Machine learning solutions to PDEs

Pinak Mandal

International Centre for Theoretical Sciences - Tata Institute of Fundamental Research

February 2023

Not all sunshine and rainbows :(

Before going into the discussion let's admit one thing first. Machine learning is not a cure-all for every possible problem. If we are sticking to just the topic of solving PDEs there can be many theoretical and practical limitations. A few of them are,

1. Coarse structures in the solutions may be captured well by ML but finer structures might be out of reach.
2. Might be slower or inefficient for lower dimensional problems.
3. Hardwares (GPUs) are often optimized for working with single precision or float32 numbers which is not enough for scientific computing.
4. Standard loss functions are non-convex and hence you might never reach the solution through vanilla gradient descent even though universal approximation theorems say "there exists a neural network close to the solution".

Why machine learning: a bit of sunshine :)

In the upcoming slides we'll talk about the Fokker-Planck equation as an example. If you do a literature study of standard numerical methods for Fokker-Planck equations like finite difference [1], [2] or finite element [3], [4] methods (a comparative study of these methods appear in [5] by Pichler et al) you'll see most works deal with only 2 or 3 dimensional (in space) equations.

There is a non-standard algorithm (hybrid of both kernel and sample based density estimation) that can deal with high-dimensional FP equations with a very specific structure and appears in [6] by Chen and Majda. It does not aim for accurate but only statistically accurate solutions or solutions with the right coarse-grained structures.

Why machine learning: a bit of sunshine :)

In even though people have been solving FP equations numerically for at least 50 years, standard numerics in literature appear mostly for 2-3 dimensional systems because they become incredibly inefficient in higher dimensions, a phenomenon known as the "curse of dimensionality" which refers to the exponential increase in difficulty with the number of dimensions.

Whereas ML for PDEs have barely existed for a decade and there are already examples of 100-dimensional (Allen-Cahn) [7] or even 200-dimensional (Black-Scholes) [8] PDEs being "solved".

Why machine learning: a bit of sunshine :)

1. ML algorithms can work in a mesh-free manner which is the first step towards beating the curse of dimensionality.
2. A broad range of problems can be tackled with a narrow range of principles.
3. Derivatives are calculated with automatic differentiation which in principle can be very accurate (can be useful for avoiding inconsistencies in finite difference methods, eg the failure of product rule, looking at you David Pfefferle).
4. This one is from personal experience, once you're familiar with it, **setting up problems** and playing around (which is what I've been doing at ANU with the help of Bob, Matthew and Zhisong) is lightning fast. Note the emphasis on "setting up", learning the solutions might take some (GPU-)time.

Flavours of "solving": initial conditions

There are many different flavours of "solving" a PDE in ML. Let's look at a few of them.

Expanding your set of initial conditions: Suppose you have already calculated solutions to your problem for some initial conditions with standard numerics. Now you would like to claim you know the solutions for "all" initial conditions. For example consider the 1D Burgers' equation with constant positive ν,

$$u_t + uu_x = \nu u_{xx}, \qquad x \in (0,1),\ t \in (0,1) \tag{1}$$

$$u(0,x) = u_0(x), \qquad x \in (0,1) \tag{2}$$

where $u_0 \in L_{per}((0,1);\mathbb{R})$. So you'd like to find a map $G : L_{per}((0,1);\mathbb{R}) \to H^r_{per}((0,1);\mathbb{R})$ defined by $u_0 \mapsto u(1,\cdot)$ [9].

Flavours of "solving": coefficients

Expanding your set of coefficients: Suppose you have already calculated solutions for some coefficients in your PDE with standard numerics. Now you would like to claim you know the solutions for "all" coefficients. For example consider the 2D Darcy flow

$$\nabla \cdot (a(\mathbf{x})\nabla u(\mathbf{x})) = f(\mathbf{x}), \qquad \mathbf{x} \in (0,1)^2 \tag{3}$$

$$u(\mathbf{x}) = 0, \qquad \mathbf{x} \in \partial(0,1)^2 \tag{4}$$

where $a \in L^\infty((0,1)^2; \mathbb{R}_{>0})$. So you'd like to find a map $G : L^\infty((0,1)^2; \mathbb{R}_{>0})) \to H_0^1((0,1)^2; \mathbb{R}_{>0})$ defined by $a \mapsto u$ [9].

