SAND2019-12497C

Nonlocal Physics Informed Neural Networks

[Nicole Buczkowski^1, Marta D'Elia^2, and Michael Parks^2
1[University of Nebraska-Lincoln], 2[Sandia National Laboratory]

BACKGROUND

Neural Networks are a powerful tool that are forwarding machine learning. Working together in networks, neurons take inputs either from data in the case of input layers or the outputs of previous neurons in the case of hidden layers. Neurons make a decision using their activation function and their inputs and output that decision. Known data is used to train the network. Through one of the many possible gradient descent algorithms, the network will tune parameters (weights and biases) in the activation function to lower a prescribed cost function.

Nonlocal models have grown in popularity over recent years due to their applicability to things like image processing and fracture mechanics. Instead of using partial differential equations to model physical laws in materials, one can use integral differential equations. Instead of taking information at a point, one can take information around a point in that point's horizon. So that points on the boundary also have this horizon, we additionally take points around the domain in a collar. In this paper, we will be consider equations of the form

$$\begin{aligned} Lu &= f, \qquad x \in \Omega \\ u &= f, \qquad x \in \Gamma \end{aligned}$$

where *Lu* denotes a nonlocal operator acting on *u*, Ω is a given domain, and Γ is the associated collar.

Incorporated into *L* is the function $\mu(x,y)$, a symmetric kernel that records the interaction between *x* and a point in its horizon, δ. Given a specific choice in kernel, the nonlocal equation converges to the local equation as δ goes to 0.

In recent studies (1),(2) it was discovered that when inputting information about *u* from an associated partial differential equation (utilizing physical information), an optimizer can predict the original function *u* with a certain degree of accuracy. This was expanded to fractional partial differential equations where there is no chain rule to utilize in fractional calculus. This was further expanded to consider data-driven discovery of partial differential equations (1). Data was used in this case to train *u* and certain parameters in the original equations (e.g. what order of partial derivative for fPINNs).

METHODOLOGY

For the forward problem, we consider equations of the form above. Similar to (1), we will be inputting a training set of N_d training points in the domain Ω for the nonlocal equation, $Lu = f$. However, differently from (2), we will be matching boundary conditions by providing points in the collar N_c for *u*.

The model will take those training points and from them, tune weights and biases to predict a function u_{NN} that models the training points. Additionally the values from training points and values nearby are used to calculate an estimate for the nonlocal equation applied to the function *u*. We use Gaussian Quadrature with *M* points in the quadrature to estimate the integral for u_{NN} . The goal is to minimize

$$loss = loss_u + loss_{Lu}$$

$$= \sum_{x_c \in N_c} (u_{NN}(x_c) - u(x_c))^2 + \sum_{x_d \in N_d} (Lu_{NN}(x_d) - Lu(x_d))^2$$

We then test the model with a set of testing points, different from those used to train the model. This tests the model's ability to predict other values.

For the inverse problem, we also consider equations of the form (1.1). The main difference in the inverse problems is that we will now be consider parameter prediction for a parameter in the kernel. So when we approximate Lu_{NN}, we utilize $\mu_{NN}(x,y)$ instead of the exact kernel. This $\mu_{NN}(x,y)$ will have a parameter in it that we would like for the code to predict. For example, if we consider the kernel $\mu_{NN}(x,y) = \frac{d}{\partial^5}(x-y)^2$. For the nonLocal Laplacian to converge to the traditional Laplacian, our goal is for the code to output $d = \frac{5}{2}$.

While the code is mostly the same as in the forward problem, we have a few key differences. We input a training set of N_d training points in the domain Ω for the nonlocal equation. We use the same training points for *u* in addition to those in the collar: $N_d + N_c = N_f$.

The code optimizes

$$loss = loss_u + loss_{Lu}$$

$$= \sum_{x_f \in N_f} (u_{NN}(x_f) - u(x_f))^2 + \sum_{x_d \in N_d} (Lu_{NN}(x_d) - Lu(x_d))^2$$

The code guesses a *u* and sets d to some given initialized value, then computing Lu_{NN} based off of that guess for u_{NN} and the current $\mu_{NN}(x,y)$ (which starts with the initialized *d*). Next it will over many iterations attempt to minimize the *loss* by changing *d* and *u*.

RESULTS: FORWARD PROBLEM

We test the model parameters to check the convergence rates and find optimal parameters. Unless otherwise noted, we use 50 training points in the domain, 50 points in each side of the collar, 150 test points, a width of 4, a depth of 4, a learning rate of $5e^{-5}$, and 20 points for the Gaussian Quadrature. We considered a few different possible sources of error: the discretization error, sampling size error, and neural network error. Later in this section, we also look at the affects of Gaussian White Noise.

