FIT5217 Tutorial Introduction to Pytorch and Word-level RNN language models (part 1 Introduction to pytorch)

Computing gradients

$$\frac{\partial loss}{\partial w} \quad \text{and} \quad \frac{\partial loss}{\partial b}$$

```
loss.backward()
print(w.grad)
print(b.grad)

tensor([[0.0900, 0.0468, 0.0449],
        [0.0900, 0.0468, 0.0449],
        [0.0900, 0.0468, 0.0449],
        [0.0900, 0.0468, 0.0449],
        [0.0900, 0.0468, 0.0449]])
tensor([0.0900, 0.0468, 0.0449])
```

get the gradient to update the parameters

loss.backward() --> update the parameters

Automatic Differentiation with torch.autograd

prediction z —— y = wx + b

w and b --> requires_grad=True

because they are parameters that need to be updated during training

if a guy is handsome — 0, 1 — binary classification — binary cross entropy

designing loss ---> advanced topic

our model --> generated prediction as ground truth

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

1. specify a simple linear model
w and b are parameters

the linear model is:
z = x @ w + b

x is the input data and **z is the prediction**

2. the prediction is compared with y (ground truth) --> to calculate the loss

```
print("Gradient function for z = ",z.grad_fn)
print("Gradient function for loss = ",loss.grad_fn)

<AddBackward0 object at 0x0079abbaabb30>
Gradient function for loss = <BinaryCrossEntropyWithLogitsBackward0 object at 0x0079abbaabb30>
```

z is the final prediction --> the last operation is + b, and thus it is <AddBackward>

get the loss gradient function (based on the loss function)

installation & cuda

I guess you guys have played around **numpy** before.

it's **doable to build a neural network in numpy** but we **choose to use pytorch** for two reasons

1. pytorch got **a lot of useful and powerful API**, save you a lot of time if you build it manually

2. **pytorch use tensors** and **numpy use arrays.**

some of you may confuse about what's the difference between tensors and numpy array

They are **basically the same things.** The key difference is **Arrays cannot be run on GPU while tensors can.**

Tensor and numpy array are interconvertible

as I have said **using gpu is much faster than cpu**, it's faster because **gpu is better for parallel tasks**

```
#Pytorch include the downloadable datasets (e.g., "torchtext")
# Install the following.
pip3 install torch torchtext numpy
```

```
import torch
print("Using torch", torch.__version__)

Using torch 2.0.0+cu118
```

import and check torch version

you need to **check the cuda version (cu118 in this case)**, you need to download the torch package with the corresponding driver

cuda toolkit (software) — support GPU

```
torch.cuda.is_available()

True

torch.cuda.device_count()

1

torch.cuda.current_device()

0
```

- gpu is available --> return True
- only have one gpu
- the current device **gpu id** is 0

Tensors

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

Tensors can be created directly from data or from NumPy arrays.

```
# Directly from data
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)

# From a NumPy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

convert to tensor from list or arrays

Tensor and numpy array are interconvertible

```
print(f"Shape of tensor: {x_data.shape}")
print(f"Datatype of tensor: {x_data.dtype}")
print(f"Device tensor is stored on: {x_data.device}")

Shape of tensor: torch.Size([2, 2])
Datatype of tensor: torch.int64
Device tensor is stored on: cpu
```

1. shape of tensor
2. datatype of the tensor
3. which device is the tensor on
(note: tensor can be on gpu)

```
shape = (2,3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

1. create a random 2x3 matrix (random sampling from 0 - 1)
2. create a all ones 2x3 matrix
3. create a all zeros 2x3 matrix

```
Random Tensor:
 tensor([[0.8900, 0.4354, 0.1276],
         [0.4873, 0.4836, 0.3832]])

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

Zeros Tensor:
 tensor([[0., 0., 0.],
         [0., 0., 0.]])
```

```
x_ones = torch.ones_like(x_data) # retains the proper
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float)
print(f"Random Tensor: \n {x_rand} \n")
```

create all ones tensors or random tensor
based on x_data shape

```
# Directly from data
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

x_data is 2x2 matrix

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

Random Tensor:
 tensor([[0.2404, 0.3624],
         [0.8899, 0.8624]])
```

Operations on Tensors

```
tensor = torch.rand(3,4)
# # We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
tensor

tensor([[0.5928, 0.4746, 0.2738, 0.6793],
        [0.6626, 0.1529, 0.0372, 0.3733],
        [0.7002, 0.0841, 0.7229, 0.5590]], device='cuda:0')
```

