**Handling images with PyTorch**

**Clouds dataset**

We will work with the clouds dataset from Kaggle containing photos of seven different cloud types. We'll build an image classifier to predict the cloud from an image. But first - what is an image?

1. 1 https://www.kaggle.com/competitions/cloud-type-classification2/data

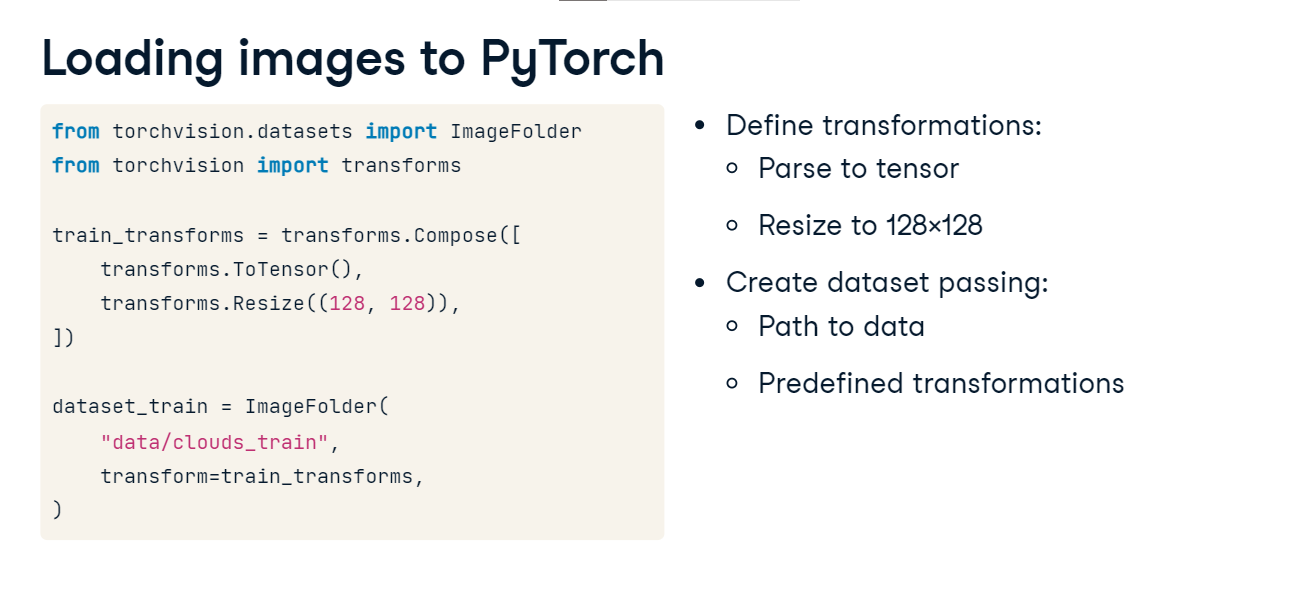
**What is an image?**

Digital images are comprised of pixels, short for "picture elements". A pixel is the smallest unit of the image. It's a tiny square that represents a single point. If we zoom into this cloud picture, we can see the pixels. Each pixel contains numerical information about its color. In a grayscale image, each pixel represents a different shade of gray, ranging from black to white which would be an integer between 0 and 255, respectively. A value of 30, for example, represents the following shade of gray. In color images, each pixel is typically described by three integers, denoting the intensities of the three color channels: red, green, and blue. For example, a pixel with red of 52, green of 171, and blue of 235 represents the following shade of blue.

**Loading images to PyTorch**

Let's build a PyTorch Dataset of cloud images. This is easiest with a specific directory structure. We have two main folders called cloud\_train and cloud\_test. Within each, there are seven directories, each representing a cloud type, or one category in our classification task. We have jpg image files inside each category folder.

With this directory structure, we can use ImageFolder from torchvision to create a Dataset. First, we need to define the transformations to apply to an image as it is loaded. To do this, we call transforms.Compose and pass it a list of two transformations: we parse the image to a torch tensor with ToTensor and resize it to 128 by 128 pixels to ensure all images are the same size. Then, we create a Dataset using ImageFolder, passing it the training data path and the transforms we defined.



