Control of a Ball on a Tilted Rail Using Neural Networks

Project for the course *MTAT.03.241 Modelling and Control of Dynamic Systems*

Konstantin Tretyakov, Darya Krushevskaya
University of Tartu

January 28, 2009

Contents

Introduction

The aim of the project is to study in practice the methods for modelling and control of dynamic systems based on neural networks. We apply the methods to a toy real-life example: controlling the position of a ball on a rail by tilting the rail. The project consists of four main stages – construction of the system, identification of a neural network-based model of the system, creating a controller based on the model, and finally, applying the controller to the real system. The description of each stage is presented in the following sections.

1 System Construction

1.1 Conceptual Model

The system that consists of a ball on a tilted rail (Figure 1) was chosen as the focus of the project mainly for its conceptual simplicity. The system only requires one input for controlling the tilt of the rail, and has one output – the position of the ball. The system is simple to construct yet quite complicated to control. It is not linear due to friction and the limited lenth of the rail. It is also highly unstable. We therefore consider a modest goal of creating a controller that would be capable of positioning the ball in the middle of the rail.

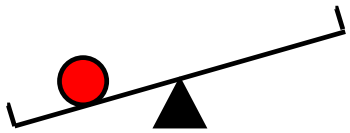


Figure 1: The "Ball on a Rail" system.

1.2 LEGO-based Implementation

We used a LEGO Mindstorms NXT^{TM} kit to construct the system (Figure 2).

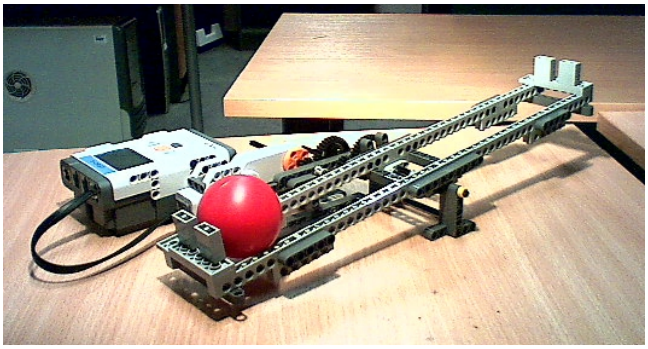


Figure 2: The system constructed from LEGO blocks.

The tilt of the rail is controlled using a motor that is connected to the rail via a set of gears. The NXT servo-motor can be controlled using two types of commands – `setMotorPosition` and `setMotorSpeed`. The former requests the motor to rotate to a prespecified absolute angle (measured in "ticks"). The latter requests the motor to rotate with a speed of a given number of "ticks" per second. Experiments showed that the former way is unreliable and prone to serious lags. When requested to rotate to a given angle the motor often "overshoots" due to inertia and then takes time to correct the position, rotating back and forth by single ticks. This seriously hinders the control of the rail. The second method of controlling the motor, by setting its rotation speed, is somewhat more reliable and therefore we selected that method for our implementation. Unfortunately, however, setting the motor rotation speed influences the actual tilt of the rail in an indirect manner, which complicates the control task further on.

The motor can be controlled by a program running on the NXT "intelligent brick" or, more conveniently, by a program running on a PC and sending commands to the NXT brick via USB or Bluetooth. The technical details of this configuration, turned out to be somewhat more complicated than we could expect, hence we provide a brief review.

Preliminary tests indicated that the most reliable way of controlling the NXT brick from PC via Bluetooth is provided by the iCommand Java library, that comes with the leJOS NXJ toolkit. We therefore installed the leJOS NXJ firmware on the brick (which was probably an optional step, because iCommand is presumably capable of controlling the original LEGO firmware, too) and created a Java proxy, providing the `setMotorSpeed` function. For various convenience reasons we had to implement the remaining code in Python and thus interfaced it to the Java-based proxy using TCP/IP. Thus, the input to the system was provided in a way illustrated in Figure 3.

Figure 3: The input signal pipeline.

1.3 Observing the Output

Clearly, the system cannot be controlled in an open-loop manner, hence we need a way to observe the location of the ball. To detect the location of the ball we used a usual laptop webcam. The position of the ball was recognized from the image using an ad-hoc algorithm, by computing the average location of red pixels. The obtained number was then normalized so that 1 would correspond to one edge of the rail and

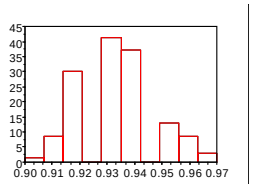


Figure 4: Observed values corresponding to the ball's leftmost position.

