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

Homework 1-ab: Introduction to Neural Networks

Students: VU Anh Thu
LE Thi Minh Nguyet

In this lab, we will first implement a simple neural network with 3 layers while experimenting with different values of various hyperparameters to have an insight into their effects on the network's performance. After that, we test an effective machine learning model, the Support Vector Machine on the Circle dataset and compare its performance with that of our neural network.

Section 1.1: Supervised dataset

(Q1) Our supervised dataset $\left(x^{(i)}, y^{(i)}\right)_{1 \leq i \leq N}$ is divided into different sets: *train*, *test* and *val* (if necessary). They are served for different purposes:

- The *train* set is used for training our classifier $f_{\mathbf{w}}$ in order to get the optimal parameters $\mathbf{w}^*$ which minimize
$$\frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$
where $\mathcal{L}$ is a loss function evaluating the difference between the predictions $\hat{y}^{(i)} = f_{\mathbf{w}}(x^{(i)})$ and the actual values $y^{(i)}$.
- In the case where our model $f_{\mathbf{w}}$ has hyperparameters that are fixed during training, the validation set *val* comes into place to determine which values of the hyperparameters yield the best performance.
- After training, we know that our classifier performs well on the training set. It is also important to assess how well it can generalize, i.e. to test whether it works on new data as effectively as it does on the training set. To do this, we use the *test* set.

(Q2) The number of examples N a significant influence on various aspects of our model's learning:

- A larger dataset allows the model to learn more patterns from the data. Thus, it has a better chance of capturing the true underlying distribution, reducing overfitting, resulting in better generalization to unseen data.
- A large N means that it is possible to have a larger validation set to tune the hyperparameters. This makes the hyperparameter selection process more reliable, as it decreases the randomness in the selection that occurs with a small validation set.
- However, more data generally leads to higher computational costs to train the model.

Section 1.2: Network architcture (forward)

(Q3) It is essential to add activation functions between linear transformations for several reasons:

- Without activation functions, the relationship between inputs and outputs of the network is constrained to be linear (or affine) which might not be true in reality.
- The activation function in the output layer allows us to obtain the desired form of the output. For example, if we want the output to predict how probable a sample belongs to each class, we could use the sigmoid function σ in binary classification problems or the `SoftMax` in multi-class classification problems.

(Q4) In Figure 1, $n_x = 2$, $n_h = 4$, and $n_y = 2$.
In practice, n_x and n_y are determined based on the dimensions of the feature vectors $x^{(i)}$ and the targets $y^{(i)}$ of the dataset, respectively. As for n_h, it can be considered as a hyperparameter of our network, thus it can be selected using a validation set.

(Q5) The vector y is a one-hot encoded vector, with the only non-zero component indicating the actual class to which the sample belongs, while the vector $\hat{y}$ is the predicted probabilities for the sample belonging to each class.
Naturally, the output $\hat{y}$ is likely to be a one-hot encoding as its actual counterpart. However, in this case, we have a more informative prediction $\hat{y}$ because it not only predicts the class of the sample but also gives us the information about how much confidence we have in our prediction.

(Q6) As mentioned in (Q3), we choose the `SoftMax` function as the output activation function because we want the output $\hat{y}$ to represent the predicted probabilities that the corresponding sample belongs to each class. The `SoftMax` function can help achieve this by transforming the intermediate output $\tilde{y}$ into a probability distribution which is our desired form.

(Q7)

- $\tilde{h} = W_h x$
- $h = \tanh\Big(\tilde{h}\Big)$ where $\tanh\Big(\tilde{h}\Big) = \Big[\tanh\Big(\tilde{h}_1\Big), \ldots, \tanh\Big(\tilde{h}_{n_h}\Big)\Big]^T$
- $\tilde{y} = W_y h$
- $\hat{y} = \texttt{SoftMax}(\tilde{y})$ where

$$\texttt{SoftMax}(\tilde{y}) = \left[\frac{\exp(\tilde{y}_1)}{\sum_j \exp(\tilde{y}_j)}, \ldots, \frac{\exp\big(\tilde{y}_{n_y}\big)}{\sum_j \exp(\tilde{y}_j)}\right]^T$$

