Surrogate Modeling with Physics-Based Neural Networks

Alex Feild, AML Group

May 11, 2022

Abstract

The purpose of surrogate modeling is to replace some target model with an approximation that mimics the target's behavior. This is typically done to replace an explicit, explainable, but comparatively slow model with an implicit, less explainable, but much faster model. We will define terms for this report as follows: Physics-Based Machine Learning is the application of mathematical physics to machine learning. Physics-Informed Neural Networks (PINNs) include conservation terms, in the form of partial differential equations (PDEs), as part of their loss function. This report covers Physics-Based Neural Networks, with a focus on the use of applied mathematics in modeling with neural networks.

1 Background: Automatic Differentiation

Automatic differentiation (autodiff) is critical to some of the techniques described in this report. Automatic differentiation makes gradient learning methods much simpler to implement compared to analytic or numerical differentiation. Unlike analytic differentiation, autodiff does not require the programmer to compute and provide derivatives. Unlike numerical differentiation, autodiff gives exact results. Autodiff is not symbolic differentiation. Autodiff uses the chain rule:

$$y = f(g(h(x))) \tag{1}$$

$$y' = f'(g(h(x)))g'(h(x))h'(x) = \frac{df(g(h(x))}{dg(h(x))}\frac{dg(h(x))}{dh(x)}\frac{dh(x)}{dx} \tag{2}$$

This is usually done with intermediate variables. Assign:

$$z_0 = x, z_1 = h(z_0), z_2 = g(z_1), z_3 = f(z_2) \tag{3}$$

Then we can rewrite the rule in terms of these intermediate variables:

$$\frac{dy}{dx} = \frac{dy}{dz_2}\frac{dz_2}{dz_1}\frac{dz_1}{dx} \tag{4}$$

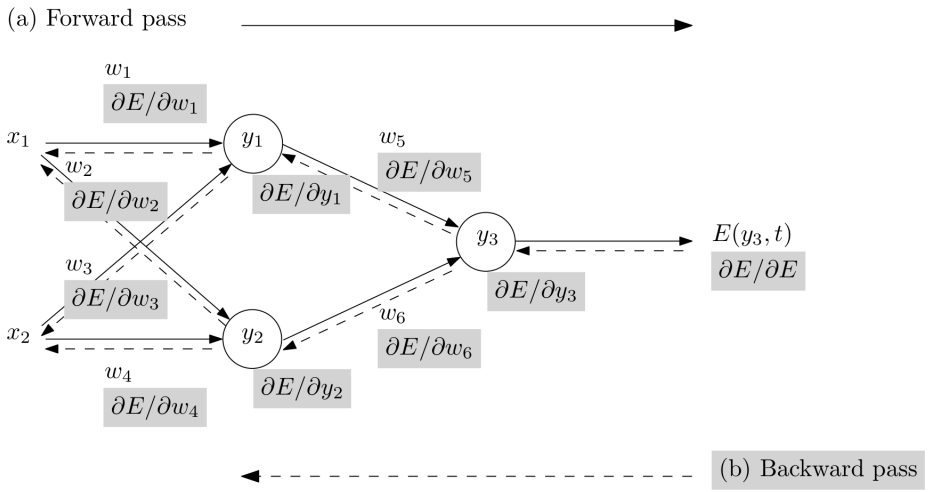


Figure 1: Autodiff graph by [7]

Autodiff exploits the fact all computer calculations can be reduced to a sequence of simple operations and functions, such as add, multiple, subtract, divide, sin, cos, exp, and log. Typically autodiff libraries record a graph of these computations. Since these simple operations and functions have know derivatives, a graph of derivatives can also be constructed. At the extremes there are two modes of computing the derivatives. Forward accumulation computes the derivative of each sub-expression while holding the independent variable fixed. And recursively substitutes the inner function with the derivative in accordance with the chain rule (while obeynig any branching in the graph).

$$\frac{dy}{dx} = \prod_{l=0}^{L} \frac{dz_{0+l}}{dx} \tag{5}$$

In contrast, reverse accumulation holds the dependent variable constant and recursively computes the derivative of each sub-expression (again, following any branching in the computational graph).

$$\frac{dy}{dx} = \prod_{l=0}^{L} \frac{dy}{dz_{L-l}} \tag{6}$$

Both full forward and reverse accumulation are extreme cases. In practice, packages like Pytorch only hold part of the graph in memory at any given moment, releasing unnecessary variables when not needed [3].

