# Using Neural Networks to Improve Mobile Robot Dead-Reckoning

Andrew Wilson

***Abstract*— This paper describes the implementation of a neural network algorithm using training data derived from an experimental mobile robot dataset. The data is drawn from experiments run at the Autonomous Space Robotics Lab at the University of Toronto. The robot's control inputs and position were captured during each test. We examine the performance of a neural network to learn a motion model for the robot and compare it to a simplified model of the robot's motion. The results show that the neural network successfully learns an improved model of the robot's motion using a two-layer network with four hidden units.**

## I. INTRODUCTION

In the realm of simulation of mobile robots, many assumptions can be made when creating a mathematical model of the robot's motion. The fidelity of the model can have a significant impact on the performance of the overall simulation and level of approximation to the physical system. Using measurement data collected by the robot can improve the performance of the simulation using filtering algorithms, such as the Extended Kalman Filter; however, filtering approaches cannot improve the core model that is used in the simulation. Machine learning algorithms provide a means of using experimental training data to learn a more accurate model of the robot which can be highly nonlinear.

This report will examine the performance of one type of machine learning algorithm - Neural Networks. A set of training data will be constructed from the UTIAS dataset. The dataset used in this implementation is the MRSLAM_Dataset4 for Robot_3. Given this training data, a model of the robot will be learned and compared to a simplified model of the robot's motion. The effect of parameters in the neural network algorithm will also be examined.

## II. LEARNING GOAL

The first task is building the training data that will be used to learn a better motion model for the robot. The fleet of robots used in the experiential data collection were five iRobot Create platforms [1]. They are two-wheeled, differential drive robot platforms equipped with a monocular camera for landmark range and orientation sensing.

Figure 1 shows the coordinate system for the dataset. The angle, $\theta = 0$ aligns the world frame with the robot's body frame. The robot's configuration states are given by $(x, y, \theta)$.

### A. Simplified Motion Model

The robots used in the dataset have two control inputs. The robot records the forward velocity commands, $v$, (along the $x$-axis) and the angular velocity commands, $\omega$ (about the $z$-axis). We will compare our learning algorithm's performance

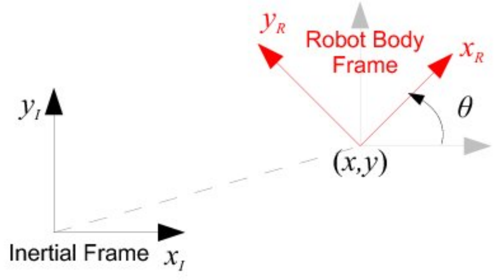


Fig. 1. Robot body frame and world frame coordinate systems [1].

to a simplified kinematic model of the robot. This model assumes that the robot's body velocity is linear in the direction of heading. In the robot's body frame, this results in the following equations of motion:

$$\begin{aligned}\dot{x}_R &= v\\ \dot{y}_R &= 0\\ \dot{\theta} &= \omega\end{aligned}$$

For the purposes of learning the robot model, we will assume that the model is independent of the robot's world position and orientation. This will allow for a lower dimensional mapping from the robot's control inputs to changes in the states in the robot's body frame. At each timestep, we will convert these changes in body states back to the world frame using a simple transformation:

$$\begin{pmatrix}\Delta x_I\\ \Delta y_I\end{pmatrix} = R_{IR}\begin{pmatrix}\Delta x_R\\ \Delta y_R\end{pmatrix}$$

where

$$R_{IR} = \begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}$$

### B. Training Dataset

With the robot's coordinates and nominal model defined, we can create a training dataset for the learning algorithm. The input training set, $X$, for the algorithm will consist of the two control inputs, averaged over a predefined timestep, $dt$. For this implementation, we will use a timestep of $dt = 2$ seconds. This will help to smooth the data and filter some of the high frequency noise from the control inputs.

The output training set, $Y$, will be the change of states in the body frame of the robot during the defined timestep. The data for this set comes from the groundtruth data from

