STATS 507
Data Analysis in Python

Week 10: PyTorch (Part 2)

Recap: What is PyTorch?

It’s a Python-based scientific computing package targeted at two sets of audiences:

1. A replacement for NumPy to use the power of GPUs
2. a deep learning research platform that provides maximum flexibility and speed

Recap: Resizing

Resizing: If you want to resize/reshape tensor, you can use `torch.view`:

```
[14] x = torch.randn(4, 4)
     y = x.view(16)
     z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
     print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

If you have a one element tensor, use `.item()` to get the value as a Python number

```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([-0.8768])
-0.8767884373664856
```

Recap: GPU speedup on Google colab

```
[1]  import torch
     import time
```

```
[2]  x_cpu = torch.randn(120000, 10000)
     y_cpu = torch.randn(10000, 1)
```

```
[3]  x_gpu = x_cpu.cuda()
     y_gpu = y_cpu.cuda()
```

```
[4]  start = time.time()
     x_cpu.mm(y_cpu)
     print(time.time() - start)

     0.39958739280700684
```

```
     start = time.time()
     x_gpu.mm(y_gpu)
     print(time.time() - start)

     0.004973173141479492
```

Matrix-vector multiplication on the GPU is nearly 100x faster!

Recap: Automatic Differentiation

```
[2]  import torch
```

Create a tensor and set requires_grad=True to track computation with it

```
[3]  x = torch.ones(2, 2, requires_grad=True)
     print(x)
```

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

Do an operation of tensor:

```
[4]  y = x + 2
     print(y)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

`y` was created as a result of an operation, so it has a `grad_fn`.

```
[5]  print(y.grad_fn)
```

```
<AddBackward0 object at 0x7f4f72142898>
```

Do more operations on y

```
[6]  z = y * y * 3
     out = z.mean()

     print(z, out)
```

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>) tensor(27.,
```

Recap: Automatic Differentiation

Let's backprop now Because `out` contains a single scalar, `out.backward()` is equivalent to `out.backward(torch.tensor(1))`.

