Solving the Avicaching Game Faster and Better

Anmol Kabra, Yexiang Xue and Carla Gomes

Summer 2017

List of Functions, Symbols and Terms

Functions

$\mathbf{A} \cdot \mathbf{B}$	(dot product) Defined as $\mathbf{AB}^T$ as per convention
batch-multiply$(\cdot)$	Operates on $m \times n \times p$ and $m \times p \times q$ tensors to give a $m \times n \times q$ tensor.
reLU$(\cdot)$	Rectified Linear Unit; defined as reLU$(z) = \max(0, z)$
softmax$(\cdot)$	Defined as softmax$(z_i) = \frac{\exp(z_i)}{\sum_i \exp(z_i)}$

Symbols

J	Number of locations in the dataset
n_F	Number of features in the dataset $\mathbf{F}$ (length of $\mathbf{F}[v][u]$)
T	Number of time units for which data is available

Terms

CPU "set"	*All* operations done on the CPU
Epoch	One training/testing period; iteration
GPU "set"	*Only Matrix/Tensor* operations done on the GPU, rest on the CPU
LP	Linear Programming
LP Standard Format	Arrangement of objective function and constraints operated on by library LP solvers - minimize $[\mathbf{c}^T\mathbf{x}]$; subject to $[\mathbf{Ax} \leq \mathbf{b}, x_i \geq 0]$
Tensor	Multi-dimensional (usually more than 2 dimensions) array

Contents

Chapter 1

Introduction

Optimizing predictive models on datasets obtained from citizen-science projects can be computationally expensive as these datasets grow in size. Consequently, running models based on Multi-layered Neural Networks, Integer Programming and other optimization routines can become more computationally difficult as the number of parameters increase, despite using the faster Central Processing Units (CPUs) in the market. Incidentally, it becomes difficult for citizen-science projects, which often deal with large datasets, to scale if the organizers do not employ special processing units to run neural networks as optimization models, which require extensive tensor operations. One such special processing unit is the Graphical Processing Units (GPUs), which offer numerous cores to parallelize computation. These can often outperform CPUs in computing such predictive models if these models *heavily* rely on large-scale tensor operations (1; 2). By using GPUs over CPUs to accelerate computation on a citizen-science project, the model could achieve better optimization in less time, enabling the project to scale.

1.1 Avicaching

Part of the eBird project, which aims to "maximize the utility and accessibility of the vast numbers of bird observations made each year by recreational and professional bird watchers" (3), Avicaching is a incentive-driven game trying to homogenize the spatial distribution of citizens' (agents') observations (4; 5). Since the dataset of agents' observations in eBird is

geographically heterogeneous (concentrated in some places like cities and sparse in others), Avicaching homogenizes the observation set by placing rewards at and attracting agents to under-sampled locations (4). For the agents, collecting rewards increases their 'utility' (excitement, fun etc.), while for the organizers, a more homogeneous observation dataset means better sampling and higher confidence in using it for other models.

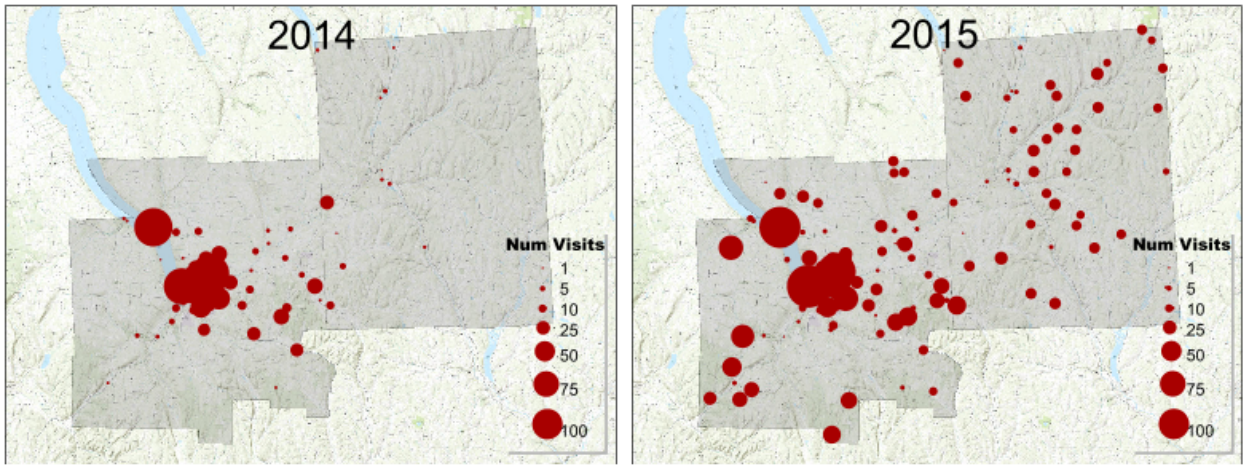


Figure 1.1: Avicaching 2014-15 Results in Tompkins and Cortland Counties, NY: With the previous model (4; 5; 3), Avicaching was able to attract 'eBirders' to under-sampled locations by distributing rewards over the location set.

To accomplish this task of specifying rewards at different locations based on the historical records of observations, Avicaching would learn how agents change their behavior when a certain sample of rewards were applied to the set of locations, and then redistribute rewards across the locations based on those learned parameters (5). This requirement naturally translates into a optimization problem, implemented using multi-layered neural networks and linear programming.

1.2 Important Questions

Although the previously devised solutions to Avicaching were conceptually effective (4; 5), using CPUs to solve Mixed Integer Programming and (even) shallow neural networks made the models impractical to scale. Solving the problems faster would have also allowed organizers to find better results (more optimized). These concerns, which form the pivot for our research,

are concisely described below.

1.2.1 Solving Faster

We were interested in using GPUs to run our optimization models because of their capability to accelerate problems based on large tensor operations (1; 2). Newer generation NVIDIA GPUs, equipped with thousands of CUDA (NVIDIA's parallel computing API) cores (6), could have empowered Avicaching's organizers to scale the game, if the underlying models were computed using simple arithmetic operations on tensors, rather than using conditional logic (see Section 1.3.2). Since even the faster CPUs - in the range of Intel Core i7 chipsets - are sequential in processing and do not provide as comparable parallel processing[1] as GPUs do, we sought to solve the problem much faster using GPUs. But **how fast could we do it?**

1.2.2 Better Results

The previously devised sub-model for learning parameters in agents' change of behavior on a fixed set of rewards delivered predictions that differed 26% from Ground Truth (5, Table 1). This model was then used to redistribute rewards in a budget. If we could get closer to the Ground Truth, i.e., better learn the parameters for the change, we could redistribute rewards with superior prediction/accuracy. Since the organizers require the *best* distribution of rewards, we will need a set of learned parameters that is closer to the Ground Truth (in terms of Normalized Mean Squared Error (5, Section 4.2)). In a gist, we aimed to **learn the parameters more suitably**, and **find the best allocation of rewards.**

1.2.3 Adjusting the Model's Features

Once the model starts delivering better results than the previously devised models, one thinks if some characteristics[2] of the model can be changed to get more preferable results (one could also build a better model). While a goal of "getting better results" is an unending struggle,

[1] CPUs often have multiple cores nowadays, but very few compared to what many GPUs have.
[2] Tinkering with hyper-parameters like learning rate or adding a weight regularizer.

there is a trade-off with practicality as these adjustments take time and computation power to test - and we didn't have unlimited resources. Therefore, we asked if one could **reasonably adjust the model's features to improve performance and optimization.**

1.3 Computation Using GPUs

The use of GPUs has changed drastically in the last decade - from rendering superior graphics to parallelizing floating point operations. Companies like NVIDIA are now providing General Purpose GPUs (GPGPUs) that are capable of executing parallel algorithms using thousands of cores and threads (6). Furthermore, by working with newly-developed parallel programming APIs like CUDA, one can handle a GPU's threads more efficiently and optimize an task's datapath (7). In the next sections, we briefly describe NVIDIA GPU's architecture, instruction set and best practices for optimization. Although we do not implement the CUDA back-end manually and use PyTorch's implementation (8), understanding the basics of the processor can be helpful for designing our models.

1.3.1 GPU Architecture

We describe the structure and abstractions of an NVIDIA GPU with Pascal architecture[3]. These devices are multi-threaded multi-processors for executing "fine-grained threads efficiently" (9, Appendix B.4). Unlike Personal Computer CPUs, which currently comprise 1-8 cores with fast and low-latency datapath, GPUs provide scalable parallel processing power with high throughput and latency[4] (9; 10). Informally, GPUs are often referred to as a collection of many 'dumb' cores, unlike the few 'smart' cores in a CPU. Nonetheless, GPUs are efficient in executing simple instructions in parallel, using hierarchies of cores, threads and memory.

[3]We use NVIDIA Quadro P4000 (Pascal architecture) for our tests.
[4]High latency is undesirable.

Core Organization

In most architectures by NVIDIA, many Scalar Processors (SP or CUDA cores) are clustered in a Streaming Multiprocessor (SM), which are then organized in Grid Processing Clusters (GPCs). A GPU can have several GPCs, as shown in Figure 1.2a (7; 1; 10).

The Streaming Multiprocessor (Figure 1.2b), which is the primary multi-threaded multiprocessor, houses a shared memory unit for all SPs along with registers and Special Function Units (SFUs), which can calculate specific mathematical functions in fewer clock cycles than threads. The SM is responsible for relaying instructions to threads (in SPs) and maintaining synchronized parallel computation (11; 10). GPCs and other hierarchies provide abstractions to memory access.

Memory Organization

Even the memory units are organized into hierarchies like multi-level caches in CPUs (Figure 1.2). In a SM, there are: dedicated local memory units for threads, register files shared by several SPs in a SM, and shared memory units for all SP Cores in a SM (11; 1).

The global internal memory of the GPU (like the RAM of a computer), which is housed separately around the clusters of SMs, stores the datasets for a program. Data is then distributed in the shared and local memory units once threads start executing an instruction (9, Appendix B). Moreover, similar to the multi-level cache structure in CPUs, GPUs may also have on-chip cache hierarchy (Figure 1.2), which enables threads to quickly access working data, thus increasing the performance. On a side note, GPUs' caches are often not big enough to hold full datasets as programs' often require large datasets, leading to more miss rates than caches in CPUs (9, Appendix B).

Thread Organization

According to NVIDIA, the parallel structure is obtained by organizing threads into a hierarchy ("grids" of "blocks" of "warps" of "threads"), where threads of a warp (an abstraction) execute in tandem per clock cycle (7; 10)(9, Appendix B). In other words, warps are the basic units of execution in a GPU, which receive single instructions from the controller, and forward

(a) Organization of Cores in NVIDIA Pascal GP100 GPU: In 6 GPCs and 60 SMs, 3840 SP or CUDA Cores (depicted by green blocks) are arranged (11; 1).

(b) Constituents of a Streaming Multiprocessor (11; 1).

Figure 1.2: The NVIDIA Pascal Architecture: A GPU, like a typical CPU, contains multiple hierarchies of memory. In addition, NVIDIA also builds a hierarchy of core organization.

them to their constituent threads for execution (7; 10). This abstraction allows the GPU to execute independent sub-tasks in parallel on a large scale(Figure 1.3), reducing the total runtime of the program.

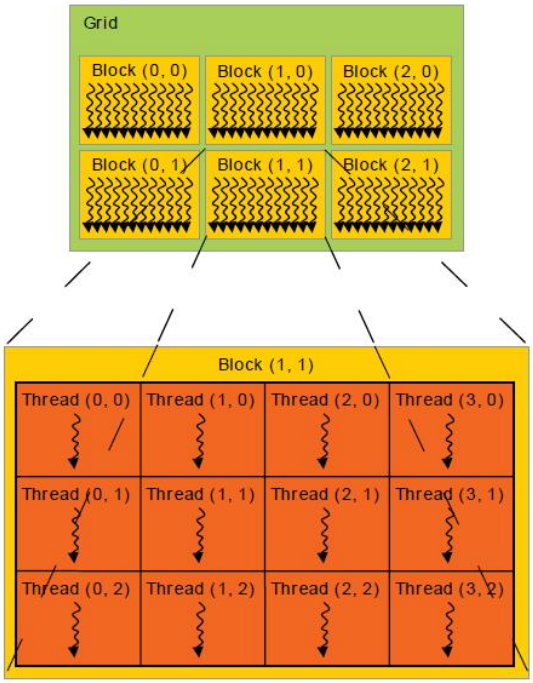


Figure 1.3: Thread Organization in a GPU: Warps are only an abstraction for identifying threads executing a single instruction; however, threads are physically organized into blocks and grids (7; 1)

NVIDIA calls this setup "Single-Instruction Multiple-Thread (SIMT)" (7) (9, Appendix B.4), where a single instruction is executed in lockstep by multiple threads in a warp (though allowed to branch and diverge). The differences between SIMT and Single-Instruction Multiple-Data (SIMD), a common feature in CPUs, are very subtle. While SIMD distributes data sequences into vectors to be executed by a single core/thread of the CPU (data-level parallelism), SIMT requires a particular thread to only operate on a scalar element of the dataset (9). In this way, having thousands of threads running in locksteps provides better throughput than doing vector operations per clock cycle. SIMT also allows programmers to write a program meant for a single, independent thread instead of managing vectors in their code, which promotes ease of use and programming (7, Chapter 4).

Instruction Set

The Instruction Set of a GPU is limited, which causes complex instructions to take several clock cycles. Even though a Special Functional Unit (SFU) computes complex functions (transcendental, reciprocal functions) in fewer clock cycles, their number is small compared to the commonplace threads (9, Appendix B). The instruction set for Pascal architecture contains basic operations to load and store memory, perform basic floating point and integer (add, subtract, multiply, divide, min, max, left-shift etc.) and binary logical operations (or, and, xor etc.), and execute jump and branching (12; 10). This comprises a very basic set of instructions unlike those in Intel CPUs, for example, which have coalesced and dedicated architecture for complex instructions[5] (9).

1.3.2 Avoiding Specific Types of Operations in GPUs

GPUs are not better than CPUs at many operations, unlike popular perception. With parallel processing comes concerns of synchronization delays, blocking instructions, program correctness etc.. Moreover, since GPUs are separate units of processors, usually connected to CPUs by PCIe lanes, back and forth memory transfer can cause slowdowns in overall performance. Even the GPUs low-level caches are not big enough to deliver hit rates as compared to CPUs. Often there exists a tradeoff when programming with GPUs, and one should take care to avoid such delays.

