Faculty of Science and Engineering - Sorbonne Université

Master's in Computer Science - DAC / IMA

RDFIA - Reconnaissance des formes pour l'analyse et l'interprétation d'images

Project Report

Basics on Deep Learning for Vision

Authored by:
Faten Racha SAID - M2 DAC
Mounir MAOUCHE - M2 IMA
Thiziri OUMAZIZ - M2 IMA

Supervised by:
Clement RAMBOUR
Matthieu CORD

October 2025

Contents

List of Figures

Chapter 1

Introduction to Neural Networks

The objective of this practical session is to construct a simple neural network, enhancing our understanding of these models and their training process through gradient backpropagation.

We will begin by examining the theoretical foundations of the learning process for a single-layer perceptron with a hidden layer. Following this, we will implement the network using the PyTorch library, initially applying it to a basic dataset to validate functionality before testing it on the more complex MNIST dataset.

1.1 Theoretical foundation

To effectively apply a neural network to a supervised learning problem, four key elements are essential: a suitable supervised dataset, an appropriate network architecture, a well-defined loss function for optimization, and an optimization algorithm to minimize the loss function. As we delve into the learning process of the perceptron, our focus will center on these fundamental components, which form the backbone of any neural network application in machine learning.

1.1.1 Supervised dataset

In this context, we are tackling a supervised classification task, which involves working with a labeled dataset comprising N pairs of features and targets $(x^{(i)}, y^{(i)})$, where $x^{(i)} \in \mathbb{R}^{n_x}$ represents a feature vector, and $y^{(i)} \in \{0, 1\}^{n_y}$ is the one-hot encoded target, indicating the class to which each sample belongs. This dataset will be divided into three subsets: a training set, a test set, and, if feasible, a validation set.

Q1. What are the train, val and test sets used for ?

- The training set is used to adjust the model by finding the optimal parameters (weights and biases). In other words, the model learns from this data.
- The validation set is used to fine-tune the model's hyperparameters. It is distinct from the test set and allows for evaluating the model on data it has not seen during training. Additionally, the validation set helps to detect overfitting, ensuring that the model does not become too specific to the training data.
- The test set is used to assess the final performance of the model on completely new data, measuring its ability to generalize.

It is important to note that the training, validation, and test sets must remain strictly separate to avoid any data leakage, ensuring a fairer evaluation of the model's performance. Furthermore, each set should generally reflect a similar data distribution to obtain reliable results during evaluation.

Q2. What is the infuence of the number of examples N ?

The number of examples N in a supervised classification task has a direct impact on the model's performance and behavior. Generally, a larger amount of data promotes better generalization, reduces the risk of overfitting, and allows for the use of more complex models while providing a more reliable evaluation of performance. Conversely, a small number of examples may hinder the model's ability to learn the underlying relationships in the data, as it lacks sufficient information. This can lead to an overly simplistic model that fails to capture the complexity of the data.

It is worth noting that after a certain amount of data, performance gains may become negligible, so it is important to find a balance between significant computation/storage time and performance improvements. Additionally, the **quality of the data** is just as important as its quantity. A large number of noisy or erroneous examples can distort the model.

1.1.2 Network architcture (forward)

A neural network f consists of a series of mathematical transformations that map input features to predictions, expressed as $f : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$.

The process of calculating the output y from the input x is referred to as the forward pass of the network. A basic neural network may involve a simple linear transformation, represented by $y = f(x) = Wx$, where W is a matrix of size $n_y \times n_x$. Typically, this transformation is enhanced with an affine transformation $f(x) = Wx + b$, where b is a bias vector. Each linear transformation is followed by a non-linear activation function, which serves two purposes: it introduces non-linearity into the model and it allows to choose an output interval different from $\mathbb{R}^{n_y}$, such as using the tanh function to confine outputs to the interval $[-1, 1]^{n_y}$.

Q3. Why is it important to add activation functions between linear transformations ?

Adding activation functions between linear transformations is essential for introducing non-linearity into the model. A neural network that relies only on linear transformations is confined to representing linear relationships, which hinders its capacity to learn and identify more complex patterns in the data. This lack of non-linearity significantly limits the model's effectiveness in capturing the intricacies present in real-world scenarios.

Q4. What are the sizes n_x, n_h, n_y in the figure given, In practice, how are these sizes chosen?

- $n_x = 2$ corresponds to the input size. This represents the number of input variables in the dataset. **In practice**, n_x can either include all available features or be refined through feature selection. For instance, Lasso regularization can encourage the neural network to focus on the most relevant variables by assigning negligible weights to less important ones.

- $n_h = 4$ designates the size of the hidden layer. **In practice**, the choice of the number of neurons in this layer depends on the complexity of the problem. If the layer contains too many neurons, it may lead to overfitting, while too few can result in underfitting. Thus, experimentation and adjustments are necessary to find the right balance. Moreover, adding neurons in the same layer increases the complexity of the model **exponentially**, as each additional neuron adds weights to all connections. In contrast, adding new successive layers increases complexity more **linearly**, as each layer acts as an additional transformation applied to the data.

- $n_y = 2$ represents the output size, **In practice**, it's determined by the number of classes in the problem to be solved.

Q5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?

- $\hat{y}$ corresponds to the model's predictions, meaning what the neural network estimates: the probabilities of belonging to classes, the predicted labels, or continuous values.

- y represents the actual data, or the target values, which we seek to predict.

The difference between $\hat{y}$ and y is known as the residual or prediction error, which is measured by the loss function. This function varies according to the type of problem: for instance, cross-entropy is often used for classification, while mean squared error is common for regression problems. The objective is to bring $\hat{y}$ as close as possible to y by adjusting the model parameters to minimize the loss function.

Q6. Why use a SoftMax function as the output activation function?

The SoftMax activation function converts the outputs of the network into a probability vector, allowing for a distribution over the various possible classes. Each output becomes a probability associated with a class, with the sum of the probabilities equaling 1, making the predictions easier to interpret.

Unlike other activation functions like Sigmoid, which mainly applies to binary problems, SoftMax is specifically suited for multiple classes problems.

Q7. Write the mathematical equations allowing to perform the forward pass of the neural network, i.e., allowing to successively produce $h, \hat{y}, y$ starting at x.

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

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

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

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

1.1.3 Loss function

To measure the difference between the predicted output $\hat{y}^{(i)}$ and the target $y^{(i)}$, a suitable loss function is chosen according to the type of problem. Typically, the global loss function $L(X, Y)$ is the average of a unit loss function $\ell(y^{(i)}, \hat{y}^{(i)})$ between predictions and targets:

$$L(X, Y) = \frac{1}{N} \sum_{i=1}^{N} \ell(y^{(i)}, \hat{y}^{(i)})$$

The two most common loss functions are:

- **Cross-entropy**, suitable for classification tasks:

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

- **Mean Squared Error (MSE)**, suitable for regression tasks:

$$\ell(y, \hat{y}) = \frac{1}{2} \sum_i (y_i - \hat{y}_i)^2$$

Q8. 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?

The primary objective during training is to minimize the loss function, which evaluates the difference between the predicted outputs $\hat{y}$ and the true values y for each data point i.

For cross-entropy, to reduce the global loss function $\ell(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$, $\hat{y}_i$ should approximate y_i as closely as possible. The partial derivative of the loss with respect to $\hat{y}_i$ is:

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

This indicates that $\hat{y}_i$ should approach 1 for the correct class predictions and closer to 0 for others.

For the squared error $\ell(y, \hat{y}) = \sum_i (y_i - \hat{y}_i)^2$, minimizing the loss also requires $\hat{y}_i$ to be as close as possible to y_i. The partial derivative in this case is:

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

This implies that $\hat{y}_i$ should reduce the difference with y_i to decrease the Mean Squared Error (MSE).

Q9. How are these functions better suited to classification or regression tasks?

Why is Cross-entropy well-suited to classification tasks ?

Cross-entropy is a loss function well-suited for classification problems, as it quantifies the difference between two probability distributions. It also encourages the model to be more confident in its predictions by heavily penalizing errors, particularly when the model assigns a high probability to an incorrect class. This helps to reduce the gap between the probabilities predicted by the model and the actual classes during backpropagation, making it a relevant choice for multi-class classification.

Why is Cross-entropy *not* suited to regression tasks ?

Cross-entropy cannot be used for regression tasks because it measures the degree of disorder between two probability distributions, making it ineffective for comparing two continuous values.

Why is MSE well-suited to regression tasks ?

Mean Squared Error is more appropriate for regression tasks, where the goal is to predict continuous values. It calculates the average of the squared differences between the actual and predicted values. This function emphasizes minimizing large errors by amplifying the most significant discrepancies.

Why is MSE *not* suited to classification tasks ?

Using MSE for classification tasks has several limitations that make it less suitable than cross-entropy.

While applying a sigmoid function before MSE can stabilize training by converting outputs to probabilities, this approach does not provide gradients as informative as those generated by cross-entropy. This is because the MSE penalizes proportionally to the squared distance between the output and target, which can slow down learning for predictions already close to 0 or 1. In contrast, cross-entropy produces higher gradients for incorrect predictions, accelerating convergence and improving class probability adjustments.

Additionally, for multi-class classification, MSE with sigmoid does not enforce class exclusivity, unlike softmax with cross-entropy, which can lead to ambiguities in predictions.

1.1.4 Optimization algorithm

The loss function quantifies the error in a neural network, guiding the adjustment of its parameters (weights and biases) to reduce this error across training examples. To optimize these

parameters, we employ gradient descent, an iterative algorithm that computes how the loss changes with respect to the parameters. By systematically updating the parameters in the direction that decreases the loss, we aim to improve the model's performance on the training data. This process can be executed using different variations: classic gradient descent on the full training set, stochastic gradient descent on random mini-batches, or online SGD using individual examples.

Backpropagation is employed to compute gradients efficiently by applying the chain rule, allowing us to propagate the gradients backward through the network without redundant calculations.

Q10. 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 ones seem the most reasonable to use in general?

The objective is to adjust the parameters of a model to minimize the loss function, using the gradient descent algorithm:

- **Classic gradient descent**: This version considers the entire dataset at each step. It is stable and offers deterministic convergence since the gradients are accurately calculated over all data. It is particularly suitable for convex or relatively smooth loss functions. However, it is computationally expensive for large datasets and can get stuck in local minima on non-convex loss functions.
- **Mini-batch Stochastic Gradient Descent** : Here, the model updates using small subsets of data (mini-batches). This provides a good compromise between efficiency and stability, especially for large datasets. This method is faster than classic gradient descent and well-suited for non-convex loss functions, as the noise introduced by mini-batches can sometimes help avoid local minima. However, this noise can also slow convergence and introduce oscillations, requiring careful tuning of hyperparameters like the learning rate.
- **Online Stochastic Gradient Descent**: Here, updates occur after each individual example. This allows for very rapid adjustments and a low memory requirement, ideal for online learning scenarios or evolving data. However, this method is more unstable and less deterministic, and the high noise in updates can make convergence harder to control, especially with a poorly adjusted learning rate.

In general, mini-batch SGD is often a good compromise between speed, efficiency, and stability, making it suitable for many contexts, especially with large datasets.

Q11. What is the influence of the learning rate on learning?

The learning rate determines how quickly the model adjusts its weights based on the gradient of the loss function. A low learning rate allows the algorithm to slowly descend the slope of the loss function by making small adjustments at each step, which may require more training epochs. This reduces the risk of missing local minima, but convergence can be very slow, potentially getting stuck on a plateau (where the gradient is very small or close to zero, causing the updates to the weights to become minimal because the loss function isn't changing significantly in that area).

Conversely, a learning rate that is too high enables the model to learn quickly by making large adjustments at each iteration, reducing the number of epochs needed. However, this can lead to rapid convergence to a suboptimal solution if the updates are too aggressive.

In practice, techniques like learning rate decay or using adaptive optimizers such as Adam or RMSprop help better adjust the learning rate over time, optimizing convergence. Additionally, strategies like cyclical learning rates offer an alternative by periodically alternating between high and low values, allowing for effective exploration of the loss function space.

Q12. 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 backpropagation algorithm.

The complexity of calculating gradients in a neural network depends on the number of layers and can vary significantly depending on the approach used.

With the naive approach, which calculates gradients directly for each parameter, complexity increases exponentially with the number of layers, making it impractical for deep networks.

In contrast, the backpropagation algorithm effectively reduces this complexity by reusing intermediate results between layers, maintaining linear complexity with respect to the number of layers. This optimization allows for much faster parameter updates and is essential for training deep networks on large datasets.

Q13. What criteria must the network architecture meet to allow such an optimization procedure?

For a network to be optimized using backpropagation, it must meet several criteria. First, the activation functions and loss function used in the architecture must be **differentiable** with respect to their parameters. This is an essential condition for gradients to be calculated during training.

Next, the network must have a **feedforward** structure (without cycles) to allow efficient forward and backward passes during gradient calculations. In recurrent or more complex architectures, variants like backpropagation through time (BPTT) may be necessary.

