Course – practical deep learning

Fadel Mamar Seydou

MSc. Computational Science and Engineering

Sampled from EPFL CS-433 Machine learning course of 2020 & Little Book of Deep learning by F. Fleuret

Table of contents

Last week’s exercises

Linear regression with PyTorch

Linear

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies an affine linear transformation to the incoming data: $y = xA^T + b$.

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Parameters

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

Getting started with PyTorch

- PyTorch tutorial 60-minute blitz

HuggingFace

What’s huggingface?

“It is a machine learning and data science platform and community that helps users build, deploy and train ML models”

https://huggingface.co/

Loading huggingface datasets

Load a dataset

Before you take the time to download a dataset, it's often helpful to quickly get some general information about a dataset. A dataset's information is stored inside DatasetInfo and can include information such as the dataset description, features, and dataset size.

Use the load_dataset_builder() function to load a dataset builder and inspect a dataset's attributes without committing to downloading it:

```
>>> from datasets import load_dataset_builder
>>> ds_builder = load_dataset_builder("rotten_tomatoes")

# Inspect dataset description
>>> ds_builder.info.description
Movie Review Dataset. This is a dataset of containing 5,331 positive and 5,331 negative processed s

# Inspect dataset features
>>> ds_builder.info.features
{'label': ClassLabel(num_classes=2, names=['neg', 'pos'], id=None),
 'text': Value(dtype='string', id=None)}
```

If you're happy with the dataset, then load it with load_dataset():

https://huggingface.co/docs/datasets/load_hub

Loading Image data

Load image data

Image datasets have Image type columns, which contain PIL objects.

> To work with image datasets, you need to have the `vision` dependency installed. Check out the installation guide to learn how to install it.

When you load an image dataset and call the image column, the images are decoded as PIL Images:

```
>>> from datasets import load_dataset, Image

>>> dataset = load_dataset("beans", split="train")
>>> dataset[0]["image"]
```

> Index into an image dataset using the row index first and then the `image` column - `dataset[0]["image"]` - to avoid decoding and resampling all the image objects in the dataset. Otherwise, this can be a slow and time-consuming process if you have a large dataset.

https://huggingface.co/docs/datasets/image_load

Sharing demos with Gradio

Documentation

4.42.0

Gradio

`gradio`

Build and share machine learning demos and web applications using the core Gradio Python library.

Interface Blocks ChatInterface Textbox Image Audio Dataframe

Python Client

`gradio-client`

Make programmatic requests to Gradio applications from Python environments.

Javascript Client

`@gradio/client`

Make programmatic requests to Gradio applications in JavaScript (TypeScript) from the browser or server-side.

Third Party Clients

Make programmatic requests to Gradio applications using third party clients built by the gradio community.

https://www.gradio.app/docs

Structuring a data science project

Increasing collaboration and Reproducibility

The working environment

- Virtual environment manager
  - Virtualenv
  - Conda
- Specify requirements
  - requirements.txt
  - environment.yml
- ReadMe
  - Write a well written ReadMe.md file which describes the project and gives indications on how to use it.

The configurations

datargs

A paper-thin wrapper around `argparse` that creates type-safe parsers from `dataclass` and `attrs` classes.

Quickstart

Install `datargs`:

```
pip install datargs
```

Create a `dataclass` (or an `attrs` class) describing your command line interface, and call `datargs.parse()` with the class:

```python
# script.py
from dataclasses import dataclass
from pathlib import Path
from datargs import parse

@dataclass  # or @attr.s(auto_attribs=True)
class Args:
    url: str
    output_path: Path
    verbose: bool
    retries: int = 3

def main():
    args = parse(Args)
    print(args)

if __name__ == "__main__":
    main()
```

https://pypi.org/project/datargs/

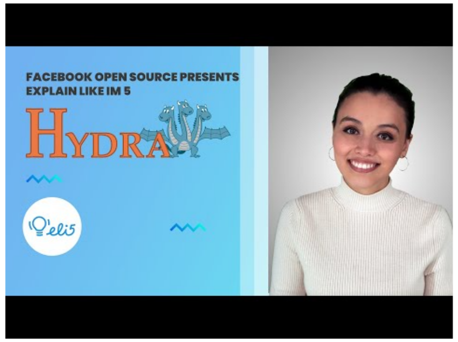
The configurations

HYDRA

pypi v1.3.2 | build invalid | license MIT | python 3.7 | 3.8 | 3.9 | 3.10 | 3.11 | downloads/month 3M | code style black

A framework for elegantly configuring complex applications.

