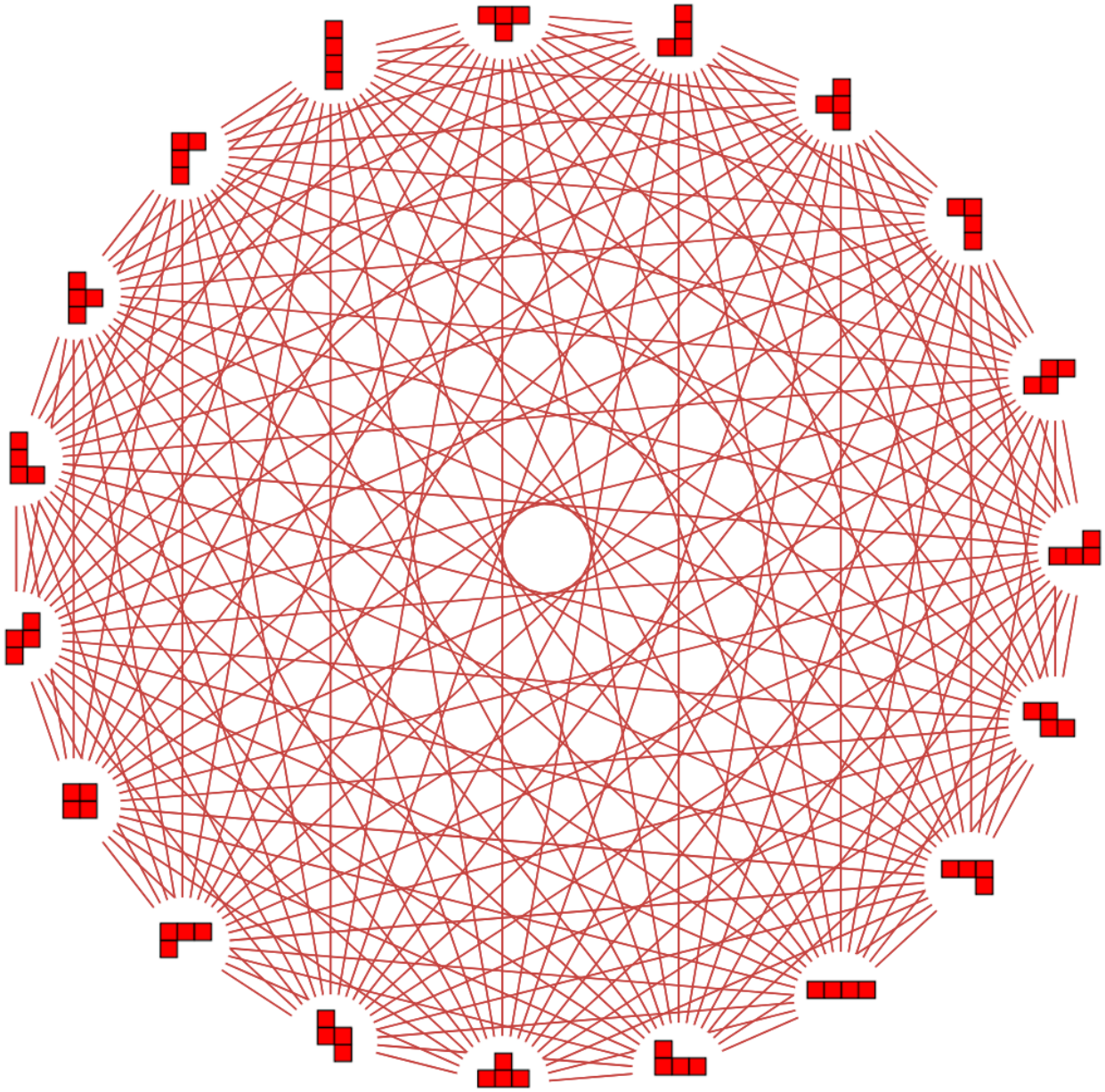
Towards Dynamic Inverse Optimization

A Data Aggregation Approach

Olaf de Vries



Delft University of Technology

TU Delft

Towards Dynamic Inverse Optimization

A Data Aggregation Approach

by

Olaf de Vries

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended on Tuesday September 23, 2025 at 14:00.

Student number:	4599241	
Project duration:	September 25, 2023 – September 23, 2025	
Thesis committee:	Prof. dr. ir. P. Mohajerin Esfahani,	TU Delft, supervisor
	Dr. P. Zattoni Scroccaro,	TU Delft, supervisor
	Prof. dr. ir. B. Atasoy,	TU Delft, committee member

TU Delft

Preface

I have always wanted to write one of these! It feels like only important people writing important documents get to do so. At the same time, it makes me happy that they are nearly exclusively used to acknowledge that writing important documents cannot be done alone. Now it is my turn to do exactly that. I don't expect to write a word of thanks many more times in my life, so at the risk of becoming sappy, please let me indulge.

I want to thank Matthijs, Kai, Max, and Jelte for proofreading my thesis. I want to thank Wito, Lisa, and especially Rozalie for pushing me to get my ADHD diagnosis. I got it late in the process, but next to it giving me a better understanding of how I work, the medicine were a life saver for the final crunch. I want to thank Bram for supporting me in the final weeks and I want to thank Marloes for looking after me and my mental health, especially after I broke my ankle. I want to thank my mom for the safety of home, and my brother for the supply of Monster Energy drinks. I want to thank Marijn for taking me into her master's thesis orphanage and letting me study with her in her office. I want to thank Robin for giving me the advice I never wanted to hear. It was always the advice I needed to hear. And of course, I also want to thank Peyman for guiding me back on track whenever I got lost, and I want to thank Pedro for the pleasant collaboration, good advice, and for infecting me with his enthusiasm. I've probably indulged a bit too much, so I shall wrap it up. Finally, I want to thank everyone who offered words of encouragement, everyone who offered to help in any way they could, and everyone who did help when I took them up on their offer.

I couldn't have done it without you.

Olaf de Vries
Delft, September 2025

Summary

Data-driven Inverse Optimization (IO) is a form of Supervised Learning where it is assumed that the output data is found by means of an optimization problem that depends on the input data. IO uses this data to approximate the optimization problem as best as possible. In the case where one wants to emulate an expert operating in a dynamic environment, the dataset obtained by measuring the expert is often contained in a small, optimal part of the total state space of the environment. When a model trained on this data finds itself in a different part of the state space, it can behave erratically.

In this thesis, we will combine Inverse Optimization with the active learning method of Dataset Aggregation (DAgger) to test if this improves model performance in dynamic settings. DAgger is an iterative process where the system is steered by the learner, creating new input data for the expert to find the best actions. This new data is then used to train a new model.

Furthermore, we propose a new algorithm, fast-DAgger, that should converge faster than the DAgger algorithm, at the possible cost of performance in the final model.

IO models trained with the DAgger and fast-DAgger algorithms are tested and compared to IO models trained on static datasets. This is done for two case studies: the Dynamic Vehicle Routing Problem as proposed by the EURO meets Neurips 2022 Vehicle Routing Competition, and the game of Tetris.

Results show the potential of combining IO with DAgger. However, DAgger is not always better than training with a static dataset. DAgger can only be helpful when the static training data is limited to a part of the total state space and when this data does not generalize well to the total state space. The fast-DAgger algorithm did not show a significant speed-up compared to the normal DAgger algorithm in the case studies. However, this is very dependent on the specifics of the model and the hyperparameters of the DAgger algorithm.

Contents

1 Introduction

With the explosion in Machine Learning (ML) research in recent years comes a similar explosion in the complexity of models trained with ML. Despite this, at its core, the goal of ML models remains simple. Given a certain input, a model should give a desired output. Given a picture of a cat or a dog, label it correctly as a cat or a dog. Given the specifications of a car (weight, number of cylinders, manufacturing year, etc.), give an estimate of its fuel usage. Given a text prompt, give an answer that makes sense and is helpful.

With Supervised Learning (SL), a model is trained by using a dataset consisting of pairs of input and corresponding desired output, such as a set of correctly labeled pictures of cats and dogs. There are many methods to train such a model, but in this thesis, we will focus on the method known as data-driven Inverse Optimization (IO). This method is most useful when the dataset of input-output pairs is found by some (unknown) optimization problem solved by an expert (Iraj & Terekhov, 2021). Take, for example, delivery drivers. Their goal is to deliver their goods as fast as possible, but the routes they choose often differ from the ones calculated by theoretical optimization algorithms. In practice, the drivers take into account extra variables such as traffic jams, safety, and difficulty to navigate certain roads. Using the locations of the goods to be delivered as input data and the drivers' preferred routes as output data, we can use IO to develop navigation algorithms that better reflect the drivers' preferences, as was done in Scroccaro, van Beek, et al. (2023).

There are many situations where the input and output data are sourced from a dynamic system. The input data reflects the current state of the system, and the corresponding output reflects the best action to take. This action then steers the system to the next state. Think, for instance, of driving a car on a racetrack with the goal of racing the fastest laps. Based on the input data, such as the current position, orientation, and velocity of the car, the driver must choose how to steer, how much gas to give, or how much to break, whether to switch transmission, etc. This optimization process happens continuously as the car progresses over the track. Since the action chosen is based on the optimization of lap time, IO can be a good method for training models.

There is a problem, however. For the best model performance, we want to sample from an expert. But an expert driver will only take optimal lines. This can create a 'garbage in, garbage out' kind of problem when we train an IO model based on this expert input-output data. When such a model finds itself on a different part of the racetrack than the optimal lines of the expert, there is no training data corresponding with the current input (garbage

in). It can therefore choose erratic and non-optimal actions (garbage out). This situation can easily happen in practice. For instance, models can never emulate the expert perfectly and will choose different actions as output. Over time, these different actions can cause the car to drift away from the optimal line, getting us into this unfortunate situation.

Dataset Aggregation (DAgger) is an algorithm that can solve this problem (Ross et al., 2011). It is a form of active learning. With DAgger, we let our model steer the car for a while, and as it drifts away from the optimal line, we record all the locations, orientations, velocities, etc., as input. We then ask the expert how it would control the car in these situations to still drive as optimally as possible. This gives us new input-output data that we can train a new model with. We can then repeat this process, each time letting the newest model control the car for a while and recording those input data points. This way, the model dynamically learns how to control the car in more situations than just the optimal lines of the original expert dataset. To the best of the author's knowledge, no research combines IO with a form of active learning such as DAgger.

In this thesis, we will therefore investigate this. We will combine the Supervised Learning method of Inverse Optimization with the DAgger training algorithm to examine whether this can create models that perform better in a dynamic environment, as compared to IO models that were trained on a static dataset. The main research question we will try to answer in this thesis is: *In a dynamic environment, can IO models trained with the DAgger algorithm outperform IO models trained with a static dataset, and if so, under what circumstances?*

Training a model with DAgger can take a long time, usually because training each new model takes progressively longer as the dataset grows every iteration. Next to investigating the main research question, we also propose a novel DAgger algorithm, called fast-DAgger. As the name suggests, this algorithm should speed up learning times, while hopefully not harming final performance.

We will test these algorithms on two dynamic problems, the video game Tetris, and a version of the Dynamic Vehicle Routing Problem (DVRP). In Tetris, the input data is the current state of the grid and which tetromino piece we have, and the output is the choice of where to drop the piece and in which orientation.

The DVRP can be interpreted as the problem that a delivery company has when it provides a same-day delivery service. As new orders come in during the day, it must decide when to drive what routes to minimize the time that delivery cars spend on the road.

The EURO Meets NeurIPS 2022 Vehicle Routing Competition (Kool et al., 2022) prescribes the DVRP variant that we will train models for. This is nice because training data was provided, and we can compare the performance of our models with those of the participants.

In chapter 2, we will discuss the theory of data-driven Inverse Optimization, which is followed by a discussion of Dataset Aggregation in chapter 3. In chapter 4 and chapter 5 we will discuss the problem definition, the IO modelling, and the results of the of the Tetris and the DVRP case study respectively. We close with a discussion of the results in chapter 6, followed by a conclusion in chapter 7.

Notation

In this thesis:

- A symbol with a 'hat' ($\hat{\cdot}$) over it denotes a recorded datapoint.
- For $N \in \mathbb{N}$, $[N]$ denotes the set $\{1, \ldots, N\}$.

2

Inverse Optimization

2.1. Definition

Data-driven Inverse Optimization, which we from now on will only refer to as Inverse Optimization or IO, is a method of making a model that imitates expert behavior (Chan et al., 2023). The key assumption of this method is that, given an input s, the expert finds its action x by minimizing some cost function:

$$\min_{x \in \mathbb{X}(s)} F(s, x). \tag{2.1}$$

We do not know what this $F(s, x)$ is, but if we can find it, we can perfectly emulate the expert. In Inverse Optimization, this is precisely what we try to do. Note that in Equation 2.1, both the objective function and the solution space can depend on the input s. If we define $\mathbb{X} = \bigcup_{s \in \mathbb{S}} \mathbb{X}(s)$, we can properly define the objective function as $F : \mathbb{S} \times \mathbb{X} \to \mathbb{R}$.

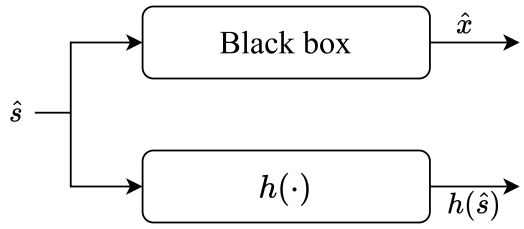
Finding $F(s, x)$ exactly is generally impossible, since the set of possible functions it could be is too massive. Therefore, we must be happy to get as close as possible. We do this by bounding the number of functions to search through by a parametric hypothesis space $\mathcal{F} = \{F_\theta : \theta \in \Theta\}$, where Θ is the parameter set. Our cost function will then become

$$\min_{x \in \mathbb{X}(s)} F_\theta(s, x). \tag{2.2}$$

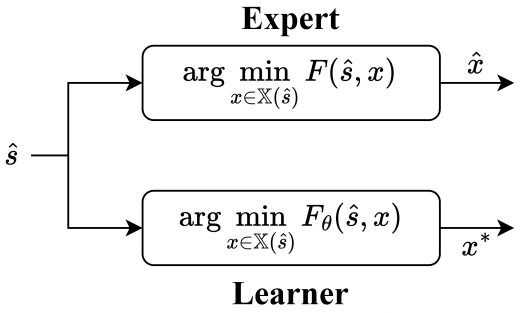
Equation 2.2 is called the *Forward Optimization Problem* (FOP). The goal of Inverse Optimization then becomes finding a θ that makes the output of our FOP match that of the expert as closely as possible. The way to do this is by creating a new optimization problem, now with $\theta \in \Theta$ as the decision variable, in order to minimize the difference between expert and learner action, given the same input. This difference is captured by some loss function $l_\theta(s, x)$. Given a dataset of expert input and action $\mathcal{D} = \{(\hat{s}_i, \hat{x}_i) : i \in [N]\}$, the *Inverse Optimization Problem* (IOP) then becomes

$$\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} l_\theta(\hat{s}_i, \hat{x}_i). \tag{2.3}$$

In the next section, we show how Inverse Optimization can be seen as a form of Supervised Learning. In section 2.3 and section 2.4, we will make the IOP concrete, by respectively defining the function space of F_θ and by choosing the loss function $l_\theta(s, x)$. In section 2.5, we show how we can solve the IOP.



(a) Given input $\hat{s}$, the black box outputs $\hat{x}$. The goal of Supervised Learning is to find a hypothesis function $h(\cdot)$ that mimics the black box as best as possible. Ideally it gives the same output as the black box for every possible input.



(b) With Inverse Optimization, the goal is the same as Supervised Learning, but now we have a more specific hypothesis function, given the assumption that the black box is an optimization problem.

2.2. Inverse Optimization as Supervised Learning

Inverse Optimization can most easily be seen as a form of Supervised Learning (SL). With SL, we have input-output data of a black box, and the goal is to use this data to create a model or hypothesis function that, given the same input, produces the same output as the black box. See Figure 2.1a. With Inverse Optimization, we do the same, but we make the extra assumption that the black box is an expert who solves an optimization problem to find the output. We then try to mimic this behavior with our own parameterized optimization problem. This optimization problem is our hypothesis function. See Figure 2.1b.

2.3. Function Space

As stated before, without restrictions, it is very difficult to find any function $F : \mathbb{S} \times \mathbb{X} \to \mathbb{R}$ for our IO purposes. There are simply too many possible functions. Instead, we will search for our IO function in a parameterized function space $\mathcal{F} = \{F_\theta : \theta \in \Theta\}$. Many parameterizations are possible (Chan et al., 2023), but we will use a function space that is affine in θ:

$$\mathcal{F} = \{\langle \theta, \phi(s, x) \rangle + f(s, x) \mid \theta \in \mathbb{R}^n\}. \tag{2.4}$$

Here, $\phi : \mathbb{S} \times \mathbb{X} \to \mathbb{R}^n$ is a feature mapping that extracts features from the input-output data (s, x). The function $f(s, x)$ can be any continuous function that is not dependent on θ. The choice of ϕ and f depends on the problem at hand and can be used to model the problem. For the case studies of Tetris and the DVRP, this choice will be discussed in detail in section 4.2 and section 5.3 respectively. Note that the function space is linear in θ, but not in the input-output pair (s, x). This means that we can model nonlinear behavior by extracting nonlinear features into $\phi(s, x)$, but it is completely up to us to define this, which can be a difficult task.

2.4. Loss Functions

The choice of loss function has a big impact on the form that the optimization problem of Equation 2.3 takes. It can change the difficulty of the optimization problem, as well as the found solution θ^*. We will shortly discuss a few possible loss functions. For more detail, see Chan et al. (2023).

Since the goal of IO is for the learner to make the same decisions as the expert, an obvious approach would be to choose the loss function as

$$l_\theta(s, x) := |F(s, x) - F_\theta(s, x)|^2,$$

which is called the identifiability loss. The closer F and F_θ are, the closer their decisions will be. On second thought, however, in most practical settings, we do not know $F(s, x)$, so it

is impossible to calculate this value. Despite this, it is still useful in settings where $F(s, x)$ is known but difficult to solve, and we want to find a new objective function that makes the optimization problem more efficient to solve. It can also be used in a theoretical setting to test different IO methods.

We usually do not know $F(s, x)$, but we do have input-output data $(\hat{s}_i, \hat{x}_i)$, so another approach is to directly compare the output $\hat{x}_i$ of the expert with the output of Equation 2.2, which we can calculate. This loss would naively look like

$$l_\theta(s, x) := \|x - \arg\min_{y \in \mathbb{X}(s)} F_\theta(s, y)\|^2,$$

but we must remember that there can be multiple minima for Equation 2.2. So instead let us define the set of minimizers as $\mathbb{X}^*_\theta(s) := \arg\min_{x \in \mathbb{X}(s)} F_\theta(s, x)$ and then we can define the predictability loss as

$$l_\theta(s, x) = \min_{y \in \mathbb{X}^*_\theta(s)} ||x - y||^2. \tag{2.5}$$

This loss is closest to the goal of IO and is shown to be statistically consistent. However, it is also shown to make the IOP of Equation 2.3 an NP-hard optimization problem, even when F_θ is convex (Aswani et al., 2018). Therefore, it can only be used with small datasets.

A more computationally attractive loss function is the first order loss (Bertsimas et al., 2014), defined as

$$l_\theta(s, x) := \max_{y \in \mathbb{X}(s)} \langle \nabla_x F_\theta(s, x), x - y \rangle. \tag{2.6}$$

Note that we can only use this if F_θ is differentiable to x. With F_θ defined by Equation 2.4, it will depend on the choice of $\phi(s, x)$ whether we can use this loss function.

Finally, the suboptimality loss is defined as

$$l_\theta(s, x) := F_\theta(s, x) - \min_{y \in \mathbb{X}(s)} F_\theta(s, y). \tag{2.7}$$

Unlike the first order loss, it does not require F_θ to be differentiable to x and is also computationally attractive, as we will show in section 2.5. This is why we will use it for this thesis. Note that, since we assume $x \in \mathbb{X}(s)$, the second term of Equation 2.7 is always smaller than the first term, and thus we do not need absolute signs.

2.5. Solving the Inverse Optimization Problem

In this section, we show our approach to solving the Inverse Optimization Problem of Equation 2.3. Specifically, we show that our choice of function space and Suboptimality Loss allows for efficient use of Subgradient Descent methods. For this, we first need a lemma.

