Contemporary Physics

ISSN: 0010-7514 (Print) 1366-5812 (Online) Journal homepage: www.tandfonline.com/journals/tcph20

Recent developments at the intersection of AI and theoretical physics

Fabian Ruehle

To cite this article: Fabian Ruehle (13 May 2025): Recent developments at the intersection of AI and theoretical physics, Contemporary Physics, DOI: 10.1080/00107514.2025.2498817

To link to this article: https://doi.org/10.1080/00107514.2025.2498817

Published online: 13 May 2025.

CONTEMPORARY PHYSICS
https://doi.org/10.1080/00107514.2025.2498817

Taylor & Francis
Taylor & Francis Group

Recent developments at the intersection of AI and theoretical physics

Fabian Ruehle

Department of Physics and Department of Mathematics, Northeastern University, Boston, MA, USA

ABSTRACT
While machine learning is nowadays used in essentially all natural sciences, the timeline of adaptation in the different disciplines, and the subfields within each discipline, varies widely. Theoretical Physics and Mathematics are among the latest adopters of machine learning techniques, which is in part due to differing requirements that these fields have as compared to others. In this article, we discuss recent developments in artificial intelligence that are specifically geared towards application and scientific discovery in theoretical and mathematical physics. Along the way, we point out other phenomena at the intersection of Physics and AI, including chaos theory, phase transitions, spin glasses, symmetries, classical and statistical mechanics, and quantum theory.

ARTICLE HISTORY
Received 31 March 2025
Accepted 23 April 2025

KEYWORDS
Theoretical physics; mathematics; neural networks; Kolmogorov-Arnold networks; interpretable AI; reinforcement learning

1. AI and physics

Artificial Intelligence (AI) is everywhere nowadays, even though there is no clear, universally agreed upon definition what AI actually means. Especially among non-domain experts, the term is often used synonymously with Machine Learning, which in turn is mostly used synonymously with Neural Networks (NNs). While this is not doing justice to the developments outside of Neural Networks (NNs), in the interest of brevity, this article will focus only on applications of NNs to questions in theoretical and mathematical physics, and vice versa. However, even this is too broad a field with too many developments to cover, so we will illustrate a few sample applications only.

Before getting into these details, however, we want to give a brief overview of research at the intersection of Physics and NNs more broadly. The two fields share a rich and successful history. In the 'Physics for ML' direction, there are multiple ways in which physical models and ideas have entered the study of machine learning. We will outline how Physics ideas and techniques have been used to study NNs and how they learn. This process, as we will see in more detail in Section 2, is nothing but a (extremely) high-dimensional parameter optimisation problem. Indeed, it is proven that neural networks can learn any function (with very mild technical assumptions) if they just have enough parameters, which is known as the Universal Approximation Theorem [1, 2]. This means that one need not worry about NNs being too restrictive to solve the problem at hand; instead, one should worry about these other two questions:

(1) If the NN can learn any function, how do I get it to learn the function I am interested in, instead of all the other functions that look similar over some ranges of inputs, but are markedly different on others?
(2) Can the NN learn the function in practise, i.e. given finite resources?

These two questions are not independent, and their answer depends on how the NN parameterises the target function, and how the optimisation process selects the function among all possible functions. As we will discuss in more detail in Section 2.2, the latter question is phrased in terms of the so-called 'loss landscape', which is a name for how much the function learned by the NN deviates from the target function we actually want to learn. This deviation depends on the parameters of the NN, and finding the right function becomes a question of finding values for the NN parameters such that the difference between the learned and the target function is small. If we think of the loss landscape as a rugged energy landscape that depends on the NN parameters, minimising the deviation is akin to letting physical systems evolve towards states where they minimise the energy. The study of complicated energy landscapes with (exponentially) many local minima is a classical subject in Condensed

CONTACT Fabian Ruehle f.ruehle@northeastern.edu

Matter Physics, for example in spin glass systems, and a lot of insight has been provided by using statistical tools developed in this context, see e.g. [3–6] for early work and [7] for a review.

The way in which the NN parameterises the target function is known as NN architecture. It impacts the shape of the loss landscape, and hence which minima are found, and how fast they can be found. Ideas from Physics have motivated NN architectures, with important contributions coming again from statistical or condensed matter Physics. Indeed, the 2024 Nobel Prize in Physics was awarded to Hopfield and Hinton 'for foundational discoveries and inventions that enable machine learning with artificial neural networks,' more precisely Hopfield Networks [8] and Boltzmann Machines [9].[1]

Relatedly, Physicists have developed Neural Networks that behave nicely under symmetry transformations. Symmetries are perhaps one of the most important guiding principles in Physics, and many breakthroughs in fundamental Physics are based on symmetry considerations. An important example is that Physics does not depend on the choice of coordinate system in which it is described. The underlying concept is called diffeomorphism invariance and is at the heart of Einstein's theory of General Relativity. We need to be a bit more careful about what we mean by 'the Physics does not depend on the choice': Consider for example the Lorentz force in classical electrodynamics:

$$\vec{F} = \vec{F}(\vec{v}, \vec{B}, \vec{E}) = q(\vec{E} + \vec{v} \times \vec{B}). \tag{1}$$

If Alice chooses to rotate their coordinate system by a three-dimensional rotation matrix $R \in \mathrm{SO}(3)$, the electric and magnetic field, as well as the velocity point in the rotated direction, and consequently, the force is also rotated:

$$\begin{aligned}\vec{F}(R\vec{v}, R\vec{B}, R\vec{E}) &= q(R\vec{E} + R\vec{v} \times R\vec{B}) = q(R\vec{E} + R(\vec{v} \times \vec{B})) \\ &= R\vec{F}(\vec{v}, \vec{B}, \vec{E}).\end{aligned} \tag{2}$$

Note that we had to use a special property of the rotation group, namely that the determinant of all rotation matrices is 1, in the second equality. Using this, we find that applying the rotation to the variables and then computing the force is the same as first computing the force and then rotating the result. Functions that behave this way under symmetry transformations are called equivariant. This case was special since both the variables and the function were vector-valued, so it was the same 3D rotation matrix acting on either. In general, we could have scalar functions that depend on vectors, vector-valued functions that depend on spinors, etc. This is important for example in NNs that classify images as cat and dog images, say: The result should be invariant under rotations, translations, and reflections. These symmetries generate the Euclidean group, and Euclidean-equivariant neural networks have been proposed in [10, 11].

Concerning finiteness of resources, scientists usually seek to minimise the following:

- Training Time: Training a NN longer usually improves its performance but also costs more.
- Model Size: The larger the model, the better its performance. But this also increases the compute resources for training the model and for querying it. Nowadays, this is especially a problem for Large Language Models with billions or even trillions of parameters.
- Dataset Size: The larger the training set, the better the accuracy. In some disciplines, data is expensive to generate or annotate. In such cases, domain experts are needed for annotation, or (often very expensive) experiments are needed to run to generate the data.

Interestingly, fixing two of these and varying the third often leads to a power-law dependence of the error on the varying parameter. This phenomenon is known as neural scaling laws [12, 13]. While the scaling laws themselves seem to be somewhat independent of the choice of model architecture, their offset seems to depend on these choices, meaning that better results can be achieved earlier by improving the architecture. Recently, scaling laws were studied using tools from theoretical Physics, namely random matrix theory, and Feynman graphs, in [14]. Interestingly, the training of NNs in the limit where the number of their independent parameters goes to infinity (known as the Neural Tangent Kernel regime) becomes very simple [15–17]: The NNs behave like an ensemble of random Gaussian functions, and the parameter updates follow a simple update rule that only depends on the NN architecture. As soon as one deals with finite NNs, this approximation breaks down, but it does so in a controlled way. One can start the analysis from the infinite parameter regime, where the theory is simple, and then study the perturbation theory around this points where perturbations are of the order $1/N$ with N the number of NN parameters. This is done in perturbation theory in (Quantum) Mechanics / Field Theory all the time. The tools developed for perturbation theory in Physics have first been applied to NNs in [18–21], see [22] for a recent review.

A final recent development in AI based on a Physics phenomenon that we want to mention here are diffusion models [23]. Diffusion is the main process by which images and videos are generated nowadays. The idea is that one starts with 'ordered pixels' that form an image

and then lets the pixels 'disorder' or diffuse in multiple steps into what looks like white (Gaussian) noise. In order to generate images from white noise, one inverts the noising steps, i.e. denoises the image. This is done using a NN (at least until we can recruit Maxwell's demon [24] for the job). We illustrate this process in Figure 1, where the picture on the left was created with the diffusion model DALL-E, and on the right, we added Gaussian noise with 0 mean and variance 5 for each pixel.

2. Introduction to neural networks, supervised ML, and RL

Before we describe research at the interface of ML and theoretical Physics, we review the key concepts and introduce some notation here. There are many textbooks on the subject, and I will be following [25]. What sets machine learning apart from other data science techniques (such as genetic algorithms or topological data analysis) is that something is *learning*. In the context of neural networks, this means that we have a map with parameters, and the learning means that the parameters are adapted to do something useful. While the precise way in which NNs learn is not understood, there is empirical evidence that the parameters are adapted such that NNs extracts features, meaning they find patterns or learn heuristics from the input data, which they then represent in some 'embedding space'. This embedding is then used to make predictions about the data.

Neural networks are just parameterised maps from some input space (which is typically represented on a computer as a collection of real numbers, $\mathbb{R}^n$) to some output space (again, represented as a collection of real numbers, $\mathbb{R}^m$). For some applications of NNs, such as interpolation or regression, the need to learn an approximate function is quite intuitive. However, learning maps can also serve to classify images (by learning a 'decision boundary' that partitions the space into regions belonging to each class), or to generate sentences (by learning a map from an incomplete sentence onto a word in the Oxford English Dictionary, which should be added to the incomplete sentence; this is how Large Language Models like ChatGPT work). This is illustrated in Figure 2.

