Data Science and Machine Learning Lab
Lab 06 - PyTorch Introduction

Politecnico di Torino

Intro

In this laboratory, you will learn the fundamental components of the PyTorch framework, one of the most widely used libraries for deep learning. By the end of this lab, you will be able to:

- Create and manipulate tensors using PyTorch.
- Load, transform, and use datasets with `Dataset` and `DataLoader`.
- Define and train simple neural network models using `nn.Module`.
- Understand the role of loss functions, optimizers, and training loops.

Note that exercises marked with a (*) are optional; you should focus on completing the other ones first.

1 Warm-up: PyTorch tensors

In PyTorch, the fundamental data structure is the `torch.Tensor`. It is similar to a NumPy `ndarray`, but it extends its functionality in several key ways. First, a tensor can live either on the CPU or on the GPU, allowing computations to be executed efficiently on different hardware without changing the code. More importantly, PyTorch tensors are integrated with the library's **automatic differentiation engine** (`autograd`), which can compute derivatives of any function built from tensor operations automatically. This feature enables the implementation of *backpropagation* with minimal effort: once a computational graph is defined, calling `loss.backward()` automatically calculates the gradients of all parameters involved, eliminating the need to derive or code them manually (we will take a look at this later!).

1.1 Tensor creation and inspection

In this section, you will learn how to create and manipulate PyTorch tensors, exploring their attributes and behavior through simple examples. Start by importing the `torch` library and creating a few small tensors manually to understand their structure and key properties:

- **Creating tensors:** Define a one-dimensional tensor containing a few floating-point values. Then, create a second tensor of the same shape and perform a simple element-wise operation (for example, addition or multiplication). Print the resulting tensor and verify its contents.
- **Inspecting tensor attributes:** Examine the tensor's shape, number of dimensions, and data type using the appropriate attributes. Check which precision is used by default and compare it to NumPy's behavior.
- **Changing tensor data types:** Experiment with converting a tensor to another data type using either the `.to()` method or the shorthand functions (for example, `.float()` or `.double()`). The `.to()` method is a versatile function that allows you to move tensors between devices (CPU and GPU) or change their properties, such as data type and precision, by passing the desired `torch.dtype` as an argument. For example, you can use `tensor.to(torch.float64)` to convert a tensor to double precision, or `tensor.to("cuda")` to move it to the GPU. After applying the conversion, verify that the tensor's `dtype` changes accordingly.

- **Performance comparison of data types:** Using the `torch.randn` function, create two large random matrices (for example, of size 1000×1000) and measure the time required to perform a matrix multiplication in both `float32` and `float64`. After these analyses, why do you think PyTorch uses `float32` as the default type instead of `float64`?

Info: You can use the `timeit` function by repeatedly running the matrix multiplication (for example, 100 times) and comparing the total times obtained for the two data types. This is a snippet of how you can use `timeit` actually to make this test.

```
from timeit import timeit

# Example timing for float64 and float32 matrix multiplication
t64 = timeit(lambda: M1_64 @ M2_64, number=100)
t32 = timeit(lambda: M1_32 @ M2_32, number=100)

print(f"Time for matrix multiplication (float64): {t64:.4f}s")
print(f"Time for matrix multiplication (float32): {t32:.4f}s")
```

2 Datasets and Dataloaders

Usually, we don't work with "just one tensor". In most cases, we have *many* samples consisting of inputs and corresponding labels, and we need a simple way to access them for training. PyTorch provides this through `Dataset`, which enables easy handling of datasets.

Once the dataset is created, we wrap it into a `DataLoader`, which provides an easy and efficient way to iterate over the data in batches. The `DataLoader` automatically groups samples, handles shuffling, and prepares batches ready to be fed to the model during training.

2.1 Synthetic regression dataset

In this first part, instead of defining a custom dataset class, we will use the built-in `TensorDataset`, which conveniently pairs input and target tensors so that each element of the dataset directly corresponds to a sample (X, y). Your goal is to generate a one-dimensional regression problem and then use a `DataLoader` to iterate over it.

- **Dataset generation:** Create a dataset containing $n = 2048$ samples. Use the `torch.randn` function to generate a one-dimensional input tensor X of shape $(n, 1)$, representing normally distributed random features. Then, define the target tensor y as a linear transformation of X with added Gaussian noise according to the relation:

  $$y = 5x + 3 + \varepsilon$$

  where ε is a small random noise term sampled from a normal distribution. Verify that both X and y have the expected shapes and compatible data types.

- **Creating a Dataset object:** Wrap the tensors X and y into a `TensorDataset` object. This class is a convenient way to combine multiple tensors into a dataset where each element corresponds to a pair (X_i, y_i). Check that indexing the dataset (for example, `dataset[0]`) returns a tuple containing one sample and its label.

