

Lecture 8 | Deep Learning Software

오늘 배울 것들

- CPU vs GPU
- Deep Learning Frameworks
 - Caffe / Caffe2
 - Theano / TensorFlow
 - Torch / PyTorch

CPU vs GPU

CPU : Central Processing Unit

- 작은 크기

GPU

- 굉장히 큼 , 쿨링 팬 있음

GPU : 렌더링이나, 게임 하는데에 사용됨.

NVIDIA vs AMD ⇒ 딥러닝에는 NVIDIA가 더 좋음.!

	# Cores	Clock Speed	Memory	Price	
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	\$339	CPU: Fewer cores, but each core is much faster and much more capable; great at sequential tasks
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: More cores, but each core is much slower and “dumber”; great for parallel tasks
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	\$399	

CPU

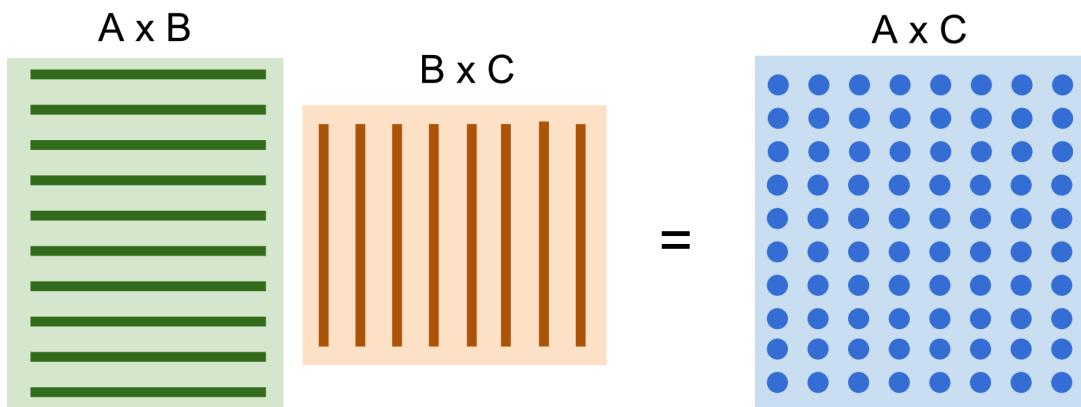
- 적은 코어 수(4 ~ 10), 8~20 쓰레드
- cache 메모리 존재. (적음)

GPU

- 상당히 큰 코어 수(수천 개)
- CPU 코어랑은 조금 다름. 각각의 코어가 독립적으로 실행 불가
- 독립적인 메모리 존재
- 캐싱 기능도 있음

예를 들어, 아래와 같은 행렬 곱 연산을 할 때, GPU는 각 계산을 병렬적으로 실행 가능. 하지만 CPU는 하나 하나 연속적으로 실행함.

Example: Matrix Multiplication



Programming GPUs

- CUDA

GPU만을 사용하여 프로그래밍이 가능하게끔 만들어져 있음.

CUDA는 NVIDIA에서만 사용 가능하며, 여러가지 API들이 존재.

- OpenCL

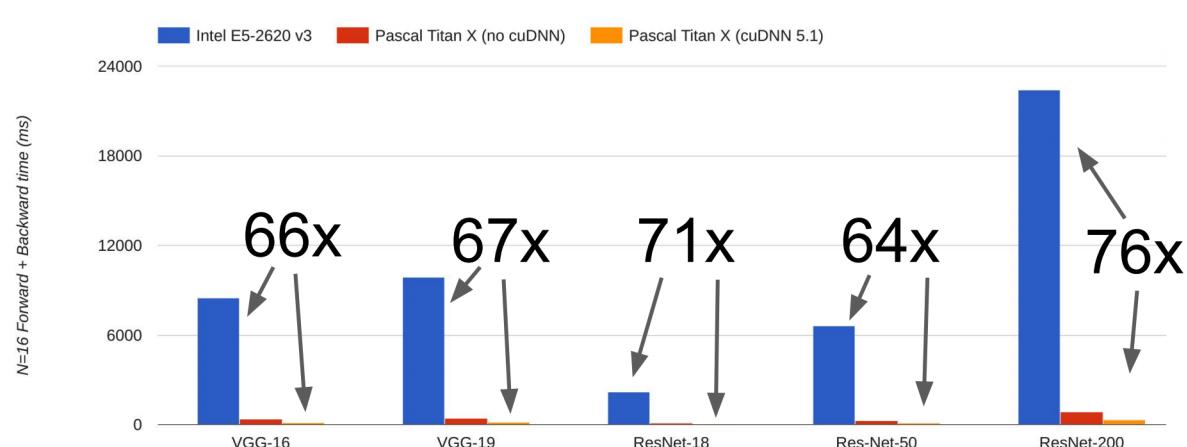
이와 유사하게 OpenCL이라는 것도 있는데, 이는 GPU만을 사용하지는 않음. 주로 CUDA보다는 느낌.

