. In particular, we will fill the forward, backward, loss and optimizer functions, first manually and then gradually simplify them thanks to Pytorch possibilities. For each part, we will present the resulting graph and display, before showing the final code of the neural network.

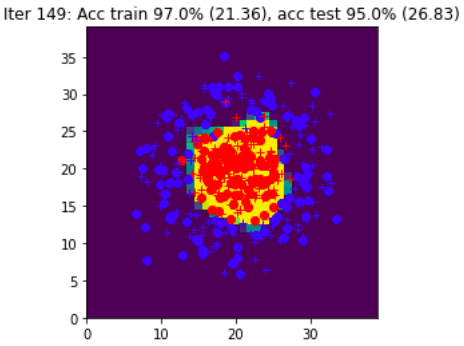
2.1 Forward and backward manuals



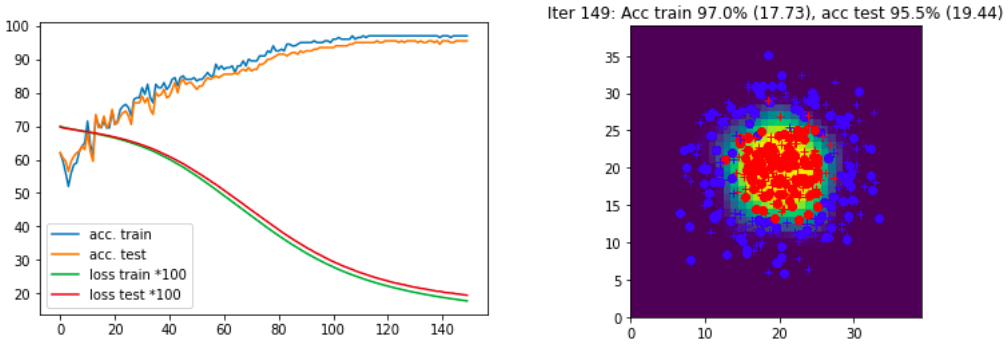


2.2 Simplification of the backward pass

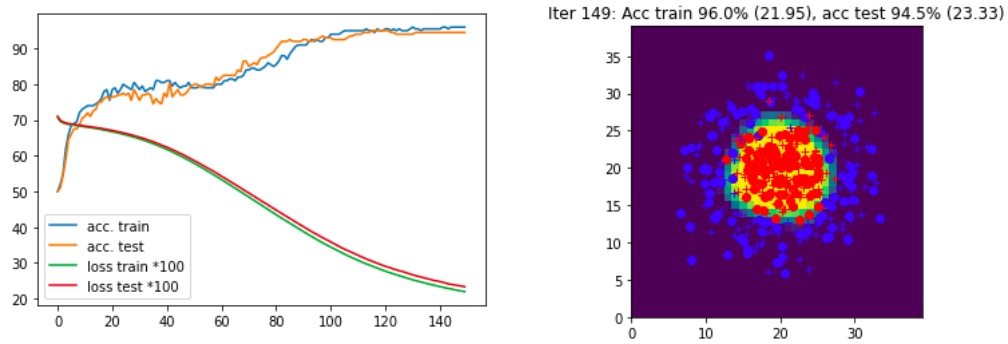




2.3 Simplification of the forward pass



2.4 Simplification of the LGD



The final code of our NN is shown below. First, the $init-model$ function, creating our NN structure and selecting our loss function and optimizer.

```
[18] def init_model(nx, nh, ny, eta):

        model = torch.nn.Sequential(
            torch.nn.Linear(nx, nh),
            torch.nn.Tanh(),
            torch.nn.Linear(nh, ny),
            )

        loss = torch.nn.CrossEntropyLoss()

        optim = torch.optim.SGD(model.parameters(), lr=eta)

        return model, loss, optim
```

Then the $loss-accuracy$ function that calculates the loss and accuracy given the predictions, the true values and the already defined loss function.

```
def loss_accuracy(loss, Yhat, Y):

    _, indsY = torch.max(Y, 1)
    predYhat = (Yhat.gather(1, indsY.view(-1,1)))
    values, ind = torch.max(Yhat, 1)

    L = loss(Yhat, indsY)
    acc = torch.mean(torch.eq(input=predYhat,
                              other=values.unsqueeze(1)).float())*100

    return L, acc
```

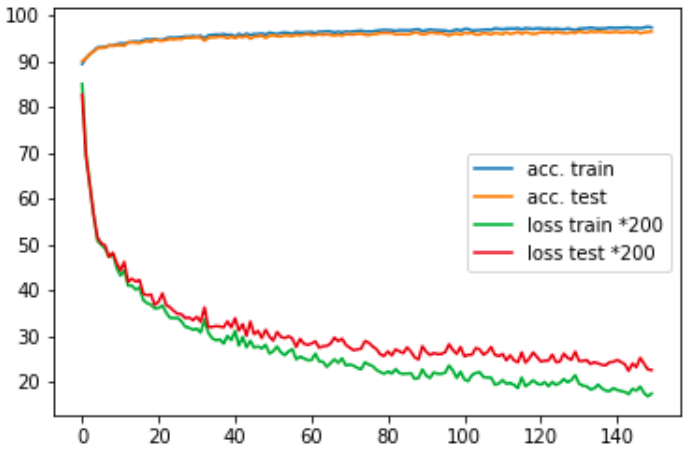
Finally, the main script with the iterations over the epochs.

```
# epoch
for iteration in range(150):

    # permute
    perm = np.random.permutation(N)
    Xtrain = data.Xtrain[perm, :]
    Ytrain = data.Ytrain[perm, :]

    # batches
    for j in range(N // Nbatch):

        indsBatch = range(j * Nbatch, (j+1) * Nbatch)
        X = Xtrain[indsBatch, :]
        Y = Ytrain[indsBatch, :]

        Yhat = model(X)
        L, acc = loss_accuracy(loss, Yhat, Y)
        optim.zero_grad()
        L.backward()
        optim.step()
```

