BULLETIN (New Series) OF THE
AMERICAN MATHEMATICAL SOCIETY
https://doi.org/10.1090/bull/1829
Article electronically published on February 15, 2024

IS DEEP LEARNING A USEFUL TOOL FOR THE PURE MATHEMATICIAN?

GEORDIE WILLIAMSON

Abstract. A personal and informal account of what a pure mathematician might expect when using tools from deep learning in their research.

1. Introduction

Over the last decade, deep learning has found countless applications throughout industry and science. However, its impact on pure mathematics has been modest. This may be surprising, as some of the tasks at which deep learning excels—like playing the board game Go or finding patterns in complicated structures—appear to present similar difficulties to problems encountered in research mathematics. On the other hand, the ability to *reason*—probably the single most important defining characteristic of mathematical enquiry—remains a central unsolved problem in artificial intelligence. Thus, mathematics can be seen as an important litmus test as to what modern artificial intelligence can and cannot do.

There is great potential for interaction between mathematics and machine learning.[1] However, there is also a lot of hype, and it is easy for the mathematician to be put off. In my experience, it remains hard to use deep learning to aid my mathematical research. But it is possible. One also has the sense that the potential, once the right tools have been uncovered, is significant.

This is a very informal survey of what a working mathematician might expect when using the tools of deep learning on mathematics problems. I outline some of the beautiful ideas behind deep learning. I also give some practical hints for using these tools. I finish with some examples where deep learning has been used productively in pure mathematics research. (I hope it goes without saying that the impact of deep learning on applied mathematics has been enormous.)

Finally, in my experience, the more one uses the tools of deep learning, the more difficult it becomes not to ask oneself foundational questions about why they work. This raises an entirely different set of questions. Although fascinating, the mathematical theory of deep learning is not the focus here.

Received by the editors May 25, 2023.
2020 *Mathematics Subject Classification.* Primary 68T07, 05E10.

I am a pure mathematician, working mostly in geometric representation theory and related fields. I began an ongoing collaboration with DeepMind in 2020 on possible interactions of machine learning and mathematics, and have been fascinated by the subject ever since.

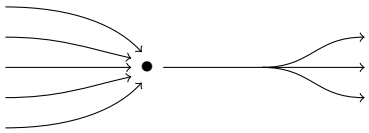
[1] In 1948, Turing [Tur48, §6] identifies games, mathematics, cryptography and language translation and acquisition as five "suitable branches of thought" in which experimentation with machine intelligence might be fruitful.

Remark 1.1. The elephant in the room of any discussion of deep learning today is the recent success of ChatGPT and other large language models. The internet is full of examples of ChatGPT doing both very well and very poorly on reasoning and mathematics problems. It seems likely that large language models will be able to interact well with proof assistants in the near future (see, e.g., [HRW+21, JWZ+23]). It is also likely that a greater role will be played in mathematics research by very large models, possibly with emergent capabilities ("foundation models" in the language of the excellent Bommasani et al. [BHA+21]). The impacts of such developments on mathematics are difficult to predict. In this article I will ignore these questions entirely. Thus I will restrict myself to situations in which deep learning can be used by mathematicians without access to these large models.

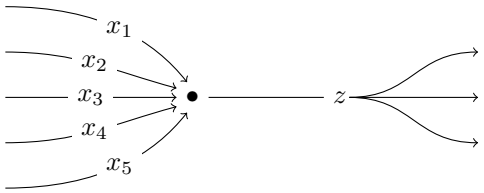
2. What is a neural network?

Artificial neural networks emulate the biological neural networks present in the brains of humans and other animals. Typically, this emulation takes place on a computer. The idea of doing so is very natural. See [MP43, Tur48] for remarkable early accounts.

A cartoon picture of a neuron imagines it as a unit with several inputs and a single output, which may then be connected to other neurons:



Neurons "fire" by emitting electrical charge along their axon. We may encode the charges arriving along each node by a real number, in which case the charge emitted by a neuron is given by



$$z = f\left(\sum x_i\right),$$

where f is a (typically monotone increasing and nonlinear) *activation function*. Soon we will assume that our activation function is fixed;[2] however, at this level of precision the reader is encouraged to imagine something like $f(x) = \tanh(x)$. The activation function is meant to model the nonlinear response curves of neurons to stimuli. For example, some neurons may not fire until a certain charge is reached at their source.[3]

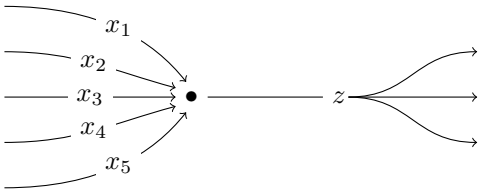
Another important feature of neurons is that their firing may be excitatory or inhibitory of downstream neurons to varying degrees. In order to account for this,

[2] and equal to "ReLU": $f(x) = \max(0, x)$

[3] In biological neural nets there is typically large variation in the responses of neurons to stimuli depending on where they are in the brain (see, e.g., [HW62]). This is one of the many features of biological neural nets that is usually ignored when building artificial neural networks.

one allows modification of the input charges via weights (the w_i):

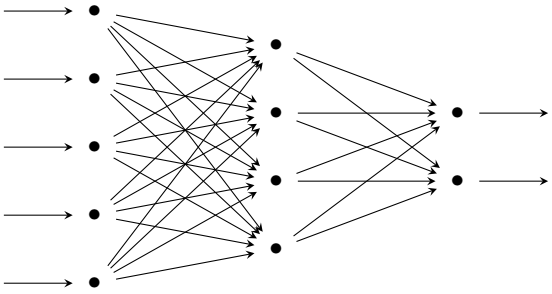
(1)



$$z = f\left(\sum w_i x_i\right).$$

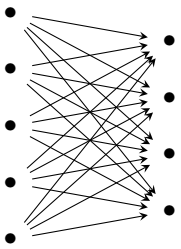
Thus positive and negative weights correspond to excitatory and inhibitory connections respectively.

