ORIGINAL ARTICLE

Approximately Optimal Fixed-Structure Controllers Using Neural Networks

Daniel G. McClement[1] | Nathan P. Lawrence[2] | Philip D. Loewen[2] | Michael G. Forbes[3] | Johan U. Backström[4] | R. Bhushan Gopaluni[1]

[1]University of British Columbia, Department of Chemical and Biological Engineering

[2]University of British Columbia, Department of Mathematics

[3]Honeywell Process Solutions

[4]Backström Systems Engineering Ltd.

Correspondence
Daniel G. McClement, Chemical and Biological Engineering, University of British Columbia, Vancouver, British Columbia, Canada
Email: daniel.mcclement@ubc.ca

Funding information
This project was funded with the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) and Honeywell Connected Plant.

Fixed structure controllers (such as proportional-integral-derivative controllers) are used extensively in industry. Finding a practical and versatile method to tune these controllers, particularly with imprecise process models and limited online computational resources, is an industrially relevant problem which could improve the efficiency of many plants. In this paper, we present two flexible neural network-based approaches capable of tuning any fixed structure controller for any control objective and process model and compare their advantages and disadvantages. The first approach is derived from supervised learning and classical optimization techniques, while the second approach applies techniques used in deep reinforcement learning. Both approaches incorporate model uncertainties when selecting controller parameters, reducing the need for costly experiments to precisely estimate model parameters in a plant. Both methods are also computationally efficient online, enabling their widespread usage.

Keywords — Process control, optimization, machine learning, PID control

1 | INTRODUCTION

Fixed-structure controllers represent the vast majority of controllers in use today. The most common example, the proportional-integral-derivative (PID) controller, is used in roughly 90% of control loops in industry [30]. Many fixed-structure controllers require frequent re-tuning as equipment ages or is replaced and feedstocks change. Re-tuning can be burdensome, considering any single plant typically contains thousands of control loops. A maintenance-free tuning method for these controllers is an appealing way to improve the efficiency and output quality of many processes while lowering costs.

Significant amounts of research have gone into the tuning of fixed-structure controllers, however almost every method is limited to certain situations: they may only apply in continuous time, or for single input single output, stable systems, or be limited to PID tuning, or a certain performance index. Moreover, some of these tuning methods depend on an accurate process model and do not incorporate model uncertainty into their tuning. Incorporation of model uncertainties into the tuning of controllers is crucial for the development of maintenance-free controllers as the accurate fitting of model parameters can often require costly perturbations to a process. Maintenance-free tuning methods must consider the uncertainties when re-tuning controller parameters.

Recently, an optimal fixed-structure controller tuning method using stochastic programming techniques has been developed by [23]. This approach takes model uncertainties into account by simulating many different realizations of a plant given by the model uncertainty and selecting the parameters which optimize a given performance index across all these simulations. This approach is appealing because it requires no assumptions about the structure of the process model or controller. However, this stochastic programming approach is computationally expensive which limits its use in certain situations. For example, it may not be feasible to optimize many control loops at once, incorporate this technique into an embedded controller with limited computing power, or update controller tunings in real time as better model parameter estimates become available. It is desirable to capture the flexibility of stochastic programming (compatible with any process model or controller structure, incorporates model uncertainties) while reducing the online computation required for implementation.

The objective of this paper is to develop a controller tuning method which retains the benefits of stochastic programming, while also requiring significantly less online computation to make its application more practical. We experiment with two different methods. First, we train a neural network to learn an explicit approximation of the stochastic programming optimization solution. Second, we train a simple reinforcement learning agent to select the controller parameters given a plant model with parameter uncertainties, eliminating the need for any stochastic programming. In both cases, we are essentially creating a set of tuning rules parameterized by a neural network to be used by fixed-structure controllers. The performance of these two methods as well as their relative advantages and disadvantages is demonstrated through a case study of tuning PID controllers for a wide variety of first order systems.

2 | BACKGROUND

2.1 | Deep Learning

Each layer of a feedforward neural network produces a column vector $x^{(l)}$ by processing the outputs of the previous layer, $x^{(l-1)}$, as follows. The inputs are multiplied by a *weight matrix*, W. Then a *bias vector*, b, is added to the product. Finally, an *activation function*, σ (which is usually nonlinear), produces the output for layer l:

$$x^{(l)} = \sigma(Wx^{(l-1)} + b). \tag{1}$$

Each component of $x^{(l)}$ is called a *neuron*, and the positive integer $\dim(x^{(l)})$ gives the *width* of layer l. The elements of W and b (and sometimes the choices for σ) typically vary from layer to layer.

Multilayer feedforward neural networks have received a significant amount of attention because they are universal function approximators [10]. As long as a neural network is sufficiently wide, it can theoretically model any arbitrarily complex function with only two layers (an input and an output layer). This property is appealing for our objective of predicting the optimal fixed-structure controller parameters given information about a plant's dynamics and the control objectives which is some unknown, nonlinear function which could be very complex in some cases. With sufficient training data, we can guarantee a neural network can learn an optimal mapping from the plant dynamics to fixed-structure controller parameters. A neural network "learns" by adapting the parameters W and b in each layer to produce a desired input-output relationship. Usually, these parameters are improved iteratively, by applying stochastic gradient descent to a loss function that penalizes discrepancies between the neural network output, $\hat{y}$, and the desired output, y.

More recently, it has been shown that "deep" neural networks (containing at least one additional layer between the input and output layers), can allow for more complex feature construction and make neural networks require far fewer parameters and data to train [9].

Activation functions (σ in (1)) allow neural networks to learn nonlinear functions. Throughout this paper, only one non-linear activation function is used: the rectified linear unit (ReLU), Equation 2.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2}$$

ReLU activations are very popular as they have good gradient propagation properties, are computationally efficient, and are scale-invariant, among other benefits [11] [7]. The identity function ($\mathrm{Id}(x) = x$) is often used as the activation for the output layer of neural networks.

Deep learning has successfully been used to learn explicit approximations of model predictive (optimization-based) controller outputs [16], [22]. Rather than learning to predict the optimal controller output given the state of a system as these past researchers have done, this work focuses on learning the optimal fixed-structure parameters of a controller given the present knowledge about the system dynamics.

2.2 | Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning in which an agent learns an optimal policy for interacting with its environment [27]. In process control applications, the desired "policy" is generally a feedback controller, π, and the environment is analogous to the plant the controller interacts with. The situation can be modeled as a stochastic Markov Decision Process where the instantaneous state of the environment is represented as a vector, $\boldsymbol{s}$. Based on $\boldsymbol{s}$, the policy π prescribes an action, $\boldsymbol{a} = \pi(\boldsymbol{s})$, which drives the system to some new state, $\boldsymbol{s}'$. The agent in RL is a software surrogate for an apprentice control engineer, whose role is to iteratively improve the policy/controller design π. Each state transition described above is associated with a reward $r = f(\boldsymbol{s}, \boldsymbol{a})$, which depends on both the desirability of the new state and the cost of the selected action. By monitoring interactions between the current policy and the environment, the RL agent can forecast the long-term reward of the current strategy π and modify it to achieve maximum long-term reward. In this analogy, the senior control engineer is a human who specifies the reward function f that influences the development of the policy by quantifying the desirability of various operating characteristics for the closed-loop system. The simplest forms of RL start with no outside knowledge about the dynamics of the system to be controlled. The agent's adjustments to π are based entirely on observations of actual state-to-state transitions and associated rewards observed as the system runs.

There has been growing interest in applying RL to process control [19, 24, 21, 2, 13]. Its exclusively data-driven, model-free approach provides a partial explanation. Researchers have explored ways to directly control processes (i.e., replace conventional controllers) with RL [26, 17]. Others have explored ways to use RL to automatically tune fixed-structure controllers online [5, 15]. However, the RL research into tuning fixed-structure controllers generally depends on online-learning and experimenting with different controller settings on the actual plant. This perturbation can negatively effect the process and learning can be slow, resulting in sub-optimal control for extended periods.