In addition to programming the known derivatives of basic operations and functions, we can use the dual numbers for automatic differentiation. Dual numbers consist of two real numbers a and b, where b is the multiple of the abstract number ε, with the property $\varepsilon^2 = 0$. The dual numbers are of the form

$a + b\varepsilon$. Using this definition we can now do some basic operations with the dual numbers:

$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon \tag{7}$$
$$(a + b\varepsilon) - (c + d\varepsilon) = (a - c) + (b - d)\varepsilon \tag{8}$$
$$(a + b\varepsilon) * (c + d\varepsilon) = ac + (ad + bc)\varepsilon \tag{9}$$
$$(a + b\varepsilon)/(c + d\varepsilon) = a/c + (b/c - ad/c^2)\varepsilon \tag{10}$$

The second real number in the dual number can be treated as the derivative of the first, giving $x + x'\varepsilon$. Below are some standard function in terms of the dual numbers.

$$sin(x + x'\varepsilon) = sin(x) + x'cos(x)\varepsilon \tag{11}$$
$$cos(x + x'\varepsilon) = cos(x) - x'sin(x)\varepsilon \tag{12}$$
$$exp(x + x'\varepsilon) = exp(x) + x'exp(x)\varepsilon \tag{13}$$
$$log(x + x'\varepsilon) = log(x) + (x'/x)\varepsilon \tag{14}$$
$$(x + x'\varepsilon)^k = x^k + x'kx^{k-1}\varepsilon \tag{15}$$
$$|x + x'\varepsilon| = |x| + x'signof(x)\varepsilon \tag{16}$$

Dual numbers will be stored in the computer as an ordered pair (x, x'). Vector gradients can be calculated by simply substituting entries with dual numbers. Using dual numbers for automatic differentiation can be achieved via operator overloading and replacing scalar values with dual numbers. Engineering experience shows that a computational graph for reverse mode accumulation may be more memory efficient, while forward mode accumulation with dual number may be easier to implement (via operator overloading).

1.1 Background: Autodiff Motivation: Backpropagation

The purpose of backpropagation is to apply gradient descent to neural networks. There are many modifications to the scheme in order to aviod local minima, but the basic gradient descent scheme is:

$$\boldsymbol{\theta}^i = \boldsymbol{\theta}^{i-1} - u^i \nabla \mathbf{L}(\boldsymbol{\theta}^{i-1}) \tag{17}$$

Recall the basic layout of a fully connected feed forward neural network shown in Figure 2. Gradient descent for each parameter becomes:

$$\theta_{j,k}^{r,i} = \theta_{j,k}^{r,i-1} + \triangle\theta_{j,k}^{r,i-1} \tag{18}$$
$$\triangle\theta_{j,k}^{r,i-1} = -u^i \frac{\partial \mathbf{L}}{\theta_{j,k}^{r,i-1}} \tag{19}$$

Where i is the iteration or timestep, r is the layer of the neuron that this parameter belongs to, j is the neuron index in r that the parameter θ belongs to, and k is the neuron in the $r - 1$ layer that this parameter connects to. Here

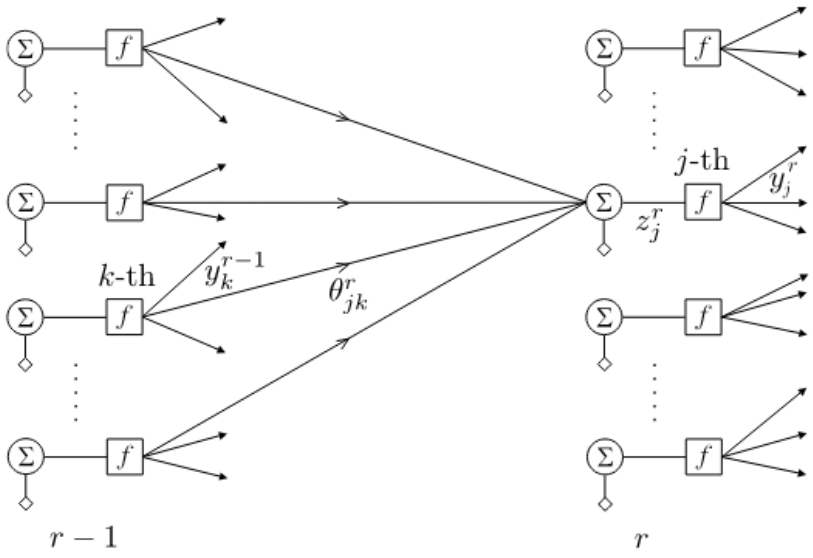


Figure 2: Basic neural network with layers, neurons, and weights labeled from [10].

the convention from [10] where 1.0 is added to the front of $y^{r-1,i}$ so that the bias is accounted for as part of the other weights in $\theta_j^{r,i}$ is used. Perhaps matrices will make things more clear:

$$\mathbf{y}_n^{r-1} = [1.0, y_{n,1}^{r-1}, ..., y_{n,K_{r-1}-1}^{r-1}] \tag{20}$$

$$\mathbf{Y}^{r-1^T} = \begin{Bmatrix} 1 & y_{0,1}^{r-1} & ... & y_{0,K_{r-1}-1}^{r-1} \\ ... & ... & ... & ... \\ 1 & y_{N-1,1}^{r-1} & ... & y_{N-1,K_{r-1}-1}^{r-1} \end{Bmatrix}$$

Where $\mathbf{Y}^{r-1}$ is transposed in preparation for multiplication, K_{r-1} is the number of outputs coming from the $r-1$ layer, and N is the number of samples in a batch.

$$\boldsymbol{\theta}^r = \begin{Bmatrix} \theta_{0,0}^r & ... & \theta_{J_r-1,0}^r \\ ... & ... & ... \\ \theta_{0,K_{r-1}-1}^r & ... & \theta_{J_r-1,K_{r-1}-1}^r \end{Bmatrix}$$

