**Introduction to deep learning with PyTorch**

Hello, I'm Maham. Welcome to this introduction to deep learning with PyTorch. Together, we'll learn about deep learning, a subfield of machine learning that has gained immense popularity over the last decade.

**What is deep learning?**

Deep learning algorithms power many recent and exciting innovations, from language translation to self-driving cars, medical diagnostics and chatbots. Deep learning outperforms traditional machine learning by extracting intricate patterns from "big" and unstructured data, such as image pixels, large corpora of text, and audio signals. Traditional machine learning relies on hand-crafted feature engineering. Deep learning models, on the other hand, using a "layered" network structure, are able to discover features from raw data, giving them that edge over traditional machine learning. This is known as feature learning or representation learning.

So what is deep learning? Deep learning is a subset of machine learning, where the fundamental model structure is a network of inputs, hidden layers, and outputs, as shown. A network can have one or many hidden layers. The original intuition behind deep learning was to create models inspired by how the human brain learns: through interconnected cells called neurons. This is why we continue to call deep learning models "neural" networks. These layered model structures require far more data to learn than other supervised learning models to derive patterns from the unstructured data in the manner we discussed. We are usually talking about at least hundreds of thousands of data points.

**PyTorch: a deep learning framework**

While there are several frameworks and packages out there for implementing deep learning algorithms, we'll focus on PyTorch, one of the most popular and well-maintained frameworks. In addition to being used by deep learning engineers in industry, PyTorch is a favored tool amongst researchers. Many deep learning papers are published using PyTorch. PyTorch is designed to be intuitive and user-friendly, sharing a lot of common ground with the Python library NumPy.

**Importing PyTorch and related packages**

The PyTorch module can be imported by calling import torch. PyTorch naturally supports tabular data in torch, but also comes with a suite of tools to support more unstructured data, including image data with torchvision, audio data with torchaudio,

**Tensors: the building blocks of networks in PyTorch**

The fundamental data structure in PyTorch is called a tensor. A tensor is essentially an array, which can support many mathematical operations, and will form a building block for our neural networks. Tensors can be created from Python lists by using the torch.tensor() class. PyTorch also supports tensor creation directly from NumPy arrays, using torch.from\_numpy(). torch.tensor() will work directly on NumPy arrays too. Like NumPy arrays, tensors are multidimensional, representing a collection of elements arranged in a grid with multiple dimensions.

**Tensor attributes**

Tensors have a variety of attributes. We can call tensor.shape() to display the shape of our newly created tensor, and tensor.dtype() for its data type, here, a 64-bit integer. tensor.device displays which device the tensor is loaded on, such as a CPU or GPU. Deep learning often requires a GPU, which, compared to a CPU, can offer parallel computing capabilities, faster training times, and better performance.

**Getting started with tensor operations**

PyTorch tensors support several operations similar to NumPy arrays. We can add or subtract tensors, provided that their shapes are compatible. When shapes are incompatible, we get an error.

**Getting started with tensor operations**

We can also perform element-wise multiplication, which involves multiplying each corresponding element from two arrays of the same shape, and many other operations, such as tensor transposition, matrix multiplication, and tensor concatenation. We'll get into these later. Most operations available for NumPy arrays can be performed on PyTorch tensors.

**Creating our first neural network**

00:00 - 00:10

Let's create some neural networks! We'll explore network structures, and come to grips with how networks take in input, perform computations, and return output.

**Our first neural network**

We saw that at their core, neural networks are stacked inputs, hidden layers, and outputs.

While a network can have any number of hidden layers, we'll begin by building a basic, two-layer network with no hidden layers.

We'll leverage the torch.nn() package extensively to create our networks. The first layer of this network will be an input layer. Say we create an input\_tensor of shape 1 by 3 as shown. We can think of this as one row with three "features" or "neurons".

