实验 1

1.

```
def sum_of_list(numbers):
    return sum(numbers)

numbers = [1, 2, 3, 4, 5]
result = sum_of_list(numbers)
print("The sum of the list is:", result)
```

```
The sum of the list is: 15

Process finished with exit code 0
```

2.

```
def unique_elements(input_list):
    return list(set(input_list))

input_list = [1, 3, 3, 2, 3, 4, 3, 4, 5]
result = unique_elements(input_list)
print("The list with unique elements is:", result)
```

```
The list with unique elements is: [1, 2, 3, 4, 5]

Process finished with exit code 0
```

3.

```
def is_palindrome(input_string):
    cleaned_string = input_string.replace(" ", "").lower()
    return cleaned_string == cleaned_string[::-1]

input_string = "nurses run"
result = is_palindrome(input_string)
print(f"Is the string '{input_string}' a palindrome? {result}")
```

```
Is the string 'nurses run' a palindrome? True

Process finished with exit code 0
```

4.

```
import numpy as np

def real_and_imaginary_parts(complex_array):
    real_parts = np.real(complex_array)
    imaginary_parts = np.imag(complex_array)
    return np.column_stack((real_parts, imaginary_parts))

complex_array = np.array([1.00000000 + 0.j, 0.70710678 + 0.70710678j])
result = real_and_imaginary_parts(complex_array)
print("Real and Imaginary parts of the array:")
print(result)
```

```
Real and Imaginary parts of the array:
[[1.         0.        ]
 [0.70710678 0.70710678]]
```

5.

```
import numpy as np
def add_ternary(a, b):
    a_decimal = int(a, 3)
    b_decimal = int(b, 3)

    sum_decimal = a_decimal + b_decimal

    return np.base_repr(sum_decimal, base=3)



a = '12'
b = '21'
result = add_ternary(a, b)
print(f"The sum of {a} and {b} in ternary is: {result}")
```

```
The sum of 12 and 21 in ternary is: 110
```

6.

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None



def add_two_numbers(l1, l2):
    dummy_head = ListNode(0)
    current = dummy_head
```

```
    carry = 0

    while l1 or l2 or carry:
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        total = val1 + val2 + carry
        carry = total // 10
        current.next = ListNode(total % 10)
        current = current.next

        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    return dummy_head.next


l1 = ListNode(2)
l1.next = ListNode(4)
l1.next.next = ListNode(3)

l2 = ListNode(5)
l2.next = ListNode(6)
l2.next.next = ListNode(4)

result = add_two_numbers(l1, l2)

output = []
while result:
    output.append(str(result.val))
    result = result.next
print(" -> ".join(output))
```

```
7 -> 0 -> 8
```

7.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                # Swap the elements
```

```
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
#example
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array:", sorted_arr)
```

```
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

8.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

    return arr
```

```
# Example
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr = merge_sort(arr)
print("Sorted array:", sorted_arr)
```

```
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```

9.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Example
arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print("Sorted array:", sorted_arr)
```

```
Sorted array: [1, 1, 2, 3, 6, 8, 10]
```

10.

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
```

```
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

    return arr

# Example
arr = [12, 11, 13, 5, 6, 7]
sorted_arr = heap_sort(arr)
print("Sorted array:", sorted_arr)
```

```
Sorted array: [1, 1, 2, 3, 6, 8, 10]
```

11.

```
import torch
import torch.nn as nn
import torch.optim as optim

x_train = torch.randn(100, 1) * 10
y_train = x_train + 3 * torch.randn(100, 1)

model = nn.Linear(1, 1)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()
    y_pred = model(x_train)

    loss = criterion(y_pred, y_train)

    loss.backward()

    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

print(f"Learned weight: {model.weight.item()}")
print(f"Learned bias: {model.bias.item()}")
```

```
Epoch 100/1000, Loss: 22345.90234375
Epoch 200/1000, Loss: 11029492.0
Epoch 300/1000, Loss: 5446113280.0
Epoch 400/1000, Loss: 2689177747456.0
Epoch 500/1000, Loss: 1327854503591936.0
Epoch 600/1000, Loss: 6.556622857038725e+17
Epoch 700/1000, Loss: 3.237491756972481e+20
Epoch 800/1000, Loss: 1.5985943009957145e+23
Epoch 900/1000, Loss: 7.89346509090713e+25
Epoch 1000/1000, Loss: 3.897577786910861e+28
Learned weight: -20204914475008.0
Learned bias: -166915391488.0
```

12.

