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 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
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
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 [$\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$, $x_i \geq 0$]

Contents

List of Tables

List of Figures

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). 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 Computation Using GPUs

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 matrices, accelerated through CUDA and cuDNN, both the Identification (Section 2.1) and the Pricing Problem (Section 2.2) were formulated as 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. 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:

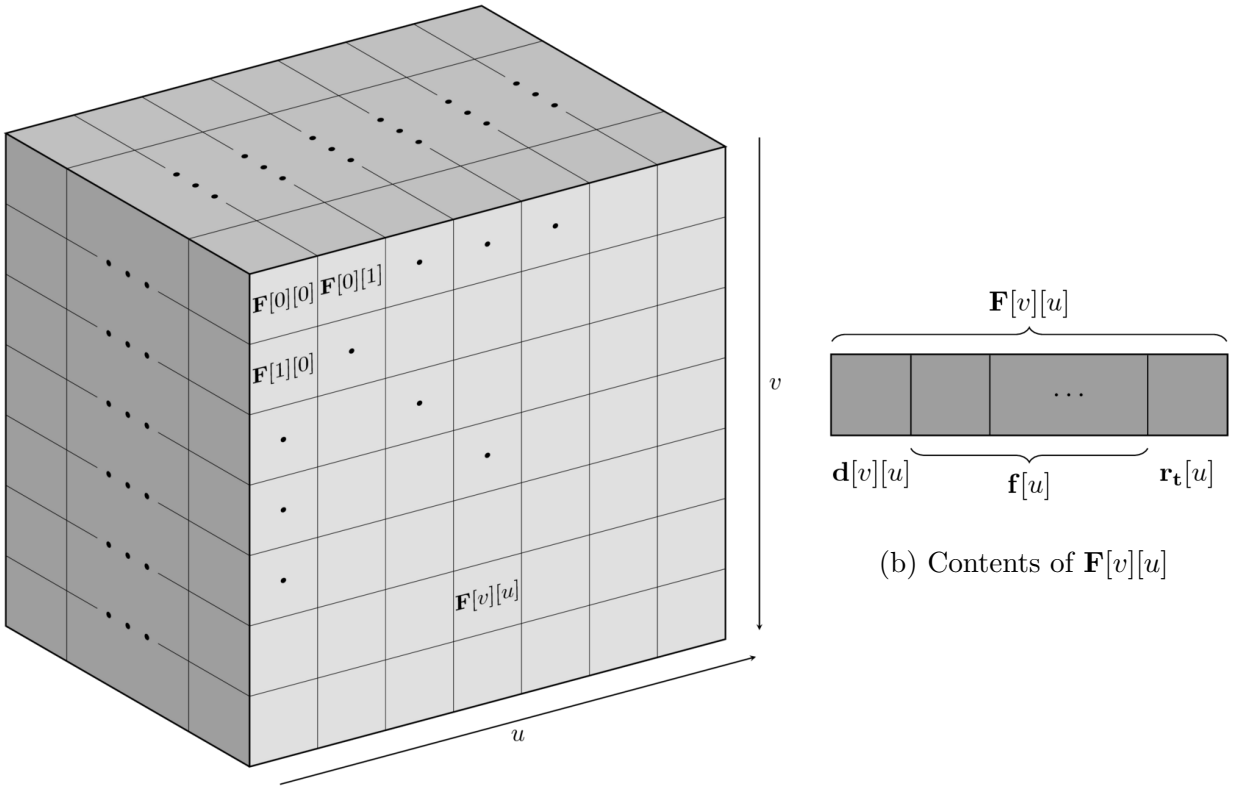
$$\underset{\mathbf{w_1},\mathbf{w_2}}{\text{minimize}} \quad Z_I(\mathbf{w_1},\mathbf{w_2}) = \sum_t (\omega(\mathbf{y_t} - \mathbf{P}(\mathbf{f},\mathbf{r_t};\mathbf{w_1},\mathbf{w_2})\mathbf{x_t}))^2 \tag{1}$$

where ω is a set of weights at time t capturing penalties relative to the importance of homogenizing at different locations and 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-epochs and promote batch operations on matrices.

