MACHINE LEARNING ON GPU

Lappeenranta-Lahti University of Technology LUT

0593633 Andry ANDRIAMANANJARA
001473021 Partha Durbar Hore

Examiners: Professor Lasse Lensi
Doctor Henri Petrow

CONTENTS

1 INTRODUCTION

1.1 Background

In the modern machine learning technology classification of data is very significant in various aspects. Graphics Processing Unit is a very powerful hardware has drawn revolution in machine learning algorithms by enabling of processing large dataset with higher speed and efficiency. In this project we focus on implementing two very effective and popular classification methods- k-nearest-neighbors (kNN) and multilayer perceptron (MLP) using the computational power of GPU. The KNN algorithm is a very frequently used non parametric supervised learning methodology which is a very robust and versatile tool. It works on finding the most similar instances within the feature space and making decision based on the majority class among K-nearest neighbors. The objective of implementing KNN algorithm in this project is to make use of the CUDA kernels and CuPy library, i.e write the code in GPU way [1, 2], to get an optimized result from the given dataset. By processing distances between the observations and performance evaluation across scenarios we try to optimize our implementation.

In the second implementation multilayer perceptron (MLP) [3] is one of the fundamental building blocks of modern feed forward Artificial Neural Network (ANN) which provides exceptional flexibility in modeling relationships within data. With its fully connected neurons and with activation function which is non-linear MLP is able to distinguish data that are not linearly separable. The primary goal of implementing MLP in this project is to model a MLP architecture using Pytorch and compare the performance in CPU and GPU platforms by computing loss over epochs.

1.2 Tasks

During the report elaboration, all tasks were discussed thoroughtly before proceeding, and before any change, a discussion between colleague were made in order to keep all members up to date in everything. The tasks were divided as follows :

- Andry : kNN implementation and results.
- Partha : MLP implementation and results.

2 METHODOLOGY

This section talks about the pre-processing techniques, in subsection 2.1, used during the training and testing sessions, the tools and techniques applied to approach K Nearest Neighbor (KNN) and Multilayer perceptron (MLP) methods that can be respectively seen in subsection 2.2 and subsection 2.3.

2.1 Pre-processing

Before applying machine learning techniques in a given dataset, it is important to check the dataset's behavior by looking at the highest, lowest and missing values in each columns. The purpose is to make sure the available dataset will give the most accurate results. To do so, re-scaling and treating missing values techniques, if necessary, are applied.

The rescaling technique that is selected for the project's implementation is the normalization technique by setting the mean to zero and the standard deviation to one that it is expressed as follow :

$$\hat{x_i} = \frac{x_i - \mu}{\sigma} \tag{1}$$

where $\hat{x_i}$, μ and σ are respectively the scaled value, the mean and standard deviation of the training dataset.

To make a machine learning approach efficient, dataset splitting is necessary. In this project the proportion of the training and testing dataset are 75% and 25% for kNN and 80% and 20% for MLP. The reason is to avoid the over-fitting issue during the training that is to say that the proportion gives a chances for the model to learn the input in more general way. The function `test_train_split` from `scikit-learn` learn library takes in charge of that tasks.

2.2 K-Nearest Neighbor (KNN)

The kNN is one of the classification algorithm to determine whether a given vector of features belongs to a class or not. It is mainly based on the distance between a given point with respective to the data point set. When the classification algorithm is hard to choose because of the non-linearity hidden within the dataset, kNN can be used as substitute

among the other algorithm.

It begins by computing the distance between a point, considered as a vector, and the remaining dataset as matrix, then shorts the distance from the smallest to the highest. Many distance are used for kNN and among them is the Euclidean distance which is used during this project. Once the distances are shorted, the algorithm proceed to take the nearest point to the new vector and check their classes and assign the new point into the majority class label available according to the number of the neighbor. The mathematical expression of the Euclidean distance is given by the following relation

$$d(v, S) = \sqrt{\sum_{s_i \in S} (v - s_i)^2} \tag{2}$$

where v denotes one vector and S is the set of all available vectors.