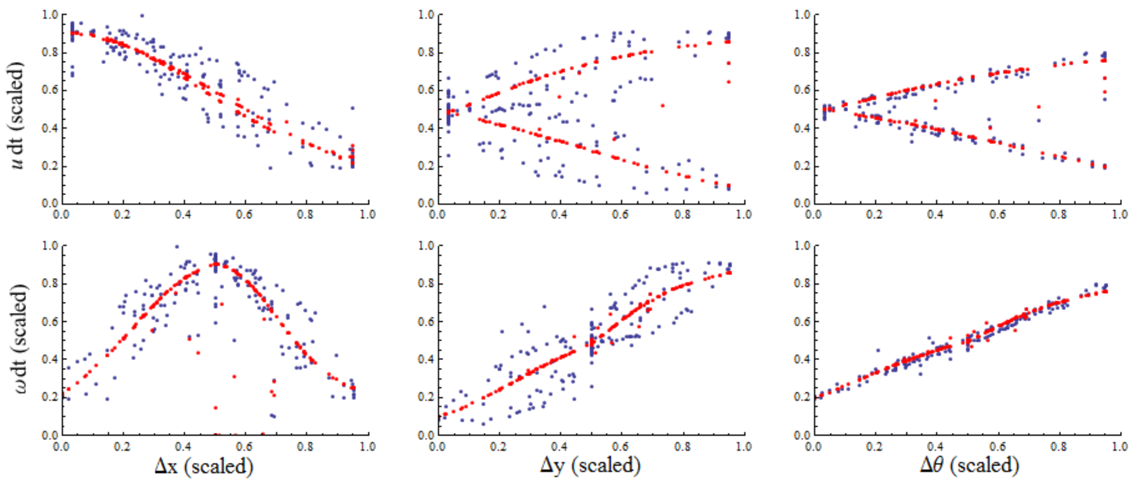


Fig. 2. Training data (blue points) and learned function values (red points).

the experimental dataset which was obtained with a external Vicon camera system tracking the robot's position in the world. In order to match timestamps for both the control and ground truth data, a first order interpolation of the data was done and training data was extracted from the interpolation.

This choice of training data should provide a much better capability to learn the robot's kinematic model from timestep to timestep. In the simplified model, the robot's motion is only along the $x_R$-axis. Now, the algorithm can learn a nonlinear function for the $x_R$ translation, as well as a $y_R$ translation which is not possible in the linear model. Additionally, a nonlinear function for the $\theta$ rotation can be created, replacing the linear model that was assumed.

For the neural network implementation, we also need to scale the output data to the range $(0, 1)$. This is due to the use of the sigmoid function in the output layers which will be discussed further in the next section. Scaling the input to $(0, 1)$ also helped the overall performance of the algorithm, although not required. The scaling parameters were set by hand for training data used. Figure 2 shows the scaled values of the training dataset represented by the blue points.

## III. Artificial Neural Networks

With a set of training data, we can attempt to learn the best function that approximates the relationship between the input and output. A popular approach to learning involves Artificial Neural Networks. The networks are inspired by biological neurons which are simple units interconnected to form a complex network.

### A. Sigmoid Units

The basic unit that will be used in this implementation of a neural network is the sigmoid unit. The sigmoid unit computes a linear combination of the inputs from the previous layer and applies a differentiable threshold to the result. This threshold function is given by the sigmoid function:

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

where $y$ is a linear combination of the inputs, $\vec{w} \cdot \vec{x}$, with weights, $\vec{w}$.

For classification problems, a different unit called the perceptron is sometimes used. This unit outputs either a 1 or -1 based on a linear combination of the inputs. The advantage of the sigmoid function is that it allows for continuous output which is better suited for regression problems. The sigmoid output is also differentiable which allows for hidden layers using the backpropagation algorithm, which will be discussed next.

### B. Backpropagation Algorithm

For a multilayer network, the backpropagation algorithm is used to learn the weights for each layer. This is accomplished by attempting to minimize the squared error between the network's output and training output [2].

The backpropagation algorithm incorporates a gradient descent weight update by computing the error in the network output compared to the training output, and then propagating the error backwards through the network to update the weights at each unit.

Each iteration of the algorithm performs the following operations:

1) The training input, $\vec{x}$ is input into the network. Outputs are computed at every layer of the network, resulting in a final output from the output layer.
2) An error for every output unit, $k$, is computed between the network output, $y_k$, and the training data output, $t_k$ using the following formula

$$\delta_k = (y_k)(1 - y_k)(t_k - y_k)$$

3) For each hidden unit, $h$, at the previous layer, an error term for each hidden unit is computed between the hidden output, $y_h$ and the following layer error, $\delta_k$ using the following formula

$$\delta_h = y_h(1 - y_h) \sum_k w_{kh} \delta_k$$

where the notation $w_{ij}$ represents the weight from unit $i$ to unit $j$ in the following layer.

4) For multiple hidden layers, the error calculation in step 3 is repeated for each layer until the input layer. For a two-layer neural network, step 3 is just performed once.
5) Each network weight is updated by the following equation

$$w_{ji}^{n} = w_{ji}^{n-1} + \eta \delta_j x_{ji}$$

where $x_{ji}$ represents the input from unit $i$ to unit $j$ in the following layer, and $\eta$ is the learning rate.

This procedure is carried out iteratively until a given termination condition is met, which could include a certain threshold on the error, sufficient decrease condition, or simply a maximum number of iterations. For this implementation, we constrained the neural network to a maximum of 2000 iterations.

## IV. Results

The primary purpose of this paper is to highlight the results and parameter choices of the neural network implementation. In this respect, we will examine the ability of the network to learn the training data and then test the network on additional data within the dataset. This method of cross-validation will allow us to ensure the network is learning an improved model while avoiding over-fitting of the data.

