## 1. PyTorch and object-oriented programming

00:00 - 00:05

Welcome! My name is Michał Oleszak, and I will be your instructor for this course.

## 2. What we will learn

00:05 - 00:26

We will learn how to train robust deep learning models in PyTorch. We will cover how to improve the training process with optimizers, mitigate frequent training issues of vanishing and exploding gradients, build Convolutional Neural Networks for image data, Recurrent Neural Networks for sequence data, and models with multiple inputs or outputs.

## 3. Prerequisites

00:26 - 00:44

Before starting, you should have a basic understanding of the training process for neural networks, including forward pass, loss computation, and backpropagation. You should also be able to train and evaluate basic models in PyTorch using Datasets and DataLoaders, as taught in the following course.

## 4. Object-Oriented Programming (OOP)

00:44 - 01:06

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).

## 5. Object-Oriented Programming (OOP)

01:06 - 01:41

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.

## 6. Object-Oriented Programming (OOP)

01:41 - 01:59

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.

### 7. Water potability dataset

**01:59 - 02:16**

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.

### 8. PyTorch Dataset

**02:16 - 03:07**

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.

### 9. PyTorch DataLoader

**03:07 - 03:33**

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.

### 10. PyTorch Model

**03:33 - 04:11**

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.

## PyTorch Dataset

Time to refresh your PyTorch Datasets knowledge!

Before model training can commence, you need to load the data and pass it to the model in the right format. In PyTorch, this is handled by Datasets and DataLoaders. Let's start with building a PyTorch Dataset for our water potability data.

In this exercise, you will define a class called `WaterDataset` to load the data from a CSV file. To do this, you will need to implement the three methods which PyTorch expects a Dataset to have:

- `.__init__()` to load the data,
- `.__len__()` to return data size,
- `.__getitem()__` to extract features and label for a single sample.

The following imports that you need have already been done for you:

```
import pandas as pd
from torch.utils.data import Dataset
```

```
class WaterDataset(Dataset):
   def __init__(self, csv_path):
       super().__init__()
       # Load data to pandas DataFrame
       df = pd.read_csv(csv_path)
       # Convert data to a NumPy array and assign to self.data
       self.data = df.to_numpy()

   # Implement __len__ to return the number of data samples
   def __len__(self):
       return self.data.shape[0]

   def __getitem__(self, idx):
       features = self.data[idx, :-1]
       # Assign last data column to label
       label = self.data[idx,-1]
       return features, label
```

## PyTorch DataLoader

Good job defining the Dataset class! The `WaterDataset` you just created is now available for you to use.

The next step in preparing the training data is to set up a `DataLoader`. A PyTorch `DataLoader` can be created from a `Dataset` to load data, split it into batches, and perform transformations on the data if desired. Then, it yields a data sample ready for training.

In this exercise, you will build a `DataLoader` based on the `WaterDataset`. The `DataLoader` class you will need has already been imported for you from `torch.utils.data`. Let's get to it!

```python
# Create an instance of the WaterDataset
dataset_train = WaterDataset('water_train.csv')

# Create a DataLoader based on dataset_train
dataloader_train = DataLoader(
   dataset_train,
   batch_size=2,
   shuffle=True,
)

# Get a batch of features and labels
features, labels = next(iter(dataloader_train))
print(features, labels)
```

## PyTorch Model

You will use the OOP approach to define the model architecture. Recall that this requires setting up a model class and defining two methods inside it:

- `.__init__()`, in which you define the layers you want to use;
- `forward()`, in which you define what happens to the model inputs once it receives them; this is where you pass inputs through pre-defined layers.

Let's build a model with three linear layers and ReLU activations. After the last linear layer, you need a sigmoid activation instead, which is well-suited for binary classification tasks like our water potability prediction problem. Here's the model defined using `nn.Sequential()`, which you may be more familiar with:

```python
net = nn.Sequential(
  nn.Linear(9, 16),
  nn.ReLU(),
  nn.Linear(16, 8),
  nn.ReLU(),
  nn.Linear(8, 1),
  nn.Sigmoid(),
)
```

Let's rewrite this model as a class!

```python
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
   def __init__(self):
       super().__init__()
       # Define the three linear layers
       self.fc1 = nn.Linear(9,16)
       self.fc2 = nn.Linear(16,8)
       self.fc3 = nn.Linear(8,1)

   def forward(self, x):
       # Pass x through linear layers adding activations
       x = nn.functional.relu(self.fc1(x))
       x = nn.functional.relu(self.fc2(x))
       x = nn.functional.sigmoid(self.fc3(x))
       return x
```

### 1. Optimizers, training, and evaluation

**00:00 - 00:07**

Welcome back! Let's look at model training and evaluation and the optimizer's role in the training process.

### 2. Training loop

**00:07 - 01:19**

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.

### 3. How an optimizer works

**01:19 - 01:29**

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.

### 4. How an optimizer works

**01:29 - 01:31**

They are passed to the optimizer

### 5. How an optimizer works

**01:31 - 01:34**

which computes an update for each parameter.

### 6. How an optimizer works

**01:34 - 01:45**

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.

### 7. How an optimizer works

**01:45 - 02:02**

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.

### 8. Stochastic Gradient Descent (SGD)

**02:02 - 02:17**

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.

### 9. Adaptive Gradient (Adagrad)

**02:17 - 02:40**

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.

### 10. Root Mean Square Propagation (RMSprop)

**02:40 - 02:51**

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.

### 11. Adaptive Moment Estimation (Adam)

**02:51 - 03:17**

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.

### 12. Model evaluation

**03:17 - 03:54**

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

## Optimizers

It's time to explore the different optimizers that you can use for training your model.

A custom function called `train_model(optimizer, net, num_epochs)` has been defined for you. It takes the optimizer, the model, and the number of epochs as inputs, runs the training loops, and prints the training loss at the end.

