DD2358 - Introduction to High Performance Computing

Project Report

Emile Le Gallic

January 2025

Contents

Chapter 1

Introduction

1.1 Background

This project focuses on the performance analysis and optimization of a Python-based artificial neural network (ANN) implementation designed for multi-class image classification. The implementation, sourced from Philip Mocz's GitHub repository, is intended to classify galaxy images derived from the Sloan Digital Sky Survey (SDSS) and the Galaxy Zoo project datasets.
The ANN follows a three-layer architecture comprising an input layer, a single hidden layer, and an output layer. The input layer represents pixel values of preprocessed galaxy images, while the output layer categorizes images into three classes: **Elliptical (E), Spiral (S), and Irregular (I)**. The hidden layer, with a predefined number of neurons, acts as a feature extractor.
The neural network utilizes the **sigmoid activation function**, with its derivative for backpropagation. Training is performed using a **nonlinear conjugate gradient algorithm**, optimizing weights through iterative minimization of a cost function that includes a **regularization term** to prevent overfitting. The model's performance is evaluated based on classification accuracy and cost reduction during training and testing.
Given the computational complexity associated with training ANNs—especially the repeated evaluations of the cost function and gradients—the objective of this project is to identify and implement optimizations that enhance efficiency without compromising classification accuracy.

1.2 Methodology

The optimization process involved several steps, each targeting different aspects of performance improvement.

1.2.1 Initial Profiling

The initial step involved profiling the original code to identify performance bottlenecks. A custom class, `TimerStats`, was developed to track execution times of key functions, including `predict`, `cost_function`, and `gradient`. This tool helped identify which functions consumed the most time. Additionally, `cProfile` provided a detailed breakdown of execution time, highlighting that the `gradient` function was the most time-consuming, followed by `cost_function` and `predict`.
To gain further insights, `line_profiler` was used for line-by-line profiling, revealing that matrix operations within the `gradient` and `cost_function` were significant bottlenecks. Lastly, `memory_profiler` was employed to understand memory usage patterns, particularly the high memory consumption in the `gradient` function due to repeated matrix operations.

1.2.2 Data Layout Optimization

Based on the profiling results, the next step focused on optimizing data layout and vectorizing operations. The gradient function was vectorized by removing loops and computing activations in matrix form. One-hot encoding of labels was implemented using NumPy indexing. Regularization was applied directly without looping, and gradient flattening was optimized using `np.ravel`. These changes aimed to enhance computational efficiency by leveraging matrix operations over iterative processes.

1.2.3 Optimization with Cython

Cython was utilized to further enhance performance by converting Python code to C, allowing for static typing and efficient memory usage. Static types were declared using Cython's `cdef` keyword, reducing the overhead of Python's dynamic typing. This optimization of NumPy operations at the C level improved the efficiency of matrix multiplications and element-wise functions. Additionally, redundant memory allocations were avoided by reusing pre-allocated arrays, particularly for bias terms, ensuring efficient memory usage throughout the computations.

1.2.4 Torch Optimization

PyTorch was integrated to leverage GPU acceleration, replacing NumPy arrays with PyTorch tensors. This integration allowed matrix operations and activation functions to be computed using PyTorch, reducing memory overhead and improving training efficiency. By utilizing GPU capabilities, the computational performance was significantly enhanced, particularly for large-scale matrix operations that are common in neural network training.

1.2.5 JAX Optimization

JAX was employed for just-in-time (JIT) compilation and automatic differentiation. Matrix operations and activation functions utilized JAX's JIT-compiled functions, reducing execution time and memory overhead. By combining JIT compilation and automatic differentiation, JAX optimized both forward and backward passes of the neural network, providing a robust framework for efficient computation and differentiation.

1.3 Experimental Setup

The computing platform utilized for this project consisted of both local and cloud-based resources.

