ERDC/ITL TR-23-4

Information Technology Laboratory

US Army Corps
of Engineers®
Engineer Research and
Development Center

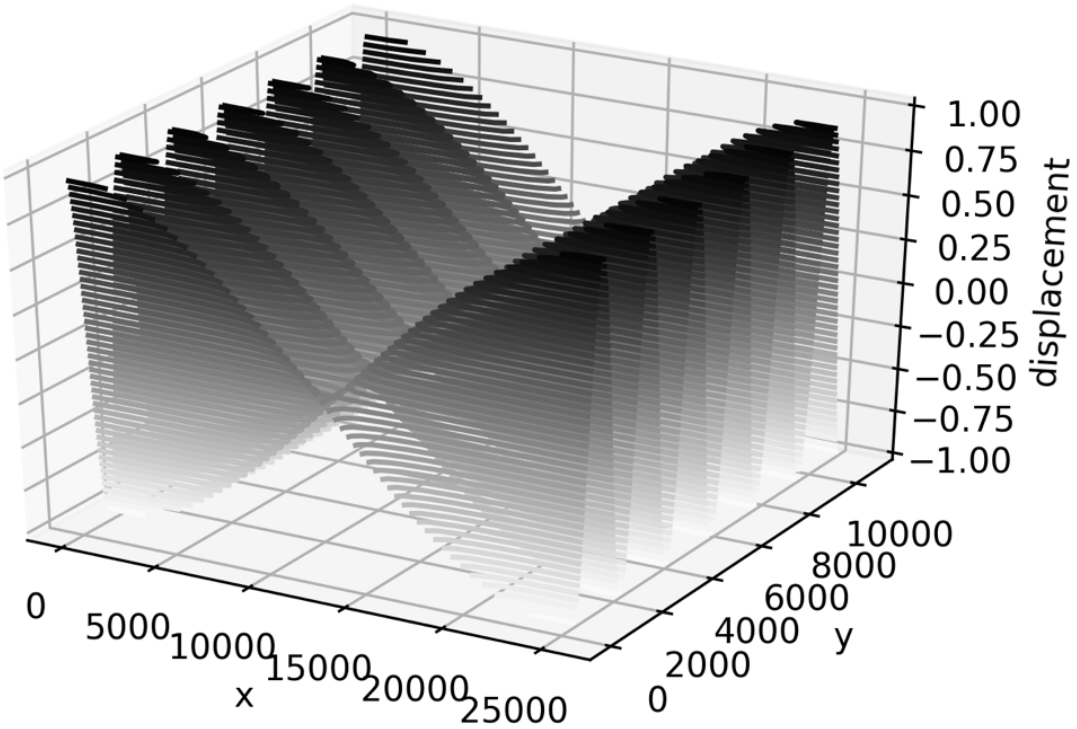
ERDC
ENGINEER RESEARCH & DEVELOPMENT CENTER

Engineered Resilient Systems

Scaling and Sensitivity Analysis of Machine Learning Regression on Periodic Functions

Corey J. Trahan and Peter G. Rivera

August 2023



The US Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdc.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at https://erdclibrary.on.worldcat.org/discovery.

ERDC/ITL TR-23-4
August 2023

Scaling and Sensitivity Analysis of Machine Learning Regression on Periodic Functions

Corey J. Trahan and Peter G. Rivera

Information Technology Laboratory
US Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Final report

Prepared for Headquarters, US Army Corps of Engineers
Washington, DC 20314-1000

Under Program Element 0603465, Project AL3

Abstract

In this report we document the scalability and sensitivity of machine learning (ML) regression on a periodic, highly oscillating, and C^{∞} function. This work is motivated by the need to use ML regression on periodic problems such as tidal propagation. In this work, TensorFlow is used to investigate the machine scalability of a periodic function from one to three dimensions. Wall clock times for each dimension were calculated for a range of layers, neurons, and learning rates to further investigate the sensitivity of the ML regression to these parameters. Lastly, the stochastic gradient descent and Adam optimizers wall clock timings and sensitivities were compared.

Contents

Figures

Preface

This report was prepared for the US Army Corps of Engineers and is a deliverable product under Program Element 0603465, Project AL3, Engineered Resilient Systems (ERS) Program, Data Analytics Work Package, Physics Informed Machine Learning Work Unit. Dr. Ian Dettwiller was the program manager, and Dr. Robert M. Wallace was the lead technical director of the ERS program.

The work was performed by the Computational Analysis Branch of the Computational Science and Engineering Division, US Army Engineer Research and Development Center, Information Technology Laboratory (ERDC-ITL). At the time of publication, Mr. David Stuart was branch chief, and Dr. Jeffrey L. Hensley was division chief. The deputy director of ITL was Dr. Jackie S. Pettway, and the director was Dr. David A. Horner.

The commander of ERDC was COL Christian Patterson, and the director was Dr. David W. Pittman.

This page intentionally left blank.

1 Introduction

Regression analysis is a predictive modeling technique that investigates the relationship between dependent (target) and independent (predictor) variables. This technique is used for forecasting, time-series modeling, and finding the causal effect between variables. Regression analysis is ubiquitous in the scientific world as it is an important tool for modeling and analyzing data. The standard regression technique involves fitting a curve/line to data points in a manner that minimizes the distance differences between the data points and the curve. By doing this, significant relationships between dependent variables and independent variables can be obtained, as can strengths of impact of multiple independent variables on dependent variables.

Machine learning (ML), a subset of artificial intelligence, is the study of computer algorithms that improve automatically through experience. ML algorithms often build a "supervised" mathematical model based on sample data, known as "training data," in order to make predictions or decisions without being explicitly programmed to do so. ML algorithms are used in a wide variety of applications and are typically divided into two types of applications: classification and regression. This technical report focuses on ML regression. Specifically, we investigate the scalability of ML regression with respect to dimensionality of a periodic or highly oscillating function along with its sensitivity to a number of ML options/parameters. Highly oscillating periodic functions can be difficult to regress because of their non-local nature and C^{∞} differentiability, yet they are commonly found in science and very common in ocean dynamics (e.g., tidal frequencies).

1.1 Background

Periodic flows are ubiquitous in both civil and military applications and can range from slow to rapid oscillations relative to the time scale of analysis. For example, when investigating ocean dynamics for storm surge applications, one must include the ocean's periodic tidal propagation, which, depending on the scale of the analysis (if years, for example), can be a highly oscillating periodic function of time. Additionally, physics-informed machine learning (PIML) has recently been recently investigated as a means of providing regressions constrained by the system's physics to

mitigate unnatural fits and low sampling region errors. For example, the Navier Stokes and shallow water equations or their turbulence equations have been utilized to increase the accuracy and speed of classic ML regression methods. This is particularly the case in areas with very little or no data. With these recent PIML efforts and the ubiquity of period functions, it is vital that we have some basic idea for scalability and hyper-parameter sensitivity of a straightforward ML regression before adding further complexities.

1.2 Objective

The objective of this report is to investigate the computational efficiency/feasibility and parameter/optimizer sensitivity of an ML regression on a highly oscillating, periodic function as the dimensionality of the function is increased. This effort serves as a precursor to the Engineer Research Development Center (ERDC) physics-informed ML work, wherein regression losses are supplemented with physical equations. The aim here is to get a better understanding for pure regression sensitivities and the computational needs for a periodic function before adding an additional layer of complexity that includes physics.

