CONVERGENCE, ADAPTIVITY, AND APPLICATIONS OF PHYSICS-INFORMED MACHINE LEARNING

A Dissertation

by

LEVI DANIEL MCCLENNY

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2022

Major Subject: Electrical Engineering

ABSTRACT

Extensive work in applying deep learning to broader fields of science and engineering have been emerging in recent times, to include materials informatics, thermodynamics, and numerous other fields of computational sciences. Advances in these areas have been of particular excitement as future materials and new and informative laws of nature can be learned from data, even if that data is less than what would typically be required of a deep learning approach. In this work, we focus on the development and democratization of Physics-Informed Deep Learning, a field of science that was proposed before the turn of the century but has recently been gaining rapid popularity among academia and industry alike.

This dissertation is centered around recent work in physics-informed deep learning, as well as other areas of deep learning applications in computational sciences, such as materials informatics. Specifically, we will address recent advances in training stability and convergence of PINN solvers to semi-linear and stiff problems where the baseline PINN fails to converge or train effectively. We will discuss specific applications of PINNs to computational science domains where it could provide a force multiplier to researchers, and work performed in deep learning estimation of phase field modeling. Additionally, we will discuss the open-source package *TensorDiffEq*, a Python package based on Tensorflow that allows for easy implementation of PINN-based forward, inverse, and data assimilation solvers.

DEDICATION

To my mother, father, stepfather, brother and loving wife. Without your unyielding support, patience, and encouragement this work would not have been possible.

ACKNOWLEDGMENTS

To my research advisor, Dr. Ulisses Braga-Neto - thank you for believing in me and giving me this opportunity. To Dr. Mulugeta Haile, from the US Army Research Lab, thank you for the unyielding support. To all my mentors and friends along the way, thank you for being a part of this journey.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Dr. Ulisses Braga-Neto [advisor, ECEN], Dr. Narasimha Reddy [ECEN], Dr. Yang Shen [ECEN, CSCE] and Dr. Raymundo Arroyave [MSEN, MEEN].

The approaches and results listed in chapter IV are in collaboration with graduate students from the departments of Petroleum Engineering (Emilio Coutinho, Marcelo Dall'Aqua) and Physics (Xingzhuo Chen). Additional credit goes to Ming Zhong (TAMIDS) for his assistance and subject matter expertise contributing to those manuscripts. Synthetic microstructure data used in section I is contributed by Vahid Attari (MSEN).

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

The authors would like to acknowledge the support of the D^{3}EM program funded through NSF Award DGE-1545403. The authors would further like to thank the US Army CCDC Army Research Lab for their generous support and affiliation.

NOMENCLATURE

PINN	Physics-Informed Neural Network
SA-PINN	Self-Adaptive Physics-Informed Neural Network
CNN	Convolutional Neural Network
DMTL-R	Deep Multimodel Transfer-Learned Regression
MSE	Mean Squared Error
ReLu	Rectified Linear Unit
ADAM	Adaptive Moment Estimator
GPU	Graphics Processor Unit
PDE	Partial Differential Equations
AC	Allen-Cahn PDE System
CH	Cahn-Hilliard PDE System
TDQ	TensorDiffEq
SGD	Stochastic Gradient Descent
GPR	Gaussian Process Regression
NTK	Neural Tangent Kernel
BL	Buckley-Leverett
PINN-RT	PINN for Radiative Transfer
MM-PCNN	Mesoscale Multi-Physics Constrained Neural Network

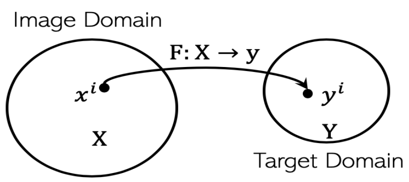
TABLE OF CONTENTS

LIST OF FIGURES

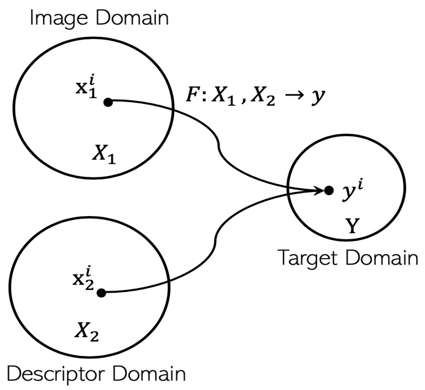
LIST OF TABLES

1. Deep Multimodal Regression in Data Poor Domains with Applications in Materials Informatics

Consider the following problem - shown in Figure 1.1 - in which we desire to apply deep regression in an application domain in which a ConvNet has not been trained, and there exists additional data which is hypothesized to assist in this regression task. In this instance, assume further that there does not exist sufficient data to train a new ConvNet from random initializations. In this article, we provide a multimodal architecture which takes advantage of model fine-tuning and transfer learning to overcome a lack of sufficient training data. The result is a regression approach that combines images and descriptive statistics which can be effectively trained on a modestly sized dataset.



(a) CNN Estimation on an image-only domain. Here, $\mathbf{F} : \mathbf{X} \rightarrow \mathbf{y}$ is a ConvNet used to predict a value in the target domain



(b) Estimation on a multimodal image-descriptor domain. Here $\mathbf{F} : \mathbf{X}_1, \mathbf{X}_2 \rightarrow \mathbf{y}$ is the proposed DMTL-R estimator

Figure 1.1: Single-domain input vs. multi-domain input for multimodal regression. In (b) the images are combined with a corresponding vector describing the image in question in another data domain

This paper makes the following major contributions:

- Develops a multimodal deep learning regression methodology that incorporates CNN-based image featurization conditioned on descriptive statistics of the input images

- Shows efficacy of the multimodal transfer-learned regression methodology in a data-poor application domain

Our approach yields a transfer-learned regressor with better residual R^2 than image-only regression alone, and presents an algorithm that allows for the combination of image and descriptive statistics that can be applied to a large variety of scientific domains. This provides a potential approach to a significant question in deep learning - how to effectively incorporate interpretable a priori descriptive information about an image-based system into an estimation task - which we address in this work via multimodal deep learning.

We validate this approach on images of phase-field simulated microstructures with accompanying descriptive statistics about the corresponding material system, and present this concatenated information to an estimator that seeks to regress parameters about the image in question. Phase-field simulation microstructures are extensively used to study the physical and mechanical properties of materials and provide a relevant example in the context of this algorithm, as microstructure data is expensive to obtain.

1.1 Related Work

Multimodal Learning The concept of combining different domains of input into a single estimator for the desired learning task has been of distinct interest in the last decade [4, 5]. Applications include speech classification using audio and video input [6], tagging or labeling of images using features and textual information [7], and numerous others. This framework is useful in approaching the problem we outline in Figure 1.1, as it allows for combinations of different types of input data, which together describe a similar location in the input space. It is worth clarifying that what could be considered *descriptive information* about the image in [7] are image captions, i.e. textual information. In this work, we refer to descriptive information as *numerical* vectors describing the image. We apply this framework for a regression task, specifically combining the image and descriptive input domains into a single estimator.

Multi-Source Domain Adaptation Domain Adaptation is considered a branch in the broad area of transfer learning [8, 9]. Specifically, multi-source domain adaptation addresses the question of multiple sensor inputs to an estimator in which there is only one target domain, and how to effectively leverage information from another domain in this new application. This can typically be accomplished either via adding up all the data sources into a single source or by training classifiers on each branch and aggregating those results for a final estimation. This particular approach, most generally, aggregates results from classifiers trained on the same *type* of data, from various sources, not necessarily different domains (or modes), such as what we see in Figure 1.1. In this work, we seek to develop a *multimodal* domain adaptive regressor using transfer learned networks.

Multi-Input Transfer Learning As mentioned above, domain adaptation is a broad branch of transfer learning. Recently, multi-source domain adaptation using pre-trained networks has caught much attention, as it generates models that allow for a smaller amount of training data for the model in question [10]. Recent work has been done in the multi-source transfer learned estimator space [11, 12, 13], using text-text and text-image combinations to perform classification. Most of these works propose frameworks that are built in an ensemble fashion, i.e. trained independently, and then the best classifier selected. In this work, we address the task of regression specifically, using a single estimator (built around a transfer-learned network) with synchronous training of parameters.

Deep Regression Recently, image-based deep regression has become a rapidly advancing application of deep learning [14]. Taking a CNN and applying it to the task of regression is a problem that has large implications in many areas of science and engineering. While the work in [14] is extremely thorough in the task of fine-tuning a CNN model using pre-trained weights, it does not incorporate the additional statistics described in this article as a component of their model. It does offer important insight into the task of using CNNs for regression, insight which is used in the formulation of the models in this article. We offer an extension of the deep regression task in [14] by taking additional descriptive information about images and incorporating that information into the estimation.

The proposed Deep Multimodal Transfer-Learned Regression (DMTL-R) algorithm is novel in that it requires the training of only one estimator, whereas most work related to multimodal transfer learned estimation requires the training of multiple estimators. Additionally, the entire model is built of connected layers, allowing backpropagation to flow through the entire network at once, removing any sort of selection requirement seen in other multi-source transfer learning-based regressors. Moreover, the results presented in section 1.4, which are gathered from a data set with a modest number of samples, indicate that our approach to multimodal regression is accurate and efficient in a data-poor environment.

1.2 Development of the Model

Fine-Tuning the ConvNet We define a ConvNet as a nonlinear function approximator that maps an image input to a target label (for prediction) or target value (for regression). The ConvNet is trained on input-target realizations $x \in \mathbf{X}$ and $y \in \mathbf{Y}$. Here we can say that the image-target pairs exist in the ILSVRC-2012 dataset [15], more commonly known as ImageNet. We'll call this domain G. Once trained, a ConvNet makes a prediction $\bar{\mathbf{y}}$ for an input image or batch of images $\mathbf{x}$. We define a prediction on a ILSVRC-2012 trained ConvNet $\overline{\mathbf{F}} : \mathbf{X} \rightarrow y$ as

$$\overline{y}(x) = \overline{F}(x, w_T),\ x, y \in G \tag{1.1}$$

where G is the ILSVRC-12 domain, x and y are the image and target domains respectively, and w_T are the trained weights of the network. In most applications, these weights are trained specifically to best featurize the images in the training domain of the ConvNet. These weights are unique to the layer to which they belong, and we can index them as $w_i,\ i \in \{1, n\}$ where n is the number of trainable layers in the network. Model fine-tuning typically takes advantage of the subset of layers used to featurize an image, i.e. excluding the fully-connected layers in the VGG16 architecture. Here we can subset those weights as the image featurization weights $w_f \in w_T$. We generate a new neural network around the w_f layers, with trainable weights w_{FT}, and create a featurization u which captures the features of a new domain $\mathbf{H} \notin \mathbf{G}$. We can define a new function

$$u(x_H) = \overline{F}^*(x_H, w_f, w_{FT}) \tag{1.2}$$

Here $\overline{F}^*$ is trained via backpropagation in a similar sense to the original model, but the featurization layers w_f are held constant - this creates a mapping of the activations of the image to a new trainable set of fully-connected layers w_{FT}.

Conditioning on Descriptive Statistics We define a descriptive variable of arbitrary length n as some associated vector describing the image data input to the ConvNet.

In Figure 1.1(b) we see the descriptor domain $\mathbf{X}_2$ where a vector realization $x_2 \in R^n$ is used as a descriptive statistic to condition the regression output of the multimodal regressor outlined in Figure 1.2. We generate a small fully-connected network with trainable weights w_{MLP} to allow the featurization of the conditioning statistics and allow for the optimal MSE regressor via backpropagation through the whole network. We will call this featurization component v, defined as

$$v(x_2) = F(x_2, w_{MLP}) \tag{1.3}$$

Featurized image components and descriptive conditioning statistics are then concatenated in a fully-connected layer, which does not affect the differentiation of backpropagation. This combines the values and allows for addition of a final few fully-connected layers, with a dropout[16] component added between most fully-connected layers induce regularization in the model via random masking of nodes. The weights w_r of these last few layers are also trainable and result in the final regression. The output layer has a linear activation function, while all the intermediate layers use ReLu activation.

We are left with a final regression algorithm $r(x_1, x_2)$ which accounts for image and descriptive statistics at sample point i as follows:

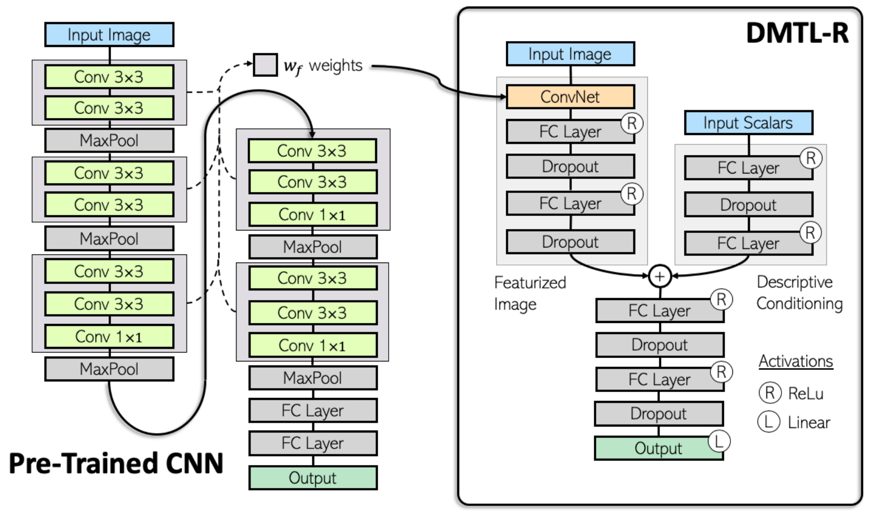


Figure 1.2: Proposed DMTL-R Estimator architecture. The left image is a block diagram of the VGG16 architecture, with its featurization weights w_f highlighted. These are an example of featurization weights that can be used in the DMTL-R (right) to estimate the target value at the output layer.

$$r_i(x_1, x_2) = F(u(x_1^i), v(x_2^i), w_r) \tag{1.4}$$

$$= F(x_1^i, x_2^i, w_f, w_{FT}, w_{MLP}, w_r) \tag{1.5}$$

This is optimized via backpropagation, in the form of batch gradient descent against an MSE loss function

$$\mathcal{L}_{batch} = \frac{1}{n}\sum_i^n (y_T - \overline{y})^2 = \frac{1}{n}\sum_i^n (y_T^i - F(u(x_1^i), v(x_2^i), w_r))^2 \tag{1.6}$$

$$= \frac{1}{n}\sum_i^n (y_T^i - F(x_1^i, x_2^i, w_f, w_{FT}, w_{MLP}, w_r))^2 \tag{1.7}$$

where n is the batch size, $i \in \{1, n\}$ is a sample of that batch, and $\overline{y}$ and y_T are the predicted and actual target values for that sample, respectively. All the trainable weights w_{FT}, w_{MLP} and w_r are updated through the entire combined regressor using the final target prediction value $r_i(x_1, x_2)$.

The intermediate functions $u(x_1), v(x_2)$ are not individually optimized.

1.3 Training the Model

1.3.1 Training Data

The training images used in this study are of material microstructures generated from a sweep of input physical coefficients to physics-based phase-field simulations, which are a powerful tool in materials science to predict complex evolution kinetics in materials processes [17]. The images utilized for this example are visual depictions of material phase separation during processing (also known as spinodal decomposition) and are results from the phase-field simulations. Our *descriptive statistics* in this example are 18-tuple sets of continuous input processing parameters used for the phase-field simulation. These tuples act as physical parameters to a nonlinear set of coupled partial differential equations which are computationally difficult to analyze and must be solved numerically via Fourier spectral method (or another similar numerical solver) across many CPU cores - potentially hundreds. Combining the two sets of inputs, we have an input data set composed of image data and corresponding vectors of numerical values. The target (output) values to be regressed are 6-tuple physical characteristic outputs from the phase field simulation, such as min/max compositions and chemical potential. These are simulated outputs from the phase-field solution that we seek to regress from the image data and corresponding input vectors using the above outlined DMTL-R approach. The results are discussed for various regression trials in section 1.4.

The training/test data consists of 2500 images of fully spinoidally decomposed microstructures obtained from the open phase-field microstructure database [18].[1] Spinoidally decomposed images were chosen as they are the most visually diverse and informative images. 2500 images is a very small dataset for a CNN architecture, and the fully-connected architecture described in this review has a few million trainable parameters, i.e. $p >> n$. It is worth noting, however, that this is far fewer trainable parameters than most commonly used CNN architectures [19]. We account for the $p >> n$ phenomenon in the model by introducing dropout to prevent overfitting via sparsity

[1]http://microstructures.net/

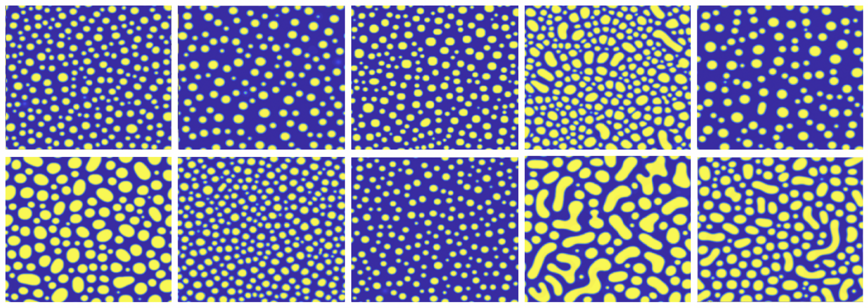
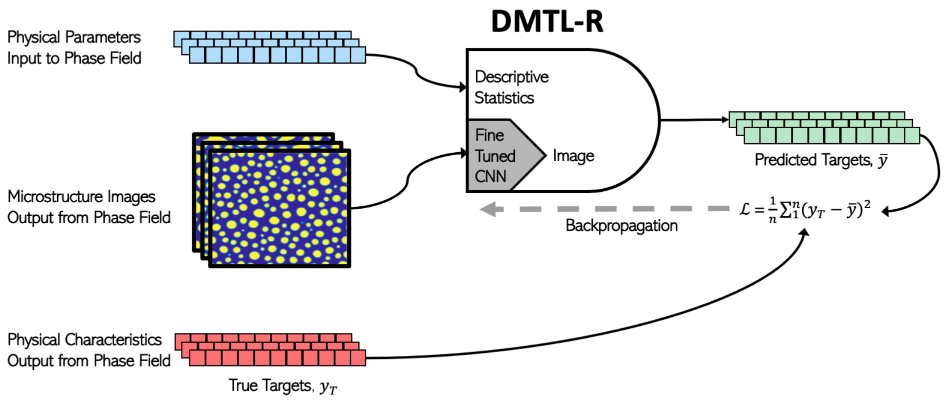


Figure 1.3: Microstructure images generated with phase-field simulation and input to the combined image-parameter regressor. The images are featurized using ConvNet architectures and conditioned with descriptive statistics. The original image sizes are 512x512x3.

induction. This is in addition to utilizing the transfer-learned ConvNet architecture, which on its own is a method of to proceed when using large deep learning methodologies with insufficient data to train a full-scale CNN model from random initializations.

The dataset is split randomly into train and test sets, with 2/3 of the data for training and 1/3 for testing, or ~1675/825. This split is done randomly and independently for each trial.


DMTL-R
Physical Parameters Input to Phase Field
Descriptive Statistics
Fine Tuned CNN
Image
Predicted Targets, $\bar{y}$
Microstructure Images Output from Phase Field
$\mathcal{L} = \frac{1}{n}\sum_1^n (y_T - \bar{y})^2$
Backpropagation
Physical Characteristics Output from Phase Field
True Targets, y_T
Figure 1.4: Training flow of the proposed Deep Multimodal Transfer-Learned Regressor (DMTL-R), the internals of which are shown in Figure 1.2

1.3.2 Experimental Setup

Once the available data is split into train and test sets, the images are resized (compressed) to the required input size (224x224 or 299x299, depending on the architecture) and mean pixel values are removed from each channel, as is common practice to centralize the image pixel distributions for ImageNet-based tasks and provide added stability in training. The descriptive statistics and the output target scalar values are mean-corrected and scaled to unit variance. Training iterations are reported in the next section for 20 training epochs. Adam optimization [20] was utilized for parameter optimization during training, with learning rate ranging from $.0001 < lr < .001$ with moderate decay. Batch size was set to 32 for all training comparisons in table 1.1. The model was built in Keras [21, 22, 23] and trained using a single Nvidia V100 GPU made available through an Amazon Web Service (AWS) P3 instance. With this setup, it takes 3-4s per epoch to train the DMTL-R network.

1.4 Results

1.4.1 Model Fine-Tuning for CNN-Based Regression

The VGG16 architecture shown on the left side of Figure 1.2 was used as a feature extractor for the microstructure images shown in Figure 1.3 and trained as a regressor with 3 additional fully-connected layers of size $[1000, 100, n_{output}]$. The final output layer is given a linear activation function, while the intermediate layers use ReLu to induce nonlinearity to the estimation.

While this architecture does accomplish the goal of predicting a target from a transfer learned ConvNet featurization, it can be determined from inspection of Figure 1.5 that the images on their own provide only weak training for the regression task. This does not mean that the model is ineffective. Rather, it implies that there is not enough information in the images alone to effectively train a regressor, which on its own is somewhat useful information about the physical problem at hand. To aid in this problem we condition on descriptive features, as shown in Figure 1.2, which is a focal point of this work. This generates a multimodal estimator that includes multi-domain input information for a single point in the input space. The results shown in Figure 1.5 should be used

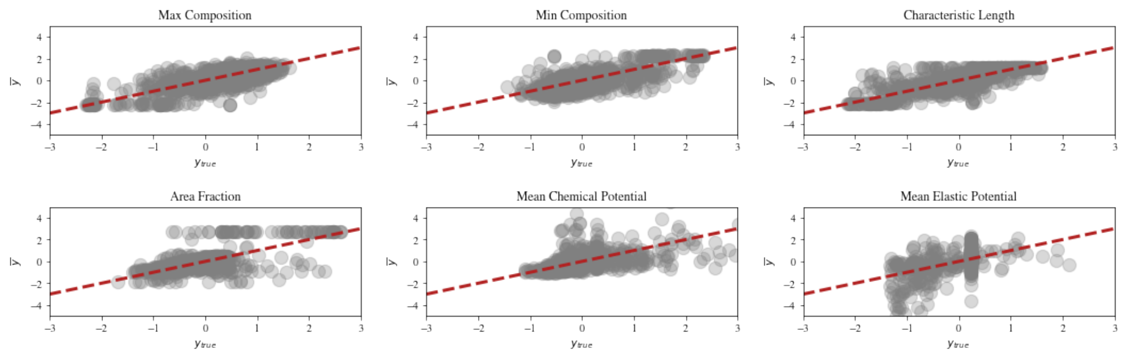


Figure 1.5: True vs. predicted estimates from CNN-based feature extraction using VGG16-based model fine-tuning. Max Composition, Min Composition, etc. are physical output characteristics of the spinoidal decomposition simulations.

as the baseline for comparison for the results in sections to follow, and are quantitatively tabulated for comparison in Table 1.1.

