Predicting Machine Failures with Reinforcement Learning

Dalton Schutte

Georgia Institute of Technology

Abstract

We investigate methods from statistical process control, deep learning, and reinforcement learning for predicting when two machines will fail. Each machine is fitted with sensors that report data at semi-regular intervals for each time step. Our results suggest that ...

1 Introduction

As time progresses, machines gradually wear down and will fail. When a machine fails, it is likely to require maintenance or repair before it can return to its normal function. Depending on the application, failure may not be an issue if the process is low-stakes or has redundancies or backups or catastrophic if the process is of critical importance. In most cases, machine failure may result in lost revenue as production is halted or goods are damaged.

Knowing when a machine is likely to fail can reduce the impact on down- and upstream processes by allowing necessary preparations to be taken before the machine is stopped for maintenance. This is preferable to the alternative where the failure remains unaddressed until it is noticed. This can add to down times and makes preparing for the downtime rushed.

1.1 Background

1.1.1 Statistical Process Control

Statistical Process Control (SPC), or, Statistical Quality Control, is a set or techniques that allow one to track the results of a process over time [1]. The general procedure is:

1. Determine which time period will be used for establishing in-control markers
2. Gather observations from this period
3. Calculate the center line
4. Calculate any control limits (such as 2 standard deviations above/below the center line)
5. Plot new observations against these limits
6. Evaluate any patterns that emerge

There are different types of charts depending on what observations are available, the nature of the process, etc. Typically, a single characteristic is measured (e.g. weight for canned goods, length of extruded metal, etc.) per chart for simplicity. Tracking multiple variables is complex and requires special methodologies.

1.1.2 Deep Learning

Deep learning is a procedure where deep neural networks are trained on a dataset using an optimization algorithm in an iterative manner. There are many flavors of neural network, but two that are particularly well suited to sequential data are the Long-Short Term Memory (LSTM) network [2] and the transformer [3].

Transformer-based architectures have been at the forefront of NLP research in recent years. While they have demonstrated excellent performance on some generative language tasks, their ability to process long-term dependencies in sequential data and the ability to pre-train models once and fine-tune for multiple other tasks suggests they could be well suited to applications in time series analysis. Indeed there has been some research on this front. Recently Goswami et al. [4] collected a large amount of time series data to pre-train a family of models called MOMENT. This family of foundational time series models is pre-trained on time series from a variety of domains, with varying time horizons, and for various tasks.

1.1.3 Reinforcement Learning

Reinforcement learning (RL) is a special subset of ML/AI focused on choosing actions in some environment [5]. This process begins with some "agent" that will use some reinforcement learning algorithm to interact with an environment which, in turn, provides a reward signal that is used to help train the agent. The ultimate goal is for the agent to learn an optimal policy, π, that can be used to choose actions given a state. Perhaps one of the most famous applications of reinforcement learning is in games. Many board games, such as Go [6], have a huge number of discrete game states that one cannot simply find the best play with brute forced.

Two of the, perhaps, most well-known techniques are Proximal Policy Optimization (PPO) [7] and Neural Guided Monte Carlo Tree Search (MCTS) [8]. The former was used to train the OpenAI 5, a collection of AI agents trained to play the video game Dota 2 [9] and the latter to train AlphaGo, which beat the then Go world champion in a set of 4-1.

1.2 Research Questions

The states for many machine sensors are not discrete, but rather, take on values in $\mathbb{R}$. The T^2 and, most, deep learning models handle this situation without issue. There are plenty of RL algorithms that can solve continuous control tasks [10, 11]. However, we wish to examine the efficacy of RL for stopping machines that are about to fail. We will seek to answer the following questions:

1. How well can MCTS and PPO perform at this task?
2. How do these algorithms compare to traditional methods from SPC?
3. How do these algorithms compare to models, such as MOMENT, designed for sequential data?
4. How do the RL methods compare to each other?

2 Methods

2.1 Data

We are using data from two machines, a blood refrigerator and a nitrogen generator [12].

A blood refrigerator is a machine designed to store blood safely at proper temperatures. The general composition of a unit is the compressor to pump coolant throughout, the condensor to remove heat from the collant and condense it, the evaporator which evaporates the coolant to cool the interior of the unit, an expansion valve to regulate the flow of coolant, and tubing to move the coolant throughout.

A nitrogen generator is a machine that separates nitrogen from other gases in compressed air. This unit typically includes an air compressor, carbon sieves for filtering, absorption vessels to collect nitrogen, and towers to increase production.

