Deep Learning Practical Work
Basics on deep learning for vision

Aymeric Delefosse & Charles Vin

2023 – 2024

Contents

1 Introduction to neural networks

1.1 Theorical Foundation

1.1.1 Supervised dataset

1. ★ What are the train, val and test sets used for? The training dataset is utilized to train the model, while the test dataset is employed to assess the model's performance on previously unseen data. Lastly, the validation set constitutes a distinct subset of the dataset employed for the purpose of refining and optimizing the model's hyperparameters.

2. What is the influence of the number of exemples N? A larger number of examples can enhance the model's capacity to generalize and improve its robustness against noise or outliers. Conversely, a smaller number of examples can make the model susceptible to overfitting. It is important to note that increasing the dataset size can also lead to an escalation in the computational complexity of the model training process.

1.1.2 Network architecture

3. Why is it important to add activation functions between linear transformations? Otherwise, we would simply be aggregating linear functions, resulting in a linear output. Activation functions introduce non-linearity to the network, enabling the model to capture and learn more intricate patterns than those achievable through linear transformations alone.

4. ★ What are the sizes n_x, n_h, n_y in the figure 1? In practice, how are these sizes chosen?

- $n_x = 2$ represents the input size (data dimension).
- $n_h = 4$ represents the hidden layer size, selected based on the desired complexity of features to be learned in the hidden layer. An excessively large size can result in overfitting.
- $n_y = 2$ represents the output size, determined according to the number of classes in y.

5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities? $y \in \{0, 1\}$ represents the ground truth, where the values are binary (0 or 1). $\hat{y} \in [0, 1]$ represents a probability-like score assigned to each class by the model. $\hat{y}$ reflects the model's level of confidence in its predictions for each class.

6. Why use a SoftMax function as the output activation function? The reason for employing the SoftMax function is to transform $\tilde{y} \in \mathbb{R}$ into a probability distribution. While there are several methods to achieve this transformation, SoftMax is a commonly utilized choice, especially in multi-class classification problems.

7. Write the mathematical equations allowing to perform the *forward* pass of the neural network, i.e. allowing to successively produce $\tilde{h}$, h, $\tilde{y}$, $\hat{y}$, starting at x. Let W_i and b_i denote the parameters for layer i, $f_i(x) = xW_i^T + b_i$ represent the linear transformation, and $g_i(x)$ be the activation function for layer i.

Calculate the weighted sum and activation for the first hidden layer:

$$\tilde{h} = f_0(x)$$
$$h = g_0(\tilde{h})$$

Proceed to the output layer by computing the weighted sum and activation for the output layer:

$$\tilde{y} = f_1(h)$$
$$\hat{y} = g_1(\tilde{y})$$

These equations describe the sequential steps involved in the forward pass of the neural network, ultimately producing the output $\hat{y}$ based on the input x.

1.2 Loss function

8. During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function $\mathcal{L}$? Our aim is to minimize the loss function $\mathcal{L}$ to train a model effectively. To decrease cross-entropy loss, make $\hat{y}_i$ closer to 1 when y_i is 1 and closer to 0 when y_i is 0.

For the cross-entropy loss, $\hat{y}_i$ should vary in a way that makes it closer to the true target value y_i for each data point. Specifically, when $y_i = 1$, $\hat{y}_i$ should be pushed towards 1. The closer $\hat{y}_i$ is to 1, the lower the loss. Conversely, when $y_i = 0$, $\hat{y}_i$ should be pushed towards 0. The closer $\hat{y}_i$ is to 0, the lower the loss.

For squared error loss, $\hat{y}_i$ should vary in a way that makes it closer to the true target value y_i for each data point. The goal is to minimize the squared difference between $\hat{y}_i$ and y_i. This means that if $\hat{y}_i$ is greater than y_i, it should decrease, and if $\hat{y}_i$ is smaller than y_i, it should increase.

9. How are these functions better suited to classification or regression tasks? Cross-entropy loss is better suited for classification tasks for several reasons:

- Cross-entropy loss is based on the negative logarithm of the predicted probability of the true class. This logarithmic nature amplifies errors when the predicted probability deviates from 1 (for the correct class) and from 0 (for incorrect classes), depicted in Figure 1. Consequently, it effectively penalizes misclassifications, making it particularly suitable for classification tasks. However, it is imperative that $\hat{y}$ remains within the range $[0, 1]$ for this loss to be effective.
- Cross-entropy loss is often used in conjunction with the SoftMax activation function in the output layer of neural networks for multi-class classification. The SoftMax function ensures that the predicted probabilities sum to 1, aligning perfectly with the requirements of the cross-entropy loss.

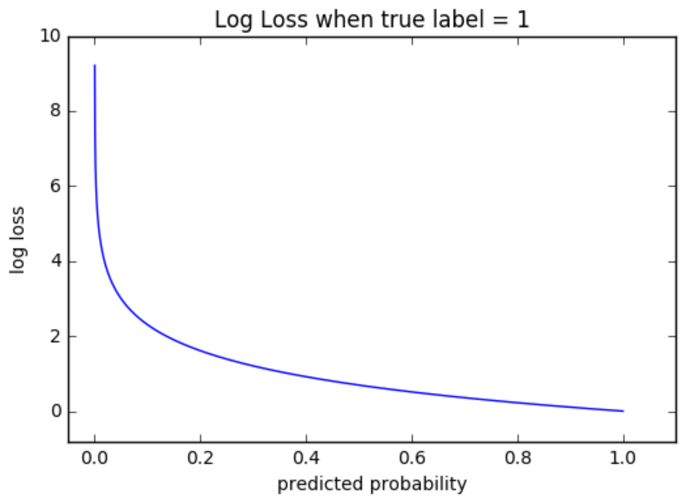

Figure 1

On the other hand, Mean Squared Error serves a different role and is better suited for regression tasks:

- MSE is ideal when dealing with $(y, \hat{y}) \in \mathbb{R}^2$ (as opposed to the bounded interval $[0, 1]$).
- MSE is advantageous due to its convexity, which simplifies optimization. However, it's important to note that it may not always be the best choice for every regression task, especially when dealing with outliers, as it can be sensitive to extreme values.

1.3 Optimization algorithm

10. What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions? Which one seems the most reasonable to use in the general case? Gradient computation becomes computationally expensive as it scales with the number of examples included. However, including more examples enhances gradient exploration with precision and stability, analogous to the exploration-exploitation trade-off in reinforcement learning.

Classic gradient descent offers stability, particularly with convex loss functions, featuring deterministic and reproducible updates. Nonetheless, its stability may lead to getting trapped in local minima when dealing with non-convex loss functions. Additionally, it is computationally intensive, especially for large datasets, as it demands evaluating the gradient with the entire dataset in each iteration.

Mini-Batch Stochastic Gradient Descent (SGD) converges faster compared to classic gradient descent by updating model parameters with small, random data subsets (mini-batches). This stochasticity aids in escaping local minima and exploring the loss landscape more efficiently. However, it may introduce noise and oscillations.

On the other hand, Online Stochastic Gradient Descent offers extremely rapid updates, processing one training example at a time, making it suitable for streaming or large-scale online learning scenarios. However, its highly noisy updates can result in erratic convergence or divergence, necessitating careful learning rate tuning.

For training deep neural networks in the general case, mini-batch stochastic gradient descent is the most reasonable choice. It strikes a balance between the computational efficiency of classic gradient descent and the noise resilience and convergence speed of online SGD.

11. ★ What is the influence of the *learning rate* η on learning? The learning rate plays a crucial role in influencing various aspects of the learning process, including convergence speed, stability, and the quality of the final model.

A higher learning rate typically results in faster convergence because it leads to larger updates in model parameters during each iteration. However, an excessively high learning rate can cause issues such as overshooting, where the optimization process diverges or oscillates around the minimum, preventing successful convergence.

Conversely, a smaller learning rate allows the optimization algorithm to take smaller steps, which can be advantageous for exploring local minima more thoroughly and escaping shallow local minima. Nonetheless, if the learning rate is too small, it may lead to slow convergence or getting stuck in local minima.

