[SQUEAKING]

[RUSTLING]

[CLICKING]

JAMISON MEINDL: So I guess we can get started. We're going to be going over some of the basics of PyTorch today, and you guys are going to be using PyTorch many times with all of your psets. And I think if you keep doing deep learning in the future, everyone's using PyTorch. And so lots of projects, research projects and real industry projects, are actually using PyTorch.

So there's a few things to know about PyTorch. And a few of the more important things are that PyTorch is fast. So we're able to run PyTorch on GPU. And this allows us to actually complete powerful deep learning models by actually running these things within a reasonable time frame. So GPUs really speed up the ability to perform operations that we use within these machine learning models, and PyTorch supports that ability. So you may have used something like NumPy before, where PyTorch and NumPy have similar data operations, but we're able to use GPU and speed up within PyTorch.

The other thing about PyTorch is it supports autograd. So we talked about gradients a little bit in lecture today. But autograd is basically the idea that we can compute the gradients automatically as we iterate through our models. And so PyTorch makes this very easy for us. It makes it very intuitive and simple for us to actually train and update the weights of our models. So that's a few reasons of why we use PyTorch. It's a great library, and I think you guys will hopefully enjoy it.

So today, just a quick overview of what we're going to do, I'm going to go over some of intro stuff of just how we use PyTorch tensors and some of the simple operations on these tensors. And then we're going to go through a little bit about autograd, and then a little bit about how we can train a model. You'll go into a lot more depth on some of the psets, but overall, this should give you a good start to be able to understand how it actually works behind the scenes and how you're going to actually be able to use it well on your psets.

So just to start off, just how many people have used PyTorch? Maybe you could raise your hand if you've used PyTorch. OK, so we definitely have a lot of newbies to PyTorch, I guess. And that's totally fine. I think you guys will all be able to pick it up and be able to be totally fine throughout the course.

So yeah, let's get started. If you guys want to pull up this notebook, you can. It's on the Piazza. I posted it. It's probably one of the latest posts. And yeah, we'll get started.

So the first thing we have to do is we have to do a few imports. PyTorch can be imported by importing torch here. And then we'll import a few other things as well just to help with this visualizing and creating tensors.

So the first topic we're going to go over is PyTorch tensors. And PyTorch tensors is basically where we can actually store data within PyTorch. So if you ever had some input data that you wanted to put into a machine learning model, or if you wanted to compare some outputs, these outputs are going to be represented as PyTorch tensors.

And so the first thing we'll go over is how we can actually initialize a tensor. So the first way we can do this here is just directly from data. You're probably not going to actually do this too often. But it's good to know all the different ways that you can do. So if we just have a Python list here where we have the inputs 1, 2, 3, and 4, we can create a tensor directly from that data. So if we run that cell, we see that we create a tensor object here, which is the same structure as that nested list that we saw before.

The next way we can do this is we can also transform and create tensors from a NumPy array. So do we have a question back there?

AUDIENCE: [INAUDIBLE]

JAMISON MEINDL: Oh, you can't see? Is that a little better? One more? Good? All right, so yeah, we can create tensors from NumPy arrays. NumPy arrays are obviously another common way we can store data in Python. And tensors and NumPy arrays store a lot of the same structure. So we can copy directly from NumPy arrays. And it actually can use the same memory in the system for both the tensor and NumPy array. So if we do the same thing here, we're copying the same data. And we can create a tensor with this torch.from_numpy function.

The next way we can create tensors is if we wanted to create a tensor with a specific shape and specific values within it, we might want to create a tensor of all ones or a tensor with random numbers to initialize some of model. And what we can do here is just use this ones_like if we wanted to make a tensor of the same shape as another object. But there's also operations-- I think I have that-- that's next, where you can input a specific shape. And then rand_like will give you a random number between 0 and 1 in each element.

Moving on, yeah, we can create tensors of specific shapes. Here, I'm using rand, ones, zeros, and you can also-- there's more that you can find in the documentation of different types of initializations of each element that you can do.

So yeah, we can also define tensors with a specific shape. So if you didn't already have a tensor defined that you were interested in the shape, you could, say, create a tensor of size 2 by 3. And we can do that in a few different ways here. We can do random. We can do ones, zeros. There's a few other options. There's a few attributes of a tensor that are good to know. So the first one that we've seen already is the shape. So if we make a random tensor of size 3 by 4, we can get the shape by doing .shape.

