Pytorch 101

Xingjian Du

October 18, 2024

Univeristy of Rochester

Table of Contents

Tensors

Tensors

1. What are Tensors?
2. Creating Tensors in PyTorch
3. Tensor Properties
4. Indexing and Slicing
5. Shape Operations

What are Tensors?

- Fundamental data structure in PyTorch
- Generalization of matrices to N dimensions
- Can represent:
  - Scalars (0D tensor)
  - Vectors (1D tensor)
  - Matrices (2D tensor)
  - N-dimensional arrays

Creating Tensors in PyTorch

```python
import torch
# From Python list
x = torch.tensor([1, 2, 3])
# From NumPy array
import numpy as np
np_array = np.array([1, 2, 3])
x = torch.from_numpy(np_array)
# Random tensors
x = torch.rand(3, 3)  # Uniform distribution
x = torch.randn(3, 3)  # Normal distribution
# Zeros and ones
x = torch.zeros(3, 3)
x = torch.ones(3, 3)
```

Tensor Properties

- Dtype: Data type (e.g., float32, int64)
- Device: CPU or GPU
- Shape: Dimensions of the tensor

```python
x = torch.randn(3, 4)
print(x.dtype)  # torch.float32
print(x.device)  # cpu
print(x.shape)  # torch.Size([3, 4])
```

Indexing and Slicing (1/2)

Similar to NumPy:

```
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
# Single element
print(x[0, 0])  # 1
# Row
print(x[1])  # tensor([4, 5, 6])
# Column
print(x[:, 1])  # tensor([2, 5, 8])
```

Indexing and Slicing (2/2)

Advanced indexing:

```python
# Slicing
print(x[0:2, 1:3])
# tensor([[2, 3],
#         [5, 6]])
# Boolean indexing
mask = x > 5
print(x[mask])  # tensor([6, 7, 8, 9])
# Fancy indexing
indices = torch.tensor([0, 2])
print(x[indices])
# tensor([[1, 2, 3],
#         [7, 8, 9]])
```

Shape Operations (1/2)

Reshaping:

```python
x = torch.tensor([1, 2, 3, 4, 5, 6])
# Reshape
y = x.reshape(2, 3)
print(y)
# tensor([[1, 2, 3],
#         [4, 5, 6]])
# View (shares memory with original tensor)
z = x.view(3, 2)
print(z)
# tensor([[1, 2],
#         [3, 4],
#         [5, 6]])
```

Shape Operations (2/2)

Other operations:

```
# Squeeze: Remove dimensions of size 1
x = torch.zeros(2, 1, 3)
print(x.squeeze().shape)  # torch.Size([2, 3])
# Unsqueeze: Add dimension of size 1
x = torch.zeros(2, 3)
print(x.unsqueeze(1).shape)  # torch.Size([2, 1, 3])
# Transpose
x = torch.randn(2, 3)
print(x.t().shape)  # torch.Size([3, 2])
```

Practice Exercise

1. Create a 3x3 tensor of random integers between 0 and 10
2. Slice out the 2x2 submatrix from the top-left corner
3. Reshape the submatrix into a 1D tensor
4. Calculate the mean of the 1D tensor

Bonus: Try to do this in one line of code!

Neural Network Operators

Neural Networks: Core Operators

- Neural networks consist of layers, each performing specific transformations on the input data.
- These transformations are referred to as operators, and they manipulate data to extract useful features or adjust representations.
- Common operators include linear transformations, activation functions, convolution, and pooling operations.

The Linear Layer

- The Linear (Fully Connected) layer is one of the most fundamental layers in neural networks.
- It applies a linear transformation to the input data:

$$y = Wx + b$$

  where W is the weight matrix, x is the input, and b is the bias term.
- Linear layers are typically used at the beginning and end of networks, but can also be found in between.

Example: Linear Layer

```python
import torch.nn as nn
# Define a linear layer with 3 input features and 2 output features
linear = nn.Linear(in_features=3, out_features=2)
# Example input tensor
input_tensor = torch.tensor([1.0, 2.0, 3.0])
output = linear(input_tensor)
```

Activation Functions

- Activation functions introduce non-linearity into the model, which is essential for the network to learn complex patterns.
- Without non-linearity, a neural network would behave like a linear model, no matter how deep it is.
- Common activation functions:
  - ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$
  - Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$
  - Tanh: $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Example: ReLU Activation

```python
import torch.nn.functional as F
# Applying ReLU activation
x = torch.tensor([-1.0, 0.0, 1.0, 2.0])
output = F.relu(x)
print(output)  # Output: tensor([0., 0., 1., 2.])
```

Convolutional Layer

- Convolutional layers are commonly used in models for image processing or spatial data.
- They apply a filter (kernel) that slides over the input, computing a dot product between the filter and the local region of the input.
- Convolutions allow for local feature extraction and enable parameter sharing, reducing the number of parameters needed compared to fully connected layers.
- The operation is defined as:

$$(f * x)(i,j) = \sum_{m,n} f(m,n) \cdot x(i+m, j+n)$$

where f is the filter, and x is the input.

