Navigating the Pachner Graph: Algorithms for Searching and Sampling Triangulations

Daniel Bruwel

An essay submitted in partial fulfillment of
the requirements for the degree of
B.Sc. (Honours)

Pure Mathematics
University of Sydney

October 2025

CONTENTS

Introduction

describing the objective and contents of the essay [**1**]

CHAPTER 1

Preliminaries

1.1. Mathematical Preliminaries

1.1.1. Manifolds. A $n-$*dimensional manifold* or a $n-$*manifold* is a topological generalisation of a surface, they are topological spaces that are "well behaved" and locally homeomorphic to $\mathbb{R}^n$. We recall the following basic definitions before precisely defining a manifold.

Definition 1.1 (Second Countable)**.** *A Topological space $\mathcal{T}$ is Second Countable if the topology $\tau_{\mathcal{T}}$ of $\mathcal{T}$ has a countable base.*

Definition 1.2 (Hausdorff)**.** *A Topological space $\mathcal{T}$ is Hausdorff if for any two points $x \neq y$ in $\mathcal{T}$ there are neighbourhoods U of x and V of y such that $U \cap V = \varnothing$*

With these we can define

Definition 1.3 ($n-$dimensional manifold)**.** *A $n-$dimensional manifold is a Hausdorff, second countable, topological space $\mathcal{M}$ where for every point $p \in \mathcal{M}$ there is a neighbourhood $U(p)$ of p that is Homeomorphic to an open subset of Euclidean space $\mathbb{R}^n$*

Remark 1.4. We have used a Homeomorphism to an open set of R^n, this is equivalent to saying that each point is either isolated (if $n = 0$) or has a neighbourhood that is Homeomorphic to all of $\mathbb{R}^n$

Remark 1.5. We will refer to these simply as Manifolds, compared to other areas of research these may be called *topological manifolds* to emphasise that they do not have any further structure (c.f. *differentiable manifolds, Riemannian manifolds, etc.*)

Some examples of Manifolds include the circles S^1, the sphere S^2, the Torus T^2, Euclidean space $\mathbb{R}^n$. Some non-examples include the disk D^2 because its boundary has no neighbourhood that is homeomorphic to an open subset of R^n, the figure-eight curve because at the "crossing" there is no way to deform it to look like an open subset of $\mathbb{R}$, or the line with two origins because this is clearly not Hausdorff [**?**].

In the study of triangulations we are often interested in a specific category of manifolds known as *piecewise-linear manifolds* or *PL-manifolds*

these are manifolds that require each point p to have a neighbourhood $U(p)$ of p that is a piecewise-linear deformation of $\mathbb{R}^n$. We say that two PL-manifolds are *PL-homeomorphic* if they can be transformed into each other via piecewise-linear deformations. This is all more formally defined in terms of charts, atlases, and transition maps which can be found in almost any introductory book on geometric topology (e.g. [**?**]).

A particularly useful result due to Tibor Radó and Edwin Moise for low dimensional topology says that for dimension 3 or less PL-manifolds and PL-homeomorphisms are the "same" as manifolds and homeomorphisms. More precisely

Theorem 1.6 (Hauptvermutung)**.** *For dimension 3 or less, every manifold is homeomorphic to a unique PL-manifold up to PL-homeomorphism.*

The proof of this theorem can be found in e.g. [**4, 3**]

Remark 1.7. This theorem does not hold for *four* or more dimensions [**2**].

1.1.2. Triangulations. We begin by defining *simplices* - the fundamental building blocks of triangulations

Definition 1.8 (simplices)**.** *An $n-$simplex Δ is the $n-$dimensional convex hull of $n+1$ vertices.*

A $0-$simplex is a point, a $1-$simplex a line, a $2-$simplex a triangle, and a $3-$simplex a tetrahedron. One way to construct an $n-$simplex is by taking the *join* of n points where we recall that

Definition 1.9 (Join)**.** *For two topological spaces X and Y, the join denoted $X * Y$ is constructed by taking $X \times Y \times [0,1]$ and quotienting by the equivalence relation that $(x, y, 0) \sim (x, y', 0)$ and $(x, y, 1) \sim (x', y, 1)$ for $x, x' \in X$ and $y, y' \in Y$.*

Any subset of $i+1$ points of an $n-$simplex forms another simplex that we call an $i-$*dimensional face*.

We can takes simplices and "glue" them together to form more complex geometric structures. Formally this is done via *face gluings*

Definition 1.10 (face gluing)**.** *For two $n-$simplices Δ_1 and Δ_2 and two $i-$dimensional faces $F_1 \subseteq \Delta_1$ and $F_2 \subseteq \Delta_2$ we define the face gluing of F_1 to F_2 by taking $\Delta_1 \sqcup \Delta_2 / \sim$ where $\sim$ is defined by a gluing map which is a homeomorphism between F_1 and F_2*

We typically create the gluing map by taking a bijection between the vertices of F_1 and F_2 and linearly interpolating to create the full homeomorphism.

We can easily extend this to define gluings of faces of the same triangulation, or even a gluing of a face to itself.