Section 1.3: Loss function

(Q8) In order to minimize the global loss, the $\hat{y}_i$ must vary such that the distance between it and y_i is as smallest as possible for both cross-entropy and squared error. Indeed,

- In the squared error case, it is clear that the closer $\hat{y}_i$ is to y_i, the smaller $(y_i - \hat{y}_i)^2$ becomes. As a result, the global loss will be decreased.
- For cross-entropy, if $y_i = 0$, regardless of how $\hat{y}_i$ varies, the term $-y_i \log(\hat{y}_i)$ is 0. On the other hand, if $y_i = 1$, $\hat{y}_i$ needs to be close to 1, i.e. close to y_i, so that $-y_i \log(\hat{y}_i)$ is small. Therefore, the global loss can be minimized.

(Q9)

- The cross entropy is better suited for classification tasks. This is due to the fact that the vector y_i has only one non-zero element, which results in fewer computations of gradients during the optimization process later on.

- The squared error is more suitable for regression tasks, since it reflects the difference between the predicted and actual values better than the cross-entropy.
  For example, the cross-entropy loss when $y_i = 0.2$ and $\hat{y}_i = 0.1$ is approximately -0.46, which is smaller than when $y_i = 0.2$ and $\hat{y}_i = 0.3$, which is approximately -0.24, even though the distance between them in both cases is the same and equals to 0.1. Therefore, we can see that the cross-entropy is not able to accurately show how $\hat{y}_i$ differs from y_i.

Section 1.4: Optimization algorithm

(Q10)
- **Classic Gradient Descent:** This is the most stable algorithm among the three since the gradient is computed over the entire dataset. However, it has the slowest convergence rate and a high computational cost for large datasets, as each update requires going through all samples.
- **Online Stochastic Gradient Descent:** In this method, only one sample is used for each update, which is like a double-edged sword. On the one hand, it is relatively cheap thanks to fewer computations. On the other hand, it is the least stable method because noise in the data can significantly affect the gradient.
- **Mini-Batch Stochastic Gradient Descent:** This is the most reasonable option to use in general, as it can accelerate GPU implementations, allowing to stay balance between computational efficiency and stability. Yet, it also requires appropriately tuning the batch size, which may be challenging in practice.

(Q11) The learning rate η determines the step size in the gradient descent algorithm. A high learning rate enables the model to learn quickly by taking large steps, but it may cause overshooting, resulting in slow convergence or, even worse, divergence of the model. On the other hand, a low learning rate ensures more stable progress, but if it is too small, the model is likely to converge very slowly.

(Q12)
- In the naive approach, the same gradient calculations are likely duplicated multiple times. For example, to compute the gradient of the *loss* with respect to a parameter of the i-th layer, we need to compute the gradient of the *loss* relative to the output of that layer, which was already calculated in the $i+1$-th layer (since the output of i-th layer is the input of the $i+1$-th layer). Therefore, as the number of layers in the network increases, the number of parameters may also increase, leading to the explosion of the number of computations.
- In the *backprop* algorithm, the gradient calculated with respect to the output of a layer will be saved and then reused in the calculations of the gradients with respect to the input and the parameters of this same layer. Thus, we need to travel along the network once only.

(Q13) It is required that the activation functions of the layers, as well as the loss function, are differentiable for the gradient decent algorithm to work, since it involves computing the gradients of the loss with respect to the weights.

(Q14) Given the *cross-entropy*

$$\ell = -\sum_i y_i \log(\hat{y}_i) \quad \text{where} \quad \hat{y}_i = \frac{\exp(\tilde{y}_i)}{\sum_j \exp(\tilde{y}_j)} \quad \text{for every} \quad i \in [[1, n_y]]$$

we have

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

Since $\sum_i y_i = 1$ due to one-hot encoding, we get

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

(Q15) Using (Q14), we have for each $i \in [[1, n_y]]$,

$$\frac{\partial \ell}{\partial \tilde{y}_i} = -y_i + \frac{\exp(\tilde{y}_i)}{\sum_j \exp(\tilde{y}_j)} = \hat{y}_i - y_i$$

