Course For NLP from LinkedIn

Heavenly Father is guiding me - this one has a text classification project.

https://www.linkedin.com/learning/natural-language-processing-with-pytorch/popular-topics-in-nlp

The teacher is Zhonyu Pan, Content Creator at LinkedIn

We'll be using PyTorch, and we'll use a Convolutional Neural Network (feature rather than position) to do our text classification.

```
Input -> Convolution -> Pooling -> ... -> Fully-connected layer -> Output
```

We are also learning about RNNs. RNN doesn't only pass data forward, but only feeds the data back into itself. CNN only goes forward. RNN can remember context before and after words in a sequence. It's usually slower that a CNN.

It's time to learn PyTorch

**PyTorch tensor**

A tensor is a data structure or data container we use in PyTorch for carrying arrays of numbers.

**a. Creating a tensor**

Generate | use pip to install PyTorch | Close

1 of 3 | Undo Changes | Use code with caution

```
# prompt: use pip to install PyTorch

#!pip install torch torchvision
!pip install torch
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.3.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.15.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch) (4.12.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch) (1.12.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2023.6.0)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (23.7 MB)
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (823 kB)
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch)
  Using cached nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (14.1 MB)
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch)
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7 MB)
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch)
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6 MB)
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch)
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6 MB)
Collecting nvidia-curand-cu12==10.3.2.106 (from torch)
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5 MB)
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch)
  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl (124.2 MB)
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch)
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl (196.0 MB)
Collecting nvidia-nccl-cu12==2.20.5 (from torch)
  Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl (176.2 MB)
Collecting nvidia-nvtx-cu12==12.1.105 (from torch)
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
Requirement already satisfied: triton==2.3.0 in /usr/local/lib/python3.10/dist-packages (from torch) (2.3.0)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-cu12==11.4.5.107->torch)
  Downloading nvidia_nvjitlink_cu12-12.5.40-py3-none-manylinux2014_x86_64.whl (21.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.3/21.3 MB 43.0 MB/s eta 0:00:00
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (2.1.5)
Requirement already satisfied: mpmath<1.4.0,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy->torch) (1.3.0)
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-runtime-c
```

Generate | import pytorch as well as numpy | Close

```
# prompt: import pytorch as well as numpy

import torch
import numpy as np
```

Generate | Construct a tensor named tensor0 from the array, array = [[1, 2], [7, 4],[5, 6]]. Print the tensor. Output "The data structure type of tensor0: ", then appropriately continue the statement using code. After, output "The data type of tensor0: " and continue the statement appropriately | Close

1 of 4 | Undo Changes | Use code with caution

```
#  prompt: Construct a tensor named tensor0 from the array,
#+ array = [[1, 2], [7, 4],[5, 6]].
#+ Print the tensor.
#+ Output "The data structure type of tensor0: ",
#+ then appropriately continue the statement using code.
#+ After, output "The data type of tensor0: " and continue
#+ the statement appropriately using code.

#import torch

# Create a tensor named tensor0 from the array
array = [[1, 2], [7, 4], [5, 6]]
tensor0 = torch.tensor(array)

## Added by DWB
print(tensor0)

# Print the tensor
print("The data structure type of tensor0: ", type(tensor0))

# Print the data type of tensor0
print("The data type of tensor0: ", tensor0.dtype)
```

```
tensor([[1, 2],
        [7, 4],
        [5, 6]])
The data structure type of tensor0:  <class 'torch.Tensor'>
The data type of tensor0:  torch.int64
```

Generate | Now, create a tensor named tensor1 from the same data, but as a numpy array. Continue the other steps as before. | Close

1 of 3 | Undo Changes | Use code with caution

```
#  prompt: Now, create a tensor named tensor0 from the same data,
#+ but as a numpy array. Continue the other steps as before.

#import torch
#import numpy as np

# Create a tensor named tensor0 from the array as a numpy array
array = np.array([[1, 2], [7, 4], [5, 6]])
tensor1 = torch.tensor(array)

## Added by DWB
print(tensor1)

# Print the tensor
print("The data structure type of tensor1: ", type(tensor1))

# Print the data type of tensor1
print("The data type of tensor1: ", tensor1.dtype)
```

```
tensor([[1, 2],
        [7, 4],
        [5, 6]])
The data structure type of tensor1:  <class 'torch.Tensor'>
The data type of tensor1:  torch.int64
```

**b. Common Methods: Slicing and Concatenation**

*Slicing*

Generate | Create tensorA = tensor([[1.,1.,1.],[2.,2.,2.]])
so that the two rows of tensorA are different.
and... create tensorB = tensor([[3.,3.,3.],[4.,4.,4.]]) | Close

1 of 4 | Undo Changes | Use code with caution

```
# prompt: Create tensorA = tensor([[1.,1.,1.],[2.,2.,2.]])
# so that the two rows of tensorA are different.
# and... create tensorB = tensor([[3.,3.,3.],[4.,4.,4.]])
# so that the two rows of tensorB are different
# using only PyTorch methods.
# Print the contents of the two tensors.

#import torch

# Create tensorA with two different rows
tensorA = torch.cat((torch.ones(1, 3), 2 * torch.ones(1, 3)), dim=0)

# Create tensorB with two different rows
tensorB = torch.cat((3 * torch.ones(1, 3), 4 * torch.ones(1, 3)), dim=0)

# Print the contents of the two tensors
print("tensorA:", tensorA)
print("tensorB:", tensorB)
```

```
tensorA: tensor([[1., 1., 1.],
        [2., 2., 2.]])
tensorB: tensor([[3., 3., 3.],
        [4., 4., 4.]])
```

