RDFIA - TME 2
Intro to Neural Networks & Convolutional Neural Networks

Lucrezia Tosato & Marie Diez

Contents

1 Intro to Neural Networks

1.1 Section 1 – Theoretical foundation

1.1.1 Supervised dataset

1. The training set is used to train our model. This set should not contain any elements from the validation set or the test set. The validation set is used to choose the best hyper-parameters on our model, we choose the hyper-parameters that give us the best score with the validation set. The test set is used to test our model and predict the scores obtained if we apply our model on an unknown data set, this way we can calculate the performance of our model without bias.

2. The influence of the number of example N is that the bigger the N the more representative of the real world is our training set, this way the bias has more possibilities to be little, and the model robust. We aim for a better generalization and increasing N we can try to achieve this goal, bu we still could have a lot of bad examples in the set, even if it's huge.

1.1.2 Network architecture (forward)

3. It is important to add activation functions between linear transformations to introduce non linearity in the network. A linear equation is simple to solve but is limited in its capacity to solve complex problems. A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.
"'' we can look at that as a threshold,a function that lights up the parts that are responsible of the loss. "''

4. nx is the size of the input data (number of dimension). nh is the number of neurons in the hidden layer. ny is the size of the predictions. We do not choose the size of the input data. Instead, the number of hidden layer neurons is a hyper-parameter to be chosen and we choose the size of the predictions according to the problem to be solved. In this example the sizes of nx, nh, ny in the figure 1 are respectively:

- nx=2
- nh=4
- ny=2

5. The vectors ŷ and y represent respectively the prediction and the ground true. The difference between these two quantities is that one is the output of our network and the real one is used for supervision and correction. We want them to be the same on the validation and of course in test. If f is our model, x our input data and y the label ŷ.

6. We use the SoftMax function in the output to obtain a probability distribution over the different classes of the prediction. This allows us to deduce the most likely class. Softmax is used as the activation function for multi-class classification problems where class membership is required on more than two class labels.

7. Mathematical equations allowing to perform the forward pass of the neural network:

$$\tilde{h} = W_h \dot{X} + B_h \tag{1}$$

$$h = tanh\tilde{h} \tag{2}$$

$$\tilde{y} = W_y \dot{h} + B_y \tag{3}$$

$$\hat{y} = SoftMax(\tilde{y}) \tag{4}$$

1.1.3 Loss function

8. For cross entropy and squared error to decrease the loss L, ŷ must tend to y.

9. Cross-entropy is preferred for classification, while mean squared error is one of the best choices for regression. This comes directly from the statement of the problems itself - in classification you work with very particular set of possible output values thus MSE is badly defined (as it does not have this kind of knowledge thus penalizes errors in incompatible way).

1.1.4 Optimization algorithm

10. The most reasonable variant of gradient descent among all the variants seems to be the mini batch gradient descent: we consider a group of image, it doesn't require too much time to compute, but still it is a good approximation of the overall data set.

11. The learning rate on learning influence the size of the step between two different iterations. If the learning rate is too large, the gradient descent will not converge due to too many jumps. If the learning rate is too small, the gradient descent will be long.

12. The back propagation algorithm will be less costly in terms of complexity than the naive approach because the back propagation algorithm only relies on the elements of the previous layer to calculate the gradients whereas the naive approach will calculate all the gradients of all the layers to obtain the desired gradient.

13. The criteria that the network architecture must meet to allow such an optimization procedure is that the set of layers and activation functions must be linear.

14. Since y_i is in one-hot encoding so only one component of this vector is equal to 1, so:
$\sum_i y_k log(\sum_j e^{\tilde{y_j}}) = log(\sum_j e^{\tilde{y_j}})$
The loss can be simplified by:

$$loss(y, SoftMax(\tilde{y})) = -\sum_i y_i log(\frac{e^{\tilde{y_j}}}{\sum_j e^{\tilde{y_j}}}) = \tag{5}$$

$$= -\sum_i (y_i \tilde{y_i} - \tilde{y_i} log(\sum_j e^{\tilde{y_j}}) = \tag{6}$$

$$= -\sum_i y_i \tilde{y_i} + \sum_k y_k \log(\sum_j e^{\tilde{y_j}}) = \tag{7}$$

$$= -\sum_i y_i \tilde{y_i} + log(\sum_j e^{\tilde{y_j}}) \tag{8}$$

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

$$\frac{\partial l}{\partial \tilde{y_i}} = -y_i + \frac{e^{\tilde{y_i}}}{\sum_j e^{\tilde{y_j}}} \tag{9}$$

16. Since we have $\frac{\partial \tilde{y_k}}{\partial W_{y,ij}} = h_k$, the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} l$ is:

$$\sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{\partial W_{y,ij}} = \sum_k (-y_k + \frac{e^{\tilde{y_k}}}{\sum_j e^{\tilde{y_j}}}) h_k \tag{10}$$

17. The othe gradients : $\nabla_{\tilde{h}} l$, $\nabla_{W_h} l$, $\nabla_{b_h} l$:

$$\frac{\partial L}{\partial \tilde{h_i}} = \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial \tilde{h_i}} = \tag{11}$$

$$= W_{y,i}(-y_i + \frac{e^{\tilde{y_i}}}{\sum_j e^{\tilde{y_j}}})(1 - tan^2(\tilde{h_i})) \tag{12}$$

$$\frac{\partial L}{\partial W_{h,i}} = \frac{\partial L}{\partial \tilde{h_i}} \frac{\partial \tilde{h_i}}{\partial W_{h,i}} = \tag{13}$$

$$= W_{y,i}(-y_i + \frac{e^{\tilde{y_i}}}{\sum_j e^{\tilde{y_j}}})(1 - tan^2(\tilde{h_i}))X_i \tag{14}$$

$$\frac{\partial L}{\partial b_{h,i}} = \frac{\partial L}{\partial \tilde{h_i}} \frac{\partial \tilde{h_i}}{\partial b_{h,i}} = \tag{15}$$

$$= W_{y,i}(-y_i + \frac{e^{\tilde{y_i}}}{\sum_j e^{\tilde{y_j}}})(1 - tan^2(\tilde{h_i})) \tag{16}$$

1.2 Section 2 – Implementation

1.2.1 Forward and backward manuals

We use the function written bellow to initialize our parameters :

```
torch.normal(0, 0.3, size=(size_1,size_2))
```

Here are some of the results :