Lemma 2.5.1. *Let the function space of Equation 2.4 be given. Assume that for every $s \in \mathbb{S}$, the action set $\mathbb{X}(s)$ is finite. Let an observation $(\hat{s}, \hat{x})$ be given. Then the subdifferential w.r.t. θ of the Suboptimality Loss, is given by*

$$\delta_\theta l_\theta(\hat{s}, \hat{x}) = \operatorname{conv}\left\{ \phi(\hat{s}, \hat{x}) - \phi(\hat{s}, x^*) \mid x^* \in \arg\min_{x \in \mathbb{X}(s)} \langle \theta, \phi(\hat{s}, x) \rangle \right\} \tag{2.8}$$

Proof. The suboptimality loss is given by

$$\begin{aligned} l_\theta(\hat{s},\hat{x}) &= \langle\theta,\phi(\hat{s},\hat{x})\rangle + f(\hat{s},\hat{x}) - \left(\min_{x\in\mathbb{X}(s)} \langle\theta,\phi(\hat{s},x)\rangle + f(\hat{s},x)\right) \\ &= \max_{x\in\mathbb{X}(s)} \langle\theta,\phi(\hat{s},\hat{x})\rangle - \langle\theta,\phi(\hat{s},x)\rangle + f(\hat{s},\hat{x}) - f(\hat{s},x) \\ &:= \max_{x\in\mathbb{X}(s)} G_{\hat{s},\hat{x}}(\theta,x) \end{aligned}$$

The result will follow directly from Danskin's Theorem (Danskin, 1967). To use this theorem, we must show that (1) $G_{\hat{s},\hat{x}}(\theta,x)$ is continuous, (2) $G_{\hat{s},\hat{x}}(\theta,x)$ is differentiable to θ for each $x \in \mathbb{X}(\hat{s})$, and (3) $\partial G_{\hat{s},\hat{x}}/\partial\theta$ is continuous w.r.t. x for all $\theta \in \Theta$.

(1) We will show that every part of the sum of $G_{\hat{s},\hat{x}}(\theta,x)$ is continuous. Firstly, $f(\hat{s},\hat{x})$ and $f(\hat{s},x)$ are continuous because f is continuous by definition. Next, since $(\hat{s},\hat{x})$ are given, $\phi(\hat{s},\hat{x})$ is a constant, and thus $\langle\theta,\phi(\hat{s},\hat{x})\rangle$ has only θ as a variable and is therefore continuous by the continuity of the inproduct. Since $\mathbb{X}(\hat{s})$ is finite, there are finitely many possible values for x in $\langle\theta,\phi(\hat{s},x)\rangle$. To prove that this inproduct is continuous, let $(\theta_n,x_n) \to (\overline{\theta},\overline{x})$ be a convergent sequence. Then, there must be an $N \in \mathbb{N}$ such that $(\theta_n,x_n) = (\theta_n,\overline{x})$ for all $n > N$, which gives $\langle\theta_n,\phi(\hat{s},\overline{x})\rangle$ for $n > N$. Now $\phi(\hat{s},\overline{x})$ is a constant vector, and since the inner product with a constant vector is continuous, $\langle\theta_n,\phi(\hat{s},\overline{x}\rangle \to \langle\overline{\theta},\phi(\hat{s},\overline{x}\rangle$, and hence this inner product is continuous.

(2) Let $x \in \mathbb{X}(\hat{s})$ be given. Now, $\phi(\hat{s},x)$ can be treated as a constant, so $G_{\hat{s},\hat{x}}(\theta,x)$ is also differentiable to θ by the differentiability of the inproduct.

(3) The partial derivative $\partial G/\partial\theta$ equals

$$\phi(\hat{s},\hat{x}) - \phi(\hat{s},x).$$

The continuity of this follows from the finitude of $\mathbb{X}(\hat{s})$ via the same logic as given in (1).

Now we can invoke Danskin's Theorem, which completes the proof. □

We can use the result of Lemma 2.5.1 to solve the minimization problem of Equation 2.3, namely by using subgradient descent methods. If, for datapoint $(\hat{s}_i,\hat{x}_i)$, we find an optimum x_i^* in the FOP of Equation 2.2, we immediately find a subgradient via Lemma 2.5.1. Let g_i be such a subgradient. Then, via the Subgradient Descent method, an update step would be

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{N}\sum_{i=1}^{N} g_i, \tag{2.9}$$

where η_t denotes the step size. In this case, we would need to solve N optimization problems to acquire 1 gradient descent step. Instead, we could also randomize the order of the dataset, for each datapoint calculate a subgradient, and then update as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{N} g_i. \tag{2.10}$$

This is called the Stochastic Gradient Descent, and it works well in practice. Now we will only need one optimization per descent step.

2.5.1. State Space Partitioning

In practical settings, the state space $\mathbb{S}$ can be huge. It could happen that a model with a certain number of variables cannot perform well on all of $\mathbb{S}$. To combat this, we can make the model more complex. If it so happens that a model performs well in one region of $\mathbb{S}$, but badly in a different region, we could partition the state space into these different regions and learn separate models for each partition. The downside of this is that there is less training data per partition, so the individual models might not perform as well. We can combat this by making more training data, but this will increase the training time. However, since our choice of function space and loss function give a relatively straightforward subgradient to calculate, training times are relatively short. It might therefore be useful to experiment with state space partitioning.

3

Dataset Aggregation

3.1. Problem with Static Learning

Inverse Optimization, as discussed in chapter 2 uses a static form of learning. A dataset of expert input and output data is given, which will be used to train a model. Whenever we learn from expert decisions in a dynamic environment, this can pose problems. For instance, it can happen that the expert will keep the state of the system only in a small optimal subset of the total state space. Think, for instance, of an expert driver on a racetrack. It will only take (near-)optimal lines, whereas the state space could have the car anywhere on or even off the track. In any practical scenario, no model will be exactly the same as the expert, and therefore it is unlikely that a model will always choose the same actions as the expert. Because of the dynamic nature of driving a car, mismatches like this could add up over time, which might steer the car away from the optimal line. If this happens, the system enters a state where no training data is available. The model has not learned how to act on this part of the state space, and therefore might perform badly.

Note that in this case where the dataset is only a small optimal subset of the total state space, both the training and test set will consist of datapoints in this subset. Therefore, it might seem like the trained model generalizes well when calculating the test loss, while it performs poorly in action. This is a strong indication that the dataset is not representative enough of the total state space, which means other training methods should be explored. The method we will explore is called Dataset Aggregation.

3.2. DAgger

Dataset Aggregation (DAgger) is a form of active learning, proposed by Ross et al. (2011). It is called active learning because the learning is done online. With DAgger, the model is let loose in the dynamic environment, creating new input datapoints. The expert is then asked what the optimal actions are in these situations, creating new output datapoints. In the example of the racecar, we let the model drive it for a while, creating new input data, and whenever it starts to drift off course, we let the expert driver correct course, creating new output data. These datapoints can then be used to train another model, that learns how to deal with drifting off course. This process can then be repeated. We let the new model drive for a while, and if it drives off course at another point of the track, we let the expert driver recover. At every iteration of this proces, we collect more data, making the next model more robust and improving the performance in the real world.

To be able to apply the DAgger algorithm, two criteria must be met. The first criterium is that dynamics must be present in the system, i.e., the action x at time t must influence the state s of the system at times $> t$. If there are no dynamics, there is no way for an expert to course correct. The second criterion is that we must have access to the expert. We need the ability to create output data, given new input data. In practice, this is expert is often a hard-to-solve optimization problem. These optimization problems cannot find an optimum fast enough in real time. We can therefore use IO and DAgger to turn this difficult optimization problem into a more easily solved optimization problem, while still finding near-optimal actions. In this case, the expert is the difficult algorithm, which we have access to.

3.2.1. Algorithm

The DAgger algorithm works as follows: We have an empty dataset $\mathcal{D}$ and an initial guess for θ, θ_1. We simulate the dynamic problem for T time steps, where the action each turn is defined by the current θ_i. For each timestep, we save the state of the system and let the expert choose the best action. These input-output datapoints are added to the dataset $\mathcal{D}$, and a new θ_i will be calculated using the expanded dataset. This process is then repeated for N steps, giving us N iterations of θ_i, and a dataset of size NT. The full algorithm is described in Algorithm 1.

Algorithm 1 DAgger

Initialize $\mathcal{D} \leftarrow \emptyset$
Initialize θ_1 to any $\theta \in \Theta$
for $i = 1 \dots N$ **do**
  Initialize $\mathcal{D}_i \leftarrow \emptyset$
  Initialize $\hat{s}_1$ to any $s \in \mathbb{S}$
  for $j = 1 \dots T$ **do**
    Calculate x_j^* using current model defined by θ_i
    Find $\hat{x}_j$ using expert
    $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{(\hat{s}_j, \hat{x}_j)\}$
    Find next state $\hat{s}_{j+1}$ using $(\hat{s}_j, x_j^*)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$
  Train θ_{i+1} using updated $\mathcal{D}$ and θ_i as initial guess.

Note that using θ_i as an initial guess for calculating θ_{i+1} is not strictly necessary. However, since it is our best guess at that time, it is useful to do so. Assuming that the training time of θ_{i+1} scales linearly with the size of $\mathcal{D}$, the total training time scales with $\frac{1}{2}N(N+1)T$. If this training is time intensive, this can limit how many DAgger iterations can be performed.

3.3. Fast-DAgger

There is another problem with the DAgger algorithm. Next to the training of θ_i scaling badly with the amount of DAgger iterations N, the first part of the dataset, $\mathcal{D}_1$ will be used for training N times. Each subsequent iteration will be less useful for improving the final model, while slowing down the training time. The most useful information for the model θ_{i+1} to train on will be inside $\mathcal{D}_i$, since this contains the new data that has not been trained on before. To improve this, we propose a new algorithm we call fast-DAgger. With fast-DAgger, we do not aggregate a dataset $\mathcal{D}$ to train a model on at each iteration. Instead, we only train on dataset $\mathcal{D}_i$ at iteration i. The full description of fast-DAgger can be seen in Algorithm 2

In the case of fast-DAgger, the total training time to find θ_{N+1} scales with NT. This can significantly improve training time compared to DAgger.

Algorithm 2 fast-DAgger

Initialize θ_1 to any $\theta \in \Theta$
for $i = 1 \dots N$ **do**
  Initialize $\mathcal{D}_i \leftarrow \emptyset$
  Initialize $\hat{s}_1$ to any $s \in \mathbb{S}$
  for $j = 1 \dots T$ **do**
    Calculate x_j^* using current model defined by θ_i
    Find $\hat{x}_j$ using Expert
    $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{(\hat{s}_j, \hat{x}_j)\}$
    Find next state $\hat{s}_{j+1}$ using $(\hat{s}_j, x_j^*)$
  Train θ_{i+1} using $\mathcal{D}_i$ and θ_i as initial guess.

Note that when training θ_{i+1}, we must take θ_i as initial guess. This ensures that there is some memory in the training. Intuitively, when θ_2 is trained on $\mathcal{D}_1$, it will not make the errors of θ_1 when we simulate the actions for T iterations. It will make new prediction errors specific to θ_2. Since we use θ_2 as an initial guess to train θ_3, we do not need to use $\mathcal{D}_1$ again for this training. In practice, this is of course not completely the case, but the training of θ_3 on $\mathcal{D}_2$ still contains at least some information on $\mathcal{D}_1$, which is 'saved' in the initial guess of θ_2. Over multiple iterations, more and more of this information is forgotten by the current model, so it is difficult to give any convergence guarantees, and it may be possible that there is no convergence. For instance, one could imagine that θ_i keeps going in circles.. However, as we will see, this is not the case for both case studies.

Furthermore, since we alter the algorithm from the DAgger algorithm of the original paper (Ross et al., 2011), we also lose the performance guarantees proven there. A theoretical analysis of the fast-DAgger algorithm falls outside the scope of the thesis, but if the practical performance is good, it could be a worthwhile investigation.

Comparing fast-DAgger to DAgger, it is important to note that, even though the training time of θ_i is significantly decreased, we need to query the expert NT times for both algorithms. If this querying takes much longer than the training of the iterative models, fast-DAgger will not deliver a significant time improvement.

3.4. Expected Ideal Performance

We want to conclude by explaining the expected performance of the proposed training algorithms compared to training a model with a static dataset. In Figure 3.1, for each of the training algorithms, the expected loss is shown as a function of the training iterations. Note that this is in the ideal case when the DAgger and fast-DAgger algorithms are most useful.

First, we should note that Figure 3.1 only shows what we expect qualitatively. We should not expect the losses to be so smooth with the case studies. Such a smoothness is never the case with real models. This is especially true when, like with our case studies, a stochastic subgradient descent method is used for training our models.

From Figure 3.1, we can see that the DAgger algorithm outperforms static training methods. This was for instance experimentally verified in Ross et al. (2011) for suitable case studies. We can also see that the DAgger algorithm outperforms the fast-DAgger algorithm eventually. We expect this because the DAgger algorithm keeps all the previous datapoints in its dataset for every training iteration. It therefore has a perfect memory of all previous training steps. The fast-DAgger does not have some memory, but loses this information over the training

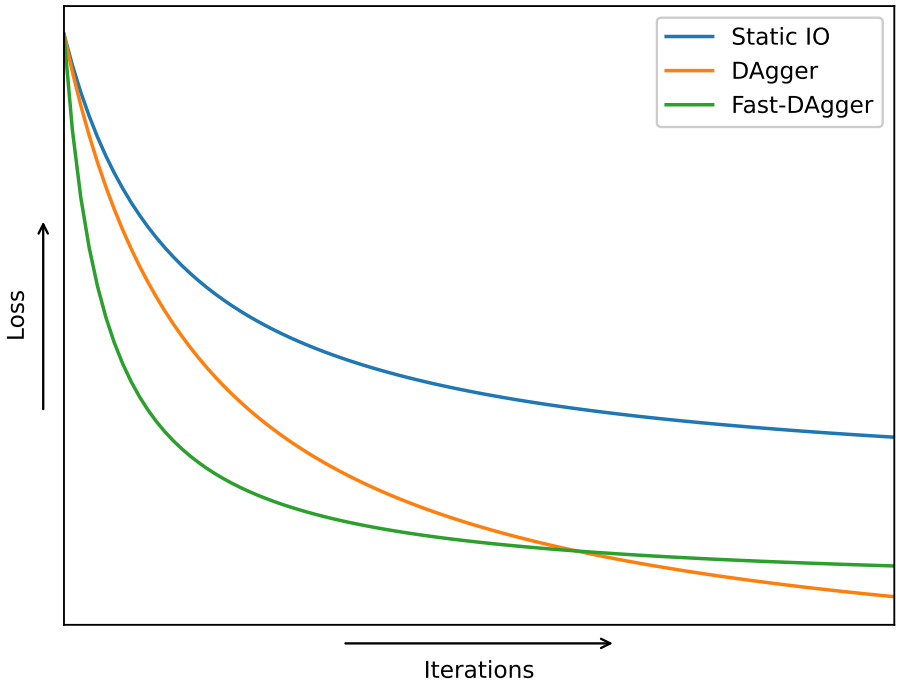


Figure 3.1: The expected loss of the models trained with static IO, DAgger and fast-DAgger over the iterations. This is the ideal use case of the DAgger and fast-DAgger algorithm.

iterations. However, since the fast-DAgger algorithm trains on more diverse data, we do expect it to outperform the static training algorithm.

We also show in Figure 3.1 that we expect the fast-DAgger algorithm to outperform the DAgger algorithm initially. This is because at every iteration, it has only new data to train on, that specifically address the biggest mistakes of the last iteration. At every iteration of the DAgger algorithm, there is also all the data from previous iterations that slow down the learning rate. Every iteration these older datapoints are used, there are diminishing returns, giving a slower overall convergence rate. Note that, depending on how the iterations are calculated, this effect may be very small or not even there. If one iteration is over the complete available dataset for instance, the standard DAgger algorithm might even converge faster, since it has more data than the fast-DAgger algorithm to train on. If an iteration is over the same amount of datapoints however, we would expect the difference in convergence rate to be more pronounced. In fact, looking only at the iterations gives a skewed view of how fast the algorithms converge. If we look at the training time, these bigger DAgger datasets are exactly what slow down the training. Therefore, the difference in convergence rate should be more visible if we plot the loss over training time instead of the iterations. We will do so in both the case studies.

4

Tetris

In the first case study, we will use IO to create a model that plays the game of Tetris. This is a dynamic environment, where an expert is likely to play in such a way that not the entire state space is explored. This means that DAgger is a suitable approach for training IO models. In section 4.1. We explain in more detail how the Tetris version used in this thesis works. In section 4.2, we explain how to model Tetris as an IO problem. In section 4.3, we share and discuss the results of both the IO models trained on static data, and the models found using the DAgger and fast-DAgger algorithms.

4.1. Problem Statement

A tetromino is a shape where four squares are laid next to each other so that each square connects at an edge to at least one other square. There are seven possible tetrominos, discounting rotational symmetry. See Figure 4.1. The game Tetris starts with an empty grid of

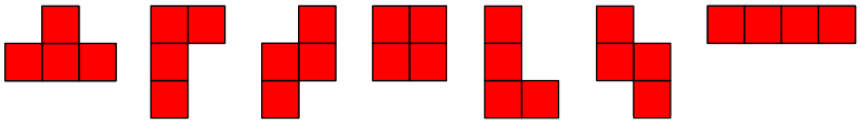


Figure 4.1: All seven tetrominoes, not taking rotation into account. From left to right, they are usually called 'T', 'J', 'Z', 'O', 'L', 'S', and 'I' pieces respectively.

10 blocks wide and 20 blocks high. One after another, tetromino pieces fall down the grid and fill it up. A falling tetromino freezes in place and becomes a static part of the grid when it can no longer fall any further. This happens when the lowest square in the grid is reached, or when it bumps into a previously filled up static part of the grid. After this happens, the next randomly chosen tetromino spawns at the top and starts to fall down. The player can choose if the tetromino is rotated by 0, 90, 180, or 270 degrees and in which columns of the grid it falls. When a horizontal line is filled completely, it disappears, and the filled squares above it are shifted down by one square. When the filled in squares in the grid are so high that a new tetromino can no longer be placed inside the grid, the game ends. The goal is to keep playing for as long as possible. As the current piece falls, the next piece is shown, letting the player plan ahead one move. See Figure 4.2 for an example of a Tetris game state.

To select the next piece when a piece is placed, one could sample uniformly from the 7 possible

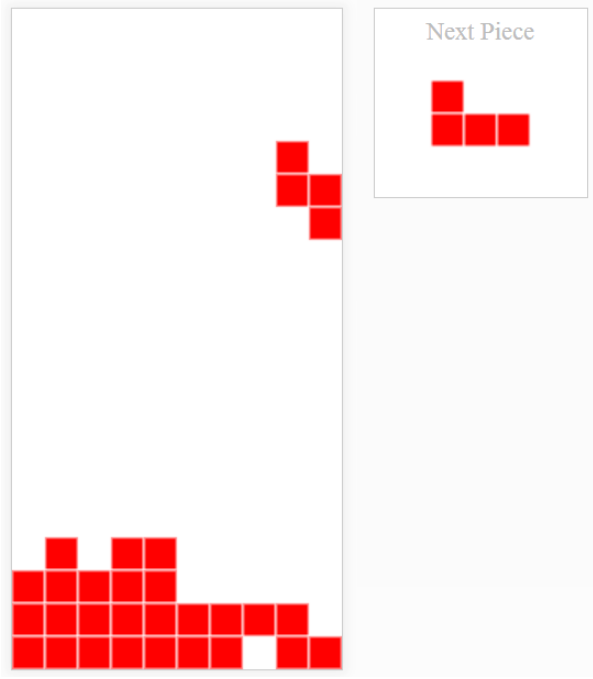


Figure 4.2: An example of a Tetris game state. The current tetromino to be placed is an S piece, and the next tetromino is a J piece. If the S piece is placed as is, the second row will be completed and thus removed. All the rows above it will be shifted down by one.

tetrominoes. However, this can lead to long sequences of the same piece, which can make the game less fun to play. In fact, it has been proven in Burgiel (1997), that a sufficiently large sequence of alternating S- and Z tetrominoes will always result in a game over. To counteract this possibility, tetromino sequences are generated differently. Instead of sampling each turn from the 7 tetrominoes, every 7 turns, a permutation of the 7 tetrominoes is sampled uniformly. This way, there can be no more than 12 tetrominoes between two of the same tetrominoes, and there can be no more than 2 of the same tetrominoes after each other.

