Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours - Données Apprentissage et Connaissances

Reconnaissance Des Formes pour l'analyse et l'interprétation d'images

Project report

Basics on deep learning for vision

Done by:
Samy NEHLIL - Allaa BOUTALEB
Supervised by:
Nicolas Thome

November 2023

Contents

List of Figures

Chapter 1

Introduction to Neural Networks

1.1 Introduction to Neural Networks

1.1.1 The use of train, val, and test sets

- **Train Set:** Used for training the model by adjusting its weights and biases.
- **Validation (val) Set:** Used for tuning hyperparameters and preventing overfitting during training.
- **Test Set:** Provides an unbiased evaluation of the model's performance on new data.

1.1.2 Influence of the number of examples N

- **Too Few Examples:** Can lead to overfitting due to insufficient data.
- **Adequate Number of Examples:** Enables the model to learn diverse patterns, improving generalization.
- **Too Many Examples:** Beneficial for deep models but may increase training times.

1.1.3 Importance of Activation Functions

Activation functions introduce non-linearity, enabling the network to model complex relationships. Without them, multiple linear layers would collapse into a single linear transformation.

1.1.4 Sizes n_x, n_h, and n_y

n_x is the input size, n_h is the hidden layer size, and n_y is the output size. In practice, n_x and n_y are determined by the data and problem, while n_h is chosen based on model complexity and experimentation.

1.1.5 Vectors $\hat{y}$ and y

$\hat{y}$ is the predicted output of the network, while y is the true target or ground truth. The difference is that $\hat{y}$ is produced by the model, while y is provided in the dataset.

1.1.6 Use of SoftMax Function

SoftMax converts the raw output scores into probabilities, ensuring they sum to 1. It's especially useful for multi-class classification tasks.

1.1.7 Mathematical Equations for Forward Pass

$$\begin{aligned}\tilde{h} &= W_h \times x + b_h \\ h &= \tanh(\tilde{h}) \\ \tilde{y} &= W_y \times h + b_y \\ \hat{y} &= \text{SoftMax}(\tilde{y})\end{aligned}$$

1.1.8 Variation of y_i to minimize the Loss Function

- **Cross-Entropy:** To minimize the loss, $\hat{y}_i$ should approach the true label y_i. If y_i is 1, $\hat{y}_i$ should be close to 1, and if y_i is 0, $\hat{y}_i$ should be close to 0.
- **Squared Error:** The predicted value $\hat{y}_i$ should be as close as possible to the true value y_i to minimize the squared difference.

1.1.9 Suitability of Loss Functions

- **Cross-Entropy:** Suited for classification tasks because it measures the difference between two probability distributions - the true label and the predicted label.
- **Mean Squared Error (MSE):** Suited for regression tasks as it measures the average squared difference between the predicted values and the actual values.

1.1.10 Advantages and disadvantages of variants of GD

Method	**Advantages**	**Disadvantages**
Batch Gradient Descent	Stable gradient; Global convergence for convex surfaces	Slow for large datasets; Stuck at local minima
Mini-batch SGD	Faster convergence; Escapes local minima	Noisy updates; Batch size tuning
Online SGD	Fast updates; Adapts to dynamic data	Very noisy; Requires decreasing learning rate

General Recommendation: Mini-batch SGD is often preferred as it balances the benefits of both classic and online SGD.

1.1.11 Influence of Learning Rate η

The learning rate η determines the step size during optimization. A high η can cause overshooting, while a low η can slow down convergence. It's crucial to find an optimal η for efficient learning.

1.1.12 Complexity of Gradient Calculation

- **Naive Approach:** In the naive approach, for each layer, we compute the gradient from scratch, starting from the output layer and working our way back to the input. For a neural network with L layers, the complexity of computing the gradient for each layer separately is proportional to $O(L^2)$ because for each layer, we would recompute gradients for all the subsequent layers.

- **Backpropagation:** Backpropagation efficiently computes gradients by using the chain rule and storing intermediate gradients. This means that once the gradient for a layer is computed, it's reused for the computation of the gradient for the previous layer. This avoids redundant calculations. As a result, the complexity of backpropagation is linear with respect to the number of layers, i.e., $O(L)$.

The backpropagation algorithm is more efficient than the naive approach, especially for deep neural networks with many layers. While the naive approach's complexity grows quadratically with the number of layers, backpropagation grows linearly. This makes backpropagation significantly faster and more scalable for deep networks.

1.1.13 Network Architecture Criteria

The network must be differentiable to use gradient-based optimization. This ensures that every component, including activation functions, has a well-defined derivative.

1.1.14 SoftMax and Cross-Entropy Simplification

The cross-entropy loss for classification tasks is given by:

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

When combined with the SoftMax function, which is defined as:

$$\hat{y}_i = \text{Softmax}(\tilde{y}_i) = \frac{\exp(\tilde{y}_i)}{\sum_j \exp(\tilde{y}_j)}$$

Substituting the SoftMax function into the cross-entropy loss, we get:

$$\begin{aligned}
l(y, \hat{y}) &= -\sum_i y_i \log\left(\frac{\exp(\tilde{y}_i)}{\sum_j \exp(\tilde{y}_j)}\right) \\
&= -\sum_i y_i \left(\tilde{y}_i + \log\left(\frac{1}{\sum_j \exp(\tilde{y}_j)}\right)\right) \\
&= -\sum_i y_i \left(\tilde{y}_i - \log\left(\sum_j \exp(\tilde{y}_j)\right)\right) \\
&= -\sum_i y_i \tilde{y}_i + \sum_i y_i \log\left(\sum_j \exp(\tilde{y}_j)\right) \\
&= -\sum_i y_i \tilde{y}_i + \log\left(\sum_j \exp(\tilde{y}_j)\right)
\end{aligned}$$

1.1.15 Gradient of Loss Relative to $\tilde{y}$

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

This can be represented as the gradient $\nabla_{\tilde{y}} l$ in vector form:

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

1.1.16 Gradient of Loss Relative to b_y, W_y

Starting with $\nabla_{W_y} l$, we can calculate the partial derivative of the loss with respect to $W_{y,ij}$ as follows:

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

This can be expressed as a matrix:

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

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

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

Now, for $k \in [1, n_h]$, $\frac{\partial \tilde{y}_k}{\partial W_{ij}^y}$ can be defined as:

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

Next, we calculate $\frac{\partial l}{\partial \tilde{y}_k}$ using the previously derived expression:

$$\frac{\partial l}{\partial \tilde{y}_k} = \hat{y}_k - y_k$$

Combining these results, we obtain the gradient of $\frac{\partial l}{\partial W_{i,j}^y}$:

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

Therefore, the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} l$ can be represented as $\nabla_{\tilde{y}}^T h$:

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

The gradient of the loss with respect to the output layer bias ∇_{b_y}:

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

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

1.1.17 Calculating $\nabla_{\tilde{h}} l$, $\nabla_{W_h} l$, $\nabla_{b_h} l$

1. To compute the gradient of the loss with respect to $\tilde{h}$ using the chain rule, we have:

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

Let's calculate the two terms involved:

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

Recalling $\frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i = \delta_i^y$, we can compute $\frac{\partial l}{\partial h_k}$ as:

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

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

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

2. The gradient $\nabla_{W^h} l$ is a matrix with elements:

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

We already have $\frac{\partial l}{\partial h_k} = \delta_k^h$, and for the other term $\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h}$ from $\tilde{h}_k = \sum_{j=1}^{n_x} W_{k,j}^h x_j + b_k^h$, we have:

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

So, the gradient $\nabla_{W^h} l$ is given by:

$$\nabla_{W^h} l = \nabla_{\tilde{h}}^T l * x$$

3. Finally, for the gradient with respect to b_i^j, we have:

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

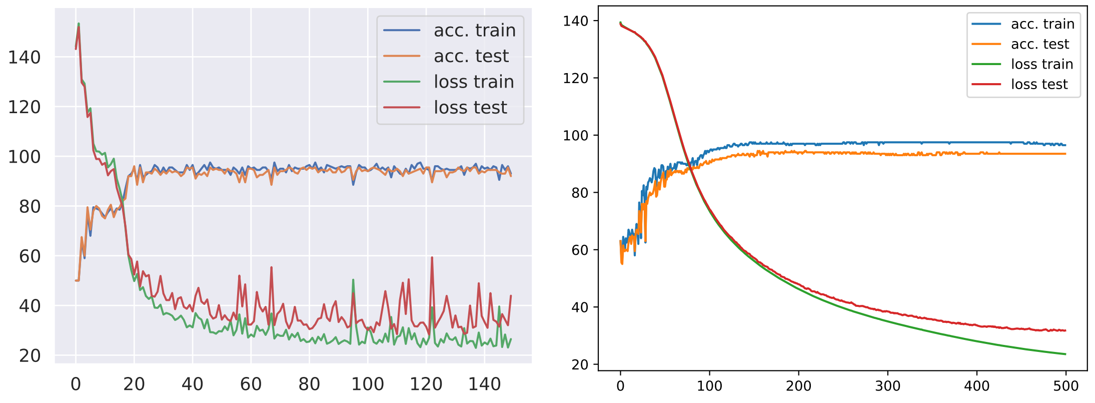
Therefore, the gradient ∇_{b^h} is equal to the gradient with respect to $\tilde{h}$, i.e., $\nabla_{b^h} = \nabla_{\tilde{h}} l$.

1.2 Implementation

In this section, the main emphasis was on investigating the capabilities of a basic neural network model in handling classification tasks that might pose challenges for traditional machine learning methods. The scope of the experiments ranged from working with circular data to tackling image classification problems. The overarching strategy revolved around validating the principles established in the preceding section while also delving into the impact of various hyperparameters on the model's performance.

1.2.1 Manual forward and backpropagation

In the context of neural network training with Stochastic Gradient Descent (SGD), it is a common practice to compute the gradients during the backward pass and then average them across the batch size. This strategy is employed to decouple the learning rate from the batch size, ensuring that updates to the model parameters remain relatively consistent regardless of variations in batch size.

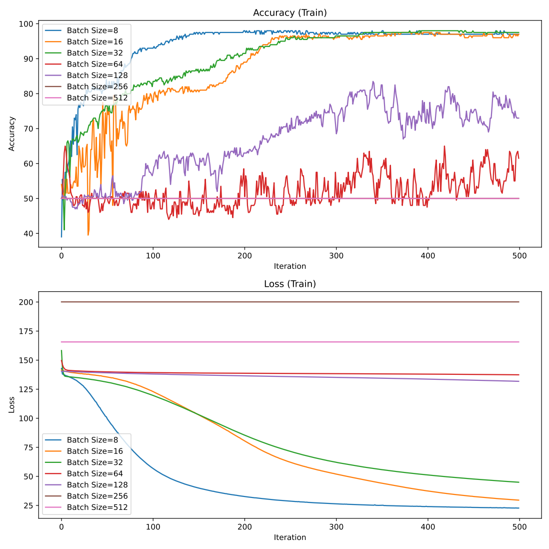


(a) Acc/Loss without batch size averaging (b) Acc/Loss with batch size averaging

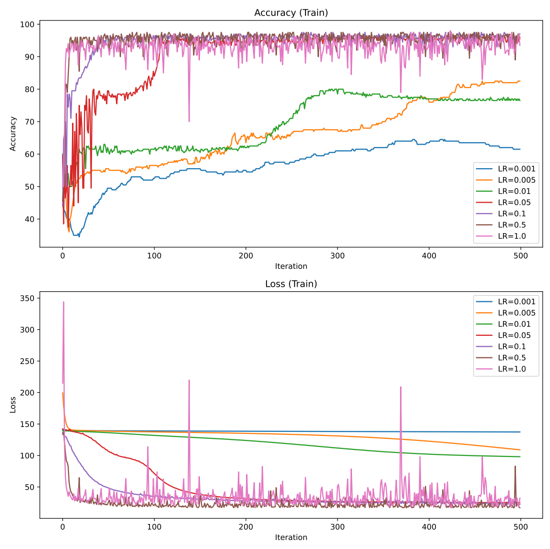
Figure 1.1: Batch vs Averaged Batch

A notable observation is that when batch averaging is utilized, the network's training process demands considerably more time to reach a certain level of performance. Nevertheless, this trade-off leads to enhanced stability during training, as seen in the smoother curves.

In this section, we continued with batch averaging and examined the effects of learning rate and batch size.



(a) Influence of batch size



(b) Influence of learning rate

Figure 1.2: Influence of learning rate and batch size

When the batch size is too big: it can cause the optimization algorithm to fail to converge because the gradient estimates are too precise and lack the stochastic noise that helps escape local minima. This can result in the loss and accuracy metrics not improving at all as the model cannot effectively learn from the data.