Example: Convolutional Layer

```python
# Convolutional layer: 1 input channel, 1 output channel, 3x3 kernel
conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3)
# Example input: 1x1x5x5 (batch_size x channels x height x width)
input_tensor = torch.randn(1, 1, 5, 5)
output = conv(input_tensor)
```

Pooling Layers

- Pooling layers reduce the spatial dimensions (height and width) of the data, while retaining the most important features.
- Max Pooling selects the maximum value in each region, effectively downsampling the input.
- Average Pooling computes the average value in each region.
- Pooling is essential for reducing computational complexity and controlling overfitting.

Example: Max Pooling

```python
# Max Pooling with 2x2 window and stride 2
pool = nn.MaxPool2d(kernel_size=2, stride=2)
# Example input: 1x1x4x4 (batch_size x channels x height x width)
input_tensor = torch.randn(1, 1, 4, 4)
output = pool(input_tensor)
```

The Output Layer

- The final layer of a neural network is typically a linear layer followed by an activation function that suits the task.
- For classification tasks, a softmax function is used to convert the network's output into probabilities:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

- For binary classification, the sigmoid function is often used, producing an output between 0 and 1.

Example: Softmax Output Layer

```python
# Output layer for classification with 10 classes
output_layer = nn.Linear(in_features=64, out_features=10)
# Applying softmax to the output
output = output_layer(torch.randn(1, 64))  # Example input
softmax_output = F.softmax(output, dim=1)
```

Summary: Neural Network Operators

- Neural networks consist of various operators (layers) that transform data to enable learning of patterns.
- Key operators include:
  - Linear layers for fully connected transformations
  - Activation functions for non-linearity
  - Convolutional layers for spatial feature extraction
  - Pooling layers for dimensionality reduction
  - Output layers for task-specific transformations (e.g., softmax for classification)
- These operators work together to allow the network to learn complex, hierarchical patterns.

Organizing Neural Network Operators

Organizing Operators in a Neural Network

- A neural network is built by stacking multiple layers (operators) in a specific order.
- Each layer processes the input data and passes it to the next layer.
- The structure of the network determines how data flows from input to output.
- PyTorch provides an easy way to organize layers using `torch.nn.Module`, where we define the architecture and the forward pass.

Defining a Neural Network Class in PyTorch

- In PyTorch, we define neural networks as subclasses of `torch.nn.Module`.
- The key components of a neural network class are:
  1. `__init__` method: where we define the layers (operators).
  2. `forward` method: where we define the data flow (how the layers are applied to the input).

Example: Defining a Simple Neural Network

```python
import torch.nn as nn

class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        # Define layers
        self.fc1 = nn.Linear(3, 5)  # Input size 3, output size 5
        self.fc2 = nn.Linear(5, 2)  # Input size 5, output size 2
        self.relu = nn.ReLU()       # ReLU activation

    def forward(self, x):
        # Apply layers in sequence
        x = self.fc1(x)
        x = self.relu(x)  # Apply activation
        x = self.fc2(x)
        return x
```

The Forward Function

- The `forward` function defines the forward pass, specifying how input data moves through each layer.
- It tells the model how to process data step-by-step, from input to output.
- Each layer's output is passed as input to the next layer.

Forward Pass Example

```python
# Create an instance of the network
model = SimpleNN()

# Example input tensor
input_tensor = torch.randn(1, 3)  # Batch size 1, input size 3

# Perform a forward pass
output = model(input_tensor)

print(output)  # Output size will be [1, 2]
```

Sequential Networks

- PyTorch provides the `nn.Sequential` module, which allows us to stack layers sequentially without explicitly defining the `forward` function.
- However, manually defining the `forward` method provides more flexibility, especially when more complex operations (e.g., skip connections) are needed.