1.3 Approach

Our approach herein is to investigate ML regression on periodic functions. Periodic functions are common solutions of the hydrostatic Navier-Stokes (shallow water) equations when a water level disturbance occurs in a closed rectangular basin or flume. In this example, the disturbance or displacement propagates back and forth in a periodic fashion along the length of the flume, resulting in a highly oscillating function for the period studied.

All ML regressions in the study were performed using TensorFlow (TF) (Abadi et al. 2015). TF is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is used for both research and production at Google. TF was developed by the Google Brain team for internal Google use and was released under the Apache License 2.0 on 9 November 2015. Here, we use TF for ML applications using neural networks.

With increasing dimensionality, the chosen periodic function is regressed with TF using a range of neurons, layers, and learning rates. Additionally, two optimizers were investigated, the Adam and stochastic gradient descent (SGD) optimizers. The wall clock timings for each of these runs were recorded and compared.

Motivation for this work comes from the investigation of 3D ML regression models for fluid dynamics with applications ranging from airflow over craft to ocean and estuary dynamics. The oscillatory test function presented here, for example, is the analytic solution to a water displacement disturbance propagating back and forth in a closed rectangular basin or flume. The test function resides on an *x-y-t* domain of [0:L] X [0:B] X [0:TF] angled at 45 degrees and given as follows:

$$f(x,y,t) = a * \cos\left(\frac{\pi}{L} * \frac{1}{\sqrt{2}}(x+y)\right) * \cos\left(\frac{\pi}{L} * \sqrt{gHo} * t\right),$$

where

g = 9.8 m/s^2,

a = 1.0 m,

L = 25,600 m,

B = 6,400 m,

TF = 10,800 s, and

Ho = 82.50 m.

In this report, 1D regressions on *f(x)* are investigated with $y = t = 0$, 2D regressions are with *f(x,t)* with $y = 0$, and 3D regressions are on the full *f(x,y,t)* domain.

All ML regressions given in this report were calculated in serial on the ERDC high performance machine, Vulcanite. Vulcanite is equipped with NVIDIA V100 PCIE accelerator nodes and has 192 GB memory/node. A significant speed up will occur over the results in this report if TF is calculated in parallel across any number of GPUs. Thus, all wall clock timings herein are worst case and may be easily decreased. Serial application was used to avoid as much as possible architectural dependencies in the scaling analysis.

2 TensorFlow Regression

2.1 Scaling of the Domain and Solutions

The first attempts at any kind of regression using TF with this function on the angled domain of given size were a complete failure until both the solution and non-rotated domain were scaled unitarily. The domain was scaled by the following:

$$x' = \frac{x - xmin}{Lr}$$
$$y' = \frac{y - ymin}{Br}$$
$$t' = \frac{t}{TF}$$

For the given *f(x,y,t)* with $a = 1$, no solution scaling was required since the range of this function is [−1:1]; however, it was found that when the solution did not have such unitary bounds, TF had trouble converging, and solution scaling was needed.

2.2 TensorFlow Model Setup

Model setup within TF can be complex, and it was found that a number of neural network configurations work to some degree if their parameters are tuned enough. That said, an even larger number of networks were attempted where either no or extremely slow convergence was found, particularly as the dimensionality of the regression increased. All deep neural networks have some number of layers composed of some number of neurons. The number of layers and neurons that make up the network can have a first order effect on the results. A range of layer/neuron configurations was investigated in this study.

In addition to the number of layers and neurons, TF allows tuning of a vast number of ML parameters, which can have great or very little effect on the rate of convergence to solution, depending on the application. The following parameters are commonly tuned for applications and were investigated in this study: activation function, learning rate, optimizer, number of epochs, batch size, and loss definition.

2.2.1 Activation Function

An activation function is a function that is applied to the output of a neural network layer, which is then passed as the input to the next layer. Activation functions are an essential part of neural networks as they provide nonlinearity, without which the neural network reduces to a mere logistic regression model. In this study, the "tanh" activation function was utilized for all layers since the range of this activation function was the same as *f(x,y,t)*, [-1:1]. This activation function is also nonlinear in nature, so layers can be stacked. Additionally, the gradient is stronger for tanh than alternative sigmoid function (i.e., the derivatives are steeper). Like sigmoid, however, tanh also has a vanishing gradient problem, which arises due to the nature of the backpropagation optimization. (Gradients tend to get smaller and smaller as we keep on moving backward.) To mitigate vanishing gradient issues encountered in higher dimensional regressions, batch normalization was used.

2.2.2 Learning Rate

The weights of a neural network must be discovered via an empirical optimization procedure called stochastic gradient descent. The optimization problem addressed by stochastic gradient descent for neural networks is challenging, and the space of solutions (sets of weights) may be composed of many good solutions (called global optima), as well as easy to find, but low in skill solutions (called local optima). The amount of change to the model during each step of this search process, or the step size, is called the "learning rate" and provides perhaps the most important hyperparameter to tune for your neural network in order to achieve good performance on your problem. Of all the parameters that were adjusted in this study, the learning rate was by far the most effective at getting to an accurate fit in a reasonable amount of time. A wide range of learning rates was investigated in this study.

2.2.3 Optimizers

Every time a neural network finishes passing a batch through the network and generating prediction results, it must decide how to use the difference between the results and the values it knows to be true to adjust the weights on the nodes so that the network steps toward a solution. The algorithm that determines that step is known as the optimization algorithm. The most popular optimizers are the (1) SGD and (2) Adams optimizers.

2.2.3.1 Stochastic Gradient Descent Optimizer

SGD is an iterative procedure. At each iteration, a batch of data is randomly sampled from the training set (this is where the stochasticity comes from). The error between the model's prediction and the training labels is then computed. This error, also called the loss, is then differentiated with respect to the model's parameters. These derivatives (or gradients) tell us how we should update each parameter to bring the model closer to predicting the correct label. Iteratively recomputing gradients and applying them to update the model's parameters is what is referred to as the descent. To summarize, the following steps are repeated until the model's performance is satisfactory:

1. Sample a minibatch of training points (x, y) where x is an input and y a label.
2. Compute loss (i.e., error) L (θ, x, y) between the model's prediction f_ θ(x) and label y where θ represents the model parameters.
3. Compute gradient of the loss L (θ, x, y) with respect to the model parameters θ.
4. Multiply these gradients by the learning rate, and apply the product to update model parameters θ.
5. Momentum is essentially a small change to the SGD parameter update so that movement through the parameter space is averaged over multiple time steps. This is done by introducing a velocity component vv. Momentum speeds up movement along directions of strong improvement (loss decrease) and also helps the network avoid local minima. It is intuitively related to the concept of momentum in physics.

Momentum is essentially a small change to the SGD parameter update so that movement through the parameter space is averaged over multiple time steps. This is done by introducing a velocity component v. Momentum speeds up movement along directions of strong improvement (loss decrease) and also helps the network avoid local minima. It is intuitively related to the concept of momentum in physics. The basic SGD algorithm updates as

$$\theta_{t+1} = \theta_t - \eta \nabla l(\theta)$$

where $l(\theta)$ is a loss function, θ are the weights and biases (parameters), and η is the learning rate. With momentum, the SGD update rule is changed to

$$v_{t+1} = uv_t - \eta \nabla l(\theta)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

where u is the momentum parameter that controls how fast the velocity can change and how much the local gradient influences long-term movement, and v is the velocity.

