# Robot Module 2: Horde and Pavlov

## Introduction

This module extends on the previous two modules [1][2] by both increasing the number of predictions made, measuring prediction error, and using those predictions in “pavlovian control”. The main aspect of this module are to 1) implement an architecture for a Horde of demons to make predictions about the datastream, 2) implement pavlovian control to have the robot respond by changing its policy based on a predicted value, and 3) measure demon learning over time.

## Horde Architecture

As outlined in Figure 1, the architecture of my Horde agent seeks to allow the agent to make decisions quickly and demons to update as accurately as possible.

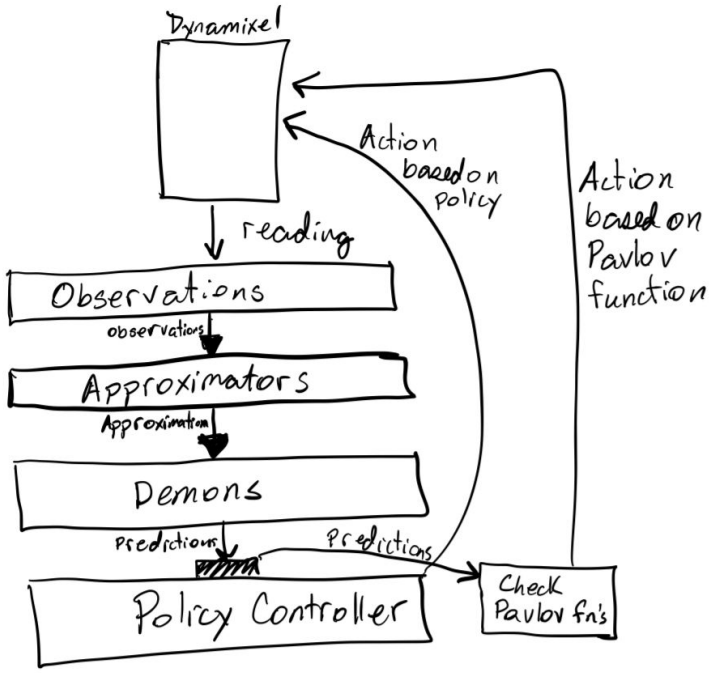


Figure 1: Horde architecture for leaning many predictions online simultaneously. Note that while the Demon’s predictions are given to the policy controller, they are not updated until after the action is chosen.

The agent gets the reading from the Dynamixel (see setup from Modules 1 and 2) [1][2], saves those observations, runs function approximation on the observed data, that function approximation is then passed to the demons. One key thing to note is that the demon’s pass on their prediction based on their current weights to the policy controller so that the policy controller can choose the action prior to the demon’s updating. This is based on the idea from [6] that in asynchronous environments the agent should always act first, rather than waiting for weights to

update. Finally the policy controller first check's a list of pavlovian functions to see if the agent's action should be interrupted by a policy based on the predictions of the agent. If there is no pavlovian response necessary the policy controller chooses the action based on the active behavior policy.

To track the progress of the agent a dashboard comprising a subset of learning measures from the agent was constructed as seen in Figure 2.

