1 Generalized Computation

- Hyperarithmetic Computation
- Infinite Time Turing Machines
- Physical Hypercomputation
Looking Back

So far we have have taken a little tour of the computational universe:

Classical Computability
Complexity Theory
Automata Theory
Connections to Logic
Connections to Algebra

Could this be all?
Remember Banach:

Good mathematicians see analogies between theorems or theories; the very best ones see analogies between analogies.

Our theory of computation focuses on the structure

\[ \mathbb{N} = \langle \mathbb{N}, +, \cdot, 0, 1 \rangle \]

But there are many other structures where one would like to compute: the reals, the complexes, some rings, fields, set-theoretic universes and so on. Is there any way to make sense out of computing in these structures?
There are some cases where it is entirely straightforward to lift our definitions. Typically, this can be handled in two ways:

- Code the elements $d$ of some discrete domain as $\gamma(d) \in \mathbb{N}$ where $\gamma$ is intuitively computable.
- Transfer the definitions directly to the new domain.

Domains $D$ that admit a suitable coding function $\gamma$ are sometimes called effectively enumerable. Typical examples are $D = \mathbb{Z}$, $\Sigma^*$, $\mathbb{HF}$.

Lifting definitions also works nicely; e.g., we can define primitive recursive functions directly on words.
Everyone would agree that computation is a key concern in analysis. Here one naturally operates in the structure

\[ \mathcal{R} = \langle \mathbb{R}, +, \cdot, 0, 1 \rangle \]

or various extensions thereof (e.g., we could add \( \log \) and \( \sin \), continuous functions, differentiable functions, and so on).

A large part of physics and engineering computes in this structure, sometimes in a slightly haphazard way (e.g. by using rational approximations without worrying too much about accuracy).

Several attempts have been made to define a rigorous theory of computation over \( \mathcal{R} \), but no single one has emerged as the clear "correct" description (computable analysis, domain theory, real RAMs, \ldots ).
This is in stark contrast to computability on the naturals: all the standard models there agree. And, for Turing machines one can make a very tightly reasoned and sound argument for the following assertion:

Turing machines capture exactly the intuitive notion of computability in discrete mathematics.

Of course, this is not a theorem, just a highly plausible assertion. Even without a complete axiomatization of physics, it seems to be well aligned with physically realizable computability.

Computation in continuous domains such as $\mathbb{R}$ is much more problematic.
A good number of “theories of computation” on structures other than the natural numbers have been developed: computation on ordinals, computation on sets, computation on algebraic structures, computation on higher types and so on.

There is even an axiomatization of computation:

J. E. Fenstad
General Recursion Theory: An Axiomatic Approach
Springer 1979

Unfortunately, the axiomatization by Fenstad feels a bit awkward and overly technical (compared to, say, Hilbert’s axiomatization of geometry or Zermelo-Fraenkel’s axiomatization of set theory), but overall it captures the fundamental ideas behind computation fairly well.
Generalized Computation

Hyperarithmetic Computation

Infinite Time Turing Machines

Physical Hypercomputation
Banach’s dictum was realized already in the 1930s when Church and Kleene started an investigation of recursive ordinals, an effective version of classical set-theoretic ordinals.

As it turns out, there is a perfectly natural way to define computation in this context, though the resulting theory is far, far removed from anything resembling realizable computation. The reward is a powerful analysis of (some important part of) the mathematical universe.
Let $A \subseteq \mathbb{N}$. Recall that the (Turing) jump of $A$ is defined as

$$A' = \{ e | \{e\}^A(e) \downarrow \}$$

So $\emptyset'$ is just the ordinary Halting set.

The $n$th jump $\emptyset^{(n)}$ is obtained by iterating the jump $n$ times. For $n$ greater than 3 or 4 it is pretty difficult to wrap one's head around these sets.