Q14. The SoftMax function and cross-entropy loss are often used together, and their gradient is very simple. Show that the loss can be simplified by: $l = -\sum_i y_i \tilde{y}_i + \log\left(\sum_i e^{\tilde{y}_i}\right)$

We know that:

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

And according to the equation from Q7 (answered in 1.1.2)

$$\hat{y} = \text{SoftMax}(\tilde{y}) = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}$$

we have then:

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

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

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

Q15. Write the gradient of the loss (cross-entropy) relative to the intermediate output $\tilde{y}$

Calculation of the gradient of the cross-entropy loss with respect to the intermediate output $\tilde{y}$:

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

Finally, we have

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

Q16. 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 $\nabla_{b_y} l$.

We begin by calculating the gradient of the loss with respect to the weights of the output layer, denoted as $\nabla_{W_y} l$. The expression is given by:

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

Where:

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

Now, we can compute the partial derivative of $\tilde{y}_k$ with respect to $W_{y,kj}$:

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

Thus, we obtain:

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

Hence, we have:

$$\nabla_{W_y} l = \begin{pmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_n - y_n \end{pmatrix} \begin{pmatrix} h_1 h_2 \dots h_n \end{pmatrix}$$

$$\nabla_{W_y} l = \nabla_{\tilde{y}} l \cdot h^T$$

Next, we focus on the gradient with respect to the bias term, $\nabla_{b_y} l$:

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

Thus, we conclude:

$$\nabla_{b_y} l = \nabla_{\tilde{y}} l$$

Q17. Compute the other gradients $\nabla_{\tilde{h}} l$, $\nabla_{W_y} l$, $\nabla_{b_y} l$.

1. $\nabla_{\tilde{h}} l$:

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

We have :

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

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

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

Thus,

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

Therefore, we obtain:

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

2. $\nabla_{W_h} l$

We know that,

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

Based on the previous questions, we already have $\frac{\partial l}{\partial \tilde{h}_k}$ and we know that:

$$\tilde{h}_k = \sum_{j=1}^{n} W_{h_{k,j}} x_j + b^h_k$$

Now, we calculate the derivatives :

$$\begin{aligned} \frac{\partial l}{\partial W_{h_{i,j}}} &= \sum_k \frac{\partial l}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W^h_{i,j}} \\ &= \delta^i_h x_j \\ \nabla_{W_h} &= \nabla_{\tilde{h}} l \cdot x^T \end{aligned}$$

3. $\nabla_{b_h} l$ Similarly :

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

We have, then:

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

1.2 Implementation

Now that we have set up the theoretical foundations for making predictions, evaluating loss, and implementing gradient descent for learning, we'll move on in this section to the practical implementation of these concepts using PyTorch. We start with the "Circle" dataset, which contains data points in two separate circles, a common setup for non-linear classification tasks. We primarily experiment with different possible configurations of this library and vary various hyperparameters to observe their impact on model performance.

1.2.1 Forward and backward manuals

For this first experiment, we built a neural network almost "by hand," which allowed us to manage the fundamental operations directly, such as initializing parameters, defining the forward pass, and manually computing gradients.

We opted for a hidden layer of 10 neurons, with the Stochastic Gradient Descent (SGD) optimizer, a learning rate of 0.03, and a batch size of 10. Training spanned 150 epochs, with shuffled batches each round for better model generalization.

The results were promising, as we achieved a training accuracy of 95.5% and a testing accuracy of 92%, as shown in the figure below.

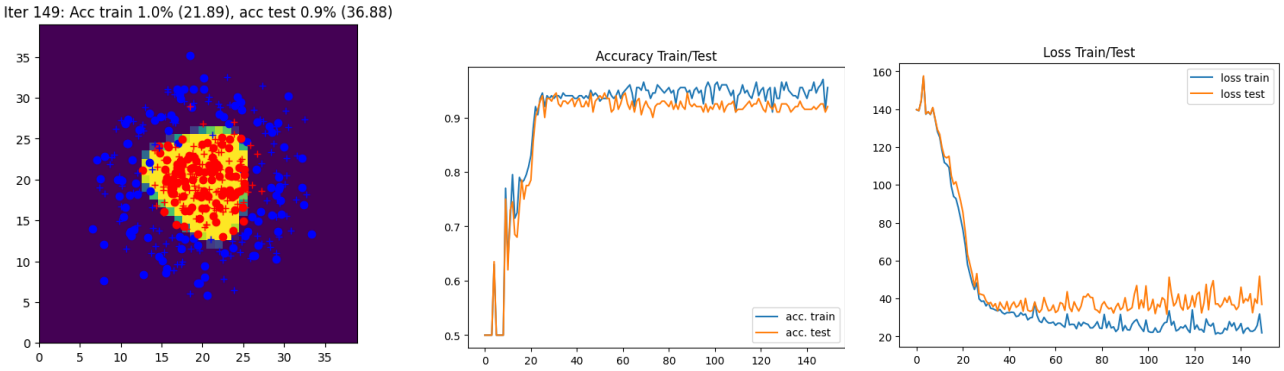

Figure 1.1: Accuracy and Loss Plots from Manual Implementation of Forward and Backward Functions on Circle Dataset.

From the figure on the left, we can see that by the end of training, the model forms circular decision boundaries that align closely with the data, demonstrating effective classification.

From the two other figures, we observe that the model converges around 30 epochs for both the loss and accuracy plots. However, beyond 120 epochs, the training and testing curves begin to diverge, suggesting potential overfitting. This implies that the model's generalization ability decreases past this point, making it preferable to stop training around this limit to achieve better performance.

Experiments on the learning rate and the batch size

In this part of the project, we tested the influence of different learning rates and batch sizes on model performance to better understand their effects on both training stability and generalization. The values used are outlined in the following table:

Learning Rates	Batch Sizes
0.01, 0.1	20, 10, 5

Table 1.1: Learning Rates and Batch Sizes Tested

We iterated over all combinations of these parameters to assess their influence on model accuracy and loss dynamics. We obtained the graphs shown in the figures below 1.2, 1.3 and 1.4.

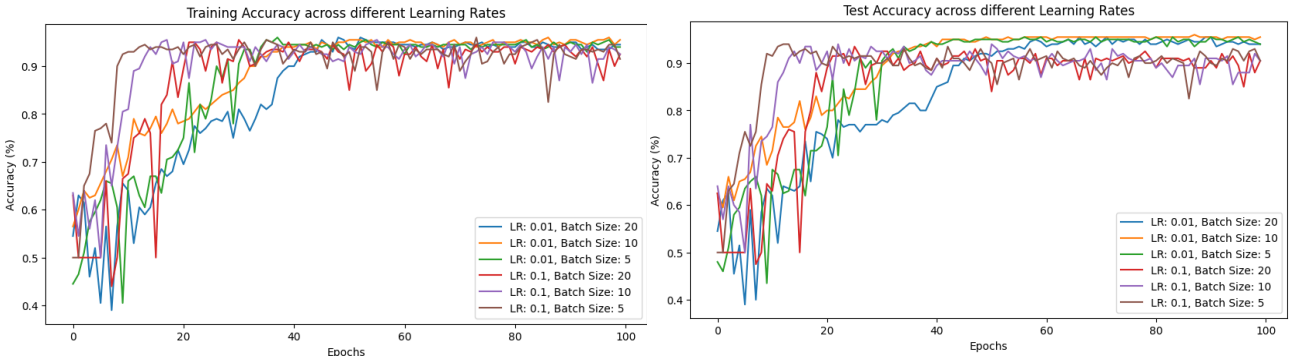

Figure 1.2: Train and Test Accuracy across different Learning Rates and batch sizes on Circle Dataset.

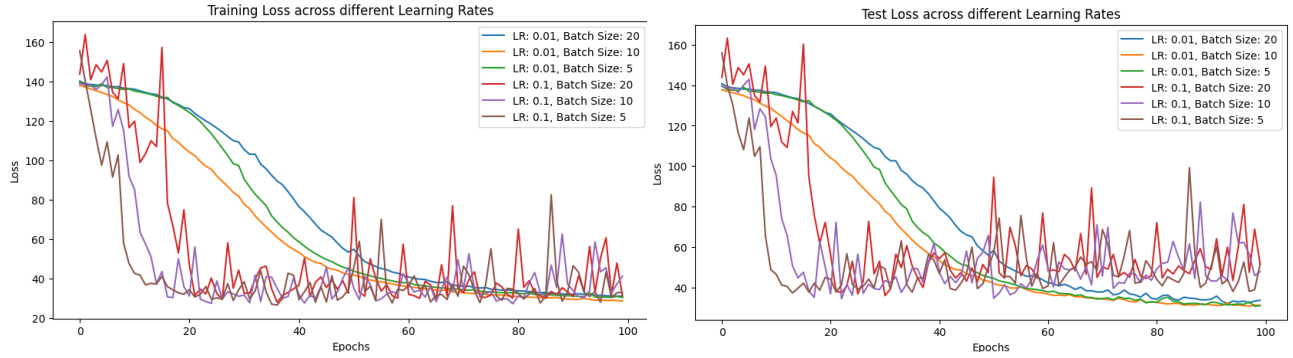

Figure 1.3: Train and Test Loss across different Learning Rates and batch sizes on Circle Dataset.

The curves for a learning rate of 0.01 show a slower convergence, compared to the others which exhibit more noise. This is because weight updates are more gradual when the learning rate is small. Convergence with this rate (0.01) begins around 50 epochs, whereas for the other curves, it occurs within around 20 epochs. However, in the long run, the lower learning rate yields the best performance, with a slight advantage for smaller batch sizes.

From this, we can infer that it is generally better to work with a relatively small learning rate, even if it means slower progress initially, as this stabilizes the model. Additionally, reducing the batch size allows for more frequent weight updates, which helps to explore the loss function's landscape more thoroughly and avoid getting stuck in local minima.

In the figures below 1.4, 1.5 , we display the same curves but with training and testing results overlaid. These figures further confirm our observations, demonstrating the model's stability with a smaller learning rate and batch size, while the other configurations exhibit noisier curves.

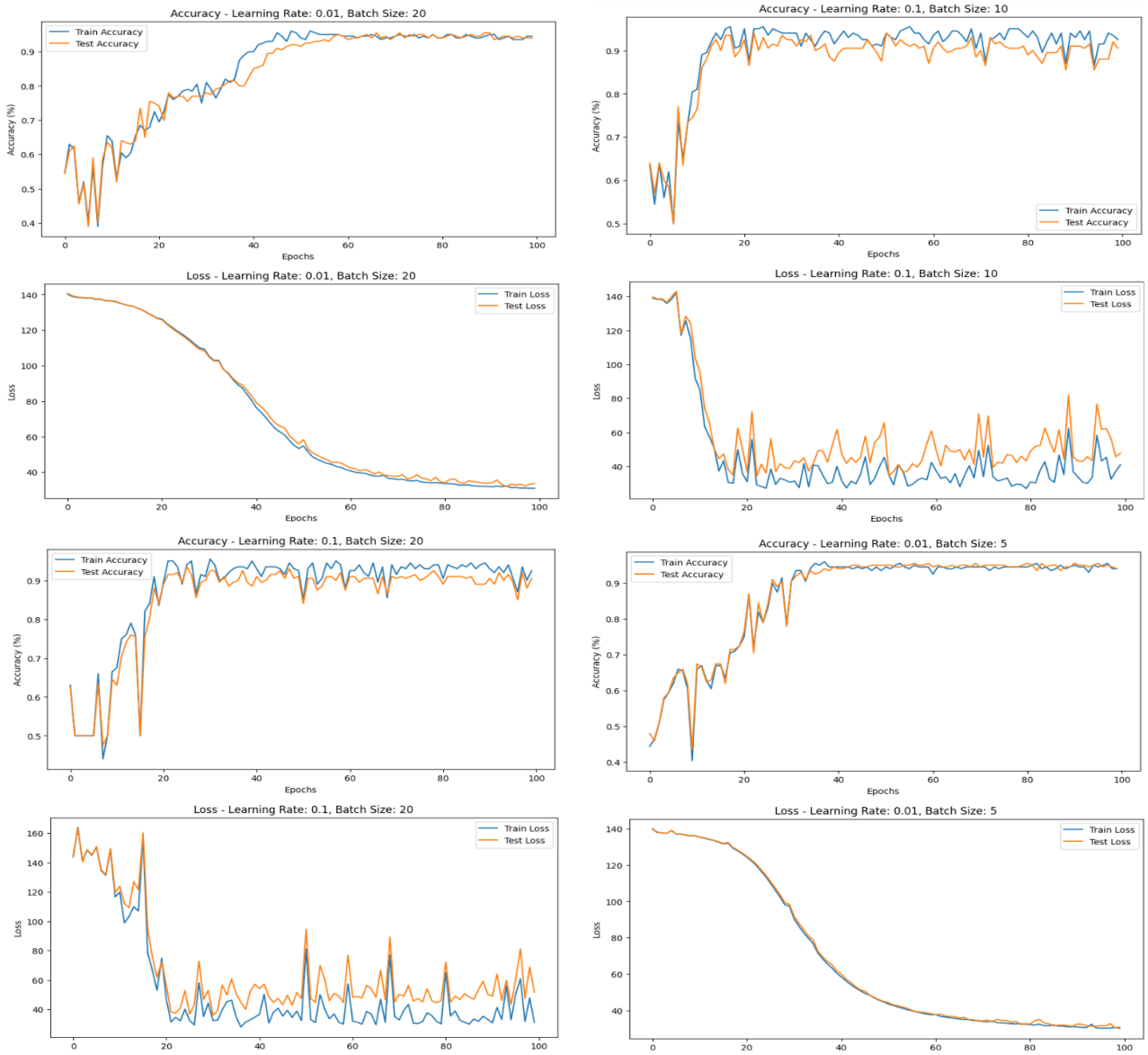

Figure 1.4: Train/Test Accuracy and Loss across Different Learning Rates and Batch Sizes on Circle Dataset

