Lecture 16: PyTorch Crash Course

In [1]:
```
import torch
import numpy as np
```

Tensor Initialization and Tensor Properties

See the full API at https://pytorch.org/docs/stable/tensors.html

Initializing a 1D tensor from a list

In [2]:
```
x=torch.tensor([1,2,3,4])
print(x)
```

```
tensor([1, 2, 3, 4])
```

Getting the shape of a tensor

In [3]:
```
x.shape
```

Out[3]:
```
torch.Size([4])
```

Reshaping a tensor

In [4]:
```
x.reshape(4,1)
```

Out[4]:
```
tensor([[1],
        [2],
        [3],
        [4]])
```

In [5]:
```
x.reshape(1,4)
```

Out[5]:
```
tensor([[1, 2, 3, 4]])
```

In [6]:
```
x.reshape(1,4).shape
```

Out[6]:
```
torch.Size([1, 4])
```

Getting the type of a tensor

In [7]:
```
type(x)
```

Out[7]:
```
torch.Tensor
```

In [8]:
```
x.dtype
```

Out[8]:
```
torch.int64
```

Setting the type of a tensor

In [9]:
```
x=torch.tensor([1,2,3,4]).type(torch.float)
x
```

Out[9]:
```
tensor([1., 2., 3., 4.])
```

In [10]:
```
x.dtype
```

Out[10]:
```
torch.float32
```

In [11]:
```
x=torch.FloatTensor([1,2,3,4])
x
```

Out[11]:
```
tensor([1., 2., 3., 4.])
```

In [12]:
```
x.dtype
```

Out[12]:
```
torch.float32
```

Initializing 2D Tensors

In [13]:
```
y = torch.FloatTensor([[1,2,3,4],[5,6,7,8]])
y
```

Out[13]:
```
tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])
```

Built in Tensor Constructors

In [14]:
```
torch.zeros(5)
```

Out[14]:
```
tensor([0., 0., 0., 0., 0.])
```

In [15]:
```
torch.ones(5)
```

Out[15]:
```
tensor([1., 1., 1., 1., 1.])
```

In [16]:
```
torch.rand(5)
```

Out[16]:
```
tensor([0.0511, 0.8955, 0.5597, 0.6825, 0.8168])
```

In [17]:
```
torch.eye(5)
```

Out[17]:
```
tensor([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]])
```

Constructing tensors for Numpy arrays

In [18]:
```
torch.FloatTensor(np.random.randn(5,3))
```

Out[18]:
```
tensor([[-1.0754, -1.0123, -0.7209],
        [-0.6453, -1.7376, -1.3521],
        [ 1.1395,  0.1085,  0.8397],
        [ 0.5225, -2.1468, -1.3777],
        [ 0.1642,  0.8596,  0.2979]])
```

Linear Algebra

Create data

In [19]:
```
x=torch.FloatTensor([[1,2,3,4]])
z=torch.FloatTensor([[1,2,3,4],[5,6,7,8]])
print("x=",x)
print("z=",z)
```

```
x= tensor([[1., 2., 3., 4.]])
z= tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])
```

Multiplication by a sclar

In [20]:
```
5*x
```

Out[20]:
```
tensor([[ 5., 10., 15., 20.]])
```

Addition of a scalar

In [21]:
```
x+5
```

Out[21]:
```
tensor([[6., 7., 8., 9.]])
```

Addition of two tensors of the same shape

In [22]:
```
x+x
```

Out[22]:
```
tensor([[2., 4., 6., 8.]])
```

Transpose of a tensor

In [23]:
```
x.T
```

Out[23]:
```
tensor([[1.],
        [2.],
        [3.],
        [4.]])
```

Matrix Multiplication

In [24]:
```
x@x.T
```

Out[24]:
```
tensor([[30.]])
```

In [25]:
```
x.T@x
```

Out[25]:
```
tensor([[ 1.,  2.,  3.,  4.],
        [ 2.,  4.,  6.,  8.],
        [ 3.,  6.,  9., 12.],
        [ 4.,  8., 12., 16.]])
```

In [26]:
```
x@z.T
```

Out[26]:
```
tensor([[30., 70.]])
```

Elementwise multiplication (Hadamard product)