To address these challenges, various techniques like learning rate schedules (gradually reducing the learning rate over time) or adaptive learning rate methods (e.g., Adam) have been developed to automate learning rate adjustments during training. The choice of learning rate may also be influenced by the batch size used in mini-batch stochastic gradient descent, as smaller batch sizes may require smaller learning rates to maintain stability.

In practice, selecting an appropriate learning rate often involves experimentation and can be problem-specific. Techniques such as grid search or random search, in combination with cross-validation, can aid in determining an optimal learning rate for a particular task.

12. ★ Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the *loss* with respect to the parameters, using the naive approach and the *back-prop* algorithm. The naive method involves calculating the gradient of the loss function with respect to each parameter independently, without leveraging the computational advantages of knowing the derivatives' interdependencies across layers. For a network with a large number of parameters, this is computationally expensive and inefficient, essentially repeating a substantial amount of calculation needlessly. The complexity becomes substantial as the number of layers (and, as a result, the number of parameters) increases. It becomes a combinatorial problem.

Backpropagation is a far more efficient strategy due to its intrinsic use of the chain rule from calculus to "remember" past calculations. This technique essentially breaks down the full problem of calculating a derivative on a complex, multi-stage function into smaller, more manageable pieces. Then, it smartly recombines these pieces, significantly reducing computational redundancy. This methodology dramatically reduces the number of computations, particularly for networks with a large number of interconnected layers and parameters.

13. What criteria must the network architecture meet to allow such an optimization procedure? For the backpropagation algorithm to be applicable, several criteria must be met. First of all, each layer function, including the activation functions and the loss function, must be differentiable with respect to their parameters. This is crucial for calculating gradients during the training process. Furthermore, as backpropagation relies on the sequential flow of information from the input layer to the output layer, the network architecture should possess a feedforward, sequential structure without loops or recurrent connections. Loops or recurrent connections can introduce complications in gradient calculations.

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

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

Let us define the cross-entropy loss as $l(y, \hat{y}) = -\sum_{i=1}^N y_i \log \hat{y}_i$, where N represents the total number of examples. The SoftMax function for a vector $\tilde{y} \in \mathbb{R}^D$ is defined as $\text{SoftMax}(\tilde{y}_i) = \frac{e^{\tilde{y}_i}}{\sum_{j=1}^D e^{\tilde{y}_j}}$. Notably, the output of the SoftMax function serves as the input for the cross-entropy loss, denoted as $\hat{y}_i = \text{SoftMax}(\tilde{y}_i)$. Let us substitute this value into the expression.

$$\begin{aligned}
l(y, \hat{y}) &= -\sum_{i=1}^N y_i \log \hat{y}_i \\
&= -\sum_{i=1}^N y_i \log \text{SoftMax}(\tilde{y}_i) \\
&= -\sum_{i=1}^N y_i \log \frac{e^{\tilde{y}_i}}{\sum_{j=1}^D e^{\tilde{y}_j}} \\
&= -\sum_{i=1}^N y_i \left[\log e^{\tilde{y}_i} - \log \sum_{j=1}^D e^{\tilde{y}_j} \right] \\
&= -\sum_{i=1}^N y_i \tilde{y}_i - y_i \log \sum_{j=1}^D e^{\tilde{y}_j} \\
&= -\sum_{i=1}^N y_i \tilde{y}_i + \sum_{i=1}^N y_i \log(\sum_j^D e^{\tilde{y}_j}) \\
&= -\sum_{i=1}^N y_i \tilde{y}_i + \log(\sum_j^D e^{\tilde{y}_j}) \sum_{i=1}^N y_i
\end{aligned}$$

Since y_i is One Hot encoded we know that $\sum_i y_i = 1$. Thus, we have the final expression for the cross-entropy loss:

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

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

$$\begin{aligned}
\frac{\partial l}{\partial \tilde{y}_i} &= -y_i + \frac{\frac{\partial}{\partial \tilde{y}_i}(\sum_{j=1}^N e^{\tilde{y}_j})}{\sum_{j=1}^N e^{\tilde{y}_j}} \\
&= -y_i + \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} \\
&= -y_i + \text{SoftMax}(\tilde{y})_i
\end{aligned}$$

$$\nabla_{\tilde{y}} l = \begin{pmatrix} \frac{\partial l}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}} \end{pmatrix} = \begin{pmatrix} \text{SoftMax}(\tilde{y})_1 - y_1 \\ \vdots \\ \text{SoftMax}(\tilde{y})_{n_y} - y_{n_y} \end{pmatrix} = \hat{y} - y$$

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

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$$

This can be expressed as a matrix:

$$\nabla_{W_y} l = \begin{pmatrix} \frac{\partial l}{\partial W_{y,1,1}} & \cdots & \frac{\partial l}{\partial W_{y,1,n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial W_{y,n_y,1}} & \cdots & \frac{\partial l}{\partial W_{y,n_y,n_h}} \end{pmatrix}$$

First, let's compute $\tilde{y}_k$

$$\begin{aligned} \tilde{y} &= hW_y^T + b^y \\ &= \begin{pmatrix} h_1 & \dots & h_{n_h} \end{pmatrix} * \begin{pmatrix} W_{1,1} & \dots & W_{1,n_y} \\ \vdots & \ddots & \vdots \\ W_{n_h,1} & \dots & W_{n_h,n_y} \end{pmatrix} + \begin{pmatrix} b_1^y & \dots & b_{n_y}^y \end{pmatrix} \\ &= \begin{pmatrix} \sum_{j=1}^{n_h} W_{1,j}^y h_j + b_1^y & \sum_{j=1}^{n_h} W_{2,j}^y h_j + b_2^y & \dots & \sum_{j=1}^{n_h} W_{n_h,j}^y h_{k,j} + b_{n_h}^y \end{pmatrix} \qquad \in \mathbb{R}^{1 \times n_h} \\ \tilde{y}_k &= \sum_{j=1}^{n_h} W_{k,j}^y h_j + b_k^y, k \in [1, n_h] \end{aligned}$$

With this expression, we can now proceed with the calculation of the partial derivative of $\tilde{y}_k$ with respect to W_{ij}^y:

$$\frac{\partial \tilde{y}_k}{\partial W_{ij}^y} = \begin{cases} h_j & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}$$

Now, we need to find $\frac{\partial l}{\partial \tilde{y}_k}$. From the previous question, we have:

$$\frac{\partial l}{\partial \tilde{y}_k} = -y_k + \text{SoftMax}(\tilde{y})_k = \hat{y}_k - y_k$$

Now, combining these results:

$$\frac{\partial l}{\partial W_{i,j}^y} = \sum_{k=1}^{n_h} \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{i,j}^y} = \frac{\partial l}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial W_{i,j}^y} = (\hat{y}_i - y_i) h_j = (\nabla_{W_y} l)_{i,j}$$

So, the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} l$ is given by $\nabla_{\tilde{y}}^T h$.

$$\begin{aligned} \nabla_{W_y} l &= \begin{pmatrix} \frac{\partial l}{\partial W_{1,1}^y} & \cdots & \frac{\partial l}{\partial W_{1,n_h}^y} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial W_{n_y,1}^y} & \cdots & \frac{\partial l}{\partial W_{n_y,n_h}^y} \end{pmatrix} \\ &= \begin{pmatrix} (\hat{y}_1 - y_1) h_1 & \dots & (\hat{y}_1 - y_1) h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y}) h_1 & \dots & (\hat{y}_{n_y} - y_{n_y}) h_{n_h} \end{pmatrix} \\ &= \begin{pmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{pmatrix} \begin{pmatrix} h_1 & h_2 & \dots & h_{n_h} \end{pmatrix} \\ &= \nabla_{\tilde{y}}^T h \end{aligned}$$

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

1. The gradient of the loss with respect to $\tilde{h}$ can be computed using the chain rule:

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i}$$

Let's compute those two terms:

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

Having $\tilde{y}_i = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y$ in mind and recovering $\frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i = \delta_i^y$ from past question:

$$\frac{\partial l}{\partial h_k} = \sum_{j=1} \frac{\partial l}{\partial \tilde{y}_j} \frac{\partial \tilde{y}_j}{\partial h_k} = \sum_{j=1} \delta_j^y W_{j,k}^y$$

