List of Functions, Symbols & Terms

Functions	
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$(\cdot)$	Defined as $\text{reLU}(z) = \max(0, z)$
softmax$(\cdot)$	Defined as $\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_i \exp(z_i)}$
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	Central Processing Unit: 1-8 cores with higher clockspeed and wider instruction set per core
CPU "set"	*All* operations done on the CPU
CUDA	NVIDIA's parallel computing API for its GPUs; stands for Compute Unified Device Architecture
Epoch	One training/testing period; iteration
GPU	Graphical Processing Unit: $>$ 1000 cores with lower clockspeed and smaller instruction set per core
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 $[\mathbf{c}^T \cdot \mathbf{x}]$; subject to $[\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b},\ x_i \geq 0]$
Tensor	Multi-dimensional (usually more than 2 dimensions) array

Contents

1 Introduction **4**

1.1 Important Questions 5

1.1.1 Solving Faster 5

1.1.2 Better Results 5

1.1.3 Adjusting the Model's Features 6

1.2 Computation Using GPUs 6

2 Problem Formulation **6**

2.1 Identification Problem 7

2.1.1 Structure of Input Dataset for Identifying Weights 8

2.1.2 Minimizing Loss for the Identification Problem 9

2.2 Pricing Problem 10

2.2.1 Input Dataset for Finding Rewards 11

2.2.2 Optimizing & Constraining Rewards 11

3 Experiment Specifications **13**

3.1 Running the Identification Problem's Model 14

3.1.1 Optimizing the Original Dataset 14

3.1.2 Testing GPU Speedup on the Random Dataset 15

3.2 Running the Pricing Problem's Model 15

3.2.1 Optimizing the Original Dataset 15

3.2.2 Testing GPU Speedup on the Random Dataset 16

4 Results **16**

4.1 Identification Problem's Results 16

4.1.1 Optimization Results 16

4.1.2 GPU Speedup Results 18

4.2 Pricing Problem's Results 19

4.2.1 Optimization Results 19

4.2.2 GPU Speedup Results 21

5 Conclusion **23**

Appendix A Implementation **26**

A.1 Specific Implementation Details for the Pricing Problem 26

A.1.1 Building the Dataset **F** 26

A.1.2 Modeling the Linear Programming Problem in the Standard Format . 27

List of Tables

1 Loss Values Calculated for Different Models for Identification Problem 17

2 Loss Values Calculated from Different Sets of Rewards 21

List of Figures

1 Visual representation of the Input Dataset 8

2 Neural network designed for the Identification Problem 9

3 Logic Flow of Algorithm 3 12

4 Test Loss Plots of Different Learning Rates to Find Weights 17

5 Comparison of Loss Values from Different Models of the Identification Problem 18

6 Predicted Probabilities of Agents Visiting Each Location Plotted on a Map (Latitude, Longitude) Representing Tompkins and Cortland Counties, NY . . 18

7 Train & Test Loss Values' Plots for One of the Runs of Different Models . . 19

8 Finding Weights - Execution Times of Different Batch-Sizes T with GPU and CPU "set" Separately 20

9 Finding Rewards - Execution Times of Different Batch-Sizes J with GPU and CPU "set" Separately 22

10 Splitting and Batch Multiplying **F** & $\mathbf{w_1}$ 27

1 Introduction

Optimizing predictive models on datasets that are obtained from citizen-science projects can be computationally expensive as these datasets grow in size with time. Consequently, models based on multiple-layered neural networks, Integer Programming and other optimization routines can prove increasingly 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 to scale if the organizers use CPUs to run optimization models. However, Graphical Processing Units (GPUs), which offer multiple cores to parallelize computation, can outperform CPUs in computing such predictive models if these models heavily rely on large-scale matrix multiplications. 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.

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", Avicaching is a incentive-driven game trying to homogenize the spatial distribution of citizens' (agents') observations [cite website]. 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 rewarding agents who visit under-sampled locations (1). For the organizers, a more homogeneous observation dataset can enable *better* mapping and generate more confidence in the dependent models' validity.

To accomplish this task of specifying rewards at different locations based on the historical records of observations, Avicaching would learn the change in agents' behavior when a certain sample of rewards were applied to the set of locations, and then distribute a newer set of rewards across the locations based on those learned parameters (2). This requirement naturally translates into a predictive optimization problem, which is implemented using multiple-layered neural networks and linear programming.

1.1 Important Questions

Although the previously devised solutions to Avicaching were conceptually effective (1)(2), using CPUs to solve Mixed Integer Programming and shallow neural networks made them impractical to scale. While wider applicability would have increased homogeneity in the observation set, Mixed Integer Programming formulations made it hard for Avicaching to scale and influence a larger segment of eBird. 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 in next sections.

1.1.1 Solving Faster

We were interested in using GPUs to run our models, with their growing capability to accelerate problems in Artificial Intelligence and Machine Learning - more specifically, large matrix and tensor operations. Newer generation NVIDIA GPUs equipped with thousands of CUDA (NVIDIA's parallel computing API) cores can empower Avicaching's organizers to scale the game, if it is based on extensive simple arithmetic operations on tensors and matrices, rather than logical steps (why? - reasoned in Section 1.2). Since even the faster CPUs (in the range of Intel Core i7 chipsets and faster) are sequential in processing and do not provide as comparable parallel processing as GPUs do, we can solve the problem much faster. **But how much faster?**

1.1.2 Better Results

The previous model for learning the parameters in agents' change of behavior on a fixed set of rewards delivered predictions that differed 26% from Ground Truth (2, // todo), which was used to distribute a new set of rewards on a budget. If we could get closer to the Ground Truth, i.e., learn the parameters for the change better, we could distribute a new set of rewards based on better learning and prediction. Since the organizers need the *best* distribution of rewards (key aspect in testing the model), we would need a set of learned parameters that give lesser difference from the Ground Truth (in terms of Normalized Mean Squared Error (2, // todo)). In a gist, **can we learn the parameters better**, and **what**

is the best allocation of rewards?

1.1.3 Adjusting the Model's Features

Once our model starts delivering better results than the previously devised models, one thinks if one can change some characteristics of the model (known as hyper-parameters such as learning rate) to get even better results (though one could also build a better model too). While a goal of "getting better results" may seem like an unending strife (any one can never get satisfied), there is a trade-off with practicality as these adjustments take time and computation power to test - and we don't have unlimited resources. Therefore, we ask if one could **reasonably adjust hyper-parameters to improve performance and optimization.** By "reasonable adjustments" we mean changes that improve performance by more than 5 times using comparable resources.