−1 – to the other edge. The VideoCapture and PyGame Python modules were used to implement this part. Although both the idea and implementation were rather crude, the average measurement error was generally less than 0.05, which seemed tolerable (see Figure 4).

1.4 Limitations and Complications

The constructed system is anything but a fine instrument. It contains numerous rough spots, which produce considerable noise. Firstly, the NXT motor does not respond to requests instantaneously and thus the timing of the response to input signals varies. In our implementation, we could achieve the sampling rate for input and output somewhere between 10Hz and 11Hz (see Figure 5). Secondly,

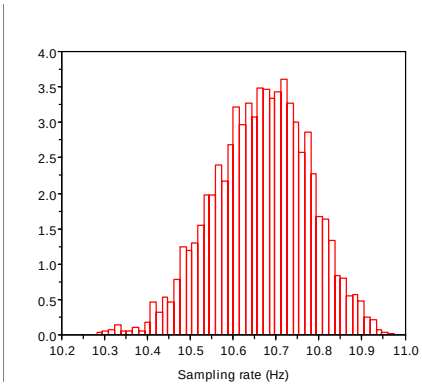


Figure 5: Variability of the sampling rate over a session.

the system contains some inertia and the gears are not ideally tight. Thirdly, the rail is not ideally smooth and the ball is not ideally round. Fourthly, the webcam and the whole system were repositioned between experiments. Finally, the changing lighting conditions influence the ball recognition algorithm. All this introduces considerable noise both in the input and output signals.

It is easy to see that the system is nonlinear, because both the input and the output are strictly bounded. Besides, once the ball hits a bound and stops, the tilt of the rail must be changed much more to make it move again, than when the ball is still moving. Another complication is that the system has a potentially infinite lag-space, because once the ball is at one of the bounds, it is possible to play with the motor and change the tilt of the rail without affecting the position of the ball. As a result, as long as we observe the ball at one of the bounds, there is no way to derive the actual tilt of the rail and thus guess the proper control input. All that makes the system quite difficult to control. In fact, when a joystick was attached to control the system, the author himself failed to balance the ball in the middle of the rail no matter how hard he tried

except for once, which was due to sheer luck.

2 System Identification

2.1 Data Aquisition

In order to create a model for the system we collected a dataset of input/output values by "playing" with the system. For that we attempted two methods. At first, we implemented a PID-controller and let it control the system while tuning its K_p, K_i and K_d parameters interactively. We found this method of data aquisition to be suboptimal due to the following reason. We did not find any parameter settings, for which the PID controller would be capable of holding the ball away from the bounds for a significant amount of time. As a result, the PID-controlled system was bouncing the ball from one bound to the other with a somewhat stable periodicity. This periodicity is well visible from the output autocorrelation function (see Figure 6, bottom right plot). We therefore considered an alternative data collection strategy, where the system was controlled manually using a joystick. The data collected this way shows less periodicity. Also the input signal spectrum coverage is probably slightly more relevant here (see Figure 7). As a final dataset we used 5000 measurements collected in a single joystick-controlled session. We split the set into two parts: 3000 points form the training set and 2000 – the test set.

2.2 Model Order Selection

Physical considerations. In order to predict the position of a ball in an idealized tilted rail system, it should, in theory, be sufficient to know the position, velocity and acceleration of the ball and the tilt, speed and acceleration of the rail. An estimate to the first three values can be inferred from the current system output $y(t)$ and its outputs one and two sampling instants ago ($y(t-1)$ and $y(t-2)$). The rotation speed and acceleration of the tilt of the rail can be estimated from the last two input signal values $u(t-1)$ and $u(t-2)$. The complication lies in the estimation of the current tilt of the rail. When the ball is moving, the current tilt is proportional to its acceleration, and could thus be inferred from $y(t), y(t-1)$ and $y(t-2)$. However, when the ball is still, the only way to determine the tilt reliably is to integrate $u(t)$ back to the moment when the tilt was known, i.e. when the ball was moving. As there is no fundamental limit to how long the ball may be still, the lag space for the idealized system is infinite.

