Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours - Données Apprentissage et Connaissances

RDFIA

Report on practical sessions

Section 1 : Basics on Deep learning for computer vision

Done by:
BENHADDAD Sabrina
BENSIDHOUM Azzedine

November 2023

Contents

1 Introduction to Neural Networks **1**

1.1 Theoretical foundation . . . 1

1.1.1 Supervised dataset . . . 1

1.1.2 Network architecture (forward) . . . 2

1.1.3 Loss function . . . 4

1.1.4 Optimization algorithm . . . 5

1.2 Implementation . . . 10

1.2.1 Forward and backward manuals . . . 10

1.2.2 Simplification of the backward pass with *torch.autograd* . . . 12

1.2.3 Simplification of the forward pass with *torch.nn* layers . . . 14

1.2.4 Simplification of the SGD with *torch.optim* . . . 15

1.2.5 MNIST application . . . 15

1.2.6 Bonus : SVM . . . 16

2 Convolutional Neural Networks **19**

2.1 Introduction to convolutional networks . . . 19

2.2 Training from *scratch* of the model . . . 22

2.2.1 Network learning . . . 23

2.3 Results improvements . . . 28

2.3.1 Standardization of examples . . . 28

2.3.2 Increase in the number of training examples by data increase . . . 34

2.3.3 Variants on the optimization algorithm . . . 37

2.3.4 Regularization of the network by dropout . . . 39

2.3.5 Use of batch normalization . . . 42

List of Figures

1.1 Train, val and test sets . . . 2
1.2 Neural network architecture with only one hidden layer . . . 3
1.3 Gradient descent with small (top) and large (bottom) learning rates. Source: Andrew Ng's Machine Learning course on Coursera . . . 6
1.4 Circle dataset . . . 10
1.5 "Results of our algorithm applied to the Circle dataset with manual implementations of the forward and backward functions" . . . 10
1.6 Experimentation of the learning rate . . . 11
1.7 Experimentation of the batch size . . . 11
1.8 "Results of our algorithm applied to the Circle dataset with **torch.autograd**" 12
1.9 Experimentation of the learning rate with autograd . . . 13
1.10 Experimentation of the batch size with autograd . . . 13
1.11 "Results of our algorithm applied to the Circle dataset with autograd and **torch.nn**" . . . 14
1.12 "Results of our algorithm applied to the Circle dataset with autograd, torch.nn and **torch.optim**" . . . 15
1.13 MNIST dataset . . . 15
1.14 "Results of our algorithm applied to the MNIST dataset" . . . 16
1.15 Linear SVM on the Circles dataset . . . 16
1.16 SVM polynomial . . . 17
1.17 SVM sigmoid - . . . 17
1.18 SVM RBF . . . 17
1.19 Influence of regularization parameter C on the Accuracy . . . 18

2.1 Convolutional layers vs fully-connected layers - fewer parametres to learn . 20
2.2 Representation of the receptive fields . . . 22
2.3 Accuracy and loss plot - MNIST dataset . . . 24
2.4 Results achieved by training our neural network architecture on the CIFAR-10 dataset . . . 26
2.5 Learning rate experiments . . . 27
2.6 Batch size experiments . . . 27
2.7 Train and test results with standardization of examples . . . 29

2.8 Results - Standardization of examples 31
2.9 Results - Data augmentation 35

Chapter 1

Introduction to Neural Networks

The goal of this practical session is to create a simple neural network and gain a better understanding of these models and how to train them with the backpropagation of the gradient.
To do this, we will start by focusing on the theoretical foundation of the learning procedure of a perceptron with a hidden layer. Then, we will implement the network using the PyTorch library. We will first apply it to a simplified dataset in order to ensure its proper functioning, then we will apply it to the MNIST dataset.

1.1 Theoretical foundation

When exploring the learning procedure of the perceptron, our focus will be on the four essential components required for applying a neural network to any machine learning problem. These components include a supervised dataset, network architecture, a loss function, and an optimization algorithm for the minimization of the loss function.

1.1.1 Supervised dataset

In this context, we're engaging in a supervised classification task. This implies that we possess a supervised dataset, which we will partition into three distinct sets: the *training set*, the *testing set*, and if feasible, a *validation set*.

Question 1

These three sets are used for :

- The **training set** is used to train or fit the model (find the best weights and biases: parameters). In other words, the model learns from this data.
- The **validation set** is used to fine-tune the model hyperparameters, helping to find the best ones. It is distinct from the testing data.
- The **testing set** is used to evaluate the performance of the fully trained model, assessing how well the model performs with test data.

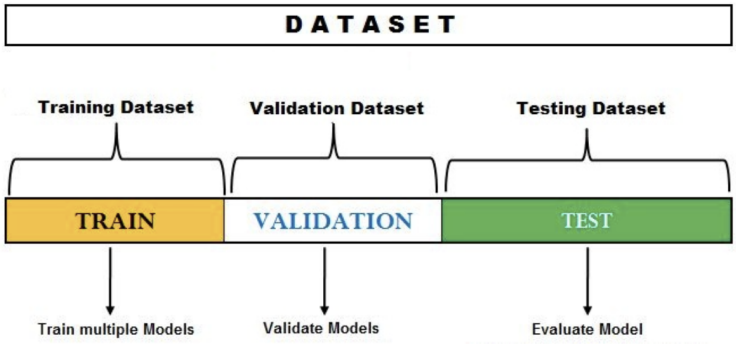

Figure 1.1: Train, val and test sets

Question 2

The number of examples N in a supervised classification task influences both the performance and the behavior of a model.
A larger dataset tends to promote better generalization and reduces the risk of overfitting, allowing for the use of more complex models. It also enables accurate performance estimation. However, a smaller N increases the risk of overfitting and can make it challenging to use complex models.

1.1.2 Network architecture (forward)

Question 3

It's important to add activation functions between linear transformations in order to introduce non-linearity.
This non-linearity allows the model to learn more complex representations (learn complex relationships between the input variables) and perform more complex tasks. We want our neural networks to work on complicated and non-linear tasks, and linear transformations would never be able to perform such tasks.

Question 4

The sizes n_x, n_h, n_y in the figure 1.2 are :

- $n_x = 2$ refers to the input size. It represents the number of input variables in the dataset.
- $n_h = 4$ refers to the hidden layer size. The choice of the number of neurons in the hidden layer depends on the complexity of the problem. A too large size can lead

to overfitting, while a too small one can lead to underfitting. That's why, we need to go through experimentation and tuning.

- $n_y = 2$ refers to the output size. It's determined by the number of classes of our problem.

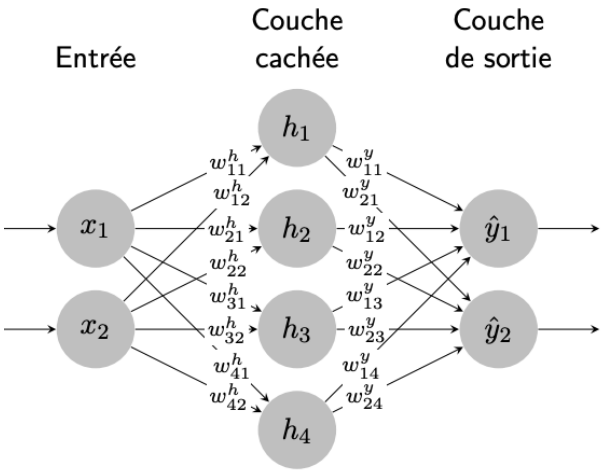

Figure 1.2: Neural network architecture with only one hidden layer

Question 5

The vector $\hat{y}$ represents the output of our neural network vector, the model's prediction (the predicted class probabilities or labels or predicted values). While y represents the ground truth, or the target values or the real data that we want to predict with our model.
The main difference between theses two vectors is called a residual or prediction error, represented in our network by the loss function.
We always want $\hat{y}$ to be as close as possible to y, which explains optimizing the model by minimizing the loss function.

Question 6

The SoftMax activation function transforms the raw outputs of the neural network into a vector of probabilities : the probability distribution over the input classes, which makes the model's outputs easier to interpret.

Question 7

The mathematical equations allowing to perform the *forward* pass of the neural network are :

$$\tilde{h} = W_h x + b_h \tag{1.1}$$

$$h = tanh(\tilde{h}) \tag{1.2}$$

$$\tilde{y} = W_y h + b_y \tag{1.3}$$

$$\hat{y} = SoftMax(\tilde{y}) \tag{1.4}$$

1.1.3 Loss function

Question 8

The main focus during training is to minimize the loss function, which measures the difference between the predicted outputs $\hat{y}$ and the ground truth values y, for each data point i.
For cross entropy, to decrease the global loss function $l(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i)$, $\hat{y}_i$ should be as close as possible to y_i. The derivative is :

$$\frac{\partial l}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

Which means that, $\hat{y}_i$ should be closer to 1 for the correct predicted classes, and closer to 0 for the other ones.
For squared error $l(y, \hat{y}_i) = \sum_i (y_i - \hat{y}_i)^2$, we also want $\hat{y}_i$ to be as close as possible to y_i in order to decrease it. The derivative is :

$$\frac{\partial l}{\partial \hat{y}_i} = -2(y_i - \hat{y}_i)$$

Which means that, $\hat{y}_i$ should be smaller than y_i in order to decrease the MSE.

Question 9

Cross Entropy is a well-suited loss function for classification problems, because it measures and minimizes the difference between two probability distributions : predicted class and real ones. CE also encourages the model to be confident in its predictions, as it penalizes incorrect classifications.
MSE on the other hand, is better suited for regression tasks, where the goal is not to classify data but to predict continuous values, because it measures the average squared difference between actual and predicted values. MSE also ensures that deviations from

the ground truth values are correctly penalized, which makes it excellent at emphasizing the importance of minimizing large errors.

1.1.4 Optimization algorithm

Question 10

In order to minimize the loss function and learn its parameters, we use the gradient descent algorithm. In fact, there are several variants of this algorithm, each one comes with its own sets of pros and cons :

- **Classic stochastic gradient** : in this case, we consider the losses of the whole training set at each single step or iteration. This version offers stability and deterministic convergence, since it computes gradients precisely on the whole dataset. It's also really great for convex or relatively smooth loss function. But, the main disadvantage of this variant is that it's computationally expensive for large dataset, since it considers all the training set for each iteration. But, the stability may also lead to get stuck in local minima with non-convex loss functions.
- **Mini-batch stochastic gradient (SGD)** : in this version, we consider just one example at a time to take a single step. It's computationally very fast and also memory efficient because of the consideration of one example at a time. SGD also converges faster and is well-suited with large datasets and non-convex loss functions. The noisy updates can also escape local minima and explore the loss landscape, but, it can slow down convergence by introducing noises and oscillations. One of the other cons is that SGD requires tuning of hyperparameters, such as the learning rate.
- **Online Stochastic Gradient Descent** : these versions can adapt really quickly to changing data distributions. It also offers very quick updates and require less memory, since it processes one example at a time, making in well-suited for large-scale online learning scenarios. However, it's less stable and deterministic compared to classic batch gradient descent, it also requires careful tuning of the learning rate, since it's noisy and high-variance updates can affect the convergence of the algorithm.

