UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

Introduction to ML
Lecture 14: Artificial Neural Networks

Hamed Tabkhi

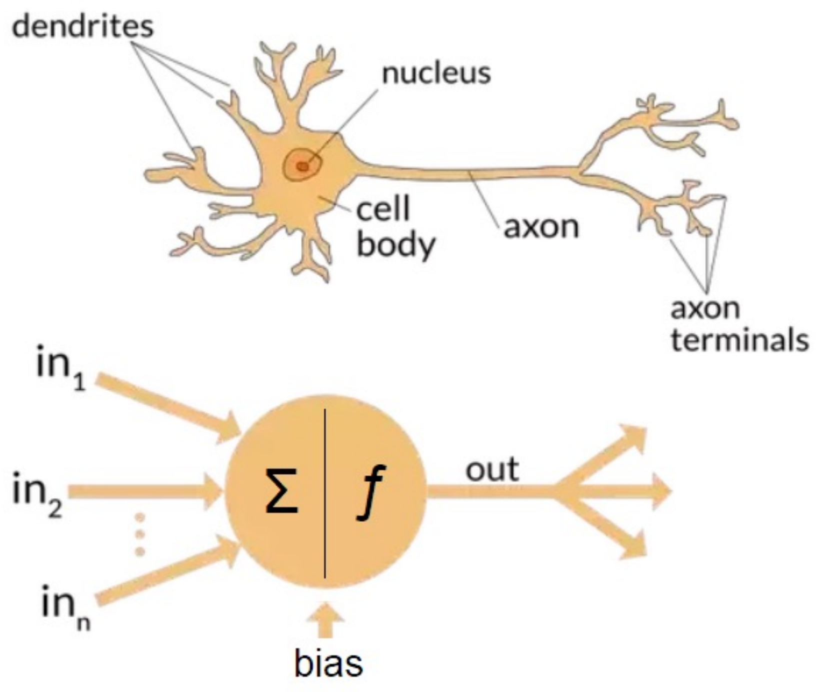
Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

Neurons

- The initial models were inspired by neuroscience
- It seems likely that both artificial and physiological neural networks use vaguely similar mathematical strategies for approximating complicated functions because that family of strategies works very effectively
- Modern artificial neural networks bear only a slight resemblance to the mechanisms of neurons
- the simplest unit in (deep) neural networks is a linear operation (scaling + offset) followed by an activation function



Mathematical Expression of Neuron

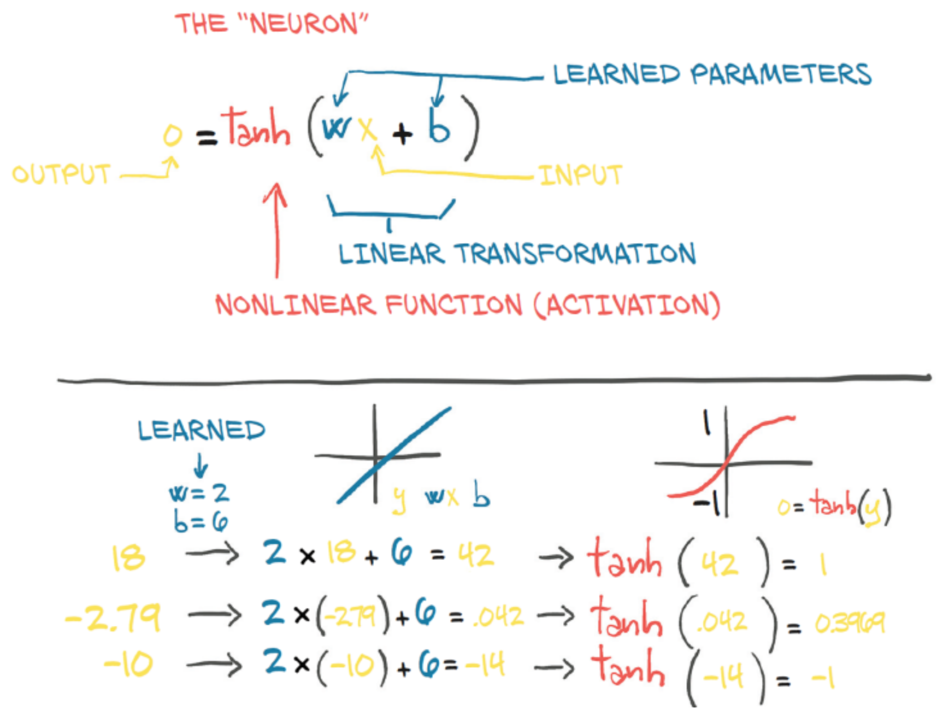
- At its core, it is nothing but a linear transformation of the input.
- Neuron multiplies the input by a number [the *weight*] and adding a constant [the *bias*]

- It then followed by the application of a fixed nonlinear function - referred to as the *activation function*

- Mathematically, we can write this out as

$$o = f(w * x + b)$$

with *x* as our input, *w* our weight or scaling factor, and *b* as our bias or offset. *f* is our activation function, set to the hyperbolic tangent, or tanh function here.