2.1.1 Analysis

The data includes a mix of binary and continuous variables with a binary target variable. The class imbalance is heavily in favor of the 'normal' class (PW_0.5h=0), which indicates the machines are operating within expecations the majority of the time, a desireable phenomenon. The ratio of normal to failed states is roughly 99:1 for the blood refrigerator and 49:1 for the nitrogen generator. In each dataset, there are several days where there is no failure state, meaning the machine operated without issue the entire day.

The measurement intervals are spaced roughly 1 minute apart for the nitrogen generator and about 34 seconds for the blood refrigerator. The label 'PW_0.5', using the notation and convention from the paper that provided this dataset [12], represents the Prediction Window (PW) of 30 minutes (0.5 hours). This means that in the 30 minute interval beginning at a time stamp t, the machine is operating outside of parameters for at least one of those time stamps. The other convention from that paper is the Reading Window over some interval. The Reading Window (RW) is the collection of sequential time stamps used as input to a model. In our experiments, for example, we use a RW of 20 minutes where we collect the minimum number of sequence of time steps necessary to span 20 minutes of time. In Pinciroli et al. [12], they explore a wide combination of PW and RW but found that most of their models performed best with a 20 minute RW, which is why that is the window size we chose to use.

Figures 1 and 2 show some examples of the time series data in our dataset. Initially, the blood refrigerator dataset had 16 sensors providing output and the nitrogen generator had 7. We removed all the output from any sensor whose standard deviation was zero over the entire dataset. This left 12 variables for the blood refrigerator and 4 for the nitrogen generator.

The time series for the features in the blood refrigerator (figure 1) highlight that there is, intuitively, some change occurring in the machine that is leading to these sensor readings. Particularly, the product temperature base, evaporator temperature, and power supply have noticeably different patterns in the red failure region than in the period before.

The nitrogen generator (figure 2) shows an instance where the failure happens early in the observation period, only a few time steps into the beginning of the period. This poses a challenge for the methods as there is very little data for them to work from before having to determine if the machine is on a trajectory to failing. However, this may be the case in real operating conditions and so is an important case to keep in our

	Blood Refrigerator		Nitrogen Generator	
	Train	Test	Train	Test
Days	25	27	29	8
Timesteps	60166	65763	40354	11162
Stops (1)	704	642	810	242

