Most machine learning workflows involve working with

- data,
- creating models,
- optimizing model parameters, and
- saving the trained models.

PyTorch is a popular open-source machine learning library that provides tools and functionalities for building and training neural networks. When working with data in PyTorch, two key components are used:

- `torch.utils.data.DataLoader` and `torch.utils.data.Dataset`. These components are fundamental for efficiently handling and processing datasets during the training and evaluation of machine learning models.

The `Dataset` holds the data and labels, while the `DataLoader` handles the logistics of loading and organizing the data into mini-batches for training and evaluation. This separation of concerns makes it easier to work with large datasets and train machine learning models effectively.

"transform" and "target_transform" are mechanisms used to preprocess and format input data and labels to be suitable for training machine learning models. These transformations ensure that the data is presented in a consistent and standardized manner, enabling the model to learn effectively. They are commonly used when creating custom datasets and data loaders for training in PyTorch and other deep learning frameworks.

A neural network or other machine learning models:

1. **Training**:
    - During the training phase, the model learns from a labeled dataset. It adjusts its internal parameters (weights and biases) to minimize a loss function and make accurate predictions on the training data.
2. **Inference**:
    - After the model is trained, it's used for inference. This involves taking new, unseen data as input and producing predictions or classifications as output.
    - The model applies the learned patterns and relationships to this new data to make its predictions.

Creating a Custom Dataset for your files

A custom Dataset class must implement three functions: **init**, **len**, and **getitem**. Take a look at this implementation; the FashionMNIST images are stored in a directory `img_dir`, and their labels are stored separately in a CSV file `annotations_file`.

- The **init** function is run once when instantiating the Dataset object.
- The **len** function returns the number of samples in our dataset.
- The **getitem** function loads and returns a sample from the dataset at the given index `idx`.

Creating Models

To define a neural network in PyTorch, we create a class that inherits from nn.Module. We define the layers of the network in the `__init__` function and specify how data will pass through the network in the `forward` function. To accelerate operations in the neural network, we move it to the GPU or MPS if available.

Tensors

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.

An ndarray is a multi-dimensional, homogeneous array of fixed-size elements. It can have any number of dimensions (referred to as axes) and is similar in concept to a mathematical array or matrix.

Operations on Tensors

Over 100 tensor operations, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling and more are comprehensively described here.

Each of these operations can be run on the GPU (at typically higher speeds than on a CPU). If you're using Colab, allocate a GPU by going to Runtime > Change runtime type > GPU.

By default, tensors are created on the CPU.

he NumPy API (Application Programming Interface) refers to the set of functions, classes, and modules provided by the NumPy library in Python. It defines the methods and data structures that developers can use to interact with and manipulate numerical data efficiently. The NumPy API provides a wide range of tools for performing mathematical, statistical, and array-based operations on multi-dimensional arrays (ndarrays) and matrices.

Tensor API and its components within TensorFlow and PyTorch:

Tensor Creation and Manipulation:

- Tensors can be created using functions like `tf.constant()` in TensorFlow and `torch.tensor()` in PyTorch. These functions allow you to create tensors from Python lists, NumPy arrays, or other data sources.

Mathematical Operations:

- Tensors support a wide range of mathematical operations, including element-wise arithmetic, matrix operations, and more. For example, you can use functions like `tf.add()`, `tf.matmul()` (TensorFlow) or `torch.add()`, `torch.mm()` (PyTorch) to perform these operations.

Indexing and Slicing:

- Just like NumPy ndarrays, tensors support indexing and slicing to access individual elements, sub-tensors, or slices of data. Indexing and slicing allow you to extract specific parts of a tensor.

Broadcasting:

- Similar to NumPy, both TensorFlow and PyTorch support broadcasting, which allows operations to be performed on tensors of different shapes and dimensions.

Aggregation and Reduction:

- Tensors provide functions for aggregating data, such as calculating the mean, sum, maximum, or minimum along specified dimensions.

Gradient Computation:

- One of the key features of TensorFlow and PyTorch is automatic differentiation, which allows you to compute gradients of tensor operations with respect to certain variables. This is essential for training neural networks using techniques like gradient descent.

GPU and Accelerator Support:

- Both libraries offer support for running tensor operations on GPUs and other hardware accelerators, which greatly speeds up computations in machine learning.

Neural Network Operations:

- TensorFlow and PyTorch are popular for building and training neural networks. They provide extensive APIs for creating layers, defining models, and implementing various neural network architectures.