1.3.1 Local Computing Platform

For local development and initial testing, a MacBook Pro was employed. It features an Apple M1 Pro chip with a total of 8 cores, comprising 6 performance cores and 2 efficiency cores. It is equipped with 16 GB of RAM. The operating system version used was 10151.121.1.
The software environment included Python 3.11.7, along with essential libraries such as NumPy (version 1.26.4) and SciPy (version 1.15.2).

1.3.2 Cloud Computing Platform

For more intensive computations and exploiting GPU acceleration, Google Colab was utilized. The Google Colab environment provided access to a Google Compute Engine backend with Python 3 support and GPU capabilities. This setup included 12.7 GB of system RAM and 15.0 GB of GPU RAM.

Chapter 2

Results

2.1 Initial Profiling

For profiling, we use `MAX_ITER` and `PLOT` to control iteration count and visualization. This speeds up performance tests by reducing training time and avoiding unnecessary plotting. Most tests use `MAX_ITER`=300 and `PLOT`=`False`.

2.1.1 Execution Time Monitoring with `TimerStats`

A custom `TimerStats` class tracks execution times of key functions, computing average, standard deviation, and generating reports via a `timefn` decorator. This is applied to `predict`, `cost_function`, and `gradient` to measure their execution times. At the end of execution, a summary identifies performance bottlenecks.

2.1.2 cProfile Profiling

For `MAX_ITER`=300, the total execution time was **62.203s**, with most time spent in optimization loops. Table 2.2 summarizes the key profiling results.

Table 2.1: Summary of Profiling Results

Function	**Tot Time (s)**	**Cum Time (s)**	**Calls**
`gradient`	27.768	49.351	566
`cost_function`	5.697	8.104	1,168
`predict`	2.190	2.942	603

Performance Bottlenecks

- **Matrix Operations**: Intensive operations in `gradient` and `cost_function`, including `np.hstack`, `np.vstack`, and `np.dot`, consume significant time.
- **Gradient Function Loop**: Iterating over training examples is a major bottleneck.
- **File I/O**: Dataset loading with `np.genfromtxt` is relatively slow.

2.1.3 `line_profiler` Profiling

Profiling results reveal that matrix operations dominate execution time:

- `gradient`: `Delta1 += (delta2 @ a1.T)` accounts for 29.3% of total time.
- `cost_function`: `a2 = g(a1 @ Theta1.T)` takes 68.8% of total time.
- `predict`: `a2 = g(a1 @ Theta1.T)` consumes 72.0% of execution time.

2.1.4 `memory_profiler` Profiling

Memory profiling highlights significant spikes in `gradient` due to repeated matrix operations.

- `predict`: Minor memory usage increments.
- `cost_function`: Memory usage increases during cost function computation.
- `gradient`: High memory consumption due to matrix operations in loops.

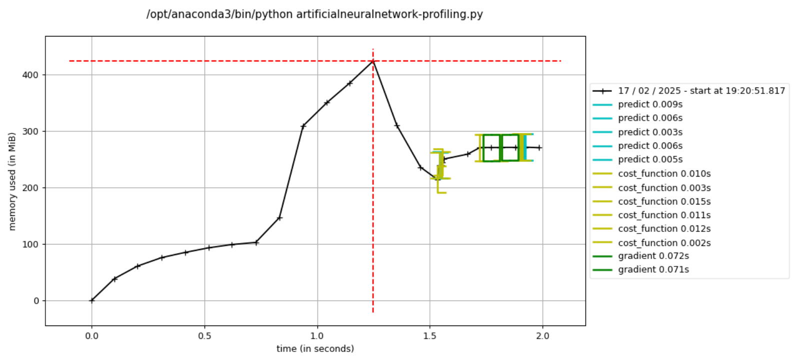

Figure 2.1: Memory Usage Over Time for One Iteration

Profiling suggests optimizing matrix operations and reducing per-iteration memory consumption for better efficiency.