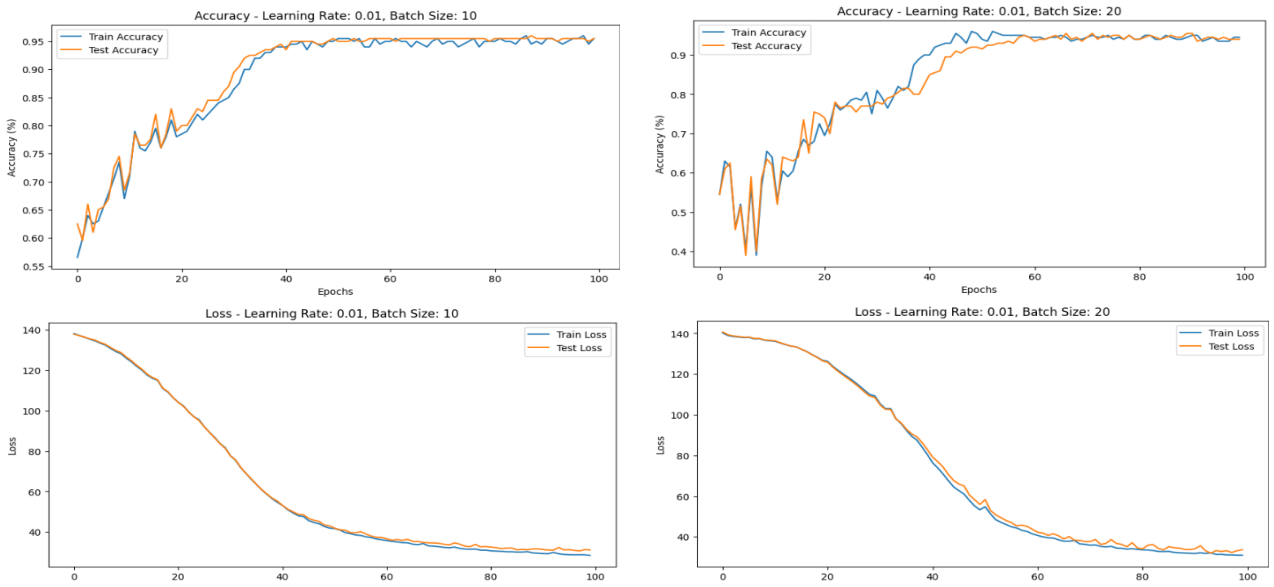

Figure 1.5: Train/Test Accuracy and Loss across Different Learning Rates and Batch Sizes on Circle Dataset (2)

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

In this section, we utilize PyTorch's automatic differentiation feature, Autograd. By setting `requires_grad` to `True` for tensors, Autograd computes gradients automatically.

With `loss.backward()`, PyTorch calculates the derivatives of the loss with respect to contributing tensors, storing them in each tensor's `.grad` attribute, like `W.grad` for tensor `W`. Note that calling `.grad.zero_()` after each update resets the accumulated gradients, ensuring that previous steps don't interfere with subsequent updates. We explore how this 'torch.autograd' simplifies gradient computation in PyTorch.

We reran the learning algorithm using the same hyperparameters as in our previous run, yielding similar results to those obtained with the manual forward and backward implementation. Specifically, the model achieved 97% accuracy on the training set and 93.5% on the test set. However, we observed that training stability was slightly lower, with the model exhibiting signs of divergence (overfitting) after about 60 epochs as shown by the loss curve.

Figure 1.6: Accuracy and Loss Plots on Circle Dataset with 'torch.autograd'.

1.2.3 Simplification of the forward pass with 'torch.nn'

In this section, we simplify the network architecture using PyTorch's "nn" package, which provides modules for building neural networks. With 'torch.nn', we can execute forward passes by directly calling the model, avoiding the need for manual implementation of the forward method.

We reran the learning algorithm for 250 epochs using the same hyperparameters as before, resulting in improved performance over previous implementations. The model achieved 96% accuracy on the training set and 95% on the test set, with a notably more stable training process and significantly better generalization, as illustrated by the loss curve

Figure 1.7: Accuracy and Loss Plots on Circle Dataset with 'torch.autograd' and 'torch.nn' .

The convergence is much slower using 'torch.nn', we can assume that The forward pass with 'torch.nn' likely employed a more cautious initialization of the weights, allowing the model to adapt gradually to the training data. In contrast, the manual approach may lead to quicker and less controlled weight updates, resulting in the model specializing too rapidly on the training data from the early epochs. This divergence in behavior highlights how initialization and update methods can significantly affect model stability and generalization.

1.2.4 Simplification of the SGD with 'torch.optim'

In this section, we transition from manually updating parameters to using PyTorch's `torch.optim` module, which provides optimizers like SGD to handle updates via `optim.step()`. This approach streamlines parameter updates and offers a range of optimization algorithms for more efficient training.

The results shown in the figure below, indicate a notably stable training process, achieving 96% accuracy on the training set and 95% on the test set, without increase in convergence time compared to the previous execution.

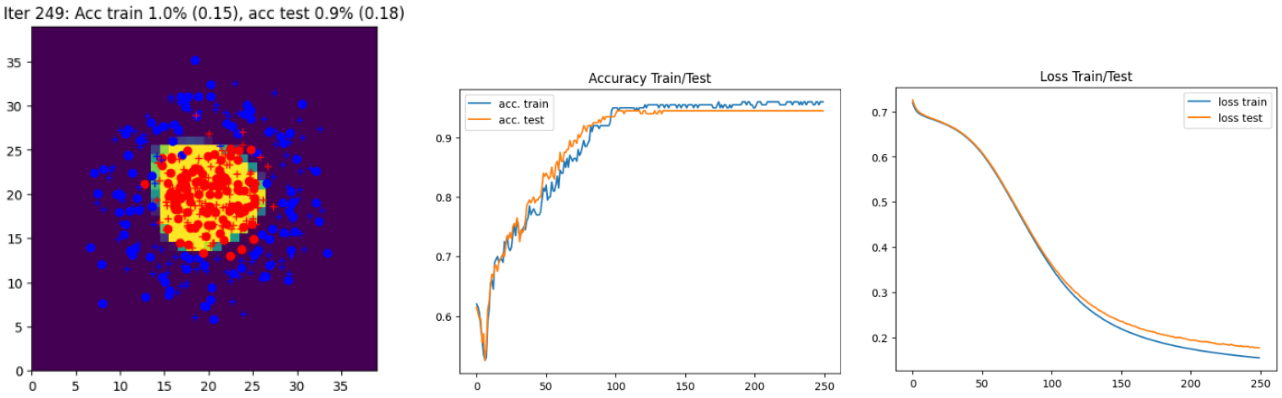

Figure 1.8: Accuracy and Loss Plots on Circle Dataset with 'torch.autograd', 'torch.nn' and 'torch.optim'.

1.2.5 MNIST application

In this section, we will test our code on the MNIST dataset, which comprises images of handwritten digits representing 10 classes. The goal is to predict the class (a digit) from an image of 28×28 pixels, which we will convert into vectors of 784 values.

When running the neural network with a batch size of 100 and a learning rate of 0.03 over 200 epochs, the model achieves approximately 94% accuracy on both the training and testing datasets, indicating strong generalization.

A slight instability can be observed, likely due to the chosen hyperparameters.

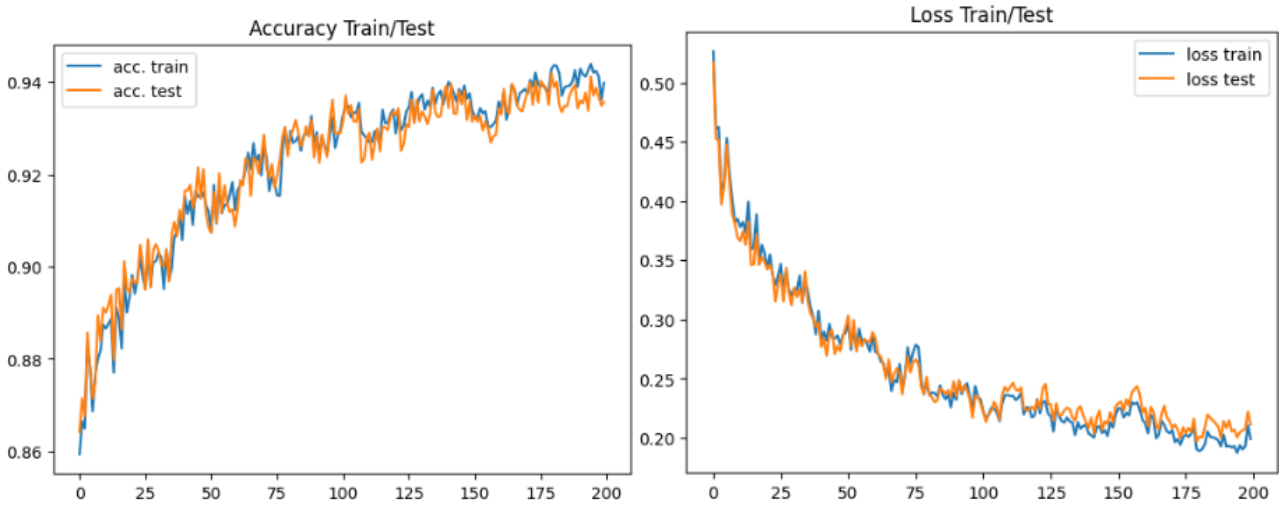

Figure 1.9: Accuracy and Loss Plots on MNIST Dataset with 'torch.autograd', 'torch.nn' and 'torch.optim'.

1.2.6 Bonus: SVM

In this section, we applied Support Vector Machine (SVM) models with different kernels to the Circles dataset, which is circular and not linearly separable.

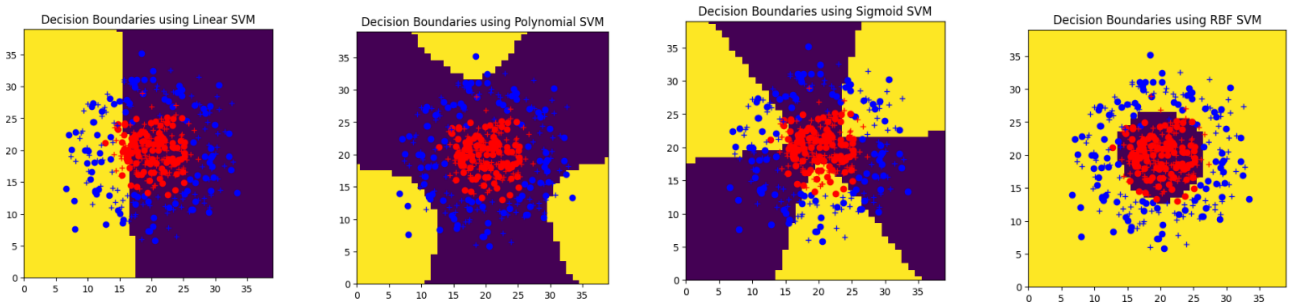

Figure 1.10: Decision Boundaries using different SVM kernels.

A linear SVM yielded poor performance, achieving only 62% accuracy, as it relies on linear decision boundaries and is therefore unsuited for capturing non-linear patterns in circular data.

In contrast, SVM models with Radial Basis Function (RBF), sigmoid, and polynomial (deg=3) kernels demonstrated progressively better performance, with the RBF kernel achieving 94% accuracy. The RBF kernel is particularly effective for circular data as it adapts flexibly to complex, non-linear relationships and can accurately model circular decision boundaries. This adaptability makes it a suitable choice for datasets with intricate, non-linear separations.