These observations provide useful hints for selecting the model order. Firstly, it does not make sense to observe many previous outputs y. An idealized system needs no more than 3. A real system could use a bit more to compensate for the noise, but presumably not more than 6 or so. The choice of the number of previous inputs to consider is less obvious. On one hand, when the ball is still, an unlimited number of previous inputs are required to model the system

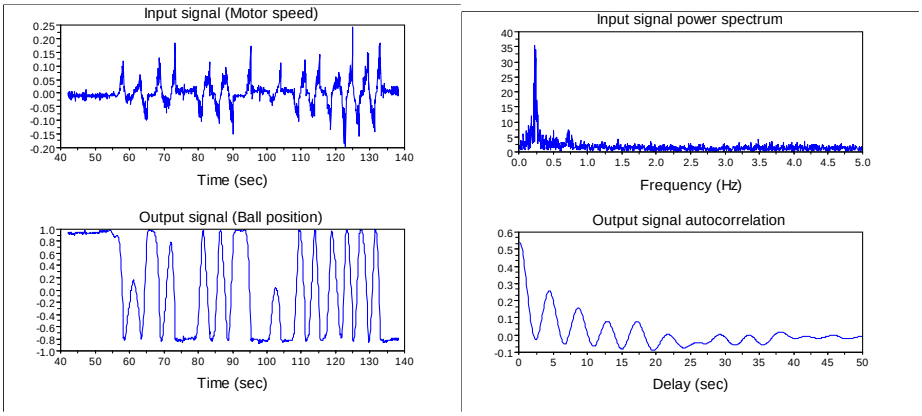


Figure 6: Data collected using a PID-controlled session.

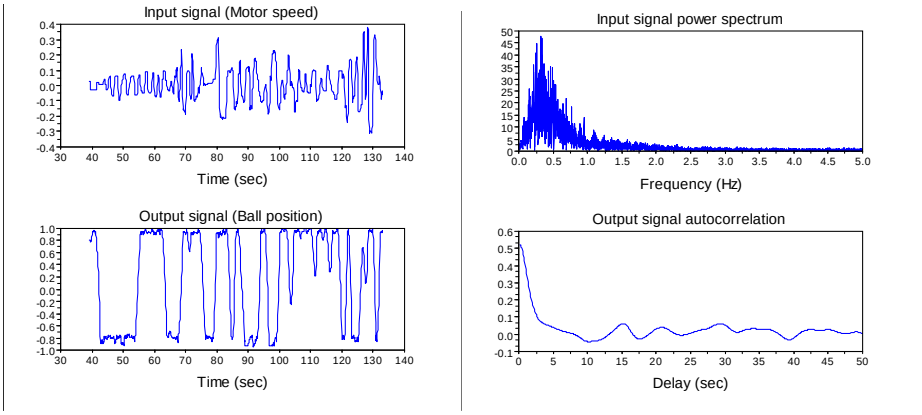


Figure 7: Data collected using a joystick-controlled session.

correctly. However, once the ball has begun moving, only two previous inputs need to be known. Thus, a model that uses just two previous inputs will be inferior to the model that uses an unlimited number of previous inputs only at those moments, when the ball starts moving. If in practice the ball does not get still for too long, it seems most reasonable to select the input order somewhere inbetween 2 and the average expected time for the ball to stand still (say, 20), with a strong preference towards smaller values.

Lipschitz coefficients. The NNSYSID toolbox provides an obscure method `lipschit` for model order selection, that uses Lipschitz coefficients and black magic. The method works by computing a certain coefficient (*order index*) for each pair of input and output orders. The pairs that correspond to smaller numbers should generally be preferred. Figure 8 demonstrates the order indices for input orders ranging from 4 to 20 and output orders ranging from 3 to 8. Judging from the picture, output order of about 5 could be well appropriate. As for the input order, it should be selected at either 7 or 12.

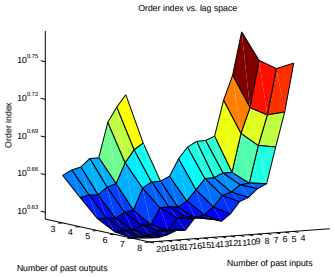


Figure 8: Input-output order indices.

Linear model check. An understanding of the lag order can be obtained by fitting simple linear ARX models of various orders and examining their performance. We use the MATLAB's `arxstruc` routine to estimate a range of ARX models corresponding to input orders $1 \ldots 20$ and output orders $1 \ldots 8$. We then use the `selstruct` function to select the optimal structure according to either the lowest validation error, or according to the MDL or AIC criteria. Figure 8 demonstrates that the obtained results agree with the Lipschitz test: the suggested output order is 5 or 6 and the input order somewhere around 10 (or 1, but with a 10-step-delay).

