Technical Note **LIGO-T2200196-v1** 2022/10/02

Developing Deep Learning Solutions for Lock Acquisition

Peter Ma — Mentor: Gabriele Vajente

1 Introduction

This project's primary goal is to investigate and develop deep learning techniques to approach the problem of LIGO's lock acquisition. Specifically, we will look into leveraging modern techniques in attention-based learning to help estimate the state of the mirrors given optical signals from the Power Recycled Michelson configuration (PRMI). We will also look into the usage of deep reinforcement techniques in controlling the mirrors directly.

In this paper, we report the project's progress and results. During the first 3 weeks of active research, we first replicated previous deep learning approaches using gated recurrent units (GRU). Secondly, we developed an attention-based state estimator using a transformer architecture. Thirdly, we developed a velocity-position uncertainty estimator in an attempt to use sensor fusion techniques to reconstruct the state of the mirrors. Furthermore, we ruled that the Encoder-Decoder model should be explored only when a working model has been established. Lastly, we implemented a Deep Deterministic Gradient Policy (DDGP) model for a reinforcement learning approach. We report that all three approaches main approaches (transformer model, position-velocity model and DDGP) have all been built and evaluated in respect to solving the control problem.

2 Problem Statement

There are two approaches to acquiring the lock for a given set of mirrors at LIGO. The safe approach is to learn an estimate of the state of the mirrors and the ambitious approach is to learn the controls directly to drive the motions close to 0. The first approach is considered sufficient since if we know the state of the mirrors, it is relatively easy to build these controllers. This is considered safer because the model only suggests additional information to the controllers, furthermore, it is more interpretable than the latter approach. The second approach removes any trace of interpretability and requires one to have faith in the model that it can do what it is set out to do and not break in production.

2.1 The State-Estimator Problem

We revisit the central issue in the state-estimator problem. What we want is, given a set of optical signals, we want to get out the relative positions of the mirrors. The problem is that there exists an infinite number of solutions where these positions satisfy the optical signals we inputted to the model. This is generally not a problem, any solution would work. The issue is, once a solution is chosen, we need to follow that solution over time to preserve the continuity of these trajectories. To simplify the nonuniqueness of the problem, we can ”wrap” the data to force the labels to have unique solutions during training.

To perform this wrapping technique we first need to understand the system we're working with. Recall that the PRCL and MICH are *sets of mirrors* and together form a system highlighted in

$$\begin{aligned}\phi_{MICH} &= k\delta L_{MICH} \pm \frac{\Omega}{c} L_{MICH} \\ \phi_{PRC} &= k\delta L_{PRC} \pm \frac{\Omega}{c} L_{PRC}\end{aligned}$$

$$\begin{aligned}r_{MICH} &= r_X t_{BS}^2 e^{i\phi_{MICH}} + r_Y r_{BS}^2 e^{-i\phi_{MICH}} \\ t_{MICH} &= t_{BS} r_{BS} \left(r_X e^{i\phi_{MICH}} - r_Y e^{-i\phi_{MICH}} \right) \\ \Psi_{PRC} &= \frac{t_{PR}}{1 - r_{PR} r_{MICH} e^{2i\phi_{PRC}}} \Psi_{IN} \\ \Psi_{REF} &= \frac{i r_{PR} - i t_{PR}^2 r_{MICH} e^{2i\phi_{PRC}}}{1 - r_{PR} r_{MICH} e^{2i\phi_{PRC}}} \Psi_{IN} \\ \Psi_{AP} &= \frac{i t_{MICH} t_{PR} e^{i\phi_{PRC}}}{1 - r_{PR} r_{MICH} e^{2i\phi_{PRC}}} \Psi_{IN}\end{aligned}$$

Figure 1: Ψ are functions of fields, however note that there are complex exponentials thus are periodic and is related to L_{MICH} and L_{PRCL}

figure 2. In this subsystem there are two relative lengths that we care about called L_{MICH}, L_{PRCL}. Algebraically they are computed as follows:

$$L_{MICH} = L_x - L_y \tag{1}$$

$$L_{PRCL} = L_{PR} + \frac{1}{2}(L_x + L_y) \tag{2}$$

These equation 1 can be physically interpreted as the path length difference between the short Michelson arms (between the beam splitter and the two input test masses) denoted by L_x, L_y for each arms. On the other hand the equation 2 denotes the path length of the power recycling mirrors L_{PR} and the average path length of the input test masses.

We can numerically recover the signals produced from the length L_{MICH} and L_{PRC} using a set of equations shown in 1. following equations 3 and equation 4, where k is the wave number and Ω is 2π times the frequency of the laser signal added ontop of the original input laser in the Pound-Drever-Hall sensing scheme, c is the speed of light:

$$\phi_{MICH} = k\delta L_{MICH} \pm \Omega/c L_{MICH} \tag{3}$$

$$\phi_{PRC} = k\delta L_{PRC} \pm \Omega/c L_{PRC} \tag{4}$$

With enough elbow grease one can reduce the field equation Ψ's such that they are related to L_{MICH} and L_{PRCL}. The issue is that leaving it in such a form it is difficult to retrieve the period of the signals analytically since there is a mix of terms with L_{PRC} and L_{MICH}. Thus to simplify this we can apply a linear transformation namely

$$Z_1 = 2(L_{PRCL} + \frac{L_{MICH}}{2})$$

$$Z_2 = 2(L_{PRCL} - \frac{L_{MICH}}{2})$$

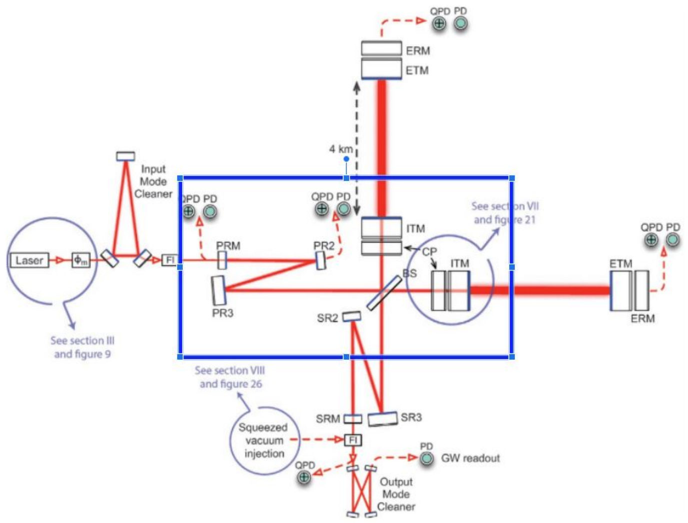

Figure 2: The rectangle boxed section is the PRCL MICH system that we're considering.

If we plug this transformation into the equations we retrieve a period of $\lambda/2$. With this transformations we can now wrap the data.

Firstly we will take the PRCL and MICH positions and follow the transformations described for Z_1 and Z_2. After that is done, we iterate through the data and linearly shift up or shift down by integer multiples of the period such that all positions are restricted between $\lambda/4 \rightarrow -\lambda/4$. Then we transform back returning the PRCL and MICH positions successfully wrapped. This helps guarantee that for every signal the corresponding wrapped position is unique!

This technique is a problem in production. When computing these we have to undo this preprocessing step and unwrap the data in a manner that retains continuity such that the reconstructions are accurate. We need to unwrap the data in order to recover the true positions. This is what we call the wrapping problem.

3 Baseline Model

Previous attempts in state estimation used gated recurrent units called GRUs [1] [2] to predict the wrapped positions. GRUs are a class of recurrent neural networks. Recurrent neural networks (RNN) are simply neural networks that work well with time-series data since have built in priors specifically a memory mechanism that exploits the sequential nature of the data. More concretely one to pass information from previous time steps to the next. Because of this, RNN's can be written with a recursive definition. At the first time step, we input data into a traditional neural network, then during the next time step, we input both the new data observed and the hidden state from the previous time step and continue until we reach the end of the sequence. Each set of feed forward networks for input data is called a cell or unit. This way we can pass information from previous observations to inform the model in the decision-making process. The classical

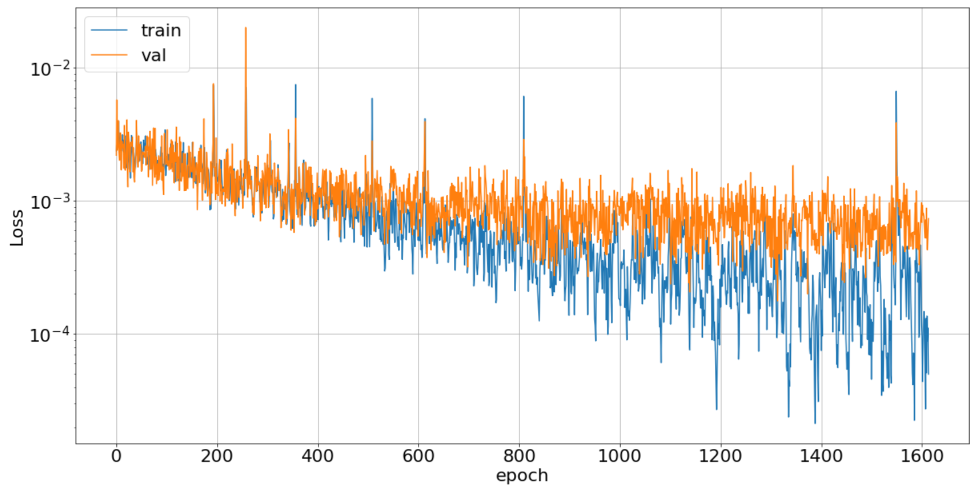

Figure 3: This is the loss curve for the model training.

RNN described suffers from a host of memory issues. With each passing of hidden states, data from the beginning may not be represented well further down the sequence, GRUs take the RNN idea further by developing specific gates within each cell that manipulate the hidden states that are passed down called the reset gate, and update gate[1]. The reset gate can selectively choose to nullify certain data that are passed from before, and the update gate decides what's important to pass on to the next state[2].

This technique was chosen because the data is time series and secondly the input data is relatively long on the order of 1000's input samples. This was the reasoning for this approach back in 2017. To that end, our first task was to replicate these results which we successfully did.

3.1 Methods

We implemented a GRU in KERAS framework[3]. The data used for training was simulated PRCL and MICH with approximately 40,000 samples each 0.5 seconds long sampled at 2048Hz. Given these optical signals, we then try to recreate a single-time step of position data.

The model architecture had 2 GRU layers, with 128 units each, followed by 4 fully connected layers of 4092, 512, 32, and 2 units respectively. Each layer used a ReLU activation except for the final layer which is given a linear activation. We used the Adaptive moment (AdaM) optimizer with a learning rate of 1e-3. The reconstructions are as follows in the following diagrams in figure 4[1].

This approach is insufficient because we cannot leave the data wrapped. This is because in reality the mirrors are free to move beyond the range which we restricted the solutions to. Thus to retrieve the true relative motions we must stitch back the reconstructions. Furthermore, when creating these shifts we artificially created instantaneous jumps in trajectory, which are unphysical and is entirely

[1]Code link

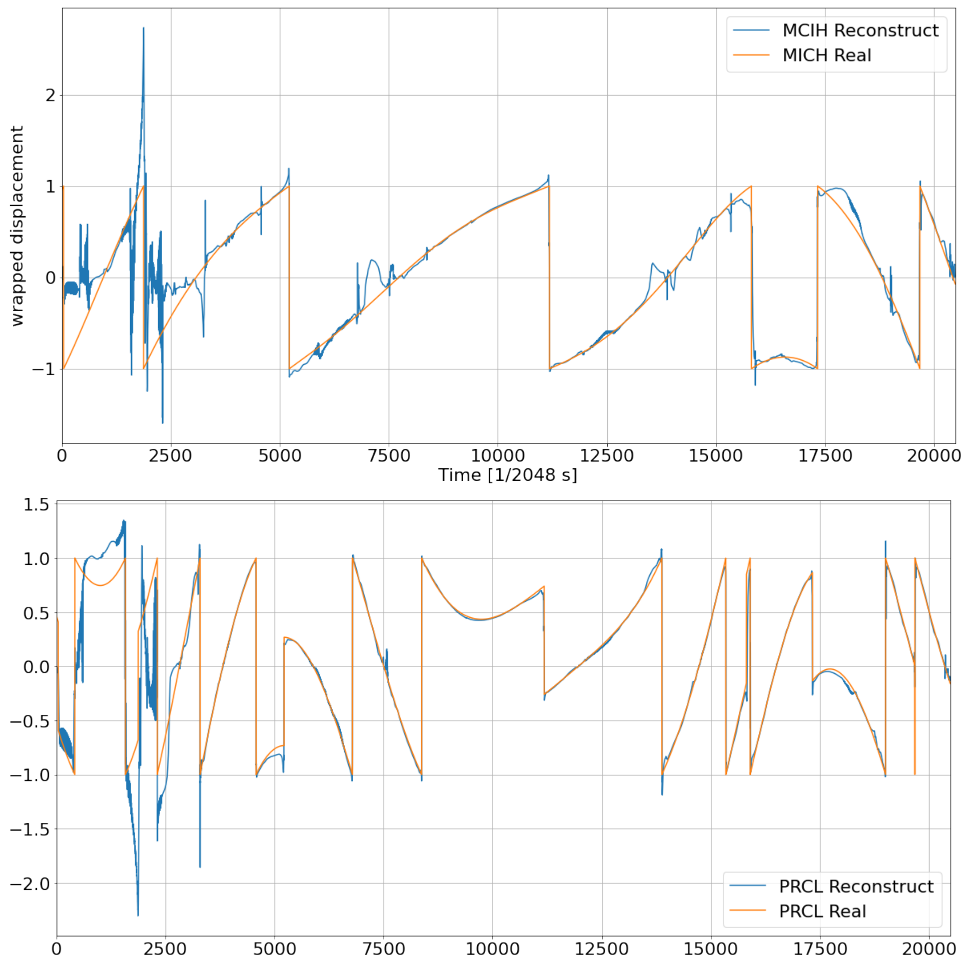

Figure 4: These are the reconstructions for PRCL and MICH.

the result of training the model on a training set where we linearly shifted the data down into the range of $\lambda/2 \rightarrow -\lambda/2$, where we know unique solutions exist. However, piecing together these solutions is a surprisingly formidable challenge.