THE "NEURON"
LEARNED PARAMETERS
o = tanh (w x + b)
OUTPUT
INPUT
LINEAR TRANSFORMATION
NONLINEAR FUNCTION (ACTIVATION)
LEARNED
w = 2
b = 6
y wx b
1
-1
o = tanh(y)
18 → 2 × 18 + 6 = 42 → tanh (42) = 1
-2.79 → 2 × (-2.79) + 6 = .042 → tanh (.042) = 0.3969
-10 → 2 × (-10) + 6 = -14 → tanh (-14) = -1

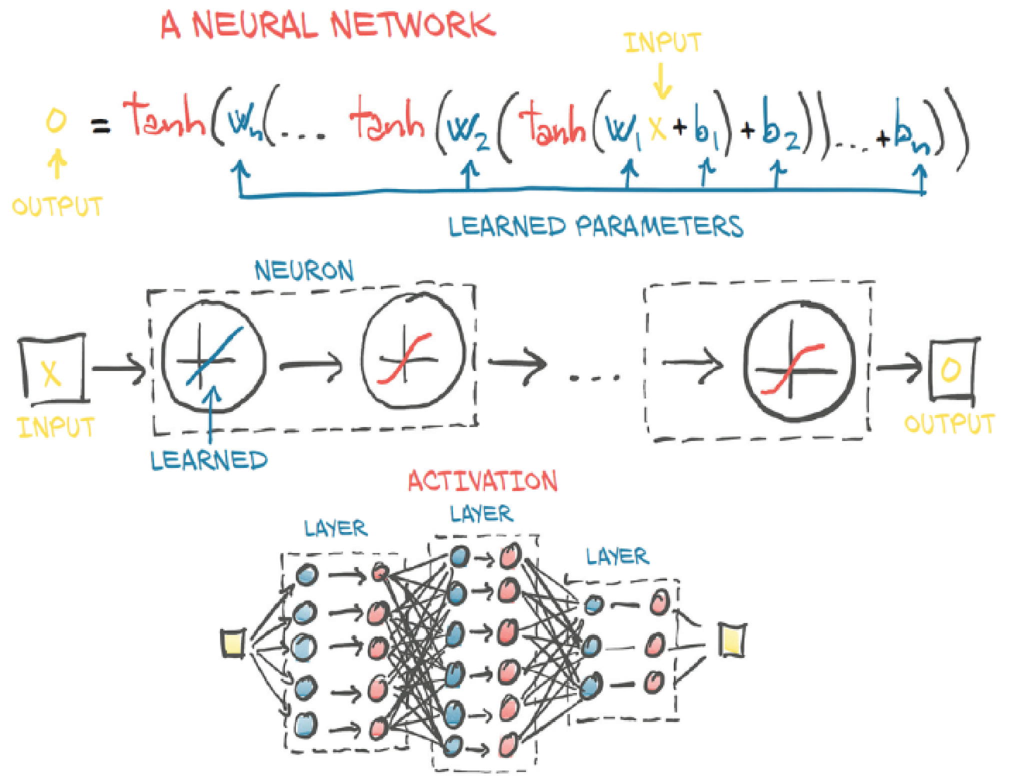
The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

layer of neurons

It represents many neurons via the multidimensional weights and biases.

```
x_1 = f(w_0 * x + b_0)
x_2 = f(w_1 * x_1 + b_1)
...
y = f(w_n * x_n + b_n)
```

- The output of a layer of neurons is used as an input for the following layer.
- Remember that `w_0` here is a matrix, and `x` is a vector!
- Using a vector allows `w_0` to hold an entire *layer* of neurons, not just a single weight



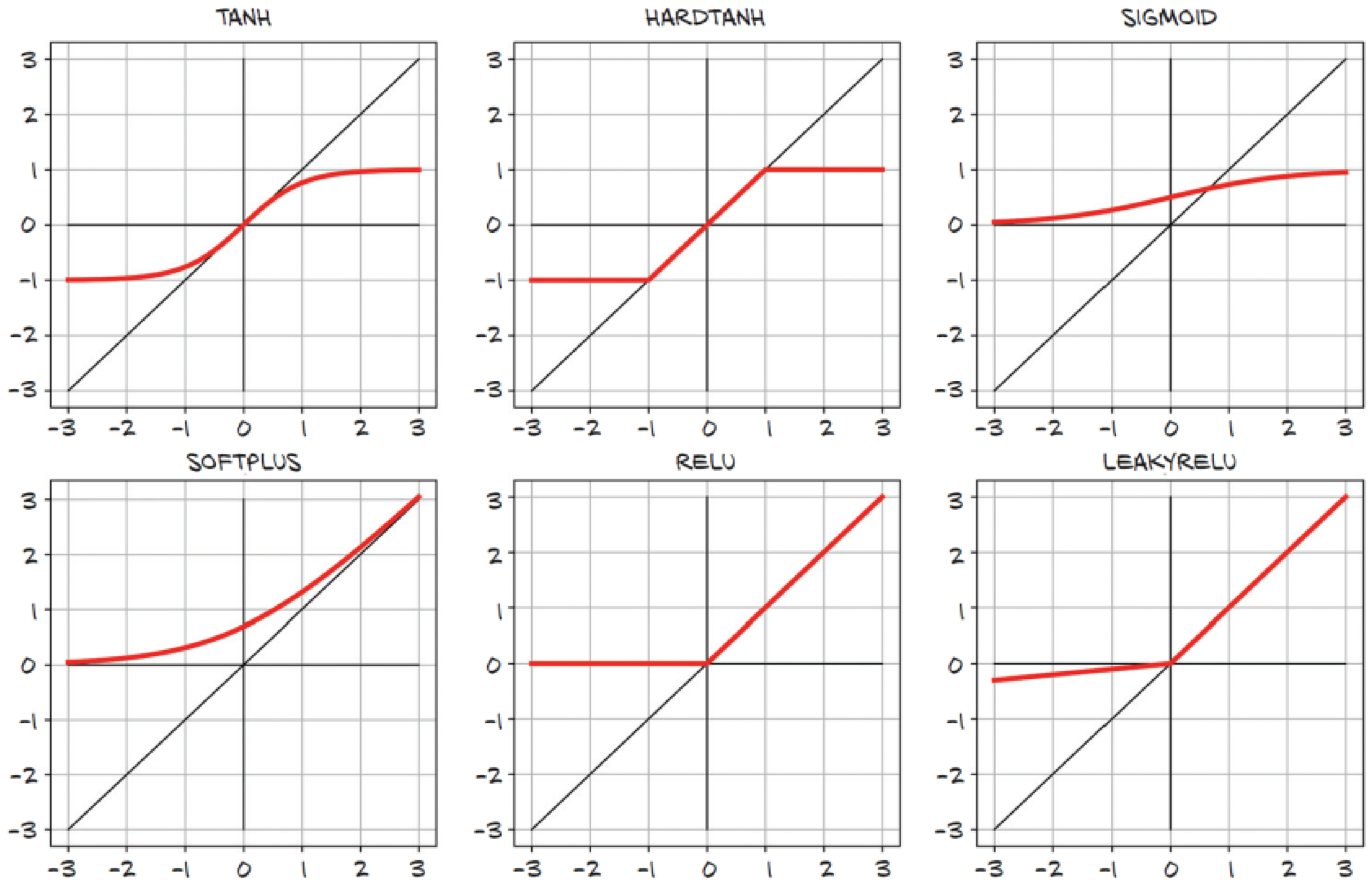
The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Activation Function

