Phần 1: Khám phá Tensor

Task 1.1

```
import torch
import numpy as np
# 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
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([[0.2031, 0.6969],
        [0.0932, 0.8614]])

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

Task 1.2: Các phép toán trên Tensor

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

# Nhân tensor với 5
mul_result = x_data * 5
```

```
print(f"Kết quả nhân với 5:\n{mul_result}\n")

# Nhân ma trận x_data với ma trận chuyển vị của nó (x_data.T)
matmul_result = x_data @ x_data.T
print(f"Kết quả nhân ma trận:\n{matmul_result}\n")
```

```
Kết quả cộng:
tensor([[2, 4],
        [6, 8]])

Kết quả nhân với 5:
tensor([[ 5, 10],
        [15, 20]])

Kết quả nhân ma trận:
tensor([[ 5, 11],
        [11, 25]])
```

Task 1.3: Indexing và Slicing

```
# Lấy hàng đầu tiên (index = 0)
first_row = x_data[0]
print(f"Hàng đầu tiên:\n{first_row}\n")

# Lấy cột thứ hai (index = 1)
second_col = x_data[:, 1]
print(f"Cột thứ hai:\n{second_col}\n")

# Lấy phần tử ở hàng thứ hai, cột thứ hai (chỉ số [1, 1])
value_22 = x_data[1, 1]
print(f"Giá trị ở hàng 2, cột 2: {value_22}\n")
```

```
Hàng đầu tiên:
tensor([1, 2])

Cột thứ hai:
tensor([2, 4])

Giá trị ở hàng 2, cột 2: 4
```

Task 1.4: Thay đổi hình dạng Tensor

```
# Tạo tensor ngẫu nhiên 4x4
x = torch.rand(4, 4)
print(f"Tensor ban đầu (4x4):\n{x}\n")

# Dùng view hoặc reshape để chuyển thành (16, 1)
x_reshaped = x.view(16, 1)  # hoặc x.reshape(16, 1)
print(f"Tensor sau khi reshape (16x1):\n{x_reshaped}\n")
```

```
# Kiểm tra shape mới
print(f"Shape mới: {x_reshaped.shape}")
```

```
Tensor ban đầu (4x4):
tensor([[0.6348, 0.4452, 0.6727, 0.2113],
        [0.8879, 0.6614, 0.0608, 0.4072],
        [0.2309, 0.9062, 0.9057, 0.5458],
        [0.4064, 0.2287, 0.2569, 0.0478]])

Tensor sau khi reshape (16x1):
tensor([[0.6348],
        [0.4452],
        [0.6727],
        [0.2113],
        [0.8879],
        [0.6614],
        [0.0608],
        [0.4072],
        [0.2309],
        [0.9062],
        [0.9057],
        [0.5458],
        [0.4064],
        [0.2287],
        [0.2569],
        [0.0478]])

Shape mới: torch.Size([16, 1])
```

Phần 2: Tự động tính Đạo hàm với autograd

Task 2.1: Thực hành với autograd

```
# 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() # 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}")
```

```
x: tensor([1.], requires_grad=True)
y: tensor([3.], grad_fn=<AddBackward0>)
grad_fn của y: <AddBackward0 object at 0x785a6e5a5150>
Đạo hàm của z theo x: tensor([18.])
```

Nếu bạn gọi lại z.backward() lần thứ hai, PyTorch sẽ báo lỗi. Vì khi bạn gọi z.backward() lần đầu tiên:

- PyTorch sử dụng và đồng thời giải phóng biểu đồ tính toán (computational graph) để tiết kiệm bộ nhớ.
- Do đó, nếu bạn muốn lan truyền ngược nhiều lần, thì phải giữ lại biểu đồ bằng cách thêm tham số retain_graph=True

Phần 3: Xây dựng Mô hình đầu tiên với torch.nn

Task 3.1: Lớp nn.Linear

```
# 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([[ 0.7390, -0.2183],
        [-0.3359,  0.1085],
        [-0.0024,  1.0385]], grad_fn=<AddmmBackward0>)
```

Task 3.2: Lớp nn.Embedding

```
# Khởi tạo lớp Embedding cho một từ điển 10 từ, mỗi từ biểu diễn bằng vector
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([[ 0.6538,  0.5125, -1.1340],
        [ 1.8118, -2.0129, -0.1645],
        [ 0.2063, -0.7799,  0.3810],
        [-1.0897, -0.5039,  0.8265]], grad_fn=<EmbeddingBackward0>)
```

Task 3.3: Kết hợp thành một nn.Module

```
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_di
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])
```