The next thing to know is the data type. So most of the tensors you're probably going to be using and that we use as inputs to machine learning models are going to be float32. So that's a 32-bit float. But you might see integers as well in some situations. But the float 32 is generally used because it's the fastest. And GPU hardware is optimized to use that. The last thing you see here is the device. We get the device the tensor is stored on by doing .device. And this is important if you're using both CPU and GPU.

So we can only do operations between tensors that are on the same device. So if you have one tensor that's on a CPU and one tensor that's on a GPU, you won't be able to do any operations between the two. You'd have to move one to the other device before you can actually do any operations between them. And so you will probably be frequently be finding out which device your tensors are on and then moving them to the device you actually want them to be on if you wanted to do some operation between the two.

The next thing we're going to talk about is just some of the basic operations that we can perform on tensors. So we can do things like, to modify a tensor or create a new tensor based on the tensor input, and these are pretty simple operations that include things like adding tensors together or multiplying them that we can do to get different data that we want.

So if we look at an example, the first thing we'll do is let's just create some of input tensor. And this notation here is basically going to create a tensor of 101 equally-spaced numbers from 0 to 5. And so that might be something if we're interested in seeing the effect along a specific range in this case 0 to 5. And if we wanted to see that worked, we can print the first five things in x by just indexing in the same way we would in NumPy or other Python environments. So we can also get the last ones by doing negative 5 to the end, which you may have seen in other Python libraries.

So let's try and perform some operations on this tensor. So if we wanted to, say, graph a sine wave or cosine, we could do x.sin. And that will perform sine on each input in the tensor. We can also do something more complex like doing x to the power of x cosine to create a more interesting output.

We can also subtract tensors. So we can subtract those two outputs or just find the specific value within that tensor. So if we wanted to find the min, you could also find max, mean, standard deviation, stuff like that. If you wanted to find some characteristic about a tensor, we can also use tensors with matplotlib. So maybe some of you have used matplotlib in other cases. And we can directly input these tensors and create plots with matplotlib. So we see here that some of our transformations of x from 0 to 5, and we can see how those different transformations occurred.

So there's a lot of different operations we can do. I'm obviously not going to be able to go through all the different operations, but just know that you can apply some of these vector operations to tensors, and you'll just get a tensor as the output.

Another thing that might be important is you can clamp tensors. So if you, say, were worried about your outputs going too far in some direction, you can use .clamp and then some specific range to actually get a specifically clamped output. So we can see this plot here of that kind of transformation. And so yeah, there's lots of different transformations that we can do. I'm not going to be able to go through all of them, but just know that if you want to do some of operation or transformation on a tensor, there's probably some ability to do it. And you'll just have to look up the documentation.

Moving on, we're going to start talking about how we can deal with multidimensional data. So we're going to be dealing with, frequently, some sort of data input that's in a batch. And I think we talked about batches in lectures today. But basically, we create a batch of some of inputs. And generally, what we do to represent that is we include the batch as the first dimension. And then this example is for an image. So you might do the channel of R, G, and B as the second dimension, and then the rest of your data, and so on. But in general I think the most important part there is that the batch index is generally the first dimension.

Moving on, we can look at some of these tensors of higher dimensions and see how we can play with their shapes and see how this affects things. So if we create a random tensor, we're going to do this shape representing two images here. So we have two RGB images that are 5 by 4. And we can create this random-- we'll just say random numbers. They're obviously not following RGB format, but that's OK for this example.

One thing we can do is permute dimensions. So if we were interested in swapping the dimensions x and y within our image, we can use this permute command. And so what you do is you basically just order the dimensions how you want them to be ordered. So if you wanted to keep the batch size the same, and then the RGB dimension the same, we'd say 0 and 1 because those are indexed. And then if we wanted to swap, say, the x and y, we'd just swap the dimensions that they were originally in our permute statement.

So if we run that, we see that we originally had a size of 2, 3, 4, 5, but in this case, we've swapped the 4 and the 5 with our .permute command.

Another thing you might frequently do is reshaping. And so there's a few different ways to do this. The first is with .view. And this is generally the more efficient way. But there's also .reshape, and it has pretty much the same functionality. And basically, what we can do is we can input specific shapes that we want. So in this case, we're just keeping the batch size. And then the negative basically just means that we're going to have all of the rest of the data that we had included in one dimension. So we end up with a shape of just 2 and 60, because this negative 1 was just the rest of the data in each-- the rest of those dimensions.

Next thing to know is indexing. So PyTorch does indexing pretty much the same way as NumPy does. So if we wanted to, say, get something like the first row, we can just input a specific number. So that would be 0. And if we wanted to get the first column, we can use this colon to represent all of the rows and then the 0 to represent the first column. And so that's the second dimension. And then if we want to get the last thing, similar to other Python methods, you can just use negative 1.

