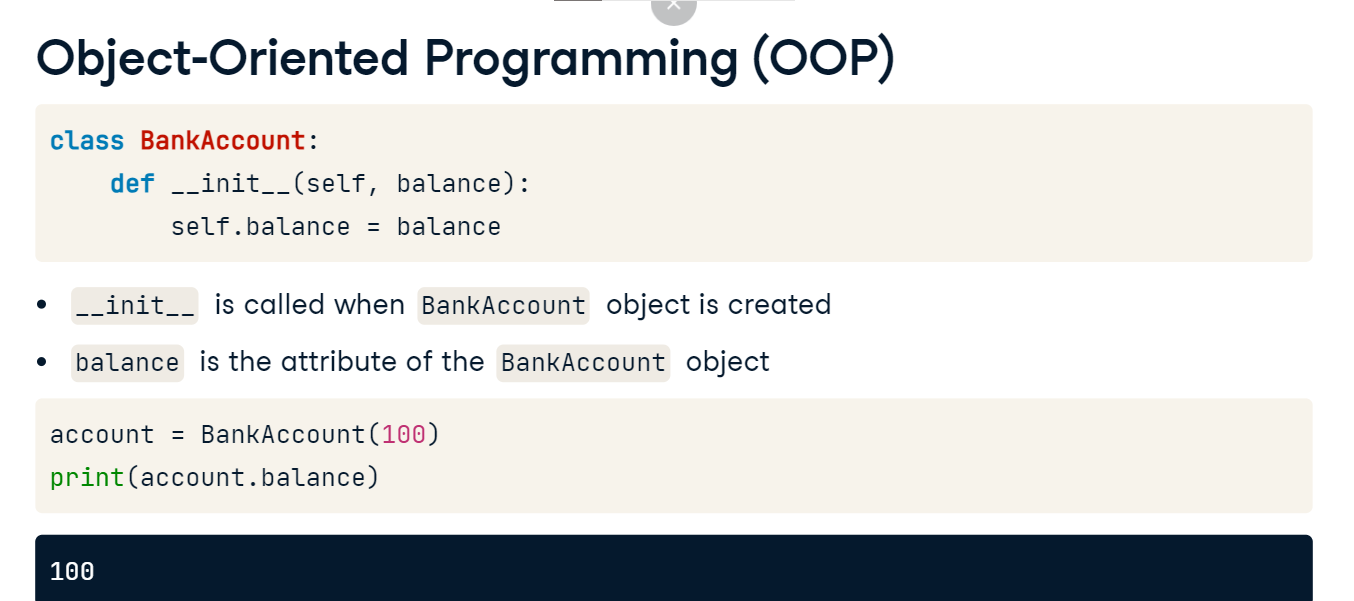
**PyTorch and object-oriented programming**

**Object-Oriented Programming (OOP)**

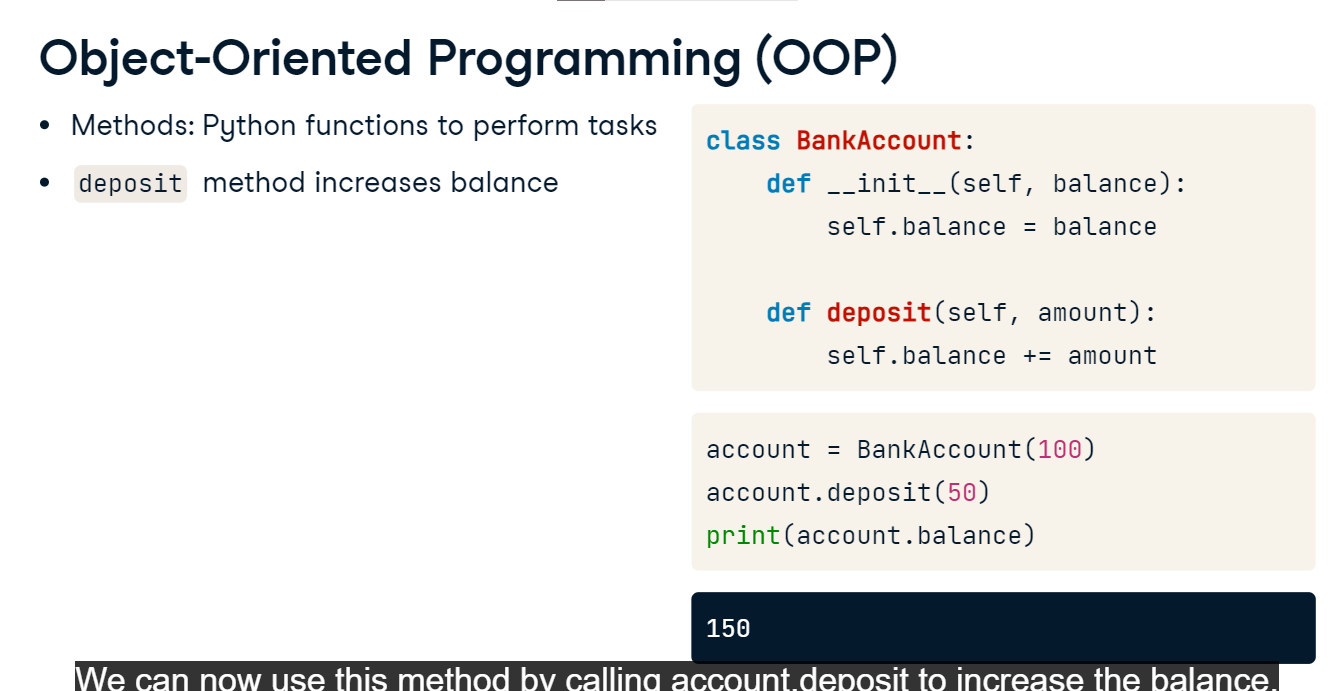
Object-oriented programming, or OOP, is used throughout the PyTorch community to define objects with more flexibility. We will use it to define PyTorch Datasets and Models. OOP is a way of writing computer programs where we create "objects" (like virtual entities), each with abilities (called methods) and data (called attributes).

