M2 M2A

Report for RDFIA Homework

Section 1
Basics on deep learning for vision

Students :
Kai MA, Ruyi TANG, Jinyi WU

October 28, 2024

Contents

1-ab: Intro to Neural Networks

1 Theoretical foundation

1.1 Supervised dataset

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

In supervised learning, the data is typically split into three sets: train set, validation set, and test set.

- **Train Set**: The train set contains both features and target variables. It is used to train the model by learning the mapping between features and target variables. During training, the model parameters are updated by minimizing the loss function to get a well-trained model.
- **Validation Set**: The validation set also contains features and target variables, but it is not used to update the model's parameters. Instead, it is used during the training process to adjust hyper-parameters (such as learning rate) and select the best model. This helps to prevent over-fitting, where the model performs well on the training data but poorly on unseen data.
- **Test Set**: The test set contains only features. The trained model, after being tuned using the train and validation sets, is used to predict the target variables of the test set. The performance on the test set evaluates the model's generalization ability.

Q2 What is the influence of the number of examples N?

In general, the dataset size N is typically split into train set, validation set, and test set in a certain proportion. When N is large, the model benefits from a larger training sample, which leads to better performance. Conversely, if N is small, the model may suffer from insufficient data, resulting in poorer performance.

1.2 Network architecture (forward)

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

First, activation functions can introduce non-linearity between features and target variables, leading to better learning outcomes.

Second, specific activation functions can control the output range of the model. For example, the output range of the tanh function is $[-1, 1]$, which makes the output

more centralized. The SoftMax function has an output range of $[0, 1]$, and the sum of all outputs equals 1, which can effectively simulate a probability distribution.

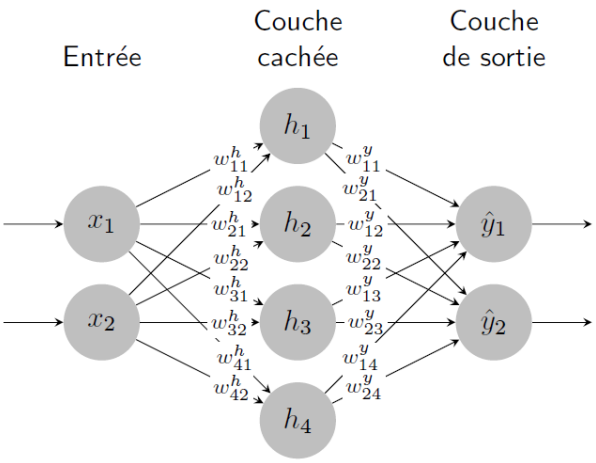


Figure 1: Neural network architecture with only one hidden layer.

Q4 What are the sizes n_x, n_h, n_y in the figure 1? In practice, how are these sizes chosen?

In the figure 1, the sizes are

$$n_x = 2, \quad n_h = 4, \quad n_y = 2$$

In practice, n_x represents the dimension of the input layer, which is the number of features for each sample. n_h represents the dimension of the hidden layer, and n_y represents the dimension of the output layer, which is the number of targets(in classification tasks, it is always the number of classes).

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

$\hat{y}$ represents the predicted value output by the model after inputting features, while y represents the given true value of target. In simple terms, the result of $\hat{y} - y$ is the error of the model.

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

The SoftMax function is defined as follows:

$$SoftMax(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n_x} e^{x_j}}$$

In multi-class classification problems, the `SoftMax` activation function converts raw scores (logits) into probabilities, producing a probability distribution over the target classes. This allows the model's output to be interpreted as the probability of each class.

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

The parameters are defined as follows:

$$W^h = \begin{bmatrix} w^h_{11} & w^h_{12} & \dots & w^h_{1n_x} \\ w^h_{21} & w^h_{22} & \dots & w^h_{2n_x} \\ \vdots & \vdots & \ddots & \vdots \\ w^h_{n_h1} & w^h_{n_h2} & \dots & w^h_{n_hn_x} \end{bmatrix}_{n_h \times n_x}, b^h = [b^h_1, b^h_2, \dots, b^h_{n_h}]_{1\times n_h}.$$

$$W^y = \begin{bmatrix} w^y_{11} & w^y_{12} & \dots & w^y_{1n_h} \\ w^y_{21} & w^y_{22} & \dots & w^y_{2n_h} \\ \vdots & \vdots & \ddots & \vdots \\ w^y_{n_y1} & w^y_{n_y2} & \dots & w^y_{n_yn_x} \end{bmatrix}_{n_y \times n_h}, b^y = [b^y_1, b^y_2, \dots, b^y_{n_y}]_{1\times n_y}.$$

- Firstly, we use a single sample $x = [x_1, \dots, x_{n_x}]_{1\times n_x}$.
  The forward pass equations are as follows:

$$\begin{cases} \tilde{h} = xW^{h^T} + b^h \\ h = \tanh(\tilde{h}) \\ \tilde{y} = hW^{y^T} + b^y \\ \hat{y} = \text{SoftMax}(\tilde{y}) \end{cases}$$

- Then, we use a batch of sample, so we have the input matrix

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n_x} \\ x_{21} & x_{22} & \dots & x_{2n_x} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{Nn_x} \end{bmatrix}_{N\times n_x}$$

  The forward pass equations becomes:

$$\begin{cases} \tilde{H} = XW^{h^T} + \text{repmat}_{N\text{raw}}(b_h) \\ H = \tanh(\tilde{H}) \\ \tilde{Y} = HW^{y^T} + \text{repmat}_{N\text{raw}}(b_y) \\ \hat{Y} = \text{SoftMax}_{\text{line}}(\tilde{Y}) \end{cases}$$

1.3 Loss function

Q8 During training, we try to minimize the loss function. For cross-entropy and squared error, how must $\hat{y}_i$ vary to decrease the global loss function L?

To decrease the loss function using the gradient descent method, in each iteration we update $\hat{y}^{(t+1)}$ as follows:

$$\hat{y}^{(t+1)} = \hat{y}^{(t)} - \alpha \cdot \nabla_{\hat{y}} l(\hat{y}, y)$$

where α is the learning rate, and $\nabla_{\hat{y}} l(\hat{y}, y)$ is the gradient of the loss function $l(\hat{y}, y)$ with respect to $\hat{y}$.

- For the **cross-entropy**

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

so we should update $\hat{y}_i$ by $\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} + \alpha \frac{y_i}{\hat{y}_i^{(t)}}$

- For the **MSE**

$$l(y, \hat{y}) = \sum_i (y_i - \hat{y}_i)^2$$
$$\nabla_{\hat{y}_i} l(y, \hat{y}) = -2(y_i - \hat{y}_i)$$

so we should update $\hat{y}_i$ by $\hat{y}_i^{(t+1)} = \hat{y}_i^{(t)} + 2\alpha(y_i^{(t)} - \hat{y}_i^{(t)})$.

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

- The cross-entropy works better on classification tasks Take a binary classification problem for example, the loss function is defined as $l(y, \hat{y}) = -(y_1 \log \hat{y}_1 + y_2 \log \hat{y}_2)$. Assuming $y = (0, 1)$, as $\hat{y}$ gets closer to y, the loss function decreases rapidly, and the gradient approaches -1. This effectively avoids the vanishing gradient problem;
- The MSE works better on regression tasks In regression problems, the target variable is continuous. The Mean Squared Error (MSE) calculates the average squared difference between the true values and the predicted values. Larger errors contribute more significantly to the loss function, which encourages the model to minimize these errors.

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 one seems the most reasonable to use in the general case?

Classic Gradient Descent:

- **Advantage:** In each iteration, the gradient of the loss function is calculated using the entire training set to update the parameters, which ensures a very stable direction for gradient updates.

- **Disadvantage:** Each update requires calculating the gradient for the entire training set, which results in high computational cost and memory usage, especially when there are many samples in the training set, leading to low efficiency.

Mini-Batch Stochastic Gradient Descent:

- **Advantage:** In each iteration, a mini-batch of samples is used to traverse the entire dataset. Each update only uses a mini-batch of samples, which makes it faster compared to classic gradient descent, while maintaining a certain level of stability.
- **Disadvantage:** The gradient for each update is based on a mini-batch of samples, which introduces some noise.

Online Stochastic Gradient Descent:

- **Advantage:** In each iteration, only one sample is used to update the parameters, which results in a high update frequency and low memory requirements. It can quickly adapt to changes in data, making it suitable for online learning and scenarios where data is generated in real-time.
- **Disadvantage:** Since each update is based on a single sample, the gradient has large fluctuations, which may lead to unstable convergence or slower convergence speed.

Summary: Mini-batch stochastic gradient descent achieves both fast updates and stable convergence, making it suitable for large datasets and allowing GPU acceleration. Therefore, it is the most commonly used in practice.

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

The learning rate η is a hyper-parameter that controls the speed of updates during optimization, following the rule:

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

A larger learning rate η results in larger update steps, which may lead to divergence and prevent convergence. Conversely, a smaller learning rate η results in smaller update steps, which may slow down the convergence process. Therefore, it is important to choose an appropriate learning rate, and a common value is $\eta = 0.01$.

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 *backprop* algorithm.

Consider a neural network with L hidden layers (figure 2). We compare the computational complexity using the naive approach and the backpropagation algorithm.

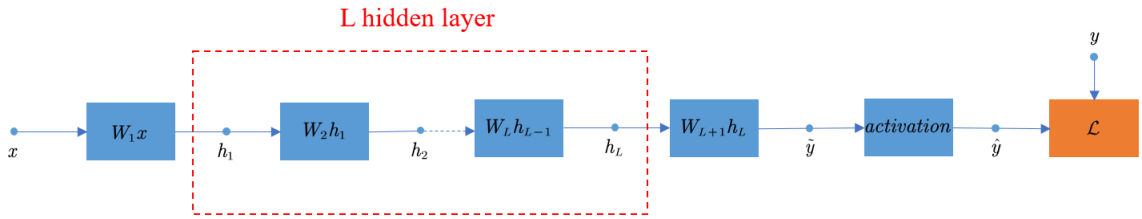


Figure 2: A neural network with L hidden layer

- **naive approach**
  Based on the chain rule for differentiation, we have:

$$\frac{\partial \mathcal{L}}{\partial W_{L+1}} = \frac{\partial \mathcal{L}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial W_{L+1}}$$
$$\frac{\partial \mathcal{L}}{\partial W_L} = \frac{\partial \mathcal{L}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial h_L} \cdot \frac{\partial h_L}{\partial W_L}$$
$$\cdots$$
$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \tilde{y}} \cdot \frac{\partial \tilde{y}}{\partial h_L} \cdot \frac{\partial h_L}{\partial h_{L-1}} \cdots \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W_1}$$

  Since calculating the gradient for a given layer requires recomputing the derivatives from the output layer back to the current layer each time, the time complexity of the naive approach is $\mathcal{O}(n^L)$, where n is the number of neurons per layer, and L is the total number of layers.

- ***backprop* algorithm**

In this approach, we can avoid redundant computations by calculating gradients layer by layer and caching the intermediate results:

$$\delta_{L+1} = \frac{\partial \mathcal{L}}{\partial \tilde{y}}, \qquad \frac{\partial \mathcal{L}}{\partial W_{L+1}} = \delta_{L+1} \cdot \frac{\partial \tilde{y}}{\partial W_{L+1}}$$
$$\delta_L = \delta_{L+1} \cdot \frac{\partial \tilde{y}}{\partial h_L}, \qquad \frac{\partial \mathcal{L}}{\partial W_L} = \delta_L \cdot \frac{\partial h_L}{\partial W_L}$$
$$\cdots \qquad \cdots$$
$$\delta_1 = \delta_2 \cdot \frac{\partial h_2}{\partial h_1}, \qquad \frac{\partial \mathcal{L}}{\partial W_1} = \delta_1 \cdot \frac{\partial h_1}{\partial W_1}$$

The gradient is computed by propagating the error backward through each layer. Each layer only requires the gradient and input from the previous layer, without needing to recompute the chain rule derivatives for all preceding layers. Therefore, the time complexity is $\mathcal{O}(nL)$, which significantly improves the computational efficiency compared to the naive approach, making gradient calculation feasible in deep learning.

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

All functions(including the loss function, the active function and the Linear layer) must be differentiable with respect to the model's output, allowing the error to be propagated backward.