- Udacity

병렬 프로그래밍의 시초라고 할 수 있음.

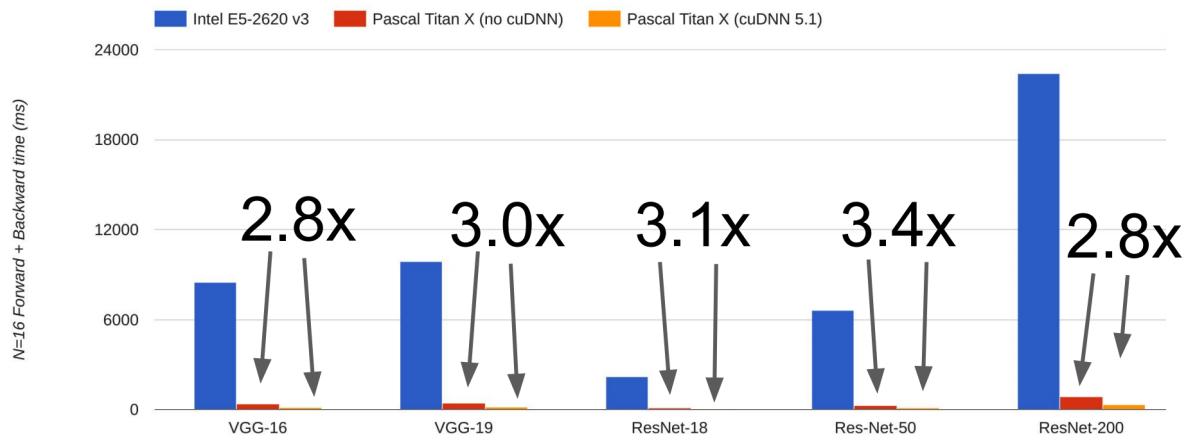
아래의 그림은 CPU와 GPU를 이용하여 딥러닝을 하였을 때 걸리는 시간을 비교한 것이다.

CPU가 비교할 수도 없게 많은 시간이 걸린다는 것을 알 수 있다.



하지만 이러한 비교는 적합하지 않다. 왜냐하면 여러가지 프로그램들이 GPU에 맞게 프로그래밍되어있기 때문이다.

그래서 cuDNN을 사용하고, 안하고 2개의 GPU를 비교해보면 cuDNN을 사용하는 경우가 3배가량 더 빠르다는 결과를 얻을 수 있다.



여기서 문제는, 우리가 GPU에서 연산을 하지만 데이터는 저장공간(HDD)에 들어있다는 사실이다. 이를 조금 개선하기 위해서는 RAM에서 데이터를 다 읽어와서 training을 하거나, SSD를 사용하고, 데이터를 prefetch할 때 CPU multi-thread를 사용하는 것이다.

Deep Learning Frameworks

이는 매년 변한다는 사실을 알고 있어야 한다. 왼쪽의 그림은 작년 수업 내용인데, 올해는 오른쪽 그림처럼 많은 것들이 만들어 졌다.



우리가 이러한 deep learning frameworks를 사용하는 이유는 다음과 같다.

1. 큰 computational graph를 만들기 쉽다.
2. computational graphs에서 gradient를 계산하기 쉽다.
3. GPU를 이용하여 효율적으로 실행시킬 수 있다.(cuDNN, cuBLAS, etc)

Numpy를 이용하여 연산하는 예시를 한번 살펴보자.

아래의 그림을 보면 numpy를 이용하여 곱셈, 덧셈, sum까지 연산은 쉽게 가능하다.

