# Lesson 4: Introduction to PyTorch

## Welcome!

Welcome! In this lesson, you'll learn how to use PyTorch for building deep learning models. PyTorch was released in early 2017 and has been making a pretty big impact in the deep learning community. It's developed as an open source project by the Facebook AI Research team, but is being adopted by teams everywhere in industry and academia. In my experience, it's the best framework for learning deep learning and just a delight to work with in general. By the end of this lesson, you'll have trained your own deep learning model that can classify images of cats and dogs.

I'll first give you a basic introduction to PyTorch, where we'll cover tensors - the main data structure of PyTorch. I'll show you how to create tensors, how to do simple operations, and how tensors interact with NumPy.

Then you'll learn about a module called autograd that PyTorch uses to calculate gradients for training neural networks. Autograd, in my opinion, is amazing. It does all the work of backpropagation for you by calculating the gradients at each operation in the network which you can then use to update the network weights.

Next you'll use PyTorch to build a network and run data forward through it. After that, you'll define a loss and an optimization method to train the neural network on a dataset of handwritten digits. You'll also learn how to test that your network is able to generalize through validation.

However, you'll find that your network doesn't work too well with more complex images. You'll learn how to use pre-trained networks to improve the performance of your classifier, a technique known as transfer learning.

Follow along with the videos and work through the exercises in your own notebooks. If you get stuck, check out my solution videos and notebooks.

## Get the notebooks

The notebooks for this lesson will be provided in the classroom, but if you wish to follow along on your local machine, then the instructions below will help you get setup and ready to learn!

All the notebooks for this lesson are available from our deep learning repo on GitHub. Please clone the repo by typing

```
git clone https://github.com/udacity/deep-learning-v2-pytorch.git
```

in your terminal. Then navigate to the intro-to-pytorch directory in the repo.

Follow along in your notebooks to complete the exercises. I'll also be providing solutions to the exercises, both in videos and in the notebooks marked (Solution).

**Dependencies**

These notebooks require PyTorch v0.4 or newer, and torchvision. The easiest way to install PyTorch and torchvision locally is by following the instructions on the PyTorch site. Choose the stable version, your appropriate OS and Python versions, and how you'd like to install it. You'll also need to install numpy and jupyter notebooks, the newest versions of these should work fine. Using the conda package manager is generally best for this,

conda install numpy jupyter notebook

If you haven't used conda before, please read the documentation to learn how to create environments and install packages. I suggest installing Miniconda instead of the whole Anaconda distribution. The normal package manager pip also works well. If you have a preference, go with that.

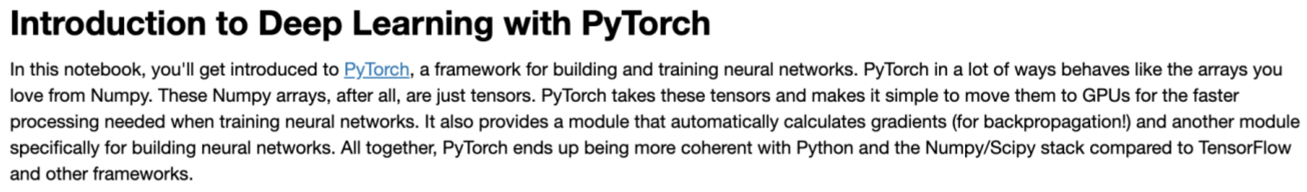
The final part of the series has a soft requirement of a GPU used to accelerate network computations. Even if you don't have a GPU available, you'll still be able to run the code and finish the exercises. PyTorch uses a library called CUDA to accelerate operations using the GPU. If you have a GPU that CUDA supports, you'll be able to install all the necessary libraries by installing PyTorch with conda. If you can't use a local GPU, you can use cloud platforms such as AWS, GCP, and FloydHub to train your networks on a GPU.

Our Nanodegree programs also provide GPU workspaces in the classroom, as well as credits for AWS.

**Feedback**

If you have problems with the notebooks, please contact support or create an issue on the repo. We're also happy to incorporate your improvements through pull requests.

Notebook-folder-intro-to-pytorch-begins



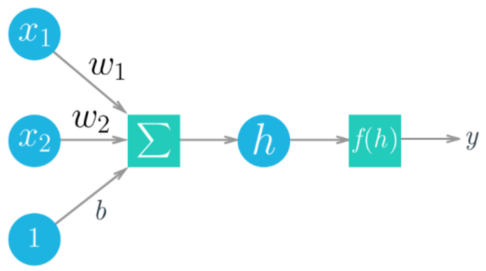
## Introduction to Deep Learning with PyTorch

In this notebook, you'll get introduced to PyTorch, a framework for building and training neural networks. PyTorch in a lot of ways behaves like the arrays you love from Numpy. These Numpy arrays, after all, are just tensors. PyTorch takes these tensors and makes it simple to move them to GPUs for the faster processing needed when training neural networks. It also provides a module that automatically calculates gradients (for backpropagation!) and another module specifically for building neural networks. All together, PyTorch ends up being more coherent with Python and the Numpy/Scipy stack compared to TensorFlow and other frameworks.

73. Hello everyone and welcome to this lesson on deep learning with PyTorch. So, in this lesson I'm going to be showing you how we can build neural networks with pyTorch and train them. By working through all these notebooks I built, you'll be writing the actual code yourself for building these networks. By the end of the lesson, you will have built your own state of the art image classifier.

## Neural Networks

Deep Learning is based on artificial neural networks which have been around in some form since the late 1950s. The networks are built from individual parts approximating neurons, typically called units or simply "neurons." Each unit has some number of weighted inputs. These weighted inputs are summed together (a linear combination) then passed through an activation function to get the unit's output.



Mathematically this looks like:

$$y = f(w_1x_1 + w_2x_2 + b)$$
$$y = f\left(\sum_i w_ix_i + b\right)$$

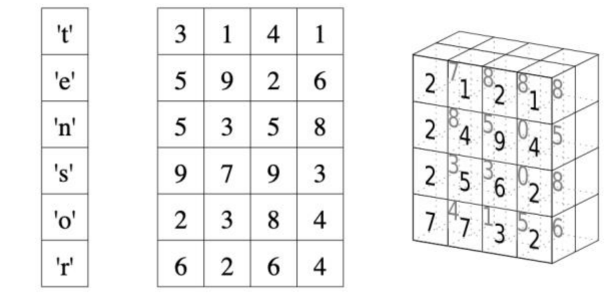
With vectors this is the dot/inner product of two vectors:

$$h = \begin{bmatrix} x_1 \, x_2 \cdots x_n \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

But first we're going to start with basics, so how do you build just a simple neural network in pyTorch? So, as a reminder of how neural networks work, in general we have some input values so here x1, x2, and we multiply them by some weights w and bias. So, this b is this bias we just multiply it by one then you sum all these up and you get some value h. Then we have what's called an activation function. So, here f of h and passing these input values h through this activation function gets you output y. This is the basis of neural networks. You have these inputs, you multiply it by some waves, take the sum, pass it through some activation function and you get an output. You can stack these up so that the output of these units, of these neurons go to another layer like another set of weights. So, mathematically this what it looks like, y, our output is equal to this linear combination of the weights and the input values w's and x's plus your bias value b passes through your activation function f and you get y. You could also write it with this sum. So, sum of wi times xi and plus b, your bias term. That gives you y. So, what's nice about this is that you can actually think of the x's, your input features, your values, as a vector, and your weights as another vector. So, your multiplication and sum is the same as a dot or inner product of two vectors. So, if you consider your input as a vector and your weights as a vector, if you take the dot product of these two, then you get your value h and then you pass h through your activation function and that gets you your output y.

## Tensors

It turns out neural network computations are just a bunch of linear algebra operations on *tensors*, a generalization of matrices. A vector is a 1-dimensional tensor, a matrix is a 2-dimensional tensor, an array with three indices is a 3-dimensional tensor (RGB color images for example). The fundamental data structure for neural networks are tensors and PyTorch (as well as pretty much every other deep learning framework) is built around tensors.



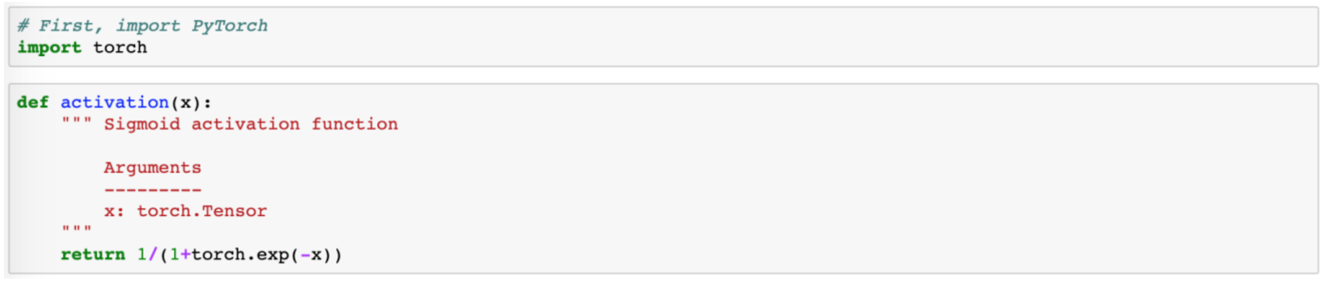
tensor of dimensions [6]
(vector of dimension 6)

tensor of dimensions [6,4]
(matrix 6 by 4)

tensor of dimensions [4,4,2]

With the basics covered, it's time to explore how we can use PyTorch to build a simple neural network.

So, now if we start thinking of our weights and our input values as vectors, so vectors are an instance of a tensor. So, a tensor is just a generalization of vectors and matrices. So, when you have these like regular structured arrangements of values and so a tensor with only one dimension is a vector. So, we just have this single one-dimensional array of values. So, in this case characters T-E-N-S-O-R. A matrix like this is a two-dimensional tensor and so we have values going in two directions from left to right and from top to bottom and so that we have individual rows and columns. So, you can do operations across the columns like along a row or you can do it across the rows like going down a column. You also have three-dimensional tensors so you can think of an image like an RGB color image as a three-dimensional tensor. So, for every pixel, there's some value for all the red and the green and the blue channels and so for every individual pixel, in a two-dimensional image, you have three values. So, that is a three-dimensional tensor. Like I said before, tensors are a generalization of this so you can actually have four-dimensional, five-dimensional, six-dimensional, and so on like tensors. It's just the ones that we normally work with are one and two-dimensional, three-dimensional tensors. So, these tensors are the base data structure that you use an pyTorch and other neural network frameworks. So, TensorFlow is named after tensors. So, these are the base data structures that you'll be using so you pretty much need to understand them really well to be able to use pretty much any framework that you'll be using for deep learning.

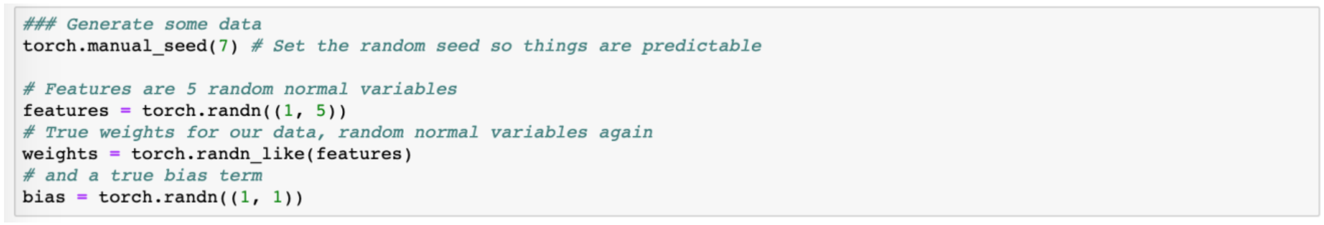


```python
# First, import PyTorch
import torch
```

```python
def activation(x):
    """ Sigmoid activation function

        Arguments
        ---------
        x: torch.Tensor
    """
    return 1/(1+torch.exp(-x))
```

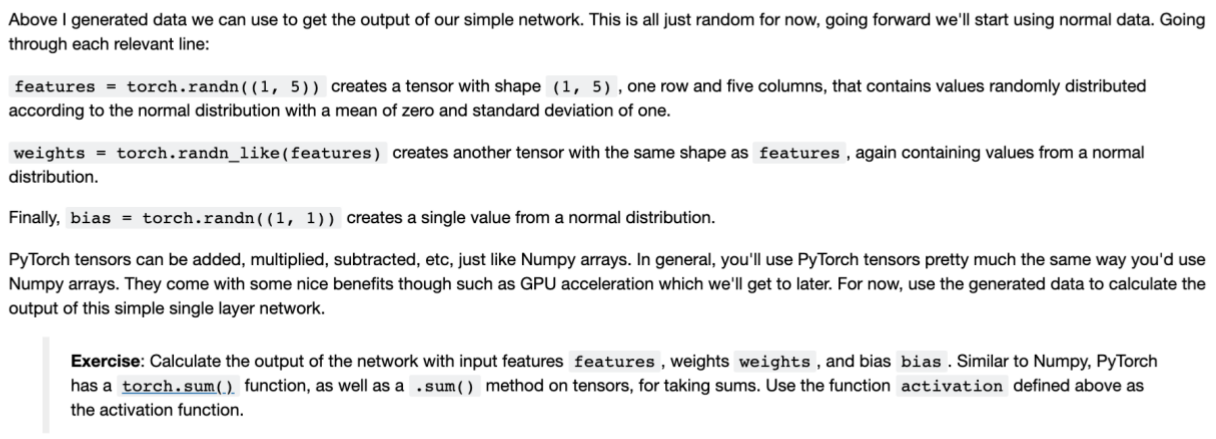
So, let's get started. I'm going to show you how to actually create some tensors and use them to build a simple neural network. So, first we're going to import pyTorch and so just import torch here. Here I am creating activation function, so this is the Sigmoid activation function. It's the nice s shape that kind of squeezes the input values between zero and one. It's really useful for providing a probability. So, probabilities are these values that can only be between zero and one. So, you're Sigmoid activation if you want the output of your neural network to be a probability, then the sigmoid activation is what you want to use.



```python
### Generate some data
torch.manual_seed(7) # Set the random seed so things are predictable

# Features are 5 random normal variables
features = torch.randn((1, 5))
# True weights for our data, random normal variables again
weights = torch.randn_like(features)
# and a true bias term
bias = torch.randn((1, 1))
```

So, here I'm going to create some fake data, I'm generating some data, I'm generating some weights and biases and with these you're actually going to do the computations to get the output of a simple neural network. So, here I'm just creating a manual seeds. So, I'm setting the seed for the random number generation that I'll be using and here I'm creating features. So, features are like the input features of the input data for your network. Here we see torch.randn. So, randn is going to create a tensor of normal variables. So, random normal variables as samples from a normal distribution. You give it a tuple of the size that you want. So, in this case I want the features to be a matrix, a 2-dimensional tensor of one row and five columns. So, you can think of this as a row vector that has five elements. For the weights, we're going to create another matrix of random normal variables and this time I'm using randn_like. So, what this does is it takes another tensor and it looks at the shape of this tensor and then it creates it, it creates another tensor with the same shape. So, that's what this this like means. So, I'm going to create a tensor of random normal variables with the same shape as features. So, it gives me my weights. Then I'm going to create a bias term. So, this is again just a random normal variable. Now I'm just creating one value. So, this is one row and one column. Here I'm going to leave this exercise up to you. So, what you're going to be doing is taking the features, weights, and the bias tensors and you're going to calculate the output of this simple neural network. So, remember with features and weights you want to take the inner product or you want to multiply the features by the weights and sum them up and then add the bias and then pass it through the activation function and from that you should get the output of your network. So, if you want to see how I did this, checkout my solution notebook or watch the next video which I'll show you my solution for this exercise.


