Machine Learning (2024) 113:5151–5181
https://doi.org/10.1007/s10994-023-06413-x

GVFs in the real world: making predictions online for water treatment

Muhammad Kamran Janjua[1] · Haseeb Shah[1] · Martha White[1,2] · Erfan Miahi[1] · Marlos C. Machado[1,2] · Adam White[1,2]

Received: 3 April 2023 / Revised: 2 August 2023 / Accepted: 3 October 2023 /
Published online: 8 November 2023

Abstract
In this paper we investigate the use of reinforcement-learning based prediction approaches for a real drinking-water treatment plant. Developing such a prediction system is a critical step on the path to optimizing and automating water treatment. Before that, there are many questions to answer about the predictability of the data, suitable neural network architectures, how to overcome partial observability and more. We first describe this dataset, and highlight challenges with seasonality, nonstationarity, partial observability, and heterogeneity across sensors and operation modes of the plant. We then describe General Value Function (GVF) predictions—discounted cumulative sums of observations–and highlight why they might be preferable to classical n-step predictions common in time series prediction. We discuss how to use offline data to appropriately pre-train our temporal difference learning (TD) agents that learn these GVF predictions, including how to select hyperparameters for online fine-tuning in deployment. We find that the TD-prediction agent obtains an overall lower normalized mean-squared error than the n-step prediction agent. Finally, we show the importance of learning in deployment, by comparing a TD agent trained purely offline with no online updating to a TD agent that learns online. This final result is one of the first to motivate the importance of adapting predictions in real-time, for non-stationary high-volume systems in the real world.

Keywords Reinforcement learning · Water treatment · Time series prediction

Editors: Yuxi Li, Alborz Geramifard, Lihong Li, Csaba Szepesvari, and Tao Wang.

✉ Muhammad Kamran Janjua
mjanjua@ualberta.ca

✉ Adam White
amw8@ualberta.ca

[1] Department of Computing Science, Alberta Machine Intelligence Institute (AMII), University of Alberta, Edmonton, AB, Canada

[2] Canada CIFAR AI Chair, University of Alberta, Toronto, ON, Canada

1 Introduction

Learning in deployment is critical for partially observable decision-making tasks (Sutton et al., 2007). If the evolution of state transitions is driven by both the agent's actions and state variables that the agent cannot observe, then the process will appear non-stationary to the agent. For example, an agent controlling chemical dosing in a water treatment plant may correctly learn the relationship between increasing chemicals to reduce turbidity in the water. However, inclement weather events can also impact water turbidity causing the agent's prediction of future turbidity—and thus choices of chemical dosing—to be suboptimal. One approach to mitigate this problem is to allow the agent to continually update its predictions and decision-making policies online in deployment.

Effective multi-step prediction forms the basis for decision-making in almost any reinforcement learning system. Classical value-based methods, such as Q-learning (Watkins and Dayan, 1992), construct a prediction of future discounted rewards in order to decide on what actions to take. Policy gradient methods such as PPO (Schulman et al., 2017) and SAC (Haarnoja et al., 2018) typically define the agent's policy through an estimate of the value function. Thus in applications, a reasonable first step is to build a prediction learning system that can predict reward and sensor values far into the future. This is not only an important step to assess the feasibility of adaptive control but is also a useful first step because the tasks of feature engineering, network architecture design, optimization, and tuning of various hyperparameters will be shared and beneficial to both a prediction learning system and a full reinforcement learning control system.

There has been growing interest in moving RL techniques out of video games and into the real world. In many applications, such as chip design (Mirhoseini et al., 2021), matrix multiplication (Fawzi et al., 2022), and even video compression (Mandhane et al., 2022), the problem setting of interest is simulation. Another approach is to design and train an agent in simulation and then deploy a fixed controller, sometimes even in the real world. This approach has been used for example in navigating stratospheric balloons (Bellemare et al., 2020), controlling plasma configurations inside a fusion reactor (Degrave et al., 2022), and robotic curling (Won et al., 2020).

In this paper, we study the application of machine learning techniques, specifically prediction methods from reinforcement learning, on a real drinking-water treatment plant. In our setting, we do not have access to a high-fidelity simulator of the plant, nor the resources to commission one. This work explores the feasibility of adaptive learning systems in the real world. Instead of relying on access to a simulator our approach extensively leverages offline data for iterating design choices or pre-training the agent.

Closer to our work, recent work on automating HVAC control used an approach where the agent is first tuned on off-line data and then a learning controller is deployed that is updated once a day (Luo et al., 2022). In this work the authors explicitly avoided offline training on operator data, citing the well-known issues of insufficient action coverage. Nevertheless, *batch* or *off-line RL* methods (Ernst et al., 2005; Riedmiller, 2005; Lange et al., 2012; Levine et al., 2020) have been successfully used in settings where a fixed policy or value function is extracted from a data-set, with several practical applications (Pietquin et al., 2011; Shortreed et al., 2011; Swaminathan et al., 2017; Levine et al., 2018).

Drinking-water treatment is basically a two-stage process, as depicted in Fig. 1. First, water is pumped into a large mixing tank where chemicals are added to cause dissolved solids to clump together. The next step is to pull the pretreated water through a filter membrane where only clean water molecules can pass through the filter membrane and

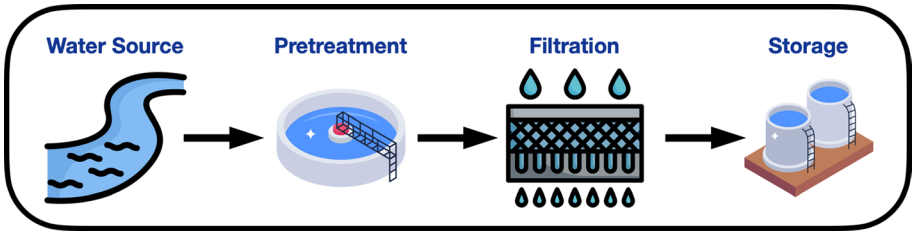

Fig. 1 An illustration of the drinking-water treatment plant. The entire plant is divided into two main stages: pretreatment and filtration. The pretreatment stage is concerned with adding chemicals to the raw water, followed by the filtration stage where the water is pumped through filters for further purification (Color figure online)

the solids and other continents remain. Periodically, the primary filter is cleaned by simply running the process backwards blasting the filter membrane clean: a process called backwashing. In Canada, the operation of a water-treatment plant can represent up to 30% of a town's municipal budget (Copeland and Carter, 2017).

Drinking-water treatment is uniquely challenging compared to other applications due to two key characteristics. The data produced by a water-treatment plant, like many real-world systems, is high-dimensional, noisy, partially observable, and often incomplete, making online, continual prediction extremely challenging. In water treatment, the plant can operate in different modes, such as production and backwashing. The mode has a profound impact on data produced by the system and even changes the range of valid sensor readings. Second, the different components of the plant operate at different timescales and decisions have delayed consequences. For example, the chemical dosing rate is typically not changed more often than once a day, backwashing happens multiple times a day, and the pretreatment-tank mixing-rate can be adjusted continuously. Each one of these choices can result in changes in sensor readings over minutes—chemical dosing changes the water pressure on the filter within 30 min—to months—too much chemical dosing can degrade filter efficiency over the long run.

In this paper, we investigate multi-variate, multi-step prediction in deployment on a real system. We provide a detailed case study on water treatment, first demonstrating the inherent nonstationarity of the problem and the benefits of learning continuously in deployment. We show that using a simple trace-based memory to overcome partial observability, we can learn accurate multi-step predictions, called general value functions (GVFs) (Sutton et al., 2011; Modayil et al., 2014), using temporal difference (TD) learning. Because GVFs can be learned with standard reinforcement learning algorithms like TD, they can easily be updated online, on every step. We show that updating online can significantly improve performance over only training from an offline log of data. The online prediction agent also benefits from this offline data, to pre-train the predictions and to set the hyperparameters for updating online in deployment. Our approach allows us to have a fully specified online prediction agent—with hyperparameters automatically selected using a simple modification on the standard validation procedure—that continues to adapt and improve in deployment.

Finally, we also contrast these GVF multi-step predictions to the more classical predictions considered in time series prediction: n-step predictions. The primary goal of this comparison is to provide intuition: n-step predictions are a more common and widely understood approach to multi-step prediction, as compared to GVFs. Our goal

Table 1 A brief summary of different measurements each of the sensor type is responsible for working out

Sensor type	Measures
Pressure	Pressure on the membrane
Flowmeter	Flow rate of the fluid
pH	Acidity and alkalinity of the solution
Temperature	Temperature of the water
Turbidity	Turbidity of the water
Total organic carbon (TOC)	Organic carbon in the water
Conductivity	Ability to pass an electric current

is to introduce GVF predictions to a wider audience, and hopefully motivate this additional modelling tool. Beyond this, we highlight that GVFs can have benefits over n-step predictions. The target for a GVF is typically smoother because it is an exponential weighting of future observations, rather than an observation at exactly n steps in the future. Consequently, we also expect this target to be lower variance and potentially simpler to learn. We find that GVF predictions have higher accuracy than the n-step predictions on our data, controlling for the same state encoding and network size, in terms of the normalized mean-squared error. Taken together, our work provides several practical insights on designing neural-network learning systems capable of learning in deployment.

2 The data of water treatment

Like any industrial control process, a water treatment plant has the potential to generate an immense amount of data. Our system is instrumented with a large number of sensors reporting both (1) water chemistry throughout the treatment pipeline, and (2) properties of the mechanical components of the plant. Taken together these sensor readings form a long and wide time series with several interesting properties that make long-term prediction difficult. In this section we highlight these properties with examples from a real plant, explaining how each makes long-term prediction challenging.

2.1 Wide, long, and fast data

Our system reports 480 distinct sensor values at a rate of one reading per second producing a large time series. One year of data consists of over 31 million observations of the plant. In contrast, the recent M5 time-series forecasting competition used a dataset with 42,840-dimensional observations and 1969 time-steps; over 84 million samples (Makridakis et al., 2022). Using multiple years of water treatment data puts us on the same scale as state-of-the-art forecasting grande challenge problems. We summarize some of the sensors in Table 1, and provide more detail in Table 2.