Check the website for more information,
or click the thumbnail below for a one-minute video introduction to Hydra.



https://github.com/facebookresearch/hydra

Cookiecutter Data Science

Cookiecutter Data Science

A logical, reasonably standardized but flexible project structure for doing and sharing data science work.

Cookiecutter Data Science (CCDS) is a tool for setting up a data science project template that incorporates best practices. To learn more about CCDS's philosophy, visit the project homepage.

> ℹ️ Cookiecutter Data Science v2 has changed from v1. It now requires installing the new cookiecutter-data-science Python package, which extends the functionality of the cookiecutter templating utility. Use the provided `ccds` command-line program instead of `cookiecutter`.

Installation

Cookiecutter Data Science v2 requires Python 3.8+. Since this is a cross-project utility application, we recommend installing it with pipx. Installation command options:

```
# With pipx from PyPI (recommended)
pipx install cookiecutter-data-science

# With pip from PyPI
pip install cookiecutter-data-science

# With conda from conda-forge (coming soon)
# conda install cookiecutter-data-science -c conda-forge
```

https://cookiecutter-data-science.drivendata.org/

Training with *Pytorch Lightning*

Interesting libraries to get pretrained models

- TIMM: https://huggingface.co/docs/timm/quickstart
- Monai: https://monai.io/
- Torchvision: https://pytorch.org/vision/stable/models.html
- Segmentation models: https://segmentation-models-pytorch.readthedocs.io/en/latest/

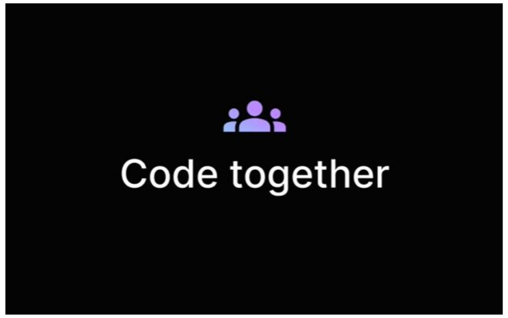
Lightning ecosystem

Lightning AI **Documentation**

Use Lightning to turn ideas into AI - Lightning fast.

Use **AI Studios** to code together, prototype, train, deploy and host AI web apps with zero setup from your browser. Use our open source libraries to develop lightning-fast models.

Get started



Docs by product

AI Studios New!
Code together. Prototype. Train. Deploy. Host AI web apps. From your browser - with zero setup

PyTorch Lightning
Finetune and pretrain AI models on GPUs, TPUs and more. Focus on science, not engineering.
27876

LitServe New!
Easily serve AI models Lightning fast. High-throughput serving engine for AI models.
1792

LitGPT New!
20+ high-performance LLMs with recipes to pretrain, finetune and deploy at scale.
9593

Lightning Fabric
Scale foundation models to 1000s of GPUs with expert-level control.
27876

TorchMetrics
90+ PyTorch metrics, optimized for distributed training
2065

Thunder
Make PyTorch models faster! Thunder is a compiler for PyTorch.
1112

Creating a Dataset

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`.

In the next sections, we'll break down what's happening in each of these functions.

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#creating-a-custom-dataset-for-your-files

Creating a Dataloader

Preparing your data for training with DataLoaders

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in "minibatches", reshuffle the data at every epoch to reduce model overfitting, and use Python's `multiprocessing` to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Iterate through the DataLoader

We have loaded that dataset into the `DataLoader` and can iterate through the dataset as needed. Each iteration below returns a batch of `train_features` and `train_labels` (containing `batch_size=64` features and labels respectively). Because we specified `shuffle=True`, after we iterate over all batches the data is shuffled (for finer-grained control over the data loading order, take a look at Samplers).