1.2 Computation Using GPUs

// todo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2 Problem Formulation

Since NVIDIA General Purpose GPUs enable faster computation on tensors and matrices, accelerated through CUDA and cuDNN, both the Identification (Section 2.1) and the Pricing

Problem (Section 2.2) were formulated as tensor-based 3-layered and 2-layered neural networks respectively using the PyTorch library.

2.1 Identification Problem

As discussed in Section 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 $\mathbf{y_t}$ and $\mathbf{x_t}$ of agents' visit densities, with and without the rewards $\mathbf{r_t}$, we want to find weights $\mathbf{w_1}$ and $\mathbf{w_2}$ that caused the change from $\mathbf{x_t}$ to $\mathbf{y_t}$, factoring in possible influence from environmental factors $\mathbf{f}$ and distances between locations $\mathbf{D}$. Although the original model proposed to learn a single set of weights $\mathbf{w}$ (2), this 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. Mathematically, the model can be formulated as:

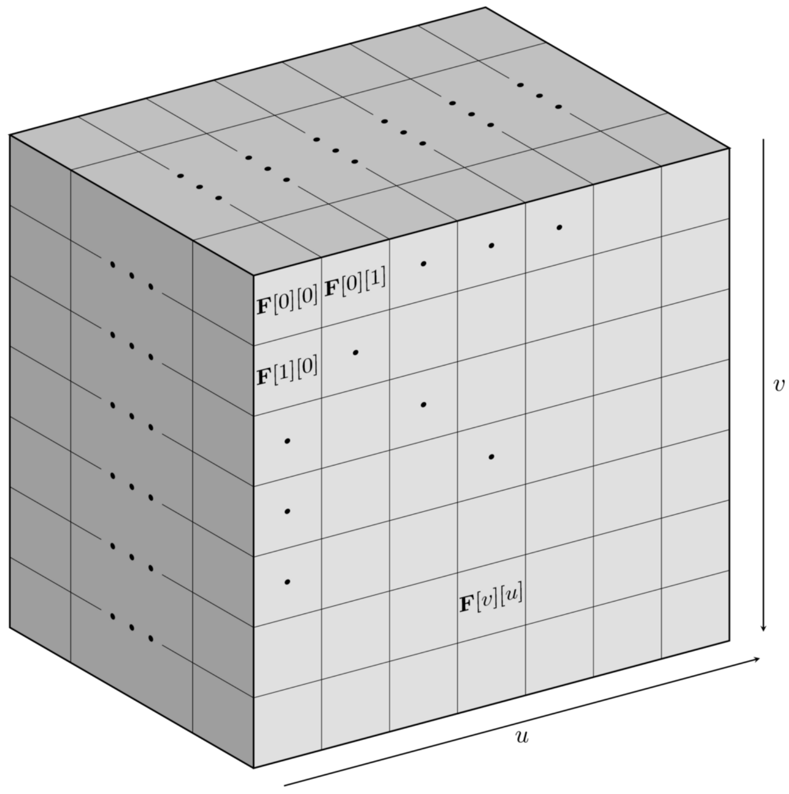
$$\underset{\mathbf{w_1},\mathbf{w_2}}{\text{minimize}} \quad Z_I(\mathbf{w_1},\mathbf{w_2}) = \sum_t (\omega_t(\mathbf{y_t} - \mathbf{P}(\mathbf{f},\mathbf{r_t};\mathbf{w_1},\mathbf{w_2})\mathbf{x_t}))^2 \tag{1}$$

where ω_t is a set of weights (not a learnable parameter) at time t capturing penalties relative to how strongly we want to homogenize locations at time t. In other words, it highlights if the organizer wishes higher homogeneity at one time over another. Elements $p_{u,v}$ of $\mathbf{P}$ are given as:

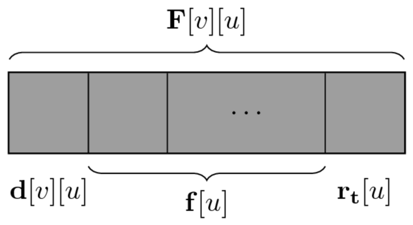
$$p_{u,v} = \frac{\exp(\mathbf{w_2} \cdot \text{reLU}(\mathbf{w_1} \cdot [d_{u,v}, \mathbf{f_u}, r_u]))}{\sum_{u'} \exp(\mathbf{w_2} \cdot \text{reLU}(\mathbf{w_1} \cdot [d_{u',v}, \mathbf{f_{u'}}, r_{u'}]))} = \frac{\exp(\Gamma_{u,v})}{\sum_{u'} \exp(\Gamma_{u',v})} = \text{softmax}(\Gamma_{u,v}) \tag{2}$$

To optimize the loss value $Z_I(\mathbf{w_1},\mathbf{w_2})$ (Equation 1), the neural network learns the set of weights through multiple epochs of backpropagating the loss using gradient descent. Furthermore, the program processes the dataset before feeding to the network to avoid unnecessary sub-epoch iterations and to promote batch operations on tensors.

2.1.1 Structure of Input Dataset for Identifying Weights



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



(b) Contents of $\mathbf{F}[v][u]$

Figure 1: Visual representation of the Input Dataset

Since preprocessing the dataset impacts the efficiency of the network, the input dataset, comprising of distance between locations $\mathbf{D}$, environmental features $\mathbf{f}$ and given rewards $\mathbf{r_t}$ (all normalized), is built in a specific manner. Since GPUs are efficient in operating on matrices and tensors, the input dataset is built into a tensor (Figure 1a) such that batch operations could be performed on slices $\mathbf{F}[v]$.