Branch and Diverge

Since NVIDIA GPUs deal with program correctness through inter-block barrier synchronization (synchronizing blocks of threads after some executed instructions), threads can often go out of sync, delaying the next instruction (9, Appendix B). This ultimately causes latency due to synchronization, which can aggregate and slowdown the performance of the program.

To avoid such delays and maximize throughput, one should elude from using extensive branching in the program. Using conditional checks can change a thread's datapath, which can possibly block other threads from proceeding at synchronization barriers. This behavior

[5]Intel x86 CPUs have Complex Instruction Set Computer (CISC) Design.

is informally referred to as 'branch and diverge' (Figure 1.4), which occurs when some threads in a warp follow a different datapath due to conditional branching, 'diverging' from the group. Now when the warp's threads are forced to converge, those diverging threads would take longer to complete the instruction, causing the non-diverging threads to wait. This radically decreases the throughput of the warp, which again diverges from the block, adding up the latency (9, Appendix B)(10).

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

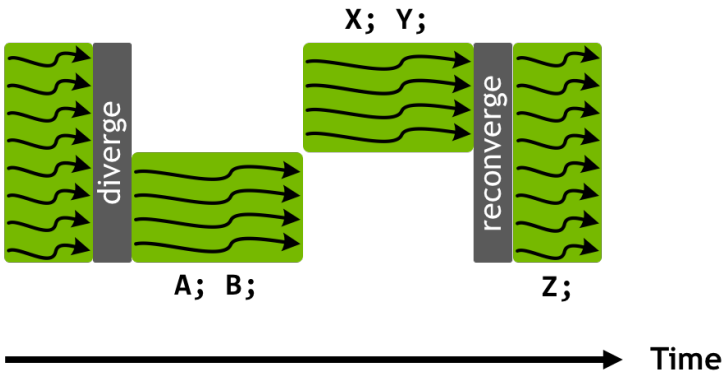


Figure 1.4: Decrease in GPU Throughput due to Branch and Diverge: In this case, while some threads execute `A, B`, others execute `X, Y`. If those sets of operations take unequal time, the threads could diverge from the group and decrease the throughput (1).

Memory Limitations

PCIe lanes are not as fast as on-chip cache access or even RAM access times (7; 1). As one would expect, transferring datasets back and forth between GPUs and CPUs can add up considerably. Often the limits of GPU's internal memory or required operations on a dedicated device can force the user to transfer datasets multiple times. Even so, one should look for optimizations that can reduce the number of transfers whenever possible, especially when large datasets are involved.

This suggestion is also applicable when transferring datasets in memory intra-GPU (between caches). Miss rates in a GPU's local caches are often high due to limited size, and data requests from higher-level caches can often be expensive.

1.3.3 GPUs' Strengths

Inferring from GPUs' weaknesses and architecture, we can say that GPUs are good at handling independent sub-tasks with less diverging sections. These independent sub-tasks can usually be clubbed into matrices and tensors, which the threads operate on in parallel. The fewer conditional branching we have in our algorithms, the better, maintaining the synchronization throughout execution.

Therefore, GPUs are particularly good at doing linear algebra (9; 1) since most arithmetic operations on matrix elements can be independently solved without branching. Avoiding branching often requires extensive dataset preprocessing; consequently, one should meticulously design the structure of the models' datasets. Tensor-based datasets are often required in machine learning problems, graphics rendering routines, graph-based algorithms etc., which GPUs can potentially accelerate (9).

Chapter 2

Problem Formulation

Our models strongly draw from the previous studies (5; 4), with modifications directed at reducing computation time, as well as getting better results. Since GPUs enable faster computation on tensors (see Section 1.3.3), both the Identification and the Pricing Problem are formulated as tensor-based 3-layered and 2-layered neural networks respectively using the PyTorch library (8). Recognizing that NVIDIA GPUs easily pair with PyTorch (8), accelerating tensor operations using NVIDIA's APIs – CUDA and cuDNN (7; 2), we aim to maximize parallel operations and minimize our models' computation time.

2.1 Identification Problem

As discussed in Chapter 1, the model should learn parameters that caused the change in agents' behavior when a certain set of rewards was applied to locations in the experiment region. Learning those parameters will help us understand how agents behave with a fixed reward distribution, and will enable organizers to redistribute rewards based on that behavior.

Specifically, given datasets $\mathbf{x_t}$ and $\mathbf{y_t}$ of agents' visit densities at time t, before and after the rewards $\mathbf{r_t}$ were placed respectively, we want to find the weights that caused the change from $\mathbf{x_t}$ to $\mathbf{y_t}$, considering possible influence from environmental factors $\mathbf{f}$ and distances between locations $\mathbf{D}$. Although the previous study proposed to learn a single set of weights $\mathbf{w}$ (5), our proposed model considers two sets of weights $\mathbf{w_1}$ and $\mathbf{w_2}$ as it may theoretically result

into higher accuracy and lower loss value. Mathematically, the model can be formulated as:

$$\underset{\mathbf{w_1},\mathbf{w_2}}{\text{minimize}} \quad Z_I(\mathbf{w_1},\mathbf{w_2}) = \sum_t (\omega_t(\mathbf{y_t} - \mathbf{P}(\mathbf{f},\mathbf{r_t};\mathbf{w_1},\mathbf{w_2})\mathbf{x_t}))^2 \tag{2.1}$$

where ω_t is a set of weights (not a learnable parameter) at time t capturing penalties relative to the priority of homogenizing different locations at time t. In other words, it highlights if the organizer wishes higher homogeneity at one time unit over another. Elements $p_{u,v}$ of $\mathbf{P}$ are given as ($\mathbf{\Theta_i}$ substituted for $\mathbf{w_i}_v^T$):

$$p_{u,v} = \frac{\exp(\mathbf{\Theta_2} \cdot \text{reLU}(\mathbf{\Theta_1} \cdot [d_{u,v}, \mathbf{f_u}, r_u]))}{\sum_{u'} \exp(\mathbf{\Theta_2} \cdot \text{reLU}(\mathbf{\Theta_1} \cdot [d_{u',v}, \mathbf{f_{u'}}, r_{u'}]))} = \frac{\exp(\Gamma_{u,v})}{\sum_{u'} \exp(\Gamma_{u',v})} = \text{softmax}(\Gamma_{u,v}) \tag{2.2}$$

To optimize the loss value $Z_I(\mathbf{w_1},\mathbf{w_2})$, the neural network learns the set of weights through multiple epochs of backpropagating the loss using gradient descent. Furthermore, the program preprocesses the dataset to avoid unnecessary sub-epoch iterations and to promote batch operations on tensors.

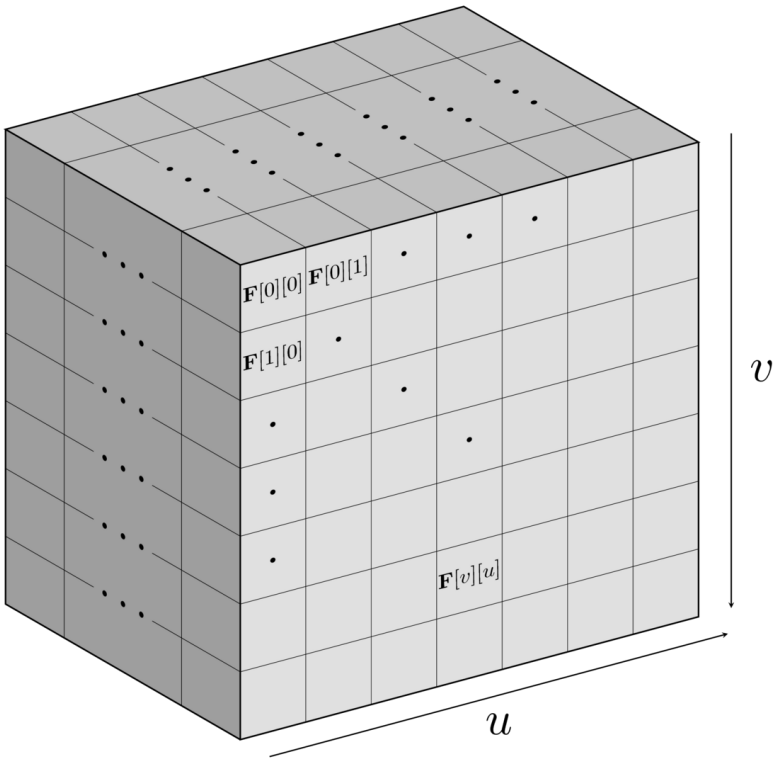
2.1.1 Structure of Input Dataset for Identifying Weights

Since preprocessing the dataset reduces data operations during model execution, the input dataset, comprising of distance between locations $\mathbf{D}$, environmental features $\mathbf{f}$ and given rewards $\mathbf{r_t}$ (all normalized), is built into a tensor (Figure 2.1a) such that operations can be performed on batches of slices $\mathbf{F}[v]$.

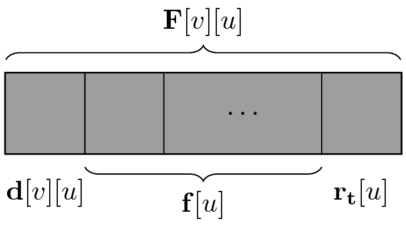
Another advantage of building the dataset as a tensor comes with the PyTorch library, which provides convenient handling and transfer of tensors residing on the Main Memory and GPUs' internal global memory (8). Algorithm 1 describes the steps to construct this dataset. During implementation, we preprocess the $\mathbf{F}$ dataset to reduce computation time during model runs (Appendix A.1.1).

2.1.2 Minimizing Loss for the Identification Problem

As shown in Figure 2.2, the neural network is made of 3 fully connected layers - the input layer, the hidden layer with rectified Linear Units (reLU), and the output layer with softmax($\cdot$)



(a) A Tensor representing the Input Dataset $\mathbf{F}$



(b) Contents of $\mathbf{F}[v][u]$: This vector (length n_F) contains quantified causes for the change in agents' behavior from $\mathbf{x_t}$ to $\mathbf{y_t}$

Figure 2.1: Visual representation of the Identification Problem's Input Dataset

Algorithm 1 Constructing the Input Dataset

function BUILD-DATASET($\mathbf{D}, \mathbf{f}, \mathbf{r_t}$)
  $\mathbf{D} \leftarrow$ NORMALIZE($\mathbf{D}$) ▷ $\mathbf{D}[u][v]$ is the distance between locations u and v
  $\mathbf{f} \leftarrow$ NORMALIZE($\mathbf{f}, axis = 0$) ▷ $\mathbf{f}[u]$ is a vector of env. features at location u
  $\mathbf{r_t} \leftarrow$ NORMALIZE($\mathbf{r_t}, axis = 0$) ▷ $\mathbf{r_t}[u]$ is the reward at location u
  for $v = 1, 2, \ldots, J$ **do**
    for $u = 1, 2, \ldots, J$ **do**
      $\mathbf{F}[v][u] \leftarrow [\mathbf{D}[v][u], \mathbf{f}[u], \mathbf{r_t}[u]]$ ▷ As depicted in Figure 2.1b
  return F

function units. The network can also be visualized as a stack of 1-dimensional layers (Figure 2.2b), with the softmax$(\cdot)$ calculated on the stack's output.

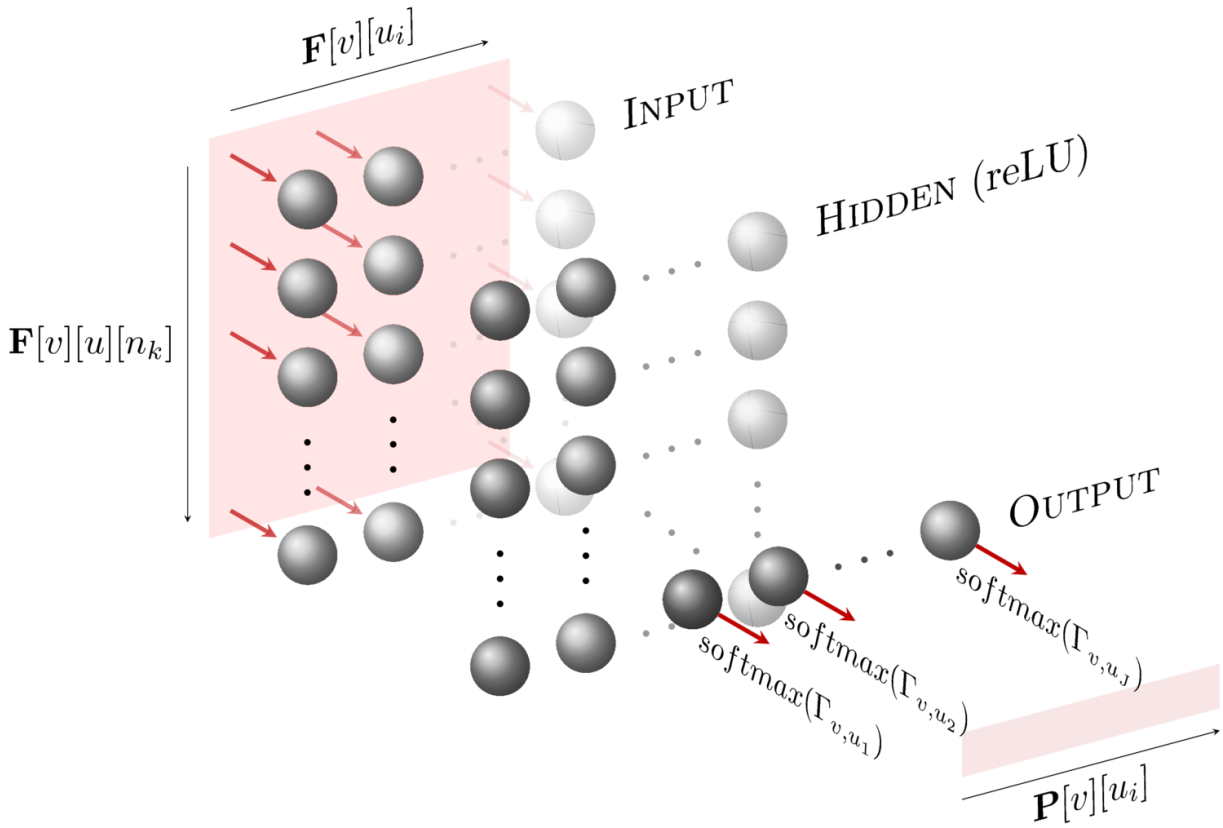
It is important to clarify that the network in Figure 2.2a, which takes in $\mathbf{F}[v]$ as shown, is a slice of the original network, which takes in the complete tensor $\mathbf{F}$ and computes the complete result $\mathbf{P}^T$ per iteration of t (implementation elaborated in Appendix A). In other words, the input and the hidden layers are 3-dimensional, and the output layer is 2-dimensional. Since it is difficult to visualize the complete network on paper, slices of the network are depicted in Figure 2.2a. Algorithm 2 details the steps for learning the parameters $\mathbf{w_1}$ and $\mathbf{w_2}$ based on Equations (2.1) and (2.2).

