Lecture8) Deep Learning Software

CPU vs. GPU

CPU

- 코어의 수가 적지만, 각 코어의 연산 속도는 GPU보다 빠르다.
- 순차적인 태스크를 처리하는데에 유리하다.

GPU

- 코어의 수가 많지만, 각 코어의 연산 속도는 CPU보다 느리다.
- 병렬 작업에 유리하다.(행렬 dot product)
- GPU Programming
    - CUDA(NVIDIA, C like code), CUDA High-level API: cuDNN 등
    - OpenCL

→ CPU와 GPU 모두 각각 메모리(RAM)를 가지고 있다.

GPU Bottleneck Problem

- 문제: 데이터를 HDD에 넣어두고 불러와서 쓰는 경우, 데이터를 처리하는 속도보다, 읽어오는 속도가 느릴 수 있다.
- 해결방법
    - 모든 데이터를 RAM으로 읽어온 후에 연산한다.
    - HDD 대신 SSD를 사용한다.
    - 여러 CPU 스레드를 통해, 데이터를 RAM에 prefetch한다.

Deep Learning Frameworks

Framework 종류

- Caffe, Torch, Theano → Caffe2, **PyTorch, TensorFlow**
- Paddle, CNTK, MXNet

Framework를 사용하는 이유

- 큰 computational graph를 만들기 쉽다.
- Computational graph에서 gradient를 계산하기 용이하다.
- GPU에서 효율적으로 실행할 수 있다.