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) are combined 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

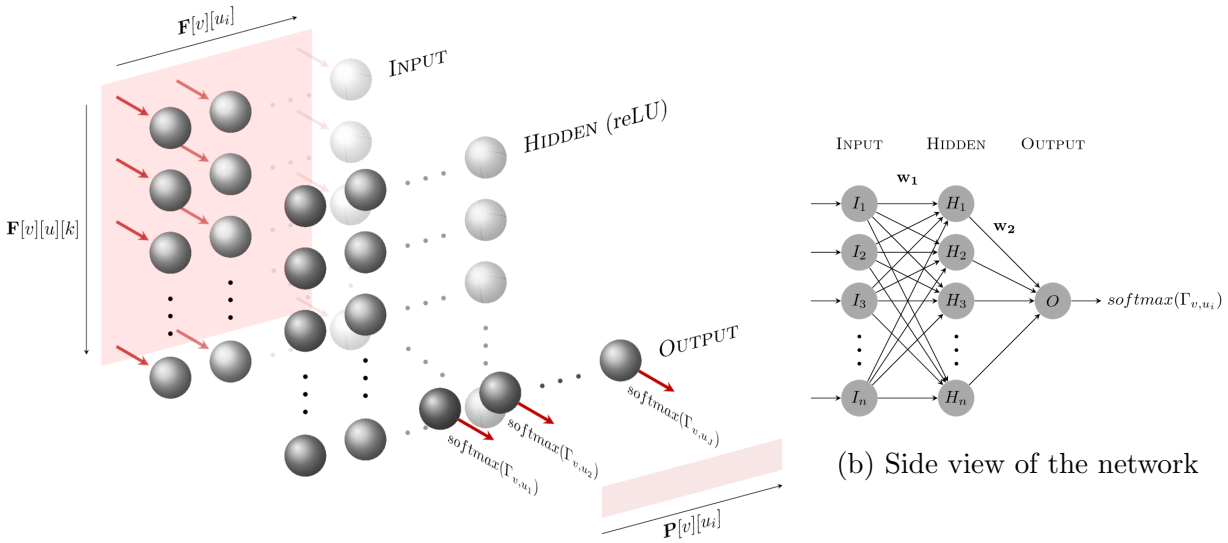
of tensors residing on CPUs and GPUs. Algorithm 1 describes the steps to construct this dataset.

Algorithm 1 Constructing the Input Dataset

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 1b
  return 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 per epoch

Require: $\mathbf{w_1}, \mathbf{w_2}, T$

function IDENTIFY-WEIGHTS($\mathbf{x}, \mathbf{y}, \mathbf{r}, \mathbf{D}, \mathbf{f}, \omega$)
  for $t = 1, 2, \ldots, T$ **do**
    $\mathbf{F} \leftarrow$ BUILD-DATASET($\mathbf{D}, \mathbf{f}, \mathbf{r}[t]$) ▷ Defined in Algorithm 1
    $\mathbf{H} \leftarrow \text{reLU}(\text{BATCH-MULTIPLY}(\mathbf{F}, \mathbf{w_1}))$ ▷ **Phase 1: Feed Forward**
    $\mathbf{O} \leftarrow \text{softmax}(\text{BATCH-MULTIPLY}(\mathbf{H}, \mathbf{w_2}))$
    $\mathbf{P} \leftarrow \mathbf{O}^T$
    $loss \leftarrow loss + (\omega(\mathbf{y}[t] - \mathbf{P} \cdot \mathbf{x}[t]))^2$
  GRADIENT-DESCENT($loss, \mathbf{w_1}, \mathbf{w_2}$) ▷ **Phase 2: Backpropagate**
  $\mathbf{w_1}, \mathbf{w_2} \leftarrow$ UPDATE-USING-GRADIENTS($\mathbf{w_1}, \mathbf{w_2}$)
  return *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. The more computationally expensive part (calculation for $\mathbf{P}$ on a CPU) 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.