This paper uses a type of RL called Deep Q-Learning [18]. Here Q is the standard notation for the state-action value function, namely, the discounted sum of future rewards the agent expects to receive under a given policy:

$$Q(\boldsymbol{s},\boldsymbol{a}) = \mathbb{E}\left[\sum_{i=0}^{N} \gamma^i r(\boldsymbol{s}_i,\boldsymbol{a}_i) \middle| \boldsymbol{s}_0 = \boldsymbol{s}, \boldsymbol{a}_0 = \boldsymbol{a}\right] \tag{3}$$

The sum above involves a discount factor γ ($0 < \gamma < 1$) which prioritizes immediate rewards over rewards in the distant future; mathematically, the decay of γ^i as i grows prevents the sum from diverging as the number of time steps simulated (N) increases. (We streamline the notation in (3) by not explicitly naming the underlying policy π for which $\boldsymbol{a}_{i+1} = \pi(\boldsymbol{s}_i)$, $i = 0, 1, \ldots$.)

Every fixed policy determines an associated state-action value function via (3). With such a Q in hand, and the system in some state $\boldsymbol{s}$, the best action $\boldsymbol{a}$ is the one that *maximizes* $\boldsymbol{a} \mapsto Q(\boldsymbol{s},\boldsymbol{a})$—assuming we intend to follow the policy that defines Q for all future steps. In general, the maximizing action for a given state $\boldsymbol{s}$ might not match the action prescribed by the policy defining Q. But such a discrepancy would imply that the long-term reward available in state $\boldsymbol{s}$ is not achieved by the current policy. For a policy π to be *optimal*, the corresponding state-action value function Q must return the optimal reward from every state $\boldsymbol{s}$, and specify the optimal action, i.e.,

$$\pi(\boldsymbol{s}) = \operatorname*{argmax}_{\boldsymbol{a}} Q(\boldsymbol{s},\boldsymbol{a}). \tag{4}$$

This is one manifestation of Bellman's *principle of optimality*: every single step along an optimal trajectory must be impossible to improve. This can also be expressed by saying, for all state-action pairs,

$$Q(\boldsymbol{s},\boldsymbol{a}) = r(\boldsymbol{s},\boldsymbol{a}) + \gamma \max_{\boldsymbol{a}'} \mathbb{E}_{\boldsymbol{s}'}\left[Q(\boldsymbol{s}',\boldsymbol{a}')\right], \tag{5}$$

where $\boldsymbol{s}' = \boldsymbol{s}'(\boldsymbol{s},\boldsymbol{a})$ denotes the state of the system produced by applying action $\boldsymbol{a}$ from state $\boldsymbol{s}$.

2.3 | PID Controllers

The most common type of fixed-structure feedback controller combines proportional, integral, and derivative (PID) terms based on the set point tracking error $e := y_{\text{sp}} - y$. A PID control law can be written in parallel form as

$$u(t) = K_P\, e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{d}{dt} e(t) \tag{6}$$

Specifying the PID controller's three scalar parameters (K_P, K_I, K_D) is a matter of immense industrial relevance, and many approaches have been proposed. Notable techniques include robust frequency-domain optimization [8], however this approach requires the level of robustness to be chosen in advance and is not directly related to the confidence in the plant model which may change over time. Simpler tuning methods such as Skogestad's internal model control (SIMC) are easy to implement, require no real-time optimization, and make the closed-loop plant behave according to a reference first or second order model [25]. However, an SIMC controller's performance depends heavily on the accuracy of the model used to tune it, and the controller may not perform well on systems with significant nonlinearities. SIMC-based PID controllers are often "detuned", meaning the closed-loop time constant of the system specified by the control engineer is set conservatively in case of model uncertainties.

Among many other tuning methods, the recent stochastic programming-based approach of [23] stands out due to its strong robustness guarantees across process disturbances and modelling uncertainties. The framework is also not PID-specific and can be applied to any fixed-structure controller (higher order systems, multiple input/multiple output systems, cascade control, discrete-time control, etc.) as well as any control objective or set of constraints.

2.4 | Stochastic Programming

Stochastic programming seeks to find the controller parameters, $\boldsymbol{a}$, which minimize some user-defined objective function, J, across all possible realizations of the process dynamics[1], $\boldsymbol{s}$ [3]. We treat each model parameter describing the plant dynamics as a random variable. Given the physical constraints imposed by the plant dynamics and the design constraints imposed by the controller's fixed structure, the problem is

$$\min_{\boldsymbol{a}} \mathbb{E}[J(\boldsymbol{a}, \boldsymbol{s})] \tag{7}$$

[23] optimize controller parameters based on a risk measure (conditional value at risk) to ensure good performance in extreme realizations of $\boldsymbol{s}$. For simplicity, we stick to the expected value of the performance index, but acknowledge more sophisticated performance indices which better incorporate risk could be used to better guarantee performance in extreme cases. [23] also point out the controller parameters can be optimized for different set points or process disturbances as well as model uncertainties. Our focus will remain on optimizing for different model uncertainties.

3 | METHODOLOGY

In this section, we will explain how our techniques, using both a supervised learning approach and an RL approach, can be applied to the tuning of PID controllers for first order systems.

For a first order system, we consider six factors relevant to selecting PID parameters:

1. K, the process gain
2. τ, the open-loop time constant
3. θ, the process dead time
4. $\sigma_K, \sigma_\tau, \sigma_\theta$, the standard deviations of our estimates for K, τ, and θ
5. I, a boolean indicating whether the system is integrating ($I = 1$) or not ($I = 0$)
6. The control performance index to be optimized (e.g., integral absolute error, integral squared error, etc.)

---

[1] We represent the possible realizations of the process dynamics with the same variable, $\boldsymbol{s}$, as the RL state. This is intentional as a description of the process dynamics will be used as the RL state later in this paper.

Lines 1–5 above describe the 7 scalar components of the vector state $\boldsymbol{s}$ in our machine learning model, as follows

$$\boldsymbol{s} = \begin{bmatrix} \hat{K} & \hat{\theta} & \hat{\tau} & \sigma_K & \sigma_\tau & \sigma_\theta & I \end{bmatrix} \tag{8}$$

Here $\hat{K}$, $\hat{\tau}$, and $\hat{\theta}$ are mean values about which the corresponding true parameters are normally distributed with the indicated standard deviations, e.g., $K \sim \mathcal{N}(\hat{K}, \sigma_K^2)$. (Our methods apply equally well to other probability distributions.) The control performance index (Line 6 above) is used to train the machine learning model and is used during the data collection process rather than as an input to the neural network. Our methodology can easily by applied to higher order systems or multiple input/output systems; the number of inputs to the machine learning model would be increased to accommodate the additional model parameters.

We define the control performance index, J, as the integral squared error (ISE) given a system and its parameter uncertainty $\boldsymbol{s} \sim \mathcal{S}$ when the system (starting at set point) undergoes a unit step change to a new set point. $\mathcal{S}$ represents the set of all possible systems and their associated parameter uncertainties within a range of values of interest. The collection of model parameters used to train our model is shown in Table 1.

TABLE 1 The range of model parameters used to train the machine learning model.

Model Parameter	$\hat{K}$	$\hat{\tau}$	$\hat{\theta}$	$\frac{\sigma_K}{\hat{K}}$	$\frac{\sigma_\tau}{\hat{\tau}}$	$\frac{\sigma_\theta}{\hat{\theta}}$	I
Minimum	0.1	1	0.01	0	0	0	0
Maximum	10	10	10	0.5	0.5	0.5	1
Distribution	Log-uniform	Uniform	Log-uniform	Uniform	Uniform	Uniform	Bernoulli

The process gains we survey span 2 orders of magnitude — if a process gain for an actual plant falls outside this range, the input data can be easily scaled to fall within this range as long as the order of the process gain is estimated *a priori*. Extending the range of gains our tuning method can accommodate is also straightforward.