Above I generated data we can use to get the output of our simple network. This is all just random for now, going forward we'll start using normal data. Going through each relevant line:

`features = torch.randn((1, 5))` creates a tensor with shape `(1, 5)`, one row and five columns, that contains values randomly distributed according to the normal distribution with a mean of zero and standard deviation of one.

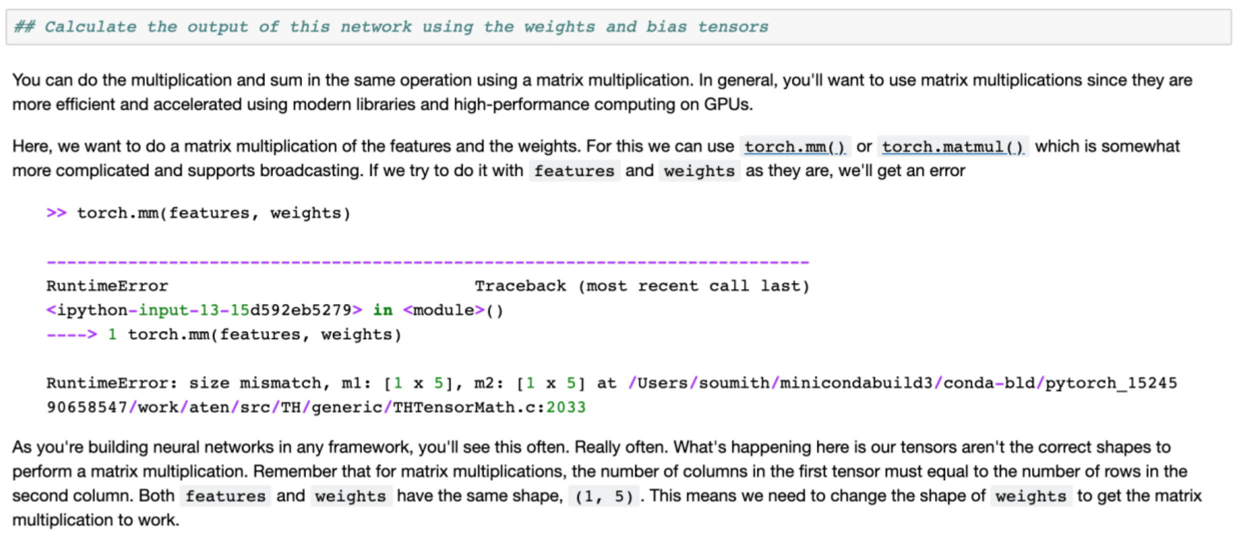
`weights = torch.randn_like(features)` creates another tensor with the same shape as `features`, again containing values from a normal distribution.

Finally, `bias = torch.randn((1, 1))` creates a single value from a normal distribution.

PyTorch tensors can be added, multiplied, subtracted, etc, just like Numpy arrays. In general, you'll use PyTorch tensors pretty much the same way you'd use Numpy arrays. They come with some nice benefits though such as GPU acceleration which we'll get to later. For now, use the generated data to calculate the output of this simple single layer network.

> **Exercise**: Calculate the output of the network with input features `features`, weights `weights`, and bias `bias`. Similar to Numpy, PyTorch has a `torch.sum()` function, as well as a `.sum()` method on tensors, for taking sums. Use the function `activation` defined above as the activation function.

74. So now, this is my solution for this exercise on calculating the output of this small simple neural network. So, remember that what we want to do is multiply our features by our weights, so features times weights. So these tensors, they work basically the same as NumPy arrays, if you've used NumPy before. So, when you multiply features times weights, it'll just take the first element from each one, multiply them together, take the second element and multiply them together and so on and give you back a new tensor, where there's element by element multiplication. So, from that we can do torch.sum to sum it all up into one value, add our bias term and then pass it through the activation function and then we get Y. So, we can also do this where we do features times weights again, and this creates another tensor, but tensors have a method.sum, where you just take a tensor do.sum and then it sums up all the values in that tensor. So, we can either do it this way or we do torch.sum, or we can just take this method, this sum method of a tensor and some upper values that way. Again, pass it through our our activation function. So, here what we're doing, we're doing this element wise multiplication and taking the sum in two separate operations.



```
## Calculate the output of this network using the weights and bias tensors
```

You can do the multiplication and sum in the same operation using a matrix multiplication. In general, you'll want to use matrix multiplications since they are more efficient and accelerated using modern libraries and high-performance computing on GPUs.

Here, we want to do a matrix multiplication of the features and the weights. For this we can use `torch.mm()` or `torch.matmul()` which is somewhat more complicated and supports broadcasting. If we try to do it with `features` and `weights` as they are, we'll get an error

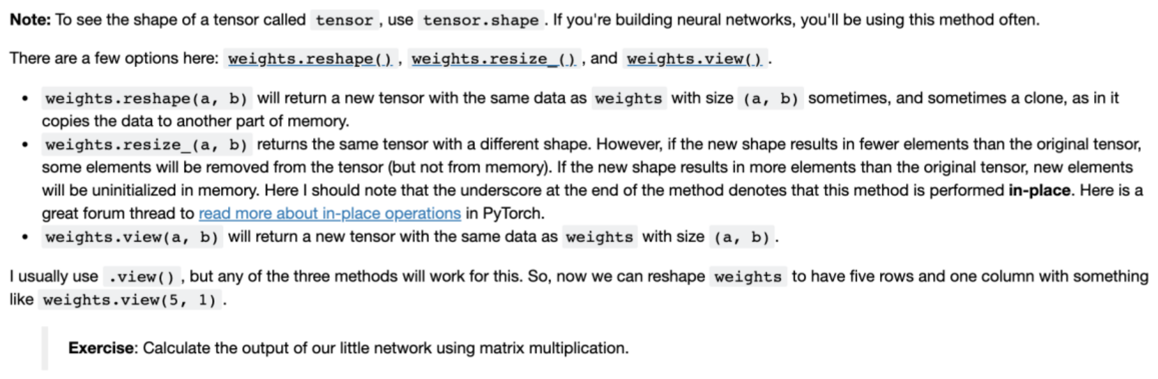
```
>> torch.mm(features, weights)

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-13-15d592eb5279> in <module>()
----> 1 torch.mm(features, weights)

RuntimeError: size mismatch, m1: [1 x 5], m2: [1 x 5] at /Users/soumith/minicondabuild3/conda-bld/pytorch_15245
90658547/work/aten/src/TH/generic/THTensorMath.c:2033
```

As you're building neural networks in any framework, you'll see this often. Really often. What's happening here is our tensors aren't the correct shapes to perform a matrix multiplication. Remember that for matrix multiplications, the number of columns in the first tensor must equal to the number of rows in the second column. Both `features` and `weights` have the same shape, `(1, 5)`. This means we need to change the shape of `weights` to get the matrix multiplication to work.

We're doing this multiplication and then we're doing the sum. But you can actually do this in the same operation using matrix multiplication (tao: dot product of two vectors is equivalent to multiplication of a row matrix and a column matrix). So, in general, you're going to be wanting to use matrix multiplications most of the time, since they're the more efficient and these linear algebra operations have been accelerated using modern libraries, such as CUDA that run on GPUs. To do matrix multiplication in PyTorch with our two tensors features and weights, we can use one of two methods. So, either torch.mm or torch.matmul. So, torch.mm, so matrix multiplication is more simple and more strict about the tensors that you pass in. So, torch.matmul, it actually supports broadcasting. So, if you put in tensors that have weird sizes, weird shapes, then you could get an output that you're not expecting. So, what I tend to use torch.mm more often, so that it does what I expect basically, and then if I get something wrong it's going throw an error instead of just doing it and continuing the calculations. So, however, if we actually try to use torch.mm with features and weights, we'll get an error. So, here we see RuntimeError, size mismatch. So, what this means is that we passed in our two tensors to torch.mm, but there's a mismatch in the sizes and it can't actually do the matrix multiplication and it lists out the sizes here. So, the first tensor, M1 is 1 * 5 and the second tensor is 1 * 5 also. So, if you remember from your linear algebra classes or if you studied it recently, when you're doing matrix multiplication, the first matrix has to have a number of columns that's equal to the number of rows in the second matrix. So, really what we need is we need our weights tensor, our weights matrix to be 5 * 1 instead of one by five.



**Note:** To see the shape of a tensor called `tensor`, use `tensor.shape`. If you're building neural networks, you'll be using this method often.

There are a few options here: `weights.reshape()`, `weights.resize_()`, and `weights.view()`.

- `weights.reshape(a, b)` will return a new tensor with the same data as `weights` with size `(a, b)` sometimes, and sometimes a clone, as in it copies the data to another part of memory.
- `weights.resize_(a, b)` returns the same tensor with a different shape. However, if the new shape results in fewer elements than the original tensor, some elements will be removed from the tensor (but not from memory). If the new shape results in more elements than the original tensor, new elements will be uninitialized in memory. Here I should note that the underscore at the end of the method denotes that this method is performed **in-place**. Here is a great forum thread to read more about in-place operations in PyTorch.
- `weights.view(a, b)` will return a new tensor with the same data as `weights` with size `(a, b)`.

I usually use `.view()`, but any of the three methods will work for this. So, now we can reshape `weights` to have five rows and one column with something like `weights.view(5, 1)`.

> **Exercise**: Calculate the output of our little network using matrix multiplication.

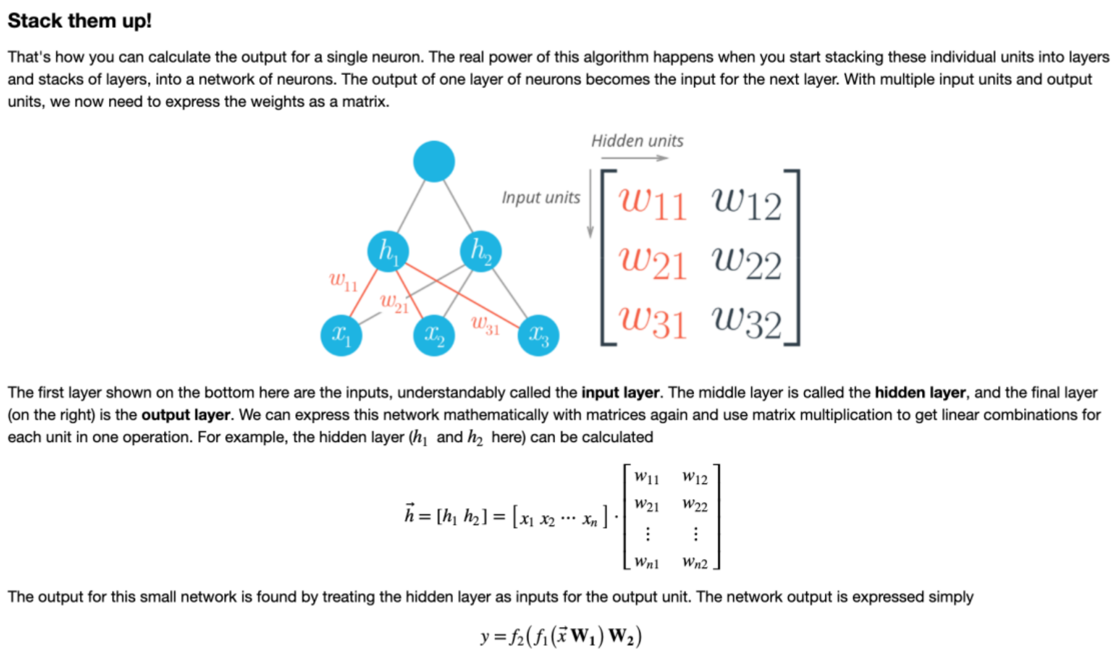
To checkout the shape of tensors, as you're building your networks, you want to use tensor.shape. So, this is something you're going to be using all the time in PyTorch, but also in TensorFlow and in other deep learning frameworks So, most of the errors you're going to see when you're building networks and just a lot of the difficulty when it comes to designing the architecture of neural networks is getting the shapes of your tensors to work right together. So, what that means is that a large part of debugging, you're actually going to be trying to look at the shape of your tensors as they're going through your network. So, remember this, tensor.shape. So, for reshaping tensors, there are three, in general, three different options to choose from. So, we have these methods; reshape, resize, and view. The way these all work, in general, is that you take your tensor weights.reshape and then pass in the new shape that you want. So, in this case, you want to change our weights to be a five by one matrix, so we'd say.reshape and then five comma one. So, reshape here, what it will do is it's going to return a new tensor with the same data as weights. So, the same data that's sitting in memory at those addresses in memory. So, it's going to basically just create a new tensor that has the shape that you requested, but the actual data in memory isn't being changed. But that's only sometimes. Sometimes it does return a clone and what that means is that it actually copies the data to another part of memory and then returns you a tensor on top of that part of the memory. As you can imagine when it actually does that, when it's copying the data that's less efficient than if you had just changed the shape of your tensor without cloning the data. To do something like that, we can use resize, where there's underscore at the end. The underscore means that this method is an in-place operation. So, when it's in-place, that basically means that you're just not touching the data at all and all you're doing is changing the tensor that's sitting on top of that addressed data in memory. The problem with the resize method is that if you request a shape that has more or less elements than the original tensor, then you can actually cut off, you can actually lose some of the data that you had or you can create this spurious data from uninitialized memory. So instead, what you typically want is that you want a method that's going to return an error if you changed the shape from the original number of elements to a different number of elements. So, we can actually do that with.view. So.view is the one that I use the most, and basically what it does it just returns a new tensor with the same data in memory as weights. This is just all the time, 100 percent of the time, all it's going to do is return a new tensor without messing with any of the data in memory. If you tried to get a new size, a new shape for your tensor with a different number of elements, it'll return an error. So, you are basically using.view, you're ensuring that you will always get the same number of elements when you change the shape of your weights. So, this is why I typically use when I'm reshaping tensors. So, with all that out of the way, if you want to reshape weights to have five rows and one column, then you can use something like weights.view (5, 1), right. So, now, that you have seen how you can change the shape of a tensor and also do matrix multiplication, so this time I want you to calculate the output of this little neural network using matrix multiplication.

```
## Solution

y = activation(torch.mm(features, weights.view(5,1)) + bias)
```