```
import torch
import torch.nn as nn
import torch.optim as optim

x_train = torch.randn(100, 1) * 10
y_train = (x_train > 0).float()
model = nn.Sequential(
    nn.Linear(1, 1),
    nn.Sigmoid()
)

criterion = nn.BCELoss()

optimizer = optim.SGD(model.parameters(), lr=0.01)

epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()

    y_pred = model(x_train)

    loss = criterion(y_pred.squeeze(), y_train)

    loss.backward()

    optimizer.step()
```

```
    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

print(f"Learned weight: {model[0].weight.item()}")
print(f"Learned bias: {model[0].bias.item()}")
```

```
Epoch 100/1000, Loss: 36713066496.0
Epoch 200/1000, Loss: 9.697117292500668e+19
Epoch 300/1000, Loss: 2.5613237442919142e+29
Epoch 400/1000, Loss: inf
Epoch 500/1000, Loss: inf
Epoch 600/1000, Loss: inf
Epoch 700/1000, Loss: inf
Epoch 800/1000, Loss: nan
Epoch 900/1000, Loss: nan
Epoch 1000/1000, Loss: nan
Learned weight: nan
Learned bias: nan
```

13.

```
import torch
import torch.nn as nn
import torch.optim as optim

x_train = torch.randn(100, 1) * 10
y_train = (x_train > 0).float() * 2 - 1

model = nn.Linear(1, 1)

def hinge_loss(y_pred, y_true):
    return torch.mean(torch.maximum(torch.zeros_like(y_pred), 1 - y_true * y_pred))

optimizer = optim.SGD(model.parameters(), lr=0.01)

epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()

    y_pred = model(x_train)

    loss = hinge_loss(y_pred.squeeze(), y_train)
```

```
        loss.backward()

        optimizer.step()

        if (epoch + 1) % 100 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")

print(f"Learned weight: {model.weight.item()}")
print(f"Learned bias: {model.bias.item()}")
```

```
Epoch 100/1000, Loss: 1.0109530687332153
Epoch 200/1000, Loss: 1.0103600025177002
Epoch 300/1000, Loss: 1.0098551511764526
Epoch 400/1000, Loss: 1.0094382762908936
Epoch 500/1000, Loss: 1.0090067386627197
Epoch 600/1000, Loss: 1.0085898637771606
Epoch 700/1000, Loss: 1.0081582069396973
Epoch 800/1000, Loss: 1.0077413320541382
Epoch 900/1000, Loss: 1.007309913635254
Epoch 1000/1000, Loss: 1.0068929195404053
Learned weight: -0.027268262580037117
Learned bias: -0.3632980287075043
```

14.
(1)

```
import torch
import torch.optim as optim

x_train = torch.randn(100, 1) * 10
y_train = (x_train > 0).float() * 2 - 1

model = torch.nn.Linear(1, 1)


def hinge_loss(y_pred, y_true):
    return torch.mean(torch.maximum(torch.zeros_like(y_pred), 1 - y_true * y_pred))

optimizer = optim.SGD(model.parameters(), lr=0.01)

epochs = 1000
```

```
for epoch in range(epochs):
    optimizer.zero_grad()

    y_pred = model(x_train)
    loss = hinge_loss(y_pred.squeeze(), y_train)

    l2_penalty = torch.norm(model.weight, p='fro')

    total_loss = loss + 0.01 * l2_penalty

    total_loss.backward()

    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss.item()}")

print(f"Learned weight: {model.weight.item()}")
print(f"Learned bias: {model.bias.item()}")
```

```
Epoch 100/1000, Loss: 0.9907679557800293
Epoch 200/1000, Loss: 0.9902843236923218
Epoch 300/1000, Loss: 0.9898843765258789
Epoch 400/1000, Loss: 0.9894845485687256
Epoch 500/1000, Loss: 0.9890847206115723
Epoch 600/1000, Loss: 0.9886847138404846
Epoch 700/1000, Loss: 0.9882848262786865
Epoch 800/1000, Loss: 0.9878847599029541
Epoch 900/1000, Loss: 0.9874847531318665
Epoch 1000/1000, Loss: 0.9870848059654236
Learned weight: 5.207854337641038e-05
Learned bias: 0.645963728427887
```

(2)