When referencing the data in this implementation, the point, $t = 0$ in the processed data corresponds to the timestamp 1288971880.0 in the raw dataset. The odometry and groundtruth data hs been interpolated to the first order, and the odometry has been sampled at a fixed timestep of 0.05s and averaged over a fixed timestep, $dt = 2$s.

For the results and tests that follow, the training dataset included 600 seconds of data beginning at $t = 100$s. The testing set included 600 seconds of data beginning at $t = 1500$s. Figure 2 shows the learned model represented by the red points given the training inputs.

### A. Learned Model vs. Simplified Model

The performance of the tuned neural network can be seen when compared with the simplified model. Figure 3 shows the dead-reckoning paths using the learned mode and the simplified model compared with the groundtruth measurements. Qualitatively, the learned model is significantly more accurate than the simplified model that was used.

The network used for this learning model was a two-layer network with four hidden units. The impact of the network structure on the performance will be discussed in the following section. The parameters used in this test were the learning rate, $\eta = 0.4$, and momentum, $\alpha = 0$. These parameters will also be discussed in following sections. To test the general performance of the learned model, the network was run on a testing dataset as well. The dead-reckoning path of the testing data can be seen in Figure 4. This also qualitatively shows much better performance of the learned model over the simplified robot motion model.

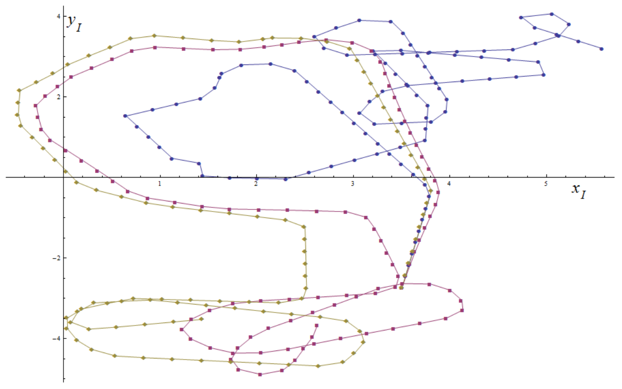


Fig. 3. Dead-reckoning paths over training set points from $t = 100$ to 300s with simplified model (blue), learned model (red), and groundtruth (yellow).

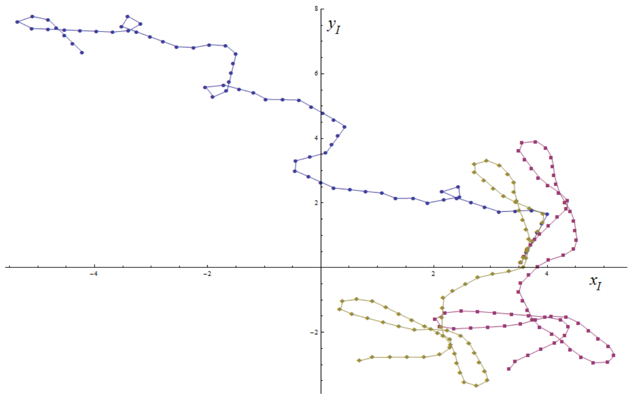


Fig. 4. Dead-reckoning paths over testing set points from $t = 1800$ to 1950s with simplified model (blue), learned model (red), and groundtruth (yellow).

As a quantitative comparison, the Euclidean norm of the error between the output of the network and the groundtruth position, as well as the error between the simplified model and groundtruth position was calculated at each timestep and the norm was taken over the entire training and testing trajectories. The results of this analysis can be seen in Table 1.

The table shows a significant decrease in the error of each state using the learning model in either the training or testing set as compared to the simplified model. The $x$ state shows an order of magnitude improvement in the error norm over the 600s trajectory. The $\theta$ state shows about 3x improvement in the error, most likely due to a high level of sensitivity in that state. The fact that the testing data error is on the same order as the training error suggests a high level of generalization of the model which is important, since the model can begin to overfit the data if learned improperly.

### B. Hidden Units

One important design decision in the neural network structure is the number of hidden layers and number of hidden units per layer. Although multiple layers of units can fit more complex data, the complexity comes at the

TABLE I

NORM OF THE ERROR OF EACH TIMESTEP OVER THE ENTIRE TRAJECTORY WITH 4 HIDDEN UNITS.

| | $\Delta x_I$ | $\Delta y_I$ | $\Delta\theta$ |
|---|---|---|---|
| Simplified (Training) | 6.16019 | 3.56681 | 4.76227 |
| Learned (Training) | 0.458173 | 0.601775 | 1.36266 |
| Learned (Testing) | 0.471169 | 0.531967 | 1.38338 |

TABLE II

NORM OF THE ERROR OVER TRAINING DATA AND COMPUTATION TIME FOR VARYING NUMBERS OF HIDDEN UNITS.

| | $\Delta x_I$ | $\Delta y_I$ | $\Delta\theta$ | Time (s) |
|---|---|---|---|---|
| Two units | 0.814169 | 0.682159 | 5.89655 | 94.3 |
| Four units | 0.458173 | 0.601775 | 1.36266 | 124.0 |
| Six units | 0.452290 | 0.601658 | 1.29548 | 153.7 |

cost of computation time and potential for overfitting the training data. We began with one hidden layer of units, and this configuration turned out to yield good performance. Therefore, multiple hidden layers of units were not used in order to keep the computation time as low as possible.