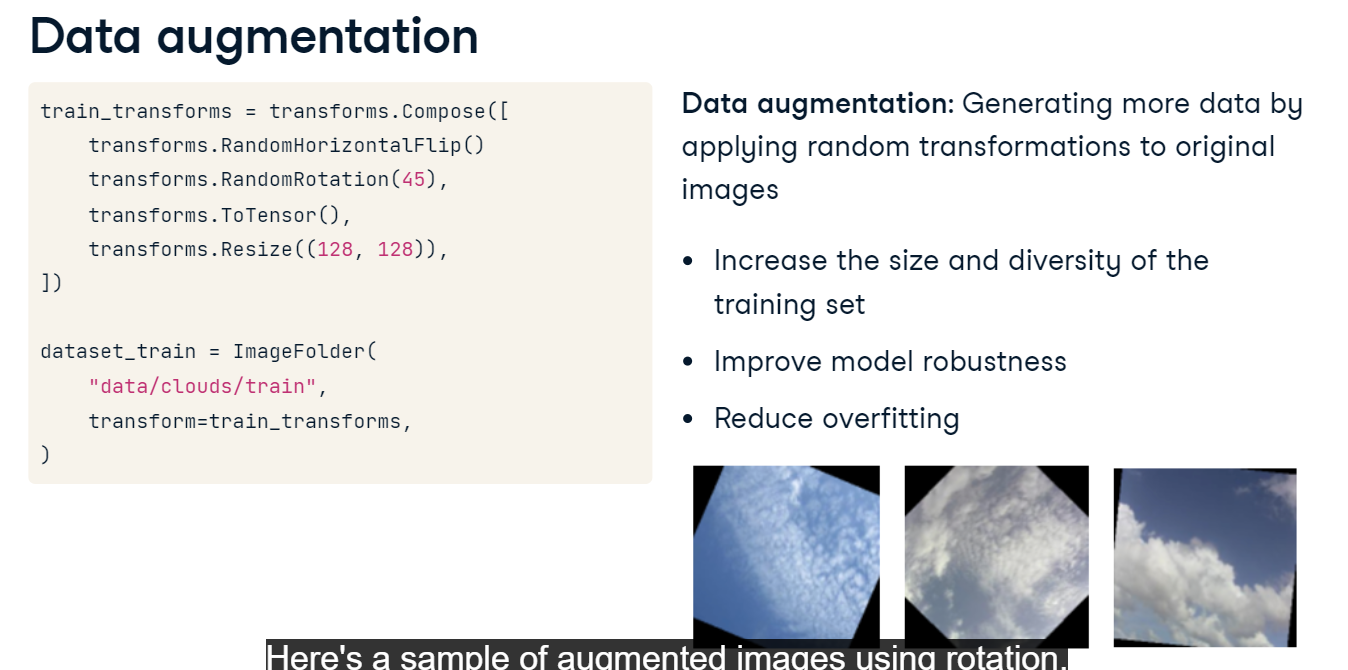
**Displaying images**

dataset\_train is a PyTorch dataset just like the WaterDataset we saw before. We can create the DataLoader from it and get a data sample. Notice the shape of the loaded image: 1 by 3 by 128 by 128. 1 corresponds to the batch size of 1, 3 - to the three color channels, and 128 by 128 represents the image's height and width. To display a color image like this, we must rearrange its dimensions so the height and width come before the channels. We call squeeze on the image to eliminate the 1-dimensions of the batch size, and then permute the order by replacing the original order of dimensions, 0-1-2, with 1-2-0: this way, we place the channel dimension at the end. For grayscale images, this permutation is not needed. This lets us call plt.imshow from matplotlib followed by plt.show to display the image.



**Data augmentation**

Recall the dataset building code. We said that upon loading, one can apply transformations to the image, such as resizing. But many other transformations are possible, too. Let's add a random horizontal flip, and rotate by a random degree between 0 to 45. Adding random transformations to the original images is a common technique known as data augmentation. It allows us to generate more data while increasing the size and diversity of the training set. It makes the model more robust to variations and distortions commonly found in real-world images, and reduces overfitting as the model learns to ignore the random transformations. Here's a sample of augmented images using rotation.



**CONVOLUTION LAYER (CNN)**

**Why not use linear layers?**

Let's start with a linear layer. Imagine a grayscale image of 256 by 256 pixels.

It has over 65 thousand model inputs.

Using a layer with 1,000 neurons, which isn't much,

would result in over 65 million parameters!

For a color image with three times more inputs, the result is over 200 million parameters in just the first layer.

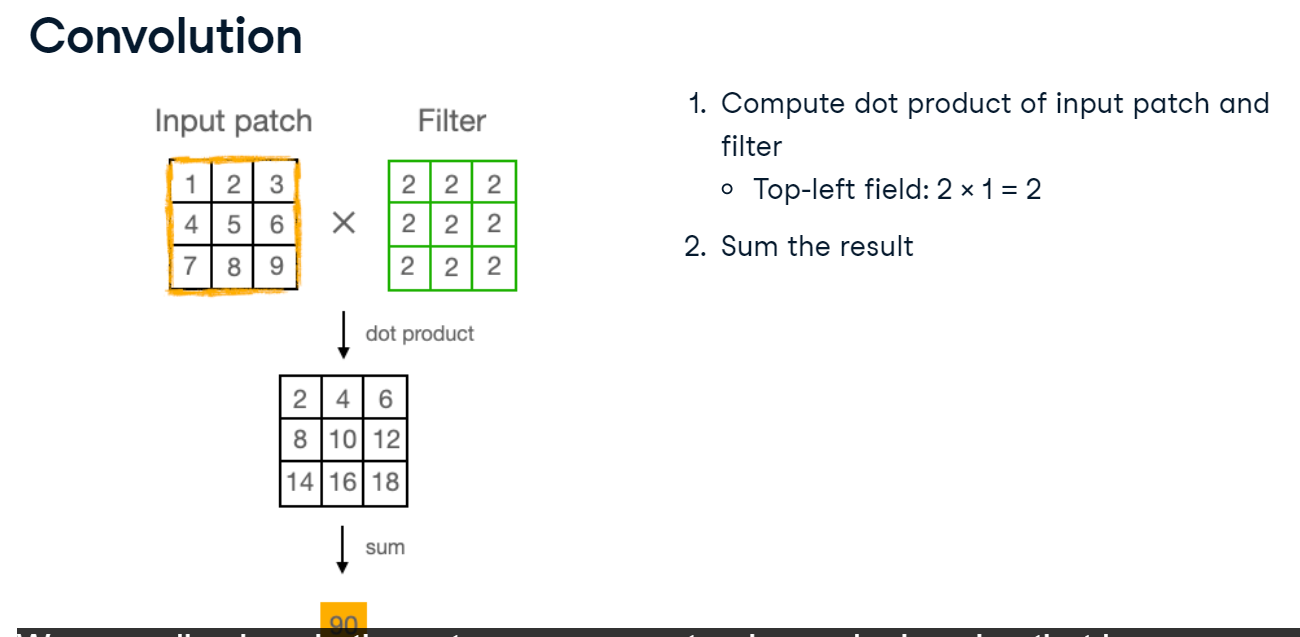
This many parameters slows down training and risks overfitting. Additionally, linear layers don't recognize spatial patterns. Consider this image with a cat in the corner. Linearly connected neurons could learn to detect the cat, but the same cat won't be recognized if it appears in a different location. When dealing with images, a better alternative is to use convolutional layers.

**Convolutional layer**

In a convolutional layer, parameters are collected in one or more small grids called filters. These filters slide over the input, performing convolution operations at each position to create a feature map. Here, we slide a 3-by-3 filter over a 5-by-5 input to get a 3-by-3 feature map. A feature map preserves spatial patterns from the input and uses fewer parameters than a linear layer. In a convolutional layer, we can use many filters. Each results in a separate feature map. Finally, we apply activations to each feature map. All the feature maps combined form the output of a convolutional layer. In PyTorch, we use nn.Conv2d to define a convolutional layer. We pass it the number of input and output feature maps, here arbitrarily chosen 3 and 32, and the kernel or filter size, 3. Let's look at the convolution operation in detail.

