Assignment 1 – Report:

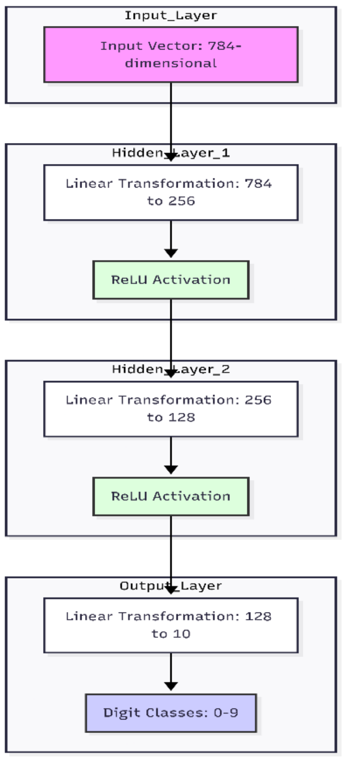
Mohammad
Student Id: 300480272
Email: fmoha077@uottawa.ca

1. Introduction:

The objective of this assignment was to implement and compare a multilayer perceptron classifier using two modern deep learning frameworks: JAX and PyTorch. The goal was not only to achieve competitive classification performance on the MNIST dataset, but also to analyze differences in programming paradigms, automatic differentiation mechanisms, and computational performance characteristics between the two frameworks.

The task focuses on supervised image classification using the **MNIST handwritten digit dataset**, which consists of 70,000 grayscale images of handwritten digits (0–9). Each image is 28×28 pixels and was flattened into a 784-dimensional input vector. The classification problem involves predicting one of 10 digit classes using a fully connected neural network.

A three-layer MLP architecture was implemented in both frameworks with identical structure to ensure fair comparison. The network consists of:


Input_Layer
Input Vector: 784-dimensional
Hidden_Layer_1
Linear Transformation: 784 to 256
ReLU Activation
Hidden_Layer_2
Linear Transformation: 256 to 128
ReLU Activation
Output_Layer
Linear Transformation: 128 to 10
Digit Classes: 0-9

- Input layer: 784-dimensional vector
- Hidden layer 1: 256 units with ReLU activation
- Hidden layer 2: 128 units with ReLU activation
- Output layer: 10 units corresponding to digit classes

Cross-entropy loss was used for optimization. In the JAX implementation, gradients were computed using jax.grad and parameters were updated using manually implemented stochastic gradient descent (SGD). In PyTorch, the model was implemented using nn.Module and trained using the Adam optimizer with automatic backpropagation via loss.backward().

To conduct a meaningful performance comparison, experiments were performed across multiple **batch sizes (64, 256, 1024)**. For each framework and batch size, the following performance metrics were measured:

- Training accuracy
- Validation accuracy
- Final test accuracy
- First epoch training time (including JIT compilation overhead for JAX)
- Steady-state epoch training time (second epoch timing)

The inclusion of both first epoch time and steady epoch time is critical for evaluating JAX, as its just-in-time (JIT) compilation introduces an initial overhead that does not reflect steady-state performance. Measuring both metrics allows us to distinguish compilation cost from true execution speed.

The experiments were conducted on a GPU-enabled system to ensure both frameworks could utilize hardware acceleration. By keeping the architecture, dataset, and hyperparameters consistent, this study isolates differences arising from framework design and execution strategy rather than model configuration.

2. Model Architecture and Experimental Setup

A. Network Architecture:

To ensure a fair comparison between JAX and PyTorch implementation , the same neural network architecture was implemented in both frameworks.

Each MNIST image (28x28 pixels) was flattened into a 784-dimensional input vector. The architecture consists of three densely connected layers.

- Input layer: 784 units
- Hidden Layer 1: 256 units with ReLU activation
- Hidden Layer 2: 128 units with ReLU activation
- Output layer: 10 units (corresponding to digit classes 0–9)

B. Data Preprocessing and Splitting

The MNIST dataset was loaded using standardised data loader. Pixel intensities were normalised to the range [0,1] by dividing by 255.0

The dataset was split into

- Training Set – 80%
- Validation Set – 10%
- Test Set – 10%

Stratified splitting was used to preserve class balance across all partitions. This ensures reliable generalization evaluation and prevents class imbalance from affecting performance metrics.

All experiments were performed using the same random seed per run to ensure reproducibility.

C. Optimization and Training Configuration

JAX Implementation

In the JAX implementation:

- Parameters were initialized manually using He initialization.
- The forward pass and loss computation were implemented as pure functions.
- Gradients were computed using jax.grad.
- Parameter updates were performed using manually implemented stochastic gradient descent (SGD).
- jax.jit was applied to the training step function to enable XLA compilation.

