Machine Learning Methods to Optimize the Geometry and Topology of Meshes

By

Arjun Narayanan

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Mechanical Engineering

in the

Graduate Division

of the

University of California, Berkeley

Spring 2024

Machine Learning Methods to Optimize the Geometry and Topology of Meshes

Abstract

Machine Learning Methods to Optimize the Geometry and Topology of Meshes

by

Arjun Narayanan

Doctor of Philosophy in Engineering - Mechanical Engineering

University of California, Berkeley

Professor Per-Olof Persson, Co-chair

Professor Shawn Shadden, Co-chair

Meshes are used ubiquitously in engineering for representing geometries, performing computational simulations, and generating computer graphics renderings. Automatically generating suitable meshes for downstream applications remains a key bottleneck in many workflows and often requires significant manual intervention. It is challenging to optimize mesh data-structures because they can be highly unstructured in their most general form.

A mesh has two fundamental attributes — geometry and topology. Geometry deals with the position and shape of objects in space. Topology is concerned with the connectivity of mesh elements. It is essential to optimize both of these attributes to generate a desirable mesh for a target application such as simulation. This dissertation explores machine learning methods to optimize both of these attributes.

The first part of this dissertation is concerned with mesh topology. We will describe a deep reinforcement learning framework to optimize the topology of 2D meshes using elementary mesh editing operations. The framework is trained purely in self-play reinforcement learning to optimize a given user defined objective function. We describe a novel neural network architecture that is able to encode the local topology of a mesh around a given mesh neighborhood. Subsequently, the neural network is trained to predict a probability distribution over the local action space in order to maximize the cumulative reward as prescribed by the given objective function. The agent is trained on randomly generated 2D polygonal shapes. We demonstrate generalization to inputs that were never seen during training. The proposed framework

is particularly effective at coarse block decomposition of polygonal shapes where the aim is to minimize the number of irregular vertices in the mesh.

We will then tackle the problem of geometry. We describe a deep learning method to automatically generate patient-specific, simulation ready 3D surface meshes of the human heart directly from clinical imaging. The proposed method is a two-stage mesh deformation process that transforms a given template mesh to match the underlying target geometry in the image data. The first stage consists of a learned affine transformation conditioned on the input image. This stage is trained to roughly align the template in terms of scale and orientation to the image data. The second stage consists of a learned local diffeomorphic deformation field conditioned on the image and the current location of the template. This stage improves the accuracy of the prediction by capturing finer details of the target geometry. We describe a novel loss function derived from the kinematics of motion of continuous bodies that penalizes undesirable phenomenon such as surface interpenetration resulting in anatomically accurate, physically realistic, simulation ready meshes. The proposed framework is validated against a large held-out test dataset and compared with prior state-of-the-art along a variety of accuracy and quality metrics.

To *my family*

Contents

Contents **ii**

List of Figures **iv**

List of Tables **xii**

1 Introduction **1**

2 Optimizing the topology of triangular and quadrilateral meshes using deep reinforcement learning **3**
- 2.1 Introduction . . . 3
- 2.2 Problem Statement . . . 6
- 2.3 Mesh Representation and Operations . . . 9
- 2.4 Formulation as a Deep Reinforcement Learning Problem . . . 12
- 2.5 Results . . . 19
- 2.6 Conclusion . . . 24
- 2.7 Future Work . . . 25

3 Optimizing the topology and geometry of general polygonal meshes using deep reinforcement learning **32**
- 3.1 Introduction . . . 32
- 3.2 Problem Statement . . . 35
- 3.3 Mesh Data Structures . . . 43
- 3.4 Formulation as a Deep Reinforcement Learning Problem . . . 45
- 3.5 Training the Agent in Self-play Reinforcement Learning . . . 54
- 3.6 Evaluation . . . 66
- 3.7 Conclusion . . . 70
- 3.8 Software . . . 73

4 LinFlo-Net: A two-stage deep learning method to generate simulation ready meshes of the heart **74**
4.1 Introduction . . . 74
4.2 Related Work . . . 76
4.3 Methods . . . 78
4.4 Results . . . 87
4.5 Discussion . . . 96
4.6 Software . . . 99

Bibliography **100**

List of Figures

2.1 Configuration (a) and (b) are related by an edge flip. Configuration (c) can be produced by splitting the interior edge in either (a) or (b). Collapsing the edge between vertex 3-6 in (d) produces (e). 8
2.2 Configuration (a) and (c) can be obtained from (b) via an edge flip. Configuration (e) is obtained from (d) via a vertex split, and the operation can be reversed via an element collapse. 9
2.3 Performing a global split on the edge between vertices 2 and 5 in the initial mesh (b) produces the mesh in (a). Alternatively, the sequence of edges between vertices $4-5-6$ in the initial mesh (b) can be deleted by merging the neighboring elements, resulting in configuration (c). 10
2.4 Representing two triangular elements using the DCEL. The half-edges in each element are shown as red arrows. Each half-edge contains a pointer to the counter-clockwise *next* half-edge in the same element e.g. in triangle T_2, half-edge 2's *next* pointer points to half-edge 3. Half-edges in the interior of the mesh have a *twin* pointer to the half-edge in the adjacent element e.g. the *twin* of half-edge 1 in triangle T_2 is half-edge 2 in triangle T_1. We additionally associate each half-edge with a unique vertex in the element. For triangles we associate the vertex opposite a given half-edge e.g. half-edge 1 in triangle T_2 is associated with vertex 4. For quadrilaterals we associate the vertex at the origin of the half-edge. . . . 11
2.5 Repeated application of the convolution operation produces an increasing field of view around every half-edge. For triangles, we associate every half-edge with the vertex opposite it in the same triangle (fig. a). A `cycle` operation gathers information from the remaining vertices in the element (fig. b). Repeated application of `twin` and `cycle` produces the ordered list of vertices in fig. (c) and (d). 16

2.6 Illustrating the action selection process on meshes. Half-edge features consisting of the degree and irregularity of the associated vertex are first projected to an embedding space. Convolution on the DCEL data-structure is performed on these embeddings. Local templates are constructed around all half-edges to obtain a local measure of mesh irregularity. Action selection is restricted to the local template with the highest measure of irregularity. The final feature matrix is flattened and passed through a softmax layer to obtain action probabilities (i.e. a policy). This distribution is sampled to determine the action to take. 17
2.7 (a) Performance of the triangle mesh agent over the training history. Solid line represents the average normalized return over 100 meshes evaluated periodically during training. Shaded region represents the 1-standard deviation envelope. (b) Performance of the trained agent over 100 rollouts. The agent incrementally improves mesh quality up to a certain number of steps. Notice that returns do not increase monotonically, indicating that a greedy strategy may not be effective for this problem. 20
2.8 Example rollout of the triangular mesh agent on a 20-sided polygon. Irregular vertices are marked in color, with the current score and optimum score shown at the top right for each figure. (a) is the initial Delaunay refinement mesh (b) is at an intermediate stage and (c) is the final mesh after 27 operations. 21
2.9 Example rollout of the triangular mesh agent on a 40-sided random polygonal shape. (a) is the initial mesh, (b) is an intermediate state, and (c) is the mesh with the lowest score during policy rollout. Notice that the agent is able to improve the irregularity score of the mesh. However, the final mesh contains irregular nodes that are separated by several mesh elements. These irregularities require complex sequences of moves to bring them together and regularize them. Further, the agent has a limited field of view controlled by the size of the local template. Irregularities outside the local template will not be seen by the agent. 22

2.10 (a) Performance of the quadrilateral mesh agent over the training history. Solid line represents the average normalized returns evaluated over 100 meshes. Shaded region represents the 1-standard deviation envelope. The curve demonstrates that the agent is able to achieve good performance quite rapidly, and the learning remains stable over many training iterations. (b) Evaluating the trained model over multiple rollouts. Solid line represents the average performance of 100 rollouts. The graph demonstrates that increasing the maximum number of operations available to the agent has a big impact on performance initially, but only up to a certain point. Returns do not increase monotonically, highlighting that a greedy strategy may not be effective in this setting. 23

2.11 Example rollout for a 10-sided polygon. Irregular vertices are marked in color. The mesh score and optimal score are shown at the top right for each figure. (a) is the initial mesh after Delaunay triangulation and catmull-clark splits, (b) is at an intermediate stage, and (c) is the final mesh after 18 operations. 25

2.12 The same agent as before is able to optimize a 20-sided polygonal shape using 40 operations. (a) initial mesh, (b) intermediate mesh, (c) final mesh produced by the agent, (d) degenerate vertices on the boundary can be post-processed using an operation similar to the global split. Note that meshes (c) and (d) have the same score as measured by eq. 2.1 thus our agent does not prefer one over the other. If the application demands that there be no degenerate vertices, these irregularities can be eliminated through a final post-processing step. 26

2.13 The triangular mesh agent demonstrates zero-shot transfer on geometries that were not seen during training, including geometries with re-entrant corners like the star shape and the single and double-notch domains. First column is the initial mesh, second column is the intermediate mesh as the agent optimizes connectivity, and the third column is the final mesh after optimization. Since our model is based purely on connectivity, we can directly transfer the model onto geometries with holes even though such geometries were never seen during training. Notice that in several of the examples including the L-domain, single-notch, and the square hole in the circle, the model achieves the optimal score and the remaining irregularities cannot be eliminated as they are intrinsic to the geometry. . 28

2.14 The quadrilateral mesh agent demonstrates zero-shot transfer on geometries that were not seen during training, including geometries with reentrant corners like the star shape and the single and double-notch domains. First column is the initial mesh, second column is the intermediate mesh as the agent optimizes connectivity, and the third column is the final mesh after optimization. Since our model is based purely on connectivity, we can directly transfer the model onto geometries with holes even though such geometries were never seen during training. We are particularly interested in coarse meshes representing block decompositions of more complex shapes. The cleanup operation is particularly useful in achieving coarse meshes. This is most evident in the single notch and double notch example in row 3 and 4. Notice that the star-shaped domain and the circular mesh with a square hole contain intrinsic irregularity that cannot be improved upon with our prescribed operations and heuristic. 30

3.1 Example geometric representations. Geometries are represented as sets of closed faces. A face is defined as a closed loop of vertices in counter-clockwise order. (a) a geometry with a single face, (b) a geometry with multiple faces each of which has a different polygonal degree, (c) a geometry with a hole. 36

3.2 Illustration of topological edit operations: (a) initial polygon which transforms to (b) after an edge insertion between vertices 0-3. A new vertex is inserted on this edge to produce (c) from which we produce (d) with an edge insert between vertices 6-4. The edge between 6-3 is deleted to give (e). Finally, vertex 6 is deleted to produce (f). 37

3.3 Sequence of mesh editing operations and the associated score changes for a simple L-shaped input geometry with $f^* = 4$. 40

3.4 A primal mesh (black) and its corresponding dual (red). The dual is constructed by transforming faces in the primal mesh to vertices in the dual mesh, and vertices in the primal mesh to faces in the dual mesh. Edges are added between vertices in the dual mesh for every pair of adjacent faces in the primal mesh. 41

3.5 An edge is inserted between two vertices that are k steps apart in a face whose degree is initially f. This results in two new faces whose degrees are respectively $k+1$ and $f-k+1$. Further, the vertices labelled 0 and k see their respective degrees increase by one. The degrees of the remaining faces and vertices in the mesh do not change. 42

3.6 Polygonal meshes can be represented by the doubly-connected edge list (DCEL). Edges are represented as a pair of oriented half-edges shown as red arrows. (Boundary half-edges are shown in gray.) Half-edges are associated to their cyclic *next* and *previous* half-edges e.g. half-edge 2 is *next* of half-edge 1 and *previous* of half-edge 3. Half-edges are associated to the vertices at their *source* and *target* e.g. vertex 3 is the *source* of half-edge 5 and the *target* of half-edge 4. Half-edges are associated to the unique face that they belong to e.g. half-edges 4-5-6-7 belong to face 1. . 43
3.7 Illustrating the template construction centered around half-edge 0. With $K = 10$, the template would consist of the half-edges $[0, 1, 5, 8, 2, 4, 9, 7, 3, 10]$. 48
3.8 Relevant topological and geometric features associated with each half-edge. Topological features include the vertex degree, vertex desired degree, and face degree. Geometric features include length and angle. . . . 50
3.9 Illustration of the DCEL convolution operation. The input feature matrix is first augmented by appending two learnable vectors at the end. The augmented matrix $\bar{F}$ undergoes the convolution operation described in eq. 3.10 to produce the output. LN is layer normalization and σ is a non-linear activation function. Note that the index vectors N, P, T are only used to facilitate the convolution. LN and σ are only applied to the feature matrix. 51
3.10 Illustration of a DCEL residual block which consists of two DCEL convolution operations followed by a skip connection. 52
3.11 Overall architecture of our neural network. The half-edge feature matrix F is first projected to F_0 in a d dimensional embedding space. These features are processed by L DCEL residual blocks. The final feature matrix F_L is shared by the actor head and critic head which consist of perceptrons with a single hidden layer. The policy head masks invalid actions with the mask matrix M to get the action logits which represent the log probability of action selection. The value head performs an average reduction operation to get a single number representing the critics estimate of the state value. 53
3.12 Scaling of initial scores with respect to polygon degree for randomly generated polygons with different desired face degrees. (a) Initial vertex score s_v^0 (b) Initial angle score s_θ^0. Scores measured over 50 trials per polygon degree. Solid line represents the mean and shaded region the 1-standard deviation envelope. Plots demonstrate that the relationship is effectively linear. For the initial boundary, the only difference between s_v^0 and s_θ^0 is the rounding operator in eq. 3.6. 55

3.13 Illustrative examples of randomly generated initial polygonal shapes. Top row is for $f^* = 3$ and bottom row is for $f^* = 4$. Vertices with non-zero vertex irregularity are marked. 56
3.14 Training trajectories of 100 optimization trials for (a) $f^* = 4$ and (b) $f^* = 3$. Trajectories are generated by periodically measuring the average return during training. Returns are averaged across 200 randomly generated environments for each evaluation step. The training trajectories suggest that most models have converged upon their optimal values. 60
3.15 Optimization history for (a) $f^* = 4$ and (b) $f^* = 3$. Each point represents the final objective value of a particular trial. The red line represents the best objective score seen so far. 61
3.16 Hyperparameter importances for (a) $f^* = 4$ and (b) $f^* = 3$ as measured by the functional Analysis of Variance (fANOVA) method [32]. The results indicate that a few parameters have an outsized impact on the final objective function. Further, we see that the relative importance of hyperaparameters depends on the choice of f^*. 62
3.17 Optimization landscape of the three most important hyperparameters for $f^* = 4$ – namely entropy coefficient c_e, GAE coefficient λ, PPO clip parameter ϵ. Note that c_e is plotted in the log-domain while λ and ϵ are plotted in the linear domain. Scatter points represent trial points. We highlight the highly non-linear optimization landscape and the non-uniform sampling strategy of the TPE algorithm that aims to explore regions with high objective value more than regions with low objective value. 63
3.18 Optimization landscape of the three most important hyperparameters for $f^* = 3$ – namely entropy coefficient c_e, number of DCEL residual blocks L, and embedding dimension d. Note that c_e is plotted in the log-domain while L and d take on discrete values. Scatter points represent trial points. We highlight the highly non-linear optimization landscape and the non-uniform sampling strategy of the TPE algorithm that aims to explore regions with high objective value more than regions with low objective value. 64
3.19 Training loss history of best-performing agent for $f^* = 4$. Agent was trained for 10^7 time-steps. 65
3.20 Training loss history of best-performing agent for $f^* = 3$. Agent was trained for 10^7 time-steps. 66

3.21 Model performance as a function of training history for $f^* = 4$. (a) Ratio of actions whose gradients are clipped according to the PPO clipped objective function. Initially, the policy undergoes rapid changes resulting in a higher fraction of actions experience clipping. As training stabilizes, a fewer fraction of actions experience gradient clipping. (b) The explained variance of the value function head is close to 1 suggesting that the model is successfully capturing the underlying environment dynamics. (c) The average returns rapidly increases and then slowly approaches a value of around 0.7 at 10^7 steps. Further training may improve the performance. 67

3.22 Model performance as a function of training history for $f^* = 3$. (a) Ratio of actions whose gradients are clipped according to the PPO clipped objective function. Initially, the policy undergoes rapid changes resulting in a higher fraction of actions experience clipping. As training stabilizes, a fewer fraction of actions experience gradient clipping. (b) The explained variance of the value function head is close to 1 suggesting that the model is successfully capturing the underlying environment dynamics. (c) The average returns rapidly increases and then slowly approaches a value of around 0.78 at 10^7 steps. 68

3.23 Average best score vs polygon degree for quadrilateral and triangular element types. The model was trained on random polygons of degree between 6-20 and evaluated on geometries with degree 6-40. For a given instance of an environment, the policy is rolled out 10 times from the same initial state and the best score is selected. We compute the average (solid line) and standard deviation (shaded region) over 10 random instances for a given polygon degree and repeat for various values of polygon degree. We observe that the model demonstrates generalization capability beyond the training regime. The quadrilateral model demonstrates some deterioration in model performance with increasing polygon degree. The drop in performance is lesser for the triangular model. 69

3.24 Example rollouts of the trained $f^* = 4$ policy for randomly generated polygonal shapes of different size. The final geometry was selected by running the policy 10 times from each initial state and picking the mesh with the smallest face score with ties broken by meshes with smallest total score. Note that in all the cases, the model consistently generates meshes with optimal face score. Irregular vertices are marked with their corresponding irregularity. 71

3.25 Example rollouts of the trained $f^* = 3$ policy for randomly generated polygonal shapes of different size. The final geometry was selected by running the policy 10 times from each initial state and picking the mesh with the smallest face score with ties broken by meshes with smallest total score. Note that in all the cases, the model consistently generates meshes with optimal face score. Irregular vertices are marked with their corresponding irregularity. 72

4.1 Illustrative examples of CT (top row) and MRI (bottom row) images of the cardiac region. The segmentation of some cardiac structures of interest are overlaid on the figures on the right. Since the myocardium is a thick muscular structure, it is clearly visible in the image. However, the tissue thickness of other structures like the aorta are not visible in these images, and only the blood pool within these structures is visible. 76
4.2 Two samples from our dataset that demonstrate the variation in scale across samples. Black wireframe is the template mesh, and red surface is the ground-truth mesh. 79
4.3 Workflow describing the training pipeline for the linear transformation module 80
4.4 Workflow illustrating the training pipeline for the flow deformation module 82
4.5 An illustration of samples generated by the data-augmentation process from a given input image and associated ground-truth segmentation. We generate these samples by applying small perturbations including random scaling, translation, rotation, shear, and local b-spline deformations to the input image and segmentation. Meshes are produced from the generated segmentations via the marching cubes algorithm. 86
4.6 Comparison of dice score on CT test data for different model choices . . 91
4.7 Comparison of dice score on MR test data for different model choices . . 91
4.8 Template mesh of the left-ventricle myocardium with thickness added to inlets and outlets 94
4.9 LT-FL (right) collapses the inlet/outlet surface. LT-FL-V (left) preserves the added thickness. 95
4.10 LT-FL-V (left) produces SIF primarily due to element collapse. LT-FL produces SIF due to surface interpenetration and element collapse. SIFs are highlighted in both images. 96
4.11 SIF due to element collapse (left) can be easily fixed by standard surface remeshing algorithms. Here we demonstrate the result of using the isotropic remeshing available in MeshLab. 97
4.12 Volume loss is effective at regularizing unphysical deformations 98

List of Tables

2.1 Hyperparameter settings used to train the agent 19
2.2 Evaluating the triangle mesh agent on various sized random polygons. The agent was trained purely on 10 - 30 sided polygons but is able to generalize to other polygon sizes with minor deterioration in performance. The agent was evaluated by picking the best of 10 rollouts per geometry, with the statistics computed over 100 randomly generated shapes. The results demonstrate the effectiveness of using a fixed-sized local template which enables better generalization to different mesh sizes. 21
2.3 Evaluating the quadrilateral mesh agent on various sized random polygons. The agent was trained purely on 10 - 20 sided polygons, but is able to generalize to larger polygonal shapes with minor deterioration in performance. The agent was evaluated by picking the best of 10 rollouts per geometry, with the statistics computed over 100 randomly generated polygonal shapes. Using a fixed-sized local template enables stronger generalization to different sized meshes. 24

3.1 Illustrating the template construction process for the example geometry shown in figure 3.7. Note that "*next* Index" refers to the *index* in the template list of the cyclic *next* half-edge and similarly for *previous* and *twin*. We use the special index -1 to denote a half-edge that exists in the geometry but is outside the template and the index -2 to denote a half-edge that is outside the geometry (i.e. a boundary half-edge). 49
3.2 Parameters used to specify the random polygon environment. 54
3.3 List of parameters and their corresponding prior range used in hyperparameter optimization. We use the same design space for all our experiments. 58
3.4 Results of hyperparameter optimization for $f^* = 4$ and $f^* = 3$. Optimal values were obtained by running hyperparameter optimization over 100 trials. Each run consisted of training the agent for 1M time-steps. 59
3.5 Hyperparameters that were taken constant for all trials. 59

4.1 Hyperparameter settings used to train the model 87
4.2 Comparing mean (standard deviation) of accuracy metrics on CT test data. ↑ (↓) indicates higher (lower) value is better. 89
4.3 Comparing mean (standard deviation) of accuracy metrics on MR test data. ↑ (↓) indicates higher (lower) value is better. 89
4.4 Comparing mean (max) percentage self-intersecting faces on CT test dataset 89
4.5 Comparing mean (max) percentage self-intersecting faces on MR test dataset 90
4.6 Ablation study – comparing mean (standard deviation) of accuracy metrics on CT test data. ↑ (↓) indicates higher (lower) value is better. 92
4.7 Ablation study – comparing mean (standard deviation) of accuracy metrics on MR test data. ↑ (↓) indicates higher (lower) value is better. . . . 92
4.8 Ablation study – comparing mean (max) percentage self-intersecting faces on CT test data 93
4.9 Ablation study – comparing mean (max) percentage self-intersecting faces on MR test data 93
4.10 LV-template mesh quality on CT and MR test data 94

Acknowledgments

First and foremost, I want to thank my parents Nirmala and Narayanan for being an endless source of support and encouragement. They fostered my interest in science and engineering from a young age and encouraged me to remain curious and constantly seek growth. My brother Aditya has been an inspiration to me for as long as I can remember. My journeys have been so much easier because he paved the path ahead of me. I want to thank my sister-in-law Supriya for her wisdom and sound advice.

I am grateful to my wife, Aditi, for her unshakable confidence in me. She believed for the both of us when I could not, and that's what powered me through difficult times. Academic manuscripts may seem devoid of emotion, but behind every page is a tapestry of emotions, fears, and insecurities. I am indebted to her for putting up with all of it with patience and kindness.