- Activation functions are nonlinear. Repeated applications of (`w*x+b`) without an activation function results in a function of the same (affine linear) form.
- The nonlinearity allows the overall network to approximate more complex functions.
- Without these characteristics, the network either falls back to being a linear model or becomes difficult to train.
- The ability of an ensemble of neurons to approximate a very wide range of useful functions depends on the combination of the linear and nonlinear behavior inherent to each neuron.

- **Joining many linear + activation units in parallel and stacking them one after the other leads us to a mathematical object that is capable of approximating complicated functions.**

Different Activation Functions



The most popular activation functions

- **`ReLU`** (for *rectified linear unit*) deserves special note, as it is currently considered one of the best-performing general activation functions; many state-of-the-art results have used it.
- The **`Sigmoid`** activation function, also known as the *logistic function*, was widely used in early deep learning work but has since fallen out of common use except where we explicitly want to move to the 0…1 range: for example, when the output should be a probability.
- **`LeakyReLU`** function modifies the standard `ReLU` to have a small positive slope, rather than being strictly zero for negative inputs (typically this slope is 0.01, but it’s shown here with slope 0.1 for clarity).

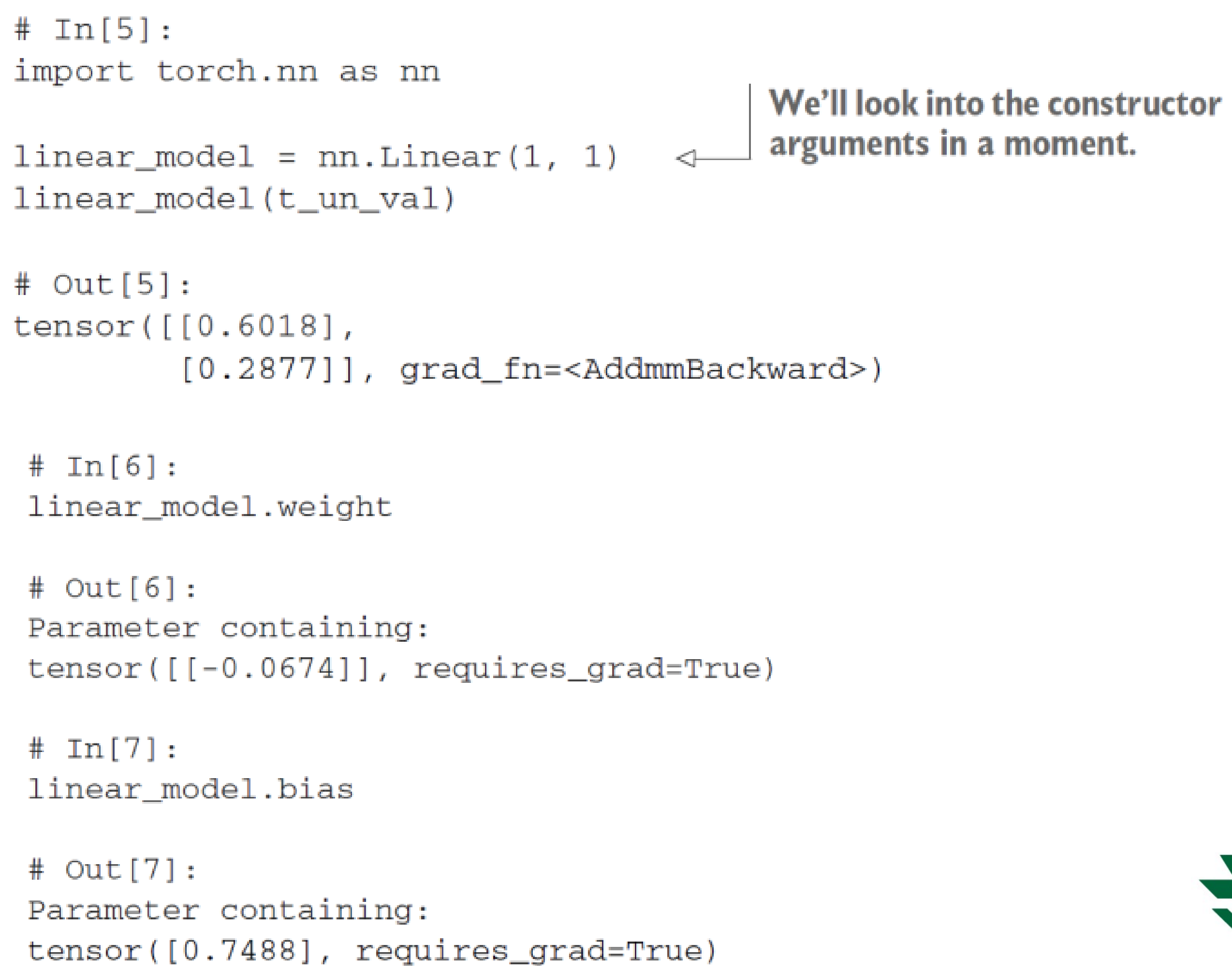
What learning means for a neural network