75. Welcome to my solution for this exercise. So, for here, I had you calculate the output of our network using matrix multiplication. So remember, we wanted to use matrix multiplication because it's more efficient than doing these two separate operations of the multiplication and the sum. But to do the matrix multiplication, we actually needed to change the size of our weights

tensor. So, to do that, just do weights.view(5, 1), and so this will change the shape of our weights tensor to be five rows and one column. If you remember, our features has the shape of one row and five columns, so we can do this matrix multiplication. So, there's just one operation that does the multiplication and the sum and just one go, and then we again add our bias term, pass it through the activation, and we get our output.



**Stack them up!**

That's how you can calculate the output for a single neuron. The real power of this algorithm happens when you start stacking these individual units into layers and stacks of layers, into a network of neurons. The output of one layer of neurons becomes the input for the next layer. With multiple input units and output units, we now need to express the weights as a matrix.

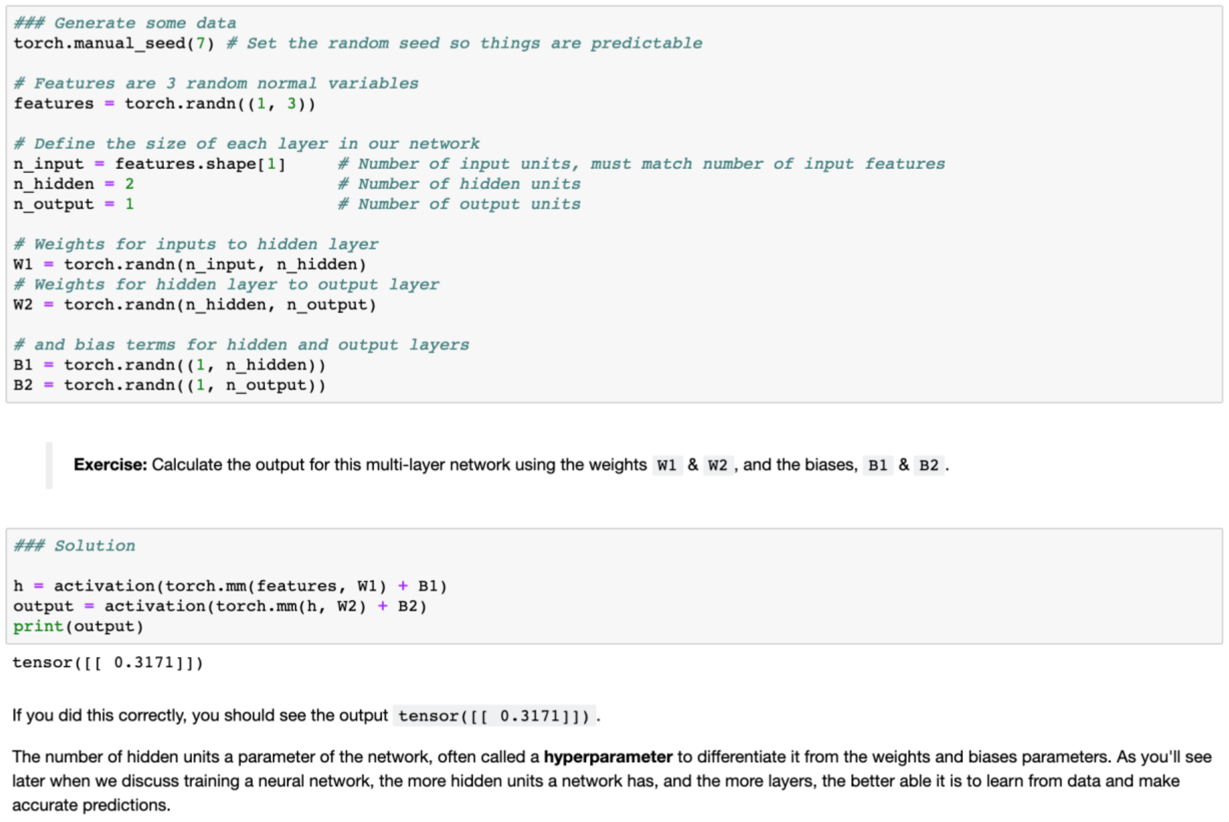
The first layer shown on the bottom here are the inputs, understandably called the **input layer**. The middle layer is called the **hidden layer**, and the final layer (on the right) is the **output layer**. We can express this network mathematically with matrices again and use matrix multiplication to get linear combinations for each unit in one operation. For example, the hidden layer ($h_1$ and $h_2$ here) can be calculated

$$\vec{h} = [h_1 \; h_2] = \begin{bmatrix} x_1 \; x_2 \cdots x_n \end{bmatrix} \cdot \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ \vdots & \vdots \\ w_{n1} & w_{n2} \end{bmatrix}$$

The output for this small network is found by treating the hidden layer as inputs for the output unit. The network output is expressed simply

$$y = f_2(f_1(\vec{x}\,\mathbf{W_1})\,\mathbf{W_2})$$

So, as I mentioned before, you could actually stack up these simple neural networks into a multi-layer neural network, and this basically gives your network greater power to capture patterns and correlations in your data. Now, instead of a simple vector for our weights, we actually need to use a matrix. So, in this case, we have our input vector and our input data x_1, x_2, x_3. You think of this as a vector of just x, which our features. Then we have weights that connect our input to one hidden unit in this middle layers, usually called the hidden layer, hidden units, and we have two units in this hidden layer. So then, if we have our features, our inputs as a row vector, if we multiply it by this first column, then we're going to get the output, we're going to get this value of h_1. Then if we take our features and multiply it by the second column, then we're going to get the value for h_2. So again, mathematically looking at this with matrices and vectors and linear algebra, we see that to get the values for this hidden layer that we do a matrix multiplication between our feature vector, this x_1 to x_n, and our weight matrix. Then as before with these values, we're going to pass them through some activation function or maybe not an activation function, maybe we just want the row output of our network. So here, I'm generating some random data, some features, and some random weight matrices and bias terms that you'll be using to calculate the output of a multi-layer network. So, what I've built is basically we have three input features, two hidden units and one output unit. So, you can see that I've listed it here. So, our features we're going to create three features and this features vector here, and then we have an input equals three, so the shape is this, and two hidden units, one output unit. So, these weight matrices are created using these values. All right. I'll leave it up to you to calculate the output for this multi-layer network. Again, feel free to use the activation function defined earlier for the output of your network and the hidden layer. Cheers.



```
### Generate some data
torch.manual_seed(7) # Set the random seed so things are predictable

# Features are 3 random normal variables
features = torch.randn((1, 3))

# Define the size of each layer in our network
n_input = features.shape[1]     # Number of input units, must match number of input features
n_hidden = 2                    # Number of hidden units
n_output = 1                    # Number of output units

# Weights for inputs to hidden layer
W1 = torch.randn(n_input, n_hidden)
# Weights for hidden layer to output layer
W2 = torch.randn(n_hidden, n_output)

# and bias terms for hidden and output layers
B1 = torch.randn((1, n_hidden))
B2 = torch.randn((1, n_output))
```

> **Exercise:** Calculate the output for this multi-layer network using the weights `W1` & `W2`, and the biases, `B1` & `B2`.

```
### Solution

h = activation(torch.mm(features, W1) + B1)
output = activation(torch.mm(h, W2) + B2)
print(output)
```

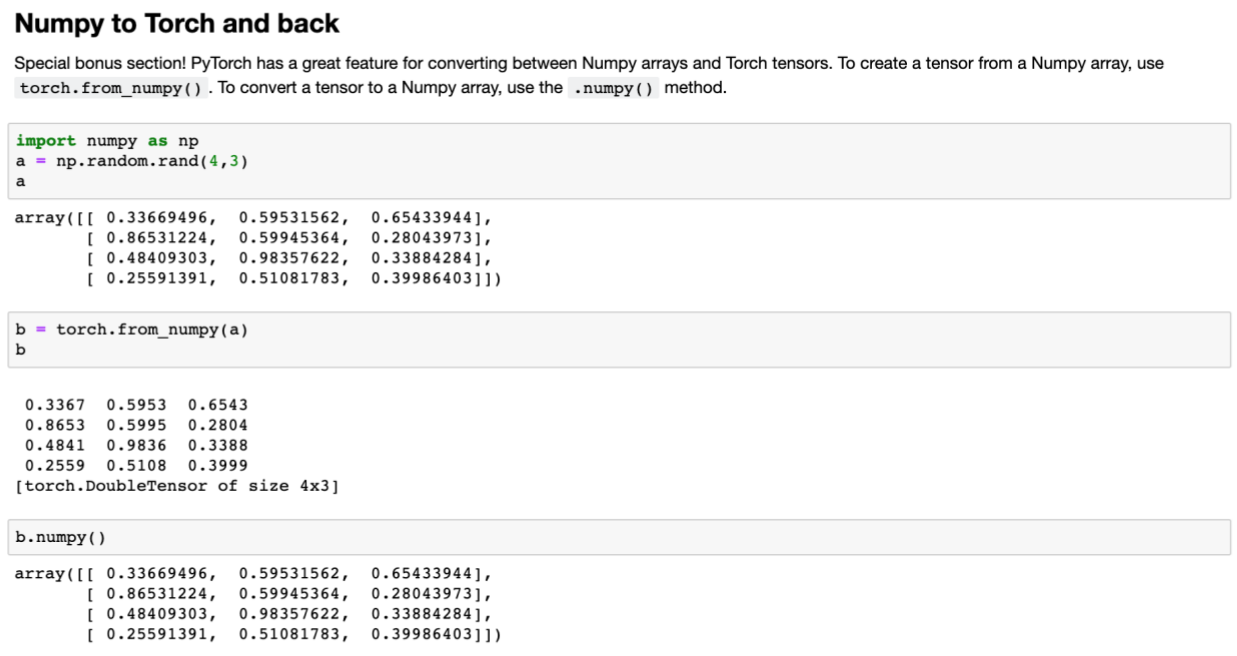
```
tensor([[ 0.3171]])
```

If you did this correctly, you should see the output `tensor([[ 0.3171]])`.

The number of hidden units a parameter of the network, often called a **hyperparameter** to differentiate it from the weights and biases parameters. As you'll see later when we discuss training a neural network, the more hidden units a network has, and the more layers, the better able it is to learn from data and make accurate predictions.

76. All right. So, here's my solution for this exercise. So, here, I had you calculate the output of this multi-layer network using the weights and features that we've defined up here. So, it was really

similar to what we did before with our single layer simple neural network. So, it's basically just taking the features and our weight matrix, our first weight matrix, and calculating a matrix multiplication. So, here's the torch.mm plus B1, and then that gives us values for our hidden layer H. Now, we can use the values H as the input for the next layer of our network. So, we just do, again, a matrix multiplication of these hidden values H, with our second weight matrix W2, and adding on our bias terms, and then we get the output.



