Open in app ↗ Sign up Sign in

Medium Search

Physics-informed Neural Networks: a simple tutorial with PyTorch

Make your neural networks better in low-data regimes by regularising with differential equations

Theo Wolf · Follow

8 min read · Apr 13, 2023

Listen Share

"A thermometer inside a coffee cup with a dark blue background" by Dall·E.

I assume that you are already familiar with neural networks, mathematical notation and calculus throughout this article.

Physics-Informed Neural Networks (PINNs) [1] are all the rage right now (or at the very least they are on my LinkedIn). But what are they? In this article, I will attempt to motivate these types of networks and then present a straightforward implementation with PyTorch. Most of the implementations currently out there are either in Tensorflow or are have been overcomplicated.

Neural Networks aren't a silver bullet for all situations

I don't think I need to explain how important neural networks are to today's changing world. ChatGPT is just a big neural network with billions of parameters. The deep learning framework is incredibly effective at learning dependencies between data and are extremely flexible as universal function approximators.

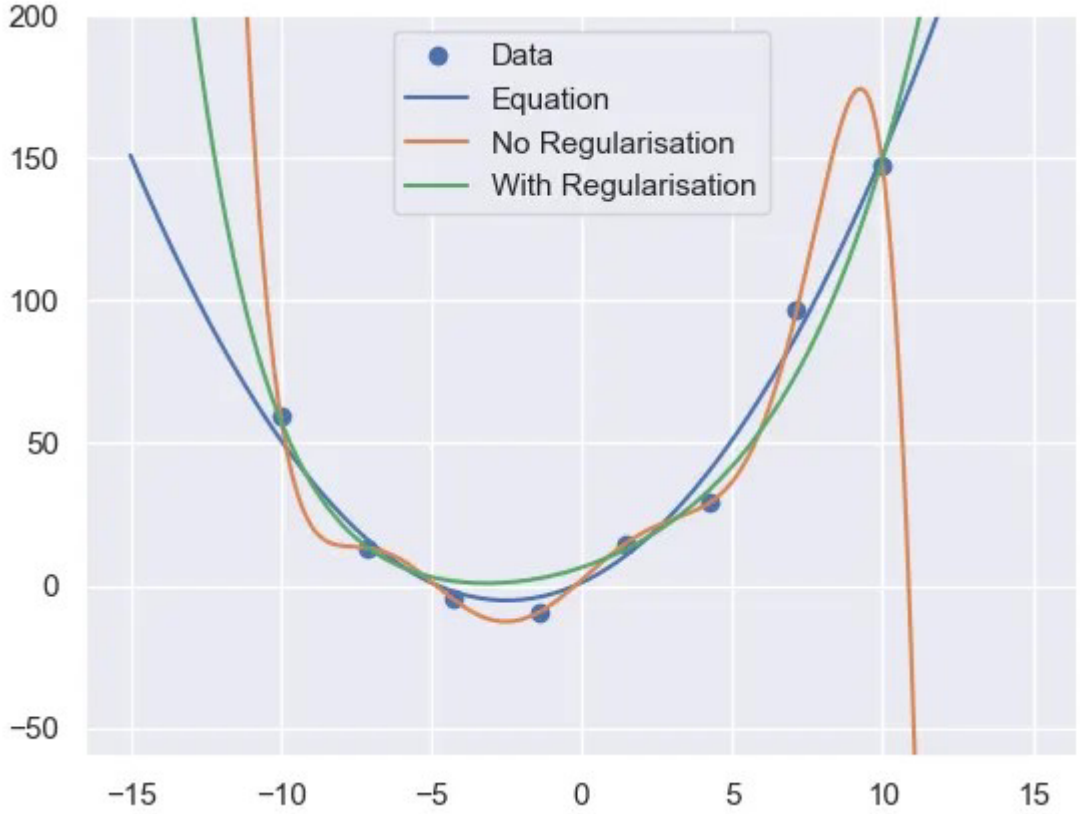
Their greatest strength is also their greatest weakness. Because neural networks are so good at approximating functions, they are very good at overfitting to the training set. As a result, for them to be effective at generalising and not just learning the training set, you need lots and lots of data as well as some clever tricks such as batching.

It is quite common in a lot of fields that collecting data is just very difficult, this is quite a headache for some ML engineers (i.e. me). When there is not enough data, using complex models such as neural networks can be risky, as there is a risk of overfitting to the small amount of data that you have and not being able to test your model accurately.

A common approach to avoid overfitting is regularisation. You can do regularisation with neural networks just like you would do it with linear regression,

$$Loss_{reg} = \frac{1}{N}\sum_{i}^{N}(f(x_i|\theta) - y_i)^2 + \lambda||\theta||_2^2$$

By minimising the weights, we ensure that no weight is absurdly high, thus giving a less clunky function. This helps to fit to data without the model "looking" bad.



Here, I generated some data using a quadratic equation and some noise. I then created some polynomial features and fitted a linear regression model with and without regularisation. Image by author.

In this plot, we can see how **not** having a regulariser has made our model pretty funky. Even in this simple example, we can see how overfitting to the data creates a model that is unreliable as soon as we leave the span of data we have.

Theory

Physics-informed priors as described in [1] are a way to regularise a neural network, but that are a bit more advanced. Essentially, they help the neural network function have the right **shape.** To do this, we imbed the network with information in the form of a differential equation. When there is little data available, being able to imbed additional information that isn't data into the network is extremely powerful.

How do we imbed this information? In the very same way we do regularisation, with a loss. The most obvious thing to do is to take the Mean Squared Error of the equation that describes the data we are seeing.