Thus, the gradient of the *loss* w.r.t to $\tilde{y}$ is

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

(Q16)

- For each $i \in [[1, n_y]]$ and $j \in [[1, n_h]]$,

$$\frac{\partial \ell}{\partial W_{y,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial W_{y,ij}} \quad \text{where} \quad \frac{\partial \tilde{y}_k}{\partial W_{y,ij}} = \begin{cases} h_j & \text{if } k = i \\ 0 & \text{if } k \neq i \end{cases}$$

  Therefore,

$$\frac{\partial \ell}{\partial W_{y,ij}} = (\hat{y}_i - y_i)\, h_j \quad \Rightarrow \quad \nabla_{W_y} \ell = \nabla_{\tilde{y}} \ell h^T$$

- For each $i \in [[1, n_y]]$, note that $\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \mathbb{1}_{k=i}$ for every $k \in [[1, n_y]]$, we have

$$\frac{\partial \ell}{\partial b_{y,i}} = \sum_k \frac{\partial \ell}{\partial \tilde{y}_k}\frac{\partial \tilde{y}_k}{\partial b_{y,i}} = \frac{\partial \ell}{\partial \tilde{y}_i}$$

  Then we obtain $\nabla_{b_y} \ell = \nabla_{\tilde{y}} \ell$.

(Q17)

- For each $i \in [[1, n_h]]$,

$$\begin{aligned}
\frac{\partial \ell}{\partial \tilde{h}_i} = \sum_k \frac{\partial \ell}{\partial h_k} \frac{\partial h_k}{\partial \tilde{h}_i} = \frac{\partial \ell}{\partial h_i} \frac{\partial h_i}{\partial \tilde{h}_i} \\
&= \left(\sum_j \frac{\partial \ell}{\partial \tilde{y}_j} \frac{\partial \tilde{y}_j}{\partial h_i} \right) \frac{\partial h_i}{\partial \tilde{h}_i} \\
&= \left(\sum_j \frac{\partial \ell}{\partial \tilde{y}_j} W_{y,ji} \right) \frac{\left(e^{\tilde{h}_i} + e^{-\tilde{h}_i}\right)^2 - \left(e^{\tilde{h}_i} - e^{-\tilde{h}_i}\right)^2}{\left(e^{\tilde{h}_i} + e^{-\tilde{h}_i}\right)^2} \\
&= \left(\sum_j \frac{\partial \ell}{\partial \tilde{y}_j} W_{y,ji} \right) \left(1 - \tanh^2\left(\tilde{h}_i\right)\right)
\end{aligned}$$

  From that, we can derive

$$\nabla_{\tilde{h}} \ell = W_y^T \nabla_{\tilde{y}} \ell \odot \left(1 - \tanh\left(\tilde{h}\right)\right)$$

- For each $i \in [[1, n_h]]$ and $j \in [[1, n_x]]$,

$$\frac{\partial \ell}{\partial W_{h,ij}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial W_{h,ij}} = \frac{\partial \ell}{\partial \tilde{h}_i} \frac{\partial \tilde{h}_i}{\partial W_{h,ij}} = \frac{\partial \ell}{\partial \tilde{h}_i} x_j$$

  then $\nabla_{W_h} \ell = \nabla_{\tilde{h}} \ell x^T$.

- Finally, for each $i \in [[1, n_h]]$,

$$\frac{\partial \ell}{\partial b_{h,i}} = \sum_k \frac{\partial \ell}{\partial \tilde{h}_k} \frac{\partial \tilde{h}_k}{\partial b_{h,i}} = \frac{\partial \ell}{\partial \tilde{h}_i} \frac{\partial \tilde{h}_i}{\partial b_{h,i}} = \frac{\partial \ell}{\partial \tilde{h}_i}$$

  We obtain $\nabla_{b_h} \ell = \nabla_{\tilde{h}} \ell$.