Nesterov momentum is a simple change to normal momentum. Here the gradient term is not computed from the current position θ_t in parameter space but instead from a position $\theta_{intermediate} = \theta_t + uv_t$. This helps because while the gradient term always points in the right direction, the momentum term may not. If the momentum term points in the wrong direction or overshoots, the gradient can still "go back" and correct it in the same update step. The revised parameter update rule is

$$v_{t+1} = uv_t - \eta \nabla l(\theta + uv_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}.$$

2.2.3.2 The Adam Optimizer

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. SGD maintains a single learning rate for all weight updates, and the learning rate does not change during training. In Adam, a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. *This method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients*.

Adam can be viewed as combining the advantages of two other extensions of SGD. Specifically,

- **Adaptive gradient algorithm** that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g., natural language and computer vision problems).
- **Root mean square propagation** that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g., how quickly it is changing). This means the algorithm does well on online and non-stationary (e.g., noisy) problems.

2.2.4 Batch Size

The batch size is the number of input data values that one introduces at once in the model, or the total number of training samples present in a single batch. For a standard ML / deep learning algorithm, choosing a batch size will have an impact on several aspects:

- The bigger the batch size, the more data you will feed at once in a model.
- Thus, RAM memory consumption will be almost linear with batch size, and there will always be a limit based on your system specs and the size of your model, above which your model will overflow.
- The bigger the batch size, the faster you will loop over your dataset *N* times to perform training 2.
- A bigger batch size will slow down your model training speed, meaning that it will take longer for your model to get one single update since that update depends on more data.
- A bigger batch size will have more data to average toward the next update of the model, hence training 2 should be smoother (smoother training/test accuracy curves).

Batch normalization is a technique designed to automatically standardize the inputs to a layer in a deep learning neural network. Once implemented, batch normalization has the effect of dramatically accelerating the training process of a neural network and in some cases improves the performance of the model via a modest regularization effect, helping to prevent vanishing gradients

2.2.5 Number of Epochs

This defines the number times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch comprises one or more batches. The number of epochs is traditionally large, often hundreds or thousands, allowing the learning algorithm to run until the error from the model has been sufficiently minimized. You may see examples of the number of epochs in the literature and in tutorials set to 10, 100, 500, 1,000, and larger. The number of epochs in this study varied depending on dimensionality of the regression.

2.2.6 Loss

Loss is the penalty for a bad prediction. That is, loss is a number indicating how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero; otherwise, the loss is greater. Loss is a fundamental and required input into TF. Just as there are a number of different error estimators in mathematics, there are also a number of ways loss can be measured. The most common loss measure, and the one utilized in this study, is the mean squared error between the samples and the predictions.

2.3 TensorFlow Regression Algorithm

All code in this report was carried out in Python, sometimes within Jupyter notebooks. The general TF Python routine used for this study is as follows:

```
def
ml(nvarIn,nvarOut,nneurons,nlayers,nepochs,activation_function,op
timizer,batch_size):
      inputs = tf.keras.Input(shape=(nvarIn,))
      yr = tf.keras.layers.Dense(nneurons,
activation=activation_function)(inputs)
      lyr = tf.keras.layers.BatchNormalization()(lyr)
      for i in range(0,nlayers-2):
            lyr = tf.keras.layers.Dense(nneurons,
      activation=activation_function)(lyr)
            lyr = tf.keras.layers.BatchNormalization()(lyr)
            outputs = tf.keras.layers.Dense(nvarOut,
      activation='linear')(lyr)
            lyr = tf.keras.layers.BatchNormalization()(lyr)
      model = tf.keras.Model(inputs=inputs, outputs=outputs)
      model.summary()
      model.compile(loss='mse',optimizer=optimizer,metrics=['mae'
])
      hist = model.fit(grid_train, sol_train,
epochs=nepochs,batch_size)
      return model
```

Note above that batch normalization has been used. It was found that without this, convergence became too erratic and sometimes no convergence was achieved. Note here that an MSE (mean squared error) loss was used and an MAE (mean absolute error) was reported back from TF but not used in the loss.

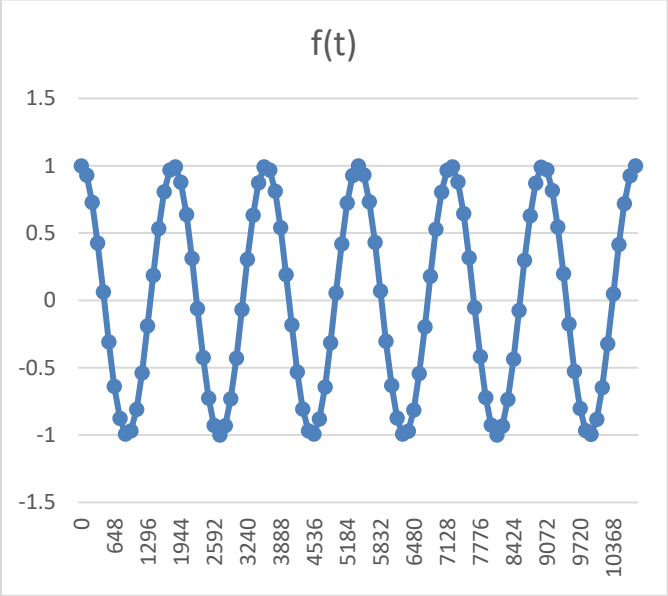
2.4 1D regression results

To begin, a 1D regression was performed on our oscillatory function above at $x = y = 0$, giving

$$f(t) = a * \cos(\frac{\pi}{L} * \sqrt{gHo} * t)$$

with parameters given earlier in this report. A total of 101 equally spaced sample points were used to train for the regression. The function with point samples is shown in Figure 1.

Figure 1. The 1D function and training points for machine learning (ML) regression.



As can be seen from this figure, this discretization was capable of capturing six wavelengths over the temporal domain with decent accuracy. Two optimizers were tested for a range of networks and learning rates. The simulations were run for a total of 2,000 epochs, a generally acceptable number for this simple regression. Only one batch was used, which included all 101 sample points.

Results for each network configuration are given as a grid of layers versus neurons, with pixel colors representing the average error over learning rates shown below. All errors for the 1D case were calculated on a prediction grid with double the resolution and difference spacings so that the training and prediction points are not aligned. Wall clock times

reported are only for the TF ML call and do not include data preparation, prediction, or output.

Figures 2 and 3 show the SGD optimizer regression clearly outperforming the Adam optimizer when it came to both the absolute and root mean squared error (RMSE). However, optimal errors for the Adam optimizer were found for a small number of layers. While both errors had minor changes as the number of neurons were increased, they were mostly dependent on the number of layers. For example, the Adam optimizer gave the best results for five-layer networks, while the SGD started to give the best results beginning at eight layers.

Figure 2. TensorFlow errors (*left*) and standard deviation of errors (*right*) over learning rates ranging from 0.015–0.03 for the 1D problem using the Adam optimizer.