Algorithm 2 Algorithm for the Identification Problem

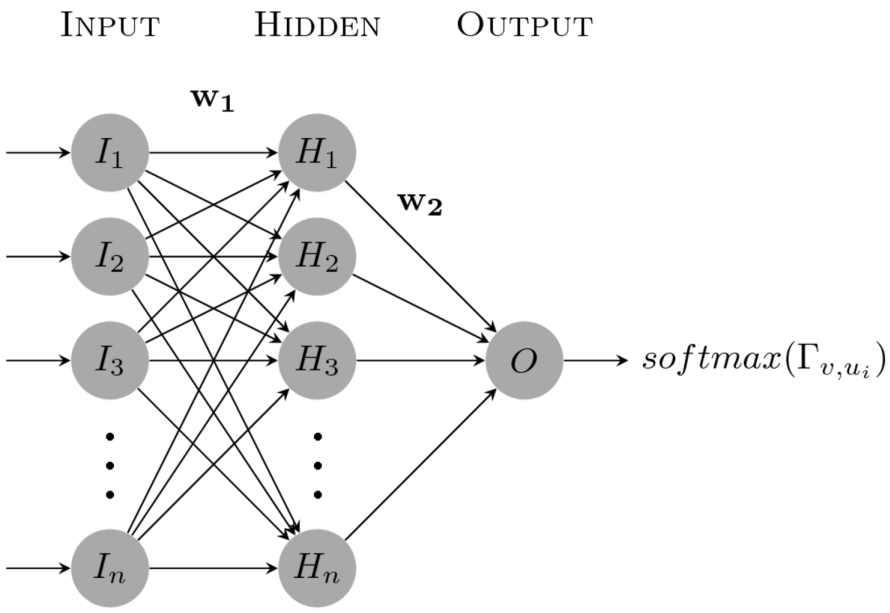
$\mathbf{w_1} \leftarrow$ RANDOM((J, n_F, n_F)) ▷ $\mathbf{w_1}$ has dimensions $J \times n_F \times n_F$
$\mathbf{w_2} \leftarrow$ RANDOM($(J, n_F, 1)$) ▷ $\mathbf{w_2}$ has dimensions $J \times n_F \times 1$
for $e = 1, 2, \ldots,$ Epochs **do**
    $loss \leftarrow 0$
    for $t = 1, 2, \ldots, T$ **do**
        $\mathbf{F} \leftarrow$ BUILD-DATASET$(\mathbf{D}, \mathbf{f}, \mathbf{r}[t])$ ▷ Defined in Algorithm 1
        $\mathbf{H} \leftarrow \text{reLU}($BATCH-MULTIPLY$(\mathbf{F}, \mathbf{w_1}))$
        $\mathbf{O} \leftarrow \text{softmax}($BATCH-MULTIPLY$(\mathbf{H}, \mathbf{w_2}))$
        $\mathbf{P} \leftarrow \mathbf{O}^T$
        $loss \leftarrow loss + (\omega[t](\mathbf{y}[t] - \mathbf{P} \cdot \mathbf{x}[t]))^2$
    GRADIENT-DESCENT$(loss, \mathbf{w_1}, \mathbf{w_2})$
    $\mathbf{w_1}, \mathbf{w_2} \leftarrow$ UPDATE-USING-GRADIENTS$(\mathbf{w_1}, \mathbf{w_2})$
    LOG-INFO$(e, loss)$

2.2 Pricing Problem

After learning the set of weights $\mathbf{w_1}$ and $\mathbf{w_2}$ highlighting the change in agents' behavior to collect observations, the Pricing Problem aims to redistribute rewards to the all locations such that the predicted behavior of agents influenced by the new set of rewards is homogeneous.



(a) 3-dimensional View of the Network Slice, Taking in $\mathbf{F}[v]$



(b) Side View of the Network: Output of one such cross-section is $p_{u_i,v}$

Figure 2.2: Neural Network Designed for the Identification Problem

Thus, given a budget of rewards $\mathcal{R}$, this optimization problem can be expressed as:

$$\begin{aligned} \underset{\mathbf{r}}{\text{minimize}} \quad & Z_P(\mathbf{r}) = \frac{1}{n}\|\mathbf{y} - \overline{\mathbf{y}}\| \\ \text{subject to} \quad & \mathbf{y} = \mathbf{P}(\mathbf{f}, \mathbf{r}; \mathbf{w_1}, \mathbf{w_2})\, \mathbf{x} \\ & \sum_i r_i \leq \mathcal{R} \\ & r_i \geq 0 \end{aligned} \tag{2.3}$$

where elements of $\mathbf{P}$ are defined as in Equation (2.2).

To allocate the rewards $\mathbf{r}$ optimally, the calculations for the pricing problem are akin to that for the Identification Problem (see Section 2.1). However, since only 1 set of rewards need to be optimized, we use an altered 2-layer (input and output layers) network instead of the 3-layered network used for the Identification Problem. While Equation (2.3) looks like a typical Linear Programming problem, only a part of the formulation uses LP to constrain the rewards. We calculate $\mathbf{P}$ using a 2-layered network that minimizes the loss function $Z_P(\mathbf{r})$ using gradient descent, and constrain the rewards using linear programming. Fundamental processes are described below, whereas specific implementation details, with code optimizations and more data preprocessing, are described in Appendix A.

2.2.1 Input Dataset for Finding Rewards

Since it is the set of rewards $\mathbf{r}$ that need to be optimized, they must serve as the "weights" of the network ("weights" here refer to the weighted edges of this network and not to the set of calculated weights $\mathbf{w_1}$ and $\mathbf{w_2}$). Therefore, the rewards $\mathbf{r}$ are no longer fed into the network but are its characteristic. Instead, the calculated weights $\mathbf{w_1}$ are fed into the network, and are "weighted" by the rewards.

The observation density datasets, $\mathbf{x}$ and $\mathbf{y}$, are also aggregated for all agents such that they give information in terms of locations u only. This is also why rewards vector $\mathbf{r}$ does not depend on t - we want a generalized set of rewards for all time t per location u. Therefore, the algorithm for constructing $\mathbf{F}$ (see Section 2.1.1) is same as Algorithm 1 but with a change – $\mathbf{r_t}$ replaced by $\mathbf{r}$.

2.2.2 Calculating Rewards

Algorithm 3 for finding $\mathbf{P}$ is very similar to Algorithm 2's first few steps but without any epochs of t. Also, since the model would predict $\mathbf{y}$, it does not need labels $\mathbf{y}$ as a dataset. Although Algorithm 3's logic flow may seem arcane, it is straight-forward - as displayed in Figure 2.3.

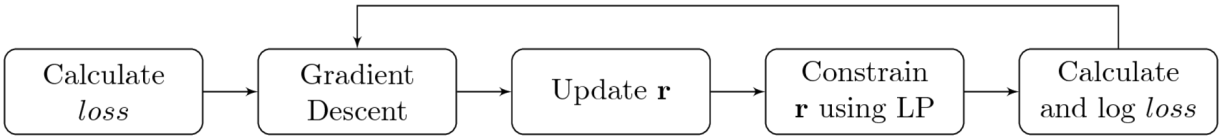


Figure 2.3: Logic Flow of Algorithm 3

Algorithm 3 Solving the Pricing Problem

function FORWARD($\mathbf{D}, \mathbf{f}, \mathbf{r}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{x}$)
    $\mathbf{F} \leftarrow$ BUILD-DATASET($\mathbf{D}, \mathbf{f}, \mathbf{r}$) ▷ Defined in Algorithm 1
    $\mathbf{O}_1 \leftarrow$ reLU(BATCH-MULTIPLY($\mathbf{F}, \mathbf{w_1}$))
    $\mathbf{O}_2 \leftarrow$ softmax(BATCH-MULTIPLY($\mathbf{O}_1, \mathbf{w_2}$))
    $\mathbf{P} \leftarrow \mathbf{O}_2^T$
    $\mathbf{y} \leftarrow \mathbf{P} \cdot \mathbf{x}$
    return $\|\mathbf{y} - \overline{\mathbf{y}}\|/J$

Main Script

$\mathbf{r} \leftarrow$ RANDOM((J)) ▷ $\mathbf{r}$ has dimensions J
$loss \leftarrow$ FORWARD($\mathbf{D}, \mathbf{f}, \mathbf{r}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{x}$)
for $e = 1, 2, \ldots,$ Epochs **do**
    GRADIENT-DESCENT($loss, \mathbf{r}$)
    $\mathbf{r} \leftarrow$ UPDATE-USING-GRADIENTS($\mathbf{r}$)
    $\mathbf{r} \leftarrow \text{LP}(\mathbf{r}, \mathcal{R})$ ▷ LP$(\cdot)$ explained in Section 2.2.3
    $loss \leftarrow$ FORWARD($\mathbf{D}, \mathbf{f}, \mathbf{r}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{x}$)
    LOG-BEST-REWARDS($loss, \mathbf{r}$) ▷ Record $\mathbf{r}$ with the lowest $loss$ yet

2.2.3 Constraining Rewards

After updating the rewards, the program constrains them using LP$(\cdot)$ such that $\sum_i r_i \leq \mathcal{R}$ and $r_i \geq 0$. To do so, the LP$(\cdot)$ finds another set of rewards $\mathbf{r'}$ such that the absolute difference between new and old rewards ($\sum_i |r'_i - r_i|$) is minimum. The mathematical formulation is given in Equation (2.4), which was implemented using SciPy's Optimize Module (13). Since the module supports a standard format for doing linear programming, Equation (2.5) is used (see Appendix A.1.2), which is mathematically equivalent to Equation (2.4).

$$\begin{aligned} \underset{\mathbf{r'}}{\text{minimize}} \quad & \sum_i |r'_i - r_i| \\ \text{subject to} \quad & \sum_i r'_i \leq \mathcal{R} \\ & r_i \geq 0 \end{aligned} \tag{2.4}$$

$$\begin{aligned} \underset{[\mathbf{r'},\mathbf{u}]}{\text{minimize}} \quad & \sum_i u_i \\ \text{subject to} \quad & r'_i - r_i \leq u_i \\ & r_i - r'_i \leq u_i \\ & \sum_i r'_i \leq \mathcal{R} \\ & r'_i, u_i \geq 0 \end{aligned} \tag{2.5}$$

We make a tradeoff by constraining rewards using LP instead of Mixed Integer Programming, which the previous study used (5). The tradeoff exists between decreasing the computation time and loosening integer constraints on each reward value. While model-learned integer rewards have worked in real-life testing also (5), we cannot say if allowing non-integer values in rewards would produce better results in real-life, corresponding to the model's prediction. In other words, we can't ensure the predicted rewards' effectiveness in ground testing before *actually* deploying them in the game. Nevertheless, we can estimate how good the predictions can be compared to several baseline indicators. These baseline sets are elaborated in Section 3.5.1.

Chapter 3

Experiment Specifications

Definition 1. *GPU Speedup: Ratio of model's execution time with GPU "set" to that with CPU "set". The script's data preprocessing runtime is ignored, but the time taken to transfer data from CPU to GPU is included in calculating GPU "set" time elapsed. (Speedup =* $\frac{CPU-time}{GPU-time+Transfer-time}$*)*

To test both our models, we conducted several tests for optimization and GPU Speedup. After initializing all parameters randomly (with specific seeds for reproduction and uniformity between tests), the models were run for 1000 or 10000 epochs depending on the complexity of the model.

3.1 Datasets

We conducted two types of tests: **optimization tests on original datasets** and **GPU Speedup tests on randomly generated datasets**. Data was loaded or generated as Floating Point 32 (FP32) units, but was stored with less precision (up to 15 significant figures) to reduce secondary memory usage.

For the GPU Speedup runs, two random datasets of 173 time units (T) were generated beforehand using NumPy (without any seed):

1. 116 locations - for Identification Problem
2. 232 locations - for Pricing Problem

We used more locations for the Pricing Problem's model because its 2-layered network was less complex and time consuming than the Identification Problem's 3-layered network. Additionally, we took this step after initially testing the Pricing Problem on 116 locations but failing to obtain a clear trend (more in Section 4.2.2). We believe that speedup tests on original datasets would give similar results, though we used randomly generated datasets because it was easier to scale and build random datasets of different batch-sizes for testing. The models were timed for the executed operations in a neural network and the LP, including transfer times of tensors between the RAM and GPU's internal memory. Time taken for preprocessing was ignored.

3.2 Test-Machine Configuration

Hardware specifications and software versions used for the experiments are listed in Table 3.1. Though we couldn't eliminate extraneous computing usage by background processes on the test-machine, we restricted background processes by switching off X (Graphical User Interface for Ubuntu OS) and performing tests in CLI (Command Line Interface), and ending user processes. Background processes and threads might give varying runtimes when other experiments are performed. However, one should obtain similar GPU Speedup results when repeating the experiments.

Table 3.1: Hardware Specifications and Software Versions Used for Experiments

Hardware	
Type	**Unit/Specs**
Desktop	Dell Precision Tower 3620
CPU	Intel Core i7-7700K[1]
RAM	16GB
GPU	NVIDIA Quadro P4000[2]

Software	
Library/Package	**Version**
Ubuntu OS	16.04.2 LTS x86_64
CUDA	8.0
cuDNN	5.1.10
MKL	2017.0.3
Python	2.7.13 (Anaconda)
PyTorch	0.1.12_2
NumPy	1.12.1
SciPy	0.19.0

[1]Hyper-threaded with 4 cores, 8 threads @ 4.20-4.50 GHz
[2]1792 CUDA Cores @ 1.2-1.5 GHz

GPU "set" and CPU "set" Clarification By GPU "set" we mean *distributing* operations in the scripts between CPU and GPU, while by CPU "set" we mean that the operations were executed *only* on the CPU. Since GPUs are inferior than CPUs at handling most operations other than simple arithmetic matrix ones due to parallelism (see Section 1.3), we used - and recommend using - both the CPU and the GPU in GPU "set" to handle operations each is superior at. However, since the models in Algorithms 2 and 3 (not the full scripts) are primarily arithmetic operations on tensors, it is clear that they were executed on the GPU when it was "set" and on the CPU when the CPU was "set". Other than this optimization, we did not specifically design any parallelized algorithm for either configurations, relying on the PyTorch's and NumPy-SciPy's inbuilt implementation.

