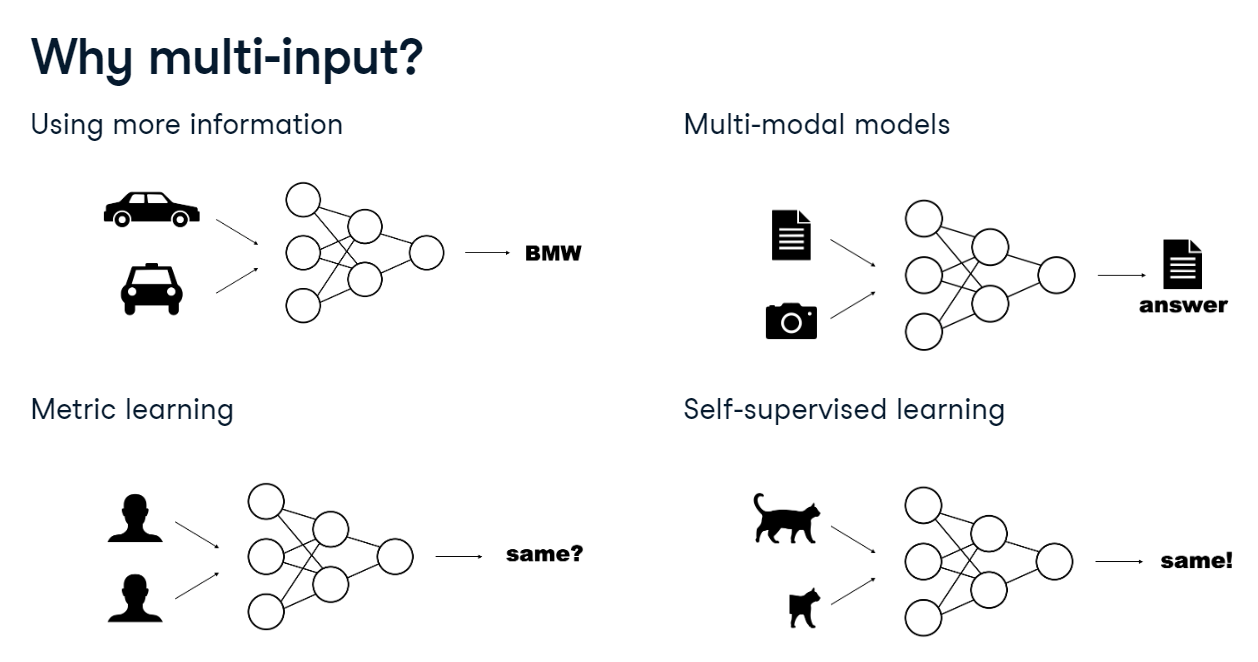
**Multi-input models**

**Why multi-input?**

Multi-input models, or models that accept more than one source of data, have many applications. First, we might want the model to use multiple information sources, such as two images of the same car to predict its model. Second, multi-modal models can work on different input types such as image and text to answer a question about the image. Next, in metric learning, the model learns whether two inputs represent the same object. Think about an automated passport control where the system compares our passport photo with a picture it takes of us. Finally, in self-supervised learning, the model learns data representation by learning that two augmented versions of the same input represent the same object. Multi-input models are everywhere!



**Omniglot dataset**

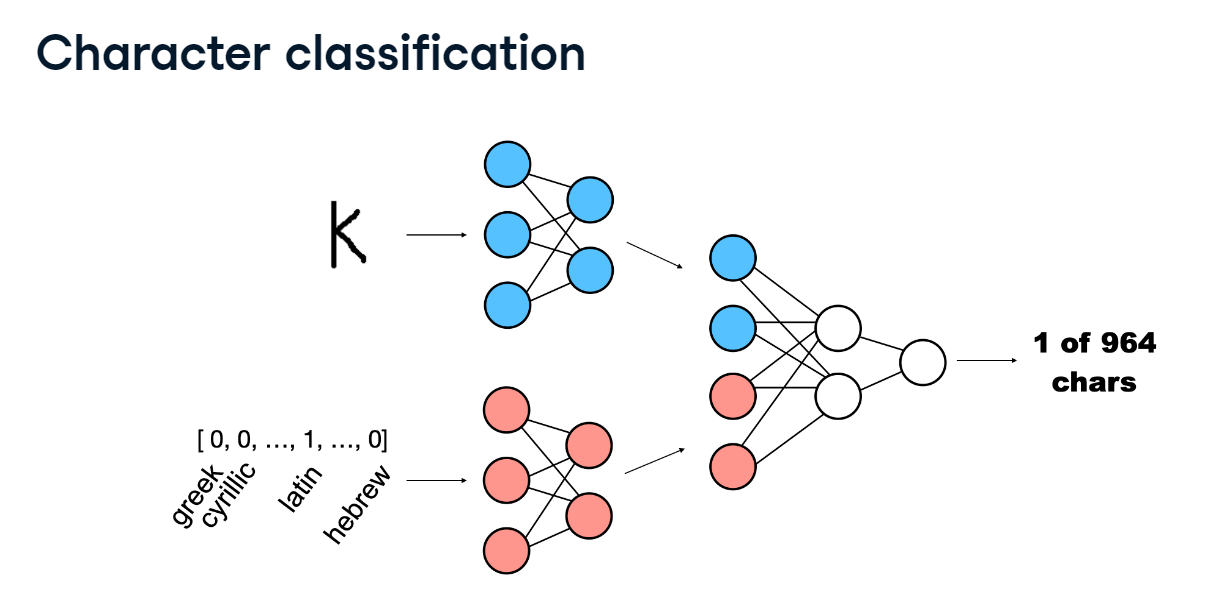
Throughout the chapter, we will be using the Omniglot dataset, a collection of images of 964 different handwritten characters from 30 different alphabets.

