PyTorch
Computation Graph and GPU

Lecture 14 for 14-763/18-763

Guannan Qu

Oct 23, 2024

Today’s Agenda

- Introduction to computation graph
- GPU acceleration

Computation Graph

```
batch_loss = [] # keep a list of losses for different batches in this epoch
batch_accuracy = [] # keep a list of accuracies for different batches in this epoch
for x_batch, y_batch in train_dataloader:
    # pass input data to get the prediction outputs by the current model
    prediction_score = mymodel(x_batch)

    # compute the cross entropy loss
    loss = loss_fun(prediction_score,y_batch)

    # compute the gradient
    optimizer.zero_grad()
    loss.backward()

    # update parameters
    optimizer.step()

    # append the loss of this batch to the batch_loss list
    batch_loss.append(loss.detach().numpy())

    # You can also compute other metrics (accuracy) for this batch here
    prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
    batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/x_batch.shape[0])
```

How the backward() works?

What is .detach()?

Why calling it before converting a tensor to numpy?

Computation Graph

- When doing tensor computations, pytorch will internally record how the tensors are related to each other via a “computation graph”.

```
x = torch.tensor(2.0, requires_grad = True)
z = x*x # this should just be 4
print(z)
```

✓ 0.2s

```
tensor(4., grad_fn=<MulBackward0>)
```

This records that z is calculated from a multiplication of the x tensor

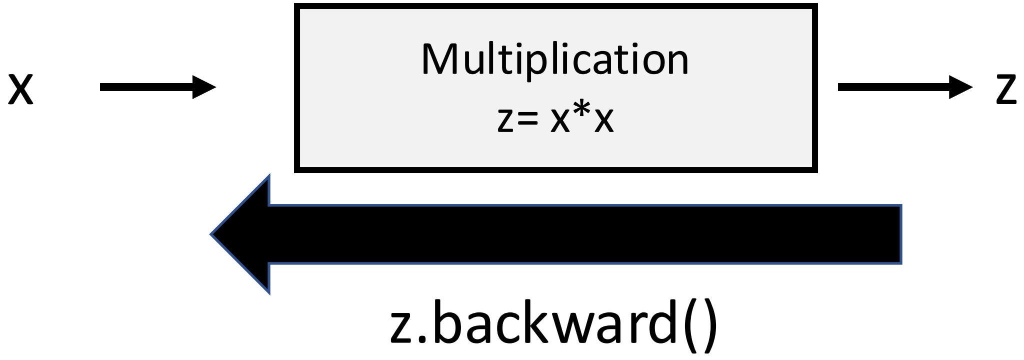
- When calling backward(), pytorch will traceback along the computation graph to compute the gradient for all tensors with requires_grad = True.

```
z.backward()

print("The gradient of z = x*x respect to x is :", x.grad)
```

✓ 0.2s

```
The gradient of z = x*x respect to x is : tensor(4.)
```

Computation Graph



- When calling backward(), pytorch will traceback along the computation graph to compute the gradient for all tensors with requires_grad = True.

```
z.backward()

print("The gradient of z = x*x respect to x is :", x.grad)
```

✓ 0.2s

```
The gradient of z = x*x respect to x is : tensor(4.)
```

Computation Graph

```
class MyLinearRegressionModel(nn.Module):
    def __init__(self,d): # d is the dimension of the input
        super(MyLinearRegressionModel,self).__init__()   # call the init f
        # we usually create variables for all our model parameters (w and
        # need to create them as nn.Parameter so that the model knows it i
        self.w = nn.Parameter(torch.zeros(1,d, dtype=torch.float))
        self.b = nn.Parameter(torch.zeros(1,dtype=torch.float))
    def forward(self,x):
        # The main purpose of the forward function is to specify given inp
        return torch.inner(x,self.w) + self.b
```

Recall the linear regression model

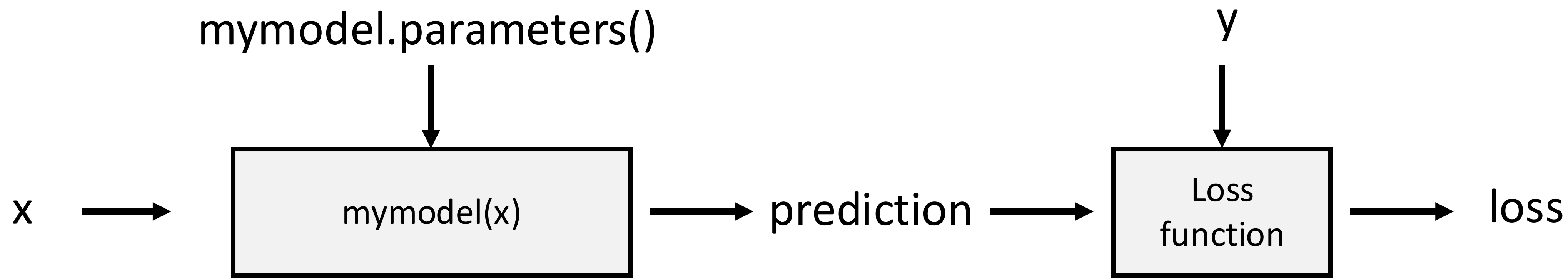
```
x = torch.arange(0,10,.1,dtype=torch.float)
x = x[:,None]
y = x*3+torch.randn(x.shape)
```

The training data x,y

```
prediction = mymodel(x)
loss = torch.mean((prediction - y)**2)

loss.backward()
```