Flavours of "solving": super resolution

Upscaling: Suppose you have already calculated solutions to your equation on lower resolution meshes and you'd like to know the solutions at "all" resolutions. Consider the 2D Navier-Stokes equation for a viscous, incompressible fluid in vorticity form on the unit torus

$$\omega_t + v \cdot \nabla \omega = \nu \Delta \omega + f(\mathbf{x}), \qquad \mathbf{x} \in (0,1)^2, \ t \in (0, T] \tag{5}$$

$$\nabla \cdot v = 0, \qquad t \in [0, T] \tag{6}$$

$$\omega(0, \mathbf{x}) = \omega_0(\mathbf{x}), \qquad \mathbf{x} \in (0,1)^2 \tag{7}$$

where $\omega = \nabla \times v$. Suppose you know the solution on a $20 \times 64 \times 64$ grid and you'd like to know the solution on a $80 \times 256 \times 256$ grid without having to calculate the solution again with standard numerics [9].

Flavours of "solving": moving forward

Extrapolating in time: Suppose you know the vorticity in the Navier-Stokes problem in the last slide up to time $t = 10$ and you would like to know the solution in $[10, T]$ for some $T > 10$ [9]. So you'd like to like to find a map
$G : C([0, 10], H^r_{per}((0, 1)^2, \mathbb{R})) \to C([10, T], H^r_{per}((0, 1)^2, \mathbb{R}))$
defined by $(t, \omega(t, \cdot)) \mapsto (\phi(t), \omega(\phi(t), \cdot))$ where
$\phi : [0, 10] \to [10, T]$ is a bijection.

Flavours of "solving": starting from scratch

You don't know the solution: Suppose you don't have any pre-computed solutions to your problem that looks something like the following,

$$\mathcal{L}u = 0, \qquad \mathbf{x} \in \Omega \tag{8}$$

$$u(\mathbf{x})|_{\partial\Omega} = g(\mathbf{x}) \tag{9}$$

And you'd like to know the solution $u : \mathbb{R}^d \to \mathbb{R}$ in the absence of any pre-generated data. For the rest of the talk, we'll stick to just this one flavour of "solving".

The recipe: write it as an optimization problem

For starting from scratch, the first step is to write the problem as an optimization problem because "learning" in machine learning essentially means solving optimization problems.

A very direct and simple way to do it is [8],

$$\min_{u} \left(\int_{\Omega} (\mathcal{L}u)^2 + \int_{\partial\Omega} (u - g)^2 \right) \tag{10}$$

The recipe: structure of a simple neural net

In the next step we'll model our desired solution $u : \mathbb{R}^d \to \mathbb{R}$ as a neural net.

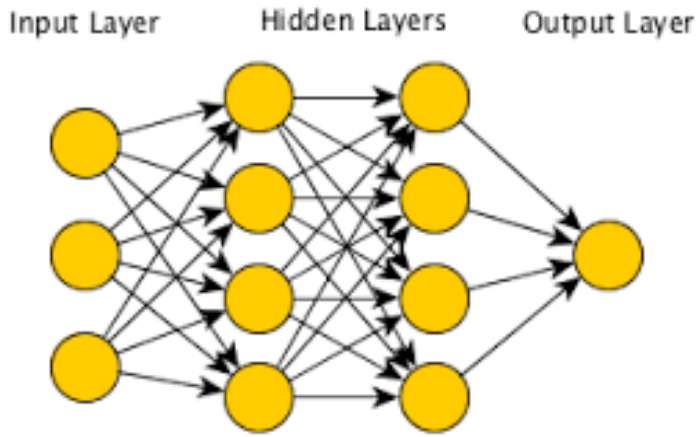


Figure: A simple network

- The network starts with d nodes (leftmost) and ends with a single node (rightmost). The leftmost layer takes values in $\Omega \subset \mathbb{R}^d$.
- There's a complete bipartite directed graph between two consecutive layers.

The recipe: structure of a simple neural net

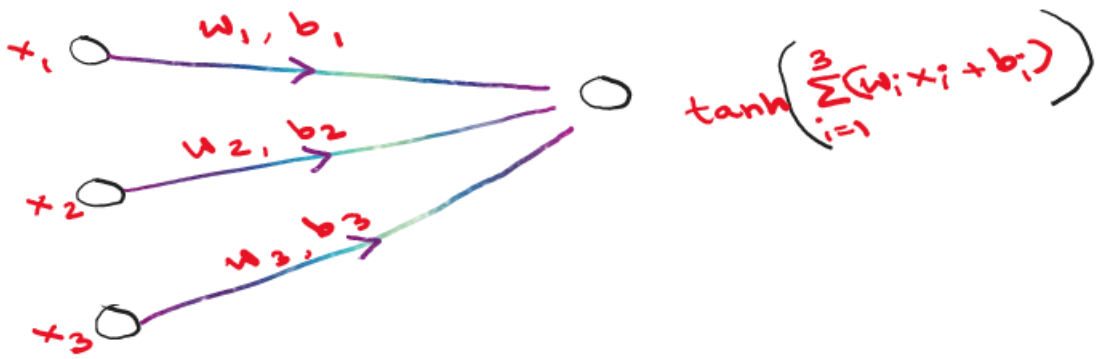


Figure: Flow of values

- Each edge has two parameters, one weight w and one bias b associated with it.
- To evaluate the network at a specific input point, values flow from left to right as depicted above until we reach the rightmost node and that's the output of the network.