- **Building a DataLoader and inspect batches:** Create a `DataLoader` from your dataset to enable efficient iteration in batches. Set a batch size of 256 samples and enable shuffling (`shuffle=True`). Iterate once over your `DataLoader` and print the shapes of the input and target batches. Verify that the shapes are consistent with the chosen batch size (for example, `[256, 1]` for both X and y). Then, change the `batch_size` parameter and observe how the shapes of the resulting batches change.

3 Building and understanding a simple linear model

In this section, you will implement and analyze a simple linear model in PyTorch to train it on the dataset you have just created. A univariate linear model is one of the simplest forms of a machine learning model and can be defined by the equation:

$$y = wx + b$$

where x is the input feature, w is the weight, b is the bias, and y is the predicted output.

3.1 Model definition: `SimpleLinearModel`

The `SimpleLinearModel` class is a minimal example of a learnable model implemented using PyTorch's module interface. It extends the base class `nn.Module`, which is used to define all neural networks in PyTorch. The model consists of a single linear layer that performs the univariate linear transformation.

- **Create the class structure:** Define a class `SimpleLinearModel` inheriting from `nn.Module`. Inside the constructor (`__init__`), create a single linear layer using `nn.Linear(input_size, output_size)`. In this exercise, both the input and output sizes are equal to 1, meaning that the model will learn to map one scalar input to a single scalar output.
- **Define the forward pass:** Implement the `forward()` method to define how the input data flows through the layer. This method should take an input tensor `x` and return the result of applying the linear transformation to it. This defines how PyTorch will compute the model's prediction during both training and inference.
- **Initialize and analyze your model:** Once the class is defined, create an instance of the model and test it with a single input value. Check that the model returns an output tensor of the correct shape. Inspect also the parameters of the model (the weight and bias) and note that they are randomly initialized. These parameters have the attribute `requires_grad=True`, meaning that PyTorch will automatically track their gradients during the backward pass.

Info: Example of model creation and test:

```
class SimpleLinearModel(nn.Module):
    def __init__(self, input_size, output_size):
        super(SimpleLinearModel, self).__init__()
        self.linear = ...

    def forward(self, x):
        return ...

# Instantiate the model
model = ...
```

3.2 Criterion and optimizer

During training, two components control the learning process: the **criterion** and the **optimizer**.

- **Criterion (loss function):** The loss function measures the discrepancy between the model's predictions and the true labels. In this exercise, since the goal is to predict a continuous value, we will use the `Mean Squared Error (MSE)` loss.
- **Optimizer:** The optimizer updates the model's parameters using the computed gradients. For this exercise, we will use `Stochastic Gradient Descent (SGD)` with a learning rate of 0.01. The optimizer adjusts the weights and bias in the direction that minimizes the loss.

3.3 Training loop

The **training loop** is the process through which the model learns from data. It involves multiple epochs, where each epoch corresponds to one full pass over the dataset. For each batch, the following sequence of steps occurs:

- **Forward pass:** Feed inputs through the model to obtain predictions.
- **Loss computation:** Compare predictions to true labels using the loss function.
- **Backward pass:** Compute gradients of the loss with respect to model parameters.
- **Parameter update:** Use the optimizer to update weights and biases.
- **Gradient reset:** Clear old gradients using `optimizer.zero_grad()` to prevent accumulation.

Info: Example of a basic training loop:

```
num_epochs = 50
losses, weights, biases = [], [], []

model.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in trainloader:
        inputs, labels = ...
        optimizer.zero_grad()

        # Forward pass
        outputs = ...
        loss = ...

        # Backward pass and optimization
        ...

        # Track values
        losses.append(loss.item())
        weights.append(model.linear.weight.item())
        biases.append(model.linear.bias.item())
        running_loss += loss.item()

    print(f"Epoch [{epoch+1}/{num_epochs}] - Loss: {running_loss/len(
    trainloader):.4f}")
```

Info: Note that we are using `loss.item()` to extract the underlying scalar value from the tensor. In this way, we store a Python float rather than a tensor object. Since `loss` is no longer being used, the corresponding computational graph will be discarded, avoiding unnecessary memory usage.

After training, plot the evolution of the loss, weight, and bias to visualize the learning process.

4 MNIST dataset

In this section, you will load and explore the **MNIST** dataset, a collection of handwritten digits from 0 to 9 that you have already encountered in previous laboratories. PyTorch provides a ready-to-use interface through `torchvision.datasets.MNIST`, which automatically downloads and organizes the data.

Info: You can easily load the MNIST training and test splits using the following snippet:

```
from torchvision import datasets, transforms

train_dataset = datasets.MNIST(root="data", train=True, download=True)
test_dataset = datasets.MNIST(root="data", train=False, download=True)
```

- **Dataset size and Accessing samples:** Check the number of samples contained in the training and test datasets by printing their lengths. Each element i of the dataset can be accessed as a pair `(image, label)` simply using `train_dataset[i]`. Retrieve one element and inspect its structure and the shape of the tensors.
- **Visual inspection:** Plot a small grid of sample images together with their corresponding labels to confirm that you can correctly access both the data and its annotations. Remember that MNIST images are stored as grayscale tensors, so you should display them using the parameter `cmap='gray'` when calling `imshow()` to avoid color distortions.