1.4.2 Single-Target DMTL Regression

The first validation experiment conducted with DMTL-R is that of single-target regression. The CNN-based featurization from section 1.4.1 is paired with descriptive statistics as suggested in the architecture in Figure 1.4. The final layer in the model from Eq. 1.5, with a linear activation function and MSE loss from Eq. 4.14, was given a node size of 1 and trained to regress a single parameter at a time. The DMTL-R results are shown in Figure 1.6, with plots of the MSE loss through training epochs shown in Figure 1.7.

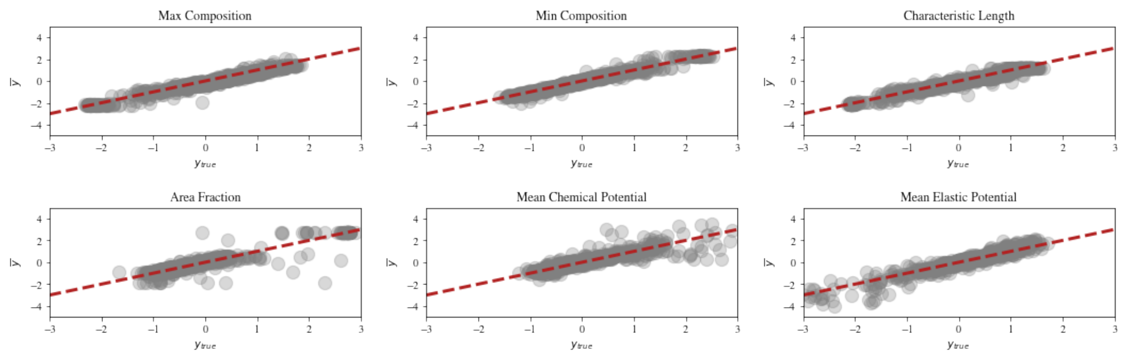


Figure 1.6: True vs. predicted estimates from single-target DMTL regression

We note here that the predicted estimate vs. true target values are, on average, very accurate with a reasonably low residual MSE. It's important to note that the regression targets, as well as the input descriptive vectors, were scaled using a standard scaler, i.e. subtracting the mean and scaled to unit variance. Therefore, the MSE values are somewhat arbitrary when discussing the predicted values of the parameters themselves. The MSE metric is, however, useful in comparing methodological and architectural differences, as well as providing a very stable loss function for training the DMTL-R network.

Figure 1.7 shows the training and test loss for each training epoch in the single-target regression case. We see that the model architecture, despite being in a position where overfitting could be preeminent in the model, does a reasonably good job in maintaining generalization to the test set. After 20 training epochs, the loss values approximately converge and the test set error does not exceed the training set error, which is typically interpreted as an indication of overfitting in the model. This would suggest that, despite the temptation of overfitting when training in a $p >> n$ environment, ConvNet transfer learning paired with standard sparsity induction techniques (such as dropout) do a reasonably good job of maintaining generalization while creating a strong multimodal regressor.

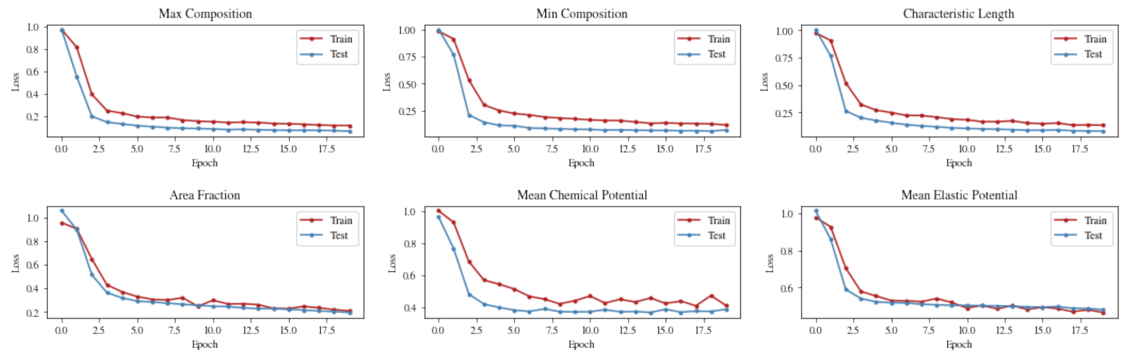


Figure 1.7: Single-target DMTL-R regression train and test set loss over 20 training epochs

1.4.3 Multi-Target DMTL Regression

Multi-target regression has been a longstanding topic in traditional pattern recognition spaces. One of the biggest strengths of deep learning is that increasing target diminsionality is as straightforward as increasing the number of nodes in the output layer. Here we extend the DMTL-R regressor to multi-target regression, in this case regressing all 6 parameters shown in Figures 1.6 at the same time. This results in little additional computational cost and can potentially have a positive effect on the accuracy of the individual dimensions estimated by the regressor, as seen in Table 1.1. The loss and regression results of the multi-target regression are shown in Figure 1.8.

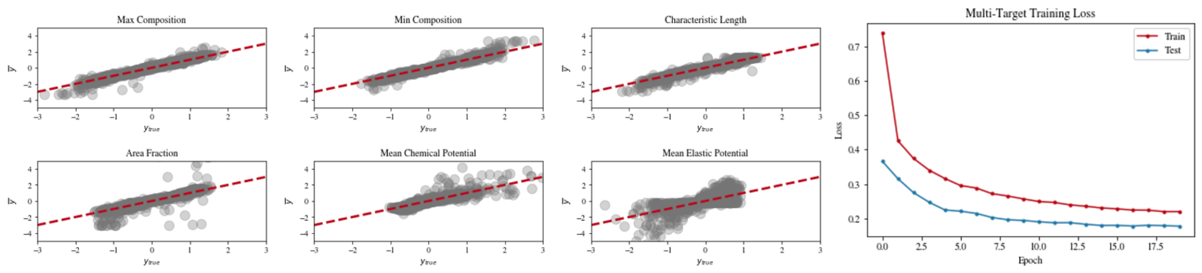


Figure 1.8: Estimates and train/test loss over 20 training epochs for the multi-target regressor

Table 1.1 outlines interpretation metrics for the regressor models, including the R^2 values for the respective regressors. This R^2 is from a linear fit for each true parameter vs. it's estimates, on the test set, for each output variable regressed. The slope, in all instances, is very close to 1, as anticipated. The R^2 value is listed as a metric by which we can assess the goodness-of-fit of the regressor. These values are listed in table 1.1 with 95% confidence intervals derived from multiple independent trials with independent splits of train/test sets for each trial, to validate generalization. We can see here that the multimodal DMTL-R regressor fares very well in regressing the target parameters, with most R^2 values well over 0.90. This method of analysis provides an intuitive illustration of a large statement - that the multimodal DMTL-R regressor provides a good, sufficiently general estimate to test data. Further, analysis of the trend of losses in Figures 1.7 and 1.8

also suggests that the models are able to maintain generality across test sets.

	Single-Target Regression			Multi-Target Regression		
	DMTL-R	Image Only	Statistics Only	**DMTL-R**	Image Only	Statistics Only
Target Index	**ResNet50**					
1	0.979 ± .0040	0.375 ± .1194	0.969 ± .0011	**0.981** ± .0020	0.660 ± .0630	0.963 ± .0060
2	0.984 ± .0058	0.290 ± .1136	0.974 ±.0009	**0.985** ± .0019	0.688 ± .0485	0.966 ± .0063
3	0.978 ± .0065	0.567 ± .1157	0.963 ± .0012	**0.979** ± .0020	0.608 ± .0621	0.956 ± .0038
4	0.899 ± .0246	0.507 ± .0673	0.895 ± .0053	**0.925** ± .0059	0.657 ± .0435	0.886 ± .0152
5	0.780 ± .0420	0.316 ± .0208	0.763 ± .0019	**0.823** ± .0108	0.405 ± .0408	0.732 ± .0186
6	**0.943** ± .0084	0.505 ± .0405	0.751 ± .0084	0.736 ± .0183	0.522 ± .0575	0.705 ± .0037
	VGG16					
1	0.973 ± .0039	0.797 ± .0210	0.969 ± .0011	**0.980** ± .0022	0.808 ± .0254	0.963 ± .0060
2	0.976 ± .0026	0.805 ± .0171	0.974 ±.0009	**0.983** ± .0021	0.817 ± .0212	0.966 ± .0063
3	0.965 ± .0028	0.821 ± .0270	0.963 ± .0012	**0.976** ± .0029	0.828 ± .0289	0.956 ± .0038
4	0.926 ± .0080	0.684 ± .0509	0.895 ± .0053	**0.939** ± .0059	0.735 ± .0390	0.886 ± .0152
5	**0.843** ± .0342	0.340 ± .1703	0.763 ± .0019	0.803 ± .0366	0.512 ± .1046	0.732 ± .0186
6	**0.786** ± .0196	0.620 ± .0312	0.751 ± .0084	0.752 ± .0231	0.786 ± .0196	0.705 ± .0037
	InceptionV3					
1	**0.983** ± .0035	0.541 ± .0607	0.969 ± .0011	0.957 ± .0043	0.439 ± .0668	0.963 ± .0060
2	**0.987** ± .0018	0.546 ± .0694	0.974 ± .0009	0.956 ± .0049	0.439 ± .0695	0.966 ± .0063
3	**0.978** ± .0020	0.509 ± .0379	0.963 ± .0012	0.950 ± .0048	0.446 ± .0521	0.956 ± .0038
4	**0.903** ± .0077	0.530 ± .0313	0.895 ± .0053	0.853 ± .0066	0.488 ± .0469	0.886 ± .0152
5	**0.794** ± .0174	0.315 ± .0285	0.763 ± .0019	0.728 ± .0334	0.321 ± .0423	0.732 ± .0186
6	**0.943** ± .0053	0.425 ± .0343	0.751 ± .0084	0.777± .0305	0.347 ± .0261	0.705 ± .0037

Table 1.1: Comparison of R^2 values (with 95% confidence intervals) for single-target and multi-target regression via DMTL-R (our approach) described in Figure 1.4. R^2 values are after 20 training epochs. Results are derived from each of the ResNet [1], VGG16 [2], and Inception [3] architectures.

For comparison, in Table 1.1, **Image-only** statistics are fine-tuned CNN regressors trained without descriptive statistics with varying transfer learned architectures as shown, which are described briefly in section 1.4.1. **Statistics-only** columns refer to a regressor trained without the image component. These are fully-connected network regressors with hidden layer sizes $[d_{input}, 100, 100, 50, d_{output}]$, which mimics the input to the descriptive statistic component of the DMTL-R model. This is trained with identical hyperparameters for learning rate, epochs, train/test split ratios, and training batch size. These results are shown in each row only for comparison and do not vary with CNN architecture.

It is interesting to note that, in most instances, the DMTL-R approach was able to improve upon the image or statistic only fits. This confirms the hypothesis that an estimator which is capable of including both modes of information, such as the DMTL-R model, is a more exhaustive and complete model. With how DMTL-R is trained and the ability to generalize, we believe DMTL-R is the superior model for the task of multimodal image-descriptor regression in the presence of small training data. We demonstrate with an improvement in R^2 by a range of 4-5% over statistics-only and 59-117% over image-only regression.

1.5 Conclusion

In this paper, we presented a Deep Multimodal Transfer-Learned Regressor (DMTL-R) for predicting target parameters in data-poor domains. The inputs to the regressor are images and a corresponding n-tuple of statistics containing information we know to be true about the image from another data domain. The suggested DMTL-R approach, built around a pre-trained CNN, featurizes the image then conditions its features with corresponding descriptive statistics. We studied a materials science application, regressing 6 dimensions of output target parameters from input images and 18-tuple input statistics. We found that with the available small training sample, our approach results in better regression accuracy (R^2) of target parameters than a similar model trained over images or descriptive parameters alone. Here we have demonstrated the efficacy of DMTL-R on materials data, but the model could be extended to other data-poor domains such as healthcare, climatology, and beyond.

2. Self-Adaptive Physically-Informed Neural Networks

2.1 Introduction

As part of the burgeoning field of scientific machine learning [24], physics-informed neural networks (PINNs) have emerged recently as an alternative to traditional numerical methods for partial different equations (PDE) [25, 26, 27, 28]. Typical data-driven deep learning methodologies do not take into account physical understanding of the problem domain. The PINN approach is based on a strong physics prior that constrains the output of a deep neural network by means of a system of PDEs. The potential of using neural networks as universal function approximators to solve PDEs had been recognized since the 1990's [29]. However, PINNs promise to take this approach to a different level by using deep neural networks, which is made possible by the vast advances in computational capabilities and training algorithms since that time [30, 31], as well as by the invention of automatic differentiation methods [32, 33].

A great advantage of PINNs over traditional time-stepping PDE solvers is that the entire spatial-temporal domain can be solved at once using collocation points distributed irregularly (rather than on a grid) across the spatial-temporal domain, in a process that can be massively parallelized via GPU. As we have continued to see GPU capabilities increase in recent years, a method that relies on parallelism in training iterations will likely emerge as the predominant approach in scientific computing.

The original continuous PINN algorithm proposed in [25], henceforth referred to as the "baseline PINN" algorithm, is effective at estimating solutions that are reasonably smooth, such as Burgher's equation, the wave equation, Poisson's equation, and Schrodinger's equation. On the other hand, it has been observed that the baseline PINN has convergence and accuracy problems when solving "stiff" PDEs [34] with solutions that contain sharp and intricate space and time transitions [27, 35]. This is the case, for example, of the Allen-Cahn and Cahn-Hilliard equations of phase-field models [36].

To address this issue, various modifications of the baseline PINN algorithm have been proposed. For example, in [27], a series of schemes are introduced, including nonadaptive weighting of the training loss function, adaptive resampling of the collocation points, and time-adaptive approaches, while in [35], a learning rate annealing scheme was proposed. The consensus has been that adaptation mechanisms are essential to make PINNs more stable and able to approximate well difficult regions of the solution.

This paper introduces Self-Adaptive PINNs, a fundamentally new method to train PINNs adaptively, which uses trainable weights as a soft multiplicative mask reminiscent of the attention mechanism used in computer vision [37, 38]. The adaptation weights are trained concurrently with the network weights. As a result, initial, boundary or collocation points in difficult regions of the solution are automatically weighted more in the loss function, forcing the approximation to improve on those points. The basic principle in Self-Adaptive PINNs is to make the weights increase as the corresponding losses do, which is accomplished by training the network to simultaneously minimize the losses and maximize the weights, i.e., to find a saddle point in the cost surface. We show that this is formally equivalent to a penalty-based solution of PDE-constrained optimization methods. Experimental results show that Self-Adaptive PINNs can solve a "stiff" Allen-Cahn PDE with significantly better accuracy than other state-of-the-art PINN algorithms, while using a smaller number of training epochs. We also report in the Appendix results obtained with easier-to-solve Burger's and Helmholtz PDEs, which confirm the trends observed in the Allen-Cahn experiments.

2.2 Background

2.2.1 Physics-Informed Neural Networks

Consider the initial-boundary value problem:

$$\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] \;=\; f(\boldsymbol{x},t)\,,\;\; \boldsymbol{x}\in\Omega\,,\; t\in(0,T]\,, \tag{2.1}$$

$$\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] \;=\; g(\boldsymbol{x},t)\,,\;\; \boldsymbol{x}\in\partial\Omega,\; t\in(0,T]\,, \tag{2.2}$$

$$u(\boldsymbol{x},0) = h(\boldsymbol{x})\,,\;\; \boldsymbol{x}\in\overline{\Omega}\,. \tag{2.3}$$

Here, the domain $\Omega \subset R^d$ in a open set, $\overline{\Omega}$ is its closure, $u : \overline{\Omega} \times [0,T] \to R$ is the desired solution, $\boldsymbol{x} \in \Omega$ is a spatial vector variable, t is time, and $\mathcal{N}_{\boldsymbol{x},t}$ and $\mathcal{B}_{\boldsymbol{x},t}$ are spatial-temporal differential operators. The problem *data* is provided by the forcing function $f : \Omega \to R$, the boundary condition function $g : \partial\Omega \times (0,T]$, and the initial condition function $h : \overline{\Omega} \to R$. Additionally, sensor data in the interior of the domain may be available. In any case, we assume that the data are sufficient and appropriate for a well-posed problem. Time-independent problems and other types of data can be handled similarly, so we will use the equations (1)-(3) as a model.

Following [25], let $u(\boldsymbol{x},t)$ be approximated by the output $u(\boldsymbol{x},t;\boldsymbol{w})$ of a deep neural network with inputs $\boldsymbol{x}$ and t (in the case of a PDE system, this would be a neural network with multiple outputs). The value of $\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t;\boldsymbol{w})]$ and $\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x},t;\boldsymbol{w})]$ can be computed quickly and accurately using *automatic differentiation* methods [32, 33].

The network weights $\boldsymbol{w}$ are trained by minimizing a loss function that penalizes the output for not satisfying (1)-(3):

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w}) + \mathcal{L}_0(\boldsymbol{w}), \tag{2.4}$$

where $\mathcal{L}_s$ is the loss term corresponding to sample data (if any), while $\mathcal{L}_r$, $\mathcal{L}_b$, and $\mathcal{L}_0$ are loss terms corresponding to not satisfying the PDE (2.1), the boundary condition (3.2), and the initial condition (3.3), respectively:

$$\mathcal{L}_s(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_s} |u(\boldsymbol{x}_s^i, t_s^i; \boldsymbol{w}) - y_s^i|^2, \tag{2.5}$$

$$\mathcal{L}_r(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_r} |\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] - f(\boldsymbol{x}_r^i, t_r^i)|^2, \tag{2.6}$$

$$\mathcal{L}_b(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_b} |\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w})] - g(\boldsymbol{x}_b^i, t_b^i)|^2, \tag{2.7}$$

$$\mathcal{L}_0(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N_0} |u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}) - h(\boldsymbol{x}_0^i)|^2. \tag{2.8}$$

where $\{\boldsymbol{x}_s^i, t_s^i, y_s^i\}_{i=1}^{N_s}$ are sensor data (if any), $\{\boldsymbol{x}_0^i\}_{i=1}^{N_0}$ are initial condition points, $\{\boldsymbol{x}_b^i, t_b^i\}_{i=1}^{N_b}$ are boundary condition points, $\{\boldsymbol{x}_r^i, t_r^i\}_{i=1}^{N_r}$ are collocation points randomly distributed in the domain Ω, and N_s, N_0, N_b and N_r denote the total number of sensor, initial, boundary, and collocation points, respectively. The network weights $\boldsymbol{w}$ can be tuned by minimizing the total training loss $\mathcal{L}(\boldsymbol{w})$ via standard gradient descent procedures used in deep learning.

2.2.2 Related Work

The baseline PINN algorithm described in the previous section, though remarkably successful in the solution of linear PDEs and nonlinear PDEs with smooth solutions, can produce inaccurate approximations around sharp space and time transitions in the solutions of "stiff" PDEs. Much of the recent literature on PINNs has been devoted to mitigating these issues by introducing modifications to the baseline PINN algorithm that can increase training stability and accuracy. We mention some of these approaches below.

2.2.2.1 Nonadaptive Weighting

In [27], it was pointed out that a premium should be put on forcing the neural network to satisfy the initial conditions closely, especially for PDEs describing time-irreversible processes, where the solution has to be approximated well early. Accordingly, a loss function of the form $\mathcal{L}(\theta) = \mathcal{L}_r(\theta) + \mathcal{L}_b(\theta) + C\,\mathcal{L}_0(\theta)$ was suggested, where $C \gg 1$ is a hyperparameter.

2.2.2.2 Learning Rate Annealing

In [35], it is argued that the optimal value of the weight C in the previous scheme may vary wildly among different PDEs so that choosing its value would be difficult. Instead they propose to use weights that are tuned during training using statistics of the backpropagated gradients of the loss function. It is noteworthy that the weights themselves are not adjusted by backpropagation. Instead, they behave as learning rate coefficients, which are updated after each epoch of training.

2.2.2.3 Adaptive Resampling

In [27], a strategy to adaptively resample the residual collocation points based on the magnitude of the residual is proposed. While this approach improves the approximation, the training process must be interrupted and the MSE evaluated on the residual points to deterministically resample the ones with the highest error. After each resampling step, the number of residual points grows, increasing computational complexity.

2.2.2.4 Stochastic Gradient Descent

A training procedure where a different subset of collocation points are randomly sampled at each iteration was proposed by [35]. While stochastic gradient descent approaches a global minimum in an infinite limit [39], it is a random method that relies on sufficient random sampling and an large training horizon, which may be computationally intractable.

2.2.2.5 Time-Adaptive Approaches

In [27], another method is suggested, which divides the time axis into several smaller intervals, and trains PINNs separately on them, either sequentially or in parallel. This approach is time-consuming due to the need to train multiple PINNs.

2.2.2.6 Neural Tangent Kernel (NTK) Weighting

Most recently, [28] introduced weights on the collocation and boundary losses, which are updated via the eigenvalues of the neural tangent kernel matrix.

2.3 Self-Adaptive Physics-Informed Neural Networks

While the various previously proposed weighting methods produce improvements in stability and accuracy over the baseline PINN, they are either nonadaptive or require brute-force adaptation at increased computational cost. Here we propose a simple procedure that uses fully-trainable weights to produce a multiplicative soft attention mask, in a manner that is reminiscent of attention mechanisms used in computer vision [37, 38]. Instead of hard-coding weights at particular regions of the solution, the proposed method is in agreement with the neural network philosophy of self-adaptation, where the weights in the loss function are updated by gradient descent side-by-side with the network weights.