2.5 MNIST application

We now apply this neural network to the MNIST data set by modifying the initialization of our parameters in the main script and slightly reducing the learning rate.

```
Iter 149: Acc train 97.3% (17.49), acc test 96.4% (22.61)
```

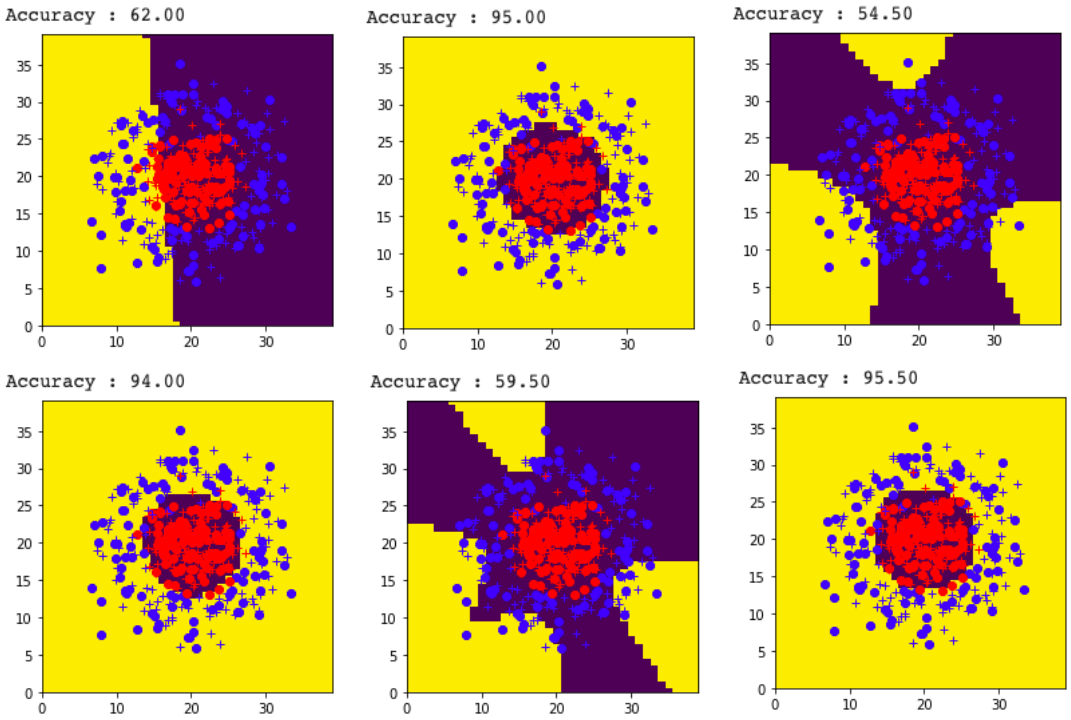


Our neural network shows great performances as it rightly classifies more than 97 percents of our test data after 150 epochs.

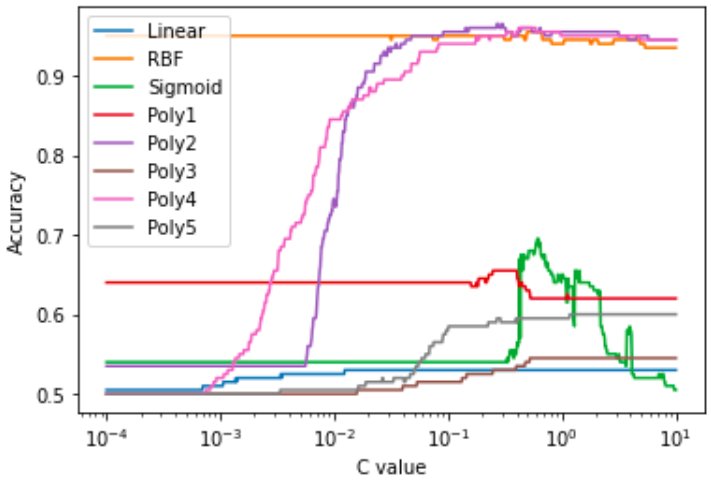
2.6 SVM application

We now train an SVM model to classify the Circles data, with the *sklearn.svm.svc* function. We tried different types of kernels. First, the linear kernel (top left) shows poor performances, which is expected as the data is obviously not linearly separable. On the contrary, the rbf kernel (top middle) shows great and robust performances. Besides, polynomial kernels with even degrees (bottom left and bottom right) give the best performance overall with particular C values while odd degree polynomial kernels (top right and bottom middle) can't apprehend this problem. These results can be

anticipated if we keep in mind the shape of Gaussian or even polynomial functions, creating a "tank" in a higher dimensional space that can easily separate circular data.