In general case, SGD (mini-batch stochastic gradient descent) strikes a good balance between efficiency and stability, showing its adaptability in various contexts.

Question 11

The learning rate is a really important hyperparameter and refers to the amount that the weights are updated during the training phase.

In other words, the learning rate controls how quickly the model is adjusting or adapting the weights of our model (with the respect, of course, of the loss gradient).
A small value of the learning rate allows the algorithm to travel slowly along the downward slope, because it makes smaller changes to the weights each update, and requires then more training epochs. It also allows making sure that we don't miss any local minima. However, this means that we will take a really long time to converge or possibly getting stuck on a plateau region.
On the other hand, a too large learning rate requires fewer training epochs and results in extremely rapid changes or updates of the weights in each iteration. But, this can lead the model to converge way too quickly to a suboptimal solution, as shown in the figure below.

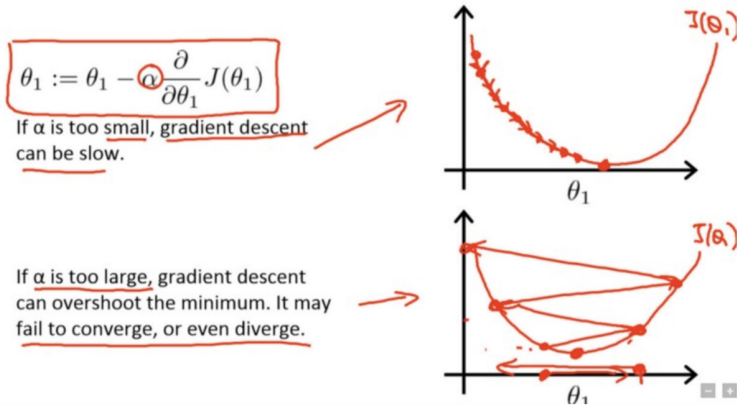

Figure 1.3: Gradient descent with small (top) and large (bottom) learning rates. Source: Andrew Ng's Machine Learning course on Coursera

To conclude, the learning rate is a very important hyperparameter that influences the convergence speed, stability, generalization, and optimization behavior of machine learning models.

Question 12

The complexity of calculating gradients in a neural network depends on the number of layers and can vary significantly between the naive approach and the backpropagation algorithm. The naive approach, which involves direct calculations for each parameter, becomes exponentially complex with more layers, making it impractical for deep networks. In contrast, the backpropagation algorithm effectively reduces complexity by sharing intermediate results between layers, resulting in linear complexity with the number of layers. This scalability makes backpropagation the preferred method for training deep neural networks.

Question 13

In order to allow optimization procedure such as backpropagation algorithm, the network architecture must use differentiable activation functions and loss functions with respect to their parameters. This is the major point for computing gradients during the training process.
Noted that the network architecture should also possess a feedforward structure and allow efficient and scalable forward and backward passes.

Question 14 - Simplifying the expression of the loss

We know that : $l(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i)$, $\hat{y}_i$ and according to the equation (1.4) $\hat{y} = SoftMax(\tilde{y}) = \frac{exp(\tilde{y}_i)}{\sum_j exp(\tilde{y}_j)}$ we have then :

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

Since $log(\sum_j e^{\tilde{y}_j})$ does not depend on i and $\sum_i y_i = 1$, we have the final expected expression of the loss :

$$l = -\sum_i y_i \tilde{y}_i + log(\sum_j e^{\tilde{y}_j}) \tag{1.5}$$

Question 15

Computation of the gradient of the cross-entropy loss relative to the intermediate output $\tilde{y}$:

$$\begin{aligned} \frac{\partial l}{\partial \tilde{y}_i} &= -y_i + \frac{\partial log(\sum_j e^{\tilde{y}_j})}{\partial \tilde{y}_i} \\ &= -y_i + \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} \\ &= -y_i + SoftMax(\tilde{y}_i) \\ &= \hat{y}_i - y_i \end{aligned}$$

Finally, we have $\nabla_{\tilde{y}} l = \hat{y} - y$

Question 16

Computation of the gradient of the loss with respect to the weights of the output layer.
Let's start with $\nabla_{W_y} l$:

$$\begin{aligned}\frac{\partial l}{\partial W_{y,ij}} &= \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{\partial W_{y,ij}} \\ &= \sum_k (\hat{y_k} - y_k) \frac{\partial \tilde{y_k}}{\partial W_{y,ij}}\end{aligned}$$

Since $\tilde{y} = W_y h + b_y$, which means that $\tilde{y_k} = \sum_j W^y_{k,j} h_j + b^y_k$
We can now, compute the partial derivative of $\tilde{y_k}$ with respect to $W^y_{k,j}$:

$$\frac{\partial \tilde{y_k}}{\partial W_{y,ij}} = \begin{cases} h_j & if \quad i = k \\ 0 & otherwise \end{cases}$$

We finally obtain the expression below :

$$(\nabla_{W_y} l)_{i,j} = \frac{\partial l}{\partial W_{y,ij}} = (\hat{y_i} - y_i) h_j$$

$$\nabla_{W_y} l = \nabla^T_{\tilde{y}} l * h$$

Let's focus now on, $\nabla_{b_y} l$:

$$\begin{aligned}\frac{\partial l}{\partial b_{y,i}} &= \sum_k \frac{\partial l}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{_{y,i}} \\ &= \sum_k (\hat{y_k} - y_k) \frac{\partial \tilde{y_k}}{\partial b_{y,i}} \\ &= (\hat{y_i} - y_i)\end{aligned}$$

We have then :

$$\nabla_{b_y} l = \nabla^T_{\tilde{y}} l$$

Question 17

Let's compute the other gradients :
1. $\nabla_{\tilde{h}} l$

$$\frac{\partial l}{\partial \tilde{h_k}} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h_i}}$$

We know that :

$$\frac{\partial h_k}{\partial \tilde{h}_i} = \frac{\partial \tanh(\tilde{h_k})}{\partial \tilde{h}_i} = \begin{cases} 1 - \tanh^2(\tilde{h}_i) = 1 - h_i^2 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

We also have $\tilde{y}_i = \sum_j W^y_{i,j} h_j + b^y_i$, so :

$$\frac{\partial l}{\partial h_k} = \sum_{j=1} \frac{\partial l}{\partial \tilde{y_j}} \frac{\partial \tilde{y}_j}{\partial h_k} = \sum_j (\hat{y}_j - y_j) W^y_{j,k}$$

Finally,

$$\begin{aligned} \frac{\partial l}{\partial \tilde{h}_i} &= \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} \\ &= (1 - h_i^2) \left(\sum_j (\hat{y}_j - y_j) W^y_{j,i} \right) \\ &= \delta^h_i \end{aligned}$$

So $\nabla_{\tilde{h}} l = (\nabla_{\tilde{y}} l * W^y) \odot (1 - h^2)$
2. $\nabla_{W_h} l$

$$\frac{\partial l}{\partial W^h_{i,j}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W^h_{i,j}}$$

According to the previous questions, we already have $\frac{\partial l}{\partial \tilde{h}_k}$ and we know that : $\tilde{h}_k = \sum_{j=1}^{n_x} W^h_{k,j} x_j + b^h_k$, it's now easier to compute then the derivative :

$$\begin{aligned} \frac{\partial l}{\partial W^h_{i,j}} &= \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W^h_{i,j}} \\ &= \delta^h_i x_j \\ \nabla_{Wh} &= \nabla^T_{\tilde{h}} l * x \end{aligned}$$

3. $\nabla_{b_h} l$

$$\frac{\partial l}{\partial b^j_i} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b^h_i} = \delta^h_i$$

We have, then :

$$\nabla_{b^h} = \nabla^T_{\tilde{h}} l$$

1.2 Implementation

Now that we have all the necessary equations for forward predictions, loss evaluation, and the backward gradient descent process, we're ready to move on to the practical implementation of these concepts using PyTorch.
Our first experiments will be concluded on the "Circle" dataset, which consists of data points distributed in two separate circles, making it a common choice for addressing non-linear classification problems.

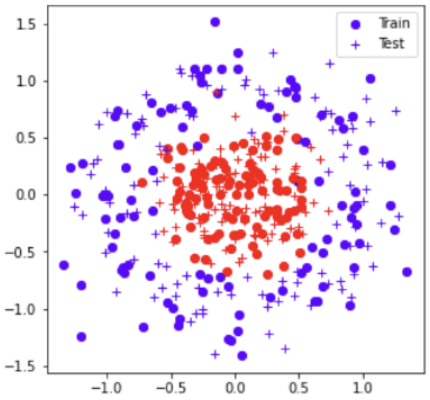

Figure 1.4: Circle dataset

1.2.1 Forward and backward manuals

In this section, we meticulously handcrafted the forward and backward functions, employing the SGD optimization algorithm with 10 batches and a learning rate of 0.03. After 150 iterations, our model impressively attains a convergence point, achieving a remarkable accuracy of 96.5% on the training data and maintaining a strong performance of 94% on the testing dataset, as shown in the figure below :

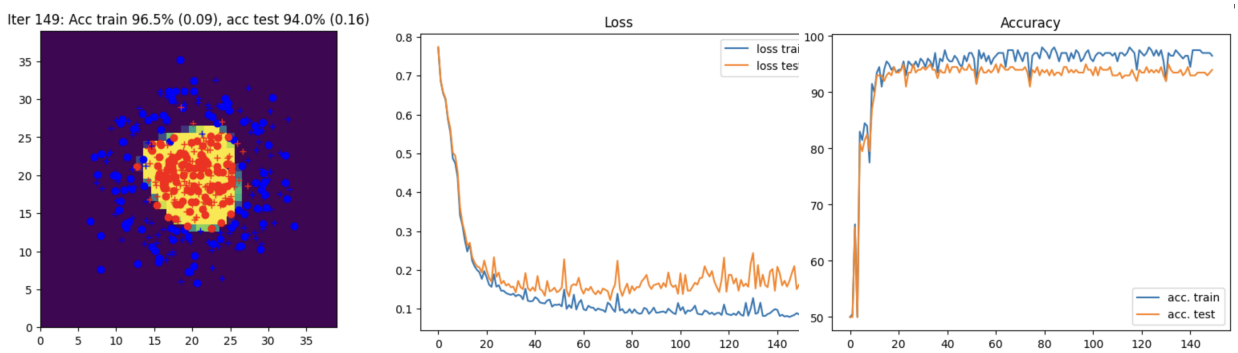

Figure 1.5: "Results of our algorithm applied to the Circle dataset with manual implementations of the forward and backward functions"

We can observe then, by the end of the training, circular decision boundaries for the circular data since the model is able to classify the data correctly.