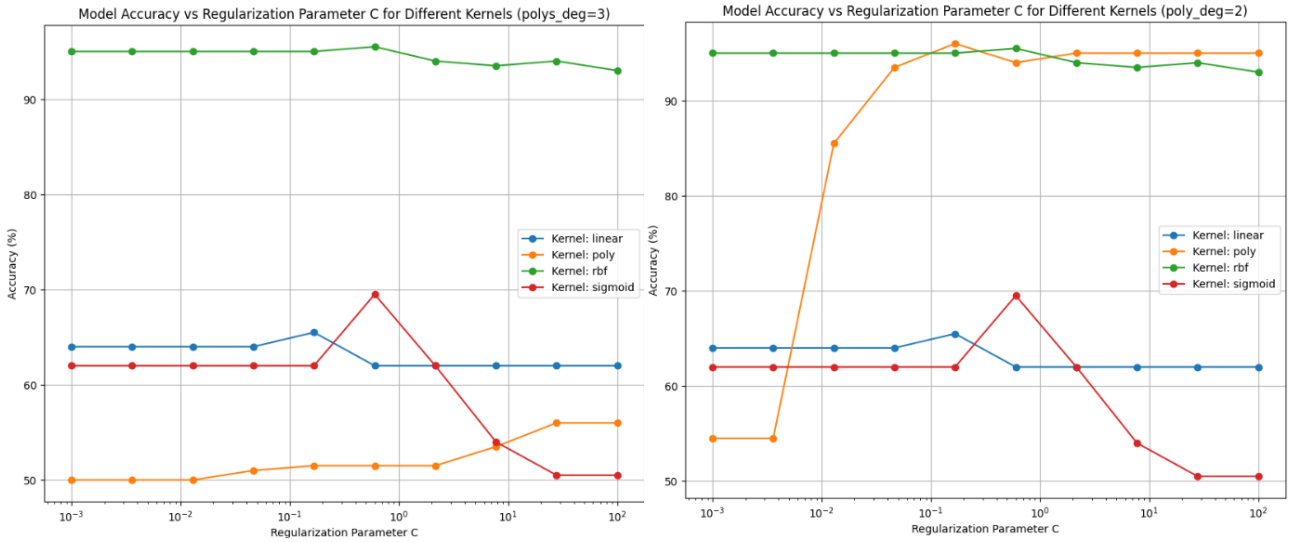

Figure 1.11: Model Accuracy vs Regularization Parameter C for Different Kernels.

Our experiments examined the impact of the regularization parameter C on Support Vector Machines (SVM), visualized in Figure 1.11. This parameter determines the trade-off between fitting the training data and generalizing to new data. Specifically, lower C values prioritize margin maximization, resulting in a simpler, more robust decision boundary that may allow some training points to be misclassified. Conversely, higher C values reduce misclassification on the training set, producing a tighter margin but increasing the risk of overfitting, especially in noisy datasets. Our results, using RBF, polynomial, and sigmoid kernels, revealed that each model required specific C values for optimal performance, highlighting how important proper tuning is based on kernel type and data characteristics.

In particular, we can observe that the polynomial kernel's effectiveness depended signifi-

cantly on the polynomial degree; results showed that a degree-2 polynomial matched the performance of RBF for $C > 0.1$, demonstrating that appropriate kernel selection and feature engineering are essential for polynomial SVMs. Linear and sigmoid kernels, however, showed lower performance overall.

These findings underscore that tuning C and carefully choosing the kernel can enhance SVM generalization and classification accuracy, making kernel selection and feature engineering important considerations for effective SVM modeling.

1.3 Conclusion

This chapter covered the fundamentals of neural networks, focusing on the theoretical framework of supervised learning, including key elements like dataset splits, loss functions, and optimization methods. We explored the construction of a single-layer perceptron, detailing the forward and backward passes and discussing common loss functions and optimization algorithms, such as mini-batch SGD. Using PyTorch, we implemented these concepts, progressing from manual calculations to automated differentiation with PyTorch's autograd and nn modules. This chapter established a solid foundation in neural network basics, preparing for more advanced models in later sections.

Chapter 2

Convolutional Neural Networks

2.1 Introduction to Convolutional Networks

Q1. 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?

Since we have only one filter, the output size will be:

$$\text{Output Size} = \text{out}_x \times \text{out}_y \times 1$$

where:

$$\text{out}_x = \frac{x - k + 2p}{s} + 1 \quad \text{et} \quad \text{out}_y = \frac{y - k + 2p}{s} + 1$$

- For a convolutional layer, the number of weights to learn includes the kernel weights plus a bias term:

$$\text{Number of Weights} = k \times k \times z + 1 \quad \text{(bias)}$$

- If a fully connected layer were to produce an output of the same size as the convolutional layer, it would need to connect each element in the input of size $x \times y \times z$ to each element in the output of size $\text{out}_x \times \text{out}_y \times 1$. Therefore, the total weights required would be:

$$\text{Total Weights (FC)} = (x \times y \times z) \times (\text{out}_x \times \text{out}_y \times 1) + (\text{out}_x \times \text{out}_y \times 1) \quad \text{(biases)}$$

- This is significantly larger than the number of weights required for a convolutional layer, which only depends on the kernel size and the input depth rather than the entire input and output size.

Q2. What are the advantages of convolution over fully-connected layers? What is its main limitation?

Advantages of convolution over fully-connected layers:

- Convolutional layers use fewer parameters, reducing computational load and preventing overfitting.
- They detect local patterns (e.g., edges, textures) by focusing on small regions.

- Convolutional layers offer translation invariance, detecting features regardless of their position.
- Early layers detect basic features (e.g., edges), while deeper layers combine these into complex structures.

Main limitation:

- Convolutions focus on local features, making it difficult to capture long-range dependencies or global patterns without additional mechanisms.

Q3. Why do we use spatial pooling?

- It reduces parameters and computation, making the model more efficient.
- Pooling increases robustness to small translations or distortions in the input.
- It highlights important parts of the input, aiding hierarchical feature learning.

Q4. Suppose we compute the output of a convolutional network for a larger input image. Can we use all or part of the layers of the network on this image?

We can use the convolutional and pooling layers on a larger image since they adapt to different input dimensions. However, fully connected layers at the end require a fixed input size; larger images would need adjustment in these layers.

Q5. Show that we can analyze fully-connected layers as particular convolutions.

To analyze fully-connected (FC) layers as particular convolutions, we can consider the following configuration:

- **Kernel Size:** In an FC layer, each neuron is connected to every input feature. We can view this as a convolution with a kernel size equal to the entire input spatial dimensions.
- **Stride and Padding:** Since the filter covers the entire input in a single application and produces only one output, the stride does not matter (it can be set to 1) and no padding is needed.
- **Number of Filters:** Each output neuron in an FC layer can be thought of as applying a unique filter over the entire input, equivalent to a set of convolution filters, where each filter corresponds to a neuron in the FC layer.

Q6. If we replace fully-connected layers with their equivalent convolutions, what is the output shape and interest?

If we replace fully-connected layers with their equivalent convolutions, we can compute the output on a larger image. The output shape will adapt proportionally to the input size. Instead of reducing all information to a single vector (as in traditional fully-connected layers), the convolutional structure preserves a spatial map, where each position in the output corresponds to a specific region of the input image.

Interest: This setup allows the network to produce spatially distributed outputs, retaining information about the location of features within the image. This is particularly useful in applications like object detection or segmentation, where spatial structure is important. Additionally, this approach enables processing inputs of varying sizes while maintaining a consistent flow throughout the network.

Q7. What are the sizes of the receptive fields of the neurons in the first and second convolutional layers?

- **First Layer:** Each neuron's receptive field corresponds to the kernel size (e.g., 3×3 or 5×5).
- **Second Layer:** In the second convolutional layer, the receptive field expands because each neuron in this layer depends on the outputs of a region from the first layer, which themselves are already dependent on local regions of the input. For example, with a 3×3 kernel in both layers and a stride of 1, the receptive field of neurons in the second layer would be 5×5 pixels in the original image. Following this formula :

$$R_L = R_{L-1} + (k_L - 1) \times S_{L-1}$$

  where R_L is the receptive field of layer L, k_L is the kernel size, and S_{L-1} is the stride of the previous layer. In deeper layers, the receptive field continues expanding, capturing larger regions and more abstract features.

- **Deeper Layers:** As we go deeper into the network, the receptive field of each neuron continues to expand, covering progressively larger portions of the input image. In very deep layers, neurons may have receptive fields that cover most or all of the input.

Interpretation:

- **Early Layers:** Capture local details like edges or textures, as their receptive fields are small.

- **Deeper Layers:** Capture more abstract features, as their larger receptive fields allow them to recognize patterns and structures involving larger regions of the input image.

2.2 Training from Scratch of the Model

2.2.1 Network Architecture

Q8. For convolutions, we want the same output spatial dimensions as the input. What padding and stride values are needed?

Given a kernel size of 5×5, to maintain the same spatial dimensions in each convolutional layer, we use:

- **Padding** $p = 2$
- **Stride** $s = 1$

Q9. For max poolings, we want to halve the spatial dimensions. What padding and stride values are needed?

Given a kernel size of 2×2, we use:

- **Stride** $s = 2$
- **Padding** $p = 0$

Q10. For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.

1. **conv1:** 32 convolutions 5×5, followed by ReLU
   Input size: $32 \times 32 \times 3$
   Output size: $32 \times 32 \times 32$
   Weights: Each filter has $5 \times 5 \times 3 = 75$ weights, plus 1 bias per filter.
   Total weights $= 32 \times (5 \times 5 \times 3 + 1) = 2432$

2. **pool1:** Max-pooling 2×2
   Input size: $32 \times 32 \times 32$
   Output size: $16 \times 16 \times 32$
   Weights: 0 (pooling layers have no learnable parameters)

3. **conv2:** 64 convolutions 5×5, followed by ReLU
   Input size: $16 \times 16 \times 32$
   Output size: $16 \times 16 \times 64$

Weights: Each filter has $5 \times 5 \times 32 = 800$ weights, plus 1 bias per filter.
Total weights = $64 \times (5 \times 5 \times 32 + 1) = 64 \times 801 = 51264$

4. **pool2:** Max-pooling 2×2
   Input size: $16 \times 16 \times 64$
   Output size: $8 \times 8 \times 64$
   Weights: 0

5. **conv3:** 64 convolutions 5×5, followed by ReLU
   Input size: $8 \times 8 \times 64$
   Output size: $8 \times 8 \times 64$
   Weights: Each filter has $5 \times 5 \times 64 = 1600$ weights, plus 1 bias per filter.
   Total weights = $64 \times (5 \times 5 \times 64 + 1) = 102464$

6. **pool3:** Max-pooling 2×2
   Input size: $8 \times 8 \times 64$
   Output size: $4 \times 4 \times 64$
   Weights: 0

7. **fc4:** Fully-connected with 1000 output neurons, followed by ReLU
   Input size: Flattened from $4 \times 4 \times 64 = 1024$
   Output size: 1000
   Weights: $1024 \times 1000 = 1024000$ weights, plus 1000 biases.
   Total weights = 1025000

8. **fc5:** Fully-connected with 10 output neurons, followed by softmax
   Input size: 1000
   Output size: 10
   Weights: $1000 \times 10 = 10000$, plus 10 biases.
   Total weights = 10010

Layer	**Output Size**	**Weights**
conv1	$32 \times 32 \times 32$	2432
pool1	$16 \times 16 \times 32$	0
conv2	$16 \times 16 \times 64$	51264
pool2	$8 \times 8 \times 64$	0
conv3	$8 \times 8 \times 64$	102464
pool3	$4 \times 4 \times 64$	0
fc4	1000	1025000
fc5	10	10010

Table 2.1: Layer specifications with output sizes and number of weights

The convolutional layers have a relatively moderate number of parameters compared to the fully connected layers (13.6% of the total). This is because they use shared weights across the spatial dimensions, which limits the number of parameters.

Q11. What is the total number of weights to learn? Compare that to the number of examples.

Adding up all the weights gives a total of **1.186.170 parameters**. Since we only use the training set for learning, the total number of examples used to train the network is **50.000**. With so many parameters compared to the number of examples, the model is at risk of overfitting, especially in the fully connected layers where most of the parameters are concentrated. However, if properly regularized, this high number of parameters could also allow the model to learn complex patterns within the data, which can improve performance if generalization is controlled.

Q12. Compare the number of parameters to learn with that of the BoW and SVM approach.

BoW Vocabulary: This process does not involve learned parameters in the same way as a CNN, but it results in a feature vector of fixed size (e.g., 1,000 dimensions) for each image.
SVM Classifier: For a linear SVM with 10 classes (as in CIFAR-10), we need to learn a set of weights for each class. With 1,000 features, the total number of parameters for the SVM is:

$$1,000 \times 10 = 10,000 \text{ weights} + 10 \text{ biases} = 10,010 \text{ parameters}$$

BoW + SVM is far more parameter-efficient but relies on manual feature extraction, whereas CNNs have much higher learning capacity but require more data and regularization to avoid overfitting due to their vast number of parameters.

2.2.2 Network Learning

Q14. 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)?

The main difference between calculating loss and accuracy during training and testing is as follows:

- **In Training:** Loss and accuracy are typically calculated *per batch* of data, and the model parameters are updated after each batch. This per-batch calculation helps the model adjust its parameters continuously throughout each epoch.
- **In Testing:** Loss and accuracy are calculated over the *entire test set* without parameter updates. The test loss gives an overall measure of performance across all test samples, and test accuracy is computed as a single value for the whole dataset.

