Phần 1: Khám phá Tensor

Task 1.1: Tạo Tensor

```
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 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([[0.7972, 0.3290],
        [0.3406, 0.5863]])

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

```
# 1. Cộng x_data với chính nó
add_result = x_data + x_data
print(f"1. x_data + x_data:\n{add_result}\n")

# 2. Nhân x_data với 5
mul_result = x_data * 5
print(f"2. x_data * 5:\n{mul_result}\n")
```

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

```
1. x_data + x_data:
tensor([[2, 4],
        [6, 8]])

2. x_data * 5:
tensor([[ 5, 10],
        [15, 20]])

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

Task 1.3: Indexing và Slicing

```
# 1. Lấy ra hàng đầu tiên (index 0)
row_first = x_data[0]
print(f"1. Hàng đầu tiên: {row_first}\n")

# 2. Lấy ra cột thứ hai (index 1)
col_second = x_data[:, 1]
print(f"2. Cột thứ hai: {col_second}\n")

# 3. Lấy ra giá trị ở hàng thứ hai (index 1), cột thứ hai (index 1)
value_1_1 = x_data[1, 1]
print(f"3. Giá trị (hàng 2, cột 2): {value_1_1}\n")
```

```
1. Hàng đầu tiên: tensor([1, 2])

2. Cột thứ hai: tensor([2, 4])

3. Giá trị (hàng 2, cột 2): 4
```

Task 1.4: Thay đổi hình dạng Tensor

```
# 1. Sử dụng torch.rand để tạo một tensor có shape (4, 4)
tensor_4x4 = torch.rand(4, 4)
print(f"Tensor 4x4 ban đầu:\n{tensor_4x4}\n")

# 2. Dùng reshape để đổi thành (16, 1)
tensor_16x1 = tensor_4x4.reshape(16, 1)
print(f"Tensor 16x1 sau khi reshape:\n{tensor_16x1}\n")
print(f"Shape cũ: {tensor_4x4.shape}")
print(f"Shape mới: {tensor_16x1.shape}")
```

```
Tensor 4x4 ban đầu:
tensor([[0.6515, 0.4333, 0.1578, 0.7543],
        [0.7542, 0.1521, 0.4136, 0.1390],
        [0.0415, 0.4913, 0.6765, 0.7910],
        [0.3594, 0.6944, 0.1474, 0.7489]])

Tensor 16x1 sau khi reshape:
tensor([[0.6515],
        [0.4333],
        [0.1578],
        [0.7543],
```

```
        [0.7542],
        [0.1521],
        [0.4136],
        [0.1390],
        [0.0415],
        [0.4913],
        [0.6765],
        [0.7910],
        [0.3594],
        [0.6944],
        [0.1474],
        [0.7489]])

Shape cũ: torch.Size([4, 4])
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 0x75701731e9b0>
Đạo hàm của z theo x: tensor([18.])
```

Câu hỏi: Chuyện gì xảy ra nếu bạn gọi z.backward() một lần nữa? Tại sao?

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

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[12], line 1
----> 1 z.backward()
      2 print(f"Đạo hàm cấp 2 của z theo x: {x.grad}")

File ~/NLP/.venv/lib/python3.12/site-packages/torch/_tensor.py:625, in Tensor.backward(self, gradient, retain_graph, create_graph, inputs)
    615 if has_torch_function_unary(self):
    616     return handle_torch_function(
    617         Tensor.backward,
    618         (self,),
   (...)    623         inputs=inputs,
    624     )
--> 625 torch.autograd.backward(
    626     self, gradient, retain_graph, create_graph, inputs=inputs
    627 )

File ~/NLP/.venv/lib/python3.12/site-packages/torch/autograd/__init__.py:354, in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables, inputs)
    349     retain_graph = create_graph
    351 # The reason we repeat the same comment below is that
    352 # some Python versions print out the first line of a multi-line function
    353 # calls in the traceback and some print out the last line
--> 354 _engine_run_backward(
    355     tensors,
    356     grad_tensors_,
    357     retain_graph,
    358     create_graph,
    359     inputs_tuple,
    360     allow_unreachable=True,
    361     accumulate_grad=True,
    362 )

File ~/NLP/.venv/lib/python3.12/site-packages/torch/autograd/graph.py:841, in _engine_run_backward(t_outputs, *args, **kwargs)
    839     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    840 try:
--> 841     return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
    842         t_outputs, *args, **kwargs
    843     )  # Calls into the C++ engine to run the backward pass
    844 finally:
    845     if attach_logging_hooks:

RuntimeError: Trying to backward through the graph a second time (or directly access saved tensors after they have already been freed). Saved intermediate values of the graph are freed when you call .backward() or autograd.grad(). Specify retain_graph=True if you need to backward through the graph a second time or if you need to access saved tensors after calling backward.
```

Nếu gọi z.backward() một lần nữa thì chương trình báo lỗi vì lý do chính là để tiết kiệm bộ nhớ:

- Đồ thị Tính toán: Khi thực hiện các phép toán trên tensor có `requires_grad=True`, PyTorch sẽ âm thầm xây dựng một "đồ thị tính toán"
- Quá trình `backward()`: Khi gọi `z.backward()`, PyTorch sử dụng đồ thị này để đi ngược lại (từ z về x), áp dụng quy tắc chuỗi để tính đạo hàm $\frac{dz}{dx}$ và lưu kết quả vào `x.grad`
- Xóa Đồ thị: Theo mặc định, ngay sau khi `z.backward()` tính toán xong và lưu gradient, PyTorch sẽ xóa đồ thị tính toán đó đi. Việc này giải phóng bộ nhớ đã dùng để lưu các giá trị trung gian cần thiết cho việc tính đạo hàm
- Lỗi ở lần gọi thứ hai: Khi gọi `z.backward()` lần thứ hai, PyTorch cố gắng tìm lại đồ thị để tính toán đạo hàm, nhưng đồ thị không còn tồn tại nữa. Do đó, chương trình bị lỗi

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

Task 3.1: Lớp nn.Linear

```
import torch

# 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([[-0.9573,  0.5969],
        [-0.1251, -0.5139],
        [-0.6276,  0.2310]], 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 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([[ 0.4616,  0.1849, -0.4434],
        [-1.7394, -0.4961, -1.0500],
        [ 1.1849,  0.1603, -2.3531],
        [-0.0363,  0.8099, -0.6961]], 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_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}")
print(f"Model output: {output_data}")
```

```
Model output shape: torch.Size([1, 4, 2])
Model output: tensor([[[-0.4086, -0.1009],
         [-0.2795, -0.2178],
         [-0.6725, -0.0585],
         [-0.4884, -0.3320]]], grad_fn=<ViewBackward0>)
```