하지만 gradient를 구할 때 각각의 경우에서 직접 다 구해야 한다. 또한, GPU를 이용하여 실행시킬 수가 없다.

Numpy

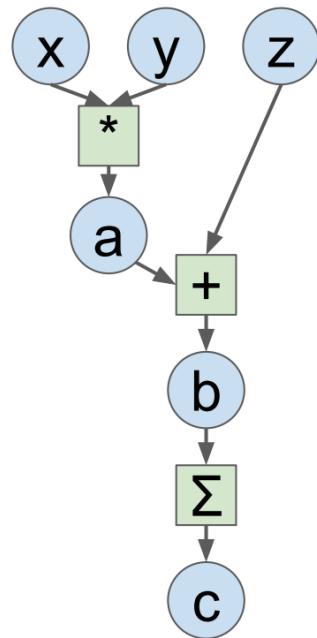
```
import numpy as np
np.random.seed(0)

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
```

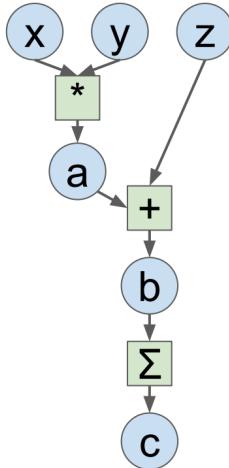


즉, 우리가 framework를 사용하여 얻고자 하는 것은 Numpy처럼 단순하게 forward 연산을 할 수 있으면서, gradient를 자동적으로 계산해주고, GPU를 사용할 수 있게끔 하는 것이다.

TensorFlow를 이용한 코드를 살펴보자.

우선, numpy와 유사하게 정말 간단한 연산을 통해 forward path 연산을 한다. 또 그 아래에 보면 gradient를 자동적으로 구할 수 있게끔 프로그래밍이 되어 있다.

Computational Graphs



Create forward computational graph

TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

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
```

또, 아래 두 사진처럼 CPU를 사용할 것인지, GPU를 사용할 것인지 고를 수 있게끔 프로그래밍이 되어 있다.

Tell
TensorFlow
to run on CPU

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

N, D = 3000, 4000

with tf.device('/cpu: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
```

Tell TensorFlow to run on GPU

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

N, D = 3000, 4000

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
```

pytorch도 마찬가지이다.

연산은 Numpy처럼 간단하게 할 수 있고, backward 함수가 있어서 쉽게 gradient를 연산할 수 있다. 또한 .cuda()를 뒤에 붙여서 GPU를 사용하게끔 할 수 있다.

```
import torch
from torch.autograd import Variable

N, D = 3, 4

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

```
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)
```

TensorFlow

코드 내부의 주석으로 설명

```
import numpy as np
import tensorflow as tf

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_w1 = tf.gradients(loss, [w1, w2])
#
# 여기까지는 graph 만드는 과정
#
#
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_w1], feed_dict=values)
#.run() 이 진짜 실행
    loss_val, grad_w1_val, grad_w2_val = out
    # 실행 후 결과 값 저장
```

```
import numpy as np
import tensorflow as tf

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_w1 = 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), }

#
# 이번에는 learning_rate를 정하고, 여러번 반복하면서 적합한 값을 찾아 나가는 과정이 아래에 있음
# Network Training 과정
learning_rate = 1e-5
for t in range(50):
    out = sess.run([loss, grad_w1, grad_w1], feed_dict=values)
    loss_val, grad_w1_val, grad_w1_val = out
    values[w1] -= learning_rate * grad_w1_val
    values[w2] -= learning_rate * grad_w1_val
# 그런데 여기서 보면 GPU와 CPU 간의 데이터 복사가 발생
# 이렇게 되면 속도가 느려지는 현상이 발생
# 이를 수정해야할 필요가 있음.

