CMSC 491/691: Computer Vision **Spring 2024**

Lecture : Pytorch Tutorial

Lecturer: Tejas Gokhale *Scribe: Sadia Nasrin Tisha*

0.1 Introduction

PyTorch is an open source machine learning library for Python and is completely based on Torch. It is primarily used for applications such as natural language processing. PyTorch is developed by Facebook's artificial-intelligence research group along with Uber's "Pyro" software for the concept of in-built probabilistic programming. This tutorial will introduce you to basic PyTorch concepts and operations, setting up a simple neural network, and training it on a dataset. Let's dive in.

0.2 Prerequisites

- Basic understanding of Python programming
- Familiarity with fundamental machine learning concepts

0.3 Installation

0.3.1 Conda Installation

Conda is a popular, open-source package and environment management system widely used in the scientific and data science communities to manage packages, dependencies, and environments for various languages like Python, R, Ruby, Lua, Scala, Java, JavaScript, C/C++, FORTRAN, and more. It is designed to simplify package management and deployment, making it easier to manage multiple data science projects with varying requirements. Conda can be installed as part of the Anaconda or Miniconda distribution. Anaconda includes Conda and a suite of other tools and libraries, while Miniconda includes only Conda and its dependencies, allowing users to install the packages they need.

0.3.1.1 Anaconda Installation:

- **Download Anaconda:** Visit the Anaconda website and download the installer for your operating system.
- **Run the Installer:** Execute the downloaded installer and follow the on-screen instructions.
- **Verify Installation:** Open a terminal (or command prompt on Windows) and type "conda list" to see the installed packages.

0.3.1.2 Miniconda Installation:

- **Download Miniconda:** Visit the Miniconda website and download the installer for your operating system.
- **Run the Installer:** Execute the downloaded installer and follow the on-screen instructions. Ensure you add Conda to your PATH during installation or initialize it manually.
- **Verify Installation:** Open a terminal (or command prompt) and type "conda list" to check the installed packages.

0.3.2 PyTorch Installation

You can download Pytorch via `pip` or `conda`, depending on your environment. Visit the PyTorch official website and select your preferences (OS, package manager, Python version, CUDA version) to get the appropriate installation command.

For example, to install PyTorch with `pip` without CUDA support:

```
pip install torch torchvision torchaudio
```

0.4 Tensor Basics

Tensors are the core component in PyTorch, similar to NumPy arrays but with GPU support.

0.4.1 Creating Tensors

```
import torch
import numpy as np

data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)

print(x_data.shape)
print(type(x_data))

torch.Size([2, 2])
<class 'torch.Tensor'>
```

0.4.2 From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see Bridge with NumPy).

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(x_np)
print(type(x_np))
```

Output:

```
tensor([[1, 2],
        [3, 4]])
<class 'torch.Tensor'>
```

0.4.3 From another tensor

The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden.

```
x_ones = torch.zeros_like(x_data, dtype=torch.float)
# retains the properties of x_data

print(f‘‘Ones Tensor: \n {x_ones} \n")

x_rand=torch.rand_like(x_data,dtype=torch.float)
#overrides the datatype of x_data

print(f‘‘Random Tensor: \n {x_rand} \n")
```

Output:

```
Ones Tensor:
 tensor([[0., 0.],
        [0., 0.]])

Random Tensor:
 tensor([[0.6087, 0.8058],
        [0.0065, 0.1806]])
```

0.4.4 Attributes of a Tensor

Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3,4)
print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Output:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

0.4.5 Operations on Tensors

By default, tensors are created on the CPU. We need to move tensors to the GPU using explicitly `.to` method(after checking for GPU availability). Remember that copying large tensors across devices can be expensive regarding time and memory!