On the graph below is shown the evolution of SVMs with different kernels regarding the C value. We won't discuss the C value impact on the different kernels that have difficulties apprehending this problem ; only two types of kernel seem interesting to study. Indeed, the C value has very little impact on the accuracy of the rbf kernel, whereas the performances of even degree polynomial kernels are hugely dependant to this hyperparameter. The C value represents the error budget we allow to our model, meaning that a too small C value will make our model unable to classify noisy data. On the contrary, too large C values will slightly reduce the performances as the classifier may be sub-optimal, allowing too much error. This behaviour is illustrated on the graph by even degree polynomial kernels, showing a peak of performance. The C value doesn't impact the performances of the rbf kernel as it is particularly efficient to classify circular data.



3 Introduction to convolutional networks

In this section we will study the theoretical aspects of a typical convolutional neural network and more particularly its convolution and pooling layers.

(1) For a single convolution filter of padding p, stride s and kernel size k, for an input of size $x * y * z$, the formula giving the output size is :

$$\boxed{\text{outputsize} = \frac{x-k+2p}{s+1} * \frac{y-k+2p}{s+1}}$$

For a single convolution filter, there are $k^2 * z$ weights to learn. If a fully connected layer were to produce an output of the same size, it would have taken outputsize $*$ inputsize weights to learn, i.e. the output size of the formula above multiplied by $x*y*z$.

(2) Fully connected layers can't apprehend spatial structures, which are the most important information on an image. Besides reducing the computation time by a lot, convolution layers can learn visual features on an input image very easily compared to fully-connected layers. Yet, a limitation of these raw convolution layers is that they record precise positions of the features in the input images. However, we want to have a feature detection at least invariant to its position on the image, or its orientation, etc...

(3) Spatial pooling is mainly used to partially solve the issue raised in the last question. Indeed, spatial pooling helps having a detection invariant to spatial characteristics of the input. In particular, these pooling layers help keeping the important structural elements of our features without the fine details that may not be relevant to our detection. Actually, it helps reducing overfitting through a particular reduction of the dimensions of our data, which is also a technical benefit in order to apply the classification part of the CNN with its fully connected layers after detecting the features.

(4) If we modify the input image of our CNN, we may encounter issues with the

fully connected layers. Indeed, the convolution and pooling layers will compute this image without issues, except the resulting flattened data may not have the same length as expected, meaning the first fully connected layer will be unusable.

(5) Fully connected layers can be seen as particular convolutions. Indeed, a convolution layer using a number of *outputsize* kernels of size the size of the input image is equivalent to a fully connected layer as all the input and outputs units are connected.

(6) If we replace fully-connected layers by their equivalent in convolution, we get rid of the dimension issues in the classification part of the CNN. Indeed, if there is for example a higher than expected 1d vector after the flattening step, the convolution operations would still apply. However, the computed features would be larger at each step leading to an issue at the very end of the CNN as we may obtain an incorrect number of predicted classes.

(7) The size of the receptive field in a convolutional layer is equal to the size of the applied kernel. Thus, in our example, the receptive fields of the two first convolutional layers are respectively equal to $k^2 * 3$ and $k^2 * 64$. At very deep layers, we may not be able to apply kernels anymore as the pooling may have reducing too much the side dimensions of the data, which may be smaller than the required receptive field. Generally speaking, the deeper we observe the detected features in the detection part of the CNN, the more detailed and plentiful they are.

4 Training from scratch of the model

In this section, we will implement a convolutional neural network based on the given layers. We will then apply this model to the $CIFAR-10$ dataset. We first will study its architecture, before implementing with *Pytorch* and testing it.

4.1 Network architecture

(8) To keep the same spatial dimension with a kernel of size 5, we need to use a padding of 2 and a stride of 1. If we compute the formula of the question (1) with these values, we obtain equal dimensions for the input and outputs images.

(9) To reduce the spatial dimension by a factor of two, we can use a null padding and a stride of 2. We can prove it with the formula as well.

(10) Number of weights to learn and output size for each layer :

Conv1 : weights : $3 * 5 * 5 * 32 = 2400$, output size : $32 * 32 * 32$
Pool1 : weights : 0, output size : $16 * 16 * 32$
Conv2 : weights : $32 * 5 * 5 * 64 = 51200$, output size : $16 * 16 * 64$
Pool2 : weights : 0, output size : $8 * 8 * 64$
Conv3 : weights : $64 * 5 * 5 * 64 = 102400$, output size : $8 * 8 * 64$
Pool3 : weights : 0, output size : 4*4*64

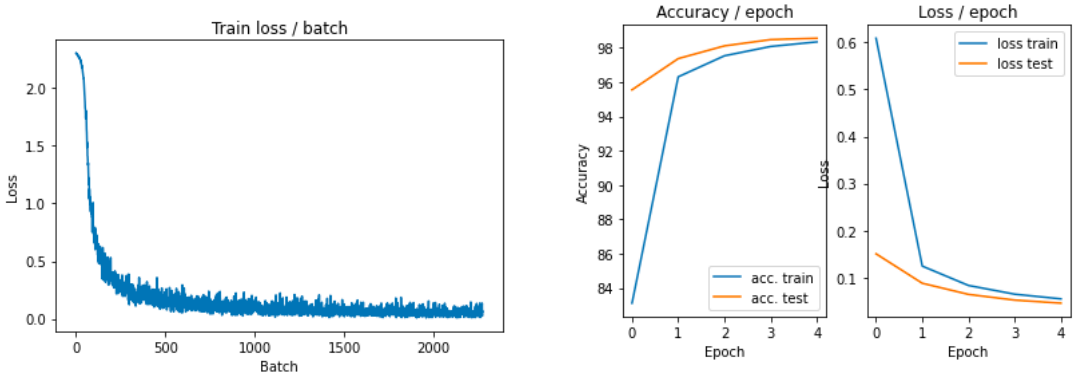
fc4 : weights : $1024 * 1000 = 1024000$, output size : 1000
fc5 : weights : $1000 * 10 = 10000$, output size : 10