Example: Using `nn.Sequential`

```python
model = nn.Sequential(
    nn.Linear(3, 5),
    nn.ReLU(),
    nn.Linear(5, 2)
)

# Forward pass through the Sequential model
input_tensor = torch.randn(1, 3)
output = model(input_tensor)
print(output)
```

Advantages of Custom `forward` Method

- Defining the `forward` function explicitly allows for:
    - Conditional logic (e.g., if-else branching based on data).
    - Complex data flows such as concatenations, element-wise operations, and skip connections (e.g., in ResNet).
    - Greater flexibility when experimenting with custom architectures.
- Sequential networks are easier to set up for simple feedforward architectures, but custom `forward` methods are more generalizable.

Custom Network Example: Adding a Skip Connection

- Some architectures, like ResNet, require custom data flows where outputs from earlier layers are combined with outputs from later layers (skip connections).
- This cannot be done with `nn.Sequential`, and requires a custom `forward` method.

Skip Connection Example

```python
class CustomNN(nn.Module):
    def __init__(self):
        super(CustomNN, self).__init__()
        self.fc1 = nn.Linear(3, 5)
        self.fc2 = nn.Linear(5, 5)
        self.fc3 = nn.Linear(5, 2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x1 = self.fc1(x)
        x1 = self.relu(x1)
        x2 = self.fc2(x1)
        # Skip connection: add input from fc1 to fc2
        x2 = x1 + x2
        x2 = self.relu(x2)
        x3 = self.fc3(x2)
        return x3
```

Summary: Organizing Operators and Forward Pass

- Neural networks are built by organizing operators (layers) in a specific sequence.
- The `forward` method defines the data flow through the network.
- Explicitly defining the `forward` method allows for complex architectures, while `nn.Sequential` is useful for simpler models.
- Understanding how to organize these operators and write the forward function is key to building custom neural networks.

Dataset and DataLoader

Introduction to Dataset and DataLoader

- Neural networks are trained on large datasets. Efficient data handling is crucial for performance.
- `torch.utils.data.Dataset` provides a way to define and manage datasets in PyTorch.
- `torch.utils.data.DataLoader` simplifies loading data in batches and supports shuffling, parallel data loading, and more.

The `Dataset` Class

- The `Dataset` class is an abstract class that represents a dataset.
- Custom datasets can be created by subclassing `Dataset` and implementing two methods:
    1. `__len__`: Returns the total number of data points.
    2. `__getitem__`: Retrieves a single data point at a given index.
- PyTorch also provides built-in datasets for common datasets like MNIST, CIFAR-10, etc.

Creating a Custom Dataset

```python
import torch
from torch.utils.data import Dataset

# Example: Custom dataset for a simple array of data
class SimpleDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, idx):
        sample = self.data[idx]
        label = self.labels[idx]
        return sample, label

# Sample data
data = torch.randn(100, 3)  # 100 samples, 3 features each
labels = torch.randint(0, 2, (100,))  # 100 labels (binary classification)
dataset = SimpleDataset(data, labels)
```

The `DataLoader` Class

- The `DataLoader` class provides an efficient way to load data in batches, shuffle the dataset, and handle parallel processing with multiple workers.
- Key features:
  - **Batching**: Split data into mini-batches for training.
  - **Shuffling**: Randomly shuffle data each epoch to improve generalization.
  - **Parallelism**: Load data in parallel using multiple CPU cores.

Using DataLoader with a Custom Dataset

```python
from torch.utils.data import DataLoader

# Create a DataLoader for the dataset
dataloader = DataLoader(dataset, batch_size=10, shuffle=True)

# Iterate through batches of data
for batch_data, batch_labels in dataloader:
    print(batch_data.size(), batch_labels.size())
    # batch_data: torch.Size([10, 3]), batch_labels: torch.Size([10])
```

Working with Audio Data using `torchaudio`

- PyTorch provides `torchaudio` for loading and preprocessing audio data.
- `torchaudio.datasets` offers built-in datasets like LIBRISPEECH and YESNO.
- Audio data is typically loaded as waveform tensors, which represent the amplitude of the audio signal over time.