Q14 The function SoftMax and the *loss* of *cross-entropy* are often used together and their gradient is very simple. Show that the loss can be simplified by:

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

We can simplify the loss by:

$$\begin{aligned}
\ell &= -\sum_i y_i log(\hat{y}_i) \\
&= -\sum_i y_i log(\frac{e^{\tilde{y}_i}}{\sum_i e^{\tilde{y}_i}}) \\
&= -\sum_i y_i \tilde{y}_i + \sum_i y_i log(\sum_i e^{\tilde{y}_i}) \\
&= -\sum_i y_i \tilde{y}_i + log(\sum_i e^{\tilde{y}_i}), \quad since \sum_i y_i = 1
\end{aligned}$$

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

$$\frac{\partial \ell}{\partial \tilde{y}_i} = \cdots \quad \Rightarrow \quad \nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \cdots$$

Based on the result from the previous question, we take the derivative with respect to $\tilde{y}_i$:

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{y}_i} &= -y_i + \frac{e^{\tilde{y}_i}}{\sum_i e^{\tilde{y}_i}} \\
&= -y_i + \hat{y}
\end{aligned}$$

Then we have:

$$\nabla_{\tilde{y}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{y}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix} = \hat{y} - y$$

Q16 Using the *backpropagation*, write the gradient of the loss with respect to the weights of the output layer$\nabla_{W_y}\ell$. Note that writing this gradient uses $\nabla_{\tilde{y}}\ell$. Do the same for $\nabla_{b_y}\ell$.

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \cdots \quad \Rightarrow \quad \nabla_{W_y}\ell = \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_yn_h}} \end{bmatrix} = \cdots$$

First we calculate $\frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$:

$$\begin{aligned} \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} &= \frac{\partial([h \cdot W_y^T + b_y]_k)}{\partial W_{y,ij}} \\ &= \frac{\partial(\sum_{p=1}^{n_h} h_p W_{y,kp} + b_k)}{\partial W_{y,ij}} \\ &= \begin{cases} h_j, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases} \end{aligned}$$

Then we have:

$$\begin{aligned} \frac{\partial \ell}{\partial W_{y,ij}} &= \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \frac{\partial \ell}{\partial \tilde{y}_j}\frac{\partial \tilde{y}_j}{\partial W_{y,ij}} = (\hat{y}_i - y_i)h_j \\ \nabla_{W_y}\ell &= \begin{bmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_yn_h}} \end{bmatrix} \\ &= \begin{bmatrix} (\hat{y}_1 - y_1)h_1 & \cdots & (\hat{y}_1 - y_1)h_{n_h} \\ \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y})h_1 & \cdots & (\hat{y}_{n_y} - y_{n_y})h_{n_h} \end{bmatrix} \\ &= (\hat{y} - y)^T h \end{aligned}$$

Q17 Compute other gradients:$\nabla_{\tilde{h}}\ell, \nabla_{W_h}\ell, \nabla_{b_h}\ell$.

- **Compute $\nabla_{\tilde{h}}\ell$**

Given that $\tanh'(x) = 1 - \tanh^2(x)$, therefore:

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{h}_i} &= \sum_{k=1}^{n_y} \frac{\partial \ell}{\partial \tilde{y}_k} \sum_{j=1}^{n_h} \frac{\partial \tilde{y}_k}{\partial h_j} \frac{\partial h_j}{\partial \tilde{h}_i} \\
&= \sum_{k=1}^{n_y} (\hat{y}_k - y_k) \frac{\partial \tilde{y}_k}{\partial h_i} \frac{\partial h_i}{\partial \tilde{h}_i} \\
&= \sum_{k=1}^{n_y} (\hat{y}_k - y_k) \frac{\partial (\sum_{p=1}^{n_h} h_p W_{y,kp} + b_k)}{\partial h_i} (1 - h_i^2) \\
&= (1 - h_i^2) \sum_{k=1}^{n_y} (\hat{y}_k - y_k) W_{y,ki}
\end{aligned}$$

$$\nabla_{\tilde{h}} \ell = \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{h}_1} \\ \vdots \\ \frac{\partial \ell}{\partial \tilde{h}_{n_h}} \end{bmatrix} = \begin{bmatrix} (1 - h_1^2) \sum_{k=1}^{n_y} (\hat{y}_k - y_k) W_{y,k1} \\ \vdots \\ (1 - h_{n_h}^2) \sum_{k=1}^{n_y} (\hat{y}_k - y_k) W_{y,kn_h} \end{bmatrix} = (1 - h^2) \odot (\nabla_{\tilde{y}} \mathbf{W}^y)$$

- **Compute $\nabla_{W_h} \ell$**

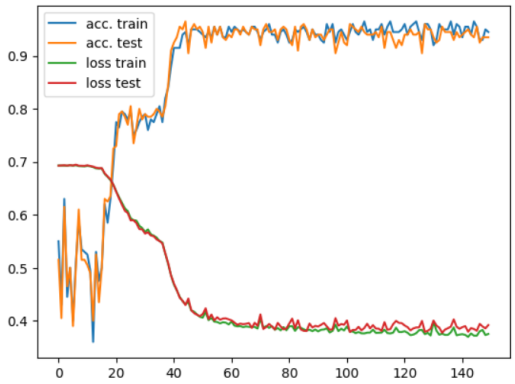
$$\begin{aligned}
\frac{\partial \ell}{\partial W_{h,ij}} &= \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} \\
&= \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial (\sum_{p=1}^{n_x} h_p W_{h,kp} + b_k)}{\partial W_{h,ij}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_i} x_j \\
\nabla_{W_h} \ell &= \begin{bmatrix} \frac{\partial \ell}{\partial W_{h,11}} & \cdots & \frac{\partial \ell}{\partial W_{h,1n_x}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{h,n_h 1}} & \cdots & \frac{\partial \ell}{\partial W_{h,n_h n_x}} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial \ell}{\partial \tilde{h}_1} x_1 & \cdots & \frac{\partial \ell}{\partial \tilde{h}_1} x_{n_x} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial \tilde{h}_{n_h}} x_1 & \cdots & \frac{\partial \ell}{\partial \tilde{h}_{n_h}} x_{n_x} \end{bmatrix} \\
&= (\nabla_{\tilde{h}} \ell)^T x
\end{aligned}$$

- **Compute $\nabla_{b_h} \ell$**

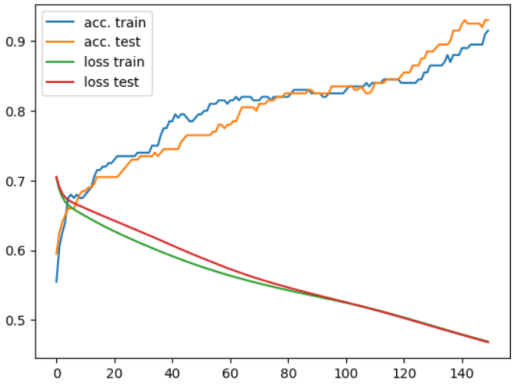
$$\begin{aligned}
\frac{\partial \ell}{\partial b_{h,i}} &= \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} \\
&= \frac{\partial \ell}{\partial \tilde{h}_k} \\
\nabla_{b_h} \ell &= (\nabla_{\tilde{h}} \ell)^T
\end{aligned}$$

2 Implementation

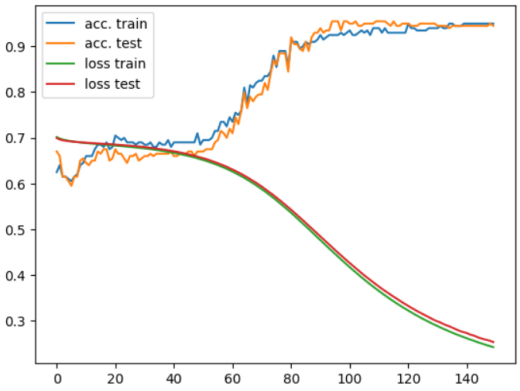
We apply different models to the CirclesData dataset and compare their performance.



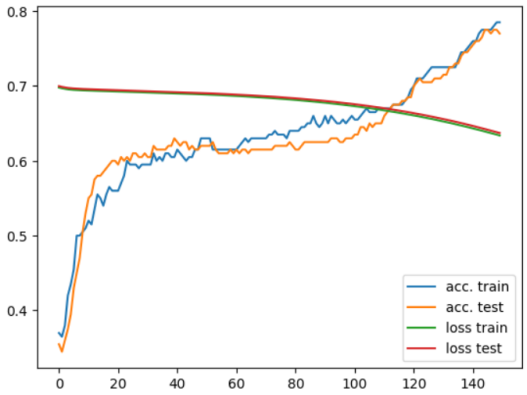
(a) Model 1 (Forward and backward manuals)



(b) Model 2 (backward pass with torch.autograd)



(c) Model 3 (forward pass with torch.nn layers)



(d) Model 4 (SGD with torch.optim)

Figure 3: Comparison of different models

Finally, the fourth model was applied to the MNIST dataset, achieving the following results:

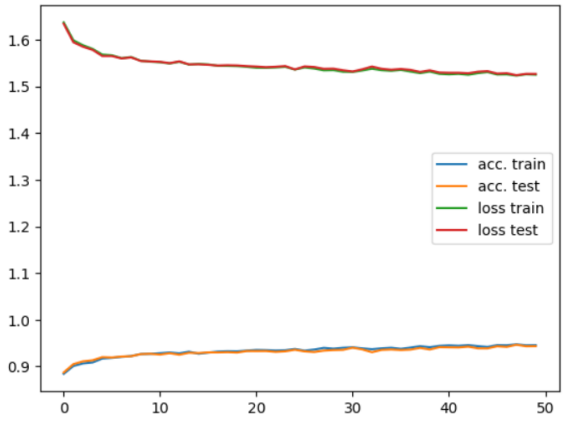


Figure 4: Model 4 (MINST dataset)

We also trained an SVM on the Circle dataset and observed that it performed well in this case, with faster execution than the neural network.

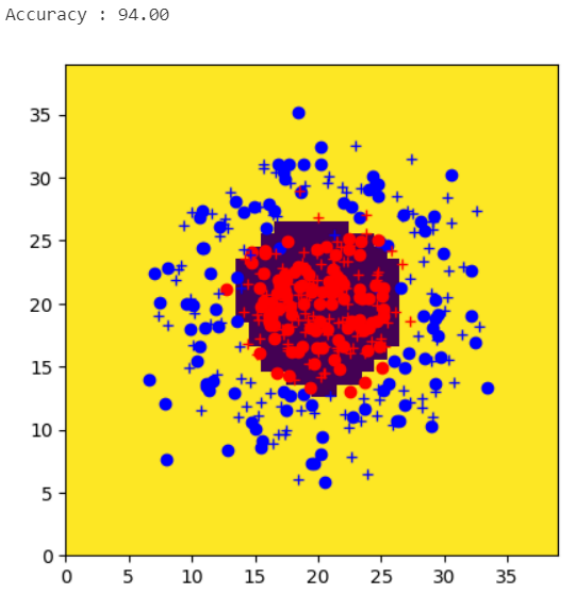


Figure 5: Model 5 (SVM)

3 Conclusion

In this section, we studied the mathematical principles of neural networks and manually implemented backpropagation, neural network construction, and parameter optimization. Finally, we used `torch.autograd`, `torch.nn`, and `torch.optim` to implement an automated backpropagation neural network.

As shown in the figure 3, the first model is the most stable, while the third model performs the best. The second and fourth models perform slightly worse, but the last three models are faster than the first one. We suspect that some internal optimizations in the torch library have caused these differences in results.

1-cd: Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have become a fundamental architecture in machine learning, particularly for image recognition tasks. This part focuses on understanding and implementing CNNs, specifically applying them to the CIFAR-10 dataset, which contains RGB images across 10 classes. The tasks covered involve building a CNN architecture, training it from scratch, and applying various techniques such as normalization, data augmentation, and regularization to improve performance.

The goal of this practical work is to deepen our understanding of CNNs through hands-on experimentation. We will analyze the network architecture and training techniques and explore methods to enhance learning stability and accuracy.

1 Introduction to Convolutional Networks

In this part, we gained an initial understanding of the architecture of convolutional neural networks (CNNs). We explored how convolutional and pooling layers function as the fundamental building blocks of CNNs, along with their roles in feature extraction and dimension reduction. Additionally, we discussed how CNNs differ from fully connected layers and the importance of receptive fields in capturing spatial information across different layers of the network.