Q16. What are the effects of the learning rate and the batch size?

Effect of varying the learning rate:
A higher learning rate can speed up training but may cause the model to skip optimal points, potentially leading to instability or divergence. A lower learning rate results in slower, more stable convergence but may get stuck in local minima or make training too slow.

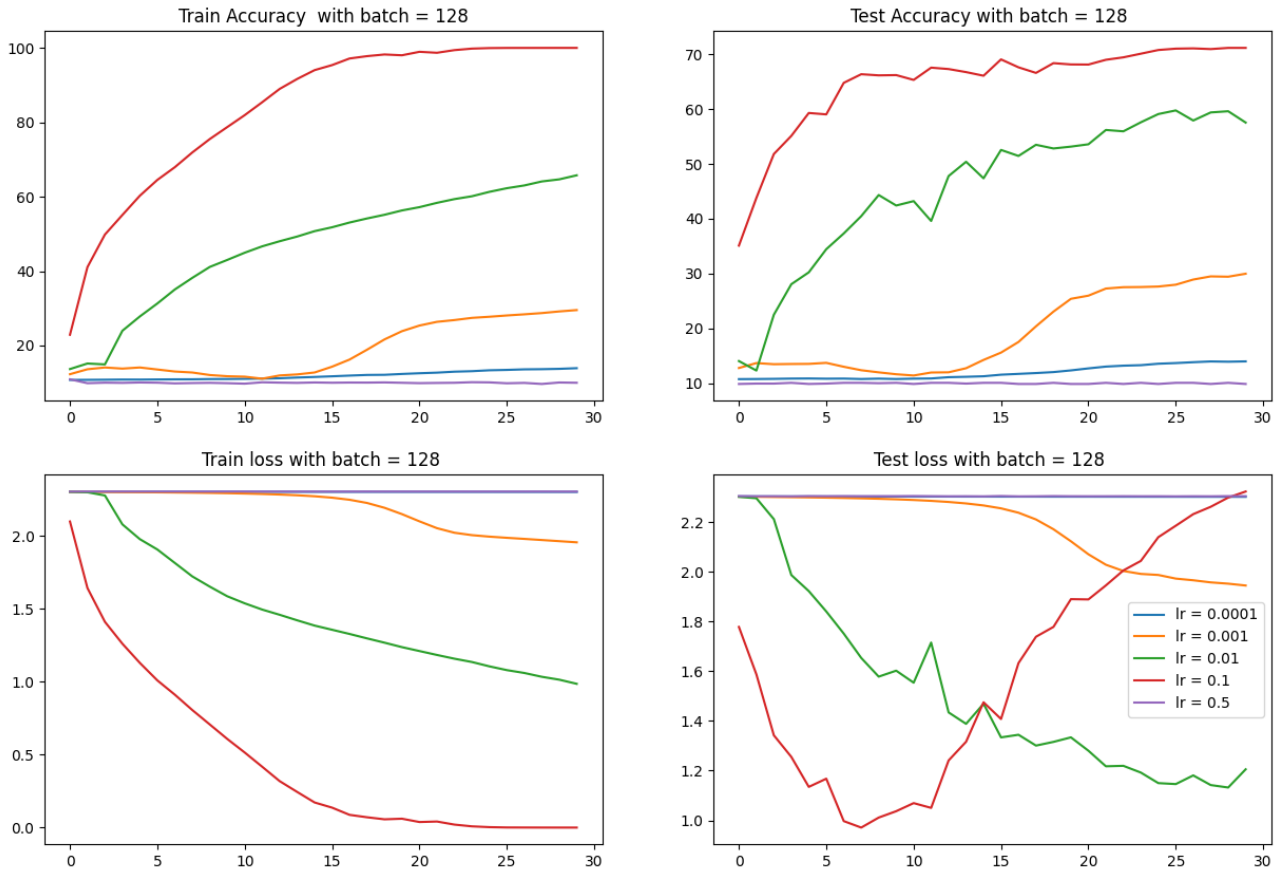

Figure 2.1: Effect of varying the learning rate with a fixed batch size of 128.

Our results are consistent with theoretical expectations. Specifically, with a very low learning rate (0.0001), training is extremely slow, likely due to the presence of local minima. Conversely, with a learning rate that is too high (0.5), the model fails to learn effectively as it skips over optimal points. Training is most stable with a learning rate of 0.01. At a learning rate of 0.1, the model achieves good accuracy but begins to overfit, which is evident in the red test loss curve. It's also worth noting that these results are influenced by the batch size, which is set to 128 in this case.

Effect of varying the Batch Size:
A smaller batch size introduces more noise in parameter updates, leading to noisier but potentially more robust convergence by exploring diverse regions of the loss landscape. A larger batch size provides smoother and more stable updates, but with risk of getting stuck in sharp

local minima. Moreover, larger batch sizes require more memory while smaller batches require less memory and may allow training on less powerful hardware.

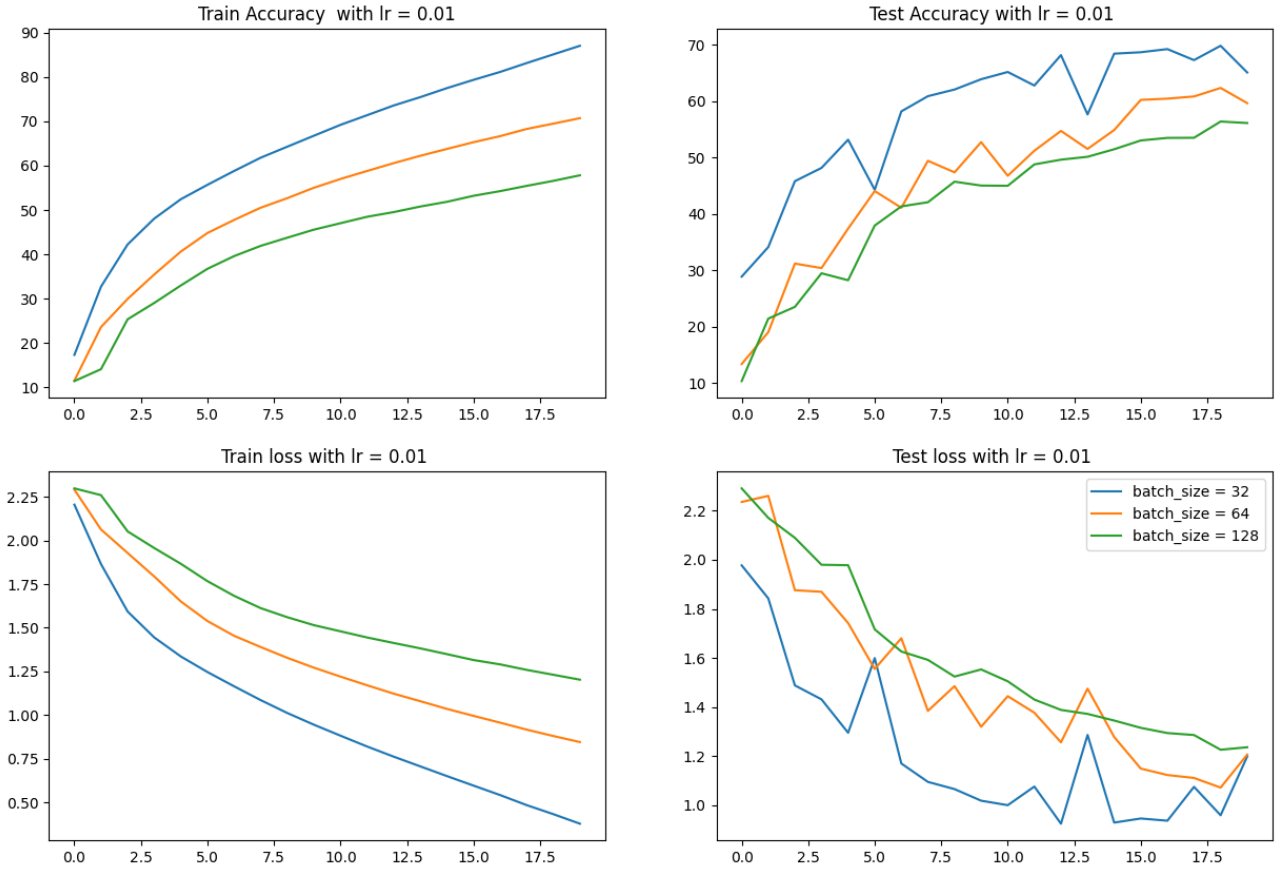

Figure 2.2: Effect of varying the Batch Size with a fixed learning rate of 0.01.

It can be observed that as the batch size increases, the training becomes more stable and smoother. However, with a low learning rate fixed at 0.01, the best results are achieved with a batch size of 32, which allows for better exploration of the loss landscape.

Q17. What is the error at the start of the first epoch, in train and test? How can you interpret this?

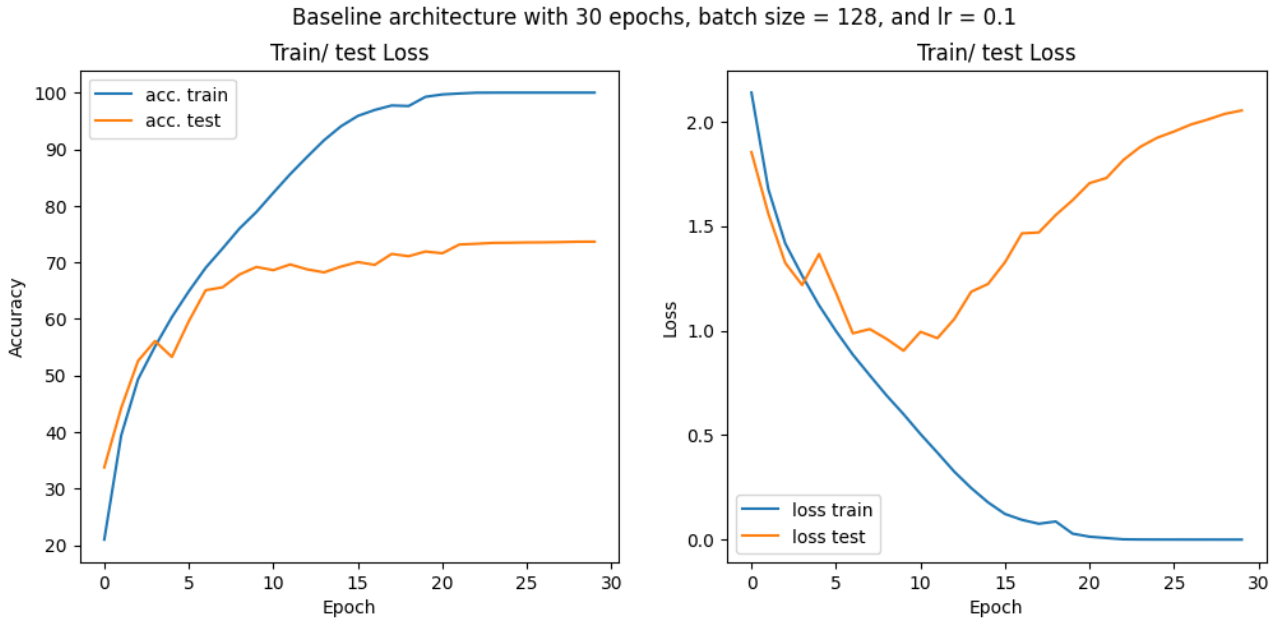

Figure 2.3: Baseline architecture with a learning rate of 0.1 and a batch size of 128.

We observe that, in the first epoch, the error on the training data is higher than on the test data. This can be interpreted by the fact that, during the early epochs, the model is still adjusting to the training data and has not yet learned meaningful features. Additionally, because the test set is smaller, it may exhibit less variability compared to the training set.

Q18. Interpret the results. What's wrong? What is this phenomenon?

We can observe in the loss curves a phenomenon of overfitting, as the loss on the test set increases from around the tenth epoch. This is confirmed by the accuracy curves, where we achieve 100% accuracy on the training set, while the test accuracy stagnates at 70%. This indicates that the model is overfitting and is unable to generalize to the test set.

2.3 Results Improvements

2.3.1 Standardization of examples

Q19. Describe your experimental results.