Note that in the version of Tetris described above, when a piece is given, we decide its location and rotation, and then let it fall straight down. We do not rotate or change the tetromino as it is falling. Allowing this significantly complicates the rules governing how a falling piece becomes an inactive part of the grid. Different versions of the Tetris game solve this in different ways, each version allowing different strategies for placing tetrominos in places that are not possible with the simple rules described above. Also, in many versions of Tetris, the player can store 1 tetromino to be used later. We will also not allow this in our version of Tetris. If our model performs well in this simple version, it will perform just as well in the other versions of Tetris, since these versions extend the options the player has. The possible actions of our Tetris version is are valid actions for the more complicated Tetris versions as well.

In order to train our model, we need an expert dataset to train on. This dataset was gathered from Lee (2013), who created a bot that can play Tetris indefinitely. Our IO model is heavily inspired by the working of this bot, as we shall see.

4.2. Modelling

4.2.1. Inverse Optimization

To model Tetris as an IO problem, we must first define what the input and output are. An expert playing our version of Tetris would see the current grid, the current tetromino, and the

next tetromino. This is the input data captured in $\mathbb{S}$. The grid can be described by a matrix of ones and zeros $\{0,1\}^{20\times 10}$. The current and next tetromino can be described as members of the set $\{T, J, Z, O, L, S, I\}$. The action of the expert is the placement of the current piece. This can be represented by a choice of column and a choice of rotation, (col, rot), where $\text{col} \in [10]$, and $\text{rot} \in [4]$. Depending on the current grid and current tetromino, there can be restrictions on which values (col, rot) can take to ensure that no tetrominos are placed (partially) outside the grid, or at locations where tetrominos have been placed previously. Also, due to rotational symmetry, we can restrict the amount of possible rotations for certain tetrominos. Namely, we can restrict the rotation to $\text{rot} \in [2]$ for the S, Z, and I tetrominos, and the O tetromino to $\text{rot} = 1$.

Remember from chapter 2 that our Inverse Optimization model will take the form of the Forward Optimization Problem

$$\arg\min_{x\in\mathbb{X}(s)} F_\theta(s,x),$$

where

$$F_\theta \in \{\langle\theta, \phi(s,x)\rangle + f(s,x) \mid \theta \in \mathbb{R}^n\}.$$

To find an IO model, we must therefore define the feature mapping $\phi : \mathbb{S}\times\mathbb{X} \to \mathbb{R}^n$ and the continuous function $f(s,x) : \mathbb{S}\times\mathbb{X} \to \mathbb{R}$.

Luckily for us, the expert bot of Lee (2013) already works in this way! In short, given a game state of Tetris at timestep t, this bot tries every possible placement of the current piece and next piece, $((\text{rot},\text{col})_t, (\text{rot},\text{col})_{t+1})$, and assigns a score to the resulting grid. The placement with the lowest score wins, and the corresponding $(\text{rot},\text{col})_t$ is chosen as the action this turn. This process repeats every turn. Note that the placement $(\text{rot},\text{col})_{t+1}$ of the optimal pair at timestep t might not be chosen at timestep $t+1$, since then the new piece at time $t+2$ will be revealed, and the optimal pair $((\text{rot},\text{col})_{t+1}, (\text{rot},\text{col})_{t+2})$ might be a different placement entirely. In practice, the expert bot switches the action of $(\text{rot},\text{col})_{t+1}$ approximately 30% of the time.

When the expert tries a placement of the current and next tetromino, it assigns a score to the resulting grid by extracting features from it and then calculating a weighted sum of those features. If we define this feature extraction as $\phi(s,x)$ and put the weights in a vector θ, the weighted sum then becomes $\langle\theta, \phi(s,x)\rangle$. we have now obtained the exact form of F_θ as described above. In this case, $f(s,x) = 0$. The only difference between the expert bot and our learner bot is how the weights are calculated, and potentially the choice of features. The features extracted will be explained in more detail in subsection 4.2.2. The weights of the expert were found by a genetic algorithm, where a generation of many different weights competed with each other, and the best performers got to 'reproduce' to create even better weights in the next generation. The goal of our model trained with IO will be to mimic the expert as well as possible, not necessarily to play Tetris as well as possible. To test this, we will train a model with the exact same feature vector as the expert. We will also train a model with different features and a tweaked method of extracting features to see the performance in a more realistic setting where we do not know how the expert finds its action.

4.2.2. Features

Given a Tetris state s and an action x, the expert bot performs the actions of placing the current and next piece and uses the resulting grid configuration to extract features. Unlike during normal Tetris play, the expert does not clear completed lines when finding the resulting grid configurations. We will explain the features extracted using the example grid state of Figure 4.3. All figures explaining the extracted features are credited to Lee (2013).

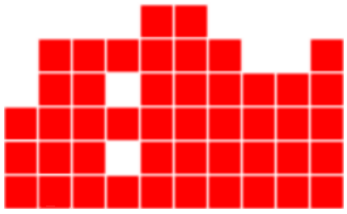


Figure 4.3: The example grid state used for explaining the features. Note that in a normal game of Tetris, this grid state is impossible since lines are cleared immediately when they are completed.

Aggregate Height

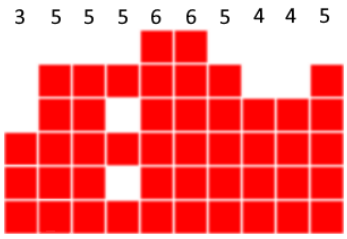


Figure 4.4: The height of the column is shown above its respective column.

The first feature to be extracted is the aggregate height of the Tetris grid. For every column, we find the height of the highest filled square and then take the sum of all these heights. This can be loosely interpreted as an integral over the grid. The goal of Tetris is to avoid a game over, so it is clear that we want to minimize this. In the example of Figure 4.4, the aggregate height would be 48.

Bumpiness

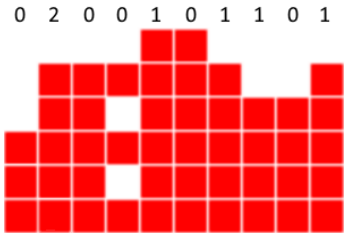


Figure 4.5: The amount of blocks a column is higher or lower than the column to its left is shown above each column.

Generally, it is easier to place tetromino pieces when the grid is smooth, which is to say there are no large 'valleys' or 'mountains'. To capture this in a feature, we define the bumpiness as the sum of the number of blocks that each column is higher or lower than the column to the left of it. In more precise terms, it is the sum of the absolute difference in height between each adjacent column. This can loosely be interpreted as the derivative of the grid state. In Figure 4.5, the bumpiness equals 6.

Holes

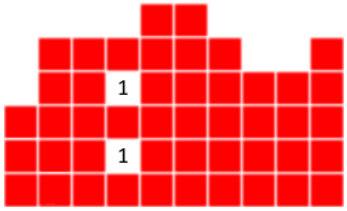


Figure 4.6: The number of holes is counted.

We define a hole as an empty grid space with at least one filled grid space above it. In the example of Figure 4.6, there are 2 holes. Holes are not good for the goal of playing Tetris indefinitely. We first need to clear the lines containing the filled blocks above the hole before we can attempt to clear the line at the height of the hole. It is clear that we want to minimize the number of holes when choosing tetromino placements. Note that if there are two empty spaces on top of each other with a filled block above it, this will count as 2 holes.

Lines Cleared

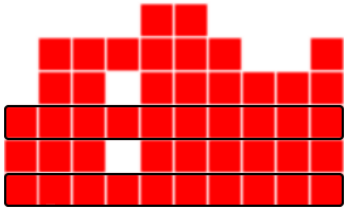


Figure 4.7: Two lines are cleared.

The only way in Tetris to delete placed blocks is by clearing full lines. This is, therefore, an obvious feature to extract. We want to clear as many lines as possible. The grid state of Figure 4.7 has cleared two lines.

4.2.3. DAgger

The Tetris bot of Lee (2013) is very good at keeping the total height, bumpiness, and number of holes low. This means that, when training data is collected from its continuous play, only a small part of the total possible state space is stored. We do not know if training a model on this data will perform well when it reaches states not represented in this training data. Since we have access to the expert and can query it online, this problem seems suitable for a dynamic DAgger approach to training. When training our model, for every action it chooses, we can query the expert to calculate the optimal action and add this to the training data for the next iteration of the model.

4.3. Results

4.3.1. Experimental Setup

When we choose the same features as the expert for our IO model to train on, both models solve the exact same FOP, up to a different weighting vector θ. It is instructive to do this to see how well IO can emulate the expert. We will call this trained model the expert model.

We will also train a different model to see the performance of IO in a more realistic scenario. We will call this model the realistic model. In this scenario, we would not know the exact inner

workings of the expert. To simulate this, we will choose an alternate feature set, and we will slightly change the inner working of our FOP. As explained in subsection 4.2.2, the expert does not clear complete lines when calculating the score for a placement of the current and next piece. In the realistic model, we will clear completed lines before calculating this score. This means that the total height will be at least 10 blocks lower when a line is cleared, decreasing this feature by a lot. Therefore, the feature that counts how many lines are cleared will be less important; hence, we will leave this feature out of the realistic model, leaving only the total height, number of holes, and bumpiness as features.

In the realistic scenario, we will add the additional goal of training a model that is faster than the expert in finding an action given a game state. The easiest way to do this is by training a model that does not take into account the next piece for executing its exhaustive search. This gives a quadratic speed up.

As we will see, both the expert and the realistic model converge very fast. When using the stochastic subgradient descent, the biggest improvement is seen within the first few descent steps. This only uses a fraction of the training set. To discuss this short term training behavior, we will show the performance of the models in these first few iterations. Additionally, there is also interesting behavior to discuss in the long term, i.e. when more training iterations have been performed. We will therefore also show the performance over more training iterations. To increase the legibility of this chapter, figures that contain the training and test loss of the models are delegated to Appendix A. We will only occasionally refer to them. We will instead focus on the performance of the models by examining how well the models can emulate the expert. We do this with the accuracy error, which is defined as the percentage of all actions our model chooses that are the same as those of the expert, given the same input.

Finally, we also want to compare the convergence speed of the DAgger and fast-DAgger algorithms to see whether the fast-DAgger algorithm realy converges faster. To do this, we will plot the accuracy error of the iterations of both models on the y-axis, and the computation time to find the iterations on the x-axis. We choose computation time on the x-axis rather than the iterations themselves because these algorithms query the expert and perform gradient descent at different times and for different amounts of data. The computation time is then a better reflection of how fast these algorithms actually converge.

4.3.2. Expert Model

Short Term

First, we will show the results of training the expert model within the first 20 iterations of the stochastic subgradient descent. In Figure 4.8, the accuracy error is plotted for the models trained with static IO, DAgger, and fast-DAgger. The size of the static training dataset contains 500 input-output pairs. Therefore, the training of the expert static IO model happens within the first IO iteration.

We want to note that, since DAgger is a method of iteratively obtaining more training data, and since we needed very little data to obtain good results for the static IO, DAgger is not likely to outperform static IO. For this model, we chose 3 DAgger iterations. Per DAgger iteration, we collect 5 input-output datapoints and perform 1 IO iteration on the resulting dataset. This gives a total of 30 stochastic gradient iterations, of which the first 20 iterations are shown. For the fast-DAgger algorithm, we generate 1 datapoint per iteration.

We can see that at the second iteration, the accuracy error of all models has already fallen below 20%. Apparently, when the models and the expert that it was trained on work in exactly the same way, just taking a step in the right direction of the true expert weighting

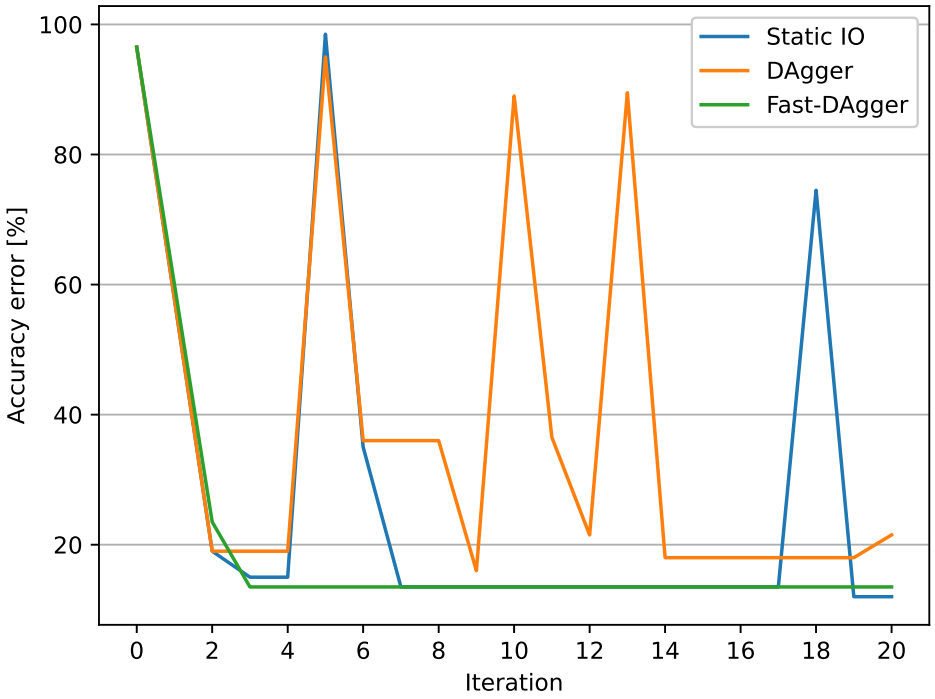


Figure 4.8: The accuracy error is plotted over the first 20 iterations of training an expert model using the static IO, DAgger, and fast-DAgger algorithms.

vector θ already ensures a good performance. At multiple iterations for the static IO model and the DAgger model, we can see spikes in the accuracy error. This is most likely due to the stochasticity of the gradient descent. This especially happens when a weight switches from positive to negative or vice versa. For instance, instead of trying to minimize the total height, it now tries to maximize it. One can imagine that this significantly reduces performance. These errors are corrected very quickly, though.

Curiously, there are no spikes for the fast-DAgger model. From iteration 3 to 20, nothing changes in performance. This also happens for the static IO model between iterations 7 and 17. From Figure A.1 and Figure A.2 in Appendix A, we can see that there are no changes in the training and test losses during these iterations either. This lack of change happens because, at every iteration, the stochastic gradient descent only takes one data point. If the expert placement decision $\hat{x}$ is the same as the model decision x^*, the model does not need to be improved. More technically, since $\hat{x} = x^*$, the subgradient of Equation 2.8 found by our algorithm will be 0, giving a 0 step update for θ_t. This is not a rare occurrence. In fact, during the 17 iteration streak of correct predictions of the fast-DAgger model, the accuracy error is 13.5%. The chance of 17 correct predictions occurring with this error rate is about 8.5%. This is not high, but well within the range of possibility.

To compare the convergence speed of the DAgger and fast-DAgger algorithms, we have plotted the accuracy error of both models over time in Figure 4.9. Looking at Figure 4.9, we first note that both models converge equally as fast in the first few iterations, implying that the fast-DAgger algorithm does not necessarily converge faster than the DAgger algorithm. Even though the fast-DAgger algorithm finds a better-performing model, this is mostl likely due to

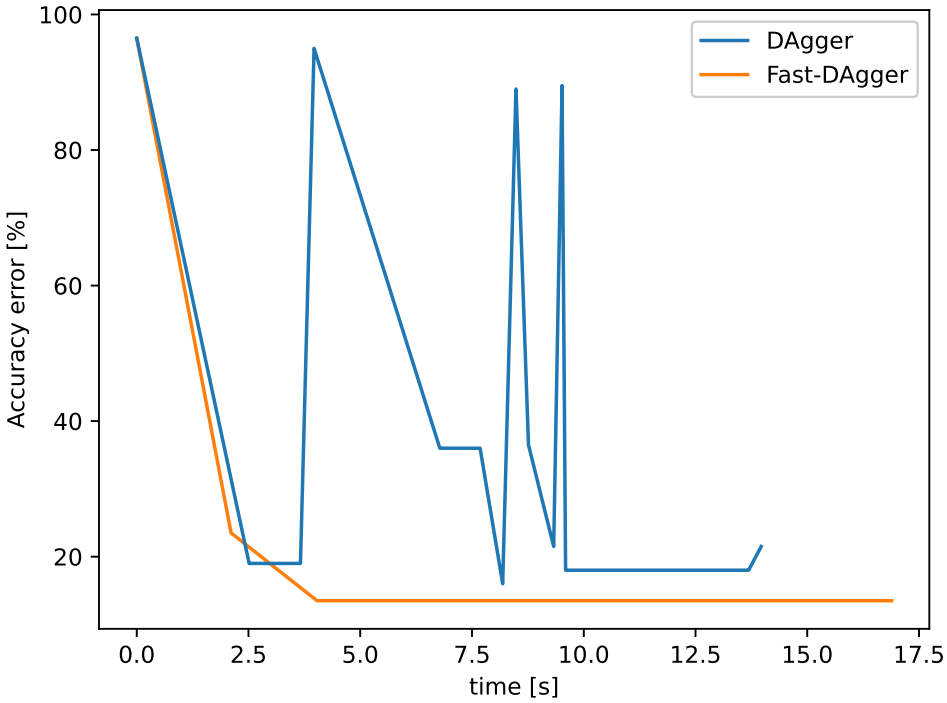


Figure 4.9: The accuracy error of the iterations of the models trained with DAgger and fast-DAgger is plotted over time.

the random nature of the stochastic subgradient descent.

It is remarkable that, despite the name, the fast-DAgger algorithm takes longer to compute 20 iterations than the normal DAgger algorithm. This can be explained by the fact that the fast-DAgger algorithm has to query the expert 20 times, whereas the DAgger algorithm only does so 15 times, since it trains on the same data multiple times. Both algorithms have to calculate 20 subgradient descent steps to get the resulting 20 iterations, implying this does not contribute to the difference in calculation time.

Long Term

In Figure 4.10, we can see the accuracy errors of the different models plotted over 500 stochastic subgradient descent steps. Of these 500 iterations, we evenly sampled 20 iterations for which we calculated the accuracy error. Since the static dataset contains 500 datapoints, the model has undergone one full IO iteration. For model trained with DAgger, there are three DAgger iterations, one IO iteration per DAgger iteration, and 100 datapoints collected per DAgger iteration. This gives a total of 600 subgradient descent steps, for which the first 500 are shown. The fast-DAgger algorithm has only one datapoint collected per iteration, so a total of 500 fast-DAgger iterations were performed to gain 500 descent steps.