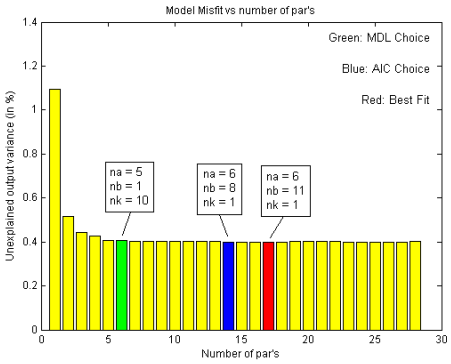
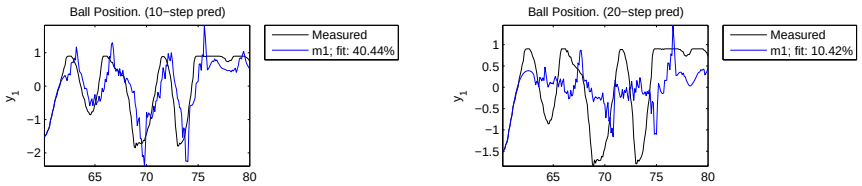


Figure 9: ARX-based model order selection.

Conclusion. Considering all of the above, we conclude that the output order of the modeled system should be chosen at either 5 or 6. The input order is one of $7, 8, 11, 12$.

2.3 Training a Linear Model

In the previous section we have already got a taste of the ARX model performance (see Figure 8). The plot clearly illustrates that, despite the nonlinearity, the system can be modeled linearly to some extent, at least for the purpose of one-step-ahead predictions. Also, the choice of the model order does not matter very much. It is important to note, however, that the performance indicators employed are rather misleading. Indeed, in terms of one-step-ahead predictions, the ARX$(1, 0)$ model (i.e., a model that does not use the information about the input signal) is *also* quite good simply because the dynamics of the system is comparably slow and by only predicting the previous output it is possible to achieve 90% fit. Figure 10 provides a more realistic outlook on the performance of ARX models by demonstrating the 10 and 20-step-ahead predictions for an ARX$(6, 11)$ model.



(a) 10-step prediction (b) 20-step prediction

Figure 10: Predictions of an ARX model

Seeking to improve the linear model performance, we also experimented with MATLAB's `oe` and `armax` procedures.

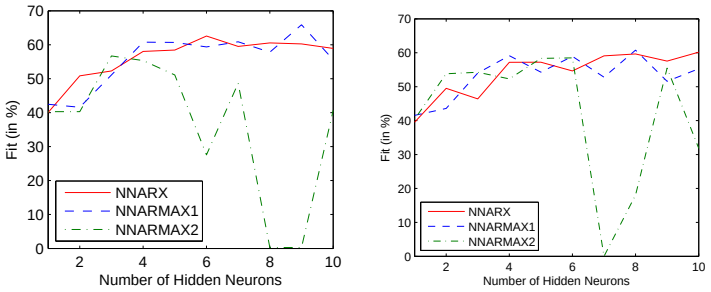
ARMAX. When we add one moving average term to an $ARX(6, 11)$ model (i.e., consider the $ARMAX(6, 11, 1)$ model), the 10-step prediction fit (evaluated on the whole validation set) increases from 48.8% to 52% and the 20-step prediction fit – from 18% to 26%. Adding further moving-average terms seems to have no visible effect.

OE. The *output error* model structure did not seem to produce any remarkable results. This is probably obvious, as the OE structure is just a special case of the ARMAX structure.

2.4 Training a Neural Network

Architecture selection. We first aim to find a NNARX or a NNARMAX model structure that is capable of performing 10-step ahead predictions significantly better than the ARMAX model from the previous section (which had 52% fit). We start from NNARX$(6, 11)$ and NNARMAX$(6, 11, 1)$ architectures with one hidden-layer tanh neuron, and add neurons one-by-one evaluating the

ten-step prediction quality (fit). The results are demonstrated in Figure 11 (a). Examination of the plot suggests a NNARX model with 6 hidden neurons or a NNARMAX1 model with 4 neurons, of which the former is preferable because it is feedback-free. We then repeat the experiment for the NNARX$(5, 7)$ and NNARMAX$(5, 7, 1)$ architectures, the corresponding plot is shown in Figure 11 (b). As the accuracy of the smaller model is slightly worse, we decide to stay with the larger model and use pruning techniques to reduce the parameter space.