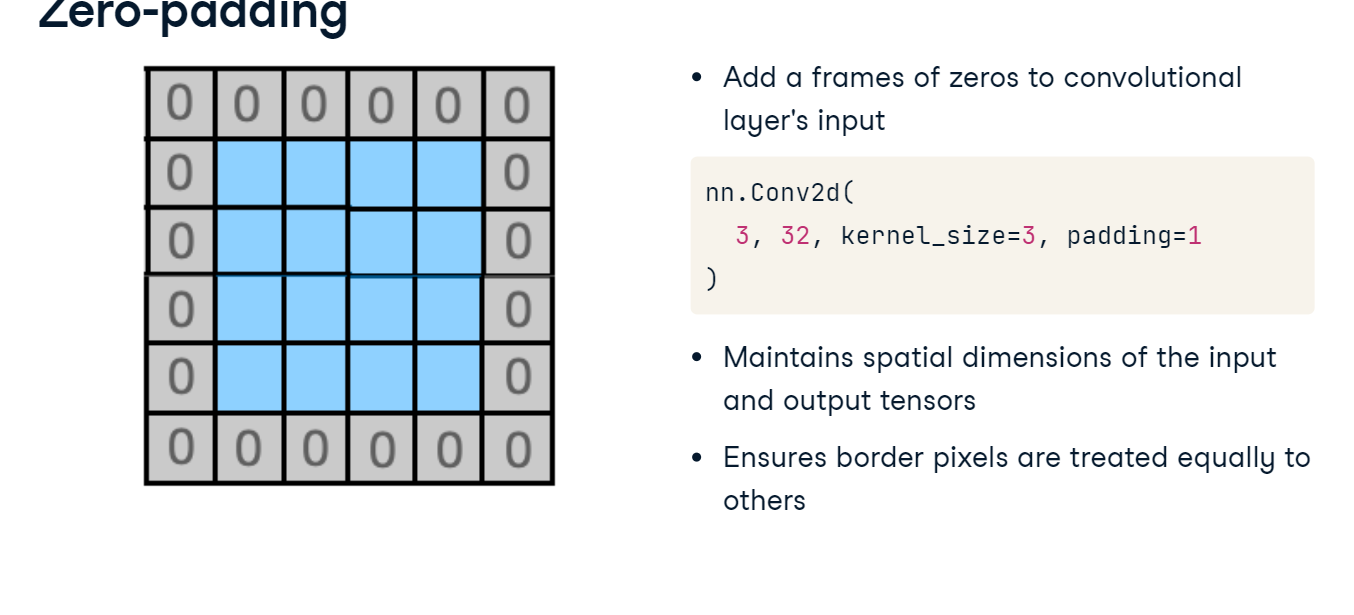
**Convolution**

In the context of deep learning, a convolution is the dot product between two arrays, the input patch and the filter. Dot product is element-wise multiplication between the corresponding elements. For instance, for the top-left field, we multiply 1 from the input patch with 2 from the filter to get 2. We sum all values in the outcome array, returning a single value that becomes part of the output feature map.



**Zero-padding**

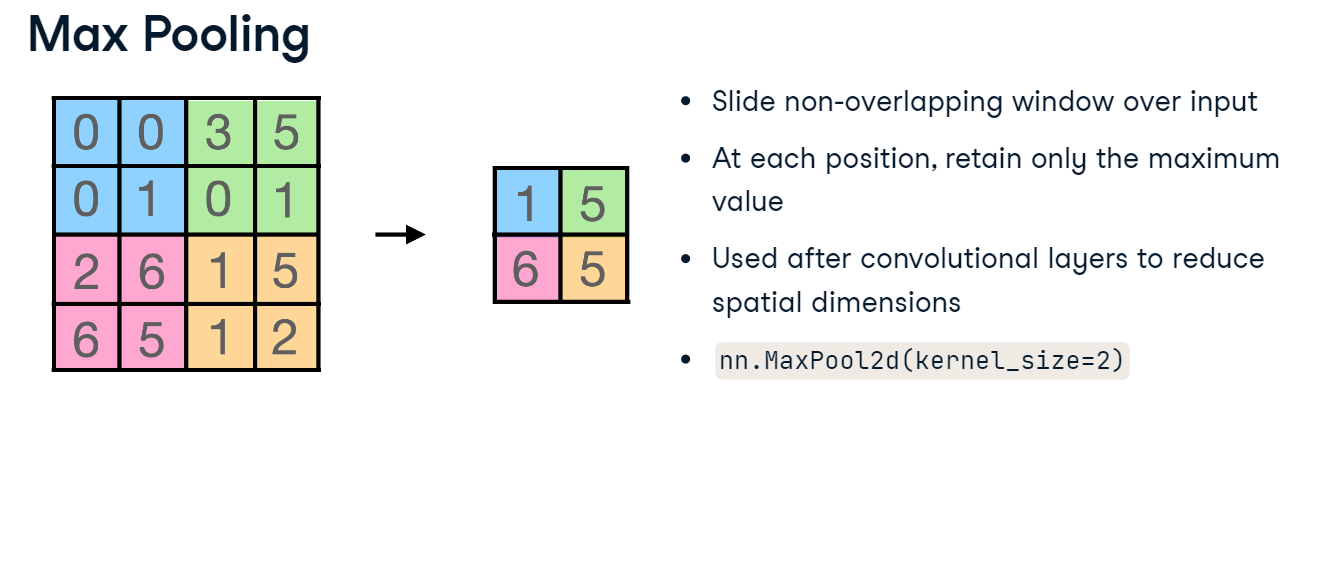
Before a convolutional layer processes its input, we often add zeros around it, a technique called zero-padding. This is done with the padding argument in the convolutional layer. It helps maintain the spatial dimensions of the input and output, and ensures equal treatment of border pixels. Without padding, the pixels at the border would have a filter slide over them fewer times resulting in information loss.