The recipe: trainable parameters and architecture

So the network is a just a parametric function $\mathbb{R}^d \to \mathbb{R}$ with a large number of parameters w and b. We'll denote the collection of all these parameters θ.

θ is the variable that we would like to "learn". Everything else about the network e.g. the number of intermediate or hidden layers, the size or the number of nodes in an intermediate layer, the choice of activation function (tanh in this example) and other choices related to the structure of the network are together called the **architecture**. Architecture is not something that is learned but chosen by the parctitioner.

The recipe: replace your infinite search space with a finite one

Let's call our favourite architecture $\mathcal{A}$. We'll denote a network with architecture as $n_\theta^{\mathcal{A}}$. And we would like to find an approximate solution to our optimization problem within the set of functions

$$A := \{n_\theta^{\mathcal{A}} : \theta \in \mathbb{R}^C\} \tag{11}$$

where C is the number of all weights and biases or trainable parameters of the network. Note that our infinite dimensional optimization problem of finding a function has now become the problem of finding a large but finite dimensional vector θ. $C \approx 20000$ in the examples we'll consider here. For brevity, we'll use $n_\theta^{\mathcal{A}}$ and n_θ interchangably.

The recipe: manage the integrals

Why is this not a terrible idea? Because universal approximation theorems [10], [11] say with a complicated enough architecture you'll find a network that is close to any point in your function space (not necessarily that A is dense in your function space). So now the problem becomes,

$$\min_{\theta} \left(\int_{\Omega} (\mathcal{L} n_\theta)^2 + \int_{\partial\Omega} (n_\theta - g)^2 \right) \tag{12}$$

The third step is to turn the integrals into something more manageable. So sample points $\{\mathbf{z}_i\}_{i=1}^N$ from Ω and $\{\mathbf{y}_i\}_{i=1}^M$ from $\partial\Omega$ uniformly and write the problem as,

$$\min_{\theta} \left(\frac{1}{N} \sum_{i=1}^{N} (\mathcal{L} n_\theta(\mathbf{z}_i))^2 + \frac{1}{M} \sum_{i=1}^{M} (n_\theta(\mathbf{y}_i) - g(\mathbf{y}_i))^2 \right) \tag{13}$$

The recipe: perform gradient descent w.r.t θ

The objective function in the last problem above is called a loss function. You will usually include a constant β_0 in the loss function akin to a typical constrained optimization problem,

$$\min_{\theta} \left(\frac{1}{N} \sum_{i=1}^{N} (\mathcal{L} n_\theta(\mathbf{z}_i))^2 + \frac{\beta_0}{M} \sum_{i=1}^{M} (n_\theta(\mathbf{y}_i) - g(\mathbf{y}_i))^2 \right) \tag{14}$$

Now all we need to do is perform gradient descent on the loss function with respect to θ and the optimal θ or θ^* will give us our approximation n_{θ^*} to the solution.

Concrete example I : Enneper's surface

Consider the minimal surface equation (written in Cartesian for cleanliness)

$$(1+u_x^2)u_{yy} - 2u_x u_y u_{xy} + (1+u_y^2)u_{xx} = 0, \qquad (x,y) \in \Omega \tag{15}$$