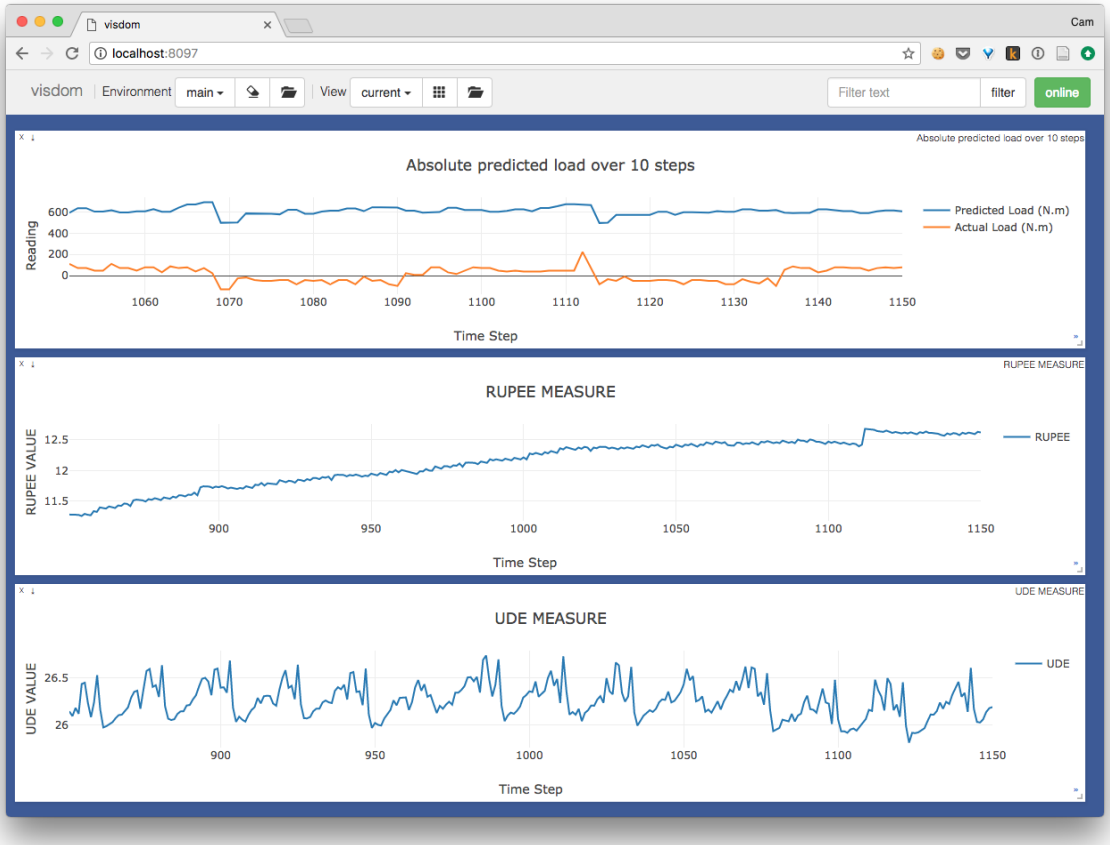


Figure 2: Dashboard of agent with predicted load, RUPEE, and UDE.

**Horde of Demons**

To test the horde architecture 10 demons were constructed to run using the Dynamixel. The on policy demons were learned using TD Lambda, the off policy demons were learned using GTD Lambda. The demons were set up as follows:

Predict Zero Angle On Policy

Predicts how many steps until zero following the current policy. Gamma was one on every time step, unless the angle was zero in which case it was zero.. Cumulant was 1 on every time step.

Predict Load Over Ten Steps On Policy

Predicts the amount of load over the next ten steps. Gamma was 0.9 on every time step. Cumulant was the load.

Predict Load Demon Off Policy

Predicts the load on the next step off policy. Policy was to return to angle zero, gamma was 0, cumulant was the load.

Predict Load Over Ten Steps Off Policy

Predicts the load on the next ten steps off policy. Policy was to return to angle zero, gamma was 0.9, cumulant was the load.

Predict Temperature Over Ten Steps Off Policy

Predicts the temperature on the next ten steps off policy. Policy was to return to angle zero, gamma was 0.9, cumulant was the temperature.

Predict Load Over Five Steps Off Policy

Predicts the load on the next five steps off policy. Policy was to return to angle zero, gamma was 0.8, cumulant was the load.

Predict Temp Over 5 Steps Off Policy

Predicts the temperature on the next five steps off policy. Policy was to return to angle zero, gamma was 0.8, cumulant was the temperature.

Predict Steps to Zero Angle Off Policy

Predicts the number of steps to return to zero off policy. Policy was return to angle zero. Gamma was one on every time step, unless the angle was zero in which case it was zero. Cumulant was 1 on every time step.

Predict Return to Angle of Two

Predicts number of steps to return to two off policy. Policy was return to angle two. Gamma was one on every time step, unless the angle was two in which case it was zero. Cumulant was 1 on every time step.