The implementation of the k-nearest neighbors (kNN) algorithm involves several detailed steps each crucial for ensuring its successful deployment and optimization. Below, a comprehensive breakdown of the steps followed for kNN implementation :

Data preparation : this step is for data loading and data normalization. `pandas` library is used for dataset loading, and normalization techniques combined with type conversion steps are managed by `cupy` library.

Distance implementation : It requires as inputs a vector and matrix then computes Euclidean distance on them. The implementation method is inspired by the existing matrix multiplication from the weekly assignment based on PyCuda Kernel. In our case, two kernels are used to achieve the goal where the first one computes the squared value of difference between vector and matrix, and the second one is dedicated for columns sum per row in a matrix only. Both kernels are written in `sourcemodule` and to access them as function, the method `.get_function` is used.

Value of k : the value of k that are going to be experienced are $1, 2, 3, 4, 5, 6$ and 7. After looking at their accuracies and execution time, some of k values are utilized for details checking by the help of `scikit-learn` library.

Prediction : after computing the distance between a point, represented as a vector, and the set of points represented as a matrix, shorting the distance values in ascending order is applied. Then fore each k, select the k first smallest distance and assign each of them a label, the function `argsort` from `cupy` manage the order. After, a majority vote among

the class label is executed by the help of `bincount().argmax()`, and the value will be assigned to the new point's label as predicted value.

Evaluation : the accuracy score, confusion matrix, and classification report are taken from `scikit-learn`.

2.3 Multilayer Perceptron (MLP)

The methodology of the project consists of data on boarding, model definition, training on CPU, training on GPU, Evaluation, Results, model accuracy for both CPU and GPU and a comparison in CPU and GPU calculation.

Data onboarding :

- The given data set was loaded using Pandas
- Features and labels were separated
- Features are normalized (we will add our method)
- Split the data set into training and testing sets
- Data conversion to Pytorch tensors

Model definition : MLP architecture model is defined using `nn.module` class in Pytorch. The MLP consists of two connected layers with ReLU activation.

Training on CPU : The model is initialized with loss function and optimizer. A pytorch DataLoader for efficient batch processing during the training. In CPU by training the model using a training loop over multiple epochs the model is implemented. The loss using specified criterion epoch is calculated and updated the model parameters. (Details after full code)

Training on GPU : Firstly PyTorch's GPU acceleration is utilized to train the MLP model on the GPU. The availability of GPU using `torch.cuda.is_available()` is checked and the model assigns data tensor to the GPU device accordingly.

Evaluation : The model performance is evaluated on the test set and the accuracy metrics

is updated by comparing the predicted labels with true label. The number of correct predictions and total sample is computed to calculate the accuracy.

Performance comparison : Comparison of the time taken by CPU and GPU implementations by calculating average time for each epoch is elucidated and visualized. Both CPU and GPU implementation accuracy is assessed for the test data set. The result is then analyzed to determine the impact of GPU acceleration on training efficiency and effectiveness.

3 EXPERIMENTS

This section presents the information the data information, exploratory data analysis, and the results after the training.

3.1 Exploratory Data Analysis (EDA)

As information from the exploration, the dataset, in Figure 1, contains 4000 rows and 11 columns where the last column is for the class label that presents 7 different class label ranging from number 3 to 9.

	0	1	2	3	4	5	6	\
0	0.233449	0.166667	0.381818	0.424594	0.164110	0.354839	0.250000	
1	0.160279	0.196078	0.263636	0.262181	0.176380	0.161290	0.346154	
2	0.083624	0.156863	0.300000	0.396752	0.191718	0.177419	0.355769	
3	0.097561	0.107843	0.390909	0.201856	0.019939	0.451613	0.288462	
4	0.073171	0.411765	0.445455	0.271462	0.007669	0.467742	0.413462	

	7	8	9	10
0	0.325581	0.166184	0.192771	6.0
1	0.255814	0.196453	0.301205	5.0
2	0.360465	0.226141	0.186747	6.0
3	0.372093	0.088490	0.216867	7.0
4	0.232558	0.128976	0.283133	5.0

Figure 1. First five line of the dataset.