Next, we pass this input\_tensor to a special kind of layer called a linear layer. A linear layer takes an input tensor, applies a linear function to it, and returns an output. nn.Linear() takes two arguments: ***in\_features***, which is the number of features in our input (three), and ***out\_features***, specifying the desired size of the output tensor (in this case two). Correctly specifying in\_features ensures our linear layer can receive the input\_tensor.

Lastly, we pass input\_tensor to linear\_layer to generate an output. Notice that the output has two features or neurons, due to the out\_features specified in our linear\_layer.

**Getting to know the linear layer operation**

Consider the linear\_layer object we created. Each linear layer has a set of weights and biases associated with it.

Now, the question arises: what operation does nn.Linear() perform? When input\_tensor is passed to linear\_layer, the linear operation performed is a matrix multiplication of input\_tensor and the weights, followed by adding in the bias. Put formally, for input X, weights W0 and bias b0, the linear operation performed is shown. The @ operator denotes a matrix multiplication. We won't perform this calculation manually, since nn.Linear() takes care of it. Initially, when we call nn.Linear(), weights and biases are initialized randomly, so they are not yet useful. Later in the course, we'll tune these weights and biases so that our linear operation output is meaningful.

**Our two-layer network summary**

And just like that, we built our two-layer network! It took a 1 by 3 input as the first layer, a linear layer with specific arguments as the second layer, and returned a 1 by 2 output. Note that networks with only linear layers are called "fully connected networks". Linear layers have connections (or arrows) between each input and output neuron, making them fully connected.

**Stacking layers with nn.Sequential()**

So far we've worked with one input layer and one linear layer. What if we wanted to stack multiple layers? ***Meet nn.Sequential()***, a PyTorch container that allows us to stack multiple neural network modules in sequence. The code provided shows three linear layers stacked within nn.Sequential(). This model takes input, passes it to each linear layer in sequence, and returns output. The first layer takes input with ten features and outputs a tensor with 18 features. The second layer takes an input of size 18 (the output size of the first layer) and outputs a tensor of size 20. The final layer takes input with the second layer's output size (20) and outputs a tensor of size five.

Say we have one row of data called input\_tensor, with ten "features", or neurons. Recall we set the first argument of our first linear layer to ten, to be compatible with this shape. We now pass input\_tensor to our multi-layer model to obtain output, just as we did before using a single linear layer. Again, output is not meaningful until each layer has tuned weights and biases.

**Discovering activation functions**

We've seen linear transformations on input layers. In this video, we'll add non-linearity to our models using activation functions.

**Stacked linear operations**

So far, we have seen neural networks made only of linear layers. Recall that each linear layer multiplies its input by its weights and adds biases. Two or more linear layers stacked in a row still effectively perform a linear operation.

**Why do we need activation functions?**

However, linear layers are not the only layer type we can add to a network. We can use activation functions to add non-linearity to a network. This non-linearity grants networks the ability to learn more complex interactions between inputs X and targets y than only linear relationships. The output will no longer be a linear function of the input. We'll call the output of the last linear layer the "pre-activation" output, which we'll pass to activation functions to obtain transformed output.

**Meet the sigmoid function**

We'll start with the sigmoid activation function, which is widely used for binary classification problems. Let's say we are trying to classify an animal as a mammal or not. We have three pieces of information: number of limbs, whether it lays eggs, and whether it has hair. The latter two are binary variables: 1 if yes, 0 if no. Passing the input to a model with two linear layers returns a single output: the number six. This number is not yet interpretable as "mammal" or not "mammal".

**Meet the sigmoid function**

Enter the sigmoid function! We pass the number six through the sigmoid, and transform it to an output between zero and one. We are now ready to perform a binary classification! If the output is closer to one (greater than 0.5), we label it as class one (mammal). If it were less than 0.5, the prediction would be zero (not a mammal).

Let's implement sigmoid in PyTorch! Here, nn.Sigmoid() takes a one-dimensional input\_tensor of value six.zero and returns an output of the same size, meaning it is also one-dimensional. The output is now bounded between zero and one.

