Feb 2023

Faculté des Sciences et Ingénierie - Sorbonne Université

RDFIA Lab Report

Section 1 : Basics on deep learning for vision

Yuxuan XI **Mojan Omidi**

Lab Supervisor : Dr. Alasdair Newson

University year :
2023-2024

Contents

List of Figures

Chapter 1

Introduction to Neural Networks

1.1 Theoretical foundation

In this part, our objective is to provide as complete and comprehensive answers as we can to address all the questions found in the document.

1.1.1 Supervised dataset

→ ⋆ **Question 1:** What are the train, val, and test sets used for?

— We normally break down the dataset into three distinct sets: Training set - Validation set - Test set.

Training set: Usually, the training set comprises the majority, often ranging from 60% to 80%, of the dataset. In each epoch, the model systematically processes and learns from every training example within this set. It undergoes repeated training cycles (for a predetermined number of epochs), continuously refining its understanding of the underlying patterns in the training data. The ultimate goal is to nurture a model that can make precise predictions when presented with previously unseen data.

Validation set: The validation set is a distinct dataset, kept apart from the training set, and serves a crucial role in model development. It acts as a checkpoint, providing valuable insights during the training stage. This validation process plays a vital role in fine-tuning hyperparameters such as learning rate, batch size, number of epochs, dropout rate, network architecture, and more. Typically, this validation process occurs after each training epoch, enabling us to assess how effectively the model generalizes to unseen data as it undergoes training.

Test set: A test set is a dataset reserved exclusively for assessing the model's performance. It's selected randomly to resemble real-world data, ensuring reliable results.

Key takeaways:

- The model's weights are updated exclusively using the training data,
- The test set is held back for a final, impartial assessment of the model's generalization performance.

- Adjusting weights based on validation set errors could result in overfitting to the validation data and render the model unreliable
- It's essential to remember that the validation set should not be used to update the model's parameters. Its purpose is for model selection and hyperparameter tuning.

→ **Question 2:** What is the influence of the number of examples N ?

— The number of examples (N) significantly influences machine learning. Increasing the dataset size can have both beneficial and detrimental effects, so it should be done carefully. A larger dataset generally improves model performance, enhances feature learning, and stabilizes the model, reducing overfitting. However, if new samples are of low quality or disrupt the dataset's balance, it can degrade performance, potentially leading to misclassification in classification tasks. Additionally, increasing dataset size may demand higher computing capacity, potentially causing time or resource constraints. Therefore, expanding the dataset size has both advantages and drawbacks, depending on various factors.

1.1.2 Network Architecture

→ **Question 3:** Why is it important to add activation functions between linear transformations?

— Activation functions find their roots in the biological behavior of neurons, where distinct neurons "fire" in response to different stimuli. In the brain's neural networks, neurons exhibit binary behavior—either active (1) or inactive (0). Artificial neural networks incorporate activation functions to emulate this behavior, determining whether a neuron should activate.

The pivotal role of activation functions lies in their capacity to inject non-linearity into the network's transformations. Without them, the network would be a series of linear transformations, resembling stacked linear regression models. This setup severely limits the network's ability to grasp intricate patterns.

Hence, after each layer in a neural network, we apply an activation function to introduce non-linearity into the computations. This non-linearity empowers the network to handle complex tasks and uncover intricate relationships in data, enhancing its learning and generalization capabilities.

Linear classifiers cannot solve linearly inseparable problems. Multiple perceptrons are still a linear classifier. Using activation functions, the output results are nonlinear, which can solve more complex problems.

→ ⋆ **Question 4:** What are the sizes nx, nh, ny in the figure 1? In practice, how are these sizes chosen?

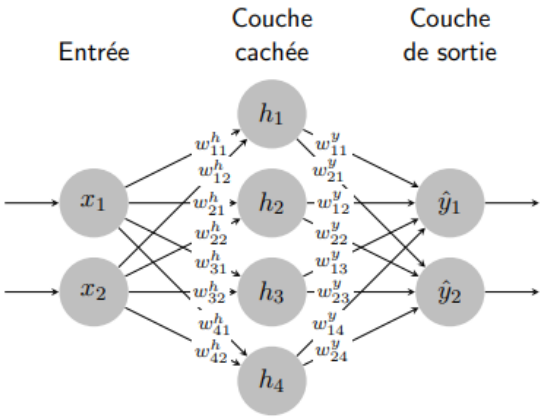

Figure 1.1 – A simple perceptron

— nx **= 2**
nh **= 4**
ny **= 2**

Input layer (nx**)**: The size "nx" of the input layer corresponds to the number of features considered for each sample. When dealing with images, the input size aligns with the number of pixels, which represents the dimensionality of the data. For instance, in the case of a grayscale image measuring 256x256, the input data size would be 65,536, as each pixel is treated as an individual feature.

Hidden layer (nh**)**: In practice, there is no one-size-fits-all choice for the ideal number of neurons in hidden layers. This hyperparameter must be carefully adjusted according to the unique attributes of your problem and how well your model is performing. Moreover, for more intricate tasks, you may discover the need to employ deeper networks with hidden layers that have different neuron counts to achieve optimal outcomes.

Output layer (ny**)**: The choice of the output layer size is determined by the nature of your problem and the desired output:

- In binary classification, a single neuron is employed. Its output serves as the probability indicating membership in one of the two classes.
- In multiclass classification, the number of neurons in the output layer should match the number of classes. Each neuron corresponds to a class and represents the associated probability or score.
- In regression tasks, a solitary neuron is utilized. The output from this neuron directly signifies the predicted continuous value.

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

— y is called the ground truth or the actual target values in a supervised learning problem. These values are the accurate or genuine results that you aim for your neural network to predict and learn from.

— $\hat{y}$ stands for the neural network's generated prediction. The goal is to make $\hat{y}$ as accurate as possible in approximating y by training the neural network to minimize the difference between them.

→ **Question 6:** Why use a SoftMax function as the output activation function?

— The softmax function is an extension of the logistic function, designed to transform values into a predefined range. In the neural network's final layer, it produces activations, often referred to as logits, that we intend to interpret as probabilities to facilitate classification.

The rationale behind employing softmax is to guarantee that these logits collectively add up to 1, adhering to the principles of a probability distribution so it is used for the output layer only (at least in most cases) to ensure that the sum of the components of the output vector is equal to 1.

Another reason why it is commonly selected for the output layer is due to its differentiability. Differentiability plays a pivotal role in training neural networks using gradient-based optimization algorithms, with backpropagation and stochastic gradient descent (SGD) being prominent examples.

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

One "hidden" layer from the input of size n_x to a vector h of size n_h, made of :

— An affine transformation using a matrix of weights W_h of size $n_h \times n_x$ and a bias vector b_h of size n_k, we note $\widetilde{h}$ the output vector of this transformation:

$$\widetilde{h} = xW_h^T + b_h$$

— The activation function $tanh$, we note h the output vector of this transformation:

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

An output layer from the hidden features of size n_h to the output vector $\hat{y}$ of size n_y, made of :

— An affine transformation using a matrix of weights W_y of size $n_y \times n_h$ and a bias vector by of size n_y, we note $\widetilde{y}$ the output vector of this transformation:

$$\widetilde{y} = hW_y^T + b_y$$

— The activation function SoftMax, we note $\hat{y}$ the output vector of this transformation:

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

1.1.3 Loss Function

→ **Question 8:** During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}$ vary to decrease the global loss function L?

— Cross-entropy loss is usually used for classification problems and Squared Error loss is usually used for regression problems. To reduce the global loss function, the output of the model ($\hat{y}$) should be as close as possible to the true value (y).

Both Mean Square Error and Cross-Entropy aim to minimize the difference or error between predicted values and actual values by calculating the error or mismatch between predicted values $\hat{y}$ and actual target values y in some way.

→ **Question 9:** How are these functions better suited to classification or regression tasks?

— Cross-Entropy loss is ideal for classification tasks, where you work with discrete class labels represented as probabilities. It accurately measures how well predicted probabilities match actual categories.

In contrast, Mean Squared Error (MSE) is suitable for regression problems, focusing on predicting continuous values. It computes the average of squared differences between predictions and actual values.

1.1.4 Optimization algorithm

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

— **Stochastic Gradient Descent**:
In SGD, we randomly select one of the training samples at each iteration to update the weights and biases.

✓ **Pros:**
- Easier to fit into memory due to a single training sample being processed by the network.
- Computationally fast as only one sample is processed at a time.
- Converges faster for larger datasets as it causes updates to the parameters more frequently.

× **Cons:**
- Highly noisy updates, which can make convergence less stable.
- Frequent updates are computationally expensive due to using all resources for processing one training sample at a time.
- It loses the advantage of vectorized operations as it deals with only a single example at a time.

— **Online Stochastic Gradient Descent**:
In online SGD, we use the most recent sample at each iteration and there is no stochasticity.

✓ **Pros:**
- Online SGD is well-suited for scenarios where data arrives continuously or in a streaming fashion. It can update the model as new data becomes available, making it suitable for real-time or near-real-time applications.
- It can adapt to changing data distributions and concept drift by continuously updating the model with the most recent data.
- Low memory usage.
- Fast updates.

× **Cons:**
- Online SGD is highly sensitive to the learning rate hyperparameter. Poor choices of learning rate can lead to divergence or slow convergence.
- Depending on how data is streamed, there can be biases in the data selection process, which may impact the model's performance.
- Noisy updates.

— **Batch(Classic) Gradient Descent**:

Batch gradient descent is when we input the entire training dataset and then perform forward and backpropagation.

✓ **Pros:**
- Benefit from the vectorization which increases the speed of processing all training samples together.
- It is computationally efficient as all computer resources are not being used to process a single sample but rather are being used for all training samples
- efficient for large datasets
- Guaranteed convergence to global optimum

× **Cons:**
- Slow convergence
- Slower updates
- It requires memory to store the entire dataset during training, which can be a limitation for large datasets that don't fit in memory.
- Can be very slow
- Does not allow online updating of the model

— **Mini Batch Gradient Descent**:

When we input several training examples simultaneously (but not the entire training dataset), we refer to it as a mini-batch.

✓ **Pros:**
- Faster convergence than classic gradient descent
- Balanced computational cost
- efficient for large datasets
- Stable convergence due to the reduction of the variance of the parameter updates

× **Cons:**
- Choice of a good mini-batch size can be challenging

Mini Batch Gradient Descent combines aspects of both stochastic and batch gradient descent. It divides the training set into batches, processes one batch at a time, and updates the neural network's parameters based on the average loss within each batch. This approach combines advantages such as efficient memory usage, computational efficiency, and stability in gradient computation. It's the most commonly used approach in practice.

→ ⋆ **Question 11:** What is the influence of the learning rate η on learning?

— The learning rate is an important hyperparameter. It controls how quickly we adjust the weights of the neural network based on the loss gradient. Most optimization algorithms, such as SGD, involve it.

The learning rate directly affects how quickly our model can converge to the local minimum, that is, achieve the best accuracy. Generally speaking, the larger the learning rate, the faster the neural network learns. If the learning rate is too small, the network is likely to fall into a local optimum; but if it is too large and exceeds the extreme value, the loss will stop declining and oscillate repeatedly at a certain position.

Achieving the right balance often requires careful choice and experimentation to match the learning rate with the specific problem and model design.

→ ⋆ **Question 12:** Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the loss concerning the parameters, using the naive approach and the backprop algorithm.

— The complexity of calculating gradients concerning the parameters in a neural network using both the naive approach and the backpropagation algorithm both scales linearly with the number of layers.