Having settled on a crude mathematical model of a single neuron, we may then assemble them together to form a *neural network*:



Implicit in this picture is the assignment of a weight to each edge. Thus our neural network yields a function which takes real valued inputs (5 in the above picture), and outputs real values (2 above), via repeated application of (1) at each node.

This is a good picture for the layperson to have in mind. It is useful to visualize the complex interconnectedness present in artificial neural networks, as well as the locality of the computation taking place. However, for the mathematician, one can explain things a little differently. The configuration



is simply a complicated way of drawing a 5×4 matrix. In other words, we can rewrite our neural network above economically in the form

$$\mathbb{R}^5 \xrightarrow{W_1} \mathbb{R}^4 \xrightarrow{f} \mathbb{R}^4 \xrightarrow{W_2} \mathbb{R}^2 \xrightarrow{f} \mathbb{R}^2,$$

where the W_i are linear maps determined by matrices of weights, and f is shorthand for the coordinatewise application of our activation function f.

For the purposes of this article, a *vanilla neural network*[4] is a gadget of the form

$$\mathbb{R}^{d_1} \xrightarrow{A_1} \mathbb{R}^{d_2} \xrightarrow{f} \mathbb{R}^{d_2} \xrightarrow{A_2} \mathbb{R}^{d_3} \xrightarrow{f} \mathbb{R}^{d_3} \xrightarrow{A_3} \dots \xrightarrow{f} \mathbb{R}^{d_{\ell-1}} \xrightarrow{A_{\ell-1}} \mathbb{R}^{d_\ell},$$

where A_i are affine linear maps. We refer to $\mathbb{R}^{d_1}, \mathbb{R}^{d_2}, \dots, \mathbb{R}^{d_\ell}$ as the *layers* of the network. In order to simplify the discussion, we always assume that our activation function f is given by ReLU (the "rectified linear unit"), that is,

$$f\left(\sum \lambda_i e_i\right) = \sum \max(\lambda_i, 0) e_i$$

where the e_i are standard basis vectors.

Remark 2.1. We make the following remarks:

(1) The attentive reader might have observed a sleight of hand above, where we suddenly allowed affine linear maps in our definition of a vanilla neural net. This can be justified as follows: In biological neural nets both the charge triggering a neuron to fire, as well as the charge emitted, varies across the neural network. This suggests that each activation function should have parameters, i.e., be given by $x \mapsto f(x + a) + b$ for varying $a, b \in \mathbb{R}$ at each node. Things just got a lot more complicated! Affine linear maps circumvent this issue: by adding the possibility of affine linear maps one gets the same degree of expressivity with a much simpler setup.
(2) We only consider ReLU activation functions below. This is one of the standard choices, and provides a useful simplification. However, one should not forget that it is possible to vary activation functions.
(3) We have tried to motivate the above discussion of neural networks as some imitation of neural activity. It is important to keep in mind that this is a very loose metaphor at best. However, I do find it useful in understanding and motivating basic concepts. For an excellent account along these lines by David Mumford, the reader is referred to [Mum20].
(4) The alert reader will notice that we have implicitly assumed above that our graphs representing neural networks do not have any cycles or loops. Again, this is a simplification, and it is desirable in certain situations (e.g., in recurrent neural networks) to allow loops.

Vanilla neural networks are often referred to as *fully-connected* because each neuron is connected to every neuron in the next layer. This is almost opposite to the situation encountered in the brain, where remarkably sparse neural networks are found. The connection pattern of neurons is referred to the *architecture* of the neural network. As well as vanilla neural networks, important artificial neural network architectures include *convolutional neural networks*, *graph neural networks*, and *transformers*. Constraints of length prohibit us from discussing these architectures in any depth.

Remark 2.2. More generally, the term "neural network" is often used to refer to any program in which the output depends in a smooth way on the input (and thus the program can be updated via some form of gradient descent). We ignore this extra generality here.

[4]One often encounters the term "Multilayer Perceptron" (MLP) in the literature.

3. Motivation for deep learning

In order to understand deep learning, it is useful to keep in mind the tasks at which it first excelled. One of the most important such examples is image classification. For example, we might want to classify hand-written digits:

Here each digit is given as (say) a 28×28 matrix of grayscale values between 0 and 255. This is a task which is effortless for us, but is traditionally difficult for computers.

We can imagine that our brain contains a function which sees a hand-written digit and produces a probability distribution on $\{0, 1, \ldots, 9\}$, i.e., "what digit we think it is".[5] We might attempt to imitate this function with a neural network.

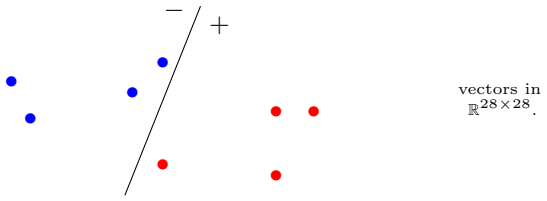
Let us consider a simpler problem in which we try to decide whether a hand-written digit is a 6 or not:

We assume that we have "training data" consisting of images labelled by "6" or "not 6". As a first attempt we might consider a network having a single linear layer:

$$\mathbb{R}^{28\times 28} \xrightarrow{A} \mathbb{R} \xrightarrow{1/(1+e^{-x})} \mathbb{R}.$$

Here A is affine linear, and the second function (the "logistic function")[6] is a convenient way of converting an arbitrary real number into a probability. Thus, positive values of A mean that we think our image is a 6, and negative values of A mean we think it is not.