But why stop there?
\[ \emptyset^{(\omega)} = \{ \langle n, e \rangle \mid e \in \emptyset^{(n)} \} \]

So \( \emptyset^{(\omega)} \) contains all the information of the \( \emptyset^{(n)} \) combined. This is already enough to decide the truth of arbitrary statements of arithmetic (first-order only, all quantifiers range over \( \mathbb{N} \)).

But why stop there?

We could still do \( (\emptyset^{(\omega)})' = \emptyset^{(\omega+1)} \) and \( (\emptyset^{(\omega+1)})' = \emptyset^{(\omega+2)} \), and so on.

Where does this end?
You might complain that $\omega + 2$ is really a meaningless symbol. True, but you can easily make sense out of it.

The standard well-order on $\mathbb{N}$ defines $\omega$:

$$0 < 1 < 2 < \ldots < n < \ldots \mid$$

But we can rearrange things a bit to get $\omega + 1$.

$$1 < 2 < \ldots < n < \ldots \mid < 0$$

Or $\omega + 2$:

$$2 < \ldots < n < \ldots \mid < 0 < 1$$

Given an ordinal $\alpha$ we can always form the successor $\alpha + 1$. For our purposes, all of these can be construed as particular well-orderings on $\mathbb{N}$. 
Now suppose we have a strictly increasing sequence

$$\lambda_0 < \lambda_1 < \ldots < \lambda_n < \ldots$$

Then we can form the limit ordinal \( \lambda = \lim_n \lambda_n \).

In set theory land, ordinals can be expressed as particular sets, typically von Neumann ordinals. Limits are not an issue here; in fact, \( \lambda = \bigcup \lambda_n \).

But we would like to keep things computable, so we will insist that the map \( n \mapsto \lambda_n \) is a total, computable function.
\( \lambda_n \) is an ordinal, how on earth could it be the output of a computable function?

It can’t be directly, but we can code all these ordinals as natural numbers:

**Zero:** 0 denotes 0 (sorry).

**Successor:** if \( n \) denotes \( \alpha \), then \( \langle 1, n \rangle \) denotes \( \alpha + 1 \).

**Limits:** If \( \{e\} \) is a total computable function such that \( \{e\}(n) \) denotes a strictly increasing sequence of ordinals, then \( \langle 2, e \rangle \) denotes the limit of this sequence.

The collection of all these code numbers is called **Kleene’s \( \mathcal{O} \)**.

The least ordinal that is not represented in \( \mathcal{O} \) is called \( \omega_1 \) Church-Kleene, in symbols \( \omega_1^{\text{CK}} \).
For each ordinal $\alpha < \omega_1^{\text{CK}}$ we can define the corresponding iterated jump $\emptyset^{(\alpha)}$.

It takes a bit of effort to show that this is well-defined (there are many choices for the function representing a limit ordinal).

**Definition**

A set $X \subseteq \mathbb{N}$ is hyperarithmetic if it is Turing reducible to some $\emptyset^{(\alpha)}$, $\alpha < \omega_1^{\text{CK}}$.

Note that the truth of arithmetic already appears at level $\omega$, so hyperarithmetic sets are hugely complicated in general.
Because these sets naturally pop up in analysis, aka second-order arithmetic.

By contrast, the classical approach to computation corresponds closely to plain first-order arithmetic, the first-order theory of

$$\mathcal{N} = \langle \mathbb{N}, +, \cdot, 0, 1 \rangle$$

In fact, one can show that a set $A \subseteq \mathbb{N}$ is semidecidable iff there is a $\Sigma_1^0$ formula that defines it:

$$x \in A \iff \mathcal{N} \models \exists z \varphi(z, x)$$

More than an existential quantifier kicks us out of the realm of computability.
To do analysis (as opposed to just arithmetic), one needs to be able to quantify over sets of naturals, not just individuals. The first really interesting class of sets that can be defined that way is $\Pi^1_1$:

$$x \in A \iff \mathbb{N} \models \forall X \varphi(x, X)$$

where $\varphi$ contains only first-order quantifiers and $X \subseteq \mathbb{N}$ is a second-order variable. Similarly one defines $\Sigma^1_1$ and $\Delta^1_1$ as the intersection of $\Sigma^1_1$ and $\Pi^1_1$.