Say we have a differential equation *g(x, y)* = 0, some data *{x_j, y_j}* and a neural network *f(x | θ)* that approximates *y*. For a PINN, we would get a loss function that looks like the following,

$$Loss_{PINN} = \underbrace{\frac{1}{N}\sum_{j}^{N} ||f(x_j|\theta) - y_j||_2^2}_{\text{Data loss}} + \lambda \underbrace{\frac{1}{M}\sum_{i}^{M} ||g(x_i, f(x_i, |\theta))||_2^2}_{\text{Physics loss}}$$

Where *x_i* are *collocation* points. These can be any value we want them to be, usually you would want them to be in the range of values we are interested in. The *x_j* and *y_j* are our data. We can also add a parameter controlling the relative strength of the data loss function and the physics loss function, here we use λ. And then just train as you would any other neural network.

And we're done? That's not that hard, but I'm sure you have some questions, which I hope the example will help with.

Some caveats

The PINN framework requires some equation that relates to your data. For most datasets, you won't have any prior knowledge or exact equation of how the outputs and inputs interact (e.g. image labelling datasets). This framework is (mostly) reserved for data that measures real-world physical phenomena. Usually, this data is noisy and hard to come by. Both of these drawbacks are helped by PINNs.

Also, it was shown that using differential equations that were very complex can actually hurt the optimisation of the NN parameters. This is because the loss landscape gets too "bumpy" and gradient descent gets stuck [3].

Example: a cooling coffee cup

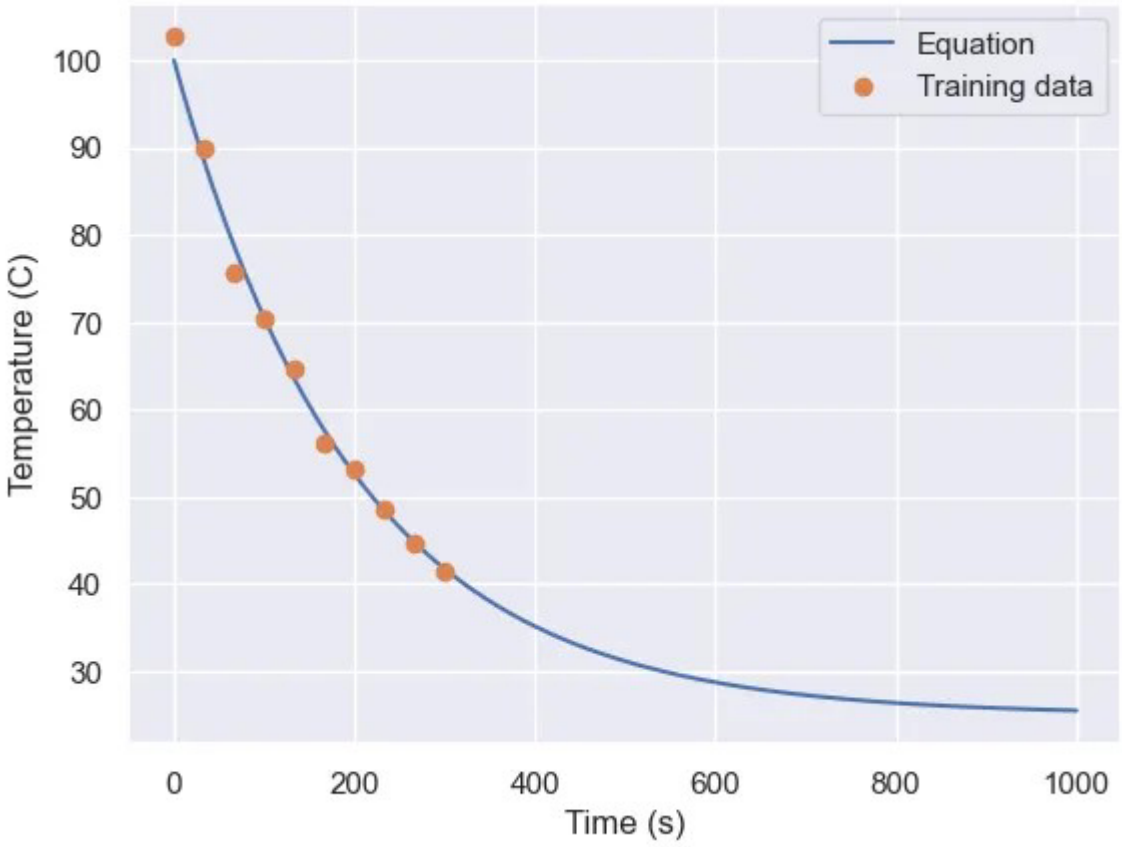
Let's say I have data on some physical process, for simplicity we will go with my cup of coffee, which is cooling. This obeys a simple law of physics,

$$\frac{dT(t)}{dt} = r(T_{env} - T(t))$$
$$T(t) : \text{temperature}$$
$$T_{env} : \text{temperature of the environment}$$
$$r : \text{cooling rate}$$

Newton's law of cooling in time only, Image by author.

Let's say that my coffee that was boiling hot is cooling over the course of 15 minutes or so, my home is currently at 25°C. I'll use a value for the cooling rate of 0.005. I don't know when my coffee will be at room temperature, but I don't want to wait to find out.

