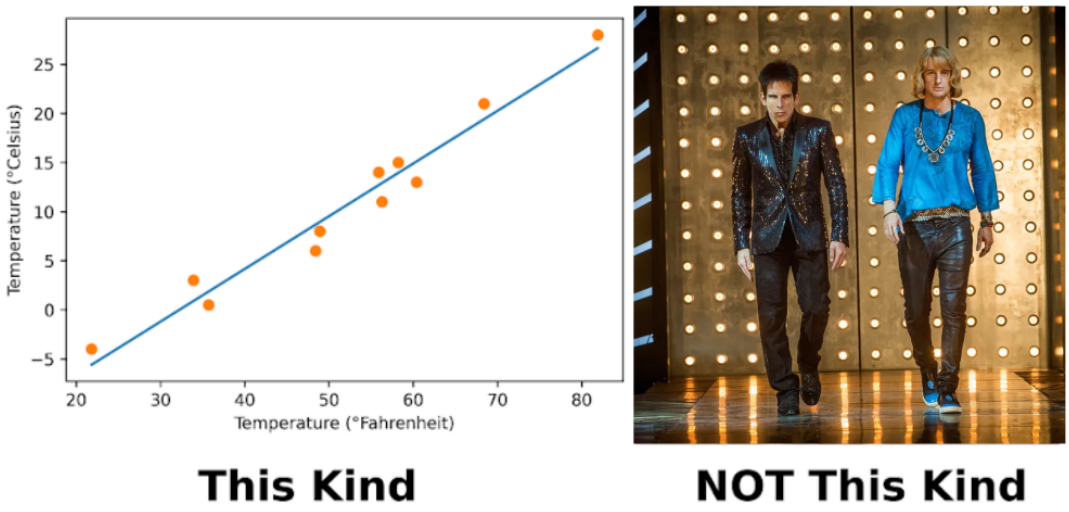
Classic Modeling Methods vs. PyTorch Modeling

> A Practice Example by Thom Ives, Ph.D.

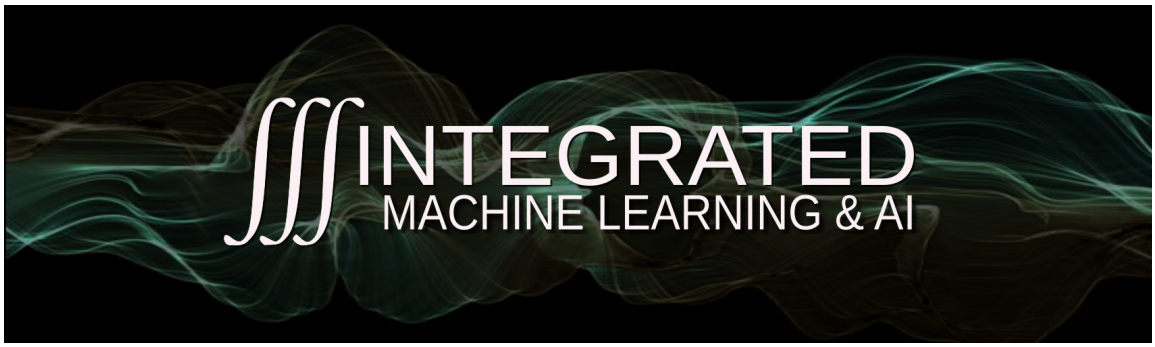
We gained important skills in loading our data in the correct format into PyTorch tensors. Hopefully you gained a good sense of the efficiency of PyTorch tensors. I know that in this review, I appreciate the raw elegance and power of PyTorch much more.

It would be natural at this point to wonder what elegance and power PyTorch brings us once we start to train models for machine and deep learning and for other types of modeling activities such as the predictive modeling of multi-physical systems.



This first part of step 7 in our PyTorch journey demonstrates just this. It is just an initial illustration though. There will be more. This illustration is a good first step.

You can find the latest revisions of this ongoing **PyTorch_Journey** in my Git Repo on **DagsHub**. This PDF is in the directory **7_Modeling**. I hope you'll sign up for your own FREE DagsHub account - the cloud based git repo service focused on practitioners of data science, machine learning, and artificial intelligence. **DagsHub** does NOT pay me to promote them. I simply believe they are on a good trajectory. And I still love GitHub - a lot.