Definition 1.11 (PL-Sphere)**.** *The PL $0-$sphere is a pair of isolated points. For integers $n > 0$ the PL $n-$sphere is a PL manifold homeomorphic to the $n-$sphere.*

Definition 1.12 (PL-Triangulation)**.** *Construct $\mathcal{T}$ from a finite collection of $n-$simplices $\{\Delta_1, ..., \Delta_k\}$, called facets, by gluing their $n-1-$dimensional faces together. We call this a PL-triangulation if the following conditions are met*

a) *If a face is glued to itself, the face gluing must happen along the identity map.*
b) *For all vertices v, the set of all simplices Δ such that the join $v * \Delta$ is in $\mathcal{T}$ forms a PL $n-1-$sphere*

This is a PL-triangulation of a manifold $\mathcal{M}$ if there is a homeomorphism from $\mathcal{T} \to \mathcal{M}$.

1.1.3. Isomorphism Signatures. We say that two PL-triangulations are *combinatorially isomorphic* if they are the same up to relabelling of vertices. Because there is no interesting mathematical information in the labelling, we typically define a canonical labelling. While the exact details of constructing a canonical labelling are specific and can depend on the software choice, the general idea is as follows.

a) For each facet, calculate some label independent invariant (e.g. the number of unique facets being glued to).
b) Partition the facets into labelled bins based on this invariant (this labelling is canonical based on the value of the invariant)
c) For each facet in each bin (containing multiple facets), look at which bins the facets neighbouring (from gluing) facets are in, use this information to construct a new label, if there are facets in the same bin that get a different new label, split the bin in some canonical way and create a new labelling.
d) Repeat the above process, constructing a splitting tree. This tree is canonical. This tree is not guaranteed to split the bins into singletons. If it gets "stuck", make an arbitrary choice and continue the algorithm until everything falls into a singleton. This is a possible canonical labelling.
e) if an arbitrary choice had to be made (or perhaps multiple), rerun the entire algorithm for each possible choice that could have been made, creating a small list of possible canonical labelling. Chose the lexicographically smallest labelling.

The details of the initial invariant, how the bins are labelled, and optimisations can be found in [**?**]. In the worst case scenario this process takes

$\mathcal{O}(n! \cdot k^2)$ where n is the dimension and k is the number of facets. For a $3-$dimensional triangulation the $n!$ term is small and this algorithm is fast.

Once a canonical labelling is found, an "isomorphism signature" is calculated. For a $3-$dimensional triangulation, the only information that needs to be stored is the number of tetrahedra, and for face of each tetrahedron what the destination tetrahedron is, and what the vertex order is. We do not need to specify the destination face because this can be determined by a combination of permutation labelling, looking at where the destinations faces are being mapped, and strict ordering. To efficiently store this, the following is done.

a) For each face of each tetrahedron, calculate $v_i = 6 \cdot$ destination id $+$ permutation id. This is valid due to their being 6 vertex permutations. This value will less than $6k$ for k facets.
b) Create a sequence $(v_0, v_1, \cdots, v_{4k-1})$ ordered based on the canonical labelling.
c) Compute $I = v_0 + v_1(6k) + v_2(6k)^2 + \cdots$
d) Convert I to base 64 and store as a string.
e) Convert k to some string and append to the front of the string representation of I.

This process form a string / number representing the unique combinatorial isomorphism signature for any given triangulation of a $3-$manifold (practically there are size restrictions on the number of facets k).

1.1.4. Computational Complexity.

Definition 1.13 (Time Complexity)**.** *For an algorithm with input size n, the time complexity is the asymptotic worst case runtime of the algorithm, written in "Big-O" notation as $\mathcal{O}(g(n))$.*

Definition 1.14 (Space Complexity)**.** *For an algorithm with input size n, the space complexity is the asymptotic worst case amount of memory the algorithm needs to allocate, written in "Big-O" notation as $\mathcal{O}(g(n))$.*

Definition 1.15 (Decision Problem)**.** *A decision problem is a type of problem that can be answer as either "yes" or "no"*

Remark 1.16. While most problems are not decision problems, it is often possible to convert many problems into a decision problem. For example, if the problem is to multiply two numbers together, we can convert this to a decision problem of "is the ith binary digit a 0?", this can be repeated for all digits.

Definition 1.17 (Complexity Class)**.** *A complexity class is a class of problems that can be solved under some model of computation within some resource bound i.e. some time or space bound. A problem belongs to a given*

complexity class if there exist an algorithm to solve the problem within this bound. For "turning machines", the main complexity classes are

a) L*: The problem can be solved using a logarithmic amount of space.*
b) NL*: A solution can be verified using a logarithmic amount of space.*
c) P*: The problem can be solved in polynomial time.*
d) NP*: A solution can be verified in polynomial time.*
e) $PSPACE$*: A solution can be found using a polynomial amount of space.*
f) $EXPTIME$*: A solution can be found using an exponential amount of time.*

There exist many other complexity classes for both turning machines and other models for computation.

Remark 1.18. A turning machine is a common model for computation, we do not define it here but there are many resources available.

Remark 1.19. We have defined NL and NP as having solutions verifiable in a specific bound, this means that if the question is for example, is there a path of length k between two nodes, someone can give a path that is a solution, and it is possible to verify that this connects the two nodes and has length k. Formally NL and NP are not defined this way, but instead using things called "non-deterministic Turing machines", but the definitions can be shown to be equivalent.

Theorem 1.20. $L \subseteq NL \subseteq P \subseteq NP \subseteq PSCPAE \subseteq EXPTIME$

Remark 1.21. There are conjectures that some of these subsets may be proper subsets or equalities, most famously the question of if $P = NP$, but there are not as of yet any solutions.