1. 1 Lake, B. M., Salakhutdinov, R., and Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. Science, 350(6266), 1332-1338.

**Character classification**

Let's use the Omniglot dataset to build a two-input model to classify handwritten characters. The first input will be the image of the character, such as this Latin letter "k". The second input will the the alphabet that it comes from expressed as a one-hot

Both inputs will be processed separately, then we concatenate their representations. Finally, a classification layer predicts one of the 964 classes. We need two elements to build such a model: a custom Dataset and an appropriate model architecture.



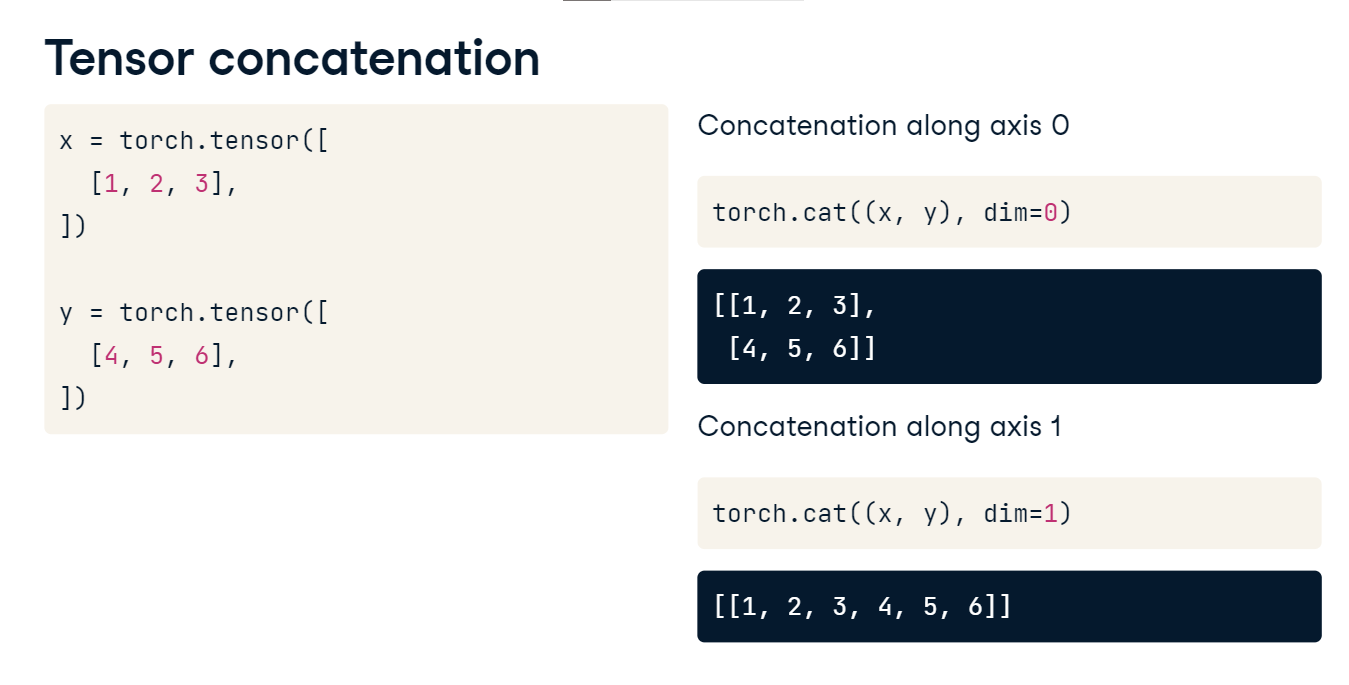
**Two-input Dataset**

Let's start with the custom Omniglot dataset. We set it up as a class based on torch Dataset. In the init method, we store transform and samples provided when instantiating the dataset as class attributes. Samples are tuples of three: image file path, alphabet as a one-hot vector, and target label as the character class index. In the exercises, samples will be provided. For personal projects, we would need to create them from data file paths. Next, we need to implement the len method that returns the number of samples. Finally, the getitem method returns one sample based on the index it receives as input. For the given index, we retrieve the sample and load the image using Image.open from PIL. The convert method with the argument "L" makes sure that the image is read as grayscale. Then, we transform the image and return a triplet: the transformed image, the alphabet vector, and the target label.