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?
How much weight is there to learn ?
How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size ?

- For this convolution filter, the output size is calculated as $x_{\text{output}} \times y_{\text{output}}$, where:

$$x_{\text{output}} = \frac{(x - k + 2p)}{s} + 1, \quad y_{\text{output}} = \frac{(y - k + 2p)}{s} + 1$$

The depth of the output is 1, as there is only a single convolution filter.

- The total number of weights to learn for this convolution filter is:

$$\text{weights}_{\text{cv}} = k \times k \times z$$

where $k \times k$ is the spatial size of the filter, and z is the depth of the input.

- To produce an output of the same size with a fully-connected layer, each output neuron must connect to every input neuron. Therefore, the number of weights needed in a fully-connected layer would be:

$$\text{weights}_{\text{fc}} = (x \times y \times z) \times (x_{\text{output}} \times y_{\text{output}})$$

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

Compared to fully-connected layers, convolutional layers have these advantages:

Fewer Parameters : Convolutional layers use shared weights so they need fewer parameters (or weights to learn) than fully-connected layers. This makes the calculations easier and helps prevent overfitting.

Focus on Local Details : Convolutional layers only look at a small part of the image at a time (like a 3x3 square). This helps them detect small features in the image, such as edges or corners.

Translation Invariance : Since the same filter slides across the image, convolutional layers can detect features even if they are shifted or translated.

The main **limitation** of convolutional layers is that they only has access to a local region of the input and cannot see the whole image at once. As the result, they might miss the overall structure of the image due to focusing on local patterns.

Q3* Why do we use spatial pooling?

We use spatial pooling to reduce the spatial dimensions of feature maps while retaining important information. This reduction lowers computation and storage needs and maintains translation invariance. By reducing the input size, pooling also decreases the number of parameters in subsequent layers, which helps prevent overfitting.

Q4* Suppose we try to compute the output of a classical convolutional network for an input image larger than the initially planned size (224×224 in the example). Can we (without modifying the image) use all or part of the layers of the network on this image?

We can only use part of the layers of this network, specifically the convolutional layers and pooling layers since these layers can handle images larger than originally planned, as they are applied locally and do not depend on the input size. However, the fully-connected layers would need to be modified, as they require a fixed-size input.

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

Fully-connected layers can be viewed as a global convolution with no padding, stride set to 1, and a kernel size equal to the input dimensions. The number of kernels matches the number of outputs in the fully-connected layer. Thus, each neuron is matched to a kernel and fully connected to the entire input.

Q6 Suppose that we therefore replace fully-connected layers by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest?

If we replace fully-connected layers with equivalent convolutional layers, the output can still be calculated. However, the shape of the output might change only if the input image is larger than the originally planned size, in this case output will be a feature map instead of a single vector. The interests are that it allows the network to adapt to different input sizes.

Q7 We call the receptive field of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers? Can you imagine what happens to the deeper layers? How to interpret it?

In the first convolutional layer, the receptive field is small and very limited since each filter only processes a limited region of the input image. By the second convolutional layer, the receptive field expands as it captures more global patterns from the output of the first layer. As the network deepens, receptive fields continue to grow, eventually covering the entire image. To interpret it, This hierarchical structure enables the network to gradually build a complete understanding of the image.

2 Training from Scratch of the Model

In this part, we implement a convolutional network from scratch and apply it to the CIFAR-10 dataset. After defining the architecture, we train the network while adjusting parameters such as batch size and learning rate, then analyze the effects on model performance and convergence.

2.1 Network architecture

The CIFAR-10 dataset contains 32×32 pixel RGB images across 10 classes. Thus, the network should be structured to accept an input of size $32 \times 32 \times 3$ and produce an output of size 10. The architecture is structured with the following layers:

- `conv1`: 32 convolutional filters of size 5×5, followed by ReLU.
- `pool1`: 2×2 max-pooling.
- `conv2`: 64 convolutional filters of size 5×5, followed by ReLU.
- `pool2`: 2×2 max-pooling.
- `conv3`: 64 convolutional filters of size 5×5, followed by ReLU.
- `pool3`: 2×2 max-pooling.
- `fc4`: Fully-connected layer with 1000 neurons, followed by ReLU.
- `fc5`: Fully-connected layer with 10 neurons, followed by softmax

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

To maintain the same spatial dimensions at the output, padding should be set to $p = \frac{k-1}{2}$ (where k is the kernel size) with a stride of 1. For a kernel size of 5, this gives $p = 2$ and $s = 1$.

Q9 For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed ?

To halve the spatial dimensions, we can set stride to $s = 2$ with no padding when using a 2×2 pooling window.

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

The output size for each layer is calculated below:

Layer	Output Size	Number of Weights
`conv1`	$32 \times 32 \times 32$	$5 \times 5 \times 3 \times 32 = 2400$
`pool1`	$16 \times 16 \times 32$	None
`conv2`	$16 \times 16 \times 64$	$5 \times 5 \times 32 \times 64 = 51200$
`pool2`	$8 \times 8 \times 64$	None
`conv3`	$8 \times 8 \times 64$	$5 \times 5 \times 64 \times 64 = 102400$
`pool3`	$4 \times 4 \times 64$	None
`fc4`	1000	$4 \times 4 \times 64 \times 1000 = 1024000$
`fc5`	10	$1000 \times 10 = 10000$

Table 1: Output size and number of weights for each layer

In terms of weight distribution, most parameters are concentrated in the last two fully-connected layers. However, as the network deepens, the number of parameters in the convolutional layers also increases, as we add more convolutional kernels to capture more complex features.

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

The total number of weights to learn is:

$$2400 + 51200 + 102400 + 1024000 + 10000 = 1190000$$

The CIFAR-10 dataset has 50,000 training images. The model has more parameters than training examples. This can lead to overfitting.

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

BoW : Assuming a visual vocabulary for image features, BoW requires around 1000 parameters.

SVM : Assuming a feature dimension of $32 \times 32 \times 3 = 3072$ (flattened CIFAR-10 images) and around 500 support vectors, the SVM model has approximately 1,536,500 parameters.

In comparison, CNNs have a significantly higher parameter count compared to BoW and SVM because they utilize deep layers to learn complex, hierarchical features.

2.2 Network learning

In this section, we'll explore training a neural network to understand the effects of key training parameters, analyze model performance, and study convergence behavior.

Q13 Read and test the code provided. You can start the training with this command : `main(batch_size, lr, epochs, cuda = True)`

The code has been read, tested, and executed successfully in practical.

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 the difference in data) ?

During training, the model's parameters are updated based on the calculated loss after each batch, involving a backward propagation and optimization step. In contrast, during testing, the model runs in evaluation mode, meaning no gradient calculations or updates occur. Additionally, some layers, like Dropout behave differently in training and testing to provide more stable and generalized predictions during testing.

Q15 Modify the code to use the CIFAR-10 dataset and implement the architecture requested above. (the class is `datasets.CIFAR10` **). Be careful to make enough epochs so that the model has finished converging.**

The code was successfully executed , with the epoch count set to 20 to confirm convergence.

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

Learning Rate : A higher learning rate speeds up training but can lead to unstable convergence or overshooting the optimal solution. A lower learning rate improves accuracy by making smaller, more precise adjustments but can slow down training.

Batch Size : A larger batch size provides more stable gradient estimates, which can lead to smoother convergence but requires more memory. Smaller batch sizes can introduce noise, which may help escape local minima but can make training less stable.

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

At the start of training, the average loss for the first batch in the first training epoch is 2.3014, while the average loss for the first test epoch is 1.9302. The high initial training error is due to the model's randomly initialized weights, meaning it has not

yet learned any meaningful patterns from the data. After the first training round, the test error shows some improvement, though there remains significant potential for further reduction. These high error rates reflect the model's lack of recognition ability.

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

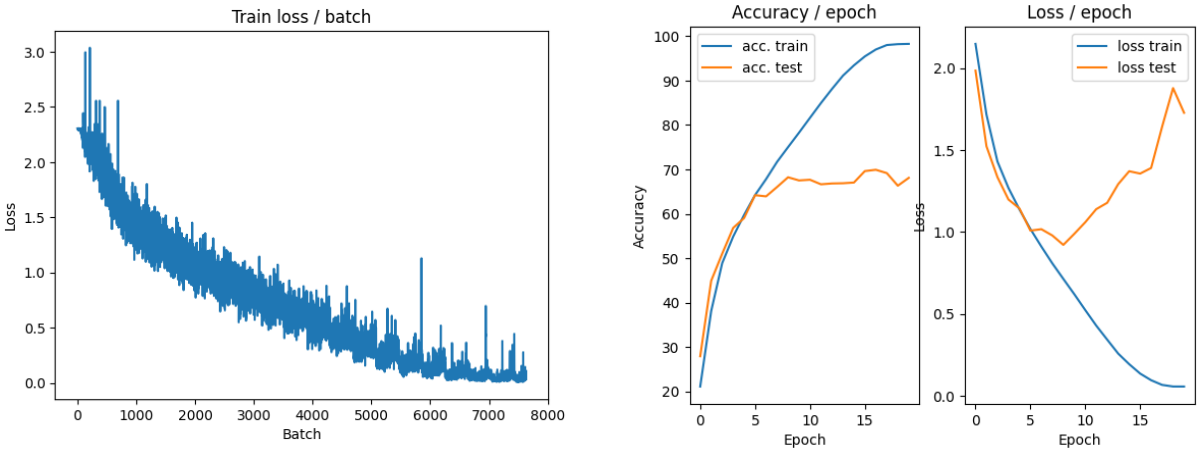


Figure 6: First Training Results

At the beginning of training, both training and test losses initially decrease. However, as training progresses, the training accuracy approaches 100%, while the test accuracy levels off around 70-75%, and the test loss begins to rise again. This pattern strongly indicates overfitting: the model learns the training data too well but fails to generalize effectively to the test data. Also, the model converges very slowly.

3 Results Improvements

We will now use several classic techniques to improve the performance of our model.

3.1 Standardization of examples

A common technique in machine learning is to standardize the examples in order to better condition the learning. When learning CNN, the most common technique is to calculate the mean value and the standard deviation of each RGB channel over the whole train. In practical, we add normalization in the data pre-processing while calling `datasets.CIFAR10` :

Q19 Describe your experimental results.

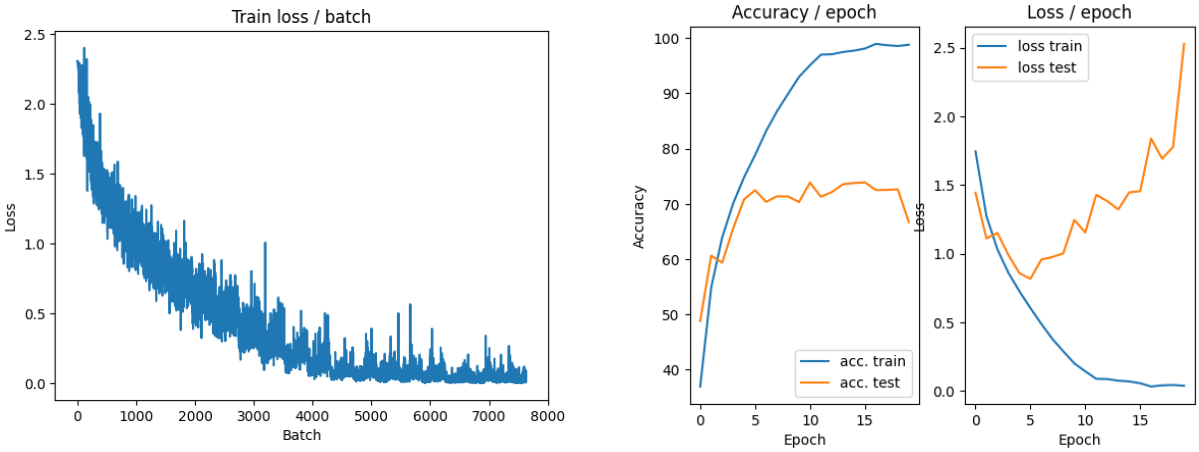


Figure 7: Results by adding standardization

With normalization, training is more stable and converges faster compared to the previous results without normalization. However, the test accuracy stabilizes around 70%, and the test loss eventually starts to increase, indicating overfitting. To conclude, normalization does not sufficiently prevent overfitting. Further regularization techniques are necessary for improved generalization.

