

cs231n. Lecture 8. DeepLearning Software

content

1. CPU vs GPU

1.1 GPU

1.2 CPU

- 1.2.1 Intro_CPU
- 1.2.2 Simple execution of CPU

1.3 Difference btn

2. Deeplearing Frameworks

3. Tensorflow in detail

3.1 The setting of our example

3.2 Learn by doing

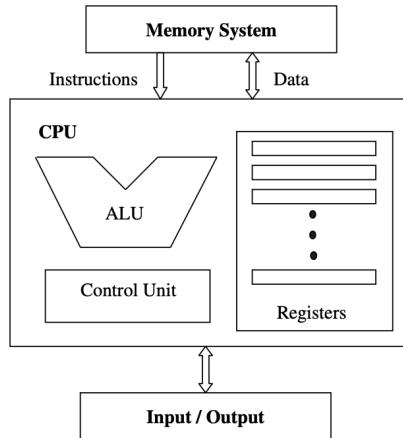
1. CPU vs GPU

1.1 GPU란 Graphic Processing Unit의 약자이고, 이것은 computer graphics를 렌더링하기 위해 만들 어졌다. 대체로 GPU 제조사는 NVIDIA와 AMD가 가장 유명한데, 이들 중 심층 학습에 더 적절한 GPU는 NVIDIA의 것이다.

1.2 반면에, CPU란 Central Processing Unit의 약자이고, 이것은 컴퓨터에서 기억, 해석, 연산, 제어라는 4 대 기능을 종합하는 장치이다. 이것은 소위 컴퓨터의 대뇌 역할을 한다고 말해진다. (추가적으로 말하자면, RAM은 컴퓨터의 단기기억을 담당하고, HDD, SSD는 장기기억을 담당한다.)

1.2.1 CPU 소개1

CPU는 보통 아래와 같이 구성되어 있고, 아래의 5 가지 역할을 수행한다고 요약될 수 있다:



CPU $\ni \{\text{레지스터 셋}, \text{ALU}, \text{CU}\}$

1. 레지스터 셋:

이것은 보통 일반 목적 레지스터와 특수 목적 레지스터를 결합하여 구성된다. 일반 목적 레지스터는 모든 목적을 위해 사용되고, 특수 목적 레지스터는 CPU 내의 특정 함수만을 갖는다.

예를 들자면, program counter (PC)는 다음에 수행되는 명령어의 주소를 저장하는 데 사용된다. 또한 instruction register(IR)은 현재 수행되는 명령어를 저장하는 데 사용된다.

2. ALU arithmetic logic unit:

이것은 산술 연산, 논리 연산을 수행하는 데 사용되고, 명령어 셋에서 요구하는 연산을 shift 한다.

3. 제어 장치 (CU):

이것은 명령어 인출, 해독, 실행에 관련된 제어 장치 생성을 책임진다.

1.2.2 Simple execution cycle of the CPU

- (1) PC에서 얻은 주소를 갖고 있는 다음에 수행될 명령어를 메모리에서 가져오고 IR에 그것을 저장한다.
- (2) 명령어 해독
- (3) 피연산자를 메모리에서 가져오고, 필요하다면, 이것을 CPU 레지스터에 저장한다
- (4) 명령어를 수행한다.
- (5) 계산 결과는 CPU 레지스터에서 메모리로 옮겨진다.

¹ Abd-El-Barr M., El-Rewini H, "Fundamentals of Computer Organization and Architecture", Wiley (2005)

1.3 차이

CPU vs GPU

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	\$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	\$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	\$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	\$399

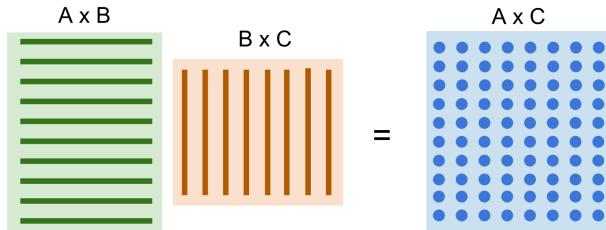
CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

GPU: More cores, but each core is much slower and “dumber”; great for parallel tasks

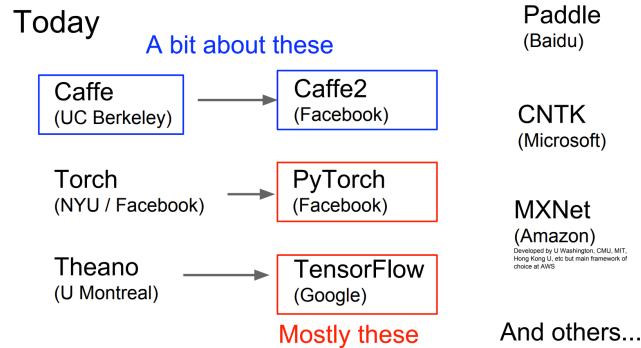
(1) CPU는 core 개수가 적지만, 각 코어 속도가 빠르다. 반면에, GPU는 코어 개수가 많지만, 각 그들의 속도는 느리다. 하지만 이들의 코어를 1:1로 비교하면 안된다. GPU 경우에, 코어의 수가 많다는 것은 일을 병렬적으로 처리한다는 것을 함축하기 때문이다.

