In the Name of God

**PyTorch User Manual**

Produced by:

Razieh Dorehgard

Bita Nasiri

Pooria Tajmehrabi

Table of Contents:

Chapter 1: Core Concepts, Datasets, and Transforms

Chapter 2: Model Architecture and Fine-Tuning

Chapter 3: Training and Evaluating Models

* Why to use PyTorch for learning
* Tensors in PyTorch
* AutoGrad in PyTorch
* Differentiation in AutoGrad
* Getting started with neural networks
* What are neural networks exactly
* How does PyTorch implement neural networks?
* Recap
* So far we covered
* What is a loss function
* Loss function in PyTorch
* Training a classifier
* Convolutional Neural Network

Chapter 3

**Training and Evaluating Models**

**Why to Use PyTorch for Learning**

To embark on explaining why you should definitely consider PyTorch in your machine learning and deep learning projects, we should state: “Because PyTorch is extremely easy to use!”. Owing to having an appealing interface and complete documents, it is quite easy to find the solution to your potential problems. Moreover, Just like [NumPy](https://numpy.org/) provides multidimensional arrays, PyTorch offers tensors. These are generalizations of matrices to N-dimensional space and serve as the basic building blocks of many Deep learning algorithms. However, unlike NumPy arrays, PyTorch tensors can be used on GPUs for accelerated computing. Additionally, in deep learning, we often need to figure out how much to adjust things (called gradients). PyTorch has a tool called [Autograd](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html) that does this for us automatically. Plus, with PyTorch’s dynamic approach, you can make changes as you work, which is great for some models and research situations.

Convinced? Great, let’s get started then.

**Tensors in PyTorch**

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 specialized hardware to accelerate computing. If you’re familiar with ndarrays, you’ll be right at home with the Tensor API. If not, follow along in this quick API walkthrough.

**Tensor Initialization Ways:**

1. Directly from data

x\_data = torch.tensor(data)

1. From a NumPy array

data = [[1, 2], [3, 4]]

np\_array = np.array(data)

x\_np = torch.from\_numpy(np\_array)

1. From another tensor

x\_ones = torch.ones\_like(x\_data)

print(f"Ones Tensor: \n {x\_ones} \n")

x\_rand = torch.rand\_like(x\_data, dtype=torch.float)

print(f"Random Tensor: \n {x\_rand} \n")

After the initialization of tensors, you can access their attributes.

**Tensor Attributes:**

Tensor attributes describe their shape, datatype, and the device on which they are stored. Some popular attributes of tensors are:

1. Shape
2. Device
3. Data type

Sample code:

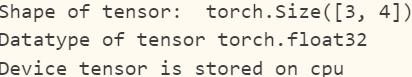
tensor = torch.rand(3, 4)

print("Shape of tensor: ", tensor.shape)

print("Datatype of tensor", tensor.dtype)

print("Device tensor is stored on", tensor.device)

Output should be:



**Tensor Operations**

Over 100 tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random sampling, and more are comprehensively described [here](https://pytorch.org/docs/stable/torch.html). Each of them can be run on the GPU, at typically higher speeds than on a CPU.

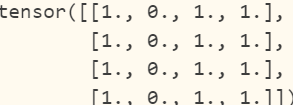
Popular tensor operations are:

1. **Indexing & Slicing**

tensor = torch.ones(4, 4)

tensor[:,1] = 0

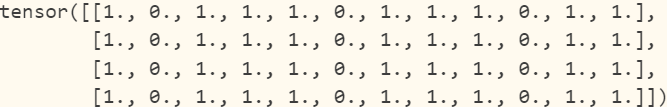
print(tensor)

****

1. **Joining**

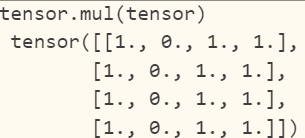
t1 = torch.cat([tensor, tensor, tensor], dim=1)

print(t1)

****

1. **Multiplying**

print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")

****

**AutoGrad in PyTorch**

PyTorch's autograd is a tool that helps train neural networks. Here's how it works:

* **Neural Networks (NNs):** These are made up of layers of functions that process input data. The functions use parameters (weights and biases) stored in tensors.
* **Training a Neural Network:**
  + **Forward Propagation:** The network guesses the output by running the input data through its layers.

prediction = model(data) # forward pass

* + **Backward Propagation:** The network adjusts its parameters based on the error in its guess. It works backward from the output, calculates the gradients (how much each parameter contributed to the error), and updates the parameters using gradient descent.

loss = (prediction - labels).sum()

loss.backward() # backward pass

Next, we load an optimizer, in this case SGD with a learning rate of 0.01 and [momentum](https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d) of 0.9. We register all the parameters of the model in the optimizer.

optim = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)

Finally, we call .step() to initiate gradient descent. The optimizer adjusts each parameter by its gradient stored in .grad.

optim.step() #gradient descent

At this point, you have everything you need to train your neural network.

