Simulating RNNs on GPUs

Working Title

Zhangsheng Lai

September 10, 2017

Mult-layer Perceptron

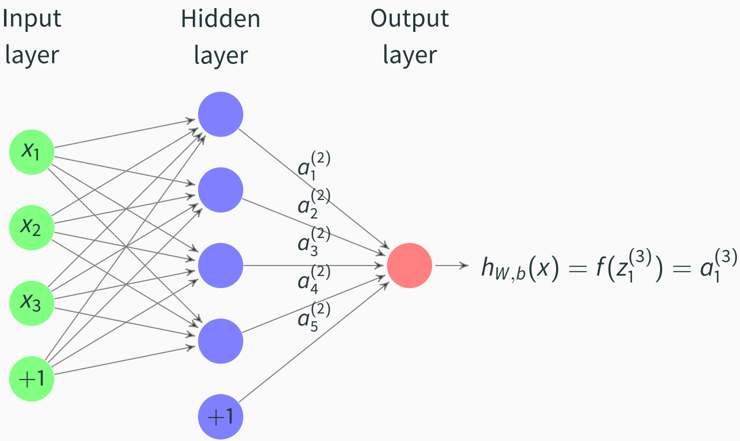
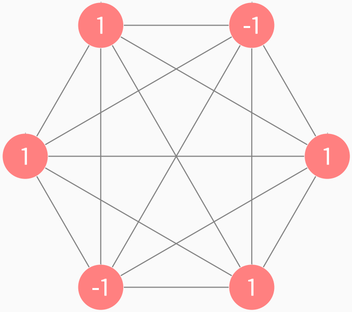


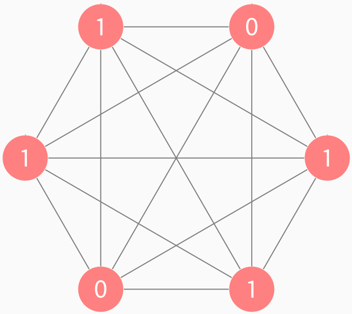
Figure 1: Multi-layer perceptron

Hopfield Networks and Boltzmann Machines

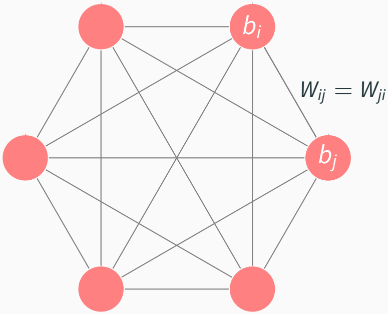
Hopfield Networks



Hopfield Networks



Hopfield Networks

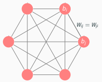


Energy configuration, $E = -\sum_{i<j} W_{ij}x_ix_j - \sum_i b_ix_i$

Energy gap, $\Delta E_i = E(x_i = 0) - E(x_i = 1) = \sum_j W_{ij}x_j + b_i$

Update rule, $x_i := \begin{cases} 1 & \sum_j W_{ij}x_j + b_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$

Hopfield Networks



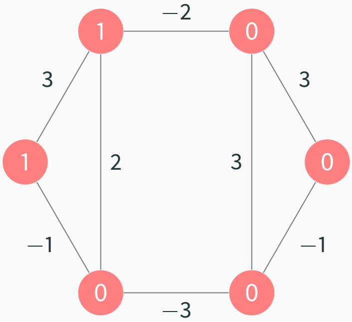
Energy configuration, $E = -\sum_{i<j} W_{ij} x_i x_j - \sum_i b_i x_i$

Energy gap, $\Delta E_i = E(x_i = 0) - E(x_i = 1) = \sum_j W_{ij} x_j + b_i$

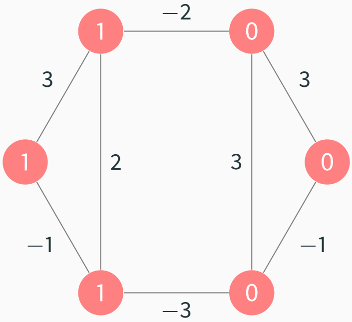
Update rule, $x_i := \begin{cases} 1 & \sum_j W_{ij} x_j + b_i \geq 0 \\ -1 & \text{otherwise} \end{cases}$