**Object-Oriented Programming (OOP)**

Take this class called BankAccount. We implement the init method inside the class definition, which is called when the BankAccount object is created. It is written with two underscores on either side, takes two arguments, self and balance, and assigns balance to the object itself using self.balance. This way, the balance becomes the object's attribute. We can now create a new account object with a balance of 100 by calling BankAccount with argument 100 and assigning it to account. We can then access it with account.balance.



Classes can also have methods. These are functions to perform various tasks. We'll add a deposit method that updates the account's balance; it's written like a normal Python function with the addition of self. We can now use this method by calling account.deposit to increase the balance.



**Water potability dataset**

In this chapter, we will be working with the water potability dataset. The task is to classify a water sample as potable or drinkable (1 or 0) based on its chemical characteristics. All features have been normalized to between zero and one.

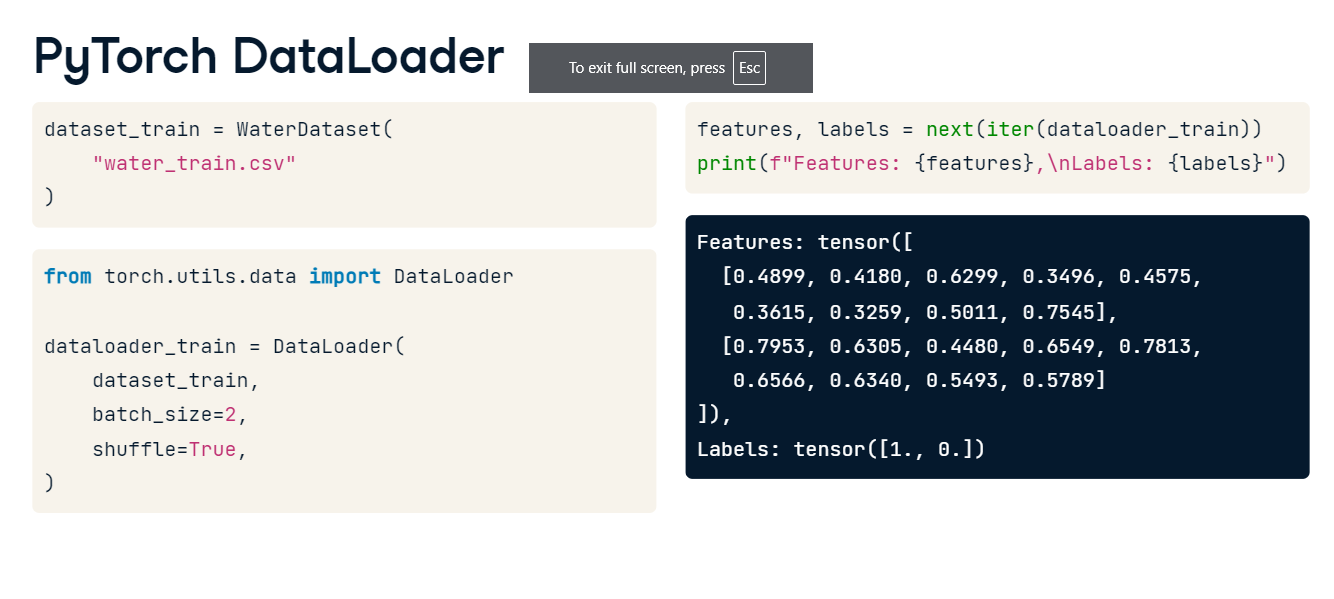
**PyTorch Dataset**

To train a model, we need to build a PyTorch Dataset, set up a DataLoader, and define the model. Let's build a custom Dataset for our water potability data using OOP. We start with the init method, which reads a CSV file into a DataFrame and stores it in the data attribute as a NumPy array. The super-init command ensures our WaterDataset class behaves like its parent class, torch Dataset. Next, PyTorch requires us to implement the len method that returns the total size of the dataset which we access as the 0th element of DataFrame's shape. Finally, we add the getitem method, which takes one argument called idx, the index of a sample, and returns the features (all columns but the last one) and the label (the final column) for that sample.