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#preparing-your-data-for-training-with-dataloaders

Packing it into a *Datamodule*

```python
class MNISTDataModule(L.LightningDataModule):
    def __init__(self, data_dir: str = "path/to/dir", batch_size: int = 32):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size

    def setup(self, stage: str):
        self.mnist_test = MNIST(self.data_dir, train=False)
        self.mnist_predict = MNIST(self.data_dir, train=False)
        mnist_full = MNIST(self.data_dir, train=True)
        self.mnist_train, self.mnist_val = random_split(
            mnist_full, [55000, 5000], generator=torch.Generator().manual_seed(42)
        )

    def train_dataloader(self):
        return DataLoader(self.mnist_train, batch_size=self.batch_size)

    def val_dataloader(self):
        return DataLoader(self.mnist_val, batch_size=self.batch_size)

    def test_dataloader(self):
        return DataLoader(self.mnist_test, batch_size=self.batch_size)

    def predict_dataloader(self):
        return DataLoader(self.mnist_predict, batch_size=self.batch_size)

    def teardown(self, stage: str):
        # Used to clean-up when the run is finished
        ...
```

https://lightning.ai/docs/pytorch/stable/data/datamodule.html

Define a lightning module

```python
import os
from torch import optim, nn, utils, Tensor
from torchvision.datasets import MNIST
from torchvision.transforms import ToTensor
import lightning as L

# define any number of nn.Modules (or use your current ones)
encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))


# define the LightningModule
class LitAutoEncoder(L.LightningModule):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        # it is independent of forward
        x, _ = batch
        x = x.view(x.size(0), -1)
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = nn.functional.mse_loss(x_hat, x)
        # Logging to TensorBoard (if installed) by default
        self.log("train_loss", loss)
        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=1e-3)
        return optimizer


# init the autoencoder
autoencoder = LitAutoEncoder(encoder, decoder)
```

https://lightning.ai/docs/pytorch/stable/model/train_model_basic.html

Trainer

4: Train the model

The Lightning Trainer "mixes" any LightningModule with any dataset and abstracts away all the engineering complexity needed for scale.

```
# train the model (hint: here are some helpful Trainer arguments for rapid idea iteration)
trainer = L.Trainer(limit_train_batches=100, max_epochs=1)
trainer.fit(model=autoencoder, train_dataloaders=train_loader)
```

The Lightning Trainer automates 40+ tricks including:

- Epoch and batch iteration
- `optimizer.step()`, `loss.backward()`, `optimizer.zero_grad()` calls
- Calling of `model.eval()`, enabling/disabling grads during evaluation
- Checkpoint Saving and Loading
- Tensorboard (see loggers options)
- Multi-GPU support
- TPU
- 16-bit precision AMP support

- https://lightning.ai/docs/pytorch/stable/starter/introduction.html
- https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.trainer.Trainer.html#lightning.pytorch.trainer.trainer.Trainer

TorchMetrics

TorchMetrics is a collection of 100+ PyTorch metrics implementations and an easy-to-use API to create custom metrics.

It offers:

- A standardized interface to increase reproducibility
- Reduces Boilerplate
- Distributed-training compatible
- Rigorously tested
- Automatic accumulation over batches
- Automatic synchronization between multiple devices

You can use TorchMetrics in any PyTorch model, or within PyTorch Lightning to enjoy additional features:

- This means that your data will always be placed on the same device as your metrics.
- Native support for logging metrics in Lightning to reduce even more boilerplate.

https://lightning.ai/docs/torchmetrics/stable/pages/quickstart.html

Logging

LOGGERS

`logger`	Abstract base class used to build new loggers.
`comet`	Comet Logger
`csv_logs`	CSV logger
`mlflow`	MLflow Logger
`neptune`	Neptune Logger
`tensorboard`	TensorBoard Logger
`wandb`	Weights and Biases Logger

https://lightning.ai/docs/pytorch/stable/api_references.html#loggers

Logging

```
pip install wandb
```

Create a *WandbLogger* instance:

```
from lightning.pytorch.loggers import WandbLogger

wandb_logger = WandbLogger(project="MNIST")
```

