RDFIA Homework

Argentieri Gabriele
Zattoni Francesco

January 14, 2026

Intro to Neural Networks

This report documents the systematic exploration and development of various Neural Network architectures applied to image classification conducted during the training sessions of the course. The methodology used was progressive, starting with simpler models and iteratively advancing towards modern, more complex solutions.
The initial focus was on establishing a robust understanding of deep learning fundamentals. This involved the implementation of core structures and the detailed analysis of optimization algorithms, such as Gradient Descent and its variants, which are critical to effective learning. The following phase addressed the inherent limitations of simpler models when handling images, particularly issues related to massive computational cost and the inability to leverage the 2D spatial structure of the image. This led to the introduction of the Convolutional Neural Networks.
Finally, the study culminated in the analysis and implementation of advanced architectures such as the Transformer applied to vision. This final step highlighted the shift from local, convolution-based pattern extraction to global, attention-based relationship modeling. The entire progression illustrates an experimental framework driven by the goal of finding stable, scalable, and highly accurate solutions for increasingly complex image classification challenges.

1 Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is a foundational class of feedforward artificial neural networks. It serves as a universal function approximator, capable of learning complex, non-linear relationships between inputs and outputs. This section details the "from-scratch" implementation of a simple two-layer MLP designed for a binary classification task, such as the circle dataset.

The core of this implementation relies on the principles discussed in the following questions: the forward pass propagates input data through the network to generate a prediction, and the backward pass (backpropagation) calculates the gradient of the loss function with respect to each network parameter. These gradients are then used by an optimization algorithm, Stochastic Gradient Descent (SGD), to update the network's weights and biases, minimizing the loss and improving model accuracy.

1-a

1. ★ What are the train, val and test sets used for?

Training set is used during training to adjust and correct the weights and the bias using backpropagation. Validation set too is used during training to show and check the loss on data that the network does not use to adjust the weights, in particular to avoid overfitting, where the network is not able to generalize on a new dataset learning 'too much' from the training set. Finally, test set is used to check performance of the trained network on a completely new dataset that it has never seen before.

2. What is the influence of the number of examples N?

Higher number of examples will lead to a slower training, that will have higher generalization capabilities. The training is slower because the number of forward passes depends on the number of examples N (actually on the number of batches that still depends on N and on the size of the batch itself). With a large N the network will see more examples of data present in the domain, and the final function $f(x)$ will more accurately approximate the real data distribution.

3. Why is it important to add activation functions between linear transformations?

Activation functions between layers introduce non-linearity. As proved by the universal approximation theorem, neural networks with activation functions can potentially approximate any function. Without non-linearity, increasing the number of layers would be completely useless, since the NN would behave as a single linear layer.

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

n_x is the input size, where the input is a vector in the space $\mathbb{R}^{n_x}$. n_h is the size of the hidden layer, i.e. the number of neurons present in the hidden layer. n_y is the size of the output $\hat{y}$; in the case of a multi-class classification task, it is the number of possible classes. n_x depends on the domain, i.e., on the size of the input data, and it can be chosen only if the data has been preprocessed. n_h is chosen by the NN designer, depending on the desired number of neurons. Finally, n_y depends on the task's classes.

5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?

The first vector $\hat{y}$ is the prediction of the NN, the second vector y is the vector encoding the information of the true label to predict. In a classification task, for example, $\hat{y}$ is the predicted classes' probability distribution given x, and y is a one-hot encoding, i.e. an array with the values all set to 0, except for the value referring to the true label that is set to 1.

6. Why use a `SoftMax` function as the output activation function?

`SoftMax` "computes" the maximum of an array as a probability distribution with an unbalance towards the largest values. As a result, the most probable classes are close to 1, while the others are significantly smaller. This function is preferred to the classical `max` because it is differentiable, a key aspect of the optimization algorithm based on backpropagation.

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

$$\tilde{h} = W_h x + b_h$$
$$h = \tanh(\tilde{h})$$
$$\tilde{y} = W_y h + b_y$$
$$\hat{y} = \texttt{SoftMax}(\tilde{y})$$

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 L?

- For cross entropy, the loss computes the "distance" between the distributions y and $\hat{y}$, and two scenarios must be considered: if y_i is 0, it is not the label of x; then the entire product is 0, thus not increasing the loss; otherwise, if y_i is 1, then to keep $\log \hat{y}_i$ low, $\hat{y}_i$ must be as close as possible to 1 so that log is closer to 0.
- For MSE, $\hat{y}_i$ will be driven towards the real y_i in both of the discussed scenarios to keep the difference $y_i - \hat{y}_i$ as low as possible.

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

The cross entropy loss is better suited for classification, handling y and $\hat{y}$ as probability distributions and making them as close as possible, pushing the $\hat{y}_i$, corresponding to the true label of x, towards 1.
Instead, the MSE is better suited for regression, a task where y has continuous values and where minimizing the MSE pushes $\hat{y}$ closer to y in $\mathbb{R}^{n_y}$.

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

- Classic gradient descent computes the true gradient on the full training set, leading to the direction of the local minimum, but it has to be computed by a mean on N samples, so it linearly scales w.r.t. N, becoming too slow for most datasets.

- Online stochastic gradient descent uses a single example chosen with i.i.d. draws (in practice shuffling is used) to approximate the true gradient, leading to a very coarse approximation, because it depends strongly on the single $\frac{\partial l}{\partial w}$, thus being easily subject to noise.

- Mini-batch stochastic gradient descent deals with this problem by computing the mean on a subset of size $B < N$. Using a subset generally leads to less noisy gradient approximations, without sacrificing too much of the performance (expecially using batches of the largest size that can fit in the GPU RAM in use).

The most reasonable choice in the general case is the **mini-batch version**, with a good trade-off between performance and robustness to noise.

11. ★ What is the influence of the *learning rate* η on learning?

The learning rate η is an important hyperparameter that controls the magnitude of the steps taken during the optimization process, and it has a direct influence on the convergence dynamics during the training of a neural network.
In particular, parameters are typically updated in this way:

$$w \leftarrow w - \eta \frac{\partial L(X, Y)}{\partial w}$$

where η multiplies the gradient of the loss function.
The selection of η is essential for convergence to a local minimum, in fact:

- **Too Small Values:** If η is too small, the steps taken are very small, leading to an excessively slow convergence during training.

- **Too Large Values:** Conversely, excessively large values can lead to overshooting the minimum, leading to oscillation or divergence.

Since a constant learning rate proves problematic, the most commonly adopted strategy is *Learning Rate Decay*. This involves initializing η with larger values to facilitate quick progress across the loss landscape and then decreasing it during training to achieve progressively smaller and more precise steps closer to the minimum.

12. ★ Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the loss with respect to the parameters, using the naive approach and the backprop algorithm.