Using the PDE in (1)-(3) as reference, the proposed Self-adaptive PINN utilizes the following loss function

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\lambda}_r) + \mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) + \mathcal{L}_0(\boldsymbol{w}, \boldsymbol{\lambda}_0)\,, \tag{2.9}$$

where $\boldsymbol{\lambda}_r = (\lambda_r^1, \ldots, \lambda_r^{N_r})$, $\boldsymbol{\lambda}_b = (\lambda_b^1, \ldots, \lambda_b^{N_b})$, and $\boldsymbol{\lambda}_0 = (\lambda_0^1, \ldots, \lambda_0^{N_0})$ are trainable, nonnegative *self-adaptation weights* for the initial, boundary, and collocation points, respectively, and

$$\mathcal{L}_r(\boldsymbol{w}, \boldsymbol{\lambda}_r) \;=\; \frac{1}{2}\sum_{i=1}^{N_r} m(\lambda_r^i)\,|\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] - f(\boldsymbol{x}_r^i, t_r^i)|^2 \tag{2.10}$$

$$\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) \;=\; \frac{1}{2}\sum_{i=1}^{N_b} m(\lambda_b^i)\,|\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] - g(\boldsymbol{x}_b^i, t_b^i)|^2 \tag{2.11}$$

$$\mathcal{L}_0(\boldsymbol{w}, \boldsymbol{\lambda}_0) \;=\; \frac{1}{2}\sum_{i=1}^{N_0} m(\lambda_0^i)\,|u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}) - h(\boldsymbol{x}_0^i)|^2. \tag{2.12}$$

where the *self-adaptation mask function* $m(\lambda)$ defined on $[0, \infty)$ is a nonnegative, differentiable, strictly increasing function of λ. A key feature of self-adaptive PINNs is that the loss $\mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0)$ is minimized with respect to the network weights $\boldsymbol{w}$, as usual, but is *maximized* with respect to the self-adaptation weights $\boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0$, i.e., the objective is:

$$\min_{\boldsymbol{w}} \max_{\boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0} \mathcal{L}(\boldsymbol{w}, \boldsymbol{\lambda}_r, \boldsymbol{\lambda}_b, \boldsymbol{\lambda}_0)\,. \tag{2.13}$$

Consider the updates of a gradient descent/ascent approach to this problem:

$$\boldsymbol{w}^{k+1} \;=\; \boldsymbol{w}^k \;-\; \eta_k\, \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \tag{2.14}$$

$$\boldsymbol{\lambda}_r^{k+1} \;=\; \boldsymbol{\lambda}_r^k \;+\; \eta_k\, \nabla_{\boldsymbol{\lambda}_r} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \tag{2.15}$$

$$\boldsymbol{\lambda}_b^{k+1} \;=\; \boldsymbol{\lambda}_b^k \;+\; \eta_k\, \nabla_{\boldsymbol{\lambda}_b} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \tag{2.16}$$

$$\boldsymbol{\lambda}_0^{k+1} \;=\; \boldsymbol{\lambda}_0^k \;+\; \eta_k\, \nabla_{\boldsymbol{\lambda}_0} \mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k)\,. \tag{2.17}$$

where η_k is the learning rate at step k.

$$\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}(\boldsymbol{w}^k,\boldsymbol{\lambda}_r^k,\boldsymbol{\lambda}_b^k,\boldsymbol{\lambda}_0^k) = \frac{1}{2}\begin{bmatrix} m'(\lambda_r^{k,1})\left|\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i,t_r^i;\boldsymbol{w}^k)] - f(\boldsymbol{x}_r^1,t_r^1)\right|^2 \\ \cdots \\ m'(\lambda_r^{k,N_r})\left|\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i,t_r^i;\boldsymbol{w}^k)] - f(\boldsymbol{x}_r^{N_r},t_r^{N_r})\right|^2 \end{bmatrix}, \tag{2.18}$$

$$\nabla_{\boldsymbol{\lambda}_b}\mathcal{L}(\boldsymbol{w}^k,\boldsymbol{\lambda}_r^k,\boldsymbol{\lambda}_b^k,\boldsymbol{\lambda}_0^k) = \frac{1}{2}\begin{bmatrix} m'(\lambda_b^{k,1})\left|\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i,t_b^i;\boldsymbol{w}^k)] - g(\boldsymbol{x}_b^1,t_b^1)\right|^2 \\ \cdots \\ m'(\lambda_b^{k,N_b})\left|\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i,t_b^i;\boldsymbol{w}^k)] - g(\boldsymbol{x}_b^{N_b},t_b^{N_b})\right|^2 \end{bmatrix}, \tag{2.19}$$

$$\nabla_{\boldsymbol{\lambda}_0}\mathcal{L}(\boldsymbol{w}^k,\boldsymbol{\lambda}_r^k,\boldsymbol{\lambda}_b^k,\boldsymbol{\lambda}_0^k) = \frac{1}{2}\begin{bmatrix} m'(\lambda_0^{k,1})\left|u(\boldsymbol{x}_0^1,0;\boldsymbol{w}^k) - h(\boldsymbol{x}_0^1,t_0^1)\right|^2 \\ \cdots \\ m'(\lambda_0^{k,N_0})\left|u(\boldsymbol{x}_0^i,0;\boldsymbol{w}^k)] - h(\boldsymbol{x}_0^{N_0})\right|^2 \end{bmatrix}. \tag{2.20}$$

Hence, since $m'(\lambda) > 0$ (the mask function is strictly increasing, by assumption), then $\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_b}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_0}\mathcal{L} \geq 0$, and any gradient component is zero if and only if the corresponding unmasked loss is zero. This shows that the sequences of weights $\{\boldsymbol{\lambda}_r^k; k = 1,2,\ldots\}$, $\{\boldsymbol{\lambda}_b^k; k = 1,2,\ldots\}$, $\{\boldsymbol{\lambda}_0^k; k = 1,2,\ldots\}$ (and the associated mask values) are monotonically increasing, provided that the corresponding unmasked losses are nonzero. Furthermore, the magnitude of the gradients $\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_b}\mathcal{L}, \nabla_{\boldsymbol{\lambda}_0}\mathcal{L}$, and therefore of the updates, are larger if the corresponding unmasked losses are larger. This progressively penalizes the network more for not fitting the residual, boundary, and initial points closely (the self-adaptive weights, i.e., the amount of penalty, is are typically initialized to small nonzero values). We remark that any of the weights can be set to fixed, non-trainable values, if desired. For example, by setting $\lambda_b^k \equiv 1$, only the weights of the initial and collocation points would be trained. The sensor data loss is not masked, as these data consist of noisy observations, and weighting them requires extra care to avoid overfitting.

The shape of the function g affects mask sharpness and training of the PINN. Examples include polynomial masks $m(\lambda) = c\lambda^q$, for $c, q > 0$, and sigmoidal masks. See Figure 2.1 for a few examples. In practice, the polynomial mask functions have to be kept below a suitable (large) value, to avoid numerical overflow. The sigmoidal masks do not have this issue, and can be used to pro-

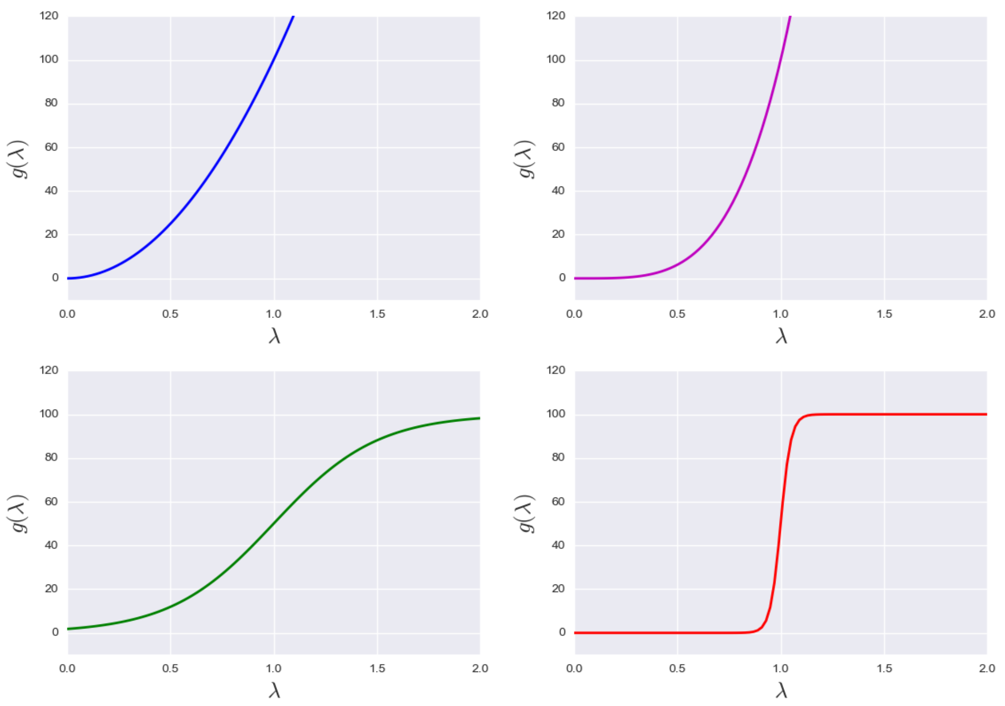


Figure 2.1: Mask function examples. From the upper left to the bottom right: polynomial mask, $q = 2$; polynomial mask, $q = 4$; smooth logistic mask; sharp logistic mask.

duce sharp masks. For example, in the bottom right example in Figure 2.1, the mask is essentially binary; it starts small for small starting values of the self-adaptive weight λ, and after these exceed a certain threshold, the mask value will quickly take on the upper saturation value. Similarly to neural network nonlinearities, sigmoid mask functions can suffer from vanishing gradients during training. This is particularly a problem at the lower starting value. Therefore, excessively sharp sigmoidal mask functions should be avoided.

For another perspective, consider the following PDE-constrained optimization problem

$$\min \ \frac{1}{2}\sum_{i=1}^{N_s} |u(\boldsymbol{x}_s^i, t_s^i; \boldsymbol{w}) - y_s^i|^2 \tag{2.21}$$

subject to

$$\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})] \ = \ f(\boldsymbol{x}_r^i, t_r^i),, \ \ i = 1, \dots, N_r \tag{2.22}$$

$$\mathcal{B}_{\boldsymbol{x},t}[u(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w})] \ = \ g(\boldsymbol{x}_r^b, t_b^i),, \ \ i = 1, \dots, N_b \tag{2.23}$$

$$u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}) \ = \ h(\boldsymbol{x}_r^i),, \ \ i = 1, \dots, N_0 \tag{2.24}$$

$$\tag{2.25}$$

Formally, Self-Adaptive PINN training corresponds to a *penalty method* to solve the previous optimization problem [40]. In a typical penalty optimization method, a constrained problem

$$\min \ \ L(\boldsymbol{x}) \tag{2.26}$$

subject to

$$r(\boldsymbol{x}) = 0 \tag{2.27}$$

is solved as a sequence of unconstrained problems

$$\min \ \ L(\boldsymbol{x}) + c^k P(\boldsymbol{x}), \quad k = 1, 2, \dots, \tag{2.28}$$

where $c^1 < c^2 < \dots$ is a fixed sequence of increasing penalty costs, and $P(\boldsymbol{x})$ is a suitable *penalty function*, which is small in the feasible region $R = \{\boldsymbol{x} \mid r(\boldsymbol{x}) = 0\}$, and large outside of it. A typical choice is the polynomial penalty $P(\boldsymbol{x}) = |r(\boldsymbol{x})|^p$, for $p > 1$. It can be shown that a limit point of a sequence of solutions of the unconstrained problem, where the solution at step k is used as the initialization for step $k+1$, is a solution of the original constrained problem [40].

One can see Self-Adaptive PINN training as a penalty method, with costs $c^k = m(\lambda^k)$. The self-adaptation weights, and mask values, produce increasing penalty costs that are not selected a-priori, but are adaptively updated by the neural network training procedure. Although the use of neural networks in regular penalty-based constrained optimization has been suggested [41, 42], neural networks have not been used in PDE-constrained problems, as far as we know.

The gradient ascent/descent step can be implemented easily using off-the-self neural network software, by simply flipping the sign of $\nabla_{\boldsymbol{\lambda}_r}\mathcal{L}$, $\nabla_{\boldsymbol{\lambda}_b}\mathcal{L}$, and $\nabla_{\boldsymbol{\lambda}_0}\mathcal{L}$. In our implementation of Self-Adaptive PINNs, we use Tensorflow 2.3 with a fixed number of iterations of Adam [20]. In some case, these are followed by another fixed number of iterations of the L-BFGS quasi-newton method [43]. This is consistent with the baseline PINN formulation in [25], as well as follow-up literature [27]. However, the adaptive weights are only updated in the Adam training steps, and are held constant during L-BFGS training, if any. A full implementation of the methodology described here has been made publicly available by the authors[1] and it is included in the open-source software *TensorDiffEq* [44].

2.4 Allen-Cahn Reaction-Diffusion PDE

In this section, we report experimental results obtained with the Allen-Cahn PDE, which contrast the performance of the proposed Self-Adaptive PINN algorithm against the baseline PINN and two of the PINN algorithms mentioned in Section 2.2.2, namely, the nonadaptive weighting and time-adaptive schemes (for the latter, Approach 1 in [27] was used).

The main figure of merit used is the L2-error:

$$L_2 \text{ error } = \frac{\sqrt{\sum_{i=1}^{N_U} |u(x_i,t_i) - U(x_i,t_i)|^2}}{\sqrt{\sum_{i=1}^{N_U} |U(x_i,t_i)|^2}}. \tag{2.29}$$

where $u(x,t)$ is the trained approximation, and $U(x,t)$ is a high-fidelity solution over a mesh $\{x_i, t_i\}$ containing N_U points. We repeat the training process over a number of random restarts and report the average L2 error and its standard deviation.

The Allen-Cahn reaction-diffusion PDE is typically encountered in phase-field models, which can be used, for instance, to simulate the phase separation process in the microstructure evolution

[1]https://github.com/levimcclenny/SA-PINNs

of metallic alloys [36, 45, 46]. The Allen-Cahn PDE considered here is specified as follows:

$$u_t - 0.0001u_{xx} + 5u^3 - 5u = 0\,, \;\; x \in [-1, 1],\; t \in [0, 1]\,, \tag{2.30}$$

$$u(x, 0) = x^2 cos(\pi x)\,, \tag{2.31}$$

$$u(t, -1) = u(t, 1)\,, \tag{2.32}$$

$$u_x(t, -1) = u_x(t, 1)\,. \tag{2.33}$$

The Allen-Cahn PDE is an interesting benchmark for PINNs for multiple reasons. It is a "stiff" PDE that challenges PINNs to approximate solutions with sharp space and time transitions, and is also introduces periodic boundary conditions (3.15, 3.16). In order to deal with the latter, the boundary loss function $\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b)$ in (2.11) is replaced by

$$\mathcal{L}_b(\boldsymbol{w}, \boldsymbol{\lambda}_b) \;=\; \frac{1}{N_b}\sum_{i=1}^{N_b} g(\lambda_b^i)(|u(1, t_b^i) - u(-1, t_b^i)|^2 + |u_x(1, t_b^i) - u_x(-1, t_b^i)|^2) \tag{2.34}$$

The neural network architecture is fully connected with layer sizes $[2, 128, 128, 128, 128, 1]$. (The 2 inputs to the network are (x, t) pairs and the output is the approximated value of u_θ.) This architecture is identical to [27], in order to allow a direct comparison of performance. We set the number of collocation, initial, and boundary points to $N_r = 20,000, N_0 = 100$ and $N_b = 100$, respectively (due to the periodic boundary condition, there are in fact 200 boundary points). Here we hold the boundary weights w_b^i at 1, while the initial weights w_0^i and collocation weights w_r^i are trained. The initial and collocation weights are initialized from a uniform distribution in the intervals $[0, 100]$ and $[0, 1]$, respectively. Training took 65ms/iteration on a single Nvidia V100 GPU.

Numerical results obtained with the Self-Adaptive PINN are displayed in figure 2.2. The average L2 error across 10 runs with random restarts was 2.1% $\pm$ 1.21%, while the L2 error on 10 runs obtained by the time-adaptive approach in [27] was 8.0% $\pm$ 0.56%. Neither the baseline PINN nor the nonadaptive weighted scheme, with initial condition weight $C = 100$, were able to solve

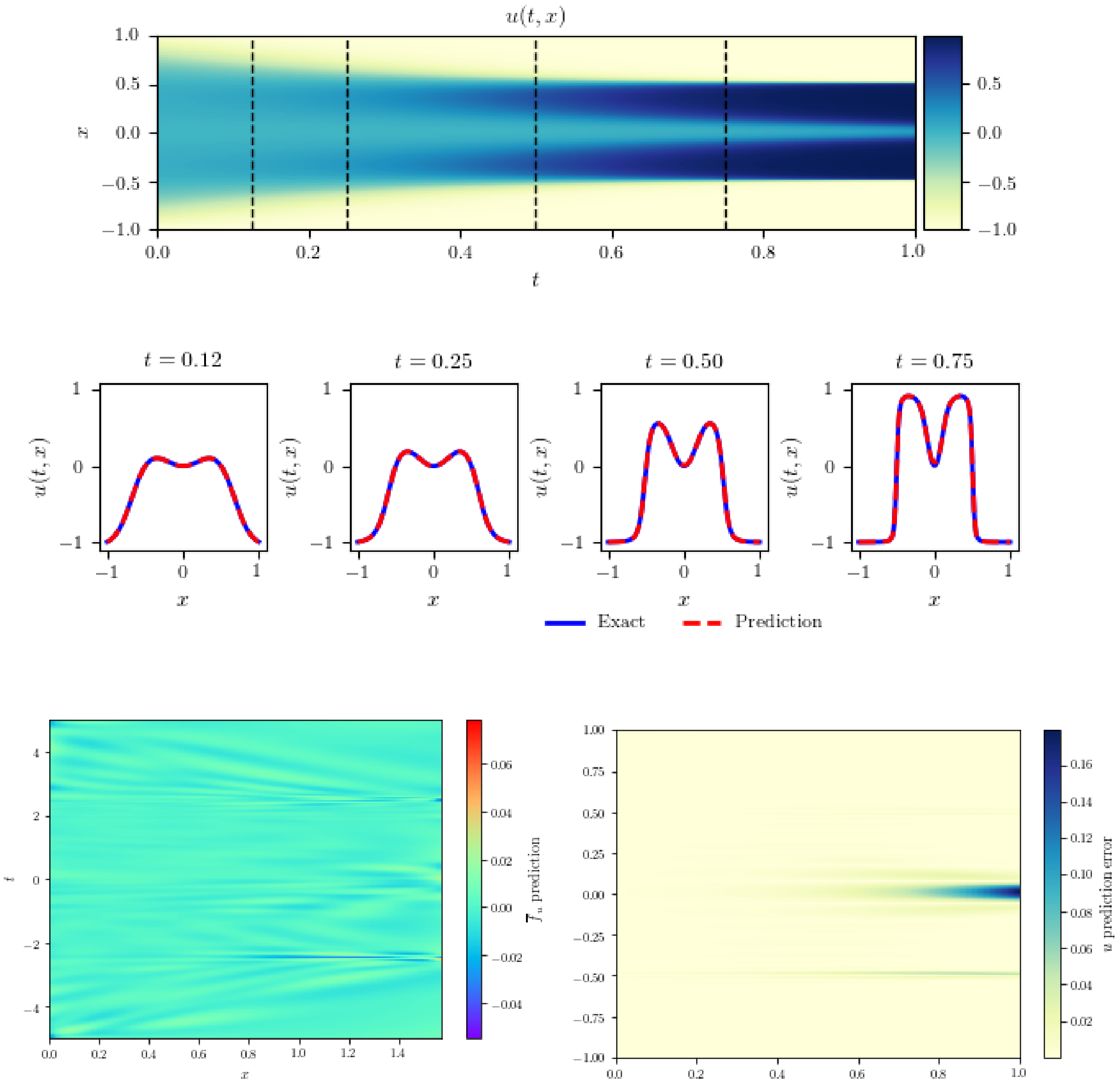


Figure 2.2: *Top:* Plot of the approximation $u(x, t)$ via the self-adaptive PINN. *Middle:* Snapshots of the approximation $u(x, t)$ vs. the high-fidelity solution $U(x, t)$ at various time points through the temporal evolution. *Bottom left:* Residual $r(x, t)$ across the spatial-temporal domain. As expected, it is close to 0 for the whole domain Ω. *Bottom right:* Absolute error between approximation and high-fidelity solution across the spatial-temporal domain.

this PDE satisfactorily, with L2 errors 96.15% $\pm$ 6.45% and 49.61% $\pm$ 2.50%, respectively (these numbers matched almost exactly those reported in [27]).

Figure 2.3 is unique to the proposed self-adaptive PINN algorithm. It displays the trained weights for the collocation points across the spatio-temporal domain. These are the weights of the multiplicative soft attention mask self-imposed by the PINN. This plot stays remarkably constant across different runs with random restarts, which is an indication that it is a property of the particular PDE being solved. We can observe that in this case, more attention is needed early in the solution, but not uniformly across the space variable. In [27], this observation was justified by the fact that the Allen-Cahn PDEs describes a time-irreversible diffusion-reaction processes, where the solution has to be approximated well early. However, here this fact is "discovered" by the self-adaptive PINN itself.

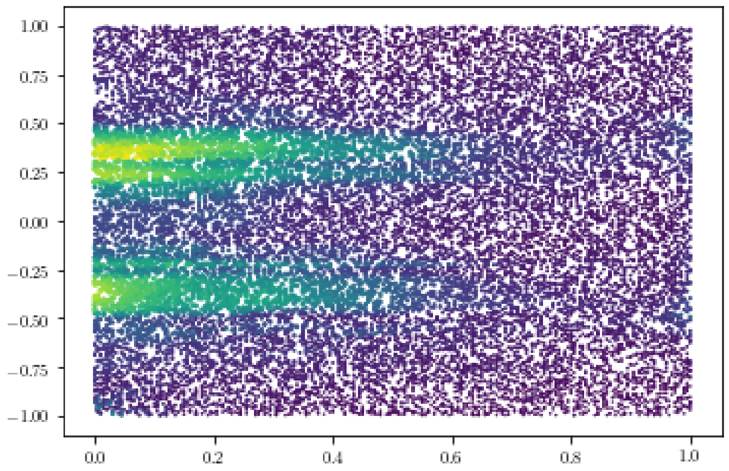


Figure 2.3: Learned weights across the spatio-temporal domain. Brighter colors and larger points indicate larger weights.

2.4.1 Average Weights with Time

Directly related to the figure shown in 2.3 is that shown in figure 2.4, a plot of average collocation points from various partitions of the solution domain. While all the weights are monotonically increasing (a behavior expected by the selection of the weight function λ^2), the rate of increasing is of importance. Note that early in the domain of the Allen-Cahn example we note that the weights increase much faster than in later parts of the domain, and that the initial condition weights increase the fastest. This indicates that the SA-PINN has resolved that the earlier components of the solution are the most important, and more heavily weights those components. A likely explaination is that, since the Allen-Cahn problem is time-diffusive, the only reliable way to generate an accurate solution in later time steps is by generating a reasonably accurate one early on. This is not to be confused with a typical time-marchin approach, where earlier time steps are solved prior to later ones, as out approach does solve all the collocation points at once, however this phenemonon does confirm that early time evolution in the solution could be viewed as “more important" than later time evolution.

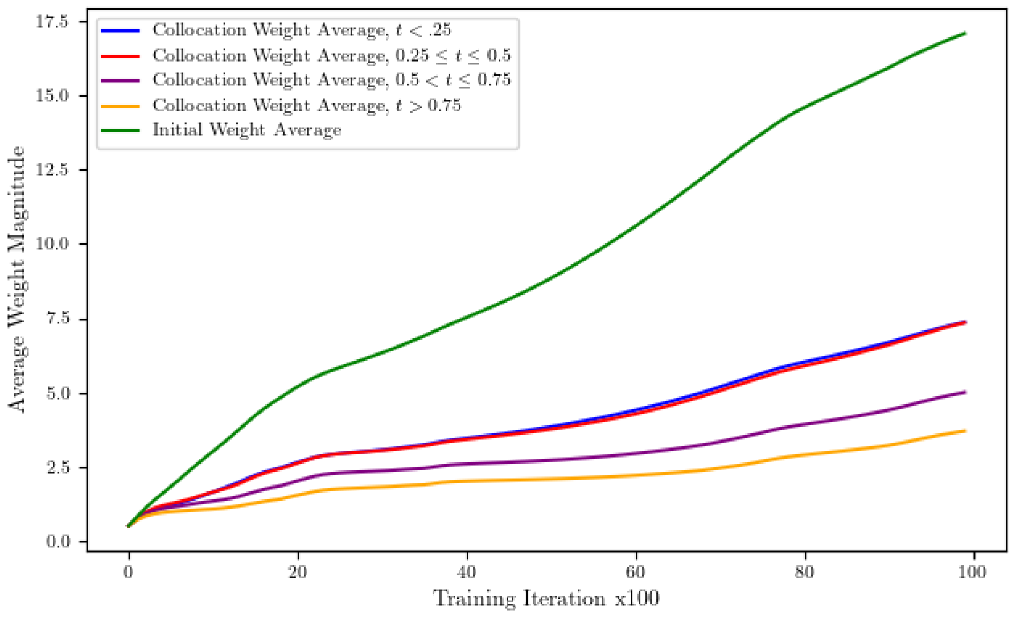


Figure 2.4: Average learned collocation weights across various partitions of the solution domain. Note that earlier times require heavier weighting, with the highest average weights being the initial condition weights. This is consistent with the rationale that earlier solutions must be correctly learned for time-diffusive processes.