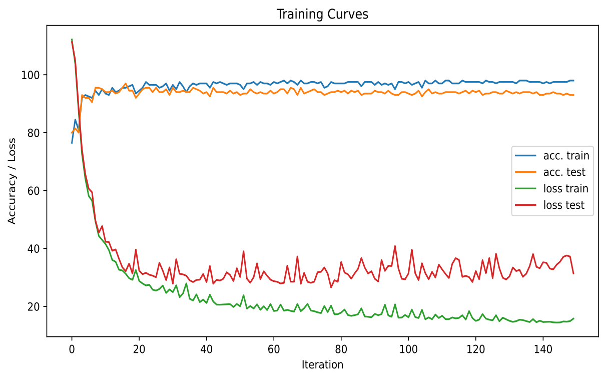
When the batch size is too small: The loss may exhibit significant fluctuations, leading to a noisy convergence path; however, this noise can actually be beneficial as it helps the model escape local minima, which can potentially lead to better generalization.

With a too large learning rate: Can cause the training to be unstable, with loss possibly diverging instead of converging as the updates may overshoot minima.

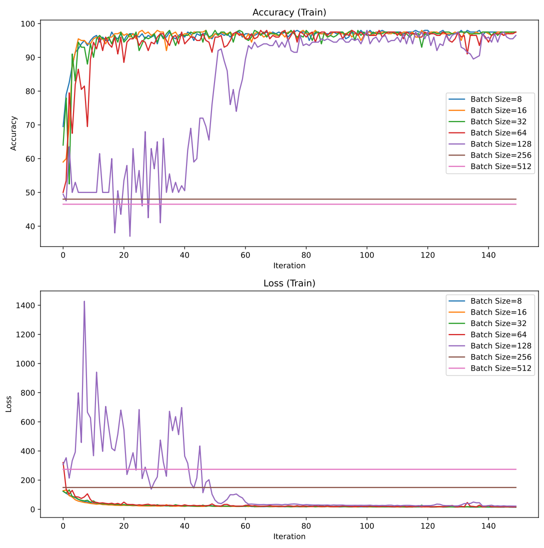
With a too small learning rate: Converges very slowly as updates are too small to make significant progress, potentially getting stuck in local minima.

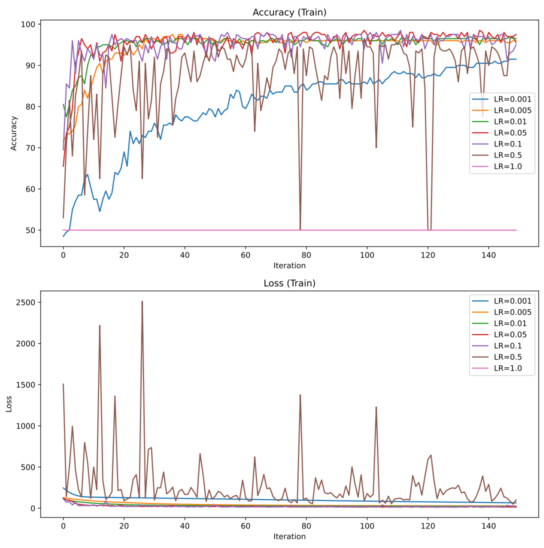
1.2.2 Simplification of the backward pass with `torch.autograd`

It is noteworthy that the results observed were comparable to those from experiments conducted without the use of batch normalization, marked by learning fluctuations.



(a) Acc/loss with `torch.autograd`



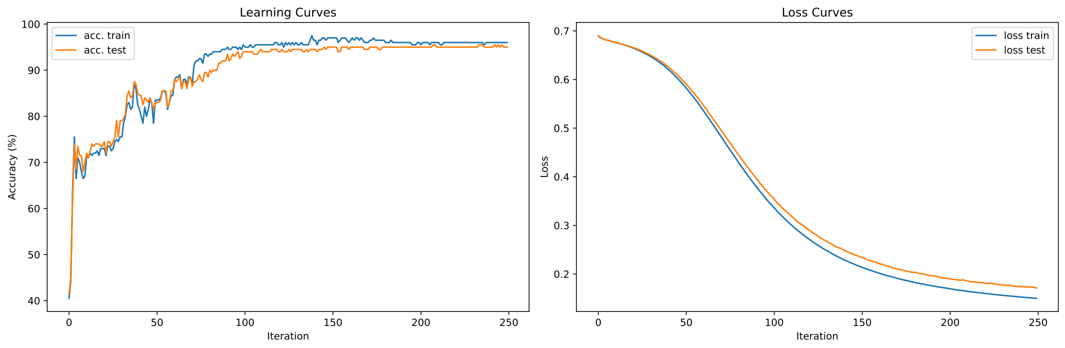


(b) Influence of batch size (c) Influence of learning rate

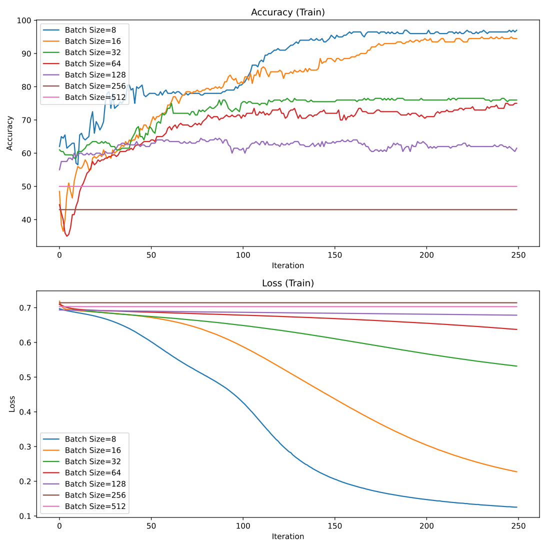
Figure 1.3: Results using `torch.autograd`

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

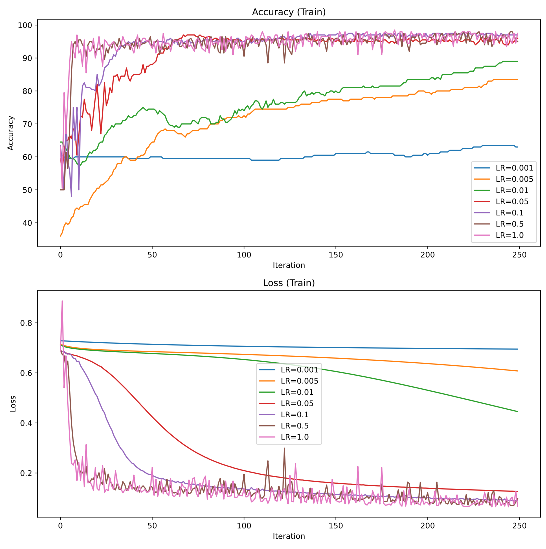
Here, we noted an extended learning duration relative to the employment of handcrafted layers, with a need for 250 epochs to attain peak performance. This extended duration may be due to the initialization methods employed by Torch for its layers. Despite this, the training exhibited greater stability, which is reflected in the behavior of the loss curve.



(a) Acc/loss using `torch.nn` layers



(b) Influence of batch size

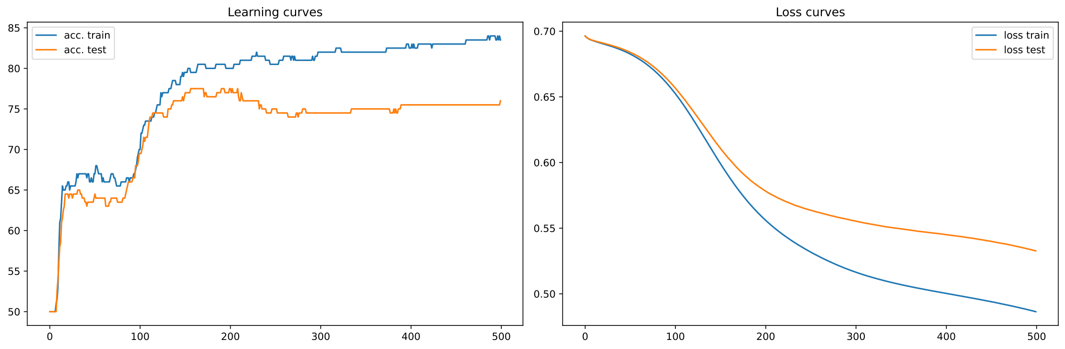


(c) Influence of learning rate

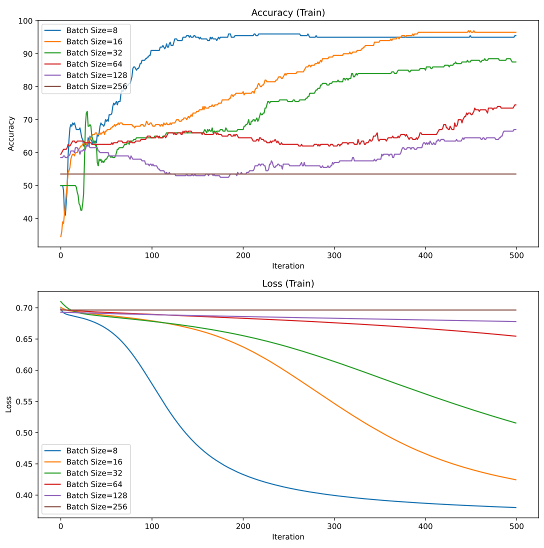
Figure 1.4: Results using `torch.nn`

1.2.4 Simplification of the SGD with `torch.optim`

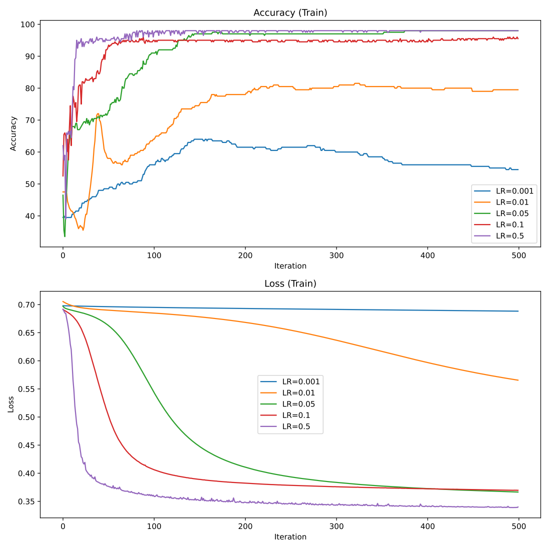
Training exhibits notably enhanced stability; however, it demands an extended period to converge, necessitating 500 epochs to achieve optimal performance.



(a) Acc/loss using `torch.optim`



(b) Influence of batch size

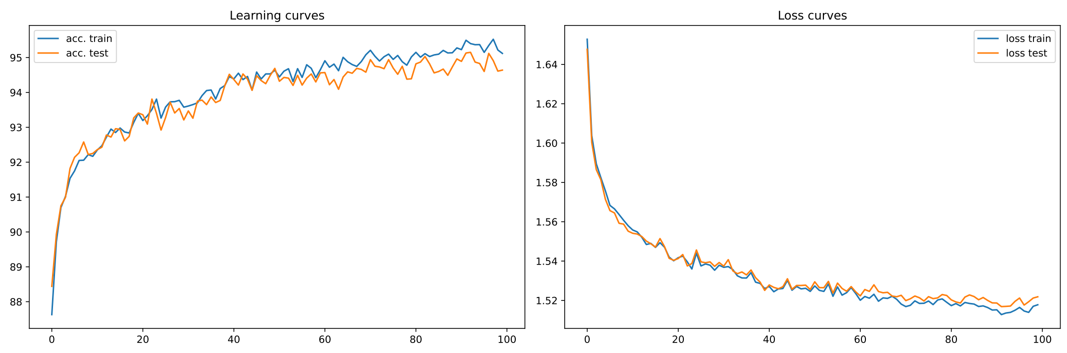


(c) Influence of learning rate

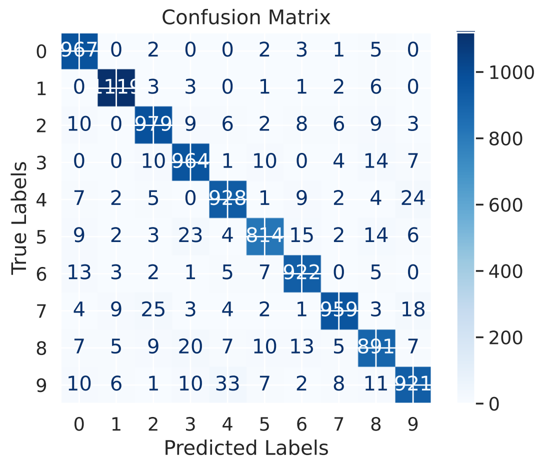
Figure 1.5: Results using `torch.optim`

1.2.5 Applying the neural network on MNIST

The results section showcases the efficacy of our straightforward model when applied to the classic MNIST dataset. Surprisingly, its performance is robust, managing to classify handwritten digits with high accuracy. However, we did encounter some oscillations in model training, indicating room for improvement. For instance, introducing a learning rate scheduler such as the StepLR or ExponentialLR could smooth out the training epochs. Alternatively, exploring optimizers beyond the standard SGD, like Adam or RMSprop, may offer a path to more consistent convergence and potentially enhance the model's efficiency in reaching its highest accuracy.