3.3 Algorithm Choice

On the algorithm side, we used Adam's algorithm for GRADIENT-DESCENT($\cdot$), after testing performances of several algorithms[3] including but not limited to Stochastic Gradient Descent (SGD) (14), Adam's Algorithm (15) and Adagrad (16). In Chapter 4, we only discuss and show tests using the Adam's algorithm, since it was found to work best with both models over all test runs.

3.4 Running the Identification Problem's Model

3.4.1 Optimizing the Original Dataset

The 3-layered neural network was run for 10000 epochs on the original dataset, which was split 80:20 for training and testing sets, with different learning rates = $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$. Since we were aiming for optimization, we ran multiple tests (5 different seeds with each learning rate) of the model only with the GPU "set".

To compare this model's optimization results with other model structures, the previously studied 2-layered network (5) and a 4-layered neural network were used. The 4-layered network had another hidden layer with reLU, equivalent to the hidden layer in the current

[3]PyTorch lets you choose the corresponding function/module

3-layered network in Figure 2.2a. The results from the 2-layered network were obtained from the previous study, and those from the 4-layered network were attained on the same original dataset with same parameter values (learning rates, epochs etc.).

3.4.2 Testing GPU Speedup on the Random Dataset

After generating a random dataset and splitting it 80:20 for training and testing to predict performance on the original dataset, we ran our 3-layered model with different batch-sizes $T = 17, 51, 85, 129, 173$ ($J = 116$) and different seeds with both GPU and CPU "set", logging the elapsed time for model execution. The total time elapsed was averaged for a batch-size on a device, which were used to generate scatter/line plots (see Section 4.1.2).

3.5 Running the Pricing Problem's Model

3.5.1 Optimizing the Original Dataset

After obtaining the set of weights $\mathbf{w_1}$ and $\mathbf{w_2}$ optimized using different seeds, we tested to find the best rewards (with the lowest loss - Equation (2.3)) with random $\mathbf{r}$ initiation. To obtain the best rewards, the model was run on all sets of weights obtained from the Identification Problem for 1000 epochs with different learning rates. In search for the best rewards with the minimum loss, we took this approach:

1. Run differently seeded rewards on all sets of weights obtained from the Identification Problem, and identify a set of weights which performed better than the others (low Z_I - Equation (2.3)) on average. The learning rate was fixed to 10^{-3} in this case.

2. Use that set of weights to run a number of tests with varying seeds and learning rates $= \{10^{-2}, 5 \times 10^{-3}, 10^{-3}, 5 \times 10^{-4}, 10^{-4}, 5 \times 10^{-5}, 10^{-5}\}$, and choose the rewards which gave the lowest loss value Z_I anytime during execution[4].

Two sets of rewards were tested for loss values as baseline comparisons to our model - a randomly generated set, and another with elements inversely proportional to the number of

[4]This means that we selected the rewards before completion if the loss at that epoch was lower than that in the end.

visits at each location. While the former was a random baseline, the latter captured the idea of allocating higher rewards to relatively under-sampled locations. The best loss values were compared for all tests with the baselines.

3.5.2 Testing GPU Speedup on the Random Dataset

Initially, we ran the Pricing Problem's model with different batch-sizes $J = 11, 35, 55, 85, 116$ ($T = 173$) and different seeds with both GPU and CPU "set". Since we couldn't find a clear trend, we tested on more locations $J = 145, 174, 203, 232$.

We relied on SciPy's Optimize Module to solve our LP sub-problem (see Section 2.2.3) because PyTorch does not provide a GPU-accelerated Simplex LP solver. Since SciPy's implementation does not utilize the GPU, we expected the LP problem to be executed on the CPU and thus deliver equal runtimes in both GPU and CPU "set" configurations.

Chapter 4

Results

4.1 Identification Problem's Results

4.1.1 Optimization Results

Running the 3-layered network with the GPU "set" with different learning rates on different seeds (to calculate average performance of each learning rate) for 10000 epochs showed that the model was constantly giving lower loss values (Equation (2.1)) with learning rate = 10^{-3}.

We observed that higher learning rates ($> 10^{-2}$) could only decrease the loss function to a limit, after which the updates in the weights caused the loss function to oscillate and increase. This phenomena is exemplified in Figure 4.1, which are plots of different runs with the same seed but different learning rates. On the other side of 10^{-3}, lower learning rates took too long to train. Runtime for the model on 10000 epochs was 1232.56 seconds (average over 20 runs = 5 seeds $\times$ 4 learning rates), and models with learning rates less than 10^{-3} did not perform better than those with learning rate = 10^{-3} on *any* run. Although the decrease in test losses were constant for learning rates less than 10^{-3}, we feel that they may be computationally expensive and temporally inconvenient to train.

Observing that the average test loss values of learning rate = 10^{-3} is the lowest, we compare its results with the previous study's 2-layered network, historical data (5, Table 1), and the 4-layered network with learning rate = 10^{-3} (see Section 3.4.1). As depicted in Figure 4.2, our 3-layered neural network outperformed the 2-layered model by **0.14 units**

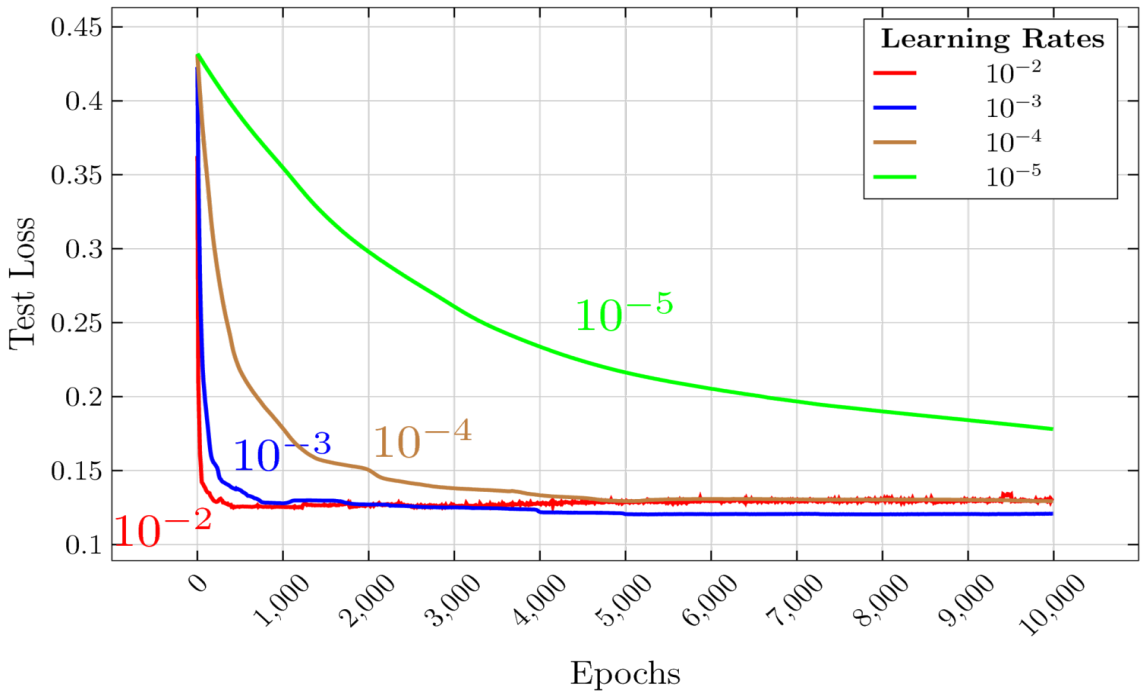


Figure 4.1: Test Loss Plots of Different Learning Rates to Find Weights: Learning rates lower than 10^{-3} couldn't reach the loss value attained by the model at learning rate = 10^{-3} even after 10000 epochs.

Table 4.1: Loss Values Calculated for Different Models for Identification Problem: For both the 3- and the 4-layered models, learning rate = 10^{-3} outperformed other learning rates. Consequently, that learning rate is used in comparison with other models in Figure 4.2.

Learning Rate	**Average Test Loss Values**	
	3-layered	4-layered
10^{-2}	0.168 ± 0.068	0.494 ± 0.083
10^{-3}	$\mathbf{0.119 \pm 0.016}$	$\mathbf{0.228 \pm 0.048}$
10^{-4}	0.151 ± 0.040	0.237 ± 0.067
10^{-5}	0.212 ± 0.040	0.320 ± 0.067

(14% more closer to Ground Truth − y) (5, Table 1), and also produced much better results (12% more closer to Ground Truth) than the 4-layered model.

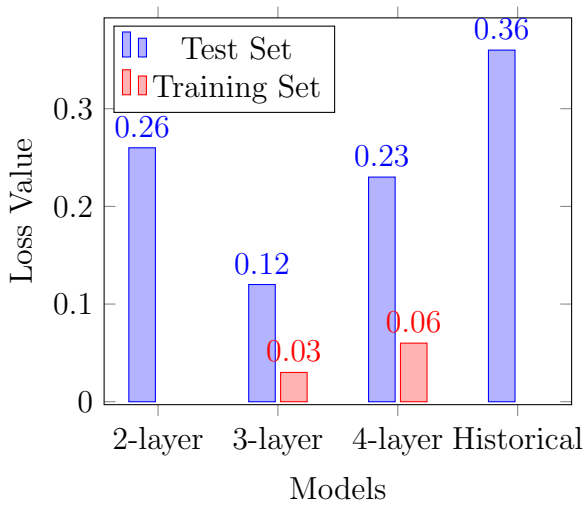


Figure 4.2: Comparison of Loss Values from Different Models of the Identification Problem: Loss values for the training set are inevitably lower than that for the test set, which is the basis for comparison.

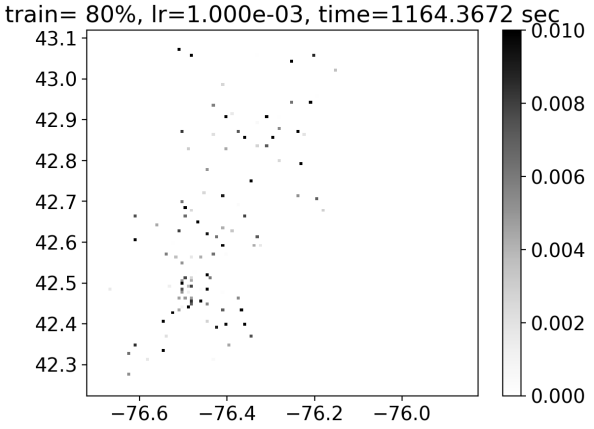
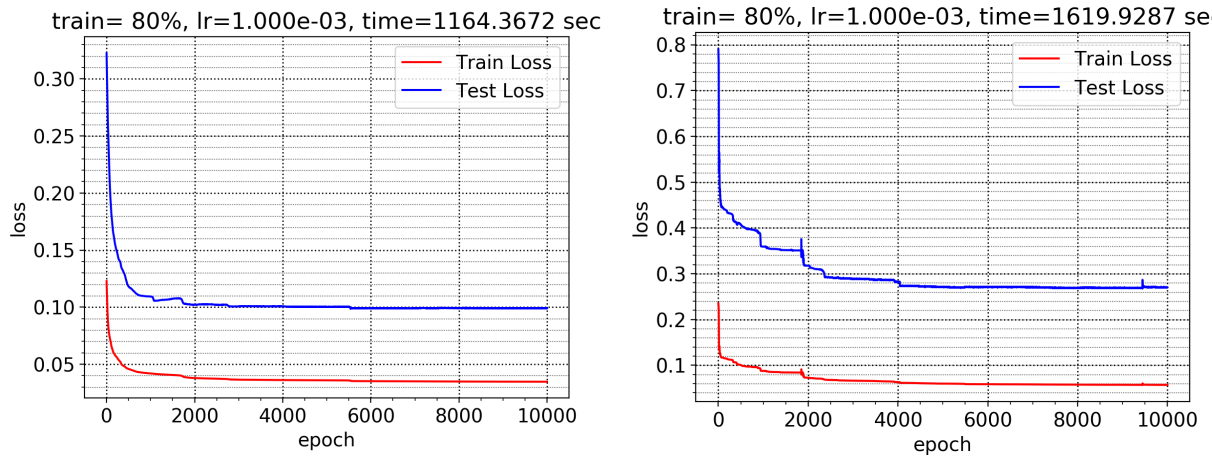


Figure 4.3: Predicted Probabilities of Agents Visiting Each Location Plotted on a Map (Latitude, Longitude) Representing Tompkins and Cortland Counties, NY: Dark dots represent high prediction of visits. This can be compared to the plots for the 2-layered network and other models (5, Figure 3).

We also generated the predicted probabilities of the agents in the Test Set, visiting each location (**Px**), and plotted it onto maps marked by the locations' latitudes and longitudes. Figure 4.3 shows such a plot generated by the 3-layered network, where each dot represents a location.

To check if the model was overfitting at learning rate $= 10^{-3}$, we plotted loss values at the end of each epoch for all tests. Although there remained $\approx 9\%$ difference (0.09 loss units) in the values of training and testing set, the 3-layered model was not drastically overfitting as an average *end* difference of $8.76 \pm 1.59\%$ persisted for many epochs, instead of increasing and tuning more to the training set. Even though one would expect overfitting with more tuned parameters, the 4-layered model also produced an average *end* difference of $16.77 \pm 4.73\%$, which persisted during the run. Figure 4.4 shows the results of a randomly selected experiment with plots of loss values at each epoch for the both networks.



(a) Plot for 3-layered Model

(b) Plot for 4-layered Model // todo

Figure 4.4: Train and Test Loss Values' Plots of Different Models: Both networks learn the set of weights quickly, as displayed in the steep descent in loss values before ≈ 1000 epochs. This quick learning is due to the choice of GRADIENT-DESCENT$(\cdot)$ function – Adam's algorithm (15). Other algorithms like SGD (14) and Adagrad (16) learn relatively slowly.

4.1.2 GPU Speedup Results

Running on batches of sizes $T = 17, 51, 85, 129, 173$ on a randomly generated dataset for 1000 epochs with both GPU and CPU "set" separately, we obtained information on full execution runtimes (combining training and testing runtimes). The average results (over 3 seeds for each batch) are plotted in Figure 4.5, which show promising GPU Speedup figures for any batch size; the GPU Speedup averaged over all tested batch-sizes is $\mathbf{9.06 \pm 0.45}$.