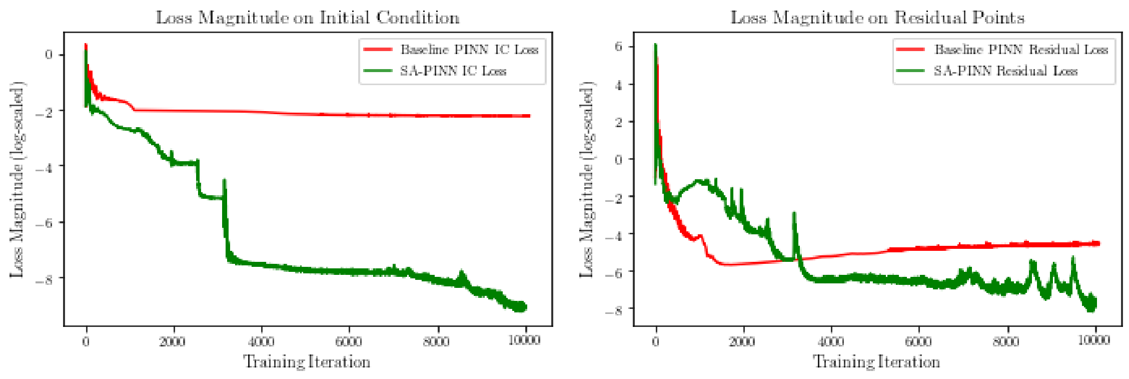


Figure 2.5: Average loss magnitude on the initial condition and residual points over 10k Adam training iterations. For the SA-PINN loss, the weights were removed from the loss value to provide a consistent comparison.

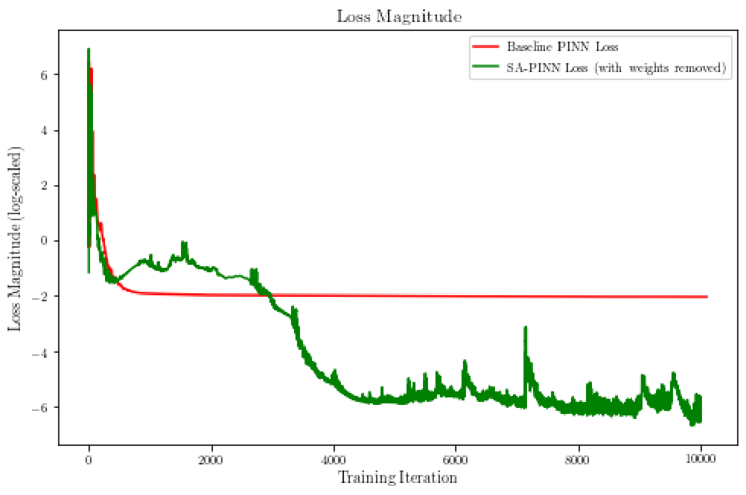


Figure 2.6: Average loss magnitude on the initial condition and residual points over 10k Adam training iterations. For the SA-PINN loss, the weights were removed from the loss value to provide a consistent comparison.

2.5 Self-Adaptive PINNs with Stochastic Gradient Descent

Stochastic gradient descent (SGD) [47] uses randomly sampled subsets of the training data to compute approximations to the gradient for training neural networks by gradient descent [48]. It has been claimed that the empirical superior performance of stochastic gradient descent over large-batch training is due to a tendency of the latter to converge to “sharp” minima in the loss surface, which have poor performance, while SGD with small batches converge to better "flat" minima [49].

The issue has not been well studied in the context of PINNs at the time of writing, though there is some empirical evidence that SGD can indeed improve the L_2 performance of PINNs with some PDEs. It should be pointed out that PINNs are well-suited to SGD since a new set of collocation, initial and boundary points can be sampled each time rather than subsampling a given set of training data points as in conventional machine learning. This is an important difference from SGD in its common vernacular, where minibatches of predefined training data are sampled to train. In the context of PINNs, it is possible to completely resample points from the domain that the training process has *never* seen before, on *each* training iteration. This rapidly increases convergence when utilized, as shown in table 2.1.

The baseline self-adaptive PINN algorithm described previously cannot take advantage of small-batch SGD since the self-adaptive weights are attached to specific training points. In this section, we examine an extension of self-adaptive PINN that allows the use of SGD. The basic idea is to use a spatial-temporal predictor of the value of self-adaptive weights for the newly sampled points. Here we use standard Gaussian processes regression due to its flexibility and power. The GP regressor acts as a *weight generating function*, extending the SA-PINN to a continuous domain, as opposed to individual weights being assigned to individual points. This has the added benefit of creating a continuous weighting map, which can be assessed for uncertainty quantification in the idealized weights of the PINN loss function.

Implementation of the GPR-based SA-PINN is a relatively straightforward modification of the original SA-PINN algorithm. After a set number of iterations (i.e. 500 epoch of training) the SA weights are used for supervised training of a Gaussian process that generates SA weights for each combination of $\{x_f, t_f\}$. A separate GP weight generating function is trained for ICs and BCs to facilitate convergence.

A problem where GPR SA-PINN based SGD seems to have a strong impact is the 1D wave equation:

$$u_{tt}(x,t) - 4 \, = \, 0 \,, \;\; x \in [0,1] \,, \; t \in [0,1] \,, \tag{2.35}$$

$$u(0,t) = 0 \,, u(1,t) = 0, \; t \in [0,1] \,, \tag{2.36}$$

$$u_t(x,0) = 0 \,, \; x \in [0,1] \,, \tag{2.37}$$

$$u(x,0) = \sin(\pi x) + \frac{1}{2}\sin(4\pi x) \,, \; x \in [0,1] \,. \tag{2.38}$$

This problem was considered in [28] to study their NTK weighting scheme. Here, we adopt the same initial condition, velocity parameter, and training sample sizes as in [28]. The problem has an analytical solution:

$$u(x,t) \; = \; \sin(\pi x)\cos(2\pi t) + \frac{1}{2}\sin(4\pi x)\cos(8\pi t) \,, \;\; x \in [0,1] \,, \; t \in [0,1] \,. \tag{2.39}$$

Results from Self-Adaptive SGD training are shown in figures 2.7 and 2.8. The L2 error across 10 trials is listed in table 2.1.

PINN method	No SGD	SGD
baseline	0.3792 ± 0.0162	0.4513 ± 0.0255
fixed weights	0.7296 ± 0.1421	0.2079 ± 0.0624
self-adaptive	0.7971 ± 0.0497	$\mathbf{0.0295 \pm 0.0070}$

Table 2.1: Wave PDE results.

2.6 Neural Tangent Kernel Training Dynamics Analysis

In this section, we investigate the dynamics of self-adaptive PINN training by studying its neural tangent kernel (NTK). First, note that (14) can be written as

$$\frac{\boldsymbol{w}^{k+1} - \boldsymbol{w}^k}{\eta_k} \; = \; -\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}^k, \boldsymbol{\lambda}_r^k, \boldsymbol{\lambda}_b^k, \boldsymbol{\lambda}_0^k) \,. \tag{2.40}$$

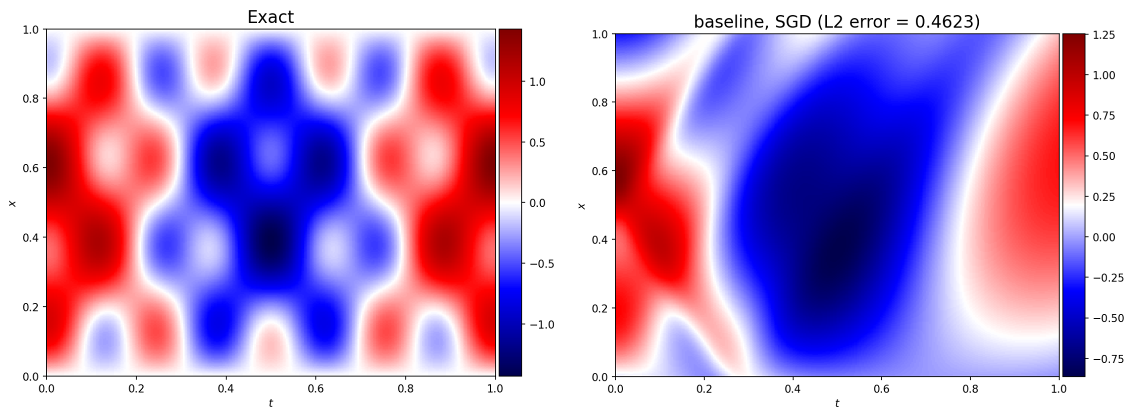


Figure 2.7: Exact 1D wave solution vs. baseline SGD training over 80k Adam iterations

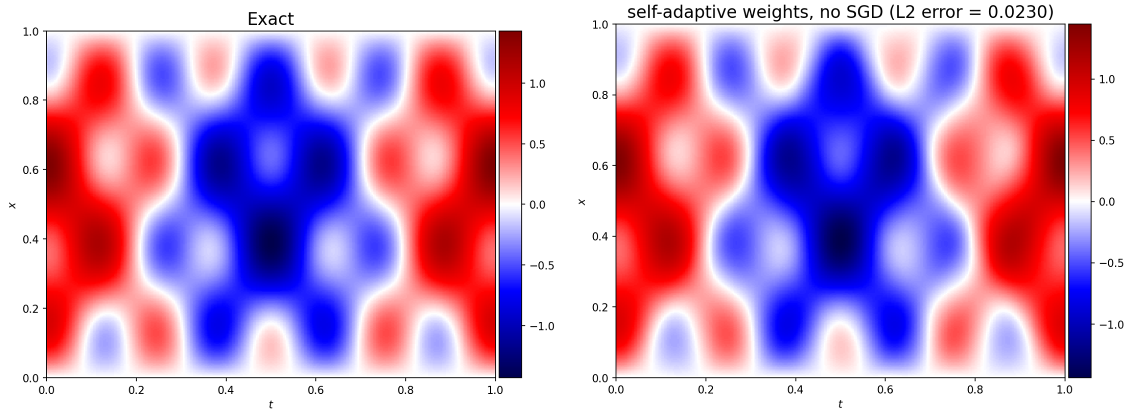


Figure 2.8: Exact 1D wave solution vs. Self-Adaptive SGD training over 80k Adam iterations

In the limit as the learning rate η_k tends to zero, the previous expression yields the *gradient flow* differential equation [**?**]:

$$\frac{d\boldsymbol{w}(\tau)}{d\tau} = -\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}(\tau), \boldsymbol{\lambda}_r(\tau), \boldsymbol{\lambda}_b(\tau), \boldsymbol{\lambda}_0(\tau)), \tag{2.41}$$

where $\tau \geq 0$ denotes the (continuous) training time. Notice that the usual gradient descent step corresponds to a forward Euler discretization of (2.41). It follows that the properties of gradient descent optimization can be investigated by studying this differential equation.

Under this vanishing learning-rate limit, the *neural tangent kernel* (NTK) [**?**] characterizes the

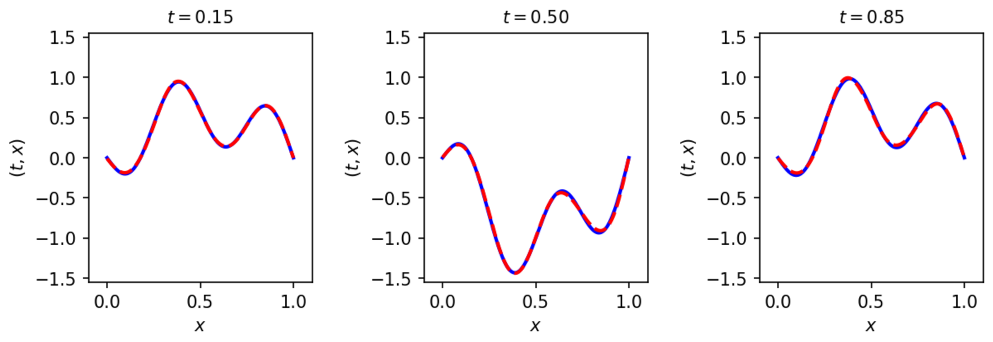


Figure 2.9: Cross-sections of the domain of 1D wave with GP-SA SGD training. L2 error on this run is 2%.

training dynamics of the neural network, i.e., the evolution of the output $u(\boldsymbol{x}, t; \boldsymbol{w}(\tau))$ as a function of training time τ. In [50], the NTK for PINNs was derived and its properties were studied. Here we show how those results are modified by the introduction of self-adaptive weights in the loss function.

Let the response vectors be

$$\mathbf{u}_s(\tau) = [u(\boldsymbol{x}_s^1, t_s^1; \boldsymbol{w}(\tau)), \dots, u(\boldsymbol{x}_s^{N_s}, t_s^{N_s}; \boldsymbol{w}(\tau))]^T \tag{2.42}$$

$$\mathbf{u}_r(\tau) = [N_{x,t}[u(\boldsymbol{x}_r^1, t_r^1; \boldsymbol{w}(\tau))], \dots, N_{x,t}[u(\boldsymbol{x}_r^{N_r}, t_r^{N_r}; \boldsymbol{w}(\tau))]]^T \tag{2.43}$$

$$\mathbf{u}_b(\tau) = [B_{x,t}[u(\boldsymbol{x}_b^1, t_b^1; \boldsymbol{w}(\tau))], \dots, B_{x,t}[u(\boldsymbol{x}_r^{N_b}, t_b^{N_b}; \boldsymbol{w}(\tau))]]^T \tag{2.44}$$

$$\mathbf{u}_0(\tau) = [u(\boldsymbol{x}_0^1, 0; \boldsymbol{w}(\tau)), \dots, u(\boldsymbol{x}_0^{N_0}, 0; \boldsymbol{w}(\tau))]^T \tag{2.45}$$

Likewise, the data vectors are denoted by

$$\mathbf{v}_s = [y_s^1, \dots, y_s^{N_s}]^T \tag{2.46}$$

$$\mathbf{v}_r = [f(\boldsymbol{x}_r^1, t_r^1), \dots, f(\boldsymbol{x}_r^{N_r}, t_r^{N_r})]^T \tag{2.47}$$

$$\mathbf{v}_b = [g(\boldsymbol{x}_b^1, t_b^1), \dots, g(\boldsymbol{x}_r^{N_b}, t_b^{N_b})]^T \tag{2.48}$$

$$\mathbf{v}_0 = [h(\boldsymbol{x}_0^1, 0), \dots, h(\boldsymbol{x}_0^{N_0}, 0)]^T \tag{2.49}$$

We will write $\mathbf{u}_p(\tau) = (u_p^1(\tau), \ldots, u_p^{N_p}(\tau))$ and $\mathbf{v}_p = (v_p^1, \ldots, v_p^{N_p})$ to identify the individual responses $u_p^i(\tau)$ and data point v_p^i, for $p = s, r, b, 0$.

The loss function at training time τ can be written as

$$\mathcal{L}(\boldsymbol{w}(\tau), \boldsymbol{\lambda}_r(\tau), \boldsymbol{\lambda}_b(\tau), \boldsymbol{\lambda}_0(\tau)) = \frac{1}{2}\sum_{j=1}^{N_s} |u_s^j(\tau) - v_s^j|^2 \tag{2.50}$$

$$+ \frac{1}{2}\sum_{q=r,b,0}\sum_{j=1}^{N_q} m(\lambda_q^j(\tau))\, |u_q^j(\tau) - v_q^j|^2 \tag{2.51}$$

Hence, the gradient flow in (30) becomes

$$\frac{d\mathbf{w}}{d\tau} = -\sum_{j=1}^{N_s} \nabla_{\boldsymbol{w}} u_s^j(\tau)(u_s^j(\tau) - v_s^j) - \sum_{q=r,b,0}\sum_{j=1}^{N_q} \nabla_{\boldsymbol{w}} u_p^j(\tau) m(\lambda_q^j(\tau))\,(u_q^j(\tau) - v_q^i) \tag{2.52}$$

$$= -\mathbf{J}_s^T(\tau)(\mathbf{u}_s(\tau) - \mathbf{v}_s) - \sum_{q=r,b,0} \mathbf{J}_q^T(\tau)\boldsymbol{\Gamma}_q(\tau)(\mathbf{u}_q(\tau) - \mathbf{v}_q) \tag{2.53}$$

where $\mathbf{J}_p(\tau)$ is the Jacobian of $\mathbf{u}_p(\tau)$ with respect to $\boldsymbol{w}$, for $p = s, r, b, 0$, and $\boldsymbol{\Gamma}_p(\tau)$ is a diagonal matrix of dimension $N_p \times N_p$ containing the self-adaptive mask values $m(\lambda_p^1(\tau)), \ldots, m(\lambda_p^{N_p}(\tau))$ in the diagonal, for $p = r, b, 0$.

It follows that

$$\begin{aligned}\frac{d\mathbf{u}_p(\tau)}{d\tau} &= \mathbf{J}_p(\tau) \cdot \frac{d\mathbf{w}(\tau)}{d\tau} \\ &= -\mathbf{J}_p(\tau)\mathbf{J}_s^T(\tau)(\mathbf{u}_s(\tau) - \mathbf{v}_s) - \sum_{q=r,b,0} \mathbf{J}_p(\tau)\mathbf{J}_q^T(\tau)\boldsymbol{\Gamma}_q(\tau)(\mathbf{u}_q(\tau) - \mathbf{v}_q)\,,\end{aligned} \tag{2.54}$$

for $p = s, r, b, 0$.

Now define

$$\mathbf{K}_{pq}(\tau) = \mathbf{J}_p(\tau)\mathbf{J}_q^T(\tau)\,, \quad p, q = s, r, b, 0\,. \tag{2.55}$$

Notice that these are matrices of dimensions $N_p \times N_q$, with i, j elements

$$\left(\mathbf{K}_{pq}\right)_{ij}(\tau) \;=\; \nabla_{\boldsymbol{w}} u_p^i(\tau)^T \cdot \nabla_{\boldsymbol{w}} u_q^j(\tau) \;=\; \sum_{w \in \mathbf{w}} \frac{du_p^i(\tau)}{dw} \cdot \frac{du_q^j(\tau)}{dw} \tag{2.56}$$

This allows us to collect the previous results in the following differential equation describing the evolution of the output of the self-adaptive PINN in the vanishing learning-rate limit:

$$\frac{d\mathbf{u}(\tau)}{d\tau} \;=\; -\mathbf{K}(\tau) \cdot (\mathbf{u}(\tau) - \mathbf{v}) \tag{2.57}$$

where

$$\mathbf{u}(\tau) \;=\; \begin{bmatrix} \mathbf{u}_s(\tau) \\ \mathbf{u}_r(\tau) \\ \mathbf{u}_b(\tau) \\ \mathbf{u}_0(\tau) \end{bmatrix}, \qquad \mathbf{v} \;=\; \begin{bmatrix} \mathbf{v}_s \\ \mathbf{v}_r \\ \mathbf{v}_b \\ \mathbf{v}_0 \end{bmatrix}, \tag{2.58}$$

and

$$\mathbf{K}(\tau) \;=\; \begin{bmatrix} \mathbf{K}_{ss}(\tau) & \mathbf{K}_{sr}(\tau)\mathbf{\Gamma}_r(\tau) & \mathbf{K}_{sb}(\tau)\mathbf{\Gamma}_b(\tau) & \mathbf{K}_{s0}\mathbf{\Gamma}_0(\tau) \\ \mathbf{K}_{rs}(\tau) & \mathbf{K}_{rr}(\tau)\mathbf{\Gamma}_r(\tau) & \mathbf{K}_{rb}(\tau)\mathbf{\Gamma}_b(\tau) & \mathbf{K}_{r0}\mathbf{\Gamma}_0(\tau) \\ \mathbf{K}_{bs}(\tau) & \mathbf{K}_{br}(\tau)\mathbf{\Gamma}_r(\tau) & \mathbf{K}_{bb}(\tau)\mathbf{\Gamma}_b(\tau) & \mathbf{K}_{b0}\mathbf{\Gamma}_0(\tau) \\ \mathbf{K}_{0s}(\tau) & \mathbf{K}_{0r}(\tau)\mathbf{\Gamma}_r(\tau) & \mathbf{K}_{0b}(\tau)\mathbf{\Gamma}_b(\tau) & \mathbf{K}_{00}\mathbf{\Gamma}_0(\tau) \end{bmatrix} \tag{2.59}$$

is the *empirical neural tangent kernel* matrix for the self-adaptive PINN. (When all the mask values are 1, this reduces essentially to the expression in Lemma 3.1 of [**?**].)