Kleene’s $\mathcal{O}$ is a classical $\Pi^1_1$ set (and even complete wrto many-one reductions), decidable well-orderings are another example.
Theorem (Kleene 1955)

*The hyperarithmetic sets are exactly the $\Delta^1_1$ sets.*

This is really quite surprising: hyperarithmetic sets are rather constructive, $\Delta^1_1$ appears to live far away in set theory land and have no obvious connection to anything having to do with computation.

Arguably, this is the first major theorem in generalized computability.
Surprisingly, it turns out that $\Pi^1_1$ sets behave very much like semidecidable sets ($\Sigma^1_1$ sets).

Even more more surprisingly, it turns out the $\Delta^1_1$ sets behave very much like finite sets (not a typo, I mean finite rather than decidable).

Loosely speaking, replace $\omega$ everywhere by $\omega^\text{CK}_1$, and, voila, a new recursion theory.
Let’s just look at one tiny example.

**Theorem (Owing’s Splitting)**

*Given two semidecidable sets $A$ and $B$, there are semidecidable sets $A_0 \subseteq A$ and $B_0 \subseteq B$ such that $A_0 \cup B_0 = A \cup B$ but $A_0$ and $B_0$ are disjoint.*

The exact same theorem holds for $\Pi^1_1$ sets.
In fact, one can axiomatize computability as a system of weak set theory, so-called Kripke-Platek (KP) set theory.

KP looks rather similar to ordinary set theory, but separation and replacement are much restricted.

It turns out that classical computability corresponds to the smallest model of KP (at level $\omega$), and hyperarithmetic computability corresponds to the second smallest model (at level $\omega_1^{CK}$).

Needless to say, this is the tip of an iceberg: there are infinitely many more models of increasing complexity (so-called admissible ordinals).
- Generalized Computation
- Hyperarithmetic Computation
- Infinite Time Turing Machines
- Physical Hypercomputation
Here is a more recent example of a theory of computation that strictly generalizes the classical theory.

The main idea is simple: find a way to explain what it means for a Turing machine to run an “infinite amount” of time (actually, measured in terms of ordinals). Of course, this is not a model that relates to anything realizable in physics, but some interesting results can be obtained this way.

We would expect to get strictly more compute power than from ordinary TMs; e.g., we would like to be able to solve the Halting Problem.
An ordinary Turing machine computes its result in some finite number $n < \omega$ of steps, if it halts at all.

A Turing machine enumerating a (infinite) semidecidable set takes $\omega$ steps to finish this task (writing down all the elements of the set on some special output tape).

Could one somehow run a Turing machine for, say, $\omega^2$ steps? Can we define an infinite time Turing machine (ITTM)?

It is easy to get from $\alpha$ steps to $\alpha + 1$ steps: the machine just performs one more step according to its ordinary transition table. The real problem is: what to do at limit stages?
So the key questions is: how can we make sense of a computation at stage $\omega$? The other limit stages will be the same.

Here is a useful model developed by Joel Hamkins and his co-workers.

We use a Turing machine with a one-way infinite tape that is subdivided into three tracks:

<table>
<thead>
<tr>
<th>input</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>...</th>
<th>$x_n$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>scratch</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>...</td>
<td>$y_n$</td>
<td>...</td>
</tr>
<tr>
<td>output</td>
<td>$z_0$</td>
<td>$z_1$</td>
<td>...</td>
<td>$z_n$</td>
<td>...</td>
</tr>
</tbody>
</table>

We can safely assume that the alphabet is $2$, so each track contains a word in $2^\omega$. We have a finite state control and a read/write head, as usual.
Now suppose the Turing machine has already performed all steps $\alpha < \omega$. We define the configuration at time $\omega$ as follows:

- The machine is in state $q_{\text{limit}}$.
- The head is in position 0.
- The content of a tape subcell is the limsup of its contents at times $\alpha < \omega$.