Let's use `train_model()` to run a few short trainings with different optimizers and compare the results!

```
import torch.optim as optim

net = Net()

# Define the SGD optimizer
optimizer = optim.SGD(net.parameters(), lr=0.001)
# Define the RMSprop optimizer
optimizer = optim.RMSprop(net.parameters(), lr=0.001)
# Define the Adam optimizer
optimizer = optim.Adam(net.parameters(), lr = 0.001)

train_model(
   optimizer=optimizer,
   net=net,
   num_epochs=10,
)
```

## Model evaluation

With the training loop sorted out, you have trained the model for 1000 epochs, and it is available to you as `net`. You have also set up a `test_dataloader` in exactly the same way as you did with `train_dataloader` before—just reading the data from the test rather than the train directory.

You can now evaluate the model on test data. To do this, you will need to write the evaluation loop to iterate over the batches of test data, get the model's predictions for each batch, and calculate the accuracy score for it. Let's do it!

```
import torch
from torchmetrics import Accuracy

# Set up binary accuracy metric
acc = Accuracy(task = 'binary')

net.eval()
with torch.no_grad():
    for features, labels in dataloader_test:
        # Get predicted probabilities for test data batch
        outputs = net(features)
        preds = (outputs >= 0.5).float()
        acc(preds, labels.view(-1, 1))

# Compute total test accuracy
test_accuracy = acc.compute()
print(f"Test accuracy: {test_accuracy}")
```

### 1. Vanishing and exploding gradients

**00:00 - 00:02**

Welcome back!

### 2. Vanishing gradients

**00:02 - 00:18**

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.

### 3. Exploding gradients

**00:18 - 00:28**

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

### 4. Solution to unstable gradients

**00:28 - 00:39**

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.

### 5. Weights initialization

**00:39 - 00:46**

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

### 6. Weights initialization

**00:46 - 01:14**

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.

### 7. Weights initialization

**01:14 - 01:26**

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.

### 8. He / Kaiming initialization

**01:26 - 01:43**

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.

### 9. He / Kaiming initialization

**01:43 - 01:46**

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.

### 11. Batch normalization

**02:38 - 03:27**

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.

### 12. Batch normalization

**03:27 - 03:52**

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.

## Initialization and activation

The problems of unstable (vanishing or exploding) gradients are a challenge that often arises in training deep neural networks. In this and the following exercises, you will expand the model architecture that you built for the water potability classification task to make it more immune to those problems.

As a first step, you'll improve the weights initialization by using He (Kaiming) initialization strategy. To do so, you will need to call the proper initializer from the `torch.nn.init` module, which has been imported for you as `init`. Next, you will update the activations functions from the default ReLU to the often better ELU.

```
class Net(nn.Module):
   def __init__(self):
       super().__init__()
       self.fc1 = nn.Linear(9, 16)
       self.fc2 = nn.Linear(16, 8)
       self.fc3 = nn.Linear(8, 1)

       # Apply He initialization
       init.kaiming_uniform_(self.fc1.weight)
       init.kaiming_uniform_(self.fc2.weight)
       init.kaiming_uniform_(self.fc3.weight, nonlinearity =
'sigmoid')

   def forward(self, x):
       # Update ReLU activation to ELU
       x = nn.functional.elu(self.fc1(x))
       x = nn.functional.elu(self.fc2(x))
       x = nn.functional.sigmoid(self.fc3(x))
       return x
```

## Batch Normalization

As a final improvement to the model architecture, let's add the batch normalization layer after each of the two linear layers. The batch norm trick tends to accelerate training convergence and protects the model from vanishing and exploding gradients issues.

Both `torch.nn` and `torch.nn.init` have already been imported for you as `nn` and `init`, respectively. Once you implement the change in the model architecture, be ready to answer a short question on how batch normalization works!

```
class Net(nn.Module):
   def __init__(self):
       super().__init__()
       self.fc1 = nn.Linear(9, 16)
       # Add two batch normalization layers
```

```
        self.bn1 = nn.BatchNorm1d(16)
        self.fc2 = nn.Linear(16, 8)
        self.bn2 = nn.BatchNorm1d(8)
        self.fc3 = nn.Linear(8, 1)

        init.kaiming_uniform_(self.fc1.weight)
        init.kaiming_uniform_(self.fc2.weight)
        init.kaiming_uniform_(self.fc3.weight,
nonlinearity="sigmoid")

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = nn.functional.elu(x)

        # Pass x through the second set of layers
        x = self.fc2(x)
        x = self.bn2(x)
        x = nn.functional.elu(x)

        x = nn.functional.sigmoid(self.fc3(x))
        return x
```

## Question

Which of the following statements is **true** about batch normalization?

Batch normalization effectively learns the optimal input distribution for each layer it precedes.

### 1. Handling images with PyTorch

**00:00 - 00:05**

Welcome back. Let's learn about deep learning with image data.

### 2. Clouds dataset

**00:05 - 00:17**

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

## 3. What is an image?

**00:17 - 01:11**

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.

## 4. Loading images to PyTorch

**01:11 - 01:32**

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.

## 5. Loading images to PyTorch

**01:32 - 02:06**

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.

## 6. Displaying images

**02:06 - 03:05**

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.

## 7. Data augmentation

**03:05 - 03:47**

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.

## Image dataset

Let's start with building a Torch Dataset of images. You'll use it to explore the data and, later, to feed it into a model.

The training data for the cloud classification task is stored in the following directory structure:

```
clouds_train
  - cirriform clouds
    - 539cd1c356e9c14749988a12fdf6c515.jpg
    - ...
  - clear sky
  - cumulonimbus clouds
  - cumulus clouds
  - high cumuliform clouds
  - stratiform clouds
  - stratocumulus clouds
```

There are seven folders inside `clouds_train`, each representing one cloud type (or a clear sky). Inside each of these folders sit corresponding image files.

The following imports have already been done for you:

```
from torchvision.datasets import ImageFolder
from torchvision import transforms
```

```
# Compose transformations
train_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize((128, 128)),
])

# Create Dataset using ImageFolder
dataset_train = ImageFolder(
    "clouds_train",
    transform=train_transforms,
)
```

## Data augmentation in PyTorch

Let's include data augmentation in your Dataset and inspect some images visually to make sure the desired transformations are applied.