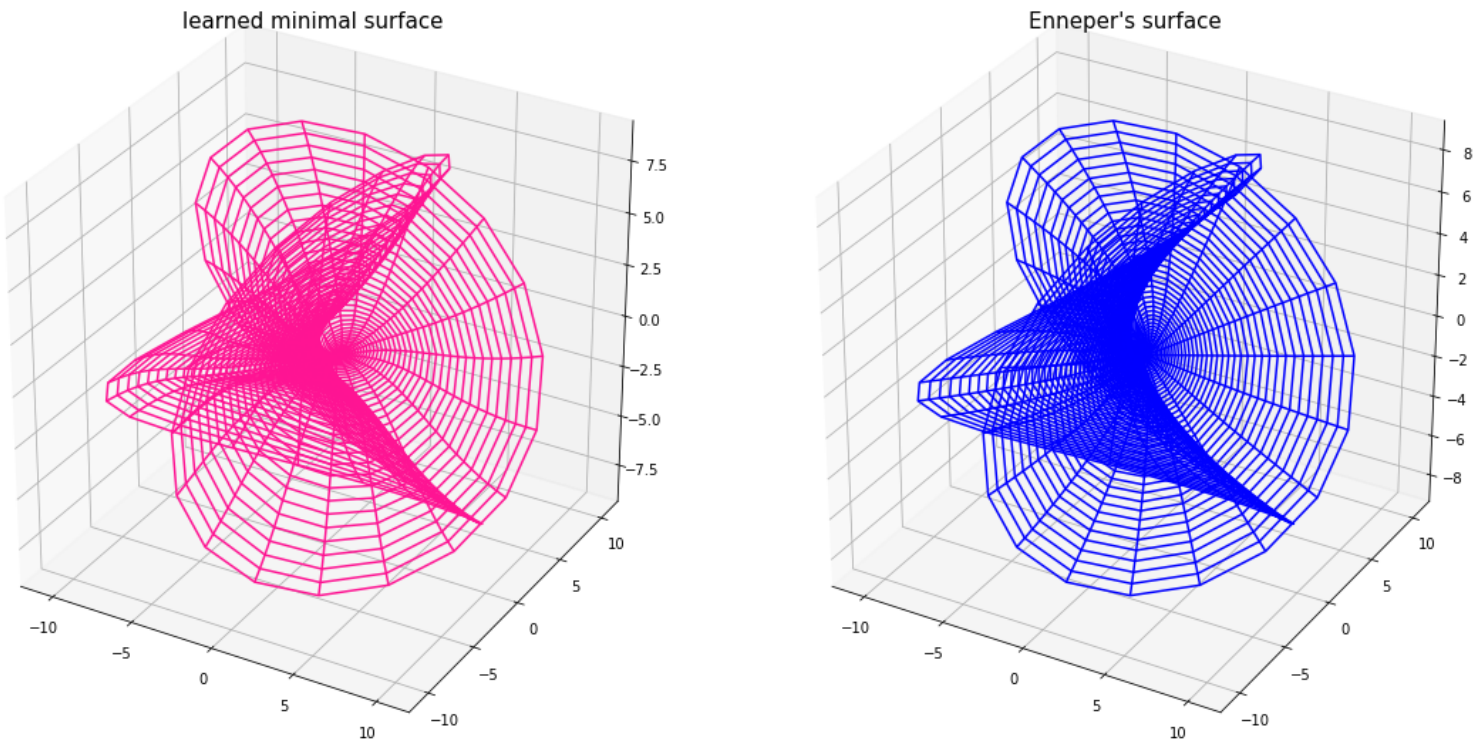
$$u|_{\partial\Omega} = g \tag{16}$$

- Here Ω is an origin centered disk of radius 3.
- Suppose g is parametrized as follows

$$\begin{aligned} x &= 3\cos\phi - 3^2\cos 3\phi \\ y &= -3\sin\phi - 3^2\sin 3\phi \\ g(x,y) &= 3^2\cos 2\phi \\ \phi &\in [-\pi, \pi) \end{aligned}$$

Concrete example I : Enneper's surface

g is an example of Enneper's wire. The associated minimal surface u is self-intersecting in Cartesian coordinates. So it's better to use $n_\theta(r, \phi)$ as your approximation. If you have followed all the steps correctly, you should get something like this.



Concrete example - II: Fokker-Planck equations

For this presentation we'll stick to equations of form

$$\mathcal{L}p = -\nabla \cdot (\mu p) + \frac{\sigma^2}{2}\Delta p = 0, \qquad \mathbf{x} \in \mathbb{R}^d \tag{17}$$

$$\int_{\mathbb{R}^d} p(\mathbf{x}) = 1 \tag{18}$$

where $\mu : \mathbb{R}^d \to \mathbb{R}^d$ and σ is a positive constant. μ is often called the drift.

Fokker-Planck: a broad classification

For ease of discussion let us classify the FP equations into two broad categories:

1. μ can be represented as $-\nabla V$ for some potential $V(\mathbf{x})$. Example: the unit circle attractor:

$$\mu(x, y) = -\nabla(x^2 + y^2 - 1)^2$$

   Here by attractor we mean attractor of the ODE system $\dot{\mathbf{x}} = \mu(\mathbf{x})$.