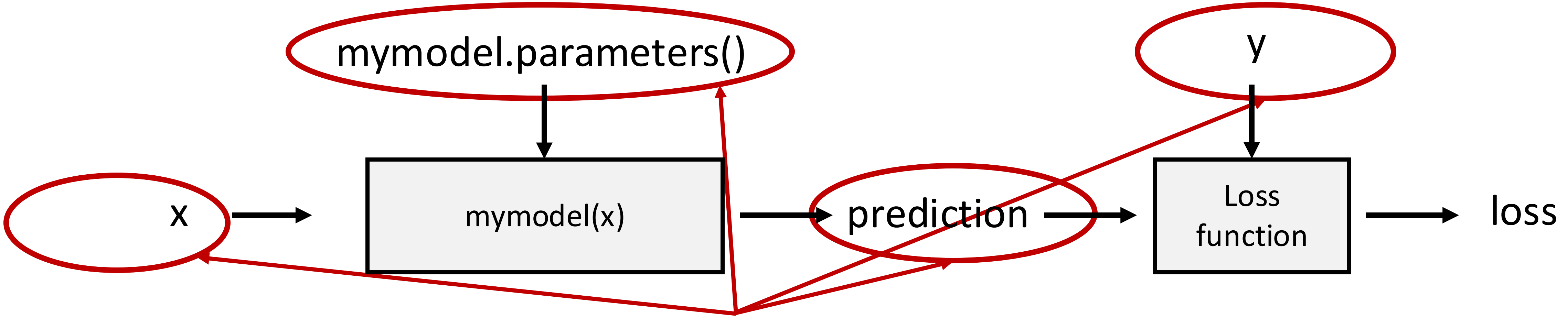
Forward/backward pass

Computation Graph



Forward: loss is computed and pytorch will record a graph representing how loss is computed
Backward: The gradient is computed via tracing back along the graph

Computation Graph

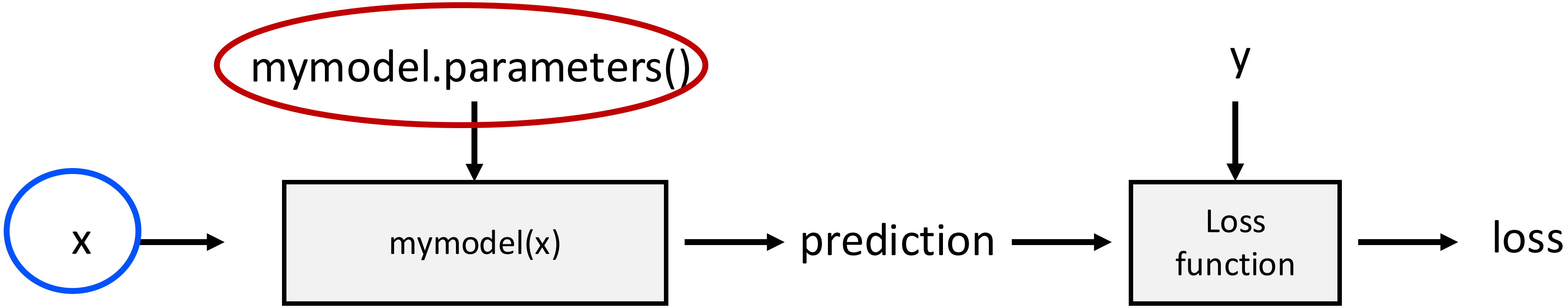


Do we need to compute all the gradients for all the tensors here?

No. Recall $loss(w, b) = \frac{1}{N}\sum_i \left(\underbrace{y_i - (wx_i + b)}_{e_i}\right)^2$

We only need to compute the gradient for w,b!

Computation Graph




```
print(f"x.requires_grad = {x.requires_grad}")
print(f"y.requires_grad = {y.requires_grad}")
```
✓ 0.0s

```
x.requires_grad = False
y.requires_grad = False
```

```
mymodel = MyLinearRegressionModel(1)
print(mymodel.w)
print(mymodel.b)
```
✓ 0.0s

```
Parameter containing:
tensor([[0.]], requires_grad=True)
Parameter containing:
tensor([0.], requires_grad=True)
```

By default, when you create a NEW tensor from scratch, the requires_grad = False

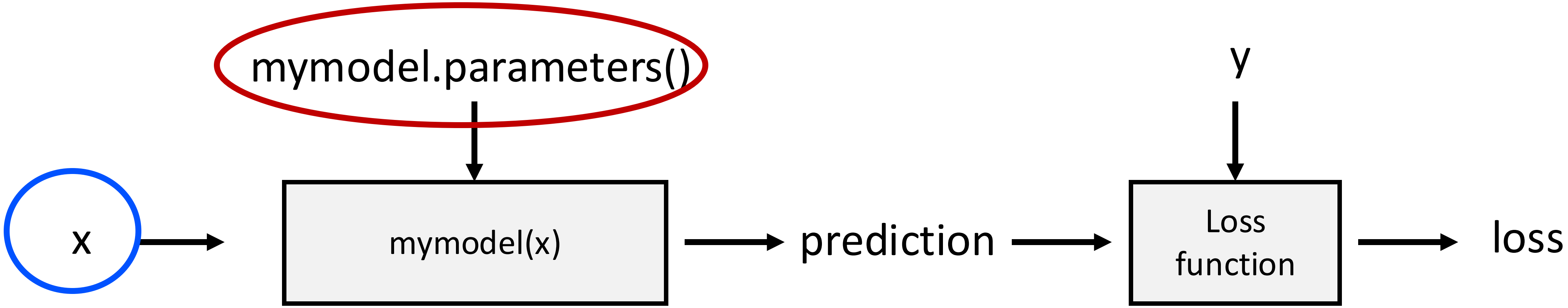
For all nn.Parameter, requires_grad = True

requires_gradient

```
from torch import nn

class MyLinearRegressionModel(nn.Module):
    def __init__(self,d): # d is the dimension of the input
        super(MyLinearRegressionModel,self).__init__()   # call the init functi
        # we usually create variables for all our model parameters (w and b in
        # need to create them as nn.Parameter so that the model knows it is an
        self.w = nn.Parameter(torch.zeros(1,d, dtype=torch.float32))
        self.b = nn.Parameter(torch.zeros(1,dtype=torch.loat32))
    def forward(self,x):
        # The main purpose of the forward function is to specify given input x,
        return torch.inner(x,self.w) + self.b
```