First, you'll add the augmenting transformations to `train_transforms`. Let's use a random horizontal flip and a rotation by a random angle between 0 and 45 degrees. The code that follows to create the Dataset and the DataLoader is exactly the same as before. Finally, you'll reshape the image and display it to see if the new augmenting transformations are visible.

All the imports you need have been called for you:

```
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torchvision import transforms
import matplotlib.pyplot as plt
```

Time to augment some cloud photos!

```
train_transforms = transforms.Compose([
    # Add horizontal flip and rotation
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(45),
    transforms.ToTensor(),
    transforms.Resize((128, 128)),
])

dataset_train = ImageFolder(
  "clouds_train",
  transform=train_transforms,
)

dataloader_train = DataLoader(
  dataset_train, shuffle=True, batch_size=1)

image, label = next(iter(dataloader_train))
# Reshape the image tensor
image = image.squeeze().permute(1, 2, 0)
```

```
# Display the image
plt.imshow(image)
plt.show()
```

### 1. Convolutional Neural Networks

00:00 - 00:05

Welcome! Let's discuss neural networks for image processing.

### 2. Why not use linear layers?

00:05 - 00:12

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

### 3. Why not use linear layers?

00:12 - 00:15

It has over 65 thousand model inputs.

### 4. Why not use linear layers?

00:15 - 00:19

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

### 5. Why not use linear layers?

00:19 - 00:23

would result in over 65 million parameters!

### 6. Why not use linear layers?

00:23 - 00:30

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

### 7. Why not use linear layers?

00:30 - 00:55

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.

### 8. Convolutional layer

00:55 - 01:52

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.

### 9. Convolution

**01:52 - 02:18**

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.

### 10. Zero-padding

**02:18 - 02:41**

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

**02:41 - 03:22**

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.

### 12. Convolutional Neural Network

**03:22 - 04:20**

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.

### 13. Feature extractor output size

**04:20 - 04:28**

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

### 14. Feature extractor output size

**04:28 - 04:37**

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

### 15. Feature extractor output size

**04:37 - 04:40**

Max pooling cuts height and width in two.

### 16. Feature extractor output size

**04:40 - 04:46**

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

### 17. Feature extractor output size

**04:46 - 04:52**

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

## Building convolutional networks

You are on a team building a weather forecasting system. As part of the system, cameras will be installed at various locations to take pictures of the sky. Your task is to build a model to classify different cloud types in these pictures, which will help spot approaching weather fronts.

You decide to build a convolutional image classifier. The model will consist of two parts:

- A feature extractor that learns a vector of features from the input image,
- A classifier that predicts the image's class based on the learned features.

Both `torch` and `torch.nn as nn` have already been imported for you, so let's get to it!

```
class Net(nn.Module):
   def __init__(self, num_classes):
       super().__init__()
       # Define feature extractor
       self.feature_extractor = nn.Sequential(
           nn.Conv2d(3, 32, kernel_size=3, padding=1),
           nn.ELU(),
           nn.MaxPool2d(kernel_size=2),
           nn.Conv2d(32, 64, kernel_size=3, padding=1),
           nn.ELU(),
           nn.MaxPool2d(kernel_size=2),
           nn.Flatten(),
       )
       # Define classifier
       self.classifier = nn.Linear(64*16*16, num_classes)

   def forward(self, x):
       # Pass input through feature extractor and classifier
       x = self.feature_extractor(x)
       x = self.classifier(x)
       return x
```

### 1. Training image classifiers

**00:00 - 00:05**

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

### 2. Data augmentation revisited

**00:05 - 00:18**

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.

### 3. Data augmentation revisited

**00:18 - 00:38**

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.

### 4. What should not be augmented

**00:38 - 00:48**

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,

### 5. What should not be augmented

**00:48 - 00:51**

but in fact, it will look more like a lime.

### 6. What should not be augmented

**00:51 - 01:26**

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!

### 7. Augmentations for cloud classification

**01:26 - 02:01**

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.

### 8. Cross-Entropy loss

**02:01 - 02:31**

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.

### 9. Image classifier training loop

**02:31 - 02:48**

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

## Dataset with augmentations

You have already built the image dataset from cloud pictures and the convolutional model to classify different cloud types. Before you train it, let's adapt the dataset by adding the augmentations that could improve the model's cloud classification performance.

The code to set up the Dataset and DataLoader is already prepared for you and should look familiar. Your task is to define the composition of transforms that will be applied to the input images as they are loaded.

Note that before you were resizing images to 128 by 128 to display them nicely, but now you will use smaller ones to speed up training. As you will see later, 64 by 64 will be large enough for the model to learn.

`from torchvision import transforms` has been already executed for you, so let's get to it!

```
# Define transforms
train_transforms = transforms.Compose([
   transforms.RandomHorizontalFlip(),
   transforms.RandomRotation(45),
   transforms.RandomAutocontrast(),
   transforms.ToTensor(),
   transforms.Resize((128,128)),
])

dataset_train = ImageFolder(
 "clouds_train",
 transform=train_transforms,
)
dataloader_train = DataLoader(
 dataset_train, shuffle=True, batch_size=16
)
```

## Image classifier training loop

It's time to train the image classifier! You will use the `Net` you defined earlier and train it to distinguish between seven cloud types.

To define the loss and optimizer, you will need to use functions from `torch.nn` and `torch.optim`, imported for you as `nn` and `optim`, respectively. You don't need to change anything in the training loop itself: it's exactly like the ones you wrote before, with some additional logic to print the loss during training.

```python
# Define the model
net = Net(num_classes = 7)
# Define the loss function
criterion = nn.CrossEntropyLoss()
# Define the optimizer
optimizer = optim.Adam(net.parameters(), lr = 0.001)

for epoch in range(3):
   running_loss = 0.0
   # Iterate over training batches
   for images, labels in dataloader_train:
       optimizer.zero_grad()
       outputs = net(images)
       loss = criterion(outputs, labels)
       loss.backward()
       optimizer.step()
       running_loss += loss.item()

   epoch_loss = running_loss / len(dataloader_train)
   print(f"Epoch {epoch+1}, Loss: {epoch_loss:.4f}")
```

### 1. Evaluating image classifiers

**00:00 - 00:03**

It's time to evaluate our cloud classifier!