- **Naive approach:** Complexity is quadratic w.r.t. L, the number of layers. This is due to the fact that for each layer in position l, we need to recompute the gradients of the following $L - l$ layers using the chain rule, so the number of times the gradients are computed is
$$\sum_{l=1}^{L}(L-l) = \frac{L(L-1)}{2} \in O(L^2)$$
- **Backprop algorithm:** Instead, here the complexity is linear, i.e. $O(L)$, because for each layer the computation of only one additional gradient is required and not of the entire chain. This efficiency is achieved by applying the chain rule such that derivatives calculated in previous steps are reused throughout the backward pass. Crucially, the computational cost for evaluating the gradient associated with a single input pattern is generally $O(W)$, where W is the total number of weights in the network, making the method highly scalable.

13. What criteria must the network architecture meet to allow such an optimization procedure?

The most important criterion is that the network architecture must meet is the differentiability of every component, since backpropagation requires to compute partial derivatives. This is achieved in the discussed network architecture, since it contains only differentiable components, such as cross-entropy loss, tanh activation function, matrix multiplication and additive bias.
Another criterion to make the optimization procedure useful is that the propagated gradient should be stable, i.e. not leading to vanishing or exploding values. This problem is relevant expecially for deeper networks and it can be solved by skip connections, as we have seen in ResNet.

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:

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

with c being the index of the correct class.

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

$$\frac{\partial l}{\partial \tilde{y}_i} = \frac{\partial}{\partial \tilde{y}_i}(-y_c\tilde{y}_c + y_c \log(\sum_k e^{\tilde{y}_k})) =$$

$$= \begin{cases} -y_c + \frac{e^{\tilde{y}_c}}{\sum_k e^{\tilde{y}_k}} & \text{if } i = c, \\ 0 + \frac{e^{\tilde{y}_i}}{\sum_k e^{\tilde{y}_k}} & \text{otherwise.} \end{cases}$$

$$= \frac{e^{\tilde{y}_i}}{\sum_k e^{\tilde{y}_k}} - y_i = \hat{y}_i - y_i$$

$$\nabla_{\tilde{y}} l = \hat{y} - y$$

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 $\nabla_{b_y} l$.

Knowing that

$$\tilde{y}_k = \sum_k W_{y,kj} h_j + b_{y,k}$$

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

then

$$\frac{\partial l}{\partial W_{y,ij}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = -y_c \frac{\partial \tilde{y}_c}{\partial W_{y,ij}} + \sum_k \hat{y}_k \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} =$$

$$= \begin{cases} -y_c h_j + \hat{y}_c h_j & \text{if } c = i, \\ 0 + \hat{y}_i h_j & \text{otherwise.} \end{cases}$$

$$\frac{\partial l}{\partial W_{y,ij}} = h_j(\hat{y}_i - y_i)$$

$$\nabla_{W_y} l = \begin{bmatrix} (\hat{y}_1 - y_1)h_1 & \dots & (\hat{y}_1 - y_1)h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_c - y_c)h_1 & \dots & (\hat{y}_c - y_c)h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y})h_1 & \dots & (\hat{y}_{n_y} - y_{n_y})h_{n_h} \end{bmatrix} = (\hat{y} - y)h^T$$

For $\nabla_{b_y} l$, we know that

$$\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \begin{cases} 0 & \text{if } k \neq i, \\ 1 & \text{otherwise.} \end{cases}$$

then

$$\frac{\partial l}{\partial b_{y,i}} = \sum_k \frac{\partial l}{\partial \tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i$$

$$\nabla_{b_y} l = (\hat{y} - y)$$

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

- Knowing that

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

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

$$\frac{\partial l}{\partial h_i} = \sum_k W_{y,ki} \frac{\partial l}{\partial \tilde{y}_k} = \sum_k W_{y,ki} (\hat{y}_k - y_k)$$

then

$$\frac{\partial l}{\partial \tilde{h}_i} = \sum_k \frac{\partial l}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} = \frac{\partial l}{\partial h_i} \frac{\partial h_i}{\partial \tilde{h}_i} = (1 - h_i^2)(W_y^T (\hat{y} - y))_i$$

$$\nabla_{\tilde{h}} l = \begin{bmatrix} (1 - h_1^2)(W_y^T (\hat{y} - y))_1 \\ \vdots \\ (1 - h_{n_h}^2)(W_y^T (\hat{y} - y))_{n_h} \end{bmatrix} = (1 - h^2) \odot (W_y^T (\hat{y} - y))$$

- Knowing that

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

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

then

$$\frac{\partial l}{\partial W_{h,ij}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} = \frac{\partial l}{\partial \tilde{h}_i} x_j$$

$$\nabla_{W_h} l = \begin{bmatrix} \frac{\partial l}{\partial \tilde{h}_1} x_1 & \cdots & \frac{\partial l}{\partial \tilde{h}_1} x_{n_x} \\ \vdots & \ddots & \vdots \\ \frac{\partial l}{\partial \tilde{h}_{n_h}} x_1 & \cdots & \frac{\partial l}{\partial \tilde{h}_{n_h}} x_{n_x} \end{bmatrix} = \nabla_{\tilde{h}} l \ x^T$$

- Knowing that

$$\frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

then

$$\frac{\partial l}{\partial b_{h,i}} = \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \frac{\partial l}{\partial \tilde{h}_i}$$

$$\nabla_{b_h} l = \nabla_{\tilde{h}} l$$

1-b

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

Default model: The first implementation of the neural network by hand uses the following parameters for training: $\eta = 0.03$, `NBatch = 10`.

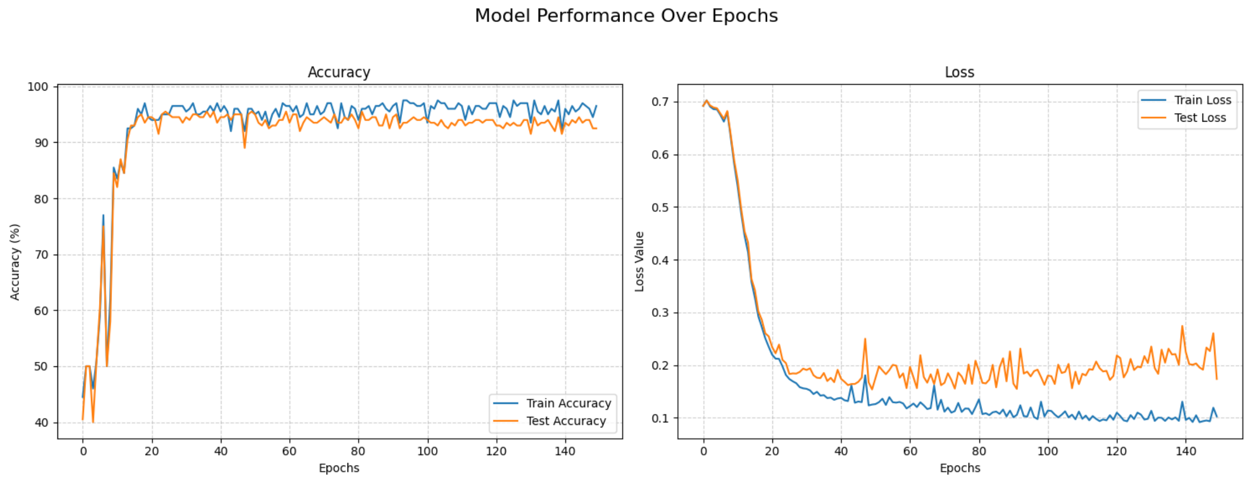

Figure 1: Accuracy and loss plots with the default batch size and learning rate.