Our data exhibits a coherent structure over the year, month, day and minute. In Fig. 2 we plot incoming water temperature at three temporal resolutions. Mechanical systems like

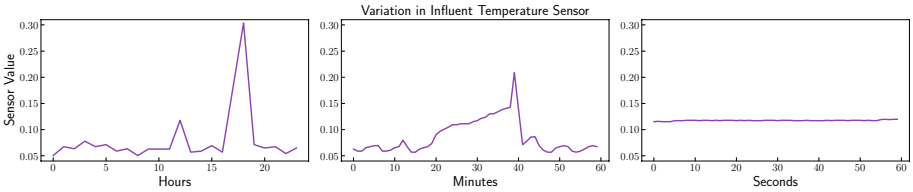

Fig. 2 The many timescales of water treatment. Each subplot shows the incoming water temperature from the river at different temporal resolutions. Viewing left to right, if we look at temperature over the entire day (sub-sampled) we see a single outlier and an otherwise fluctuating baseline. In the middle subplot, looking at a single hour of data, we see the spike has more structure. Finally, the rightmost subplot shows one minute of data sampled at the fastest possible timescale of the system (no sub-sampling), which shows how in a short timescale measurements can even appear constant (Color figure online)

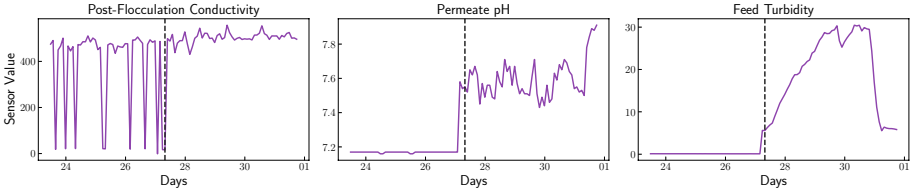

Fig. 3 Raw values of some of the sensors before and after the cleaning. The black dotted line indicates when the sensors were manually cleaned by the plant operators. Note that the data is sub-sampled to avoid congestion in the plot (Color figure online)

ours often support sampling at rates of 1 Hz or greater, whereas data sets commonly used in time-series forecasting are wide and short; typically sampled once a day.[1] In water treatment, high-temporal resolutions are relevant because the data can be noisy (as highlighted in Fig. 3) and averaging is lossy. In addition, if one were to change process set-points (the ultimate end-goal of prediction), this may require rapid adjustment (for example, adjusting PID control parameters during a backwashing operation).

2.2 Sudden, unpredictable events

Our data exhibit substantial distribution shifts, largely due to unpredictable events. For example, Fig. 3 shows the impact of cleaning different sensors. Most of these sensors get physically dirty over time due to a variety of factors. Sometimes water gets accumulated in the sensor enclosure, or moisture develops on the physical sensors, causing the readings to become noisy and unreliable. The plant operators manually clean the sensors to make sure they are as noise-free as possible and are reliably operating. Often times the sensor patterns indicate that they have recently undergone cleaning. This change in pattern manifests itself as the sensor signal stabilizes over time post-cleaning.

A water treatment plant operates in different modes which dramatically impacts the data generated. The main modes of operation are production and backwash. In

[1] Taking an extreme example, the well-known Sunspots dataset is unidimensional and contains 3240 data points.

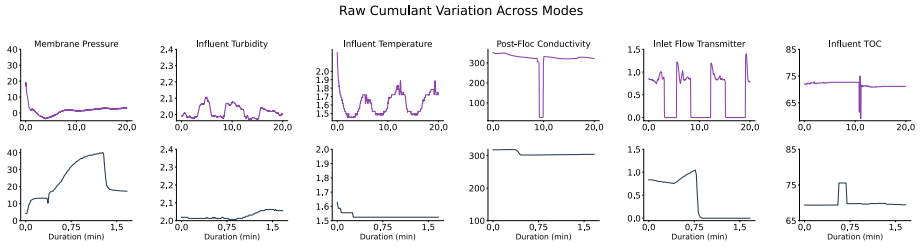

Fig. 4 Variation across modes of different sensors. For brevity, we only produce two important modes, namely production (PROD), and backwashing (BW). The top row corresponds to the production mode, while the bottom row corresponds to the backwashing mode. The x-axis of the backwash data (second row) is plotted over a much shorter time scale because backwash only lasts for a couple of minutes, whereas production durations are much longer (Color figure online)

production, the water is drawn through the filter to remove containments and it is moved to storage. In backwashing—the process of cleaning the filters—water moves backward through the system from storage, through the filters and eventually into the waste (reject) drain. In Fig. 4 we can see the impact of these two modes across several sensors.

In our plant, mode change is driven either by a fixed schedule or human intervention. Maintenance, for example, occurs every day at 4:30 am, triggering the Membrane Integration Test (MIT) mode, whereas backwashing occurs on a strict schedule. Sensor changes due to these mode changes should be predictable from the time series itself, however, more ad hoc operator interventions are better represented as unpredictable external events; for example, when the plant is shut down. In addition, unscheduled maintenance occurs periodically—it is conceivable that such maintenance could be predicted based on the state of the plant, but there are other constraints like staffing constraints that can drive mode change. Later we discuss how we encoded plant modes, which was key for successful prediction.

2.3 Sensor drift and seasonal change

Water treatment is predominately driven by the conditions of incoming river water which changes throughout the year. These changes are driven by seasonal weather patterns. In the dead of Winter, the river is frozen and cool, clean, low turbidity water flows under the ice into the intake valves. During the Spring thaw—called the freshet—snow and ice all along the watershed of the river melt, increasing volume, flow, turbidity, and organic compounds in the river. Early Summer is dominated by a mixture of melted snow and ice higher up in the mountains and heavy rains that cause second and third freshets. Over the Summer, precipitation reduces, causing the late Summer and Fall to exhibit similar patterns as the Winter. All of these patterns are clearly visible in Fig. 5.

Change also happens within a single day. In Fig. 6 we see how two different sensors evolve over a single day, on different days. As we can see in the plot of Feed Turbidity, some days are similar, but others, such as May 31, 2022, exhibit dramatically different dynamics. In some applications like HVAC control (Luo et al., 2022), it is sufficient to perform learning on a batch of data once a day. In water treatment, the sensor dynamics provide the opportunity to observe sensor changes throughout the day.

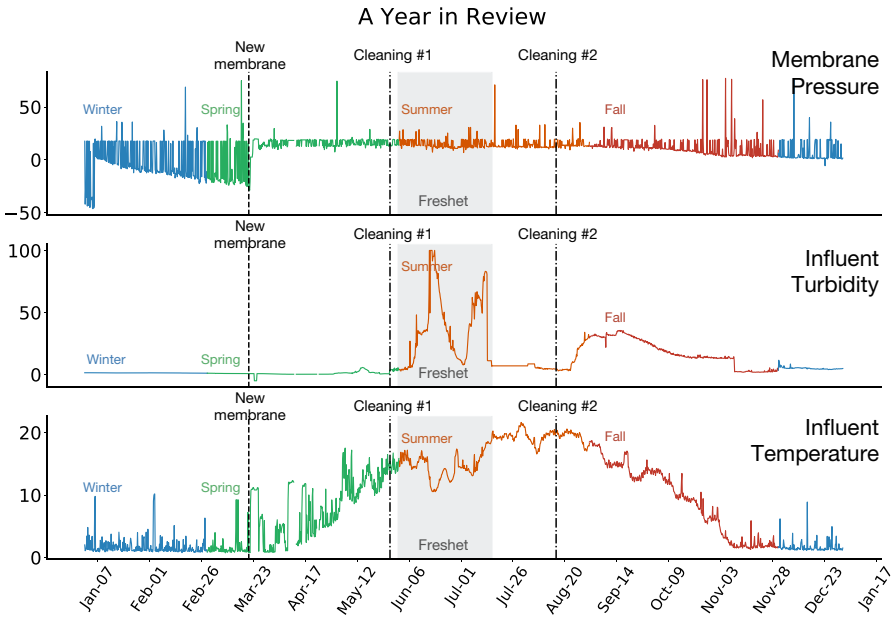

Fig. 5 A year's worth of data for three different sensors. These three sensors are representative of the impacts that seasonal variations, or changes in the physical state of plant's components, have on the underlying telemetric stream of data. Note that the data is sub-sampled to avoid congestion in the plot. There are gaps in the data because the plant was down for maintenance (Color figure online)

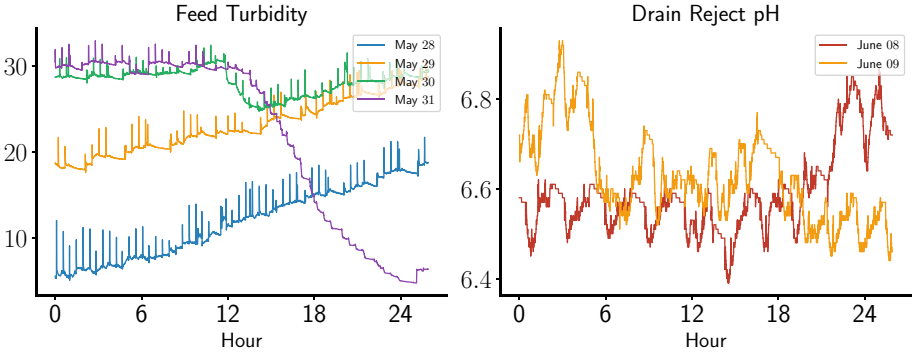

Fig. 6 Feed Turbidity and Drain Reject pH sensors, respectively. An example of data drift in sensor values over hours. Each line represents a sensor plotted over a different day. We see variation in sensor readings both within a day and over multiple consecutive days (Color figure online)

2.4 The state of a water-treatment plant?

What information would we need to predict water treatment data many steps into the future, with high accuracy? The plots above paint a clear picture of a partially observable complex dynamical system. Consider the Spring freshet. The volume and flow of the river will be driven by weather patterns and by the snow accumulation all along the watershed

throughout the Winter. Digging deeper, the turbidity and other metrics are also driven by erosion and composition of the riverbed, which changes all the time. The chemical makeup of the water could spike if there is a change in farming practices in the area—water runoff from fields along the river. Even everyday things like a fire in the town can add huge pressure demands on the plant—many plants have dedicated pumps just for fires.

In all the examples above, it would be impractical to sensorize these events so they could be detected in the plant. In fact, we would need to predict these events in advance of their occurrence (including the weather) in order to accurately predict our data in advance. Perhaps, we could simply make predictions based on the entire history of the time-series. The history would still only approximate the state, because we do not know the starting conditions: data from five and ten years ago. In addition, such an approach is not scalable if the end goal is to build a continual learning system that runs for years generating tens of millions of samples a year.

In the end, capturing the true underlying state is likely impossible and we must be content using learning methods that continue to learn in deployment in order to achieve accurate prediction. Such methods track the changing underlying state of the plant. The idea is to use computation and extra processing of the recent data to overcome the limitations of the agent's state representation (Sutton et al., 2007; Tao et al., 2023), similar to how an approximate model of the world can be used to deal with non-stationary tasks in reinforcement learning. In the next section, we will discuss different algorithms for learning and tracking in deployment and later show their advantages on water-treatment data.

3 Multi-step prediction

In this paper we are interested in scalar predictions of multi-dimensional time-series, many steps into the future. On each discrete time-step, $t = 1, 2, ...$, the learning algorithm observes a new observation vector, $o_t \in \mathbb{R}^d$, which form a sequence of vectors from the beginning of time.

$$o_{0:t} \doteq o_0, o_1, o_2, ..., o_t.$$

We do not assume knowledge of the underlying process that generates the series. That is, the next generation of observation vector may depend not just on $o_{0:t}$, but also on other quantities not observable to the learning system. For example, the future turbidity of the river water is impacted by future weather which is not observable and generally not predictable.

The goal is to estimate some scalar function of the future values of the time-series on time-step t, given $o_{0:t}$. In this paper, we focus on classical n-step predictions from time-series forecasting and exponentially weighted infinite horizon predictions commonly used in reinforcement learning, which we discuss in the following sections.

