RDFIA

Basics on deep learning for vision

29/10/2024

AYED Hatem 21316185
BENAHMED Neil 21200977

Table des matières

Chapter 1

Theoritical foundation

1.1 Supervised dataset

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

The training set is used to adjust the model's weights by minimizing the loss on this data. The validation set is employed during training to tune hyperparameters (e.g., learning rate, regularization) and evaluate the model's performance on data not directly trained on. The test set is reserved for final evaluation after training to measure how well the model generalizes to unseen data.

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

The larger the number of examples N, the more information we obtain about the distribution of X and Y. This allows us to train our model on a sample that is increasingly representative of these distributions, leading to a better model's generalization ability, provided that the data is well-distributed. With fewer examples, the model is prone to overfitting, learning specific details of the training data but failing to generalize well to new data.

1.2 Network architecture (forward)

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

Activation functions introduce non-linearity between linear transformations. Non-linearity enables the network to model complex and non-linear relationships in the data. Otherwise, the model would simply be aggregating linear functions, resulting in a linear output.

1.2.2 4. What are the sizes nx, nh, ny in figure 1? How are these sizes chosen in practice?

In figure 1, nx represents the number of input features, nh is the number of hidden units, and ny is the number of output units (classes). These sizes are chosen as follows:

- $nx = 2$ is the number of features in the input data.
- $nh = 4$ is a hyperparameter, it represents the size of the hidden layer, chosen to capture the complexity of features learned by the model. A too large hidden layer may lead to overfitting, and a too small layer may lead to underfitting.a
- $ny = 2$ corresponds to the number of output classes for classification problems.

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

$\hat{y}$ is the output vector predicted by the neural network, while y is the ground truth (or label) from the dataset. The difference is that y is the actual correct label with binary values $y \in \{0, 1\}$, whereas $\hat{y} \in [0, 1]$ is the network's predicted output, representing a probability-like score for each class. $\hat{y}$ reflects the model's confidence in its predictions for each class.

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

The SoftMax function normalizes the outputs of the network into a probability distribution, with values between 0 and 1 that sum to 1. This is particularly useful for multi-class classification problems, where we need to interpret the outputs as probabilities corresponding to different classes.

1.2.5 7. Write the mathematical equations allowing to perform the forward pass of the neural network.

The forward pass of the neural network can be written as follows:

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

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

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

$$\hat{y} = \text{SoftMax}(\tilde{y})$$

where W_h and W_y are weight matrices, b_h and b_y are bias vectors, and $\hat{y}$ is the final prediction of the network.

1.3 Loss function

1.3.1 8. How must the $\hat{y}_i$ vary to decrease the global loss function L?

To minimize the loss, the predicted values $\hat{y}_i$ must approach the true values y_i. In the case of cross-entropy, this means that the probability for the correct class (where $y_i = 1$) must increase, and the probabilities for the incorrect classes must decrease.

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

Cross-entropy is more suitable for classification tasks because it strongly penalizes incorrect predictions in cases where the outputs represent probabilities. The mean squared error (MSE) is more suited for regression tasks because it measures the absolute difference between the predicted and true values, making it appropriate for continuous outputs.

Cross-Entropy Loss for Classification: Cross-entropy loss is particularly suited for classification tasks because it measures the dissimilarity between the predicted probability distribution and the actual distribution of the classes. This loss function penalizes predictions that are far from the actual class by taking the logarithm of the predicted probabilities, meaning that incorrect predictions with high confidence are penalized more severely.
For example, in a binary classification task, When the model's prediction ($\hat{y}$) is far from the actual class (y), the log term becomes large, leading to a higher loss value. This encourages the model to adjust its weights to better match the true class distribution. The same principle extends to multi-class classification using a generalized cross-entropy formula with the softmax function to handle multiple output classes.

Mean Squared Error (MSE) for Regression: The Mean Squared Error (MSE) is better suited for regression tasks, where the goal is to predict a continuous value rather than a discrete class.
The squared difference penalizes the model based on how far the prediction is from the true value. Since MSE directly measures the distance between the predicted value and the actual value, it is ideal for tasks where outputs are continuous and can take on any value within a range. MSE ensures that small errors are penalized less, while large errors are penalized more, leading the model to minimize the average squared deviation.

Why They Are Not Interchangeable:

- **Cross-Entropy in Regression:** Cross-entropy is not suitable for regression because it is designed for probability distributions and categorical outcomes. In regression, the output is a continuous variable, and the concept of probability distribution over classes doesn't apply.
- **MSE in Classification:** in classification we work with a very particular set of possible output values thus MSE is badly defined (as it does not have this kind of knowledge thus penalizes errors in incompatible way).

1.3.3 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 (Batch Gradient Descent):

- **Advantages**: The gradient is computed over the entire dataset, leading to a stable and accurate direction for the gradient update.
- **Disadvantages**: It can be computationally expensive, especially for large datasets, as it requires evaluating the entire dataset before each update. It can also be slow to converge due to the large amount of data processed at each iteration.

Mini-Batch Stochastic Gradient Descent (Mini-Batch SGD):

- **Advantages**: Provides a balance between the stability of classic gradient descent and the efficiency of online stochastic gradient descent. It allows for faster convergence as it updates the model more frequently than classic gradient descent. It also reduces the variance in gradient updates, leading to a smoother and more reliable convergence.
- **Disadvantages**: The choice of batch size can influence performance and convergence speed.

Online Stochastic Gradient Descent (SGD):

- **Advantages**: The model is updated after each training example, leading to quick iterations and faster updates. It can escape local minima due to the noisy nature of updates.
- **Disadvantages**: High variance in gradient updates, which may cause the model to oscillate around the minimum, making it less stable and harder to tune, especially the learning rate.

The most reasonable choice in general is **Mini-Batch SGD** because it provides a good balance between stability and efficiency. It is widely used in practice, especially with modern deep learning frameworks.

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

The learning rate η controls the size of the steps taken during gradient descent:

- **If η is too large**: The updates might overshoot the minimum, leading to divergence or oscillation around the minimum.

- **If η is too small**: The model will converge slowly, potentially requiring a large number of iterations to reach an optimal, and also may stuck in a local minima.

Choosing an appropriate learning rate is crucial for efficient and stable convergence. There are some favored ways to dynamically adjust η during training. as LR schedules or adaptive LR (e.g., using optimizers like Adam).

1.3.5 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**: The naive approach calculates gradients independently for each parameter, leading to a complexity of $O(n^2)$ for a network with n parameters, as each layer's gradient computation repeats the calculation from the previous layer. This approach is inefficient as it does not reuse previously computed gradients, resulting in redundant computations.
- **Backpropagation**: Backpropagation uses the chain rule to propagate the gradients backward through the network, reusing the previously computed gradients for subsequent layers. This reduces the complexity to $O(n)$. This efficient reuse of gradients makes backpropagation the preferred method for training deep neural networks, allowing for faster computation and scalability to larger networks.

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

For backpropagation and gradient descent optimization to work effectively, the network architecture must satisfy the following criteria:

- **Differentiability**: All functions within the network (activations, transformations) must be differentiable (must have derivatives that exist and can be computed), ensuring that gradients can be computed for each parameter.
- **Sequential Layers**: The architecture should allow the application of the chain rule, meaning that the network layers should be organized sequentially, allowing the output of one layer to be the input of the next.
- **Activation Functions**: Non-linear activation functions (e.g., ReLU, tanh) should be used to enable the network to learn complex patterns, ensuring that the model is not reduced to a single linear transformation.

1.3.7 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:

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

To show this, recall the cross-entropy loss function combined with the SoftMax output:

$$\ell(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

Given $\hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}$, substitute into the cross-entropy formula:

$$\ell = -\sum_i y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}\right)$$

$$= -\sum_i y_i \left[\tilde{y}_i - \log \left(\sum_j e^{\tilde{y}_j} \right) \right]$$

$$= -\sum_i y_i \tilde{y}_i + \sum_i y_i \log \left(\sum_j e^{\tilde{y}_j} \right)$$

Since $\sum_i y_i = 1$ (for one-hot encoded labels), the second term simplifies:

$$\ell = -\sum_i y_i \tilde{y}_i + \log \left(\sum_j e^{\tilde{y}_j} \right)$$

1.3.8 15. Write the gradient of the loss (cross-entropy) relative to the intermediate output $\tilde{y}$: $\frac{\partial \ell}{\partial \tilde{y}_i}$.

To find $\frac{\partial \ell}{\partial \tilde{y}_i}$, we take the derivative of this expression with respect to $\tilde{y}_i$:

- The derivative of the first term $-\sum_i y_i \tilde{y}_i$ is:

$$\frac{\partial}{\partial \tilde{y}_i} \left(-\sum_i y_i \tilde{y}_i \right) = -y_i$$

- The derivative of the second term $\log \left(\sum_i e^{\tilde{y}_i} \right)$ using the chain rule is:

$$\frac{\partial}{\partial \tilde{y}_i} \log \left(\sum_i e^{\tilde{y}_i} \right) = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} = \hat{y}_i$$

Combining these results:

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

This shows that the gradient of the cross-entropy loss with respect to the intermediate output $\tilde{y}_i$ is the difference between the predicted probability $\hat{y}_i$ and the true label y_i.

1.3.9 16. Using the backpropagation, write the gradient of the loss with respect to the weights of the output layer $\nabla W_y \ell$. Note that writing this gradient uses $\nabla \tilde{y} \ell$. Do the same for ∇b_y.

For the weights W_y:

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

So:

$$\nabla W_y \ell = (\hat{y} - y) \cdot h^T$$

For the biases b_y:

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

Thus:

$$\nabla b_y \ell = \hat{y} - y$$

1.3.10 17. Compute other gradients: $\nabla \tilde{h} \ell$, $\nabla W_h \ell$, $\nabla b_h \ell$.

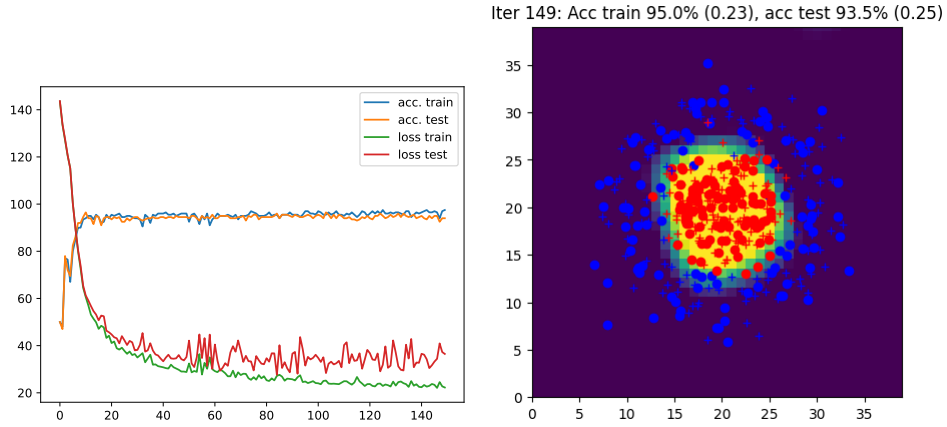
- $\nabla \tilde{h} \ell = (\nabla \tilde{y} \ell) \cdot W_y \cdot \tanh'(\tilde{h})$
- $\nabla W_h \ell = \nabla \tilde{h} \ell \cdot x^T$
- $\nabla b_h \ell = \nabla \tilde{h} \ell$

1.4 Implementation

With all the essential equations established for forward predictions, loss evaluation, and backpropagation through gradient descent, we are now prepared to proceed with the practical implementation of these concepts in PyTorch. Our initial experiments will be conducted on the "Circle" dataset, a popular choice for exploring non-linear classification challenges due to its distribution of data points across two distinct circles.