While the time constant and time delay parameter ranges may seem to limit our model, these values are unitless: if an appropriate time scale is selected, our model will work for any system where the time constant is 0.1 – 1000 times the time delay; a very large range that most industrially relevant systems can reasonably be expected to fall within.

Data needs to be collected across many different simulated processes to train the neural networks. The process gain and time delay span multiple orders of magnitude, so $\hat{K}$ and $\hat{\theta}$ are sampled log-uniformly, e.g., $\log(\hat{K}) \sim \mathcal{U}(-1, 1)$.

The standard deviations for the parameters are normalized by the parameter estimate's value so the uncertainty corresponds with the magnitude of the parameter (an uncertainty of ±1 would be reasonable for a process gain of 10, but unreasonable for a process gain of 0.1). The actual parameters can range from being exactly equal to the parameter estimate ($\frac{\sigma_K}{\hat{K}} = 0$) to ±50% of the parameter's value ($\frac{\sigma_K}{\hat{K}} = 0.5$).

As detailed below, we use $\boldsymbol{a}$ to denote a vector formed from the gains of a PID controller:

$$\boldsymbol{a} = \begin{bmatrix} K_P & K_I & K_D \end{bmatrix}. \tag{9}$$

We restrict each component of $\boldsymbol{a}$ to be nonnegative.

The optimal PID parameters can be found by solving the stochastic optimization problem (10), below. The “optimal" parameters depend on the performance index J determined by a plant operator. In this paper, our performance index is a discretization of the ISE with an additional penalty on the change in control action at each time step, Δu, to avoid overactive control and wear on actuators. We write $e(t)$ for the set point tracking error at time t.

$$\begin{aligned} \boldsymbol{a}_{\text{opt}} &= \underset{\boldsymbol{a}}{\text{argmin}}\ J(\boldsymbol{a}|\boldsymbol{s}), \text{ where} \\ J(\boldsymbol{a}|\boldsymbol{s}) &= \mathbb{E}[\sum_{i=0}^{N} \left[e(i\Delta t)^2 + \Delta u(i\Delta t)^2 \right] \Delta t] \end{aligned} \tag{10}$$

The question is: given a vector $\boldsymbol{s}$ describing a class of systems, which PID parameters minimize the given control objective, J? In the following sections we outline two different approaches to creating a neural-network approximation to the mapping $\boldsymbol{s} \mapsto \boldsymbol{a}_{\text{opt}}$. Both methods involve solving an optimization problem and training a neural network. The order in which these two steps are performed (optimization first or neural network training first) gives two different approaches, each with its own advantages and disadvantages. Solving the optimization problem first, herein referred to as the "optimize then train" method, results in a supervised learning problem; training the neural network first, referred to as the "train then optimize" method, results in a problem formulation similar to reinforcement learning.

3.1 | An "Optimize then Train" Approach

Supervised learning trains a neural network to map a system class $\boldsymbol{s}$ to a parameter triple $\boldsymbol{a}$ by providing a large number of desirable input-output pairs to imitate and generalize. Thus, before the neural network can be trained, a data set of systems and their corresponding optimal PID parameters must be generated. To do this, we choose various $\boldsymbol{s}$ from $\mathcal{S}$ and use stochastic programming to solve for $\boldsymbol{a}_{\text{opt}}$ in (10).

Training a neural network in advance instead of seeking optimal PID parameters online pays off by moving computational effort out of the control loop (and into the offline training process). Evaluating the output of a trained neural network requires just a few matrix multiplications and vector additions, whereas solving (10) online would require significantly more computational resources and may be infeasible to implement across the thousands of PID loops in a plant simultaneously. It also would struggle to adapt to changing plant dynamics in real time. The supervised learning approach requires more computation offline during training, but very little computation once online.

3.1.1 | Training Data

Training data are generated using the stochastic programming approach presented by Renteria *et al.* [23]. First, a vector $\boldsymbol{s} \sim \mathcal{S}$ is drawn from the distribution of possibilities defined in Table 1. This $\boldsymbol{s}$ describes a collection of first-order systems, from which 100 choices are made by sampling the model parameters (e.g., $K \sim \mathcal{N}(\hat{K}, \sigma_K)$). The objective $\mathbb{E}(J(\boldsymbol{a}|\boldsymbol{s}))$ in (10) is replaced with the sample average of the values $J_i(\boldsymbol{a})$ over the 100 specific systems selected. The same triple of PID parameters, $\boldsymbol{a}$, is applied across all 100 realizations. We used PySP, a Python library for stochastic programming [29] and the interior point optimizer (Ipopt) software package [28] to perform the optimization.

The system dynamics are simulated using an explicit Euler discretization with a step size of $\Delta t = 0.1$. We experimented with a 4^{th} order accurate Runge Kutta discretization scheme to improve the accuracy of the "optimal" PID parameters, however this greatly increased the computational complexity of the stochastic programming problem, making its large scale use across thousands of different systems intractable. The number of time steps used in the simulation was adapted to the specific system class sampled from $\mathcal{S}$. Systems with faster dynamics required fewer steps to reach their final steady state after a set point change, necessitating fewer time steps be simulated and making the optimization problem quicker to solve.

The final data set consisted of 9500 unique draws from $\mathcal{S}$ (5000 self-regulating and 4500 integrating). A visualization of the data is shown in Figures 1 and 2. We can see that the optimal PID parameters follow trends expected when tuning PID controllers. For instance, the proportional gain is inversely correlated with the plant gain.

3.1.2 | Neural Network Training

No pre-processing of the PID data collected through stochastic programming was performed. The Monte Carlo simulations had a moderate number of samples due to the significant additional computational time required to add more. As a result, there was a chance the sampled distribution for any given Monte Carlo simulation may not have accurately matched the underlying system distribution. However, as the neural network is trained across a large meta-distribution of these Monte Carlo simulations, we expect the effects of unlucky sampling on any given Monte Carlo simulation to be reduced.

The PyTorch machine learning framework [20] was used to implement the neural network model. The data set was split into a 1000-sample testing data set and an 8,500-sample training data set. The training data set was further divided into 5 partitions for 5-fold cross validation. The Optuna Python library was used to optimize the network hyperparameters [1].

Different numbers of hidden layers, hidden layer sizes, initial learning rates, and different regularization penalties were experimented with and tuned. Bayesian experimental design [4] was used to efficiently explore the hyperparameter space with few experiments to avoid overfitting to the training data. (100 hyperparameter experiments were performed in total.) The network was relatively insensitive to most hyperparameter choices, with the exception of the initial learning rate. The final model structure is shown in Figure 3. Due to the rather small size of the training data set, complete gradient descent could be performed (as opposed to stochastic gradient descent). An L_2-regularization penalty of 10^{-5}, a learning rate of 0.003, and 100 epochs were used to train the final model. The adaptive moment estimator (Adam) optimizer was used to perform the gradient descent [14].

Feature transformations were also experimented with to improve the model's accuracy. Neural networks tend to converge faster when all features have the same scale and the feature distributions are not skewed. Due to the log-uniform sampling of the process gain and time delay, these features were positively skewed. We experimented with substituting in $\log(\hat{K})$ and $\log(\hat{\theta})$ to centre the distribution while simultaneously scaling these features to more closely match the normalized standard deviations. However, this modification had a negligible effect on the final model performance.