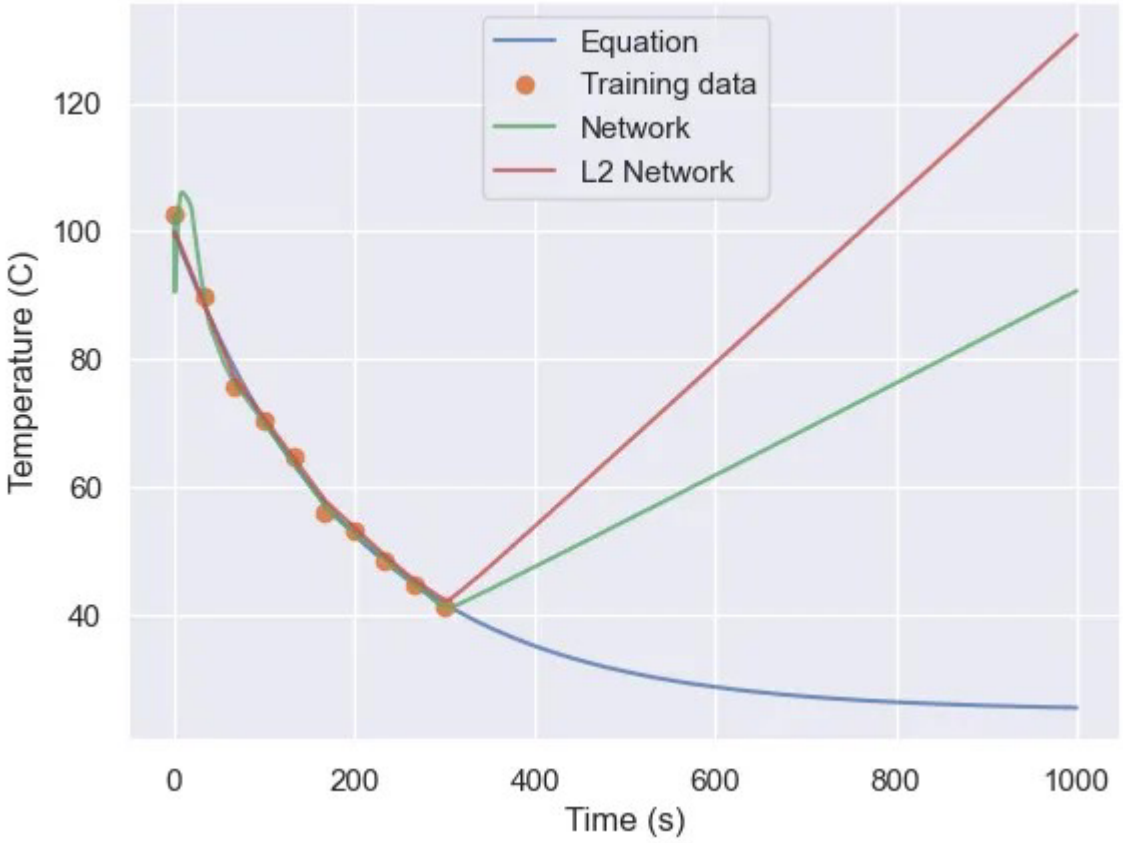
What does this look like? Let's plot and it and make some training data from this equation. We make 10 training points in the first 5 minutes.



The temperature of a cup of coffee versus time. Data samples are noisy measurements of the temperature. Image by author.

Let's train a vanilla linear network (with ReLU activation) and then an L2

regularised network on this data.



Functions learned by a linear network and an L2 linear network on this data. Image by author.

Both networks become completely inaccurate as we leave the realm of the training data, which is to be expected: they have no information where there is no data. We also have some funkiness in the vanilla network in between the first and second point.

Let's make a PINN instead

If we have a neural network *f(t|θ)* that predicts the temperature of the cup *T* given time *t* then we can construct a physics loss of this data:

$$g(t, T) = \frac{dT(t)}{dt} - r(T_{env} - T(t)) = 0$$

$$g(t, f(t|\theta)) = \frac{df(t|\theta)}{dt} - r(T_{env} - f(t|\theta))$$

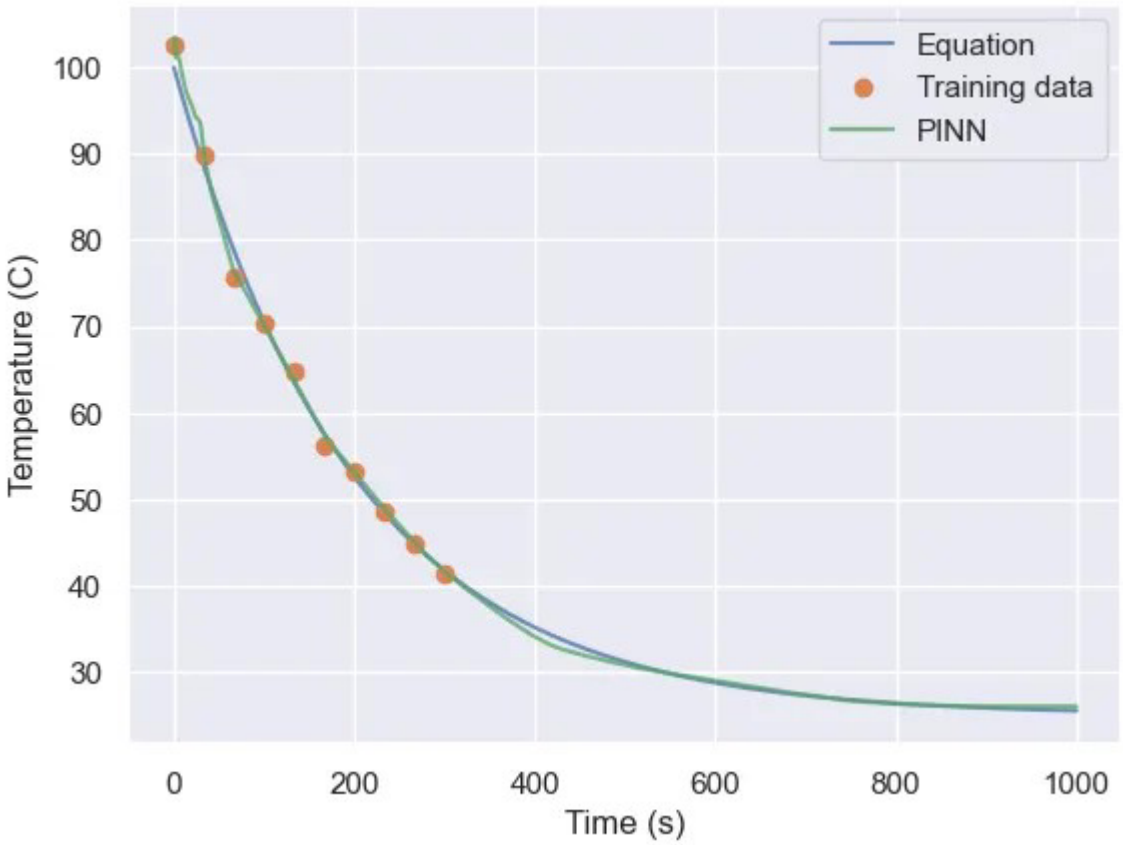
$$Loss_{PINN} = \underbrace{\frac{1}{10}\sum_{j}^{10}(f(t_j|\theta) - T_j)^2}_{\text{data loss}} + \lambda \underbrace{\frac{1}{M}\sum_{i}^{M}\left(\frac{df(t_i|\theta)}{dt_i} - r(T_{env} - f(t_i|\theta))\right)^2}_{\text{physics loss}}$$