**Tensor concatenation**

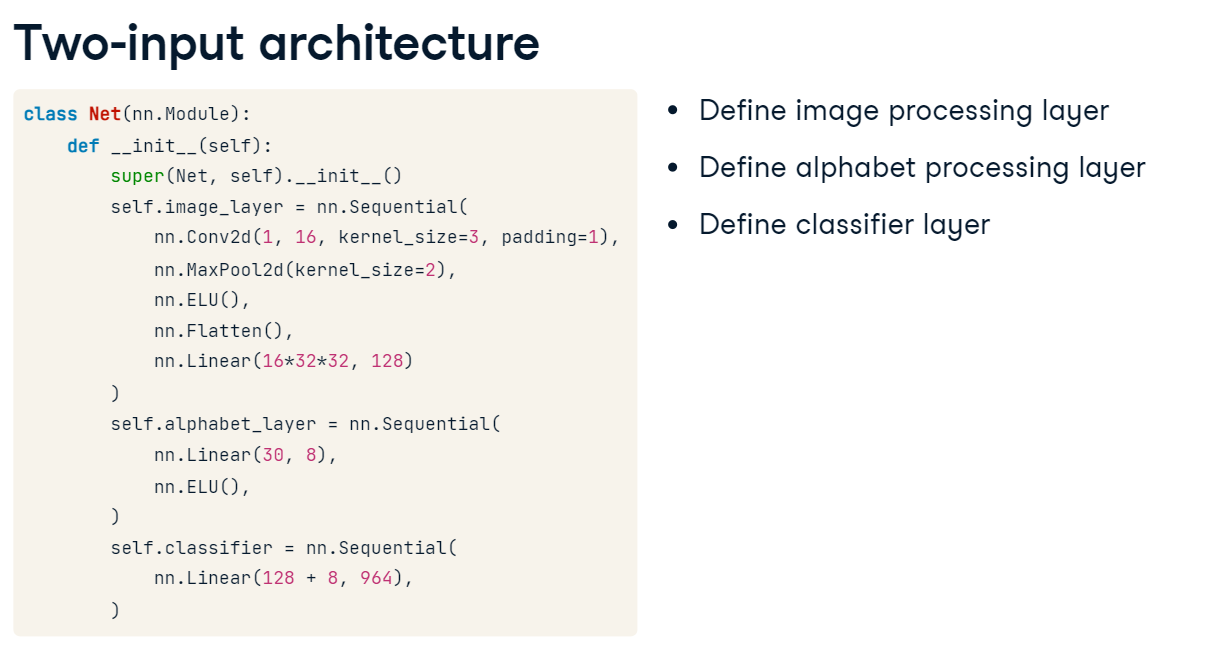
Before we proceed to building the model, we need to understand tensor concatenation. torch.cat concatenates tensors along a specified dimension. We pass it the tensors and the dimension: for 2D tensors, 0 stands for "horizontal" and 1 stands for "vertical" concatenation.

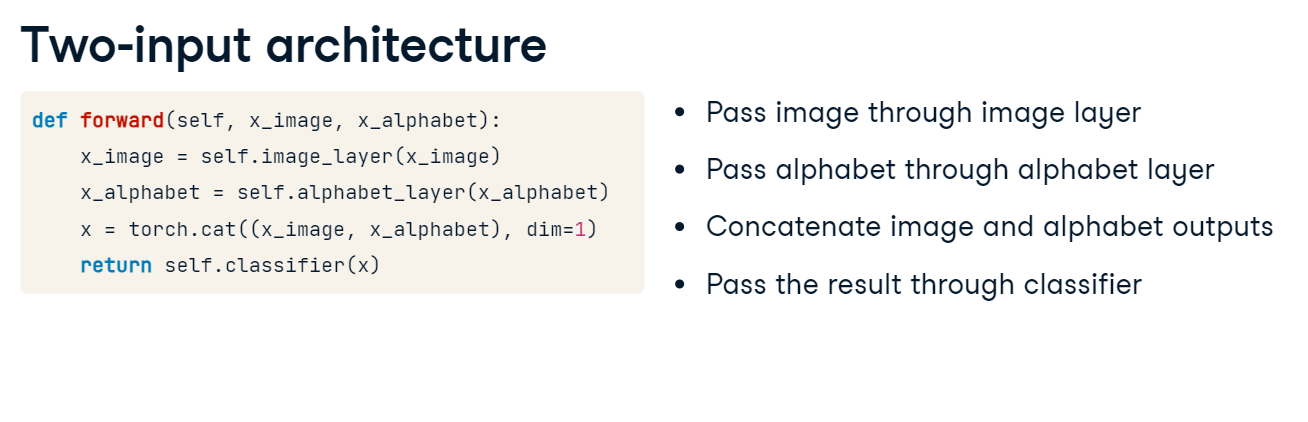


**Two-input architecture**

It's time to define our two-input model. We start with defining a sub-network or layer to process our first input, the image. It should look familiar: a convolution, max pool, elu activation, flattened to a linear layer of shape 128 in the end. Next, we define a layer to process our second input, the alphabet vector. Its input size is 30, the number of alphabets, and we map it to an arbitrarily chosen output size of 8. Then, a classifier would accept input of size 128 plus 8 (image and alphabet outputs concatenated) and produce the output of size 964, the number of classes.