The decision that was made in this implementation was the best number of hidden units in the single layer. Table 2 shows the error norm associated with several different choices of hidden units and the computation time associated with learning the weights.. Two units provides the fastest learning time over 2000 iterations; however, the quality of the learned model is significantly lower than with four units. The increase to six units provides marginal improvement in the error norms; however, the computation time also increases by about 30 seconds. For this implementation, the best number of hidden units was determined to be four based on the error norm analysis.

## C. Learning Parameters

Two design parameters can be chosen for the implementation of the artificial neural network - the learning rate, $\eta$ and the momentum, $\alpha$. The learning rate is a proportional term which multiplies the weight update, $\Delta w_{ij}$ in step 5 of the backpropagation algorithm. Increasing the learning parameter to 1 resulted in a faster convergence of the algorithm, and decreasing the parameter to 0.1 resulted in slower convergence.

In both cases, the network converged to approximately the same values as in Table 1, which suggests that the algorithm is reaching an area with many local minima close together. Since the parameter didn't have a large effect on reaching the minima, we set it to a middle value of 0.4 to ensure that the algorithm learns quickly while attempting to avoid overtraining the weights early in the training data.

The second parameter is the momentum, $\alpha$. This is an optional term which creates a dependence of the weight update on the previous weight. Using a non-zero $\alpha$ results in the following weight update which is used in step 5 of the backpropagation algorithm.

$$w_{ji}^n = w_{ji}^{n-1} + \Delta w_{ji}^n + \alpha \Delta w_{ji}^{n-1}$$

where $\Delta w_{ji} = \eta \delta_j x_{ji}$.

The momentum parameter can help the algorithm avoid local minima and converge quicker by continuing to step in a descent direction that is proportional to the previous iteration. This can provide "momentum" in gradient descent, pushing the algorithm through any local minima. In the case of this implementation, setting $\alpha$ between 0 and 1 resulted in a very similar trained model, again suggesting that the set of minima are fairly wide and close together. Since the parameter did not have an appreciable effect, it was set to 0 for the tests.

## V. CONCLUSION

Overall, the neural network has been shown to successfully learn an improved model for the robot's motion. The resulting dead-reckoning path has less error than the simplified model which could be used to improve other algorithms such as Kalman filtering algorithms to incorporate measurements into the system.

For this implementation, the design choice that has the most significant impact is the number of hidden units used in the hidden layer. A balance is found that provides enough learning capability while limiting the complexity and computation time of the algorithm. Other parameters, such as the learning rate and momentum had a small effect on the converged weights of the model; however, given a different system, these parameters can have significant effects on the overall performance of the algorithm.

To ensure that the model was not overfitting the data, the testing dataset was used to validate the performance. Since the error in the testing data was quite close to that of the training data, the learned model appears to have very good generalization properties to data from other parts of the dataset. More rigorous cross validation could be carried out to ensure that all parts of the dataset are covered by the learned model.

Although neural networks can be slow to train, this implementation required only 124 seconds to train 600 seconds of data. After processing the training data, queries on the model are significantly faster than realtime, ensuring that the learned model could be implemented with an experimental setup along with other filtering algorithms, such as the Extended Kalman Filter.

## REFERENCES

[1] Leung K Y K, Halpern Y, Barfoot T D, and Liu H H T. The UTIAS Multi-Robot Cooperative Localization and Mapping Dataset. International Journal of Robotics Research, 30(8):969974, July 2011.

[2] Mitchell, Thomas M. Machine Learning. McGraw-Hill, Inc., New York, NY, USA, 1997.

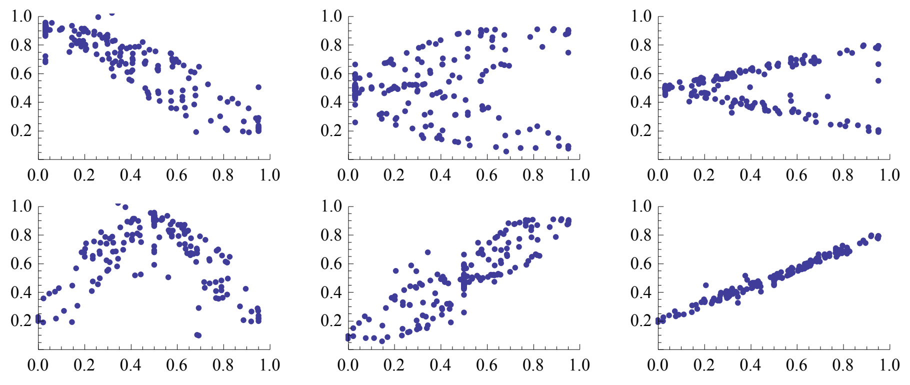
# Artificial Neural Network - Dataset #1