To appreciate the problem, how one would typically approach unwrapping the data is to write a program that iterates through the reconstructions and detect sharp changes in position. Once the discontinuity is found we transform the data into the Z_1/Z_2 space and linearly adjust by integer multiples of $\lambda/2$ such that the discontinuity is minimized. The way in which we can detect these discontinuities is to look at the derivative as finite differences. The big problem is that neural networks are smooth mappings from input to outputs and in this case the outputs are also relatively smooth thus it makes recovering the original discontinuities difficult see, figure 4. Although it appears to be easy to detect by eye, to detect these sharp changes numerically is harder because numerically because they appear smoother and can be confused with imperfect reconstructions. Thus we're met with the wrapping problem.

4 Attention Based Estimators

Perhaps we can defeat the wrapping problem, by ignoring the wrapping problem? We approach the problem with *brute force*. We assume that given sufficient history of signals, we can hope to extract unique solutions for the input signal. The major challenge here is that even the GRU's used in the original implementation still fail to generalize to *very* long sequences of data. The problem of retaining memory of various parts of a long sequence is a well-known shortcoming of recurrent neural networks. Luckily in 2017, a novel approach in the domain of natural language processing sought the advancement of transformers and the use of self-attention which effectively solved this problem [4]. To leverage this technique we need we need to understand self attention.

4.1 Self Attention

In principle, self-attention is an intuitive concept[2]. Traditional attention helps assigns a weight to certain inputs of data as to how important they are to solving a particular problem. However there is an issue, we lack context. For example, in the sentence, “I picked up some cash from the bank and went for a swim at the bank of a river” the word bank has 2 different meanings. Humans interpret this by looking at the wo. To get machines to do the same is called self-attention. We look at the relationship of the word “bank” with the word “cash” and then we look at the relative attention of the words “bank” and “river” to infer these meanings. We look at the attention relative to words within the sentence itself, hence self-attention. We want this behaviour because we know in order to learn the dynamics historical information is important in informing the current state of the mirrors. Now this is superior to the RNN's memory mechanism because originally the constant capacity of the hidden state was an restriction to the amount of information that can be carried to sequential cells during inference. Self-attention mechanism avoids this by effectively ”remember-

[2]Good video on general attention here

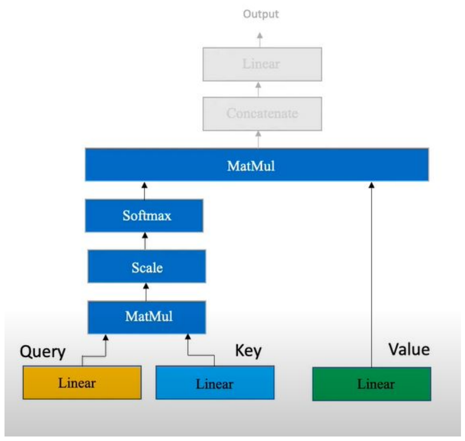

Figure 5: This is a concrete diagram showing the flow of data for self attention.[4]

ing everything". This magic is achieved by learning a continuous function that approximates a hash map of the input data called a "key". Now during inference time we can perform a soft search (looking for similar matches) against a "query" to retrieve an attention weighting. This "memory" scheme let us jump back and forth in time with no restriction by information flow.

Armed with intuition, we build such an algorithm [3][4]. We have three main elements, the KEY, the QUERY, and the VALUE each will be three separate linear layers 5. This is motivated by how retrieval systems work. We have a query, and we find the matching key and return the value. Attention is this search process. We only pay attention to the most similar items that we look for and disregard the rest. This suggests that we want to find the highest amount of similarity with itself as this is self-attention. Hench the KEY and the QUERY will have the same inputs to each linear layer. Concretely one finds similarity by looking at the cosine similarity. This is easily computed via the dot product. To do this with the matrices of the KEY and the QUERY, we simply matrix multiply the outputs of the linear layers. This encodes the ATTENTION FILTER. We then scale this layer and compute the softmax of this filter to arrive at numbers that sum to 1. Then we multiply this with the VALUE, which up until now is just the input passed through a linear layer. This matrix of numbers then helps us weigh and select the values from the original input and thus gives us the FILTERED VALUES. This process is called self-attention with a single head. If we want multiheaded self-attention, we repeat the entire process but with multiple layers and concatenate the final results together!

Attention works well, however, there is a key problem, these models don't consider ordering of the data. One of the transformer's great power is that it processes data in parallel instead of sequentially. This means that in the layer for self attention the aggregation is done on its own. More concretely, the attention values will always be the same no matter the order in which they appear. If we think about it, all the attention mechanism is doing is computing a weighted scalar between any pair of data in the sequence this scalar is created with 0 information about positions. However this would be a problem because in reality we know position affects how much attention is given

[3]tutorial on transformers KERAS here
[4]Very good Youtube video on self attention here

to each data point. Thus before the transformer step we need to encode the positions, [5].

To fix this, we apply a positional embedding of the input, whereby we take the embedded data (in NLP terms means converting text to vectors) we learn an embedding by adding vectors to the embedded word vector. This small offset helps encode information about the positional information of the vector WITHOUT changing the contextual representation of the input. For example, the appearance of the word "Queen" as the second word in the sentence should appear nearby in the vector space from other appearances of the word "Queen" but they all should nonetheless be slightly different from each other depending on where in the sentence they appear. However, they can't be so far apart such that the word "QUEEN" appears to be an entirely different world altogether! These "offsets" are what is learned by positional embeddings.

Here we arrive at the second major issue, positional embeddings work well for language tasks, however, general time series data are not words. NLP tasks involve discrete chunks of inputs, that are finite, for generic time series problems, we don't have that luxury as data is continuous and infinite. Hence positional embeddings become a problem. The solution involves representation learning with an algorithm called the TIME2VEC embedding technique[6] [5].

4.2 TIME2VEC

Time to vector is an attempt to embed temporal information in time-series data. Firstly there are 2 major requirements for this embedding[5].

1. 1. Embedded data should be invariant to time rescaling. In other words, if the data were taken at increments of 1 day or 2 months, or 3 picoseconds it shouldn't affect how the data appears in the end.
2. 2. Representations of time should both capture periodic patterns and nonperiodic patterns [5].

To satisfy these constraints we (of course very naturally) arrive at the following formulations[5]. The i is the index of the number of hidden dimensions (number of features to encode the data into), τ is the timeseries data itself. The w_i, φ_i are the learned parameters in which the

$$\mathbf{t2v}(\tau)[i] = \begin{cases} w_i\tau + \varphi_i, \text{if} i = 0, \\ \mathcal{F}(w_i\tau + \varphi_i), \text{if} 1 \leq i \leq k \end{cases} \tag{5}$$

I have yet to truly appreciate the mathematics of why this works but, concretely the first element represents the linear behaviors of the data and the second represents the periodic behaviors of the data where $\mathcal{F}$ is described as the a periodic activation function. Read more here [5].

[5] Good video on positional embeddings here

[6] Good Blog post on implementation of time2vec here

Armed with these new techniques, we approach defeating the wrapping problem.

4.3 Results Stage 1

In stage 1, we want to first and foremost, ensure that we can at least replicate the performance of the GRU on wrapped data with the transformers. We implemented the following model:

Time2Vec embedding Layer [no hyperparameters]

6 X Transformer Blocks [embed dim = 12, fully connected encoder =1024 , num heads=8]

5x Fully Connected [RELU, dims= 4096,800, 128, 32, 1]

Transformers can be difficult to train, as such we need a custom "warm-up" period whereby the learning rate is low but grows exponentially to the initialized learning rate. Then the learning rate decays for the remaining period until its reached a base learning rate. The warmup learning rate is$1 \times 10^{-6}$and the peak learning rate is 1×10^{-3} and the base learning rate is 1×10^{-6} the warmup time is 500 epochs[7].

Note: we were only able to achieve comparable performance with the GRU models if we trained separate neural networks for the PRCL and the MICH data.

Here are the reconstructions after training in 6

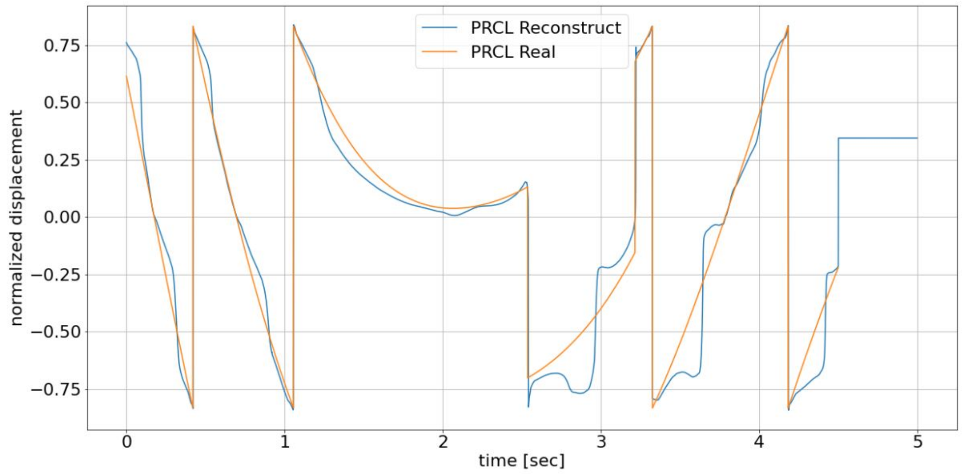

Figure 6: We see that the result is relatively similar and on par with the GRU model.

We note that the training loss on the same data had a factor of 2 mean squared error more than the traditional recurrent model. This model has approximately the same number of parameters as the GRU model about 12 million.

[7]code here

Now why does the transformer model perform worse than a GRU? Firstly transformers work well when there is an abundant amount of data. Transformers are ”data sponges”. The upside is that these models obey neural scaling laws [6] (more data, more compute $\Rightarrow$ almost guaranteed performance improvements) which the industry exploits heavily, such as large language models. However the downside is that when dealing with smaller models and smaller datasets the boost in performance over traditional models is no longer guaranteed. Why? Because the RNN’s priors are often efficient in producing reasonable performance with limited datasets, whereas a transformer needs to effectively learn these priors from scratch. For example, in our use case we needed to learn a time embedding for each input, meaning a transformer has to *learn* that its even dealing with time series data whereas the RNN starts of already exploiting that prior from the get go. In simpler terms, transformers just need a lot of data to ”get going” but once they do get ”going” the model continues to scale in performance whereas traditional models hit a wall. More concretely, we see a drop in performance because limitations in our data and compute to actually produce an edge over RNN’s. We conducted our experiments with the same dataset and approximately the same number of parameters. Nonetheless, we still carry on with the experiment.

4.4 Stage 2

In stage 2 we now try and and produce a transformer model that can reconstruct the positions of unwrapped data. However before we attempt such we first train and evaluate a baseline model using the GRU model on unwrapped data as a benchmark

4.4.1 Baseline GRU Model on Unwrapped Data

To be true to the problem at heart we use the same model design as with the wrapped data except we extend the number of timesteps the GRU sees during training. In this case we extend it to 5 seconds of data. This was chosen due to memory constraints on the GPU when comparing this with the transformer model later on. Transformer has more memory intensive operations and was thus a bottleneck.

We have made the assumption that given sufficient amount of data we can accurate reconstruct the unique positions despite the non-uniqueness of the signals corresponding to any given position.