Where J is the number of neurons in this layer r. Multiplying them yields:

$$\mathbf{Z}^r = \mathbf{Y}^{r-1^T}\theta^r = \begin{Bmatrix} z_{0,0}^r & ... & z_{J_r-1,0}^r \\ ... & ... & ... \\ z_{0,N-1}^r & ... & z_{J_r-1,N-1}^r \end{Bmatrix}$$

Applying the activation function $f()$ gets $\mathbf{Y}^r$.

$$f(\mathbf{Z}^r) = \mathbf{Y}^r = \begin{Bmatrix} y^r_{0,0} & \cdots & y^r_{J_r-1,0} \\ \cdots & \cdots & \cdots \\ y^r_{0,N-1} & \cdots & y^r_{J_r-1,N-1} \end{Bmatrix}$$

If r is the last layer $r = L$, the loss function will compute on all J_L outputs.

$$\mathbf{Y}^L = \begin{Bmatrix} y^L_{0,0} & \cdots & y^L_{J_L-1,0} \\ \cdots & \cdots & \cdots \\ y^L_{0,N-1} & \cdots & y^L_{J_L-1,N-1} \end{Bmatrix}$$

$$\mathbf{L}_n(\boldsymbol{\theta}) = \begin{Bmatrix} L_0 \\ \cdots \\ L_{N-1} \end{Bmatrix}$$

$$\mathbf{L}(\boldsymbol{\theta}) \sum_{n=0}^{N-1} \mathbf{L}_n(\boldsymbol{\theta}) \tag{21}$$

Using the chain rule we can find the partial derivative of the error with respect the the weight of the last layer.

$$\frac{\partial \mathbf{L}_n}{\partial z^L_{j,n}} = \frac{\partial \mathbf{L}_n}{\partial y^L_{j,n}} \frac{\partial y^L_{j,n}}{\partial z^L_{j,n}} \tag{22}$$

$$\frac{\partial \mathbf{L}_n}{\partial \theta^L_{j,k}} = \frac{\partial \mathbf{L}_n}{\partial z^L_{j,n}} \frac{\partial z^L_{j,n}}{\partial \theta^L_{j,k}} \tag{23}$$

$$\frac{\partial \mathbf{L}}{\partial \theta^L_{j,k}} = \sum_{n=0}^{N-1} \frac{\partial \mathbf{L}_n}{\partial \theta^L_{j,k}} \tag{24}$$

Recall from above $z^r_{j,n}$ is the inner product of the neuron weights and the outputs of the previous layer.

$$z^L_{j,n} = {y^{L-1}_{n,k}}^T \theta^L_{j,k} \tag{25}$$

Recall that k_{r-1} is the number of neurons at the $r-1$ layer. Because z is the result of the inner product of y^{r-1} and θ, we get:

$$\frac{\partial z^L_{j,n}}{\partial \theta^L_{j,k}} = y^{L-1}_{n,k} \tag{26}$$

$$\frac{\partial z^L_{j,n}}{\partial \theta^L_{j}} = \mathbf{y}^{L-1}_{n} \tag{27}$$

The adjustment to $\boldsymbol{\theta}$ then becomes a vector of length K_{r-1}:

$$(\triangle \boldsymbol{\theta}^L_j)_n = -u * \frac{\partial \mathbf{L}_n}{\partial z^L_{j,n}} \mathbf{y}^{L-1}_n \tag{28}$$

$$\triangle \boldsymbol{\theta}^L_j = \sum_{n=0}^{N-1} (\triangle \boldsymbol{\theta}^L_j)_n \tag{29}$$

Automatic differentiation allows us to use nearly arbitrary loss functions and activation functions, as long as they are differentiable and ideally the loss function should have it's minimum at zero. However, let's do an analytical demonstration to joggle our minds. If the loss function is the sum of errors squared:

$$\mathbf{L}_n = \frac{1}{2}\sum_{j=0}^{J_L-1}(\hat{y}_{n,j} - y_{n,j})^2 \tag{30}$$

$$\frac{\partial \mathbf{L}_n}{\partial y^L_{j,n}} = (\hat{y}_{n,j} - y_{n,j}) \tag{31}$$

If the activation function is logistic sigmoid then:

$$\frac{\partial y^r_{j,n}}{\partial z^r_{j,n}} = f'(z^r_{j,n}) = \alpha f(z^r_{j,n})(1 - f(z^r_{j,n})) \tag{32}$$

Getting back to the general case useful for autodiff. Due to the successive dependence of the layers:

$$\frac{\partial \mathbf{L}_n}{\partial z^{r-1}_{n,j}} = \sum_{k=0}^{K_r-1}\frac{\partial \mathbf{L}_n}{\partial z^r_{k,n}}\frac{\partial z^r_{k,n}}{\partial z^{r-1}_{n,j}} \tag{33}$$