```
[8]  out.backward()
```

print gradients d(out)/dx

```
[9]  print(x.grad)
```

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

You should have got a matrix of `4.5`. Let's call the `out` *Tensor* "o". We have that $o = \frac{1}{4}\sum_i z_i$, $z_i = 3(x_i + 2)^2$ and $z_i\big|_{x_i=1} = 27$. Therefore, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$, hence $\frac{\partial o}{\partial x_i}\big|_{x_i=1} = \frac{9}{2} = 4.5$.

You can do many crazy things with autograd!

Gradient Descent

&

Stochastic Gradient Descent (SGD)

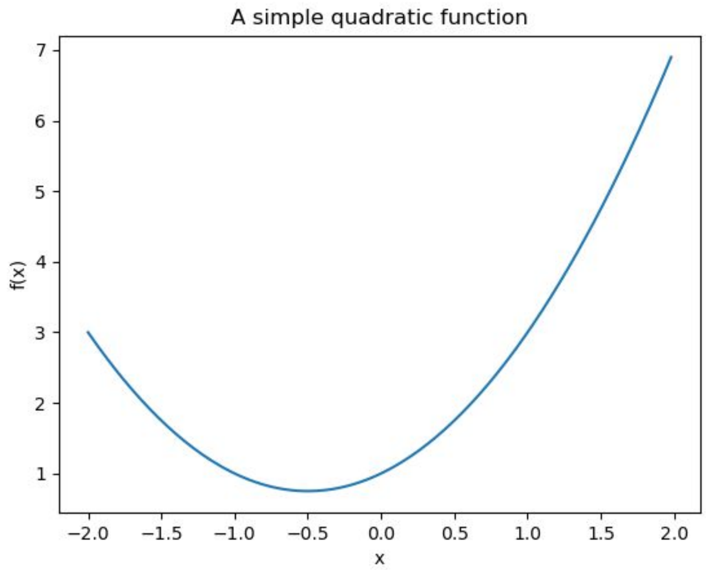
Numerical Optimization

Suppose we want to minimize

$$f(x) = x^2 + x + 1$$

Easy!

$$\nabla f(x) = 0$$
$$\implies 2x + 1 = 0$$
$$\implies x = -1/2$$



Numerical Optimization

Now consider

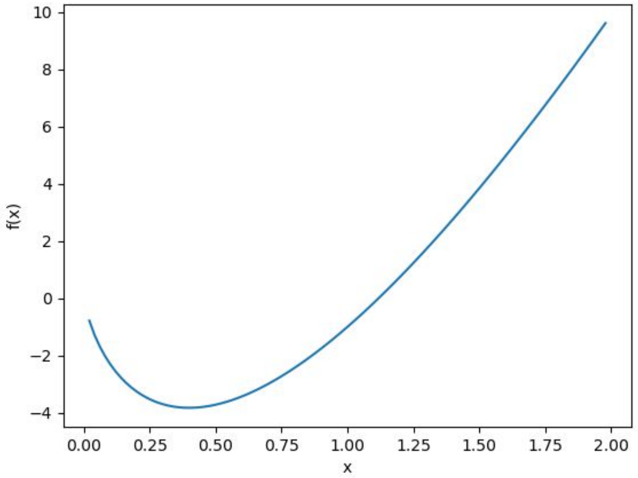
$$f(x) = -x^2 + 10x\log(x)$$

We know

$$\nabla f(x) = -2x + 10\log(x) + 10$$

But what value of x sets

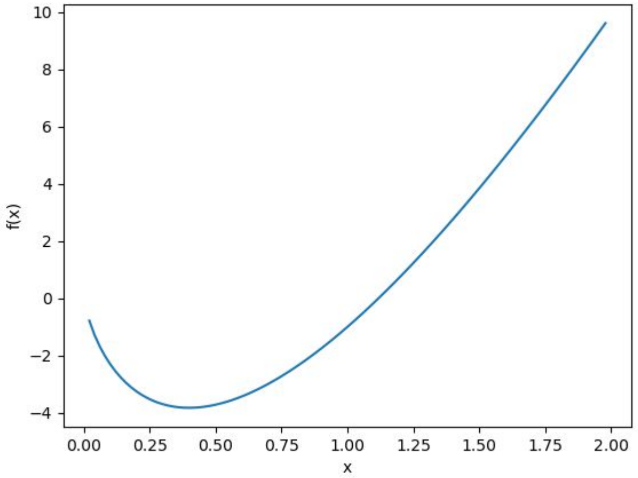
$$\nabla f(x) = 0 \quad ?$$



Iterative gradient-based solvers

In [82]:
```python
import torch

def f(x):
    return -x**2 + 10 * x * x.log()

x = torch.ones(1, requires_grad=True)
y = f(x)
y.backward()
x.grad
```

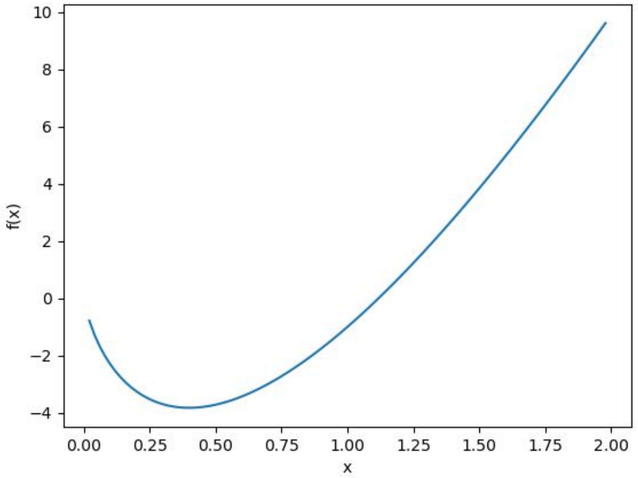
Out[82]: `tensor([8.])`



Gradient Descent