(2) 코어들이 독립적으로 일을 수행하는 CPU는 범용 처리에 더 능하지만, 병렬 처리에 더 특화되어 있는 GPU는, CPU 보다 행렬곱 연산 같은 것에 더 능하다.

Example: Matrix Multiplication



2. Deep Learning Frameworks (대세적인 딥러닝 프레임워크는 학계 → 산업계로 옮겨갔다)



Q. 왜 위 같은 딥러닝 프레임워크를 사용해야 하는가?

- (1) 이것을 사용하면 크고 복잡한 연산 그래프를 직접 만들지 않아도 된다.
- (2) gradient를 자동으로 계산해준다
- (3) 효과적으로 GPU를 다룰 수 있게 해준다.

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
```

TensorFlow

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

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

    grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                  feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch
from torch.autograd import Variable

N, D = 3, 4

x = Variable(torch.randn(N, D).cuda(), requires_grad=True)
y = Variable(torch.randn(N, D).cuda(), requires_grad=True)
z = Variable(torch.randn(N, D).cuda(), requires_grad=True)

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

c.backward()

print(x.grad.data)
print(y.grad.data)
print(z.grad.data)
```

- **Numpy case:** 직접 gradient를 구해줘야 하는 번거로움이 있다. 또한, gpu로 실행 못시킨다.

- **TensorFlow case:** placeholder를 통해 순전파 계산을 쉽게 만들어주고, gradients 함수를 통해 grads를 자동으로 구해준다. 또한, tf.device('/gpu(or cpu):0')을 통해 gpu/cpu 선택 사용이 가능하다.

- **Pytorch case:** 위와 마찬가지로 gradients를 자동적으로 구해주는 기능이 있다. 마찬가지로, cuda()를 사용하여 gpu 사용이 가능하다.

3. TensorFlow 자세히 알아보기

3.1 Setting of the example:

Two-Layer ReLU network on random data with L2 loss

3.2 Learn by doing:

TensorFlow: Neural Net

First define
computational graph

Then run the graph
many times

```
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
```

(1) 연산 그래프 정의하기 (Loss, grads 구하기 위해 그래프 만들기)

(2) 위 그래프 (외부에서) 여러 번 실행

TensorFlow: Neural Net

Problem: copying
weights between CPU /
GPU each step

Train the network: Run
the graph over and over,
use gradient to update
weights

```
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),}
    learning_rate = 1e-5
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2],
                      feed_dict=values)
        loss_val, grad_w1_val, grad_w2_val = out
        values[w1] -= learning_rate * grad_w1_val
        values[w2] -= learning_rate * grad_w2_val
```

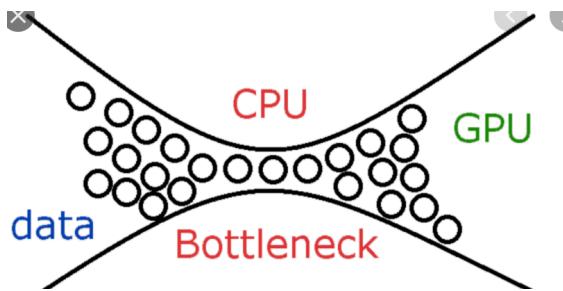
연산 그래프 외부에서 가중치 업데이트: 그래디언트 계산 => np_array로 가중치 업데이트

Analysis of (2):

3.2.1 그래프 외부에서 가중치 업데이트

여러 번 그래프를 실행하려면 for 문을 사용하면 된다. sess.run()이 실행될 때마다 loss, grads를 구해준다. session을 실행해주기 위해 np_array인 x,w1,w2,y를 넣어주면, 그 출력값으로 np_array인 loss, grad1, grad2가 나온다.

그런데 문제가 있다. Forward pass 시 그래프가 실행될 때마다 가중치를 넣어feed주어야 한다. 그래프가 한 번 실행될 때마다 가중치와 동일한 크기를 가진 grads가 반환된다. 이 반환이 함의하는 바는 그래프 실행 마다 Tensorflow에서 numpy array인 가중치 행렬을 복사한다는 것이다. 문제는 이 것이다: 이 경우에 GPU와 CPU 간 메모리 상의 데이터 교환이 있을 수 있는데, — 네트워크가 크고 가중치가 많은 경우에 — 이것은 GPU의 병목현상bottleneck을 일으킬 수 있다.