Pass the logger instance to the *Trainer*:

```
trainer = Trainer(logger=wandb_logger)
```

Log metrics

Log from `LightningModule`:

```
class LitModule(LightningModule):
    def training_step(self, batch, batch_idx):
        self.log("train/loss", loss)
```

Use directly wandb module:

```
wandb.log({"train/loss": loss})
```

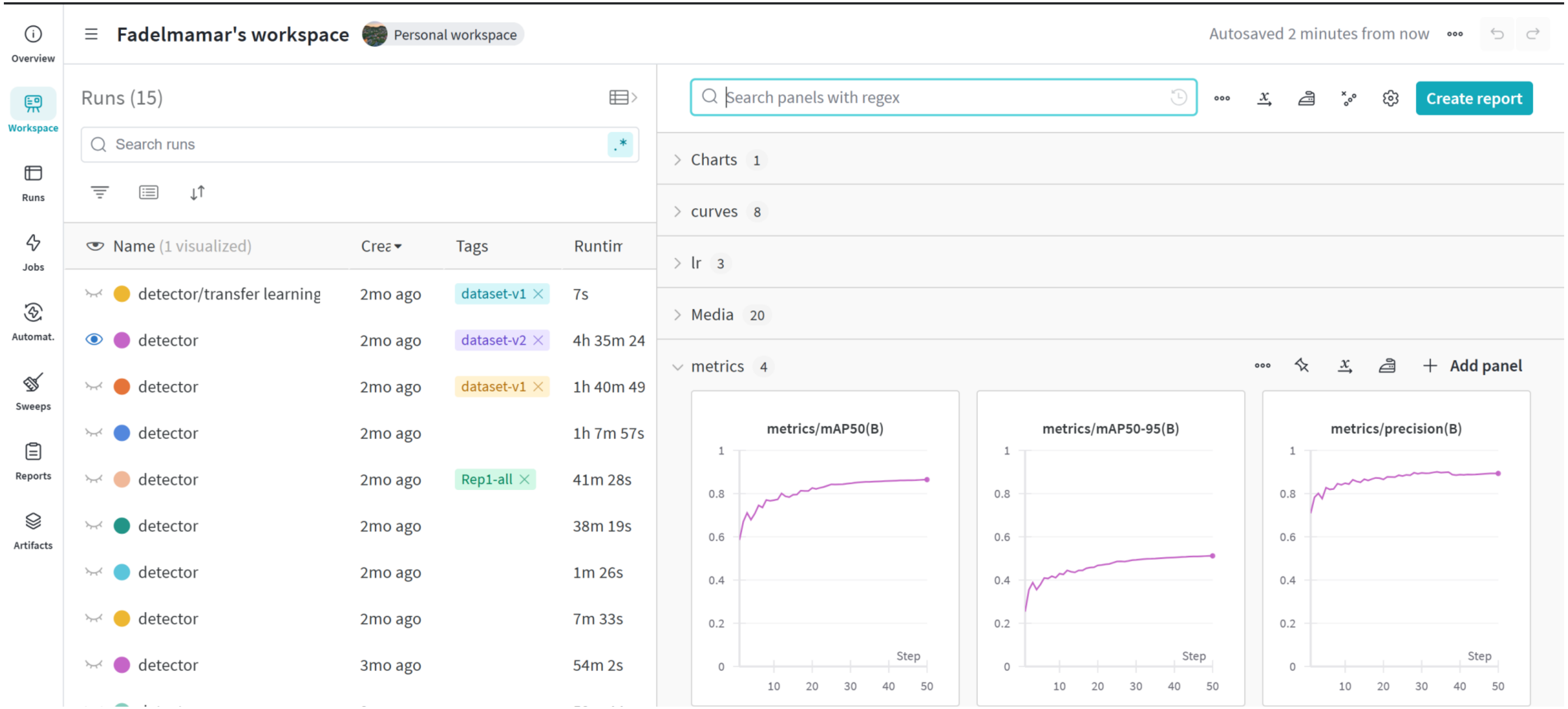
Log hyper-parameters

Save `LightningModule` parameters:

```
class LitModule(LightningModule):
    def __init__(self, *args, **kwarg):
        self.save_hyperparameters()
```

https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.loggers.wandb.html#module-lightning.pytorch.loggers.wandb