Table 1: Summary statistics of our datasets

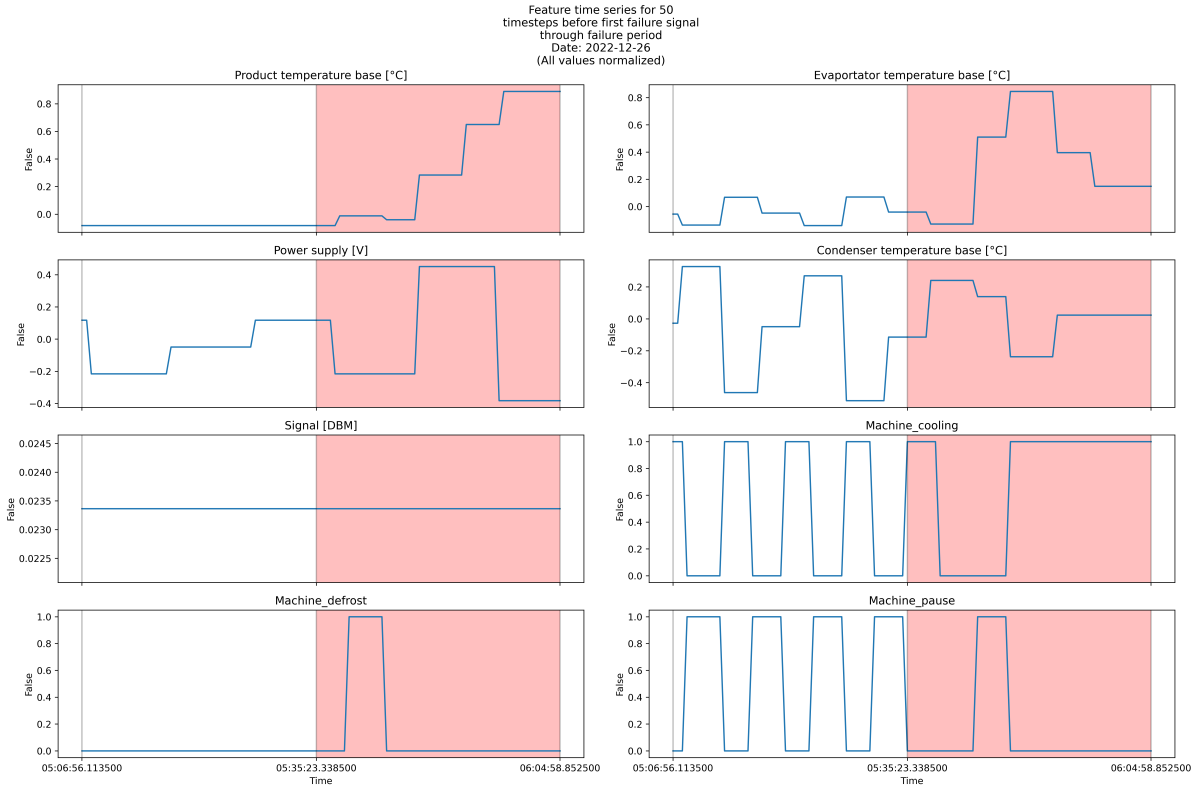

Figure 1: Plots of variables around the point of failure for the blood refrigerator. Red regions are where regions where the data label indicates failure (1)

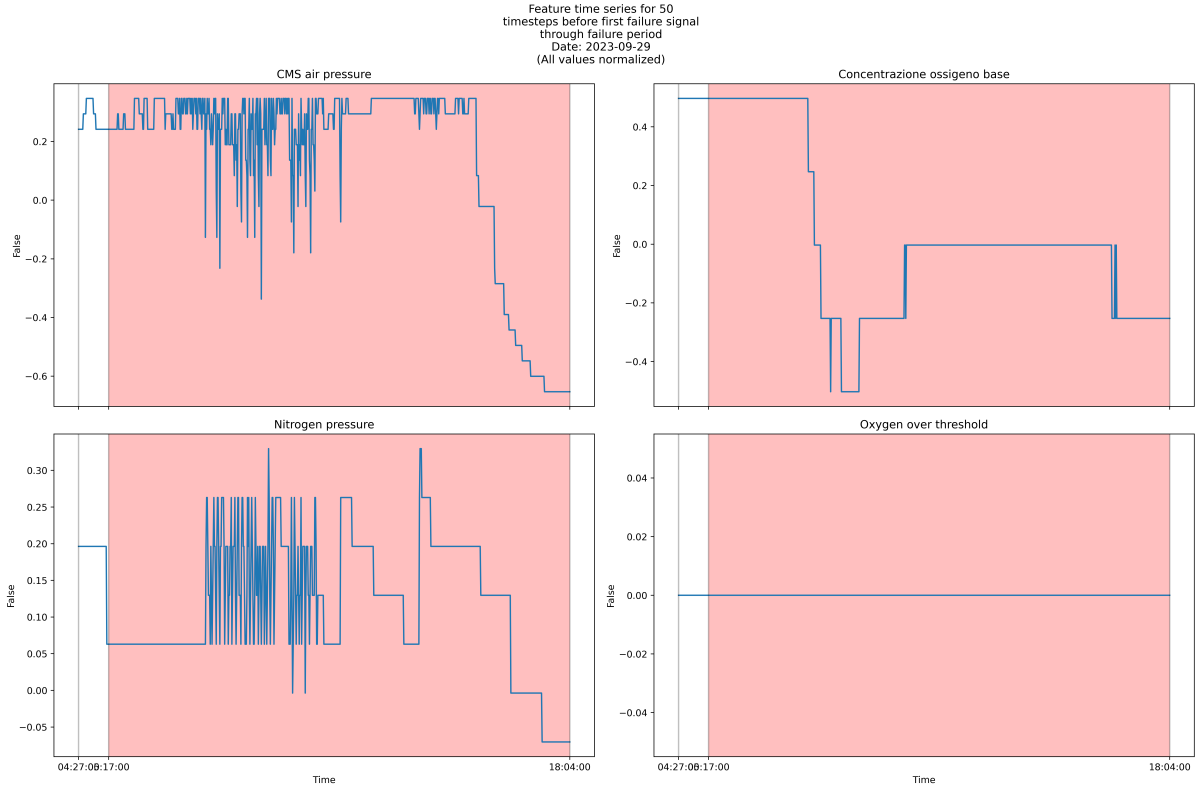

Figure 2: Plots of variables around the point of failure for the nitrogen generator. Red regions are where regions where the data label indicates failure (1)

dataset.