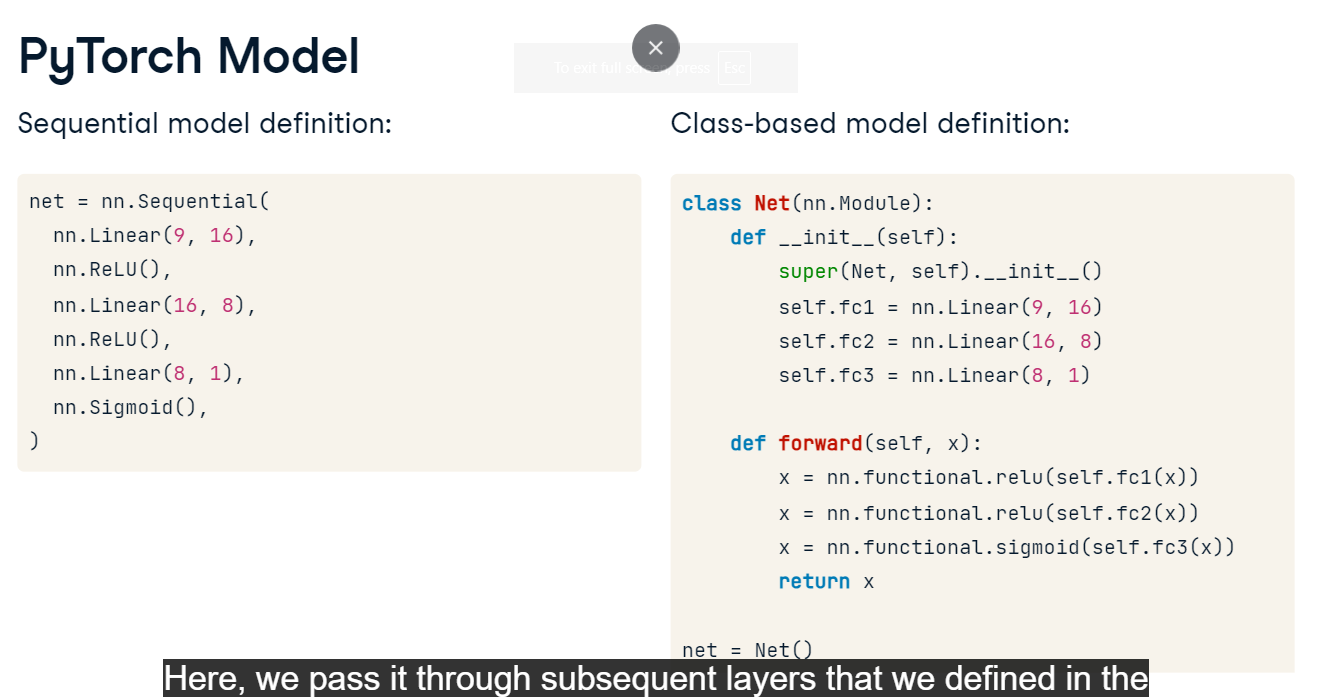
**PyTorch DataLoader**

With the WaterDataset class defined, we create an instance of the Dataset, passing it the training data file path. Then, we pass the Dataset to the PyTorch DataLoader, setting the batch size to two and shuffling the training samples randomly. We use the next-iter-combination to get one batch from the DataLoader. With a batch size of two, we get two samples, each consisting of nine features and a target label.



**PyTorch Model**

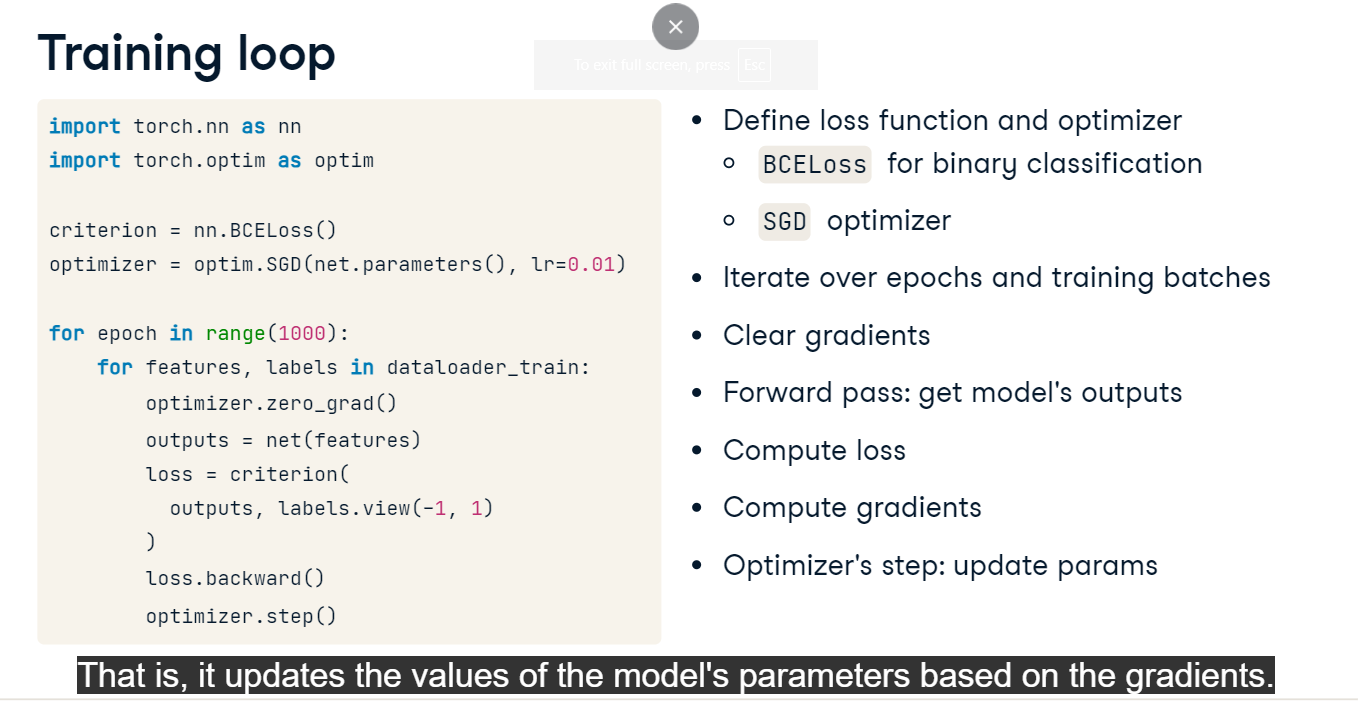
PyTorch models are also best defined as classes. We may have seen sequential models defined like this before. That's fine for small models, but using classes gives us more flexibility to customize as complexity grows. We can rewrite this model using OOP. The Net class is based on the nn.Module, PyTorch's base class for neural networks. We define the model layers we want to use in the init method. The forward method describes what happens to the input when passed to the model. Here, we pass it through subsequent layers that we defined in the init method and wrap each layer's output in the activation function.