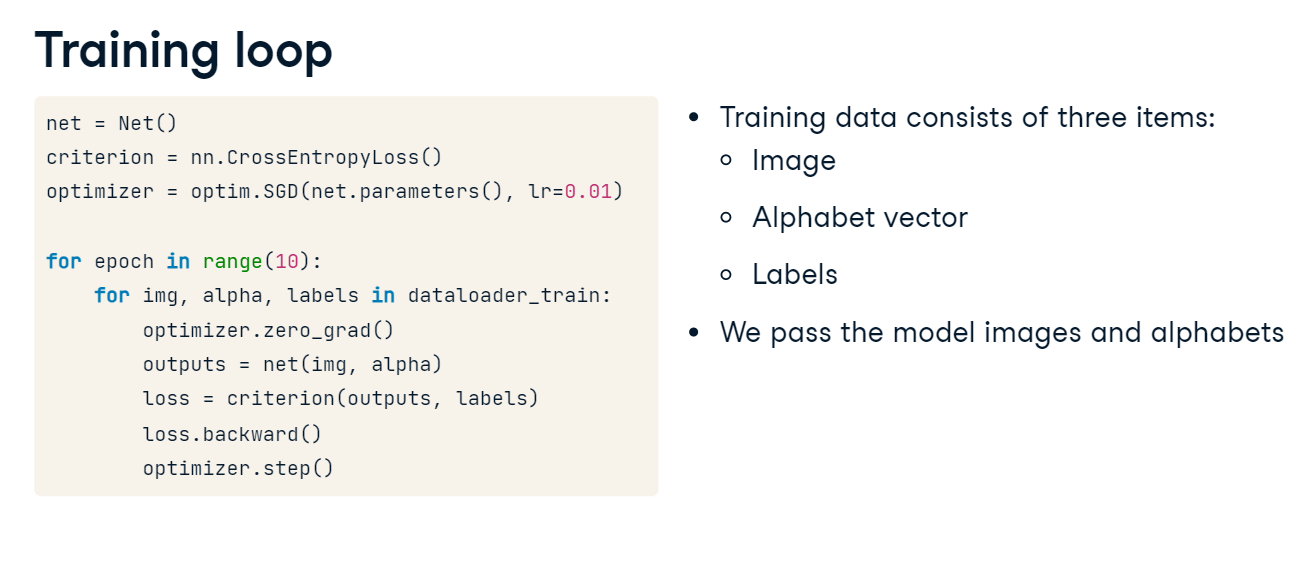
In the forward method, we pass each input through its corresponding layer. Then, we concatenate the outputs with torch.cat. Finally, we pass the result through the classifier layer and return.



****

**Training loop**

The training loop looks just like all the ones we have seen so far. The only difference is that now the training data consists of three items: the image, the alphabet vector, and the labels, and we pass the images and alphabets to the model.



**1. Multi-output models**

00:00 - 00:06

Welcome back! In this video, we'll look at multi-output models.

**2. Why multi-output?**

00:06 - 00:55

Just like multi-input models, multi-output architectures are everywhere. Their simplest use-case is for multi-task learning, where we want to predict two things from the same input, such as a car's make and model from its picture. In multi-label classification problem, the input can belong to multiple classes simultaneously. For instance, an image can depict both a beach and people. For each of these labels, a separate output from the model is needed. Finally, in very deep models built of blocks of layers, it is a common practice to add extra outputs predicting the same targets after each block. These additional outputs ensure that the early parts of the model are learning features useful for the task at hand while also serving as a form of regularization to boost the robustness of the network.

**3. Character and alphabet classification**

00:55 - 01:07

Let's use the Omniglot dataset again to build a model to predict both the character and the alphabet it comes from based on the image. First, we will pass the image through some layers to obtain its embedding.

**4. Character and alphabet classification**

01:07 - 01:12

Then we add two independent classifiers on top, one for each output.

**5. Two-output Dataset**

01:12 - 01:42

The good news is that we have already done much of the work needed. We can reuse the OmniglotDataset we built before, with just one small difference in the samples we pass it. When the alphabet was an input to the model, we represented it as a one-hot vector. Now that it is an output, all we need is the integer representing the class label, just like with the other output, the character. This will be a number between 0 and 29 since we have 30 alphabets in the Dataset.

**6. Two-output architecture**

01:42 - 02:12

Let's look at the model's architecture. We start with defining a sub-network for processing the image identical to the one we used before. Then, we define two classifier layers, one for each output, with the output shape corresponding to the number of alphabets (30) and characters (964), respectively. In the forward method, we first pass the image through its dedicated sub-network, and then feed the result separately to each of the two classifiers. Finally, we return the two outputs.

**7. Training loop**

02:12 - 02:56

Let's examine the training loop. The beginning should look familiar, except for the fact that now the model produces two outputs instead of one. Having produced these outputs, we calculate the loss for each of them separately using the appropriate target labels. Next, we need to define the total loss for the model to optimize. Here, we just sum the two partial losses together, indicating that the accuracy of predicting the alphabet and the character is equally important. If that is not the case, we can weigh the partial losses with some weights to reflect their relative importance. We will explore this idea later in the next video. Finally, we run backpropagation and the optimization step as always.