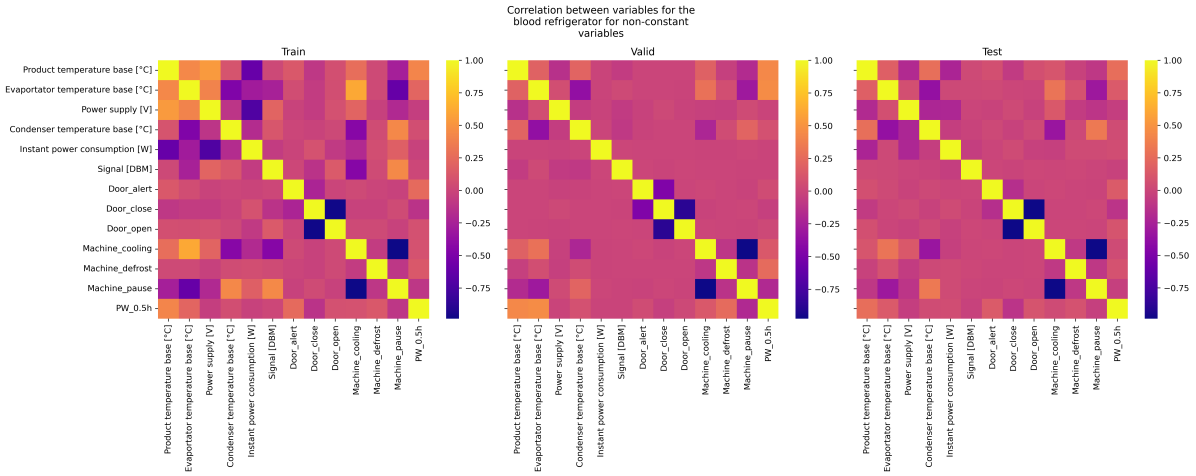

Figure 3: Correlation between variables in the blood refrigerator dataset splits. The label column is 'PW_0.5h'.

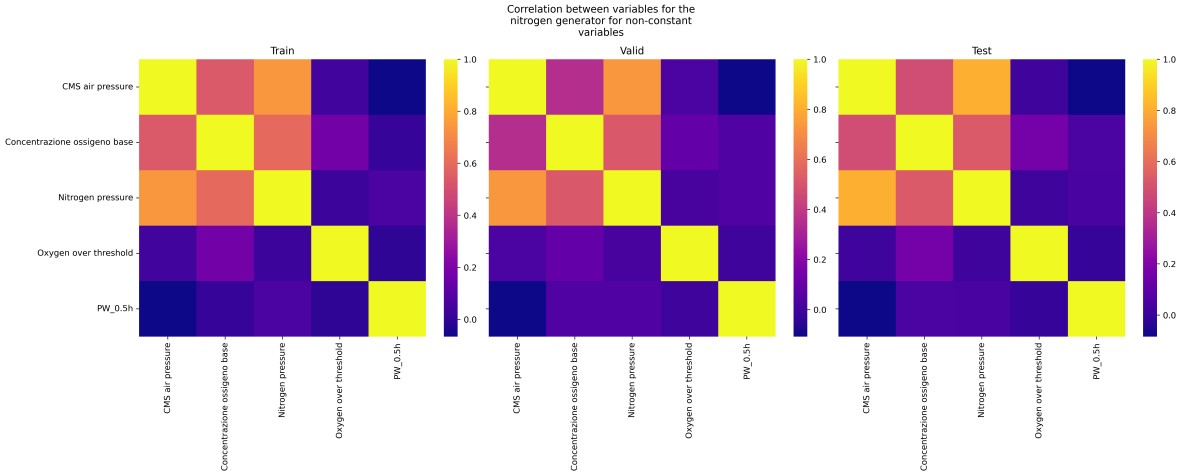

Figure 4: Correlation between variables in the nitrogen generator dataset splits. The label column is 'PW_0.5h'.

Correlations Upon examination of the correlations, shown in figures 3 and 4, between variables for each dataset, there were no instances where any one variable was significantly correlated with the target variable, 'PW_0.5h'. We felt no need to drop additional features based on these observations.

Feature Distributions Figures 5 and 6 show the distributions of the continuous variables for both machines. The blood refrigerator has, good representation of most variables in the training set that appear in the validation and test splits.

For additional detailed discussion and analysis of the datasets, the reader is referred to [12].