We will be successful if we can find a hyperplane separating all vectors corresponding to sixes (red dots) from those that do not represent sixes (blue dots):



Of course, it may not be possible to find such a hyperplane. Also, even if we find a hyperplane separating red and blue dots, it is not clear that such a rule would *generalize* to correctly predict whether an unseen image (i.e., image not in our training data) represents a 6 or not. Remarkably, techniques of this form (for example *logistic regression*, *Support Vector Machines (SVMs)*, ...) *do* work in many simple learning scenarios. Given training data (e.g., a large set of vectors

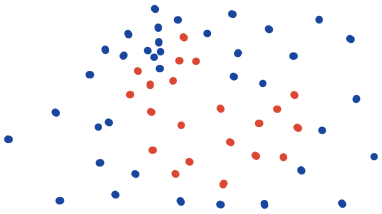
[5] I can convince myself that my brain produces a probability distribution and not a yes/no answer by recalling my efforts to decipher my grandmother's letters when I was a child.

[6] a.k.a. *sigmoid* in the machine learning literature

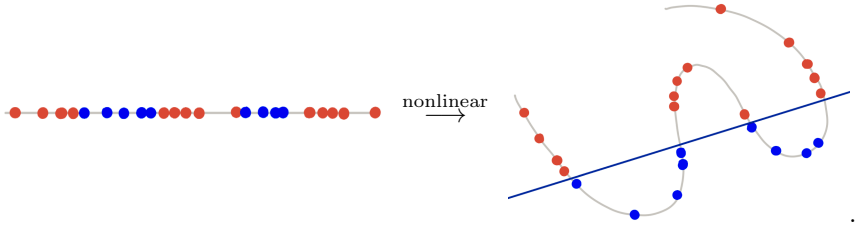
labelled with "yes" and "no") the optimal separating hyperplane may be found easily.[7]

4. What is deep learning?

In many classification problems the classes are not linearly separable.



Linear methods such as SVM can nevertheless still be used in many cases, after application of a suitable *feature map*, namely a (non-linear) transformation whose application on the data makes linear separation of classes possible:



It is on such more difficult learning tasks that deep learning can come into its own. The idea is that successive layers of the neural net transform the data gradually, eventually leading to an easier learning problem.[8]

In the standard setting of supervised learning, we assume the existence of a function

$$\phi : \mathbb{R}^n \to \mathbb{R}^m,$$

and know a (usually large) number of its values. The task is to find a reasonable approximation of ϕ given these known values. (The reader should keep in mind the motivating problem of §3, where one wants to learn a function

$$\phi : \mathbb{R}^{28\times 28} \to \mathbb{R}^{10}$$

giving the probabilities that a certain 28×28-pixel grayscale image represents one of the 10 digits 0, 1, ..., 9.)

We fix a network architecture, which, in our simple setting of a vanilla neural net, means that we fix the number of layers ℓ and layer dimensions $n_2, \dots, n_{\ell-1}$. We then build a neural net (see §2) which serves as our function approximator:

$$\phi_{\approx} : \mathbb{R}^n \xrightarrow{A_1} \mathbb{R}^{n_2} \xrightarrow{f} \mathbb{R}^{n_2} \xrightarrow{A_2} \mathbb{R}^n_3 \xrightarrow{f} \mathbb{R}^{n_3} \xrightarrow{A_3} \dots \xrightarrow{f} \mathbb{R}^{d_{n-1}} \xrightarrow{A_{\ell-1}} \mathbb{R}^m, \tag{2}$$

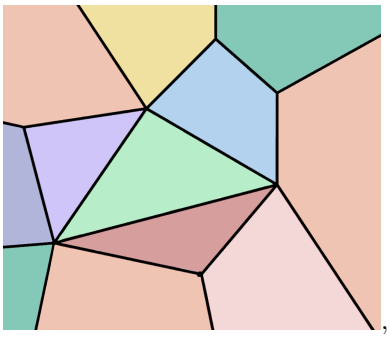
[7]For a striking mathematical example of support vector machines see [HK22], where SVMs are trained to distinguish simple and nonsimple finite groups, by inspection of their multiplication table.

[8]This idea seems to have been present in the machine learning literature for decades, see, e.g., [LBBH98]. It is well explained in [GBC16, §6]. For illustrations of this as well as the connection to fundamental questions in topology, see the work of Olah [Ola14].

To begin with, the affine linear maps A_i are initialized via some (usually random) initialization scheme, and hence the function $\phi_\approx$ output by our neural network will be random and have no relation to our target function ϕ. We then measure the distance between our function $\phi_\approx$ and ϕ via some *loss function* L. (For example, L might be the mean squared distance between the values of ϕ and $\phi_\approx$.)[9] A crucial assumption is that this loss function is differentiable in terms of the weights of our neural network. Finally, we perform gradient descent with respect to the loss function in order to update the parameters in (2) to (hopefully) better approximate ϕ.

In order to get an intuitive picture of what is happening during training, let us assume that $m = 1$ (so we are trying to learn a scalar function), and that our activation functions are ReLU. Thus $\phi_\approx$ is the composition of affine linear and piecewise linear functions, and hence is piecewise linear. As with any piecewise linear function, we obtain a decomposition of $\mathbb{R}^n$ into polytopal regions

(3)



such that $\phi_\approx$ is affine linear on each region. As training progresses, the affine linear functions move in a similar way to the learning of a line of best fit, but more complex since the regions we are dealing with may also move, disappear, or spawn new regions.