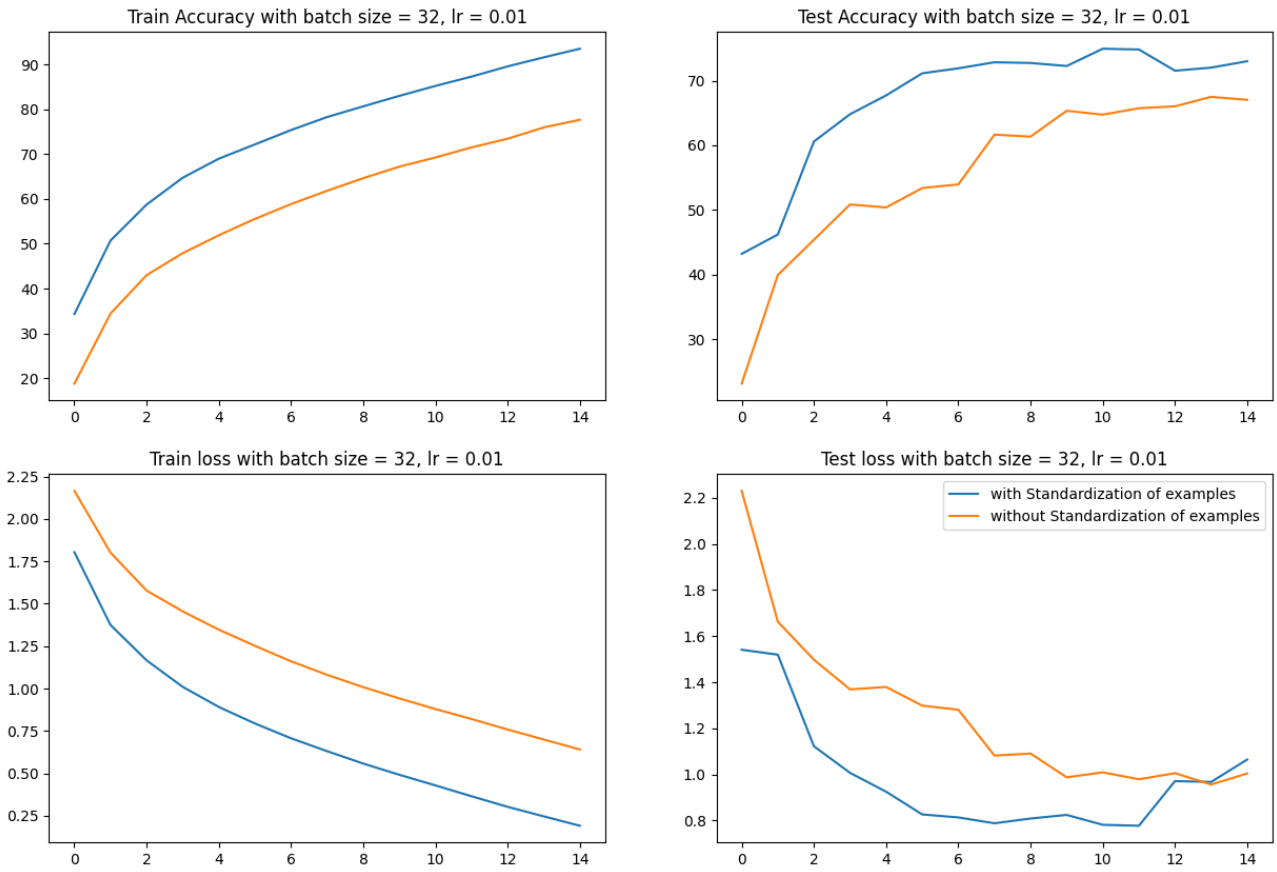

Figure 2.4: Standardization of examples with a batch size of 32 and a learning rate of 0.01

Standardizing images by channel mean and standard deviation centers and scales the data, improving learning by ensuring that the model trains on inputs with a consistent range and distribution, which accelerates convergence and ensures stability.

Q20. Why only calculate the average image on the training examples and normalize the validation examples with the same image?

We calculate the mean and standard deviation on the training set only to avoid *data leakage* and ensure *consistency*. This way, the model sees data in a stable range during both training and validation, which improves generalization to unseen data.

Q21. Bonus: Other Normalization Schemes

There are other normalization schemes, such as **ZCA (Zero-phase Component Analysis) normalization**, which can enhance feature decorrelation and sometimes improve convergence.

ZCA maintains image structure while standardizing, making it suitable for visual tasks.

We will test **L1** and **L2 normalization** on the data to examine the results. Briefly, L1 normalization divides each value by the sum of absolute values of all values, while L2 normalization divides each value by the square root of the sum of squares of all values.

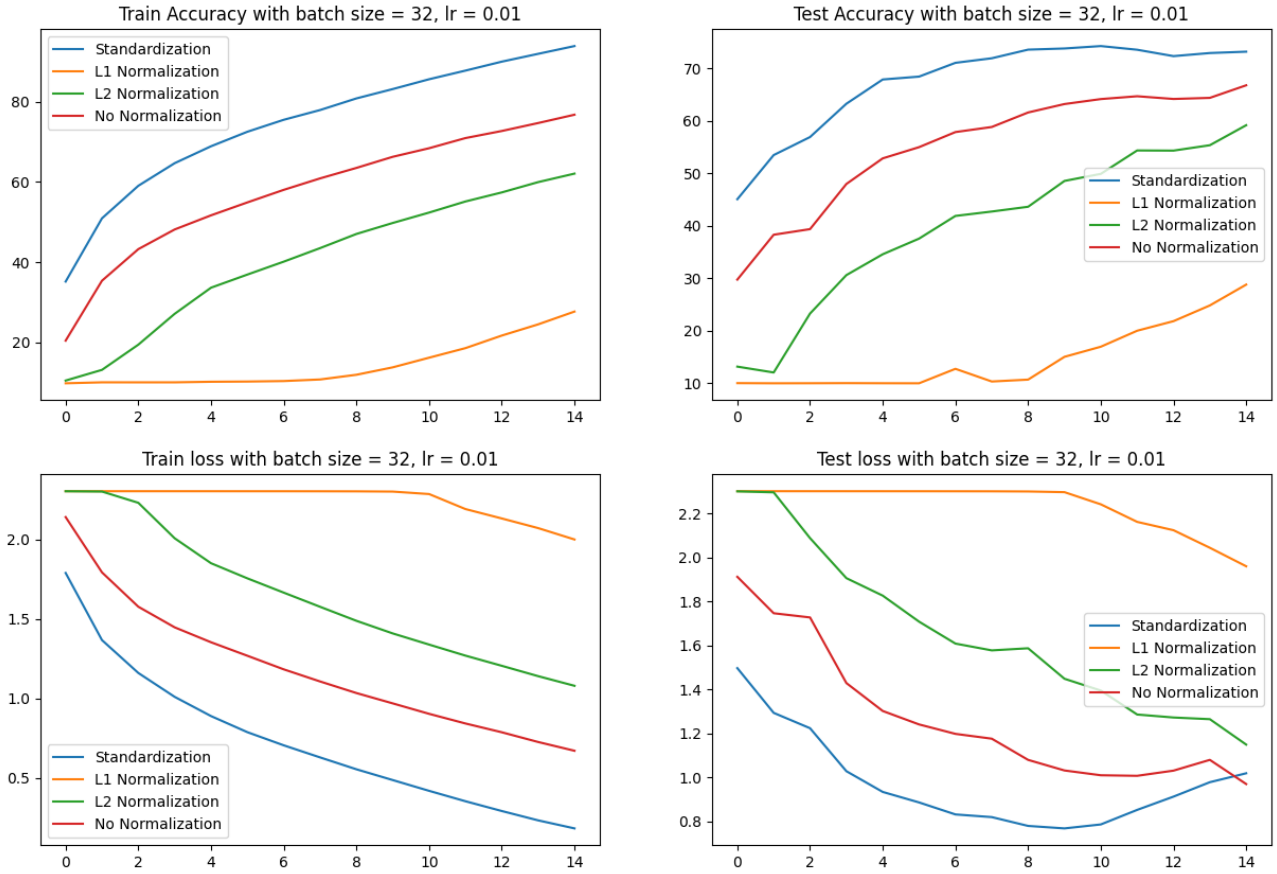

Figure 2.5: Effects of other types of normalization

We can see that L1 and L2 normalizations did not improve our results, unlike standardization. This can be explained by the fact that, for images with pixel values already between 0 and 1, L1 and L2 normalizations tend to uniformize the values, reducing contrasts between pixels and potentially altering important features for classification.

2.3.2 Increase in the Number of Training Examples by Data Augmentation

Q22. Describe your experimental results and compare them to previous results.

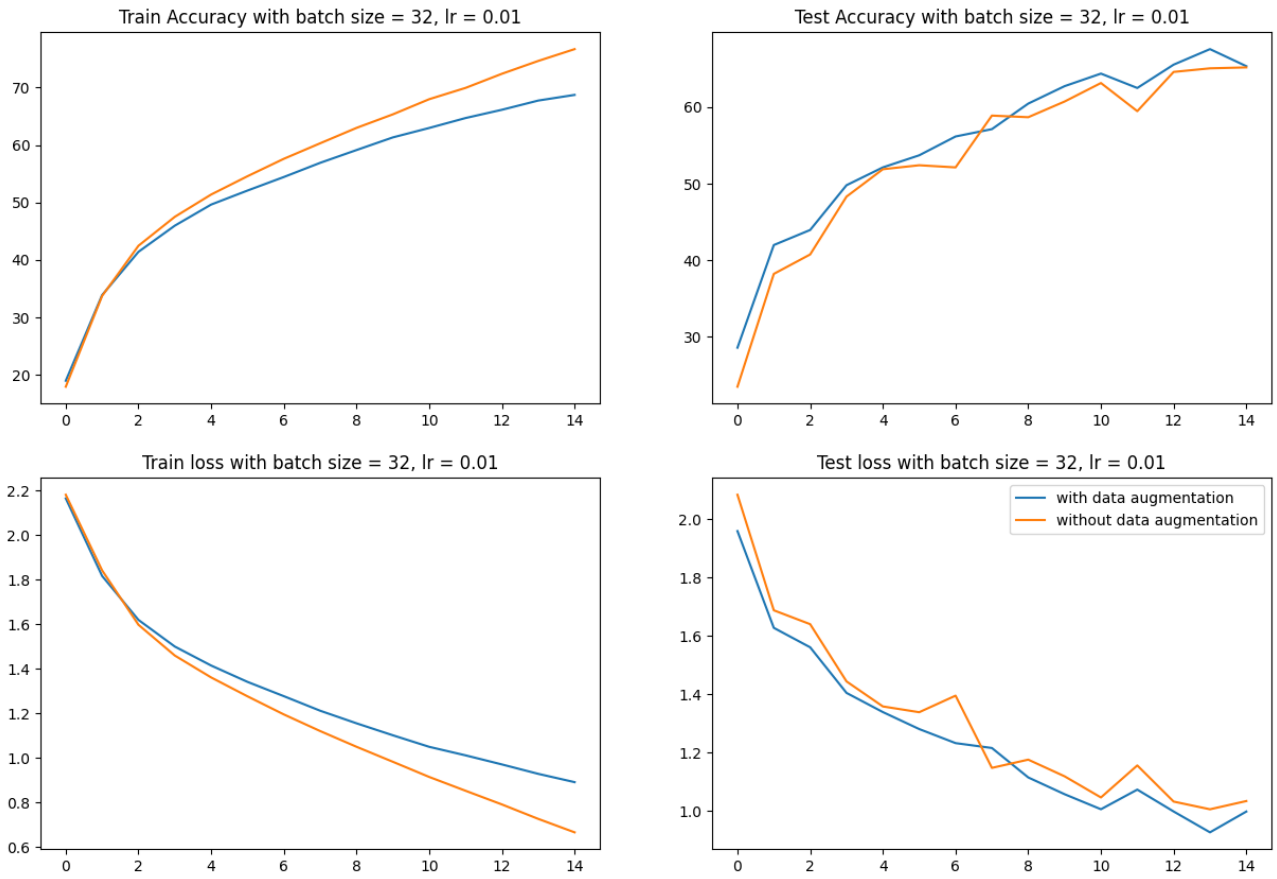

Figure 2.6: Effect of Data Augmentation

We can see that data augmentation improved the results on the test set by diversifying the training set, which helps the model generalize better by making it more robust to variations in the data.

Q23. Is the Horizontal Symmetry Approach Usable for All Types of Images?

The horizontal symmetry approach is not suitable for all types of images. It is effective for images where flipping does not alter the semantics, such as landscapes. However, it is unsuitable for images with inherent directionality or asymmetry, such as text or medical images. In these cases, horizontal flipping would misrepresent the content, potentially leading to incorrect learning and poorer generalization.

Q24. What Limits Do You See in This Type of Data Increase by Transformation?

This type of data augmentation by transformations has several limitations:

- Random crops and horizontal flips only slightly modify the dataset without introducing new variations in content.
- The model may learn to rely on specific transformations, such as cropped areas or symmetry, rather than more robust features.
- Augmentation does not generate new classes or scenarios; it only modifies existing images, which may not sufficiently help when the primary challenge is limited label variety or underrepresented classes in the dataset.

Q25. Bonus: Other Data Augmentation Methods

Other common data augmentation methods include:

- Adding random noise to simulate variations.
- Masking random sections of the image.
- Rotating images by small random angles.
- Altering brightness, contrast, saturation, or hue.
- More advanced techniques include **Generative Adversarial Networks (GANs)** and **Autoencoders**, which can generate realistic synthetic images, enhancing the dataset diversity.