(a) $n_a = 6$, $n_b = 11$ (b) $n_a = 5$, $n_b = 7$

Figure 11: Fit quality versus number of hidden neurons.

Weight decay parameter. An important side issue is the choice of the weight decay parameter α. In the previous experiments we used $\alpha = 10^{-4}$ but it could be suboptimal. Figure 12 demonstrates that the validation fits for networks trained with various values of α do not differ significantly as long as $\alpha \leq 0.01$. We thus continue using $\alpha = 10^{-4}$.

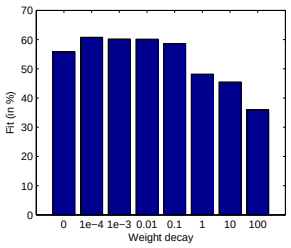
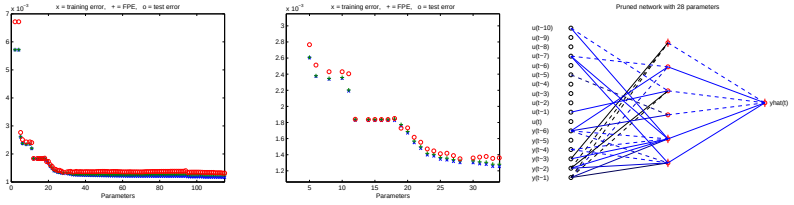


Figure 12: Fit quality versus number of hidden neurons.

Network pruning. Recall that we have focused on a NNARX$(6, 11)$ model. The performance of this model was suspiciosly close to the performance of a smaller NNARX$(5, 7)$ model. It follows that the model could benefit from a reduction in the parameter space by pruning. The pruning session implemented in the `nnprune` method of the NNSYSID package eliminates weights one-by-one, retraining the network on each step. Figure 13 (a) illustrates the output of the session by plotting the model error versus number of nonzero parameters. Note that the plot somewhat resembles Figure 9 – the model error is more-or-less constant for all parameter counts greater than 28 or so.



(a) Model error vs number of parameters.

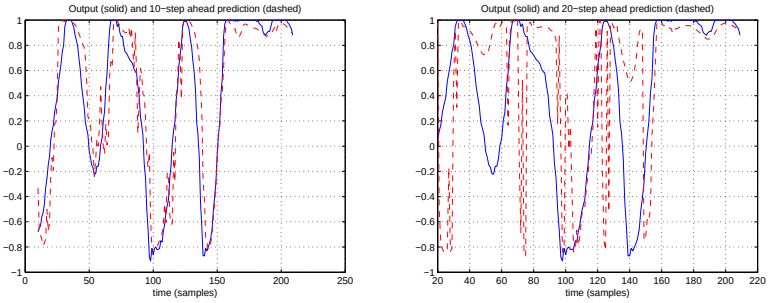
(b) Zoom of (a)

(c) Final network

Figure 13: Network pruning.

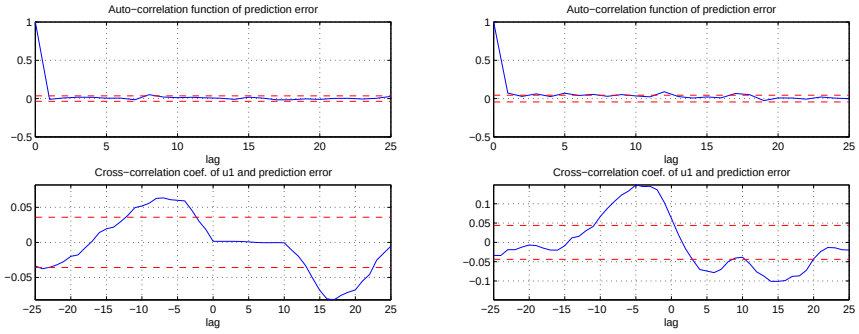
Network validation. Let us examine the obtained model in more detail. The 10-step-ahead predictions (see Figure 14) have become somewhat more reliable (with a fit increased to 58%), yet the 20-step-ahead predictions are still useless, which is probably due to inherent system properties and thus tolerable. The auto- and cross-correlation plots (Figure 15) demonstrate that the model is not completely perfect – the cross-correlation of input $u(t)$ and the prediction error is slightly outside of the "safe" bounds (which correspond to a 95% confidence interval around 0). However, considering that the cross-correlation between prediction error and $u(t)$ only becomes large at lags greater than the ones we deem important and also remembering that the input $u(t)$ was in fact pruned away as irrelevant, we conclude that the found model is nonetheless more-or-less the best that we could derive. As a final test we ran the model on a *second* test dataset, that was not used before at all – the dataset of 2000 points collected in a PID-controlled session. The results were better than those on the original test set: 62.7% fit and a somewhat milder cross-correlation misfit.