3.1 Classical time-series forecasting

The first prediction problem we consider is simply predicting a component of the time-series on the next time-step, $o_{t+1}^{[i]}$. We denote the i-th component of x_t as $x_t^{[i]}$. This scalar one-step prediction $\hat{v}_t$ at time t can be approximated as a function of a finite history of the time-series:

$$\hat{v}_t \doteq f_{\text{TS}}(o^{[i]}_{t-\tau:t}, w_t) \approx o^{[i]}_{t+1}, \tag{1}$$

where $w_t \in \mathbb{R}^k$ is the learned weights and τ is the number of previous observation vectors used to construct the prediction. For a classical autoregressive model, f_{TS} is a linear function of this history $o^{[i]}_{t-\tau:t}$. More generally, f_{TS} can be any nonlinear function, such as one learned by a neural network.

In order to predict more than one step into the future we can iterate a one-step prediction model. The naive approach is to simply feed the model's prediction of the next observation into itself as input to predict the next step, now 2 steps into the future, and so on. For example a three-step prediction:

$$\hat{v}_{t+2} \doteq f_{\text{TS}}([o^{[i]}_{t-\tau:t-1}, \hat{v}_{t+1}, \hat{v}_t], w_t) \approx o^{[i]}_{t+3}. \tag{2}$$

Notice how two components of the history of the time series have been replaced by estimates. As we iterate the model beyond τ steps into the future all the inputs to f_{TS} will become model estimates.

Another approach is to directly learn a k-step prediction and avoid iterating altogether. One-step models are convenient because they can be updated at every timestep. Unfortunately, if the one-step model is inaccurate the model produces worse and worse predictions as you iterate it further. A *direct method* estimates a k prediction as a function of the history of the series:

$$\hat{v}_t \doteq f_{\text{DE}}(o^{[i]}_{t-\tau:t}, w_t) \approx o^{[i]}_{t+1+k}. \tag{3}$$

In many applications, we are interested in multi-dimensional data and in predicting many steps in the future. We can go beyond auto-regressive approaches by simply considering these time series prediction problems as supervised learning problems. For example, we can learn a neural network f_{DE} that inputs the last k multi-dimensional observation vectors $o_{t-k:t}$ and predicts $o^{[i]}_{t+1+k}$, trained by constructing a dataset of pairs $(o_{t-k:t}, o^{[i]}_{t+1+k})$. We can also go beyond finite k-length histories, and use recurrent neural networks, which is becoming a more common practice in time series prediction (see Hewamalage et al. (2021, Sect. 2.3.1)). When we start using this supervised learning framing, we lose some of the classical strategies for dealing with correlation in the data, but in general, evidence is mounting that we can obtain improved performance (Crone et al., 2011; Hewamalage et al., 2021).

3.2 GVFs and temporal difference learning

In reinforcement learning, multi-step predictions are formalized as value functions. Here the objective is to estimate the discounted sum of all the future values of some observable signal, with discount $\gamma \in [0, 1)$:

$$G_t \doteq \sum_{j=0}^{\infty} \gamma^j o^{[i]}_{t+1+j}. \tag{4}$$

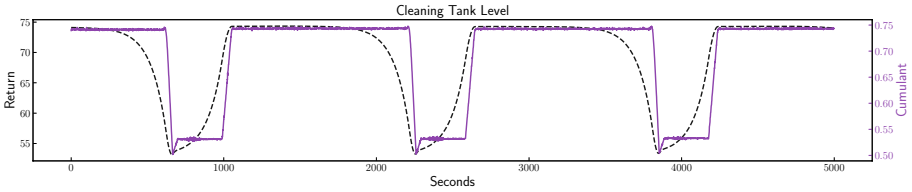

Fig. 7 A sample time-series of tank level from a real water-treatment plant and an idealized prediction (labelled return). The x-axis is the time-step, corresponding to one second. The prediction is ideal in the sense that we can simply compute the exponentially weighted sum in Eq. (4) given a dataset—the idealized prediction is not the output of some estimation procedure. Later we will show learned predictions and how they match the ideal. Notice how the idealized prediction increases well before the time-series reaches its maximum value, and falls well before the time-series does. In this way, the idealized prediction at any point in time provides an anticipatory measure of the rise or fall of the data in the future. This is discussed extensively in prior work (Modayil and Sutton, 2014), so we do not belabour the point here (Color figure online)

Technically G_t summarizes the infinite future of the time-series, but values of $o^{[i]}$ closer to time t contribute most to the sum.[2] These exponentially weighted summaries of the future automatically smooth the underlying data $o^{[i]}$—potentially making estimation easier—and provide a continuous notion of anticipation of the future as discussed in Fig. 7. For this reason, they have been called "Nexting" predictions (Modayil and Sutton, 2014), but more generally were introduced as general value functions (GVFs) (Sutton et al., 2011), where they generalize the notion of a value by allowing any cumulant to be predicted beyond a reward.

GVF predictions can be learned using temporal difference learning. As before, the prediction is approximated with a parameterized function, $f_{\text{TD}}(s_t, w) \approx G_t$, where s_t is a summary of the entire series, $o_{0:t}$, up to time t. For example, we could use an RNN to summarize this history; we opt for an even simpler approach—memory traces—which we describe in Sect. 4.1. The prediction on time-step t is updated using the temporal-difference error

$$w_t \leftarrow w_t + \alpha(c_t + \gamma f_{\text{TD}}(s_t, w) - f_{\text{TD}}(s_{t-1}, w))\nabla f_{\text{TD}}(s_{t-1}, w), \tag{5}$$

where $\alpha \in (0, 1]$ and $c_t \doteq o_t^{[i]}$.

4 Methods

In this paper, we investigate methods that can be pre-trained from offline logged data and perform fine-tuning in deployment. The algorithms we investigate can be used offline, online, or a combination of the two. Offline algorithms can randomly sub-sample and update from the offline data as much as needed (i.e. until the training loss converges). Online data, generated in the *deployment phase* can only be resampled from a replay buffer once it has been observed. Using the online data is restricted: the algorithms cannot look ahead into the future of the time-series, they must wait for each data point to become

[2] Note that we do not explicitly give the (partially observable) Markov decision process formalism because we do not need that precise notation to explain the concepts. Further, the predictions we consider are all on-policy predictions, so we do not need to know the explicit decision-making policy in order to do the update. For this reason, we avoid introducing all the notation around actions and policies, since they will not be used.

available step-by-step. After a sample is observed it can be resampled over and over via a replay buffer. In this section, we outline the algorithms, and how they can combine offline and online learning.

4.1 Constructing agent-state

These algorithms can be used in real-time, making and updating predictions live as the plant is operating. For simplicity in our experiments, we only simulate that setting here using a static offline dataset. The agent can iterate, in order, on the batch of offline data and it is equivalent to having made predictions live on the plant. The offline batch of data is $\mathcal{D}_{\text{offline}} = \{(o_t, c_{t+1}, o_{t+1})\}_{t=1}^{N}$, where N is the total number of transitions, $o_t \in \mathbb{R}^d$ is the observation vector, $c_{t+1} \in \mathbb{R}$ is the signal to predict or *cumulant*, o_{t+1} is the next observation vector.

The data, however, is partially observable and the agent should construct an approximate state. A typical approach used in machine learning is to use RNNs, to summarize history (Hochreiter and Schmidhuber, 1997; Cho et al., 2014; Hausknecht and Stone, 2015; Vinyals et al., 2019). However, we found for our sensor-rich problem setting, that a simpler trace-based memory approach was just as effective and much easier to train. The general idea is to use an exponentially weighted moving average of the observations; such an exponential memory trace has previously been shown to be effective (c.f. Mozer 1989; Tao et al. 2023; Rafiee et al. 2023). We include more explicit details on how we created our approximate state observation vector in Appendix A.

Once we have constructed this approximate state vector, which we denote $\hat{s}_t \in \mathbb{R}^{d+k}$, we then apply the algorithms directly on this $\hat{s}_t$ without further considering history or state estimation. In other words, we construct an augmented dataset $\mathcal{D}_{\text{augmented}} = \{(\hat{s}_t, c_{t+1}, \hat{s}_{t+1})_{t=1}^{N}\}$ and apply our algorithms as if we have access to the environment state—namely as if we are in the fully observable setting. All the algorithms we consider use a neural network f to compute the prediction $f_{w_t}(\hat{s}_t)$, where w_t are the parameters of the neural network. The predictions may either be GVF predictions or n-step time series predictions, with the algorithms described in the next two sections.

4.2 Algorithms for GVFs

The goal for GVF predictions it to estimate the expected discounted sum of future cumulants, as described in Sect. 3.2. The simplest approach is to simply use the textbook 1-step temporal difference (TD) learning (Sutton and Barto, 1998). Data is processed as a stream, one sample at a time. The approach is summarized in Algorithm 1.

We can also adapt this update to an offline dataset. We can use TD offline, making multiple passes over the data set, and updating the network weights via mini-batches. Here we follow the standard approach used in offline RL for the fully observable setting. In other words, we can treat each tuple $(\hat{s}_t, c_{t+1}, \hat{s}_{t+1})$ separately, without having to keep the data in order. In contrast, if we were using a recurrent neural network, we would need to maintain the dataset order more carefully. In each epoch, we shuffle the dataset $\mathcal{D}_{\text{augmented}}$ and update the neural network using a mini-batch TD update. Algorithm 2 summarizes the approach.

We use the Adam optimizer (Kingma and Ba, 2015) to update with the mini-batch TD updates. We set all but the stepsize η to the typical default values: momentum parameter to 0.9, exponential average parameter to 0.99, and the small constant in the normalization to 10^{-4}. The algorithm returns the state of the optimizer—such as the

exponential averages of squared gradients and momentum—because our online variants continue optimizing online starting with this optimizer state.

Algorithm 1 OnlineTD

Hyperparameters: stepsize $\eta > 0$
Initialize $\boldsymbol{w_0}$: the weights of the network (e.g., uniform)
Obtain initial observation o_t for $t = 0$, set $\hat{s}_0 = o_0$
while in deployment **do**
    Observe next observation o_{t+1} and cumulant c_{t+1}
    $\hat{s}_{t+1} \leftarrow U(o_{t+1}, \hat{s}_t)$ ▷ compute augmented observation vector
    $v_{t+1} \leftarrow f_{w_t}(\hat{s}_{t+1})$ ▷ compute prediction
    $\delta_t \leftarrow c_{t+1} + \gamma v_{t+1} - f_{w_t}(\hat{s}_t)$ ▷ compute the TD error
    $w_{t+1} \leftarrow w_t + \eta \delta_t \nabla f_{w_t}(\hat{s}_t)$ ▷ or Adam using $-\delta_t \nabla f_{w_t}$ as a gradient
    $t \leftarrow t + 1$
end while

Algorithm 2 OfflineTD

Hyperparameters: stepsize $\eta > 0$, batchsize k, number of epochs n_{epochs},
Input $\mathcal{D}_{\text{augmented}} = \{(\hat{s}_t, c_{t+1}, \hat{s}_{t+1})\}$
Initialize w: the weights of the network (e.g., uniform)
Initialize s_{opt}: the state of the optimizer (e.g., zero momentum, zero exponential average)
for epoch in n_{epochs} **do**
    for batch in $\mathcal{D}_{\text{offline}}$ **do**
        $\Delta \leftarrow -\frac{1}{k} \sum_{i \in \text{batch}} \left(c_{i+1} + \gamma f_w(\hat{s}_{i+1}) - f_w(\hat{s}_i)\right) \nabla f_w(\hat{s}_i)$
        $w, s_{\text{opt}} \leftarrow \texttt{opt}(w, \Delta, \eta, s_{\text{opt}})$
    end for
end for
Return w, s_{opt}