I want to thank my Co-chairs Professor Per-Olof Persson and Professor Shawn Shadden. I was like a rudderless ship lost at sea in the middle of my PhD. Their mentorship and support guided me back to safe harbour. I am thankful to Professor Panayiotis Papadopoulos and Professor Tarek Zohdi for their guidance and feedback. A special thank you to my collaborators Fanwei Kong, Numi Sveinsson, and Lewis Pan for stimulating conversations – many of which materialized into the pages here.

I am grateful for financial support from the Office of Advanced Scientific Computing Research, U.S. Department of Energy (contract No. DE-AC02-05CH11231) and the National Science Foundation (Award 1663671).

Chapter 1

Introduction

Mesh generation is the process of decomposing a complex shape into a collection of geometric primitives e.g. triangles and quadrilaterals in 2D and tetrahedra and hexahedra in 3D. Meshes are widely used in Computer-Aided Design (CAD) and physics based simulation. While there has been tremendous progress in the past several decades, mesh generation still remains a significant bottleneck in many simulation workflows. Indeed, the NASA Computational Fluid Dynamics (CFD) Vision 2030 [64] study identified mesh generation and adaptivity as "significant bottlenecks in the CFD workflow".

In almost all practical scenarios, engineers are interested in running simulation on complex geometries that require so-called "unstructured" meshing. This is in general a challenging problem that is hard to automate, often requiring significant manual intervention. Further, existing algorithms are often based on human-derived heuristics that may be incomplete. Rule-based algorithms quickly become very complicated because of the highly unstructured nature of meshes.

Machine learning (ML) is a technique that has become the *de facto* solution approach for other unstructured problems in computer science such as image classification and voice recognition wherein it is challenging to develop rule-based algorithms. ML frames a given task as an optimization problem and leverages large amounts of data to automatically "learn" useful features through the optimization process.

This dissertation aims to explore the application of machine learning techniques to select problems from mesh generation. Meshes have two fundamental attributes – geometry and topology. Geometry deals with the position and shape of objects in space. For a mesh, this is determined by the position (i.e. coordinates) of its vertices. Topology is concerned with how things are connected to each other. In the case of a mesh, this is determined by the connectivity of vertices, edges, faces, and volumes. Chapter 2 presents a novel ML technique to optimize the topology of a given 2D mesh. Chapter 3 extends the prior method to include certain geometric considerations. Chapter 4 presents an ML technique to optimize the geometry of a given 3D surface mesh with a specific focus on generating patient-specific models of the human heart from clinical imaging.

The intersection of machine learning and mesh generation is an area under active research with myriad techniques and applications. The chapters in this dissertation are designed to be self-contained with each chapter including a discussion of relevant prior research work and the similarities / differences with the techniques presented herein. We hope this dissertation stands as a useful contribution and sparks further investigation into this exciting field.

Chapter 2

Optimizing the topology of triangular and quadrilateral meshes using deep reinforcement learning

We present a learning based framework for mesh quality improvement on unstructured triangular and quadrilateral meshes. Our model learns to improve mesh quality according to a prescribed objective function purely via self-play reinforcement learning with no prior heuristics. The actions performed on the mesh are standard local and global element operations. The goal is to minimize the deviation of the node degrees from their ideal values, which in the case of interior vertices leads to a minimization of irregular nodes.

2.1 Introduction

Mesh generation is a crucial part of many applications, including the numerical simulation of partial differential equations as well as computer animation and visualization. While it can be discussed exactly what makes a mesh appropriate for a given situation, it is widely accepted that fewer number of irregular nodes lead to better quality meshes. Therefore, many mesh generation and mesh improvement methods have been proposed that aim to maximize the regularity of the mesh, in particular in the case of quadrilateral elements.

For triangular meshes, some of the most popular algorithms are the Delaunay refinement method [61] and the advancing front method [52]. The resulting meshes might be improved by local operations or smoothing, although typically based on element qualities rather than the regularity of the connectivities. Some quadrilateral mesh generators are also based on a direct approach, such as the paving method [8], but most use an indirect approach of creating quadrilateral elements from a triangular mesh. These methods include the popular Q-Morph method [46], element matching methods such as the Blossom-Quad method [57], and so-called regularization or mesh simplification methods which improve an initial mesh using various mesh modification techniques [17, 68, 3, 9].

Although many of these mesh modification methods produce impressive results, we note that the algorithms for how they apply the various mesh operations are usually highly heuristic in nature [23, 3]. This is expected, since finding an optimal strategy is a complex discrete optimization problem. Indeed, Canann et al. [12] highlight the need to "recognize and process patterns of irregularities that occur over larger groups of elements" in order to achieve meshes with desirable connectivity and they demonstrate the efficacy of complex mesh edits by composing sequences of multiple local operations. Developing such heuristics is challenging and laborious, and it is difficult to adequately explore the state space and action space. Identifying optimal sequences of mesh editing operations that improve mesh quality is not trivial and is highly dependent on the type of mesh being considered. For instance, the heuristics that optimize triangular meshes are different from the heuristics that optimize quadrilateral meshes. Further, traditional optimization methods such as mixed integer programming are computationally expensive and the computational cost grows exponentially with problem size making it challenging to adapt these methods to meshes of different size.

To address the above challenges, we explore deep reinforcement learning as a solution approach for this optimization problem. A major advantage of such an approach is that optimal sequences of actions can be discovered by the reinforcement learning agent purely by interacting with a mesh environment through self-play. This circumvents the requirement for human crafted heuristics. The agent can adapt to different mesh types by training it on an appropriate mesh environment. This provides a unified approach that can be applied to different mesh types. We formulate our problem in the language of reinforcement learning (RL) [66] wherein actions available to the agent are local mesh edit operations and the rewards are the improvement of mesh regularity as measured by a prescribed objective function. By converting our optimization problem into a sequential decision process consisting of local mesh editing operations, we avoid the exponential cost of global solution strategies like mixed integer programming.

In this work, we consider the case of planar straight-sided polygonal geometries. However, since our method is based purely on mesh connectivity, it may be applied to geometries with curved boundaries as well so long as the regularity of vertices on these boundaries is specified. We generate a coarse initial triangular mesh using the Delaunay refinement algorithm. In the case of quadrilateral meshes, we perform Catmull-Clark splits of the triangles, and we also introduce global mesh operations. One of these is the clean-up, which aims to reduce the total number of elements which is suitable for generation of block decompositions.

A key component of our framework is the employment of the half-edge data structure, which in particular allows us to define a convolutional operation on unstructured meshes. A neural network is trained to produce a probability distribution for the various actions on local neighborhoods of the mesh, i.e., a policy. The policy is sampled to determine the next operation to perform. A powerful property of our method is that it generalizes to both triangular and quadrilateral meshes with minimal modifications to account for the different actions available on these meshes. We limit action selection to local mesh neighborhoods, allowing the learned policy to generalize well to a variety of mesh types and sizes that were not present in the training data. We demonstrate our methods on several polygonal shapes, where we consistently obtain meshes with optimal regularity. Extension of this method to arbitrary polygonal elements is reserved for future work.

Machine learning has been applied to numerous mesh generation problems before. Pointer networks [73] have been used to generate convex hulls and Delaunay triangulations. Deep RL has been used to learn quadrilateral element extraction rules for mesh generation [48, 49]. RL has also been employed to learn adaptive mesh refinement strategies [78, 65]. In [20], RL was used to perform block decomposition of planar, straight-sided, axis aligned shapes using axis aligned cuts.

Our work differs from prior work in several key ways. Our objective function is purely based on the connectivity of the mesh and our framework aims to minimize the number of irregular vertices. We consider local topological edit operations as our action space. Our novel convolution operation on the half-edge data-structure provides a powerful, parameterized way of constructing state representations that encode neighborhood connectivity relationships. We employ a local neighborhood selection technique that allows us to generalize to different mesh sizes. These key features enable our method to work on both triangular and quadrilateral meshes of various sizes.

The half-edge data-structure is able to represent arbitrary polygonal shapes in 2D. Thus our state representation method naturally extends to all such polygonal shapes. Our reinforcement learning framework can be applied to these shapes so long as an appropriate action space is defined. We hypothesize that the technique can be extended to 3-dimensions by leveraging the equivalent of the half-edge data-structure in higher dimensions [21, 24]. Prior work has explored the action space in 3D e.g.

tetrahedral [62, 34, 26] and hexahedral meshes [39, 69].

2.2 Problem Statement

In the present work we are interested in optimizing the connectivity of triangular and quadrilateral meshes. The overall objective is to produce meshes where all the vertices have a specific number of incident edges. We refer to this as the *desired degree* of a vertex. A vertex whose degree is the same as the desired degree is called *regular*. A vertex whose degree is different from the desired degree is called *irregular*, with the difference between the degree and the desired degree being a measure of the *irregularity* of the vertex. Our framework allows the user to specify the desired degree on all vertices. The user is allowed to specify the desired degree of any newly introduced vertex.

While there exist robust algorithms for triangular and quadrilateral meshing such as Delaunay triangulation and paving, these algorithms are not designed to produce meshes with a specific connectivity structure. A common approach is to use these algorithms as a starting point and improve the connectivity of the mesh through various topological mesh editing operations [22]. We adopt this approach and frame our problem as a Markov Decision Process.

Objective function

Consider a mesh with N_v vertices. Let vertex i have degree d_i and desired degree d_i^*. Then its irregularity is $\Delta_i = d_i - d_i^*$. We compute a global score s as the L1 norm of Δ, which is a measure of the total irregularity in the mesh.

$$s = \sum_{i=1}^{N_v} |\Delta_i| \tag{2.1}$$

Clearly, a mesh with all regular vertices will have a score $s = 0$.

Heuristics to determine desired degree

Our heuristic for triangular (quadrilateral) meshes is based on achieving an interior angle of 60° (90°) in all elements. The desired degree of any vertex in the interior is 6 (4). The desired degree of a boundary vertex is chosen such that the average included angle in all elements incident on that boundary vertex is approximately 60° (90°) . The desired degree according to this heuristic can be expressed as,

$$d^* = \begin{cases} 360/\alpha & \text{interior vertex} \\ \max\left(\lfloor \theta/\alpha \rceil + 1, 2\right) & \text{boundary vertex} \end{cases} \tag{2.2}$$

where $\lfloor \cdot \rceil$ is the round to nearest integer operator, θ is the angle of the boundary at the vertex in question, and α is $60°$ ($90°$) for triangles (quadrilaterals). We observed that rounding to the nearest integer resulted in better performing models than using d^* as a continuous value on the boundary. According to this heuristic, the desired degree of a new vertex introduced on the boundary is set to 4 (3) since we assume that the edge on which the new vertex is introduced is a straight edge.

Topological operations on meshes

We define the following local operations on triangular meshes. See figure 2.1 for an illustration.

- **Edge Flip:** An interior edge in a triangular mesh can be deleted and the resultant quadrilateral can be re-triangulated across its other diagonal. This can be seen as "flipping" an edge between two possible states.
- **Edge Split:** Any edge in a triangular mesh can be split by inserting a new vertex on the edge and connecting it to the opposite vertices in the adjacent triangles.
- **Edge Collapse:** An interior edge in a triangular mesh can be collapsed resulting in the deletion of the two triangles associated with this edge.

Similarly, we define the following local operations on quadrilateral meshes. See figure 2.2 for an illustration.

- **Edge Flip:** An interior edge in a quadrilateral mesh can be deleted, and the resultant hexagon can be quad-meshed in two new ways. This can be seen as "flipping" an edge clockwise or counter-clockwise.
- **Vertex Split:** A vertex in a quad mesh can be split along an interior edge incident at that vertex. This results in the insertion of a new vertex and a new element into the mesh.
- **Element Collapse:** A quadrilateral element can be collapsed along either diagonal by merging the two opposite vertices. The collapse operation can be seen as the inverse of the split operation defined above.

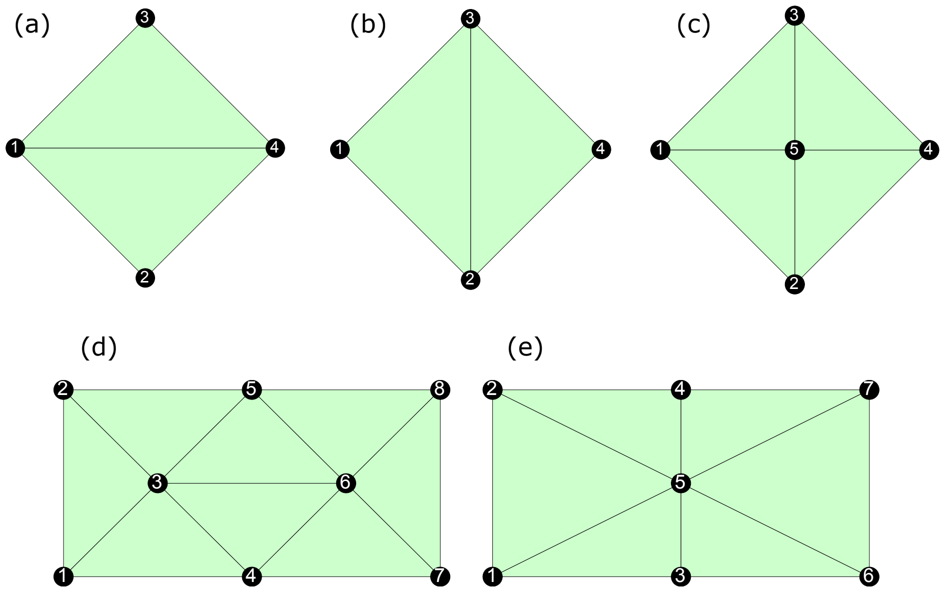


Figure 2.1: Configuration (a) and (b) are related by an edge flip. Configuration (c) can be produced by splitting the interior edge in either (a) or (b). Collapsing the edge between vertex 3-6 in (d) produces (e).

For quadrilateral meshes we also define the following global mesh editing operations. They are global in the sense that they can affect the topology of the mesh far away from where they are applied. See figure 2.3 for an illustration.

- **Global Split:** This operation splits an edge by inserting a quadrilateral element and introducing vertices on the edges in the two adjacent quadrilateral elements. The introduced vertices are hanging vertices – therefore we recover an all-quadrilateral mesh by propagating edges from the hanging vertices and sequentially splitting elements until the split terminates on a boundary.
- **Global Cleanup:** In some situations, global lines – which represent a sequence of edges – can be deleted by merging adjacent elements. The global line either terminates on the boundaries of the mesh or forms a closed loop. We currently handle the situation where the global line terminates on the boundaries. (For meshes representing closed surfaces it would be important to consider the case of closed loops.) This operation results in the deletion of a sequence of vertices and elements. Vertices are distinguished into geometric and non-geometric

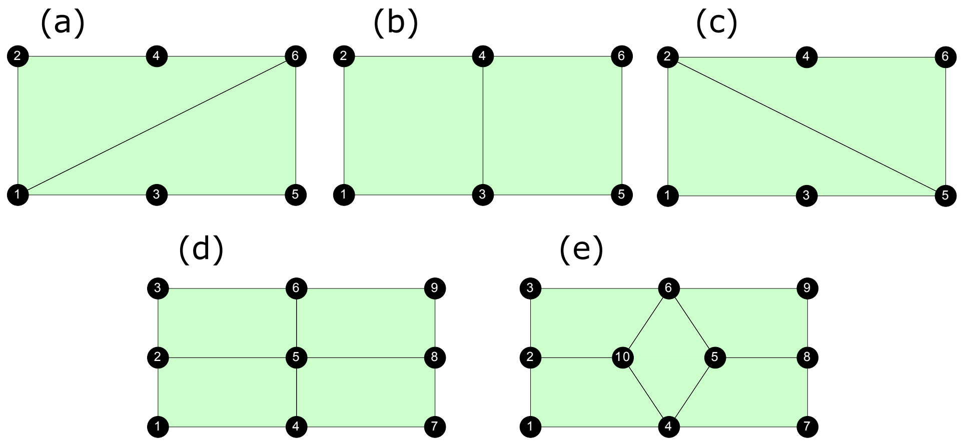


Figure 2.2: Configuration (a) and (c) can be obtained from (b) via an edge flip. Configuration (e) is obtained from (d) via a vertex split, and the operation can be reversed via an element collapse.

vertices. Geometric vertices are those vertices which are integral in defining the geometry – these vertices cannot be deleted. The conditions under which we can perform this cleanup operation are (a) the end-points are on the boundary, are non-geometric, and have degree 3, and (b) all interior vertices are non-geometric and have degree 4. The cleanup operation is a powerful operation since it simplifies the problem and brings irregular vertices closer together. This strategy is particularly relevant for block decomposition of polygonal shapes.

2.3 Mesh Representation and Operations

The half-edge data structure

We employ the doubly-connected edge list (DCEL), also known as the half-edge data-structure, to represent our meshes. The advantage of the DCEL is that (a) it enables efficient implementations of the mesh editing operations described in 2.2, and (b) we utilize fundamental DCEL operations to represent the local topology in a given mesh region which is important to determine the appropriate action to be applied. The DCEL can be used to represent any planar, manifold mesh and as such

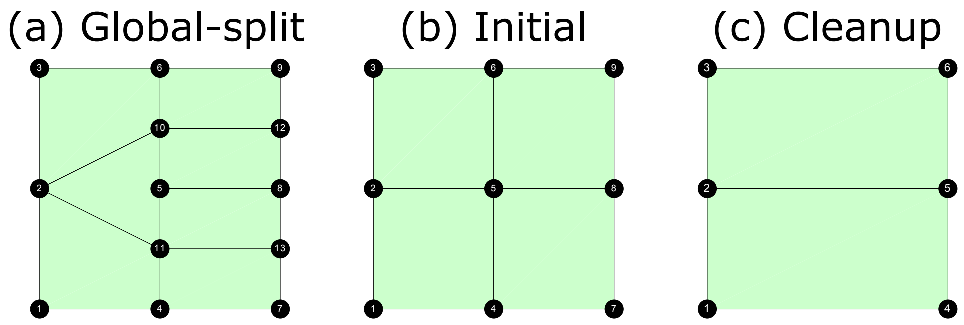


Figure 2.3: Performing a global split on the edge between vertices 2 and 5 in the initial mesh (b) produces the mesh in (a). Alternatively, the sequence of edges between vertices 4 – 5 – 6 in the initial mesh (b) can be deleted by merging the neighboring elements, resulting in configuration (c).

allows our method to work on all such meshes. Extensions to the DCEL have been developed for non-manifold meshes and 3D volumetric meshes [21, 24].

Briefly, the DCEL exploits the fact that each mesh edge is shared by exactly two mesh elements (except on the boundary). The DCEL represents each mesh edge as a pair of oriented half-edges pointing in opposite directions. Each half-edge contains a pointer to the counter-clockwise *next* half edge in the same element, and a pointer to the *twin* half-edge in the adjacent element. Each element contains a pointer to one of its half-edges (chosen arbitrarily) which induces an ordering on the half-edges in an element. Elements can be ordered by their global index in the mesh – this induces a global ordering on half-edges in the mesh. Each half-edge may be associated with a unique vertex. For triangles, we associate each half-edge with its opposite vertex. For quadrilaterals, we associate each half-edge with the vertex at its origin. The exact choice of the association does not matter as long as it is consistent. See fig. 2.4 for an illustration. Further details about the DCEL can be found in a standard resource on computational geometry, for example [44].

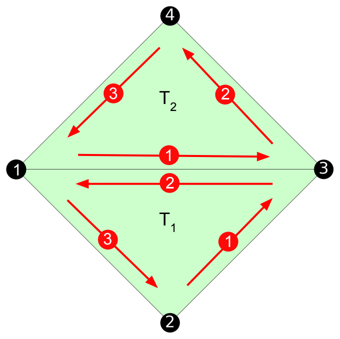


Figure 2.4: Representing two triangular elements using the DCEL. The half-edges in each element are shown as red arrows. Each half-edge contains a pointer to the counter-clockwise *next* half-edge in the same element e.g. in triangle T_2, half-edge 2's *next* pointer points to half-edge 3. Half-edges in the interior of the mesh have a *twin* pointer to the half-edge in the adjacent element e.g. the *twin* of half-edge 1 in triangle T_2 is half-edge 2 in triangle T_1. We additionally associate each half-edge with a unique vertex in the element. For triangles we associate the vertex opposite a given half-edge e.g. half-edge 1 in triangle T_2 is associated with vertex 4. For quadrilaterals we associate the vertex at the origin of the half-edge.

Algorithmic complexity and parametrization of mesh editing operations

All of the local editing operations defined in 2.2 for triangles and quadrilaterals can be executed using the DCEL in constant time (assuming an upper bound on the maximum degree of a vertex). This is a powerful advantage offered by the DCEL compared to other mesh representations. For instance, when flipping a particular half-edge it is important to know which are the two neighboring elements across that edge – this is readily available in the DCEL.

The global operations defined for quadrilateral meshes in 2.2 requires connectivity editing operations that can propagate through several elements of the mesh before terminating. The algorithm for these operations scales linearly with the size of the mesh. In particular we disallow situations where global splits may result in the formation of loops or that do not terminate in a fixed number of iterations proportional to the size of the mesh. For the cleanup, we observe that every half-edge either lies on a cleanup path or does not. Performing a cleanup on a path does not affect the ability to cleanup other paths. Therefore all cleanups possible in a mesh can be performed by visiting every half-edge exactly once.

Our framework optimizes a policy to perform sequences of mesh editing operations to achieve a given objective. All operations other than the global-cleanup are valid operations that can be learned by the policy. Whenever a global-cleanup is valid, it is always performed. We choose to do this because the cleanup simplifies the problem size and brings irregular vertices together making it easier to improve the connectivity of the mesh. A cleanup only deletes regular vertices according to our heuristic and never introduces any new irregular vertices in the mesh. Further, the cleanup is very useful in performing block decompositions of polygons.

We parametrize all the mesh editing operations in terms of half-edges. In a given mesh, specifying a particular half-edge and a particular type of edit determines an operation on the mesh. We have 3 operations per half-edge in the case of triangular meshes – flip, split, and collapse. Further, we have 5 operations per half-edge in the case of quadrilateral meshes – right-flip, left-flip, split, collapse, and global-split. There is some redundancy in this representation of actions on the mesh. For instance, flipping a half-edge and its twin are equivalent operations. We choose to retain this redundancy because (a) it fits in well with our half-edge framework, (b) the size of the state representation is larger only by a constant factor, and (c) it exposes the symmetries in the half-edge representation and may be seen as data augmentation in our state representation leading to more robust learning. Further, some actions – like the quadrilateral split – are not equivalent when performed on a half-edge and its twin.