However, backpropagation is more efficient than naive gradient calculation because it reuses intermediate values during both forward and backward passes and automates gradient calculations through the chain rule.

→ **Question 13:** What criteria must the network architecture meet to allow such an optimization procedure?

— For effective backpropagation optimization, the network architecture should have differentiable activation functions, learnable parameters, a defined loss function, proper weight initialization, regularization, suitable batch size, adaptive learning rates, and normalization techniques to stabilize training and enable effective error minimization during training.

→ **Question 14:** The function SoftMax and the loss of cross-entropy are often used together and their gradient is very simple. Show that the loss can be simplified by:

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

We know the cross-entropy loss is defined as:

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

And the SoftMax function is defined as follows:

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

The output of the SoftMax function serves as the input for the cross-entropy loss, so we can obtain:

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

And then, we can calculate them:

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

→ **Question 15:** Write the gradient of the loss (cross-entropy) relative to the intermediate output $\hat{y}$.

$$\frac{\partial \ell}{\partial \tilde{y}_i} = -y_i + \frac{\frac{\partial}{\partial \tilde{y}_i}(\sum_{j=1}^{N} e^{\tilde{y}_i})}{\sum_{j=1}^{N} e^{\tilde{y}_j}} = -y_i + \frac{e^{\tilde{y}_i}}{\sum_{j=1}^{n_y} e^{\tilde{y}_j}} = -y_i + SoftMax(\tilde{y})_i$$

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

—

$$\frac{\partial e}{\partial \tilde{y}_i} = -y_i + \frac{e^{\tilde{y}_i}}{\sum e^{y_i}} = \hat{y}_i - y_i$$

→ **Question 16:** Using 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 involves using $\nabla \tilde{y} \ell$. Do the same for $\nabla b_y \ell$.

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

First, we need to calculate $\tilde{y_k}$.

$$\begin{aligned} \tilde{y} &= h W_y^T + b^y \\ &= (h_1 \cdots h_{n_h}) * \begin{pmatrix} W_{1,1} & \cdots & W_{1,n_y} \\ \vdots & \ddots & \vdots \\ W_{n_h,1} & \cdots & W_{n_h,n_y} \end{pmatrix} + (b_1^y \cdots b_{n_y}^y) \\ &= \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^y \end{aligned}$$

So now, we can calculate $\frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$. And the calculator is as follows :

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

And then, we continue to calculate $\frac{\partial \ell}{\partial \tilde{y_k}}$. From the previous question, we have the equation as follows :

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

Finally, we can calculate $\frac{\partial \ell}{\partial W_{y,ij}}$ as following:

$$\begin{aligned} \frac{\partial \ell}{\partial W_{y,ij}} &= \sum_{k=1}^{n_h} \frac{\partial \ell}{\partial \tilde{y_k}} \frac{\partial \tilde{y_k}}{\partial W_{i,j}^y} \\ &= (\hat{y_i} - y_i) h_j \\ &= (\nabla_{W_y} \ell)_{i,j} \end{aligned}$$

And then, we calculate this gradient involves using $\nabla_{\tilde{y}} \ell$.

$$
\begin{aligned}
\nabla_{W_y}\ell &= \begin{pmatrix} \frac{\partial \ell}{\partial W_{y,11}} & \cdots & \frac{\partial \ell}{\partial W_{y,1n_h}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \ell}{\partial W_{y,n_y 1}} & \cdots & \frac{\partial \ell}{\partial W_{y,n_y n_h}} \end{pmatrix} \\
&= \begin{pmatrix} (\hat{y_1} - y_1)h_1 & \cdots & (\hat{y_1} - y_1)h_{n_h} \\ \vdots & \ddots & n\vdots \\ (\hat{y_{n_y}} - y_{n_y})h_1 & \cdots & (\hat{y_{n_y}} - y_{n_y})h_{n_h} \end{pmatrix} \\
&= \begin{pmatrix} (\hat{y_1} - y_1) \\ \vdots \\ (\hat{y_{n_y}} - y_{n_y}) \end{pmatrix} \begin{pmatrix} h_1 & \cdots & h_{n_h} \end{pmatrix} \\
&= \nabla_{\tilde{y}}\ell
\end{aligned}
$$

→ ⋆ **Question 17:** Compute other gradients: $\nabla_{\hat{h}}\ell$, $\nabla_{W_h}\ell$, $\nabla_{b_h}\ell$.

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

And at first, we calculate $\frac{\partial h_k}{\partial \tilde{h_i}}$:

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

And we have $\tilde{y} = \sum_{j=1}^{n_h} W_{i,j}^y h_j + b_i^j$, so we can calculate $\frac{\partial \ell}{\partial \tilde{y_i}}$.

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

In the end, we calculate:

$$\frac{\partial \ell}{\partial \tilde{h_i}} = \sum_k \frac{\partial \ell}{\partial h_k}\frac{\partial h_k}{\partial \tilde{h_i}} = \sum_j W_{j,i}^y(\hat{y_j} - y_j)(1 - \tan^2(\tilde{h_i}))$$

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

We calculate $\frac{\partial \tilde{h}_k}{\partial W_{i,j}^h}$, and we have $\tilde{h_k} = \sum_{j=1}^{n_x} W_{k,j}^h x_j + b_k^h$.

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

$$\frac{\partial \ell}{\partial W_{h,i}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W^h_{i,j}} = \sum_j W^y_{j,i}(\hat{y}_j - y_j)(1 - \tan^2(\tilde{h}_i))x_j$$

$$\frac{\partial \ell}{\partial b^j_i} = \sum_k \frac{\partial \ell}{\partial \tilde{h_k}} \frac{\partial \tilde{h_k}}{\partial b^h_i}$$

$$\frac{\partial \tilde{h_k}}{\partial b^h_i} = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial \ell}{\partial b^j_i} = \sum_k \frac{\partial \ell}{\partial \tilde{h_k}} \frac{\partial \tilde{h_k}}{\partial b^h_i} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W^h_{i,j}} = \sum_j W^y_{j,i}(\hat{y}_j - y_j)(1 - \tan^2(\tilde{h}_i))$$

1.2 Implementation

In this part, all codes and corresponding explanations, as well as understanding and conclusions are in the **1-ab_Yuxuan_Mojan.ipynb** file.

1.2.1 Forward and backward manuals

$init\_params(nx, nh, ny)$ initializes the weights of a network Wh, Wy, bh, by,by entering the sizes nx, nh, ny.
I use $torch.trandn() \times 0.3$ to represent the normal distribution of mean 0 and standard deviation 0.3.

$forward(params, X)$: The size of enter X is $n_{batch} \times n_x$.
In this part, we enter X, and in the hidden layer, we get $htilde$ the output vector of the affine transformation, and by using the action function $tanh$, we get h the output vector. And then, in the output layer, we get $htilde$ the output vector of the affine transformation, and by using the action function $Softmax$, we get $yhat$ the output vector. So in the dictionnay $outputs$, we have X, $htilde$, h, $ytilde$,$yhat$.

$loss_accuracy(Yhat, Y)$: We got $yhat$ by a forward pass, and we can use the loss function to calculate the loss L.
To obtain classification accuracy, at first, we use $torch.max(Y, 1)$ to obtain the category index corresponding to the maximum probability of each sample $indesY$ and the prediction $indesYhat$. We used the function $Softmax$ before, which can obtain predicted probabilities for each category by the neural network. Then, we calculate the accuracy, the proportion of the number of the actual label $indesY$ and the predicted label $indesYhat$ equal to the total number.
The loss function is used to optimize the parameters of the model during training, while the accuracy tells us how accurate the model is in the classification problem.

$backward(params, outputs, Y)$: Here we computationally perform backpropagation of the neural network, calculating the gradients of the individual weight and bias parameters. So in the dictionnay $grads$, we have Wy, Wh, by, bh, and we add $Ytilde$ and $hitlde$ for the convenience of calculation.

$sgd(params, grads, eta)$: We implement the Stochastic Gradient Descent (SGD) optimization algorithm to update the parameters of the neural network Wy, Wh, by, bh.
The SGD algorithm implements iterative updating of parameters so that the loss function gradually decreases. The learning rate eta controls the step size of each parameter update and usually needs to be adjusted to obtain the best training results.

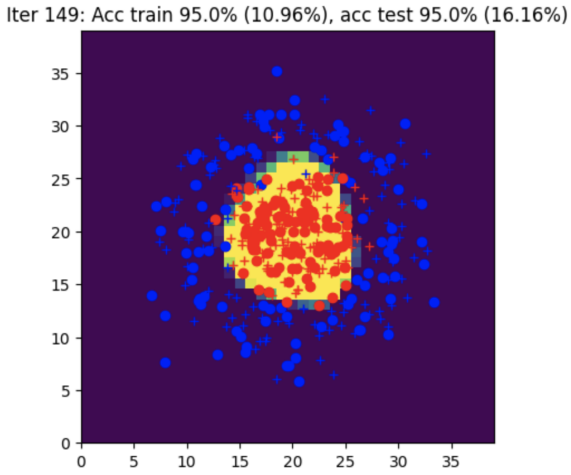

Figure 1.2

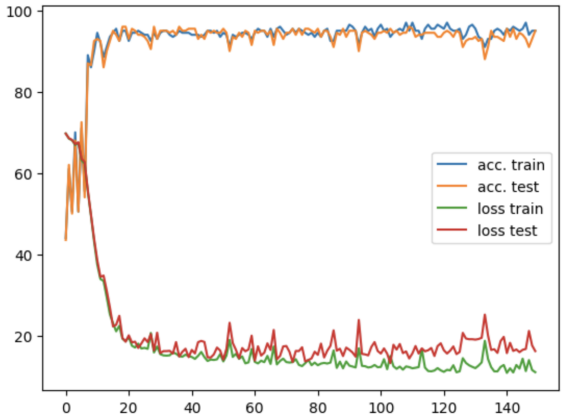

Figure 1.3

We perform the complete neural network training and visualization process record training, and test accuracy and loss values at each iteration.
150 iterations were conducted. In each iteration, the training data was randomly selected, the data were processed in batches, forward propagation and backward propagation were performed, and the parameters were updated using the sgd function. Finally, we get the accuracy and loss on the training set and test set.
On the training set, the main purpose of the model is to minimize the loss function. We usually do not pay attention to accuracy on the training set, because the model may overfit on the training set, resulting in high accuracy but poor performance on unknown data. On the test set, we focus on accuracy and evaluate the model's performance in real applications.
From the data images, we can see that as the number of iterations increases, the color

coverage becomes more and more concentrated on red data, classifying red data and blue data.
Judging from the image, for the accuracy curve, the train set and the test set are basically consistent, but the loss function on the test set is higher than the loss function. Moreover, after 40-60 iterations, the curve flattens out and the accuracy reaches about 95%, the loss value is about 12%.

1.2.2 Simplification of the backward pass with torch.autograd

$init\_params(nx, nh, ny)$: Adding the $requires\_grad = True$ condition when initializing the parameters, tells pyTorch that it needs to calculate the gradient of the tensor. Automatically calculate the gradients of parameters during training, allowing for backpropagation and optimization.

The functions $forward$ and $backward$ don't need to be modified with the method $torch.autograd$, we don't need to use the function $backward$ in the above method, we only need to add $L.backward()$ after we have the loss value, and it computes the gradients directly.
This method performs backpropagation on the loss function, calculates the gradient of each parameter to the loss function, and stores it.