2.1. Neural networks

The initial idea behind (artificial) Neural Networks was to model the human brain: Neurons are connected by synapses that transport a signal. Once the signal is strong enough, the neurone 'fires' and stimulates/activates other neurones. However, instead of the analogy with biology, let us introduce NNs as parameterised maps $f_\theta(x)$, where we denoted all parameters of the NN collectively as θ. Of course, there is a huge freedom in how we parameterise the map, but the way it is done in the simplest NN (known as a feed-forward NN, or a fully connected or dense NN, or a multi-layer perceptron) is to alternate affine maps and non-linear maps. An affine map

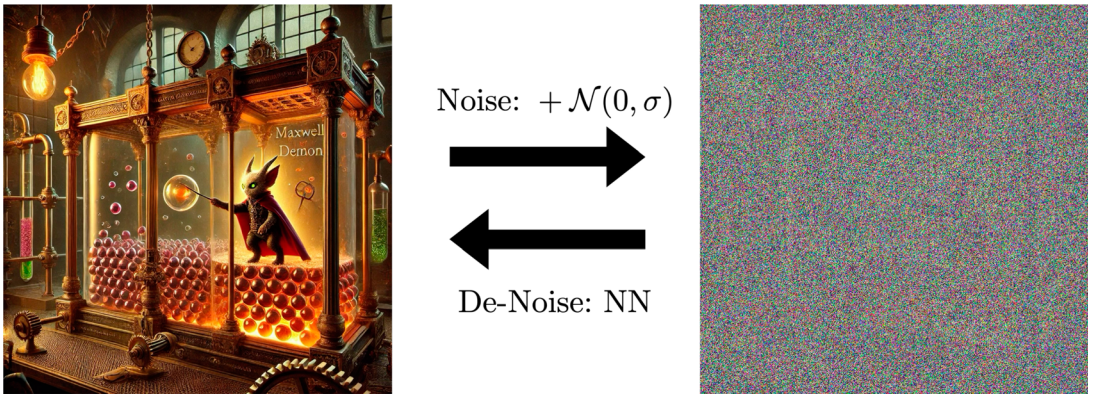

Figure 1. Illustration of image generation via diffusion. An image of Maxwell's demon is diffused by adding Gaussian noise with zero mean and variance 5, $\mathcal{N}(0, 5)$.

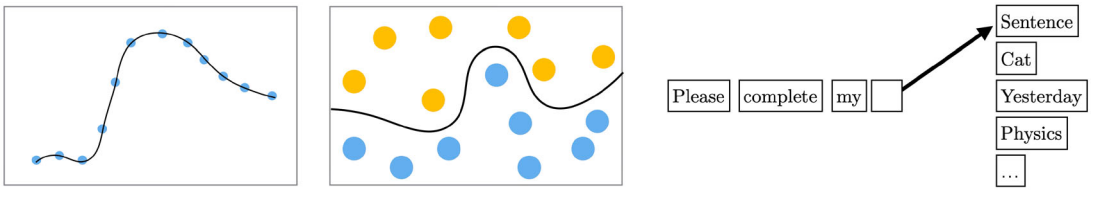

Figure 2. NNs learn maps that can serve as regression of given data, decision boundaries in classification tasks, and next-word-prediction in autoregessive generative models such as ChatGPT.

is multiplying the input by a matrix (called the *weight matrix* W) and adding a vector (called the *bias* $\vec{b}$),

$$\vec{y} = W\vec{x} + \vec{b}, \tag{3}$$

where $\vec{x} \in \mathbb{R}^n$ and $\vec{y} \in \mathbb{R}^m$. Thus, the weight matrix is an $m \times n$ matrix and the bias is a vector in $\mathbb{R}^m$. The parameters of this map are the weights (i.e. the entries w_{ij} of the weight matrix) and the biases b_i, $\theta = (w_{ij}, b_i)$. Collectively, these parameterise all affine maps from $\mathbb{R}^n$ to $\mathbb{R}^m$, but this is of course a boring, highly restricted class of functions, which is not very powerful or expressive. One might try to make it better by stacking multiple affine maps,

$$\vec{y} = W^{(2)}\left(W^{(1)}\vec{x} + \vec{b}^{(1)}\right) + \vec{b}^{(2)}, \tag{4}$$

but that is just equivalent to a single affine map of the form (3) with $W = W^{(2)}W^{(1)}$ and $b = W^{(2)}\vec{b}^{(1)} + \vec{b}^{(2)}$. Put more formally: affine maps are closed under composition. However, if we apply a non-linear function to (each component of) $W\vec{x} + \vec{b}$ before applying the next affine map, one cannot simply factor the composition as in (4), and one obtains a non-trivial and more general map. The parameters of this map are still all weights and biases; the non-linearities, also called activation functions $a^{(i)}$, are typically simple functions such as $\tanh(x)$ that act on each component of the vector independently (but any non-polynomial activation function will do), and do not have any parameters. For example a (feed forward, artificial) NN f_θ of depth 2 and tanh activation function would be

$$\begin{aligned} &f_\theta : \mathbb{R}^n \to \mathbb{R}^m \\ &\vec{x}_i \mapsto \vec{y}_i = \sum_{j=1}^{h} W_{ij}^{(2)} \left(\tanh\left(\sum_{k=1}^{h} W_{jk}^{(1)} \vec{x}_k + \vec{b}_j^{\,(1)} \right) \right. \\ &\qquad \left. + \vec{b}_i^{\,(2)} \right) \end{aligned} \tag{5}$$

This is usually represented as a computational graph where each node (called a neurone) represents a component of a vector on which the non-linear activation functions act, each affine map adds another layer to the graph, and the layer connexions in the graph correspond to the weights and biases. The number of layers (i.e. the number of affine maps $\mathbb{R}^{h_i} \to \mathbb{R}^{h_{i+1}}$) is called the depth of the NN, the 'intermediate' layers are called hidden layers (since they are never output), and the dimension h_i of the hidden $\mathbb{R}^{h_i}$ are called the width of the NN. We give an example in Figure 3(a).

While parameterised maps f_θ might seem like a strange way of writing a function, we actually encounter similar functions in many places in Physics, for example in Fourier or Taylor expansions of a function (taken from $\mathbb{R} \to \mathbb{R}$ for simplicity here),

$$f(x) = \sum_i w_i^{(2)} \cos(w_i^{(1)} x + b_i), \tag{6a}$$

$$f(x) = \sum_i w_i^{(1)} \left(x - b_i^{(1)}\right)^i, \tag{6b}$$

where in the Taylor expansion $w_i^{(1)} = \frac{\partial_x^i f(x_0)}{i!}$, $b_i^{(1)} = x_0$. The difference to the (deep) NN parametrization is that the latter is not well-adapted to be understood by a human (ironically, since NNs were modelled after the human brain). In contrast, the meaning of the parameters in the Fourier expansion (amplitude, frequency, phase shift) and Taylor expansion (expansion point, importance of higher order corrections) is very intuitive. This illustrates one of the biggest problems in NNs: Their parametrization is not human-interpretable (especially not in modern neural networks with tens of layers and billions or trillions of parameters), which is why NNs are often called *black box models*.

This begs the question: If NNs are black box, why should one choose a parametrization as in (5)?[2] While I do not know a great answer to this, the Universal Approximation Theorem at least ensures that we can write our function in this way. A second, equally little satisfying answer is that it just works great across many disciplines. Thirdly, NN show an interesting phenomenon known as 'double descent' [27, 28]. A classic result known as the bias-variance tradeoff in fitting functions dictates that

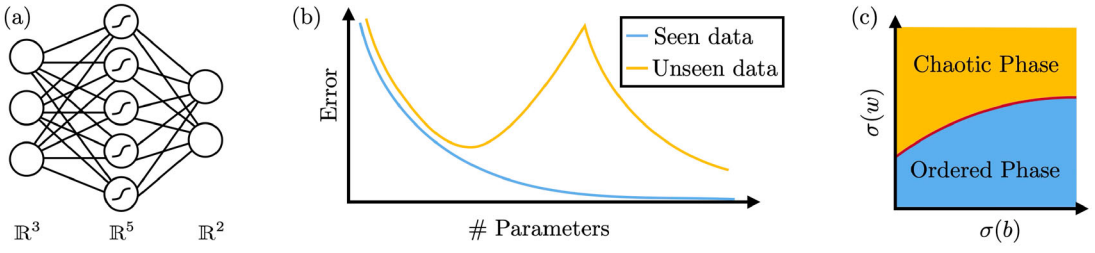

Figure 3. (a) Computational graph of the NN in (5) with $n = 2$, $h = 5$, $m = 3$. (b) Schematic illustration of double descent. (c) Phase diagram of the NN parameter space for tanh activation, with the edge of chaos in red, see [26].

one should use just as many free parameters as necessary. If there are too few, the function is not complex enough to fit all points; if there are too many, the function is too complex and it fits to noise or is ill-behaved away from training points. Both lead to poor performance on unseen data known as the generalisation gap. This is particularly pronounced for fitting high-degree polynomials akin to the Taylor expansion (6b). Assume that we want to learn a cubic function from a collection of 100 data points $(x, y = x^3 + \epsilon)$, where ϵ models small random noise. If we only include 2 parameters in our polynomial ansatz, we can only learn a linear polynomial, meaning that the parameterised function is not powerful enough. If we include 100 terms, we have more than enough parameters to fit all 100 data points exactly. However, the resulting polynomial is of degree 99, which means that its derivative is of degree 98 and can thus have (up to) 98 extrema. This leads to wild fluctuations in between the 100 fixed data points, the function looks nothing like a cubic function, and the generalisation error is huge. Of course, in general it is very difficult to determine what the right number of parameters is. Luckily for NNs, if one vastly overparameterizes the NN, the error on unseen data starts to go down again (hence the name double descent) and the model generalises very well; see Figure 3(b) for an illustration.

By studying limiting cases, as is often useful in Physics, we can give some intuition for what the weights and biases are doing. This also connects to other research areas in Physics, that of phase transitions and chaos. Consider the limit of very large biases, $b \gg Wx$. In this case, we can neglect Wx against b, which means that the NN becomes essentially independent of the input x. This is called the ordered phase, since the NN is well behaved (it is essentially constant). In the other extreme regime of large weights, $Wx \gg b$, a tiny change Δx in the input will be amplified and typically result in huge changes of the output. This is the hallmark of chaotic systems, and this phase is called the chaotic phase. As a consequence, there is a phase boundary in the (W, b) parameter space between these two regions (the precise shape of which depends on the activation function). This phenomenon was described in [29, 30]. The interesting region, where the NN is neither constant nor chaotic, is at this phase boundary, which is known as the 'edge of chaos', see Figure 3(c).