System controllability. By examining the pruned network it is easy to note that most of the edges connecting to inputs u have been pruned away. It is therefore of interest, whether the knowledge of the input signal plays any significant role in predicting the output at all, or is it rather the case that the system output can be predicted from previous outputs alone. The latter would indicate that the system is completely impossible to control. To test that we trained a NNARX$(6, 0)$ model with 6 hidden neurons as before. The fit of the resulting model was around 44% which is, fortunately, somewhat smaller than the 58% achievable by the network that "knows" the output. This provides some weak hope, that the system could be controllable to some extent.



(a) 10-step prediction (b) 20-step prediction

Figure 14: Predictions of the final NNARX model.



(a) Training set (b) Test set

Figure 15: Auto- and cross-correlation plots for the final NNARX model.

3 System Control

3.1 PID Controller

As already noted above, early in the beginning we attempted using a manually-tuned PID controller to control the actual system. The tuning of the controller went as follows. It was quite easy to note that the integral term K_i was unnecessary. In our system the ball tends to stay at the bounds most of the time and goes from one bound to the other within instants. With nonzero K_i the system tends to "charge" too much while the ball is on one side, and this leads to serious overshoots when the ball moves to the other bound. Next, we noted that large values of the proportional term K_p make the system act in an unnecessarily "rapid" manner when the ball was at the bound. This does not correspond to what we expect from the controller that is capable of keeping the ball off the bound. Finally, the differential term K_d had to be chosen reasonably large so that the controller could try to stop the ball immediately once it started moving away from a bound. By testing various combinations of small K_p and large K_d interactively, we concluded that a PID controller is incapable of positioning the ball in the middle.

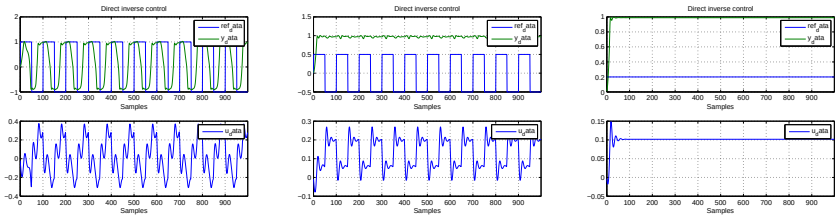
3.2 Direct Inverse Control and Optimal Control

The next natural candidate control method is *direct inverse control*, i.e. a neural network, directly predicting the next control input from previous inputs and outputs. The task is thus equivalent to the system identification task with inputs and outputs interchanged. We therefore first of all performed the same model order selection procedure that we did in Section 2. This time we avoid excess documentation, though.

We use the same training and test sets as for the identification task. The Lipschitz coefficients tests suggests using at least 9 previous inputs and 5 previous outputs for predictions. Somewhat different results are obtained by the `arxstruc` linear model selection procedure, which recommends observing $6-8$ previous inputs and 2 previous outputs. It is notable that the best linear model one-step-prediction *mis*fit is large this time: 6%.

The number of hidden neurons in the network did not seem to matter much, with a 5-neuron NNARX$(9, 5)$ model looking optimal. Analogously, the exact value of the weight decay parameter was not too important. After pruning, the final model achieved 76.5% one-step-ahead prediction fit.

The controller is capable of tracking the amplitude 1 square signal to some extent (Figure 16), but fails miserably when requested to control amplitude 0.5 square wave or even a simple constant 0.2 level (which would correspond to keeping the ball somewhere in the middle). The use of "specialized training" procedures, that optimize the error in the resulting output rather than the error in the input prediction, and that are provided by the `special1`, `special2` and `special3` modules of the NNCTRL package, did not generally improve the situation (Figure 17). Moreover, the training would not converge on most



(a) Square signal (b) Small square signal (c) Constant signal

Figure 16: Direct inverse control.

attempts independently of the chosen algorithm.