Looking at Figure 4.10, we observe that the fast initial convergence has already happened before the first sampled iteration. After that, the convergence slows down significantly and essentially behaves the same for each model. This is mostly because the low accuracy error means many correct predictions, and for each correct prediction, θ_t is not updated. The accuracy error reaches 0 at approximately iteration 325 for the static IO algorithm and at iteration 380 for the fast-DAgger algorithm. When this happens, the weighting vector θ_t of the model is so close

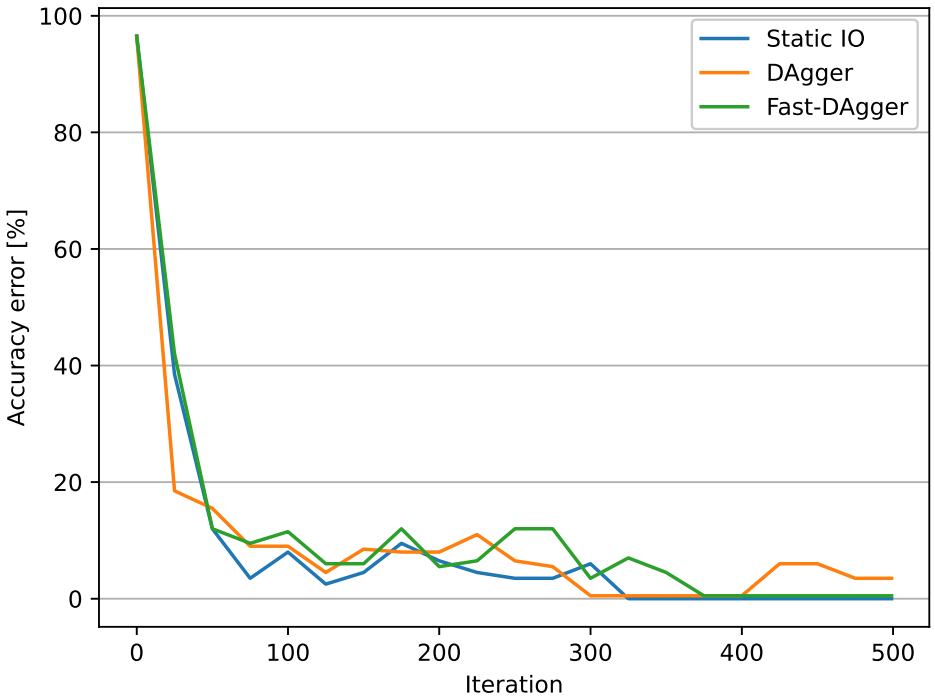


Figure 4.10: The accuracy error is plotted over the first 500 iterations of training an expert model using the static IO, DAgger, and fast-DAgger algorithms.

to that of the expert that they essentially become the same model. We have then emulated the expert as perfectly as possible, demonstrating the power of Inverse Optimization as a Supervised Learning method. The static IO models reaches the 0 error before the DAgger and fast-DAgger models. This shows that novel data outside the optimal training set is not necessary for training a good Tetris model. The static training dataset generalizes just as well as the more diverse training datasets of the DAgger models. At iteration 300, the DAgger model nearly hits the 0% error, but after 100 correct predictions, at iteration 400, it makes a mistake and diverges again. When letting DAgger run for longer, it will eventually hit the 0 error, just like the static IO and fast-DAgger.

When training multiple models, DAgger consistently reaches the 0 error slower, showing that, at least for the expert model, iterating more often over the same data is a worse choice than performing once over more different data, as both static IO and fast-DAgger do.

4.3.3. Realistic Model

Short Term

For the realistic model as described above, the convergence is still very fast, but not as fast as the expert model. We therefore show the results for the first 50 iterations. Just like the figures showing the long term training of the expert models, we sampled 20 iterations evenly. For these iterations, the accuracy error was calculated. These results can be seen in Figure 4.11.

From Figure 4.11, we notice first that fast-DAgger has the fastest convergence. However, training multiple models shows that this is nothing more than a random behavior from the stochastic subgradient descent.

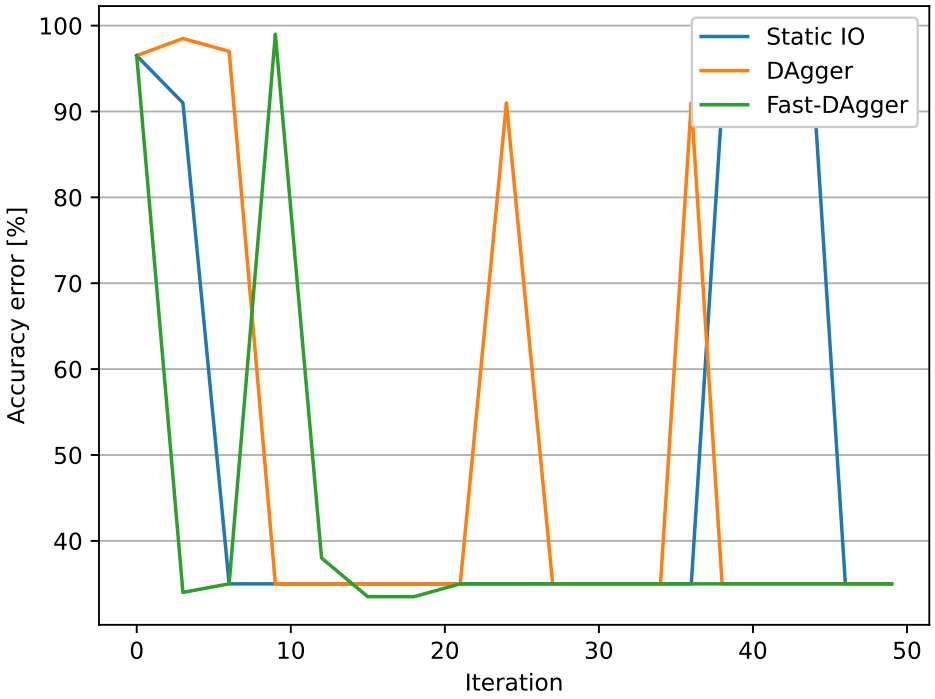


Figure 4.11: The accuracy error is plotted over the first 50 iterations of training the realistic models using the static IO, DAgger, and fast-DAgger algorithms.

Just like the expert models, there are spikes in the accuracy error during training. This happens for the same reason as it happens for the expert model. It is a more pernicious problem for the realistic models, though. We will discuss this in more detail in the discussion of the long term realistic models.

Finally, we notice that all models seem to have a maximum accuracy error of around 30%. For the fast-DAgger, for instance, it stays there for nearly the last 30 iterations in a row. With such a high accuracy error, it is extremely unlikely that the model predicts the correct expert action so many iterations in a row. Looking at the training and test loss of Figure A.9 in Appendix A, we see that they do change over the same iterations, showing that the model predicts incorrectly many times. It therefore seems that the 30% accuracy error is the best the realistic model can perform.

This makes sense when we consider the fact that the model does not take into account the next tetromino, whereas the expert does. As explained before, the expert finds two placements of a piece. One is the optimum one in the current time step, and the other is the placement it found when calculating the optimum placement of the previous tetromino in the previous time step. When the current optimal placement is calculated, it takes into account the newly gained information of what the next tetromino is. This new information can cause it to choose a different placement of the current piece than the placement it found in the last turn. In reality, the expert 'switches' choice for a given tetromino 30% of the time. Since our realistic models do not take into account the next piece, we cannot hope for their performance to improve past this 30% accuracy error limit. To the realistic model, these switched placement seem like noise. Curiously, the fast-DAgger model seems to dip below this limit for a few iterations. This is

most likely just a coincidence, where it performs just slightly better on the test set by pure chance.

In Figure 4.12, we have plotted the accuracy error of the DAgger and fast-DAgger iterations over the computation time of said iterations.

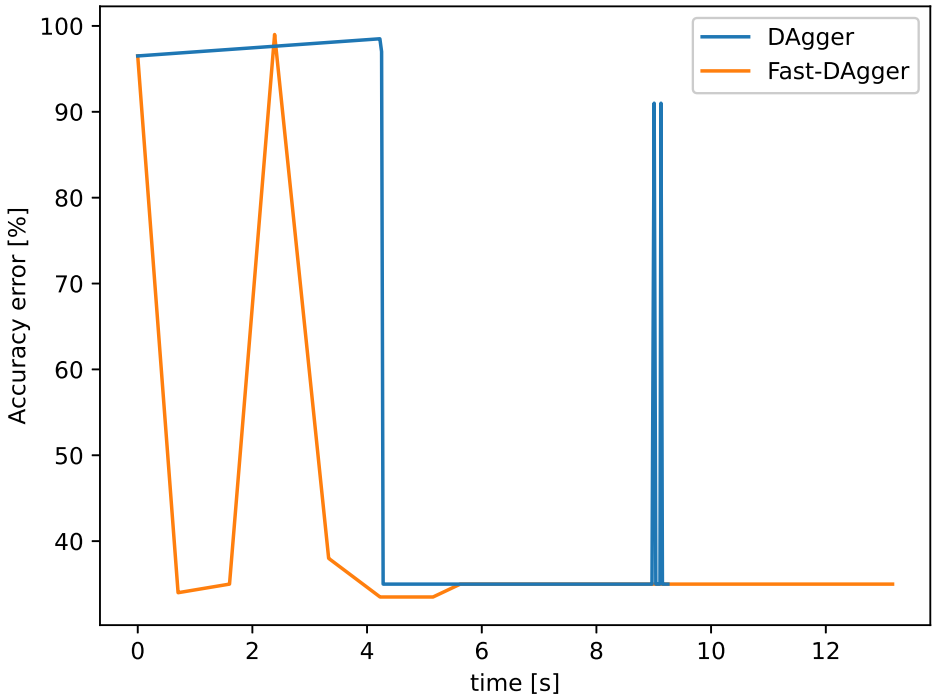


Figure 4.12: The accuracy error of the iterations of the models trained with DAgger and fast-DAgger is plotted over time.

From Figure 4.12, we can see that the fast-DAgger algorithm seems to converge faster than the normal DAgger algorithm, but as we said before, this is most likely due to the stochasticity of the subgradient descent.

Furthermore, we can clearly see how both algorithms differ in their order of querying the expert and calculating subgradient descent steps. In a single fast-DAgger iteration, the model chooses one action, queries the expert, and then calculates a subgradient descent step. This gives evenly spaced update steps. In a single DAgger iteration, the model first chooses actions for multiple consecutive time steps, queries the expert for these steps, adds this data to the total dataset, and then performs the subgradient descent over the aggregated dataset. Since the model is much faster than the expert, querying the expert is the bottleneck in this algorithm. In Figure 4.12, we can see that the DAgger algorithm queries the expert between 0 and 4 seconds. Then a few gradient descent steps are performed quickly, and between 4 and 9 seconds, the expert is queried again. This is followed again by a few subgradient descent steps, in which we can see a few spikes in short succession.

Again, the fast-DAgger takes longer to execute in total, since it needs to query the expert more times.

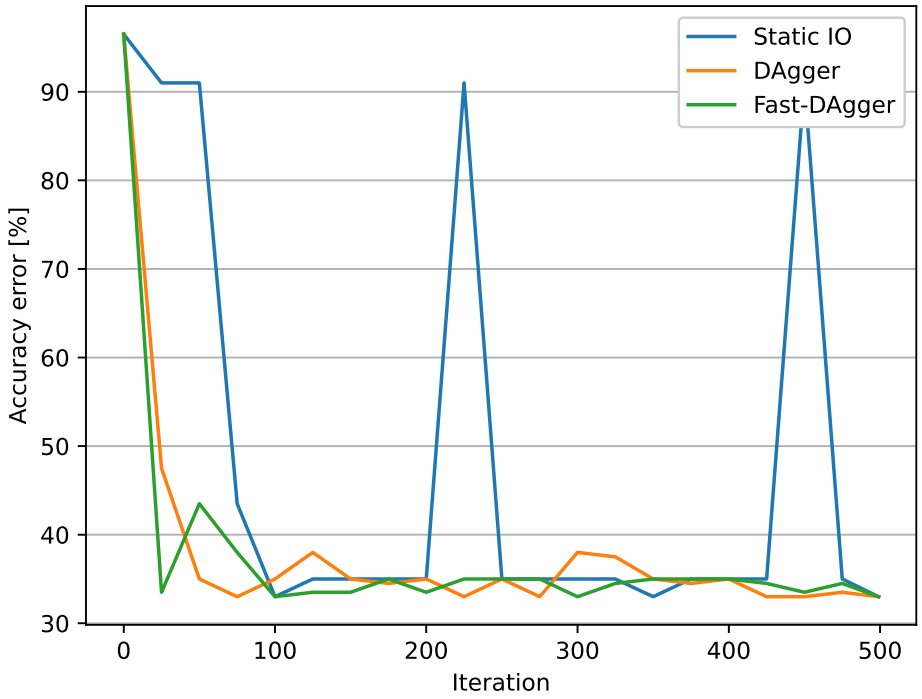


Figure 4.13: The accuracy error is plotted over the first 500 iterations of training an expert model using the static IO, DAgger, and fast-DAgger algorithms.

Long Term

In Figure 4.13, the training of the realistic model is shown for 500 iterations. Again, we have evenly sampled the iterations and calculated the accuracy error for those iterations.

From Figure 4.13, we see that, despite having longer to calculate, the models cannot get below the 30% accuracy error.

We also see some spikes in later iterations for the static IO model. These spikes also happen for the DAgger and fast-DAgger model, however, they happen in between the sampled iterations, so they are not shown in the figure. Like the expert model, one would expect these spikes to occur only in the short term training, as the step size is still large, and the model is still converging to a set of weights. This is not the case however. No matter how long we iterate, these spikes keep happening. Also, playing around with how fast the step size η_t converges will not change this. What happens is that, while the accuracy error stays about constant over the iterations, the weighting vector θ_t does converge. And it converges to a value where some of the weights are very close to 0. So close that a bad stochastic descent step can push one of these weight to a negative number, which makes the performance very bad immediately. We conclude that θ_t converges to weights that are very sensitive to small changes. There is a large set of weights that perform at or near the 30% accuracy error limit, so it is unclear why θ_t converges to these specific weights. Maybe it is just where the optimum lies, but there could be other explanations. For instance, it could be that the subgradient descent converges to a point at the boundary of this weight set, which happens to be in a very unstable point. At the very least, it warrants a closer examination, which falls outside the scope of this thesis.

4.3.4. Model Performance

Finally, we want to see how well the models actually perform at Tetris. For this, we simulated a game of Tetris and let the models play it. When a thousand lines were cleared, we considered the bot as being able to play indefinitely.

All the expert models could play indefinitely. This is not unexpected, since they can emulate the expert quite well. The expert can, of course, play indefinitely.

All the realistic models could play indefinitely as well, though it was not necessarily the final iteration that could do so. This is not what we want of course, but due to the sensitivity of the weights, the final iteration could very well perform badly. However, we found that there are always previous iterations that can play indefinitely.

5

Dynamic Vehicle Routing Problems

5.1. Problem Statement

Vehicle Routing Problems (VRPs) are a class of optimization problems that are important in the field of Operations Research. Many variants exist, see for instance Mor and Speranza (2022). We will only discuss the variants relevant to this thesis. The easiest way to think of a VRP is to think of the problem that arises during the so-called 'last mile' of a delivery chain. In this last mile, a fleet of vehicles must deliver packages to customers from a central depot. The objective is to find the routes that minimize the cost of distribution. See Figure 5.1 for an illustration. In the next sections, we will more formally define these problems and incrementally build up to the Dynamic Vehicle Routing Problem of interest, namely that of the EURO Meets NeurIPS 2022 Vehicle Routing Competition (Kool et al., 2022), which we from now on will also refer to as the 'EURO Meets NeurIPS Competition'

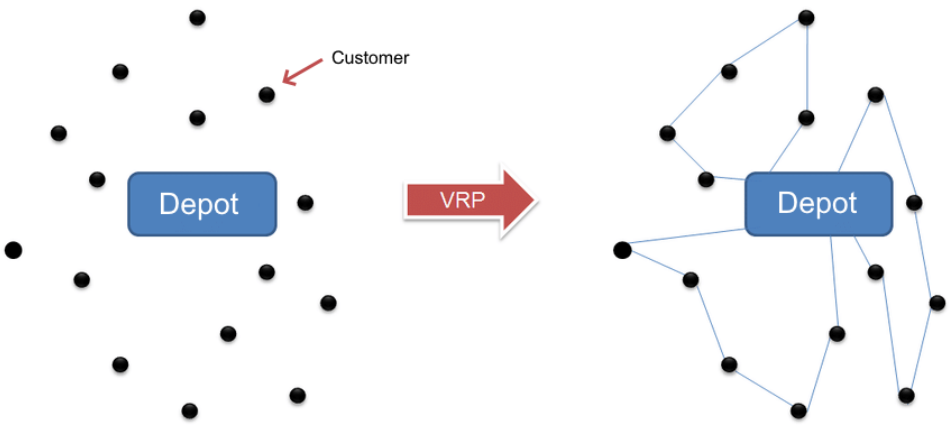


Figure 5.1: An example of a VRP. We start with a central depot, along with several customers that must be visited. The VRP attempts to find the most efficient routes.

5.1.1. Capacitated Vehicle Routing Problem

Let us start by defining Vehicle Routing Problems more formally. One of the simplest VRPs is the Capacitated Vehicle Routing Problem (CVRP). Let (V, E) be a complete directed graph with $V := \{0, \ldots, N\}$. Here, 0 represents the depot where vehicles will start and end their delivery routes, and the numbers $1, \ldots, N$ represent the customer requests. Each customer requests a positive quantity $q_i > 0$, and each vehicle has a maximum capacity $Q > 0$. Let

$T \in \mathbb{R}_+^{(N+1)\times(N+1)}$ be the distance matrix, where the element t_{ij} represents the time it takes to travel from vertex i to vertex j. Note that this matrix is generally not symmetric. A feasible solution is a set of directed cycles on the graph where:

- Each cycle contains the depot.
- Each request is in exactly one cycle
- The sum of request quantities of each cycle is less than or equal to Q.

We can interpret a cycle as a route that a vehicle will drive to deliver to the requests. The objective is to minimize the total time that the vehicles are on the road. To formalize the cost function, let $x_{ij} \in \{0, 1\}$ where $x_{ij} = 1$ if edge (i, j) is on a route, and $x_{ij} = 0$ if it is not. Then, the cost becomes

$$\sum_{(i,j)\in E} x_{ij} t_{ij} \tag{5.1}$$

Many other constraints can be added. For instance, there can be a maximum number of vehicles, or there can be vehicles with different capacities. However, these will not be considered in this thesis. An extra set of constraints that will be looked at is the addition of time windows for deliveries, as will be discussed next.

5.1.2. Vehicle Routing Problem with Time Windows

In the Vehicle Routing Problem with Time Windows (VRPTW), each customer has an associated time window and servicing time. The delivery must start within the time window, and the vehicle can only leave for the next customer when the servicing time has passed. These added constraints imply that cycles in the solution must have a start point with a start time and an end point with an end time. We will assign the depot as the start and end point, so that a vehicle leaves the depot at the start time for deliveries, and returns to the depot at the end time. For customer $i \in V \setminus \{0\}$, the servicing time is denoted as s_i, and the time window is denoted as $[e_i, l_i]$, where $e_i < l_i$. A solution where a delivery starts before or after the time window is infeasible. A vehicle can arrive earlier than e_i, but it must wait until this time to start the delivery. The vehicles still have a maximum capacity Q. In the VRPTW, we still try to minimize the total time that vehicles are driving on the road. We could alternatively minimize the total time vehicles are away from the depot, which includes servicing time and waiting time, but since this is not done in the EURO Meets NeurIPS Competition, we will not do so either. The cost function is therefore the same as that of Equation 5.1.

5.1.3. Dynamic Vehicle Routing Problem

The intuition for the Dynamic Vehicle Routing Problem (DVRP) is that the delivery company offers same day delivery, and customers can request new deliveries during the day. They also have delivery time windows. This is convenient for the customers, but creates a big problem for the delivery company. With the static VRPs such as the CVRP and the VRPTW, we have all the request information in advance, meaning we can solve them in advance and have the optimal routes at the start of the day. In the dynamic variant, we no longer have all the information in advance. This makes it difficult to even define the DVRP as an optimization problem.

The first step in doing this is to define a static variant of this problem where we do have all the information in advance. The delivery company knows at what time during the day the orders will be placed. To model this, we associate a release time r_i with each request i. This indicates the time this request becomes known to the delivery company. Now, a feasible

solution becomes a set of directed cycles with start times on the graph where:

- Each cycle contains the depot.
- Each request is in exactly one cycle.
- The sum of request quantities of each cycle is less than Q.
- The delivery time of each request is inside the time window.
- The start time of each cycle is greater than or equal to the release time of each request in the cycle.

The last constraint states that we cannot start a delivery cycle before each request in the cycle is revealed to the delivery company. At the end of a day, when all the requests have been revealed, an instance of this static problem can be defined. We therefore call this static variant of the DVRP the hindsight DVRP. When we develop the DVRP that we can use during the day, we must find routes that satisfy the same constraints as the hindsight DVRP. We will call this variant the real-time DVRP.

There are multiple ways to ensure feasible solutions of the hindsight DVRP when we are in real time. We will use the way prescribed by the EURO Meets NeurIPS Competition. Here, the day is split into discrete times $\{t_1, ..., t_K\}$, where each time interval $[t_i, t_{i+1})$ is denoted as epoch i. For each epoch $i \in [K-1]$, a static VRPTW is created by the current active requests. A request j is active if it is known at the current time, (i.e., $r_j < t_i$,) and it is not part of a delivery route of a previous epoch. As the day progresses, deliveries are made, and new requests will become active. Each epoch will therefore give a different static VRPTW to solve.