Custom parameters:

- $\eta = 0.3$ **:** Figure 2 shows that a large value for η results in a faster convergence to an accuracy higher than 90%, followed by large jumps back and forth in the loss landscape leading to oscillations in the accuracy.

- $\eta = 0.001$ **:** This chosen value for η is too small, resulting in very slow training and underfitting. Indeed, the final values for both accuracy and loss are worse than those in previous experiments. The main visible advantage in figure 3 is that the explored loss landscape is smoother.

- `NBatch = 1`**:** This scenario tests classical gradient descent, by using a single batch. Figure 4 does not show significant differences w.r.t. default parameters used in figure 1.

- `NBatch = N`: Instead, online stochastic gradient descent results in large oscillations, caused by the very noisy direction taken by the loss at each iteration.

The answers to the other questions can be found in the attached *Jupyter Notebook*.

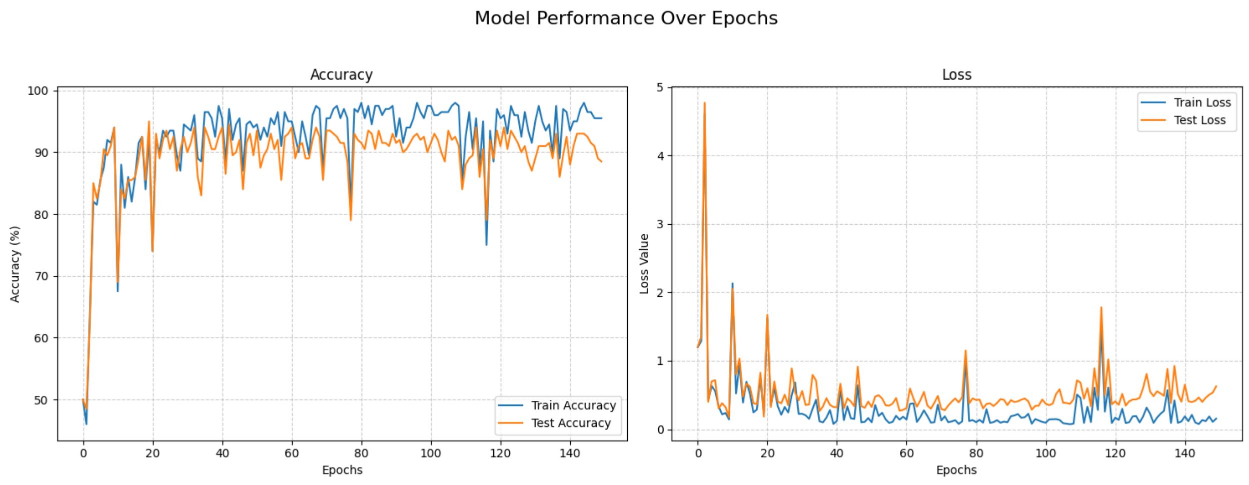

Figure 2: Accuracy and loss plots with the given batch size and learning rate $\eta = 0.3$.

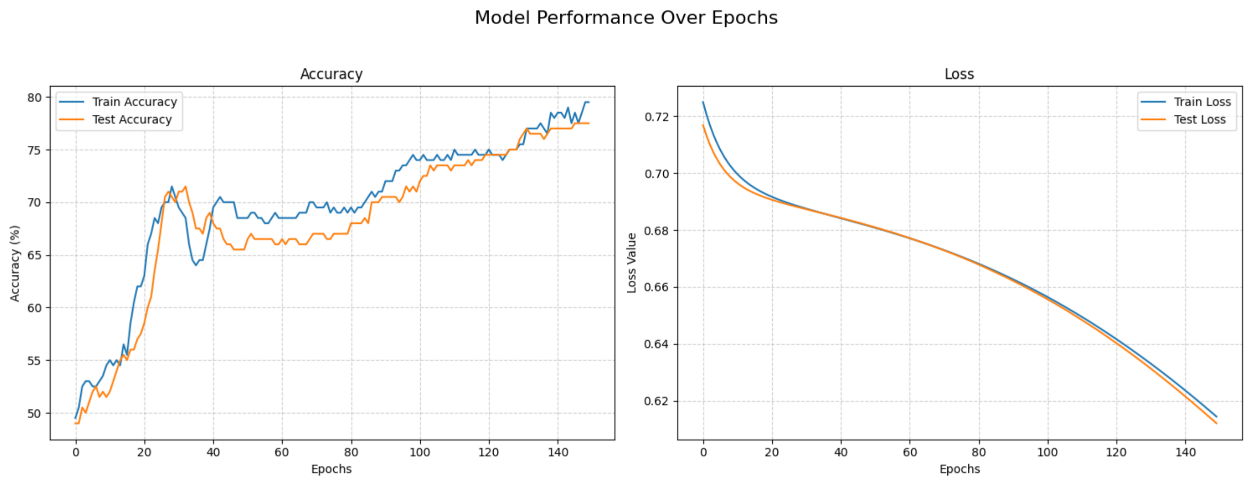

Figure 3: Accuracy and loss plots with the given batch size and learning rate $\eta = 0.001$.