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" 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 $\mathbf{x}$ is a vector rather than a matrix. Also, since the program would predict $\mathbf{y}$, it does not need labels $\mathbf{y}$ as a dataset. 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 ($\sum_i |r'_i - r_i|$)

Algorithm 3 Solving the Pricing Problem

function OPTIMIZE-REWARDS($\mathbf{x}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{D}, \mathbf{f}$)
Require: $\mathbf{r}, \mathcal{R}$
$\mathbf{F} \leftarrow$ BUILD-DATASET($\mathbf{D}, \mathbf{f}, \mathbf{r}$) ▷ Defined in Algorithm 1
$\mathbf{O}_1 \leftarrow \text{reLU}(\text{BATCH-MULTIPLY}(\mathbf{F}, \mathbf{w_1}))$ ▷ **Phase 1: Solve for P and** *loss*
$\mathbf{O}_2 \leftarrow \text{softmax}(\text{BATCH-MULTIPLY}(\mathbf{O}_1, \mathbf{w_2}))$
$\mathbf{P} \leftarrow \mathbf{O}_2^T$
$\mathbf{y} \leftarrow \mathbf{P} \cdot \mathbf{x}$
$loss \leftarrow \|\mathbf{y} - \overline{\mathbf{y}}\|/J$ ▷ J is the number of locations
GRADIENT-DESCENT($loss, \mathbf{r}$) ▷ **Phase 2: Backpropagate**
$\mathbf{r} \leftarrow$ UPDATE-USING-GRADIENTS($\mathbf{r}$)
$\mathbf{r} \leftarrow \text{LP}(\mathbf{r}, \mathcal{R})$ ▷ LP($\cdot$) explained below
return *loss*

is minimum. The mathematical formulation is given in Equation 4, which was implemented 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

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 with 1792 CUDA Cores was used for all experiments, which were run on original datasets for optimization tests and randomly generated datasets for GPU speedup tests. We believe that speedup tests on original datasets would give similar results, though we used randomly generated datasets because it was easier to scale the sizes of random data and test on a variety of dataset sizes.

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 they are 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".

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) [], Adam's Algorithm [] and RMSProp [] (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.

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 learning rate = 0.01. Since we were aiming for optimization, we ran multiple tests (5 different seeds) 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 study, and those from the 4-layered network were attained on the same original dataset with same parameter values (learning rate, 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".

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 average the results, the model was run several times on the same set of weights, using different set of weights (obtained from running the Identification Problem with different seeds) each time. Specifically, we ran the Pricing Problem's model 5 times (each with different seeds) for each set of weights, which were itself 5 in number.

Two sets of rewards were tested for loss values as baseline comparisons to our model - a randomly generated set, and another with elements proportionate 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 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 our 3-layered 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 equivalent results in both GPU and CPU "set" configurations.

4 Results

4.1 Identification Problem's Results

4.1.1 Optimization Results

Minimizing the loss function in the Identification Problem (Section 2.1) on the original dataset for 10000 epochs took an average of 1260.62 seconds with the GPU "set".

- The average loss values at the end of 10000 epochs were 0.04 on the training set and 0.14 on the test set.
- The average loss values for the 4-layered model (for comparison) with same experiment specifications were 0.14 on the training set and 0.49 on the test set.

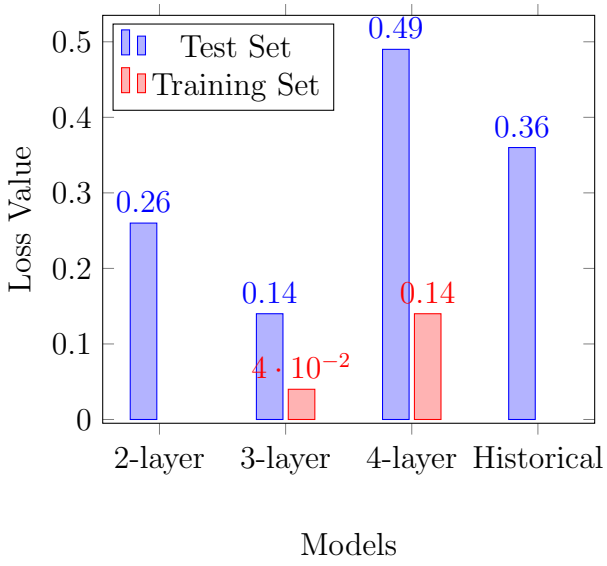


Figure 3: Comparison of Loss Values from Different Models of the Identification Problem: Loss values for the training set are inevitably lower than that for the test set, which should be the basis for comparison