In **Deep Learning with PyTorch**, the awesome book that I am using as my primary learning source on this journey, the authors truly outdo themselves in teaching style in chapter 5. I strongly encourage you to own and read this book. It's not only informative; it is very well written; it's even inspiring in teaching style. They are especially good at teaching model training using history and natural common intuitions.

Rather than try to recover their excellent teaching style in chapter 5, which I hope you will all read, I will instead summarize how model training in PyTorch provides powerful advantages over the traditional model training. I hope that you will read up to 5.5.1 in chapter 5 before reading this, but if you don't yet have the book, I believe this will still be informative for you.

The next several cells would be used in both the traditional approach and in a PyTorch approach (watch for `# comments` in the cells too). I will specifically note the differences. I hope you will run the cells yourself from the cloned notebooks - model_1.ipynb (for classical approach) and model_2.ipynb (for PyTorch approach). You can run them side by side in VS Code. This PDF was generated from the model_train_compare.ipynb, but I wouldn't want to claim that it would run without mishap since I am mixing the two cases.

In [ ]:
```
import torch  # both approaches
```

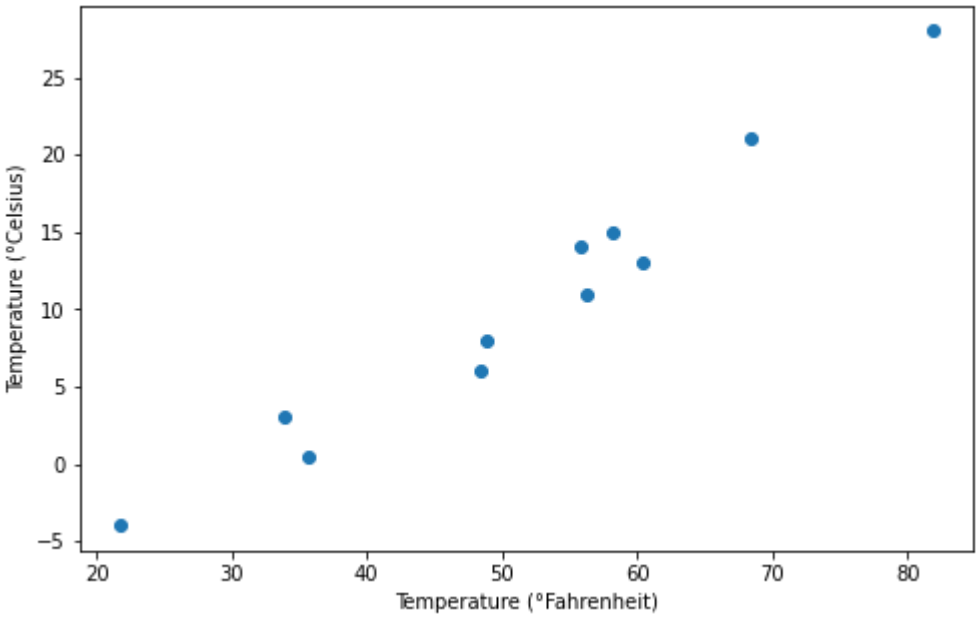
I love the authors of **Deep Learning with PyTorch**. They not only honor the great minds before us, but they use fake data and excellent storytelling. Their story around the data below is awesome, and they even wove it around this fake data!

In [ ]:
```
# both approaches
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c)
t_u = torch.tensor(t_u)
```

Let's do an initial visualization of this fake temperature data.

In [ ]:
```
%matplotlib inline
from matplotlib import pyplot as plt


fig = plt.figure(figsize=(8, 5))
plt.xlabel("Temperature (°Fahrenheit)")
plt.ylabel("Temperature (°Celsius)")
plt.plot(t_u.numpy(), t_c.numpy(), 'o');
```



So our job is to find the best fit line through these noisy fake data points for the unmarked thermometer explained in chapter 5 of **Deep Learning with PyTorch**.

In [ ]:
```
def model(t_u, w, b):  # both approaches
    return w * t_u + b
```

In [ ]:
```
def loss_fn(t_p, t_c):  # both approaches
    squared_diffs = (t_p - t_c)**2

    return squared_diffs.mean()
```

The Differences

Classical Approach

The next cell would be used in an classical approach when not using PyTorch advantages. **NOTE** that the principles are the same in both cases. PyTorch is simply taking care of specific detailed modeling tasks with added and greater flexibility for us to reduce the amount of our work.