```
for i in range(20):
    x.grad.zero_()
    y = f(x)
    y.backward()
    print("x=%.3f  fx=%.3f  dfdx=%.3f" % (x, y, x.grad))
    step_size = 0.02
    with torch.no_grad():
        x -= step_size * x.grad
```

```
x=1.000  fx=-1.000  dfdx=8.000
x=0.840  fx=-2.170  dfdx=6.576
x=0.708  fx=-2.944  dfdx=5.137
x=0.606  fx=-3.404  dfdx=3.775
x=0.530  fx=-3.645  dfdx=2.595
x=0.478  fx=-3.756  dfdx=1.669
x=0.445  fx=-3.801  dfdx=1.012
x=0.425  fx=-3.817  dfdx=0.587
x=0.413  fx=-3.823  dfdx=0.330
x=0.406  fx=-3.824  dfdx=0.182
x=0.403  fx=-3.825  dfdx=0.099
x=0.401  fx=-3.825  dfdx=0.054
x=0.400  fx=-3.825  dfdx=0.029
x=0.399  fx=-3.825  dfdx=0.016
x=0.399  fx=-3.825  dfdx=0.008
x=0.399  fx=-3.825  dfdx=0.005
x=0.398  fx=-3.825  dfdx=0.002
x=0.398  fx=-3.825  dfdx=0.001
x=0.398  fx=-3.825  dfdx=0.001
x=0.398  fx=-3.825  dfdx=0.000
```



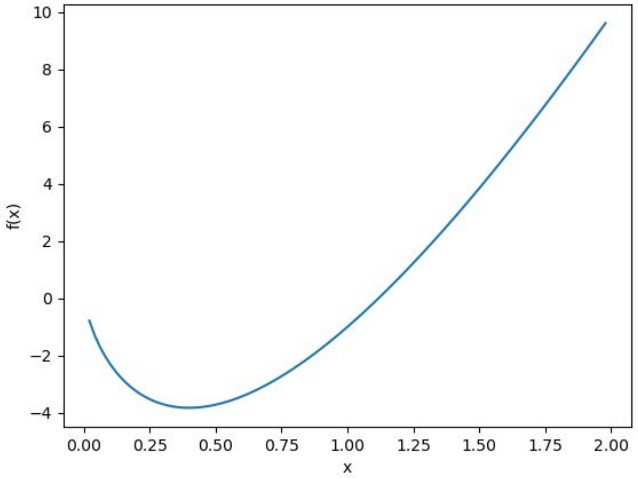
Practice Problem 3:

Minimizing our function with gradient descent

Stochastic Gradient Descent

```
def noisy_f(x):
    return -x**2 + 10 * x * x.log() + 0.1 * x * torch.randn(1)