Beyond the final layer we need to accumulate error from the previous layer:

$$e^{r-1}_{n,j} = \sum_{k=0}^{K_r-1}\frac{\partial \mathbf{L}_n}{\partial z^r_{k,n}}\theta^r_{k,j} \tag{34}$$

Notice the indices notation has changed, due to the perspective of $e^{r-1}_{n,j}$. The parameter adjustment is then:

$$\triangle \boldsymbol{\theta}^{r-1}_j = -u\sum_{n=0}^{N-1} e^{r-1}_{n,j} f^{r-1\prime}(z^{r-1}_{j,n})\mathbf{y}^{r-2}_n \tag{35}$$

All of these derivatives can be calculated with autodiff. Part of the reason for it's use is to allows for more complicated and arbitrary loss functions and neural network architectures beyond the simple one demonstrated here. The version of gradient descent show here was also the simplest version, with now preventative measures to avoid local minima. It is also useful for evaluating neural networks as solutions to PDEs, which is reviewed below.

1.2 Background: Autodiff Motivation: PDEs

Suppose we want our neural network to satisfy the condition:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0 \tag{36}$$

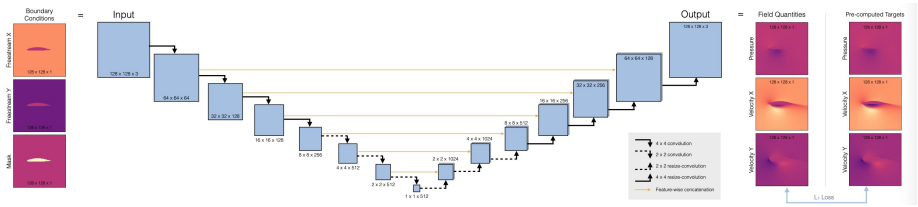


Figure 3: Data driven U-Net from [11] learning an airfoil problem.

After one pass through a neural network, we can use autodiff to get:

$$(u(x,y,z), \frac{\partial^u}{\partial x}, \frac{\partial^u}{\partial y}, \frac{\partial^u}{\partial z}) \tag{37}$$

We can then pass the first derivatives and use autodiff to get the second derivatives:

$$(\frac{\partial^u}{\partial x}, \frac{\partial^2 u}{\partial x^2}), (\frac{\partial^u}{\partial y}, \frac{\partial^2 u}{\partial y^2}), (\frac{\partial^u}{\partial z}, \frac{\partial^2 u}{\partial z^2}) \tag{38}$$

We can now use the first and second partial derivatives to calculate PDE constraint violations.

2 Physics-Based Neural Networks

2.1 Data Driven Methods

The data driven methods in this section do not include mathematical physics information in their loss function. However, they may exhibit architectures inspired by solutions to physics problems, such as Fourier Neural Operators (FNOs).

2.1.1 Traditional Methods

With sufficient data, traditional neural network methods can learn to approximate the physical phenomenon under investigation. In the example in figure 3 from [11] a U-net learns to infer velocity, $\mathbf{u} = u_x, u_y$, and pressure, p, after a timestep from an initial $mask, u_x, u_y$. A U-Net is a convolutional neural network with skip-layer connections. This network is trained with the typical data loss for supervised learning:

$$argmin_\theta \frac{1}{2} \sum_{k=0}^{K_L - 1} (N(\mathbf{x}_k, \boldsymbol{\theta}) - \hat{\mathbf{y}}_k)^2 \tag{39}$$

2.1.2 Neural Operators: Fourier Neural Operator

Recall that an operator takes one function and gives another. The purpose of neural operators is to extend spatially aware neural networks from finite-dimension to infinite dimension. Like the functions above can have arbitrary resolution in parameter space $u(x) = y$, the same property is desired for neural network that can take and map an arbitrary temporal-spatial arrangement of states and parameters, as a function, at arbitrary resolution. [8] formally describes neural operators as well as provide proof of their universal approximation. Formally, he purpose of neural operators in to learn a map between infinite dimensional spaces. More intuitively the purpose of neural operators is to learn a mapping between spaces of infinite resolution, from samples of finite resolution.

$$\mathcal{G}_\theta : \mathcal{A} \to \mathcal{U} \tag{40}$$

Where $\theta \in R^p$ of finite dimension and $\mathcal{G}^\dagger$ is the actual map. Neural operators can be trained to minimize the empirical risk:

$$\min_{\theta \in R^p} \frac{1}{n} \sum_{j=0}^{N-1} ||u_j - \mathcal{G}(a_j)||_u^2 \tag{41}$$

Where $a_j \sim u$ are independent and identically distributed samples so that $u_j = \mathcal{G}(a_j) + \mathcal{N}$.

While a_j and u_j are treated as functions, they need to be sampled for computations. If D_j is the n_j point discretization then $a_j|D_j, u_j|D_j \in R^{n_j}$ will be the collection of input output pairs. In practice, image planes are discretized to their resolution. Important to the neural operator approach is that it is independent of the discretization [8]. In other words a model can be trained on multiple resolutions and infer on arbitrary resolutions.