## Numpy to Torch and back

Special bonus section! PyTorch has a great feature for converting between Numpy arrays and Torch tensors. To create a tensor from a Numpy array, use `torch.from_numpy()`. To convert a tensor to a Numpy array, use the `.numpy()` method.

```
import numpy as np
a = np.random.rand(4,3)
a
```

```
array([[ 0.33669496,  0.59531562,  0.65433944],
       [ 0.86531224,  0.59945364,  0.28043973],
       [ 0.48409303,  0.98357622,  0.33884284],
       [ 0.25591391,  0.51081783,  0.39986403]])
```

```
b = torch.from_numpy(a)
b
```

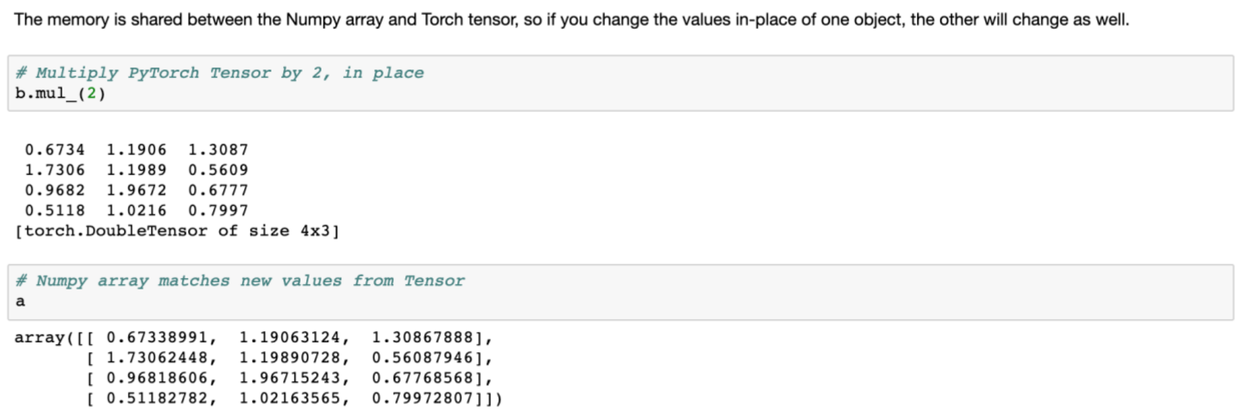
```

 0.3367  0.5953  0.6543
 0.8653  0.5995  0.2804
 0.4841  0.9836  0.3388
 0.2559  0.5108  0.3999
[torch.DoubleTensor of size 4x3]
```

```
b.numpy()
```

```
array([[ 0.33669496,  0.59531562,  0.65433944],
       [ 0.86531224,  0.59945364,  0.28043973],
       [ 0.48409303,  0.98357622,  0.33884284],
       [ 0.25591391,  0.51081783,  0.39986403]])
```

So, my favorite features of PyTorches is being able to convert between Numpy arrays and Torch tensors, in a very nice and easy manner. So, this is really useful because a lot of the times, you'll be preparing your data and to do some preprocessing using Numpy, and then you want to move it into your network, and so, you have to bridge these Numpy arrays, what you're using for your data, and then the Torch tensors that you're using for your network. So, actually, to do this, we can actually get a tensor from a Numpy array using torch.from_numpy. So, here I've just created a random array, a four-by-three array, and then we can create a Torch tensor from this array just by doing.from Numpy, and passing an array. So, this creates a nice tensor for us. So, this is a tensor in PyTorch, we can use with all of our Torch methods and eventually, use it in a neural network. Then, we can go backwards, so we can take a tensor such as B here. This is our Torch tensor and we can go back to a Numpy array doing b.numpy. So, this gives us back our Numpy array.



The memory is shared between the Numpy array and Torch tensor, so if you change the values in-place of one object, the other will change as well.

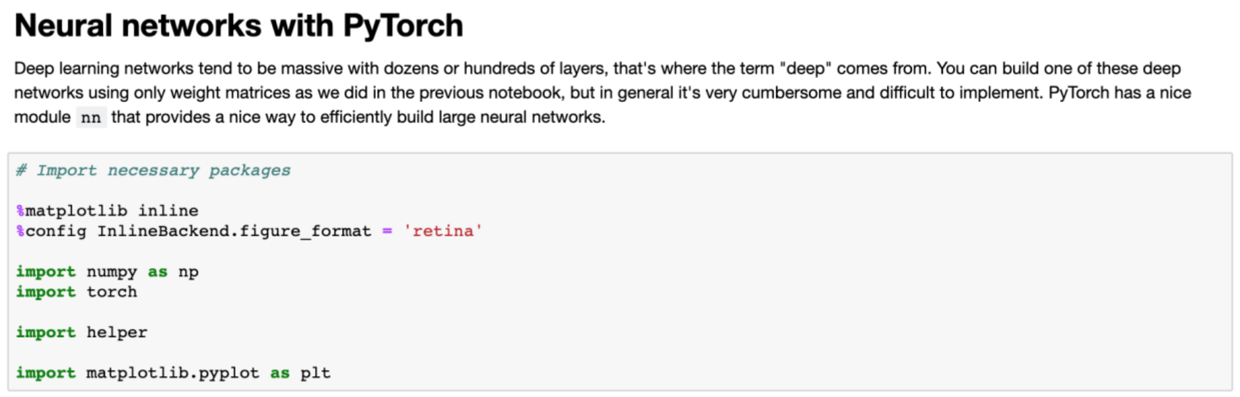
```
# Multiply PyTorch Tensor by 2, in place
b.mul_(2)
```

```
 0.6734  1.1906  1.3087
 1.7306  1.1989  0.5609
 0.9682  1.9672  0.6777
 0.5118  1.0216  0.7997
[torch.DoubleTensor of size 4x3]
```

```
# Numpy array matches new values from Tensor
a
```

```
array([[ 0.67338991,  1.19063124,  1.30867888],
       [ 1.73062448,  1.19890728,  0.56087946],
       [ 0.96818606,  1.96715243,  0.67768568],
       [ 0.51182782,  1.02163565,  0.79972807]])
```

So, one thing to remember when you're doing this, is that the memory is actually shared between the Numpy array and this Torch tensor. So, what this means, is that if you do any operations in place on either the Numpy array or the tensor, then you're going to change the values for the other one. So, for example, if we do this in-place operation of multiplying by two, which means that we're actually changing the values in memory, and not creating a new tensor, then we will actually change the values in the Numpy array. So, you see here, we have our Numpy array. So initially, it's like this, convert it to a Torch tensor, and here, I'm doing this in-place multiplication, and we've changed our values for this tensor. Then, if you look back at the Numpy array, the values have changed. So, that's just something to keep in mind as you're doing this, so you're not caught off guard when you're seeing your arrays, your Numpy arrays, being changed because of operations you're doing on the tensor. See you in the next video, cheers.



## Neural networks with PyTorch

Deep learning networks tend to be massive with dozens or hundreds of layers, that's where the term "deep" comes from. You can build one of these deep networks using only weight matrices as we did in the previous notebook, but in general it's very cumbersome and difficult to implement. PyTorch has a nice module `nn` that provides a nice way to efficiently build large neural networks.

```
# Import necessary packages

%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import numpy as np
import torch

import helper

import matplotlib.pyplot as plt
```

77. Hello everyone and welcome back. So, in this notebook and series of videos, I'm going to be showing you a more powerful way to build neural networks and PyTorch. So, in the last notebook, you saw how you can calculate the output for network using tensors and matrix multiplication. But PyTorch has this nice module, nn, that has a lot of my classes and methods and functions that allow us to build large neural networks in a very efficient way.

Now we're going to build a larger network that can solve a (formerly) difficult problem, identifying text in an image. Here we'll use the MNIST dataset which consists of greyscale handwritten digits. Each image is 28x28 pixels, you can see a sample below

So, to show you how this works, we're going to be using a dataset called MNIST. So, MNIST it's a whole bunch of grayscale handwritten digits. So ,0, 1, 2, 3, 4 and so on through nine. Each of these images is 28 by 28 pixels and the goal is to actually identify what the number is in these images. So, that dataset consists of each of these images and it's labeled with the digit that is in that image. So, ones are labeled one, twos are labeled two and so on. So, what we can do is we can actually show our network and image and the correct label and then it learns how to actually determine what the number and the image is. This dataset is available through the torchvision package. So, this is a package that sits alongside PyTorch, that provides a lot of nice utilities like datasets and models for doing computer vision problems.

Our goal is to build a neural network that can take one of these images and predict the digit in the image.

First up, we need to get our dataset. This is provided through the `torchvision` package. The code below will download the MNIST dataset, then create training and test datasets for us. Don't worry too much about the details here, you'll learn more about this later.

```python
### Run this cell

from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                               transforms.Normalize((0.5,), (0.5,)),
                               ])
# Download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

We have the training data loaded into `trainloader` and we make that an iterator with `iter(trainloader)`. Later, we'll use this to loop through the dataset for training, like