Influence of the learning rate and the batch size

Let's now experiment with various learning rates and batch sizes to observe their respective influences on the model's performance.
The results presented in the figure below align with the expectations discussed earlier.

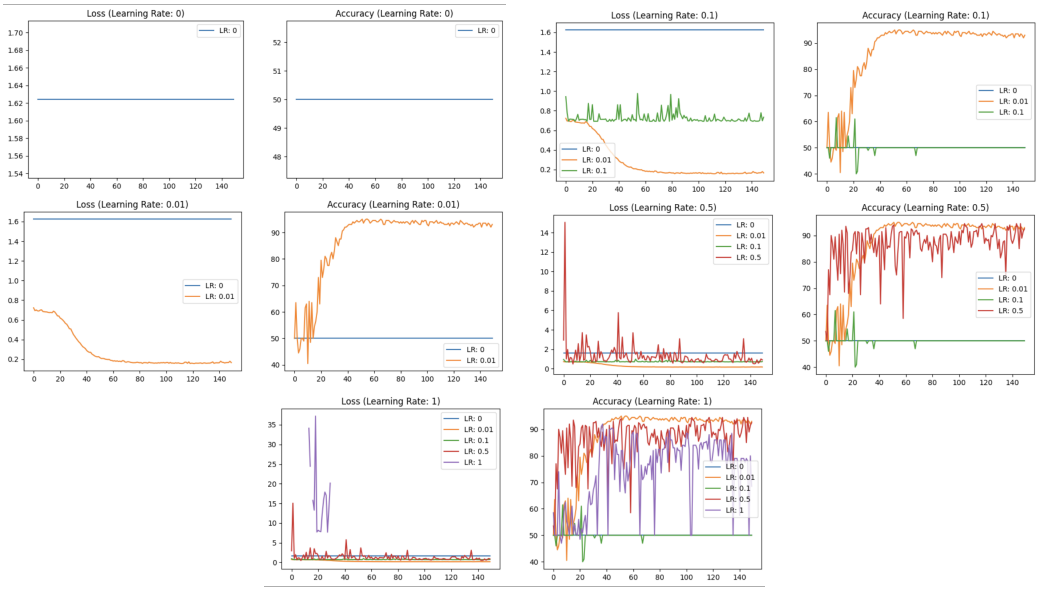

Figure 1.6: Experimentation of the learning rate

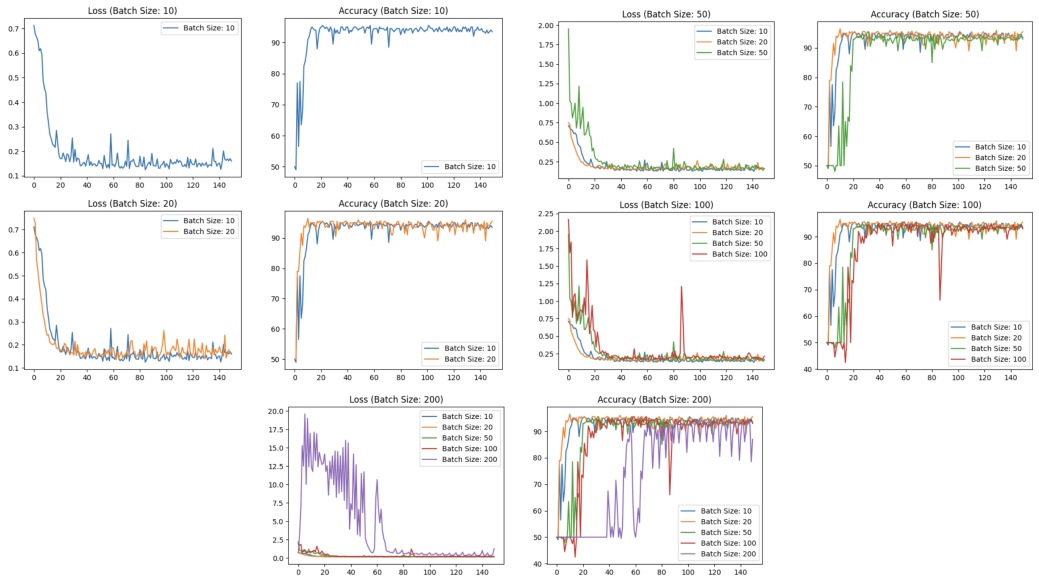

Figure 1.7: Experimentation of the batch size

1.2.2 Simplification of the backward pass with *torch.autograd*

In this section, we are going to use the PyTorch's automatic differentiation mechanism, known as Autograd. By default, tensors in PyTorch do not enable this feature, but we can activate it by setting requires_grad to True when creating a tensor or by defining the flag afterward. Autograd allows us to compute gradients, and with a simple loss.backward(), PyTorch efficiently computes the derivatives of the loss with respect to all tensors that contributed to it. These gradients are stored in the grad attribute of the respective tensors, such as W.grad for a leaf tensor W. We'll explore how this mechanism enhances our ability to perform automatic differentiation in PyTorch.

We rerun the learning algorithm with this modification and obtain the results shown in the following figure. It's worth noting that we used the same hyperparameters as in the previous run. Consequently, we achieved an accuracy of 96.5% for training and 94.5%for testing, which is consistent with the manual implementation of the previous forward and backward functions. Noted that the training process seems to be a bit more stable, as

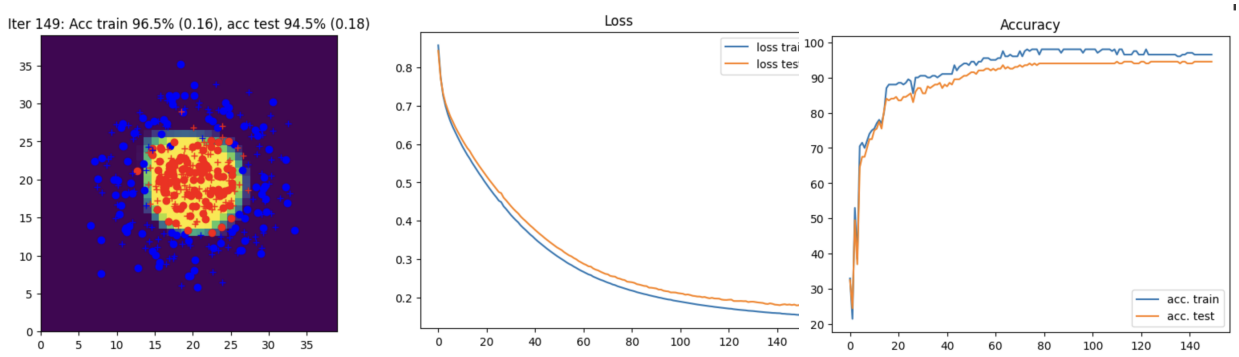

Figure 1.8: "Results of our algorithm applied to the Circle dataset with **torch.autograd**"

shown by the loss curve.

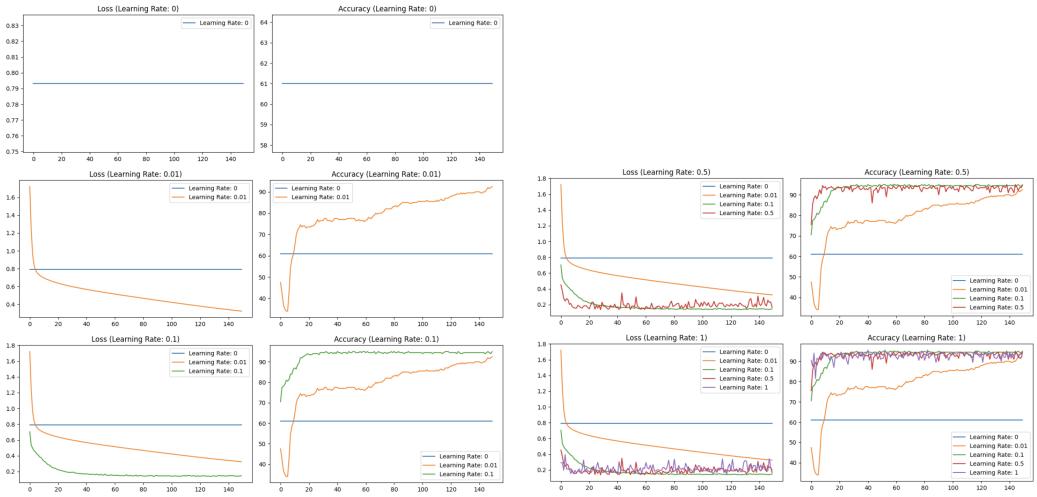

Figure 1.9: Experimentation of the learning rate with autograd

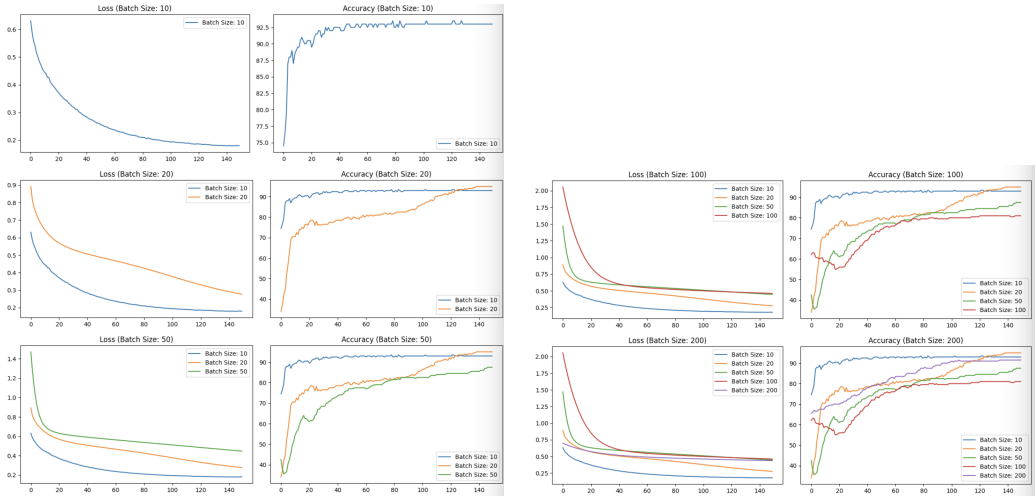

Figure 1.10: Experimentation of the batch size with autograd

1.2.3 Simplification of the forward pass with *torch.nn* layers

In this section, we will simply even more our network architecture, thanks to the Pytorch's "nn" package, which defines a set of modules for building neural networks.
By using 'torch.nn' to define the neural network, we can now apply forward passes directly by invoking the model, eliminating the need to manually implement the 'forward' method.