Let us see how a GRU might perform against such a setup consider figure. 7

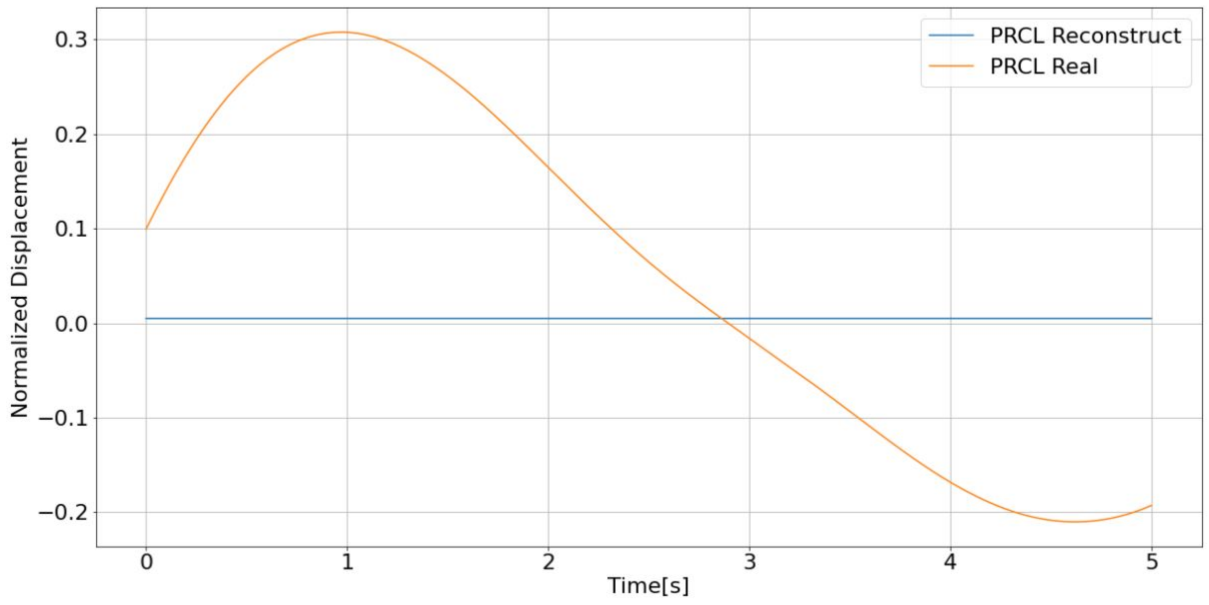

Figure 7: We see the predictions from the GRU is just a straight line.

The model was trained with early stopping with 300 epochs worth of patience. The model training only lasted 300 epochs since the loss does not chnage and stays at a constant 0.07 MSE.

We can directly conclude that the GRU is unable to produce the real unwrapped positions. There are two potential reasons: 1) the GRU suffers memory issues due to long sequences or 2) There isn't enough data to inform the model which solution to pick and thus can not solve the wrapping problem. We will investigate the issue by investigating the attention based model.

4.4.2 Attention Based Transformer Encoder Model

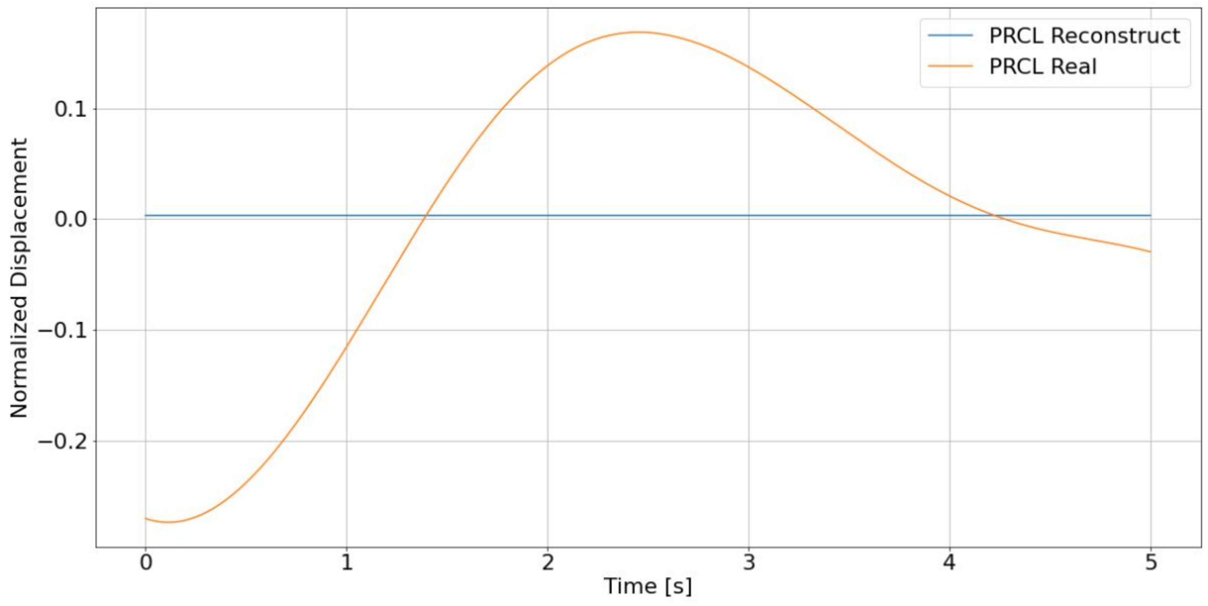

Figure 8: We are getting just a constant function once again.

We now take the same attention based model used for the wrapped data but instead we are now training it on unwrapped positions. Should our assumption be correct, that this is indeed a memory issue with the GRU, then an attention based Transformer should alleviate such a problem. Below we see a reconstruction after training it for 300 epochs with early stopping with a patience of 300 epochs. We once again note that the model produces a constant error of 0.06 MSE during training. Consider the reconstructions below in figure 8

We see that the model clearly fails at reconstructing the positions as it predicts a constant function which happens to be the average of the positions. We note that this is most likely not a failure with the algorithm but an issue with the data. If the problem was remotely solve-able with our technique it should not fail as spectacularly as it did here. In otherwords it should have learned something if at all. If it was a problem with the number of parameters or the number of layers etc then it should at least hint at learning something. When a model is a constant it hints that the problem might not be well defined for the approach or there is a fault in implementation. From previous implementation of the baseline test with wrapped data we ruled out the possibility of the latter.

4.4.3 Attention Based Transformer Encoder Model Part 2

Despite our unsuccessful attempt, we further pursue this approach with a slight twist. During the data generation process we restrict the initial start to be within the $\lambda/4 \rightarrow -\lambda/4$ range. The idea is that if we restrict the model to see data that start from the same interval range, the model can learn to reconstruct positions starting from the same initial state. This could potentially remove some of the ”confusion” during training.

We note the results below. First note the training loss curve is jagged this time rather than a constant, 9. This is in part due to the cycles of 10 epochs where we regenerate data. However we note that during each cycle the training loss remains relatively constant indicating its not learning anything.

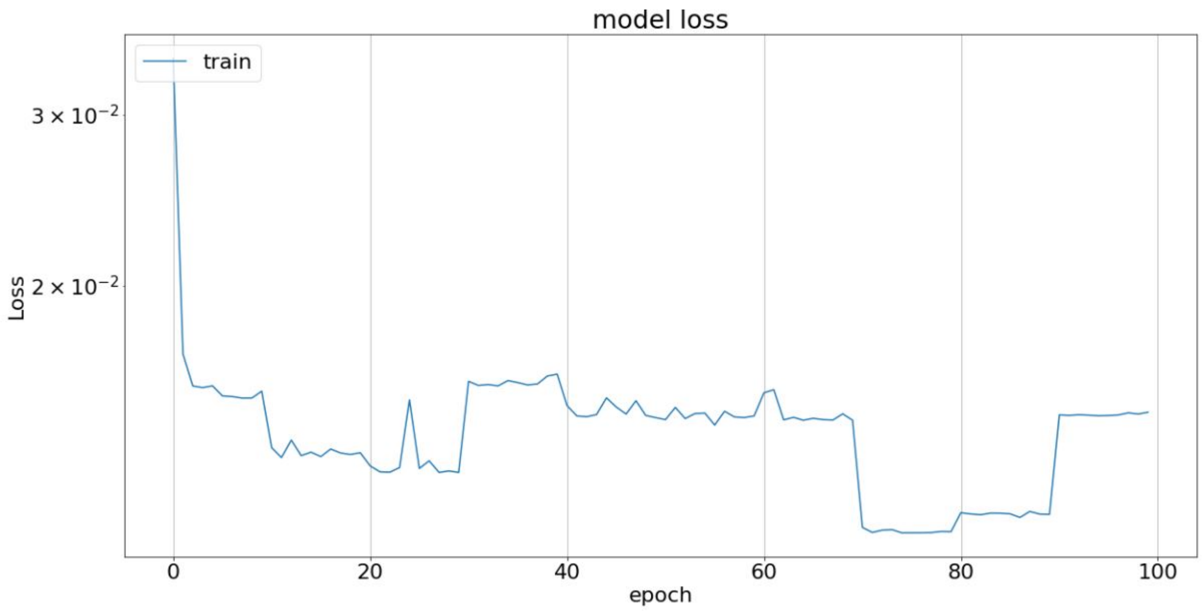

Figure 9: The loss curve for second transformer approach.

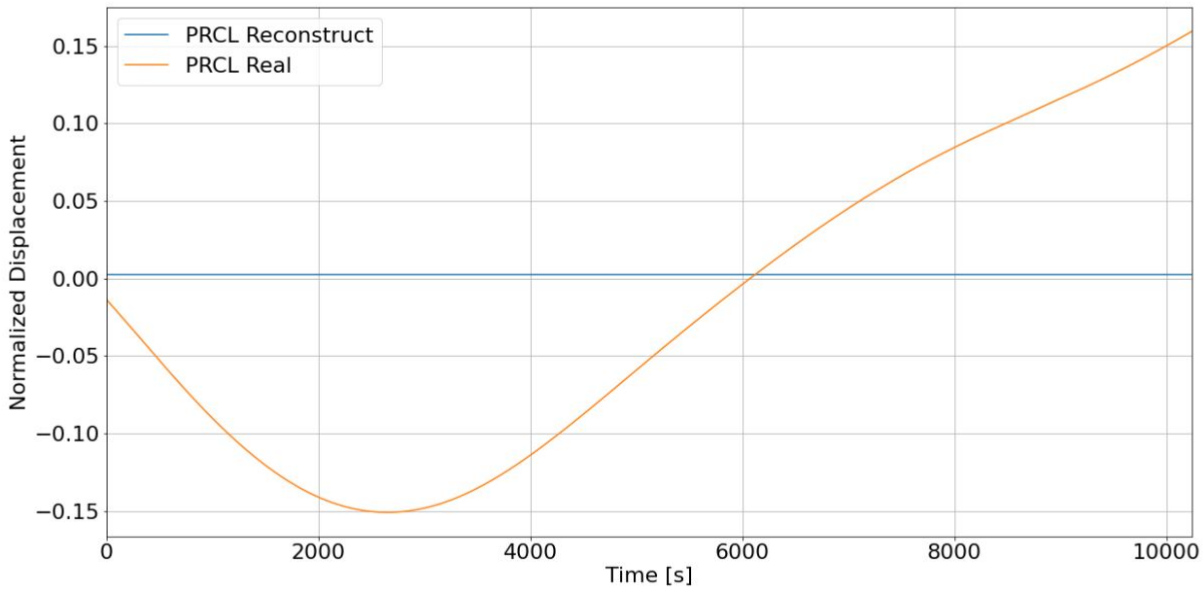

Figure 10: Reconstructions from the second transformer approach.

We once again are faced with the same situation, the model fails to learn meaningful representations of the data see the reconstruction in figure 10. This let us conclude that the attention based methods appear unfruitful from our investigation.

Note: this does not mean we've *proven* that attention based models will always fail for our problem, we just concluded that for this project this direction appears unfruitful and due to time constraints we decided to allocate time in other promising approaches.

4.5 Discussion On Attention Based Estimators

There are a number of reasons we decided to conclude this direction of investigation.

1. 5 - 10 seconds of historical signals appear insufficient in uniquely reconstructing the unwrapped position.

2. Computing constraints and production implementation challenges.

Firstly, we see from our investigation with 5-10 seconds of historical data we fail to reconstruct the positions in a satisfactory way. From our experiments we conclude that more historical data is needed to have any hope of solving the wrapping problem and even then there is no guarantee of success. This brings us to the computing constraints. In production we cannot expect to save all the historical data and recompute them on demand with every sequential update in timestep. Unlike the GRU where we can simply take the last hidden state and store it in memory and then recompute a single forward time step given the hidden state which is computationally cheap compared to running 6 transformer layers each new timestep. Beyond the implementation issues, even just training the model on the GPU is an issue as 5-10 seconds is the maximal limit of data we can compute. Note that 10 seconds of data is approximately 20,000 time steps. If this were a language task even for large language models this is rather unorthodox. In a language setting this is like getting a large language model to read a prompt that the size of 1/3 of Harry Potter's Philosopher's Stone and be able to produce meaningful results from it. Not surprised this approach wasn't fruitful yet.