```python
for image, label in trainloader:
    ## do things with images and labels
```

You'll notice I created the `trainloader` with a batch size of 64, and `shuffle=True`. The batch size is the number of images we get in one iteration from the data loader and pass through our network, often called a *batch*. And `shuffle=True` tells it to shuffle the dataset every time we start going through the data loader again. But here I'm just grabbing the first batch so we can check out the data. We can see below that `images` is just a tensor with size `(64, 1, 28, 28)`. So, 64 images per batch, 1 color channel, and 28x28 images.

We can run this cell to download and load the MNIST dataset. What it does is it gives us back an object which I'm calling trainloader. So, with this trainloader we can turn into an iterator with iter and then this will allow us to start getting good at it or we can actually just use this in a loop, in a for loop and so we can get our images and labels out of this generator with four image, comma label and trainloader. One thing to notice is that when I created the trainloader, I set the batch size to 64. So, what that means and every time we get a set of images and labels out, we're actually getting 64 images out from our data loader. So, then if you look at the shape and the size of these images, we'll see that they are 64 by one by 28 by 28. So, 64 images and then one color channels so it's grayscale, and then it's 28 by 28 pixels is the shape of these images and so we can see that here. Then our labels have a shape of 64 so it's just a vector that's 64 elements which with a label for each of our images and we can see what one of these images looks like this is a nice number four. So, we're going to do here is build a multi-layer neural network using the methods that we saw before. By that I mean you're going to initialize some weight matrices and some bias vectors and use those to calculate the output of this multi-layer network. Specifically, we want to build this network with 784 input units, 256 hidden units, and 10 output units. So, 10 output units, one for each of our classes. So, the 784 input units, this comes from the fact that with this type of network is called a fully connected network or a dense network. We want to think of our inputs as just one vector. So, our images are actually this 28 by 28 image, but we want to put a vector into our network and so what we need to do is actually convert this 28 by 28 image into a vector and so, 784 is 28 times 28. When we actually take this 28 by 28 image and flatten it into a vector then it's going to be 784 elements long. So, now what we need to do is take each of our batches which is 64 by one by 28 by 28 and then convert it into a shape that is to another tensor which shapes 64 by 784. This is going to be the tensor that's the input to our network. So, go and give this a shot. So again, build the networks 784 input units, 256 hidden units and 10 output units and you're going to be generating your own random initial weight and bias matrices. Cheers.

Stopa
Not_read_begin

78. Here is my solution for this multi-layer neural network for classifying handwritten digits from the MNIST dataset. So, here I've defined our activation function like before, so, again this is the sigmoid function and here I'm flattening the images. So, remember how to reshape your tensors. So, here I'm using.view. So, I'm just grabbing the batch size. So, images.shape. The first element zero here, gives you the number of batches in your images tensor. So, I want to keep the number of batches the same, but I want to flatten the rest of the dimensions. So, to do this, you actually can just put in negative one. So, I could type in 784 here but a kind of a shortcut way to do this is to put in negative one. So, basically what this does is it takes 64 as your batch size here and then when you put a negative one it sees this and then it just chooses the appropriate size to get the total number of elements. So, it'll work out on its own that it needs to make the second dimension, 784 so that the number of elements after reshaping matches the number elements before reshaping. So, this is just a kind of quick way to flatten a tensor without having to know what the second dimension used to be. Then here I'm just creating our weight and bias parameters. So, we know that we want an input of 784 units and we want 256 hidden units. So, our first weight matrix is going to be 784 by 256. Then, we need a bias term for each of our hidden units. So we have 256 bias terms here in b1. Then, for our second weight's going from the hidden layer to the output layer we want 256 inputs to 10 outputs. Then again 10 elements in our bias. Before we can do a matrix multiplication of our inputs with the first set of weights, our first weight parameters, add in the bias terms and passes through our activation functions so that gives us the output of our hidden layer. Then we can use that as the input to our output layer, and again, a matrix multiplication with a second set of weights and the second set of bias terms. This gives us the output of our network. All right. So, if we look at the output of this network, we see that we get those 64. So, first let me print the shape just to make sure we did that right. So, 64 rows for one of each of our sort of input examples and then 10 values, so, basically it's a value that's trying to say this image belongs to this

class like this digit. So, we can inspect our output tensor and see what's going on here. So, we see these values are just sort of all over the place. So, you got like six and negative 11 and so on. But we really want is we want our network to kind of tell us the probability of our different classes given some image. So, kind of we want to pass in an image to our network and then the output should be a probability distribution that tells us which are the most likely classes or digits that belong to this image. So, if it's the image of a six, then we want a probability distribution where most of the probability is in the sixth class. So, it's telling us that it's a number six. So, we want it to look something like this. This is like a class probability. So, it's telling us the probabilities of the different classes given this image that we're passing in. So, you can see that the probability for each of these different classes is roughly the same, and so it's a uniform distribution. This represents an untrained network, so it's a uniform probability distribution. It's because it hasn't seen any data yet, so it hasn't learned anything about these images. So, whenever you give an image to it, it doesn't know what it is so it's just going to give an equal probability to each class, regardless of the image that you pass in. So, what we want is we want the output of our network to be a probability distribution that gives us the probability that the image belongs to any one of our classes. So for this, we use the softmax function. So what this looks like is the exponential. So,you pass in your 10 values. So, for each of those values, we calculate the exponential of that value divided by the sum of exponentials of all the values. So, what this does is it actually kind of squishes each of the input values x between zero and one, and then also normalizes all the values so that the sum of each of the probabilities is one. So, the entire thing sums up to one. So, this gives you a proper probability distribution. What I want you to do here is actually implement a function called softmax that performs this calculation. So, what you're going to be doing is taking the output from this simple neural network and has shaped 64 by 10 and pass it through a dysfunction softmax and make sure it calculates the probability distribution for each of the different examples that we passed in. Right? Good luck.

79. Welcome back. Here is my solution for the softmax function. Here in the numerator, we know we want to take the exponential, so it's pretty straight forward with torch.exp. So we're going to use the exponential of x, which is our input tensor. In the denominator, we know we want to do something like, again take exponentials so torch.exp, and then take the sum across all those values. So, one thing we need to remember is that we want the sum across one single row. So, each of the columns in one single row for each example. So, for one example, we want to sum up those values. So, for here in torch.sum, we're going to use dimension equals one. So, this is basically going to take the sum across the columns. What this does, torch.sum here, is going to actually going to give us a tensor, that is just a vector of 64 elements. So, the problem with this is that, if this is 64 by 10, and this is just a 64-long vector, it's going to try to divide every element in this tensor by all 64 of these values. So, it's going give us a 64 by 64 tensor, and that's not what we want. We want our output to be 64 by 10. So, what you actually need to do is reshape this tensor here to have 64 rows, but only one value for each of those rows. So, what that's going do, it's going look at for each row in this tensor, is going to look at the equivalent row in this tensor. So, since each row in this tensor only has one value, it's going to divide this exponential by the one value in this denominator tensor. This can be really tricky, but it's also super important to understand how broadcasting works in PyTorch, and how to actually fit all these tensors together with the correct shape and the correct operations to get everything out right. So, if we do this, it look what we have, we pass our output through the softmax function, and then we get our probabilities, and we can look the shape and it is 64 by 10, and if you take the sum across each of the rows, then it adds up to one, like it should

with a proper probability distribution. So, now, we're going to look at how you use this nn module to build neural networks. So, you'll find that it's actually in a lot of ways simpler and more powerful. You'll be able to build larger and larger neural networks using the same framework. The way this works in general, is that we're going to create a new class, and you can call it networking, you can call it whatever you want, you can call it classifier, you can call it MNIST. It doesn't really matter so much what you call it, but you need to subclass it from nn.module. Then, in the init method, it's __init method. You need to call it super and run the init method of nn.module. So, you need to do this because then, PyTorch will know to register all the different layers and operations that you're going to be putting into this network. If you don't do this part then, it won't be able to track the things that you're adding to your network, and it just won't work. So, here, we can create our hidden layers using nn.Linear. So, what this does, is it creates a operation for the linear transformation. So, when we take our inputs x and then multiply it by weights and add your bias terms, that's a linear transformation. So, what this does is calling NN.Linear, it creates an object that itself has created parameters for the weights and parameters for the bias and then, when you pass a tensor through this hidden layer, this object, it's going to automatically calculate the linear transformation for you. So, all you really need to do is tell it what's the size of the inputs, and then what are the size of the output. So, 784 by 256, we're going to use 256 outputs for this. So, it's kind of rebuilding the network that we saw before. Similarly, we want another linear transformation between our hidden units and our output. So, again, we have 256 hidden units, and we have 10 outputs, 10 output units, so we're going to create a output layer called self.output, and create this linear transformation operation. We also want to create a sigmoid operation for the activation and then, softmax for the output, so we get this probability distribution. Now, we're going to create a forward method and so, forward is basically going to be, as we pass a tensor in to the network. It's gonna go through all these operations, and eventually give us our output. So, here, x, the argument is going to be the input tensor and then, we're going to pass it through our hidden layer. So, this is again, like this linear transformation that we defined up here, and it's going to go through a sigmoid activation, and then through our output layer or output linear transformation, we have here, and then through the sigmoid function, and then finally return the output of our softmax. so we can create this. Then, if we kind of look at it, so it'll print it out, and it'll tell us the operations, and not necessarily the order, but at least it tells us the operations that we have defined for this network. You can also use some functional definitions for things like sigmoid and softmax, and it kind of makes the class the way you write the code a little bit cleaner. We can get that from torch.nn.functional. Most of the time, you'll see is like import torch.nn.functional as capital F. So, there's kind of that convention in PyTorch code. So, again, we define our linear transformations, self.hidden, self.output but now in our forward method. So, we can call self.hidden to get like our values for hidden layer, but then, we pass it through the sigmoid function, f.sigmoid, and the same thing with the output layers. So, we have our output linear transformations of the output, and we pass it through this softmax operation. So, the reason we can do this because, when we create these linear transformations, it's creating the weights and bias matrices on its own. But for sigmoid and softmax, it's just an element wise operation, so it doesn't have to create any extra parameters or extra matrices to do these operations, and so we can have these be purely functional without having to create any sort of object or classes. However, they are equivalent. So this way to build the network is equivalent to this way up here, but it's a little bit more succinct when you're doing it with these kind of functional pattern. So far, we've only been using the sigmoid function as an activation function, but there are, of course, a lot of different ones you want to use. Really the only requirement is that, these activation functions should typically be non-linear. So, if you want your network to be able to learn non-linear correlations and patterns, and we want the output to be non-linear, then you need to use non-linear activation functions in your hidden layers. So, a sigmoid is one example. The hyperbolic tangent is another. One that is pretty much used all the time, like almost exclusively as activation function and hidden layers, is the ReLU, so the rectified linear unit. This is basically the simplest non-linear function that you can use, and it turns out that networks tend to train a lot faster when using ReLU as compared to sigmoid and hyperbolic tangent, so ReLU was what we typically use. Okay. So, here, you're going to build your own neural network, that's larger. So, this time, it's going to have two hidden layers, and you'll be using the ReLU activation function for this on your hidden layers. So using this object-oriented class method within a.module, go ahead and build a network that looks like this, with 784 input units, a 128 units in the first hidden layer, 64 units and the second hidden layer, and then 10 output units. All right. Cheers.

80. Hello, everyone, and welcome back. So, in this video and in this notebook, I'll be showing you how to actually train neural networks in PyTorch. So, previously, we saw how to define neural networks in PyTorch using the nn module, but now we're going to see how we actually take one of these networks that we defined and train it. So, what I mean by training is that we're going to use our neural networks as a universal function approximator. What that means is that, for basically any function, we have some desired input for example, an image of the number four, and then we have some desired output of this function. In this case a probability distribution that is telling us the probabilities of the various digits. So, in this case, if we passed it in image four, we want to get out a probability distribution where there's a lot of probability in the digit four. So, the cool thing about neural networks is that if you use non-linear activations and then you have the correct dataset of these images that are labeled with the correct ones, then basically you pass in an image and the correct output, the correct label or class, and eventually your neural network will build to approximate this function that is converting these images into this probability distribution, and that's our goal here. So, basically we want to see how in PyTorch, we can build a neural network and then we're going to give it the inputs and outputs, and then adjust the weights of that network so that it approximates this function. So, the first thing that we need for that is what is called a loss function. So, it's sometimes also called the cost, and what this is it's a measure of our prediction error. So, we pass in the image of a four and then our network predicts something else that's an error. So, we want to measure how far away our networks prediction is from the correct label, and we do that using loss function. So, in this case, it's the mean squared error. So, a lot of times you'll use this in regression problems, but use other loss functions and classification problems like this one here. So, the loss depends on the output of our network or the predictions our network is making. The output of a network depends on the weight. So, like the network parameters. So, we can actually adjust our weights such that this loss is minimized, and once the loss is minimized, then we know that our network is making as good predictions as it can. So, this is the whole goal to adjust our network parameters to minimize our loss, and we do this by using a process called gradient descent. So, the gradient is the slope of the loss function with respect to our perimeters. The gradient always points in the direction of fastest change. So, for example if you have a mountain, the gradient is going to always point up the mountain. So, you can imagine our loss function being like this mountain where we have a high loss up here and we have a low loss down here. So, we know that we want to get to the minimum of our loss when we minimize our loss, and so, we want to go downwards. So, basically, the gradient points upwards and so, we just go the opposite direction. So, we go in the direction of the negative gradient, and then if we keep