2.2 Data Layout Optimization

2.2.1 Gradient Function Vectorization

A function that could benefit from `NumPy` vectorization was the `gradient` function. Indeed, I removed loops by computing activations in matrix form, used a one-hot encoding of labels using NumPy indexing instead of iterating through samples, replaced iterative updates of gradients with matrix multiplications, applied regularization directly without looping and used optimized gradient flattening using `np.ravel` instead of `np.flatten`.

The changes can be summarized as follows:

Vectorized Gradient Computation

$$X' = \begin{bmatrix} 1 & x_1 & x_2 & \dots & x_n \end{bmatrix} \quad \text{(Add bias)} \tag{2.1}$$

$$Z_2 = X'\Theta_1^T \tag{2.2}$$

$$A_2 = \sigma(Z_2) \tag{2.3}$$

$$A_2' = \begin{bmatrix} 1 & A_2 \end{bmatrix} \quad \text{(Add bias)} \tag{2.4}$$

$$Z_3 = A_2'\Theta_2^T \tag{2.5}$$

$$A_3 = \sigma(Z_3) \tag{2.6}$$

$$Y = \text{One-hot encoding of labels} \tag{2.7}$$

Errors

$$\Delta_3 = A_3 - Y \tag{2.8}$$

$$\Delta_2 = (\Delta_3\Theta_2[:, 1:]) \odot \sigma'(Z_2) \tag{2.9}$$

Gradients

$$\Theta_1^{\text{grad}} = \frac{1}{m}\Delta_2^T X' + \frac{\lambda}{m}\begin{bmatrix} 0 & \Theta_1[:, 1:] \end{bmatrix} \tag{2.10}$$

$$\Theta_2^{\text{grad}} = \frac{1}{m}\Delta_3^T A_2' + \frac{\lambda}{m}\begin{bmatrix} 0 & \Theta_2[:, 1:] \end{bmatrix} \tag{2.11}$$

Flattening

$$\text{grad} = \text{concatenate}(\Theta_1^{\text{grad}}, \Theta_2^{\text{grad}}) \tag{2.12}$$

2.2.2 Performance Analysis

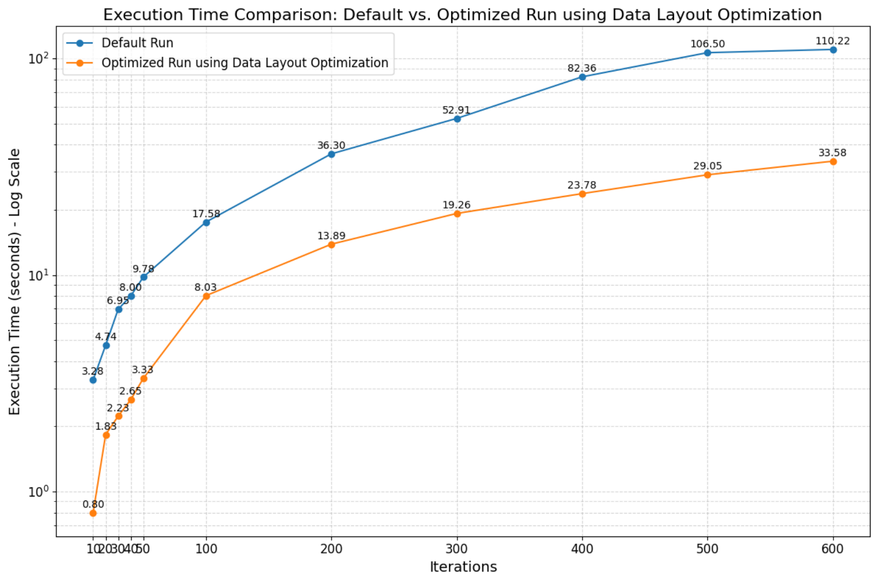

Figure 2.2: Execution Time Comparison: Default vs. Optimized Run

The graph illustrates the execution time comparison between the default implementation and the optimized approach using data layout techniques and vectorization. The optimized run significantly reduces execution time, particularly at lower iteration counts, demonstrating the efficiency of better data organization and parallel computations. As iterations increase, the performance gain remains substantial, with the optimized approach maintaining lower execution times throughout.