In [ ]:
```
# w = torch.ones(0)  # causes an issue
# w = torch.ones(1)  # creates a 1D tensor of length 1
w = torch.ones(())  # The RIGHT WAY to create a 0D tensor - a scalar
b = torch.zeros(())
print(w, b)

# w and b comprise our model parameters
t_p = model(t_u, w, b)
t_p
```

Out[ ]:
```
tensor(1.) tensor(0.)
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000, 21.8
000,
        48.4000, 60.4000, 68.4000])
```

Next, we will run the loss function initially.

In [ ]:
```
loss = loss_fn(t_p, t_c)
loss
```

Out[ ]:
```
tensor(1763.8848)
```

Next, we need to think about how to code the gradient for our loss functions with respect to each of our model parameters so that we can use gradient descent to find the best parameters that minimize our losses during model training.

We will use the chain rule as shown below. We see that our gradient function has two parts: a gradient to analyze changes for our model weight `w`; a gradient to analyze changes for our model weight `b`.

So we need to code partial derivatives for the gradient of our loss function with respect to our model, and for our model with respect to each model parameter.

$$\nabla L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right) = \left(\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b} \right)$$

For our loss function with respect to our model, we have the following function.

In [ ]:
```
def dloss_fn(t_p, t_c):
    dsq_diffs = 2 * (t_p - t_c) / t_p.size(0)

    return dsq_diffs
```

For the partial derivative of the model with respect to `w`, we have the next function.

In [ ]:
```
def dmodel_dw(t_u, w, b):
    return t_u
```

Then, in the next cell, we have our function for the partial derivative of the model with respect to `b`.

In [ ]:
```
def dmodel_db(t_u, w, b):
    return 1.0
```

Finally, we put these functions together in our function for the gradient and return the gradient for each parameter as a stacked tensor.

If there's a part of the code you are not clear on, separate the code statements into separate cells and run each one and check the intermediate output. That's how you answer you own questions better than anyone else can.

In [ ]:
```
def grad_fn(t_u, t_c, t_p, w, b):
    dloss_dtp = dloss_fn(t_p, t_c)
    dloss_dw = dloss_dtp * dmodel_dw(t_u, w, b)
    dloss_db = dloss_dtp * dmodel_db(t_u, w, b)

    return torch.stack([dloss_dw.sum(), dloss_db.sum()])
```

Now we can put everything together into our training loop function.

In [ ]:
```
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        w, b = params
        t_p = model(t_u, w, b)
        loss = loss_fn(t_p, t_c)
        grad = grad_fn(t_u, t_c, t_p, w, b)
        params = params - learning_rate * grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
            print(f'\t Params:    {params}')
            print(f'\t Gradients: {grad}')

    return params
```

A Matter of Numerical Magnitude Management

To make the training go smoother, we don't want our two gradients to be so wildly different, so let's scale the large of the two. The method below is crude, but more than adequate. In real work, it would be best to use something like standard scaling or such. We really just want to focus on model training comparisons for now. We will use this crude scaling for both approaches.