Finally,

$$\begin{aligned} \frac{\partial l}{\partial \tilde{h}_i} &= \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} \\ &= \sum_k (\sum_j \delta_j^y W_{j,k}^y) \times \begin{cases} 1 - h_i^2 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} \\ &= (1 - h_i^2)(\sum_j \delta_j^y W_{j,i}^y) \\ &= \delta_i^h \end{aligned}$$

So, the gradient $\nabla_{\tilde{h}} l$ is a vector with elements δ_i^h:

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

2. $\nabla_{W^h} l$ is a matrix composed of elements

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

From the previous question, we already have $\frac{\partial l}{\partial h_k} = (1 - h_k^2)(\sum_j \delta_j^y W_{j,k}^y) = \delta_k^h$. So let's compute the other term $\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h}$ from $\tilde{h}_k = \sum_{j=1}^{n_x} W_{k,j}^h x_j + b_k^h$

$$\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h} = \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

So:

$$\begin{aligned} \frac{\partial l}{\partial W_{i,j}^h} &= \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{i,j}^h} \\ &= \sum_k \delta_k^h \times \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} \\ &= \delta_i^h x_j \\ \nabla_W^h &= \nabla_{\tilde{h}}^T l * x \end{aligned}$$

3. Last but not least:

$$\frac{\partial l}{\partial b_i^j} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_i^h} = \sum_k \delta_k^h * \begin{cases} 1 & \text{if } k = i \\ 0 & \text{else} \end{cases} = \delta_i^h$$

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

2 Introduction to convolutional networks

2.1 Questions

1. Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size $x \times y \times z$ what will be the output size? How much weight is there to learn? How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size? Let's note $x_{\text{out}}, y_{\text{out}}, z_{\text{out}}$ our output size. Then, given a convolution filter of padding p, stride s and kernel size k:

$$x_{\text{out}} = \frac{x + 2 \times p - k}{s} + 1 \tag{1}$$
$$y_{\text{out}} = \frac{y + 2 \times p - k}{s} + 1 \tag{2}$$
$$z_{\text{out}} = \frac{z + 2 \times p - k}{s} + 1 \tag{3}$$

We have a single convolution filter. Given our kernel of size $k \times k$, we thus have $1 \times k \times k = k^2$ weights to learn. In a fully-connected layer, we would have $(x \times y \times z \times k^2)$ weights to learn.

2. ★ What are the advantages of convolution over fully-connected layers? What is its main limit? Using fully-connected layers with images involves a substantial number of parameters, one for each pixel of the image. Fully-connected layers are also sensitive to variations in images, including simple translations.

On the other hand, convolutional layers address these issues. By utilizing convolution with a shared set of weights (for filters/kernels) across the entire input, they enable the learning and detection of local spatial patterns and features at various locations. Convolutional layers also possess the ability to hierarchically combine features. Lower layers capture low-level features such as edges and textures, while higher layers capture more intricate features and representations of objects.

However, convolutional layers have limitations in terms of global context. They primarily focus on local patterns and may struggle to comprehend relationships between distant parts of the input, which can be crucial in certain applications (medical imaging, autonomous vehicles...).

3. ★ Why do we use spatial pooling? Spatial pooling allows for increased invariance, particularly in the context of translation. By summarizing a local region with a pooled value, the precise feature location becomes less critical. This constitutes a form of regularization that prioritizes the most pertinent information.

Pooling also facilitates dimensionality reduction by decreasing the spatial dimensions of the resulting feature map. This reduction in dimensionality significantly lowers the computational cost of subsequent layers.

Max and average pooling layers are the most commonly employed pooling techniques. Max pooling aids in achieving translation invariance and reducing spatial dimensions. It effectively preserves the most essential features in the input data. On the other hand, average pooling also reduces spatial dimensions and contributes to achieving translation invariance. It can exhibit greater robustness to outliers in the input data when compared to max pooling.

4. ★ Suppose we try to compute the output of a classical convolutional network for an input image larger than the initially planned size. Can we (without modifying the image) use all or part of the layers of the network on this image? A convolution layer in a convolutional network is size-agnostic regarding its input: it simply performs convolutions across the entire span of the input image and outputs a correspondingly altered image. Sequentially, when convolution and pooling layers are stacked together, the resulting output size is purely a function of the input size. To rephrase, the input size of the image is not a critical hyperparameter for the convolution layer.

Within the conventional architecture of a convolutional network, the output derived from successive convolutional and pooling layers is typically "flattened" to form the input for a subsequent fully-connected layer. The size of this flattened vector is contingent on the size of the image output from the preceding convolution layers.

The challenge arises when we involve fully connected layers, as their initialization requires a priori knowledge of their input size to properly set up parameters such as weights. Consequently, when an input image is larger than initially planned, the network can operate normally up to the point of the fully connected layers, which anticipate input of a predetermined, fixed size.

5. Show that we can analyze fully-connected layers as particular convolutions. In a fully-connected layer, each pixel of the flattened image is associated with one weight. Both the weights and the flattened image are vectors. However, if we rearrange the weights into a matrix that matches the shape of the image, we can perform element-wise multiplication between this matrix and the image, which is akin to a one slide convolution!

This 1×1 convolution employs a stride equal to the width of the input image, ensuring that the filter remains fixed and does not slide to different positions. Furthermore, the filter size is exactly the same as the image. The number of filters used in this convolution is equivalent to the number of neurons in the fully-connected layer.

6. Suppose that we therefore replace fully-connected by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest? Not sure : For me, as with the fully connected layer, we still have to know the input size to fix the filter size and the stride ahead. So it will still not work.

7. We call the receptive field of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers? Can you imagine what happens to the deeper layers? How to interpret it?

2.2 Training *from scratch* of the model

2.2.1 Network architecture

8. For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed? We want $x = x_{out}, y = y_{out}$: let's use our equations 1, 2, 3 to find the values of p and s.

$$p = \frac{(s-1)x + k - s}{2}$$

Knowing $k = 5$ and that our images are of size 32×32, we can pick any $p = \frac{31s-27}{2}$. Because we generally want to use every pixel (and not padding pixels) so we pick $s = 1$ so $p = 2$ to minimise them both.

9. For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed? A pooling layer acts like a convolution: let's proceed as in question 8. This time we want $x_{out} = \frac{1}{2}x, y_{out} = \frac{1}{2}y$.

$$p = \frac{(\frac{1}{2}s - 1)x + k - s}{2}$$

Knowing $k = 2$ and that our images are of size 32×32, we can pick any $p = \frac{15s-30}{2}$. We generally want to use every pixel (and not padding pixels) so we pick $s = 2$ so $p = 0$ to minimise them both.

10. ★ For each layer, indicate the output size and the number of weights to learn. Comment on this repartition. To simplify the presentation, we have provided a plot of the CNN architecture in Figure 2. In this architecture, we begin with a 3-channel image measuring 32 by 32 pixels. As mentioned in questions 8 and 9, convolutional layers maintain the same spatial dimensions in the outputs as in the inputs, while pooling layers reduce the spatial dimensions by a factor of two. The following is a breakdown of the number of parameters:

- conv1: 32 filters of 5 by 5 for 3 channels $\rightarrow 32 \times 5 \times 5 \times 3 = 2,400$ parameters
- conv2: 64 filters of 5 by 5 for 32 channels $\rightarrow 64 \times 5 \times 5 \times 32 = 51,200$ parameters
- conv3: 64 filters of 5 by 5 for 64 channels $\rightarrow 64 \times 5 \times 5 \times 64 = 102,400$ parameters
- 1000 neurons with a 1024 values as input $\rightarrow (1000 + 1) \times 1024 = 1,025,024$ parameters. The "+1" is for biases.
- 10 neurons with a 1000 values as input $\rightarrow 11 \times 1000 = 11,000$ parameters