(a) Accuracy and losses curves of our model on MNIST



(b) Confusion Matrix

Figure 1.6: Performance on MNIST

Observations:

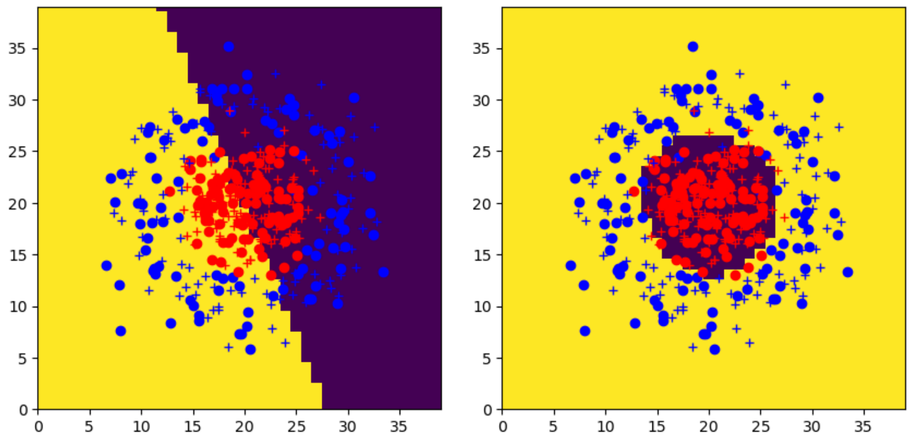
- **Accuracy:** Both the training and test accuracy curves plateau near the value of 1 (or 100%), indicating that the model performs very well on both datasets. This high accuracy suggests that the model is able to generalize from the training data to unseen data in the test set.

- **Loss:** Both the training and test loss curves settle near 0, which corresponds to the high accuracy observed.

- **Overfitting:** There is no evidence of overfitting, as the test loss decreases and test accuracy increases in tandem with the training loss and training accuracy, respectively.

- **Convergence:** The model seems to converge quickly and maintains stable performance from an early stage in the training process, as indicated by the flattening of both accuracy and loss curves. This could suggest that the model capacity is well-suited to the complexity of the task, the learning rate is appropriately set, and the optimization process is effective.

1.2.6 SVMs



(a) Linearm SVM (53% acc)   (b) SVM + RBF Kernel (94% acc)

Figure 1.7: Classification results using SVMs on the circle dataset

Observations:

- **Linear SVM (a):**
  - **Analysis:** The presence of misclassified points implies that the data is not linearly separable. The linear SVM model is likely experiencing both high bias and high variance in this context, suggesting underfitting: it is too simplistic to capture the complex structure of the data.
  - **Why this occurs:** Linear SVM assumes that the data is linearly separable, which is not the case here. Real-world data often exhibit complex relationships that linear models cannot capture.
- **SVM with RBF Kernel (b):**
  - **Analysis:** The SVM with the RBF kernel appears to have a good balance of bias and variance, fitting the training data well without apparent overfitting. This is evident by the reduction in misclassifications and the more intuitive grouping of data points by class.
  - **Why this occurs:** The RBF kernel transforms the feature space into a higher dimension where a linear decision boundary is possible, accounting for the non-linear patterns in the data. This flexibility allows for capturing more complex structures.
- **Influence of the C parameter:** The parameter C in SVM is the regularization parameter. It controls the trade-off between maximizing the margin and minimizing

the classification error. A smaller C leads to a larger margin but may allow some training examples to be misclassified (soft margin). It helps in avoiding overfitting. A larger C places more emphasis on classifying each training example correctly (hard margin) but may lead to a smaller margin. It can lead to overfitting if the data is noisy. The choice of the C parameter depends on the problem's nature and the level of noise in the data. We can use techniques like grid search or cross-validation to find the optimal C value for a specific problem. In practice, it's common to start with a small C and gradually increase it while monitoring the model's performance on a validation set to strike the right balance between margin size and classification accuracy.

Chapter 2

Convolutional neural networks CNNs

2.1 Introduction to convolutional networks (CNNs)

2.1.1 Answer 1. Convolution Output Size

For a 2D convolution, the spatial dimensions (width and height) of the output can be computed as:

$$W_{out} = \frac{W_{in} - k + 2p}{s} + 1, \quad H_{out} = \frac{H_{in} - k + 2p}{s} + 1$$

Given that the input size is $x \times y \times z$, and assuming that the number of output channels is n:

$$\text{Output size} = \left(\frac{x - k + 2p}{s} + 1\right) \times \left(\frac{y - k + 2p}{s} + 1\right) \times n$$

Weights in Convolution

For a single filter of size $k \times k$, the number of weights is $k \times k \times z$. Additionally, there's one bias term for each output channel. Therefore, total weights $= n \times (k \times k \times z + 1)$.

2.1.2 Answer 2. Advantages of Convolutional Layers

Parameter Efficiency

Convolutional layers use a shared-weight architecture (kernels or filters), which significantly reduces the number of parameters compared to a fully connected layer of similar size. This means they require less memory and are less prone to overfitting.

Exploitation of Spatial Hierarchies

Convolutional layers are designed to exploit the 2D structure of input data. They recognize patterns with respect to space (like edges, textures, and gradients) and build up complex patterns using simpler ones through the hierarchy of layers.

Translation Invariance

Once a feature is learned at one location in the image, a convolutional network can recognize it at a different location, thanks to the translational invariance imparted by the convolution operation and pooling layers. This is not the case with dense layers, where the network would need to learn the pattern anew if it appeared at a different location.

Local Connectivity

In a convolutional layer, each neuron is connected only to a local region in the input. This means the network can focus on local features before assembling them into a larger picture, which is a more efficient way of learning and reduces computational complexity.

Channel-wise Processing

Convolutional layers can process multiple input channels (e.g., RGB color channels) simultaneously, and they are able to learn features specific to combinations of channels.

Main Limitations of Convolutional Layers

- **Lack of Spatial Hierarchy** in Some Data Convolutional layers are primarily beneficial for data that has a clear spatial hierarchy, such as images. In data where spatial relationships are not as meaningful (like tabular data), the advantages of convolutional layers may not apply.
- **Invariance Only to Translation** While convolutional layers are invariant to translation, they are not inherently invariant to other transformations such as scaling and rotation. This can be partially mitigated with data augmentation and specialized network architectures.
- **Receptive Field Size** The receptive field of a convolutional layer (the size of the input area to which a filter is applied) is fixed, which can limit the layer's ability to capture features of varying sizes. This issue is typically addressed with layers of multiple scales or filters of different sizes.
- **Boundary Effects** Convolution operations can be sensitive to the boundaries of the input space, often requiring padding strategies to maintain dimensionality, which can introduce artifacts or reduce the network's performance at the image boundaries.
- **Computational Complexity** Despite being more efficient than fully connected layers, convolutional layers, especially in deep networks, can still be computationally intensive, requiring significant processing power and memory, particularly during training.

Conclusion

In conclusion, convolutional layers offer a structured, efficient way to learn features from spatially-correlated data like images, providing significant advantages in terms of parameter efficiency and the ability to capture spatial hierarchies in data. However, their design also brings specific limitations, which must be mitigated through careful network design and additional techniques.

2.1.3 Answer 3. Why do we use spatial pooling

Spatial pooling, also known as subsampling or down-sampling, is a crucial component in convolutional neural networks (CNNs). It serves multiple purposes, which contribute to the effectiveness and efficiency of CNNs in analyzing visual data. Here's a detailed explanation of the reasons for using spatial pooling in CNNs:

- **Dimensionality Reduction:** Spatial pooling helps reduce the dimensionality of each feature map in the network, decreasing the number of parameters and computations required in the layers that follow. This reduction in dimensionality not only curtails the computational cost but also the memory usage, making the network more efficient.

- **Overfitting Prevention:** By downsampling the feature maps, pooling layers help prevent overfitting. Overfitting occurs when a model learns the training data too well, including the noise and details that do not generalize to new data. Pooling achieves this by providing an abstracted form of the representation, allowing the network to generalize better from the input data.

- **Translation Invariance:** Pooling increases the translation invariance of the network. In practical terms, after pooling, the exact location of features becomes less important. This is beneficial for tasks like image classification where it's more important to know that a feature exists rather than its exact position. For example, whether a cat's ear is slightly higher or lower in an image, it should still be recognized as a cat.

- **Feature Aggregation:** Pooling combines semantically similar features into one. This process, which is sometimes referred to as feature pooling, helps the network in learning to recognize patterns that are more complex and abstract higher up in the network.

- **Noise Suppression:** It can suppress (or at least mitigate) the effects of noise and variations within the input data. By taking the maximum (in max pooling) or average (in average pooling), the pooling operation can smooth over minor variations and noisy activations in the feature maps.

- **Consistency and Robust Feature Detection:** Pooling helps the network to detect features in a way that is somewhat invariant to scale and orientation changes. When a feature like an edge or a texture pattern appears at different scales or orientations, pooling can help ensure that the network recognizes the feature as the same.

The most common types of spatial pooling are max pooling and average pooling:

- **Max Pooling:** Returns the maximum value from the portion of the image covered by the kernel. It is useful for when the presence of a particular feature is more important than its absence.

- **Average Pooling:** Calculates the average value of the portion of the image. It can be beneficial when all the values are important to provide a smooth, generalized feature map.

Despite the benefits, it's also important to recognize that pooling operations result in the loss of some information. For example, max pooling may retain the strongest feature in a local patch, but it discards all other information. This is a trade-off in CNN design where the benefits of pooling usually outweigh the disadvantages, especially for tasks such as image classification.

Recent developments in CNN architecture have explored alternative ways to reduce dimensionality and aggregate features without losing as much information as traditional pooling layers. These include strided convolutions, which perform subsampling directly within the convolutional operation, and architectures that eliminate pooling in favor of careful design of convolutional layers (e.g., "all-convolutional networks").

2.1.4 Answer 4. Using a ConvNet on a Larger Image

Convolutional layers can handle variable input sizes, but fully-connected layers expect a fixed-size input. For a larger image, convolutional layers can be applied, but the resulting spatial dimensions won't match the expected input size of the fully-connected layers. In classical convolutional neural networks (CNNs), the architecture is usually defined for a specific input image size. However, due to the nature of convolutional operations, it is indeed possible to apply a CNN to images of different sizes under certain conditions. Here's a detailed explanation:

- **Convolutional Layers:** The convolutional layers themselves are generally agnostic to the size of the input image. Convolutional filters slide across the input image regardless of its dimensions, applying the same operation. This means that for purely convolutional layers, we can apply the network to larger images without any issue. The network will produce a correspondingly larger feature map after each convolutional layer.

- **Pooling Layers:** Pooling layers also do not require a fixed-size input because they operate on local regions (e.g., 2x2 pixels), reducing the spatial size of the feature maps by a factor, regardless of the feature map's dimensions.

- **Fully Connected Layers:** Problems arise when the network contains fully connected (FC) layers, which expect a fixed-size input. FC layers are not spatially invariant; they require the same number of elements in the input feature vector. If the image is larger than expected, the feature maps resulting from the last convolutional or pooling layer will also be larger, and thus the total number of features going into the FC layer will be more than what the layer was designed to handle. In such cases, we cannot use the FC layers without modifying the architecture or the input.

2.1.5 Answer 5. Fully-Connected as Convolution

Fully connected layers and convolutional layers differ in their connectivity patterns, with fully connected layers connecting every neuron to all elements in the previous layer and convolutional layers connecting neurons to local regions of the input. Both processes give a way of summarizing the whole image with a set of numbers that can be used for further processing, like classification. The key difference is in how we think about arranging the weights and the computations, but conceptually, they can achieve very similar things. However, it's possible to view a fully connected layer as a specialized case of a convolutional layer through the following explanations:

- 1x1 Convolution: A 1x1 convolution is a particular type of convolution that can be seen as a fully connected layer applied to each element of the input feature map. When the input feature map is flattened into a single long vector, a 1x1 convolution operation is equivalent to a fully connected operation. This means that a fully connected layer can be conceptually transformed into a 1x1 convolution layer.

- Fully Connected Layer: In a fully connected layer, every neuron considers all the pixels in the input simultaneously, with each pixel having its own weight. This results in a list or vector of output values.

- 1x1 Convolution: A 1x1 convolution, when applied with a stride that covers the entire image and a filter size equal to the size of the image, also simultaneously considers all pixels in the input, assigning one weight per pixel, and produces a list of output values.