The definition for an arbitrary limit ordinal $\lambda$ is exactly the same.

Thus, the entry will be 1 at stage $\lambda$ iff for all $\beta < \lambda$ there is a $\beta < \alpha < \lambda$ such that the entry at time $\alpha$ is 1.
Note that the limit state $q_{\text{limit}}$ is the same for each limit stage $\lambda$; there is no special $q_{\omega}$, $q_{\omega \cdot 2}$, $q_{\omega \cdot 3}$ and so on.

So the only way to preserve information across limit stages is via tape cells; state and head position are always the same.

For example, we can have the first cell on the work tape hold a 1 at a limit stage iff something happened over and over arbitrarily close to the limit stage.

An ordinary Turing machine simply cannot do this.
Consider the Halting problem for ordinary Turing machines: given an index $e$ we want to know if $\{e\}$ halts on the empty tape.

This can be handled by an ITTM $\mathcal{M}$: $e$ is written on the input track. $\mathcal{M}$ then simulates the ordinary computation of $\{e\}$ on empty tape using the scratch track.

If $\{e\}$ halts after $s < \omega$ steps, $\mathcal{M}$ also halts and accepts.

Otherwise we reach the limit stage $\omega$. Then $\mathcal{M}$ halts and rejects at time $\omega + 1$.

So ordinary Halting is easily decidable by an ITTM (we don’t need the limsup mechanism, getting to a limit stage is enough).
Halting is at the bottom level of the arithmetical hierarchy, but we can go further.

Recall $\text{Inf} = \{ e \mid W_e \text{ infinite} \}$, the collection of all Turing machines that halt on infinitely many inputs. So $\text{Inf}$ is $\Pi^0_2$-complete, a bit more complicated than Halting.

Claim

*We can decide membership in $\text{Inf}$ by an ITTM $M$."

As before, $e$ is written on the input track. Then $M$ runs the following program:

```plaintext
foreach $x \in \mathbb{N}$ do
  if $\{e\}(x) \downarrow$ then
    $y_0 = 1$;
    $y_0 = 0$;
```
If $\mathcal{M}$ finds that $\{e\}$ converges on $n$, it turns bit $y_0$ on, and then off again at the next step. Of course, $\mathcal{M}$ will not use subcell $y_0$ for the simulation.

The check for each $n$ may require up to $\omega$ steps, so the total running times is between $\omega$ and $\omega^2$.

A similar trick allows one to check when the main loop is done.

Upon completion of the main loop, subcell $y_0$ will hold a 1 iff there were infinitely many good $n$’s.
In fact, ITTMs can handle any $\emptyset^{(n)}$, $n < \omega$.

To see why, suppose we already have an ITTM $M$ that decides $\emptyset^{(n)}$.

To get $\emptyset^{(n+1)}$, we construct a new ITTM $M'$ that runs $\{e\}_{\sigma}^{\emptyset^{(n)}}$ for all finite stages $\sigma$. The oracle is replaced by calls to $M$.

As before, if there is convergence, we can stop, otherwise we go to the next limit stage and know that the computation diverges.

At the last limit stage, where all these computations have finished, we can have a 0 or 1 in the first output cell depending on whether we discovered a bad $n$.

The whole computation can be handled in $\omega^2$ steps.
This actually shows that an ITTM can decide the truth of an arbitrary arithmetic sentence.

The key is that any $\Sigma^0_k$ sentence only requires $\emptyset^{(k)}$ as an oracle to decide, and we have just seen that the latter iterated jump is ITTM decidable.

This means that ITTM decidability is not even located in the arithmetical hierarchy, we get strictly more power this way.
An ordinary Turing machine can halt, enter a loop or diverge (run through an ω-sequence of non-repeating configurations).

By contrast, an ITTM either halts or enters a loop: even if it “diverges” for a while, it will ultimately end up in a limit cycle.