- Building models out of stacks of linear transformations followed by differentiable activations leads to models that can approximate highly nonlinear processes
- Deep neural networks give us the ability to approximate highly nonlinear phenomena without having an explicit model for them.

- **What makes using deep neural networks so attractive is that it saves us from worrying too much about the exact function that represents our data—whether it is quadratic, piecewise polynomial, or something else.**
  - With a deep neural network model, we have a universal approximator and a method to estimate its parameters.
  - This approximator can be customized to our needs, in terms of model capacity and its ability to model complicated input/output relationships, just by composing simple building blocks.

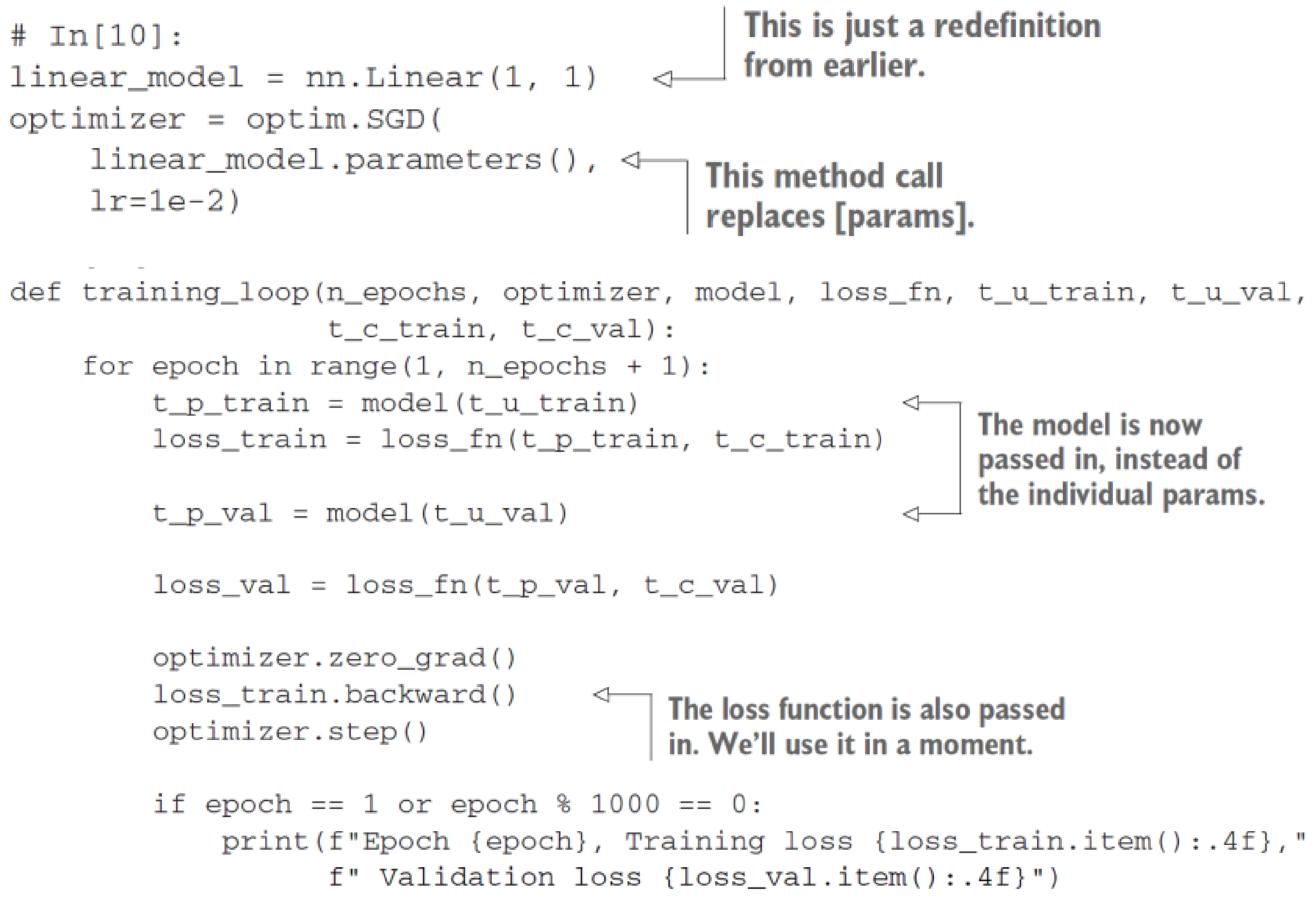
PyTorch nn Module

- PyTorch has a whole submodule dedicated to neural networks, called `torch.nn`.
- It contains the building blocks (parlance / *layers)* needed to create all sorts of neural network architectures.



```
# In[5]:
import torch.nn as nn
                                            We'll look into the constructor
linear_model = nn.Linear(1, 1)              arguments in a moment.
linear_model(t_un_val)

# Out[5]:
tensor([[0.6018],
        [0.2877]], grad_fn=<AddmmBackward>)

# In[6]:
linear_model.weight

# Out[6]:
Parameter containing:
tensor([[-0.0674]], requires_grad=True)

# In[7]:
linear_model.bias

# Out[7]:
Parameter containing:
tensor([0.7488], requires_grad=True)
```