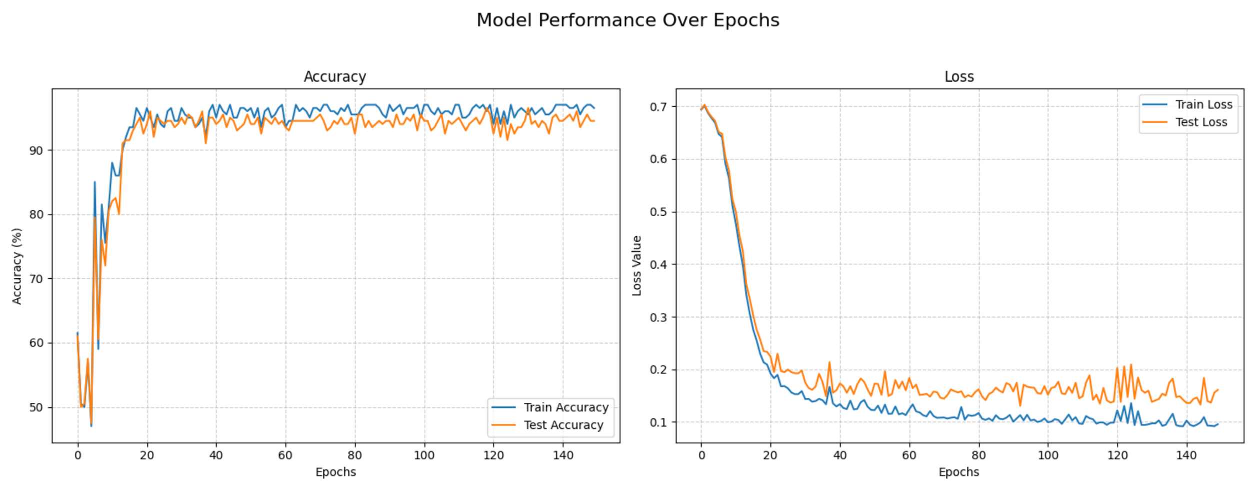

Figure 4: Accuracy and loss plots with the classic gradient descent (`NBatch = 1`) and given learning rate.

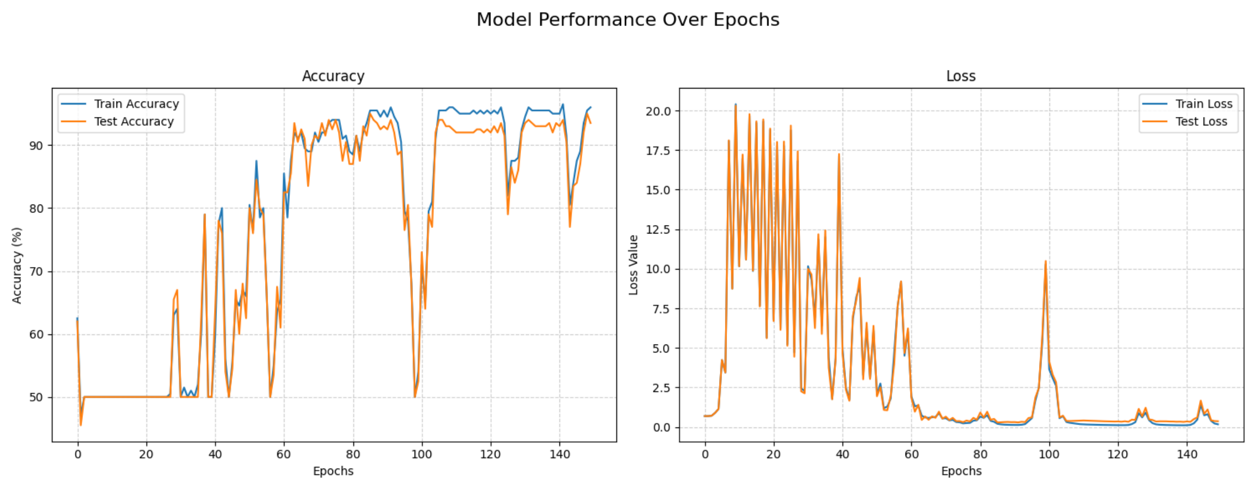

Figure 5: Accuracy and loss plots with the online stochastic gradient descent (`NBatch = N`) and given learning rate.

Convolutional Neural Networks (CNN)

While Multi-Layer Perceptrons (MLPs) are effective function approximators, they suffer from significant drawbacks when applied to images, leading to massive computational cost and a high risk of overfitting. Furthermore, MLPs discard the crucial 2D spatial structure of the image, treating pixels that are far apart the same as pixels that are adjacent.
CNNs solve this problem, since are built upon different principles, which are explored in the following questions: sparse interactions, parameter sharing, pooling layers. This architecture has proven to be very effective for image classification tasks, such as those on the CIFAR-10 dataset explored in this section.

1-c

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?

Defining S as the stride applied to the convolution, k as the kernel size, p as the size of the padding, then the output size will be $x_{out} \times y_{out} \times z_{out}$, where:

- $z_{out} = 1$
- $x_{out} = \lfloor \frac{x-k+2p}{S} \rfloor + 1$
- $y_{out} = \lfloor \frac{y-k+2p}{S} \rfloor + 1$

The number of weights to learn is k^2 since there's only one filter.
If we had to learn a FC layer to produce an output of the same size we would have needed $x_{in} \times y_{in} \times z_{in} \times x_{out} \times y_{out} \times z_{out}$ weights.

2. ★ What are the advantages of convolution over fully-connected layers? What is its main limit?

Convolutional layers address the limitations of Multi-Layer Perceptrons applied directly to raw image data, which were related to scalability and geometric properties.
The primary advantages of convolution are:

1. **Sparse Interactions:**
   - Every unit in a in a convolutional layer is only connected to a small local region of the input, its *receptive field.* In a FC layer, every output unit interacts with every input unit.
   - This drastically **reduces the computational runtime** from $O(m \times n)$ to $O(k \times n)$ operations per example (where m is the number of inputs, n is the number of outputs, and k is the kernel size, which is much smaller than m).

2. **Parameter Sharing:**
   - While FC layers use a unique parameter for every connection, convolutional layers reuse the same set of kernel weights across the entire input field.
   - This property greatly reduces the number of parameters to learn and helps preserve the spatial topology of the input data.
3. **Translation Equivariance:**
   - The parameter sharing implemented by convolution makes it **equivariant to translation**, so if the object in the input image is shifted, its output will shift by the same amount.

Nevertheless, convolution also comes with some limitations. For example, it cannot take into account interactions between distant patches of the image, because it is limited by the fixed size of the filter at that specific layer. Furthermore, standard convolution is **not naturally equivariant** to other important transformations, such as changes in the scale or rotation of an image, requiring additional mechanisms (like Pooling) to achieve stability (invariance) to these types of deformations.

3. ★ Why do we use spatial pooling ?

Spatial pooling is a technique useful to make the model invariant w.r.t. 2D transformations of the input, so addressing an impactful problem of the basic convolutional models.
It works by reducing the number of inputs to the next layer of feature extraction, for example by taking the average or the max in the neighbourhood.

4. ★ Suppose we try to compute the output of a classical convolutional network (for example the one in Figure 2) for an input image larger than the initially planned size (224 $\times$ 224 in the example). Can we (without modifying the image) use all or part of the layers of the network on this image ?

While the convolutional and pooling layers are robust to variable input dimensions, this is not true for the fully-connected ones.

Convolutional Layers: In fact, the number of trainable parameters in a convolutional layer depends only on the kernel size and the number of channels, and not on the spatial dimensions of the input image.
If the input image is larger, the output feature map will scale accordingly in size, so the convolution operation will simply be applied a greater number of times.
Thus, in that case, the initial sequence of convolution and pooling will be successful, producing a feature map tensor whose spatial dimensions are simply larger than originally planned.