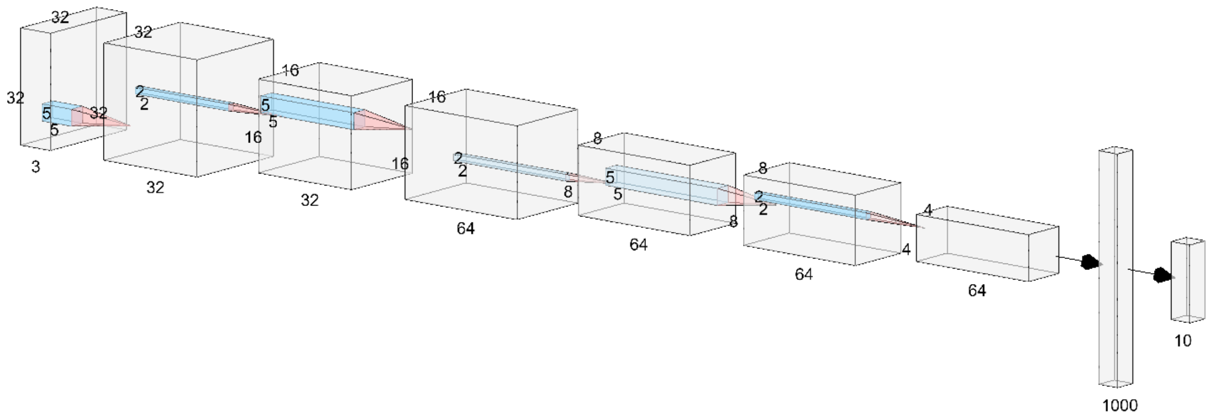

Figure 2: Network architecture

11. What is the total number of weights to learn? Compare that to the number of examples. It makes a total of $1,192,024$ parameters to train with 60,000 (6,000 images per class), which is reasonable.

12. Compare the number of parameters to learn with that of the BoW and SVM approach. De quoi il parle ??? feur

2.2.2 Network learning

14. ★ In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the the difference in data)? The difference in the code lies in the fact that we do not provide an optimizer to the `epoch()` function. Consequently, we do not train the network using test examples, and there is no backward pass to calculate and update the loss.

16. ★ What are the effects of the learning rate and of the batch-size? The learning rate and batch size are critical hyperparameters that significantly impact training performance, speed, and stability:

- A high learning rate can cause the model to overshoot the minimum, causing divergent behavior and an inability for the model to learn effectively. On the other hand, having a low learning rate can slow down the training process significantly, possibly leading to a premature convergence to suboptimal solutions. In some cases, a lower learning rate results in a more precise convergence by allowing the model to explore the parameter space thoroughly. Figure 3 illustrates the effects of different learning rates with a fixed batch size.
- Larger batch size provides a more accurate estimate of the gradient, potentially leading to more stable and consistent training steps. However, this comes with increased memory usage. Conversely, smaller batch sizes can introduce noise, making the path to convergence less direct and possibly longer in terms of epochs. Smaller batches reduce memory requirements, making them suitable for resource-constrained systems. Figure 4 illustrates the impact of various batch sizes with a fixed learning rate.

It's important to note that learning rate and batch size are often interrelated. Changes in batch size may require adjusting the learning rate to maintain training quality. This relationship arises from the need to balance the aggressiveness of updates when averaging gradients across more samples. Typically, a linear scaling rule is applied: multiplying the batch size by a factor k also multiplies the learning rate by k, while keeping other hyperparameters constant. This relationship is shown in Figure 5.