Definition 1.22 (Hard Problem)**.** *For a given complexity class C, a problem P is called a $C-$hard problem if every problem $P' \in C$ can be "reduced" to P via some algorithm that is "efficient" relative to C.*

Remark 1.23. We do not formally define what it means to be "efficient" relative to C as it is getting into the details too much, but for NP and $PSCPAE$ problems, a polynomial time reduction is efficient.

Definition 1.24 (Complete Problem)**.** *For a given complexity class C, a problem P is called a $C-$complete problem if it is a $C-$hard problem and $P \in C$*

Practically, problems that belong to P can be solved tractable on a modern computer, we can typically scale resources and solve the problem. However, all the algorithms we have for NP complete problems require exponential resources to run, which becomes computationally infeasible quickly.

As such, if we find a problem to be $NP-$complete or harder, it means we can't just "throw more compute" at the problem to solve it and may have to resort to suboptimal solutions.

1.1.5. Pachner Moves and the Pachner Graph. A Pachner move is a specific process that changes a PL-manifold without changing the underlying topological space.

Definition 1.25 (Complementary Triangulation)**.** *Let Δ a $d+1$simplex, let A be a a connected sub complex of $\partial\Delta$ - that is a set of $d-$simplexes that form a connected topological space. The complementary triangulation of A is $B = \partial\Delta \backslash A$.*

From this we can abstractly define a Pachner move

Definition 1.26 (Pachner Move)**.** *For a d-dimensional PL-Manifold, identify an $i-$dimensional simplex σ^i, the set of all facets that contain σ^i form a sub complex of the boundary of some $d+1$dimensional triangulation. Replace this set with the complementary triangulation of this sub complex. This Pachner move is called an $i-$move.*

Theorem 1.27. *Two triangulations $\mathcal{T}$ and $\mathcal{T}'$ are reachable in finitely many Pachner moves if and only if they represent the same PL-manifold.*

For triangulations of $3-$manifolds, we can take $i = 0, 1, 2, 3$, where $i = 0, 3$ are inverses of each other, and likewise $i = 1, 2$. For a $2-$move, we identify a triangle and find all the tetrahedra that share that triangle, there are two of these, we replace these with 3 tetrahedra that now share a common edge. The $1-$move is the inverse of this. We often call these moves $(3, 2)$ and $(2, 3)$ moves to explicitly state how we are going, for example, from 3 tetrahedra to 2. For a $0-$move, we identify a vertex with 4 tetrahedra that share it, you replace these with a single tetrahedron. The $3-$move is the inverse. These are often called $(4, 1)$ and $(1, 4)$ moves as above.

Because all triangulations of a specific PL-manifold can be reached through Pachner moves, we define

Definition 1.28 (Pachner Graph)**.** *For a given PL-manifold $\mathcal{M}$, the Pachner graph for $\mathcal{M}$ us the graph (V, E) given by $V = \{$triangulations of $\mathcal{M}\}$ and for $v_i, v_j \in V$, $e_{ij} = (v_i, v_j) \in E \iff v_i, v_j$ are related via a single Pachner move.*

Because of the above theorem, the Pachner graph is fully connected for any PL-manifold. The Pachner graph is however, infinite. We additionally endow the Pachner graph with "levels" where each level encodes the number of facets in the triangulation.

Theorem 1.29. *The Pachner graph grows super exponentially in its level.*

We also have that

Theorem 1.30. *The shortest path between v_i and v_j two vertices of the Pachner graph is PSPACE hard problem.*

Remark 1.31. It is not currently known if the graph traversal is $PSPACE$ complete

We notice that $(4,1)$ and $(1,4)$ moves change the number of vertices, but the $(3,2)$ and $(2,3)$ moves do not. We are often interested in "single vertex" triangulations, for these we can exclusively use $(3,2)$ and $(2,3)$ moves. A particularly useful result is that

Theorem 1.32. *Any single vertex triangulation of the $3-$sphere S^3 can be reached through only $(3,2)$ and $(2,3)$ moves.*

We analogously define the Pachner graph for $1-$vertex triangulations of S^3, which is once again a fully connected graph. Unlike the general Pachner graph, it is currently unknown if the Pachner graph for $1-$vertex triangulations of S^3 grow super exponentially or exponentially in its level.

1.2. Machine Learning Preliminaries

1.2.1. Markov Chain Monte Carlo.

Definition 1.33 (Markov Chain)**.** *A Markov Chain is a sequence of values $x(t)$ indexed by "time" where the sequence has the Markov Property - that is $x(t+1)$ depends only on $x(t)$, typically probabilistically.*

Monte Carlo methods are a class of methods that use random sampling to approximate something. Often, they are used to approximate the integral of some complex function, however not exclusively.