Some of you might be wondering how to (in practice) take the derivative of your neural network, *torch.autograd* module has a handy function called *grad()* which does exactly that (you can even take higher order derivatives). Just ensure that *create_graph* is set to True in order to create a new computational graph without reusing an old one [4].

```
def grad(outputs, inputs):
    """Computes the partial derivative of
    an output with respect to an input."""
    return torch.autograd.grad(
        outputs,
        inputs,
        grad_outputs=torch.ones_like(outputs),
        create_graph=True
    )

def physics_loss(model: torch.nn.Module):
    """The physics loss of the model"""
    # make collocation points
    ts = torch.linspace(0, 1000, steps=1000,).view(-1,1).requires_grad_(True
    # run the collocation points through the network
    temps = model(ts)
    # get the gradient
    dT = grad(temps, ts)[0]
    # compute the ODE
    ode = dT - R*(Tenv - temps)
    # MSE of ODE
    return torch.mean(ode**2)
```

Training the network with 1000 collocation points:



PINN trained on the temperature data. Image by author.

The fit is very nice, we see that the model accurately predicts the temperature beyond the data. Amazing!

Or is it?

Some of you might have noticed that I have somewhat cheated. I have given the answer of what the equation should look like to the network. I could've just solved it myself.

The equation we are working with is actually a very simple differential equation. It can be solved with some separation of variables and applying boundary conditions (left as an exercise to the reader 😉),

$$T(t) = (T(0) - T_{env})e^{-rt} + T_{env}$$

And there's our function, why did we bother making a neural network at all? We got the answer with some 1st year level calculus.

The explanation is that this was very much a toy example. The differential equation we use here has all the inputs and outputs that we are interested in. In most cases, we would have a network $y = f(t, x \mid \theta)$ and an equation $g(t, y) = 0$, where the differential equation does **not** fully describe the data the network is trained on. I chose this example just for its simplicity and less for its actual use.

Adding flexibility: PINNs for equation discovery

Reality rarely follows physics exactly, an equation such as the one we use here is incredibly simplistic as it doesn't consider lots of other effects. For example, the cooling rate will be affected by the thickness and material of the mug I use.

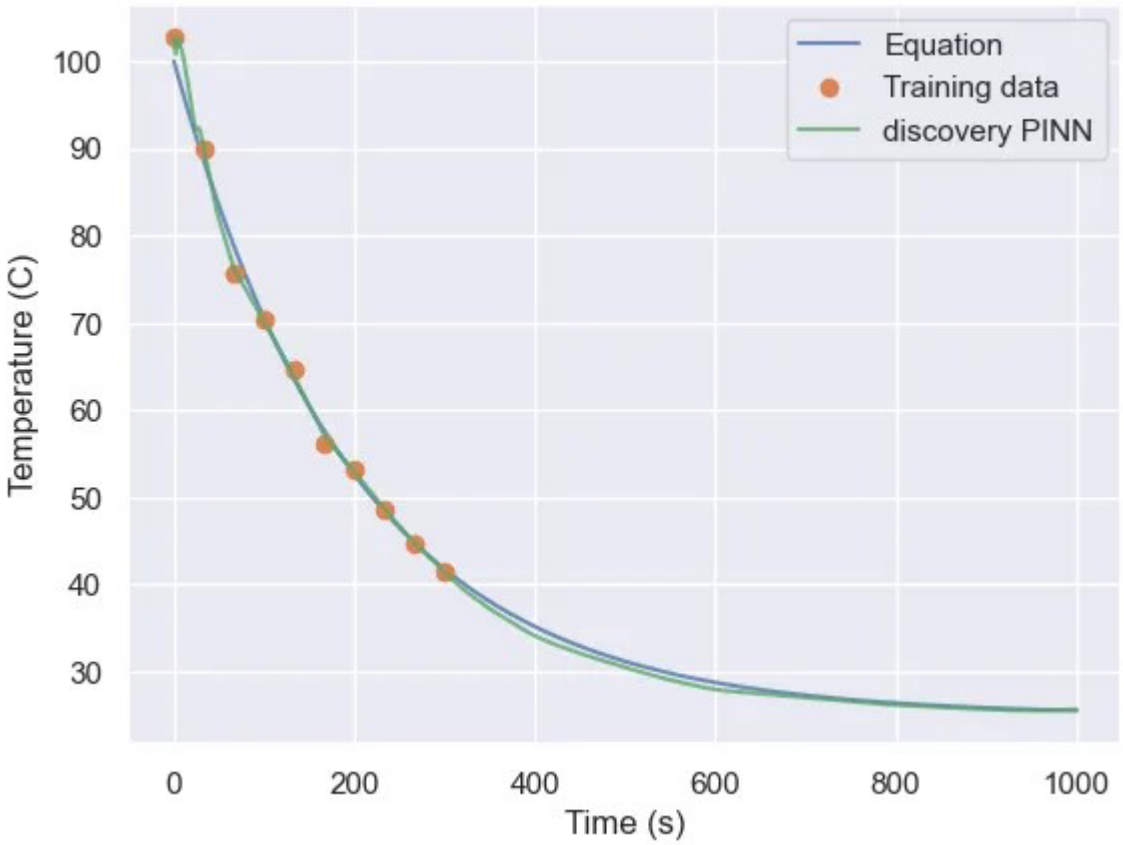
Say that one of the parameters of our differential equation is unknown. In this case, we take the cooling rate, the one which is the hardest to measure. Our differential equation is then $g(t, T \mid r) = 0$ where r is unknown. Thanks to PyTorch, all we need to do is just one small change: add r as a differentiable parameter. This is very easy, just add the variable in your network initialisation, PyTorch will do the rest! And make sure you appropriately change your physics loss.

```
class Net(nn.Module):
  def __init__(self, *args):
    ...
    # make r a differentiable parameter included in self.parameters()
    self.r = nn.Parameter(data=torch.tensor([0.]))
    ...

def physics_loss_discovery(model: torch.nn.Module):
    ts = torch.linspace(0, 1000, steps=1000,).view(-1,1).requires_grad_(True
    temps = model(ts)
    dT = grad(temps, ts)[0]
    # use the differentiable parameter instead
    pde = model.r * (Tenv - temps) - dT

    return torch.mean(pde**2)
```