The square matrices $\mathbf{K}_{pp}(\tau)$ are symmetric and positive semidefinite, with nonnegative eigenvalues $\mu_p^1(\tau), \ldots, \mu_p^{N_p}(\tau)$ for $p = s, r, b, 0$. The square matrices $\mathbf{K}_{pp}(\tau)\mathbf{\Gamma}_p(\tau)$ are not symmetric, for $p = r, b, 0$, unless all values in the diagonal of $\mathbf{\Gamma}_p(\tau)$ are identical. However, under the assumption that $\mathbf{K}_{pp}(\tau)$ is positive definite, and thus invertible, the matrices $\mathbf{K}_{pp}(\tau)\mathbf{\Gamma}_p(\tau)$ are diagonalizable. To see this, note that

$$\mathbf{K}_{rr}(\tau)^{-\frac{1}{2}}\mathbf{K}_{rr}(\tau)\mathbf{\Gamma}_p(\tau)\mathbf{K}_{rr}(\tau)^{\frac{1}{2}} \;=\; \mathbf{K}_{rr}(\tau)^{\frac{1}{2}}\mathbf{\Gamma}_p(\tau)\mathbf{K}_{rr}(\tau)^{\frac{1}{2}}\,. \tag{2.60}$$

$$\mathbf{K}(\tau)^{-\frac{1}{2}}\mathbf{K}(\tau)\mathbf{\Gamma}(\tau)\mathbf{K}(\tau)^{\frac{1}{2}} \;=\; \mathbf{K}(\tau)^{\frac{1}{2}}\mathbf{\Gamma}(\tau)\mathbf{K}(\tau)^{\frac{1}{2}}\,. \tag{2.61}$$

But $\mathbf{K}_{pp}(\tau)^{\frac{1}{2}}\mathbf{\Gamma}_p(\tau)\mathbf{K}_{pp}(\tau)^{\frac{1}{2}}$ is a product of symmetric matrices, and thus symmetric itself. Hence, $\mathbf{K}_{pp}(\tau)\mathbf{\Gamma}_p(\tau)$ is similar to a real symmetric matrix, and thus diagonalizable.

$$\frac{d}{d\tau}\begin{bmatrix}\mathbf{u}_s(\tau)\\ \mathbf{u}_r(\tau)\\ \mathbf{u}_b(\tau)\\ \mathbf{u}_0(\tau)\end{bmatrix} \;=\; -\mathbf{K}\cdot\begin{bmatrix}\mathbf{u}_s(\tau)-\mathbf{v}_s\\ \mathbf{u}_r(\tau)-\mathbf{v}_r\\ \mathbf{u}_b(\tau)-\mathbf{v}_b\\ \mathbf{u}_0(\tau)-\mathbf{v}_0\end{bmatrix} \tag{2.62}$$

which has the solution

$$\begin{bmatrix}\mathbf{u}_s(\tau)\\ \mathbf{u}_r(\tau)\\ \mathbf{u}_b(\tau)\\ \mathbf{u}_0(\tau)\end{bmatrix} \;=\; (\mathbf{I}-e^{-\mathbf{K}t})\cdot\begin{bmatrix}\mathbf{v}_s\\ \mathbf{v}_r\\ \mathbf{v}_b\\ \mathbf{v}_0\end{bmatrix} \tag{2.63}$$

$$\begin{bmatrix}\mathbf{u}_s(\tau)-\mathbf{v}_s\\ \mathbf{u}_r(\tau)-\mathbf{v}_r\\ \mathbf{u}_b(\tau)-\mathbf{v}_b\\ \mathbf{u}_0(\tau)-\mathbf{v}_0\end{bmatrix} \;=\; -\mathbf{Q}^T e^{-\mathbf{M}t}\mathbf{Q}\cdot\begin{bmatrix}\mathbf{v}_s\\ \mathbf{v}_r\\ \mathbf{v}_b\\ \mathbf{v}_0\end{bmatrix} \tag{2.64}$$

that is

$$\mathbf{Q}\begin{bmatrix}\cdot\mathbf{u}_s(\tau)-\mathbf{v}_s\\ \mathbf{u}_r(\tau)-\mathbf{v}_r\\ \mathbf{u}_b(\tau)-\mathbf{v}_b\\ \mathbf{u}_0(\tau)-\mathbf{v}_0\end{bmatrix} \;=\; -e^{-\mathbf{M}t}\mathbf{Q}\cdot\begin{bmatrix}\mathbf{v}_s\\ \mathbf{v}_r\\ \mathbf{v}_b\\ \mathbf{v}_0\end{bmatrix} \tag{2.65}$$

where $\mathbf{Q}$ is the matrix of eigenvectors and $\mathbf{M}$ is a diagonal matrix containing the eigenvalues μ^i of $\mathbf{K}$.

let $\mu^1(\tau) \geq \cdots \geq \mu^n(\tau)$ and $\gamma^1(\tau) \geq \cdots \geq \gamma^n(\tau)$ be the eigenvalues of $K(\tau)$ and $K(\tau)\Gamma(\tau)$ respectively. Also let $\lambda^1(\tau) \geq \cdots \geq \lambda^n(\tau)$ be the self-adaptive weights sorted by magnitude.

Then it can be shown that

$$\gamma^{i+j-1} \leq m(\lambda_i)\mu^j \tag{2.66}$$

$$\gamma^{i+j-n} \geq m(\lambda_i)\mu^j \tag{2.67}$$

$$\tag{2.68}$$

2.6.1 SA-PINN NTK Analysis - 1D Advection PDE

To analyze the SA-PINN further, we view the underlying gradient flow via the SA-PINN's NTK, similar to the studies performed in [28]. Here we examine the application of PINNs to the the classical univariate advection of a tracer in a moving fluid, with a Riemann initial condition.

The evolution of the tracer concentration $q(x, t)$ in a pipe, for $0 \leq x \leq L$ and $t > 0$, is governed by the univariate linear advection hyperbolic PDE [51]:

$$q_t + uq_x = 0 \tag{2.69}$$

where u is the constant velocity. The Riemman initial condition is a piecewise function described by

$$q(x,0) = \begin{cases} q_l, & 0 \leq x < x_0\,, \\ q_i, & x_i < x \leq x_{i+1}\ \forall i \in 1...6 \\ q_r, & x_6 < x \leq L\,. \end{cases} \tag{2.70}$$

where x_0 = 1, x_i = [1, 0.25, 0.5, .075, 1.25, 1.5, 1.75], q_l = 4, q_i = [0.4, 4, 1.4, 3, 0.4, 4] and q_r

= 0.4. This simple problem has as solution:

$$q(x,t) = \begin{cases} q_l, & 0 \leq x < x_0 + ut\,, \\ q_i, & x_i + ut < x \leq x_{i+1} + ut \ \forall i \in 1...6 \\ q_r, & x_6 + ut < x \leq L\,. \end{cases} \tag{2.71}$$

for $0 \leq t < (L - x_0)/u$. In other words, the initial discontinuity in concentration is simply advected to the left with speed u. In this case, we provide a piecewise initial condition that is more complex than the typical hyperbolic "single shock" typically modeled using the advection PDE system. This system is quite difficult to solve numerically, and discretized methods have been proposed to do so [51]. We see in figures 2.10 and 2.11 that the baseline PINN also struggles to capture the nonlinear high-frequency shock at the boundaries in the system. The SA-PINN, however, is capable of capturing the dynamics significantly more effectively, and training in a matter of seconds on GPU. A cross-section of the solution can be seen in figure 2.11 and the spatio-temporal solution can be seen in figure 2.10.

The results presented in figures 2.10, 2.11, and 2.12 are generated with a neural network architecture of [2, 400, 400, 400, 400, 1], trained for 10k Adam iterations with a neural network weight learning rate of 0.001 and a self-adaptive weight learning rate of 0.1. Glorot Normal initialization was utilized, and all training was completed in Tensorflow on a single V100 GPU with an average training time of 7 seconds for 10k iterations. At the end of 10k training iterations, the baseline PINN failed to grasp even the high level structure of the solution, but the SA-PINN was able to approximate the solution with 5% L2 error. This error can be further reduced by training with a learning rate schedule and for more training epochs. However, for NTK analysis performed in this section, a simple training scheme was utilized with no learning rate schedule.

An analysis of NTK eigenvalues similar to that performed in [28] is demonstrated in figure 2.12. We can see that the eigenvalues of the NTK are increased greatly in magnitude, facilitating training, but are also more closely matched in scale between K_{uu} and K_{rr}. This implies that training

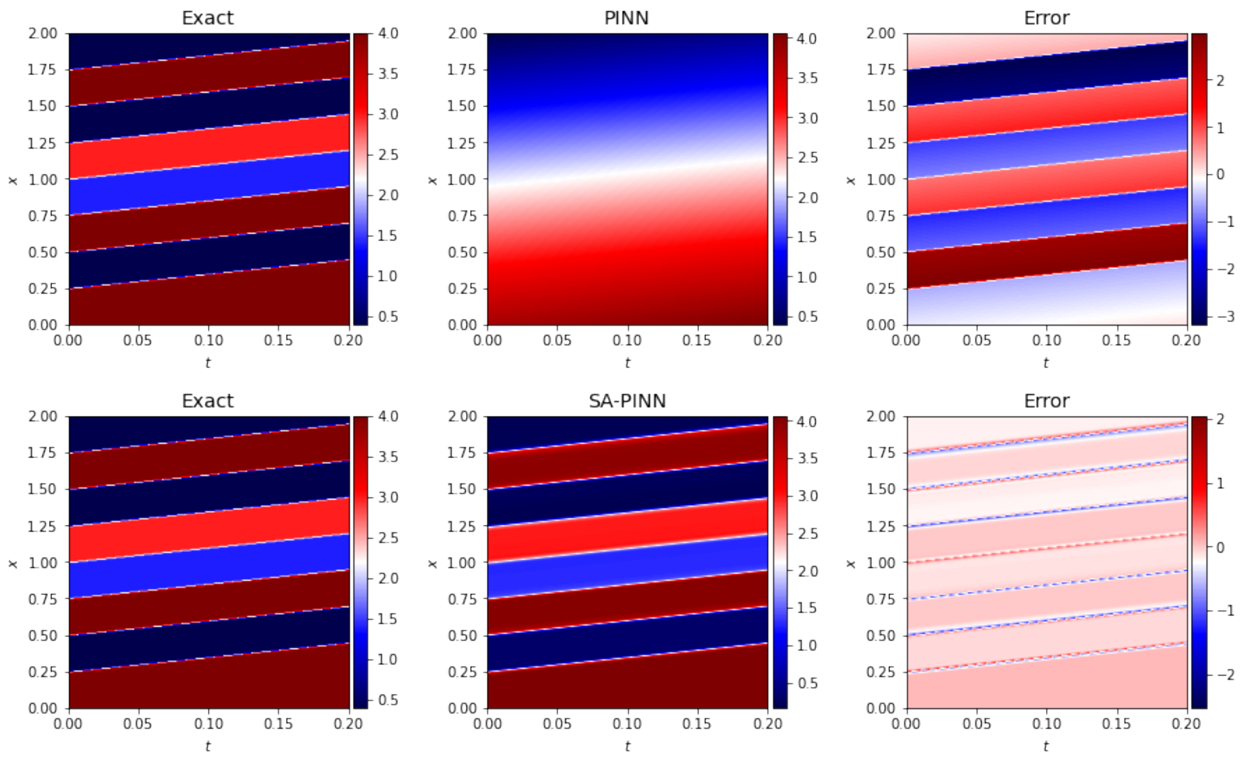


Figure 2.10: *Top:* Plot of the approximation $u(x, t)$ via the baseline PINN, showing the exact solution vs predicted solution vs absolute error. *Bottom:* The SA-PINN results, L2 error decreases by an order of magnitude and the SA-PINN closely captures the exact solution.

steps taken while training the network weights are more accurately describing the direction and magnitude of training the *whole* loss function, instead of over or under-weighted parts of it. Many times in PINN training, large and unbalanced gradient magnitudes will dominate training and lead to an approximation that fails to converge to the true solution.

2.7 Additional Examples with SA-PINNs

Here we present additional experimental results with Burger's and Helmholtz PDEs, which confirmed the trends observed previously.

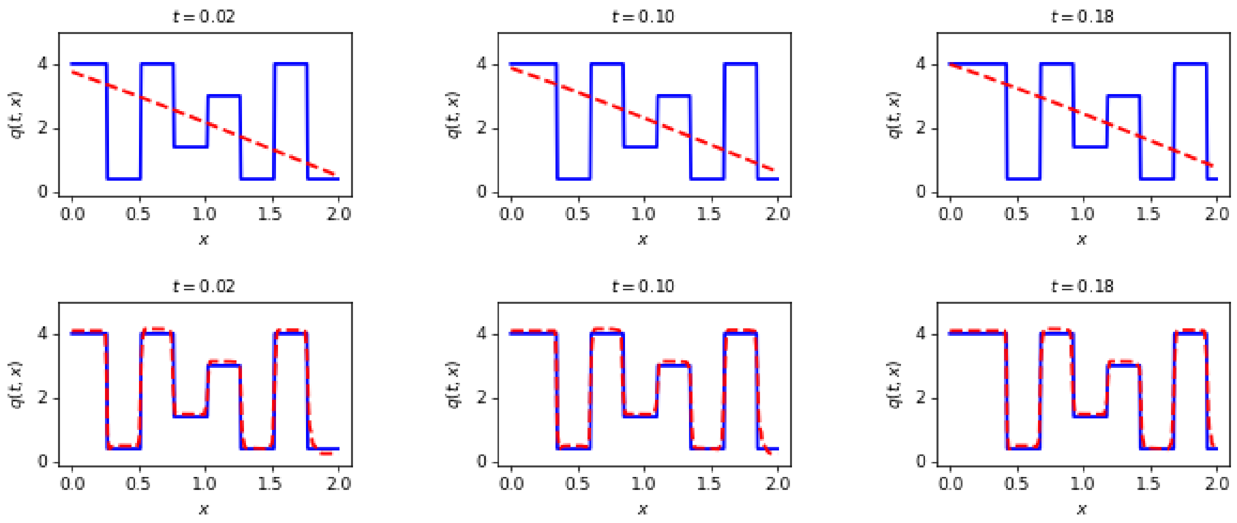


Figure 2.11: *Top:* Plot of the approximation $u(x,t)$ via the baseline PINN, showing cross sections of the spatial domain at $t = 0.02, 0.10, 0.18$ *Bottom:* The SA-PINN results at the same time steps, with the same number of epochs (10k Adam) and all other parameters held constant.

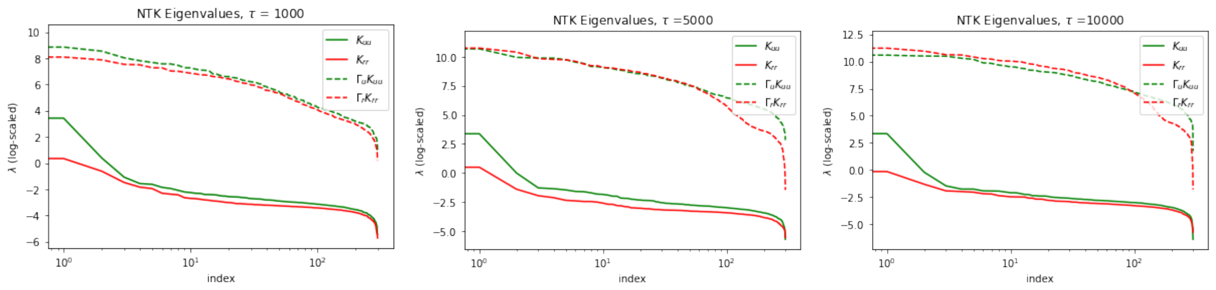


Figure 2.12: NTK eigenvalues of the baseline PINN (solid) vs. the SA-PINN (dashed) for τ =1000, 5000, and 10000 training iterations. It can be observed that the SA-PINN accurately matches the magnitudes of the NTK eigenvalues between terms of the loss function, in this case the initial condition K_{uu} and the residual loss K_{rr}

2.7.1 Burgers' Equation

The viscous Burgers' PDE considered here is

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0\,, \;\; x \in [-1, 1],\; t \in [0, 1]\,, \tag{2.72}$$

$$u(0, x) = -sin(\pi x)\,, \tag{2.73}$$

$$u(t, -1) = u(t, 1) = 0\,. \tag{2.74}$$

All results for Burgers equation were generated from a fully-connected network with input later size 2 corresponding to inputs of (x, t), 8 hidden layers of 20 neurons each, and an output layer of size 1 corresponding to the output of the approximation $u(x, t)$. This directly mimics the setup of the results presented in [25]. All training is done for 10k iterations of Adam, followed by 10k iterations of L-BFGS to fine tune the network weights, consistent with related work. Additionally, the number of points selected for the trials shown are N_0 = 100, N_b = 200, and N_r = 10000. Training with this architecture took 96ms/iteration on a single Nvidia V100 GPU.

We achieved an L2 error of 4.803e-04 $\pm$ 1.01e-4 over 10 random restarts, which is a smaller error than the errors reported in [25] in 1/5 of the number of training iterations for an identical fully-connected architecture. The high-fidelity and predicted solutions are displayed in figure 2.13. Figure 2.14 demonstrate the accuracy of the proposed approach, using a significantly shorter training horizon than the baseline PINN.

Figure 2.15 shows that the sharp discontinuity at $x = 0$ in the solution has correspondingly large weights, indicating that the model must pay extra attention to those particular points in its solution, resulting in an increase in approximation accuracy and training efficiency.

2.7.2 Helmholtz Equation

The Helmholtz equation model is typically used to describe the behavior of wave and diffusion processes, and can be employed to model evolution in a spatial domain or combined spatial-temporal domain. Here we study a particular Helmholtz PDE existing only in the spatial (x, y)

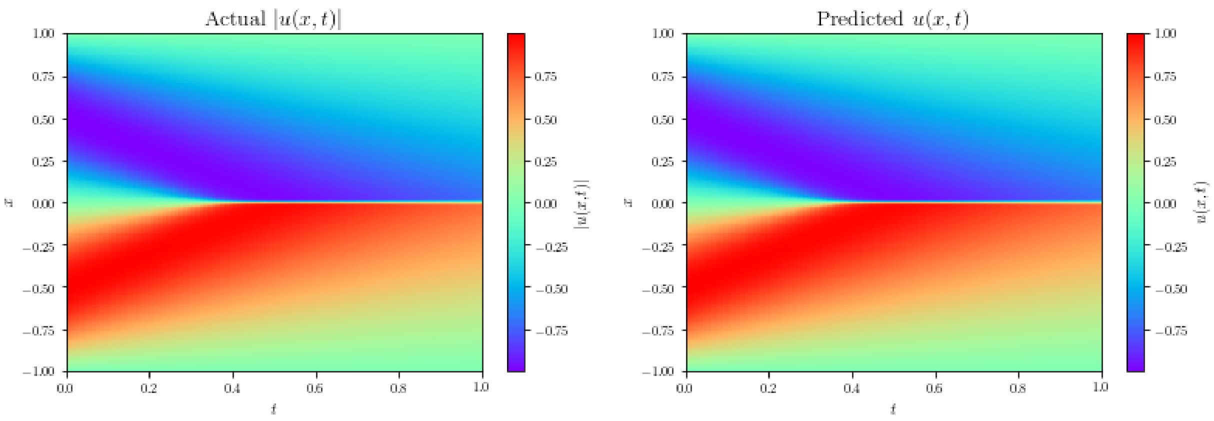


Figure 2.13: High-fidelity (*left*) vs. predicted (*right*) solutions for Burgers' equation

domain, described as:

$$u_{xx} + u_{yy} + k^2 u - q(x,y) = 0 \tag{2.75}$$

$$u(-1,y) = u(1,y) = u(x,-1) = u(x,1) = 0 \tag{2.76}$$

where $x \in [-1,1], y \in [-1,1]$ and

$$\begin{aligned} q(x,y) = &- (a_1\pi)^2 \sin(a_1\pi x)\sin(a_2\pi y) \\ &- (a_2\pi)^2 \sin(a_1\pi x)\sin(a_2\pi y) \\ &+ k^2 \sin(a_1\pi x)\sin(a_2\pi y) w \end{aligned} \tag{2.77}$$

is a forcing term that results in a closed-form analytical solution

$$u(x,y) = \sin(a_1\pi x)\sin(a_2\pi y)\,. \tag{2.78}$$

To allow a direct comparison to the results reported in [35], we take a_1 = 1 and a_2 = 4 and use the same neural network architecture with layer sizes [2, 50, 50, 50, 50, 1]. Our architecture is trained for 10k Adam and 10k L-BFGS iterations, again keeping the self-adaptive mask weights constant

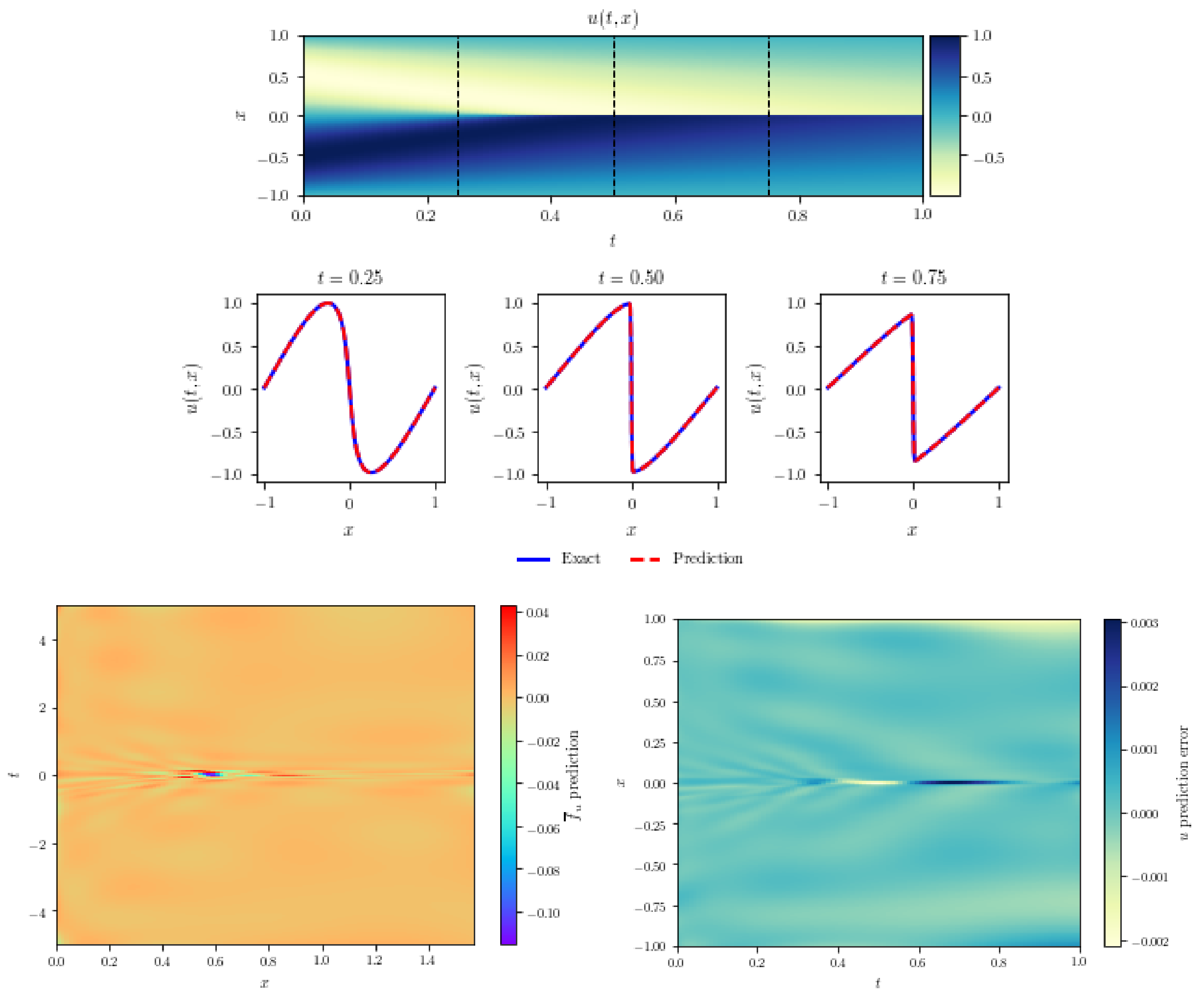


Figure 2.14: *Top:* predicted solution of Burger's equation. *Middle:* Cross-sections of the approximated vs. actual solutions for various x-domain snapshots. *Bottom left:* Residual $r(x,t)$ across the spatial-temporal domain. *Bottom right:* Absolute error between prediction and high-fidelity solution across the spatial-temporal domain.

through the L-BFGS training iterations and only allowing those to train via Adam. We sample N_b = 400 (100 points per boundary). Given the steady-state initialization and constant forcing term, there is no applicable initial condition and consequently no N_0. We create a mesh of size (1001,1001) corresponding to the $x \in [-1,1], y \in [-1,1]$ range, yielding 1,002,001 total mesh points, from which we select N_r=100k residual collocation points.