Markov Chain Monte Carlo is a class of methods to sample from a distribution by constructing a Markov Chain that's steady state solution is the distribution. More formally we define

Definition 1.34 (Markov Transition Kernel)**.** *Take $(\mathcal{X}, \mathcal{F})$ a measurable space with $\mathcal{F}$ a sigma algebra. A Markov Transition Kernel is a function $P : \mathcal{X} \times \mathcal{F} \to [0,1]$ such that*

a) *for any $x \in \mathcal{X}$, the function $A \mapsto P(x,A)$ is a probability measure on $(\mathcal{X}, \mathcal{F})$*
b) *for any $A \in \mathcal{F}$, the function $x \mapsto P(x,A)$ is a measurable function.*

The Markov Transition Kernel is used to "evolve" the state. That is, if you are currently "located" at $x \in \mathcal{X}$, $P(x,A)$ tells you the probability

of "ending up" in A. More precisely, if at time t the system is distributed according to ν_t, then the distribution at $t+1$ is

$$\nu_{t+1} = \nu_t P(A) = \int_{\mathcal{X}} P(x, A) d\nu_t(x) \tag{1.35}$$

Definition 1.36 (Target Distribution)**.** *π is called a target distribution for a Markov Transition Kernel P if the following two conditions hold*

a) ***Stationarity**: $\pi P = \pi$*
b) ***Ergodicity**: For $\pi-$almost all points x_0,*

$$\lim_{n\to\infty} \|P^n(x_0, \cdot) - \pi(\cdot)\|_{TV} = 0$$

Remark 1.37. A statement is true for "$\pi-$almost all x_0" means that it is true for all x_0 except a set that has measure 0 under π

Typically proving stationarity and ergodicity is difficult, so the following is used instead

Theorem 1.38. *For a Markov transition kernel P, if the following are true for some distribution π*

a) *$\pi-$**irreducibility**: For any $A \in \mathcal{F}$ with $\pi(A) > 0$, there exists some n such that $P^n(x, A) > 0$ for all $x \in \mathcal{X}$*
b) ***Aperiodicity**: The chain is aperiodic*
c) ***Recurrent**: For any measurable set $A \in \mathcal{F}$, and any starting point $x \in \mathcal{X}$ the hitting time τ_A is finite almost surely, where τ is the amount of steps to go from x to A.*
d) ***Reversibility**: For any two $A, B \in \mathcal{F}$ we have*

$$\int_A P(x, B) d\pi(x) = \int_B P(x, A) d\pi(x)$$

then π is the target distribution for the Markov transition kernel P.

In particular, if we have some process $\hat{P}$ that takes in some $x(t)$ and produces some $x(t+1) = \hat{P}(x(t))$ according to some Markov transition kernel P, we can create a Markov Chain, by continuously generating samples, and applying $\hat{P}$ to them recurrently, these samples will be "distributed according to" π. More precisely, the partial sums $S_n = \frac{1}{n}\sum_i^n f(x_i) \to \mathbb{E}^\pi[f]$ at a "rate" $\mathcal{O}(\sqrt{N})$.

Definition 1.39 (Metropolis Hastings)**.** *For some function f, not necessarily a probability density, and some "proposal function" g the Metropolis Hastings algorithm on f with proposal g is the following: Initialise some x_0, for each t do the following*

a) *Propose some x' according to $g(x'|x_t)$*
b) *Compute $\alpha = f(x')/f(x_t)$*

c) *Sample some* $u \in [0,1]$ *uniformly*
d) *If* $\alpha > u$ *accept* x' *by setting* $x_{t+1} = x'$*, otherwise set* $x_{t+1} = x_t$

Theorem 1.40. *If the proposal function* g *is symmetric - that is* $g(x|y) = g(y|x)$*, and if* f *is proportional to some probability distribution* π*, then the Metropolis Hastings algorithm for* f *with proposal* g *will generate samples according to* π*.*

Remark 1.41. If the proposal distribution is not symmetric, a factor known as the Hastings ratio can be introduced to find $\alpha_H = \alpha \cdot \frac{g(x'|x)}{g(x|x')}$ which is used in the acceptance step.

1.2.2. Simulated Annealing. For a measurable space $(\mathcal{X}, \mathcal{F}, \mu)$ with base measure μ (typically the Lebesgue, or count measure), we may have some measurable function $E : \mathcal{X} \to \mathbb{R}$ known as the energy. In thermal physics, for a thermodynamic system at temperature T, the probability of finding a particle in a given energy E is given by the Gibbs distribution $\pi_T(A) = \frac{1}{Z}\int_A e^{-E(A)/T} d\mu(x)$ where $Z = \int_{\mathcal{X}} e^{-E(A)/T} d\mu(x)$ is a normalising factor. Typically as a system "cools down" it "finds its way" to low energies. Simulated annealing is inspired by this thermodynamic concept, simulating a particle over time as the temperature $T \to 0$ heuristically conjecturing that it will find its way to the lowest energy state. The simulation at a given temperature T is typically done using the Metropolis Hastings algorithm.

While the concept above is inspired as a Heuristic from concepts in thermal physics, it has valid grounding in probability and machine learning. At a high level, this is because for any $x, x' \in \mathcal{X}$, $\pi_T(x')/\pi_T(x) = e^{-(E(x')-E(x))/T}$, which if $E(x') > E(x)$ tends to 0 as $T \to 0$, but if $E(x') < E(x)$ tends to ∞. As such, a sharp delta function is formed around $x_{min} = \arg\min(E)$. While this does mean that at $T = 0$ we have that $\pi(x) = \delta(x - x_{min})$ is the stationary solution to our Metropolis Hastings Markov Transition Kernel, at $T = 0$ the measure of valid starting points $\{x_0\}$ tends to have measure 1 under π but measure 0 under ν. Informally, at low T we almost always accept points where $E(x') < E(x)$ and almost never accept points where $E(x') > E(x)$, so this is just hill climbing and unless E is convex, the sampling will likely get stuck in a local minima.