We achieve a higher training accuracy of 98% and maintain a testing accuracy of 94.5% when using 'torch.nn.' However, the training process is slightly longer with 'torch.nn' compared to manual implementation. The increased training time can be attributed to the additional overhead introduced by the abstraction and flexibility of PyTorch's 'nn' package, which allows for dynamic computation graphs and automatic differentiation. While this overhead may result in longer training times, it often leads to greater stability and ease of use in more complex models and scenarios. The stability observed in the plots can be attributed to the robustness and built-in optimization provided by PyTorch's 'nn' package.

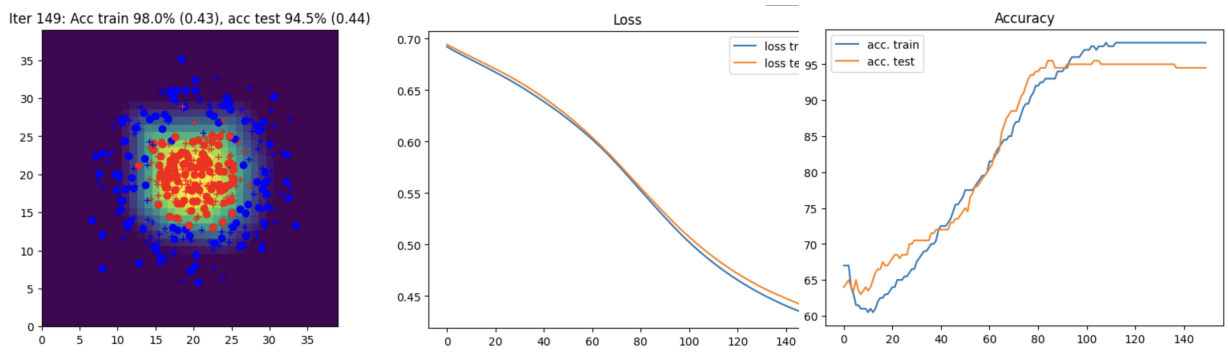

Figure 1.11: "Results of our algorithm applied to the Circle dataset with autograd and **torch.nn**"

1.2.4 Simplification of the SGD with *torch.optim*

Up until now, we have been manually updating the parameters during the SGD learning algorithm. However, the 'torch.optim' module includes several optimizers, including SGD, that take care of the parameter updates using the 'optim.step()' method. This simplifies the process of parameter updates and offers various optimization algorithms for more efficient and automated training.

The results, as depicted in Figure 1.8, demonstrate that the training process is significantly more stable than other computations in the algorithm and yields superior performance in the test phase (97% accuracy for training and 95% for testing).

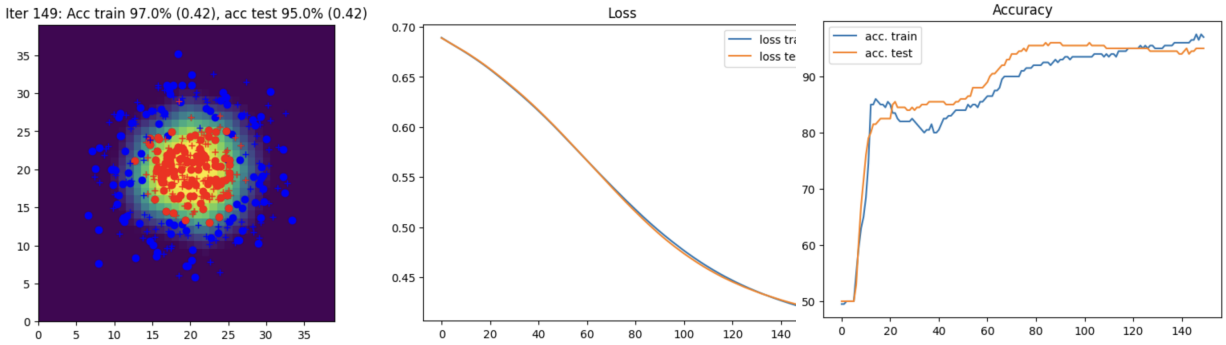

Figure 1.12: "Results of our algorithm applied to the Circle dataset with autograd, torch.nn and **torch.optim**"

1.2.5 MNIST application

In the upcoming section, we will apply our code to the MNIST dataset, a collection of handwritten digit images. The MNIST dataset consists of images of digits, each representing one of the 10 possible classes ($n_y = 10$). These digit images are 28×28 pixels in size, and we will represent them as vectors of 784 values.

Figure 1.13: MNIST dataset

The objective of our simplified neural network is to predict the class (a digit) from a handwritten image of that image.

The figure below shows the evolution of the loss and accuracy as a function of the number of iterations for our neural network applied to the MNIST dataset. We can observe that the model performs exceptionally well with approximately 95% accuracy, both on the training and testing datasets. However, it's worth noting that a slight instability may be attributed to the model's hyperparameters. Additionally, the confusion matrix provides strong support for our observations.

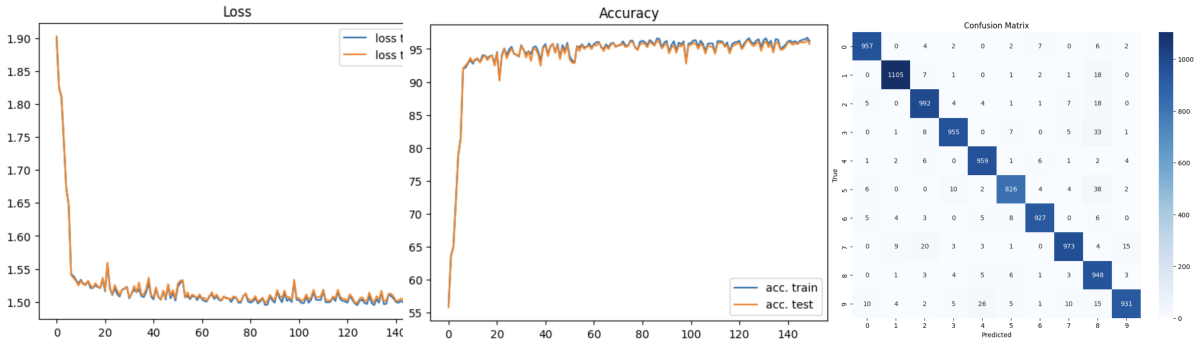

Figure 1.14: "Results of our algorithm applied to the MNIST dataset"

1.2.6 Bonus : SVM

In this section, we will train an SVM model on the Circles dataset.
First, we will try a linear SVM. The results are shown in the figure below :

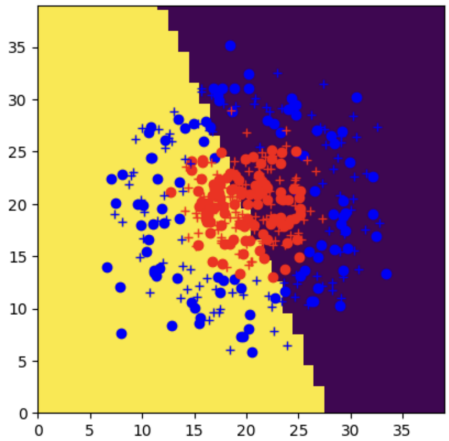

Figure 1.15: Linear SVM on the Circles dataset

With an accuracy of 53.0%, it appears that this version of the SVM is performing a linear separation (with a line) that is not well-suited for this problem. Consequently, its performance is not necessarily better than that of a random model. This suggests that the SVM

with this configuration is unable to capture the complex relationships in the data, and it may be necessary to explore more advanced methods or different parameters to improve performance.

Since Linear SVM doesn't perform well with circular data, let's try more complex kernels.
We have implemented three SVM models using three different kernels: radial basis function (rbf), sigmoid, and polynomial (poly). The results are as follows: The polynomial SVM

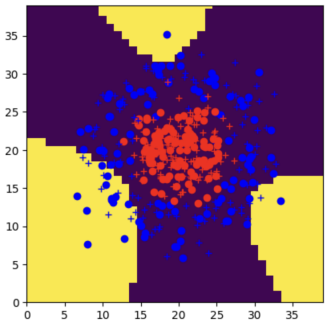

Figure 1.16: SVM polynomial

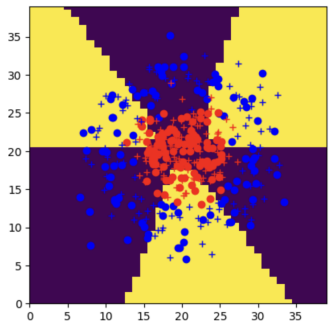

Figure 1.17: SVM sigmoid -

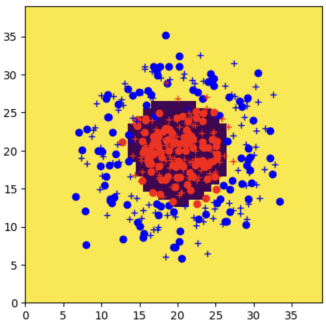

Figure 1.18: SVM RBF

achieved an accuracy of 54.5%, the sigmoid SVM achieved 65%, and the RBF kernel SVM achieved 94%. Therefore, we can conclude that the Radial Basis Function (RBF) kernel is the most suitable for this classification task, which is clearly evident in the accompanying figure.

The results are not surprising, as the Radial Basis Function (RBF) kernel is often the most suitable choice for circular data, given its ability to effectively capture complex and nonlinear relationships and patterns. Additionally, it can readily learn circular decision boundaries.

Now, let's proceed with a final series of experiments, focusing on the influence of the regularization parameter C in the SVM.
The regularization parameter C controls the trade-off between fitting the training data and preventing overfitting in a machine learning model. A smaller value of C results in stronger regularization, while a larger value allows the model to fit the training data more closely.

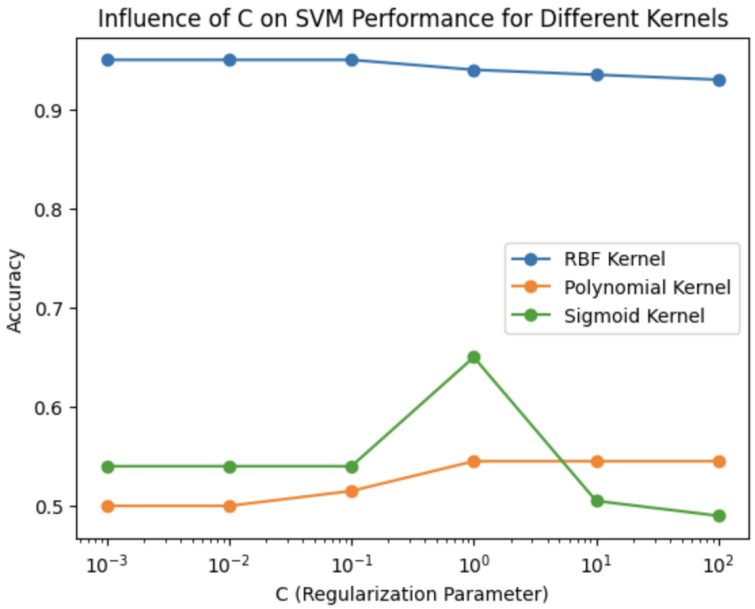

Figure 1.19: Influence of regularization parameter C on the Accuracy

In summary, our experimental investigation focused on the impact of the regularization parameter C on Support Vector Machines (SVM) employing three distinct kernels: RBF, polynomial, and sigmoid. We observed that the choice of C wielded significant influence on model performance.