The framework of a neural operator includes 3 steps [8]:

1. Lifting: A pointwise function lifts the input into a higher dimensional space $R^{d_a} \to R^{d_{v_0}}$, where typically $d_{v_0} > d_a$. For images this is a 1x1 convolutional layer. Recall that convolutions have depth across the input channels (parameters).

2. Iterative Kernel Integration: Each hidden representation is mapped to the next $(v_t : D_t \to R^{d_{v_t}}) \to (v_{t+1} : D_{t+1} \to R^{d_{v_{t+1}}})$ by the sum of local linear operators, non-local integral operator, and bias function.

3. Projection. The last layer projects down to discrete space $R^{d_{v_T}} \to R^{d_u}$, where typically $d_{v_T} > d_u$ in a reflection of the first step. The neural operator is an iterative architecture with a sequence of functions, operating on input after it is lifted into higher dimensions by a neural network $P(a(x))$. Then several iterations of updates $v_t \to v_{t+1}$. [9] defines this as:

$$v_{t+1}(x) := \sigma(\mathcal{W} v_t(x) + (\mathcal{K}(a;\phi)v_t)(x)), \forall x \in D \tag{42}$$

Which is the same as Figure 4. Where the general kernel operator $\mathcal{K}$ is defined as:

$$(\mathcal{K}(a;\phi)v_t)(x) = \int_D \mathcal{K}(x, y, a(x), a(y); \phi)v_t(y)dy, \forall x \in D \tag{43}$$

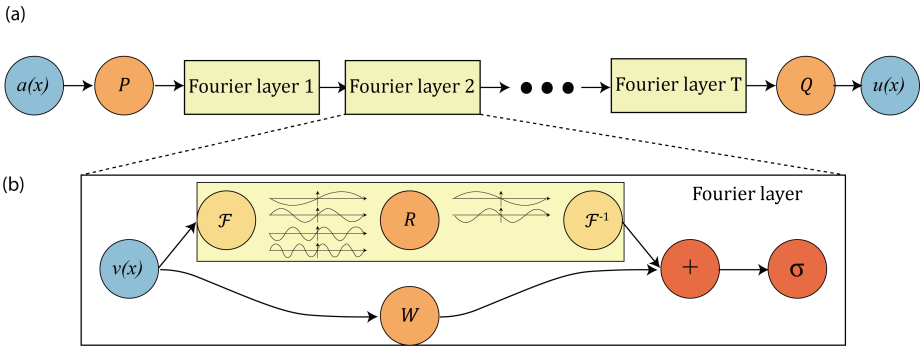


Figure 4: Full architecture of an FNO from [9]. Step 1.) Lift $a(x) \to P$, for example with a 1x1 convolutional layer with stacks of 2D input parameters and states. Step 2.) Apply layers of integral operators and activation functions. Step 3.) Project back down to discrete samples $Q(x) \to u(x)$, where $Q(x)$ is a neural network. The Fourier layers copy the input $v(x)$ in two branches. Top branch applies the Fouier transform $\mathcal{F}$, follow by a linear transform on the lower modes (filtering out the higher modes), and completed by an inverse Fourier transfer $\mathcal{F}^{-1}$. The bottom branch applies a local linear transform $\mathcal{W}$.

For FNOs in particular, the Fourier Integral Operator is defined by [9] as:

$$(\mathcal{K}(a;\phi)v_t)(x) = \mathcal{F}^{-1}(R_\phi \cdot (F_{v_t})(x), \forall x \in D \tag{44}$$

Where ϕ are learnable parameters. Which matches the architecture in figure 4. Thus a full description of the network can be written as it is in [1]:

$$u_{net}(\Phi;\theta) = \mathcal{Q}\circ\sigma(W_L+\mathcal{K}_L)\circ...\circ\sigma(W_1+\mathcal{K}_1)\circ\mathcal{P}(\Phi), \quad \Phi = \{v_t(x); \forall x \in D\} \tag{45}$$

Let's look at another example from Nvidia Modulus [1], solving Darcy Flow with Fourier Neural Operator. The Darcy PDE is:

$$-\nabla \cdot (k(\mathbf{x})\nabla u(\mathbf{x})) = f(\mathbf{x}), \quad \mathbf{x} \in D, \tag{46}$$

Where $u(\mathbf{x})$ is the flow pressure, $k(\mathbf{x})$ is the permiability, and $f(\cdot)$ is the forcing function. The boundary equation $u(\mathbf{x}), \mathbf{x} \in \partial D$ will be used for this example. $\mathbf{U}$ and $\mathbf{K}$ will be discretized as 2D matrices (images essentially). The problem develops a map between the permeability field and the pressure field $\mathbf{K} \to \mathbf{U}$. This particular problem was solved with a 2D FNO with 4 Fourier convolution layer with 32 features each. There are no boundary conditions for this data-driven problem. However the supervised training condition can be treated as a constraint. The model is able to predict pressure fields for permeability fields it has not seen previously.

Neural operators are able to learn physical relationships generally, with arbitrary space parameterizations. This makes them far more generally applicable than PINNs. They are able to produce results for geometry or fields not in their training set.