The last consideration for our model was the loss function used for training. Both mean squared error (L_2 loss) and mean absolute error (L_1 loss) were tested. Mean absolute error is a more robust loss function which was desirable given a small number of Monte Carlo simulations may have been outliers if the sampled parameters did not happen to be representative of their underlying distribution. We found using mean absolute error was able to help improve the accuracy of the model's PID parameter predictions compared to mean squared error.

3.2 | A "Train then Optimize" Approach

Rather than using RL to find the optimal controller parameters for a single process as researchers have previously tried, we consider a large distribution of possible processes and use a RL-inspired approach to build a set of controller tuning rules in advance of any use on a physical system. The resulting model (analogous to a RL agent) requires no

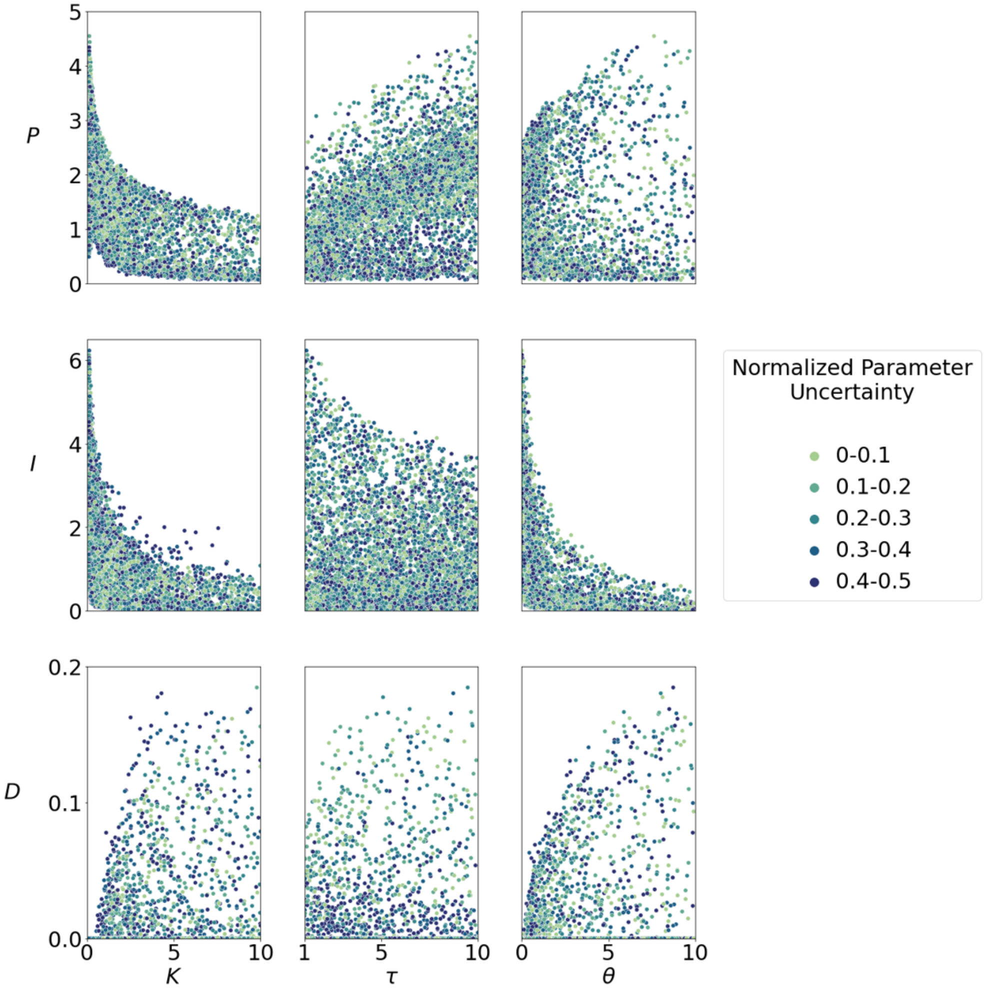

FIGURE 1 The approximately optimal PID parameters for self-regulating first-order systems plotted against the system's parameters. Color is used to denote the normalized parameter standard deviation ($n = 5000$).

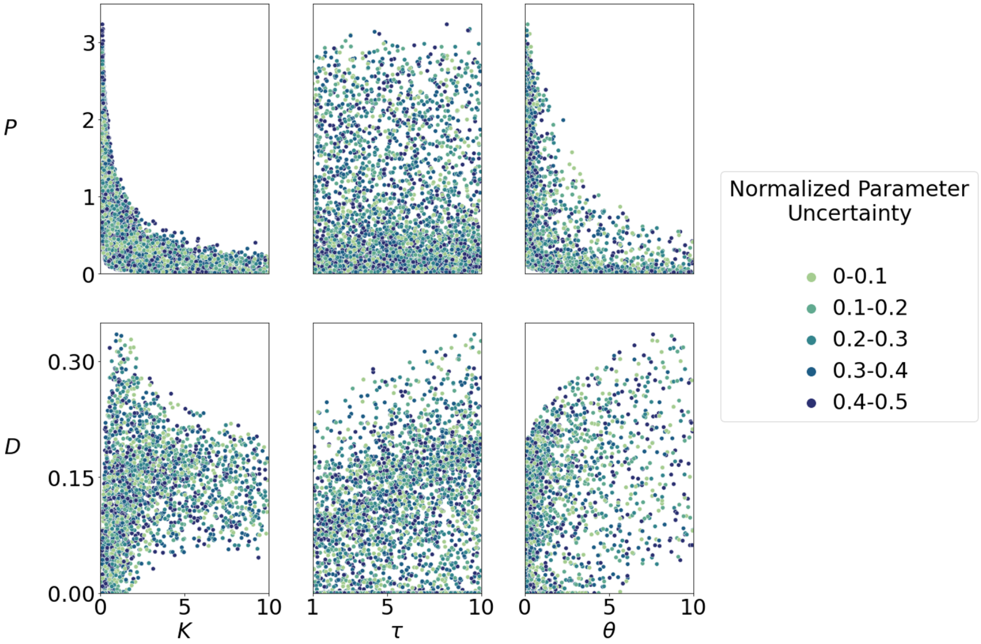

FIGURE 2 The approximately optimal PD parameters for integrating first-order systems plotted against the system's parameters. Color is used to denote the normalized parameter standard deviation ($n = 4500$). Only proportional and derivative gains are plotted as the optimal integral gain when optimizing for ISE is always zero on integrating systems.

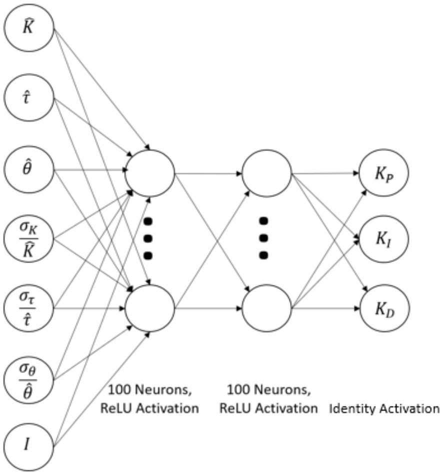
$\hat{K}$
$\hat{\tau}$
$\hat{\theta}$
$\frac{\sigma_K}{\hat{K}}$
$\frac{\sigma_\tau}{\hat{\tau}}$
$\frac{\sigma_\theta}{\hat{\theta}}$
I
K_P
K_I
K_D
100 Neurons, ReLU Activation
100 Neurons, ReLU Activation
Identity Activation

FIGURE 3 Structure of the neural network.

online learning, which results in significantly less perturbation to processes.