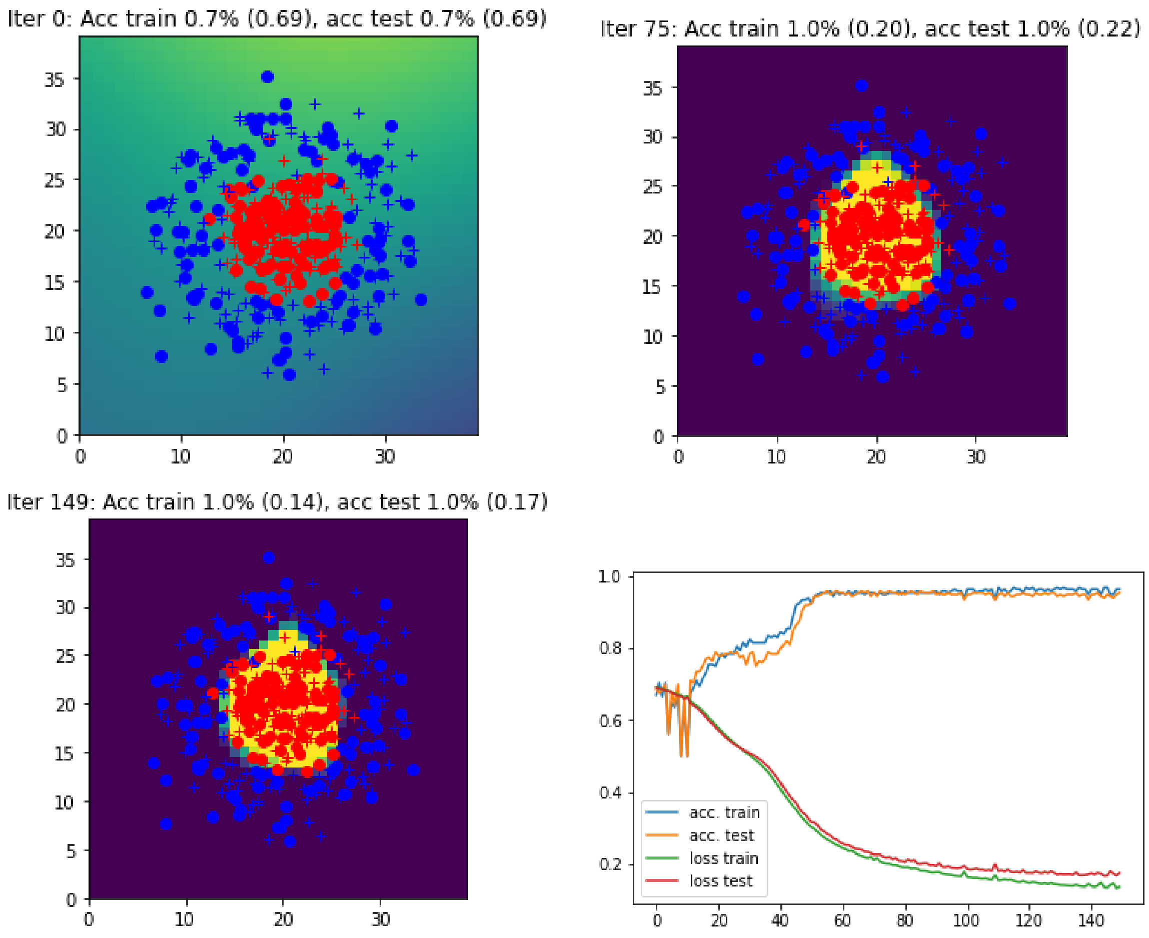


Figure 1: Illustration of parameters at different iteration and results

We can see that the points are well classified and that the error in train and in test decreases (so the accuracy increases) until reaching a low error of about 0.15% (with CrossEntropy loss). We can see that the loss is decreasing in the train curve that normal, but the important point is that the loss is also decreasing in the test curve, that mean's that we are not overfitting or underfitting our model.

1.2.2 Simplification of the backward pass with torch.autograd

We can simplify our functions with torch.autograd that enable to compute the backward pass.

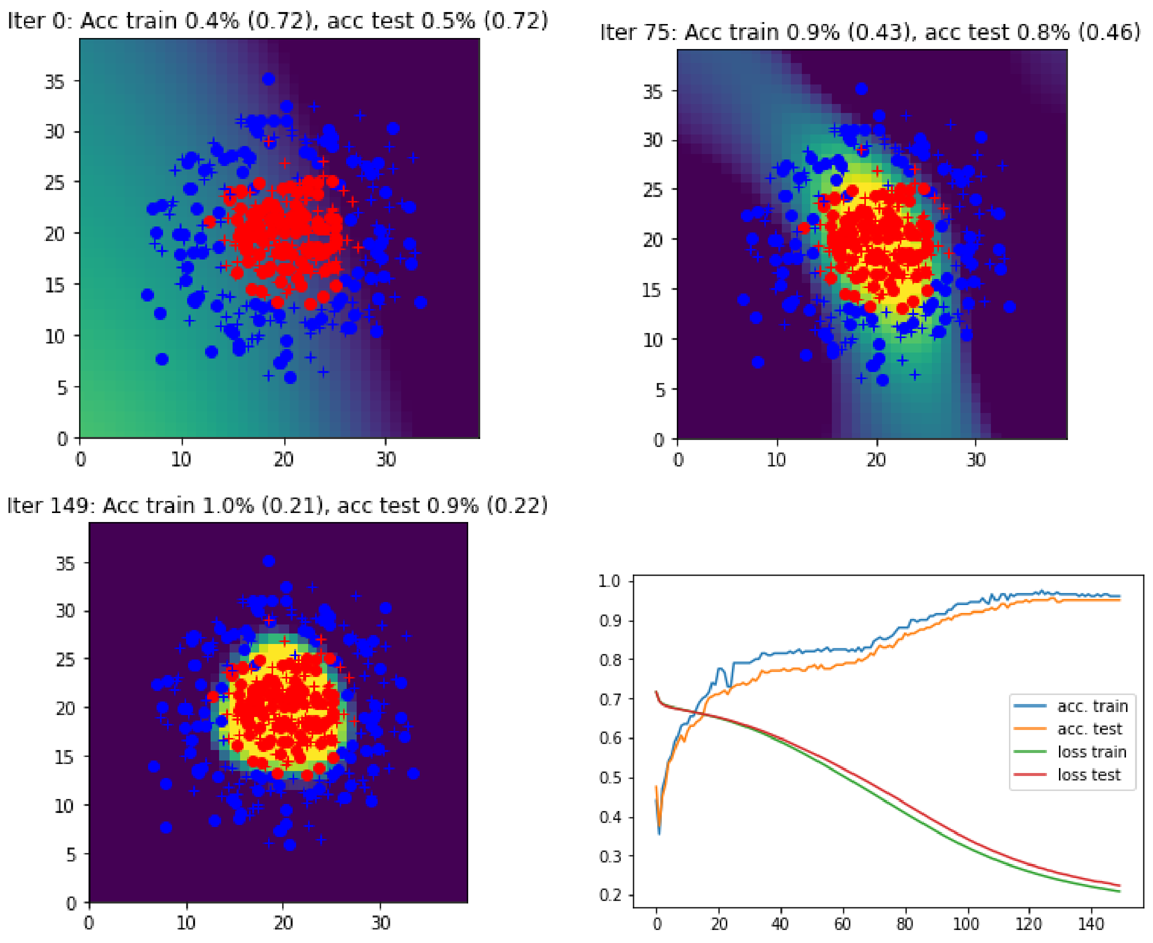


Figure 2: Illustration of parameters at different iteration and results

Parameters seem to evolve slightly differently during the minimisation of the gradients, however the final result is very similar, we can make the same comments on the performance curves.

1.2.3 Simplification of the forward pass with torch.nn layers

We can again simplify our functions by using torch.nn layers, this would enable to create a torch model for our network, we will not need anymore ou forwar function.