following this down, then eventually we get to the bottom of this mountain, the lowest loss. With multilayered neural networks, we use an algorithm called backpropagation to do this. Backpropagation is really just an application of the chain rule from calculus. So, if you think about it when we pass in some data, some input into our network, it goes through this forward pass through the network to calculate our loss. So, we pass in some data, some feature input x and then it goes through this linear transformation which depends on our weights and biases, and then through some activation function like a sigmoid, through another linear transformation with some more weights and biases, and then that goes in, from that we can calculate our loss. So, if we make a small change in our weights here, W1, it's going to propagate through the network and end up like results in a small change in our loss. So, you can think of this as a chain of changes. So, if we change here, this is going to change. Even that's going to propagate through here, it's going to propagate through here, it's going to propagate through here. So, with backpropagation, we actually use these same changes, but we go in the opposite direction. So, for each of these operations like the loss and the linear transformation into the sigmoid activation function, there's always going to be some derivative, some gradient between the outputs and inputs, and so, what we do is we take each of the gradients for these operations and we pass them backwards through the network. Each step we multiply the incoming gradient with the gradient of the operation itself. So, for example just starting at the end with the loss. So, we pass this gradient or the loss dldL2. So, this is the gradient of the loss with respect to the second linear transformation, and then we pass that backwards again and if we multiply it by the loss of this L2. So, this is the linear transformation with respect to the outputs of our activation function, that gives us the gradient for this operation. If you multiply this gradient by the gradient coming from the loss, then we get the total gradient for both of these parts, and this gradient can be passed back to this softmax function. So, as the general process for backpropagation, we take our gradients, we pass it backwards to the previous operation, multiply it by the gradient there, and then pass that total gradient backwards. So, we just keep doing that through each of the operations in our network, and eventually we'll get back to our weights. What this does is it allows us to calculate the gradient of the loss with respect to these weights. Like I was saying before, the gradient points in the direction of fastest change in our loss, so, to maximize it. So, if we want to minimize our loss, we can subtract the gradient off from our weights, and so, what this will do is it'll give us a new set of weights that will in general result in a smaller loss. So, the way that backpropagation algorithm works is that it will make a forward pass through a network, calculate the loss, and then once we have the loss, we can go backwards through our network and calculate the gradient, and get the gradient for a weights. Then we'll update the weights. Do another forward pass, calculate the loss, do another backward pass, update the weights, and so on and so on and so on, until we get sufficiently minimized loss. So, once we have the gradient and like I was saying before, we can subtract it off from our weights, but we also use this term Alpha which is called the learning rate. This is basically just a way to scale our gradients so that we're not taking too large steps in this iterative process. So, what can happen if you're update steps are too large, you can bounce around in this trough around the minimum and never actually settle in the minimum of the loss. So, let's see how we can actually calculate losses in PyTorch. Again using the nn module, PyTorch provides us a lot of different losses including the cross-entropy loss. So, this loss is what we'll typically use when we're doing classification problems. In PyTorch, the convention is to assign our loss to its variable criterion. So, if we wanted to use cross-entropy, we just say criterion equals nn.crossEntropyLoss and create that class. So, one thing to note is that, if you look at the documentation for cross-entropy loss, you'll see that it actually wants the scores like the logits of our network as the input to the cross-entropy loss. So, you'll be using this with an output such as softmax, which gives us this nice probability distribution. But for computational reasons, then it's generally better to use the logits which are the input to the softmax function as the input to this loss. So, the input is expected to be the scores for each class and not the probabilities themselves. So, first I'm going to import the necessary modules here and also download our data and create it in, like you've seen before, as a trainloader, and so, we can get our data out of here. So, here I'm defining a model. So, I'm using nn.Sequential, and if you haven't seen this, checkout the end of the previous notebook. So, the end of part two, will show you how to use nn.Sequential. It's just a somewhat more concise way to define simple feed-forward networks, and so, you'll notice here that I'm actually only returning the logits, the scores of our output function and not the softmax output itself. Then here we can define our loss. So, criterions equal to nn.crossEntropyLoss. We get our data with images and labels, flatten it, pass it through our model to get the logits, and then we can get the actual loss by bypassing in our logits and the true labels, and so, again we get the labels from our trainloader. So, if we do this, we see we have calculated the loss. So, my experience, it's more convenient to build your model using a log-softmax output instead of just normal softmax. So, with a log-softmax output to get the actual probabilities, you just pass it through torch.exp. So, the exponential. With a log-softmax output, you'll want to use the negative log-likelihood loss or nn.NLLLoss. So, what I want you to do here is build a model that returns the log-softmax as the output, and calculate the loss using the negative log-likelihood loss. When you're using log-softmax, make sure you pay attention to the dim keyword argument. You want to make sure you set it right so that the output is what you want. So, go and try this and feel free to check out my solution. It's in the notebook and also in the next video, if you're having problems. Cheers.

81. Hi and welcome back. Here's my solution for this model that uses a LogSoftmax output. It is a pretty similar to what I built before with an index sequential. So, we just use a linear transformation, ReLU, linear transformation, ReLU, another linear transformation for output and then we can pass this to our LogSoftmax module. So, what I'm doing here is I'm making sure I set the dimension to one for LogSoftmax and this makes it so that it calculates the function across the columns instead of the rows. So, if you remember, the rows correspond to our examples. So, we have a batch of examples that we're passing to our network and each row is one of those examples. So, we want to make sure that we're covering the softmax function across each of our examples and not across each individual feature in our batches. Here, I'm just defining our loss or criterion as the negative log likelihood loss and again get our images and labels from our train loader, flatten them, pass it through our model to get the logits. So, this is actually not the largest anymore, this is like a log probability, so we call it like logps, and then you do that. There you go. You see we get our nice loss. Now, we know how to calculate a loss, but how do we actually use it to perform backpropagation? So, PyTorch towards actually has this really great module called Autograd that automatically calculates the gradients of our tensors. So, the way it works is that, PyTorch will keep track of all the operations you do on a tensor and then when you can tell it to do a backwards pass, to go backwards through each of those operations and calculate the gradients with respect to the input parameters. In general, you need to tell PyTorch that you want to use autograd on a specific tensor. So, in this case, you would create some tensor like x equals torch.zeros, just to make it a scalar, say one and then give it requires grad equals true. So, this tells PyTorch to track the operations on this tensor x, so that if you want to get the gradient then it will calculate it for you. So, in general, if you're creating a tensor and you don't want to calculate the gradient for it, you want to make sure this is set to false. You can also use this context torch.no grad to make sure

all the gradients are shut off for all of the operations that you're doing while you're in this context. Then, you can also turn on or off gradients globally with torch.set grad enabled and give it true or false, depending on what you want to do. So, the way this works in PyTorch is that you basically create your tensor and again, you set requires grad equals true and then you just perform some operations on it. Then, once you are done with those operations, you type in.backwards. So, if you use x, this tensor x, then calculate some other tensor z then if you do z.backward, it'll go backwards through your operations and calculate the total gradient for x. So, for example, if I just create this random tensor, random two-by-two tensor, and then I can square it like this. What it does, you can actually see if you look at y, so y is our secondary or squared tensor. If you look at y.grad function, then it actually shows us that this grad function is a power. So, PyTorch just track this and it knows that the last operation done was a power operation. So, now, we can take the mean of y and get another tensor z. So, now this is just a scalar tensor, we've reduced y, y is a two-by-two matrix, two by two array and then we take in the mean of it to get z. Ingredients for tensor show up in this attribute grad, so we can actually look at what's the gradient of our tensor x right now, and we've only done this forward pass, we haven't actually calculated the gradient yet and so it's just none. So, now if we do z.backward, it's going to go backwards through this tiny little set of operations that we've done. So, we did a power and then a mean and let's go backwards through this and calculate the gradient for x. So, if you actually work out the math, you find out that the gradient of z with respect to x should be x over two and if we look at the gradient, then we can also look at x divided by two then they are the same. So, our gradient is equal to what it should be mathematically, and this is the general process for working with gradients, and autograd, and PyTorch. Why this is useful, is because we can use this to get our gradients when we calculate the loss. So, if remember, our loss depends on our weight and bias parameters. We need the gradients of our weights to do gradient descent. So, what we can do is we can set up our weights as tensors that require gradients and then do a forward pass to calculate our loss. With the loss, you do a backwards pass which calculates the gradients for your weights, and then with those gradients, you can do your gradient descent step. Now, I'll show you how that looks in code. So, here, I'm defining our model like I did before with LogSoftmax output, then using the negative log-likelihood loss, get our images and labels from our train loader, flatten it, and then we can get our log probabilities from our model and then pass that into our criterion, which gives us the actual loss. So, now, if we look at our models weights, so model zero gives us the parameters for this first linear transformation. So, we can look at the weight and then we can look at the gradient, then we'll do our backwards pass starting from the loss and then we can look at the weight gradients again. So, we see before the backward pass, we don't have any because we haven't actually calculated it yet but then after the backwards pass, we have calculated our gradients. So, we can use these gradients in gradient descent to train our network. All right. So, now you know how to calculate losses and you know how to use those losses to calculate gradients. So, there's one piece left before we can start training. So, you need to see how to use those gradients to actually update our weights, and for that we use optimizers and these come from PyTorch's Optim package. So, for example, we can use stochastic gradient descent with optim.SGD. The way this is defined is we import this module optim from PyTorch and then we'd say optim.SGD, we give it our model parameters. So, these are the parameters that we want this optimizer to actually update and then we give it a learning rate, and this creates our optimizer for us. So, the training pass consists of four different steps. So, first, we're going to make a forward pass through the network then we're going to use that network output to calculate the loss, then we'll perform a backwards pass through the network with loss.backwards and this will calculate the gradients. Then, we'll make a step with our optimizer

that updates the weights. I'll show you how this works with one training step and then you're going to write it up for real and a loop that is going to train a network. So, first, we're going to start by getting our images and labels like we normally do from our train loader and then we're going to flatten them. Then next, what we want to do is actually clear the gradients. So, PyTorch by default accumulates gradients. That means that if you actually do multiple passes and multiple backwards like multiple forward passes, multiple backwards passes, and you keep calculating your gradient, it's going to keep summing up those gradients. So, if you don't clear gradients, then you're going to be getting gradients from the previous training step in your current training step and it's going to end up where your network is just not training properly. So, for this in general, you're going to be calling zero grad before every training passes. So, you just say optimizer.zero grad and this will just clean out all the gradients and all the parameters in your optimizer, and it'll allow you to train appropriately. So, this is one of the things in PyTorch that is easy to forget, but it's really important. So, try your hardest to remember to do this part, and then we do our forward pass, backward pass, then update the weights. So, we get our output, so we do a forward pass through our model with our images, then we calculate the loss using the output of the model and our labels, then we do a backwards pass and then finally we take an optimizer step. So, if we look at our initial weights, so it looks like this and then we can calculate our gradient, and so the gradient looks like and then if we take an optimizer step and update our weights, then our weights have changed. So, in general, what has worked is you're going to be looping through your training set and then for each batch out of your training set, you'll do the same training pass. So, you'll get your data, then clear the gradients, pass those images or your input through your network to get your output, from that, in the labels, calculate your loss, and then do a backwards pass on the loss and then update your weights. So, now, it's your turn to implement the training loop for this model. So, the idea here is that we're going to be looping through our data-set, so grabbing images and labels from train loader and then on each of those batches, you'll be doing the training pass, and so you'll do this pass where you calculate the output of the network, calculate the loss, do backwards pass on loss, and then update your weights. Each pass through the entire training set is called an epoch and so here I just have it set for five epochs. So, you can change this number if you want to go more or less. Once you calculate the loss, we can accumulate it to keep track, so we're going to be looking at a loss. So, this is running loss and so we'll be printing out the training loss as it's going along. So, if it's working, you should see the loss start falling, start dropping as you're going through the data. Try this out yourself and if
you need some help, be sure to check out my solution. Cheers.

82. Hi again. So, here's my solution for the train pass that I had you implement. So, here we're just defining our model like normal and then our negative log-likelihood loss using stochastic gradient descent and pass in our parameters. Then, here is our training pass. So, for each image in labels in trainloader, we're going to flatten it and then zero out the gradients using optimizer. zero_ grad. Pass our images forward through the model and the output and then from that, we can calculate our loss and then do a backward pass and then finally with the gradients, we can do this optimizer step. So, if I run this and we wait a little bit for to train, we can actually see the loss dropping over time, right? So, after five epochs, we see that the first one, it starts out fairly high at 1.9 but after five epochs, continuous drop as we're training and we see it much lower after five epochs. So, if we kept training then our network would learn the data better and better and the training loss would be even smaller. So, now with our training network, we can actually see what our network thinks it's seen in these images. So, for here, we can pass in an image. In this case, it's the image of a

number two and then this is what our network is predicting now. So, you can see pretty easily that it's putting most of the probability, most of its prediction into the class for the digit two. So we try it again and put in passes in number eight and again, it's predicting eight. So, we've managed to actually train our network to make accurate predictions for our digits. So next step, you'll write the code for training a neutral network on a more complex dataset and you'll be doing the whole thing, defining the model, running the training loop, all that. Cheers.

83. Welcome back. So, in this notebook, you'll be building your own neural network to classify clothing images. So, like I talked about in the last video, MNIST is actually a fairly trivial dataset these days. It's really easy to get really high accuracy with a neural network. So instead, you're going to be using Fashion-MNIST, and this is basically just a drop-in replacement for MNIST so we have 28 by 28 grayscale images, but this time it's clothing. So, you have a lot more variation in the classes, and it just ends up being a much more difficult problem to classify like there's a t-shirt, there's pants, there's a sweater, there's shoes instead of handwritten digits. So it's a better representation of datasets that you'd use in the real world. So, I've left this up to you to actually build a network and train it. So here you can define your network architecture, then here you will create your network to define the criterion and optimizer and then write the code for the training pass. Once you have your network built and trained, you can test out your network. So here, you'd want to do a forward pass, get your logits, calculate the class probabilities, maybe output of your network, and then pass in one of these images from the test set and check out if your network can actually predict it correctly. If you want to see my solution, it's in the next notebook, part five, and you'll also see it in the next video. Cheers.

84. Again. So, in the last video, I hand you try out building your own neural network to classify this fashion in this dataset. Here is my solution like how I decided to build this. So first, building our network. So, here, I'm going to import our normal modules from PyTorch. So, nn and optim, so, nn is going to allow us to build our network, and optim is going to give us our optimizers. I must going to import this functional modules, so, we can use functions like ReLU and log softmax. I decided to define my network architectures using the class. So, in nn.modules subclassing from this, and it's called a classifier. Then I created four different linear transformations. So, in this case, it's three hidden layers and then one output layer. Our first hidden layer has 256 units. The second hidden layer has a 128, one after that has 64. Then our output has 10 units. So, in the forward pass, I did something a little different. So, I made sure here that the input tensor is actually flattened. So now, you don't have to flatten your input tensors in the training loop, it'll just do it in the forward pass itself. So, to do this is do x.view, which is going to change our shape. So, x.shape zero is going to give us our batch size. Then the negative one here is going to basically fill out the the second dimension with as many elements as it needs to keep the same total number of elements. So, what this does is it basically gives us another tensor, that is the flattened version of our input tensor. It doing a pass these through our linear transformations, and then ReLU activation functions. Then finally, we use a log softmax with a dimension set to one, as our output, and return that from our forward function. With the model defined, I can do model equals classifiers. So, this actually creates our model. Then we define our criterion with the negative log likelihood loss. So, I'm using log softmax as the output my model. So, I want to use the NLLLoss as the criterion. Then, here I'm using the Adam optimizer. So, this is basically the same as stochastic gradient descent, but it has some nice properties where it uses momentum which speeds up the actual fitting process. It also adjust the learning rate for each of the individual parameters in your model. Here, I wrote my

training loop. So, again I'm using five epochs. So, for e in range epoch, so, this is going to basically loop through our dataset five times, I'm tracking the loss with running loss, and just kind of instantiated it here. Then, getting our images. So, from images labels in train loader, so I get our log probabilities by passing in the images to a model. So, one thing to note, you can kind of do a little shortcut. If you just pass these in to model as if it was a function, then it will run the forward method. So, this is just a kind of a shorter way to run the forward pass through your model. Then with the log probabilities and the labels, I can calculate the loss. Then here, I am zeroing the gradients. Now I'm doing the lost up backwards to calculating our gradients, and then with the gradients, I can do our optimizer step. If we tried it, we can see at least for these first five epochs that are loss actually drops. Now the network is trained, we can actually test it out. So, we pass in data to our model, calculate the probabilities. So, here, doing the forward pass through the model to get our actual log probabilities and with the log probabilities, you can take the exponential to get the actual probabilities. Then with that, we can pass it into this nice little view classify function that I wrote, and it shows us, if we pass an image of a shirt, it tells us that it's a shirt. So, our network seems to have learned fairly well, what this dataset is showing us.