In summary, while fully connected layers and convolutional layers have different connection patterns, a fully connected layer can be conceptually mapped to a 1x1 convolutional layer, which performs a similar operation by considering all input elements with one weight per element. This insight demonstrates a connection between these two layer types in deep learning architectures.

2.1.6 Answer 6. Replace Fully-Connected with Convolutions

Replacing Fully-Connected Layers with 1x1 Convolutions: When we replace fully-connected layers with equivalent convolutions, specifically using 1x1 convolutions, we effectively remove the dependency on a fixed input size. This is because convolutions are applied locally and slide across the input space, generating an output feature map that reflects the size of the input.
Handling Larger Input Images: For an input image that is larger than what the network was initially designed for:

- The convolutional layers (including the new convolutional layers replacing fully-connected ones) will process the larger input as they normally do, outputting feature maps that are proportionally larger than what would be expected with the original input size.
- The output shape of the network will thus be larger, reflecting the larger input size.

The interest or advantage of this approach is that the network becomes fully convolutional and can handle inputs of variable sizes. The network can process larger images and potentially capture more detailed information that might be present in these larger inputs.
Variable Output and Dense Prediction: The output in such cases will not be a fixed-size vector but a set of feature maps that have spatial dimensions. These can be interpreted as a dense prediction across the input space, which is useful in tasks like semantic segmentation where we want to make a prediction for every pixel or region in the input image.

2.1.7 Answer 7. Receptive Field.

- First Layer: The receptive field is the size of the kernel itself, $k \times k$.

- Second Layer: For a stride of 1, the receptive field becomes $2k - 1$.
- Deeper Layers: The receptive field grows exponentially with depth, capturing larger parts of the input image.
- Interpretation: Deeper layers capture more global, higher-level features, while shallower layers focus on local, low-level features.

2.2 Training from scratch of the model

Assuming CIFAR-10 images are 32x32x3 in size:

2.2.1 Answer 8. Convolution Padding and Stride

To maintain the same spatial dimensions for a 5x5 convolution:

$$\text{Padding (p)} = 2, \quad \text{Stride (s)} = 1$$

2.2.2 Answer 9. Max-Pooling Padding and Stride

To reduce spatial dimensions by a factor of 2 using 2x2 pooling:

$$\text{Padding (p)} = 0, \quad \text{Stride (s)} = 2$$

2.2.3 Answer 10. Layer-by-Layer Output Size and Weights

2.3.1 conv1

- Output size = 32x32x32 (due to 32 filters)
- Weights = 5x5x3x32 = 2400

2.3.2 pool1

- Output size = 16x16x32
- Weights = 0 (pooling has no learnable parameters)

2.3.3 conv2

- Output size = 16x16x64
- Weights = 5x5x32x64 = 51200

2.3.4 pool2

- Output size = 8x8x64
- Weights = 0

2.3.5 conv3

- Output size = 8x8x64
- Weights = 5x5x64x64 = 102400

2.3.6 pool3

- Output size = 4x4x64
- Weights = 0

2.3.7 fc4

- Output size = 1000
- Weights = 4x4x64x1000 = 1024000

2.3.8 fc5

- Output size = 10
- Weights = 1000x10 = 10000

2.2.4 Answer 11. Total number of weights to learn

Total weights = 2400 + 51200 + 102400 + 1024000 + 10000 = 1168000

Weights per Example

CIFAR-10 has 50,000 training examples, so on average, there are roughly 23 weights per example.

2.2.5 Answer 12. Comparison with Bag-of-Words (BoW) and SVM Approach

The Bag-of-Words (BoW) approach depends on the vocabulary size (V). If we have a vocabulary of size V and we feed this into an SVM, the number of parameters will be roughly V (one per word). This can grow large if V is big, but typically would be less than the number of parameters in a deep CNN like the one described.
The comparison here is qualitative: BoW with SVM has a fixed-size input (the vocabulary size) and is often more interpretable, but CNNs have the capacity to learn more complex, hierarchical representations from raw pixel values.

Answer 13.

We test the given code with the following set of parameters: batch size = 128, learning rate = 0.01, epochs = 32

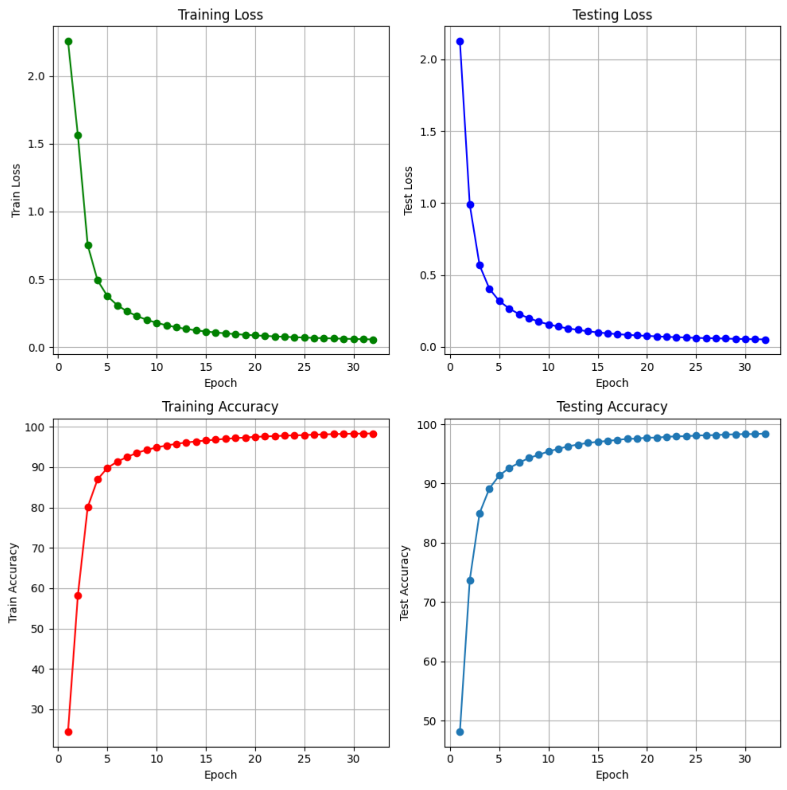


Figure 2.1: Evolution of loss and accuracy on train/test - MNIST dataset

2.2.6 Answer 14.

In a machine learning model's training and evaluation process, loss and accuracy are computed differently during training and testing phases, not only because they are based on distinct sets of data but also due to the difference in operations performed.
During the training phase, the model iteratively learns from the training dataset by performing forward passes to compute the loss, followed by backward passes to adjust the model's weights via an optimization algorithm. This dynamic adjustment aims to minimize the average loss over batches, which is biased since the model's weights are continuously updated throughout the process.
In contrast, the testing phase evaluates the model's learned patterns by computing loss and accuracy over the entire test dataset, providing an unbiased assessment of the model's generalization capabilities. This evaluation is performed without weight adjustments, ensuring that the model's performance reflects its true predictive power on data it has not encountered during training.

2.2.7 Answer 15.

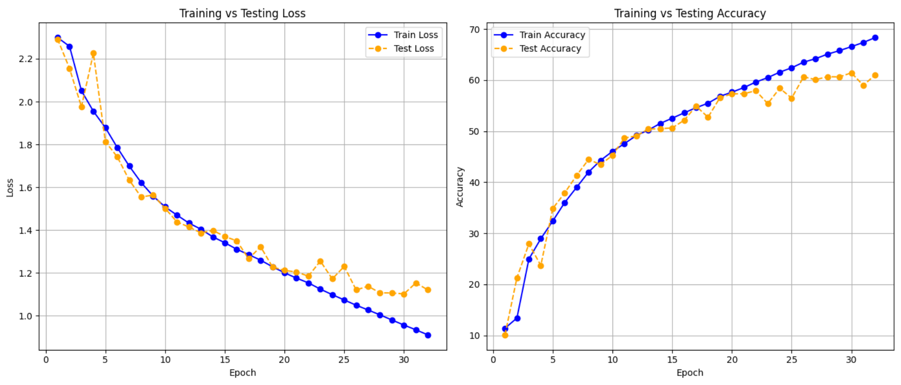


Figure 2.2: Evolution of loss and accuracy on train/test - Cifar10 dataset

2.2.8 Answer 16.

Learning Rate effects:

- Convergence: A proper learning rate helps the model to converge to a minimum. A rate too high can cause the model to overshoot the minimum, while a rate too low might result in a very slow convergence, possibly getting stuck in a local minimum.
- Training Speed: Higher learning rates can accelerate the training as they take larger steps during weight updates. However, if it is too high, it might prevent the model from converging by causing the loss to fluctuate or even diverge.
- Stability: Lower learning rates often lead to more stable training, with the loss decreasing smoothly. A higher learning rate can cause unstable training with a fluctuating loss.
- Generalization: Some research suggests that smaller learning rates might lead to better generalization in some cases. However, this is still a subject of ongoing research and debate.

Batch Size effects

- Memory Constraints: Larger batch sizes require more memory. As a result, the maximum batch size might be limited by the hardware's capabilities.
- Stochasticity: Smaller batch sizes introduce more noise in the gradient estimation, which can help escape local minima but also can make the training process less stable.

- Convergence Quality: Larger batches provide a more accurate estimate of the gradient but might result in less robust convergence properties as they may get stuck in sharper minima, which are less generalizable. Small batch sizes often lead to convergence to flatter minima, which might generalize better.
- Training Speed: Larger batch sizes can make each epoch faster because of more efficient matrix operations in parallelized computations. However, since the gradient estimate is more accurate, it may require fewer epochs to reach convergence.
- Learning Dynamics: There is evidence that the batch size can affect the learning dynamics. Large batch sizes might smooth out the optimization landscape, while small batch sizes can exploit the curvature of the landscape for faster convergence.

In practice, finding the right combination of learning rate and batch size requires empirical testing and can be dependent on the model architecture, the nature of the dataset, and the computational resources available. Techniques like learning rate schedules (gradually decreasing the learning rate during training) and adaptive learning rate algorithms (like Adam, RMSprop) are often used to help mitigate some of the sensitivities related to the choice of learning rate. Similarly, techniques like gradient accumulation can be used to simulate larger batch sizes on hardware with limited memory.

Experimental study of the effect of the learning rate and batch size on the model:

In this study we aim to perform some experiments in order to evaluate the effects of the different values that can take the learning rate and batch size on the overal performance of our CNN.

We fix the batch size = 128 and iterate over multiple values of the learning rate and batch size, and observe the performance on the test set:

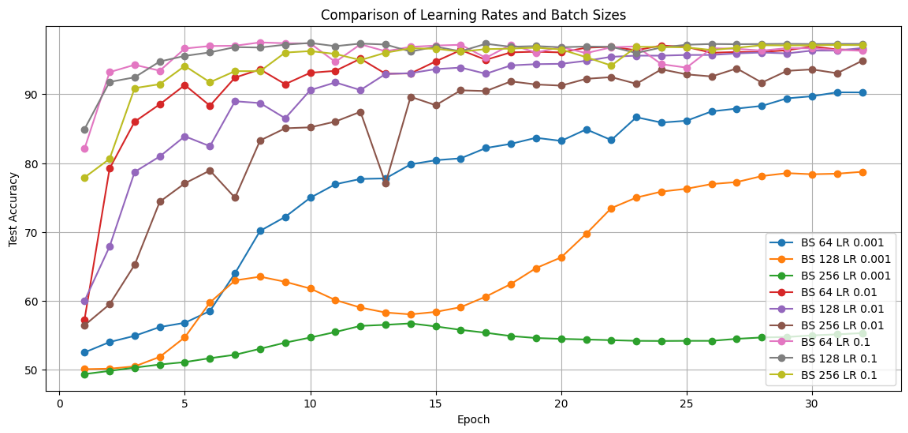


Figure 2.3: Influence of batch size and learning rate on the performance

2.2.9 Answer 17.

For the following experiments, we will compare our results with those presented in figure 2.5, where a batch size of 128 and a learning rate of 0.01 were employed (For 100 epochs).