(11) The total number of weights to learn is 1190000, in which the vast majority belongs to the first fully connected layer. Besides, the total number of examples is 60000. In comparison, if we substitute the first convolutional layer by a fully connected one, the number of weights to learn is of more than 100 millions.

(12)

4.2 Network learning

(13) We first test the implemented code with a batch size of 128, a learning rate of 0.1 and 5 epochs.



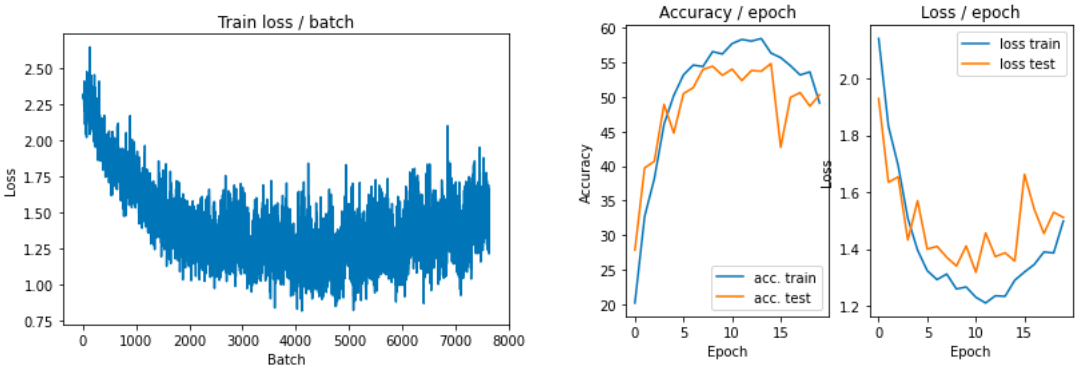
The loss is continuously decreasing and converging to a small value, while the accuracy increases until reaching nearly 100, with few variations, meaning our model is correctly classifying the different pictures of the $MNIST$ dataset.

(14) In test phase, we do not update our model. Indeed we only test the obtained model from the train phase meaning that we do not compute the backward operations used to update the parameters through the optimizer. Consequently, the test loss and accuracy are "one step ahead" than the train ones at each batch step, as the model is updated between these two computations.

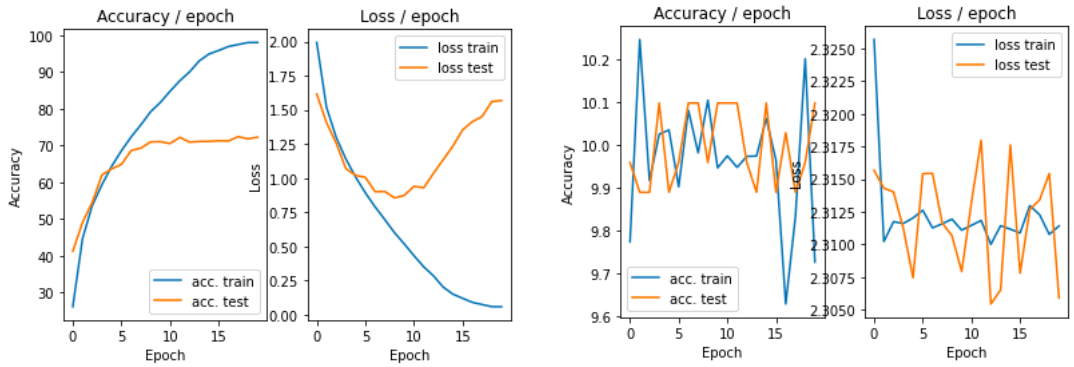
(15) We modify our NN class by implementing the requested architecture with the following code.

```python
class ConvNet(nn.Module):
    """
    This class defines the structure of the neural network
    """

    def __init__(self):
        super(ConvNet, self).__init__()
        # We first define the convolution and pooling layers as a features extra
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, (5, 5), stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2, padding=0),
            nn.Conv2d(32, 64, (5, 5), stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2, padding=0),
            nn.Conv2d(64, 64, (5, 5), stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d((2, 2), stride=2, padding=0, ceil_mode=True),
        )
        # We then define fully connected layers as a classifier
        self.classifier = nn.Sequential(
            nn.Linear(1024, 1000),
            nn.ReLU(),
            nn.Linear(1000, 10),
            # Reminder: The softmax is included in the loss, do not put it here
        )
```

We then test our CNN on the $CIFAR10$ dataset with the same batch size (128) and learning rate (0.1) as before, but this time training on 20 epochs.



(16) The learning rate controls how much to change the model in response to the estimated error each time the model weights are updated. If the learning rate is too small, our model may take too much time to converge or get stuck whereas a learning rate too large may cause our model to have difficulties converging to optimal values. This can be seen on the figures below, the first one being made with a learning rate of 0.01 and the second one with a learning rate of 0.5.