Algorithm 3 OnlineTD using Offline Pretraining

Hyperparameters: offline stepsize $\eta > 0$, batchsize k, number of epochs n_{epochs}, online stepsize $\alpha > 0$, number of replay steps n_{replay}
Input $\mathcal{D}_{\text{augmented}} = \{(\hat{s}_t, c_{t+1}, \hat{s}_{t+1})\}$
$\boldsymbol{w_0}, s_{\text{opt}} = \text{OfflineTD}(\mathcal{D}_{\text{augmented}}, \eta, k, n_{\text{epochs}})$
Initialize the replay buffer $\mathcal{B}$ with last n_{replay} samples in $\mathcal{D}_{\text{augmented}}$
Obtain initial observation o_t for $t = 0$, set $\hat{s}_0 = o_0$
while in deployment **do**
    Observe next observation o_{t+1} and cumulant c_{t+1}
    $\hat{s}_{t+1} \leftarrow U(o_{t+1}, \hat{s}_t)$ ▷ compute augmented observation vector
    $v_{t+1} \leftarrow f_{w_t}(\hat{s}_{t+1})$ ▷ compute prediction
    $\delta_t \leftarrow c_{t+1} + \gamma v_{t+1} - f_{w_t}(\hat{s}_t)$ ▷ compute the TD error
    $w_{t+1}, s_{\text{opt}} \leftarrow \texttt{opt}(w_t, -\delta_t \nabla f_{w_t}(\hat{s}_t), \alpha, s_{\text{opt}})$
    $t \leftarrow t + 1$
end while

We expect purely offline methods to perform poorly in our non-stationary (partially observable) setting compared with those that also update in deployment. The offline data may not perfectly reflect what the agent will see in deployment, and, in general, tracking—namely updating with the most recent data—can also help under partial observability.

We can combine the offline and online methods, by pre-training offline and then allowing the agent to continue learning online. The primary nuance here is that we can either continue to use a replay buffer to update online or switch to the simplest online variant of TD that simply updates once per sample. We found that the simpler update was typically just as good as the variant using replay, so we use this simpler variant in this work. We summarize this procedure in Algorithm 3, and for completeness include the replay variant and results comparing to it in Appendix C.

It is worth mentioning that we could further improve these algorithms with the variety of advances combining TD and neural networks. TD methods can diverge when used with neural networks (Tsitsiklis and Van Roy, 1997), and several new algorithms have proposed gradient-based versions of TD that resolve the issue (Dai et al., 2017, 2018; Patterson et al., 2022). In control, a common addition is the use of target networks, which fix the bootstrap targets for several steps (Mnih et al., 2015). We found for our setting that the simple TD algorithm was effective, so we used this simpler approach.

Algorithm 4 OnlineNStep using Offline Pretraining

Hyperparameters: offline stepsize $\eta > 0$, batchsize k, number of epochs n_{epochs}, online stepsize $\alpha > 0$
Input $\mathcal{D}_{\text{n-step}} = \{(\hat{s}_t, c_{t+n})\}$
$\boldsymbol{w_0}, s_{\text{opt}} = \text{OfflineNStep}(\mathcal{D}_{\text{n-step}}, \eta, k, n_{\text{epochs}})$
Create size n circular array PastStates set index ind $\leftarrow 0$
Obtain initial observation o_t for $t = 0$, set $\hat{s}_0 \leftarrow o_0$
PastStates[ind] $\leftarrow \hat{s}_0$, and ind $\leftarrow 1$
for $n - 1$ steps **do** ▷ store first n inputs
    Observe next observation o_{t+1} and cumulant c_{t+1}
    $\hat{s}_{t+1} \leftarrow U(o_{t+1}, \hat{s}_t)$
    PastStates[ind] $\leftarrow \hat{s}_{t+1}$
    $t \leftarrow t + 1$ and ind $\leftarrow$ ind $+ 1$
end for
ind $\leftarrow 0$
while in deployment **do**
    Observe next observation o_{t+1} and cumulant c_{t+1}
    $(s, c) \leftarrow (\text{PastStates[ind]}, c_{t+1})$
    $\Delta \leftarrow \big(f_{w_t}(s) - c\big) \nabla f_{w_t}(s)$
    $w_{t+1}, s_{\text{opt}} \leftarrow \texttt{opt}(w_t, \Delta, \alpha, s_{\text{opt}})$
    $\hat{s}_{t+1} \leftarrow U(o_{t+1}, \hat{s}_t)$
    PastStates[ind] $\leftarrow \hat{s}_{t+1}$
    $t \leftarrow t + 1$ and ind $\leftarrow \text{mod}(\text{ind}, n)$
end while

4.3 Algorithms for n-step predictions

We can similarly consider the offline and online variants of n-step predictions. The offline dataset[3] consists instead of $\mathcal{D}_{\text{n-step}} = \{(\hat{s}_t, c_{t+n})\}_{t=0}^{N-n}$ where we predict the cumulant n steps into the future from t, given the approximate state $\hat{s}_t$. The targets for GVF predictions were returns G_t—discounted sums of cumulants into the future—whereas the targets for n-step predictions are the cumulants exactly n steps in the future. Learning f_w offline corresponds to a regression problem on this dataset, which can be solved using any standard techniques. Similarly to OfflineTD, we use stochastic mini-batch gradient descent and the Adam optimizer.

As a supervised learning problem, it is straightforward to update in deployment, online. However, there is one interesting nuance here, that the targets are not observed until n steps into the future. The online algorithm, therefore, has to *wait* to update the prediction $f_w(\hat{s}_t)$ until it sees the outcome c_{t+n} at time step $t + n$. This involves

[3] The underlying data is the same as in the TD setting, but the targets are different, and so we explicitly construct a supervised learning dataset from this underlying data.

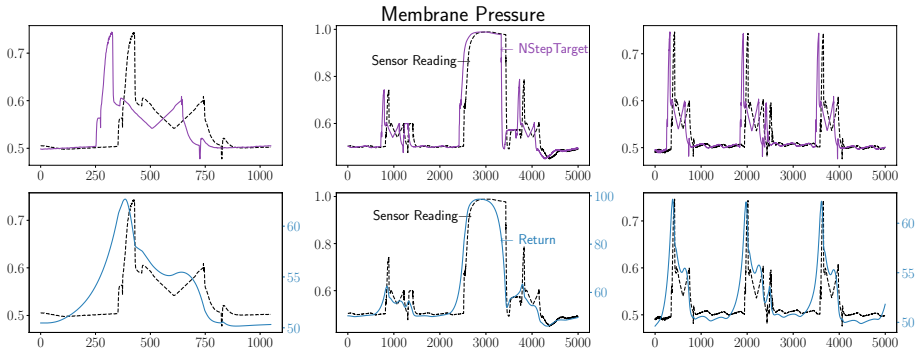

Fig. 8 Samples of Membrane Pressure sensor time-series and the corresponding idealized GVF and n-step predictions. Consider the first row of subplots. Each subplot shows a different snippet of time: the dashed black line depicts the time-series of Membrane Pressure and the purple line shows the corresponding prediction target or ideal prediction. Since the purple line is an ideal prediction of the black line, the purple line is shifted forward in time (to the left) revealing events before they occur in the dashed black time-series. The second row, similar to the first, compares the pressure sensor reading time-series and its ideal prediction for TD (labelled "return") (Color figure online)

maintaining a short buffer of size n, until we can obtain the pair $(\hat{s}_t, c_{t+n})$. This procedure is summarized in Algorithm 4.

Though seemingly a minor issue, it is less ideal that the OnlineNStep algorithm has to wait n steps to update the prediction for input $\hat{s}_t$. The TD algorithm for GVF predictions, on the other hand, does not have to wait to update, because it bootstraps off of its own estimates. Instead of using bootstrapping, we could have used a Monte Carlo algorithm, that regresses $\hat{s}_t$ towards computed returns, turning this into a supervised learning problem like for the n-step time series problem. However, it has been shown that being able to update immediately can result in faster tracking (Sutton et al., 2007; Sutton and Barto, 1998), and typically TD algorithms are preferred to Monte Carlo algorithms. The issue is worse for Monte Carlo than for n-step targets, because the returns extend further than n steps into the future, but nonetheless, there is some suggestive evidence that algorithms that need to wait could be disadvantaged.

N-step and GVF predictions are quite similar in the sense that their fundamental role is to summarize the future of a time-series, which is easy to see when looking at real data. Figure 8 plots the prediction targets for n-step predictions and GVF predictions (learned by TD) on real sensor data.

One might wonder why we chose to use the same agent-state construction and neural network for the N-step targets, as for the GVF targets, when there are many time series prediction approaches to chose from. Our primary reason is that we found this supervised approach to be effective, in terms of forecasting accuracy. This finding actually well-matches recent analysis, that highlights that for larger, multivariate time series data, neural network approaches can be more effective than the simpler time-series approaches (Hewamalage et al., 2021). Essentially, the nonlinear modeling power of neural networks becomes useful in these bigger data regimes, whereas the simpler methods remain preferable for the typically smaller datasets in the time series literature. We did test a time-series approach called NLinear that has been recently shown to be competitive with state-of-the-art prediction methods, including methods based on

transformers (Zeng et al., 2023; Zhang and Yan, 2023). The performance was worse than our proposed approach for the N-step predictions, as we discuss in Sect. 6.

4.4 Selecting hyperparameters for deployment

The above algorithms have many hyperparameters. Fortunately, we can use a simple validation strategy to select them, including the *online* stepsize parameters. The key idea is to treat the validation just like a deployment scenario, where the agent updates in temporal order on the dataset. For example, consider selecting the offline stepsize η and online stepsize α, assuming all other hyperparameters are specified (number of offline epochs is fixed, etc.). Then we can evaluate each hyperparameter pair (η, α) by

1. splitting the dataset into a training and validation set,
2. pre-training with offline stepsize η on the training set,
3. updating online with the stepsize α on the validation set (in one pass) as if it is streaming, recording the prediction accuracy as the agent updates.

The online prediction accuracy is computed as follows. For the current weights w_t, the agent gets $\hat{s}_t$ and makes a prediction $\hat{v}_t = f_{w_t}(\hat{s}_t)$. Because we (the experimenter) can peek ahead in the validation set, we can compute the error $\text{err}_t = (\hat{v}_t - c_{t+n})^2$. The agent, of course, cannot peek ahead, since it would not be able to do so in deployment. After going through the validation set once, we have our set of errors. Note that we only evaluate w_t on the pair $(\hat{s}_t, c_{t+n})$. Right after this step, we update the weights to get w_{t+1} and then evaluate the prediction under these new weights for the next step: $\hat{v}_{t+1} = f_{w_{t+1}}(\hat{s}_{t+1})$.

This validation procedure helps us pick a suitable pair of (η, α) precisely because validation mimics deployment. We want η to be chosen to produce a good initialization and we want α to be chosen to facilitate tracking when updating online. For example, if α is too big for tracking (or fine-tuning), then the validation error will be poor because the weights will move away from a good solution while updating on the validation set and the errors will start to get larger, resulting in a poor final average validation error. As another example, if η is too small and does not converge on the training set within the given number of epochs, then the initialization will not be as good and the validation errors will start higher than they otherwise could, until the online updating starts to reduce them.

