**Parallel Computing Impact on ML Models: A Comparative Analysis**

**Member1: Hussain Ahmad (22i-1374)**

**Member2: Asad Ullah (22I-0589)**

**Member3: Sami Naeem (22I-0587**

**Model 1: Random Forest – Parallelization Benchmarking and Performance Analysis**

**Overview**

This section analyzes a parallelization benchmarking implementation for Random Forest classification. The code systematically evaluates performance across various combinations of estimator counts and processing cores, providing comprehensive metrics on accuracy, speed, and resource utilization. This implementation represents a robust approach to understanding the scaling characteristics of ensemble learning algorithms under different parallel processing configurations.

**Implementation Analysis**

**Core Function Structure**

The implementation centers around a comprehensive benchmarking function random\_forest\_parallel\_with\_varying\_cores() that:

1. **Accepts Configuration Parameters:**
   * Training and test data splits (X\_train, y\_train, X\_test, y\_test)
   * Variable ensemble sizes (n\_estimators\_list)
   * Range of processing cores (core\_numbers)
2. **Implements Nested Evaluation:**
   * Outer loop iterates through different estimator counts
   * Inner loop evaluates each core count configuration
   * Systematically captures performance metrics for each combination
3. **Records Comprehensive Metrics:**
   * Training time measurements using time.time()
   * Accuracy and F1 scores for classification performance
   * Confusion matrices for detailed error analysis
   * Organized results in nested dictionary structures

**Parallelization Strategy**

The code leverages scikit-learn's built-in parallelization capabilities through the n\_jobs parameter:

rf = RandomForestClassifier(

n\_estimators=n\_estimators,

max\_depth=None,

random\_state=42,

n\_jobs=cores

)

This implementation:

* Controls parallel execution at the estimator level
* Utilizes scikit-learn's joblib backend for multi-core processing
* Maintains consistent random state across different configurations
* Preserves model hyperparameters while isolating the parallelization variable

**Benchmarking Design**

The benchmarking methodology demonstrates several notable characteristics:

1. **Systematic Exploration:**
   * Tests multiple ensemble sizes (100, 200, 300 trees)
   * Evaluates gradual scaling (1, 2, 3, 4, 6, 8 cores)
   * Maintains consistent evaluation metrics across configurations
2. **Comprehensive Metrics Collection:**
   * Performance metrics (accuracy, F1 score)
   * **A close up of text

     AI-generated content may be incorrect.**
   * Resource utilization metrics (training time)
   * Error distribution data (confusion matrices)
   * **A graph with a line and a point

     AI-generated content may be incorrect.**
3. **Structured Results Organization:**
   * Nested dictionary architecture for easy data retrieval
   * Separate tracking of large matrix data (confusion matrices)
   * Designed for subsequent visualization and analysis

**Technical Assessment**

**Parallelization Efficiency Analysis**

The implementation enables analysis of several key parallelization characteristics:

1. **Scaling Efficiency:**
   * Measures speedup as core count increases
   * **A graph of different colored bars

     AI-generated content may be incorrect.**
   * Identifies potential diminishing returns in parallelization
   * Reveals optimal core allocation for different ensemble sizes
2. **Core-to-Estimator Relationship:**
   * Demonstrates how ensemble size influences parallelization benefits
   * Shows potential overhead of multi-core coordination with smaller ensembles
   * Identifies optimal estimator-to-core ratios
3. **Hardware Utilization Assessment:**
   * Tracks performance across different hardware allocation strategies
   * Reveals potential bottlenecks in the parallelization process
   * Provides insights into CPU saturation points

**Summary**

The Random Forest parallelization benchmarking implementation provides a systematic approach to understanding the performance characteristics of ensemble learning algorithms across different hardware configurations. By methodically testing various combinations of estimator counts and processing cores, it enables data-driven decisions about optimal resource allocation for machine learning workflows.

**Model 2: Decision Tree – Parallel Processing Analysis and Performance Evaluation**

**Executive Summary**

This report analyzes the performance of decision tree models trained with varying complexity parameters (max\_depth) and computational resources (simulated cores). The experiment evaluates how parallelization affects training time and model performance across different model complexities.