- composed of primitive computing elements called units
- units has two states, on or off, represented by $\{1, -1\}$ or $\{1, 0\}$
- connected to each other by bi-directional links
- adopts these states as a function of the states of its neighbouring units and weights of its links to them, it is a probabilistic function for a Boltzmann machine.
- weights can take on any real value
- a unit being on or off is taken to mean that the system currently accepts or rejects some elemental hypothesis of the domain
- weight on a link represents a weak pairwise constrain between two hypothesis
- positive (negative) weights indicate that two hypothesis support (contradict) one another with other things being equal
- link weights are symmetric, having the same strength in both directions

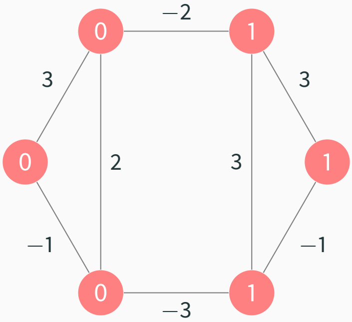
Hopfield Networks



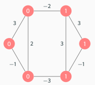
Hopfield Networks



Hopfield Networks

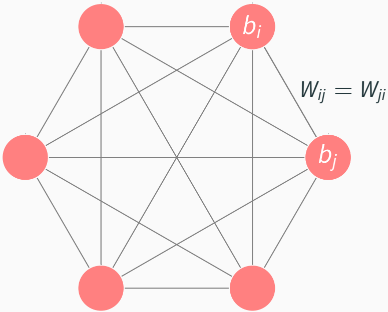


HOPFIELD NETWORKS



- Updating of Hopfield networks:
  Asynchronous: one unit is updated at a time, can be done randomly or in a pre-defined order.
  Synchronous: all units are updated at the same time.
- Hopfield networks always make decisions to reduce the energy and makes it impossible to escape from local minima.
- random noise can help us escape from poor minima, by starting with lots of noise so its easy to cross energy barriers and gradually decrease the noise so the system ends in a deep minimum. This is called simulated annealing.

Boltzmann Machines

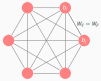


$$E = -\sum_{i<j} W_{ij} x_i x_j - \sum_i b_i s_i$$

$$\Delta E_i = E(x_i = 0) - E(x_i = 1) = \sum_j W_{ij} x_j + b_i$$

$$\mathbb{P}(x_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}}$$

Boltzmann Machines



$$E = -\sum_{i<j} W_{ij} x_i x_j - \sum_i b_i s_i$$

$$\Delta E_i = E(x_i = 0) - E(x_i = 1) = \sum_j W_{ij} x_j + b_i$$

$$\mathbb{P}(x_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}}$$

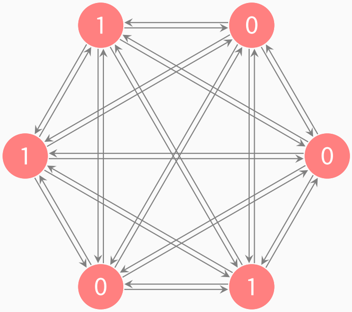
- replace the binary threshold units by binary stochastic units that make biased random decisions
- temperature variable controls the amount of noise

Boltzmann Machines

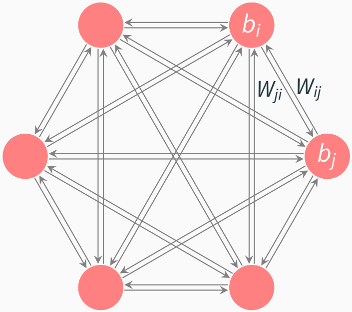
$$\mathbb{P}(x_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}}$$

McCulloh-Pitts Machines

McCulloh-Pitts Machines



McCulloh-Pitts Machines



$$E(y, x|\theta) = -\sum_{ji \in E} W_{ji} y_j x_i - \sum_{j \in V} b_j s_j - \sum_{i \in V} b_i s_i$$

$$\Gamma_{yx} = \exp\left(-\frac{1}{2\tau} E(y, x|\theta) + \frac{1}{2\tau} E(x, x|\theta)\right)$$

where $G = (V, E)$ digraph, $\theta = (W, b, \tau)$

McCulloh-Pitts Machines

$$E(y, x|\theta) = -\sum_{ji \in E} W_{ji} y_j x_i - \sum_{j \in V} b_j s_j - \sum_{i \in V} b_i s_i$$

$$\Gamma_{yx} = \exp\left(-\frac{1}{2\tau} E(y, x|\theta) + \frac{1}{2\tau} E(x, x|\theta)\right) := \lambda_j = \exp\left(\frac{1}{2\tau} s_j z_j\right)$$