Batch size controls the accuracy of the estimate of the error gradient during the training phase, there is a trade-off between batch size and the speed and stability of the learning process.

(17) The train loss at the start of first epoch can be interpreted at a random classification as the model hasn't been trained on the data yet. Besides, the first test loss is computed after the backward computation on the first batch meaning the weights have been update once based on the obtained loss. The difference between these two losses represents the improvement of our model at each step. The improvement per batch being very slight implies the difference between the losses is too, being around 2.3 at the beginning of training.

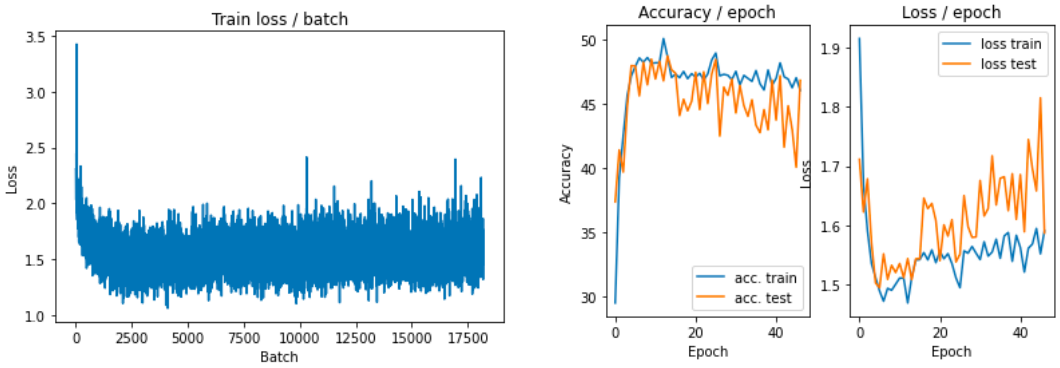
(18) On the first two figures of the question (15), we can see the loss and accuracy curves reaching an extremum before continuously decreasing in performance. An issue would be that our learning rate is too large which causes our trajectory to get away from the minimum of the loss function after several epochs. Besides, our model tend to have underfitting, overfitting and variance issues.

5 Result improvements

In this section we will study several typical techniques to improve the performance of our model and try to solve the issues our raw model is facing.

5.1 Standardization of examples

(19) We add a normalization in the data pre-processing thanks to the *Normalize*() *torchvision* function and the given mean values. Below are the loss and accuracy curves we then obtain. In comparison with the question (15) graphs, the curves reach their extremum earlier, meaning the algorithm converges more quickly. However, our model still presents poor performances and its other issues. This normalization may not be the most suitable to our problem.

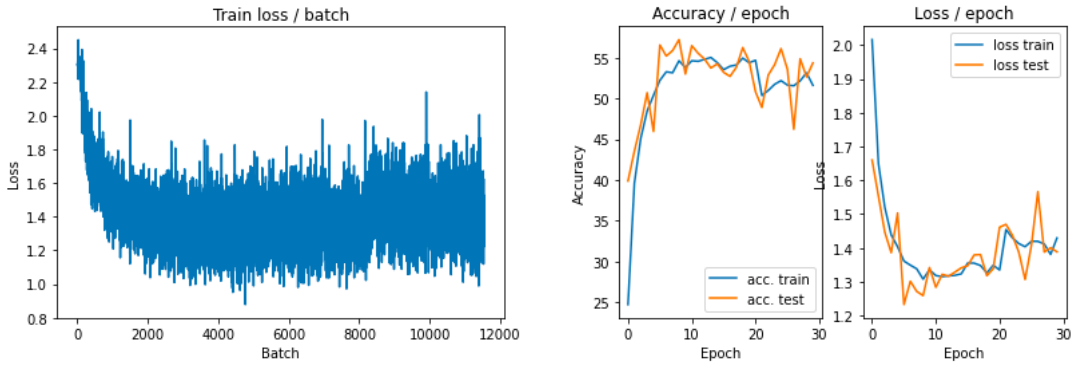


(20) We only calculate the average image on the training example to avoid any data leakage during the model evaluation process. Indeed, if we take into account the characteristics of the validation set pictures, we will train our model with information apart from our training set, which leads to overfitting.

(21) Other common normalization schemes are whitening transformation, also called sphering transformation as it transforms the covariance matrices of our data to identidy ones, "sphering" the data. It helps making the computations during the gradient descent process easier.

5.2 Training data increase

(22) We add to the dataset the required transformations thanks to the *RandomCrop*() and *RandomHorizontalFlip*() *torchvision* functions and plot the usual graphs.