```

```

import numpy as np
import tensorflow as tf

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)))
# w1 과 w1를 placeholder에서 Variable 변수로 교체함
# Variable은 computational graph 내부에 존재하는 변수라고 생각하면 됨
# 이전에는 밖에서 값을 정의하고, 실행할 때 random함수를 이용하여 초기화 해주었다면
# 이제는 바로 computational graph 내부로 들어가기 때문에 random을 이용하여 정의
# 실제로 초기화하는 것이 아니라, 나중에 초기화한다는 것을 나타내는 표현이라고 함

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_w1 = 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)
# 이제 w1 과 w2가 graph 내부로 들어감.
# 이전에는 연산 후에 값을 copy해와서, 이를 업데이트 했다고 하면 이제는
# assign 함수를 이용하여 내부에서 변하게끔 설정할 수 있음.

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
# w1과 w2의 초기값 설정을 위해 한번 실행시킴
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),}
    for t in range(50):
        loss_val, = sess.run([loss], feed_dict=values)
# 여기서 학습을 시키기 위해 여러번 반복하여 training 진행
#

```

그런데 위의 코드대로 실행하면 정상적으로 training이 되지 않는다.

TensorFlow에서는 우리가 변화시키기 원하는 값들만을 변화시키게끔 프로그래밍이 되어 있다. 즉, 우리가 w1과 w2를 변화시켜주려면, 단지 assign시키는 것 뿐 아니라 graph에게 w1과 w2를 매 시행마다 update할 것이라고 알려주어야 하는 코드가 포함되어야 한다.

그런데, 그냥 값을 복사해서 대입한다면 위에서 우리가 마주한 CPU - GPU 데이터 교환이라는 문제가 발생할 것이다. 이를 해결할 수 있는 tricky한 방법이 아래에 소개되어 있다.

```
import numpy as np
import tensorflow as tf

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)
## 이 부분에 w1와 w2를 업데이트할 dummy node를 만들어 준다.
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H),}
    losses = []
    for t in range(50):
        loss_val, = sess.run([loss, updates], feed_dict = values)
# sess.run을 할 때, 위에서 만든 dummy node를 추가해주면 된다.
# 이렇게 되면 assign을 통해 graph내부에 포함되어 있는 변수 w1, w2가
# 내부 동작에 의해 값이 Update된다.
```

이를 조금 최적화한 방법으로 ,optimizer을 사용할 수 있다.

또한 내부 함수를 이용하여 common losses를 미리 정의할 수 있다.

```
import numpy as np
import tensorflow as tf

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)
# 이 부분이 losses를 미리 정의하는 부분.
```

```

# tf 내부 함수를 이용하여 정의 가능
optimizer = tf.train.GradientDescentOptimizer(1e-5)
updates = optimizer.minimize(loss)
# 위처럼 optimizer을 만들고, 인자는 learning_rate를 넣어주면 된다.
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    values = {x: np.random.randn(N, D),
              w1: np.random.randn(D, H), }
    for t in range(50):
        loss_val, = sess.run([loss, updates], feed_dict=values)
    # 당연히 여기에도 위에서 만든 update부분을 추가해 주어야 한다.

```

또, 편의를 위한 다양한 라이브러리가 존재하는데, 그중 초기화 과정에서 사용하는 xavier initializer이 있다.

```

import numpy as np
import tensorflow as tf

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)
# 위처럼 initialize가 가능하다.
# 자세히보면, w1, w2 (weight)가 정의되어 있지 않은데 이 방법을 이용하면
# 자동적으로 weight가 정의된다.
loss = tf.losses.mean_squared_error(y_pred,y)

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),
              w1: np.random.randn(D, H), }
    for t in range(50):
        loss_val, = sess.run([loss, updates], feed_dict=values)

```

Keras

또 다른 편한 API로는 Keras가 있다.

```

import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

```

```

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))
# 위처럼 초기 layers들을 정의
#
optimizer = SGD(lr=1e-0)
model.compile(loss = 'mean_squared_error' , optimizer = optimizer)
# model을 만드는데, loss function을 만든다고 보면 된다.
#
x = np.random.randn(N,D)
y = np.random.randn(N,D)
history = model.fit(x,y,nb_epoch=50, batch_size=N, verbose=0)
# 이렇게 한줄을 이용해서 train이 가능하다.