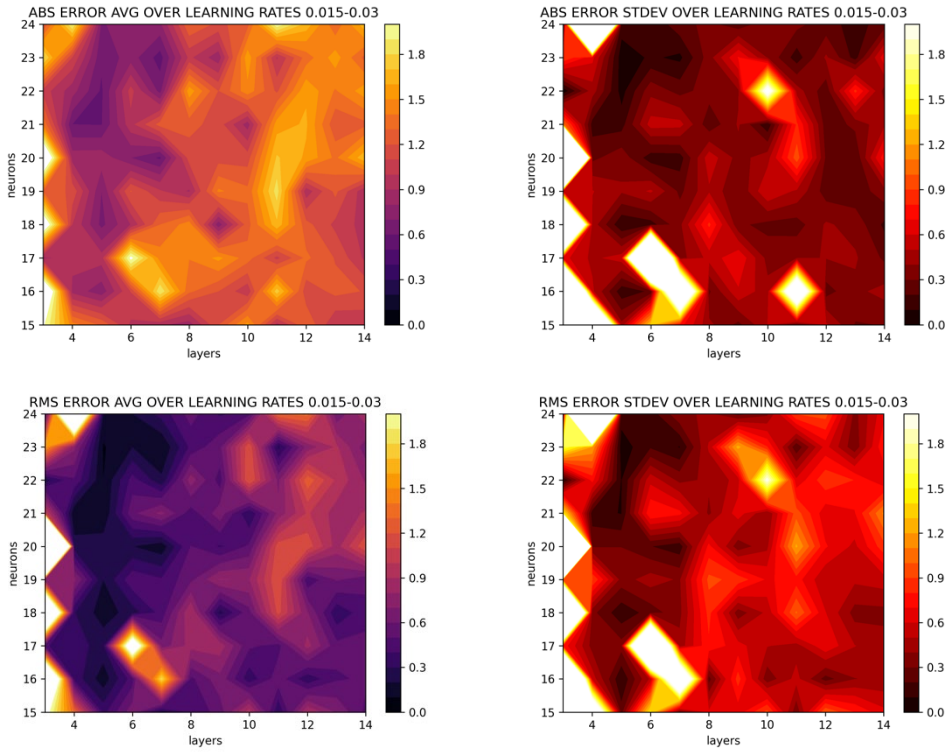
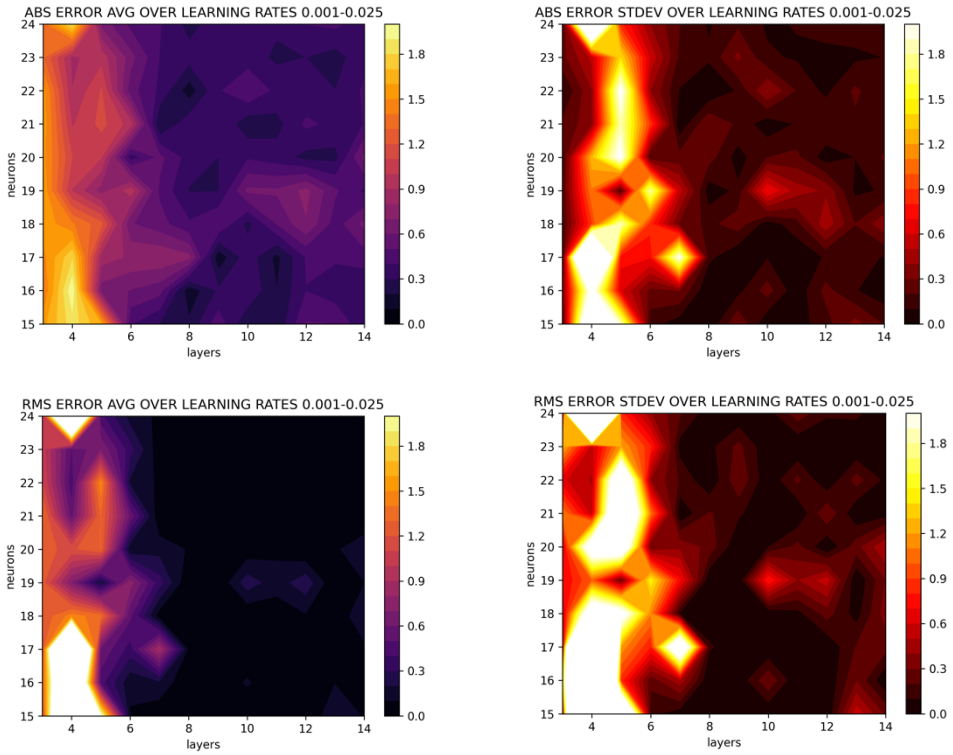


Figure 3. TensorFlow errors (*left*) and standard deviation of errors (*right*) over learning rates ranging from 0.001–0.025 for the 1D problem using the SGD optimizer.



In Figure 4 and Figure 5, we see that the wall clock timings scaled linearly with the number of layers in the network and that neither optimizer had a substantial advantage over the other when it came to wall clock.

Figure 4. Wall clock timings for the TensorFlow Regression for the Adam (*left*) and SGD (*right*) optimizers on the 1D function.

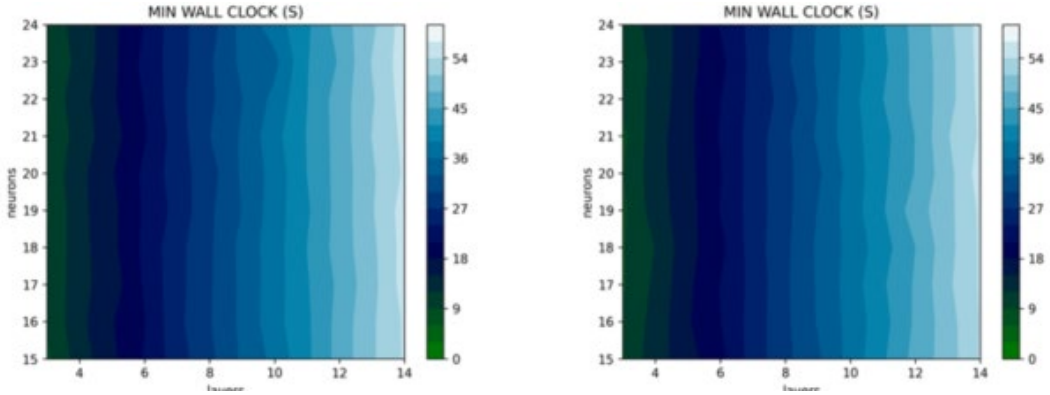
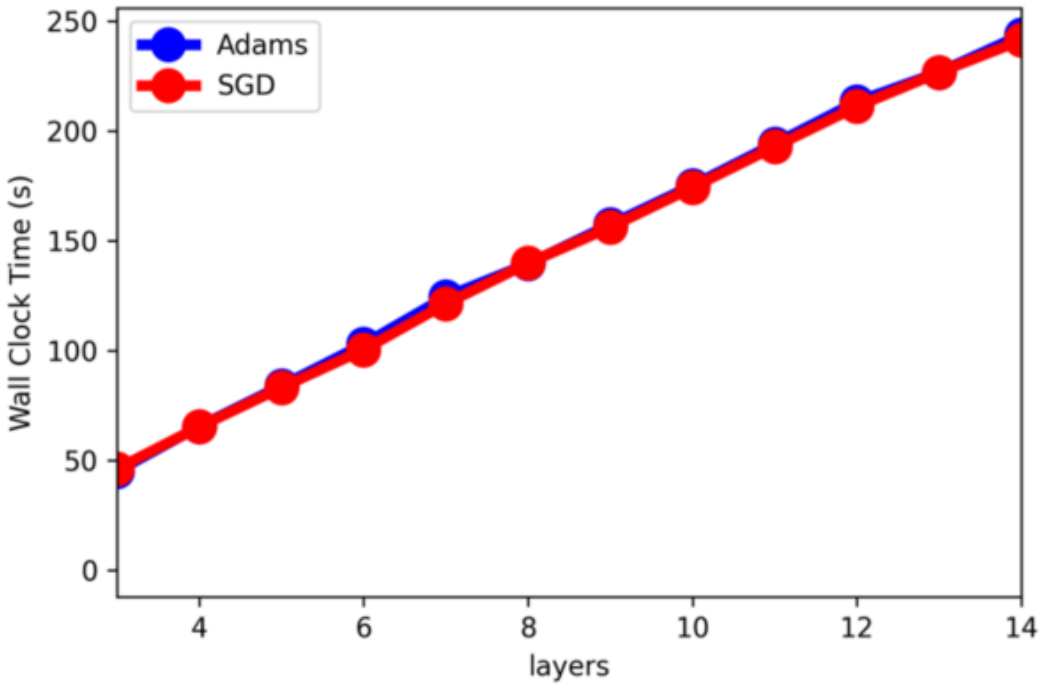


Figure 5. Wall clock timing comparison for TensorFlow Regression for the Adam and SGD optimizers on the 1D function.



