Application of Deep Learning to Text and Image Data

Module 1, Lab 1, Notebook 1: Getting Started with PyTorch

This notebook will introduce you to the PyTorch deep learning framework, which is the tool that you will use throughout this course to implement neural network models. For more information, see the PyTorch documentation.

This notebook has been divided into two parts. In the first part of the notebook, you will learn how to use PyTorch to manipulate data to be ready for use in model training. In the second part, you will practice a crucial step in nearly all deep learning optimization algorithms: *differentiation*.

To learn these topics, you will examine how PyTorch stores and manipulates data in *n*-dimensional arrays, which are also called *tensors*. To define the tensors and arrays, you will use *NumPy*, which is the most widely used scientific computing package in Python.

Other frameworks are available, but this lab focuses on PyTorch, which has two key features. First, GPU is well-supported to accelerate the computation, whereas NumPy supports only CPU computation. Second, the tensor class supports automatic differentiation. These properties make the tensor class suitable for deep learning.

You will learn the following:

- How to explore tensors
- Why you index and slice tensors
- How to index and slice tensors
- Common tensor operations
- How to perform tensor operations on data
- How to convert tensors to other Python objects

You will be presented with activities throughout the notebook:

No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.

Index

Data manipulation

In this section, you will practice basic data manipulation.

In [1]:
```
# Install libraries
!pip install -U -q -r requirements.txt
```

```
ERROR: pip's dependency resolver does not currently take into account all the pack
ages that are installed. This behaviour is the source of the following dependency
conflicts.
autovizwidget 0.21.0 requires pandas<2.0.0,>=0.20.1, but you have pandas 2.0.3 whi
ch is incompatible.
hdijupyterutils 0.21.0 requires pandas<2.0.0,>=0.17.1, but you have pandas 2.0.3 w
hich is incompatible.
sparkmagic 0.21.0 requires pandas<2.0.0,>=0.17.1, but you have pandas 2.0.3 which
is incompatible.
```

In [2]:
```
# Import basic libraries to work with data and tensors
import numpy as np
import pandas as pd
import torch

# Import utility functions for activities
from MLUDTI_EN_M1_Lab1_quiz_questions import *
```

Exploring tensors

A tensor represents an array of numerical values. A one-axis tensor corresponds to a one-dimensional vector in math, and a two-axis tensor corresponds to a matrix. Tensors with more than two axes don't have special mathematical names.

You can use the `arange` operation to create a row vector x that contains the first 12 integers, starting with 0. Unless otherwise specified, a new tensor will be stored in main memory and designated for CPU-based computation.

Note: Integers are created as floats by default.

In [3]:
```
# Create a tensor with values in the range 0-11
x = torch.arange(12)

# Print the tensor
x
```

Out[3]:
```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

You can view a tensor's shape (the length along each axis) by reviewing its `shape` property.

In [4]:
```
# Check for tensor size
x.shape
```

Out[4]:
```
torch.Size([12])
```

You can use the `reshape` function to transform the tensor x from a row vector with shape (12) to a matrix with shape (3, 4).

In [5]:
```
# Reshape the tensor into a matrix with three rows and four columns
x_reshaped = x.reshape(3, 4)

# Print the reshaped tensor
x_reshaped
x.reshape(3, 4)
```

Out[5]:
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

You can also reshape a tensor into a matrix by choosing one dimension and letting the other dimension be calculated implicitly (automatically). This is done by using -1 for the dimension that you want to be determined automatically.

For example, instead of calling `x.reshape(3,4)`, you can call either `x.reshape(-1,4)` or `x.reshape(3,-1)` to get the same result.

It's important to initialize a tensor in memory with the desired shape. You can do this by initializing with zeros, ones, other constants, or numbers that are randomly sampled from a specific distribution.

You can use the `zeros()` function to create a tensor with all elements set to 0 and a shape of (2, 3, 4).

In [6]:
```
# Create two matrices with three rows and four columns respectively and fill with 0
zeros_tensor = torch.zeros((2, 3, 4))

# Print the matrices
zeros_tensor
```

Out[6]:
```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Similarly, you can use the `ones()` function to create tensors where each element is set to 1.