5 Velocity Position Uncertainty Estimate

Having approached the problem with brute force as with attention, we concurrently investigate more elegant means of reconstructing the positions. In this approach, we accept our doomed fate and wrap the positions of the estimator. The trick is, that we know that the velocity of the mirrors is unique. Given this information, we attempt to recreate both the velocity and the position of the data. Together we can attempt to fuse these two data points together to reconstruct the accurate positions.[8]

[8]Please find the COMPLETE tutorial demonstrations of this section here `https://gitlab.com/gabrielevajente/prmi-ml/-/blob/main/tutorial/complete_tutorial.ipynb`

To build intuition, for a given position, the next forward position will have an infinite number of solutions. With the knowledge of the velocity estimate, we can use that to select a position (out of the infinite solutions) such that the new velocity has the least uncertainty. To accomplish this task we can use the Kalman Filter [7] to do such magic. Thus we reduced the problem to: “can we build neural networks that can predict the inherent uncertainty in the prediction”.

5.1 Probabilistic Models

We take inspiration from Variational Autoencoders[8]. Consider a classical feed-forward neural network for a regression problem. Now consider that the last layer splits into two layers which we denote as the mean and the other as the log variance. Then when optimizing the model, we compute the probability that the target appears in our predicted probability distribution and we try to maximize this value. Intuitively this makes sense in capturing the error of the estimator.If the model were poor at making predictions, the variance should be high, this way there is a higher random chance that the model predicts the correct solution. However, if the model is highly accurate, then the variance should drop and the mean should align with the target value this way there is a higher probability the predicted data points fall close to the target. Thus in theory as the model improves, the variance should drop and the mean should approach the target value. We can mathematically formulate this by using the PDF of a Gaussian given by:

$$P(y|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{1}{2\sigma^2}(-(y-\mu)^2)} \tag{6}$$

5.1.1 Position and Velocity Model

Let us take this probabilistic model and try to reproduce both the position and the velocity and their associated uncerainties. To do so we construct two models one for position and its uncertainty and similarly another for velocity. These models are identical in architecture.

The design is the exact same as the GRU baseline model except for the last layer where it splits into a mean and a standard deviation which is the error. The standard deviation uses ELU activation instead of RELU or linear.

We then construct a loss function which computes the negative log of the probability density function of a Gaussian given the mean and standard deviation produced by the model and the value x which is the actual position or velocity. For given position or velocity y, and where μ, σ are the associated mean and standard deviation, we want to maximize the likelihood. We search for θ which are the weights of the model.

$$\max_{\theta} P(y|\mu,\sigma^2)) \tag{7}$$

This is an equivalent optimization problem of maximizing the log likelihood which is the same as minimize the negative log - likelihood.

$$\min_{\theta}[-\ln(P(y|\mu,\sigma^2)))] \tag{8}$$

We implement this in the code as a VNET as this takes inspiration from variational autoencoders.[9]

5.1.2 Wrapping Positions Preprocessing

One issue we faced was correctly wrapping the data in a way that was consistent with the unwrapping method we are about to implement. Most notably, what linear shifts can we make to the PRCL and MICH positions such that the signals will be preserved? The method we sought to use is a simple transformation.

We know that the PRCL/MICH space, the field equations produces an oscillatory terms that contain a dependence on $PRCL \pm \frac{1}{2}MICH$. This makes finding solutions a bit cumbersome so we perform a change of variables stating $z_1 = PRCL + \frac{1}{2}MICH$ and $z_2 = PRCL - \frac{1}{2}MICH$ This produces the following linear map:

$$\varphi(PRCL, MICH) = \begin{bmatrix} 1 & \frac{1}{2} \\ 1 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} PRCL \\ MICH \end{bmatrix} \tag{9}$$

We also know that this oscillatory term has a period of $\frac{\lambda}{2}$. Thus we can directly wrap the data in the space bounded between $\lambda/4, -\lambda/4$. Then we can map back to the PRCL/MICH space after these transformations using φ^{-1}. The matrix is invertible from inspection.

[9] code: here

5.1.3 Position and Velocity Training Results

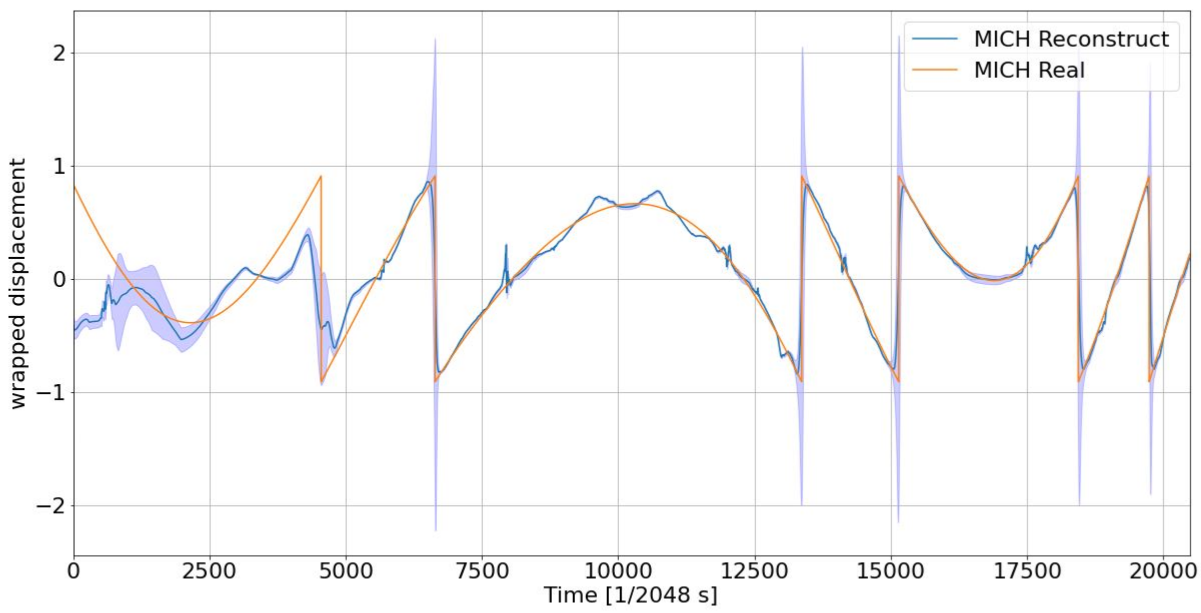

Figure 11: These are the MICH reconstructions the shaded is $\pm 3\sigma$ from the mean

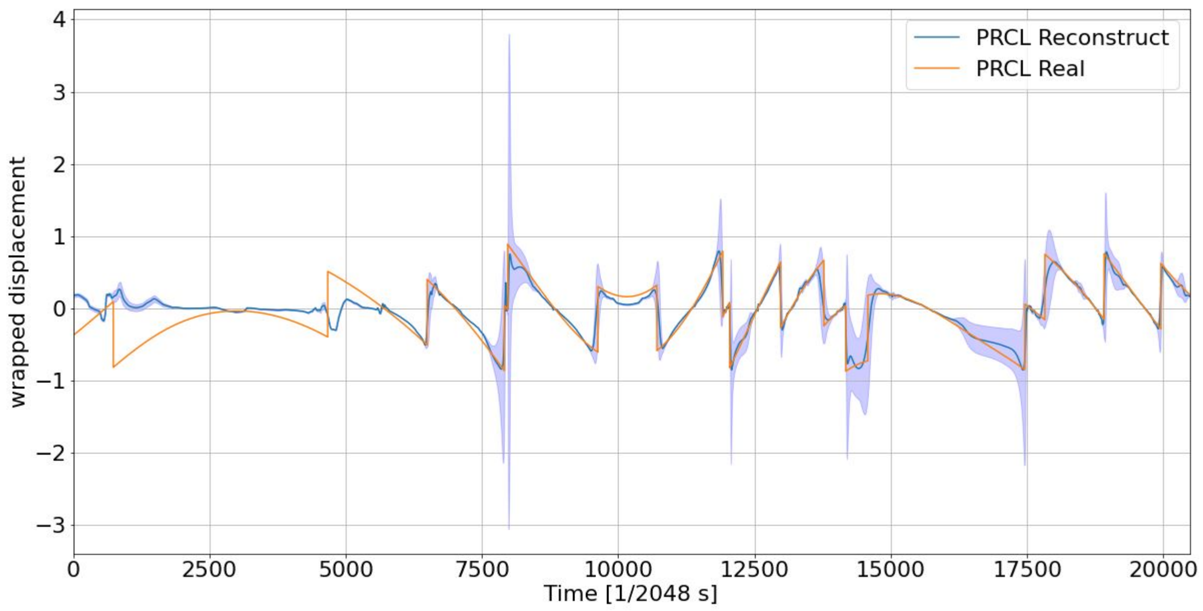

Figure 12: These are the PRCL reconstructions.the shaded is $\pm 3\sigma$ from the mean

We can see that this kind of construction is exactly what we wanted! The error rates are high in areas with the wrapping issue. The notebook that reproduces this is [10]

[10]Code for the Position Estimator Model

We repeat this for the velocities. We note that velocity does NOT suffer from the non-uniqueness and thus there is no need to wrap the velocities. Thus we proceed as usual and below is the training result and reconstructions. See figures 13 and 14.

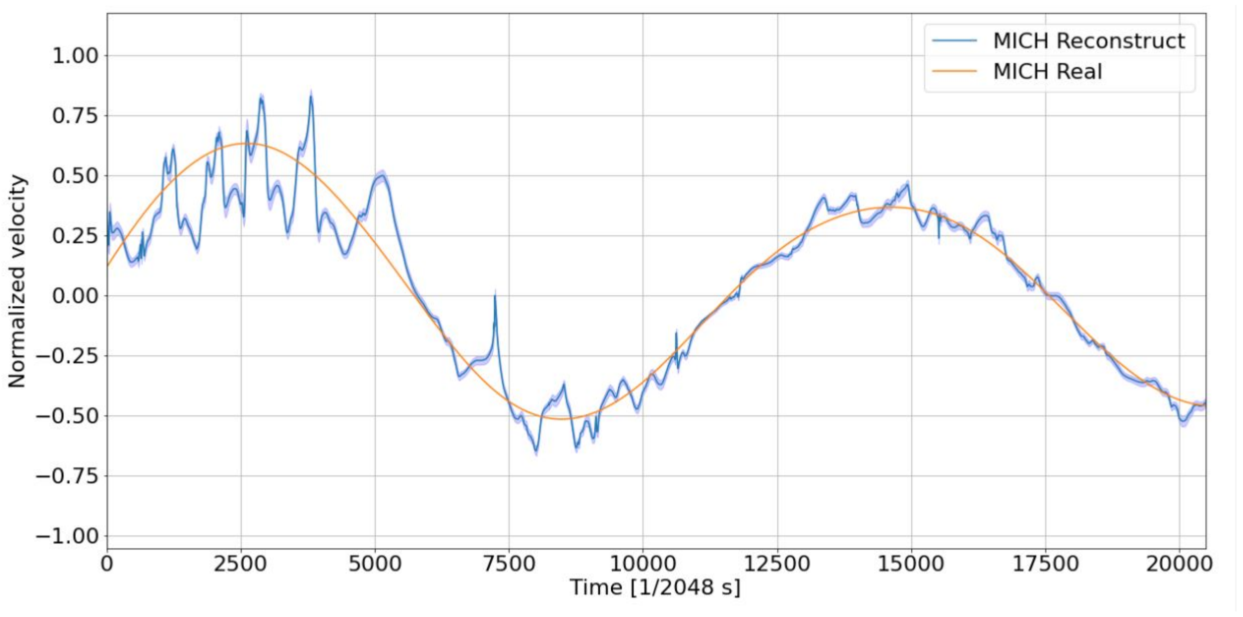

Figure 13: These are the MICH velocity reconstructions the shaded is $\pm 3\sigma$ from the mean