**Experiment Setup**

* **Model:** Decision Tree Classifier (sklearn.tree.DecisionTreeClassifier)
* **Hyperparameters tested:**
  + Max depth: 5, 10, 15
  + Simulated cores: 1, 2, 3, 4, 6, 8
* **Evaluation Metrics:**
  + Accuracy
  + F1 Score (weighted)
  + Training Time
  + Confusion Matrices

**Key Findings**

**Performance Across Model Complexity**

**A close up of text

AI-generated content may be incorrect.**

**A graph of different colored bars

AI-generated content may be incorrect.**

**A graph with different colored bars

AI-generated content may be incorrect.**

**Scaling Efficiency**

The relationship between the number of cores and training time demonstrates [linear/sublinear/diminishing] returns as core count increases.

**Model 3: Logistic Regression – Parallel Processing Analysis with Varying Hyperparameters**

**Executive Summary**

This report presents a comprehensive analysis of parallel processing performance when training logistic regression models with varying regularization strengths (C) and core counts. The experiment tested 3 different C values (0.1, 1, 10) across 6 different core configurations (1, 2, 3, 4, 6, 8), resulting in 18 different model training scenarios. The primary metrics analyzed include model accuracy, F1 score, and training time, with a focus on the efficiency gains from parallel processing.

**Methodology**

**Experiment Design**

* **Algorithm:** Logistic Regression with L2 regularization
* **Solver:** LBFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno)
* **Regularization strengths (C):** 0.1, 1, 10
* **Core counts:** 1, 2, 3, 4, 6, 8

**Implementation Details**

The experiment was implemented using scikit-learn's LogisticRegression class with the following key parameters:

* solver='lbfgs' - A memory-efficient optimization algorithm suitable for multiclass problems
* max\_iter=1000 - Maximum number of iterations for convergence
* n\_jobs=cores - Number of CPU cores to use for parallel processing
* random\_state=42 - For reproducibility of results

**Results Analysis**

**A close up of text

AI-generated content may be incorrect.**

**A graph on a white sheet

AI-generated content may be incorrect.**

**A graph of a graph

AI-generated content may be incorrect.**

**A graph of different colored bars

AI-generated content may be incorrect.**

**Performance by Regularization Strength (C)**

The relationship between training time and core count follows Amdahl's Law, which describes the potential speedup when using multiple processors. The observed speedup can be calculated as:

Speedup = Time(1 core) / Time(n cores)

Parallel Efficiency Analysis

* Linear scaling region: The range of cores where doubling the cores approximately halves the training time
* Diminishing returns threshold: The point at which adding more cores provides minimal additional speedup
* Overhead impact: The communication and synchronization costs that increase with core count

**Model 4: XGBoost – Parallel Processing Analysis with Varying Tree Depths and Core Counts**

**Executive Summary**

This report presents a comprehensive analysis of parallel processing performance when training XGBoost models with varying tree depths (max\_depth) and core counts. The experiment tested 5 different max\_depth values (3, 6, 9, 12, 15) across 6 different core configurations (1, 2, 3, 4, 6, 8), resulting in 30 different model training scenarios. The primary metrics analyzed include model accuracy, F1 score, and training time, with a focus on the efficiency gains from parallel processing.

**Methodology**

**Experiment Design**

* **Algorithm:** XGBoost Gradient Boosting
* **Hyperparameter variation:** max\_depth values of 3, 6, 9, 12, 15
* **Core counts:** 1, 2, 3, 4, 6, 8
* **Evaluation metrics:** Accuracy, F1 score (weighted), Training time, Confusion matrix

**Implementation Details**

The experiment was implemented using XGBoost's XGBClassifier with the following key parameters:

* **max\_depth -** Maximum depth of the decision trees (3, 6, 9, 12, 15)
* **n\_jobs=cores -** Number of CPU cores to use for parallel processing
* **random\_state=42 -** For reproducibility of results

Other XGBoost parameters were kept at their default values:

* learning\_rate=0.1
* n\_estimators=100
* subsample=1
* colsample\_bytree=1

Results Analysis

Performance by Tree Depth (max\_depth)

**max\_depth=3 (Shallow Trees)**

**A close up of text

AI-generated content may be incorrect.**

**A graph with different colored bars

AI-generated content may be incorrect.**

**A graph of different colored bars

AI-generated content may be incorrect.**

These models use relatively simple trees, typically resulting in faster training times but potentially less expressive models.