A RL-inspired approach to the optimal PID tuning problem allows us to bypass the computationally expensive stochastic programming data generation step required in the “optimize then train” framework. Rather than building a neural network which maps from $\boldsymbol{s} \rightarrow \boldsymbol{a}_{\text{opt}}$, we instead create a neural network, called Q, which maps from $[\boldsymbol{s}, \boldsymbol{a}] \rightarrow J$. It is very quick to simulate a system and set of PID parameters to calculate the objective function value (so long as we do not have to optimize the PID parameters). This makes collecting data to train Q significantly faster than collecting data to train the neural network in Section 3.1. We can use this network to solve for the PID parameters $\boldsymbol{a}$ which minimize the objective function given $\boldsymbol{s}$. In summary, we learn a state-action value function which we use to select the controller parameters which minimize the expected cost (akin to maximizing the expected reward):

$$\boldsymbol{a}_{\text{opt}} = \underset{\boldsymbol{a}}{\text{argmin}}\ Q(\boldsymbol{s}, \boldsymbol{a}) \tag{11}$$

By training the neural network first and using it as part of the optimization process afterwards, the optimization becomes significantly easier to solve. Rather than simulating and optimizing over thousands of time steps and dozens of different Monte Carlo samples, the optimization problem is reduced to a single-stage, deterministic problem.

This formulation also has a unique property which makes it easier to solve than traditional RL problems. Namely, the model’s actions have *no* influence over the environment’s state. Regardless of which PID parameters are selected, the process gain, time delay, etc., are unchanged. The Markov model representation of this problem would be a series of disconnected states. This eliminates the dependency of the value of one state on other states, simplifying the Q function. Usually, the optimal Q function satisfies

$$\begin{aligned} & Q(\boldsymbol{s}, \boldsymbol{a}) = r(\boldsymbol{s}, \boldsymbol{a}) + \gamma Q(\boldsymbol{s}', \pi(\boldsymbol{s}')), \\ \text{where} \qquad & \pi(\boldsymbol{s}) = \underset{\boldsymbol{a}}{\text{argmax}}\ Q(\boldsymbol{s}, \boldsymbol{a}) \end{aligned} \tag{12}$$

However, since $\boldsymbol{s}'$ must always be the same as $\boldsymbol{s}$, the state-action value function reduces to a scaled version of the reward:

$$Q(\boldsymbol{s}, \pi(\boldsymbol{s})) = \frac{r(\boldsymbol{s}, \boldsymbol{a})}{1 - \gamma} \tag{13}$$

For simplicity, we set $\gamma = 0$. In most RL problems, the Q network’s output is hard to interpret and is treated as a black box. However, the Q network’s output in our problem formulation has a clear definition: it is the expected value of the performance index for a given system and set of PID parameters. Not only can this neural network model be used to optimize the PID tunings, it can predict how well its tunings will perform on a given system as measured by the performance index specified by the user. This feature could be used to identify poor performing control loops at a plant which may need more sophisticated controllers or modifications made to the equipment to improve the quality of control.

Because there are no state transitions in the Markov Decision Process, the Q network function is not recursive and this could be viewed as a supervised learning problem where the neural network simply predicts the reward given a state-action pair. It would be possible to collect a large amount of data in the state-action space and train the Q network in a supervised fashion. However, the state-action space is ten-dimensional and would require large amounts of data to train across. By treating this problem as we would an RL problem where data is collected through the agent’s interactions with the environment, we can focus the training data set to parts of the state-action space the current iteration of the model indicates are most important (i.e. regions of the state-action space where the model predicts the optimal PID parameters are located), improving the sample efficiency. This rationale matches a new perspective being considered by RL researchers, who view RL as supervised learning on optimized data [6].

Algorithm 1 shows how the Q-network is trained. An ε-greedy approach is used, meaning that with probability $1 - \varepsilon$ we collect data according to what the Q network believes are the optimal PID parameters given a system $\boldsymbol{s}$, but decide with probability ε to test a random set of PID parameters. This helps explore other parts of the state-space and prevent the Q-network from converging to a poor local minimum. In cases where the predicted optimal PID parameters are selected, a small amount of Gaussian noise is added to further perturb the data collection and explore the region of the state-action space around the Q-network's predicted minimum more thoroughly.

Algorithm 1: Deep Q Network Training Algorithm

```
Output: Trained $Q$ Network
Initialize: Initial learning rate $\alpha$, initial $Q$ network parameters $w$, replay buffer $\mathcal{B}$, method to calculate the objective
            function $J(\boldsymbol{a}, K, \tau, \theta)$, batch size $N$
for each training episode do
    Sample state $\boldsymbol{s} \sim \mathcal{S}$
    Reward $r \leftarrow 0$
    Sample $\varepsilon \sim \mathcal{U}(0, 1)$
    if $\varepsilon < 0.05$ then
        $\boldsymbol{a} \leftarrow$ random PID parameters
    else
        $\boldsymbol{a} \leftarrow \operatorname{argmin}_{\boldsymbol{a}} Q_w(\boldsymbol{s}, \boldsymbol{a}) + \mathcal{N}(0, 0.02)$
    for each Monte Carlo sample do
        Sample $K, \tau, \theta \sim \boldsymbol{s}$
        Monte Carlo sample's reward $\leftarrow J(\boldsymbol{a}, K, \tau, \theta)$
    $r \leftarrow$ mean(Monte Carlo Sample Reward)
    Add $(\boldsymbol{s}, \boldsymbol{a}, r)$ to $\mathcal{B}$
    Sample batch of $N$ training examples $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}) \sim \mathcal{B}$
    $\hat{\boldsymbol{r}} \leftarrow Q_w(\boldsymbol{s}, \boldsymbol{a})$
    $\mathcal{L} \leftarrow \frac{1}{N}(\boldsymbol{r} - \hat{\boldsymbol{r}})^2$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla \mathcal{L}(\boldsymbol{w})$
```

The reward function used to train the Q-network also needed to be modified from the original cost function in Equation 10. Poor selection of **a** may destabilize the system and result in the cost function J achieving very large values, resulting in very large gradients in the Q-network which are disruptive during training. The reward function is shown in Equation 14. For reasonable selections of **a**, the reward is a scaled version of J. For unreasonable selections of **a** which destabilize the system, the reward saturates to avoid large values. A small $\log(J)$ term is added to ensure a non-zero gradient and make the Q-network easier to optimize online. $\log(600)$ is subtracted from r to keep the piecewise function continuous and hence easier to optimize.

$$r(\boldsymbol{s}, \boldsymbol{a}) = \begin{cases} \frac{1}{10} J(\boldsymbol{s}, \boldsymbol{a}) & J(\boldsymbol{s}, \boldsymbol{a}) \leq 600 \\ 60 + \log(J) - \log(600) & J(\boldsymbol{s}, \boldsymbol{a}) > 600 \end{cases} \tag{14}$$

Hyperparameter optimization is a more complex task in RL settings because the training data is itself dependent on the choice of hyperparameters. We tested several different Q network structures to find a good set of hyperpa-

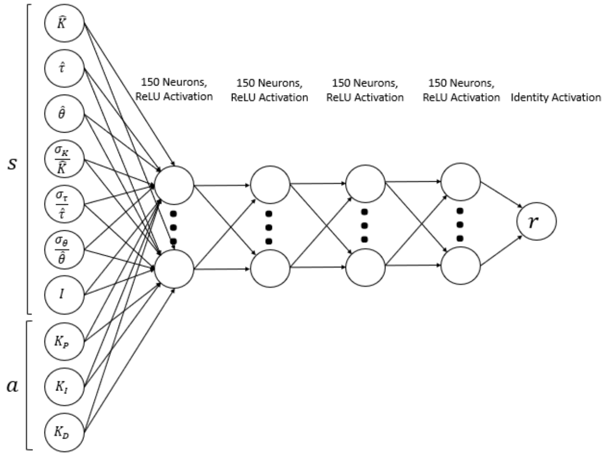
$\hat{K}$
$\hat{\tau}$
$\hat{\theta}$
$\frac{\sigma_K}{\hat{K}}$
$\frac{\sigma_\tau}{\hat{\tau}}$
$\frac{\sigma_\theta}{\hat{\theta}}$
I
K_P
K_I
K_D
s
a
150 Neurons, ReLU Activation
150 Neurons, ReLU Activation
150 Neurons, ReLU Activation
150 Neurons, ReLU Activation
Identity Activation
r

FIGURE 4 Structure of the Q network.

rameters, resulting in the network structure shown in Figure 4. An initial learning rate of 0.001 and a L_2-regularization penalty of 10^{-5} were used. The network was trained using stochastic gradient descent with a batch size of 1000. In total, 100,000 training episodes were performed.