Joining Tensors

Joining tensors You can use `torch.cat` to concatenate a sequence of tensors along a given dimension. See also torch.stack, another tensor joining operator that is subtly different from `torch.cat`.

Bridge with NumPy

Tensors on the CPU and NumPy arrays can share their underlying memory locations, and changing one will change the other.

- `t: tensor([1., 1., 1., 1., 1.])`:

In this representation, `t` is a tensor created using PyTorch. The tensor contains five elements, each of which has a value of 1.0. The syntax `tensor([...])`

- `n: [1. 1. 1. 1. 1.]`:

In this representation, `n` seems to be a NumPy-like array (or list) containing five elements, each of which is a floating-point number with a value of 1.0. The format `[...]` is typically used in Python to represent a list or array.

- Now that we have a model and data it's time to train, validate and test our model by optimizing its parameters on our data. Training a model is an iterative process; in each iteration the model makes a guess about the output, calculates the error in its guess (*loss*), collects the derivatives of the error with respect to its parameters (as we saw in the previous section), and **optimizes** these parameters using gradient descent

Hyperparameters

Hyperparameters are adjustable parameters that let you control the model optimization process.

- **Number of Epochs** - the number times to iterate over the dataset
- **Batch Size** - the number of data samples propagated through the network before the parameters are updated
- **Learning Rate** - how much to update models parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior

during training.

Inference

"Inference" refers to the process of using a trained machine learning model to make predictions or draw conclusions based on new, unseen data. When a model is trained, it learns patterns and relationships from a training dataset. Inference is what happens when the model applies this learned knowledge to new data that it hasn't seen before.

Loss Function

Common loss functions include nn.MSELoss (Mean Square Error) for regression tasks, and nn.NLLLoss (Negative Log Likelihood) for classification. nn.CrossEntropyLoss combines `nn.LogSoftmax` and `nn.NLLLoss`.

Optimizer

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed (in this example we use Stochastic Gradient Descent). All optimization logic is encapsulated in the `optimizer` object. Here, we use the SGD optimizer; additionally, there are many different optimizers available in PyTorch such as ADAM and RMSProp, that work better for different kinds of models and data.

TRANSFORMS

Data does not always come in its final processed form that is required for training machine learning algorithms. We use **transforms** to perform some manipulation of the data and make it suitable for training.

All TorchVision datasets have two parameters - `transform` to modify the features and `target_transform` to modify the labels - that accept callables containing the transformation logic. The torchvision.transforms module offers several commonly-used transforms out of the box.

Transforms are common image transformations available in the `torchvision.transforms` module. They can be chained together using `Compose`. Most transform classes have a function equivalent: functional transforms give fine-grained control over the transformations. This is useful if you have to build a more complex transformation pipeline (e.g. in the case of segmentation tasks).

Most transformations accept both PIL images and tensor images, although some transformations are PIL-only and some are tensor-only. The Conversion may be used to convert

to and from PIL images, or for converting dtypes and range

print(f"Gradient function for z = {z.grad_fn}")
print(f"Gradient function for loss = {loss.grad_fn}")

weight

In the context of a neural network, a "weight" is a numerical value associated with the connections between neurons. It determines the strength and direction of the signal passed between neurons. When information flows from one neuron to another, it's multiplied by the weight of the connection.

loss function in pytorch

In PyTorch, a loss function (also known as a cost function or objective function) is a crucial component in training machine learning models, particularly neural networks. It quantifies the difference between the predicted values generated by the model and the actual ground truth values. The goal of training a model is to minimize this loss function, which essentially means making the model's predictions as close as possible to the true target values.

in simple word: the loss function as a kind of score that shows how far off the computer's guesses are from the actual truth. The goal of the computer is to minimize this score, meaning it wants to make its guesses as accurate as possible. The computer does this by adjusting its internal settings, like fine-tuning a camera, to improve its guesses

Just like in a game, the computer wants the score to be as low as possible, so it gets better at its job of recognizing things.

AUTOMATIC DIFFERENTIATION WITH `TORCH.AUTOGRAD`

When training neural networks, the most frequently used algorithm is **back propagation**. In this algorithm, parameters (model weights) are adjusted according to the **gradient** of the loss function with respect to the given parameter.

To compute those gradients, PyTorch has a built-in differentiation engine called `torch.autograd`. It supports automatic computation of gradient for any computational graph.