4.1 Preprocessing and transforms

Before training a model, image data must often be transformed into a suitable numerical format and standardized for more stable learning. In PyTorch, this process is handled by the `torchvision.transforms` module, which provides a collection of ready-to-use operations that can be composed together in a pipeline.

- **Conversion to tensors:** Raw MNIST images are loaded as PIL images. The `ToTensor()` transform converts them into PyTorch tensors of shape $1 \times 28 \times 28$, automatically scaling pixel values from the range $[0, 255]$ to $[0, 1]$. The first dimension, 1, represents the number of "channels" of the image: since the images are grayscale, there is a single color channel. Colored images are typically represented using 3 channels (Red, Green, Blue, or RGB).
- **Normalization:** To make learning more efficient, it is common to normalize the data so that pixel intensities have approximately zero mean and unit variance. This is achieved using the transform:

$$x_{\text{norm}} = \frac{x - \mu}{\sigma}$$

  where $\mu = 0.1307$ and $\sigma = 0.3081$ are the empirical mean and std of the MNIST dataset.
- **Data augmentation:** Simple transformations, such as small random rotations, can be added to slightly modify the images during training. These are well-known data augmentation techniques, that help the model better generalize and be more robust to spatial variations.

Info: The following snippet shows how to define a preprocessing pipeline using `transforms.Compose`:

```
from torchvision import transforms

transform = transforms.Compose([
    transforms.RandomRotation(45),
    ...
    # add the transforms to convert to tensor and apply the
    normalization
])
```

- **Applying a transform to a single image:** Retrieve one image from the dataset and apply the defined transform directly to it. This will convert the image into a normalized tensor that can be fed into a model. Verify that the transformed image is now a tensor with shape $[1, 28, 28]$.

- **Applying a transform to an existing dataset:** If a dataset was created without a transform, you can still attach one afterwards by assigning it to the `transform` attribute (`train_dataset.transform = transform`). From this point on, every time a sample is retrieved, the transform will be applied automatically.
- **(*) Experimenting with transformations:** Extend the preprocessing pipeline by adding and testing other transformations available in `torchvision.transforms`. Refer to the official documentation to explore the various possible transformations and understand the functionality of each one. Try including operations such as `RandomAffine`, `RandomCrop`, or `ColorJitter`, and visualize a few transformed images to observe their effects. Reflect on which transformations are appropriate for MNIST and which could distort the digits too much for reliable classification.

5 A more complex neural network

In this final exercise, you will extend the concepts learned so far by defining and training a **multi-layer neural network** using PyTorch. This will allow you to understand how deeper architectures can model more complex relationships in the data compared to the single-layer models explored earlier.

5.1 Model definition: `SimpleNN`

You will define a simple feedforward neural network composed of multiple fully connected (`Linear`) layers. Each layer performs a linear transformation followed by a non-linear activation function, introducing flexibility into the model.

- **Define the class:** Define a class `SimpleNN` that inherits from `nn.Module`. In the constructor (`__init__`), specify three fully connected layers using `nn.Linear`. The input layer should receive the flattened image of size 28×28, the first hidden layer should produce 512 features, the second hidden layer 256 features, and the output layer should produce 10 outputs (corresponding to the number of classes to predict).
- **Prepare the forward method:** Implement the `forward()` method. First, flatten the input tensor (keeping the batch dimension). Then, apply the first and second layers followed by `ReLU` activations, and finally use the output layer without any activation.

5.2 Training setup

Once the model is defined, set up the components required for training:

- **Prepare the data:** Use a `DataLoader` to create mini-batches from your dataset, with a batch size of 1024. Enable shuffling for the training loader to ensure that batches are sampled differently at each epoch.
- **Define the loss function and optimizer:** Initialize the model and move it to the selected device (CPU or GPU). Use the `CrossEntropyLoss` criterion, which is appropriate for multi-class classification tasks. Choose `Stochastic Gradient Descent (SGD)` as the optimizer with a learning rate of 0.01 and momentum of 0.9.
- **Create a validation function:** Before starting the training process, define a small validation function to evaluate model accuracy on the test set. The function should:
  1. set the model to evaluation mode using `model.eval()`;
  2. disable gradient computation with `torch.no_grad()` to save memory and computation time;
  3. iterate over the dataloader, obtain the model's predictions, and compare them with the proper labels to compute, at the end, the percentage of correctly classified samples.

Info: Use `torch.max(outputs.data, 1)` to obtain the predicted class for each sample.

5.3 Training loop

You will now implement the training loop for the model. This process should be similar to the one used in the linear model, but adapted for the classification task. After training the model for five epochs, use your validation function to compute and print the final test accuracy on the test set.