Remark 4.1. For an excellent interactive animation of a simple neural network learning a classification task, the reader is urged to experiment with the Tensor Flow Playground [SC]. Karpathy's convolutional neural network demo [Kar] is also illustrative.

Remark 4.2. Some remarks:

(1) Typically, one splits the known values of ϕ into two disjoint sets consisting of *training data* and *validation data*. Steps of gradient descent are only performed using the training data, and the validation data allows us to periodically check whether our model is also making reasonable predictions at points not present in the training data ("validation error"). It is sometimes useful to have an additional set of *test data*, completely unseen during training, which one can use to compute the performance of the trained model ("test error").

[9] There are many subtleties here, and a good choice of loss function is one of them. In my limited experience, neural networks do a lot better learning probability distributions than general functions. When learning probability distributions, *cross entropy* [GBC16, §3.13] is the loss function of choice.

(2) In most applications of machine learning, the training data is enormous and feeding it all through the neural network (2) in order to compute the loss function is unduly expensive. Thus one usually employs *stochastic gradient descent*: at every step the gradient of the loss function is computed using a small random subset (a "minibatch") of the training data.
(3) Using a model with a small number of parameters (as traditionally done in statistics and some ML methods) has advantages for interpretability and computation. It can also help avoid *overfitting*, where the chosen predictor may fit the data set so closely it ends up fitting noise and fails to adequately capture the underlying data-generating process. Deep learning is different in that often there are enough parameters to allow overfitting. What is surprising is that often neural nets generalize well (i.e,. don't overfit) even though they could in principle (this is an enormous subject, see, e.g., [BHMM19]).

5. Simple examples from pure mathematics

It is important to keep in mind that the main motivating applications for deep learning research are very different from those arising in pure mathematics. For example, the "recognize a hand-written digit" function considered in the previous two sections is rather different to the Riemann zeta function![10]

This means that the mathematician wanting to use machine learning should keep in mind that they are using tools designed for a very different purpose. The hype that "neural nets can learn anything" also doesn't help. The following rules of thumb are useful to keep in mind when selecting a problem for deep learning:

(1) *Noise stable.* Functions involved in image and speech recognition motivated much research in machine learning. These functions typically have very high-dimensional input (e.g., $\mathbb{R}^{100\times 100}$ for a square 100×100 grayscale image) and are noise stable. For example, we can usually recognise an image or understand speech after the introduction of a lot of noise. Neural nets typically do poorly on functions which are very noise-sensitive.[11]
(2) *High dimensional.* If one thinks of a neural network as a function approximator, it is a function approximator that comes into its own on high-dimensional input. These are the settings in which traditional techniques like Fourier series break down due to the curse of dimensionality. Deep learning should be considered when the difficulty comes from the dimensionality, rather than from the inherent complexity of the function.
(3) *Unit cube.* Returning to our (unreliable) analogy with biological neural nets, one expects all charges occurring in the brain to belong to some fixed small interval. The same is true of artificial neural networks: they perform best when all real numbers encountered throughout the network from input to output belong to some bounded interval. Deep learning packages are often written assuming that the inputs belong to the unit cube $[0,1]^n \subset \mathbb{R}^n$.

[10] One should keep in mind that neural networks are universal approximators: a large enough neural network can approximate any continuous function accurately [Wik]. However, in practice some functions are much more easily learnt than others.

[11] This point should be read with some caution. For example, evaluation of board positions in Go is not a particularly noise-stable problem.

(4) *Details matter.* Design choices like network architecture and size, initialization scheme, choice of learning rate (i.e., step size of gradient descent), choice of optimizer, etc., matter enormously. It is also important how the inputs to the neural network are encoded as vectors in $\mathbb{R}^n$ (the *representation*).[12] Overcoming these difficulties is best done with a collaborator who has experience in deep learning research and implementation.

With these rules of thumb in mind we will now discuss three examples in pure mathematics.

5.1. **Learning the parity bit.** Consider the parity bit function

$$\begin{aligned} \sigma : \{0,1\}^m &\to \{0,1\} \\ (x_i) &\mapsto \sum_{i=1}^{m} x_i \mod 2. \end{aligned}$$

We might be tempted to use a neural network to try to learn a function

$$\sigma_{\approx} : \mathbb{R}^m \to \mathbb{R},$$

which agrees with σ under the natural embedding $\{0,1\}^m \subset \mathbb{R}^m$.

This is a classic problem in machine learning [MP17, I §3.1]. It generalizes the problem of learning the XOR function (the case $m = 2$), which is one of the simplest problems that cannot be learned without nonlinearities. There exist elegant neural networks extending σ to the unit cube, and given a large proportion (e.g., 50%) of the set $\{0,1\}^m$ a neural network can be trained to express σ [RHW85, pp. 14–16]. However, given only a small proportion of the values of σ (e.g., 10% for $m = 10$) a vanilla neural network will not reliably generalize to all values of σ (for experiments, see [GGW22, 'Playing with parity']).

The issue here is that σ is highly noise sensitive. (Indeed, σ is precisely the checksum of signal processing!) This is an important example to keep in mind, as many simple functions in pure mathematics resemble σ. For an example, see [GGW22, Week 2] where we attempt (without much luck!) to train a neural network to learn the Möbius function from number theory.

5.2. **Learning descent sets.** Consider the symmetric group Σ_n consisting of all permutations of $1, 2, \ldots, n$. Given a permutation we can consider its *left* and *right descent sets*:

$$\mathcal{L}(x) = \{1 \leq i < n \mid x^{-1}(i) > x^{-1}(i+1)\}, \tag{4}$$

$$\mathcal{R}(x) = \{1 \leq i < n \mid x(i) > x(i+1)\}. \tag{5}$$

Obviously, $\mathcal{L}(x^{-1}) = \mathcal{R}(x)$ and $\mathcal{R}(x^{-1}) = \mathcal{L}(x)$. The left and right descent sets are important invariants of a permutation.