2.2. Supervised machine learning

After choosing a NN and initialising the weights and biases with random numbers, we have a random map. The next task is to train the NN, i.e. adjust its parameters θ such that this random initial map morphs into the map we are actually interested in. A classic example for how to do this is supervised ML, where one has a collection of $(\vec{x}, \vec{y})$-pairs. The inputs $\vec{x}$ are called features and the outputs $\vec{y}$ are called labels. To train the network, we need a measure for how far off the current NN prediction $\hat{\vec{y}} = f_\theta(\vec{x})$ is from the true values $\vec{y}$. These distance functions are called loss functions. There are many choices for loss functions, but all should be non-negative and only zero if $\hat{\vec{y}} = \vec{y}$. A simple choice is the mean squared error,

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N} \sum_{\alpha=1}^{N} |\hat{\vec{y}}_\alpha - \vec{y}_\alpha|^2, \tag{7}$$

where $\alpha = 1, 2, \ldots, N$ labels the different input-output pairs in the dataset. During training, one updates the parameters θ (meaning all weights and biases in all layers) such that the loss is minimised. While the loss is a simple, nice, convex function (a parabola) in the outputs, it is typically a complicated function in the (millions of) parameters that are used to compute the outputs. This complicated function is known as the loss landscape, and finding minima in such a high-dimensional space is very challenging, and only the simplest (first-order) minimisation methods are feasible. This is why NNs are typically trained using gradient descent optimisers, which back-propagate the error at the output layer through the network layer by layer, and adjust the parameters via steepest descent, meaning the parameters are adjusted by subtracting the gradient of the loss function.

This method has the obvious disadvantage that these updates stop as soon as the gradient vanishes, which is the case for local extrema or even saddle points. There are many Physics-motivated optimisation techniques that try to improve on that by borrowing tricks and techniques that Nature uses to minimise energy, something that it is very good at in systems with large degrees of freedom.[3] Given how big the field of Physics-inspired optimisation is, we can only give a small, incomplete overview. A simple modification is to add a momentum term to the gradient descent, such that the parameter updates 'roll past' a local minima or saddle point instead of getting stuck there [31]. This is an instance of a class of optimisers based on classical Physics, more precisely Hamiltonian or Lagrangian dynamics (see e.g. [32] for a review). Other ideas include adding friction terms or use (relativistic) Born-Infeld dynamics and impose energy conservation [33]. Yet others borrow ideas from thermodynamics and add entropy [34] or noise [35] regularizers, or use simulated or quantum annealing to find good minima [36]. Quantum tunnelling [37] can help overcoming local minima, and the algorithm (or the entire NN) can run on quantum computers [38].

2.3. Reinforcement learning

Reinforcement learning (RL) is a different branch of ML where we also train a NN, albeit without explicit labels. RL is based on behavioural psychology, where the NN performs actions which are either positively or negatively reinforced, depending on a predefined goal we want to achieve. This makes RL ideal for applications where we know what we want to achieve but do not know how to achieve it. Perhaps the most famous RL application is Deepmind's algorithm that was used to beat the world champion in the board game Go [39, 40]. Nowadays, RL is also widely used in robotics and for training large language models, but we will focus on games.

To be more specific, we will apply RL to problems in theoretical Physics and Mathematics by 'gamifying' the underlying scientific problem. We illustrate how this works using Go as an example. Go is a two-player game played on a 19×19 grid. Players take turns in placing one stone onto any free grid point, and if a player surrounds all stones of the opponent with his stones, the opponent's stones are removed from the board. The game ends if no player can or wants to place more stones, and the player whose stones surround the bigger area wins. We immediately see that there are $3^{19\times 19} \approx 1.7 \times 10^{172}$ possible Go boards (since each grid point can have a black stone, a white stone, or no stone). To put this into perspective, the visible Universe has about 10^{80} atoms. In contrast, the number of possible moves at any point in the game is (at most) $19 \times 19 = 361$. A naive strategy to coding Go on a computer would be a search tree, with the root node being the current Go board. From there, one checks all possible next moves and the resulting new boards. This is then repeated for each of the new Go boards, until a winning strategy is found, meaning a path through the search tree that ends in a Go board that is winning for the current player.[4] Of course, this is completely unfeasible in practise, since the search tree is too big to be stored, even if each atom in the Universe could store the state of a Go board. So instead of unrolling (forward-simulating) the entire search tree, one uses heuristics that decide which branch to focus on, i.e. which move to play next. As the reader might guess, this decision is outsourced to a NN. To describe the algorithm, we introduce the RL vocabulary that is typically used:

- **Agent:** The player(s) of the game.
- **State space:** The set of all possible Go boards. The elements of this set are called states.
- **Terminal states:** The subset of all states where the game ends (either winning or losing).
- **Action space:** The set of all possible moves. The elements of this set are called states.
- **Reward function:** The (positive or negative) reinforcement given to the agent for their actions.
- **Policy:** The function that chooses the next action.
- **Episode:** A game, i.e. a sequence of states and actions that lead to a terminal state.

Mathematically, RL is solving a (Markov) decision problem. There are many RL algorithms, which often involve a second NN in addition to the policy NN, which estimates how good the current state is. The NNs are trained based on the reward function, such that over time they propose more winning moves and learn to realistically estimate how good certain positions (states) are. In more detail, the algorithm is given the current state, from which it computes the next action and (optionally) a score for how good the state is. The action is carried out, which leads to a new state. Based on the quality of the new state, the agent is rewarded or punished according to the reward function. After that, the process is repeated with the new state given as input to the NN, until a terminal state is reached, at which point the game is reset. This is illustrated for a robot exploring a (discrete) landscape with the goal of finding the global minimum in Figure 4.

For two-player games like Go, NNs are trained via self-play, meaning that two AI agents are playing against each other over and over. A problem with RL can be sparse rewards: for example in Go, one typically chooses a reward function of +1 if the terminal state that is reached is winning and -1 if it is losing. However, this means that one needs to finish an episode before a reward can be assigned. It is also important to note that the NN is trained to maximise its return, which is defined to be the accumulated reward over many actions, instead of just its next reward. Otherwise the policy could get stuck in 'local minima', where the NN only plays moves that are beneficial in the short run but do not lead to a good or winning strategy in the long run. Indeed, RL agents playing chess will happily accept gambits or pawn sacrifices for a better long-term position on the board.

3. Limitations: computational complexity

A big advantage and reason for the success of NNs is that they are model-independent, which means they can be applied to all sorts of problems and areas. Most problems can be formulated as maps, and by the Universal Approximation Theorem, NNs can learn these maps. However, this does not necessarily mean that NNs will be successful (or outperform other algorithms). It is very difficult to know in advance whether a NN will be the best option for a given problem. As a rule of thumb, one should ask whether there already exist efficient, deterministic algorithms tailored to the problem one is trying

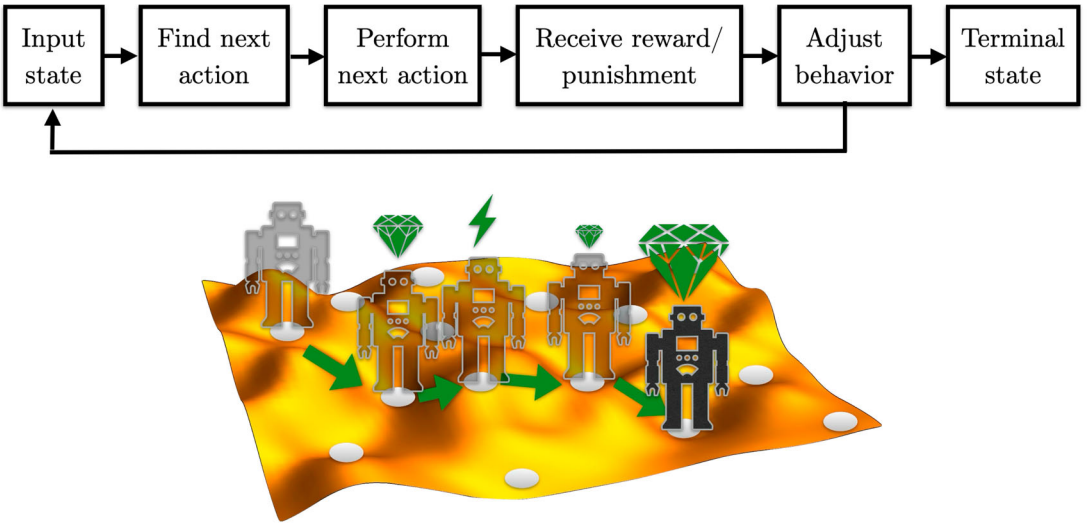

Figure 4. An agent explores a discrete state space by performing actions that are rewarded (diamond) or punished (lightning bolt) with the goal (terminal state) of finding the lowest point in the landscape. It adapts its behaviour based on the accumulated rewards and punishments.

to solve. If the answer is yes, it is typically better to use this algorithm.

This begs the question what 'efficient algorithm' means, and whether there could be more efficient algorithms. This is the subject of computational complexity theory. To simplify the discussion, one focuses on decision problems. These are an infinite family of problems whose input size is given by some parameter N, and whose output is either *yes* or *no*. One then classifies problems into different complexity classes depending on how long it would take to either find or check an answer as a function of the input size N. There is a whole hierarchy worth of these, and we will only cover the most important ones:

- **P:** The answer to problems in P can be decided in a time that grows at most polynomially in the input size N. This means there exists an algorithm that computes the answer in polynomial time. Problems in P are the simplest (or fastest to solve) in computational complexity theory.
- **NP:** The truth certificates (a set of numbers that can be used to decide that the answer is yes) can be verified in a time that grows at most polynomially. Note that this does not make any assumptions about how the truth certificate was found; it just postulates that given a solution to the problem, one can quickly check that it is a solution. It also does not make any assumptions about the time it would take to check that something is not a solution.
- **co-NP:** Like NP, but for the 'no'-instances
- **EXP:** Like NP, but it can take at most exponentially long to verify the yes instances
- **DEC:** The set of all decidable problems, no matter how long it takes.

From the definition, it is clear that $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP} \subseteq \mathrm{DEC}$. This is just the statement that if I have found an algorithm that solves the problem in polynomial time, I can check the yes instances in polynomial time. What is far from clear is whether for example $\mathrm{NP} \subseteq \mathrm{P}$, and thus $\mathrm{P} = \mathrm{NP}$. In fact, this is a millenium problem, one of the biggest open problems in Mathematics (or theoretical computer science). We will say more about complexity in a bit, but to illustrate the difficulties, let us give an example.

INVERTIBLE: Decide whether a given $N \times N$ matrix M is invertible.