1.2.3. Transformers.

Definition 1.42 (Row Operator)**.** *For a field* F*, and a function* $s : F^n \to F^n$ *define the row operator* $\mathcal{R}_s : M_n(F) \to M_n(F)$ *as the act of applying* s *to each row of* $M_n(F)$*. That is*

$$\mathcal{R}_s\left((r_1, r_2, ..., r_n)^T\right) = (s(r_1), s(r_2), ..., s(r_n))^T$$

Definition 1.43 (Abstract Transformer Head)**.** *For a collection of vectors $(v_1, v_2, ..., v_n)$ where $v_i \in V$ an vector space over a field F. Form $\dot{V}$ by equipping V with a bilinear form denoted $B : V \times V \to F$. Let $G : F^n \otimes V \to M_n(F)$ denote the "Gram operator". Let $L : V \to W$ be an arbitrary linear operator, and $s : F^n \to F^n$ an arbitrary function.*

An abstract transformer head is a function $H : F^n \otimes V \to F^n \otimes W$ defined by

$$H(x) = (\mathcal{R}_f(G(x)) \otimes L)(x) \tag{1.44}$$

Definition 1.45 (Abstract Multi-Head Attention)**.** *For a collection of vectors $(v_1, v_2, ..., v_n)$ where $v_i \in V$ an vector space over a field F. Form spaces $V_1, V_2, ..., V_q$ from V by equipping V with a, possibly unique, inner product. Let $\iota_i : V \to V_i$ be the operation of endowing V with the inner product structure of V_i. Let H_i denote the transformer head from $F^n \otimes V_i \to F^n \otimes W_i$ where W_i, W_j may be distinct. Define $O : W_0 \oplus W_1 \oplus \cdots \oplus W_q \to V$*

The multi-head attention is an operation $A : F^n \otimes V \to F^n \otimes V$ defined as

$$A(x) = (I \otimes O)\left(\bigoplus_i H_i(\iota_i(x))\right) \tag{1.46}$$

Definition 1.47 (Abstract Feedforward Neural Network)**.** *For a vector space U over a field F, a Feedforward is a operator $N : U \to W$ for a pair of linear operators $L_1 : U \to V$ and $L_2 : V \to W$ and a non-linear function $a : V \to V$ is $N(x) = L_2(a(L_1(x)))$*

Definition 1.48 (Abstract Transformer Block)**.** *For a collection of vectors $(v_1, v_2, ..., v_n)$ where $v_i \in V$ an vector space over a field F. A transformer block $T : F^n \otimes V \to F^n \otimes V$ is an operator defined as*

$$T(x) = x + A(x) + (I \otimes N)(x + A(x))$$

While these abstract formulations of the Transformer are theocratically rich, in practice we are working over $F = \mathbb{R}$ and our vector space V is finite dimensional. This allows us to write $V = \mathbb{R}^\kappa$, write all the linear transformations as matrix multiplications, and write the bilinear forms as $B(v_1, v_2) = v_2^T M V_1$ for some matrix M. Also, the non-linear functions used such as f in the abstract transformer head, and a in the feedforward neural network are fixed, or depend on some small set of learnable parameters. This gives a finite set of scaler parameters θ that defines the entire transformer block. Additionally, some further components are introduced into the transformer block to ensure computationally stability and regularisation. This lets us define

Definition 1.49 (Standard Transformer Head)**.** *For a collection of vectors $(v_1, v_2, ..., v_n)$ where $v_i \in \mathbb{R}^d$ we construct $X \in \mathbb{R}^{n\times d}$ by stacking these vectors up as columns. The head is calculated as*

$$h(X) = \text{softmax}\left(\frac{(XW^Q)(XW^k)}{\sqrt{d_k}} + M\right)(ZW^V) \tag{1.50}$$

Where the softmax *is applied along each row. The $W^Q, W^K, W^V \in \mathbb{R}^{n\times d_k}$ are called the "query", "key", and "value" weights. $M \in M_n(\mathbb{R} \cup \{-\infty, \infty\})$ is called the causal mask.*

Definition 1.51 (Standard Multi-Head Attention)**.** *For X defined as above, $MHA(X) = \text{Concat}(h_1(X), ..., h_h(X))W^O$*

Definition 1.52 (Standard Feedforward Neural Network)**.** *For $x \in \mathbb{R}^\kappa$, $FFN(x) = \sigma(xW_1 + b_1)W_2 + b_2$ where the W_i are called weights and the b_i are called biases. σ is a parameter-less "activation function".*

Definition 1.53 (Layer Norm)**.** *For a vector $x \in \mathbb{R}^d$ define μ and σ as the mean standard deviation of x. The layer norm of x is*

$$LN(x)_i = \gamma_i \cdot \left(\frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta_i \tag{1.54}$$

Where $\gamma, \beta \in \mathbb{R}^n$ are learned parameters, and ϵ is some fixed number for numerical stability.

Remark 1.55. This was not included in the abstract definition of the transformer block as it is primarily used for numerical stability.

Definition 1.56 (Standard Transformer Block)**.** *For X as defined above*

$$Y = X + MHA(LN(X)) + FFN(LN(X + MHA(LN(X)))) \tag{1.57}$$

Where Y is the result from the transformer block.

Because these transformer blocks map from $F^n \otimes V \to F^n \otimes V$, or from $\mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$, they can be chained together.