How far do we have to go before the computation becomes stuck in a loop?

**Theorem**

*Every computation of a ITTM either halts or enters a loop after countably many steps.*

So we do not have to run the machine $\aleph_1$ or $\aleph_{17}$ many steps, some $\alpha < \aleph_1$ will do.
We can code a well-order of the natural numbers as a binary $\omega$-word $W$:

$$x < y \iff W(\langle x, y \rangle) = 1$$

**Theorem**

*It is ITTM decidable whether $W \in 2^\omega$ codes a well-order.*

This may seem like a neither-here-nor-there result but it is actually very important.

For those familiar with the analytical hierarchy: checking a well-order is $\Pi^1_1$-complete.
One might wonder why the tape of an ITTM is split into three tracks. A single plain tape, say, with a binary alphabet would seem more natural.

If one does not look too closely, one could easily convince oneself that this one-track version can simulate the three-track version and is thus equivalent.

Wrong! This simulation breaks and one-track ITTM are strictly weaker (and far less interesting) than the three-track kind.
There are two fairly natural ways to make one-track machines have the same compute power as the three-track kind.

- **Extra cell**
  Add a special scratch cell separate from the standard tape. The extra cell is **not** used for input/output.

- **Larger alphabet**
  Enlarge the alphabet to contain a special blank symbol, at limit stages every cell is updated to its limit if it exists, blank otherwise.

Note that this is really not a good sign: ordinary Turing machines are incredibly robust, any reasonable change produces the same class of computable functions. Not so for ITTMs.
■ Generalized Computation

■ Hyperarithmetic Computation

■ Infinite Time Turing Machines

4 Physical Hypercomputation
There are some areas of intellectual discourse where the experts can engage in furious debates about the basic usefulness of certain theories, even beyond the question of whether the theories are correct. Psychology, philosophy, literature, history, economics and so forth come to mind.

However, in the hard sciences, these debates are quite rare and tend to focus on particular technical issues (is string theory falsifiable, is it still physics?).

Amazingly, we now have an example of an entirely meaningless “theory” in mathematics.
The term gained notoriety after a 1999 paper by Jack Copeland and Diane Proudfoot in the Scientific American titled "Alan Turing’s Forgotten Ideas in Computer Science".

The article makes the most astounding claim that Turing “anticipated hypercomputation” in his technical work.

A clever PR trick, of course. Hiding behind one of the greats is usually a good idea.
To support this conclusion, the authors misinterpret Turing’s concept of an oracle machine in patently absurd ways. There is a nice picture of an oracle machine in the paper:

The idea is that the big, ominous, gray box (the oracle) has access to a copy of the Halting set, living in the blue box, here called $\tau \in 2^\omega$. 
The authors comment:

Obviously, without $\tau$ the oracle would be useless, and finding some physical variable in nature that takes this exact value might very well be impossible. So the search is on for some practicable way of implementing an oracle. If such a means were found, the impact on the field of computer science could be enormous.

Exact value? How about Heisenberg?

No doubt, the impact would be enormous. The problem is that there seems to be no way to get an infinite amount of information into a finite physical system.
Hodges has written the definitive biography of Turing (incidentally, very well worth reading). He found it necessary to comment on the Copeland/Proudfoot article, an unusual step in the genteel world of math.

http://www.turing.org.uk/publications/sciam.html

There is also pdf of an article at this site that is quite interesting.
Is the universe a gigantic computer, say, a cellular automaton? Some other mathematical structure?

Many objections could be raised to this proposal. The most relevant for us is that abstract mathematical entities are not the right kind of entity to implement a computation. Time and change are essential to implementing a computation: computation is a process that unfolds through time, during which the hardware undergoes a series of changes (flip-flops flip, neurons fire and go quiet, plastic counters appear and disappear on a Go board, and so on). Abstract mathematical objects exist timelessly and unchangingly. What plays the role of time and change for this hardware? How could these Platonic objects change over time to implement distinct computational steps? And how could one step “give rise” to the next if there is no time or change? Even granted abstract mathematical objects exist, they do not seem the right sort of things to implement a computation.
A search for “hypercomputation” generates 115,000 hits on Google, but “Kardashians” produces a healthy 196,000,000 hits. Insufficient evidence for a real thing.