This is clearly an infinite family characterised by the size N^2 of the matrix, and it clearly has a yes/no answer. The problem is also in NP, since given an inverse matrix L as truth certificate, one can check in polynomial time whether $ML = LM = \mathbb{1}$. However, we cannot use the same as a certificate for the no-instances (if no inverse matrix exists, the algorithm cannot return one), so the problem is not obviously in co-NP. Another way one could try to tackle this problem is to compute the determinant of M. All yes-instances would have a non-zero determinant and all no-instances would have a zero determinant. So the determinant can serve as a certificate for the yes- and no-instances. If one naively applied the Leibniz formula, which expands the determinant as

products of signed permutations

$$\det(M) = \sum_{\sigma \in S_N} \prod_{i=1}^{N} m_{\sigma(i),i}, \tag{8}$$

we would have to compute $N!$ many terms. This grows faster than any polynomial, so in particular finding the solution is not in P. Moreover, with the Leibniz formula, the certificate (the determinant) could not be checked in polynomial time, which would mean that the problem is in EXP. However, the problem is actually in P: One can compute the determinant by diagonalising the matrix (which amounts to solving a linear system of equations, which takes time $\mathcal{O}(N^3)$). Then, the eigenvalues are on the diagonal and the determinant is just the product of the eigenvalues, which can be computed in time $\mathcal{O}(N)$. Hence the total time for computing the determinant grows as a cubic polynomial. However, if we hadn't had the idea of computing the determinant via the eigenvalues, we would have concluded that the problem is in EXP and hence desperately searched for any algorithmic speedup.

The example illustrates that it is difficult to know whether one has found 'the best' algorithm; there might exist better ones, but they require a trick that we haven't thought of so far. This is why it is so difficult to answer whether P=NP. Just because I have a problem that I cannot solve fast does not mean that tomorrow someone will have a brilliant idea that solves the problem in polynomial time. Having said all that, most experts believe that P≠NP, meaning that some problems are genuinely harder than others, and we will make this assumption for the rest of this chapter.

Computational complexity theory has (at least) two surprises that sound completely insane if you hear them for the first time. The first is about the existence of so-called NP-complete problems [41, 42]. These are problems that lie themselves in NP and that are as difficult to solve as *any* problem in NP. It seems impossible to make such a statement, since we do not even know all NP problems. However, NP-complete problems are so general that one can show that any problem in NP can be mapped onto them in polynomial time. In turn, this means that if I could solve any NP-complete problem in polynomial time, I could map all problems in NP onto the NP-complete problem, use the polynomial time solver for this, and hence solve all NP problems in polynomial time, thus concluding that P=NP. Hence, under our assumptions, there is no fast way of solving NP complete problems. The second surprise is that there exist problems that are undecidable, so there exist problems that do not have a solution in any time complexity [43]! One might think that these are some over-engineered pathological examples that never appear in Physics, but that is actually not the case.

Many simple problems are NP complete. A simple example for an NP-complete problem is the following:

SUBSET-SUM: Given a set of N integers, decide whether there exists a subset of them which sum to zero.

The naive algorithm is just building all 2^N subsets and trying all of them (which is clearly not doable in polynomial time); however, if someone gives you a subset, you can easily verify that the integers sum to zero. So the problem is in NP. Showing that it is NP-complete is more work. Similarly, the following simple problem is undecidable:

DIOPHANTINE: Given a K polynomial equations with integer coefficients, does there exist a simultaneous zero of all equations with integer values for the variables, so in the simplest case

$$\exists ?\, x \in \mathbb{Z}: \quad a_0 + a_1 x + \ldots + a_N x^N = 0, \quad a_i \in \mathbb{Z}. \tag{9}$$

Undecidability of this problem was proven by Matiyasevich [44] and answered one of Hilbert's 23 problems [45].

NP-complete or even undecidable problems appear in Physics (and the real world) all the time. For example, the authors of [46] studied a model in which answering why the Cosmological Constant is so small in our Universe can be mapped onto the subset sum problem. Diophantine equations are also ubiquitous in Physics since problems with integer coefficients and integer solutions appear as soon as one quantises a theory [47]. Moreover, if we accept the notion that some problems cannot be solved fast, and others cannot be solved at all on any computer, we should also accept that the Universe is bound by the same rules. This means that one needs to be able to find a solution to the equations that govern our Universe and give rise to the things we observe now within the timeframe of 13 billion years, which is the lifetime of the Universe. Moreover, we cannot live in a Universe whose solution would require solving an undecidable problem. These ideas have been formulated in [46, 48, 49]. Theorists have also studied computational complexity in the context of black hole Physics [50].

Coming back to computational complexity and ML: If not even the Universe itself can solve NP-complete (or worse) problems fast, a NN most certainly cannot either. Since the runtime of NNs is polynomial – they just multiply matrices and apply a tanh here and there – they should only be able to solve problems in P. However, before jumping to conclusions, one should keep a few things in mind. First, while NP complete problems might not be solvable fast, it could be that (infinite) subfamilies

of the problem are actually solvable in P. Maybe physical or mathematical consistency conditions select those problems that can actually be solved quickly, albeit with a hitherto unknown algorithm. Another possibility is that some problem in the family is hard or undecidable, but on average the problems are easy to solve, i.e. they have a lower average case complexity. Third, perhaps we do not need to solve the problem exactly but only approximate a solution (and then maybe round it to the nearest integer). Indeed, hard problems can have approximation schemes that are in P. Fourth, NNs are often stochastic, so they might only be able to solve the problem two thirds of the time, say. Problems that can be solved in polynomial time probabilistically (for example on a quantum computer) are in the complexity class BQP. The most famous example is factorisation of integers into its primes. This problem is in NP (by trying out all prime numbers up to the given number). It is not known to be in P, but it is in BQP [51].[5] In summary, NNs could still perform well on such problems, either because the problems fed to the NN are not as difficult as they could be, or because the NN is finding a good approximation, or because the NN is allowed to fail on some instances.

4. NNs and differential equations

Arguably most of Physics is about describing how a system or object changes in time, which leads naturally to differential equations of the form

$$\dot{u}(t) = F(t, \vec{x}, u, \vec{\nabla} u, \ldots). \tag{10}$$

To uncover a new Physics law, Physicists traditionally study experimental data and find the equation governing the time evolution. In theoretical or mathematical Physics, one often postulates an action (based on a model of the real world). Demanding that the action is extremized then leads to the equation of motion of the object described by the action. This is a set of partial differential equations (PDEs) which one then needs to solve and compare to experiment to decide whether the model of the real world was good or bad. Moreover, typically the experiment (or the action) depends on multiple physical parameters. Either situation can be modelled by NNs, and the areas that study them are known as *Neural ODE/PDE*, *Physics-Informed NN* (PINN) and *Neural Operators*, respectively. Let us describe them in turn, using a simple example that probably every Physicist has studied: the harmonic oscillator. We know from Hooke's law that springs (within reasonable ranges of deformation, where 'reasonable' depends on the spring material), exert a force that is a linear response to displacement, $F = -kx$. On the other hand, Newton taught us that $F = ma = m\dot{v} = m\ddot{x}$. So together

$$\ddot{x} = -\frac{k}{m}x = -\omega^2 x, \tag{11}$$

and the solution is of course $A\sin(\omega t + \phi_0)$ where the amplitude A and phase shift ϕ_0 are integration constants that need to be determined by the initial conditions. This example is not quite the same as (10), which assumed we are interested in the rate of change of some variable u, while (11) has two derivatives, so it is a second order ODE in the position x. However, we can easily translate our equation into a system of first-order ODEs where each equation is of the form (10) by introducing multiple (auxiliary) variables u_i and functions F_i on the right-hand side,

$$\begin{aligned} \dot{u} &= F_1(t, u, v, \ldots) = v \\ \dot{v} &= F_2(t, u, v, \ldots) = -\omega^2 u. \end{aligned} \tag{12}$$

In this case, the function $u(t)$ is simply the position of x at time t and the auxiliary variable is the velocity $v = \dot{x} = A\omega\cos(\omega t + \phi_0)$.

Neural ODE

In a neural ODE, one replaces the right-hand side, meaning the unknown function F in (10), by a NN. The NN learns F and thus the rate of change of u from measured data $(t, \dot{u})$. For the sake of this example, let us assume we did not know about Hooke's law and Newton's law. Instead, we just have a spring with a weight attached to it. We move the spring from its equilibrium position and are interested in describing how the position and velocity change over time. To do so, we measure these quantities, such that we have a time series of data $(t, v = \dot{u}, a = \ddot{u})$, and we can fit two NNs to this data (or just one with two outputs) to learn F_1 and F_2. For the example where we use a single NN to learn both F_1 and F_2, we set up the NN to get inputs $(t_i, x(t_i), v(t_i))$ in $\mathbb{R}^3$ and outputs $(v(t_i), a(t_i))$ in $\mathbb{R}^2$. For training, we can simply use the Mean-Squared Error (MSE) loss

$$\begin{aligned} \mathcal{L}_{\mathrm{N-ODE}} &= \mathrm{MSE}(\hat{\vec{y}}(t_i), (v(t_i), a(t_i))) \\ &= \frac{1}{N}\sum_{i=1}^{N} |\hat{\vec{y}}_{1,i} - v(t_i)|^2 + |\hat{\vec{y}}_{2,i} - a(t_i)|^2. \end{aligned} \tag{13}$$

In this way, the NN learns the functions

$$\begin{aligned} \hat{\vec{y}}_1(t, u, v) &= F_1(t, u, v) = v, \\ \hat{\vec{y}}_2(t, u, v) &= F_2(t, u, v) = -\omega^2 u. \end{aligned} \tag{14}$$

So in a way, neural ODEs use time series data to interpolate the data. Instead of getting data from an experiment,

one can also get it in atheoretical setup from a different (more expensive) differential equation solver like a Runge-Kutta or finite element solver, and use this to train the NN. The whole procedure can also be generalised to neural PDEs.

PINNs

For PINNs, we assume that we have the differential equation, so we know the right-hand side F, but are interested in finding the function u that solves the PDE. The idea in this case is that we let u be the NN (which can represent any function, so in particular the function that solves the PDE), take derivatives of the NN with respect to its input,[6] and use the PDE itself as a loss:

$$\mathcal{L}_{\text{PINN}}(t,u) = \text{MSE}(\dot{u}, F). \tag{15}$$

Note that the loss is zero if and only if the function learned by the NN is a solution to the PDE.