텐서플로우에서는 이 문제를 해결할 방법이 있다. 다음 같이 placeholder로 저장하던 것을 Variable로 저장해주는 것이다. variable은 매 시간마다 연산 그래프 내에 있는 변수이다.

3.2.2 placeholder ==> variable

이제 이 변수가 그래프 내에 있기 때문에, 우리는 이것을 초기화해줄 필요가 있다. tf.random_normal는 어떻게 변수들이 초기화되어야 하는지 알려준다.

TensorFlow: Neural Net

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

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}

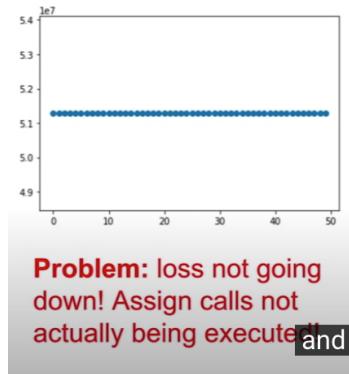
for t in range(50):
    loss_val, = sess.run([loss], feed_dict=values)
```

Run graph once to
initialize w1 and w2

Run many times to train

또한, 위 코드에서 나타나듯이, 우리는 이제 그래프 내부에서 assign 함수를 이용하여 w1, w2를 업데이트 시킬 수 있다.

session()을 통해 학습을 할 때, 그래프 내에 있는 변수들을 초기화시켜주기 위해, tf.global_variables_initializer()을 사용한다. 그 뒤에 그래프를 계속 반복하여 실행시켜준다.



하지만 위 같이 버그가 나타난다. 손실이 내려가지 않았다. 즉 학습이 제대로 되지 않았다. 왜 그런 걸까?

3.2.3 그래프에 더미 노드 추가하기

TensorFlow: Neural Net

```

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

learning_rate = 1e-5
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)
updates = tf.group(new_w1, new_w2)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D)}
    losses = []
    for i in range(50):
        _, loss_val = sess.run([loss, updates],
                             feed_dict=values)
        losses.append(loss_val)

```

Add dummy graph node that depends on updates

Tell graph to compute dummy node

these values as outputs.

위 곤경을 해소할 방안은 바로 그래프에 더미 노드를 추가하는 다소 tricky한 방식이다. 우리는 이 허위 데이터 의존성 fake data dependencies을 이용하여, 이 더미 노드가 새로운 w1, w2를 업데이트

해준다고 말할 수 있다. 이 더미 노드는 어떤 것도 반환하지 않지만, 이 의존성 때문에 우리는 값을 업데이트 할 수 있고, 그래서 실제로 업데이트 연산을 수행하는 것이다.

3.3 최적화

TensorFlow: Optimizer

Can use an **optimizer** to
compute gradients and
update weights

Remember to execute the
output of the optimizer!

```
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.random_normal((D, H)))
w2 = tf.Variable(tf.random_normal((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 * diff, axis=1))

optimizer = tf.train.GradientDescentOptimizer(1e-5)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    losses = []
    for t in range(50):
        kind of stuff for you. sess.run([loss, updates],
                                       feed_dict=values)
```

또한 우리는 위 함수를 이용하여 grad를 구하고 W를 업데이트 하기 위해 최적화를 사용할 수 있다. 이 함수는 환상적이다. 왜냐하면 이 w1, w2 변수가 디폴트에 의해 학습될 수 있다고 나타나기 때문에, 이 optimizer.minimize 함수는 그 안에서 내부적으로 w1, w2에 대한 손실의 그래디언트를 계산하는 그래프에 노드들을 추가해주기 때문이다.

3.4 손실

TensorFlow: Loss

Use predefined
common losseses

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

h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-3)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        loss_val, _ = sess.run([loss, updates],
                              feed_dict=values)
```

L2 정규화도 이 한 줄을 가지고 사용할 수 있다.

3.5 예

TensorFlow: Layers

Use Xavier
initializer

tf.layers automatically
sets up weight and
(and bias) for us!

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

init = tf.contrib.layers.xavier_initializer()
h = tf.layers.dense(inputs=x, units=H,
                     activation=tf.nn.relu, kernel_initializer=init)
y_pred = tf.layers.dense(inputs=h, units=D,
                         kernel_initializer=init)

loss = tf.losses.mean_squared_error(y_pred, y)

optimizer = tf.train.GradientDescentOptimizer(1e-0)
updates = optimizer.minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              y: np.random.randn(N, D),}
    for t in range(50):
        So one example that ships with TensorFlow, [feed_dict=values], [updates], [feed_dict=values], [updates]
```

- 데이터와 레이블 즉 x,y만을 placeholder로 선언한 것에 주목하라.
- tf.layers.dense()은 내부적으로 그래프 내부에 있는 올바른 shape을 가진 w1, b1를 설정해준다. 하지만 이것은 우리에게 보이지 않는다.