### 2. Data augmentation at test time

**00:03 - 00:31**

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.

### 3. Precision & Recall: binary classification

**00:31 - 00:50**

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.

### 4. Precision & Recall: multi-class classification

**00:50 - 01:11**

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.

### 5. Averaging multi-class metrics

**01:11 - 02:02**

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.

### 6. Averaging multi-class metrics

**02:02 - 02:46**

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.

### 7. Evaluation loop

**02:46 - 03:28**

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.

### 8. Analyzing performance per class

**03:28 - 03:55**

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.

### 9. Analyzing performance per class

**03:55 - 04:26**

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!

## Multi-class model evaluation

Let's evaluate our cloud classifier with precision and recall to see how well it can classify the *seven* cloud types. In this multi-class classification task it is important how you average the scores over classes. Recall that there are four approaches:

- Not averaging, and analyzing the results per class;
- Micro-averaging, ignoring the classes and computing the metrics globally;
- Macro-averaging, computing metrics per class and averaging them;
- Weighted-averaging, just like macro but with the average weighted by class size.

Both `Precision` and `Recall` are already imported from `torchmetrics`. It's time to see how well our model is doing!

```
# Define metrics
metric_precision = Precision(task="multiclass", num_classes=7,
average='macro')
metric_recall = Recall(task="multiclass", num_classes=7,
average='macro')
```

```python
net.eval()
with torch.no_grad():
    for images, labels in dataloader_test:
        outputs = net(images)
        _, preds = torch.max(outputs, 1)
        metric_precision(preds, labels)
        metric_recall(preds, labels)

precision = metric_precision.compute()
recall = metric_recall.compute()
print(f"Precision: {precision}")
print(f"Recall: {recall}")
```

## Analyzing metrics per class

While aggregated metrics are useful indicators of the model's performance, it is often informative to look at the metrics per class. This could reveal classes for which the model underperforms.

In this exercise, you will run the evaluation loop again to get our cloud classifier's precision, but this time per-class. Then, you will map these score to the class names to interpret them. As usual, `Precision` has already been imported for you. Good luck!

```python
# Define precision metric
metric_precision = Precision(
    task = 'multiclass', num_classes = 7, average = None
)

net.eval()
with torch.no_grad():
    for images, labels in dataloader_test:
        outputs = net(images)
        _, preds = torch.max(outputs, 1)
        metric_precision(preds, labels)
precision = metric_precision.compute()

# Get precision per class
precision_per_class = {
    k: precision[v].item()
    for k, v
    in dataset_test.class_to_idx.items()
}
print(precision_per_class)
```

## 1. Handling sequences with PyTorch

**00:00 - 00:06**

We've learned to handle tabular and image data. Let's now discuss sequential data.

## 2. Sequential data

**00:06 - 00:33**

Sequential data is ordered in time or space, where the order of the data points is important and can contain temporal or spatial dependencies between them. Time series, data recorded over time like stock prices, weather, or daily sales is sequential. So is text, in which the order of words in a sentence determines its meaning. Another example is audio waves, where the order of data points is crucial to the sound reproduced when the audio file is played.

## 3. Electricity consumption prediction

**00:33 - 00:53**

In this chapter, we will tackle the problem of predicting electricity consumption based on past patterns. We will use a subset of the electricity consumption dataset from the UC Irvine Machine Learning Repository. It contains electricity consumption in kilowatts, or kW, for a certain user recorded every 15 minutes for four years.

1 Trindade,Artur. (2015). ElectricityLoadDiagrams20112014. UCI Machine Learning Repository. https://doi.org/10.24432/C58C86.

## 4. Train-test split

**00:53 - 01:28**

In many machine learning applications, one randomly splits the data into training and testing sets. However, with sequential data, there are better approaches. If we split the data randomly, we risk creating a look-ahead bias, where the model has information about the future when making forecasts. In practice, we won't have information about the future when making predictions, so our test set should reflect this reality. To avoid the look-ahead bias, we should split the data by time. We will train on the first three years of data, and test on the fourth year.

## 5. Creating sequences

**01:28 - 01:59**

To feed the training data to the model, we need to chunk it first to create sequences that the model can use as training examples. First, we need to select the sequence length, which is the number of data points in one training example. Let's make each forecast based on the previous 24 hours. Because data is at 15 minute intervals, we need to use 24 times 4 which is 96 data points. In each example, the data point right after the input sequence will be the target to predict.

### 6. Creating sequences in Python

**01:59 - 02:58**

Let's implement a Python function to create sequences. It takes the DataFrame and the sequence length as inputs. We start with initializing two empty lists, xs for inputs and ys for targets. Next, we iterate over the DataFrame. The loop only goes up to "len(df) - seq_length", ensuring that for every iteration, there are always seq_length data points available in the DataFrame for creating the sequence and a subsequent data point to serve as the target. For each considered data point, we define inputs x as the electricity consumption values starting from this point plus the next sequence length points, and the target y as the subsequent electricity consumption value. The 1 passed to the iloc method stands for the second DataFrame column, which stores the electricity consumption data. Finally, we append the inputs and the target to pre-initialized lists, and after the loop, return them as NumPy arrays.

### 7. TensorDataset

**02:58 - 03:28**

Let's use our function to create sequences from the training data. This gives us almost 35 thousand training examples. To convert them to a torch Dataset, we will use the TensorDataset function. We pass it two arguments, the inputs and the targets. Each argument is the NumPy array converted to a tensor with torch.from_numpy and parsed to float. The TensorDataset behaves just like all other torch Datasets and it can be passed to a DataLoader in the same way.