**Differentiation in AutoGrad**

Let’s take a look at how autograd collects gradients. We create two tensors a and b with requires\_grad=True. This signals to autograd that every operation on them should be tracked.

[See the Autograd Source Example](https://github.com/rose-giant/PyTorch-User-Mannual/blob/main/Traning%20and%20Evaluation/Autograd.ipynb)

**Getting Started with Neural Networks**

After learning about tensors and ways to differentiate computational graphs using autoGrad, you are almost prepared to train neural networks using PyTorch.

**What are Neural Networks exactly?**

trainNeural networks are a subset of machine learning, inspired by the structure and function of the human brain. They consist of layers of nodes, or "neurons," that are interconnected. Here's a simplified breakdown:

1. **Input Layer**: This is where data enters the network. Each node in this layer represents a feature of the data.
2. **Hidden Layers**: These layers lie between the input and output layers. They process the input data by performing a series of transformations. The number of hidden layers can vary, leading to "deep" neural networks.
3. **Output Layer**: This layer produces the final result or prediction. Each node in this layer corresponds to a potential outcome or category.

The magic happens through the connections between neurons. Each connection has a "weight" that adjusts during the learning process to improve the network's accuracy. The network learns by minimizing the difference between its predictions and the actual results, using algorithms like backpropagation.

Neural networks are incredibly powerful for tasks like image recognition, natural language processing, and even playing games. They're behind many of the AI technologies we interact with daily.

**How Does PyTorch Implement Neural Networks?**

PyTorch implements neural networks using the torch.nn module, which provides a wide range of building blocks for creating and training neural networks. Here's a high-level overview of how it works:

1. **Define the Network**: You create a class that inherits from nn.Module. This class will define the layers and the forward pass of the network2.
2. **Initialize Layers**: Inside the class, you initialize the layers in the \_\_init\_\_ method. This can include convolutional layers (nn.Conv2d), linear layers (nn.Linear), activation functions (nn.ReLU), etc.
3. **Define the Forward Pass**: You define the forward method, which specifies how the input data flows through the network.
4. **Instantiate the Network**: You create an instance of your network class.
5. **Train the Network**: You iterate over your dataset, pass the data through the network, compute the loss, backpropagate the gradients, and update the weights.

**Simple Example**

In this section we aim to build super simple neural networks together. Let’s go step by step.

**Step#1**: Importing Libraries; We start by importing the necessary PyTorch libraries. torch is the main package, torch.nn provides the modules to build neural networks, and torch.nn.functional contains functions for operations like activation functions.

import torch  
import torch.nn as nn  
import torch.nn.functional as F

**Step#2**: Defining the Neural Network Class; class SimpleNet(nn.Module): We define a class SimpleNet that inherits from nn.Module. This class represents our neural network.

def \_\_init\_\_(self): This is the constructor method where we define the layers of our network.

self.conv1 = nn.Conv2d(1, 6, 5): This defines a 2D convolutional layer with 1 input channel, 6 output channels, and a 5x5 kernel.

self.fc1 = nn.Linear(16 \* 5 \* 5, 120): This defines a fully connected (linear) layer with 16 \* 5 \* 5 input features and 120 output features.

self.fc2 = nn.Linear(120, 84): Another fully connected layer with 120 input features and 84 output features.

self.fc3 = nn.Linear(84, 10): The final fully connected layer with 84 input features and 10 output features (assuming 10 classes for classification).

class SimpleNet(nn.Module):  
 def \_\_init\_\_(self):  
 super(SimpleNet, self).\_\_init\_\_()  
 self.conv1 = nn.Conv2d(1, 6, 5)  
 self.fc1 = nn.Linear(16 \* 5 \* 5, 120)  
 self.fc2 = nn.Linear(120, 84)  
 self.fc3 = nn.Linear(84, 10)

**Step#3**: Defining the Forward Pass; def forward(self, x): This method defines the forward pass of the network, specifying how data flows through the layers.

x = F.relu(self.conv1(x)): The input x is passed through the first convolutional layer followed by a ReLU activation function.

x = F.max\_pool2d(x, (2, 2)): The output is then passed through a max-pooling layer with a 2x2 window.

x = torch.flatten(x, 1): The tensor is flattened into a 1D vector, starting from the first dimension.

x = F.relu(self.fc1(x)), x = F.relu(self.fc2(x)): The flattened output is passed through the first and second fully connected layers with ReLU activations.