4.2 Pricing Problem's Results

4.2.1 Optimization Results

Taking the approach mentioned in Section 3.5.1, we obtained consistent loss values for each set of weights (even with differently seeded rewards). The best performing set of weights was set-2, using which the average loss value for the Pricing Problem hovered around 0.0079%. Next, running differently seeded rewards with different learning rates on set-2 of weights, we obtained the lowest loss value of **0.0069%**.

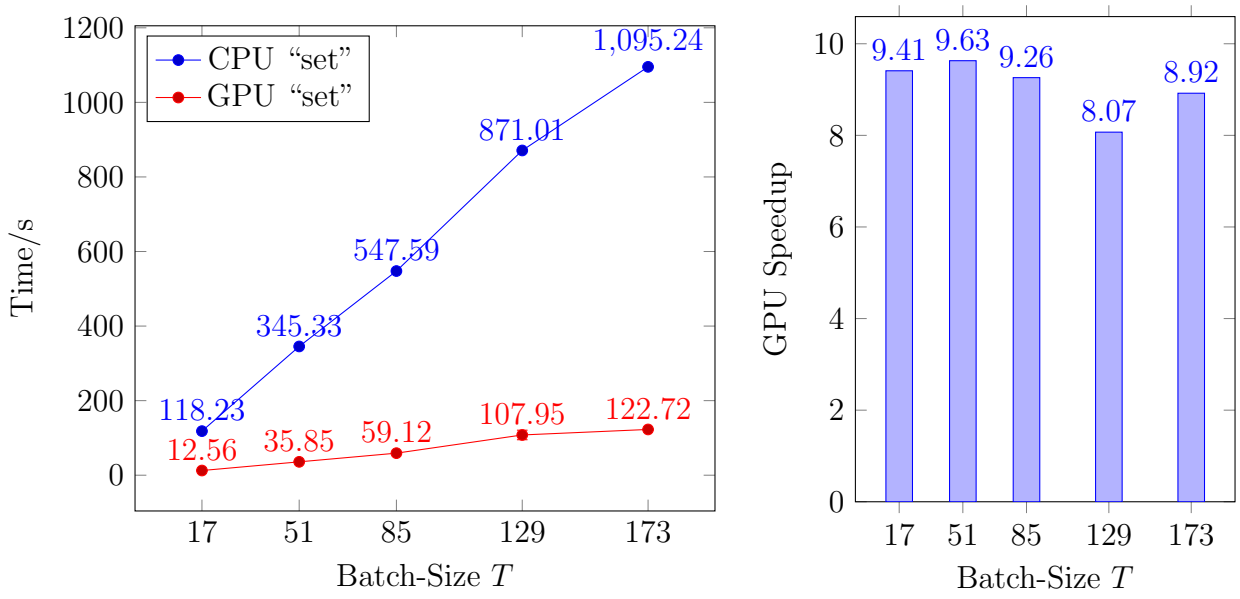


Figure 4.5: Finding Weights – Execution Times of Different Batch-Sizes T with GPU and CPU "set" Separately: The GPU delivers faster computation than CPU even as the datasets' sizes grow – the average speedup is 9.06 ± 0.45. Also, the error bars are indiscernible because they are too small ($< 1\%$)

.

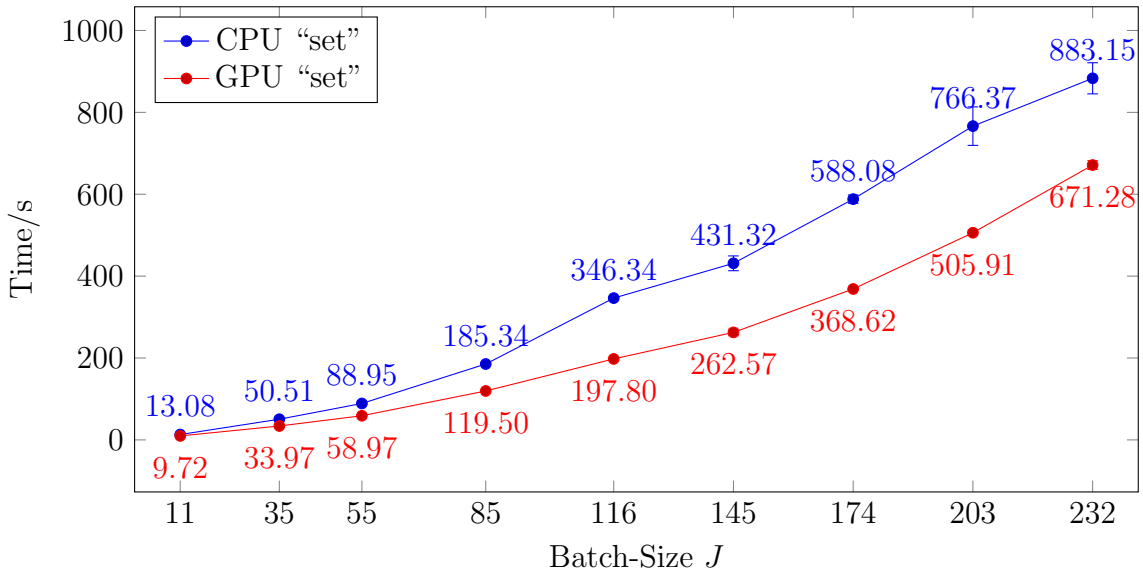
Compared to the proportional reward distribution (loss values calculated using set-2 of weights), our model's set of rewards produced loss value $\approx$ **3 times** lower. Table 4.2 lists the best loss values obtained on each type of reward allocation (model's predicted, random and proportional – Section 3.5.1).

Table 4.2: Loss Values Calculated from Different Sets of Rewards: The values are small because the loss function $Z_P(\mathbf{r})$ (Equation (2.3)) is averaged over the number of locations

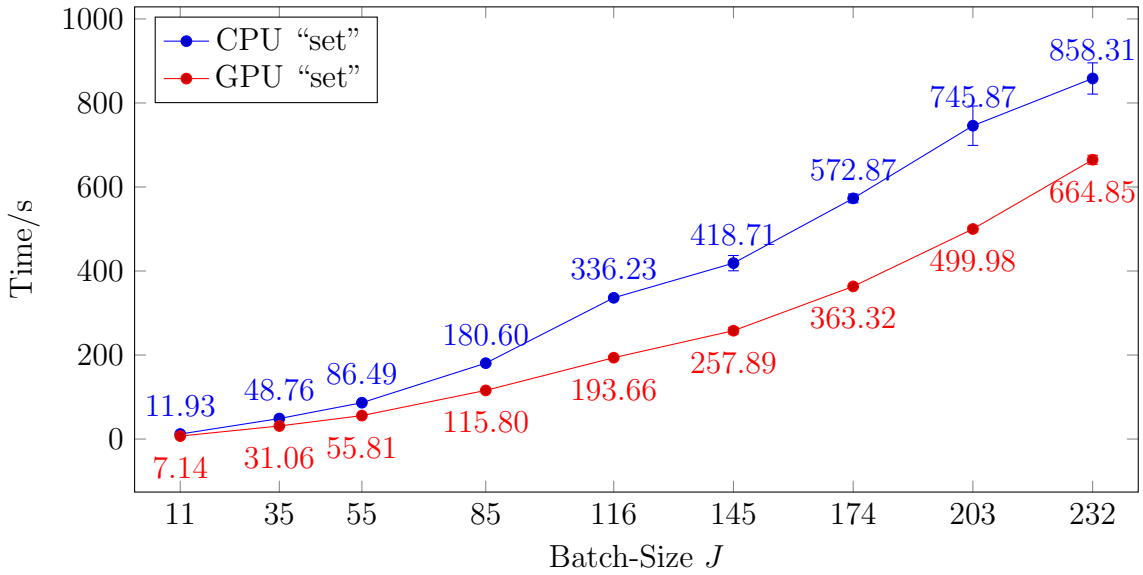
Rewards Obtained From	**Best Loss Values (In %)**
Model's Prediction	0.0069
Random Initialization	0.0331
Proportional Distribution	0.0235

4.2.2 GPU Speedup Results

After running on different batch-sizes $J = 11, 35, 55, 85, 116, 145, 174, 203, 232$, we **did not observe drastic GPU speedup** for the full model. Figure 4.6a shows the Speedup trend: GPU Speedup for the full model was a mere **1.53 $\pm$ 0.10**.



(a) Time Taken by the Full Model: GPU Speedup over all batch-sizes is only **1.53** ± **0.10**.



(b) Time taken by the LP: GPU Speedup over all batch-sizes is only **1.56** ± **0.08**. As discussed in Section 4.2.2 and Appendix B, we shouldn't have witnessed this speedup as both configurations' LPs were computed on the CPU. Hence, we expected the GPU Speedup value for LP to be ≈ 1.

Figure 4.6: Finding Rewards – Execution Times of Different Batch-Sizes J with GPU and CPU "set" Separately

As the the low GPU Speedup was disappointing, we looked for operations that were causing the program to slow down on the GPU. Since the 2-layered network was quite small, we suspected the LP problem (Equations (2.4) and (2.5)) to influence the runtimes. Thus, we recorded execution times for both the neural network and the LP separately. As we suspected, the LP did impact the runtime more than the neural networks did, and accounted for $\approx 90\%$ of the total model runtime (Figure 4.6). However, the GPU Speedup in LP runtime was unusual.

Strange GPU Speedup in LP Computation

The Speedup is exceptional as we intentionally transferred the needed matrices/tensors to the CPU for solving the LP using Simplex. Since SciPy's Optimize Module uses a single CPU core and does not utilize the GPU, we expected similar runtimes for both configurations. Our efforts to determine the reasons are collected in Appendix B instead of digressing here. As stated later in this section, we disregard the old GPU Speedup results for finding rewards and use the more suited ones.

We found that the LP in CPU "set" was slower than normal because of latency in thread synchronization[1] (Appendix B.4). Since Algorithm 3 performs operations sequentially (Calculate loss $\rightarrow$ Gradient-Descent and Update $\mathbf{r} \rightarrow$ Constrain using LP), we suggest these possibilities:

1. The 2-layered network is parallelized but `scipy.optimize`'s LP solver is not. Since both sub-problems are solved by independent frameworks, we could not thread the whole algorithm without implementing OpenMP (17) back-end manually and allowing multiple copies of the script to run simultaneously. Currently, the LP problem acts as a shared resource, requiring all network threads to synchronize before processing, causing time lag before all elements/matrices are available.

2. The simultaneously running threads downgrade the LP solver's performance. We do not endorse this possibility because the CPU resources are independent when using multi-processing over CPU's cores. We checked if the same script running simultaneously on

[1]Pytorch uses OpenMP (8; 17) when the CPU is "set", which threads the 2-layered network.

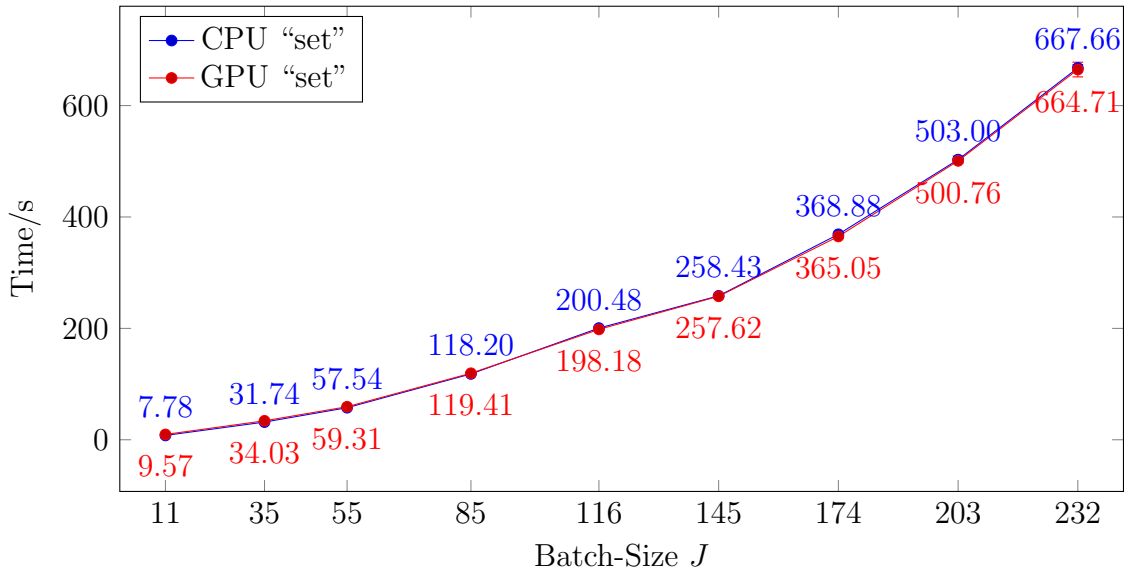
another core hampered the performance of the original copy, but did not witness any dependence.

We believe the first possibility to be the reason (elaborated in Appendix B). While one might contend the lack of this behavior in GPU "set" even though it needs thread synchronization before starting the LP solver, we reiterate that we intentionally impose a synchronization barrier when transferring the matrices to the CPU after the network has been executed, adding the time elapsed in the network's time logs. This is not case with CPU "set" because the network remains in execution in other cores while the LP has started. Therefore, when we restrict the script's access to a single CPU core, we do not witness any GPU Speedup in LP runtime. This only affects CPU "set" performance because the GPU "set" configuration independently executes the network on the GPU.

