```
import torch

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
import torch.nn as nn
```

Create a column matrix of X values

```
X = torch.linspace(1,50,50).reshape(-1,1)
# Equivalent to X = torch.unsqueeze(torch.linspace(1,50,50), dim=1)
```

Create a "random" array of error values

We want 50 random integer values that collectively cancel each other out.

```
torch.manual_seed(71) # to obtain reproducible results
e = torch.randint(-8,9,(50,1),dtype=torch.float)
print(e)
```

```
tensor([[ 2.],
        [ 7.],
        [ 2.],
        [ 6.],
        [ 2.],
        [-4.],
        [ 2.],
        [-5.],
        [ 4.],
        [ 1.],
        [ 2.],
        [ 3.],
        [ 1.],
        [-8.],
        [ 5.],
        [ 5.],
        [-6.],
        [ 0.],
        [-7.],
        [-8.],
        [-3.],
        [-1.],
        [ 2.],
        [-6.],
        [-3.],
        [ 3.],
        [ 2.],
        [ 3.],
        [ 4.],
```

```
        [ 5.],
        [ 1.],
        [ 7.],
        [ 6.],
        [-1.],
        [-6.],
        [-5.],
        [-3.],
        [ 7.],
        [ 0.],
        [ 8.],
        [-1.],
        [-2.],
        [ 2.],
        [-8.],
        [-1.],
        [ 6.],
        [-8.],
        [-3.],
        [-7.],
        [-2.]])
```

Create a column matrix of y values

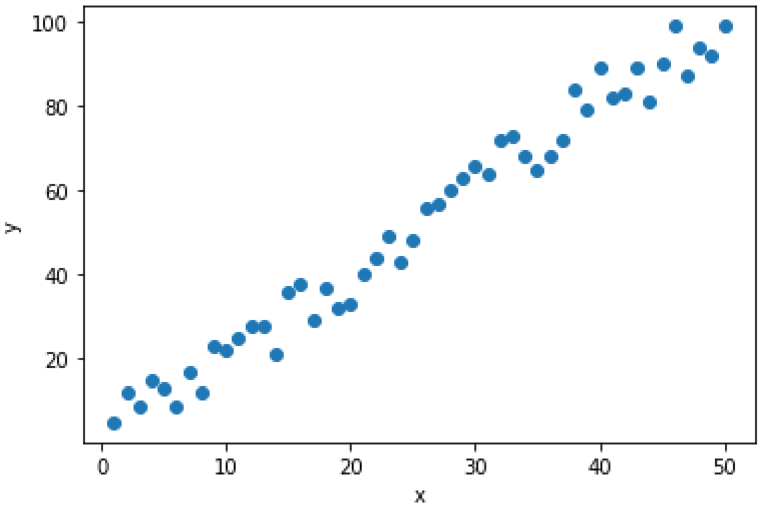
weight=2,bias=1,error amount=e.

```
y = 2*X + 1 + e
print(y.shape)
```

```
torch.Size([50, 1])
```

Plot the results

```
plt.scatter(X.numpy(), y.numpy())
plt.ylabel('y')
plt.xlabel('x');
```



SIMPLE LINEAR REGRESSION

---

how the built-in nn.Linear() model preselects weight and bias values at random.

```
torch.manual_seed(59)
model = nn.Linear(in_features=1, out_features=1)
print(model.weight)
print(model.bias)
```

```
Parameter containing:
tensor([[0.1060]], requires_grad=True)
Parameter containing:
tensor([0.9638], requires_grad=True)
```

models as object classes that can store a single linear layer.(Linear layers are also called "fully connected" or "dense" layers.)

```
class Model(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.linear = nn.Linear(in_features, out_features)


    def forward(self, x):
        y_pred = self.linear(x)
        return y_pred
```

```
torch.manual_seed(59)
model = Model(1, 1)
print('Weight: ',model.linear.weight)
print('Bias:  ', model.linear.bias)
```

```
Weight:  Parameter containing:
tensor([[0.1060]], requires_grad=True)
Bias:   Parameter containing:
tensor([0.9638], requires_grad=True)
```

```
for name, param in model.named_parameters():
    print(name, '\t', param.item())
```

```
linear.weight    0.10597813129425049
```

```
linear.bias      0.9637961387634277
```

```
x = torch.tensor([2.0])
print(model.forward(x))   # equivalent to print(model(x))
#f(x)=(0.1060)(2.0)+(0.9638)=1.1758
```

```
tensor([1.1758], grad_fn=<AddBackward0>)
```

Plot the initial model

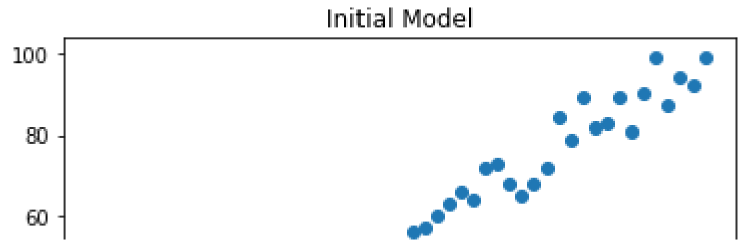
```
x1 = np.linspace(0.0,50.0,50)
print(x1)
```

```
[ 0.          1.02040816  2.04081633  3.06122449  4.08163265  5.10204082
  6.12244898  7.14285714  8.16326531  9.18367347 10.20408163 11.2244898
 12.24489796 13.26530612 14.28571429 15.30612245 16.32653061 17.34693878
 18.36734694 19.3877551  20.40816327 21.42857143 22.44897959 23.46938776
 24.48979592 25.51020408 26.53061224 27.55102041 28.57142857 29.59183673
 30.6122449  31.63265306 32.65306122 33.67346939 34.69387755 35.71428571
 36.73469388 37.75510204 38.7755102  39.79591837 40.81632653 41.83673469
 42.85714286 43.87755102 44.89795918 45.91836735 46.93877551 47.95918367
 48.97959184 50.        ]
```

```
w1= 0.1059
b1= 0.9637
y1 = x1*w1 + b1
print(y1)
```

```
[0.9637     1.07176122 1.17982245 1.28788367 1.3959449  1.50400612
 1.61206735 1.72012857 1.8281898  1.93625102 2.04431224 2.15237347
 2.26043469 2.36849592 2.47655714 2.58461837 2.69267959 2.80074082
 2.90880204 3.01686327 3.12492449 3.23298571 3.34104694 3.44910816
 3.55716939 3.66523061 3.77329184 3.88135306 3.98941429 4.09747551
 4.20553673 4.31359796 4.42165918 4.52972041 4.63778163 4.74584286
 4.85390408 4.96196531 5.07002653 5.17808776 5.28614898 5.3942102
 5.50227143 5.61033265 5.71839388 5.8264551  5.93451633 6.04257755
 6.15063878 6.2587    ]
```

```
plt.scatter(X.numpy(), y.numpy())
plt.plot(x1,y1,'r')
plt.title('Initial Model')
plt.ylabel('y')
plt.xlabel('x');
```



Set the loss function

```
linear_loss_function = nn.MSELoss()
```

Set the optimization

- *Here we'll use Stochastic Gradient Descent (SGD) with an applied learning rate (lr) of 0.001.Learning rate tells the optimizer how much to adjust each parameter on the next round of calculations. Too large a step and we run the risk of overshooting the minimum, causing the algorithm to diverge. Too small and it will take a long time to converge.*
- *For multivariate data, you might also consider passing optional momentum and weight_decay arguments.Momentum allows the algorithm to "roll over" small bumps to avoid local minima that can cause convergence too soon. Weight decay (also called an L2 penalty) applies to biases.*

```
optimizer = torch.optim.SGD(model.parameters(), lr = 0.001)
# Equivalent to optimizer = torch.optim.SGD(model.parameters(), lr = 1e-3)
```

Train the model

Let's walk through the steps we're about to take:

1. Set a reasonably large number of passes epochs = 50
2. Create a list to store loss values. This will let us view our progress afterward. losses = [] for i in range(epochs):
3. Bump "i" so that the printed report starts at 1 i+=1
4. Create a prediction set by running "X" through the current model parameters y_pred = model.forward(X)
5. Calculate the loss loss = criterion(y_pred, y)
6. Add the loss value to our tracking list losses.append(loss)
7. Print the current line of results print(f'epoch: {i:2} loss: {loss.item():10.8f}')
8. Gradients accumulate with every backprop. To prevent compounding we need to reset the stored gradient for each new epoch. optimizer.zero_grad()

9. Now we can backprop loss.backward()
10. Finally, we can update the hyperparameters of our model optimizer.step()

```
epochs = 50
losses = []

for i in range(epochs):
    i+=1
    #Predicting on forward pass
    y_pred = model.forward(X)
    #Calculate our loss(error)
    loss = linear_loss_function(y_pred, y)
    #Record that error
    losses.append(loss)
    print(f'epoch: {i}  loss: {loss.item()}  weight: {model.linear.weight.item(
bias: {model.linear.bias.item()}')
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

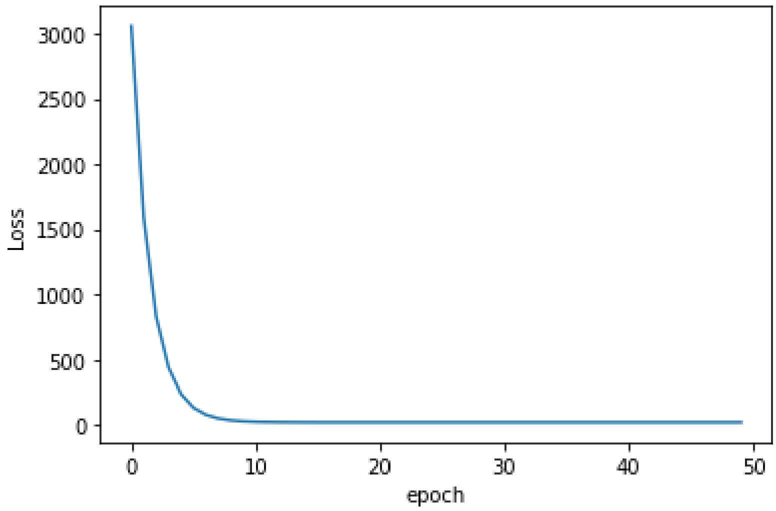
```
epoch: 1  loss: 3057.216796875  weight: 0.10597813129425049  bias: 0.96379613
epoch: 2  loss: 1588.53076171875  weight: 3.334900140762329  bias: 1.06046366
epoch: 3  loss: 830.2999267578125  weight: 1.014832854270935  bias: 0.9922628
epoch: 4  loss: 438.8521423339844  weight: 2.6817994117736816  bias: 1.042521
epoch: 5  loss: 236.76144409179688  weight: 1.4840213060379028  bias: 1.00766
epoch: 6  loss: 132.4291229248047  weight: 2.3446059226989746  bias: 1.033964
epoch: 7  loss: 78.56573486328125  weight: 1.7262253761291504  bias: 1.016321
epoch: 8  loss: 50.75775909423828  weight: 2.170504093170166  bias: 1.0302516
epoch: 9  loss: 36.4012336730957  weight: 1.8512457609176636  bias: 1.0214954
epoch: 10  loss: 28.98923110961914  weight: 2.0806007385253906  bias: 1.02903
epoch: 11  loss: 25.16238784790039  weight: 1.9157683849334717  bias: 1.02487
epoch: 12  loss: 23.186473846435547  weight: 2.034165620803833  bias: 1.02911
epoch: 13  loss: 22.166122436523438  weight: 1.9490584135055542  bias: 1.0273
epoch: 14  loss: 21.639110565185547  weight: 2.010172128677368  bias: 1.02985
epoch: 15  loss: 21.366769790649414  weight: 1.9662237167358398  bias: 1.0292
epoch: 16  loss: 21.225919723510742  weight: 1.997764229774475  bias: 1.03094
epoch: 17  loss: 21.152944564819336  weight: 1.9750648736953735  bias: 1.0309
epoch: 18  loss: 21.115013122558594  weight: 1.991337537765503  bias: 1.03220
epoch: 19  loss: 21.09518051147461  weight: 1.9796085357666016  bias: 1.03258
epoch: 20  loss: 21.084684371948242  weight: 1.9879988431930542  bias: 1.0335
epoch: 21  loss: 21.07901382446289  weight: 1.981933355331421  bias: 1.034103
epoch: 22  loss: 21.075830459594727  weight: 1.9862544536590576  bias: 1.0349
epoch: 23  loss: 21.07394027709961  weight: 1.9831126928329468  bias: 1.03558
epoch: 24  loss: 21.072702407836914  weight: 1.9853330850601196  bias: 1.0363
epoch: 25  loss: 21.071819305419922  weight: 1.9837009906768799  bias: 1.0370
epoch: 26  loss: 21.07110595703125  weight: 1.9848365783691406  bias: 1.03781
epoch: 27  loss: 21.070484161376953  weight: 1.9839837551116943  bias: 1.0385
epoch: 28  loss: 21.069913864135742  weight: 1.9845597743988037  bias: 1.0392
epoch: 29  loss: 21.06937026977539  weight: 1.9841090440750122  bias: 1.03995
epoch: 30  loss: 21.068838119506836  weight: 1.9843961000442505  bias: 1.0406
epoch: 31  loss: 21.068307876586914  weight: 1.984152913093567  bias: 1.04140
epoch: 32  loss: 21.067781448364258  weight: 1.9842908382415771  bias: 1.0421
epoch: 33  loss: 21.0672664642334  weight: 1.9841549396514893  bias: 1.042843
epoch: 34  loss: 21.066740036010742  weight: 1.9842157363891602  bias: 1.0435
epoch: 35  loss: 21.066225051879883  weight: 1.9841355085372925  bias: 1.0442
epoch: 36  loss: 21.065706253051758  weight: 1.9841564893722534  bias: 1.0450
```

```
epoch: 37  loss: 21.065185546875  weight: 1.9841045141220093  bias: 1.0457227
epoch: 38  loss: 21.06467056274414  weight: 1.9841052293777466  bias: 1.04644
epoch: 39  loss: 21.064157485961914  weight: 1.9840680360794067  bias: 1.0471
epoch: 40  loss: 21.063640594482422  weight: 1.984058141708374  bias: 1.04787
epoch: 41  loss: 21.063121795654297  weight: 1.984028697013855  bias: 1.04859
epoch: 42  loss: 21.062604904174805  weight: 1.9840131998062134  bias: 1.0493
epoch: 43  loss: 21.062095642089844  weight: 1.98398756980896  bias: 1.050029
epoch: 44  loss: 21.061574935913086  weight: 1.9839695692062378  bias: 1.0507
epoch: 45  loss: 21.061071395874023  weight: 1.9839458465576172  bias: 1.0514
epoch: 46  loss: 21.06055450439453  weight: 1.9839262962341309  bias: 1.05217
epoch: 47  loss: 21.060043334960938  weight: 1.9839037656784058  bias: 1.0528
epoch: 48  loss: 21.059534072875977  weight: 1.9838833808898926  bias: 1.0536
epoch: 49  loss: 21.05901527404785  weight: 1.9838614463806152  bias: 1.05432
epoch: 50  loss: 21.058507919311523  weight: 1.9838409423828125  bias: 1.0550
```

Plot the loss values

```
plt.plot(range(epochs), losses)
plt.ylabel('Loss')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



Plot the current model

```
x= np.linspace(0.0,50.0,50)
current_weight = model.linear.weight.item()
current_bias = model.linear.bias.item()
predicted_y = current_weight * x + current_bias
print(predicted_y)
```

```
[  1.05575156   3.08005679   5.10436203   7.12866726   9.15297249
  11.17727772  13.20158295  15.22588818  17.25019342  19.27449865
  21.29880388  23.32310911  25.34741434  27.37171957  29.39602481
  31.42033004  33.44463527  35.4689405   37.49324573  39.51755096
  41.5418562   43.56616143  45.59046666  47.61477189  49.63907712
```

```
 51.66338236  53.68768759  55.71199282  57.73629805  59.76060328
 61.78490851  63.80921375  65.83351898  67.85782421  69.88212944
 71.90643467  73.9307399   75.95504514  77.97935037  80.0036556
 82.02796083  84.05226606  86.07657129  88.10087653  90.12518176
 92.14948699  94.17379222  96.19809745  98.22240268 100.24670792]
```

```
plt.scatter(X.numpy(), y.numpy())
plt.plot(x,predicted_y,'r')
plt.title('Current Model')
plt.ylabel('y')
plt.xlabel('x')
```

```
Text(0.5, 0, 'x')
```