1.4.1 Forward and backward manuals

In this section, we developed custom forward and backward functions, applying the SGD optimization algorithm with 10 batches and a learning rate of 0.03. Following 150 iterations, the model reached a stable convergence, achieving an impressive 96.5% accuracy on the training set and a robust 95% accuracy on the test set, as illustrated in the figures below:



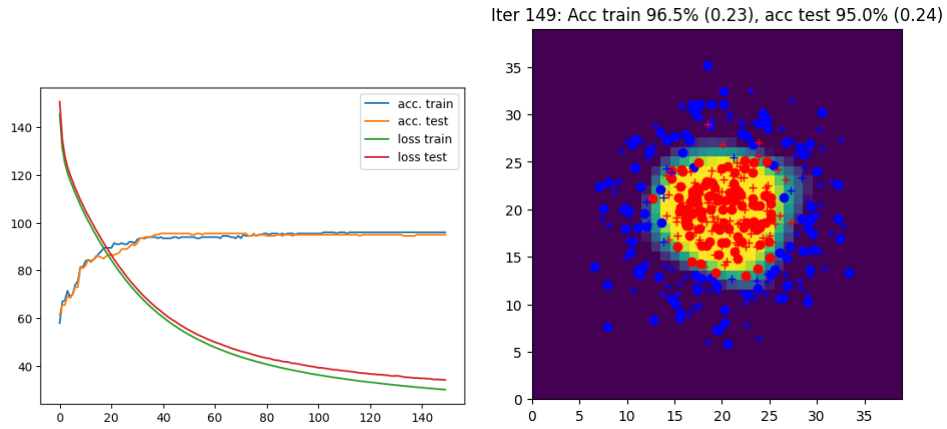
(a) Learning curves **(b)** Decision boundaries

Figure 1.1

1.4.2 Simplification of the backward pass with torch.autograd

In this section, we utilize PyTorch's automatic differentiation, Autograd. Although disabled by default, this feature can be activated by setting `requires_grad=True` when creating a tensor, enabling PyTorch to compute gradients automatically. With a simple `loss.backward()` command, PyTorch calculates the derivatives of the loss with respect to all contributing tensors, storing these gradients in the `grad` attribute (e.g., `W.grad` for a tensor `W`). This enhances our ability to perform automatic differentiation effectively.

After rerunning the training algorithm with Autograd and keeping the same hyperparameters, we obtained similar results: 96.5% accuracy on training and 94.5% on testing, closely matching our previous manual implementation. The training process also appeared slightly more stable, as shown in the figurs below:

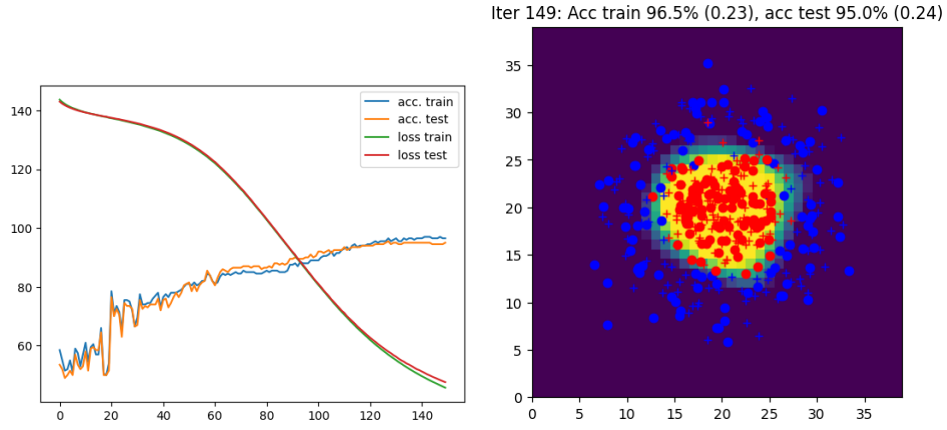


(a) Learning curves **(b)** Decision boundaries

Figure 1.2

1.4.3 Simplification of the forward pass with torch.nn layers

In this section, we further simplify our network architecture using PyTorch's `nn` package, which provides a set of modules for constructing neural networks. By defining the network with `torch.nn`, we can directly perform forward passes through the model without manually implementing a `forward` method.

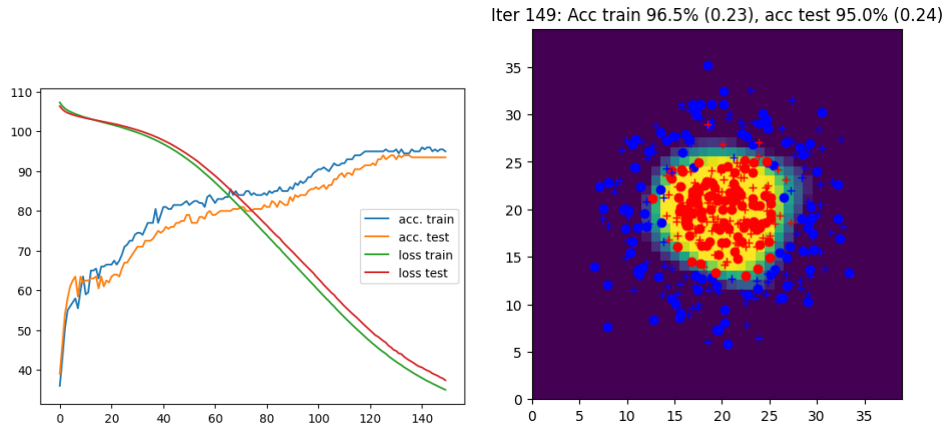


(a) Learning curves **(b)** Decision boundaries

Figure 1.3

1.4.4 Simplification of the SGD with torch.optim

Up to this point, we manually updated parameters during the SGD learning process. However, PyTorch's `torch.optim` module provides various optimizers, including SGD, that automatically handle parameter updates via the `optim.step()` method. This simplifies parameter management and enables access to multiple optimization algorithms for efficient training.



(a) Learning curves **(b)** Decision boundaries

Figure 1.4

1.4.5 Rapport des Expérimentations et Conclusions

1.4.6 Impact du Learning Rate

The study on the impact of the learning rate with a batch size of 32 and different LR values (0.1, 0.03, 0.01, 0.001) over multiple runs (10) to limit variance and calculate the mean showed that this parameter has a major effect on convergence and stability in deep learning models :

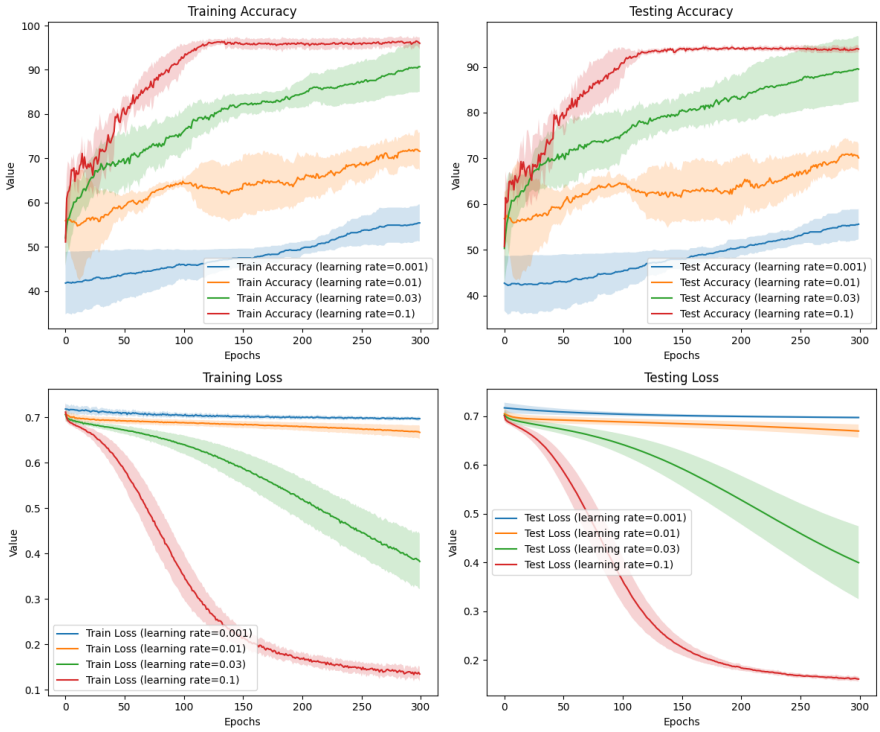


Figure 1.5: Learning curves with different learning rates

1.4.7 Impact of Batch Size

Batch size influences both the stability of learning and computational efficiency, tested with a fixed learning rate :

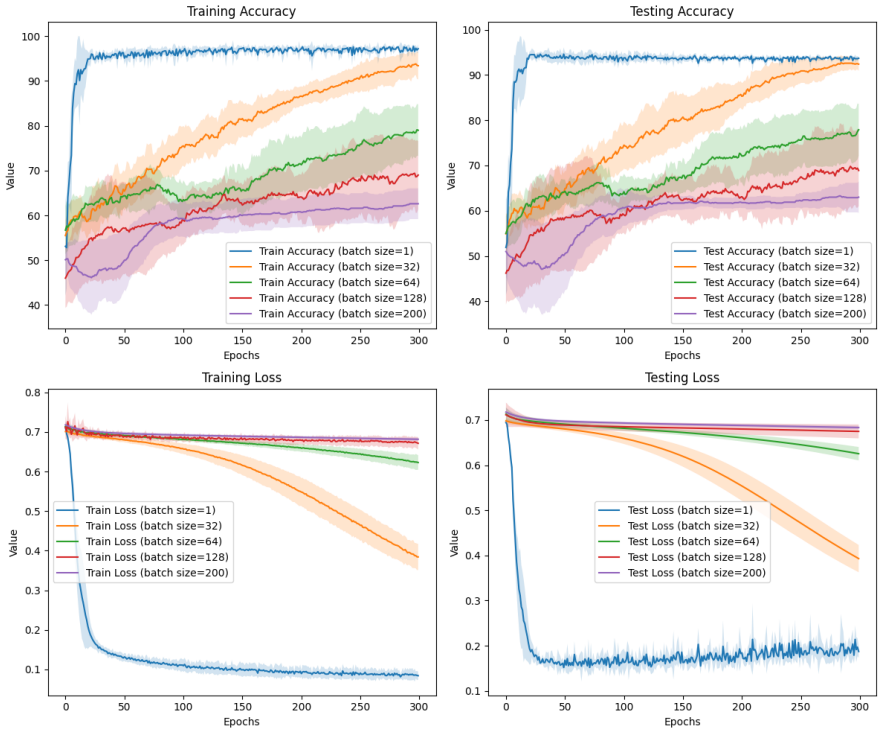


Figure 1.6: Learning curves with different learning rates

1.4.8 Adjustment of the Learning Rate with Batch Size

A dynamic adjustment of the learning rate according to batch size was tested since batch size should have an adapted learning rate, based on the following formula:

$$\text{new\_learning\_rate} = \text{initial\_learning\_rate} \times \frac{\text{initial\_batch\_size}}{\text{new\_batch\_size}}$$

This approach allows the learning rate to be adapted for larger batch sizes, compensating for the loss of stochastic noise and maintaining convergence speed.

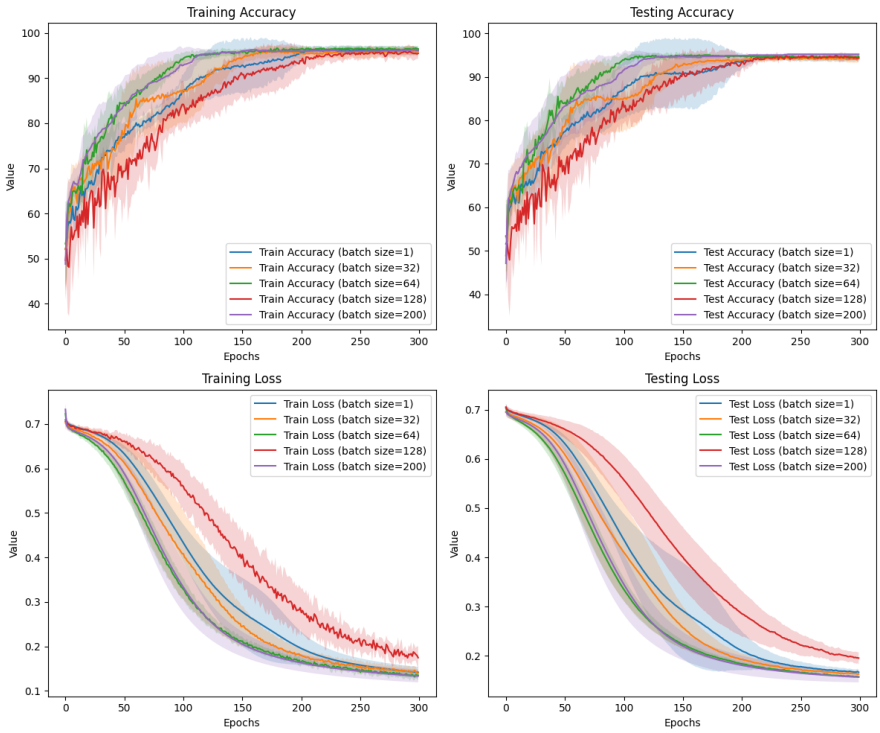


Figure 1.7: Learning curves with different learning rates

1.4.9 MNIST Data Analysis

Applying classification models to the MNIST dataset provided insights into the performance of a simple image classification model:

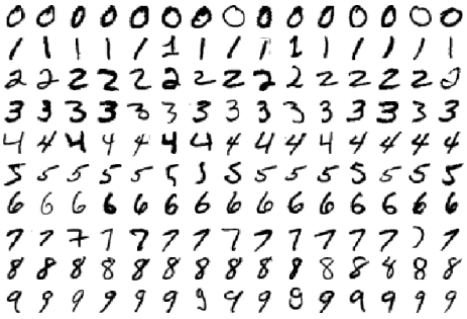


Figure 1.8: Mnist dataset examples

- **Training Accuracy** : The model demonstrated good training accuracy, confirming its ability to capture discriminative patterns.
- **Test Accuracy** : Test accuracy allowed us to evaluate the model's generalization. Confusion matrices revealed specific errors, such as confusion between similar digits (e.g., between 3 and 8).