2.3 Optimization with Cython

The code was optimized using Cython to improve the performance of critical operations in a neural network. Several changes were introduced to leverage Cython's capabilities:

2.3.1 Type Declarations and Static Typing

Cython allows for the use of static types, which eliminates the overhead of Python's dynamic typing. By using the `cdef` keyword to declare variable types (e.g., `np.float64`, `np.int64`), we can directly use low-level C-like performance for array operations. For example:

```
cdef np.ndarray[np.float64_t, ndim=2] a1 = np.hstack((np.ones((m, 1)), X))
```

This ensures faster execution, as Cython can generate optimized C code with pre-allocated types instead of Python objects.

2.3.2 Vectorized Operations with NumPy

We take full advantage of NumPy's vectorized operations, which Cython can optimize at the C level. Operations such as matrix multiplications and element-wise functions (e.g., the sigmoid function) are computed much faster when performed on entire arrays rather than looping over individual elements. For instance, the forward propagation step:

```
cdef np.ndarray[DTYPE_t, ndim=2] z2 = a1 @ Theta1.T
```

is computed more efficiently with Cython's static typing and optimization over regular Python loops.

2.3.3 Efficient Memory Usage

By using Cython's `cdef` to specify the size and type of NumPy arrays, unnecessary memory copies are avoided. This is particularly important for large datasets, where operations like adding a bias term to the input layer could otherwise require redundant memory allocations. For example, instead of calling `np.hstack()` multiple times for the bias term, the bias column is computed once and reused throughout the forward pass.

2.3.4 Performance Analysis

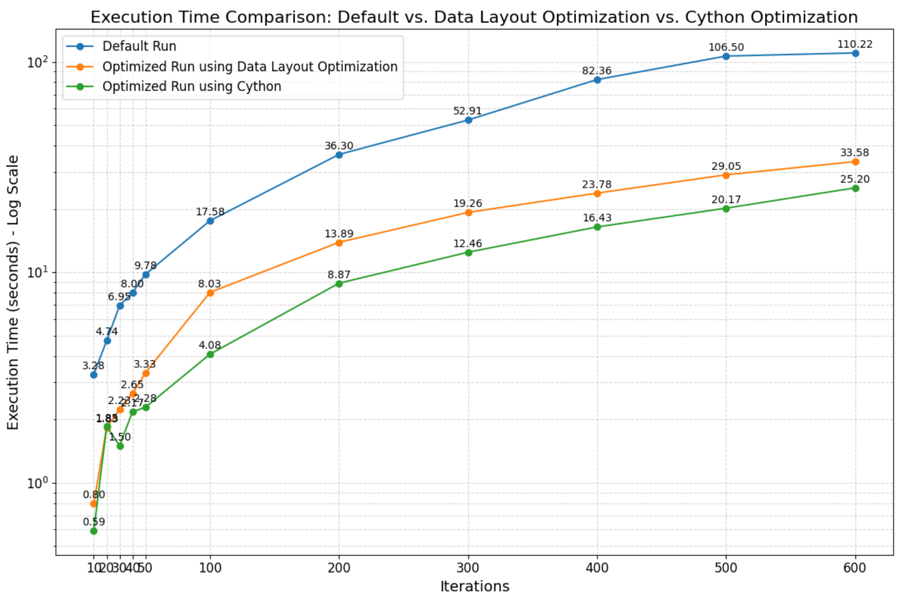

Figure 2.3: Execution Time Comparison: Default vs. Data Layout Optimization vs. Cython Optimization

This graph illustrates that both data layout optimization and Cython significantly enhance execution efficiency compared to the default configuration. Cython optimization provides the most substantial performance gains.