In our example of the harmonic oscillator, assume we want to solve the ODE (11). We set up a NN with one input (the time t) and one output (the position $u(t)$). We generate some inputs in the input domain (so time points $t \in [t_0, t_1]$), compute two derivatives of the NN with respect to t and use the loss

$$\mathcal{L}_{\text{PINN}}(t,u) = \frac{1}{N}\sum_i |\ddot{\hat{y}}(t) - \omega^2 \hat{y}(t)|^2. \tag{16}$$

If we want to include the initial values (or boundary conditions) as well, we can add these as additional loss terms,

$$\mathcal{L}_{\text{PINN}}(t,u) = \frac{1}{N}\sum_i |\ddot{\hat{y}}(t) - \omega^2 \hat{y}(t)|^2 + |u(t_0) - \hat{y}(t_0)|^2 + |v(t_0) - \dot{\hat{y}}(t_0)|^2. \tag{17}$$

If we had a boundary value problem, we would simply replace the last term by $|u(t_1) - \hat{y}(t_1)|^2$. Note that this is much easier than using shooting methods for boundary value problems. Also, PINNs need not be adapted to stiff PDEs, and have been found to scale better for PDEs in higher dimensions than finite element methods, which suffer from the curse of dimensionality, which means one needs an exponentially large number of points (and hence elements or simplices built from these points) to approximate the solution of the PDE.

4.1. Neural operators

The solution found by the PINN depends on the spring constant k and mass m of the attached object (or rather, on the ratio ω^2 of the two). Since ω enters directly in the loss function, the function u and hence the NN is specific to and only valid for this chosen, fixed ω. In neural operators, the parameters get promoted to inputs, i.e. we would train a neural operator NN $u_\omega(t)$ to solve the ODE (11), where both t and ω are inputs to the NN. Of course, we would still only differentiate the NN with respect to its input t. This approach can be understood as interpolating PINNs for different parameters ω of the differential equation. The solutions to the neural operators are thus a family of functions depending on ω.

4.2. Example: Calabi-Yau metrics

String theory is a promising candidate for a unified description of all known fundamental forces in the Universe. A robust predictions of string theory is the existence of extra dimensions. In the best-understood case, string theory predicts six extra spatial dimensions. Searches for extra dimensions, for example at the Large Hadron Collider (LHC), have not detected any extra dimensions yet, which is why we know that, if they exist, they must be curled up and small (the current upper bound for one extra dimension is around 1 μm). However, the six extra curled-up dimensions in string theory cannot be just anything. String theory imposes equations of motion for all quantities that are described by the theory, and since it is a theory of everything, it imposes constraints on everything. One of the constrained objects is the metric of all dimensions (i.e. of the four dimensions of our Universe as well as the extra six, curled-up ones). As predicted by Einstein's theory of General Relativity, the metric is not constant, but rather a function of space and time. The equation of motion for the metric in the extra dimensions derived from string theory is a very complicated partial differential equation (to be precise, it is a fourth order PDE). The PDE demands that the extra dimensions cannot be 'too curved', meaning that a measure for curvature known as the Ricci curvature tensor, has to vanish. Metrics with these properties are called Ricci-flat.

To this day, nobody knows how to solve this equation analytically (and it is not clear that it has an analytic solution at all). Surprisingly enough, Calabi conjectured [52] that one can always find a solution if a certain topological invariant (the first Chern class of the tangent bundle), which can be easily computed for the given curled-up dimensions, is zero. Calabi showed that the solution is unique (without knowing whether it actually exists, which is something mathematicians like to do). Yau proved Calabi's conjecture and with it the existence of a unique solution to the equations of motion imposed by string theory on the metric in 1977 [53], which earned him the Fields Medal in 1982. Moreover, these spaces

became known as Calabi-Yau manifolds (CYs). However, Yau's proof is not constructive, which means he shows the existence of a solution without constructing it (which is also something mathematicians like to do). Within the 45 years that passed since his proof, nobody has managed to find a constructive proof except for very special cases. Not knowing the metric is a problem in studying the theory, since everything couples to gravity and hence the metric appears everywhere. Not knowing the metric meant that string theorists had to get creative (and learn algebraic geometry) to nevertheless extract Physics from String Theory. This has led surprisingly far, but some quantities are out of reach without an explicit metric.

The need to solve a complicated PDE of high degree in 6D, together with the fact that in all likelihood numerical methods will be the only way of addressing the problem, makes NN an ideal method to try. The first papers that attempted this were published on the same day by two independent groups [54, 55], and a third group published their results a few months later [56]. There have been multiple follow-up works since then; in particular [57] generalised the approach to essentially all known Calabi-Yau manifolds. The approach [54] used Neural ODEs, PINNs, and Neural Operators, although we did not call it that (the names were popularised in [58–60], respectively, which were published after our paper was written).

Let us discuss different techniques we employed to approximate the metric. For all methods, we need to have points in the region of interest. For the harmonic oscillator, these were points in some time interval $t \in [t_0, t_1]$. In the case at hand, we need points on the curled up extra dimensions. These are described by sets of equations with six (independent) variables, and the simultaneous solution to all equations gives points that lie on the Calabi-Yau manifold, see Figure 5(a). While solving the equations is complicated and tedious, it is a classical algorithm that does not involve AI, and we won't discuss it further but just assume we have a collection of points on the CY we can use as inputs for a NN.

However, this illustrates an important difference between AI for theoretical Physics / Mathematics and AI for other fields. In the former, one can just generate mathematical data. This data will be exact (solutions to some equations) and does not require expensive lab equipment to measure or human annotators or domain experts to generate, curate and label the data. In this sense, mathematical data is cheap. Moreover, the notion of input data is much more precise in this case. It is clear what each input feature means (in this case, a coordinate on curled up dimensions) and even what type of object all the inputs describe within the entirety of the input space $\mathbb{R}^n$. In contrast, if the inputs are the pixels of an image, there is also some subspace in the space of all images where all cat images lie, and a classification NN will identify the subspace and learn a decision boundary that separates it from the rest. However, the notion of a 'cat manifold' in pixel space is not well defined. Surely for any image that represents a cat, I can make a single pixel half a percent more red and it would still be a cat. Also, the CCD chip of the camera with which the cat image was taken had thermal and readout noise, meaning there are noisy pixels in the image that have the wrong colour anyways. On the other hand, a CY manifold is a mathematically well-defined object, and we can find arbitrarily many points on it with arbitrarily high precision.

Let us first discuss the neural PDE setup. Obviously, we cannot take a ruler and measure distances in the six extra dimensions to get some data points for the metric we are trying to fit. However, we can approximate the metric using a numerical algorithm developed by Donaldson [61]. Donaldson addressed the issue by solving a different problem (as mathematicians like to do) of finding a so-called balanced metric. He then showed that in a certain limit the balanced metric becomes the Ricci-flat metric. A practical problem is that this limit is computationally very expensive to take, which is why Donaldson's algorithm scales badly. Nevertheless, one can compute the approximate metric at some points and then use a NN

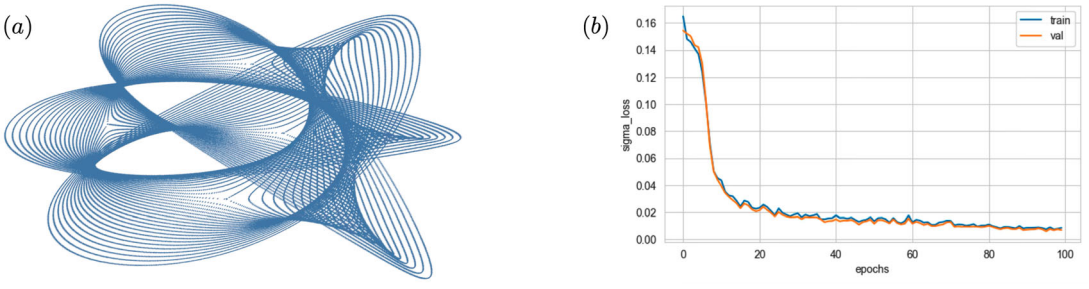

Figure 5. (a) A 2D image of a 3D projection of the extra 6 curled up dimensions (CY) in string theory. We have obtained the point samples by solving the corresponding polynomial equations. (b) Deviation from solving the differential Equation (19) for the PINN that approximates the Ricci-flat metric. The NN has 3 hidden layers with 64 neurones each and is trained with 100 k points for 100 epochs. The loss on the unseen points is the same as on the training points, indicating that the NN is not overfitting.

to interpolate the solution to many more points. This was also used in [62] to approximate the determinant of the metric.

A second approach is to use PINNs and discard Donaldson's algorithm altogether. This means we set up a NN that gets as input a point on the CY and the NN maps it to a real 6×6 matrix that describes the metric at this point (for a fixed choice of coordinate system). As mentioned above, this requires solving a fourth order PDE, meaning one needs 4 derivatives of all 36 outputs of the NN with respect to all 6 inputs, which is computationally intense. One can improve on this in two ways. The first one is to use mathematical insight that was also used to prove the Calabi conjecture. As it turns out, one can rewrite the fourth order PDE in terms of a special second order PDE (of so-called Monge-Ampere type). Moreover, instead of computing the 36 entries of the 6×6 matrix, one can compute a single scalar, from which the metric can be computed by taking more derivatives. However, this scalar function has to satisfy certain symmetry properties, which can be imposed by constructing an equivariant (or invariant) NN with respect to these symmetries, as explained in the introduction.

Armed with this NN, we can use another trick known as 'boosting' in the ML community. Instead of learning the entire metric from scratch, we start from some approximation and the NN only learns a correction to this starting point. There is actually a canonical metric one can use for the extra dimensions, called the Fubini-Study metric g_{FS}. While it is a perfectly fine metric, it is not Ricci-flat, meaning it is not the one that solves string theory's equation of motion. This is why we correct it with a NN to give the Ricci-flat metric g_{CY},

$$g^{ab}_{\text{CY}}(x_1,\dots,x_6) = g^{ab}_{\text{FS}}(x_1,\dots,x_6) + \frac{\partial}{\partial x_a}\frac{\partial}{\partial x_b}\phi_{\text{NN}}(x_1,\dots,x_6). \quad (18)$$

The x_a, $a = 1,\dots,6$ are the points on the CY and ϕ_{NN} is the correction term learned by the NN, from which we compute the metric by taking derivatives. The loss for the PINN is

$$\mathcal{L}_{\text{PINN}} = \frac{1}{N}\sum_{i=1}^{N}\left|\left|\Omega(\vec{x}_i)\right|^2 - \kappa \det\left[g^{a,b}_{\text{FS}}(\vec{x}_i) + \frac{\partial}{\partial x_a}\frac{\partial}{\partial x_b}\phi_{\text{NN}}(\vec{x}_i)\right]\right|^2, \quad (19)$$

where $\vec{x}_i = (x_1,\dots,x_6)_i$, $i = 1,\dots,N$, is a collection of N points on the CY used for training, κ is a constant which we will return to further below, and $|\Omega(\vec{x}_i)|^2$ is a known function of $\vec{x}$ that can be computed just from the input points. We can think of the training process in which we change ϕ as traversing or flowing through the space of metrics, starting from the Fubini-Study metric and ending up at the Ricci-flat metric [63].