Training this network...



PINN trained on the same data but with unknown cooling rate parameter. Image by author.

Nice! Starting from a value of 0.0, our network finds a cooling rate of 0.0051 for this data. The true value was 0.0050, so we got pretty close with just 10 data points. This can be incredibly useful when dealing with physical systems where you know the general relationships between variables but not the parameter values.

Conclusion

PINNs are a way to regularise your network **if** you have knowledge about how some of the inputs and outputs interact. In scenarios where little data is available, but you have some prior knowledge on the data, this can be very useful to learn other dependencies in the data. It's true that if you have bucket loads of data, then it doesn't really matter, as the dependencies are imbedded in the data and the neural network will eventually figure them out.

Link to Code

Go check out the GitHub repo, where all the code is there: github.com/TheodoreWolf/pinns. Feel free to give feedback and let me know if something wasn't clear!

References

[1] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis, *Physics Informed Deep Learning*, https://maziarraissi.github.io/PINNs/

[2] Ben Mosley, *So, what is a physics-informed neural network?*https://benmoseley.blog/my-research/so-what-is-a-physics-informed-neural-network/

[3] Krishnapriyan, A. S. *Characterizing possible failure modes in physics-informed neural networks,* NeurIPS 2021, https://arxiv.org/abs/2109.01050

[4] PyTorch, *Computational graphs,* https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

Machine Learning | Physics | Neural Networks | Pytorch

Follow

Written by Theo Wolf

874 Followers · 19 Following

DPhil @ Oxford University, passionate about sustainability, physics, chemistry and AI, theodorewolf.github.io/

Responses (4)

What are your thoughts?

Respond

Jorgecardete
about 1 year ago

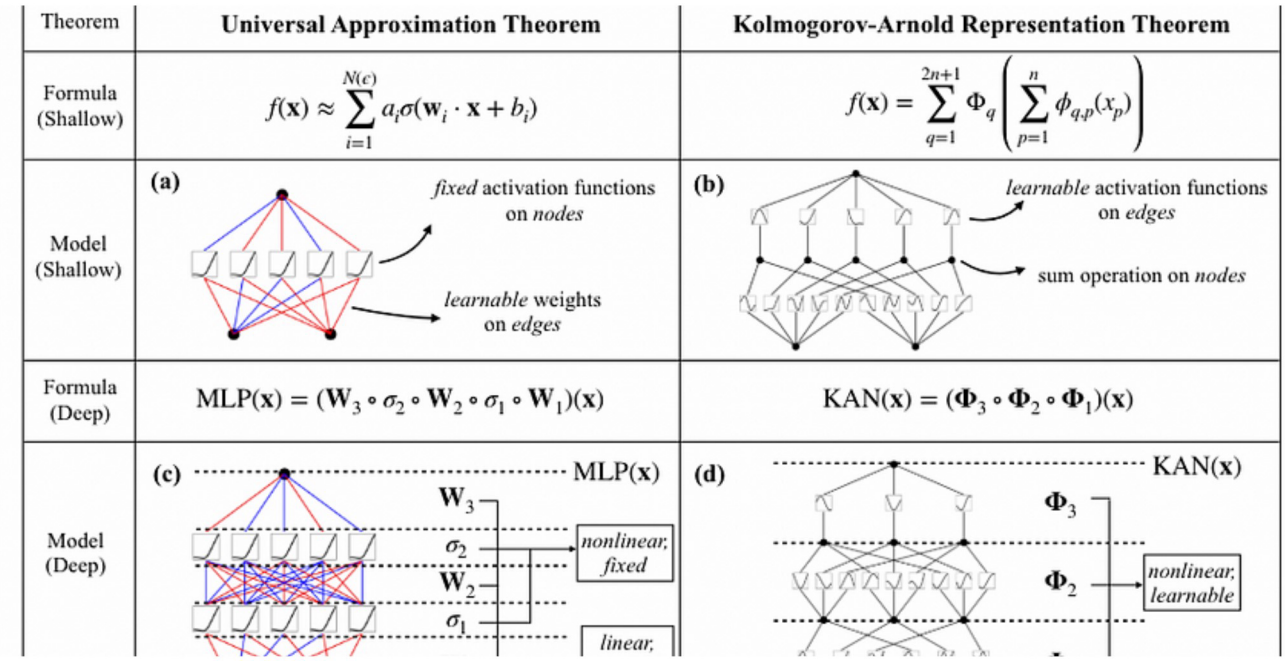
Great article!