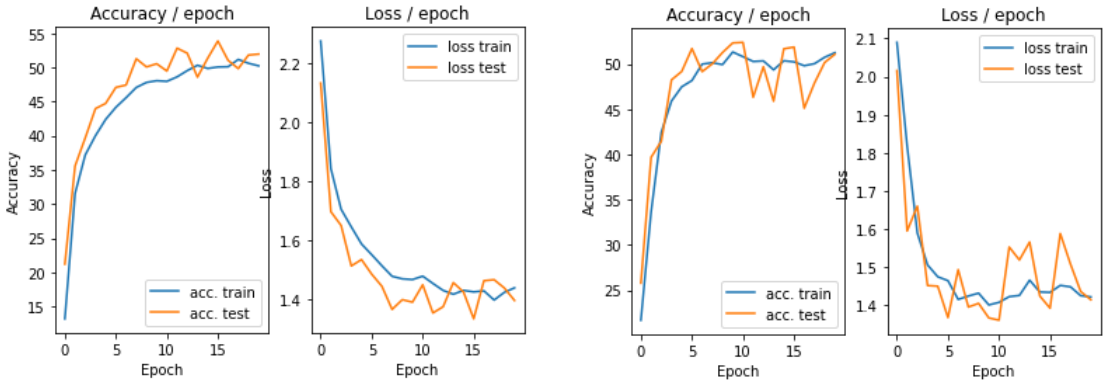
This technique seems to reduce a lot our model overfitting as the train and loss curves are very similar. However, it didn't help to solve the other issues at all and the global performance are still poor.

(23) The symmetrical approach in data augmentation may not be relevant for any input images. Indeed, if our images contains unsymmetrical information, this can lead to high levels of miss classification. An example would be to apply a vertical flip to a M letter as it might then be nearly impossible for our model to distinguish M to W letters. This approach is really helpful to create an orientation invariance, but only if it is relevant to our data. Vehicles and animals will keep the same nature whether we apply a symmetry or not, that's why it is effective. However, horizontal symmetry

seems to be way more relevant than the vertical one as we rarely encounter pictures of flipped over animals or vehicles. We can train our model to apprehend these situations but it will lead to weaker global performances.

(24) The main limitation of data augmentation arises from the data bias, indeed the augmented data distribution can be quite different from the original one. We then must try not to accentuate specific features because of behavioural tendencies when manually augmenting the data, otherwise we may distinguish too much the features distributions present in training and evaluation sets, leading to poor performances of our model.

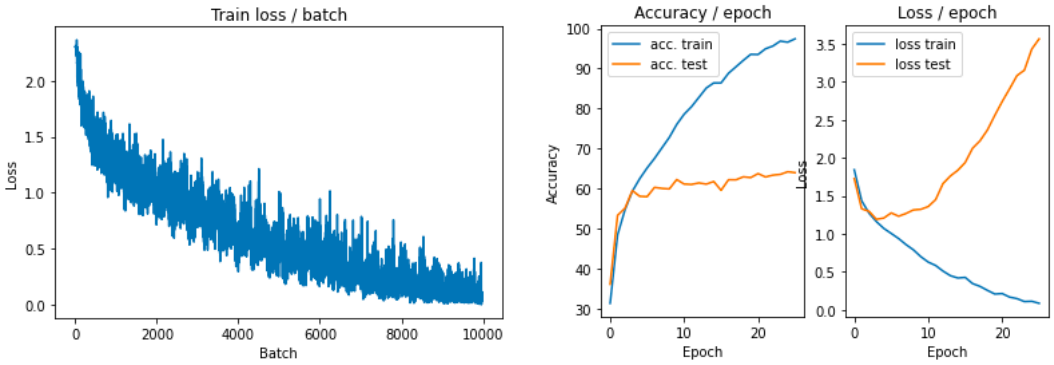
(25) Many data augmentations methods exist and some are more useful to specific problems. In our case, the dataset is composed of small photos of object and animals, meaning that we should avoid distorting the images too much. We could test for example, small translations (around 3 pixels) and shear, which distorts the image along an axis to help the model learning with vision angle changes. We tested these two data augmentation techniques and we show the resulting loss and accuracy graphs below.



We can see that even separately, these techniques reduce overfitting a lot. Indeed the loss and train curves match quite well until the 20th epoch at least. However we must be aware that some combinations may not be as effective as the others, it usually interesting to test multiple combinations given several techniques to find the optimal one.

5.3 Variants on the optimization algorithm

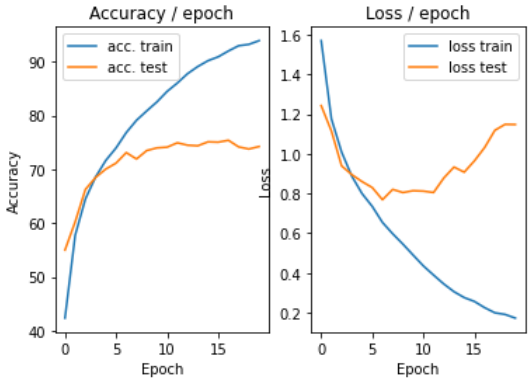
(26) We implement a learning rate scheduler with an exponential decay and a momentum parameter using the values of the problem statement.

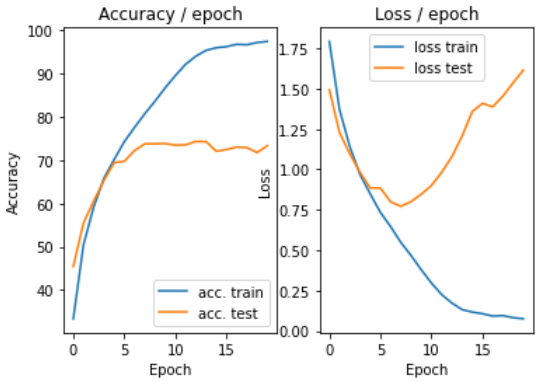


On these graphs, the continuous decrease of the train loss function shows that our learning rate scheduler helps countering the converging issue we raises in question (15), showing that decreasing the learning rate through process time is suitable for our network. However the model is still quickly overfitting a lot as the differences between the train and test loss curves are growing very rapidly through the epochs. Yet, the performances concerning classification are slightly better as we reach a 60 percent rate and the variance is also reduced.