### 8. Applicability to other sequential data

**03:28 - 03:52**

Everything we have learned here can also be applied to other sequential data. For example, Large Language Models are trained to predict the next word in a sentence, a problem similar to predicting the next amount of electricity used. For speech recognition, which means transcribing an audio recording of someone speaking to text, one would typically use the same sequence-processing model architectures we will learn about soon.

## Generating sequences

To be able to train neural networks on sequential data, you need to pre-process it first. You'll chunk the data into inputs-target pairs, where the inputs are some number of consecutive data points and the target is the next data point.

Your task is to define a function to do this called `create_sequences()`. As inputs, it will receive data stored in a DataFrame, `df` and `seq_length`, the length of the inputs. As outputs, it should return two NumPy arrays, one with input sequences and the other one with the corresponding targets.

As a reminder, here is how the DataFrame `df` looks like:

```
                  timestamp  consumption
0      2011-01-01 00:15:00    -0.704319
...                    ...          ...
140255 2015-01-01 00:00:00    -0.095751
```

```python
import numpy as np

def create_sequences(df, seq_length):
   xs, ys = [], []
   # Iterate over data indices
   for i in range(len(df) - seq_length):
       # Define inputs
       x = df.iloc[i:(i+seq_length), 1]
       # Define target
       y = df.iloc[i+seq_length, 1]
       xs.append(x)
       ys.append(y)
   return np.array(xs), np.array(ys)
```

## Sequential Dataset

Good job building the `create_sequences()` function! It's time to use it to create a training dataset for your model.

Just like tabular and image data, sequential data is easiest passed to a model through a torch Dataset and DataLoader. To build a sequential Dataset, you will call `create_sequences()` to get the NumPy arrays with inputs and targets, and inspect their shape. Next, you will pass them to a `TensorDataset` to create a proper torch Dataset, and inspect its length.

Your implementation of `create_sequences()` and a DataFrame with the training data called `train_data` are available.

```python
import torch
from torch.utils.data import TensorDataset

# Use create_sequences to create inputs and targets
```

```
X_train, y_train = create_sequences(train_data, 24*4)
print(X_train.shape, y_train.shape)

# Create TensorDataset
dataset_train = TensorDataset(
   torch.from_numpy(X_train).float(),
   torch.from_numpy(y_train).float(),
)
print(len(dataset_train))
```

### 1. Recurrent Neural Networks

**00:00 - 00:04**

It's time to discuss recurrent neural networks!

### 2. Recurrent neuron

**00:04 - 00:38**

So far, we built feed-forward neural networks where data is passed in one direction: from inputs, through all the layers, to the outputs. Recurrent neural networks, or RNNs, are similar, but also have connections pointing back. At each time step, a recurrent neuron receives some input x, multiplied by the weights and passed through an activation. Out come two values: the main output y, and the hidden state, h, that is fed back to the same neuron. In PyTorch, a recurrent neuron is available as nn.RNN.

### 3. Unrolling recurrent neuron through time

**00:38 - 00:58**

We can represent the same neuron once per time step, a visualization known as unrolling a neuron through time. At a given time step, the neuron represented as a gray circle receives input data x-zero and the previous hidden state h0 and produces output y-zero and a hidden state h1.

### 4. Unrolling recurrent neuron through time

**00:58 - 01:05**

At the next time step, it takes the next value x1 as input and its last hidden state, h1.

### 5. Unrolling recurrent neuron through time

**01:05 - 01:26**

And so it continues until the end of the input sequence. Since at the first time step there is no previous hidden state, h0 is typically set to zero. Notice that the output at each time step depends on all the previous inputs. This allows recurrent networks to maintain memory through time, which allows them to handle sequential data well.

## 6. Deep RNNs

**01:26 - 01:42**

We can also stack multiple layers of recurrent cells on top of each other to get a deep recurrent neural network. In this case, each input will pass through multiple neurons one after another, just like in dense and convolutional networks we have discussed before.

## 7. Sequence-to-sequence architecture

**01:42 - 02:06**

Depending on the lengths of input and output sequences, we distinguish four different architecture types. Let's look at them one by one. In a sequence-to-sequence architecture, we pass the sequence as input and make use of the output produced at every time step. For example, a real-time speech recognition model could receive audio at each time step and output the corresponding text.

## 8. Sequence-to-vector architecture

**02:06 - 02:34**

In a sequence-to-vector architecture, we pass a sequence as input, but ignore all the outputs but the last one. In other words, we let the model process the entire input sequence before it produces the output. We can use this architecture to classify text as one of multiple topics. It's a good idea to let the model "read" the whole text before it decides what it's about. We will also use the sequence-to-vector architecture for electricity consumption prediction.

## 9. Vector-to-sequence architecture

**02:34 - 02:54**

One can also build a vector-to-sequence architecture where we pass a single input and replace all other inputs with zeros but make use of all the outputs from each time step. This architecture can be used for text generation: given a single vector representing a specific topic, style, or sentiment, a model can generate a sequence of words or sentences.

## 10. Encoder-decoder architecture

**02:54 - 03:19**

Finally, in an encoder-decoder architecture, we pass the input sequences, and only then start using the output sequence. This is different from sequence-to-sequence in which outputs are generated while the inputs are still being received. A canonical use case is machine translation. One cannot translate word by word; rather the entire input must be processed before output generation can start.

## 11. RNN in PyTorch

**03:19 - 04:25**

Let's build a sequence-to-vector RNN in PyTorch. We define a model class with the init method as usual. Inside it, we assign the nn.RNN layer to self.rnn, passing it an input size of 1 since we only have one feature, the electricity consumption, an arbitrarily chosen hidden size of 32 and 2 layers, and we set batch_first to True since our data will have the batch size as its first dimension. We also define a linear layer mapping from the hidden size of 32 to the output of 1. In the forward method, we initialize the first hidden state to zeros using torch.zeros and assign it to h0. Its shape is the number of layers (2) by input size, which we extract from x as x.size-zero, by hidden state size (32). Next, we pass the input x and the first hidden state through the RNN layer. Then, we select only the last output by indexing the middle dimension with -1, pass the result through the linear layer, and return.