3

Reply

See all responses

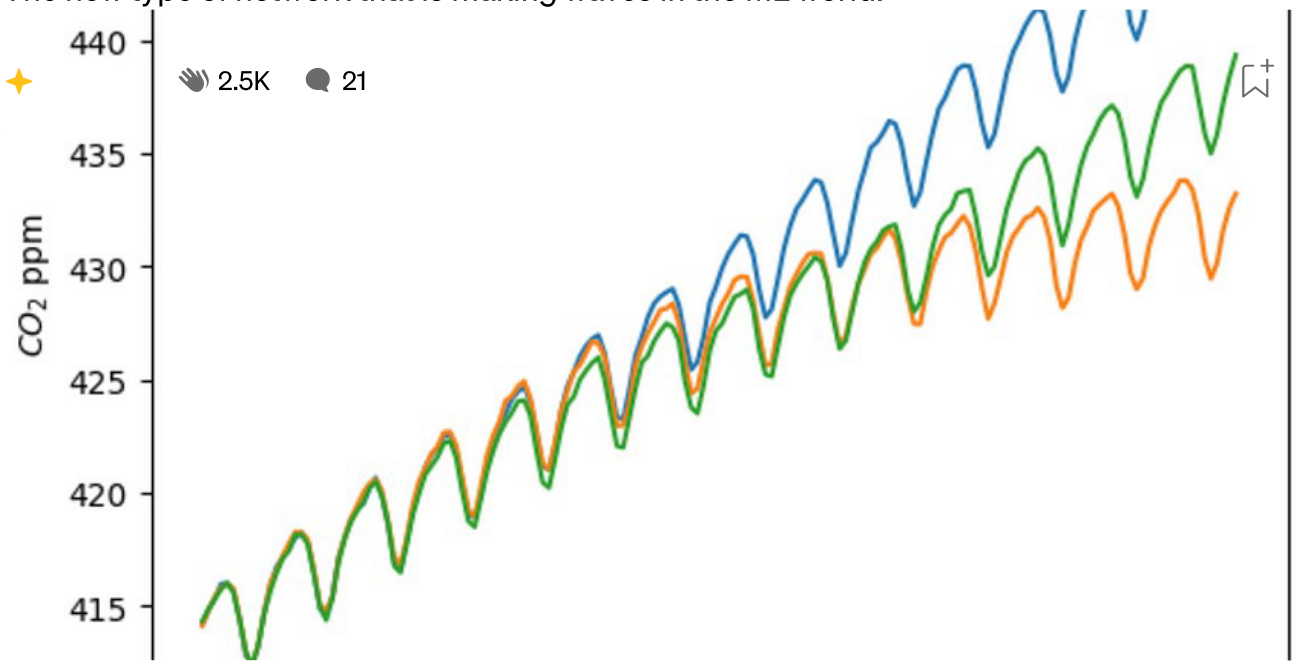
More from Theo Wolf



In Towards Data Science by Theo Wolf

Kolmogorov-Arnold Networks: the latest advance in Neural Networks, simply explained

The new type of network that is making waves in the ML world.



In Towards Data Science by Theo Wolf

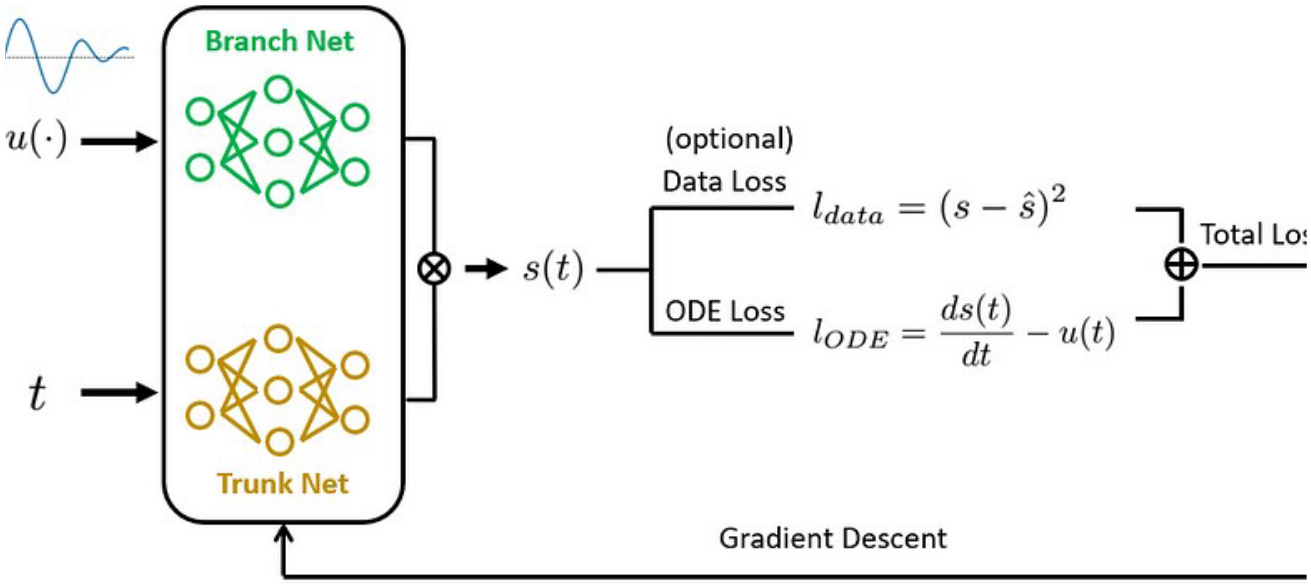
Gaussian Processes from Scratch

Gain a deeper understanding of Gaussian processes by implementing them with only NumPy