x = torch.ones(1, requires_grad=True)
for i in range(1, 20):
    y = noisy_f(x)
    y.backward()
    print("x=%.3f  fx=%.3f  dfdx=%.3f" % (x, f(x), x.grad))
    step_size = 0.1 / (i**1.5)
    with torch.no_grad():
        x -= step_size * x.grad
    x.grad.zero_()

```

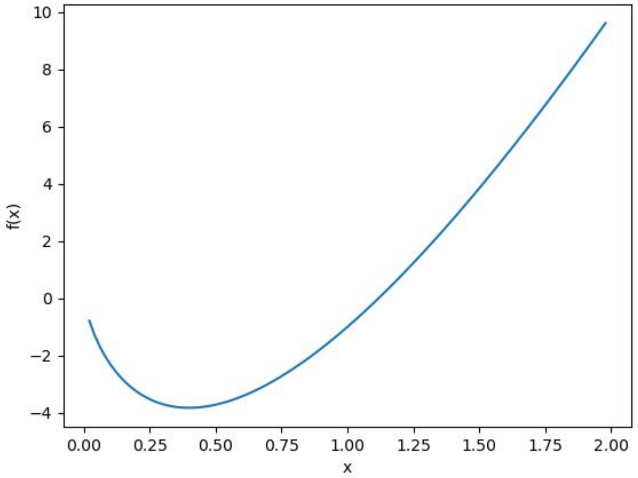
```
x=1.000  fx=-1.000  dfdx=7.844
x=0.216  fx=-3.355  dfdx=-5.812
x=0.421  fx=-3.819  dfdx=0.553
x=0.410  fx=-3.824  dfdx=0.268
x=0.407  fx=-3.824  dfdx=0.167
x=0.406  fx=-3.825  dfdx=0.053
x=0.405  fx=-3.825  dfdx=0.103
x=0.405  fx=-3.825  dfdx=0.141
x=0.404  fx=-3.825  dfdx=0.212
x=0.403  fx=-3.825  dfdx=-0.020
x=0.403  fx=-3.825  dfdx=0.285
x=0.403  fx=-3.825  dfdx=0.233
x=0.402  fx=-3.825  dfdx=0.021
x=0.402  fx=-3.825  dfdx=0.116
x=0.402  fx=-3.825  dfdx=0.107
x=0.402  fx=-3.825  dfdx=0.132
x=0.401  fx=-3.825  dfdx=0.070
x=0.401  fx=-3.825  dfdx=0.222
x=0.401  fx=-3.825  dfdx=0.120
```



PyTorch Optimizers

```python
x = torch.ones(1, requires_grad=True)

optimizer = torch.optim.SGD([x,], lr=0.02)

for i in range(1, 15):
    y = noisy_f(x)
    y.backward()
    print("x=%.3f  fx=%.3f  dfdx=%.3f" % (x, f(x), x.grad))
    optimizer.step()
    x.grad.zero_()

```

```
x=1.000  fx=-1.000  dfdx=8.171
x=0.837  fx=-2.193  dfdx=6.391
x=0.709  fx=-2.942  dfdx=5.235
x=0.604  fx=-3.410  dfdx=3.756
x=0.529  fx=-3.648  dfdx=2.561
x=0.478  fx=-3.757  dfdx=1.597
x=0.446  fx=-3.800  dfdx=0.977
x=0.426  fx=-3.816  dfdx=0.587
x=0.414  fx=-3.822  dfdx=0.409
x=0.406  fx=-3.824  dfdx=0.107
x=0.404  fx=-3.825  dfdx=0.081
x=0.403  fx=-3.825  dfdx=0.218
x=0.398  fx=-3.825  dfdx=-0.066
x=0.400  fx=-3.825  dfdx=0.087
```



Loss functions as objective functions

```
n = 10000
p = 5
dataset_x = torch.randn(n, p)  # a synthetic dataset

beta = torch.zeros(p, 1)
beta[2] = -1.0
beta[4] = 3.0

dataset_y = torch.mm(dataset_x, beta)  # and a response

def loss(x, y, beta_hat):
    yhat = torch.mm(x, beta_hat)
    return ((yhat - y)**2).mean()

beta_hat = torch.ones((p, 1), requires_grad=True)

optimizer = torch.optim.SGD([beta_hat,], lr=2e-1)

for i in range(1, 15):
    mse = loss(dataset_x, dataset_y, beta_hat)
    mse.backward()
    print("mse = %.3f" % (mse.item()))
    optimizer.step()
    beta_hat.grad.zero_()
```

Loss functions as objective functions

```
n = 10000
p = 5
dataset_x = torch.randn(n, p)  # a synthetic dataset

beta = torch.zeros(p, 1)
beta[2] = -1.0
beta[4] = 3.0

dataset_y = torch.mm(dataset_x, beta)  # and a response

def loss(x, y, beta_hat):
    yhat = torch.mm(x, beta_hat)
    return ((yhat - y)**2).mean()

beta_hat = torch.ones((p, 1), requires_grad=True)

optimizer = torch.optim.SGD([beta_hat,], lr=2e-1)

for i in range(1, 15):
    mse = loss(dataset_x, dataset_y, beta_hat)
    mse.backward()
    print("mse = %.3f" % (mse.item()))
    optimizer.step()
    beta_hat.grad.zero_()
```

```
mse = 11.019
mse = 3.956
mse = 1.421
mse = 0.511
mse = 0.184
mse = 0.066
mse = 0.024
mse = 0.009
mse = 0.003
mse = 0.001
mse = 0.000
mse = 0.000
mse = 0.000
mse = 0.000
```

```
beta_hat
```

```
tensor([[ 3.8991e-04],
        [ 6.4804e-04],
        [-9.9832e-01],
        [ 9.9378e-04],
        [ 2.9984e+00]], requires_grad=True)
```

Stochastic optimization with minibatches

```
n = 10000
p = 5
dataset_x = torch.randn(n, p)  # a synthetic dataset

beta = torch.zeros(p, 1)
beta[2] = -1.0
beta[4] = 3.0

dataset_y = torch.mm(dataset_x, beta)  # and a response

def loss(x, y, beta_hat):
    yhat = torch.mm(x, beta_hat)
    return ((yhat - y)**2).mean()

beta_hat = torch.ones((p, 1), requires_grad=True)

optimizer = torch.optim.SGD([beta_hat,], lr=2e-1)

for i in range(1, 15):
    minibatch = torch.randint(n, (128,))
    mse = loss(dataset_x[minibatch], dataset_y[minibatch], beta_hat)
    mse.backward()
    print("mse = %.3f" % (mse.item()))
    optimizer.step()
    beta_hat.grad.zero_()
```

```
mse = 12.523
mse = 3.308
mse = 1.214
mse = 0.388
mse = 0.186
mse = 0.055
mse = 0.030
mse = 0.007
mse = 0.002
mse = 0.001
mse = 0.000
mse = 0.000
mse = 0.000
mse = 0.000
```

```
beta_hat
```

```
tensor([[ 2.4282e-04],
        [ 7.5062e-04],
        [-9.9902e-01],
        [ 1.2907e-03],
        [ 2.9988e+00]], requires_grad=True)
```

Datasets and DataLoaders

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

tt_split = int(n * .9)
train_ds = TensorDataset(dataset_x[:tt_split], dataset_y[:tt_split])
test_ds = TensorDataset(dataset_x[tt_split:], dataset_y[tt_split:])

train_dl = DataLoader(train_ds, batch_size=128, shuffle=True, num_workers=0)
test_dl = DataLoader(test_ds, batch_size=128, shuffle=False, num_workers=0)

beta_hat = torch.ones((p, 1), requires_grad=True)
optimizer = torch.optim.SGD([beta_hat,], lr=2e-1)

for i, (x, y) in enumerate(train_dl):
    mse = loss(x, y, beta_hat)
    mse.backward()
    print("mse = %.3f" % (mse.item()))
    optimizer.step()
    beta_hat.grad.zero_()
```

```
mse = 9.103
mse = 4.647
mse = 1.902
mse = 0.829
mse = 0.231
mse = 0.104
mse = 0.030
```

MNIST dataset

```python
import torchvision
import torchvision.transforms as transforms

train_ds = torchvision.datasets.MNIST(root='.',
                                      train=True,
                                      transform=transforms.ToTensor(),
                                      download=True)

test_ds = torchvision.datasets.MNIST(root='.',
                                     train=False,
                                     transform=transforms.ToTensor())
```

```
0.1%
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./MNIST/raw/train-images-idx3-ubyte.gz
100.1%
Extracting ./MNIST/raw/train-images-idx3-ubyte.gz to ./MNIST/raw
113.5%
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz to ./MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./MNIST/raw/t10k-images-idx3-ubyte.gz
180.4%
Extracting ./MNIST/raw/t10k-images-idx3-ubyte.gz to ./MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./MNIST/raw/t10k-labels-idx1-ubyte.gz
Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw
Processing...
Done!
```