A variation of the "specialized" direct inverse control is the so-called "optimal control", where the optimization criterion consists of both the sum of squared output errors as well as the sum of squared input signal magnitudes, weighed by a penalty constant ρ. However, when ρ was larger than 10^{-4} or so, the results of the corresponding `opttrain` procedure were worse than that of the usual direct inverse control (Figure 17). The training algorithm quickly diverged from the initial approximation (which was taken as the previous direct inverse controller) and could not settle to a suitable state. Figure 17 illustrates a sample from a typical training epoch. When ρ was set to a very small value such as 10^{-10} optimal training did not result in a controller much different from the plain direct inverse one.

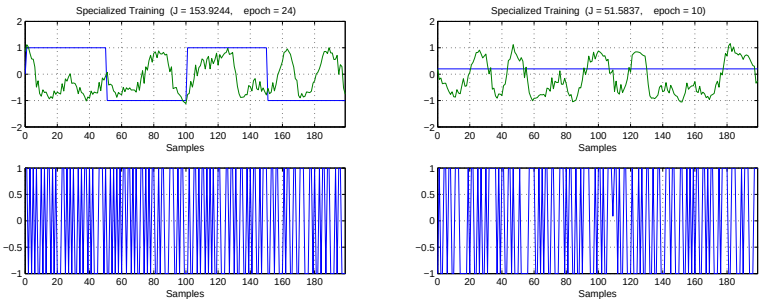
3.3 Feedback Linearization

As a second attempt we select the feedback linearization strategy. We trained the $f(\cdot)$ and $g(\cdot)$ networks using the `nniol` procedure, following the architecture selected in Section 2, namely $NNARX(6, 11)$ with 6 hidden neurons. For technical reasons, no pruning was attempted. The resulting forward model seemed to perform somewhat worse than the one identified in Section 2, but not significantly (94.82% versus 95.13% one-step-ahead prediction fit).

The control task failed to track both the square wave as well as to keep the constant level. In addition, the controller was regularly producing input values significantly exceeding the allowed maximum (Figure 18).

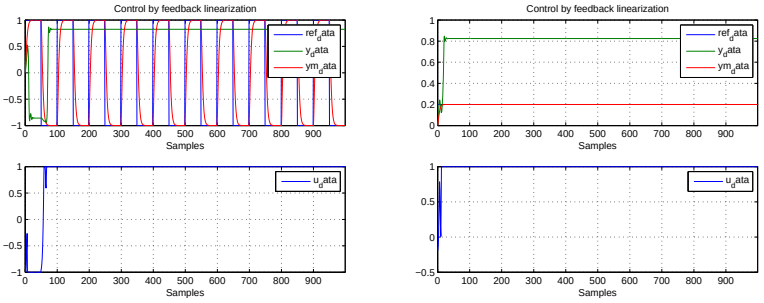
3.4 Nonlinear Predictive Control

In terms of the NNCTRL toolbox, *nonlinear predictive control* strategy is somewhat similar to the optimal control, but instead of training one network that would solve the control task for the whole trajectory in terms of one-step-ahead predictions, a separate optimization is performed at each step of the control trajectory. At each time point a control input is chosen such that if it were held for the next k samples, the system output would best match the reference.



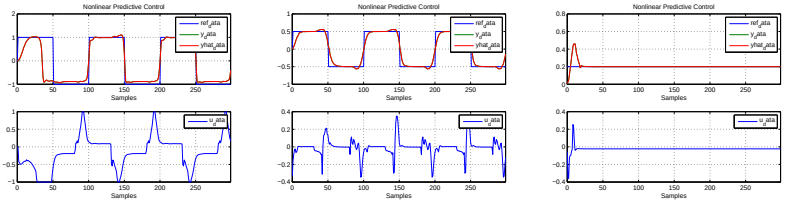
(a) Square signal (b) Constant signal

Figure 17: Specialized and optimal control ($\rho = 0.001$).