We test the transformation that applies a random rotation and another that applies a Gaussian filter of size 3x3. We obtain the following results:

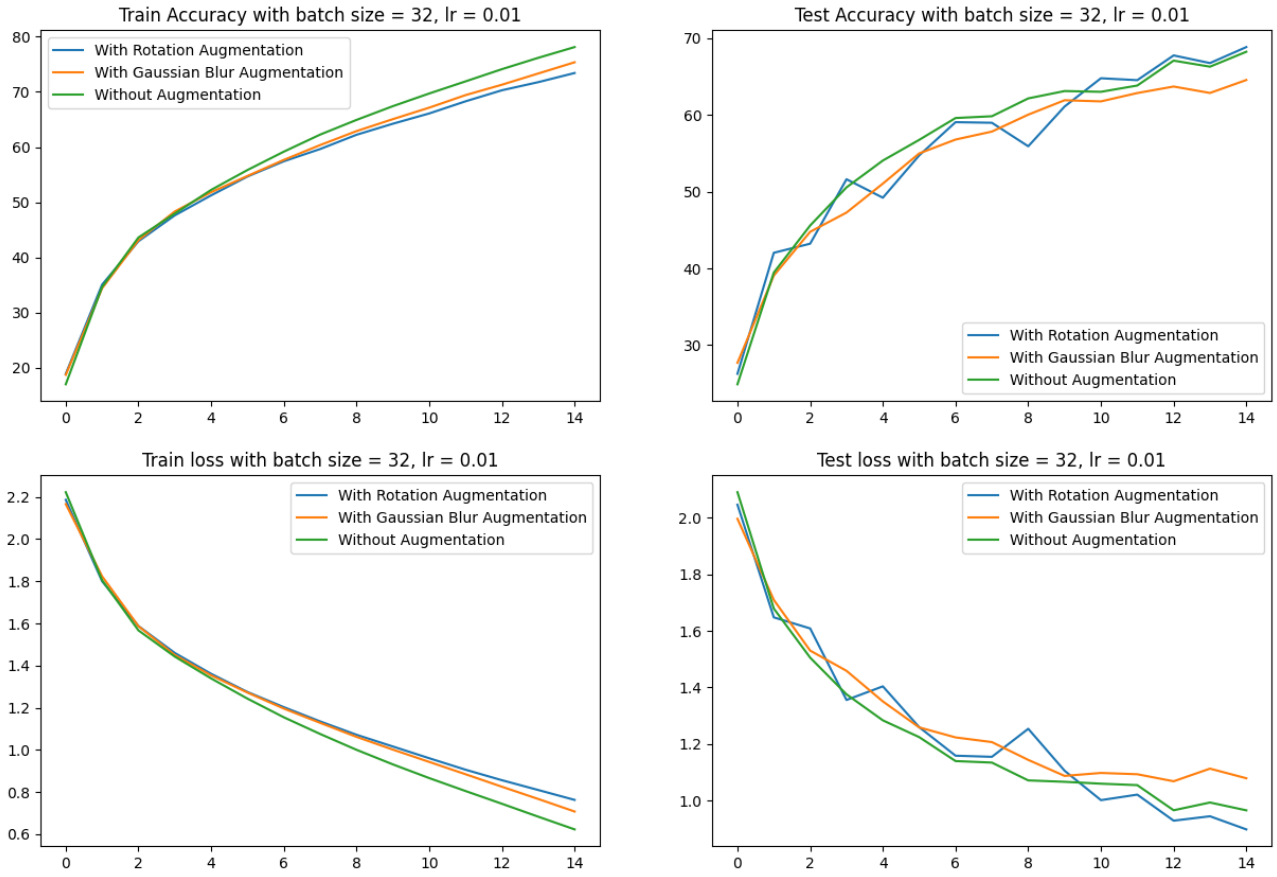

Figure 2.7: Effects of other types of Data Augmentation

We can observe that these data augmentation methods help reduce the effect of overfitting and lead to improved results on the test set by enhancing the model's ability to generalize.

2.3.3 Variants on the Optimization Algorithm

Q26. Describe your experimental results and compare them to previous results, including learning stability.

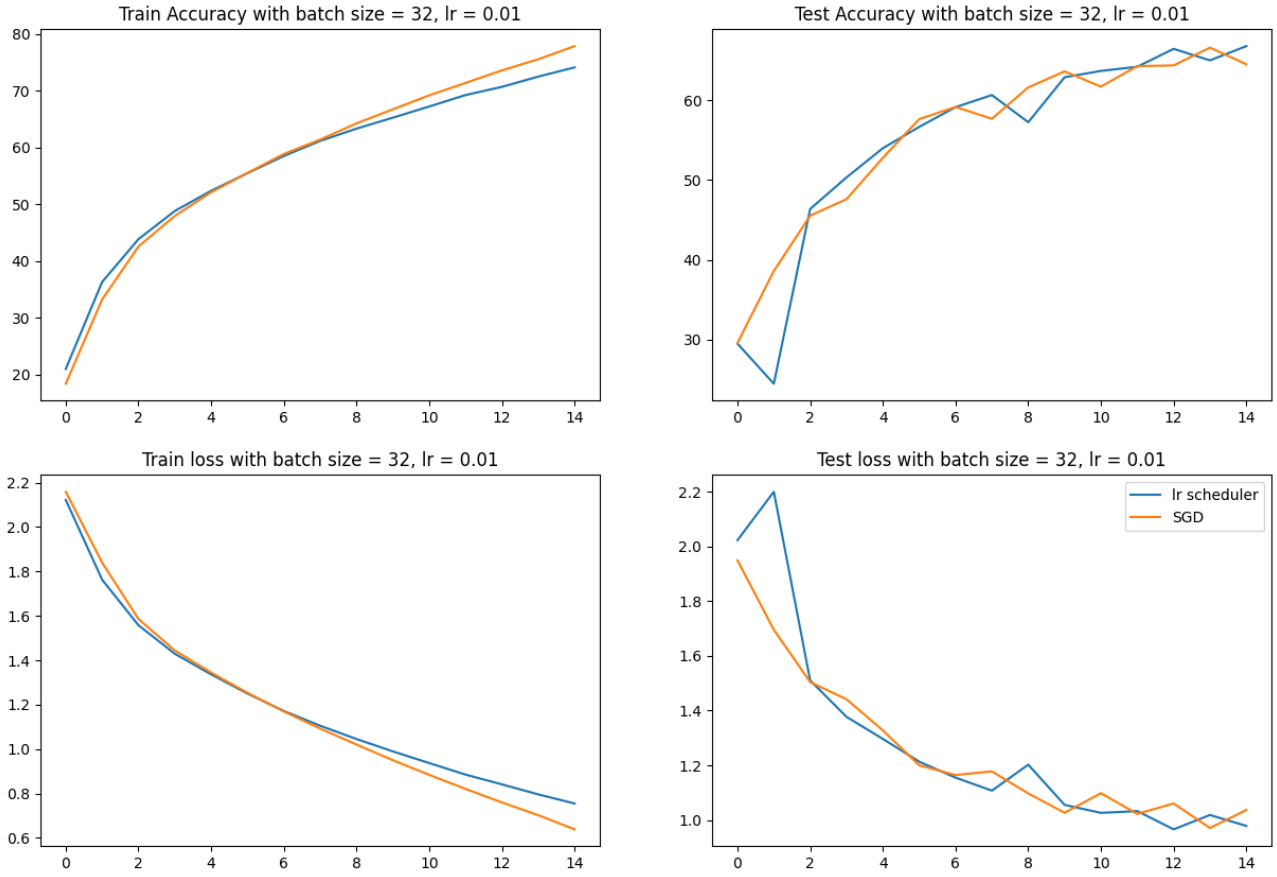

Figure 2.8: Effect of Using a Learning Rate Scheduler

In our case, the results are very similar using the two different approaches.

Q27. Why Does This Method Improve Learning?

This method improves learning by gradually reducing the learning rate over time. Starting with a higher learning rate enables the model to explore the parameter space more broadly and avoid getting stuck in local minima. As the learning rate decreases, the updates become smaller, helping the model settle into a stable, more optimal solution.

Q28. Bonus: Variants of SGD and Learning Rate Scheduling

Many other SGD variants exist, such as **Adam** and **RMSprop**, which adapt the learning rate per parameter.

2.3.4 Regularization of the Network by Dropout

Q29. Describe your experimental results and compare them to previous results.

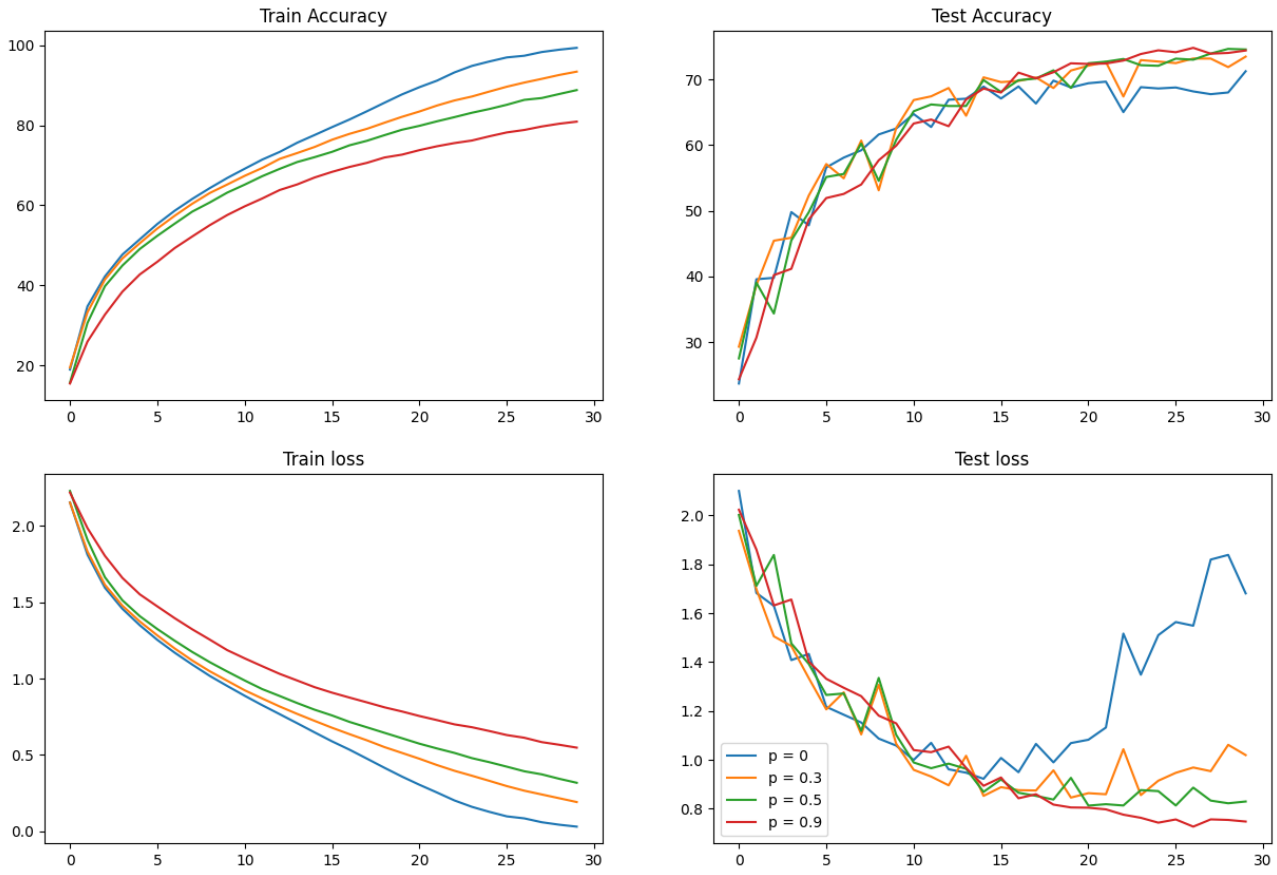

Figure 2.9: Effect of dropout with batch size = 32 and learning rate = 0.01

We can observe that dropout helps reduce the effect of overfitting (Blue Test Accuracy Curve) by improving the model's generalization on test data. Note that if the dropout rate is too high, the model may struggle to learn important features due to the excessive deactivation of neurons during training. However, in this case, we have a sufficient number of neurons to maintain model performance, even with a 90% dropout rate.

Q30. What is Regularization in General?

Regularization is a technique to prevent overfitting by adding constraints or modifications during training that encourage the model to learn simpler, more general patterns instead of memorizing the training data.

Q31. Discuss Possible Interpretations of Dropout's Effect on Network Behavior

Dropout forces neurons to learn more independently, preventing over-reliance on specific features. Moreover, by adding noise during training, it encourages the network to develop more robust representations, which helps reduce overfitting.

Q32. What is the Influence of the Dropout Hyperparameter?

The hyperparameter of the dropout layer controls the probability of deactivating each neuron during training. Higher values increase regularization, while lower values reduce it.