Section 2: Implementation

(Q1) The Circle dataset contains 200 examples in the training set and 200 examples in the test set, divided into two classes. We first implemented a simple neural network to classify this dataset. The network consists of three layers: an input layer, a hidden layer, and an output layer. The input dimension (`nx`) is set to the number of features, which is 2; the hidden layer contains 10 units; and the output dimension (`ny`) corresponds to the number of classes, which is 2. The batch size (`Nbatch`) is set to 10, the learning rate (`eta`) is 0.03, and the model is trained for 150 iterations. We will later test different values for the learning rate, batch size, and hidden layer size to observe their effects on the network's performance. Despite the simplicity of this initial neural network, the results demonstrate strong performance for this particular dataset. The results are illustrated in the following figure:

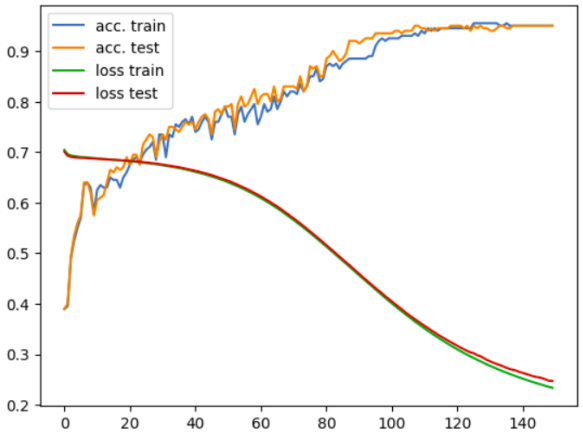


Figure 1: Learning curves for the initial neural network on the Circle dataset

It can be observed that both the training and testing loss decrease steadily over time, reaching 0.22 and 0.24 at the end of the training, respectively, while accuracy gradually improves, reaching 96% for training and 94.5% for testing.

We now present observations on how different hyperparameters affect performance:

- Learning rate:

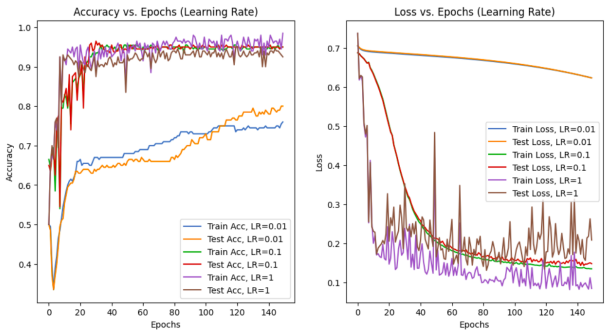


Figure 2: Learning curves for different learning rates

From the figure, we can see the effects of both too high ($\texttt{lr} = 1$) and too low ($\texttt{lr} = 0.01$ learning rates. For $\texttt{lr} = 0.01$, the training process was smooth, but the convergence rate was slow, leading to an accuracy of around 0.8 on the training set and 0.75 on the test set after 150 epochs. For $\texttt{lr} = 1$, while the algorithm still converged, it did so with instability and large fluctuations in both accuracy and loss. The learning rate of $\texttt{lr} = 0.1$ was the most optimal among three, showing both fast and stable convergence, with high accuracy on both train and test sets at the end of the training.

- Batch size:

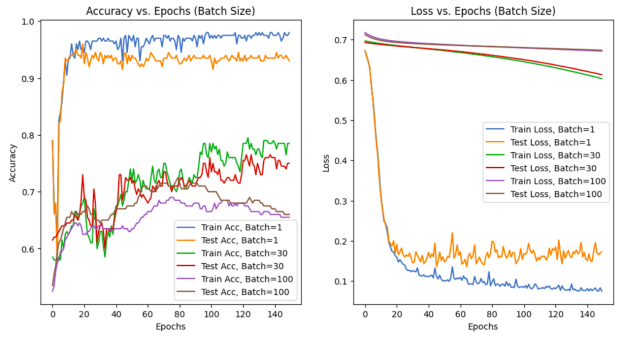


Figure 3: Learning curves for different batch sizes

We can see that large batch sizes can lead to a stable but slow convergence of the model. For batch sizes of 30 and 100, the convergence was quite smooth in both cases, with a faster convergence observed with a batch size of 30. Meanwhile, when using a very small batch size, such as 1, the convergence became more erratic, with noticeable fluctuations. Despite this, the model converged much faster and achieved significantly better accuracy.