With the RBF kernel, elevating C generally improved accuracy up to a certain threshold, beyond which it led to overfitting. In contrast, the polynomial kernel's performance remained relatively stable across different C values, exhibiting less sensitivity to regularization strength. However, when it came to the sigmoid kernel, higher C values resulted in reduced accuracy, implying that excessive regularization hindered its performance.

These findings underscore the importance of judiciously selecting the appropriate regularization parameter C, contingent on the dataset's characteristics and the kernel in use. Proper C tuning can lead to enhanced model generalization and improved classification accuracy. Hence, when employing SVMs with various kernels, meticulous C parameter tuning is vital to achieving optimal results.

Chapter 2

Convolutional Neural Networks

The goal of this practical session is to become familiar with convolutional neural networks. First, we will study the classical layers of this type of network and then, set up an initial network trained on the standard CIFAR-10 dataset.
Finally, we will discuss various techniques to improve the network learning process, including standardization of examples, data augmentation, variants of SGD, dropout and, lastly, batch normalization.

2.1 Introduction to convolutional networks

Question 1

Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size $x \times y \times z$, the output size will be :

- $x^{'} = \frac{x + 2p - k}{s} + 1$
- $y^{'} = \frac{y + 2p - k}{s} + 1$
- $z^{'} = 1$ since we're dealing with a single convolution filter

We divide by the stride because, when we move between two convolutions, we are essentially reducing or dividing the output size by that number. We also add $2p$ because the padding is applied to both sides of the input.

The number of weights to learn in this case is : $k \times k \times z$ (z is the number of channels).

For the fully-connected layer, to produce an output of the same size as the input, we would need to learn $x \times y \times z$ weights.

Question 2

The advantages of convolution over fully-connected layers :

- Invariance and robustness of the representation : Unlike fully connected layers, convolutional layers keep spatial typology, allowing them to capture and identify hierarchical patterns within the spatial arrangement of an input. Convolutional layers

can capture relevant features from an image at different levels, much like the human brain.

- Fewer parameters to learn: Due to weight sharing and local connectivity, convolutional layers have fewer weights to learn. This makes CNNs more efficient in terms of memory and complexity.

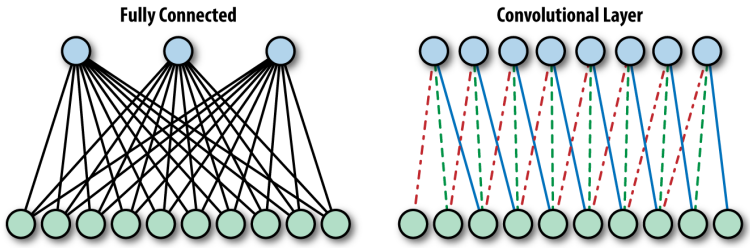

Figure 2.1: Convolutional layers vs fully-connected layers - fewer parametres to learn

- Better generalization and performance thanks to the shared weights. This is because the shared weights allow the network to learn more robust and generalized features that are effective across different parts of the input.

Its main limit :
As observed in the course lectures, while CNNs excel at capturing local spatial patterns, they may have difficulties in recognizing global spatial patterns.

Question 3

After convolution layers, we use spatial pooling in order to reduce the spatial dimensions of the features maps to reduce the amount of parameters and computation in the network, and hence to also control overfitting.
Pooling is also used to make the detection of features invariant to their position in the input. The network become then less sensitive to small translations in the input, which is essential for tasks like object recognition.

Question 4

Suppose we try to compute the output of a classical convolutional network for an input image larger than the initially planned size (224×224 in the example). Without modifying the image, we can indeed use parts of the layers of the network.
This is because convolutional layers are independent of the size of the image (it needs only the number of input and output channels, kernel size and eventually padding and stride settings).
However, we may have a problem when we reach with the fully-connected layers in the

network. These layers expect a fixed size input, and adapting them to accommodate the larger image may require architectural adjustments.

Question 5

We actually can analyze fully-connected layers as particular convolutions.
To do so, we simply need to consider a convolutional layer with a kernel (filter) that is the same size as the input feature map (without padding and stride = 1) and a number of filters equivalent to the number of output neurons of the fully-connected layer. The convolution operation will calculate the weighted sum of all the elements of the input feature map, which is exactly what a fully-connected layer does.

Question 6

By replacing the fully-connected layers by their equivalent in convolutions, the network can process images of different sizes, since it can adapt to the size of the input. However, this adaptation may not always be ideal, one of the useful solution to address the issue of making CNNs more flexible while dealing with different sizes of images would be *Global Average Pooling*.

Question 7

A receptive field is the region in the input space that affects the features of a particular layer. It's the part of a tensor that after convolution results (It's the portion of the image used to compute the activation of a neuron in a feature map).
The receptive field is determined by the size and stride of the convolutional kernels and the number of layers in the network.

The sizes of the receptive field of the neurons :
- The first convolutional layer : let's take the example of Figure 1, the kernel size is 3 and stride is 1, so the receptive fields (number of pixels used) is 3×3.
- The second convolutional layer we consider the 3×3 neighborhood from the previous layer plus additional border pixels, the receptive fields will be 5×5.

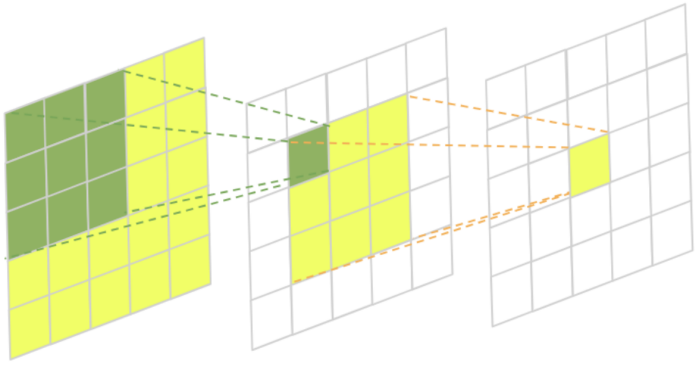

Figure 2.2: Representation of the receptive fields

The receptive fields become larger as we go deeper into the network. We can interpret this by how much context a neuron can see in the input.
Deeper layers can understand global pattern while early layers focus on local details only such as edges.

2.2 Training from *scratch* of the model

Question 8

In order to preserve the spatial dimensions of the input, we fix the stride to 1.
The expression of the padding will be then : $p = \frac{k-1}{2}$ with k representing the size of the kernel.
In our architecture, $k = 5$, the padding will be then equal to 2.
To conclude, we need $p = 2$ and $s = 1$ values.

Question 9

The pooling layers are in charge of downsampling the spatial dimensions of the input.
So, to reduce the spatial dimensions by a factor of 2, we generally use max-pooling layers with zero padding and a stride of 2. This downsampling by 2 occurs both in width and height.

Question 10

In this question, we're going to indicate, for each layer, the output size and the number of weights to learn. The input size is $32 \times 32 \times 3$, the number of weights to learn is $k \times k \times z$ for a single convolution, we need then to multiply it by the number of convolutions filters used.

- Conv1 - output size : $32 \times 32 \times 32$, number of weights to learn $5 \times 5 \times 3 \times 32 = 2400$

- Pool1 - output size : $16 \times 16 \times 32$
- Conv2 - output size : $16 \times 16 \times 64$, number of weights to learn $5 \times 5 \times 32 \times 64 = 51200$
- Pool2 - output size : $8 \times 8 \times 64$
- Conv3 - output size : $8 \times 8 \times 64$, number of weights to learn $5 \times 5 \times 64 \times 64 = 102400$
- Pool3 - output size : $4 \times 4 \times 64$
- fc4 - output size : 1000, number of weights to learn $4 \times 4 \times 64 \times 1000 + 1 \times 1000$ (we add the 1 due to the biase)
- fc5 - output size : 10, number of weights to learn $1000 \times 10 + 10$

Comments :
The output size and the number of weights are influenced by the hyperparameters of the network.
The number of weights in fully-connected layers, as mentioned before, is significantly high compared to the number of weights to learn in convolutional layers, making the network more computationally demanding.

Question 11

The total number of weights to learn is : $2400 + 51200 + 102400 + 4 \times 4 \times 64 \times 1000 + 1000 + 1000 \times 10 + 10 = 1191010$ which is much smaller than the number of examples to learn $50k$ images in the *train* dataset. This observation highlights an important aspect of deep learning : the number of parameters to learn is usually smaller than the number of training examples.

Question 12

With the BOW and SVM approach, the number of parameters to learn is determined by the vocabulary size (visuals words) and the number of classes to predict.
In our architecture, we have $10k$ images and 10 classes, so the number of parameters to learn would be 1000000, which is much smaller than CNNs.
This difference highlights one of the key distinctions between the two approaches, in terms of model complexity and the ability to capture complex relationships in data.

2.2.1 Network learning

To learn the network, we start by reading and experimenting the code given.

Question 13

First, we initiate the training process with the following parameters :

- Batch size= 128
- Learning rate lr = 0.1
- Number of epochs = 50
- CUDA = True (in order to execute only with GPU : GPU Acceleration)

The figures below, show the accuracy and loss plot obtained by training the network on the MNIST dataset.