Jan 7 651 4

See all from Theo Wolf

Recommended from Medium

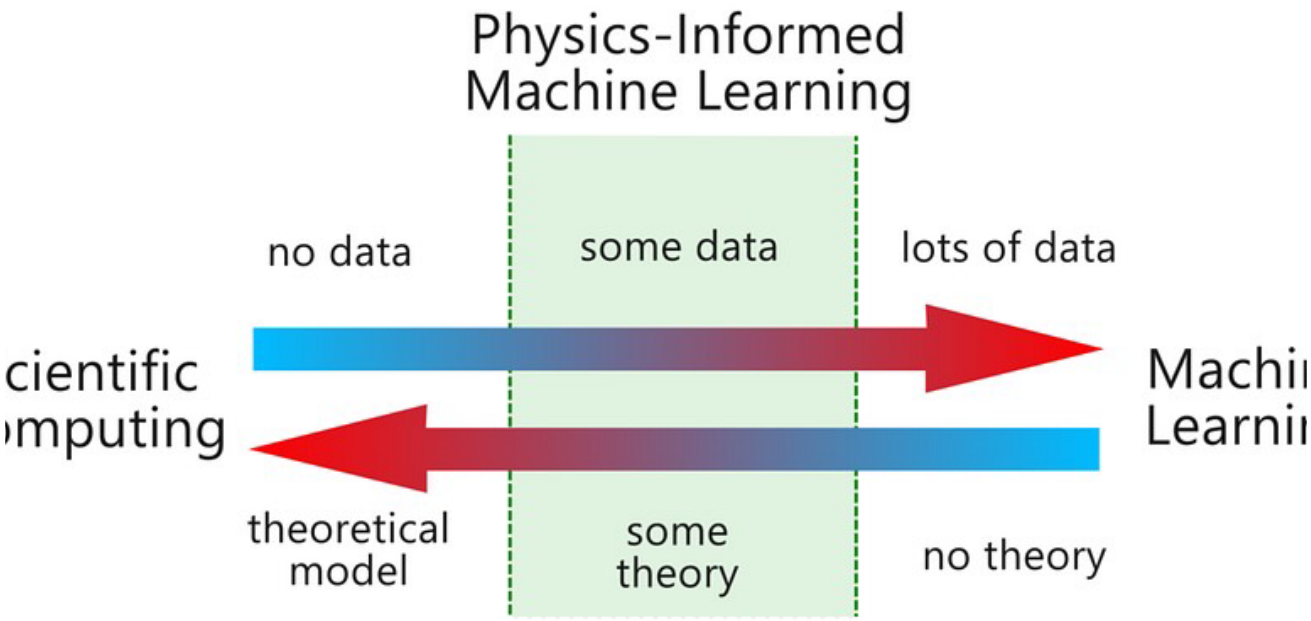


In Towards Data Science by Shuai Guo

Operator Learning via Physics-Informed DeepONet: Let's Implement It From Scratch

A deep dive into the DeepONets, physics-informed neural networks, and physics-informed DeepONets

Jul 7, 2023 450 4



Joris C.

Physics-Informed Machine Learning — PIML

A Primer on How to Combine Machine Learning and Physics

Sep 13 23

Lists

Predictive Modeling w/ Python

20 stories · 1700 saves

Practical Guides to Machine Learning

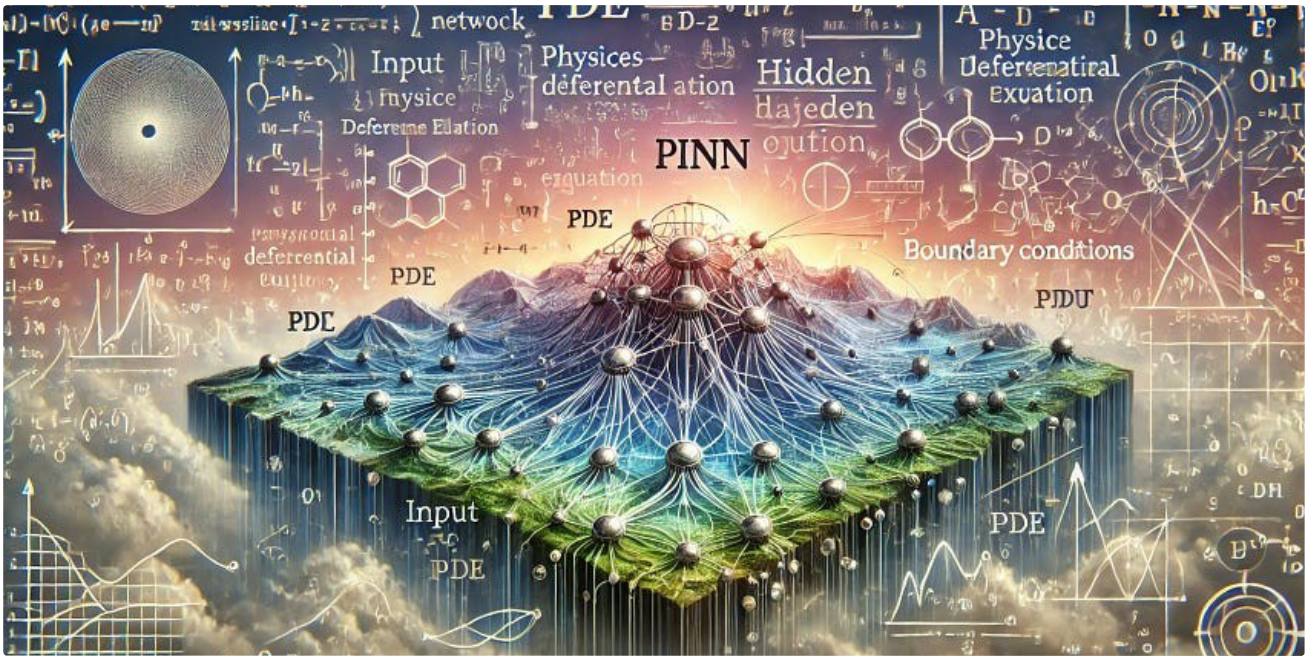
10 stories · 2068 saves

Natural Language Processing

1842 stories · 1466 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 518 saves



In Tech Spectrum by Aarafat Islam

Physics-Informed Neural Networks (PINNs)

How Can Physics and Machine Learning Come Together?