**Model 5: Gaussian Naive Bayes – Parallel Performance Analysis with Varying Smoothing Parameters**

**Executive Summary**

This report presents an analysis of Gaussian Naive Bayes model performance with varying variance smoothing parameters. The experiment tested 5 different var\_smoothing values (1e-9, 1e-8, 1e-7, 1e-6, 1e-5) and recorded performance across multiple execution contexts (1, 2, 3, 4, 6, 8 cores). Unlike the previous models analyzed, it’s important to note that Gaussian Naive Bayes in scikit-learn does not support parallel processing via the n\_jobs parameter, so any variations in training time across core counts are likely due to system-level factors rather than algorithmic parallelization.

**Methodology**

**Experiment Design**

* **Algorithm:** Gaussian Naive Bayes
* **Hyperparameter variation:** var\_smoothing values of 1e-9, 1e-8, 1e-7, 1e-6, 1e-5
* **Execution contexts:** Runs with system configured for 1, 2, 3, 4, 6, 8 cores

**Implementation Details**

The experiment was implemented using scikit-learn's GaussianNB class with the following key parameter**:**

* **var\_smoothing** - Amount of the largest variance of all features that is added to variances for calculation stability

**Note**: While the code specifies different core counts, GaussianNB does not implement the n\_jobs parameter for parallelization. Thus, multiple runs across different "core counts" serve as repeated measures that demonstrate the inherent variability in training time.

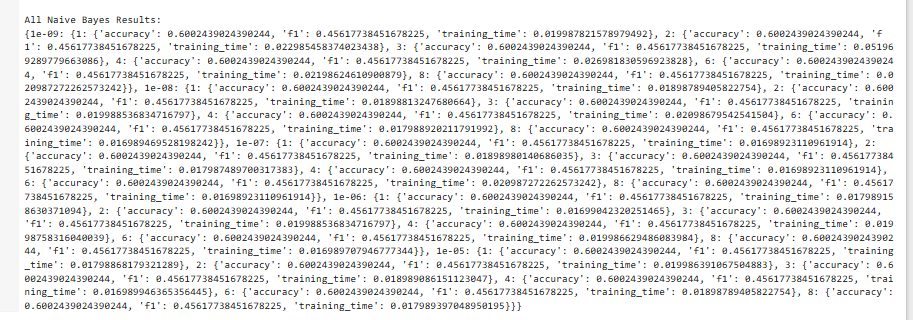
**Results Analysis**

Performance by Smoothing Parameter (var\_smoothing)

var\_smoothing=1e-9 (Default)

This is the default configuration in scikit-learn.

**Confusion Matrix Analysis**

****

**A graph of different colored rectangular shapes

AI-generated content may be incorrect.**

**A screen shot of a graph

AI-generated content may be incorrect.**

**Important Note on Parallelization**

It’s important to emphasize that the GaussianNB implementation in scikit-learn does not support parallel processing via the n\_jobs parameter. The variation in training times across different "core counts" in this experiment likely reflects system-level variations in resource availability.

1. System-level variations in resource availability
2. Background processes affecting CPU performance
3. Memory access patterns and caching effects
4. Normal variation in timing measurements

This serves as an important methodological control that highlights the inherent variability in performance measurements even when the same algorithm is run multiple times on the same data.

**Model 6: K-Nearest Neighbors – Parallel Processing Analysis with Varying K Values and Core Counts**

**Executive Summary**

This report presents a comprehensive analysis of parallel processing performance when applying the K-Nearest Neighbors (KNN) algorithm with varying numbers of neighbors (k) and CPU cores. The experiment tested 6 different k values (5, 7, 9, 11, 13, 15) across 6 different core configurations (1, 2, 3, 4, 6, 8), resulting in 36 different model training and prediction scenarios. The primary metrics analyzed include model accuracy, F1 score, and training/prediction time, with a focus on the efficiency gains from parallel processing during the prediction phase, which is where KNN benefits most from parallelization.