```
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
    print ("Got GPU")
```

Output:

```
Got GPU
```

0.4.6 PyTorch Indexing and Slicing

In PyTorch, tensors are multi-dimensional arrays analogous to numpy arrays. This tutorial covers the essentials of indexing and slicing operations in PyTorch, illustrated with a simple tensor example.

0.4.6.1 Creating a Tensor

First, we create a 4x4 tensor with random values:

```
import torch

tensor = torch.rand(4, 4)
print(tensor)
```

Output:

```
tensor([[0.1426, 0.7897, 0.5363, 0.0358],
        [0.9354, 0.2958, 0.8098, 0.7668],
        [0.6238, 0.3185, 0.8677, 0.3414],
        [0.4549, 0.0550, 0.6263, 0.0781]])
```

The `torch.rand` function generates a tensor filled with random values between 0 and 1.

0.4.6.2 Indexing a Tensor

PyTorch supports standard Python indexing and slicing. Here's how to access specific elements or slices of the tensor:

0.4.6.3 Accessing the First Row

```
print(f"First row: {tensor[0]}")
```

Output:

```
First row: tensor([0.1426, 0.7897, 0.5363, 0.0358])
```

This command selects the first row of the tensor.

0.4.6.4 Accessing the First Column

```
print(f"First column: {tensor[:, 0]}")
```

Output:

```
First column: tensor([0.1426, 0.9354, 0.6238, 0.4549])
```

To select the first column, we use the `:` symbol to specify all rows and `0` for the first column.

0.4.6.5 Accessing the Last Column

There are two ways to access the last column:

```
print(f"Last column: {tensor[..., -1]}")
# or equivalently
print(f"Last column: {tensor[:, -1]}")
```

Output:

```
Last column: tensor([0.0358, 0.7668, 0.3414, 0.0781])
```

Both methods retrieve the last column of the tensor.

0.4.6.6 Slicing a Tensor

You can modify slices of a tensor as follows:

```
tensor[:, 1::2] = 0
print(tensor)
```

Output:

```
tensor([[0.1426, 0.0000, 0.5363, 0.0000],
        [0.9354, 0.0000, 0.8098, 0.0000],
        [0.6238, 0.0000, 0.8677, 0.0000],
        [0.4549, 0.0000, 0.6263, 0.0000]])
```

This code sets elements in the second and fourth columns to 0 by slicing the tensor with `[:, 1::2]`.

0.4.7 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 subtly different from `torch.cat`.

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1, t1.shape)
print(tensor)
```

Output:

```
tensor([[0.5211, 0.3680, 0.3529, 0.0719, 0.5211, 0.3680, 0.3529, 0.0719, 0.5211,
         0.3680, 0.3529, 0.0719],
        [0.0061, 0.9109, 0.6086, 0.8240, 0.0061, 0.9109, 0.6086, 0.8240, 0.0061,
         0.9109, 0.6086, 0.8240],
        [0.6420, 0.6554, 0.0579, 0.1223, 0.6420, 0.6554, 0.0579, 0.1223, 0.6420,
         0.6554, 0.0579, 0.1223],
        [0.9969, 0.6384, 0.5178, 0.8625, 0.9969, 0.6384, 0.5178, 0.8625, 0.9969,
         0.6384, 0.5178, 0.8625]]) torch.Size([4, 12])
```

```
t1 = torch.stack([tensor, tensor, tensor], dim=2)
print(t1, t1.shape)
print(tensor)
```

Output:

```
tensor([[[0.1426, 0.1426, 0.1426],
         [0.0000, 0.0000, 0.0000],
         [0.5363, 0.5363, 0.5363],
         [0.0000, 0.0000, 0.0000]],

        [[0.9354, 0.9354, 0.9354],
         [0.0000, 0.0000, 0.0000],
         [0.8098, 0.8098, 0.8098],
         [0.0000, 0.0000, 0.0000]],

        [[0.6238, 0.6238, 0.6238],
         [0.0000, 0.0000, 0.0000],
         [0.8677, 0.8677, 0.8677],
         [0.0000, 0.0000, 0.0000]],

        [[0.4549, 0.4549, 0.4549],
         [0.0000, 0.0000, 0.0000],
         [0.6263, 0.6263, 0.6263],
         [0.0000, 0.0000, 0.0000]]]) torch.Size([4, 4, 3])
```