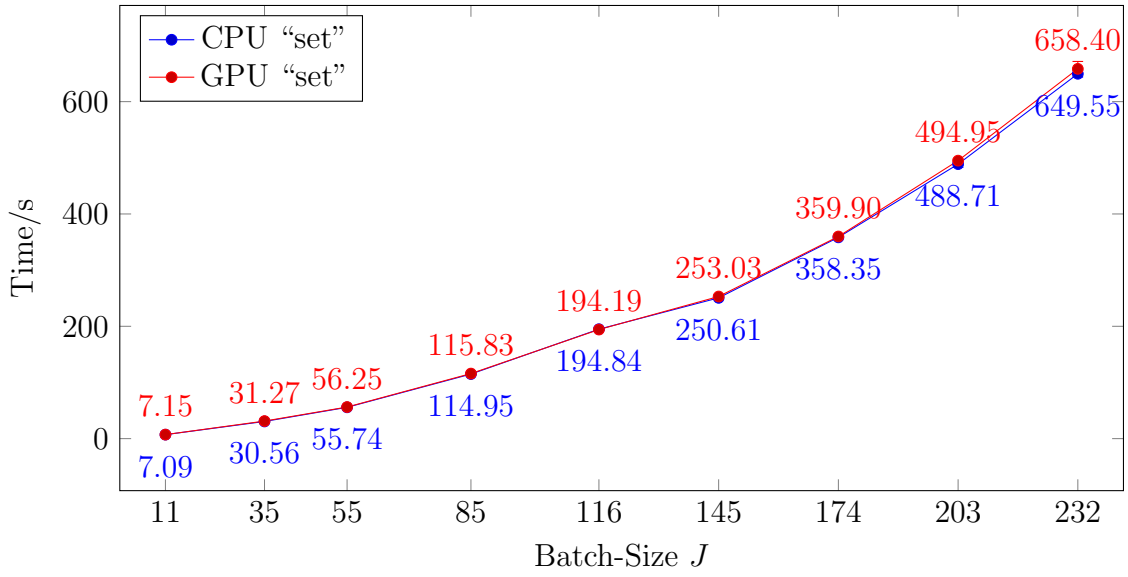
Final Results

The results after restricting script's access to a single CPU cores are plotted in Figure 4.7. We prefer using the results after restricting access (over previous results – Figure 4.6), as it gives the better performance for CPU "set". The time elapsed for the Neural Network includes time taken to transfer tensors to and from the GPU, which results in overhead – as seen in higher runtimes for smaller datasets (lower batch-size) in Figures 4.7c and 4.7f. However, as the batch-size increases, we see the computation dominating over the transfer time, resulting in higher CPU "set" runtimes; the GPU "set" runtimes almost grow linearly for the tested batch-sizes.

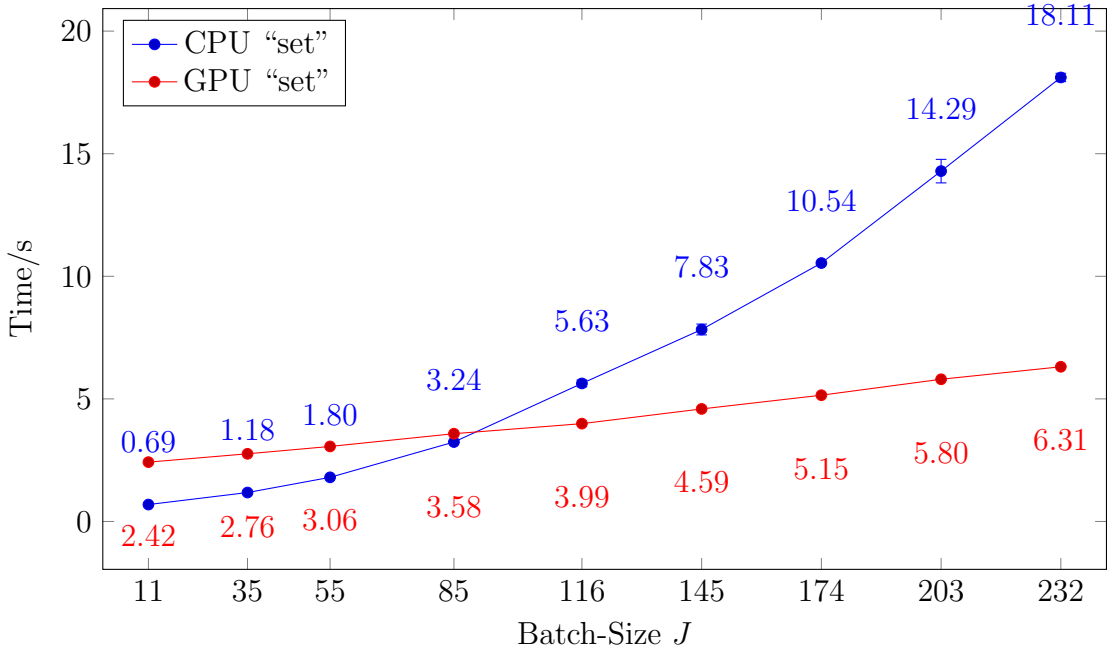
Moreover, the LP subproblem highly impacts the full model, accounting for more than 94% of the total runtime. Therefore, even though the Neural Network gets sped up at bigger datasets, the GPU Speedup is not reflected in the full model.



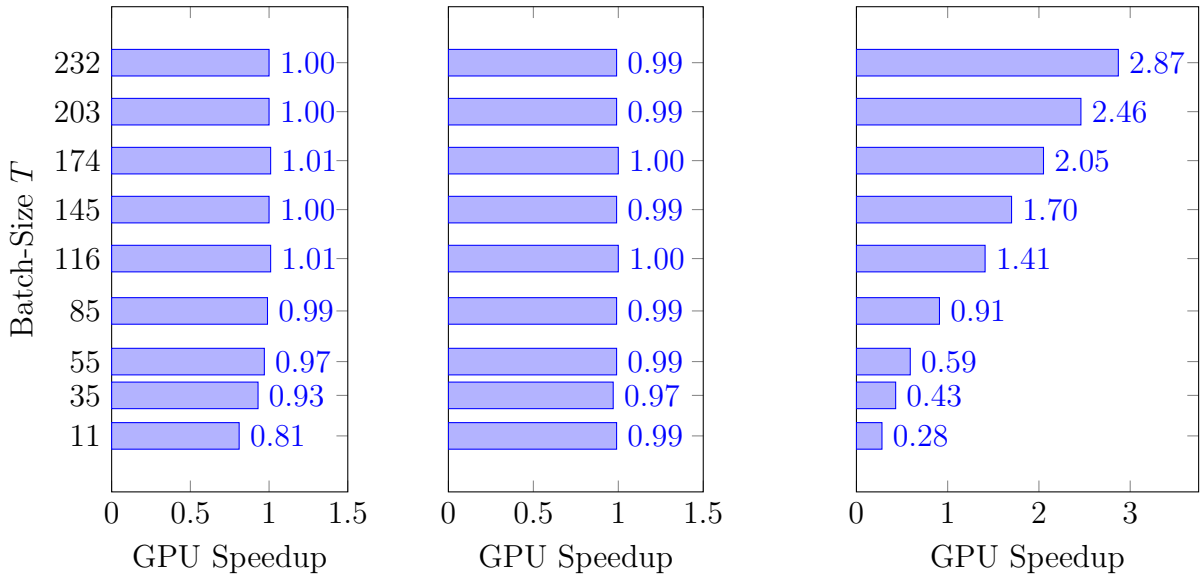
(a) Time Taken by the Full Model: GPU Speedup over all batch-sizes is only $\mathbf{0.97 \pm 0.04}$ – heavily impacted by LP runtimes.



(b) Time taken by the LP: GPU Speedup over all batch-sizes is only $\mathbf{0.99 \pm 0.005}$. Expectedly, there is no GPU Speedup in LP runtimes.



(c) Time taken by the Neural Network: GPU Speedup over all batch-sizes is $\mathbf{1.41 \pm 0.76}$ – as transfer time between RAM and GPU's internal memory dominates for smaller datasets. However, as batch-size increases, computation time dominates over transfer time – GPU "set" performs better than CPU "set".



(d) Speedup for the Full Model (e) Speedup for the LP (f) Speedup for the Neural Network

Figure 4.7: Finding Rewards After Restricing Script's Access to CPU Cores – Execution Times of Different Batch-Sizes J with GPU and CPU "set" Separately: Scaling is strongly hampered by the LP solver. Comparing the contributions of LP and Neural Network to total runtime on GPU "set", LP accounts for $\mathbf{94.57 \pm 5.01}\%$ of the total time.

Chapter 5

Conclusion

Our models for the Identification and the Pricing Problem outperformed previously studied ones (5) and other baseline comparisons (Sections 4.1.1 and 4.2.1 and Tables 4.1 and 4.2). For the Identification Problem, the average loss value was 14% lower than the previous 2-layered model, and 12% better than the 4-layered model, giving us better results than any other tested model. While we did not test deeper networks, we contend that using more hidden layers will only aggravate overfitting and won't provide better results - as is partly the case with the 4-layered network. The Pricing Problem's model also delivered at least $3\times$ lower loss values than other baseline comparisons for reward distribution.

On the other hand, we can definitively conclude that the Identification Problem ran $\approx 9\times$ faster on the GPU than the CPU, mainly because the model was based on tensors and a neural network, accelerated by a GPU and NVIDIA's APIs. With an approximate GPU Speedup of 9.06 for the Identification Problem, we can scale to large datasets more efficiently on the GPU than the CPU. The Pricing Problem's neural network only performed better with higher batch-sizes, with transfer times hampering performance on smaller datasets. Although the Pricing Problem's full model did not deliver a noticeable speedup (with the LP problem heavily impacting the runtime), the 2-layered network for finding rewards gave a speedup of 1.41 ± 0.76 (mean over all tested batch-sizes). This shows that neural network are inherently quick to optimize on a GPU only if the batch-sizes are large enough. This issue of high transfer times is also discussed in Section 1.3 and recent literature (9, Appendix B), inferring that one can witness GPU Speedups only when transfer time are tiny compared to

computation time. Using faster GPUs like NVIDIA Quadro GP100, and those based on the newer Volta architecture will most likely decrease the models' runtimes. There still exists enormous scope for improvement on both fronts – optimized results and faster computation.

5.1 Interesting Inferences

One may also notice compelling reflections from the results:

- One interesting observation in Table 4.2 is that the loss value from the Proportional Distribution (0.0235%) and Random Initialization (0.0331%) are very close, highlighting that the set of weights obtained from the Identification Problem are dependent on other factors $(\mathbf{f}, \mathbf{D})$ as well. In other words, only incentivizing under-sampled locations more is as good as random distribution of rewards – as agents consider environmental features and distances between locations to make decisions as well.

- By looking at the model's generated rewards (Table 5.1), one can infer that the model chooses to place large rewards in very under-sampled locations, rather than distributing more evenly over all locations. Although this non-uniform distribution is unintuitive to the human perspective, it gives much lower results than proportionally allocated rewards.

- Algorithms that use different libraries for sub-routines' implementation can have difficulty in parallelizing. As we saw with the unexpected GPU Speedup in LP runtimes in the Pricing Problem (Section 4.2.2 and Appendix B), naïve synchronization barriers can affect performance in CPUs' multi-threading mechanism as well as GPUs' threads, if the logic requires branching (Section 1.3.2). Therefore, one might consider threading the *full* algorithm instead of using external libraries for threading some sub-routines.

- GPUs are not always effective in executing all kinds of models. Factors that affect the performance include the size of the model and datasets, extent of conditional statements in the program, requirement of synchronization barriers, data transfer between RAM and GPU, algorithm's 'parallel-friendliness' and many more. Reducing

model complexity always helps, but when the models are increasingly simple and small, CPUs can do a much better job than GPUs.

Table 5.1: Example Rewards Prediction by the Pricing Problem's Model: The prediction is relatively sparse and non-uniform. Parameters: $\mathcal{R} = 1000$, Loss value = 0.0068%, Epochs = 1000, Learning Rate = 5×10^{-5}, Weights: Set-2. (values rounded)

0.00	38.02	0.00	0.00	0.00	0.00	14.12	0.00	24.63	4.43	3.18	24.35	0.00	19.53	0.00
1.63	6.40	0.00	0.00	0.00	36.31	31.15	0.00	0.00	29.16	2.22	6.02	23.12	0.00	16.42
0.00	0.00	28.45	0.00	37.50	0.00	20.04	34.19	0.00	0.00	18.29	0.00	0.00	21.33	0.00
0.00	23.73	24.77	0.00	23.18	1.75	0.00	20.78	22.60	0.00	0.00	0.00	0.00	0.00	4.79
4.67	0.00	0.00	34.82	0.00	9.47	0.00	0.00	31.43	0.00	4.35	16.55	28.51	6.02	0.00
21.24	21.38	0.00	22.95	27.17	21.44	24.16	21.07	0.00	0.00	25.48	0.00	0.00	0.00	2.59
12.28	0.00	0.00	0.00	0.00	0.00	22.09	0.00	0.00	16.50	0.00	2.88	1.57	0.00	43.45
0.00	0.00	15.25	0.02	0.00	0.00	0.00	0.00	0.18	0.32	0.00				

5.2 Limitations

We could not parallelize some sub-routines of Algorithms 2 and 3 and had to rely on PyTorch's implementation for the most part. While PyTorch's implementation (8) may be very efficient, we do not know if it suits our models best – a typical problem when using external libraries.

On the other hand, we did not venture in adjusting the model's parameters much as the models took long time to tests, and there lied multiple possibilities. Tinkering with the models' characteristics might have optimized the models more if we had devised a way to experiment efficiently. This also remains a topic for future study.

We also could not venture in-depth of quirky behavior and ascertain reasons with high confidence. Our research project limited us to focus on improving the Avicaching game, restricting available time to rigorously study thread synchronization delays and GPU's architecture.

5.3 Further Research

There exist numerous possibilities for solving the problems better and faster – from more complex models to better preprocessing to more parallel algorithms with fewer synchronization

barriers. Some important suggestions are listed below:

Choice of Gradient-Descent Algorithm

Figure 4.4a shows how the choice of Adam's algorithm (15) for GRADIENT-DESCENT(·) helps the model to learn quickly. However, we also witness long periods of saturation after a few epochs. This was the case for several other algorithms (SGD (14) and Adagrad (16)) as well, but with different paces of learning. Since the organizers would want to further optimize the set of weights, research could be done on avoiding the long, unwavering saturation phase. This may involve using other algorithms for GRADIENT-DESCENT(·) (Algorithm 2) and/or altering the loss function (Z_P – Equation (2.1)).

Modeling LP Differently to Reduce Runtimes

LP is a simple tool for optimizing different problems, with various algorithms for solving LPs - Simplex, Criss-Cross and other Interior Point techniques. While it gives optimal results, it can be computationally expensive if the matrices are large (as depicted in Figure 4.7b). One can try several approaches to reduce computation time here:

- Implement GPU or OpenMP backend for the LP.
- Constrain Rewards differently (Section 2.2.3). Parallelized algorithms for constraining rewards can decrease the runtime for the Pricing Problem's model considerably as the current LP accounts for $\approx 94\%$ of the total runtime.