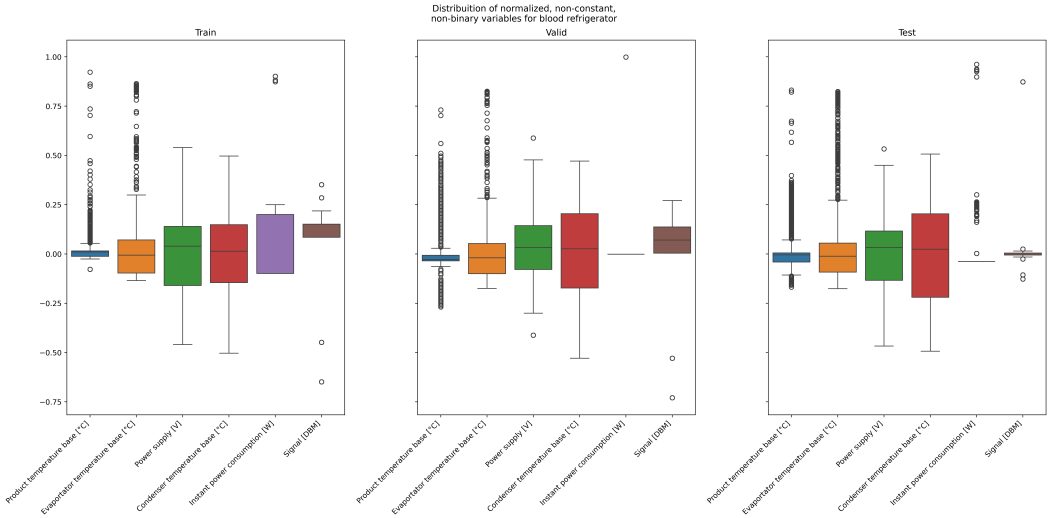

Figure 5: Distribution of normalized, continuous variables for the blood refrigerator dataset splits.

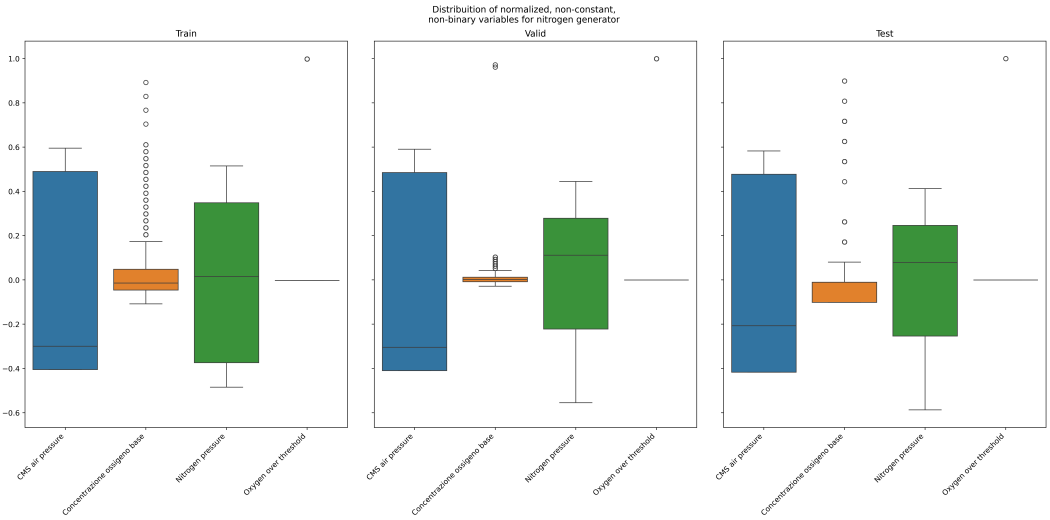

Figure 6: Distribution of normalized, continuous variables for the nitrogen generator dataset splits.

2.1.2 Processing

Pre-processing of the data was kept fairly minimal. As mentioned above, we decided to drop any columns with zero variance as these would not contribute any meaningful signal for any of our methods to learn from. This resulted in 12 columns for the blood refrigerator data and 4 columns for the nitrogen generator data. The remaining continuous variables were normalized with:

$$\hat{x_{it}} = \frac{x_{it} - \bar{x_i}}{\max x_i - \min x_i} \tag{1}$$

for variable i at time t.