```
import torch
import torch.optim as optim

x_train = torch.randn(100, 1) * 10
y_train = (x_train > 0).float() * 2 - 1

model = torch.nn.Linear(1, 1)
```

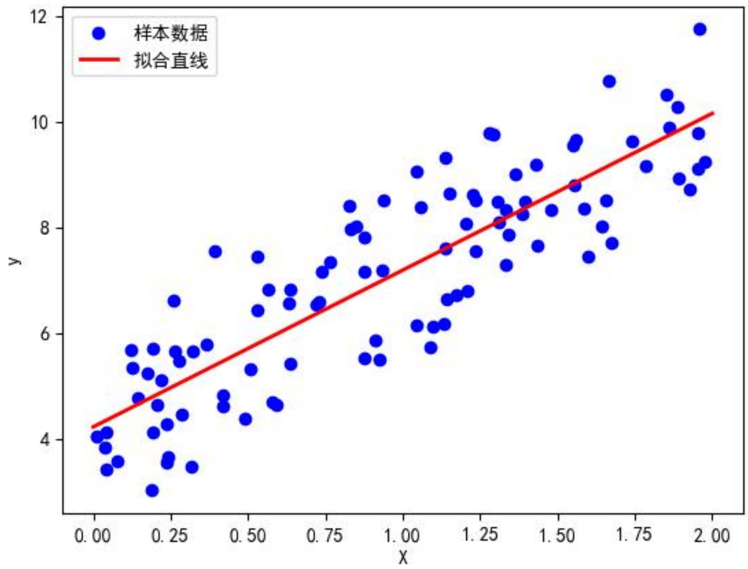
```
def hinge_loss(y_pred, y_true):
    return torch.mean(torch.maximum(torch.zeros_like(y_pred), 1 - y_true * y_pred))


optimizer = optim.SGD(model.parameters(), lr=0.01)

epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()

    y_pred = model(x_train)
    loss = hinge_loss(y_pred.squeeze(), y_train)

    l2_penalty = torch.sqrt(torch.sum(model.weight ** 2))
    total_loss = loss + 0.01 * l2_penalty


    total_loss.backward()

    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss.item()}")

print(f"Learned weight: {model.weight.item()}")
print(f"Learned bias: {model.bias.item()}")
```

```
Epoch 100/1000, Loss: 1.0391082763671875
Epoch 200/1000, Loss: 1.0352703332901
Epoch 300/1000, Loss: 1.0314338207244873
Epoch 400/1000, Loss: 1.0276097059249878
Epoch 500/1000, Loss: 1.023779273033142
Epoch 600/1000, Loss: 1.0199412107467651
Epoch 700/1000, Loss: 1.0161030292510986
Epoch 800/1000, Loss: 1.0122754573822021
Epoch 900/1000, Loss: 1.008449912071228
Epoch 1000/1000, Loss: 1.004662036895752
Learned weight: 0.052991293370723724
Learned bias: 0.1065046563744545
```

15.

（1）线性回归

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']    #  或  'Microsoft YaHei'


# 生成线性数据：y = 2x + 1，加上一些噪声
np.random.seed(0)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# 创建并训练线性回归模型
lin_reg = LinearRegression()
lin_reg.fit(X, y)

print("线性回归模型的截距：", lin_reg.intercept_[0])
print("线性回归模型的系数：", lin_reg.coef_[0][0])

plt.scatter(X, y, color='blue', label='样本数据')
X_new = np.array([[0], [2]])
y_predict = lin_reg.predict(X_new)
plt.plot(X_new, y_predict, color='red', linewidth=2, label='拟合直线')
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```



（2）逻辑回归

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 加载鸢尾花数据集
iris = load_iris()
X, y = iris.data, (iris.target == 0).astype(int)  # 只分类是否为“类别 0”

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 训练逻辑回归模型
log_model = LogisticRegression()
log_model.fit(X_train, y_train)

# 进行预测
y_pred = log_model.predict(X_test)

# 评估模型
accuracy = accuracy_score(y_test, y_pred)
print(f"分类准确率: {accuracy}")
```

```
分类准确率: 1.0
```

（3）SVM

```
from sklearn.svm import SVC

# 训练 SVM 分类器
svm_model = SVC(kernel='linear')  # 选择 'linear' 线性核，也可以用 'rbf' 高斯核等
svm_model.fit(X_train, y_train)

# 进行预测
y_pred_svm = svm_model.predict(X_test)

# 评估模型
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f"SVM 分类准确率: {accuracy_svm}")
```

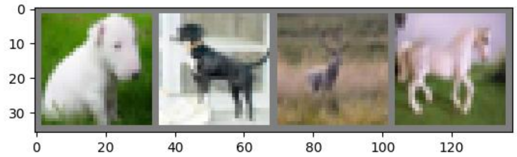
```
SVM 分类准确率: 1.0
```

16.

```
import torch
import torchvision
```

```
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

def main():
    # 定义图像转换，将 PIL 图像转换为 tensor，并进行归一化处理
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # 均值与标准差
    ])

    # 下载 CIFAR-10 数据集（训练集）
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                            download=True, transform=transform)

    # 使用 DataLoader 加载数据
    trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                              shuffle=True, num_workers=2)

    # CIFAR-10 数据集的类别名称
    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

    # 定义函数来显示图像
    def imshow(img):
        # 将 tensor 反归一化
        img = img / 2 + 0.5
        npimg = img.numpy()
        # 转换维度：C x H x W -> H x W x C
        plt.imshow(np.transpose(npimg, (1, 2, 0)))
        plt.show()

    # 获取随机批次的图像
    dataiter = iter(trainloader)
    images, labels = next(dataiter)  # 使用 next() 获取数据

    # 显示图像及对应的类别标签
    imshow(torchvision.utils.make_grid(images))
    print(' '.join('%5s' % classes[labels[j]] for j in range(4)))

if __name__ == '__main__':
    main()
```

```
Files already downloaded and verified
  dog   dog  deer horse
```