In [27]:
```
x*x
```

Out[27]:
```
tensor([[ 1.,  4.,  9., 16.]])
```

Broadcasting

Create data

In [28]:
```
x=torch.FloatTensor([[1,2,3,4]])
y=torch.FloatTensor([[-1,-2]]).T
z=torch.FloatTensor([[1,2,3,4],[5,6,7,8]])
print("x=",x,"\n")
print("y=",y,"\n")
print("z=",z,"\n")
```

```
x= tensor([[1., 2., 3., 4.]])

y= tensor([[-1.],
        [-2.]])

z= tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])
```

Addition with broadcasting

In [29]:
```
print("x=",x,"\n")
print("y=",y,"\n")
a = x+y
print("x+y=",a)
```

```
x= tensor([[1., 2., 3., 4.]])

y= tensor([[-1.],
        [-2.]])

x+y= tensor([[ 0.,  1.,  2.,  3.],
        [-1.,  0.,  1.,  2.]])
```

In [30]:
```
print("x=",x,"\n")
print("z=",z,"\n")
a = x+z
print("x+z=",a)
```

```
x= tensor([[1., 2., 3., 4.]])

z= tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])

x+z= tensor([[ 2.,  4.,  6.,  8.],
        [ 6.,  8., 10., 12.]])
```

In [31]:
```
print("y=",y,"\n")
print("z=",z,"\n")
a = y+z
print("y+z=",a)
```

```
y= tensor([[-1.],
        [-2.]])

z= tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])

y+z= tensor([[0., 1., 2., 3.],
        [3., 4., 5., 6.]])
```

Elementwise multiplication with broadcasting

In [32]:
```
x*y
```

Out[32]:
```
tensor([[-1., -2., -3., -4.],
        [-2., -4., -6., -8.]])
```

In [33]:
```
x*z
```

Out[33]:
```
tensor([[ 1.,  4.,  9., 16.],
        [ 5., 12., 21., 32.]])
```

In [34]:
```
y*z
```

Out[34]:
```
tensor([[ -1.,  -2.,  -3.,  -4.],
        [-10., -12., -14., -16.]])
```

Elementwise Functions

In [35]:
```
x=torch.FloatTensor([[1,2,-3,-4]])
x
```

Out[35]:
```
tensor([[ 1.,  2., -3., -4.]])
```

In [36]:
```
torch.pow(x,2)
```

Out[36]:
```
tensor([[ 1.,  4.,  9., 16.]])
```

In [37]:
```
x**2
```

Out[37]:
```
tensor([[ 1.,  4.,  9., 16.]])
```

In [38]:
```
torch.exp(x)
```

Out[38]:
```
tensor([[2.7183, 7.3891, 0.0498, 0.0183]])
```

In [39]:
```
torch.log(x)
```

Out[39]:
```
tensor([[0.0000, 0.6931,    nan,    nan]])
```

In [40]:
```
torch.sin(x)
```

Out[40]:
```
tensor([[ 0.8415,  0.9093, -0.1411,  0.7568]])
```

In [41]:
```
torch.abs(x)
```

Out[41]:
```
tensor([[1., 2., 3., 4.]])
```

Matrix Functions

In [42]:
```
z = torch.FloatTensor([[2,1],[1,3]])
z
```

Out[42]:
```
tensor([[2., 1.],
        [1., 3.]])
```

In [43]:
```
torch.det(z)
```

Out[43]:
```
tensor(5.)
```

In [44]:
```
torch.inverse(z)
```

Out[44]:
```
tensor([[ 0.6000, -0.2000],
        [-0.2000,  0.4000]])
```

In [45]:
```
torch.trace(z)
```

Out[45]:
```
tensor(5.)
```

Aggregation Functions

Create data

In [46]:
```
x=torch.FloatTensor([[1,2,3,4]])
z=torch.FloatTensor([[1,2,3,4],[5,6,7,8]])
print("x=",x,"\n")
print("z=",z)
```

```
x= tensor([[1., 2., 3., 4.]])

z= tensor([[1., 2., 3., 4.],
        [5., 6., 7., 8.]])
```

Summing over all tensor elements

In [47]:
```
x.sum()
```
Out[47]:
```
tensor(10.)
```

In [48]:
```
z.sum()
```
Out[48]:
```
tensor(36.)
```

Summing over a specific axis

In [49]:
```
z.sum(axis=0)
```
Out[49]:
```
tensor([ 6.,  8., 10., 12.])
```

In [50]:
```
z.sum(axis=1)
```
Out[50]:
```
tensor([10., 26.])
```

In [51]:
```
z.sum(axis=1,keepdims=True)
```
Out[51]:
```
tensor([[10.],
        [26.]])
```