Rebuilding our Linear Model with nn Module



```
# In[10]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(
    linear_model.parameters(),
    lr=1e-2)

def training_loop(n_epochs, optimizer, model, loss_fn, t_u_train, t_u_val,
                  t_c_train, t_c_val):
    for epoch in range(1, n_epochs + 1):
        t_p_train = model(t_u_train)
        loss_train = loss_fn(t_p_train, t_c_train)

        t_p_val = model(t_u_val)

        loss_val = loss_fn(t_p_val, t_c_val)

        optimizer.zero_grad()
        loss_train.backward()
        optimizer.step()

        if epoch == 1 or epoch % 1000 == 0:
            print(f"Epoch {epoch}, Training loss {loss_train.item():.4f},"
                  f" Validation loss {loss_val.item():.4f}")
```

This is just a redefinition from earlier.

This method call replaces [params].

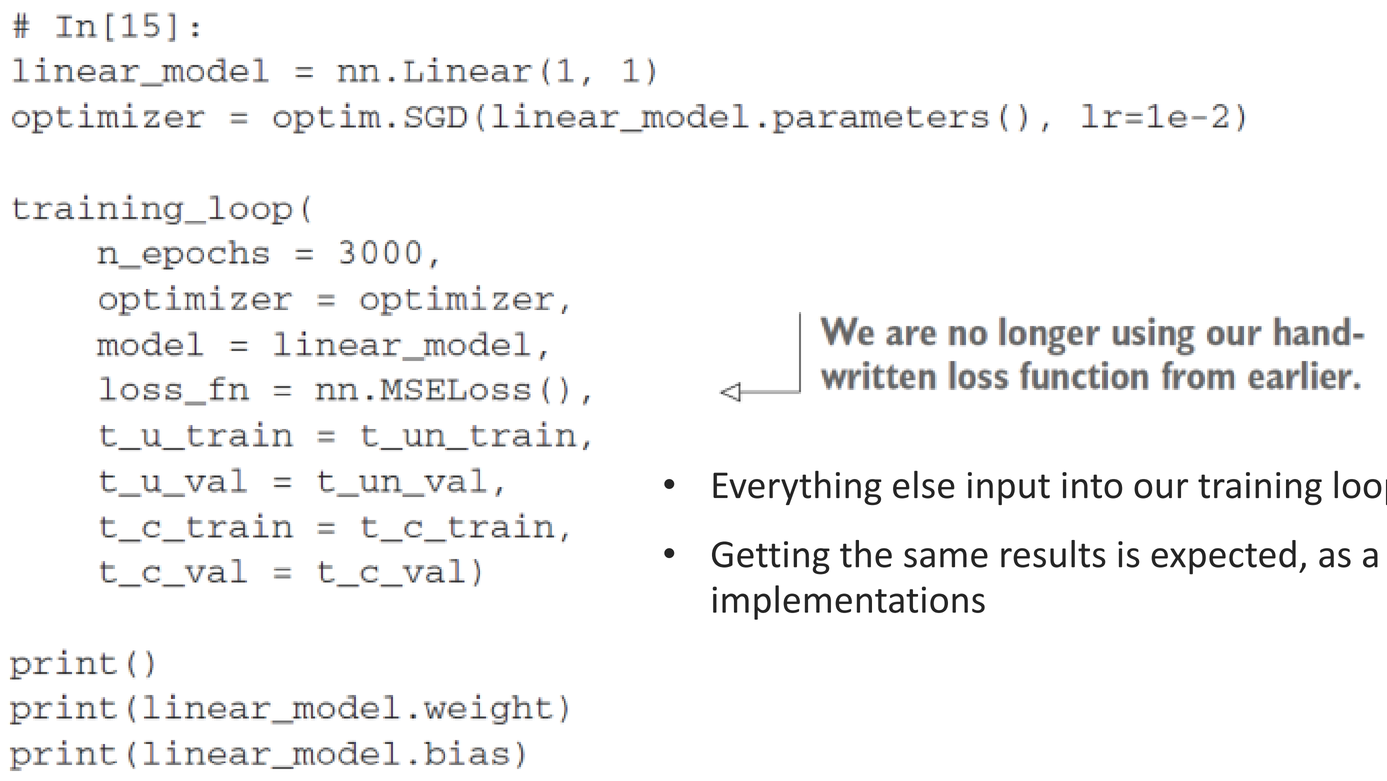
The model is now passed in, instead of the individual params.

The loss function is also passed in. We'll use it in a moment.

It hasn't changed practically at all, except that now we don't pass `params` explicitly to `model` since the model itself holds its `Parameters` internally.