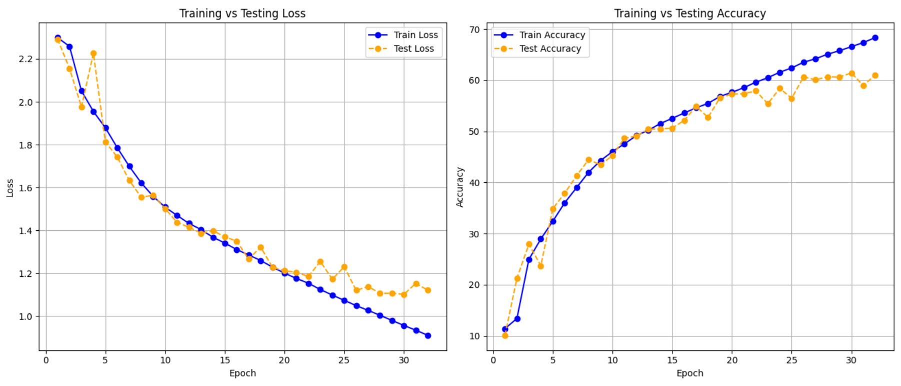


Figure 2.4: Evolution of loss and accuracy on train/test set for 32 epochs

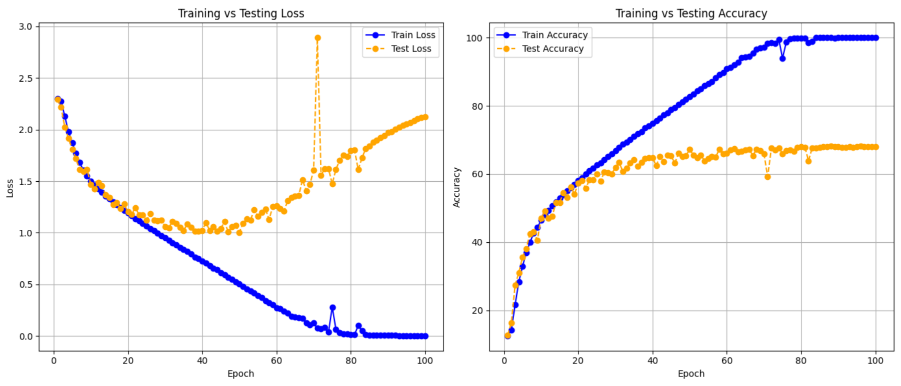


Figure 2.5: Evolution of loss and accuracy on train/test set for 100 epochs

Interpretation :

Train Error (2.2993): This metric indicates the model's performance on the training dataset. Initially, when training begins, the model is likely making predictions with probabilities close to 1/nbClasses (where nbClasses is the number of classes), essentially suggesting that no meaningful learning has occurred yet.

Test Error (2.2944): This metric reflects the model's performance on the test dataset, which it hasn't encountered during training. At the early stages of training, the test error being slightly lower than the train error isn't particularly meaningful. This difference can be attributed to the initial set of weights and biases assigned to the model and the specific samples present in the test set.

The critical point to understand is that these errors are high initially because the model is in a state of maximum uncertainty; it's essentially making educated guesses. As the training process advances, we would expect these errors to decrease, reflecting the model's increasing ability to make informed predictions based on the features it learns during training.

2.2.10 Answer 18.

We observe Overfitting in the training progression when we notice a divergence between the training and testing performance. By the thirtieth epoch, we see that the training accuracy continues to ascend, reflecting an improved ability of the model to predict outcomes on the training dataset.

However, the testing accuracy plateaus around the 0.6 mark, indicating a stagnation in the model's predictive power on the testing dataset. This discrepancy suggests that while the model has become adept at recognizing patterns within the training data, it is not extending this predictive capability to the testing data, which is unseen during training.

This overfitting implies that the model is likely memorizing the training data rather than learning generalizable features that apply to both the training and testing sets. The goal in such a scenario is to improve the model's generalization by possibly simplifying the model, employing regularization techniques, or augmenting the training data to encompass a broader range of examples.

2.3 Results improvements

2.3.1 Standardization of examples

Answer 19.

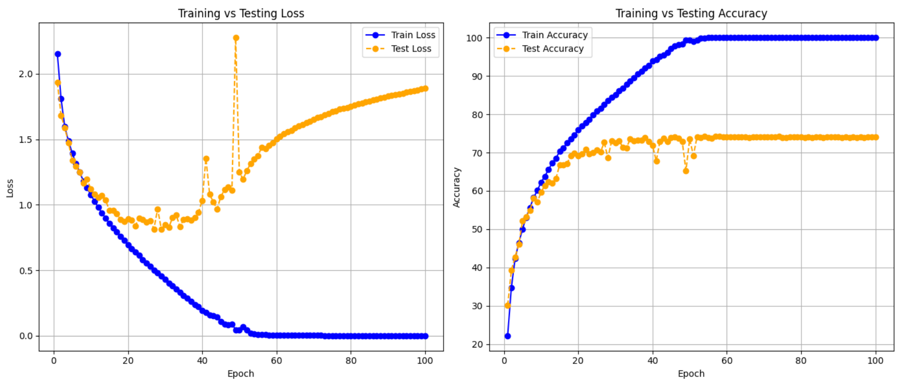


Figure 2.6: Evolution of accuracy and loss on train/test using standardization

Standardization plays a crucial role in preparing data for model training by addressing three main challenges:

- Discrepancies in Data Scales: Raw data can come from different sources and scales, leading to significant variance in the values. Standardization normalizes these values, bringing them to a comparable scale.
- Uniform Data Distribution: By standardizing, the data is transformed to fit a consistent distribution, which is often assumed by many machine learning algorithms to perform optimally.
- Numerical Instability: Different scales can lead to numerical instability during the optimization process, causing difficulties in model convergence and potentially leading to poorer performance.

In our experiments, we observed the following benefits from standardizing the data:

- Efficiency in Training: The model reached convergence in the training phase with 20 fewer epochs compared to without standardization, indicating a more efficient learning process.
- Enhanced Stability: There was a noticeable improvement in the stability of both the loss and accuracy metrics, suggesting that standardization helped the model to learn and generalize better.

In summary, standardization is a critical step that aids in harmonizing the input data, thereby facilitating a smoother and more reliable training process.

Answer 20.

Using the average image from the training set to normalize both training and validation data avoids data leakage, ensures consistent preprocessing, simulates real-world conditions, and utilizes the largest, most representative data sample for calculating the normalization parameters. This approach is standard practice to ensure that the model generalizes well to new, unseen data.

Answer 21. Impact of different standardization techniques

In our analysis, we conducted a comparison of various techniques, and the findings are presented in Figure 13. It is evident from the results that the minmax technique exhibits a slower convergence rate compared to other methods. On the contrary, techniques such as zca and standardization demonstrate significantly better performance in terms of convergence speed.

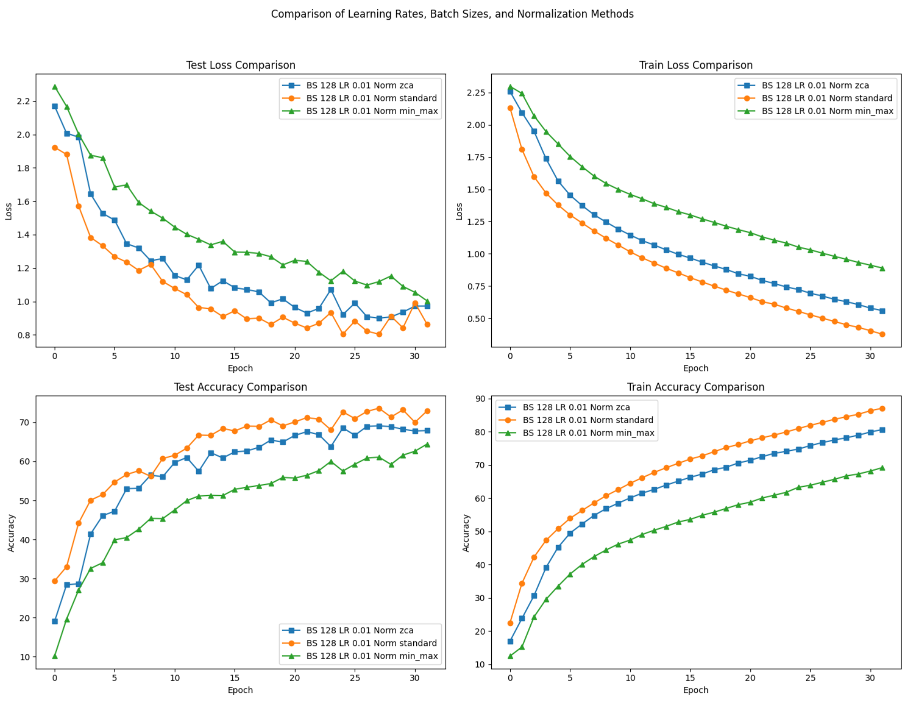


Figure 2.7: Impact of the standardization technique on network's performance

2.3.2 Increase in the number of training examples by data increase

Answer 22. Description of the experimental results and compare them to previous results.

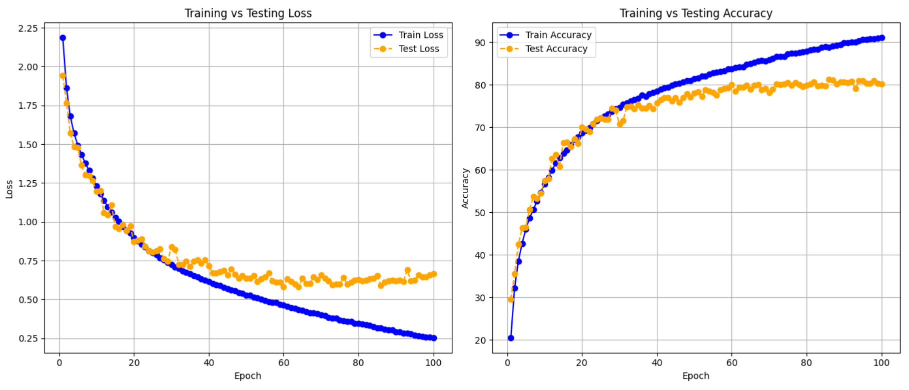


Figure 2.8: Evolution of loss and accuracy on train/test set using data augmentation

Our experimental results indicates a significant improvement in the model's generalization capabilities. This is evident from the reduced discrepancy between the performance on the training set and the validation or test sets. Previously, the model exhibited a tendency to overfit, as demonstrated by high accuracy on the training data but substantially lower accuracy on unseen data. Now, the model maintains robustness when exposed to new, unencountered examples.

Answer 23. The horizontal symmetry approach

The horizontal symmetry approach used as a data augmentation technique, may not be universally applicable or beneficial for all types of images or tasks. Here are some scenarios where it can be either beneficial or not:

When Horizontal Symmetry Can Be Beneficial:

- Generic Object Recognition: For tasks like generic object recognition where the orientation of objects can vary widely (e.g., recognizing cats and dogs), horizontal flipping can be useful to make the model invariant to such changes.
- Balanced Perspective: In scenarios where images can be taken from different orientations, such as satellite imagery or landscape photos, horizontal symmetry can help in learning features that are orientation-agnostic.
- Robustness to Orientation: When training on datasets that may not have a uniform distribution of image orientations, flipping can artificially balance the dataset, potentially making the model more robust.

When it May Not Be Beneficial:

- Text and Numbers: For images containing text or numbers, horizontal flipping would render the text unreadable and the numbers incorrect, which could severely degrade performance for tasks like optical character recognition (OCR).
- Contextual Directionality: In images where direction is important, such as street signs or vehicle orientations in traffic images, flipping could lead to incorrect interpretations.

In summary, whether to use horizontal symmetry as an augmentation technique should be decided based on the specifics of the task, the dataset, and the desired invariances of the model. Careful consideration is needed to ensure that augmentation does not introduce misleading information into the training process.

Answer 24. Limits in this type of data increase by transformation of the dataset

- **Realism:** Some transformations may create non-realistic samples, potentially confusing the model.
- **Distribution Shift:** Augmented data might not accurately represent the true data distribution.
- **Overfitting:** Models may overfit to features specific to augmented data, harming real-world performance.

- **Computational Cost:** Data augmentation increases computational demands due to larger dataset sizes and processing needs.
- **Bias:** Overuse of certain transformations can introduce bias, impacting model fairness and reliability.
- **Context Loss:** Transformations such as excessive cropping or flipping may remove critical contextual information.
- **Annotation Correction:** Augmented images require corresponding updates in annotations, which can be complex.
- **Diminishing Returns:** Beyond a certain point, additional augmented data may not yield performance improvements.
- **Transformation Relevance:** Not all transformations contribute meaningfully to model training and may even be counterproductive.
- **Domain-Specific Concerns:** Certain domains, like medical imaging, may be sensitive to inappropriate transformations.

Answer 25. Data augmentation methods

There are several other data augmentation methods beyond rotations. The most common ones are:

- **Cropping:** Extracting subparts of the image to create variations.
- **Scaling:** Changing the size of the image.
- **Translation:** Shifting the image along the X or Y axis.
- **Flipping:** Mirroring the image horizontally or vertically.
- **Rotation:** Rotating the image by a certain angle.
- **Noise Injection:** Adding random noise to the image.
- **Brightness Adjustment:** Varying the brightness of the image.
- **Contrast Adjustment:** Modifying the contrast of the image.
- **Saturation Adjustment:** Altering the saturation levels in the image.