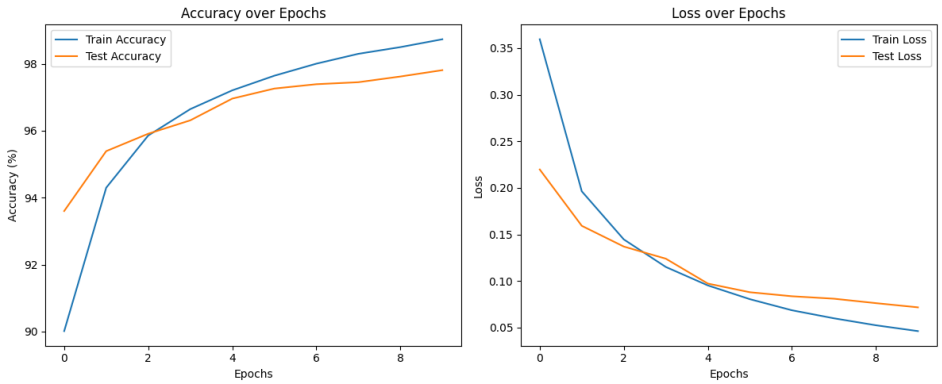
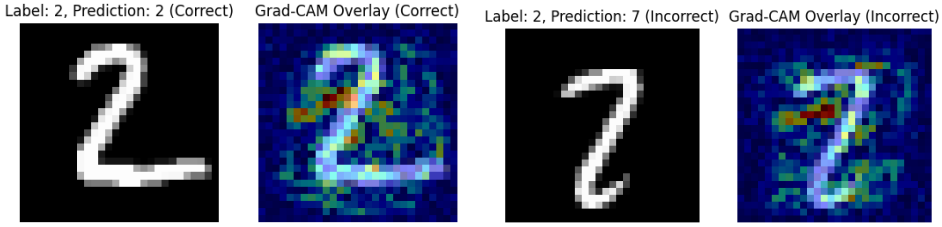


Figure 1.9: Learning curves with mnist dataset

- **Grad-CAM Analysis** : Using Grad-CAM helped identify image regions contributing most to each prediction. This revealed attention biases on certain parts of the digits, showing the key areas the model uses for classification.



(a) Grad-CAM Analysis with correct prediction

(b) Grad-CAM Analysis with wrong prediction

Figure 1.10: Grad-CAM AnalysisGrad-CAM Analysis

1.4.10 SVM Experiments (Circles Dataset)

The study of SVMs on the Circles dataset (non-linearly separable) allowed us to compare different kernels and examine the effect of regularization:

- **Linear SVM** : A linear separation was inadequate due to the concentric structure of the data, resulting in low accuracy.

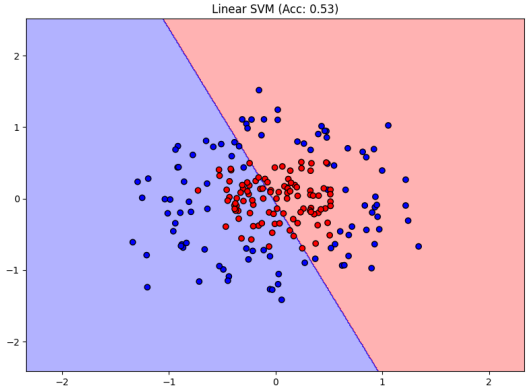


Figure 1.11: Decision Boundaries of a Linear SVM

- **Complex Kernels** :
  - **Polynomial Kernel** : Although it improved separation compared to the linear kernel, its performance was lower than that of the RBF kernel, as lower-degree polynomials are not well-suited to capture the circular structure.

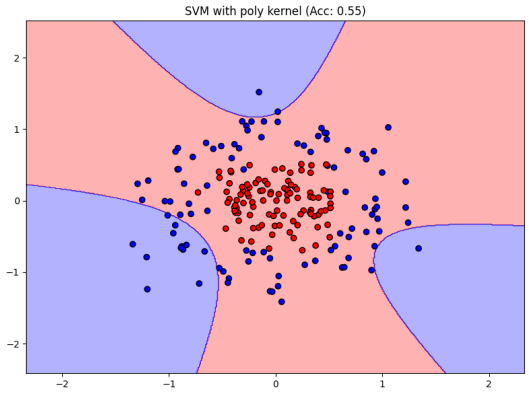


Figure 1.12: Decision Boundaries of a Polynomial Kernel

  - **RBF (Radial Basis Function) Kernel** : The RBF kernel was particularly well-suited to the Circles dataset, enabling effective separation by mapping the data into a higher-dimensional space where it becomes linearly separable. This kernel provided the best results, achieving high accuracy.

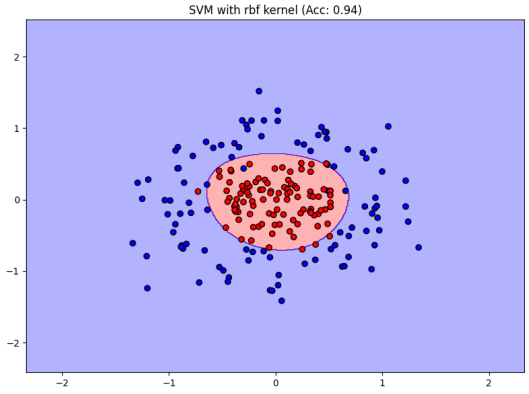


Figure 1.13: Decision Boundaries of a RBF Kernel

  - **Sigmoid Kernel** : This kernel, which simulates a sigmoid activation function, performed moderately. It was able to capture non-linear patterns to some extent but did not achieve the same accuracy as the RBF kernel.

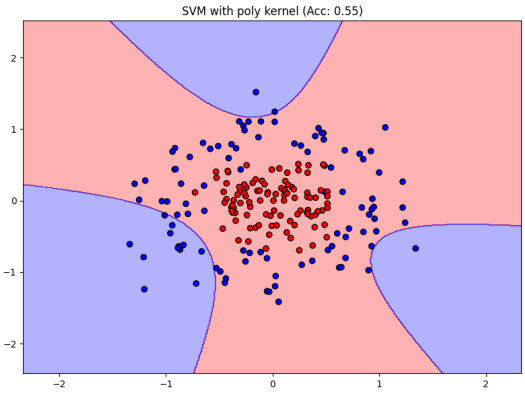
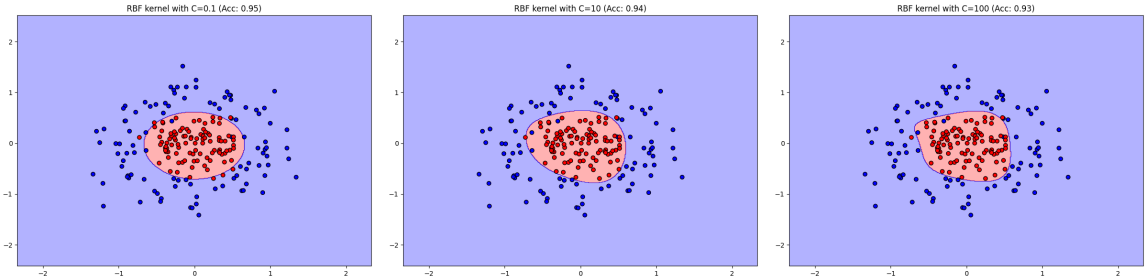


Figure 1.14: Decision Boundaries of a Sigmoid Kernel

- **Impact of the Regularization Parameter** C :
  - The C parameter controls the balance between maximizing the margin and minimizing classification error. Low C values favor a wider margin even if some classification errors are allowed, while high C values aim to reduce misclassification, potentially at the risk of overfitting.



(a) Value C = 0.1 **(b)** Value C = 10 **(c)** Value C = 100

Figure 1.15: Decision boundaries for different values of C.

Chapter 2

Introduction to convolutional networks

2.1 Part 1 - Introduction to Convolutional Networks

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

- **Output Size:** The output dimensions of the convolution operation depend on the input size (x, y, z), padding p, stride s, and kernel size k. The output width (n'_x) and height (n'_y) can be calculated using:

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

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

  The depth of the output feature map remains as the number of filters C applied.

- **Number of Weights to Learn:** If we have C filters, each of size $k \times k \times z$, the total number of weights to learn is:

$$C \times (k \times k \times z) + C$$

  The additional C is for the biases for each filter.

- **Weights for a Fully-Connected Layer:** If a fully-connected layer produced an output of the same size, it would require learning:

$$(x \times y \times z) \times (n'_x \times n'_y \times C) + (n'_x \times n'_y \times C)$$

  This shows that convolutional layers significantly reduce the number of weights compared to fully-connected layers, especially as the input size increases.

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

- **Advantages:**
  - **Parameter Efficiency:** Convolutional layers share weights across spatial dimensions, reducing the number of parameters compared to fully-connected layers.
  - **Spatial Locality:** They capture local spatial patterns effectively, which is crucial for image data where nearby pixels are often related.
  - **Translation Invariance:** By using shared weights (kernels), convolutional layers are naturally robust to small translations in the input data.
- **Main Limit:**

* Convolutional layers are limited to learning local patterns. To capture long-range dependencies, deeper networks or additional techniques like global pooling or fully-connected layers are needed.

3. ★ Why do we use spatial pooling?

- **Purpose:**
  * Spatial pooling reduces the spatial dimensions of the feature maps, thereby reducing the computational load and the number of parameters in the subsequent layers.
  * It also provides a form of spatial invariance, making the network less sensitive to small translations or distortions in the input image.
- **Common Types:**
  * **Max Pooling:** Retains the maximum value within a window, highlighting the most prominent feature in each region.
  * **Average Pooling:** Takes the average value within the window, giving an overall representation of each region.

4. ★ Suppose we try to compute the output of a classical convolutional network (e.g., VGG16) for an input image larger than the initially planned size (224×224). Can we (without modifying the image) use all or part of the layers of the network on this image? Yes, we can use the convolutional layers without modification since they operate with a sliding window, meaning they can handle varying input sizes. However, fully-connected layers expect a fixed-size input. For larger input sizes, we can use all the convolutional layers and pooling layers, but the fully-connected layers would require modification or adaptation (e.g., converting them to convolutional layers) to handle variable input sizes.

5. Show that we can analyze fully-connected layers as particular convolutions. Fully-connected layers can be seen as convolutions with The kernel size of the convolutional layer is set to match the entire spatial dimensions of the input, i.e., $k_x = x$ and $k_y = y$. and no padding or stride (i.e., the filter covers the entire spatial dimension). and the number of channels should match the FC layer's output size, This configuration ensures that the convolution operation covers the entire input, effectively mimicking the behavior of an FC layer. The output dimensions will be $1 \times 1 \times N$, which, when flattened, matches the FC layer's output size.

This perspective is useful when converting fully-connected layers to convolutional layers, especially for handling variable input sizes.

6. Suppose that we therefore replace fully-connected layers by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest? If fully-connected layers are replaced with their convolutional equivalents, the network can handle variable input sizes throughout all layers. The output shape will depend on the dimensions of the input image and the architecture specifics (padding, stride, etc.).