**Optimizers, training, and evaluation**

**Training loop**

Let's review the PyTorch training loop. First, we define the loss function, conventionally called criterion, and the optimizer. We'll use Binary Cross-Entropy, or BCE Loss, commonly used for binary classification tasks. We use Stochastic Gradient Descent, or SGD, as the optimizer and tell it which parameters to optimize - here, it's all of net's parameters. Then we pass it the learning rate of point-zero-one. We start the loop by iterating over epochs and batches of training data. Next, we clear the gradients to start from zero for the new batch followed by a forward pass to get the model's outputs. Then, we compare the model's outputs to the ground-truth labels to compute the loss. We reshape the labels with the view method to match the shape of the outputs. We compute the gradients of the model's parameters for the loss using the backward method. These gradients contain information about the direction and size of the changes required to minimize the loss. Finally, we pass the gradients to the optimizer, which performs an optimization step. That is, it updates the values of the model's parameters based on the gradients. Let's take a closer look at the optimization step.

****

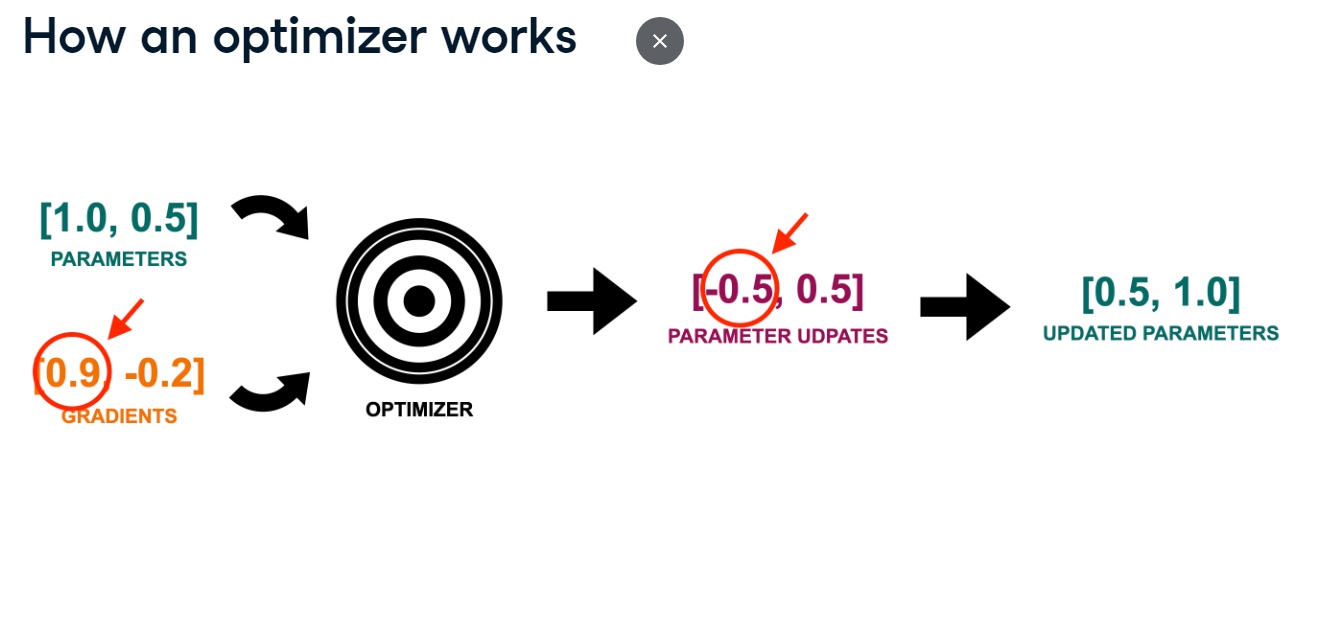
**How an optimizer works**

In practice, neural networks can have billions of parameters. Let's consider an example with only two. Imagine we have the following parameter values and gradients.