Q33. What is the Difference in Behavior of the Dropout Layer Between Training and Testing?

During training, dropout randomly deactivates neurons to improve generalization, while during testing, all neurons are active for stable predictions.

2.3.5 Use of Batch Normalization

Q34. Describe your experimental results and compare them to previous results.

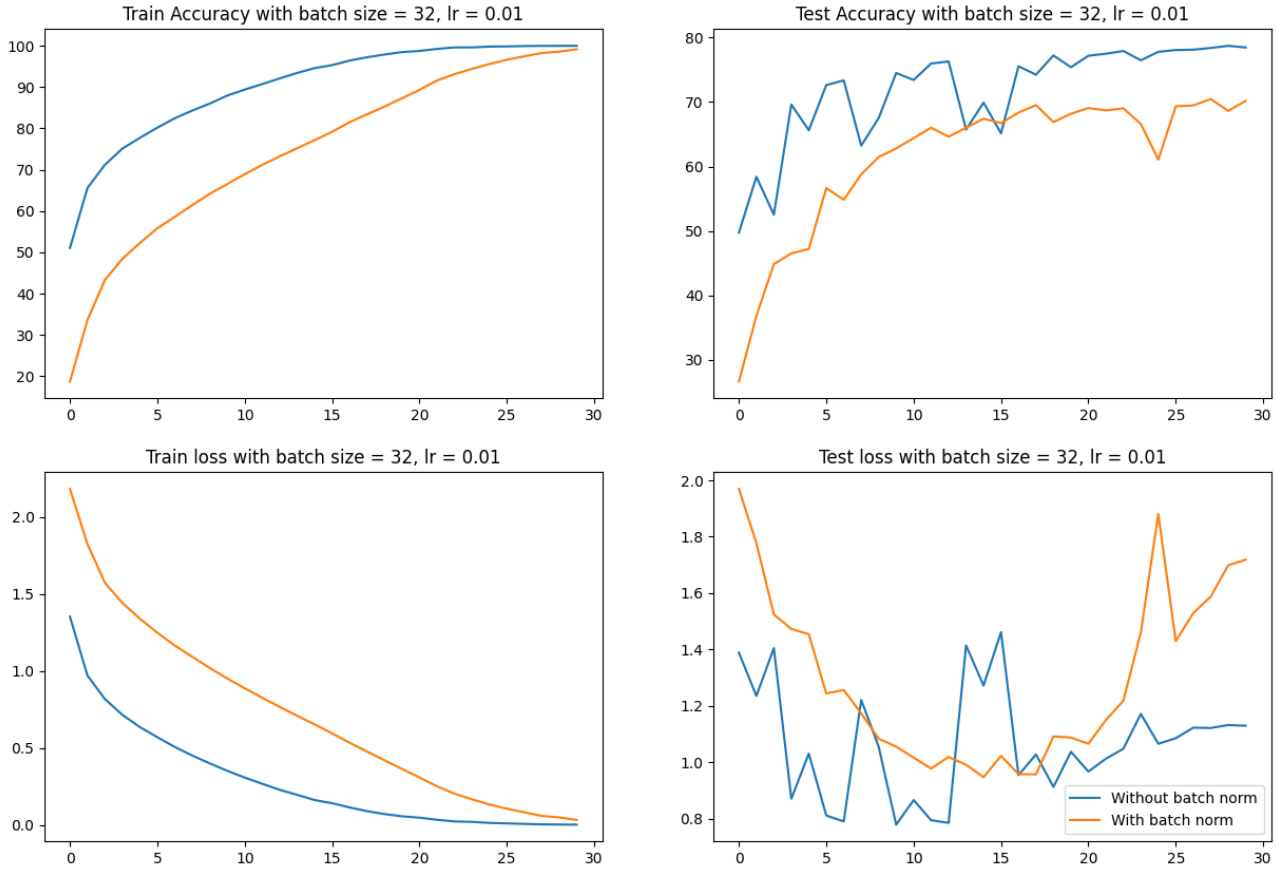

Figure 2.10: Effect of batch normalization with a batch size of 32 and a learning rate of 0.01

Although batch normalization is intended to stabilize learning by normalizing the outputs of intermediate layers, normalizing the data on each batch caused the model to fit too closely to the training data, leading to overfitting. As a result, batch normalization did not improve the model's performance in this scenario.

2.3.6 Application of Dropout and Standardization

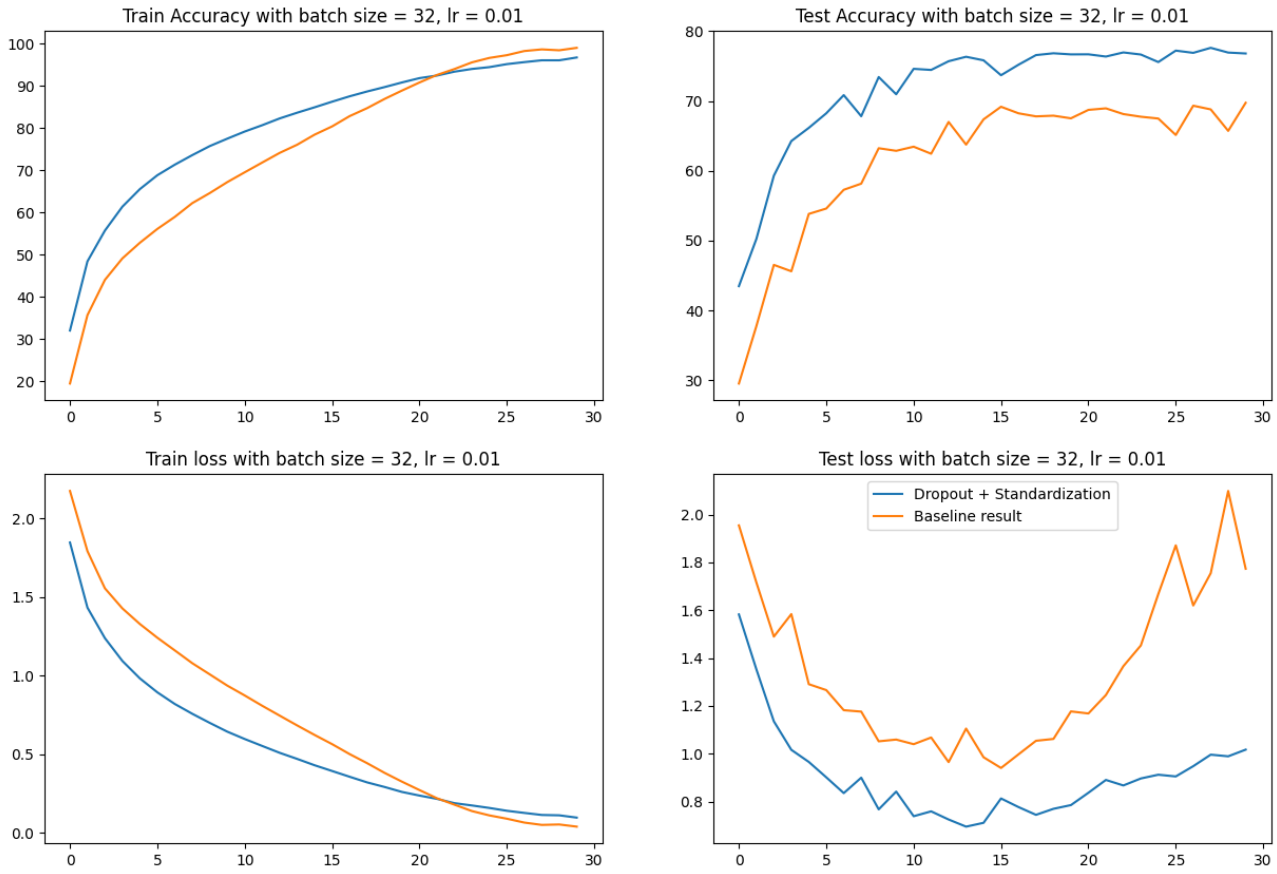

Figure 2.11: Combined effect of dropout and standardization with a Batch Size of 32 and a Learning Rate of 0.01

The combined effect of dropout and standardization helped to reduce overfitting and improve performance on the test set compared to the baseline architecture. Dropout introduces regularization by randomly deactivating neurons during training, which reduces the model's dependency on specific features of the training set and encourages better generalization. Standardization, on the other hand, normalizes the input data, stabilizing the training process and facilitating convergence. Together, these techniques create a more robust model that is better able to generalize to the test data.

2.4 Conclusion

In this chapter, we explored the foundational structure and advantages of Convolutional Neural Networks (CNNs), which are particularly effective in capturing spatial hierarchies in image data through convolutional and pooling layers. CNNs leverage local connectivity, weight sharing, and translation invariance to efficiently recognize patterns in images with fewer parameters than fully connected networks. The chapter also covered techniques to train CNNs effectively, highlighting the importance of architectural choices, such as kernel sizes, padding, and pooling, to control the spatial dimensions and ensure model efficiency. Furthermore, methods to improve CNN performance were investigated, including standardization, data augmentation, and various regularization strategies like dropout and batch normalization. The combination of these techniques contributed to a more robust model capable of generalizing better to unseen data.

Chapter 3

Transformers

Note : All comments and observations about the experiments are included in the Jupyter notebook.

3.1 Self-attention

Q1. What is the main feature of self-attention, especially compared to its convolutional counterpart ?

Self-attention captures global dependencies by considering the relationship between each patch and every other within the image, enabling the model to gather contextual information from the entire input. In contrast, convolutional layers focus on local information by computing relationships between neighboring pixels within a limited window, thus limiting the scope to local features.

Q2. What is its main challenge in terms of computation/memory?

The main challenge in terms of computation and memory lies in the quadratic time and space complexity of self-attention: $O(n\check{s})$ relative to the number of input tokens. The process requires calculating attention scores between every pair of patches, which significantly increases both the computation time and memory usage as the input size grows. This limitation is particularly challenging for high-resolution images, such as biomedical images.

3.2 Full ViT model

Q3. Explain what is a Class token and why we use it?

A class token is an artificial token added to the image tokens that is optimized throughout the learning process alongside the calculations of the self-attention, in order to capture a global representation of the image. This token summarizes all the information learnt into a single vector, which can be used for downstream tasks such as classification by aggregating the most relevant features of the image into a compact form.

Q4. Explain what is the positional embedding (PE) and why it is important?

The positional embedding is a representation of the absolute position of the patch in the image which is additioned to the inputs of the transformer. It is not learned and can be as simple as

the actual number of the position of the patch in the image, or it can be calculated through a specific function such as sinusoidal ones. Such addition is useful because Transformers capture relative information between the patches, but don't naturally keep track of the inherent spatial information of each one of them.

3.3 Larger Transformers

Q5. Load the model using the timm library without pretrained weights. Try to apply it directly on a tensor with the same MNIST images resolution. What it is the problem and why we have it?

The problem is that MNIST images (28*28*1) are not compatible with the architecture of the model, which has been designed for the ImageNet dataset composed of (224*224*3) images. We need to resize our images to be able to fit in that dimension, by duplicating the gray scale channel and applying an interpolation. It is important to note that this kind of manipulation can alter the quality and integrity of the images, which can in turn lead to lower performances.

Q6. Explain if we also have such a problem with CNNs.

Thanks to the convolutional layers, this represents less of a problem because they are size agnostic. Nevertheless, we need to assure that the number of channels is compatible with the one the model was intended for. The problem arises when we reach the fully connected layer of the CNN, whose neurons are determined by the number of outputs of the Conv layers, which may vary according to the size of the image. A solution to this can be substituting the Fully-Connected layers with Global Average Pooling layers, which average each feature map into one value, creating a vector whose size depends on the internal parameters of the model, rather than an external factor.

Q7. Comment the final results and provide some ideas on how to make the transformer work on small datasets. You can take inspiration from some recent work.

The final results showcase the benefits of transfer learning, as we can create better models in a more efficient way by training one generalist model, and then modifying it to suit our specific task. This method is often used with models such as ViT, Bert, and ResNet.

Although Transformers fundamentally require large amounts of data, there are plenty of ways adapt adapt them to smaller datasets:

- **Fine-tuning**: Instead of beginning with random weights, we can start with a model pre-trained on a large, general-purpose dataset. This way, the model already knows basic, low-level features and image primitives and only needs to learn the specific characteristics of the small dataset. To further reduce the risk of overfitting, we can freeze the lower layers

during training to preserve the low-level information contained in them. Regularization techniques like dropout and weight decay can also be considered.

- **Data augmentation**: Another technique we can use to tackle the small amount of data problem is data augmentation, which is widely used in CNNs. It allows the creation of artificial inputs from the dataset by applying transformations like crops, rotations, contrast enhancements etc. to the images. This helps the model to generalize as it is not restricted to the images of the initial dataset, while still seeing relevant and coherent examples that are in the same initial distribution. Another way of doing data augmentation can be by using generative models like GANs or diffusion models, that can create images by applying realistic transformations encountered in their training data.