**Multi-output models**

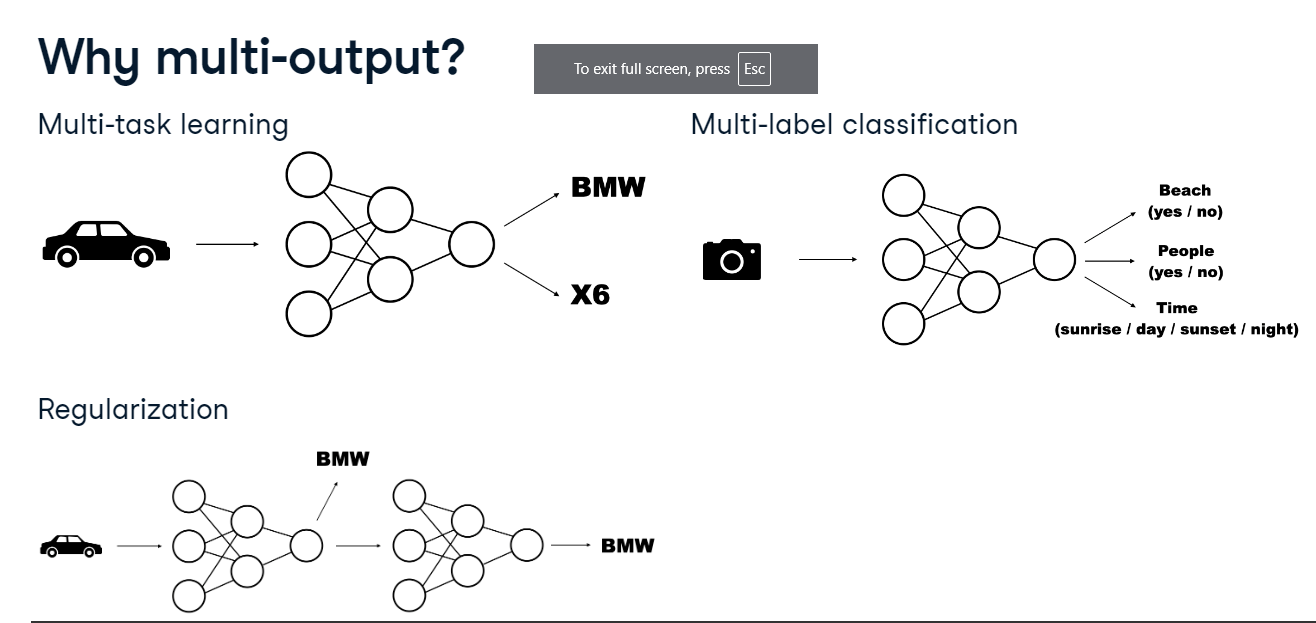
**Why multi-output?**

Just like multi-input models, multi-output architectures are everywhere. Their simplest use-case is for multi-task learning, where we want to predict two things from the same input, such as a car's make and model from its picture. In multi-label classification problem, the input can belong to multiple classes simultaneously. For instance, an image can depict both a beach and people. For each of these labels, a separate output from the model is needed. Finally, in very deep models built of blocks of layers, it is a common practice to add extra outputs predicting the same targets after each block. These additional outputs ensure that the early parts of the model are learning features useful for the task at hand while also serving as a form of regularization to boost the robustness of the network.

**Character and alphabet classification**

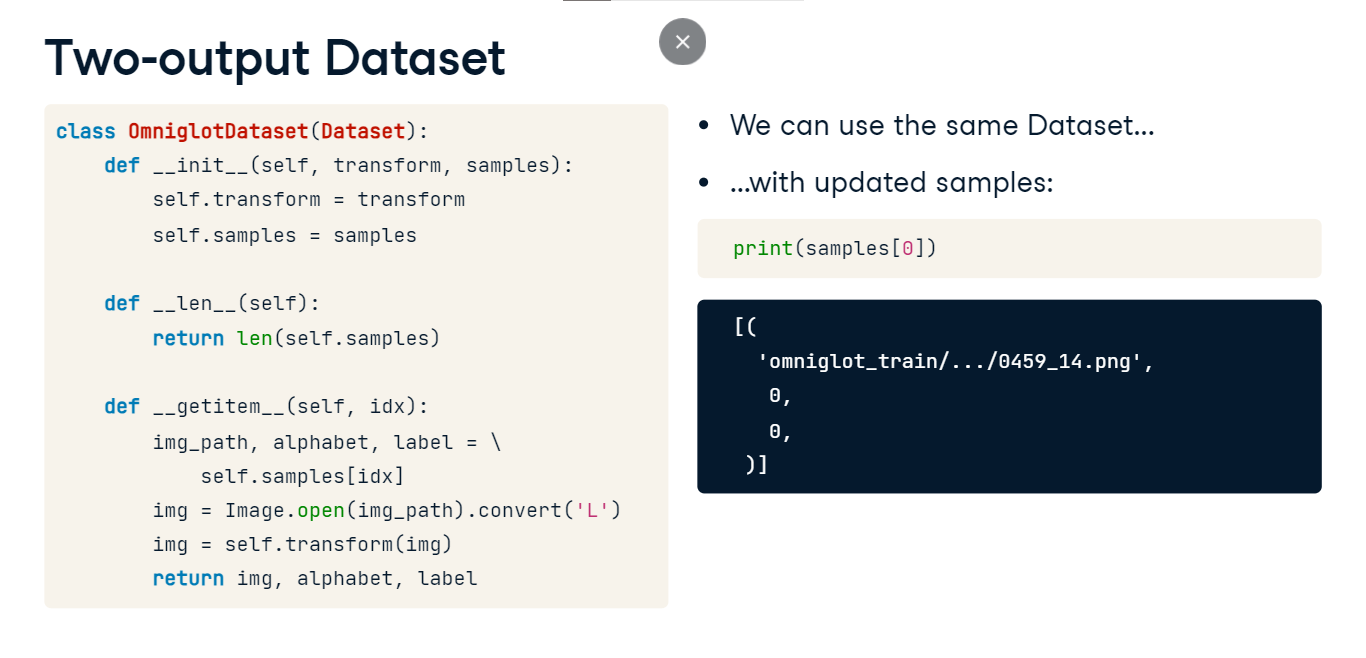
Let's use the Omniglot dataset again to build a model to predict both the character and the alphabet it comes from based on the image. First, we will pass the image through some layers to obtain its embedding.