Fully-Connected Layers: The main problem with these layers is that the standard matrix multiplication requires the input vector to have a fixed and predetermined size. Thus, since the size of the vector resulting from the flattened output of the last pooling layer will be different than expected from the first fully-connected layer, the forward pass will fail.
A way to overcome this problem would be converting the network to a fully-convolutional one, so eliminating the fully-connected part that was causing the problem.

5. Supposing that we are only interested only in one, central, output pixel of a convolution, how can we modify the convolutional filter to make it equivalent to a fully-connected layer.

To make a convolutional layer act like a fully-connected layer, we must set the kernel's spatial dimensions equal to the input's spatial dimensions (kernel size $H_{in} \times W_{in}$). With padding set to zero, this filter can only be applied in one single position. This operation computes a dot product, making it mathematically equivalent to a single neuron output in a fully-connected layer.

6. There is a modification of convolution called "unshared convolution". This maintains the "sparse connectivity" pattern of convolutions, but each input pixel has its own weights (no weight sharing). For an unshared convolution layer, with an input of shape (B;C;H;W) and Q filters of size 5 × 5, how many parameters (not including biases) would this layer have ?

In "unshared convolution", where there is no weight sharing, the number of parameters will depend also on the number of input pixels. Supposing padding set to 2, to keep the same spatial dimensions, then the number of parameters is $5^2 \times C \times H \times W \times Q$.

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 ?

Defining k_1 the size of the first kernel, k_2 the size of the second kernel and supposing the stride set to 1, then the sizes of the receptive fields of the neurons of the first and second convolutional layers are:

- **First layer:** Equivalent to the kernel sizes, $k_1 \times k_1$;
- **Second layer:** The receptive field gets larger, because each of the k_2^2 pixels is the result of k_1^2 pixels of the image. The receptive field's size increases of $k_1 - 1$ pixels, divided in the two opposite sides. The receptive field is $(k_2 + k_1 - 1) \times (k_2 + k_1 - 1)$.

Deeper layers will have larger and larger receptive fields, since each output pixel is the result of k^2 overlapping input pixel. This aspect allow deeper neurons to have a larger amount of local information, allowing the detection of more complex features, like objects or parts of them.

1-d

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 ?

To keep the same spatial dimensions at the output as at the input it's needed to set the stride to $S = 1$ and padding to $p = \lfloor \frac{k}{2} \rfloor$, such that the stride doesn't reduce the number of pixels and that the padding allows the convolution to have enough space to compute all the values.

9. For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed ?

For this purpose $S = 2$ and $p = 0$ are needed.

10. ★ For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.

Since the images in the dataset are RGB, the input size needs to count also the colour channels, so it is $32 \times 32 \times 3$. Supposing that each convolutional layer keeps the same spatial dimensions H and W, the padding and stride hyperparameters are set to $S_{conv} = 1$ and $p = \lfloor \frac{5}{2} \rfloor = 2$. Also, it is supposed that max-pooling is used to half the spatial dimension, so $S_{pool} = 2$.
The output sizes and number of weights are respectively:

- **conv1**: $32 \times 32 \times 32$ and $5 \times 5 \times 32 = 800$ weights
- **pool1**: $16 \times 16 \times 32$
- **conv2**: $16 \times 16 \times 64$ and $5 \times 5 \times 64 = 1600$ weights
- **pool2**: $8 \times 8 \times 64$
- **conv3**: $8 \times 8 \times 64$ and $5 \times 5 \times 64 = 1600$ weights
- **pool3**: $4 \times 4 \times 64$
- **fc4**: $1000 \times 1 \times 1$ and $1024 \times 1000 = 1024000$ weights
- **fc5**: $10 \times 1 \times 1$ and $1000 \times 10 = 10000$ weights

This repartition highlights a relevant characteristic of AlexNet: most of the weights ($\approx 99,6\%$) are in the last section of the network, i.e. in the fully connected layers.
In the convolutional layers, instead, the number of weights do not depend on the input size but only on the number and size of the filters.

11. What is the total number of weights to learn? Compare that to the number of examples.

The total number of weights is the sum of the ones in the convolutional layers and in the fully connected layers:

$$\sum_{i=1}^{N_{layers}} W_i = 800 + 1600 + 1600 + 1024000 + 10000 = 1038000$$

Since the dataset has 50k images in train and 10k images in test, the model has roughly 20 weights for every training example.

12. Compare the number of parameters to learn with that of the BoW and SVM approach.

Using BoW in conjunction with SVM needs a significantly lower number of parameters that the nearly one million of the architecture specified above.
Bag of words, in fact, generates a feature vector whose size depends on the dictionary size K. This resulting histogram is the input of the SVM, thus the final result depends linearly on K and on the number of classes.

14. ★ In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the difference in data) ?

During training, the loss is used for backpropagation (gradient computation via `loss.backward()`) to update model parameters through `optimizer.step()`.
During testing, the loss is computed only as an evaluation metric, with no gradient computation or parameter updates (no `backward()` call).
The semantic difference is that in training, loss drives learning; in testing, loss only measures performance. Accuracy measures performance in both parts.

16. ★ What are the effects of the *learning rate* and of the *batch-size* ?

- **Lower learning rate:** By setting $\eta = 0.01$, i.e. a value 10 times smaller than the ones used in previous experiments, the training gets much slower, being able to reach an accuracy of only 56%, a clear sign of underfitting, as visible in figure 6.
- **Small batch size:** By setting $B = 32$, i.e. a value 4 times smaller than the ones used in previous experiments, the loss updates are much more noisy, since the gradient approximation is more coarse, as it is possible to notice in figure 7. The final accuracy is similar to the test accuracy obtained with $B = 128$, leading to a similar overfitting effect.

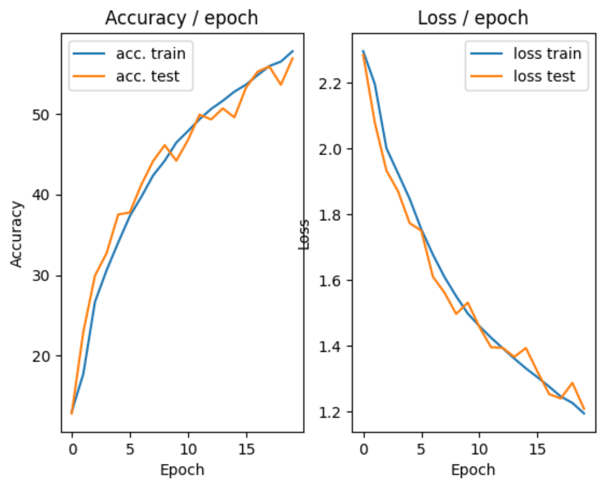

Figure 6: Accuracy and loss plots for basic CNN on CIFAR-10 with $\eta = 0.01$.

