```
import torch
import numpy as np
```

Task 1.1

```
# Tạo tensor từ list
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
print(f"Tensor từ list:\n {x_data}\n")

# Tạo tensor từ NumPy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(f"Tensor từ NumPy array:\n {x_np}\n")

# Tạo tensor với các giá trị ngẫu nhiên hoặc hằng số
x_ones = torch.ones_like(x_data) # tạo tensor gồm các số 1 có cùng shape với x_data
print(f"Ones Tensor:\n {x_ones}\n")
x_rand = torch.rand_like(x_data, dtype=torch.float) # tạo tensor ngẫu nhiên
print(f"Random Tensor:\n {x_rand}\n")

# In ra shape, dtype, và device của tensor
print(f"Shape của tensor: {x_rand.shape}")
print(f"Datatype của tensor: {x_rand.dtype}")
print(f"Device lưu trữ tensor: {x_rand.device}")
```

```
Tensor từ list:
 tensor([[1, 2],
        [3, 4]])

Tensor từ NumPy array:
 tensor([[1, 2],
        [3, 4]])

Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.7221, 0.2829],
        [0.9172, 0.2422]])

Shape của tensor: torch.Size([2, 2])
Datatype của tensor: torch.float32
Device lưu trữ tensor: cpu
```

Task 1.2

```
# Cộng x_data với chính nó
add_result = x_data + x_data
print(f"Cộng x_data với chính nó:\n {add_result}\n")

# Nhân x_data với 5
mul_result = x_data * 5
print(f"Nhân x_data với 5:\n {mul_result}\n")

# 3. Nhân ma trận x_data với x_data.T
matmul_result = x_data @ x_data.T
print(f"Nhân ma trận x_data với x_data.T:\n {matmul_result}")
```

```
Cộng x_data với chính nó:
 tensor([[2, 4],
        [6, 8]])

Nhân x_data với 5:
 tensor([[ 5, 10],
        [15, 20]])

Nhân ma trận x_data với x_data.T:
 tensor([[ 5, 11],
        [11, 25]])
```

Task 1.3

```
print(x_data[0])
print(x_data[:, 1])
print(x_data[1, 1])
```

```
tensor([1, 2])
tensor([2, 4])
tensor(4)
```

Task 1.4

```
x_rand4x4 = torch.rand(4, 4)
x_reshaped = x_rand4x4.view(16, 1)
print(x_rand4x4)
print(x_reshaped)
```

```
tensor([[0.7962, 0.4388, 0.3002, 0.6546],
        [0.2433, 0.6690, 0.9760, 0.2870],
        [0.1419, 0.9924, 0.0437, 0.2568],
        [0.7920, 0.5128, 0.9045, 0.4681]])
tensor([[0.7962],
        [0.4388],
        [0.3002],
        [0.6546],
        [0.2433],
        [0.6690],
        [0.9760],
        [0.2870],
        [0.1419],
        [0.9924],
        [0.0437],
        [0.2568],
        [0.7920],
        [0.5128],
        [0.9045],
        [0.4681]])
```

Task 2.1

```
# Tạo một tensor và yêu cầu tính đạo hàm cho nó
x = torch.ones(1, requires_grad=True)
print(f"x: {x}")
# Thực hiện một phép toán
y = x + 2
print(f"y: {y}")
# y được tạo ra từ một phép toán có x, nên nó cũng có grad_fn
print(f"grad_fn của y: {y.grad_fn}")
# Thực hiện thêm các phép toán
z = y * y * 3
# Tính đạo hàm của z theo x
z.backward(retain_graph=True) # tương đương z.backward(torch.tensor(1.))
# Đạo hàm được lưu trong thuộc tính .grad
# Ta có z = 3 * (x+2)^2 => dz/dx = 6 * (x+2). Với x=1, dz/dx = 18
print(f"Đạo hàm của z theo x: {x.grad}")

z.backward()
print(f"Đạo hàm của z theo x lần 2: {x.grad}")
```

```
x: tensor([1.], requires_grad=True)
y: tensor([3.], grad_fn=<AddBackward0>)
grad_fn của y: <AddBackward0 object at 0x7816c4d2d990>
Đạo hàm của z theo x: tensor([18.])
Đạo hàm của z theo x lần 2: tensor([36.])
```

Nếu gọi `z.backward()` thêm lần nữa mà không có `retain_graph=True` ở `z.backward()` trước đó thì PyTorch sẽ giải phóng tất cả tensor trung gian trong graph tính toán ngay sau lần `.backward()` đầu tiên để tiết kiệm bộ nhớ.

Task 3.1

```
# Khởi tạo một lớp Linear biến đổi từ 5 chiều -> 2 chiều
linear_layer = torch.nn.Linear(in_features=5, out_features=2)
# Tạo một tensor đầu vào mẫu
input_tensor = torch.randn(3, 5) # 3 mẫu, mỗi mẫu 5 chiều
# Truyền đầu vào qua lớp linear
output = linear_layer(input_tensor)
print(f"Input shape: {input_tensor.shape}")
print(f"Output shape: {output.shape}")
print(f"Output:\n {output}")
```

```
Input shape: torch.Size([3, 5])
Output shape: torch.Size([3, 2])
Output:
 tensor([[ 0.0970, -0.6830],
        [-0.1226,  0.8289],
        [-0.1983,  0.5737]], grad_fn=<AddmmBackward0>)
```

Task 3.2

```
# Khởi tạo lớp Embedding cho một từ điển 10 từ, mỗi từ biểu diễn bằng vector 3 chiều
embedding_layer = torch.nn.Embedding(num_embeddings=10, embedding_dim=3)
# Tạo một tensor đầu vào chứa các chỉ số của từ (ví dụ: một câu)
# Các chỉ số phải nhỏ hơn 10
input_indices = torch.LongTensor([1, 5, 0, 8])
# Lấy ra các vector embedding tương ứng
embeddings = embedding_layer(input_indices)
print(f"Input shape: {input_indices.shape}")
print(f"Output shape: {embeddings.shape}")
print(f"Embeddings:\n {embeddings}")
```

```
Input shape: torch.Size([4])
Output shape: torch.Size([4, 3])
Embeddings:
 tensor([[-0.6250,  0.7744, -0.3051],
        [ 0.2832, -1.3191,  0.7597],
        [ 1.0509, -0.5575, -1.1793],
        [ 0.2490, -1.3176, -0.8299]], grad_fn=<EmbeddingBackward0>)
```

Task 3.3

```
from torch import nn

class MyFirstModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):
        super(MyFirstModel, self).__init__()
        # Định nghĩa các lớp (layer) bạn sẽ dùng
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, hidden_dim)
        self.activation = nn.ReLU() # Hàm kích hoạt
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, indices):
        # Định nghĩa luồng dữ liệu đi qua các lớp
        # 1. Lấy embedding
        embeds = self.embedding(indices)
        # 2. Truyền qua lớp linear và hàm kích hoạt
        hidden = self.activation(self.linear(embeds))
        # 3. Truyền qua lớp output
        output = self.output_layer(hidden)
        return output

# Khởi tạo và kiểm tra mô hình
model = MyFirstModel(vocab_size=100, embedding_dim=16, hidden_dim=8, output_dim=2)
input_data = torch.LongTensor([[1, 2, 5, 9]]) # một câu gồm 4 từ
output_data = model(input_data)
print(f"Model output shape: {output_data.shape}")
```

```
Model output shape: torch.Size([1, 4, 2])
```