2.5 2D regression results

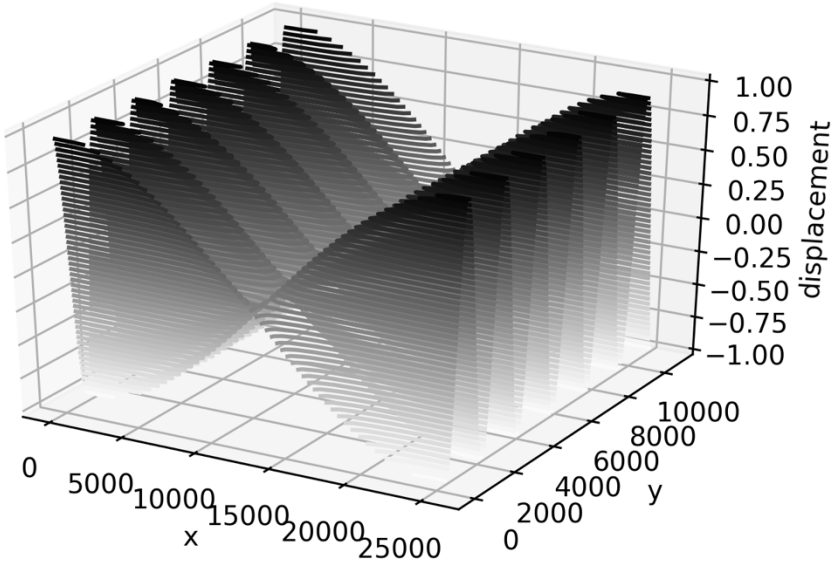
For the 2D regression, the oscillatory function given was used with $y = 0$,

$$f(x,t) = a * \cos\left(\frac{\pi}{L} * \frac{1}{\sqrt{2}}x\right) * \cos\left(\frac{\pi}{L} * \sqrt{gHo} * t\right)$$

with parameters given earlier in this report. For this regression, a total of 101 equally spaced sample points in time were used and 11 points along the x-axis. The 2D function with point samples is shown in Figure 6.

This discretization should be able to capture six wavelengths over the temporal domain and a half-wavelength in the x-direction with decent accuracy. Once again, the Adam and SGD optimizers were tested for a range of networks and learning rates. The simulations were run for a total of 10,000 epochs in order to maintain similar errors to that of the 1D regression. One batch that included all sample points was used.

Figure 6. The 2D function and training points for ML regression.



Results for each network configuration are given as a grid of layers versus neurons, with pixel colors representing the average error over learning rates shown below. All errors for the 2D case were calculated on a prediction grid with double the resolution and difference spacings so that points are not aligned. Wall clock times reported are only for the TF ML call and do not include data preparation, prediction, or output.

Figures 7 and 8 display the ABS and RMSE accuracy of the Adam and SGD optimized regressions, respectively. The Adam optimizer seemed to perform better for ABS errors and worse for RMSE. Since the RMSEs are squared before they are averaged, this error gives a relatively high weight to large errors. This means the RMSE should be more useful when large errors are particularly undesirable. If this is the case, then the SGD optimizer is preferred.

While the optimal number of layers for Adam seemed to be six, nine layers were optimal for SGD.

Figure 7. TensorFlow errors (*left*) and standard deviation of errors (*right*) over learning rates ranging from 0.001–0.02 for the 2D problem using the Adam optimizer.

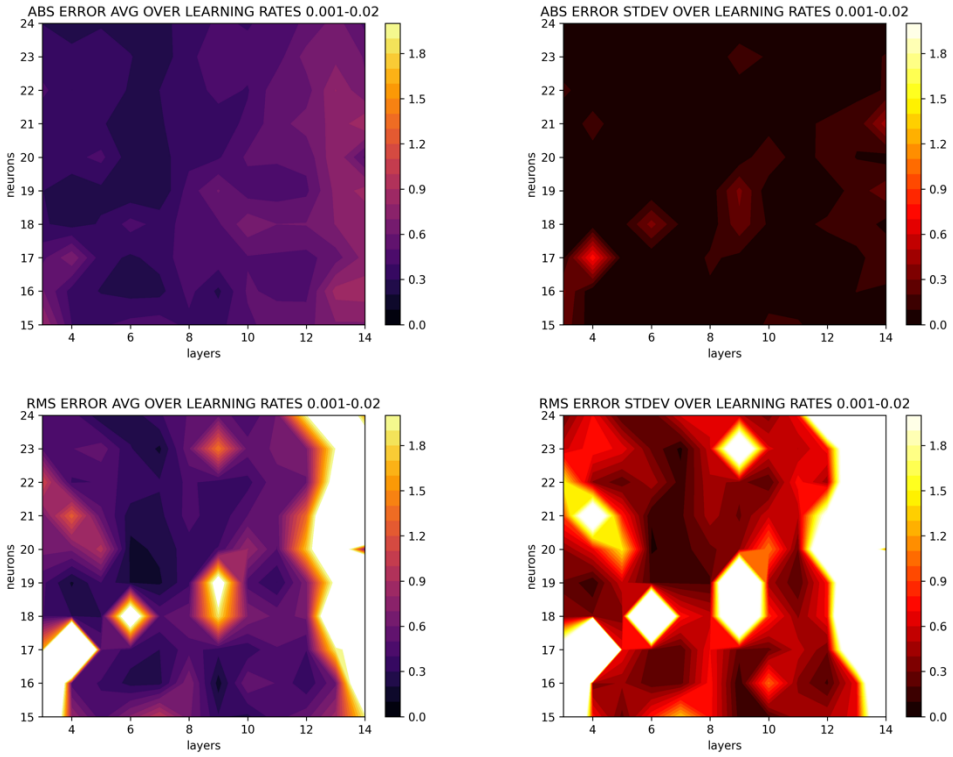
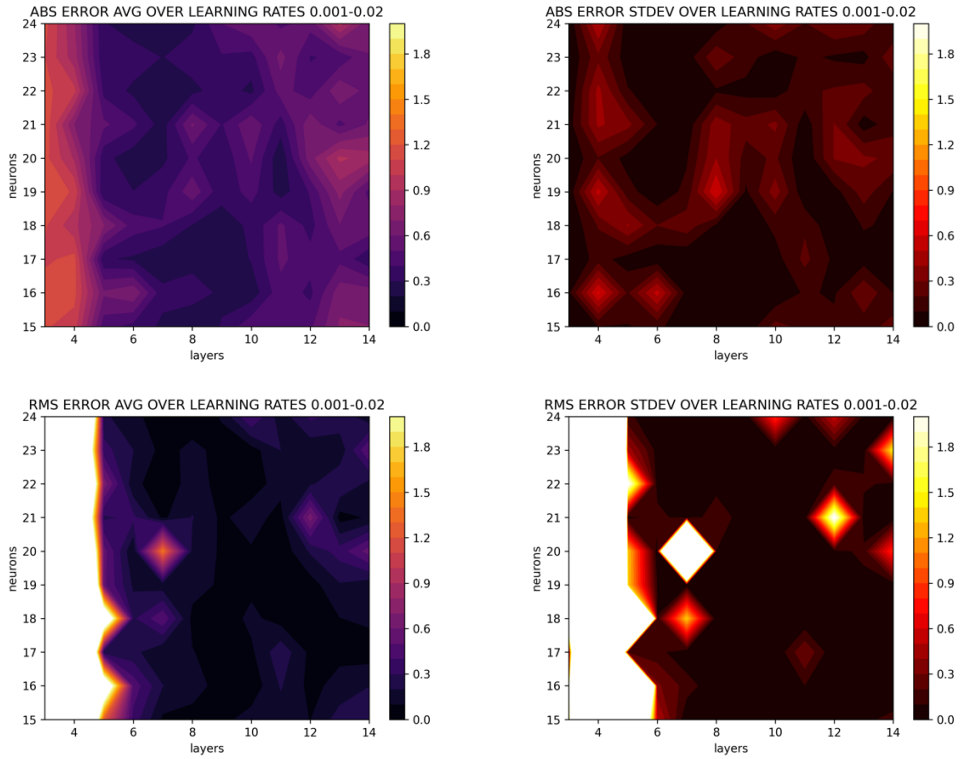