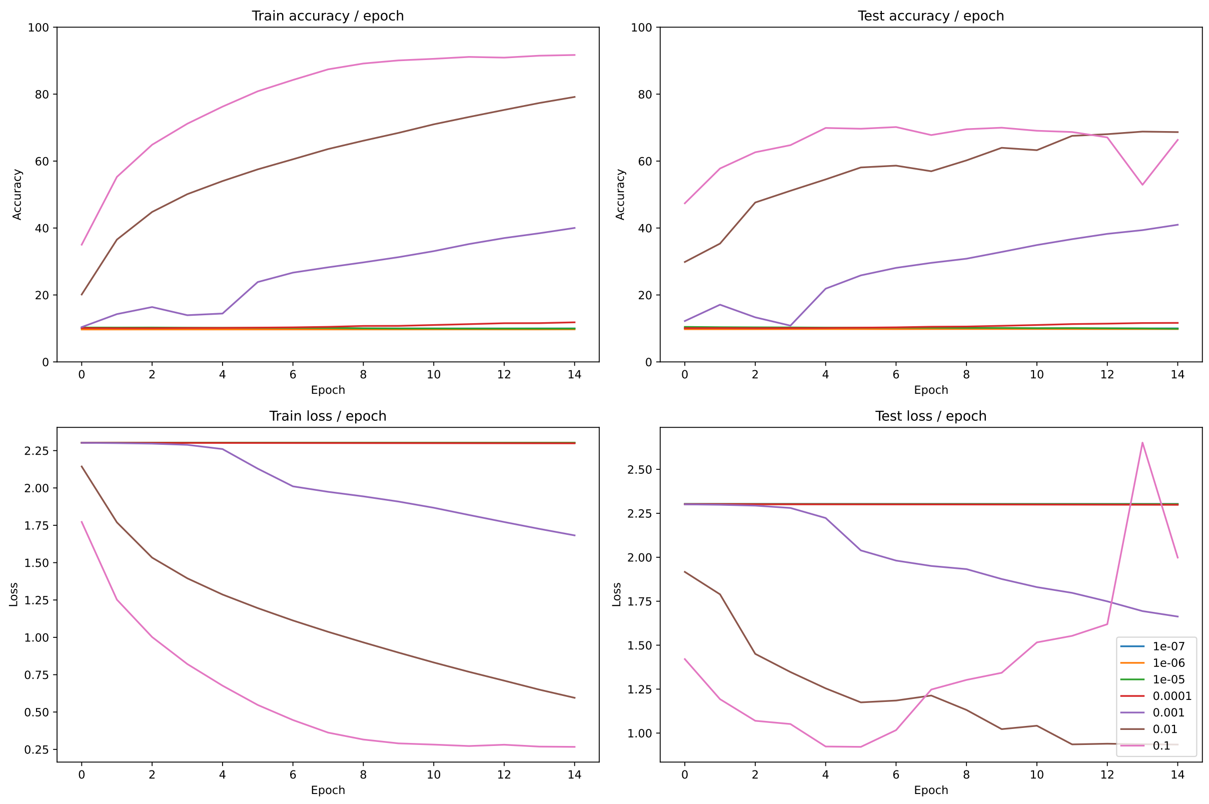

Figure 3: Influence of learning rate with a batch size fixed at 32

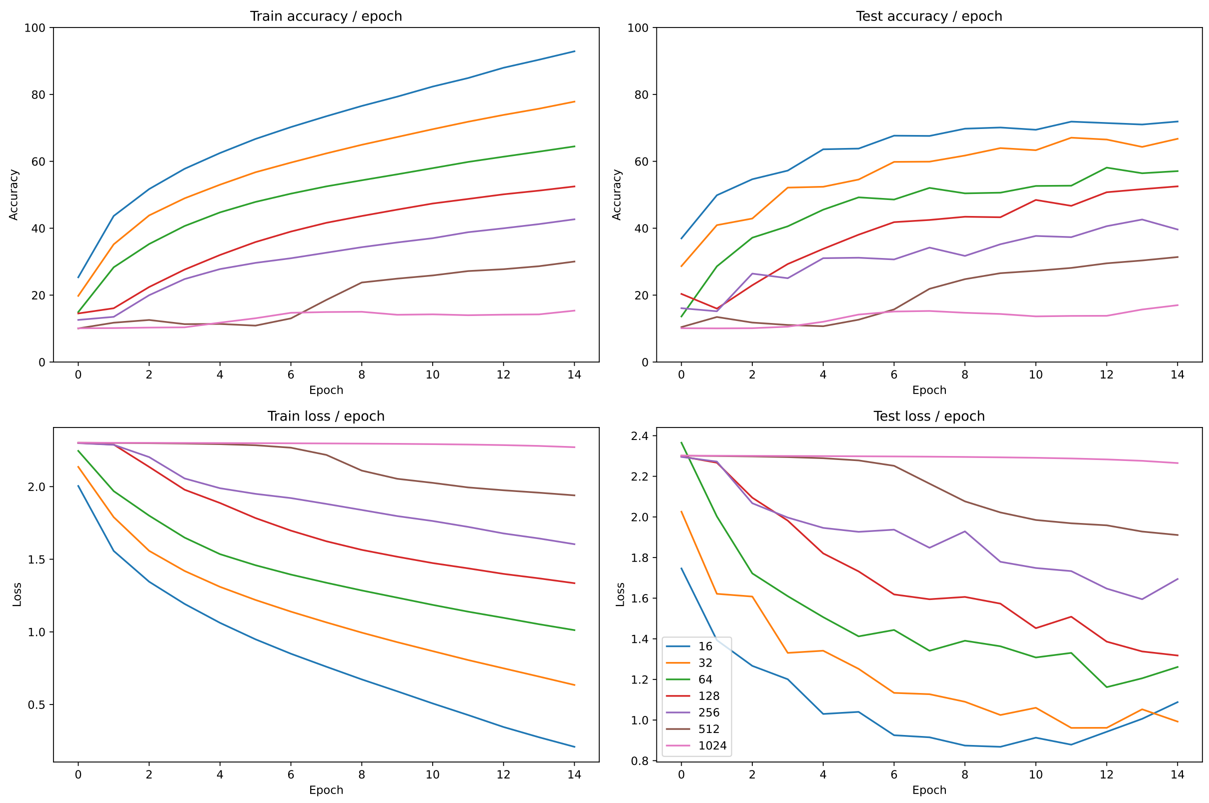

Figure 4: Influence of batch size rate with a learnig rate fixed at 0.01

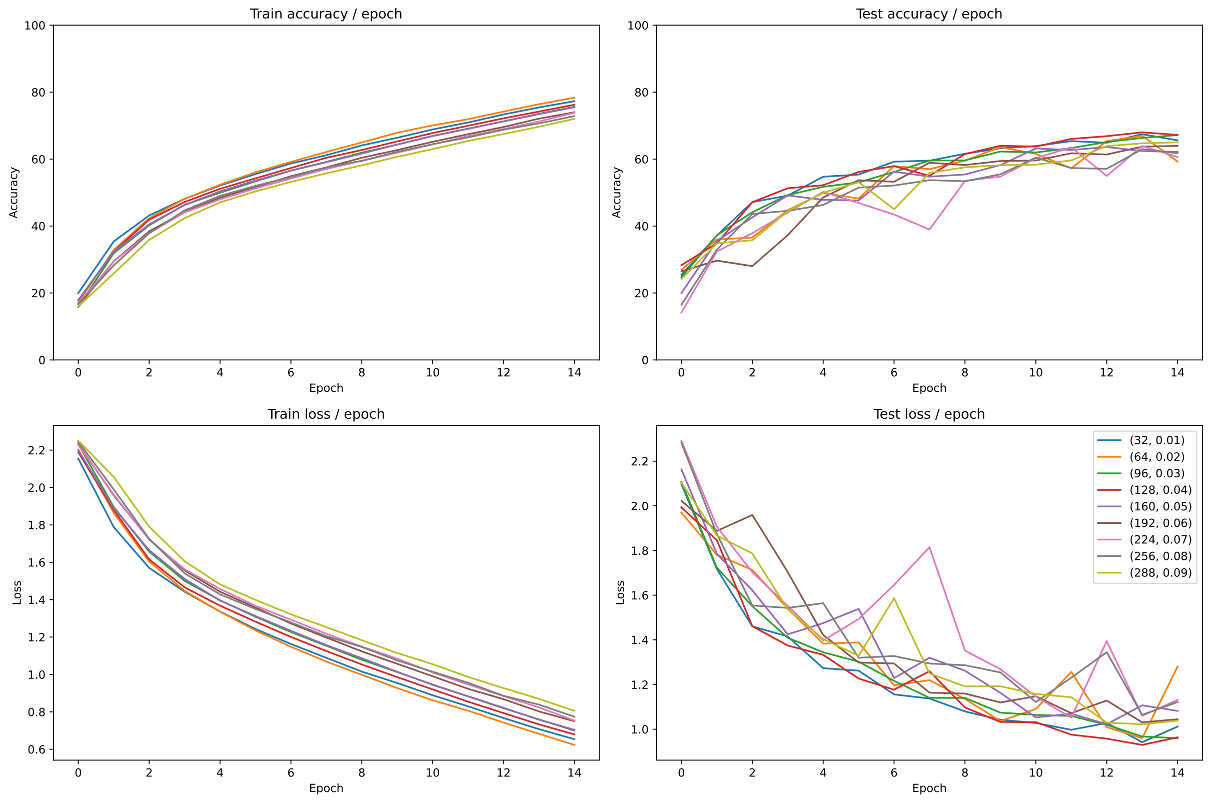

Figure 5: Relationship between batch size and learning rate

17. What is the error at the start of the first epoch, in train and test? How can you interpret this?
For the following experiments, we will compare our results with those presented in Figure 6, where a batch size of 128 and a learning rate of 0.01 were employed.

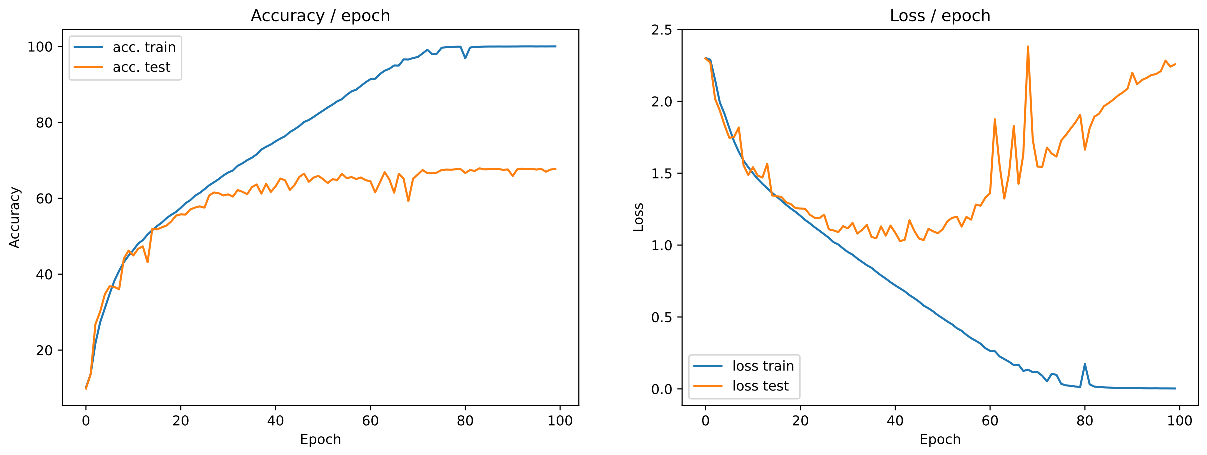

Figure 6: Accuracy and losses in train and test

At the first epoch, we have respectively in train and test an accuracy of 0.0101 and of 0.0989 for a loss of 2.3007 and 2.2976.

The accuracy on the test set is better that the accuracy on train. It's because. We must be aware that the training loss and accuracy is calculated **at every batch** so before the end of the first epoch, parameters and so result (loss and accuracy) are variating a lot. It mean that at the first batch of the first epoch we take into account the loss calculated only with *batch_size* examples.

Contrary to the test loss and accuracy that are computed at the end of the first epoch, after the model have seen all training example. Those value are more true that those from train at the first epoch

18. ★ Interpret the results. What's wrong? What is this phenomenon? We observe overfitting. We can see that starting from epoch three or four that our accuracy on test cross the curve of the accuracy of train. It mean that we are becoming better on train sample but worse on the test sample. The model can't generalize on new image.