**11. Max Pooling**

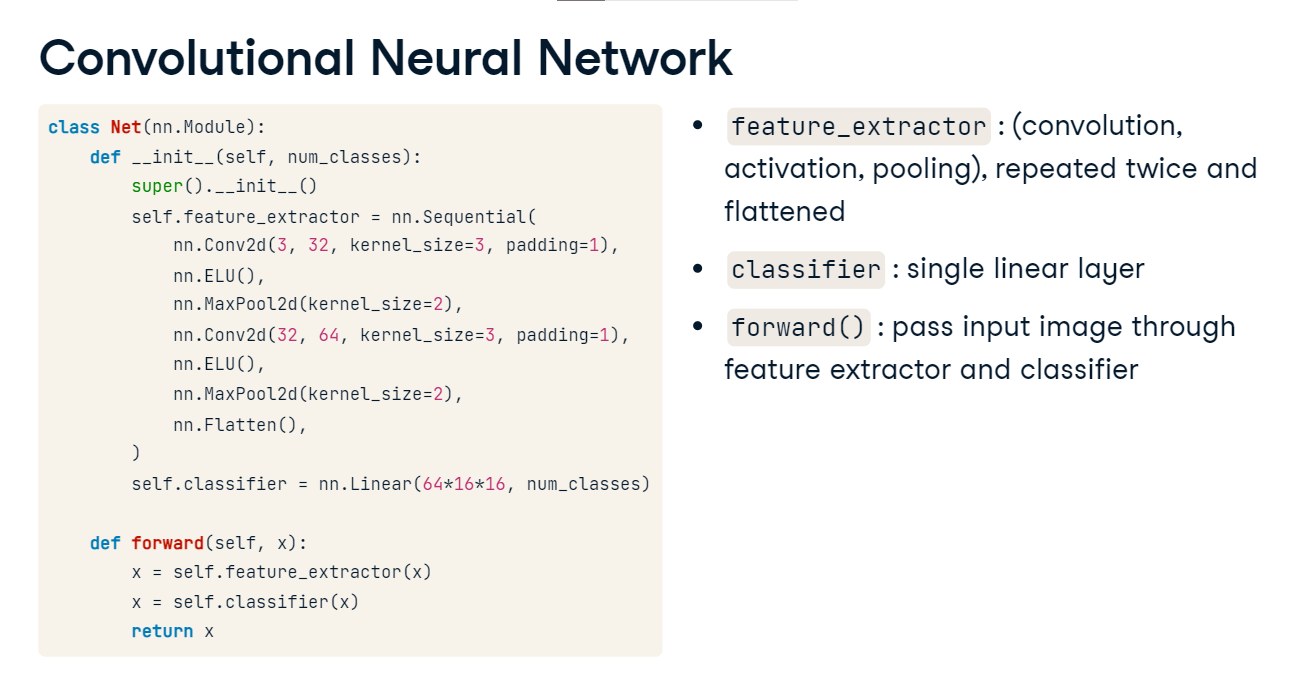
**Max Pooling**

Max pooling is another operation commonly used after convolutional layers. In it, we slide a non-overlapping window, marked by different colors here, over the input. At each position, we select the maximum value from the window to pass forward. For example, for the green window position, the maximum is five. Using a window of two-by-two as shown here halves the input's height and width. This operation reduces the spatial dimensions of the feature maps, reducing the number of parameters and computational complexity in the network. In PyTorch, we use nn.MaxPool2d to define a max pooling layer, passing it the kernel size.



**Convolutional Neural Network**

Let's build a convolutional network! It will have two parts: a feature extractor and a classifier. Feature extractor has convolution, activation, and max pooling layers repeated twice. The first two arguments in Conv2d are the numbers of input and output feature maps. The first Conv2d has three input feature maps corresponding to the RGB channels. We use filters of size 3 by 3 set by the kernel\_size argument and zero-padding by setting padding to 1. For max pooling, we use the MaxPool2d layer with a window of size 2 to halve the feature map in height and width. Finally, we flatten the feature extractor output into a vector. Our classifier consists of a single linear layer. We will discuss how we got its input size shortly. The output is the number of target classes, the model's argument. The forward method applies the extractor and classifier to the input image.



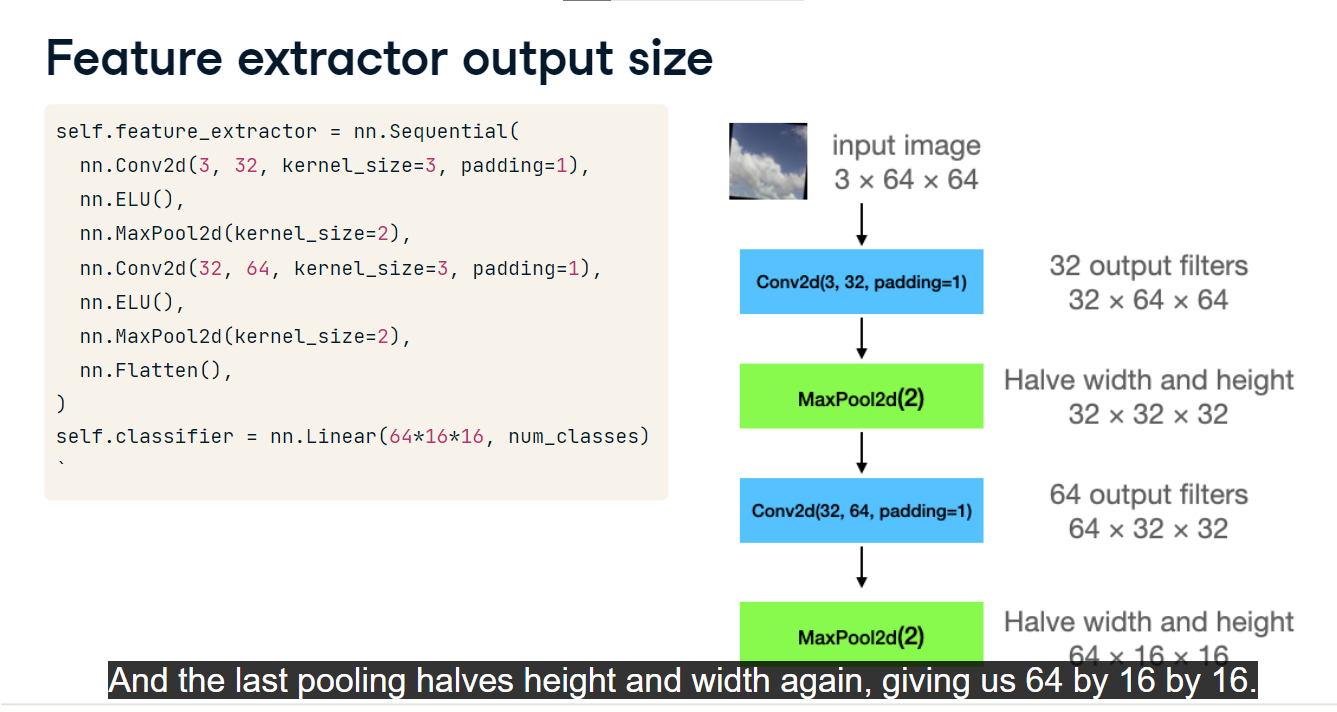
**Feature extractor output size**

To determine the feature extractor's output size, we start with the input image's size of 3 by 64 by 64.