It is interesting to see whether a neural network can be trained to learn the left and right descent sets. In other words, we would like to train a neural network

$$\phi_{\approx} : \mathbb{R}^n \to \mathbb{R}^{n-1},$$

[12] It seems silly to have to write that details matter in any technical subject. However, many people I have spoken to are under the false impression that one model works for everything, and that training happens "out of the box" and is easy. For an excellent and honest summary by an expert of the difficulties encountered when training large models, see [Kar19].

which, given the vector $(x(1), x(2), \dots, x(n))$ returns a sequence of $n-1$ probabilities given whether or not $1 \leq i < n$ belongs to the left (resp., right) descent set.

This example is interesting in that (5) implies that the right descent set can be predicted perfectly with a single linear layer. More precisely, if we consider

$$\begin{aligned} \gamma : \mathbb{R}^n &\to \mathbb{R}^{n-1} \\ (v_1, \dots, v_n) &\mapsto (v_1 - v_2, v_2 - v_3, \dots, v_{n-1} - v_n) \end{aligned}$$

then the ith coordinate of γ evaluated on a permutation $(x(1), \dots, x(n))$ is positive if and only if $i \in \mathcal{R}(x)$. On the other hand, it seems much harder to handcraft a neural network which extracts the left descent set from $(x(1), \dots, x(n))$.

This might lead us to guess that a neural network will have a much easier time learning the right descent set than the left descent set. This turns out to be the case, and the difference is dramatic: a vanilla neural network with two hidden layers of dimensions 500 and 100 learns to predict right descent sets for $n = 35$ with high accuracy after a few seconds. Whereas the same network struggles to get even a single correct answer for the left descent set, after significant training![13] It is striking that using permutation matrices as inputs rather than the vectors $(x(1), \dots, x(n))$ gives perfect symmetry in training between left and right.[14] The issue here is the *representation*: how the model receives its input can have a dramatic effect on model performance.

5.3. **Transformers and linear algebra.** Our final example is much more sophisticated, and illustrates how important the choice of training data can be. It also shows how surprising the results of training large neural networks can be.

A *transformer* is a neural network architecture which first emerged in machine translation [VSP+17]. We will not go into any detail about the transformer architecture here, except to say that it is well suited to tasks where the input and output are sequences of tokens ("sequence to sequence" tasks):

More precisely, the input sequence ("xyz") determines a probability distribution over all tokens. We then sample from this distribution to obtain the first token ("a"). Now the input and sequence sampled so far ("xyz" + "a") provides a new distribution over tokens, from which we sample our second token ("b"), etc.

In a recent work [Cha21] Charton trains a transformer to perform various tasks in linear algebra: matrix transposition, matrix addition, matrix multiplication, determination of eigenvalues, determination of eigenvectors, etc. For example, the eigenvalue task is regarded as the "translation":

$$\begin{array}{c} \text{real } 5 \times 5\text{-symmetric matrix} \\ M = (m_{11}, m_{12}, m_{13}, \dots, m_{55}) \end{array} \longrightarrow \boxed{\text{transformer}} \longrightarrow \begin{array}{c} \text{list of eigenvalues} \\ \lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_5. \end{array}$$

Charton considers real symmetric matrices, all of whose entries are signed floating point numbers with three significant figures and exponent lying between -100 and

[13] Decreasing n and allowing longer training suggests that the network can learn the left descent set, but it is *much* harder.

[14] For a colab containing all of these experiments, see [GGW22, Classifying descent sets in S_n].

100.[15] The transformer obtains impressive accuracy on most linear algebra tasks. What is remarkable is that for the transformer the entries of the matrix (e.g., 3.14, -27.8, 0.000132, ...) are simply tokens—the transformer doesn't "know" that 3.14 is close to 3.13, or that both are positive; it doesn't even "know" that its tokens represent numbers!

Another remarkable aspect of this work concerns generalization. A model trained on Wigner matrices (e.g., entries sampled uniformly from $[-10, 10]$) does not generalize well at all to matrices with positive eigenvalues. On the other hand, a model trained on matrices with eigenvalues sampled from a Laplace distribution (which has heavy tails) does generalize to matrices whose eigenvalues are all positive, even though it has not seen a single such matrix during training! The interested reader is referred to Charton's paper [Cha21, Table 12] and his lecture on YouTube [Cha22].

6. Examples from research mathematics

We now turn to some examples where deep learning has been used in pure mathematics research.

6.1. **Counterexamples in combinatorics.** One can dream that deep learning might one day provide a mathematician's "bicycle for the mind": an easy to use and flexible framework for exploring possibilities and potential counterexamples. (I have certainly lost many days trying to prove a statement that turned out to be false, with the counterexample lying just beyond my mental horizon.)

We are certainly not there yet, but the closest we have come to witnessing such a framework is provided in the work of Adam Wagner [Wag21]. He focuses on conjectures of the form: over all combinatorial structures X, an associated numerical quantity Z is bounded by B. He considers situations where there is some simple recipe for generating objects in X, and that the numerical quantity Z is efficiently computable.

For example, a conjecture in graph theory states that for any connected graph G on $n \geq 3$ vertices, with largest eigenvalue λ and matching number μ we have

$$\lambda + \mu - \sqrt{n-1} - 1 \geq 0. \tag{6}$$

(It is not important for this discussion to know what the matching number or largest eigenvalue are!)