2.3 Results improvements

2.3.1 Standardization of examples

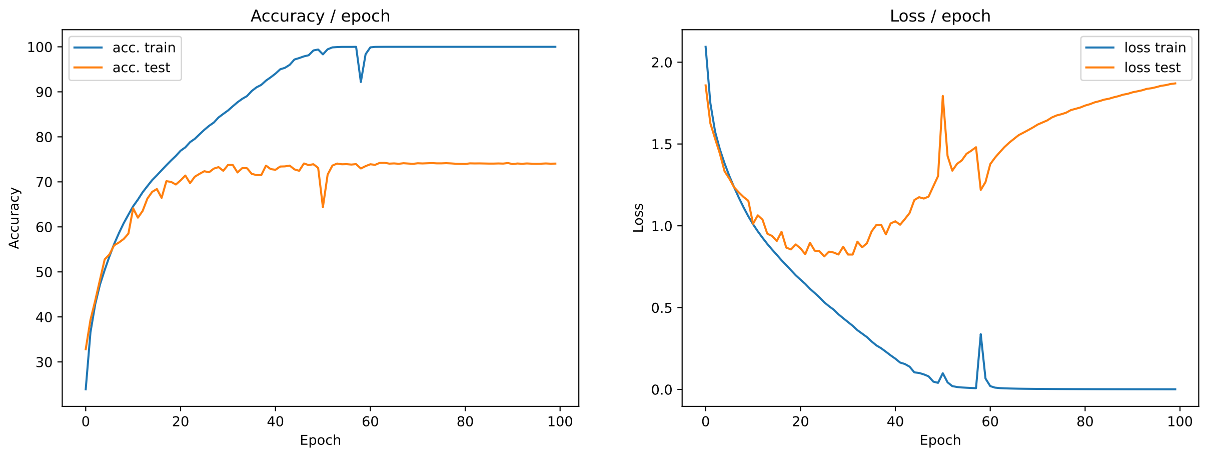

Figure 7: Accuracy and losses in train and test

19. Describe your experimental results.

20. Why only calculate the average image on the training examples and normalize the validation examples with the same image?

21. Bonus : There are other normalization schemes that can be more efficient like ZCA normalization. Try other methods, explain the differences and compare them to the one requested.

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

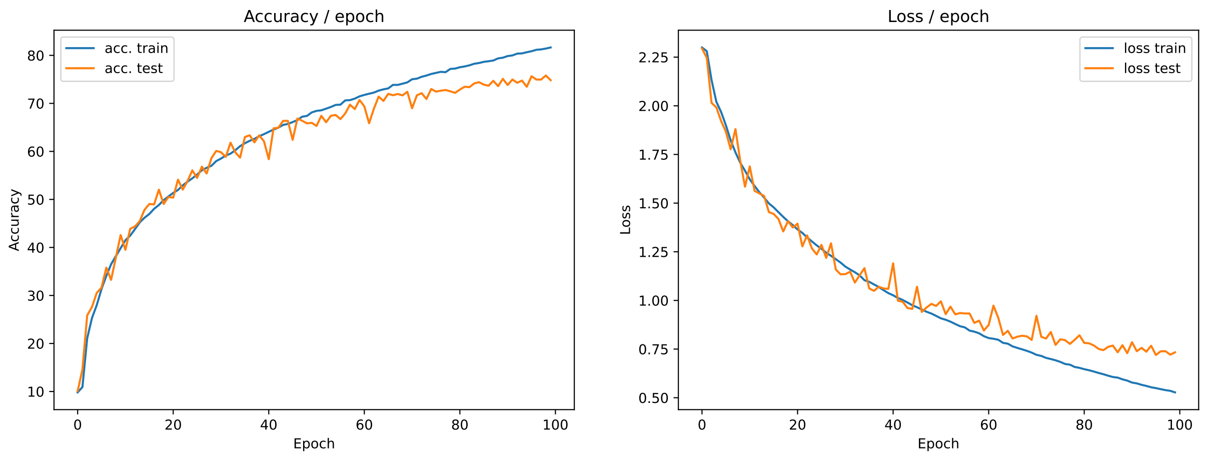

Figure 8: Accuracy and losses in train and test

22. Describe your experimental results and compare them to previouscrease results.

23. Does this horizontal symmetry approach seems usable on all types of images? In what cases can it be or not be?

24. What limits do you see in this type of data increase by transformation of the dataset?

24. What limits do you see in this type of data increase by transformation of the dataset?

25. Bonus : Other data augmentation methods are possible. Find out which ones and test some. rotation, flips, zoom in/out, change perspective, affine transformation, change color/brightness/contrast/saturation/hue/sharpnes
gaussian blur, polarize, solarize, erase pixels, permute channels, grayscale,