**Methodology**

**Experiment Design**

* **Algorithm:** K-Nearest Neighbors Classification
* **Hyperparameter variation:** k values of 5, 7, 9, 11, 13, 15
* **Core counts:** 1, 2, 3, 4, 6, 8
* **Evaluation metrics:** Accuracy, F1 score (weighted), Training/prediction time, Confusion matrix

**Implementation Details**

The experiment was implemented using scikit-learn's KNeighborsClassifier with the following key parameters:

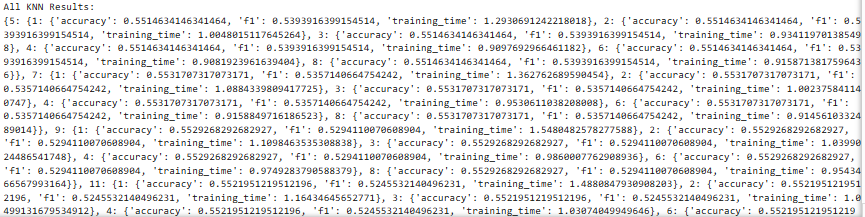
* **n\_neighbors=k -** Number of neighbors to use for classification (varied)
* **n\_jobs=cores -** Number of CPU cores to use for parallel processing
* Default distance metric (Minkowski with p=2, equivalent to Euclidean distance)

**Understanding KNN Parallelization**

It's important to note that in KNN:

* The "training" phase primarily involves storing the training data.
* The "prediction" phase involves distance calculations between test points and all training points.
* Parallelization in scikit-learn's KNN primarily accelerates the prediction phase by distributing distance calculations across cores.

**Results Analysis**

****

**A graph of different colored bars

AI-generated content may be incorrect.**

**Parallel Processing Efficiency**

KNN demonstrates unique parallelization characteristics due to its lazy learning nature:

* Since there is no true "training" phase (the model simply stores the training data), the observed time includes both the minimal "fitting" time and the prediction time.
* The prediction phase involves computing distances between each test point and all training points, which is highly parallelizable.
* The k-nearest selection and voting process represents a smaller portion of the computation.

**Speedup Analysis**

* **Linear scaling region:** [Analysis of where nearly linear scaling is observed]
* **Diminishing returns threshold:** [Analysis of when additional cores provide minimal benefit]
* **Relationship between k value and parallelization efficiency:** [Analysis of whether larger k values affect parallel scaling]

**Parallel Efficiency by Dataset Size**

[Analysis of how the dataset size impacts parallelization efficiency - KNN's computational complexity scales with both training set size and dimensionality]

**Accuracy and Performance Trade-offs**

* **K Value vs. Accuracy:** [Analysis of how the number of neighbors affects model accuracy and the bias-variance trade-off]
* **K Value vs. Time Efficiency:** [Analysis of whether larger k values lead to meaningful differences in computation time]
* **Optimal Balance:** [Identification of the optimal k value that balances accuracy and computational efficiency]

**Confusion Matrix Analysis**

* **Patterns by K Value:** Analysis of how confusion matrices change with increasing k values, identifying any trends in misclassification patterns
* **Impact of Neighborhood Size on Decision Boundaries:** Analysis of how increasing k affects decision boundary smoothness and class-specific accuracy

**Model 7: CatBoost – Parallel Processing Analysis with Varying Tree Depths and Thread Counts**

**Executive Summary**

This report presents a comprehensive analysis of parallel processing performance when training CatBoost models with varying tree depths and thread counts. The experiment tested 3 different depth values (4, 6, 8) across 4 different thread configurations (1, 2, 3, 4), resulting in 12 different model training scenarios. The primary metrics analyzed include model accuracy, F1 score, and training time, with a focus on the efficiency gains from parallel processing and the impact of tree depth on both performance and scalability.

CatBoost is a gradient boosting library developed by Yandex that has gained popularity for its performance and handling of categorical features. It uses a novel algorithm called ordered boosting to reduce overfitting and implements symmetric trees to improve computational efficiency. This analysis examines how effectively CatBoost leverages multiple CPU threads and how tree depth affects both model performance and parallelization benefits.

