Pytorch

Ramex

June 2024

1 Tensors

A Tensor is like an n-dimensional array (multi-dimensional array) in numpy, but with additional features. A tensor can be created from a list or a numpy array. The tensor can be converted to a numpy array using the `numpy()` method. The tensor itself can be used for gpu computations.

1.1 Initializing a Tensor

A tensor can be initialized by a number of methods, for example:

- 1. Directly from data
  `data = [[1,2,3],[4,5,6]],`
  `data_ = torch.tensor(data)`
- 2. From a numpy array
  `data = np.array(data),`
  `data_x = torch.tensor(data)`
- 3. From another tensor (creates a copy of the tensor with ones only)
  `torch.ones_like(x_data)`
- 4. Random or constant values (this takes the shape as input)
  `x_ones = torch.ones_like(x_data)`

1.2 Attributes of a Tensor

A tensor has the following attributes:

- 1. `shape` : The shape of the tensor
- 2. `dtype` : The data type of the tensor
- 3. `device` : The device on which the tensor is stored
- 4. `size` : The number of elements in the tensor
- 5. `numel` : The number of elements in the tensor

- 6. `T` : The transposed tensor
- 7. `contiguous` : The contiguous tensor
- 8. `view` : The view of the tensor
- 9. `requires_grad` : The gradient required for the tensor
- 10. `grad` : The gradient of the tensor
- more ... ⇒ Pytorch Documentation

1.3 Indexing and Slicing

Indexing and slicing works the same as in numpy. For example:

- 1. `x[0]` : The first element of the tensor
- 2. `x[0,0]` : The first element of the first row
- 3. `x[0,:]` : The first row of the tensor
- 4. `x[:,0]` : The first column of the tensor
- 5. `x[0:2,0:2]` : The first two rows and columns of the tensor

1.4 Joining tensors

Tensors can be joined using the `torch.cat()` method. For example:

- 1. `torch.cat([x,y], dim=0)` : Concatenates the tensors along the rows
- 2. `torch.cat([x,y], dim=1)` : Concatenates the tensors along the columns
- 3. `torch.stack([x,y], dim=0)` : Stacks the tensors along the rows
- 4. `torch.stack([x,y], dim=1)` : Stacks the tensors along the columns

1.5 Single-element tensors

A single-element tensor is a tensor with one element. For example:

- 1. `x.item()` : Returns the value of the tensor as a python number
- 2. `x.tolist()` : Returns the value of the tensor as a python list
- 3. `x.numpy()` : Returns the value of the tensor as a numpy array
- 4. `x.to(device)` : Moves the tensor to the specified device

1.6 Tensor to NumPy array

A tensor can be converted to a numpy array using the `numpy()` method. Example:

- 1. `x.numpy()` : Converts the tensor to a numpy array
- 2. `x.cpu().numpy()` : Converts the tensor to a numpy array on the cpu
- 3. `x.cuda().numpy()` : Converts the tensor to a numpy array on the gpu

2 Datasets & DataLoaders

2.1 Loading a Dataset

Loading datasets in PyTorch involves using the torch.utils.data module, which provides utilities for efficiently loading and processing data. The key components include Dataset and DataLoader.

2.1.1 Key Components

- 1. Dataset: An abstract class representing a dataset. You need to subclass this and implement two methods:
- 1.1. __len__: Returns the size of the dataset.
- 1.2. __getitem__: Supports indexing such that dataset[i] can be used to get the ith sample.
- 2. DataLoader: Combines a dataset and a sampler, and provides an iterable over the given dataset. It supports batching, shuffling, and parallel data loading.

2.1.2 Example: Loading a Custom Dataset

Let's go through an example of creating a custom dataset and loading it using PyTorch.

2.1.3 Step 1: Import Required Libraries

```python
import torch
from torch.utils.data import Dataset, DataLoader
import pandas as pd
from sklearn.preprocessing import LabelEncoder
```

2.1.4 Step 2: Create a Custom Dataset

Assume we have a CSV file data.csv with the following structure:

```
text,label
"I love PyTorch", positive
"I hate bugs", negative
```

We will create a custom dataset to load this data.

```
class TextDataset(Dataset):
def __init__(self, csv_file):
    self.data = pd.read_csv(csv_file)
    self.texts = self.data['text'].values
    self.labels = self.data['label'].values

    # Encode labels as integers
    self.label_encoder = LabelEncoder()
    self.labels = self.label_encoder.fit_transform
        (self.labels)

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

def __getitem__(self, idx):
    text = self.texts[idx]
    label = self.labels[idx]
    return text, label
```

2.1.5 Step 3: Instantiate the Dataset and DataLoader

```
# Create an instance of the dataset
dataset = TextDataset(csv_file='data.csv')

# Create a DataLoader for the dataset
dataloader = DataLoader(dataset, batch_size=2, shuffle
    =True, num_workers=2)
```

2.1.6 Explanation of DataLoader Parameters

- dataset: The dataset from which to load the data.
- batch_size: How many samples per batch to load.
- shuffle: Set to True to have the data reshuffled at every epoch.
- num_workers: How many subprocesses to use for data loading. 0 means that the data will be loaded in the main process.

2.2 Step 4: Iterate Through the DataLoader

A dataset can be iterated over using a for loop. For example:

```
    for batch in dataloader:
    texts, labels = batch
    print(texts)
    print(labels)
```

2.3 Using Built-In Datasets

PyTorch also provides utilities for loading several standard datasets, such as MNIST, CIFAR-10, and ImageNet, through the torchvision package.