2.3.3 Variants of the optimization algorithm

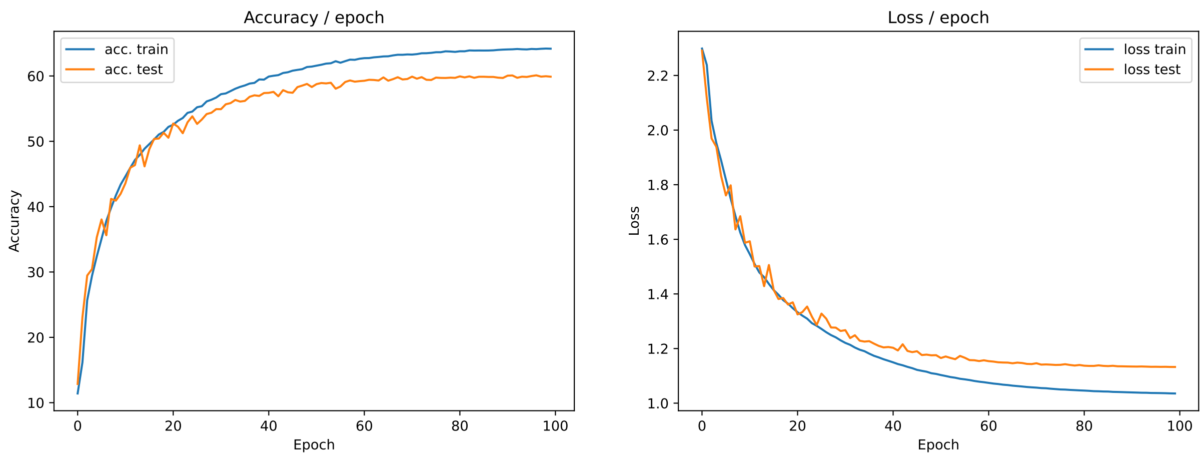

Figure 9: Using a scheduler with SGD

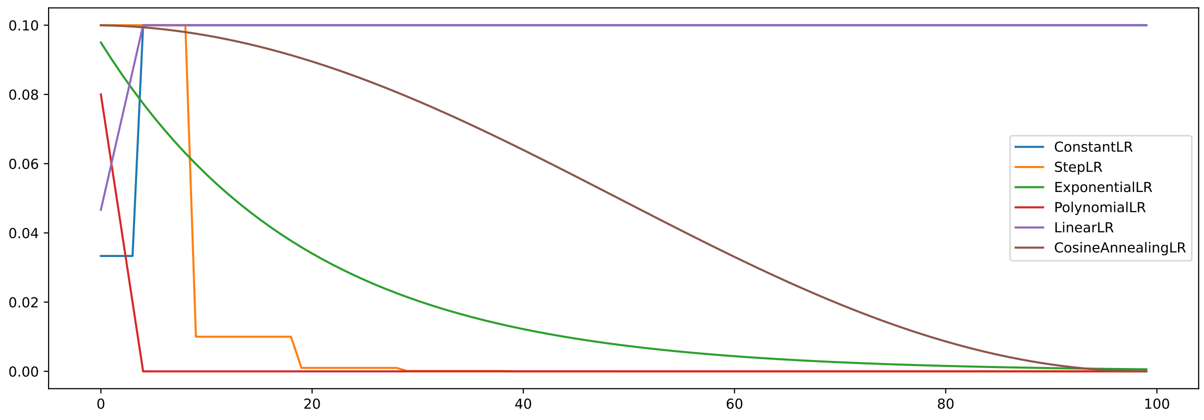

Figure 10: Comparison of schedulers on SGD

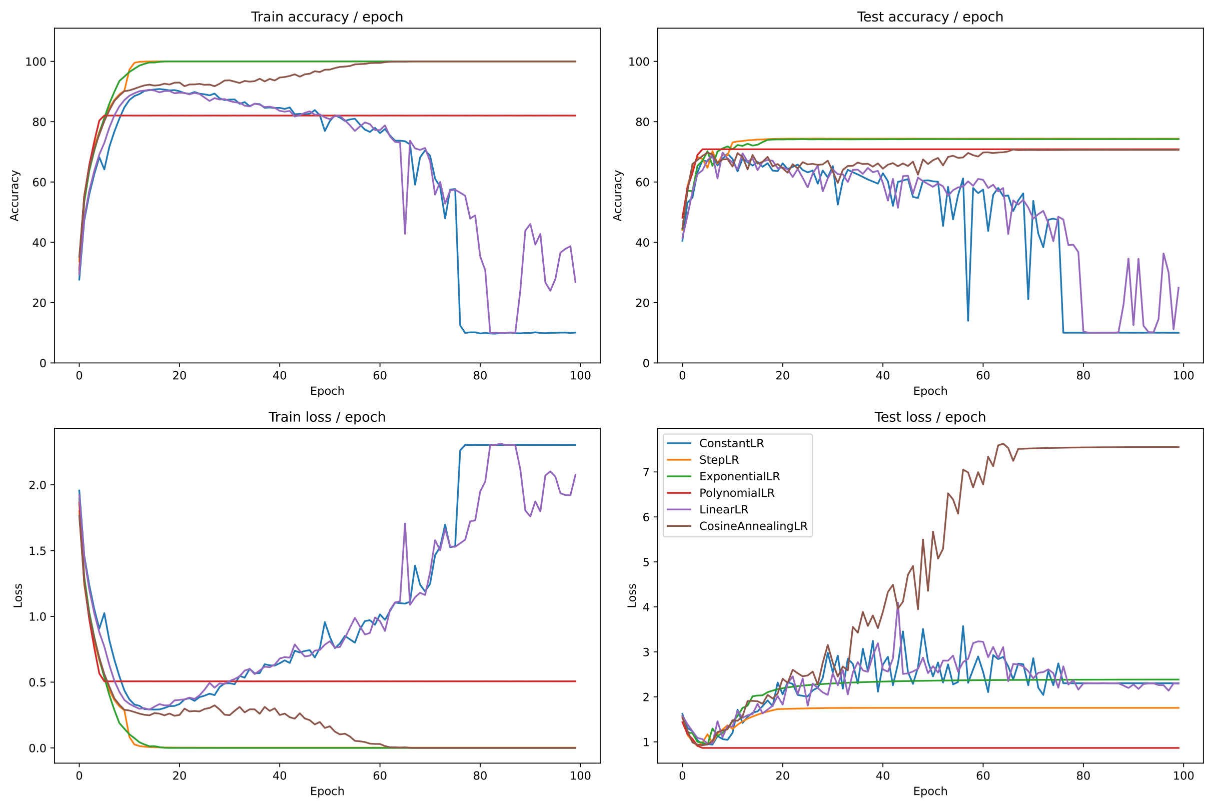

Figure 11: Learning rates curves

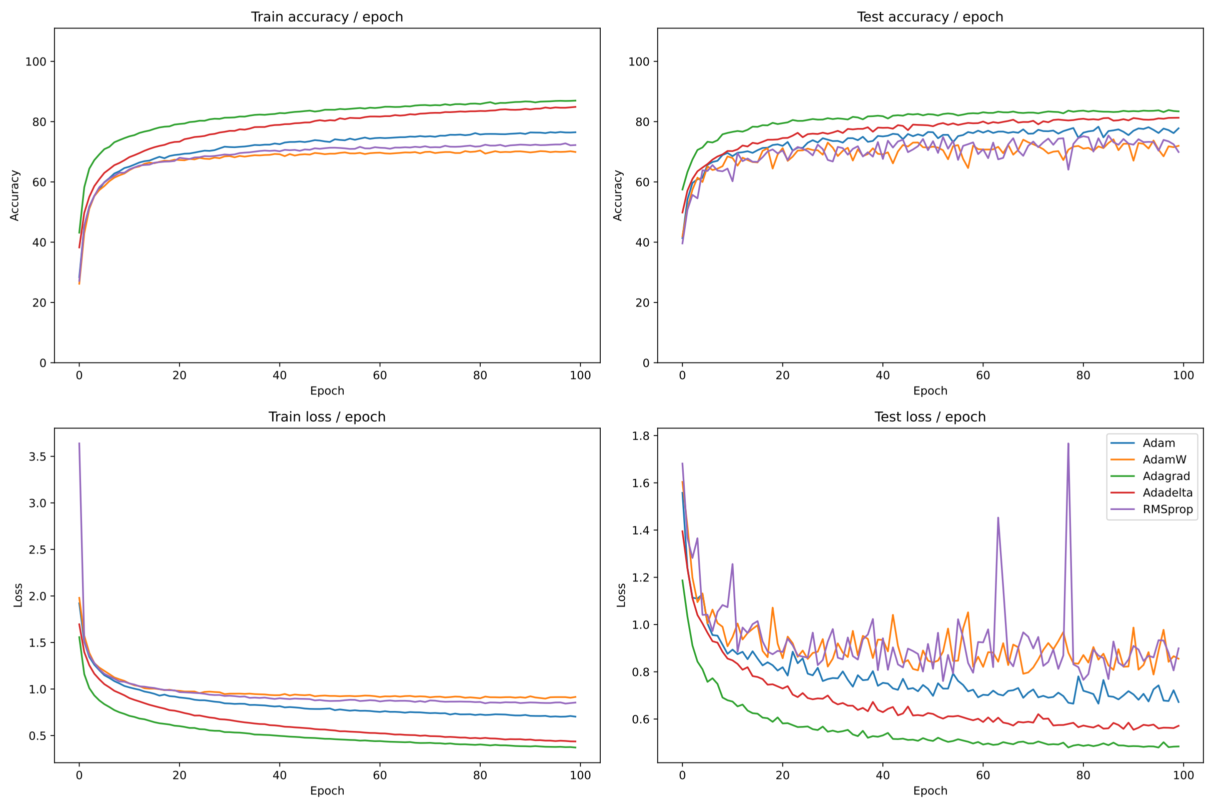

Figure 12: Influence of different optimizers

26. Describe your experimental results and compare them to previous results, including learning stability.

27. Why does this method improve learning?

2.3.4 Regularization of the network by ***dropout***

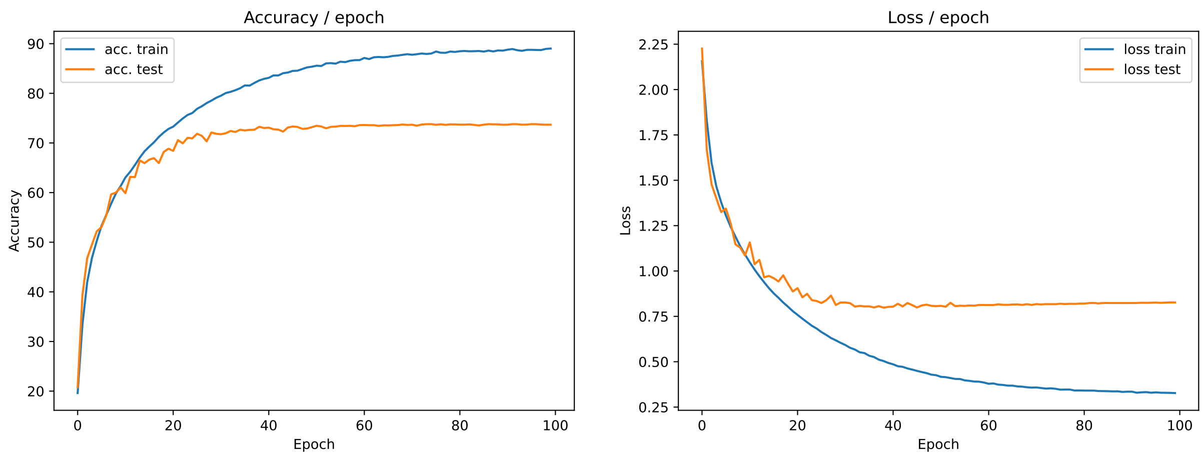

Figure 13: Accuracy and losses in train and test, using a dropout on `fc4`

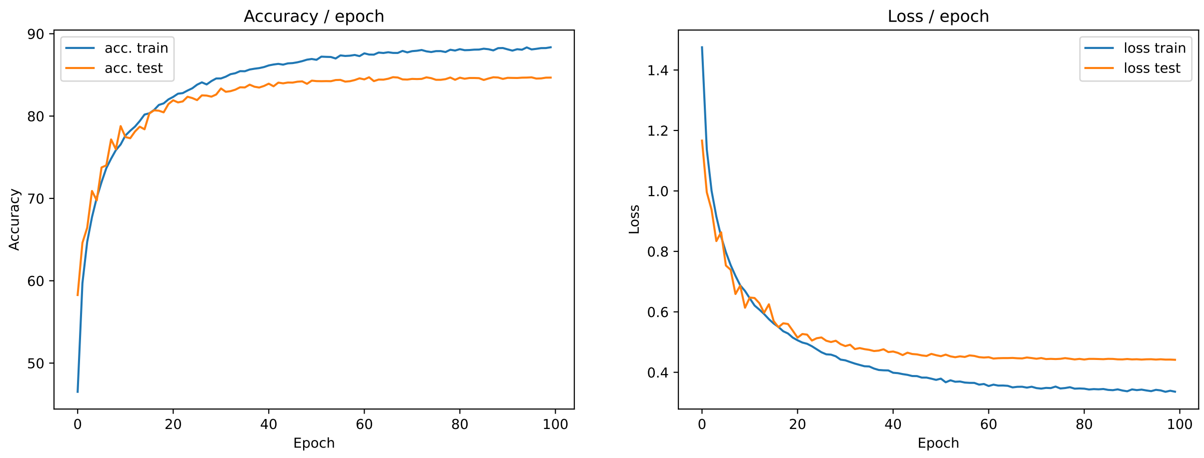

Figure 14: Accuracy and losses in train and test, using a dropout on `fc4` and a scheduler

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

30. What is regularization in general?

31. Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it?

32. What is the influence of the hyperparameter of this layer?