(27) Using momentum in gradient descent allows to overcome the oscillations of noisy gradients and to easily get through flat spots or pathological curvatures such as ravines of the search space, by taking into account the inertia of our trajectory. It accelerates the learning process. The learning rate schedule causes the learning rate to decrease with each epoch iteration, allowing large weight changes in the beginning of the learning process and small changes or fine-tuning towards the end. It also helps countering the gradient estimator error due to random sampling of examples, that may cause our model to have difficulties converging at the end of the learning process.

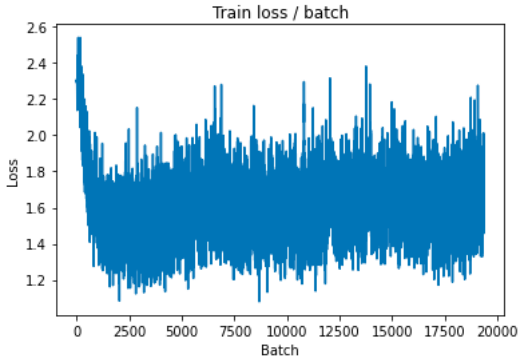
(28) A very efficient and used optimizer along CNNs is the Adam optimizer, it basically maintains a learning rate for each network weight and separately adapts them as learning unfolds. Other recent optimizers present even higher performances. For example, the RAdam, Adabelief and AdaBound optimizers which are slightly improved versions of the Adam optimizer, the LookAhead optimizer which intuitively chooses a search direction by looking ahead at the sequence of fast weights generated by another, or the Adagrad optimizer which uses the SGD algorithm with an optimized learning rate being fixed by the algorithm. We plotted the usual graphs with the Adam optimizer on the left and RAdam optimizer on the right. The performances of these optimizers are very good and better than the configured SGD, especially RAdam.

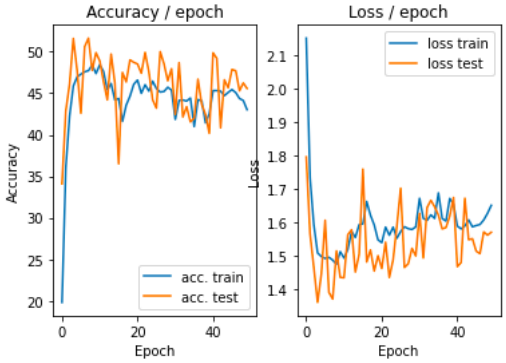




5.4 Regularization of the network by dropout

(29) We add a dropout layer between the two fully-connect ones with a dropout rate of 0.5 which usually is optimal according to the *Srivastacaetal.* article.





As shown by the train loss and accuracy curves, the performances remain quite poor, our model seems to underfit. However, there is a huge reduction in overfitting. Thus, combining this improvement with others helping on reducing underfitting seem to be a solution to reach higher global performances.

(30) According to the *DeepLearning* book, "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error".

(31) Dropout randomly selects a subset of features in each layer based on the ratio value we implement during the training process. Just like in random forest algorithm, we select a subset of samples randomly to avoid overfitting over some particular features. Dropout reduces the complex co-adaptation of the different hidden units on the training data and make it more robust. Indeed, it has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. Randomly dropping out nodes in the network is a simple and effective regularization method.

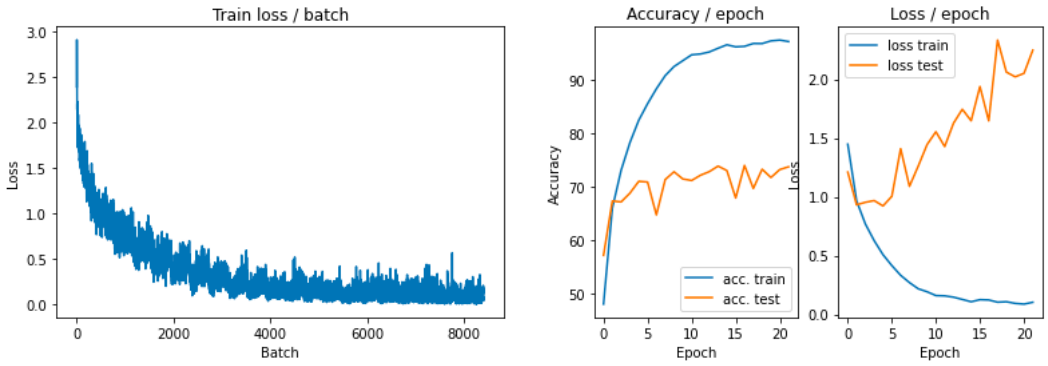
(32) The default interpretation of the dropout hyperparameter is the probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no

outputs from the layer. In the *Srivastacaetal*. article, small values tend to underfit while too large ones tend to overfit, the optimal and robust value seems to be 0.5 and more generally in a 0.4 to 0.6 interval, depending of other hyperparametres of the NN.