Then we add two independent classifiers on top, one for each output.



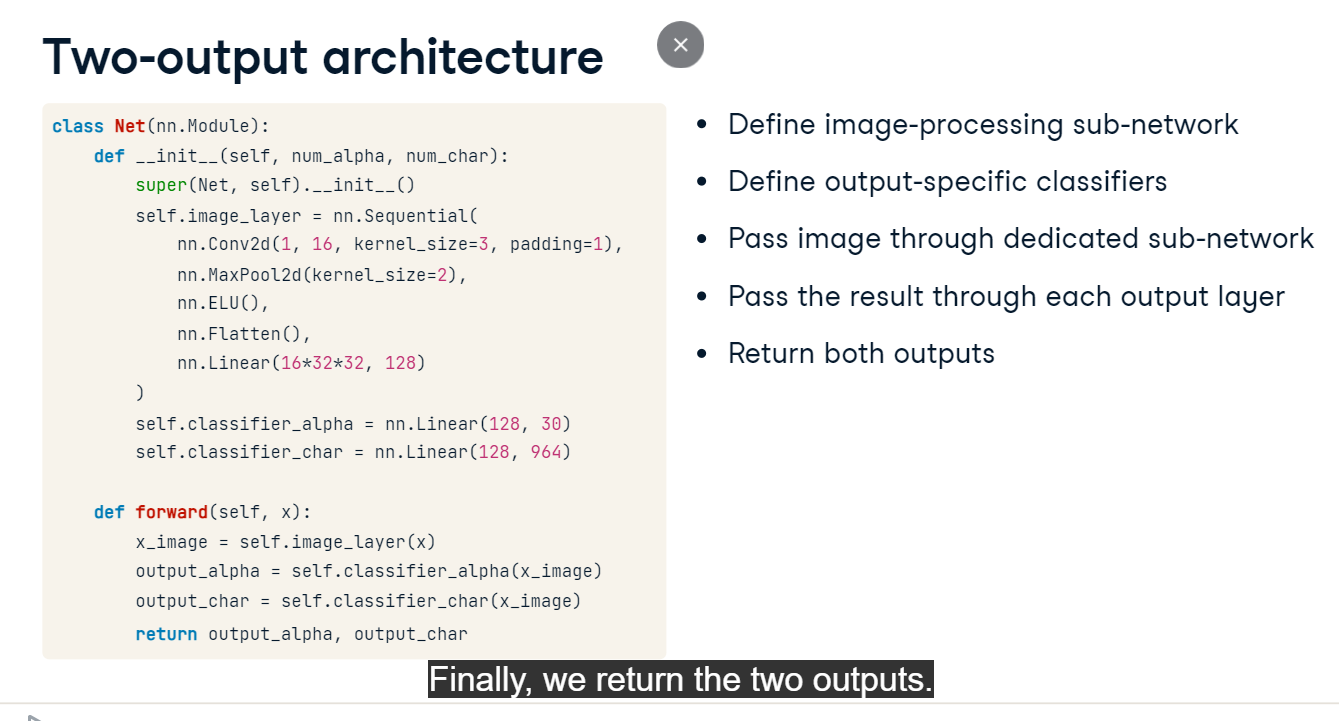
**Two-output Dataset**

The good news is that we have already done much of the work needed. We can reuse the OmniglotDataset we built before, with just one small difference in the samples we pass it. When the alphabet was an input to the model, we represented it as a one-hot vector. Now that it is an output, all we need is the integer representing the class label, just like with the other output, the character. This will be a number between 0 and 29 since we have 30 alphabets in the Dataset.