2.4 Formulation as a Deep Reinforcement Learning Problem

Constructing the reward function

Clearly, a mesh with all regular vertices will have a score $s = 0$. Under the assumption of the heuristic described sec. in 2.2, all the topological edit operations described in sec. 2.2 are *zero-sum* leaving the quantity $s^* = |\sum_i \Delta_i|$ invariant for a given mesh. This does not hold true if we change the heuristic for the desired degree of newly introduced vertices from what we described in sec. 2.2. If a mesh contains irregular vertices all of the same sign then its global score eq. 2.1 cannot be improved. s^* provides a lower bound on the score s,

$$s^* = \left| \sum_{i}^{N_v} \Delta_i \right| \leq \sum_{i}^{N_v} |\Delta_i| = s \tag{2.3}$$

We call s^* the *optimum score*. It is not clear if a score s^* can always be attained for a given mesh, however it serves as a useful measure of performance. The goal of our reinforcement learning framework is to learn sequences of actions that minimize s for a given mesh. In particular, consider a mesh M_t with score s_t at some time t. We now perform a mesh editing operation a_t on it to obtain mesh M_{t+1} with score s_{t+1}. Our agent is trained with reward r_t,

$$r_t = s_t - s_{t+1} \tag{2.4}$$

An agent starting with an initial mesh M_1 transformed through a sequence of n operations $a_1, a_2, \dots a_n$ collects reward $r_1, r_2, \dots r_n$. We consider the discounted return from state M_t as,

$$G_t = \sum_{k=t}^{n} \gamma^{k-t} r_k \tag{2.5}$$

with discount factor γ. (We use $\gamma = 1$ in all of our experiments.) Observe that the maximum possible return from this state is $G^* = s_t - s^*$. Thus, we consider the normalized return $\overline{G_t}$ as the *advantage* function to train our reinforcement learning agent,

$$\overline{G_t} = \frac{G_t}{s_t - s^*} \tag{2.6}$$

The return eq. 2.5 collected on meshes of different sizes will be different simply because larger meshes tend to have more irregularities. By normalizing the return in eq. 2.6, we ensure that actions are appropriately weighted during policy optimization. The mesh environment terminates when the mesh score $s_t = s^*$ or when a given number of mesh editing steps have been taken. We choose the maximum number of steps to be proportional to the number of mesh elements in the initial mesh.

While our current experiments are based on the objective described above, one could consider several modifications. Depending on the application, irregularities on the boundary may be more or less desirable than irregularities in the interior of the domain. The objective function can capture this difference in preference by weighting the contribution of boundary vertices and interior vertices differently when computing the score s. We could also consider improvement in element quality in the objective function. However, this would require a consideration of geometry along with topology and is reserved for future work.

Convolution operation on the DCEL data-structure

All of the actions, apart from the global-cleanup, affect the topology of the mesh locally. In order to determine if an action produces desirable outcomes in a particular neighborhood of the mesh, we need to understand the topology of this neighborhood. We require a representation of the local topology around each half-edge in order to select a suitable operation. In the language of reinforcement learning, this representation of the local topology is the *state* of a half-edge. The connectivity information in the immediate neighborhood of a half-edge is most relevant to determine the appropriate action to take in this neighborhood. We present here a convolution operation on the DCEL data-structure that encodes topological information around every half-edge. Indeed, this operation may be interpreted as a convolution on the graph induced by the half-edge connectivity. Iterative application of this convolution encodes topological information in a growing field-of-view around every half-edge. Further, this convolution operations can be efficiently implemented on modern GPU hardware.

Determining the appropriate action to take on a given half edge requires us to inspect the degree and irregularity of vertices in a neighborhood around the half-edge. Since the meshes we consider are unstructured, it is not immediately obvious which vertices to consider and in what order to consider them in. Our key observation is that the fundamental DCEL operations can be leveraged to construct a state representation for each half-edge that has a specific ordering. Our convolution operation requires two fundamental pieces of information both of which are easily available from the DCEL. For each half-edge we need to know the indices of (a) all the cyclic-next half-edges from the given element, and (b) the twin half-edges from the neighboring element. (a) is easily achieved by using the `next` operation repeatedly – 3 for triangles and 4 for quadrilaterals. (b) is fundamentally part of the DCEL data-structure.

As described in sec. 2.3, there is a natural global ordering for all the half-edges in the mesh. Half-edges from the same element appear sequentially in this global ordering. If the half-edges are stored in this order, the cycle operation can be implemented efficiently as a sequence of matrix `reshape` operations which are provided by most array based programming languages. Consider a mesh with N_h half-edges with the state of each half-edge represented by an N_f dimensional vector. This data when stored in sequential order can be represented by a matrix $x \in \mathbb{R}^{N_f \times N_h}$. Algorithm 1 describes the cycle operation applied to this state matrix for triangular meshes. The extension to quadrilateral meshes or other polygonal meshes is straightforward. (We assume that n-dimensional arrays are stored in column-major order. We adopt a syntax that closely follows the Julia/MATLAB Programming Language.)

Algorithm 1 Cycle operation on triangular meshes

```
  Input x $\in \mathbb{R}^{N_f \times N_h}$
  Output y $\in \mathbb{R}^{3N_f \times N_h}$
x ← reshape(x, N_f, 3, :)
x1 ← reshape(x, 3N_f, 1, :)
x2 ← reshape(x[:, [2, 3, 1], :], 3N_f, 1, :)
x3 ← reshape(x[:, [3, 1, 2], :], 3N_f, 1, :)
y ← concatenate x1, x2, and x3 along the second dimension (i.e. columns)
y ← reshape(y, 3N_f, :)
```

Information from twin half-edges is easily obtained by selecting the appropriate columns from the feature matrix. We use a learnable vector as the twin feature for edges on the boundary. This vector is part of the agent's parameter space and may be used by the agent to represent a useful signal indicating the boundary of the geometry. The same vector is used as the twin feature of all half-edges on the boundary. Our basic convolution operation involves cycling the current feature matrix, obtaining the features from the twin half-edges, and concatenating all of the features together. The resultant matrix is processed by a linear layer, followed by normalization and a non-linear activation function. We use LeakyReLU as our activation function. We refer to this operation as a *DCEL convolution block*. Under the operation of each convolution block, every half-edge receives information from all the half-edges within the same element and the twin half-edge from the adjacent element. After repeated application of such blocks, the final feature matrix will contain an encoding of the local topology in a field-of-view around every half-edge. The size of this field of view grows linearly with the number of convolution blocks. We use five convolution blocks in all of our experiments.

The initial feature matrix fed to the model is $x_0 \in \mathbb{R}^{2 \times N_h}$. Recall from sec. 2.3 that each half-edge is associated with a vertex. The initial feature matrix consists of the degree and irregularity of the associated vertices for every half-edge. This initial feature matrix is projected to a high dimensional embedding space using a linear layer on which the convolution described above is applied. The final layer projects the features into an $N_a \times N_h$ matrix where N_a is the number of actions per half-edge.

Action selection by the agent

The size of meshes can vary as the agent manipulates the mesh. The total number of actions available to the agent varies with the size of the mesh. To ensure that

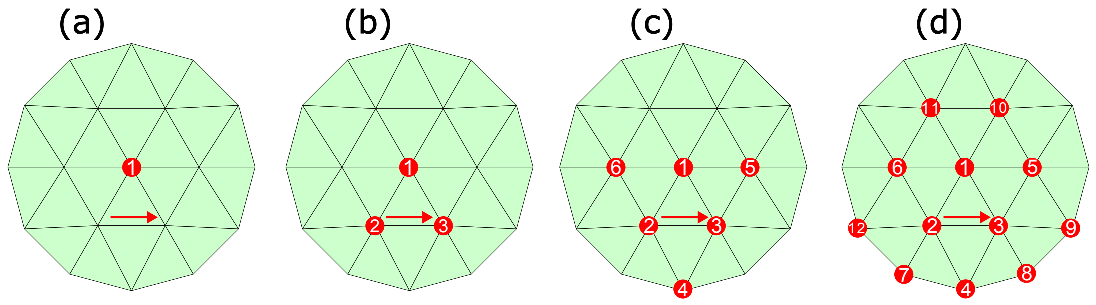


Figure 2.5: Repeated application of the convolution operation produces an increasing field of view around every half-edge. For triangles, we associate every half-edge with the vertex opposite it in the same triangle (fig. a). A `cycle` operation gathers information from the remaining vertices in the element (fig. b). Repeated application of `twin` and `cycle` produces the ordered list of vertices in fig. (c) and (d).

the agent can generalize across different mesh sizes, we found it important that our policy is represented by a fixed sized vector representing the probabilities of selecting various actions.

To do this, we generate a list of vertices which we call the *template* around each half-edge (see fig. 2.5.) The template can be constructed using operations similar to the convolution described in sec. 2.4. Initially, every half-edge has the index of the vertex it is associated with. After a *cycle* operation, every half-edge receives the indices of the vertices that are cyclic *next*. After a *twin* operation, every half-edge receives vertex indices from neighboring elements. Notice that there is some repetition in indices which can be avoided by selecting appropriate rows of the index matrix after a *cycle* or *twin* operation. These operations are repeatedly applied to grow the size of the template. We use dummy vertices if the template goes outside the boundary of the mesh. We then compute the score eq. 2.1 restricted to each template. This is a measure of the local irregularity around every half-edge. The irregularity of dummy vertices is set to zero ensuring that they do not contribute to the score of the template. Action selection is then restricted to the half-edges in the template with the highest local score with ties broken randomly. Thus we consider an $N_a \times N_l$ subset of the output feature matrix from sec. 2.4 where N_l is the number of half-edges in the template. This matrix is flattened and passed through a softmax layer to obtain a probability distribution over actions in the template. We sample from this distribution to take a step into a new mesh state. Figure 2.6 shows an

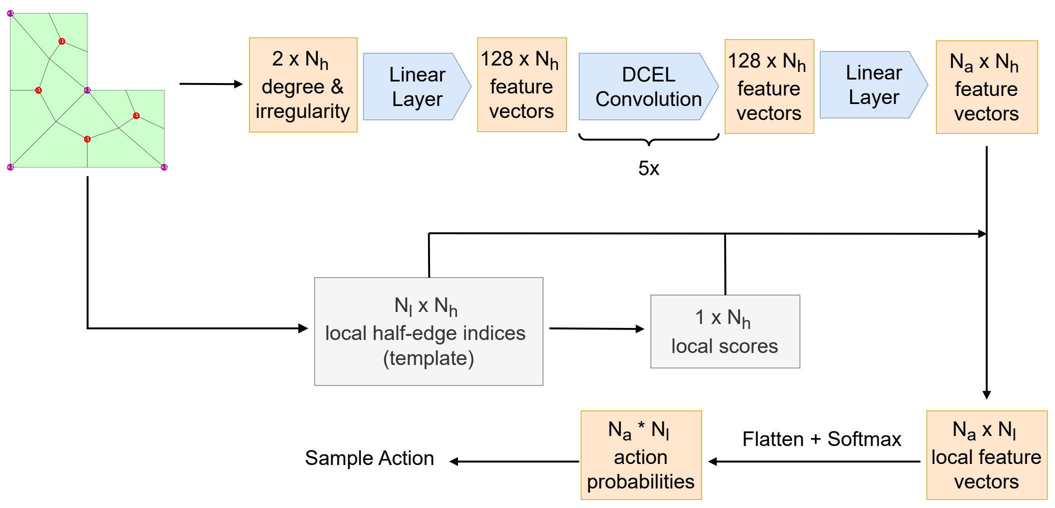

2 x N_h degree & irregularity
Linear Layer
128 x N_h feature vectors
DCEL Convolution
5x
128 x N_h feature vectors
Linear Layer
N_a x N_h feature vectors
N_l x N_h local half-edge indices (template)
1 x N_h local scores
N_a x N_l local feature vectors
Flatten + Softmax
N_a * N_l action probabilities
Sample Action

Figure 2.6: Illustrating the action selection process on meshes. Half-edge features consisting of the degree and irregularity of the associated vertex are first projected to an embedding space. Convolution on the DCEL data-structure is performed on these embeddings. Local templates are constructed around all half-edges to obtain a local measure of mesh irregularity. Action selection is restricted to the local template with the highest measure of irregularity. The final feature matrix is flattened and passed through a softmax layer to obtain action probabilities (i.e. a policy). This distribution is sampled to determine the action to take.

illustration of the action-selection process for a given mesh.

In the current implementation, convolution is performed on the entire mesh. Subsequently, the local irregularity measure is used to restrict action selection to the local template. While this works fine for our small examples, it would be computationally wasteful on larger meshes since convolution is performed on many half-edges that may not be included in the final template. In future implementations we will first choose the local template and only perform convolutions on this subset of half-edges.

The main purpose of constructing the template is to obtain a local measure of the irregularity score which enables *fast* selection of candidate regions. This ensures that action selection is restricted to regions with a high potential for reward. Further, we are able to consider a fixed size action space restricted to this candidate region, enabling the agent to generalize across mesh sizes. Alternate strategies may be adopted for candidate selection instead. For example, using breadth-first-search to find the K-nearest neighbors of a half-edge in the graph induced by the half-edge connectivity. The score eq. 2.1 restricted to this K-neighborhood can be used as a measure of the local irregularity in the mesh.

Training the agent in self-play

The initial states for self-play are randomly generated polygonal shapes. We randomize the degree of polygon (i.e. number of sides of the polygon) between set bounds. We perform Delaunay refinement meshing of this shape and use that as the input to the triangular mesh agent. For the quadrilateral agent, we perform the Catmull-Clark splits on the triangles to get an all quad initial mesh.

The agent is allowed to interact with the mesh and perform operations on it for a finite number of steps or until the agent achieves the optimum score s^* whichever comes first. Because the size of the mesh environment can be variable, the *value* or expected reward that an agent can receive from a given mesh environment is variable and cannot be inferred purely from the local representation of state that we employ. This makes it challenging to use value function based algorithms such as Deep Q-Learning [45] or Soft Actor-Critic [28]. Therefore, we train our agent using an actor-only version of the Proximal Policy Optimization (PPO) [60] algorithm without a critic (i.e. value function) network. PPO is one of the most widely used deep-reinforcement learning algorithms. It falls within the family of policy-gradient algorithms [67] with additional constraints that prevent large changes in the policy distribution during training. This ensures training stability and uniform policy improvement over the course of training. We use eq/ 2.6 as the advantage function in the PPO algorithm. We add an entropy regularization to the loss function to avoid local minima and balance exploration with exploitation. Full details of the hyperparameters used are provided in table 2.1.

The agents were trained for approximately 24 hours on a single Nvidia GTX 2080TI GPU. As the learning curves in figs. 2.7a and 2.10a show, the agent reaches its optimal performance fairly quickly after which performance plateaus implying that the full 24 hours was not necessary to train the model to a good level of performance. The inference cost of each step of the agent is on the order of a few milliseconds and is hardware dependent. Note that we did not optimize for model throughput.

Hyperparameter	Value
PPO ϵ parameter	0.05
Minibatch size	128
Epochs per PPO iteration	5
Trajectories sampled per PPO iteration	200
Number of PPO iterations	2000
Weight of entropy loss	0.001
Learning rate	10^{-4}
Discount factor γ	1
Number of DCEL convolution blocks	5
DCEL convolution hidden layer size	128

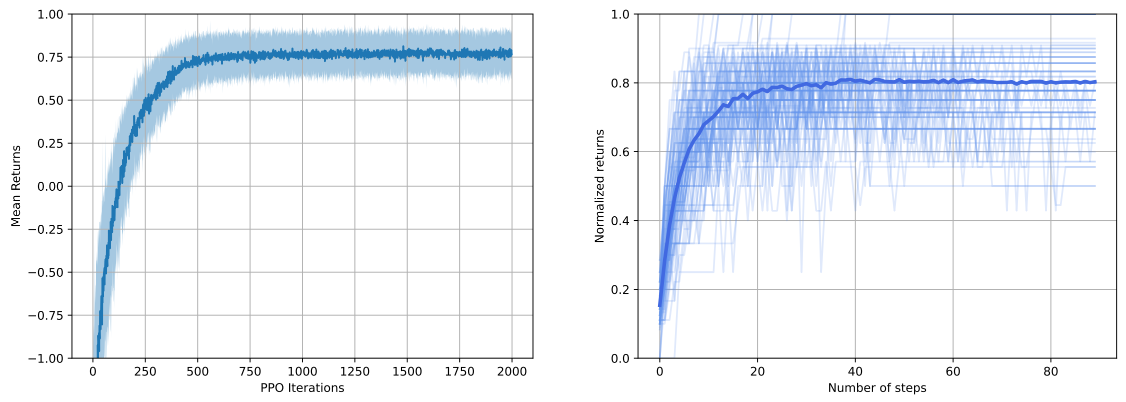
Table 2.1: Hyperparameter settings used to train the agent

2.5 Results

Triangular meshes

The triangular mesh agent was trained on random shapes consisting of 10 to 30 sided polygons. The initial mesh was a Delaunay refinement mesh generated by the Triangle package [61]. Figure 2.7 shows the learning curves of our agent over training history, and the performance of the trained model over 100 rollouts. The average normalized single-shot performance over 100 meshes was about 0.81 ($\sigma = 0.11$). However, since the learned policy is stochastic, a simple way to improve the performance is to run the policy k times from the same initial state and pick the best mesh. Using $k = 10$ samples per mesh and averaging over 100 random meshes, the performance improved to 0.86 ($\sigma = 0.08$).

Table 2.2 demonstrates the generalization capability of the learned policy. By using a fixed sized local template, the same agent can be evaluated on meshes of various sizes with good results. We do observe some reduction in model performance on larger meshes. Irregularities tend to be separated by greater distances on larger meshes, requiring longer sequences of operations to effectively remove them. We illustrate an example of the trained agent's performance on a 40-sided polygon in fig. 2.9. While the agent is able to eliminate some irregularities that are near each other, the final mesh still contains quite a few irregularities that are separated by large distances. We need to bring irregularities near each other in order to regularize them. This requires a complex sequence of moves that our agent is unable to learn.



(a) Average performance over training history (b) Evaluating the trained agent on multiple rollouts

Figure 2.7: (a) Performance of the triangle mesh agent over the training history. Solid line represents the average normalized return over 100 meshes evaluated periodically during training. Shaded region represents the 1-standard deviation envelope. (b) Performance of the trained agent over 100 rollouts. The agent incrementally improves mesh quality up to a certain number of steps. Notice that returns do not increase monotonically, indicating that a greedy strategy may not be effective for this problem.

Further, since our state-representation relies on a local template, when irregularities are separated by large distances, our agent is unable to detect them through the local state representation.

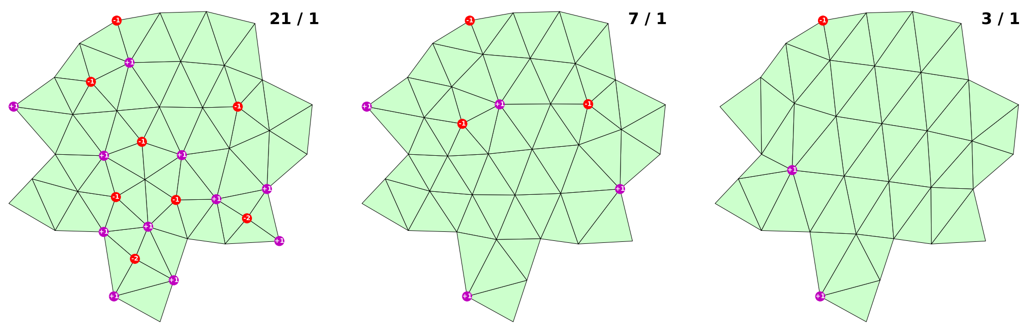
We note that the normalized performance of the agent in the quadrilateral mesh environment is significantly better (see sec. 2.5.) Our experiments indicate that the use of global mesh editing operations such as the global-split and global-cleanup are instrumental in achieving this performance. The global cleanup, in particular, is effective at coarsening the mesh without introducing new irregularities. The cleanup operation brings irregularities closer to each other making it easier to combine them and regularize them. Defining such a clean-up operation for triangles is non-trivial and remains to be addressed through future work.

Quadrilateral meshes

The quadrilateral mesh agent was trained on random shapes consisting of 10 to 30 sided polygons. Figure 2.10a shows the average normalized returns over training for the quadrilateral mesh agent. We observe that the agent quickly learns operations

Polygon degree	Average	Standard deviation
3 - 10	0.83	0.19
10 - 20	0.87	0.08
20 - 30	0.83	0.10
30 - 40	0.78	0.08
40 - 50	0.75	0.07

Table 2.2: Evaluating the triangle mesh agent on various sized random polygons. The agent was trained purely on 10 - 30 sided polygons but is able to generalize to other polygon sizes with minor deterioration in performance. The agent was evaluated by picking the best of 10 rollouts per geometry, with the statistics computed over 100 randomly generated shapes. The results demonstrate the effectiveness of using a fixed-sized local template which enables better generalization to different mesh sizes.



(a) (b) (c)

Figure 2.8: Example rollout of the triangular mesh agent on a 20-sided polygon. Irregular vertices are marked in color, with the current score and optimum score shown at the top right for each figure. (a) is the initial Delaunay refinement mesh (b) is at an intermediate stage and (c) is the final mesh after 27 operations.