Figure 8. TensorFlow errors (*left*) and standard deviation of errors (*right*) over learning rates ranging from 0.001–0.02 for the 2D problem using the SGD optimizer.



In Figures 9 and 10, we see again that timing differences between the Adam and SGD optimizers were not substantial and only really start to deviate for large layer counts. Also, the wall clock timings varied linearly with increasing layer count, and no substantial wall clock difference was obtained as the number of neurons grew.

Figure 9. Wall clock timings of the TensorFlow regression for the Adam (*left*) and SGD (*right*) optimizers on the 2D function.

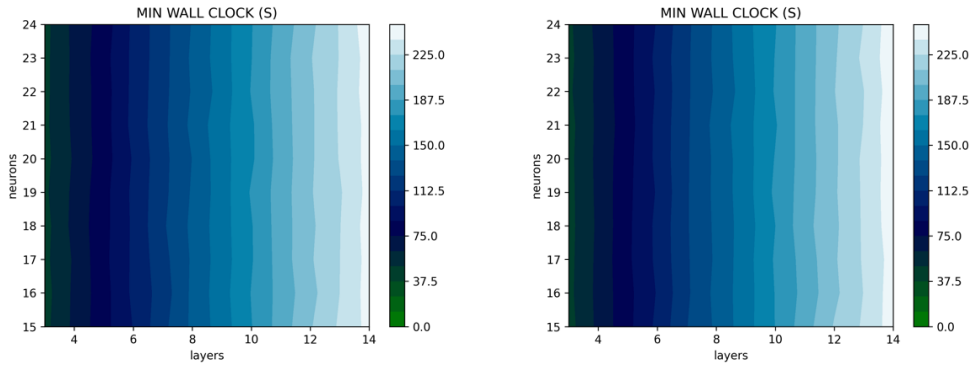
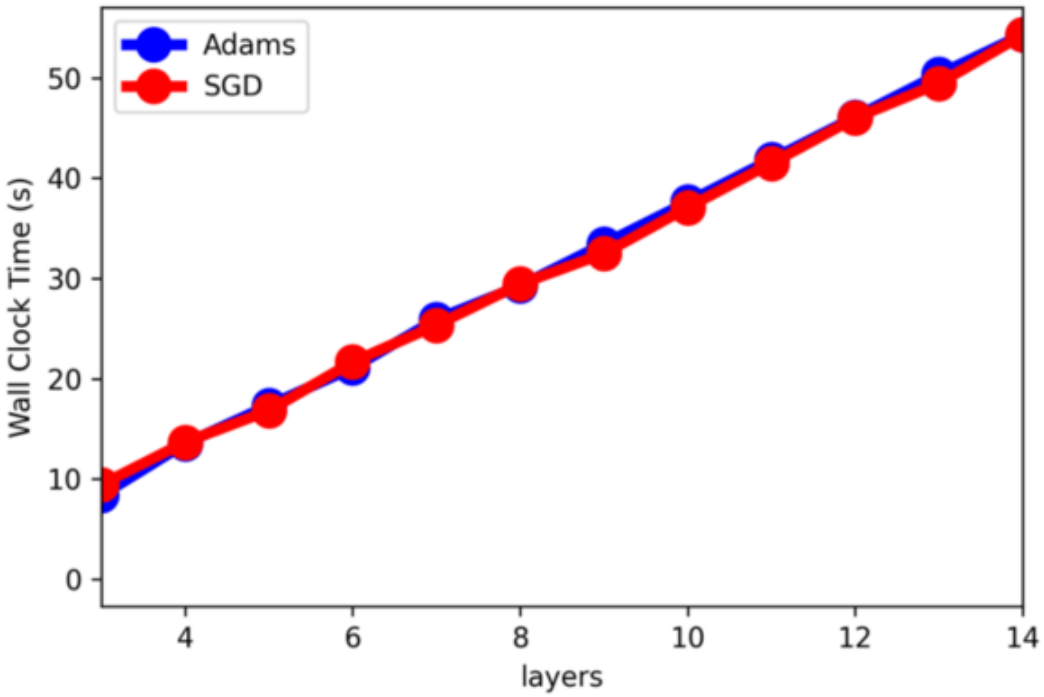


Figure 10. Wall clock timing comparison of the TensorFlow regression for the Adam and SGD optimizers on the 2D function.



2.6 3D Regression Results

For the 3D regression, the equation

$$f(x,y,t) = a * \cos\left(\frac{\pi}{L} * \frac{1}{\sqrt{2}}(x+y)\right) * \cos\left(\frac{\pi}{L} * \sqrt{gHo} * t\right)$$

was used with parameters given earlier in this report. For this regression, a total of 101 equally spaced sample points in time were used and 21 points along the *x*,*y*-axis. The Adam and SGD optimizers were tested for a range of networks and learning rates for this regression. The simulations were run for a total of 30,000 epochs in order to maintain a similar magnitude of error as the 1D and 2D regressions herein. One batch that included all sample points was used.

Results for each network configuration are given as a grid of layers versus neurons, with pixel colors representing the average error over learning rates shown below. All errors for the 3D case were calculated on a prediction grid with double the resolution and difference spacings so that points are not aligned. Wall clock times reported are only for the TF ML call and do not include data preparation, prediction, or output.

Figures 11 and 12 show the ABS and RMS errors for the 3D regression. It is clear from the figures that the Adam optimized regressions outperformed the SGD in both types of errors. In fact, the SGD results report the worst RMS errors of them all, emphasizing erratic behaviors for some learning rates and resulting in large errors spikes, most obvious in RMSE. For the Adam optimizer, optimal errors did not particularly favor the number of neurons per layer and were best found using 8 layers. For the SGD case, optimal ABS errors were found around 10 layers with 20 neurons or more and 12 layers with 25 neurons or more for RMSE.

Figure 11. TensorFlow errors (*left*) and standard deviation of errors (*right*) over learning rates ranging from 0.0005–0.004 for the 3D problem using the Adam optimizer.