We can also use this slicing to assign values. So if we did wanted to assign the first column or, I guess, this second column to 0, we could do this using this assignment. The next thing we might want to do is join tensors together. So we might want to combine different maybe batches or just specific elements to create a larger tensor. And so we'll go through two different ways here.

The first is just concatenating. So we use this torch.cat operator. Then we input our tensors in a list here. We also define a dimension that we want to concatenate along. And basically, what this will do is it will just add all of these input tensors together along that specific dimension. So we see here we have this tensor that's a 4 by 4. And if we concatenate the three tensors together along the first dimension dim equals 1. We're going to end up with a 4 by 12. And you can see how this works out with our second column and our tensor, all 0's.

The next thing you can do is torch.stack. And instead of concatenating along one dimension, you're going to actually create a new dimension. So say if you had a few different inputs and you wanted to create a batch, you might do a stack along dimension equals 0, and you'll end up with an additional dimension at the start here.

The next thing we can do is things like matrix multiplication. So there's a few different ways we can do this. All of these operations here do the same thing. I think using the at symbol is probably the easiest way, but you can pick whatever you would like. And this just performs matrix multiplication, as you would expect. And so, in this case, we made this tensor, and we'll just apply matrix multiplication to it.

We can also compute element-wise products. So that will just be the multiplication symbol or this tensor.mul. And so that will do just an element-wise product. So if we had this tensor of 1, 2, 3, 4, we'll just end up with the squares element-wise product.

Next thing to know is we can use some aggregators to find some different statistics or values within our tensors. So we can do things like sum, mean, max, min, stuff like that, and get the value along a specific dimension or along the whole tensor. So that concludes the overall tensor part of this. Definitely going a little slower than I was hoping, but we'll try and get through as much as we can.

So the next thing to talk about is the autograd of PyTorch. And autograd is one of the most important parts of PyTorch because it allows us to actually compute gradients and update our models so that they can actually learn and produce reasonable outputs. And so we can first use autograd to just calculate a simple gradient of a vector. So I think we saw a similar example here before, where we made this x, which is the values between 0 and 5. And then we applied an operator on this. So we're squaring it, and then taking the cosine here.

And we can find the derivative of this by using the autograd features of PyTorch. And so if we want to take a derivative, we actually need to have a scalar as an output. So we take the sum here, and then we can use this torch.autograd of that output with respect to x to get our actual derivative here, dy by dx.

And so this is the backend, how you could specifically calculate one specific gradient or one derivative here. But when we're actually doing a model, PyTorch takes care of exactly which derivatives we need. And we don't have to do these really specific calculations of which differential we actually want.

But we can see in this plot here that we have our x squared cosine, where we end up with a shorter period as we go to the right. And then we can see our gradient here, where it's the correct gradient of that value. We can do this with another function here. So if you have a polynomial and we want to compute the gradient, we can do this in the same way as we previously did. And we can run this, and we can see the result is the same. We end up with our plot, and we have our gradient of our plot as well.

So these are this really specific way we can calculate this exact gradient. But PyTorch allows us to actually calculate all the gradients all together. And so we don't need to specifically calculate one gradient. Instead, we can just do a specific command, and we will get all the gradients that we want. And so that's what you'll actually be doing if you're running a PyTorch model. You won't need to calculate and store each different part individually.

And so if we look at this, we have the same example as we had before, but instead of doing our autograd and using the brackets, as we saw before, we can just use this .backward, and that will store the gradient of each part within the tensor. And so if we just get the use .grad, we can actually get the specific gradient using the .backward command. So you'll never do the autograd.grad that we had up here. You'll always use the .backward to just calculate all the gradients that you need.

So yeah, as I was mentioning, if you aren't going to use the gradient, you shouldn't calculate it. And we can use-- I don't know if I have it listed here, but you would say torch.no_grad, and that would allow you to not use the gradient and speed up your computation. Oh yeah, here we go with torch.no_grad that would be what we would use.

Next thing we're going to talk about is optimizers. So we figured out how we can calculate a gradient. But the next thing we need to do is actually use that gradient to optimize some function, some loss function, something like that.

So this naive gradient descent-- obviously, it works for this scenario, but in a lot of cases we want to be more adaptive and speed up our training. So I think we saw stochastic gradient descent in lecture. But this is another method. We import this optimizer. And if we do optimizer.step it takes care of the gradient update. So it will use the gradient, and it will update the weights of our model.

So if we run this, we see, again, our model minimizes. Maybe it got to the minimum a little faster, and then it's jumping around a little bit. This is just some little toy examples, so we're not too concerned exactly about what's happening.