Example: Loading the YESNO Dataset with torchaudio

```python
import torchaudio
from torch.utils.data import DataLoader

# Load the YESNO dataset (contains "yes" and "no" spoken in Hebrew)
dataset = torchaudio.datasets.YESNO(root='data', download=True)

# Create a DataLoader for the YESNO dataset
dataloader = DataLoader(dataset, batch_size=5, shuffle=True)

# Iterate over the dataset
for waveforms, labels in dataloader:
    print(waveforms.size(), labels)
    # waveforms: torch.Size([5, 1, n_samples]), labels: tensor of 5 labels
```

Handling Audio Data with `torchaudio.transforms`

- `torchaudio.transforms` provides common audio preprocessing functions, such as:
  - **MelSpectrogram**: Converts waveforms to mel-spectrograms.
  - **Resample**: Resamples the audio to a different sample rate.
  - **AmplitudeToDB**: Converts amplitudes to decibels.
- These transformations are useful for converting raw waveforms into formats suitable for neural network training.

Example: Applying Audio Transformations

```python
import torchaudio.transforms as transforms

# Define a transformation: convert waveform to mel-spectrogram
transform = transforms.MelSpectrogram(sample_rate=16000, n_mels=64)

# Apply the transformation to an example waveform from the dataset
waveform, label = dataset[0]  # Get the first data sample
mel_spectrogram = transform(waveform)
print(mel_spectrogram.size())  # Output: torch.Size([1, 64, time_steps])
```

Summary: Dataset and DataLoader with Audio Data

- `Dataset` and `DataLoader` classes are essential for organizing and efficiently loading data during training.
- `torchaudio` provides tools for handling audio datasets and applying preprocessing transformations.
- Audio data, like other data types, can be loaded in batches and transformed for neural network training using PyTorch's built-in tools.

Training and Evaluation of the Model

Training a Neural Network

- Training involves updating the model's parameters to minimize the error on the training data.
- The key components for training:
  - **Loss Function**: Measures how well the model's predictions match the ground truth.
  - **Optimizer**: Updates the model's parameters based on gradients from backpropagation.
- The training process involves multiple iterations over the dataset (epochs).

Loss Functions

- The loss function measures the difference between the model's predictions and the actual labels.
- Common loss functions:
  - **Cross-Entropy Loss**: Used for classification tasks.
  - **Mean Squared Error (MSE)**: Used for regression tasks.

```python
import torch.nn as nn

# Define a cross-entropy loss for a classification task
criterion = nn.CrossEntropyLoss()
```

Optimizers

- The optimizer updates the model's parameters to minimize the loss.
- It uses the gradients computed during backpropagation to make small adjustments to the weights.
- Common optimizers:
    - **SGD** (Stochastic Gradient Descent)
    - **Adam**: Adaptive learning rate optimization algorithm.

Example: Defining an Optimizer

```python
import torch.optim as optim

# Define the model, optimizer, and loss function
model = SimpleNN()
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

The Training Loop

- The training loop consists of:
  1. **Forward pass**: Compute model predictions for a batch of inputs.
  2. **Loss computation**: Calculate how far the predictions are from the actual labels.
  3. **Backward pass**: Compute gradients via backpropagation.
  4. **Optimizer step**: Update the model's parameters based on the computed gradients.

Example: Training Loop

```
for epoch in range(10):  # Loop over the dataset multiple times
    for batch_data, batch_labels in dataloader:
        # Forward pass
        outputs = model(batch_data)
        loss = criterion(outputs, batch_labels)

        # Backward pass
        optimizer.zero_grad()  # Zero the parameter gradients
        loss.backward()

        # Optimize
        optimizer.step()

    print(f'Epoch [{epoch+1}/10], Loss: {loss.item():.4f}')
```

Evaluating the Model

- After training, it's important to evaluate the model on unseen data to measure its generalization ability.
- Metrics such as accuracy, precision, recall, or F1-score are often used in classification tasks.
- During evaluation, gradients are not needed, so we use ‘torch.no_grad()‘ to avoid unnecessary computations.

Example: Evaluation Loop

```
correct = 0
total = 0
with torch.no_grad():  # Turn off gradient calculation for evaluation
    for batch_data, batch_labels in dataloader:
        outputs = model(batch_data)
        _, predicted = torch.max(outputs, 1)  # Get the class with the highest s
        total += batch_labels.size(0)
        correct += (predicted == batch_labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy: {accuracy:.2f}%')
```

Training and Evaluation Workflow

- **Training**: Use the training data to update the model's parameters by minimizing the loss.
- **Evaluation**: After training, evaluate the model on a separate validation or test set to measure its performance.
- This workflow is repeated until the model achieves satisfactory performance.

Summary: Training and Evaluation

- Training involves forward passes, computing the loss, backward passes, and optimizer steps to update model parameters.
- After training, evaluation is crucial to check the model's generalization on unseen data.
- The combination of loss functions, optimizers, and metrics defines the training and evaluation process.