Utilizing a Multi-Layer Perceptron to Find Initial Conditions of a Damped One Dimensional Spring

Nick Schwartz

Abstract

Background: The one dimensional spring is a well known physics problem that has multiple different analyzable cases making it ideal for a proof of concept machine learning project.

Purpose: Create a model that can accurately predict the k and b values for a damped spring. All three cases of damping are possible with the generated data.

Methods: The model and the optimizer will be varied to determine the optimal one. The data was created using Odeint.

Results: Adam and Adamax were the most effective optimizers, both tested models were comparably effective. The k and b values were able to be accurately predicted.

Conclusion: A proof of concept was given, but a much more complex model may still yet not be possible with this approach.

Background

What is a Multilayer Perceptron?

A multilayer perceptron (MLP) solves the XOR problem. The XOR problem is that given four points there is a set that is not linearly separable (DeepAI). The MLP is able to classify such a set of data and therefore is able to do much more complex regression and classification tasks.

A MLP works through a feed forward architecture where the input layer is fed through each of the hidden layers. Each hidden layer has a given activation function and weights associated with it. The weights of the hidden layers are what is trained as a model undergoes training on a dataset. The model learns what is ”good” and what is ”bad” through the loss function. A loss function for regression can be mean squared error. After a model runs through the data once and calculates the loss, it adjusts the weights of the hidden layer accordingly.

The Differential Equation

For damped harmonic motion the differential equation will be:

$$0 = m\ddot{x} + b\dot{x} + kx \tag{1}$$

This is the equation that will be passed to odeint from Scipy in order to generate the data. Since m scales the data due to affecting both terms equally, it will be held at a constant value of 1. This won't have a large impact on how the model performs for new real world data because the data should be normalized like the data being trained here will be.

The one dimensional spring has multiple cases, over, under, and critical damping. When analyzing the system being able to predict the kind of damping is important. They are predicted from the relationship between b^2 and $4mk$ Below are the solutions, these were checked with Taylor, J. R. (2005):

Case 1: Critical Damping

$b^2 = 4mk$

Critical Damping has the general solution as follows:

$$x(t) = C_1 e^{\lambda_1 t} + C_2 t e^{\lambda_1 t} \tag{2}$$

$$\text{with } \lambda_1 = -\frac{b}{2m}$$

Case 2: Over Damping

$b^2 > 4mk$

Over Damping has the general solution as follows:

$$x(t) = C_1 e^{\lambda_1 t} + C_2 e^{\lambda_2 t} \tag{3}$$

$$\text{with } \lambda_1 = -\frac{-b + \sqrt{b^2 - 4mk}}{2m}, \ \lambda_2 = -\frac{-b - \sqrt{b^2 - 4mk}}{2m}$$

Case 3: Under Damping

$b^2 < 4mk$

Under Damping has the general solution as follows:

$$x(t) = C_1 e^{\alpha\, t} \cos(\beta\, t) + C_2 e^{\alpha\, t} \sin(\beta\, t) \tag{4}$$

$$\text{with } \alpha = -\frac{b}{2m}, \ \beta = \frac{\sqrt{4mk - b^2}}{2m}$$

In all three cases the general solutions are determined by the m, k, and b values. Therefore with this system, training a model to predict these parameters can be used to classify damping. The m value being the least important because it is a scalar that acts as a scaling variable.

This sort of problem is referred to in the literature by "inverse problems" where you take physical data and try to discover unknown parameters of a physical system (Hao et al. 2022).

Motivation

The goal of this project is to be able to predict the k (spring constant) and b (damping constant) values of a damped one dimensional spring. In this project due to time constraints the chosen problem is admittedly simplified, but still useful in illustrating that physical parameters can be found from position and velocity data. In order to create the Neural Network, tensorflow-keras will be utilized. The purpose of this goal is to be able to look at real life spring analogous systems and predict the k and b values of those systems. In order to do this a large plethora of varied data with varied starting constants will be needed. Since such a dataset is not readily available, it will need to be created. The creation of the dataset here adds bias to the model, but unfortunately there is not another option given the lack of available data.

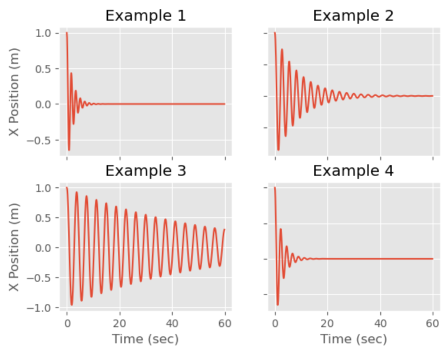
The benefit to doing this project is that such a model could be used to analyze experimental data. Essentially, the model would be able to look at observed behavior and then predict the experimental values that created the behavior. This is important because it can reduce the overhead needed to analyze experimental results, saving both time and money. The long term goal of the project is to showcase that inverse problems can be useful in real life applications.