**Wikipedia**
Hypercomputation or super-Turing computation refers to models of computation that can provide outputs that are not Turing computable. For example, a machine that could solve the halting problem would be a hypercomputer; so too would one that can correctly evaluate every statement in Peano arithmetic.

OK, but how is this any different from, say, or ITTM?
The new machines go beyond Turing’s attempts to formalize the rote calculations of a human clerk and instead involve operations which may not even be physically possible. This difference in flavor is reflected in the terminology: they are hypermachines and perform hypercomputation. Can they be really said to “compute” in a way that accords with our pre-theoretic conception? It is not clear, but that is no problem: they hypercompute. Hypercomputation is thus a species of a more general notion of computation which differs from classical Turing computation in a manner that is difficult to specify precisely, yet often easy to see in practice.

This is taken from a 2006 article titled “The many forms of hypercomputation.”

Many, indeed. The subject is known as generalized recursion theory.

And, yes, specifying anything precisely in this context is very hard indeed.
Let us suppose, however, that hypercomputation does turn out to be physically impossible—what then? Would this make the study of hypercomputation irrelevant? No. Just as non-Euclidean geometry would have mathematical relevance even if physical space was Euclidean, so too for hypercomputation. Perhaps, we will find certain theorems regarding the special case of classical computation easier to prove as corollaries to more general results in hypercomputation. Perhaps our comprehension of the more general computation will show us patterns that will guide us in conjectures about the classical case.

Absolutely. This is exactly what generalized recursion theory was created for.

Note the evolution on the issue of implementability, apparently the oracles are still at large.
Thus the claims that such problems are “undecidable” or “unsolvable” are misleading. As far as we know, in 100 years time these problems might be routinely solved using hypermachines. Mathematicians may type arbitrary Diophantine equations into their computers and have them solved. Programmers may have the termination properties of their programs checked by some special software. We cannot rule out such possibilities with mathematical reasoning alone. Indeed, even the truth or otherwise of the Church-Turing Thesis has no bearing on these possibilities. The solvability of such problems is a matter for physics and not mathematics.

So now implementability is critical again, we actually want to build and run our hypermachines.

And undecidability is a matter of physics, not math. Yessir.
Das ist nicht einmal falsch!

Translation: This is not even wrong.

A comment made by Pauli about a chaotic and incoherent paper by a young aspiring physicist.

Pauli is also on record saying: “Gödel, Blödel.”

This roughly translates to “Gödel, dimwit” and represents Pauli’s response to Gödel’s construction of a circular space-time solution to the Einstein equations.
It seems clear that hypercomputationist really should cling to physics for dear life: without implementability they are adrift in generalized recursion theory, a field that they apparently do not understand.

To get any traction, one has to ask whether the physics of our actual universe somehow support hypercomputation. Of course, this is an exceedingly difficult question: Hilbert’s problem #6 is still unanswered: no one knows how to axiomatize physics in its entirety.

Mathematical Treatment of the Axioms of Physics.

The investigations on the foundations of geometry suggest the problem: To treat in the same manner, by means of axioms, those physical sciences in which already today mathematics plays an important part; in the first rank are the theory of probabilities and mechanics.
Ideally we could build an axiomatization $\Gamma$ of physics, in some sufficiently powerful logic. So the real world would be a structure that models $\Gamma$ (and perhaps is not uniquely determined). $\Gamma$ would be a Theory of Everything.

Then, with a bit of effort, one might be able to show that

$$\Gamma \vdash \exists M \text{ (device } M \text{ solves Halting)}$$

Or, we might be able to prove the negation. $\Gamma$ could be loop quantum gravity, or string theory, or some such.

