CMU 10-403/703 Recitation 1
A Short Guide to PyTorch & Gymnasium

1 PyTorch

PyTorch is a powerful machine learning library known for its flexibility and Pythonic design. It provides two primary features: GPU-accelerated Tensor computation and a robust framework for building and training neural networks using its automatic differentiation engine, Autograd.

1.1 Tensors: The Fundamental Building Block

A Tensor is the core data structure in PyTorch. It is a multi-dimensional array, similar to a NumPy `ndarray`, but with the ability to be processed on a GPU for significant computational speedups.

1.1.1 Creating Tensors

You can create Tensors from Python lists, or by using built-in functions to generate Tensors of a specific size and content.

```
import torch

# Create a tensor from a Python list
x_data = torch.tensor([[1, 2], [3, 4]])

# Create a tensor of random numbers with shape (2, 3)
x_rand = torch.randn(2, 3)

# Create a tensor of all ones with the same shape as another
x_ones = torch.ones_like(x_data)
```

1.1.2 Tensor Attributes and GPU Acceleration

Every tensor has attributes describing its `.shape`, `.dtype` (data type), and the `.device` (CPU or GPU) where it is stored. Moving a Tensor to the GPU is a key operation for accelerating training.

```
# Check for GPU availability
device = "cuda" if torch.cuda.is_available() else "cpu"

# Create a tensor and move it to the selected device
tensor = torch.ones(4, 4, device=device)

print(f"Shape: {tensor.shape}")
```

```
print(f"Datatype: {tensor.dtype}")
print(f"Device: {tensor.device}")
```

1.2 Autograd: The Automatic Differentiation Engine

Autograd is the engine that powers a neural network's learning process. It works by building a computation graph on the fly. When a tensor's `requires_grad` attribute is set to `True`, Autograd tracks all operations performed on it. When `.backward()` is called on the final output (the loss), Autograd traverses this graph backward, computing gradients via the chain rule and storing them in the `.grad` attribute of the tensors that require them.

```
# Tensors for model parameters require gradient tracking
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
x = torch.tensor([2.0])
y_true = torch.tensor([5.0])

# Perform a forward pass
y_pred = w * x + b
loss = (y_pred - y_true).pow(2)

# Trigger Autograd to compute gradients
loss.backward()

# Gradients are now stored in the .grad attribute
print(f"Gradient for w: {w.grad}") # d(loss)/dw

# Use the torch.no_grad() context manager during inference
# to prevent tracking and save memory
with torch.no_grad():
  new_pred = w * x + b
```

1.3 `nn.Module`: Building Models Systematically

The `torch.nn` package provides a systematic way to build models by creating a class that inherits from `nn.Module`. This class has two essential methods:

- `__init__()`: Defines and initializes all the layers the model will use (e.g., `nn.Linear`, `nn.ReLU`). This defines the "what".
- `forward()`: Defines the data flow, specifying how an input `x` passes through the layers to produce an output. This defines the "how".

```
import torch.nn as nn

class SimpleMLP(nn.Module):
  def __init__(self, input_size, hidden_size, output_size):
    super(SimpleMLP, self).__init__()
    # Define the layers
```

```
    self.layer1 = nn.Linear(input_size, hidden_size)
    self.activation = nn.ReLU()
    self.layer2 = nn.Linear(hidden_size, output_size)

  def forward(self, x):
    # Define the data flow
    out = self.layer1(x)
    out = self.activation(out)
    out = self.layer2(out)
    return out
```

1.4 The Standard Training Loop

Nearly all PyTorch training follows this five-step loop for each batch of data:

1. **Zero the Gradients**: `optimizer.zero_grad()`. This is necessary because PyTorch accumulates gradients on subsequent backward passes.
2. **Forward Pass**: `outputs = model(inputs)`. Get predictions from the model.
3. **Compute Loss**: `loss = criterion(outputs, labels)`. Compare predictions to true labels.
4. **Backward Pass**: `loss.backward()`. Compute gradients of the loss with respect to model parameters.
5. **Update Weights**: `optimizer.step()`. Update model parameters using the computed gradients.

2 Gymnasium

Gymnasium is a toolkit for developing and comparing reinforcement learning algorithms. It provides a standardized API for RL environments, which decouples the agent's logic from the environment's implementation. This allows for easy benchmarking and testing across a wide variety of tasks.

2.1 Key Components: Environment and Spaces

The environment is the world the agent interacts with. The spaces define the "rules of the game."

- **Observation Space** (`env.observation_space`): Describes what the agent sees. Common types are `Box` (continuous values, like images) and `Discrete` (a fixed set of categories).
- **Action Space** (`env.action_space`): Describes what the agent can do. Also uses types like `Box` (continuous actions) and `Discrete` (discrete actions).

```
import gymnasium as gym

env = gym.make("CartPole-v1")
print(f"Observation Space: {env.observation_space}")
print(f"Action Space: {env.action_space}")
```

2.2 The Agent-Environment Interaction Loop

The core of RL is the interaction loop, governed by two primary functions: `reset()` and `step()`.

```
# 1 Reset the environment to get the first observation
observation, info = env.reset(seed=42)

for _ in range(200):
   # 2 Agent chooses an action
   action = env.action_space.sample() # Random action

   # 3 Environment takes a step based on the action
   observation, reward, terminated, truncated, info = env.step(action)

   # 4 Check if the episode is over
   if terminated or truncated:
      print("Episode Finished!")
      observation, info = env.reset() # Start new episode

env.close()
```

2.2.1 The `step()` Function Return Values

The `env.step(action)` function is crucial. It returns five pieces of information:

1. `observation`: The new state of the environment.
2. `reward`: The scalar feedback signal for the last action.

3. `terminated` (bool): `True` if the episode ended naturally (e.g., agent won or lost). The value of the next state is 0.
4. `truncated` (bool): `True` if the episode was cut short by an external limit (e.g., a time limit). The environment could have continued, so you should bootstrap from the value of the next observation when training.
5. `info` (dict): Diagnostic information not to be used for training.

References and Further Reading

- Official PyTorch Tutorials:
  https://pytorch.org/tutorials/
- Deep Learning with PyTorch: A 60 Minute Blitz:
  https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- Official PyTorch Documentation:
  https://pytorch.org/docs/stable/index.html
- Official Gymnasium Documentation:
  https://gymnasium.farama.org/
- Gymnasium Basic Usage Guide:
  https://gymnasium.farama.org/content/basic_usage/