The first convolution has 32 output feature maps, increasing the first dimension to 32. Zero-padding doesn't affect height and width.

Max pooling cuts height and width in two.

The second convolution again increases the number of feature maps in the first dimension to 64.

And the last pooling halves height and width again, giving us 64 by 16 by 16.

**Training image classifiers**

Welcome back! In this video, we will train the cloud classifier.

**Data augmentation revisited**

Before we proceed to the training itself, however, let's take one more look at data augmentation and how it can impact the training process. Say we have this image in the training data with the associated label: cat.

We apply some augmentations, for example rotation and horizontal flip, to arrive at this augmented image, and we assign it the same cat label. Both images are part of the training set now. In this example, it is clear that the augmented image still depicts a cat and can provide the model with useful information. However, this is not always the case.

**What should not be augmented**

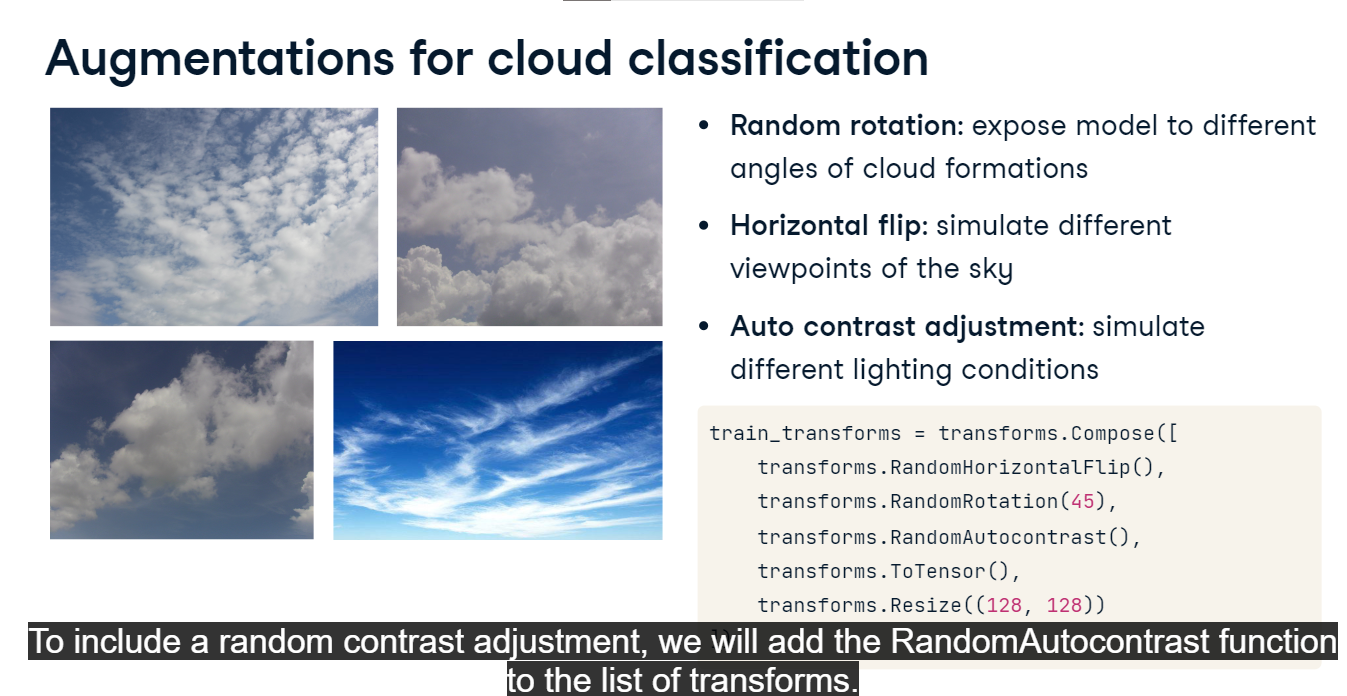
Imagine we are doing fruit classification, and decide to apply a color shift augmentation to an image of the lemon. The augmented image will still be labeled as lemon,but in fact, it will look more like a lime.

**What should not be augmented**

Another example: classification of hand-written characters. If we apply the vertical flip to the letter "W" it will look like the letter "M". Passing it to the model labeled as "W" will confuse the model and impede training. These examples show that, sometimes, specific augmentations can impact the label. It's important to notice that an augmentation could be confusing depending on the task. We could apply the vertical flip to the lemon or the color shift to the letter "W" without introducing noise in the labels. Remember to always choose augmentations with the data and task in mind!

**Augmentations for cloud classification**

So, what augmentations will be appropriate for our cloud classification task? We will use three augmentations. Random rotation will expose the model to different angles of cloud formations. Horizontal flip will simulate different viewpoints of the sky. Automatic contrast adjustment simulates different lighting conditions and improves the model's robustness to lighting variations. We have already used the RandomHorizontalFlip and RandomRotation transforms. To include a random contrast adjustment, we will add the RandomAutocontrast function to the list of transforms.

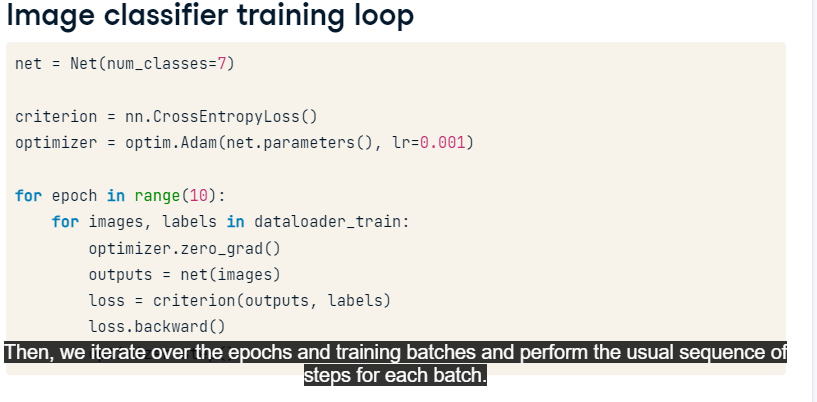


**Cross-Entropy loss**

In the clouds dataset, we have seven different cloud types, which means this is a multi-class classification task. This calls for a different loss function than we used before. The model for water potability prediction we built before was solving a binary classification task, for which the BCE or binary cross-entropy loss function is appropriate. For multi-class classification, we will need to use the cross-entropy loss. It's available in PyTorch as nn.CrossEntropyLoss.