0.4.8 PyTorch Arithmetic Operations: Matrix Multiplication

In this PyTorch tutorial, we delve into the world of arithmetic operations on tensors, focusing on matrix multiplication and element-wise multiplication. Tensors are the backbone of PyTorch, and understanding

how to manipulate them is crucial for developing deep learning models. We start by exploring the matrix multiplication operation, using 'matmul' and the '@' operator to compute the product of a tensor with its transpose, showcasing three different methods to achieve the same result for enhanced understanding. Then, we transition to element-wise multiplication, which multiplies corresponding elements of two tensors, again demonstrating multiple ways to perform this operation for equivalent outcomes.

To multiply a tensor by its transpose:

```
result = t1.matmul(t1.T)
print("Matrix multiplication of t1 and its transpose:\n", result)
```

Output:

```
tensor([[1.6100, 1.8372, 1.8148, 2.9975],
        [1.8372, 5.6376, 2.2108, 4.8406],
        [1.8148, 2.2108, 2.5800, 3.5816],
        [2.9975, 4.8406, 3.5816, 7.2405]])
```

```
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)

# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

Output:

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

0.4.9 Single-element tensors

If you have a one-element tensor, for example by aggregating all values of a tensor into one value, you can convert it to a Python numerical value using `item()`:

```
agg = tensor.sum()
print (agg, agg.shape)
agg_item = agg.item()
print(agg_item, type(agg_item))
```

Output:

```
tensor(6.9210) torch.Size([])
6.920967102050781 <class 'float'>
```

0.4.10 In-place operations

Operations that store the result into the operand are called in-place. They are denoted by $a\_$ suffix. For example: $x.copy\_(y), x.t\_()$, will change x.

```
print(f"{tensor} \n")
tensor.add_(5)
print(tensor)
print(tensor+5)
```

Output:

```
tensor([[0.1975, 0.4421, 0.9542, 0.2363],
        [0.4391, 0.4525, 0.3842, 0.0386],
        [0.4936, 0.1211, 0.1713, 0.1251],
        [0.9990, 0.8259, 0.4485, 0.5920]])

tensor([[5.1975, 5.4421, 5.9542, 5.2363],
        [5.4391, 5.4525, 5.3842, 5.0386],
        [5.4936, 5.1211, 5.1713, 5.1251],
        [5.9990, 5.8259, 5.4485, 5.5920]])
tensor([[10.1975, 10.4421, 10.9542, 10.2363],
        [10.4391, 10.4525, 10.3842, 10.0386],
        [10.4936, 10.1211, 10.1713, 10.1251],
        [10.9990, 10.8259, 10.4485, 10.5920]])
```

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

0.5 Working with Data

0.5.1 The FashionMNIST Dataset

The FashionMNIST Dataset is a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747. 28×28 grayscale images 70,000 fashion products 10 categories 7,000 images per category The training set has 60,000 images, and the test set has 10,000 images.

Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Figure 0.1: Fashion MNIST Dataset with labels, descriptions and examples

0.5.2 Loading the Dataset

PyTorch provides a handy utility called `torchvision` for loading datasets. Here's how you can load the FashionMNIST dataset:

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
```

```
    download=True,
    transform=ToTensor()
)
```

