ML/DL for Everyone with PYTORCH

Lecture 5: Linear regression in PyTorch way

Sung Kim <hunkim+ml@gmail.com> HKUST

Code: https://github.com/hunkim/PyTorchZeroToAll

Slides: http://bit.ly/PyTorchZeroAll

Videos: http://bit.ly/PyTorchVideo

Call for Comments

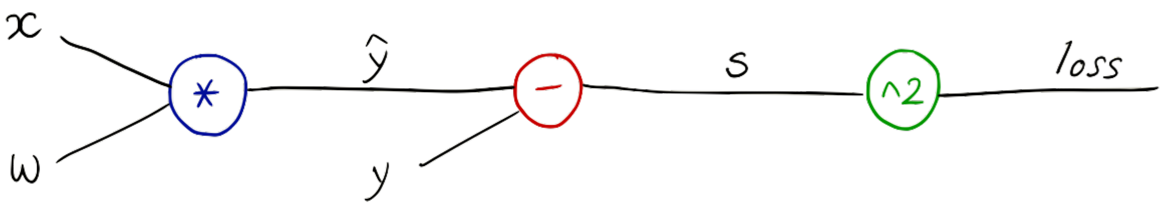
Please feel free to add comments directly on these slides.

Other slides: http://bit.ly/PyTorchZeroAll

Picture from http://www.tssablog.org/archives/3280

PyTorch forward/backward

```python
w = Variable(torch.Tensor([1.0]),  requires_grad=True)  # Any random value

# our model forward pass
def forward(x):
   return x * w

# Loss function
def loss(x, y):
   y_pred = forward(x)
   return (y_pred - y) * (y_pred - y)

# Training loop
for epoch in range(10):
   for x_val, y_val in zip(x_data, y_data):
       l = loss(x_val, y_val)
       l.backward()
       print("\tgrad: ", x_val, y_val, w.grad.data[0])
       w.data = w.data - 0.01 * w.grad.data

       # Manually zero the gradients after updating weights
       w.grad.data.zero_()

   print("progress:", epoch, l.data[0])
```



PyTorch Rhythm

1. Design your model using class with Variables

   모델 생성

2. Construct loss and optimizer
   (select from PyTorch API)

   목적 함수, 학습 방식 지정

3. Training cycle 반복 학습
   (forward, backward, update)

   전방계산 Gradient 계산 가중치 업데이트

Data definition (3x1)

```python
from torch import nn
import torch
from torch import tensor

x_data = tensor([[1.0], [2.0], [3.0]]) # torch.tensor -> torch
y_data = tensor([[2.0], [4.0], [6.0]])
```

❶ Model class in PyTorch way

상속 (호환성을 위해)

```python
class Model(nn.Module):
    def __init__(self):
```

생성자

```python
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
```

부모 클래스 생성자 호출

```python
        self.linear = torch.nn.Linear(1, 1)  # One in and one out
```

Learnable Parameter 초기화

(2, 1) $y = W_1x_1 + W_2x_2 + b$

(2, 2) $y_1 = W_{11}x_1 + W_{12}x_2 + b$ → 입력 출력 개수 설정

$y_2 = W_{21}x_1 + W_{22}x_2 + b$

```python
    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the construction as
        well as arbitrary operators on Variables.
        """
```

$\hookrightarrow \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + b$

```python
        y_pred = self.linear(x)
        return y_pred
```

전방 계산 (예측치 계산)

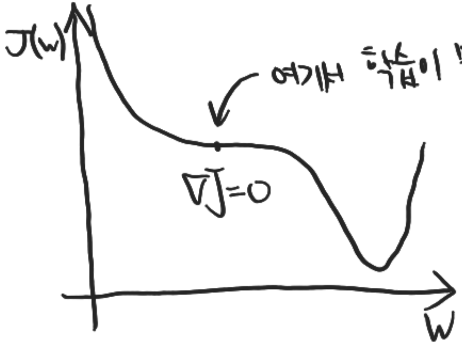
```python
# our model
model = Model()
```

② Construct loss and optimizer

↳ 하이퍼 파라미터

```
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

적응적 학습률 : 상황에 따라 lr이 달라짐



여기서 학습이 멈춰버림

↓

모멘텀 기술

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

https://github.com/jcjohnson/pytorch-examples

❸ Training: forward, loss, backward, step

```python
# Training loop
for epoch in range(500):
    # 1) Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # 2) Compute and print loss
    loss = criterion(y_pred, y_data)
    print(f'Epoch: {epoch} | Loss: {loss.item()} ')