They are passed to the optimizer which computes an update for each parameter.

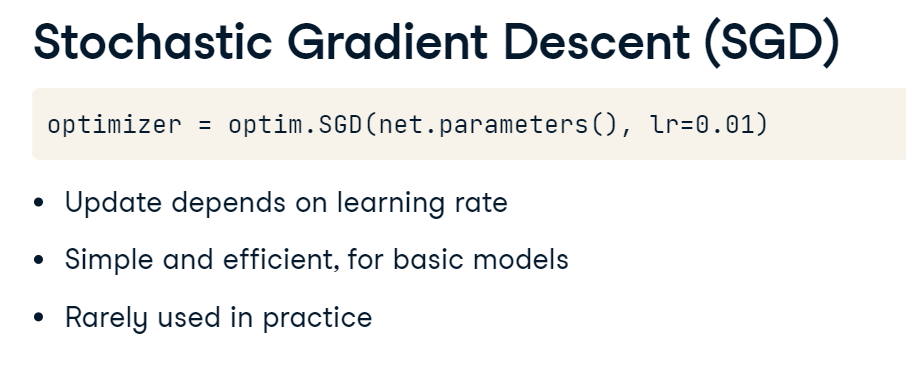
The updates are applied to the parameters and the optimizer step is finished. But how does the optimizer know how much to update and in which direction? The direction depends on the gradient's sign.

The first parameter, for example, has a positive gradient, so it should be decreased in order to decrease the loss. Hence, the parameter update is negative. What about the size of the update? Different optimizers use different approaches to decide how much to update.



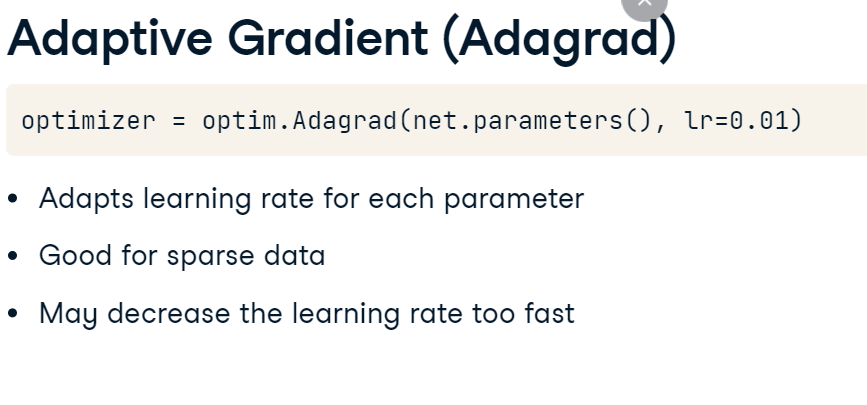
**Stochastic Gradient Descent (SGD)**

In Stochastic Gradient Descent, or SGD, the size of the parameter update depends only on the learning rate, a predefined hyperparameter. SGD is computationally efficient, but because of its simplicity, it's rarely used in practice.



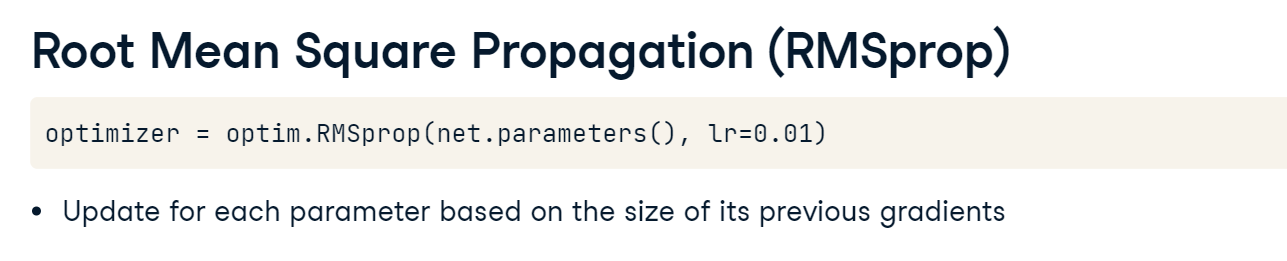
**Adaptive Gradient (Adagrad)**

Using the same learning rate for each parameter cannot be optimal. Adaptive Gradient, or Adagrad, improves on it by decreasing the learning rate during training for parameters that are infrequently updated. This makes it well-suited for sparse data, that is, data in which some features are not often observed. However, Adagrad tends to decrease the learning rate too fast.