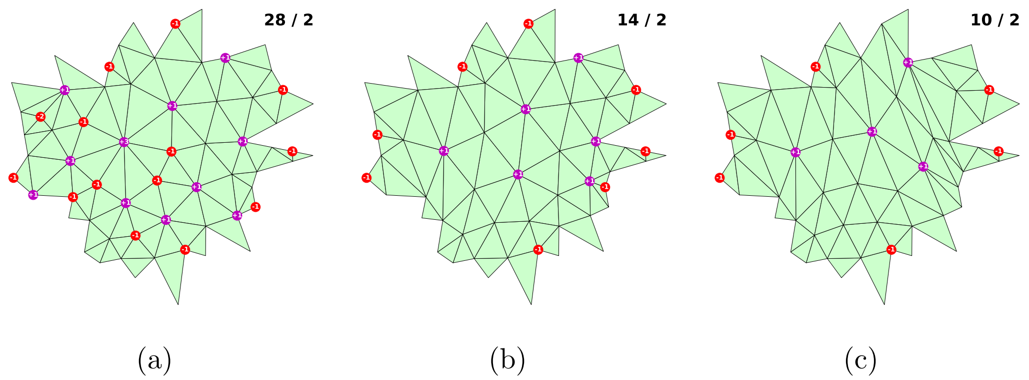
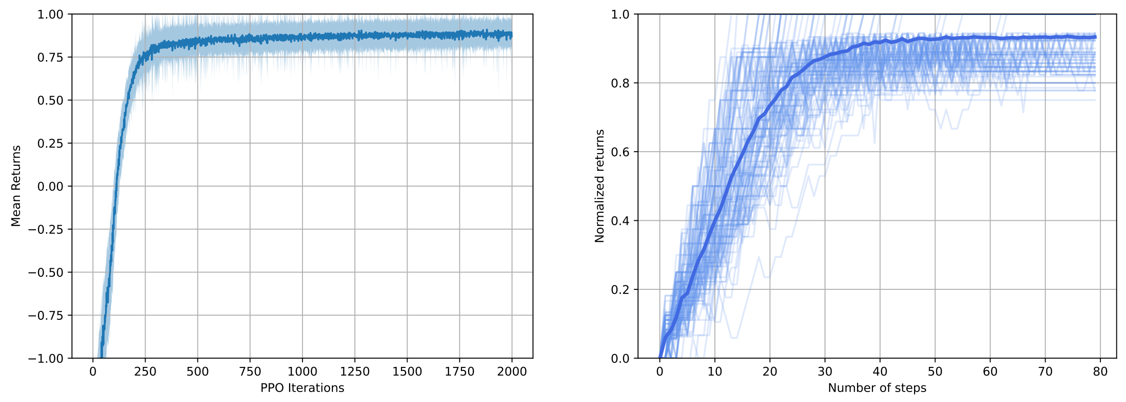


Figure 2.9: Example rollout of the triangular mesh agent on a 40-sided random polygonal shape. (a) is the initial mesh, (b) is an intermediate state, and (c) is the mesh with the lowest score during policy rollout. Notice that the agent is able to improve the irregularity score of the mesh. However, the final mesh contains irregular nodes that are separated by several mesh elements. These irregularities require complex sequences of moves to bring them together and regularize them. Further, the agent has a limited field of view controlled by the size of the local template. Irregularities outside the local template will not be seen by the agent.

that significantly improves the connectivity of the mesh to nearly optimal. Performance was assessed periodically during training by evaluating the model on 100 randomly generated meshes. Figure 2.10b shows the evaluation of the best performing model on 100 trajectories. We observe that performance depends on the maximum number of steps given to the agent up to a certain point. The average normalized single-shot performance over 100 meshes was about 0.95 ($\sigma = 0.05.$). Using $k = 10$ samples per mesh and averaging over 100 random meshes, the performance improved to 0.992 ($\sigma = 0.02$).

Since our state representation is a fixed-sized local template around a half-edge of interest, our model generalizes well to polygons that were not part of the training dataset. Table 2.3 shows the performance of a model trained on 10 - 20 sided polygons that is able to generalize to larger sized polygonal shapes. We do observe some drop in the performance of the agent when mesh sizes are increased. This is consistent with our observations for triangular meshes. Irregularities are often separated by several mesh elements in larger meshes, requiring longer range sequences of operations to regularize them. The complexity of these operations, coupled with the local nature of our state representation likely causes the deterioration in the agent's performance.



(a) Average performance over training history

(b) Evaluating the trained agent on multiple rollouts

Figure 2.10: (a) Performance of the quadrilateral mesh agent over the training history. Solid line represents the average normalized returns evaluated over 100 meshes. Shaded region represents the 1-standard deviation envelope. The curve demonstrates that the agent is able to achieve good performance quite rapidly, and the learning remains stable over many training iterations. (b) Evaluating the trained model over multiple rollouts. Solid line represents the average performance of 100 rollouts. The graph demonstrates that increasing the maximum number of operations available to the agent has a big impact on performance initially, but only up to a certain point. Returns do not increase monotonically, highlighting that a greedy strategy may not be effective in this setting.

Figures 2.11 and 2.12 show some example rollouts on various polygon sizes. Note that the "optimal" mesh produced by the agent in fig. 2.12c contains an irregular vertex with degree 2 on the bottom boundary. This mesh is considered optimal according to our objective function eq. 2.1 since its score is equal to the optimal score s^* for this configuration. However, there are many applications wherein a configuration such as fig. 2.12d is preferred, with the irregularity moved to the interior of the domain. This latter configuration is achieved via a post-processing step by applying an edit similar to the global split. Observe that both of these configurations have exactly the same objective score and are thus considered equivalent by our metric. If irregular vertices on the boundary are not preferred, they may be fixed by post-processing steps. Alternatively, the objective function may be modified by using a higher weight on irregularities on the boundary. This can encourage the agent to learn to move irregularities away from the boundary into the interior of the domain.

Polygon degree	Average	Standard deviation
10 - 20	0.98	0.03
20 - 30	0.97	0.06
30 - 40	0.94	0.14
40 - 50	0.91	0.13

Table 2.3: Evaluating the quadrilateral mesh agent on various sized random polygons. The agent was trained purely on 10 - 20 sided polygons, but is able to generalize to larger polygonal shapes with minor deterioration in performance. The agent was evaluated by picking the best of 10 rollouts per geometry, with the statistics computed over 100 randomly generated polygonal shapes. Using a fixed-sized local template enables stronger generalization to different sized meshes.

Generalization to new geometries

Figures 2.13 and 2.14 show zero-shot transfer to never before seen geometries like L-shape, star-shape, etc. The agent is able to handle geometries with re-entrant corners and notches which were not explicitly part of the training space. Further, our model, which was trained exclusively on genus-0 geometries with no interior holes is able to generalize zero-shot to genus-1 shape consisting of a square hole in a circular shape.

We highlight the use of our approach in block decomposition of complex shapes into coarse quadrilateral elements. The global cleanup operation is particularly effective for this application as it is effective at coarsening the geometry without introducing new irregularities, and bringing existing irregularities closer to each other which makes it simpler to regularize them.

2.6 Conclusion

We presented here a method that learns to improve the connectivity of triangular and quadrilateral meshes through self-play reinforcement learning without any human input. A key contribution of this work is a parameterized method to generate a representation of the local topology in mesh neighborhoods. This enables appropriate selection of standard topological mesh editing operations which result in the reduction of irregular vertices in the mesh. Our method is built on the DCEL data-structure which allows the same framework to work on any planar 2D mesh with the discussion in this paper restricted to triangular and quadrilateral meshes.

When optimizing for connectivity, it is recommended to work with the coarsest possible mesh that captures the details of the geometry being considered. Optimal

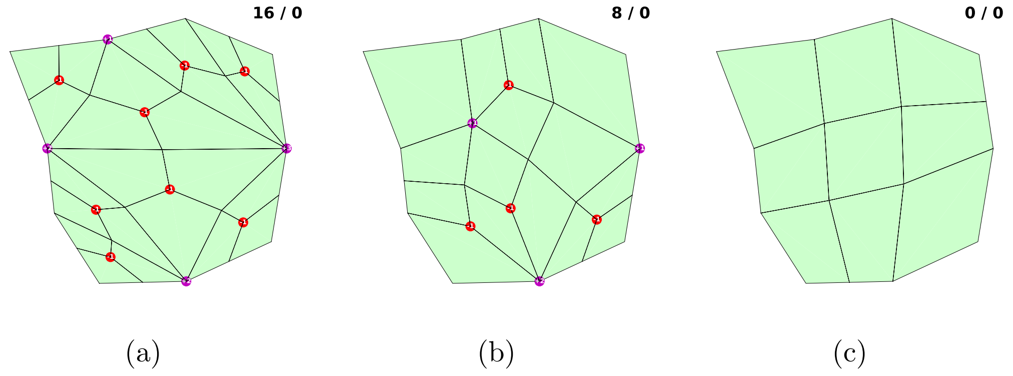


Figure 2.11: Example rollout for a 10-sided polygon. Irregular vertices are marked in color. The mesh score and optimal score are shown at the top right for each figure. (a) is the initial mesh after Delaunay triangulation and catmull-clark splits, (b) is at an intermediate stage, and (c) is the final mesh after 18 operations.

strategies to regularize meshes consist of sequences of operations that combine irregularities together. It is easier to regularize coarser meshes because irregularities are relatively closer to each other on these meshes. Irregularities often become isolated on finer meshes, and require longer sequences of complex operations to regularize them. Both triangles and quadrilateral meshes can be globally refined without introducing irregular vertices. Some applications, like numerical simulation, demand finer meshes for the sake of simulation accuracy. Thus, once the connectivity of a coarse mesh has been optimized, it can be easily refined to the desired resolution while maintaining its regularity.

A major advantage of artificial intelligence is its ability to discover heuristics that are too laborious and cumbersome for humans to identify, formulate, and prescribe. There are several areas in mesh generation where the automatic discovery of such heuristics can significantly aid engineers in their work. We hope that this paper demonstrates one such use-case.

2.7 Future Work

There are several exciting directions of future research that we highlight here,

- **Incorporating value function:** Most deep reinforcement learning methods benefit from having a value function as this can help speed-up training. Our

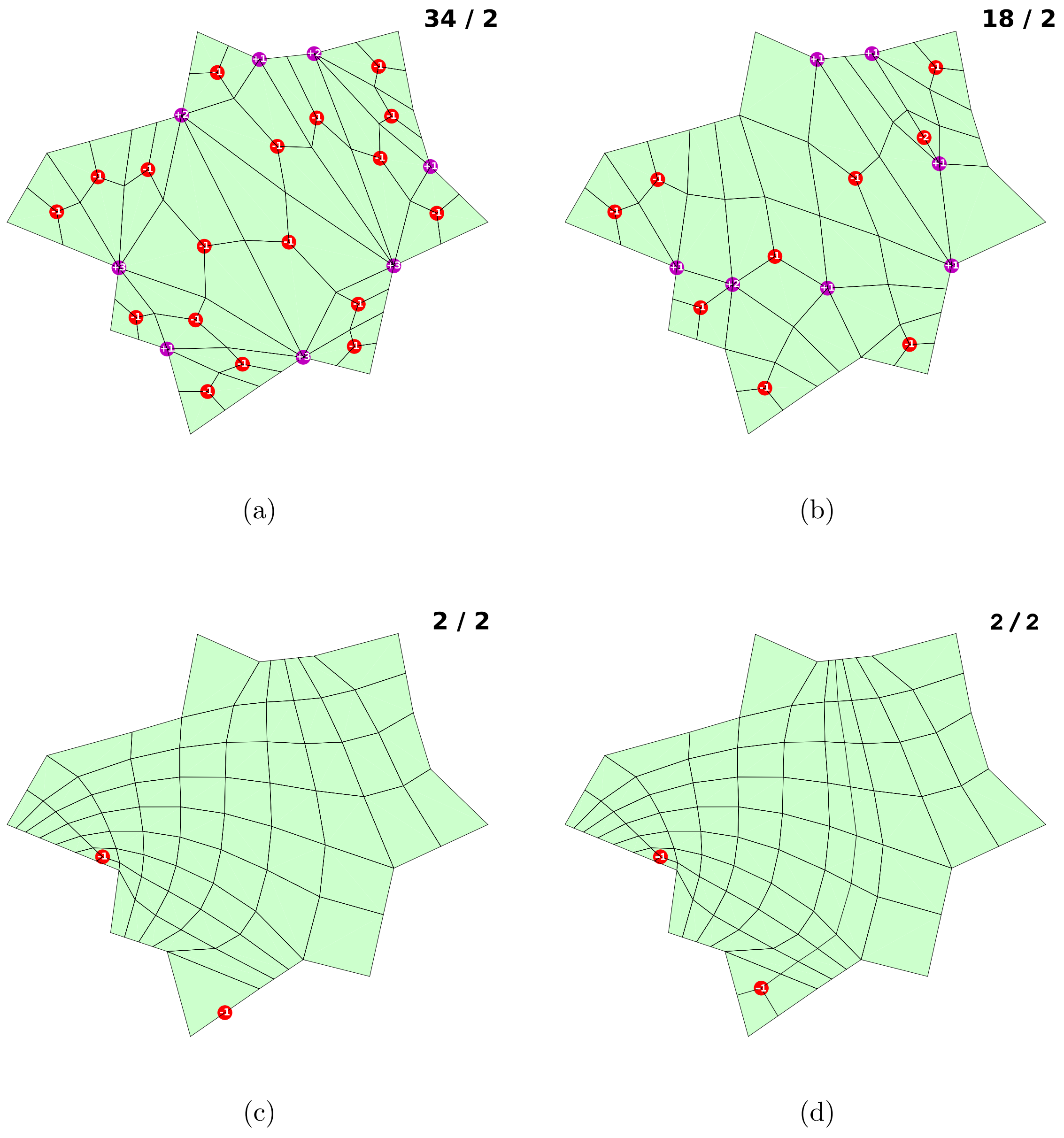
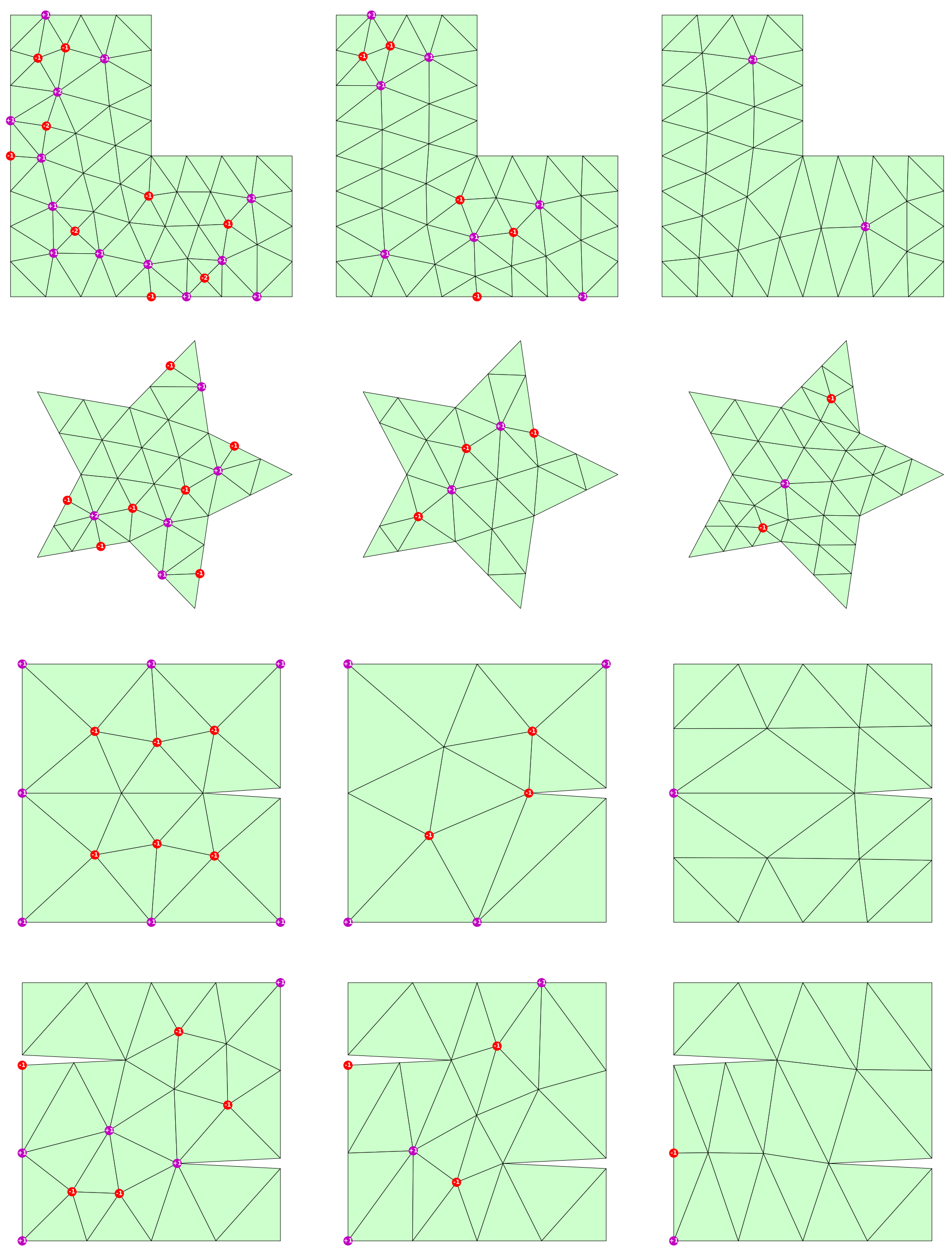


Figure 2.12: The same agent as before is able to optimize a 20-sided polygonal shape using 40 operations. (a) initial mesh, (b) intermediate mesh, (c) final mesh produced by the agent, (d) degenerate vertices on the boundary can be post-processed using an operation similar to the global split. Note that meshes (c) and (d) have the same score as measured by eq. 2.1 thus our agent does not prefer one over the other. If the application demands that there be no degenerate vertices, these irregularities can be eliminated through a final post-processing step.

current formulation makes it challenging to estimate state value because we



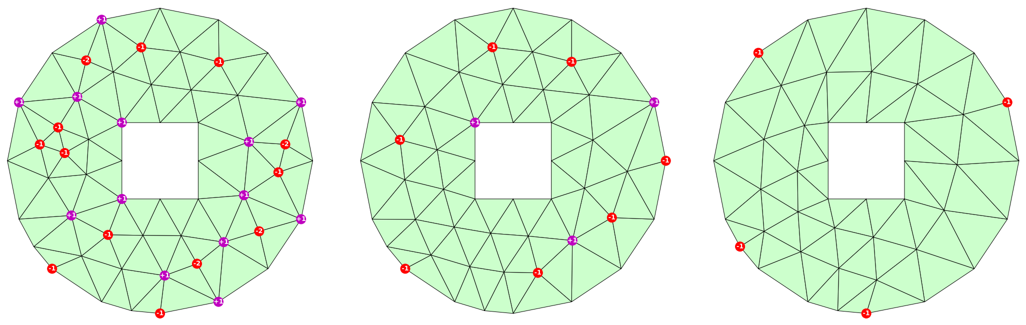
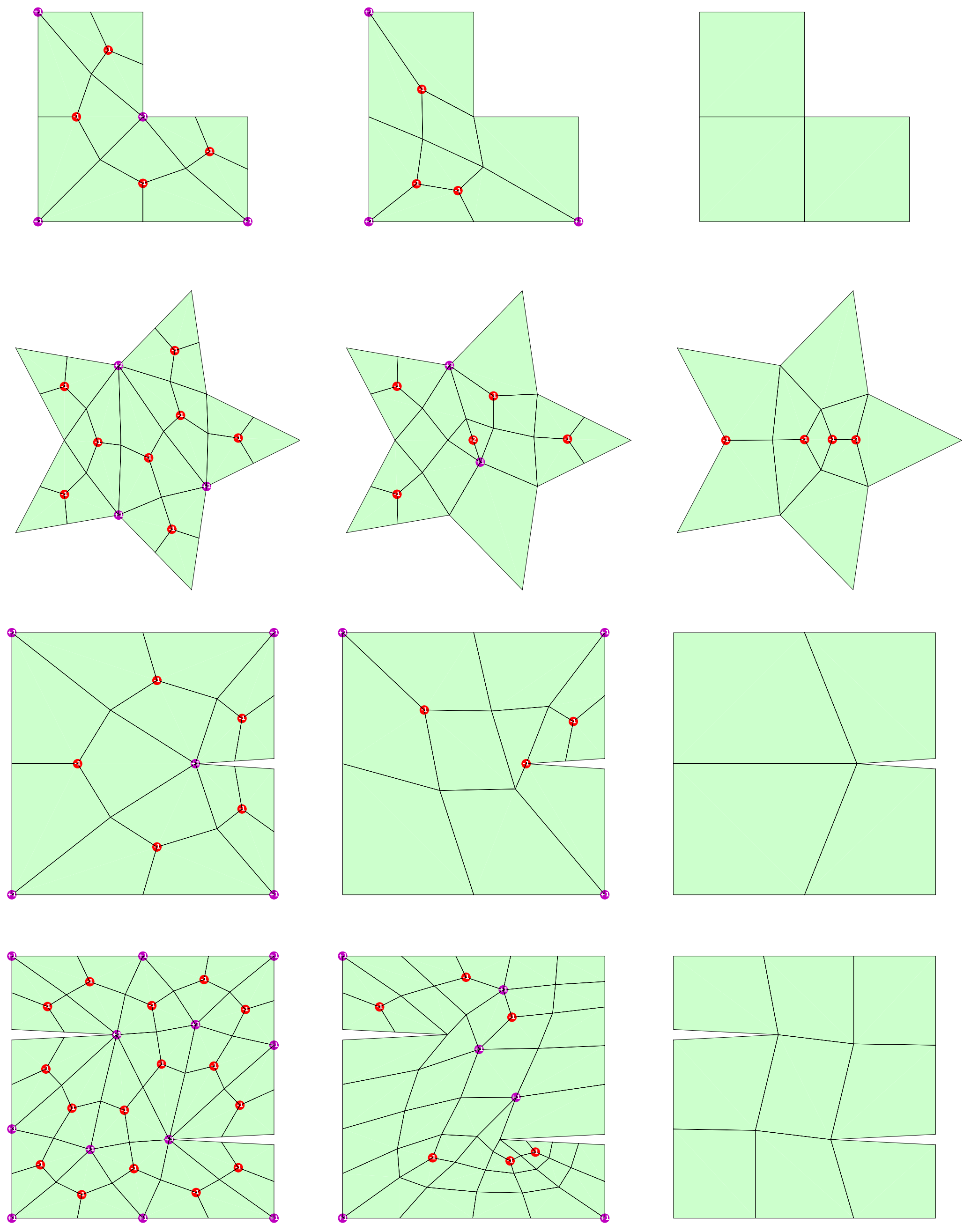


Figure 2.13: The triangular mesh agent demonstrates zero-shot transfer on geometries that were not seen during training, including geometries with re-entrant corners like the star shape and the single and double-notch domains. First column is the initial mesh, second column is the intermediate mesh as the agent optimizes connectivity, and the third column is the final mesh after optimization. Since our model is based purely on connectivity, we can directly transfer the model onto geometries with holes even though such geometries were never seen during training. Notice that in several of the examples including the L-domain, single-notch, and the square hole in the circle, the model achieves the optimal score and the remaining irregularities cannot be eliminated as they are intrinsic to the geometry.

employ a local representation of state that does not provide sufficient information to estimate global value. Addressing this challenge is the focus of our current work.

- **Policy improvement with tree search:** our learned policy is stochastic and may be combined with e.g. Monte Carlo Tree Search (MCTS) [16] to efficiently search for optimal meshes for a specific geometry. The performance improvements that we observe from our naive best-of-k method in sec. 2.5 and 2.5 indicates that MCTS could be effective at improving the performance of our trained model. Such an approach would be similar to the AlphaZero [63] system.