0, 1, 2, 3 — 4 — tensor.to('cuda:2')

put tensor to GPU --> cuz we wanna speed up!

```
print("First row:", tensor[0]")
print("First column:", tensor[:, 0]")
print("Last column:", tensor[..., -1]")
tensor[:,1] = 0
print(tensor)

First row: tensor([0.1928, 0.4746, 0.2738, 0.6793], device='cuda:0')
First column: tensor([0.1928, 0.6626, 0.7002], device='cuda:0')
Last column: tensor([0.6793, 0.3733, 0.5590], device='cuda:0')
tensor([[0.1928, 0.0000, 0.2738, 0.6793],
        [0.6626, 0.0000, 0.0372, 0.3733],
        [0.7002, 0.0000, 0.7229, 0.5590]], device='cuda:0')
```

1. get row/column or the last column of the tensor

2. assign the second column's value as 0

vertically — 2 1

horizontally

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)

tensor([[0.5928, 0.0000, 0.2738, 0.6793, 0.5928, 0.0000, 0.2738, 0.6793,
         0.5928, 0.0000, 0.2738, 0.6793],
        [0.6626, 0.0000, 0.0372, 0.3733, 0.6626, 0.0000, 0.0372, 0.3733,
         0.6626, 0.0000, 0.0372, 0.3733],
        [0.7002, 0.0000, 0.7229, 0.5590, 0.7002, 0.0000, 0.7229, 0.5590,
         0.7002, 0.0000, 0.7229, 0.5590]], device='cuda:0')
```

tensor concatenation
dim = 1 --> concat horizontally

[3, 4] @ [4,3]

example of matrix multiplication

```
# This computes the matrix multiplication between
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(tensor)
torch.matmul(tensor, tensor.T, out=y3)
```

```
<ipython-input-14-64ae95d9e5de>:6: UserWarning: An outp
  torch.matmul(tensor, tensor.T, out=y3)
tensor([[0.5732, 0.3816, 0.7126],
        [0.3816, 0.5799, 0.7048],
        [0.7126, 0.7048, 1.3366]], device='cuda:0')
```

matrix multiplication:

all of them will have the same results

y3 = torch.rand_like(tensor) --> here just initialize a random matrix in the shape of 2 x3

torch.matmul(tensor, tensor.T, out=y3)
--> out=y3 here means the multiplication output is stored in y3

```
tensor([[0.5732, 0.3816, 0.7126],
        [0.3816, 0.5799, 0.7048],
        [0.7126, 0.7048, 1.3366]], device='cuda:0')
```

```
tensor([[0.5732, 0.3816, 0.7126],
        [0.3816, 0.5799, 0.7048],
        [0.7126, 0.7048, 1.3366]], device='cuda:0')
```

Element-wise

```
# This computes the element-wise product. z1, z2, z3 wil
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)

tensor([[0.0360, 0.0000, 0.0750, 0.4614],
        [0.4391, 0.0000, 0.0014, 0.1394],
        [0.5815, 0.0000, 0.5225, 0.3125]], device='cuda:0')
```

element-wise product instead of matrix multiplication

[0, 1, 2] = 3

```
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))

4.208225250244141 <class 'float'>
```

sum up all the elements in tensor

tensor + 5

```
print(f"{tensor} \n")
tensor.add_(5)
print(tensor)

tensor([[0.1928, 0.0000, 0.2738, 0.6793],
        [0.6626, 0.0000, 0.0372, 0.3733],
        [0.7002, 0.0000, 0.7229, 0.5590]], device='cuda:0')
tensor([[5.1928, 5.0000, 5.2738, 5.6793],
        [5.6626, 5.0000, 5.0372, 5.3733],
        [5.7002, 5.0000, 5.7229, 5.5590]], device='cuda:0')
```

tensor addition --> each element add 5

```
tensor = tensor.cpu()
print(f"t: {tensor}")
n = tensor.numpy()
print(f"n: {n}")

t: tensor([[5.1928, 5.0000, 5.2738, 5.6793],
           [5.6626, 5.0000, 5.0372, 5.3733],
           [5.7002, 5.0000, 5.7229, 5.5590]])
n: [[5.191952  5.        5.273803  5.679258 ]
    [5.6626415 5.        5.0371656 5.373328 ]
    [5.7001985 5.        5.722862  5.5590167]]
```

if converting from tensor to numpy --> have to put tensor on cpu first

n = tensor.numpy()

n is specified as tensor.numpy()

```
# A change in the tensor reflects in the N
tensor.add_(1)
print(f"t: {tensor}")
print(f"n: {n}")

t: tensor([[6.1928, 6.0000, 6.2738, 6.6793],
           [6.6626, 6.0000, 6.0372, 6.3733],
           [6.7002, 6.0000, 6.7229, 6.5590]])
n: [[6.191952  6.        6.273803  6.679258 ]
    [6.6626415 6.        6.0371656 6.373328 ]
    [6.7001985 6.        6.722862  6.5590167]]
```

here adding one will apply to both tensor and the numpy array

--> as the array is created using the tensor