Recall: A parameter is a special tensor that indicates the tensor is a trainable parameter in a model. By default, all parameter has requires_grad = True

Computation Graph




```
prediction = mymodel(x)
loss = torch.mean((prediction - y)**2)

loss.backward()

print(f"mymodel.w.grad = {mymodel.w.grad}, mymodel.b.grad = {mymodel.b.grad}")
print(f"x.grad = {x.grad}, y.grad = {y.grad}")
```

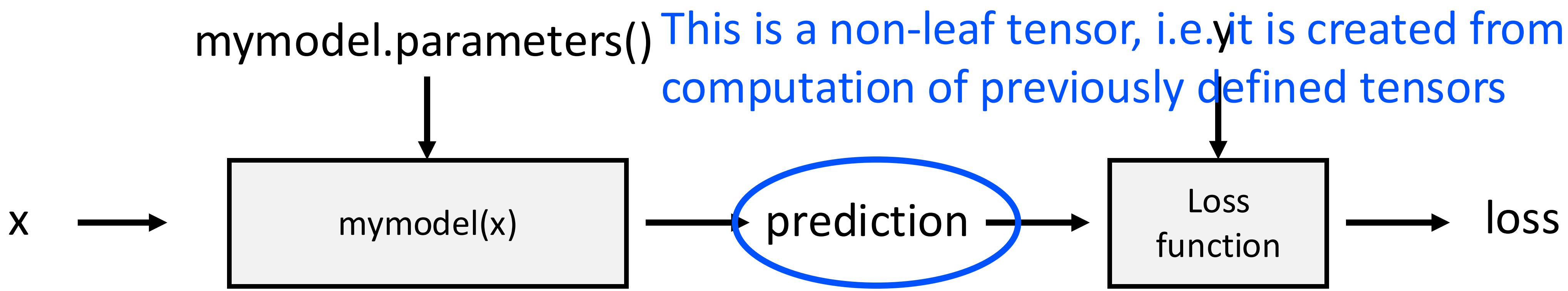
✓ 0.0s

Gradient for w,b computed

```
mymodel.w.grad = tensor([[-391.8783]]), mymodel.b.grad = tensor([-58.8862])
x.grad = None, y.grad = None
```

Gradient for x, y not computed

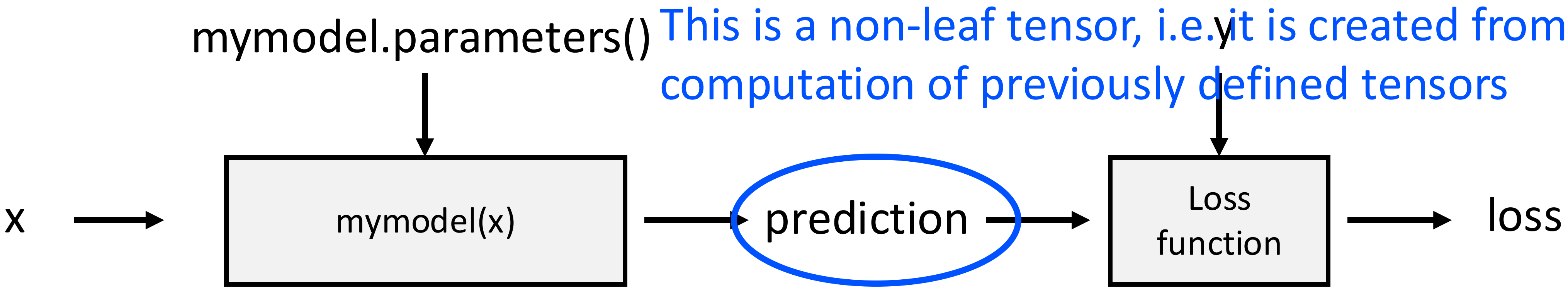
Computation Graph



For non-leaf tensors, if one of the prior tensors has requires_grad = True, then this tensor will have requires_grad = True

- Prediciton is computed from mymodel.w, mymodel.b, both of which have requires_grad = True, so prediction.requires_grad = True
- The reason is due to ”back-propagation”, the underlying algorithm for backward(). The gradient for prediction is needed for computating gradient of mymodel.w, mymodel.b

Computation Graph




```python
print(f"prediction.requires_grad = {prediction.requires_grad}, prediction.grad = {prediction.grad}" )
```

✓ 0.0s

```
prediction.requires_grad = True, prediction.grad = None
```

- For non-leaf tensors, despite the requires_grad = True, the gradient is computed but DISCARDED after the backward, as this gradient is usually not useful.
- If you want to keep this gradient value, use the retain_grad() method to this tensor.