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

The validation set should be independent of the training process to avoid data leakage. This way, the model doesn't "see" or adjust to the test data during training, otherwise the model performance could be over-estimates.

Q21 Bonus: There are other normalization schemes that can be more efficient like ZCA normalization. Try other methods, explain the differences and compare them to the one requested.

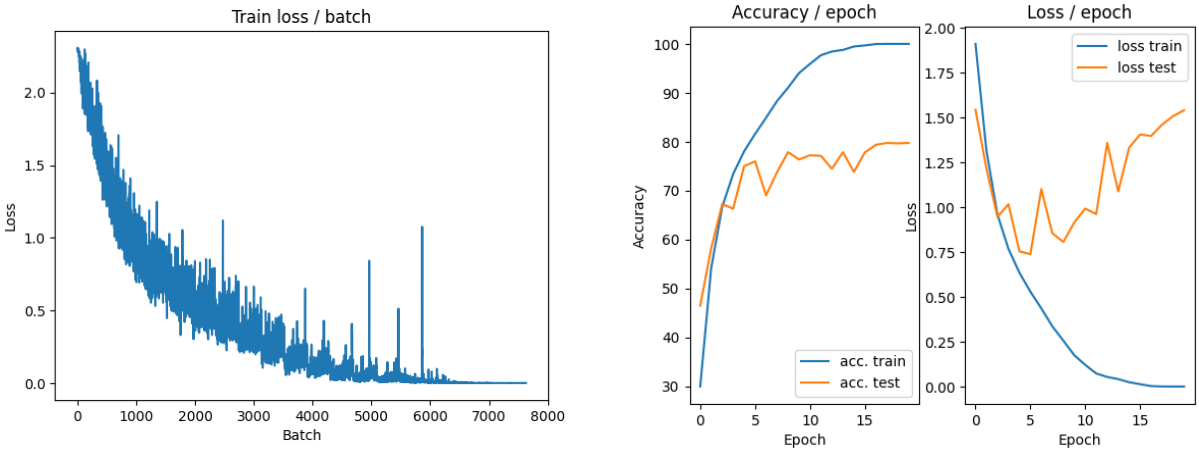


Figure 8: Results by adding ZCA normalisation

We implemented ZCA normalization, which aims to decorrelate the features while preserving spatial relationships in the data. Unlike standard normalization tech-

niques like min-max scaling or z-score normalization, which only scale each feature independently, ZCA focuses on transforming features in a way that reduces redundancy in the data. This helps retain the structure within the dataset while making features more statistically independent.

The results show that ZCA normalization can accelerate the convergence of the training loss due to the reduced redundancy and improved statistical independence. However, similar to other normalization techniques, it does not directly address overfitting, as the performance gap between the training and validation sets remains.

3.2 Increase in the number of training examples by data increase

When training convolutional networks on the relatively small CIFAR-10 dataset, we can use data augmentation to prevent overfitting by artificially increasing training samples.Here, we test two most common transformations : a random crop (of fixed size) and a random horizontal symmetry of the image. We applied these two transformations only to the training set and used a centered crop in the test set to ensure consistent evaluation and avoid random variations in the test data that could impact model performance measurements. Moreover, to maintain the same number of model parameters (for easier comparison of results), the `pool3` layer is modified with `ceil_mode=True` to keep the output size fixed.

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

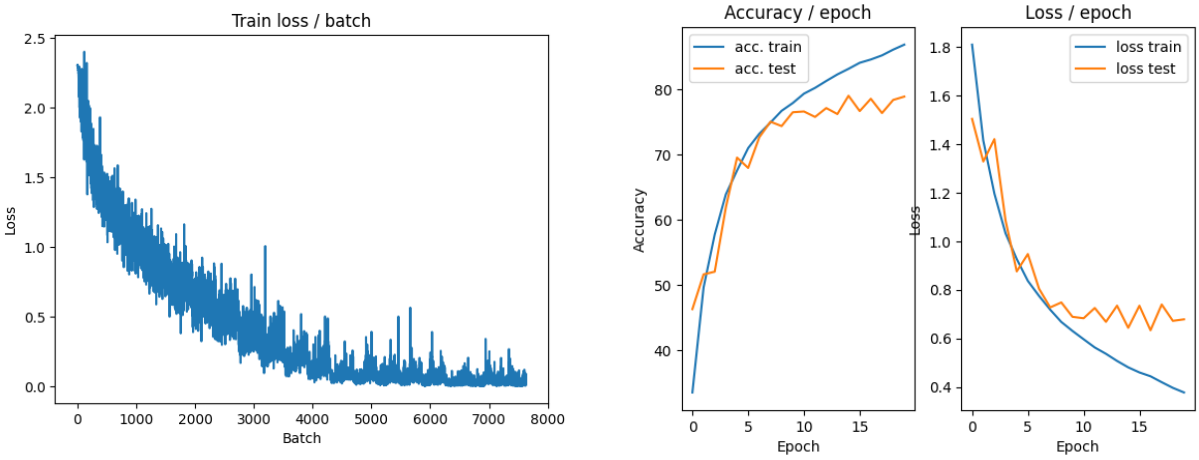


Figure 9: Results by data increase

The current results show that the model's training loss gradually decreases in a stable manner, indicating a smooth learning process and steady convergence. The training accuracy approaches 90%, while the test accuracy stabilizes around 75-78%, reflecting an improvement in generalization. Additionally, the test loss curve remains relatively stable with minor fluctuations, suggesting a reduction in overfitting. However, the convergence speed hasn't significantly increased.

Compared to the previous results, the addition of data augmentation techniques—such as random cropping and horizontal flipping—has indeed improved the model's performance on the test set. The test accuracy is higher and more consistent, showing that the model's adaptability to different data has improved. Furthermore, both the training and test loss curves are smoother, indicating that the model has learned more generalizable features from the diverse data.

Q23 Does this horizontal symmetry approach seems usable on all types of images ? In what cases can it be or not be ?

No, horizontal flipping isn't suitable for all images. Although horizontal flipping works well for images where orientation doesn't change the meaning, like landscapes or animals, for images that rely on a specific direction, like text or traffic signs, flipping can create misleading or incorrect data, as the direction can be essential to understanding the image.

Q24 What limits do you see in this type of data increase by transformation of the dataset?

Augmentation through transformations only generates variations of existing data rather than truly new data. For data with distinct patterns or limited samples, simply transforming images might not be enough to cover all possible cases. In these situations, the model could still struggle with entirely new data types or extreme variations.

Q25 Bonus: Other data augmentation methods are possible. Find out which ones and test some.

We identified several common **data augmentation methods** that help improve model generalization and prevent overfitting. These methods include:

- **Color Jitter**: Randomly changing the image's brightness, contrast, saturation, and hue simulates different lighting conditions, helping the model adapt to variations in color and lighting.
- **Gaussian Blur**: Applying slight blurring to images enhances the model's robustness to noise and blurry conditions.
- **Random Rotation**: Randomly rotating images within a specified angle range improves the model's ability to recognize objects at different orientations.

We experimented with combinations of the Color Jitter and Random Rotation techniques to verify their effectiveness.

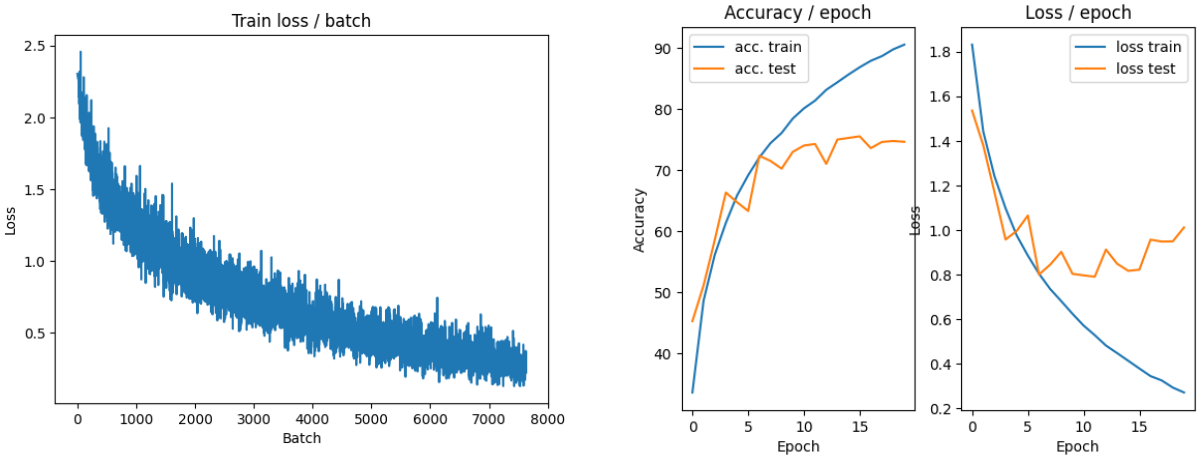


Figure 10: Results by using other data augmentation methods

Although the results are not as immediately impactful as those from cropping and symmetry, we can observe that overfitting in the model has been notably reduced.

3.3 Variants on the optimization algorithm

We can also improve our model by refining the optimization process. We introduce here a learning rate scheduler, which dynamically adjusts the learning rate during training to enhance convergence and adapt to changing gradients.

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

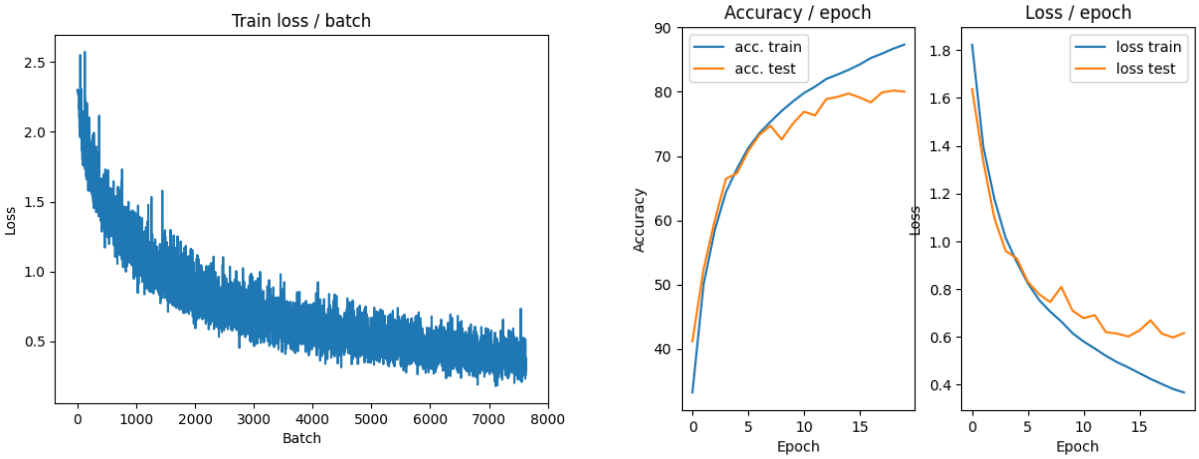


Figure 11: Results by using lr_scheduler

The training loss decreases more steadily and rapidly, indicating that exponential decay helps the model converge faster. The accuracy on the training set continues to rise, approaching around 90%, showing good model fit. The test accuracy stabilizes around 80%, reflecting improved generalization. The test loss curve is also smoother

and stabilizes at a lower level, showing reduced overfitting and more consistent performance across batches.

Compared to using a fixed learning rate, the exponential decay strategy has led to faster convergence, with the training loss reaching lower levels in fewer epochs. Test accuracy and loss reach stable values sooner, demonstrating improved adaptability to unseen data. Additionally, the fluctuations in the test loss curve have decreased, indicating that the model's performance across different test batches is more consistent, which further reduces overfitting.

Q27 Why does this method improve learning ?

Decaying the learning rate allows the model to make large, quick updates in the beginning and then focus on finer adjustments as training progresses. This avoids taking too-large steps that could miss the optimal solution, helping the model converge more stable and quickly by reducing its movement as it nears convergence.

Q28 Bonus: Many other variants of SGD exist and many *learning rate* planning strategies exist. Which ones ? Test some of them.