Bibliography

[1] NVIDIA Developer Blog Contributors. Parallel For All Blog. [Online]. Available: https://devblogs.nvidia.com/parallelforall/

[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *CoRR*, 2014. [Online]. Available: https://arxiv.org/abs/1410.0759

[3] Cornell Lab of Ornithology. eBird. [Online]. Available: https://ebird.org/content/ebird/about/

[4] Y. Xue, I. Davies, D. Fink, C. Wood, and C. P. Gomes, "Avicaching: A Two Stage Game for Bias Reduction in Citizen Science," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, ser. AAMAS. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 776–785. [Online]. Available: https://dl.acm.org/citation.cfm?id=2936924.2937038

[5] ——, "Behavior Identification in Two-Stage Games for Incentivizing Citizen Science Exploration," in *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, 2016, pp. 701–717. [Online]. Available: https://doi.org/10.1007/978-3-319-44953-1_44

[6] NVIDIA. NVIDIA Deep Learning Resources. [Online]. Available: https://www.nvidia.com/en-us/deep-learning-ai/developer/

[7] ——, *CUDA Toolkit Documentation - Programming Guide*, June 2017. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[8] Torch and Pytorch Contributors, *Pytorch Documentation*. [Online]. Available: https://pytorch.org/docs/0.1.12/

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016.

[10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[11] NVIDIA. Pascal Architecture Whitepaper. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[12] ——, *CUDA Binary Utilities*, June 2017. [Online]. Available: https://docs.nvidia.com/pdf/CUDA_Binary_Utilities.pdf

[13] SciPy Community, *SciPy Optimization Module Documentation*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html

[14] L. Bottou, *Large-Scale Machine Learning with Stochastic Gradient Descent*. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. [Online]. Available: https://dx.doi.org/10.1007/978-3-7908-2604-3_16

[15] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR*, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[16] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-24, Mar 2010. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html

[17] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998. [Online]. Available: https://dx.doi.org/10.1109/99.660313

[18] SciPy Community, *NumPy Documentation*. [Online]. Available: https://docs.scipy.org/doc/numpy-1.12.0/reference/index.html

[19] A. Kabra. Avicaching Code Repository. Cornell University. [Online]. Available: https://github.com/anmolkabra

[20] Arvind and R. A. Iannucci, *Two Fundamental Issues in Multiprocessing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 61–88. [Online]. Available: https://doi.org/10.1007/3-540-18923-8_15

Appendix A

Implementation

The code for the study, including all scripts and plotted results, is collected in a repository.

Both the Identification and the Pricing Problem were programmed in Python 2.7 using NumPy 1.12.1, SciPy 0.19.0 and PyTorch 0.1.12 modules (18; 13; 8) [results plotted with Matplotlib 2.0.2]. With some code optimizations, the input dataset $\mathbf{F}$ was built using NumPy's `ndarray` and PyTorch's `tensor` functions. Since PyTorch offers NumPy-like code base but with dedicated neural network functions and submodules, PyTorch's `relu` and `softmax` functions were used along with other matrix operations.

A.1 Specific Implementation Details for the Pricing Problem

Among all the code optimizations in both models, some in that for the Pricing Problem are worth discussing, as they drastically differ from Algorithm 3 or are intricate. Most optimizations relevant to the Identification Problem are trivial and relate directly to those for the Pricing Problem. Therefore, only those in the Pricing Problem model are discussed.

A.1.1 Building the Dataset F

Notice that we build the dataset $\mathbf{F}$ and batch-multiply it with $\mathbf{w_1}$ on each iteration/epoch (lines 2-3 of Algorithm 3). Doing these steps is repetitive as most elements of $\mathbf{F}$ – distances

$\mathbf{D}$ and environmental feature vector $\mathbf{f}$ – do not change unlike rewards $\mathbf{r}$. Moreover since $\mathbf{w_1}$ is fixed, Algorithm Algorithm 3 would repetitively multiply the $\mathbf{f}$ and $\mathbf{D}$ components of $\mathbf{F}$ with $\mathbf{w_1}$. To avoid these unnecessary computations, we preprocessed most of $\mathbf{F}$ by batch-multiplying with $\mathbf{w_1}$ and only multiplied $\mathbf{r}$ with the corresponding elements of $\mathbf{w_1}$. Figure A.1 describes the process graphically. Although this preprocessing might seem applicable for the model in Identification Problem too, it does not apply fully. Since the weights $\mathbf{w_1}$ are updated on each iteration/epoch, we cannot multiply them with parts of $\mathbf{F}$ beforehand (Algorithm 2). However, we can combine $\mathbf{D}$ and $\mathbf{f}$ in the preprocessing stage and simply append $\mathbf{r}[t]$ on each iteration, reducing execution time.

A.1.2 Modeling the Linear Programming Problem in the Standard Format

The `scipy.optimize` module's `linprog` function requires that the arguments are in standard LP format. As discussed in Section 2.2.3, Equation (2.5) resembles the standard format more closely than Equation (2.4), but it may not be clear how so.

Considering $\mathbf{u}$ and $\mathbf{r'}$ as variables $\mathbf{x}$, Equation (2.5) translates into Equation (A.1) (J is the number of locations).

$$\begin{aligned} \text{minimize} \quad & \begin{bmatrix} \mathbf{0_J} \\ \mathbf{1_J} \end{bmatrix}^T \begin{bmatrix} \mathbf{r'} \\ \mathbf{u} \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} I_J & -I_J \\ -I_J & -I_J \\ \mathbf{1}_J^T & \mathbf{0}_J^T \end{bmatrix} \begin{bmatrix} \mathbf{r'} \\ \mathbf{u} \end{bmatrix} \leq \begin{bmatrix} \mathbf{r} \\ -\mathbf{r} \\ \mathcal{R} \end{bmatrix} \\ & r'_i, u_i \geq 0 \end{aligned} \tag{A.1}$$

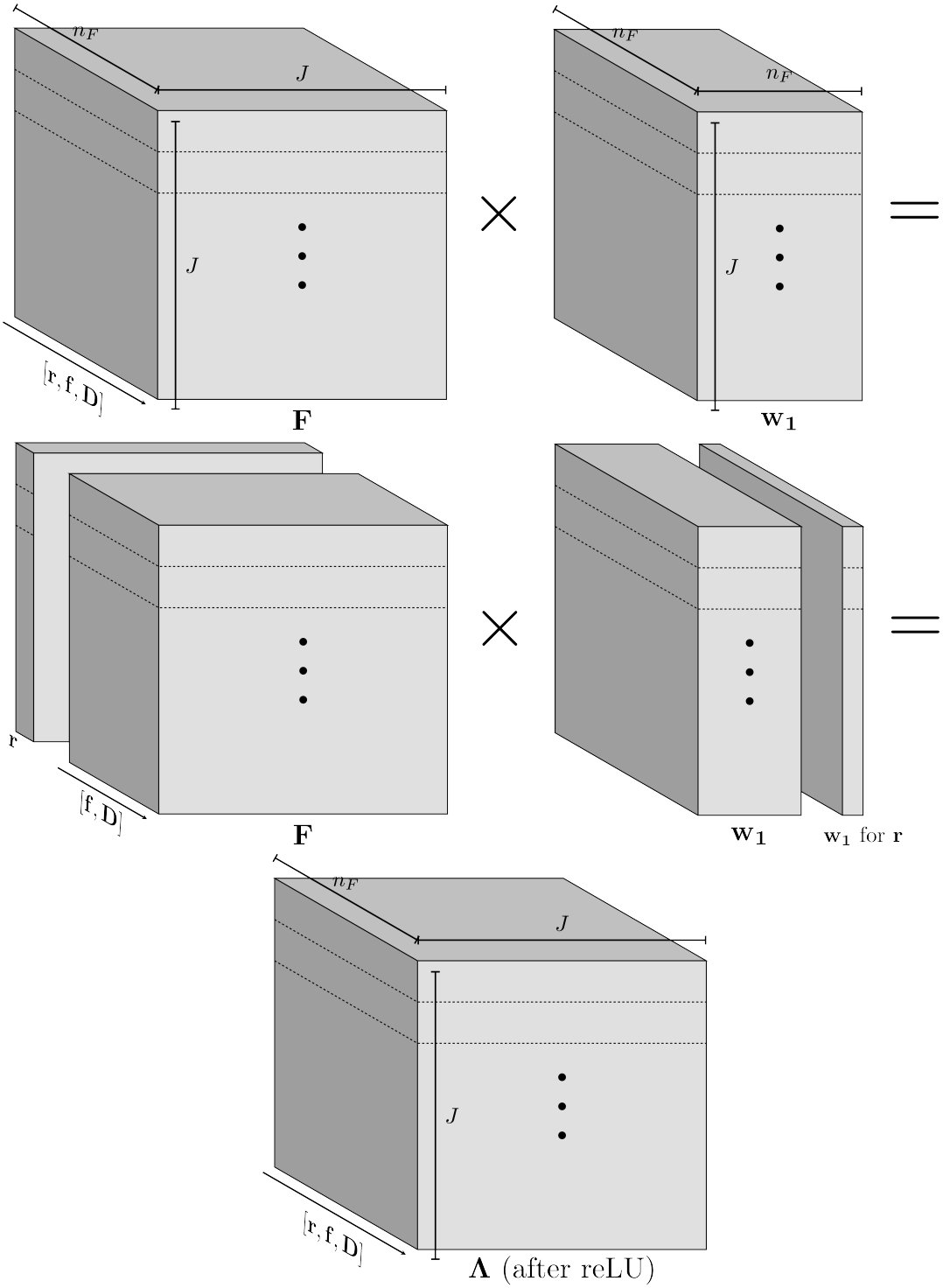


Figure A.1: Splitting and Batch Multiplying $\mathbf{F}$ and $\mathbf{w_1}$: This splitting is possible if one thinks of these 3-D tensors as batches of slices 2-D matrices depicted by the dotted lines. For example, when the top figure is split into horizontal batches, we multiply a $J \times n_F$ with a $n_F \times n_F$ matrix.

Appendix B

GPU Speedup in LP Computation

Even though we intentionally transferred the rewards vector to and constrained it using `scipy.optimize` module's `linprog` function on the CPU, we obtained an unexpected GPU Speedup in the LP runtimes (see Section 4.2.2 and Figure 4.6b). Confounded by this weird behavior, we wanted to pinpoint the reason(s). It was clear that `scipy.optimize.linprog()` could not have differentiated between the configurations and delivered different results. However, since this was not our study's prime motive, we did not take a very rigorous quantitative approach in determining the cause(s).

B.1 Possible Reasons for GPU Speedup

There could have been many reasons for this bizarre behavior, including but not limited to:

1. SciPy's Optimize module differentiating between configurations. This can be ruled out because the module could not have known the configuration during which it was called. This is because only PyTorch's tensors are executable on the GPU, whereas NumPy tensors are operable on the CPU only[1] (8; 18; 13). SciPy's Optimize Module identifying the configurations is just supernatural.

2. CPU "set" exploiting more main memory than GPU "set". We suspected that since CPU "set" configuration's operations were executed solely on the CPU, the residing

[1]We didn't mind SciPy.Optimize using MKL, BLAS or any other linear algebra libraries. Even if it had, there wouldn't be a distinction between CPU and GPU "set" configurations!

datasets could have used more main memory than when GPU "set" was running. This could have hampered the performance of LP with CPU "set", as the LP had lesser space to operate in. Unlike the 1st possibility, this would have meant that CPU "set" was slowing down the LP, and not that GPU "set" was speeding up the LP.

3. Neural network in CPU "set" using more CPU threads than that in GPU "set". The Intel i7-7700K processor is quad-core with 8 threads. Since PyTorch uses OpenMP (8; 17), a parallel processing API for CPUs, we fancied the neural network to run on simultaneous processors/threads, thus allowing less available threads for the LP to run/taking up CPU's resources/causing latency due to synchronization locks[2].

B.2 LP Slowing Down or Speeding Up?

First we determined whether the LP runtime was being sped up in GPU "set" or slowed down in CPU "set". To test this, we created a copy of our Pricing Problem's model, which focused only on logging LP runtimes at each epoch. We used 100 locations, 20 time units, and 10 total features including distances and excluding rewards (all randomly initialized with a single seed). For a baseline comparison, we scripted the same LP without the neural network, which gave us the true runtimes for the LP ('Only LP' setting), without any involvement of PyTorch modules or functions.

Comparing the former runtimes (CPU and GPU "set") with 'Only LP' runtime, we observed that the LP in the CPU "set" configuration took longer to execute than that in 'Only LP' setting during each epoch (Figure B.1). We also noticed little to no interaction between the neural network in GPU "set" with the LP, as the runtimes of LP in GPU "set" were similar to those of LP in 'Only LP' setting. This confirmed that CPU "set" was slowing down the LP and GPU "set" was not speeding it up. But why?

[2]Algorithm 3 for the Pricing Problem is sequential in that calculated tensors have to be fully available before LP starts - a 'Fork and Join' synchronization task (20, Section 2.2).

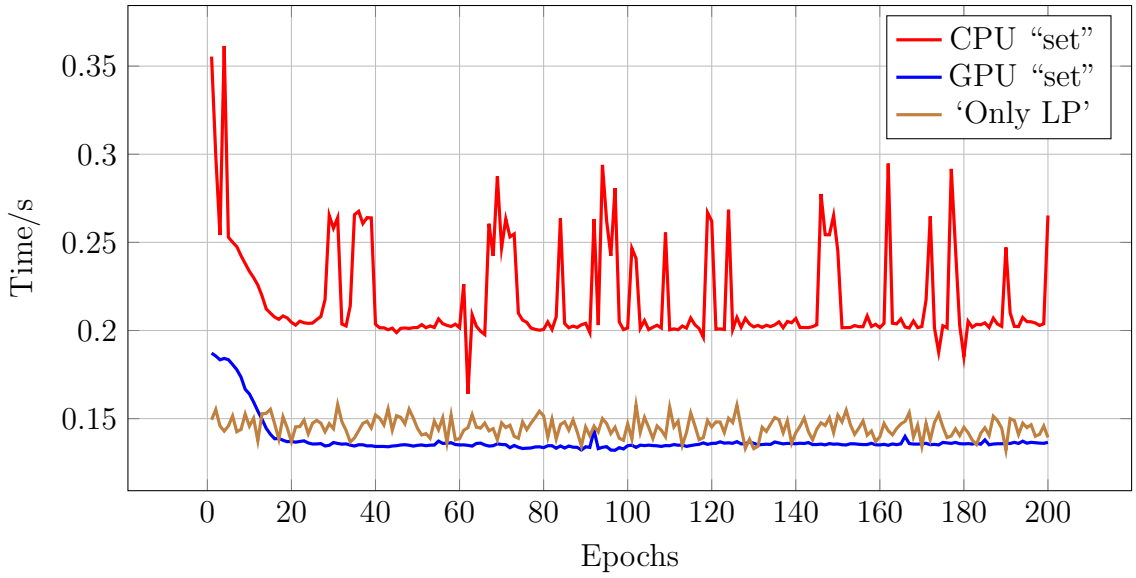


Figure B.1: LP Runtime Example for Different Configurations: LPs in both CPU and GPU "set" start running slowly, but pick up speed after ≈ 20 epochs. A possible reason for spikes in CPU "set" is given in Appendix B.4. The test was done on a random dataset for 200 epochs, while the other experiment specifications were same as in Chapter 3.