2.4 Torch Optimization

The code was then enhanced to integrate PyTorch and use GPU acceleration for faster computations. NumPy arrays are replaced with PyTorch tensors, improving performance in matrix operations.

2.4.1 Optimized Computations

Matrix operations and activation functions are now computed using PyTorch, reducing memory overhead and improving training efficiency. The sigmoid activation function, crucial for neural network computations, is now implemented efficiently with PyTorch tensors:

```python
def g(x):
    """ Sigmoid activation function optimized with PyTorch """
    return 1.0 / (1.0 + torch.exp(-x))

def grad_g(x):
    """ Gradient of sigmoid function using PyTorch tensors """
    gx = g(x)
    return gx * (1.0 - gx)
```

Furthermore, forward propagation is also accelerated by utilizing tensor operations:

```python
def predict(Theta1, Theta2, X):
    """ Neural network prediction using PyTorch tensors """
    m = X.shape[0]
    a1 = torch.cat((torch.ones((m,1), device=device), X), dim=1)    # Add bias
    a2 = g(a1 @ Theta1.T)                                            # Hidden layer
    a2 = torch.cat((torch.ones((m,1), device=device), a2), dim=1)   # Add bias
    a3 = g(a2 @ Theta2.T)                                            # Output layer
    return torch.argmax(a3, dim=1).reshape((m,1))                   # Get predictions
```

2.4.2 Performance Analysis

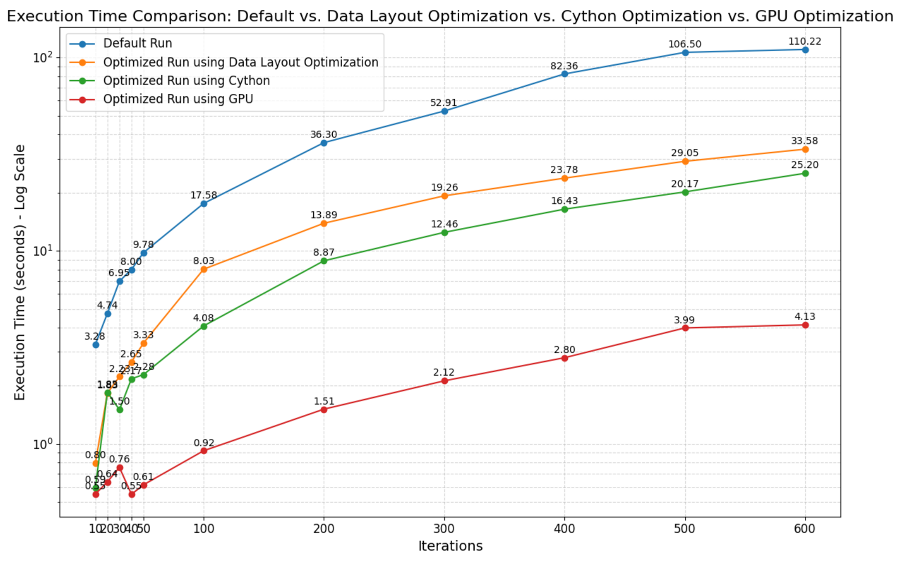

Figure 2.4: Execution Time Comparison: Default vs. Data Layout Optimization vs. Cython Optimization

And here is how the performance varies across different optimization techniques: GPU optimization dramatically reduces execution time, making it the most efficient method, while Cython and data layout optimizations also provide significant improvements over the default configuration.

2.5 JAX Optimization

The code was further optimized by integrating JAX to leverage just-in-time (JIT) compilation and automatic differentiation.