We explored several SGD variants and learning rate scheduling strategies that can potentially improve model convergence and stability. These methods include:

- **SGD with Momentum**: Adds momentum to the standard SGD algorithm, which helps accelerate convergence and reduces oscillations by averaging gradients over time.
- **NAG**: An improvement over momentum-based SGD that anticipates the gradient direction for a more accurate update.
- **Adam**: Combines momentum with adaptive learning rates, making it well-suited for models with noisy gradients and sparse data.
- **Cosine Annealing**: Gradually decreases the learning rate following a cosine curve, allowing the model to settle into minima effectively.

We experimented with combinations of the SGD with Momentum and Exponential Decay scheduler to see if it works better than SGD.

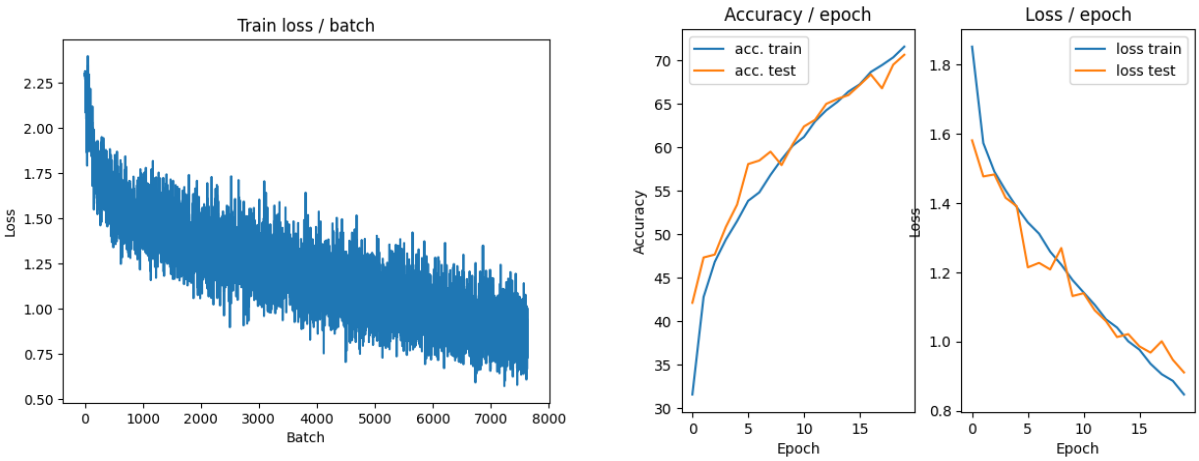


Figure 12: Results by using SGD with Momentum and Exponential Decay scheduler

The results indicate that this combination works better than using SGD with the Exponential Decay scheduler alone, and together with previous adjustments, it has almost completely resolved the overfitting issue in our model.

3.4 Regularization of the network by dropout

When a fully connected layer contains a large number of weights, it becomes susceptible to overfitting. To address this, we can add a dropout layer between two fully connected layers, which effectively reduces this risk by randomly deactivating neurons during training.

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

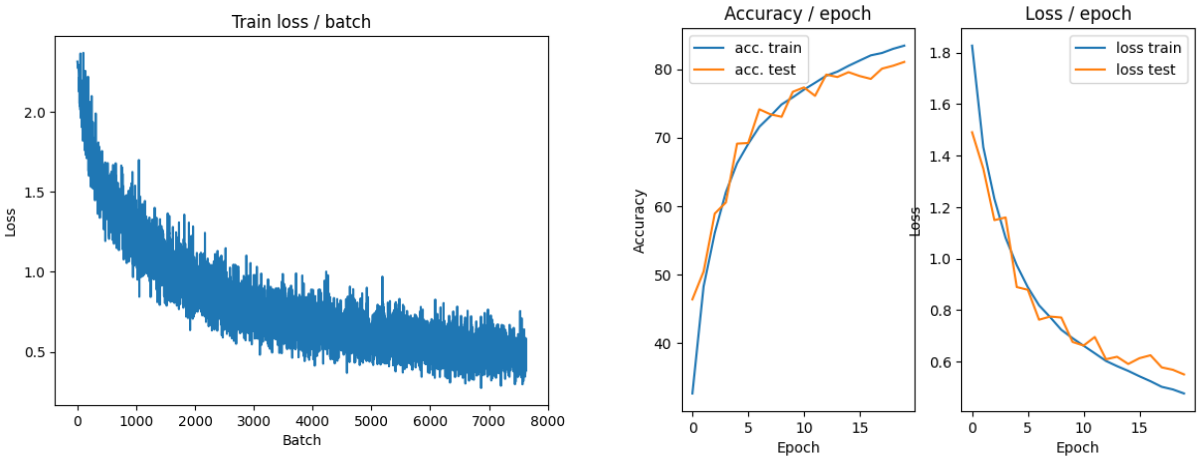


Figure 13: Results by adding a dropout layer

After adding the Dropout layer, the overall performance of the model improved. Training loss continued to decrease but showed slightly more fluctuation, which is expected due to the random neuron dropout from Dropout. Training accuracy steadily rose to around 85-88%, while test accuracy stabilized between 82-85%, indicating better generalization. Additionally, the test loss curve became more stable and closer to the training loss, showing a reduction in overfitting.

Compared to the model without Dropout, adding Dropout resulted in higher test accuracy, demonstrating improved performance on new data. The gap between test loss and training loss also decreased, reducing the risk of overfitting. Meanwhile, the fluctuation in the training loss curve increased, which is due to the random dropout of neurons; however, this fluctuation had a limited impact on the final performance, making the model overall more robust.

Q30 What is regularization in general ?

Regularization is a strategy to prevent overfitting by encouraging the model to generalize. It works by limiting the model's ability to learn very specific patterns in the training data making it learn more generalized patterns rather than overly specific details, which can lead to better performance on data it hasn't seen before.

Q31 Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it ?

When training a model with Dropout, it randomly "turns off" some neurons during each training step. This means parts of the network take a temporary "break," so the model can't rely on just certain neurons to make predictions. Instead, it must learn how to solve the problem using different combinations of active neurons. This reduces over-reliance on any single neuron, making the network more resilient and encouraging it to learn broadly useful features rather than specific details.

Q32 What is the influence of the hyperparameter of this layer ?

The dropout rate controls the fraction of neurons that are "dropped" in each layer. Higher dropout rates increase regularization, but if too high, they can lead to underfitting by removing too many neurons. Lower dropout rates provide minimal regularization and may not fully prevent overfitting.

Q33 What is the difference in behavior of the *dropout* layer between training and test ?

During training, dropout randomly deactivates neurons, while during testing, all neurons are active. To compensate for the missing neurons in training, the outputs are scaled down (typically by the dropout rate) to maintain a consistent scale between training and testing. This ensures that the model's predictions are stable when switching between training and testing modes.

3.5 Use of batch normalization

A final learning strategy that we are going to test is the batch normalization. This is a layer that learns to renormalize the outputs of the previous layer, allowing to stabilize the learning. Here we add a 2D batch normalization layer immediately after each convolution layer.

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

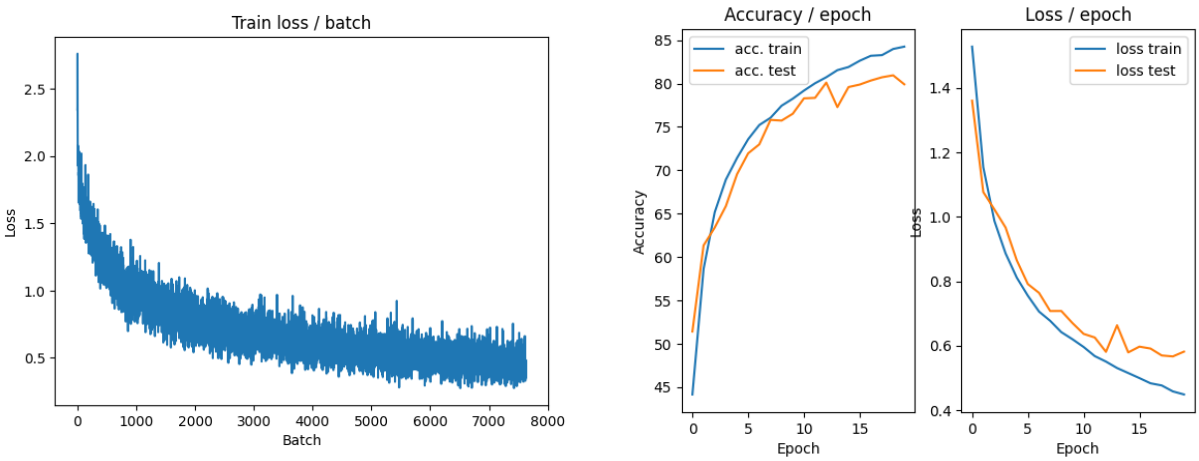


Figure 14: Results by adding batch normalisation

With the addition of Batch Normalization, both training and test accuracy steadily increased, showing a smoother upward trend, with the very small gap between them. The test loss curve became more stable and closely aligned with the training loss, indicating reduced overfitting and improved generalization.

Compared to previous results, Batch Normalization not only enhanced the model's generalization ability but also accelerated the convergence speed. Overall, the model is getting more robust and reliable on unseen data.

4 Conclusion

In this practical session, we first explored the foundational concepts of convolutional neural networks (CNNs), including convolutional and pooling layers and their applications in image processing tasks. Next, we designed and trained a CNN from scratch on the CIFAR-10 dataset, creating an architecture inspired by AlexNet. Finally, we implemented several techniques to improve the model's performance, including standardization, data augmentation, learning rate decay, dropout regularization, and batch normalization.

Each of these methods has significantly improved the model's learning process, and to be precise:

- **Standardization**: This technique involves scaling pixel values by the dataset's mean and standard deviation, which conditions the data for faster convergence and more stable training.
- **Data Augmentation**: We applied random transformations like cropping and horizontal flips to artificially increase the dataset size, which reduces overfitting by exposing the model to more diverse variations of the images.
- **Learning Rate Decay**: By gradually decreasing the learning rate over time, this technique helps the model converge more smoothly and prevents overshooting the optimal parameters.
- **Dropout Regularization**: Dropout randomly deactivates neurons during training, reducing overfitting by ensuring that no single neuron overly influences the model's predictions.
- **Batch Normalization**: This method normalizes inputs within each batch, stabilizing the gradient flow and accelerating training, resulting in better generalization and learning efficiency.

Overall, these methods enhanced the model's ability to generalize to new data, underscoring the importance of data preprocessing and optimization techniques.

1-e: Transformers

In this part, we focus on the architecture of Vision Transformer model by implementing a naive version of the ViT model, experimenting on MNIST dataset, exploring the influences of different hyper parameters.

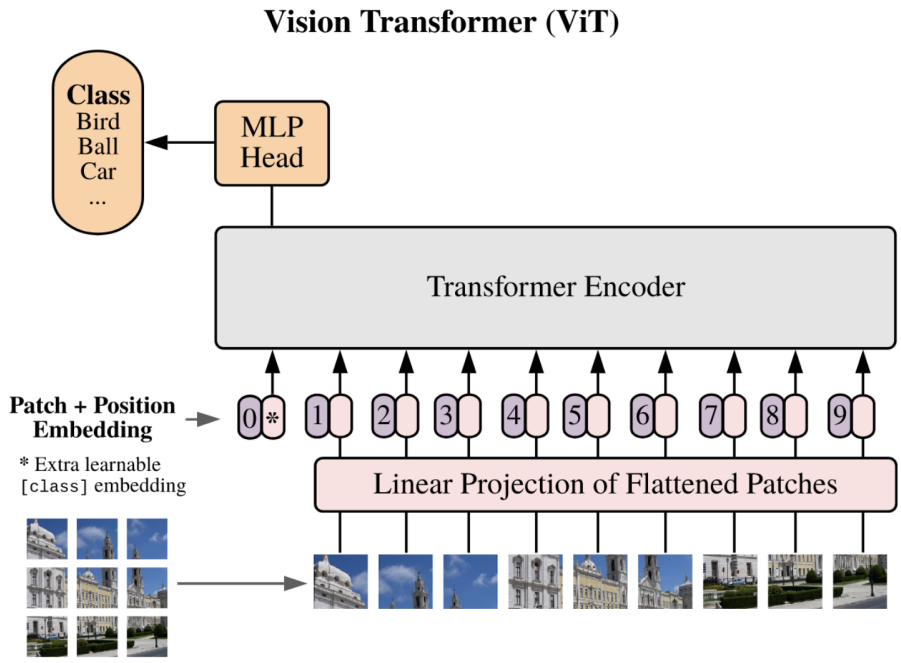


Figure 15: Vision Transformer model

It is demonstrated in the figure that an image is first split into fixed-size patches,then we linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence.