We here choose to test RandomHorizontalFlip technique. Results are displayed in the figure below :

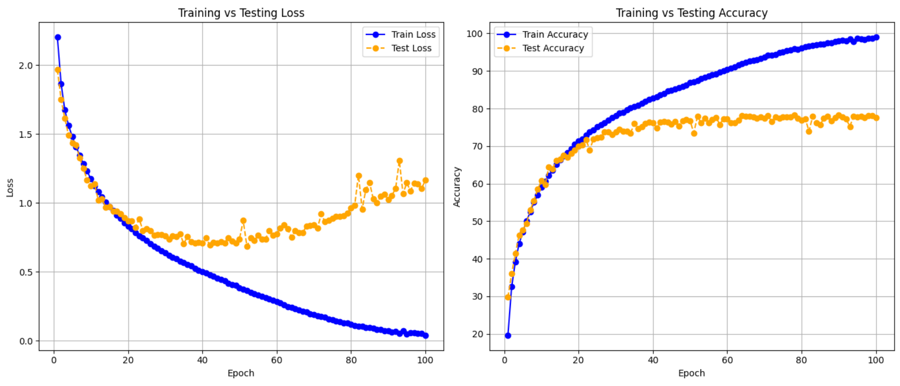


Figure 2.9: Evolution of loss and accuracy on train/test set using flip augmentation technique

2.3.3 Variants on the optimization algorithm

Answer 26. Description of experimental results and comparing them to previous results, including learning stability

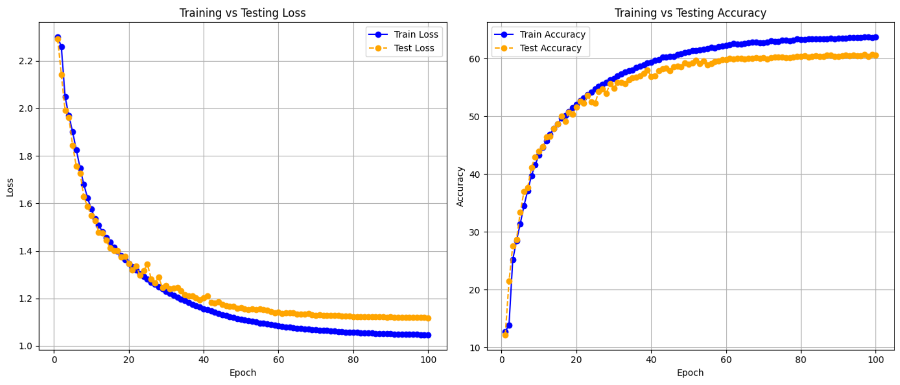


Figure 2.10: Evolution of loss and accuracy on train/test set using an exponential decay scheduler with SGD

Utilizing learning rate schedulers introduces a dynamic component to the training process that can significantly improve the stability and efficiency of the learning. In contrast to the fixed, potentially suboptimal approach of constant learning rates, schedulers offer a tailored learning experience that adapts to the model's needs over time.

The adaptability of learning rate schedulers usually results in faster convergence, avoids long periods of stagnation in plateaus, contributes to training stability, potentially finds better local minima, and can result in a more efficient training process overall.

Answer 27. Learning improvment

Key Advantages of Learning Rate Schedulers:

- Controlled Model Convergence: A scheduler enables the use of a higher learning rate at the beginning for rapid progress and a lower rate later to fine-tune model weights, thus aiding in efficient convergence.
- Avoiding Plateaus: Learning rate schedulers can modify the learning rate to escape or quickly move past plateaus in the loss landscape, ensuring continuous learning progress.
- Enhanced Training Stability: Gradually decreasing the learning rate can stabilize training by preventing drastic weight updates that could derail the learning process.
- Reaching Better Minima: A well-designed learning rate schedule can help in steering the optimization process towards more favorable local minima or even the global minimum.
- Efficient Use of Training Time: By dynamically adjusting the learning rate, schedulers can help reduce the time required to train a model by making each epoch more effective.

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

Types of Learning Rate Schedulers:

- **Step Decay:** Lowering the learning rate at specific epochs.
- **Exponential Decay:** Continuously decreasing the learning rate by a constant factor.
- **Cosine Annealing:** Adjusting the learning rate in a cosine waveform, simulating restarts to find better minima.
- **ReduceLROnPlateau:** Diminishing the learning rate when progress stalls, typically monitored by validation loss.
- **Cyclic Learning Rates:** Varying the learning rate between two boundaries to encourage exploration of the loss landscape.

Choosing the right scheduler depends on the specific model and problem. By experimenting with different scheduling strategies, one can find the most effective approach for the task at hand.

The Polynomial scheduler demonstrates the swiftest convergence, although it achieves relatively poor performance, reaching approximately 45 accuracy on both the training and

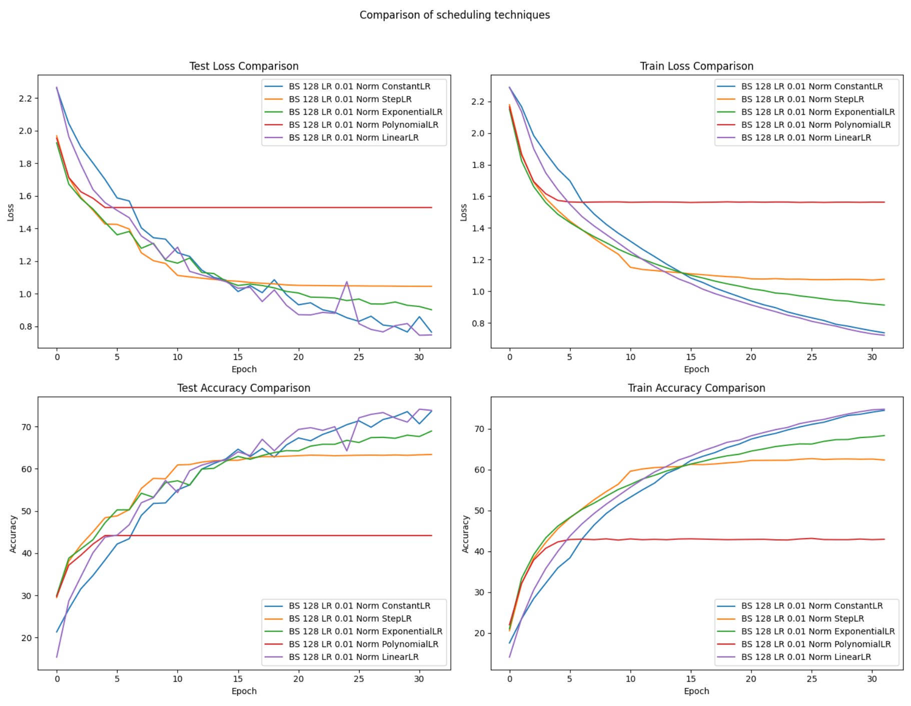


Figure 2.11: Evolution of loss and accuracy on train/test using multiple lr planning strategies

test set. Among the schedulers, the top performers are Exponential and Step and linear, all displaying analogous behaviors.

2.3.4 Regularization of the network by dropout

Answer 29.

The experimental results with dropout regularization show a notable improvement compared to previous results. Dropout effectively reduces overfitting, resulting in a model that generalizes better to unseen data. This leads to improved performance on both the training and test datasets, indicating that dropout has successfully mitigated overfitting issues observed in the previous experiments.

Answer 30. Regularization in general

Regularization is a fundamental technique in machine learning that aims to reduce overfitting, thereby improving the model's ability to generalize to new data.

It works by adding a penalty to the model's complexity, which discourages the learning algorithm from fitting the noise in the training data.

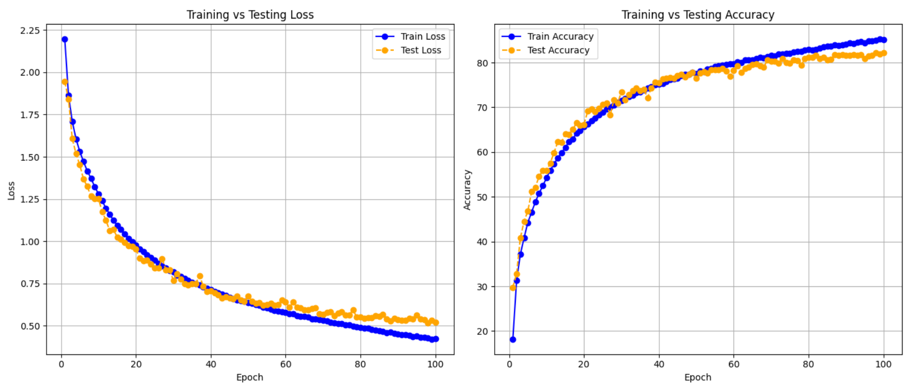


Figure 2.12: Evolution of loss and accuracy on train/test set using dropout

This penalty comes in different forms: L1 regularization encourages sparsity in the model coefficients, potentially setting some to zero for feature selection; L2 regularization encourages smaller, spread-out values among the coefficients, preventing any single feature from having too much weight; and techniques like dropout in neural networks randomly deactivate certain pathways in the network during training to promote redundancy and reduce sensitivity to specific training data patterns. Regularization parameters are typically tuned to balance the trade-off between fitting the training data well and keeping the model simple enough to perform well on unseen data.

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

Dropout is a widely used regularization technique in deep learning that can significantly improve the performance of neural networks by reducing overfitting. Here are some interpretations of how dropout affects the behavior of a network:

- **Prevents Co-adaptation of Features:** Dropout randomly drops neurons during training, encouraging the network to learn robust features that are useful in conjunction with various subsets of neurons.

- **Ensemble Interpretation:** Dropout can be viewed as training an ensemble of networks with shared weights. During training, each mini-batch trains a different model with a subset of neurons. At test time, it averages predictions from these thinned networks, enhancing generalization.

- **Noise Injection:** Dropout introduces noise during training, preventing overfitting by encouraging the network to learn general data patterns rather than relying on exact neuron inputs.

- **Information Bottleneck:** Dropout creates an information bottleneck by dropping different sets of neurons, promoting compact and relevant information flow through the network, leading to more effective data compression.
- **Feature Ranking:** Dropout assigns importance scores to input features, aiding in feature selection and model interpretability.
- **Regularization Effect:** Dropout acts as dynamic and non-linear regularization, different from traditional L2 regularization, fostering a wider range of network paths and feature diversity.
- **Training vs. Inference Behavior:** During training, dropout ensures that neuron activation isn't guaranteed on every pass, preventing reliance on a single path. In inference, all neurons are active, effectively averaging the predictions of thinned networks for a stable and reliable output.

Answer 32. Influence of the hyperparameter of this layer

Influence of Dropout Rate (p) in Dropout Layers:

- **Dropout Strength:** The dropout rate "p" determines the fraction of neurons dropped out during each training step. Higher values (e.g., 0.5) result in stronger dropout, effectively preventing overfitting but potentially slowing training.
- **Regularization Strength:** Dropout is a form of regularization. Higher "p" values provide stronger regularization, while lower values allow the model to fit the training data more closely.
- **Trade-off:** Selecting "p" involves a trade-off. Too high values may hinder learning, while too low values may not offer sufficient regularization.
- **Empirical Tuning:** "p" is often chosen empirically. Different values (e.g., 0.2, 0.5, 0.7) are tested, and the one leading to the best validation performance is selected.

The dropout rate "p" directly influences dropout's regularization strength and its impact on overfitting. Experimentation is essential to determine the optimal "p" for a given model and dataset.

Answer 33.

The dropout layer behaves differently during training and testing phases to ensure that it serves its purpose as a regularizer during training while allowing for a stable output during testing.

During Training:
Randomly 'drops' or 'deactivates' a subset of neurons in the layer where dropout is applied, with a certain probability (for example, p=0.5 would deactivate each neuron with a 50The neurons that are not dropped out have their outputs scaled up by 1/(1-p) to compensate for the reduced number of active neurons. This ensures that the expected sum of the outputs remains roughly the same as it would be without dropout. This random omission of neurons during training prevents overfitting by ensuring that the network cannot rely on any individual neuron and must instead learn more robust features that are distributed across multiple paths in the network.
During Testing:
All neurons are active; none are dropped. No scaling is applied to the neuron outputs because we're no longer randomly deactivating neurons. This is equivalent to using the expected value of the neuron's output (considering the dropout probability during training). The rationale is that during testing, we want to use the full strength of the network to make accurate predictions. Dropping neurons randomly would introduce unnecessary noise into the predictions.
The idea is to train a larger, sparser network and then to use a denser, more reliable network at test time. Dropout is a form of model averaging, where training different 'thinned' networks corresponds to training different sub-models. At test time, using all the neurons is akin to averaging the predictions of these sub-models.

2.3.5 Use of batch normalization

Answer 34.