Like many other function approximations, the Q-network's predictions for the value of different actions becomes unreliable when extrapolated outside the range of data seen during earlier training episodes, leading the network to sometimes be overoptimistic about the performance of nonsensical PID gains (e.g. very large magnitudes, negative values). Therefore, when the model is used to select PID gains, the optimization is constrained to a reasonable range of parameter values (based on the PID gains observed from the data collected to train the "optimize then train" model).

4 | RESULTS

4.1 | Performance

Figure 5 shows the ISE observed on the test data set using different tuning methods. For the SIMC tuning results shown, the closed-loop time constant (a tuneable parameter) was optimized to minimize ISE. The SIMC tuning rules are not designed to optimize ISE but are presented as a familiar baseline for comparison.

Of the four methods tested, stochastic programming produced the best PID tunings in terms of minimizing ISE, however the "train then optimize" tuning approach performs nearly as well (6% and 10% higher mean and median ISE, respectively). The "optimize then train" method also performs quite well with a median ISE only 2% higher than the stochastic programming ISE, however this method performs less consistently across the test data set resulting in a mean ISE 31% higher than stochastic programming's mean ISE. This is likely a result of the limited number of training examples used in the "optimize then train" method. Figure 6 shows the "optimize then train" neural network's test loss as a function of training examples. We can see the loss is still decreasing considerably by the time we reach all 8,500 training examples, suggesting more training data would continue to improve the model's performance and the high mean ISE produced by the "optimize then train" method is likely due to certain test systems lying outside the

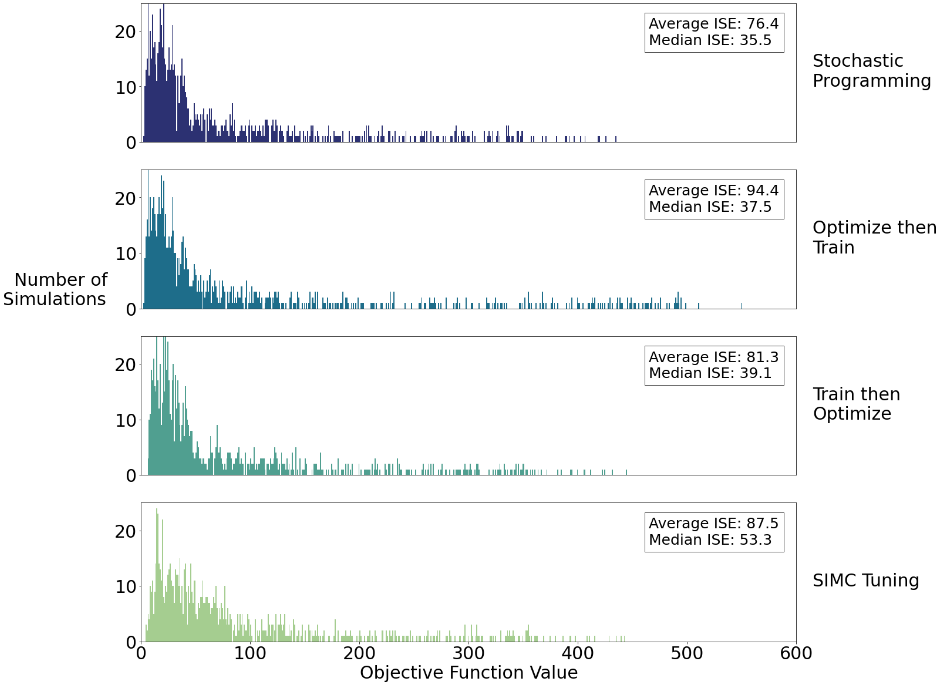

FIGURE 5 ISE produced by different PID tuning methods across the test dataset (n=1000).

parts of the state-space seen during training. Both neural network tuning methods outperform SIMC optimal tunings in terms of the objective function.

Figure 7 shows the various controller tuning methods applied to controlling a flow rate through a pipe when the system undergoes a unit setpoint change under different levels of model parameter uncertainty. In this example, the parameter uncertainties (σ_K, σ_τ, σ_θ) are all set to the same value. We see the tuning methods are responsive to the uncertainty in the model used to tune them. The controller gains are increased slightly as the model becomes more certain. Note the tuning methods were optimized for the *average* ISE, meaning the “optimal” tunings may result in significant oscillations depending on the actual system parameters. For a real application, we may be more interested in guaranteeing a good performance for any possible realization of a system rather than optimizing the average performance. This can be accomplished one of two ways: modifying the objective function to penalize simulations where there is excessive oscillation or optimizing for the conditional value at risk (cVaR) as [23] do in their paper using stochastic programming.

From Figures 5 and 7, we can see it is possible to tune fixed structure controllers explicitly using neural networks to achieve performance similar to the tunings produced by stochastic programming. The neural network approximations tend to be slightly different: both the “optimize then train” and “train then optimize” methods tend to use less derivative action and the “train then optimize” method tends to use more integral action.

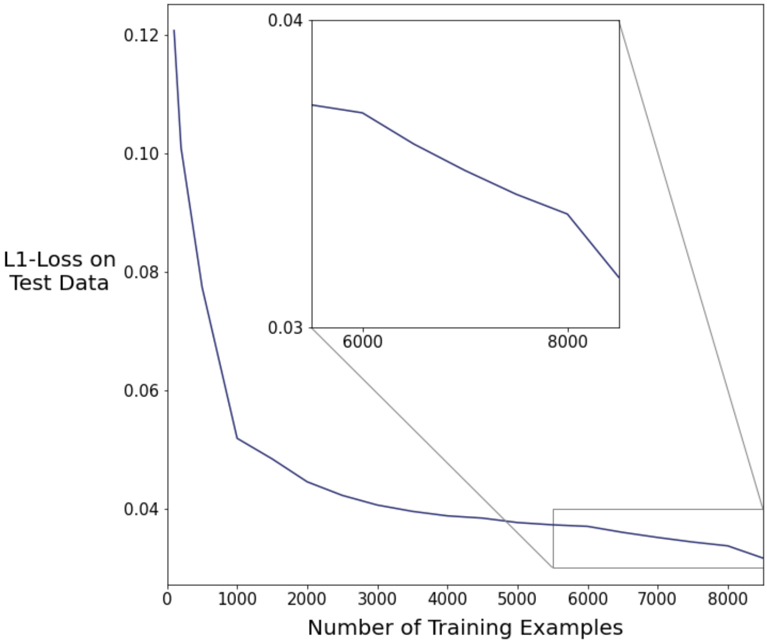

FIGURE 6 L1-loss on test data vs. the number of training samples used for the “optimize then train” method.

4.2 | Computational Efficiency

In Section 4.1, we saw neural network approximations could be trained to perform similarly to stochastic programming. In this section, we highlight the computational advantage of using neural networks.