Computation Graph

The most common scenario where you want to change the requires_grad attribute is the following:

- The detach() method
- Inference mode: with torch.no_grad()

Other more advanced scenarios where requires_grad needs to be changed:

- Fine-tuning of pretrained models
- Reinforcement learning
- Some generative AI models like GAN, diffusion

Detach()

detach() method: Make a copy of a tensor and detach it from a computation graph.

```
print("loss = ", loss, f"loss.requires_grad = {loss.requires_grad}")
loss_detached = loss.detach()
print("loss_detached = ", loss_detached, f"loss_detached.requires_grad = {loss_detached.requires_grad}")
```

✓ 0.0s

```
loss =  tensor(293.0902, grad_fn=<MeanBackward0>) loss.requires_grad = True
loss_detached =  tensor(293.0902) loss_detached.requires_grad = False
```

Detach()

detach() method: Make a copy of a tensor and detach it from a computation graph.

- When a tensor is part of a graph, pytorch does not allow you to convert it to numpy
- Therefore, we need to call loss.detach().numpy() to convert loss to numpy array!

```
print("loss_detached.numpy() = ", loss_detached.numpy())

print("loss.numpy() = ", loss.numpy())
```

⊗ 0.0s

```
loss_detached.numpy() =  293.0902

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/Users/coolq/Library/CloudStorage/Box-Box/Teaching/Tool Chain/Toolchain 2023 Fall/notebooks/Lecture_1
      1 print("loss_detached.numpy() = ", loss_detached.numpy())
----> 3 print("loss.numpy() = ", loss.numpy())

RuntimeError: Can't call numpy() on Tensor that requires grad. Use tensor.detach().numpy() instead.
```

torch.no_grad()

- Sometimes, we only want to do forward, not the backward
- E.g. for validation and testing, we only want to compute the loss, without needing to do backward()
- Recall: the validation loop below

```
# Validation loop
validate_batch_loss = [] # keep a list of losses for different validate batches in this epoch
validate_batch_accuracy = []
for x_batch, y_batch in validate_dataloader:
    # pass input data to get the prediction outputs by the current model
    prediction_score = mymodel(x_batch)

    # compare prediction and the actual output and compute the loss
    loss = loss_fun(prediction_score,y_batch)

    # append the lost of this batch to the validate_batch_loss list
    validate_batch_loss.append(loss.detach())

    # You can also compute other metrics (like accuracy) for this batch here
    prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
    validate_batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/x_batch.shape[0])
```

We only did forward, NOT backward

Regular forward pass builds the computation graph

```
prediction = mymodel(x)
loss = torch.mean((prediction - y)**2)
print(f"prediction.requires_grad = {prediction.requires_grad}", f"prediction.grad_fn = {prediction.grad_fn}")
print(f"loss.requires_grad = {loss.requires_grad}", f"loss.grad_fn = {loss.grad_fn}")
```

```
prediction.requires_grad = True prediction.grad_fn = <AddBackward0 object at 0x7f9778d34280>
loss.requires_grad = True loss.grad_fn = <MeanBackward0 object at 0x7f9788e1c2e0>
```

Place the forward under with torch.no_grad() will temporarily disable building the computation graph

```
with torch.no_grad():
    prediction = mymodel(x)
    loss = torch.mean((prediction - y)**2)
    print(f"prediction.requires_grad = {prediction.requires_grad}", f"prediction.grad_fn = {prediction.grad_fn}")
    print(f"loss.requires_grad = {loss.requires_grad}", f"loss.grad_fn = {loss.grad_fn}")
    # if you try to do loss.backward() here, an error will occur
    # loss.backward()
```

```
prediction.requires_grad = False prediction.grad_fn = None
loss.requires_grad = False loss.grad_fn = None
```

Fine Tuning

- Suppose there are existing models pretrained on a generic large dataset (e.g. animal classification, dog vs cat vs …)
- Suppose you want to do ML for a more specific animal classification problem, e.g. predicting the type of the dog (husky vs. shepherd vs …)

- Instead of building a neural network and train from scratch, it is often easier to use a pretrained model and fine-tune on your specific dataset
- What does fine-tune mean? It often means ONLY TRAIN the LAST FEW LAYERS!

```
import torchvision.models as models

resnet18 = models.resnet18(pretrained=True)

print(resnet18)
```

Importing pretrained model

Fine Tuning

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
...
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

We only want to train the last layer

```python
# Freeze all layers
for param in resnet18.parameters():
    param.requires_grad = False


# Unfreeze last layer
for param in resnet18.fc.parameters():
    param.requires_grad = True
```