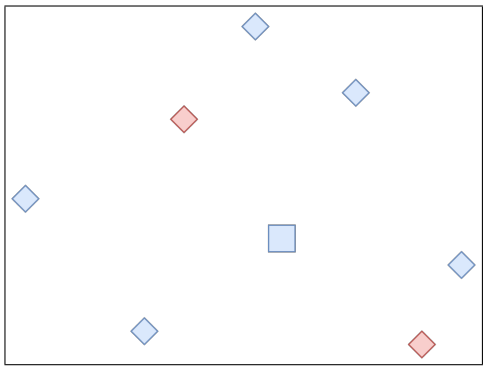
When we model the real-time DVRP like this, something interesting happens. To see this, we first note that for each customer, there is a final epoch where it must be included in a route, so that the time window constraint is not violated. This is not necessarily the same epoch when the request becomes active. Consequently, when we create the static DVRPTW of an epoch when a request is active, but it is not its final epoch yet, we have a choice. Instead of including it in the VRPTW of the current epoch, we can also exclude it and postpone it to a future epoch. As a result, we now have two problems every epoch: a static VRPTW, but also the decision problem of which active requests to include in said static VRPTW and which requests to postpone to the next epoch. Note that this decision problem is what adds dynamics to the DVRP. The decision made in each epoch influences the VRPTW instance of the next epoch. See Figure 5.2 for an example of a real-time DVRP with 3 epochs.

The real-time variant of the DVRP is the problem of interest in this thesis. As it turns out, the decision problem of deciding which active requests to add to the VRPTW of the current epoch is far from trivial to solve, mainly due to the incomplete information we have available at each epoch.

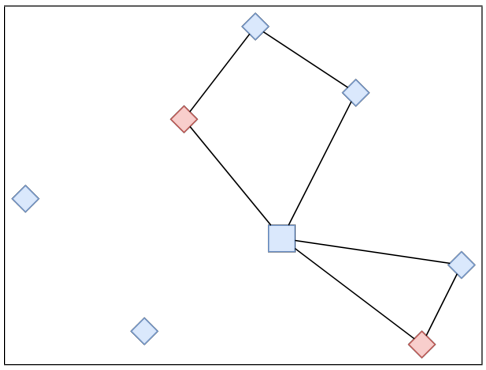
From here on out, when we refer to the DVRP, we mean the real-time variant of the problem.

5.1.4. Prize Collecting Vehicle Routing Problem

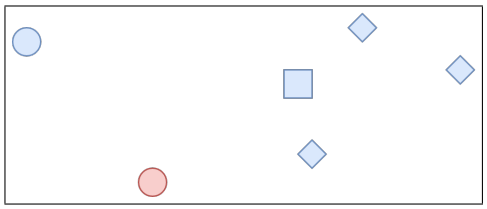
Even though we have defined the VRP of interest for this thesis, we want to discuss one more static VRP that will be used to solve the DVRP. This variant is called the Prize Collecting Vehicle Routing Problem (PCVRP). In this variant, it is no longer necessary to deliver to every customer. Instead, each customer i has an associated prize p_i. If a customer is visited, the prize is collected. The goal is then to maximize the prizes collected while minimizing the travel



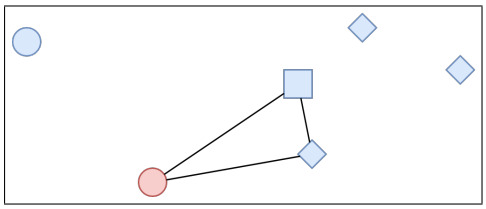
(a) Requests of epoch 1, including depot represented by a square. Note that all the requests are new, so they are displayed as diamonds. If they are not served this epoch, they will become circles. Red requests must be dispatched this epoch so as not to violate time window constraints.



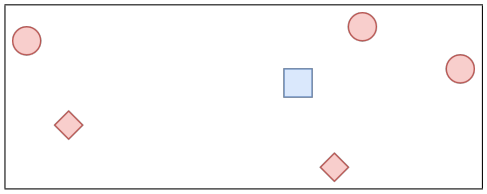
(b) Optimal route calculated at first epoch. Note that all the red requests are visited.



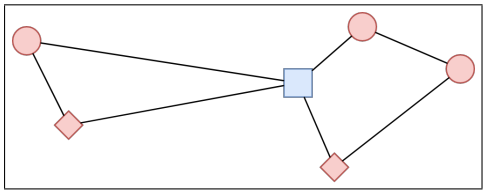
(c) Requests of epoch 2. The visited requests of epoch 1 are now inactive, and are thus removed. Three new ones have been added. The active requests that were not delivered to last epoch have now become circles.



(d) The optimal route of epoch 2.



(e) Two new requests are added for the third epoch. Note that all the requests must be delivered this epoch since it is the final epoch of the day.



(f) Optimal routes of the third epoch.

Figure 5.2: An example of a Dynamic Vehicle Routing Problem with Time Windows. The blue square is the depot, diamonds are new active requests added since the last epoch, and circles are active requests from previous epochs. The requests are red if they must be dispatched this epoch due to time window constraints.

times. The cost function then becomes

$$\sum_{(i,j)\in E} x_{ij}p_j - \sum_{(i,j)\in E} x_{ij}t_{ij}. \tag{5.2}$$

Note that the optimization problem is now a maximization problem. See Figure 5.3 for an example solution to an instance of the PCVRP. Intuitively, requests with a high prize that are close to the depot and other requests are more likely to be included in a route, whereas requests with a low prize that are far away from the depot and other requests will not be included in a route.

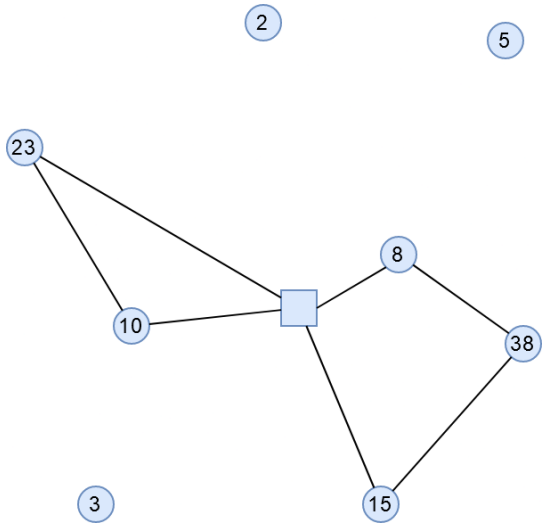


Figure 5.3: An instance and possible solution of a PCVRP. Note that not all nodes are visited. The cost of visiting those nodes is higher than the collected prize.

5.1.5. NP-hardness of VRPs

We conclude this section with a note on the hardness of VRPs. We start with a theorem.

Theorem 5.1.1. *All the VRP variants discussed in this section are NP-hard.*

Proof. We will reduce the Traveling Salesman Problem (TSP), which is well known to be NP-hard, to any of the discussed VRPs. Given an instance of the TSP with N vertices and distance matrix T, we can cast it as a CVRP by taking the same vertices and distance matrix, setting the maximum number of vehicles to 1, the request quantity to 1 for every edge, and the vehicle capacity at $Q = N + 1$. When considering the VRPTW, we additionally set the time windows large enough. For instance, let τ be the sum of all elements of T, i.e., $\tau := \sum_{i=1}^{N}\sum_{j=1}^{N} t_{ij}$. We can then set the time windows of each edge to $[0, \tau + 1]$. In the case of a DVRP, we additionally set the number of epochs to 1 by setting $t_1 = 0$ and $t_2 = \tau + 1$. We must also set all the release times to 0. In the case of a PCVRP, we set the prizes higher than the sum of all the costs, i.e., $p_i = \tau + 1$ for each $i \in [N]$. □

The NP-hardness of VRPs means that finding an optimum is very difficult. Luckily, in this thesis, we do not care about the real optima, but about solutions that are good enough. Good heuristics exist for the static VRPs (Mor & Speranza, 2022). To solve static VRPs, we will use the PyVRP Python package, which uses a state-of-the-art hybrid genetic search algorithm to efficiently find good solutions (Wouda et al., 2024).

5.2. EURO Meets NeurIPS Vehicle Routing Competition

The Dynamic VRP of interest in this thesis is prescribed by the EURO meets NeurIPS Competition in Kool et al. (2022). This specific VRP was chosen for several reasons. Firstly, real-world data was published to train models. This will allow us to and test test our IO models in a realistic scenario. Secondly, since the competition is over, the results of the contestants are known. We can therefore easily compare the results of our method to the results of the participants. Finally, one of the goals of the competition is to stimulate the use of Machine Learning methods in Operations research. Since IO can be seen as a supervised learning method, the competition is well suited for our needs. We will now discuss the VRP as prescribed by the EURO meets NeurIPS Vehicle Routing Competition in detail.

5.2.1. Problem Statement

The EURO meets NeurIPS Vehicle Routing Competition consists of two Vehicle Routing Problems, a static and a dynamic one. The static VRP is a VRPTW as described in subsection 5.1.2 based on real-world data. We note that there is no constraint on the number of vehicles that can be used. This ensures that there is always a feasible solution.

The dynamic VRP used in the competition is the same as described in subsection 5.1.3 and is built from a static VRPTW instance as follows. The epoch times $t_1, \ldots, t_K$ are defined in seconds by $t_i = 3600(i-1)$. We start the day at $t_1 = 0$. Each epoch takes an hour, and the day ends after K hours. At the start of each epoch, 100 requests are sampled uniformly from the static VRP, where the location, service time, quantity requested, and time windows are sampled independently from the static VRP requests. It will happen that, as the day progresses, newly sampled requests are infeasible due to their time windows closing before they can be visited. Only the feasible requests of the 100 sampled requests are added to the VRP of the current epoch. When no sampled request is feasible, the day ends. In the problem statement of the competition, there is a dispatch margin of an hour, meaning that if a request is sampled at the start of epoch i, a vehicle that visits it can only be dispatched at the start of epoch $i+1$. This is to model the time it takes between receiving the request and dispatching a vehicle, which includes picking the requested quantity and loading the truck. Depending on which static VRP the dynamic VRP is sampled from, K ranges from 5 to 9.

5.2.2. Data

For the EURO meets NeurIPS Competition, a dataset consisting of 250 static VRP instances has been made available for training models. They are real-world anonymised instances from a US based grocery delivery platform. Each VRP contains between 200 and 880 requests. From these instances, dynamic VRPs can be sampled as described above. There are also 100 static instances that were used to evaluate and rank the final models of the contestants. These were unknown to the contestants, but have been published after the end of the competition. We will evaluate our models on the same 100 instances, so that we can see how they rank among the contestant models.

The competition rules state that, per epoch, every model gets 2 minutes of calculation time on a single core CPU to find the most efficient routes. Our models will get the same calculation time on a single CPU core to calculate the final ranking. One caveat is that we do not know how the CPU used in the competition compares to the CPU used for training and evaluating our models, which can cause some discrepancies in how our results compare to those of the competitors.

Note that in the sampled DVRP, if 100 requests are sampled every epoch, and there are at maximum only 880 static requests, many requests will be sampled multiple times for the same

day. This might not sound very realistic, but it ensures that the sampled DVRP instances are difficult enough to solve given the computation constraints.

To obtain an expert dataset of input-output pairs $(\hat{s}, \hat{x})$ to train on, we can simply use the sampled DVRP instances as described above. Since these have all the information available, including the release times, we can solve them as hindsight DVRPs. The PyVRP package is able to solve these problems. Note that a hindsight DVRP with K epochs gives K different data points to be added to the dataset.

5.3. Modelling

The first thing to do when modelling the DVRP as an Inverse Optimization problem is to define the input $s \in \mathbb{S}$ and the output $x \in \mathbb{X}$ at each time step. During the day, at each epoch, we are tasked to find routes given the current active requests. The natural choice then is to have as input all the available knowledge at each time step. These include all the active requests and also which epoch it is. In the way the DVRP is modelled in the EURO meets NeurIPS Competition, we do also know how many epochs there are in the day, which we can use as input. This might sound a bit unrealistic, but in real life, a delivery company is likely to use a set number of epochs per day, in which case it is also known information that can be used as input data. The output at each epoch will be the routes to dispatch, i.e., a vector $x \in \{0, 1\}^{|E|}$ telling which edges are part of routes and which are not.

The challenge now becomes to design the Forward Optimization Problem

$$\arg\min_{x \in \mathbb{X}(s)} \langle \theta, \phi(s, x) \rangle + f(s, x)$$

in such a way so that the minimizer comprises a selection of requests to deliver to, and routes for the vehicles to drive. Luckily, the choice of $\phi(s, x)$ and $f(s, x)$ allows enough freedom to let us model the FOP as a Prize Collecting VRP. A solution of the PCVRP immediately gives a selection of requests and their optimal routes.

To do this, let us first define the feature vector per request. Given a set of N requests, let a request be denoted by $i \in [N] \setminus \{0\}$. Then let $\psi(i)$ be the feature vector of request i. This vector contains information such as the distance to the depot t_{0i}, request quantity q_i, time window $[e_i, l_i]$, etc. We will later discuss the specifications of ψ in more detail. If we set the weight vector θ as the same dimension as $\psi(i)$, we can then define the inner product $\langle \theta, \psi(i) \rangle$ as the prize associated with request i. Filling this in in the cost function of the PCVRP of Equation 5.2, we get the following optimization problem:

$$\begin{aligned}
&\arg\max_{x \in \mathbb{X}} \sum_{(i,j) \in E} x_{ij} \langle \theta, \psi(j) \rangle - \sum_{(i,j) \in E} x_{ij} t_{ij} \\
&= \arg\max_{x \in \mathbb{X}} \left\langle \theta, \sum_{(i,j) \in E} x_{ij} \psi(j) \right\rangle - \sum_{(i,j) \in E} x_{ij} t_{ij} \\
&= \arg\min_{x \in \mathbb{X}} \left\langle \theta, - \sum_{(i,j) \in E} x_{ij} \psi(j) \right\rangle + \sum_{(i,j) \in E} x_{ij} t_{ij}.
\end{aligned}$$

Now we have the form of the FOP that we want, with

$$\phi(s, x) = - \sum_{(i,j) \in E} x_{ij} \psi(j) \qquad \text{and} \qquad f(s, x) = \sum_{(i,j) \in E} x_{ij} t_{ij}$$

Intuitively, when given the input data s of the current epoch, our model will calculate the prize $\langle \theta, \psi(i) \rangle$ associated with each request. It then solves the resulting PCVRP, which can also be done with the PyVRP package. By solving the Inverse Optimization Problem of Equation 2.3, we will hopefully find a θ that assigns prizes in such a way that the solution of the resulting PCVRP finds efficient routes for the DVRP of the whole day.

We want to highlight why the method of using an ML model to translate the dynamic problem to a PCVRP is a good method. As discussed in Bogyrbayeva et al. (2022), heuristics generally still outcompete ML methods in classic VRP variants. The strength of ML algorithms in the context of VRPs lies not in the routing itself, but more in its ability to infer statistical information from the dataset to make good decisions in the uncertain present. Classic heuristic algorithms are not able to do this. Therefore, hybrid methods combining ML with classic heuristics show much promise. This is also what our method does. We let the IO algorithm deal with the uncertainty by creating prizes, and subsquently let a heuristic solve the resulting PCVRP. A more pure ML algorithm to find routes given the input data at each time step, would be to get a binary vector x with routes immediately from the input s, without translating to a PCVRP as intermediate step. The difficulty is then that the ML algorithm must also somehow learn the constraints of the DVRP exactly to prevent it from giving infeasible solutions as output. A more hybrid approach is to create a VRPTW as an intermediate step instead of a PCVRP. A model takes the input s, and gives a subset of requests that will be dispatched in the current epoch. This then defines a static VRPTW, for which a heuristic algorithm can find routes. The problem with this method is that it gives the ML algorithm a binary choice, which still means it would need to intuit a lot of information about solving VRPs. Using the PCVRP as intermediate step requires the IO algorithm to assign a score (the prize) that tells how confident the ML algorithm is that a request should be dispatched in a route. However, the heuristic PCVRP algorithm ultimately decides which requests are dispatched. The binary choice is no longer with the ML algorithm. This method should therefore find a good balance between the predictive power of ML with the strong routing decisions of heuristic algorithms.

It must be noted that our method of modelling the DVRP is very similar to that of the winners of the EURO Meets NeurIPS 2022 Vehicle Routing Competition (Baty et al., 2023). Their approach also translates the epoch input s into a PCVRP and lets a heuristic solve it. The main difference is in the translation method. The winners use a Neural Network to learn the translation, whereas we use IO. Though the NN evidently works well, we think IO is also well-suited for this translation step, since we know that the input-output data is found by an optimization problem (Iraj & Terekhov, 2021).

Furthermore, we would like to point out the similarity of the approach of both us and the winners of the Vehicle Routing Competition to (Parmentier, 2022). In this paper, a method is proposed by which difficult variants of optimization problems in operations research are translated to simpler variants using ML. These simpler variants have well-performing heuristics. A solution to such a simpler variant can then be translated back to a solution of the original difficult problem. In our case of the DVRP, we translate the difficult decision problem of which customers to dispatch in the current epoch to a simpler PCVRP. The solution to this problem need not be translated back, as it is the same solution to the original problem.

5.3.1. Feature Vector Per Request

Selecting which features to include in the feature vector per request ψ is one of the modelling parameters. It is difficult to reason in advance what works best, so we have chosen to train models with multiple feature vectors to test which works best.

The most simple feature vector contains information of the request, such as information regarding its location, delivery quantity, and time window. More specifically, for request i, $\psi(i)$ contains:

- The driving time from the depot to the request, t_{0i}.
- the quantity requested q_i.
- The servicing time s_i.
- The current epoch k.
- The start time of the time window relative to the current epoch k, $e_i - t_k$.
- The end time of the time window relative to the current epoch k, $l_i - t_k$.
- The release time r_i.

To expand this feature vector, we can add quantile information about the other currently active requests. This can embed some general properties of the current VRPTW. We add quantile information on two properties. The first one is the travel duration to the other requests. This gives information on how close the current request is to the other requests. The second property is time window slack, which shows how much time there is to deliver from the earliest delivery time of the current request to the latest delivery time of the other requests. Intuitively, this should show how easy it is to add the current request in a route without breaking time window constraints.

Furthermore, we can add feature information of the closest few neighbors of the current request. If the features of a given request give a low prize, but the 2 closest neighbors give a high prize, it might be worth it to make a small detour to deliver to this request as well. Strictly speaking, this is already taken into account in the PCVRP itself, but we can always try to see if it helps.

Since VRPs are a highly nonlinear problem, we will also add cross terms between all the features, so that we also take into account quadratic information in the input. This significantly increases the size of the feature vector ψ and therefore of θ, but this is no big problem for learning, since we have an explicit subgradient, which is easy to calculate, even for higher-dimensional feature vectors. With higher dimensionality, we do need more data to learn from, which can become a problem if the feature vector becomes too big. Only experiments with different feature vectors will show us what the best balance is between the size of the feature vector as compared to the size of the given dataset.

5.3.2. State Space Partitioning

To extend the size of the model, we can also apply State Space Partitioning. The most natural way to do this is to partition the state space into the different epochs, meaning we create a different model for each epoch. We can imagine that at different epochs, different strategies might be better. In the earlier epochs, we would want to delay more requests since they generally have more epochs ahead of them before they must be dispatched. Therefore, there is a greater chance of finding more efficient routes with new requests in the future. However, in the later epochs, we might want to add more requests to routes since there are fewer options in the future before they must be dispatched. It is therefore less likely to find more efficient routes in future epochs than we already have in the current one. We will also train models to see if state space partitioning is helpful.

5.3.3. DAgger

Since the choice of routes in one epoch impacts the active requests in future epochs, the problem is a dynamic one. Furthermore, we have acces to an expert, namely the hindsight DVRPTW. Hence, we can use the DAgger algorithm. Compared to the case study of Tetris however, it is less clear whether the expert only operates in a small part of the state space. The way a Dynamic VRP is sampled uniformly from a static VRP seems to suggest that large parts of the state space are already explored in the training data. We should therefore be cautious in expecting the DAgger algorithm to outperform static IO, but the only way to know for sure is to try.