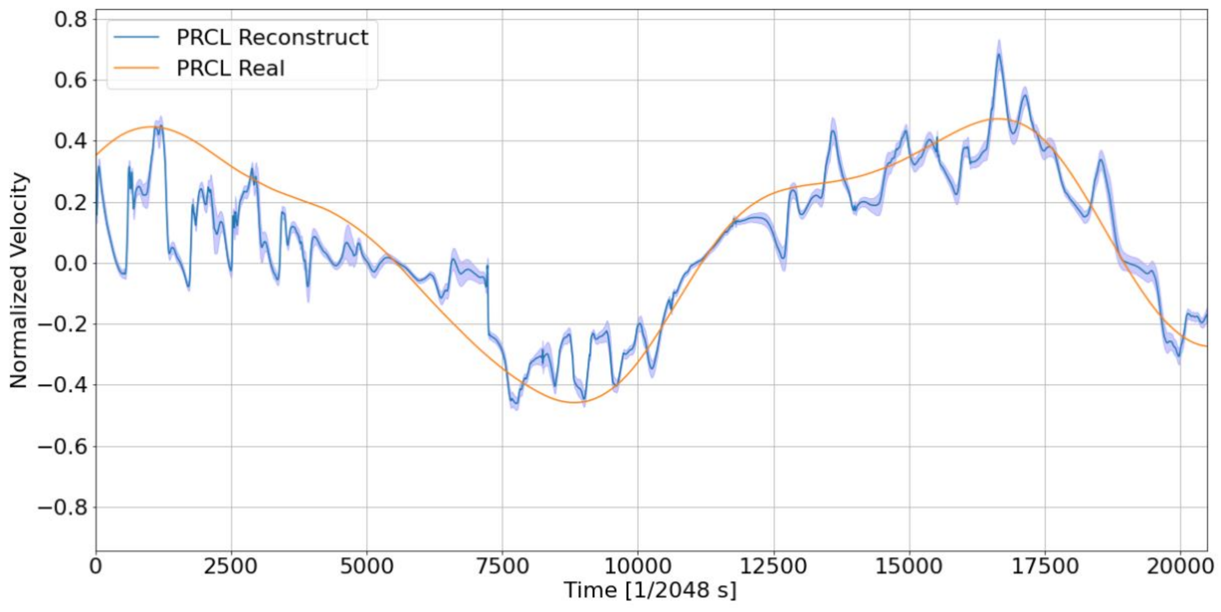

Figure 14: These are the PRCL velocity reconstructions. the shaded is $\pm 3\sigma$ from the mean

We note once again that these reconstructions appear relatively okay for our use case. Now the reason why we see that the uncertainty is smaller than what is expected could be attributed to overfitting. We noticed that when we added Gaussian noise to the signals on the order of 10^{-3} of the normalization factor the variance estimation became more reasonable however this led to issues in the accuracy of the mean prediction. This could be improved on by training it on larger datasets (we maxed out memory on our compute node with 100k samples). One possible explanation is that the velocity has more variability than the positions. The velocity is technically unbounded where as the positions are forced to be between a certain interval.

The implementation can be found here [11]

5.1.4 Kalman Filter

The last part of the puzzle is to implement the Kalman filter for our specific use case. As usual we begin with intuition. Firstly we have both velocity and position data and their associated uncertainties.

Now for every given new time step we can predict the wrapped position and we can predict its velocity. Now there are two things we can do with this information, first we can take the previous position and take the velocity and use simple to kinematics estimate the next position. Secondly we can also just get the next wrapped position, however to get back the unwrapped position we know there is an infinite number of these each adjusted by some multiple of the wavelength. As the usual wrapping problem states, which of these infinite solutions do we pick? Since we can use kinematics to estimate a forward position we can use this information to help us "select" one of the infinite solutions.

What we do is we can take the uncertainties to propagate forward the errors when constructing our kinematic solution for the next time step, this will be our prior distribution. Then we lay out all the possible solutions forming a Gaussian mixture from our next predicted wrapped position. Each of the solutions will be a candidate likelihood distribution. Now for each possible solution to pick from, we can obtain the corresponding posterior distribution by using Bayes Theorem. We simply multiplying the two distributions together. What we want to pick is the likelihood distribution that gives us the maximum posterior probability. This posterior distribution with the highest peak will be the best fitted position estimate!

Let us formalize this in an algorithm, please see pseudo code 1 for the structure of the algorithm. Let us define the model $p(\theta_p, x_t)$ for the position and $v(\theta_v, x_t)$ for velocity where the θ are the respective weights. This gives us then the mean and covariance μ_{p_t}, Σ_{p_t} and μ_{v_t}, Σ_{v_t} where the subscript denotes either position or velocity at times t. x_t is the optical signals received at time step t. θ is the weights. The total time is T. We denote P^* as the best prediction of position. $\hat{p}_t$ is the position predicted by the dynamics. dt is the change in time.

[11] Code for the Velocity Estimator Model

Algorithm 1 Kalman Filter

```
Initialize: we set $m$ as the number of solutions we search for.
Initial estimate
$\mu_{p0}, \Sigma_{p0} \leftarrow p(\theta_p, x_0)$
$\mu_{v0}, \Sigma_{v0} \leftarrow v(\theta_v, x_0)$
Best estimate initially comes directly from the model.
$P^*_0 \leftarrow \mu_p$
while $t \in \{1, 2 \cdots T\}$ do
    We first use the dynamics and update
    $\mu_{\hat{p}_t} \leftarrow P^*_{t-1} + \mu_{v_{t-1}} \cdot dt$
    $\Sigma_{\hat{p}_t} \leftarrow \Sigma_{p_{t-1}} + \Sigma_{v_{t-1}} \cdot dt^2$
    Now we construct a Gaussian mixture of various solutions.
    Get Wrapped position looking at one step ahead
    $\mu_{p_t}, \Sigma_{p_t} \leftarrow p(\theta_p, x_t)$
    $\mu_{v_t}, \Sigma_{v_t} \leftarrow v(\theta_v, x_t)$
    We construct $m$ possible solutions which have the same covariance but have shifted means
    we denote $\mu^i_{p_t}, \forall i \in (0, 1, \cdots m)$
    Compute products of Kalman Filter Propagation
    while $i < m$ do
        $K \leftarrow \frac{\Sigma_{\hat{p}_t}}{\Sigma_{\hat{p}_t} + \Sigma_{p_t}}$
        $\mu^i_{p_t} \leftarrow \mu_{\hat{p}_t} + K(\mu^i_{p_t} - \mu_{\hat{p}_t})$
        $\Sigma {p_t}^i \leftarrow (\Sigma \hat{p}_t + K(\Sigma_{p_t} - \Sigma_{p_t}))K^T$
        Normalization Constant
        $C^i \leftarrow \frac{1}{\sqrt{\det(2\pi(\Sigma_{p_t} + \Sigma_{\hat{p}_t}))}} \exp[\frac{-1}{2}(\mu^i_{p_t} - \mu_{\hat{p}_t})^T (\Sigma_{p_t} + \Sigma_{\hat{p}_t})^{-1} (\mu^i_{p_t} - \mu_{\hat{p}_t})]$
    end while
    Sort for the $i$ that maximizes $C^i$. Denote the best value as $i^*$
    $P^*_t \leftarrow \mu^{i^*}_{p_t}$
    Now we update the positions
    $\Sigma_{p_t} \leftarrow \Sigma {p_t}^{i^*}$
    Update the velocity with the predictions from before.
end while
```

5.1.5 Kalman Filter Results

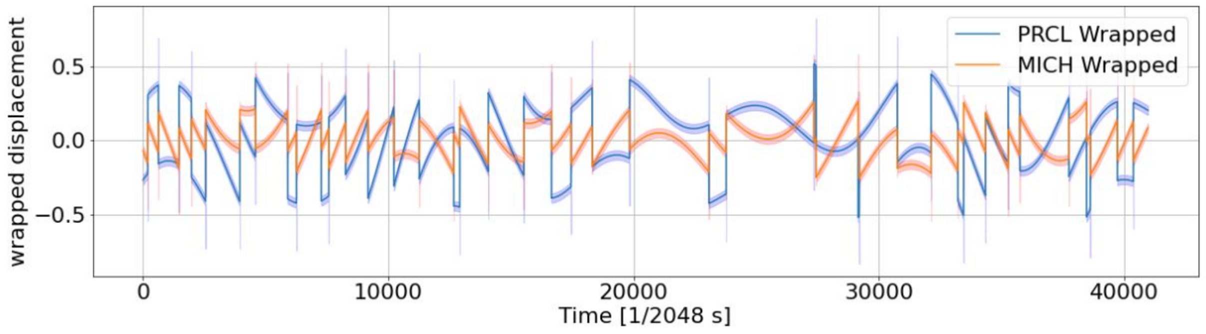

Figure 15: Wrapped positions with shaded area as the error.

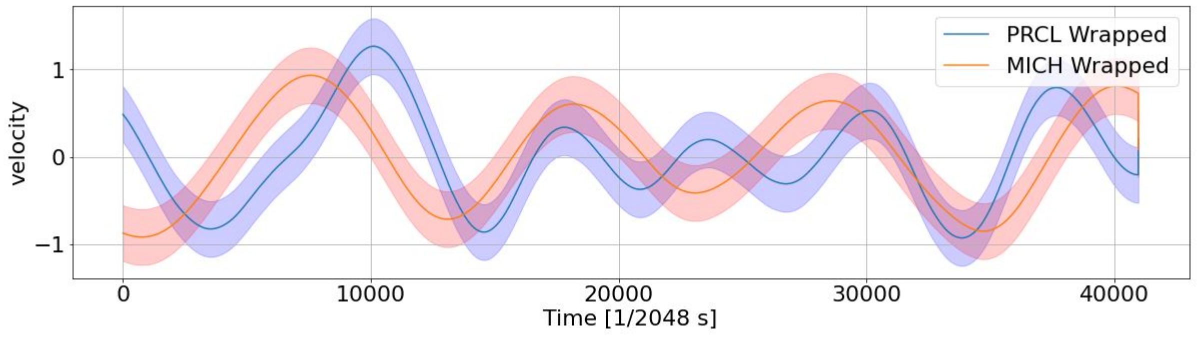

Figure 16: The velocity where errors are the shaded.

For the first initial implementation we preformed a brute force search. Specifically we construct a grid of possible solutions, each with a Gaussian centered around them. This can be seen in figure 17. Then we use these possible adjustments and adjust the original wrapped data. We upgrade this to a smarter search in the C implementation. [12]

[12]code for the idealized Kalman filter implementation

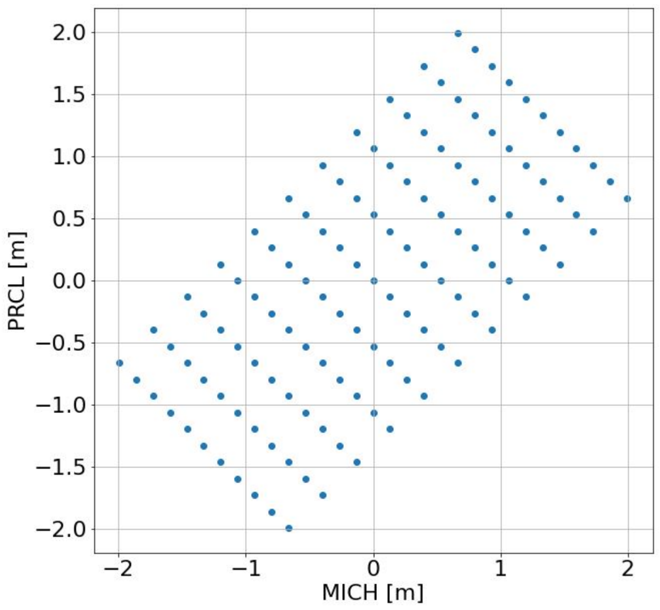

Figure 17: This is a grid of valid adjustments to the positions in the PRCL and MICH space such that it would result in the same signals.

Thus we can finally implement the algorithm to stitch back our solutions. We can see the results in figure 18

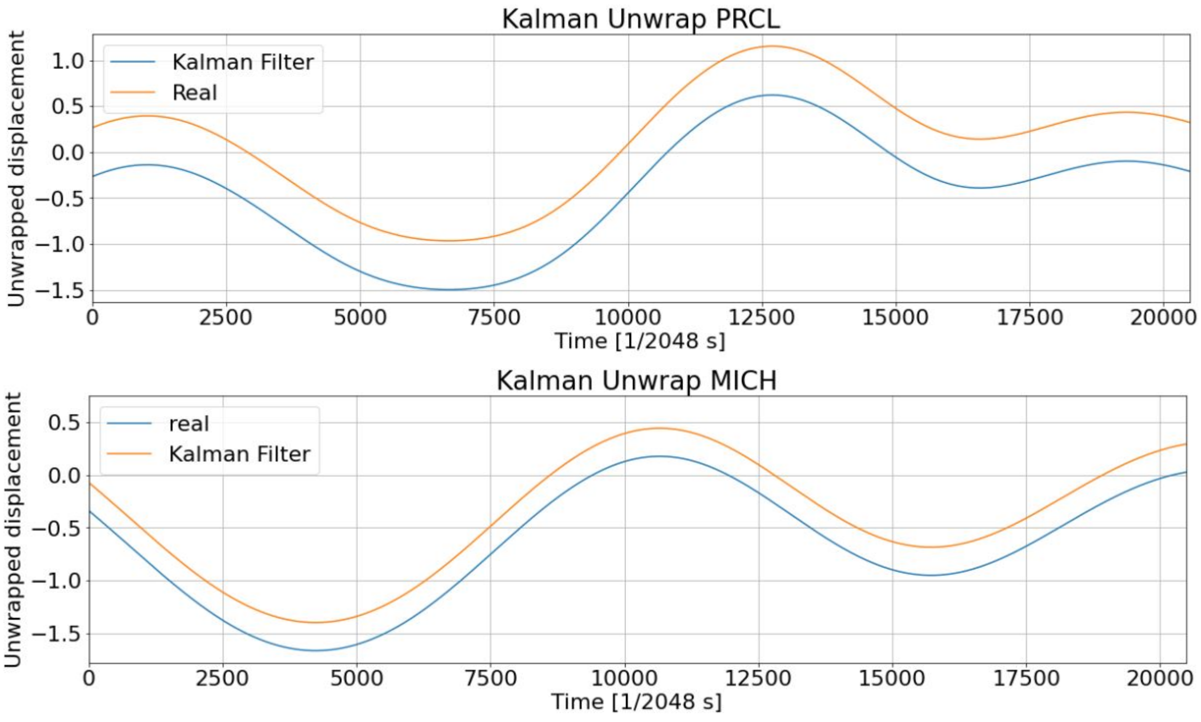

Figure 18: This shows that we've successfully unwrapped the idealized positions with our approach!