For many tasks which transformers excel in (language modelling, time series forecasting) the input sequence is sequential, and often not encoded into a vector in a meaningful way. As such we have

Definition 1.58 (Token Embedding)**.** *For an input vectors $v \in \mathbb{R}^V$, the token embedding of v is $h = x^T E$ where $E \in \mathbb{R}^{V\times d}$.*

Remark 1.59. For tasks where the input is categorical, v is usually formed with some one-hot-encoding, that is if there are V categories we define for

category c $v_c = (0, ..., 0, 1, 0, ..., 0)$ as a vector in $\mathbb{R}^n$ that has a 1 in the position corresponding to category c (arbitrary), and zeros elsewhere.

Often (e.g. language modelling) there are many categories, so $d < V$ and the embedding becomes a useful technique for encoding "semantic" meaning into a vector.

Once we have embedded our vectors, there is still often a sequential order to them, i.e $(v_0, v_1, v_2, ...)$. The transformer block as described above, is invariant under changing of this order. As such, we use

Definition 1.60 (Positional Encoding)**.** *For a sequence of vectors indexed by time, $(v_0, v_1, ..., v_n)$ each $v_n \in \mathbb{R}^d$. For T_{max} the longest possible sequence, and a positional embedding matrix $P \in \mathbb{R}^{T_{max} \times d}$, the positional embedding of $(v_0, v_1, ..., v_n)$ is defined with $v_i \mapsto v_i + P_i$ where P_i is the ith row.*

A prototypical example of applying these steps is in GPT-2, described as follows. For a sequence of words $(w_1, w_2, ..., w_n)$, each word in a vocabulary of size V, encode each word via one hot encoding. Perform token embedding into $\mathbb{R}^d$, and perform a positional encoding. Apply a series of transformer blocks sequentially, apply a single layer norm, apply a linear projection back into $\mathbb{R}^V$ for each "word" (via some matrix multiplication), and finally apply softmax to each output "word". The final vector is trained to represent the probability of the next word in the sequence of words. So for position 1 the prediction is for w_2, for position n the prediction is for word w_{n+1} not in the data. To train GPT-2 we use an objective function that calculates the cross entropy between the prediction at each step and the actual next word, this is done for each "word" in the output layer.

1.2.4. Gradient Descent. If we have a neural network $N(x \mid \theta)$ that takes in some input $x \in V$ and some parameter set $\theta \in \mathbb{R}^n$ and produces some output $y \in W$ where V, W are vector spaces over a field F, typically $\mathbb{R}$, and typically finite dimensional. We typically have some "training data", which is a set of $\{(x_i, y_i) \mid x_i \in V, y_i \in W\}$ where each y_i is the output that we "want" from $N(x_i, \theta)$, what we mean by "want" is that some function $\mathbb{E}_{x_i}[\mathcal{L}(N(x_i, \hat{\theta}), y_i)]$ is minimised by our parameter choice $\hat{\theta}$. To find this target $\hat{\theta}$. Because N is typically complicated, and has a complex dependence on θ, we cannot directly minimise this objective function to find $\hat{\theta}$, as such we use gradient decent. The concept behind gradient decent is to take some θ_i, find a local, linear approximation of $\mathbb{E}_{x_i}[\mathcal{L}(N(x_i, \hat{\theta}_i), y_i)]$, and find some "penalty" for "trusting" the approximation too much. Precisely, we

compute

$$\theta_{i+1} = \arg\min_{\theta} \left(\langle g^T, (\theta - \theta_i) \rangle + \frac{\lambda}{2} \|\theta - \theta_i\|^2 \right) \tag{1.61}$$

This is done recursively, typically for some number of steps. Here g is the gradient, often we would consider using $\nabla_\theta \mathbb{E}_{x_i}[\mathcal{L}(N(x_i, \hat{\theta}_i), y_i)]$, but computing this over the entire "training" data set is slow, so we typically a small "batch" of data is taken from the training data and the gradient is taken over this batch. Because this batch is "random" a small subset of the true data, our estimate of the gradient is going to be somewhat stochastic, as such many gradient decent algorithms use some sort of exponential smoothing step for the gradient - this is often called the momentum. Different choices of exponential smoothing, and different choices of the norm, and different choices of the gradient lead to different types of optimisers. A few common ones are *SGD* which uses the frobenius norm, the standard gradient, and no smoothing; *adam*, which uses the $\ell_1 \to \ell_\infty$ norm, the standard gradient, and exponential smoothing; *shampoo*, which uses the spectral norm, the standard gradient, and exponential smoothing; and *NGF* which uses the frobenius norm, the natural gradient, and no exponential smoothing.

1.2.5. Reinforcement Learning. For our specific usecase, reinforcement learning is a technique used when we have some "reward function" $R(y) : Y \to \mathbb{R}$ that tells us how good a specific sample is, and we want to update the parameters of some probability distribution / sampling function $Y \sim \pi_\theta$ to maximise the expected reward, that is

$$J(\theta) = \mathbb{E}_{y \sim \pi_\theta}[R(y)] \tag{1.62}$$

While reinforcement learning is typically more general than this, and applies to policy learning, this will suffice for our purpose.

The most direct way to update π_θ is to simply take a number of samples from π_θ and calculate

$$g = \nabla_\theta J(\theta) = \mathbb{E}_{y \sim \pi_\theta}[\nabla_\theta \ln(\pi_\theta) R(y)] \tag{1.63}$$

and then use this g in gradient decent as described above.