### Data Import & Processing

```
odomdata = Drop[Import["Robot3_Odometry.dat"], {1, 5}];
knownpose =  Drop[Import["Robot3_Groundtruth.dat"], {1, 4}];
odomdata1 = odomdata - Join[{ConstantArray[odomdata[[1, 1]], Length[odomdata]]}ᵀ,
    ConstantArray[0, {Length[odomdata], 2}], 2];
knownpose1 = knownpose - Join[{ConstantArray[odomdata[[1, 1]], Length[knownpose]]}ᵀ,
    ConstantArray[0, {Length[knownpose], 3}], 2];
odominterp = {Interpolation[Thread[{odomdata1[[All, 1]], odomdata1[[All, 2]]}],
     InterpolationOrder → 1][s], Interpolation[
     Thread[{odomdata1[[All, 1]], odomdata1[[All, 3]]}], InterpolationOrder → 1][s]};
knowninterp = {Interpolation[Thread[{knownpose1[[All, 1]], knownpose1[[All, 2]]}],
     InterpolationOrder → 1][s], Interpolation[
     Thread[{knownpose1[[All, 1]], knownpose1[[All, 3]]}], InterpolationOrder → 1][s],
   Interpolation[Thread[{knownpose1[[All, 1]], knownpose1[[All, 4]]}],
     InterpolationOrder → 1][s]};
groundtruth[t_] := knowninterp /. s → t
modeldata[t_] := odominterp /. s → t
```

## ■ Build Training & Testing Datasets

```
(* Training data timestep and averaging timestep *)
dt = 2; del = 0.05;
(* Training dataset timestamps *)
t0 = 100;  tf = 700;
(* Testing dataset timestamps *)
t0test = 1500;  tftest = 2100;

Rbw[x_, θ_] := x.{{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}};
Rwb[x_, θ_] := {{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}}.x;

(* Scaling input and output to (0,1) *)
xscaling = {{20, -5.65}, {1 / 4, 0.5}};
yscaling = {{8, -1.5}, {3, 0.5}, {1 / 4, 0.5}};

(* Compute input & output training sets and input testing set *)
xtrain =
  Table[{xscaling[[1, 1]] Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) dt +
      xscaling[[1, 2]], xscaling[[2, 1]] Sum[modeldata[j][[2]], {j, t, t + dt, del}] /
        (dt / del + 1) dt + xscaling[[2, 2]]}, {t, t0, tf, dt}];
ytrain = Table[{yscaling[[1, 1]] Rbw[groundtruth[t + dt][[1 ;; 2]] - groundtruth[t][[1 ;; 2]],
          groundtruth[t][[3]]][[1]] + yscaling[[1, 2]],
     yscaling[[2, 1]] Rbw[groundtruth[t + dt][[1 ;; 2]] - groundtruth[t][[1 ;; 2]],
          groundtruth[t][[3]]][[2]] + yscaling[[2, 2]],
     yscaling[[3, 1]] Which[-Pi ≤ groundtruth[t + dt][[3]] - groundtruth[t][[3]] ≤ Pi,
         groundtruth[t + dt][[3]] - groundtruth[t][[3]],
         (groundtruth[t + dt][[3]] - groundtruth[t][[3]]) > Pi,
         groundtruth[t + dt][[3]] - groundtruth[t][[3]] - 2 Pi,
         (groundtruth[t + dt][[3]] - groundtruth[t][[3]]) < -Pi, groundtruth[t + dt][[3]] -
          groundtruth[t][[3]] + 2 Pi] + yscaling[[3, 2]]}, {t, t0, tf, dt}];
xtest = Table[{xscaling[[1, 1]] Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) dt +
      xscaling[[1, 2]],
     xscaling[[2, 1]] Sum[modeldata[j][[2]], {j, t, t + dt, del}] / (dt / del + 1) dt +
      xscaling[[2, 2]]}, {t, t0test, tftest, dt}];

(* Output plots of training data *)
GraphicsGrid[Table[ListPlot[Thread[{xtrain[[All, a]], ytrain[[All, b]]}],
    PlotRange → {{0, 1}, {0, 1}}], {a, 1, 2}, {b, 1, 3}]]
```