```

다양한 Frameworks

Keras (<https://keras.io/>)

TFLearn (<http://tflearn.org/>)

TensorLayer (<http://tensorlayer.readthedocs.io/en/latest/>)

tf.layers (https://www.tensorflow.org/api_docs/python/tf/layers)

TF-Slim (<https://github.com/tensorflow/models/tree/master/inception/inception/slim>)

tf.contrib.learn (https://www.tensorflow.org/get_started/tflearn)

Pretty Tensor (<https://github.com/google/prettytensor>)

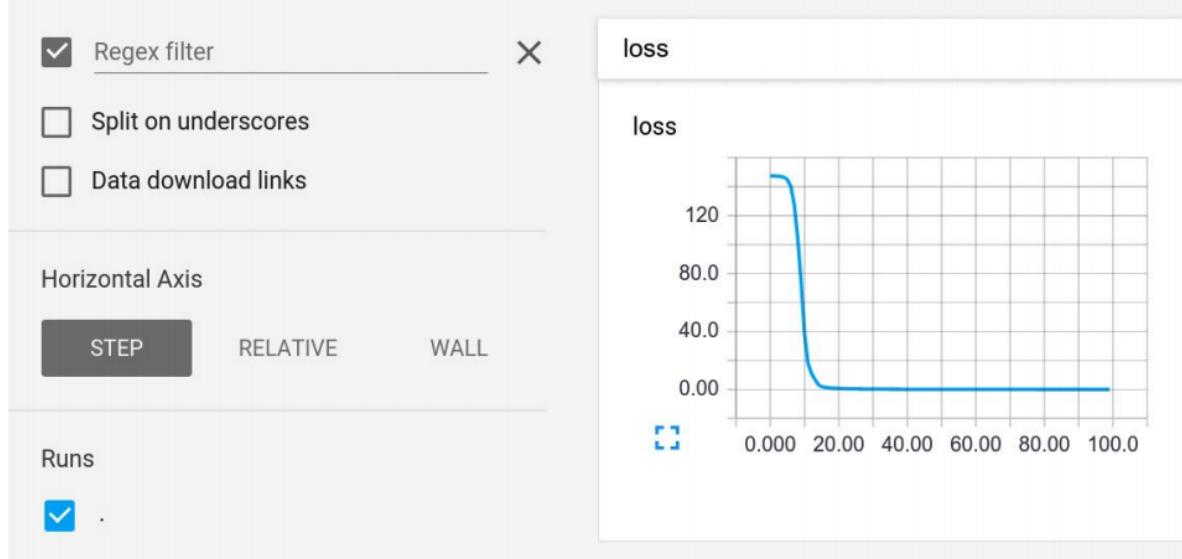
TensorFlow's pretrained model

- Keras
- TF-Slim

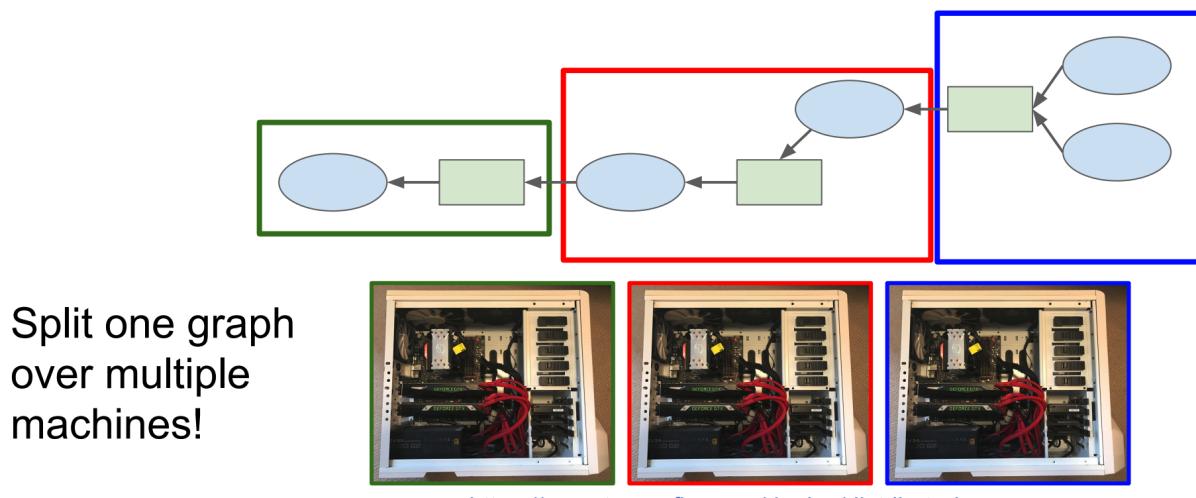
Tensorboard

그래프를 그려줌!