$torch.no\_grad()$ is used to control the following, which means that computed gradients will not be tracked in the code below. Because during gradient update, we just want to update the parameters rather than calculate new gradients.
$grad.zero\_()$ clears the gradients for these parameters. Ensure that old information is cleared before the next gradient calculation to avoid interfering with subsequent gradient calculations.

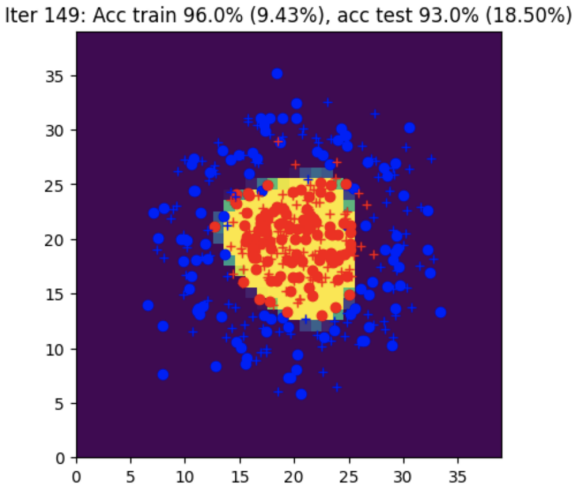

Figure 1.4

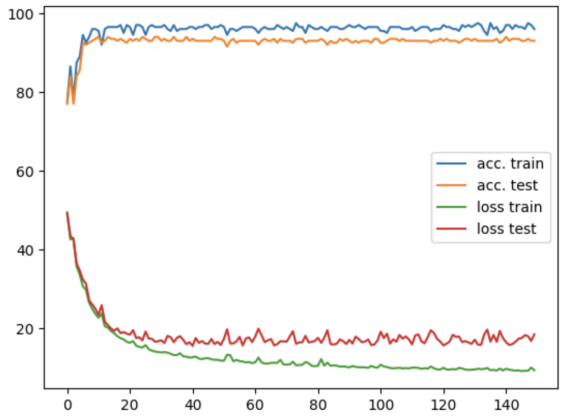

Figure 1.5

The accuracy of the training set is higher than the accuracy of the test set, but for the loss equation, the test set is higher, but there are not much difference.
For the accuracy, after about 20 iterations, it levels out. The training set is about 97%, and the test set is about 96%. However, as the number of iterations increases, the value of the loss equation gradually decreases, and the gap between the training set and the test set becomes larger and larger, causing the problem of overfitting. At 150 iterations, the training set is about 9%, and the test set is about 18%.

1.2.3 Simplification of the forward pass with torch.nnn layers

$torch.nn.Sequential$ creates a sequential neural network model. This model consists of a series of layers, including an input linear layer, a hidden layer with a tanh activation function, and a linear output layer to which $Softmax()$ is applied. At the end, we apply $torch.nn.CrossEntropyLoss()$ to represent the loss function.

The method of calculating accuracy is the same as before. For the loss function, we call $loss = torch.nn.CrossEntropyLoss()$, and then, we calculate by $loss(Yhat, Y)$.

Apply the gradient descent method to traverse the parameters of the model and update them. Repeatedly during the iterative process, the parameters of the model are gradually optimized to reduce the value of the loss function and improve model performance. At the same time, the gradient information is cleared after each iteration to prepare for the next round of backpropagation.

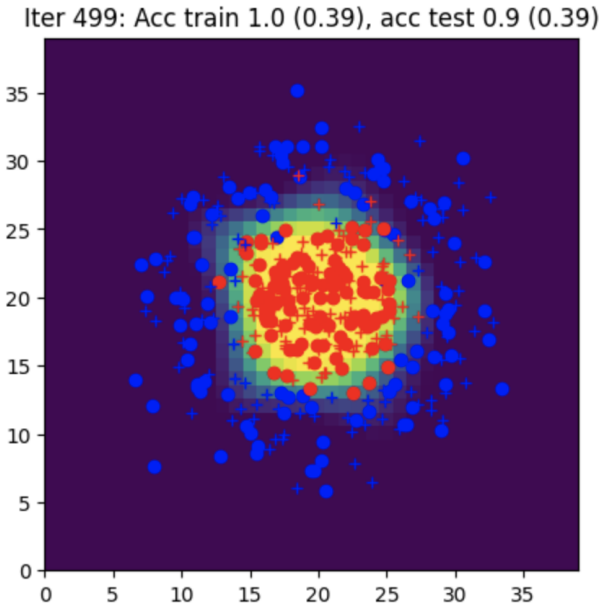

Figure 1.6

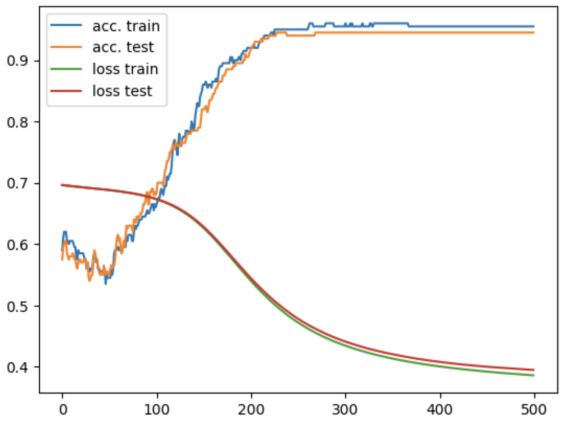

Figure 1.7

More iterations are needed, so we try 500 times of iterations.
The loss function continues to decrease, and the results of the training set and the test set are consistent. However, after 300 iterations, a gap appears, and attention needs to be paid to the overfitting problem. The loss value is relatively large.
At 500 iterations, it is still 0.4. For accuracy, after 200 iterations, it reaches a stable bar, but the accuracy of the training set is better than that of the test set, at 95% for the training set and 92% for the test set.

1.2.4 Simplification of the SGD with torch.optim

Different from before, we added the $torch.optim.SGD$ method and created a stochastic gradient descent SGD optimizer.

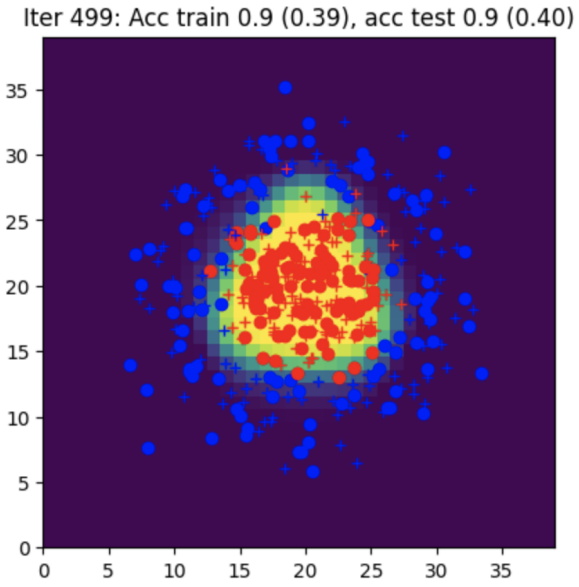

Figure 1.8

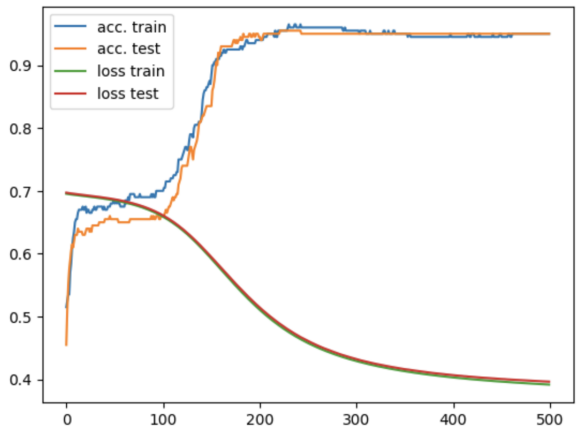

Figure 1.9

500 iterations are performed.
The loss function continues to decrease, and the results of the training set and the test set are consistent. However, after 300 iterations, a gap appears, and attention needs to be paid to the overfitting problem. The loss value is relatively large.
At 500 iterations, it is still 0.4.
For accuracy, after 200 iterations, it reaches a stable bar, but the accuracy of the training set is better than that of the test set, at 95% for the training set and 92% for the test set.
We can make the same comments as before, parameters seems to evolved slightly differently, but we have similar results.

1.2.5 MNIST application

Apply the code from the previous part code to the MNIST dataset.

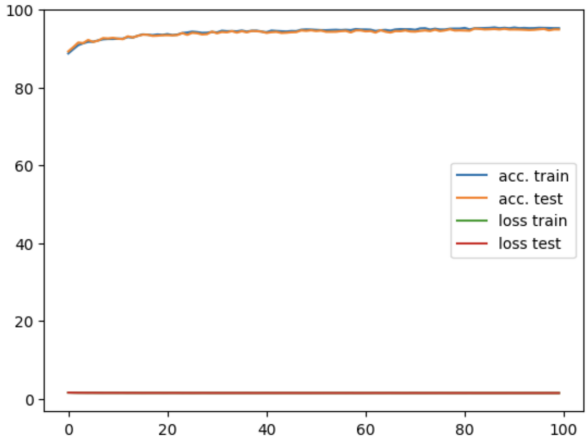

Figure 1.10

Here, I used Algorithme global d'apprentissage (avec autograd, les couches torch.nn et torch.optim) to train the MNIST dataset.
After 100 iterations, the training effect is very good, the loss function is around 1.5, the accuracy reaches more than 95%, and the training set and test set are consistent.

1.2.6 Bonus: SVM

1.2.6.1 A linear SVM (sklearn.svm.LinearSVC dans sci-kit-learn)

$sklearn.svm.LinearSVC$ is a linear support vector machine, used to process linearly separable data. The data type we provide is non-linear, so the effect is very poor, as can be seen from the figure.

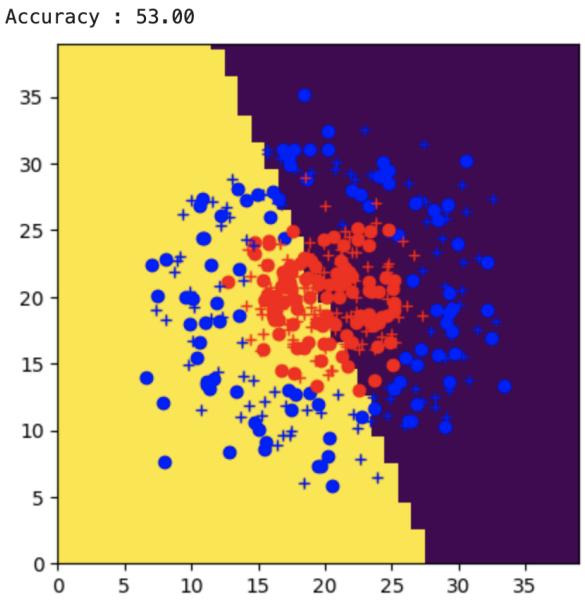

Figure 1.11

1.2.6.2 More complex kernels (sklearn.svm.SVC)

$sklearn.svm.SVC$ is used to support vector machine classification and supports the use of different kernel functions to handle linear and nonlinear classification problems.

First, we choose a linear kernel function to train the data. We can see that the conclusion is the same as the above method, which can only classify linear data.