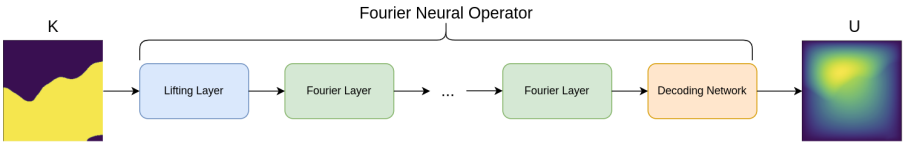


Figure 5: 2D Darcy flow with Fourier Neural Operator from [1].

2.1.3 Neural Operators: Adaptive Fourier Neural Operator

[NEED TO WRITE]

2.2 Physics-Informed Neural Networks

Physics-Informed Neural Networks include a conservation equation, in the form of a PDE, in their loss function. When the conservation term is zero, there is zero error contributed by that part of the error term.

$$\mathbf{L}_n = \frac{1}{J_L}\sum_{j=0}^{J_L-1}(\hat{u}_{n,j} - u_{n,j})^2 + \frac{1}{J_f}\sum_{j=0}^{J_f-1}(f())^2 \tag{47}$$

$f() = 0$ when the physical quantity of concern is conserved. PINNs take advantage of automatic differentiation to compute the necessary partial derivatives. Two passed through the network are required for second derivatives. Taking the Burger's equation example with Dirichlet boundary conditions from [6]:

$$u_t + uu_x - \frac{0.01}{\pi}u_{xx} = 0, x \in [-1, 1], t \in [0, 1], \tag{48}$$

$$u(0, x) = -\sin(\pi x), \tag{49}$$

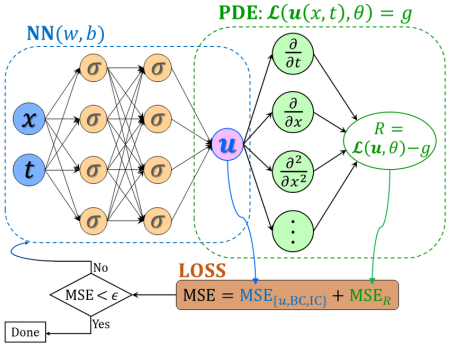
$$u(t, -1) = u(t, 1) = 0 \tag{50}$$

Then f(t,x) can be defined as:

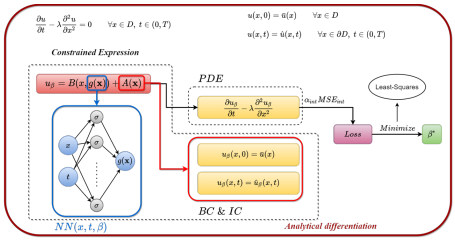
$$f := u_t + uu_x - \frac{0.01}{\pi}u_{xx} \tag{51}$$

There are several ways to include boundary equations, including as part of the loss equation such as in Figures [6a] and [6b]. In Nvidia Modulus [1] boundary conditions and initial conditions can be implemented as data constraints (i.e. as labels for a given u(x,t)) and loss for them is handled in a similar manner to data loss.

Let's look at a simple example in Nvidia's Modulus, the lid driven cavity [1]. In this example, incompressible fluid is in a container whose lid moves in the +x direction and the other walls do not move. For this problem, continuity



(a) Diagram of PINN from [12].



(b) Xtfc of PINN schematic from wikipedia.

and momentum equations are needed:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{52}$$

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{\partial p}{\partial x} - V(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) = 0 \tag{53}$$

$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{\partial p}{\partial x} - V(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}) = 0 \tag{54}$$

Which creates the contraints:

$$\mathbf{L} = \frac{V}{N}\sum_{n=0}^{N-1}((0-continuity(x_n,y_n))^2+(0-momentum_x(x_n,y_n))^2+(0-momentum_y(x_n,y_n))^2) \tag{55}$$

Initial conditions and boundary conditions are implemented in Modulus as data contraints, sampled from the problem domain. The interior of the container is constrained to the PDEs. Samples from the top wall have an output labels of $u = 1.0, v = 0.0$. The samples from the other walls have labels $u = 0.0, v = 0.0$. In order to help the neural network learn boundary constraints, the engineer can determine how many samples should come from the boundary and how many should come from the interior. The corner between the top wall and the side walls cause a discontinuity. To help the neural network learn faster, modulus can take a lambda weighting for the boundary conditions. In this case the top wall is modified with $\lambda = (u = 1.0 - 20.0|x|, 1.0)$. This is visualized in Figure 7.

The models trains fairly quickly on a relatively old computer (when making concessions due to limited memory). The tensorboard in Figure 8 shows the results. If you have the compute and memory resources you can compare results to the CFD software package OpenFOAM.