17.

```python
import torch
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
import torchvision.datasets as datasets

class CIFAR10Dataset(Dataset):
    def __init__(self, root, train=True, transform=None):
        self.dataset = datasets.CIFAR10(root=root, train=train, download=True)
        self.transform = transform if transform else transforms.ToTensor()

    def __len__(self):
        """返回数据集的大小"""
        return len(self.dataset)

    def __getitem__(self, idx):
        """获取索引 idx 处的图像及其标签"""
        image, label = self.dataset[idx]
        if self.transform:
            image = self.transform(image)
        return image, label

# 测试 Dataset
dataset = CIFAR10Dataset(root="./data", train=True, transform=transforms.ToTensor())

# 转换成 DataLoader
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

```
# 取出一个 batch 进行测试
images, labels = next(iter(dataloader))
print(f"Batch size: {images.shape}")   # 输出形状 (32, 3, 32, 32)
print(f"Labels: {labels}")
```

```
Files already downloaded and verified
Batch size: torch.Size([32, 3, 32, 32])
Labels: tensor([5, 9, 8, 0, 4, 2, 5, 9, 6, 6, 0, 3, 6, 3, 9, 2, 6, 4, 3, 4, 9, 5, 2, 7,
        0, 8, 7, 1, 8, 9, 9, 3])
```

18.

常用的变换及其说明：

Resize: 调整图像大小。

CenterCrop: 中心裁剪图像。

RandomCrop: 随机裁剪图像。

RandomHorizontalFlip: 随机水平翻转图像。

ToTensor: 将图像转换为 Tensor 格式。

Normalize: 规范化图像（减去均值并除以标准差）。

```
import torch
from torchvision import transforms
from PIL import Image
import torchvision.datasets as datasets

# 定义变换
transform = transforms.Compose([
    transforms.Resize((256, 256)),                 # 调整图像大小到 256x256
    transforms.RandomCrop(224),                    # 随机裁剪为 224x224
    transforms.RandomHorizontalFlip(),             # 随机水平翻转
    transforms.ToTensor(),                         # 将图像转为 Tensor 格式
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),   # 规范化
])

# 加载图像数据集
dataset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

# 查看变换后的图像
image, label = dataset[0]   # 取第一个样本
print("Image shape:", image.shape)
print("Label:", label)
```

```
Files already downloaded and verified
Image shape: torch.Size([3, 224, 224])
Label: 6
```

19.

```python
import time
import torch
import torchvision
import torchvision.transforms as transforms

def main():
    # 定义数据预处理：仅转换为 tensor
    transform = transforms.Compose([
        transforms.ToTensor()
    ])

    # 下载并加载 CIFAR-10 数据集（训练集）
    dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

    # 定义要测试的参数配置
    batch_sizes = [1, 4, 64, 1024]
    num_workers_list = [0, 1, 4, 16]
    pin_memory_options = [False, True]

    print("开始测试加载时间...\n")
    for batch_size in batch_sizes:
        for num_workers in num_workers_list:
            for pin_memory in pin_memory_options:
                # 构造 DataLoader
                dataloader = torch.utils.data.DataLoader(
                    dataset,
                    batch_size=batch_size,
                    shuffle=False,
                    num_workers=num_workers,
                    pin_memory=pin_memory
                )

                # 记录开始时间
                start_time = time.time()
                # 遍历整个数据集（一个 epoch）
                for _ in dataloader:
                    pass
                elapsed = time.time() - start_time

                config = f"batch_size={batch_size}, num_workers={num_workers}, pin_memory={pin_memory}"
                print(f"{config} -> 加载耗时: {elapsed:.4f} 秒")
```