Mean and Product

In [52]:
```
x.mean()
```
Out[52]:
```
tensor(2.5000)
```

In [53]:
```
x.prod()
```
Out[53]:
```
tensor(24.)
```

Min and Max

In [54]:
```
x.min()
```
Out[54]:
```
tensor(1.)
```

In [55]:
```
x.max()
```
Out[55]:
```
tensor(4.)
```

In [56]:
```
z.min()
```
Out[56]:
```
tensor(1.)
```

In [57]:
```
v,ind = z.min(axis=0)
print(v)
print(ind)
```

```
tensor([1., 2., 3., 4.])
tensor([0, 0, 0, 0])
```

In [58]:
```
v,ind = z.min(axis=1)
print(v)
print(ind)
```

```
tensor([1., 5.])
tensor([0, 0])
```

Automatic Differentiation

Initializing tensors

In [59]:
```
x=torch.FloatTensor([[1,2,3,4],[-1,-2,-3,-4]])
y=torch.tensor([[1.0],[0.0]])
print("x=",x,"\n")
print("y=",y,"\n")
```

```
x= tensor([[ 1.,  2.,  3.,  4.],
        [-1., -2., -3., -4.]])

y= tensor([[1.],
        [0.]])
```

Initializing tensors with gradient tracking

In [60]:
```
w=torch.tensor([[-1.0],[-2.0],[2.0],[1.0]],requires_grad=True)
b=torch.tensor(-1.0,requires_grad=True)
print("w=",w,"\n")
print("b=",b,"\n")
```

```
w= tensor([[-1.],
        [-2.],
        [ 2.],
        [ 1.]], requires_grad=True)

b= tensor(-1., requires_grad=True)
```

Checking if gradient tracking is enabled

In [61]:
```
x.requires_grad
```

Out[61]:
```
False
```

In [62]:
```
w.requires_grad
```

Out[62]:
```
True
```

In [63]:
```
b.requires_grad
```

Out[63]:
```
True
```

Performing Operations

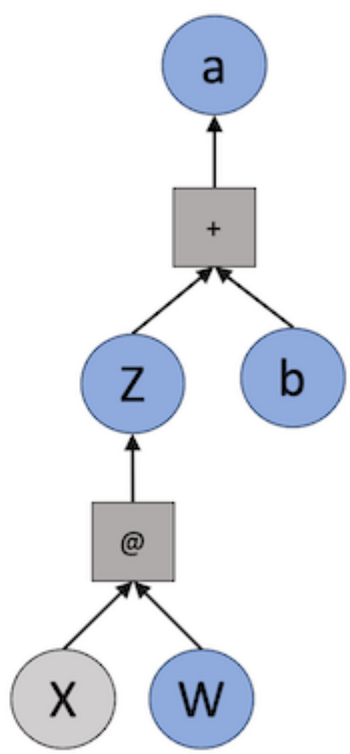
In [64]:
```
a=x[0,:]@w +b
a
```

Out[64]:
```
tensor([4.], grad_fn=<AddBackward0>)
```

In [65]:
```
a.requires_grad
```

Out[65]:
```
True
```

Computation Graph



Getting gradient values

In [66]:
```
a.backward()
print("grad_w:",w.grad,"\n")
print("grad_b:",b.grad)
```

```
grad_w: tensor([[1.],
        [2.],
        [3.],
        [4.]])

grad_b: tensor(1.)
```

Gradient accumulation and clearing

In [67]:
```python
a=x[0,:]@w +b
a.backward()
print("grad_w:",w.grad,"grad_b:",b.grad)
```

```
grad_w: tensor([[2.],
        [4.],
        [6.],
        [8.]]) grad_b: tensor(2.)
```

In [68]:
```python
w.grad=None
b.grad=None
a=x[0,:]@w +b
a.backward()
print("grad_w:",w.grad,"grad_b:",b.grad)
```

```
grad_w: tensor([[1.],
        [2.],
        [3.],
        [4.]]) grad_b: tensor(1.)
```