The partition per class label is shown in table 1, revelating that the data points are predominantly concentrated in labels 5, 6 and 7. Consequently, the data exhibits imbalance, where the correct prediction are mostly associated in these three labels and the remaining labels can be difficult to predict.

After visualizing the whole dataset, shown in Figure 2, the minimum and maximum value are respectively zero and one. Columns 2, 5 and 7 present a huge value within the data whereas columns 4 and 8 do not present a remarkable pick or variation.

Inspecting the correlation between columns, presented in Figure 3, we found that the most correlated columns are 5 and 8 and the less correlated columns are in 1, 7 and 9.

Table 1. Repartition per class label.

Class	Number	Contribution (%)
3	16	0.40
4	140	3.35
5	1185	29.63
6	1773	44.33
7	735	18.38
8	147	3.68
9	4	0.10
Total	4000	100

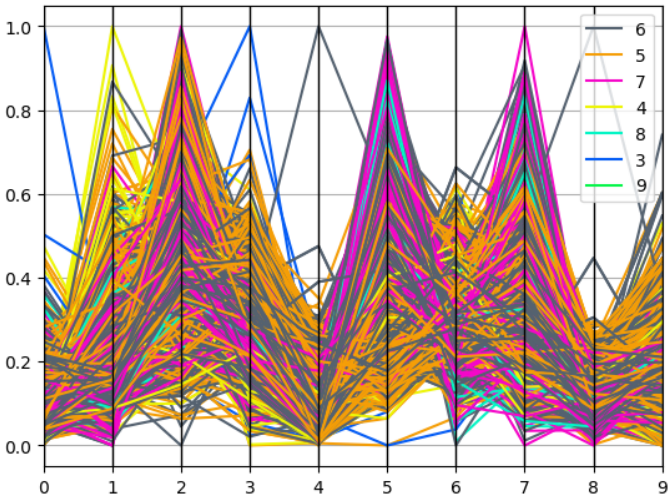


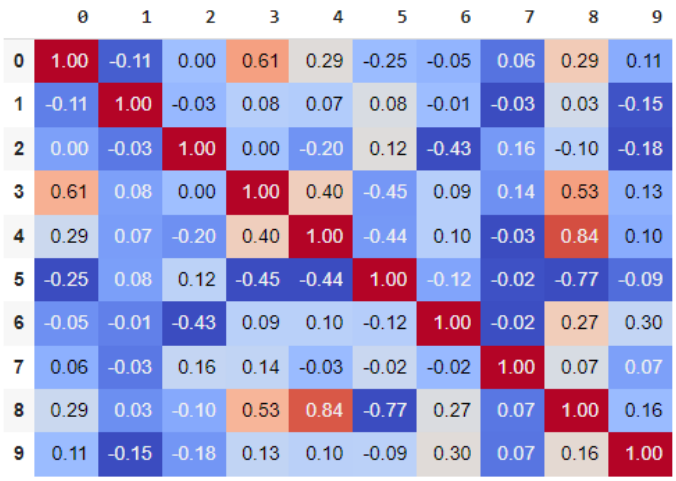
Figure 2. Parallel plot for all dataset.

3.2 Results

This section talks about results and observation during the experiments. For MLP implementation, datasets are splitted into 3200 (80%) and 800 (20%) for training and testing. For kNN, the model does not need any training session, but the train and test splits are respectively used for majority vote purpose and for measuring model performance, its split is as follow 3000 (75%) for trand 1000 (25%).

3.2.1 KNN

The results in table 2 and table 3, represent the accuracy and time execution according to the value of k before and after data normalization. Both tables show that for some value of



