Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Intel Industrial AI

24/06/2021

Project Documentation

Team members

Ciaran Dowds – cd2518
Frederick McCallum – fjm518
Ludwig Jonsson – lj1818
Jacob Jenner – jj318
Nikita Swaroop – nss18

Supervisor

Dr. John Wickerson

Intel Team

Ben Jeppesen
Jakub Pludowski
Mark Jervis
Joshua Levine
Ogheneuriri Oderhohwo

Imperial College London

Contents

1 Introduction and Overview

1.1 Project Introduction

Motors in industrial machinery have limitations on input voltage, current, and acceleration. Non-linear controllers can overcome these restrictions by learning more complex functions. This can achieve faster and more energy-efficient control than traditional PI controllers. This is of considerable interest to Industrial customers, as faster position control in motors could lead to higher outputs, and energy savings directly equate to cost savings.

As Intel is interested in exploring the broader use-cases of their FPGA (Field-Programmable-Gate-Array) chips, it makes sense to try and utilise their fundamental advantage in performing parallelised computation to improve upon traditional PI controllers. One way of doing this is to implement a neural network, which is a non-linear function capable of competing with conventional controllers. A neural network can naturally be implemented on an FPGA by connecting logic, (as opposed to being emulated in software), and can also be highly effective when using low precision fixed-point units, which can be manipulated efficiently on FPGAs.

Our task was to try and exploit these advantages to create a neural network-based controller that can be deployed to an FPGA and match or improve upon a conventional PI controller.

1.2 Proposed Solution

We have created several neural-network based motor controllers for the Tandem Motion-Power 48 V kit and generated HDL code for the most promising neural network for implementation on an Intel Max 10® FPGA chip.
The Max 10® FPGA chips were targeted due to their low cost and ability to compete with traditional microcontrollers used in industrial applications.
Our simulations estimate that our best controller improves control signal tracking error rate by 37% compared to traditional PI controllers.
We believe this is an ideal low-cost solution to increase production efficiency for industrial customers.
We propose that further experimentation using reinforcement learning techniques can expand neural-network controller application for different motor kits and multi-axis control. Making these designs available to customers via Intel's IP suite in Quartus Prime® will attract more industrial customers.

1.3 Using this document

This document serves as a technical guide for the Intel FPGA team.
Detailed information about process implementation and results can be found in section 2.
Commercial considerations for marketing and business development are in section 4.
Our code and Simulink files can be found in the section 3.
To view a video demonstration of the full Simulink workflow please see the video at:
https://youtu.be/8jF9seg6sp4

2 Implementation (Process and Results)

2.1 Research stage

Neural Network Theory

A neural network is a means of performing machine learning.
They consist of interconnected layers of neurons. Figure 1 shows a picture example.

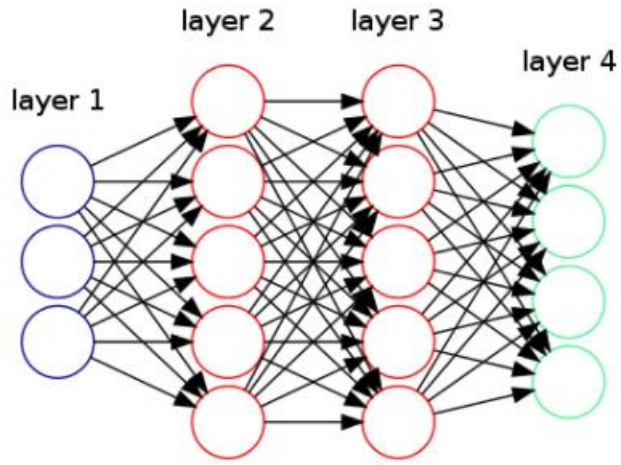


Figure 1 – Example image of neural network and its connections.

A neuron performs weighted sums on its inputs. Each input has its own weight. With the right weights, Neural networks can act as powerful function approximators.
A trainable constant bias value may also be added to the total value for each Neuron.
Finally, the neuron applies an activation function to the summation value. This activation function introduces non-linearity to the output of the neuron. Examples include tanh, sigmoid, and ReLU function. See Figure 2.

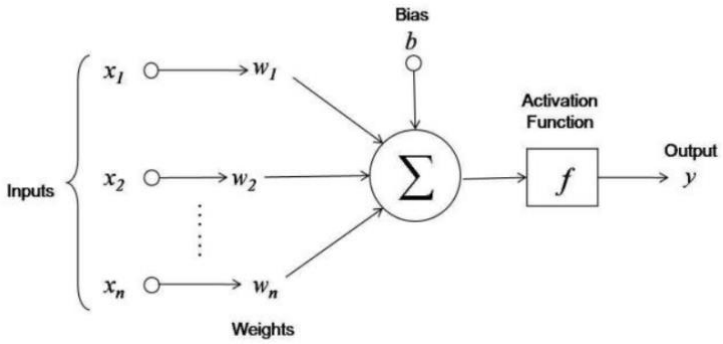


Figure 2 – Example of what goes on inside a neural network's 'neuron'.

The process of finding the best weights and bias for a network is known as training.
Source: https://towardsdatascience.com/first-neural-network-for-beginners-explained-with-code-4cfd37e06eaf

Types of machine learning and backpropagation

There are three general areas of machine learning:

Supervised learning generally refers to system in which labelled input/output data is available, and the objective is to find a function that connects them.

Unsupervised learning is used to draw conclusions about **unlabelled** input data.

Reinforcement learning is where an agent (neural network) learns how to behave in an environment by performing actions and observing their rewards, as determined by a **reward function.**

To train a neural network as part of a supervised learning problem, the backpropagation algorithm is used:

- Calculate the forward propagation of each input/output pair.
- Evaluate the error term for output of the final layer and the target output.

$$\delta_1^m = g_o'(a_1^m)\,(\hat{y}_d - y_d)\,.$$

Equation 1

- Backpropagate the error terms for all the hidden layers

$$\delta_j^k = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

Equation 2

- Evaluate the partial derivatives of each individual error with respect to its connection weight.

$$\frac{\partial E_d}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}.$$

Equation 3

- Get total gradient by combining gradients for each input-output pair.

$$\frac{\partial E(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial}{\partial w_{ij}^k} \left(\frac{1}{2} \left(\hat{y}_d - y_d\right)^2 \right) = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial E_d}{\partial w_{ij}^k}.$$

Equation 4

- Update weights by subtracting by product of total gradient and learning rate.

$$\Delta w_{ij}^k = -\alpha \frac{\partial E(X,\theta)}{\partial w_{ij}^k}.$$

Equation 5

Source: https://brilliant.org/wiki/backpropagation/

RNNs

RNNs (Recurrent Neural Networks) are a type of neural network architecture that introduce short-term memory. See Figure 3 for an example.

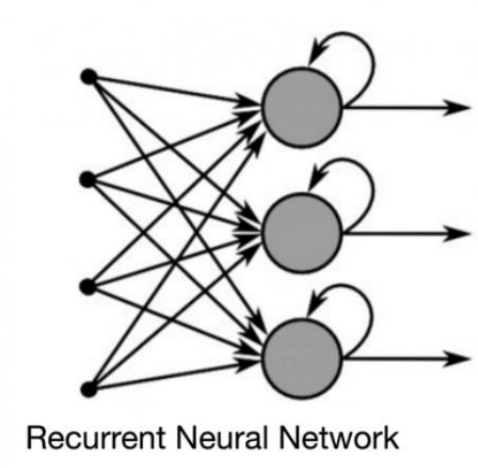


Figure 3 – Recurrent Neural Network example of action.

RNN memory is usually short-term and suffer from the infamous 'Vanishing Gradient' problem. LSTM (Long Short-term Memory) units is a variant of RNNs which is supposed to help with this and introduce longer lasting memory.

LSTMs have three gates:

- **Input** – decides whether to let new input in.
- **Forget** - deletes information when it has been determined to be unimportant.
- **Output gate** – determines the importance of the output at the current timestep.

This memory cell decides whether to store or delete information based on its importance. Weights assign importance to information and are learned through backpropagation. See Figure 4 for more detail.

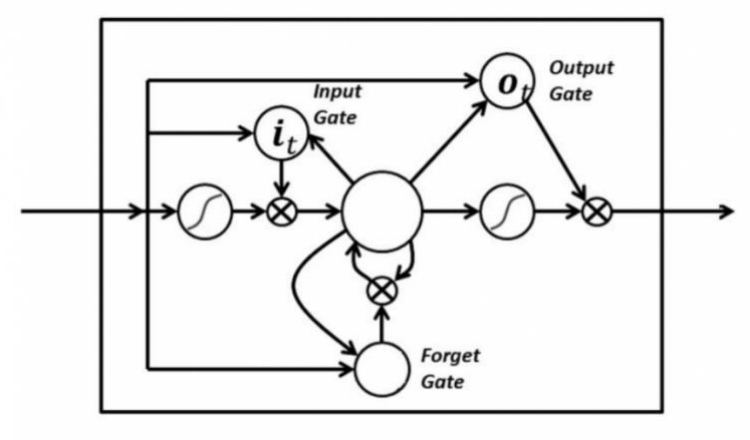


Figure 4 – Inner workings of an LSTM unit.

Source: https://builtin.com/data-science/recurrent-neural-networks-and-lstm

2.2 Initial Attempts

Naïve ANN

The first attempt at finding appropriate neural network weights was to construct a simple three-layer Feedforward Neural Network in Simulink, as shown in Figure 5, and generate random weights through a MATLAB script.

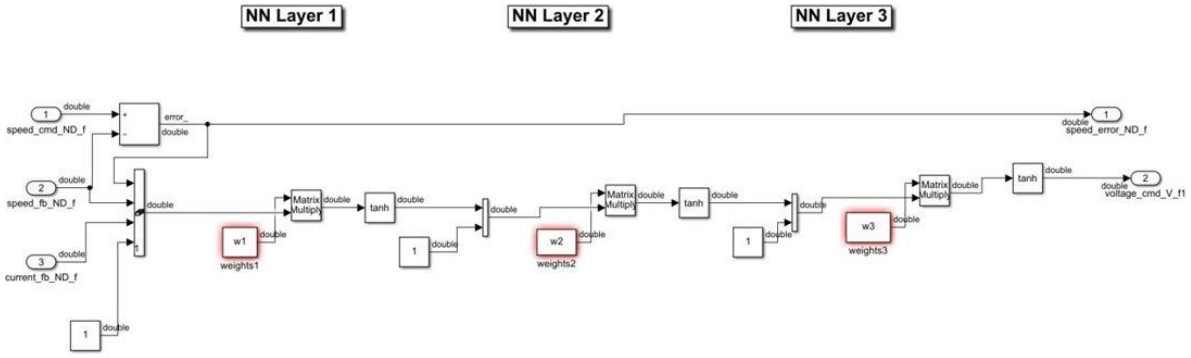


Figure 5 – Feedforward Neural Network in Simulink.

By repeatedly simulating the model with random weights and saving those that gave the lowest speed error across the simulation, a lower speed error was achieved than the PI controller within 10 minutes of searching. Shown in Figure 6.

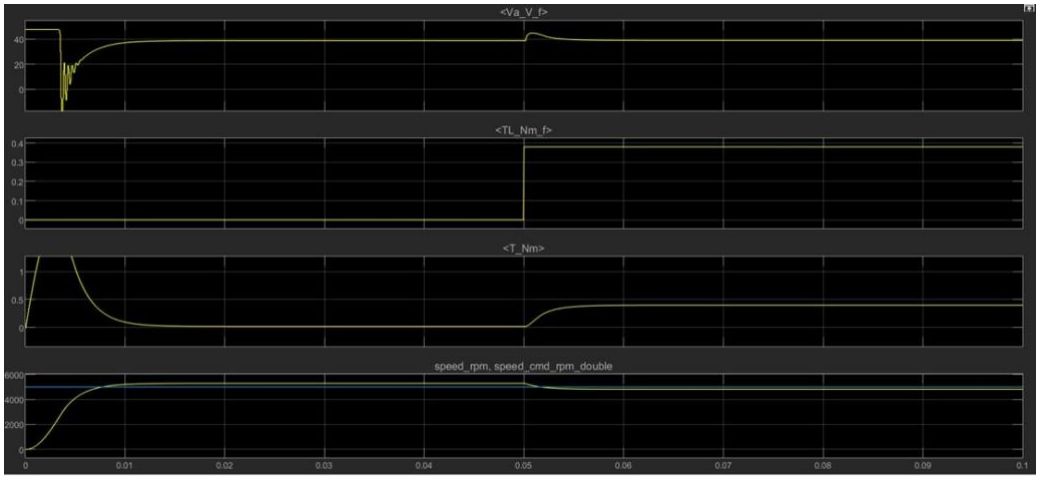


Figure 6 Feedforward Neural Network Response with weights found through random trials of 5000 rpm reference signal

However, there was a steady state error when the load torque was introduced and when a different speed command was given than the constant 5000 rpm signal the weights were found to perform badly. When the training sequence was increased to incorporate different speed commands and load torques, the results show that this method of finding weights is not sufficient as there was a large error. See Figure 7.

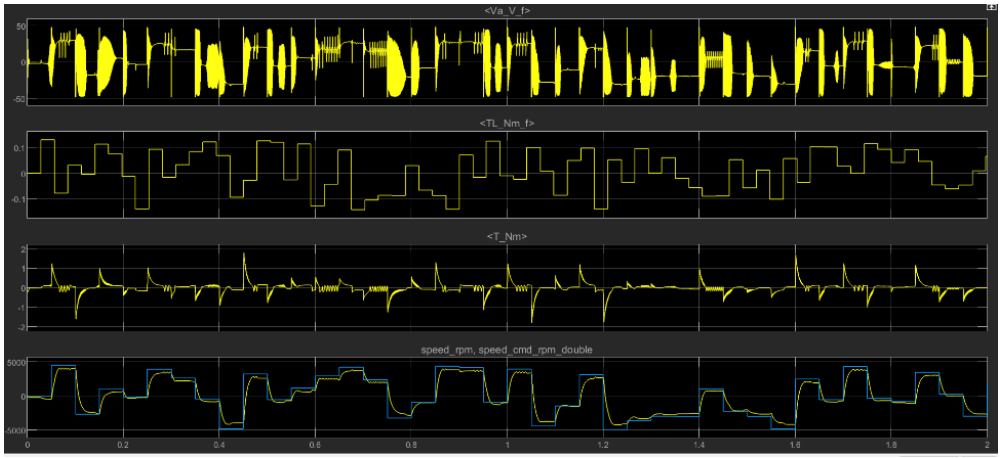


Figure 7 Feedforward neural network response with weights found through random trials on when trained on longer sequence

It was also clear this method would not be able to handle a more complex problem as the search space increases for larger networks or when increasing the number of observations.

Naïve RNN

A Recurrent Neural Network (RNN) was created in Simulink (see figure 8 and 9) and random weights were generated by the same method as the Naïve ANN and trained on a constant speed command with load torque introduced halfway through the simulation.

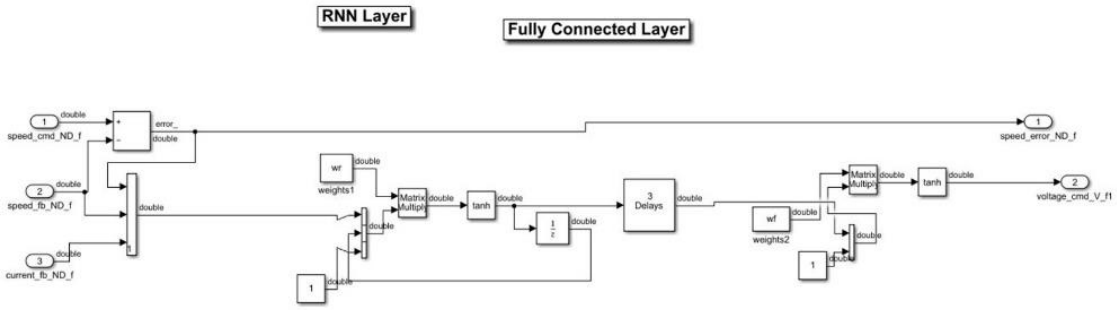


Figure 8 – Recurrent Neural Network modelled in Simulink.