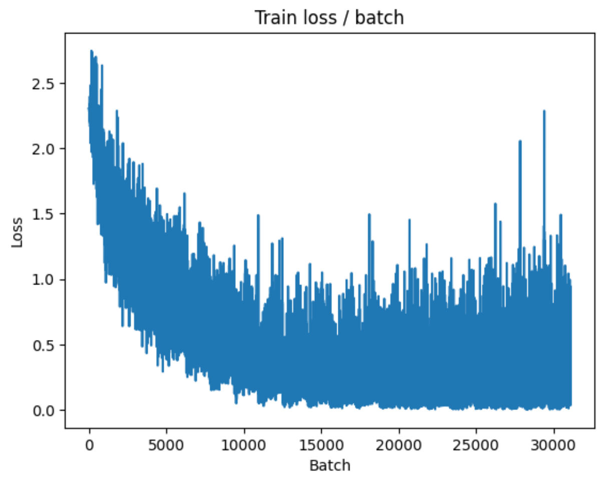

Figure 7: Accuracy and loss plots for basic CNN on CIFAR-10 with batch size $B = 32$.

17. What is the error at the start of the first epoch, in train and test ? How can you interpret this ?

At the beginning the error rate is high in both test and training, primarily because the weights had been randomly initialized.

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

Figure 8 clearly shows an overfitting problem: the train's loss is very close to 0, while the test's loss has started to increase before epoch 10. Also, the difference of train accuracy and test accuracy of 30% is an evident hint that the base model without data augmentation and regularization techniques, like dropout, is not able to generalize.
From epoch 10, the test accuracy did not improve at all, instead train accuracy got signifi-

cantly better, leading to a nearly 100% accuracy. This is due to the fact that the model, to decrease the loss, gave too much importance to features mainly present in training data.

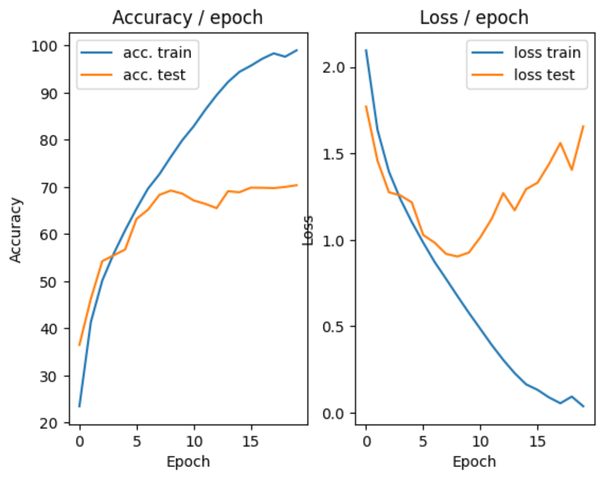

Figure 8: Accuracy and loss plots for basic CNN on CIFAR-10.

19. Describe your experimental results.

Normalization does not introduce regularization, but instead it makes learning easier. By standardizing the input data (to *mean* ≈ 0, *std* ≈ 1), it ensures pixel channels are on a similar scale. It ensures an easier to navigate loss landscape, which allows a much faster and more stable convergence.
Figure 9 confirms this: the training loss drops very quickly. However, this efficient learning also leads to evident overfitting, as normalization itself does not prevent the model from memorizing the training data.

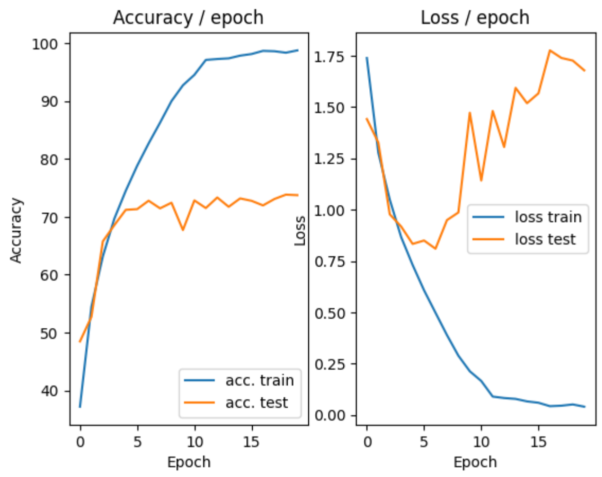

Figure 9: Accuracy and loss plots for CNN on normalized CIFAR-10.

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

Calculating the average image also with the validation set data would lead to its statistics being embedded into the transformation applied to the training data.
That would be unacceptable because the validation set is used to assess the model's ability to generalize to new data, and obviously in the real world the model can't know the statistics of unknown data.

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.

ZCA is a transformation that decorrelates the input features and normalizes their variance. In a typical image, nearby pixels are highly correlated. ZCA transforms the data so that the new features are no longer linearly related (i.e., the covariance matrix of the data becomes the identity matrix). Then, it scales each of these new, decorrelated features so that they all have a variance of 1.

Figure 10 shows results similar to figure 9: the training loss drops very quickly and it has an evident overfitting problem.

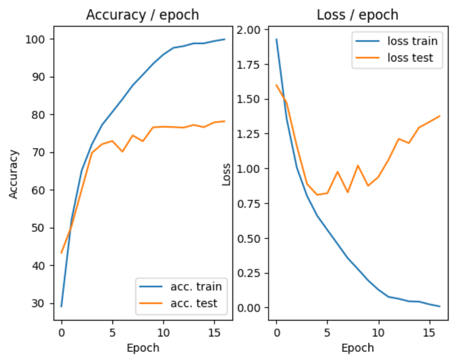

Figure 10: Accuracy and loss plots for CNN on normalized CIFAR-10 by ZCA.

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

The data augmentation acts as a strong regularizer. The techniques used are:

- `RandomCrop((28))`: Forces the model to recognize objects even when they are not perfectly centered or fully visible.
- `RandomHorizontalFlip()`: Teaches the model orientation invariance (e.g., a cat facing left is still a cat).

Compared to previous results, the gap between training and test curves, for both accuracy and loss, is much smaller. Also, the training accuracy no longer reaches 100%, showing the model is being challenged by the augmented data and is not just memorizing the training set, since the training is harder. Finally, the final test accuracy is higher, reaching 80% (compared to 70% previously), indicating better generalization.

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

The horizontal symmetry approach is not usable on all types of images.

- **Usable:** When the image label is invariant to horizontal flipping. This works well for general object recognition (e.g. CIFAR), where a mirrored version is still a valid example.
- **Not Usable:** When flipping changes the meaning or label of the image. This is common in text/digits recognition, like MNIST dataset.