2.3.1 Example: Loading the MNIST Dataset

```
import torchvision.transforms as transforms
from torchvision.datasets import MNIST

# Define a transform to normalize the data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Download and load the training dataset
train_dataset = MNIST(root='mnist_data', train=True,
    download=True, transform=transform)

# Create a DataLoader for the training dataset
train_loader = DataLoader(train_dataset, batch_size
    =64, shuffle=True, num_workers=2)

# Iterate through the DataLoader
for batch in train_loader:
    images, labels = batch
    print(images.shape)
    print(labels)
    break
```

3 Transforms

Transformers are a type of neural network architecture designed for handling sequential data, such as text. They have become a cornerstone of modern natural language processing (NLP) due to their ability to capture long-range dependencies and parallelize training. Here's an explanation of Transformers and how to implement them using PyTorch.

3.1 Transformer Architecture

The Transformer model was introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017. It consists of an encoder-decoder structure, where both the encoder and decoder are composed of a stack of identical layers.

3.2 Key Components

- 1. Multi-Head Self-Attention Mechanism: Allows the model to focus on different parts of the input sequence simultaneously.
- 2. Positional Encoding: Adds information about the position of words in the sequence.
- 3. Feed-Forward Neural Network: Applied to each position separately and identically.
- 4. Layer Normalization and Residual Connections: Improve training dynamics by normalizing intermediate layers and adding shortcuts to skip connections.

3.3 PyTorch Implementation

PyTorch provides a built-in module for the Transformer model through torch.nn.Transformer. Here's a step-by-step guide to implementing a basic Transformer model in PyTorch.

3.3.1 Step 1: Import Required Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import math
```

Step 2: Positional Encoding

Positional encoding helps the model to understand the order of the sequence.

```
class PositionalEncoding(nn.Module):
def __init__(self, d_model, max_len=5000):
    super(PositionalEncoding, self).__init__()
```

```python
        self.pe = torch.zeros(max_len, d_model).
            unsqueeze(0)
        position = torch.arange(0, max_len).unsqueeze
            (1)
        div_term = torch.exp(torch.arange(0, d_model,
            2) * -(math.log(10000.0) / d_model))
        self.pe[:, :, 0::2] = torch.sin(position *
            div_term)
        self.pe[:, :, 1::2] = torch.cos(position *
            div_term)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

Step 3: Transformer Model

```python
    class TransformerModel(nn.Module):
    def __init__(self, input_dim, model_dim,
        output_dim, nhead, num_encoder_layers,
        num_decoder_layers, dim_feedforward,
        max_seq_length):
        super(TransformerModel, self).__init__()
        self.model_type = 'Transformer'
        self.pos_encoder = PositionalEncoding(
            model_dim, max_seq_length)
        self.encoder = nn.Embedding(input_dim,
            model_dim)
        self.decoder = nn.Embedding(output_dim,
            model_dim)
        self.transformer = nn.Transformer(d_model=
            model_dim, nhead=nhead,
             num_encoder_layers=num_encoder_layers,
             num_decoder_layers=num_decoder_layers,
                 dim_feedforward=dim_feedforward)
        self.fc_out = nn.Linear(model_dim, output_dim)
        self.model_dim = model_dim

    def forward(self, src, tgt, src_mask=None,
        tgt_mask=None):
        src = self.encoder(src) * math.sqrt(self.
            model_dim)
        tgt = self.decoder(tgt) * math.sqrt(self.
            model_dim)
        src = self.pos_encoder(src)
```

```
        tgt = self.pos_encoder(tgt)
        output = self.transformer(src, tgt, src_mask, tgt_mask)
        output = self.fc_out(output)
        return output
```

Step 4: Example Usage

```
    # Define the model parameters
input_dim = 1000  # Size of the input vocabulary
model_dim = 512   # Dimension of the model
output_dim = 1000 # Size of the output vocabulary
nhead = 8         # Number of attention heads
num_encoder_layers = 6
num_decoder_layers = 6
dim_feedforward = 2048
max_seq_length = 100

# Instantiate the model
model = TransformerModel(input_dim, model_dim, output_dim, nhead, num_encoder_layers, num_decoder_layers, dim_feedforward, max_seq_length)

# Define input and output sequences (batch_size, sequence_length)
src = torch.randint(0, input_dim, (10, 32))  # Example source sequence
tgt = torch.randint(0, output_dim, (10, 32)) # Example target sequence

# Forward pass
output = model(src, tgt)

print(output.shape)  # Output shape will be (sequence_length, batch_size, output_dim)
```

4 Build the Neural Network

```
import os   # import the os library
import torch    # import the torch library
from torch import nn    # import the nn library from torch
```

```python
from torch.utils.data import DataLoader # import the DataLoader class
from torchvision import datasets, transforms # import the datasets and transforms library
```

4.1 Get Device for Training

```python
device = (
    "cuda" if torch.cuda.is_available() # check if GPU is available
    else "mps" # use CPU in case GPU is not available
    if torch.backends.mps.is_available() # check if multi-process service is available
    else "cpu" # use CPU in case multi-process service is not available
    else "cpu"
)
print(f"Using {device} device")
```

4.2 Define the Class

We define our neural network by subclassing nn.Module, and initialize the neural network layers in __init__. Every nn.Module subclass implements the operations on input data in the forward method.

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(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
```

4.3 Model Layers

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

```
print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
```

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

5 Automatic Differentiation with torch.autograd

6 Optimizing Model Parameters