Hiding operations from gradient tracking

In [69]:
```python
w.grad=None
b.grad=None
with torch.no_grad():
    a=x[0,:]@w +b
a
```

Out[69]:
```
tensor([4.])
```

Example: OLS gradient

In [70]:
```python
def mse(y,yhat):
    return torch.mean((y-yhat)**2)

w.grad=None
b.grad=None

yhat = x@w+b
loss = mse(y,yhat)
loss.backward()
print("grad_w:",w.grad,"\n")
print("grad_b:",b.grad)
```

```
grad_w: tensor([[ 9.],
        [18.],
        [27.],
        [36.]])

grad_b: tensor(-3.)
```

Neural Network Modules

See the full API at https://pytorch.org/docs/stable/nn.html

Generate data

In [71]:
```
X = torch.randn(1000,2)
Y = -3*X[:,[0]] + 2*X[:,[1]]**2 + 3
Y[:5,:]
```

Out[71]:
```
tensor([[ 1.3281],
        [ 4.5852],
        [ 1.1751],
        [ 5.3190],
        [12.0424]])
```

Defining a linear layer from scratch

In [72]:
```
import torch.nn as nn

class linear(nn.Module):
    def __init__(self,d,k):
        super(linear, self).__init__()
        self.w = nn.Parameter(torch.rand(d,k))
        self.b = nn.Parameter(torch.rand(1,k))

    def forward(self,x):
        return x@self.w + self.b
```

Instatiating the layer and computing output

In [73]:
```
model = linear(2,1)
model.forward(X[:10,:])
```

Out[73]:
```
tensor([[ 0.7800],
        [ 0.3021],
        [ 0.6498],
        [ 0.1447],
        [-0.1138],
        [-0.2512],
        [ 0.1483],
        [ 0.1373],
        [ 0.2449],
        [ 0.6503]], grad_fn=<AddBackward0>)
```

Computing a loss on the output

In [74]:
```python
Yhat = model.forward(X)
loss = mse(Yhat,Y)
loss
```

Out[74]:
```
tensor(39.8705, grad_fn=<MeanBackward0>)
```

Manually setting parameters

In [75]:
```python
model.w.data = torch.zeros(2,1)
model.b.data = torch.zeros(1,1)
print("w:",model.w)
print("b:",model.b)
```

```
w: Parameter containing:
tensor([[0.],
        [0.]], requires_grad=True)
b: Parameter containing:
tensor([[0.]], requires_grad=True)
```

In [76]:
```python
model.forward(X[:10,:])
```

Out[76]:
```
tensor([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]], grad_fn=<AddBackward0>)
```

Defining a relu MLP using built-in layers

In [77]:
```python
class relu_mlp(nn.Module):
    def __init__(self,d,k):
        super(relu_mlp, self).__init__()
        self.l1   = nn.Linear(d,k)
        self.relu = nn.ReLU()
        self.out  = nn.Linear(k,1)

    def forward(self,x):
        return self.out(self.relu(self.l1(x)))
```

Inspecting the model

In [78]:
```python
model = relu_mlp(2,3)
print("l1:", model.l1)
print("relu:",model.relu)
print("out:",model.out)
```

```
l1: Linear(in_features=2, out_features=3, bias=True)
relu: ReLU()
out: Linear(in_features=3, out_features=1, bias=True)
```

Inspecting the parameters

In [79]:
```
print("l1 weight", model.l1.weight,"\n")
print("l1 bias", model.l1.bias)
```

```
l1 weight Parameter containing:
tensor([[ 0.0766, -0.4320],
        [-0.6615, -0.0574],
        [-0.4294,  0.2210]], requires_grad=True)

l1 bias Parameter containing:
tensor([-0.4012, -0.5149, -0.4955], requires_grad=True)
```

Computing output

In [80]:
```
model.forward(X[:10,:])
```

Out[80]:
```
tensor([[0.4863],
        [0.4863],
        [0.4863],
        [0.4863],
        [0.4823],
        [0.6364],
        [0.4834],
        [0.6093],
        [0.4862],
        [0.4862]], grad_fn=<AddmmBackward0>)
```