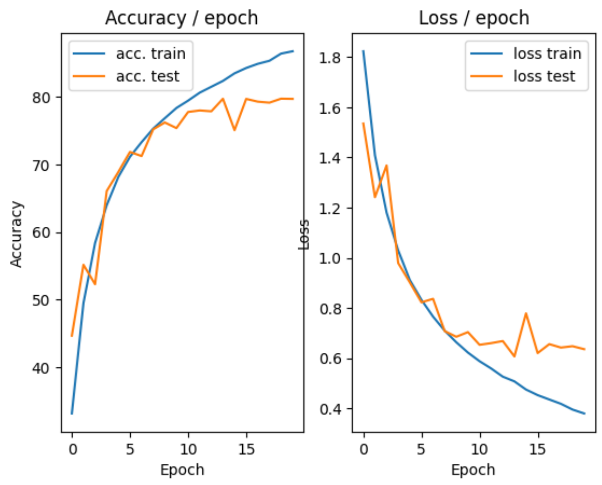

Figure 11: Accuracy and loss plots for CNN data augmented CIFAR-10.

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

- **Not new information:** The augmented data is highly correlated with the original data. It cannot introduce new concepts, backgrounds, or objects that were not in the original dataset.
- **Semantic change:** Transformations can be "unsafe" and change the image's meaning, as discussed in previous answer, or even lose completely the meaning if the augmentation is too aggressive.

25. Bonus : Other data augmentation methods are possible. Find out which ones and test some.

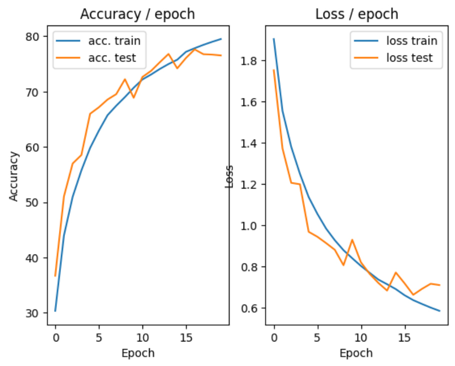

Figure 12: Accuracy and loss plots for CNN data augmented CIFAR-10 by random affine transformations.

The new augmentation method is `RandomAffine`, which applies random rotations, translations, and scaling to the images. Combined with `RandomCrop`, it acts as a extremely strong regularizer. Indeed, the training and test curves (both loss and accuracy) are now almost identical. Finally, the final test accuracy remains high, at approximately 80%, demonstrating that the regularization has not led to underfitting.

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

SGD with exponential LR decay does not act as a regularizer. Its goal is to improve optimization stability. Indeed, overfitting is visible as a clear gap re-emerged between the training and test curves. The training curves are very smooth, showing the LR decay provides stable optimization on the training set.
The test curves, especially the test loss are unstable and noisy. This instability suggests that while the model is fitting the training data well, it is not finding a stable, generalizable solution for the test data.
Compared with figure 12, it contrasts sharply, which had no overfitting and very stable test curves. This shows that LR scheduling helps optimization, but data augmentation is more crucial for generalization.

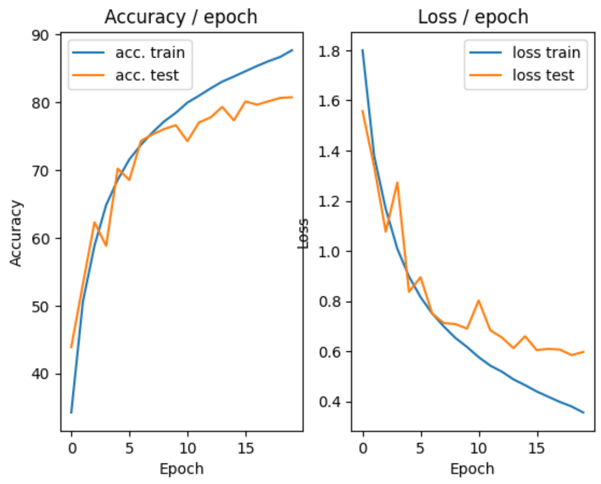

Figure 13: Accuracy and loss plots for CNN on CIFAR-10 by SGD with exponential decay on the learning rate.

27. Why does this method improve learning ?

Learning Rate Decay improves optimization by balancing speed and stability.
It starts with a relatively high learning rate, allowing the optimizer to take large steps, make quick progress, and escape shallow local minima early in training. As training progresses, the LR is gradually decreased. This allows the optimizer to take smaller, more precise steps as it gets closer to a loss minimum, enabling it to find a good solution instead of oscillating back and forth around it.

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

We tested two other variants:

- **SGD with step LR decay:** This method, like exponential decay, is an optimization technique, not a regularizer, so mild overfitting is present

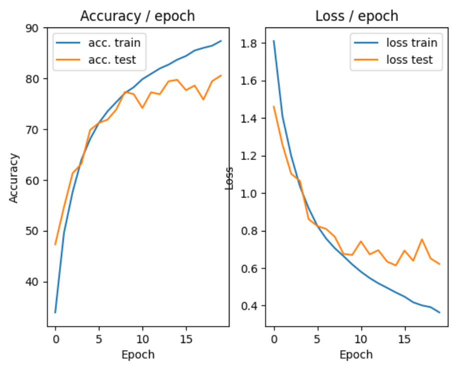

Figure 14: Accuracy and loss plots for CNN on CIFAR-10 by SGD with step decay on the learning rate.

- **Adam optimizer:** Adam converges faster in the initial epochs compared to SGD. The training loss drops more steeply at the beginning. The key difference is stability. The test loss curve is significantly more stable and less noisy than SGD decay methods. After its initial descent, it settles into a smooth plateau.

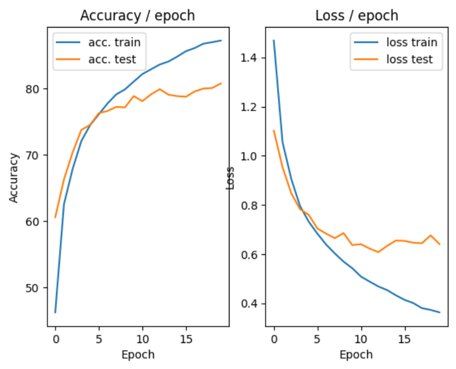

Figure 15: Accuracy and loss plots for CNN on CIFAR-10 by Adam optimizer

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

Both training and testing accuracies increase over 20 epochs, reaching approximately 80% and the gap between the two curves is minimal. Also, both training and testing losses decrease.

A model without dropout would typically overfit, as shown in previous experiments without any added regularization. This would be visible as a large, widening gap between the training and testing curves.