In [7]:
```
# Create two matrices with three rows and four columns respectively and fill with 1
ones_tensor = torch.ones((2, 3, 4))

# Print the matrices
ones_tensor
```

Out[7]:
```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

In other situations, you will want to create a tensor with randomly sampled values from a probability distribution.

For example, when you construct arrays to serve as parameters in a neural network, you typically initialize their values randomly. To do this, you can use the `randn()` function to create a tensor where each of its elements is randomly sampled from a standard Gaussian (normal) distribution with a mean of 0 and a standard deviation of 1.

In [8]:
```
# Create a 3x4 matrix where the fill values are randomly sampled from a normal dist
x = torch.randn(3, 4)

# Print the matrix
x
```

Out[8]:
```
tensor([[-0.5771, -0.3523, -0.9491,  0.0322],
        [-0.7628,  1.5510,  1.2814, -0.1270],
        [ 1.4167, -0.9029, -1.6442, -0.1052]])
```

As you work with tensors, you will need to be able to verify that they have the expected shape, size, and type of stored values:

- Use the `shape` attribute to determine the shape of the tensor.
- Use the `numel()` function to determine the number of elements in the tensor. This is equal to the product of the components of the shape.
- Use the `.dtype` attribute to determine the data type of the stored values.

In [9]:
```
# Check shape (rows and columns), total number of elements, and what data type is s
x.shape, x.numel(), x.dtype
```

Out[9]: (torch.Size([3, 4]), 12, torch.float32)

Try it yourself!

To test your understanding of basic tensor functionality, run the following cell.

In [11]:
```
# Run this cell to display the question and check your answer
question_1

# For help, read the section "Exploring Tensors".
```

Out[11]:

Which option would you use to create a 2x4 matrix that is filled with 0s?

torch.ones(2,4)

torch.zeros(4,2)

torch.zeros(2,4)

Submit

Indexing and slicing tensors

You can access elements in a tensor by index the same way that you would access elements in a Python array.

In [12]:
```
# Create a 3x4 matrix where the fill values are randomly sampled from a normal dist
x = torch.randn(3, 4)

# Print the full tensor, the last row, and the last two rows
x, x[-1], x[1:3]
```

Out[12]:
```
(tensor([[ 1.2636, -0.4582,  1.9915,  1.8292],
         [-0.2902,  0.1495, -1.0261, -0.3824],
         [-0.4889,  1.2626, -0.1553,  0.6989]]),
 tensor([-0.4889,  1.2626, -0.1553,  0.6989]),
 tensor([[-0.2902,  0.1495, -1.0261, -0.3824],
         [-0.4889,  1.2626, -0.1553,  0.6989]]))
```

You can access values for a specific location by specifying all of the location indices.

In [13]:
```
x[0, 0]
```

Out[13]:
```
tensor(1.2636)
```

You can also write elements of a matrix by specifying indices.

In [14]:
```
# Assign a specific value in a given row and column
x[1, 2] = 9

# Print the tensor with the newly assigned value
x
```

Out[14]:
```
tensor([[ 1.2636, -0.4582,  1.9915,  1.8292],
        [-0.2902,  0.1495,  9.0000, -0.3824],
        [-0.4889,  1.2626, -0.1553,  0.6989]])
```

You can also use multidimensional slicing to replace numbers inside the tensor.

In [15]:
```
# Assign the same value to a slice of rows and columns
x[0:2, :] = 12

# Print the tensor with the newly assigned values
x
```

Out[15]:
```
tensor([[12.0000, 12.0000, 12.0000, 12.0000],
        [12.0000, 12.0000, 12.0000, 12.0000],
        [-0.4889,  1.2626, -0.1553,  0.6989]])
```

Try it yourself!

 Activity

To test your understanding of indexing and slicing tensors, run the following cell.

In [17]:
```
# Run this cell to display the question and check your answer
question_2