```
if __name__ == '__main__':
    main()
```

```
batch_size=1, num_workers=0, pin_memory=False -> 加载耗时: 3.8834 秒
batch_size=1, num_workers=0, pin_memory=True -> 加载耗时: 3.9382 秒
batch_size=1, num_workers=1, pin_memory=False -> 加载耗时: 17.6623 秒
batch_size=1, num_workers=1, pin_memory=True -> 加载耗时: 17.9278 秒
batch_size=1, num_workers=4, pin_memory=False -> 加载耗时: 15.1477 秒
batch_size=1, num_workers=4, pin_memory=True -> 加载耗时: 15.0977 秒
batch_size=1, num_workers=16, pin_memory=False -> 加载耗时: 40.6001 秒
batch_size=1, num_workers=16, pin_memory=True -> 加载耗时: 39.5142 秒
batch_size=4, num_workers=0, pin_memory=False -> 加载耗时: 2.7961 秒
batch_size=4, num_workers=0, pin_memory=True -> 加载耗时: 2.7971 秒
batch_size=4, num_workers=1, pin_memory=False -> 加载耗时: 7.6210 秒
batch_size=4, num_workers=1, pin_memory=True -> 加载耗时: 7.5929 秒
batch_size=4, num_workers=4, pin_memory=False -> 加载耗时: 10.0628 秒
batch_size=4, num_workers=4, pin_memory=True -> 加载耗时: 10.0753 秒
batch_size=4, num_workers=16, pin_memory=False -> 加载耗时: 34.3441 秒
batch_size=4, num_workers=16, pin_memory=True -> 加载耗时: 34.2145 秒
batch_size=64, num_workers=0, pin_memory=False -> 加载耗时: 2.2315 秒
batch_size=64, num_workers=0, pin_memory=True -> 加载耗时: 2.2826 秒
batch_size=64, num_workers=1, pin_memory=False -> 加载耗时: 4.5408 秒
batch_size=64, num_workers=1, pin_memory=True -> 加载耗时: 4.5407 秒
batch_size=64, num_workers=4, pin_memory=False -> 加载耗时: 8.7924 秒
batch_size=64, num_workers=4, pin_memory=True -> 加载耗时: 8.7653 秒
batch_size=64, num_workers=16, pin_memory=False -> 加载耗时: 32.4927 秒
batch_size=64, num_workers=16, pin_memory=True -> 加载耗时: 32.7306 秒
batch_size=1024, num_workers=0, pin_memory=False -> 加载耗时: 6.2150 秒
batch_size=1024, num_workers=0, pin_memory=True -> 加载耗时: 5.9896 秒
```

```
batch_size=1024, num_workers=0, pin_memory=False -> 加载耗时: 6.2150 秒
batch_size=1024, num_workers=0, pin_memory=True -> 加载耗时: 5.9896 秒
batch_size=1024, num_workers=1, pin_memory=False -> 加载耗时: 5.4252 秒
batch_size=1024, num_workers=1, pin_memory=True -> 加载耗时: 4.1933 秒
batch_size=1024, num_workers=4, pin_memory=False -> 加载耗时: 8.7335 秒
batch_size=1024, num_workers=4, pin_memory=True -> 加载耗时: 8.7954 秒
batch_size=1024, num_workers=16, pin_memory=False -> 加载耗时: 32.5220 秒
batch_size=1024, num_workers=16, pin_memory=True -> 加载耗时: 32.5405 秒
```

20.

```
import torch
import torchvision
import torchvision.transforms as transforms

# 加载 CIFAR-10 训练集（不进行归一化）
dataset = torchvision.datasets.CIFAR10(root="./data", train=True, download=True, transform=transforms.ToTensor())

# 获取所有图像数据
```

```
data_loader = torch.utils.data.DataLoader(dataset, batch_size=10000, shuffle=False)
images, _ = next(iter(data_loader))    # 获取所有图片 (10000, 3, 32, 32)

# 计算均值和标准差
mean = images.mean(dim=[0, 2, 3])    # 在通道（C）维度计算均值
std = images.std(dim=[0, 2, 3])    # 在通道（C）维度计算标准差

print(f"Mean (R, G, B): {mean.tolist()}")
print(f"Std (R, G, B): {std.tolist()}")
```

```
Mean (R, G, B): [0.4934569299221039, 0.4833766520023346, 0.4471793472766876]
Std (R, G, B): [0.24762117862701416, 0.24458514153957367, 0.2626110017299652]
```

21.

```
from PIL import Image

def image_to_char_art(image_path, output_txt, new_width=100):

    # 定义用于转换灰度值的字符列表（由暗到亮）
    # 列表中的字符越靠前，表示像素越暗；越靠后，表示像素越亮
    char_list = "@%#*+=-:. "
    num_chars = len(char_list)

    # 打开图像
    img = Image.open(image_path)

    # 调整图像大小：根据新宽度，保持纵横比
    width, height = img.size
    aspect_ratio = height / width
    # 调整后的高度可以适当缩小（因为字符的纵横比例通常不为 1）
    new_height = int(aspect_ratio * new_width * 0.55)
    img = img.resize((new_width, new_height))