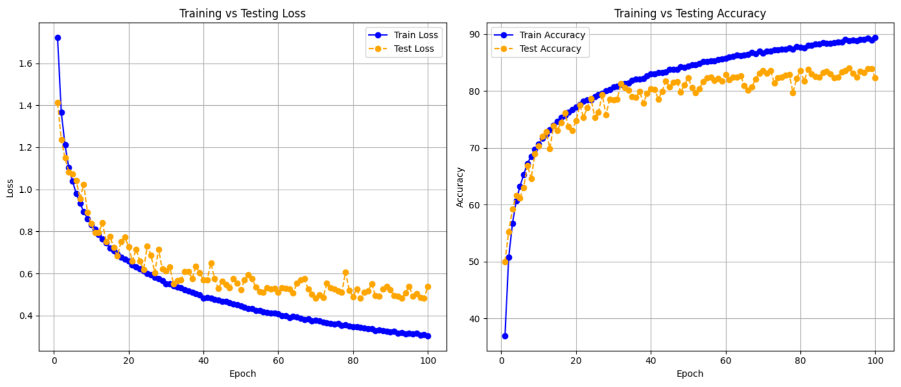


Figure 2.13: Evolution of loss and accuracy on train/test set using batch normalization

Batch normalization (BatchNorm) enhances accuracy by normalizing layer activations, stabilizing gradients, and mitigating overfitting. It leads to faster convergence, better training dynamics, and improved generalization, resulting in increased accuracy in neural networks

Chapter 3

Transformers

3.1 General questions about Transformers

3.1.1 Single head self-attention

Q: What is the main feature of self-attention, especially compared to its convolutional counterpart?
A: Self-attention is a mechanism that allows each element in the input to focus on different parts of the input, thereby capturing dependencies regardless of their positions in the input sequence. Unlike convolutional layers which have a local receptive field and are translation-invariant, self-attention can capture long-range dependencies by computing interactions between all pairs of positions in the input sequence.

Q: What is its main challenge in terms of computation/memory?
A: The primary challenge with self-attention in terms of computation and memory arises from the computation of the attention matrix, which is of size $N\ddot{O}N$ (where N is the sequence length). This leads to a time and space complexity of $O(N^2)$, which can be prohibitive for long sequences.

Q: Write the equations in the simple case of a single head self-attention.
A: Here is the simplified self-attention mechanism (for a single head) broken down into steps:

1. Compute Query (Q), Key (K), and Value (V) representations: $Q = XW_Q$, $K = XW_K$, $V = XW_V$
2. Compute attention scores: $scores = \frac{QK^T}{\sqrt{d_k}}$
3. Apply Softmax: $attention = softmax(scores)$
4. Compute output: $output = attention.V$
5. Final linear projection: $final\ output = output \cdot W_O$

3.1.2 Multi-head self-attention

Q: Write the equations describing multi-heads self-attention mechanism.
A: To avoid redundancy and streamline the description, the multi-head self-attention

mechanism will be incorporated within the complete transformer encoder block process, providing a more concise and coherent overview.

3.1.3 Transformer Block (Encoder)

Q: Write the equations describing the full transformer encoder block in a ViT model.

A: Here is a full breakdown of what happens inside the encoder block as shown in the reference image:

1. **Embeddings:** The input to the Transformer Encoder block is a sequence of embedded patches, $X \in \mathbb{R}^{N\times d}$, where N represents the number of patches and d is the dimensionality of the embeddings.

2. **Norm 1:** The first normalization step is applied to the input embeddings X:

$$X_{\text{norm}} = \text{LN}(X) \tag{3.1}$$

   where LN denotes the layer normalization operation:

$$\text{LN}(x_i) = \gamma \left(\frac{x_i - \mu}{\sigma} \right) + \beta \tag{3.2}$$

   Here, μ and σ are the mean and standard deviation of the inputs, and γ and β are learnable parameters for scaling and shifting, respectively.

3. **Multi-Head Attention (MHA):** The normalized input is then processed by the Multi-Head Attention mechanism, which includes a residual connection:

$$Z = \text{MHA}(X_{\text{norm}}) + X \tag{3.3}$$

   where MHA is defined as:

$$\text{MHA}(X_{\text{norm}}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{3.4}$$

   Each head head_i for $i \in [1, h]$ is computed as:

$$\text{head}_i = \text{Attention}(X_{\text{norm}}W_i^Q, X_{\text{norm}}W_i^K, X_{\text{norm}}W_i^V) \tag{3.5}$$

   with the Attention function defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} \right) V \tag{3.6}$$

In this context, W_i^Q, W_i^K, W_i^V are learnable weight matrices for queries, keys, and values, respectively; W^O is the output weight matrix; and $d_k = \frac{d}{h}$ is the dimension of the key vectors in each head.

4. **Norm 2:** Apply normalization to the output of the MHA step:

$$Z_{\text{norm}} = \text{LN}(Z) \tag{3.7}$$

5. **MLP:** Finally, the normalized output from the MHA block is fed into a Multi-Layer Perceptron (MLP) with a GeLU activation function, followed by a residual connection:

$$\text{Output}_{\text{encoder}} = \text{MLP}(Z_{\text{norm}}) + Z \tag{3.8}$$

The MLP with one hidden layer is defined as:

$$\text{MLP}(x) = \text{GeLU}(xW_1 + b_1)W_2 + b_2 \tag{3.9}$$

where W_1, W_2 are learnable weight matrices, and b_1, b_2 are bias vectors. The GeLU activation is given by:

$$\text{GeLU}(x) = x \cdot \Phi(x) \tag{3.10}$$

where $\Phi(x)$ is the standard Gaussian cumulative distribution function.

3.1.4 Full ViT Model

- **Class Token:** In the Vision Transformer (ViT), a special token known as the Class Token is prepended to the sequence of flattened image patches. This token is used as a learnable representation for the entire image, and its final representation at the end of the transformer encoder is used for classification. The Class Token allows the model to have a global representation which aggregates information across the entire image, facilitating the final classification task.

- **Positional Embedding (PE):** The transformer architecture is permutation-invariant, meaning it doesn't have an inherent understanding of the order of tokens in a sequence. However, in image processing, the relative positions of patches are crucial for understanding the spatial structure of the image. Positional embeddings are added to the patch embeddings to provide the model with information about the spatial location of each patch. They are usually implemented as a fixed or learnable addition to the patch embeddings. By incorporating positional information, the model can learn spatial hierarchies and relationships among patches, which is important for capturing the 2D structure of the image.

3.2 Experiments

To investigate the impacts of different hyperparameters (embed_dim, nb_blocks, patch_size), we chose to analyze each hyperparameter individually while keeping the others constant. This decision was driven by the impracticality of conducting an exhaustive grid search, as well as the interdependent nature of these parameters. By studying them separately, we aim to provide a clearer and more straightforward understanding of how each parameter influences the model's performance in terms of training time, as well as train and test loss/accuracy.

3.2.1 Embedding Dimension (embed_dim)

In this section, we experiment with varying the embedding dimension while keeping the other parameters constant at the following values: **nb_blocks = 2, patch_size = 7, and nb_heads = 4.**

- **Dim 32 and 64 (Blue and Orange):** The lower capacity models converge to a less optimal solution, as indicated by higher training and test loss, and lower accuracy. This suggests they may be underfitting and not complex enough to capture the patterns in the data adequately.
- **Dim 128 (Green):** This seems to be the sweet spot for this dataset and training configuration, providing a good trade-off between learning capacity and generalization ability. The learning curves are relatively smooth and consistently improving.
- **Dim 256 (red) and 512 (purple):** Although a larger embedding dimension typically enables the capture of more intricate features within the image patches, these specific values serve as evident instances of overfitting. Notably, both the training and testing outcomes do not exhibit commensurate improvements, with Dimension 512 demonstrating even more pronounced variability. This unmistakably indicates that these models possess excessive capacity, leading to the generation of noise in the embeddings.
- **Complexity and Training Duration:** Contrary to intuition, it appears that smaller embedding dimensions lead to an extended training and evaluation time for the model on the test set. This phenomenon may be attributed to the following explanation:

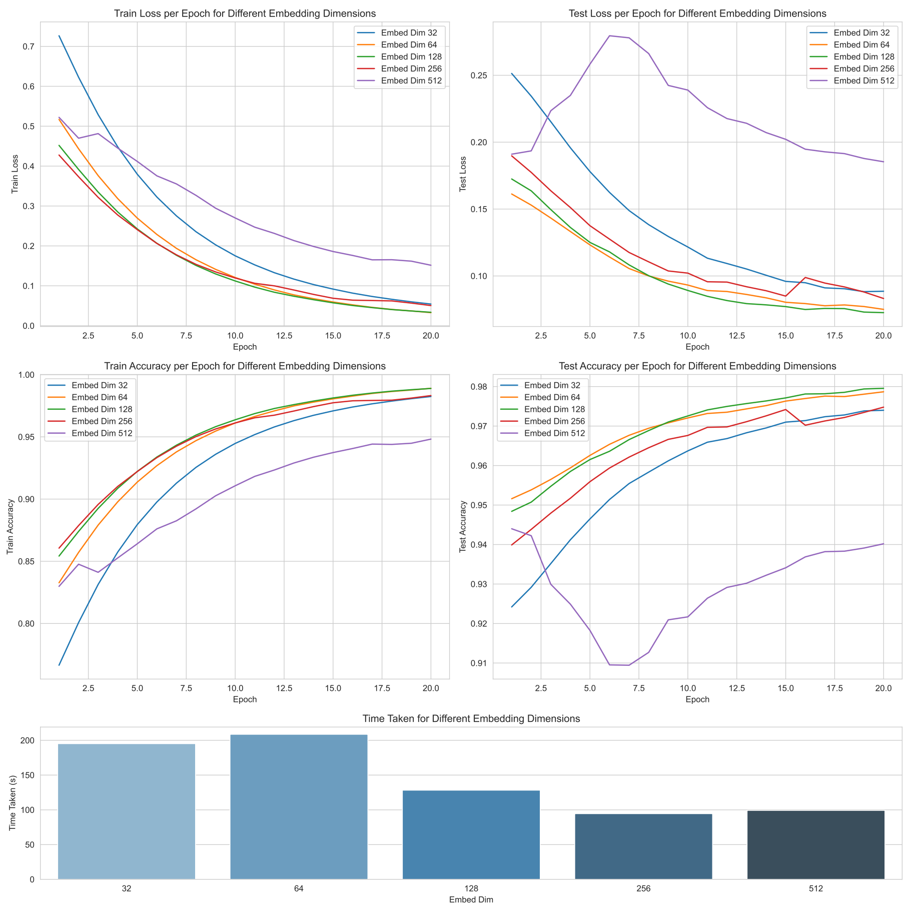


Figure 3.1: Influence of embedding dimension on the ViT model in terms of loss, accuracy and time

- **GPU Utilization and Memory Efficiency:** GPUs excel at parallel processing, and their efficiency is partly dependent on operating at full capacity. Smaller embedding dimensions may not optimally utilize GPU parallelism or memory bandwidth, leading to inefficient processing. This inefficiency arises because smaller data loads do not saturate the GPU's compute cores nor fully exploit the memory transfer rates, causing longer training times despite a theoretically less complex model.