The most obvious use case of PINNs is to learn a sparsely sampled problem. We can consider a sparsely sampled problem as one where the order of magnitude of the number of observations is less than the order of magnitude of the number

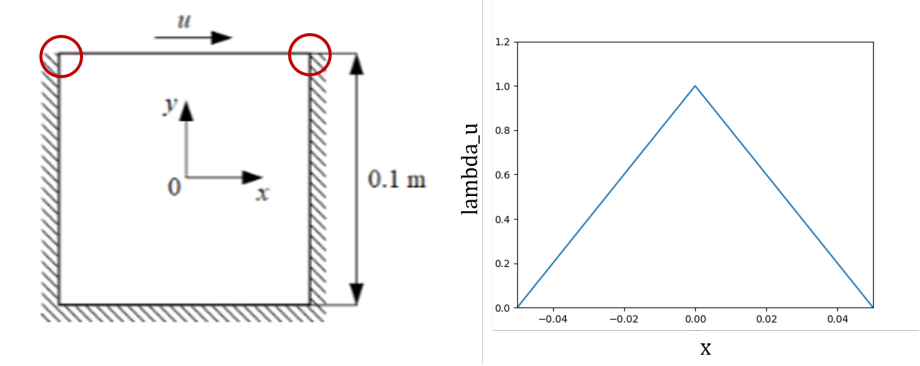


Figure 7: Visualization of lambda weighting for the top wall boundary constraint in the lid driven cavity example from [1].

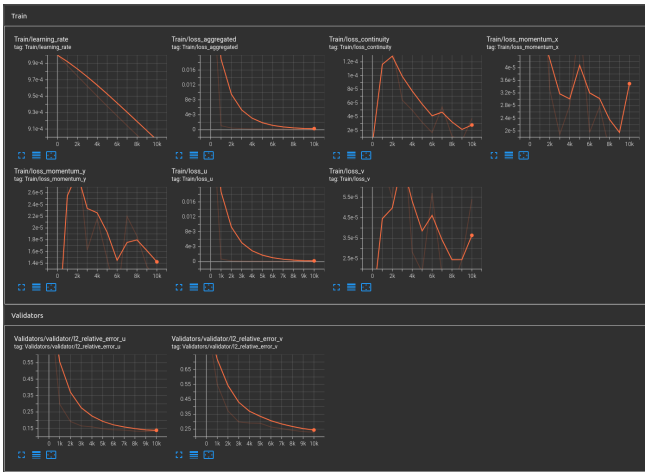


Figure 8: Results from training a neural network on the lid driven cavity example from Modulus.

of parameters. In such a case, including PDE based losses may augment the data and help the network learn the problem. An important advantage of PINNs over traditional solvers is the ability to do solve them over parameterized geometries. [8] notes that PINNs are only useful for learning a single problem. Any spatial dependencies are encoded into the network.

2.2.1 Neural Operators: Physics-Informed Neural Operator

[NEED TO WRITE]

2.3 Transfer Learning from Simulations

Transfer learning involves re-purposing a neural network, trained on one task for another. This can involve replacing and training the final few layers of a neural network or retraining the whole thing. In the case that a high fidelity physical simulation exists, a surrogate would first be trained to approximate the simulation. Then the surrogate can be further trained on real data.

As an example, we have a signal power loss application. In order to enable transfer learning, we had to implement the state-of-the-art numerical approximation. We enabled parallel evaluation with CUDA, in order to enable large data generation quickly. We implement the functions as described in [5]:

$$Loss = f(\mathbf{h}_0, \mathbf{h}_1, \mathbf{h}_2, \mathbf{d}_1, \mathbf{d}_2, \mathbf{freq}) \tag{56}$$

Where the inputs are all vectors to enable parallel computation of many example. Every entry i is an example.

$$Loss_i = FSPL_i + LOS_i + NLOS_i \tag{57}$$

Where FSPL is the loss that occurs in the Fresnel Zone, LOS is the loss that occurs when the Freznel Zone is obstructed but the line-of-sight remains unobstructed, and NLOS is the loss that occurs when the line-of-sight is obstructed. The losses are defined as:

$$FSPL_i = \begin{cases} 20\log(d_{1,i} + d_{2,i}) + 20\log(f_{GHz}) + 92.45dB & \text{if } v \leq -1.0 \\ 0 & \text{if } otherwise \end{cases}$$

$$LOS_i = \begin{cases} 6.0(1.0 - \frac{C_{obs}}{R_{FR}})dB & \text{if } 0.0 < v < 1.0 \\ 0 & \text{if } otherwise \end{cases}$$

$$NLOS_i = \begin{cases} 6.9 + 20\log(\sqrt{(v-0.1)^2 + 1.0} + v - 0.1)dB & \text{if } v \geq 0.0 \\ 0 & \text{if } otherwise \end{cases}$$

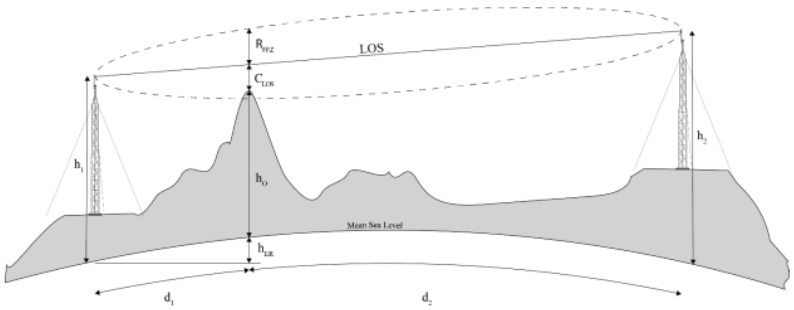


Figure 9: Signal Loss graphic from [5].