    # 将图像转换为 RGB 模式
    img = img.convert("RGB")

    # 初始化字符画字符串
    char_art = ""

    # 遍历图像每个像素，计算灰度值，并映射到字符上
    for y in range(new_height):
        for x in range(new_width):
            r, g, b = img.getpixel((x, y))
            # 根据公式计算灰度值
            gray = 0.2126 * r + 0.7152 * g + 0.0722 * b
```

```
            # 将灰度值映射到字符列表中，灰度值范围是 0-255
            # 计算对应字符的索引：越暗对应列表前面的字符，越亮对应列表后面的字符
            char_index = int((gray / 255) * (num_chars - 1))
            char_art += char_list[char_index]
        # 每行结束后添加换行符
        char_art += "\n"

    # 将字符画保存到文件
    with open(output_txt, "w", encoding="utf-8") as f:
        f.write(char_art)

    print("字符画已保存到:", output_txt)

# 示例使用：
if __name__ == "__main__":
    image_path = "input.jpg"
    output_txt = "char_art.txt"
    image_to_char_art(image_path, output_txt, new_width=100)
```

输入和输出分别为：

Input.jpg　char_art.txt

22.

```
import numpy as np

# Part 1：笛卡尔坐标转换为极坐标
def cartesian_to_polar(cart_coords):

    x = cart_coords[:, 0]
    y = cart_coords[:, 1]
    r = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    polar_coords = np.stack((r, theta), axis=1)
    return polar_coords

cart_coords = np.random.rand(10, 2)   # 随机生成 10×2 矩阵
polar_coords = cartesian_to_polar(cart_coords)
print("原始笛卡尔坐标:\n", cart_coords)
print("转换后的极坐标:\n", polar_coords)


# Part 2：创建对称矩阵子类
class SymmetricArray(np.ndarray):
    def __new__(cls, input_array):
```

```
        # 将输入数组转为 numpy 数组，并生成副本，再转换为我们的子类类型
        obj = np.asarray(input_array).copy().view(cls)
        # 如果输入数组不对称，则以 (A + A.T) / 2 强制对称化
        obj = (obj + obj.T) / 2
        # 重新转换为子类类型
        obj = np.asarray(obj).view(cls)
        return obj

    def __array_finalize__(self, obj):
        if obj is None:
            return

A = np.random.rand(4, 4)
sym_A = SymmetricArray(A)
print("\n 原始数组 A:\n", A)
print("对称化后的 sym_A:\n", sym_A)
print("sym_A 是否对称：", np.allclose(sym_A, sym_A.T))


# Part 3：计算点到直线的距离
def point_to_line_distance(P0, P1, P):
    n = P0.shape[0]
    m = P.shape[0]
    distances = np.zeros((n, m))

    for i in range(n):
        v = P1[i] - P0[i]
        norm_v = np.linalg.norm(v)
        # 防止除以 0
        if norm_v == 0:
            distances[i, :] = 0
            continue
        # 对于每个点，计算 w = P - P0[i]
        w = P - P0[i]    # shape (m,2)
        # 计算叉积的绝对值： |v[0]*w[:,1] - v[1]*w[:,0]|
        cross_mag = np.abs(v[0] * w[:, 1] - v[1] * w[:, 0])
        distances[i, :] = cross_mag / norm_v
    return distances

# 定义两条直线：
# 第 1 条直线：起点 (0,0)，终点 (1,0)
# 第 2 条直线：起点 (1,1)，终点 (2,1)
P0 = np.array([[0, 0], [1, 1]])
P1 = np.array([[1, 0], [2, 1]])
```

```
# 定义三个测试点
P = np.array([[0, 1], [1, 2], [2, 2]])
distances = point_to_line_distance(P0, P1, P)
print("\n 每个点到每条直线的距离:\n", distances)
```

```
原始笛卡尔坐标:
 [[0.44143129 0.27801206]
 [0.37039331 0.36887412]
 [0.12169281 0.01908343]
 [0.75125849 0.76976462]
 [0.75615045 0.00688137]
 [0.82347668 0.33450064]
 [0.40005118 0.66951547]
 [0.48643841 0.26368433]
 [0.58276372 0.87448422]
 [0.15708298 0.8379732 ]]
转换后的极坐标:
 [[0.52168217 0.56204133]
 [0.52274212 0.78334318]
 [0.12318002 0.15554964]
 [1.07560536 0.79756446]
 [0.75618176 0.00910028]
 [0.88882199 0.38584435]
 [0.77993071 1.03219982]
 [0.55330982 0.49673559]
 [1.05087402 0.98297248]
 [0.85256914 1.38549102]]
```

```
原始数组 A:
 [[0.73557823 0.8260878  0.29000557 0.37381723]
 [0.97091224 0.54027814 0.79045518 0.6124517 ]
 [0.63722507 0.66753052 0.79645806 0.70404292]
 [0.18427829 0.84611333 0.79754198 0.20470114]]
对称化后的 sym_A:
 [[0.73557823 0.89850002 0.46361532 0.27904776]
 [0.89850002 0.54027814 0.72899285 0.72928251]
 [0.46361532 0.72899285 0.79645806 0.75079245]
 [0.27904776 0.72928251 0.75079245 0.20470114]]
sym_A 是否对称： True

每个点到每条直线的距离:
 [[1. 2. 2.]
 [0. 1. 1.]]
```

23.