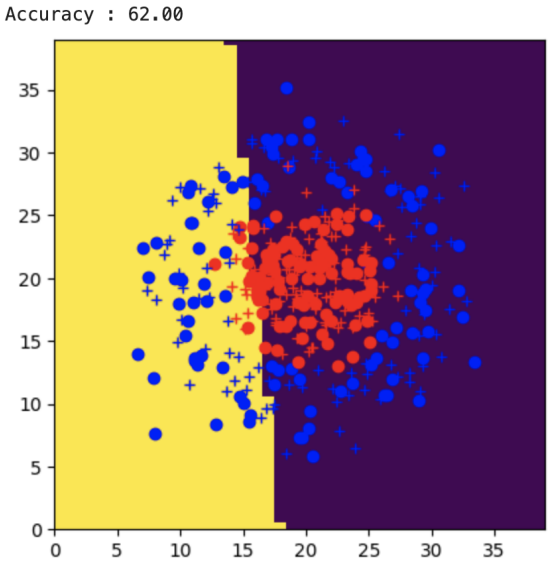

Figure 1.12

We choose the Gaussian radial basis kernel function rbf here, which is a nonlinear kernel function.

As can be seen from the figure, the Gaussian radial basis kernel function can classify the data we provide well, and when the regularization parameter $C = 1$, the accuracy is 94%.

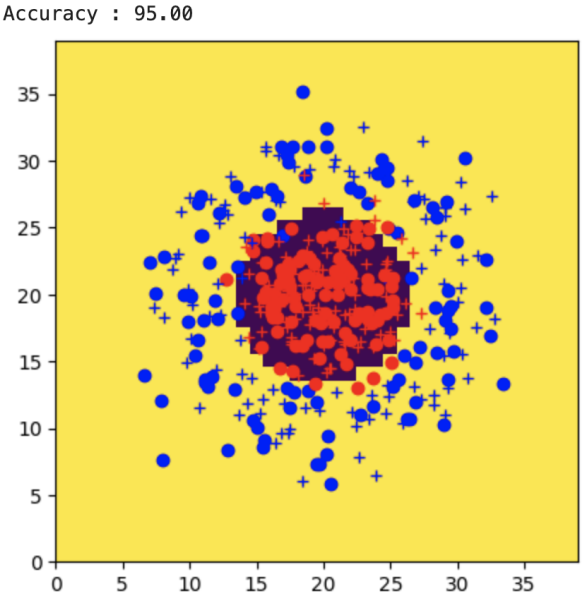

Figure 1.13

We try different regularization parameters C and study the impact of regularization parameters on SVM. Smaller values of C lead to stronger regularization, potentially leading to underfitting, while larger values of C lead to weaker regularization, potentially leading to overfitting. C is about 0.5, we have the best accuracy of about 95%.

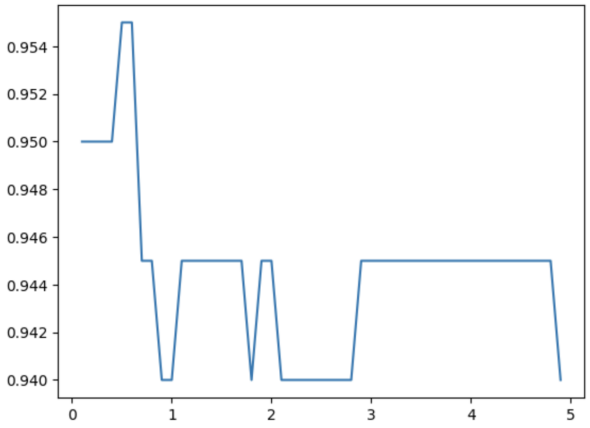

Figure 1.14

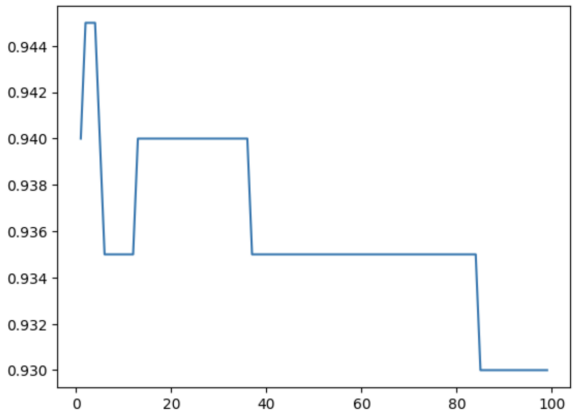

Figure 1.15

Chapter 2

Convolutional Neural Networks

2.1 Introduction to convolutional networks

→ **Question 1:** Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size $x \times y \times z$ what will be the output size? How much weight is there to learn ? How much weight would it have taken to learn if a fully connected layer were to produce an output of the same size?

Padding p: Padding in CNN is the technique of adding extra pixels around the input image or feature map to maintain spatial dimensions during the convolution operation. It helps prevent information loss at the edges and plays a vital role in the architecture and performance of convolutional neural networks.

Stride s: Stride is a component of convolutional neural networks, or neural networks tuned for the compression of image data. Stride is a parameter of the neural network's filter that modifies the amount of movement over the image. For example, if a neural network's stride is set to 1, the filter will move one pixel, or unit, at a time. The size of the filter affects the encoded output volume, so stride is often set to a whole integer, rather than a fraction or decimal.

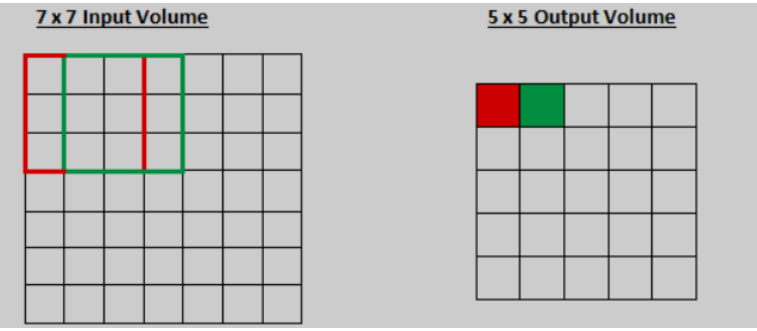

Figure 2.1 – Stride of 1

The formula for computing the output size of a convolutional layer:

— An image of dimensions $x \times y \times z$
— A filter of dimensions $k \times k$, and m is the number of filters.

— Stride s and padding p

And let's note the size of the output is $x_{out} \times y_{out} \times z_{out}$.

$$\begin{aligned} x_{out} &= \lfloor \frac{x - k + 2 \times p}{s} + 1 \rfloor \\ y_{out} &= \lfloor \frac{x - k + 2 \times p}{s} + 1 \rfloor \\ z_{out} &= m \end{aligned} \tag{2.1}$$

The output value of z is usually defined by the network architecture.

* How much weight is there to learn ?

  To calculate the total number of weights in the convolutional layer, we can use the following formula:

$$w = m \times (k \times k \times z + 1) \tag{2.2}$$

  m : Number of filters.
  $k \times k$: Size of each kernel.
  z: Depth of the input data.
  1 : Bias term associated with each filter (one bias term per filter).

  In a convolutional layer, each filter is applied across the entire input volume, and the weights are shared across different spatial locations in the input.

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

  The fully connected layer expands the feature map matrix obtained by the last layer of convolution into a one-dimensional vector. Its function is to integrate the features and provide input to the classifier. Passing a fully connected layer is equivalent to performing a convolution operation, using an $x_{out} \times y_{out} \times z_{out}$ filter to convolve the output of the last layer of convolution, and the result is the output of a neuron of a fully connected layer. This output is a value. On this basis, how many neurons there are will output such a vector. So if we want to get outputs that produce the same size, we need $(x_{out} \times y_{out} \times z_{out}) \times (x \times y \times z + 1)$ weights.

→ ⋆ **Question 2:** What are the advantages of convolution over fully-connected layers ? What is its main limit ?

— **Compared with fully-connected layers, convolution has the following advantages and limits**:

✓ **Advantages:**
- Convolutional layers can process multi-scale images, and fully connected layers must limit the size of the input image.

- The computational complexity of fully connected layers increases dramatically as the input dimensions increase, which results in longer training and inference times. And having a large number of parameters increases the complexity of the model. When the data set is relatively small, it can easily lead to over-fitting.

× **Limits:**

- Convolutional layers are mainly used to capture local features and therefore may not be as effective as fully connected layers in handling global patterns or global dependencies.
- Fully connected layers can establish global connections between inputs and outputs and therefore can be used to capture global patterns and complex dependencies. Convolutional layers are mainly used to capture local features and therefore may not be as effective as fully connected layers in handling global patterns or global dependencies.

→ ⋆ **Question 3:** Why do we use spatial pooling?

— First of all, in convolutional neural networks, a pooling layer is usually added between adjacent convolutional layers.

Because the pooling layer can effectively reduce the size of the parameter matrix, thereby reducing the number of parameters in the last connected layer. Adding a pooling layer can speed up calculations and prevent over-fitting.

Pooling can retain most of the important information while reducing the dimensionality of each feature map.

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

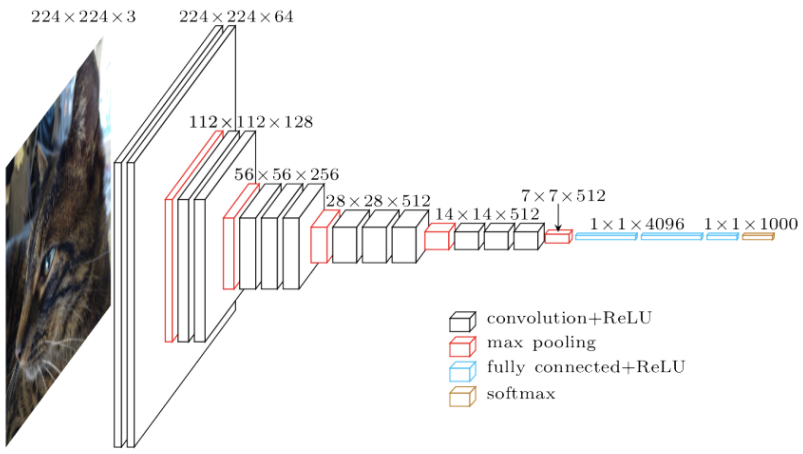

Figure 2.2 – VGG16 network

The convolutional layer is not restricted by the size of its input, it simply performs a

convolution operation on the entire input image and outputs a correspondingly changed image. This means that the convolutional layer can handle input images of different sizes because the convolution kernel slides over the entire input, regardless of the specific size of the input image.

The size of the input image issue mainly involves fully connected layers. A fully connected layer is densely connected, with each neuron connected to every neuron in the previous layer, so we need to be clear about the input image size.

For VGC-16 network, before the Fully connected layer, we have the size of $7 \times 7 \times 512$, it's exactly of the size of $1 \times 1 \times 4096$.

→ **Question 5:** Show that we can analyze fully-connected layers as particular convolutions.

The fully connected layer can be considered a special case of a convolutional layer where the filter(kernel) size is equal to the size of the input.

We can think of fully connected layers as a specific type of convolutional operation. In a fully connected layer, each neuron is interconnected with every neuron in the previous layer and has a weight.

In actual use, a fully connected layer can be implemented by a convolution operation: a fully connected layer whose front layer is fully connected can be converted into a convolution with a convolution kernel of 1×1; and a fully connected layer whose front layer is a convolution layer can be converted into The convolution kernel is the global convolution of $h \times w$, and h and w are the height and width of the previous layer convolution result respectively.