1 Basic Concepts

1.1 Self-Attention

Q1 What is the main feature of self-attention, especially compared to its convolutional counterpart. What is its main challenge in terms of computation/memory?

- The main feature of self-attention is its ability to dynamically compute the relevance between different elements in an input sequence, allowing for the adjustment of each element's representation based on these computed relationships. This enables the model to capture long-range dependencies and global contextual information across the entire sequence in a single operation. In contrast, convolutional operations typically focus on local features within a fixed receptive field and require multiple layers to achieve a similar level of global understanding.

- The main challenge of self-attention in terms of computation and memory lies in its high computational complexity, which is $O(N^2 \cdot d)$, where N is the length of the input sequence and d is the embedding dimension. This means that as the sequence length increases, the computational load grows quadratically, making it resource-intensive. Additionally, self-attention requires storing a large attention matrix of size $N \times N$, leading to significant memory consumption, especially when dealing with long sequences or high-resolution data. This can become a bottleneck during training deep models.

Q2 At first, we are going to only consider the simple case of one head. Write the equations and complete the following code. And don't forget a final linear projection at the end!

- the input X is transformed into three different representations : the Query Q, the Key K, and the Value V. This transformation is achieved through matrix multiplication with learnable weight matrices. Each vector type plays a distinct role in the attention mechanism:
  - the Query Q: Represent the current position that is looking for relevant information.
  - the Key K: Represent the information available in other positions that can be matched against the queries.
  - the Value V: Contain the actual information that will be retrieved based on the attention weights.

$$
\begin{aligned}
Q &= XW_q, \\
K &= XW_k, \\
V &= XW_v.
\end{aligned}
$$

- The attention scores are computed by taking the dot product of the query vectors with the key vectors,QK^T. This results in a matrix that quantifies the similarity between each query and all keys.

- The scores are then scaled by dividing by$\sqrt{d_k}$ (where d_k is the dimensionality of the query and key vectors) to prevent the dot products from becoming too large, which helps stabilize the gradients during training.

- The softmax function is applied to the scaled scores to convert them into a probability distribution, ensuring that the attention weights sum to one. This step highlights the most relevant values based on the calculated attention scores.

- Finally, the attention weights are used to compute a weighted sum of the value vectors V. This results in the output of the self-attention mechanism, where each position in the sequence contributes to the representation based on its relevance to the query.

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

- Optionally, it's possible to do a last final linear projection at the then, in this case we have

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

1.2 Multihead-Attention

Q3 Write the equations and complete the following code to build a Multi-Heads Self-Attention.

- To concatenate multiple attention heads, given h heads, we compute the attention output for each head and concatenate them:

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$, W_O is the output weight matrix.

1.3 Transformer Block

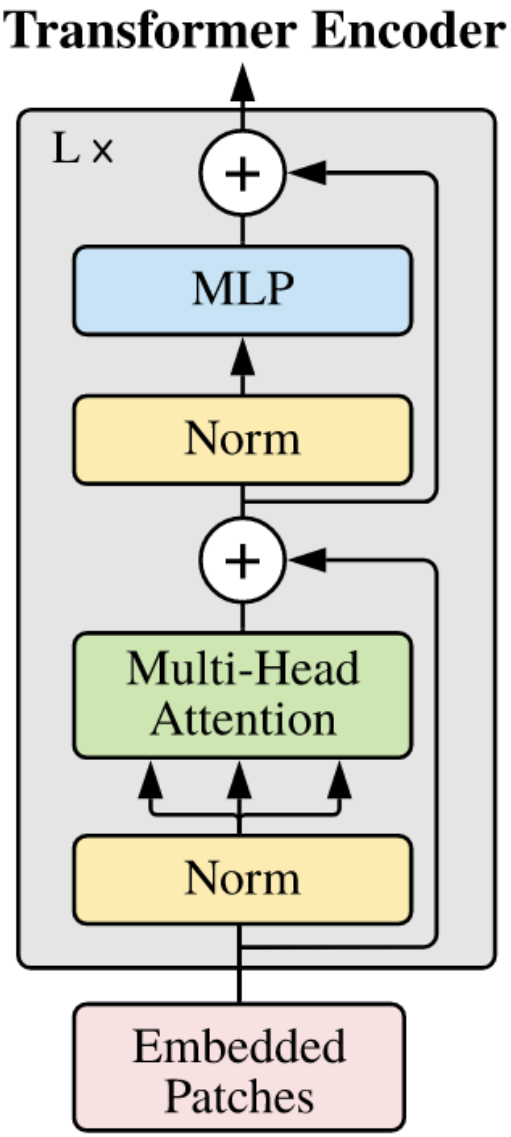


Figure 16: Transformer Encoder

Now, we need to build a Transformer Block as described in the image above.

Q4 Write the equations and complete the following code.

1. Given an input X, where N is the sequence length and D is the embedding dimension, we first normalize the input:

$$\hat{X}_1 = \text{LayerNorm}(X).$$

2. Then, we compute the multi-head self-attention:

$$Q = \hat{X}_1 W_q,$$
$$K = \hat{X}_1 W_k,$$
$$V = \hat{X}_1 W_v.$$

$$\text{Attention} = \text{MultiHeadAttention}(Q, K, V).$$

3. We add a residual connection:

$$X_{\text{att}} = X + \text{Attention}.$$

4. we normalize the result again:

$$\hat{X}_2 = \text{LayerNorm}(X_{\text{att}}).$$

5. The MLP consists of two linear transformations with a GELU activation function in between:

$$\text{MLP}(\hat{X}_2) = \text{GELU}(\hat{X}_2 W_1 + b_1) W_2 + b_2.$$

where $W_1 \in \mathbb{R}^{D \times \text{mlp\_ratio} \cdot D}$ and $W_2 \in \mathbb{R}^{\text{mlp\_ratio} \cdot D \times D}$.

6. We add another residual connection:

$$X_{\text{out}} = X_{\text{att}} + \text{MLP}(\hat{X}_2).$$

Thus, the final output of the block is given by:

$$\text{Output} = X_{\text{out}}.$$

1.4 Full ViT model

Now we need to build a ViT model based on what are coded in the previous questions. There are additional components that should be coded such as the Class token, Positional embedding and the classification head.

Q5 Explain what is a Class token and why we use it?

- The class token acts as an extra token that aggregates information from all patches during the self-attention process.
- **Purpose:** The output corresponding to the class token is used as the final representation for classification tasks. It gathers information from all the other patches through the self-attention mechanism, enabling the model to predict the class label based on the holistic understanding of the entire input image.
- **Why Use It?** The class token provides a mechanism for the model to produce a fixed-size output for classification regardless of the input sequence length. This approach is similar to how BERT uses a [CLS] token for natural language processing tasks.

Q6 Explain what is the the positional embedding (PE) and why it is important?

- Positional embeddings are used to provide information about the position of each patch within the sequence and are added to the patch embeddings before being fed into the transformer layers.
- **Purpose:** The purpose of the positional embedding is to inject information about the spatial arrangement of patches in the original image, allowing the model to understand the relative positions of different patches.
- **Why is it Important?** In images, the arrangement of patches is crucial for recognizing shapes, textures, and objects. Without positional embeddings, the transformer would treat the input patches as a "bag of patches" without any notion of spatial arrangement, leading to suboptimal performance in vision tasks.

Here we use sinusoidal encoding, it uses fixed sinusoidal functions of different frequencies to generate positional embeddings. The advantage is that it does not need to be learned and can generalize to longer sequences than seen during training.

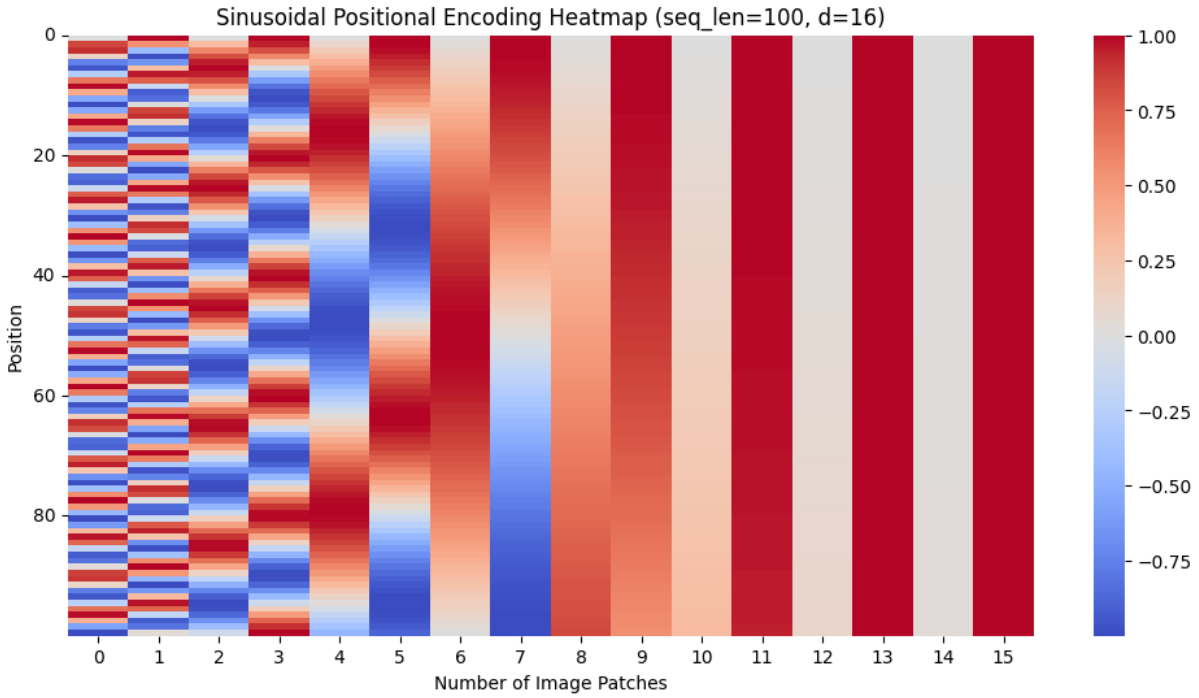


Figure 17: sinusoidal positional encoding with $seqlen$=100, d=16

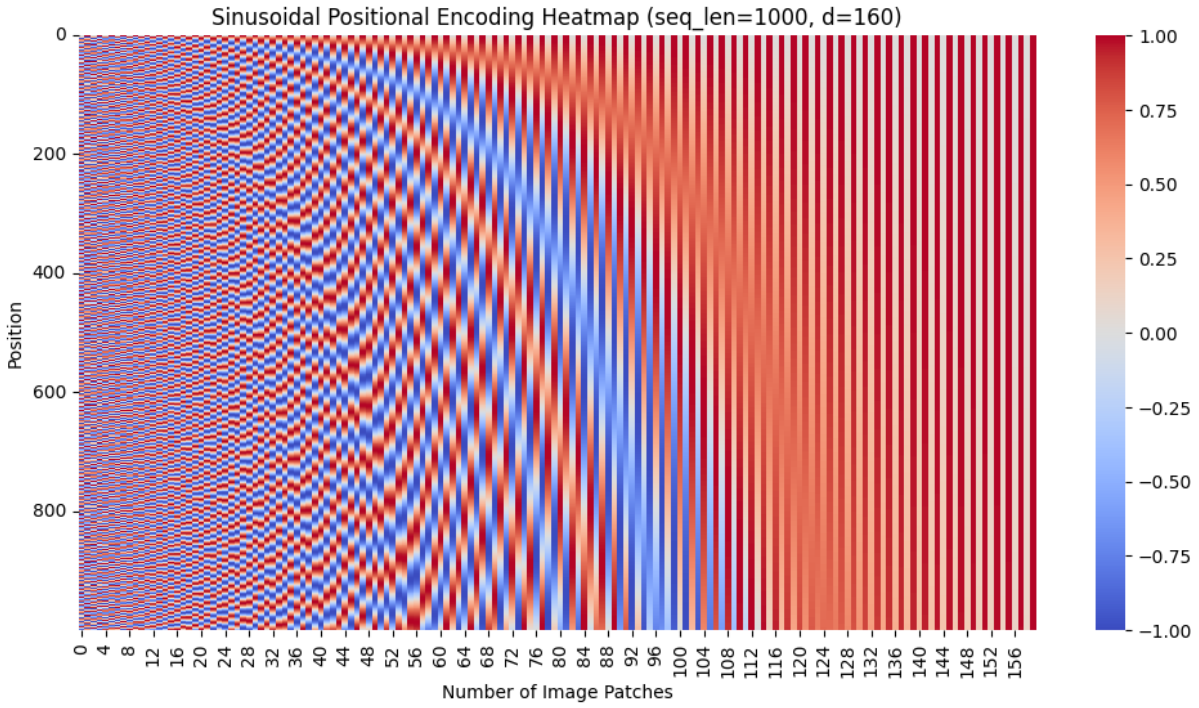


Figure 18: sinusoidal positional encoding with $seqlen$=1000, d=160

Figure 17 and Figure 18 are employed to visualize the appearance of sinusoidal encoding. In the context of an image transformer, the x-axis represents the number of image patches created, while the y-axis represents the size of an image patch embedding.