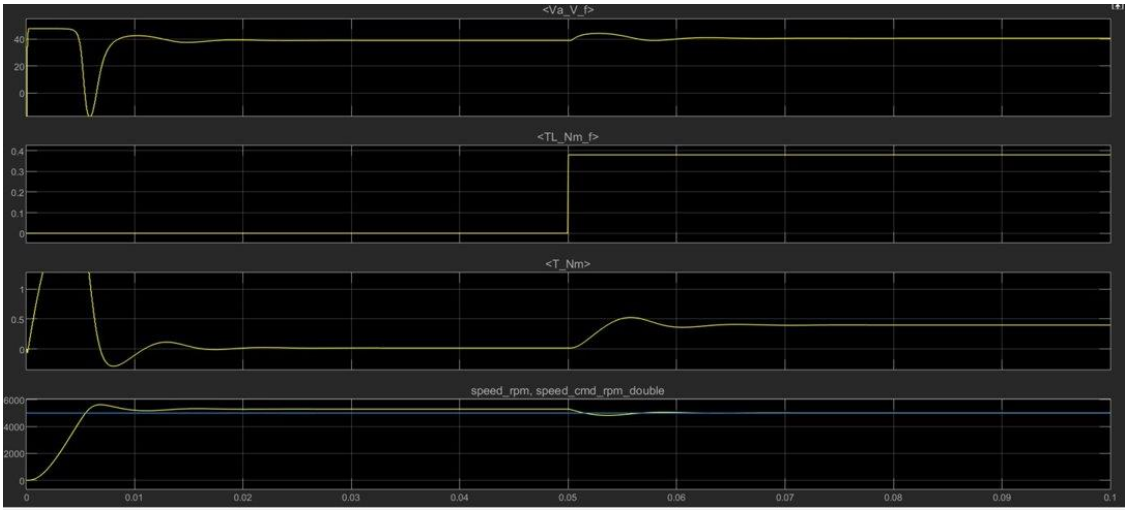


Figure 9 - RNN response with weights found through random trials on constant reference speed.

The results suggest an improvement in the steady state error when compared to the feedforward network trained using the same method.

2.3 Reinforcement Learning

Reinforcement Learning Theory

In the RL (Reinforcement Learning) context, we refer to our controller as an Agent whose goal is to learn a behaviour policy – it must choose the best Action (which is output voltage) to perform in each state (where the state is defined by the motor speed and current feedback at that instant in time).
The motor model becomes our **Environment,** and the agent will interact with this.
The agent will behave randomly at first to **explore** the effect of different actions but will learn with experience to choose better actions. See Figure 10.

We feedback how 'good' actions were by defining a Reward Function. For example, we may choose to give a positive reward for an action which makes the speed tracking error smaller or give a negative reward for an action which uses a high voltage and therefore lots of energy.
The 'brains' of the Agent is our neural network, and the weights are updated in such a way as to learn a policy that maximises the reward.

There are many different types of RL algorithm, and we spent much of our project time experimenting with and comparing different agents. Fortunately, MathWorks provides the RL Toolbox which supports many different types of agents and makes the reinforcement learning much easier to set up.

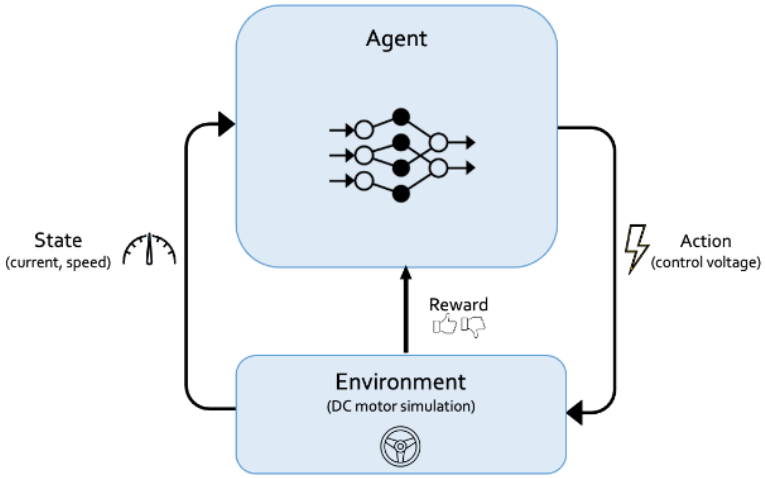


Figure 10 – Flow Diagram of Actor and Environment interactions.

States and Actions

The state consists of observed measurements from our plant (reference speed command, speed feedback, current feedback).

We also experiment with supplying extra pre-processed observations (speed error, current error, energy usage, previous output from last step) so that the network has less complexity to learn. We may also experiment with adding memory (recurrent feedback connections) so that the network can have internal state memory. The action is simply the scalar output voltage supplied to the motor.

Target Functions (relevant to TD3)

To increase stability during training, time delayed copies of the actor and critic networks are also stored, known as the target actor and critic networks. These targets are updated less frequently than the main actor/critic networks so provide a (relatively) more stable estimate of the actor and critic functions (avoids having a moving target). The target functions are used to calculate the loss function when updating the weights of the critic network.

Periodically (once every 15 time-steps) the "main" networks are used to update the target networks. A "hard" update means that the parameter values from the main actor/critic networks are copied directly the target networks. In practice a smoothing factor is used to update each target parameter to a new value which is a weighted average between its previous value and the corresponding "new" value from the main network.

Reward Function

A reward function is used to quantify how well the agent is performing at controlling the motor and is one of the most important hyperparameters we can tune to improve the agent. We start by using the squared speed error and the squared current measurements with negative weightings to encourage low error and penalise energy usage.

Episode

Training is performed for a maximum number of "episodes". Each episode involves running the DC motor simulation from the start with a random speed command and load torque disturbances. Reasonable performance so far has been observed within 10 episodes which can take several hours to train.

Experience Buffer

At each time step, corresponding state, action, and reward data is added to a circular "experience buffer". Specifically this is a record of each state, action taken, corresponding reward and next state (S,A,R,S') that the agent has experienced during the training. In our case the buffer can hold up to 2e6 samples before it overwrites old data.

Off-Policy vs On-Policy

One large differentiator between different Reinforcement Learning algorithms is whether it learns 'On-Policy' vs 'Off-Policy'.

The difference between them can be summarised as: On-Policy uses the same Policy (or Behaviour) which it is currently following in order to make future actions. The current policy is then updated based on the reward of that action.

Comparatively, Off-Policy uses the action of a policy which differs from the one it currently has. The current policy is then compared the reward this new policy's action gave and updated appropriately.

As an example, On-policy is usually quite useful for when you have an Agent which has the possibility to explore many different scenarios. Whilst Off-policy is useful when you have an Agent which might encounter many different scenarios when deployed but might not have the ability to train on all of these before-hand.

Markov Decision Process

To simplify greatly for our purposes, a Markov Decision Process is a process where the knowledge of the current state gives us the ability to map all future ones.

An example would be a game of chess, the knowledge of the current positions of each piece on the board allows us to know all possible future positions.

Many times, we might not be able to know the complete current state. An example would be a photograph of a particle in space. From the picture, we would only be able to deduce the current position of the particle, but not its velocity nor acceleration.This is then called a POMDP, or "Partially-Observable-Markov-Decision-Process". This is, according to our beliefs, what we are dealing with in this task.

Overview of Actor-Critic

This video resource: https://youtu.be/7cF3VzP5EDI gives a good overview of different approaches to Reinforcement Learning. We use an Actor-Critic method because our action space is continuous.

The **actor** is a policy network. Put simply, it is a Neural Network which decides which action to take (i.e output voltage value) based on the current state (feedback measurements + internal state).
It may be deterministic (outputs the action directly) or stochastic (outputs a probability distribution of actions which is sampled).

The **critic** is a Q-value network which takes an action and state as input and produces the Q-value as output. The Q-value of an action-state pair is the **expected long-term reward** of taking that action in that state.

During training the critic will start to learn what actions are "good or bad" in a given state. The actor will use the critic's knowledge of "good and bad" actions to learn the best actions to take in each state. This is detailed in figure 11.

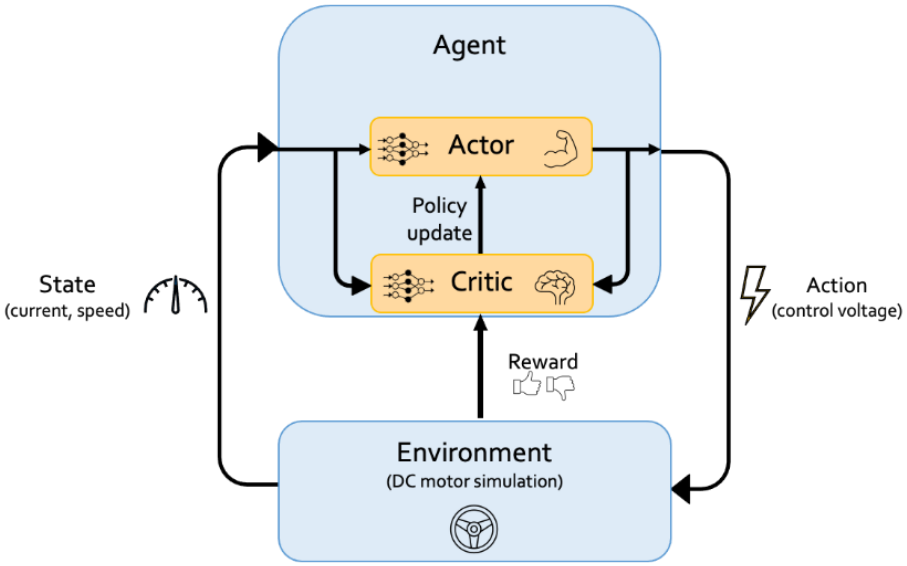


Figure 11 – Actor-Critic Flow Diagram for interacting with environment.

Implementing Reinforcement Learning in Simulink using the RL Toolbox

MATLAB Script

The Simulink model interfaces with the reinforcement learning process through the RL Agent block provided by the RL Toolbox

The RL Agent block can be used to implement many different kinds of network architectures.
A MATLAB script is used to define the agent architecture and hyperparameters used for training.
The architecture of the. agent and critics is created by specifying the layers and activation functions of the network as part of a layerGraph. An example is shown in figure 12.

```
%Define critic network architecture
statePath = [featureInputLayer(numObservations,'Normalization','none','Name','State')
    fullyConnectedLayer(64,'Name','fc1')];
actionPath = [featureInputLayer(numActions, 'Normalization', 'none', 'Name','Action')
    fullyConnectedLayer(64, 'Name','fc2')];
commonPath = [additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(32, 'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(16, 'Name','fc4')
    fullyConnectedLayer(1, 'Name','CriticOutput')];


criticNetwork = layerGraph();
criticNetwork = addLayers(criticNetwork,statePath);
criticNetwork = addLayers(criticNetwork,actionPath);
criticNetwork = addLayers(criticNetwork,commonPath);
criticNetwork = connectLayers(criticNetwork,'fc1','add/in1');
criticNetwork = connectLayers(criticNetwork,'fc2','add/in2');
```

Figure 12 – Script for defining the Neural Network in MatLab.

Small changes in the hyperparameters were found to have a large effect on results such as the training time, exploration of different policies and whether the weights converged to a local maximum.

Smoothing Factor controls how the weights of the target agent are updated. A smoothing factor of 1 means no smoothing, so the weights can change dramatically on each update. Small smoothing factors close to 0 reduce the speed of learning as the weights take more updates to change significantly.

A discount rate of 0 means only the immediate reward at each timestep is considered when choosing an action. Since the agent is part of a system involving inertia the negative effect of an action may be delayed by a few timesteps so using solely the immediate reward does not work well. Discount rates closer to 1 were found to have smoother but slower responses.

As the agent explores the action space during the training, the final weights may not provide the highest reward. By including "SaveAgentCriteria" and "SaveAgentValue" in the training options, the weights from all episodes with a total reward above a certain number are saved in a folder in the workspace and can be loaded after training. By viewing the episode vs. reward graph that opens at the start of training, the episode numbers that gave the highest reward can be found. See an example in figure 13.

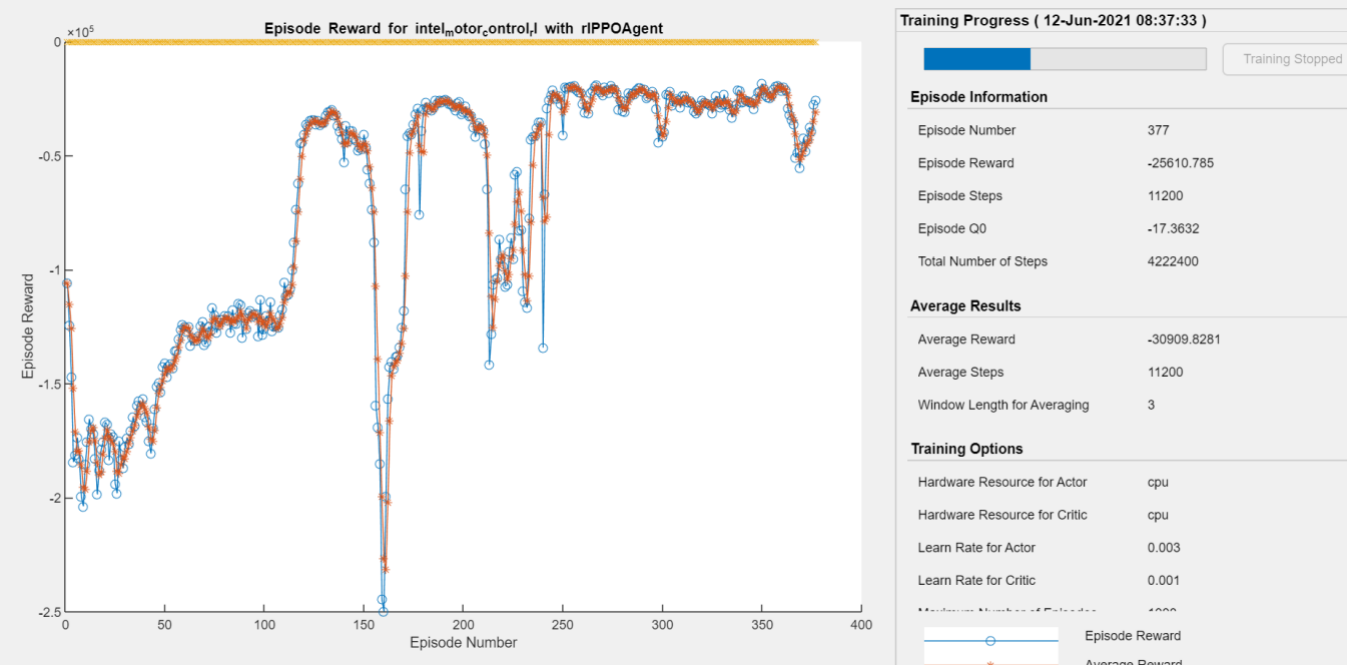


Figure 13 - Example graph of reward vs. episodes that appears when training RL agents

Environment

To avoid overfitting the data to a single training sequence of reference speeds and load torque, a random stepped signal is generated for every episode of training. Without this addition to the training process, the behaviour of the network can be unpredictable for some combinations of speed and load torque.

The time between steps is an important consideration as shorter steps would benefit agents with quicker rise times and longer steps would encourage a better steady state response. These choices should be based off a consideration of the real use for the motor.

The current and speed feedback values were delayed by one and two timesteps respectively in order to simulate the processing time introduced in a real system.

Good performance was found when the output of the agent was summed with all of the previous outputs instead of just outputting the voltage. This was introduced to the model to try and make the output voltage settle to a constant value where the agent would output 0. Simulations suggested that it improved the generalisation to new combinations of reference speed and load torque.