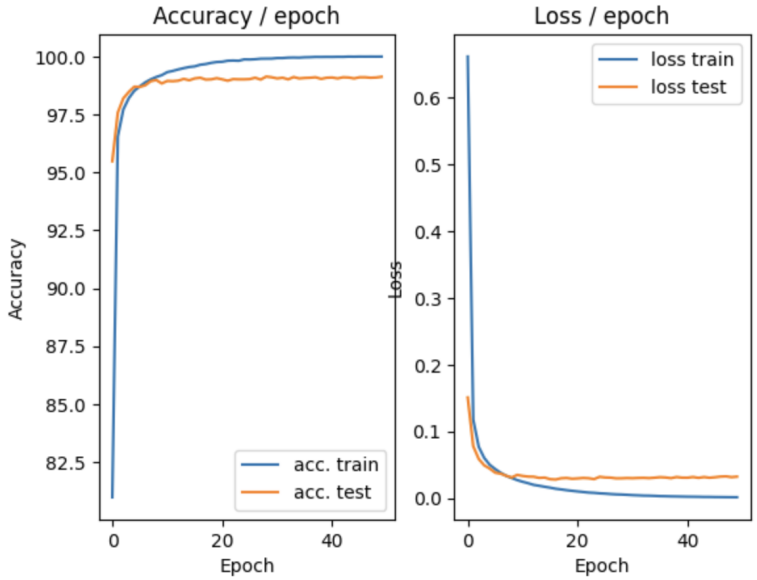

Figure 2.3: Accuracy and loss plot - MNIST dataset

Question 14

In the code provided, the following instruction **model.eval() if optimizer is None else model.train()** is used to switch the model between the two modes : training and evaluation, depending on the presence or absence of the optimizer.
If the model is in evaluation mode, the computations related to gradient calculations required for the training phase, such as calling ***backward()*** on the loss (***backpropagation***) and parameters updates, are omitted.
This leads to faster code execution and memory savings.
To conclude, during training, we compute and backpropagate the loss in order to update the parameters of the mode, which is not done during evaluation.

Question 15

Implementation of the architecture requested above on the CIFAR-10 dataset.

```
def __init__(self):
        super(ConvNet, self).__init__()
        # We first define the convolution and pooling layers as a features
            extractor
        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)
        )
        # We then define fully connected layers as a classifier
        self.classifier = nn.Sequential(
            nn.Linear(1024, 1000), #1024 = 4*4*64
            nn.ReLU(),
            nn.Linear(1000, 10),
            # Reminder: The softmax is included in the loss, do not put it here
        )

    #Instructions to use the CIFAR-10 dataset
    train_dataset = datasets.CIFAR10(PATH, train=True, download=True,
    transform=transforms.Compose([
            transforms.ToTensor()
        ]))
    val_dataset = datasets.CIFAR10(PATH, train=False, download=True,
        transform=transforms.Compose([
            transforms.ToTensor()
        ]))
```

Note that we used 100 epochs so that the model has finished converging. The figure below displays the results achieved by training our neural network architecture on the CIFAR-10 dataset.

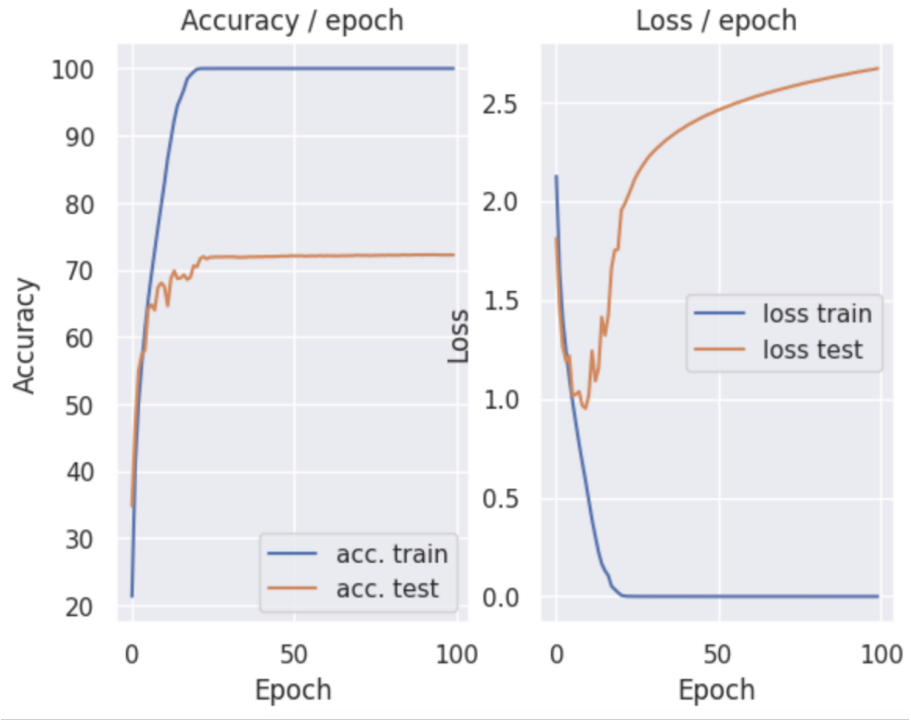

Figure 2.4: Results achieved by training our neural network architecture on the CIFAR-10 dataset

Question 16

The learning rate and batch size are two critical hyperparameters that have a significant impact on the training of neural networks.

- **The learning rate**
  The learning rate is a hyperparameter used in the training of neural networks in order to control how quickly the model is adapted to the problem.
  A learning rate too large during the training of a CNN, may lead to instability, overfitting and divergence. The model becomes indeed excessively tuned to the training data, losing its ability to generalize well. This can also cause the model to converge too quickly to a suboptimal solution.
  Whereas, a learning rate too small can cause the process to get stuck, slow convergence and longer training times, as shown in the figure below.

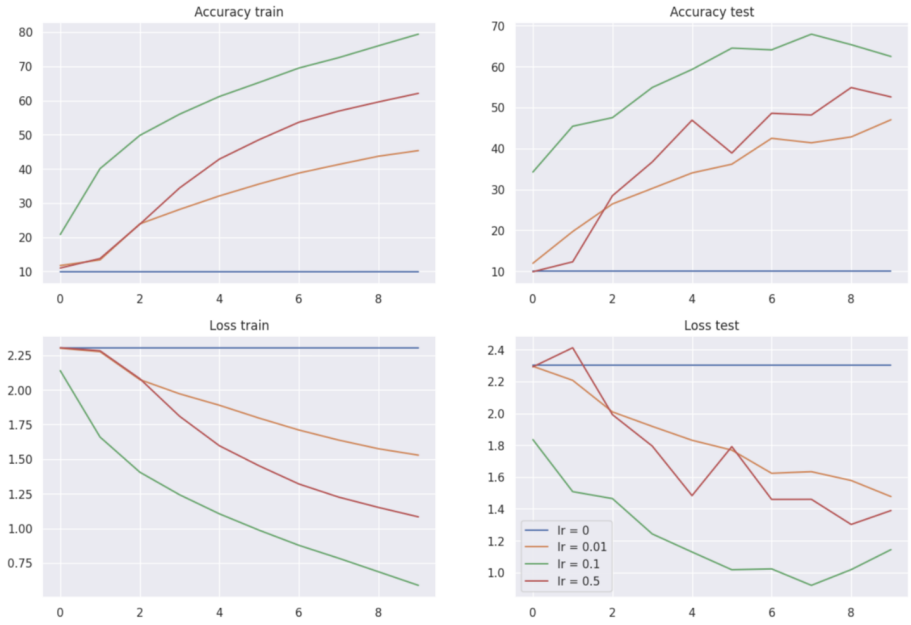

Figure 2.5: Learning rate experiments

- **The batch size**

  The batch size is the number of training examples (samples) used in one iteration (before the model is updated). It affects the efficiency and stability of the training process.

  A large batch size accelerates training since more examples are used in each iteration (it stabilizes the loss very quickly and early in the training process), but requires more memory resources.

  On the other hand, a smaller batch size can lead to slower convergence and make the loss more unstable, as shown on the figure below.

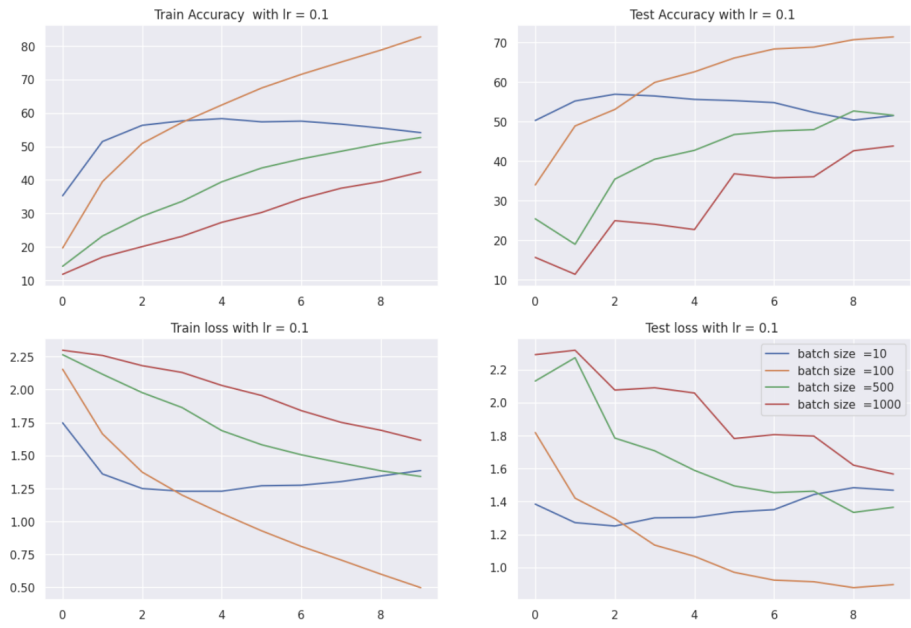

Figure 2.6: Batch size experiments

Question 17

At the start of the first epoch, the error in train and test corresponds to the initial random weight initialization of our network.

Question 18

As shown in the figure 1.3, we immediately observe that the test loss starts to increase after around 13 epochs. The phenomenon is called **overfitting**, which means that our model learns to fit the training data too closely, rather than learning the underlying patterns.

2.3 Results improvements

In this section, we will see several classic techniques to improve the performance of our model.

2.3.1 Standardization of examples

Note
All experiments are based on lr = 0.1, batch_size = 128 and epoch = 40.

