CSC 461: Machine Learning

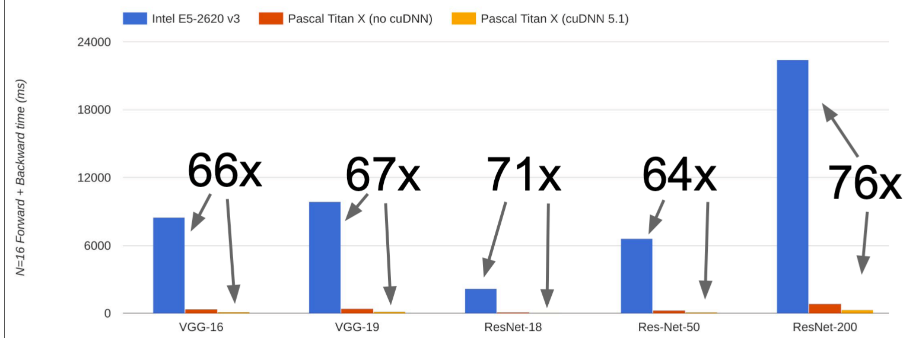
Fall 2024

PyTorch / Autograd

Prof. Marco Alvarez, Computer Science
University of Rhode Island

CPU vs GPU in practice

(CPU performance not well-optimized, a little unfair)



From CS231 @ Stanford

Autograd

Computational Graphs

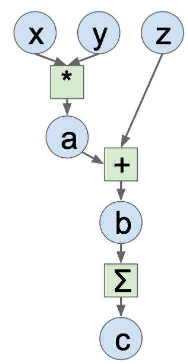
Numpy

```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



From CS231 @ Stanford

Computational Graphs

Numpy

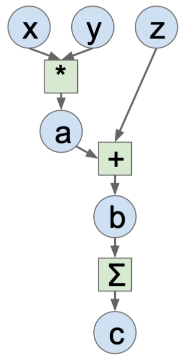
```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Computational Graphs

Numpy

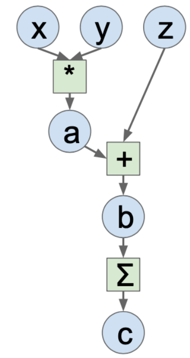
```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



Good:
Clean API, easy to write numeric code

Bad:
- Have to compute our own gradients
- Can't run on GPU

Computational Graphs

Numpy

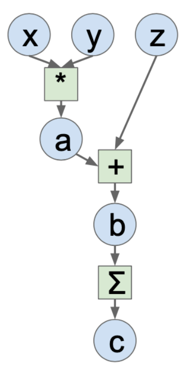
```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```python
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

Computational Graphs

Numpy

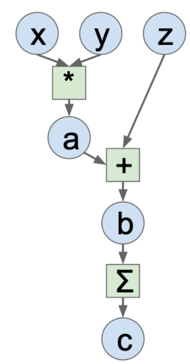
```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```python
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

Computational Graphs

Numpy

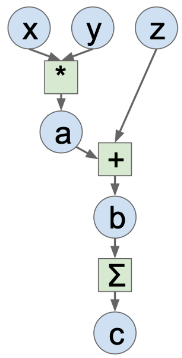
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

From CS231 @ Stanford

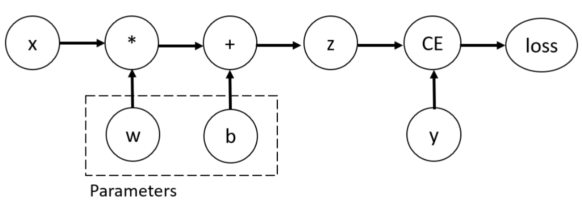
Automatic differentiation

- Modern machine learning
  - parameters (model weights) are adjusted according to the gradient of the loss function with respect to the given parameter
- PyTorch has a built-in differentiation engine
  - called `torch.autograd`
  - supports automatic computation of gradient for any computational graph

Example

- Consider multinomial logistic regression
  - input x, parameters w and b, and some loss function

```
x = torch.rand(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 = x @ w + b
loss = F.binary_cross_entropy_with_logits(z, y)
```



Example

- Computing gradients

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

```
tensor([[0.0360, 0.2630, 0.0477],
        [0.0218, 0.1590, 0.0289],
        [0.0367, 0.2680, 0.0486],
        [0.0181, 0.1320, 0.0240],
        [0.0221, 0.1613, 0.0293]])
tensor([0.0436, 0.3186, 0.0578])
```

Gradient tracking

- Tracking gradients
  - by default all tensors with `requires_grad=True` are tracking their computational history and support gradient computation
- Stop tracking computations
  - there are some cases when we do not need tracking
    - e.g., trained model and just want to perform inference, i.e. we only want to do forward computations
  - advantages:
    - mark some parameters as **frozen parameters**
    - speed up computations when you are only doing forward pass (more efficient)

```
with torch.no_grad():
    z = x @ w + b
```

Forward and backward computation

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = 1 / (1 + torch.exp(-(w@x + b)))

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

Forward and backward computation

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = torch.sigmoid(w @ x + b)

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

Datasets and Dataloaders

Working with datasets

- PyTorch provides two data primitives
  - `torch.utils.data.Dataset`
    - allows you to use pre-loaded datasets as well as your own data
    - stores the samples and their corresponding labels
  - `torch.utils.data.DataLoader`
    - wraps an iterable around the Dataset to enable easy access to the samples
- PyTorch domain libraries provide a number of pre-loaded datasets
  - datasets subclass torch.utils.data.Dataset and implement functions specific to the particular data
  - can be used to prototype and benchmark models
  - include: Image Datasets, Text Datasets, and Audio Datasets
- Example: FashionMNIST
  - dataset of Zalando's article images consisting of 60,000 training examples and 10,000 test examples
  - each example comprises a 28×28 grayscale image and an associated label from one of 10 classes

Forward and backward computation

```
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Working with datasets

- Datasets
  - retrieves our dataset's features and labels one sample at a time
- Dataloaders
  - iterable that provides efficient mini-batch handling
  - can reshuffle the data at every epoch and use Python's multiprocessing to speed up data retrieval

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(
    training_data,
    batch_size=64,
    shuffle=True)

test_dataloader = DataLoader(
    test_data,
    batch_size=64,
    shuffle=False
)
```

```
X = torch.rand(20, 5)
loader = DataLoader(X, batch_size=5, shuffle=True)

for batch in loader:
    print(batch.shape)
```

Building models

- Models comprise of layers/modules that perform operations on data
  - `torch.nn` namespace provides all the needed building blocks
  - every module subclasses the `nn.Module`
  - every `nn.Module` subclass implements the operations on input data in the `forward` method

```
class MultinomialLR(nn.Module):
    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.linear = nn.Linear(in_dim, out_dim)
        self.activation = nn.Softmax(dim=1)

    def forward(self, x):
        logits = self.linear(x)
        probs = self.activation(logits)
        return probs

model = MultinomialLR().to(device)
print(model)
```

Training/Testing a model

```
# set the model to training mode
model.train()
for batch in dataloader:
    pred = model(batch.x)
    loss = loss_fn(pred, batch.y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

```
# set the model to evaluation mode
model.eval()
test_loss = 0
with torch.no_grad():
    for batch in dataloader:
        pred = model(batch.x)
        test_loss += loss_fn(pred, y).item()
test_loss /= len(dataloader)
```