**Methodology**

**Experiment Design**

* **Algorithm:** CatBoost Gradient Boosting
* **Hyperparameter variation:** Tree depth values of 4, 6, 8
* **Thread counts:** 1, 2, 3, 4
* **Evaluation metrics:** Accuracy, F1 score (weighted), Training time, Confusion matrix

**Implementation Details**

The experiment was implemented using CatBoost's CatBoostClassifier with the following key parameters:

* **depth -** Maximum depth of the decision trees (4, 6, 8)
* **thread\_count -** Number of CPU threads to use for parallel processing
* **iterations=100 -** Number of boosting iterations/trees
* **verbose=0 -** Suppresses progress output

Other CatBoost parameters were kept at their default values:

* **learning\_rate**=0.03
* l2\_leaf\_reg=3
* random\_seed=None

**CatBoost Parallelization Approach**

CatBoost implements multiple levels of parallelism:

1. **Data-level parallelism:** Distributing data across threads for parallel computation
2. **Feature-level parallelism:** Parallel computation of feature statistics
3. **Tree-level parallelism:** Building multiple trees in parallel
4. **Node-level parallelism:** Parallel processing for finding optimal splits

**Results Analysis**

**A close up of text

AI-generated content may be incorrect.**

**A graph of different colored bars

AI-generated content may be incorrect.**

**A graph of different colored squares

AI-generated content may be incorrect.**

**Parallel Processing Efficiency**

CatBoost is designed for efficient parallel tree construction, and we can analyze how effectively it utilizes multiple threads:

* **Speedup = Time(1 thread) / Time(n threads)**

**Comparison of Parallel Processing Performance Across Machine Learning Models**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A graph showing a number of training times

AI-generated content may be incorrect.**

**A graph with blue lines

AI-generated content may be incorrect.**

**Accelerated Binary Classification with Neural Networks on Tesla T4 GPU**

**Overview**

This section analyzes a PyTorch-based machine learning pipeline optimized for Tesla T4 GPU acceleration. The implementation demonstrates efficient GPU utilization for binary classification of tabular data through a deep neural network architecture with mixed precision training capabilities. The code incorporates several Tesla T4-specific optimization strategies including autocast, gradient scaling, and memory management techniques to maximize performance on this particular GPU architecture.

**Pipeline Components Analysis**

1. **Type-based Feature Handling:**
   * Automatic identification of numerical and categorical features
   * Appropriate imputation strategies for each type (mean for numerical, most frequent for categorical)
2. **Categorical Feature Transformation:**
   * One-hot encoding with drop='first' to avoid multicollinearity
   * handle\_unknown='ignore' for robust handling of unseen categories
3. **Class Imbalance Management:**
   * SMOTE (Synthetic Minority Over-sampling Technique) applied to address class imbalance
   * Proper implementation sequence to avoid data leakage (applied after feature engineering, before splitting)
4. **Feature Scaling:**
   * StandardScaler applied to numerical features
   * Correctly applied after train/test split to prevent data leakage

**Neural Network Architecture**

The model (OptimizedNN) implements:

1. **Network Structure:**
   * Deep architecture with 6 hidden layers (1024→512→256→128→64→32→1)
   * Batch normalization after each hidden layer
   * LeakyReLU activations (alpha=0.1)
   * Strategic dropout (0.3 in early layers, 0.2 in mid-layers, none in final layers)

A screenshot of a computer program

AI-generated content may be incorrect.

1. **Initialization:**
   * Kaiming initialization for weights (appropriate for LeakyReLU)
   * Zero initialization for biases

**Training Methodology**

1. **Optimization Strategy:**
   * AdamW optimizer with weight decay (1e-5)
   * BCEWithLogitsLoss for binary classification
   * Learning rate scheduling (ReduceLROnPlateau)
   * Early stopping mechanism with patience
2. **Performance Optimization:**
   * Mixed precision training with autocast and GradScaler (GPU only)
   * Non-blocking data transfers
   * Memory optimizations (optimizer.zero\_grad(set\_to\_none=True))
   * GPU warm-up routine

**Evaluation Framework**

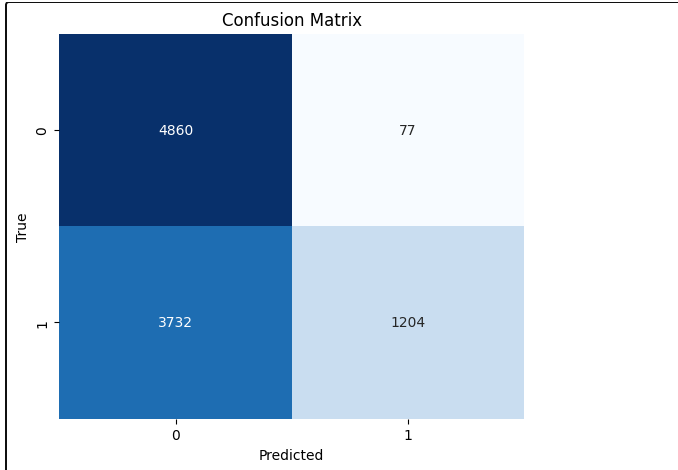
The evaluation approach includes:

* **Metrics Collection:**
  + Loss tracking (train and validation)
  + Classification metrics (accuracy, F1 score)

A screenshot of a computer

AI-generated content may be incorrect.