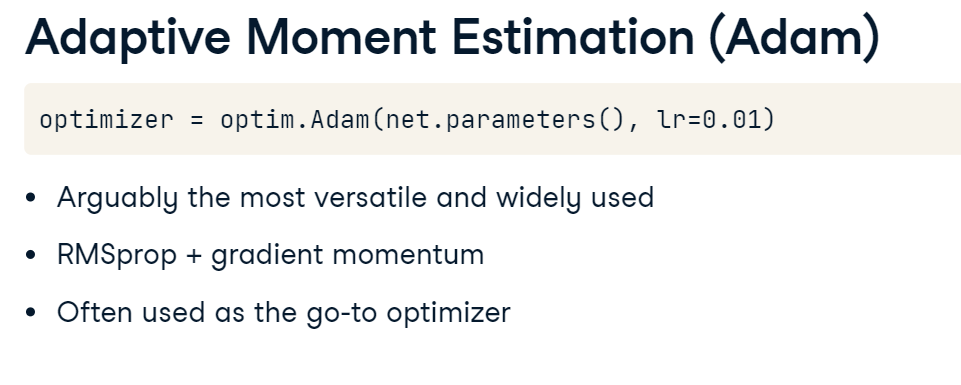
**Root Mean Square Propagation (RMSprop)**

Root Mean Square Propagation, or RMSprop, addresses Adagrad's aggressive learning rate decay by adapting the learning rate for each parameter based on the size of its previous gradients.



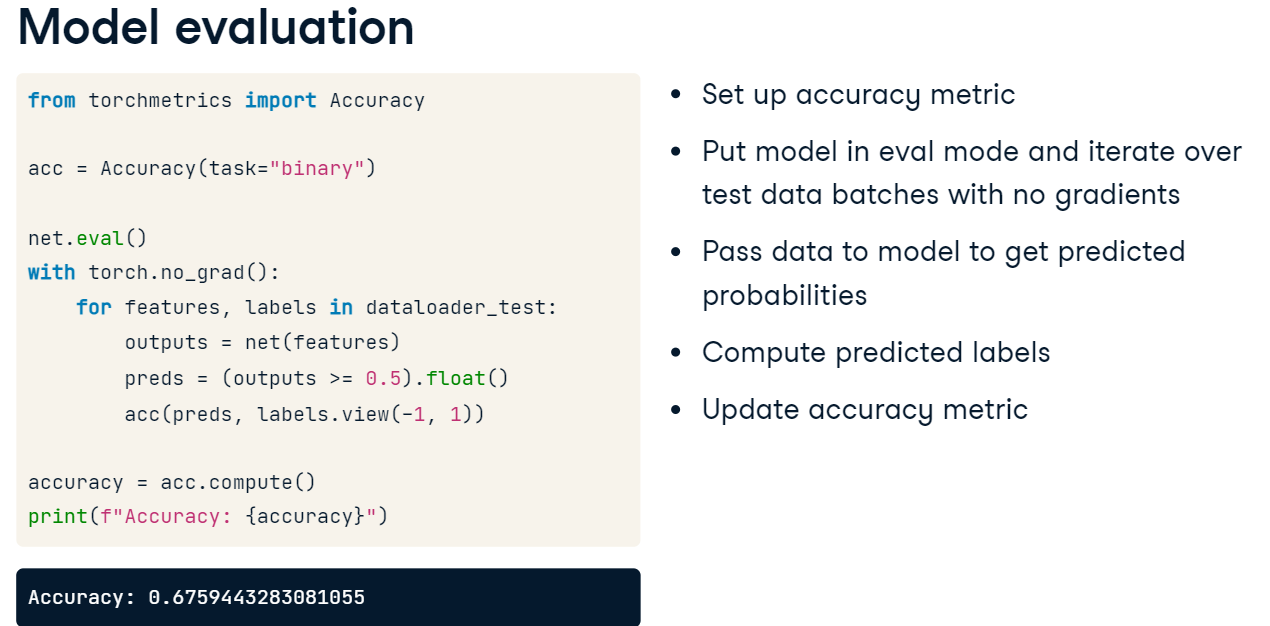
**Adaptive Moment Estimation (Adam)**

Finally, Adaptive Moment Estimation or Adam is arguably the most versatile and widely used optimizer. It combines RMSprop with the concept of momentum: the average of past gradients where the most recent gradients have more weight. Basing the update on both gradient size and momentum helps accelerate training. Adam is often the default go-to optimizer, and we will use it throughout the course.



**Model evaluation**

Once the model is trained, we can evaluate its performance on test data. First, we set up the binary accuracy metric from torchmetrics. Then, we put the model in the evaluation mode with net.eval and iterate over the dataloader\_test with no gradients calculation and do the forward pass to get predicted probabilities, which we then transform into predicted labels based on the 0.5 threshold. Finally, we update the accuracy score. Let's compute and print the overall accuracy. We got over 67%, not bad for a basic model and small dataset!



**Vanishing and exploding gradients**

**Vanishing gradients**

Neural networks often suffer from gradient instability during training. Sometimes, the gradients get smaller during the backward pass. This is known as vanishing gradients. As a result, earlier layers receive hardly any parameter updates and the model doesn't learn.

**Exploding gradients**

In other cases, the gradients get increasingly large, leading to huge parameter updates and divergent training. This is known as exploding gradients.