## Building a forecasting RNN

It's time to build your first recurrent network! It will be a sequence-to-vector model consisting of an RNN layer with two layers and a `hidden_size` of `32`. After the RNN layer, a simple linear layer will map the outputs to a single value to be predicted.

The following imports have already been done for you:

```
import torch
import torch.nn as nn
```

```
class Net(nn.Module):
   def __init__(self):
       super().__init__()
       # Define RNN layer
       self.rnn = nn.RNN(
           input_size=1,
           hidden_size=32,
           num_layers=2,
           batch_first=True,
       )
       self.fc = nn.Linear(32, 1)

   def forward(self, x):
       # Initialize first hidden state with zeros
       h0 = torch.zeros(2, x.size(0), 32)
       # Pass x and h0 through recurrent layer
       out, _ = self.rnn(x, h0)
       # Pass recurrent layer's last output through linear
layer
       out = self.fc(out[:, -1, :])
```

```
        return out
```

### 1. LSTM and GRU cells

00:00 - 00:05

Let's discuss recurrent architectures more powerful than a plain RNN.

### 2. Short-term memory problem

00:05 - 00:47

Because RNN neurons pass the hidden state from one time step to the next, they can be said to maintain some sort of memory. That's why they are often called RNN memory cells, or just cells for short. However, this memory is very short-term: by the time a long sentence is processed, the hidden state doesn't have much information about its beginning. Imagine trying to translate a sentence between languages; as soon as we have read it, we don't remember how it started. To solve this short-term memory problem, two more powerful types of cells have been proposed: the Long Short-Term Memory or LSTM cell and the Gated Recurrent Unit or GRU cell.

### 3. RNN cell

00:47 - 01:08

Before we look at LSTM and GRU cells, let's visualize the plain RNN cell. At each time step t, it takes two inputs, the current input data x and the previous hidden state h. It multiplies these inputs with the weights, applies activation, and outputs two things: the current outputs y and the next hidden state.

### 4. LSTM cell

01:08 - 01:57

The LSTM cell has three inputs and outputs. Next to the input data x, there are two hidden states: h represents the short-term memory and c the long-term memory. At each time step, h and x are passed through some linear layers called gate controllers which determine what is important enough to keep in the long-term memory. The gate controllers first erase some parts of the long-term memory in the forget gate. Then, they analyze x and h and store their most important parts in the long-term memory in the input gate. This long-term memory, c, is one of the outputs of the cell. At the same time, another gate called the output gate determines what the current output y should be. The short-term memory output h is the same as y.

### 5. LSTM in PyTorch

01:57 - 02:30

Building an LSTM network in PyTorch is very similar to the plain RNN we have already seen. In the init method, we only need to use the nn.LSTM layer instead of nn.RNN. The arguments that the layer takes as inputs are the same. In the forward method, we add the long-term hidden state c and initialize both h and c with zeros.

Then, we pass h and c as a tuple to the LSTM layer. Finally, we take the last output, pass it through the linear layer and return just like before.

### 6. GRU cell

**02:30 - 02:45**

The GRU cell is a simplified version of the LSTM cell. It merges the long-term and short-term memories into a single hidden state. It also doesn't use an output gate: the entire hidden state is returned at each time step.

### 7. GRU in PyTorch

**02:45 - 03:00**

Building a GRU network in PyTorch is almost identical to the plain RNN. All we need to do is replace the nn.rnn with nn.gru when defining the layer in the init method, and then call the new gru layer in the forward method.

### 8. Should I use RNN, LSTM, or GRU?

**03:00 - 03:40**

So, which type of recurrent network should we use: the plain RNN, LSTM, or GRU? There is no single answer, but consider the following. Although plain RNNs have revolutionized modeling of sequential data and are important to understand, they are not used much these days because of the short-term memory problem. Our choice will likely be between LSTM and GRU. GRU's advantage is that it's less complex than LSTM, which means less computation. Other than that, the relative performance of GRU and LSTM varies per use case, so it's often a good idea to try both and compare the results. We will learn how to evaluate these models soon.

## LSTM network

As you already know, plain RNN cells are not used that much in practice. A more frequently used alternative that ensures a much better handling of long sequences are Long Short-Term Memory cells, or LSTMs. In this exercise, you will be build an LSTM network yourself!

The most important implementation difference from the RNN network you have built previously comes from the fact that LSTMs have two rather than one hidden states. This means you will need to initialize this additional hidden state and pass it to the LSTM cell.

`torch` and `torch.nn` have already been imported for you, so start coding!

```
class Net(nn.Module):
   def __init__(self, input_size):
       super().__init__()
       # Define lstm layer
       self.lstm = nn.LSTM(
           input_size=1,
           hidden_size=32,
           num_layers=2,
           batch_first=True,
       )
       self.fc = nn.Linear(32, 1)

   def forward(self, x):
       h0 = torch.zeros(2, x.size(0), 32)
       # Initialize long-term memory
       c0 = torch.zeros(2, x.size(0), 32)
       # Pass all inputs to lstm layer
       out, _ = self.lstm(x, (h0, c0))
       out = self.fc(out[:, -1, :])
       return out
```

## GRU network

Next to LSTMs, another popular recurrent neural network variant is the Gated Recurrent Unit, or GRU. It's appeal is in its simplicity: GRU cells require less computation than LSTM cells while often matching them in performance.

The code you are provided with is the RNN model definition that you coded previously. Your task is to adapt it such that it produces a GRU network instead. `torch` and `torch.nn as nn` have already been imported for you.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # Define RNN layer
        self.gru = nn.GRU(
            input_size=1,
            hidden_size=32,
            num_layers=2,
            batch_first=True,
        )
        self.fc = nn.Linear(32, 1)

    def forward(self, x):
        h0 = torch.zeros(2, x.size(0), 32)
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

## 1. Training and evaluating RNNs

00:00 - 00:04

Welcome back! Let's train and evaluate our RNN.