TensorBoard



또한, TensorFlow는 긴 graph를 여러 개로 쪼개서 각각의 machine으로 사용할 수 있다.



TensorFlow와 유사한 것으로, Theano라는 것도 있다.

```

import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)

```

PyTorch (Facebook)

Object

1. Tensor

- numpy에서 사용하는 것과 유사한 배열. 하지만 GPU를 사용한다.
- TensorFlow에서의 numpy array

2. Variable :

- computational graph의 노드. 데이터와 gradient를 저장한다.
- TensorFlow에서의 Tensor, Variable, Placeholder

3. Module

- Neural network의 layer. 상태나, weight들을 저장한다.
- TensorFlow에서의 tf.layers, TFSlim 등등..

Tensor

```
import torch

dtype = torch.FloatTensor
#
# dtype = torch.cuda.FloatTensor : GPU 사용하려면 이렇게 쓴다.
#

N, D_in, H, D_out = 64, 1000, 100, 10
x=torch.randn(N,D_in).type(dtype)
y=torch.randn(N,D_out).type(dtype)
w1=torch.randn(D_in,H).type(dtype)
w2=torch.randn(H,D_out).type(dtype)
# 데이터와 weights를 위해 random tensors를 만듦.
#

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

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h<0] = 0
    grad_w1 = x.t().mm(grad_h)
    # backward path 연산
    # gradient 연산
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
    # 위에서 구한 gradient를 이용하여 gradient descent 진행
```

Variable

x.data : Tensor

x.grad : Variable of gradient

x.grad.data : Tensor of gradient

```

import torch
from torch.autograd import Variable

N, D_in, H, D_out = 64, 1000, 100, 10
x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True)
w2 = Variable(torch.randn(H, D_out), requires_grad=True)
# Variable을 정의할 때, gradient를 계산할지 안할지 뒤에 플래그 형식으로 붙여준다.
#
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()
    # forward pass
    # Tensor version이랑 같아 보이지만 현재는 모두 Variable임.
    if w1.grad:
        w1.grad.data.zero_()
    if w2.grad:
        w2.grad.data.zero_()
    loss.backward()
    # backward pass
    # gradient 계산
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
    # gradient descent

```

New Autograd Functions

Pytorch에서는 자신만의 Autograd function을 정의할 수 있다.

```

class ReLU(torch.autograd.Function):
    def forward(self, x):
        self.save_for_backward(x)
        return x.clamp(min=0)

    def backward(self, grad_y):
        x, = self.saved_tensors
        grad_input = grad_y.clone()
        grad_input[x<0]= 0
        return grad_input

```

nn

Higher-level wrapper

Keras와 유사함

```

import torch
from torch.autograd import Variable

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

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
# 새로운 모델을 정의함
loss_fn = torch.nn.MSELoss(size_average = False)
# loss function도 정의
learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    # Forward pass

    model.zero_grad()
    loss.backward()
    # backward pass
    for param in model.parameters():
        param.data -= learning_rate * param.grad.data

```

Optimizer

Tensorflow에서 본 것처럼, pytorch에도 optimizer가 존재한다.

```

import torch
from torch.autograd import Variable

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

model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out))
loss_fn = torch.nn.MSELoss(size_average = False)
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
## optimizer 정의
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    # Forward pass

    optimizer.zero_grad()
    loss.backward()
    # backward pass

```

```
optimizer.step()
# 이렇게 한줄로 하여 모든 parameter들을 gradient를 이용해 업데이트 한다.
```

nn / Define new Modules

```
import torch
from torch.autograd import Variable

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
# define new model
# backward는 autograd가 알아서 처리해주기 때문에 따로 정의해주지 않아도 됨

N, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in), requires_grad=False)
y = Variable(torch.randn(N, D_out), requires_grad=False)

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

model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = criterion(y_pred, y)
# Forward pass
    optimizer.zero_grad()
    loss.backward()
# backward pass
    optimizer.step()
```

DataLoaders

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

```

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)

loader = DataLoader(TensorDataset(x, y), batch_size=8)
# DataLoader : minibatching, shuffling, multithreading 제공
#
model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for epoch in range(10):
    for x_batch, y_batch in loader:
        x_var, y_var = Variable(x), Variable(y)
        # 여기서 Variable을 만들어서 사용
        y_pred = model(x_var)
        loss = criterion(y_pred, y_var)

        optimizer.zero_grad()
        loss.backward()
    # backward pass
    optimizer.step()

```

Pretrained Models

pytorch에서는 여러 pretrained model을 제공한다.