$s_j = 1 - 2x_j$ and $z_j = \sum_{ji \in E} W_{ji} x_i + b_j$

McCulloh-Pitts Machines

$$E(x^{(n+1)}, x^{(n)}|\theta) = -\sum_{ji \in E} W_{ji} x_j^{(n+1)} x_i^{(n)} - \sum_{j \in V} b_j s_j - \sum_{i \in V} b_i s_i$$

$$\Gamma_{x^{(n+1)} x^{(n)}} = \exp\left(-\frac{1}{2\tau} E(x^{(n+1)}, x^{(n)}|\theta) + \frac{1}{2\tau} E(x^{(n)}, x^{(n)}|\theta)\right)$$

McCulloh-Pitts Machines

Introduction

Homotopy type theory is a new branch of mathematics that combines *homotopy theory* and *type theory*.

- **Homotopy theory:** an outgrowth of algebraic topology and homological algebra, with relationship with higher category.
- **Type theory:** branch of mathematical logic and theoretical computer science.

Russell's Paradox

Let R be the set of sets that are not members of themselves. If R is not a member of itself, then it must contain itself, and if it contains itself, it then contradicts its own definition as the set of all sets that are not members of themselves. This contradiction is Russell's paradox.

$$R = \{x \mid x \notin x\}, \text{ then, } R \in R \Leftrightarrow R \notin R$$

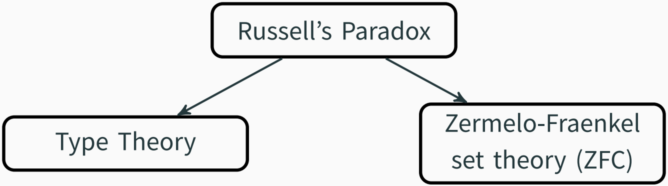


Figure 2: Ways to avoid Russell's Paradox

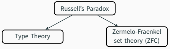


Figure 2: Ways to avoid Russell's Paradox

- the set of all airplanes are not members of themselves, but the set of all non-airplanes, are members of themselves.
- The essential difference between Russell's and Zermelo's solution to the paradox is that Zermelo altered the axioms of set theory while preserving the logical language in which they are expressed (the language of ZFC, with the help of Skolem, turned out to be first-order logic) while Russell altered the logical language itself.

Analysing the paradox, the problem stems from the set theory's liberal formation principles of allowing inhomogeneous sets. More specifically, a collection of objects in a set may contain members which can only be defined by means of the collection as a whole.

Example

Consider the collection of propositions, which supposed to contain a proposition stating that "all propositions are either true or false". The analysis we made earlier suggests that this proposition would not be legitimate unless "all propositions" referred to some already definite collection.

Over the years, type theory was further developed by many other people. The type theory that we consider in HoTT is *Martin Löf type theory*, which has applications in computer science and the theory of programming languages.

The clear reasoning principles associated with the construction of types also form the basis of modern *computer proof assistants*, with Coq and Agda being some of the more popular ones.

Over the years, type theory was further developed by many other people. The type theory that we consider in HoTT is *Martin Löf type theory*, which has applications in computer science and the theory of programming languages.

The clear reasoning principles associated with the construction of types also form the basis of modern *computer proof assistants*, with Coq and Agda being some of the more popular ones.

- In short, we could see it as some sort of self referencing; an element of the set is referencing to the set it is contained in and this is not good!
- The *type* in *type theory* is similar to the data-types we have in programming languages.

A *homotopy* between a pair of continuous mappings $f : X \to Y$ and $g : X \to Y$ is a continuous map

$$H : X \times [0,1] \to Y$$

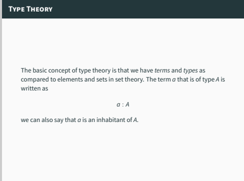
satisfying $H(x,0) = f(x)$ and $H(x,1) = g(x)$, i.e. some form of "continuous deformation" of f into g.