(a) Square signal (b) Constant signal

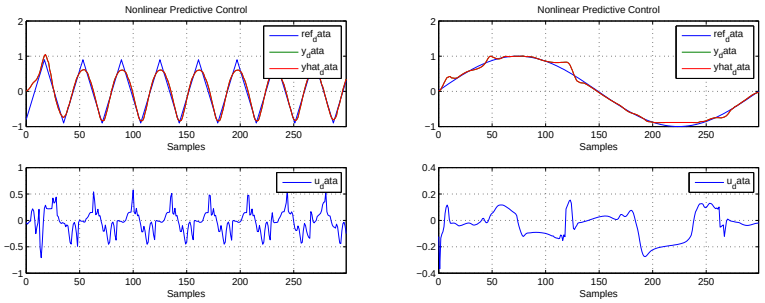
Figure 18: Control by feedback linearization.



(a) Square signal (b) Small square signal (c) Constant signal

Figure 19: Nonlinear predictive control ($\rho = 0.5$, $N_2 = 20$).

This is certainly a reasonable strategy for the system under consideration. When using a joystick to control the system it becomes clear rather quickly that the only way to at least somewhat control the system is to "plan ahead". This logic is confirmed by experiment, which demonstrates that, indeed, predictive control is at least capable of satisfactorily controlling the NNARX$(6, 11)$ model of the system from Section 2 (Figure 19, Figure 20). Although it is not a convincing proof that the original real-life system can be controlled, it is a significant improvement over the previous controller techniques. The algorithm requires tuning two parameters: the control input penalty coefficient ρ and the prediction horizon N_2. After some manual fiddling we selected the values of $\rho = 0.5$ and $N_2 = 20$ as the best.



(a) Sawtooth signal

(b) Sinewave signal

Figure 20: Nonlinear predictive control ($\rho = 0.5$, $N_2 = 20$).

3.5 Conclusion

To summarize, we have explored a range of neural-network based control techniques, aiming to control the NNARX model of the system, obtained in Section 2. We discovered that all of the nonpredictive control methods failed, with the best nonpredictive solution being the direct inverse neural network controller. In comparison with other methods, this controller was trained directly on the data rather than a model of the system, and this probably was an important advantage in our case.

In contrary, the predictive controller performed much better, being able to satisfactorily track various standard reference trajectories. In fact, the performance of the controller was somewhat "too good" due to the fact that on each step it was selecting the control input carefully tuned to control exactly the NNARX model it was given. It seems quite improbable that the real system could be tracked with such precision by any controller at all. Therefore, it might turn out that although the controller is good at tracking the neural-network model of the system, it will still fail in practice. We shall test this in the next section.

4 Controlling the Real System

Finally, we attempted to apply the controller from the previous section to control the real system. To interface Matlab with Python code we used plaintext files (i.e. Python code would write system observed state into a file, Matlab code would read that file, use the obtained value to derive control input, and write the control input into a second file, which Python would then read to obtain the value, etc).

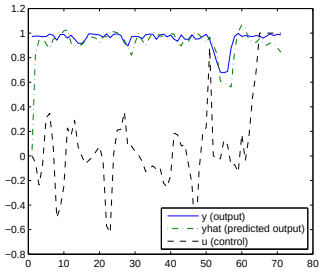


Figure 21: Control of the real system.

Unfortunately, the results are completely unsatisfactory. In all experiments the controller would bounce the ball back and forth for some 3-6 seconds and then start spinning the motor rapidly in one direction (i.e. send the control signal 1), effectively destroying the system. Figure 21 displays a typical control session. The reasons for that behaviour are unclear, especially considering the fact that the predictions of the forward model seem to be reasonably close to the true values.

Conclusion

Although the final result did not satisfy all of our expectations, we do not consider this as something extraordinarily bad, because the expectations were clearly inflated. The project, nonetheless, provided exciting experience and useful observations.

- Although a PID controller was incapable of properly controlling the real system, it was still far better than all of the nonlinear controllers, because, at least, it was not destructive. This supports nicely the popular rule of thumb, that only simple things usually work in real life.
- The *model* of the system, however, could be best controlled by a nonlinear predictive controller. This is somewhat of a cheat, though. Because the model of a system is noise-free, the predictive controller can "tune" its input to achieve nearly perfect control. When applied to a real and thus noisy system this strategy can fail, which is what we observed in practice. "In theory, there is no difference between theory and practice, but in practice there is." True.
- Although neural networks were used as a successful means of system identification, all of the neural-network-driven control methods failed. The predictive controller is not inherently specific to neural networks, it used a neural network model simply because we had one to offer.