Table 2 shows the computation time associated with each controller tuning method. Stochastic programming requires no offline pre-training, however calculating controller parameters online is slow, requiring around 90 seconds per parameter update. The “optimize then train” method is very fast online, requiring less than a millisecond for forward propagation through the neural network model. However, collecting the data required to train the “optimize then train” model is very slow. 9,500 stochastic programming examples were generated to train this model, requiring 9 days to train using a single central processing unit (CPU). As discussed in Section 4.1, this approach would benefit from even more training data as well, requiring even more training time. Last is the “train then optimize” method which is significantly faster to train since no stochastic programming problems need to be solved. This model required around 8 hours to train. The “train then optimize” approach is slightly slower online as a small optimization problem needs to be solved to find the optimal controller parameters, but still only requires a few milliseconds to compute.

All three methods can benefit from parallelization. The training time for both neural networks reduces linearly with the addition of more CPUs. The online computation time of stochastic programming can similarly be reduced linearly with more CPUs [23].

4.3 | Importance of Optimized Data

Within the “train then optimize” method, we explored the importance of the RL-inspired training procedure where data is collected through the model’s interactions with its environment. We trained an additional model where actions

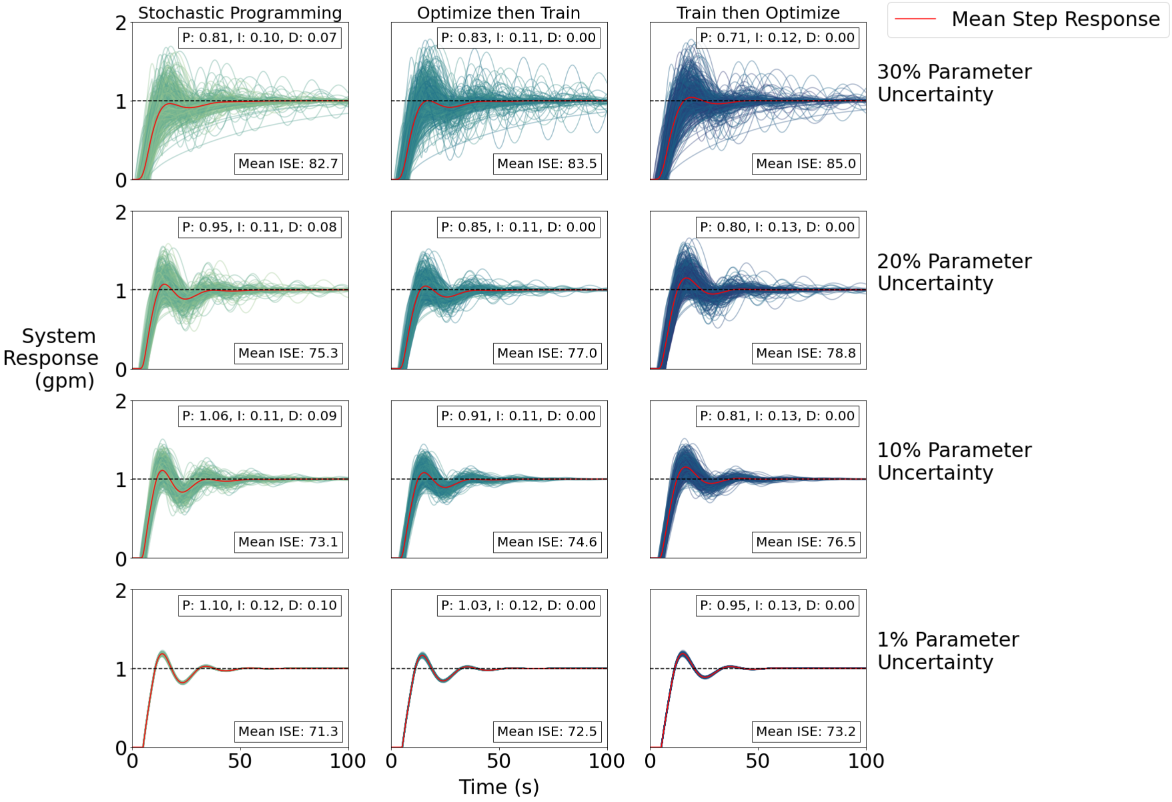

FIGURE 7 Step responses produced by the various tuning methods across 500 simulations of a PID controller used to control the flow rate through a pipe. $\hat{K} = 1$, $\hat{\tau} = 5$, $\hat{\theta} = 5$, $I = 0$. The red line shows the mean step response (i.e. the response which being optimized for).

TABLE 2 Time required to train the neural networks (offline computation time) and time required to select controller parameters online using an Intel® Core™ i7-10750 processor.

Method	Offline Computation Time (s)	Online Computation Time (s)
Stochastic Programming	–	90
Optimize then Train	8 x 10^5	6 x 10^{-5}
Train then Optimize	3 x 10^4	4 x 10^{-3}

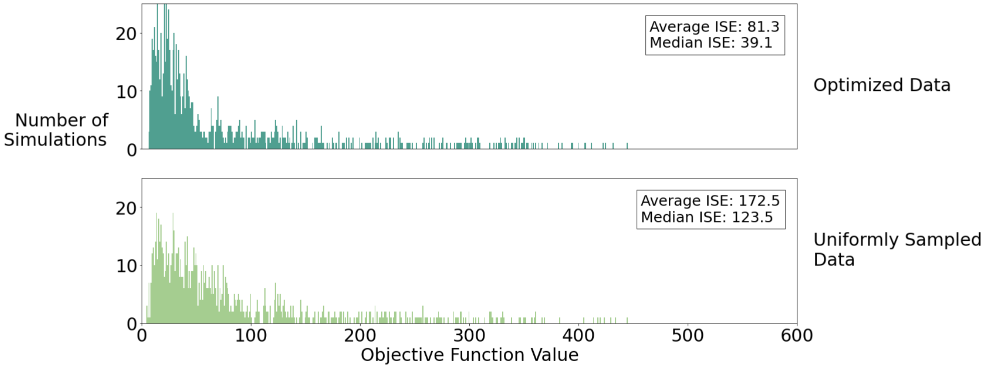

FIGURE 8 ISE produced by different "train then optimize" neural network models across the test dataset (n=1000).

were sampled uniformly from the action-space, $\mathcal{A}$. Both models were trained with the same number of training examples (100,000) and the same states sampled from $\mathcal{S}$. The performance of a model trained with optimized data vs. uniformly sampled data is shown in Figure 8. Using optimized data markedly improves the model's performance.

To further investigate the differences between the two models, we visualize their predicted objective function values across a few systems in Figure 9 and compare their predicted objective function values to the true cost function.

The state-action value function produced using uniformly sampled data appears more accurate overall. However, the approximate state-action value function produced using optimized data more accurately predicts the location of the function's minimum value. Most of the data used to train the optimized model will tend to be around the function's minimum value, explaining why it more accurately models the Q-function in this region of the state-action space while being less accurate further away from this region.

It is also worth highlighting the relative sample efficiency of this method. RL is generally thought of as sample inefficient [12], however in this simple formulation where the state-action value function is not recursive, a near-optimal policy can be learned much more efficiently. After 100,000 interactions with the environment, the model can reliably select a near-optimal action to take. Consider that with the 10-dimensional state-action space, a factorial experiment design would only be able to test 3 different variable levels along each dimension of the state-action space within the 100,000 samples used to train the network.