While the above techniques makes logical sense, the use of $J(\theta)$ being the expected reward can become problematic, for example if our reward function is binary, returning a score of 1 if the output meets some criteria, and returning a score of -1 if it does not, once the parameter set has settled into a stable configuration where each output is valid and produces a score of 1, $J(\theta)$ continues to update θ which can lead to oscillations. As such we often use an "advantage" function A which is an estimate of how the

score $R(y_i)$ for a specific sample compares to the expected score. That is $A(y_i) = R(y_i) - \mathbb{E}_{y\sim\pi_\theta}[R(y)]$. We can estimate the expected score by simply sampling from π_θ a few times, and taking the average, and for a simple problem this is sufficient.

Remark 1.64. This average often has a high variance, so a "critic" model is often trained instead to try and learn the baseline. We won't explore this technique here.

We can use the advantage to get a refined objective function $J(\theta) = \mathbb{E}_{y\sim\pi_\theta}[A(y)]$. We can calculate the gradient once again as above, and apply gradient accent. The primary disadvantage here is that each time we take a gradient step, we have to resample from π_θ to get a new batch to calculate the new gradient. The new distribution with parameter set θ will only be slightly different to the original parameter set before the gradient update θ_{old}, and as such we can keep the old samples $y_i \sim \pi_{\theta_{old}}$ and then use a technique called importance weighting to find that

$$\mathbb{E}_{y\sim\pi_\theta}[A(y)] = \mathbb{E}_{y\sim\pi_{\theta_{old}}}[r(\theta)A(y)] \tag{1.65}$$

Where $r(\theta) = \frac{\pi_\theta(y)}{\pi_{\theta_{old}}(y)}$ is called the importance ratio. This allows us to take batch of samples from $\pi_{\theta_{old}}$, perform a few steps of gradient accent using this batch, importance reweighing at each step. While this works in expectations, typically after a few gradient steps our samples are no longer representative, and our reweighing erodes. Therefore, it is typical to use a "trust region" wherein we can trust an update. One of the most common ways to do this is to "cap" positive updates by restricting $r(\theta) \leq 1 + \epsilon$ if $A(y) > 0$, that is if if we generated good samples from θ_{old} we want to increase the likelihood of generating them, but not so much so that we can no longer trust the importance sampling as the ratio $r(\theta)$ has become too big, but on the other hand, if the advantage $A < 0$, we want to continue to have a learning signal. This yields the proximal policy optimisation (PPO) objective

$$L^{CLIP}(\theta) = \mathbb{E}_{y\sim\pi_{\theta_{old}}}[\min\left(r(\theta)A, \text{clip}(r, 1-\epsilon, 1+\epsilon)A\right)] \tag{1.66}$$

CHAPTER 2

Search and Sampling Algorithms

We are often interested in exploring the characteristics and properties of triangulations of a manifold $\mathcal{M}$. Due to the unwieldy size of $\mathcal{T}(\mathcal{M})$, it is typical to sample from $\mathcal{T}(\mathcal{M})$ and study these samples. This is the task of a sampling algorithm. There are particular situations in which we may want to find a triangulation that has a particularly "high score". This is a search task, with an "objective function" $O : \mathcal{T}(\mathcal{M}) \to \mathbb{R}$ where we aim to find a triangulation $\Delta \in \mathcal{T}(\mathcal{M})$ such that $O(\Delta)$ is "high" (we use high somewhat informally, as we do not know much of the geometry of different objective functions over $\mathcal{T}(\mathcal{M})$).

2.1. Classical Techniques

2.1.1. Direct Accent. For an objective function $O : \mathcal{T}(\mathcal{M}) \to \mathbb{R}$, for small triangulations (number of tetrahedra around 6 or lower), we can typically calculate O exhaustively from a census. In doing this, we may find that O tends to increase with more tetrahedra, this leads to the notion of direct accent. The basic idea is to start at some small triangulation, enumerate all of its neighbours that have more tetrahedra, and then sample one of these according to its score. This process can then be repeated to generate a chain of tetrahedra of increasing size, and the whole process repeated to generate multiple chains. The detailed algorithm is described in algorithm 1

2.1.2. Markov Chain Monte Carlo.

2.1.3. Simulated Annealing. Because the above Markov Chain Monte Carlo algorithm leads to a uniform distribution, we can use it as a symmetric proposal distribution **(need to check this)**. We implement a standard simulated aneling algorithm as described in algorithm 2

2.2. Transformers

2.2.1. Base Transformer. We use a standard GPT-2 style transformer with dropout and pre-layer normalisation. The final transformer model takes the sequence, performs token and positional embedding, dropout, and then n_{layers} of the standard transformer block as described in definition **??**.

Algorithm 1 Direct Accent

Let Δ_0 be the initial triangulation.
Let $\beta > 0$ be a fixed parameter.
let $T \in \mathbb{N}$ be a fixed chain size.
for $t = 1$ to T **do**
  Generate all neighbours Δ_{ti} of Δ_{t-1} by applying $(2-3)$ or $(1-4)$ moves.
  for each neighbours Δ_{ti} **do**
    Calculate the percentage advantage $A_i = \frac{O(\Delta_{ti}) - O(\Delta_{t-1})}{O(\Delta_{t-1})}$.
  end for
  for each neighbours Δ_{ti} in the set **do**
    Calculate the selection probability $p_i = \frac{e^{\beta A_i}}{\sum_j e^{\beta A_j}}$.
  end for
  Sample Δ_t from the set of neighbours $\{\Delta_{ti}\}$ with probability p_i.
end for
Return Δ_T