"Freeze" all parameters by setting requires_grad=False

"UnFreeze" the last layer requires_grad=True

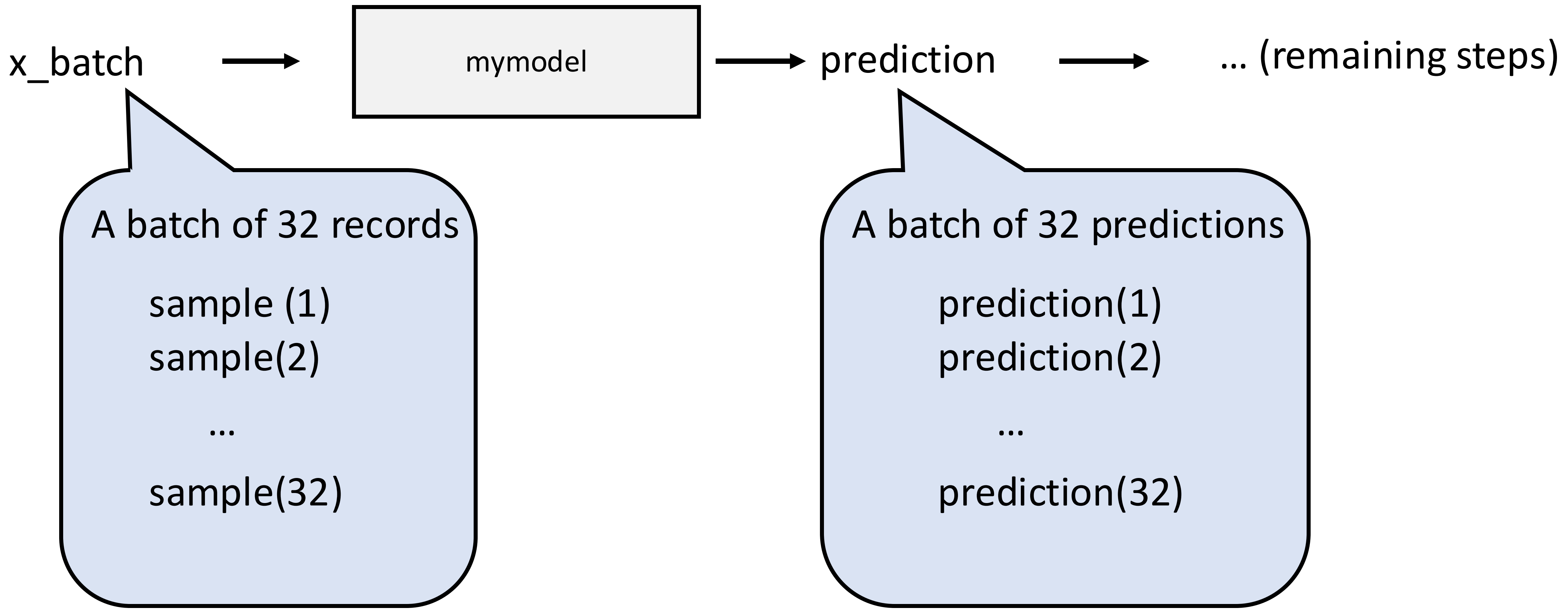
```python
# Create random data
inputs = torch.randn(5, 3, 224, 224)
labels = torch.randint(0, 10, (5,))

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(resnet18.fc.parameters(), lr=0.001, momentum=0.9)

# Training loop
for epoch in range(5):
    optimizer.zero_grad()
    outputs = resnet18(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch+1}/5, Loss: {loss.item()}')
```

Optimizer only trains the last layer's parameters

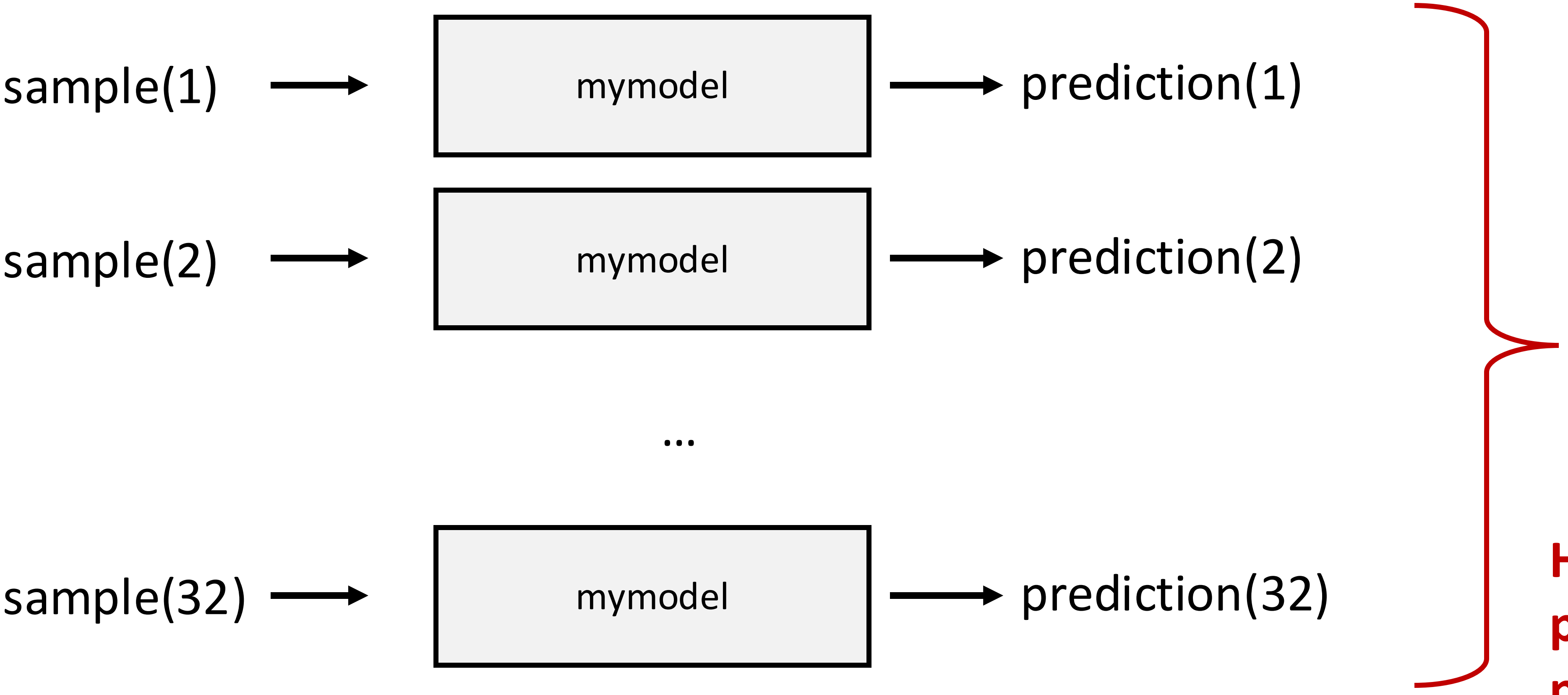
Today’s Agenda

- Introduction to Computation graph
- **GPU Acceleration**

Parallelization in Forward/Backward/Step



The forward is a parallelizable process!