Data Aggregation applied to the DVRP looks as follows:

1. With our current model, we simulate a number of DVRP instances for the full day. Here we acquire new input data $\hat{s}$
2. We let the hindsight algorithm find the best action for each of the states we get. Here we acquire the corresponding $\hat{x}$
3. we add these input-output pairs to the dataset and retrain our model using all the available data.

For the fast-DAgger algorithm, we only keep the data from the current iteration as training data.

5.4. Results

5.4.1. Experimental Setup

To create the training and testing datasets, the 250 static VRPTWs given as training data in the EURO Meets NeurIPS 2022 Vehicle Routing Competition were sampled as prescribed to create 250 DVRPs. These were solved as a hindsight DVRP by the state-of-the-art heuristic algorithm of PyVRP, which was given an hour to find optimal solutions. These solutions gave 1890 epoch datapoints, which were split into a training dataset consisting of 1359 datapoints and a test set consisting of 531 datapoints.

All the static and dynamic Inverse Optimization models were trained on the DelftBlue Supercomputer (DHPC, 2025), using 48 cores of either 2 Intel Xeon Gold 2648R processors or 2 Intel Xeon Gold 6488Y processors. All models were trained for approximately 10 hours.

5.4.2. Baseline

The EURO Meets NeurIPS Competition included a few baseline strategies for the selection of requests to dispatch every hour. These baseline strategies are:

- Lazy: Every request that can be delayed, will be delayed.
- Greedy: Every available request is dispatched.
- Random: Every request that can be delayed has an even chance of being delayed and being dispatched.

We can also use the hindsight expert solution as a baseline result. Furthermore, we add a final baseline strategy that we call the Lazy-Greedy (L-G) strategy. L-G follows the lazy strategy in the first epoch of the day, and the greedy strategy for the rest of the epochs. The route solutions were found using the PyVRP package and were given 2 minutes to find routes per epoch, as prescribed by the competition. The average optimal cost of Equation 5.1 of the baseline strategies is shown in Figure 5.4. The instances used to calculate these averages are

the 100 instances used in the finale of the EURO Meets NeurIPS Competition.

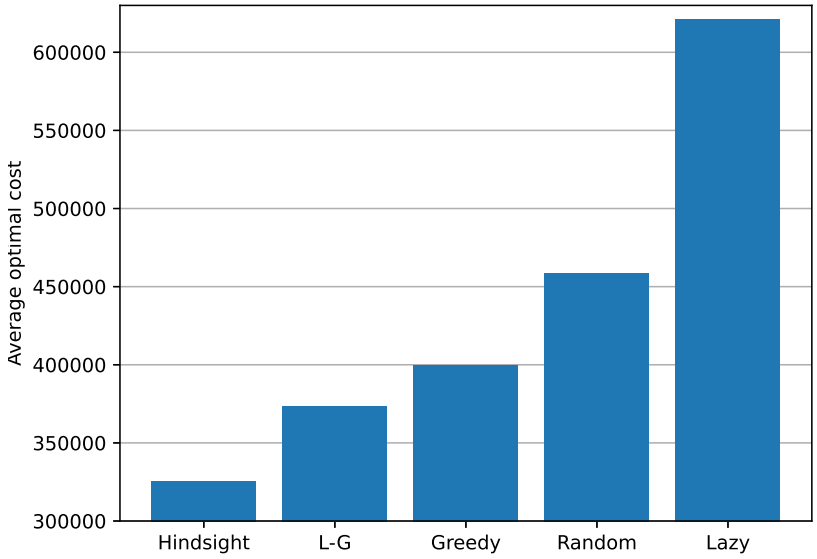


Figure 5.4: Average performance of the baseline strategies.

5.4.3. Static Inverse Optimization

For the IO model trained on the static dataset, the best performing feature vector per request ψ did not use state space partitioning, but did include the quantile information. No information about neighboring requests was used. The results per iteration of the IO algorithm are shown in Figure 5.5.

Looking at Figure 5.5, we would first like to discuss iteration 0. The training and testing loss are small for the first iteration, but the final cost is very large. This iteration is the model with initial value θ_0, which was set at 0. We then have $\langle \theta_0, \phi(s,x) \rangle = 0$ in the suboptimality loss of Equation 2.7. Still, $f(s,x) \neq 0$ in the suboptimality loss, but it turns out that it is small in comparison to the former inner product. At this initial iteration with $\theta_0 = 0$, the prizes of each customer in the PCVRP will be set to 0 at each epoch, which implies it is not worth visiting any customers. This version of the PCVRP then essentially becomes the lazy baseline strategy, which is verified by the final average cost of iteration 0 matching that of the lazy baseline in Figure 5.4.

For the other iterations of Figure 5.5, we can see that the training loss and test loss follow the same downward-trending curve. This would suggest that the model is learning well. However, when we test the models with the final dataset, the average optimal cost shows a different picture. Though the general trend is the same as the training and test loss, there are many outliers where the performance is significantly worse. This suggests some instability in the iterations that is not accounted for in the calculation of the loss. However, we do see that as the iterations increase, there are less outliers and they become smaller as well, suggesting a convergence to more stable models. Increasing the number of iterations is likely to reduce these outliers completely. Perhaps experimenting with the convergence rate of the step size η_t of the subgradient descent can also help alleviate the spikes.

If we were to compete in the EURO Meets NeurIPS 2022 Vehicle Routing Competition, based

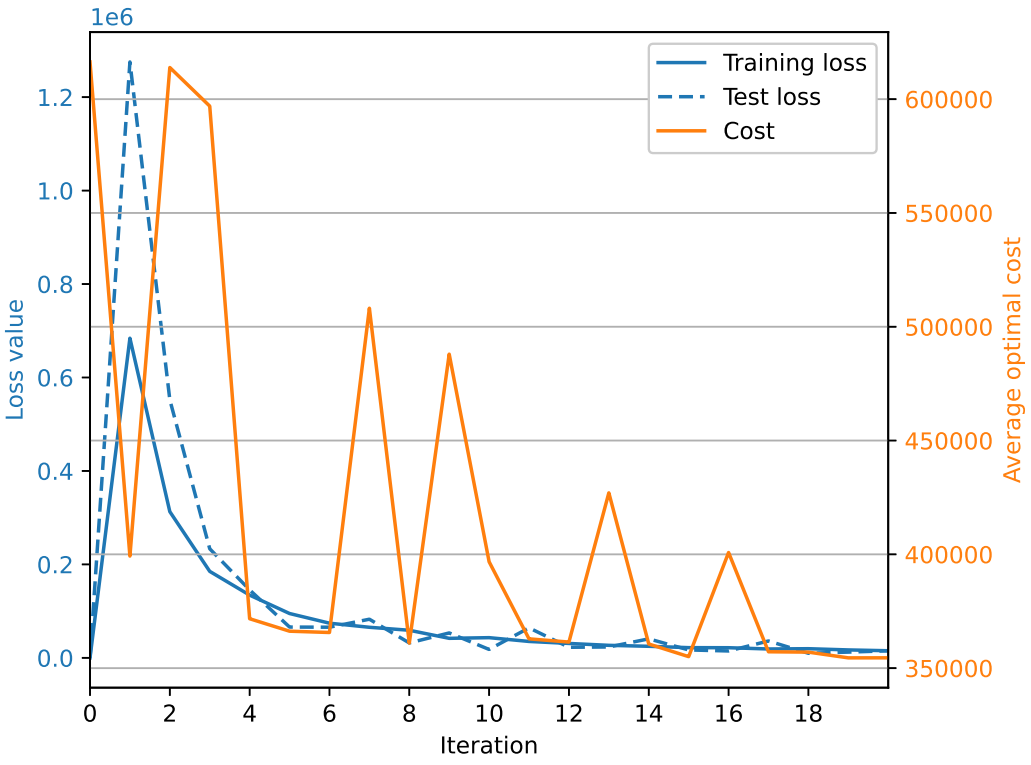


Figure 5.5: At each iteration of the model, the training and testing loss are plotted in blue, and the performance of the final dataset is plotted in orange. This performance is expressed as the average optimal cost over the 100 final DVRP instances.

on the training and test loss, the most logical model to compete with is that of the last iteration. This model would also perform best of all the iterations, with a final average optimal cost of 354512.73, ranking it at the second place among the original competitors.

5.4.4. Dynamic Inverse Optimization

DAgger

The feature vector per request ψ of the best DAgger model used state space partitioning, no quantile information, and no information about the neighboring requests. There were 5 DAgger iterations with 4 IO iterations per DAgger iteration. This gives 20 model iterations total. The training loss, test loss, and average optimal cost of the dataset used in the finals of the competition are shown in Figure 5.6.

The first thing we notice in Figure 5.6 is a periodic peak, especially in the training loss at iterations 1, 5, 9, 13, and 17. These iterations mark the start of a new DAgger iteration in which a new model is trained with the expanded dataset. When the model is retrained, the step size η_t resets to the chosen η_0. The sudden increase in step size causes a performance drop, as the stochastic subgradient descent now overshoots its target. Resetting the step size to η_0 makes sense if we retrain the model from scratch at each iteration, meaning we also reset θ_0 to 0 every time we retrain the model. However, since we take the θ_t of the last iteration as a more accurate first guess when retraining, we can also continue from the last step size. For the first three DAgger iterations, we can see that this peak decreases as more good datapoints are added to the total dataset, but after that, the peaks stay more or less constant, indicating that this resetting of the step size is bottlenecking the learning capabilities of the DAgger

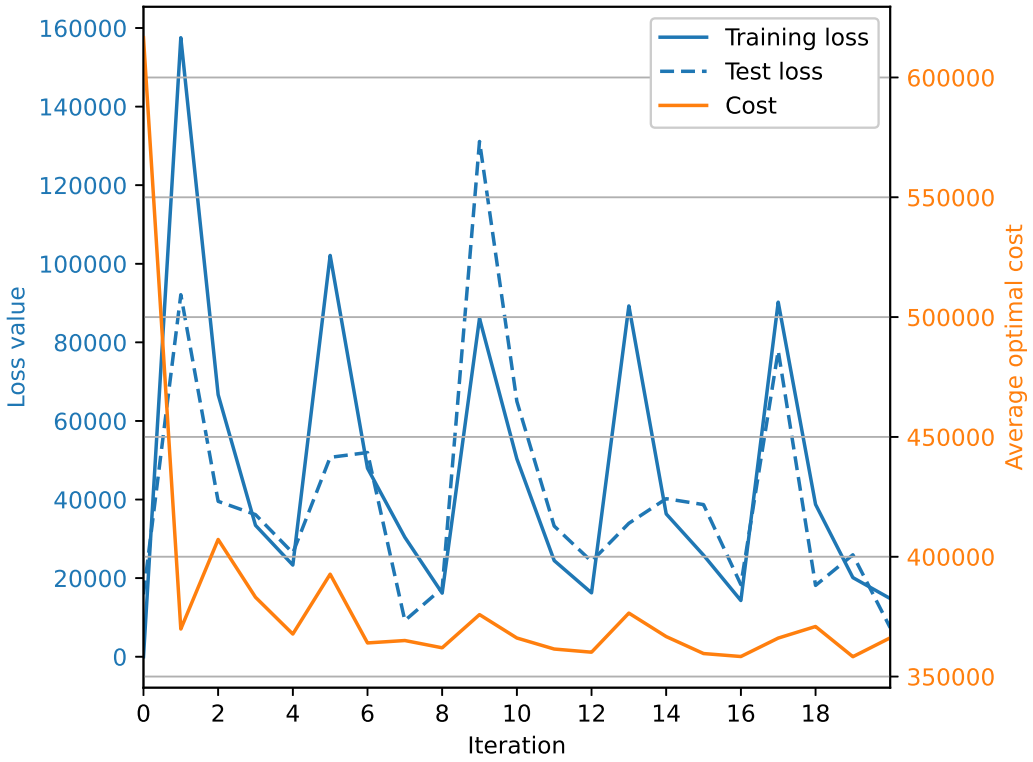


Figure 5.6: At each iteration of the model, the training and testing loss are plotted in blue, and the performance of the final dataset is plotted in orange. This performance is expressed as the average optimal cost over the 100 final DVRP instances.

algorithm. It is therefore recommended not to reset the step size when retraining the model at a new DAgger iteration, but instead to scale it with the total number of IO iterations that have come before.

If we were to pick a model out of the iterations to submit to the final competition, it would be the last iteration, as both the training and test loss are at its lowest in this iteration, indicating good performance and generalization. However, when we compare this to the average optimal cost of the final dataset, we have not chosen the best performing model. If we had participated in the EURO Meets NeurIPS 2022 Vehicle Routing Competition, the model of the second to last iteration would have given the best performance. Though unfortunate, this is most likely a random fluctuation, since the training and test loss of the other iterations generally correlate well with the performance on the final dataset. The model of iteration 19 would have an average optimal cost of 358278.97, ranking it as third place among the final competitors. This performance is worse than that of the best-performing static IO model. This could be due to the aforementioned bottleneck, but it could also be that the DAgger method is not well suited for this particular problem, as discussed in subsection 5.3.3.

Fast-DAgger

For the fast-DAgger algorithm, the best-performing feature vector per request ψ used state space partitioning, no quantile information, and the request information of the nearest neighbor. The training, testing, and final optimal costs are shown in Figure 5.7. Note that, since each iteration uses 48 CPU cores in parallel, the dataset used to train each iteration consists of 48 DVRP instances, which each contain a maximum of 8 epoch datapoints. This means that a

maximum of 384 datapoints was used for training per fast-DAgger iteration.

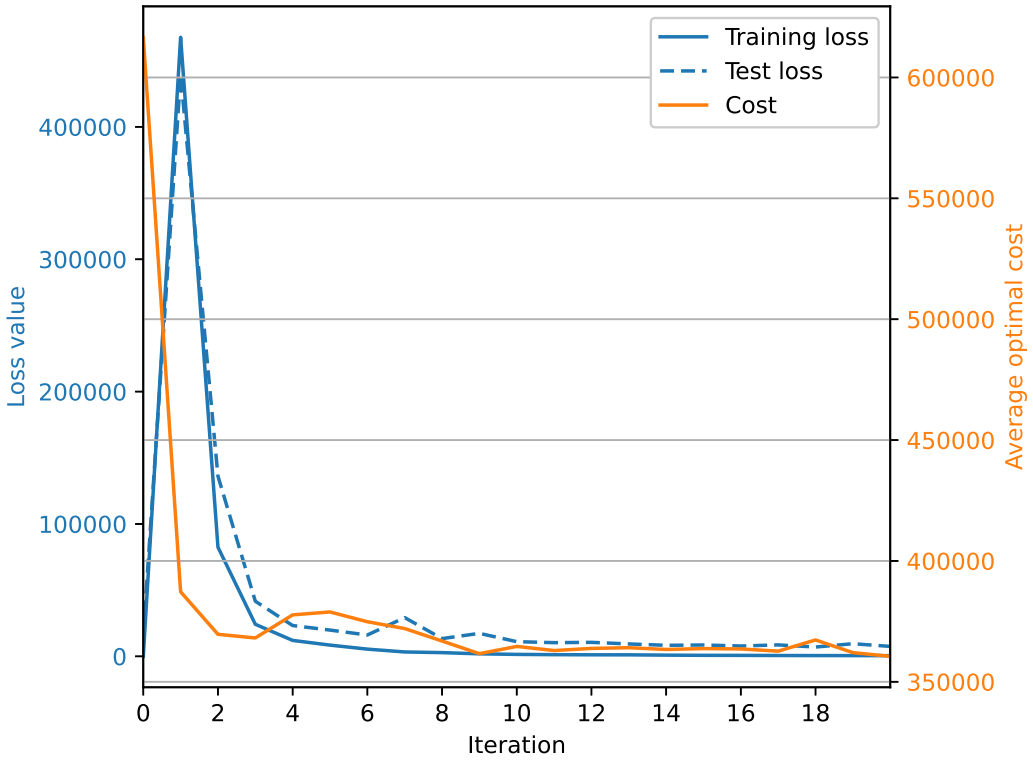


Figure 5.7: At each iteration of the model, the training and testing loss are plotted in blue, and the performance of the final dataset is plotted in orange. This performance is expressed as the average optimal cost over the 100 final DVRP instances.

Of all models discussed, the trainin and testing loss curves and the final performance curve of the fast-DAgger model are most well behaved and straight forward to interpret. Both the training loss and test loss drop quickly and become more or less constant at around iteration 10. After that, the improvement is minimal. This suggests a stable optimum model is found that should generalize well. This is verified by the performance on the final competition dataset, whose average optimal cost hardly improves after iteration 11.

According to the training and test loss, the last iteration is the best iteration to submit to the competition. This is also the best performing iteration for the final dataset with an average optimal cost of 360543.88, ranking it a fourth place among the original competition entries.

Comparison

In Figure 5.8, we show the performance of the models trained with the static dataset, the DAgger algorithm, and the fast-DAgger algorithm, when tested on the final dataset.

Most noticeable in Figure 5.8 is how big the peaks are of the static IO model as compared to that of the DAgger model. For both models, the peaks decrease in size over the iterations, but for the DAgger model there is a limit. Resetting the step size η_t every DAgger iteration prevents the model from converging. The step size η_t does keep decreasing with every iteration of the static IO model, which explains why the peaks keep decreasing in size as well. From iteration 17 and on, the static IO model outperforms every iteration of the DAgger and fast-DAgger model, showing that the static training data is sufficiently diverse already. Even though the DAgger model at iteration 19 outperforms all the fast-DAgger model iterations, this could be

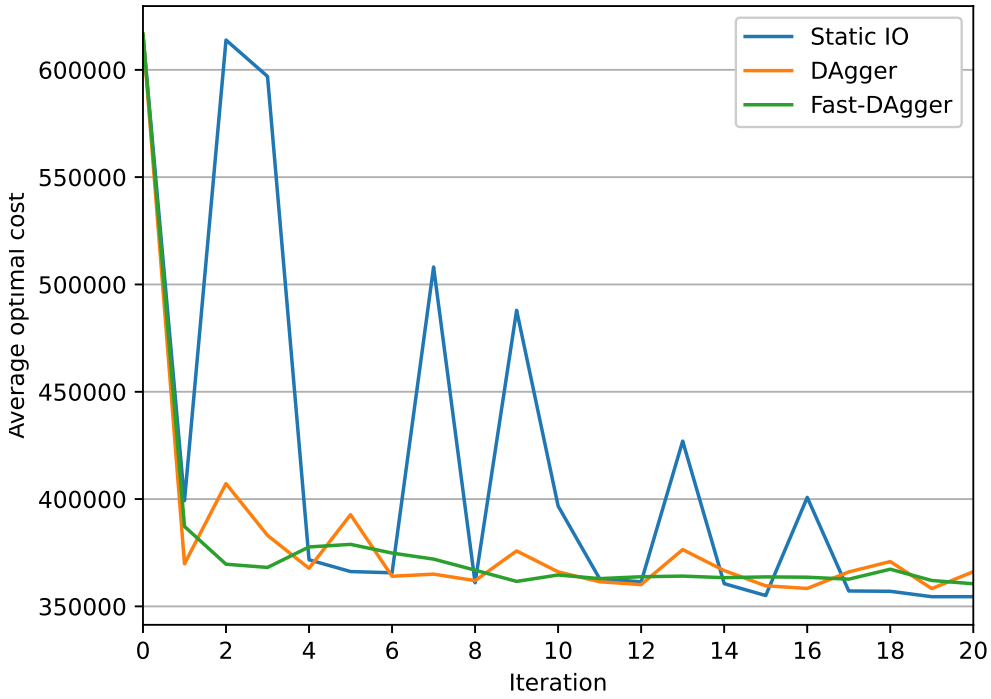


Figure 5.8: For the models trained with the static dataset, the DAgger algorithm, and the fast-DAgger algorithm, the

due to the more wild nature of the DAgger training. The fast-DAgger algorithm seems to be the safer choice for training models, as it shows no peaks and converges quickly.

We will now focus on the results for the DAgger and fast-DAgger algorithms. As discussed in section 3.3, we do not necessarily expect the fast-DAgger to outperform the normal DAgger algorithm, since we forfeit a more robust training scheme for a potential speedup in training time. Just as with Tetris, to compare the DAgger and fast-DAgger algorithm, we shall focus on the computation time, not on the iterations. Since the algorithms query the expert at different times during the optimization, the iteration times will differ as well. Thus, using the time it takes to calculate each iteration gives a more accurate picture of convergence times. In Figure 5.9, the average optimal cost of the competition finale dataset for both the DAgger and the Fast-DAgger algorithm is shown over time.