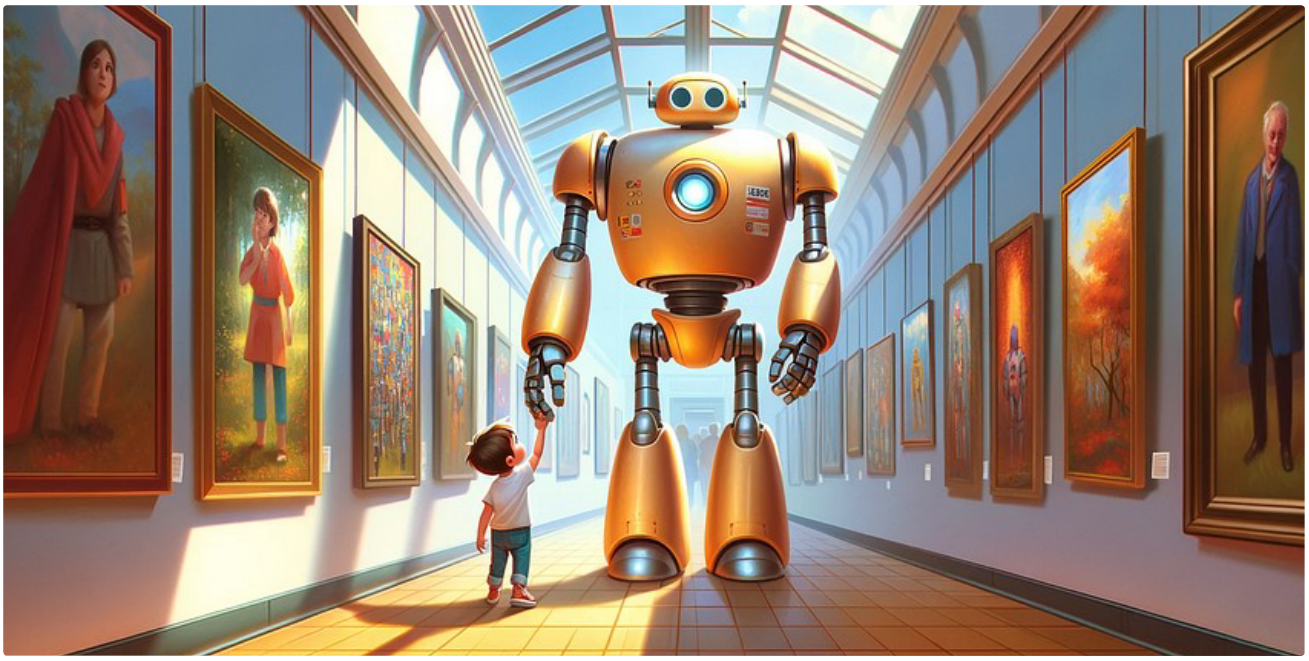
Nov 8 47

Md Junayed Hasan

How a Cup of Coffee Helped Me Understand Physics-Informed Neural Networks (PINN)

Let's talk about coffee. Specifically, how long it takes for that piping hot cup to reach a drinkable temperature. Now, imagine if you...

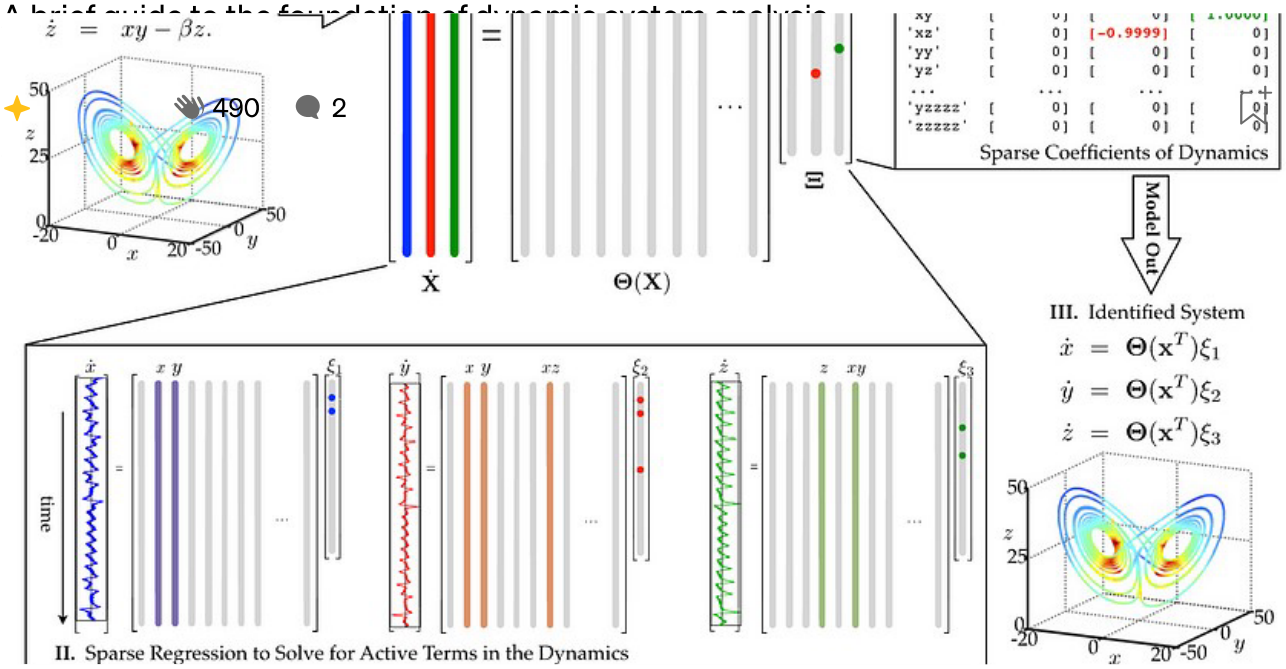
Jun 26 3



In The Deep Hub by Jorgecardete

Gentle introduction to State Space Models

A brief guide to the foundation of dynamic system analysis



Xinyu Chen (陈新宇)

Interpretable Convolutional Kernels for Time Series

Addressing the Problem of Convolutional Kernel Learning in Univariate, Multivariate, and Multidimensional Time Series Data, Crucial for...

Oct 27 6 1

See more recommendations