<https://github.com/pytorch/vision>

```

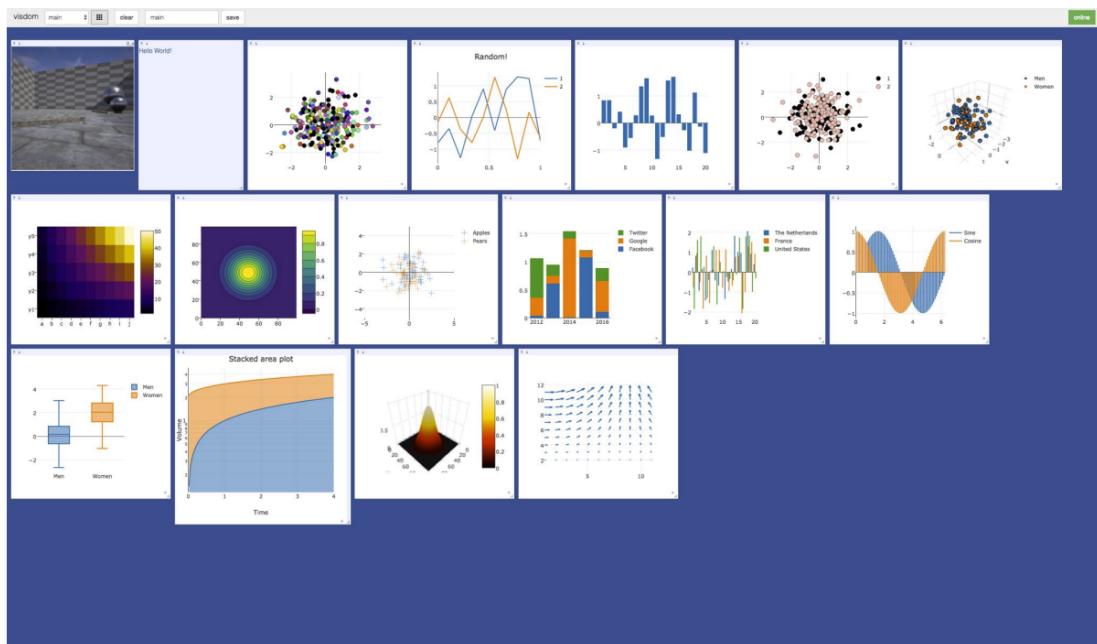
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)

```

Visdom

여러 그래프들을 그려주는 것도 존재함. TensorBoard와 유사



Torch

pytorch는 python 기반이지만, lua 기반으로 만들어진 torch도 있다.

비교해보면 아래와 같은 장단점이 있다.

Torch

- (-) Lua
- (-) No autograd
- (+) More stable
- (+) Lots of existing code
- (0) Fast

PyTorch

- (+) Python
- (+) Autograd
- (-) Newer, still changing
- (-) Less existing code
- (0) Fast

Static vs Dynamic Graphs

static graph는, 그래프를 한번 만들고 그것을 반복하는 것이다.(TensorFlow)

이에 비해 Dynamic Graph는 매번 새로운 그래프가 만들어지면서 training된다.(PyTorch)

TensorFlow: Build graph once, then run many times (**static**)

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

Build graph

Run each iteration

PyTorch: Each forward pass defines a new graph (**dynamic**)

```
import torch
from torch.autograd import Variable

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

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()

    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
    loss.backward()

    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data
```

New graph each iteration

Optimization

static graph의 경우, training을 시작하기 전, graph를 만드는 과정에서 최적화를 한번 하고 나면 graph가 후에는 변하지 않기 때문에 계속 최적화된 그래프를 이용할 수 있다.