From Figure 5.9 it is clear that the Fast-DAgger algorithm converges faster, though this is mostly because the DAgger algorithm has trouble converging at all. The first model approaching an average optimal cost of 360000 happens at around 10000 seconds for the DAgger algorithm, and at around 170000 seconds for the fast-DAgger algorithm, suggesting that the fast-DAgger algorithm is not necessarily faster than the normal DAgger algorithm in finding good models. Ultimately, not too much can be concluded from Figure 5.9. Only one model was trained for both DAgger and fast-DAgger, using a stochastic gradient descent for both. This leaves too much up to random variation to draw any strong conclusions.

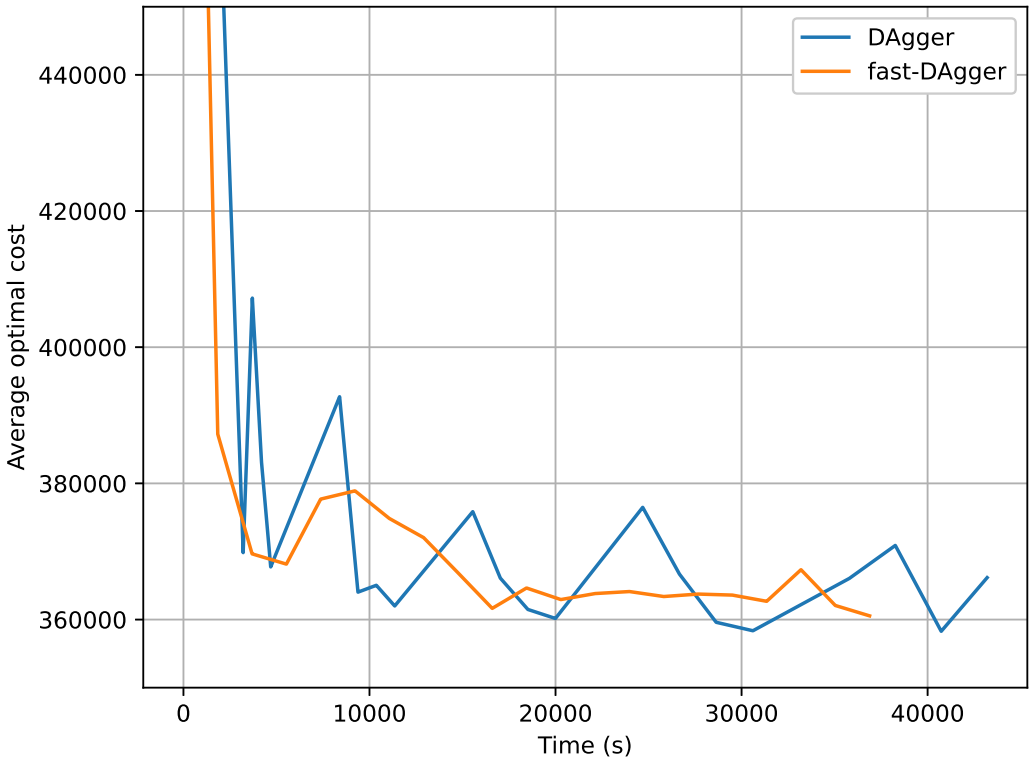


Figure 5.9: Average optimal cost value of the final dataset in the competition of both the DAgger and the Fast-DAgger algorithm. The iterations are measured in computation time.

5.4.5. Final Rankings

In Figure 5.10, the final scores of the best-performing static IO, DAgger, and fast-DAgger are plotted among the original competitors of the EURO Meets NeurIPS 2022 Vehicle Routing Competition, along with the scores of the hindsight and lazy-greedy baselines.

The static IO model performs best out of our models, again indicating that an active learning method like DAgger and fast-DAgger might not be the best approach for this particular problem. The static IO method also most closely resembles that of the winner, where the main difference lies in the method of assigning a prize to the PCVRP each epoch. We use IO, whereas the winners used a neural network. The better performance of the neural network could be because they are better suited to model the nonlinearities inherent in VRP problems. Alternatively, it could be due to tuning of the ML algorithms. Neural networks are better studied than IO methods, so more is known about how to tune these algorithms for good results.

Note that we picked the best models based on the final average cost, not based on the training and testing loss. This is a bit unfair to the other competitors, who did not have the luxury of picking their best models like this. However, we do this to show the best possible results we could get using our techniques. This only makes a difference for the DAgger model, where we would have picked a different model based solely on the training and test loss.

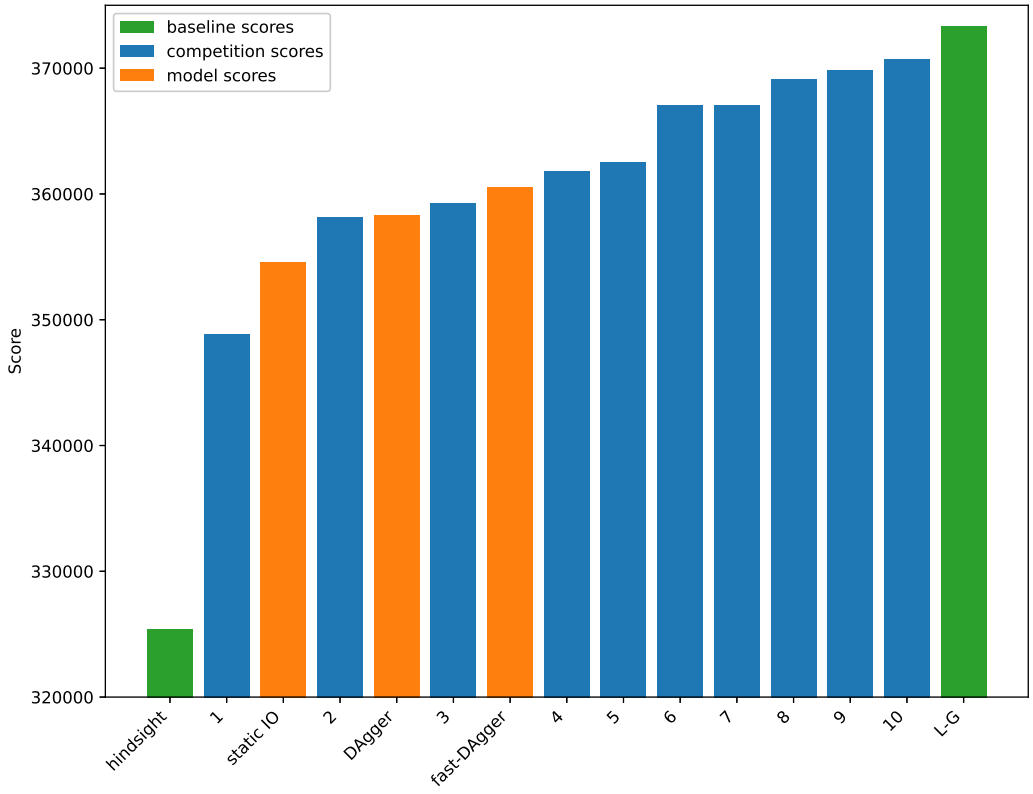


Figure 5.10: Final ranking of our models among those of the original competitors. The hindsight baseline shows how much the algorithms can potentially improve, whereas the lazy-greedy baseline shows how well a simple strategy compares to the competition.

6 Discussion

6.1. Inverse Optimization

We want to start the discussion by highlighting the performance of Inverse Optimization as a machine learning algorithm. The choice of function space leaves a lot of room to model a variety of different problems, considering the large difference between the case studies. Both in Tetris and the DVRP, we created models that perform well. In the EURO Meets NeurIPS competition, we even managed to rank second with an IO model! This is quite impressive, considering that with our choice of function space parameterization and feature vector, we essentially only create a quadratic model.

The choice of an affine function space also has drawbacks. For instance, if we want to model more complex non-linearities, we must do so through the choice of the feature vector, which is not an easy task. We have also observed that the model performance is very sensitive to our choice of features. It is difficult to predict which features work well and which will not. Luckily, the combination of an affine function space and the use of the suboptimality loss allows a relatively short training time, meaning we can test many feature vectors in a short time. Unfortunately, as we have seen with the DVRP, the performance of feature vectors can even be dependent on how the model is trained. For instance, the DAgger model had a different optimal feature vector than the static IO model. Since the training of DAgger and fast-DAgger models can take a long time, it would be useful to quickly find good feature vector candidates using static IO from a dataset, and then improving this using either DAgger or fast-DAgger. But alas, the results suggest that we cannot do this.

6.2. DAgger

It is important to note that we must be careful to draw strong conclusions about the performance and applicability of DAgger for the method of IO. We have only studied two specific case studies, and the results we base these conclusions on are single models. Considering the stochastic nature of both case studies and the training process, stronger conclusions can only be drawn when models are trained multiple times so that some statistical analyses can be performed.

Having said that, generally, DAgger and fast-DAgger combine well with Inverse Optimization, demonstrating the promise of training IO models with active learning methods. The models created using DAgger and fast-DAgger generally perform well, although they do not outperform the static IO models in the case studies. In the DVRP, they even performed a bit worse. It

is therefore fruitful to discuss whether the case studies of Tetris and the DVRP were suitable for use with DAgger and fast-DAgger. Obviously, we had a dynamic system with access to the expert with both case studies, so we could use the DAgger and fast-DAgger algorithm. But it does not follow that we therefore should use these algorithsm. From the case studies, we can distill some extra lessons about when to use DAgger when one is training IO models.

We reasoned before that DAgger can be useful when the expert dataset only represents a small subset of the full state space. Even though this is the case with Tetris, DAgger still did not perform better than static IO. The expert model trained with static IO managed to perfectly emulate the expert. From this, we can conclude that, even though the training data only represents a small part of the complete state space, it generalizes very well to the rest of the state space. The lesson to learn is that if the training data generalizes well, DAgger is not necessary.

We also reasoned in chapter 3 that, when the training data is a good representation of the complete state space, DAgger is most likely not necessary. With the DVRP case study, due to the uniform sampling of static VRPs to obtain DVRP instances, the training data already represents a large part of the state space. We could confirm via experiments that in this case, DAgger is indeed not necessary. This is further confirmed in Greif et al. (2024), where, an optimization problem closely related to the DVRP is solved using the method of translating it first to the PCVRP using ML and then solving that with heuristics, as proposed by the winners of the EURO Meets NeurIPS Competition in Baty et al. (2023). This closely related problem is called the Dynamic Inventory Routing Problem (DIRP). Like us, Greif et al. combined the scheme with DAgger to train a model, and like the EURO Meets NeurIPS 2022 Vehicle Routing Competition, dynamic instances of the DIRP were sampled from static IRP instances. When this sampling was done uniformly, the trained DAgger model did not perform extremely well. However, when the sampling was done in a more realistic way, the DAgger model improved a lot upon the state-of-the-art heuristics. This improvement in performance can be explained by the fact that the change in sampling makes the training dataset less representative of the whole state space, and thus the addition of DAgger can improve performance.

Another problem we ran into when using DAgger is that during the training, the expert must be queried online. For the DVRP case study, this expert is the hindsight DVRP, which is a difficult problem to solve. To keep the training time reasonable, we cannot give the expert a long time to find a good solution. When making a static dataset using the same expert, we can let the expert solve for much longer, as the dataset only needs to be made once. This static dataset can therefore have higher-quality input-output pairs than the dataset aggregated using DAgger.

Overall, we recommend careful deliberation to decide whether DAgger is a useful addition when training IO models. First, a static IO model should always be made. Only in the case where it does not generalize well due to a limited training set, should DAgger be considered. Other considerations include how easily the expert can be accessed or how much time a model can take to train, as DAgger is a resource intensive training method.

fast-DAgger From both case studies, we must conclude that the proposed fast-DAgger algorithm does not necessarily converge faster than the standard DAgger algorithm. Since convergence for the Tetris models was extremely fast, it is difficult to draw conclusions about the convergence speed of the fast-DAgger algorithm. For the DVRP, the fast-DAgger algorithm also showed no significant convergence speedup over the standard DAgger algorithm. This is because the speed up of fast-DAgger is not achieved during the dataset collection, but during training, especially if there are many DAgger iterations. The shorter training time means the

fast-DAgger algorithm collects more data in the same amount of time as compared to the normal DAgger algorithm. If the data collection (i.e., querying the expert) takes longer than training, the fast-DAgger algorithm even runs the risk of training and converging more slowly than the standard DAgger algorithm. The speed up of fast-DAgger will become most apparent when there are many DAgger iterations. However, precisely because the DAgger algorithm slows down a lot with more iterations, the number of DAgger iterations will most likely be kept low. This gives little space for the fast-DAgger algorithm to shine, especially since we lose any convergence properties the standard DAgger does have, as shown in Ross et al. (2011).

6.3. Further research

We want to conclude with some recommendations for further research.

Firstly, we would like to test whether our conclusions explaining why DAgger did not improve model performance are true or not. These conclusions are mostly based on reasoning. We do not have data to back it up. We would like to train models for different dynamic problems where the training data is a small subset of the total state space and does not generalize well. This was evidently not true for both of our case studies. We could for instance, use the DVRP from the EURO Meets NeurIPS 2022 Vehicle Routing Competition, but sample differently from the static VRP instances. Maybe then the DAgger algorithm shows an improvement, like it does for the DIRP as shown in Greif et al. (2024).

Secondly, the pipeline for solving the competition DVRP, as laid out by Baty et al. (2023), shows promise of working for different Dynamic Vehicle Routing Problems. It has already been shown to work for a related dynamic routing problem in Greif et al. (2024). This pipeline could, for instance, also be applied for variants of the Taxi Dispatch Problem (Salanova et al., 2011). Inverse Optimization and perhaps DAgger seem like suitable methods to investigate this.

Thirdly, we are interested in the use of IO and perhaps DAgger in the context of policy learning for Markov Decision Processes (MDPs). These models are dynamic, and the action at every iteration is found by a difficult optimization problem. This seems like a good fit for training IO models. Many dynamic problems can be cast as MDPs, including both the Tetris and DVRP case studies of this thesis. Much research already exists for policy learning of MDPs (Hussein et al., 2017), but to the best of the authors' knowledge, IO has not been tried yet for this goal.

Fourthly, the observation that the weights θ_t for the realistic Tetris model converge to a value that is very sensitive to small changes is curious, especially when there are so many different weights that perform very similarly. We do not know why it happens, and we would like to find a way to prevent it. It would be better to converge to weights that are more robust to small changes. Perhaps using the Augmented Suboptimality Loss of Scroccaro, Atasoy, and Esfahani (2023) will help, as this loss is specifically designed to converge to more robust weights.

Finally, even though the fast-DAgger algorithm did not outperform DAgger for the two case studies investigated in this thesis, it did not necessarily perform much worse than the DAgger algorithm either. Both case studies also seem to hint that the fast-DAgger algorithm converges more smoothly than the DAgger and static algorithms, though there definitely is not enough data to support this. We believe there are cases where fast-DAgger would be preferable over the standard DAgger approach. More research is needed to investigate if and when this is the case. Perhaps a theoretical approach can also prove some convergence properties.

7

Conclusion

In this thesis, we have investigated, using the case studies of Tetris and the DVRP of the EURO Meets NeurIPS 2022 Vehicle Routing Competition, whether the active learning method of Dataset Aggregation (DAgger) can increase the performance of models trained by the Machine Learning algorithm of Inverse Optimization (IO).

It turns out that, even though the IO models performed well in both case studies, the addition of DAgger does not automatically improve the performance of the IO models. We have concluded that DAgger can be useful only when the training dataset is not representative of the full state space of possible inputs, and when the datapoints do not generalize well. This was not the case in both of the case studies, which explains why the IO models trained with DAgger did not outperform the IO models trained with a static dataset.

We also proposed a novel variation of the DAgger algorithm, fast-DAgger, that should improve the training time. However, results show that, at least for the two case studies, fast-DAgger did not give faster model convergence. We believe fast-DAgger will only converge faster when there are many DAgger iterations, or when calculating the new iterations is a lot slower than querying the expert. More research must be done to confirm this.

references

Aswani, A., Shen, Z.-J. (, & Siddiq, A. (2018). Inverse Optimization with Noisy Data. *Operations Research*, *66*(3), 870–892. https://doi.org/10.1287/opre.2017.1705

Baty, L., Jungel, K., Klein, P. S., Parmentier, A., & Schiffer, M. (2023, April). Combinatorial Optimization enriched Machine Learning to solve the Dynamic Vehicle Routing Problem with Time Windows. https://doi.org/10.48550/arXiv.2304.00789

Bertsimas, D., Gupta, V., & Paschalidis, I. C. (2014). Data-Driven Estimation in Equilibrium Using Inverse Optimization. (arXiv:1308.3397). https://doi.org/10.48550/arXiv.1308.3397

Bogyrbayeva, A., Meraliyev, M., Mustakhov, T., & Dauletbayev, B. (2022, May). Learning to Solve Vehicle Routing Problems: A Survey. https://doi.org/10.48550/arXiv.2205.02453

Burgiel, H. (1997). How to lose at Tetris. *The Mathematical Gazette*, *81*(491), 194–200. https://doi.org/10.2307/3619195

Chan, T. C., Mahmood, R., & Zhu, I. Y. (2023). Inverse optimization: Theory and applications. *Operations Research*.

Danskin, J. M. (1967). *The Theory of Max-Min and its Application to Weapons Allocation Problems* (M. Beckmann, R. Henn, A. Jaeger, W. Krelle, H. P. Künzi, K. Wenke, & Ph. Wolfe, **typeredactors**Vol. 5). Springer. https://doi.org/10.1007/978-3-642-46092-0

DHPC, D. H. P. C. C. (2025). DelftBlue Supercomputer (Phase 2).

Greif, T., Bouvier, L., Flath, C. M., Parmentier, A., Rohmer, S. U. K., & Vidal, T. (2024, February). Combinatorial Optimization and Machine Learning for Dynamic Inventory Routing. https://doi.org/10.48550/arXiv.2402.04463

Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, *50*(2), 1–35.

Iraj, E. H., & Terekhov, D. (2021, February 22). *Comparing Inverse Optimization and Machine Learning Methods for Imputing a Convex Objective Function*. arXiv: 2102.10742 [math]. https://doi.org/10.48550/arXiv.2102.10742

Kool, W., Bliek, L., Numeroso, D., Zhang, Y., Catshoek, T., Tierney, K., Vidal, T., & Gromicho, J. (2022). The euro meets neurips 2022 vehicle routing competition. *Proceedings of the NeurIPS 2022 Competitions Track*, 35–49.

Lee, Y. (2013, April 14). *Tetris AI – The (Near) Perfect Bot*. Code My Road. Retrieved August 27, 2025, from https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

Mor, A., & Speranza, M. G. (2022). Vehicle routing problems over time: A survey. *Annals of Operations Research*, *314*(1), 255–275.

Parmentier, A. (2022). Learning to Approximate Industrial Problems by Operations Research Classic Problems. *Operations Research*, *70*(1), 606–623. https://doi.org/10.1287/opre.2020.2094

Ross, S., Gordon, G. J., & Bagnell, J. A. (2011, March). A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. https://doi.org/10.48550/arXiv.1011.0686

Salanova, J. M., Estrada, M., Aifadopoulou, G., & Mitsakis, E. (2011). A review of the modeling of taxi services. *Procedia - Social and Behavioral Sciences*, *20*, 150–161. https://doi.org/10.1016/j.sbspro.2011.08.020

Scroccaro, P. Z., Atasoy, B., & Esfahani, P. M. (2023, May). Learning in Inverse Optimization: Incenter Cost, Augmented Suboptimality Loss, and Algorithms. https://doi.org/10.48550/arXiv.2305.07730

Scroccaro, P. Z., van Beek, P., Esfahani, P. M., & Atasoy, B. (2023, July). Inverse Optimization for Routing Problems. https://doi.org/10.48550/arXiv.2307.07357

Wouda, N. A., Lan, L., & Kool, W. (2024). PyVRP: A High-Performance VRP Solver Package. *INFORMS Journal on Computing*. https://doi.org/10.1287/ijoc.2023.0055

A

Training And Test Loss Tetris Models

In Appendix A, we will show the training and testing loss as well as the accuracy error for every model discussed in section 4.3. Note that the test dataset used to calculate both the test loss and the accuracy error was sampled in the same way as the training set, implying that it includes only datapoints that the expert explores, i.e., with minimal total grid height, bumpiness, and number of holes.