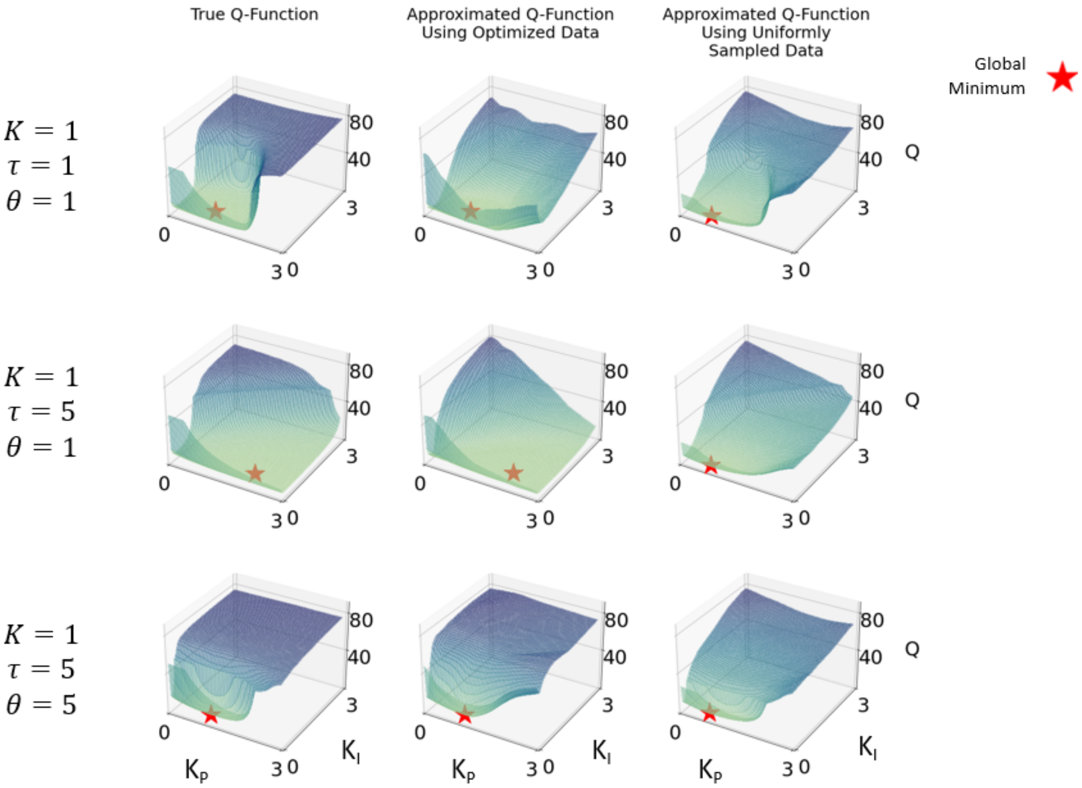

FIGURE 9 Visualization of the state-action value function (Q) produced by the neural networks compared to the true Q function. In each example, there is zero parameter uncertainty and $K_D = 0$. All systems shown are self-regulating ($I = 0$).

5 | CONCLUSIONS

In this paper, we explore different methods for near-optimally tuning fixed-structure controllers. We are interested in a universal controller tuning method which is compatible with any objective function, process model, and controller structure. Moreover, we desire a tuning method capable of accounting for model uncertainties when selecting controller parameters to reduce the need for costly experiments to precisely estimate model parameters, helping bring plants one step closer towards maintenance-free control. Lastly, we desire a tuning method which is computationally efficient and can be used in real-time across many different controllers in a plant.

We explored the use of neural networks to approximate the stochastic programming solution to controller tuning problems, however collecting data to train such a model is very computationally intensive even for linear systems. We also developed a novel controller tuning method inspired by deep Q-learning. We found this method could achieve comparable performance to stochastic programming and be trained much more efficiently, however it introduces an online optimization step, requiring slightly more online computation than the stochastic programming approximation method.

Acknowledgements

We gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) and Honeywell Connected Plant for their financial support of this project.

References

[1] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework, 2019.

[2] Yaoyao Bao, Yuanming Zhu, and Feng Qian. A Deep Reinforcement Learning Approach to Improve the Learning Performance in Process Control. *Industrial & Engineering Chemistry Research*, page acs.iecr.0c05678, 2021.

[3] John R. Birge and Franois Louveaux. *Introduction to Stochastic Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[4] K. Chaloner and I. Verdinelli. Bayesian experimental design: A review. *Statistical Science*, 10(3):273–304, 1995.

[5] Andreas Doerr, Duy Nguyen-Tuong, Alonso Marco, Stefan Schaal, and Sebastian Trimpe. Model-based policy search for automatic tuning of multivariate PID controllers. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5295–5301, 2017.

[6] Ben Eysenbach, Aviral Kumar, and Abhishek Gupta. *Reinforcement Learning is Supervised Learning on Optimized Data*. Berkeley Artificial Intelligence Research Blog, September 2020.

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[8] Martin Hast, Karl Johan Åström, Bo Bernhardsson, and Stephen P. Boyd. PID design by convex-concave optimization. In *Proceedings 2013 European Control Conference (ECC)*, pages 4460–4465. IEEE - Institute of Electrical and Electronics Engineers Inc., 2013.

[9] Geoffrey E. Hinton. Learning multiple layers of representation. *Trends in Cognitive Sciences*, 11(10):428–434, 2007.

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[11] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, 2009.

[12] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for Atari. *CoRR*, abs/1903.00374, 2019.

[13] Jong Woo Kim, Byung Jun Park, Haeun Yoo, Tae Hoon Oh, Jay H. Lee, and Jong Min Lee. A model-based deep reinforcement learning method applied to finite-horizon optimal control of nonlinear control-affine system. *Journal of Process Control*, 87:166–178, 2020.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[15] Nathan P. Lawrence, Gregory E. Stewart, Philip D. Loewen, Michael G. Forbes, Johan U. Backstrom, and R. Bhushan Gopaluni. Optimal PID and Antiwindup Control Design as a Reinforcement Learning Problem. *IFAC-PapersOnLine*, 53(2):236–241, 2020.

[16] Sergio Lucia and Benjamin Karg. A deep learning-based approach to robust nonlinear model predictive control. *IFAC–PapersOnLine*, 51(20):511–516, 2018.

[17] Daniel G. McClement, Nathan P. Lawrence, Philip D. Loewen, Michael G. Forbes, Johan U. Backström, and R. Bhushan Gopaluni. A meta-reinforcement learning approach to process control. *11th Symposium on Advanced Control of Chemical Processes*, 2021.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[19] Rui Nian, Jinfeng Liu, and Biao Huang. A review On reinforcement learning: Introduction and applications in industrial process control. *Computers & Chemical Engineering*, 139:106886, 2020.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[21] P. Petsagkourakis, I.O. Sandoval, E. Bradford, D. Zhang, and E.A. del Rio-Chanona. Reinforcement learning for batch bioprocess optimization. *Computers & Chemical Engineering*, 133:106649, 2020.

[22] Steven Spielberg Pon Kumar, Aditya Tulsyan, Bhushan Gopaluni, and Philip Loewen. A deep learning architecture for predictive control. *IFAC–PapersOnLine*, 51(18):512–517, 2018. 10th IFAC Symposium on Advanced Control of Chemical Processes ADCHEM 2018.

[23] J.A. Renteria, Y. Cao, A.W. Dowling, and V.M. Zavala. Optimal PID controller tuning using stochasticprogramming techniques. *American Institute of Chemical Engineers*, 64(8):2297–3010, 2017.

[24] Joohyun Shin, Thomas A. Badgwell, Kuang-Hung Liu, and Jay H. Lee. Reinforcement Learning – Overview of recent progress and implications for process control. *Computers & Chemical Engineering*, 127:282–294, 2019.

[25] Sigurd Skogestad. Probably the best simple PID tuning rules in the world. *Journal of Process Control*, 2001.

[26] Steven Spielberg, Aditya Tulsyan, Nathan P Lawrence, Philip D Loewen, and R Bhushan Gopaluni. Toward self-driving processes: A deep reinforcement learning approach to control. *AIChE Journal*, 2019.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] A. Wachter and L.T. Biegler. On the implementation of an interior-pointfilter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–27, 2006.

[29] J.P. Watson, D.L. Woodruff, and W.E. Hart. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation*, 4(2):109–142, 2012.

[30] K.J. Åström and T. Hägglund. The future of PID control. *Control Engineering Practice*, 9(11):1163–1175, 2001.