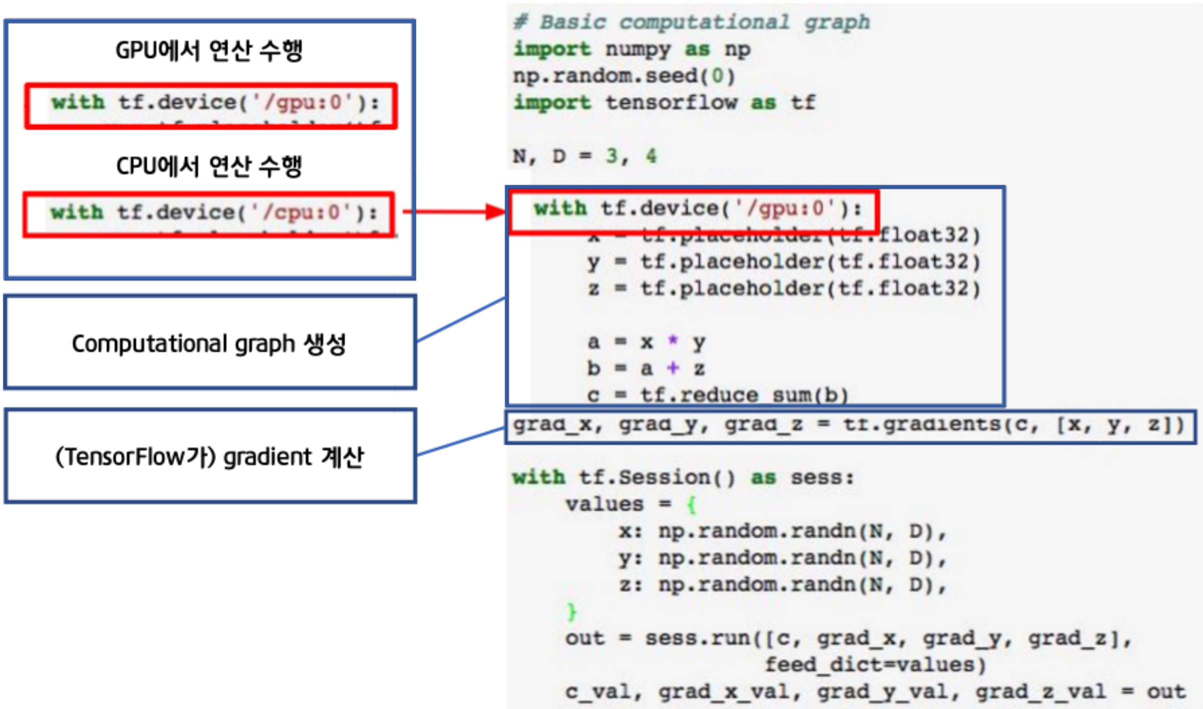
Numpy

- 코드

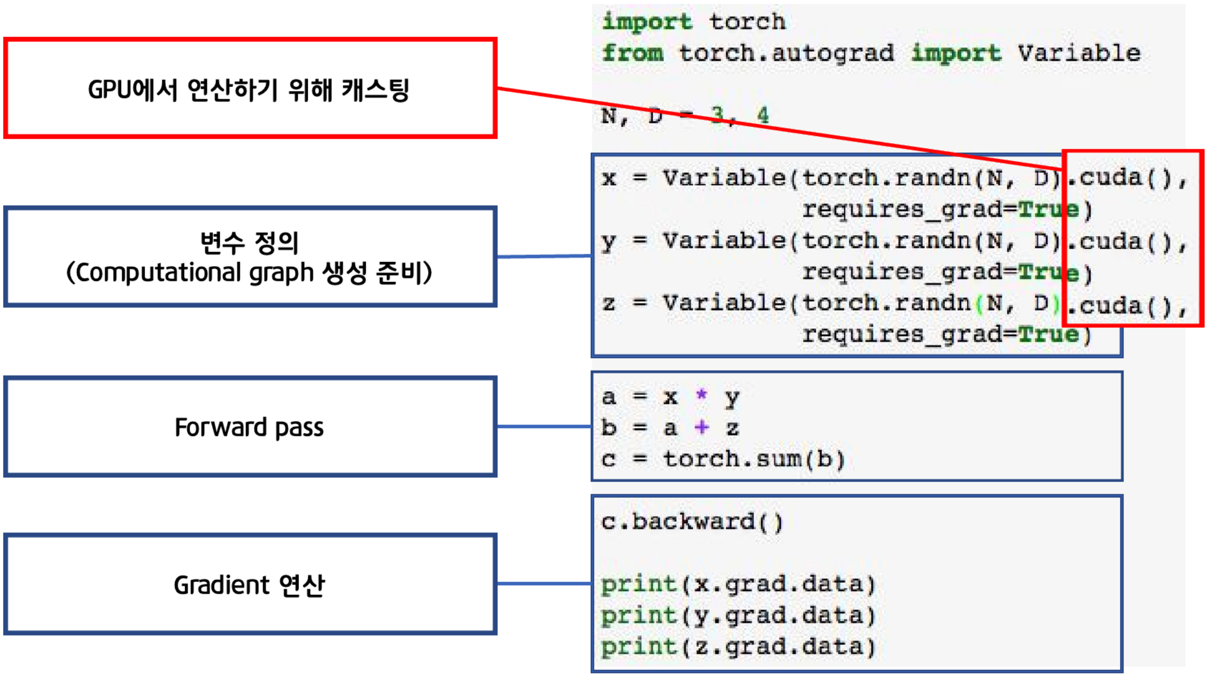
```
import numpy as np
np.random.seed(0)

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
```

- 단점
    - GPU에서 실행할 수 없다.
    - gradient를 직접 계산해야한다.

TensorFlow



Pytorch



→ TensorFlow, PyTorch는 자동으로 gradient를 계산해준다.

TensorFlow

Neural Net

```
'''V1'''
import numpy as np
import tensrotflow as tf

# 1. define computational graph
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D)) # create placeholders
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.placeholder(tf.float32, shape=(D, H))
w2 = tf.placeholder(tf.float32, shape=(H, D))

# forward pass / no computation. just build.
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)) # L2 dist(y, y_pred)

# loss of gradient / no computation. just build.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w1])

# 2. run the graph many times with feeding data
with tf.Session() as sess:
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                w1: np.random.randn(D, H),
                w2: np.random.randn(H, D),
                y: np.random.randn(N, D), }
    learning_rate = 1e-5
    # train the network
    for t in range(50):
        out = sess.run([loss, grad_w1, grad_w2], feed_dict=values) # run the graph
        loss_val, grad_w1_val, grad_w2_val = out # output: arrays

        values[w1] -= learning_rate * grad_w1_val # use gradient to update weights
        values[w2] -= learning_rate * grad_w2_val

'''
Problem: copying weights between CPU & GPU each step
'''
```

```
'''V2'''
import numpy as np
import tensrotflow as tf

# 1. define computational graph
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D)) # create placeholders
y = tf.placeholder(tf.float32, shape=(N, D))
''''''
w1 = tf.Variable(tf.float32, shape=(D, H)) # create Variables
w2 = tf.Variable(tf.float32, shape=(H, D))

# forward pass / no computation. just build.
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)) # L2 dist(y, y_pred)

# loss of gradient / no computation. just build.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w1])

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

# 2. run the graph many times with feeding data
with tf.Session() as sess:
    ''''''
    sess.run(tf.global_variables_initializer()) # run graph once to initialize w1, w2
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                y: np.random.randn(N, D), }

    ''''''
```

