Lab 1: Introduction to PyTorch and GPU Computing

COMP 395 - 2 : Deep Learning

Due Date: 1-30-2026

Overview

In this assignment, you will set up your deep learning development environment, gain familiarity with PyTorch tensors, and empirically measure the computational speedup achieved by running matrix operations on a GPU compared to a CPU. Understanding the performance benefits of GPU acceleration is fundamental to deep learning, where large-scale matrix operations form the backbone of neural network computations.

Learning Objectives

By completing this assignment, you will:

- Install and configure PyTorch in a Jupyter notebook environment
- Understand the basics of PyTorch tensor creation and manipulation
- Learn how to move tensors between CPU and GPU memory
- Measure and compare execution times for matrix operations on CPU vs. GPU
- Gain practical experience with Google Colab (if needed)

Prerequisites

- Basic Python programming knowledge
- Familiarity with Jupyter notebooks
- A computer with Python 3.8+ installed, **OR** access to Google Colab

Hardware Requirements

You will need access to a GPU for this assignment. You have the following options:

Option A: Local GPU

If you have an NVIDIA GPU, you can run PyTorch locally with CUDA support.

Option B: Apple Silicon (M1/M2/M3)

If you have a Mac with Apple Silicon, you can use PyTorch's MPS (Metal Performance Shaders) backend.

Option C: Google Colab (Required if no local GPU)

If you do not have a GPU or Apple Silicon, you **must** use Google Colab, which provides free GPU access. Go to `https://colab.research.google.com`.

1 Environment Setup (20 points)

1.1 Step 1: Create Your Notebook

For Local Setup:

1. Create a new virtual environment (recommended):

```
python -m venv deeplearning_env
source deeplearning_env/bin/activate  # On Windows: deeplearning_env\
    Scripts\activate

```

2. Install Jupyter and PyTorch:

```
pip install jupyter numpy matplotlib

```

3. Install PyTorch based on your system. Visit `https://pytorch.org/get-started/locally/` and select your configuration. For example:
   - **NVIDIA GPU:** `pip install torch torchvision torchaudio --index-url https://download.p`
   - **Apple Silicon:** `pip install torch torchvision torchaudio`
   - **CPU only:** `pip install torch torchvision torchaudio`
4. Launch Jupyter and create a new notebook:

```
jupyter notebook

```

For Google Colab:

1. Go to `https://colab.research.google.com`
2. Create a new notebook
3. Enable GPU: `Runtime` → `Change runtime type` → Select `T4 GPU`
4. PyTorch is pre-installed on Colab

1.2 Step 2: Verify Your Setup

In your notebook, run the following code to verify your installation and detect available hardware:

```
import torch
import time
import numpy as np
import matplotlib.pyplot as plt

print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
print(f"MPS available: {torch.backends.mps.is_available()}")

# Determine the best available device
if torch.cuda.is_available():
    device = torch.device("cuda")
    device_name = torch.cuda.get_device_name(0)
    print(f"Using CUDA device: {device_name}")
elif torch.backends.mps.is_available():
    device = torch.device("mps")
```

```
    device_name = "Apple Silicon (MPS)"
    print(f"Using MPS device: {device_name}")
else:
    device = torch.device("cpu")
    device_name = "CPU only"
    print("No GPU available, using CPU")
    print("WARNING: Please use Google Colab for this assignment!")
```

Deliverable: Include a screenshot or output showing your PyTorch version and detected device.

2 Matrix Generation and GPU Transfer (20 points)

2.1 Step 3: Create Large Random Matrices

Create a function to generate two large random matrices and prepare them for both CPU and GPU computation:

```
def create_matrices(size, device):
    """
    Create two random matrices of shape (size, size).

    Args:
        size: The dimension of the square matrices
        device: The device to create tensors on ('cpu', 'cuda', or 'mps')

    Returns:
        Two PyTorch tensors on the specified device
    """
    # TODO: Create two random matrices A and B of shape (size, size)
    # using torch.randn() with dtype=torch.float32 on the specified device
    pass

# Test with a small matrix first
size = 1000
A_cpu, B_cpu = create_matrices(size, torch.device("cpu"))
print(f"Matrix A shape: {A_cpu.shape}")
print(f"Matrix A device: {A_cpu.device}")
print(f"Matrix A dtype: {A_cpu.dtype}")
```

Deliverable: Report the shape, device, and dtype of your test matrices.