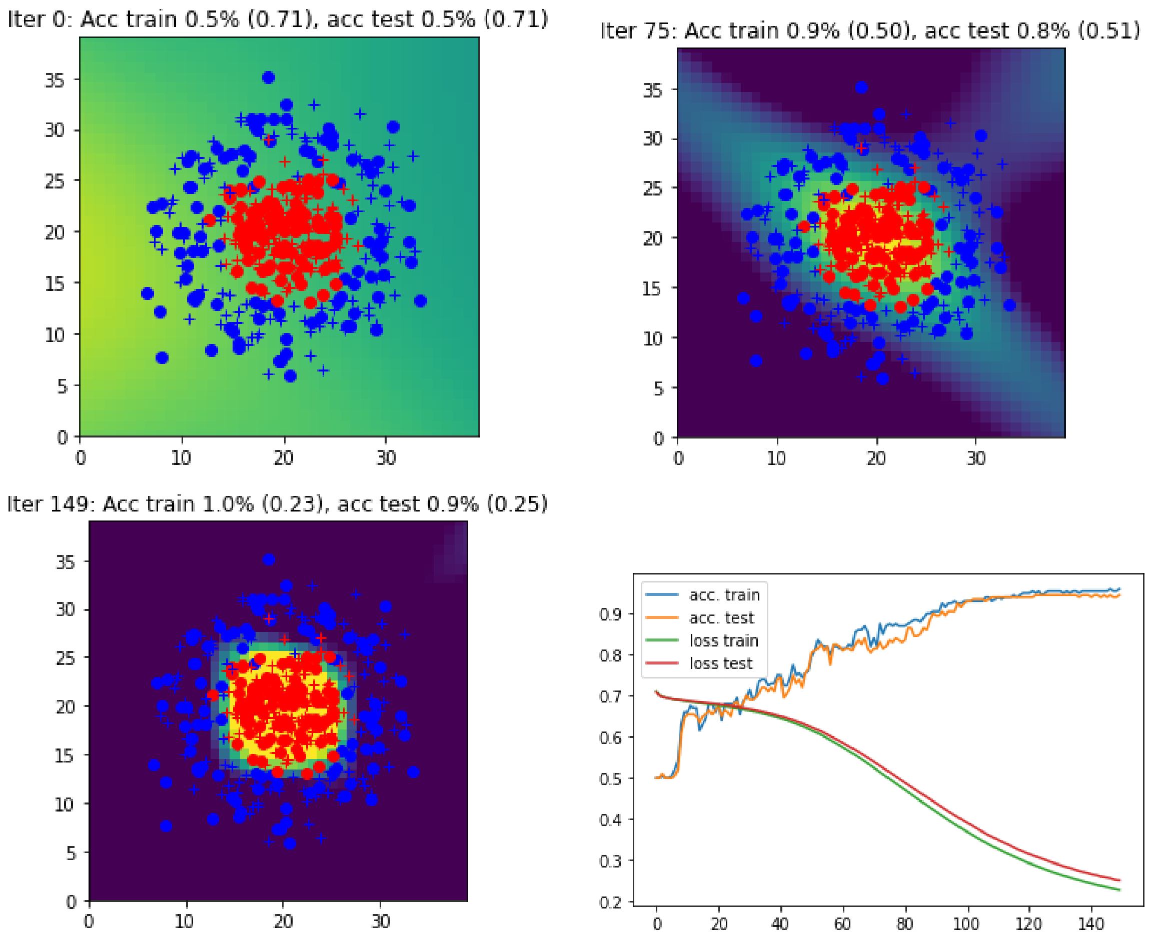


Figure 3: Illustration of parameters at differents iteration and results

We can make the same comments a before, parameters seems to evolved slightly differently but we have similar results.

1.2.4 Simplification of the SGD with torch.optim

We can even more simplify our function by using torch.optim.

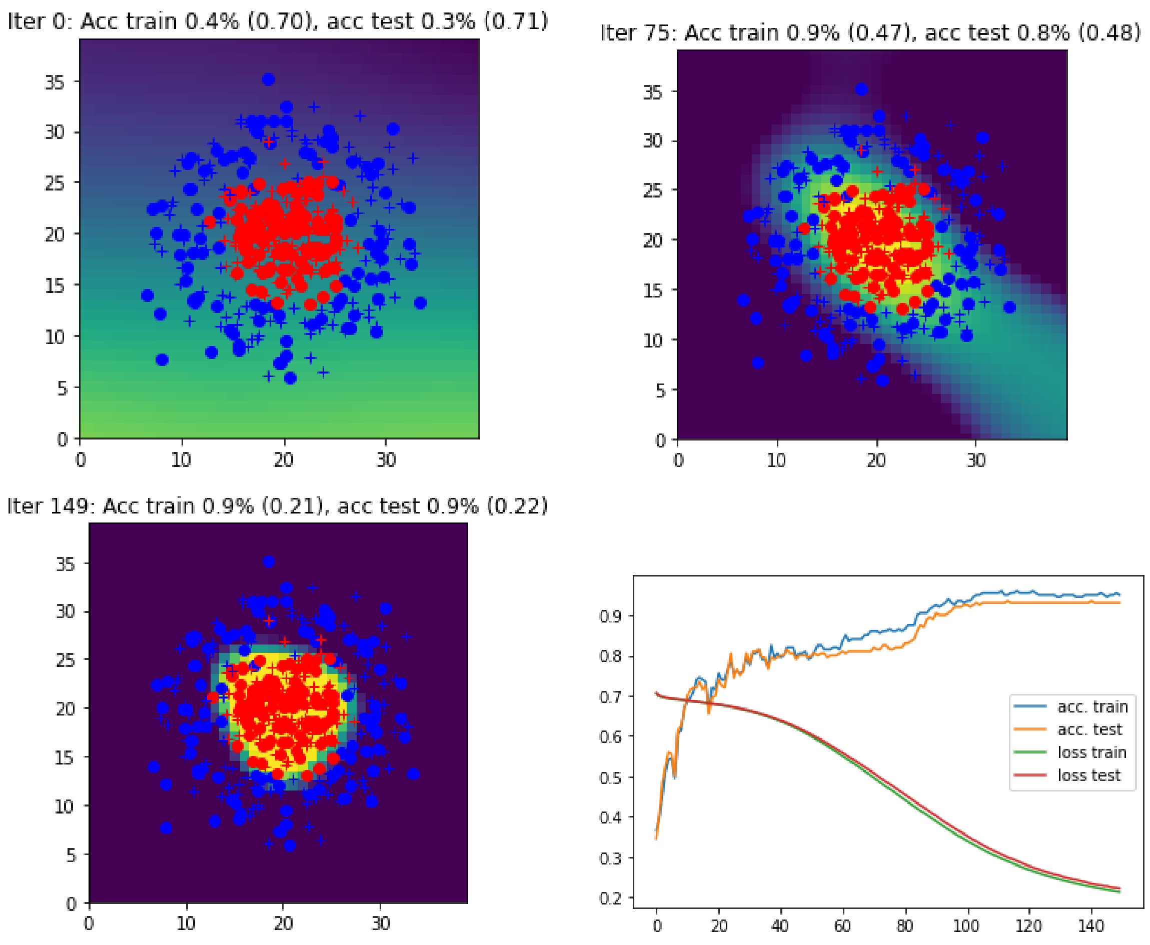


Figure 4: Illustration of parameters at differents iteration and results

We can make the same comments a before, parameters seems to evolved slightly differently but we have similar results.