0.5.3 Exploring the Dataset

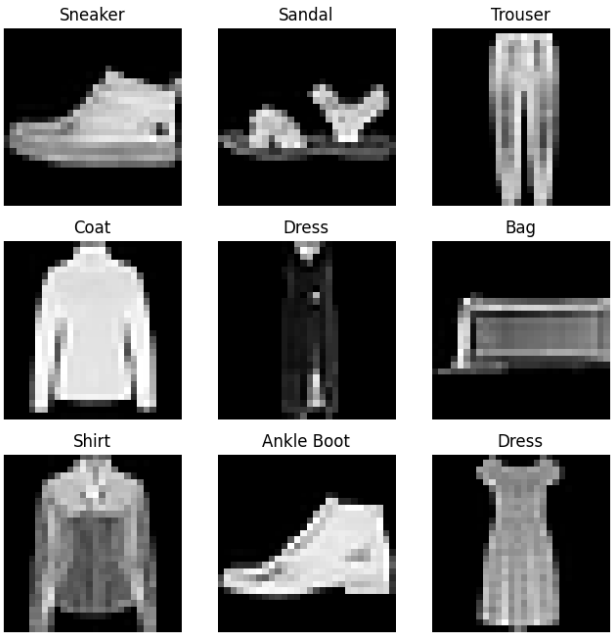
Let's visualize some of the training images to understand what we're working with. Here we can index Datasets manually like a `list: training_data[index]`. We use matplotlib to visualize some samples in our training data.

```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]

    print("img.shape", img.shape)
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

Output:

```
img.shape torch.Size([1, 28, 28])
```



0.5.4 Creating a Custom Dataset for your files

In the realm of machine learning, particularly when dealing with specialized or proprietary datasets, the necessity to tailor data loading mechanisms to specific formats becomes paramount. PyTorch's custom dataset functionality caters to this need, allowing for the creation of bespoke dataset classes that integrate seamlessly with its broader ecosystem. The `‘CustomImageDataset‘` class exemplifies this, leveraging pandas for annotation management and `‘torchvision‘` for image processing. By defining key operational methods—initialization for setup, length for determining dataset size, and item retrieval for accessing individual data points—this class provides a robust framework for ingesting and manipulating data. Such customization empowers developers and researchers to adapt their models to a wide array of unique data scenarios, enhancing the versatility and applicability of machine learning solutions.

```
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
```

0.5.5 Preparing Data for Training

To prepare the data for training, we will use the ‘`DataLoader`‘ utility to batch, shuffle, and distribute the process across multiple cores.

```
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)
```

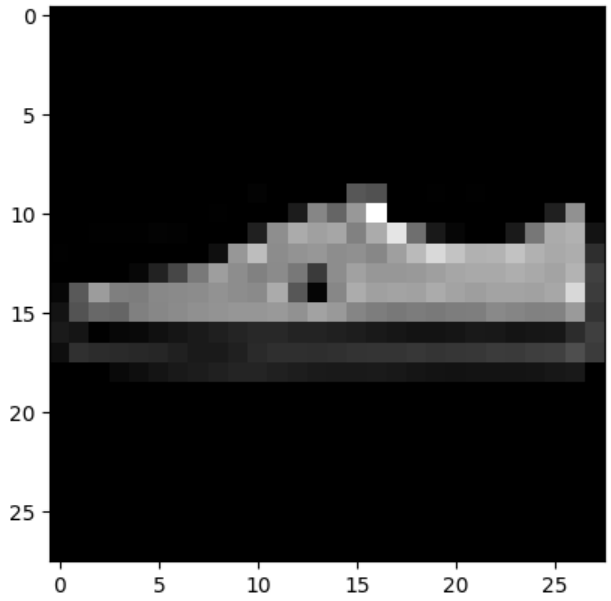
0.5.6 Iterate through the DataLoader

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}")
```

Output:

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
```



0.5.7 Transforms

The FashionMNIST features are in `PIL` Image format, and the labels are integers. For training, we need the features as normalized tensors, and the labels as one-hot encoded tensors. To make these transformations, we use `ToTensor` and `Lambda`.

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros
    (10, dtype=torch.float).scatter_(0, torch.tensor(y),
    value=1))
)
```

0.6 Building a Simple Neural Network

0.6.1 Initialization

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

```
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

0.6.2 Get Device for Training

Leveraging the optimal computing resources for training deep learning models significantly enhances performance and efficiency. In PyTorch, this entails selecting the most suitable hardware accelerator available—be it a GPU with CUDA, Apple's Metal Performance Shaders (MPS) for leveraging Apple Silicon, or defaulting back to the CPU. The snippet provided smartly detects and selects the best available option for your training environment. By querying `torch.cuda.is_available()` and `torch.backends.mps.is_available()`, it ensures your model utilizes the highest-performing hardware at hand, whether you're working on NVIDIA GPUs, Apple Silicon, or in environments where only CPU is available. This automatic detection and selection process is crucial for developing scalable, hardware-agnostic models that can be deployed across various platforms without requiring manual configuration, thus streamlining the development and deployment process.