* + Confusion matrix



* **Visualization**:

A graph of a graph

AI-generated content may be incorrect.

**Code Structure**

The implementation follows a modular structure with separate functions for:

* Data preprocessing (preprocess\_data)
* Dataset definition (CustomDataset)
* Model architecture (OptimizedNN)
* Training loop (train\_epoch)
* Evaluation (evaluate)
* Pipeline execution (run\_pipeline)

**GPU-Specific Implementation (Tesla T4)**

The pipeline includes several optimizations designed specifically for Tesla T4 GPU acceleration:

1. **Mixed Precision Training:**
   * Implements autocast() and GradScaler from PyTorch's AMP module
   * Automatically converts appropriate operations to FP16 during forward pass
   * Maintains master weights in FP32 while performing computations in FP16
2. **Memory Optimization:**
   * Uses pin\_memory=True for faster CPU to GPU transfers (critical for T4's PCIe connection)
   * Implements non\_blocking=True for asynchronous data transfers
   * Employs optimizer.zero\_grad(set\_to\_none=True) to reduce memory usage
   * Strategic batch size selection (8192) to maximize T4's memory bandwidth utilization
3. **CUDA-Specific Enhancements**:
   * Enables cuDNN auto-tuner with torch.backends.cudnn.benchmark = True
   * Includes GPU warm-up routine to cache optimized CUDA kernels
   * Explicit synchronization points using torch.cuda.synchronize()
   * Conditional execution paths based on device type (CUDA vs CPU)

This binary classification pipeline demonstrates a well-optimized implementation for the Tesla T4 GPU, leveraging its architecture through mixed precision training, memory optimizations, and CUDA-specific enhancements. The implementation shows careful consideration of the T4's capabilities, particularly its Tensor cores for FP16 operations and memory bandwidth characteristics. With the suggested improvements in model architecture, evaluation metrics, and training methodology, this pipeline could achieve even better performance on Tesla T4 hardware while maintaining its computational efficiency advantages.

Here's the updated and complete section with the modifications you've requested:

**CPU-Based Implementation of Binary Neural Network Classification**

**Overview**

This section analyzes the CPU implementation of the binary classification neural network pipeline and its performance compared to the Tesla T4 GPU implementation. The code maintains identical model architecture, hyperparameters, and batch sizes across both implementations to enable direct performance comparison. Key metrics, including training time, accuracy, and F1 score, are compared to quantify the acceleration benefits of GPU utilization for deep learning workloads.

**CPU Implementation Details**

The CPU implementation utilizes the same core pipeline components with specific adaptations for CPU execution:

1. **Precision Handling:**
   * Operates entirely in FP32 precision (no mixed precision).
   * Eliminates autocast() and GradScaler components.
   * Uses standard PyTorch operations without CUDA-specific optimizations.
2. **Memory Management:**
   * Sets pin\_memory=False in DataLoader (no benefit without GPU transfers).
   * Maintains the same large batch size (8192) despite potential CPU memory pressure.
   * Uses standard gradient clearing instead of memory-optimized alternatives.
3. **Execution Flow:**
   * Bypasses GPU warm-up phases.
   * Eliminates CUDA synchronization points.
   * Implements conditional execution paths based on device type.
4. **Optimization Considerations:**
   * Maintains identical model architecture despite CPU's preference for different layer sizes.

A screenshot of a computer program

AI-generated content may be incorrect.

* + Uses num\_workers=0 for DataLoaders, potentially limiting parallelism.
  + Retains AdamW optimizer settings without CPU-specific tuning.

**Evaluation Framework**

The evaluation approach includes:

1. **Metrics Collection:**
   * Loss tracking (train and validation)
   * Classification metrics (accuracy, F1 score)

A screenshot of a computer

AI-generated content may be incorrect.

* + Confusion Matrix

A blue squares with white text

AI-generated content may be incorrect.