→ **Question 6:** Suppose that we, therefore replace fully-connected 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 the fully connected layers in a neural network with their convolutional equivalents, we can handle different image sizes than the ones for which the network was originally trained. Using an input image that is larger than the image size originally expected by the network, the convolution operation will generate larger feature maps in the intermediate layers, and these feature maps will be propagated through the network. The final convolutional layers, which replace the fully connected layers, apply their filters to the full spatial extent of the incoming feature map. The output shape in this case will no longer be the typical single row of class scores. Instead, it will be a spatial score map whose dimensions depend on the input size, the properties of the convolutional layer, and the operations applied during forward propagation. This means that the output will be a two-dimensional matrix, where each position represents the corresponding score or activation value in the image.

→ **Question 7:** We call the receptive field of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers ? Can you imagine what happens to the

deeper layers ? How to interpret it?

Receptive field: The size of the region in the feature map output by each network layer in a convolutional neural network that maps back to the original input feature. In other words, an element point on the network layer output feature map is mapped from the area in the original input, and its receptive field is how large it is.

Starting from the current layer and reverse mapping to the original input, first calculate the receptive field size of the current layer output in the output of the previous network layer, and then pass it to the original input in turn to get the receptive field size owned by the current layer. Calculated as follows:

$$RF_{i+1} = RF_i + (K_i - 1) \times S_i \tag{2.3}$$

Among them, for a certain operating layer (convolution layer, pooling layer), RF_i represents the receptive field of the input feature image element of the operating layer, RF_{i+1} represents the receptive field of the output feature image element, and K_i represents the operating kernel of the operating layer. Size S_i represents the product of the strides of all previous operation layers (excluding this layer).

The deeper the network layer, the larger the corresponding receptive field of the elements of its output features.

2.2 Training from scratch of the model

2.2.1 Network architecture

→ **Question 8:** For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed?

For convolutions, we want to maintain the same spatial dimensions at the output as at the input. That is to say, $x_{out} = x_{in}$ and $y_{out} = y_{in}$. By derivation from formula (2.1), we can get the formulas of padding and stride:

$$\begin{aligned} x_{out} &= \frac{x - k + 2 \times p}{s} + 1 \\ (x-1) \times s &= x - k + 2p \\ p &= \frac{(x-1) \times s - x + k}{2} \end{aligned} \tag{2.4}$$

And for our model, the size of image is 32×32 and $x_{in} = y_{in} = 32$. The kernel of convolution is 5×5. So we can calculate p :

$$p = \frac{31s - 27}{2} \tag{2.5}$$

We want to use every pixel, so we choose $s = 1$ and then $p = 2$.

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

By the max pooling, we have $x_{out} = \frac{1}{2}x_{in}$ and $y_{out} = \frac{1}{2}y_{in}$. And as the same, we apply the formula (2.1):

$$\begin{aligned} x_{out} &= \frac{x_{in} - k + 2 \times p}{s} + 1 \\ \frac{1}{2}x_{in} &= \frac{x_{in} - k + 2 \times p}{s} + 1 \\ (\frac{1}{2}x - 1) \times s &= x - k + 2p \\ p &= \frac{(\frac{1}{2}x - 1) \times s - x + k}{2} \end{aligned} \tag{2.6}$$

And we have $x = 32$ and $k = 2$, so we can calculate p and s :

$$p = \frac{15s - 30}{2} \tag{2.7}$$

Same as the previous question, we want to use every pixel, so we choose $s = 2$ and then $p = 0$.

→ ⋆ **Question 10:** For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.

1. conv1 : 32 convolutions 5×5, followed by ReLU
   - Input size: $32 \times 32 \times 3$
     The size of the image is 32×32, but the image is by RGB, so there are 3 channels.
   - Output size after the convolution: $32 \times 32 \times 32$
     The kernel size is 5×5, we suppose the stride is 1 and the padding is 2. So we can calculate the output with the formula (2.1):

     $$((32 - 5 + 2 \times 2) + 1) \times ((32 - 5 + 2 \times 2) + 1) = 32 \times 32$$

     And then we apply 32 filters, so the depth is 32.
   - Weights: $(3 \times 5 \times 5 + 1) \times 32$
     5×5 for 3 is for each filter, and we have 32 filters of 5×5 for 3 channels, so there are parameters: $(3 \times 5 \times 5 + 1) \times 32$.

2. pool1 : max-pooling 2×2

- Input size: $32 \times 32 \times 32$
  The input of the pool1 layer is the output of the conv1 layer.
- Output size after max-pooling: $16 \times 16 \times 32$
  Max-pooling with a 2×2 window reduces the size by a factor of 2, and the depth isn't changed.
- Weights: In the pooling layer, we do not learn any weights because, in this layer, there are no operations involving parameters.

3. conv2 : 64 convolutions 5×5, followed by ReLU
   - Input size: $16 \times 16 \times 32$
     The input of the conv2 layer is the output of the pool1 layer.
   - Output size after the convolution: $16 \times 16 \times 64$
     The kernel size is 5×5, we suppose the stride is 1 and the padding is 0. So we can calculate the output with the formula (2.1):

     $$((16 - 5 + 2 \times 2) + 1) \times ((14 - 5 + 2 \times 2) + 1) = 16 \times 16$$

     And then we apply 64 filters, so the depth is 64.
   - Weights: $(32 \times 5 \times 5 + 1) \times 64$
     $5 \times 5 \times 32$ is for each filter, and we have 64 filters of 5×5 for 32 channels, so there are parameters: $(32 \times 5 \times 5 + 1) \times 64$.

4. pool2 : max-pooling 2×2
   - Input size: $16 \times 16 \times 64$
     The input of the pool2 layer is the output of the conv2 layer.
   - Output size after max-pooling: $8 \times 8 \times 64$
     Max-pooling with a 2×2 window reduces the size by a factor of 2, and the depth isn't changed.
   - Weights: In the pooling layer, we do not learn any weights because, in this layer, there are no operations involving parameters.

5. conv3 : 64 convolutions 5×5, followed by ReLU
   - Input size: $8 \times 8 \times 64$
     The input of the conv2 layer is the output of the pool2 layer.
   - Output size after the convolution: $8 \times 8 \times 64$
     The kernel size is 5×5, we suppose the stride is 1 and the padding is 0. So we can calculate the output with the formula (2.1):

     $$((8 - 5 + 2 \times 2) + 1) \times ((5 - 5 + 2 \times 2) + 1) = 8 \times 8$$

     And then we apply 64 filters, so the depth is 64.
   - Weights: $(64 \times 5 \times 5 + 1) \times 64$
     $5 \times 5 \times 64$ is for each filter, and we have 64 filters of 5×5 for 64 channels, so there are parameters: $(64 \times 5 \times 5 + 1) \times 64$.

6. pool3 : max-pooling 2×2
   - Input size: $8 \times 8 \times 64$
     The input of the pool3 layer is the output of the conv3 layer.

- Output size after max-pooling: $4 \times 4 \times 64$
  Max-pooling with a 2×2 window reduces the size by a factor of 2, and the depth isn't changed.
- Weights: In the pooling layer, we do not learn any weights because, in this layer, there are no operations involving parameters.

7. fc4 : fully-connected, 1000 output neurons, followed by ReLU
   - Input size: $4 \times 4 \times 64$
     The input of the fc4 layer is the output of the pool3 layer.
   - Output size:1000 neurons
   - Weights: $(1 + 1000) \times (4 \times 4 \times 64)$
     1000 neurons with the input $1 \times 1 \times 64$, so we have the nomber of parameters: $(1 + 1000) \times (4 \times 4 \times 64)$.

8. fc5 : fully-connected, 10 output neurons, followed by ReLU
   - Input size: 1000
     The input of the fc5 layer is the output of the fc4 layer.
   - Output size:10 neurons
   - Weights: $1000 \times (10 + 1)$
     10 neurons with the input 1000, so we have the number of parameters: $1000 \times (10 + 1)$.

→ **Question 11:** What is the total number of weights to learn ? Compare that to the number of examples.

$$(3 \times 5 \times 5 + 1) \times 32 = 2432$$

$$(32 \times 5 \times 5 + 1) \times 64 = 51264$$

$$(64 \times 5 \times 5 + 1) \times 64 = 102464$$

$$(1 + 1000) \times (4 \times 4 \times 64) = 1025024$$

$$1000 \times (10 + 1) = 10010$$

$$2432 + 51264 + 102464 + 1025024 + 10010 = 1192024$$

So the total number of weights to learn is 1192024. Comparing that to the number of examples, a dataset of 50000 training and 10000 test samples, so 50000 examples of size $32 \times 32 \times 3$ makes 51200000 pixels.

→ **Question 12:** Compare the number of parameters to learn with that of the BoW and SVM approach.

For BoW and SVM, we have to learn the words of the dictionary, 1000 vectors for 10 classes, so $(1000 + 1) \times 10 = 10010$ parameters are to be learned. The total number of parameters is calculated by our model.

2.2.2 Network learning

To learn the network, start from the $base.py$ file provided. This file learns a small historical convolutional network, called LeNet5, based on the MNIST database. Start by reading and experimenting with this code before modifying it to do what you want.

→ **Question 13:** Read and test the code provided. You can start the training with this command : $main(batch\_size, lr, epochs, cuda = True)$

I test the code provided with the parameters $batch\_size = 5, lr = 0.1, epochs = 5, cuda = True$). Finally, we get the following result:

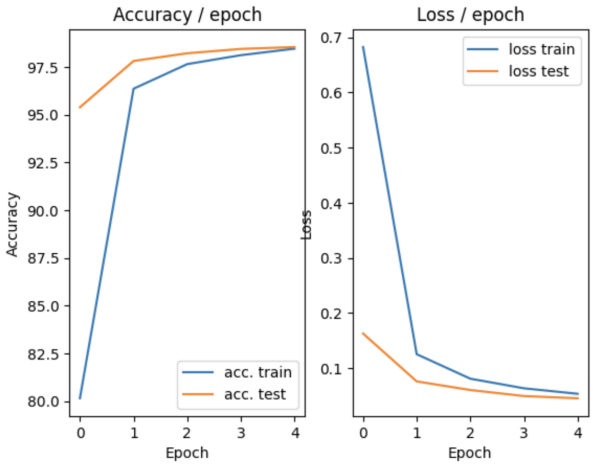

Figure 2.3 – Accuracy and Loss of MNIST

→ ⋆ **Question 14:** In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the difference in data) ?

In the code provided, we calculate loss and accuracy in train and test using the $epoch$ function: epoch(data, model, criterion, optimizer=None, cuda=False). When calculating loss and accuracy in train, the parameters are epoch(train, model, criterion, optimizer, cuda), and here, optimizer = torch.optim.SGD(model. parameters(), lr). But when calculating loss and accuracy in the test, the parameters are epoch(test, model, criterion, cuda=cuda), and there is no optimizer by default. If an optimizer is given, perform a training epoch using the given optimizer, otherwise, perform an evaluation epoch (no backward) of the model.

→ **Question 15:** 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.

I test the code provided with the parameters $batch\_size = 5, lr = 0.1, epochs = 5, cuda = True$). Finally, we get the following result:

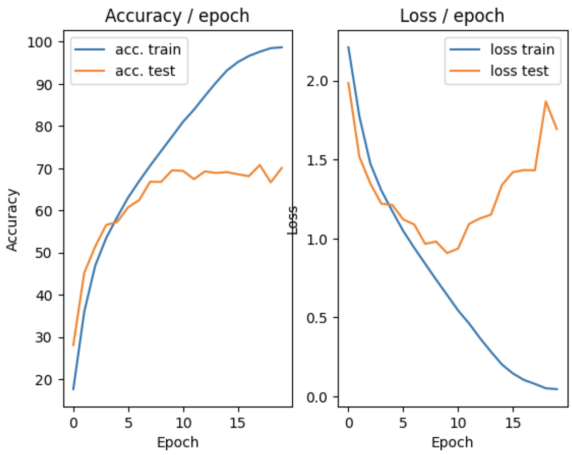

Figure 2.4 – Accuracy and Loss of CIFAR-10

→ ⋆ **Question 16:** What are the effects of the learning rate and of the batch-size ?