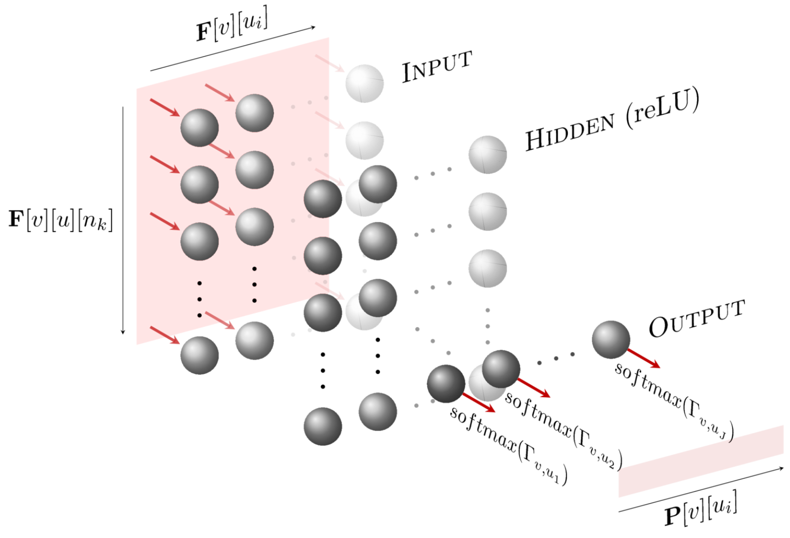
Another advantage of building the dataset as a tensor comes with the PyTorch library, which provides convenient handling and transfer of tensors residing on CPUs and GPUs. Algorithm 1 describes the steps to construct this dataset.

Algorithm 1 Constructing the Input Dataset

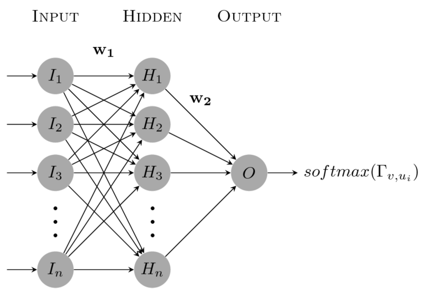
```
1: function BUILD-DATASET($\mathbf{D}, \mathbf{f}, \mathbf{r_t}$)
2:     $\mathbf{D} \leftarrow$ NORMALIZE($\mathbf{D}$)    ▷ $\mathbf{D}[u][v]$ is the distance between locations $u$ and $v$
3:     $\mathbf{f} \leftarrow$ NORMALIZE($\mathbf{f}, axis = 0$)    ▷ $\mathbf{f}[u]$ is a vector of env.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

2.1.2 Minimizing Loss for the Identification Problem

As shown in Figure 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 generating the results using the softmax(·) function. The network can also be visualized as a collection of 1-dimensional layers (Figure 2b), with the softmax(·) calculated on the collection's output. It



(a) 3-dimensional view of the network slice, taking in $\mathbf{F}[v]$



(b) Side view of the network

Figure 2: Neural network designed for the Identification Problem

is important to clarify that the network in Figure 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. 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 2a. Algorithm 2 details the steps for learning the parameters $\mathbf{w_1}$ and $\mathbf{w_2}$ based on Equations 1 & 2.

Algorithm 2 Algorithm for the Identification Problem

```
1: $\mathbf{w_1} \leftarrow$ RANDOM( $(J, n_F, n_F)$ )                  ▷ $\mathbf{w_1}$ has dimensions $J \times n_F \times n_F$
2: $\mathbf{w_2} \leftarrow$ RANDOM( $(J, n_F, 1)$ )                   ▷ $\mathbf{w_2}$ has dimensions $J \times n_F \times 1$
3: for $e = 1, 2, \ldots,$ Epochs do
4:     $loss \leftarrow 0$
5:     for $t = 1, 2, \ldots, T$ do
6:         $\mathbf{F} \leftarrow$ BUILD-DATASET($\mathbf{D}, \mathbf{f}, \mathbf{r}[t]$)          ▷ Defined in Algorithm 1
7:         $\mathbf{H} \leftarrow$ reLU(BATCH-MULTIPLY($\mathbf{F}, \mathbf{w_1}$))
8:         $\mathbf{O} \leftarrow$ softmax(BATCH-MULTIPLY($\mathbf{H}, \mathbf{w_2}$))
9:         $\mathbf{P} \leftarrow \mathbf{O}^T$
10:        $loss \leftarrow loss + (\omega(\mathbf{y}[t] - \mathbf{P} \cdot \mathbf{x}[t]))^2$
11:    GRADIENT-DESCENT($loss, \mathbf{w_1}, \mathbf{w_2}$)
12:    $\mathbf{w_1}, \mathbf{w_2} \leftarrow$ UPDATE-USING-GRADIENTS($\mathbf{w_1}, \mathbf{w_2}$)
13:    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. Thus, given a budget of rewards $\mathcal{R}$, this optimization problem can be expressed as:

$$\begin{aligned}
\underset{\mathbf{r}}{\text{minimize}} \quad & Z_P(\mathbf{r}) = \frac{1}{n}\|\mathbf{y} - \overline{\mathbf{y}}\| \\
\text{subject to} \quad & \mathbf{y} = \mathbf{P}(\mathbf{f}, \mathbf{r}; \mathbf{w_1}, \mathbf{w_2})\, \mathbf{x} \\
& \sum_i r_i \leq \mathcal{R} \\
& r_i \geq 0
\end{aligned} \tag{3}$$

where elements of $\mathbf{P}$ are defined as in Equation 2.

To allocate the rewards $\mathbf{r}$ optimally, the calculations for the pricing problem are akin to that for the Identification Problem (Section 2.1). However, since only 1 set of rewards need to be optimized, we use an altered 2-layer network instead of the 3-layer network used to identify the weights. While Equation 3 looks like a typical Linear Programming (LP) problem, only a part of the formulation uses LP to constrain the rewards. Calculation for $\mathbf{P}$ is modeled as a 2-layer network that minimizes the loss function $Z_P(\mathbf{r})$ using gradient descent. Although this use of a neural network may seem similar to that of the Identification Problem, there are major changes in the structure of the network used here. These alterations for the Pricing Problem and differences from the Identification Problem are discussed further in the following sections. 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 (note that "weights" here refer to the 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 $\mathbf{r}$ does not depend on t - we want a generalized set of rewards for all time t per location u. Therefore, the algorithm for constructing $\mathbf{F}$ (Section 2.1.1) is same as Algorithm 1 but with a change - $\mathbf{r_t}$ replaced by $\mathbf{r}$.

2.2.2 Optimizing & Constraining Rewards

The algorithm for finding $\mathbf{P}$ is very similar to Phase 1 of Algorithm 2 but without any epochs of t, as $\mathbf{x}, \mathbf{y}, \mathbf{r}$ are vectors rather than matrices. Also, since the program (Algorithm 3) would predict $\mathbf{y}$, it does not need labels $\mathbf{y}$ as a dataset. Although Algorithm 3 might look arcane in the logic flow, it is straightforward - as displayed in Figure 3. The algorithm only arranges the commands in a particular way to optimize implementation and execution.