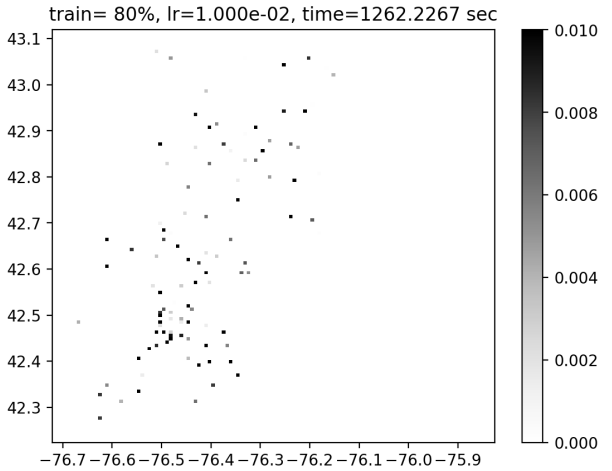
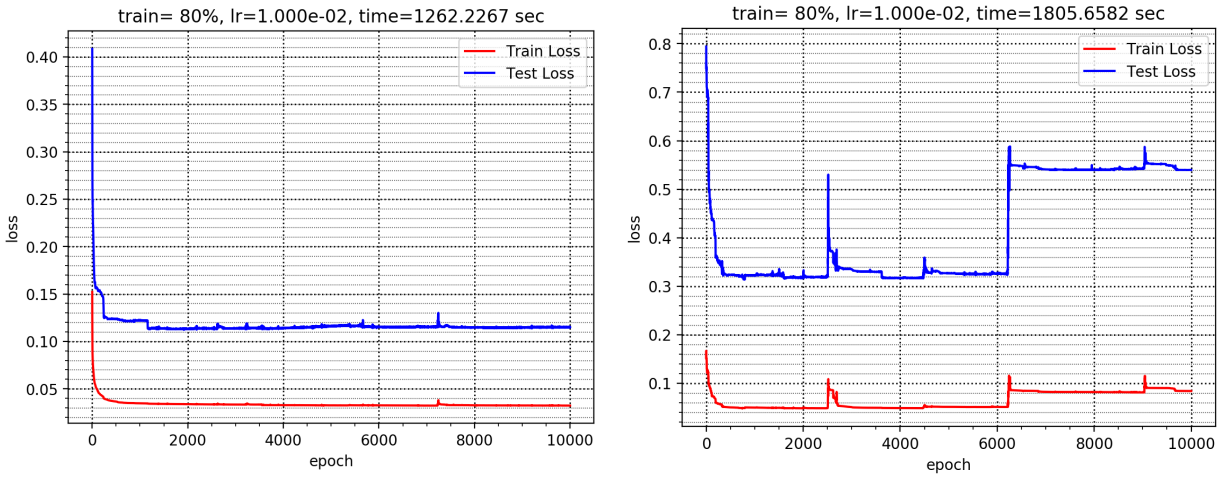


Figure 4: Predicted Probabilities of Agents Visiting Each Location Plotted on a Map (Latitude, Longitude) Representing Tompkins and Cortland Counties, NY: Dark dots represent high prediction of visits. This can be compared to the plots for the 2-layered network and other models (2, Figure 3).

As depicted in Figure 3, our 3-layered neural network outperformed the previous 2-layered model by 0.12 units (12% more closer to Ground Truth (2, Table 1)), and also produced much better results (35% more closer to Ground Truth) than the 4-layered model.

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 4 shows such a plot generated by the 3-layered network, where each dot represents a location.

Although there remained a 10% difference (0.10 loss units) in the values of training and testing set, the 3-layered model was not starkly overfitting as an average *end* difference of $10.7 \pm 5.5\%$ 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 $34.8 \pm 8.9\%$, but erratically hovering between 20% and 50%. This result is shown in Figure 5 with plots of loss values at each epoch for the 3- and the 4-layered network.



(a) Plot for 3-layered Model

(b) Plot for 4-layered Model

Figure 5: 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 ≈ 1000 epochs. This quick learning is due to the choice of GRADIENT-DESCENT$(\cdot)$ function - Adam's algorithm []. Other algorithms like SGD [] and RMSProp [] learn relatively slowly.