3 Benchmarking CPU vs GPU (40 points)

3.1 Step 4: Create a Timing Function

Implement a function to measure matrix multiplication time:

```
def benchmark_matmul(A, B):
    """
    Benchmark matrix multiplication with proper synchronization.

    Args:
        A, B: Input matrices

    Returns:
        Execution time in milliseconds
    """
    device = A.device

    # Synchronize before timing (crucial for accurate GPU timing)
    if device.type == "cuda":
        torch.cuda.synchronize()
    elif device.type == "mps":
        torch.mps.synchronize()

    start = time.perf_counter()
    C = torch.mm(A, B)

    # Synchronize after operation
    if device.type == "cuda":
        torch.cuda.synchronize()
    elif device.type == "mps":
        torch.mps.synchronize()

    end = time.perf_counter()
    return (end - start) * 1000  # Convert to milliseconds
```

Important: The `synchronize()` calls are critical! GPU operations are asynchronous, meaning the CPU continues execution before the GPU finishes. Without synchronization, your timing measurements will be incorrect.

3.2 Step 5: Run the Benchmark

Compare CPU and GPU performance across multiple matrix sizes:

```
# Matrix sizes to test
sizes = [512, 1024, 2048, 4096, 8192]

# Store results
results = {
    "sizes": sizes,
    "cpu_times": [],
    "gpu_times": [],
    "speedups": []
}

```

```
print("=" * 60)
print(f"{'Size':>8} | {'CPU (ms)':>12} | {'GPU (ms)':>12} | {'Speedup':>10}")
print("=" * 60)

for size in sizes:
    # CPU benchmark
    A_cpu, B_cpu = create_matrices(size, torch.device("cpu"))
    cpu_time = benchmark_matmul(A_cpu, B_cpu)

    # GPU benchmark
    A_gpu, B_gpu = create_matrices(size, device)
    gpu_time = benchmark_matmul(A_gpu, B_gpu)

    # Calculate speedup
    speedup = cpu_time / gpu_time

    # Store results
    results["cpu_times"].append(cpu_time)
    results["gpu_times"].append(gpu_time)
    results["speedups"].append(speedup)

    print(f"{size:>8} | {cpu_time:>10.2f}ms | {gpu_time:>10.2f}ms | {speedup:>9.2f}x")

print("=" * 60)
```

3.3 Step 6: Visualize Your Results

Create plots to visualize the performance comparison:

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot 1: Execution times
ax1 = axes[0]
x = np.arange(len(sizes))
width = 0.35

bars1 = ax1.bar(x - width/2, results["cpu_times"], width,
                label='CPU', color='steelblue')
bars2 = ax1.bar(x + width/2, results["gpu_times"], width,
                label='GPU', color='coral')

ax1.set_xlabel('Matrix Size (N x N)')
ax1.set_ylabel('Time (milliseconds)')
ax1.set_title('Matrix Multiplication: CPU vs GPU Execution Time')
ax1.set_xticks(x)
ax1.set_xticklabels([str(s) for s in sizes])
ax1.legend()
ax1.set_yscale('log')  # Log scale to see both clearly
ax1.grid(True, alpha=0.3)