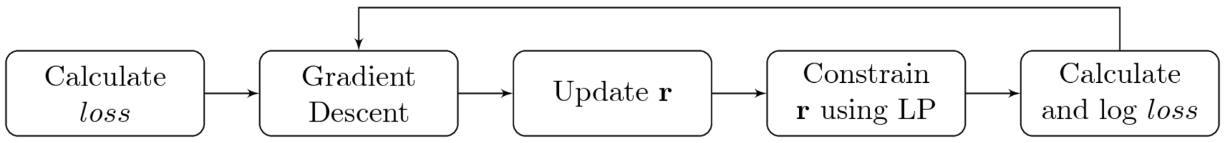


Figure 3: Logic Flow of Algorithm 3

Algorithm 3 Solving the Pricing Problem

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

Main Script

```
8: $\mathbf{r} \leftarrow$ RANDOM( $(J)$ )          ▷ $\mathbf{r}$ has dimensions $J$
9: $loss \leftarrow$ FORWARD($\mathbf{D}, \mathbf{f}, \mathbf{r}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{x}$)
10: for $e = 1, 2, \ldots,$ Epochs do
11:     GRADIENT-DESCENT($loss, \mathbf{r}$)
12:     $\mathbf{r} \leftarrow$ UPDATE-USING-GRADIENTS($\mathbf{r}$)
13:     $\mathbf{r} \leftarrow$ LP($\mathbf{r}, \mathcal{R}$)          ▷ LP($\cdot$) explained in Section 2.2.2
14:     $loss \leftarrow$ FORWARD($\mathbf{D}, \mathbf{f}, \mathbf{r}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{x}$)
15:     LOG-BEST-REWARDS($loss, \mathbf{r}$)          ▷ Records $\mathbf{r}$ with the lowest $loss$ yet
```

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 4, which was implemented (see Appendix A) using SciPy's Optimize Module (3). Since the module supports a standard format for doing linear programming, Equation 5 (after rearranging constraints and building $\mathbf{A}, \mathbf{b}$ and $\mathbf{c}$) is used, which is mathematically equivalent to Equation 4.

$$\begin{aligned} \underset{\mathbf{r'}}{\text{minimize}} \quad & \sum_i |r'_i - r_i| \\ \text{subject to} \quad & \sum_i r'_i \leq \mathcal{R} \\ & r_i \geq 0 \end{aligned} \tag{4}$$

$$\begin{aligned} \underset{[\mathbf{r'},\mathbf{u}]}{\text{minimize}} \quad & \sum_i u_i \\ \text{subject to} \quad & r'_i - r_i \leq u_i \\ & r_i - r'_i \leq u_i \\ & \sum_i r'_i \leq \mathcal{R} \\ & r'_i, u_i \geq 0 \end{aligned} \tag{5}$$

3 Experiment Specifications

Definition 1. *GPU Speedup: Ratio of increase in time elapsed in GPU and that of CPU per unit increase in batch-size. We define GPU Speedup in this way because we are interested in how the models will scale with increase in dataset, giving us more information than the commonly used formula (Speedup* $= \frac{CPU-time}{GPU-time\,+\,Transfer-time}$*) would do.*

To test both our models, we conducted several tests for optimization and GPU speedup over CPU. After initializing all parameters randomly and reading data from files, the models were run for 1000 to 10000 epochs depending on the complexity of the model and any potential benefits emerging with more epochs.

A Dell Precision Tower 3620 with Intel Core i7-7700K Processor [], 16GB RAM and NVIDIA Quadro P4000 GPU [] was used for all experiments. These experiments were either optimization tests on original datasets or GPU speedup tests randomly generated datasets.

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. The models were timed only for the executed operations in a neural network or the LP, excluding transfer or tensors between the RAM and GPU's memory and preprocessing.

Note that operations in the scripts were *distributed between* CPU and GPU when GPU is mentioned as "set", while the operations were executed *only* on the CPU when the "CPU" is mentioned as set. Since GPUs are inferior than CPUs at handling most operations other than simple arithmetic matrix ones, we used - and recommend using - both the CPU and the GPU in the former case (GPU "set") to handle operations each is superior at. However, since the models in Sections 2.1 and 2.2 (not the full scripts) are primarily arithmetic operations on matrices and 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 Library's and NumPy-SciPy's inbuilt implementation.

On the algorithm side, we used Adam's algorithm for Gradient-Descent($\cdot$), after testing performances of several algorithms including but not limited to Stochastic Gradient Descent (SGD) (4), Adam's Algorithm (5) and Adagrad (6) (PyTorch lets you choose the corresponding function). Since Adam's algorithm was found to work best with both models over all test runs, all experiments were done using Adam's algorithm. Hence, all results were also obtained using Adam's algorithm.

3.1 Running the Identification Problem's Model

3.1.1 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 (2) 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-layered network (Figure 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.).

3.1.2 Testing GPU Speedup on the Random Dataset

After generating random datasets of different sizes ($T = 17, 85, 173$ - Section 2.1.2, we ran our 3-layered model on each dataset size with different seeds with both GPU and CPU "set", logging time elapsed. The total time elapsed was averaged for a batch-size on a device, which were used to generate scatter/line plots (Section 4.1.2).

3.2 Running the Pricing Problem's Model

3.2.1 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 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 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 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 $= \{10^{-2}, 5 \times 10^{-3}, 10^{-3}, 5 \times 10^{-4}, 10^{-4}, 5 \times 10^{-5}, 10^{-5}\}$.

3. Choose the best.

Two sets of rewards were tested for loss values as baseline comparisons to our model - a randomly generated set, and another with elements proportional to the reciprocal of number of 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 average loss values (on different sets of weights) were compared for all tests with the baselines.

3.2.2 Testing GPU Speedup on the Random Dataset

After generating random datasets of different sizes ($J = 11, 55, 116$ - Section 2.1.2), we ran the Pricing Problem's model on each dataset size with different seeds with both GPU and CPU "set".

Since PyTorch does not provide a GPU-accelerated Simplex LP solver, we relied on SciPy's Optimize Module to solve the that part of the model. Since SciPy's implementation does not utilize the GPU *conspicuously*, we expected the LP problem to be executed on the CPU and thus deliver equal runtimes in both GPU and CPU "set" configurations.