# x[0] would return the first row, not the very first element.
# x[1] would return the second row. Keep in mind, indexing starts with 0 in Python.
# For help, read the section "Indexing and Slicing Tensors".
```

Out[17]:

Which option would you use to access the first element that is stored in tensor x with shape 4x6?

x[1]

x[0,0]

x[0]

Submit

Tensor operations

You can use common arithmetic operators (`+` , `-` , `*` , `/` , and `**`) for element-wise operations on any identically shaped tensors of arbitrary shape.

In the following example, a five-element tuple is created where each element is the result of an element-wise operation.

In [19]:
```
# Create two tensors: x and y filled with some values
x = torch.tensor([1.0, 2.0, 4.0, 8.0])
y = torch.tensor([2.0, 2.0, 2.0, 2.0])

# Sum, difference, element-wise multiplication, division, exponentiation (operator
x + y, x - y, x * y, x / y, x**y
```

Out[19]:
```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

You can apply many more element-wise operations, including unary operators such as exponentiation.

In [20]:
```
# Calculate the exponential for each element in the tensor
torch.exp(x)
```

Out[20]:
```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

In addition to element-wise computations, you can calculate linear algebra operations including vector dot products and matrix multiplication.

The dot product is an important concept in ML because it can be used to quantify similarity.

In [21]:
```
# Calculate the dot product between two tensors
torch.dot(x, y)
```

Out[21]:
```
tensor(30.)
```

You can also calculate the dot product of two vectors manually by performing an element-wise multiplication and then summing the result.

In [22]:
```
# Calculate the dot product and sum for two tensors
torch.sum(x * y)
```

Out[22]:
```
tensor(30.)
```

To perform matrix multiplication of tensors, PyTorch offers the `matmul()` function.

In [23]:
```
# Product of a 3x4 matrix and 4x1 vector
A = torch.arange(12, dtype=torch.float).reshape(3, 4)
print(A)

x = torch.arange(4, dtype=torch.float)
print(x)

# Perform the matrix multiplication
torch.matmul(A, x)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
tensor([0., 1., 2., 3.])
```

Out[23]:
```
tensor([14., 38., 62.])
```

In [24]:
```
# The matmul can be calculated several ways
A.matmul(x)
```

Out[24]:
```
tensor([14., 38., 62.])
```

Note that matrix multiplication is not communicative. If you have matrices that aren't shaped properly for multiplication, you will receive a runtime error.

In [25]:
```
# Uncomment and run the following cell. You will get a runtime error.

# torch.matmul(x, A)
```

PyTorch helps you do complex operations. You can create tensors, reshape them, and then multiply them to get the result with a few lines of code. These operations will make it easier for you to train and validate a model.

In [26]:
```
# Initialize matrix A
A = torch.arange(12).reshape(6, 2)
```

```
# Initialize matrix B
B = torch.arange(10).reshape(2, 5)

# Calculate the matrix multiplication between A and B and the resulting shape
C = torch.matmul(A, B)
print(C)
print(C.shape)
```

```
tensor([[  5,   6,   7,   8,   9],
        [ 15,  20,  25,  30,  35],
        [ 25,  34,  43,  52,  61],
        [ 35,  48,  61,  74,  87],
        [ 45,  62,  79,  96, 113],
        [ 55,  76,  97, 118, 139]])
torch.Size([6, 5])
```

For more information about linear algebra operations, see Linear Algebra on the Dive into Deep Learning site.

For more information about the PyTorch `matmul`, `dot`, and `mm` operations, see the PyTorch documentation: matmul, dot, and mm.

Try it yourself!

To test your understanding of tensor operations, run the following cell.

In [28]:
```
# Run this cell to display the question and check your answer
question_3
```

Out[28]:

True or false? Matrix multiplication is commutative, so torch.matmul(A, B) == torch.matmul(B, A).

True | False

Submit

Conversion to other Python objects

It's important to be able to convert between PyTorch and NumPy tensors. When you convert between different types, they don't share memory. This means that you need more memory resources; however, computations are not halted when different computations need to be performed on the CPU compared to the GPU. Because they don't share memory, no wait time occurs while deciding whether the NumPy package or PyTorch needs to perform an operation because they aren't using the same chunk of memory.

In [29]:
```
# Create a NumPy tensor
A = x.numpy()

