Muhammad Nur Maajid_1103228145

Introduction to PyTorch

1. **What is PyTorch?**
    - PyTorch is an open-source platform for both machine learning and deep learning.
2. **Applications of PyTorch:**
    - Enables data manipulation and processing.
    - Facilitates the creation of machine learning algorithms using Python code.
3. **Usage in Industry and Research:**
    - Utilized by major tech firms like Meta (Facebook), Tesla, and Microsoft.
    - Employed in AI research by entities such as OpenAI.
    - Example: Tesla's use in autonomous vehicle vision models.
4. **Benefits of PyTorch:**
    - Preferred by machine learning researchers.
    - As of February 2022, it's the most popular deep learning framework on Papers With Code.
    - Supports GPU acceleration for faster code execution.

Section: Introduction to Tensors

1. **What are Tensors?**
    - Tensors are the fundamental building blocks in machine learning.
    - They are used for numerical data representation.
2. **Overview of Tensors:**
    - Tensors represent various data types, e.g., an image as a tensor of shape [3, 224, 224] (color channels, height, width).
    - Tensors have dimensions, like the three-dimensional image tensors.
3. **Creating Tensors:**
    - Scalar: A tensor with a single number (zero dimensions).
    - Vector: One-dimensional tensors with multiple numbers.
    - Matrix: Two-dimensional, versatile data representation tensors.
    - Tensor: Multi-dimensional tensors for representing almost anything.
4. **Tensor Manipulation:**
    - Retrieving tensor information with ndim (number of dimensions) and shape.
    - Handling dimensions, including the creation of 3D tensors.
5. **Random Tensors:**
    - Creating random number tensors using torch.rand().
6. **Tensors of Zeros and Ones:**
    - Generating tensors filled with zeros or ones using torch.zeros() and torch.ones().
7. **Range and Similar Tensors:**
    - Creating tensors from a range of numbers with torch.arange().
    - Generating tensors identical to others with torch.zeros_like() and torch.ones_like().
8. **Tensor Data Types:**
    - Various tensor data types like torch.float32, torch.float16, torch.int8, etc.

    - Precision data types relate to the level of detail in representing numbers.
9. **Common Issues:**
    - Typical problems include dimension, data type, and device issues when interacting with tensors.
10. **Device and Dtype:**

- Being mindful of device and dtype issues is important in PyTorch for consistency.

Gathering Information from Tensors

1. **Common Attributes:**
    - Three frequently used attributes to get information about tensors:
    - shape: The dimension or structure of the tensor.
    - dtype: The data type of the tensor's elements.
    - device: The hardware (GPU or CPU) where the tensor is stored.

Manipulating Tensors (Tensor Operations)

1. **Basic Operations:**
    - Involves addition, subtraction, and multiplication.
    - Values in tensors remain unchanged unless reassigned.
2. **Reassigning Tensors:**
    - Tensors can be modified by reassigning the tensor variable.
3. **Built-in Functions:**
    - PyTorch offers built-in functions like torch.mul() and torch.add() for basic tensor operations.
4. **Element-wise Operations:**
    - These operations apply to each element in a tensor.
5. **Matrix Multiplication:**
    - A common operation in machine learning.
    - PyTorch provides torch.matmul() for matrix multiplication.
6. **Key Notes:**
    - Understanding matrix operations is crucial in deep learning.
    - Using built-in functions is recommended for speed over manual implementation.
7. **Computation Time:**
    - torch.matmul() is faster than manually implementing matrix operations.

Feedback and Corrections

1. **Feedback:**
    - This tutorial offers a solid understanding of tensor manipulation in PyTorch.
    - Explanations on matrix operations, torch.matmul() usage, and visualizations at http://matrixmultiplication.xyz/ are helpful.
    - Introducing shape errors and their resolution with matrix transposing is crucial.
2. **Additional Clarifications/Corrections:**

    - In the "Change tensor datatype" section, further elaboration on computational precision differences would enhance understanding for non-math/computing backgrounds.
    - More examples or explanations in the "Reshaping, stacking, squeezing and unsqueezing" section would provide deeper insights.

Indexing in Tensors (Selecting Data from Tensors):

- In PyTorch, indexing is used to select specific data from tensors. The indexing progresses from outer to inner dimensions. For example, in a tensor x of shape (1, 3, 3), x[0][0][0] selects a specific value. The use of ':' selects all values in a dimension, and ',' separates dimensions.
- Examples:
    - x[:, 0] returns all values in the first dimension at the first index of the second dimension.
    - x[:, :, 1] returns all values from the first and second dimensions at the second index of the third dimension.
- Indexing allows for precise data manipulation and selection in tensors.

PyTorch Tensors & NumPy:

- PyTorch facilitates interaction with NumPy, a popular numerical computing library in Python. Two main methods are used for conversion between NumPy and PyTorch:

1. **Conversion Methods:**
    - torch.from_numpy(ndarray): Converts a NumPy array into a PyTorch tensor.
    - tensor.numpy(): Converts a PyTorch tensor into a NumPy array.

- Usage examples:
    - torch.from_numpy(array) turns a NumPy array into a PyTorch tensor.
    - tensor.numpy() changes a PyTorch tensor into a NumPy array.
- Note that the original data type is retained during conversion unless explicitly changed. To alter the data type, use .type() for tensors or set dtype in NumPy arrays.

Reproducibility in PyTorch:

- Reproducibility refers to achieving the same or very similar results when running code on different computers. In machine learning and neural networks, uncertainties can arise from pseudorandomness, the result of computer-generated random numbers.
- In PyTorch, torch.manual_seed(seed) controls randomness for repeatable results. For instance, two random tensors generated with the same seed will have identical values, enabling reproducible experiments.

Running Tensors on GPU (Enhancing Computation Speed)

- Deep learning algorithms require extensive numerical operations, typically performed on CPUs by default. However, GPUs, or graphics processing units, are often much faster for certain operations needed by neural networks, like matrix multiplication.
- Computers may be equipped with GPUs, and if so, using the GPU for neural network training is likely to dramatically speed up training time.
- Several steps are needed first to access the GPU and then to make PyTorch use the GPU.

1. **Accessing a GPU:**
    - Google Colab: Easy, free, minimal setup, easy to share work via links. Downsides include not saving your outputs, limited computing, and time restrictions.
    - Own GPU: Medium difficulty, runs locally on your machine. Downsides include the cost of the GPU and initial expenses.
    - Cloud computing (AWS, GCP, Azure): Medium to hard difficulty, low initial cost, nearly unlimited computing access. Downsides include potential high costs if run continuously and the need for proper setup.
2. **Getting PyTorch to Run on GPU:**
    - Ensure GPU access with torch.cuda.is_available(). If True, PyTorch can see and use the GPU.
    - Define the device to use (CPU or GPU) with torch.device("cuda" if torch.cuda.is_available() else "cpu").
3. **Placing Tensors (and Models) on GPU:**
    - Use to(device) on tensors (and models) to place them on a specific device (CPU or GPU).
    - Remember, to(device) returns a copy of the tensor. To overwrite it, reassign the result back to the same tensor.
4. **Returning Tensors to CPU:**
    - Use Tensor.cpu() to move tensors back to the CPU. This is necessary for interactions with NumPy, as NumPy does not utilize GPUs.