- **Interest:**
  * This approach allows for processing images of different sizes and even applying the network as a sliding window detector across large images.
  * The output retains spatial information, which can be valuable for localization tasks.

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? For the **first convolutional layer**,

each neuron's receptive field is determined directly by the size of the convolutional kernel (filter). If the kernel size is $k_1 \times k_1$, then the receptive field R_1 is simply:

$$R_1 = k_1$$

This means that each neuron in the first layer depends on $k_1 \times k_1$ pixels of the input image.

Moving to the **second convolutional layer**, the receptive field expands due to the accumulation of the previous layer's receptive fields and the parameters of the current layer. The receptive field R_2 can be calculated using the following formula:

$$R_2 = R_1 + (k_2 - 1) \times s_1$$

Where:

- R_1 is the receptive field size of the first layer.
- k_2 is the kernel size of the second layer.
- s_1 is the stride of the first layer.

This equation accounts for the fact that each neuron in the second layer is connected to a region in the first layer, which in turn corresponds to a region in the input image.

As we consider **deeper layers**, the receptive field continues to grow, allowing neurons to integrate information from larger portions of the input.

Interpretation:

- In the *early layers*, neurons have small receptive fields and focus on local features such as edges, textures, and simple patterns. - In *deeper layers*, the receptive fields become larger, enabling neurons to capture more complex and global features like shapes, objects, or even entire regions of the image.

This expansion of the receptive field with network depth allows CNNs to build hierarchical feature representations, transitioning from low-level to high-level abstractions. It explains how CNNs can effectively recognize and interpret complex structures within images by progressively integrating information from larger contexts.

2.2 Part II - Training from scratch of the models

2.2.1 2.1 Network Architecture

The network to be implemented is a modified AlexNet for the CIFAR-10 dataset, consisting of the following layers:

- `conv1`: 32 convolutions of 5×5, followed by ReLU
- `pool1`: max-pooling 2×2
- `conv2`: 64 convolutions of 5×5, followed by ReLU
- `pool2`: max-pooling 2×2
- `conv3`: 64 convolutions of 5×5, followed by ReLU
- `pool3`: max-pooling 2×2
- `fc4`: fully-connected layer with 1000 output neurons, followed by ReLU
- `fc5`: fully-connected layer with 10 output neurons, followed by softmax

Questions and Answers

8. For the convolutions, we want to keep the same spatial dimensions for the output as the input. What values of padding and stride are needed?

To keep the output dimensions the same as the input dimensions in a convolutional layer, we need to calculate the padding and set the stride accordingly. The output dimension n' can be calculated as:

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

where:

- n is the input dimension,
- k is the kernel size,
- p is the padding, and
- s is the stride.

Since we want the output size n' to be equal to the input size n, we set $n' = n$ and rearrange the equation to solve for p:

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

Solving for p, we have:

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

In our case:

- We have $k = 5$,
- We set $s = 1$ to keep the stride consistent with the input size.

Substituting these values:

$$p = \frac{(1-1) \cdot n + 5 - 1}{2} = \frac{4}{2} = 2$$

Thus, a padding of 2 and a stride of 1 will keep the output dimensions equal to the input dimensions.

9. For max-pooling, we want to reduce the spatial dimensions by a factor of 2. What values of padding and stride are required? A pooling layer operates similarly to a convolution layer, so we can proceed as in question 8. This time, however, we want the output dimensions n' to be half of the input dimensions n, so $n' = \frac{n}{2}$.

Using the pooling formula:

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

we substitute $n' = \frac{n}{2}$ and rearrange the formula to solve for p:

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

Solving for p:

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

In this case:

– We use a pooling kernel size $k = 2$,
– To achieve the desired output size, we set the stride $s = 2$.

Substituting these values, we have:

$$p = \frac{(2-1) \cdot n + 2 - 2}{2} = 0$$

Thus, to reduce the dimensions by half, we set the pooling layer parameters to a kernel size $k = 2$, stride $s = 2$, and padding $p = 0$.

10. ★ For each layer, indicate the output size and the number of weights to learn. Comment on this distribution. Below is an overview of each layer with its output size and number of weights:

– `conv1`: Output $32 \times 32 \times 32$, Weights: $(5 \times 5 \times 3 + 1) \times 32 = 2432$
– `pool1`: Output $16 \times 16 \times 32$, no weights (pooling has no learnable parameters).
– `conv2`: Output $16 \times 16 \times 64$, Weights: $(5 \times 5 \times 32 + 1) \times 64 = 51264$
– `pool2`: Output $8 \times 8 \times 64$, no weights.
– `conv3`: Output $8 \times 8 \times 64$, Weights: $(5 \times 5 \times 64 + 1) \times 64 = 102464$
– `pool3`: Output $4 \times 4 \times 64$, no weights.
– `fc4`: Output 1000, Weights: $(4 \times 4 \times 64 + 1) \times 1000 = 1025000$
– `fc5`: Output 10, Weights: $(1000 + 1) \times 10 = 10010$

Comment: The majority of weights to learn are in the fully-connected layers, especially in `fc4`, which has over a million parameters. This makes the model more prone to overfitting on CIFAR-10 unless regularization techniques are applied.

11. What is the total number of weights to learn? Compare it to the number of examples. The total number of weights is the sum of weights across all layers:

$$2432 + 51264 + 102464 + 1025000 + 10010 = 1191170$$

The CIFAR-10 training set contains 50,000 images, meaning the network has far more parameters than examples. This can lead to overfitting if the model is not properly regularized or if data augmentation techniques aren't used.

12. Compare the number of parameters to learn with that of the BoW and SVM approach. In a Bag of Words (BoW) model:

– **Feature Representation**: A BoW model represents each image as a vector of feature counts or frequencies, ignoring spatial relationships. This typically results in a much smaller feature vector.
– **SVM Model**: An SVM with a linear kernel is used to classify based on these features. SVMs generally have far fewer parameters because they only require a set of weights for each feature in the input vector, not the hierarchical structure seen in CNNs.

For example:

– Suppose we use a BoW vector with 500 features per image. For a linear SVM, we would need 500 weights (plus potentially one bias term per class, depending on the SVM setup).
– Comparing this to CNNs, a BoW + SVM approach would have around **500 parameters**, which is **significantly fewer** than the 1,191,170 parameters of a CNN.

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 main difference is that we do not pass an optimizer to the `epoch()` function. As a result, the network is not trained on the test examples, and no backward pass is performed to calculate or update the loss.

16. ★ What are the effects of the learning rate and of the batch-size? The learning rate and batch size affect model training speed, stability, and resource requirements.

- **Learning Rate**: A higher learning rate can speed up convergence but risks instability, while a lower rate provides stable, gradual updates, potentially improving convergence at the cost of longer training times.
- **Batch Size**: Large batch sizes yield stable gradients, which can enhance convergence but demand more memory. Smaller batch sizes introduce randomness, which may help escape local minima but may also slow convergence.

Learning rate and batch size are closely connected; changes in batch size often require a corresponding adjustment in learning rate to ensure effective training. Typically, a linear scaling approach is used: if the batch size is multiplied by a factor of k, the learning rate should also be scaled by k, with other hyperparameters held constant, as shown in Figure 2.3.