# Convert the NumPy tensor to a PyTorch tensor
B = torch.tensor(A)

# Print the resulting types
type(A), type(B)
```

Out[29]:
```
(numpy.ndarray, torch.Tensor)
```

To convert a size-1 tensor to a Python scalar, you can use the `item` function or one of Python's built-in functions.

In [30]:
```
a = torch.tensor([3.5])

a, a.item(), float(a), int(a)
```

Out[30]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

Automatic differentiation

In this section, you will practice automatic differentiation and see how to use PyTorch to take advantage of GPUs.

You can train a deep learning model on a CPU or GPU. The most computationally demanding piece in a neural network is multiple matrix multiplications. In general, when training on a CPU, each operation will be done sequentially. When using a GPU, all the operations will be done in parallel, which makes GPU faster than CPU.

CUDA is a parallel computing platform that focuses on general computing on GPUs. PyTorch natively supports CUDA, and you can access it with the `torch.cuda` library. To find out whether you have a GPU at your disposal and set your device accordingly, you can use `cuda.is_available()`.

In [31]:
```
# Set to GPU if GPU is available; otherwise, use CPU
device = "cuda" if torch.cuda.is_available() else "cpu"

# Print device type for reference
device
```

Out[31]:
```
'cuda'
```

PyTorch can allocate the tensors to the GPU on object creation by specifying the `device` parameter.

In [32]:
```
# Create a tensor and allocate memory with 'requires_grad', store on GPU
a = torch.arange(4, requires_grad=True, dtype=torch.float, device=device)
a
```

Out[32]:
```
tensor([0., 1., 2., 3.], device='cuda:0', requires_grad=True)
```

Differentiation is a crucial step in nearly all deep learning optimization algorithms. In this section, you will examine how PyTorch's automatic differentiation expedites this work by automatically calculating derivatives, which enables the system to backpropagate gradients.

Consider an example where you want to differentiate a function $f(\mathbf{x}) = 0.6x^2$ with respect to parameter x. Start by assigning an initial value of x.

In [33]:
```
# Print tensor that was created in the previous section
x
```

Out[33]:
```
tensor([0., 1., 2., 3.])
```

Before you calculate the gradient of $f(x)$ with respect to x, you need a place to store it.

It's important not to allocate new memory every time you take a derivative with respect to a parameter because the same parameters might be updated thousands or millions of times. This will cause memory to run out.

Note: A gradient of a scalar-valued function with respect to a vector x is itself vector valued and has the same shape as x.

In [34]:
```
# Allocate memory for a tensor's gradient by invoking .requires_grad
# Note that the tensor can be created already with an attached gradient by writing
x.requires_grad_(True)

# After calculating the gradient taken with respect to x, you can access it by usin
# The grad attribute's values are initialized with None
x.grad
```

Now, calculate $f(x)$.

In [35]:
```
# Calculate the dot product and multiply with .6 (as in the toy function example)
y = 0.6 * torch.dot(x, x)
```

```
# Print new tensor
y
```

Out[35]:
```
tensor(8.4000, grad_fn=<MulBackward0>)
```

Next, you can automatically calculate the gradient of $f(x)$ with respect to each component of x by calling the function for backpropagation and printing the gradient.

In [36]:
```
# Calculate the gradient
y.backward()

# Print the gradient values
x.grad
```

Out[36]:
```
tensor([0.0000, 1.2000, 2.4000, 3.6000])
```

Now, determine if this is the expected output. The gradient of the function $f(x) = 0.6x^2$ with respect to x should be $1.2x$.

Verify that the desired gradient was calculated correctly.

In [37]:
```
# Check if the calculated gradient matches the manual calculation
x.grad == 1.2 * x
```

Out[37]:
```
tensor([True, True, True, True])
```

Conclusion

In this notebook, you practiced using PyTorch to perform different mathematical calculations.

Next lab

In the next lab, you will learn the basics of neural networks and train your first one.

Observation.

1