Finally, let us discuss neural operators. So far, we have suppressed the fact that the metric depends on parameters, so-called moduli, that determine the shape and size of the curled-up dimensions. This parameter dependence is hidden in κ in (19),[7] much like it was in ω in the harmonic oscillator. What sets the values for these parameters is an active, ongoing research direction in string theory, but in the end they will also be dynamical and change to minimise the energy just as everything else in Physics. However, in the neural PDE or PINN setup, we find solutions at a fixed point in moduli space. In order to adapt the solution to changes in the moduli, we can pass them as inputs to the NN and have the NN interpolate the solution across the moduli space. In the neural PDE setup, this requires running Donaldson's algorithm multiple times, once for each point in the moduli space. In the PINN setup, we simply train with different values of κ and different points as inputs.

5. Interpretable AI

Knowing the metric, or some approximation thereof, allows to compute physical observables that are otherwise inaccessible. While it would be better to know an analytic expression for the metric, this might simply not exist, and we have to resorted to NNs to address this problem. After training, we have an explicit expression for the CY metric, i.e. the NN. So in principle, the CY metric is given by $\tanh(W^{(2)} \cdot \tanh(W^{(1)} \cdot \vec{x} + \vec{b}^{(1)}) + \vec{b}^{(2)})$, or whatever our NN architecture for ϕ_{NN} in (18) is. While this function is fully explicit, it is not human-interpretable. The parametrization does not make the functional dependence on the coordinates (and on the moduli, in case of neural operators) interpretable to a human.

However, there are techniques that allow to elucidate the dependence on the inputs, with the hope that domain experts can interpret this dependence [64]. Finding out which inputs (i.e. features) are relevant for which outputs is known as a feature saliency analysis. There are multiple techniques to assign saliency or importance scores to inputs. A simple approach is called gradient saliency, which was popularised in the context of interpretable AI for Mathematics in [65]. For this, one simply takes the derivative (more precisely the gradient) of the NN with respect to its inputs (we had to do this anyways to solve the PDE). The derivate measures the rate of change of the output with a change of the input. This means that the larger the absolute value of the gradient of output

node j with respect to input feature i, the stronger its dependence on this particular input.

Another simple technique that does not even require taking derivatives and is thus suitable for other ML techniques that do not allow to take a derivative of the algorithm is feature corruption. This means that one takes the trained NN and randomises one of its input features. If the output changes significantly (or at least if the loss goes up significantly), this feature was important for the final prediction of the NN. While there are many ways of corrupting a feature, a common choice is feature permutation, meaning one just picks one of the features, randomly permutes the values across all data in the test set while keeping all other inputs the same, and then runs the NN on this corrupted input. Feature permutation has the advantage that the distribution of data seen by the NN is unchanged, just the relative composition of data in which the feature appears changes. This ensures that the accuracy is not impacted simply from choosing wildly out-of-distribution values with respect to the data with which the NN was trained.

While feature scoring can tell you what the NN bases its predictions on, it does not necessarily tell you how it finds its prediction. This is the realm of symbolic regression: Instead of the deep composition of affine maps and non-linearities, we want to find a function that is expressed in terms of standard functions we have a better feeling for, like polynomials, exponentials and logarithms, and trigonometric functions. Of course, if a domain expert has reason to believe that a certain function might be relevant (for example because the NN is approximating the solution to a hypergeometric differential equation), one can explicitly include these functions as well.

Using a process called feature engineering, one can turn symbolic regression into a simple linear regression problem. To give an example, assume that our NN was trained to learn the function

$$f(x) = x^3 - 17x + 5 + \cos(x). \tag{20}$$

Instead of just providing x as the input feature, we engineer new features based on the old ones, and also provide x^2, x^3, $\sin(x)$ and $\cos(x)$. We then use linear regression, meaning a NN with no hidden layers and no activation functions,

$$f_\theta(x, x^2, x^3, \sin(x), \cos(x)) = W \cdot \vec{x} + b = (w_1\ w_2\ w_3\ w_4\ w_5) \cdot \begin{pmatrix} x \\ x^2 \\ x^3 \\ \sin(x) \\ \cos(x) \end{pmatrix} + b. \tag{21}$$

After training, we will find $W = (-17\ 0\ 1\ 0\ 1)$ and $b = 5$, which gives us the symbolic expression (20). While this works in this simple example, complications arise from composite functions. If the function had been $\cos(x^3 - 17x + 5)$, our feature engineering would have failed. We will explore an ansatz to deal with this next.

5.1. Symbolic regression and Kolmogorov-Arnold networks

One of the main reasons why NNs are black boxes despite us being able to write down the functional form, is the huge number of parameters. Modern NNs have billions or even trillions of parameters. So let us take a step back and ask why we need all these parameters. A (part of the) answer are the neural scaling laws, and the double descent phenomenon discussed in Sections 1 and 2.1, respectively. This means if we could find a NN architecture that

- can approximate any function,
- is not prone to overfitting,
- makes the individual functions of which the network is composed of more easily interpretable,
- has neural scaling laws with better exponents,

we could potentially use a much smaller network with much fewer parameters that is still performing well, but can be interpreted. This is the idea of Kolmogorov-Arnold networks which we proposed in 2024 [66]. Let us go through the different points in turn.

First, if we change the architecture, we need to make sure that we still have something akin to the Universal Approximation Theorem to ensure that the function we want to learn can actually be expressed in the way we set up the algorithm. For this, we use the Kolmogorov-Arnold representation theorem, which states that any function can be written in terms of sums of unary functions (meaning functions that just take a single variable). This means that sums are the only binary operation needed. For example, we can write

$$f(x, y) = (x + y)y = \exp[\log(x + y) + \log(y)], \tag{22}$$

which expresses the product as a sum. However, this means that we need to compose two unary functions (exp and log in this example), so we need to build deep function compositions. This motivates building a computational graph where we apply unary functions at the edges and then sum up the outcome of all these functions at the nodes. Deeper functions are then built from deeper computational graphs. We present the computational graph for (22) in Figure 6. We call this computational graph a Kolmogorov-Arnold Network (KAN).

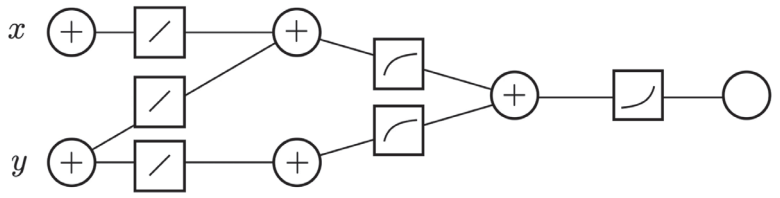

Figure 6. KAN representation of Equation (22). In the first layer we form the pairs $1x + 1y$ and $0x + 1y$. So the edges are just the identity function, and we omit the edge with the zero function, In the next step, the edges apply the log and sums them. In the last step, the edge applies the exponential function.

The next question we need to address is how to approximate the unary functions on the edges of the KAN. One possibility would be to fit a polynomial or a NN to the function. However, this introduces problems. As argued when we discussed the double descent, fitting a polynomial suffers from poor generalisation: One needs enough parameters to make the polynomial general enough, but then it wildly oscillates in between the fitted data points, giving a large error on unseen data. If we fit an overparameterized NN, we can avoid that, but then we have again the problem of non-interpretability. Instead, we choose to fit (cubic) splines. These are piecewise cubic functions that interpolate between the data points. Being only cubic, they cannot oscillate wildly between the training points, which prevents overfitting while only needing very few parameters. Their graph can be analysed visually to see whether the spline resembles any analytic function such as a parabola, a logarithm, or a cosine, cf. Figure 6. If so, we can replace the spline by this function and obtain a symbolic expression given by the composition of sums of analytic functions. The reason why one chooses cubic splines rather than something of even lower degree is that one wants enough parameters to ensure that the cubic polynomial goes through the two fixed endpoints, and that the derivative at these points matches with the derivative of the neighbouring splines, such that we get a continuous, differentiable function.

Since KANs have much fewer parameters N, one might worry that their error, which should drop like $N^{-\alpha}$ according to neural scaling laws, is large. However, if the neural scaling law of KANs have larger exponents α, we can compensate for the smaller N. As mentioned above, the NN scaling laws are not fully understood, but one proposal is that $\alpha = 1/d$, where d is the 'intrinsic dimensionality of the data manifold', meaning the dimension of the subspace within the entire input space on which the relevant features live. As mentioned in Section 4, this is not terribly well defined for real world data but can be made precise for mathematical features/inputs. In any case, it illustrates a problem known as the curse of dimensionality: If we have a high-dimensional input space, α will be small, and the error will go down much slower than for larger α. Using the approximation theory for splines (as well as emperical studies), one can show that for KANs with cubic splines, $\alpha = 1/4$, independent of the number (or dimension) of input features. This means that KANs can converge much faster to smaller errors even with smaller numbers of parameters.

Finally, we want to mention another advantage of KANs as compared to NNs. KANs are 'local' or 'sparse' in the sense that each input enters a single unary function, and this function is built out of piecewise cubic polynomials. This means that adjusting the function to data in a given range requires only updating a single spline element, while leaving the rest of the learned function intact. This is in stark contrast to (dense or fully connected) feed-forward NNs, where all nodes of layer i enter into all nodes of layer $i+1$ as in Figure 3. This leads to 'non-locality', meaning we cannot adjust to learn features locally. Thus, adjusting the function may overwrite good fits that were already learned in a completely different region of input feature space, a phenomenon called catastrophic forgetting. Sparse NNs are an interesting subject in their own right. If one can successfully sparsify or prune a huge NN, one can reduce computation time, energy consumption, and memory footprint, which is necessary if we want to include NNs on chips to be placed in detectors, smartphones, cars, etc.

5.2. Reinforcement learning for problems in physics and mathematics

The final approach for AI for theoretical Physics and Mathematics that we want to discuss is reinforcement learning. Here, we give up on trying to interpret *how* the NN made its decision but rather try to understand *why* it did what it did. After all, when we study a proof in Mathematics, we do not ask how the brain of the mathematician works and what caused them to come up with the individual steps in the proof, but rather we try to understand how the next step follows from the previous one. In terms of reinforcement learning, this means we train an agent

to find a sequence of moves and then try to understand what the NN is trying to achieve. This is done by studying episodic rollouts, which means one follows the sequence of states visited in the decision tree from the start state to the terminal state. In games like Go or chess, expert players can deduce the tactics learned by the agent this way, see e.g. [67].