Though this hyperparameter selection approach is described specifically for n-step predictions with the offline and online stepsizes, it can be used for TD as well as for other hyperparameters. The key point is that, even though the offline hyperparameters are only used on the training set and the online hyperparameters only when updating on the validation, they are both jointly evaluated based on validation error. The primary difference for TD is simply that the target is different. Again, because we the experimenter can look ahead in the data, we can simply compute the return on the future data, and compute the errors $\text{err}_t = (f_{w_t}(\hat{s}_t) - G_t)^2$.

5 Experimental setup

We investigate a scenario where the agent pre-trains on offline data and it's prediction accuracy is then tested in deployment. The agent makes predictions on every time step in deployment, and we can retroactively check the accuracy of those predictions once we see

the future—either after N steps or after enough steps to compute the return. For our experiments, we simply collect a test set and then have the agent predict incrementally on this test set, as if it is in deployment. This is perfectly equivalent to making predictions on the real system, but importantly allows us to run different algorithms on the same deployment (test) data. Further, it allows us to take our year of collected data, and select different time periods to split into train and test.

We consider two different scenarios: learning on 4 days, testing on 1 day and learning on 30 days, testing on 7 days. Most of our experiments consist of a dataset of five consecutive days of data from the middle of November 2022. The first four days are used as the offline training logs while the fifth day is used for the deployment phase. We use the final 4k steps of the offline training logs as the validation data, which is used for selecting hyperparameters. We also re-run all of experiments on another time period of five consecutive days in May—chosen because water conditions will be notably different from November—to ensure conclusions are not specific to November data.

For the final experiment, we use data from the duration of an entire month: the data from the entire month of November is used as the offline training logs and the next week (December 1st to December 7th) is used as the deployment data. The final 7 days of the offline training logs are used as the validation set. In addition, we subsample, so that the timescale of sensors readings is every 10 s, rather than every second. The goal of this final experiment is to test the agent in a setting where deployment is further from training, likely making it more important to update during deployment. Note also that this final setting is more challenging, because the time horizon itself is further: a 100-step prediction for the five day data is 100 s in the future, whereas it is 1000 s for this final experiment.

All the methods share similar settings. We train a 2-layer feed-forward neural network with 512 units in each layer with ReLU activation functions. The input to the network is an augmented observation vector of length 384, which is constructed by concatenating the 185 sensor values from the plant[4] with 185 memory traces and an additional 14 inputs encoding the plant mode (this is described in detail in Sect. 4 and Appendix A). In total, the network contains 722,945 weights. In the experiments that use five days of data, this network is optimized for 4000 epochs using the Adam optimizer with an L2 weight decay rate of $\lambda = 0.003$ and a batch size of 512, in the offline phase. After the offline training phase ends, we save the optimizer state variables and use them to initialize the optimizer during the deployment phase. In deployment, the algorithms update using one sample at a time, and use a different online step-size.

For all the methods we use the validation procedure described in Sect. 4.4 to select the step-size parameter. We swept over offline learning rates $\eta \in \{1\text{x}10^{-3}, 1\text{x}10^{-4}, 1\text{x}10^{-5}, 1\text{x}10^{-6}, 1\text{x}10^{-7}\}$ and online learning rates $\alpha \in \{1\text{x}10^{-4}, 1\text{x}10^{-5}, 1\text{x}10^{-6}, 1\text{x}10^{-7}, 1\text{x}10^{-8}\}$. The validation procedure is done separately for each algorithm and sensor.

[4] The raw sensor vector was length 480. We removed all constant sensor readings, leaving 185 sensors.

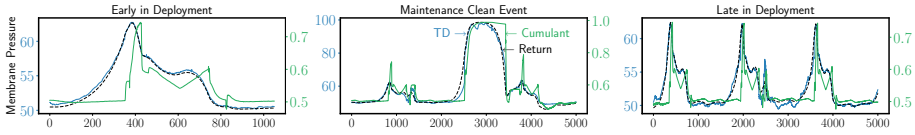

Fig. 9 Predictions of the filter membrane pressure roughly 100 s into the future. The plot shows the pressure sensor in green labelled cumulant (whose magnitude corresponds to the right y-axis). We show three snippets of the deployment data. The first subplot shows (on the x-axis) a thousand time steps (seconds) at the beginning of deployment. The middle subplot shows data during a maintenance clean, and the last subplot features data near the end of the deployment phase (24 h later). Each subplot highlights a different characteristic pattern in pressure change. The blue curve shows the TD prediction, first trained offline, then updated in deployment. The return represents the ideal prediction and is plotted in black. Note both the TD prediction and the return use the left blue axis. The TD predictions tightly match the target's pattern in all three scenarios (Color figure online)

6 Experiments and results

A natural first question is can we predict the time series well in deployment, given the size, complexity, and partially observable characteristics of our data. From there we contrast the GVF predictions to n-step predictions, to better understand the GVF results relative to a well-understood multistep prediction. Finally, we investigate one of the key claims in this work: does learning in deployment help or is offline learning all we need?

GVF predictions are accurate in deployment The object of our first set of results is to gain some intuition about GVF predictions. Although widely used in RL to model the utility or value of a policy, exponentially weighted predictions are uncommon. In Fig. 9 we visualize predictions from the OnlineTD approach[5] on one sensor at three different periods of time in deployment. Here we plot the cumulant (sensor value to be predicted into the future), the prediction, and the return—our stand-in for an *idealized prediction*. The time series of the return changes before the cumulant, because the return summarizes the future values of the cumulant. A good prediction should closely match the return as we see in the figure.

In the middle subplot of Fig. 9 we see a large perturbation in the cumulant corresponding to a difficult to predict event. This event, a maintenance clean, happens in the early morning. This causes a large increase in pressure on the filter, and unlike the vast majority of the training data, this increase is sustained for a long period of time. We can see the prediction correctly anticipates this event but does not get the precise shape of the prediction correct.

The particular time of year had a minimal impact on the quality of predictions learned. Figure 14, in the appendix is a replication of Fig. 9 with different training and deployment data, but the same sensor. We used the same architecture, preprocessing, and training scheme as described in the previous section and we see the predictions closely match the return as before.

[5] All of our results are with pre-training, as this performed significantly better than without using the offline data at all. This result is to be expected. Furthermore, our OnlineTD algorithm with pre-training also leverages the offline data to automatically set all hyperparameters, providing a fully specified algorithm. The conclusion for our setting is that it simply makes the most sense to leverage offline data, rather than learning from scratch.

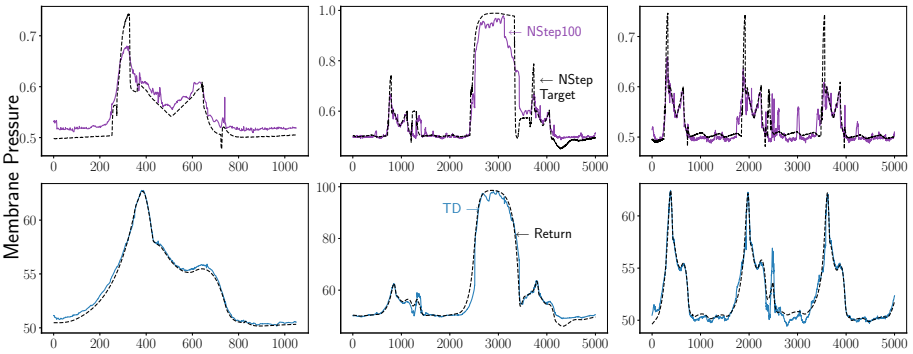

Fig. 10 Comparing GVF predictions (blue) and n-step predictions (purple) of filter membrane pressure. The top row shows the n-step predictions on the same three segments of deployment data used in Fig. 9. As before, the x-axis is time-steps or seconds. Here we only plot the prediction (labelled TD and NStep100), and the ideal prediction (labelled return and NStep target). Although both types of predictions are well aligned with their respective targets, however, sometimes the n-step prediction is off. Figure 11 includes the results for several other sensors (Color figure online)

Comparing GVF and n-step predictions To the uninitiated, GVF predictions can seem somewhat alien. To help calibrate our performance expectations, and provide a point of comparison, we also learned and plotted the more conventional 100-step predictions of future membrane pressure in deployment in Fig. 10. We chose a horizon of 100 steps to provide rough alignment with the horizon of a $\gamma = 0.99$ GVF prediction. The horizon for a GVF prediction is typically said to be about $\frac{1}{1-\gamma}$ (Sutton et al., 2011). The figure shows the n-step prediction and the GVF prediction on the same segments of data in deployment.

The plot of the n-step prediction and the shifted cumulant (labelled NStep Target) should align if the predictions are accurate. At least for membrane pressure, the GVF predictions better match their prediction target (the return) compared with n-step predictions.

To get a better sense of the quality of these learned predictions, we also compared against a simple linear baseline commonly used in time-series forecasting. This method called NLinear has been recently shown to be competitive with state-of-the-art prediction methods, including methods based on transformers (Zeng et al., 2023; Zhang and Yan, 2023). NLinear simply learns a linear map from a history of the normalized sensor values to the n-step future target. We experimented with a short history length (336) closer to the length of the prediction horizon ($n = 100$), and a much longer history (4000). The much longer history performed better, but generally replicated the periodicity of the sequence in its future predictions, overall leading to much worse performance compared to our n-step baseline. The results can be found in Fig. 15 in the Appendix.

Generally, across sensors, the learned GVF predictions are smoother than their n-step counterparts as shown in Fig. 11. This is perhaps to be expected because the γ weighting in the GVF prediction targets smooths the raw sensor data. If there are sharp, one-time-step spikes, as we see in the Inlet Pressure date, the n-step target itself will be spikey—that is, the ideal prediction is not smooth. Otherwise, the main objective of Fig. 11 is to allow you the reader to better understand GVF predictions by simply visually comparing them with n-step predictions—something that is easy to interpret and you might have more natural intuitions for.