Reward

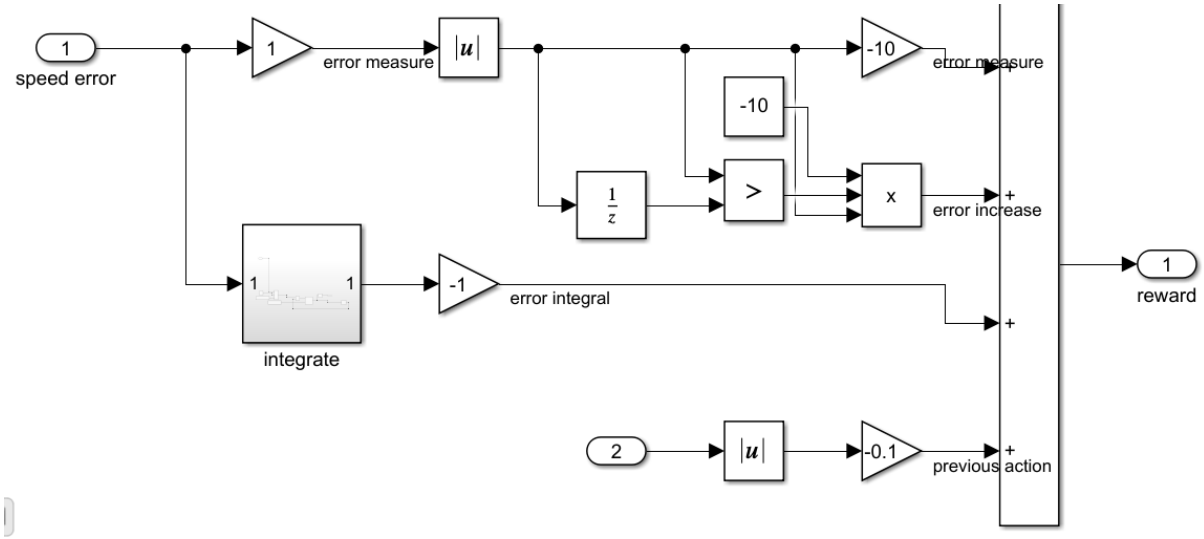


Figure 14 - Reward function used in TD3 model

Since the environment remains the same for all agents, the same reward functions can be used for different network architectures. An example of a reward function in Simulink is shown in figure 14.
Most reward terms penalised bad states with a negative reward so the agent tried to maximise the reward.

Using the absolute value instead of the squared error was found to improve the steady state error since there is less bias in the total reward towards the large error at the beginning of each reference speed step.

Reward terms involving conditionals were experimented with such as by giving a negative reward when error increased and a positive reward when error decreased or no negative reward when inside a margin of the reference speed. All these examples were found to decrease the time taken to learn a good response.

A squared previous action term was used to penalise large control signals. This was also used when the voltage output was the sum of all agent outputs to encourage smaller increments in the output as the motor speed approached the reference speed.

Finding the balance between different terms in the reward function proved difficult. When using a large weighting for reward terms involving the energy consumption, a steady state error was introduced. However, too small a weighting had very little effect on the final output.

Observations

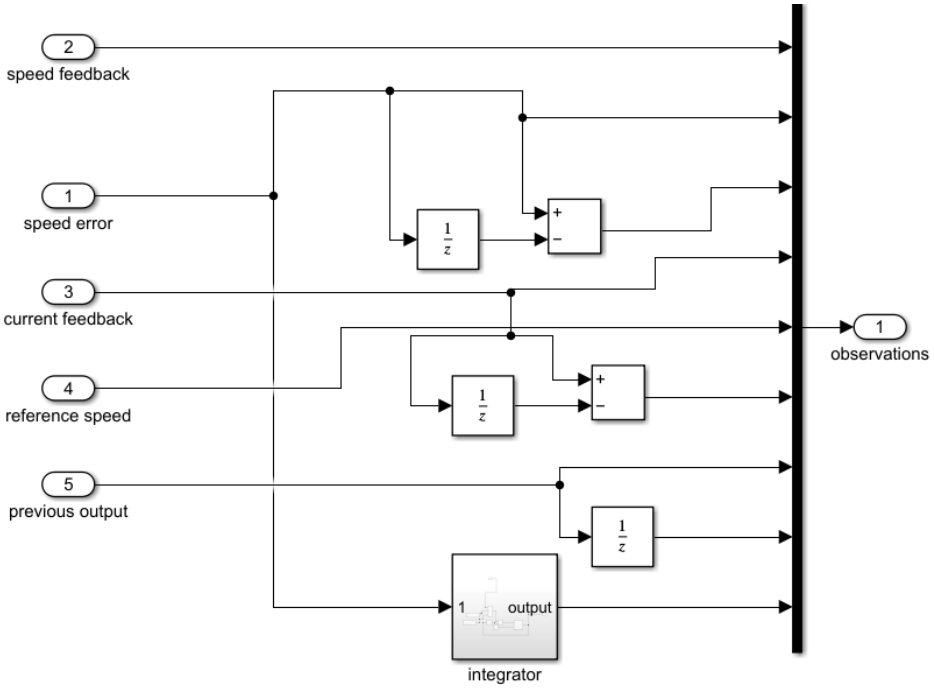


Figure 15 - Observations used in TD3 model

The three observations provided in the initial Simulink model were reference speed, speed feedback and current feedback.
The delayed output of the actor was fed back into the observations as "previous input" to encourage a smoother response.

PI controllers remove the steady-state error found in P controllers by using the integral of the speed error. PD controllers are known to have a faster response than P controllers by using the derivative of the speed error. In order to encourage these positive responses in the neural network, past speed error values can be included in the observations.

Introducing the integral of the error as an observation was found to reduce the steady state error but created additional hyperparameters such as the limit of the size of the integral.

Providing the past error values so that the network could learn to calculate the integral and differential of the error was also successful in removing most of the steady state error but the network took longer to converge and was not as consistent as when using the integral as an observation with some torque and speed commands still retaining an error. This may be because the network or training process was not complex enough to learn the transfer function from these inputs.
See an example of the observations in Simulink in figure 15.

TD3 Agent

The Twin in TD3 comes from using two critics and takes minimum as the loss to avoid overestimation of Q value. The policy update algorithm is shown in detail in figure 16.

Critics network normally updates after each timestep while target actor network is "delayed" as it is updated periodically. This is to avoid the agent training diverging when it is using a bad policy. Less updates also means more efficient, quicker training.

The motor is controlled using the actor network which chooses the next action for the given state. Some gaussian noise is added to the actor output to allow some random exploration which should help the actor "discover" new policies. As the training progresses, the variance of the noise is decayed (this causes the agent to explore less as it becomes more "confident" in its decisions).

We have found TD3 can learn a good response for controlling a D.C motor with very few neurons in the actor as a larger critic network can improve the training of the network.

TD3 reduces the number of simulations that need to be run in training compared to an On-policy like PPO as it stores state transitions, actions and rewards in an experience buffer. It randomly samples this buffer and "replays" the state transitions in training to improve its policy. The main benefit of this is that it prevents correlations between samples from neighbouring timesteps influencing the policy.

More information can be found at: https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93 and for the MATLAB implementation at https://uk.mathworks.com/help/reinforcement-learning/ref/rltd3agent.html.

Algorithm 1 Twin Delayed DDPG

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute target actions

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$$

      Compute targets

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      if j mod `policy_delay` $= 0$ **then**
        Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \mu_\theta(s))$$

        Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

      end if
    end for
  end if
until convergence

Figure 16 – Algorithm for policy updates for TD3.

Source: https://spinningup.openai.com/en/latest/algorithms/td3.html#pseudocode

The code of the Agent implementations can be found in resources.

SAC Agent

SAC maximises the lifetime rewards and the entropy of the policy. High entropy encourages exploration. The policy assigns equal probabilities to actions that have same or nearly equal rewards and ensures that training does not get stuck in a local minimum. It is an off-policy agent. The policy update algorithm is shown in detail in figure 17.

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

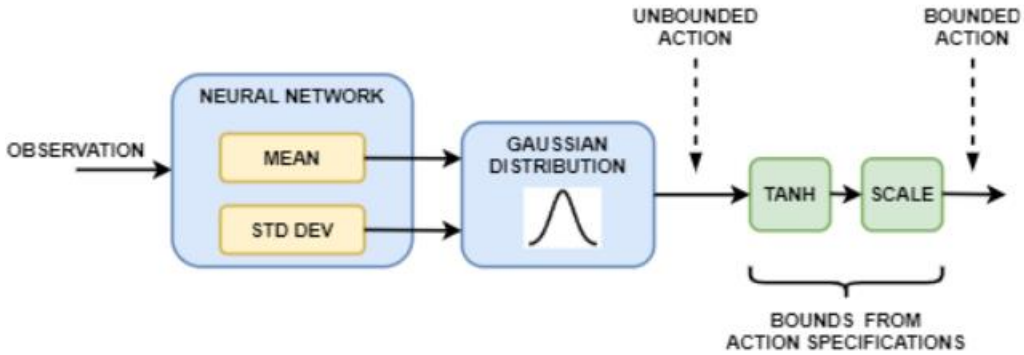
      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Figure 17 – Policy update algorithm for Soft-Actor-Critic.

Source: https://spinningup.openai.com/en/latest/algorithms/sac.html



Source: https://uk.mathworks.com/help/reinforcement-learning/ug/sac-agents.html

We implement one stochastic actor network. The network takes in observations and returns a Gaussian probability density function of potential actions (output voltages). Then the agent randomly selects an action based on the density function and scales it into the limits specified in action locations.

We also create 2 Q-value critic networks which take in the observation and action and feedback the expected long-term reward and entropy.

The actor size is restricted by the hardware limitations of the FPGA, so we make the critic networks much deeper than the actor network to learn more complex features.

PPO Agent

Introduced in 2017 by OpenAI, PPO (Proximal Policy Optimisation) is an On-Policy Actor-Critic style of Reinforcement Learning Agent.

To summarize how it works:
After a number of actions have been taken and experiences recorded (i.e. the reward given for those actions), the Agent proposes a new policy update which in essence is a traditional gradient optimisation of the Neural Network's weights. The Agent then truncates this gradient update to be within a value chosen by us - in our case we always chose 0.2. Meaning that the difference between policy updates cannot exceed 0.2. This helps make sure the Agent doesn't stray too far from past policies. The Agent also includes an Entropy factor, meaning that at each iteration the Agent is incentivized to take actions differing from its policy to some degree.

More information can be found at:
https://towardsdatascience.com/understanding-and-implementing-proximal-policy-optimization-schulman-et-al-2017-9523078521ce

The algorithm for policy updates in PPO is detailed in figure 18.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Figure 18 – Policy update algorithm for PPO.

Source : https://spinningup.openai.com/en/latest/algorithms/ppo.html

More information regarding the PPO Agent can be found in MATLAB's Documentation:
https://www.mathworks.com/help/reinforcement-learning/ref/rlppoagent.html

The code of the Agent implementations can be found in resources.

2.4 Testbench Results

The most promising models were loaded onto a testbench where their performance was measured for a chosen sequence of speed commands and reference torques.

MODEL	CUMALATIVE SQUARED SPEED ERROR	CUMULATIVE SQUARED ENERGY	OBSERVATIONS
PI	0.06178	93.89	Slow rise time cause of bad error
TD3	0.03864	122.3	Some overshoot, some high frequency elements in control
SAC (Sum of actions output, run deterministically)	0.04336	108.7	Steady-state error for some torques
PPO LSTM, continuous voltage output, deterministic	0.042	128.6	Good steady state and torque response, few high frequency elements, some overshoot
PPO LSTM, continuous voltage, non-deterministic	0.04237	131.7	Higher frequency voltage signal than deterministic version

By defining **Rise Time** as the total time between 10% and 90% of the step height and **% Overshoot** as the overshoot(rpm)/step size(rpm) x100, the following values can also be extracted.

Model	Rise Time (Speed 0 ->5000, Load Torque 0->0.2)/sec	% Overshoot	Rise Time (Speed 5000 ->-5000, Load Torque -0.4->0)/sec	% Overshoot
PID	0.00828	8.22	0.0123	4.11
TD3	0.00373	5.48	0.00390	3.42
SAC	0.00698	0	0.00438	2.05
Deterministic LSTM PPO	0.00475	6.16	0.00389	10.3

NB: There is a one accidental difference between the SAC and TD3 Simulink models which is that the limit of the size of the integral in the reward and observation is lower for SAC, is has not been investigated yet whether this is caused by the difference in response or if it is due to the different network architectures/hyperparameters.

