

# ML/DL for Everyone with PYTORCH

## Lecture 5: Linear regression in PyTorch way

Sung Kim <[hunkim+ml@gmail.com](mailto: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>



# ML/DL for Everyone with PYTORCH

## Lecture 5: Linear Regression in PyTorch way

Sung Kim <[hunkim+ml@gmail.com](mailto:hunkim+ml@gmail.com)> HKUST

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

Slides: <http://bit.ly/PyTorchZeroAll>

Videos: <http://bit.ly/PyTorchVideo>



# PyTorch forward/backward

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

```
# our model forward pass
```

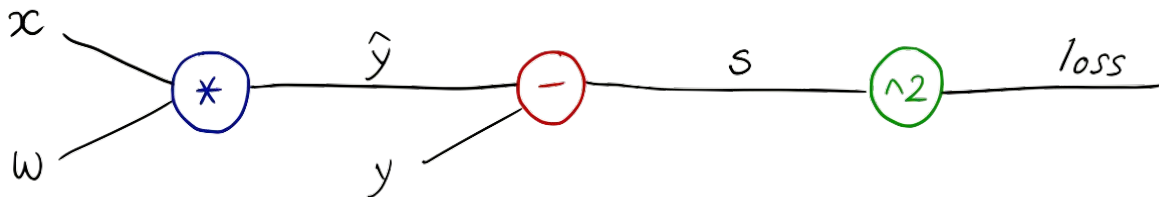
```
def forward(x):  
    return x * w
```

```
# 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
- 2** Construct loss and optimizer  
(select from PyTorch API)
- 3** Training cycle  
(forward, backward, update)

# Data definition (3x1)



```
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]]))
```

1

# Model class in PyTorch way



```
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()
```

## 2 Construct loss and optimizer



```
# 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)
```



3

# Training: forward, loss, backward, step



```
# 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()
```

3

# Training: forward, loss, backward, step



```
# 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*

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

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

# Testing 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])
```

# Output



```
# 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])
```

```
470 1.52139027704834e-05
471 1.4996051504567731e-05
472 1.4781335266889073e-05
473 1.4567947800969705e-05
474 1.4360077329911292e-05
475 1.4153701158647891e-05
476 1.3949686035630293e-05
477 1.3749523532169405e-05
478 1.3551662959798705e-05
479 1.3357152056414634e-05
480 1.3165942618797999e-05
481 1.2975904610357247e-05
482 1.2790364962711465e-05
483 1.2605956726474687e-05
484 1.2424526175891515e-05
485 1.2245835932844784e-05
486 1.2070459888491314e-05
487 1.1897350304934662e-05
488 1.1724299838533625e-05
489 1.155646714323666e-05
490 1.1392002306820359e-05
491 1.1226966307731345e-05
492 1.1066998922615312e-05
493 1.090722162189195e-05
494 1.0750130059022922e-05
495 1.0595314961392432e-05
496 1.0444626241223887e-05
497 1.029352642945014e-05
498 1.0146304703084752e-05
499 9.999960639106575e-06
predict (after training) 4 7.996364593505859
```



```
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])
```



```
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()
```

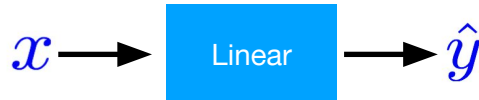
```
# After training
```

```
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

```
#####  
# 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')
```

airplane



automobile



bird



cat



deer



dog



frog



horse



ship



truck



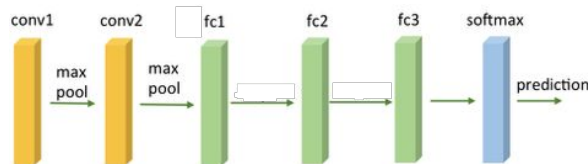
# Training CIFAR10 Classifier

```
#####  
# 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
```

1

Design your model using class



```
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)
```

2

Construct loss and optimizer  
(select from PyTorch API)

```
#####  
# 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
```

3

Training cycle  
(forward, backward, update)

```
    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')
```





# Exercise 5-1: Try other 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`

## Exercise 5-2: Read more PyTorch examples

- [http://pytorch.org/tutorials/beginner/pytorch\\_with\\_examples.html](http://pytorch.org/tutorials/beginner/pytorch_with_examples.html)



## Lecture 6: Logistic regression