4.1.2 GPU Speedup Results

Running on batches of sizes $T = 17, 85, 173$ on a randomly generated dataset, with both GPU and CPU "set" separately, we obtained full execution time (including both training and testing) information. The average results (3 different seeds for each batch) are plotted in

Figure 6, which show promising GPU speedup over CPU figures for any batch size - for every unit increase in batch-size T, CPU "set" takes ≈ 4.66 seconds, while GPU "set" takes only ≈ 0.74 more seconds (slopes of best-fit trendlines).

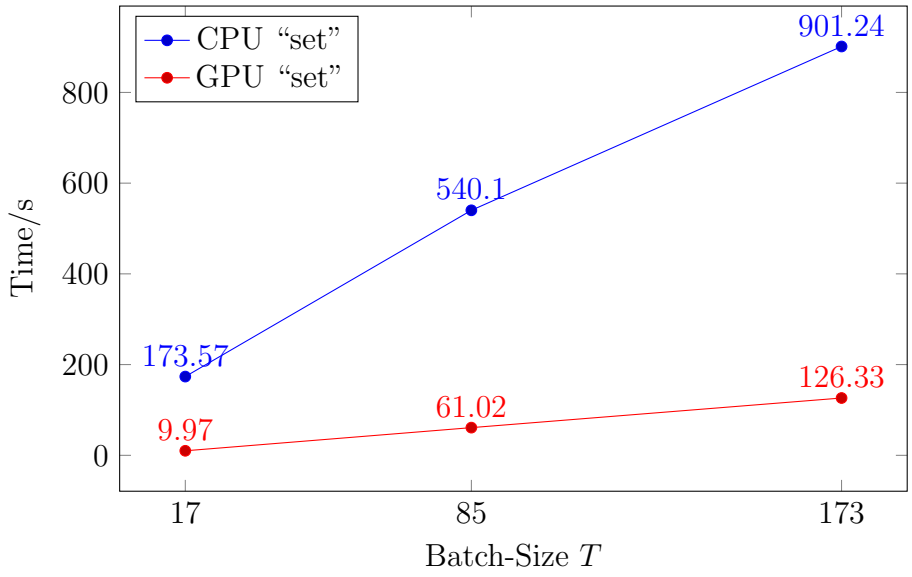


Figure 6: Finding Weights - Execution Times of Different Batch-Sizes T with GPU and CPU "set" Separately: Scaling on a GPU is more efficient (about $\frac{4.66}{0.74} \approx 6.25$ times) than doing so on a CPU.

4.2 Pricing Problem's Results

4.2.1 Optimization Results

After doing tests on the original for loss minimization and comparing with the baseline scores, we found that our model was performing ≈ 8 times better than both randomly generated and proportionally distributed rewards. Table 1 lists the average loss values obtained on each type of reward allocation (model's predicted, random and proportionate).

Table 1: 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	**Average Loss Values (In %)**
Model's Prediction	0.021
Random Initialization	0.160
Proportional Distribution	0.161

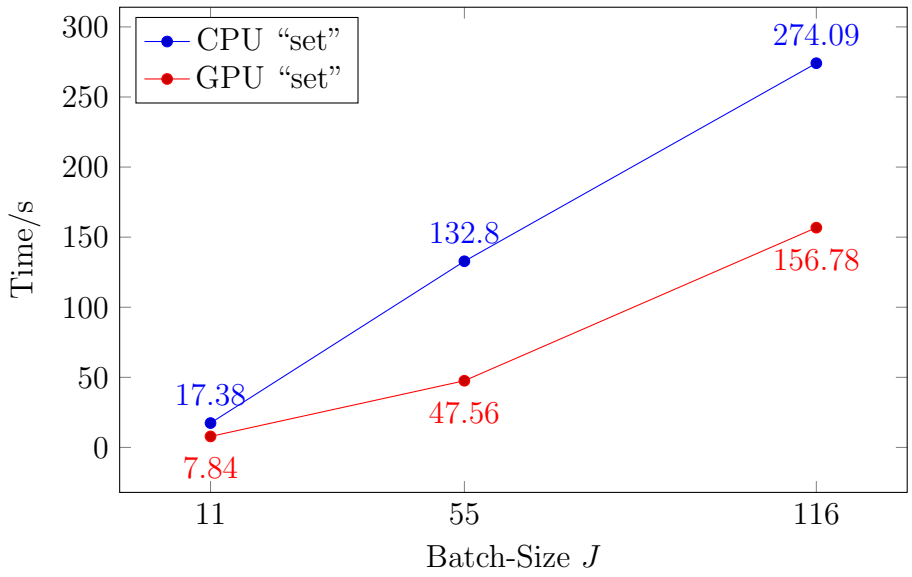
4.2.2 GPU Speedup Results

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 7a 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.