Testbench Screenshots

PID

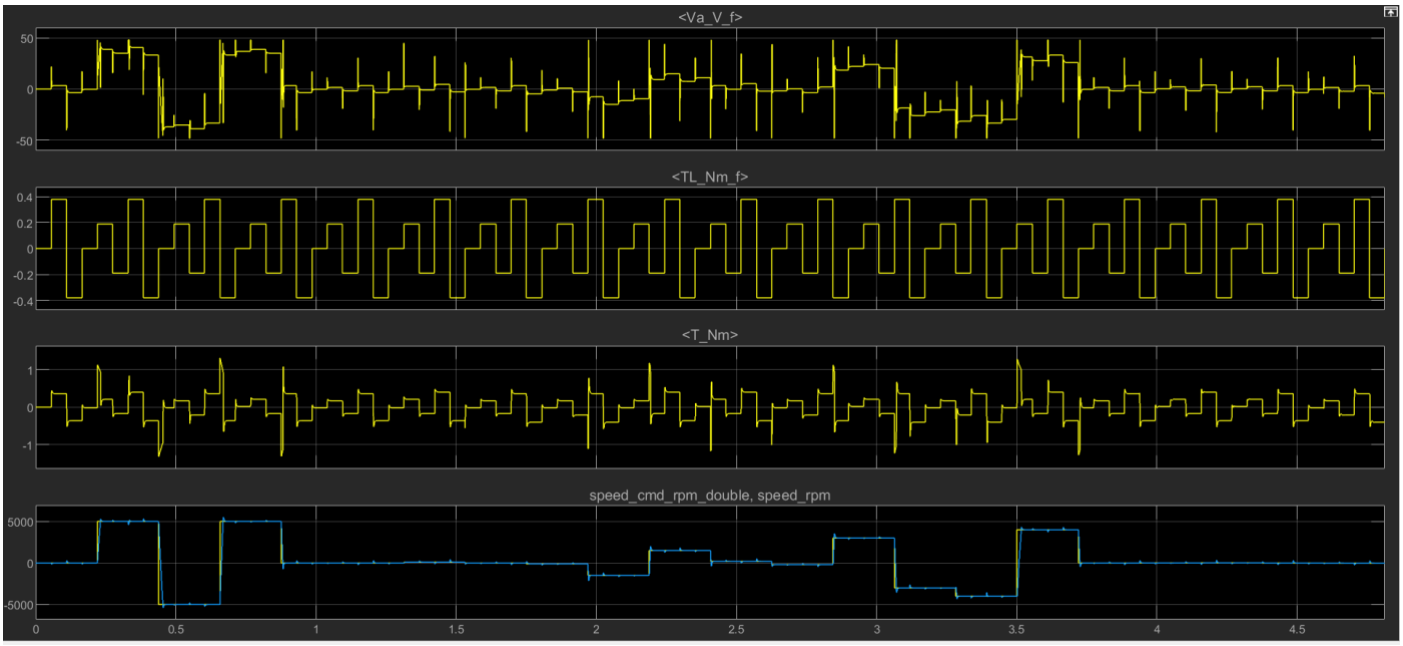


Figure 19 Simulation results when using PID controller on test sequence

TD3

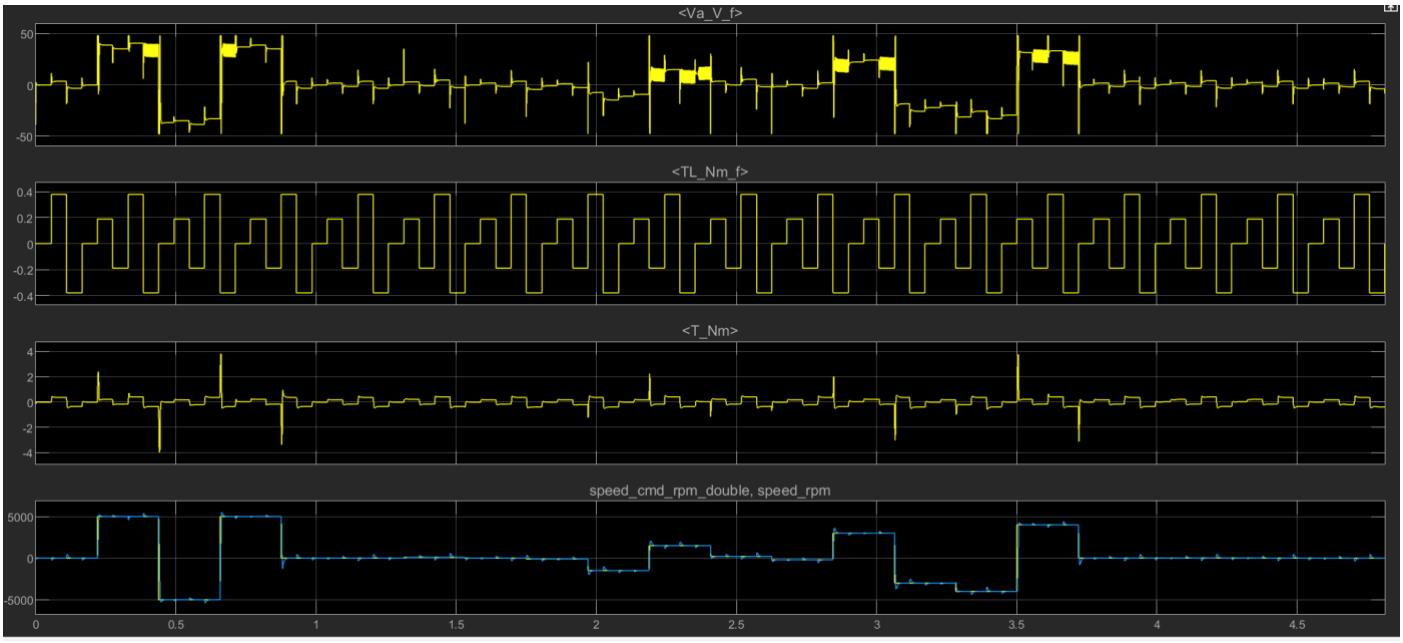


Figure 20 Simulation results when using TD3 controller on test sequence

TD3, 0.2-0.5s

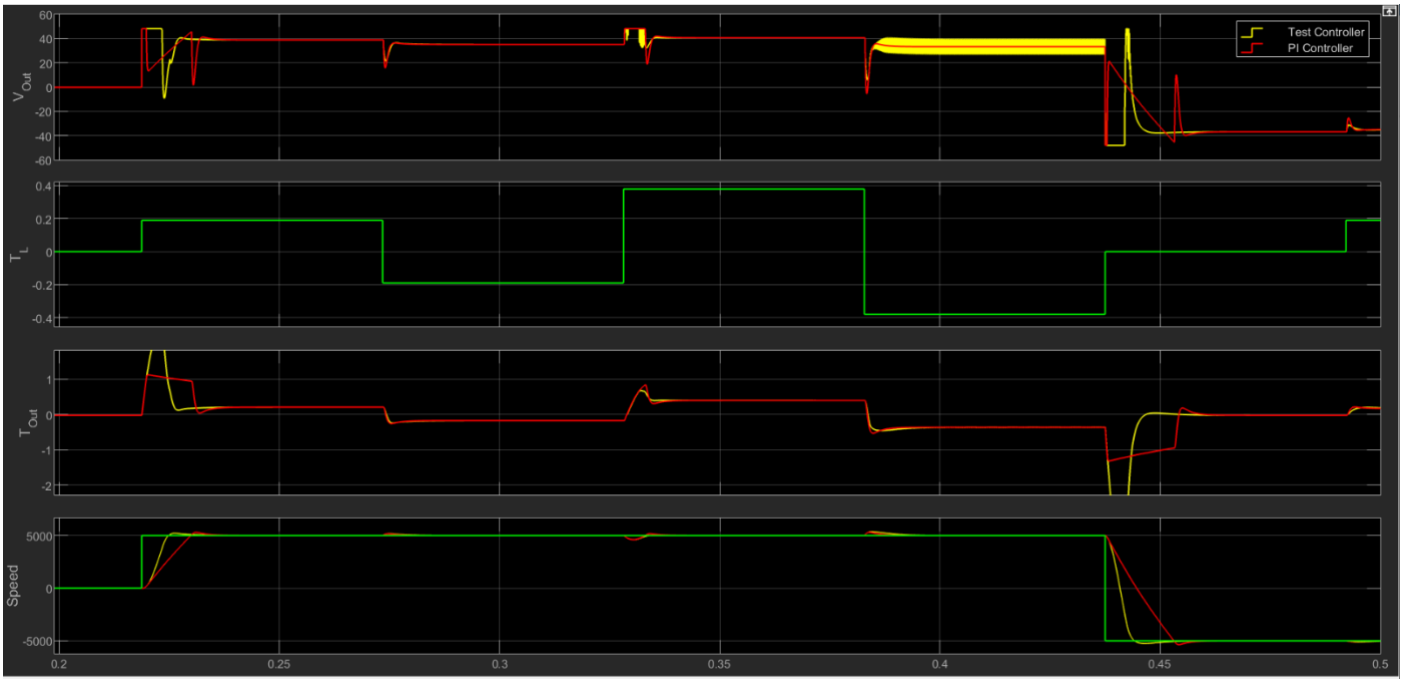


Figure 21 Closeup of TD3 response to test sequence

SAC

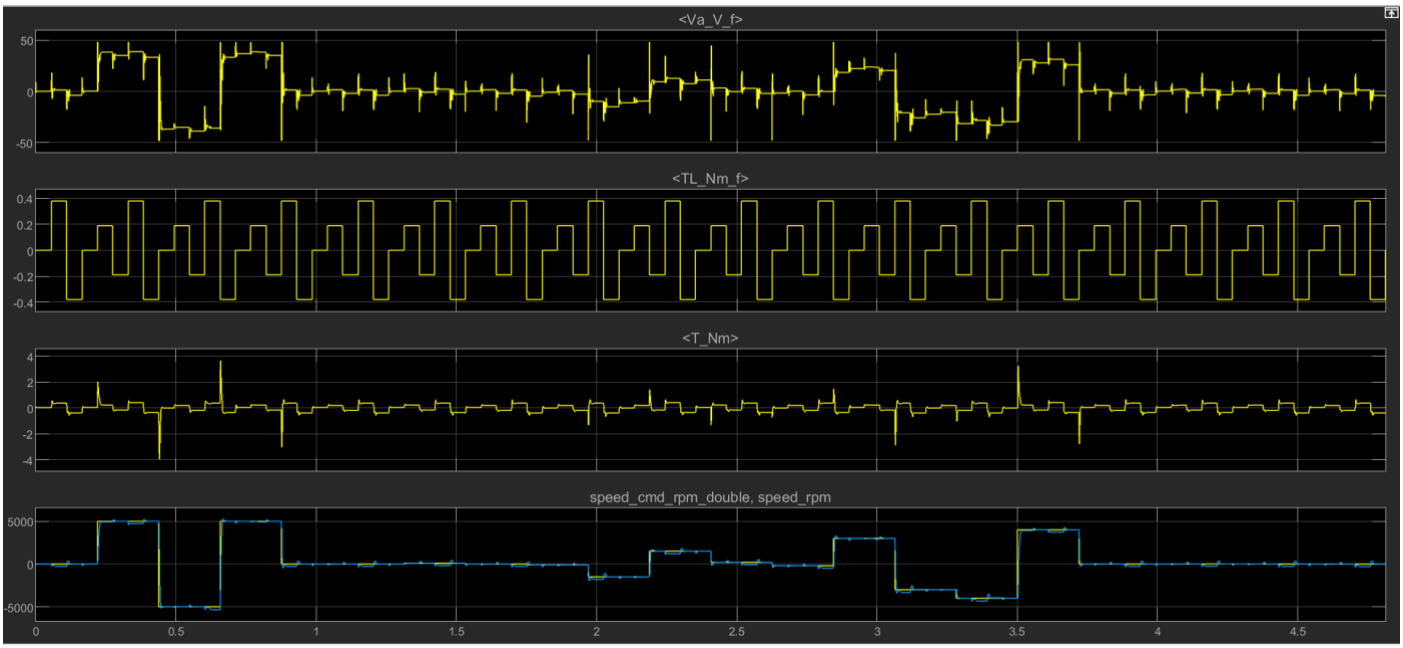


Figure 22 Simulation results when using SAC controller on test sequence

SAC 0.2-0.5s

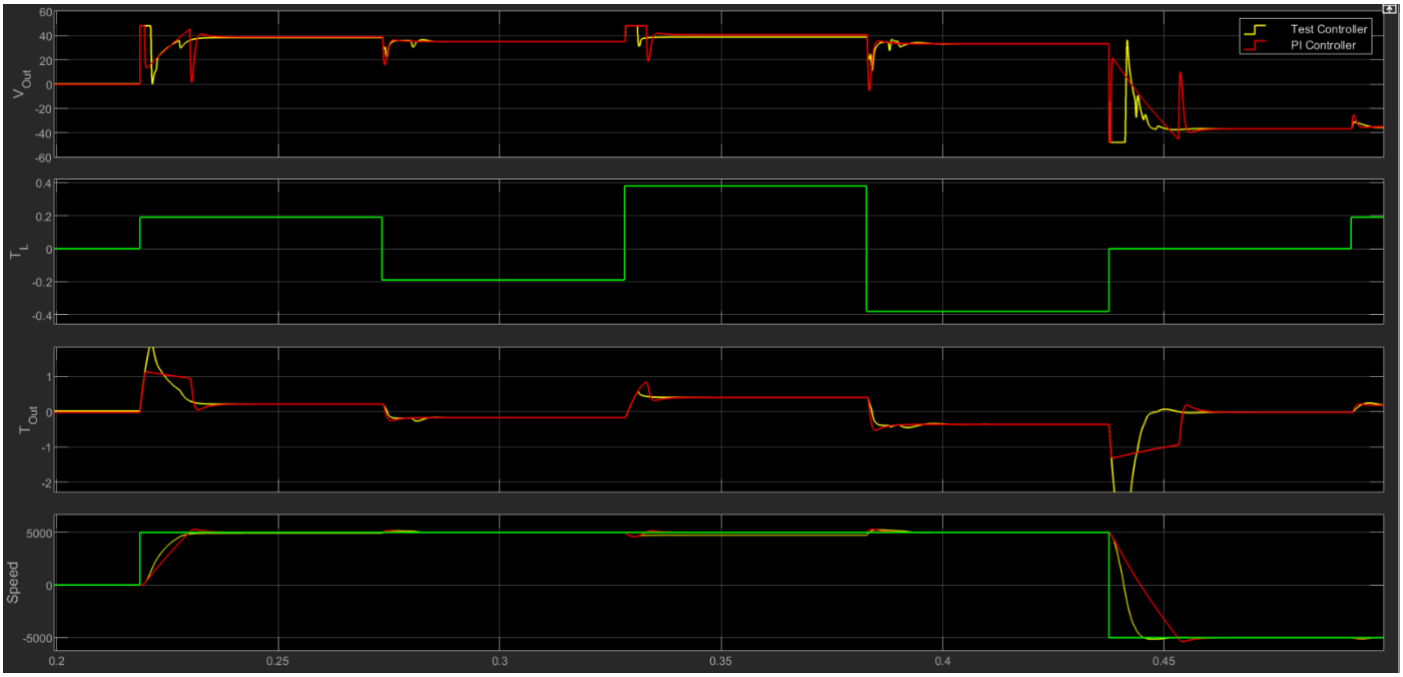


Figure 23 Closeup of SAC results for test sequence

PPO LSTM, continuous voltage output, deterministic

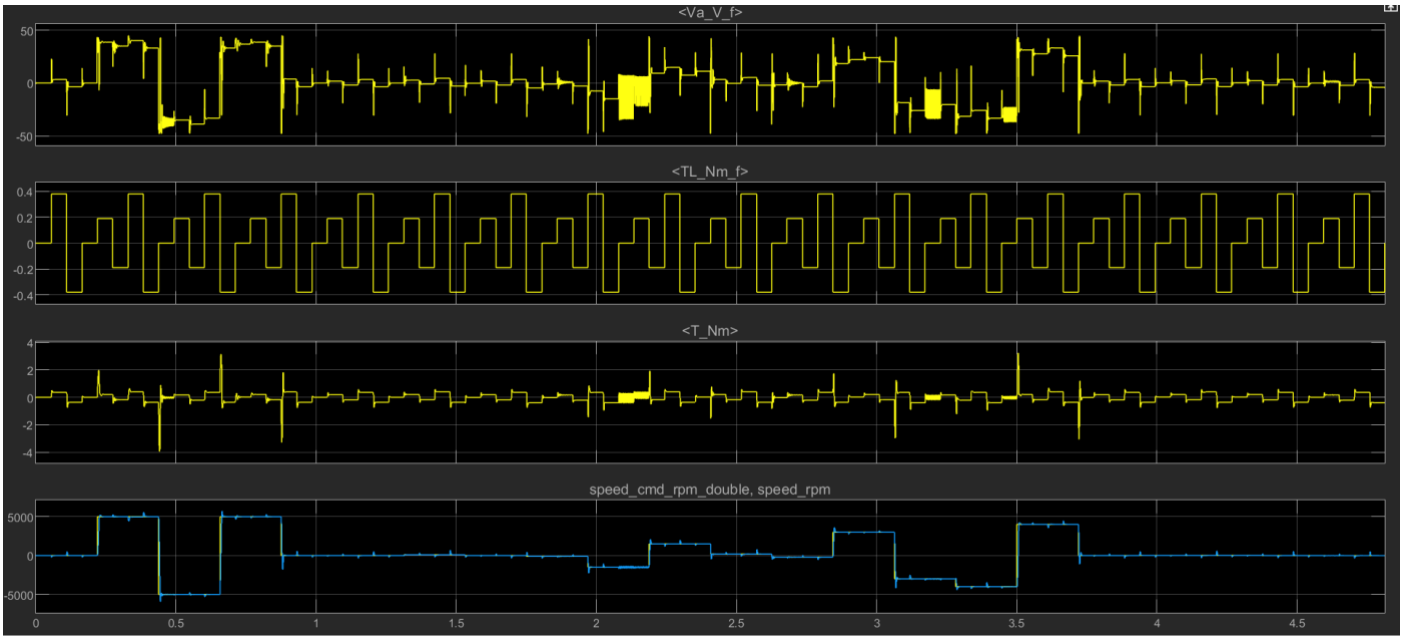


Figure 24 Simulation results when using PPO controller deterministically on test sequence