Where,

$$R_{FR,i} = 0.6(547.533\sqrt{\frac{d_{1,i}d_{2,i}}{f_{MHz}(d_{1,i}+d_{2,i})}}) \tag{58}$$

$$C_{obs,i} = h_i = h_{0,i} + h_{ER,i} - h_{1,i} - \frac{(h_{2,1}-h_{1,i})}{(d_{1,i}+d_{2,i})}d_{1,i} \tag{59}$$

$$v_i = h_i\sqrt{\frac{2(d_{1,i}+d_{2,i})}{\lambda d_{1,i}d_{2,i}}} \tag{60}$$

$$h_{ER,i} = \frac{d_{1,i}d_{2,i}}{16.944} \tag{61}$$

$$\lambda = \frac{299792458}{freq_{Hz}} \tag{62}$$

We can use this simulation to train a neural network to approximate it's results. We can then use transfer learning on a limited dataset. Such a training methodology makes use of prior physical knowledge and available data.

2.4 Validation with High Fidelity Simulations

Another way to take advantage of high fidelity simulations is to use them as part of validation. This technique can effectively use the limited, expensive data from these simulations as part of hyper-parameter tuning. This induces a bias in the surrogate model towards the results of the simulation.

2.5 Neural Adapter

A neural adapter, as described by [13] is similar to transfer learning. Instead of retraining a neural network or adding layers to it, a new neural network is trained to adapt a previously trained model to the new problem domian. Figure 10 shows the general architecture of the problem. Not only could this be down

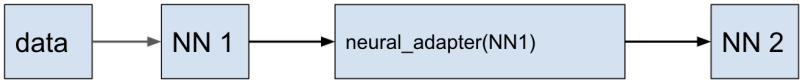


Figure 10: Neural adapter architecture by [13]

.

with neural networks, but the results from other models could be corrected by a neural network to better fit the problem domain. The results of existing physical models could be adjusted to better fit real data. The problem would be of the form:

$$argmin_\theta \frac{1}{2} \sum_{k=0}^{K_L-1} (N(S(\mathbf{x}_k), \boldsymbol{\theta}) - \hat{\mathbf{y}}_k)^2 \tag{63}$$

Where $S(\mathbf{x}_k)$ is the output from the simulation.

2.6 Differentiable Simulations

[NEED TO WRITE]

2.7 Inverse Problems

[NEED TO WRITE]

2.8 Implicit Model Representations

2.8.1 Neural Radiance Fields

[NEED TO WRITE]

2.9 Uncertainty in Physics-Based Neural Networks

[NEED TO WRITE]

3 Packages

Physics-based neural networks can be implemented fairly easily in any neural network framework, as [11] does with PyTorch. In fact, since none of the core techniques are black boxes, any of these methods could be re-implemented in a fast language if necessary. However, there are some packages available that make quick development of physics-based neural networks easier.

1.) Nvidia Modulus, [1] is easy to use, well documented, and implements much of the latest research in physic-bases neural networks. Works with PyTorch.

2.) NeuralPDE.jl, [13] is part of SciML. And is a Julia language package with PINNs and deep Forward-Backwards Stochastic Differential Equations solvers.

3.) SciAnn: Neural Networks for Scientific Computations, [2], a package using TensorFlow and Keras.

4.) DeepXDE, [4], a library containing algorithms for PINNs and DeepONets. Compatible with Tensorflow and PyTorch.

References

[1] Modulus user guide.

[2] Ehsan Haghighat and Ruben 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*, 373:113552, 2021.

[3] Adam Paszkem, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, Adam Lerer. Automatic differentiation in pytorch. In *32st Conference on Neural Processing Systems (NIPS 2017)*, 2017.

[4] Lu Lu, Xuhui Meng, Zhiping Mao, and George Em Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, 63(1):208–228, 2021.

[5] Nicole Patterson. Signal propagation equations.

[6] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part 1): Data-driven solutions of nonlinear partial differential equations.

[7] Atilim G"unes Bayden, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey.

[8] Nikola Kovachki,* Zongyi Li,* Burigede Liu, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar.

[9] Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar,. Fourier neural operator for parametric partial differential equations.

[10] Sergios Theodoridis. *Machine Learning: A Bayesian and Optimization Perspective.*

[11] N. Thuerey, P. Hall, M. Mueller, P. Schnell, F. Trost, K. Um. *Physics-based Deep Learning.*

[12] Xuhui Meng, Zhen Li, Dongkun Zhang, and George Em Karniadakis. Ppinn: Parareal physics-informed neural network for time-dependent pdes.

[13] Kirill Zubov, Zoe McCarthy, Yingbo Ma, Francesco Calisto, Valerio Pagliarino, Simone Azeglio, Luca Bottero, Emmanuel Luján, Valentin Sulzer, Ashutosh Bharambe, et al. Neuralpde: Automating physics-informed neural networks (pinns) with error approximations. *arXiv preprint arXiv:2107.09443*, 2021.