B.3 CPU and Main Memory Usage

While logging the LP runtimes, we also recorded an estimate of the amount of computer resources both configurations were using. Using the `top` package in Ubuntu, we polled the resource monitor every 0.1 seconds while the python script was running[3]. Figures B.2 and B.3 shows how much main memory and CPU resource each setting was using.

CPU Usage

We see that CPU "set" constantly used more than 4 out of 8 available threads during execution, i.e., $> 400\%$ CPU usage, while GPU "set" only used a single thread. Also, since we polled at every 0.1 second, and the LP took a minimum of 0.14 seconds (Figure B.1), the data plotted in Figure B.2 must show resource use *while* the LP was running. Considering that the LP in 'Only LP' setting only used a single thread (100%), it makes sense that GPU "set" would use 1 thread for execution – the neural network operations were performed on the

[3] Running processes were polled every 0.1 seconds - contributing to a 'log'. The longer the script ran, the more number of logs collected.

GPU, leaving the CPU empty for management and LP.

On the other hand, it is apparent that CPU "set" had multi-threaded operations running simultaneously (#3 in Appendix B.1). Since we know from the 'Only LP' setting that the LP only used a single thread, the other threads in CPU "set" must have been the neural network. Although this counters our reasoning that the neural network threads should have fully synchronized and terminated before the LP started, it seems that those threads were still active. Therefore, PyTorch and OpenMP continued calculating for the neural network even after the LP had started. Nevertheless, this activity does not impact correctness, as found from optimization tests on CPU "set"[4].

Main Memory Usage

GPU "set" was using 10 times as much main memory as CPU "set" or 'Only LP', in addition to storing and operating on tensors in the GPU's internal memory. Not only this is weird, but it is also opposite of what we expected to happen: CPU "set" using more main memory and hampering LP performance. It is contradictory that the LP performs better (even as good as 'Only LP') on GPU "set" if we consider RAM usage as the reason – GPU "set" uses a lot more main memory than CPU "set". Clearly, main memory usage cannot be a criterion for assessing LP performance on different configurations.

B.4 Parallelism Doesn't Always Help

Now that we had determined that multi-processing/threading might be the root of the strange GPU Speedup, we asked why. Were the simultaneous threads gobbling up valuable CPU resources or were the neural network threads causing latency due to prolonged synchronization delays?

Testing the first possibility was trivial enough - manually run LPs simultaneously on different threads (in 4 available hyper-threaded cores) and see if they impact each other's performance. If they didn't, multi-threading would prove independent in terms of CPU

[4]CPU "set" tests were done for optimization on original datasets to check this. Since we got the same results as for GPU "set" optimization tests Section 4.2.1, the results are not shown in the report.

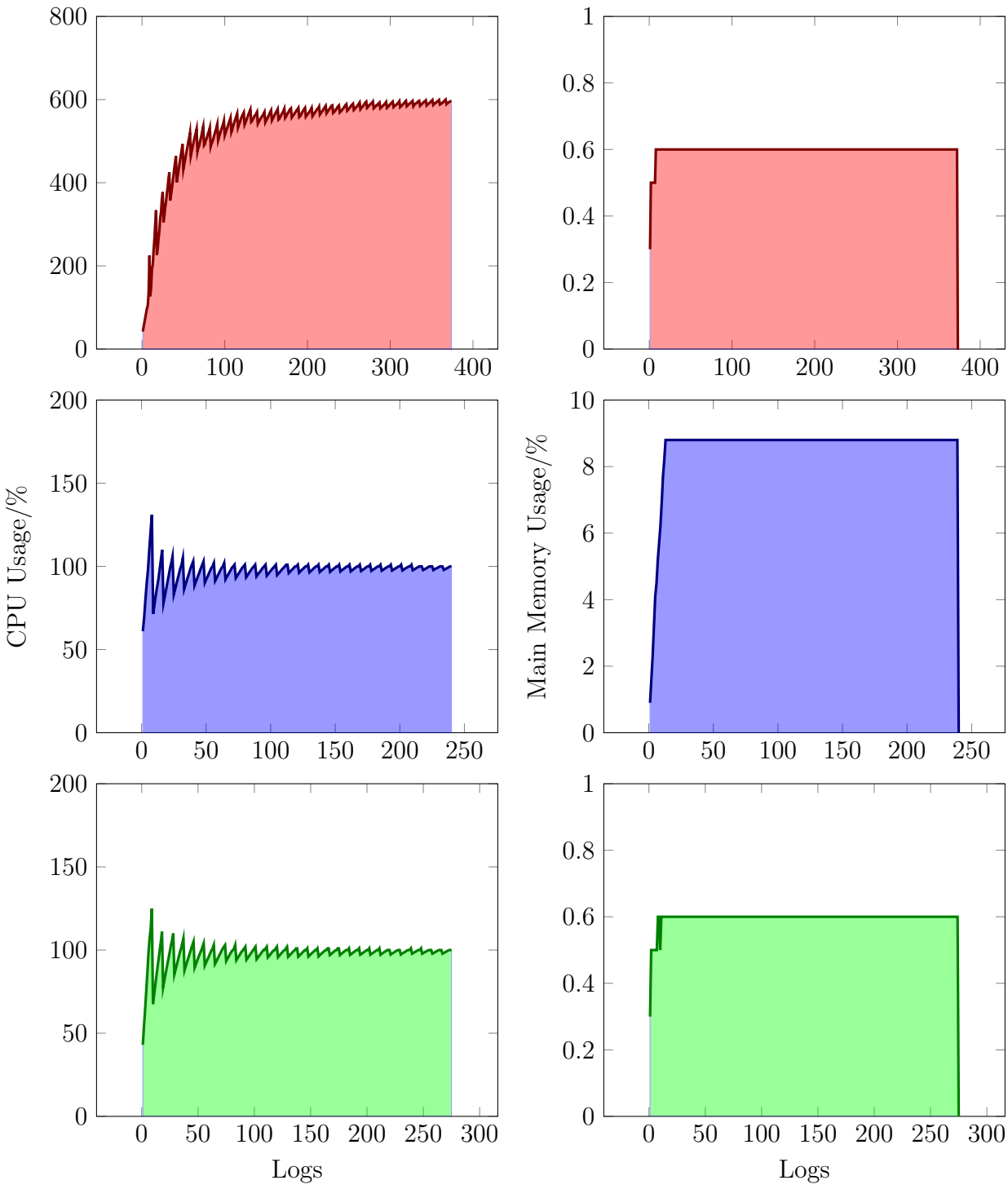


Figure B.2: CPU Usage by Different Configurations[3]: From top – CPU "set", GPU "set", 'Only LP'. CPU Usage for GPU "set" and 'Only LP' are very similar as operations other than the LP run on the GPU.

Figure B.3: Main Memory Usage by Different Configurations[3]: From top – CPU "set", GPU "set", 'Only LP'. The neural network doesn't occupy much main memory in CPU "set" – could be due to Python/PyTorch's garbage collection.

resources, and there would be no reason to believe that simultaneous threads would eat up each other's CPU resources. Running several instances of the LP on multiple threads and cores, we found no impact of multi-processed scripts on each other when it came to CPU resources. The scripts were unaffected by any other processes on other cores of CPU. One might very well argue that the independence of the LPs in these tests does not highlight the dependence of the LP to neural network in the Pricing Problem, and that these tests are wrongly based. Well, we contend that the argument pertains to the second possibility and not the first one.

CPU "set" was utilizing more than half of the available 8 threads, and we were concerned if there was latency due to thread synchronization. Since we used `scipy.optimize.linprog()`'s Simplex solver, which was distinct from PyTorch, this led to a 'Fork and Join' task-synchronization requirement (20, Section 2.2) after the neural network was complete. We suspected this causing latency as the LP waited for the neural network to finish. To test this hunch, we ran the scripts with restricted access to CPU threads. Using the `taskset` command in Ubuntu, we manually forced the script to use specific CPU threads. If CPU "set" slowed down even after restriction, then there would not be synchronization delays. Whereas, a gradual increase in execution time with the number of available threads would show that more threads for the neural network create delays in LP's execution.

Figure B.4 shows how the access to CPU's threads affected the LP's runtime. Clearly, restricting access to a single thread avoids synchronization delays in the Fork and Join task, allowing the LP to run faster on all epochs. Based on this finding, we restricted the Pricing Problem model's access to the all CPU threads, giving us better CPU "set" performance (Section 4.2.2) and removing the unexpected GPU Speedup (Figures 4.7b and 4.7e).

Well, does it reduce the neural network's performance (higher runtime due to lower parallelism)? Shockingly, it does the opposite: increase the performance. Explaining this phenomena requires in-depth understanding of PyTorch's implementation with OpenMP, but there can be some possibilities:

- More parallelism could result in competition between threads for RAM access (20, Section 2.1).

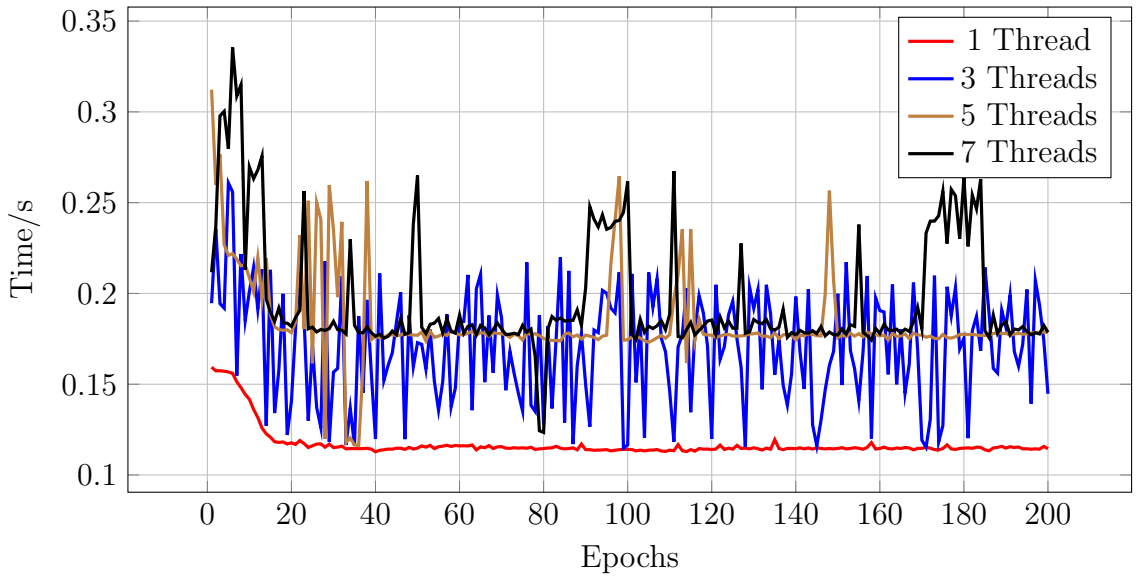


Figure B.4: LP Runtime Example with Restricted Thread Access: With only 1 thread accessible to the whole script (neural network and LP), the LP performs as good as 'Only LP' setting (Appendix B.2). We start seeing erratic synchronization delays for 3 threads, and constant delays with more threads.

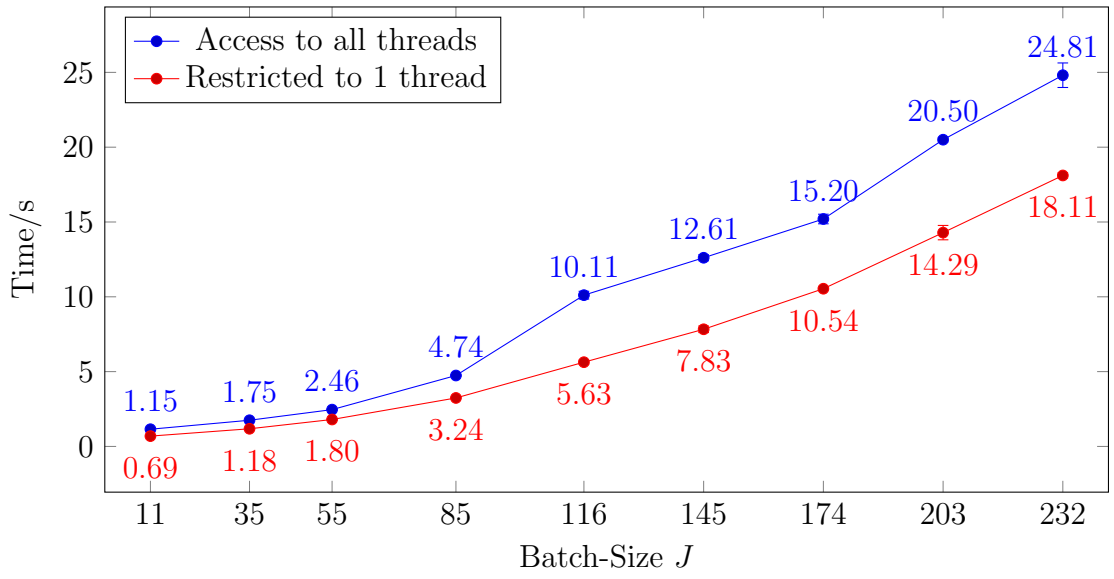


Figure B.5: Neural Network's Performance (CPU "set") with and without Thread Access Restrictions: Data from the Pricing Problem Tests in Section 4.2.2. Contrary to one's expectation, fewer threads decreases the network's execution time.

- There could still be inconspicuous synchronization requirements in the network, caused due to our algorithm's structure.

We cannot ensure one of them nor deny any other possibility. This area requires further research and we are open to suggestions. As for the results, we use the restricted access results for CPU "set" to calculate GPU Speedups in Section 4.2.1.