	0	1	2	3	4	5	6	7	8	9
0	1.00	-0.11	0.00	0.61	0.29	-0.25	-0.05	0.06	0.29	0.11
1	-0.11	1.00	-0.03	0.08	0.07	0.08	-0.01	-0.03	0.03	-0.15
2	0.00	-0.03	1.00	0.00	-0.20	0.12	-0.43	0.16	-0.10	-0.18
3	0.61	0.08	0.00	1.00	0.40	-0.45	0.09	0.14	0.53	0.13
4	0.29	0.07	-0.20	0.40	1.00	-0.44	0.10	-0.03	0.84	0.10
5	-0.25	0.08	0.12	-0.45	-0.44	1.00	-0.12	-0.02	-0.77	-0.09
6	-0.05	-0.01	-0.43	0.09	0.10	-0.12	1.00	-0.02	0.27	0.30
7	0.06	-0.03	0.16	0.14	-0.03	-0.02	-0.02	1.00	0.07	0.07
8	0.29	0.03	-0.10	0.53	0.84	-0.77	0.27	0.07	1.00	0.16
9	0.11	-0.15	-0.18	0.13	0.10	-0.09	0.30	0.07	0.16	1.00

Figure 3. Correlation.

k except for $k = 5$ and $k = 6$, there is an improvement within the accuracy after applying normalization techniques.

Table 2. Accuracy and GPU time execution for different value of k before normalization.

k	1	2	3	4	5	6	7
Accuracy	57.8	53.8	53.5	52.9	55.2	54.5	52.0
GPU execution time	4.27	4.39	3.81	3.78	4.76	3.72	3.75

Table 3. Accuracy and GPU time execution for different value of k after normalization.

k	1	2	3	4	5	6	7
Accuracy	59.4	56.2	52.7	52.0	54.7	53.0	54.9
GPU execution time	5.00	3.91	4.94	4.81	3.93	3.73	5.035

From table 3, the best candidates are :

- $k = 1$, $k = 2$ and $k = 5$, based on accuracy only.
- $k = 2$ and $k = 5$, based on accuracy and execution time.

For us, the second option would be fear because it takes into account both the execution time and the performance. Its detailed information are shown in table 4 and table 5.

Table 4. Classification report detail for $k = 2$.

Class	Precision	Recall	f1-score	Support
3	0.00	0.00	0.00	1
4	0.28	0.50	0.36	28
5	0.52	0.72	0.61	290
6	0.62	0.58	0.60	461
7	0.63	0.37	0.47	180
8	0.50	0.10	0.17	39
9	0.00	0.00	0.00	1
Accuracy	-	-	0.56	1000
Macro avg	0.36	0.33	0.31	1000
Weighted avg	0.58	0.56	0.55	1000

Table 5. Classification report detail for $k = 5$.

Class	Precision	Recall	f1-score	Support
3	0.00	0.00	0.00	1
4	0.23	0.21	0.22	28
5	0.54	0.62	0.58	290
6	0.59	0.62	0.60	461
7	0.49	0.39	0.44	180
8	0.36	0.10	0.16	39
9	0.00	0.00	0.00	1
Accuracy	-	-	0.55	1000
Macro avg	0.32	0.28	0.29	1000
Weighted avg	0.54	0.55	0.54	1000

Comparing both table 4 and table 5, it is clear that for many neighbors, the model performance tends to give same precision in each class except for class 3 and 9. Furthermore, during the majority vote, the predicted label can be influenced by the largest number of labels. However, with fewer neighbors, the model is able to predict 5 classes out 7 with precision more than 0.50 and accuracy of 56%.

We also experimented with dimensionality reduction techniques such as dimensionality value decomposition (SVD) and Kernel PCA (kPCA) to address non-linearity in the dataset. Initially, we utilized existing libraries for quick exploration. If any potential improvements were identified, we planned to implement them from scratch. However, despite our efforts, the accuracy did not improve.

As conclusion, after evaluating the GPU time execution, model accuracy and model details, it is determinated that the best candidate for the project is when $k = 2$.

3.2.2 MLP

The trained model, summarized in table 6, is evaluated we can observe accuracy of 53.0% in CPU and it drops a bit while computing in GPU and the accuracy is 53.5%. So we can observe a slight discrepancy on running the model in CPU and GPU. We plot the loss over epochs for both CPU and GPU, in Figure 4, to compare and evaluate the performances in using the both models. When we plot the CPU and GPU loss in a figure we can observe significance and similarity in steadily decreasing loss rate over epochs while implementing our MLP model. For both CPU and GPU implementation the loss graph shows similarity on loss over epochs. We have tried with 100 epochs to evaluate the performance comparison for implementation in both CPU and GPU.