Rebuilding our Linea Model with nn Module

- `nn` comes with several common loss functions, among them `nn.MSELoss` (MSE stands for Mean Square Error), which is exactly what we defined earlier as our `loss_fn`.
- Loss functions in `nn` are still subclasses of `nn.Module`
- In our case, we get rid of the handwritten `loss_fn` and replace it:

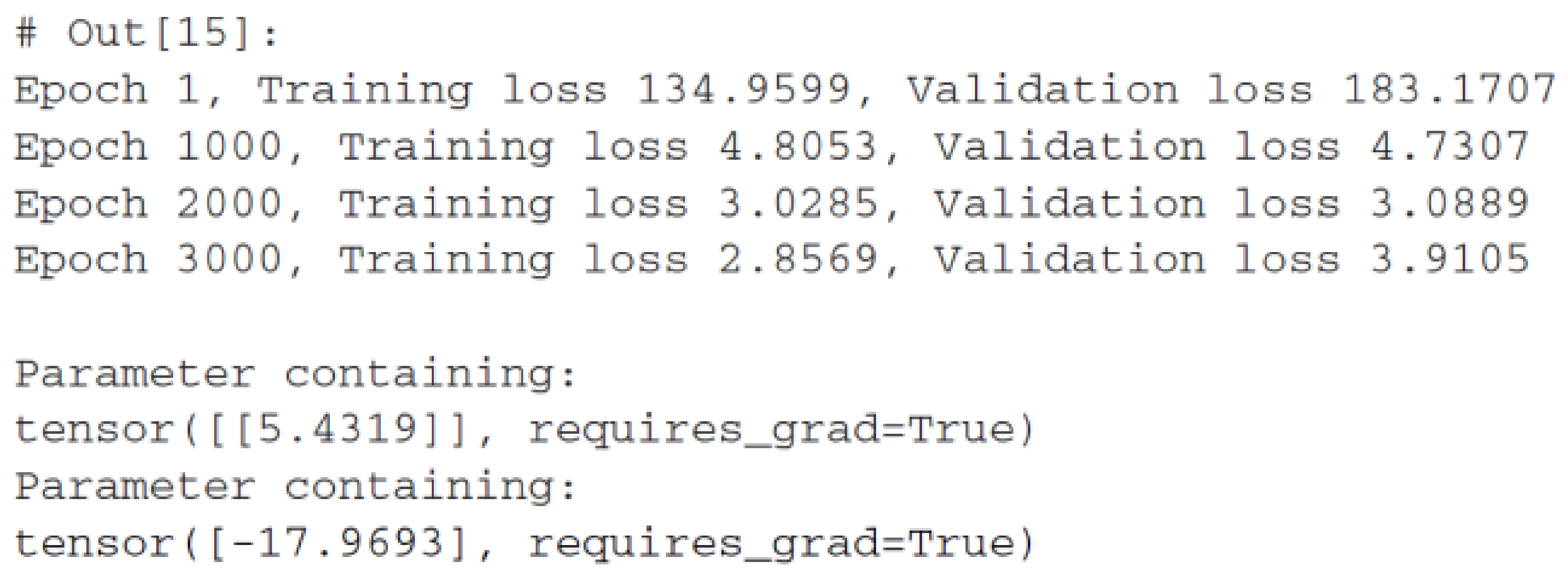


```
# In[15]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(linear_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    model = linear_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print()
print(linear_model.weight)
print(linear_model.bias)
```

We are no longer using our hand-written loss function from earlier.

- Everything else input into our training loop stays the same. Even our results remain the same as before
- Getting the same results is expected, as a difference would imply a bug in one of the two implementations

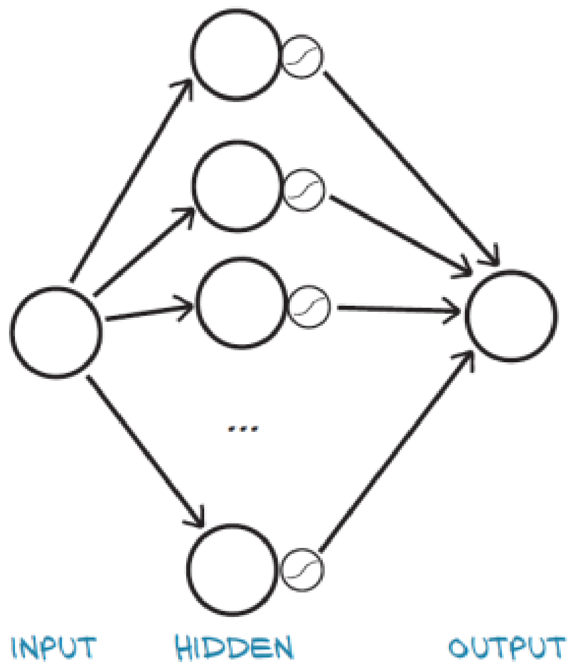
Output should be the same as Linear Regression We did before

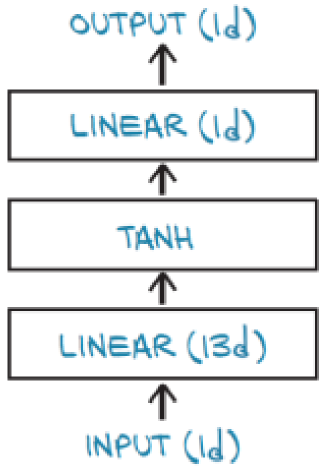


```
# Out[15]:
Epoch 1, Training loss 134.9599, Validation loss 183.1707
Epoch 1000, Training loss 4.8053, Validation loss 4.7307
Epoch 2000, Training loss 3.0285, Validation loss 3.0889
Epoch 3000, Training loss 2.8569, Validation loss 3.9105

Parameter containing:
tensor([[5.4319]], requires_grad=True)
Parameter containing:
tensor([-17.9693], requires_grad=True)
```

Hidden Layer

- Let's build the simplest possible neural network: a linear module, followed by an activation function, feeding into another linear module.
- The **linear + activation layer** is commonly referred to as a ***hidden* layer**, since its outputs are not observed directly but fed into the output layer.
- While the input and output of the model are both of size 1, the size of the output of the first linear module (hidden layer) is usually larger than 1.
- This can lead different units to respond to different ranges of the input, which increases the capacity of our model.