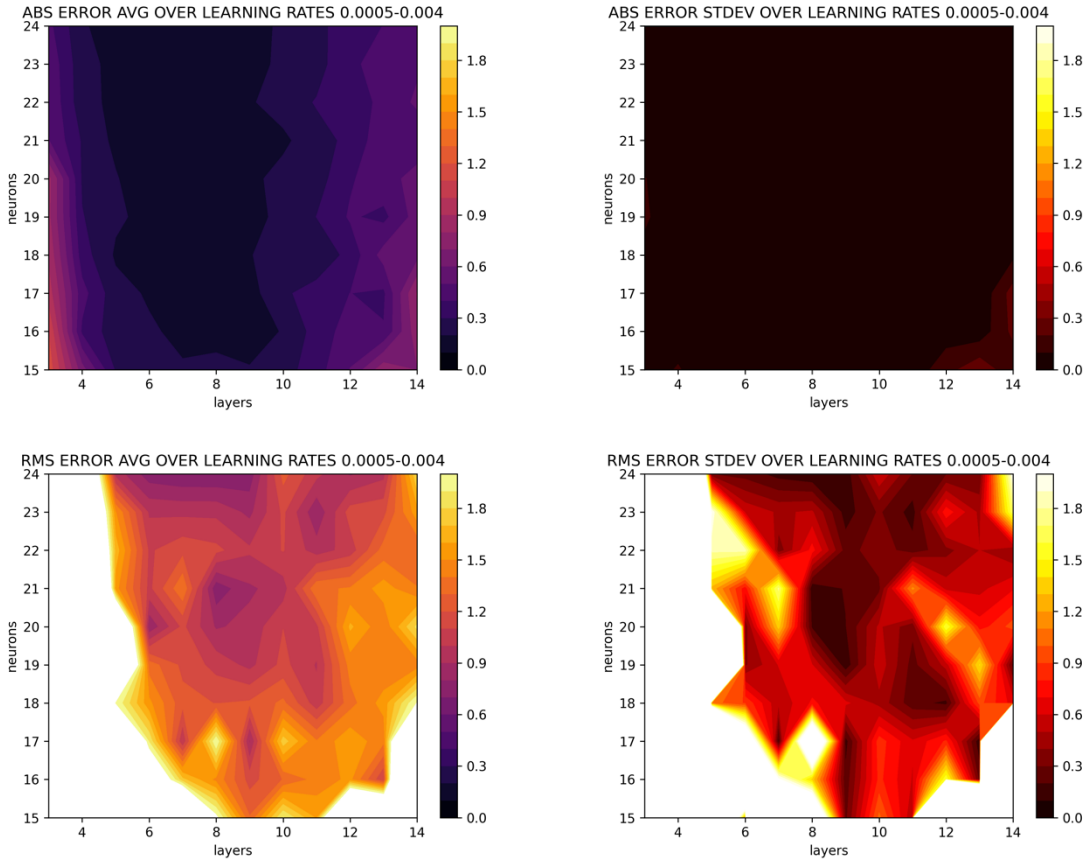
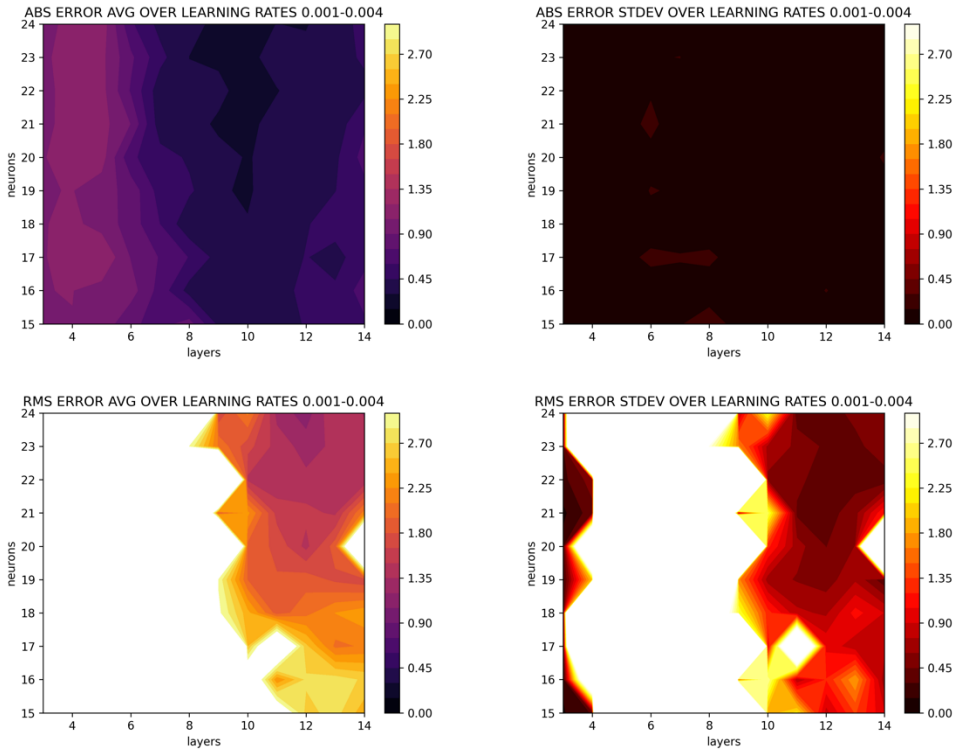


Figure 12. TensorFlow errors (*left*) and standard deviation of errors (*right*) over learning rates ranging from 0.001–0.004 for the 3D problem using the SGD optimizer.



In Figure 13 and Figure 14, it is once again found that timing differences between the Adam and SGD optimizers were not substantial and only really start to deviate for large layer counts. Also, the wall clock timings vary linearly with increasing layer count, and no substantial wall clock difference was obtained as the number of neurons grew.

Figure 13. Wall clock timings for the TensorFlow regression for the Adam (*left*) and SGD (*right*) optimizers on the 3D function.