In [ ]:
```
t_un = 0.1 * t_u
```

Running the Training Loop Function

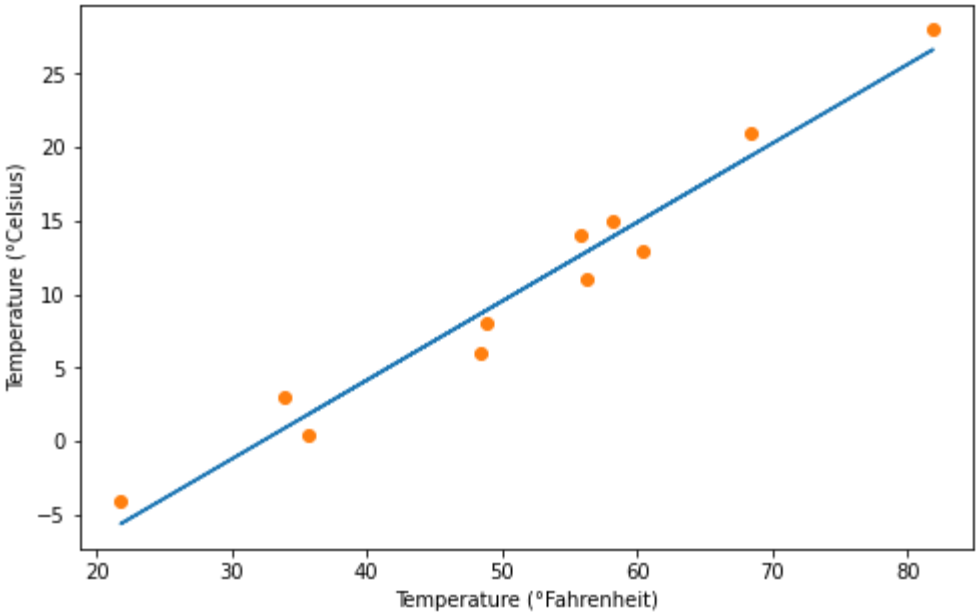
In [ ]:
```
params = training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un,
    t_c = t_c)
```

```
Epoch 500, Loss 7.860115
         Params:    tensor([ 4.0443, -9.8133])
         Gradients: tensor([-0.2252,  1.2748])
Epoch 1000, Loss 3.828538
         Params:    tensor([  4.8021, -14.1031])
         Gradients: tensor([-0.0962,  0.5448])
Epoch 1500, Loss 3.092191
         Params:    tensor([  5.1260, -15.9365])
         Gradients: tensor([-0.0411,  0.2328])
Epoch 2000, Loss 2.957698
         Params:    tensor([  5.2644, -16.7200])
         Gradients: tensor([-0.0176,  0.0995])
Epoch 2500, Loss 2.933134
         Params:    tensor([  5.3236, -17.0549])
         Gradients: tensor([-0.0075,  0.0425])
Epoch 3000, Loss 2.928648
         Params:    tensor([  5.3489, -17.1980])
         Gradients: tensor([-0.0032,  0.0182])
Epoch 3500, Loss 2.927830
         Params:    tensor([  5.3597, -17.2591])
         Gradients: tensor([-0.0014,  0.0078])
Epoch 4000, Loss 2.927680
         Params:    tensor([  5.3643, -17.2853])
         Gradients: tensor([-0.0006,  0.0033])
Epoch 4500, Loss 2.927651
         Params:    tensor([  5.3662, -17.2964])
         Gradients: tensor([-0.0002,  0.0014])
Epoch 5000, Loss 2.927648
         Params:    tensor([  5.3671, -17.3012])
         Gradients: tensor([-0.0001,  0.0006])
```

Let's visualize our actual and predicted values.

In [ ]:
```
%matplotlib inline
from matplotlib import pyplot as plt


t_p = model(t_un, *params)
fig = plt.figure(figsize=(8, 5))
plt.xlabel("Temperature (°Fahrenheit)")
plt.ylabel("Temperature (°Celsius)")
plt.plot(t_u.numpy(), t_p.detach().numpy())
plt.plot(t_u.numpy(), t_c.numpy(), 'o');
```



The PyTorch Approach (Uses the Classical Elements **AND** Coding Elegance)

With a PyTorch approach, we want to do two things at this point. Create a single tensor that stores our parameters, and then, seeing that tensors are powerful data storage objects (classes) that store much meta data regarding themselves, we will even let this tensor store gradient information for the model training.

In [ ]:
```
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

Let's check the initial state of the grad attribute for this tensor class instance for params. It's None. That's what we'd expect. No training has happened yet.

In [ ]:
```
params.grad is None
```

Out[ ]:
```
True
```

Now, we will run our model through our loss function. The input for the params is a tensor with `required_grad=True`. Since the loss_fn output is a tensor that also has `required_grad=True`, we can run the tensor class's `backward` method on that loss tensor. Then, we can look at the current gradients calculated for our `params` tensor.

In [ ]:
```
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()

params.grad
```

Out[ ]:
```
tensor([4517.2969,   82.6000])
```

WHAT?!?! What kind of magic was that?!?! Yes. I agree. This is amazing.

Back in grad school, when creating tensors much like PyTorch tensors for coding neural networks in C, I had conceived of storing gradient information, but I had not thought of doing it near as elegantly as PyTorch does it. This is about to get good.

And is params.grad still None? No. It has been populated now.

In [ ]:
```
params.grad is None
```

Out[ ]:
```
False
```

Note how we don't now need our gradient functions. PyTorch is handling that for us. But, there are some changes to our training loop function that may not be clear.

First, notice that we zero out our params.grad in each training epoch (loop). Why? The params.grad actually accumulates. We don't yet want that. We will use those accumulations in the future, but not in this case yet.

Next, we call the forward pass model, then the loss function, and finally run the AMAZING backward method on our loss tensor as explained previously.

In [ ]:
```
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
            print(f'\t Params:    {params}')
            print(f'\t Gradients: {params.grad}')

    return params
```

When we pause to consider what we've witnessed above, WOW! We thought PyTorch tensors were cool, but this is already over the top AWESOME. AND there is more coming!

Let's now run the training to find our best parameters that minimize our loss function.

In [ ]:
```
t_un = 0.1 * t_u

training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),
    t_u = t_un,
    t_c = t_c)
```