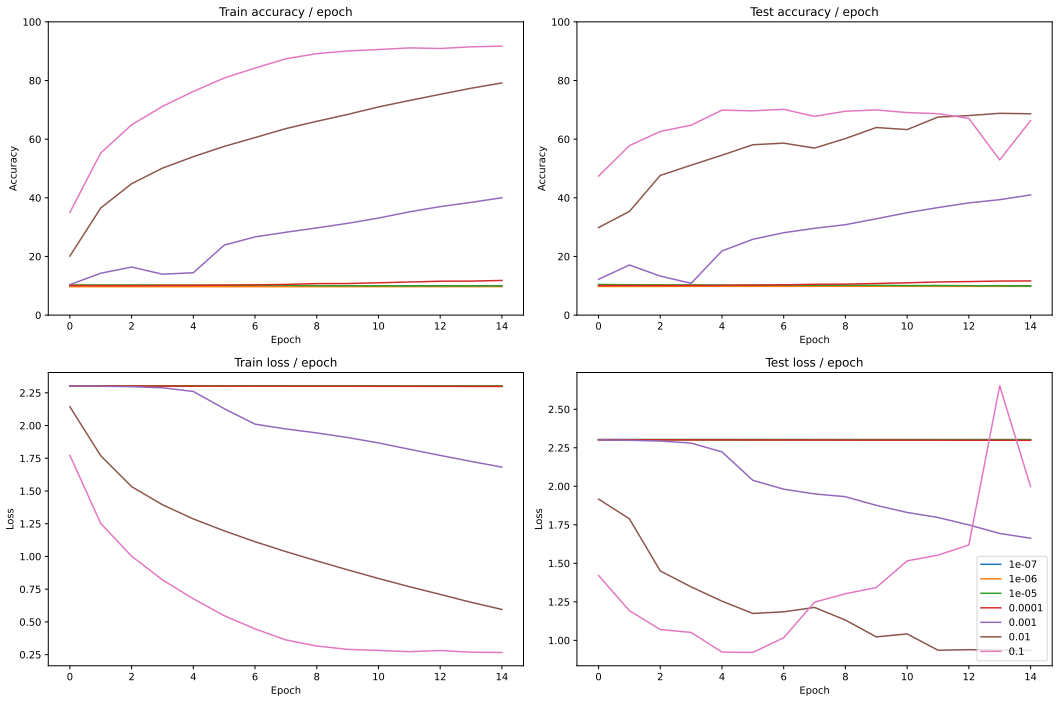


Figure 2.1: Influence of learning rate with a batch size fixed at 32

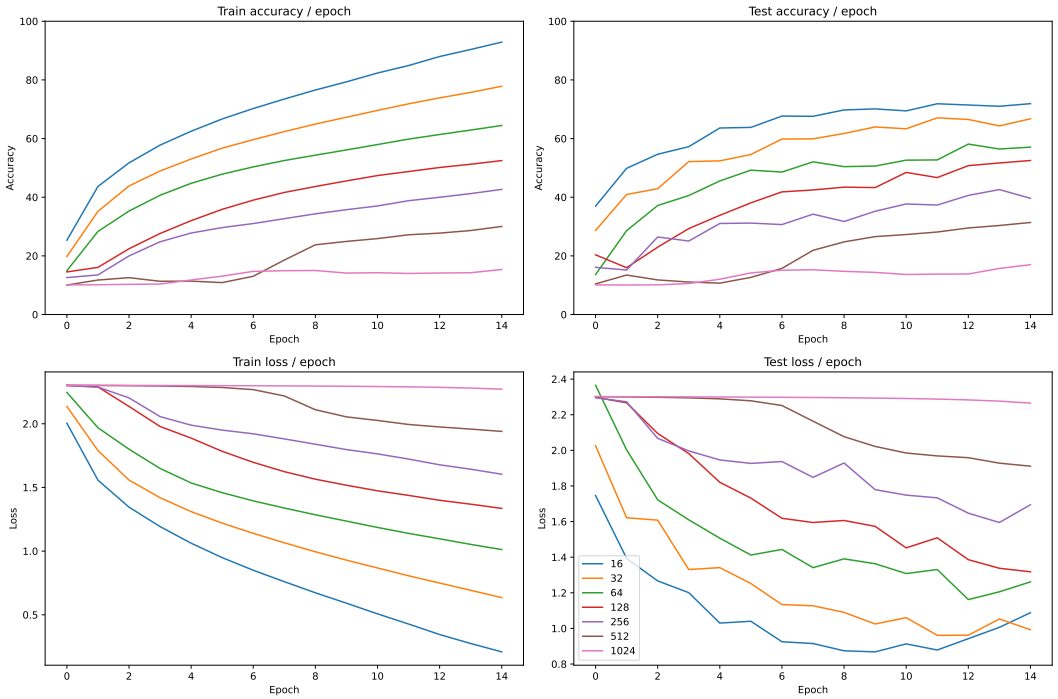


Figure 2.2: Influence of batch size rate with a learnig rate fixed at 0.01

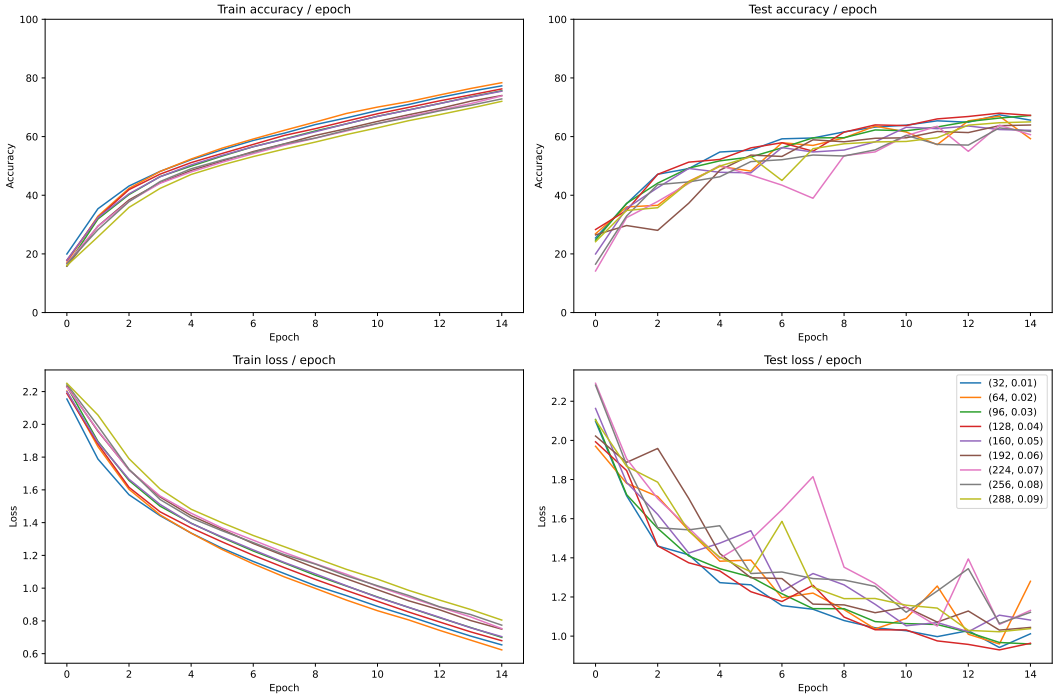


Figure 2.3: Relationship between batch size and learning rate

17. What is the error at the start of the first epoch, in train and test? How can you interpret this? At the beginning of the first epoch, the training and test errors reflect the initial random weight initialization of the network.

18. Interpret the results. What's wrong? What is this phenomenon? Overfitting is observed, becoming evident around the thirtieth epoch when test accuracy plateaus, while training accuracy continues to improve. This indicates that although the model's performance on the training data is increasing, it struggles to generalize effectively to unseen images.

2.3 Results improvements

In the following experiments, our efforts centered on refining the model by exploring various regularization techniques, optimization strategies, and data augmentation methods. We primarily tested each component individually to avoid introducing bias, though some techniques were combined when beneficial. This approach aimed to identify best practices for designing and training CNNs effectively, while addressing common challenges such as overfitting and instability during training.

2.3.1 Standardization of examples

19. Describe your experimental results. Standardization plays a crucial role in addressing inconsistencies in the scales and ranges of raw data, leading to a more uniform data distribution and reducing issues with numerical instability. This process enhances model training by allowing the network to learn faster and more consistently. In our experiments, standardization led to a noticeable improvement in training speed, with the model reaching convergence 20 epochs sooner compared to unstandardized data. Additionally, we observed more stable trends in both loss and accuracy throughout training, indicating smoother and more reliable model updates.

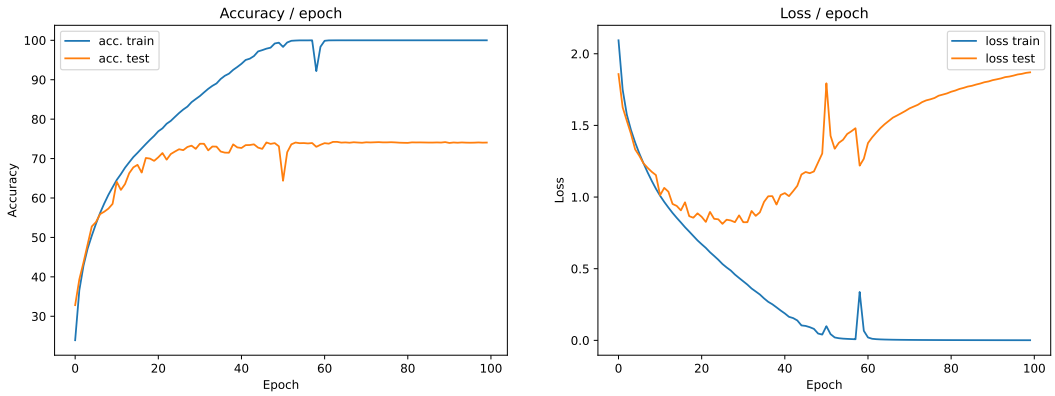


Figure 2.4: Accuracy and losses in train and test using standardization

20. Why only calculate the average image on the training examples and normalize the validation examples with the same image? To ensure an unbiased assessment of the model's generalization ability, it is critical to evaluate performance on validation data processed in the same way as training data, without including any specific information from the validation set itself. Computing statistics (such as means and standard deviations) from the validation data would lead to "data leakage," giving the model unintended insights into the validation set and potentially inflating performance metrics. Avoiding data leakage is essential for an accurate and impartial evaluation of the model's ability to perform on unseen 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. here are some normalization methods we have prior knowledge about (not tested) :

– **PCA/ZCA Whitening**: These linear algebra-based methods decorrelate features, making them less redundant and potentially improving learning efficiency. ZCA maintains a representation similar to the original data, unlike PCA, which completely alters it. Despite their benefits, both techniques can be computationally expensive.

– **L1/L2 Normalization**: This technique scales image pixel values by dividing each pixel by the L1 norm (sum of absolute pixel values) or the L2 norm (square root of the sum of squared pixel values). L1/L2 normalization is beneficial when the focus is on the relative importance of pixel values, rather than their absolute magnitudes.

– **Min-Max Scaling**: This method rescales pixel intensity values based on the minimum and maximum values in the dataset. Although simple, it may be less effective with outliers.

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

22. Describe your experimental results and compare them to previous results. Expanding our dataset with synthetic variations helps the model become more versatile, as it learns from a broader range of examples. By applying techniques like random cropping and flipping, we add subtle shifts and transformations that simulate real-world diversity, pushing the model to identify features beyond the exact patterns in the training set.

Random crops encourage the model to interpret different sections of an image, preventing an over-reliance on specific regions. Meanwhile, horizontal flips train the model to recognize features regardless of orientation. Together, these adjustments introduce a controlled degree of randomness that improves the model's ability to handle unfamiliar data, making it more likely to perform well in varied conditions and less prone to overfitting.

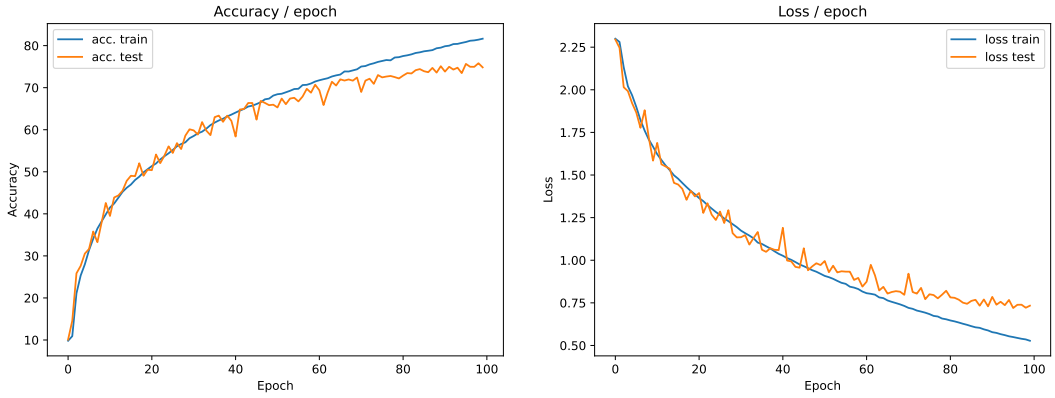


Figure 2.5: Accuracy and losses in train and test using data augmentation

The results of our experiments reveal that the model is better at generalizing, with overfitting notably reduced, which is a promising step forward. However, testing still shows occasional fluctuations, indicating that some instability persists in the model's performance.

23. Does this horizontal symmetry approach seems usable on all types of images? In what cases can it be or not be? Horizontal flipping can be a helpful augmentation method, as it usually doesn't alter the inherent features of the object. For instance, a natural scene or a common object often remains recognizable even when mirrored, making horizontal flips effective for improving model robustness in these contexts without confusing the learning process.

However, in cases where orientation is vital to an object's identity or the context of the scene, horizontal flipping may be inappropriate. For example, in applications where directional cues are essential—such as facial recognition or certain types of technical imaging—flipping could mislead the model and impact classification.

24. What limits do you see in this type of data increase by transformation of the dataset? Firstly, synthetic transformations (such as flipping, cropping, and rotation) merely expand existing data and may not fully capture the diversity of real-world scenarios, especially for complex or rare features. This can lead to a model that struggles with genuinely novel data. Additionally, excessive transformations may result in overfitting to specific augmented patterns that don't generalize well outside the training set. In some cases, transformations might even introduce unrealistic variations, potentially confusing the model rather than enhancing its adaptability.

Moreover, augmentation techniques are often task-specific; not all transformations are appropriate for every dataset, and certain transformations can distort meaningful features or create misleading information. This can limit the broader applicability of data augmentation and its potential to fully address dataset variability.

25. Bonus: Other data augmentation methods are possible. Find out which ones and test some. **rotation**, **zoom**, **translation**, and **shearing**, each of which introduces positional or scale variations to help models recognize objects in diverse orientations and placements. **Gaussian noise** and **brightness/contrast adjustments** simulate different lighting and image quality conditions, making the model more resilient to variations in image appearance. **Cutout** masks parts of images, forcing the model to focus on diverse features, while **elastic distortion** and **channel shuffling** alter pixel arrangements to handle deformation and color variation.

A unique technique is **mosaic augmentation**, which combines portions of four different images into one. This method introduces high variability and complex background context in a single image, allowing models to better detect objects in cluttered or diverse scenes. Mosaic is particularly effective in object detection tasks, where it can enhance feature extraction across varied environments.

2.3.3 Variants of the optimization algorithm

26. Describe your experimental results and compare them to previous results, including learning stability. Using a learning rate scheduler allows us to control the model's pace of learning, helping it adapt more effectively to the task. By gradually reducing the learning rate, we guide the model toward a smoother, steadier convergence to the global minimum of the loss function. This gradual adjustment also helps the model avoid flat regions in the loss landscape, where training can stall with minimal accuracy improvement. Ultimately, a learning rate schedule promotes better training stability and consistency.

Our results, shown in Figure 2.6, highlight this stability, with the loss curve demonstrating smoother progress compared to the baseline, which lacked such scheduling.

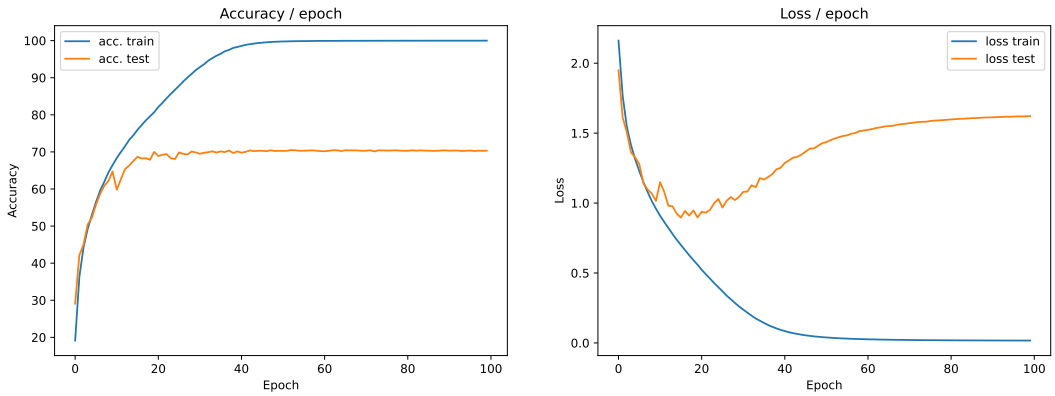


Figure 2.6: Accuracy and losses using an exponential decay scheduler with SGD

27. Why does this method improve learning? The effectiveness of exponential decay in learning rate scheduling lies in its ability to adapt dynamically to the evolving demands of the training process. At the outset, when the model's parameters are far from optimal, a higher learning rate allows for faster progression by enabling larger, more impactful updates. This helps the model explore a broader parameter space. As training continues, the focus shifts from broad exploration to precise adjustments. By gradually reducing the learning rate, the model can make finer updates, essential for achieving a well-tuned solution.