Here we use the stochastic descent algorithm to optimize the CNN model. The learning rate and batch size directly determine the weight update of the model and are the most important parameters that affect the convergence of model performance. The learning rate directly affects the convergence state of the model, while the batch size affects the generalization performance of the model, and they also influence each other.

The learning rate determines the step size of the weight iteration. If the learning rate is too large, the model will not converge, but if it is too small, the model will converge very slowly or be unable to learn. As the learning rate increases, the loss will gradually become smaller and then increase, and the best learning rate can be selected from the area with the smallest loss. However, as the learning rate increases, the model may also transition from under-fitting to over-fitting.

At first, the batch decides the direction of descent. Within a certain range, increasing the batch size helps the stability of convergence, but as the batch size increases, the performance of the model will decrease. When batch size becomes large (above a critical point), it will reduce the generalization ability of the model.

If the learning rate is increased, it is best to increase the batch size as well, so that the convergence is more stable. Usually, when we increase the batch size to n times the original, we must ensure that the updated weights after the same sample are equal. According to the linear scaling rule, the learning rate should be increased to n times the original. But if you want to ensure that the variance of the weight remains unchanged, the learning rate should be increased to the original $\sqrt{n}$ times.

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

From **Figure 2.4**, we can observe at the start of the first epoch, the error of the train is more than that of the test, and the accuracy of the train is less than that of the test. Because in the beginning of training, due to the initialization of model parameters and the influence of data subsets, started with randomly initialized weights, training loss and accuracy will

be affected. Test loss and accuracy are generally more stable because they are calculated on the entire data set. We stabilize it by training the model.

→ ⋆ **Question 18:** Interpret the results. What's wrong? What is this phenomenon ?

Same as the previous question, we observe from **Figure 2.4**. In the beginning, the model's performance was very poor, but with iteration, the performance gradually improved. At about the sixth or seventh iteration, the figure has a change. The Accuracy of the test curve gradually stabilizes, and the Accuracy of the train curve continues to rise. This phenomenon is called over-fitting. Over-fitting is the phenomenon in which a deep learning model performs well on training data but performs poorly on new, unseen data.

2.3 Results improvements

2.3.1 Standardization of examples

Implementation:

— Add normalization in the data pre-processing.

— This should be done when calling datasets.CIFAR10 :

```
    train_dataset = datasets.CIFAR10(PATH, train=True, download=True,
    transform=transforms.Compose([
            transforms.ToTensor(),
            normalize
        ]))
    val_dataset = datasets.CIFAR10(PATH, train=False, download=True,
        transform=transforms.Compose([
            transforms.ToTensor(),
            normalize
        ]))
```

And for CIFAR-10, the values are $\mu = [0.491, 0.482, 0.447]$ and $\sigma = [0.202, 0.199, 0.201]$.

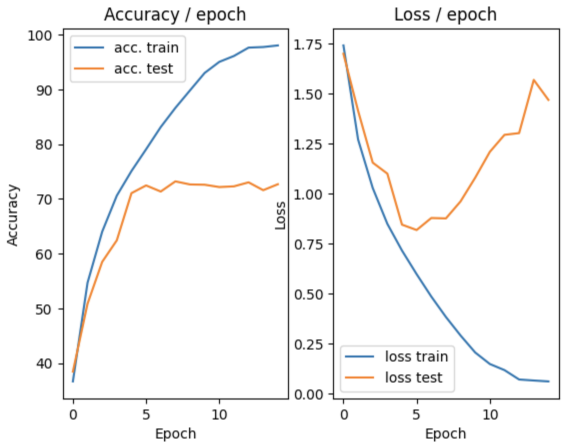

Figure 2.5 – Standardization of examples

→ **Question 19:** Describe your experimental results.

Normalizing the CIFAR-10 dataset, we can see that it can help accelerate the convergence process of the model, the model performance is also improved, and we have better accuracy and less loss. But the over-fitting situation still exists.

→ **Question 20:** Why only calculate the average image on the training examples and normalize the validation examples with the same image?

First, we normalize the training data by the method z-score to ensure that each sample of the training data is standardized during the training process. From this, we calculate the mean and standard deviation for the training set data.

Then, we use the same mean and standard deviation as the training data in the validation set to ensure consistency between the training set and validation set. Because when we train the model, the weights and parameters of the model are adjusted and learned on the training data. If we use different means and standard deviations to normalize data in validation, the distribution of the input data will not be consistent with the distribution of the training data, causing the model to perform poorly on validation or test data.

→ **Question 21:** **Bonus:** There are other normalization schemes can be more efficient like ZCA normalization. Try other methods, explain the differences, and compare them to the one requested.

ZCA normalization centers the data by subtracting the mean of the data for each feature so that the data is centered around the origin. Calculate the covariance matrix of the data, find the eigenvalues and eigenvectors of the covariance matrix, project the center data onto the eigenvectors, multiply the result by the square root of the corresponding eigenvalue, and then multiply the result by the square root of the eigenvalue again. Finally, the decorrelated and rescaled data are multiplied by the transpose of the feature vector to obtain ZCA whitened data.

ZCA normalization helps to retain the original structure of the data, reduces the correlation between features, and makes it easier to learn effective features, but it is more complex and computationally expensive.

2.3.2 Increase in the number of training examples by data increase

Standardization makes the mean of the data close to 0 and the standard deviation close to 1, helping to scale the data into a uniform range. ZCA normalization first standardizes the data and also converts the covariance matrix of the data into a diagonal matrix through a rotation operation, which reduces the correlation between data features, improves the separability of the data and improves the feature learning effect.

Implementation

— Modify the call to $datasets.CIFAR10$ to add in train a random crop size 28 and a random horizontal symmetry ; and in test a centered crop size 28.

So here, we use $transforms.RandomCrop(28)$ is used to cut randomly and size is 28×28. For a random horizontal symmetry, we use $transforms.RandomHorizontalFlip()$.

— Modify the pool3 layer to add the option $ceil\_mode = True$.

$ceil\_mode$: When matching data according to Stride movement, the data does not meet the size of $kernel\_size$, which involves a trade-off problem. If $ceil\_mode$ is True, it will be retained. If it is False, it will not be retained. Here, we add this option to the pool3 layer.

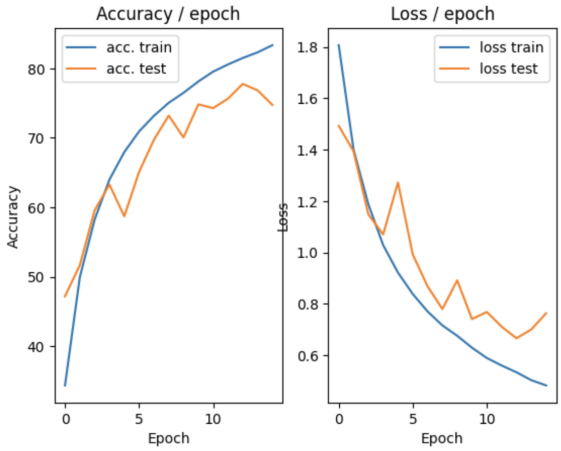

Figure 2.6 – data augmentation

→ **Question 22:** Describe your experimental results and compare them to previous results.

Augment the dataset and from **Figure 2.6**, we can observe that the model has better generalization ability, but the over-fitting problem is limited. But we can also observe that it's not as stable as the test effect.

→ **Question 23:** Does this horizontal symmetry approach seem usable on all types of images? In what cases can it be or not be?

Data enhancement through the horizontal symmetry approach is not suitable for all image data. This depends on the nature of the image data and the requirements of the task. In general, the horizontal symmetry approach can increase the diversity of training data and improve the invariance of features, thereby better-capturing features in images and improving classification accuracy. However, when the image we need to detect contains specific directional information, such as text or directional patterns, information may be lost and incorrect category prediction may occur.

→ **Question 24:** What limits do you see in this type of data increase by transformation of the dataset ?

- Some information in the image is lost. For example, when we apply strong image transformations.
- Causing data inconsistency. As in the previous question, when we recognize text images, we change the directional information in the image.

- May introduce noise and errors.
- More beneficial for a specific task, we need to choose the appropriate data augmentation method.

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

We have many data augmentation methods, such as resizing the image, cropping the image, rotating, increasing image contrast, flipping, etc. Here I tried to randomly rotate the image. I set the rotation degree to up to 30 degrees. The method is as follows:

```
transforms.RandomRotation(30)
```

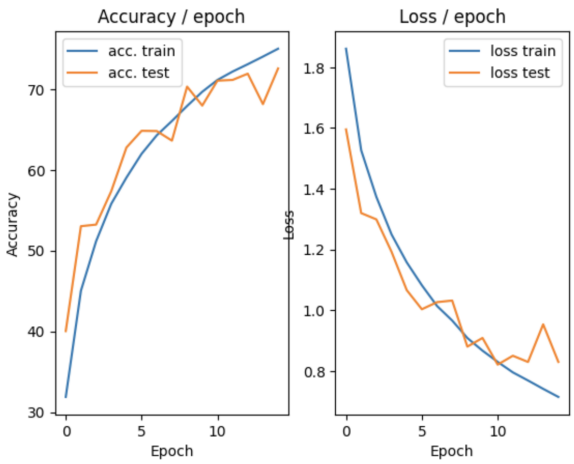

Figure 2.7 – Random rotation of the image

We can see from the results that the training process is smoother, but it takes longer due to data enhancement.

2.3.3 Variants on the optimization algorithm

Implementation

— Add a learning rate scheduler and use an exponential decay with a coefficient of 0.95.

So in the function *main*, we add the code following:

```
optimizer = torch.optim.SGD(model.parameters(), lr, momentum=0.9)
lr_sched =  torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
```

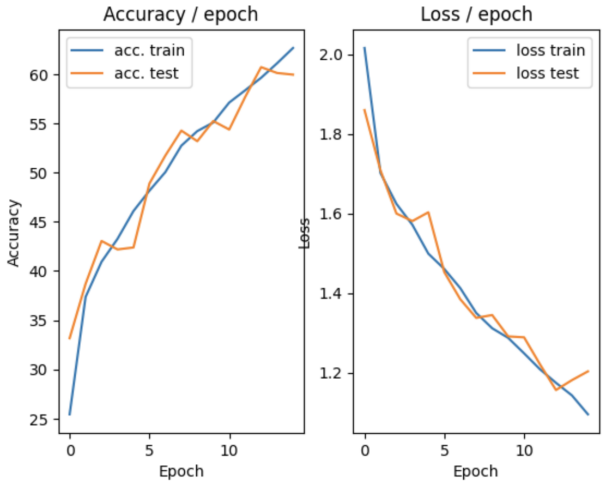

Figure 2.8 – Variants on the optimization algorithm

→ **Question 26:** Describe your experimental results and compare them to previous results, including learning stability.

Add a learning rate scheduler and use an exponential decay with a coefficient of 0.95. As training progresses, gradually reduce the size of the learning rate. The learning rate will decrease exponentially. Use the decay coefficient to control the speed of decline. We can observe that the training process is more stable, reducing the risk of oscillations in training. The generalization ability is improved and the model converges better to the minimum value of the loss function.