Wagner fixes an enumeration $e_1, e_2, \ldots,$ of the edges E in a complete graph on n-vertices. Graphs are generated by playing a single-player game: the player is offered e_1, e_2 etc., and decides at each point whether to accept or reject the edge, the goal being to minimize (6). A move in the game is given by a 01-vector indicating edges that have been taken so far, together with a vector indicating which edge is under consideration. For example, when $n = 4$ the pair $((1,0,1,1,0,0),(0,0,0,0,1,0))$ indicates that edge number 5 is under consideration, and that edges 1, 3, and 4 have already been selected, and 2 rejected. Moves are sampled according to a neural network

$$\mu : \mathbb{R}^E \oplus \mathbb{R}^E \to \mathbb{R}, \tag{7}$$

which (after application of sigmoid) gives the probability that we should take the edge under consideration.

[15] Charton considers various encodings of these numbers via sequences of tokens of various lengths, see [Cha21].

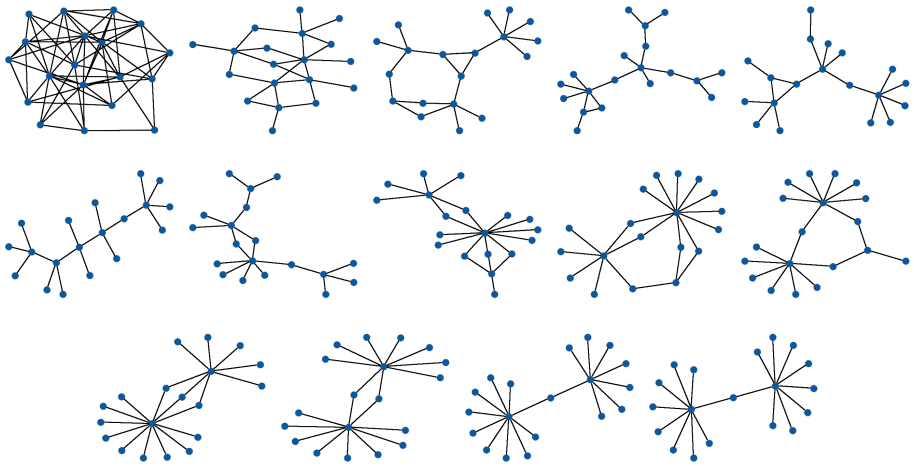


FIGURE 6.1. The evolution of graphs towards Wagner's counterexample, from [Wag21], with permission.

Wagner then employs the *cross entropy method* to gradually train the neural network. A fixed (and large) number of graphs are sampled according to the neural network (7). Then a fixed percentage (say, 10%) of the games resulting in the smallest values of the LHS of (6) are used as training data to update the neural network (7). (That is, we tweak the weights of the neural network to make decisions that result in graphs that are as close as possible to providing a counterexample to (7).) We then repeat. This method eventually finds a counterexample to (6) on 19 vertices. The evolution of graphs sampled from the neural network is shown in Figure 6.1—note how the neural network learns quickly that tree-like graphs do best. Exactly the same method works to discover counterexamples to several other conjectures in combinatorics; see [Wag21].

6.2. **Conjecture generation.** The *combinatorial invariance conjecture* is a conjecture in representation theory which was proposed by Lusztig and Dyer in the early 1980s [Bre04]. To any pair of permutations $x, y \in \Sigma_n$ in the symmetric group one may associate two objects: the *Bruhat graph* (a directed graph); and the *Kazhdan–Lusztig polynomial* (a polynomial in q), see Figure 6.2 for an example of both. The conjecture states that an isomorphism between Bruhat graphs implies equality between Kazhdan–Lusztig polynomials. A more optimistic version of this conjecture asks for a recipe which computes the Kazhdan–Lusztig polynomial from the Bruhat graph. One interesting aspect of this conjecture is that it is (to the best of my knowledge) a conjecture born of pure empiricism.

For the Bruhat graph, the definition is simple, but the resulting graph is complicated. On the other hand, the definition of the Kazhdan–Lusztig polynomial is complicated, however the resulting polynomial is simple. Thus, there is at least a passing resemblance to traditional applications of machine learning, where a simple judgement (e.g., "it's a cat") is made from complicated input (e.g., an array of pixels).

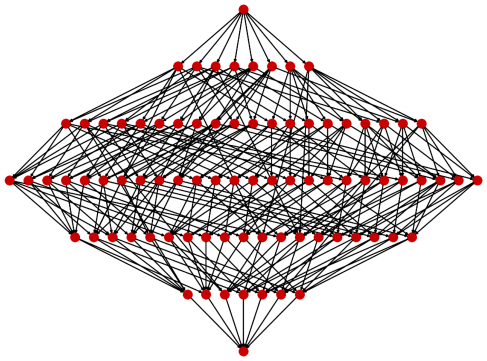


FIGURE 6.2. Bruhat interval and Kazhdan–Lusztig polynomial for the pair of permutations $x = (1, 3, 2, 5, 4, 6)$ and $y = (3, 4, 5, 6, 1, 2)$ in Σ_6, from [BBD+22].

It is natural to use neural networks as a testing ground for this conjecture: if a neural network can easily predict the Kazhdan–Lusztig polynomial from the Bruhat graph, perhaps we can too! We trained a neural network to predict Kazhdan–Lusztig polynomials from the Bruhat graph. We used a neural network architecture known as a graph neural network, and trained the neural network to predict a probability distribution on the coefficients of q, q^2, q^3, and q^4.[16] The neural network was trained on $\approx 20,000$ Bruhat graphs, and achieved very high accuracy ($\approx 98\%$) after less than a day's training. This provides reasonable evidence that there is *some* way of reliably guessing the Kazhdan–Lusztig polynomial from the Bruhat graph.