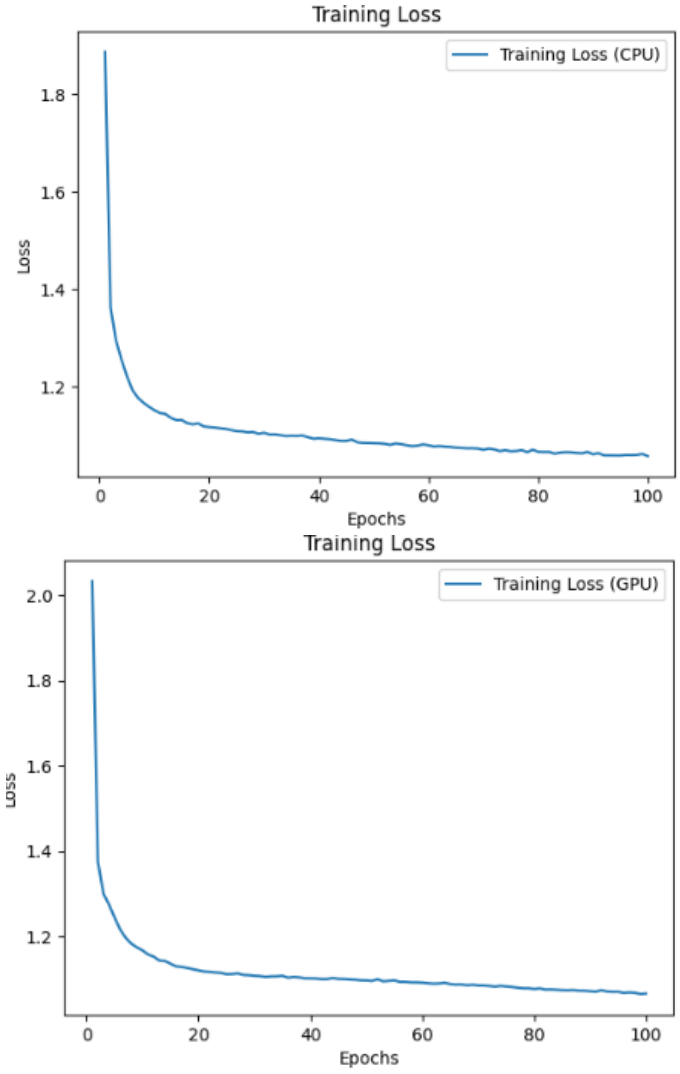


Figure 4. CPU loss (up) and GPU loss (down).

As each epoch is a complete pass through the training dataset. We can analyze that over epochs our model is updating its parameter to optimize the model by minimizing its loss. So it can be said, that the model is effectively increasing its performance after each pass through. With the accuracy and the loss graph in both CPU and GPU implementation it can be concluded that there's no over-fitting issue in this modeling as the model does not fit the training data too closely. When the training loss over each epoch is compared in a single graph we can observe similar representation although there is a very small discrepancy.

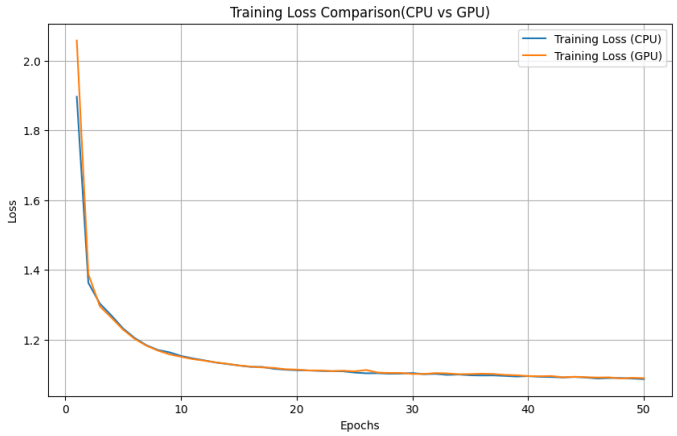


Figure 5. Training Loss comparison(CPU vs GPU).