85. Hey there. So now, we're going to start talking about inference and validation. So, when you have your trained network, you typically want to use it for making predictions. This is called inference, it's a term borrowed from statistics. However, neural networks have a tendency to perform too well on your training data and they aren't able to generalize the data that your network hasn't seen before. This is called overfitting. This happens because as you're training more and more and more on your training set, your network starts to pick up correlations and patterns that are in your training set but they aren't in the more general dataset of all possible handwritten digits. So, to test for overfitting, we measure the performance of the network on data that isn't in the training set. This data is usually called the validation set or the test set. So, while we measure the performance on the validation set, we also tried to reduce overfitting through regularization such as dropout. So, in this notebook, I'll show you how we can both look at our validation set and also use dropout to reduce overfitting. So, to get the training set for your data like from PyTorch, then we say train equals true and for fashionMNIST. To get our test set, we're actually going to set train equals false here. Here, I'm just defining the model like we did before. So, the goal of validation is to measure our model's performance on data that is not part of our training set. But what we mean by performance is up to you, up to the developer, the person who's writing the code. A lot of times, it'll just be the accuracy. So, like how many correct classifications did our model make compared to all of the predictions? And other options for metrics are precision and recall, and the top five error rate. So here, I'll show you how to actually measure the accuracy on the validation set. So first, I'm going to do a forward pass that is one batch from the test set. So, see in our test set we get our probabilities. So, just 64 examples in a batch. Then 10 columns like one for each of the classes. So, the accuracy, we want to see if our model made the correct prediction of the class given the image. The prediction we can consider it to be whichever class has the highest probability. So, for this, we can use this top-k method on our tensors. This returns the k highest values. So, if we pass in one, then this is going to give us the one highest value. This one highest value is the most likely class that our network is predicting. So, for the first ten examples, and this batch of test data that I grabbed, we see that the class four and class five are what are being predicted for these. So, remember that this network actually hasn't been trained yet, and so it's just making these guesses randomly because it doesn't really know anything about the data yet. So, top-k actually returns a tuple with two tensors. So, the first tensor is the actual probability values, and the second

tensor are the class indices themselves. So typically, we just want this top class here. So, I'm calling top-k here and I'm separating out the probabilities in the classes. So, we'll just use this top class going forward. So, now that we have the predicted classes from our network, we can compare that with the true labels. So, we say, we can say like top class equals equals labels. The only trick here is that we need to make sure our top class tensor and the labels tensor has the same shape. So, this equality actually operates appropriately like we expect. So, labels from the test loader is actually a 1D tensor with 64 elements, but top class itself is a 2D tensor, 64 by one. So here, I'm just like changing the shape of labels to match the shape of top class. This gives us this equals tensor. We can actually see it looks like. So, it gives us a bunch of zeros and ones. So, zeros are where they don't match, and then ones are where they do match. Now, we have this tensor that's all just a bunch of zeros and ones. So, if we want to know the accuracy, right? We can just sum up all the correct things, all the correct predictions, and then divide by the total number of predictions. If you're tensor is all zeros and ones, that's actually equivalent to taking the mean. So for that, we can do torch.mean, but the problem is that equals is actually a byte tensor, and torch.mean won't work on byte tensors. So, we actually need to convert equals until a float tensor. If we do that, then we can actually see our accuracy for this one particular batch is 15.6 percent. So, this is roughly what we expect. So, our network hasn't been trained yet. It's making pretty much random guesses. That means that we should see our accuracy be about one in ten for any particular image because it's just uniformly guessing one of the classes, okay? So here, I'm going to have you actually implement this validation loop, where you'll pass in data from the test set through the network and calculate the loss and the accuracy. So, one thing to note, I think I mentioned this before. For the validation paths, we're not actually going to be doing any training. So, we don't need the gradients. So, you can actually speed up your code a little bit if you turn off the gradients. So, using this context, so with torch.no_grad, then you can put your validation pass in here. So, for images and labels in your test loader and then do the validation pass here. So, I've basically built a classifier for you, set all this up. Here's the training pass, and then it's up to you to implement the validation pass, and then print out the accuracy. All right. Good luck, and if you get stuck or want any help, be sure to check out my solution.

86. Welcome back. So, here's my solution for the validation pass. So, here, our model has been defined, our loss, and our optimizer, and all this stuff. I've set to 30 epochs so we can see like how this trains or how the training loss drops, and how the validation loss changes over time. The way this works is that after each pass, after each epoch, after each pass through the training set, then we're going to do a validation pass. That's what this else here means. So, basically, four of this stuff and then else basically says after this four loop completes, then run this code. That's what this else here means. As seen before, we want to turn off our gradients so with torch.no_grad, and then we're going to get our images and labels from a test set, pass it into the images into our model to get our log probabilities, calculate our loss. So, here, I am going to just be updating our test_loss. So, test_loss is just an integer that's going to count up our loss on our test set as we're training, as we're doing more of these validation passes. So, this way, we can actually track the test loss over all the epochs that we're training. So, from the law of probabilities, I can get our actual probability distributions using torch.exponential. Again, topk1 gives us our predicted classes and we can measure, we can calculate the equalities. So, here, we can get our probabilities from our log probabilities using torch.exp, taking the exponential of a log gives you back into the probabilities. From that, we do ps.topk, so one, and this gives us the top_class or predicted class from the network. Then, using checking for equality, we can see where our predicted classes match with the true

classes from labels. Again, measure our calculator accuracy. So, using torch.mean and changing equals into a FloatTensor. So, I'm going to run this and then let it run for a while, and then we can see what the actual training and validation losses look like as we were training this network. Now, the network is trained, we can see how the validation loss and the training loss actually changed over time like as we continue training on more and more data. So, we see is the training loss drops but the validation loss actually starts going up over time. There's actually a clear sign of overfitting so our network is getting better and better and better on the training data, but it's actually starting to get worse on the validation data. This is because as it's learning the training data, it's failing to generalize the data outside of that. Okay. So, this is what the phenomenon of overfitting looks like. The way that we combine it, the way that we try to avoid this and prevent it is by using regularization and specifically, dropout. So, deep behind dropout is that we randomly drop input units between our layers. What this does is it forces the network to share information between the weights, and so this increases this ability to generalize to new data. PyTorch adding dropout is pretty straightforward. We just use this nn.Dropout module. So, we can basically create our classifier like we had before using the linear transformations to do our hidden layers, and then we just add self.dropout, nn.Dropout, and then you give it some drop probability. In this case, this is 20 percent, so this is the probability that you'll drop a unit. In the forward method, it's pretty similar. So, we just pass in x, which is our input tensor, we're going to make sure it's flattened, and then we pass this tensor through each of our fully connected layers into an activation, relu activation and then through dropout. Our last layer is the output layer so we're not going to use dropout here. There's one more thing to note about this. So, when we're actually doing inference, if we're trying to make predictions with our network, we want to have all of our units available, right? So, in this case, we want to turn off dropout when we're doing validation, testing, when we're trying to make predictions. So, to do that, we do model.eval. So, model.eval will turn off dropout and this will allow us to get the most power, the highest performance out of our network when we're doing inference. Then, to go back in the train mode, use model.train. So, then, the validation pass looks like this now. So, first, we're going to turn off our gradients. So, with torch.no_grad, and then we set our model to evaluation mode, and then we do our validation pass through the test data. Then, after all this, we want to make sure the model is set back to train mode so we do model.train. Okay. So, now, I'm going to leave it up to you to create your new model. Try adding dropout to it and then try training your model with dropout. Then, again, checkout the training progress of validation using dropout. Cheers.

87. Hi. Here's my solution for your building and training this network using dropout now. Just like I showed you before, we can define our dropout module as self.dropout and then nn.dropout to give it some drop probability. So, in this case, 20 percent, and then just adding it to our forward method now on each of our hidden layers. Now, our validation code looks basically the same as before except now we're using model.eval. So, again, this turns our model into evaluation or inference mode which turns off dropout. Then, like the same way before, we just go through our data and the test say, calculate the losses and accuracy and after all that, we do model.train to set the model back into train mode, turn dropout back on, and then continue on in train smart. So, now, we're using dropout and if you look at again the training loss and the validation loss over these epochs that we're training, you actually see that the validation loss sticks a lot closer to the train loss as we train. So, here, with dropout, we've managed to at least reduce overfitting. So, the validation losses isn't as low as we got without dropout being is still, you can see that it's still

dropping. So, if we kept training for longer, we would most likely manage to get our validation loss lower than without dropout.

88.

89. In this video, I'll be showing you how to load image data. This is really useful for what you'll be doing in real projects. So previously, we used MNIST. Fashion-MNIST were just toy datasets for testing your networks, but you'll be using full-size images like you'd get from smartphone cameras and your actual projects that you'll be doing with deep learning networks. So with this, we'll be using a dataset of cat and dog photos, super cute. That come from Kaggle. So, if you want to learn more about it, you can just click on this link. So, you can see our images are now much larger, much higher resolution and they're coming in different shapes and sizes than what we saw with MNIST and fashion-MNIST. So, the first step to using these is to actually load them in with PyTorch. Then once you have them in, you can train a network using these things. So, the easiest way to load in our image data is with datasets.ImageFolder. This is from torchvision, that datasets module. So basically, you just pass in a path to your dataset, so into the folder where your data is sitting into image folder and give us some transforms, which we talked about before. I'll go into some more detail about transforms next. So, the image folder, it expects your files and directories to look like this, where you have some root directory that's where all your data. Then each of the different classes has their own folder. So in this case, we have two classes. We have dog and cat. So, we have these two folders, dog and cat. Get more classes like for MNIST, now you have ten classes. There will be one folder for each of the different digits, right? Those are our classes or labels. Then within each of the specific class folders, you have your images that belong to those classes. So, in your dog folder are going to be all of your dog pictures and the cat folder are going to be all of your cat pictures. So, if you're working in a workspace, then the data should already be there, but if you're working on your local computer, you can get the data by clicking here. I've also already split this into a training set and test set for you. When you load in the image folder, you need to define some transforms. So, what I mean by this is you'll want to resize it, you can crop it, you can do a lot of things like typically you'll want to convert it to a PyTorch tensor and it is loaded in as a pillow image. So, you need to change the image into a tensor. Then you combine these transforms into a pipeline of transforms, using transforms.compose. So, if you want to resize your image to be 255 by 255, then you say transforms.resize 255 and then you take just the center portion, you just crop that out with a size of 224 by 224. Then you can convert it to a tensor. So, these are the transforms that you'll use and you pass this into ImageFolder to define the transforms that you're performing on your images. Once you have your dataset from your image folder, defining your transforms and then you pass that to dataloader. From here, you can define your batch size, so it's the number of images you get per batch like per loop through this dataloader and then you can also do things like set shuffle to true. So basically, what shuffle does is it randomly shuffles your data every time you start a new epoch. This is useful because when you're training your network, we prefer it the second time it goes through to see your images in a different order, the third time it goes through you see your images in a different order. Rather than just learning in the same order every time because then this could introduce weird artifacts in how your network is learning from your data. So, the thing to remember is that this dataloader that you get from this class dataloader, the actual dataloader object itself, is a generator. So, this means to get data out of it you actually have to loop through it like in a for loop or you need to call iter on it, to turn into an iterator. Then call next to get the data out of it. Really what's happening here in this for loop,

this for images comma labels in dataloader is actually turning this into an iterator. Every time you go through a loop, it calls next. So basically, this for loop is an automatic way of doing this. Okay. So, I'm going to leave up to you is to define some transforms, create your image folder and then pass that image folder to create a dataloader. Then if you do everything right, you should see an image that looks like this. So, that's the basic way of loading in your data. You can also do what's called data augmentation. So, what this is is you want to introduce randomness into your data itself. What this can do is you can imagine if you have images, you can translate where a cat shows up and you can rotate the cat, you can scale the cat, you can crop different parts of things, you can mirror it horizontally and vertically. What this does is it helps your network generalized because it's seen these images in different scales, at different orientations and so on. This really helps your network train and will eventually lead to better accuracy on your validation tests. Here, I'll let you define some transforms for training data. So here, you want to do the data augmentation thing, where you're randomly cropping and resizing and rotating your images and also define transforms for the test dataset. So, one thing to remember is that for testing when you're doing your validation, you don't want to do any of this data augmentation. So basically, you just want to just do a resize and center crop of your images. This is because you want your validation to be similar to the eventual like in state of your model. Once you train your data, you're going to be sending in pictures of cats and dogs. So, you want your validation set to look pretty much exactly like what your eventual input images will look like. If you do all that correctly, you should see training examples are like this. So, you can see how these are rotated. Then you're testing examples should look like this, where they are scaled proportionally and they're not rotated. Once you've loaded this data, you should try to build a network based on what you've already learned that can then classify cats and dogs from this dataset. I should warn you this is actually a pretty tough challenge and it probably won't work. So, don't try too hard at it. Before you used MNIST and fashion-IMNIST. Those are very simple images, right? So, there are 20 by 28. They only have grayscale colors. But now these cat and dog images, they're much larger. Their colors, so you have those three channels. Just in general, it's going to be very difficult to build a classifier that can do this just using this fully connected network. The next part, I'll show you how to use a pre-trained network to build a model that can actually classify these cat and dog images. Cheers.

90. Hello, and welcome back. So, here are my solutions for the exercises I had you do on loading image data. Here, I had you define some transforms and then load the actual dataset with image folder and then turn that into a data loader using this torch utils data loader class. So, here, I chose a couple transforms. So, first, I'm resizing the images to be 255 by 255 squares. So, basically, even if your image is actually a rectangle, then this will resize it to be square with 255 pixels on each size. The first transform I used was resize. So, this resizes your images to be squares with 255 pixels on each side. So, even if your original image is a rectangle, this will change it into a square. Then, I did a center crop with 224 pixels. So, this crops a square out of the center of the image with 224 pixels on each side. Then, I convert it into a tensor which we can then use in our networks. With the transform defined, we can pass that into this image folder and along with the path to our dataset and that creates a dataset object. Then, with the dataset object, we can pass that to data loader. So, this will give us back a generator were we actually can get our images and labels. So, here, I just chose a batch size of 32 and this shuffle set to true. So, basically, every time you loop through the generator again like multiple times, every time you do that, it'll randomly shuffle the images and labels. So, that loaded, here's what it looks like. We have a nice little dogs now here. So, here, I had you define transforms for our training data and our testing data. So, like I was

saying before, with training data, you typically want to do data augmentation. So, that means rotating it, resizing it, flipping it, et cetera, to create this simulated dataset of more images than we actually have. Firstly, it just gives you more data to actually train with. But secondly, it helps the network generalize to images that aren't in the training set. So, my transformations here, I first chose to do a random rotation with 30 degrees. So, this is going to rotate in either direction up to 30 degrees. Then, I did a random resize crop. So, this is going to randomly resize the image and then take a crop from the center of 224 pixels square. Then, after that crop, then it do a random horizontal flip. So, it's going to mirror it horizontally and change it to a tensor. Then, with the test transforms kind of the same as before resize it to 255 pixels and then do a center crop 224, and it finally change it to a tensor. Then, here with the train data and test data, we can pass our data directories and our transforms through this image folder. I should actually load the data, and then give our loaded data to our data loaders to actually get our load our datasets so that we can see data from the train loader, it looks like this, and we can see data from our test loader, so like that.