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 hinted at the model performing the best with learning rate $= 10^{-3}$.

We observed that higher learning rates ($> 10^{-2}$) could only decrease the loss function (Z_I - Equation 1) to a limit, after each the updates in the weights caused the loss function to oscillate and increase. This phenomena is exemplified in Figure 4, which are plots of different models 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 an average 1232.56 seconds (20 runs = 5 seeds $\times$ 4 learning rates), and models with learning rates $< 10^{-3}$ 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 $< 10^{-3}$, we feel that they may be computationally expensive and temporally inconvenient to train.

Table 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 5.

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

Table 1 gives the average *end* test loss results. 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 (2, // todo), and a 4-layered network with learning rate = 10^{-3} (Section 3.1.1).

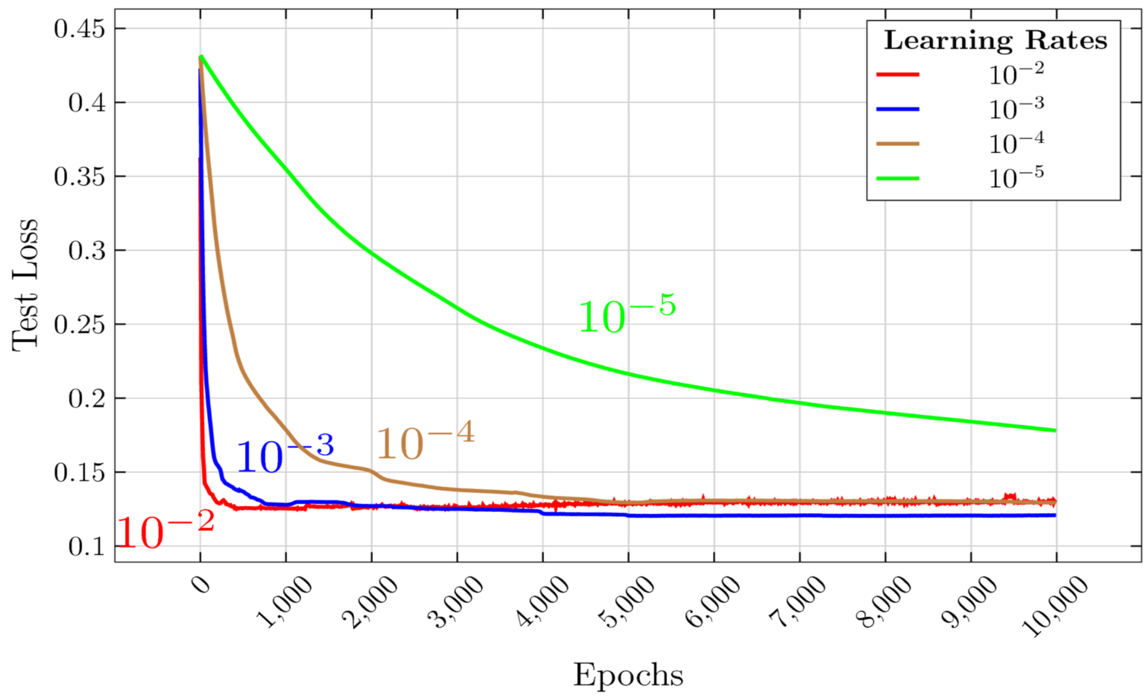


Figure 4: Test Loss Plots of Different Learning Rates to Find Weights

As depicted in Figure 5, our 3-layered neural network outperformed the previous 2-layered model by **0.14 units (14% more closer to Ground Truth - y)** (2, Table 1), and also produced much better results (12% more closer to Ground Truth) than the 4-layered model.

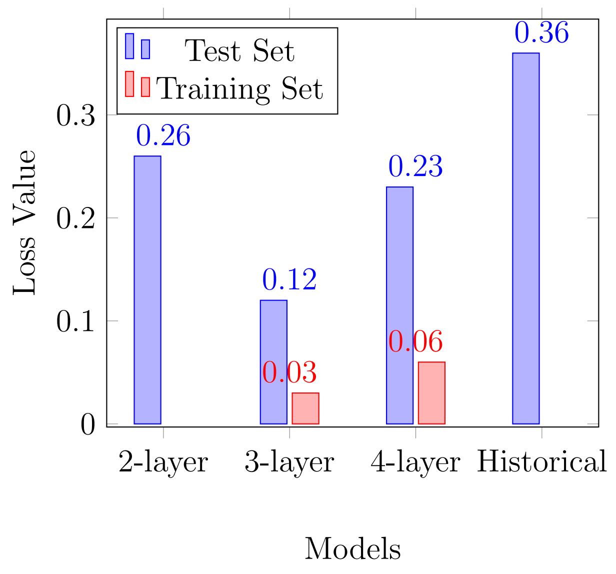


Figure 5: Comparison of Loss Values from Different Models of the Identification Problem: Loss values for the training set are inevitably lower than that for the test set, which should be the basis for comparison

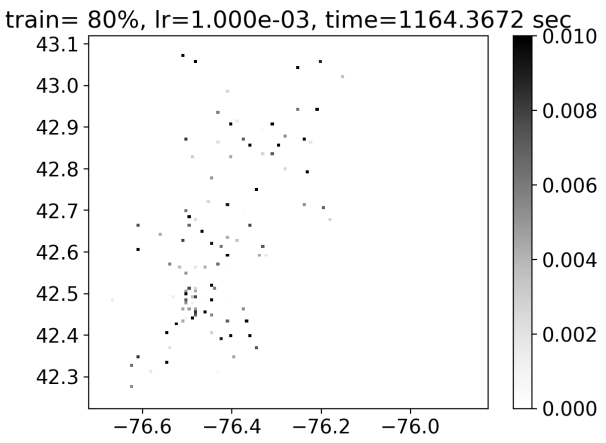


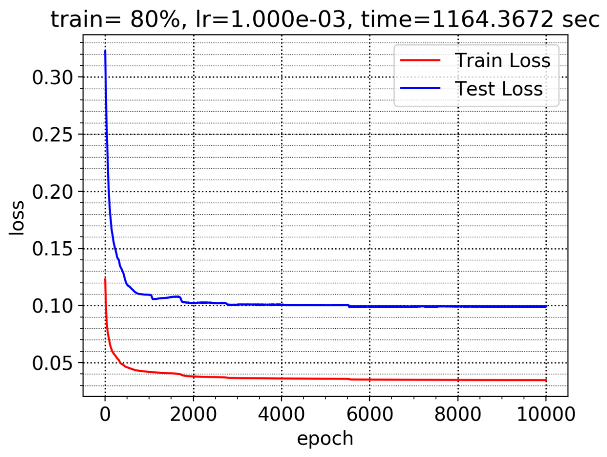
Figure 6: Predicted Probabilities of Agents Visiting Each Location Plotted on a Map (Latitude, Longitude) Representing Tompkins and Cortland Counties, NY: Dark dots represent high prediction of visits. This can be compared to the plots for the 2-layered network and other models (2, Figure 3).