Even if one cannot understand why the NN chose a particular sequence of actions, RL is still good at producing provably correct results in this way, and this is what we use when we 'gamify' science. In particular, we want to use RL for decision problems of the form

PROPERTY X: Does object *O* have property *X*.

This is (intentionally) vague, meaning the technique can in principle be applied to any question of this form. We set up an RL algorithm that decided the 'yes' instances and do not make any assumption about the behaviour of the algorithm on the 'no' instances. The idea is to train an RL algorithm to take a series of actions that manipulate the object *O* (so *O* and all its manipulated versions are the state space) until it is manifest that *O* has property *X*. In this case, the algorithm halts, returns `true` and the truth certificate, which is just the series of manipulations it applied to *O*. Note that despite us not being able to interpret the NN and the NN behaving stochastically while traversing the search tree, the result is nevertheless rigorous and provably correct to the extent that it would satisfy any mathematician. Although we said we make no assumption about the behaviour for objects *O* that do not have property *X*, we essentially know what the algorithm is doing – it will still manipulate *O* in a way that maximises the accumulated rewards.

This algorithm seems to be ideally suited for NP problems: There is an exponentially large search tree, but the RL algorithm finds a path through it to a terminal state, which we can verify to be correct. However, this does not mean that NNs can solve NP-complete problems in polynomial time. First, it could take exponentially many states to reach the terminal state. Second, and much more likely, the algorithm does not find the terminal state of all 'yes' instances, i.e. it fails to produce a truth certificate for some instances that have one, so it is not perfect. In practise, one can try to combat this by running the (stochastic) search multiple times. While this idea was, to the best of my knowledge, first implemented to solve Diophantine equations in string theory in [68], we will illustrate the procedure using a simpler example taken from knot theory [69]. We want to answer the question:

UNKNOT: Given a knot, can it be fully unknotted, i.e. manipulated without cutting and gluing it to just be a standard circle without any knots on it.[8]

This is the so-called unknot decision problem, which is known to be in NP, but not known to be in P. An approach to gamifying the problem is the following: One moves strands of the knot around until the knot is untied. Formulated this way, the problem sounds like it has a continuous action space (since one can move the strands continuously to any position in $\mathbb{R}^3$). However, as it turns out, the precise position of the strands does not matter for this question (or for describing the knot as a mathematical object); all that matters is whether the strand is weaved over or under another strand (when looking from any arbitrary position in $\mathbb{R}^3$), and the order of over- and undercrossings, see Figure 7(a). Starting from a random point on the (closed) string that has been knotted up, we can characterise the knot by keeping track of all crossings and whether the strand goes over or under another strand at these crossings.

The next question one might worry about (probably if and only if one is a mathematician) is whether this problem is trivial: What if all knots can be unknotted? Reidemeister was worried about that, and he proved that this is not the case: There exist infinitely many non-trivial knots. He proved this by showing that a given knot can be manipulated to give any equivalent knot by three different types of manipulations, known as the three Reidemeister moves [70], see Figure 7(b).

Now it is clear how to formulate the problem as a one-player game or a discrete search problem, which we solve with RL. We use the following setup:

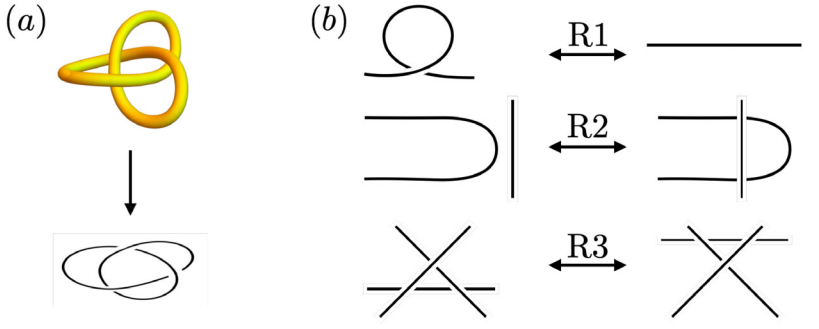

Figure 7. (a) A knot (the so-called trefoil) and its projection. (b) The three Reidemeister moves.

- **State space:** The set of all knots. Note that this space is infinite, but finite if we fix the maximal number of crossings that can be in a knot.
- **Terminal states:** The unknot.
- **Action space:** The set of Reidemeister moves, which is a rather small set for any given knot.
- **Reward function:** We only punish the agent, with punishment proportional to the number of crossings in the knot. The unknot is the unique knot with no crossings, and reaching it is the only way for the agent to avoid punishment.

First note that the game is similar in spirit to Go, in that it has a huge state space but a rather small action space. We also want to comment on the reward function. First, rewards are not sparse, meaning we do not have to play the game all the way to the end to assign a reward. Instead, we can reward each action, depending on whether it increased or decreased the number of crossings in the knot. This is good, since, in contrast to chess or Go, our game is not guaranteed to ever end: it will never end on the no-instances, meaning on the knots that simply cannot be unknotted. The agent would still try to simplify the knot as much as possible to minimise the punishments it receives each time and then stall by doing moves of type R3 in Figure 7, which do not increase the number of crossings and with it the punishments received. In practise, one can prevent the NN from stalling in this way by declaring a move illegal if it leads to a state that the agent encountered before (in chess, there is a similar rule that ends the game in a draw upon threefold repetition of a position). Since we are trying to solve a Markov decision process, all that matters is the state itself, not how it was reached, so forbidding to visit the same state twice cannot impact whether or not the agent can find a solution. A second thing to note is that there exist so-called hard unknots, which are knots that can be fully unknotted, but only by temporarily increasing the number of crossings, and hence the amount of punishment incurred by our reward function. The agent will still choose to do this, since it is trained to minimise the total amount of punishments it receives. This illustrates why it is important to train the algorithm to maximise its long-term return rather than its immediate rewards.

Notes

1. Hinton also popularised the backpropagation algorithm [71], with which essentially all NNs are trained nowadays, as well as ImageNet [72], a NN architecture that lead to a breakthrough in image classification. With over 900k citations, Hinton might become the first 'citation millionaire' in the world.
2. The particular parameterisation in (5) is still somewhat interpretable, but this ceases to be the case for deeper networks.
3. Although Nature also fails sometimes, for example for glass. Glass is a liquid, but its energy landscape has so many local minima in which the configuration gets stuck constantly, which means glass is flowing very slowly.
4. This is essentially how chess engines (used to) work; they just looked ahead and simulated moves farther ahead than humans could. This is impossible for Go, which is why it took another 20 years after Deep Blue [73] and a paradigm shift until computers could beat humans.
5. Since many cryptography algorithms (and cryptocurrencies) rely on the fact that we cannot quickly factor integers (as I said, most assume P≠NP), we need to rethink these once Quantum Computers become powerful enough to break their encryption.
6. Note that for training via backpropagation, we also need to take derivatives of the NN with respect to its parameters.
7. It is of course also hidden in the input points: As the shape of the CY changes, the points on the CY will change as well.
8. In contrast to a knot that one would tie into shoe laces, for example, knots in Mathematics always have the two endpoints glue together as in Figure 7(a).

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

The work of FR is supported by the NSF grants PHY-2210333 and PHY-2019786 (The NSF AI Institute for Artificial Intelligence and Fundamental Interactions), as well as startup funding from Northeastern University.

Notes on contributor

Fabian Ruehle is an Assistant Professor at Northeastern University. He obtained his PhD from the University of Bonn in 2013. He held Postdoctoral Research Positions at DESY Hamburg, the University of Oxford, and CERN before joining Northeastern University in 2021. He is also a Senior Investigator of the NSF AI Institute for Artificial Intelligence and Fundamental Interactions (IAIFI). His research lies at the intersection of theoretical physics, mathematics, and computer science, where he has mostly worked on string theory, low-dimensional topology, and machine learning, respectively.

ORCID

Fabian Ruehle http://orcid.org/0000-0002-8409-9823

References

[1] Cybenko G. Approximations by superpositions of sigmoidal functions. Math Control Signals Syst. 1989;2: 303–314. doi: 10.1007/BF02551274

[2] Leshno M, Lin VY, Pinkus A, et al. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Netw.