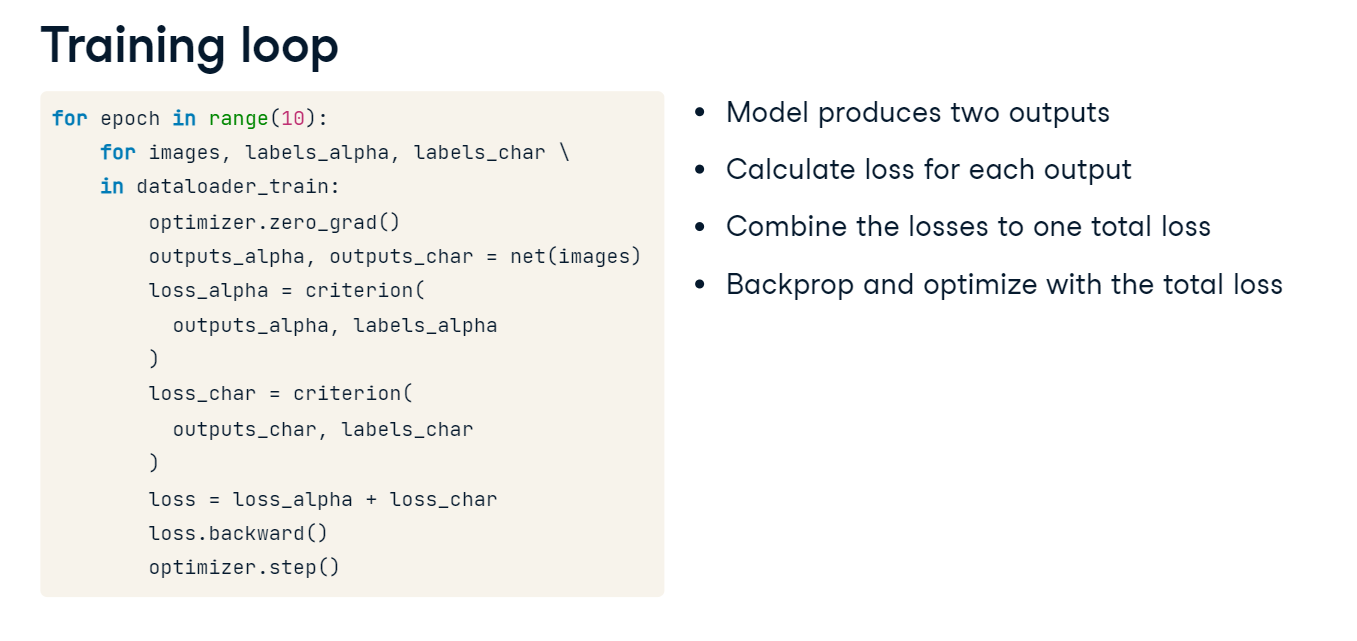
**Two-output architecture**

Let's look at the model's architecture. We start with defining a sub-network for processing the image identical to the one we used before. Then, we define two classifier layers, one for each output, with the output shape corresponding to the number of alphabets (30) and characters (964), respectively. In the forward method, we first pass the image through its dedicated sub-network, and then feed the result separately to each of the two classifiers. Finally, we return the two outputs.



**Training loop**

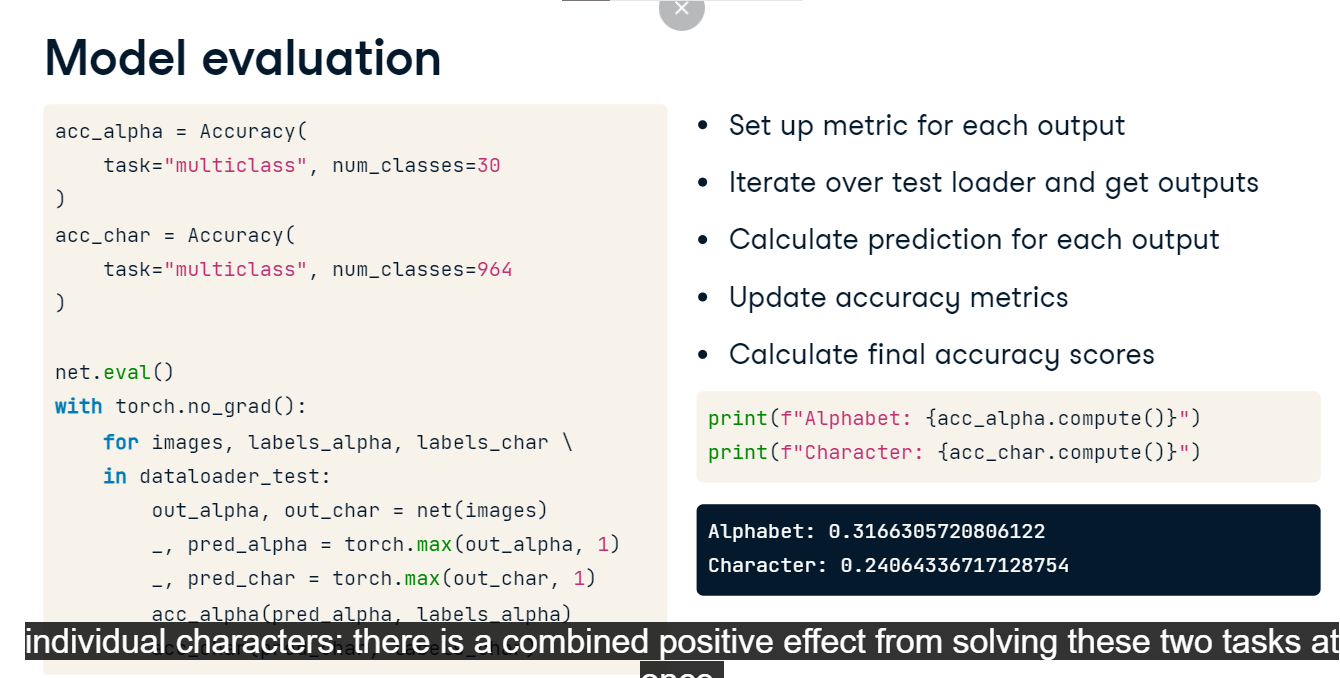
Let's examine the training loop. The beginning should look familiar, except for the fact that now the model produces two outputs instead of one. Having produced these outputs, we calculate the loss for each of them separately using the appropriate target labels. Next, we need to define the total loss for the model to optimize. Here, we just sum the two partial losses together, indicating that the accuracy of predicting the alphabet and the character is equally important. If that is not the case, we can weigh the partial losses with some weights to reflect their relative importance. We will explore this idea later in the next video. Finally, we run backpropagation and the optimization step as always.