One perhaps surprising conclusion from Fig. 11 is that the GVF and n-step predictions look surprisingly similar, and thus it is reasonable to ask if there are reasons to prefer one

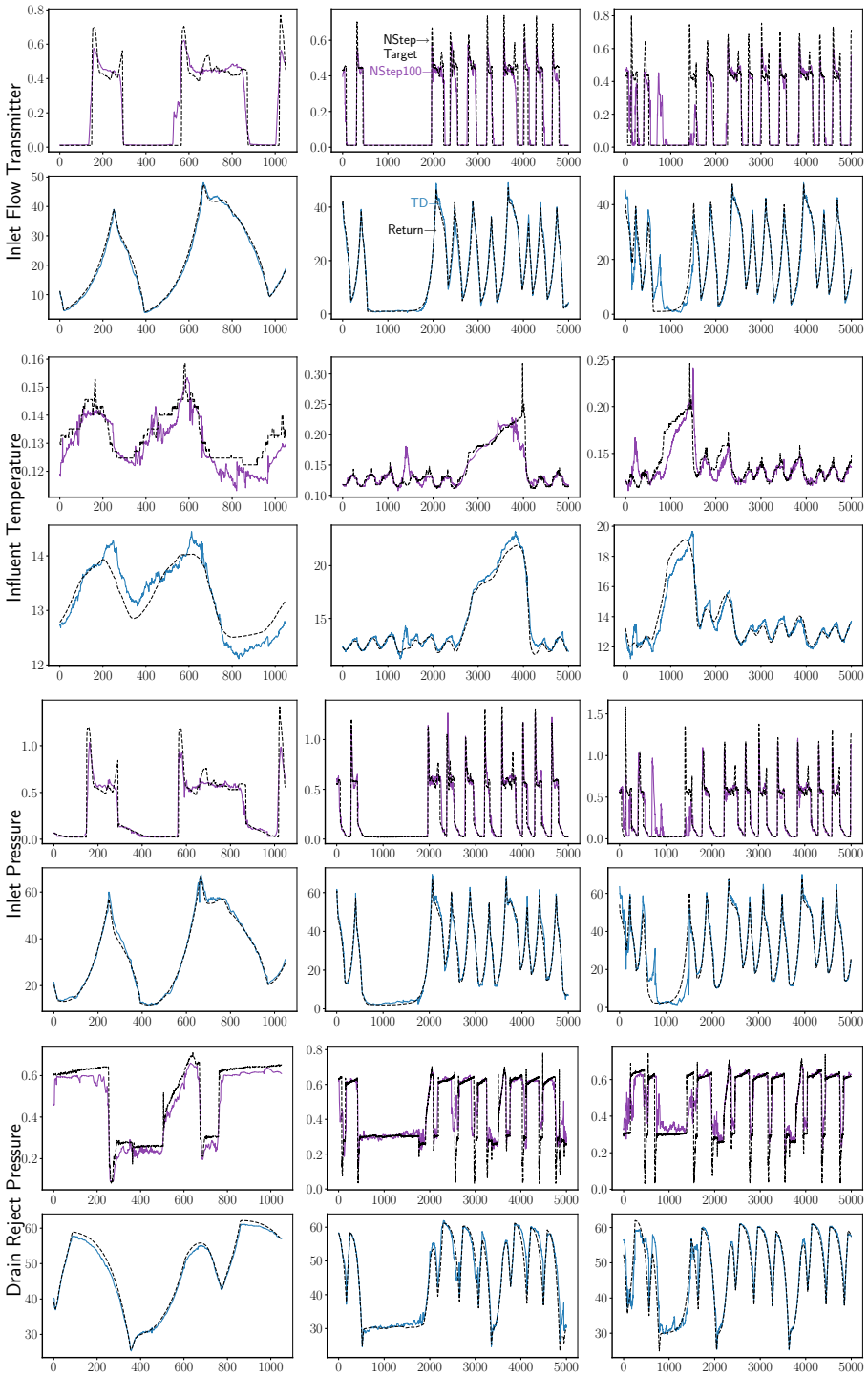

Fig. 11 Comparing n-step predictions (Row 1) and GVF Predictions (Row 2) across several sensors. The structure of this plot mirrors Fig. 10 (Color figure online)

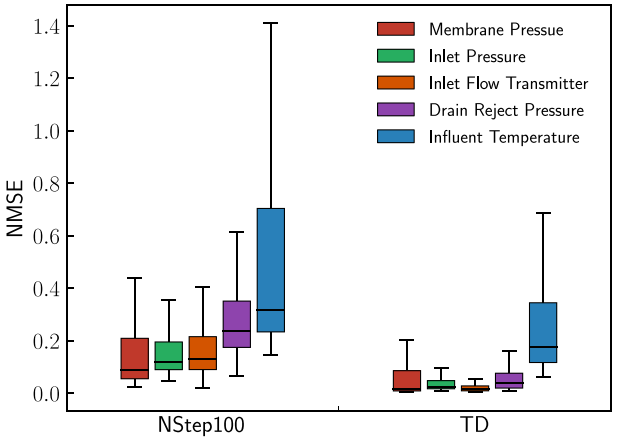

Fig. 12 Standard box plots generated using the sequences of Normalized MSE observed throughout the entirety of the deployment data for GVF and n-step predictions on 5 different sensors. The horizontal line in the middle denotes the median value. The top and bottom box boundaries represent the 75th and 25th percentiles, respectively. The whisker boundaries are drawn at the data point that is located closest to the distance of 1.5 times the Interquartile Range. Note that the NMSE values for TD and NStep100 are not directly comparable since their prediction targets are different. However, in both cases, it is desirable to have a lower NMSE value. The outliers are not plotted in order to make the visualization easier (Color figure online)

to the other. From a performance perspective, we compare the two in Fig. 12 reporting the Normalized Mean Squared Error (NMSE) over the deployment data:

$$\text{NMSE}_t \doteq \frac{\text{MSE}_t}{\overline{\sigma^2(G_t)}} \quad \text{where} \quad \text{MSE}_t \doteq \overline{(\hat{v}_t - G_t)^2}$$

and $\overline{x}$ denotes the exponentially weighted moving average of the squared GVF prediction error over the deployment data. Similarly, $\overline{\sigma^2(G_t)}$ denotes the variance of the returns up to time t computed using an exponentially weighted variant of Welford's online algorithm (Welford, 1962). The NMSE is equivalent to the variance unexplained and is a simple ratio measure of the MSE of the predictor to the MSE of the mean prediction. NMSE less than one indicates that the prediction explains more variance in the data than a mean prediction. We use the exponential moving average variants of these measures because our data is non-stationary. Finally, NMSE for the n-step prediction can be computed by replacing G_t with $o^{[i]}_{t+100}$ in the above equations. Across five sensors, the NMSE is lower for GVF predictions compared with n-step predictions, as shown in Fig. 12.

Algorithmically, GVF predictions are interesting for several reasons. GVF predictions can be updated, via TD, online and incrementally from a stream of data, whereas n-step predictions involve storing the data and waiting 100 steps until the prediction targets are observed. The longer the prediction horizon, the longer the system must wait without updating the predictions in between. In contrast, TD methods by their recursive construction have memory and computational requirements independent of the prediction horizon—independent of γ. These points highlight the potential of GVF predictions for time-series prediction, as an additional choice for multistep predictions. For any given application, the ultimate choice of prediction type and learning method will be driven by many factors.

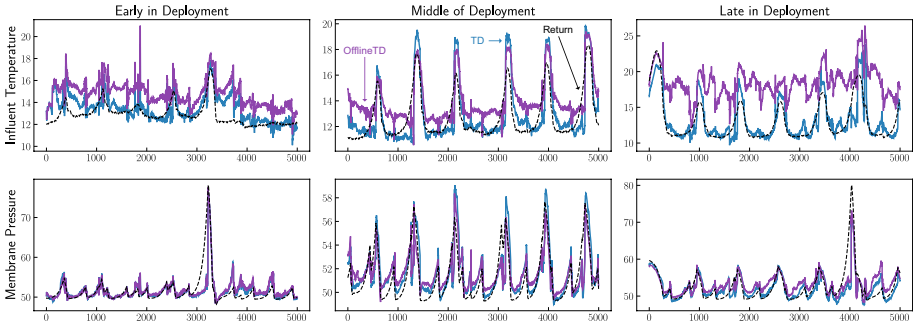

Fig. 13 OfflineTD and OnlineTD predictions of Influent Temperature and Membrane Pressure on deployment data a week after training data had ended. We sub-sample the data at the rate of 1 sample per 10 time-steps. Both agents are trained offline on data from November 1st to November 23rd and validated using the data from November 24th to November 30th. The deployment data was taken from a 7-day period from December 1st to December 7th. Mirroring previous figures, the x-axis reports time steps (in tens of seconds) in deployment. In this case, we expect a significant distribution shift between training and deployment data. The result clearly shows this, both predictors start off far from the ideal target (return); *predictions updated online in deployment (OnlineTD) can close the gap* (Color figure online)

Mitigating partial observability via online adaption As shown extensively in Sect. 2, the data from our plant is highly partially observable, appearing non-stationary when plotted. For the smaller dataset we used in the experiments so far, however, we find that the agent trained only on offline data predicts the deployment data well. This outcome is not surprising because the training data was collected from only four days of operation and the deployment was the following 24-hour period. It is reasonable to expect that the data is mostly stationary during this period since there would be no major seasonal weather changes, unexpected events like fires are rare, and sensor fouling takes weeks to show up in the data stream.

To highlight the need for online learning and demonstrate how changes in the data can significantly impact non-adaptive approaches, we used a dataset of 23 days for training, 7 days of validation followed by the next 7 days treated as the deployment phase. In order to reduce the size of this dataset, we sub-sample to a rate of once for every 10 s, rather than every second. This also makes the prediction task harder: now a 100 step prediction corresponds to 1000 s. Figure 13 compares GVF predictions of a frozen pre-trained agent with the online TD agent that was pre-trained on the sample data but continues to learn in deployment. Due to the differences in the training data and the deployment data, both predictors start far from the ideal prediction (the return), but only online TD can adjust as shown in the first subplot. Throughout the remainder of the deployment data online TD predictions continued to match their targets.

This result not only highlights when online methods can be beneficial but also mimics a fairly realistic deployment scenario. Oftentimes, when working with real systems, we cannot always access the most recent data. An industrial partner might have limited data logs; or sometimes technical problems cause logs to be lost. In our specific application, water treatment, the training data might be out of date because the plant could have been out of commission. Regardless of the reason, it is useful that simple online methods like TD can adapt to such situations.

We pre-trained the Influent Temperature predictions for 25,000 epochs and that of Membrane Pressure for 50,000 epochs. Due to resource constraints, the hyper-parameter

selection is done differently in this experiment: we swept over offline learning rates $\eta \in \{1\text{x}10^{-5}, 1\text{x}10^{-6}, 1\text{x}10^{-7}\}$ and saved the network with the lowest normalized MSE on the training data. Afterwards, we used the best saved network and swept over 20 online learning rates $\alpha \in [1\text{x}10^{-3}, ..., 2\text{x}10^{-8}]$ range generated through a geometric progression with a common ratio of 0.5 on the validation data.

7 Conclusion and future work

In this paper we took the first steps toward optimizing and automating water treatment on a real plant. Before we can hope to control such a complex industrial facility, we must first ensure that learning of any kind is feasible. This paper represents such a feasibility study. We provided extensive visualization and analysis of our plant's data, highlighting how it generates a large, high-dimensional data stream that exhibits interesting structure at the second, minute, day, and month timescales. Unlike the data commonly used in RL benchmarks, ours is subject to seasonal trends, and mechanical wear and tear, making it highly non-stationary. Through a combination of feature engineering and extensive offline pre-training on the operator data, we were able to learn accurate multi-step predictions encoded as GVFs. Compared with classical n-step methods used in time-series predictions, the GVF predictions were more accurate and could be learned incrementally in deployment.

The next steps for this project involve control: automating subproblems within water treatment. There are numerous such subproblems, for example, controlling the rate at which chemicals are added in pre-treatment. Backwashing is also promising because it is, by far, the most energy-intensive part of the operation. We could control the duration of backwashing or how often to backwash. At the lowest level, we can adapt the parameters of the PID controllers that control the pumps during backwashing. Classical PID controllers are not sensitive to the state of the plant; they are tuned when the plant is first commissioned and can become uncalibrated over time.