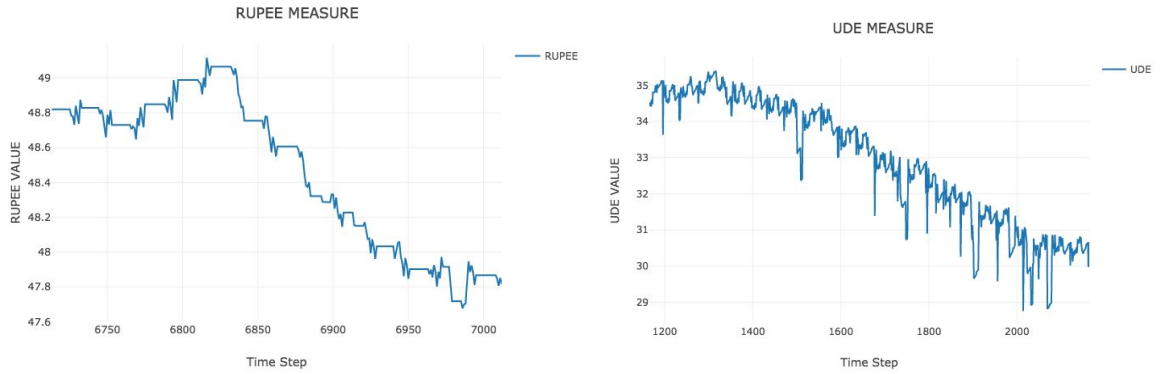


Figure 3: Learning of both RUPEE and UDE.

As seen in figure three the agent started to improve it's RUPEE measure by around time step 6800. UDE improved much sooner, around time step 1200.

**Surprising the agent**

To test the predictor's ability to react to sudden changes in predictions the agent was "surprised" at 7000 time steps by suddenly multiplying it's weights by 10. This caused the spikes in RUPEE and UDE seen in Figure 4. This shows that the agent can detect sudden changes in its predictions with spikes in the error of the demons, and a large change in the UDE. As

mentioned in Adam White's thesis [7] this is something that can be used to control the agent's behavior policy.

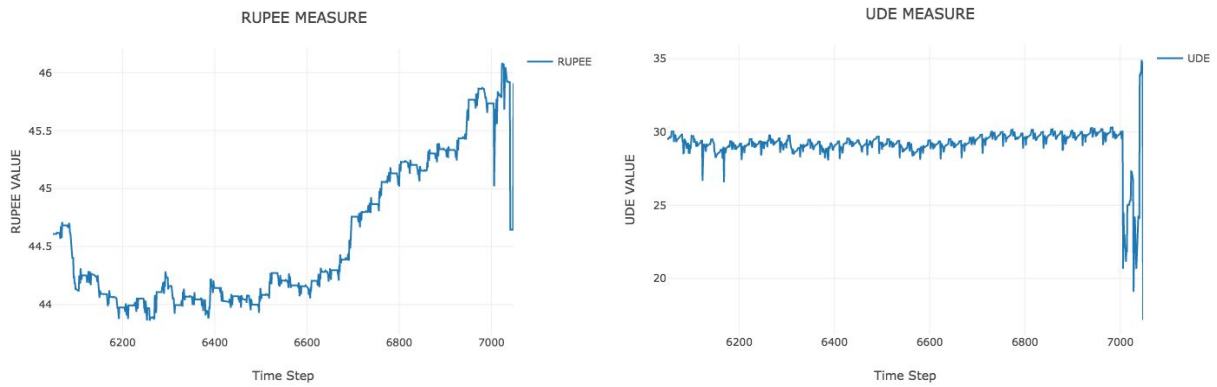


Figure 4: Reaction of RUPEE and UDE to "surprise" change of the weights of all predictors by 10x.

## Pavlovian Control

Pavlovian control [4] was implemented on the robot to give it the ability to change its policy to act differently based on a prediction. This can be seen as a reaction by the robot to a predicted upcoming stimulus. An example from nature would be the reaction of blinking to a prediction of something coming toward your eye. In this case the agent used the prediction of load over 10 steps for the agent to stop what it is doing. This is important as, after playing with the robot for a bit, load over time is the best predictor of the robot overheating and shutting off. The agent was hard coded to stop and disable torque when the agent's prediction of 10 step load was greater than 850. This allowed the agent to have a second to "cool off", potentially release the load held on it, and prevent overheating or damage.