1.2.5 MNIST application

We can apply our code to the MNIST dataset.

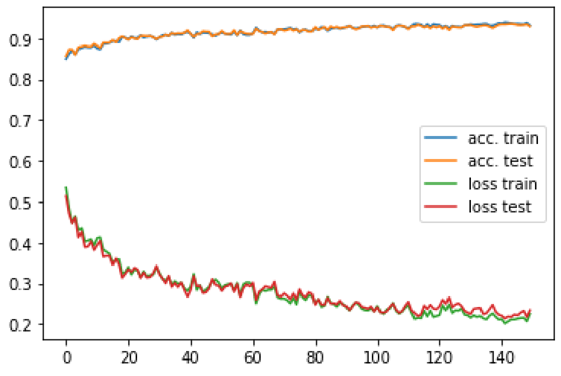
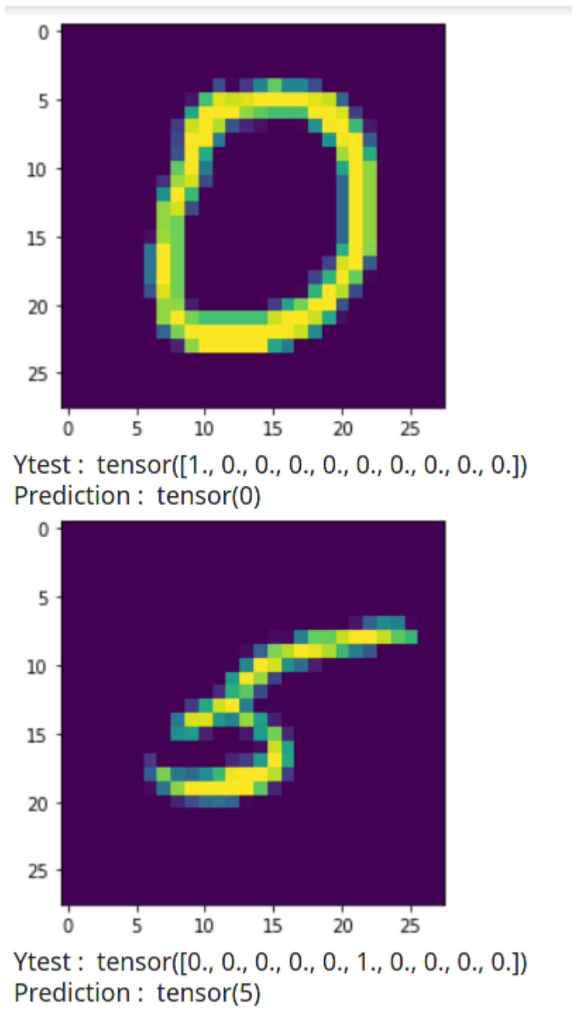


Figure 5: Curves results

We can test our model with illustration :



```
Ytest : tensor([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
Prediction : tensor(0)
```

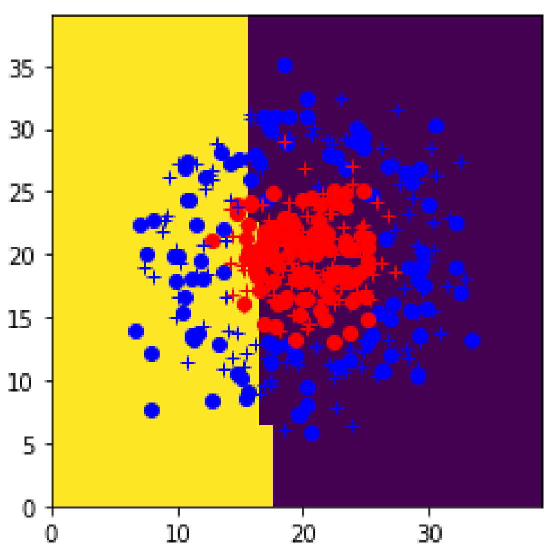
```
Ytest : tensor([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
Prediction : tensor(5)
```

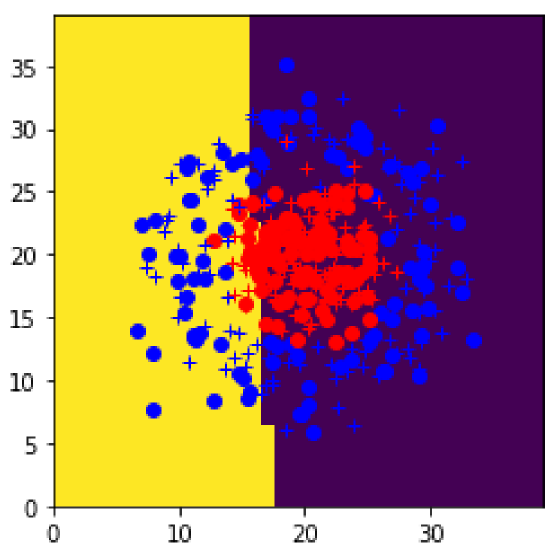
Figure 6: Curves results

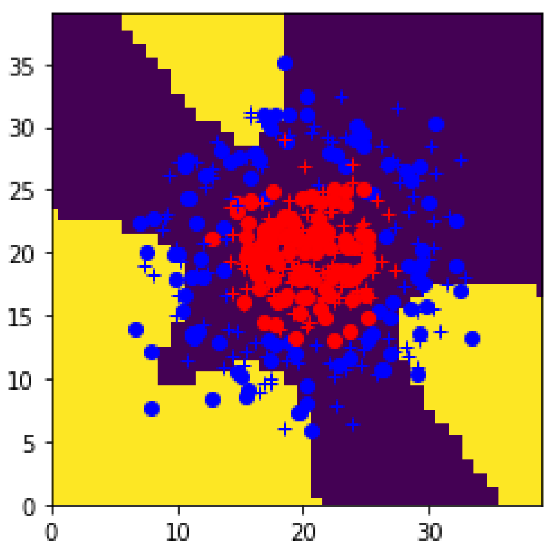
We can see that our model predict correctly this 2 examples.

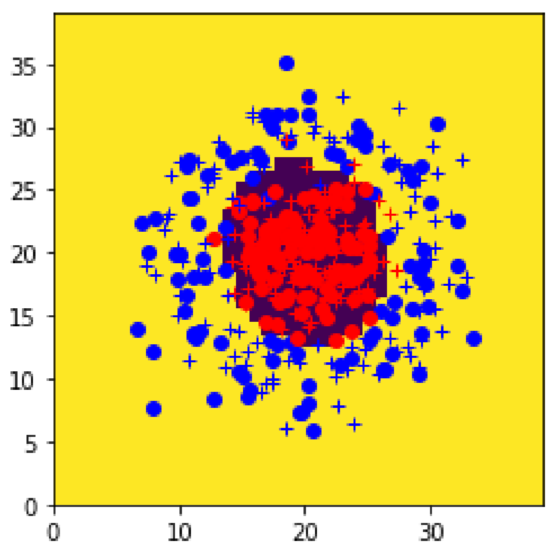
1.2.6 Bonus: SVM

We can test svm classification on the circle dataset.