2.2 Models

2.2.1 T^2 Control Chart

One method from SPC that is commonly used is the T^2-Control Chart with Upper Control Limit (UCL). Given a set of n correlated characteristics X_i, assumed to follow a multivariate-normal distribution, the T^2 statistic is:

$$T^2 = m(\bar{x} - \mu)\Sigma^{-1}(\bar{x} - \mu) \tag{2}$$

Where m is the number of samples in the subgroup ($m = 1$ is referred to as individual control charts), $\bar{x}$ is the mean of a sample of m vectors of measurements, Σ is the covariance matrix (assuming it is known), and μ is the in-control process mean (assuming it is known) [13]. For simplicity, it should be assumed that all discussion is with regards to individual T^2-control charts unless otherwise stated.

The UCL is χ^2 distributed and is calculated at $\frac{\alpha}{2}$. In practice,

$$\text{UCL}_{\frac{\alpha}{2}} = \frac{(k-1)^2}{k} f\left(1 - \frac{\alpha}{2}; \frac{p}{2}, \frac{(k-p-1)}{2}\right) \tag{3}$$

where f is the beta distribution. The Lower Control Limit (LCL) is set to zero.

It is often the case that Σ and μ are not known. These can be estimated from the sample by:

$$\mu \approx \hat{\mu} = \frac{1}{k}\sum_{i=1}^{k} \hat{x}_i \tag{4}$$

where $\hat{x}_i$ are the observations from the in-control process sample, and

$$\Sigma \approx \hat{\Sigma} = \begin{bmatrix} \hat{\sigma}_{11}^2 & \dots & \hat{\sigma}_{1n}^2 \\ \vdots & \ddots & \vdots \\ \hat{\sigma}_{n1}^2 & \dots & \hat{\sigma}_{nn}^2 \end{bmatrix} \tag{5}$$

where $\hat{\sigma}_{ij}^2$ is the variance between characteristics X_i and X_j from the in-control process sample.

To plot the i^{th} point on the individual chart, (1) becomes:

$$T_i^2 = (x_i - \hat{\mu})\hat{\Sigma}^{-1}(x_i - \hat{\mu}) \tag{6}$$

As new observations are made and new points added to the plot various signals may arise. Perhaps the most apparent is a breach of the UCL (a new point with a T^2-statistic $>$ UCL). This could suggest that something unusual happened with that

single instance. There are many other patterns that may arise. A continuous streak of 3-5 points beyond the 2 standard deviation mark, 7 or more beyond 1 standard deviation, 8-9 points alternating sides of the center line in a zig-zag pattern. All of these suggest different things about the underlying process and that either the process should be investigated or a new in-control sample taken to establish new charts.

We will be using the above technique largely as-is because it is a standard methodology and has demonstrated itself to be very effective. We will use the last 25%, 50%, 75%, and all of the training data to establish the control parameters. This will allow us to evaluate the effects of recency on the results. To determine when a "stop" signal should be raise, we have defined two patterns that accept a variety of parameters. When the conditions specified by the patterns are met, a 1 ("stop") is returned. Because this method is not trained in an iterative manner, we do not train or evaluate using the validation time series.

During the experiments, we found that the performance when using the training data to construct control charts for the test data yielded very poor results. So we included a set of results where the control limits are set using the first 150 or 70 time steps for the blood refrigerator and nitrogen generator, respectively, for each day in the test set. In other words, the control chart for a given day would be determined based on the the sensor readings during the first part of its normal operation for a day.

The patterns we use are "n sequential breaches of the UCL or LCL" and "n total breaches of the UCL or LCL in a window of length t". For the former case, if we raise a stop signal at the first instance of any window where all n observations are above or below (including a mix of above and below) the control limits. For the latter, we raise a stop signal at the first instance of a window of t time steps there are n breaches above or below the control limits.

2.2.2 Transformer

The transformer is an architecture with several components: a tokenizer, an embedding layer, transformer layers, a task-specific head [3]. The transformer layer is a collection of alternating multi-head attention layers and linear feed forward layers. These models are often pre-trained on large collections of text as foundation models then fine-tuned for specific tasks by adding a task-specific head that converts the representation from the last hidden layer of the transformer body into a useable output. It is possible to train transformers on more than just text. Many variations exist that operate on video, audio, time series, and combinations of the above (multi-modal).

A family of open-source foundational time series transformer models was recently released [4] that has been pre-trained on a rich collection of time series data and can be fine-tuned for specific tasks such as anomaly detection or classification. The models (MOMENT) were pre-trained with univariate time series data with a variety of time horizons. When given multivariate time series data, each variable is treated separately in its own channel.