The forward step needs to repeat this prediction step for every record in the batch!

How about parallelize the process?

Caution: the parallelization is WITHIN-BATCH

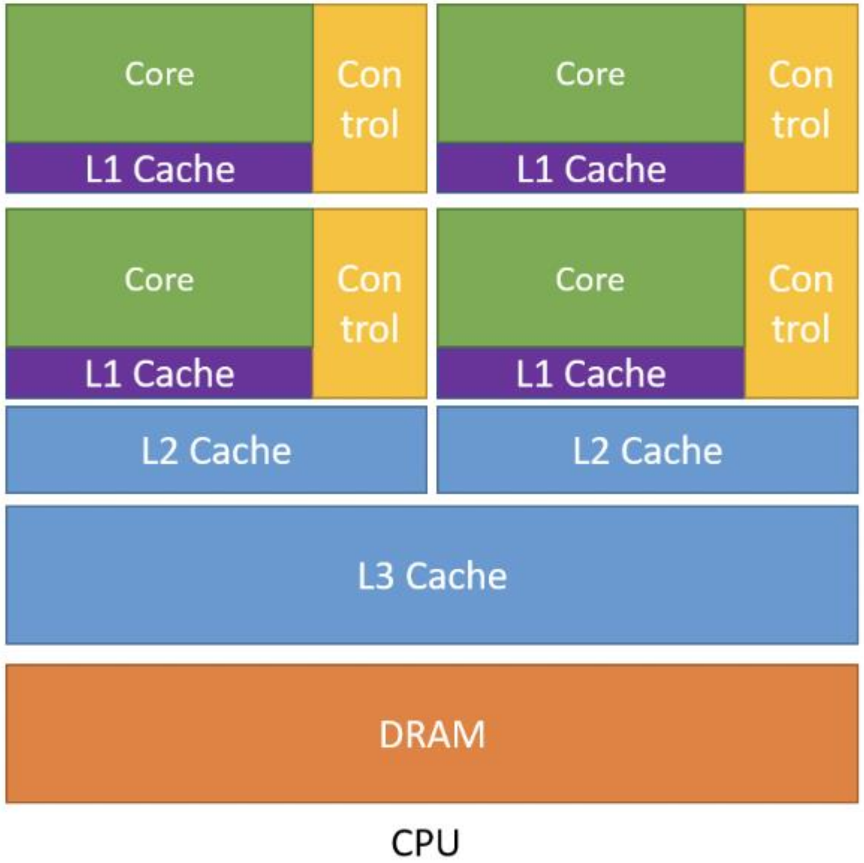
- Does GPU parallelize within a batch or across different batches? The answer is WITHIN THE BATCH.

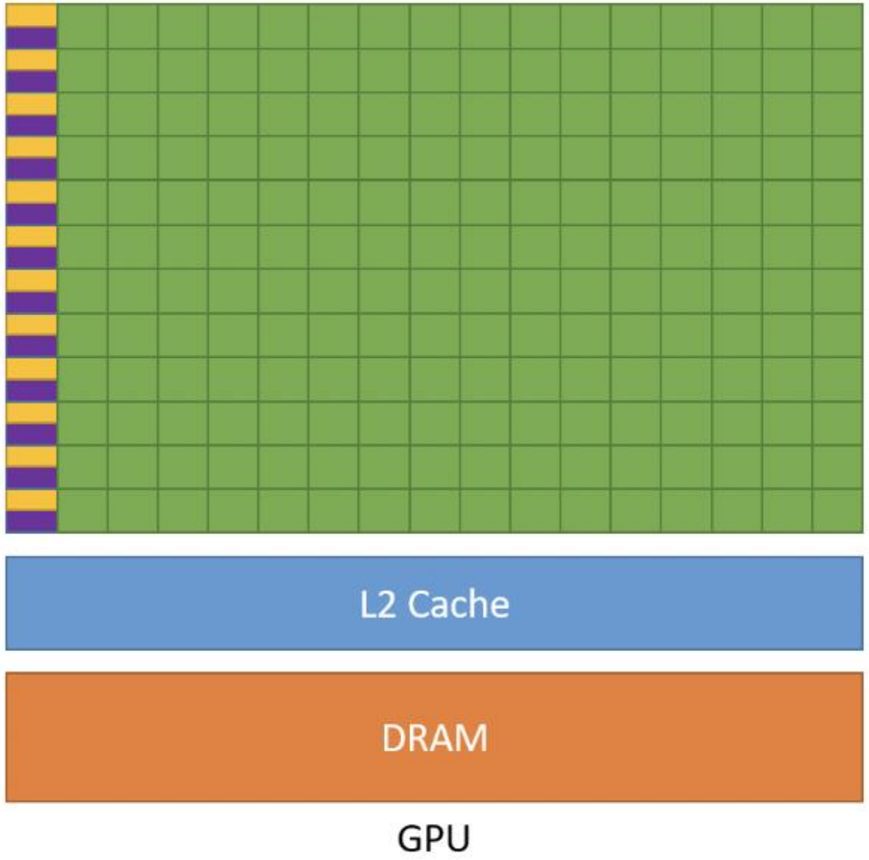
```python
for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
    start_time = time.time()
    # data to device
    x_batch = x_batch.to(device)
    y_batch = y_batch.to(device)

    # pass input data to get the prediction outputs by the current model
    prediction = mynn(x_batch)

    # compare prediction and the actual output label and compute the loss
    loss = nn.functional.cross_entropy(prediction,y_batch)

    # compute the gradient
    optimizer.zero_grad()
    loss.backward()

    # update parameters
    optimizer.step()
```

Batch comes after batch in a for loop, so parallelization is not across the batches

Parallelization happens inside a batch

GPU





For CPU, all computation happens sequentially on a single core

GPU has many cores (hundreds/thousands) suitable for parallel computations!