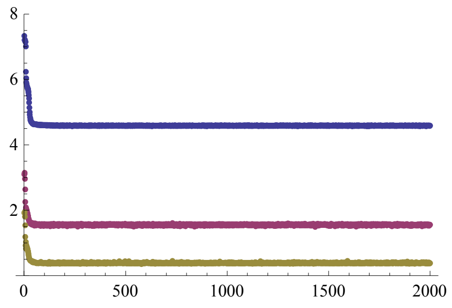


## ■ Neural Network Backpropegation Algorithm

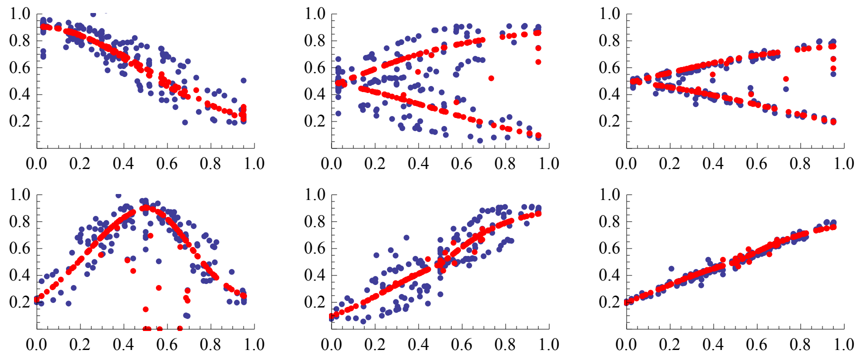
```
(* Neural Net Parameters *)
in = 2;
out = 3;
hid = 4;

η = 1;
α = 0;
sig[y_] := 1 / (1 + Exp[- y]);

(* Initialize random weights *)
w[1] = RandomReal[{-0.1, 0.1}, {hid, in}];
w[2] = RandomReal[{-0.1, 0.1}, {out, hid}];
Δw[1] = 0;
Δw[2] = 0;

(* Stochastic gradient descent formulation *)
For[n = 1, n < 2000, n++,
  order = RandomSample[Table[i, {i, 1, Length[xtrain]}]];
  For[i = 1, i ≤ Length[xtrain], i++,
   x[1] = xtrain[[order[[i]]]];
   y = ytrain[[order[[i]]]];

   x[2] = Table[sig[x[1].w[1][[k]]], {k, 1, hid}];
   x[3] = Table[sig[x[2].w[2][[k]]], {k, 1, out}];

   δ[3] = Table[x[3][[k]] (1 - x[3][[k]]) (y[[k]] - x[3][[k]]), {k, 1, out}];
   δ[2] = Table[x[2][[k]] (1 - x[2][[k]]) (w[2][[All, k]].δ[3]), {k, 1, hid}];

   Δw[2] = Table[η δ[3][[j]] x[2][[k]], {j, 1, out}, {k, 1, hid}] + α Δw[2];
   Δw[1] = Table[η δ[2][[j]] x[1][[k]], {j, 1, hid}, {k, 1, in}] + α Δw[1];
   w[2] += Δw[2];
   w[1] += Δw[1];

   err[i] = Table[Norm[y[[k]] - x[3][[k]]], {k, 1, out}];
  ];
  error[n] = Table[Norm[Table[err[i][[m]], {i, 1, Length[xtrain]}]], {m, 1, out}];
 ];
(* Learned function using weights derived from the neural network *)
output[x_] := Table[sig[Table[sig[x.w[1][[k]]], {k, 1, hid}].w[2][[k]]], {k, 1, out}]

(* Saved Weights from reported trial *)
w[1] = {{0.21330859461453386, -2.182284932023064},
    {-8.927196359036031, -4.851646501308502}, {2.558008801503831, 0.0788159832243044},
 {1.7923314621072692, 8.40078944455011}};
w[2] = {{2.2005261772113, -8.220568043155712, -9.363556131738397, 7.255621485516451},
    {-9.159651108985003, 0.8165804887279737, 2.0362235495862095, 1.1399772777398702},
 {-5.781968386706752, 0.5410942259399777, 1.238882067198802, 0.7732224084238557}};
```

## ■ Plots & Analysis

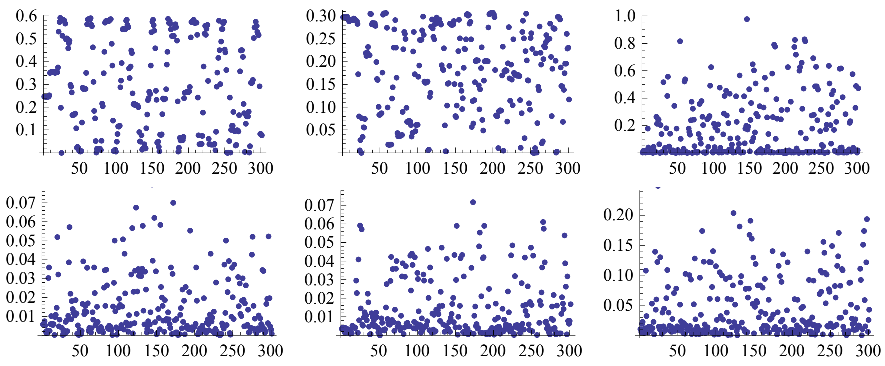
```
(* Convergence Plot in scaled coordinates *)
ListPlot[Table[Table[error[i][[j]], {i, 1, n - 1}], {j, 1, 3}], PlotRange → {0, 8}]
```



```
GraphicsGrid[Table[Show[ListPlot[
    Thread[{xtrain[[All, a]], ytrain[[All, b]]}], PlotRange → {{0, 1}, {0, 1}}], ListPlot[
    Thread[{xtrain[[All, a]], Table[output[xtrain[[i]]][[b]], {i, 1, Length[xtrain]}]}],
    PlotStyle → Red]], {a, 1, 2}, {b, 1, 3}]]
```