91. Hello everyone, welcome back. So in this network, we will be using a pre-trained network to solve this challenging problem of creating a classifier for your cat and dog images. These pre-trained networks were trained on ImageNet which is a massive dataset of over one million labeled images from 1,000 different categories. These are available from torchvision and this module, torchvision.models. And so, we see we have six different architectures that we can use, and here's a nice breakdown of the performance of each of these different models. So, AlexNet gives us the top one error and the top five error. So basically, as you see, some of these networks and these numbers here, 19, 11, 34, and so on, they usually indicate the number of layers in this model. So, the larger this number, the larger the model is. And accordingly, the larger the model is, you get better accuracy, you get lower errors. At the same time, again, the larger the model is, the longer it's going to take to compute your predictions and to train and all that. So when you're using these, you need to think about the tradeoff between accuracy and speed. So, all these networks use an architecture called convolutional layers. What these do, they exploit patterns and regularities in images. I'm not going to get into the details but if you want to learn more about them, you can watch this video. So we're saying, these deep learning networks are typically very deep. So that means, they have dozens or even hundreds of different layers, and they were trained on this massive ImageNet dataset. It turns out that they were astonishingly well as future detectors for images that they weren't trained on. So using a pre-trained network like this on a training set that it hasn't seen before is called transfer learning. So basically, what's learned from the ImageNet dataset is being transferred to your dataset. So here, we're going to use transfer learning to train our own network to classify our cat and dog photos. What you'll see is you'll get really good performance with very little work on our side. So again, you can download these models from torchvision.models, this model here, so we can include this in our imports, right here. Most of these pre-trained models require a 224 by 224 image as the input. You'll also need to match a normalization used when these models were trained on ImageNet. So when they train these models, each color channel and images were normalized separately. And you can see the means here and the standard deviations here. So, I'm going to leave it up to you to define the transformations for the training data and the testing data now. And if you're done, we can get to a new one. Now, let's see how we can actually load in one of these models. So here, I'm going to use the Densenet-121 model. So you see, it has very high accuracy on the ImageNet dataset and it's one 121 tells us that it has 121 layers. To load this in our code and use it, so we just say model models.densenet121 and then we say pretrained equals true. So this is going to download the pre-trained network, the

weights, the parameters themselves, and then load it into our model. So now, we can do that and then we can look at what the architecture of this model. And this is what our DenseNet architecture looks like. So, you'll notice that we have this features part here and then a bunch of these layer. So this is like a convolutional layer which again I'm not going to talk about here but you don't really need to understand it to be able to actually use this thing. There's two main parts that we're interested in. So firstly, again, this features part, but then if we scroll all the way to the bottom, we also see this classifier part. So we see here is that we have the classifier. This has been defined as a linear combination layer, it's a fully connected dense layer, and it has 1,024 input features and then 1,000 output features. So again, the ImageNet dataset has 1,000 different classes. And so, the the number of outputs of this network should be 1,000 for each of those classes. So, the thing to know is that this whole thing was trained on ImageNet. Now, the features will work for other datasets but the classifier itself has been trained for ImageNet. So this is the part that we need to retrain, the classifier. We want to keep the feature part static. We don't want to update that, but we just need to update the classifier part. So then, the first thing we need to do is freeze our feature parameters. To do that, we go through our parameters in our model. And then, we just say, requires_grad equals false. So what this will do is that when we run our tensors through the model, it's not going to calculate the gradients. It's not going to keep track of all these operations. So firstly, this is going to ensure that our our feature parameters don't get updated but it also will speed up training because we're not keeping track of these operations for the features. Now, we need to replace the classifier with our own classifier. So here, I'm going to use a couple of new things. I'm going to use the sequential module available from PyTorch. And so, what this does, you basically just give it a list of different operations you want to do and then it will automatically pass a tensor through them sequentially. So, you can pass in an ordered dict to name each of these layers. So I'll show you how this works. So we want a fully connected layer, so I'll just name it FC1, and then that is a fully connected layer coming from 1,024 inputs and I'm going to say 500 for this hidden layer. And then we want to pass this through ReLu activation and then this should go through another fully connected layer and this will be our output layer. So, 500 to two, so we have cat and dog, so we want two outputs here. And finally, our output is going to be the LogSoftmax like before. Okay, and that is how we define the classifier. So now, we can take this classifier, just a classifier built from fully connected layers, and we can attach it to our model.classifier. So now, the new classifier that we built that is untrained is attached to our model and this model also has the features parts. The features parts are going to remain frozen. We're not going to update those weights but we need to train our new classifier. Now, if we want to train our network that we're using, this Densenet-121 is really deep and it has 121 layers. So, if we can try to train this on the CPU like normal, it's going to take pretty much forever. So instead, what we can do is use the GPU. GPUs are built specifically for doing a bunch of linear algebra computations in parallel and our neural networks are basically just a bunch of linear algebra computations. So if we run these on the GPU, they're done in parallel and we get something like 100 times increase speeds. In PyTorch, it's pretty straightforward to use the GPU. If you have your model, so model, the idea is that your model has all these parameters in there tensors that are sitting in your memory on your computer, but we can move them over to our GPU by saying model.cuda. So what this does is it moves the parameters for your model to the GPU and then all of the computations and the processing and are going to be done on the GPU. Similarly, if you have a tensor like your images, select images, if you want to run your images through your model, you have to make sure that the tensors that you're putting through your model or on the GPU if your model's on the GPU. So you just have to make those match up. So to do that, to move a tensor from computer to the GPU, you just, again, say

images.cuda. So that will move a tensor, that's images, to the GPU. Then oftentimes, you'll want to move your model and your tensors back from the GPU to your local memory and CPU, and so, to do that, you just say like model.cpu or images.cpu, so this'll bring your tensors back from the GPU to your local computer to run on your CPU. Now, I'm going to give you a demonstration of how this all works and the amazing increased speed we get by using the GPU. So here, I'm just going to do for cuda and false, true. So this way, I'm going to be able to basically like loop through and try it once where we're not using the GPU, and once where we are using the GPU. So let's define my criterion which is going to be natural log_loss like we'd normally do, define our optimizer. So again, here, remember that we only want to update the parameters for the classifier. So we're just going to pass in model.classifier.parameters. This will work and that it's going to update the premise for our classifier but it's going to lead the parameters for the feature detector part of the model static. So I typically do is, say like, if cuda, then we want to move our model to the GPU. Otherwise, let's leave it on the CPU. And then I'm going to write a little training loop. We'll get our inputs and our labels, changes into variables like normal, then again, if we have cuda enabled, so if we have GPUs, then we can do inputs, labels, and we'll just move these over to the GPU. We're using the GPU now and we're also using this pre-trained network, but in general, you're going to do the training loop exactly the same way you have been doing it with these feed forward networks that you've been building. So first, I'm actually going to define a start time just so I can time things, then you just do your training pass like normal. So, you just do a forward pass through your model and you can calculate the loss, do your backward pass. Finally, update your weights with your optimizer. So I'm going to do here, I'm going to break this training loop after the first three iterations. So I want to time the difference between using a GPU and not using the GPU. What happens is the very first batch to go through the training loop tends to take longer than the other
batches, so I'm just going to take the first three or four and then average over those just so we get a better sense of how long it actually takes to process one batch. So, that will just print out our training times. So we can see that if we're not using the GPU, then each batch takes five and a half seconds to actually go through this training step. Whereas, with the GPU, it only takes 0.012 seconds. So, I mean, this is a speedup of over 100 times. So here, I basically set cuda manually but you can also check if a GPU is available so you say torch.cuda is available, and this will give you back true or false depending if you have a GPU available that can use cuda. Okay, so from here, I'm going to let you finish training this model. So you can either continue with a DenseNet model that is already loaded or you can try ResNet which is also a good model to try out. I also really like VGGNet, I think that one's pretty good. It's really up to you. Cheers.

92. Hi everyone, here is my solution for the transfer learning exercise that I had to do. So, this one's going to be a little different. I'm going to be typing it out as I do it so you can understand my that process is kind of the combination of everything you've learned in this lesson. So, the first thing I'm going do is, if I have a GPU available, I'm going to write this code in agnostic way so that I can use the GPU. So, what I'm going to say, device = torch.device and then this is going to be cuda. So, it's going to run on our GPU if torch.cuda is available, else CPU. So, what this will do is, if our GPU is available, then this will be true and then we'll return cuda here and then otherwise, it'll be CPU. So, now we can just pass device to all the tensors and models and then it will just automatically go to the GPU if we have it available. So, next, I'm going to get our pre-trained model. So, here I'm actually going to use ResNet. So, to do this, model dot models. So, we already imported models from torch vision then we can kind of like took out all the ones they have.

So, there's ResNet there. So, I'm just going to use a fairly small one, ResNet 50 and then we want pre-trained true and that should get us our model. So, now if we look, so we can just print it out like this and it will tell us all the different operations and layers and everything that's going on. So, if we scroll down, we can see that at the end here has fc. So, this is the last layer, this fully connected layer that's acting like a classifier. So, we can see that it has, it expects 2,048 inputs to this layer and then the out features are 1,000. So, remember that this was trained on ImageNet and so ImageNet is typically trained with 1,000 different classes of images. But here we're only using cat and dog, so we just need to output features and in our classifier. So, we can load the model like that and now I'm going to make sure that our models' perimeters are frozen so that when we're training they don't get updated. So, I'll just run this make sure it works. So, now, we can load the model and we can turn off gradients. Turn off gradients for our model. So, then, the next step is we want to define our new classifier which we will be training. So, here, we can make it pretty simple. So, models= nn.sequential. You can define this in a lot of different ways, so I'm just using an industrial sequential here. So, our first layer, so linear, so remember we needed 248 inputs and then let's say, let's drop it down to 512 at a ReLu layer, a dropout. Now our output layer, 512 to two and then we're going to do log softmax. I should change this to be a classifier. Okay. So, that's to finding our classifier and now we can attach it to our model, so to say, model.fc= classifier. Now, if we look at our model again, so we can scroll down to the bottom here. So, we see now this fully-connected module layer here is a sequential classifier linear operation ReLu, dropout, another linear transformation and then log softmax. So, the next thing we do is define my loss, my criterion. So, this is going to be the negative log like we had loss. Then, define our optimizer, optim.Adam. So, we want to use the parameters from our classifier which is fc here and then set our learning rate. The final thing to do is to move our model to whichever device we have available. So, now we have the model all set up and it's time to train it. Here's is the first thing I'm going to do is define some variables that we're going to be using during the training. So, for example, I'm going to set our epochs, so I'm going to set to one. I'll be tracking the number of train steps we do, so set that to zero. I'll be tracking our loss, so also set this to zero, and finally, we want to kind of set a loop for how many steps we're going to go before we print out the validation loss. So, now we want to loop through our epochs. So, for epoch and range epochs. Now, we're going to loop through our data for images, labels in trainloader, cumulate steps. So, basically, every time we go through one of these batches, we're going to increment steps here. So, now that we have our images and our labels, we're going to want to move them over to the GPU, if that's available. So, we're just going to do is images.to (device), labels.to(device). Now we're just going to write out our training loop. So, the first thing we need to do is zero our gradients. So, it's very important, don't forget to do this. Then, get our log probabilities from our model, model passed in the images with the log probabilities, we can get our loss from the criterion in the labels. Then do a backwards pass and then finally with our optimizer we take a step. Here, we can increment are running loss like so. So, this way we can keep track of our training loss as we are going through more and more data. All right. So, that is the training loop. So, now every once in a while so which is set by this like print every variable. We actually want to drop out of the train loop and test our network's accuracy and loss on our test dataset. So, for step modulo print_every, this is equal to zero, then we're going to go into our validation loop. So, what we need to do first is set model.eval. So, this'll turn our model into evaluation inference mode which turns off dropout. So, we can actually accurately use our network for making predictions instead a test loss and accuracy. So, now we'll get our images and labels from our test data. Now we'll do our validation loop. So, with our model so we'll pass in the images. So, these are the images from our test set. So, we're going to get our logps from our test set so again, get the loss with our criterion and keep track of our loss to test loss plus+= loss.item. So, this will allow us to keep track of our test loss as we're going through these validation rules. So, next we want to calculate our accuracy. So, probabilities= torch.exponential(logps). So, remember that our model is returning log softmax, so it's the log probabilities of our classes and to get the actual probabilities, we're going to use torch.exponential. So we get our top probabilities and top classes from ps.topk(1). So, that's going to give us our first largest value in our probabilities. Here, we need to make sure we set dimension to one to make sure it's actually like looking for the top probabilities along the columns. Go to the top classes, now we can check for equality with our labels and then with the equality tensor, we can update our accuracy. So, here remember we can calculate our accuracy from equality. Once we change it to a FloatTensor then we can do torch.mean and get our accuracy and so again just kind of incremented accumulated into this accuracy variable. All right. Now, we are in this loop here so this four step every print_every. So basically, now we have a running loss of our training loss and we have a test loss that we passed our test data through our model and measured the loss in accuracy. So now we can print all this out and I'm just going to copy and paste this because it's a lot to type. So, basically here, we're just printing out our epochs. So, we can keep track and know where we are and keep track of that. So, running_loss divided by print_every so basically we're taking the average of our training loss. So every time we print it out, we're just going to take the average. Then, test_loss over length the testloader. So basically length test loader tells us how many batches are actually in our test dataset that we're getting from testloader. So, since we're we're summing up all the losses for each of our batches, if we take the total loss and divide by the number of batches and that gives us our average loss, we do the same thing with accuracy. So, we're summing up the accuracy for each batch here and then we just divide by the total number of batches and that gives us our average accuracy for the test set. Then at the end, we can set our running loss back to zero and then we also want to put our model back into training mode. Great. So, that should be the training code and we'll see if it works. Now, this should be an if instead of a for. So, here I forgot this happens a lot. I forgot to transfer my tensors over to the GPU. So, hopefully this will work. All right. So, even like pretty quickly, we see that we can actually get our test accuracy on this above 95 percent. So, this is, remember that we're printing this out every five steps and so this is a total of 15 batches, training batches that were updated in the model. So, we're able to easily fine tune these classifiers on top and get greater than a 95 percent accuracy on our dataset.

## Tips, Tricks, and Other Notes

### Watch those shapes

In general, you'll want to check that the tensors going through your model and other code are the correct shapes. Make use of the .shape method during debugging and development.

### A few things to check if your network isn't training appropriately

Make sure you're clearing the gradients in the training loop with optimizer.zero_grad(). If you're doing a validation loop, be sure to set the network to evaluation mode with model.eval(), then back to training mode with model.train().

### CUDA errors

Sometimes you'll see this error:

RuntimeError: Expected object of type torch.FloatTensor but found type torch.cuda.FloatTensor for argument #1 ‘mat1’

You'll notice the second type is torch.cuda.FloatTensor, this means it's a tensor that has been moved to the GPU. It's expecting a tensor with type torch.FloatTensor, no .cuda there, which means the tensor should be on the CPU. PyTorch can only perform operations on tensors that are on the same device, so either both CPU or both GPU. If you're trying to run your network on the GPU, check to make sure you've moved the model and all necessary tensors to the GPU with .to(device) where device is either "cuda" or "cpu".

Notebook-folder-intro-to-pytorch-ends

Not_read_end