```
import math

def bilinear_interpolation(A, point):
    x, y = point
    # 将 1-索引转换为 0-索引
    x0 = x - 1
    y0 = y - 1

    # 计算左上角像素的索引和偏移量
    i = int(math.floor(x0))
    j = int(math.floor(y0))
    dx = x0 - i
    dy = y0 - j

    # 若恰好落在整数位置，则直接返回对应的值
    if dx == 0 and dy == 0:
        return A[i][j]

    # 获取四个邻近像素值
    # 注意：这里假设输入坐标不会超出矩阵边界
    Q11 = A[i][j]
    Q12 = A[i][j+1]
    Q21 = A[i+1][j]
```

```
        Q22 = A[i+1][j+1]

        # 双线性插值公式
        interpolated_value = (Q11 * (1 - dx) * (1 - dy) +
                              Q12 * (1 - dx) * dy +
                              Q21 * dx * (1 - dy) +
                              Q22 * dx * dy)

        # 返回整数值（四舍五入），也可返回浮点数
        return round(interpolated_value)

# 测试样例
A = (
    (110, 120, 130),
    (210, 220, 230),
    (310, 320, 330)
)

result1 = bilinear_interpolation(A, (1, 1))
result2 = bilinear_interpolation(A, (2.5, 2.5))

print("BilinearInterpolation(A, (1, 1)) =", result1)      # 期望输出 110
print("BilinearInterpolation(A, (2.5, 2.5)) =", result2)  # 期望输出 275
```

```
BilinearInterpolation(A, (1, 1)) = 110
BilinearInterpolation(A, (2.5, 2.5)) = 275
```

24.

```
import itertools

def cartesian_product(*vectors):
    # 使用 itertools.product 生成所有组合
    product_result = list(itertools.product(*vectors))
    # 将每个组合转换为列表形式
    return [list(item) for item in product_result]

# 示例使用：
v1 = [1, 2, 3]
v2 = [4, 5]
v3 = [6, 7]

result = cartesian_product(v1, v2, v3)
print("笛卡尔积结果:")
```

```
for item in result:
    print(item)
```

```
笛卡尔积结果:
[1, 4, 6]
[1, 4, 7]
[1, 5, 6]
[1, 5, 7]
[2, 4, 6]
[2, 4, 7]
[2, 5, 6]
[2, 5, 7]
[3, 4, 6]
[3, 4, 7]
[3, 5, 6]
[3, 5, 7]
```

25.

```
import numpy as np

def extract_subarray(array, shape, fill, position):
    # Calculate the dimensions of the desired subarray
    subarray_rows, subarray_cols = shape
    row_center, col_center = position

    row_start = row_center - subarray_rows // 2
    row_end = row_start + subarray_rows
    col_start = col_center - subarray_cols // 2
    col_end = col_start + subarray_cols

    subarray = np.full(shape, fill, dtype=array.dtype)

    original_row_start = max(row_start, 0)
    original_row_end = min(row_end, array.shape[0])
    original_col_start = max(col_start, 0)
    original_col_end = min(col_end, array.shape[1])

    # Calculate the offsets
    subarray_row_start = original_row_start - row_start
    subarray_col_start = original_col_start - col_start

    # Fill the subarray with the values from the original array
    subarray[subarray_row_start:subarray_row_start + (original_row_end - original_row_start),
             subarray_col_start:subarray_col_start + (original_col_end - original_col_start)] =
(
                 array[original_row_start:original_row_end,
                       original_col_start:original_col_end]
```

```
                    )

        return subarray

#  示例用法
Z = np.random.randint(0, 10, (5, 5))
shape = (4, 4)
fill = 0
position = (1, 1)

print("Original Array:\n", Z)

result = extract_subarray(Z, shape, fill, position)

print("Extracted Subarray:\n", result)
```

```
Original Array:
 [[2 3 6 2 2]
 [6 0 0 5 8]
 [9 8 9 0 2]
 [6 2 1 4 6]
 [0 7 0 6 7]]
Extracted Subarray:
 [[0 0 0 0]
 [0 2 3 6]
 [0 6 0 0]
 [0 9 8 9]]
```

26.