Serialization

static graph의 경우 한번 그래프를 만들어 놓으면 변하지 않기 때문에, 만들어 두고 나중에 코드에 재 접근하지 않고도 재사용할 수 있다.

이에비해 Dynamic graph의 경우는 이것이 불가능하다.

Conditional

조건에 따라 그래프가 변하는 경우를 생각해보면, Dynamic Graph의 경우는 상당히 간단하게 코드를 작성할 수 있지만, Static은 불가능하다.

$$y = \begin{cases} w1 * x & \text{if } z > 0 \\ w2 * x & \text{otherwise} \end{cases}$$

PyTorch: Normal Python

```
N, D, H = 3, 4, 5

x = Variable(torch.randn(N, D))
w1 = Variable(torch.randn(D, H))
w2 = Variable(torch.randn(D, H))

z = 10
if z > 0:
    y = x.mm(w1)
else:
    y = x.mm(w2)
```

TensorFlow: Special TF control flow operator!

```
N, D, H = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(N, D))
z = tf.placeholder(tf.float32, shape=None)
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(D, H))

def f1(): return tf.matmul(x, w1)
def f2(): return tf.matmul(x, w2)
y = tf.cond(tf.less(z, 0), f1, f2)

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        z: 10,
        w1: np.random.randn(D, H),
        w2: np.random.randn(D, H),
    }
    y_val = sess.run(y, feed_dict=values)
```

Loops

위와 유사하게, 앞의 결과가 뒤의 결과에 영향을 끼치는 등의 작업이 있을 때에도 pytorch가 훨씬 편하다

$$y_t = (y_{t-1} + x_t) * w$$

PyTorch: Normal Python

```
T, N, D = 3, 4
y0 = Variable(torch.randn(D))
x = Variable(torch.randn(T, D))
w = Variable(torch.randn(D))

y = [y0]
for t in range(T):
    prev_y = y[-1]
    next_y = (prev_y + x[t]) * w
    y.append(next_y)
```

TensorFlow: Special TF control flow

```
T, N, D = 3, 4, 5
x = tf.placeholder(tf.float32, shape=(T, D))
y0 = tf.placeholder(tf.float32, shape=(D,))
w = tf.placeholder(tf.float32, shape=(D,))

def f(prev_y, cur_x):
    return (prev_y + cur_x) * w
y = tf.foldl(f, x, y0)

with tf.Session() as sess:
    values = {
        x: np.random.randn(T, D),
        y0: np.random.randn(D),
        w: np.random.randn(D),
    }
    y_val = sess.run(y, feed_dict=values)
```

Dynamic Graph Applications

- Recurrent network
- Recursive network
- Modular Network

Caffe

Caffe Overview

- Core written in C++
- Has Python and MATLAB bindings
- Good for training or finetuning feedforward classification models
- Often no need to write code!
- Not used as much in research anymore, still popular for deploying models

Caffe: Training / Finetuning

No need to write code!

1. Convert data (run a script)
2. Define net (edit prototxt)
3. Define solver (edit prototxt)
4. Train (with pretrained weights) (run a script)

Caffe Pros / Cons

- (+) Good for feedforward networks
- (+) Good for finetuning existing networks
- (+) Train models without writing any code!
- (+) Python interface is pretty useful!
- (+) Can deploy without Python
- (-) Need to write C++ / CUDA for new GPU layers
- (-) Not good for recurrent networks
- (-) Cumbersome for big networks (GoogLeNet, ResNet)

Caffe2

Caffe와 유사하게 Facebook에서 만든 것이 caffe2이다.

Caffe2 Overview

- Very new - released a week ago =)
- Static graphs, somewhat similar to TensorFlow
- Core written in C++
- Nice Python interface
- Can train model in Python, then serialize and deploy without Python
- Works on iOS / Android, etc

Facebook에서는 PyTorch, Caffe2 2개를 만들었다. PyTorch는 보다 더 연구용이라고 볼 수 있다. 이는 다른 환경(모바일 등)에 적용하기가 쉽지 않다. 그런 상황에서는 Caffe2를 사용하는 것이 더 낫다.

이에 비해 Google에서는 TensorFlow를 만들었다. 이는 여러 환경에서 모두 적합하도록 만들어져있다.