```
Epoch 500, Loss 7.860115
         Params:    tensor([ 4.0443, -9.8133], requires_grad=True)
         Gradients: tensor([-0.2252,  1.2748])
Epoch 1000, Loss 3.828538
         Params:    tensor([  4.8021, -14.1031], requires_grad=True)
         Gradients: tensor([-0.0962,  0.5448])
Epoch 1500, Loss 3.092191
         Params:    tensor([  5.1260, -15.9365], requires_grad=True)
         Gradients: tensor([-0.0411,  0.2328])
Epoch 2000, Loss 2.957698
         Params:    tensor([  5.2644, -16.7200], requires_grad=True)
         Gradients: tensor([-0.0176,  0.0995])
Epoch 2500, Loss 2.933134
         Params:    tensor([  5.3236, -17.0549], requires_grad=True)
         Gradients: tensor([-0.0075,  0.0425])
Epoch 3000, Loss 2.928648
         Params:    tensor([  5.3489, -17.1980], requires_grad=True)
         Gradients: tensor([-0.0032,  0.0182])
Epoch 3500, Loss 2.927830
         Params:    tensor([  5.3597, -17.2591], requires_grad=True)
         Gradients: tensor([-0.0014,  0.0078])
Epoch 4000, Loss 2.927679
         Params:    tensor([  5.3643, -17.2853], requires_grad=True)
         Gradients: tensor([-0.0006,  0.0033])
Epoch 4500, Loss 2.927652
         Params:    tensor([  5.3662, -17.2964], requires_grad=True)
         Gradients: tensor([-0.0003,  0.0014])
Epoch 5000, Loss 2.927647
         Params:    tensor([  5.3671, -17.3012], requires_grad=True)
         Gradients: tensor([-9.7275e-05,  6.1280e-04])
```

Out[ ]:
```
tensor([  5.3671, -17.3012], requires_grad=True)
```

Nice. We got the same answers with both methods, but PyTorch truly economized our work. Also note that we were using PyTorch tensors in the classical method. Imagine how much more work we'd need to do if we hadn't used those. We will forgo the plot of the data seeing that the params were the same.

If you are like me, you are thinking that the above was amazing! Well, there is more coming. I hope you will continue on this journey with me, and I hope you will get a copy of **Deep Learning with PyTorch**. I really love this book, and chapter 5 is really nice!

Optimizers and Train Test Splits for Model Training

Next, we'll cover optimizers for model training and splitting our data sets into training and test (validation) sets. These are two distinct topics. It has nothing to do with optimizing the amount of train test splits nor with physical splits.

Optimizers for Model Training

The method of model training used in the above examples, both with and without PyTorch, was stochastic gradient descent. But PyTorch makes available many popular solvers / optimizers for solving models with training data. We import the optimizers with `torch.optim` and view them with the built-in `dir` function.

In [ ]:
```
import torch.optim as optim
```

In [ ]:
```
dir(optim)
```

Out[ ]:
```
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'AdamW',
 'Adamax',
 'LBFGS',
 'NAdam',
 'Optimizer',
 'RAdam',
 'RMSprop',
 'Rprop',
 'SGD',
 'SparseAdam',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_functional',
 '_multi_tensor',
 'lr_scheduler',
 'swa_utils']
```

Let's start by defining an optimizer of SGD (Stochastic Gradient Descent).

In [ ]:
```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-5
optimizer = optim.SGD([params], lr=learning_rate)
```

In the next cell, we make one step toward improved parameters by:

1. making a prediction with our current parameters,
2. calculating the losses,
3. performing back propagation, and
4. using the gradients calculated in `backward` in an optimizer step.

That last step uses the optimizer to adjust our parameters by one step using the gradients. The step size is the gradients run through the optimizer and scaled by the learning rate as was accomplished in the last line in the previous cell of code. When the `optim.SGD` class is instantiated, it receives the params and the `learning_rate`, so that when the `optim.step()` method is called, it can update the params according to the chosen optimizer using the latest gradients.

In [ ]:
```
t_p = model(t_u, *params)
loss = loss_fn(t_p, t_c)
loss.backward()
optimizer.step()

params
```

Out[ ]:
```
tensor([ 9.5483e-01, -8.2600e-04], requires_grad=True)
```

Now that we have that working, let's creating a training loop to do it repeatedly to reduce our losses to a very low value.

In [ ]:
```
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