Deterministic PPO with LSTMs, 0.2-0.5s

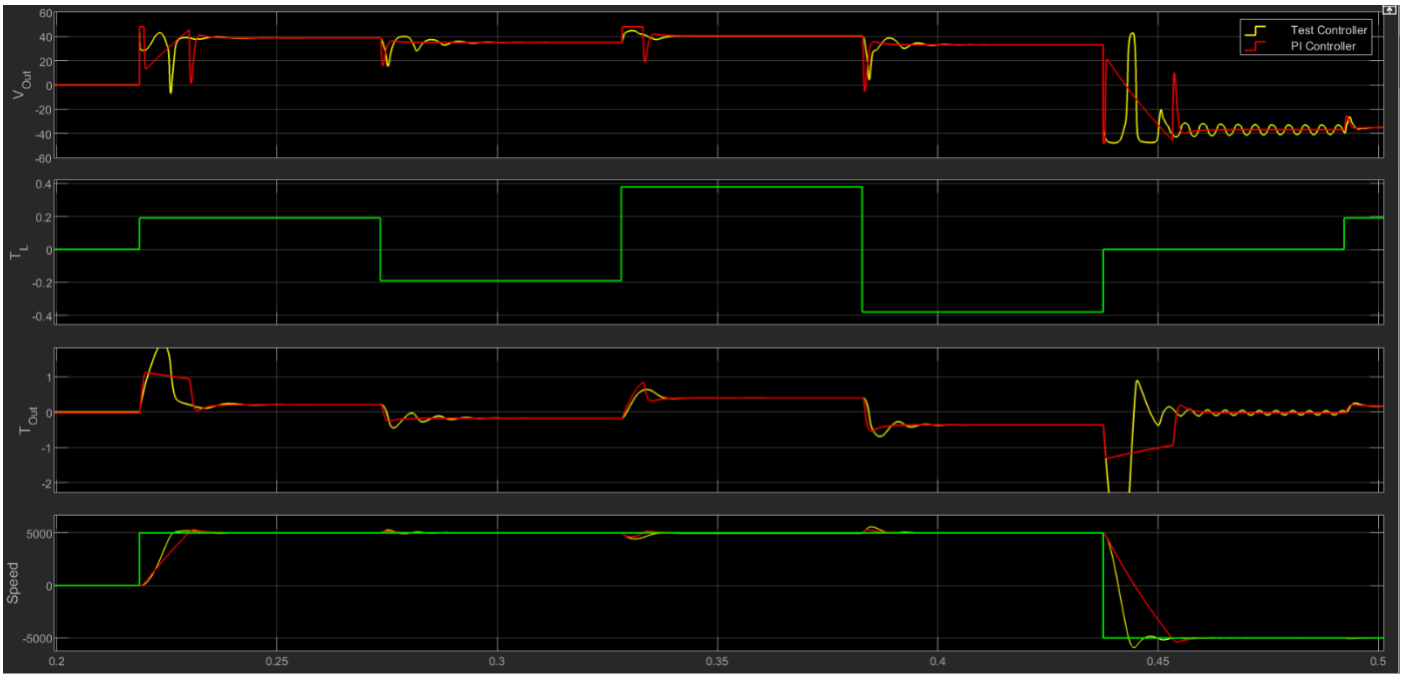


Figure 25 Closeup of deterministic response of PPO for test sequence

PPO LSTM, continuous voltage output, non-deterministic

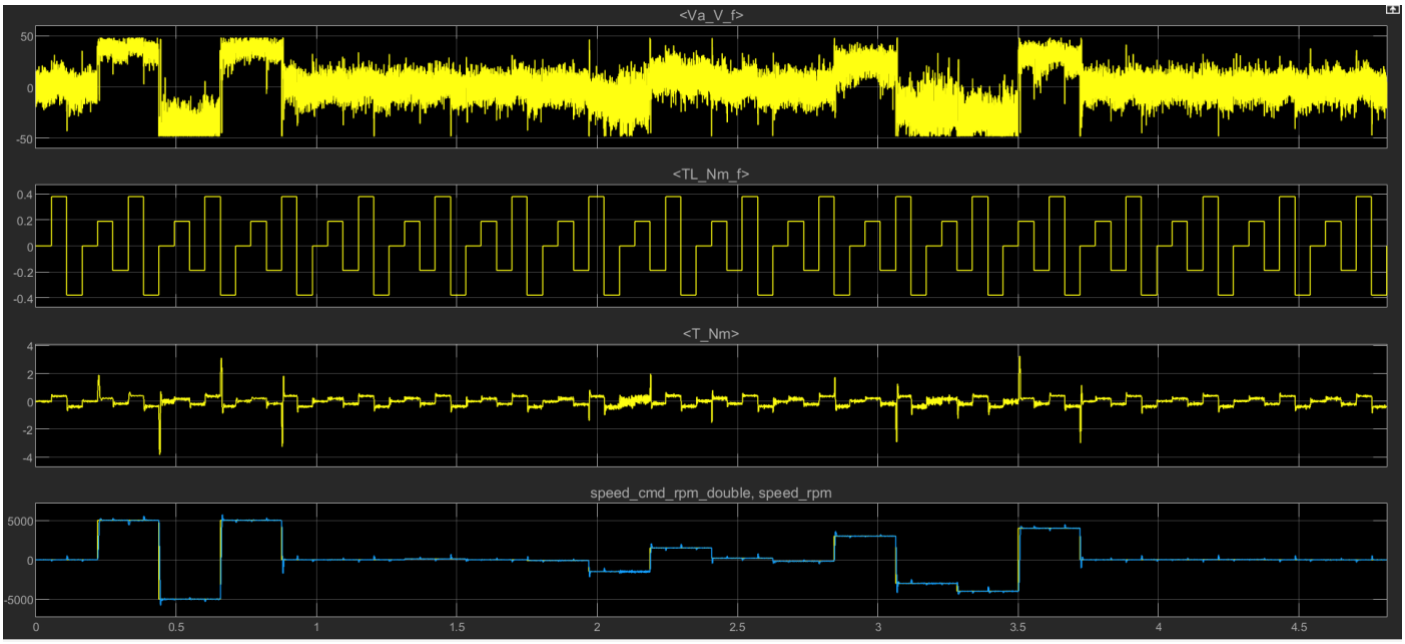


Figure 26 Simulation results when using PPO controller non-deterministically on test sequence

Non-Deterministic PPO with LSTMS, 0.2-0.5s

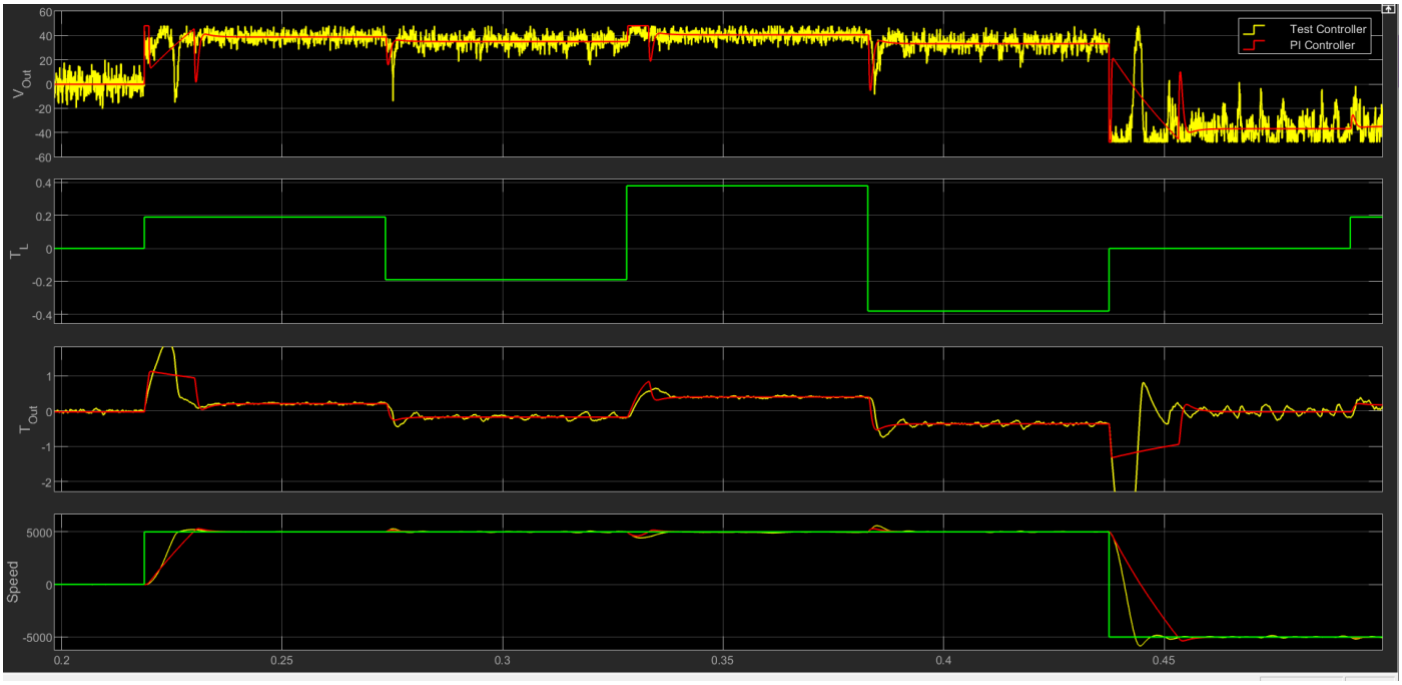


Figure 27 Closeup of non-deterministic PPO response to test sequence

2.5 HDL Conversion

Overview of HDL Conversion

To deploy the controller to an FPGA, it must be converted to Hardware Description Language (HDL) such as Verilog.

MATLAB's **HDL Coder** tool provides a simple means of performing this conversion, however the RL Toolbox Blocks are not currently compatible with HDL Coder.

In order to convert using the tool, the **Actor Network** must be implemented using compatible and HDL optimised blocks from the HDL Coder toolbox.

The default double-precision floating-point units used by the model should be converted to fixed-point units of appropriate accuracy using the fixed-point tool.

Finally, the HDL coder can be used to produce something that can be loaded onto an FPGA.

Block Level ANN

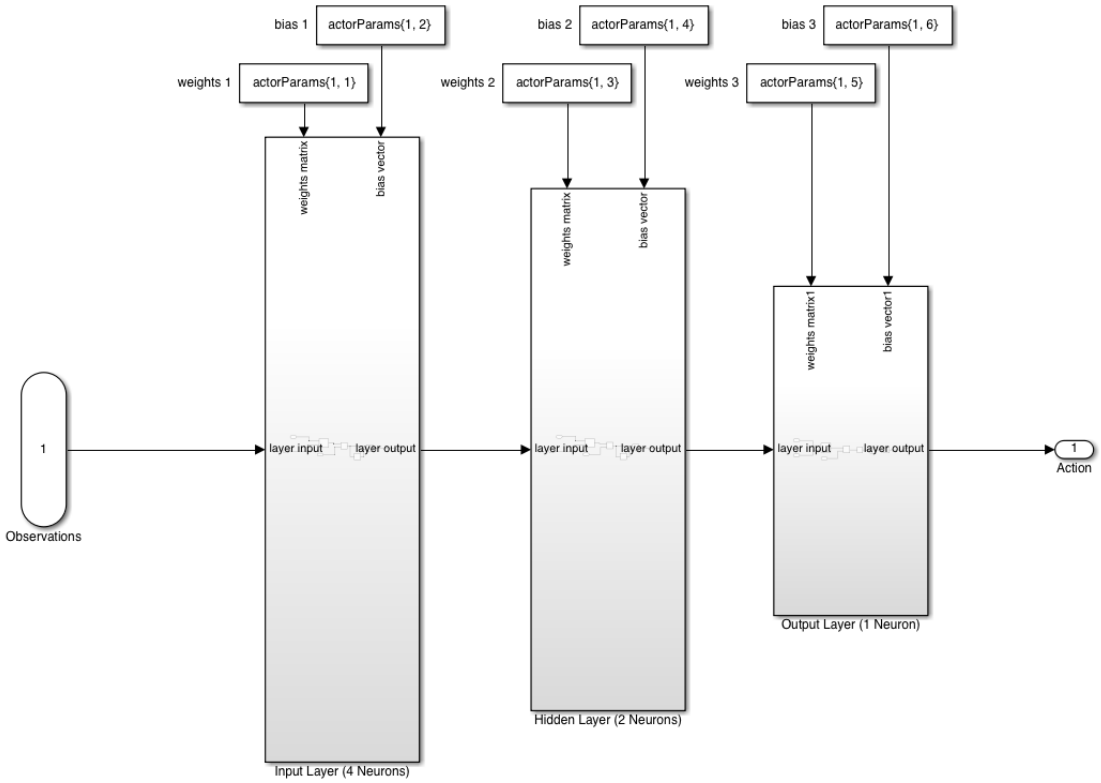


Figure 28 Block Level Implementation of Shallow Artificial Neural Network (ANN)

The above is a Simulink implementation of a 3-layer, fully connected neural network with each layer implemented as a subsystem. This implementation was tested for the TD3 agent.
All function blocks used are from the HDL coder toolbox which means they have been optimised for HDL generation.
The weighted sum within each layer can be achieved with a matrix multiplication with the weight's matrix followed by adding the bias vector. Each layer also contains either a ReLU or Tanh as its activation function.

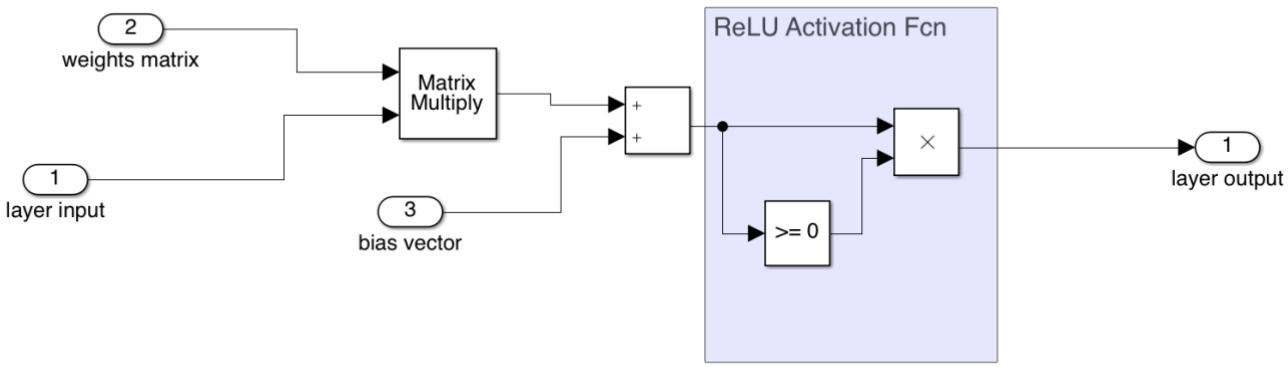


Figure 29 ReLU Activated FC Layer Subsystem (Input and HIdden Layers)

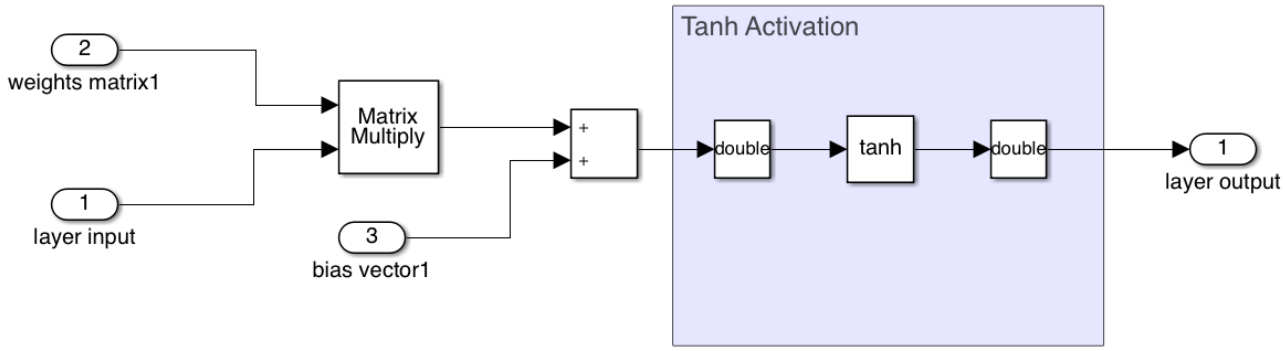


Figure 30 Tanh Activated FC layer Subsystem (Output Layer)

The ReLU can be implemented simply by setting any negative input to 0. The final layer uses a tanh function to scale the final output in the range (-1,1). There is a tanh block optimised for HDL coder which can be set to use Intel's (Altera's) optimised tanh IP block.

Fixed-Point Conversion

Once a trained actor network has been replaced by the simple neural network in Simulink, the Fixed-Point tool can be used to convert the double precision floating point number representations to a fixed-point representation. This tool runs the simulation multiple times to determine the range needed to express each signal, which can then be used to select the number of bits to dedicate to the whole and fractional parts.

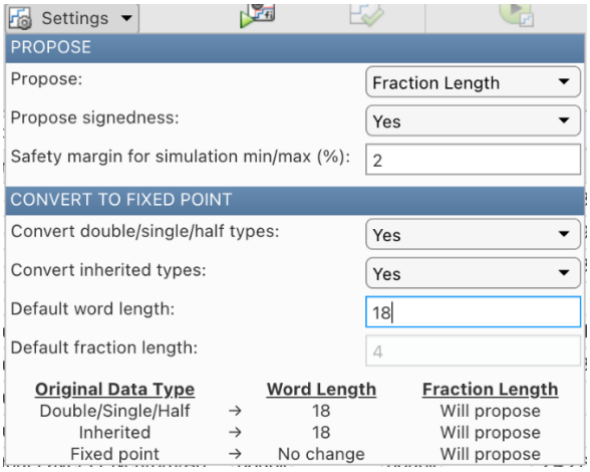


Figure 31 The desired word length for the fixed-point units can be adjusted in the settings pane. 18-bits works well for the 18-bit multipliers which are standard on FPGAs, though they could be made smaller to minimise resource utilisation.