Methods

Creation of the Dataset

In order to create a dataset large enough, about 10,000 entries, a method will need to be devised in order to create varied, non-duplicate, and limitedly biased data. Using previous experiences of creating generated spring data from previous projects covering damped harmonic motion, reasonable k values to be between 1 and 15 and reasonable b values to be between 0 and 1. From these ranges of values numpy arrays were arranged with small step sizes to create a total possible number of combinations totaling 30,000. Through this, random k and b values can be randomly selected and plugged into odeint to generate varied data of time, position, and velocity data for each of the 10,000 randomly sampled constants.

Below is an example of some of the data behaviors:



Four randomly selected sets of data taken from the data set; see Appendix for the code that generated this plot.

The above shows that there are a varied amount of behaviors in the data set, in the appendix you can run the code used to generate this plot $(graph_random_data())$ and see the varied behaviors in the dataset. Importantly, the data created has bias because of the constrained k and b values with an assumed m value of one. The m value is not as impactful because it functions as a scaling of the dataset, and the input data is normalized before being input into the model regardless. Therefore, assuming the real world data being predicted is within the no damping to heavy damping cases with any mass it should be able to be estimated when considering the data being normalized.

Model Selection

For this project there will be two candidate models that will be compared using 5 k-fold cross validation. The first will be a standard MLP which will have multiple layers with descending node numbers. The second model will have the same amount of layers as the first, but every layer except the last one will be fully connected layers. These models were chosen for the simplicity due to hardware and time concerns. More models would increase runtime and comparisons by an incredible amount

The "best model" will be chosen based on consistency and overall loss performance. Due to the size the of the loss being expected to me less than 1, RMSE error will be used because MSE would make the loss seem smaller than it actually is, where RMSE shows the actually average loss across all the predictions. The best model will have an associated best optimizer that will be selected for each model.

The other thing to keep in mind here is that the magnitude differences between the K and B value will drag up the error of the b values. When looking individually at the numbers, the error separately may be larger or smaller than the overall RMSE respectively. An example of the predicted versus real values will be printed in the appendix. In the end the RMSE will be a good enough estimate of the model efficacy even if it doesn't completely get the context of the difference in magnitude of the two physical parameters.

Loss Function Creation

Since the data is in a somewhat odd format in the output, the RMSE function will be created to evaluate the predictions. This ensures that the loss is what it is expected to be. The function can be seen in the .py file with the name "$calc_test_RMSE$". This loss function was selected because the error is expected to be less than one. For errors less than one, the mean squared error will make the loss seem smaller than it actually is. Therefore to get a better understanding of how far off the model is with its' predictions, RMSE is the better choice that is just a converted version of MSE which is the standard regression loss function.

Optimizer Selections

In order to verify that the best optimizer for the project has been chosen, all non-experimental and applicable optimizers provided by the keras package will be tested on the two models. Explanations of the models are below:

Adam: stochastic gradient descent (SGD) with adaptive first order and second order moments

Adadelta: SGD method that is used to address two drawbacks: the continual decay of learning rates through training, and the need for a manually selected global learning rate

Adamax: variant of adam based on the infinity norm

Adagrad: parameter specific learning rates

SGD: gradient descent with momentum optimizer

The rest of the optimizers provided by keras are either labeled as experimental or not applicible to this specific problem. All the optimizers will be initialized with the same learning rate, and trained for the same amount of epochs for each k-fold test.

The best optimizer for each tested model will be chosen based on both the best loss and the consistency across the k-fold validation. The reasoning for choosing not only on the best score is that there can be outliers based on how the data is trained, so for this kind of application, consistency is much more important when choosing the best optimizer.

ML Package and Hardware

The machine learning package to be used for this project is tensorflow. The reason for this is two-fold, with Pytorch the data needs to be converted to pytorch tensors. However, Tensorflow can be used simply with Numpy arrays and the data that was created for this project is in the form of Numpy arrays because it was generated with Odeint from Scipy. The other is the obvious ability for tensorflow to utilize both Nvidia GPUs and the apple M1 chip which has GPU capabilities for machine learning as well.

The project is developed on both a m1 macbook air, and a windows desktop with a RTX3070 and i-7 12700-kf cpu.