It is notoriously difficult to go from a trained neural network to some kind of human understanding. One technique to do so is known as *saliency analysis*. Recall that neural networks often learn a piecewise linear function, and hence one can take derivatives of the learned function to try to learn which inputs have the most influence on a given output.[17] In our example, saliency analysis provided subgraphs of the original Bruhat graph which appeared to have remarkable "hypercube"-like structure (see Figure 6.3 and [DVB+21, Figure 5a]). After considerable work this eventually led to a conjecture [BBD+22], which would settle the combinatorial invariance conjecture for symmetric groups if proven, and has stimulated research on this problem from pure mathematicians [GW23, BG23b, BG23a, BM23].

In a parallel development, Davies, Juhász, Lackenby, and Tomasev were able to use saliency analysis to discover a new relationship between the signature and hyperbolic invariants of knots [DJLT22]. The machine learning background of both works is explained in [DVB+21]. It would be very interesting to find further examples where saliency leads to new conjectures and theorems.

6.3. **Guiding calculation.** Another area where deep learning has promise to impact mathematics is in the guiding of calculation. In many settings a computation

[16]The coefficient of q^0 is known to always equal 1. In our training sets no coefficients of q^5 or higher occur.

[17]This technique is often called "vanilla gradient" in the literature. Apparently it is very brittle in real-world applications.

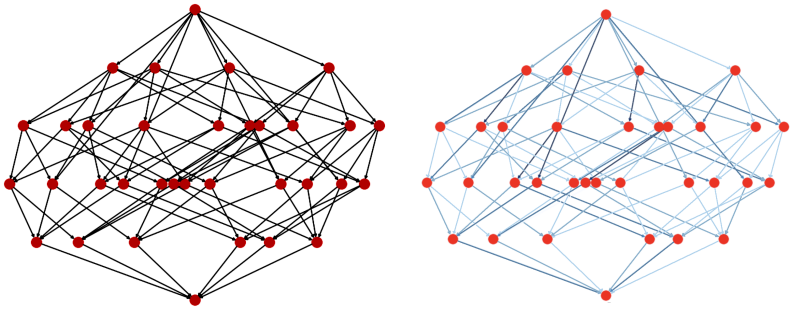


FIGURE 6.3. Bruhat interval pre- and post-saliency analysis.

can be done in many ways. Any choice will lead to a correct outcome, but choices may drastically effect the length of the computation. It is interesting to apply deep learning in these settings, as false steps (which deep learning models are bound to make) effects efficiency but not accuracy.

Over the last three years there have been several examples of such applications. In [PSHL20], the authors use a machine learning algorithm to guide selection strategies in Buchberger's algorithm, which is a central algorithm in the theory of Gröbner bases in polynomial rings. In [Sim21], Simpson uses deep neural networks to simplify proofs in the classification of nilpotent semi-groups. In [HKS22], the authors use a deep neural network to predict computation times of period matrices, and use it to more efficiently compute the periods of certain hypersurfaces in projective space.

6.4. **Prediction.** Due to limitations of space, we cannot begin to survey all the work done in this infant subject. In particular, there has been much work (see, e.g., [BHH$^+$21, BCDL20]) training neural networks to predict difficult quantities in mathematics (e.g., volumes of polytopes, line bundle cohomology, etc.).

7. Conclusion

The use of deep learning in pure mathematics is in its infancy. The tools of machine learning are flexible and powerful, but need expertise and experience to use. One should not expect things to work "out of the box". Deep learning has found applications in several branches of pure mathematics including combinatorics, representation theory, topology and algebraic geometry. Applications so far support the thesis that deep learning most usefully aids the more intuitive ("system 1") parts of the mathematical process: spotting patterns, deciding where counter-examples might lie, choosing which part of a calculation to do next. However, the possibilities do seem endless, and only time will tell.

Acknowledgments

My understanding of this landscape has benefitted enormously from discussions with Charles Blundell, Lars Buesing, Alex Davies, Joel Gibson, Georg Gottwald, Camilo Libedinsky, Sébastien Racaniere, Carlos Simpson, Grzegorz Swirszcz, Petar Veličković, Adam Wagner, Théophane Weber, and Greg Yang. Without their help this journey would have been slower and much more painful. This article is based

on a lecture given at the Fields Institute Symposium on the future of mathematical research, which was held at the instigation of Akshay Venkatesh. I would like to thank the organizers for organising a wonderful and thought-provoking meeting, and in particular Maia Fraser for very useful feedback on an earlier version of this article.

References

[BBD+22] Charles Blundell, Lars Buesing, Alex Davies, Petar Veličković, and Geordie Williamson, *Towards combinatorial invariance for Kazhdan-Lusztig polynomials*, Represent. Theory **26** (2022), 1145–1191, DOI 10.1090/ert/624. MR4510816

[BCDL20] Callum R. Brodie, Andrei Constantin, Rehan Deen, and Andre Lukas, *Machine learning line bundle cohomology*, Fortschr. Phys. **68** (2020), no. 1, 1900087, 14, DOI 10.1002/prop.201900087. MR4059246

[BG23a] Grant Barkley and Christian Gaetz, *Combinatorial invariance for elementary intervals*, Preprint, `arXiv:2303.15577`, (2023).

[BG23b] Grant T. Barkley and Christian Gaetz, *Combinatorial invariance for lower intervals using hypercube decompositions*, Sém. Lothar. Combin. **89B** (2023), Art. 86, 8. MR4659594

[BHA+21] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al., *On the opportunities and risks of foundation models*, Preprint, `arXiv:2108.07258`, (2021).