```
#  矩阵加法
def add(A, B):
    return [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in range(len(A))]


#  矩阵减法
def subtract(A, B):
    return [[A[i][j] - B[i][j] for j in range(len(A[0]))] for i in range(len(A))]


#  矩阵数乘
def scalar_multiply(A, scalar):
    return [[A[i][j] * scalar for j in range(len(A[0]))] for i in range(len(A))]
```

```
# 矩阵乘法
def multiply(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    if cols_A != rows_B:
        raise ValueError("Matrix dimensions do not match for multiplication")
    return [[sum(A[i][k] * B[k][j] for k in range(cols_A)) for j in range(cols_B)] for i in range(rows_A)]


# 生成单位矩阵
def identity(n):
    return [[1 if i == j else 0 for j in range(n)] for i in range(n)]


# 矩阵转置
def transpose(A):
    return [[A[j][i] for j in range(len(A))] for i in range(len(A[0]))]


# 计算矩阵的逆
def inverse(A):
    n = len(A)
    # 构造增广矩阵 [A | I]
    augmented = [A[i] + identity(n)[i] for i in range(n)]

    for i in range(n):
        # 如果主对角线元素为 0，则寻找一个非零行并交换
        if augmented[i][i] == 0:
            for k in range(i + 1, n):
                if augmented[k][i] != 0:
                    augmented[i], augmented[k] = augmented[k], augmented[i]
                    break
            else:
                raise ValueError("Matrix is singular and cannot be inverted")

        # 归一化主对角线元素
        pivot = augmented[i][i]
        for j in range(2 * n):
            augmented[i][j] /= pivot

        # 消去其他行的 i 列
```

```
        for k in range(n):
            if k != i:
                factor = augmented[k][i]
                for j in range(2 * n):
                    augmented[k][j] -= factor * augmented[i][j]

    # 提取逆矩阵
    return [row[n:] for row in augmented]

# 重新测试
matrix_d = [[3, 0, 2], [2, 0, -2], [0, 1, 1]]
print(inverse(matrix_d))



# 测试
matrix_a = [[12, 10], [3, 9]]
matrix_b = [[3, 4], [7, 4]]
matrix_c = [[11, 12, 13, 14], [21, 22, 23, 24], [31, 32, 33, 34], [41, 42, 43, 44]]
matrix_d = [[3, 0, 2], [2, 0, -2], [0, 1, 1]]

print(add(matrix_a, matrix_b))    # [[15, 14], [10, 13]]
print(subtract(matrix_a, matrix_b))    # [[9, 6], [-4, 5]]
print(scalar_multiply(matrix_b, 3))    # [[9, 12], [21, 12]]
print(multiply(matrix_a, matrix_b))    # [[106, 88], [72, 48]]
print(identity(3))    # [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
print(transpose(matrix_c))
# [[11, 21, 31, 41], [12, 22, 32, 42], [13, 23, 33, 43], [14, 24, 34, 44]]
print(inverse(matrix_d))
# [[0.2, 0.2, 0.0], [-0.2, 0.3, 1.0], [0.2, -0.3, 0.0]]
```

```
[[0.19999999999999998, 0.2, 0.0], [-0.2, 0.30000000000000004, 1.0], [0.2, -0.30000000000000004, -0.0]]
[[15, 14], [10, 13]]
[[9, 6], [-4, 5]]
[[9, 12], [21, 12]]
[[106, 88], [72, 48]]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
[[11, 21, 31, 41], [12, 22, 32, 42], [13, 23, 33, 43], [14, 24, 34, 44]]
[[0.19999999999999998, 0.2, 0.0], [-0.2, 0.30000000000000004, 1.0], [0.2, -0.30000000000000004, -0.0]]
```

27.

```
def GCD(a, b):
    #使用辗转相除法计算最大公约数
    a, b = abs(a), abs(b)    # 处理负数情况
    while b:
        a, b = b, a % b
    return a
```

```
# 测试样例
print(GCD(3, 5))      # 1
print(GCD(6, 3))      # 3
print(GCD(-2, 6))    # 2
print(GCD(0, 3))      # 3
```

```
1
3
2
3
```

28.

```
def find_consecutive_sums(N):
    results = []
    k = 1    # 连续序列的长度
    while k * (k - 1) // 2 < N:
        remainder = N - (k * (k - 1)) // 2
        if remainder % k == 0:    # 只有当 (N - k(k-1)/2) 可以被 k 整除时，才是合法解
            x = remainder // k
            if x > 0:
                results.append(list(range(x, x + k)))
        k += 1
    return results

# 计算 N = 1000 的所有解
sequences = find_consecutive_sums(1000)

# 打印结果
for seq in sequences:
    print(seq)
```

```
[1000]
[198, 199, 200, 201, 202]
[55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70]
[28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52]
```