We also generated the predicted probabilities of the agents visiting each location $(\mathbf{P} \cdot \mathbf{x})$ in the Test Set, and plotted it onto a map marked by the locations' latitudes and longitudes. Figure 6 shows such a plot generated by the 3-layered network, where each dot represents a location.

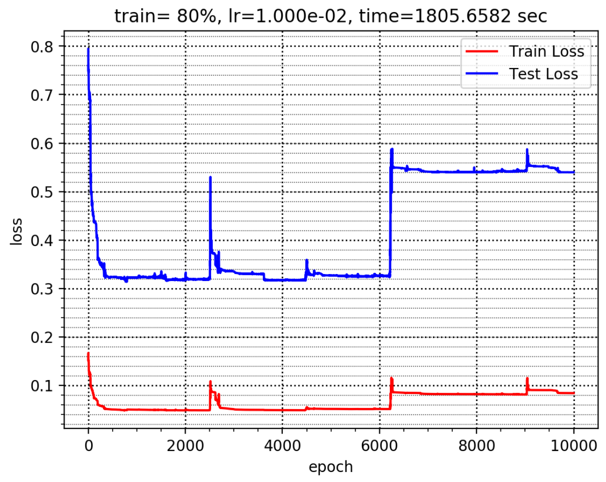
Although there remained a $\approx 9\%$ difference (0.09 loss units) in the values of training and testing set, the 3-layered model was not starkly overfitting as an average *end* difference of $8.76 \pm 1.59\%$ persisted for many epochs, instead of increasing and tuning more to the training set. On the other hand, overfitting is more remarked in the case of 4-layered model, producing an average *end* difference of $16.77 \pm 4.73\%$, but erratically hovering between 10% and 30%. This result is shown in Figure 7 with plots of loss values at each epoch for the 3- and the 4-layered network.

4.1.2 GPU Speedup Results

Running on batches of sizes $T = 17, 85, 173$ on a randomly generated dataset for 1000 epochs, with both GPU and CPU "set" separately, we obtained full execution time (including both



(a) Plot for 3-layered Model



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

Figure 7: Train & Test Loss Values' Plots for One of the Runs of Different Models: Both networks learn the set of weights quickly, as displayed in the steep descent in loss values before $\approx$ 1000 epochs. This quick learning is due to the choice of GRADIENT-DESCENT($\cdot$) function - Adam's algorithm (5). Other algorithms like SGD (4) and Adagrad (6) learn relatively slowly.

training and testing) information. The average results (3 different seeds for each batch) are plotted in Figure 8, which show promising GPU speedup over CPU figures for any batch size - for every unit increase in batch-size T, CPU "set" takes $\approx$ 4.64 seconds, while GPU "set" takes only $\approx$ 0.74 more seconds (slopes of best-fit lines).

4.2 Pricing Problem's Results

4.2.1 Optimization Results

Taking the approach mentioned in Section 3.2.1, different sets of weights gave consistent loss values (even with differently seeded rewards), which are shown in Figure []. Next, running differently seeded rewards with different learning rates on set-2 of weights, which performed the best, we obtained the lowest loss value of 0.0068%.

Compared to other reward distributions, our model optimized the rewards such that the loss value was $\approx$ **23 times** lower. We should also clarify that we compared the best loss values, as the organizer expects to find a distribution that is as optimal as possible. Table 2 lists the best loss values obtained on each type of reward allocation (model's predicted,

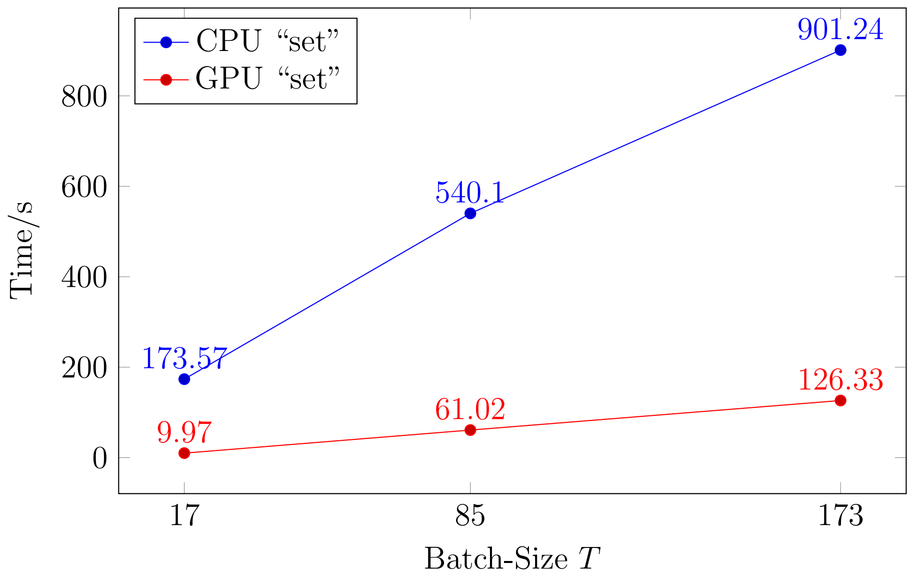
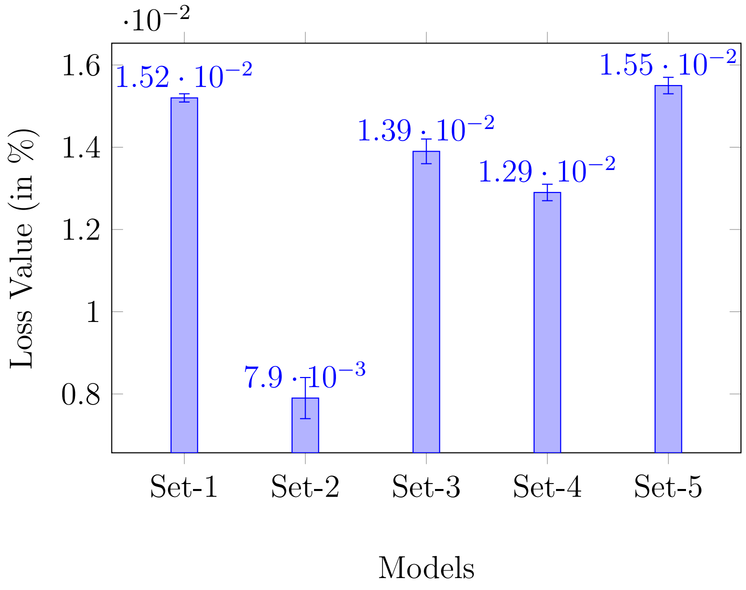