5.2 Final Reconstruction Results

We can finally take all the components and collectively reconstruct the positions such that it is smooth and continuous. Below is the implementation of such with real predictions made by our neural network! See figures 19 and 20.

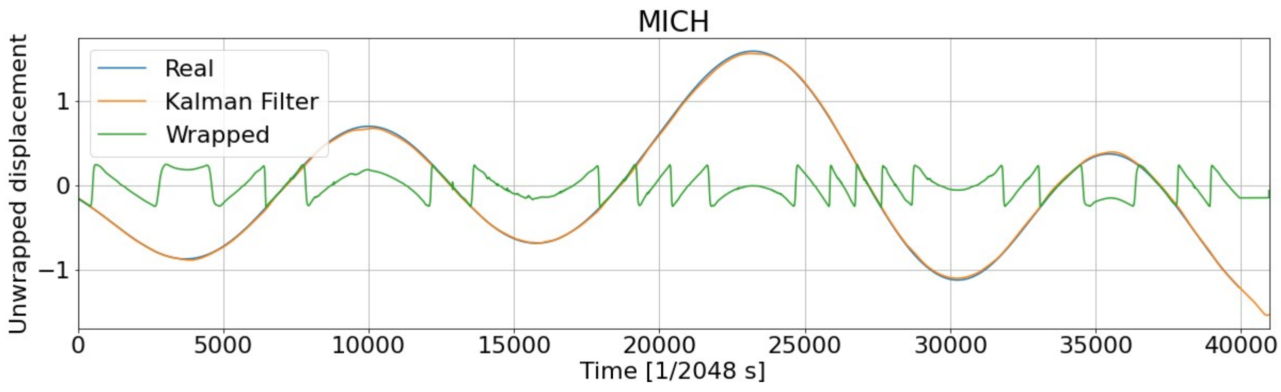

Figure 19: These are the MICH reconstructions from the real neural network

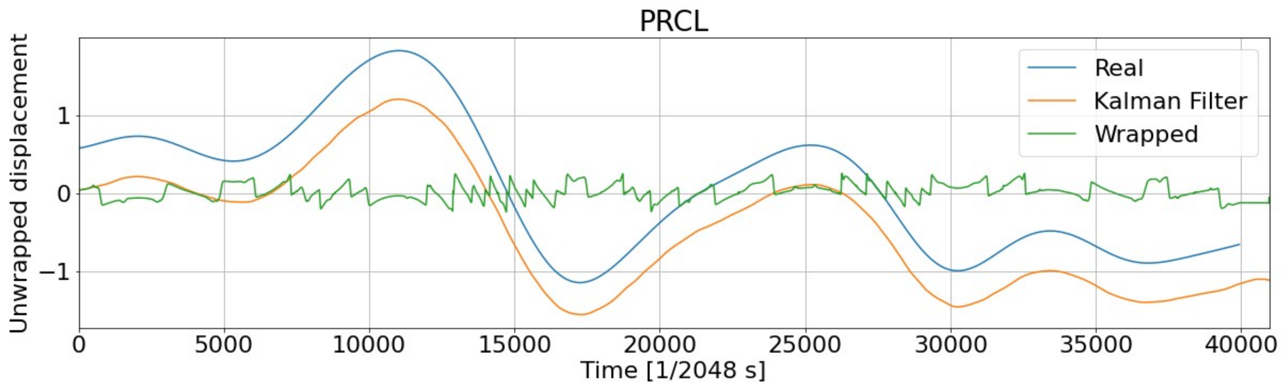

Figure 20: These are the PRCL reconstructions from the real neural network.

5.3 Smaller Models

Although the initial approach produced favourable results, we are faced with a new challenge of running the model and the Kalman filter in real-time at a sample rate of 2048 Hz. Furthermore, we are subject to hardware constraint. In production we will have to run this pipeline on a single CPU core and not on GPU's. Thus to meet such requirements we first make the model smaller in size.

5.3.1 Position Lite Model

For this model we have the following parameter sizes: 15 units for GRU 1, 128 units for GRU 2 and then 64, 32, 16 parameters for the dense layers and finally 2 parameters for the standard deviation and the mean layer. All layers used a 'LeakyReLU' activation with a slope of '0.3' and the final layer had a linear activation for the mean and a softplus activation for the standard deviation. This produces a 66,000 parameter model. [13]

However there was a substantial challenge in training this model. It was difficult to train the model using the same setup as we had with the large parameter model, most notably the mean reconstructions were poor using the same training settings as with the 11 million parameter model. Thus to mitigate this we append an additional loss term with the MSE between the mean and the true position value and we scaled this term with a parameter α. We found that $\alpha = 10$ produced favourable results yet preserving the effectiveness of the uncertainty predictions. Furthermore we found that changing the batchsize affected the training drastically. Previously we trained with a batchsize of 2000, in this case we used a batchsize of 64. The reason why we expected this to have worked better is because it allows the optimizer to make "finer" adjustments. Since we're using the ADAM optimizer, we compute the expected gradient from a mini batched sampled from the batched samples, and so if we have a larger sample space our adjustments are coarser. For a larger model we didn't observe this behaviour to have drastically affected its results but for a smaller model this produced much needed improvements. Together, the now 180x smaller model produces reconstructions on par with the original 11 million parameter. See figures 23 and 24

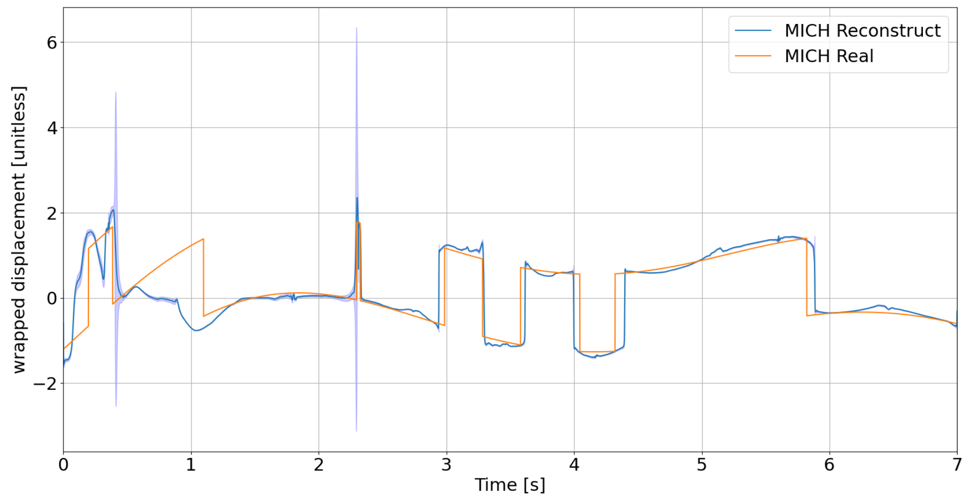

Figure 21: Wrapped positions of the MICH using smaller neural network with shaded area as the error.

[13]Please find the notebook here `https://gitlab.com/gabrielevajente/prmi-ml/-/blob/main/ml_experiment/uncertainty-position-lite/position_estimator_lite.ipynb`

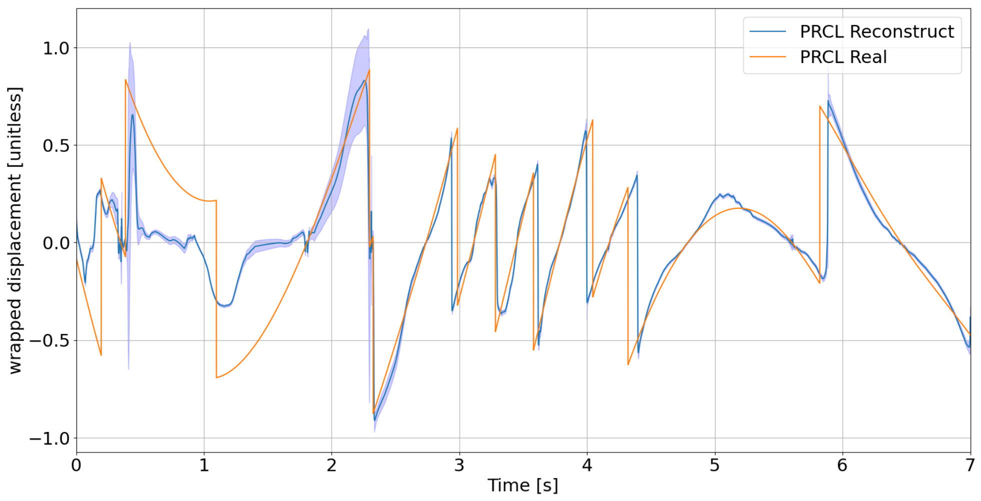

Figure 22: Wrapped positions of the PRCL with the smaller neural network with shaded area as the error.

5.3.2 Velocity Lite Model

We follow the same procedure and attempt to replicate similar results for the velocity model. This time our model has 20 units for GRU 1, 128 units for GRU 2 and then 64, 32, 17 parameters for the dense layers and finally 2 parameters for the standard deviation and the mean layer. All layers used a 'LeakyReLU' activation with a slope of '0.3' and the final layer had a linear activation for the mean and a softplus activation for the standard deviation. This produces a model with 77,000 parameters versus an 11 million parameter model. [14]

This time we set the MSE loss scaling factor to $\alpha = 50$. The mean reconstruction is particularly difficult in the velocity model. The reason why is because the velocity range is much larger than position since it is unbounded. Furthermore, we realized that the free motions of the mirrors due to seismic motions are not representative of motions of the mirrors when driven by the servos. Thus to reduce generalization error we simulate motions of mirrors where the velocity is high by randomly driving the mirrors. This made up $\frac{1}{4}$ of all training and testing data. This provides us with reasonable reconstructions that allow us to drive down the motions of the mirrors. Finally we used a batchsize of 64 for this model.

[14]Please find the notebook here `https://gitlab.com/gabrielevajente/prmi-ml/-/blob/main/ml_experiment/uncertainty-position-lite/velocity_estimator_lite.ipynb`

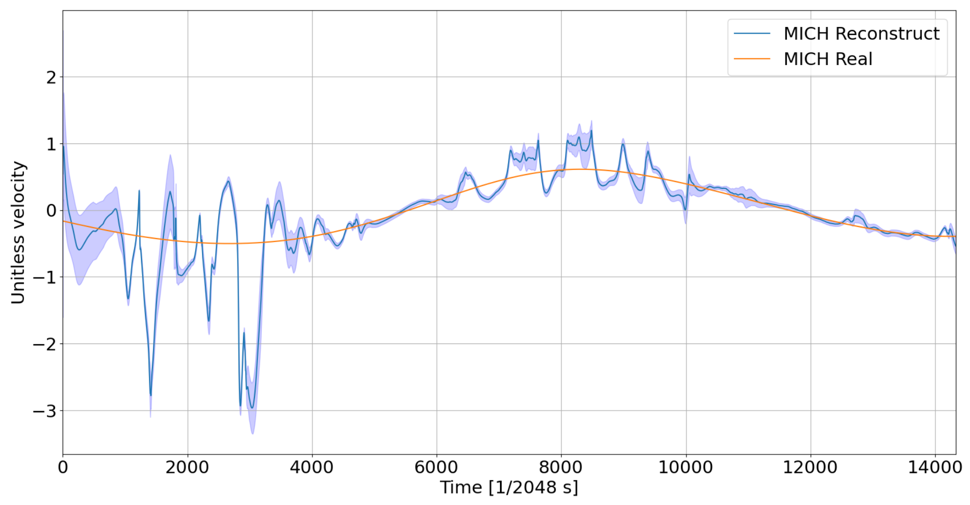

Figure 23: Velocity of the MICH using smaller neural network with shaded area as the error.