We use the following problem set up to check the convergence of u_{NN}. The exact solution and equation are

$$u = x^2 - x^4$$
$$\mu(x,y) = \frac{3}{2\delta^3}$$
$$Lu = \frac{6}{5}\delta^2 - 12x^2 + 2$$

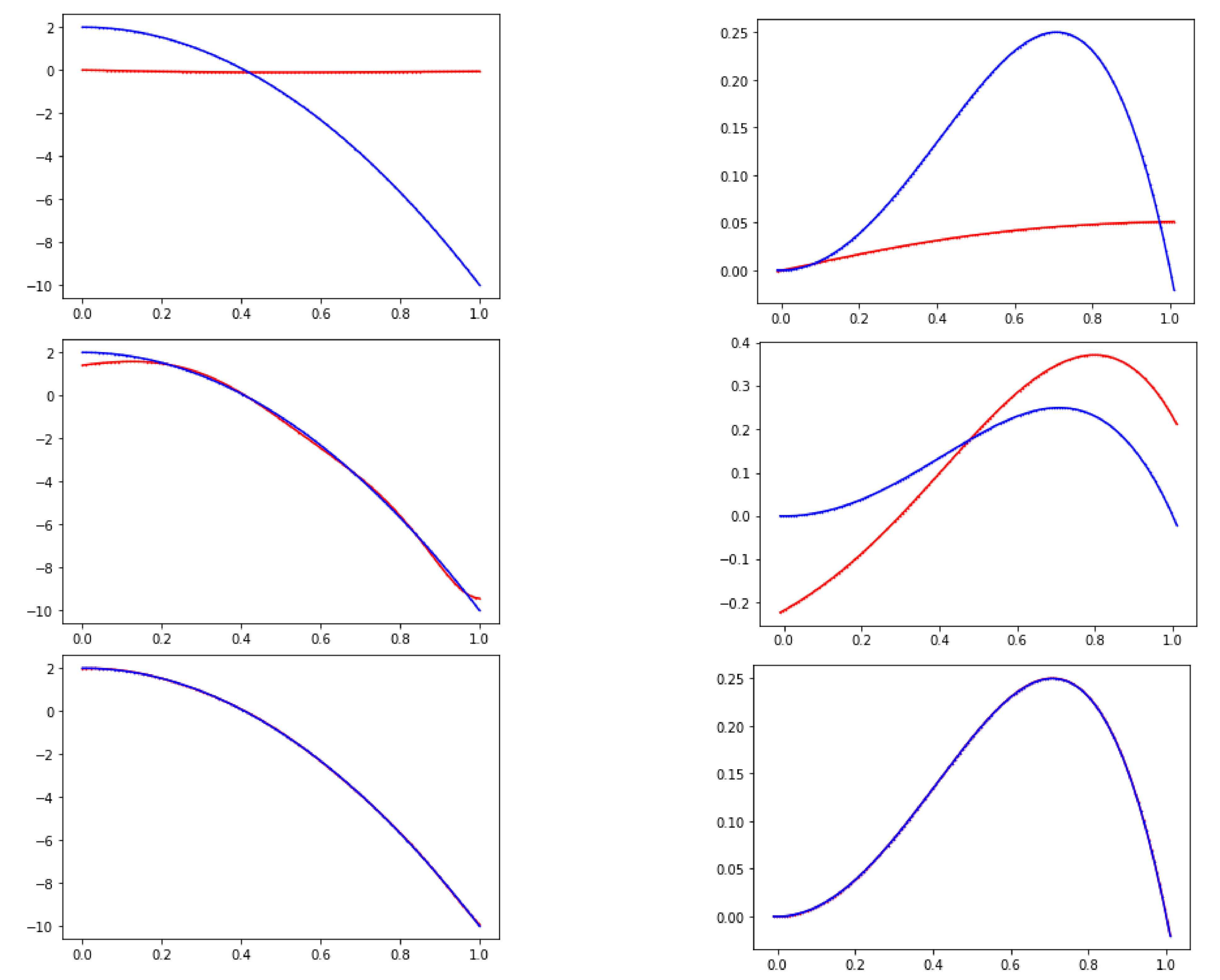
over the domain Ω = (0,1) with δ =0.01, so that Γ =(- δ,0)U(1, 1+ δ)$.

In the plots below we show how *u* converges as well as how *Lu* converges.

Since we are using Gaussian Quadrature to estimate the integral of our predicted u_{NN}, we also know that the error is O(u^{2n}). So in our example the if u_{NN} is well approximated by polynomials of degree 19 or higher, this error will be approximately 0. For the error generated by the training points, we consider varying the number of points in the domain N_d, the points in the collar N_c , and both simultaneously. We found that before 40 in each we have a little bit of instability.