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```python
for x_val, y_val in zip(x_data, y_data):
    ...
    w.data = w.data - 0.01 * w.grad.data
```

https://github.com/jcjohnson/pytorch-examples

Testing Model

```
# After training
hour_var = tensor([[4.0]])
y_pred = model(hour_var)
print("Prediction (after training)",  4, model(hour_var).item())
```

https://github.com/jcjohnson/pytorch-examples

Output

```
# Training loop
for epoch in range(500):
    # 1) Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # 2) Compute and print loss
    loss = criterion(y_pred, y_data)
    print(f'Epoch: {epoch} | Loss: {loss.item()} ')

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
# After training
hour_var = tensor([[4.0]])
y_pred = model(hour_var)
print("Prediction (after training)",  4, model(hour_var).item())
```

```
Epoch: 484 | Loss: 0.00010594200284685 94
Epoch: 485 | Loss: 0.00010441897757118568
Epoch: 486 | Loss: 0.00010291816579410806
Epoch: 487 | Loss: 0.00010143408871954307
Epoch: 488 | Loss: 9.99805488390848e-05
Epoch: 489 | Loss: 9.85444276011549e-05
Epoch: 490 | Loss: 9.713131294120103e-05
Epoch: 491 | Loss: 9.573066927259788e-05
Epoch: 492 | Loss: 9.435827087145299e-05
Epoch: 493 | Loss: 9.299971134169027e-05
Epoch: 494 | Loss: 9.166491508949548e-05
Epoch: 495 | Loss: 9.034640243044123e-05
Epoch: 496 | Loss: 8.905060531105846e-05
Epoch: 497 | Loss: 8.776798495091498e-05
Epoch: 498 | Loss: 8.65022448124364e-05
Epoch: 499 | Loss: 8.526088717617 1e-05
Prediction (after training) 4 7.98938512802124
```

https://github.com/jcjohnson/pytorch-examples

```python
import torch
from torch.autograd import Variable

x_data = Variable(torch.Tensor([[1.0], [2.0], [3.0]]))
y_data = Variable(torch.Tensor([[2.0], [4.0], [6.0]]))

class Model(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate two nn.Linear module
        """
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(1, 1)  # One in and one out

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        y_pred = self.linear(x)
        return y_pred

# our model
model = Model()


# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x_data)

    # Compute and print loss
    loss = criterion(y_pred, y_data)
    print(epoch, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()


# After training
hour_var = Variable(torch.Tensor([[4.0]]))
print("predict (after training)",  4, model.forward(hour_var).data[0][0])
```

1 Design your model using class

2 Construct loss and optimizer
(select from PyTorch API)

3 Training cycle
(forward, backward, update)

Training CIFAR10 Classifier

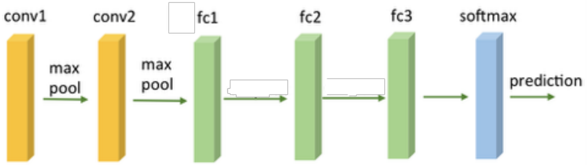
```python
########################################################################
# 1. Define a Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Copy the neural network from the Neural Networks section before and modify it to
# take 3-channel images (instead of 1-channel images as it was defined).

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

########################################################################
# 2. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Let's use a Classification Cross-Entropy loss and SGD with momentum
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

########################################################################
# 3. Train the network
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to the
# network and optimize
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Training CIFAR10 Classifier

```python
########################################################################
# 1. Define a Neural Network
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Copy the neural network from the Neural Networks section before and modify it to
# take 3-channel images (instead of 1-channel images as it was defined).

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

########################################################################
# 2. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Let's use a Classification Cross-Entropy loss and SGD with momentum
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

########################################################################
# 3. Train the network
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to the
# network and optimize
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.data[0]
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

1 Design your model using class



2 Construct loss and optimizer (select from PyTorch API)

3 Training cycle (forward, backward, update)

http://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Exercise 5-1: Try other optimizers and summarizes the properties and behaviors of the optimizers

- torch.optim.Adagrad
- torch.optim.Adam
- torch.optim.Adamax
- torch.optim.ASGD
- torch.optim.LBFGS
- torch.optim.RMSprop
- torch.optim.Rprop
- torch.optim.SGD

Ada: 적응적 학습률

http://pytorch.org/docs/master/optim.html#algorithms

Exercise 5-2: Read more PyTorch examples and smummarize them with

- http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

WHAT NEXT

Lecture 6: Logistic regression