Great. Now we define our initial params and make sure "requires gradient" (`requires_grad`) is set to True, establish a `learning_rate` , and instantiate the stochastic gradient descent optimizer. With those established, we call our training loop function passing in 5000 epochs, our optimizer, our params, tu, and tc.

We quickly converge to known good parameters. Notice that we passed in our scaled inputs.

In [ ]:
```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)
training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    t_c = t_c)
```

```
Epoch 500, Loss 7.860115
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957698
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647
```

Out[ ]:
```
tensor([  5.3671, -17.3012], requires_grad=True)
```

Adam is a very powerful optimizer, and it's even insensitive to features with large variations in magnitude. Thus, we won't use the scaled version of t_u. We can also get away with using a large learning rate when using Adam, which also allows us to converge with fewer epochs.

In [ ]:
```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
    params = params,
    t_u = t_u,
    t_c = t_c)
```

```
Epoch 500, Loss 7.612900
Epoch 1000, Loss 3.086700
Epoch 1500, Loss 2.928579
Epoch 2000, Loss 2.927644
```

Out[ ]:
```
tensor([  0.5367, -17.3021], requires_grad=True)
```

Train Test Split

In the book, only 10 points were used. This is OK for a very simple example, but let's create more points for our thermometer over a wider range that includes freezing water and boiling ice. This way, we we do a train test split of our data, the test side (validation part) will have 20 points instead of only 2, and the train part will have 80.

In [ ]:
```
import numpy as np

t_c = np.hstack((np.random.uniform(0, 100, 98), np.array([0, 100])))
np.random.shuffle(t_c)
t_c
```

Out[ ]:
```
array([ 34.28343009,  70.96021811,  18.15663365,  33.37667389,
        13.97795805,  15.61863329,  60.36185823,  81.8839969 ,
        33.44027665,  98.79415124,  55.15146253,   2.57207224,
        98.70995502,  28.01628349,  19.93638095,  50.81853423,
        95.2324126 ,  32.96707772,  79.31359244,   1.34741748,
        25.64876124,   0.60545099,  50.23222931,  88.49448188,
        87.33522037,   3.20829193,  98.79124443,  86.28969339,
        26.06667988,  73.11816701,  59.30406023,  35.47745183,
        68.46767188,   4.61657849,  57.45158125,  99.39526851,
        92.18223884,   6.26224142,  15.78956106,  21.82730539,
        63.39581831,  72.184266  ,  73.64467721,  92.5400599 ,
        45.29690111,   2.33753663,  40.27366012,  45.04632538,
         4.81469443,  58.0767224 ,  91.40962829,  46.38556352,
        59.25235502,  77.42879083,  58.07563529,   1.44872474,
        19.52000249,  45.91322475,  83.1927989 ,  88.92988273,
        72.70916434,  57.56763747,  82.94513822,  80.92303478,
        92.30107783,  13.69199627,  56.2829343 , 100.        ,
        14.14924561,  21.21636928,  99.45547799,  37.27270148,
        88.8649768 ,  80.22742039,  65.44427781,   2.75830376,
        15.42781024,  22.64789119,  52.61508799,  33.60742384,
         6.23593482,  84.76750824,  22.47375851,  24.3653151 ,
        77.25255429,   0.        ,  59.56674011,  44.97421949,
        19.31481319,   1.01203282,  90.33024546,  88.96676538,
        52.5139942 ,  81.85275482,  94.84059067,  88.87164357,
        63.6692429 ,  46.92070778,  38.79127986,  75.02623643])
```