Figure 8: Finding Weights - Execution Times of Different Batch-Sizes T with GPU and CPU "set" Separately: Scaling on a GPU is more efficient (about $\frac{4.64}{0.74} \approx$ **6.22 times**) than doing so on a CPU. Also, the error bars are indiscernible because they are too small ($< 1\%$)

.



random and proportionate).

Table 2: Loss Values Calculated from Different Sets of Rewards: The values are small because the loss function $Z_P(\mathbf{r})$ (Equation 3) is averaged over the number of locations

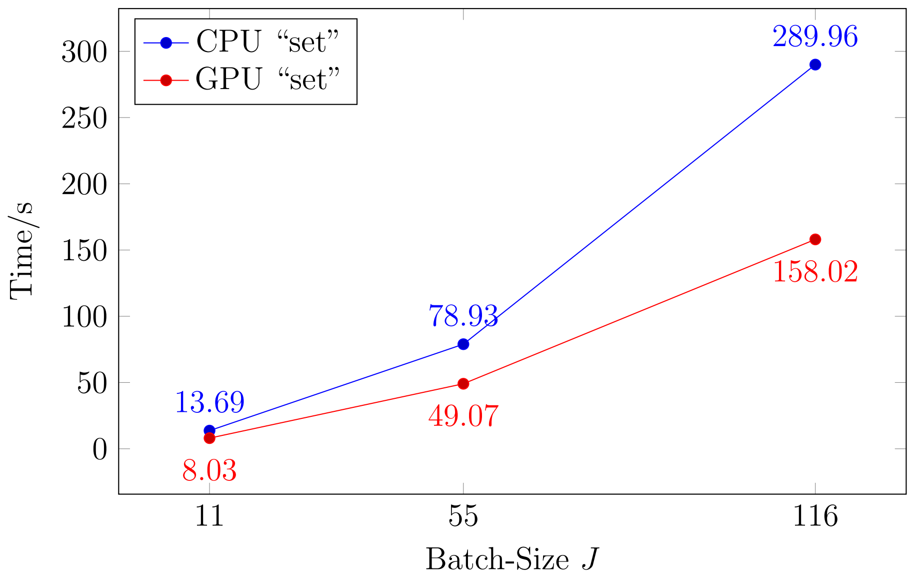
Rewards Obtained From	**Best Loss Values (In %)**
Model's Prediction	0.0068
Random Initialization	0.1600
Proportional Distribution	0.1610

4.2.2 GPU Speedup Results

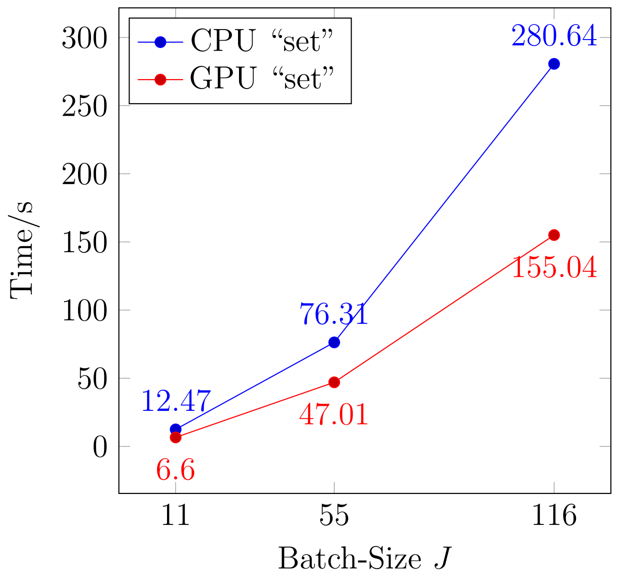
After running on different batch-sizes $J = 11, 55, 116$ (the Pricing Problem's model does not iterate over T - Algorithm 3), we did not observe profound speedup for the full model. Figure 9a shows the decreasing speedup trend, as the GPU "set" configuration started struggling to complete all epochs faster than with CPU "set". The GPU "set" config. took ≈ 1.42 seconds more for unit increase in batch-size J, whereas the CPU "set" config. took ≈ 2.44 seconds more - a speedup of just 1.72 times.

However, after realizing that the LP problem (Equations 4 & 5) might be influencing these logged times, we recorded running 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 the GPU speedup for the Neural Network was expectedly high with growing batch-sizes - about 12.48 times (Figure 9c).

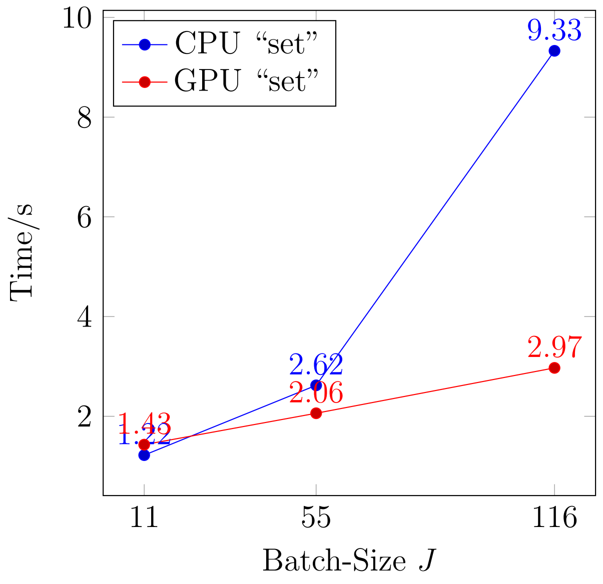
Furthermore, although we expected SciPy's Optimize Module (3) to run on the CPU (Section 3.2.2), the LP problem ran faster when the GPU was "set". We did not anticipate this behavior, and we couldn't pick out a reason for the Optimize Module supposedly doing operations differently. Since our GPU "set" and CPU "set" configurations were only enforceable on user-created functions and datasets, we do not know why the LP solver delivered different speeds while solving the same problem. If it used the CPU or the GPU, it should have done so in both cases and the runtime would have been equal. While there could be differences in implementation, we cannot affirm or deny any reason behind this quirky behavior. Figure 9b shows the unexpected speedup of $\frac{2.35}{1.41} \approx 1.66$x delivered by SciPy Optimize Module's LP solver.