**Solution to unstable gradients**

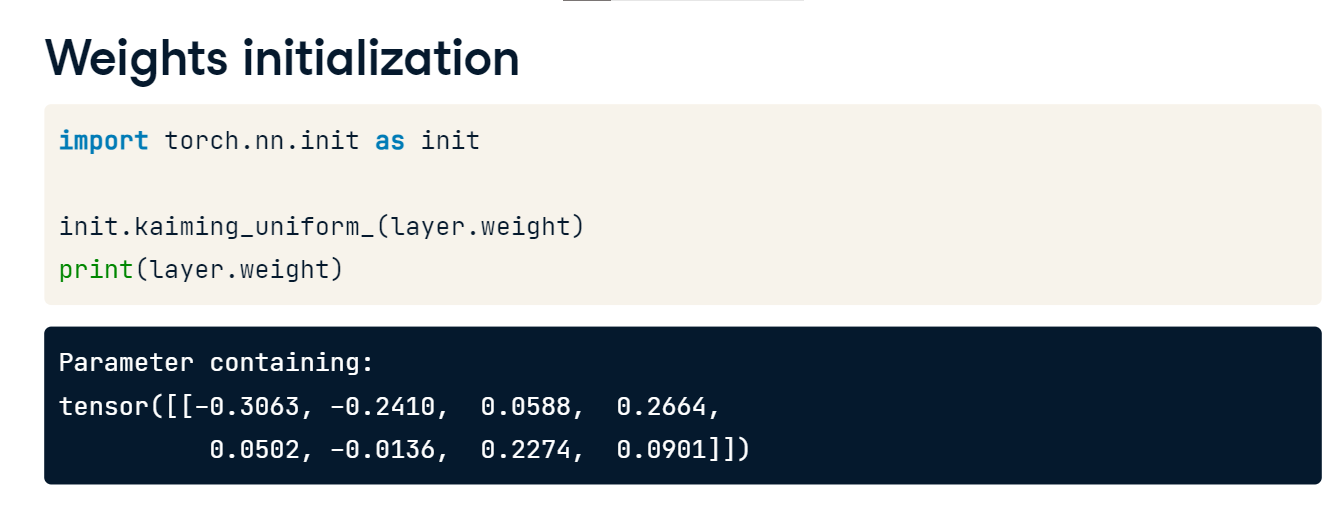
To address these problems, we need a three-step solution consisting of proper weights initialization, good activations, and batch normalization. Let's review these steps.

**Weights initialization**

Whenever we create a torch layer, its parameters stored in the weight attribute get initialized to random values.

To prevent unstable gradients, research showed that initialization should ensure that the variance of the layer's inputs is close to that of its outputs and the variance of the gradients is the same before and after passing through the layer. The way to achieve this is different for each activation function. For ReLU, or Rectified Linear Unit, and similar activations, we can use He initialization, also known as Kaiming initialization.

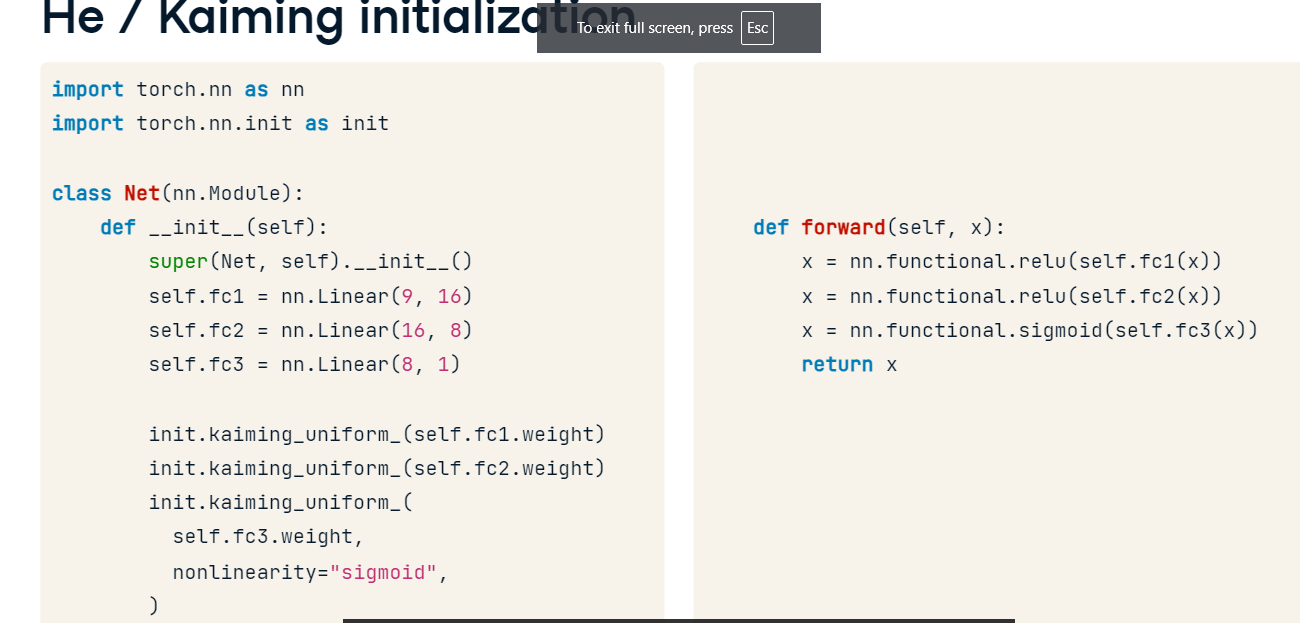
To apply this initialization, we call kaiming-underscore-uniform-underscore from torch.nn.init on the layer's weight attribute. This ensures the desired variance properties.