→ **Question 27:** Why does this method improve learning ?

In the previous questions, we can see that the results of the training are not as good as the results of the test at the beginning of the study. This is because the initial values of the model parameters may be far away from the optimal values. Therefore, the attenuated learning rate scheduler can be given a large initial learning rate, which can quickly approach the optimal solution. At the same time, by gradually reducing the learning rate, the model can learn more stably in the early stages without skipping the optimal solution. As the learning rate decreases, the magnitude of parameter updates gradually decreases, and the model gets closer to the minimum value of the loss function. And we observe that over-fitting occurs late in training, and a smaller learning rate helps prevent over-fitting later in training.

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

Regarding SGD variants: Mini-batch SGD uses a small batch of samples in each iteration to calculate the gradient. AdaGrad adaptively adjusts the learning rate based on the historical gradient of each parameter. Different parameters can have different learning rates. RMSprop: RMSprop is similar to AdaGrad, but it uses a moving average to adjust the learning rate to reduce the learning rate. of rapid decline.

About the learning rate scheduler: The StepLR learning rate scheduler reduces the learning rate at the end of scheduled training epochs. The ReduceLROnPlateau learning rate scheduler monitors performance on the validation set and reduces the learning rate when performance no longer improves. This is helpful for adaptively adjusting the learning rate for better performance. The CyclicLR learning rate scheduler allows the learning rate to change cyclically within a range to help the model jump out of local minima.

Learning rate schedulers are often used with optimizers such as SGD. The learning rate can be automatically adjusted according to the progress of training, which helps to improve the performance of the model and speed up the training convergence speed.

I tried using the RMSprop optimizer and the StepLR learning rate scheduler as follows:

```
    optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001, alpha=0.9)
    lr_sched = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma
    =0.5)
```

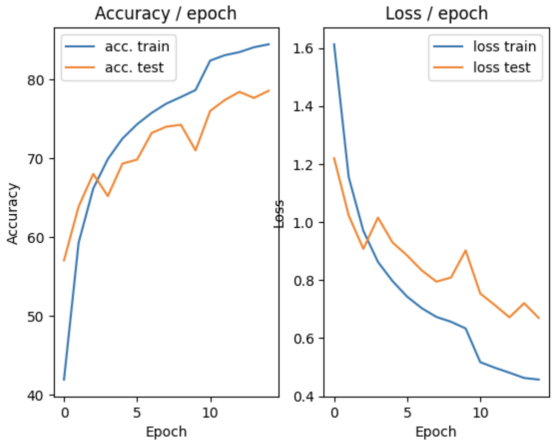

Figure 2.9 – RMSprop and StepLR

From the result, we can see the effect is not good. So we need to choose different models for different problems.

2.3.4 Regularization of the network by dropout

Implementation

— Add a dropout layer between fc4 and fc5.

So we add $nn.Dropout()$ between fc4 layer and fc5 layer as following:

```
        self.classifier = nn.Sequential(
            nn.Linear(1024, 1000),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(1000, 10),
        )
```

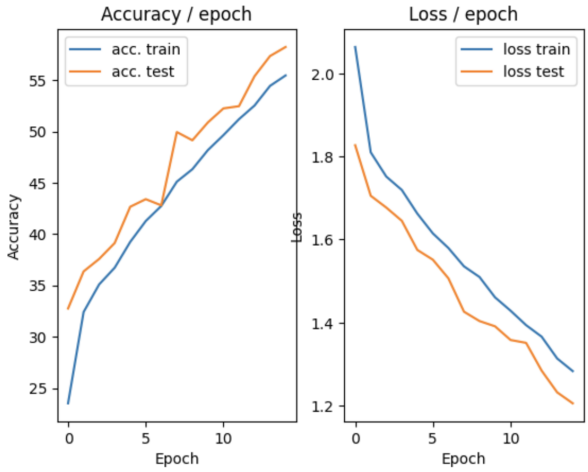

Figure 2.10 – Regularization by dropout

→ **Question 29:** Describe your experimental results and compare them to previous results.

With the dropout method, we have less over-fitting, but the convergence is slower.

→ **Question 30:** What is regularization in general ?

How to reduce generalization error is the core issue of machine learning. The goal of regularization is to make the model generalize better to new data, not just to remember the training data. Regularization aims to prevent the model from over-fitting the training data. Over-fitting is when a model performs very well on training data but performs poorly on unseen data.

The most direct regularization is to add a penalty term to the loss function, such as L2 regularization. Weight decay focuses on the square root of the sum of squares of weights, which is to make the weight in the network close to 0 but not equal to 0. In L1 In regular expressions, what matters is the absolute value of the weight, which may be compressed to 0. In deep learning, L1 will tend to produce a small number of features, while other features are 0, while L2 will select more features, and these features will be close to 0.

The most commonly used regularization technique is dropout, which randomly drops some neurons or weights. During the training process, we use dropout to regularize the hidden layer weights.

→ **Question 31:** Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it ?

During forward propagation, when passing through a certain layer of neurons, some of the neurons in this layer are discarded with a certain probability p, and only another part of the neurons is allowed to perform calculations. At this time, the output value of the network will not depend on discarded neurons, the weight gradients associated with the discarded neurons are all 0 during back-propagation. Since neurons are dropped randomly during the training process, this can prevent the model from relying too much on certain local features, thereby playing a regularization role when training the model, and

can be used to deal with the problem of over-fitting.

→ **Question 32:** What is the influence of the hyperparameter of this layer ?

The hyperparameters of dropout mainly include the dropout rate and the position of the dropout layer. The explanation is as follows:

- The dropout rate represents the probability of turning off a neuron, usually between 0 and 1. A lower dropout rate generally reduces over-fitting better, but may result in the model under-fitting the training data. A higher dropout rate may increase the generalization ability of the model, but if the dropout rate is too high, it may cause a decrease in model performance.
- We can choose different layers, which will produce different effects in different positions.

→ **Question 33:** What is the difference in behavior of the dropout layer between training and test ?

The function of the $nn.Dropout()$ layer is to randomly turn off a part of the neurons. During training, in each batch, some neurons will be turned off, while other neurons will remain active, randomly operating. This is so that the model does not rely too much on specific neurons and reduces the risk of over-fitting. During the test, all neurons remained active and no neurons were turned off. This is because we want to get consistent, repeatable results and don't want to introduce randomness into the training.

2.3.5 Use of batch normalization

Implementation

— Add a 2D batch normalization layer immediately after each convolution layer.

Here, we add $nn.BatchNorm2d(n)$ between the $nn.Conv2d()$ and $nn.ReLU()$, n is the size of the convolution layer output. The following is the first convolution layer.

```
nn.Conv2d(3, 32, (5, 5), stride=1, padding=2),
nn.BatchNorm2d(32),
nn.ReLU(),
```

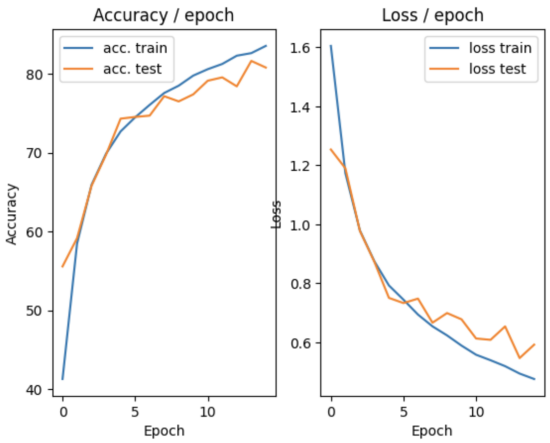

Figure 2.11 – Batch normalization

→ **Question 34:** Describe your experimental results and compare them to previous results.

From **Figure 2.9**, we can observe that the convergence is faster and the over-learning is limited. In each training batch of data, the input to each layer is normalized, so each layer no longer depends on the feature distribution of the previous layer, and each layer learns useful features more independently.

Chapter 3

Transformers

3.1 Self-attention

→ **Question 1:** What is the main feature of self-attention, especially compared to its convolutional counterpart? What is its main challenge in terms of computation/memory?

Main features:

— Self-attention mechanism facilitates parallelization significantly which increases efficiency in modern hardware processing, ultimately leading to a much faster training phase.

— Self-attention is adept at capturing relationships between elements in a sequence, regardless of how far apart they are. In contrast, convolutional layers possess fixed receptive fields and can only discern local connections within that defined region. This renders self-attention particularly suitable for tasks involving the modeling of distant relationships between words or tokens in a sentence.

— Self-attention operates independently of the specific positions of elements within a sequence. It can flexibly adapt to the order of elements, maintaining its consistency regardless of how elements are arranged. Conversely, convolutional layers are influenced by the precise order of elements in the input

Challenges:

— Self-attention mechanism is very data hungry. Robust self-attention models require large and diverse datasets for training, which may not always be available.

— The computational requirements of self-attention models might mandate robust hardware for effective training and inference. This factor could be a primary reason for the continued popularity of CNNs for at least a few more years.

— Self-attention models require a significant amount of memory to store the attention weights and intermediate activations during both training and inference

— Transfer learning with self-attention models can sometimes be less straightforward compared to other architectures.

3.2 Full ViT model

→ **Question 2:** Explain what is a Class token and why we use it.

— Class Tokens are initial tokens randomly added to the start of the input sequence. These tokens do not contain any meaningful information upon initialization. However, as the Transformer network processes the input through multiple layers, the Class Token gradually accumulates information from the other tokens in the sequence. When the Vision Transformer performs the final sequence classification, it employs a Multilayer Perceptron (MLP) head that exclusively considers data from the last layer's Class Token, without taking other information into account. This suggests that the Class Token serves as a kind of placeholder data structure used to store information derived from other tokens in the sequence. By assigning an empty token for this purpose, the Vision Transformer aims to reduce the likelihood of biasing the final output in favor of or against any individual token in the sequence

→ **Question 3:** Explain why the positional embedding (PE) and why is important?

— ViT incorporates a positional encoding for each patch in the image. This is a crucial step in enabling the model to understand the spatial arrangement of the patches within the input image.
They provide details about where each patch is located within the image. This is crucial because, unlike conventional text-based transformers, images do not naturally possess a sequential order, like words or tokens in a sentence. The absence of positional encoding would result in the model being unable to discern the spatial arrangement of different patches in the image.
PE allows the model to understand the spatial context of the image and make sense of the relationships between different regions in the input, which is essential for various computer vision tasks, such as object detection or image classification.