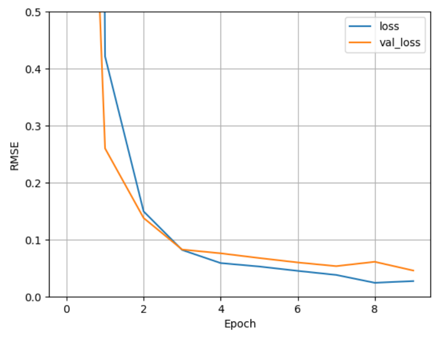
On the windows machine, tensorflow 2.10.0 tensorflow-gpu 2.10.0, cudnn 8.1.0.77, and cudatoolkit 11.2.2 which are all compatible with the RTX3070 Nvidia graphics card. Depending on graphics card different versions may be needed or the GPU component of tensorflow may not even work.

On the apple machine, tensorflow-macos 2.9.0 and tensorflow-metal 0.5 were utilized on the m1 macbook air. The m1 chip is a CPU, but the architecture allows for GPU like processing when it comes to machine learning. Tensorflow has this integrated, but the version of metal and tensorflow-macos need to be compatible with the overall operating system as well.

Both machines utilize GPU or GPU-like components, but the windows machine with the RTX 3070 runs much faster. Therefore, unless using a higher end macbook, a dedicated NVIDIA GPU is recommended for running the code in the appendix.

Results

Below is an example of how the model is trained. As can be seen below it very quickly gets to less than 1 RMSE, but due to the scale of the graph the incremental changes from about 0.4 to around 0.2 RMSE cannot be seen very clearly. When testing on the testing data, the model does not start to overfit until after about 300 epochs depending heavily on the optimizer chosen. Some optimizers performed wildly better than others regardless of the amount of epochs trained. This is probably due to some of the optimizers not being a good choice for this problem. None of the tested optimizers performed terribly, but as noted, two performed way better.



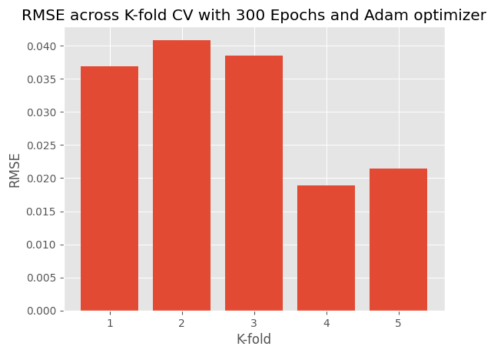
Example of the loss and validated loss with a basic MLP trained for 10 epochs.

Above the plot shows a sharp decrease in loss and then a slow burn of a decrease in loss afterwards as epochs reach the 100 to 500 range. Due to this, the amount of epochs was capped at 500 due to overfitting concerns.

Both models were tested for the best RMSE across common epoch numbers of 100, 200, 300, 400, and 500. Below are the two best outcomes for both the fully connected and descending model.

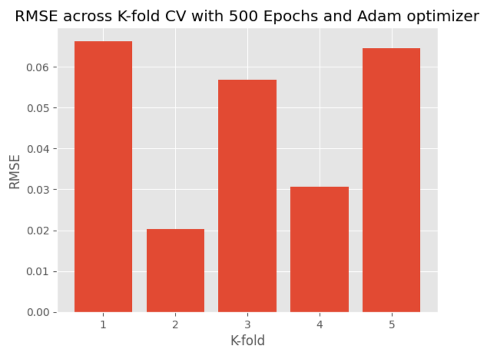
The fully connected model with the Adamax optimizer performed the best when trained for 500 epochs and the descending model performed similarly but minorly worse with the Adam optimizer when trained for 300 epochs. Both optimizers results are shown below:

The descending model with the Adam optimizer performed the best when trained for 300 epochs and is shown below:



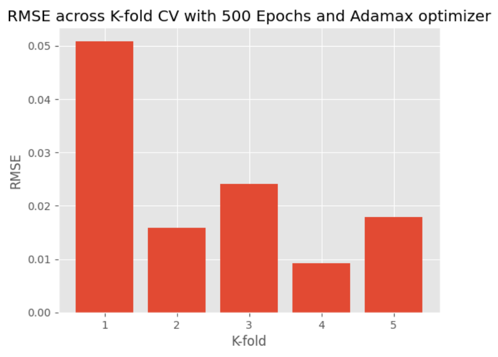
The best cross validated performance of the Adam optimizer with the descending model

The fully connected model with the Adam optimizer performed the best when trained for 500 epochs and is shown below:



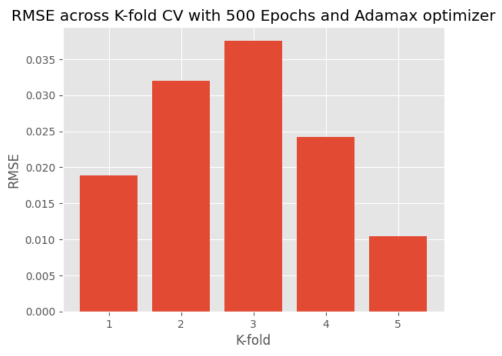
The best cross validated performance of the Adam optimizer with the fully connected model

The descending model with the Adamax optimizer performed the best when trained for 500 epochs and is shown below:



The best cross validated performance of the Adamax optimizer with the descending model

The fully connected model with the Adamax optimizer performed the best when trained for 500 epochs and is shown below:



The best cross validated performance of the Adamax optimizer with the fully connected model

All of these are fairly similar in their great performance in terms of RMSE. They could all be used as the deployment mode, but if a model must be chosen, the Adamax optimizer with the fully connected model would be the one chosen.