Computing a loss on the output

In [81]:
```
Yhat = model.forward(X)
loss = mse(Yhat,Y)
loss
```

Out[81]:
```
tensor(36.7757, grad_fn=<MeanBackward0>)
```

Manipulating the parameters built-in layers

In [82]:
```
with torch.no_grad():
    model.l1.weight.data = torch.zeros_like(model.l1.weight.data)
    model.l1.bias.data   = torch.zeros_like(model.l1.bias.data)
    model.out.weight.data = torch.zeros_like(model.out.weight.data)
    model.out.bias.data   = torch.zeros_like(model.out.bias.data)

print("l1 weight:", model.l1.weight,"\n")
print("l1 bias:", model.l1.bias,"\n")
print("out weight:", model.out.weight,"\n")
print("out bias:", model.out.bias,"\n")
```

```
l1 weight: Parameter containing:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]], requires_grad=True)

l1 bias: Parameter containing:
tensor([0., 0., 0.], requires_grad=True)

out weight: Parameter containing:
tensor([[0., 0., 0.]], requires_grad=True)

out bias: Parameter containing:
tensor([0.], requires_grad=True)
```

In [83]:
```
model.forward(X[:10,:])
```

Out[83]:
```
tensor([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]], grad_fn=<AddmmBackward0>)
```

Optimization

Defining a basic fit function

In [84]:
```
def fit(model, lr, max_iter):
    optimizer = torch.optim.SGD(model.parameters(), lr=lr)
    losses=[]
    for i in range(max_iter):
        Yhat = model.forward(X)
        loss = mse(Yhat,Y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.detach().numpy().item())
        if(i%10==0):
            print("%d %.2f"%(i, losses[-1]))

    return(losses)
```

Instantiating models

In [85]:
```
models = {"Linear":linear(2,1),"RELU MLP":relu_mlp(2,20)}
```

Fitting models

In [86]:

```
losses = {}
for m in models:
    print("Learning model: %s"%m)
    losses[m] = fit(models[m], 0.05, 100)
    print()
```

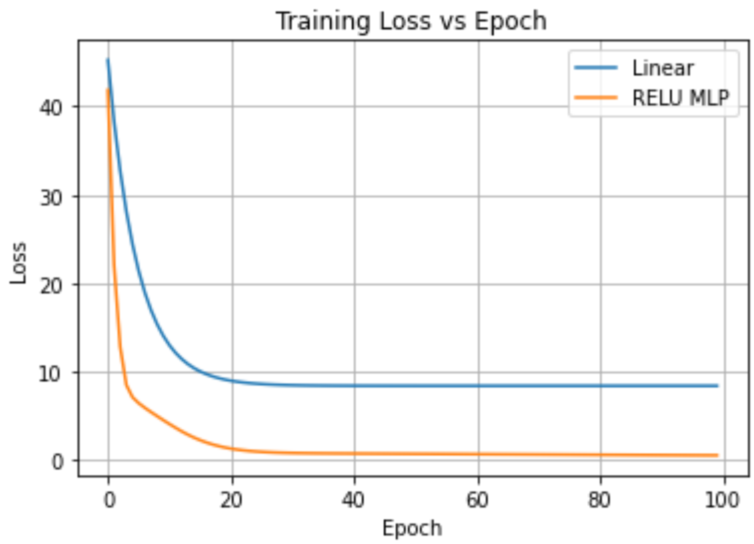
```
Learning model: Linear
0 45.25
10 13.00
20 8.98
30 8.48
40 8.41
50 8.41
60 8.40
70 8.40
80 8.40
90 8.40

Learning model: RELU MLP
0 41.85
10 4.09
20 1.30
30 0.81
40 0.75
50 0.71
60 0.68
70 0.64
80 0.61
90 0.57
```

Plotting losses

In [87]:

```
import matplotlib.pyplot as plt
plt.figure()
for m in models:
    plt.plot(losses[m])
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(list(models.keys()))
plt.title("Training Loss vs Epoch")
plt.grid(True)
```



Computing error of fit models with eval and no_grad

In [88]:

```
Xte = torch.randn(1000,2)
Yte = -3*Xte[:,[0]] + 2*Xte[:,[1]]**2 + 3

te_err={}
for m in models:
    models[m].eval()
    with torch.no_grad():
        te_err[m]=mse(Yte,models[m].forward(Xte))
        print("%s test error: %.2f"%(m,te_err[m]))
```

```
Linear test error: 9.36
RELU MLP test error: 0.73
```

In [ ]: