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

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.sum())
```

```
tensor(0.)
```

Create a column matrix of y values

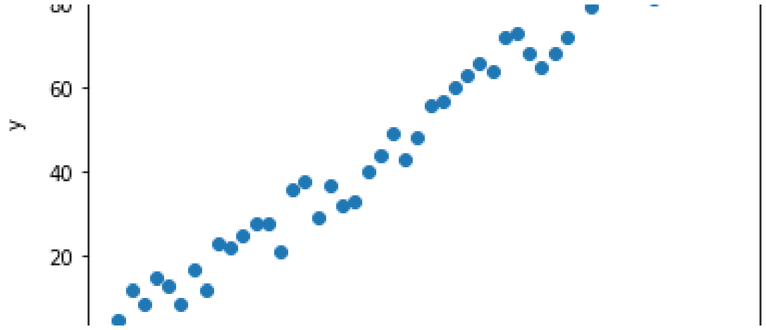
`* weight=2,bias=1,error amount=e.*`

```
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');
```



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(model)
print('Weight:', model.linear.weight.item())
print('Bias:  ', model.linear.bias.item())
```

```
Model(
  (linear): Linear(in_features=1, out_features=1, bias=True)
)
Weight: 0.10597813129425049
```

```
Bias:   0.9637961387634277
```

```
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.array([X.min(),X.max()])
print(x1)
```

```
[ 1. 50.]
```

```
w1,b1 = model.linear.weight.item(), model.linear.bias.item()
print(f'Initial weight: {w1:.8f}, Initial bias: {b1:.8f}')
y1 = x1*w1 + b1
print(y1)
```

```
Initial weight: 0.10597813, Initial bias: 0.96379614
[1.0697743 6.2627025]
```

```
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 = []

```

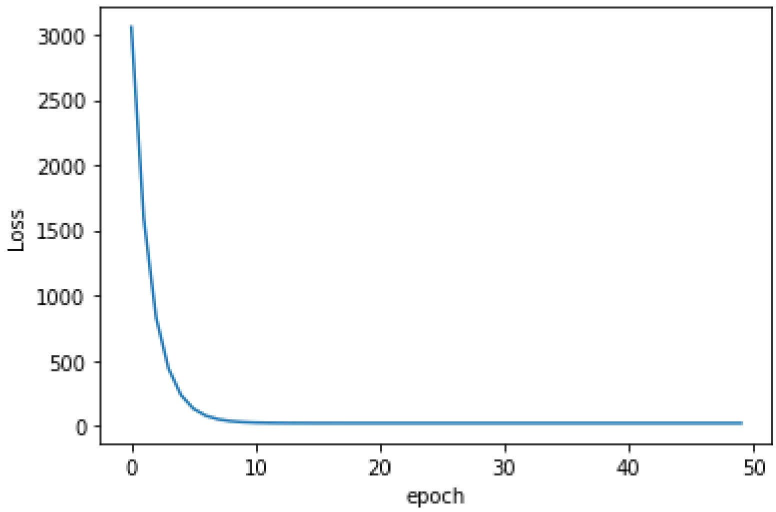
```
for i in range(epochs):
    i+=1
    y_pred = model.forward(X)
    loss = linear_loss_function(y_pred, y)
    losses.append(loss)
    print(f'epoch: {i:2}  loss: {loss.item():10.8f}  weight: {model.linear.weig
bias: {model.linear.bias.item():10.8f}')
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
epoch:  1  loss: 3057.21679688  weight: 0.10597813  bias: 0.96379614
epoch:  2  loss: 1588.53076172  weight: 3.33490014  bias: 1.06046367
epoch:  3  loss: 830.29992676  weight: 1.01483285  bias: 0.99226284
epoch:  4  loss: 438.85214233  weight: 2.68179941  bias: 1.04252183
epoch:  5  loss: 236.76144409  weight: 1.48402131  bias: 1.00766504
epoch:  6  loss: 132.42912292  weight: 2.34460592  bias: 1.03396463
epoch:  7  loss: 78.56573486  weight: 1.72622538  bias: 1.01632178
epoch:  8  loss: 50.75775909  weight: 2.17050409  bias: 1.03025162
epoch:  9  loss: 36.40123367  weight: 1.85124576  bias: 1.02149546
epoch: 10  loss: 28.98923111  weight: 2.08060074  bias: 1.02903891
epoch: 11  loss: 25.16238785  weight: 1.91576838  bias: 1.02487016
epoch: 12  loss: 23.18647385  weight: 2.03416562  bias: 1.02911627
epoch: 13  loss: 22.16612244  weight: 1.94905841  bias: 1.02731562
epoch: 14  loss: 21.63911057  weight: 2.01017213  bias: 1.02985907
epoch: 15  loss: 21.36676979  weight: 1.96622372  bias: 1.02928054
epoch: 16  loss: 21.22591972  weight: 1.99776423  bias: 1.03094459
epoch: 17  loss: 21.15294456  weight: 1.97506487  bias: 1.03099668
epoch: 18  loss: 21.11501312  weight: 1.99133754  bias: 1.03220642
epoch: 19  loss: 21.09518051  weight: 1.97960854  bias: 1.03258383
epoch: 20  loss: 21.08468437  weight: 1.98799884  bias: 1.03355861
epoch: 21  loss: 21.07901382  weight: 1.98193336  bias: 1.03410351
epoch: 22  loss: 21.07583046  weight: 1.98625445  bias: 1.03495669
epoch: 23  loss: 21.07394028  weight: 1.98311269  bias: 1.03558779
epoch: 24  loss: 21.07270241  weight: 1.98533309  bias: 1.03637791
epoch: 25  loss: 21.07181931  weight: 1.98370099  bias: 1.03705311
epoch: 26  loss: 21.07110596  weight: 1.98483658  bias: 1.03781021
epoch: 27  loss: 21.07048416  weight: 1.98398376  bias: 1.03850794
epoch: 28  loss: 21.06991386  weight: 1.98455977  bias: 1.03924775
epoch: 29  loss: 21.06937027  weight: 1.98410904  bias: 1.03995669
epoch: 30  loss: 21.06883812  weight: 1.98439610  bias: 1.04068720
epoch: 31  loss: 21.06830788  weight: 1.98415291  bias: 1.04140162
epoch: 32  loss: 21.06778145  weight: 1.98429084  bias: 1.04212701
epoch: 33  loss: 21.06726646  weight: 1.98415494  bias: 1.04284394
epoch: 34  loss: 21.06674004  weight: 1.98421574  bias: 1.04356635
epoch: 35  loss: 21.06622505  weight: 1.98413551  bias: 1.04428422
epoch: 36  loss: 21.06570625  weight: 1.98415649  bias: 1.04500473
epoch: 37  loss: 21.06518555  weight: 1.98410451  bias: 1.04572272
epoch: 38  loss: 21.06467056  weight: 1.98410523  bias: 1.04644191
epoch: 39  loss: 21.06415749  weight: 1.98406804  bias: 1.04715967
epoch: 40  loss: 21.06364059  weight: 1.98405814  bias: 1.04787791
epoch: 41  loss: 21.06312180  weight: 1.98402870  bias: 1.04859519
epoch: 42  loss: 21.06260490  weight: 1.98401320  bias: 1.04931259
epoch: 43  loss: 21.06209564  weight: 1.98398757  bias: 1.05002928
epoch: 44  loss: 21.06157494  weight: 1.98396957  bias: 1.05074584
epoch: 45  loss: 21.06107140  weight: 1.98394585  bias: 1.05146194
epoch: 46  loss: 21.06055450  weight: 1.98392630  bias: 1.05217779
epoch: 47  loss: 21.06004333  weight: 1.98390377  bias: 1.05289316
epoch: 48  loss: 21.05953407  weight: 1.98388338  bias: 1.05360830
epoch: 49  loss: 21.05901527  weight: 1.98386145  bias: 1.05432308
```

```
epoch: 50  loss: 21.05850792  weight: 1.98384094  bias: 1.05503750
```

Plot the loss values

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

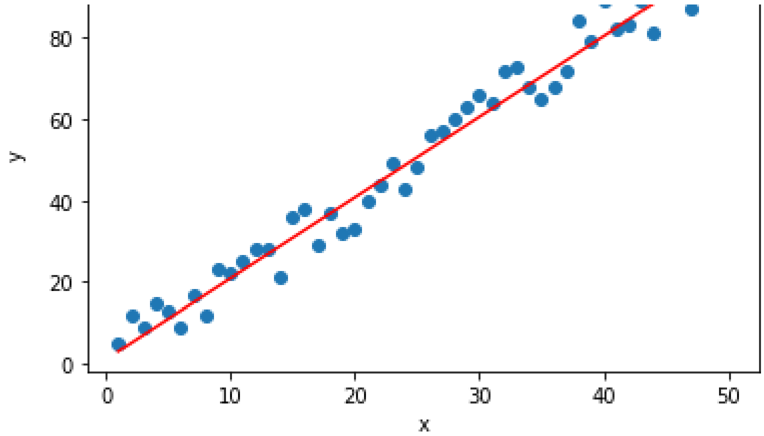


Plot the current model

```
w1,b1 = model.linear.weight.item(), model.linear.bias.item()
print(f'Current weight: {w1:.8f}, Current bias: {b1:.8f}')
y1 = x1*w1 + b1
print(x1)
print(y1)
```

```
Current weight: 1.98381913, Current bias: 1.05575156
[ 1. 50.]
[  3.0395708 100.246704 ]
```

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


y
x
0
20
40
60
80
0
10
20
30
40
50