Baseline model results:

Training Data :

- Time taken : 9 seconds
- Average Loss : 0.0001
- Precision at 1 and 5 : Both 100%

Test Data :

- Time taken : 1 second
- Average Loss : 2.2823
- Precision at 1 : 72.81%
- Precision at 5 : 97.55%

Question 19

Training Data :

- Time taken : 14 seconds
- Average Loss : 0.0004
- Precision at 1 : 100.00%
- Precision at 5 : 100.00%

Test Data :

- Time taken : 3 seconds
- Average Loss : 1.7286
- Precision at 1 : 76.97%
- Precision at 5 : 98.14%

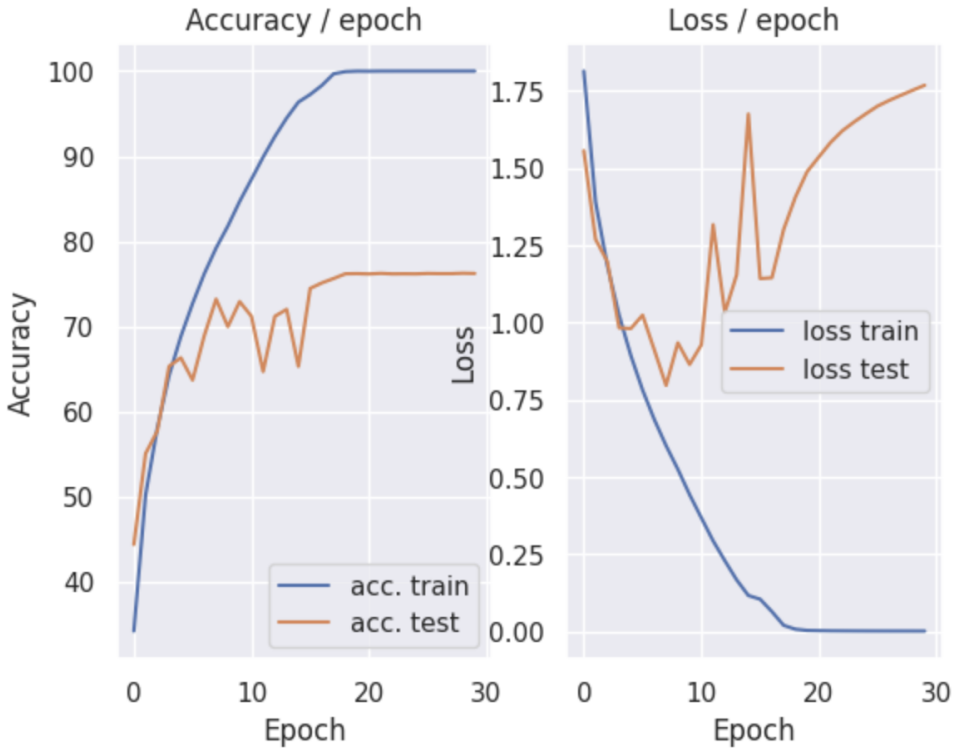

Figure 2.7: Train and test results with standardization of examples

From the given image, during the training phase, the model's accuracy steadily increased, eventually reaching close to 100%. The training loss showed a sharp decline as epochs progressed, indicating that the model learned effectively from the training data.

Conversely, in the testing phase, while the accuracy remained relatively stable around $76-78\%$, the test loss increased significantly after initial epochs, suggesting the potential onset of overfitting.

Comparison with results (without standardization of examples):

Training Data (without standardization):

- Time taken : 9 seconds (faster compared to with standardization)
- Average Loss : 0.0001 (lower than standardized data)
- Precision at 1 and 5 : Both 100%

Test Data (without standardization):

- Time taken: 1 second (quicker than with standardization)
- Average Loss: 2.2823 (higher than standardized data)
- Precision at 1 : 72.81% (lower by 4.16% than standardized data)
- Precision at 5 : 97.55% (lower by 0.59% than standardized data)

Comparison Analysis :

- *Training Performance :* The non-standardized data resulted in a slightly lower average loss and took less time. Precision was consistent at 100% for both cases.
- *Test Performance :* Non-standardized data showed a higher average loss and lower precision, indicating worse performance.
- *Overfitting :* Potential overfitting is observed in both cases, but more pronounced without standardization.

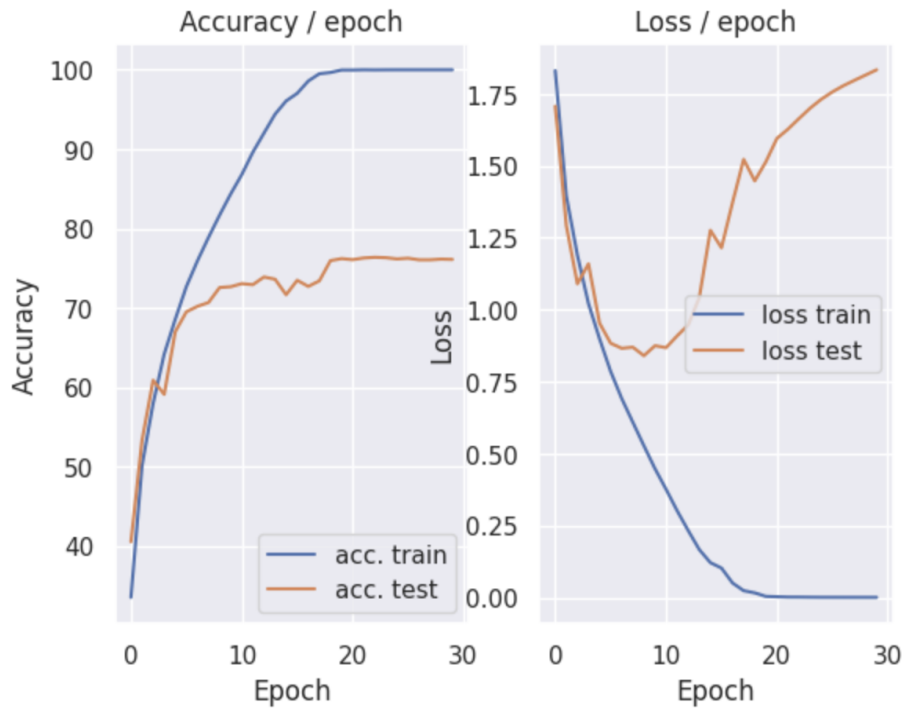

Figure 2.8: Results - Standardization of examples

In conclusion, standardizing the examples provides better generalization on the test set, resulting in improved performance metrics despite a slight increase in computational times.

Question 20

In general, learning algorithms work on the principle that the test set corresponds to a data set that the model has never seen before. As a result, normalizing the examples in the validation set with the average image in the training set can seem confusing, but this can be explained in different ways:

- *Avoid Overfitting :* If you include validation/test data to calculate the mean and standard deviation, you indirectly introduce information from these sets into your training set. This can lead to overfitting, where your model might perform well on the validation/test set not because it generalizes well, but because it had access to information from these sets during normalization.
- *Simulate a Real-World Scenario :* In a real-world scenario, when you receive new data (e.g., a new image), you don't want to recalculate the mean and standard deviation every time. You want to use a consistent normalization method that can be applied to any new data. By using only the mean and standard deviation from the training set, you simulate this scenario.

- *Stability of Estimates :* The training set is typically larger than the validation or test sets. Therefore, statistics calculated from the training set (like mean and standard deviation) are usually more stable and less prone to random fluctuations than if they were calculated on smaller sets.

In conclusion, calculating the mean and standard deviation only on the training set and using these values to normalize all sets ensures that the model is trained fairly, avoids overfitting, and simulates a real-world application where normalization needs to be consistent for all new data.

Bonus question 21 - ZCA Whitening

Given a dataset X where each row is a sample and each column is a feature (e.g., a pixel in an image), the ZCA Whitening process is as follows:

1. **Mean subtraction** : Subtract the mean of the data to center the data around the origin.
$$\bar{x} = \frac{1}{m}\sum_{i=1}^{m} x_i$$
$$x_i \leftarrow x_i - \bar{x}$$

2. **Compute the covariance matrix** :
$$\Sigma = \frac{1}{m} X^T X$$

3. **Perform eigen-decomposition** : Decompose the covariance matrix into its eigenvectors and eigenvalues.
$$\Sigma = Q \Lambda Q^T$$
where Q is the matrix of eigenvectors and Λ is a diagonal matrix of eigenvalues.

4. **Compute the whitening matrix** : The whitening transformation is then given by:
$$W = Q \Lambda^{-\frac{1}{2}} Q^T$$

5. **Apply the whitening transformation** :
$$X_{\text{white}} = XW$$

After the above steps, X_{white} is the ZCA-whitened data.

Comparison of Classic vs ZCA Normalization

Training Performance

- **Training Time** :
  - Classic normalization: 14s
  - ZCA normalization: 81s

  ZCA normalization takes significantly longer (almost 6 times) during training, likely due to the overhead from the eigen-decomposition involved in the ZCA process.

- **Training Loss** :
  - Classic normalization: 0.0004
  - ZCA normalization : 0.0006

  Both methods achieve a very low training loss. The difference in the small loss values may not be practically significant.

- **Training Precision** :
  - Classic normalization : Prec@1 100.00%, Prec@5 100.00%
  - ZCA normalization : Prec@1 100.00%, Prec@5 100.00%

  Both normalization techniques achieve perfect precision on the training data.

Testing Performance

- **Testing Time** :
  - Classic normalization : 3s
  - ZCA normalization : 16s

  ZCA again takes longer in the testing phase, possibly due to transformation overhead.

- **Testing Loss** :
  - Classic normalization: 1.7286
  - ZCA normalization : 1.5942

  The ZCA normalization results in a lower testing loss, suggesting better generalization to the test data.

- **Testing Precision**:
  - Classic normalization: Prec@1 76.97%, Prec@5 98.14%
  - ZCA normalization: Prec@1 77.97%, Prec@5 98.53%

  Models trained with ZCA normalized data achieve a slightly better prediction accuracy on the test set.

Conclusion

While ZCA normalization offers slightly improved test performance in terms of loss and precision, it incurs a significantly longer computational time both in training and testing. The choice between the two should weigh the trade-offs of computational efficiency against performance gains. For critical applications where a slight boost in accuracy is vital and computational time isn't a constraint, ZCA might be the preferred choice. For rapid experiments and prototyping, classic normalization offers a more time-efficient alternative.

2.3.2 Increase in the number of training examples by data increase

Question 22

Experimental Results with Data Augmentation

- **Training Performance**:
  - Training Time: 18s
  - Average Loss: 0.2208
  - Precision@1: 92.31%
  - Precision@5: 99.92%
- **Testing Performance**:
  - Testing Time: 2s
  - Average Loss: 0.6978
  - Precision@1: 80.80%
  - Precision@5: 98.56%

Comparison to Model without Data Augmentation

When compared to the model trained without data augmentation, we can make the following observations:

- **Training Performance**:
  - The training time increased from 14s to 18s, suggesting that the augmented datasets require more processing.
  - The average loss rose from a near-perfect 0.0004 to 0.2208, indicating a less overfitting tendency when using data augmentation.
  - Precision@1 decreased from 100.00% to 92.31% and Precision@5 remained nearly perfect at 99.92%.
- **Testing Performance**:
  - Testing time decreased slightly from 3s to 2s.
  - The average loss decreased significantly from 1.7286 to 0.6978, pointing to better generalization with data augmentation.
  - Both Precision@1 and Precision@5 metrics improved with data augmentation, increasing from 76.97% to 80.80% and from 98.14% to 98.56% respectively.