## Evaluation of multi-output models and loss weighting

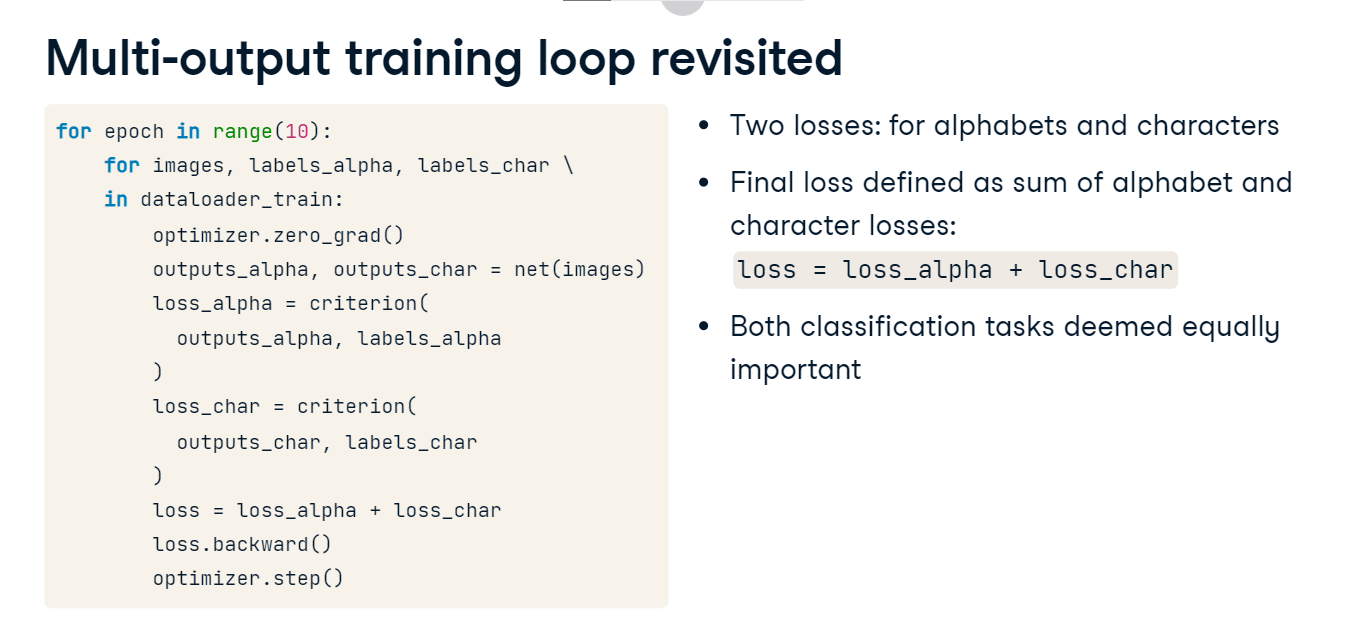
**Model evaluation**

Let's start with the evaluation of a multi-output model. It's very similar to what we have done before. However, with two different outputs, we need to set up two accuracy metrics: one for alphabet classification and one for character classification. We iterate over the test DataLoader and get the model's predictions as usual. Finally, we update the accuracy metrics, and after the loop, we can calculate their final values. The accuracy is higher for alphabets than for characters, which is not surprising: predicting the alphabet is an easier task with just 30 classes to choose from; for characters, there are 964 possible labels. The difference in accuracy scores is not very large, however: 31 versus 24 percent. This is because learning to recognize the alphabets helped the model recognize individual characters: there is a combined positive effect from solving these two tasks at once.



**Multi-output training loop revisited**

Let's now take a look at the training loop for our last model predicting characters and alphabets. Because the model solves two classification tasks at the same time, we have two losses: one for alphabets, and another one for characters. However, since the optimizer can only handle one objective, we had to combine the two losses somehow. We chose to define the final loss as the sum of the two partial losses. By doing so, we are telling the model that recognizing characters and recognizing alphabets are equally important to us. If that is not the case, we can combine the two losses differently.



**Varying task importance**

Let's say that correct classification of characters is twice as important for us as the classification of alphabets. To pass this information to the model, we can multiply the character loss by two to force the model to optimize it more. Another approach is to assign weights to both losses that sum up to one. This is equivalent from the optimization perspective, but arguably easier to read for humans, especially with more than two loss components.

****

**Warning: losses on different scales**

There is just one caveat: when assigning loss weights, we must be aware of the magnitudes of the loss values. If the losses are not on the same scale, one loss could dominate the other, causing the model to effectively ignore the smaller loss. Consider a scenario where we're building a model to predict house prices, and use MSE loss. If we also want to use the same model to provide a quality assessment of the house, categorized as "Low", "Medium", or "High", we would use cross-entropy loss. Cross-entropy is typically in the single-digit range, while MSE can reach tens of thousands. Combining these two would result in the model ignoring the quality assessment task completely. A solution is to scale each loss by dividing it by the maximum value in the batch. This brings them to the same range, allowing us to weight them if desired and add together.