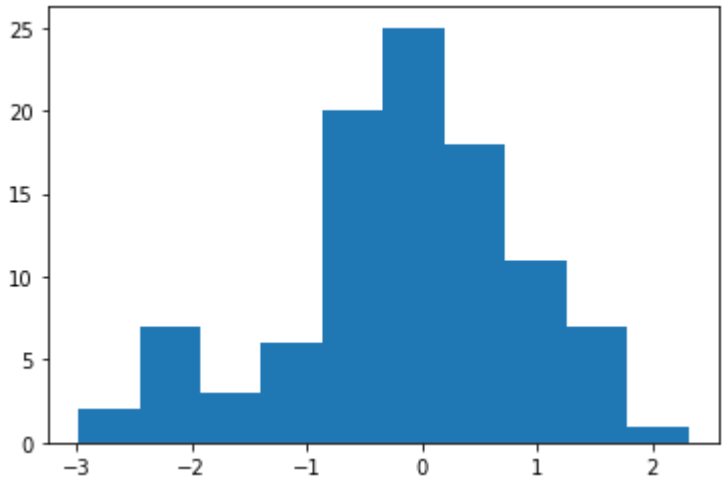
Next, we convert our unknown temperature scale numbers to Fahrenheit, because we did know these were Fahrenheit. We are being fake data prophet for study purposes, and that is OK.

In [ ]:
```
t_u = t_c * 1.8 + 32.0
t_u
```

Out[ ]:
```
array([ 93.71017416, 159.72839259,  64.68194057,  92.078013  ,
        57.16032449,  60.11353992, 140.65134482, 179.39119443,
        92.19249797, 209.82947224, 131.27263255,  36.62973003,
       209.67791903,  82.42931028,  67.8854857 , 123.47336161,
       203.41834268,  91.34073989, 174.76446639,  34.42535147,
        78.16777022,  33.08981179, 122.41801276, 191.29006739,
       189.20339667,  37.77492547, 209.82423998, 187.32144811,
        78.92002379, 163.61270063, 138.74730842,  95.85941329,
       155.24180938,  40.30984128, 135.41284625, 210.91148332,
       197.92802991,  43.27203455,  60.4212099 ,  71.2891497 ,
       146.11247297, 161.9316788 , 164.56041898, 198.57210781,
       113.53442201,  36.20756594, 104.49258822, 113.08338568,
        40.66644997, 136.53810032, 196.53733092, 115.49401433,
       138.65423904, 171.3718235 , 136.53614353,  34.60770453,
        67.13600447, 114.64380455, 181.74703801, 192.07378891,
       162.87649582, 135.62174745, 181.30124879, 177.66146261,
       198.1419401 ,  56.64559328, 133.30928174, 212.        ,
        57.46864209,  70.1894647 , 211.01986038,  99.09086267,
       191.95695824, 176.4093567 , 149.79970005,  36.96494677,
        59.77005843,  72.76620414, 126.70715839,  92.49336291,
        43.22468268, 184.58151483,  72.45276531,  75.85756718,
       171.05459773,  32.        , 139.2201322 , 112.95359509,
        66.76666374,  33.82165908, 194.59444182, 192.14017768,
       126.52518957, 179.33495868, 202.71306321, 191.96895843,
       146.60463722, 116.457274  , 101.82430374, 167.04722557])
```

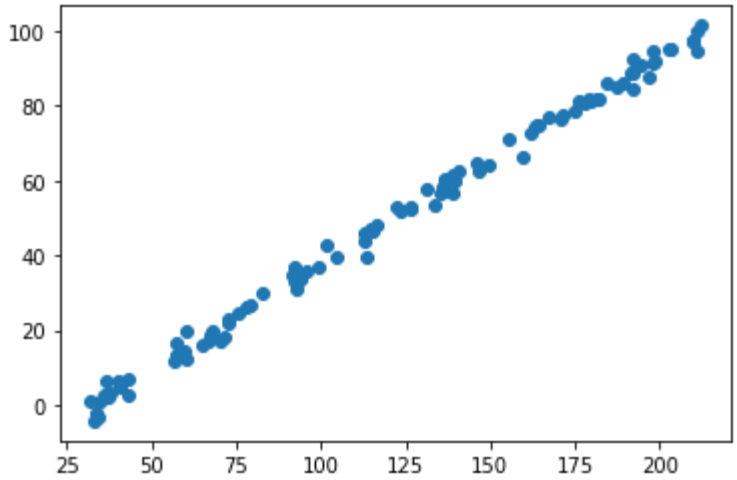
Our fake data isn't realistic until we add some noise to it. So let's add some random normally distributed noise with a standard deviation of 1 degree. Let's also make sure our fake noise distribution is roughly normal.

In [ ]:
```
noise = np.random.normal(0.0, 1.0, 100)
plt.hist(noise);
```



Now, we can add our fake noise to our celsius readings and view the effects.