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


'''
Problem: loss not going down
'''
```

```
'''V3'''
import numpy as np
import tensrotflow as tf

# 1. define computational graph
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D)) # create placeholders
y = tf.placeholder(tf.float32, shape=(N, D))
w1 = tf.Variable(tf.float32, shape=(D, H)) # create Variables
w2 = tf.Variable(tf.float32, shape=(H, D))

# forward pass / no computation. just build.
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)) # L2 dist(y, y_pred)

# loss of gradient / no computation. just build.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w1])

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) # add dummy graph node

# 2. run the graph many times with feeding data
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) # run graph once to initialize w1, w2
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                y: np.random.randn(N, D), }

    ''''''
    # train the network
    for t in range(50):
        loss_val, _ = sess.run([loss, updates], feed_dict=values) # run the graph & compute dummy node(null return)
```

Optimization

```
'''original'''
import numpy as np
import tensrotflow as tf

# 1. define computational 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.Variable(tf.float32, shape=(D, H))
w2 = tf.Variable(tf.float32, shape=(H, D))

# forward pass / no computation. just build.
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))

# loss of gradient / no computation. just build.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w1])

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) # add dummy graph node

# 2. run the graph many times with feeding data
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) # run graph once to initialize w1, w2
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                y: np.random.randn(N, D), }

    ''''''
    # train the network
    for t in range(50):
        loss_val, _ = sess.run([loss, updates], feed_dict=values) # run the graph & compute dummy node(null return)
```

```
'''optimization'''
import numpy as np
import tensrotflow as tf

# 1. define computational 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.Variable(tf.float32, shape=(D, H))
w2 = tf.Variable(tf.float32, shape=(H, D))

# forward pass / no computation. just build.
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))

''''''
optimizer = tf.train.GradientDescentOprimizer(1e-5) # optimizer: compute grad & update W
updates = optimizer.minimize(loss)

# 2. run the graph many times with feeding data
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) # run graph once to initialize w1, w2
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                y: np.random.randn(N, D), }

    losses = []
    # train the network
    for t in range(50):
        loss_val, _ = sess.run([loss, updates], feed_dict=values) # run the graph & exec optimizer
```

Loss

```
'''Loss'''
import numpy as np
import tensrotflow as tf

# 1. define computational 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.Variable(tf.float32, shape=(D, H))
w2 = tf.Variable(tf.float32, shape=(H, D))

# forward pass / no computation. just build.
h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
''''''
loss = tf.losses.mean_squared_error(y_pred, y) # predefined loss

optimizer = tf.train.GradientDescentOprimizer(1e-3) # optimizer: compute grad & update W
updates = optimizer.minimize(loss)

# 2. run the graph many times with feeding data
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) # run graph once to initialize w1, w2
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                y: np.random.randn(N, D), }

    losses = []
    # train the network
    for t in range(50):
        loss_val, _ = sess.run([loss, updates], feed_dict=values) # run the graph & exec optimizer
```

Layers

```
'''Layers'''
import numpy as np
import tensrotflow as tf

# 1. define computational graph
N, D, H = 64, 1000, 100
x = tf.placeholder(tf.float32, shape=(N, D))
y = tf.placeholder(tf.float32, shape=(N, D))

''''''
# automatically set up W (tf.layers)
init = tf.contrib.layers.xavier_initializer() # 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)

# forward pass / no computation. just build.
h = tf.maximum(tf.matmul(x, w1), 0)
y_pred = tf.matmul(h, w2)
loss = tf.losses.mean_squared_error(y_pred, y) # predefined loss

optimizer = tf.train.GradientDescentOprimizer(1e-3) # optimizer: compute grad & update W
updates = optimizer.minimize(loss)

# 2. run the graph many times with feeding data
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) # run graph once to initialize w1, w2
    values = {  x: np.random.randn(N, D), # numpy arrays to fill placeholders
                y: np.random.randn(N, D), }

    losses = []
    # train the network
    for t in range(50):
        loss_val, _ = sess.run([loss, updates], feed_dict=values) # run the graph & exec optimizer
```

- Tensorflow's high-level wrapper
    - Keras, TFLearn, TensorLayer, tf.layers, TF-Flim, tf.contrib.learn, Pretty Tensor, Sonnet
- Pretrained models
    - TF-Slim, Keras
- Tensorboard: loss, status, 등을 볼 수 있는 logging tool(서버에서 돌아가고, 그래프 출력해준다.)

Keras

- High-level tensorflow wrapper

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD

N, D, H = 64, 1000, 100

# define model obj(sequence of layers)
model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=D))

# define optimizer obj
optimizer = SGD(lr=1e0)
# build model, specify loss func
model.compile(loss='mean_squared_error', optimizer=optimizer)

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

# train model
history = model.fit(x, y, nb_epoch=50, batch_size=N, verbose=0)
```