The rest of the outcomes of the tests of the epoch training can be seen in the Appendix along with the runtimes from utilizing the CUDA functionality of the NVIDIA RTX 3070. They are mostly not comparable in terms of RMSE where Adam and Adamax are far and above the rest of the pack.

From this, both models' best loss scores are fairly similar across the different levels, but the descending node model consistently had better scores across the cross validation levels. The Adam and Adamax optimizer is by and far away the best of the tested, and considering it has an RMSE of ¡0.04 at the worst, other optimizers should not be considered utilizing for this project since minimal if any improvement can be made on this due to the size of the parameters being estimated by the model. The Adam and Adamax performed well and above the rest of the optimizers for both models. All other tests and graphs and be seen in the appendix. The majority of the other optimizers besides Adam and Adamax performed similarly to each other across the tested epochs.

Discussion

From the exploration of this project, the method laid out in Hao et al. 2022, an inverse problem was successfully solved, albeit in a simplified case due to time constraints. For the 1d spring, the most important physical constants of a specific problem are the spring constant k and the damping constant b. In order to have data that resembles traditional spring behavior, the k and b values were constrained during data creation in order to not get behaviors that would not typically be seen if this model were to be applied to experimental data. The MLP was able to very accurately predict the k and b values of this simulated system. From above, the RMSE is less than 0.04 across a 5 k-fold cross validation when trained for 500 epochs.

The Adam and Adamax optimizers worked across the board more efficiently and more effectively than the other optimizers. Even when training for fewer epochs (100 v 300) the Adam optimizer performed better in terms of RMSE. As can be seen in the Appendix, the Adam model also doesn't overfit when training for longer (900 epochs), however, there was almost no improvement in RMSE when considering the increase in training time. Therefore, if computational cost is an issue, Adam and or Adamax is the best of the optimizers tested because it can get similar results with less computational cost.

The natural extension of this project would be using real world data to predict more complex systems. Theoretically, a model could be pre-trained on a similar simulated dataset in order to have it be more effective at predicting real world data. Essentially, train on simulated data, train on known real data, and then estimate the target unknown data with similar behaviors. This would allow for the efficient evaluation of experimental data as long as you know beforehand what the system is.

Something else to consider here is computational cost when finding the best optimizer. When running all the k-fold validiations, it took tensorflow utilizing the cuda cores of an RTX3070 over 10 hours to complete. Without access to a GPU, TPU, or similar system, the process of hyperparameter tuning would be very computationally expensive. Furthermore, with a more complex system, a more complex model may be needed. The model tested in this project had about 1 million trainable parameters, but it can obviously get much more complex than that which would heavily impact computational cost.

Conclusion

The goal of the project was achieved, a model was trained that could very accurately predict the one dimensional spring with damping case. The model was able to predict physical constants just based on position and velocity behavior. Utilizing fully connected layers is less effective than more complex models from the testing of different MLP architectures and optimizers. Starting with a higher number of nodes and steadily decreasing them continuously performed better with the same amount of epochs and optimizers. The model would be able to classify the damping cases as well since that is a simple inequality of $b^2 = 4mk$.

This project is a proof of concept of what can be done with deep learning in predicting experimental constants, and with the proper datasets, more complex experimental constants could theoretically be predicted numerically using machine learning.

Something the project failed to do is prove a model could be trained for a more complex system. In order to create data Odeint or a similar differential equation solver must be used. This limits the scope to first order and first order coupled differential equations which are all that Odeint can work with. However, if there was data created a different way with known solved values a very similar approach should work.

Citations

Hao, Z., Liu, S., Zhang, Y., Ying, C., Feng, Y., Su, H., & Zhu, J. (2022). Physics-Informed Machine Learning: A Survey on Problems, Methods and Applications. arXiv preprint arXiv:2211.08064.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Taylor, J. R. (2005). Classical Mechanics. University Science Books.

DeepAI. “Multilayer Perceptron.” DeepAI, DeepAI, 17 May 2019, https://deepai.org/machine-learning-glossary-and-terms/multilayer-perceptron.

Appendix

To see the code, got to the following Github repository:
https://github.com/schwartznicholas/SpringInverseProblem