- **Optimizing for element quality:** to achieve this, our model would need to additionally receive geometric information (e.g. vertex coordinates) as input. This can be easily achieved by including the coordinates of vertices as part of the input features to our model (see sec. 2.4). We expect that the coordinates need to be normalized e.g. affine transform half-edges (and all vertices in its template) to a normalized coordinate system (e.g. $[0, 1]$.)



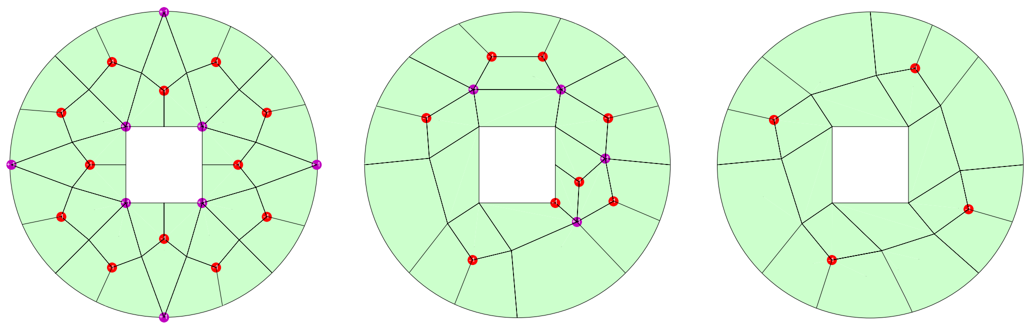


Figure 2.14: The quadrilateral mesh agent demonstrates zero-shot transfer on geometries that were not seen during training, including geometries with re-entrant corners like the star shape and the single and double-notch domains. First column is the initial mesh, second column is the intermediate mesh as the agent optimizes connectivity, and the third column is the final mesh after optimization. Since our model is based purely on connectivity, we can directly transfer the model onto geometries with holes even though such geometries were never seen during training. We are particularly interested in coarse meshes representing block decompositions of more complex shapes. The cleanup operation is particularly useful in achieving coarse meshes. This is most evident in the single notch and double notch example in row 3 and 4. Notice that the star-shaped domain and the circular mesh with a square hole contain intrinsic irregularity that cannot be improved upon with our prescribed operations and heuristic.

- **Extension to 3D:** We expect that our method can leverage the equivalent of the half-edge data-structure in 3D [21] to learn topological mesh editing operations on tetrahedral and hexahedral meshes. Determining optimal sequences of operations in 3D is highly challenging, and a self-learning method would have significant use.

We anticipate that extension to 3D applications can have significant utility and will likely attract further research interest. We discuss here some challenges that we foresee and possible methods to address these challenges.

- A suitable action space needs to be clearly defined for 3D mesh types. It is preferable that these actions are local in nature to minimize computational cost. Prior work defining actions on tetrahedral meshes [62, 34, 26] and hexahedral meshes [39, 69] will be useful to consider in this regard.

- Our framework parameterizes mesh editing operations in terms of geometric primitives. For 2D, our geometric primitive was the half-edge. Selecting a half-edge and an associated edit operation unambiguously identifies an edit operation on a mesh. A similar parametrization needs to be developed for 3D meshes to adopt our framework. We anticipate that both half-edges and *half-faces* would be required to parameterize mesh editing operations in 3D. For example, the popular 2-3 face swap operation in tetrahedral meshing can be parameterized by selecting a half-face and prescribing the swap operation on it.

- We require an extension of the convolution operation on the 3D mesh data-structures. In 3D, there is a connectivity structure on half-faces along with half-edges [21]. The convolution operation needs to operate on both of these connectivities.

- Selecting a candidate region in which to evaluate the reinforcement learning agent will be equally important in 3D applications due to the large variations in the number of elements. We anticipate that an objective score similar to eq. 2.1 can provide a simple way of evaluating regions with the potential for significant quality improvement.

- In 2D, the enclosed angle at a boundary vertex provides a simple method of identifying the desired degree. Specifying the desired connectivity, particularly on boundaries, is essential to obtaining a well-defined method in 3D.

Chapter 3

Optimizing the topology and geometry of general polygonal meshes using deep reinforcement learning

We present a reinforcement learning (RL) agent that is able to generate meshes of 2D shapes. Our mesh environment supports arbitrary 2D polygonal element types, and a unified action space that is independent of element type. This enables our agent to learn to mesh with arbitrary polygonal elements in a completely self-supervised way. We demonstrate the inclusion of various objective measures based on the regularity of faces, vertices, and induced angles in the optimization worflow.

3.1 Introduction

In the previous chapter, we discussed a methodology to improve the connectivity of triangular and quadrilateral meshes. We framed the problem as a sequential decision making process, constructed a state representation of meshes, and trained a neural network to automatically learn move sequences to optimize a given objective function which is a measure of mesh regularity.

There are three key areas where we can improve upon the previous method,

- The method relies on an initial mesh generation algorithm and as such is a *mesh improvement* method. Further, we empirically observed that our RL agents from Chapter 2 performed better on coarse meshes. Our intuition is that irregularities are separated by smaller distances on coarser meshes, therefore requiring fewer operations to regularize.

- The method requires that meshes are homogeneous with a single element type. This constrains the action space to be specialized towards a specific element type e.g. triangular or quadrilateral elements. Specifying a new element type would require the specification of an entirely new action space. This also means that the agent can only perform operations that will result in a topologically valid mesh at each step. This rules out exploring invalid intermediate states that may result in valid final states with better objective score.

- The state representation and objective function was only concerned with mesh connectivity and not with geometric quantities of interest such as induced angles, element quality, etc.

In this chapter, we will explore the possibility of loosening the above constraints. Firstly, we design a method whose input is any user provided 2D geometric shape thereby removing the dependency on an existing mesh generation algorithm. The user provided input shape is typically the coarsest geometric representation, which aligns well with our model's performance regime. Secondly, we allow our meshes to have heterogeneous element types and a unified common action space that can be utilized to generate meshes of *any* type. An agent for a new element type can be trained by changing a single input parameter. Finally, we demonstrate the inclusion of geometric quantities in the optimization workflow by including the the regularity of induced angles into the objective function.

There has been significant interest in leveraging machine learning (ML) methods to generate meshes from boundary representations. Early efforts in this area include [79] who adopted an element extraction approach. In [48, 49], the authors frame element extraction as a reinforcement learning (RL) problem and use mature solution algorithms such as Soft-Actor Critic (SAC). At each step, they build a finite dimensional state representation around a reference point on the current geometric boundary. The agent then chooses one of three primitive element extraction operations and executes it. DiPrete et al. [20] leverage a similar local state representation but instead train their agent to learn optimal axis-aligned cuts to decompose 3D planar axis-aligned shapes. A key challenge when working with unstructured data structures like meshes is to develop a finite dimensional *state* representation. While not a mesh generation method, Lim et al. [42] present a novel local state representation on triangular meshes by constructing a spiral template of vertices around a reference vertex. Papagiannopoulos et al. [50] adopt a supervised learning approach. They train 3 different neural networks to predict the number, location, and connectivity of vertices given a 2D contour. The neural networks are trained based on ground-truth data generated from a Constrained Delaunay Triangulation of 2D contours. Rakatosaona et al. [55] developed a differentiable triangulation system by leveraging Weighted Delaunay Triangulation. The triangulation is differentiable with respect to the vertex positions and weights allowing for optimizing e.g. mesh density and edge alignment.

While the above aim to *generate* triangulations, ML methods are sometimes used to adapt a mesh or generate input for a traditional meshing algorithm. Some of the earliest work on combining (ML) with mesh generation [4] is concerned with adapting a mesh based on error estimation on an initial coarse mesh. Adaptation can be made in general with respect to a nodal probability distribution function. Tingfan et al. [70] use neural networks to quickly estimate the solution field on a moving mesh thereby allowing for the mesh to adapt to the simulation. Other methods such as [31, 80] aim to train machine learning models to predict desired mesh densities but rely on traditional meshers to fulfill the meshing task. Rakatosaona et al. [55] tackle the problem of recovering a 3D surface mesh from a point cloud that was sampled from a surface. They train machine learning models to identify and map local patches of the point cloud to a 2D domain using logarithmic maps. They utilize Delaunay triangulation in the parametric domain and map the result back to the 3D surface while constraining neighboring patches to produce watertight manifold meshes. Chen et al. [13] train neural network models to learn valid mappings from a parametric reference domain to the physical domain thereby allowing to fit a reference mesh to a given geometry. In [19], a neural network is trained to infer direction fields based on human generated quadrilateral meshes. These direction fields are used as input to a quadrangulation algorithm [11] to produce the final mesh. Deng et al. [18] train neural network models to generate a 3D surface and associated Conjugate Direction Field (CDF) given a human sketch. The 3D surface and CDF are used to generate the final quadrilateral mesh.

The present work is distinguished from prior approaches in several key ways,

- We consider a highly general action space that encompasses the necessary operations for several different element types. We demonstrate results on triangles and quadrilaterals as a showcase, though other element types are trivially possible.

- We provide an efficient, parametrizable, local state-representation based on the Doubly Connected Edge List (DCEL) data-structure. Our representation is applicable to arbitrary, unstructured, heterogeneous 2D meshes. We describe the key local geometric and topological features that are necessary in our state representation.

- We illustrate a novel convolution operation on the state-representation above that encodes the geometry and topology from a user-defined field-of-view in a neighborhood of the mesh.
- We provide open-source software implementations of our mesh environments, training pipelines, and automated hyperparameter optimization workflow to provide a baseline for future work.

3.2 Problem Statement

We are concerned with decomposing a given two-dimensional shape into polygonal elements. We refer to any decomposition of the geometry as a *mesh* – the input is also, trivially, a mesh. Our meshes can contain heterogeneous element types consisting of polygons of arbitrary degree. The user is expected to provide the input geometry along with the degree (i.e. number of faces) of the polygon that the input mesh is to be decomposed into. We frame the problem as a sequential decision making process where we utilize simple mesh editing operations to perform the decomposition. Additionally, we design the method to minimize irregular vertices and improve element quality as measured by a prescribed objective function.

Input geometry

We support the representation of 2D geometries by defining their boundary representation as a sequence of straight edges. The geometry is represented by a set of vertices and a set of closed faces. Each face is expected to be a polygon of arbitrary degree. We define a face by specifying the vertices it is comprised of in counter-clockwise order. This representation also supports geometries with holes (see fig. 3.1 for some example geometries.)

Topological editing operations

Since we are dealing with very general geometric representations, we aim to design an action space that is suitably general. This ensures that an appropriate mesh decomposition of a given input geometry can be achieved through the composition of the available elementary actions.
We support 4 basic topological edit operations consisting of:

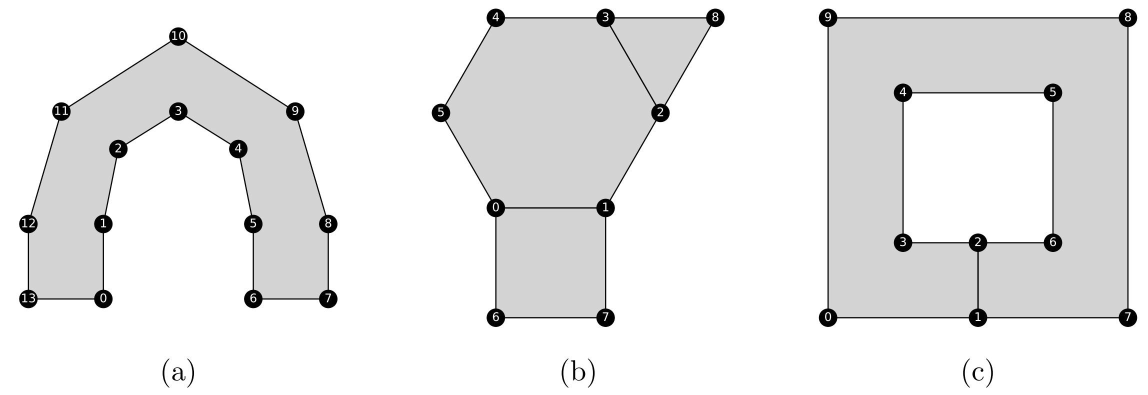


Figure 3.1: Example geometric representations. Geometries are represented as sets of closed faces. A face is defined as a closed loop of vertices in counter-clockwise order. (a) a geometry with a single face, (b) a geometry with multiple faces each of which has a different polygonal degree, (c) a geometry with a hole.

- Edge insertion – an edge can be inserted between two distinct vertices in a face, thereby splitting a face into two new faces.
- Edge deletion – an interior edge that is shared by two faces can be deleted, thereby merging the two faces. An edge on the boundary cannot be deleted. An edge between the same face (e.g. the edge 1-2 in fig. 3.1c) cannot be deleted.
- Vertex insertion – a new vertex may be inserted on any edge.
- Vertex deletion – a degree-2 vertex may be deleted thereby merging the two edges incident on that vertex. The input may optionally contain a set of vertices that are not allowed to be deleted – for example a vertex that is intrinsic to defining the geometry.

Faces and vertices are constrained to have degree at least 2. Operations that result in the violation of this bound are disallowed. The topological edit operations are illustrated through example in fig. 3.2.

Objective function

Consider a mesh with N_f faces and N_v vertices. Let f_i be the degree of face i and v_j be the degree of vertex j. The degree of a face is the number of edges in the face

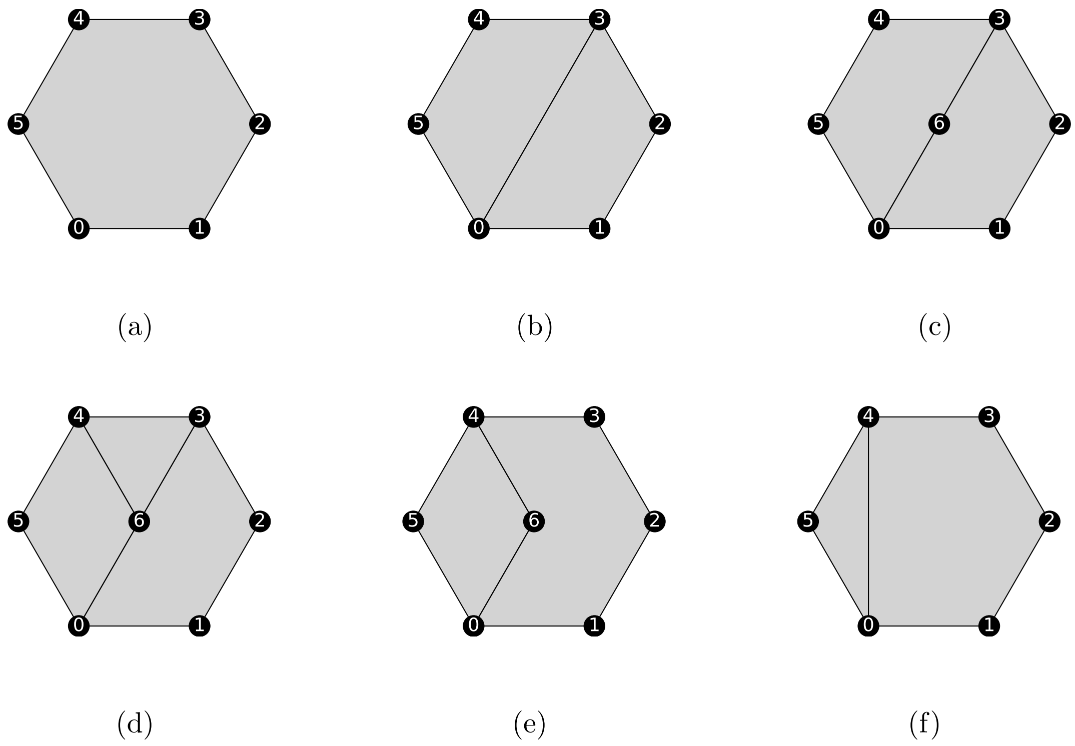


Figure 3.2: Illustration of topological edit operations: (a) initial polygon which transforms to (b) after an edge insertion between vertices 0-3. A new vertex is inserted on this edge to produce (c) from which we produce (d) with an edge insert between vertices 6-4. The edge between 6-3 is deleted to give (e). Finally, vertex 6 is deleted to produce (f).

(i.e. its polygonal degree). The degree of a vertex is the number of edges incident on that vertex. Let f^* be the desired polygon degree of the mesh elements that the geometry is to be decomposed into. Further, let v_j^* be the desired degree of vertex j. Notice that we take the desired face degree f^* to be constant for the entire mesh whereas we allow the desired vertex degree to be different for each vertex. We define the face irregularity as $\Delta_i^f = f_i - f^*$ and the vertex irregularity as $\Delta_j^v = v_j - v_j^*$. The face score s_f and vertex score s_v are then respectively,

$$s_f = \sum_{i=1}^{N_f} |f_i - f^*| \tag{3.1}$$

$$s_v = \sum_{j=1}^{N_v} |v_j - v_j^*| \tag{3.2}$$

A face score $s_f = 0$ implies that all the elements in the mesh have polygon degree equal to f^* which is the element type that we want to decompose the geometry into. A vertex score $s_v = 0$ implies that all vertices in the mesh are regular (i.e. they have degree equal to their corresponding desired degree) – this is often desirable in applications such as block decomposition and computer graphics rendering. While we would ideally aim to generate meshes with $s_f = s_v = 0$, this ideal score cannot always be achieved. Therefore, we frame this as an optimization problem instead where we aim to *minimize* the objective scores thereby creating meshes with minimum face and vertex irregularity.

The face score and vertex score are *discrete* in nature. However, it is common in the meshing literature to optimize for other quantities of interest such as element quality which are commonly *continuous* in nature. There are a variety of element quality metrics that are designed for specific element types such as triangles and quadrilaterals. Since we are dealing with general polygons, we consider a simple quality metric based on the induced angles in the mesh. If the desired face degree is f^*, the desired angle θ^* is taken to be the induced angle in a *regular* polygon with degree f^*. (Recall that a *regular* polygon has all angles equal.) The ideal angle, in degrees, is computed as,

$$\theta^* = \frac{f^* - 2}{f^*} \times 180^\circ \tag{3.3}$$

e.g. $\theta^* = 90^\circ$ for quadrilaterals ($f^* = 4$) and $\theta^* = 60^\circ$ for triangles ($f^* = 3$).

The angle irregularity is then defined as $\Delta_k^\theta = (\theta_k - \theta^*)/\theta^*$. If there are N_a angles in the mesh, the angle score s_θ is defined as,

$$s_\theta = \sum_{k=1}^{N_a} \left| \frac{\theta_k - \theta^*}{\theta^*} \right| \tag{3.4}$$

While s_f and s_v depend only on the degree of faces and vertices and not on the location (i.e. coordinates) of vertices, s_θ depends on the coordinates since this determines the induced angles in the mesh. When including s_θ as part of the objective during optimization, we typically need to smooth the mesh after performing a mesh edit operation to update the coordinates of the vertices. A simple choice is to use Laplace smoothing. More sophisticated smoothers based on element quality may also be used.

For the optimization procedure, we consider a weighted sum of the different scores as the final objective score,

$$s = w_f s_f + w_v s_v + w_\theta s_\theta \tag{3.5}$$

with the weights w_f, w_v, and w_θ suitably chosen depending on the end-user application.

Heuristics to determine desired degree

Our framework allows the desired degree of all vertices to be specified as part of the input. However, for the sake of simplicity, we adopt the following heuristics. The face desired degree f^* is expected to be a part of the user input. A regular f^*-sided polygon has as interior angle given by eq. 3.3 e.g. $\theta^* = 90°$ for quadrilateral elements. We set the desired degree of vertices such that the average included angle at each vertex is equal to θ^* when the vertex is regular. The desired degree of vertices is then,

$$v_j^* = \begin{cases} 360/\theta^* & \text{interior vertex} \\ \max\left(\lfloor \theta/\theta^* \rceil + 1, 2\right) & \text{boundary vertex} \end{cases} \tag{3.6}$$

where $\lfloor \cdot \rceil$ is the round to nearest integer operator and θ is the included angle of the input geometry at a particular boundary vertex.

Figure 3.3 shows a sequence of mesh editing operations applied to a simple L-shaped input geometry with the desired face degree $f^* = 4$. The corresponding desired angle is $\theta^* = 90°$.

Duality of meshes for face and vertex score

Meshes have a dual property wherein the *dual* of a mesh can be constructed by introducing a vertex for each face and edges for every pair of adjacent faces. (See fig. 3.4 for an example.) Duality transforms faces to vertices and vertices to faces. The dual of a regular mesh is a suitably regular mesh with a possibly different polygonal

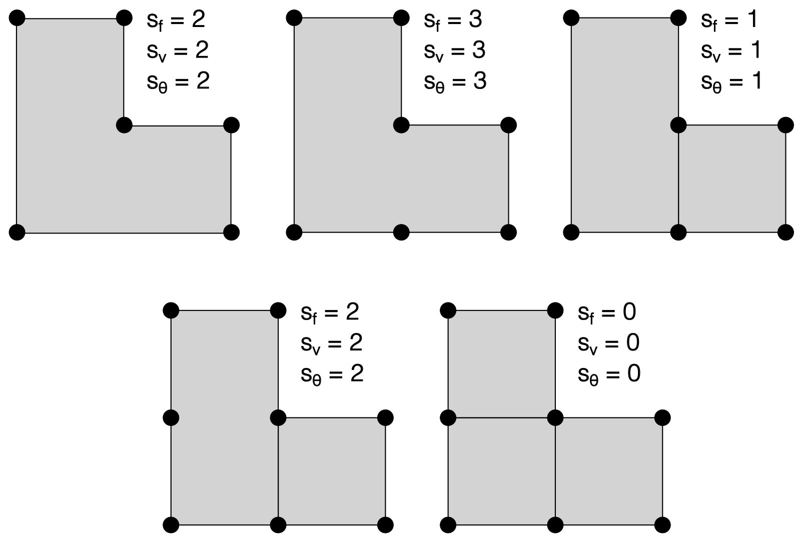


Figure 3.3: Sequence of mesh editing operations and the associated score changes for a simple L-shaped input geometry with $f^* = 4$.

element. For example, the primal mesh in fig. 3.4 (shown in black) is a regular triangular mesh of a hexagon. All faces are regular with degree 3 and all vertices are regular according to the heuristic eq. 3.6. The dual mesh (shown in red) is a regular mesh with *hexagonal* elements. According to our heuristic described earlier, the desired face degree of the dual mesh is equal to the desired interior vertex degree of the primal. (In the case of triangles, the desired vertex degree in the interior is 6; hence the desired face degree of the dual is 6 resulting in hexagons.) The desired vertex degrees and desired angle of the dual can then be computed as before.