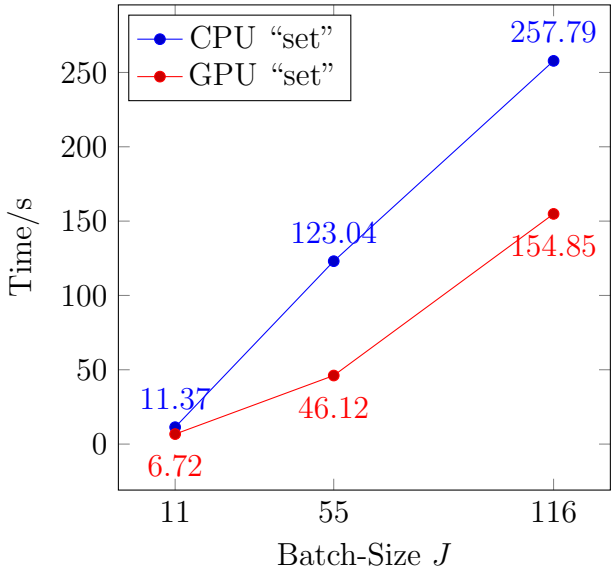
However, after realizing that the LP problem (Equations 4 & 5) might be influencing these logged times, we recorded running times for both the neural network and the LP separately. As we suspected, the LP *did* impact the runtime more than the neural networks did, and the GPU speedup for the Neural Network was expectedly high with growing batch-sizes - about 12.48 times (Figure 7c). 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 7b shows the unexpected speedup delivered by SciPy Optimize Module's LP solver.

5 Conclusion

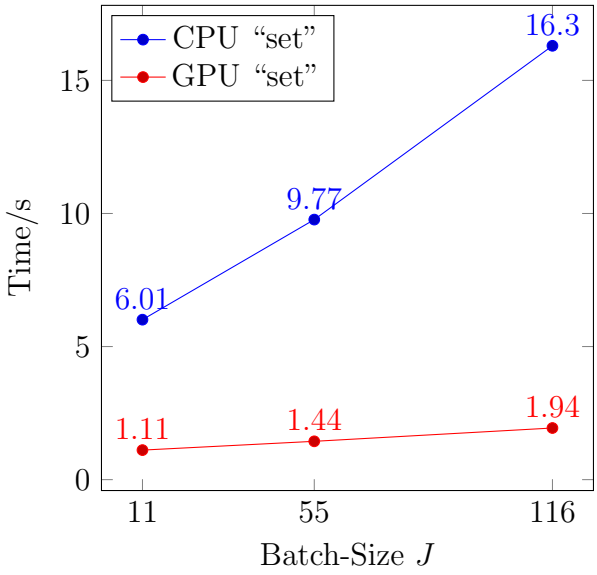
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie 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



(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"; 2.44 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 7: 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.

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] ——. 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)(4). [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 8 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

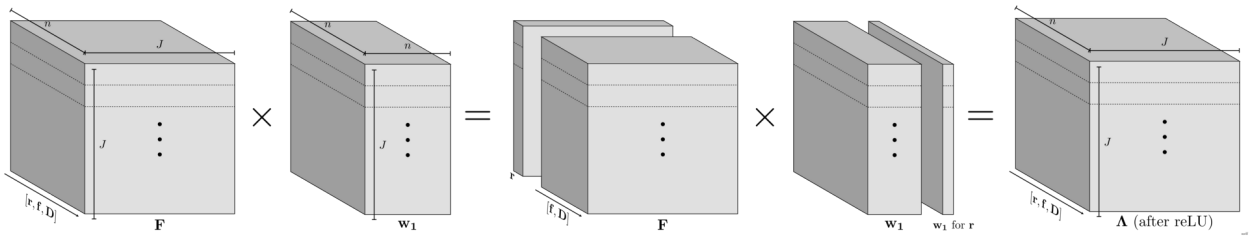


Figure 8: Splitting and Batch Multiplying $\mathbf{F}$ & $\mathbf{w_1}$

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

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