```
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
```

0.6.3 Defining the Network Structure

Now, let's define a simple neural network. Our network will consist of a sequence of layers: a flattening layer followed by two linear layers with `ReLU` activation and a final linear layer for the output.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(1*28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

Output:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

To use the model, we pass it the input data. This executes the model's forward, along with some background operations. Do not call `model.forward()` directly!

Calling the model on the input returns a 2-dimensional tensor with dim=0 corresponding to each output of 10 raw predicted values for each class, and dim=1 corresponding to the individual values of each output. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```
X = torch.rand(1, 1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
print(logits)
y_pred = pred_probab.argmax(1)
print(pred_probab)
print(pred_probab.sum())
print(f"Predicted class: {y_pred}")
```

Output:

```
tensor([[-0.0618, -0.0455, -0.0247, -0.0107,  0.0277,  0.0921,  0.0142, -0.0620,
          0.0164,  0.0599]], grad_fn=<AddmmBackward0>)
tensor([[0.0938, 0.0954, 0.0974, 0.0988, 0.1026, 0.1095, 0.1013, 0.0938, 0.1015,
         0.1060]], grad_fn=<SoftmaxBackward0>)
tensor(1.0000, grad_fn=<SumBackward0>)
Predicted class: tensor([5])
```

0.7 Autograd

PyTorch's automatic differentiation engine, Autograd, helps with the computation of gradients—crucial for backpropagation. It consider the simplest one-layer neural network, with input x, parameters w and b, and some loss function. Tensors have an attribute `requires_grad` that tracks all operations on them for gradient computation. It can be defined in PyTorch in the following manner:

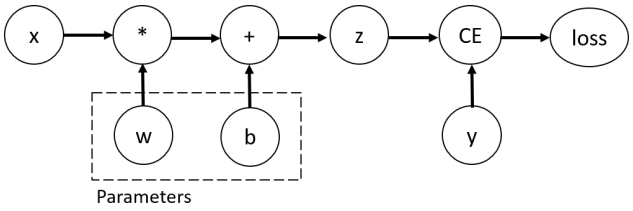


Figure 0.2: Tensors, Functions and Computational graph. In this network, w and b are parameters, which we need to optimize. Thus, we need to be able to compute the gradients of loss function with respect to those variables. In order to do that, we set the requires$_g radpropertyofthosetensors$.

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

0.8 Computing Gradients

In the realm of neural network training with PyTorch, the manipulation and understanding of gradients play a crucial role in optimizing model parameters. The .backward() method is pivotal for computing gradients, allowing for the adjustment of weights (w) and biases (b) through gradient descent by accessing their gradients via w.grad and b.grad. However, there are scenarios where gradient tracking is unnecessary or computationally expensive, especially during inference. PyTorch provides mechanisms to temporarily disable gradient tracking, enhancing computational efficiency. Using torch.no_grad() suspends the recording of gradients for operations performed within its context, rendering operations like torch.matmul(x, w)+b gradient-free. Alternatively, the detach() method offers a more granular approach by creating a new tensor that does not require gradients, as seen in z_det = z.detach(), effectively achieving the same computational benefit. These features are instrumental in reducing memory usage and computational overhead during the forward pass when updating model parameters is not needed.

```
loss.backward()
print(w.grad)
print(b.grad)
```

Output:

```
tensor([[0.3128, 0.0550, 0.2104],
        [0.3128, 0.0550, 0.2104],
        [0.3128, 0.0550, 0.2104],
        [0.3128, 0.0550, 0.2104],
        [0.3128, 0.0550, 0.2104]])
tensor([0.3128, 0.0550, 0.2104])
```

0.8.1 Disabling Gradient Tracking

```
z = torch.matmul(x, w)+b


with torch.no_grad():
    z = torch.matmul(x, w)+b
```

Another way to achieve the same result is to use the `detach()` method on the tensor:

```
z = torch.matmul(x, w)+b
z_det = z.detach()
```

0.9 Training the Model

0.9.1 Optimization

0.9.1.1 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.