The specific model we will be using has an architecture similar to the T5 encoder [14] and has approximately 385MM parameters. We will be fine-tuning a freshly initialized linear layer to perform the classification task.

For this model, we will extract rolling windows comprising 20 minutes (determined using the time stamps provided in the data) as the input sequence and the label for the time step immediately after the end of the window as the target for learning. MOMENT requires all input sequences are 512 tokens in length, so all inputs will be padded with zeros to meet the length requirement and masks will be generated so the loss is calculated only from features related to the input sequence. We chose 20 minutes since previous research found that was the window where the models typically performed best [12].

MOMENT will be fine tuned using the Adam optimizer [15] with a learning rate of 0.001 that uses a cosine annealing schedule to a minimum of 10^{-6} for binary classifi-

cation. Weight decay is 0.1 and β_1, β_2 are 0.9 and 0.999, respectively. We use a batch size of 8. Fine-tuning is carried out over 5 epochs and the model is checkpointed at the end of each epoch if the validation loss decreased. Due to the high degree of class imbalance, we use normalized class weights in the loss function equal to:

$$[w_0, w_1] = [1/p_0, 1/p_1] * (1/p_0 + 1/p_1)^{-1} \tag{7}$$

2.2.3 Reinforcement Learning

Reinforcement learning is defined in the context of solving a Markov Decision Process (MDP). An MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P})$ with a state space, action space, reward function, and a probability to transition from state to state given an action. A state in the state space $s_i \in \mathcal{S}$ is a representation of the environment at time i. In our case, each s is a vector in $\mathbb{R}^n$. The action space is comprised only of "Do nothing", "Stop", or, $0, 1 = \mathcal{A}$. Our reward function is defined so that the agent receives a positive reward if it returns 0 while the label is still 0 or 1 when the label is 1. The agent will receive a negative reward if it returns a 1 when the label is 0 and vice-versa. There is an additional penalty that is incurred for each time step that passes where the agent has not stopped the machine (returned a 0) but the label is 1, specifically:

$$\mathcal{R}(\int, \dashv) = \begin{cases} a = 1 \wedge \text{PW\_0.5h} = 1 & \rightarrow +1 \\ a = 1 \wedge \text{PW\_0.5h} = 0 & \rightarrow -10 \\ a = 0 \wedge \text{PW\_0.5h} = 1 & \rightarrow \sum_{i=0}^{k} i \text{ where } k \text{ is the time since the first stop signal} \\ a = 0 \wedge \text{PW\_0.5h} = 0 & \rightarrow +0.01 \end{cases}$$

where PW_0.5h= $0, 1$ indicates the label and a the action chosen by the agent at that step. The transition probabilities are defined by the temporal sequencing naturally present in the dataset and is deterministic in nature.

However, we observed extremely brittle policies by leaving the time series for each day in temporal order. So, we shuffle all of the days during each episode to prevent the agent from trying to learn dependencies day-to-day. While there is an argument to be made for the importance of day-to-day patterns, that is a problem that warrants its own paper. For simplicity, we will proceed only trying to have the agents learn how to act within each day.

Monte Carlo Tree Search The Monte Carlo Tree Search (MCTS) with neural guidance stores information in a tree structure with states as nodes and actions connecting from that node to the next state that it transitioned to as a result of that action. It gathers a number of paths by running simulations down the tree by selecting the action at each node that leads to the node with the largest expected value, starting at the root, and propagating values back up through the path based on a combination of the expected value for that state and the number of visits to that node over the simulations. Some randomness is used to ensure some exploration can occur to increase the likelihood that the algorithm will find the optimal policy.

The MCTS algorithm has 4 steps:

1. Starting from the root, select child nodes until a leaf node is reached
2. If the leaf node is not a state that is equal to the process being stopped, create child nodes for each possible action
3. Perform a simulation by collecting paths from moving down the tree

4. Pass the result from this simulation (rollout) up through all nodes in their respective paths

Step 3 can be done randomly or with some sort of direction. This can be as simple as a greedy search (select the child node with the largest reward) or can use a more complex method such as a neural network [6,8]. The neural network takes the position in the tree and returns a vector of probabilities to select each action (the policy). These policy vectors are proportional to the exponential of the visit count for each node (i.e. $\pi \propto N(s,a)^{1/\tau}$, where N is the visit count and τ a scaling value). The neural network has two heads, one to return a policy and one to return a value for the node. Training is done by minimizing the cross entropy loss between the policy and the actions yielding the highest reward, plus, the mean squared error of the value of the current node and the discounted expectation of future returns. The ideal situation is for the network to learn to correctly choose an action that will keep moving the agent towards high-value states that yield, in aggregate, the largest cumulative reward.