(a) Time Taken by the Full Model: Unit increase in J leads to 1.42 additional seconds with GPU "set", compared to 2.44 seconds with CPU "set".



(b) Time taken by the LP: Unit increase in J increases runtime by 1.41 seconds with GPU "set"; 2.35 seconds with CPU "set"



(c) Time taken by the Neural Network: Unit increase in J increases runtime by 0.0079 seconds with GPU "set"; 0.0981 seconds with CPU "set"

Figure 9: Finding Rewards - Execution Times of Different Batch-Sizes J with GPU and CPU "set" Separately: Scaling is strongly hampered by the LP solver. With GPU "set", while each unit increase in J results in 0.0079 additional seconds for the Neural Network, it takes 1.41 seconds more for the LP.

5 Conclusion

Our models for the Identification and the Pricing Problem outperformed previously studied ones and other baseline comparisons. For the Identification Problem, the average loss value was 12% lower than the previous 2-layered model, and 35% better than the 4-layered model, giving us better results than any other model. While we did not test deeper networks, ve that using more hidden layers will only aggravate overfitting and won't provide better results - as is the case with the 4-layered network. The Pricing Problem's model also delivered better results than other baseline comparisons for reward distribution, though the difference was not drastically large.

On the other hand, we can definitively conclude that these models ran faster on the GPU than the CPU, mainly because they were based on tensors and matrices. With an approximate speedup of 6.25 for unit increase in batch-size T for the Identification Problem, we can scale to large datasets more efficiently on the GPU than the CPU. Although the Pricing Problem's model only delivered a speedup of —— (with the LP problem heavily impacting the runtime), the 2-layered network for finding rewards gave a speedup of ——. This shows that neural network are inherently quick to optimize on a GPU. One can further use a GPU-accelerated LP solver or model the LP in the network itself (if possible) to get faster results.

Figure 7a shows how the choice of Adam's algorithm (5) for Gradient-Descent($\cdot$) helps the model to learn quickly. However, we also witness long periods of saturation after few epochs. This was the case for several other algorithms (SGD (4) and Adagrad (6)), but with different paces of learning. Since the organizers would want to further optimize the set of weights even, research could be done on avoiding the long, unchanging saturation phase. This may involve using other techniques for Gradient-Descent($\cdot$) (Algorithm 2) and/or altering the loss function $Z_P(\mathbf{w_1}, \mathbf{w_2})$ (Equation 1).

One may also notice compelling arguments reflected by the results. Although these models are an *upgrade* to the previous models, they deliver exciting inferences:

- One interesting observation in Table 2 is that the Loss Value from the Proportional Distribution (0.161%) and Random Initialization (0.160%) 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 and not just rewards. In other words, incentivizing under-sampled locations more is as good as random distribution of rewards - as agents don't get more heavily influenced by rewards than any other factor to visit locations.

Moreover, by looking at the model's generated rewards, one can infer that the model chooses to place large rewards in

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

References

[1] 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 '16. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 776–785. [Online]. Available: http://dl.acm.org/citation.cfm?id=2936924.2937038

[2] ——, "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

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

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

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

[6] 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: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html

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

Appendices

A Implementation

The code can be found here[].

Both the Identification and the Pricing Problem were programmed in Python 2.7 using NumPy 1.12.1, SciPy 0.19.1 and PyTorch 0.1.12 modules [web cites] (3)(7). [Results from Python plotted in 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 are 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 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 10 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

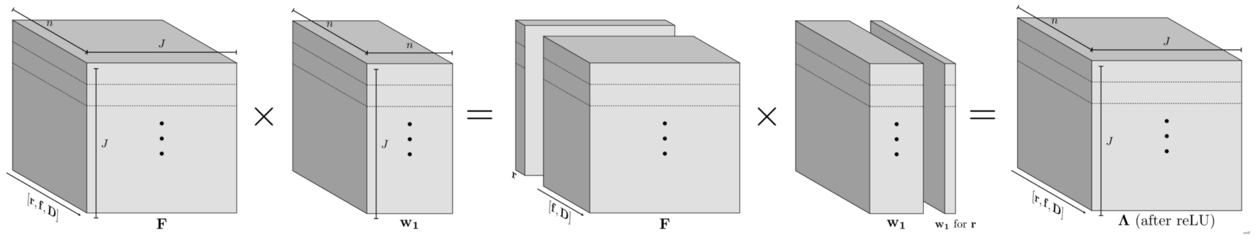


Figure 10: Splitting and Batch Multiplying **F** & $\mathbf{w_1}$

cannot multiply them with parts of **F** beforehand (Algorithm 2). However, we can combine **D** and **f** in the preprocessing stage and simply append $\mathbf{r}[t]$ on each iteration, saving computation 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 2.2.2, Equation 5 resembles the standard format more closely than 4, but it may not be clear how so.

Considering **u** and $\mathbf{r}'$ as variables **x**, Equation 5 translates into Equation 6 (J is the number of locations).

$$\begin{aligned}
&\text{minimize} && \begin{bmatrix} \mathbf{0_J} \\ \mathbf{1_J} \end{bmatrix}^T \cdot \begin{bmatrix} \mathbf{r}' \\ \mathbf{u} \end{bmatrix} \\
&\text{subject to} && \begin{bmatrix} I_J & -I_J \\ -I_J & -I_J \\ \mathbf{1}_J^T & \mathbf{0}_J^T \end{bmatrix} \cdot \begin{bmatrix} \mathbf{r}' \\ \mathbf{u} \end{bmatrix} \leq \begin{bmatrix} \mathbf{r} \\ -\mathbf{r} \\ \mathcal{R} \end{bmatrix} \\
& && r'_i, u_i \geq 0
\end{aligned} \tag{6}$$