# Plot 2: Speedup
ax2 = axes[1]
```

```
ax2.plot(sizes, results["speedups"], 'go-', linewidth=2, markersize=10)
ax2.axhline(y=1, color='r', linestyle='--', label='No speedup (1x)')
ax2.set_xlabel('Matrix Size (N x N)')
ax2.set_ylabel('Speedup (CPU time / GPU time)')
ax2.set_title('GPU Speedup Factor vs Matrix Size')
ax2.grid(True, alpha=0.3)
ax2.legend()

plt.tight_layout()
plt.savefig('cpu_vs_gpu_benchmark.png', dpi=150, bbox_inches='tight')
plt.show()
```

Deliverables:

1. The complete timing table output
2. The generated visualization showing execution times and speedup
3. A brief paragraph (3–5 sentences) explaining why GPU performance is better for large matrices

4 Analysis Questions (20 points)

Answer the following questions in your notebook using Markdown cells:

1. **(5 points)** What is the maximum speedup you observed? At what matrix size did this occur?
2. **(5 points)** Why is the `synchronize()` call necessary for accurate GPU timing? What would happen if you removed it?
3. **(5 points)** Examine your results closely. Do you notice any trends or patterns in how speedup changes with matrix size? Research the architecture of your specific GPU or Apple Silicon device (number of cores, memory bandwidth, compute units, etc.) and write a hypothesis explaining why your timing results look the way they do. What architectural features might account for the performance characteristics you observed?
4. **(5 points)** Based on your results, at approximately what matrix size does using a GPU become worthwhile? Explain your reasoning.

Bonus: Finding the Speedup Threshold (Up to 10 extra points)

Bonus Challenge

Investigate: *At what matrix size does GPU acceleration stop providing significant benefits, or even become slower than CPU?*

For very small matrices, the overhead of transferring data to the GPU and launching GPU kernels can exceed the computational savings. Your task is to find this crossover point experimentally.

Bonus Task

1. **(4 points)** Test matrix sizes ranging from very small (e.g., 16, 32, 64) up to moderate sizes (e.g., 256, 512). Find the approximate matrix size where GPU speedup becomes greater than 1.0x.

```
# Suggested sizes to test for the lower bound
small_sizes = [16, 32, 64, 128, 192, 256, 384, 512, 768, 1024]

# Run benchmarks and find crossover point
# Your code here...

```

2. **(4 points)** Test very large matrix sizes (e.g., 8192, 10240, 12288, 16384) until you encounter memory limitations or diminishing returns. Document:
   - The largest matrix size your GPU can handle
   - How speedup changes at very large sizes
   - Any memory errors encountered and how you handled them

3. **(2 points)** Create a comprehensive plot showing speedup across the full range of matrix sizes (small to large). Mark the "crossover point" where GPU becomes faster than CPU. Write a paragraph explaining the shape of this curve and the factors that influence it.

Hint: Use try-except blocks to gracefully handle out-of-memory errors:

```
try:
    A_gpu, B_gpu = create_matrices(size, device)
    gpu_time = benchmark_matmul(A_gpu, B_gpu)
except RuntimeError as e:
    if "out of memory" in str(e).lower():
        print(f"Size {size}: Out of GPU memory")
        # Clear GPU memory
        if device.type == "cuda":
            torch.cuda.empty_cache()
        gpu_time = float('inf')
    else:
        raise e
```

Submission Requirements

Prepare the following for next friday, we will submit together in GH classroom:

1. **Jupyter Notebook** (`.ipynb`): Your complete notebook with all code cells executed and outputs visible
2. **PDF Export**: A PDF version of your notebook (File → Download as → PDF, or Print to PDF)
3. **Visualization**: The `cpu_vs_gpu_benchmark.png` image file

Grading Rubric

Component	Points
Environment Setup & Verification	20
Matrix Generation Implementation	20
Benchmarking Code & Results	40
Analysis Questions	20
Total	**100**
Bonus: Speedup Threshold Investigation	+15

Tips for Success

- Start early—setting up your environment can take time if you encounter issues
- If using Colab, remember that sessions can time out; save your work frequently
- Comment your code to explain what each section does
- If you encounter errors, include them in your notebook along with how you resolved them
- The warmup and synchronization steps are not optional—they are essential for accurate measurements

Resources

- PyTorch Documentation: `https://pytorch.org/docs/stable/index.html`
- PyTorch CUDA Semantics: `https://pytorch.org/docs/stable/notes/cuda.html`
- Google Colab: `https://colab.research.google.com`
- PyTorch Installation Guide: `https://pytorch.org/get-started/locally/`

Good luck, and welcome to deep learning!