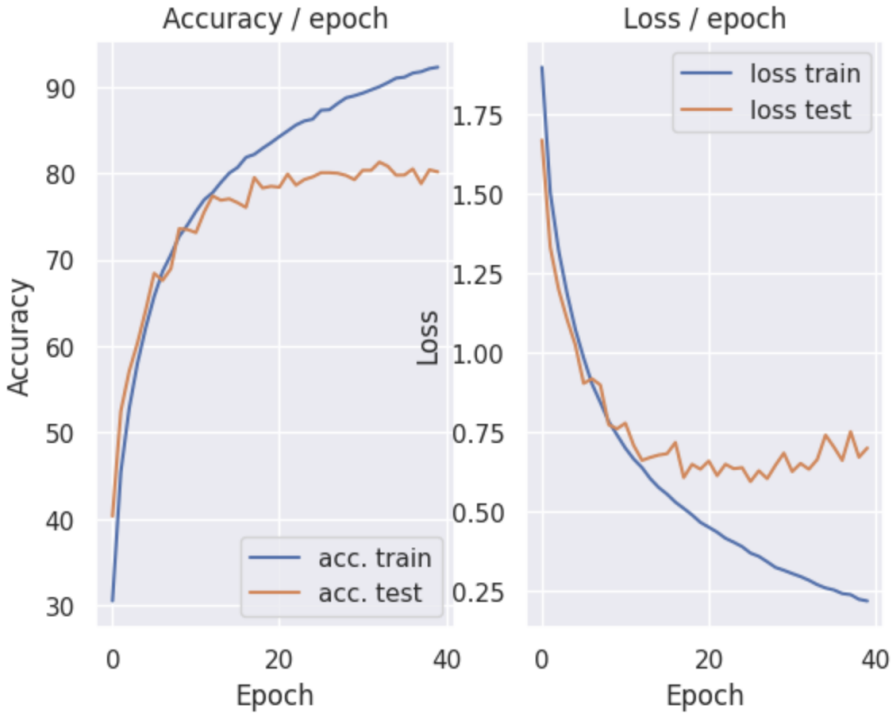

Figure 2.9: Results - Data augmentation

Conclusion

The utilization of data augmentation appears to mitigate overfitting, as seen by the higher training loss and slightly reduced training precision. Notably, the model demonstrates superior generalization on the test data when trained with data augmentation. This underscores the utility of data augmentation in enhancing model robustness and its effectiveness in improving model performance on unseen data.

Question 23

Horizontal flipping is beneficial for general object recognition and symmetric objects. However, it's not suitable when orientation holds significance, such as in traffic signs, text recognition, or cultural contexts. Always consider the dataset's nature and problem context before applying this augmentation.

Question 24

Increasing data through transformations, such as horizontal flipping, random crops, rotations, and other augmentations, is a widely adopted strategy for enhancing dataset diversity and bolstering model generalization. Despite its advantages, this approach is not devoid of limitations.

Artificial Representations: Not all augmentations yield representations encountered in real-world scenarios. For instance, specific rotations or flips might generate images unobserved in the actual application, possibly leading to deceptive training signals.

Overfitting to Augmentations: Over-reliance on certain augmentations might cause a model to overfit to these specific variations rather than extracting genuine patterns intrinsic to the data.

Limited Diversity: Although transformations can yield variations of the present data, they do not genuinely add new, unseen scenarios. Consequently, the model remains constrained by the original dataset's scope.

Contextual Inaccuracies: Certain augmentations might alter an image's inherent context or meaning. For instance, horizontally flipping an image containing a "No Left Turn" sign might inadvertently change its intent.

Computational Overhead: Especially intricate augmentations can elongate training durations due to the additional computational requirements they introduce.

Diminishing Returns: After a certain augmentation threshold, the incremental performance benefits might plateau or even retract.

Conclusion

In summation, while transformations in data augmentation can be advantageous, it's imperative to judiciously employ them, taking into account the specific characteristics of the dataset and the problem domain.

Bonus question 25

Check out the results and comments on the notebook provided with that report

2.3.3 Variants on the optimization algorithm

Question 26

Two experiments were conducted to evaluate the impact of using an optimizer alongside data augmentation on model performance.

Experiment: Data Augmentation with Optimizer

- Training Time: 20s
- Average Loss (Train): 0.2225
- Precision@1 (Train): 92.28%
- Precision@5 (Train): 99.86%
- Average Loss (Test): 0.6119
- Precision@1 (Test): 82.09%
- Precision@5 (Test): 98.95%

Discussion

The optimizer introduces a minor computational overhead, evident in the increased training time. However, the model's generalization, as observed in the test metrics, benefits from its inclusion. Notably, there's a reduction in test loss and a slight enhancement in Precision@1 with the optimizer. The findings highlight the optimizer's role in potentially boosting performance on unseen data, despite similar training metrics in both experiments.

Question 27

Here's why this method can improve learning:

- Optimized Convergence: Optimizers improve the speed and quality of model convergence. Advanced optimizers adjust the learning rate during training or consider the gradient's history, ensuring that the model converges to a good solution faster and more reliably.
- Escape from Poor Local Minima: Some optimization techniques introduce momentum or adaptive learning rates, which can help the model escape from poor local minima or saddle points in the loss landscape.
- Effective Utilization of Compute Resources: An optimizer can make the training process more computationally efficient by requiring fewer epochs to reach a similar or even better performance level, thus making better use of available compute resources.

Bonus question 28 - Variants of Stochastic Gradient Descent (SGD)

1. **Momentum SGD:** Enhances vanilla SGD by introducing a momentum term, which is a fraction of the previous update to help accelerate gradients in the right directions.
2. **Nesterov Accelerated Gradient (NAG):** A modification of momentum where the gradient is calculated after the current velocity is applied, offering a more accurate estimation of future gradients.
3. **Adaptive Gradient Algorithm (AdaGrad):** Adjusts the learning rate of each parameter based on the historical squared gradient for that parameter.

Learning Rate Scheduling Strategies

1. **Step Decay:** Reduces the learning rate after a fixed number of epochs.
2. **Exponential Decay:** Decreases the learning rate at each epoch (or batch) following an exponential law.
3. **Time-based Decay:** Modifies the learning rate according to a pre-defined function which decreases over time.

2.3.4 Regularization of the network by dropout

Question 29

Training Phase:

- Total time: 20 seconds
- Average loss: 0.3253
- Average Precision@1: 88.56%
- Average Precision@5: 99.67%

Testing Phase:

- Total time: 2 seconds
- Average loss: 0.5146
- Average Precision@1: 83.26%
- Average Precision@5: 99.15%

Analysis

The model with dropout, while having a slightly higher training loss and lower training precision, generalizes better to the test data, which is evident from the lower testing loss and higher Precision@1 during testing.
Dropout is acting as a regularization technique in this case, reducing the overfitting and improving generalization to unseen data.
In conclusion, while the model without dropout performs better on the training data, the model with dropout generalizes better to the test data. This demonstrates the value of dropout as a regularization technique to combat overfitting.

Question 30 - Regularization in Machine Learning

Regularization is a technique used in statistical modeling and machine learning to prevent **overfitting**. Overfitting occurs when a model becomes too complex, capturing the noise in the training data. Noise represents random variations or errors in the data rather than the true underlying relationships.

Regularization prevents overfitting by adding a penalty to the loss function, making it costly for the model to have overly complex or extreme parameter values. This encourages the model to have simpler or smaller coefficients, leading to a more generalized solution.

Regularization techniques play a crucial role in building models that generalize well to new data, ensuring that they are not overly tailored to the training data at hand.

Example: L2 Regularization

L2 regularization, also known as Ridge Regression, adds a penalty to the loss function based on the magnitude of the coefficients. The modified loss function with L2 regularization is given by:

$$\text{Loss}_{L2} = \text{Original Loss} + \lambda \sum_{i=1}^{n} \theta_i^2$$

Where:

- Original Loss is the loss without regularization (e.g., mean squared error for regression).
- λ is the regularization strength, a hyperparameter to be chosen. A higher value of λ results in more regularization.
- θ_i represents the model's coefficients.
- The sum runs over all coefficients in the model.

This penalty discourages large coefficients, thereby reducing the model's complexity.

Question 31 - Interpretation of the effects of Dropout on Neural Networks

1. **Regularization Effect**: Dropout acts as a regularizer, preventing the model from fitting too closely to the training data and thus helping to avoid overfitting. By randomly deactivating certain neurons during training, the model is discouraged from relying too heavily on any single neuron, promoting better generalization.
2. **Noise Injection**: Dropout introduces randomness into the training process. Each time a batch is processed, different neurons are "dropped out" or deactivated, effectively injecting noise into the process. This can help the network become more robust and prevent overfitting.
3. **Thinning the Network**: By randomly deactivating neurons, the effective capacity of the network is reduced during training, making it behave like a thinner or less complex version of itself. This ensures the model doesn't become overly complex relative to the problem it's solving.
4. **Computational Advantages**: With dropout, fewer neurons are active during training, potentially speeding up the computations in both the forward and backward passes. However, this can be offset by the need for more epochs due to the introduced noise.

5. **Increasing Training Time**: The noise introduced by dropout can slow down the network's convergence rate, potentially requiring more epochs for the model to achieve similar training accuracy as one without dropout.

Question 32 - Influence of Dropout Rate on Neural Networks

The primary hyperparameter of the dropout layer is the dropout rate, which determines the fraction of input units to set to zero during training. Its influence includes:

1. **Network Overfitting**:
   - *Low Dropout Rate*: Might not provide strong regularization, leading to potential overfitting.
   - *High Dropout Rate*: Prevents overfitting, but can lead to underfitting if too high.
2. **Training Stability**:
   - *Low Dropout Rate*: Offers a more stable training process.
   - *High Dropout Rate*: Can lead to less stable training due to dramatic architectural changes between batches.
3. **Training Duration**:
   - *Low Dropout Rate*: Faster convergence due to less noise during training.
   - *High Dropout Rate*: Slower convergence because of increased noise, potentially requiring more epochs.

Question 33 - Behavior of Dropout Layer: Training vs. Test

The dropout layer exhibits distinct behaviors during training and test phases:

1. **During Training**:
   - Neurons are randomly "dropped out" or set to zero based on the specified dropout rate in each forward pass. These neurons do not participate in the forward and subsequent backward passes.
2. **During Test**:
   - Dropout is **not** applied. All neurons are active, and none are set to zero because of dropout. The model leverages all learned features for predictions.

- To account for the increased neuron activity compared to training, the neuron outputs (or weights, depending on implementation) are scaled by a factor equivalent to the dropout rate. For instance, with a dropout rate of 0.5 during training, neuron outputs might be scaled by a factor of 0.5 during inference to maintain expected output consistency.

2.3.5 Use of batch normalization

Question 34 - Experimental Results with Batch Normalization

	Total Time	**Avg Loss**	**Avg Prec@1 (%)**	**Avg Prec@5 (%)**
Train	19s	0.3081	89.34	99.74
Test	2s	0.4519	84.45	99.32

Table 2.1: Performance metrics of a model with batch normalization.

From the numbers, it's clear that incorporating batch normalization not only improved the model's learning but also its ability to predict new, unseen data accurately. This means it was better at generalizing what it had learned.