We pass our model's output logits to nn.CrossEntropyLoss, which will normalize the logits and compute the prediction error.

```
loss_fn = nn.CrossEntropyLoss()
```

0.9.1.2 Optimizer

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

```
learning_rate = 1e-3
batch_size = 64
epochs = 5

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Inside the training loop, optimization happens in three steps:

- Call `optimizer.zero_grad()` to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to `loss.backward()`. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call `optimizer.step()` to adjust the parameters by the gradients collected in the backward pass.

0.9.2 Training and Testing Loops

0.9.2.1 Training Loop

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)

    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

0.9.2.2 Testing Loop

```
def test_loop(dataloader, model, loss_fn):
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

0.10 Implementation

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
```

```
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

0.11 Save and Load the Model

In PyTorch, efficiently saving and loading model weights and architectures is crucial for both the continuation of training and the deployment of trained models. PyTorch offers a straightforward and flexible approach for this process through its saving and loading mechanisms. Below is a detailed explanation organized around key practices for saving and loading both model weights and entire model architectures.

Saving and Loading Model Weights

Saving Model Weights

PyTorch models encapsulate the learned parameters in an internal state dictionary, known as `state_dict`. This dictionary can be easily persisted and retrieved using PyTorch's `torch.save` method. Here's how you can save the `state_dict` of a pretrained VGG16 model:

```
import torch
from torchvision import models

# Instantiate and load a pretrained VGG16 model
model = models.vgg16(weights='IMAGENET1K_V1')

# Save the model's state_dict
torch.save(model.state_dict(), 'model_weights.pth')
```

This saves the learned parameters (weights and biases) of the model to a file named `'model_weights.pth'`.

Loading Model Weights

To load the saved model weights, you must first instantiate an instance of the model's class, ensuring it has the same architecture as the saved model. You can then load the saved `state_dict` into this model instance:

```
# Create an instance of the same model
model = models.vgg16()  # Note: We do not specify ‘weights‘ here

# Load the saved state_dict
model.load_state_dict(torch.load('model_weights.pth'))

# Set the model to evaluation mode
model.eval()
```

It's crucial to invoke `model.eval()` after loading the weights to set the model in evaluation mode. This is especially important for models with layers that behave differently during training and inference, such as dropout and batch normalization layers.

Saving and Loading Entire Models

While saving and loading only the model weights is often sufficient, there are scenarios where you might want to save the entire model, including its architecture. This can be particularly useful for sharing models or deploying them without requiring the model class definition to be available.

Saving the Entire Model

To save a model along with its architecture, you can pass the entire model object to `torch.save`:

```
# Save the entire model
torch.save(model, 'model.pth')
```

This method serializes the whole model using Python's `pickle` module, saving the model architecture along with its weights and biases.

Loading the Entire Model

Loading a saved model along with its architecture is straightforward, as you don't need to instantiate the model class beforehand:

```
# Load the entire model
model = torch.load('model.pth')

# Set the model to evaluation mode
model.eval()
```

When loading a complete model, ensure the code that defines the model class is accessible in the environment where the model is loaded, unless the model was saved using a method like `torch.jit`, which can save models in a more portable format.

The tutorial covered the basics of PyTorch. PyTorch offers much more, including advanced neural network layers and models, complex training techniques, and working with large datasets. To dive deeper into PyTorch, explore the official documentation and tutorials.

The best way to learn is by doing. To improve your skills, try implementing different types of neural networks, such as convolutional neural networks for image processing, and work on real datasets.