**Image classifier training loop**

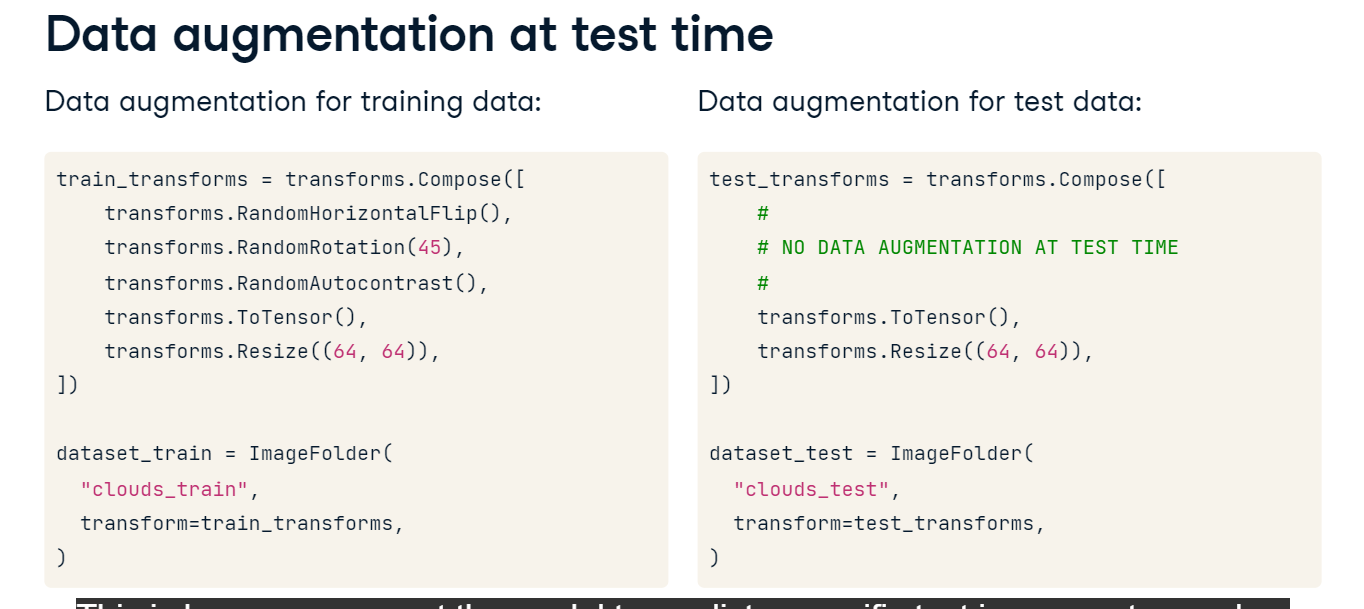
Except for the new loss function, the training loop looks the same as before. We instantiate the model we have build with seven classes and set up the cross-entropy loss and the Adam optimizer. Then, we iterate over the epochs and training batches and perform the usual sequence of steps for each batch.



# **Evaluating image classifiers**

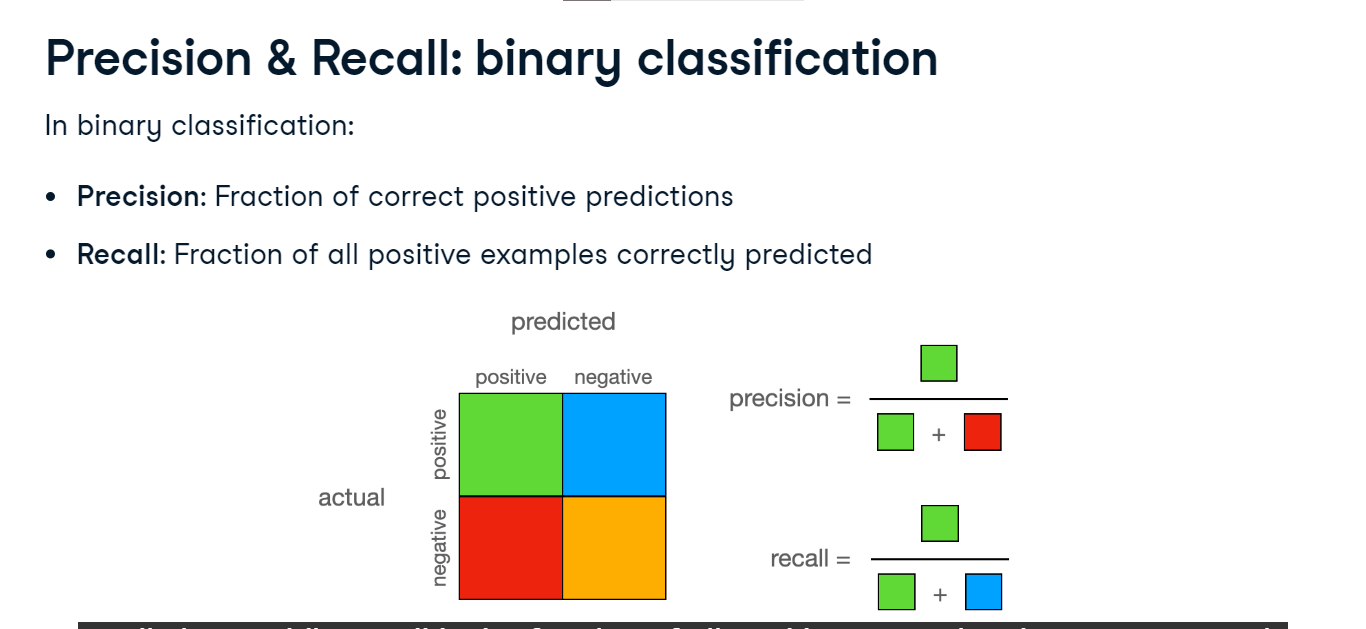
**Data augmentation at test time**

First, we need to prepare the Dataset and DataLoader for test data. But what about data augmentation? Previously we defined the training dataset passing it training transforms, including our augmentation techniques. For test data, we need to define separate transforms without data augmentation! We only keep parsing to tensor and resizing. This is because we want the model to predict a specific test image, not a random transformation of it.