2 Experiment on MNIST

Q7 Test different hyperparameters and explain how they affect the performance. In particular embed_dim, patch_size, and nb_blocks.

In this section we analyze the influences of different hyperparameters on the performance of transformer model, and we test it on MNIST dataset.

2.1 Influences of embed dimension

First, we take a look at the influences of embed dimension. In the experiment we test the performances of the model with embed dimension at [16,32,64,128].

In the context of a ViT model, the embed dim (embedding dimension) plays a crucial role in determining the model's capacity to represent and learn features from the input data.

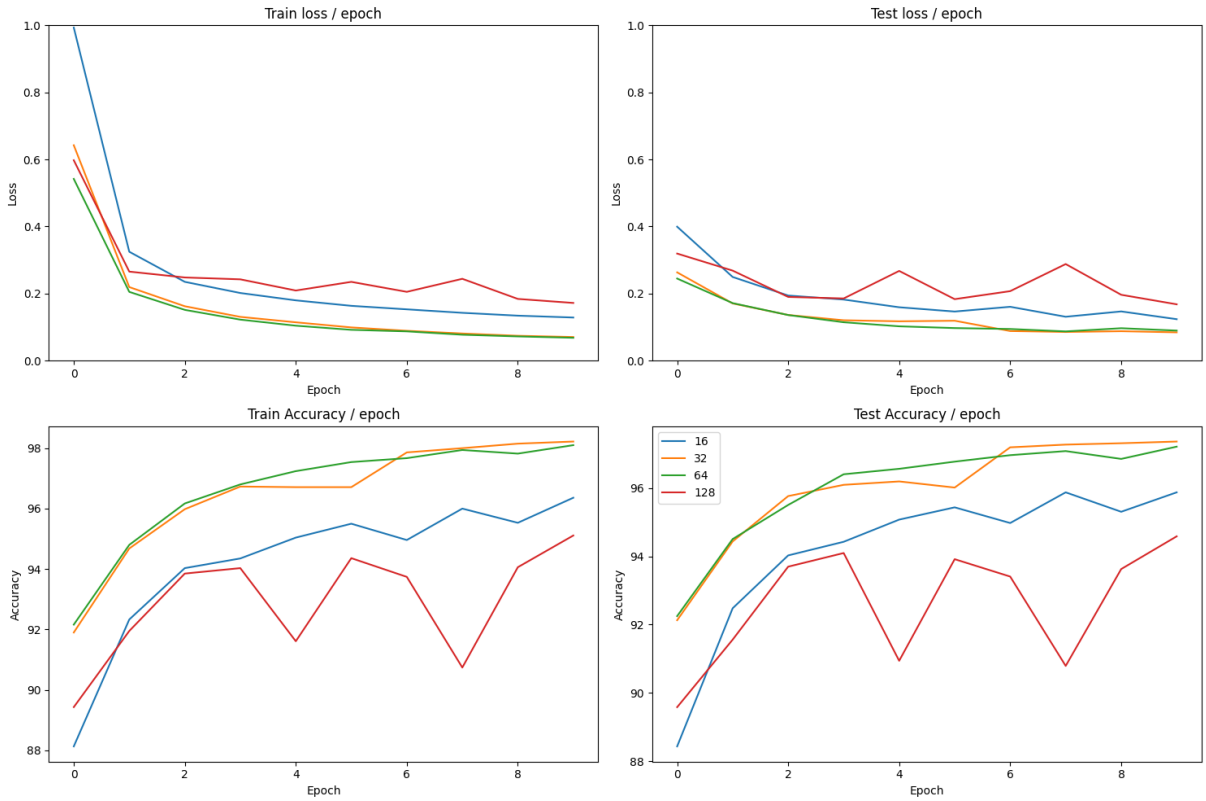


Figure 19: Influences of embed dimension

From Figure 19, we can see that the embed dimension of 16 converges slightly slower and the performance of embed dimension at 128 is worse than the others. And due to the increase of parameters, the embed dimension at 128 requires more training time as well. Meanwhile the embed dimension of 32 has the best performance with lower losses and higher accuracies in both training and testing.

2.2 Influences of patch size

Q8 What is the complexity of the transformer in terms of number of tokens? How you can improve it?

We now examine the influences of patch sizes. We will also answer the question 8 since the number of tokens is closely related to the patch size.

Patch size by definition determines the size of each patch, and since the size of MNIST image is fixed with 28×28, meaning that with the increase of patch size, each patch covers a larger size of the image therefore we will have less patches and smaller number of tokens.

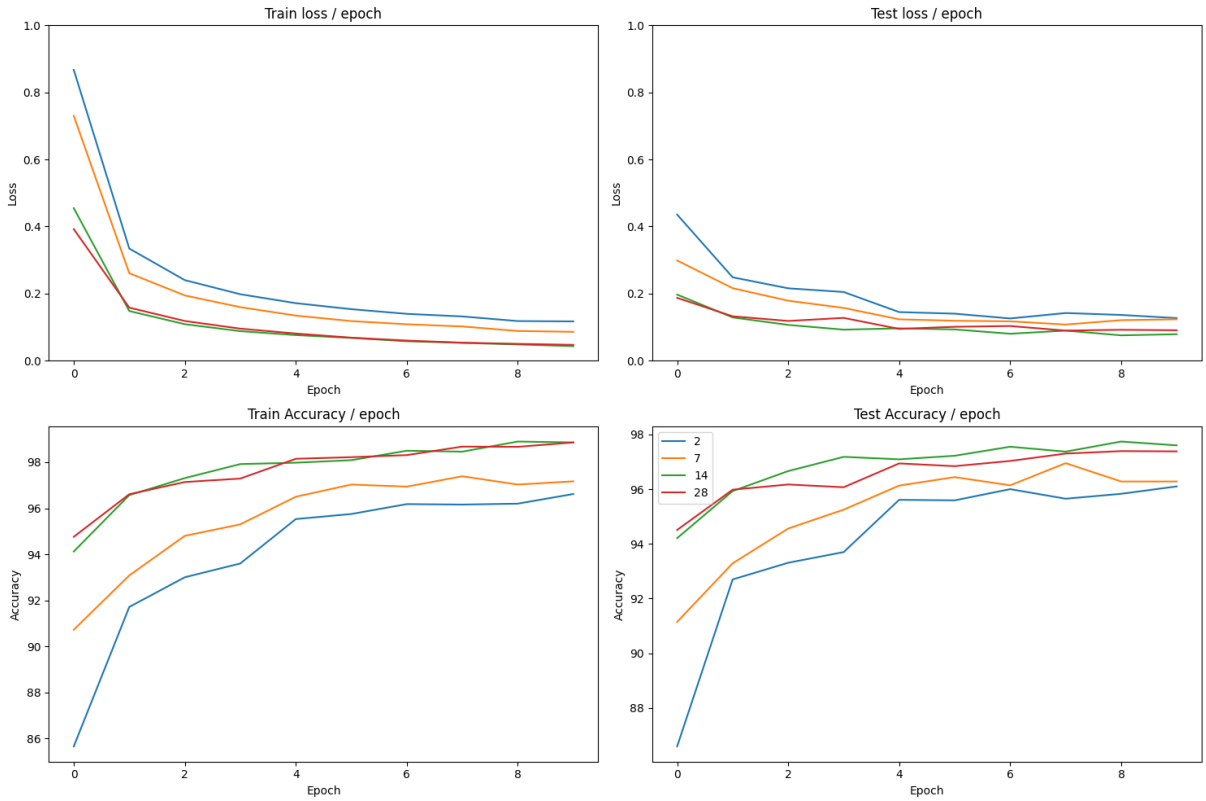


Figure 20: Influences of different patch sizes

From Figure 20, we can see that when the patch size is 2×2 pixels, the model has problem with the learning. And as for the patch size of 28×28, this is exactly the size of a MNIST image and we can consider it as no patch formed. We can conclude that when the patch size is set to 14×14, the results are ideal with more stable convergence and slightly lower losses and higher accuracies in the end.

2.3 Influences of number of blocks

Finally we focus on the influences of number of blocks. And in this section we experiment on different number of blocks on [2,4,6,8].

This parameter directly affects the depth of the model. More blocks mean a deeper architecture, which can capture more complex patterns and relationships in the data.

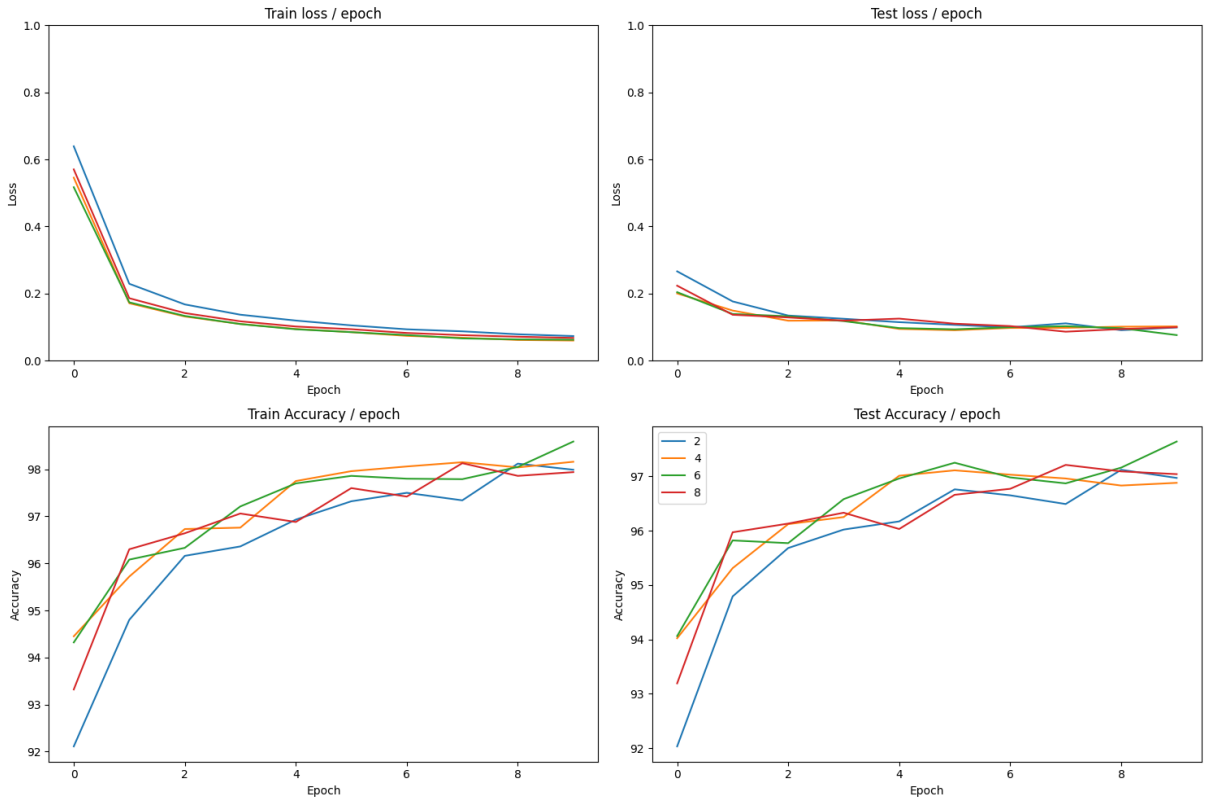


Figure 21: Influences of number of blocks

We can see from Figure 21 that when there are only 2 blocks, it seems to higher initial loss and lower initial accuracy but converges to a relatively good result. Meanwhile, we can also conclude that the higher number of blocks may result in instability as demonstrated when there are 8 blocks. And we see that the one with best performance is that of 6 blocks.