2. μ can't be expressed as gradient of a potential. Example: the Lorenz 63 system with butterfly attractor:

$$\mu(x, y, z) = \begin{bmatrix} 10(y - x) \\ x(28 - z) - y \\ xy - \frac{8}{3}z \end{bmatrix}$$

Fokker-Planck: closed form solutions

The equation for $\log p$ has the following nice factorization,

$$\mathcal{L}_{\log} \log p \stackrel{\text{def}}{=} (\nabla \cdot + \nabla \log p \cdot) \left(\frac{\sigma^2}{2} \nabla \log p - \mu \right) = 0 \tag{19}$$

1. When $\mu = -\nabla V$ we can set the 2nd factor to 0 and get,

$$\frac{\sigma^2}{2} \nabla \log p = -\nabla V \tag{20}$$

$$\implies p \propto \exp\left(-\frac{2V}{\sigma^2} \right) \tag{21}$$

2. But when $\mu \neq -\nabla V$, closed form solutions are not known in general.

Fokker-Planck: challenges

- The density condition

$$\int_{\mathbb{R}^d} p = 1$$

  is not reasonably implementable beyond 1 or 2 dimensions which is just reflective of the fact that numerical integration is hard and suffers from the curse of dimensionality. There's just too much space to cover!
- So let's give up on trying to find a density solution and instead try to find an unnormalized solution cp where c is a non-zero constant. You can still do a lot of interesting things even with the unnormalized solution.

Fokker-Planck: challenges

- Recall the loss function. Without the density condition it just looks like,

$$\frac{1}{N}\sum_{i=1}^{N}(\mathcal{L}n_\theta(\mathbf{z}_i))^2 \tag{22}$$

Note that cp is a zero of the operator $\mathcal{L}$ for any c so the loss function is far from convex. But we're happy to find an $n_\theta \approx cp$ for any c except $c = 0$ where you lose all information. If we perform gradient descent on the loss function above then it's incredibly easy to end up with the zero solution.

Fokker-Planck: challenges

- To avoid the zero solution we can instead try to solve for $\log p$. $\log(cp)$ is a zero of $\mathcal{L}_{\log}$ iff $c \neq 0$. So if we try to figure out $n_\theta \approx \log(cp)$ with the loss function

$$\frac{1}{N}\sum_{i=1}^{N}(\mathcal{L}_{\log} n_\theta(\mathbf{z}_i))^2, \tag{23}$$

gradient descent doesn't result in any zero-information solution and we finally have our approximation for the unnormalized solution $\exp(n_\theta) \approx cp$.

The entire algorithm

1. Sample $\{\mathbf{z}_i\}_{i=1}^N$ from $\mathcal{D}$, the domain of interest. Select the desired architecture for n_θ. Select adaptive learning rate α and number of epochs E.
2. For $k = 1, 2 \cdots, E$

   Compute ∇_θ of $L_{\log} = \frac{1}{N} \sum_{i=1}^N (\mathcal{L}_{\log}(n_\theta(\mathbf{z}_i)))^2$.

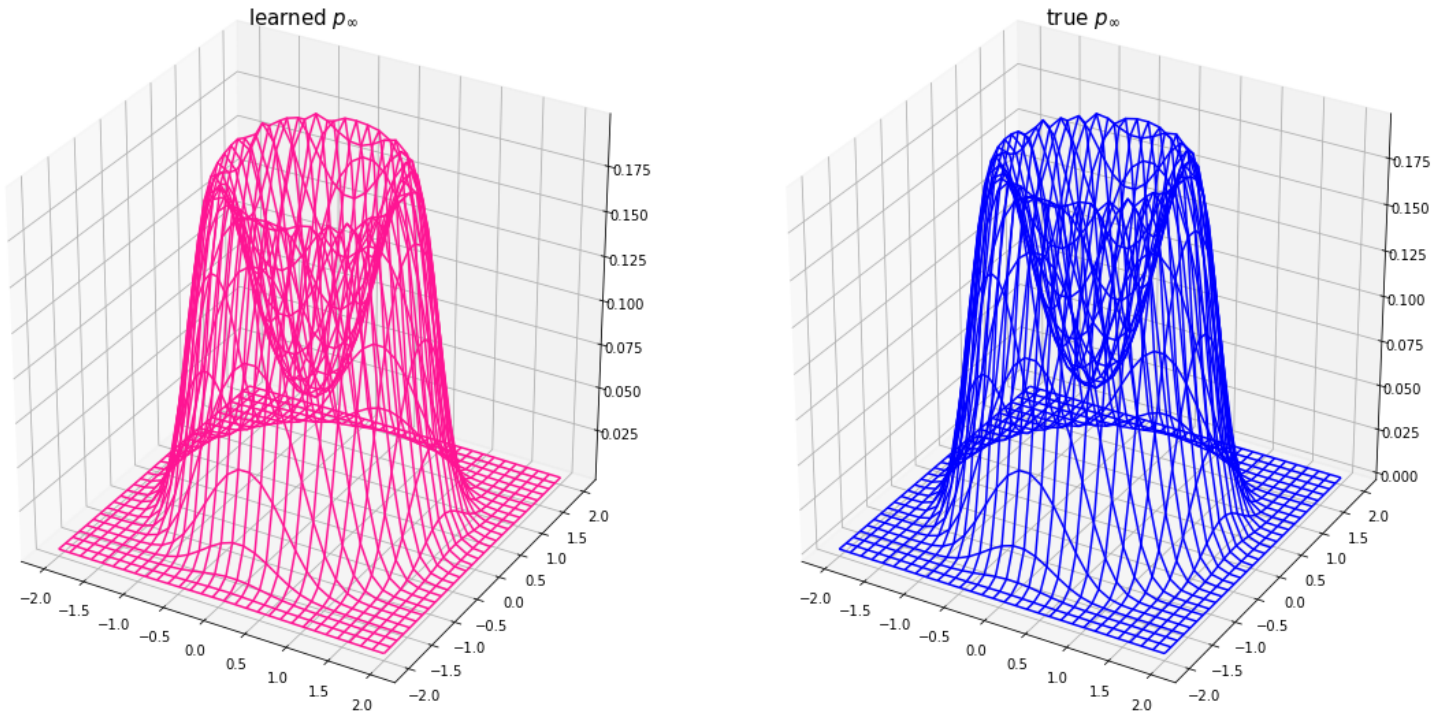
   where $\mathcal{L}_{\log} f = -\nabla \cdot \mu - \mu \cdot \nabla f + D\left(\|\nabla f\|_2^2 + \Delta f\right)$