This approach also acknowledges the varying landscape of the loss function, which may include steep regions, flat areas, and complex curvatures. In areas of steep descent, larger learning rates are beneficial for efficient progress. Conversely, in flatter or more intricate sections of the loss surface, smaller updates are crucial for honing in on the minimum without overshooting. Exponential decay inherently adjusts the learning rate to these diverse regions, promoting a more refined training process overall.

However, selection of both the initial learning rate and the decay factor is critical. If the decay occurs too quickly, the learning rate may drop prematurely, preventing the model from sufficiently exploring the parameter space. On the other hand, a slower decay could keep the learning rate high for too long, causing instability, oscillations, or even failure to converge properly. Thus, balancing these parameters is essential to achieving both effective and efficient training.

2.3.4 Regularization of the network by *dropout*

29. Describe your experimental results and compare them to previous results. Introducing a dropout layer into a neural network serves as a powerful tool to prevent overfitting, especially in dense, fully connected layers with numerous parameters. Dropout randomly disables a subset of neurons during each training step, which compels the network to rely on a broader range of features and reduces dependency on specific activations. This technique promotes generalization, helping the model perform better on unseen data by making it less prone to memorizing training examples.

Our results, as seen in Figure 2.7 and Figure 2.8, show that dropout led to meaningful reductions in overfitting, improving the network's overall performance. When paired with a learning rate scheduler, dropout achieved the highest performance gains in our tests. This combined approach likely benefits from dropout's regularization effect alongside the learning rate scheduler's role in fine-tuning the training process, resulting in smoother convergence and improved resilience against overfitting.

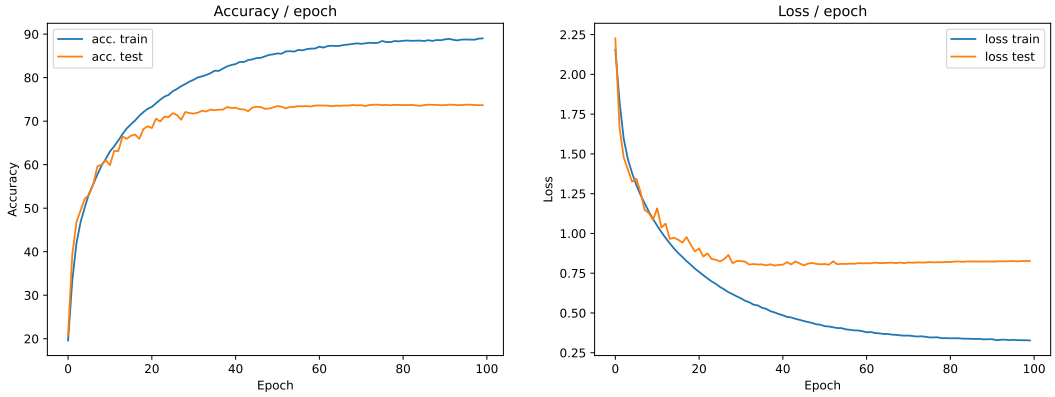


Figure 2.7: Accuracy and losses in train and test, using a dropout on `fc4`

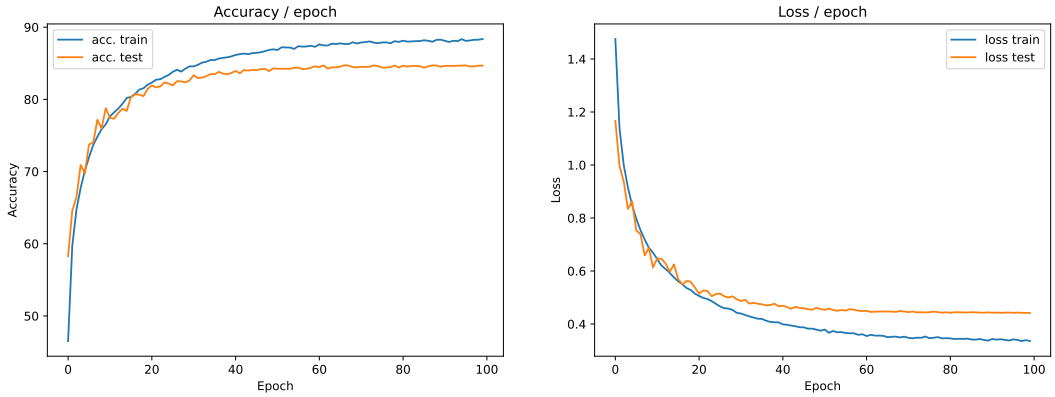


Figure 2.8: Accuracy and losses in train and test, using a dropout on `fc4` and a scheduler

30. What is regularization in general? Regularization is a technique used to prevent overfitting by adding a penalty to the model's complexity. The main goal is to discourage the model from becoming overly complex or excessively tailored to the training data, which can lead to poor generalization on new, unseen data. By adding constraints or adjustments to the learning process, regularization helps the model focus on general patterns rather than specific details of the training data.

Some common types of regularization include:

- **L1 and L2 Regularization**: These methods add a penalty to the loss function based on the size of the model's weights. *L1 regularization* encourages sparsity by penalizing the absolute values of weights, driving some weights to zero, while *L2 regularization* penalizes the squared values of weights, resulting in smaller weights overall without setting them to zero.
- **Dropout**: as said before, dropout randomly "drops out" (or deactivates) a subset of neurons during each training iteration, which forces the model to rely on multiple pathways to make predictions. This reduces the dependence on specific neurons and promotes generalization.
- **Data Augmentation**: Although not traditionally a term in the loss function, data augmentation can be viewed as a regularization technique because it introduces more variability into the training data.
- **Early Stopping**: By monitoring validation performance during training and stopping the process when performance plateaus, early stopping helps prevent the model from fitting noise in the training data, thereby avoiding overfitting.

31. Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it? Dropout encourages neurons to develop stronger and more independent feature representations by making them less reliant on the presence of specific neurons. Without dropout, neurons may become "co-adapted," depending on the outputs of particular neurons, which can lead to overfitting. By randomly deactivating neurons during each training step, dropout breaks up these dependencies, ensuring that each neuron learns features that contribute independently to the model's predictions.

Another way to interpret dropout is as a form of implicit model averaging. In each training pass, a different subset of the network's neurons is active, creating many "thinned" versions of the network. At inference, the complete network uses the aggregate knowledge from these numerous thinner models, which generally enhances robustness and generalization on unseen data.

Dropout can also be viewed as injecting a degree of randomness, or "noise," into the network's hidden layers. This added variability acts as a regularizer, preventing the network from too closely matching the training data, much like stochastic data augmentation techniques.

32. What is the influence of the hyperparameter of this layer? The key hyperparameter in a dropout layer, known as the "dropout rate," determines the probability that a given neuron will be deactivated during training. A high dropout rate means a larger proportion of neurons are randomly dropped in each iteration. While this can help reduce overfitting in complex models, an excessively high dropout rate may hinder learning by discarding too much information, ultimately leading to underfitting.

Conversely, a lower dropout rate keeps more neurons active, which retains more information per iteration. However, if the rate is too low, it may provide insufficient regularization, leaving the model prone to overfitting.

Our experiment, shown in Figure 2.9. illustrates the effect of varying the dropout rate in our architecture. With a minimal dropout rate, the model initially performs well in training but quickly overfits to the training data. Increasing the dropout rate significantly reduces overfitting, resulting in better generalization. setting the dropout rate to $p = 1$, where all neurons are dropped, prevents the model from learning as it has no parameters to update during training.

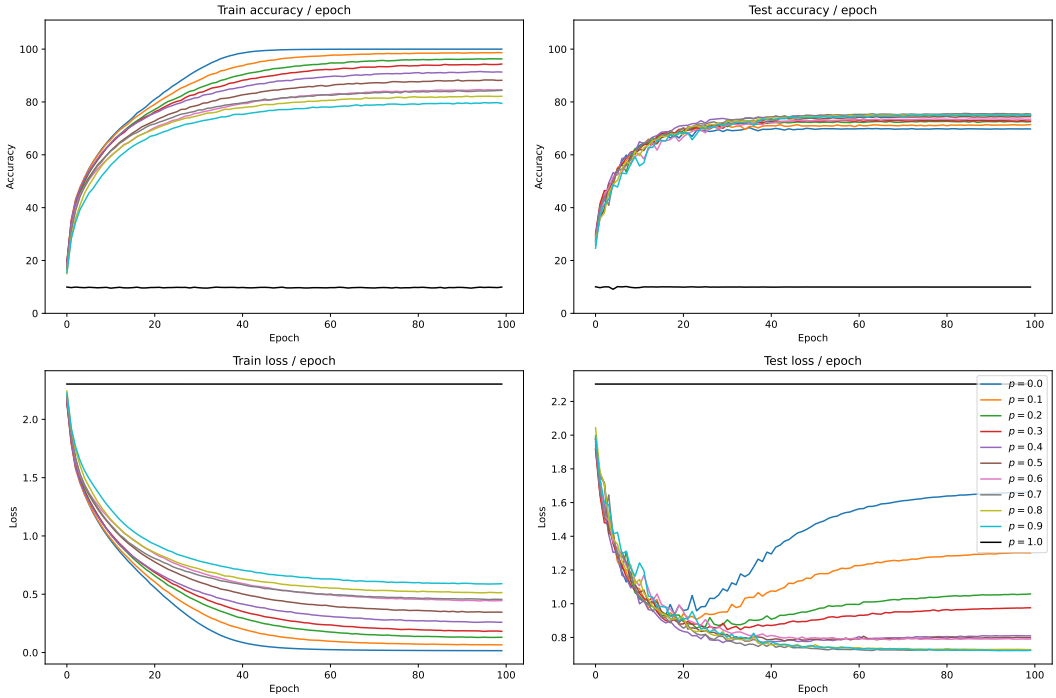


Figure 2.9: Influence of dropout rate p

33. What is the difference in behavior of the dropout layer between training and test? During training, a fraction of neurons are randomly deactivated in each batch, with the dropout rate controlling the proportion of neurons that are temporarily "dropped." This random deactivation occurs with every new batch, allowing the model to learn more robust, independent feature representations as it cycles through varied neuron configurations.

At test time, however, dropout is disabled, meaning all neurons remain active. This approach ensures that the model's predictions on new data are consistent and not influenced by the randomness introduced during training.

2.3.5 Use of *batch normalization*

34. Describe your experimental results and compare them to previous results. We examined the effects of batch normalization, both with and without dropout, while using a learning rate scheduler. Batch normalization addresses internal covariate shift by stabilizing input distributions across layers, which accelerates training and reduces the number of epochs

needed for convergence. This method consistently improves model quality compared to networks without batch normalization.

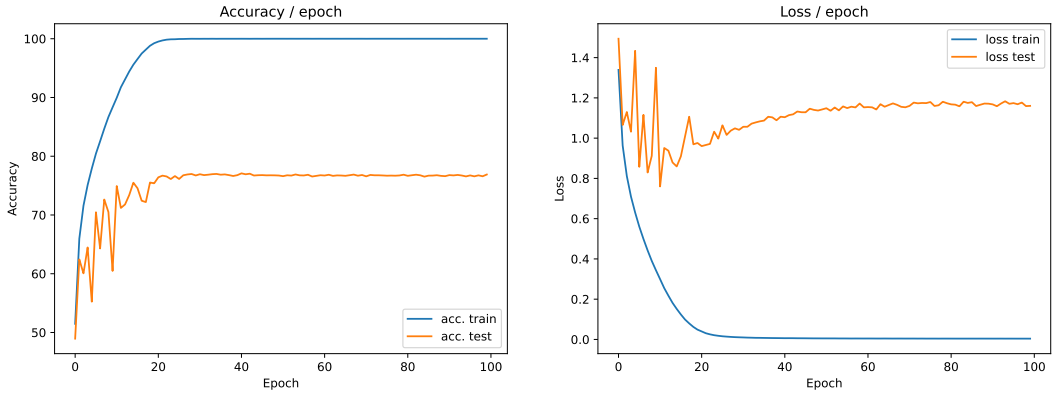


Figure 2.10: Accuracy and losses in train and test using batch normalization and a scheduler

Applying batch normalization (Figure 2.10) improved performance over the baseline (**??**), with enhanced stability observed in the test loss. Notably, the combination of dropout and a learning rate scheduler without batch normalization (Figure 2.8) yielded even stronger results, reaching a test accuracy of approximately 85%, compared to just under 80% with batch normalization alone. Interestingly, combining both batch normalization and dropout showed minimal additional benefit, indicating that batch normalization's regularization effect may reduce the need for further techniques like dropout.