## 2. Mean Squared Error Loss

00:04 - 00:44

Up to now, we have been solving classification tasks using cross-entropy losses. Forecasting of electricity consumption is a regression task, for which we will use a different loss function: Mean Squared Error. Here is how it's calculated. The difference between the predicted value and the target is the error. We then square it, and finally average over the batch of examples. Squaring the errors plays two roles. First, it ensures positive and negative errors don't cancel out, and second, it penalizes large errors more than small ones. Mean Squared Error loss is available in PyTorch as nn.MSELoss.

## 3. Expanding tensors

00:44 - 01:23

Before we take a look at the model training and evaluation, we need to discuss two useful concepts: expanding and squeezing tensors. Let's tackle expanding first. All recurrent layers, RNNs, LSTMs, and GRUs, expect input in the shape: batch size, sequence length, number of features. But as we loop over the DataLoader, we can see that we got the shape batch size of 32 by the sequence length of 96. Since we are dealing with only one feature, the electricity consumption, the last dimension is dropped. We can add it, or expand the tensor, by calling view on the sequence and passing the desired shape.

## 4. Squeezing tensors

01:23 - 02:00

Conversely, as we evaluate the model, we will need to revert the expansion we have applied to the model inputs which can be achieved through squeezing. Let's see why that's the case and how to do it. As we iterate through test data batches, we get labels in shape batch size. Model outputs, however, are of shape batch size by 1, our number of features. We will be passing the labels and the model outputs to the loss function, and each PyTorch loss requires its inputs to be of the same shape. To achieve that, we can apply the squeeze method to the model outputs. This will reshape them to match the labels' shape.

## 5. Training loop

02:00 - 02:18

The training loop is similar to what we have already seen. We instantiate the model and define the loss and the optimizer. Then, we iterate over epochs and training data batches. For each batch, we reshape the input sequence as we have just discussed. The rest of the training loop is the same as before.

### 6. Evaluation loop

**02:18 - 02:47**

Let's look at the evaluation loop. We start by setting up the Mean Squared Error metric from torchmetrics. Then, we iterate through test data batches without computing the gradients. Next, we reshape the model inputs just like during training, pass them to the model, and squeeze the outputs. Finally, we update the metric. After the loop, we can print the final metric value by calling compute on it, just like we did before.

### 7. LSTM vs. GRU

**02:47 - 03:15**

Here is our LSTM's test Mean Squared Error again. Let's see how it compares to a GRU network. It seems that for our electricity consumption dataset, with the task defined as predicting the next value based on the previous 24 hours of data, both models perform similarly, with GRU achieving even a slightly lower error. In this case, GRU might be preferred as it achieves the same or better results while requiring less processing power.

## RNN training loop

It's time to train the electricity consumption forecasting model!

You will use the LSTM network you have defined previously, which has been instantiated and assigned to `net`, as is the `dataloader_train` you built before. You will also need to use `torch.nn` which has already been imported as `nn`.

In this exercise, you will train the model for only three epochs to make sure the training progresses as expected. Let's get to it!

```
net = Net()
# Set up MSE loss
criterion = nn.MSELoss()
optimizer = optim.Adam(
 net.parameters(), lr=0.0001
)

for epoch in range(3):
   for seqs, labels in dataloader_train:
       # Reshape model inputs
       seqs = seqs.view(32, 96, 1)
       # Get model outputs
       outputs = net(seqs)
       # Compute loss
       loss = criterion(outputs, labels)
       optimizer.zero_grad()
       loss.backward()
       optimizer.step()
   print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

## Evaluating forecasting models

It's evaluation time! The same LSTM network that you have trained in the previous exercise has been trained for you for a few more epochs and is available as `net`.

Your task is to evaluate it on a test dataset using the Mean Squared Error metric (`torchmetrics` has already been imported for you). Let's see how well the model is doing!

```
# Define MSE metric
mse = torchmetrics.MeanSquaredError()

net.eval()
with torch.no_grad():
    for seqs, labels in dataloader_test:
        seqs = seqs.view(32, 96, 1)
        # Pass seqs to net and squeeze the result
        outputs = net(seqs).squeeze()
        mse(outputs, labels)

# Compute final metric value
test_mse = mse.compute()
print(f"Test MSE: {test_mse}")
```

### 1. Multi-input models

**00:00 - 00:03**

Let's learn to build multi-input models!

### 2. Why multi-input?

**00:03 - 00:49**

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!

### 3. Omniglot dataset

**00:49 - 00:59**

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

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

### 4. Character classification

**00:59 - 01:10**

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".

### 5. Character classification

**01:10 - 01:16**

The second input will the the alphabet that it comes from expressed as a one-hot vector.

### 6. Character classification

**01:16 - 01:21**

Both inputs will be processed separately, then we concatenate their representations.

### 7. Character classification

**01:21 - 01:33**

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.

### 8. Two-input Dataset

**01:33 - 02:33**

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.

### 9. Tensor concatenation

**02:33 - 02:52**

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.

### 10. Two-input architecture

**02:52 - 03:34**

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.

### 11. Two-input architecture

**03:34 - 03:47**

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.

### 12. Training loop

**03:47 - 04:00**

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.

## Two-input dataset

Building a multi-input model starts with crafting a custom dataset that can supply all the inputs to the model. In this exercise, you will build the Omniglot dataset that serves triplets consisting of:

- The image of a character to be classified,
- The one-hot encoded alphabet vector of length 30, with zeros everywhere but for a single one denoting the ID of the alphabet the character comes from,
- The target label, an integer between 0 and 963.

You are provided with `samples`, a list of 3-tuples comprising an image's file path, its alphabet vector, and the target label. Also, the following imports have already been done for you, so let's get to it!