3.3 Experimenting With Different Hyperparameters

3.3.1 Case 1

— epochs = 10
— embed_dim = 32
— patch_size = 7
— nb_blocks = 2

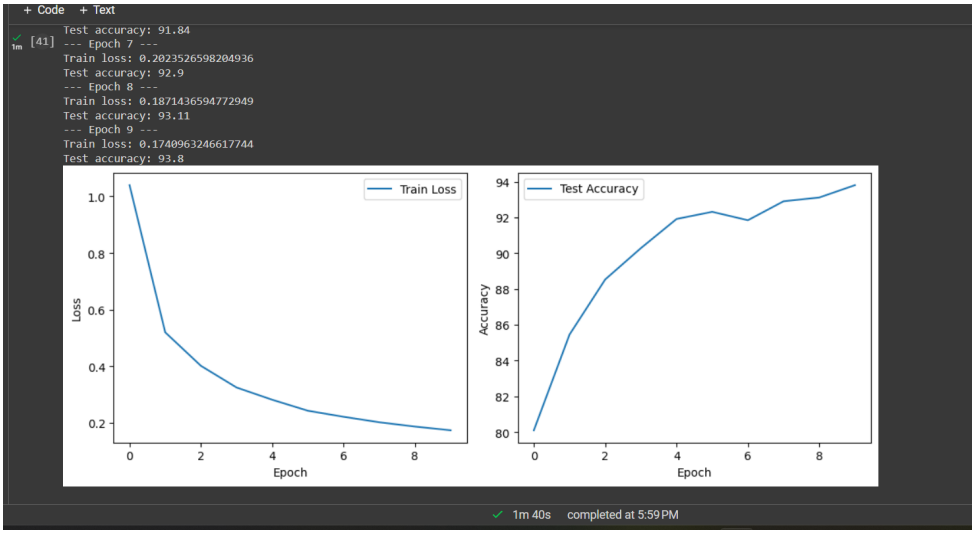

Figure 3.1 – Final Test Accuracy of 93.8

3.3.2 Case 2

It's noticeable that increasing the embedding dimension led to significant fluctuations in the graph. Surprisingly, even with an extended training period of 20 epochs, we achieved slightly lower accuracy.

— epochs = 20
— embed_dim = 128
— patch_size = 7
— nb_blocks = 2

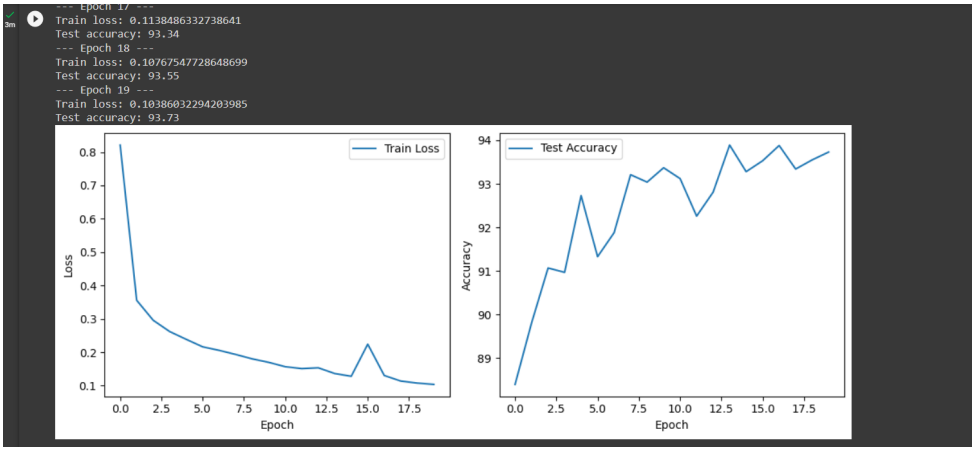

Figure 3.2 – Final Test Accuracy of 93.73

3.3.3 Case 3

We achieved a higher accuracy but there are still fluctuations in the plot.

— epochs = 20
— embed_dim = 128
— patch_size = 14
— nb_blocks = 2

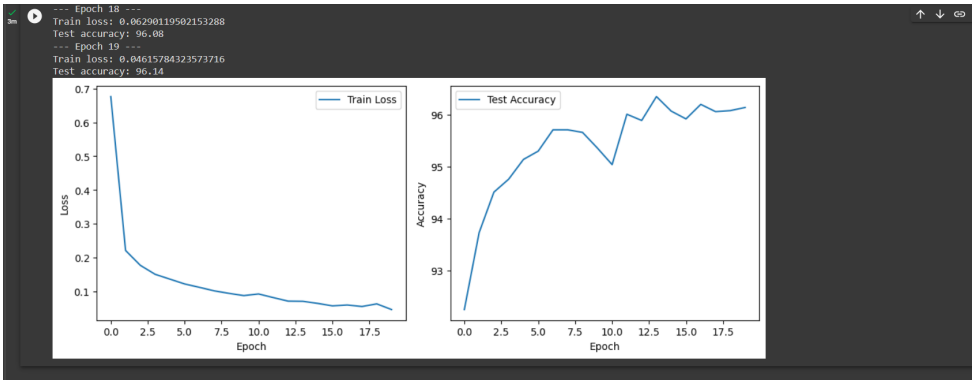

Figure 3.3 – Final Test Accuracy of 96.14

3.3.4 Case 4

We decreased the embedding dimension again, which resulted in fewer fluctuations.

— epochs = 20
— embed_dim = 64
— patch_size = 14
— nb_blocks = 2

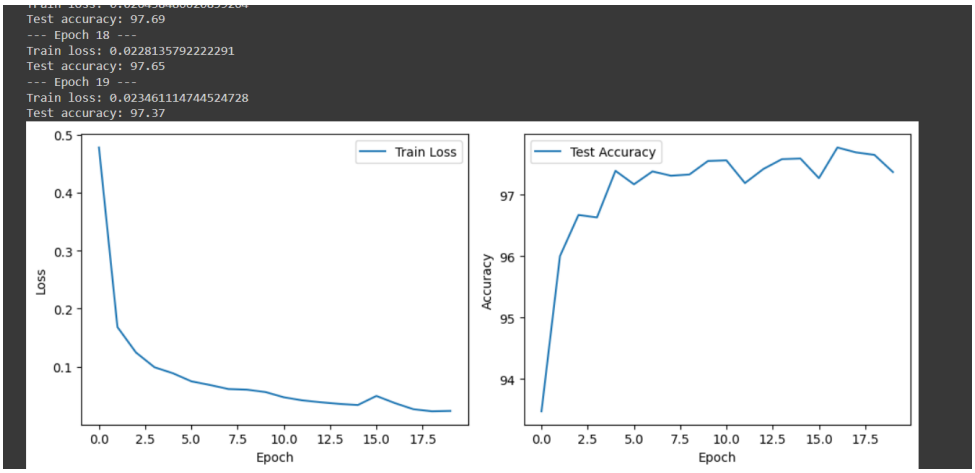

Figure 3.4 – Final Test Accuracy of 97.37

3.3.5 Case 5

We decreased the embedding dimension again, which resulted in fewer fluctuations.

— epochs = 20
— embed_dim = 32
— patch_size = 14
— nb_blocks = 2

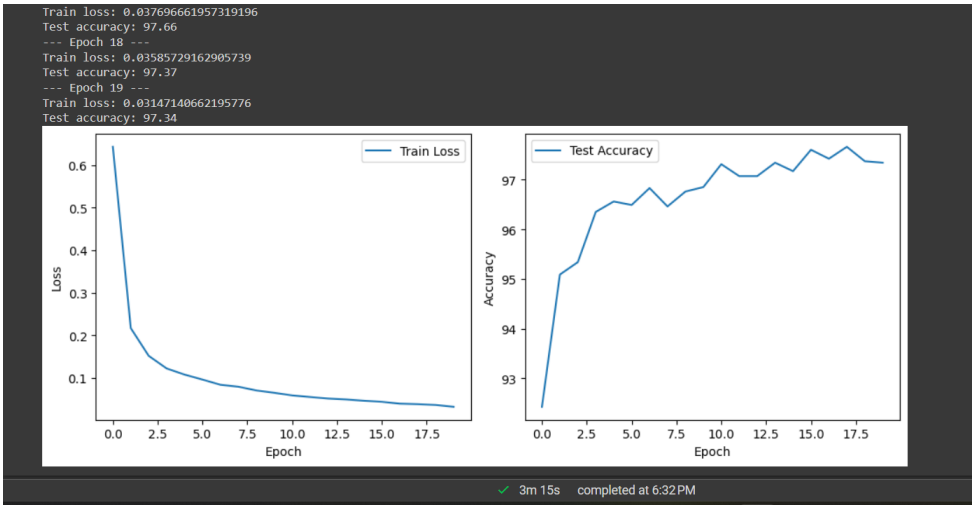

Figure 3.5 – Final Test Accuracy of 97.34

3.4 larger transformers

→ **Question 8:** Load the model using the timm library without pre-trained weights. Apply it directly on a tensor with the same MNIST image resolution. What is the problem and

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

— When we attempt to load the ViT model from the timm library without the pre-trained weights and apply it directly to the MNIST dataset, we run into errors which are due to differences in the number of input channels and the size mismatch for input features.
MNIST data is composed of grayscale images(one single channel), while ViT architecture is designed in a way that expects RGB images(3 channels). One way to address this issue is to duplicate the single grayscale channel so that we end up with two additional channels. This means that every pixel's value from the grayscale channel is copied into all 3 channels to fit the ViT architecture.

The original CNN designed by Yann LeCum was primarily designed for single-channel images, however later on the CNN architecture evolved and made them suitable for both gray-scale and multi-channel RGB images. We can modify modern CNN architecture to meet our needs and handle both scenarios.

ViT models are designed for large images. The common input size for many ViT models is 224x224 or 384x384. However, MNIST images are 28x28 which makes them unsuitable for ViT the way it is. There are multiple ways to tackle this issue:

- We can resize the input image by using interpolation techniques like nearest-neighbour or bilinear.
- We can change the first layer of the model's architecture to match the MNIST images. Both Pytorch and Tensorflow allow us to modify the model's architecture simply and adapt them to our needs.
- We can pad the MNIST images with zeros to increase their size to match the expected input size, but this approach can affect the model's performance drastically since zeros contain no meaningful information
  This issue happens with CNNs as well but they are generally more flexible and adaptable than ViT models since they handle a large variety of input sizes. Nevertheless, the solution to this problem is consistent for both CNNs and ViT models.

→ **Question 8-d:** Comment on the final results and provide some ideas on how to make transformer work on small datasets. You can take inspiration from some recent work.

— Training without the pre-trained weights resulted in an accuracy of 87.47%, while training with the weights led to a slightly improved accuracy of 89.21%. This difference can be attributed to several factors. For example, utilizing the pre-trained weights from ImageNet equips the model with prior knowledge of general patterns and features. As a result, the model doesn't need to learn these features from scratch using the MNIST dataset, which can expedite training and enhance overall performance.

However, our initial expectation was that pre-trained weights would yield significantly better accuracy, especially given the smaller size of the MNIST dataset compared to ImageNet. The reality turned out differently. This could be due to the

simplicity of the MNIST dataset, which exhibits limited variations and complexity. Training the Vision Transformer (ViT) from scratch can still yield impressive results, even on a relatively small dataset like MNIST. In such cases, the full capacity of pre-trained weights may not be fully harnessed. The full power of pre-trained models is often more pronounced in tasks involving more complex datasets, where they excel in capturing fine and intricate details.

Another potential issue might be the domain mismatch between the datasets. While ImageNet is extensive and diverse, it may not be highly suitable for capturing the unique features of MNIST. This mismatch could be a contributing factor to the limited boost in performance and accuracy when using pre-trained weights on MNIST

To enable Transformers to perform effectively on small datasets, we can employ data augmentation techniques such as random rotation, translation, and scaling. These methods introduce more diversity into our dataset. Additionally, we can leverage transfer learning and fine-tuning approaches to provide the model with prior knowledge, reducing the need to learn everything from scratch when dealing with a limited dataset (which could otherwise lead to overfitting).

We should experiment with Early Stopping techniques to prevent overfitting and choose a checkpoint at which the model performed the best. Furthermore, we can explore other regularization techniques such as dropout, L1, and L2 regularization to determine which one is most effective for our small datasets.

In cases where we require a larger dataset, we can consider using Generative Adversarial Networks (GANs) to generate synthetic images. If these generated images are of high quality and suitable, they can be incorporated into the Transformer model as additional data.