GPU Lineups

Series	Target Users	Model Number	Memory
GeForce RTX	Desktops/Laptops, gaming	4070/4080/4090	12 GB – 24 GB
RTX Series (formerly Quadro)	Workstations (professional video creation/editing…)	A4000	20GB
		A5000	32 GB
		A6000	48 GB
Data Center (formerly Tesla)	Data center, for ML	V100 (2018)	16 GB/32 GB
		A100 (2020)	40 GB / 80 GB
		H100 (2022)	80 GB
		B100 (2024/2025)	192 GB

(source: nvidia.com)

CUDA

- CUDA:
  - Developed by Nvidia and is in C++
  - General purpose parallel computing platform for NVIDIA GPUs
- torch.cuda:
  - Developed by PyTorch
  - A PyTorch library that uses CUDA to achieve GPU acceleration in PyTorch
- How to check if GPU is available for PyTorch to use?
  - Run torch.cuda.is_available()

Where to find GPUs?

- Buy Personal Computer/Workstation with GPU
- Free GPUs in Google Colab
- Create a cloud computing instance with GPU
- Many universities/departments/labs have GPUs

How to use GPU acceleration in PyTorch

Key: "move" model and data to GPU!

```
# device = torch.device('cpu')
device = torch.device('cuda:0')
```

Create a "device" object and set it to "cuda:0" which is the first available GPU.

```
mynn = LeNet()
mynn = mynn.to(device = device)
```

After creating a model (LeNet()) in this case, use .to(device = device) to move it to GPU device

How to use GPU acceleration in PyTorch

Key: “move” model and data to GPU!

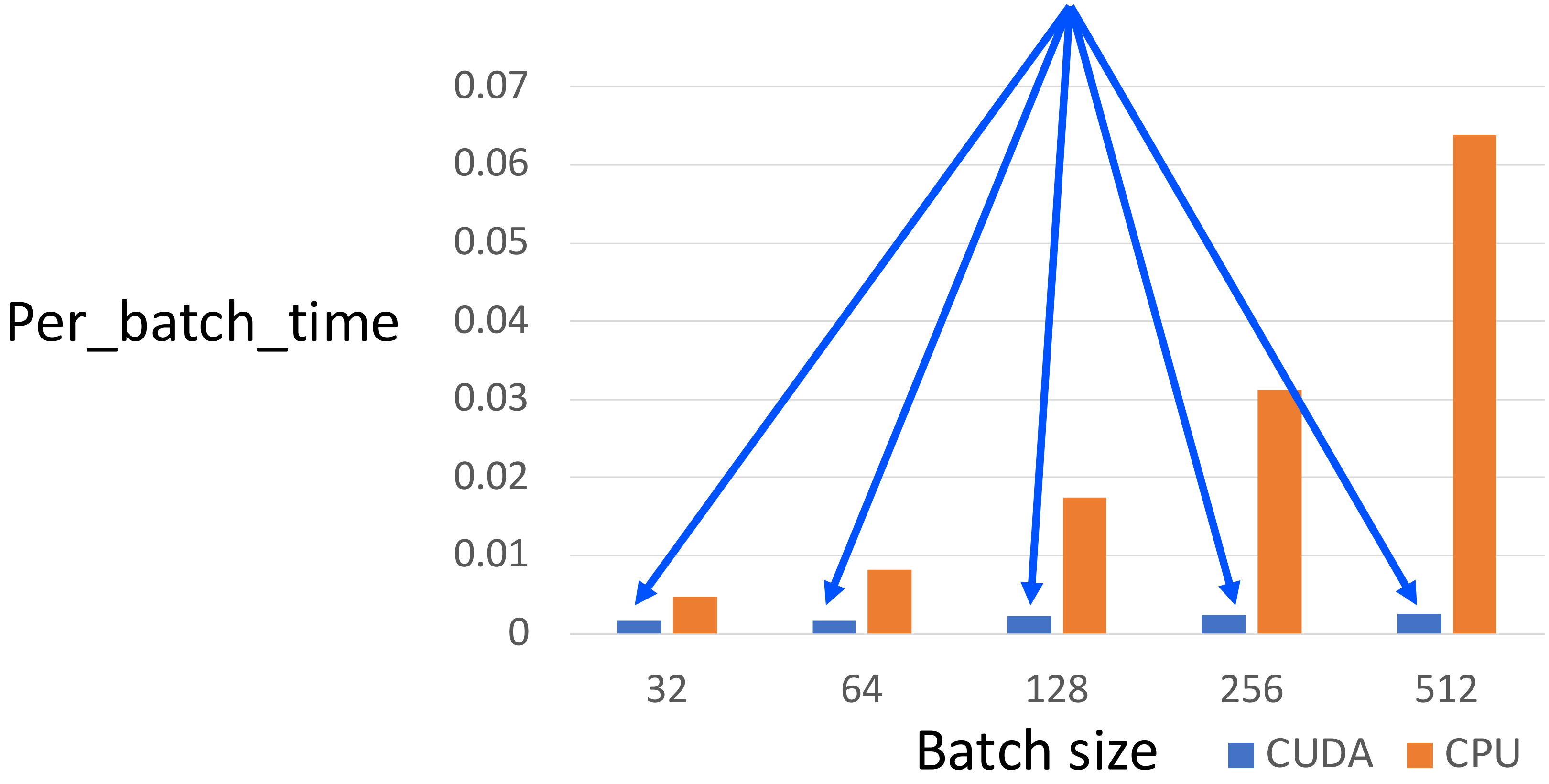
```
for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
    start_time = time.time()
    # data to device
    x_batch = x_batch.to(device)
    y_batch = y_batch.to(device)

    # pass input data to get the prediction outputs by the current model
    prediction = mynn(x_batch)

    # compare prediction and the actual output label and compute the loss
    loss = nn.functional.cross_entropy(prediction,y_batch)

    # compute the gradient
    optimizer.zero_grad()
    loss.backward()

    # update parameters
    optimizer.step()
```