```
class OmniglotDataset(Dataset):
   def __init__(self, transform, samples):
   # Assign transform and samples to class attributes
       self.transform = transform
       self.samples = samples

   def __len__(self):
   # Return number of samples
       return len(self.samples)

   def __getitem__(self, idx):
       # Unpack the sample at index idx
       img_path, alphabet, label = self.samples[idx]
       img = Image.open(img_path).convert('L')
       # Transform the image
       img_transformed = self.transform(img)
       return img_transformed, alphabet, label
```

## Two-input model

With the data ready, it's time to build the two-input model architecture! To do so, you will set up a model class with the following methods:

- `.__init__()`, in which you will define sub-networks by grouping layers; this is where you define the two layers for processing the two inputs, and the classifier that returns a classification score for each class.
- `forward()`, in which you will pass both inputs through corresponding pre-defined sub-networks, concatenate the outputs, and pass them to the classifier.

`torch.nn` is already imported for you as `nn`. Let's do it!

```
class Net(nn.Module):
   def __init__(self):
       super().__init__()
       # Define sub-networks as sequential models
       self.image_layer = nn.Sequential(
           nn.Conv2d(1, 16, kernel_size=3, padding=1),
           nn.MaxPool2d(kernel_size=2),
           nn.ELU(),
           nn.Flatten(),
           nn.Linear(16*32*32, 128)
       )
       self.alphabet_layer = nn.Sequential(
           nn.Linear(30, 8),
           nn.ELU(),
       )
       self.classifier = nn.Sequential(
           nn.Linear(128 + 8, 964),
       )

   def forward(self, x_image, x_alphabet):
       # Pass the x_image and x_alphabet through appropriate
layers
       x_image = self.image_layer(x_image)
       x_alphabet = self.alphabet_layer(x_alphabet)
       # Concatenate x_image and x_alphabet
       x = torch.cat((x_image, x_alphabet), dim=1)
       return self.classifier(x)
```

### 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.

## Two-output Dataset and DataLoader

In this and the following exercises, you will build a two-output model to predict both the character and the alphabet it comes from based on the character's image. As always, you will start with getting the data ready.

The `OmniglotDataset` class you have created before is available for you to use along with updated `samples`. Let's use it to build the Dataset and the DataLoader.

The following imports have already been done for you:

```
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
```

```
# Print the sample at index 100
print(samples[100])

# Create dataset_train
dataset_train = OmniglotDataset(
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Resize((64, 64)),
    ]),
    samples=samples,
)

# Create dataloader_train
dataloader_train = DataLoader(
    dataset_train, shuffle =True , batch_size =32
)
```

## Two-output model architecture

In this exercise, you will construct a multi-output neural network architecture capable of predicting the character and the alphabet.

Recall the general structure: in the `.__init__()` method, you define layers to be used in the forward pass later. In the `forward()` method, you will first pass the input image through a couple of layers to obtain its embedding, which in turn is fed into two separate classifier layers, one for each output.

`torch.nn` is already imported under its usual alias, so let's build a model!

```
class Net(nn.Module):
   def __init__(self):
       super().__init__()
       self.image_layer = nn.Sequential(
           nn.Conv2d(1, 16, kernel_size=3, padding=1),
           nn.MaxPool2d(kernel_size=2),
           nn.ELU(),
           nn.Flatten(),
           nn.Linear(16*32*32, 128)
       )
       # Define the two classifier layers
       self.classifier_alpha = nn.Linear(128, 30)
       self.classifier_char = nn.Linear(128, 964)

   def forward(self, x):
       x_image = self.image_layer(x)
       # Pass x_image through the classifiers and return both
results
       output_alpha = self.classifier_alpha(x_image)
       output_char = self.classifier_char(x_image)
       return output_alpha, output_char
```

## Training multi-output models

When training models with multiple outputs, it is crucial to ensure that the loss function is defined correctly.

In this case, the model produces two outputs: predictions for the alphabet and the character. For each of these, there are corresponding ground truth labels, which will allow you to calculate two separate losses: one incurred from incorrect alphabet classifications, and the other from incorrect character classification. Since in both cases you are dealing with a multi-label classification task, the Cross-Entropy loss can be applied each time.

Gradient descent can optimize only one loss function, however. You will thus define the total loss as the sum of alphabet and character losses.

```
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.05)

for epoch in range(1):
   for images, labels_alpha, labels_char in dataloader_train:
       optimizer.zero_grad()
       outputs_alpha, outputs_char = net(images)
       # Compute alphabet classification loss
       loss_alpha = criterion (outputs_alpha, labels_alpha)
       # Compute character classification loss
       loss_char = criterion (outputs_char, labels_char)
       # Compute total loss
       loss = loss_alpha + loss_char
       loss.backward()
       optimizer.step()
```

### 1. Evaluation of multi-output models and loss weighting

**00:00 - 00:09**

Welcome back! In this final video of the course, we will discuss loss weighting and evaluation of multi-output models. Let's dive in!

### 2. Model evaluation

**00:09 - 01:04**

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.

### 3. Multi-output training loop revisited

**01:04 - 01:37**

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.

### 4. Varying task importance

**01:37 - 02:05**

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.

### 5. Warning: losses on different scales

**02:05 - 02:57**

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.

## Multi-output model evaluation

In this exercise, you will practice model evaluation for multi-output models. Your task is to write a function called `evaluate_model()` that takes an alphabet-and-character-predicting model as input, runs the evaluation loop, and prints the model's accuracy in the two tasks.

You can assume that the function will have access to `dataloader_test`. The following imports have already been run for you:

```
import torch
from torchmetrics import Accuracy
```

Once you have implemented `evaluate_model()`, you will use it in the following exercise!

```
def evaluate_model(model):
   # Define accuracy metrics
   acc_alpha = Accuracy(task="multiclass", num_classes=30)
   acc_char = Accuracy(task="multiclass", num_classes=964)

   model.eval()
   with torch.no_grad():
       for images, labels_alpha, labels_char in
dataloader_test:
           # Obtain model outputs
           outputs_alpha, outputs_char = model(images)
           _, pred_alpha = torch.max(outputs_alpha, 1)
           _, pred_char = torch.max(outputs_char, 1)
           # Update both accuracy metrics
           acc_alpha(pred_alpha, labels_alpha)
           acc_char(pred_char, labels_char)

   print(f"Alphabet: {acc_alpha.compute()}")
   print(f"Character: {acc_char.compute()}")
```