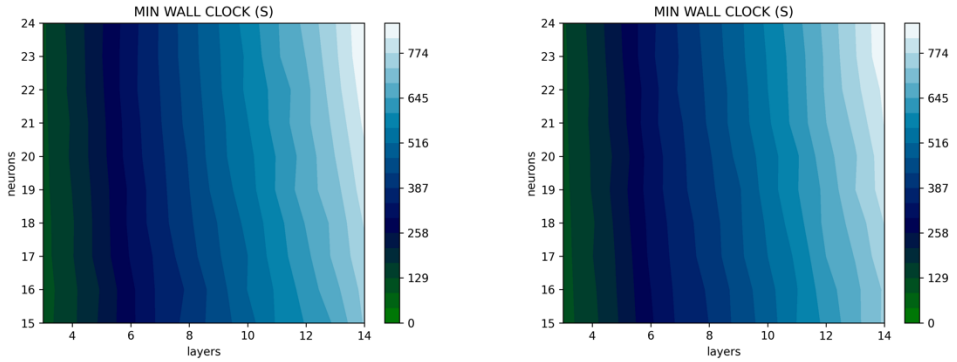
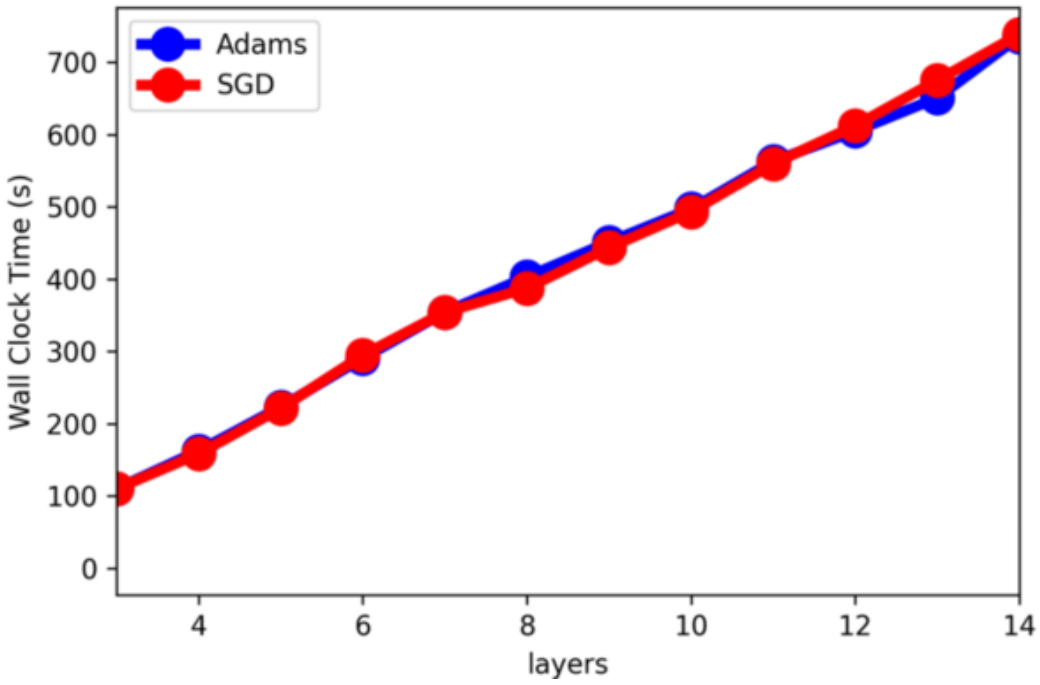


Figure 14. Wall clock timing comparison for the TensorFlow regression for the Adam and SGD optimizers on the 3D function.



3 Conclusions and Recommendations

3.1 Conclusions

3.1.1 Layer Count Sensitivity

For all dimensional regressions in this report, there was a linear trend in the wall clock timings as the number of layers was increased from 3 to 14 This was true for both optimizers. Additionally, the most notable trends in accuracy for both optimizers were obtained under varying layer counts. Lastly, there was a noticeable trend toward optimal errors for slightly lower layer counts for the Adam optimizer. The layer count seemed to be one of the most sensitive parameters for all dimensions for both optimizers.

3.1.2 Neuron Count Sensitivity

Neuron count seems to be the least sensitive parameter when it comes to wall clock and prediction accuracy for both optimizers, especially at lower dimensions. For the 3D regression, there was a minimal effect on the wall clock, but this trend was very minor. Although trends could be found at times, in 1D and 2D the solution accuracy can highly fluctuate in an unpredictable way with the number of neurons. The SGD optimizer did show slightly more stability as the neurons increased for the lower dimensions. On the other hand, results for the 3D accuracies were far more stable for the range of neurons investigated and do show a natural trend toward better accuracy as the number of neurons is increased for both optimizers. This may be because the 3D regression was calculated for 30,000 epochs, 3 times that of the lower dimensional problems. More epochs were used for the 3D function regression to achieve about the same accuracy as the 1D and 2D regressions.

3.1.3 Optimizer Sensitivity

Neither optimizer showed a competitive advantage with respect to wall clock timings; however, there were quite noticeable differences in the regression errors and standard deviation of the errors (recall the standard deviation is with respect to the learning rates for a given number of layers and neurons). For the 1D and 2D regressions, the SGD outperformed Adam when it came to regression accuracy. The SGD optimizer also showed nice trends in accuracy at these dimensions with less random

jumps. SGD also favored lower learning rates. On the other hand, the Adam optimizer dramatically outperformed the SGD optimizer for the 3D regression, seemingly pointing to the Adam optimizer as the better of the two for higher dimensional regressions. Additionally, the Adam optimizer gave optimal errors for lower layer counts.

3.1.4 Learning Rate Sensitivity

In general, there was a rather large sensitivity shown with respect to learning rate for both optimizers for all regressions, particularly as the number of layers changed. This erratic behavior is responsible for the non-smooth regions in the error plots for all dimensional applications. Normally, this happens when the learning rates are too large, as solution increments can jump too far and miss an area near a local minimum. This can also happen in poorly sampled regions of the domain. In the problems herein, however, not only were the sample points evenly distributed, but error jumps were seen over a wide range of learning rates, indicating a need for more regularization in the TF regression algorithm. Some ways to increase the regularization of the ML regression would be to introduce dropouts, early stopping, and lasso regression / ridge regression (L1/L2) regularization. This study did not look into those type of regularization methods.

3.1.5 Scalability

Scalability for the two optimizers was about the same for the functions considered. As mentioned, for a given dimensionality, the overall wall clock scaled linearly with the number of layers, and a very minor dependence was found on the number of neurons (which was only seen in 3D). As the dimensionality was increased, however, the regression seemed to scale quadratically. One important note on this scaling assessment is that the number epochs were increased from 2,000 in 1D, 10,000 in 2D to 20,000 in 3D. This was done in an attempt to maintain similar solution errors while increasing dimensionality. Because of the complexity of functions in higher dimensionalities, a greater number of epochs for regression should be expected, and this number will govern the feasibility of application on a particular machine.

3.2 Recommendations

Periodic functions can be difficult to perform regressions on as they can be highly oscillatory and nonlocal by nature. Outside of the number of epochs, the number of layers used in the deep network is the wall clock bottleneck, at least for functions similar in form to those studied in this report. The number of layers and learning rate parameters were the most sensitive when it came to prediction errors. Additionally, it is recommended that some form of regularization is used to reduce the sensitivity of the learning rate on the solution accuracy.

Reference

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, et al. 2015. *TensorFlow: Large-scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org.

Abbreviations

ERDC	Engineer Research Development Center
MAE	Mean absolute error
ML	Machine learning
MSE	Mean squared error
PIML	Physics-informed machine learning
RMSE	Root mean squared error
SGD	Stochastic gradient descent
TF	TensorFlow

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
August 2023	Final technical report	FY2020	FY2022

4. TITLE AND SUBTITLE

Scaling and Sensitivity Analysis of Machine Learning Regression on Periodic Functions

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT
		0603465
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
AL3		

6. AUTHOR(S)

Corey J. Trahan and Peter G. Rivera

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Information Technology Laboratory
US Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

8. PERFORMING ORGANIZATION REPORT NUMBER

ERDC/ITL TR-23-4

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Headquarters US Army Corps of Engineers Washington, DC 20314-1000	USACE	

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this report we document the scalability and sensitivity of machine learning (ML) regression on a periodic, highly oscillating, and C^∞ function. This work is motivated by the need to use ML regression on periodic problems such as tidal propagation. In this work, TensorFlow is used to investigate the machine scalability of a periodic function from one to three dimensions. Wall clock times for each dimension were calculated for a range of layers, neurons, and learning rates to further investigate the sensitivity of the ML regression to these parameters. Lastly, the stochastic gradient descent and Adam optimizers wall clock timings and sensitivities were compared.

15. SUBJECT TERMS

Computer algorithms; Machine learning; Ocean circulation; Periodic functions; Regression analysis; Tidal Currents; Tides

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	34

19a. NAME OF RESPONSIBLE PERSON	19b. TELEPHONE NUMBER (include area code)