Efficient updates to objective after actions

The topological edit operations described in sec. 3.2 are local. Each operation potentially changes the face degree and vertex degrees of only a few faces/vertices in the mesh. Although eqs. 3.1 and 3.2 are defined as a sum over all the faces/vertices in the mesh, since a given operation is local in nature, the score can be updated efficiently by only considering the few faces/vertices involved in the operation.

Figure 3.5 illustrates the change in face degree and vertex degree upon edge

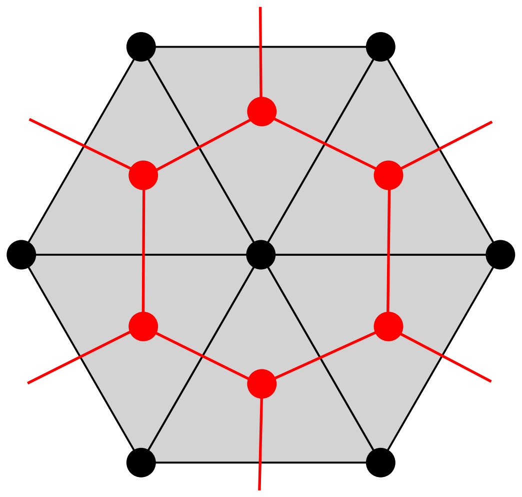


Figure 3.4: A primal mesh (black) and its corresponding dual (red). The dual is constructed by transforming faces in the primal mesh to vertices in the dual mesh, and vertices in the primal mesh to faces in the dual mesh. Edges are added between vertices in the dual mesh for every pair of adjacent faces in the primal mesh.

insertion. We see that edge insertion results in the creation of two new faces each of whose degrees can be inferred from the degree of the original face and the number of vertices between the two vertices participating in the edge insertion. Consider a face with degree f. We insert a new edge between a vertex j and a vertex $j+k$ which is k steps ahead in counter-clockwise order. This results in two new faces whose degrees are $k+1$ and $f-k+1$. Further, only the vertex degrees of the vertices participating in the edge insertion see their degree incremented by one. The face degree and vertex degree of the remaining faces/vertices in the mesh remain the same after edge insertion. Thus, the scores s_f and s_v may be updated efficiently and need not be recomputed globally.

Deleting an edge can be seen as the reverse of the above process. The two faces adjacent to the edge under question are merged after the delete. If the two adjacent faces have degrees f_1 and f_2, the degree of the new face after the delete operation will be $f_1 + f_2 - 2$. The vertex degree of the two vertices at the end-points of the deleted edge is decremented by one. The degrees of the remaining faces and vertices in the mesh remain the same.

Inserting a vertex on an edge increases the degree of the faces adjacent to the edge

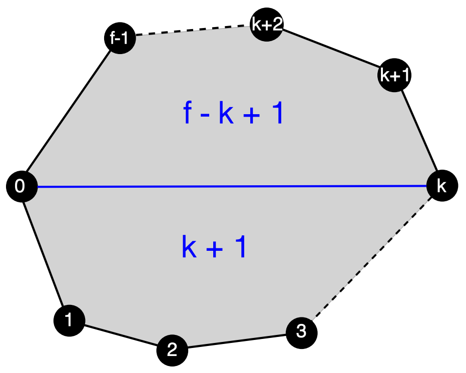


Figure 3.5: An edge is inserted between two vertices that are k steps apart in a face whose degree is initially f. This results in two new faces whose degrees are respectively $k+1$ and $f-k+1$. Further, the vertices labelled 0 and k see their respective degrees increase by one. The degrees of the remaining faces and vertices in the mesh do not change.

by 1. (If the edge is on a boundary, there is only one face adjacent to it. If it is in the interior, there are two faces adjacent to it.) The newly introduced vertex always has degree 2. The remaining faces and vertices in the mesh remain unaffected. Deleting a vertex reduces the degree of the faces adjacent to the edge by 1. Additionally, the contribution of the deleted vertex to the score s_v can be updated.

Based on the above discussion, we conclude that the discrete scores s_f and s_v can be updated efficiently in constant time after each mesh edit operation. In contrast, the continuous angle score s_θ cannot by updated in a similar way. As described in the section on Objective Function, the mesh typically needs to be smoothed after performing an edit operation to update the coordinates of vertices. This will impact the induced angles in the mesh and therefore the score s_θ. Clearly, performing a global smoothing of the mesh will incur high cost and scale with the size of the mesh. Alternatively, it is likely that smoothing only needs to be performed in the neighborhood where a topological edit was performed. A local smoother, whose action is restricted to a small neighborhood around the vertices involved in a topological edit operation, will be more computational efficient in this situation. The details of such a local smoother are beyond the scope of this dissertation.

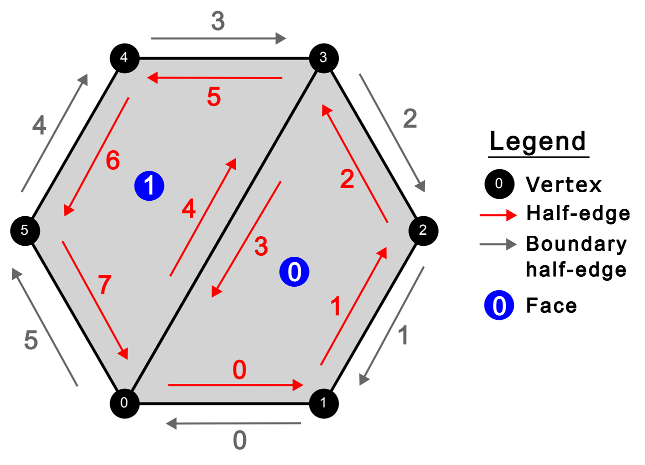


Figure 3.6: Polygonal meshes can be represented by the doubly-connected edge list (DCEL). Edges are represented as a pair of oriented half-edges shown as red arrows. (Boundary half-edges are shown in gray.) Half-edges are associated to their cyclic *next* and *previous* half-edges e.g. half-edge 2 is *next* of half-edge 1 and *previous* of half-edge 3. Half-edges are associated to the vertices at their *source* and *target* e.g. vertex 3 is the *source* of half-edge 5 and the *target* of half-edge 4. Half-edges are associated to the unique face that they belong to e.g. half-edges 4-5-6-7 belong to face 1.

3.3 Mesh Data Structures

Representing a mixed-element polygonal mesh

We represent a given polygonal mesh using the doubly-connected edge list (DCEL) also known as the half-edge data-structure. Briefly, the DCEL exploits the fact that each edge in a mesh is shared by exactly two faces (except on the boundary). The DCEL represents each edge in the mesh by a pair of oriented half-edges pointing in opposite directions. Every half-edge contains a pointer to the cyclic *next* half-edge in the same face and to its *twin* half-edge in the neighboring face. Half-edges belonging to a given face form a loop under the *next* operation.
Our half-edge data-type has six attributes:

- *next* – The cyclic next half-edge in the same face.
- *previous* – The cyclic previous half-edge in the same face.

- *twin* – In the interior of the geometry, this points to the half-edge in the adjacent face that is associated with the same edge. On the boundary of the geometry, this points to a *boundary* half-edge.
- *source* – The vertex at the origin of the half-edge.
- *target* – The vertex at the destination of the half-edge.
- *face* – The face that the half-edge belongs to.

See fig. 3.6 for an illustration of the above associations. The *twin* of half-edges on the boundary are defined as *boundary* half-edges which do not have an associated face – however, the *next*, *previous*, *source*, and *target* operations are well defined for these boundary half-edges. In the rest of this work, *half-edge* will always refer to an *interior* half-edge. *Boundary* half-edges will always be explicitly qualified.

The half-edge attributes enable efficient queries such as determining the vertices at the end-points of a given half-edge. To facilitate efficient queries in the opposite direction – i.e. given a face find all the half-edges associated with it – we construct an underlying graph data-structure. The nodes of this graph are the half-edges, vertices, and faces. We add an undirected edge from every half-edge to its associated vertices and face. This further enables efficient querying of the degrees of vertices and faces in the mesh. Further, there is a one-to-one correspondence between half-edges and angles in the mesh – e.g. every half-edge can be associated with the angle induced between itself and its previous half-edge. Therefore, the angle score eq. 3.4 can be computed by iterating over half-edges and summing up the irregularities of their associated angles.

The DCEL representation of polygonal meshes makes it possible to efficiently check in $\mathcal{O}(1)$ time if a particular topological edit operation described in sec. 3.2 can be performed on a given mesh. All of the operations described in sec. 3.2 can be implemented efficiently using the DCEL representation of polygonal meshes. In particular, the complexity of the edge-insertion and edge-deletion operations are $\mathcal{O}(f)$ where f is the degree of the face in which the edge is to be inserted or deleted. The complexity of vertex insertion and deletion is $\mathcal{O}(1)$. Further, the DCEL representation contains all the information required to efficiently update s_f and s_v after performing a topological edit operation.

Parameterizing mesh edit operations

We parameterize all the mesh edit operations described in sec. 3.2 in terms of half-edges.

- Edge insertion - given a half-edge h and an integer k, insert an edge between the source of half-edge h and the target of the half-edge that is k *next* steps ahead of h. Specifying edge insertion in terms of half-edges automatically determines the face in which the edge is to be inserted.
- Edge deletion - given a half-edge h, delete it (and its twin) and merge the neighboring faces.
- Vertex insertion - given a half-edge h, insert a new vertex v and additional half-edges as necessary. We set v as the new *target* vertex of h.
- Vertex deletion - given a half-edge h, delete the vertex at its *source*.

We emphasize that an operation is only performed so long as it meets the constraints described in sec. 3.2. In particular, for edge insertion, we require that $0 \leq k < f - 1$ where f is the degree of the associated face. This ensures that the newly created face has degree at least 2. These constraints may be modified according to target applications.

3.4 Formulation as a Deep Reinforcement Learning Problem

Recall that we have set up an optimization problem wherein the goal is to minimize a given objective function such as eq. 3.5 given an input geometry and f^* using a sequence of elementary operations described in sec. 3.2. We view this as a Markov Decision Process (MDP) where the state is the mesh at a given point in the edit procedure, and the actions are the edit operations described in sec. 3.2. Performing an action results in a state transition to a new mesh. In this section we describe the details of the reinforcement learning (RL) formulation of our problem.

Reward function

The face score s_f (eq. 3.1), vertex score s_v (eq. 3.2), and angle score s_θ (eq. 3.4) provide a measure of the total irregularity in a polygonal mesh. We wish to determine a finite sequence of topological edit operations that minimize a weighted sum of s_f, s_v, and s_θ (eq. 3.5). After every topological edit operation is executed, the scores are updated. We take the reward to be equal to the difference between the scores before and after an operation is performed normalized by the respective initial score:

$$\begin{aligned} r_f(t) &= \frac{s_f(t) - s_f(t-1)}{s_f^0} \\ r_v(t) &= \frac{s_v(t) - s_v(t-1)}{s_v^0} \\ r_\theta(t) &= \frac{s_\theta(t) - s_\theta(t-1)}{s_\theta^0} \end{aligned} \tag{3.7}$$

where the corresponding normalizing factor is taken to be $s^0 = \max(s(0), 1)$ to avoid divide by zero errors (it is possible that the initial mesh is already optimal.) The time step t is simply a counter for the topological edit operations – we increment t by one after every action. The overall reward for a particular time-step is then,

$$r(t) = w_f r_f(t) + w_v r_v(t) + w_\theta r_\theta(t) \tag{3.8}$$

The weights w_f, w_v, and w_θ are user parameters that represent the relative preference between face, vertex, and angle irregularities and is application dependent. We take $w_f + w_v + w_\theta = 1$ without loss of generality. The discounted cumulative reward (i.e. the discounted return) is computed as,

$$G(T) = \sum_{t=1}^{T} \gamma^{T-t} r(t) \tag{3.9}$$

with discount factor $0 < \gamma <= 1$. Since s_f^0, s_v^0, and s_θ^0 represent the maximum possible contribution to the reward due to improvement in face, vertex, and angle regularity, the cumulative return $G(T)$ is upper bounded i.e. $G(T) \leq 1$. Indeed, the size of the initial mesh could be significantly different, and may have higher or lower potential for reward purely due to its size. Scaling rewards by the initial irregularity measure ensures that cumulative returns are properly weighted and independent of problem size.

Local State Representation

Our meshes are dynamic data-structures that transform as edit operations are performed on them. As such, their overall size can in principle be unconstrained. However, an important observation is that in the vast majority of cases, determining the appropriate edit operation to perform in a particular region of a mesh largely depends on the local topology around the region of interest. This motivates the development of a local state representation by leveraging the operations available through the DCEL data-structure. This has the added benefit that the size of the state representation may be fixed instead of scaling with the size of the mesh. Additionally, action selection can be restricted to the fixed set of actions available within the neighborhood around the region of interest.

K-nearest neighbor template construction

Given a half-edge h^*, we use breadth-first-search (BFS) to determine the K-nearest neighbor (KNN) half-edges centered around h^* using the fundamental DCEL operations. During BFS, the DCEL operations are always executed in the order *next*, *previous*, and *twin* starting from h^* – this is only for reproducibility and not essential to our approach. The connectivity associated with these K half-edges represents the local topology around h^*. The KNN algorithm is outlined in alg. 2.

Algorithm 2 K-nearest neighbor half-edges.

```
KNN($h^*, k$)
    queue ← queue initialized with $h^*$
    neighbors ← empty list
    count ← 0

while queue is not empty and count < $k$ do
  $h$ ← remove half-edge from head of queue
  add $h$ to neighbors
  Increment count
  if count == $k$ then
    break
  end if
  Add next($h$) to queue if it is not visited
  Add previous($h$) to queue if it is not visited
  Add twin($h$) to queue if it is not visited
end while
return neighbors
```

To determine h^* we compute a score restricted to each half-edge which is the sum of the irregularities of its associated face, source vertex, and the induced angle at its source. h^* is then taken to be the half-edge with the maximum such score with

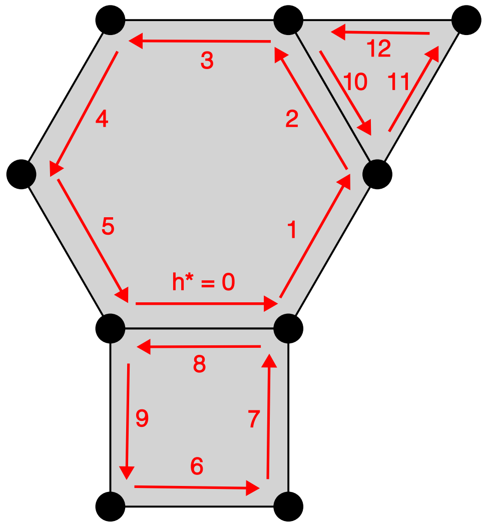


Figure 3.7: Illustrating the template construction centered around half-edge 0. With $K = 10$, the template would consist of the half-edges $[0, 1, 5, 8, 2, 4, 9, 7, 3, 10]$.

ties broken randomly. In our experiments, we determined h^* after each time-step by iterating over all half-edges in the mesh. Future implementations may use a priority queue to efficiently select h^* at each time step, where the priority value is the half-edge restricted score described above. The locality of the topological edit operations ensures that the priorities of only a few half-edges needs to be updated at every time step.

The KNN algorithm initialized from h^* produces a list of K half-edges which we refer to as the *K-template* centered at h^*. We also produce a reverse index using a hash table that maps half-edges to their index in the K-template. We use this reverse index to generate arrays containing the index in the template list of the *next*, *previous*, and *twin* half-edge for each half-edge in the template. There are two situations where we may not have a valid index. (1) the requested half-edge exists in the mesh but is outside the K-neighborhood or (2) the requested half-edge is a boundary half-edge. We use special indices – e.g. -1 and -2 – to denote these two situations.

Template Half-edge	0	1	5	8	2	4	9	7	3	10
Index	0	1	2	3	4	5	6	7	8	9

Template Half-edge	0	1	5	8	2	4	9	7	3	10
next Index	1	4	0	6	8	2	-1	3	5	-1
previous Index	2	0	5	7	1	8	3	-1	4	-1
twin Index	3	-2	-2	0	9	-2	-2	-2	-2	4

Table 3.1: Illustrating the template construction process for the example geometry shown in figure 3.7. Note that "*next* Index" refers to the *index* in the template list of the cyclic *next* half-edge and similarly for *previous* and *twin*. We use the special index -1 to denote a half-edge that exists in the geometry but is outside the template and the index -2 to denote a half-edge that is outside the geometry (i.e. a boundary half-edge).

Half-edge features

We identify the following as the relevant topological and geometric features associated with each half-edge (see fig. 3.8 for an illustration):

- The degree of its associated face
- The degree of its source vertex
- The desired degree of its source vertex
- The length of the half-edge
- The induced angle at its source

Topological features include vertex degree, vertex desired degree, and face degree. We do not include the face desired degree and the desired angle as half-edge features since we take these to be constant over the entire mesh. Geometric features include length and angle. Geometric features only need to be included if the objective score includes a component dependent on geometry such as s_θ.

We use the above information to construct a feature vector for each half-edge. Notice that while f^* and v_j^* are usually bounded and relatively small, f_i and v_j can be unbounded in principle. Thus we threshold the face and vertex degrees before including it in the feature vector as $\min(f_i, f_{\max})$ and $\min(v_j, v_{\max})$. This thresholding is essential for the generalization performance of our model. Intuitively, when the

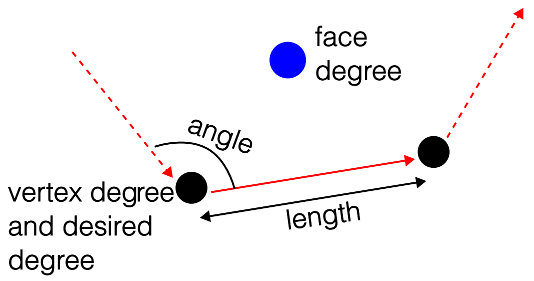


Figure 3.8: Relevant topological and geometric features associated with each half-edge. Topological features include the vertex degree, vertex desired degree, and face degree. Geometric features include length and angle.

face degree or the vertex degree is large enough, the optimal edit is likely going to be the same regardless of the exact value of the degree. The local state of the K-template may now be represented by,

- $F \in \mathbb{R}^{K \times 5}$ – half-edge feature matrix composed of the feature vectors of the corresponding half-edges in the template.
- $N \in \mathbb{Z}^K$ – index vector containing the index of the *next* half-edge for each half-edge in the template.
- $P \in \mathbb{Z}^K$ – index vector containing the index of the *previous* half-edge for each half-edge in the template.
- $T \in \mathbb{Z}^K$ – index vector containing the index of the *twin* half-edge for each half-edge in the template.

Note that the indices in N, P, T are the *local* template indices of the half-edges (see fig. 3.7 and table 3.1 for an illustrative example.)

DCEL convolution operation

We design a neural network whose fundamental operations is a convolution on the half-edge feature matrix utilizing the connectivity information in the N, P, T index vectors. Recall that the index vectors contain special indices to handle two boundary cases as described in sec. 3.4. Therefore, we first augment the half-edge feature matrix by appending two learnable vectors to the end of the feature matrix F that correspond to the two boundary cases. This acts as an effective signal to the model

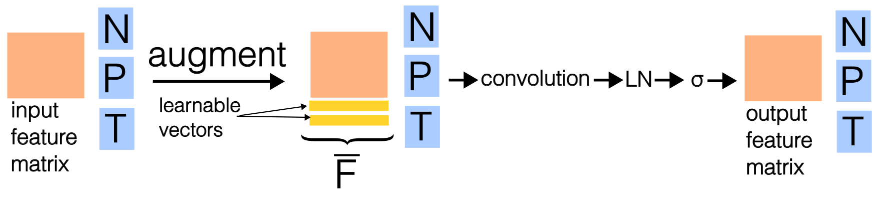


Figure 3.9: Illustration of the DCEL convolution operation. The input feature matrix is first augmented by appending two learnable vectors at the end. The augmented matrix $\bar{F}$ undergoes the convolution operation described in eq. 3.10 to produce the output. LN is layer normalization and σ is a non-linear activation function. Note that the index vectors N, P, T are only used to facilitate the convolution. LN and σ are only applied to the feature matrix.

regarding the geometric- and template- boundaries of the local template neighborhood. The special indices used in template construction are updated to point to the two learnable vectors. (In a language that uses cyclic indexing – e.g. Python – we do not need to update the special indices since -1 and -2 point to the last and second-to-last element in an array. Therefore, the learnable vectors will be indexed correctly if they are appended to the *end* of the feature matrix.)

Suppose F is the input to the DCEL convolution, we denote the augmented feature matrix by $\bar{F}$. The DCEL convolution operation can be written as,

$$\text{conv}(F) = \sigma(\text{norm}(A\bar{F} + B\bar{F}[N] + C\bar{F}[P] + D\bar{F}[T])) \tag{3.10}$$

Where norm represents layer normalization, σ is a non-linear activation function, and $A, B, C,$ and D are learnable matrices. We illustrate the above operation in fig. 3.9.

The basic unit of our neural network architecture consists of two convolution operations followed by a skip (i.e. residual) connection. Residual connections are widely used in modern neural network architectures to handle the problem of vanishing gradients [30]. We refer to this trainable unit as a DCEL residual block. (See fig. 3.10 for an illustration.) Suppose F_{l-1} are the output features of a particular block, the output of the next block is obtained as

$$F_l = \text{conv}(\text{conv}(F_{l-1})) + F_{l-1} \tag{3.11}$$

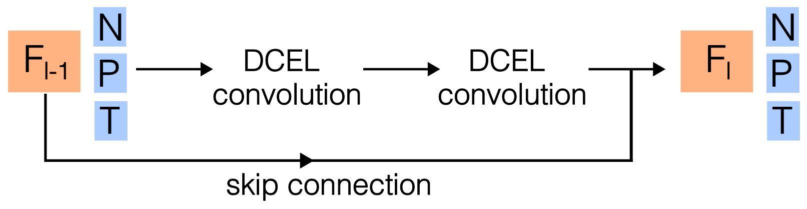


Figure 3.10: Illustration of a DCEL residual block which consists of two DCEL convolution operations followed by a skip connection.

Our *feature extractor* consists of a stack of L residual blocks. The output of the feature extractor is an encoding of the local geometry and topology around each half-edge in the K-template. Notice that the encoding contains geometric and topological information from a local neighborhood whose size grows linearly with the number of convolution operations. The "field-of-view" can at most be of size K since the template does not contain any information outside of this neighborhood. We point out that because we adopt a local state representation, our problem is more precisely categorized as a Partially Observable Markov Decision Process (POMDP).