We can see in figure 2.16 that the Self-Adaptive PINN prediction is very accurate and indistinguishable from the exact solution. We achieve a relative L2 error of 3.2e-3 $\pm$ 2.2e-4, which

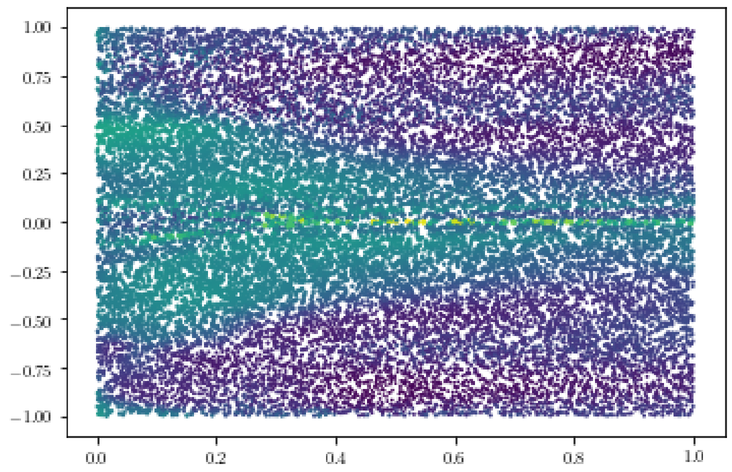


Figure 2.15: Trained weights for collocation points across the domain Ω. Larger/brighter colored points correspond to larger weights.

improves upon the learning-rate annealing weighted scheme proposed in [35], and begins to encroach on the accuracy of their improved fully-connected scheme with no additional modifications to the network structure itself. It is also worth noting that the Self-Adaptive PINN is trained for 1/2 of the training iterations listed in [35] as well (at 10k Adam and 10 L-BFGS vs. 40k Adam), and achieves better L2 accuracy than a comparable architecture listed in table 2 of [35].

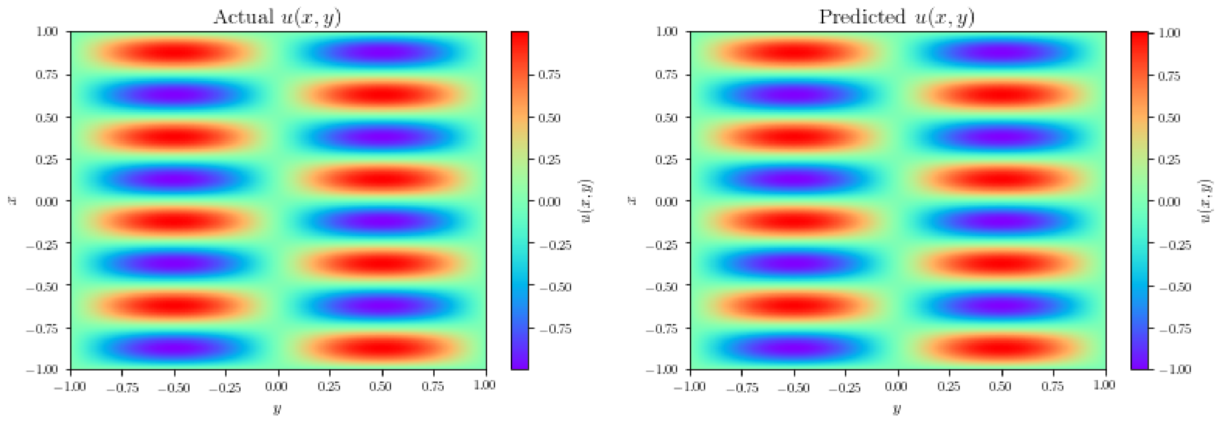
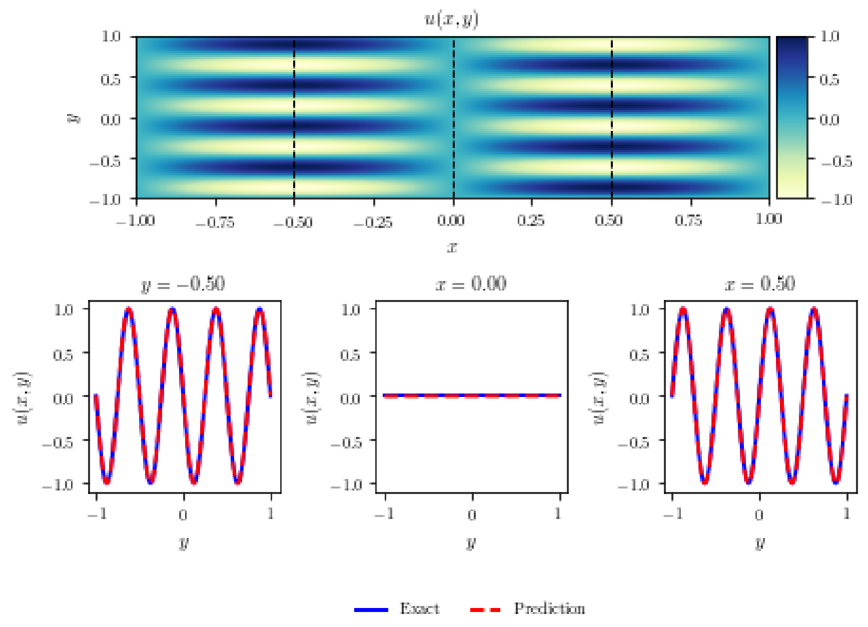


Figure 2.16: Exact (*left*) vs. predicted (*right*) solutions for Helmholtz equation



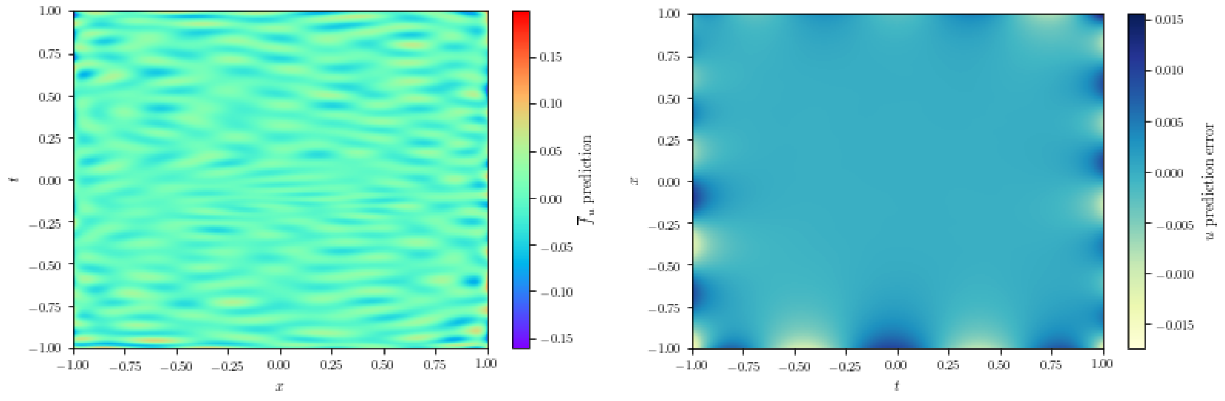


Figure 2.17: *Top* predicted solution of Helmholtz equation. *Bottom* Cross-sections of the approximated vs. actual solutions for various x-domain snapshots

Figure 2.17 shows individual cross-sections of the Helmholtz solution, demonstrating the Self-Adaptive PINN's ability to accurately approximate the sinusoidal solution on the whole domain. Figure 2.18 shows that the Self-Adaptive PINN largely ignores the flat areas in the solution, while focusing its attention on the nonflat areas.

2.8 Conclusion

In this paper, we introduced Self-Adaptive Physics-Informed Neural Networks, a novel class of physics-constrained neural networks. This approach uses a similar conceptual framework as soft

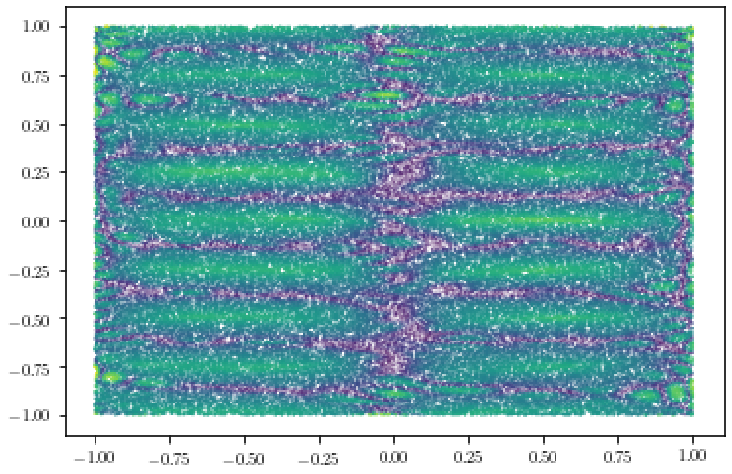


Figure 2.18: Self-learned weights after training via Adam for the Helmholtz system. Brighter/larger points correspond to larger weights.

self-attention mechanisms used in Computer Vision, in that the network identifies which inputs are most important to its own training. It was shown that training of the Self-Adaptive PINN is formally equivalent to solving a PDE-constrained optimization problem using penalty-based method, though in a way where the monotonically-nondecreasing penalty coefficients are trainable. Experimental results with Burgers', Helmholtz, and Allen-Cahn PDEs indicate that Self-Adaptive PINNs allow generate more accurate solutions of PDEs with smaller computational cost than other state-of-the-art PINN algorithms.

We believe that self-adaptive PINNs open up new possibilities for the use of deep neural networks in forward and inverse modeling in engineering and science. However, there is much that is not known yet about this class of algorithms, and indeed PINNs in general. For example, the use of standard off-the-shelf optimization algorithms for training deep neural networks, such as Adam, may not be appropriate, since those algorithms were mostly developed for image classification problems. How to obtain optimization algorithms specifically tailored to PINN problems in an open problem. In addition, the relationship between PINNs and constrained-optimization problems, hinted at here, is likely a profound and fruitful topic of future study.

3. *TensorDiffEq*: Scalable Multi-GPU Forward and Inverse Solvers for Physics Informed Neural Networks

3.1 Introduction

As part of the burgeoning field of scientific machine learning [24], physics-informed neural networks (PINNs) have emerged recently as an alternative to traditional partial different equation (PDE) solvers [25, 26, 27, 28], and have given rise to the larger field of study in neural network approximation of PDE systems, generally referred to as Neural PDEs. Typical black-box deep learning methodologies do not take into account the underlying physics of the problem domain. The Neural PDE approach is based on constraining the output of a deep neural network to satisfy a physical model specified by a PDE. PINNs typically perform this task via PDE-constrained regularization of a residual function defined by the approximation of the solution network `u` and forward-pass calculations through the physics of the PDE model, with the applicable derivatives of `u` calculated via reverse-mode automatic differentiation in a modern deep learning framework such as Tensorflow [30].

The potential of using neural networks as universal function approximators to solve PDEs had been recognized since the 1990's [29, 52]. However, Physics-Informed Neural Networks promise to take this approach to a different level through deep neural networks, the exploration of which is now possible due to the vast advances in computational capabilities and training algorithms since that time [30, 31] and modern congenial automatic differentiation software [32, 33].

A great advantage of the PINN architecture over traditional time-stepping PDE solvers is that the entire spatial-temporal domain can be solved at once using collocation points distributed quasi-randomly (rather than on a grid) across the spatial-temporal domain, in a process that can be massively parallelized via GPU. As we have continued to see GPU capabilities increase in recent years, a method that relies on parallelism in training iterations could begin to emerge as the predominant approach in scientific computing. To this end, while other software suites exist to define and solve

PINNs [53, 54, 55, 56], many of those platforms are either restricted to single-GPU implementation or are not fully open-source. Additionally, with full support and customization capabilities of the Keras neural network ecosystem built in to the package, researchers and practitioners can define and train their own custom neural network architectures to approximate the solution of their problem domains. *TensorDiffEq* provides these scalable, modular, and customizable multi-GPU architectures and solvers in a fully open-source platform, tapping into the collective intelligence of the field to improve the implementation of the software and provide input on the direction, structure, and feature coverage of the framework.

3.2 Mathematical Underpinnings of PINNs

Consider a general nonlinear PDE of the form:

$$\mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t)] = 0\,,\ \ \boldsymbol{x} \in \Omega\,,\ t \in [0,T]\,, \tag{3.1}$$

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t)\,,\ \ \boldsymbol{x} \in \partial\Omega,\ t \in [0,T]\,, \tag{3.2}$$

$$u(\boldsymbol{x},0) = h(\boldsymbol{x})\,,\ \ \boldsymbol{x} \in \Omega\,, \tag{3.3}$$

where $\boldsymbol{x} \in \Omega$ is a spatial vector variable in a domain $\Omega \subset R^d$, t is time, and $\mathcal{N}_{\boldsymbol{x},t}$ is a spatial-temporal differential operator. Following [25], let $u(\boldsymbol{x},t)$ be approximated by the output $u(\boldsymbol{x},t;\boldsymbol{w})$ of a deep neural network with inputs $\boldsymbol{x}$ and t. Define the residual network $r(\boldsymbol{x},t;\boldsymbol{w})$, which share the same network weights $\boldsymbol{w}$ as the approximation network $u(\boldsymbol{x},t;\boldsymbol{w})$, and satisfies:

$$r(\boldsymbol{x},t;\boldsymbol{w}) := \mathcal{N}_{\boldsymbol{x},t}[u(\boldsymbol{x},t;\boldsymbol{w})]\,, \tag{3.4}$$

where all partial derivatives can be computed by automatic differentiation methods [32, 33]. The shared network weights $\boldsymbol{w}$ are trained by minimizing a loss function that penalizes the output for not satisfying (1)-(3):

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_s(\boldsymbol{w}) + \mathcal{L}_r(\boldsymbol{w}) + \mathcal{L}_b(\boldsymbol{w}) + \mathcal{L}_0(\boldsymbol{w})\,, \tag{3.5}$$

where $\mathcal{L}_s$ is the loss corresponding to sample data (if any), $\mathcal{L}_r$ is the loss corresponding to the residual (3.4), $\mathcal{L}_b$ is the loss due to the boundary conditions (3.2), and $\mathcal{L}_0$ is the loss due to the initial conditions (3.3):

$$\mathcal{L}_s(\boldsymbol{w}) = \frac{1}{N_s}\sum_{i=1}^{N_s} |u(\boldsymbol{x}_s^i, t_s^i; \boldsymbol{w}) - y_s^i|^2, \tag{3.6}$$

$$\mathcal{L}_r(\boldsymbol{w}) = \frac{1}{N_r}\sum_{i=1}^{N_r} r(\boldsymbol{x}_r^i, t_r^i; \boldsymbol{w})^2, \tag{3.7}$$

$$\mathcal{L}_b(\boldsymbol{w}) = \frac{1}{N_b}\sum_{i=1}^{N_b} |u(\boldsymbol{x}_b^i, t_b^i; \boldsymbol{w}) - g_b^i|^2, \tag{3.8}$$

$$\mathcal{L}_0(\boldsymbol{w}) = \frac{1}{N_0}\sum_{i=1}^{N_0} |u(\boldsymbol{x}_0^i, 0; \boldsymbol{w}) - h_0^i|^2. \tag{3.9}$$

where $\{\boldsymbol{x}_s^i, t_s^i, y_s^i\}_{i=1}^{N_s}$ are sample data (if any), $\{\boldsymbol{x}_0^i, h_0^i = h(\boldsymbol{x}_0^i)\}_{i=1}^{N_0}$ are initial condition point, $\{\boldsymbol{x}_b^i, t_b^i, g_b^i = g(\boldsymbol{x}_b^i, t_b^i))\}_{i=1}^{N_b}$ are boundary condition points, $\{\boldsymbol{x}_r^i, t_r^i\}_{i=1}^{N_r}$ are collocation points randomly distributed in the domain Ω, and N_s, N_0, N_b and N_r denote the total number of sample data, initial points, boundary points, and collocation points, respectively. The network weights $\boldsymbol{w}$ can be tuned by minimizing the total training loss $\mathcal{L}(\boldsymbol{w})$ via standard gradient descent procedures used in deep learning.

3.3 Using *TensorDiffEq* for Forward Problems

TensorDiffEq has a boilerplate model that can loosely be followed in most instances of usage of the package. In forward problems, this process is generally described in the following order:

1. Define the problem domain
2. Describe the physics of the model
3. Define the Initial Conditions and Boundary Conditions (IC/BCs)
4. Define the neural network architecture
5. Select and define the solver

6. Solve the PDE using the `fit` method

Each of these steps has multiple options and definitions in the *TensorDiffEq* solution suite. The following sections will provide a brief overview of some of the built-in functionality of the package.

3.3.1 Define the Problem Domain

A `Domain` object is the first essential component of defining a problem in *TensorDiffEq*. The domain object contains primitives for defining the problem scope used later in your definitions of boundary conditions, initial conditions, and eventually to sample collocation points that are fed into the PINN solver.

The `Domain` object is defined iteratively. As many dimensions as are required can simply be added to the domain using the `add` method. This means *TensorDiffEq* can be used to solve spatial (steady-state) or spatiotemporal 2D, 3D, or ND problems.

3.3.2 Describe the Physics of the Model

Since *TensorDiffEq* is built on top of Tensorflow [57] physics of the model can be defined via a strong-form PDE, with gradients defined using the built-in `tf.gradients` function. This allows for a definition similar to that seen in [25]. An example of defining the PDE for a viscous Burger's system is shown below:

```
def f_model(u_model, x, t):
    u = u_model(tf.concat([x, t], 1))
    u_x = tf.gradients(u, x)
    u_xx = tf.gradients(u_x, x)
    u_t = tf.gradients(u, t)
    f_u = u_t + u * u_x - (0.01 / tf.constant(math.pi)) * u_xx
    return f_u
```

Due to the nature of how the PDE system is defined in *TensorDiffEq*, one could define a separate system of `u` and allow for a coupled PDE definition using a similar style as the one shown above.

3.3.3 Define the ICs/BCs

TensorDiffEq supports various types of ICs and BCs and the list will continue to grow. The ICs and BCs that require functions allow for intuitive definitions of those functions of system variables as a Python `function`, which allows for nonlinear and non-continuous function definitions of state variables. One could define piece-wise functions, Boolean functions, etc using this verbiage and it would be valid input to *TensorDiffEq*'s solvers. At the time of this writing, *TensorDiffEq* supports constant Dirichlet, Function Dirichlet, and periodic BCs, as well as function-based ICs. *TensorDiffEq* takes the ICs and BCs as a list, therefore one can add as many as necessary to define the system. If a BC is not defined on a particular boundary or it is overlooked in the problem definition then the solver will attempt to approximate that boundary using PDE-constrained regularization of the inner points on or around that boundary.

3.3.4 Define the Neural Network Architecture

The default architecture of the neural network is a fully connected MLP defined in the Keras API [19]. To take advantage of the built-in MLP, a list of hidden layer sizes is passed into the solver. However, this baseline architecture can be overwritten by any Keras neural network. Currently, the solver requires the number of inputs of the neural network to be the same as the number of dimensions of the system, and the output is the scalar value of the approximation of $u(\mathbf{X})$ at that combination of input points. However, this “single-network” output architecture is actively being expanded at the time of this writing.

In the event one desires to add batch norm, residual blocks, etc, then the Keras API could be used to define the model and the internal parameters of *TensorDiffEq's* solvers could be modified to use that network as the solution network for $u(\mathbf{X})$. In this way, so long as the input to the neural network has the correct dimensionality for the system (i.e. 3 nodes for a problem with `x, y, t` dimensions) and the output node is the correct number of dimensions then one could build any architecture the Keras API allows and pass it into the solver. This features also allows for custom neural network layer support using the Keras `lambda` layer ecosystem, allowing for complete autonomy in the definition of the neural network model internals and training via built-in Keras optimizers.

3.3.5 Select and Define The Solver

TensorDiffEq is a suite designed to provide forward and inverse PINN solvers. As such, there are various solvers to perform these tasks. At the time of this writing, there are *Collocation Method* solvers for forward modeling and the *Discovery Model* for inverse modeling.

Hyperparameter selection can be modified by the user by overwriting the default Adam optimizer [20] with any of the other available optimizers in Keras, to include AdaDelta [58], Root-Mean-Square Propagation, SGD, and others. Some of these different optimization techniques prove more stable in training than others, and there exist various methods of modifying the loss function of the collocation solver to improve convergence [28, 35]. To this end, *TensorDiffEq* supports self-adaptive training methods, which have proven to be effective in helping semi-linear PDE systems, such as Allen-Cahn [59], converge where the baseline collocation method fails [60]. Other methods of improving convergence in Neural PDE and PINN training are continuously being considered.

3.3.6 Solve the PDE

Each solver has a `compile` and `fit` method, to give the package a feel similar to modern popular machine learning or deep learning frameworks such as Keras [19] or scikit-learn [61]. In most instances, the `compile` function places parameters such as the domain size and shape, neural network sizes, BCs/ICs, etc. into the solver, and the `fit` function takes only the number of iterations of Keras optimizer runs or newton solver runs.

A feature unique to *TensorDiffEq* is that the Keras neural network model can be exported and saved for later use. This could allow for training on a data center platform, but inference on a local machine. Additionally, being able to export the Keras neural network model opens the door to transfer learning possibilities previously difficult with the versions of Neural PDE solvers currently in circulation. In the case of *TensorDiffEq*, this is a natural result of leaning on the

Tensorflow/Keras APIs.

3.4 Solving Inverse Problems

TensorDiffEq comes with a base class solver for inverse problems. Inverse problems can imply parameter estimation or even estimate the interactions between nonlinear operators [62] from data. *TensorDiffEq* contains solvers that perform parameter estimation in a PDE system. These parameters can be mobility parameters, diffusivity parameters, etc, where there is some level of a priori physical knowledge about the system in question, but a specific parameter may be unknown. *TensorDiffEq* contains built-in support for solving of such systems that can be solved in ND cases. Parameters are defined as variables that are learned over the course of the training, therefore a natural output is a trained $u(\mathbf{X}, t)$ solution as well as the estimate of the parameters in question.

3.5 Conclusion

In this article, the authors introduce *TensorDiffEq*, a scalable multi-GPU solver for PINNs/NeuralPDEs. Some of the main highlights of the software are covered, and more features are currently underway. *TensorDiffEq* contains support for various types of initial conditions, boundary conditions, and allows the user to custom-define their PDE system for their specific problem. In the event that inverse modeling is required, *TensorDiffEq* contains solvers that will accommodate parameter estimation of a PDE system. Currently, *TensorDiffEq* is the only software suite to support self-adaptive solving, demonstrated to improve training convergence and accuracy of the final solution. *TensorDiffEq* takes a step forward in modern implementations of PINN solvers, and fills a unique niche of being a fully open-source multi-GPU PINN solver in the current ecosystem of Scientific Machine Learning software offerings.