- Number of hidden units:

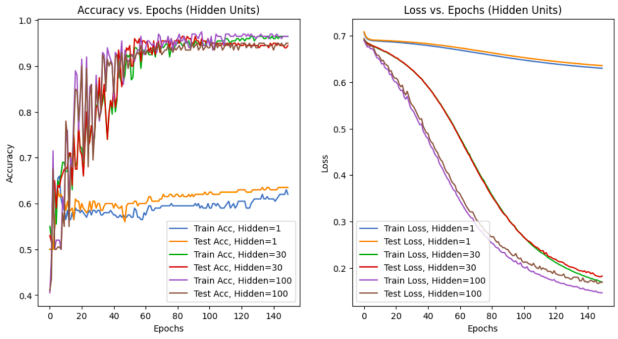


Figure 4: Learning curves for different numbers of hidden units

We observed that using a small number of hidden units, such as 1, resulted in a slow convergence, as the model's architecture was not rich enough to allow faster learning within the same number of iterations. Meanwhile, using a large number of hidden units, such as 100, caused highly fluctuating behaviors in the early stages of training, likely due to the increased complexity of the model. Moreover, this configuration could lead to a slight overfitting, as the test accuracy with 100 hidden units was slightly lower than with 30 units, while the opposite trend was observed in training accuracy.

We then applied our neural network with 3 layers with 100 hidden units on the MNIST dataset, using a learning rate of 0.03, a batch size of 100, and running for 150 iterations. This combination of a moderate batch size and learning rate, along with a simple hidden layer structure, proved effective, leading to a significant reduction in loss and achieving high accuracy (around 90%) at the end of the training. The results are presented below:

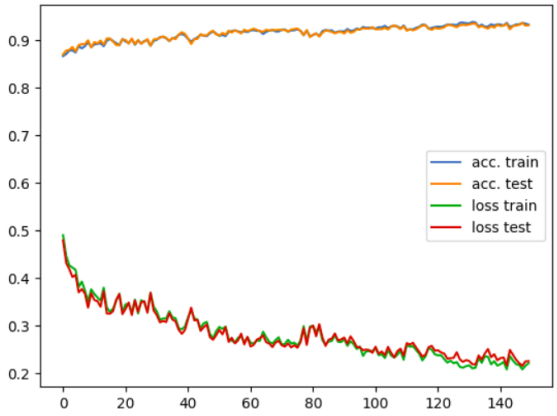


Figure 5: Learning curves for the MNIST dataset

Section 2.6: Bonus: SVM

- First, we used a linear SVM model with a regularization parameter `C` $= 1.0$ and obtained an accuracy of 53.00% on the test set:

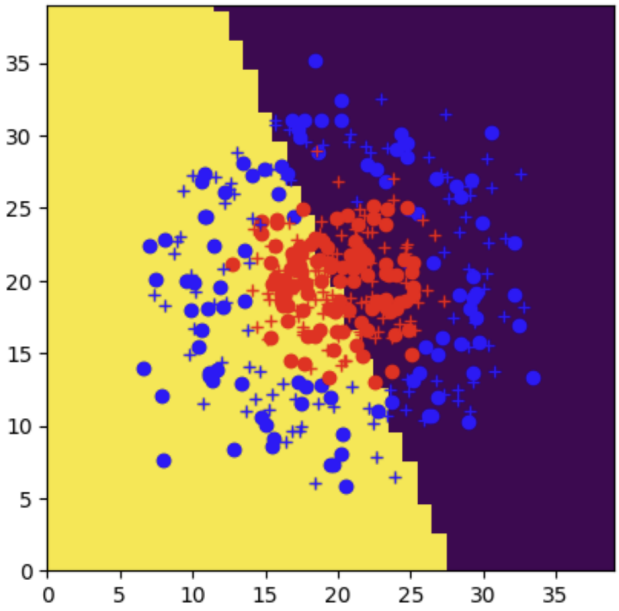
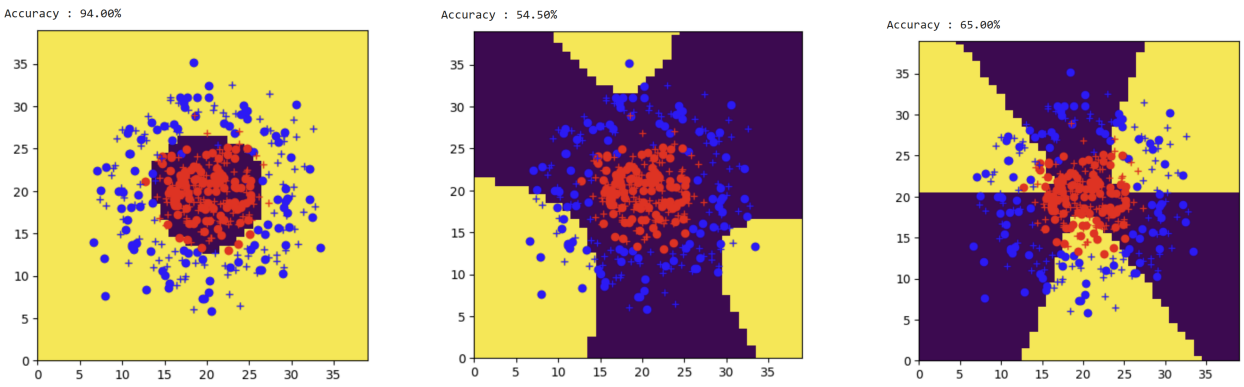


Figure 6: Linear SVM

  Since a linear SVM tries to find an optimal hyperplane to separate two classes, it does not work well on this circle dataset because the classes are not linearly separable.

- Second, we applied a nonlinear SVM model with different kernels: Gaussian with an adapted `gamma` $= \frac{1}{\texttt{n\_features} \times \texttt{X\_train.var()}}$ (where `X_train.var()` is the average of the variances calculated across each feature), polynomial of degree 3 and sigmoid, all using the same regularization parameter, `C` $= 1.0$ and obtained the following results:



(a) Gaussian kernel **(b)** Polynomial kernel with degree 3 **(c)** Sigmoid kernel

Figure 7: Non linear SVM

As we can see from the results, the SVM model with Gaussian kernel yields the best performance. This can be explained by the fact that polynomial and sigmoid kernels try to find decision boundaries along polynomial-shaped and sigmoid-shaped curve, which do not align with the circular nature of this dataset. Meanwhile, Gaussian kernel provides localized decision boundaries controled by the parameter `gamma`, which allows it to capture the circular pattern of the data more effectively. Moreover, the SVM with Gaussian kernel performs as well as our initial three-layer neural network in which the hidden layer consists of 10 neurons.

- Finally, we tested a SVM model with Gaussian kernel for different values of the regularization parameter `C` and obtained the following results:

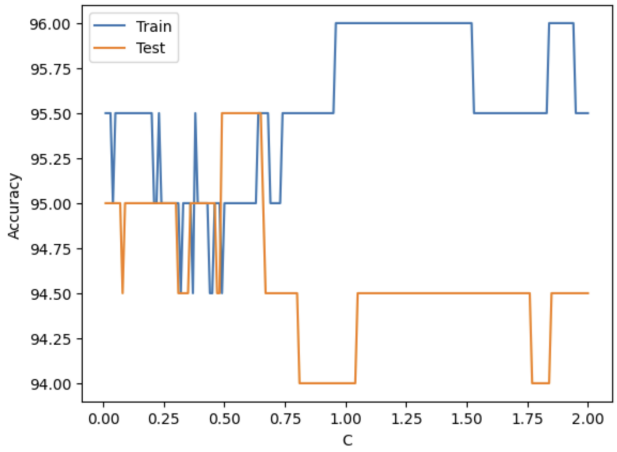


Figure 8: Accuracy curves with different values of `C`

The parameter `C` controls the strength of the regularization imposed on the model in such a way that the complexity of the model is proportional to `C`. In other words, the larger the value of `C`, the more complex the model becomes. Therefore, if C becomes larger, the model is more likely to overfit, leading to higher training accuracy but lower test accuracy, as can be seen from the figure.