The feature matrix to the first residual block is obtained by projecting the half-edge feature matrix F to a high dimensional embedding space $F_0 \in \mathbb{R}^{K\times d}$ using a single linear layer. The embedding dimension d is kept fixed across blocks to facilitate the skip connection. Thus, the complexity of our neural network is parameterized by specifying d and L. We use the LeakyReLU activation function for σ.

Actor-critic neural network architecture

The DCEL convolutional neural network described in the previous section is used to extract high dimensional features F_L that encode the local topological state around each half-edge in the template. This network is shared by the actor and critic. After feature extraction, the network is split into an actor head and a critic head.

The actor head consists of a simple multi-layer perceptron (MLP) with a single hidden layer that projects F_L to $\Pi \in \mathbb{R}^{K\times A}$ where A is the number of actions per half-edge. Π represents the log probability of action selection by the agent. We mask invalid actions in Π by adding a mask matrix M to it. The entries in M are 0 for valid actions and $-\infty$ for invalid actions. This ensures that, after applying the softmax function, the probability of selecting an invalid action is zero. The final matrix Π is flattened into a vector and passed through a softmax layer to obtain action probabilities i.e. a policy.

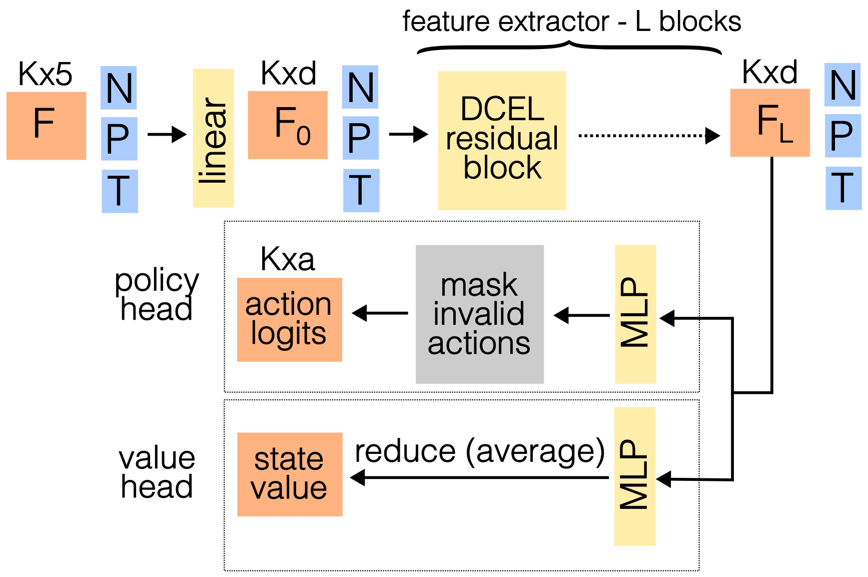


Figure 3.11: Overall architecture of our neural network. The half-edge feature matrix F is first projected to F_0 in a d dimensional embedding space. These features are processed by L DCEL residual blocks. The final feature matrix F_L is shared by the actor head and critic head which consist of perceptrons with a single hidden layer. The policy head masks invalid actions with the mask matrix M to get the action logits which represent the log probability of action selection. The value head performs an average reduction operation to get a single number representing the critics estimate of the state value.

The critic network has similar structure, however since we are interested in obtaining a single *state-value* for a given state as a whole and not for each half-edge, we perform a reduction operation across the template to get a single vector for each template. This vector is passed through a linear layer to obtain the critic's estimate of the state-value. We use the mean across the template dimension as the reduction operation. The overall architecture is illustrated in fig. 3.11.

Parameter	**Symbol**	**Value**
Polygon degree range	-	$[6, \ldots, 20]$
Radius range	-	$[0.2, 1.0]$
Local template size	K	20
Edge addition steps range	k	$[0, \ldots, 4]$
Max. steps factor	-	2
Face reward weight	w_f	0.5
Vertex reward weight	w_v	0.25
Angle reward weight	w_θ	0.25
Face degree threshold	f_{max}	15
Vertex degree threshold	v_{max}	8

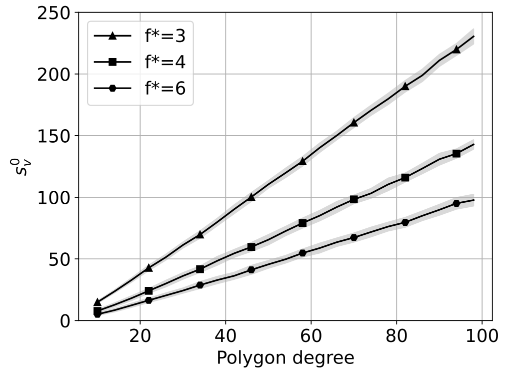
Table 3.2: Parameters used to specify the random polygon environment.

3.5 Training the Agent in Self-play Reinforcement Learning

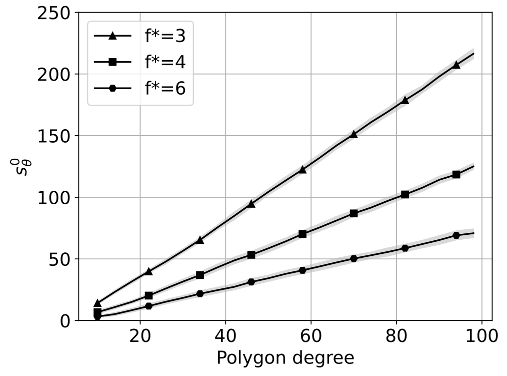
Training environment

We trained our policy on randomly generated polygonal shapes of varying sizes for a given desired face degree f^*. For each instantiation of the environment, we determine the initial polygon degree by sampling uniformly from a prescribed range. The vertex coordinates are determined in polar coordinates – with the radii being uniformly sampled from a given range and equal angular increments between successive points in the face-loop. The desired vertex degrees and desired angles on the boundary of the initial geometry are determined using the heuristics 3.2. The environment is terminated after a finite number of steps is taken by the agent or if the agent achieves a score of zero. Table 3.2 collects the parameters used to specify the random polygon environment.

The initial face score s_f^0 (eq. 3.1) scales linearly with the face degree of the initial geometry. The scaling of the initial vertex score s_v^0 and angle score s_θ^0 are not immediately obvious. However, fig. 3.12 illustrates that the scaling of these scores are also linear for $f^* \in \{3, 4, 6\}$. This observation, combined with the fact that the reward for any particular action is independent of the size of the mesh motivates



(a)



(b)

Figure 3.12: Scaling of initial scores with respect to polygon degree for randomly generated polygons with different desired face degrees. (a) Initial vertex score s_v^0 (b) Initial angle score s_θ^0. Scores measured over 50 trials per polygon degree. Solid line represents the mean and shaded region the 1-standard deviation envelope. Plots demonstrate that the relationship is effectively linear. For the initial boundary, the only difference between s_v^0 and s_θ^0 is the rounding operator in eq. 3.6.

us to set the maximum number of steps before terminating the environment to be proportional to the initial size of the input polygon. (This is reported in table 3.2 as Max. steps factor.) We highlight that the only difference between s_v^0 and s_θ^0 for such randomly generated shapes is the effect of the rounding operation in eq. 3.6. Because vertex irregularity and angle irregularity measure similar metrics in slightly different ways, we weigh the vertex reward and angle reward equally, and weigh their sum and the face reward equally. This is the motivation for setting the values of w_f, w_v, w_θ in table 3.2. "Edge addition steps range" in table 3.2 refers to the valid choices of the parameter k discussed in sec. 3.3 which specifies the distance between the two vertices participating in an edge insertion.

The environment is implemented as a sub-class of the `gym` environment from the Gymnasium library [72]. Gymnasium is widely used in the reinforcement learning (RL) literature and provides a consistent Application Programming Interface (API) for RL environments. This allows easy interoperability with existing RL libraries.

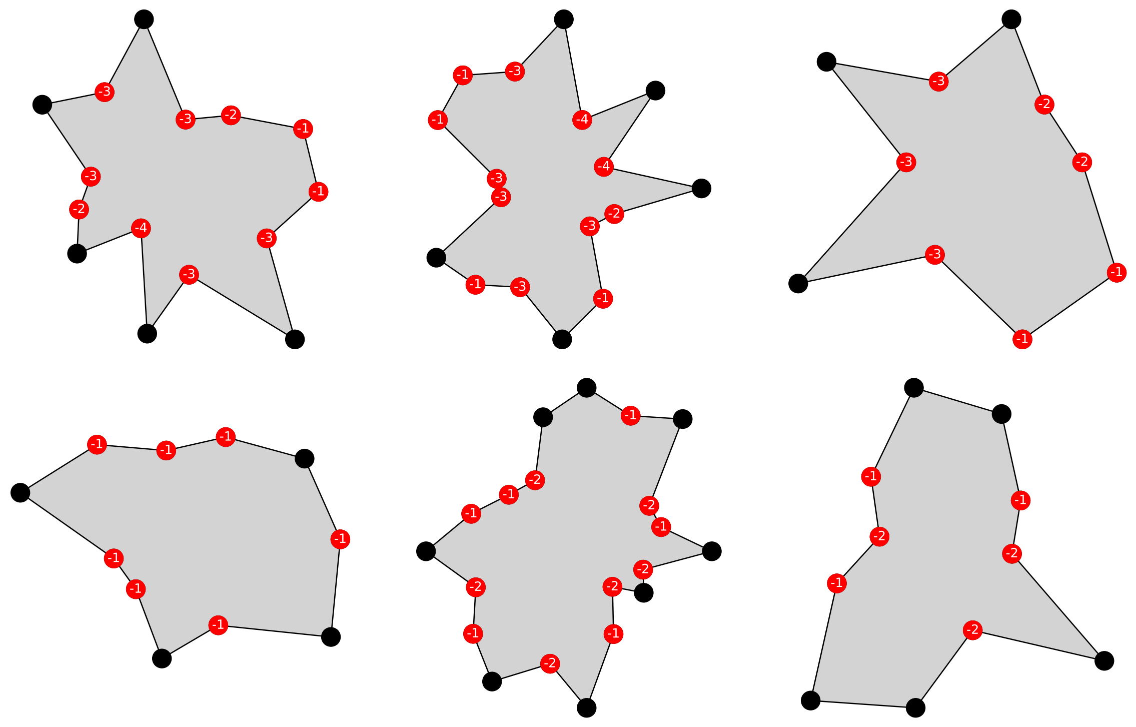


Figure 3.13: Illustrative examples of randomly generated initial polygonal shapes. Top row is for $f^* = 3$ and bottom row is for $f^* = 4$. Vertices with non-zero vertex irregularity are marked.

Training algorithm

We train our agent using the Proximal Policy Optimization (PPO) [60] algorithm. We utilize the PPO implementation available through the stable-baselines3 library [54]. PPO is one of the most widely used deep-reinforcement learning algorithms. It falls within the family of policy-gradient algorithms [67] with additional constraints that prevent large changes in the policy distribution during training. This ensures training stability and uniform policy improvement over the course of training. Our neural network architecture illustrated in fig. 3.11 is well designed to fit within the paradigm of Actor-Critic PPO. We will not review the PPO method here and instead direct the interested reader to the original article on the subject [60].

Automatic hyperparameter optimization

As with most machine learning methods, the performance of models is sensitive to a number of non-trainable parameters (i.e. hyperparameters). This is particularly true with reinforcement learning algorithms [25]. The first-cut approach to hyperparameter optimization (HPO) is often hand-tuning based on trial and error, domain knowledge, and experience. However, this can be time-consuming and ineffective. An alternative is to utilize grid-search which exhaustively searches the hyperparameter space to determine optimal values. When the design space is large, grid-search can be computationally expensive. Recently, mature libraries have emerged for automatic hyperparameter optimization (HPO) e.g. Ray Tune [41] and Optuna [2]. In this dissertation, we leverage Optuna for automatic HPO.

We use the Tree-structured Parzen Estimator (TPE) algorithm [7]. Briefly, the TPE algorithm first runs several trials by random sampling from the design space of hyperparameters. The cumulative return at the end of training is taken as the objective function for a given trial. The algorithm then splits the trials into two sets – top-performing trials (typically the top 10-20% best performing trials) and the remaining trials. The core idea of the TPE algorithm is to draw new samples from the hyperparameter design space that are likely to be in the top-performing set. The algorithm achieves this by building a surrogate function that models the likelihood of a given point in design space belonging to each group. Consider that x is a point in design space, $l(x)$ is the likelihood of this point producing a model that is in the top 20% and $g(x)$ is the probability of this point producing a model that is in the bottom 80%. (The likelihood functions are constructed using Parzen Window Estimation – hence the name of the algorithm.) The TPE algorithm selects the next trial point to be the one that maximizes the expected improvement $E(x) = l(x)/g(x)$.

Table 3.3 describes the various hyperparameters that were included in the design study, along with the prior sampling ranges. These ranges were determined based on a combination of literature values (e.g. Schulman et al. [60] use $\epsilon = 0.20$ and $\lambda = 0.95$) and trial-and-error for our specific problem.

Table 3.4 reports the hyperparameters with best average score after running 100 trials of hyperparameter optimization for $f^* = 4$ and $f^* = 3$. Each trial consisted of sampling a set of hyperparameters according to the TPE strategy and training an agent for 10^6 steps.

Table 3.5 lists hyperparameters that were kept fixed over all trials. The discount rate γ is a measure of the trade-off between immediate reward and future reward. We set the discount rate $\gamma = 1$ since we are concerned with optimizing the final objective value and are not concerned with short-term reward. The number of environment steps collected in rollout should be set to collect enough samples from the rollout

Parameter	**Symbol**	**Discrete/ Continuous**	**Range**	**Sampling Domain**
Learning rate	η	continuous	$[10^{-6}, 10^{-3}]$	log
Generalized advantage estimator (GAE) coefficient	λ	continuous	$[0.5, 0.999]$	linear
Entropy loss coefficient	c_e	continuous	$[10^{-8}, 0.5]$	log
Value loss coefficient	c_v	continuous	$[10^{-8}, 0.5]$	log
PPO clip parameter	ϵ	continuous	$[10^{-2}, 0.2]$	linear
Max. gradient norm	-	continuous	$[0.1, 5.0]$	linear
Orthogonal initialization of network weights	-	discrete	True/ False	-
Number of DCEL residual blocks	L	discrete	$[2, \ldots 10]$	-
Embedding dimension	d	discrete	$\{64, 128, 256, 512, 1024\}$	-

Table 3.3: List of parameters and their corresponding prior range used in hyperparameter optimization. We use the same design space for all our experiments.

phase so that the gradient estimates during policy improvement are sufficiently accurate. This value will vary based on the complexity of the environment and we determined it based on trial-and-error. The batch-size should be sufficiently large for meaningful gradient estimates but small enough to include some stochasticity in gradient descent in order to avoid local minima. GPU memory was not a constraint for our problem, but it may be part of the decision making for environments with larger K-templates in state-representation.

Analysis of hyperparameter optimization for $f^* = 4$ and $f^* = 3$

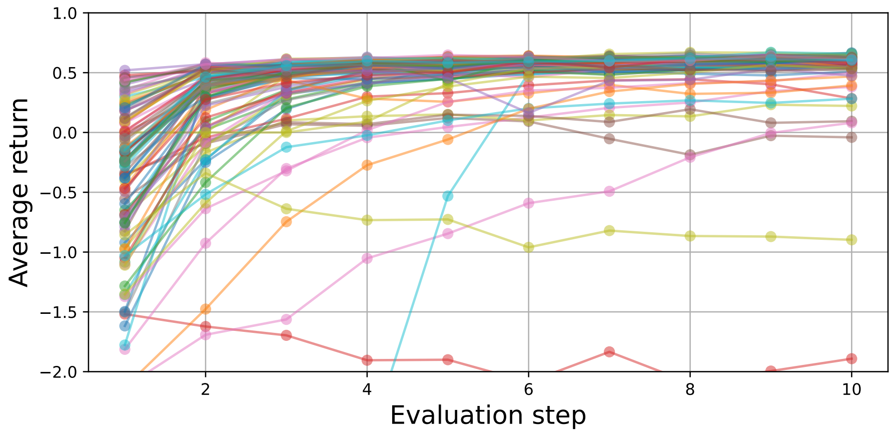
The training trajectories of 100 optimization trials are reported in fig. 3.14 for $f^* = 4$ and $f^* = 3$. The trajectories were generated by evaluating models periodically during training by measuring the average returns from 200 random instances of the training environment. The trajectories indicate that the vast majority of trials have converged upon their optimal values with each trajectory corresponding to a different

Parameter	**Symbol**	**Optimized value $f^* = 4$**	**Optimized value $f^* = 3$**
Learning rate	η	1.3×10^{-4}	5.9×10^{-4}
Generalized advantage estimator (GAE) coefficient	λ	0.86	0.86
Entropy loss coefficient	c_e	2.6×10^{-8}	4×10^{-5}
Value loss coefficient	c_v	0.149	0.05
PPO clip parameter	ϵ	0.189	0.15
Max. gradient norm	-	1.3	2.55
Orthogonal initialization of network weights	-	False	True
Number of DCEL residual blocks	L	5	3
Embedding dimension	d	128	128

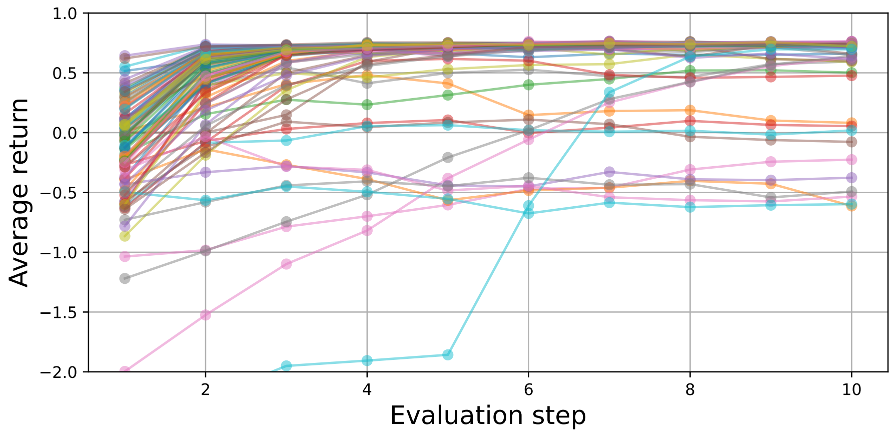
Table 3.4: Results of hyperparameter optimization for $f^* = 4$ and $f^* = 3$. Optimal values were obtained by running hyperparameter optimization over 100 trials. Each run consisted of training the agent for 1M time-steps.

Parameter	**Symbol**	**Value**
Discount rate	γ	1
Number of steps collected in rollout	-	10,240
Batch size	B	512

Table 3.5: Hyperparameters that were taken constant for all trials.



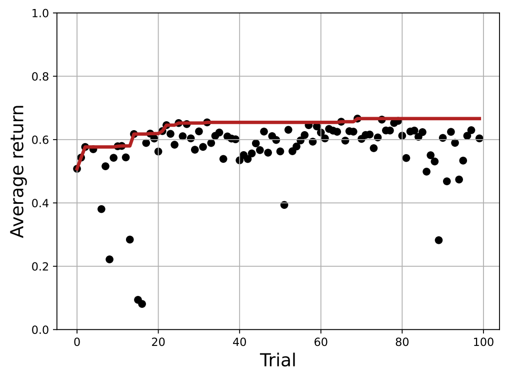
(a) $f^* = 4$



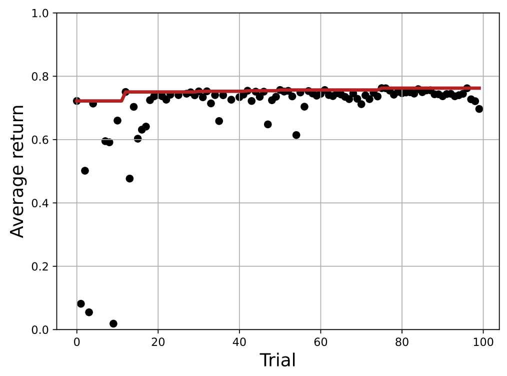
(b) $f^* = 3$

Figure 3.14: Training trajectories of 100 optimization trials for (a) $f^* = 4$ and (b) $f^* = 3$. Trajectories are generated by periodically measuring the average return during training. Returns are averaged across 200 randomly generated environments for each evaluation step. The training trajectories suggest that most models have converged upon their optimal values.

choice of hyperparameters. The final objective scores and best score over all trials is shown in fig. 3.15. The range of converged values demonstrates that choosing the right hyperparameters can have a significant impact on the final performance of the model.



(a) $f^* = 4$



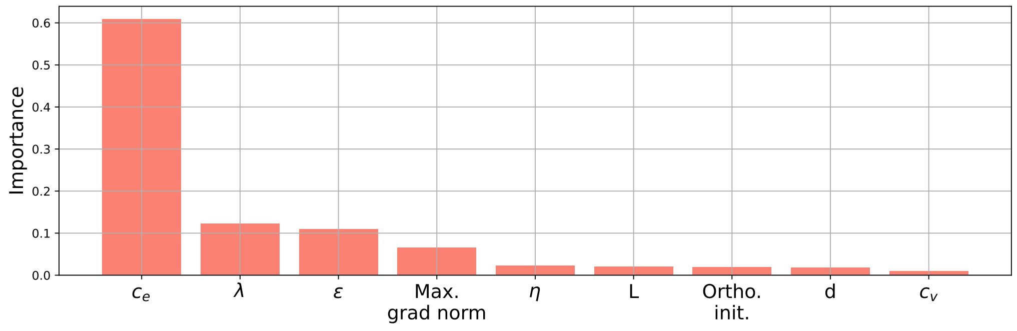
(b) $f^* = 3$

Figure 3.15: Optimization history for (a) $f^* = 4$ and (b) $f^* = 3$. Each point represents the final objective value of a particular trial. The red line represents the best objective score seen so far.