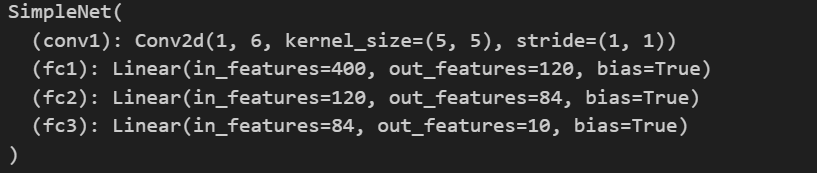
x = self.fc3(x): The output is passed through the final fully connected layer.

return x: The final output is returned.

def forward(self, x):  
 x = F.relu(self.conv1(x))  
 x = F.max\_pool2d(x, (2, 2))  
 x = torch.flatten(x, 1)  
 x = F.relu(self.fc1(x))  
 x = F.relu(self.fc2(x))  
 x = self.fc3(x)  
 return x

**step#4**: Instantiating the Network

net = SimpleNet()  
print(net)  
[Click to see the source example](https://github.com/rose-giant/PyTorch-User-Mannual/blob/main/Traning%20and%20Evaluation/CNN_architecture.ipynb)

Here’s output:  


**Recap**

* torch.Tensor - A *multi-dimensional array* with support for autograd operations like backward(). Also *holds the gradient* w.r.t. the tensor.
* nn.Module - Neural network module. *Convenient way of encapsulating parameters*, with helpers for moving them to GPU, exporting, loading, etc.
* autograd.Function - Implements *forward and backward definitions of an autograd operation*. Every Tensor operation creates at least a single Function node that connects to functions that created a Tensor and *encodes its history*.

**So Far We Covered**

* Defining a neural network
* Processing inputs and calling backward

**What Is a Loss Function?**

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

**Loss Function in PyTorch**

There are several different loss functions under the nn package . A simple loss is: nn.MSELoss which computes the mean-squared error between the output and the target.

[Click to see the source example of loss function](https://github.com/rose-giant/PyTorch-User-Mannual/blob/main/Traning%20and%20Evaluation/LossFunction.ipynb)

**Training a Classifier**

This is it. You have seen how to define neural networks, compute loss and make updates to the weights of the network. In this section we aim to create an image classifier.  
We will do the following steps in order:

1. Load and normalize the CIFAR10 training and test datasets using torchvision
2. Define a Convolutional Neural Network
3. Define a loss function
4. Train the network on the training data
5. Test the network on the test data

[Click to watch the image classifier tutorial](https://drive.google.com/file/d/1q9ig0nwBcDmbtQSO4QWjfwM2iY3eC5EZ/view?usp=sharing)

[Click to see the source example](https://github.com/rose-giant/PyTorch-User-Mannual/blob/main/Traning%20and%20Evaluation/ImageClassifier.ipynb)

**Convolutional Neural Networks**

CNNs (Convolutional Neural Networks) are designed to automatically and adaptively learn spatial hierarchies of features from input images. They consist of several types of layers:

1. Convolutional Layers: Apply filters to the input image, detecting patterns such as edges, textures, or more complex shapes.
2. Pooling Layers: Reduce the dimensionality of the data by down-sampling, which helps in reducing computational cost and controlling overfitting.
3. Fully Connected Layers: At the end of the network, these layers process the high-level features and perform the final classification.

### Basic Structure of a CNN:

1. Input Layer: The raw image data.
2. Convolutional Layer: Applies multiple filters to the input to create feature maps.
3. Activation Function (ReLU): Applies a non-linear transformation.
4. Pooling Layer: Reduces the spatial dimensions.
5. Fully Connected Layers: The features are flattened and passed through one or more fully connected layers for classification.

**Implementing CNN in PyTorch**

### 1. nn.Conv2d

Creates a 2D convolutional layer.

* Syntax: nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0)
* Example: nn.Conv2d(1, 32, 3, 1) creates a convolutional layer with 1 input channel, 32 output channels, and a 3x3 kernel.

### 2. F.relu

Applies the ReLU (Rectified Linear Unit) activation function.

* Syntax: F.relu(input)
* Example: F.relu(x) applies the ReLU activation to the input x.

### 3. F.max\_pool2d

Performs 2D max pooling over an input signal.

* Syntax: F.max\_pool2d(input, kernel\_size, stride=None, padding=0)
* Example: F.max\_pool2d(x, 2) applies 2x2 max pooling to the input x.

### 4. torch.flatten

Flattens a contiguous range of dims into a tensor of rank 1.

* Syntax: torch.flatten(input, start\_dim=0, end\_dim=-1)
* Example: torch.flatten(x, 1) flattens the input x starting from the first dimension.

### 5. nn.Linear

Creates a fully connected (linear) layer.

* Syntax: nn.Linear(in\_features, out\_features)
* Example: nn.Linear(128, 10) creates a fully connected layer with 128 input features and 10 output features.

### 6. F.log\_softmax

Applies the log softmax function to an input tensor.

* Syntax: F.log\_softmax(input, dim=None)
* Example: F.log\_softmax(x, dim=1) applies log softmax to the input x along dimension 1.

[Click to see the CNN tutorial](https://drive.google.com/drive/folders/1xa_1iwvsApZB6_Unk6q_aZvPGfE3hTBr?usp=sharing)

[Click to see the source example](https://github.com/rose-giant/PyTorch-User-Mannual/blob/main/Traning%20and%20Evaluation/CNN_training.ipynb)