****

**Precision & Recall: binary classification**

Previously, we evaluated a model based on its accuracy, which looks at the frequency of correct predictions. Let's review other metrics. In binary classification, precision is the fraction of correct positive predictions, while recall is the fraction of all positive examples that were correctly predicted.



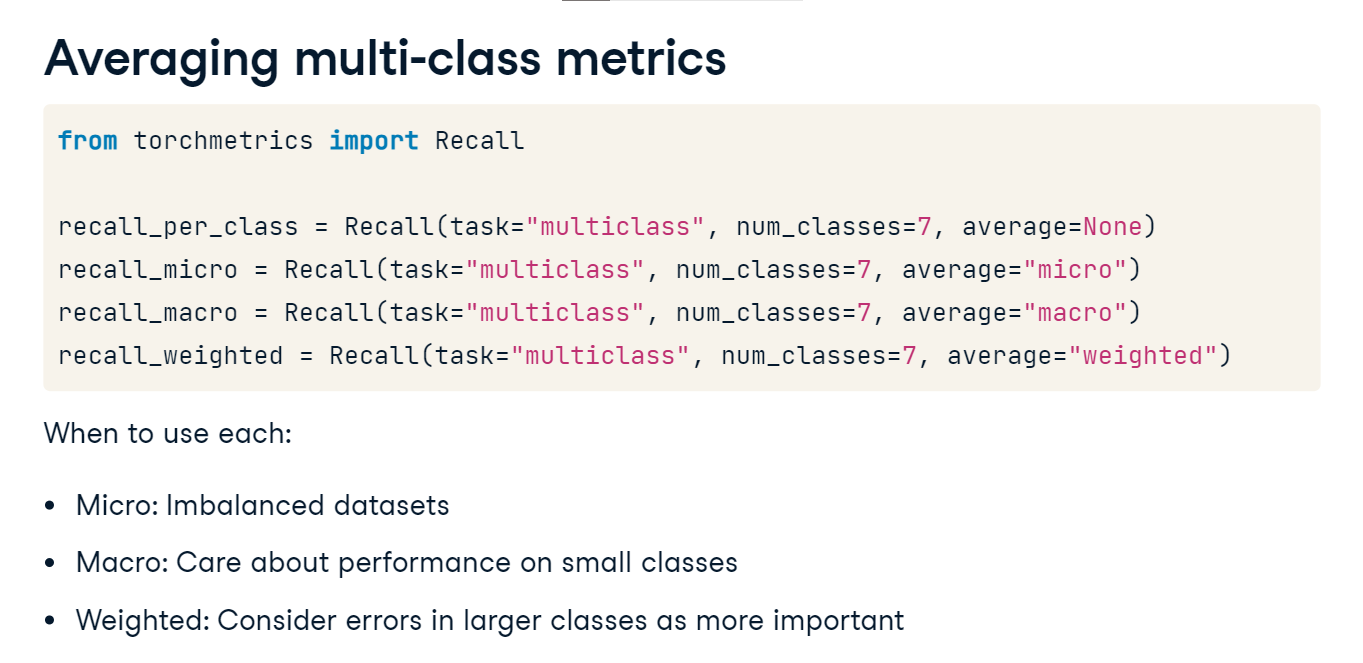
**Precision & Recall: multi-class classification**

For multi-class classification, we can get a separate recall and precision score for each class. For example, precision of the cumulus cloud class will be the fraction of cumulus-predictions that were correct, and the recall for the cumulus class will be the fraction of all cumulus clouds examples that were correctly predicted by the model.

**Averaging multi-class metrics**

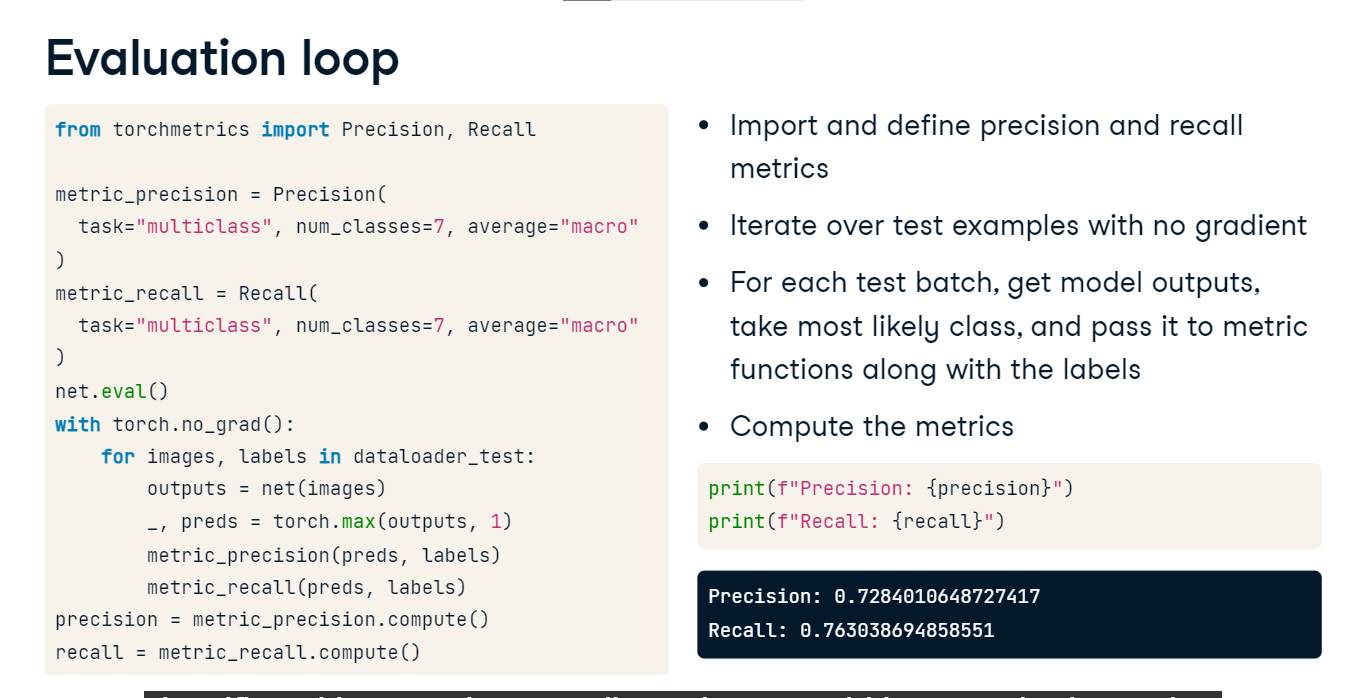
With 7 cloud classes, we have 7 precision and 7 recall scores. We can analyze them individually for each class or aggregate them. There are three ways to do so. Micro average calculates the precision and recall globally by counting the total true positives, false positives, and false negatives across all classes. It then computes the precision and recall using these aggregated values. Macro average computes the precision and recall for each class independently and takes the mean across all classes. Each class contributes equally to the final result, regardless of its size. Weighted average calculates the precision and recall for each class independently and takes the weighted mean across all classes. The weight applied is proportional to the number of samples in each class. Larger classes have a greater impact on the final result.