The use of jit transforms the training step into an optimized computation graph, allowing kernel fusion and reducing Python overhead during steady-state execution.

What JIT does is it creates a more optimized versions of the function we are transforming in this step we complete by pass the python interpreter ,and this is the reason why the functions are not meant to be active functions for example it should not contain any input() , print() functions inside as it will not be executed except the first run.

PyTorch Implementation

In the PyTorch implementation:

- The model was defined using nn.Module.
- Automatic differentiation was handled using loss.backward().
- The Adam optimizer was used for parameter updates.
- Training was performed using mini-batch gradient descent.

Unlike JAX, PyTorch executes operations eagerly without a compilation phase, meaning there is no explicit warm-up overhead before steady execution.

Meaning as we had seen a significant difference between first epoch and later epochs timings, this will not be the case for pytorch implementation.

D. Batch Size Experiments

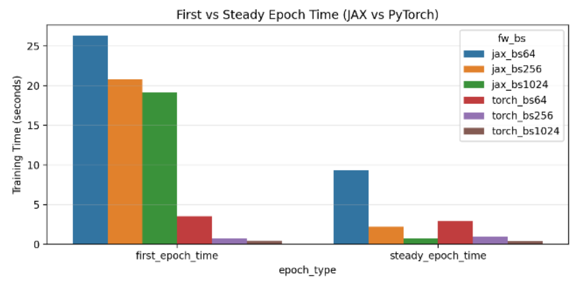
To evaluate scalability and computational efficiency, experiments were conducted using three different batch sizes:

- 64
- 256
- 1024

Batch size plays a significant role in GPU utilization and computational efficiency. Larger batch sizes generally improve throughput by increasing arithmetic intensity and reducing kernel launch overhead, while smaller batch sizes may offer improved generalization but incur higher per-iteration cost.

For each batch size and framework combination, the following metrics were recorded:

- Training loss and accuracy per iteration
- Validation accuracy per epoch
- Final test accuracy
- First epoch training time
- Steady-state epoch training time



The figure above shows the comparison between different batch sizes and the frameworks.

E. Performance Measurement Protocol

To evaluate computational performance fairly, two timing metrics were recorded:

1. **First Epoch Time**

- Includes JAX JIT compilation overhead.
- Reflects the cost of tracing and compiling the computation graph.

2. **Steady-State Epoch Time**

- Measured using the second epoch.
- Represents true execution speed after compilation is complete.

For PyTorch, both first and second epoch times were measured for consistency, although no compilation phase occurs.

All experiments were executed on a GPU-enabled system to ensure both frameworks utilized hardware acceleration. Synchronization calls were used where necessary to ensure accurate timing measurement.

3. JAX vs PyTorch Paradigm Comparison

Although JAX and PyTorch are both deep learning frameworks, they differ significantly in programming design, gradient computation and execution model. These differences influence both implementation style and performance behavior

A. Programming Model

JAX follows a **functional programming paradigm**, where parameters are immutable and explicitly passed to functions. The forward pass, loss computation, and gradient calculation are defined as pure functions. Parameter updates return new parameter objects rather than modifying state in-place. Random number generation is also explicit via PRNG key splitting, enforcing deterministic and side-effect-free computation.

In contrast, PyTorch follows an **object-oriented, stateful design**. Models inherit from nn.Module, and parameters are stored internally. Gradients are accumulated in-place, and calling optimizer.step() directly updates model parameters. Randomness and state management are handled implicitly.

Thus, JAX emphasizes functional purity and transformation compatibility, whereas PyTorch prioritizes ease of use and intuitive abstractions.

B. Automatic Differentiation

JAX computes gradients using jax.grad, which transforms a mathematical function into its gradient function. The gradient is computed without storing a persistent computational graph.

PyTorch uses a dynamic computation graph constructed during the forward pass. Calling loss.backward() traverses this graph to compute gradients. This eager approach simplifies debugging but does not naturally support global graph transformations.

C. Execution Model and JIT Compilation

The most important performance distinction arises from execution strategy.

PyTorch uses **eager execution**, meaning operations are executed immediately as Python statements. There is no compilation phase, resulting in consistent epoch timing.

JAX supports **just-in-time (JIT) compilation** through XLA. When jax.jit is applied, the first execution traces and compiles the computation graph. This introduces a noticeable first-epoch overhead. However, subsequent executions reuse the compiled graph, enabling kernel fusion and reduced Python overhead.