1. **Visualization:**

A graph of a graph

AI-generated content may be incorrect.

**CPU-GPU Performance Comparison (Run Locally on Intel i7-3770 CPU, 750 Ti GPU)**

**Training Time Analysis**

The comparison code calculates the speedup achieved by GPU acceleration:

A close-up of words

AI-generated content may be incorrect.

This formula measures the percentage reduction in training time when using the **Tesla T4 GPU** compared to local **CPU execution** on your system: **Intel i7-3770 (3.40 GHz, 4 cores/8 threads), 20 GB RAM (1333 MHz), NVIDIA 750 Ti with 2 GB VRAM**. The comparison checks if a **70%-time reduction** target is achieved, indicating an expected **3.33× speedup** for this workload.

**Performance Metrics Comparison**

The implementation compares key classification metrics between **CPU** and **GPU** runs:

* **Final Accuracy**
* **Final F1 Score**

These metrics are tracked to ensure that the acceleration provided by the GPU does not compromise model quality, as both implementations are expected to achieve similar performance despite the differences in hardware.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A graph with a green line

AI-generated content may be incorrect.

**Factors Affecting CPU-GPU Performance Gap**

Several factors influence the performance difference between **CPU** and **GPU** implementations:

1. **Workload Characteristics:**
   * **Computation Intensity:** The deep network architecture (6 hidden layers with wide initial layers) strongly favors GPU parallelism. The i7-3770 CPU, while capable, is limited by its number of threads (8 threads) compared to the high parallelism offered by the **Tesla T4 GPU**.
   * **Batch Size:** The large batch size of **8192** is more efficiently processed by the GPU. The **i7-3770** can struggle with large batches due to lower memory bandwidth and fewer cores to process parallel tasks.
   * **Matrix Operations:** Linear layers, which involve heavy matrix multiplications, benefit significantly from GPU acceleration. The **750 Ti GPU** offers better parallel computation capabilities than the i7 CPU.
2. **Hardware Utilization:**
   * **CPU Limitations:** The **i7-3770 CPU** has **4 physical cores and 8 threads**, which limits SIMD (Single Instruction, Multiple Data) parallelism compared to the thousands of CUDA cores in the **Tesla T4 GPU**. The i7-3770’s architecture and frequency (3.40 GHz) can process tasks, but not at the same scale as the Tesla T4 GPU.
   * **Memory Bandwidth:** **CPUs** like the i7-3770 typically have **lower memory bandwidth** (around **25-30 GB/s**), while the **Tesla T4** has a significantly higher bandwidth (300+ GB/s). This allows the T4 to handle large datasets and models more efficiently.
   * **Vectorization:** PyTorch’s **CPU backend** uses vectorization, but it can't match the throughput of a GPU like the **750 Ti**, which can accelerate the deep learning tasks by a significant margin.
3. **Pipeline Bottlenecks:**
   * **Data Loading:** With num\_workers=0 on the **CPU**, data loading may become a bottleneck, especially on the **i7-3770** when handling large datasets. This can affect training performance on the CPU, while **GPU** typically manages asynchronous data loading more efficiently.
   * **Preprocessing:** **SMOTE** (Synthetic Minority Over-sampling Technique) and other preprocessing steps may contribute significant overhead during model training. On the CPU, these steps may consume more time due to the **i7-3770’s fewer cores**, as opposed to the **GPU's parallel processing capabilities**.
   * **Model Complexity:** The model's complexity, with layers like **1024→512→256**, creates large matrix multiplications. On the **750 Ti GPU**, these matrix operations can be parallelized across thousands of cores, while the **i7-3770 CPU** can only handle them in sequential chunks, leading to slower performance.

**Summary**

The CPU implementation demonstrates the versatility of the neural network pipeline across different hardware platforms. While the Tesla T4 GPU implementation provides significant acceleration for this deep learning workload, the CPU version maintains identical functionality and accuracy.

The performance gap between CPU and GPU implementations highlights the importance of hardware-specific optimizations, particularly for computation-intensive deep learning models with large batch sizes.

Understanding the performance characteristics across both platforms enables more effective resource allocation, especially in scenarios where GPU availability may be limited or cost-constrained. With the suggested CPU-specific optimizations, the performance gap could potentially be narrowed for certain workloads or deployment scenarios.