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

```
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__()
```

```
        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

```
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
```

```
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

```
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.

```
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

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. For a more detailed walkthrough of this process, check out this video on backpropagation from 3Blue1Brown.

6.1 Prerequisite Code

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

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

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

train_dataloader = DataLoader(training_data,
   batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

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

model = NeuralNetwork()
```

6.2 Hyperparameters

Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates.

We define the following hyperparameters for training:

- 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.

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

6.3 Optimization Loop

Once we set our hyperparameters, we can then train and optimize our model with an optimization loop. Each iteration of the optimization loop is called an epoch.

Each epoch consists of two main parts:

- The Train Loop - iterate over the training dataset and try to converge to optimal parameters.
- The Validation/Test Loop - iterate over the test dataset to check if model performance is improving.

6.4 Loss Function

When presented with some training data, our untrained network is likely not to give the correct answer. Loss function measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training. To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value.

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.

```
# Initialize the loss function
loss_fn = nn.CrossEntropyLoss()
```

6.5 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.

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

```
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.

6.6 Full Implementation

We define train_loop that loops over our optimization code, and test_loop that evaluates the model's performance against our test data.

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for
       batch normalization and dropout layers
    # Unnecessary in this situation but added for best
        practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
```

```
12          loss.backward()
13          optimizer.step()
14          optimizer.zero_grad()
15
16          if batch % 100 == 0:
17              loss, current = loss.item(), batch *
                    batch_size + len(X)
18              print(f"loss: {loss:>7f}  [{current:>5d}/{
                    size:>5d}]")
19
20
21  def test_loop(dataloader, model, loss_fn):
22      # Set the model to evaluation mode - important for
            batch normalization and dropout layers
23      # Unnecessary in this situation but added for best
            practices
24      model.eval()
25      size = len(dataloader.dataset)
26      num_batches = len(dataloader)
27      test_loss, correct = 0, 0
28
29      # Evaluating the model with torch.no_grad()
            ensures that no gradients are computed during
            test mode
30      # also serves to reduce unnecessary gradient
            computations and memory usage for tensors with
            requires_grad=True
31      with torch.no_grad():
32          for X, y in dataloader:
33              pred = model(X)
34              test_loss += loss_fn(pred, y).item()
35              correct += (pred.argmax(1) == y).type(
                    torch.float).sum().item()
36
37      test_loss /= num_batches
38      correct /= size
39      print(f"Test Error: \n Accuracy: {(100*correct)
            :>0.1f}%, Avg loss: {test_loss:>8f} \n")
40
41  loss_fn = nn.CrossEntropyLoss()
42  optimizer = torch.optim.SGD(model.parameters(), lr=
        learning_rate)
43
44  epochs = 10
45  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!")
```

7 Save and Load the Model

In this section we will look at how to persist model state with saving, loading and running model predictions.

```
import torch
import torchvision.models as models
```

7.1 Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called state_dict. These can be persisted via the torch.save method:

```
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using load_state_dict() method.

```
model = models.vgg16() # we do not specify ``weights
    ``, i.e. create untrained model
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```

7.2 Saving and Loading Models with Shapes

When loading model weights, we needed to instantiate the model class first, because the class defines the structure of a network. We might want to save the structure of this class together with the model, in which case we can pass model (and not model.state_dict()) to the saving function:

```
torch.save(model, 'model.pth')
```

We can then load the model like this:

```
model = torch.load('model.pth')
```