Algorithmically, we plan to investigate using our learned predictions for control, directly. Traditionally, one would define a reward function and use a reinforcement learning method such as Actor-Critic to directly control aspects of the plant operation. These methods are notoriously brittle and difficult to tune. A more practical approach is to use the predictions to directly build a controller. Prior work has explored using learned predictions inside basic if-then-else control rules to control mobile robots (Modayil and Sutton, 2014). The advantage of this approach is that the control-rules are easy to explain to human operators, but since control is triggered by predictions that are continually updated in deployment the resultant controller adapts to changing conditions. An extension of this idea is to use GVF predictions—like the ones we learned in this work—as input to a neural-network based RL agent, similarly to how it was done for autonomous driving (Graves et al., 2020; Jin et al., 2022). This work provides the foundations for these next steps in industrial control with RL.

Appendix A: Details on construction of state

As discussed in Sect. 2, learning directly on the raw data from a Water Treatment Plant (WTP) is very challenging due to the noisy, stochastic and partially-observable nature of the data. In addition to this, different sensors operate at different timescales and

Table 2 Summary of a few sensors measuring pump speeds, setpoints valves, blowers, and PID control

Sensor name	Measures
Feed flow PID	PID control for feed flow
Pump flow PID	PID control for feed/drain pump flow
Permeate pump flow PID	PID control for permeate pump flow
Feed water sample	Condition of feed water sampling valve, indicating if it is open or not
Post flocculation sample	Condition of post flocculation sample isolation valve, indicating if it is open or not
Process/permeate pump control speed output	Speed control for process/permeate pump
Sulphuric acid pump dose speed	Speed of sulphuric acid pump dosing
Hypochlorite pump	Hypochlorite pump dosing
Sodium hydroxide pump dose speed	Sodium hydroxide pump dosing
Citric acid pump	Citric acid pump dosing
Feed inlet valve	Condition of feed inlet valve, indicating if it is open or not
Feed/waste pump inlet	Condition of feed/waste pump inlet valve, indicating if it is open or not
Feed/waste pump outlet	Condition of feed/waste pump outlet valve, indicating if it is open or not
Membrane tank outlet valve	Condition of membrane tank outlet valve, indicating if it is open or not
Membrane tank recirculation Valve	Condition of membrane tank recirculation valve, indicating if it is open or not
Permeate pump recirculation Valve	Condition of permeate pump recirculation valve, indicating if it is open or not
Permeate outlet value	Condition of permeate outlet valve, indicating if it is open or not
BP/CIP tank inlet valve	Condition of cleaning tank inlet valve, indicating if it is open or not
BP/CIP tank recirculation valve	Condition of cleaning tank recirculation valve, indicating if it is open or not
Blower inlet valve	Condition of inlet blower's valve (A/B/C), indicating if it is open or not
Membrane aeration blower control speed output	Control speed output of membrane aeration blower
Aeration controller	Mode of the aeration (cyclic, constant, etc.)
Plant mode	Mode of the plant (production, backwashing, etc.)

All of these combine to form the input to our learning system

frequencies; we summarize some of the sensors in Table 2. In order to minimize the effect of these issues on the predictions, we take a series of preprocessing steps on the raw data that are described in the sections below.

Note that we do not have significant missing data issues. Our system rarely misses sensor readings. However, in the rare case where we do have a missing value, we simply use zero-imputation and fill in the missing values with zeros.

A.1 Categorical observations

Some of the observations are recorded in the form of discrete categorical variables as opposed to continuous real numbers. One example is the observation which records the current mode of the plant. For such observations, we encode them in a one-hot vector format. For a categorical observation which can only take on values from one of k categories: we convert it into a binary vector of size k in which only the corresponding index of the category is set to 1.

A.2 Data normalization

Since different sensors have different ranges, we normalize their values into the [0, 1] range. For each individual sensor value $o_t^{[i]}$ in the observation vector, we compute the minimum $o_{min}^{[i]}$ and maximum $o_{max}^{[i]}$ using the logs over the duration of a previous year, where i is an index within the observation vector. Afterwards, for every discrete time-step $t = 1, 2, \ldots$, in our dataset, we compute the normalized sensor value $o'^{[i]}_t$ as:

$$o'^{[i]}_t = \frac{o_t^{[i]} - o_{min}^{[i]}}{o_{max}^{[i]} - o_{min}^{[i]}} \tag{A1}$$

A.3 Encoding time of day

The observations contain the information regarding the current time of the day in seconds. This is important since there are certain events that happen at a particular time of the day. Additionally, there are some events that are repeated at regular intervals. Let $\mathcal{S} = [s_0, s_1, s_2, \ldots]$ denote the time-stamp in seconds for that day, then we encode it using sine and cosine transforms as:

$$s_t^{(sin)} = \sin\left(\frac{2\pi s_t}{86400}\right) \tag{A2}$$

$$s_t^{(cos)} = \cos\left(\frac{2\pi s_t}{86400}\right) \tag{A3}$$

where 86,400 is the total number of seconds present within a day. It is the maximum value that s_t can take.

A.4 Encoding plant mode length

Understanding which mode the plant is in, and when the mode change will happen is crucial for the agent. This information is only available as a binary indicator, as mode value 1 against a certain mode indicates that the plant is currently in this mode, while it is 0 otherwise. This limitation to binary indication adds to the partial observability inherent in the state-space. We find that cyclically encoding the mode provides extra information that

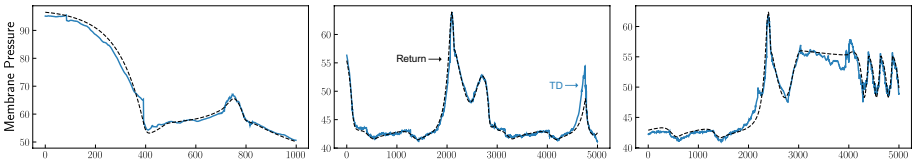

Fig. 14 Predictions of the filter membrane pressure roughly 100 s into the future *using data from May*. This plot mirrors Fig. 9. The architecture, preprocessing, learning algorithms, GVF prediction, and sensor predicted are all the same. The only difference is that Fig. 9 reports the predictions learned from data from the month of November, and this plot uses data from May. As before, the TD predictions tightly match the target's pattern in all three scenarios (Color figure online)

alleviates this associated partial observability. Since the agent has access to when the mode starts and ends as binary indicators, we utilize it to construct a cyclical thermometer encoding of the mode.

For each mode indicator in our observation vector, we define two thermometers $w_{sine}^{[i]}, w_{cos}^{[i]} \in \mathbb{R}^7$ initialized to zeros, and the total mode length $m_l^{[i]} \in \mathbb{R}$. Let s be the timestamp in seconds. Since the mode is characterized with respect to an observation that is observed at a certain time-step, we avoid explicitly denoting mode length with time-step for clarity. At each time-step, the thermometers get filled up as:

$$w_{sine}[j] = \sin\left(2^j \pi \left(\frac{s}{m_l^{[i]}}\right)\right), w_{cos}[j] = \cos\left(2^j \pi \left(\frac{s}{m_l^{[i]}}\right)\right)$$

These thermometers have sine and cosine waves between the start and end of each mode, and their rotations about the period increase by a factor of 2^j at every time-step.

A.5 State approximation and summarizing history

In order to make use of the historical information during predictions, we compute memory traces of the observations. The state is constructed by appending these memory traces to the original observation vector, in addition to the mode length and time of day described in the above two sections. For each normalized observation $o'^{[i]}_t$ at time-step t, we compute its memory trace $z_t^{[i]}$ using:

$$z_t^{[i]} = \beta z_{t-1}^{[i]} + (1-\beta) o'^{[i]}_t \tag{A4}$$

where β is the trace decay rate hyper-parameter. All the memory traces are initialized with zeros and are updated in an incremental manner when iterating over the dataset.

Appendix B: Additional results

In order to verify that our results are not dependent on the time of year that we use for training, we perform an additional experiment with the same experimental setup that was used to produce Fig. 9. The only difference between the experiment in Figs. 9 and 14 is that the former is trained on the data from November, 2022 whereas the latter is trained on the

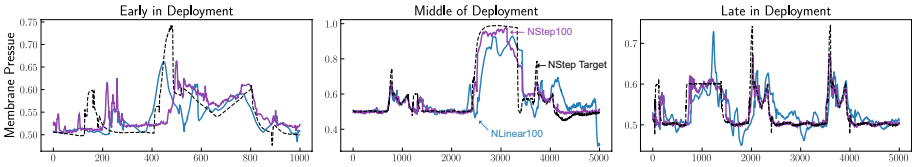

Fig. 15 N-step predictions of the filter membrane pressure roughly 100 s into the future learned using NLinear. We show three snippets of the deployment data. The first subplot shows (on the x-axis) 1000 time-steps (seconds) at the beginning of deployment. The middle subplot shows data during a maintenance clean, and the last subplot features data near the end of the deployment phase (24 h later). Each subplot highlights a different characteristic pattern in pressure change. The blue curve shows the $n = 100$ step prediction learned by NLinear: first trained offline, then updated in deployment. The return represents the ideal n-step prediction and is plotted in black. The blue curve shows the $n = 100$ step prediction learned by our non-linear n-step baseline described in Sect. 4.3 (Color figure online)

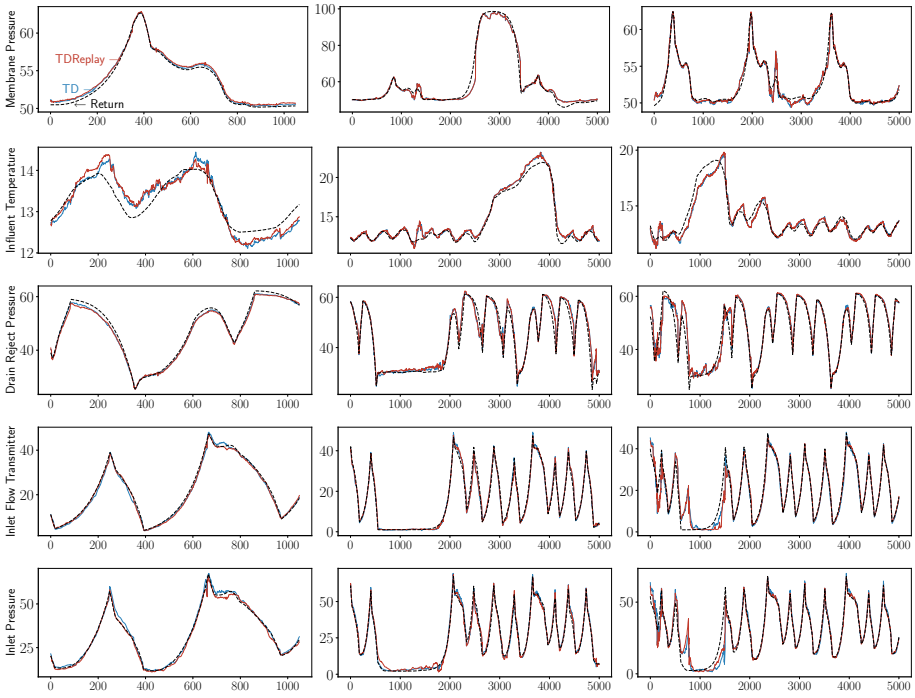

Fig. 16 Comparing online TD and online TD with replay for GVF Predictions across several sensors. The structure of this plot mirrors Fig. 10 (Color figure online)

data from May, 2023. These results show that the predictions closely match the return even when the data is taken from a different period of time.