Algorithm 2 Simulated Annealing

Let Δ_0 be some starting triangulation.
Let T be the chain length.
Let M be some hash table memory.
Let $O : \Delta \to \mathbb{R}$ be some objective function.
Let T_t be some temperature schedule.
for $t = 1$ to T **do**
  Propose $\tilde{\Delta}_t$ from Δ_{t-1} by using $1-$step of the described MCMC algorithm.
  Retrieve $o_{t-1} = O(\Delta_{t-1})$ from M
  Check if $o_t = O(\tilde{\Delta}_t)$ is in M, if it is - retrieve it, if not - compute it and store it in M.
  Compute $\alpha = e^{-(o_{t-1} - o_t)/T_t}$
  Generate some $p \sim \mathcal{U}([0,1])$
  if $p \leq \alpha$ **then**
    Accept $\Delta_t = \tilde{\Delta}_t$
  else
    Let $\Delta_t = \Delta_{t-1}$
  end if
end for

Between each transformer block there is a dropout of 0.1 applied. This architecture is the standard GPT-2 style architecture. We use an embedding dimension of 64, and the feed forward neural networks are also all of dimension 64. The vocabulary size includes all the letters of the training data, plus three special tokens "[BOS]", "[EOS]", and "[PAD]" for the beginning of sentence, end of sentence, and padding respectively. For training we use AdamW, with a cross entropy objective function, and a batch size of 32.

2.2.2. Reinforcement on the Topology. The transformer may not generate sequences which represent our target manifold, or any manifold at all. As such after training is complete, reinforcement learning can be applied with a score function which returns 1 if the triangulation is of the correct manifold type, and 0 if it is not. A softer scoring function can be used that returns intermediate values depending on if the isomorphism signature corresponds to an invalid structure, a valid manifold of the wrong type, or a valid manifold of the correct type.

2.2.3. Reinforcement on the Objective. Once the transformer is generating samples with a relatively high chance of being valid isomorphism signatures, if we have a specific objective function $O : \Delta \to \mathbb{R}$ which we seek to maximise, we can use this score function in the standard reinforcement learning algorithms described in **??**.

CHAPTER 3

Numerical Experiments

3.1. Classical Optimisation on the Alexander Polynomial

We consider single vertex triangulations of S^3, call the set of all of these $\mathcal{T}_1(S^3)$. For any $\Delta \in \mathcal{T}_1(S^3)$, we can consider an edge e_α, because the triangulation has only one vertex, e_α forms a loop. This loop can be knotted in S^3. Call the associated knot for e_α K_α. The Alexander polynomial $\Delta_{K_\alpha}(t)$ has coefficient vector $\vec{a} = [a_0, a_1, ..., a_n]$. It is known that "more complex" Alexander polynomials lead to "more complex" knots. One example of this is that $2g(K) \geq \deg(\Delta_K(t))$, another is that the determinant $|\Delta_K(-1)|$ related to the knot colouring. This inspires two objective functions on $\mathcal{T}_1(S^3)$ that we may be interested in optimising. Firstly we have $O_{deg} : \mathcal{T}_1(S^3) \to \mathbb{R}$ found by taking the average degree of the Alexander polynomial across all edges. Another is $O_{det} : \mathcal{T}_1(S^3) \to \mathbb{R}$ computed by taking the average determinant of the Alexander polynomial for each edge. We seek to optimise these objectives as they will be correlated with triangulations that have "more complex" knot structure.

3.1.1. Markov Chain Monte Carlo. As a baseline, we use Markov Chain Monte Carlo to generate a sample of triangulations, this is used to examine how much better our search algorithms are compared to the "average" triangulation.

3.1.2. Direct Ascent. We use the direct accent algorithm as described in section **??**, starting with the census of all $\Delta \in \mathcal{T}_1(S^3)$ that are formed with 5-tetrahedra. There are ... of these, for each one we perform direct accent for 20 steps to get a triangulation with 25 tetrahedra. This is done with both objective functions for a range of different temperatures.

3.1.3. Simulated Annealing. We perform simulated annealing on O_{det} and O_{deg}. We use a constant temperature, and a cooling schedule of ....

3.2. Classical Optimisation on Number of Generators of the Fundamental Group

3.3. Baseline Transformer Efficiency on Isomorphism Signatures

We are interested in how the efficiency (percentage of valid samples) scales as the number of tetrahedra increases. It is understood that as the number of tetrahedra increases, the percentage of valid triangulations compare to the percentage of writeable isomorphism signatures tends to zero. This means that any naive generation strategy becomes vanishingly unlikely to generate a valid isomorphism signature as the number of tetrahedra grows. We are therefore interested for a transformer of a fixed size, to see what the efficiency is for generating triangulations as the number of triangulations gets larger.

CHAPTER 4

Discussion and Future Work

...

References

[1] Eduardo G Altmann and Jonathan Spreer. Sampling triangulations of manifolds using monte carlo methods. *Experimental Mathematics*, pages 1–15, 2025.

[2] John W. Milnor. Two complexes which are homeomorphic but combinatorially distinct. *Annals of Mathematics*, 74(3):575–590, 1961.

[3] Edwin E. Moise. *Geometric Topology in Dimensions 2 and 3*, volume 47 of *Graduate Texts in Mathematics*. Springer-Verlag, New York-Heidelberg, 1977.

[4] Tibor Radó. Über den begriff der riemannschen fläche. *Acta Litterarum ac Scientiarum Universitatis Hungaricae Francisco-Josephinae, Sectio Scientiarum Mathematicarum*, 2:101–121, 1925.