1993;6(6):861–867. doi: 10.1016/S0893-6080(05)80131-5 Available from: http://www.sciencedirect.com/science/article/pii/S0893608005801315
[3] Choromanska A, Henaff M, Mathieu M, et al. The loss surfaces of multilayer networks. In: Lebanon G, Vishwanathan SVN, editors. Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics; (Proceedings of machine learning research; Vol. 38); 09–12 May; San Diego, California, USA. PMLR; 2015. p. 192–204.
[4] Advani M, Ganguli S. Statistical mechanics of optimal convex inference in high dimensions. Phys Rev X. 2016;6:031034. doi: 10.1103/PhysRevX.6.031034
[5] Sagun L, Evci U, Guney VU, et al. Empirical analysis of the hessian of over-parameterized neural networks; 2017. Available from: arXiv preprint; arXiv:1706.04454.
[6] Baity-Jesi M, Sagun L, Geiger M, et al. Comparing dynamics: deep neural networks versus glassy systems. In: Dy J, Krause A, editors. Proceedings of the 35th International Conference on Machine Learning; (Proceedings of machine learning research; Vol. 80); 10–15 Jul. PMLR; 2018. p. 314–323.
[7] Zdeborová L. Understanding deep learning is also a job for physicists. Nat Phys. 2020;16(6):602–604. doi: 10.1038/s41567-020-0929-2
[8] Hopfield JJ. Neural networks and physical systems with emergent collective computational abilities. Proc Natl Acad Sci USA. 1982;79(8):2554–2558. doi: 10.1073/pnas.79.8.2554
[9] Ackley DH, Hinton GE, Sejnowski TJ. A learning algorithm for Boltzmann machines. Cogn Sci. 1985;9(1):147–169.
[10] Thomas N, Smidt T, Kearnes S, et al. Tensor field networks: rotation-and translation-equivariant neural networks for 3d point clouds; 2018.
[11] Geiger M, Smidt T. e3nn: Euclidean neural networks; 2022. Available from: https://arxiv.org/abs/2207.09453
[12] Hestness J, Narang S, Ardalani N, et al. Deep learning scaling is predictable, empirically; 2017.
[13] Bahri Y, Dyer E, Kaplan J, et al. Explaining neural scaling laws. Proc Natl Acad Sci. 2024;121(27):e2311878121. doi: 10.1073/pnas.2311878121
[14] Maloney A, Roberts DA, Sully J. A solvable model of neural scaling laws; 2022.
[15] Jacot A, Gabriel F, Hongler C. Neural tangent kernel: convergence and generalization in neural networks. In: Advances in neural information processing systems (NeurIPS). Vol. 31; 2018. Available from: https://arxiv.org/abs/1806.07572
[16] Lee J, Xiao L, Schoenholz SS, et al. Wide neural networks of any depth evolve as linear models under gradient descent; 2019. Available from: https://arxiv.org/abs/1902.06720
[17] Arora S, Du SS, Hu W, et al. On exact computation with infinite-width neural networks; 2019. Available from: https://arxiv.org/abs/1904.11955.
[18] Yaida S. Non-gaussian processes and neural networks at finite widths; 2019. Available from: arXiv:1910.00019.
[19] Halverson J, Maiti A, Stoner K. Neural networks and quantum field theory. Mach Learn Sci Tech. 2021;2(3):035002. doi: 10.1088/2632-2153/abeca3
[20] Yaida S. Nonperturbative effects in neural networks at finite widths; 2020. Available from: arXiv:2012.03219.
[21] Roberts DA, Yaida S, Hanin B. The principles of deep learning theory. Cambridge: Cambridge University Press; 2022.
[22] Halverson J. TASI lectures on physics for machine learning; 2024.
[23] Ho J, Jain A, Abbeel P. Denoising diffusion probabilistic models. In: Advances in neural information processing systems. Vol. 33. Curran Associates, Inc.; 2020. p. 6840–6851.
[24] Knott CG. Life and scientific work of peter guthrie tait. Cambridge: Cambridge University Press; 1911. Quote from undated letter from Maxwell to Tait.
[25] Ruehle F. Data science applications to string theory. Phys Rept. 2020;839:1–117. doi: 10.1016/j.physrep.2019.09.005
[26] Hayou S, Doucet A, Rousseau J. On the impact of the activation function on deep neural networks training. In: Chaudhuri K, Salakhutdinov R, editors. Proceedings of the 36th International Conference on Machine Learning; (Proceedings of machine learning research; Vol. 97). PMLR; 2019. p. 2672–2680.
[27] Belkin M, Hsu D, Ma S, et al. Reconciling modern machine learning and the bias-variance tradeoff. Proc Natl Acad Sci (PNAS). 2019;116(32):15849–15854. doi: 10.1073/pnas.1903070116Available from: https://www.pnas.org/content/116/32/15849
[28] Nakkiran P, Kaplun G, Bansal Y, et al. Deep double descent: where bigger models and more data hurt. J Stat Mech: Theory Exp. 2021;2021(12):124003. doi: 10.1088/1742-5468/ac3a74Available from: https://arxiv.org/abs/1912.02292
[29] Bertschinger N, Natschläger T. Real-time computation at the edge of chaos in recurrent neural networks. Neural Comput. 2004;16(7):1413–1436. doi: 10.1162/089976604323057443
[30] Poole B, Lahiri S, Raghu M, et al. Exponential expressivity in deep neural networks through transient chaos. In: Advances in neural information processing systems; 2016. p. 3360–3368.
[31] Polyak BT. Some methods of speeding up the convergence of iteration methods. USSR Comput Math & Math Phys. 1964;4(5):1–17. doi: 10.1016/0041-5553(64)90137-5
[32] Betancourt M. A conceptual introduction to hamiltonian Monte Carlo; 2017. Available from: arXiv preprint; arXiv:170102434.
[33] De Luca GB, Silverstein E. Born-infeld (BI) for AI: energy-conserving descent (ECD) for optimization. In: Chaudhuri K, Jegelka S, Song L, et al., editors. Proceedings of the 39th International Conference on Machine Learning; (Proceedings of machine learning research; Vol. 162). PMLR; 2022. p. 4918–4936. Available from: https://proceedings.mlr.press/v162/de-luca22a.html
[34] Chaudhari P, Choromanska A, Soatto S, et al. Entropy-SGD: biasing gradient descent into wide valleys. In: International Conference on Learning Representations (ICLR); 2017. Available from: https://arxiv.org/abs/1611.01838.
[35] Welling M, Teh YW. Bayesian learning via stochastic gradient langevin dynamics. In: Proceedings of the 28th International Conference on Machine Learning (ICML); 2011. p. 681–688. Available from: https://proceedings.mlr.press/v28/welling13.html.

[36] Adachi SH, Henderson MP. Application of quantum annealing to training of deep neural networks; 2015. Available from: https://arxiv.org/abs/1510.06356
[37] Kieferová M, Wiebe N. Tomography and generative training with quantum Boltzmann machines. Phys Rev A. 2017;96(6):062327. doi: 10.1103/PhysRevA.96.062327
[38] Higham CF, Bedford A. Quantum deep learning by sampling neural nets with a quantum annealer. Sci Rep. 2023;13(1):3939. doi: 10.1038/s41598-023-30910-7
[39] Silver D, Huang A, Maddison CJ, et al. Mastering the game of go with deep neural networks and tree search. Nature. 2016;529:484–489. doi: 10.1038/nature16961Available from: https://www.nature.com/articles/nature16961
[40] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of go without human knowledge. Nature. 2017;550:354–359. doi: 10.1038/nature24270Available from: https://www.nature.com/articles/nature24270
[41] Cook SA. The complexity of theorem-proving procedures. In: Association for computing machinery; STOC '71. New York (NY), USA; 1971. p. 151–158. doi: 10.1145/800157.805047
[42] Levin L. Universal sequential search problems. Probl Peredachi Inf. 1973;9:115–116.
[43] Turing AM. On computable numbers, with an application to the entscheidungsproblem. Proc Lond Math Soc. 1936;42:230–265.
[44] Matiyasevich YV. Enumerable sets are diophantine. Dokl Akad Nauk SSSR. 1970;191:279–282. English translation in *Soviet Mathematics*, 11(2):354–357.
[45] Hilbert D. Mathematical problems. Bull Am Math Soc. 1902;8(10):437–479. doi: 10.1090/bull/1902-08-10
[46] Denef F, Douglas MR. Computational complexity of the landscape: part i. Ann Phys (NY). 2007;322(5):1096–1142. doi: 10.1016/j.aop.2006.07.013Available from: https://www.sciencedirect.com/science/article/pii/S0003491606001382
[47] Halverson J, Plesser M, Ruehle F, et al. Kähler Moduli stabilization and the propagation of decidability. Phys Rev D. 2020;101(4):046010. doi: 10.1103/PhysRevD.101.046010
[48] Denef F, Douglas MR, Greene B, et al. Computational complexity of the landscape II – cosmological considerations. Annals Phys. 2018;392:93–127. doi: 10.1016/j.aop.2018.03.013
[49] Halverson J, Ruehle F. Computational complexity of vacua and near-Vacua in field and string theory. Phys Rev D. 2019;99(4):046015. doi: 10.1103/PhysRevD.99.046015
[50] Aaronson S. The complexity of quantum states and transformations: from quantum money to black holes; 2016. Available from: https://arxiv.org/abs/1607.05256
[51] Shor PW. Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS); 1994. p. 124–134.
[52] Calabi E. On Kähler manifolds with vanishing canonical class. Algebraic geometry and topology A symposium in honor of S Lefschetz; 1957. p. 78–89.
[53] Yau ST. Calabi's conjecture and some new results in algebraic geometry. Proc Nat Acad Sci. 1977;74:1798–1799. doi: 10.1073/pnas.74.5.1798
[54] Anderson LB, Gerdes M, Gray J, et al. Moduli-dependent Calabi-Yau and SU(3)-structure metrics from machine learning. JHEP. 2021;05:013. doi: 10.1007/JHEP05(2021)013
[55] Douglas MR, Lakshminarasimhan S, Qi Y. Numerical Calabi-Yau metrics from holomorphic networks; 2020.
[56] Jejjala V, Mayorga Pena DK, Mishra C. Neural network approximations for Calabi-Yau metrics. JHEP. 2022;08:105. doi: 10.1007/JHEP08(2022)105
[57] Larfors M, Lukas A, Ruehle F, et al. Numerical metrics for complete intersection and Kreuzer–Skarke Calabi–Yau manifolds. Mach Learn Sci Tech. 2022;3(3):035014. doi: 10.1088/2632-2153/ac8e4e
[58] Chen RTQ, Rubanova Y, Bettencourt J, et al. Neural ordinary differential equations. In: Advances in neural information processing systems (NeurIPS); 2018.
[59] Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. J Comput Phys. 2019;378:686–707. doi: 10.1016/j.jcp.2018.10.045
[60] Li Z, Kovachki N, Azizzadenesheli K, et al. Fourier neural operator for parametric partial differential equations. In: International Conference on Learning Representations (ICLR); 2021.
[61] Donaldson SK. Some numerical results in complex differential geometry; 2005.
[62] Ashmore A, He YH, Ovrut BA. Machine learning Calabi–Yau metrics. Fortsch Phys. 2020;68(9):2000068. doi: 10.1002/prop.v68.9
[63] Halverson J, Ruehle F. Metric flows with neural networks. Mach Learn Sci Tech. 2024;5(4):045020. doi: 10.1088/2632-2153/ad8533
[64] Gukov S, Halverson J, Ruehle F. Rigor with machine learning from field theory to the Poincaré conjecture. Nature Rev Phys. 2024;6(5):310–319. doi: 10.1038/s42254-024-00709-0
[65] Davies A, Veličković P, Buesing L, et al. Advancing mathematics by guiding human intuition with ai. Nature. 2021;600(7887):70–74. doi: 10.1038/s41586-021-04086-x
[66] Liu Z, Wang Y, Vaidya S, et al. KAN: Kolmogorov-Arnold networks; 2024.
[67] Sadler M, Regan N. Game changer: Alphazero's groundbreaking chess strategies and the promise of ai. New In Chess; 2019.
[68] Halverson J, Nelson B, Ruehle F. Branes with brains: exploring string vacua with deep reinforcement learning. JHEP. 2019;06:003. doi: 10.1007/JHEP06(2019)003
[69] Gukov S, Halverson J, Ruehle F, et al. Learning to unknot. Mach Learn Sci Tech. 2021;2(2):025035. doi: 10.1088/2632-2153/abe91f
[70] Reidemeister K. Elementare begründung der knotentheorie. Abh Math Semin Univ Hambg. 1927;5(1):24–32. doi: 10.1007/BF02952507
[71] Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986;323(6088):533–536. doi: 10.1038/323533a0
[72] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In: Pereira F, Burges CJC, Bottou L, et al., editors. Advances in neural information processing systems (NIPS); Vol. 25. Curran Associates; 2012. p. 1097–1105.
[73] Hsu FH. Behind deep blue: building the computer that defeated the world chess champion. Princeton (NJ): Princeton University Press; 2002.