   Update $\theta \leftarrow \theta - \alpha_k \nabla_\theta L_{\log}$

   Resample $\{\mathbf{z}_i\}_{i=1}^N$ every few epochs to make sure the domain of interest is being explored sufficiently.
3. e^{n_θ} is the final approximation for unnormalized p.

Fokker-Planck example 1

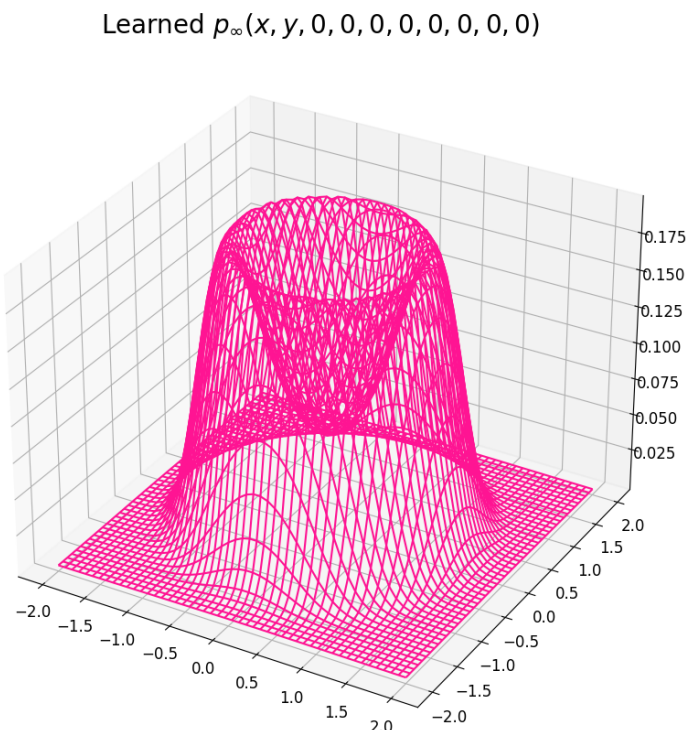
$$\mu(x, y) = -\nabla(x^2 + y^2 - 1)^2$$