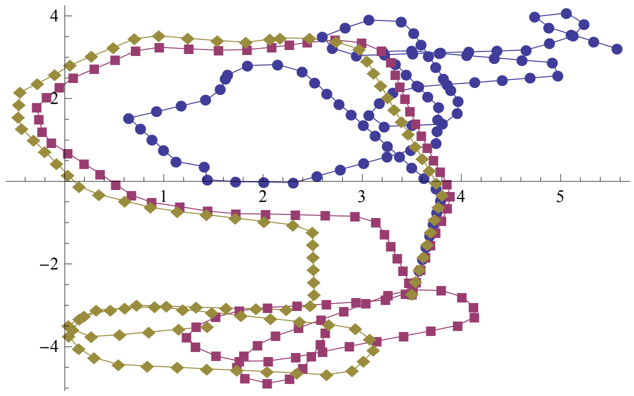
```
(* Compute error between model, training,
testing data sets and the groundtruth from step to step *)
modelerror = Table[{
     Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1)
      Cos[groundtruth[t0 + (i - 1) dt][[3]]] dt,
     Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1)
      Sin[groundtruth[t0 + (i - 1) dt][[3]]] dt,
     Sum[modeldata[j][[2]], {j, t, t + dt, del}] / (dt / del + 1) dt}, {t, t0, tf, dt}] -
   Table[{
     groundtruth[t0 + i dt][[1]] - groundtruth[t0 + (i - 1) dt][[1]],
     groundtruth[t0 + i dt][[2]] - groundtruth[t0 + (i - 1) dt][[2]],
     Which[-Pi ≤ groundtruth[t0 + i dt][[3]] - groundtruth[t0 + (i - 1) dt][[3]] ≤ Pi,
      groundtruth[t0 + i dt][[3]] - groundtruth[t0 + (i - 1) dt][[3]],
      (groundtruth[t0 + i dt][[3]] - groundtruth[t0 + (i - 1) dt][[3]]) > Pi,
      groundtruth[t0 + i dt][[3]] - groundtruth[t0 + (i - 1) dt][[3]] - 2 Pi,
      (groundtruth[t0 + i dt][[3]] - groundtruth[t0 + (i - 1) dt][[3]]) < -Pi,
      groundtruth[t0 + i dt][[3]] - groundtruth[t0 + (i - 1) dt][[3]] +
       2 Pi]}, {i, 1, Length[xtrain]}];
trainerror = Table[Append[
     Rwb[{(output[xtrain[[(t - t0) / dt + 1]]][[1]] - yscaling[[1, 2]]) / yscaling[[1, 1]],
        (output[xtrain[[(t - t0) / dt + 1]]][[2]] - yscaling[[2, 2]]) / yscaling[[2, 1]]},
      groundtruth[t][[3]]],
     (output[xtrain[[(t - t0) / dt + 1]]][[3]] - yscaling[[3, 2]]) / yscaling[[3, 1]]],
    {t, t0, tf, dt}] -
   Table[{
     groundtruth[t + dt] [[1]] - groundtruth[t] [[1]],
     groundtruth[t + dt] [[2]] - groundtruth[t] [[2]],
     Which[- Pi ≤ groundtruth[t + dt] [[3]] - groundtruth[t] [[3]] ≤ Pi,
      groundtruth[t + dt] [[3]] - groundtruth[t] [[3]],
      (groundtruth[t + dt] [[3]] - groundtruth[t] [[3]]) > Pi,
      groundtruth[t + dt] [[3]] - groundtruth[t] [[3]] - 2 Pi,
      (groundtruth[t + dt] [[3]] - groundtruth[t] [[3]]) < - Pi,
      groundtruth[t + dt] [[3]] - groundtruth[t] [[3]] + 2 Pi]}, {t, t0, tf, dt}];
testerror = Table[Append[
     Rwb[{(output[xtest[[(t - t0test) / dt + 1]]] [[1]] - yscaling[[1, 2]]) / yscaling[[1, 1]],
        (output[xtest[[(t - t0test) / dt + 1]]] [[2]] - yscaling[[2, 2]]) / yscaling[[2, 1]]},
      groundtruth[t] [[3]]],
     (output[xtest[[(t - t0test) / dt + 1]]] [[3]] - yscaling[[3, 2]]) / yscaling[[3, 1]]],
    {t, t0test, tftest, dt}] -
   Table[{
     groundtruth[t + dt] [[1]] - groundtruth[t] [[1]],
     groundtruth[t + dt] [[2]] - groundtruth[t] [[2]],
     Which[- Pi ≤ groundtruth[t + dt] [[3]] - groundtruth[t] [[3]] ≤ Pi,
      groundtruth[t + dt] [[3]] - groundtruth[t] [[3]],
      (groundtruth[t + dt] [[3]] - groundtruth[t] [[3]]) > Pi,
      groundtruth[t + dt] [[3]] - groundtruth[t] [[3]] - 2 Pi,
      (groundtruth[t + dt] [[3]] - groundtruth[t] [[3]]) < - Pi,
      groundtruth[t + dt] [[3]] - groundtruth[t] [[3]] + 2 Pi]}, {t, t0test, tftest, dt}];