As a result:

- JAX typically has slower first epoch timing.
- JAX often achieves improved steady-state performance.
- PyTorch shows consistent but non-compiled execution behavior.

D. Summary of Differences

Aspect	**JAX**	**PyTorch**
Programming Style	Functional	Object-oriented
Parameter Updates	Immutable	In-place

Gradient Computation	Functional transformation	Dynamic graph
Execution	JIT compiled (XLA)	Eager
First Epoch Time	Slower (compilation)	Consistent
Steady-State	Optimized	Moderate

4. Performance Analysis and Results Interpretation

A. Experimental Results Overview.

Experiments were conducted for three batch sizes (64, 256, 1024) across both JAX and PyTorch implementations. For each configuration, the following were measured:

- First epoch training time
- Steady-state epoch training time (second epoch)
- Final test accuracy

The distinction between first epoch and steady epoch time is especially important for JAX due to JIT compilation overhead.

A summary of representative results is shown in the Tables section. Table no 1.

B. First Epoch Time: Compilation Overhead in JAX

The first epoch time for JAX is significantly larger than its steady-state epoch time. This behavior is expected due to JAX's JIT compilation process:

1. The training step is traced.
2. A computation graph is constructed.
3. XLA compiles the graph into optimized GPU kernels.
4. The compiled graph is cached for reuse.

This one-time compilation cost explains why the first epoch is slower in JAX compared to PyTorch. In contrast, PyTorch uses eager execution and does not incur a compilation phase, resulting in more consistent epoch timings.

C. Steady-State Epoch Time: Execution Efficiency

Once compiled, JAX exhibits significantly improved steady-state performance, particularly at larger batch sizes.

The performance improvement can be attributed to:

- Kernel fusion via XLA
- Reduced Python dispatch overhead
- Static graph optimization
- Improved GPU utilization

In PyTorch, operations are executed eagerly, meaning each operation is dispatched independently. Although backend kernels are efficient, the absence of global graph optimization limits kernel fusion opportunities.

As batch size increases, both frameworks benefit from better GPU utilization.

D. Effect of Batch Size on Accuracy

Final test accuracy showed a slight decline as batch size increased.

This behavior is consistent with known generalization dynamics:

- Smaller batch sizes introduce gradient noise.
- Gradient noise can act as implicit regularization.
- Larger batch sizes produce more stable but less exploratory updates.

Thus, while larger batch sizes improve computational efficiency, they may slightly reduce generalization performance.

E. Practical Interpretation

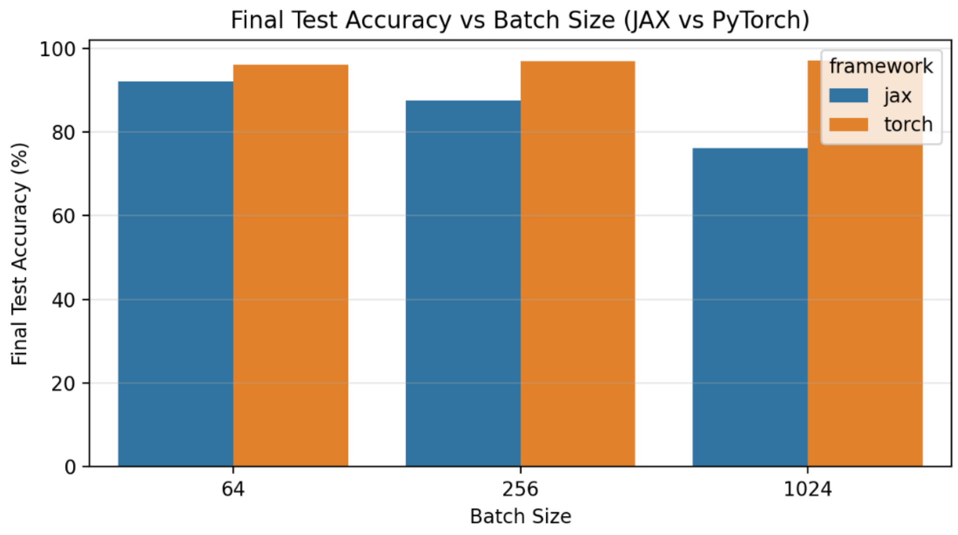
In practical scenarios:

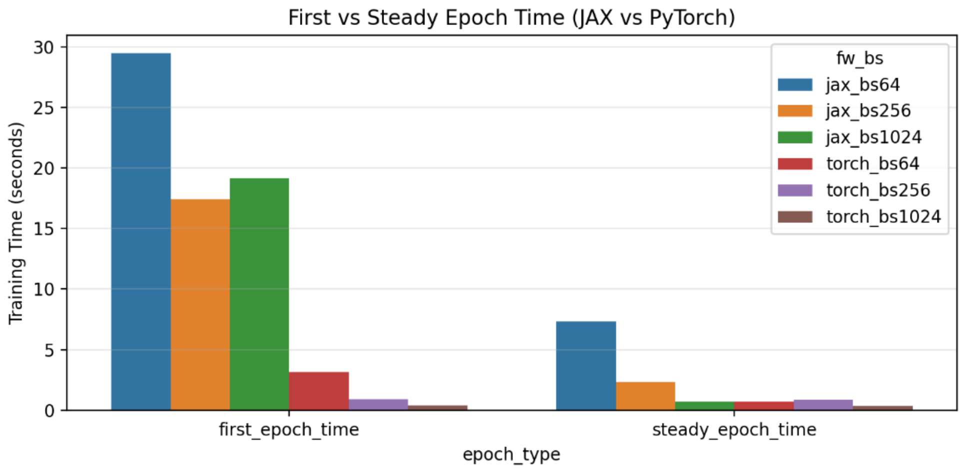
- If training runs are short or models change frequently, JAX compilation overhead may reduce efficiency.
- For long-running training or repeated execution, JAX's optimized steady-state performance can provide computational advantages.
- PyTorch provides a simpler development experience with predictable execution timing.

Tables:

Table : 1 , Epoch time comparison between different frameworks for different batch sizes.

framework	seed	batch_size	first_epoch_time_s	steady_epoch_time	final_test_acc	final_training_acc
jax	42	64	29.49099	7.323946	0.920857	92.70166
jax	42	256	17.40519	2.353691	0.876286	87.17477
jax	42	1024	19.13682	0.69011	0.762143	72.94952
torch	42	64	3.147912	0.72574	0.961286	96.50351
torch	42	256	0.935396	0.850996	0.97	97.56424
torch	42	1024	0.394275	0.369561	0.971571	97.54281





5. Code Workflow Overview

This section briefly summarizes the logical structure of the implementation to clarify how the experimental pipeline was organized.

A. Configuration and Initialization

The execution begins by loading the configuration file (config.yaml), which defines:

- Random seeds
- Batch sizes
- Learning rate
- Number of epochs
- Output paths

For each seed and batch size combination, experiments are executed independently to ensure reproducibility and controlled comparisons.

B. Data Preprocessing

The MNIST dataset is loaded and processed through the following steps:

1. **Dataset Loading** – MNIST images are loaded and flattened into 784-dimensional vectors.
2. **Normalization** – Pixel values are scaled to the range [0, 1].
3. **Train/Validation/Test Split** – Data is stratified to maintain class balance.
4. **One-Hot Encoding (JAX only)** – Labels are one-hot encoded for the JAX implementation, since the loss function expects explicit probability targets.

C. Framework-Specific Training Logic

After preprocessing, the workflow branches depending on the selected framework.

- **JAX Implementation**
- Parameters are initialized using jax.random.
- A pure functional train_step is defined.
- Gradients are computed using jax.grad.
- Parameters are updated using manual SGD.
- The training step is wrapped with jax.jit for compilation.
- The first epoch includes JIT compilation overhead.
- Subsequent epochs reflect steady-state performance.

D. PyTorch Implementation

- An nn.Module MLP is initialized.
- The Adam optimizer and CrossEntropyLoss are defined.
- A DataLoader handles batching and shuffling.
- The training loop performs:
  - Forward pass
  - Loss computation
  - loss.backward()
  - optimizer.step()

E. Performance Recording and Evaluation

For both frameworks:

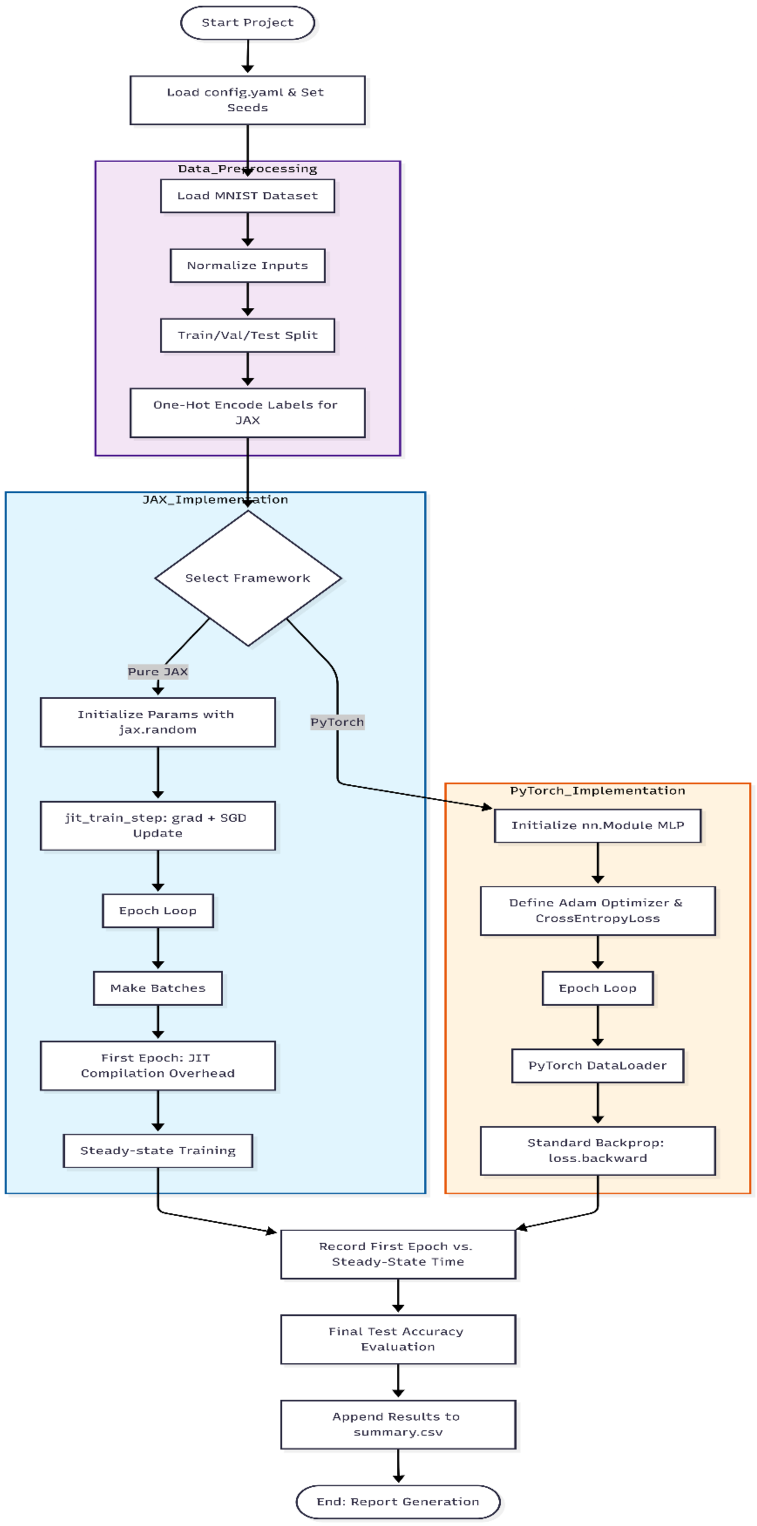
- First epoch training time is recorded.
- Steady-state epoch time (second epoch) is recorded.
- Training and validation accuracy are computed.
- Final test accuracy is evaluated.

F. Result Aggregation

After training:

- Iteration-level metrics are saved for learning curve analysis.
- Epoch-level metrics are saved for timing analysis.
- Visualization scripts generate:
  - Iteration vs seed plots
  - Iteration vs batch size plots
  - Mean ± SD and Mean ± SE bands
  - Timing comparison plots
  - Final accuracy comparison plots

This structured design separates configuration, training logic, and visualization, making the experimental workflow modular and reproducible.


Start Project
Load config.yaml & Set Seeds
Data_Preprocessing
Load MNIST Dataset
Normalize Inputs
Train/Val/Test Split
One-Hot Encode Labels for JAX
JAX_Implementation
Select Framework
Pure JAX
PyTorch
Initialize Params with jax.random
jit_train_step: grad + SGD Update
Epoch Loop
Make Batches
First Epoch: JIT Compilation Overhead
Steady-state Training
PyTorch_Implementation
Initialize nn.Module MLP
Define Adam Optimizer & CrossEntropyLoss
Epoch Loop
PyTorch DataLoader
Standard Backprop: loss.backward
Record First Epoch vs. Steady-State Time
Final Test Accuracy Evaluation
Append Results to summary.csv
End: Report Generation

The flowchart will help us visualize the code logic.