(a) C=20, kernel=linear


(b) C=20, kernel=poly, deg=1


(c) C=20, kernel=poly, deg=5


(d) C=20, kernel=poly, deg=6

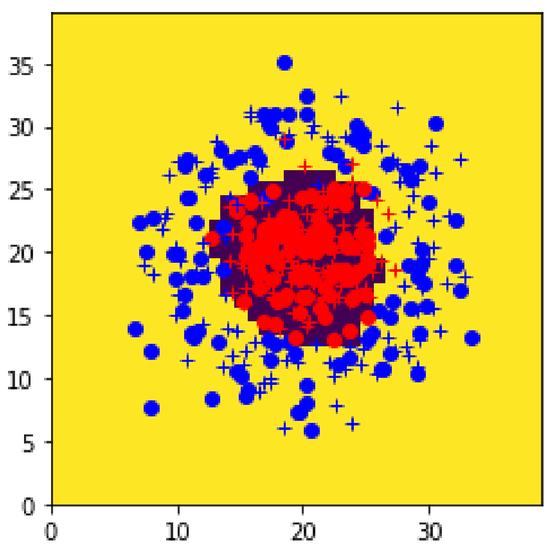

(e) C=20, kernel=rbf

Figure 7: Classification results and circle dataset with differents SVM hyperparameters

We can see that we need a non-linear boundary so the linear and poly kernels for degree = 1 do not work. We can use the poly kernel by increasing the degree, from degree 6 the kernel is usable. One could also increase the dimensionality of the data to make the linear kernels work. (test for various other hyperparameters)

2 Convolutional Neural Networks

2.1 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 × y × z what the output size of the image will be $x' * y' * C$:

$$x' = \lfloor \frac{x - k + 2p}{s} \rfloor + 1$$

$$y' = \lfloor \frac{y - k + 2p}{s} \rfloor + 1$$

$$z' = C$$

There is $(k * k * z + 1) * C$ parameters for a single convolution (C=1) with the biais.
To produce an output of the same size with FC : $x * y * z * x' * y' * C$ (C=1).
The number of parameter without the convolution is a lot more bigger.

2. The strength of convolutional layers over fully connectes layers is that have lot less parameters and they enable to keep the spatial information in images. They are robust to small shift thanks to the max pooling operator. The strength of convolutional layers over fully connected layers is precisely that they represent a narrower range of features than fully-connected layers. A neuron in a fully connected layer is connected to every neuron in the preceding layer, and so can change if any of the neurons from the preceding layer changes. A neuron in a convolutional layer, however, is only connected to "nearby" neurons from the preceding layer within the width of the convolutional kernel. As a result, the neurons from a convolutional layer can represent a narrower range of features in the sense that the activation of any one neuron is insensitive to the activations of most of the neurons from the previous layer.

3. Spatial pooling is a strategy for creating "spatial invariance across small lateral shift" in visual object recognition. Spatial pooling in a hierarchical feature model makes each feature resilient to minor shifts in position, since multiple positions are pooled together. If this technique is applied repeatedly and hierarchically, then an object can be recognized despite substantial spatial distortions, making object recognition more robust and less fragile. Moreover this enable to reduce the number of parameters.

4. We can compute the whole convolutional part, but when we will apply FC layers, the size will not match. The network was trained on specific size of images so the FC layers has a specific input size (see schema below) make him enable to treats differences input size which is the case when the input size is larger.
In this example the FC layer will be 4096 filters of size 7x7x512, the input size depend on the image size.

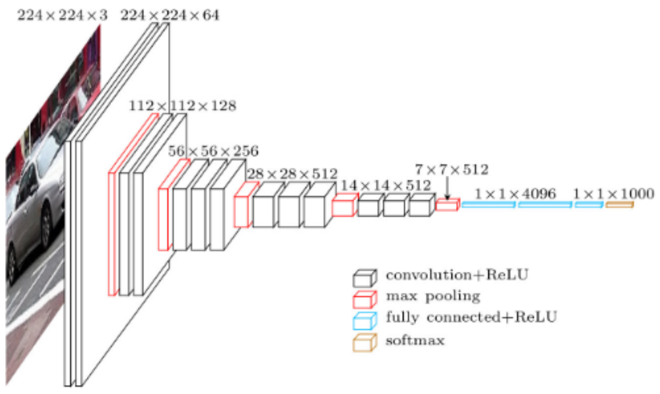


Figure 8: Example Neural Net

5. We can analyze fully-connected layers as particular convolutions considering a kernel of the same size of the input image size, and the number of convolution filters correspond to the dimensions $n_x^{'} \times n_y^{'}$ of the output.

6. If we replace the fully-connected part by its equivalent in convolution the problem will be solve. We can use differents approaches seen in course, like image based global pooling (GPA or max) or region based method like DeepMIL and Weldon that use convolution instead of FC.

7. The receptive field sizes of the first convolution layer are k1×k1 where k1 is the kernel size of the first convolution. These sizes can vary depending on the padding and the neuron considered (on a border or in a corner). If the considered neuron takes into account padding values, then the size of the receptive field is smaller. We thus obtain the following equations:

$$n' = \frac{n - k + 2p}{s} + 1$$

$$j' = j * s$$

$$r' = r + (k - 1) * j$$

- The first equation calculates the number of output elements as a function of the number of input elements, the padding, the stride and the kernel size.
- The second equation calculates the jump, which is the number of elements that are jumped as a function of the stride and the previous stride (and therefore the jump of the previous layers), it's represents the cumulative stride.
- Finally, the third equation calculates the size of the receptive field as a function of the size of the receptive field of the previous layer. Thus, we notice that if k $>$ 1 and j $>$ 0 then the receptive field size is increasing. Therefore, the deeper the layer, the greater the number of pixels considered.

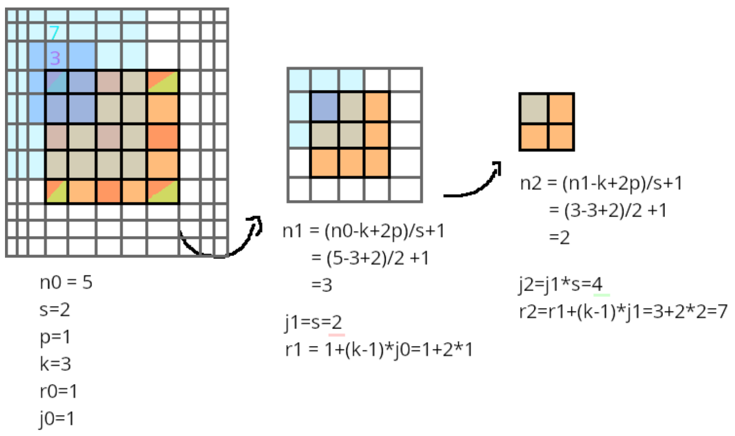


Figure 9: Schema

2.2 Training from scratch of the mode

8. Since the output size equations is :

$$n' = \frac{n-k+2p}{s} + 1$$

so if we want $n' = n$:

$$s = \frac{n-k+2p}{n-1}$$

$$p = \frac{sn-s-n+k}{2}$$

Generally, to keep the dimension equal, use a stride of 1 a padding of $\frac{K-1}{2}$ with K dimension of the kernel.