In order to test this an increase in load was put on the agent as it rotated in the negative direction, this increased the load in the robot. Interesting it did not cause a very high spike in load overall, but a persistence to the load. It was this persistence that we wanted to predict, and use to shut of the agent to prevent overheating. In figure _ you can see why this prediction was better than using random spikes of load. Every two hundred steps or so the load would spike on the agent, and if this was used the agent would be randomly shutting down for no reason. Instead I used a prediction of load over time, using the ten step on policy predictor of load outlined above. This allowed the agent to shut down in prediction of increased load over time, and shut down the agent before an overheating or damage could occur. In the operation over 10,000 time steps the agent would occasionally have a higher prediction of load in error, but these were rare, in the range of one every 10,000 time steps, in comparison to the hard coded load spikes which occurred in the range of every 200 time steps.

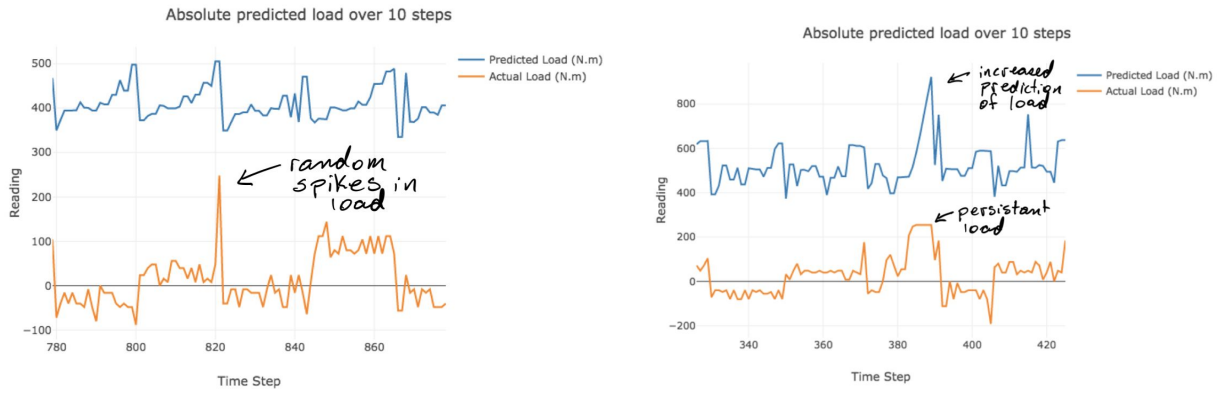


Figure _: Left: the load randomly spiked above 200 making load a difficult reactive predictor for the agent to make a decision to stop motion on. Using this reactive system caused the agent to randomly stop every couple of hundred time steps when it spiked over 200, even though it didn't need to and no additional load was added. Right: Persistent load, below, caused the agent to predict an overall increase in load over 10 steps and stop activity, thus preventing overheating.

**Discussion**

This section showed how a horde of demons could be implemented to predict many things about a robot using multiple policies, gamma functions, and cumulants. Additionally it showed how progress could be measured off-policy to track how well the agent is learning to make predictions. Finally it showed how an on-policy predictor could be used to implement pavlovian control, and use it's prediction to prevent the robot from overheating.

More demons were added without significant increase in time through the observe, approximate, predict, act, update loop. With one demon the loop took on average 0.11269903183 seconds, and with ten demons it took on average 0.120605945587 seconds. This was without many optimizations to improve scalability. Even within this architecture many more demons could be added to make even more predictions.

Finally, the predictions learned here were not as successful as the ones learned during module two. Different sets of parameters and binning sizes were tried with some increase in accuracy, but not to the level achieved in Module 2. Further work to improve predictions in the Horde architecture I implemented needs to be done.

Code for this module can be found here [3]
Video of the robot in operation can be found here [5]

[1] https://github.com/camlinke/607/tree/master/module1
[2] https://github.com/camlinke/607/tree/master/module2
[3] https://github.com/camlinke/607/tree/master/module3

[4] J. Modayil and R.S. Sutton. Prediction driven behavior: Learning predictions that drive Fixed responses. In AAAI Workshop - Technical Report, volume WS-14-01, pages 36{42, Reinforcement Learning and Artificial Intelligence Laboratory, University of Alberta, 2014
[5] https://youtu.be/7zm5de_LNxo
[6] Reactive Reinforcement Learning in Asynchronous Environments: https://arxiv.org/abs/1802.06139
[7] White, A. (2015) Developing a predictive approach to knowledge. Doctoral thesis, University of Alberta. Ch. 4.