Logging: wandb



Early stopping

```
>>> from lightning.pytorch import Trainer
>>> from lightning.pytorch.callbacks import EarlyStopping
>>> early_stopping = EarlyStopping('val_loss')
>>> trainer = Trainer(callbacks=[early_stopping])
```

https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.callbacks.EarlyStopping.html#lightning.pytorch.callbacks.EarlyStopping

Model checkpointing

```
# save any arbitrary metrics like `val_loss`, etc. in name
# saves a file like: my/path/epoch=2-val_loss=0.02-other_metric=0.03.ckpt
>>> checkpoint_callback = ModelCheckpoint(
...     dirpath='my/path',
...     filename='{epoch}-{val_loss:.2f}-{other_metric:.2f}'
... )
```

https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.callbacks.ModelCheckpoint.html#lightning.pytorch.callbacks.ModelCheckpoint

Exporting model for faster inference

ONNX, Torchscript, Quantization and Pruning

ONNX

ONNX Tutorials

Open Neural Network Exchange (ONNX) is an open standard format for representing machine learning models. ONNX is supported by a community of partners who have implemented it in many frameworks and tools.

Getting ONNX models

- Pre-trained models (validated): Many pre-trained ONNX models are provided for common scenarios in the ONNX Model Zoo
- Pre-trained models (non-validated): Many pre-trained ONNX models are provided for common scenarios in the ONNX Model Zoo.
- Services: Customized ONNX models are generated for your data by cloud based services (see below)
- Convert models from various frameworks (see below)

https://github.com/onnx/tutorials

Torchscript

TorchScript is a way to create serializable and optimizable models from PyTorch code. Any TorchScript program can be saved from a Python process and loaded in a process where there is no Python dependency.

We provide tools to incrementally transition a model from a pure Python program to a TorchScript program that can be run independently from Python, such as in a standalone C++ program. This makes it possible to train models in PyTorch using familiar tools in Python and then export the model via TorchScript to a production environment where Python programs may be disadvantageous for performance and multi-threading reasons.

For a gentle introduction to TorchScript, see the Introduction to TorchScript tutorial.

For an end-to-end example of converting a PyTorch model to TorchScript and running it in C++, see the Loading a PyTorch Model in C++ tutorial.

https://pytorch.org/docs/stable/jit.html

Quantization

Quantization refers to techniques for performing computations and storing tensors at lower bitwidths than floating point precision. A quantized model executes some or all of the operations on tensors with reduced precision rather than full precision (floating point) values. This allows for a more compact model representation and the use of high performance vectorized operations on many hardware platforms. PyTorch supports INT8 quantization compared to typical FP32 models allowing for a 4x reduction in the model size and a 4x reduction in memory bandwidth requirements. Hardware support for INT8 computations is typically 2 to 4 times faster compared to FP32 compute. Quantization is primarily a technique to speed up inference and only the forward pass is supported for quantized operators.

PyTorch supports multiple approaches to quantizing a deep learning model. In most cases the model is trained in FP32 and then the model is converted to INT8. In addition, PyTorch also supports quantization aware training, which models quantization errors in both the forward and backward passes using fake-quantization modules. Note that the entire computation is carried out in floating point. At the end of quantization aware training, PyTorch provides conversion functions to convert the trained model into lower precision.

At lower level, PyTorch provides a way to represent quantized tensors and perform operations with them. They can be used to directly construct models that perform all or part of the computation in lower precision. Higher-level APIs are provided that incorporate typical workflows of converting FP32 model to lower precision with minimal accuracy loss.

https://pytorch.org/docs/stable/quantization.html