It might be unexpected that the loss starts at 0 for al the models. This is because our initial choice for θ is the zero vector. Remember from section 4.2 that we found that $f(s, x) = 0$ in the function space of Equation 2.4 when modelling Tetris as an IO problem. This means the suboptimality loss of Equation 2.7 becomes $\langle \theta, \phi(s, x) \rangle = \langle 0, \phi(s, x) \rangle = 0$.

A.1. Expert Models

A.1.1. Short Term

Static IO

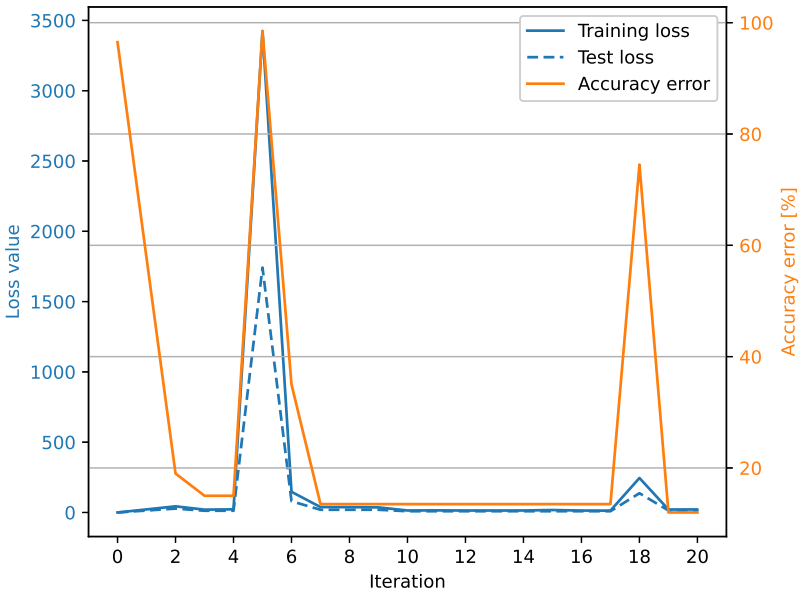


Figure A.1: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

DAgger

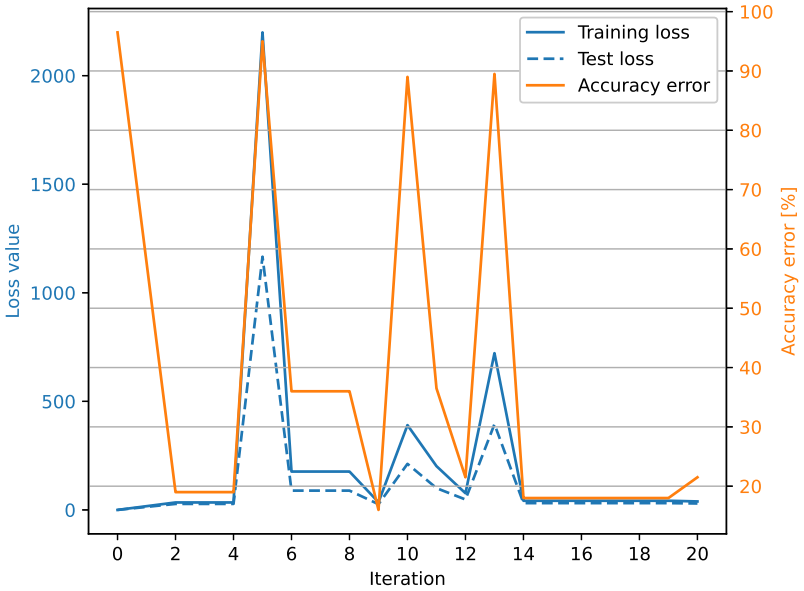


Figure A.2: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

Fast-DAgger

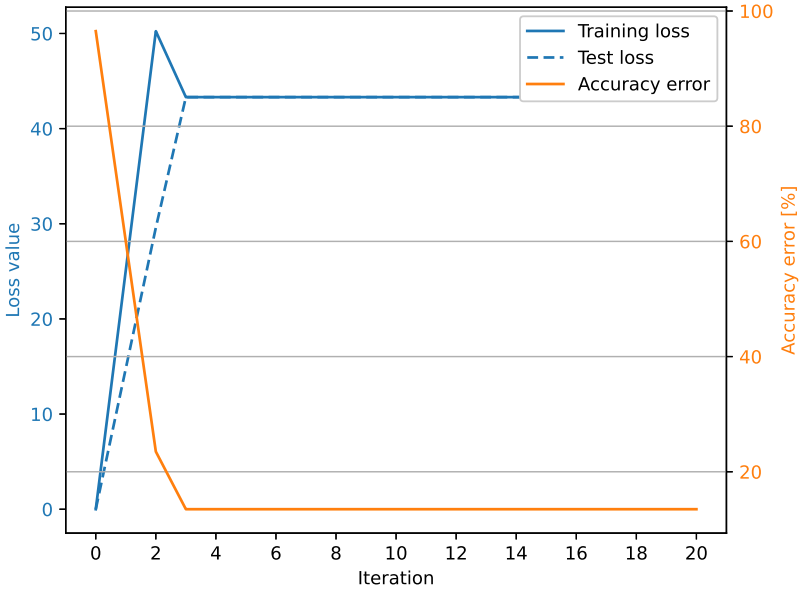


Figure A.3: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

Notice that, even though the graph shape of the training and test loss of Figure A.3 looks very different than those of the static IO and DAgger models, nothing weird is going on. Since there are no spikes in performance, the maximum loss value is a lot lower, leading to a different scale for the loss when its plotted. This means lower losses are plotted much higher on the graph than for the other models.

A.1.2. Long Term

Static IO

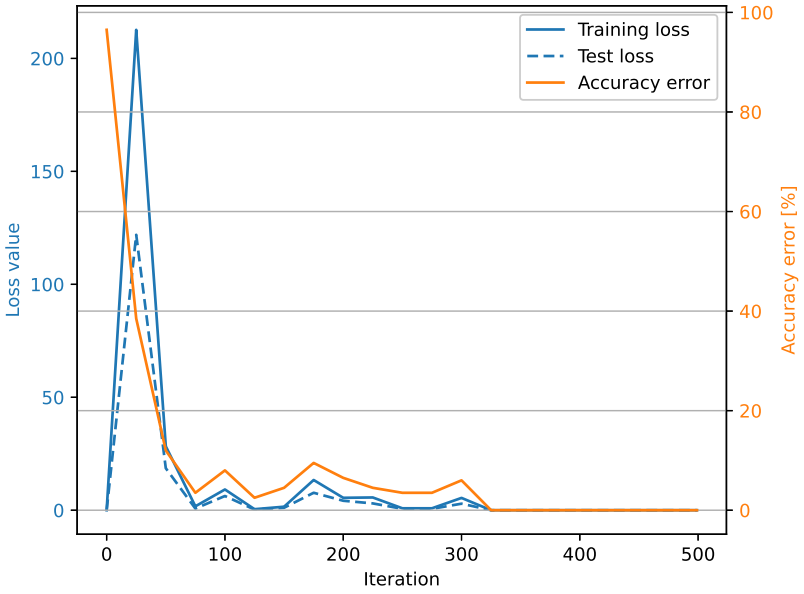


Figure A.4: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

DAgger

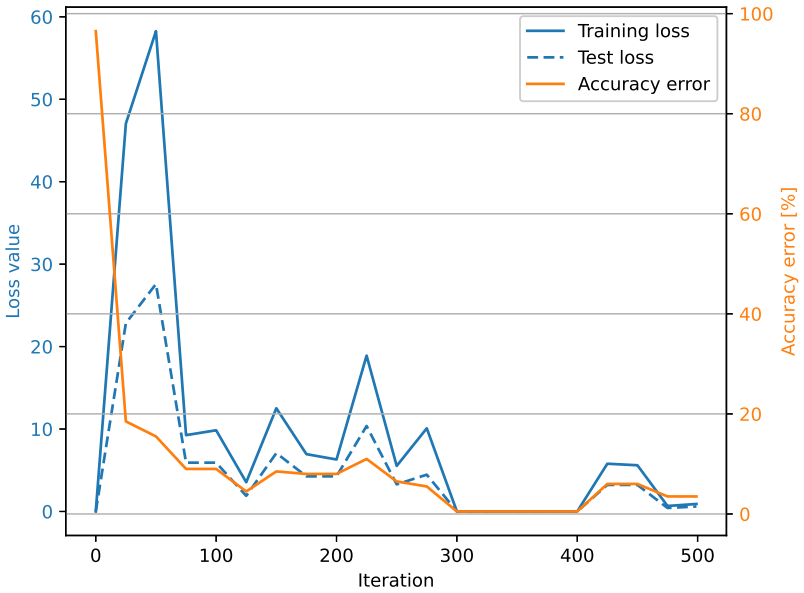


Figure A.5: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

Fast-DAgger

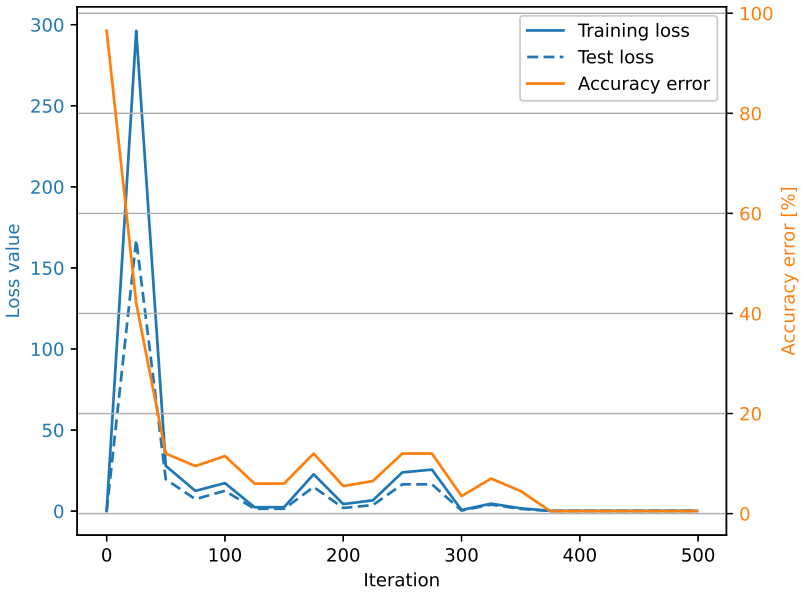


Figure A.6: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

A.2. Realistic Models

A.2.1. Short Term

Static IO

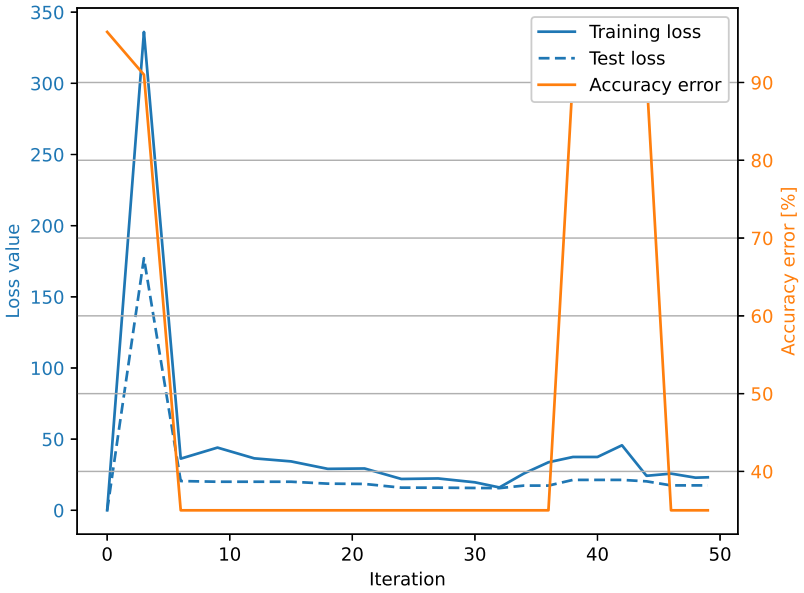


Figure A.7: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

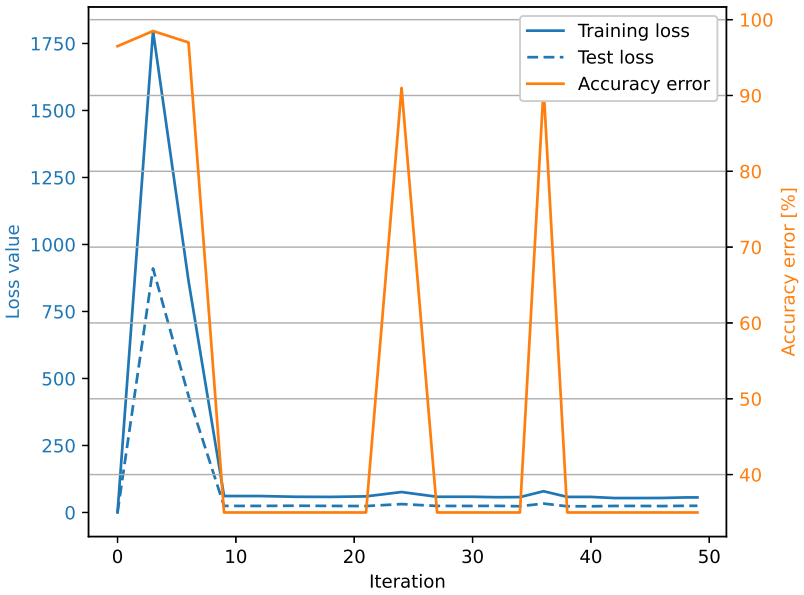


Figure A.8: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

DAgger
Fast-DAgger

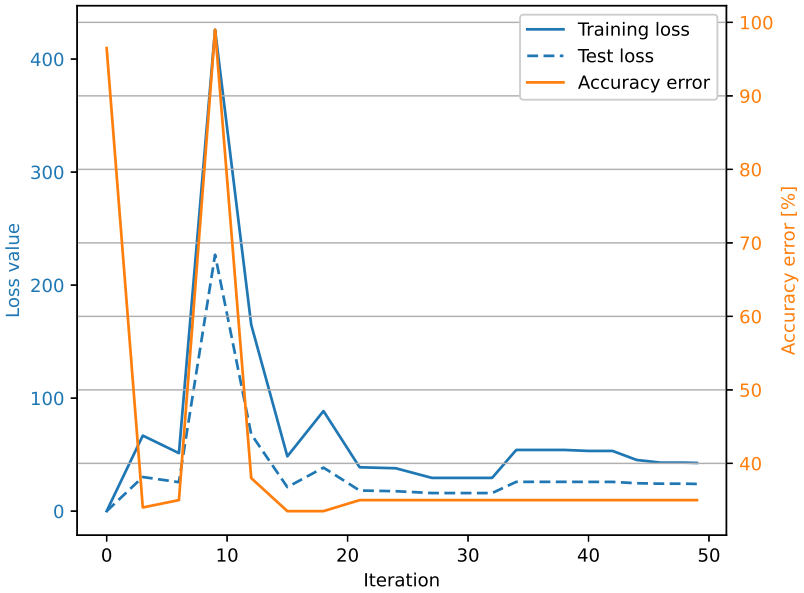


Figure A.9: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

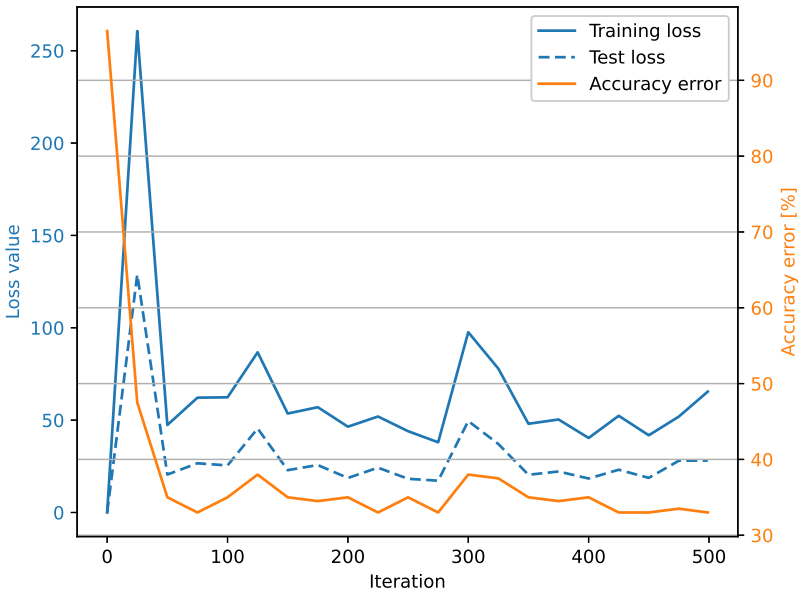


Figure A.11: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

A.2.2. Long Term
Static IO

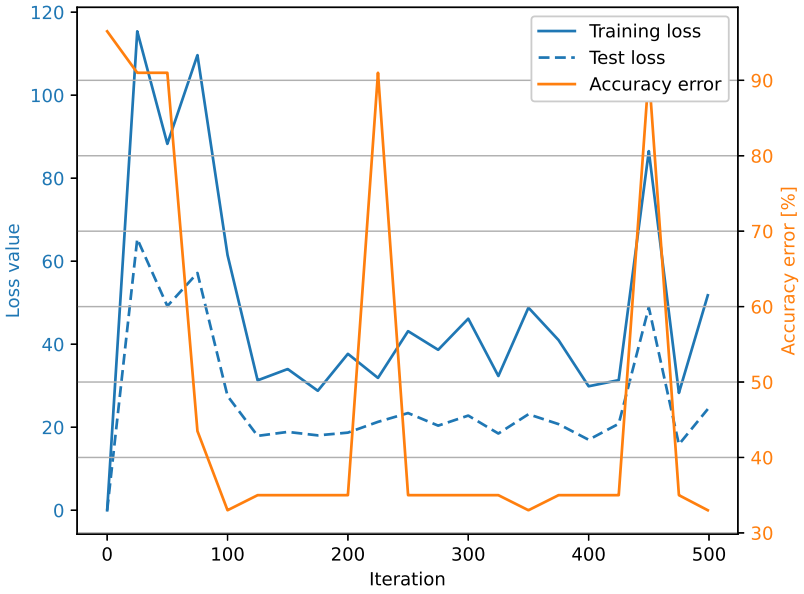


Figure A.10: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.

DAgger
Fast-DAgger

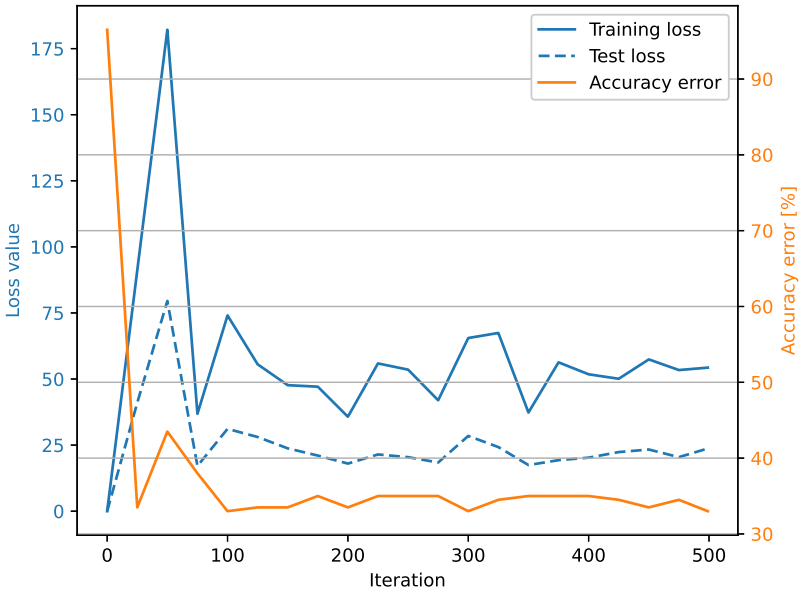


Figure A.12: The training and testing loss over the iterations are plotted in blue. The prediction error percentage of the iterations is plotted in orange.