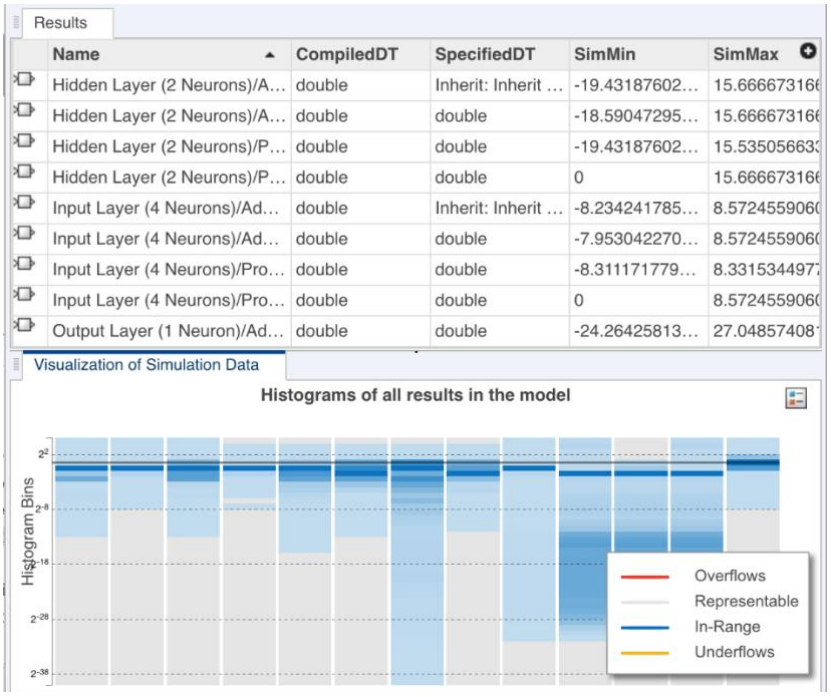


Figure 32 A histogram generated by the fixed-point tool that shows the numerical range of each signal and the distribution of signal values within the NN subsystem.

After the fixed-point tool has generated this data through iterative simulations, the ‘Propose Data Types’ button can be used to see the tool’s suggestions for fixed-point representations.

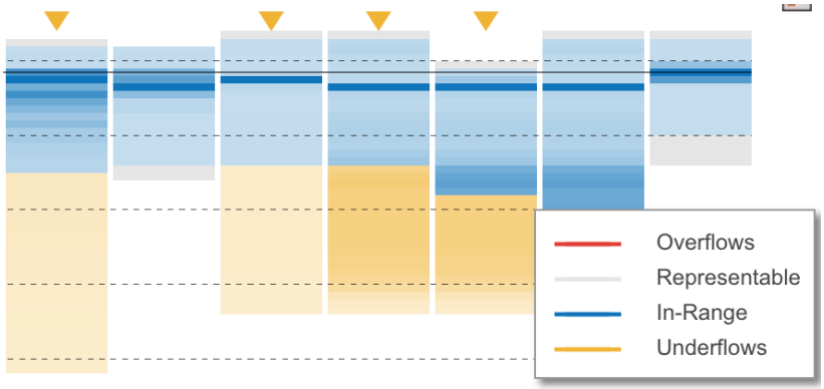


Figure 33 After proposing data types, the fixed-point tool can show where there are likely to be over/underflows when compared to a floating-point representation of the signal value.

This can affect the representable range and result in under/overflows, however the tool tries to select a representation that minimises this, within the 18-bit constraint. Selecting 'Apply Data Types' and then 'Simulate' allows for us to verify performance with the new representation.
NB: Data type conversion blocks must be used to convert between the double-precision floating-point units used throughout the rest of the Simulink model and the new custom fixed-point units used for the NN subsystem.

Figure 34 Convert blocks must be used to swap between data types within the Simulink model.

Also, the tanh function must use floating-point units to ensure compatibility with HDL coder, therefore another data-type conversion must be used around the tanh block.

Performance of TD3 Agent with Fixed-Point Units.

After converting to fixed-point units, the model can be tested again in the testbench. When using an 18-bit fixed point representation performance is almost identical to double precision floating point, as can be seen by comparing Figures 35 and 36 to earlier Figures 20 and 21. The integrated error and energy performance was unchanged by the conversion.

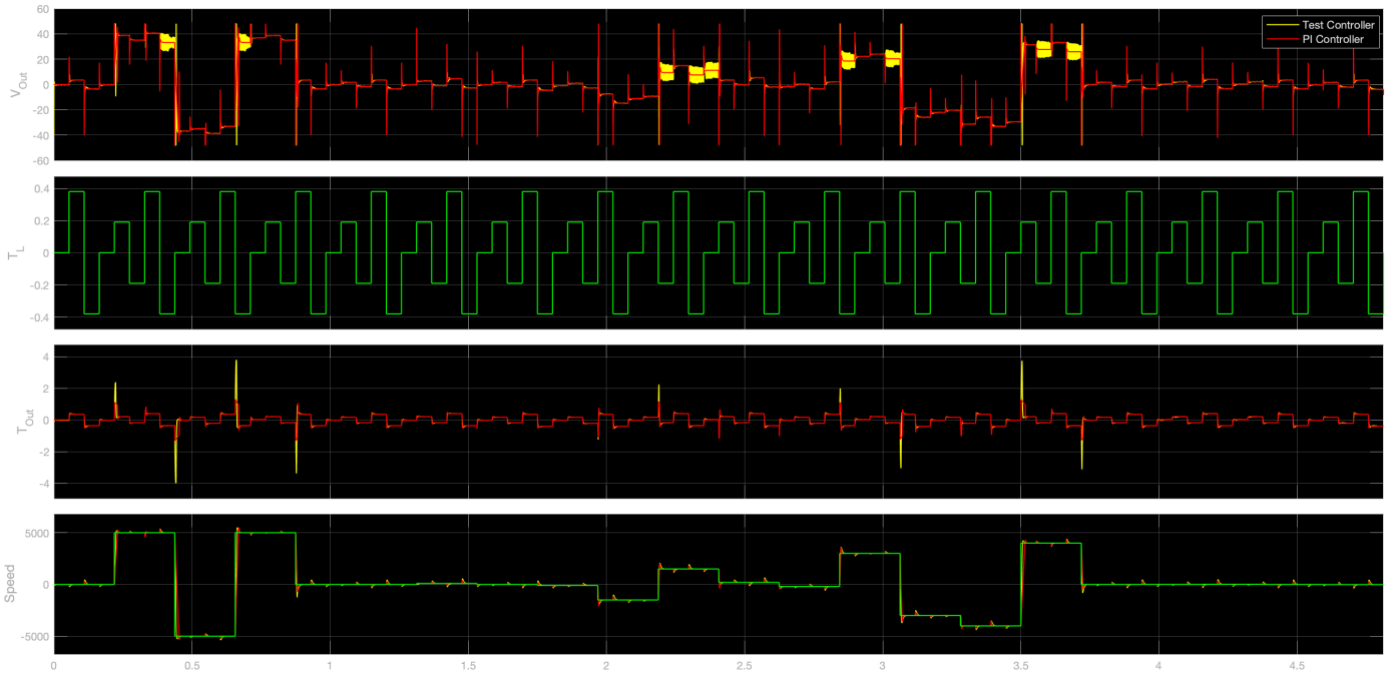


Figure 35 Testbench results of TD3 agent with 18-bit fixed-point representation of NN.

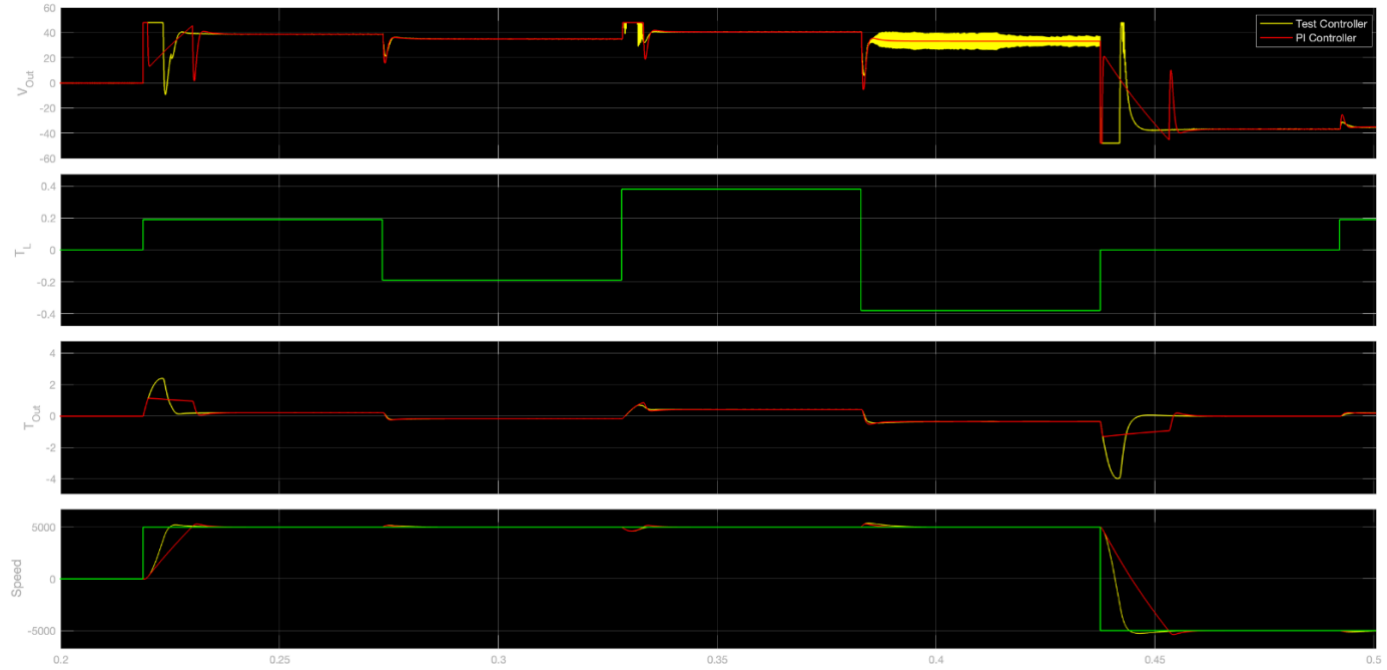


Figure 36 Close Up of TD3 agent with 18-bit fixed-point representation of NN.

However when using even smaller 9-bit fixed point representations, there is a significant difference in controller behaviour. It can be clearly seen in Figures 37 & 38 that the control voltage is much more oscillatory and therefore the speed tracking response also oscillates. This results in a higher integral of error measurement and increased energy output. However, despite this, the controller still responds faster and with less error than the baseline PI controller.

This demonstrates that an 18-bit fixed point representation is preferred however a 9-bit representation would suffice if resource usage needed to be minimised.

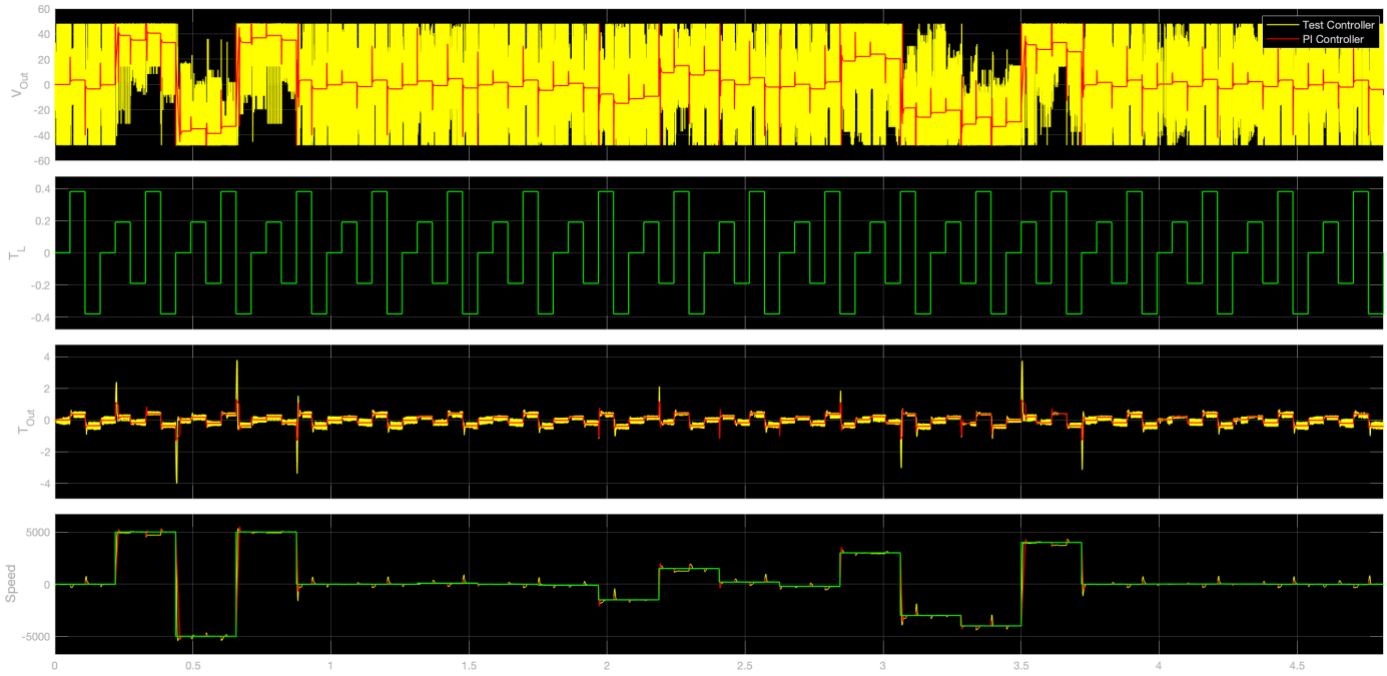


Figure 37 Testbench results of TD3 agent with 9-bit fixed-point representation of NN.

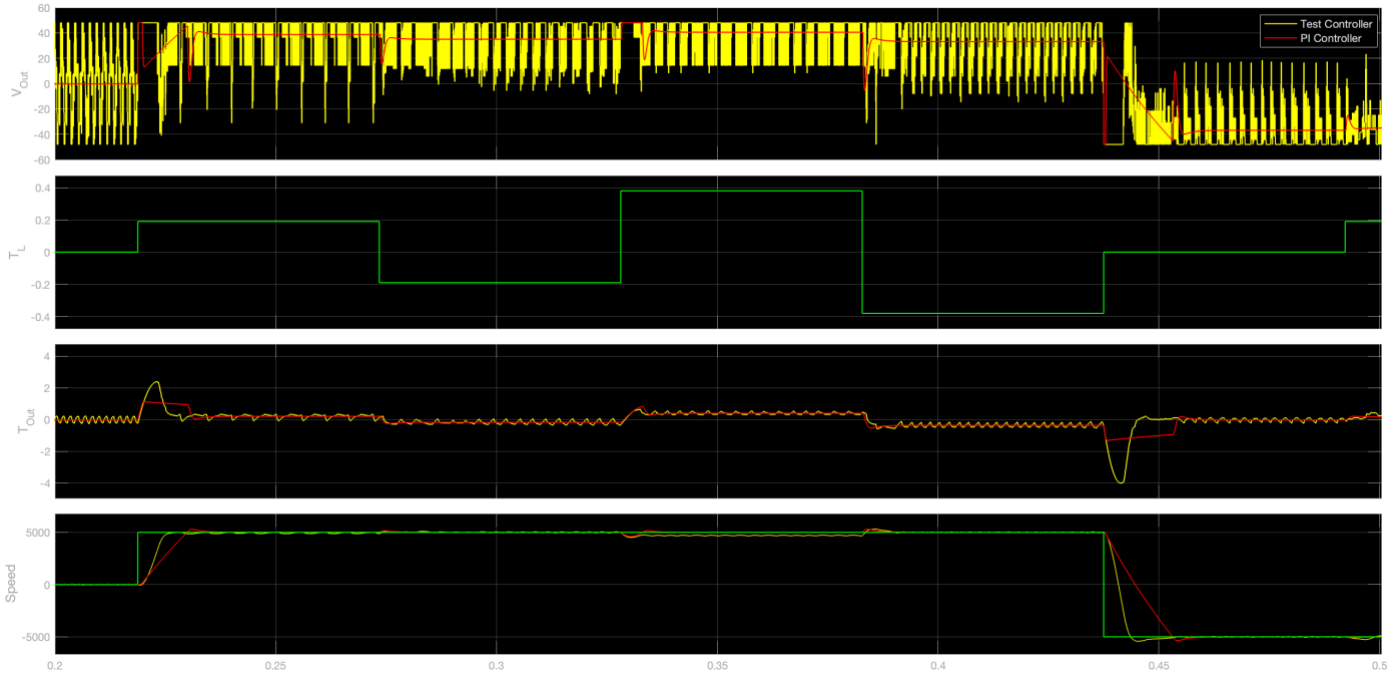


Figure 38 Testbench results of TD3 agent with 9-bit fixed-point representation of NN.