Generate | Impressive, but you can pass in '[[1.,1.,1.],[2.,2.,2.]]' for tensorA
and '[[3.,3.,3.],[4.,4.,4.]]' for tensorB. | Close

1 of 4 | Undo Changes | Use code with caution

```
# prompt: Impressive, but you can pass in '[[1.,1.,1.],[2.,2.,2.]]' for tensorA
# and '[[3.,3.,3.],[4.,4.,4.]]' for tensorB.

## Create tensorA with two different rows
#tensorA = torch.cat((torch.ones(1, 3), 2 * torch.ones(1, 3)), dim=0)
#
## Create tensorB with two different rows
#tensorB = torch.cat((3 * torch.ones(1, 3), 4 * torch.ones(1, 3)), dim=0)
#
## Print the contents of the two tensors
#print("tensorA:", tensorA)
#print("tensorB:", tensorB)

# Pass in '[[1.,1.,1.],[2.,2.,2.]]' for tensorA
# and '[[3.,3.,3.],[4.,4.,4.]]' for tensorB
tensorA = torch.tensor([[1.,1.,1.],[2.,2.,2.]])
tensorB = torch.tensor([[3.,3.,3.],[4.,4.,4.]])

# Print the contents of the two tensors
print("tensorA:", tensorA)
print("tensorB:", tensorB)
```

Generate | "Slicing the first two rows of tensorA" + \
"(index one inclusive, index two exclusive)",
then write the code to do so and print the result | Close

1 of 4 | Use code with caution

```
# prompt: Output
# "Slicing the first two rows of tensorA" + \
# "(index one inclusive, index two exclusive)",
# then write the code to do so and print the result
# Follow the same procedure with the text,
# "Slicing the first two columns of tensorA" + \
# "(take all rows, then slice columns)".

# Slicing the first two rows of tensorA (index one inclusive, index two exclusive)
print("Slicing the first two rows of tensorA (index one inclusive, index two exclusive):")
print(tensorA[1:2, :])

# Slicing the first two columns of tensorA (take all rows, then slice columns)
print("Slicing the first two columns of tensorA (take all rows, then slice columns):")
print(tensorA[:, :2])
```

Her code

```
# Slicing is all the same as numpy arrays
print('Slicing the first two rows of tensorA (index one inclusive index two exclusive): ')
print(tensorA[:2])
print("Slicing the first two columns of tensorA (take all rows, then slice columns):")
print(tensorA[:, :2])
```

```
Slicing the first two rows of tensorA (index one inclusive index two exclusive):
tensor([[1., 1., 1.],
        [2., 2., 2.]])
Slicing the first two columns of tensorA (take all rows, then slice columns):
tensor([[1., 1.],
        [2., 2.]])
```

*Concatenation*

Generate | Output
"Vertically concatenate tensorA and tensorB: (default: dim=0)"
then do so, using the variable concat_v. Then, print the variable. | Close

1 of 4 | Undo Changes | Use code with caution

```
# prompt: Output
# "Vertically concatenate tensorA and tensorB: (default: dim=0)"
# then do so, using the variable concat_v. Then, print the variable.

## It did too much again : )
#import torch
#
## Create tensorA with two different rows
#tensorA = torch.tensor([[1.,1.,1.],[2.,2.,2.]])
#tensorB = torch.tensor([[3.,3.,3.],[4.,4.,4.]])
#
## Print the contents of the two tensors
#print("tensorA:", tensorA)
#print("tensorB:", tensorB)

# Vertically concatenate tensorA and tensorB: (default: dim=0)
concat_v = torch.cat((tensorA, tensorB), dim=0)
print("Vertically concatenate tensorA and tensorB: (default: dim=0)")
print(concat_v)
```

```
Vertically concatenate tensorA and tensorB: (default: dim=0)
tensor([[1., 1., 1.],
        [2., 2., 2.],
        [3., 3., 3.],
        [4., 4., 4.]])
```

Generate | Follow the same procedure as before, but with the text,
"Horizontally concatenate tensorA and tensorB (dim=1)"
and the variable, concat_h | Close

1 of 1 | Use code with caution

```
# prompt: Follow the same procedure as before, but with the text,
# "Horizontally concatenate tensorA and tensorB (dim=1)"
```

```
# and the variable, concat_h

# Horizontally concatenate tensorA and tensorB (dim=1)
concat_h = torch.cat((tensorA, tensorB), dim=1)
print("Horizontally concatenate tensorA and tensorB (dim=1)")
print(concat_h)
```

```
Horizontally concatenate tensorA and tensorB (dim=1)
tensor([[1., 1., 1., 3., 3., 3.],
        [2., 2., 2., 4., 4., 4.]])
```

Her code was a bit different.

```
concat_v = torch.cat([tensorA, tensorB])
concat_h = torch.cat([tensorA, tensorB], dim=1)

print(concat_v)
print(concat_h)
```

```
tensor([[1., 1., 1.],
        [2., 2., 2.],
        [3., 3., 3.],
        [4., 4., 4.]])
tensor([[1., 1., 1., 3., 3., 3.],
        [2., 2., 2., 4., 4., 4.]])
```

The CoPilot generation was fun. I got to learn prompt engineering, especially with the original 2x3 tensors. It did it with only PyTorch methods, but I had to be pretty explicit. Then, I told it that it could pass in arrays.

```
!date +'%s_%Y-%m-%dT%H:%M:%S%z'
```

```
1718947145_2024-06-21T05:19:05+0000
```