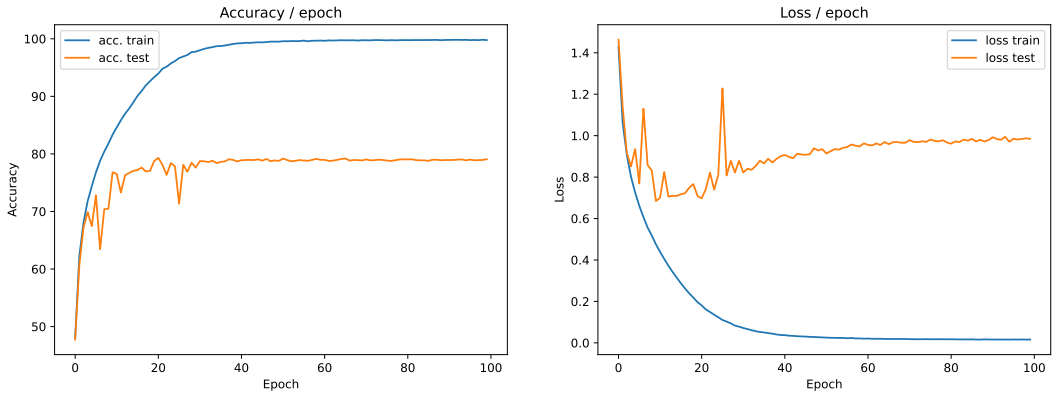


Figure 2.11: Accuracy and losses in train and test, using batch normalization, a dropout layer and a scheduler

Normalizing inputs across layers, batch normalization reduces the risk of extreme values that can cause issues like gradient explosion or vanishing gradients. This added stability is shown in Figure 2.12, where removing the learning rate scheduler still yielded stable results. Batch normalization also enables effective learning with higher initial learning rates; as seen in Figure 2.13, using a rate of 0.1 instead of 0.01 maintained loss within an acceptable range. This highlights batch normalization's ability to make training more less dependent on exact learning rate settings.

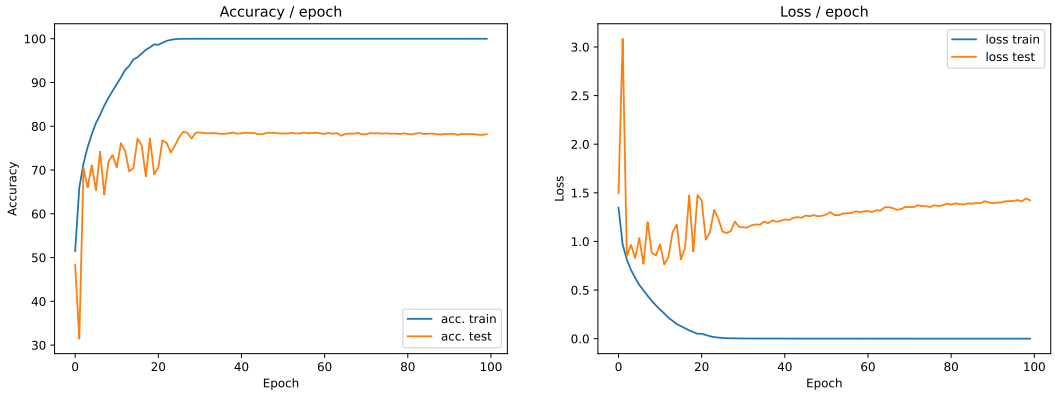


Figure 2.12: Accuracy and losses in train and test, using batch normalization and no learning rate scheduler

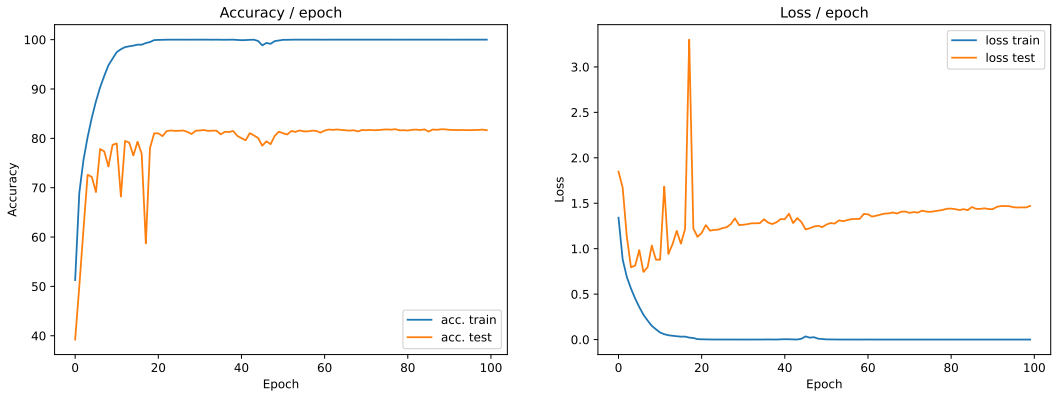
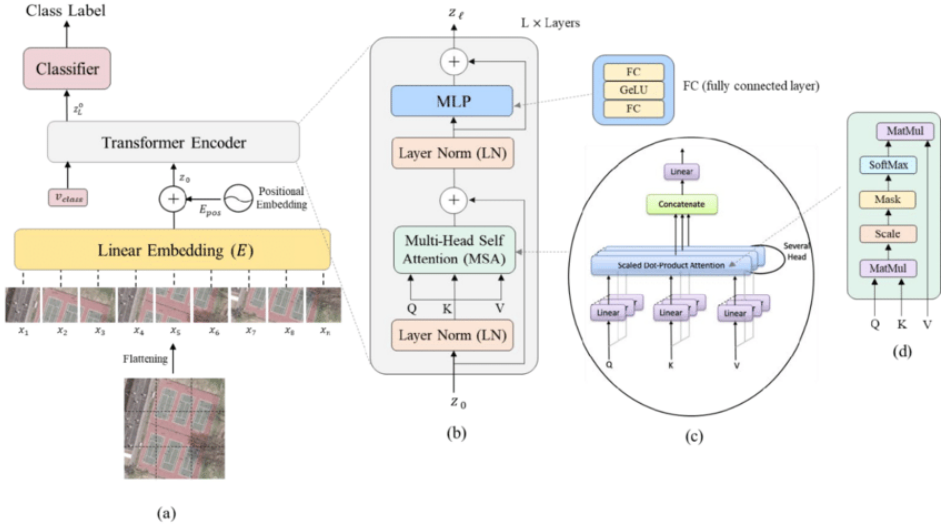


Figure 2.13: Accuracy and losses in train and test, using batch normalization, no learning rate scheduler and a learning rate of 0.1

Chapter 3

Introduction to transformers



The Vision Transformer architecture: (a) the main architecture of the model; (b) the Transformer encoder module; (c) the Multiscale-self attention (MSA) head, and (d) the self-attention (SA) head. Bazi, Yakoub & Bashmal, Laila & Al Rahhal, Mohamad & Dayil, Reham & Ajlan, Naif. (2021). Vision Transformers for Remote Sensing Image Classification. Remote Sensing. 13. 516. 10.3390/rs13030516.

3.1 Self-Attention

What is the main feature of self-attention, especially compared to its convolutional counterpart? What is its main challenge in terms of computation/memory? The main feature of self-attention is its ability to capture global dependencies by allowing each element in an input sequence (here an image) to attend to all other elements. Unlike convolutional operations that focus on local neighborhoods with fixed receptive fields, self-attention provides a flexible mechanism to weigh the influence of distant elements dynamically.

The primary challenge of self-attention in terms of computation and memory is its quadratic complexity with respect to the sequence length. Computing attention weights between all pairs of elements requires significant computational resources and memory storage, making it inefficient for long sequences.

Initially, we will focus on the simple case of single-head attention. To begin, we compute three different linear projections of the input X: the Query Q, the Key K, and the Value V:

$$Q = XW_q,$$
$$K = XW_k,$$
$$V = XW_v,$$

where W_q, W_k, and W_v are the weight matrices for the Query, Key, and Value projections, respectively.

The core of the attention mechanism is the dot product between the Query and Key vectors. These vectors are learned representations that enable the model to determine where to focus its attention. A higher dot product between a Query and a Key indicates greater relevance, guiding the model to pay more attention to specific positions in the input.

However, when the values of the Query and Key vectors are large, their dot products can become excessively large. This can lead to very large values in the softmax function, potentially causing issues like vanishing gradients during backpropagation. To mitigate this, we scale the dot products by dividing by $\sqrt{d_k}$, where d_k is the dimensionality of the Query and Key vectors. This scaling keeps the gradients in a manageable range and maintains consistent variance as d_k changes.

The scaled dot-product attention is computed as:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

The Value matrix V contains the actual content of the input tokens. After computing the attention scores, we use them to weight the Value vectors, effectively aggregating information based on relevance.

3.2 Multi-head self-attention

Write the equations of Multi-Heads Self-Attention.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O.$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$. The Attention function is the same as described in the single-head attention, and W^O is a final linear layer's weights.

3.3 Transfomer block

Write the equations of the transformer block. Let's define the operations within a Transformers block:

1. Let x be the input to the Transformers block, which in our case is a patch from the image plus a positional embedding. We begin by normalizing the input using Layer Normalization (LayerNorm):

$$z = \text{LayerNorm}(x)$$

2. We then compute the Query (Q), Key (K), and Value (V) matrices, which are used in the Multi-Head Attention mechanism as explained in the previous question:

$$Q = z \cdot W_q, \quad K = z \cdot W_k, \quad V = z \cdot W_v$$

3. Next, we pass these matrices Q, K, and V into the Multi-Head Attention mechanism:

$$\text{AttOutput} = \text{MultiHead}(Q, K, V)$$

4. We add the initial input x to the output of the Multi-Head Attention:

$$\text{MidLayerOutput} = z + \text{AttOutput}$$

5. We use another LayerNorm;

$$\text{y} = \text{LayerNorm2}(\text{MidLayerOutput})$$

6. The MLP (Multi-Layer Perceptron) head is a two-layer linear network with a GeLU (Gaussian Error Linear Unit) activation function, defined as follows:

$$\text{MLP}(x) = \text{ReLU}(x \cdot W_1 + b_1) \cdot W_2 + b_2$$

Finally, we pass y into the MLP and add it to the MidLayerOutput to obtain the final output of the Transformers block:

$$\text{MLPOutput} = \text{MLP}(\text{y})$$

$$\text{FinalOutput} = \text{MLPOutput} + \text{MidLayerOutput}$$

After this last addition, we obtain the final output of a Transformers block.

3.4 Full ViT model: Questions

Explain what is a Class token and why we use it? A Class token is a special token added to the input sequence, at the beginning in our case. Its purpose is to capture a summary representation of the entire sequence. During processing, the Class token interacts with all other tokens (other parts of the images) through self-attention mechanisms, aggregating information from them. After passing through the Transformer layers, we use that token in order to classify the image.

Explain what is the positional embedding (PE) and why it is important? Positional encoding is a technique used in Transformer models to inject information about the positions of tokens in a sequence into their embeddings. Since Transformers process all tokens simultaneously and lack inherent sequential order (unlike recurrent models), positional encodings provide the necessary context about token positions. This is important because the meaning in sequences often depends on the order of elements; without positional encoding, the model wouldn't capture the sequence structure essential for understanding context and relationships between tokens. There are 2 main ways to insert PE in a Transformer architecture. You can either learn it, or use a fixed one. We chose the second option with sinusoidal encoding, which provides an unique and easy way to generate encoding for the position.

3.5 Experiments on MNIST: Influence of Hyperparameters

In this section, we analyze the influence of different hyperparameters on the performance of a Transformer model applied to the MNIST dataset. The evaluated metrics include the final accuracy, final loss, and total training time.

3.5.1 Embedding Dimension

We first examine the impact of the embedding dimension on the model's performance. The tested dimensions are 16, 32, and 64. The results are presented in Table 3.1.