In [ ]:
```
t_c += noise
plt.scatter(t_u, t_c);
```



Next, because we are using PyTorch, and PyTorch uses tensors, we must convert these numpy arrays to PyTorch tensors.

In [ ]:
```
t_u = torch.tensor(t_u)
t_c = torch.tensor(t_c)
```

Now we can actually split our larger set of fake data temperatures into train and test (validation) sets.

In [ ]:
```
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)
shuffled_indices = torch.randperm(n_samples)
train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]
train_indices, val_indices
```

Out[ ]:
```
(tensor([67, 52, 71, 48, 15, 89, 77,  2, 94, 20, 23, 69, 96, 31, 88, 74, 6
6, 22,
         30, 91, 21, 18, 43, 40,  7, 49, 79, 59, 16, 64, 83, 35, 54,  3, 4
7, 70,
         41, 14, 55, 36, 63, 27, 60,  9, 56, 10, 33,  4, 19, 51, 80,  5, 4
6, 13,
         99, 65, 11, 81, 50, 68, 93, 78, 37, 58, 85,  1, 25, 97, 82, 92, 2
8, 75,
         17, 72, 73, 42, 12, 87, 62, 34]),
 tensor([ 0, 38, 45, 39, 24,  8, 95, 29, 53,  6, 32, 84, 57, 61, 26, 76, 4
4, 90,
         86, 98]))
```

We use the above indices to separate out values for train and test (validation) tensors.

In [ ]:
```
train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]
val_t_u = t_u[val_indices]
val_t_c = t_c[val_indices]
train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u
```

Now we can setup a training loop, leveraging from the previous training loop functions, but this time we look at training losses and validation losses during the training.

In [ ]:
```
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u,
                  train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        val_t_p = model(val_t_u, *params)
        val_loss = loss_fn(val_t_p, val_t_c)

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print(f"Epoch {epoch}, Training loss {train_loss.item():.4f},"
                  f" Validation loss {val_loss.item():.4f}")

    return params
```

Now we run our train and test losses while training.

In [ ]:
```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    params = params,
    train_t_u = train_t_un,
    val_t_u = val_t_un,
    train_t_c = train_t_c,
    val_t_c = val_t_c)
```

```
Epoch 1, Training loss 1944.1927, Validation loss 3137.4920
Epoch 2, Training loss 1825.5388, Validation loss 2952.3610
Epoch 3, Training loss 1710.9032, Validation loss 2773.1303
Epoch 500, Training loss 10.0687, Validation loss 9.0932
Epoch 1000, Training loss 1.3390, Validation loss 1.3581
Epoch 1500, Training loss 1.2508, Validation loss 1.0975
Epoch 2000, Training loss 1.2507, Validation loss 1.0914
Epoch 2500, Training loss 1.2507, Validation loss 1.0914
Epoch 3000, Training loss 1.2507, Validation loss 1.0914
```

Out[ ]:
```
tensor([  5.5478, -17.7563], requires_grad=True)
```

Note that in the book, with the fewer train and validation values, the final training loss was less than the final validation loss. However, in this particular case, and having more values in each set, the validation loss came out less. This can happen occasionally. Run it multiple times from the initial randomization of values and see how often the final training losses are greater than the validation losses.

There was a slight inefficiency in our last operations. Gradients were being tracked in our validation set calculations but not used, because `backward` was not being called on those. Tracking these gradients can become a noteworthy inefficiency on much larger models. We can remedy this situation by adding a context manager. Note the block of code that begins with the `with torch.no_grad():` content manager. This causes all the lines indented under it to be run without gradient tracking. Note also that right after that context managed code block, we are careful to zero out the gradients before updating the parameters as we discussed previously. We are still not ready to let them accumulate.

In [ ]:
```
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u,
                  train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        with torch.no_grad():
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print(f"Epoch {epoch}, Training loss {train_loss.item():.4f},"
                  f" Validation loss {val_loss.item():.4f}")

    return params
```

The code below shows yet another way to decide whether or not gradients are included in calculations. We could use this in place of the context manager block in the latest training loop function.

In [ ]:
```
def calc_forward(t_u, t_c, is_train):
    with torch.set_grad_enabled(is_train):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

    return loss
```

Summary

This is only a start. We will continue to discover many more elegant advantages to PyTorch as we move forward. I hope you will continue to journey with me.