**Activation function as the last layer**

The sigmoid is commonly used as the last step in a neural network when performing binary classification. In the previous step we passed the sigmoid an input\_tensor we defined. More commonly though, we add nn.Sigmoid() as the last step in nn.Sequential(), and the last linear layer's output will automatically be passed to the sigmoid. Note that a sigmoid as the last step in a network of only linear layers is equivalent to a logistic regression using traditional machine learning!

**Getting acquainted with softmax**

We used sigmoid for binary classification. For multiclass classification, involving more than two class labels, we use softmax, another popular activation function. Say we have three classes, bird (0), mammal (1), reptile (2). In this model softmax takes a three-dimensional pre-activation and generates an output of the same shape, one by three. The output is a probability distribution because each element is between zero and one, and values sum to one. In the output shown, the prediction is the second class, mammal, which is the class with the highest probability, 0.842.

In PyTorch, we use nn.Softmax(). dim equals minus one indicates that softmax is applied to input\_tensor's last dimension, which is usually what we do. Similar to sigmoid, softmax can be the last layer in nn.Sequential.

**The sigmoid output cannot return any float value: the output returned by your binary classifier is bounded between zero and one**

**Running a forward pass**

We've learned about tensors, created small networks and learned about activation functions. We'll now dive deeper into generating predictions from models. This is called "running a forward pass" through a network.

**What is a forward pass?**

When we pass input data through a neural network in the forward direction to generate outputs, or predictions, the input data flows through the model layers. At each layer, computations performed on the data generate intermediate representations, which are passed to each subsequent layer until the final output is generated. The purpose of the forward pass is to propagate input data through the network and produce predictions or outputs based on the model's learned parameters (weights and biases). This is used for both training and generating new predictions. The final output can be binary classifications, multi-class classifications, or numerical predictions (regressions).

**Is there also a backward pass?**

All this begs the question: is there also a backward pass? Indeed! We will cover this later in the course, but a backward pass, or backpropagation, is the process by which layer weights and biases are updated during training. All this is part of something called a "training loop". This involves propagating data forward, comparing outputs to true values, then propagating backwards to improve each layer's weights and biases using some handy math. We repeat several times until the model is tuned with meaningful weights and biases. In short, during training, the backward pass is the complementary step to the forward pass.

**Binary classification: forward pass**

Alright, back to the forward pass. Let's combine what we've learned about linear layers and activation functions for binary classification. We use the torch and torch.nn packages as before. Say we have input\_data of five animals, with six features, or neurons, per datapoint. We create a small network with two linear layers and one sigmoid activation function in sequence. The first argument of the first layer matches the number of neurons in our sample data. Correspondingly, the first argument of the second layer matches the output dimension of the first layer. We generate output by passing input\_data to our model.

The output of our binary classification is a single probability between zero and one for each of our five samples. Recall that we commonly use a threshold of point-5 to turn these probabilities into class labels, such one (mammal) or zero (not mammal). This output will not be meaningful until we use backpropagation to update layer weights and biases.

**Multi-class classification: forward pass**

If we wanted to run multi-class classification the model would be mostly similar. Say we are predicting three classes: mammal, bird or reptile. We specify our model has three classes, setting this value as the last linear layer's output dimension. We use softmax instead of sigmoid, with dim equals minus one to indicate the five samples have the same last dimension as the last linear layer's output. Let's use the same input as before. The output shape is five by three.

In each row, we have three probabilities: one for each class. Each row sums to one. The highest probability per row is assigned the class, without thresholds. The first and second rows are mammals, the third is a reptile, and so on.

**Regression: forward pass**

The last model we'll look at is regression: predicting continuous numerical values. Let's say we use the same data on five animals as before, but this time are predicting weights of animals based on their properties. Again, this looks mostly similar to models we've seen. However, there is no activation function at the end; and the last linear layer's last dimension returns an output with one feature. Output dimensions are five by one: five continuous values, one for each row.