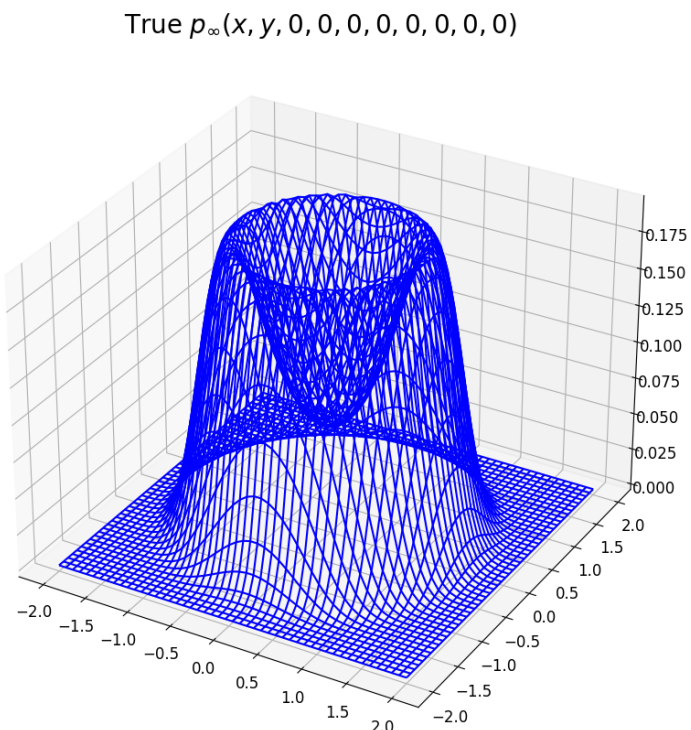


Fokker-Planck example 2

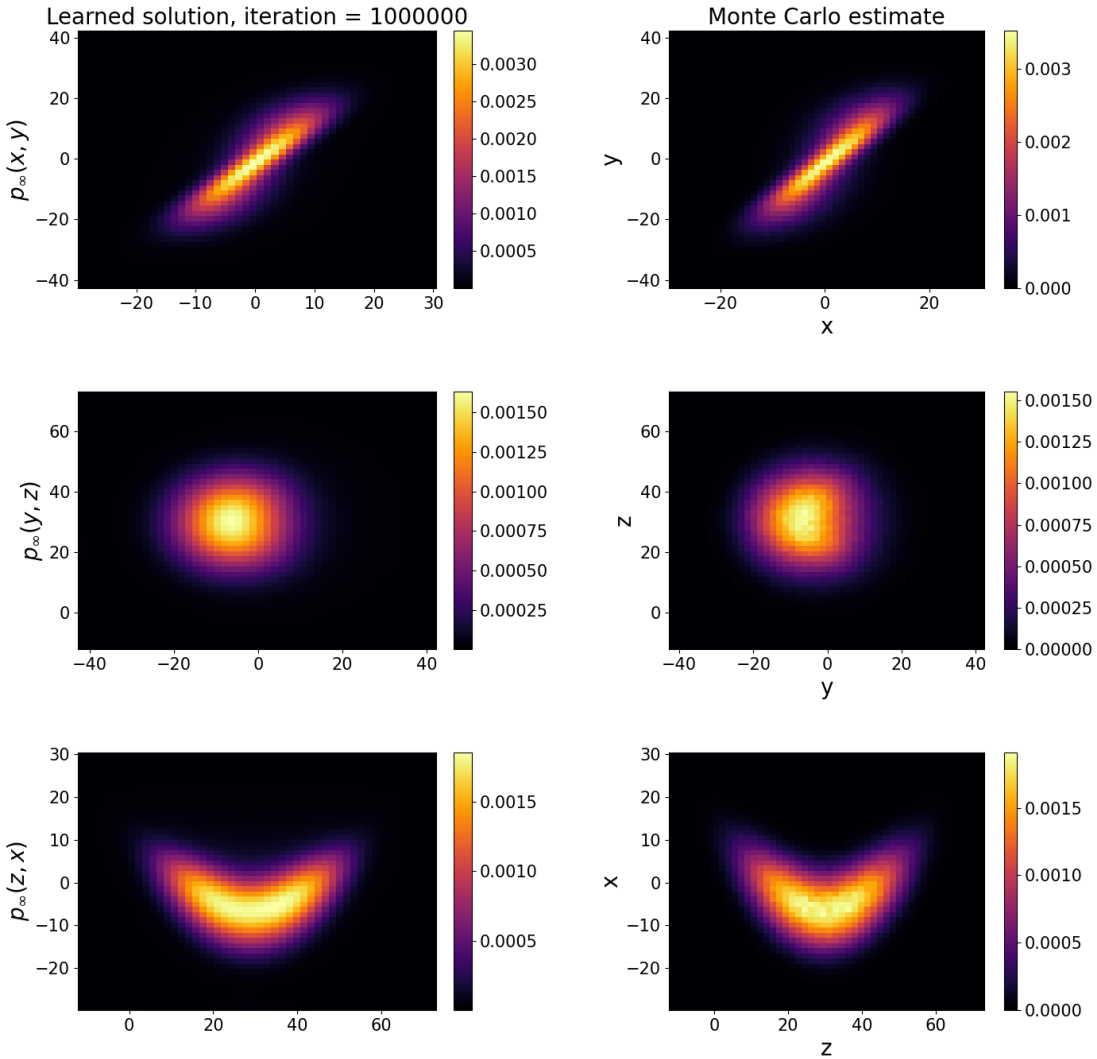
$$\mathbf{x} = (x, y, x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4) \in \mathbb{R}^{10} \tag{24}$$

$$\mu(\mathbf{x}) = -\nabla[\underbrace{(x^2 + y^2 - 1)^2 + \cdots + (x_4^2 + y_4^2 - 1)^2}_{5 \text{ terms}}] \tag{25}$$