Exploring MNIST

```python
x, y = train_ds[0]
```

```python
y
```

```
5
```

```python
x.shape
```

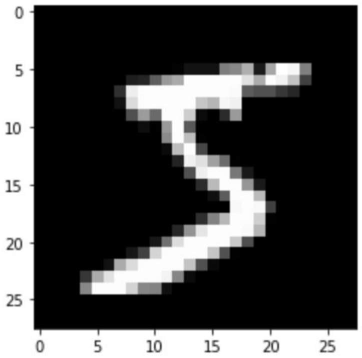
```
torch.Size([1, 28, 28])
```

```python
x[0,6:14,6:14]
```

```
tensor([[0.0000, 0.0000, 0.1176, 0.1412, 0.3686, 0.6039, 0.6667, 0.9922],
        [0.0000, 0.1922, 0.9333, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922],
        [0.0000, 0.0706, 0.8588, 0.9922, 0.9922, 0.9922, 0.9922, 0.9922],
        [0.0000, 0.0000, 0.3137, 0.6118, 0.4196, 0.9922, 0.9922, 0.8039],
        [0.0000, 0.0000, 0.0000, 0.0549, 0.0039, 0.6039, 0.9922, 0.3529],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.5451, 0.9922, 0.7451],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0431, 0.7451, 0.9922],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1373, 0.9451]])
```

```python
import matplotlib.pyplot as plt
plt.imshow(x[0], cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7f471c18c390>
```



Practice Problems 2, 3 & 4:

Minimizing a loss function using SGD

Reporting test set loss

```
beta_hat = torch.ones((p, 1), requires_grad=True)
optimizer = torch.optim.SGD([beta_hat,], lr=2e-1)

for epoch in range(5):
    train_mse = 0.0
    for i, (x, y) in enumerate(train_dl):
        mse = loss(x, y, beta_hat)
        mse.backward()
        optimizer.step()
        beta_hat.grad.zero_()
        train_mse += mse.item()

    test_mse = 0.0
    for i, (x, y) in enumerate(test_dl):
        mse = loss(x, y, beta_hat)
        test_mse += mse.item()

    train_mse /= len(train_dl)
    test_mse /= len(test_dl)

    print("train_mse = %.3f   test_mse = %.3f" % (train_mse, test_mse))
```

```
train_mse = 0.248   test_mse = 0.000
train_mse = 0.000   test_mse = 0.000
train_mse = 0.000   test_mse = 0.000
train_mse = 0.000   test_mse = 0.000
train_mse = 0.000   test_mse = 0.000
```

Refactoring

```
beta_hat = torch.ones((p, 1), requires_grad=True)
optimizer = torch.optim.SGD([beta_hat,], lr=2e-1)

def train_loop():
    train_mse = 0.0
    for i, (x, y) in enumerate(train_dl):
        mse = loss(x, y, beta_hat)
        mse.backward()
        optimizer.step()
        beta_hat.grad.zero_()
        train_mse += mse.item()

    return train_mse / len(train_dl)

def test_loop():
    test_mse = 0.0
    for i, (x, y) in enumerate(test_dl):
        mse = loss(x, y, beta_hat)
        test_mse += mse.item()

    return test_mse / len(test_dl)

for epoch in range(5):
    train_mse = train_loop()
    test_mse = test_loop()
    print("train_mse = %.3f   test_mse = %.3f" % (train_mse, test_mse))
```

Modules

```
from torch import nn


class LinearRegression(nn.Module):

    def __init__(self, p):
        super(LinearRegression, self).__init__()
        self.beta_hat = nn.Parameter(torch.ones((p, 1)))

    def forward(self, x):
        yhat = torch.mm(x, self.beta_hat)
        return yhat


lr = LinearRegression(p)
optimizer = torch.optim.SGD(lr.parameters(), lr=2e-1)
getmse = torch.nn.MSELoss()

def train_loop():
    train_mse = 0.0
    for i, (x, y) in enumerate(train_dl):
        yhat = lr(x)
        mse = getmse(y, yhat)
        lr.zero_grad()
        mse.backward()
        optimizer.step()
        beta_hat.grad.zero_()
        train_mse += mse.item()

    return train_mse / len(train_dl)
```