2.5.1 Optimized Computations

Matrix operations and activation functions now utilize JAX's `jit`-compiled functions, reducing execution time and memory overhead. The sigmoid activation function is implemented using JAX's NumPy wrapper (`jax.numpy`), ensuring efficient computation:

```
import jax.numpy as jnp
from jax import jit, grad

@jit
def g(x):
    """ Sigmoid activation function optimized with JAX """
    return 1.0 / (1.0 + jnp.exp(-x))

@jit
def grad_g(x):
    """ Gradient of sigmoid function using JAX """
    gx = g(x)
    return gx * (1.0 - gx)
```

Forward propagation is also optimized using JAX's `jit` for efficient computation:

```
@jit
def predict(Theta1, Theta2, X):
    """ Neural network prediction using JAX """
    m = X.shape[0]
    a1 = jnp.hstack((jnp.ones((m,1)), X))    # Add bias
    a2 = g(a1 @ Theta1.T)                    # Hidden layer
    a2 = jnp.hstack((jnp.ones((m,1)), a2))   # Add bias
    a3 = g(a2 @ Theta2.T)                    # Output layer
    return jnp.argmax(a3, axis=1).reshape((m,1))  # Get predictions
```

Furthermore, the cost function and gradient computation leverage JAX's `jit` and `grad` for improved performance:

```
from jax import lax
from functools import partial

@partial(jit, static_argnums=[1, 2, 3])
def cost_function(theta, input_layer_size, hidden_layer_size, num_labels, X, y, lmbda):
    """ Neural net cost function for a three-layer classification network. """
    Theta1, Theta2 = reshape(theta, input_layer_size, hidden_layer_size, num_labels)
    m = len(y)
    a1 = jnp.hstack((jnp.ones((m, 1)), X))
    a2 = g(a1 @ Theta1.T)
    a2 = jnp.hstack((jnp.ones((m, 1)), a2))
    a3 = g(a2 @ Theta2.T)
    y_mtx = jnp.equal.outer(y.ravel(), jnp.arange(num_labels)).astype(float)
    J = jnp.sum(-y_mtx * jnp.log(a3) - (1.0 - y_mtx) * jnp.log(1.0 - a3)) / m
    J += lmbda / (2.0 * m) * (jnp.sum(Theta1[:, 1:] ** 2) + jnp.sum(Theta2[:, 1:] ** 2))
    return J

@partial(jit, static_argnums=[1, 2, 3])
def gradient(theta, input_layer_size, hidden_layer_size, num_labels, X, y, lmbda):
    return grad(cost_function)(theta, input_layer_size, hidden_layer_size, num_labels, X, y,
        ↪ lmbda)
```

By combining JIT compilation and automatic differentiation, JAX optimizes both forward and backward passes of the neural network.