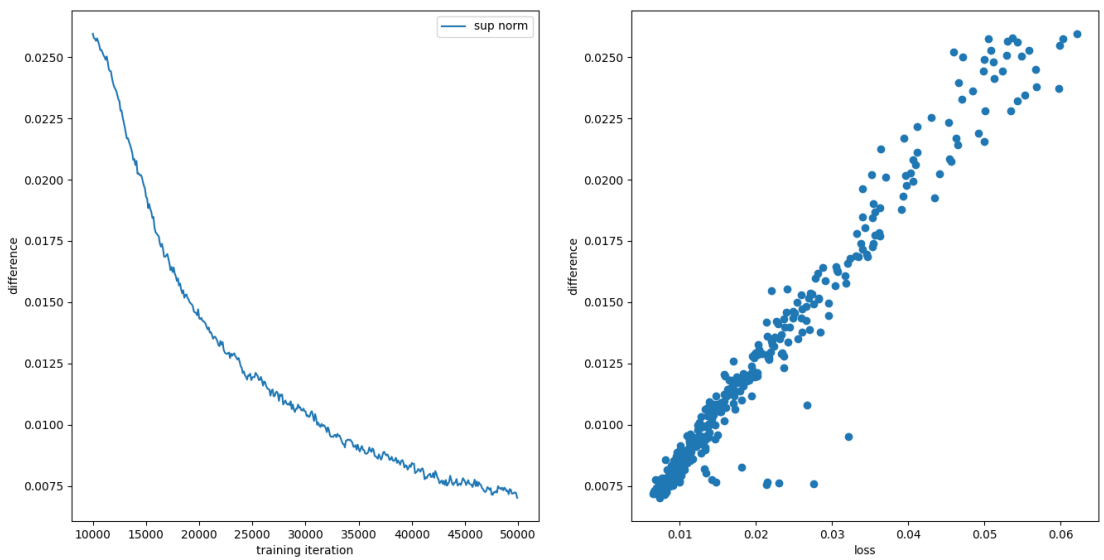


Fokker-Planck example 3, μ corresponds to L63



Does a decreasing loss function imply convergence to the truth?

For example 1 in $\mathbb{R}^2$ it's easy enough to normalize the approximate solution and we know the analytic solution and therefore we can study the relationship loss and the distance from the truth.



Here difference $= \|p - \text{normalized } e^{n_\theta}\|_\infty$.

Acknowledgements

My visit to Australian National University was funded by TIFR-Infosys Leading Edge Travel Grant (2023) and the International Research Connections Fund (2022). I thank Matthew Hole for hosting me and Zhisong Qu and Robert Dewar for useful discussions.

References I

[1] Y. A. Berezin, V. Khudick, and M. Pekker, "Conservative finite-difference schemes for the fokker-planck equation not violating the law of an increasing entropy," *Journal of Computational Physics*, vol. 69, no. 1, pp. 163–174, 1987.

[2] J. C. Whitney, "Finite difference methods for the fokker-planck equation," *Journal of Computational Physics*, vol. 6, no. 3, pp. 483–509, 1970.

[3] J. Náprstek and R. Král, "Finite element method analysis of fokker–plank equation in stationary and evolutionary versions," *Advances in Engineering Software*, vol. 72, pp. 28–38, 2014.

References II

[4] A. Masud and L. A. Bergman, "Application of multi-scale finite element methods to the solution of the fokker–planck equation," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 12-16, pp. 1513–1526, 2005.

[5] L. Pichler, A. Masud, and L. A. Bergman, "Numerical solution of the fokker–planck equation by finite difference and finite element methods—a comparative study," in *Computational Methods in Stochastic Dynamics*, Springer, 2013, pp. 69–85.

References III

[6] N. Chen and A. J. Majda, "Efficient statistically accurate algorithms for the fokker–planck equation in large dimensions," *Journal of Computational Physics*, vol. 354, pp. 242–268, 2018.

[7] J. Han, A. Jentzen, and W. E, "Solving high-dimensional partial differential equations using deep learning," *Proceedings of the National Academy of Sciences*, vol. 115, no. 34, pp. 8505–8510, 2018.

[8] J. Sirignano and K. Spiliopoulos, "Dgm: A deep learning algorithm for solving partial differential equations," *Journal of computational physics*, vol. 375, pp. 1339–1364, 2018.

References IV

[9] Z. Li, N. Kovachki, K. Azizzadenesheli, *et al.*, "Fourier neural operator for parametric partial differential equations," *arXiv preprint arXiv:2010.08895*, 2020.

[10] A. Pinkus, "Approximation theory of the mlp model in neural networks," *Acta numerica*, vol. 8, pp. 143–195, 1999.

[11] T. De Ryck, S. Lanthaler, and S. Mishra, "On the approximation of functions by tanh neural networks," *Neural Networks*, vol. 143, pp. 732–750, 2021.