33. What is the difference in behavior of the dropout layer between training and test?

2.3.5 Use of *batch normalization*

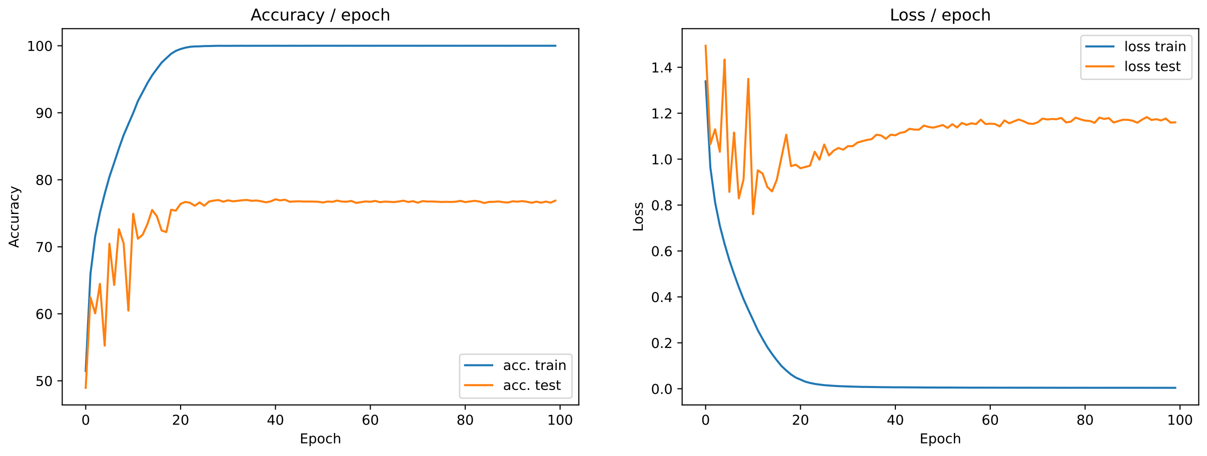

Figure 15: Accuracy and losses in train and test

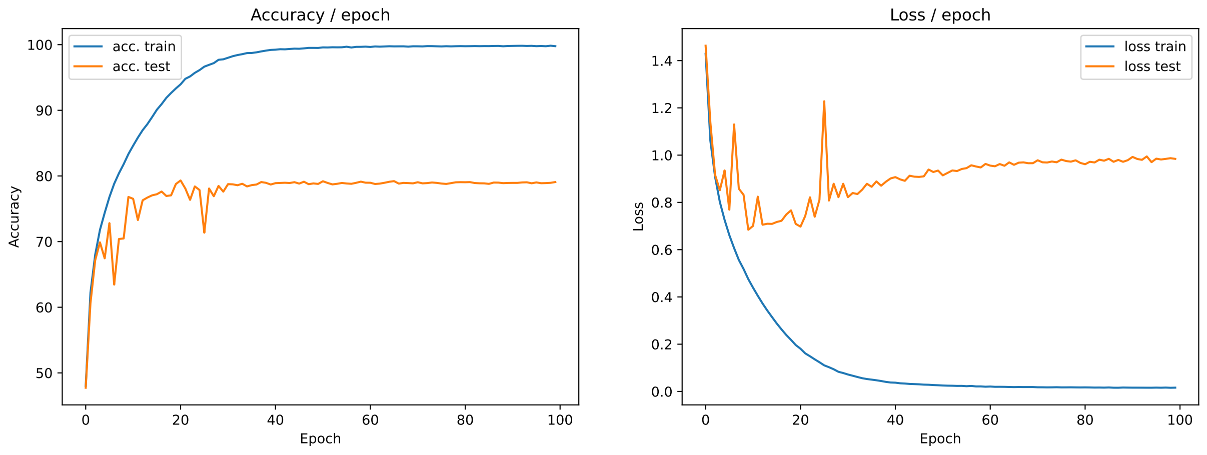

Figure 16: Accuracy and losses in train and test, with dropout layer

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

2.3.6 Combination of all methods

Now that we have individually explored various methods to enhance our results within a relatively straightforward convolutional network architecture, one might wonder what happens when we combine all these techniques, much like the Power Rangers assembling with their mechs. Our expectation is that this comprehensive approach will yield significantly improved results characterized by minimal overfitting, strong generalization, and enhanced robustness. Results confirm our expectation and are displayed in Figure 17.

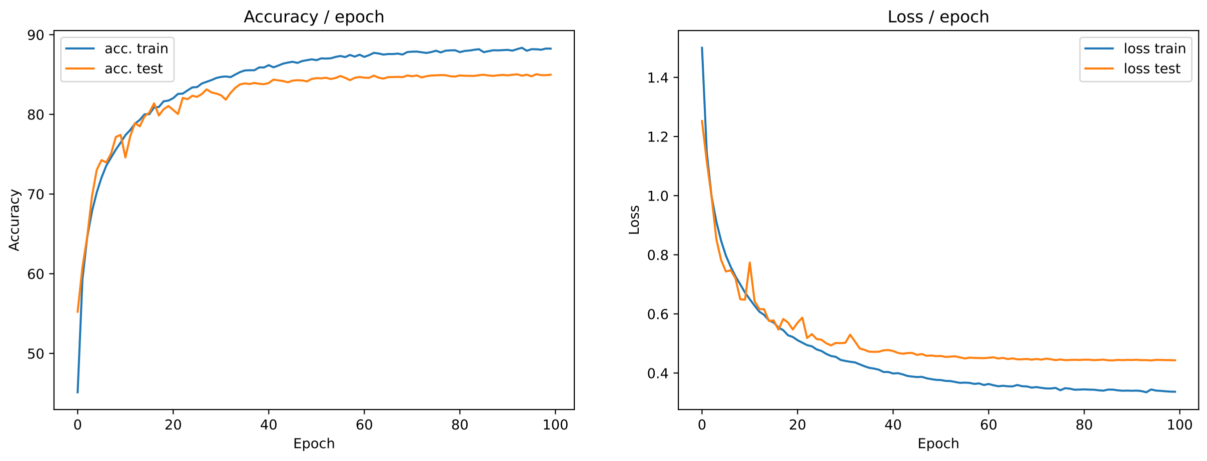

Figure 17: Accuracy and losses in train and test