9. Using the same logic as in the question before, we want $n' = \frac{n}{2}$, so:

$$s = 2\frac{n+2p-k}{n-2}$$

$$p = \frac{sn-2s-2n+2k}{4}$$

Generally, to reduce the dimension by two, we do a max pooling with a kernel of size 2, a stride of 2 and no padding.

10.

- For the first layer, we have 32 convolutions with a kernel size equal to 5 and an image of size $32 \times 32 \times 3$. Thus the number of weights to learn for this layer is : $(5^2 \times 3 + 1) \times 32 = 2432$ and an output size $16 \times 16 \times 32$ with max pooling reducing the height and width of our output by 2.
- For the second layer, For the second layer, we have 64 convolutions with a kernel size equal to 5 and an output of the previous layer of size $16 \times 16 \times 32$. Thus the number of weights to learn for this layer is : $(5^2 \times 32 + 1) \times 64 = 51264$ and an output size that is $8 \times 8 \times 64$ with max pooling reducing the height and width of our output by 2.
- For the third layer, we have 64 convolutions with a kernel size equal to 5 and an output of the previous layer of size $8 \times 8 \times 64$. Thus, the number of weights to be learned for this layer is: $(5^2 \times 64 + 1) \times 64 = 102464$ and an output size that is $4 \times 4 \times 64$ with max pooling reducing the height and width of our output by 2.
- For the fourth layer, we have a fully connected with 1000 neurons as output followed by a ReLU. Thus, the number of weights to learn is $4 \times 4 \times 64 \times 1000 = 1024000$. The output is a vector of size 1000 since the ReLU does not change the output size.

- For the fifth layer, we have a fully connected with 10 neurons in output followed by a Softmax. Thus, the number of weights to learn is $1000 \times 10 = 10000$. The output is a vector of size 10 since the Softmax does not change the output size.

11. The total number of weights to learn is therefore $2432 + 51264 + 102464 + 1024000 + 10000 = 1190160$. We can compare this number to the number of examples in learning which includes 50 000 of size $32 \times 32 \times 3$ what makes 51 200 000 pixels coded in RGB.

12. The number of parameters to learn with that of the BoW and SVM approach required respectively:

- BoW: We have to learn the words of our dictionary, so 1000 vector for 10 classes, 10 000 parameters.
- SVM: 10 parameters . The total number of parameter to be computed for the NN.

2.3 Network learning

14. The difference between calculating loss and accuracy in train and in evaluation is that we do not update the parameters of our network with a back-propagation in the case of evaluation.

16. The effects of the learning step and the size of the mini batch is the speed of convergence of our model, so the speed through which our model learns.
If the learning step is:

- too small: there will be almost no learning, the parameters do not change and the accuracy doesn't improve.
- too big: the result can diverge.

There are optimiser such as ADAM for example that will increase the learning step for a more efficient convergence.

If the size of the mini batch is:

- too small: The computation will be fast and take low memory but there will be a big variance when learning, indeed smaller batch sizes are noisy. We will end up in the case of the stochastic gradient descent.
- too big: The computation will be super slow and take a lot of memory.

A compromise must be found between the two.

17. The error of the first epoch corresponds to the error of the network which started with randomly initialized weights and therefore is very high. Our model consequently makes very random predictions at the beginning predictions that it corrects as it goes along.

18. As said in the previous question the first epoch has very bad performance in accuracy/loss, but this performance improves with iterations as we can see in the last epoch.
In figure 11, at start the performance of the accuracy increase and the loss is decreasing as expected. But at a certain point (around 6/7 epoch) the performance descrease. This phenomen is called overfitting, it's occurs when the model is to specific to the train set and therefore less generic to other set. This is why the performance is test become worse but the performance in training are still improving.

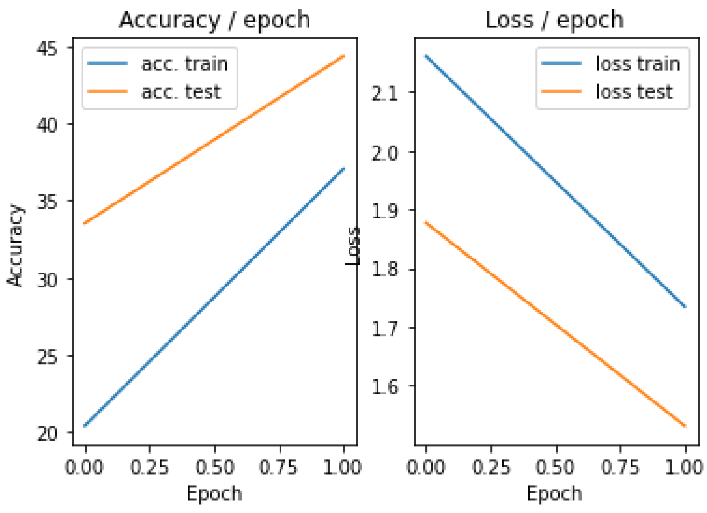


Figure 10: First epoch

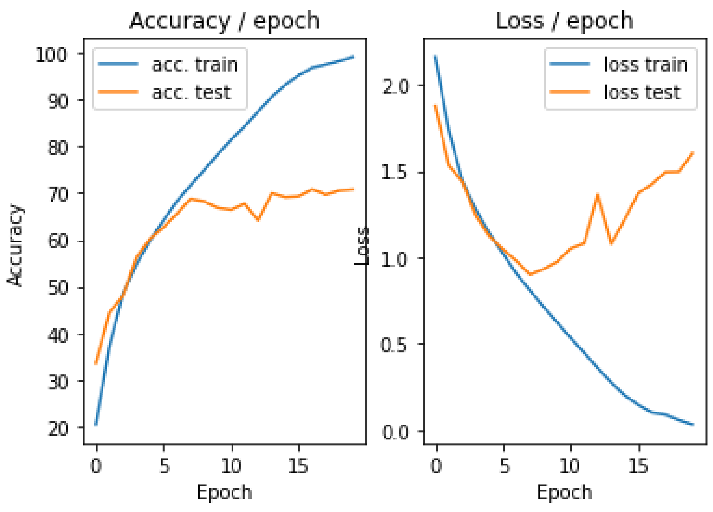


Figure 11: Last epoch

3 Results improvements

3.1 Standardization of examples

With CIFAR normalization :

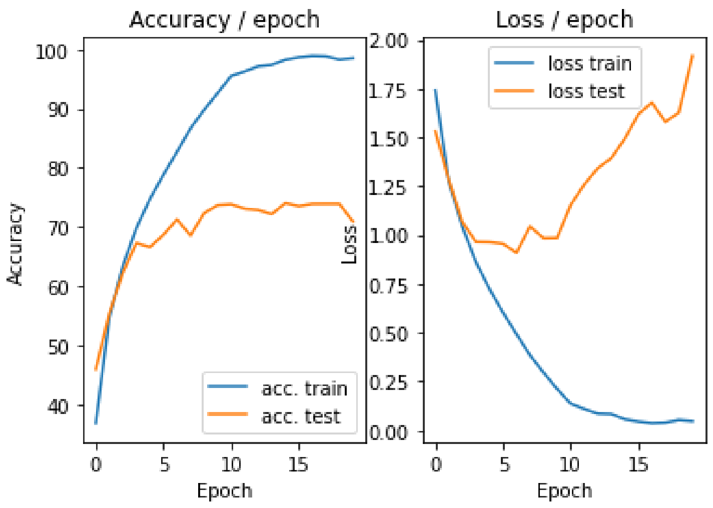


Figure 12: Norm CIFAR

19. Data normalisation through standardisation improves learning conditioning. In practice, this improves the performance in train and in test. The curves in train and in test do not have the same trend: we are still in a case of overfitting. There is a faster convergence with the normalisation.