We also considered a few causes for the neural network error. The first two, width and depth, refer to the architecture of the neural network (depth being the number of layers and width the neurons per layer). We also consider the impact of the learning rate. We found that after a width of 4 the error saturates. After a depth of 4, the error is of the same magnitude for this iteration. So we hold our depth at 4. For the impact of learning rate, we consider several constant learning rates. Outside e^{-3} and e^{-5} the loss and errors don't converge well. We take a closer look at the learning rates $5e^{-3}$, $5e^{-4}$, and $5e^{-5}$. The error and losses initially decrease and then increase and are unstable in $5e^{-3}$, $5e^{-4}$. We use $5e^{-5}$ as it is much more stable.

Now we consider Gaussian white noise added to the forcing term. We consider the case where *u* is not given points in the domain. As noise increased, the error of *u* increased, but the neural network did not suffer from over-fitting. However surprisingly, the optimizer did significantly worse when points were added in the domain for noise, even though it did better when there was no noise.



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

RESULTS: INVERSE PROBLEM

We next consider results for the inverse problem. Still predicting a u_{NN}, we also have the optimizer find a parameter *d* in the kernel. We first consider the case with a polynomial *u* and a constant kernel. We consider again the setup

$$u = x^2 - x^4$$
$$\mu(x,y) = \frac{3}{2\delta^5}$$
$$Lu = \frac{6}{5}\delta^2 - 12x^2 + 2$$

but now when computing the nonlocal Laplacian for the predicted u_{NN}, we utilized $\mu_{NN}(x,y) = \frac{d}{\delta^5}$ where the actual value of *d* is $\frac{3}{2}$.

There are two troubling cases in this parameter prediction due to bad initializations: distance from actual value and difference in sign from the initial value. The former can be seen when initialized at 10. Convergence here is significantly slower. To speed up this process, one solution is to multiply $loss_u$ by a scalar larger than 1, so that the optimizer predicts a better *u*, then forces the parameter to converge more quickly in order to match the nonlocal equation.

The second issue is initializing at a value of opposite sign to the desired value (or at 0 sometimes), such as -1 in this case. Not only does the value of the kernel parameter seem to be diverging to negative infinity for incorrect signed initializations, but the error for *u* is again significantly worse than that of the case with a good parameter prediction. However we can actually utilize this poor error in our favor. One option is to re-initialize to a new value at a certain iteration given a certain error. We can do this without disrupting the current weights and biases in the neural network. Another option is to restrict values to be in a certain range as in (2) for fractional PDEs.

We can also consider the case where the kernel is not simply a constant:

$$u = x^2 - x^4$$
$$\mu(x,y) = \frac{3}{2\delta^5}(x-y)^2$$
$$Lu = \frac{4d}{5} - \frac{4\partial^3 d}{7} - \frac{24dx^3}{5}$$

but now when computing the nonlocal Laplacian for the predicted u_{NN}, we utilized $\mu_{NN}(x,y) = \frac{5}{2\delta^5}(x-y)^2$ where the actual value of *d* is $\frac{5}{2}$. Notice that the structure of the kernel was changed so that we are still only predicting the parameter *d*. We consider several different initializations of the parameter *d* and consider the 100,000th iteration in the equation above. We see that we when started close enough, the parameter converges well. Even if we are further away the convergence appears to be slower.

Different Initializations for *d*

Initialization	Loss	Loss of *u*	Loss of *Lu*	Error of *u*	Error of *Lu*	Predicted *d*
-1	9.88717e-03	9.88570e-03	1.47844e-06	9. 9.60048e-01	2.90485e-04	-11.45071
0	5.05779e-08	9.82847e-12	5.05681e-08	3.03250e-05	5.37230e-05	2.50008
1	5.17000e-07	4.73172e-10	5.16527e-07	1.22691e-04	1.71699e-04	2.50043
15	4.41259e-03	4.41162e-03	9.72492e-07	6.41333e-01	2.35594e-04	13.43008

CONCLUSIONS

So far, physics-informed neural networks have been used in partial differential equations and fractional partial differential equations. We use nonlocal physics-informed neural networks to investigate their effectiveness on nonlocal equations. Nonlocal physics-informed neural networks have great accuracy in predicting a function *u* given information about *u* on the collar of its domain and information about *Lu* within the domain. We can consider the error coming from a few different sources: discretization error (when *u* is not integrable), the sampling size error, and the neural network error. Furthermore, nPINNs can accurately predict single parameters in the collar μ(x,y), though the results for more than one parameter are not quite as accurate.

As we continue this work, we look to find a way to more accurately predict more than one parameter in the kernel. We also aim to look at multidimensional equations.

Sources:
(1) Raissi, Maziar, Paris Perdikaris, and George 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 378 (2019): 686-707.
(2) Pang, Guofei, Lu Lu, and George Em Karniadakis. "fpinns: Fractional physics-informed neural networks." SIAM Journal on Scientific Computing 41.4 (2019): A2603-A2626.