2.5.2 Performance Analysis

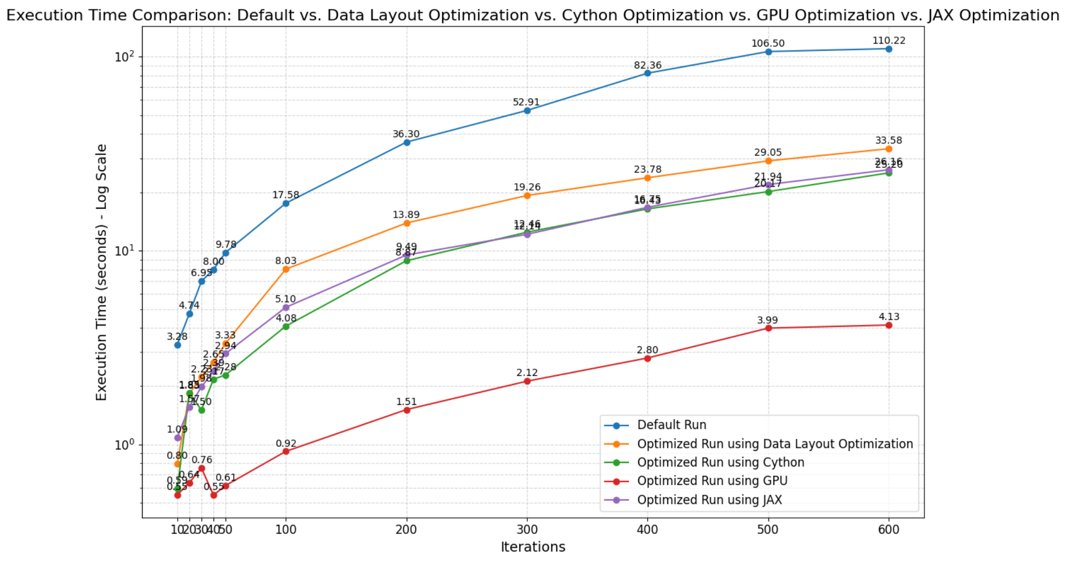

Figure 2.5: Execution Time Comparison: Default vs. JAX Optimization

The performance analysis from the provided graph shows that JAX optimization significantly reduces execution time compared to the default implementation. JIT compilation eliminates redundant computations, while JAX's efficient handling of array operations and automatic differentiation further improves performance.

2.6 Results Summary

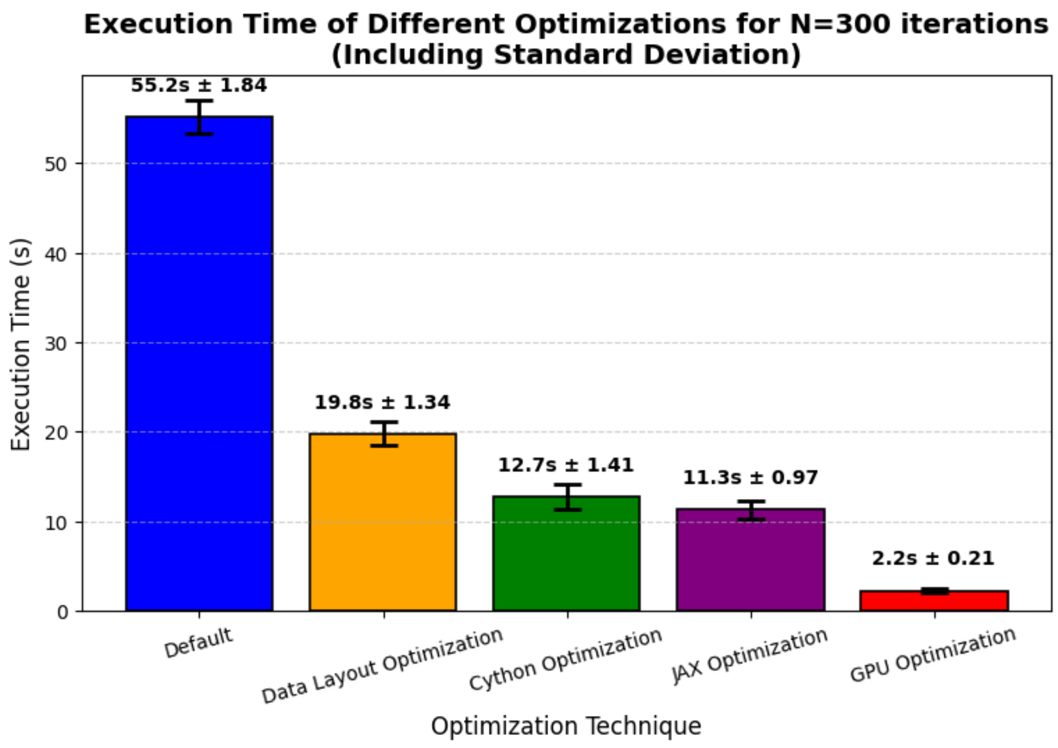

Figure 2.6: Execution Time of Different Optimizations (Including Standard Deviation).

The execution time (in seconds) and standard deviation for each optimization technique are summarized in Table 2.2.