But in the absence of $\Gamma$, we’re just producing (bad) poesy.
OK, so this is not quite true.

It is an excellent exercise to fix some particular theory $\Gamma$ of physics (not a ToE) and try to show that in $\Gamma$ it is possible to “construct a device” that solves the Halting problem.

For example, assume Newtonian physics: gravitating point masses, no relativity theory, no quantum theory.

It’s a nice exercise, but it has no bearing on implementability.
Here is a way to solve the Halting problem: find a way to speed up the computation of a (Turing) machine. The first step takes one second, the second 1/2 a second, the $n$th takes $2^{-n+1}$ seconds.

A whole infinite computation takes a mere 2 seconds.

Of course, there is a minor problem: we don’t know how to achieve this constant doubling in computation speed, Moore’s law notwithstanding. For any number of physical reasons, this type of machine is not realizable.
Example: The Brain
Neural nets are a fascinating idea: since no one has any clue how to formalize intelligence, why not try and synthesize the underlying wetware?

Alas, human brains are exceedingly complex:

- some 90 billion neurons
- dendrites connect to some 10000 neurons
- synaptic connections are analogue and complicated

Alone determining the exact topology of this network is very, very difficult. No one has a precise model of the actual dynamic interconnections.

Still . . .
With a sufficient dose of abstraction we wind up with a neural nets.

We are given a collection \(x_1, x_2, \ldots, x_n\) of “neurons” plus external inputs \(u_1, u_2, \ldots, u_m\), all reals.

The state of a neuron at time \(t + 1\) is determined by the state of all neurons and the inputs at time \(t\) as follows:

\[
x_i(t + 1) = \sigma \left( \sum_j a_{ij} x_j(t) + \sum_j b_{ij} u_j(t) + c_i \right)
\]

The weights \(a_{ij}\) and \(b_{ij}\) (and the offsets \(c_i\)) determine the functionality of the net.
The function $\sigma$ is typically a sigmoid function like these.

Note that this is very similar to Boolean threshold functions.
The simplest yet useful arrangement is to have the “neurons” arranged in three layers: input, hidden layer, output.

Of course, we could have multiple hidden layers. We could even have feedback.
One can provide a (long) list of input/output pairs, and have the net adjust its internal weights so as to produce the appropriate behavior, or at least something similar (supervised learning).

The hope is that the network will then produce proper output even for inputs it has not previously encountered.

Surprisingly, this works quite well for a range of pattern recognition, including text and speech. A great application of linear algebra and statistics.
Siegelmann and Sontag have shown that a neural net with

- binary inputs and outputs
- rational weights
- linear sigmoid functions

is computationally equivalent with Turing machines. So we have yet another model of computation, this time one inspired by neural processes.

This requires some effort, since there is no immediate analog of a tape: one needs to store tape inscriptions in terms of rational weights and manipulate them accordingly.
Then Siegelmann steps into the abyss: she claims that a net can achieve hypercomputation if one allows real valued weights: the weight can be a non-computable real number and can thus encode all kinds of information such as the Halting problem or Chaitin’s $\Omega$.

Perfectly true, but no more than a little exercise: this cannot pass the implementability test. In fact, even if we could somehow construct a device with a weight, say, $\Omega$, quantum physics prevents us from reading off the bits. And, of course, there is zero evidence that the device could be built in the first place.
Neural nets with real valued weights can “solve” the Halting problem as can ITTMs. Why should one care about one but not the other?

Because ITTMs produce an interesting theory of computation that has close connections to other areas of generalized recursion theory along the lines discussed above. The proof techniques are interesting and quite complicated.

Real-weighted neural nets, on the other hand, are quite useless, just a fancy-schmancy gadget that is of no importance anywhere else.
In 2006, Martin Davis could not stand it any longer, and published a paper

The Myth of Hypercomputation

In the paper, he very sweetly demolishes, annihilates and eviscerates the idea of “hypercomputation.”

Say goodnight, Gracie.