(* Ouput plots of error norm at each training point *)
GraphicsGrid[
 {Table[ListPlot[Table[Norm[modelerror[[j, i]]], {j, 1, Length[modelerror]}]], {i, 1, 3}],
  Table[ListPlot[Table[Norm[trainerror[[j, i]]], {j, 1, Length[trainerror]}]], {i, 1, 3}]}]
```



```
(* Output norms of model, training, and testing error *)
GridBox[{Table[Norm[modelerror[[All, i]]], {i, 1, 3}], Table[Norm[trainerror[[All, i]]],
    {i, 1, 3}], Table[Norm[testerror[[All, i]]], {i, 1, 3}]}] // DisplayForm
```

```
 6.16019    3.56681    4.76227
 0.45143    0.616673   1.62608
0.477586    0.53162    1.63782
```

```
(* Plot of the dead-
 reckoning simplified model and learned model vs. the groundtruth on training data *)
x[t0] = groundtruth[t0];
x1[t0] = groundtruth[t0];
For[t = t0, t < tf, t += dt,
  x[t + dt] = {Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) Cos[x[t][[3]]] dt,
     Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) Sin[x[t][[3]]] dt,
     Sum[modeldata[j][[2]], {j, t, t + dt, del}] / (dt / del + 1) dt} + x[t];
  x1[t + dt] = Append[Rwb[{(output[xtrain[[(t - t0) / dt + 1]]][[1]] - yscaling[[1, 2]]) /
         yscaling[[1, 1]], (output[xtrain[[(t - t0) / dt + 1]]][[2]] - yscaling[[2, 2]]) /
         yscaling[[2, 1]]}, x1[t][[3]]],
     (output[xtrain[[(t - t0) / dt + 1]]][[3]] - yscaling[[3, 2]]) / yscaling[[3, 1]]] + x1[t];
 ];
ListPlot[{Table[x[s][[1 ;; 2]], {s, t0, 300, dt}], Table[x1[s][[1 ;; 2]], {s, t0, 300, dt}],
  Table[groundtruth[s][[1 ;; 2]], {s, t0, 300, dt}]}, Joined → True, PlotMarkers → Automatic]
```



```
(* Plot of the dead-
 reckoning simplified model and learned model vs. the groundtruth on testing data *)
t0test = 1800; tftest = 1950;
xtest = Table[
   {xscaling[[1, 1]] Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) dt + xscaling[[
      1, 2]], xscaling[[2, 1]] Sum[modeldata[j][[2]], {j, t, t + dt, del}] / (dt / del + 1) dt +
     xscaling[[2, 2]]}, {t, t0test, tftest, dt}];
x[t0test] = groundtruth[t0test];
x1[t0test] = groundtruth[t0test];
For[t = t0test, t < tftest, t += dt,
  x[t + dt] = {Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) Cos[x[t][[3]]] dt,
     Sum[modeldata[j][[1]], {j, t, t + dt, del}] / (dt / del + 1) Sin[x[t][[3]]] dt,
     Sum[modeldata[j][[2]], {j, t, t + dt, del}] / (dt / del + 1) dt} + x[t];
  x1[t + dt] = Append[Rwb[{(output[xtest[[(t - t0test) / dt + 1]]][[1]] - yscaling[[1, 2]]) /
        yscaling[[1, 1]], (output[xtest[[(t - t0test) / dt + 1]]][[2]] - yscaling[[2, 2]]) /
        yscaling[[2, 1]]}, x1[t][[3]]],
     (output[xtest[[(t - t0test) / dt + 1]]][[3]] - yscaling[[3, 2]]) /
      yscaling[[3, 1]]] + x1[t];
 ];
ListPlot[{Table[x[s][[1 ;; 2]], {s, t0test, tftest, dt}],
  Table[x1[s][[1 ;; 2]], {s, t0test, tftest, dt}],
  Table[groundtruth[s][[1 ;; 2]], {s, t0test, tftest, dt}]},
 Joined → True, PlotMarkers → Automatic]
```