Optimization Technique	Execution Time (s)	Standard Deviation
Default	55.2	1.84
Data Layout Optimization	19.8	1.34
Cython Optimization	12.7	1.41
JAX Optimization	11.3	0.97
GPU Optimization	2.2	0.21

Table 2.2: Execution time and standard deviation for different optimization techniques.

2.6.1 Analysis and Observations

The results indicate significant improvements in execution time when applying optimizations:

- **Default:** The baseline execution time is **55.2s ± 1.84**, making it the slowest method.
- **Data Layout Optimization:** Execution time of **19.8s ± 1.34**, a **64% improvement** over the default.
- **Cython Optimization:** Speeds up training to **12.7s ± 1.41**, a **77% reduction** over default.
- **JAX Optimization:** Brings execution time down to **11.3s ± 0.97**.
- **GPU Optimization:** Achieves the fastest execution at **2.2s ± 0.21**, a **96% improvement**.

2.6.2 Conclusion

The GPU-accelerated approach provides the best performance, reducing execution time by **96%** compared to the default method. Other optimizations, such as JAX and Cython, also yield substantial gains.

2.7 Discussion and Conclusion

The performance analysis of the artificial neural network (ANN) implementation revealed several critical insights into the computational bottlenecks and the effectiveness of various optimization techniques. Initially, profiling identified that matrix operations, particularly within the gradient and cost functions, were significant time and memory consumers. This led to a series of optimizations aimed at enhancing computational efficiency.

2.7.1 Performance Bottlenecks

The primary performance bottlenecks were found in the repeated evaluations of matrix operations and the iterative nature of the gradient computations. These operations are fundamental to neural network training but can be computationally intensive, especially with large datasets. The initial profiling using tools like `cProfile`, `line_profiler`, and `memory_profiler` provided a clear picture of where time and resources were being spent, guiding the optimization efforts.

2.7.2 Optimization Techniques and Results

Several optimization techniques were applied, each targeting different aspects of performance improvement:

1. **Data Layout Optimization**: By vectorizing operations and removing loops, this optimization significantly reduced execution time. The use of NumPy for efficient matrix computations and one-hot encoding further streamlined the process.
2. **Cython Optimization**: Converting Python code to C using Cython allowed for static typing and more efficient memory usage. This optimization showed substantial performance gains, particularly in matrix operations, by leveraging low-level C-like performance.
3. **Torch Optimization**: Integrating PyTorch for GPU acceleration provided the most significant performance improvements. Replacing NumPy arrays with PyTorch tensors allowed for efficient computation on GPUs, drastically reducing execution time for large-scale operations.
4. **JAX Optimization**: JAX's just-in-time (JIT) compilation and automatic differentiation further optimized both forward and backward passes of the neural network. This approach reduced execution time and memory overhead, providing a robust framework for efficient computation.

2.7.3 Challenges and Limitations

While the optimizations yielded substantial improvements, several challenges and limitations were encountered:

- **Parallelization Attempts**: Efforts to incorporate parallelization using Python's `multiprocessing` module and Dask did not yield satisfactory results. These approaches aimed to distribute computations across multiple cores or machines but did not improve performance due to overhead and synchronization issues.
- **Federated Learning**: Exploring federated learning as a means to parallelize computations across decentralized data sources also did not enhance performance. The complexity of coordinating computations across different nodes and the communication overhead outweighed the potential benefits.
- **Hardware Constraints**: The local computing platform, while powerful, had limitations in terms of GPU availability and memory capacity. This constrained the extent to which certain optimizations could be tested and implemented effectively.

2.7.4 Conclusion

GPU acceleration with PyTorch emerged as the most effective technique, followed by JAX and Cython optimizations. These methods significantly reduced execution time and improved efficiency without compromising the accuracy of the neural network.
Future work could explore more advanced parallelization techniques or distributed computing frameworks that better handle the overhead and synchronization challenges. Additionally, leveraging more powerful hardware or cloud-based GPU resources could further enhance performance, particularly for large-scale neural network training tasks.