20. Using on the validation set the normalization values calculated only from the training set would result in bias and not manage to fulfill the purpose of normalization as the given mean and standard deviation do not necessarily reflect the reality of the validation set.

21. There are other methods of normalisation. We can limit the intensity of a minimum and a maximum. We can also use a smoothing: after centring the data by subtracting the mean, we calculate the covariance matrix, then apply a PCA to this matrix. The number of components chosen allows us to reduce the size, and therefore to keep only the data that contain the greatest variance.

3.2 Increase in the number of training examples by data increase

With data augmentation :

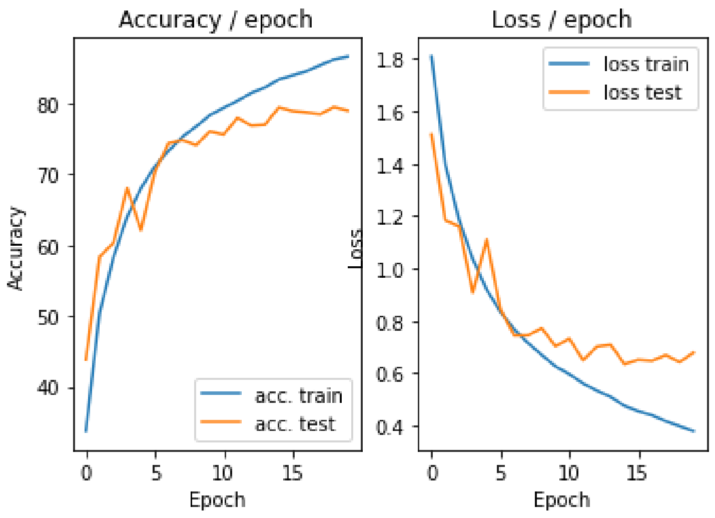


Figure 13: With data augmentation

22. We augmented the data by applying a random crop on our data and a horizontal symmetry with a probability of 0.5. With data augmentation we limit overfitting, but the convergence is little bit smaller, indeed there are more data.

23. Horizontal symmetry cannot be applied to all images, for example for letter or number recognition it will distort our data. However, for object or animal recognition it can be a good method of data augmentation.

24. Data augmentation with transformation of the dataset can bring new exemple in term of cropping, rotation but the exemples came from the same images, the same object. The risk of biasing and distorting images is high with data augmentation. It will also be necessary to apply pre-processing to our data in test for each image.

25. There are other data augmentation techniques such as rotation, translation and translation, adding noise to the image or combine several images. You will find an example of this at the end of the report.

3.3 Variants on the optimization algorithm

With optimisation :

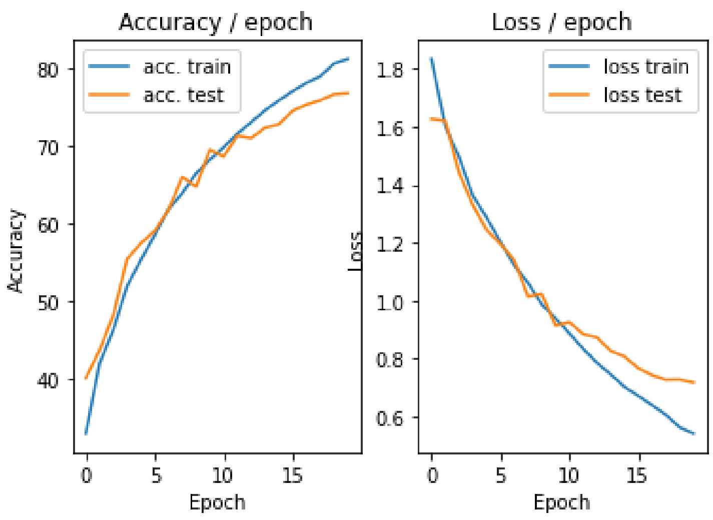


Figure 14: With optimisation

26. We modified the code to add a momentum of 0.9 to our SGD optimizer and to use a learning rate scheldurer for our gradient descent. We have less overfitting and more leaning stability.

27. For stochastic gradient descent it is common to oscillate around the local minimum, and therefore take longer to converge. Momentum accelerates the convergence by keeping the update done at the previous step and weighting it with a term $\gamma < 1$. When the new update is in the same direction as the previous one, then the descent is accelerated, otherwise when it is in a different or opposite direction, the descent is decreased. The learning rate scheldurer is an adaptive learning step, it is sometimes useful to have a large learning step at the beginning of the training in order to extract from a local minimum. On the other hand, as we learn, we would like our learning step to be reduced so that we can converge. We therefore have a better chance with this method of finding the global minimum of our cost function.

28. The most common and widely used optimizer is the ADAM optimizer which is effective for most Deep Learning tasks. There is also the Adagrad adaptive step optimizer, suitable for sparse data and others like Nadam, AMSGrad etc..

3.4 Regularization of the network by dropout

With dropout :

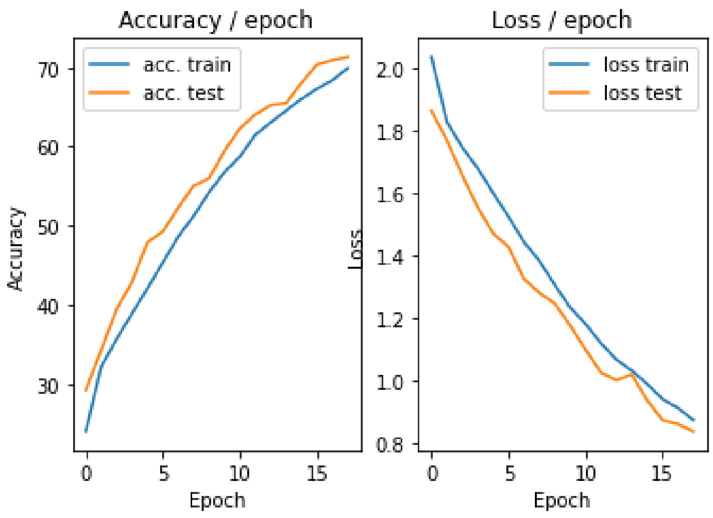


Figure 15: With dropout

29. With dropout we have less overfitting but slower convergence, indeed we loose data. The dropout is thus effective for the regularization of our network.

30. Regularisation is a process of penalising the complexity of the model. This reduces the risk of overfitting. Regularization enable to not be specific to a dataset but general.