3 Larger Transformers

In this section we try to build model with bigger transformer and utilizes the timm library.

3.1 Questions

Q9 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? Explain if we have also such problem with CNNs. As ViT takes RGB images, the input tensor should have 3 channels.

- **The problem:** The `vit_base_patch16_224` are typically pre-configured to work with higher image resolutions 224x224. When using a ViT model without pretrained weights, the input tensor is still expected to have the corresponding dimensions. When we try to use a smaller resolution like 28x28 for MNIST images , the ViT

model will not work as intended because the patch embedding layer and positional encoding might not be compatible with this input size.

- **The case with CNNs: Convolutional Neural Networks (CNNs)** are more flexible when it comes to input resolution because they operate in a fully convolutional manner. The convolutional layers can process different input sizes without requiring a specific image resolution. The only constraint in a CNN occurs in the fully connected layers, where the size of the input feature map needs to match the expected dimensions. In contrast, the ViT model has a strict requirement for the number and size of patches due to the self-attention mechanism, which is why using the wrong resolution leads to issues.

Q10 Provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work.

1. **DINO (Data-efficient Image Transformer):** By digging down the paper of "Emerging Properties in Self-Supervised Vision Transformers " we see how the DINO model applies self-supervised learning with Vision Transformers (ViTs) to achieve better performance on small datasets by using a self-distillation approach without labels.

   (a) **Self-Supervised Pretraining for ViTs:** Traditional ViT approaches rely on large-scale, supervised pretraining, but DINO adopts a self-supervised strategy inspired by NLP tasks like BERT's masked language modeling. The model learns by creating pretext tasks that use image transformations, rather than relying on predefined labels.

   (b) **Knowledge Distillation Framework**: DINO employs a knowledge distillation approach, where a student network learns to predict the output of a teacher network. The teacher network is built using a momentum encoder (an exponential moving average of the student parameters), which provides a high-quality, dynamic target for the student to match. This eliminates the need for a fixed pretrained teacher, making the model adaptable to different data scenarios.

   (c) **Efficient Training with Limited Resources**: DINO demonstrates competitive performance on benchmarks like ImageNet using only moderate computing resources (two 8-GPU servers over three days), which is significantly more efficient than some other self-supervised approaches. The framework's simplicity and flexibility allow it to be applied to both ViTs and conventional convolutional networks without requiring architecture changes.

   Overall, DINO combines the strengths of self-supervised learning, knowledge distillation, and multi-crop training strategies to make ViTs more effective on smaller datasets, providing a versatile and resource-efficient approach.

2. **SPT(Shifted Patch Tokenization)** and **LSA (Locality Self-Attention) :** By exploring in the paper of "Vision Transformer for Small-Size Datasets" , we found two methods to improve the ViT for small datasets:

(a) **SPT (Shifted Patch Tokenization)** : SPT aims to enhance the locality inductive bias of ViTs by incorporating more spatial information into the visual tokens. This is achieved by spatially shifting the input image in multiple directions (e.g., four diagonal directions) and then concatenating these shifted versions with the original image. By increasing the receptive field of tokenization (the amount of spatial information each token covers), SPT helps the ViT learn better local structures in the data.

(b) **LSA (Locality Self-Attention)** : LSA aims to make the self-attention mechanism of ViTs more locally focused by sharpening the attention score distribution. It uses **diagonal masking** that removes self-token relations by masking the diagonal elements of the similarity matrix (computed from the Query and Key), thereby emphasizing the relationships between different tokens. And it also uses the **adjusted softmax temperature** during training to sharpen the attention distribution. This helps mitigate the smoothing effect that standard softmax has on attention scores. By focusing more on inter-token relationships and adjusting the temperature, LSA increases the model's ability to learn local features, improving its performance on smaller datasets.

Together, these two methods aim to boost the locality inductive bias of ViTs, making them more effective for visual tasks on small datasets by enhancing their capacity to learn local patterns.

3.2 Experiments

Learning from scratch

We tried the model with `pretrained=False`, which turned out to be a bad idea, because the learning process takes too long as it took nearly one hour to give results of all epochs, and the accuracies starts from aroung 30 percent.

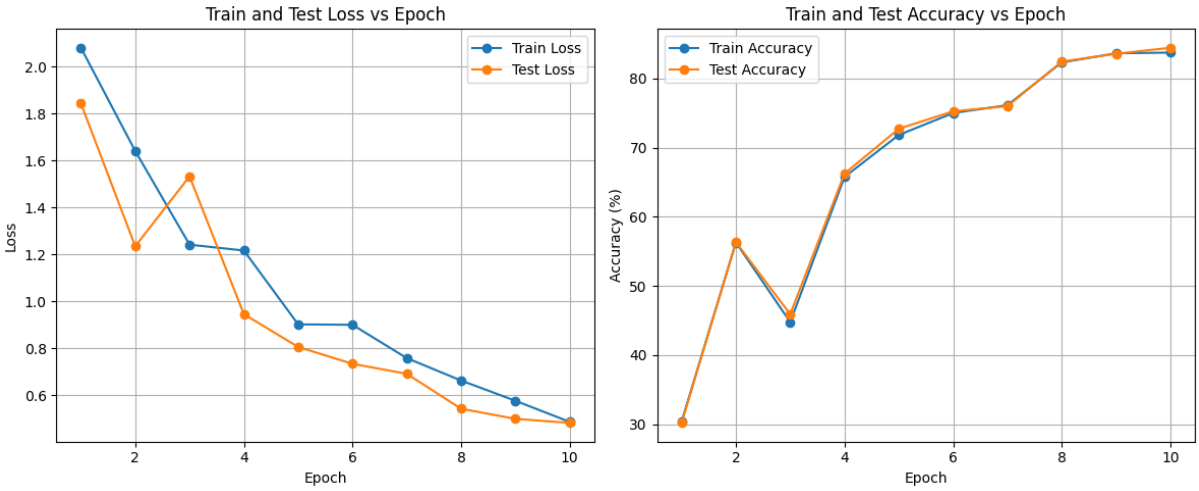


Figure 22: Learning from scratch

From the Figure 22 above we can see that the model starts from a high loss and very low accuracy and the results at the end are not ideal either. So this may be attributed to

the limited dataset. And the learning process takes up too much space as well as time, which is not efficient in any way.

Learning with model pretrained on ImageNet

Conducting the experiments in the colab, we actually explored 2 approaches. Since the size of MNIST image are not compatible with the ViT model with higher resolutions. So we either transform the MNIST datasets to make it compatible with the model or the other way around. And by using the "trick" in timm library, we came to the conclusion that the latter where we use the `pretrained_cfg_overlay` is the better approach that learns much quicker.

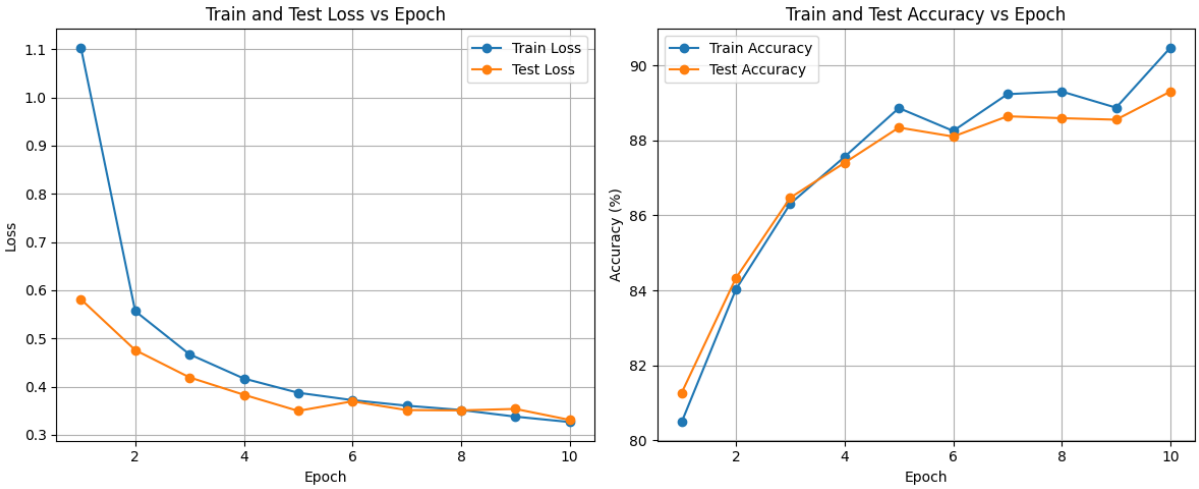


Figure 23: model pretrained

Compared to Figure 22, we can see from above that the Figure 23 shows a better result with the model pretrained on ImageNet. Compared to learning from scratch, the pretrained model starts with lower losses and higher accuracies and achieves better results.

Learning with smaller pretrained model

Finally we tried a smaller model pretrained on ImageNet, `vit_small_patch16_224`

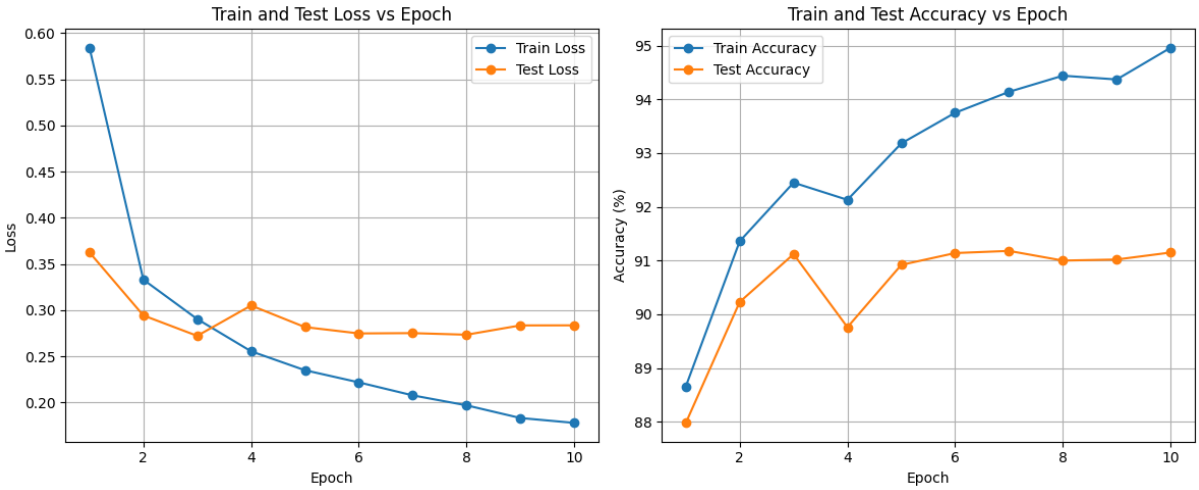


Figure 24: smaller model pretrained

In this smaller model, we obtain a even better result with much lower loss and higher accuracies. But the instability shown on the test accuracy and the insignificancy of the changes on test loss indicate that there is still space to improve the learning process of the model on smaller datasets as given in the answers of **Question 10**.

4 Conclusion

It is quite challenging to fine-tune the ViT model to obtain the optimal results.We need to choose proper patch size and number of blocks while avoiding overly large embedding s=dimensions. And from the experimental results with MNIST dataset, we can derive that the pretrained model evidently demonstrates better results and performs even better with smaller ViTs. We can also improve them with self-supervised learning approaches like **DINO** (Self-Distillation with No Labels) or **MAE** (Masked Autoencoder) and methods to boost capacity to learn local patterns like **SPT(Shifted Patch Tokenization) and LSA (Locality Self-Attention) .** We can also consider introducing additional regularization techniques to reduce overfitting on the small dataset, examples including Dropout and Stochastic Depth (that randomly drop layers during training, improving generalization.).