3.2.2 Number of blocks (nb_blocks)

In this section, we experiment with varying the number of blocks while keeping the other parameters constant at the following values: **embed_dim = 128, patch_size = 7, and nb_heads = 4.**

- **nb_blocks 2 and 4 (Blue and Orange):** These configurations achieve the lowest training loss within 20 epochs, suggesting an effective fit on the training dataset. However, their performance is weaker on the test set in terms of both loss and

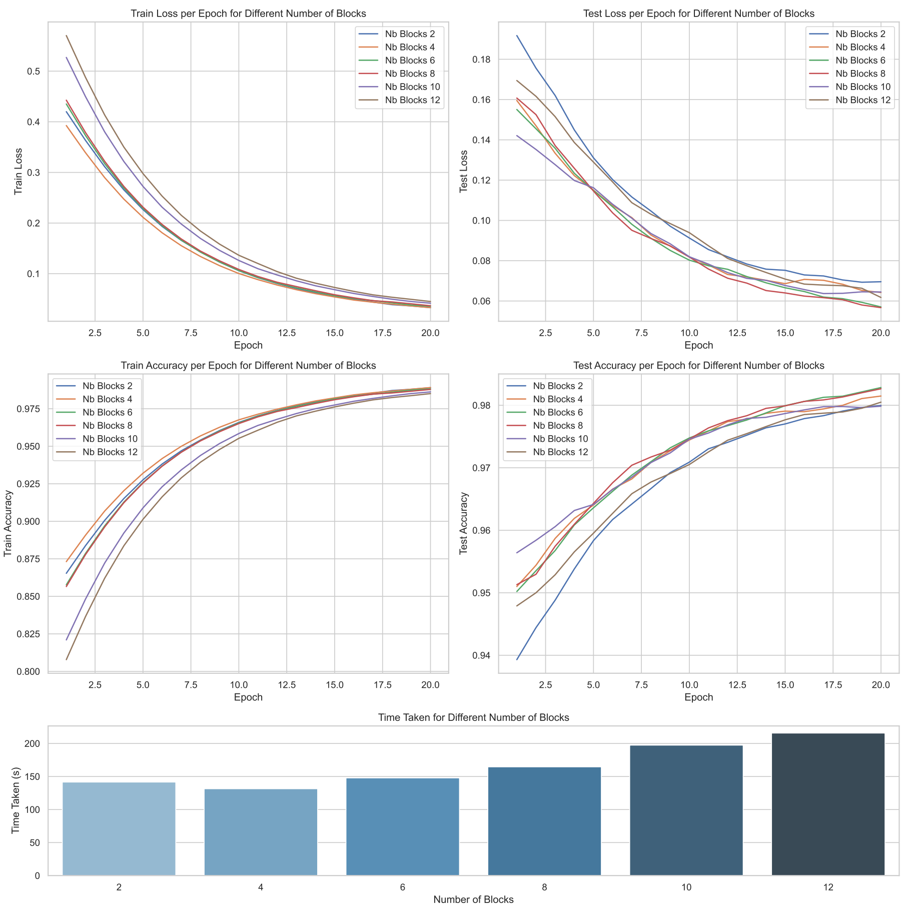


Figure 3.2: Influence of the number of blocks in the ViT model in terms of loss, accuracy and time

accuracy. This indicates underfitting, where the model's limited complexity doesn't allow it to capture enough information from the input patches to generalize well to unseen data.

- **nb_blocks 6 and 8 (Green and Red):** With a slight trade-off in training accuracy, these numbers of blocks yield the best accuracy on the test set. The model with 6 blocks converges faster than the one with 8, making it the more efficient choice. This suggests that 6 blocks are sufficient for the model to process and understand the input data adequately to perform well on both seen and unseen data, marking it as the optimal number in this scenario.

- **nb_blocks 10 and 12 (Purple and Brown):** These higher values do not perform as well as the 6 and 8 block configurations on either the training or the test sets. This is indicative of overfitting, where the model starts to learn noise and overly specific patterns from the training data that do not generalize to the test set.

- **Complexity and Training Duration:** As the number of encoder blocks increases, so does the model's complexity. More blocks mean the model has a greater number of parameters to learn and update during backpropagation, and the computational graph becomes more extensive. This not only increases the memory footprint but also the computational burden. As a result, each epoch will take longer to complete, leading to a longer overall training time. This is an important consideration when designing models for practical applications, where training time and resource consumption are critical factors. Balancing the complexity with the available resources and the required performance is key to building efficient and effective models.

Based on these plots, **nb_blocks = 6** seems to provide the best balance for the current scenario.

3.2.3 Patch sizes (patch_size)

In this section, we experiment with varying the patch sizes while keeping the other parameters constant at the following values: **embed_dim = 128, nb_blocks = 2, and nb_heads = 4.**

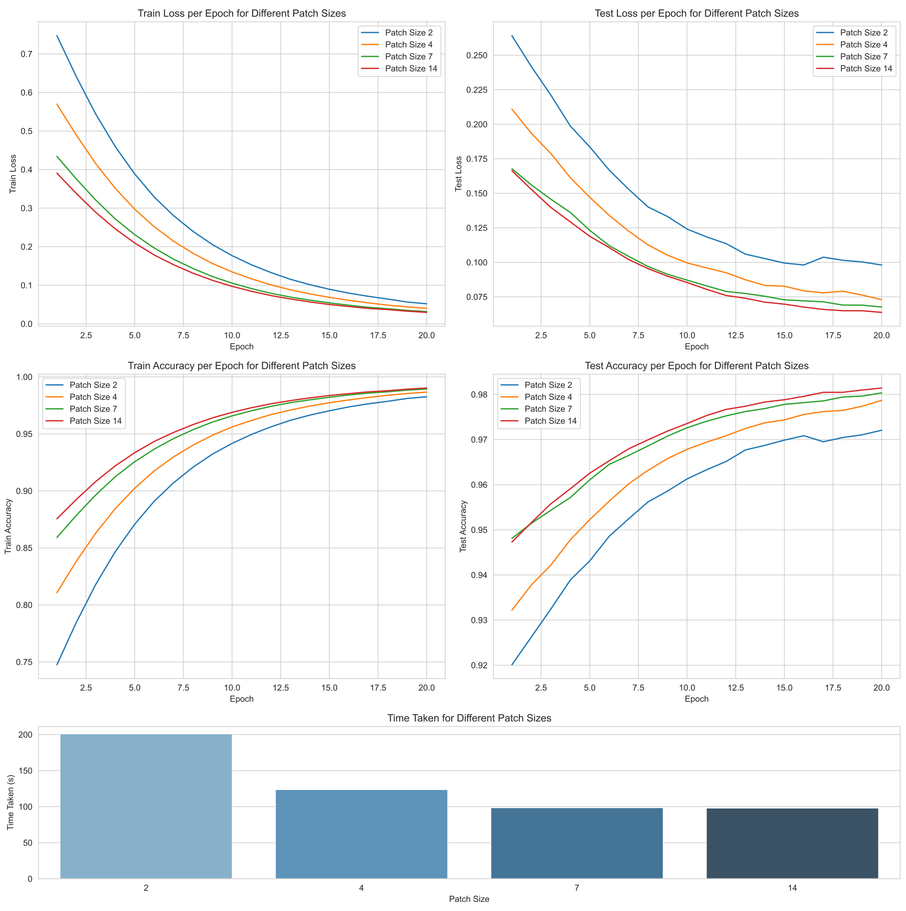


Figure 3.3: Influence of the patch size in the ViT model in terms of loss, accuracy and time

- **Patch Size 2 (Blue):** The smallest patch size results in the highest number of patches (196), which increases the model's complexity due to a focus on fine details. This is not ideal for a simple dataset like MNIST, leading to lower accuracy in both training and test datasets, as the model may overfit to the minute details rather than learn the broader, more general patterns. Small patch sizes, such as 1x1 pixels, are more applicable in contexts where fine details are crucial, such as medical imaging or facial recognition tasks.

- **Patch Sizes 4, 7, and 14 (Orange, Green, and Red):** These larger patch sizes produce fewer patches (49, 16, and 4 respectively), which aligns better with the simplicity of the MNIST dataset. Larger patches efficiently capture the main features of handwritten digits, such as loops or lines, without overcomplicating the model. They also result in shorter sequence lengths for the Transformer, simplifying the training process and aiding in faster learning. The larger the patch size, up to a certain point, the better the balance between capturing relevant features and model complexity, which can help to prevent overfitting. This is particularly relevant for MNIST, where important features have significant spatial extent and the global structure of the digits is more critical than local details.

- **Complexity and Training Duration:** There is a clear inverse relationship between patch size and training time. Smaller patch sizes lead to longer sequences, increasing computational load and training duration. Conversely, larger patch sizes shorten the sequence length, which reduces complexity and the time required for training. Therefore, choosing the right patch size is crucial to balance model complexity with computational efficiency. From the data provided, it is evident that for the given task and dataset, a patch size of 14 strikes an optimal balance, yielding the best training and test accuracy while also being efficient in terms of training time.

3.2.4 Final model and performance improvements:

By employing a model with optimized hyperparameters—specifically, selecting the best settings from our prior experiments (Number of blocks = 6, Embedding Dimension = 128, and Patch size = 14)—we achieved good results. After 20 epochs of training, the model attained an impressive accuracy of 97.78% on the test dataset, accompanied by a minimal training loss of 0.0275. Our strategy of selecting the parameter values isn't ideal since it assumes independance between the hyperparameters, which isn't the case. We fully acknowledge that a more thorough gridsearch of parameter combinations might offer better results.

To further enhance the performance of the ViT, there's a few strategies to explore:

- **Hyperparameter Optimization:** Conduct a grid search or use automated tools like Bayesian optimization to fine-tune hyperparameters like learning rate, batch size, and number of attention heads.
- **Data Augmentation:** Implement strategic data augmentation techniques such as rotations, scaling, or color jitter to improve generalization.
- **Regularization Techniques:** Use dropout or weight decay to reduce overfitting and enhance model robustness.
- **Learning Rate Scheduling:** Employ a learning rate scheduler to adjust the learning rate during training, aiding in better convergence.

3.3 Larger Transformers

3.3.1 Issues of applying ViTs from timm on MNIST:

- **Input Resolution:** ViTs are generally designed and pre-trained on images of a certain resolution, often larger than 28x28 pixels (MNIST's resolution). If we load a ViT model without pretrained weights, it still expects input images to match the resolution it was designed for. Feeding it smaller images like MNIST's can lead to issues where the patch size might not divide the image dimensions evenly.
- **Channel Mismatch:** ViTs, much like many modern CNN architectures, are typically designed for three-channel (RGB) color images. MNIST images, however, are single-channel (grayscale). If we pass a single-channel image to a ViT, it will expect three channels and thus will not be able to process the image properly.

To address the issues, we have to:

- **Resize the images:** Upscale the MNIST images to match the input size expected by the ViT model.
- **Adjust the channel dimensions:** Convert the single-channel grayscale images to three-channel by replicating the single grayscale channel three times.

For CNNs, these issues can also exist, but CNNs are generally more flexible in terms of input size and can handle varying resolutions better than ViTs due to their use of convolutional layers, which slide over the image regardless of its size. CNNs also can be more easily adapted to different channel inputs by adjusting the number of filters in the first convolutional layer.

3.3.2 Pretrained vs non-pretrained ViT models

In this section, we used the **vit_small_patch16_224** transformer, by specifying in the parameters **img_size as 28** and **num_classes as 10 (the default values are img_size = 224 and num_classes = 1000)**. The greyscale channel has been duplicated three times over in order to avoid dimension incompatibility errors. Here are the results when using a pretrained version vs a not pretrained one:

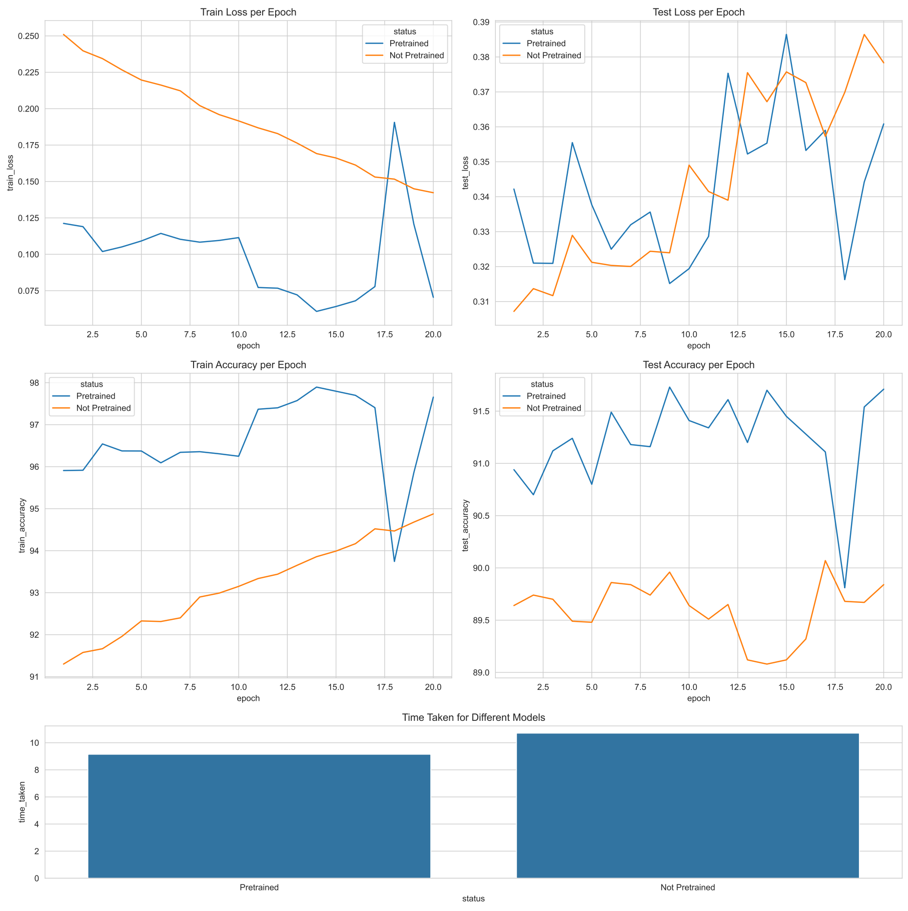


Figure 3.4: Pretrained vs non-pretrained transformers