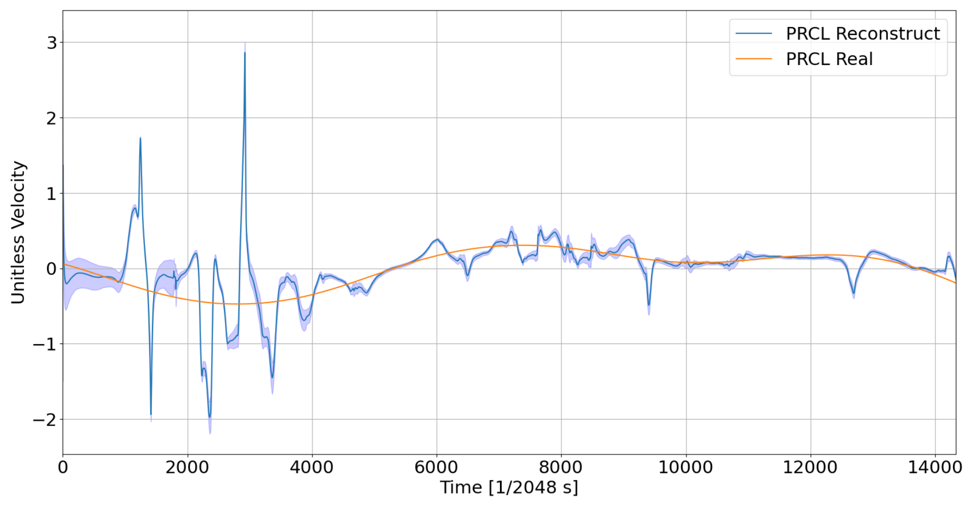

Figure 24: Velocity of the PRCL with the smaller neural network with shaded area as the error.

5.3.3 Kalman Reconstruction with Smaller Models

When adapting the Kalman filter to the smaller neural networks we found that the velocity model continuously produces poor reconstructions than the position model. This is anticipated since during training we realized that the velocity model needs to map to a much larger range and thus generalization is more difficult than the position model. However due to parameter constraints this model was still the best performing out of all the experiments. Thus to mitigate the generalization error, in the Kalman filter we artificially scaled the uncertainty of the velocity to be 15x greater than the position. SAlthough this biases against the velocity model trusting more the position model we can justify this since we know that uncertainties of the velocity model is less trustworthy due to our overfitting explained previously. With this change it gave us a Kalman filter that produced residuals on the order of $\frac{1}{100}$ of one half of a wavelength. Please see figures 25,26,27,29

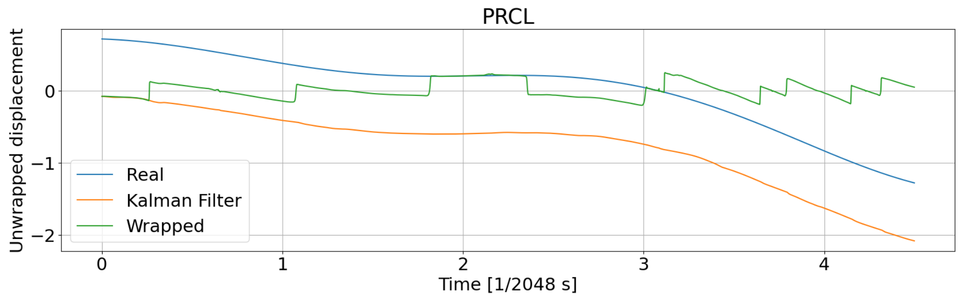

Figure 25: Kalman Unwrapping using the smaller neural networks this is PRCL.

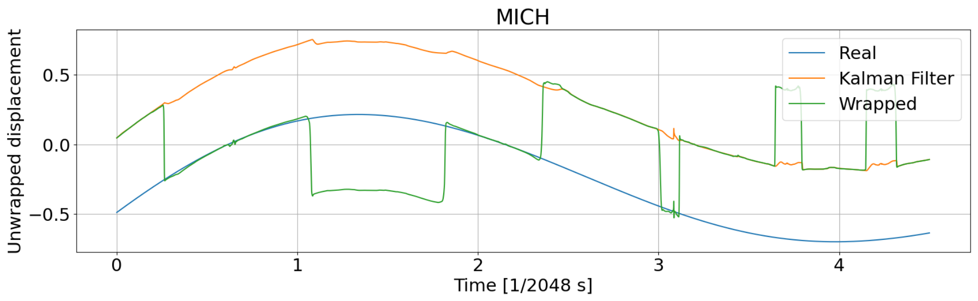

Figure 26: Kalman Unwrapping using the smaller neural networks this is PRCL.

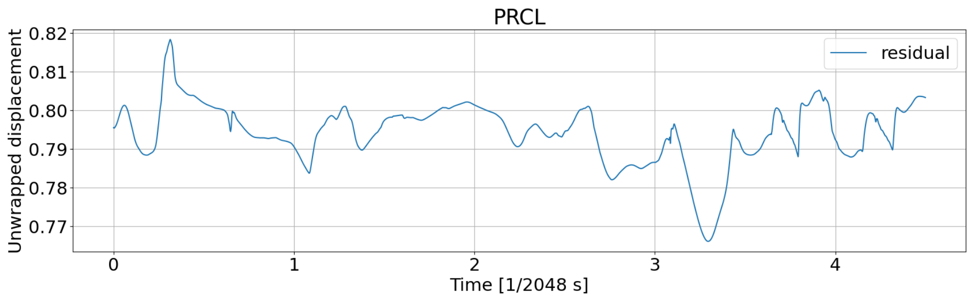

Figure 27: PRCL residuals we notice are on the order of $\frac{1}{100}$ of the normalization factor which is a half of the wavelength. Note that the offset hovers about -0.8, one of the offsets allowed by our wrapping is 0.798 which means we were correct!

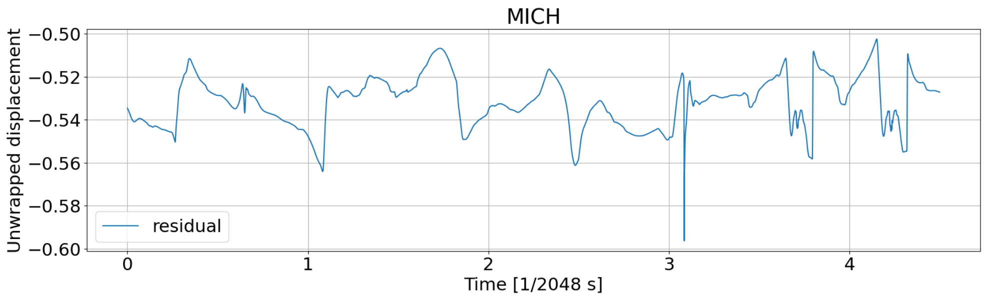

Figure 28: MICH residuals we notice are on the order of $\frac{1}{100}$ of the normalization factor which is half of the wavelength. (values vary between $-0.50 \rightarrow 0.56 \approx 0.53 \pm 0.03$ which is on the order of $\frac{1}{100}$ Once again, 0.53 is one of the allowed offsets introduced by our wrapping technique.

5.4 Implementing Model Into C

Furthermore we need to re-implement everything C such that we can use this on a real device. This is to make sure that it can run on a single CPU core and meet our speed constraints. To do so we implement a sparse matrix multiplication method in C. We take the the weights, (matrices) and save them as a unrolled vectors and we keep column and row vectors that keep track of the associated index. If the matrix is sparse we skip operations entirely. From our technique described above we do not need to make the operations sparse, we can simply use the model as is. [15]

Our Kalman filter implementation with neural networks operate just about real time, floating between 2000Hz on average sample rate. [16]

5.5 Acquiring Lock

Now that we have all the pieces assembled, let us verify that we can drive down the motions of the mirrors successfully. We implement a locking strategy that drives the mirrors in two stages, firstly we dampen the velocity of the mirrors, then we engage an integrator that drives down the mirrors towards 0. Within 10 seconds we can successfully drive down the motions to approximately small motions on the order of $\frac{1}{100}$ of the original amplitudes. This results in high power of the optical signals which demonstrates the success. [17]

The two stage lock, had its force clipped to $25 \times 10^{-3} N$ and the forces were ramped up over the course of $1.5s$. This demonstrates that we are finally successful in our work.

[15]Convert the model using this notebook `https://gitlab.com/gabrielevajente/prmi-ml/-/blob/main/kalman_c/convert_model_c.ipynb`

[16]Please find the C verions of Kalman filter here `https://gitlab.com/gabrielevajente/prmi-ml/-/blob/main/kalman_c/kalman_filter_nn_c_gabriele.ipynb`

[17]Please review this notebook for locking demonstration `https://gitlab.com/gabrielevajente/prmi-ml/-/blob/main/kalman_c/FullStateLock_Damp-neural_net.ipynb`

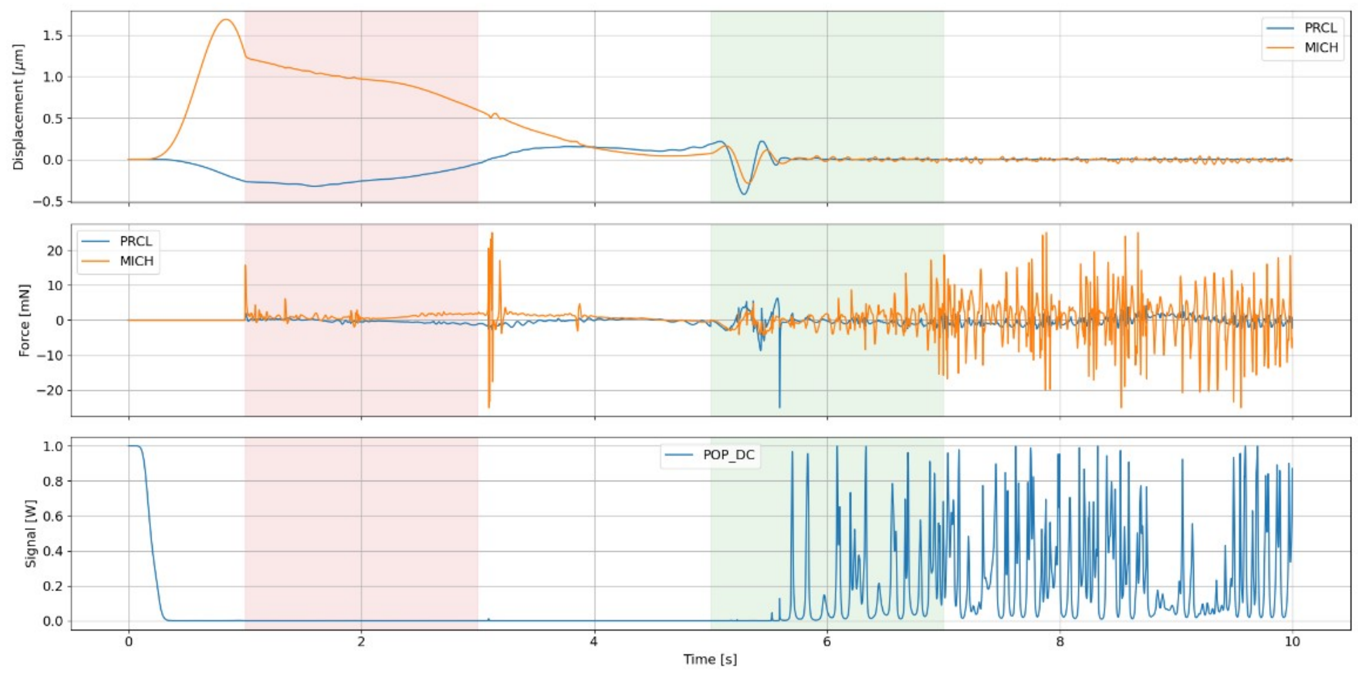

Figure 29: Here we demonstrate the ability to drive down the motions of the mirrors up to $10^{-8}m$ using our technique. Note that the RED demonstrates the region where we ramp up the forces for the dampening stage, and the green region demonstrates the region where we ramp up the forces for locking stage.