Type Theory

The basic concept of type theory is that we have *terms* and *types* as compared to elements and sets in set theory. The term a that is of type A is written as

$$a : A$$

we can also say that a is an inhabitant of A.



There is an inclination to just pass it off as just a change in notation from $\in$ to :, but there is more than this to it.

Type Theory

There are two ways of interpreting a type:

- when A is used to represent a proposition, then the a in $a : A$ may be seen as a witness to the provability of A or evidence to the truth of A. This important concept of *proposition as types* is how it is used to formalize mathematics and verify the correctness of proofs.
- the type A can also be treated more like a set than a proposition, then "$a : A$" is analogous to "$a \in A$" but they differ in the the former is a *judgment* whereas the latter is *proposition*.

Example

Here we have a proposition “A is isomorphic to B” stated in type theory notation:

$$\mathrm{Iso}(A, B) :\equiv \sum_{f:A\to B} \sum_{g:B\to A} \left(\left(\prod_{x:A} g(f(x)) = x \right) \times \left(\prod_{y:B} f(g(y)) = y \right) \right)$$

and to be able to find a term $p : \mathrm{Iso}(A, B)$ is same as constructing an isomorphism between A and B. Proofs are treated as mathematics objects like functions, groups and permutations.

Example

There exists two irrational numbers such that its sum is rational.

$$\exists\, x, y \in \mathbb{R} \setminus \mathbb{Q} \text{ such that } x + y \in \mathbb{Q}$$

Comparison with Set Theory

Set Theory	Type Theory
"membership" is a relation that may or may not hold between an element a and a set A	every term belongs to some type; A type needs to be formed first before introduction rules are used to construct the terms of the type.
$\mathbb{N}$ in set theory: $\{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \ldots\}$ $\{0, 1, 2, \ldots\}$	$\mathbb{N}$ in type theory: $0 : \mathbb{N}$ and succ: $\mathbb{N} \to \mathbb{N}$ $1 :\equiv \mathrm{succ}(0), 2 :\equiv \mathrm{succ}(1), \ldots$

Equalities

Judgmental equality or *Definitional equality* is equality of the syntax and is denote by $:\equiv$.

For example,

$$(x+1)^2 :\equiv x^2 + 2x + 1$$

since it is easily obtained by algebraic expansion.

Propositional equality on the other hand is though of as a type[1]. For terms $a, b : A$, we have the type "$a =_A b$". Only when $a =_A b$ is inhabited can we say that a and b are (propositionally) equal, and this inhabitation comes in the form of a term for the type $a =_A b$, i.e. $p : a =_A b$ and p should be viewed as evidence for the equality to hold.

[1]since equality is a proposition and proposition are types

Type Theory

Equalities
Judgmental equality or *Definitional equality* is equality of the syntax and is denote by $:\equiv$.
For example,

$$(x+1)^2 :\equiv x^2 + 2x + 1$$

since it is easily obtained by algebraic expansion.

Propositional equality on the other hand is though of as a type[1]. For terms $a, b : A$, we have the type "$a =_A b$". Only when $a =_A b$ is inhabited can we say that a and b are (propositionally) equal, and this inhabitation comes in the form of a term for the type $a =_A b$, i.e. $p : a =_A b$ and p should be viewed as evidence for the equality to hold.

[1]since equality is a proposition and proposition are types

From the HoTT book, it says that it is helpful to think of this meaning "equality by definition" which is really exactly what it says.

Function Types

Let A and B be some types, then the type $A \to B$ is formed with domain A and codomain B. For $f : A \to B$, giving it $a : A$ will produce an inhabitant of B, denoted $f(a)$.

The terms of this type is constructed by direct definition or $\lambda-$abstraction. Direct definition means we simply define $f : A \to B$ by giving an equation

$$f(x) :\equiv \Phi \tag{1}$$

where Φ is an expression which may contain x. Formation using $\lambda-$abstraction allows us to skip the naming of the term by writing

$$\lambda x.\Phi : A \to B$$

which is the same term in (1). For the construction to be valid, we need to verify that $\Phi : B$ given $x : A$.

Type Theory

Function Types

Let A and B be some types, then the type $A \to B$ is formed with domain A and codomain B. For $f : A \to B$, giving it $a : A$ will produce an inhabitant of B, denoted $f(a)$.