In Fig. 15, we evaluate NLinear and compare it with the N-step method the exact same data as in Fig. 9. We used a prediction horizon of n=100 and a history length (look-back window) of 4000 time-steps. We compare the NLinear results with the N-step since both of these methods share the same prediction target i.e. the cumulant value that is 100 steps into the future. These results showcase the limitations of the

NLinear methods: they can only make correct predictions when there is a clear periodicity within the data. Additionally, the look-back window size needs to be large enough so that the linear layer can make future predictions according to the repeated past patterns. As a result of this, linear methods are simply insufficient for learning a good prediction model on our current dataset.

Appendix C: Comparing to TD with replay

We considered both the simpler online TD update in deployment, as well as using TD with replay. The TD with replay algorithm is summarized in Algorithm 5. We found, though, that they performed very similarly (see Fig. 16), so we used the simpler online TD update in the main body.

Algorithm 5 TDwithReplay using Offline Pretraining

Hyperparameters: offline stepsize $\eta > 0$, batchsize k, number of epochs n_{epochs}, online stepsize $\alpha > 0$, number of replay steps n_{replay}
Input $\mathcal{D}_{\text{augmented}} = \{(\hat{s}_t, c_{t+1}, \hat{s}_{t+1})\}$
$\boldsymbol{w_0}, s_{\text{opt}} = \text{OfflineTD}(\mathcal{D}_{\text{augmented}}, \eta, k, n_{\text{epochs}})$
Initialize the replay buffer $\mathcal{B}$ with last kn_{replay} samples in $\mathcal{D}_{\text{augmented}}$
Obtain initial observation o_t for $t = 0$, set $\hat{s}_0 = o_0$
while in deployment **do**
    Observe next observation o_{t+1} and cumulant c_{t+1}
    $\hat{s}_{t+1} \leftarrow U(o_{t+1}, \hat{s}_t)$
    $v_{t+1} \leftarrow f_{w_t}(\hat{s}_{t+1})$
    $\delta_t \leftarrow c_{t+1} + \gamma v_{t+1} - f_{w_t}(\hat{s}_t)$
    $w \leftarrow w_t + \alpha\delta_t \nabla f_{w_t}(\hat{s}_t)$
    Add tuple $(\hat{s}_t, c_{t+1}, \hat{s}_{t+1})$ to $\mathcal{B}$
    for n_{replay} steps **do**
        Sample a random mini-batch `batch` of size k from $\mathcal{B}$
        $\Delta \leftarrow -\frac{1}{k} \sum_{i \in \text{batch}} \left(c_{i+1} + \gamma f_{\boldsymbol{w_t}}(\hat{s}_{i+1}) - f_w(\hat{s}_i)\right) \nabla f_w(\hat{s}_i)$
        $w, s_{\text{opt}} \leftarrow \texttt{opt}(w, \Delta, \alpha, s_{\text{opt}})$
    end for
    $w_{t+1} = w$
    $t = t + 1$
end while

Author Contributions All authors made significant contributions to the project. Janjua, Shah, and Miahi wrote all the code and ran all the experiments. White, White and Machado lead the project, helped write the paper and make plots, and funded the work. All authors contributed to writing the paper.

Funding This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant program and the Canada CIFAR AI Chairs program. Computational resources provided by Digital Research Alliance of Canada.

Availability of data and materials We are working with our industrial partner to open-source the data. This requires approval from several levels including the town municipal government. We hope to have this done by camera ready.

Code availability We are working with our industrial partner to open-source the code. This requires approval from several levels including the town municipal government. We hope to have this done by camera ready. Naturally, we cannot open source any code involved in commercialization, and thus we are extracting the algorithmic code for release.

Declarations

Conflict of interest Not applicable.

Ethical approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

Bellemare, M. G., Candido, S., Castro, P. S., Gong, J., Machado, M. C., Moitra, S., Ponda, S. S., & Wang, Z. (2020). Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature, 588*(7836), 77–82.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on empirical methods in natural language processing*.

Copeland, C., & Carter, N. T. (2017). *Energy-water nexus: The water sector's energy use* (CRS Report No. R43200, Washington).

Crone, S. F., Hibon, M., & Nikolopoulos, K. (2011). Advances in forecasting with neural networks? Empirical evidence from the nn3 competition on time series prediction. *International Journal of Forecasting, 27*(3), 635–660.

Dai, B., He, N., Pan, Y., Boots, B., & Song, L. (2017). Learning from conditional distributions via dual embeddings. In *Artificial intelligence and statistics* (pp. 1458–1467). PMLR.

Dai, B., Shaw, A., Li, L., Xiao, L., He, N., Liu, Z., Chen, J., & Song, L. (2018). SBEED: Convergent reinforcement learning with nonlinear function approximation. In *International conference on machine learning*.

Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B. D., Carpanese, F., Ewalds, T., Hafner, R., Abdolmaleki, A., de Las Casas, D., Donner, C., Fritz, L., Galperti, C., Huber, A., Keeling, J., Tsimpoukelli, M., Kay, J., Merle, A., Moret, J., & Riedmiller, M. A. (2022). Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature, 602*(7897), 414–419.

Ernst, D., Geurts, P., & Wehenkel, L. (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research, 6*, 503–556.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., Ruiz, F. J. R., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., & Kohli, P. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature, 610*(7930), 47–53.

Graves, D., Nguyen, N. M., Hassanzadeh, K., & Jin, J. (2020). Learning predictive representations in autonomous driving to improve deep reinforcement learning. CoRR arXiv:2006.15110.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*.

Hausknecht, M. J., & Stone, P. (2015). Deep recurrent Q-learning for partially observable MDPs. In *AAAI fall symposia*.

Hewamalage, H., Bergmeir, C., & Bandara, K. (2021). Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting, 37*(1), 388–427.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780.

Jin, J., Graves, D., Haigh, C., Luo, J., & Jagersand, M. (2022). Offline learning of counterfactual predictions for real-world robotic reinforcement learning. In *International conference on robotics and automation (ICRA)* (pp. 3616–3623).

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *International conference on learning representations (ICLR)*.

Lange, S., Gabel, T., & Riedmiller, M. A. (2012). Batch reinforcement learning. *12*, 45–73.

Levine, S., Kumar, A., Tucker, G., & Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. CoRR arXiv:2005.01643.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., & Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *International Journal of Robotics Research, 37*(4–5), 421–436.

Luo, J., Paduraru, C., Voicu, O., Chervonyi, Y., Munns, S., Li, J., Qian, C., Dutta, P., Davis, J. Q., Wu, N., Yang, X., Chang, C., Li, T., Rose, R., Fan, M., Nakhost, H., Liu, T., Kirkman, B., Altamura, F., Cline, L., Tonker, P., Gouker, J., Uden, D., Bryan, W. B., Law, J., Fatiha, D., Satra, N., Rothenberg, J., Carlin, M., Tallapaka, S., Witherspoon, S., Parish, D., Dolan, P., Zhao, C., & Mankowitz, D. J. (2022). Controlling commercial cooling systems using reinforcement learning. CoRR arXiv:2211.07357.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2022). The M5 competition: Background, organization, and implementation. *International Journal of Forecasting, 38*(4), 1325–1336.

Mandhane, A., Zhernov, A., Rauh, M., Gu, C., Wang, M., Xue, F., Shang, W., Pang, D., Claus, R., Chiang, C., Chen, C., Han, J., Chen, A., Mankowitz, D. J., Broshear, J., Schrittwieser, J., Hubert, T., Vinyals, O., & Mann, T. A. (2022). MuZero with self-competition for rate control in VP9 video compression. CoRR arXiv:2202.06626.

Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Nazi, A., Pak, J., Tong, A., Srinivasa, K., Hang, W., Tuncer, E., Le, Q. V., Laudon, J., Ho, R., Carpenter, R., & Dean, J. (2021). A graph placement methodology for fast chip design. *Nature, 594*(7862), 207–212.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., & Ostrovski, G. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533.

Modayil, J., & Sutton, R. S. (2014). Prediction driven behavior: Learning predictions that drive fixed responses. In *Workshops at the 28th AAAI conference on artificial intelligence*.

Modayil, J., White, A., & Sutton, R. S. (2014). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior, 22*(2), 146–160.

Mozer, M. C. (1989). A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, *3*(4), 137–169.

Patterson, A., White, A., & White, M. (2022). A generalized projected Bellman error for off-policy value estimation in reinforcement learning. *Journal of Machine Learning Research*, *23*(1), 6463–6523.

Pietquin, O., Geist, M., Chandramohan, S., & Frezza-Buet, H. (2011). Sample-efficient batch reinforcement learning for dialogue management optimization. *ACM Transactions on Speech and Language Processing, 7*(3), 7–1721.

Rafiee, B., Abbas, Z., Ghiassian, S., Kumaraswamy, R., Sutton, R. S., Ludvig, E. A., & White, A. (2023). From eye-blinks to state construction: Diagnostic benchmarks for online representation learning. *Adaptive Behaviour, 31*(1), 3–19.

Riedmiller, M. A. (2005). Neural fitted Q iteration—First experiences with a data efficient neural reinforcement learning method. In *European conference on machine learning*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. CoRR arXiv:1707.06347.

Shortreed, S. M., Laber, E. B., Lizotte, D. J., Stroup, T. S., Pineau, J., & Murphy, S. A. (2011). Informing sequential clinical decision-making through reinforcement learning: An empirical study. *Machine Learning, 84*(1–2), 109–136.

Sutton, R. S., Koop, A., & Silver, D. (2007). On the role of tracking in stationary environments. In *Proceedings of the 24th international conference on machine learning* (pp. 871–878).

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., & Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *International conference on autonomous agents and multiagent systems* (pp. 761–768).

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning—An introduction*. MIT Press.

Swaminathan, A., Krishnamurthy, A., Agarwal, A., Dudík, M., Langford, J., Jose, D., & Zitouni, I. (2017). Off-policy evaluation for slate recommendation. In *Advances in neural information processing systems (NeurIPS)*.

Tao, R. Y., White, A., & Machado, M. C. (2023). Agent-state construction with auxiliary inputs. *Transactions on Machine Learning Research (TMLR)*.

Tsitsiklis, J. N., & Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control, 42*(5), 674–690.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature, 575*(7782), 350–354.

Watkins, C. J. C. H., & Dayan, P. (1992). Technical note: $Q-$*learning*. *Machine Learning*, *8*(3–4), 279–292.

Welford, B. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics, 4*(3), 419–420.

Won, D.-O., Müller, K.-R., & Lee, S.-W. (2020). An adaptive deep reinforcement learning framework enables curling robots with human-like performance in real-world conditions. *Science Robotics, 5*(46), 9764.

Zeng, A., Chen, M., Zhang, L., & Xu, Q. (2023). Are transformers effective for time series forecasting?. In *AAAI conference on artificial intelligence*.

Zhang, Y., & Yan, J. (2023). Crossformer: Transformer utilizing cross-dimension dependency for multivariate time series forecasting. In *The 11th international conference on learning representations*.