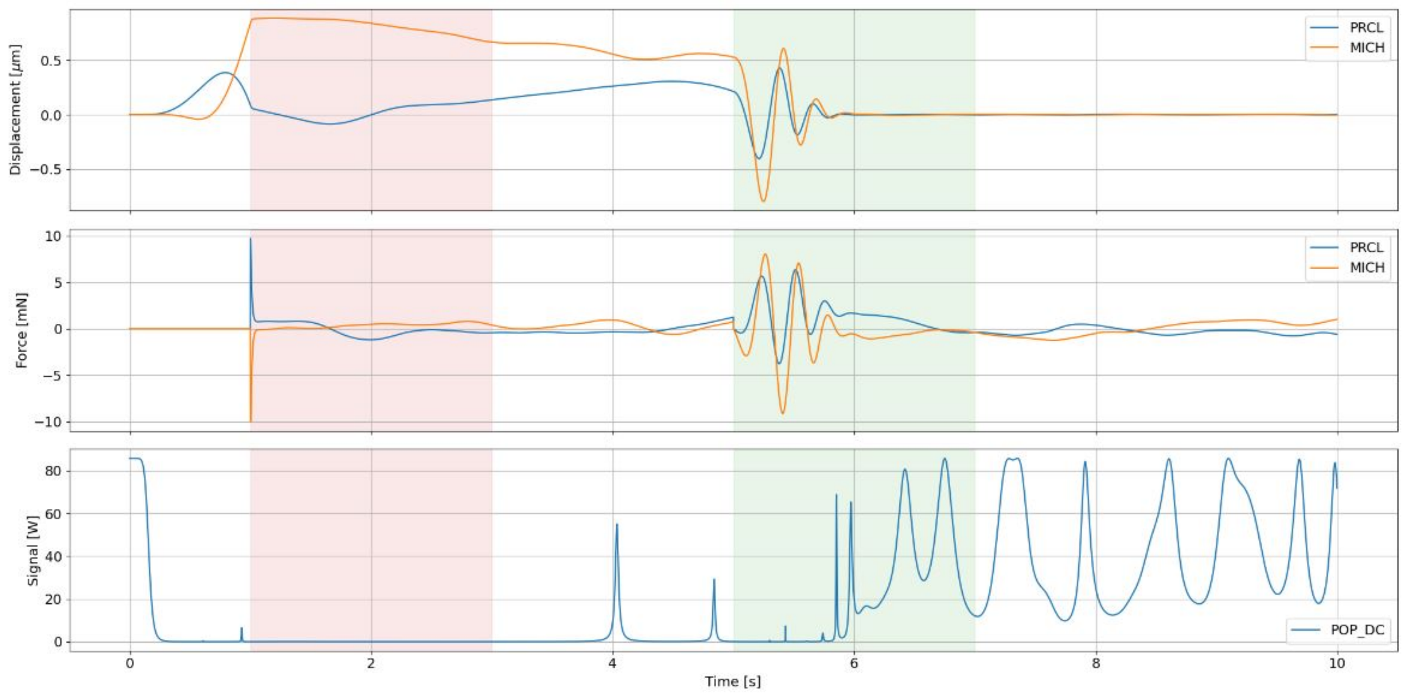

Figure 30: Here we demonstrate locking with perfect knowledge of the positions

6 Deep Deterministic Policy Gradient

6.1 Motivation

Generally speaking, some of the greatest advances in machine learning come from increasingly abstracting away tasks and simply letting the machine do everything. This has loosely been the trend in ML research for a number of years. Back in the 'prehistoric eras', ML models used to work only with highly hand-crafted features for input, that was until the advent of deep learning when we saw these features learned implicitly by our algorithms. We increasingly try to move the human further away from the picture and surprisingly found huge improvements from computer vision tasks to all the way to NLP. This similar philosophy is adopted by some of the most successful ML researchers tackling various kinds of hard problem. Quoting Deepmind's CEO Demis Hassabis, "let the gradients run through everything". We approach hard problems initially with an intermediate solution that often requires lots of hand-tuning, to which then practitioners try to generalize the approach to the bigger problem. We saw this with the invention of the CNN, we saw this with the invention of Alphafold1 $\rightarrow$ Alphafold2[9], we saw this with the invention of transformers, and we will probably continue to see this trend until either we create the Terminator and kneel before our AI overlords, or figure out some other way to solve intelligence. This is scary because we lack agency, but time and time again we've been proven wrong, in a good way. This possible upside in "superhuman" abilities alone is enough to warrant at least an investigation into its potential. Because of this, despite my supervisor's skepticism in letting me build an AI to control LIGO mirrors directly, I wanted to approach this problem in the most general way possible which is directly learning the controls of these mirrors.

More concretely, RL could be interesting because we know that giving only the information about the signals might not be enough to recreate the positions. What might hint that RL can solve the problem is that we can provide models with additional information, specifically the reward signal at certain time steps. The optical signals received plus the expected reward for certain actions could help us gauge what kinds of controls engage. This could provide unique solutions to the control problem. However, this is not trivially clear that this is the case and further investigation is needed. I briefly diverted some time on this problem.

6.2 DDPG Method

In classical approaches to reinforcement learning, we typically use tabular methods. These algorithms work by searching a space of actions and storing these optimal actions at a given state in Q tables. These tables help inform how to perform tasks given the state you're at. Typically these solutions could involve dynamic programming, temporal difference learning, or Monte Carlo tree searches. These methods are only tractable when the state space is finite. However, in most practical cases these state spaces are near-infinite. For this, we need approximate solution methods. A well-known means of approximation is to use neural networks such as Deep Q learning [10]. Given some arbitrary continuous state, we can predict the expected reward value for a particular action, this is the q function in RL terms. However, neural networks have a finite number of predictions, thus this method also relies on the fact that the action space itself is finite. What happens when the action space is also infinite? Just like how we used a neural network to predict the action's value we use a second neural network to then evaluate the network's performance in an ACTOR and CRITIC setup. Similar to a generator and a discriminator in GANs for computer vision, the actor, given the input produces actions that are continuous values. Then we have a critic that takes in the state, and the predicted actions and tries to predict the expected discounted reward on that action taken. This allows an agent to maneuver and learn from a continuous state space and action space which is important in our setup.

To use this technique we have numerical simulations that provide actions based on the continuous state of the optical signals. We also want to predict continuous actions and direct how much force each of the servos should engage to drive down the motions. Thus the first RL approach we should take would be Deep Deterministic Policy Gradient (DDPG) methods[18] [11].

6.3 RDPG Method

Not only do we need to implement a DDPG method, we also need to implement a recurrent version of DDPG called RDPG. This is in part because we know in order to predict the dynamics we need to get the velocity somehow and this requires us to use a recurrent model. Now this RDPG will use the exact same set up with the original DDPG method except the replay buffer now indexes a history of observations and actions to which we then feed the actor and the critic. The training step is identical as a traditional DDPG except both models will get both the state/observation history and the corresponding history of previous actions. We implement the the algorithm from the following paper [12] section 3.1.2

6.4 First Iteration RDPG

For the first iteration of the RDPG we wanted to iterate quickly and thus we have resorted to down sampling the simulation from a 2048 Hz to 20 Hz. Now the reason why aggressively downsampling is an issue is because we loose a lot of information about the dynamics of the mirrors. For example if the mirrors were moving fast and the mirrors are moving towards the correct position, we could miss this crossing since the sample rate isn't high enough to actually reflect this crossing in the data. Nonetheless this is meant as demonstration purposes to iterate on various ideas before we tackle the bigger problem.

We implemented a Critic with the following architectures. Signal history is fed into a GRU with 128 units followed by another GRU of same size. Then we feed the action history into another set of GRU with 128 units followed by a second GRU of the same size. The outputs of the two are then concatenated and fed into a feedforward neural network of size 512 followed by hidden layers of 256, 32 and 1 respectively. The model uses RELU activations. The Actor is symmetrical meaning design wise it is the same except it has 2 outputs and the outputs are activated using tanh

[18]code: here

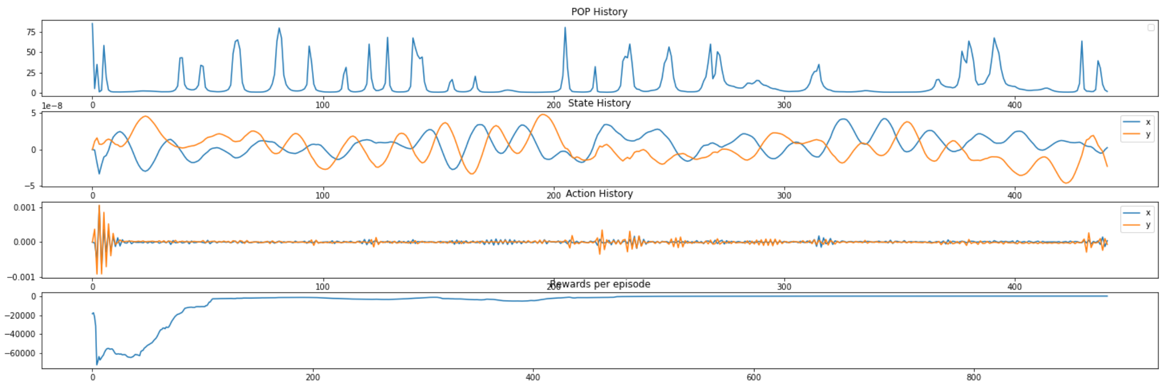

Figure 31: This shows promising results!The reason this is promising is because the motions of the mirrors are on the order of $1/100$ of a wavelength shown in the state history plot. The reward is improving as the model sees more episodes in rewards per episode plot. We also see that the power of the signal is high shown in POP history.

since the actions need to be bounded.

We also have various hyperparameters. We restricted actions to a force of 10^{-3} and we gave the initalization of the last layer of the actor $\pm 4 \times 10^{-4}$ so that values initially stay close to 0. For exploration we added noise to the predictions in the policy with a variance of 10^{-4} with mean 0. We also have soft updates and gave $\tau = 10^{-6}$

Another caveat to this iteration is that we are using the real positions as inputs for the reward function. The reward function takes the negative MSE of the position, thus the smaller the movements the greater the reward the model receives. The reason why we have not used only the signals for the rewards is because crafting such a reward function is difficult and we want to first test if this idea could even work. Hence the first iteration.

6.4.1 First Iteration Results

We see from first glance, the results appear promising! We see that the model has begun to learn and is improving over time. Now the actual performance is still quite poor, we haven't spent time tuning or investigating this just yet. We believe this preliminary result is enough see figure 31

6.5 Reward Function

Reward function or the reward signal is the objective of the problem. On each timestep the environment sends the agent a single number and the goal is to maximize this reward over a long term period. We know our goal is to push the mirrors into a linear regime. We can detect this linear regime by looking at the signal responses specifically POP. When the power of the optical signals shown by POP is high and constant, then we will know we can easily acquire the lock. To mathematically formulate this, we craft a reward function which is the inverse square of the derivative of the power. This way, when the change is constant we get maximal reward whereas any large change won't receive any. We square it to make the reward invariant to the sign of the derivative. We add an ε term to make this numerically stable. We also have a step function for the reward. Given high power, we reward the model for achieving such. S_t is the state received from the environment i.e the signals at time t. We set some *const* as the threshold for when we consider it to be high enough power. We set α as a constant hyperparameter in scaling the rewards.

$$R_t = \begin{cases} [(\frac{dS_t}{dt})^2 + \varepsilon]^{-1} + \alpha & \text{if } S_t > const \\ -\alpha, & \text{else} \end{cases} \tag{10}$$

7 Forward

In the short 10 week time frame we explored 3 methods of acquiring the lock. We investigated the use of attention based state estimators to brute force the non-uniqueness problem, then we implemented a probabilistic model + Kalman filter to lock the motions of the mirrors, and finally we investigated how reinforcement learning using models like recurrent DDPG techniques could have promising results. We believe that we've already devised an acceptable solution to locking acquisition problem using the probabilistic position-velocity model.

Looking forward we seek to devise a means of testing this on real hardware to verify our claim. Furthermore, should this solution prove favourable we then will look into adjusting this work to more complicated setups instead of just the simple PRMI mirrors.

References

[1] Cho, K., van Merrienboer, B., Bahdanau, D. & Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches (2014). URL https://arxiv.org/abs/1409.1259.

[2] Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate (2014). URL https://arxiv.org/abs/1409.0473.

[3] Chollet, F. *et al.* Keras. https://keras.io (2015).

[4] Vaswani, A. *et al.* Attention is all you need (2017). URL https://arxiv.org/abs/1706.03762.

[5] Kazemi, S. M. *et al.* Time2vec: Learning a vector representation of time (2019). URL https://arxiv.org/abs/1907.05321.

[6] Kaplan, J. *et al.* Scaling laws for neural language models (2020). URL `https://arxiv.org/abs/2001.08361`.

[7] Kalman, R. E. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering* **82**, 35–45 (1960).

[8] Kingma, D. P. & Welling, M. Auto-encoding variational bayes (2013). URL `https://arxiv.org/abs/1312.6114`.

[9] Jumper, J. *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature* **596**, 583–589 (2021). URL `https://doi.org/10.1038%2Fs41586-021-03819-2`.

[10] Mnih, V. *et al.* Playing atari with deep reinforcement learning (2013). URL `https://arxiv.org/abs/1312.5602`.

[11] Quillen, D. *et al.* Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods (2018). URL `https://arxiv.org/abs/1802.10264`.

[12] Heess, N., Hunt, J. J., Lillicrap, T. P. & Silver, D. Memory-based control with recurrent neural networks (2015). URL `https://arxiv.org/abs/1512.04455`.