Embed_dim	Patch_size	Nb_blocks	Accuracy (%)	Loss	Time (s)
16	4	4	94.33	0.185698	397.67
32	4	4	97.48	0.089253	538.46
64	4	4	97.37	0.077009	949.16

Table 3.1: Impact of the embedding dimension on model performance

We observe that increasing the embedding dimension improves the final accuracy, from 94.33% to 97.48% when the dimension increases from 16 to 32. However, a further increase to 64 only provides a marginal improvement while significantly increasing the training time. Therefore, an embedding dimension of 32 seems to offer a good trade-off between performance and computational cost.

3.5.2 Patch Size

Next, we evaluate the influence of the patch size, testing sizes of 4, 7, and 14 pixels. The results are summarized in Table 3.2.

Embed_dim	Patch_size	Nb_blocks	Accuracy (%)	Loss	Time (s)
32	4	4	97.48	0.089253	538.46
32	7	4	97.20	0.086433	215.45
32	14	4	96.95	0.069291	136.03

Table 3.2: Impact of patch size on model performance

The results indicate that larger patches reduce the training time without significantly compromising accuracy. A patch size of 14 offers a good balance, with reduced training time and a high accuracy of 96.95%.

3.5.3 Number of Transformer Blocks

We explore the effect of the number of Transformer blocks, comparing 1, 2, and 4 blocks. Table 3.3 presents the corresponding performances.

Embed_dim	Patch_size	Nb_blocks	Accuracy (%)	Loss	Time (s)
32	14	1	96.82	0.096762	77.74
32	14	2	97.52	0.075507	97.33
32	14	4	96.95	0.069291	136.03

Table 3.3: Impact of the number of Transformer blocks on model performance

Increasing the number of blocks from 1 to 2 improves the accuracy from 96.82% to 97.52%. However, adding more blocks does not significantly enhance performance and increases the training time. Therefore, two blocks seem sufficient for this model.

3.5.4 Best Configurations

Analyzing all results, the best performances are obtained with the following configurations:

Embed_dim	Patch_size	Nb_blocks	Accuracy (%)	Loss	Time (s)
64	7	4	**97.66**	0.072242	353.61
64	14	2	97.63	**0.061663**	115.93
32	4	4	97.48	0.089253	538.46
32	14	2	97.52	0.075507	97.33

Table 3.4: Configurations offering the best performance

Configurations with an embedding dimension of 64 and a patch size of 7 or 14 offer the highest accuracies but at the cost of increased training time. The configuration with an embedding dimension of 32, a patch size of 14, and 2 blocks presents an excellent compromise between performance and efficiency, with an accuracy of 97.52% and a reasonable training time.

3.5.5 Final Performance Analysis and Improvements

Comment and discuss the final performance that you get. How to improve it? The final performance of our Transformer model on the MNIST dataset achieves an accuracy of up to 97.66%. While this is a strong result for this dataset, there are several ways to potentially improve the model's performance:

- **Data Augmentation**: Applying data augmentation techniques such as rotations, translations, and scaling can increase the diversity of the training data, helping the model generalize better.
- **Hyperparameter Tuning**: Further tuning of hyperparameters like learning rate, batch size, and the number of epochs might lead to improved performance. While limiting the number of epoch to 5 seemed like a reasonable choice considering the learning curves (see below), one could argue that it was the main weakness of our approach.

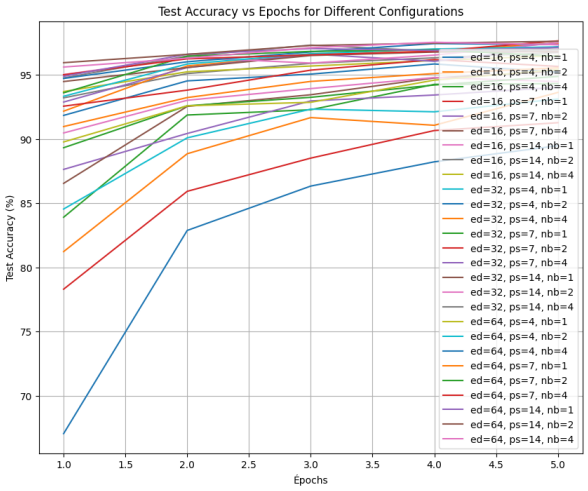


Figure 3.1: Test accuracy for each configurations

- **Increasing Model Capacity**: Experimenting with a larger embedding dimension, more Transformer blocks, or additional attention heads could enhance the model's ability to capture complex patterns, though this comes with increased computational cost and risk of overfitting.

– **Regularization Techniques**: Incorporating dropout layers or weight decay can prevent overfitting and improve generalization.

Balancing these strategies would have allowed us for sure to attain better performances - A well trained CNN can easily achieve a 0.99 accuracy score on MNIST. That fact highlights that there was room for improvement in our model.

3.5.6 Complexity of the Transformer and Improvements

What is the complexity of the transformer in terms of number of tokens? How you can improve it? The computational complexity of the Transformer model is primarily driven by the self-attention mechanism, which has a complexity of $\mathcal{O}(N^2 \cdot d)$, where N is the number of tokens (patches in our case) and d is the embedding dimension. This quadratic dependency arises because the attention mechanism computes pairwise interactions between all tokens in the sequence.

To improve the efficiency and reduce the computational complexity, especially for longer sequences, the following approaches can be considered:

– **Reducing Sequence Length**:
  * **Increasing Patch Size**: Larger patch sizes reduce the number of tokens N by grouping more pixels into each token. In our previous experiments, we saw how great of an influence patch sizes have on the time complexity of our model.
  * **Pooling Mechanisms**: Introducing pooling layers or downsampling techniques to decrease the sequence length during processing.
– **Efficient Attention Mechanisms**:
  * **Sparse Attention**: Limiting attention computations to a subset of tokens, thereby reducing complexity to $\mathcal{O}(N \cdot \log N)$ or $\mathcal{O}(N)$.
  * **Low-Rank Approximations**: Utilizing methods to approximate the attention matrix efficiently.

Implementing these strategies can be a way to significantly reduce the computational burden of the Transformer model.

3.6 Larger transformers

3.6.1 Questions

What is the problem and why we have it? Explain if we have also such problem with CNNs. The model `vit_base_patch16_224` is trained on 224×224 images from the ImageNet dataset. It's crucial to maintain the same image size as the one the model was designed for : if we give the model images of different size (as 28×28 images from MNIST in our case), the number of patches will change. But the ViT is designed in a way that the number of patches (N) is a fixed parameter:

```
class ViT(nn.Module):
  def __init__(self, embed_dim, nb_blocks, patch_size, nb_classes=10):
    super().__init__()

    num_patches = (28 // patch_size) ** 2 # number of patches N, fixed before getting any data

    self.pos_embed = get_positional_embeddings(num_patches+1, embed_dim)
```

Therefore, if we give the model our MNIST images, a mismatch will occur when we will try to add the positional embedding.

A solution would be resizing the images to 224×224 RGB pixels. However, this may not be ideal as resizing could distort the images and potentially affect the model's performance.

As for the CNN, as discussed in question 4 of chapter 2, a CNN would not have such a problem until the fully-connected layer.

3.6.2 Testing Imported ViT Model

In addition to building a custom Transformer model, we experimented with an imported Vision Transformer model to evaluate its performance on the MNIST dataset. We are first using a not pretrained model. We observed that the model was less efficient than the custom model we developed. Specifically, the training process of the ViT model was slower, and its convergence on MNIST was more gradual. Despite training for the same number of epochs (five epochs) to match previous experiments, the ViT model achieved lower accuracy compared to our custom Transformer. This lower performance is due to the ViT model's design, which is optimized for larger and more complex datasets like ImageNet. Its complexity does not translate well to the simpler MNIST dataset, leading to increased training time without significant benefits in accuracy.

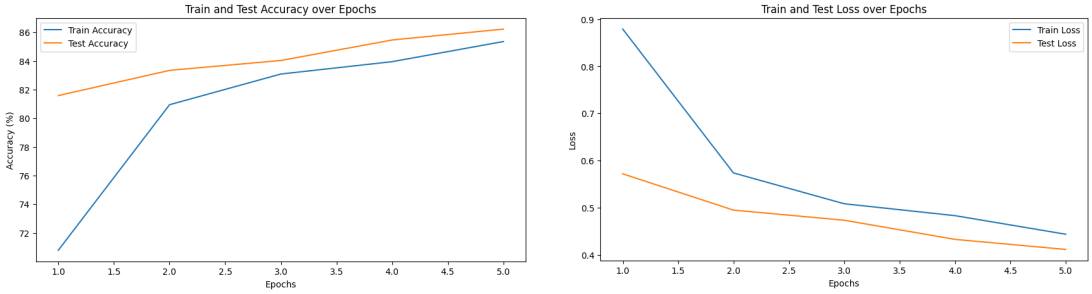


Figure 3.2: Accuracy and losses in train and test when training a pretrained Large ViT

3.6.3 Pre-trained weights

Results improved significantly when using the same model pretrained on the ImageNet dataset, as depicted seen below. However, despite the improvement, the performances are still too low compared to what we achieved with our own ViT.

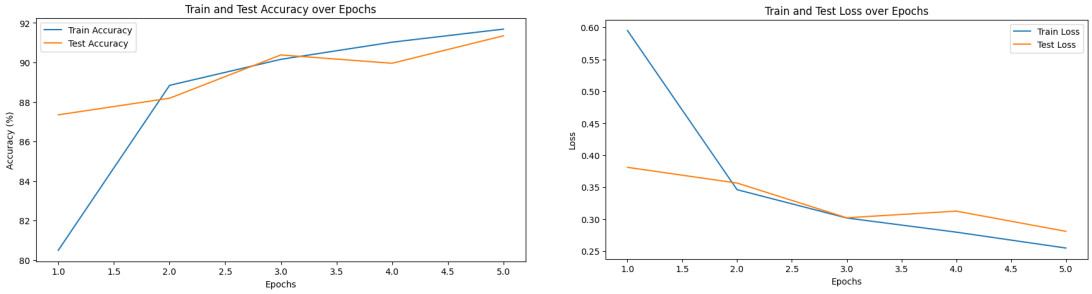


Figure 3.3: Accuracy and losses in train and test when training a pretrained Large ViT

Provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work. We found a lot of "classical" solution when searching how to make thing work on small dataset. Stuff like transfer learning, data augmentation, regularization to avoid overfiting... We'll instead focus on a paper we came across, titled "ConViT: Improving Vision Transformers with Soft Convolutional Inductive Biases" by d'Ascoli et

al., which offers some interesting ideas to address the issue Transformers have while working on small datasets.

Integrating Convolutional Biases The main idea from the paper is to combine Transformers with some properties of Convolutional Neural Networks. CNNs are really good at capturing local patterns in images because they use convolutional layers that focus on nearby pixels. Transformers, on the other hand, are excellent at modeling global relationships but might miss out on these local details, especially when trained on smaller datasets.

The authors introduce a model called **ConViT**, which stands for Convolutional Vision Transformer. It includes a mechanism called **Gated Positional Self-Attention (GPSA)**. This GPSA allows the Transformer to have a kind of "soft" convolutional bias, meaning it can focus more on nearby tokens (like pixels) in a way that's similar to CNNs.

They use a gating parameter that controls how much the model relies on this convolutional bias versus the standard self-attention mechanism. This flexibility helps the model to better capture local features when the dataset isn't very big.

Using a Hierarchical Structure Another point from the paper is using a hierarchical architecture. This is similar to how CNNs reduce the resolution of feature maps as data moves through the layers. By doing this, the model can learn both detailed local features and broader global features effectively.

Benefits for Small Datasets By combining these ideas, the ConViT model performs better on small datasets like CIFAR-10 and CIFAR-100 compared to standard Vision Transformers. The convolutional biases help the model to focus on important local patterns without needing a huge amount of data to learn these relationships.

3.7 Conclusion:

Through this project, we have gained valuable experience in developing and assessing Transformer architectures. Our findings suggest that for tasks involving simple or small datasets, custom models (such as the one we developed) tailored to the specific characteristics of the data can outperform more complex, general-purpose models (such as `vit_base_patch16_224`). Future work could explore techniques to adapt large Transformers to small datasets, such as incorporating convolutional inductive biases or employing parameter-efficient transfer learning methods, to bridge the gap between model capacity and data availability.