3.6 Example - Solving a Nonlinear PDE in TensorDiffEq

Here we demonstrate the basic usage of the package. More examples are available at `github.com/tensordiffeq/` and the most up-to-date documentation is available at `docs.tensordiffeq.io`

The viscous Burgers' PDE considered here is

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0\,, \quad x \in [-1,1],\ t \in [0,1]\,, \tag{3.10}$$

$$u(0,x) = -sin(\pi x)\,, \tag{3.11}$$

$$u(t,-1) = u(t,1) = 0\,. \tag{3.12}$$

and is a fundamental example in PINN literature [25]. To solve this PDE system in *TensorDiffEq*, the following code could be used:

```python
import math
import tensordiffeq as tdq
from tensordiffeq.boundaries import *
from tensordiffeq.models import CollocationSolverND

# Define the problem domain
Domain = DomainND(["x", "t"], time_var='t')
Domain.add("x", [-1.0, 1.0], 256)
Domain.add("t", [0.0, 1.0], 100)

N_f = 10000
Domain.generate_collocation_points(N_f)

# Define the Initial Condition (IC)
def func_ic(x):
    return -np.sin(x * math.pi)

# Define the IC/BCs
init = IC(Domain, [func_ic], var=[['x']])
upper_x = dirichletBC(Domain, val=0.0, var='x', target="upper")
lower_x = dirichletBC(Domain, val=0.0, var='x', target="lower")

BCs = [init, upper_x, lower_x]
```

```python

# Define the physics in tf.gradients syntax
def f_model(u_model, x, t):
    u = u_model(tf.concat([x, t], 1))
    u_x = tf.gradients(u, x)
    u_xx = tf.gradients(u_x, x)
    u_t = tf.gradients(u, t)
    f_u = u_t + u * u_x - (0.01 / tf.constant(math.pi)) * u_xx
    return f_u

# List of layer sizes for the FC network
layer_sizes = [2, 20, 20, 20, 20, 20, 20, 20, 20, 1]

# Define and compile the model
model = CollocationSolverND()
model.compile(layer_sizes, f_model, Domain, BCs)

# to reproduce results from Raissi and the SA-PINNs paper, train for 10k
    newton and 10k adam
model.fit(tf_iter=10000, newton_iter=10000)
```

This trains a neural network in the `model` class that can later be called with a `predict` method to allow for visualization of the solution, similar to that show in fig 3.1,

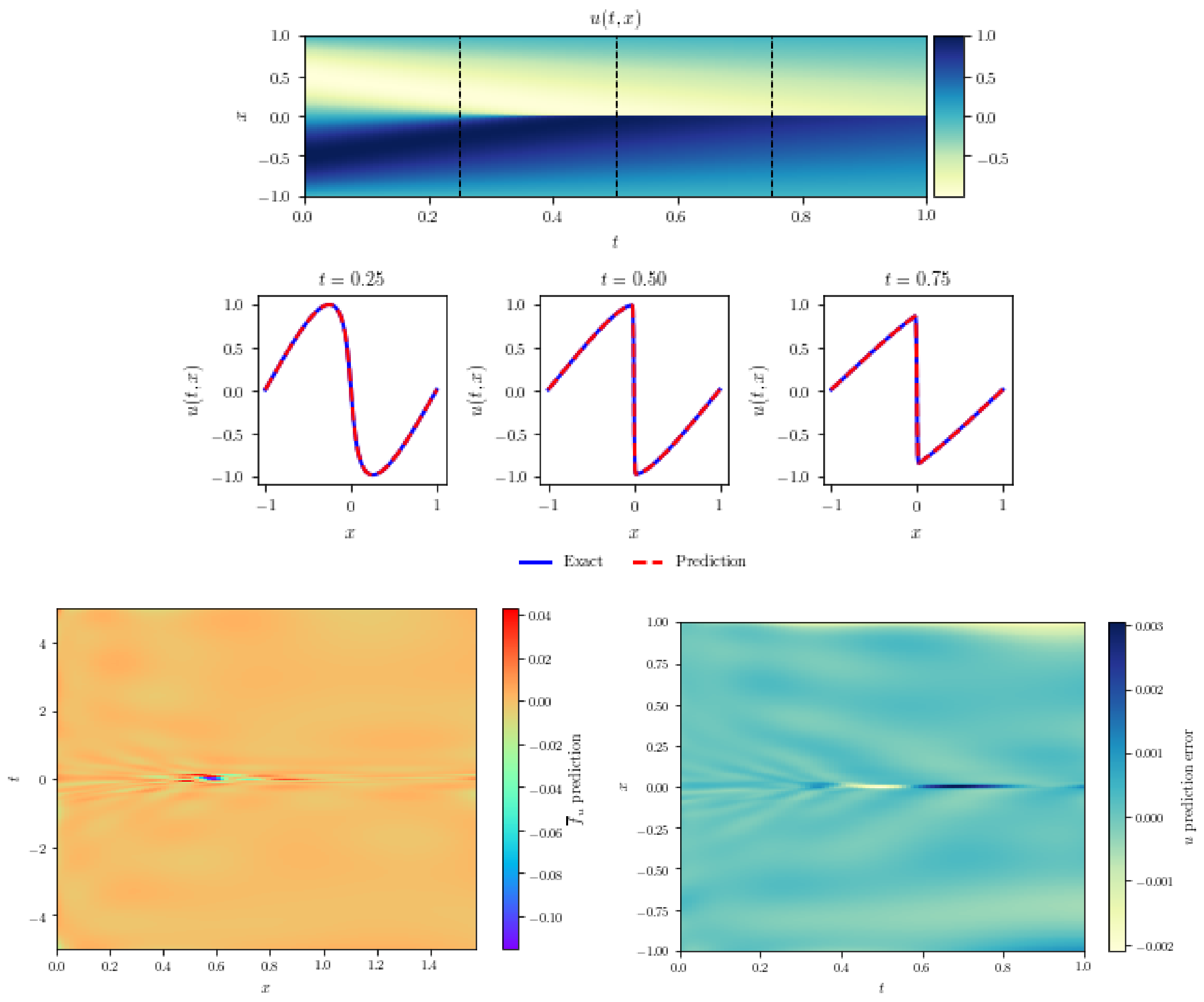


Figure 3.1: *Top:* predicted solution of Burger's equation. *Middle:* Cross-sections of the approximated vs. actual solutions for various x-domain snapshots. *Bottom left:* Residual $r(x, t)$ across the spatial-temporal domain. *Bottom right:* Absolute error between prediction and high-fidelity solution across the spatial-temporal domain.

3.7 Example - Solving the Semi-linear Allen-Cahn PDE System in TensorDiffEq

Another well-explored example is the Allen-Cahn PDE system for reaction-diffusion, describing the process of phase separation in multi-component alloy systems.

An example of the system is shown below:

$$u_t - 0.0001u_{xx} + 5u^3 - 5u = 0\,,\;\; x \in [-1,1],\; t \in [0,1]\,, \tag{3.13}$$

$$u(x,0) = x^2 cos(\pi x)\,, \tag{3.14}$$

$$u(t,-1) = u(t,1)\,, \tag{3.15}$$

$$u_x(t,-1) = u_x(t,1)\,. \tag{3.16}$$

In order to implement this system, and solve it effectively, we must employ self-adaptive solving [60]. Otherwise, the baseline PINN will fail to train or accurately capture the system evolution dynamics in the solution approximation. Additionally, the below example highlights the ability of *TensorDiffEq* to implement custom-defined Keras neural networks and include batch norm layers, as an example.

```
import math
import tensordiffeq as tdq
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization
from tensordiffeq.models import CollocationSolverND
from tensordiffeq.boundaries import *

# Define the problem domain
Domain = DomainND(["x", "t"], time_var='t')
Domain.add("x", [-1.0, 1.0], 512)
Domain.add("t", [0.0, 1.0], 201)

N_f = 10000
Domain.generate_collocation_points(N_f)

# Define the Initial Conditions (ICs)
def func_ic(x):
    return x ** 2 * np.cos(math.pi * x)
```

```python


# Conditions to be considered at the boundaries for the periodic BC
def deriv_model(u_model, x, t):
    u = u_model(tf.concat([x, t], 1))
    u_x = tf.gradients(u, x)[0]
    return u, u_x


init = IC(Domain, [func_ic], var=[['x']])
x_periodic = periodicBC(Domain, ['x'], [deriv_model])

BCs = [init, x_periodic]


def f_model(u_model, x, t):
    u = u_model(tf.concat([x, t], 1))
    u_x = tf.gradients(u, x)
    u_xx = tf.gradients(u_x, x)
    u_t = tf.gradients(u, t)
    c1 = tdq.utils.constant(.1)
    c2 = tdq.utils.constant(5.0)
    f_u = u_t - c1 * u_xx + c2 * u * u * u - c2 * u
    return f_u

# Define initial and residual collocation weight vectors
col_weights = tf.Variable(tf.random.uniform([N_f, 1]), trainable=True, dtype=
    tf.float32)
u_weights = tf.Variable(100 * tf.random.uniform([512, 1]), trainable=True,
    dtype=tf.float32)

# still required to define an FC network, will be overwritten later
layer_sizes = [2, 128, 128, 128, 128, 1]
```

```

# Define a custom naural network architecture in Keras for use in tdq’s solver
model_bn = Sequential()
model_bn.add(Dense(128, input_dim=2, activation=tf.nn.tanh,
    kernel_initializer="glorot_normal"))
model_bn.add(BatchNormalization())
model_bn.add(Dense(128, activation=tf.nn.tanh,  kernel_initializer="
    glorot_normal"))
model_bn.add(BatchNormalization())
model_bn.add(Dense(128, activation=tf.nn.tanh,  kernel_initializer="
    glorot_normal"))
model_bn.add(BatchNormalization())
model_bn.add(Dense(128, activation=tf.nn.tanh,  kernel_initializer="
    glorot_normal"))
model_bn.add(Dense(1, activation=tf.nn.tanh,  kernel_initializer="
    glorot_normal"))


# Define the model and compile with the self-adaptive weights
model = CollocationSolverND()
model.compile(layer_sizes, f_model, Domain, BCs, isAdaptive=True, col_weights=
    col_weights, u_weights=u_weights)

# replace the default FC network in the model with our new one, to include
    batch norm layers
model.u_model = model_bn

model.fit(tf_iter=1000)
```

4. PINN Expansion and Adoption

This section contains up-and-coming work related to utilization of PINNs in various collaborative areas of science and engineering. The applications of PINNs is abound, and the schema can be utilized in almost any instance where a system of governing PDEs controls a system. In some instances, the systems can be difficult to train and require modifications to the baseline PINN algorithm to successfully perform forward or inverse modeling. This section highlights some of those instances wherein the self-adaptive PINN or the or some other derivative PINN has been successful in modelling of problems where baseline PINNs have failed, or where PINNs are emerging as a potential successful training methodology.

4.1 Solving Hyperbolic PDEs with PINNs

Recently, work has been completed in analysis of PINNs for hyperbolic PDEs. Studies have been conducted to employ PINNs to solve a classic problem in petroleum reservoir engineering referred to the Buckley-Leverett equation [63, 64, 65]. It has been demonstrated that PINNs fail to find the solution of the PDE when it has hyperbolic behavior with shocks and contact discontinuities in the solution. Adding artificial viscosity to reduce the hyperbolicity of the PDE is a well-known approach in traditional scientific computation [66]. In the context of PINNs, [67] proposed using Welge's method [68] to handle the shock front in the Buckley-Leverett problem. Welge's method transforms the fractional flow function to assure that the entropy condition is satisfied but notably this method is only valid with homogeneous initial conditions.

In this series of work, performed in parallel with Texas A&M Dept. of Petroleum Engineering, various methods of PINN-native solvers are proposed to handle the hyperbolic nature of the Buckley-Leverett PDE system. 3 major modifications to the PINN methodology incorporating characteristics specific to Buckley-Leverett are analyzed in this work:

1. Learnable Artificial Viscosity

2. Parameterized Artificial Viscosity Map

3. Residual-based Viscosity Map

The bulk of experimentation in this section is credited to our collaborators in Texas A&M PETE[1] and was supported by work performed using the SA-PINN.

4.1.1 PINNs with a Learnable Artificial Viscosity

To address the issue of overparameterization of the model, or arbitrary selection of a viscosity parameter, a learnable artificial viscosity parameter is introduced that is trainable against the overall loss function of the PINN. This requires adjustment of the residual to include an adaptive viscosity term, i.e.

$$r(\boldsymbol{x}, t, \boldsymbol{w}, \nu) = \frac{\partial u(\boldsymbol{x}, t, \boldsymbol{w})}{\partial t} + \mathcal{N}(u(\boldsymbol{x}, t, \boldsymbol{w})) - \nu \frac{\partial^2 u(\boldsymbol{x}, t, \boldsymbol{w})}{\partial x^2}, \tag{4.1}$$

And the proposed training loss will be modified to

$$\mathcal{L}(\boldsymbol{w}, \nu) = \mathcal{L}_r(\boldsymbol{w}, \nu) + \mathcal{L}_b(\boldsymbol{w}) + \mathcal{L}_0(\boldsymbol{w}) + \alpha_{visc}\mathcal{L}_{visc}(\nu) \tag{4.2}$$

This loss is subsequently minimized via a separate stage of gradient descent, independent of network optimization, but is performed simultaneously in the same network training loop.

4.1.2 Parameterized Artificial Viscosity Map

Parameterized artificial viscosity relies on the a-priori intuition that a shock front will manifest immediately after initialization. This method removes artificial viscocity from the *entire* problem domain and allows for a trainable artificial viscosity only along the shock front, which facilitates training along the discontuinity but does not comprimize accuracy along other portions of the domain. A problem etup such as this is not possible with traditional solvers of Buckley-Leverett, and is unique to PINNs.

[1]Released in a manuscript entitled *Physics-Informed Neural Networks with Adaptive Localized Artificial Viscosity [69]*, primarily authored by Emilio Coutinho

In this instance, we learn the shock front viscocity and only apply it to the front as it moves in the time domain, however in order to perform this operation we must first identify the velocity of the shock front itself. Therefore, the PINN formulation simultaneously learns the shock front velocity as well as the value of the artificial viscosity required to train the forward model. Therefore, the final PDE minimized via the PINN formulation is

$$r(\boldsymbol{x}, t, \boldsymbol{w}, \nu) = \frac{\partial u(\boldsymbol{x}, t, \boldsymbol{w})}{\partial t} + \mathcal{N}(u(\boldsymbol{x}, t, \boldsymbol{w})) - \nu_{\max}\nu(\boldsymbol{x}, t)\frac{\partial^2 u(\boldsymbol{x}, t, \boldsymbol{w})}{\partial x^2}, \tag{4.3}$$

where $\nu(\boldsymbol{x}, t)$ is a spatial-temporal map that has its values bounded between 0 and 1, and $\nu_{\max}$ is the maximum value of artificial viscosity. The map $\nu(\boldsymbol{x}, t)$ can be parameterized, for example using the shock front velocity. In this case, we are not learning an individual weight or solution for each collocation point, as is the case for the SA-PINN, but rather a set of hyperparameters used to build a map of artificial viscosity at the shock front.

4.1.3 Residual-Based Artificial Viscosity Map

The final approach modeled is a residual-based artificial viscosity parameter that is relient on the gradients of the residual, i.e. areas with higher residual gradients (wrt spatiotemporal variables, not loss function gradients as in the SA-PINN) indicates discontinuities in the solution, and therefore is likely the location of the shock in the solution. In this regime, the residual of the PDE is used to target the location of the residual by solving the residual form

$$r(\boldsymbol{x}, t, \boldsymbol{w}, \nu) = \frac{\partial u(\boldsymbol{x}, t, \boldsymbol{w})}{\partial t} + \frac{\partial f(u)}{\partial x} - \nu_{\max}\nu(\boldsymbol{x}, t)\frac{\partial^2 u(\boldsymbol{x}, t, \boldsymbol{w})}{\partial x^2}, \tag{4.4}$$

where $\nu(\boldsymbol{x}, t)$ is a spatial-temporal map individualized to each collocation point, a major difference from the aforementioned approach in the previous section.

The residual-based artificial viscosity map is defined as:

$$\nu = \min(\nu_1, \nu_r), \tag{4.5}$$

where ν_1 is called first-order viscosity vector and ν_r is the high-order residual viscosity vector. At each collocation point i the first-order viscosity is calculated by:

$$\nu_{1,i} = \max_{loc} |f'(u)_i|, \tag{4.6}$$

where the notation $\max_{loc}$ represents the maximum value taken over the neighbors of the collocation point with index i. The high-order residual viscosity at collocation point index i is defined as:

$$\nu_{r,i} = \max_{loc} \frac{|R(u)_i|}{n(u)_i}, \tag{4.7}$$

where the $R(u)$ is the inviscid PDE residual. The normalization term $n(u)$ is chosen as:

$$n(u)_i = \left| \tilde{u}_i - \|u - \overline{u}\|_{L^\infty(\Omega)} \right|, \tag{4.8}$$

where:

$$\tilde{u}_i = \max_{loc} u_i - \min_{loc} u_i, \tag{4.9}$$

$\overline{u}$ is the mean of u, and the notation $\min_{loc}$ is defined similarly as the $\max_{loc}$.

The artificial viscosity map obtained from Eq. 4.5 has its numerical values obtained from the fraction flow curve $f(x)$ and a relation of the inviscid residual equation. So, the ν map will be normalized between 0 and 1 using the following equation:

$$\hat{\nu} = \frac{\nu - \nu_{\min}}{\nu_{\max} - \nu_{\min}}, \tag{4.10}$$

and since this is a simple transformation, we will remove the upper hat notation and keep denoting the normalized artificial viscosity map simply as ν. The learnable parameter $\nu_{\max}$ will control the magnitude of the viscosity values applied to the diffusion term.

4.2 PINNs for Radiative Transfer

In this work, we evaluate the ability of PINNs to expand to solving problem in astrophysics through our collaboration with Texas A&M Physics and Astronomy. Specifically, we address the ability of PINNs in solving the spectrum of a Type 1a supernovae. This problem is interesting in the realm of astrophysics because the explosion mechanisms are not entirely clear due to the complexity of the underlying physical systems, typically resulting in complex calculations using models that must take physical "short cuts" to generate results. This is particularly true in the nucleosynthesis and the hydrodynamics of the system, in which the complexity/dimensionlaity can be overwhelming. In this case, we are interested in solving the inverse problem to yield the specific intensity (spectrum) of the explosion using the radiative transfer equation [70] and the inverse modeling power of PINNs.

In the one-dimensional spherical symmetric coordinate, the radiative transfer equation in the rest frame is:

$$\cos(\varphi)\frac{\partial I}{\partial r} - \sin(\varphi)\frac{\partial I}{\partial \varphi}\frac{1}{r} - j_{\text{em}}\left(\frac{\bar{\nu}}{\nu}\right)^{-2} + k_{\text{abs}}\left(\frac{\bar{\nu}}{\nu}\right) I = 0 \ , \tag{4.11}$$

where I, as a function of spatial coordinate, viewing direction and frequency, is the specific intensity, r is the radius, φ is the angle between the viewing direction and the radius vector, k_{abs} is the absorption term, j_{em} is the emission term, $\left(\frac{\bar{\nu}}{\nu}\right)$ is the frequency ratio in the comoving frame and the rest frame which observes the following relation:

$$\frac{\bar{\nu}}{\nu} = \gamma[1 - \cos(\varphi)\beta] \ , \tag{4.12}$$

where $\gamma = (1 - \beta^2)^{-0.5}$ is the Lorentz factor, $\beta = v/c$ is the velocity of the material divided by the speed of light [71]. Additional information on the specifics of the terms present in 4.11 can be found in the full manuscript of this work[2] and are unique to ratiative transfer and its various

[2]To be released in a manuscript tentatively titled *Using Physics Informed Neural Networks for Supernova Radiative*

modeling subtleties. Additionally, the time independent gamma-ray radiative transfer equation is:

$$\cos(\varphi)\frac{\partial I_\gamma}{\partial r} - \sin(\varphi)\frac{\partial I_\gamma}{\partial \varphi}\frac{1}{r} + (k_{\mathrm{C}} + k_{\mathrm{p}})I_\gamma - j_{\mathrm{C}} - j_{\mathrm{r}} = 0 \ , \tag{4.13}$$

where j_{r} is the gamma ray source in the supernova atmosphere.

In the case of our work, done in collaboration with Texas A&M University Physics and Astronomy, we seek to use a PINN-based inverse model to capture the intensity of the spectrum $I_\nu = f(r, \varphi, w)$, where w represents the trainable parameters in the neural network, I_ν is a vector representing the intensity at a given spectral sampling grid.

To train the neural network, we sample upper BC, lower BC, and collocation points from the interior of the domain (r, φ). The first set of collocation points are randomly sampled in the parameter space from uniform distributions: $r_{i,p} \in U(r_{\min}, r_{\max})$, $\varphi_{i,p} \in U(0, \pi)$, denote as reesidual collocation points. The second set of collocation points are sampled in $r_{j,l} = r_{\min}$, $\varphi_{j,l} \in U(0, \pi/2)$, and are lower BC points. The third set of collocation points are sampled in $r_{k,u} = r_{\max}$, $\varphi_{k,u} \in U(\pi/2, \pi)$, and are upper BC points.

The PDE collocation points are used into equation (4.11) or equation (4.13) to calculate the residual $R_{i,p}$. To notice, the computation processes in the neural network are addition and multiplication of matrices and non-linear differentiable activation functions (i.e., $\tanh$, ReLU), the partial differential terms in equation (4.11) or equation (4.13) are thus calculated analytically using the chain rule of derivation, and there is no need to sample $(r_{i,p} + dr, \varphi_{i,p} + d\varphi)$ for numerical gradients. The upper and lower boundary collocation points are directly used to calculate the predicted specific intensities: $I_{k,u} = f(r_{\max}, \varphi_{k,u}, w)$, $I_{j,l} = f(r_{\min}, \varphi_{j,l}, w)$, then calculate the residual with respect to the pre-defined boundary conditions $R_{j,l}$, $R_{k,u}$.

The loss function is written as:

$$L = w_p \sum_{i,\nu} R_{i,p}^2 + w_l \sum_{j,\nu} R_{j,l}^2 + w_u \sum_{k,\nu} R_{k,u}^2 \ , \tag{4.14}$$

Transfer Simulation, primarily authored by Xingzhuo Chen where w_p, w_l, w_u are the weight parameters which should be specified before training, and the summation is over collocation points and spectral sampling pixels.

This approach provides insight into the possibilities of PINNs in radiative transfer, and the results of both the spectrum and the temperature vs. velocity plots closely match as compared to real data from SN 2011fe at 12.35 days after explosion, a spectrum [72] observed by Double Spectrograph (DBSP) mounted on Palomar 200-inch (P200) Telescope.

4.3 PINNs for Solving Cahn-Hilliard

The work in this section is performed in collaboration with Texas A&M University Dept. of Materials Science and Engineering[3], and primarily revolves around solving the phase-field equations for material decomposition.