31. As a reminder, the dropout layer allows to randomly deactivate at each pass a part of the of the neurons in the network to artificially reduce the number of parameters. Thus, this layer prevents overlearning because it prevents some nodes of the network from being more influential than others in the prediction. Indeed, if predictions are mainly made on a few nodes of the network, then we expose ourselves to overlearning because our network will have little flexibility. If we disable these dominant nodes during learning, this will force the other nodes to become more important and thus avoid overlearning. Furthermore, with this layer, the network weights will be higher than without. In cases where we don't have much data, the dropout makes the network more robust to image transformations (translations, rotations...).

32. This hyperparameter enable to control how much we dropout be defining a probability p where the weight will be set to 0 or not.
If this hyper-parameter is too small then the risk of overlearning is increased because the dominant nodes will be deactivated less often. On the other hand, if the hyper-parameter is too large, the weights of the network will be much higher than normal and therefore the final network will be more unstable.

33. During training, the dropout layer randomly deactivates some of the neurons in the network on each pass. On the other hand, during validation, the dropout layer will have no effect on the network, it will not deactivate any neurons in the network but just not use them. This is why we use model.train() and model.eval() which allows us to specify to Pytorch that we are either in training or in validation.

3.5 Use of batch normalization

With batchnorm :

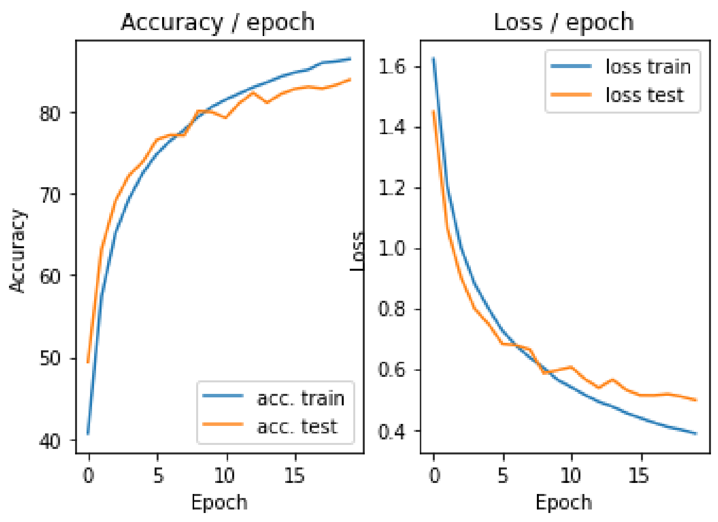


Figure 16: With batchnorm

34. This time we are experimenting with batch normalization. The principle is to no longer normalise on all the data, but only to focus on the examples we have in our batch. This allows each layer to learn more independently from the other layers of the network, and thus avoids over-learning. With batch normalisation we have a faster convergence.

Bonus 25. Additional data augmentation
The test are done with all the improvements of this praticals, we add an affine transformation in addition to the data augmentation done in question 22 with : transforms.RandomAffine(degrees=90, scale=(.9, 1.1), shear=0) :

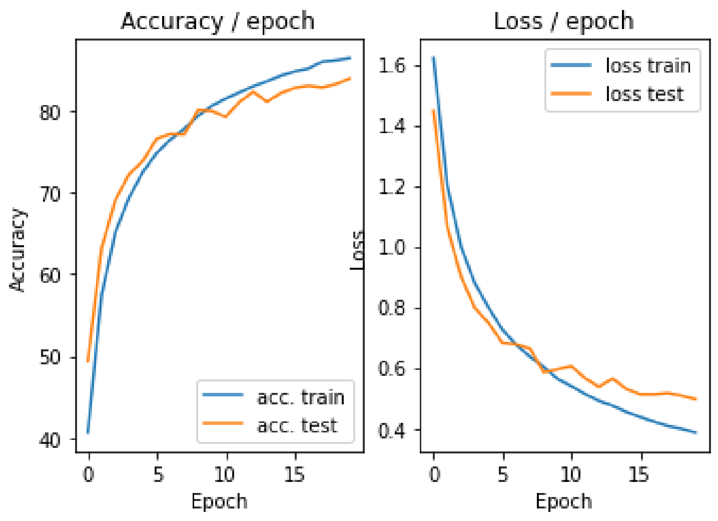


Figure 17: All improvement from praticals suject

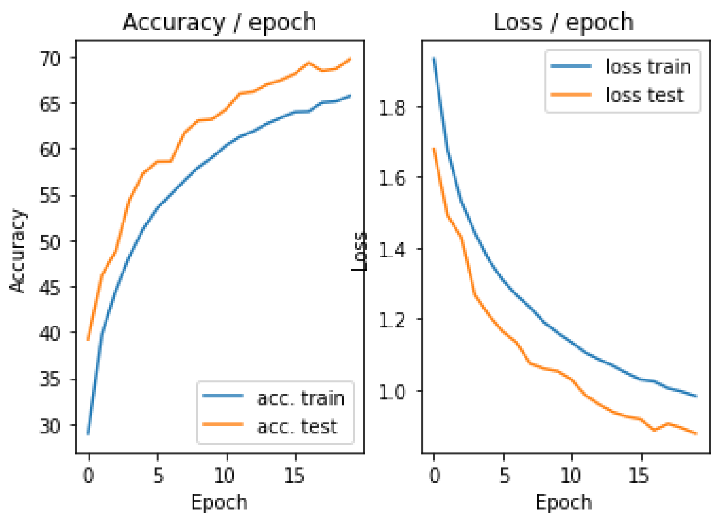


Figure 18: All improvement from praticals suject plus additional data augmentation

We can see even less overfitting when we increase the data augmentation transformation, however the convergence decrease because of the amount of data added.

Bonus 28. Adam optimizer
The test are done with all the improvements of this praticals, we just change SGD optimizer to Adam :

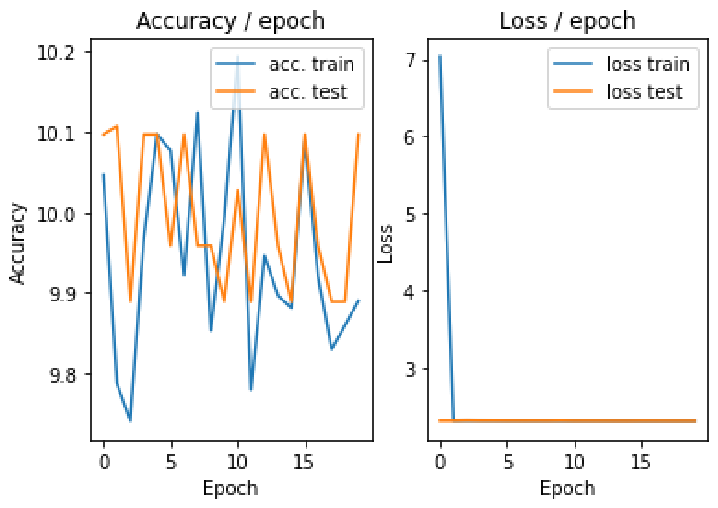


Figure 19: With Adam optimizer

We have bad result when we change change the optimizer, we need to find the best learning step of this optimizer to get better results.