In PyTorch, we specify the average type when defining a metric. For example, for recall, we pass average as none to get seven recall scores, one for each class, or we can set it to micro, macro, or weighted. But when to use each of them? If our dataset is highly imbalanced, micro-average is a good choice because it takes into account the class imbalance. Macro-averaging treats all classes equally regardless of their size. It can be a good choice if you care about performance on smaller classes, even if those classes have fewer data points. Weighted averaging is a good choice when class imbalance is a concern and you consider errors in larger classes as more important.



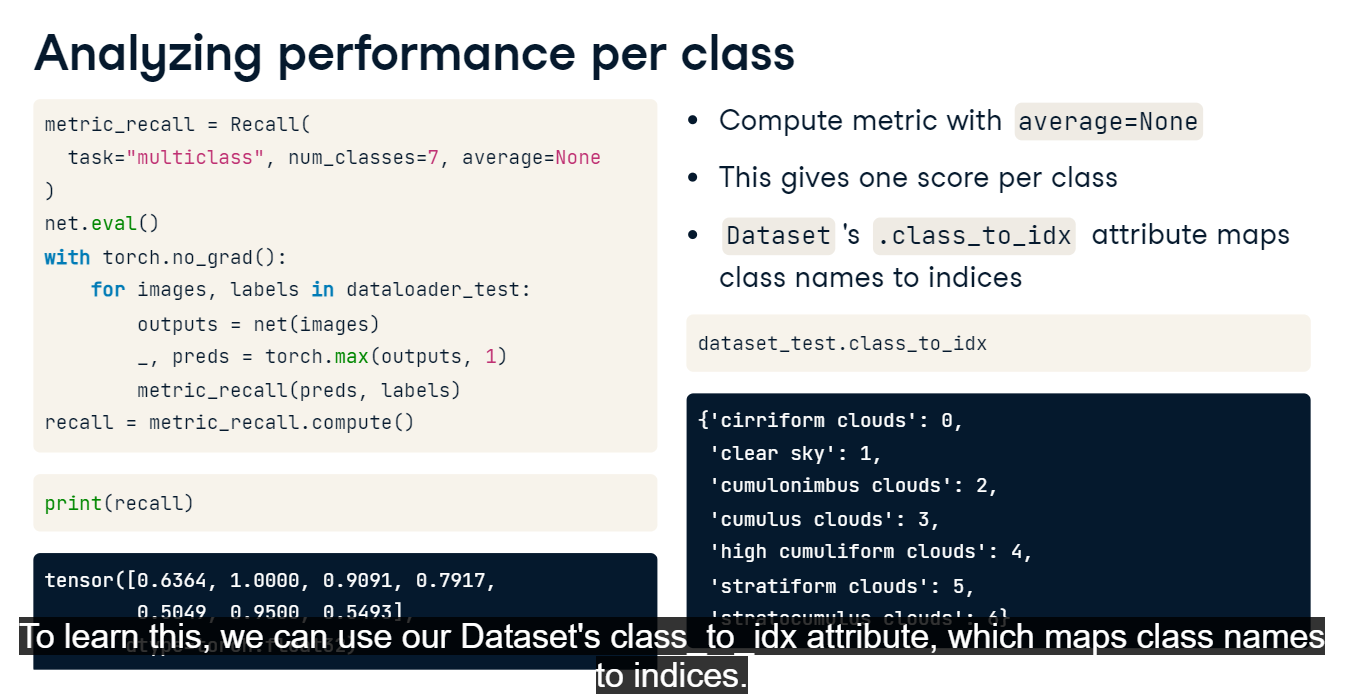
**Evaluation loop**

We start the evaluation by importing and defining precision and recall metrics. We will use macro averages for demonstration. Next, we iterate over test examples with no gradient calculation. For each test batch, we get model outputs, take the most likely class, and pass it to metric functions along with the labels. Finally, we compute the metrics and print the results. We got a recall higher than precision, meaning the model is better at correctly identifying true positives than avoiding false positives. Note that using larger images, more convolutional layers, and a classifier with more than one linear layer could improve both metrics.



**Analyzing performance per class**

Sometimes it is informative to analyze the metrics per class to compare how the model predicts specific classes. We repeat the evaluation loop with the metric defined with average equals None. This time, we only compute the recall. We get seven scores, one per class, but which score corresponds to which class? To learn this, we can use our Dataset's class\_to\_idx attribute, which maps class names to indices.



**Analyzing performance per class**

We can use a dictionary comprehension to map each class name (k) to its recall score by indexing the list of all scores called recall with the v class index from the class\_to\_idx method. This will be a tensor of length one, so we call dot-item on it to turn it into a scalar. Looking at the results, a recall of 1.0 indicates that all examples of clear sky have been classified correctly, while high cumuliform clouds were harder to classify and have the lowest recall score!