Built-in Linear modules

These two blocks are equivalent:

```
class LinearRegression(nn.Module):

    def __init__(self, p):
        super(LinearRegression, self).__init__()
        self.beta_hat = nn.Parameter(torch.ones((p, 1)))

    def forward(self, x):
        yhat = torch.mm(x, self.beta_hat)
        return yhat


lr = LinearRegression(p)
```

```
lr = nn.Linear(p, 1, bias=False)
```

Nonlinear data fit by a linear model

```
n = 10000
p = 5
dataset_x = torch.randn(n, p)
dataset_y = (dataset_x ** 4).sum(1)
```

```
model = Linear(p, 1)
criterion = MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)
```

Nonlinear data fit by a linear model

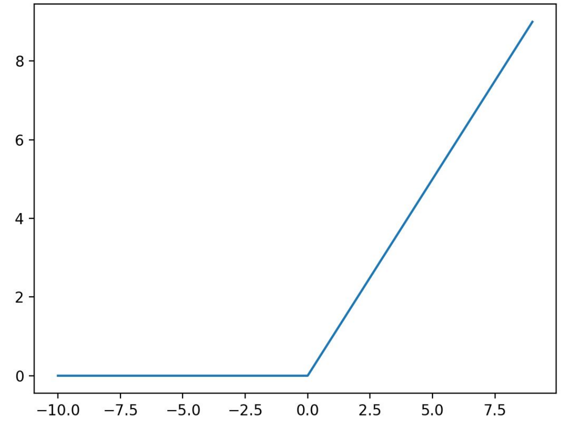
```python
for epoch in range(1000):
    train_mse = train_loop()
    test_mse = test_loop()
    print("train_mse = %.3f  test_mse=%.3f" % (train_mse, test_mse))

```

```
train_mse = 561.076  test_mse=496.446
train_mse = 488.869  test_mse=484.272
train_mse = 484.442  test_mse=483.528
train_mse = 484.006  test_mse=483.461
train_mse = 484.199  test_mse=483.510
train_mse = 484.396  test_mse=483.412
train_mse = 484.245  test_mse=483.409
train_mse = 484.343  test_mse=483.417
train_mse = 484.450  test_mse=483.555
train_mse = 484.334  test_mse=483.538
train_mse = 484.409  test_mse=483.567
train_mse = 484.254  test_mse=483.723
train_mse = 484.212  test_mse=483.500
```

Nonlinear data fit by a nonlinear model

```
hidden_dim = 32
model = Sequential(
    Linear(p, hidden_dim),
    ReLU(),
    Linear(hidden_dim, 1),
)
```

ReLU
(Rectified Linear Unit)



$$f(x) = \begin{cases} x \text{ if } x > 0 \\ 0 \text{ otherwise} \end{cases}$$

Nonlinear data fit by a nonlinear model

```python
for epoch in range(1000):
    train_mse = train_loop()
    test_mse = test_loop()
    print("train_mse = %.3f  test_mse=%.3f" % (train_mse, test_mse))

```

```
train_mse = 344.467  test_mse=237.223
train_mse = 202.776  test_mse=159.583
train_mse = 161.530  test_mse=140.376
train_mse = 147.999  test_mse=160.638
train_mse = 139.355  test_mse=128.781
train_mse = 134.093  test_mse=116.675
train_mse = 127.323  test_mse=129.027
train_mse = 119.114  test_mse=106.170
train_mse = 112.629  test_mse=100.606
train_mse = 103.340  test_mse=90.888
train_mse = 95.971  test_mse=90.992
 ...
train_mse = 3.553  test_mse=3.333
train_mse = 3.392  test_mse=3.245
```