****

**He / Kaiming initialization**

To implement it, we need one small change in our model's init method: for each layer, we call kaiming\_uniform\_ on its weight attribute. For the last layer, where we use sigmoid activation in the forward method, we also specify nonlinearity as sigmoid.

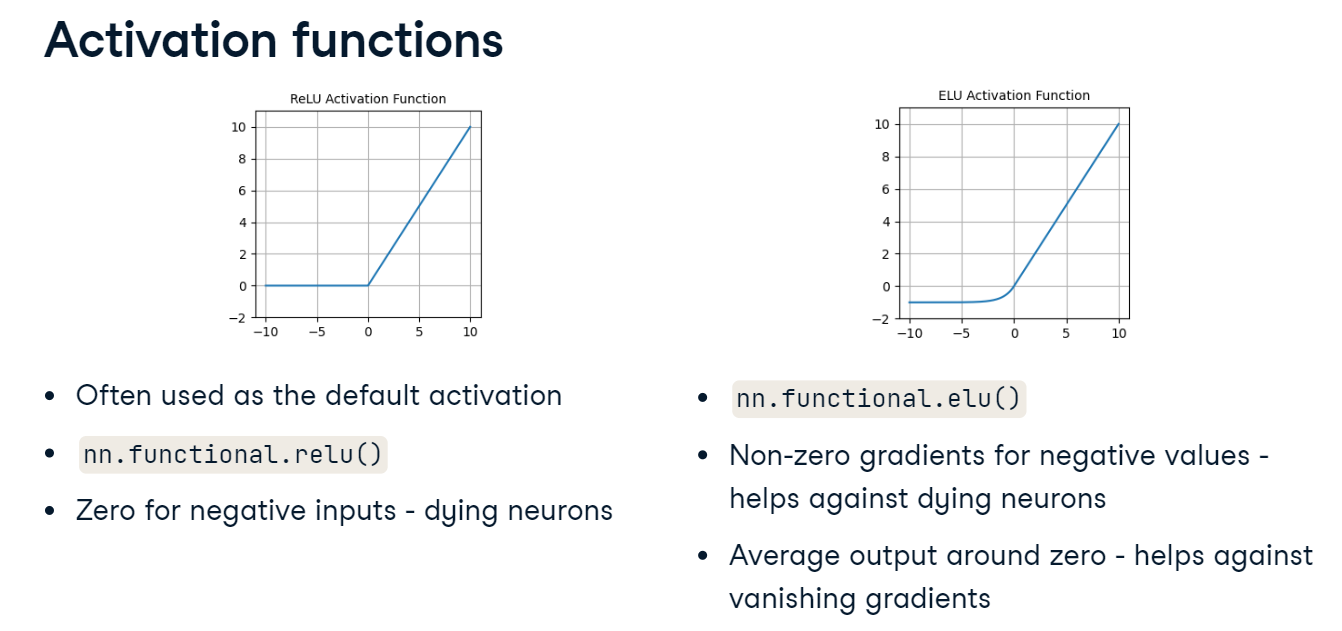
This is what it looks like within the full model.

****

**10. Activation functions**

01:46 - 02:38

Let's discuss activation functions now. The ReLU, or Rectified Linear Unit, is arguably the most commonly used activation. It's available as nn.functional.relu. It has several advantages, but also an important drawback. It suffers from the dying neuron problem: during training, some neurons only output a zero. This is caused by the fact that ReLU is zero for any negative value. If inputs to a neuron become negative, it effectively dies. The ELU or Exponential Linear Unit is one activation designed to improve upon ReLU. It's available as nn.functional.elu. Thanks to non-zero gradients for negative values, it doesn't suffer from the dying neurons problem. Additionally, its average output is near zero, so it's less prone to vanishing gradients.



**Batch normalization**

A good choice of initial weights and activation functions can alleviate unstable gradients at the beginning of training, but it doesn't prevent them from returning during training. A solution to this is batch normalization. Batch normalization is an operation applied after a layer, in which the layer's outputs are first normalized by subtracting the mean and dividing by the standard deviation. This ensures the output distribution is roughly normal. Then, the normalized outputs are scaled and shifted using shift and scale parameters that the batch normalization learns just like linear layers learn their weights. Effectively, batch norm allows the model to learn the optimal distribution of inputs to each layer before it is applied. This speeds up the loss decrease and makes it more immune to unstable gradient issues.

To add batch normalization to a PyTorch model, we must define the batch norm layer using nn.BatchNorm1d in the init method. Here, we call it "bn1". We pass it the input size, which needs to be equal to the preceding layer's output size, in this case 16. Then, in the forward method, we pass the linear layer's output to the batch norm layer and pass the result to the activation function.

