Deeplearning Hardware and Software

1. CPU vs. GPU

① CPU : few cores. can do 20 things parallel (V). fast. work pretty independently → has cache but really small. use RAM to work → good for general

② GPU : thousands of cores. much slower and dumber. can't work independently → has own large RAM on the chip. & cache system → specialized for highly parallelized algorithms
e.g. matrix multiplication

(1) CUDA (NVIDIA only) : write C-like code that runs directly on GPU ← tricky..
→ libraries : cuBLAS, cuFFT, cuDNN etc. : cuDNN much faster than unoptimized CUDA

(2) OpenCL : usually slower

③ TPU : specialized HW for DL

2. GPU Communication

잘못하면 data 읽거나 GPU로 가져올 때 병목현상 일어날 수도. GPU가 너무 빨라서 data 그냥 읽는건 bottleneck 유발가능

→ Solution:

① data가 작거나 RAM 용량이 크면 disc에서 데이터 전부를 한 번에 RAM으로 읽어오기

② HDD 말고 SSD 쓰기

③ 여러 CPU 쓰레드 to prefetch data

3. Deep Learning SW

⇒ Why DL framwork?

(1) build big computation graphs easily

(2) compute gradients in computational graphs easily

(3) run efficiently on GPU → cuDNN 같은 거 몰라도 됨

4. PyTorch

① Fundamental concepts

(1) Tensor : numpy array 같은 것. can run on GPU

(2) Autograd : package for building computational graph. and compute gradients

(3) module : A NN layer ; may store state or learnable weights

원하는 경우 forward/backward pass를 만들어서 넣을 수도 있음.

② Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)
```

x, y는 gradient 사용 x.
w1, w2는 gradient 사용 o.

```
learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

Pytorch가 알아서 gradient 계산해주므로
⇒ manually programming할 때처럼 중간 산출물 알 필요 ⊗
⇒ grad 계산
→ 이부분에 대해서는 compatational graph 그리지 마라!

.zero_() inplace!

③ nn

```
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))
```

→ learnable weights define

```
learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

→ loss function
→ backward pass (model.Sequential 내 모든 layer도 requires_grad)
→ GD

④ optim

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

→ learn할 params
→ loss fn
→ backward
⇒ Optimize

⑤ nn define new modules

· module can contain module

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

learnable
→ module define. backward는 autograd로
→ module (forward)
→ optimizer (backward)
→ loss
→ grad
→ update

⑥ custom module subclass

```
import torch

class ParallelBlock(torch.nn.Module):
    def __init__(self, D_in, D_out):
        super(ParallelBlock, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, D_out)
        self.linear2 = torch.nn.Linear(D_in, D_out)
    def forward(self, x):
        h1 = self.linear1(x)
        h2 = self.linear2(x)
        return (h1 * h2).clamp(min=0)

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
            ParallelBlock(D_in, H),
            ParallelBlock(H, H),
            torch.nn.Linear(H, D_out))

optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

→ layer 만들기
→ Sequential 안에 만든 module 넣어서 사용

⑦ data loaders

```
import torch
from torch.utils.data import TensorDataset, DataLoader

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

loader = DataLoader(TensorDataset(x, y), batch_size=8)
model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
for epoch in range(20):
    for x_batch, y_batch in loader:
        y_pred = model(x_batch)
        loss = torch.nn.functional.mse_loss(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

→ Dataloader : 이게는 batch로 나누는
→ data 가져오는

⑧ pretrained model (e.g. vgg 16) 이나 다양한 시각화 플랫폼 사용 가능

⑨ Dynamic computation graphs : feed forward 하고 loss 계산할 때마다 그래프 다시 그림 → 같은 algorithm 반복인 경우 Inefficient

Static computation graphs ; iteration 전에 build-graph() 해서 graph 만들어 두고 iteration에서는 graph run 만

5. Tensorflow : 2.0 이전에는 static graph가 default. 현재는 dynamic graph로 ..

① 2.0 이전 : static graph

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
diff = y_pred - y
loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))

grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

with tf.Session() as sess:
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),
              w2: np.random.randn(H, D),
              y: np.random.randn(N, D),}
    out = sess.run([loss, grad_w1, grad_w2],
                   feed_dict=values)
    loss_val, grad_w1_val, grad_w2_val = out
```

② 2.0 이후 : dynamic . eager mode

tensor는 변경 불가. Variable은 변경 가능

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
gradients = tape.gradient(loss, [w1, w2])

w1.assign(w1 - learning_rate * gradients[0])
w2.assign(w2 - learning_rate * gradients[1])
```

→ dynamic computation graph
→ tf.losses 내장된 loss function 사용 가능
→ 만든 그래프 통해서 weight의 gradient 계산
→ update

③ Keras : High-level Wrapper

tf.keras.Sequential, tf.keras.layers, optimizer.apply_gradients, model.compile, tf.keras.losses 등 사용해서 NN 구성

④ @tf.function : compile static graph

→ 이거 뒤에 함수 정의하고 그 뒤 iteration 안에서 train하면 매번 graph 새로 그릴 필요 ⓧ ⇒ 속도 조금 빨라짐

forward pass 하고 loss 계산해서
y_pred, loss return

⑤ tensor board 등 플랫폼 사용 가능. 한 그래프 나눠서 여러 device에서 compute하는 것도 가능

6. Dynamic graph : RNN, Recursive, Modular 등 architecture가 일정하지 않은 모델에 사용

7. Pytorch vs. Tensorflow

① Pytorch : default dynamic . Static ver : Caffe2, ONNX

② Tensorflow : default Static . Static : @tf.function
(pre-2.0)
dynamic : Eager
(post-2.0)