The last optimizer here we're going to look at is Adam. So Adam is a pretty common optimizer. And it adaptively does a lot of the steps. And it generally is the most efficient. So a lot of people use that in real scenarios as opposed to just naive gradient descent or stochastic gradient descent. So we can run this one. It's the same structure as the SGD. But we see in this example, it's not quite doing as well as some of the simpler ways, but in general, this is a pretty good method of optimizing.

I think we can try and get through a quick bit of neural networks and how we can use PyTorch for neural networks. So obviously, that's a big part of deep learning is neural networks. And so there's a lot of modules within PyTorch that we can use to create networks from just really simple MLPs to more complex models. So the first building block that we see here is the linear layer. And so this is basically just going to do a transformation from one dimension to another dimension with a linear transformation. So we can generate that, and we can set the number of input features, the number of output features, along with a bias term as well.

And so we could just run an input through our linear network. We put an input of size 3 and get an output of size 2. So yeah, then we can pass an input to that, and we see we had an input of 3. And then we end up with an output of size 2. We can also use a batch. So we have four inputs of size 3 here. And natively, it will just work to support that, and we get four outputs of size 2.

If we can take a look at some of the weights and bias of this linear layer, we can just get the actual numbers. And so these are the numbers that it's going to multiply the inputs by to actually get an output. And the bias terms are added to the outputs. And you'll note the requires_grad=True on this tensor here, which shows that it's calculating the gradient as you go along. We can set specific things like the bias to 0 if we wanted to. And then we can see that our outputs will not add that bias term at the end.

We can look at some of the parameters. Just if we do this net.name_parameters, and that will get the exact values within those weights as well. And we can see that we set it to 0, and the bias term is now 0.

We can look at a few different features, the shapes of our weights. We can load them. So yeah, we saved the weights, and then we can load the weights here. Sorry, I'm trying to go-- let's see if we can get to something a little more interesting.

Maybe if we wanted to calculate a loss, so if we're trying to optimize this linear layer, and let's say we wanted to just-- kind of a dumb example, but if we wanted to just always output 1 comma 1, we could try and train our linear layer to always output 1 comma 1. So if we see here, first, we get an output. And it will calculate this loss, where we're doing squared loss here. And we could use the loss.backward to get the gradients. And then we could use those gradients to update our linear layer.

So here, if we wanted to actually train this model to learn to always output 1 comma 1, we could do an update loop. So here we're just looping doing a batch input and then calculating the loss. We're getting the gradients here with loss.backward. And then we're just using gradient descent to update our parameters. And we can see if we run this, we should get a graph of what our loss is. So we end up with the following weights. You can see the weights within the linear layer just went to 0, and our bias went to 1. And so our loss went to near 0 because we've learned to always output 1 comma 1.

I'll try and work through maybe one more part here, but feel free to go if you need to go. So the last thing I want to talk about is that we can create a more complex network using this sequential command. This will combine different layers together. So here, we see a combination of linear and ReLU layers. So ReLU is the activation that we've seen in lecture. And then we can combine this together with this sequential framework. And so this actually creates a little bit of a more interesting network that we can use to create outputs.

So we can see the different parts of this. We have sequential parts, we have linear parts, ReLU. And then, continuously, we're creating a network-- I think this one is 3 layers in size. So we can use this to train a more interesting model. So here, what we're trying to do is we're trying to train a model that takes an input in the space-- a 2D input in the space negative 2 to 2. And it's going to try and output-- try and classify points as either above a sine wave or below a sine wave.

And so what we do is we want to use the Adam optimizer. Here, we have a classify target, which is going to say is our input either above or below the sine wave. And that's going to give us our loss when we use this crossentropy because we can actually generate what the actual target is.

So what we do in this loop is we're iterating. We're generating a batch of random inputs. We're classifying those random inputs as either above the sine wave or below the sine wave. And then we're calculating a loss using crossentropy, which is a loss typically used for classification model. And basically, it's going to give a higher value if your classification result is worse and a lower value if it's better. And then we continue with this iteration. We're doing loss.backward to get our gradients, and we're doing an optimizer step. And then the last bit here is just giving us some information about our accuracy.

And so if we run this-- oh, I didn't create the-- sorry, must have not ran another cell above. Must have not ran a few cells above. But basically, if we want to run this, we're going to be able to train a model that is able to classify these points as either above the sine wave or below the sine wave. Let me see if I can find where that's not loaded. Not sure where that is. But yeah, I think-- it's a little strange. I think, yeah, we'll just leave that there.