Move batch data to GPU device!

The forward/zero_grad/backward/step are identical as before.

Effect of batch size

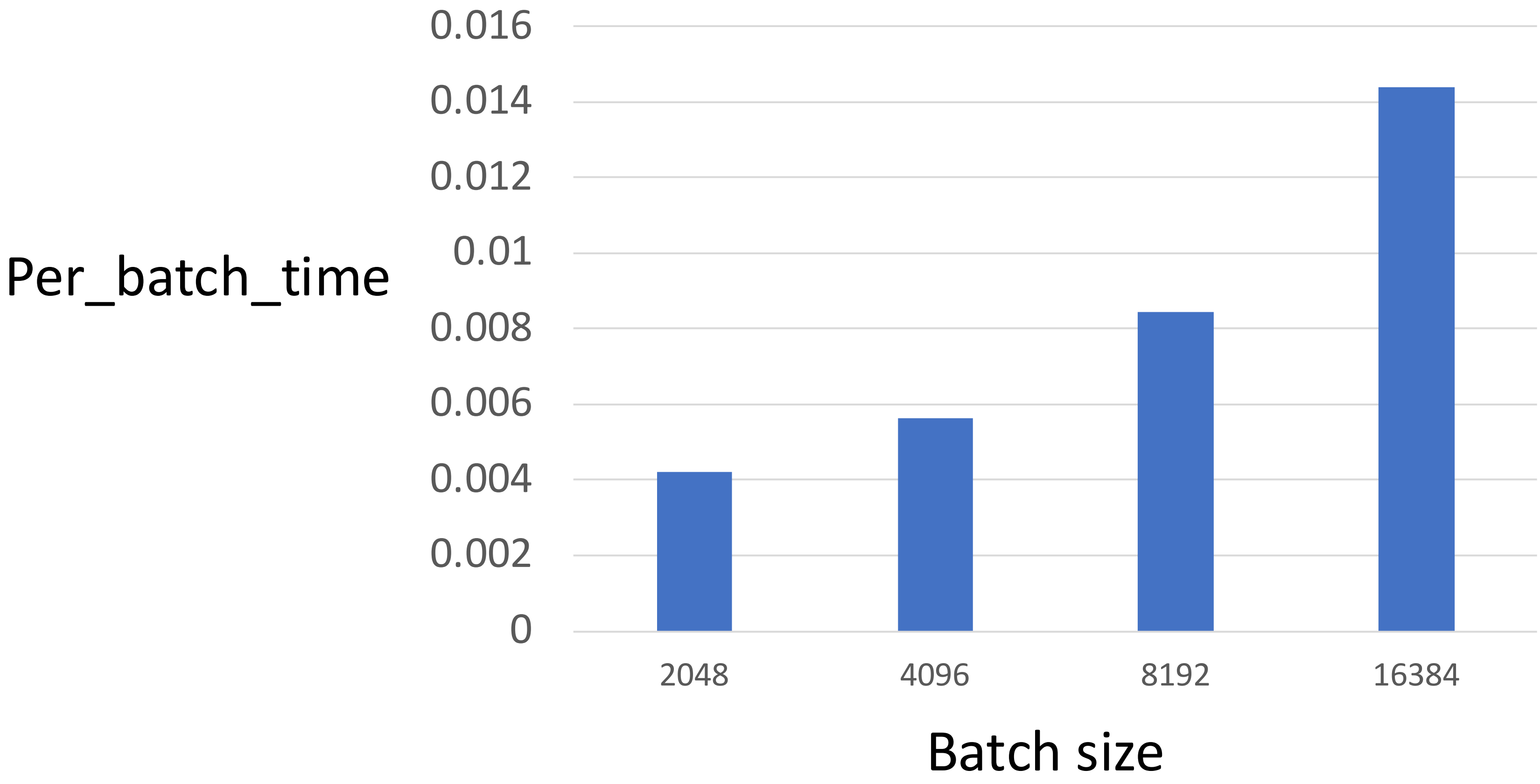
GPU computing time almost does not increase with batch_size thanks to parallelization!



Test environment: Google CoLab

Effect of batch size

When the batch_size is too large, GPU per batch time DOES INCREASE with batch size!



Test environment: Google CoLab

Tips of using GPU

- Make sure the data and the model is on the **SAME DEVICE**!
- When converting tensors to numpy, make sure it is on CPU!