The GPU loss varies from 1.09 to 2.05 where the CPU loss varies from 1.08 to 1.89. So there is a minute variation between loss while training in CPU vs Training in GPU. The graphical presentation(Figure 5) on comparison between Training Loss in CPU vs GPU elucidates the losses over epochs. The execution time taken by CPU for training is 5.50 seconds approximately. On the other hand the execution time in GPU for training 4.97 seconds. So training in GPU takes less time compared to training in CPU. The time taken for CPU execution in each epoch rarely varies and it stays in between 0.09 seconds to 0.14 seconds. On the other hand the time taken in GPU execution in each epoch varies from 0.09 to 0.12 seconds. So execution time in GPU is higher than execution time in CPU.

For both training and testing time of execution shows similarity in CPU and GPU.(Figure 6) For both training and testing execution time is similarly low for CPU and GPU respectively. Execution time for on test dataset in CPU is 0.015 seconds approximately and the execution time on test dataset in GPU is almost similar: 0.105 seconds.

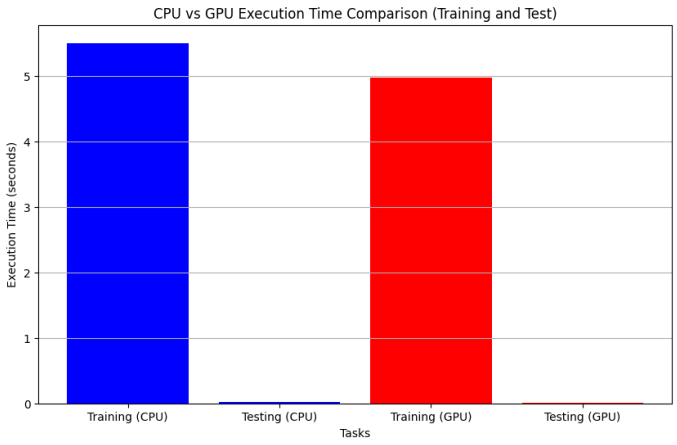


Figure 6. CPU vs GPU Execution Time Comparison (Training and Test)

Table 6. Accuracy(percentage) and time execution time(seconds) on test dataset in CPU and GPU

Class	CPU	GPU
Accuracy	52	51.125
Execution time(training)	5.50	4.97
Execution time(test)	0.0158	0.0150

The test Accuracy on CPU and GPU gives very similar results, as shown in table 7, in terms of percentage. After training the training dataset when the model is evaluated with the test data set the accuracy on both CPU and GPU is very similar which signifies correctness, parallelism in hardware utilization and optimization. It also indicates that leveraging GPU in modeling CPU shows significant computational capability of CPU. We have also studied by trying different numbers of neurons but input number of features:10; number of neurons in first hidden layer:64; number of neurons in second hidden layer:128; number of output classes: 10 provides optimum solution for MLP implementation.

Table 7. Number of neurons impact on accuracy and execution time

	Parameters	Parameters
Input	10	10
Layer1	64	32
Layer2	64	128
Accuracy(CPU)	50.28%	53%
Accuracy(GPU)	52%	53.5%

4 CONCLUSION

The project involved adapting kNN and MLP models in GPU. It was demonstrated that code written in GPU is faster than in CPU. However, access to GPU ressources from google colab is limited, and making optimization of working time is crucial. Python library such as PyCUDA, CuPy and PyTorch are very useful during the project. In GPU, the prediction time for kNN and MLP are 4 seconds and 0.015 seconds, and the accuracy are respectively 56% and 53.5%. In situations where class separability is challenging, kNN would be benefical.

REFERENCES

[1] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. *International Symposium on Computer Science and Computational Technology (ISCSCT)*, 01 2009.

[2] Polychronis Velentzas, Michael Vassilakopoulos, Antonio Corral, and Christos Antonopoulos. Gpu-based algorithms for processing the k nearest-neighbor query on spatial data using partitioning and concurrent kernel execution. *International Journal of Parallel Programming*, 51(6):275–308, 2023.

[3] Marius-Constantin Popescu, Valentina Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8, 07 2009.