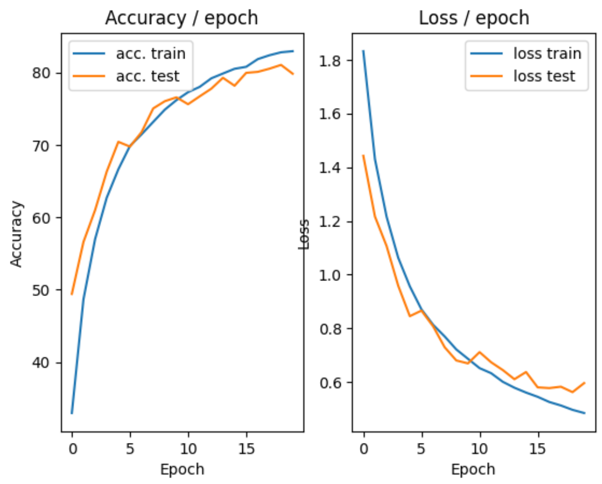

Figure 16: Accuracy and loss plots for CNN on CIFAR-10 with dropout

30. What is regularization in general ?

Regularization is a collection of techniques used to prevent overfitting in machine learning models. It works by adding a constraint, or ”noise” to the learning process, discouraging the model from becoming overly complex and memorizing the training data. The primary goal is to improve the model's generalization, meaning its ability to perform well on new, unseen data.
In the previous examples, the main regularization techniques used were:

- **Data Augmentation** (e.g., `RandomCrop`, `RandomHorizontalFlip`, `RandomAffine`)
- **Dropout**

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

Dropout's primary effect is regularization, specifically to prevent overfitting by breaking up neuron co-adaptation. One of the authors of dropout's paper[1], Geoffrey Hinton, explained it with a ”conspiracy” analogy:

- **Without dropout:** Neurons can form a large, complex ”conspiracy” (co-adaptation). They become highly dependent on each other to fit the training data. This ”conspiracy” is fragile and fails when faced with new, unseen data.

[1]2012, *”Improving neural networks by preventing co-adaptation of feature detectors”*, Hinton et al.

- **With dropout:** By randomly deactivating neurons, dropout removes these large "conspiracies". It forces the network to learn many smaller, independent "conspiracies" (robust features) that work without relying on specific neighboring neurons.

This forces the network to learn more resilient features, leading to better generalization on test data.

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

The main hyperparameter is the dropout rate p (the probability of dropping a neuron), that directly controls the strength of the regularization.

- A higher p (e.g., 0.5) provides stronger regularization, forcing the network to learn more robust features and preventing overfitting.
- If p is too high, it can lead to underfitting (the network becomes too simple to learn the task).
- A lower p (e.g., 0.2) provides weaker regularization.

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

- **During Training:** The dropout layer randomly sets a fraction p of its input units to 0 at each update. To compensate, it scales the remaining units by a factor of $\frac{1}{1-p}$, as implemented in the `pytorch` library.
- **During Testing:** The dropout layer passes all inputs through unchanged, behaving as an identity layer. The scaling done during training ensures that the expected output of any neuron is the same during training and testing, so no modifications are needed.

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

Batch Normalization acts as a powerful regularizer, in addition to its primary role of stabilizing and accelerating training. The overfitting seen before almost completely gone. The training and test curves (for both accuracy and loss) track each other extremely closely. The model converges quickly, with both training and test accuracy rising rapidly in the first few epochs.
Despite the good generalization, the test curves are very noisy. This high variance from epoch to epoch suggests that while BN is regularizing, it's also leading to unstable validation performance.

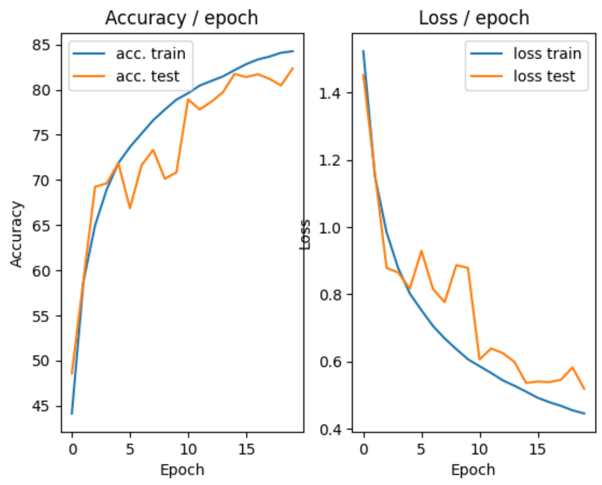

Figure 17: Accuracy and loss plots for CNN on CIFAR-10 with batch normalization.

★ **For this part, in the end of session report, indicate the methods you have successfully implemented and for each one explain in one sentence its principle and why it improves learning.**

- **Standard Normalization (Mean/Std):** Its principle is to rescale all input pixels to have a mean of 0 and standard deviation of 1, which improves learning by ensuring all features are on a similar scale, leading to faster and more stable optimizer convergence.
- **ZCA Whitening:** Its principle is to transform the input data so its features are decorrelated and have unit variance, which improves learning by simplifying the data's structure, allowing the optimizer to converge very efficiently.
- **Data Augmentation (Crop & Flip):** Its principle is to randomly crop and horizontally flip images during training, which improves learning by acting as a regularizer that teaches the model invariance to position and orientation, thus improving generalization.
- **Data Augmentation (Affine):** Its principle is to apply a wider range of random transformations (rotation, translation, scaling), which improves learning by providing even stronger regularization that forces the model to learn highly robust features, further improving generalization.
- **Learning Rate Schedulers (Exp/Step Decay):** Their principle is to gradually reduce the learning rate η during training, which improves learning by allowing the optimizer to make large progress initially and then "fine-tune" more precisely as it approaches a loss minimum, leading to better optimization.
- **Adam Optimizer:** Its principle is to use adaptive, per-parameter learning rates and momentum estimates, which improves learning by converging faster and (in this case) producing a more stable, less noisy test loss curve than standard SGD.

- **Dropout:** Its principle is to randomly set a fraction of neuron outputs to zero during training, which improves learning by acting as a regularizer that prevents neuron co-adaptation, forcing the network to learn more robust features.
- **Batch Normalization:** Its principle is to re-normalize the activations within each mini-batch at every layer, which improves learning by stabilizing the network's internal distributions (reducing "internal covariate shift"), allowing for much faster convergence and also providing a regularization effect.

2 Transformer Architecture

Vision Transformers (ViT) represent a paradigm shift in computer vision, applying the Transformer architecture, originally developed for Natural Language Processing, directly to image data. Unlike CNNs, which rely on local interactions, ViT processes an image by splittting it into a sequence of non-overlapping patches, allowing the model to learn global interactions thanks to the self-attention mechanism. The following excercises explore the implementation of the architecture, also highlighting how different choices of hyperparameters affect performance.

1-b

The answers to the questions can be found in the attached *Jupyter Notebook*.