[BHH+21] Jiakang Bao, Yang-Hui He, Edward Hirst, Johannes Hofscheier, Alexander Kasprzyk, and Suvajit Majumder, *Hilbert series, machine learning, and applications to physics*, Phys. Lett. B **827** (2022), Paper No. 136966, 8, DOI 10.1016/j.physletb.2022.136966. MR4385221

[BHMM19] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal, *Reconciling modern machine-learning practice and the classical bias-variance trade-off*, Proc. Natl. Acad. Sci. USA **116** (2019), no. 32, 15849–15854, DOI 10.1073/pnas.1903070116. MR3997901

[BM23] Francesco Brenti, *Kazhdan-Lusztig polynomials: history problems, and combinatorial invariance*, Sém. Lothar. Combin. **49** (2002/04), Art. B49b, 30. MR2006565

[Bre04] Francesco Brenti, *Kazhdan-Lusztig polynomials: history problems, and combinatorial invariance*, Sém. Lothar. Combin. **49** (2002/04), Art. B49b, 30. MR2006565

[Cha21] François Charton, *Linear algebra with transformers*, CoRR **abs/2112.01898**, (2021).

[Cha22] Francois Charton, *Math with Transformers*, `https://www.youtube.com/watch?v=81o-Uiop5CA`, October 2022, Accessed on 20 March, 2023.

[DJLT22] Alex Davies, András Juhász, Marc Lackenby, and Nenad Tomasev, *The signature and cusp geometry of hyperbolic knots*, Geometry and Topology, (2022).

[DVB+21] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, Marc Lackenby, Geordie Williamson, Demis Hassabis, and Pushmeet Kohli, *Advancing mathematics by guiding human intuition with AI*, Nature **600** (2021), no. 7887, 70–74.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, Adaptive Computation and Machine Learning, MIT Press, Cambridge, MA, 2016. MR3617773

[GGW22] Joel Gibson, Georg Gottwald, and Geordie Williamson, *Machine Learning for the Working Mathematician*, `https://sites.google.com/view/mlwm-seminar-2022`, June 2022, Accessed on 20 March, 2023.

[GW23] Maxim Gurevich and Chuijia Wang, *Parabolic recursions for Kazhdan–Lusztig polynomials and the hypercube decomposition*, Preprint, `arXiv:2303.09251` (2023).

[HK22] Yang-Hui He and Minhyong Kim, *Learning algebraic structures: Preliminary investigations*, International Journal of Data Science in the Mathematical Sciences (2022), 1–20.

[HKS22] Kathryn Heal, Avinash Kulkarni, and Emre Can Sertöz, *Deep learning Gauss-Manin connections*, Adv. Appl. Clifford Algebr. **32** (2022), no. 2, Paper No. 24, 41, DOI 10.1007/s00006-022-01207-1. MR4386394

[HRW+21] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu, *Proof artifact co-training for theorem proving with language models*, Preprint `arXiv:2102.06203` (2021).

[HW62] David H Hubel and Torsten N Wiesel, *Receptive fields, binocular interaction and functional architecture in the cat's visual cortex*, The Journal of physiology **160** (1962), no. 1, 106.

[JWZ+23] Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample, *Draft, sketch, and prove: Guiding formal theorem provers with informal proofs*, 2023.

[Kar] Andrej Karpathy, *Convnet Javascript Demo*, `https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html`, Accessed on 18 March, 2023.

[Kar19] Andrej Karpathy, *A Recipe for Training Neural Networks*, http://karpathy.github.io/2019/04/25/recipe/, Apr 25, 2019, Accessed on 20 March, 2023.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE **86** (1998), no. 11, 2278–2324.

[MP43] Warren S. McCulloch and Walter Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bull. Math. Biophys. **5** (1943), 115–133, DOI 10.1007/bf02478259. MR10388

[MP17] Marvin Minsky and Seymour A Papert, *Perceptrons, reissue of the 1988 expanded edition with a new foreword by léon bottou: An introduction to computational geometry*, MIT press, 2017.

[Mum20] David Mumford, *The Astonishing Convergence of AI and the Human Brain*, October 1, 2020, Accessed on 13 March, 2023.

[Ola14] Christopher Olah, *Neural Networks, Manifolds, and Topology*, `https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/`, April 6, 2014, Accessed on 18 March, 2023.

[PSHL20] Dylan James Peifer, *Reinforcement Learning in Buchberger's Algorithm*, ProQuest LLC, Ann Arbor, MI, 2021. Thesis (Ph.D.)–Cornell University. MR4297517

[RHW85] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, *Learning internal representations by error propagation*, Tech. report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[SC] Daniel Smilkov and Shan Carter, *The Tensorflow Playground*, `https://playground.tensorflow.org`, Accessed on 18 March, 2023.

[Sim21] Carlos Simpson, *Learning proofs for the classification of nilpotent semigroups*, Preprint, `arXiv:2106.03015`, (2021).

[Tur48] A. M. Turing, *Intelligent machinery, a heretical theory*, Philos. Math. (3) **4** (1996), no. 3, 256–260, DOI 10.1093/philmat/4.3.256. MR1406760

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin, *Attention is all you need*, Advances in neural information processing systems **30** (2017).

[Wag21] Adam Zsolt Wagner, *Constructions in combinatorics via neural networks*, Preprint, `arXiv:2104.14516`, (2021).

[Wik] Wikipedia, *Universal approximation theorem*, `https://en.wikipedia.org/wiki/Universal_approximation_theorem`, Accessed on 17 May, 2023.

Sydney Mathematical Research Institute, University of Sydney, A14 - Quadrangle, NSW 2006, Sydney, Australia

Email address: `g.williamson@sydney.edu.au`