PyTorch

- 3 abstraction level
    - Tensor: GPU에서 실행되는 ndarray

      → TF: numpy array
    - Variable: computational graph 내의 노드(데이터와 gradient를 저장)

      → TF: Tensor, Variable, Placeholder
    - Module: NN layer(state, learnable weight 저장)

      → TF: High-level framework
- PyTorch는 매번 새로운 그래프를 만들고, TensorFlow는 명시적인 그래프를 만들어놓고 재활용한다.

```
import torch

# Tensor: like numpy, can run on GPU
# dtype = torch.FloatTensor     # CPU
dtype = torch.cuda.FloatTensor  # GPU

N, D_in, H, D_out = 64, 1000, 100, 10
# create random tensors(data, weight)
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_in).type(dtype)

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

    # backward pass: compute grad
    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)

    # gradient descent step on W
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Autograd

```
'''V1'''
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
''''''
# create Variables(nodes in graph)
# x.data: Tensor
# x.grad: Variable of gradients
# x.grad.data: Tensor of gradients
x = Variable(torch.randn(N, D_in), requires_grad=False) # requires_grad=False: no grad
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True) # requires_grad=True: want grad
w2 = Variable(torch.randn(H, D_in), requires_grad=True)

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

    # backward pass: compute grad
    if w1.grad: w1.grad.data.zero_()
    if w2.grad: w2.grad.data.zero_()
```

```
    loss.backward()

    # gradient descent step on W
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

```
'''V2'''
import torch

''''''
# define own autograd function
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

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

x = Variable(torch.randn(N, D_in), requires_grad=False) # requires_grad=False: no grad
y = Variable(torch.randn(N, D_out), requires_grad=False)
w1 = Variable(torch.randn(D_in, H), requires_grad=True) # requires_grad=True: want grad
w2 = Variable(torch.randn(H, D_in), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    ''''''
    # use own autograd runc in forward pass
    relu = ReLU()
    y_pred = relu(x.mm(w1)).mm(w2)
    loss = (y_pred - y).pow(2).sum()

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

    # gradient descent step on W
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

nn

- nn: higher-level wrapper

```
import torch
from torch.autograd import Variable

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

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

''''''
# define model as a sequence of layers
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-6
for t in range(500):
    # forward pass: feed data to model, pred to loss func
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    # backward pass: compute grad
    model.zero_grad()
    loss.backward()

    # gradient descent step on W
```

```
    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

optim

```
import torch
from torch.autograd import Variable

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

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

# define model as a sequence of layers
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)

''''''
# optimizer with update rules
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for t in range(500):
    # forward pass: feed data to model, pred to loss func
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    # backward pass: compute grad
    model.zero_grad()
    loss.backward()

    ''''''
    # gradient descent step on W
    optimizer.step()
```

Modules

- Module: neural net layer
    - input, output은 Variables이다.
    - 모듈은 모듈을 담거나 weights를 담을 수 있다.
    - autograd를 사용하여 모듈을 정의할 수 있다.
    - autograd는 backward를 정의할 필요가 없다.

```
import torch
from torch.autograd import Variable

''''''
# define model as a single module
class TwoLayerNet(torch.nn.Module):
    # initializer set up // two children modules
    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)
    # define forward pass // child modules, autograd ops on Variables
    # no need to backward
    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 = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# construct model
model = TwoLayerNet(D_in, H, D_out)
```

```
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# train model
for t in range(500):
    # forward pass: feed data to model, pred to loss func
    y_pred = model(x)
    loss = criterion(y_pred, y)

    # backward pass: compute grad
    model.zero_grad()
    loss.backward()

    # gradient descent step on W
    optimizer.step()
```

DataLoader

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

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

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

''''''
# DataLoader wraps Dataset, provides minibatching/shuffling/multithreading
# can load custom data(with own Dataset class)
loader = DataLoader(TensorDataset(x, y), batch_size=8)