Testbench Results for Different Numerical Representations			
MODEL	CUMALATIVE SQUARED SPEED ERROR	CUMULATIVE SQUARED ENERGY	OBSERVATIONS
PI	0.06178	93.89	
TD3 (Double Precision Floating Point)	0.03864	122.3	
TD3 (18 Bit Fixed Point with optimised ranges)	0.03864	122.3	Same as TD3 with Double Precision
TD3 (9 Bit Fixed Point with optimised ranges)	0.04078	136.4	Much more oscillation in control voltage. Leads to oscillations in speed tracking and therefore greater error and actuator effort

HDL Code Generation Report

After the model has been converted to fixed point, HDL coder can be used to generate HDL, simply by right clicking the network subsystem and selecting "Convert Subsystem to HDL". This will generate Verilog files for each of the subsystems within the model and will create a series of linked HTML files which form a "Code Generation Report".

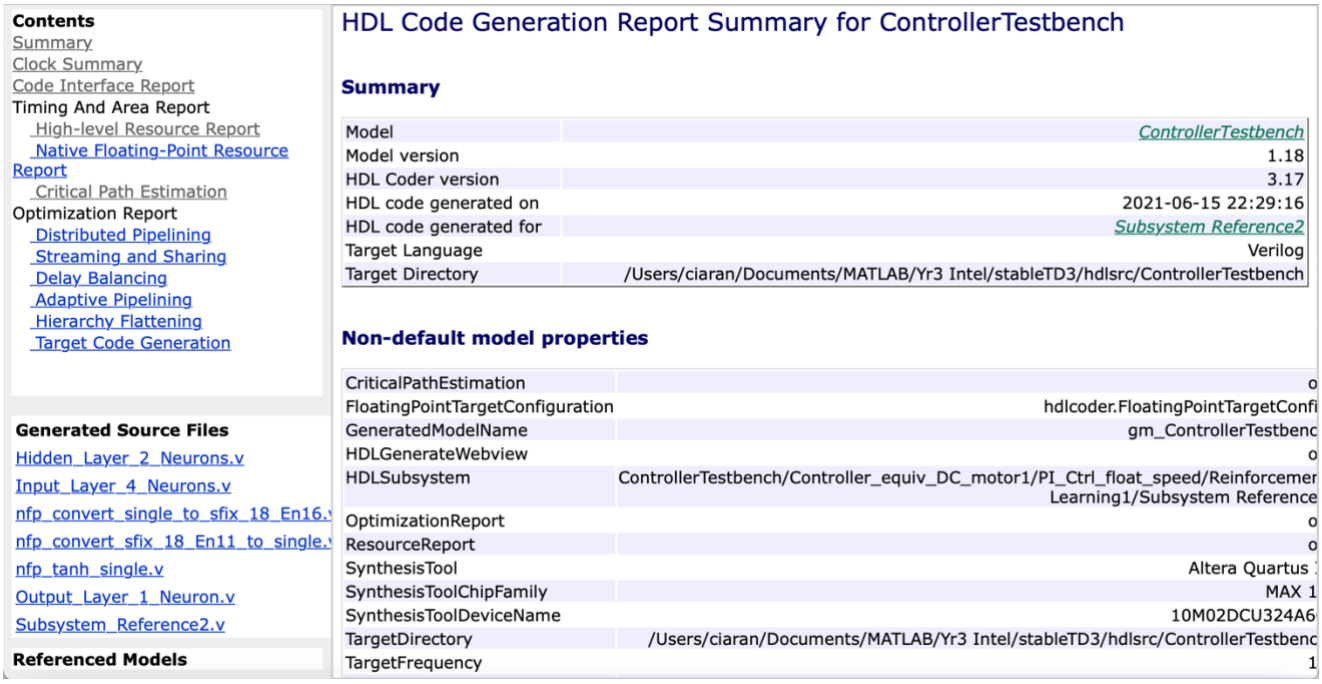


Figure 39 Code Generation Report for 18-Bit Fixed Point TD3 Agent

This gives an overview of resource usage and clock/timing information when the HDL is deployed to an FPGA.

File: Hidden_Layer_2_Neurons.v

```
// -------------------------------------------------------------
//
// File Name: /Users/ciaran/Documents/MATLAB/Yr3 Intel/stableTD3/hdlsrc/ControllerTestbench/ControllerTestbench/H
// Created: 2021-06-15 22:29:09
//
// Generated by MATLAB 9.9 and HDL Coder 3.17
//
// -------------------------------------------------------------


// -------------------------------------------------------------
//
// Module: Hidden_Layer_2_Neurons
// Source Path: ControllerTestbench/Controller_equiv_DC_motor1/PI_Ctrl_float_speed/Reinforcement Learning1/Subsys
// Reference2/Hidden Layer (2 Neurons
// Hierarchy Level: 1
//
// -------------------------------------------------------------

`timescale 1 ns / 1 ns

module Hidden_Layer_2_Neurons
          (clk,
           reset,
           enb,
           layer_input_0,
           layer_input_1,
           layer_input_2,
           layer_input_3,
           weights_matrix_0,
           weights_matrix_1,
           weights_matrix_2,
           weights_matrix_3,
           weights_matrix_4,
           weights_matrix_5,
           weights_matrix_6,
           weights_matrix_7,
           bias_vector_0,
           bias_vector_1,
```

Figure 40 Verilog Code for Hidden 2-Neuron Hidden Layer.

An interesting comparison can be made between the 18-Bit and 9-Bit fixed point representations. As far as HDL coder is concerned, the resource usage is almost identical apart from 1-Bit registers. This could be due to HDL coder not being capable of making smart optimisations. For example, sharing a single 18 bit multiplier to perform two 9-bit multiplications.

Multipliers	49
Adders/Subtractors	152
Registers	724
Total 1-Bit Registers	10986
RAMs	0
Multiplexers	292
I/O Bits	184
Static Shift operators	51
Dynamic Shift operators	7

Figure 41 FPGA Estimated Resources, 18-Bit Fixed Point

Multipliers	49
Adders/Subtractors	152
Registers	724
Total 1-Bit Registers	9204
RAMs	0
Multiplexers	292
I/O Bits	94
Static Shift operators	51
Dynamic Shift operators	7

Figure 42 FPGA Estimated Resources, 9-Bit Fixed Point

After generating HDL code, it should be possible to debug and verify it using the HDL Verifier Tool. Deploying to an FPGA using FPGA-in-the-loop is the natural next step in the process. However, we were

unable to take this forward in the remaining time as the HDL-Verifier Tool requires a fully licensed installation of ModelSim.

2.6 Further work

We believe there are still a lot of further steps that could be investigated with potential to improve on the speed, efficiency, and utility of motor controllers.

As the agent interacts with a system that involves delays and inertia, the introduction of recurrent neural networks should allow for a system memory. Our early LSTM tests suggested an improved steady state error, but longer training times and more hyperparameter choices meant that we did not find a better solution involving RNNs.

The non-deterministic aspect of training SAC and PPO agents is known to improve the exploration of the agent during training and in our experience, we have found these agents more successful at escaping local minima.

On-line agents like PPO were found to take longer to converge to a good solution but potentially allow for the network weights to be fine-tuned while the controller is running on the FPGA.

We have found that increasing the network size resulted in a longer convergence time so we focussed on finding hyperparameters for a smaller network, but the added neurons would allow for the network to approximate more complex non-linear functions in order to solve tougher optimization problems involving factors such as the reducing the energy consumption of the motor.

A larger network may also be needed when applying this solution to more realistic simulation models and the more complex 3-phase motor which is commonly used in industry.

Once the size of the networks has been investigated against the hardware limits of FPGA there is potential for multiple controllers to be deployed on the same FPGA, allowing for multi-axis control.
Additionally, our early tests have found the networks can be trained to achieve similar performance when simply outputting a positive or negative maximum voltage. This actor could then be deployed on the FPGA and run at a higher clock frequency as it eliminates the need for the slower PWM, allowing for even faster control.

3 Materials

3.1 Useful Resources

RL Agents:

TD3

Documentation:

https://uk.mathworks.com/help/reinforcement-learning/ug/td3-agents.html

https://uk.mathworks.com/help/reinforcement-learning/ref/rltd3agent.html

https://uk.mathworks.com/help/reinforcement-learning/ref/rltd3agentoptions.html

Motor Example:

https://uk.mathworks.com/videos/reinforcement-learning-for-field-oriented-control-of-a-permanent-magnet-synchronous-motor-1587727861081.html

SAC

Documentation:

https://uk.mathworks.com/help/reinforcement-learning/ug/sac-agents.html

https://uk.mathworks.com/help/reinforcement-learning/ref/rlsacagent.html

https://uk.mathworks.com/help/reinforcement-learning/ref/rlsacagentoptions.html

PPO

Documentation:

https://uk.mathworks.com/help/reinforcement-learning/ug/ppo-agents.html

https://uk.mathworks.com/help/reinforcement-learning/ref/rlppoagent.html

https://uk.mathworks.com/help/reinforcement-learning/ref/rlppoagentoptions.html

Useful Paper

On-line Reinforcement Learning for Nonlinear Motion Control: Quadratic and Non-Quadratic Reward Functions (Jan-Maarten Engel and Robert Babuska)
https://folk.ntnu.no/skoge/prost/proceedings/ifac2014/media/files/2042.pdf

Video Resources

MathWorks + Brian Douglas (reinforcement learning video series):
https://uk.mathworks.com/videos/reinforcement-learning-part-1-what-is-reinforcement-learning-1551974943006.html
Steve Brunton (Reinforcement learning for control):
Reinforcement Learning: Machine Learning Meets Control Theory

3.2 Our Code and Simulink Models

https://github.com/LudwigAJ/Intel_Industrial_AI

4 Commercial Considerations

4.1 Cost

To create a low-cost solution for industrial customers, who would likely need purchase FPGAs in bulk, we targeted the Max 10 FPGAs, which start at around $16.74.

Customers would also have to purchase a $3,995 yearly subscription to Quartus Prime to access the neural-network design if they do not already subscribe to it.

The combination of a product and subscription business model allows Intel to profit off this product in the long term, while only putting in the added effort of making the design available on their IP suite.

To continue to improve this solution, see the Further steps recommended in section 2.6.

4.2 Ethics

Safe control needs to be ensured with the neural network as a "black box" system that has no guarantee of stability. Since the inputs and outputs are continuous it is difficult to confirm that the system will not become unstable for some values. This could lead to unpredictable behaviour of the motor that has potential to disrupt manufacturing processes and in the worst case could harm humans and machinery. Already we have included limits to keep the output within a safer operating range.

Additional safety measures could include a separate system monitoring the speed error for signs of divergence with the ability to switch to a stable PI controller or to stop motor operation completely. Ultimately, the safety of the system would most likely need to be proven through rigorous empirical testing. This would also certainly involve testing the design in the real world on FPGAs before commercial deployment.

Provided the controller is stable, however, faster control could potentially enable higher levels of safety within an industrial setting. For instance, if a robot using a position controller were able to react to its environment more quickly, it may be able to reduce the risk of harm being caused to humans working in its vicinity.

4.3 Sustainability

The goals of this project were to make a faster and more energy-efficient controller for an electric motor. Electric motors account for 47% of the world's energy consumption [1] therefore even small reductions in energy consumption for motors will help reduce emissions in an effort against global warming.

Faster control (shorter rise time) often requires more energy. This is because to maximise speed, the output voltage is driven to its limits, which leads to slightly higher power usage. This meant that our best models had more energy loss during test bench simulation than the traditional PI controller but had faster rise times [see 2.4].

Despite this, the faster response times could have other effects that would mitigate the disparity in energy usage. For instance, faster control on a manufacturing line could enable higher productivity, which would increase the efficiency of other aspects of the manufacturing process.

When running larger Machine Learning models, an FPGA also draws less energy than what a GPU would have done. This in turn helps preserve energy uses. They also are able to have many different designs running at once, meaning that you can use less chips for more work. They are also generally designed to last longer in service than most other GPUs might.[2] Furthermore, the reprogrammable nature of FPGAs mean that the chips can be re-used for other applications even after the motor it was controlling is decommissioned.

Though we have been focussed on creating a faster control method, it's plausible that with more fine-tuning of the reward function, a controller could be created that achieves lower energy consumption than traditional controllers with a similar speed performance.

Also, by training our networks in a simulated environment, we can speed up training and therefore minimise time and material costs.

Sources:
[1] European Commission 2021, https://ec.europa.eu/info/energy-climate-change-environment/standards-tools-and-labels/products-labelling-rules-and-requirements/energy-label-and-0ecodesign/energy-efficient-products/electric-motors_en
[2] Intel https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/fpga-gpu.html

5 Project Management Details

5.1 Meeting minutes record

Intel Meeting 1 - 30/04/21

Meeting #1 - Product introduction:

- Called Industrial AI because the group has a particular interest in finding novel AI applications for FPGAs for industrial customers.
- Existing solutions applying ML techniques are quite slow, however in this case we want to make something quick.
- Existing models use field-oriented control to maximise the torque for a given field strength.
- Outer controls calculate for two fields perpendicular to each other, instead of three phases separated by an angle of 120 degrees.
- Transforms are done on FPGA which enables them to be done much faster than in software.
- Decisions must be made about what parts of the design should be done in software vs hardware depending on the necessary update frequency.
- Imperial team will be provided with a simulation of the FPGA + motor kit.
- Customers are very interested in speed of control - the faster control algorithms can be implemented the better, as this allows for more precise control of things like robotics etc.
- The controller must allow for a motor to be set to a specific position / operation mode, but also must be able to compensate for sudden disturbances.
- Another primary performance metric for this controller is the power consumption - in this case a "better" controller could be one that implements the same level of control but with increased efficiency.
- This is not likely to be a deep learning project, as having many layers will likely be overly complex for an FPGA.
- This could largely be seen as an optimisation problem - there are plenty of optimisation algorithms available in MATLAB that could be let loose to solve these.
- Have a look at what Bosch or Siemens are doing, but they are largely using shallow multi-layer perceptrons.
- We may also want to use well-defined algorithms to calculate speed, or other mission-critical values.
- Third harmonic injection control algorithm enables greater performance from provided DC voltages.
- The model that intel have provided needs to run at a high frequency in order to enable the simulation of more low level componentry that operates at such frequencies.

Meeting Summary

Here are some notes from our first meeting:

- We did personal introductions – a good mix of AI and embedded programming skills in the Imperial team.
- We reviewed the project introductory slides and discussed how we can measure performance of the controller in principle for a learning algorithm.
- Aspects of the project that you might want to consider how to manage in your team: project management, simulation model, control algorithms, FPGA advantage (what algorithms can an FPGA do better than a microcontroller?)
- Tracking of tasks – email, spreadsheet, Trello, your choice

Initial project aims:

- Create simulations for realistic motor control (may need to include power dissipation in diodes and MOSFETs) with control algorithm and measurement of performance (position control deviation, energy dissipation)
- Initially we may want to keep the model as simple and fast to run as possible while you develop the optimization process
- Brainstorm and try new algorithms, maybe create at least 3 styles of algorithm to take forward
- Optimize Algorithms
- Report on performance and limitations
- Review to decide on next priorities – alternative algorithms or focus on code generation

This week's tasks:

- MATLAB installation:
    - MATLAB Release 2020b or 2021a including
    - Simulink (required)
    - Simscape + SimElectronics (or SimPowerSystems) – may be needed later for more advanced simulations of power dissipation (type 'ver' at the MATLAB command line to see what you have installed)

- Read the references from the project description slides.
- Read up on the multi-layer perceptron neural network – how to choose the size and structure and how to train it.
- Try to run motor control model shared by Intel in MATLAB/Simscape
- Action: Intel Ben/Jakub to provide the model.

Intel Meeting 2 - 7/05/21

- Research replacing a PI controller with a neural network
- MATLAB Simulink embedded coder
- Model updates every 62.5 microseconds, chosen to match the 16 kHz used by the real motors.
- Use units in the signal name in Simulink
- We want to start by replacing the speed PI controller.
- U, v, w == a, b, c
- Use worst-case deviation in the speed controller otherwise use rms speed control error to measure performance to begin with, this could then be moved onto a model that tries to minimise power.
- There will be a speed command signal coming in, that we can compare against to obtain the speed error.