4.3.1 Solving 1D Cahn-Hilliard with PINNs

In the 1D case, the system can be defined as follows:

$$u_t - (\gamma_2(u^3 - u) - \gamma_1 u_{xx})_{xx} = 0\,,\;\; x \in [-1,1],\; t \in [0,1]\,, \tag{4.15}$$

$$u(x,0) = -cos(2\pi x)\,, \tag{4.16}$$

$$u(t,-1) = u(t,1)\,, \tag{4.17}$$

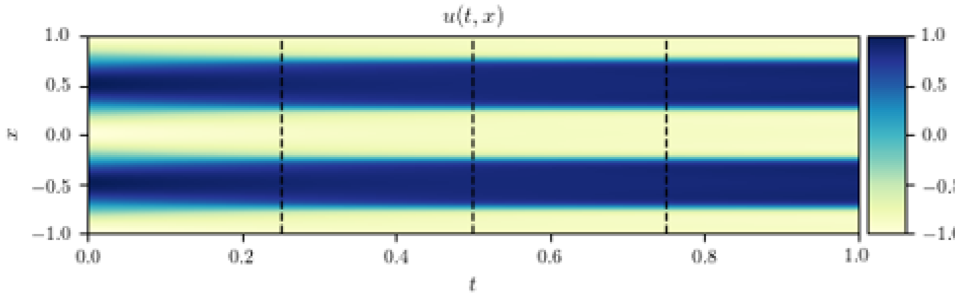
$$u_x(t,-1) = u_x(t,1)\,. \tag{4.18}$$

This system is typically solved using Finite Differencing or Fourier Spectral Methods when used in the context of microstructure informatics [73, 18], however it can also be solved using the SA-PINN with 3% error, as shown in figure 4.3.

4.3.2 Mesoscale Multi-Physics Constrained Neural Network

In the 2D case, the model itself is more complex, but can be shown as the 4th order PDE:

[3]in close collaboration with Dr. Dehao Liu, Binghamton University New York



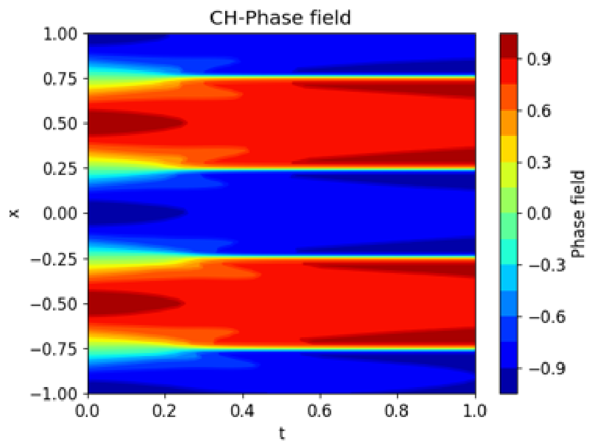


Figure 4.1: *Top:* PINN-learned solution of 1D Cahn-Hilliard. *Bottom:* Exact phase-field solution of 1D CH. The L2 error is approx. 3%.

$$F(c, \nabla c) = \frac{1}{4}\beta c^4 - \frac{1}{2}\alpha c^2 + \frac{1}{2}\kappa|\nabla c|^2 \tag{4.19}$$

$$\frac{\partial c}{\partial t} = \nabla M \cdot \nabla \frac{\partial F}{\partial c} \tag{4.20}$$

which is generally solvable via Fourier spectral methods[18]. IN this work, we seek to solve the 2D CH equations using PINNs, specifically a Mesoscale Multi-Physics Constrained Neural Network. There are a few varities of MM-PCNN attempted here and the results are still in progress. The snapshots shown haere are examples of training runs of the MC-PCNN for instances in which only training data are utulized, i.e. data from the evolution is used to train a neural network that predicts a microstructure at a specified time. The other example is the MM-PCNN trained on the same data, but with the physics included in the model. A thorough understanding of Multiscale

Modeling via PINNs, as well as phase field methods, will also be able to assist in fracture modeling, such as modeling the dynamics shown in [74] and [75].

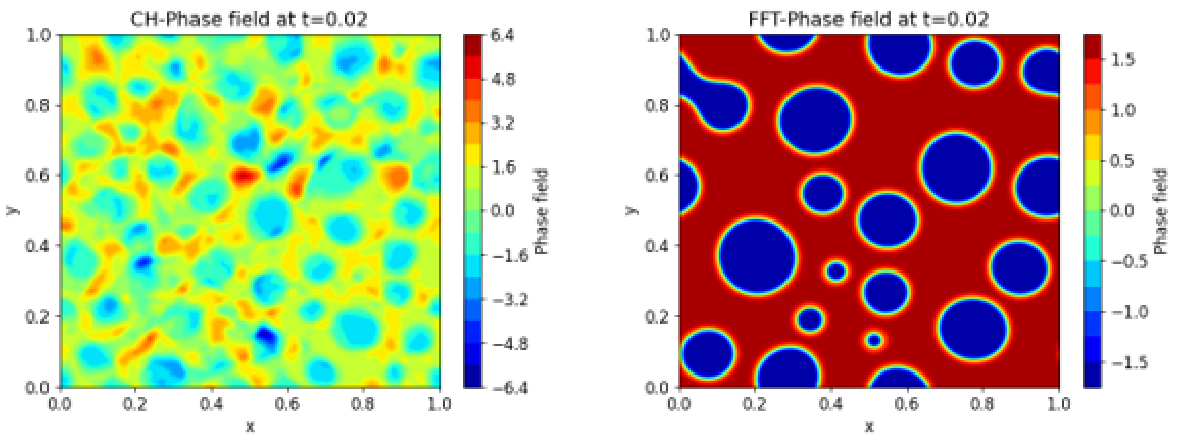


Figure 4.2: *left:* MM-PCNN-learned solution of 2D Cahn-Hilliard without physics of the model included. *right:* Exact phase-field solution of 2D CH.

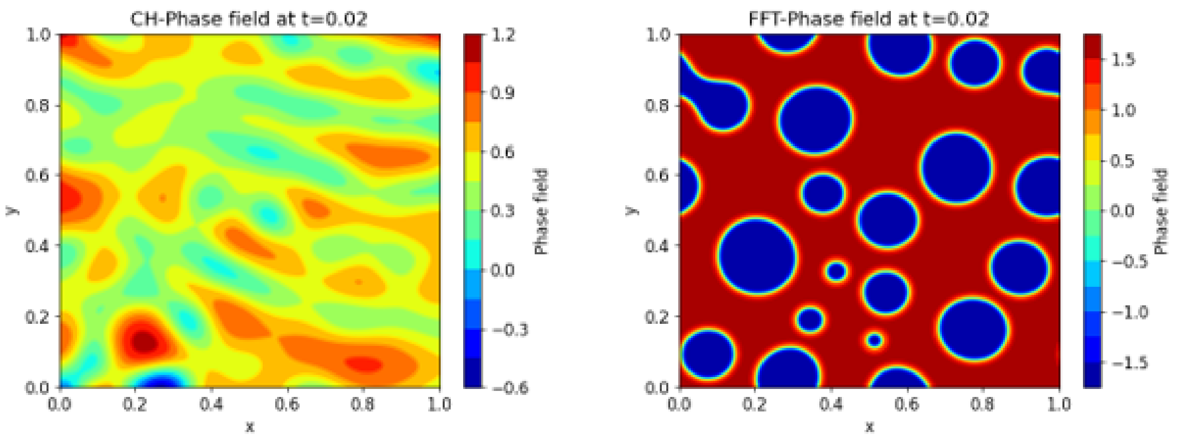


Figure 4.3: *left:* MM-PCNN-learned solution of 2D Cahn-Hilliard with physics of the model included. *right:* Exact phase-field solution of 2D CH.

5. SUMMARY AND CONCLUSIONS

In this work, we discussed applications and advancements of scientific machine learning - from machine learning with physics applications leading into PINNs, related software, and ongoing applications in various fields of science and technology.

In the section I, we described a technique to blend microstructural image data into with descriptive statistics to better perform regression of target parameters, in this case the atomic and phsae compositions. We showed that by incorporating the physics of the model, along with the image data, we could effectively regress the parameters better than the descriptive or image data alone.

In section II, we described a new training paradigm for PINNs called the Self-Adaptive PINN (SA-PINN). The algorithm enhances PINN training by targeting the training points where the loss is the highest, and forcing the neural network function approximator to decrease the loss at those points faster than the rest. This process drastically improves training and increases the overall L2 error convergence of the PINN forward approximation on various benchmark systems. We consistently show that the SA-PINN increases convergence over the baseline, as well as many of the emerging weighting schemes in current literature. We also analyze utilizing a Gaussian process weight generating function, which enables SGD training of the SA-PINN and conclude with NTK analysis to provide a theoretical justification of the improved SA-PINN training.

In section III, we introduce TensorDiffEq, a software suite designed to solve systems of PDEs using multi-GPU training of PINNs. The software is designed such that a user can input their system of PDEs into the solver in a human-readable fashion and call a simple `model.fit` method to train their system. Numerous BCs types are included, and training in a multi-GPU environment can be enabled with a simple boolean flag, with no change to the underlying code.

In section IV, we introduce emerging research in the PINNs space, including modification to the PINN algorithm to accommodate hyperbolic systems, wherein we can effectively model a shock front, as well as radiative transfer equations, where we demonstrate that PINNs can capture incredibly complex physical dynamics of a supernovae explosion.

5.1 Challenges

Physics-Informed Neural Networks present extremely unique challenges in their training, accuracy, and computational complexity. Active research is going in to improving all those factors but the fact remains that, in most instances, the traditional numerical solvers that have been used leading up to this point are, in fact, faster and generally more accurate. However, there is a strong motivation for the modeling characteristics that PINNS can bring, specifically in inverse modeling and in digital twins [76, 77], that traditional numerical methods simply cannot provide due to their nature. It is for this reason that, despite their challenges and current drawbacks, PINNs have gained an incredible amount of traction in the last 5 years as an emerging and disruptive technology, with the original Raissi PINNs paper series [78] gaining almost 3000 citations since 2017.

5.2 Further Study

Further study in the field of PINNs is truly unlimited. Arguably, the PINN paradigm is one of the most disruptive applied mathematical and computational endeavors in the last 5 years. The field is still emerging, and massive strides are made every week. The volume of information is quickly becoming overwhelming, and new PINN papers are released almost daily. Further study, from a high level, will likely entail using PINNs for application specific inverse modelling, as well as moving into the more applied domain via digital twin modelling. Both of these applications move PINNs out of the novelty space and into the impact space, where PINNs are actively contributing to technology development and learning information from physical experiments. This was one of the original intents of Raissi *et. al* when the seminole PINN paper was published, and a section highlighted heavily in tohe original manuscript. It still rings true today that PINNs can be heavily utilized in solving real-world problem, and account for real-world dynamics, which is what makes the concept so compelling for academia.

In regards to specific, actionable next steps to research - a digital twins model for materials design could be attainable with a good amount of effort and creativity, which will likely be the author's next academic focus. With a small amount of real sampled data, there is a potential for massive insights in next generation design of materials, potentially for military applications or otherwise, that could be fertile ground for future work.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[4] D. Ramachandram and G. W. Taylor, "Deep multimodal learning: A survey on recent advances and trends," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 96–108, 2017.

[5] J. Gao, P. Li, Z. Chen, and J. Zhang, "A survey on deep learning for multimodal data fusion," *Neural Computation*, vol. 32, no. 5, pp. 829–864, 2020.

[6] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, "Multimodal deep learning," 2011.

[7] N. Srivastava and R. R. Salakhutdinov, "Multimodal learning with deep boltzmann machines," in *Advances in neural information processing systems*, pp. 2222–2230, 2012.

[8] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[9] S. Sun, H. Shi, and Y. Wu, "A survey of multi-source domain adaptation," *Information Fusion*, vol. 24, pp. 84–92, 2015.

[10] J. Lee, P. Sattigeri, and G. Wornell, "Learning new tricks from old dogs: Multi-source transfer learning from pre-trained networks," in *Advances in Neural Information Processing Systems*, pp. 4372–4382, 2019.

[11] J. Li, W. Wu, D. Xue, and P. Gao, “Multi-source deep transfer neural network algorithm,” *Sensors*, vol. 19, no. 18, p. 3992, 2019.

[12] Z. Xu and S. Sun, “Multi-source transfer learning with multi-view adaboost,” in *International conference on neural information processing*, pp. 332–339, Springer, 2012.

[13] J. Guo, W. Che, D. Yarowsky, H. Wang, and T. Liu, “A representation learning framework for multi-source transfer parsing,” in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[14] S. Lathuilière, P. Mesejo, X. Alameda-Pineda, and R. Horaud, “A comprehensive analysis of deep regression,” *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “ImageNet: A Large-Scale Hierarchical Image Database,” in *CVPR09*, 2009.

[16] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[17] I. Steinbach, “Phase-field model for microstructure evolution at the mesoscopic scale,” *Annual Review of Materials Research*, vol. 43, pp. 89–107, 2013.

[18] V. Attari, P. Honarmandi, T. Duong, D. J. Sauceda, D. Allaire, and R. Arroyave, “Uncertainty propagation in a multiscale calphad-reinforced elastochemical phase-field model,” *Acta Materialia*, vol. 183, pp. 452–470, 2020.

[19] F. Chollet *et al.*, “Keras.” `https://keras.io`, 2015.

[20] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[21] F. Chollet, *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

[22] A. Rosebrock, “Keras, regression, and cnns,” Jan 2019.

[23] A. Rosebrock, “Keras: Multiple inputs and mixed data,” Apr 2020.

[24] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, “Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence,” 2 2019.

[25] M. Raissi, P. Perdikaris, and G. E. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.

[26] M. Raissi, “Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations,” *arXiv preprint arXiv:1804.07010*, 2018.

[27] C. L. Wight and J. Zhao, “Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks,” *arXiv preprint arXiv:2007.04542*, 2020.

[28] S. Wang, X. Yu, and P. Perdikaris, “When and why pinns fail to train: A neural tangent kernel perspective,” *arXiv preprint arXiv:2007.14527*, 2020.

[29] M. Dissanayake and N. Phan-Thien, “Neural-network-based approximations for solving partial differential equations,” *communications in Numerical Methods in Engineering*, vol. 10, no. 3, pp. 195–201, 1994.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[31] J. Revels, M. Lubin, and T. Papamarkou, “Forward-mode automatic differentiation in julia,” *arXiv preprint arXiv:1607.07892*, 2016.

[32] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic differentiation in machine learning: a survey,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 5595–5637, 2017.

[33] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.

[34] R. L. Burden and D. J. Faires, “Numerical analysis,” 1985.

[35] S. Wang, Y. Teng, and P. Perdikaris, “Understanding and mitigating gradient pathologies in physics-informed neural networks,” *arXiv preprint arXiv:2001.04536*, 2020.

[36] N. Moelans, B. Blanpain, and P. Wollants, “An introduction to phase-field modeling of microstructure evolution,” *Calphad*, vol. 32, no. 2, pp. 268–294, 2008.

[37] F. Wang, M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang, and X. Tang, “Residual attention network for image classification,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3156–3164, 2017.

[38] Y. Pang, J. Xie, M. H. Khan, R. M. Anwer, F. S. Khan, and L. Shao, “Mask-guided attention network for occluded pedestrian detection,” in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4967–4975, 2019.

[39] Y. Zhou, J. Yang, H. Zhang, Y. Liang, and V. Tarokh, “Sgd converges to global minimum in deep learning via star-convex path,” *arXiv preprint arXiv:1901.00451*, 2019.

[40] D. G. Luenberger and Y. Ye, *Linear and nonlinear programming*. Springer, 3rd ed., 2008.

[41] W. E. Lillo, M. H. Loh, S. Hui, and S. H. Zak, “On solving constrained optimization problems with neural networks: A penalty method approach,” *IEEE Transactions on neural networks*, vol. 4, no. 6, pp. 931–940, 1993.

[42] V. M. Mladenov and N. Maratos, “Neural networks for solving constrained optimization problems,” *Proc. of CSCC’00, Athens, Greece,(N. Mastorakis*, 2000.

[43] D. C. Liu and J. Nocedal, “On the limited memory bfgs method for large scale optimization,” *Mathematical programming*, vol. 45, no. 1-3, pp. 503–528, 1989.

[44] L. D. McClenny, M. A. Haile, and U. M. Braga-Neto, “Tensordiffeq: Scalable multi-gpu forward and inverse solvers for physics informed neural networks,” *arXiv preprint arXiv:2103.16034*, 2021.

[45] J. Shen and X. Yang, "Numerical approximations of allen-cahn and cahn-hilliard equations," *Discrete & Continuous Dynamical Systems-A*, vol. 28, no. 4, p. 1669, 2010.

[46] C. Kunselman, V. Attari, L. McClenny, U. Braga-Neto, and R. Arroyave, "Semi-supervised learning approaches to class assignment in ambiguous microstructures," *Acta Materialia*, vol. 188, pp. 49–62, 2020.

[47] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[48] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[49] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[50] S. Wang, X. Yu, and P. Perdikaris, "When and why PINNs fail to train: A neural tangent kernel perspective," *Journal of Computational Physics*, vol. 449, p. 110768, Jan. 2022.

[51] R. J. LeVeque *et al.*, *Finite volume methods for hyperbolic problems*, vol. 31. Cambridge university press, 2002.

[52] I. E. Lagaris, A. Likas, and D. I. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE transactions on neural networks*, vol. 9, no. 5, pp. 987–1000, 1998.

[53] C. Rackauckas and Q. Nie, "Differentialequations.jl – a performant and feature-rich ecosystem for solving differential equations in julia," *The Journal of Open Research Software*, vol. 5, no. 1, 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[54] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, "DeepXDE: A deep learning library for solving differential equations," *SIAM Review*, vol. 63, no. 1, pp. 208–228, 2021.

[55] O. Hennigh, S. Narasimhan, M. A. Nabian, A. Subramaniam, K. Tangsali, M. Rietmann, J. d. A. Ferrandis, W. Byeon, Z. Fang, and S. Choudhry, "Nvidia simnetˆ{TM}: an ai-accelerated multi-physics simulation framework," *arXiv preprint arXiv:2012.07938*, 2020.

[56] E. Haghighat and R. Juanes, "Sciann: A keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks," *Computer Methods in Applied Mechanics and Engineering*, vol. 373, p. 113552, 2021.

[57] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[58] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[59] S. M. Allen and J. W. Cahn, "Ground state structures in ordered binary alloys with second neighbor interactions," *Acta Metallurgica*, vol. 20, no. 3, pp. 423–433, 1972.

[60] L. McClenny and U. Braga-Neto, "Self-adaptive physics-informed neural networks using a soft attention mechanism," *arXiv preprint arXiv:2009.04544*, 2020.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[62] L. Lu, P. Jin, and G. E. Karniadakis, "Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators," *arXiv preprint arXiv:1910.03193*, 2019.

[63] O. Fuks and H. A. Tchelepi, “Limitations of Physics Informed Machine Learning for Nonlinear Two-Phase Transport in Porous Media,” *Journal of Machine Learning for Modeling and Computing*, vol. 1, no. 1, 2020. Publisher: Begel House Inc.

[64] C. G. Fraces, A. Papaioannou, and H. Tchelepi, “Physics Informed Deep Learning for Transport in Porous Media. Buckley Leverett Problem,” *arXiv:2001.05172 [physics, stat]*, Jan. 2020. arXiv: 2001.05172.

[65] S. Buckley and M. Leverett, “Mechanism of Fluid Displacement in Sands,” *Transactions of the AIME*, vol. 146, pp. 107–116, Dec. 1942.

[66] J. Reisner, J. Serencsa, and S. Shkoller, “A space–time smooth artificial viscosity method for nonlinear conservation laws,” *Journal of Computational Physics*, vol. 235, pp. 912–933, Feb. 2013.

[67] C. G. Fraces and H. Tchelepi, “Physics Informed Deep Learning for Flow and Transport in Porous Media,” *arXiv:2104.02629 [physics]*, Apr. 2021. arXiv: 2104.02629.

[68] H. J. Welge, “A Simplified Method for Computing Oil Recovery by Gas or Water Drive,” *Journal of Petroleum Technology*, vol. 4, pp. 91–98, Apr. 1952.

[69] E. J. R. Coutinho, M. Dall’Aqua, L. McClenny, M. Zhong, U. Braga-Neto, and E. Gildin, “Physics-informed neural networks with adaptive localized artificial viscosity,” *arXiv preprint arXiv:2203.08802*, 2022.

[70] S. Mishra and R. Molinaro, “Physics informed neural networks for simulating radiative transfer,” *Journal of Quantitative Spectroscopy and Radiative Transfer*, vol. 270, p. 107705, Aug 2021.

[71] J. I. Castor, “Radiative Transfer in Spherically Symmetric Flows,” , vol. 178, pp. 779–792, Dec. 1972.

[72] R. Pereira, R. C. Thomas, G. Aldering, P. Antilogus, C. Baltay, S. Benitez-Herrera, S. Bongard, C. Buton, A. Canto, F. Cellier-Holzem, J. Chen, M. Childress, N. Chotard, Y. Copin, H. K. Fakhouri, M. Fink, D. Fouchez, E. Gangler, J. Guy, W. Hillebrandt, E. Y. Hsiao,

M. Kerschhaggl, M. Kowalski, M. Kromer, J. Nordin, P. Nugent, K. Paech, R. Pain, E. Pécontal, S. Perlmutter, D. Rabinowitz, M. Rigault, K. Runge, C. Saunders, G. Smadja, C. Tao, S. Taubenberger, A. Tilquin, and C. Wu, "Spectrophotometric time series of SN 2011fe from the Nearby Supernova Factory," , vol. 554, p. A27, June 2013.

[73] L. McClenny, M. Haile, V. Attari, B. Sadler, U. Braga-Neto, and R. Arroyave, "Deep multimodal transfer-learned regression in data-poor domains," *arXiv preprint arXiv:2006.09310*, 2020.

[74] L. D. McClenny, M. I. Butt, M. G. Abdoelatef, M. J. Pate, K. L. Yee, H. Rajendran, D. Perez-Nunez, W. Jiang, L. H. Ortega, S. M. McDeavitt, *et al.*, "Experimentally validated multiphysics modeling of fracture induced by thermal shocks in sintered uo2 pellets," *arXiv preprint arXiv:2112.06645*, 2021.

[75] W. Jiang, T. Hu, L. K. Aagesen, and Y. Zhang, "Three-dimensional phase-field modeling of porosity dependent intergranular fracture in uo2," *Computational Materials Science*, vol. 171, p. 109269, 2020.

[76] Y. A. Yucesan and F. A. Viana, "A hybrid physics-informed neural network for main bearing fatigue prognosis under grease quality variation," *Mechanical Systems and Signal Processing*, vol. 171, p. 108875, 2022.

[77] R. G. Nascimento, M. Corbetta, C. S. Kulkarni, and F. A. Viana, "Hybrid physics-informed neural networks for lithium-ion battery modeling and prognosis," *Journal of Power Sources*, vol. 513, p. 230526, 2021.

[78] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, Feb. 2019.