# construct model
model = TwoLayerNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# load minibatch to loader over each iter
for epoch in range(10):
    for x_batch, y_batch in loader:
        # loader returns Tensor => need to wrap with Variable
        x_var, y_var = Variable(x), Variable(y)
        y_pred = model(x)
        loss = criterion(y_pred, y)

        # backward pass: compute grad
        model.zero_grad()
        loss.backward()

        # gradient descent step on W
        optimizer.step()
```

Pretrained Models

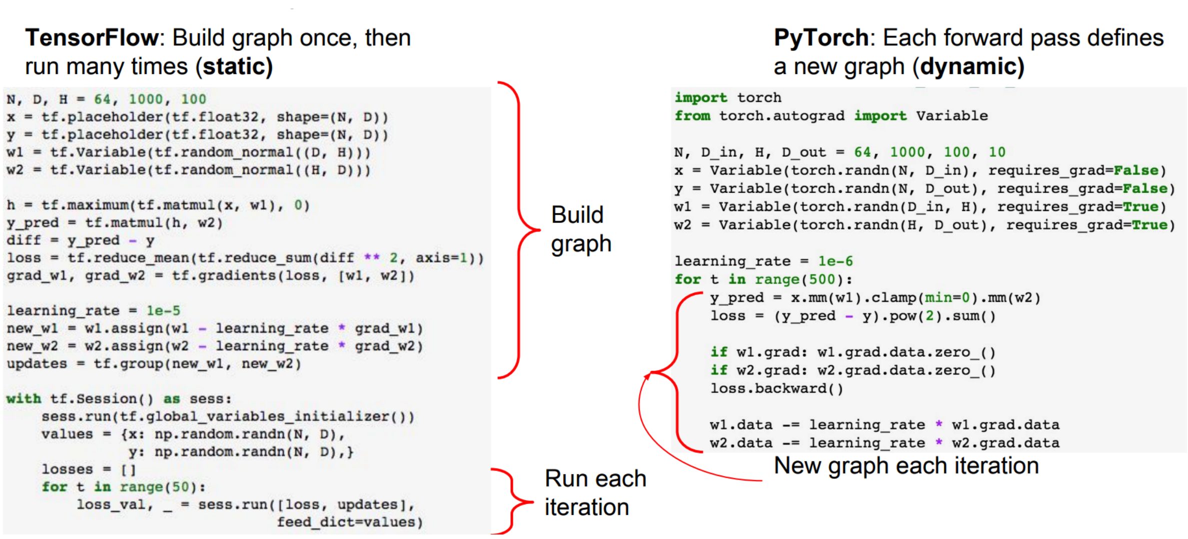
```
import torch
import torchvision

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

Torch vs. PyTorch

- Torch
    - Lua
    - No autograd
    - more stable

- Lots of existing code
- Fast
- PyTorch (추천)
  - Python
  - Autograd
  - Newer, still changing
  - ~~Less existing code~~
  - Fast

Static vs. Dynamic Graphs



- TensorFlow
  - 그래프를 한번 만들어두고, 여러번 재활용한다. → static graph
  - Optimization: 수행 전에 그래프 최적화 가능(각 연산을 융합할 수 있다.)
  - Serialization: 한번 그래프를 생성하면, disk 내에 직렬화가 가능하고, 빌드없이 코드를 돌릴 수 있다.
  - // TensorFlow Fold = TF의 Dynamic Graph
- PyTorch
  - 각 forward pass마다 그래프를 새로 생성한다. → dynamic graph
  - Non-serialization: 그래프 빌드와 수행이 얽혀있으므로, 코드를 계속 돌아다닌다.
  - Conditional: 조건에 따라 다른 코드를 수행할 수 있다.
  - Loops: K-fold 등을 사용하기 좋다.(각 반복마다 다른 데이터 등을 줄 수 있으므로)
- Dynamic Graph 적용
  - Recurrent network, Recursive network, Modular Networks(ex. img captioning, NLP)

Caffe2

- Static graph(TF와 유사)
- C++ core
- Python 인터페이스가 잘 갖춰져있다.

- Python으로 model을 train한 다음, python 없이 serialize, deploy 할 수 있다.
- iOS, Android 등에서 작동한다.

📌 [추천]
• **TensorFlow**
프로젝트에 사용하기 안정적이다. 커뮤니티가 잘 되어있다. High-level wrapper가 잘돼있다.
여러 machine에 하나의 그래프를 돌리기 위해서 적합하다.
• **PyTorch**
research에 적합하다.
• **TensorFlow, Caffe/Caff2:** Production deployment에 적합(ex. mobile)