Provided Motor Model:

- <<intel_motor_control.mdl>>
- Created specially for the initial part of this project.
- Reduced the 3-phase motor to an equivalent DC motor so we have fewer signals and do not need to worry about the various transforms for a 3-phase motor.
- The model does speed control instead of position control, again to make the model simpler.
- Verified that the motor model achieves the operating point in our known Tamagawa motor datasheet.
- PI controllers tuned to give a good response for a given speed command, including the operating point of 5000 rpm.
- The controller works with 'non dimensional' ('ND') signals so it is easier to tune, and hopefully easier to replace with an AI equivalent. The output of the controller is therefore a maximum of '1', which is multiplied by the available voltage.
- The simulation starts by spinning the motor up to 5000 rpm and then introduces the motor's rated load torque suddenly and the controller has to respond to maintain the speed.
- You can toggle the switch to try open-loop control with a fixed voltage instead.
- Performance measures added in the form of squared and integrated speed error and motor current. A good controller should try to reduce the final values of both quantities, **with the speed error being most important to reduce.**
- **Model to be demonstrated and explained in detail in Friday's meeting (14/05/21)**

Intel Meeting 3 - 14/05/21

Meeting #3, Notes with reference to the Intel Motor Model

- Re-scaling of signals before the controller:
- Trying to ensure that the controller is as generic as possible.
- Command value = speed_cmd
    - All divided by rated value to limit range to -1 to 1
- Feedback value = delay_speed_measure
    - Converted to rpm and divided by rated rpm
- Delays have been added to make the simulation more realistic - it takes time to measure something, and it takes time to implement control, these values may need to be tweaked.
    - Current is too fast to measure -> Z^{-1}
    - Speed needs at least two consecutive position measurements, so it is particularly slow -> Z^{-2} (two time steps)
    - In this case a quadrature encoder is used to produced position measurements, i.e a sequence of 00, 01, 11, 10 is used, and the rate at which the motor moves through the steps gives a speed measurement.

The Controller:

- Internally there are two very similar PI controllers - one for speed, and one for current.
- Kp = proportional gain, Ki = integral gain
- There is a limit on the error size that can be passed as input, and a limit on the current that can be produced as an output.
- The speed controller observes the requested speed, the speed error, and produces an output that can be considered as a 'torque request' that is passed to the current controller.
- Ben has hand-tuned the PI controllers:
    - Set integral gain to 0
    - Adjust proportional gain until it can get very close to 0 error without oscillating or anything.
    - Keep adding integral gain until it can remove the last bit of error not too long after.

- Turning on the motor with the PI control has a few effects observable in the scope.
    - Initially the voltage saturates, as the speed error is very large.
    - This produces a large torque that then brings the speed to the desired value quickly.
    - Furthermore, following the sudden application of the rated torque, the PI controllers are able to bring the motor back to the original speed, as opposed the slightly lower equilibrium value achieved with open-loop control.

- Two primary metrics are available to measure performance:
    - A cumulative squared error measure
    - A total energy measure

- **Action:** Try out a very simple neural network by hand, i.e. Hand adjust the weights until something kind of works.

Intel Meeting 4 - 21/05/21

Meeting #4 - Demonstration of Initial Concepts:

- Demonstrated the models that we tested out with a PowerPoint. (Slides below)
- Discussed how the simple neural network with randomly selected weights tended to result in a low error **that was only specific to the original test scenario.**
    - It definitely makes more sense to train the network for a range of different speed commands and load torques.
    - NB: Torque values can also be negative, likely up to the same value, so it may be wise to add some values that account for that.
    - This could also highlight any errors that result from the simulation of friction.
- The main concern with our existing networks is the fact that they fail to remove a steady state error.
    - A few options were discussed here:
    - It seems from the RNN model that by introducing some form of memory, the controller is able to handle steady state errors better - i.e. Memory allows the controller to approximate the integral controller.
- Have a look at the multilayer shallow neural network built into MATLAB
    - https://www.mathworks.com/help/deeplearning/ug/design-neural-network-predictive-controller-in-simulink.html?s_tid=srchtitle
- Major takeaway points are that we should:
    - Find a way to ensure minimal steady state error.
    - Investigate methods for converting our network architectures into HDL, either using HDL coder or DSP Builder
    - See how well these solutions work on the three-phase motor

Meeting Summary (from Ben)

- You showed three solutions:
    - 3-layer fully connected neural net, 4 input including bias, 3 neurons followed by 2 neurons followed by 1 neuron, repeating the simulation with many random weights and choosing the best outcome.
        - Around 20 parameters needed for this structure
    - 3 layer RNN with tapped delays in the middle, optimizing by using script to apply random weights and repeat.
    - 3-layer, 7 input including energy, using actor-critic reinforcement learning

- You made some good changes to the simulation to give better control across the operating range:
    - Changed command so it changes in multiple steps, positive and negative.
    - Changed load so it changes between different values similarly, but all positive – suggest you introduce negative loads as well similarly.

- You noticed that NN solutions so far do a good job of getting fast transient corrections (you are already beating the original controller error metrics based on this) but do not eliminate steady-state error.

- Ideas to deal with steady-state speed error:
    - Try adding several previous error values with different delays into the algorithm to stimulate 'integral-like' response
    - Try changing the measurement error metric to be nonlinear, e.g. a tolerance band of +/- 1 rpm with no cost and a fixed cost for all results outside this band.

- You noted that reinforcement learning is working but has not 'converged' yet so there seems to be room for improvement, but the algorithm takes a long time to run.

- Riri suggests looking at multi-layer shallow NN example in MATLAB
    - https://www.mathworks.com/help/deeplearning/ug/design-neural-network-predictive-controller-in-simulink.html?s_tid=srchtitle

- You have based some work on a MATLAB example that applied an NN to current control of a 3-phase motor, suggesting the NN and training should be able to work with the Intel 3-phase model.

- **Action**: Ben and Jakub to provide 3-phase model for next week, and more realistic measurements for position and voltage.
- **Action**: Imperial (Jacob) to share Imperial models with Intel via Teams or emailed .zip.

Intel Meeting 5 - 28/05/21

Relevant points for creating a leaflet:

- Everyone wants their production line to go faster.
    - Speed of movement is therefore essential in maximising output in an industrial setting.
    - Faster control is an excellent way to increases precision as well, so it ensures that faster movements can be implemented safely.
- Energy efficiency is a good thing to focus on especially for something that is going to be on for continuous periods of time.
    - Some very high percentage of all energy dissipation occurs in motors.
        - There are about 8 billion electric motors in use in the EU, consuming nearly 50% of the electricity EU produces.

- The target FPGA is the Intel Max 10
    - This is one of the lowest cost FPGAs offered by Intel, and as it has to compete with existing microcontrollers (e.g. Those offered by Texas Instruments) this would be ideal for a simple motor controller.
    - For a more sophisticated system (e.g. An industrial robot) it may make more sense to target a device such as the Cyclone V SoC, which has a faster fabric, and could potentially control multiple different joints within a robot.

- Highlight how the use of AI is part of a wider movement in industry to adopt machine learning techniques to improve upon conventional ones.

- Target markets include:
    - Power generation - there are motors (though they operate as generators) in devices such as wind turbines.
    - Many cars have electric turbochargers in them now, to enable them to have a small (e.g. 1L) engine, that can still generate a decent amount of power.

- We're focusing on permanent magnet synchronous motors (PMSMs).
- There is an industry trend towards smaller and smaller motors, as by reducing the mass of motors, we can increase the efficiency.
    - The downside to this, is that in order to generate the same amount of power, we need the motor to move a lot faster, which therefore means we need faster control.
    - Faster control also equates to being able to handle much higher frequency disturbances.
    - FPGAs already offer improvements in Industrial Control applications, when compared to existing controllers.

- Existing applications of AI in these sorts of industrial settings so far have only really focused on much slower things such as fault detection. Actually, using AI on something very quick is quite novel, which will certainly make this product interesting to industrial customers.

- **Progress update and next steps:**

- Maybe consider a controller that outputs either an on or off signal, as opposed to a continuous voltage signal, as ultimately this device uses a PWM signal anyway.
- Taking several previous outputs as inputs is a good idea - the right number will have to figured out experimentally.
- Make the commands and disturbances a bit more randomised.

Meeting Summary (from Ben)

In today's meeting, we covered a few points:

- Project marketing leaflet to be created for the end of the coming week: Ask Joshua Levine for feedback if you have something to share early next week. We thought of these points to highlight:
    - Market is higher performance (fast responding and energy efficient) motor controllers like industrial servo drives and robotics (see the attached slides) below.
    - The AI based control is expected to provide faster response to command changes and disturbances and more energy efficient.
    - Having an implementation of AI on FPGA enables future upgrades to new AI hardware designs that may improve performance further as AI improves, due to the field-programmable nature of FPGAs.
    - Target customers in the servo drive and robotics markets include ABB, Siemens, Bosch Rexroth, Rockwell Automation, Yaskawa robots, Panasonic servo drives.
    - <<Arrow EMEA Motor Control Tech Snack.pdf>>
    - <<IntelFPGA_ArrowTechLunch_motor_control_210519.pdf>>
- Technical Progress:

- You reported trying some new NN refinements like including the integral of error as input, or feedback values for several past steps.
- Also you tried the alternative performance measure of keeping the output in a tolerance band compared to totalling squared errors. With these measures you were able to reduce steady-state errors though the performance varied for different command inputs.

- For technical next steps, we suggested:

- Replace stepped command and disturbance signals by random values within the same range, to try to improve steady-state error for many command values.
- Remove need for PWM waveform generation by replacing the continuous control output by just two states, max voltage or 0 V.
    - You will need to feed back the current setting (or last few settings) of the output voltage (max DC or 0, or just 1/0) as input to the NN so the NN can make a good decision about whether to switch the output voltage or keep it the same.

- Further steps (needing some modification to the motor model) will be:

- Position control instead of speed control (try making a changing position command and using position feedback from the motor model)
- 3-phase instead of single-phase motor – we need to provide you with the equivalent 3-phase motor model (**Action**: Ben, Jakub)

- **Final Note:** Next meeting will occur on Monday 07/06/21, due to holidays.

Intel Meeting 6 - 07/06/21

- Discussed plans for the final weeks of the project and explained the main points for each of the different deadlines we'll be having to meet in the coming weeks. - see email to be sent by Ben.
- Proposed technical model improvements:
    - Replace stepped command and disturbance signal try random values within the same range, to try to improve the steady-state error for many command values. i.e. A sequence of more random steps more akin to white noise.
    - Freddie showed progress with implementing position control, we acknowledged that it will likely be necessary to account for the number of rotations in a particular direction.
- Things potentially worth including in documentation / notes.
    - Any research we've done into using neural networks for control
        - We discussed how RL does seem to be the only option to produce a robust replacement for a conventional controller.
        - Other examples include using neural networks to try and generate a model of the plant in a control problem, which can then have traditional control techniques applied to it.
        - Evaluation of the potential of using different actor structures to what we currently seem to have settled, e.g. Would it be possible to introduce ideas such as recurrency etc.

Meeting Summary (Ben's Email)

Plan for the remaining weeks of the project:

- Technical model improvements (set last time)
    - Replace stepped command and disturbance signals by random values within the same range, to try to improve steady-state error for many command values.
        - To try, week 7
        - For position control, keep commands within one revolution.
        - Remove need for PWM waveform generation by replacing the continuous control output by just two states, max voltage or 0 V.
        - You will need to feed back the current setting (or last few settings) of the output voltage (max DC or 0, or just 1/0) as input to the NN so the NN can make a good decision about whether to switch the output voltage or keep it the same.
            - **Done for speed control.**
        - Position control instead of speed control (try making a changing position command and using position feedback from the motor model)
            - **Done**
- Week 7:
    - (Stretch goal) 3-phase instead of single-phase motor – we need to provide you with the equivalent 3-phase motor model (**Action:** Ben, Jakub)
    - Settle on a preferred NN architecture and training process using toolboxes that Intel has available (Jakub and Riri to try on Intel side)
    - For at least one model, convert the controller subsystem to use standard Simulink blocks suitable for HDL coder
    - For at least one model, use HDL coder to generate HDL code for the controller subsystem.

- Week 8+:
    - Run the generated HDL controller simulation in ModelSim or with HDL Verifier
    - Run at least one model as HDL code on FPGA hardware, using SignalTap to view the output (may need Intel support for this - Jakub)

Intel Meeting 7 - 11/06/21

Notes on recent completed work (goals set previously):

- Replace stepped command and disturbance signals by random values within the same range, to try to improve steady-state error for many command values.
    - To try, week 7 – found it difficult to learn, maybe commands are changing too quickly for the controllers to respond to, ends up like noise around an average command. Found multiple steps more reliable, can extend simulation to include more steps as alternative.
    - For position control, keep commands within one revolution.
- Remove need for PWM waveform generation by replacing the continuous control output by just two states, max voltage or 0 V. You will need to feed back the current setting (or last few settings) of the output voltage (max DC or 0, or just 1/0) as input to the NN so the NN can make a good decision about whether to switch the output voltage or keep it the same.
    - **Done for speed control.**
    - **11th June: applied to position control. Trains quicker with 1/0 output and produces fast stable control but final position is noisier (small oscillations).**
- Position control instead of speed control (try making a changing position command and using position feedback from the motor model)
    - **Done**

Further notes on Week 7 progress:

- Settle on a preferred NN architecture and training process using toolboxes that Intel has available (Jakub and Riri to try on Intel side)
    - Freddie, Nikita and Ludwig have been working on enhancing Reinforcement Learning approach to include LSTM and gaussian noise into Actor. LSTM takes 16+ hours to train compared to ~2 hours for non-RNN
    - Actor is currently 3-layer fully connected, 8-9 feedback values into 64 neurons followed by 64 neurons followed by 16 (please correct me?). Results in ~10,000 tunable parameters.
- For at least one model, convert the controller subsystem to use standard Simulink blocks suitable for HDL coder.
    - Ciaran and Jacob worked on this, created Simulink matrix mult representation and Fixed Point Toolbox to optimize to around 18-bit numbers.
    - For at least one model, use HDL coder to generate HDL code for the controller subsystem.
    - Successfully generated HDL code targeted at MAX 10.

Plans for Week 8+

- Create presentation for 17th June. (10-11 am, Intel are invited)
- Provide HDL Coder report with MAX 10 target to show resources used
- Try halving the precision (or reducing by experiment) on fixed-point numerical formats to see when performance degrades noticeably in Simulation.
- Provide draft documentation from the project and conclusions on ‘best known methods’ so far
- Run the generated HDL controller simulation in ModelSim or with HDL Verifier
- 3-phase instead of single-phase motor – we need to provide you with the equivalent 3-phase motor model (Action Ben, Jakub)
- Run at least one model as HDL code on FPGA hardware, using SignalTap to view the output (may need Intel support for this - Jakub)

Intel Meeting 8 - 11/06/21

Plan for the remaining weeks of the project &
Notes on recent completed work:

Week 8 +

- Create presentation for 17th June. (10-11 am, Intel are invited).
    - **Done** – great presentation!
- Provide HDL Coder report with MAX 10 target to show resources used
    - Ciaran showed in the meeting – 11,000 registers, 49 multipliers in MAX 10, 18-bit design
- Ciaran/Jacob: Try halving the precision from 18 bits to 9 bit. (So far, seen 16 bit is OK). Record summary FPGA resource results for 18 bit and 9 bit for comparison. Record Simulink sim result for 18 bit and 9 bit for visual comparison.
- For Tuesday June 22nd: Provide draft documentation from the project and conclusions on 'best known methods' so far (models, method for using RL Toolbox to train the network etc). Condense Teams repository and share as a .zip file (if it gets > 20 MB, split into smaller sizes and email separately).
- Jacob: Run the generated HDL controller simulation in ModelSim or with HDL Verifier. (Apple MAC not compatible, will try if time allows).
- 3-phase instead of single-phase motor – we need to provide you with the equivalent 3-phase motor model (Action Ben, Jakub)
- Follow HDL Verifier flow to try to get physical MAX 10 FPGA into the loop – help Jakub to get Simulink model working that can generate HDL code and then work together to move through HDL Verifier workflow with Intel Quartus tools and FPGA board.
- Riri to present her results with back-propagation training method – move final meeting to Friday 25th, 4pm UK time. Also invite BU team (Ben to do).