(33) In training phase, the dropout layer will randomly deactivate a part of the neurons, and so the weighted links between a layer to a a following one. However, at test time, we want to average the results of the different neural networks. It's computationally infeasible to manually apply and average each neural network for each test. However, a very simple approximate averaging is to use a single neural net at test time without dropout, with weights being scaled-down versions of the trained ones. To ensure that for any hidden unit, the expected output is the same as the actual output at test time, the outgoing weights of an unit are multiplied by p at test time if it was retained with probability p during training.

5.5 Use of batch normalization

(34) We now add a 2D batch normalization layer after each convolution layer.

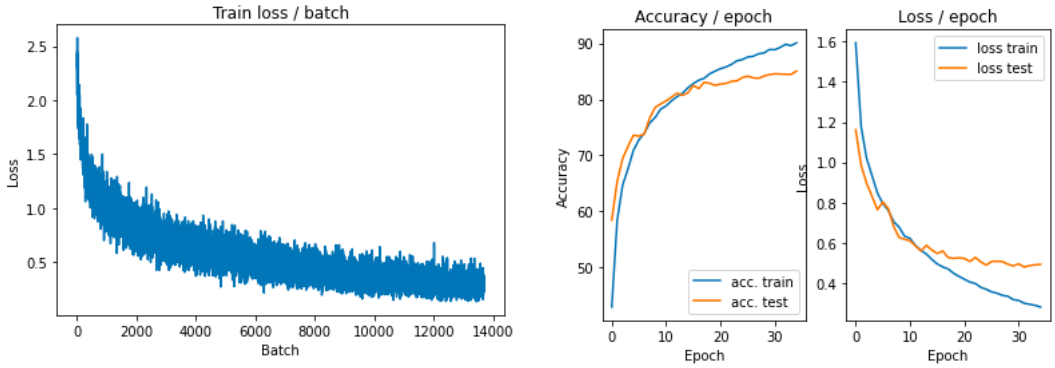


Batch normalization is the modification that helped reaching the highest accuracy among the five tested ones, meaning our model is underfitting way less. Besides, it also seems to help reducing the issue of the train loss convergence.

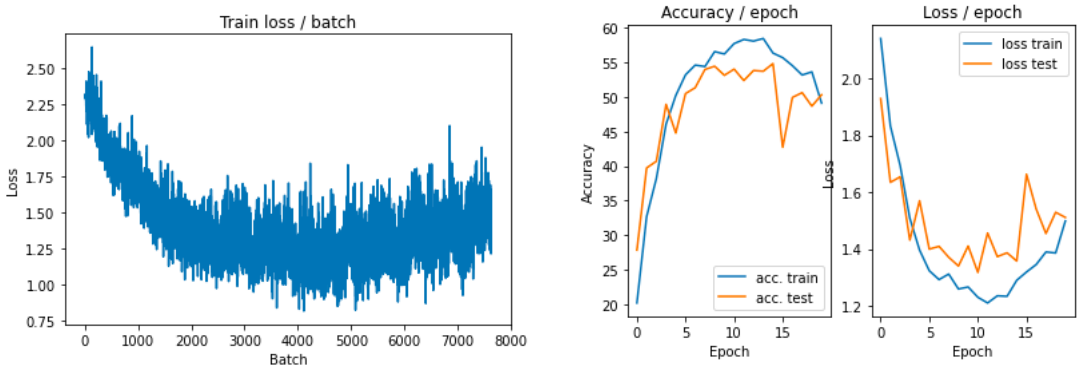
5.6 Improvements summary

In the last sections, we separately tested a few possible improvements for our CNN model and we saw the specific benefits of each one of them. In question (15), the model encountered four main issues : the train loss and accuracy curves didn't converge, our train extremums were not satisfying, meaning our model was underfitting and we could observe a slight overfitting after several epochs. In addition, our model seemed to have a variance issue. These are the conclusions drawn from this section studies.
In subsection 5.1, we saw that the standardization of examples helped our model to converge faster towards the loss minimum. In addition, the late increase of the train loss function is reduced but not suppressed. The normalization of the data forces the shared weights of the network to have similar calibrations for different features, which makes the cost function to converge more rapidly and effectively.
In subsection 5.2, we saw that data augmentation helped reducing overfitting a lot as

the train and test curves are very close. Wisely augmenting data to train our model can counter the issue of our model trying to learn by heart its training set, which is accentuated when using small train sets.
In subsection 5.3, we saw that implementing a learning rate scheduler and a momentum factor in our optimizer helped solving the non convergence of our train curves, while slightly improving the classification performances and reducing the variance of our model. The benefits of the momentum being to overcome complicated structures of the loss spatial representation, and the learning rate scheduler allowing our model to refine its trajectory at the end of the process.
In subsection 5.4, we saw that regularizing our network with a dropout layer also helped reducing overfitting by limiting the co-adaptation of the different layers units on the training data.
In subsection 5.5, we saw that batch normalization helped reaching higher classification performances by reducing underfitting, and also solving the non convergence issue of our model. Batch normalization normalizes the contributions to a layer for every mini-batch. This has the impact of settling the learning process and drastically decreasing the number of required training epochs.
Each one of these modifications has its own benefits to our model. By combining them, we may obtain a very robust model. Here are the usual curves when applying all of them at the same time.



And as a reminder here are the curves of our raw model of the question (15).



These modifications improved our model by a lot, helping to reach a 0.85 accuracy rather than 0.55. The model now converges easily, has very few variance, while maintaining a very low overfitting during the first 10 epochs. Besides, the performances of

the train loss function are way better, meaning our model doesn't underfit as much as before.