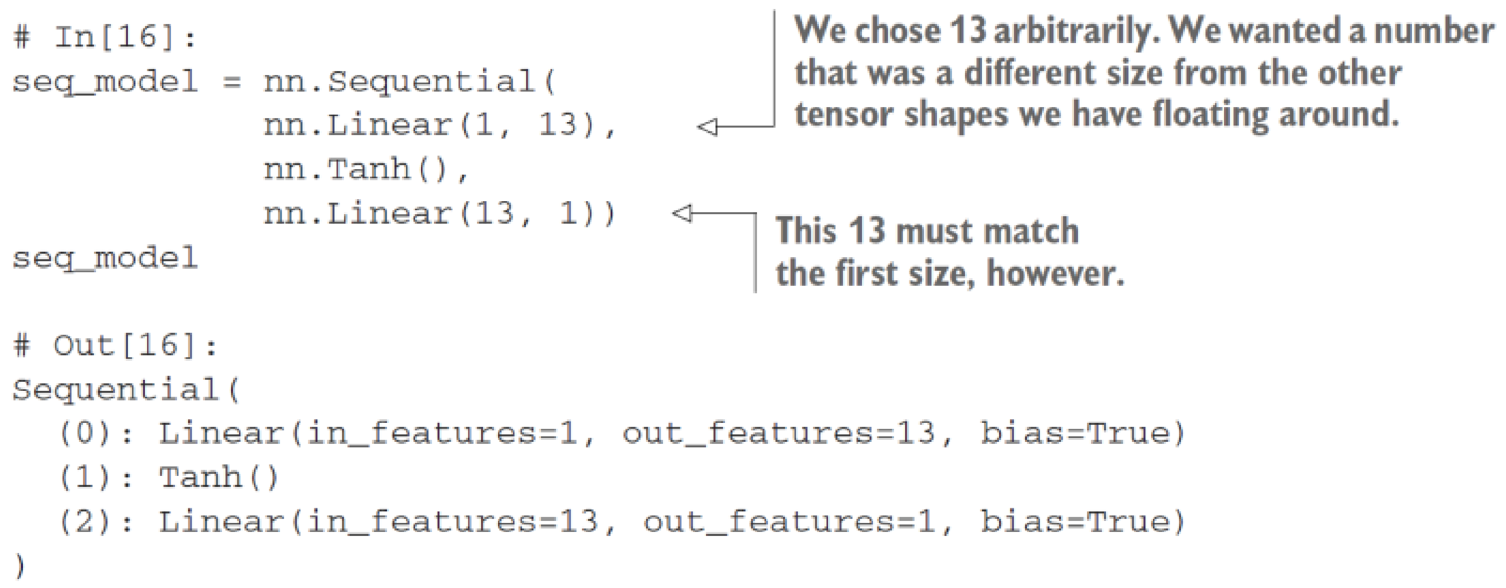




The WILLIAM STATES LEE COLLEGE of ENGINEERING
UNC CHARLOTTE

Our First Real Neural Network

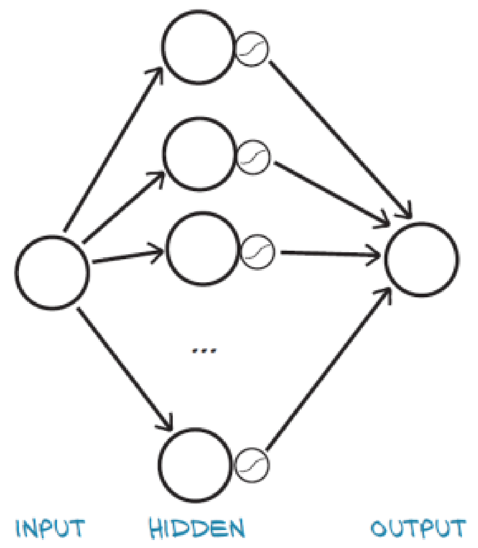
- `nn.Sequential` provides a simple way to concatenate modules.
- 1 input feature to 13 hidden features, passes them through a `tanh` activation, and linearly combines the resulting 13 numbers into 1 output feature.

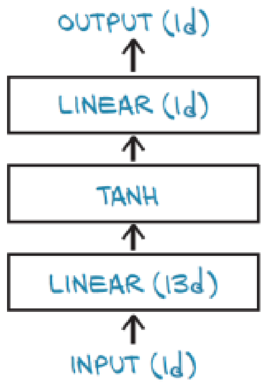


```
# In[16]:
seq_model = nn.Sequential(
            nn.Linear(1, 13),
            nn.Tanh(),
            nn.Linear(13, 1))
seq_model

# Out[16]:
Sequential(
  (0): Linear(in_features=1, out_features=13, bias=True)
  (1): Tanh()
  (2): Linear(in_features=13, out_features=1, bias=True)
)
```

We chose 13 arbitrarily. We wanted a number that was a different size from the other tensor shapes we have floating around.

This 13 must match the first size, however.





- The end result is a model that takes the inputs expected by the first module specified as an argument of `nn.Sequential`, passes intermediate outputs to subsequent modules, and produces the output returned by the last module.