We explored this method due to the unique nature of the policy optimization process and believed that it would provide an interesting and challenging implementation.

The network is trained using the Adam optimizer [15] with an initial learning rate of 0.0001 and uses cosine annealing to a minimum of 10^{-6}. Weight decay is set to 0.01 and β_1, β_2 are 0.9 and 0.999, respectively. The discount factor for future rewards is $\gamma = 0.99$. At the end of each time series for a day, end of an episode, or every 4^{th} time step, a "learning" process is triggered. The process begins by collecting 100 simulations (paths) from the current tree using the current network. The agent samples 128 tuples from the 10,000 most recent experiences (s_t, a_t, r_t, s_{t+1}) to train the network. Each epoch is considered complete when the agent has progressed through all of the days' time series in the training set. We train the agent for 30 epochs and use early stopping if there is no improvement in the mean reward on the validation set after 5 epochs.

Proximal Policy Optimization The proximal policy optimization (PPO) [7] is an extention of their earlier trust-region policy optimization algorithm [16]. The agent uses both a neural network to learn a policy, that is sampled from using a categorical distribution, to determine what action to take in a given state and a separate network to learn the expected value of a given state. We sample from a distribuition determined by the policy to allow some amount of exploration to persist during training. This can help reduce the likelihood an agent gets stuck in a locally optimal policy and unable to find the globally optimal policy. The PPO algorithm contains some simplifications and improves the sample efficiency by using multiple policy updates per episode (or epoch). During each episode, the policy and value networks are updated k times, each time using a different minibatch of recent experiences stored in the agent's memory.

We choose to explore this algorithm since it has demonstrated excellent performance in a variety of continuous state spaces.

2.3 Evaluation

The trained model or agent is used to generate predictions on the held-out test set. We collect the recall, precision, and F_1 (in both strict and relaxed settings) as well as the mean-time-from-event (MTFE). The MTFE measures the mean mistiming of the method, mathematically:

$$MTFE = \frac{1}{K} \sum_{k} (t_k m - t_k e) \tag{8}$$

The strict version of the classification metrics is determined where the model or agent returns the stop signal only on the very first occurence of the stop signal in that

time series or, in the case where a time series does not stop, if the agent never raises the stop signal. The relaxed variation considers any stop signal returned by the model at any stop label in the time series as correct.

The precision, recall, and F_1 are calculated in the usual manner. The only difference arises in the use of strict or lenient evaluation protocols.

Each method is trained on the training split then evaluated on the held-out test split. The control chart and transformer return predictions for each time step while the RL agents stop making predictions for a given day until they return a stop action or reached the end of the day's time series. The predictions are calculated on a per-day basis for each test split (e.g. the 27 days for the blood refrigerator test set are each given a single prediction).

3 Results

3.1 T^2 Control Charts

3.2 Transformer

3.3 Reinforcement Learning

4 Discussion

4.1 Findings

4.2 Limitations

4.3 Future Work

5 Conclusion

References

[1] William F. Guthrie. NIST/SEMATECH e-Handbook of Statistical Methods (NIST Handbook 151), 2020.

[2] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs].

[4] Mononito Goswami, Konrad Szafer, Arjun Choudhry, Yifu Cai, Shuo Li, and Artur Dubrawski. MOMENT: A Family of Open Time-series Foundation Models, May 2024.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 1998.

[6] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton,

Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[8] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[9] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub W. Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019.

[10] Vitchyr H. Pong, Shixiang Shane Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep rl for model-based control. *ArXiv*, abs/1802.09081, 2018.

[11] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[12] Nicolò Oreste Pinciroli Vago, Francesca Forbicini, and Piero Fraternali. Predicting Machine Failures from Multivariate Time Series: An Industrial Case Study. *Machines*, 12(6):357, May 2024.

[13] John Lawson. *An introduction to acceptance sampling and SPC with R.* CRC Press, Boca Raton London New York, first edition edition, 2021.

[14] Colin Raffel, Noam M. Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2019.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] John Schulman, Sergey Levine, P. Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. *ArXiv*, abs/1502.05477, 2015.