The terms of this type is constructed by direct definition or $\lambda-$abstraction. Direct definition means we simply define $f : A \to B$ by giving an equation

$$f(x) :\equiv \Phi \tag{1}$$

where Φ is an expression which may contain x. Formation using $\lambda-$abstraction allows us to skip the naming of the term by writing

$$\lambda x.\Phi : A \to B$$

which is the same term in (1). For the construction to be valid, we need to verify that $\Phi : B$ given $x : A$.

- unlike set theory, functions are primitive concepts in type theory; we explain what functions types are by prescribing what we can do with functions, how to construct them and what equalities they induce.
- in set theory, a function $f : A \to B$ are defined as a subset $f \subseteq A \times B$
  - for all $a \in A$ there exist $b \in B$ such that $(a, b) \in f$
  - for all $a \in A$ and $b, b' \in B$, if $(a, b) \in f$ and $(a, b') \in f$, $b = b'$.

We can do computation by using either the definition or $\lambda-$abstraction.

$$f(a) \equiv \Phi' \equiv (\lambda x.\Phi)(a)$$

where Φ' is the expression Φ in which all the occurrences of x have been replaced by a.

Lastly, for any $f : A \rightarrow B$ we can construct a $\lambda-$abstraction $\lambda x.f(x)$, we can consider it to be definitionally equal to f

$$f :\equiv (\lambda x.f(x))$$

and this equality is the *uniqueness principle of function types*, because it shows that f is uniquely determined by its values.

Example

Let both A, B be the type of natural numbers, $\mathbb{N}$.

Direct definition means we simply define $f : \mathbb{N} \to \mathbb{N}$ by giving an equation

$$f(x) :\equiv x + x \tag{2}$$

where Φ is an expression which may contain x. Formation using $\lambda-$abstraction allows us to skip the naming of the term by writing

$$\lambda x.x + x : \mathbb{N} \to \mathbb{N}$$

which is the same term in (2). We can do computation by

$$(\lambda x.x + x)(2) \equiv 2 + 2$$

Type Forming Rules:

1. **Formation.** Tells us how to form new types.
2. **Introduction.** Also known as a *constructor*, which specifies how to construct terms of a newly formed type.
3. **Elimination.** How to use the terms of that type.
4. **Computation.** Expresses how an eliminator acts on a constructor.
5. **Uniqueness Principle.** Expresses the uniqueness of maps into or out of that type.

Inference Rules

$$\frac{\vdash X : Type \qquad \vdash A : Type}{\vdash (X \to A) : Type} \qquad \text{(Type Formation)}$$

$$\frac{x : X \vdash a(x) : A}{\vdash (x \mapsto a(x)) : X \to A} \qquad \text{(Type Introduction)}$$

$$\frac{\vdash f : (X \to A) \qquad \vdash x : X}{x : X \vdash f(x) : A} \qquad \text{(Type Elimination)}$$

$$\frac{x : X \vdash a(x) : A \qquad \vdash x : X}{\vdash (\lambda(x : X).a)(y) \equiv a[y\,/\,x] : A[y\,/\,x]} \qquad \text{(Type Computation)}$$

$$\frac{\vdash f : (X \to A)}{\vdash f \equiv (\lambda x.f(x)) : X \to A} \qquad \text{(Uniqueness)}$$

the $\vdash$ symbol is called the turnstile and for $x : A \vdash a(x) : A$, it should be read as

> *In the context of variable x of type X, the expression $a(x)$ has type A.*

Comparison of the Different Points of View

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\varnothing, \{\varnothing\}$	$\varnothing, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \implies B$	set of functions	function space
$\sum_{x:A} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{x:A} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{(x, x) \mid x \in A\}$	path space of A^I

Table 1: Comparing the points of view on type-theoretic operations.

References

- J. Macor.
  A Brief Introduction to Type Theory and the Univalence Axiom
  http://math.uchicago.edu/ may/REU2015/REUPapers/Macor.pdf
- The Univalent Foundations Program
  ***Homotopy Type Theory: Univalent Foundations of Mathematics*.**
  https://homotopytypetheory.org/book
- The n-Category Café
  From Set Theory to Type Theory
  https://golem.ph.utexas.edu/category/2013/01/from_set_theory_to_type_theory.html
- The *n*Lab
  Function Type
  https://ncatlab.org/nlab/show/function+type