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Inspecting the Parameters

- Calling `model.parameters()` will collect `weight` and `bias` from both the first and second linear modules

```
# In[17]:
[param.shape for param in seq_model.parameters()]

# Out[17]:
[torch.Size([13, 1]), torch.Size([13]), torch.Size([1, 13]), torch.Size([1])]
```

```
# In[18]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[18]:
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

- These are the tensors that the optimizer will get.
- After we call `model.backward()`, all parameters are populated with their `grad`, and the optimizer then updates their values accordingly during the `optimizer.step()` call.

Inspecting the Parameters

`Sequential` also accepts an `OrderedDict`, in which we can name each module passed to `Sequential`

```
# In[19]:
from collections import OrderedDict

seq_model = nn.Sequential(OrderedDict([
    ('hidden_linear', nn.Linear(1, 8)),
    ('hidden_activation', nn.Tanh()),
    ('output_linear', nn.Linear(8, 1))
]))

seq_model

# Out[19]:
Sequential(
  (hidden_linear): Linear(in_features=1, out_features=8, bias=True)
  (hidden_activation): Tanh()
  (output_linear): Linear(in_features=8, out_features=1, bias=True)
)
```

Inspecting the Parameters

- We can get more explanatory names for submodules:

```
# In[20]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[20]:
hidden_linear.weight torch.Size([8, 1])
hidden_linear.bias torch.Size([8])
output_linear.weight torch.Size([1, 8])
output_linear.bias torch.Size([1])
```

- We can also access a particular `Parameter` by using submodules as attributes

```
# In[21]:
seq_model.output_linear.bias

# Out[21]:
Parameter containing:
tensor([-0.0173], requires_grad=True)
```

- This is useful for inspecting parameters or their gradients: for instance, to monitor gradients during training, as we did at the beginning of this chapter.

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Running our First Real Neural Network

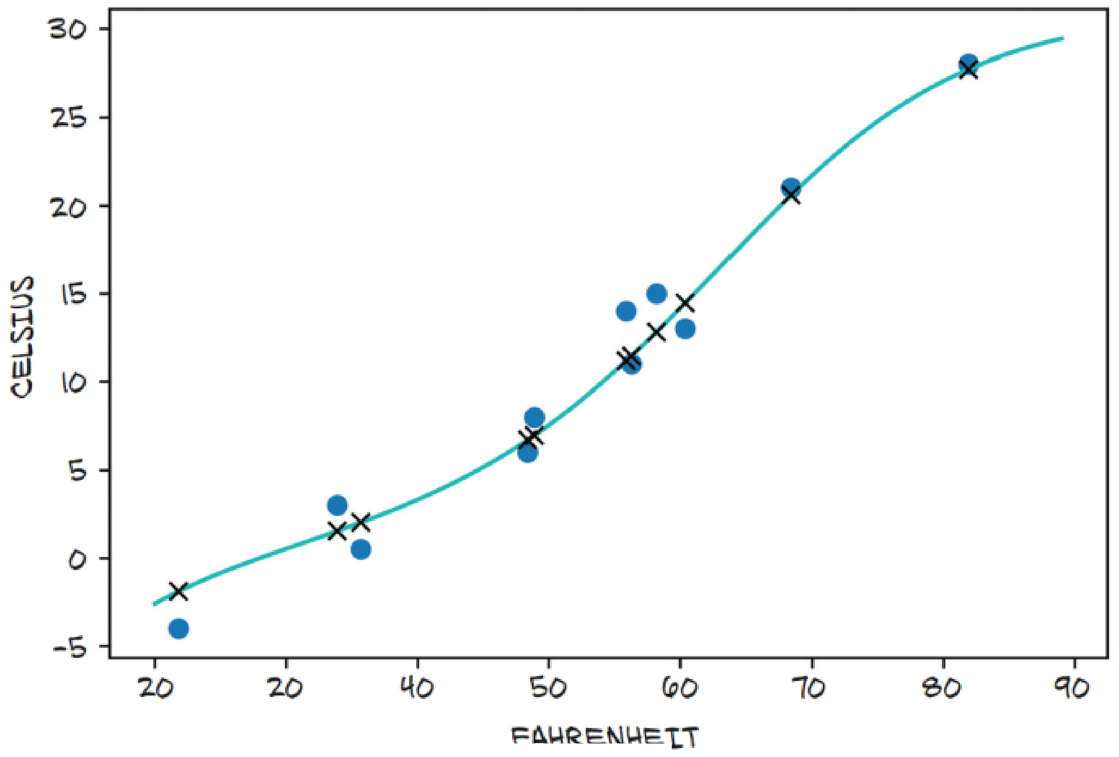
```
# In[22]:
optimizer = optim.SGD(seq_model.parameters(), lr=1e-3)

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    model = seq_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print('output', seq_model(t_un_val))
print('answer', t_c_val)
print('hidden', seq_model.hidden_linear.weight.grad)
```

We've dropped the learning rate a bit to help with stability.

```
# Out[22]:
Epoch 1, Training loss 182.9724, Validation loss 231.8708
Epoch 1000, Training loss 6.6642, Validation loss 3.7330
Epoch 2000, Training loss 5.1502, Validation loss 0.1406
Epoch 3000, Training loss 2.9653, Validation loss 1.0005
Epoch 4000, Training loss 2.2839, Validation loss 1.6580
Epoch 5000, Training loss 2.1141, Validation loss 2.0215
output tensor([[-1.9930],
        [20.8729]], grad_fn=<AddmmBackward>)
answer tensor([[-4.],
        [21.]])
hidden tensor([[ 0.0272],
        [ 0.0139],
        [ 0.1692],
        [ 0.1735],
        [-0.1697],
        [ 0.1455],
        [-0.0136],
        [-0.0554]])
```

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE

Plotting the Results



```
# In[23]:
from matplotlib import pyplot as plt

t_range = torch.arange(20., 90.).unsqueeze(1)

fig = plt.figure(dpi=600)
```

```
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
plt.plot(t_range.numpy(), seq_model(0.1 * t_range).detach().numpy(), 'c-')
plt.plot(t_u.numpy(), seq_model(0.1 * t_u).detach().numpy(), 'kx')
```

Congratulations!

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING
UNC CHARLOTTE