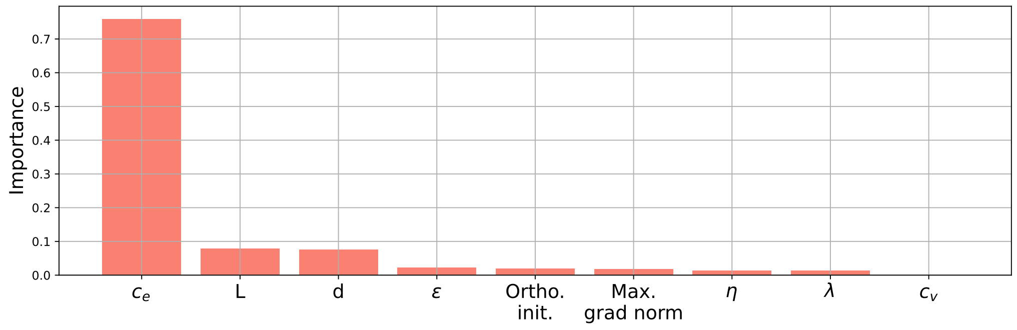
While we allow for the optimization of several parameters, a few parameters have an outsized impact on the objective function. The importance of hyperparameters may be estimated using the method outlined in Hutter et al. [32]. Briefly, the method works by fitting a random forest regression model to the logged optimization data. The random forest model is trained to predict model objective value from a given hyperparameter configuration. Marginal predictions are then extracted from this random forest and the importance of hyperparameters is estimated using the functional Analysis of Variance (fANOVA) method. (The Optuna library provides functionality to extract this data from the logged optimization data.)

Figure 3.16 shows the relative importance of various hyperparameters involved in the optimization procedure. The results underscore the fact that a few hyperparameters have an outsized impact on the final objective score. Further, we observe that the relative importance of hyperparameters depends on the choice of f^*. For $f^* = 4$ we see that the entropy coefficient c_e, the generalized advantage estimation coefficient λ, and the PPO clip coefficient ϵ are among the most important parameters. Whereas for $f^* = 3$ the top-3 hyperparameters in terms of importance are c_e, neural network depth L, and neural network embedding dimension d. An important aspect to keep in mind is that the relative importance of hyperparameters is also influenced by the range of the corresponding hyperparameter in the search space defined in table 3.3 and may not remain the same for a different choice of design space.

We can visualize the optimization landscape for the top-3 most important hyper-



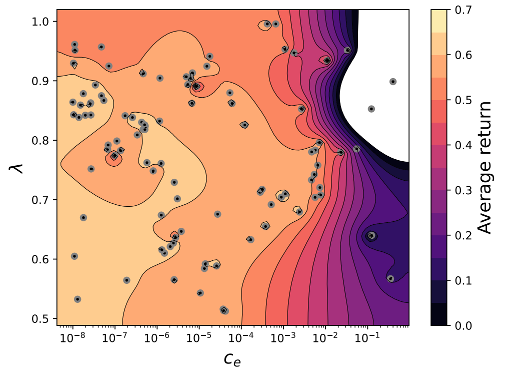
(a) $f^* = 4$



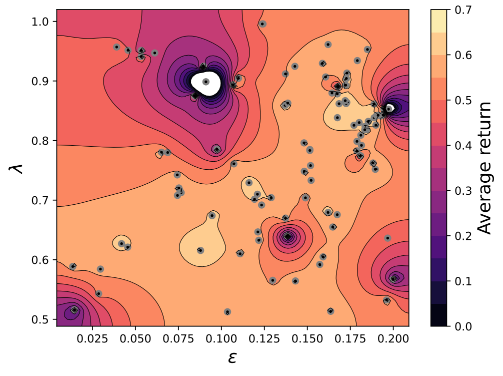
(b) $f^* = 3$

Figure 3.16: Hyperparameter importances for (a) $f^* = 4$ and (b) $f^* = 3$ as measured by the functional Analysis of Variance (fANOVA) method [32]. The results indicate that a few parameters have an outsized impact on the final objective function. Further, we see that the relative importance of hyperaparameters depends on the choice of f^*.

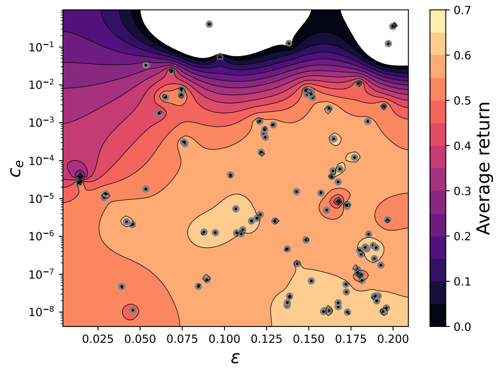
parameters via contour plots of the objective value as a function of the hyperparameters taken pairwise. Figures 3.17 ($f^* = 4$) and 3.18 ($f^* = 3$) show the optimization landscape for the two element types being assessed in this work. The plots show that the optimization landscape is highly non-linear with several local minima/maxima. We also see that the TPE sampling strategy results in non-uniform sampling in the design space and tries to place more sample points in regions with higher potential for reward.



(a) GAE λ vs Entropy coefficient c_e

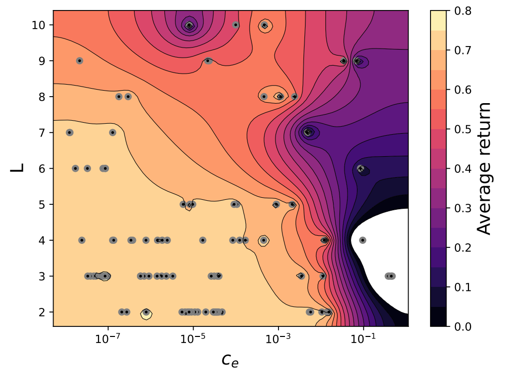


(b) GAE λ vs PPO clip parameter ϵ

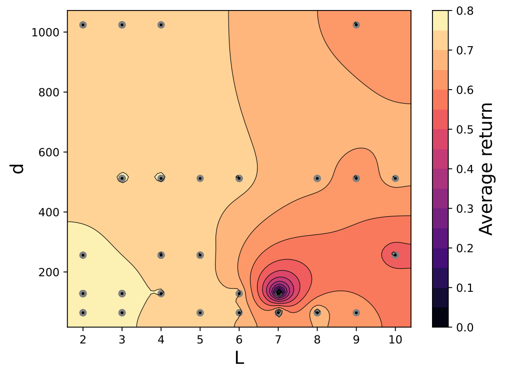


(c) Entropy coefficient c_e vs PPO clip parameter ϵ

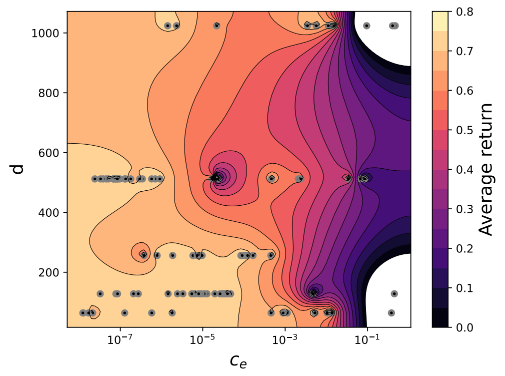
Figure 3.17: Optimization landscape of the three most important hyperparameters for $f^* = 4$ – namely entropy coefficient c_e, GAE coefficient λ, PPO clip parameter ϵ. Note that c_e is plotted in the log-domain while λ and ϵ are plotted in the linear domain. Scatter points represent trial points. We highlight the highly non-linear optimization landscape and the non-uniform sampling strategy of the TPE algorithm that aims to explore regions with high objective value more than regions with low objective value.



(a) DCEL residual blocks L vs. entropy coefficient c_e

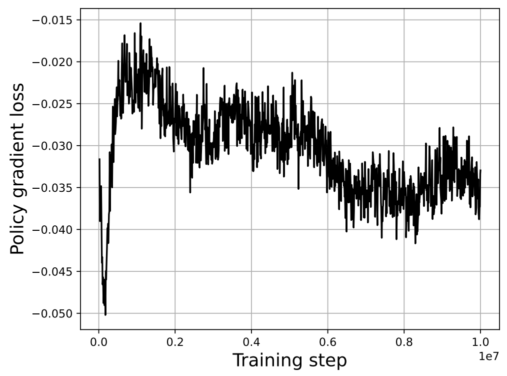


(b) Embedding dimension d vs. DCEL residual blocks L

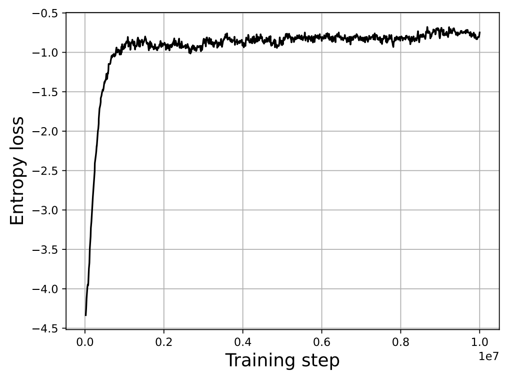


(c) Embedding dimension d vs entropy coefficient c_e

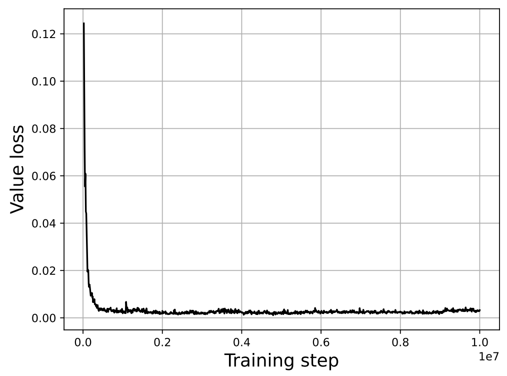
Figure 3.18: Optimization landscape of the three most important hyperparameters for $f^* = 3$ – namely entropy coefficient c_e, number of DCEL residual blocks L, and embedding dimension d. Note that c_e is plotted in the log-domain while L and d take on discrete values. Scatter points represent trial points. We highlight the highly non-linear optimization landscape and the non-uniform sampling strategy of the TPE algorithm that aims to explore regions with high objective value more than regions with low objective value.



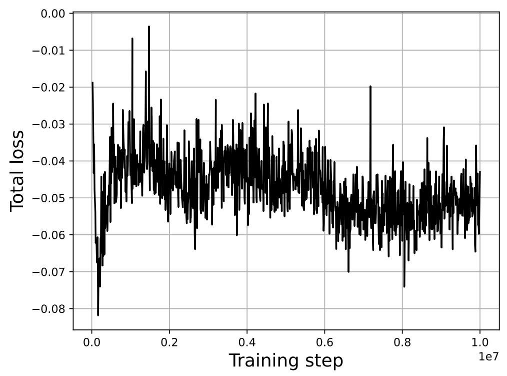
(a) PPO policy gradient loss



(b) Entropy loss



(c) Value loss

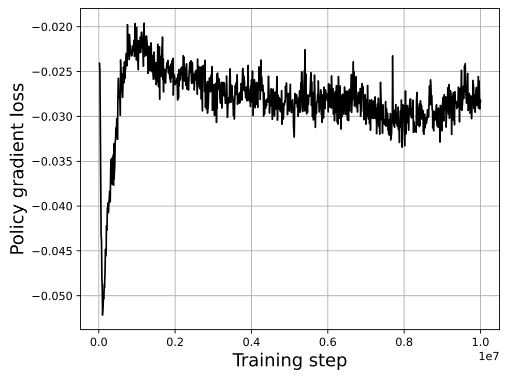


(d) Total loss

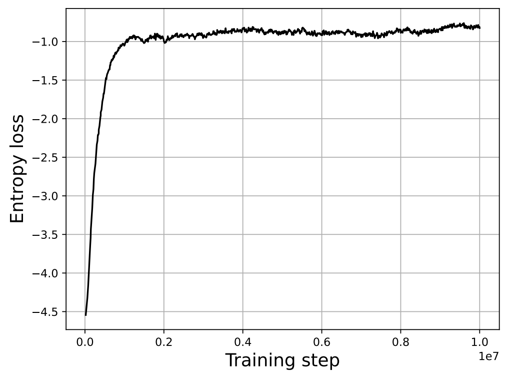
Figure 3.19: Training loss history of best-performing agent for $f^* = 4$. Agent was trained for 10^7 time-steps.

Fine-tuning best-performing model

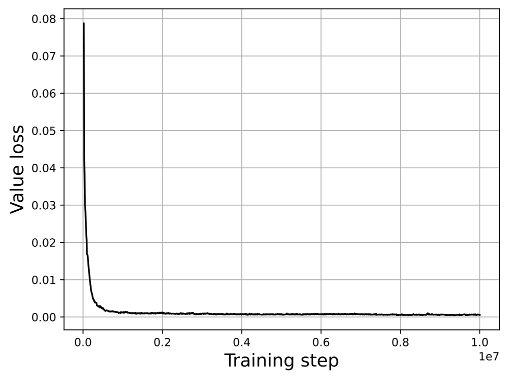
The hyperparameters corresponding to the best-performing model configuration during hyperparameter optimization is reported in table 3.4. We select these parameters and train f^* specialized agents in self-play for 10^7 time-steps. Figures 3.19 and 3.20 show the training loss history of various loss components. Figures 3.21 and 3.22 show the performance of the agents during the course of training. We see that the explained variance is around 0.95 for $f^* = 4$ and 1 for $f^* = 3$. Explained variance measures the proportion of the variance observed in state value that can be captured by the value function head and is upper bounded at 1. The relatively high values of explained variance for both quadrilaterals and triangles suggests that the feature



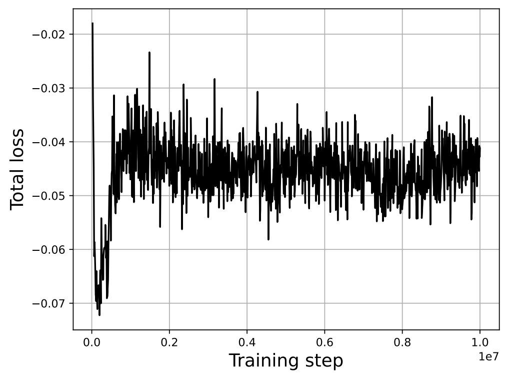
(a) PPO policy gradient loss



(b) Entropy loss



(c) Value loss



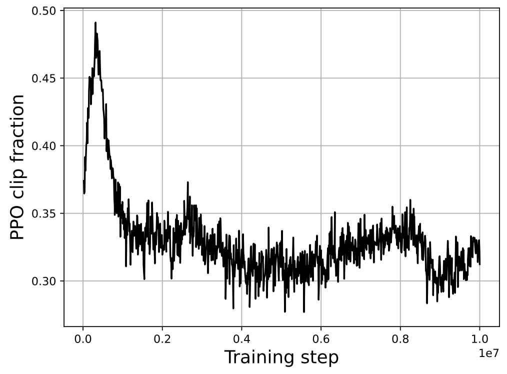
(d) Total loss

Figure 3.20: Training loss history of best-performing agent for $f^* = 3$. Agent was trained for 10^7 time-steps.

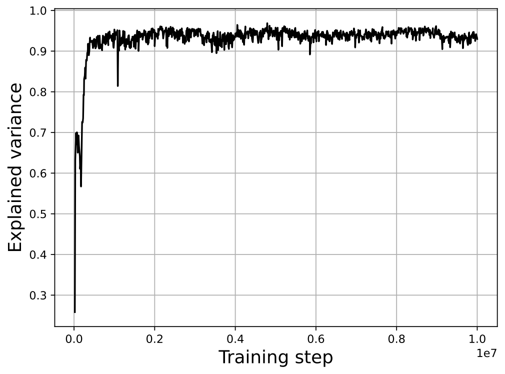
extractor and value function head are able to capture the underlying dynamics of the environment. We also note that the clip fraction is initially high as the policy undergoes rapid change during optimization, but gradually reduces as the model stabilizes. Finally the average returns measured over the course of training increases rapidly and slowly stabilizes around 0.7 for $f^* = 4$ and 0.78 for $f^* = 3$.

3.6 Evaluation

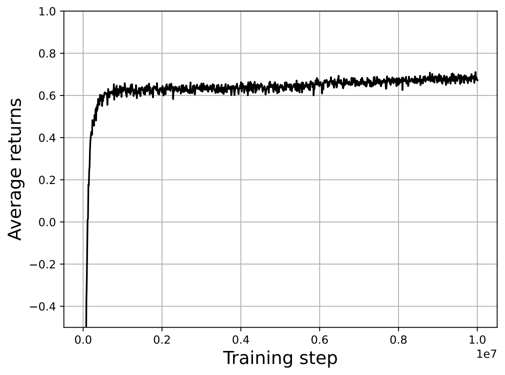
Figure 3.23 shows the average best score vs polygon degree for the quadrilateral ($f^* = 4$) and triangle ($f^* = 3$) model. For a given polygon degree, we instantiate 10 random



(a) PPO clip fraction

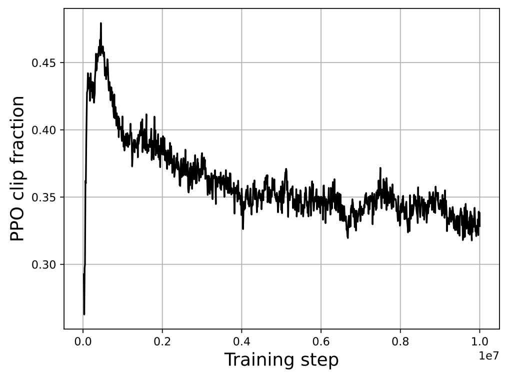


(b) Explained variance

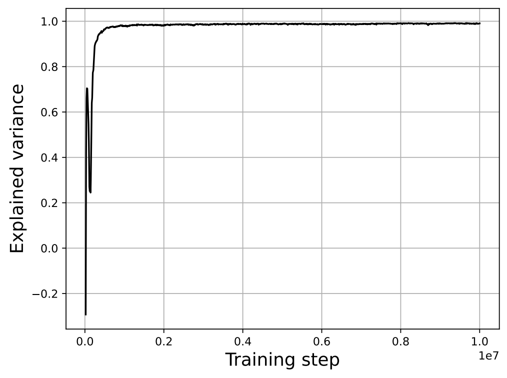


(c) Average returns

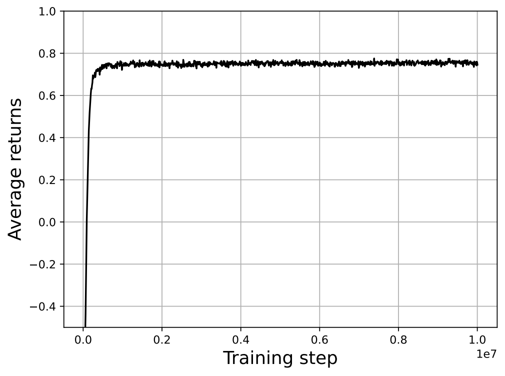
Figure 3.21: Model performance as a function of training history for $f^* = 4$. (a) Ratio of actions whose gradients are clipped according to the PPO clipped objective function. Initially, the policy undergoes rapid changes resulting in a higher fraction of actions experience clipping. As training stabilizes, a fewer fraction of actions experience gradient clipping. (b) The explained variance of the value function head is close to 1 suggesting that the model is successfully capturing the underlying environment dynamics. (c) The average returns rapidly increases and then slowly approaches a value of around 0.7 at 10^7 steps. Further training may improve the performance.



(a) PPO clip fraction

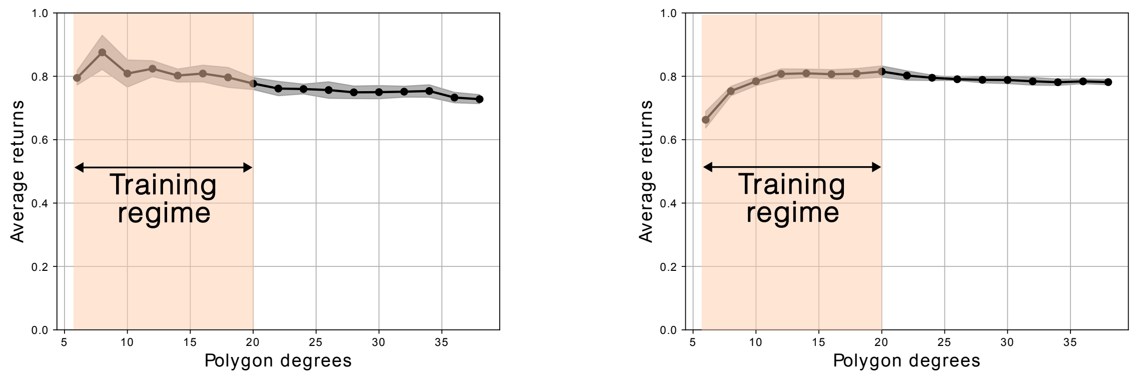


(b) Explained variance



(c) Average returns

Figure 3.22: Model performance as a function of training history for $f^* = 3$. (a) Ratio of actions whose gradients are clipped according to the PPO clipped objective function. Initially, the policy undergoes rapid changes resulting in a higher fraction of actions experience clipping. As training stabilizes, a fewer fraction of actions experience gradient clipping. (b) The explained variance of the value function head is close to 1 suggesting that the model is successfully capturing the underlying environment dynamics. (c) The average returns rapidly increases and then slowly approaches a value of around 0.78 at 10^7 steps.



(a) $f^* = 4$ (b) $f^* = 3$

Figure 3.23: Average best score vs polygon degree for quadrilateral and triangular element types. The model was trained on random polygons of degree between 6-20 and evaluated on geometries with degree 6-40. For a given instance of an environment, the policy is rolled out 10 times from the same initial state and the best score is selected. We compute the average (solid line) and standard deviation (shaded region) over 10 random instances for a given polygon degree and repeat for various values of polygon degree. We observe that the model demonstrates generalization capability beyond the training regime. The quadrilateral model demonstrates some deterioration in model performance with increasing polygon degree. The drop in performance is lesser for the triangular model.

environments. The policy is repeatedly rolled out 10 times for each environment instance and the best score for each instance is computed. The average across the 10 instances and standard deviation are computed for each polygon degree. While the model was trained on 6-20 sided polygons, it demonstrates almost similar levels of performance on polygons up to 40-sided. The quadrilateral model demonstrates some deterioration in performance. The deterioration is smaller for the triangular agent.

We believe that the key reason for the generalization capability is our use of a local state representation that looks at a fixed sized neighborhood. We also use trainable embeddings to represent geometric and template boundaries – we can see these embeddings as representing a“belief” about the rest of the state. The generalization trend relies on the fact that actions are local and only influence the topology in a small neighborhood of the current template center.

Figures 3.24 and 3.25 show illustrative examples of the result of evaluating the policy on a given geometry. The initial geometry is shown on the left and is a random instantiation of the training environment. The policy is rolled out 10 times from each initial state. The mesh with the smallest face score with ties broken by smallest total score is shown on the right. We see that the model is in general capable of decomposing the given shape into elements of the right type. Further, we achieve a block mesh with few irregularities and angles that are not very skewed.

The performance of the agent is not optimal, however. For example, in a quadrilateral mesh, any edge between vertices that have irregularities +1 and -1 at its end-points can be regularized using the "global-split" operation discussed in chapter 2. We see several examples of such vertices in fig. 3.24. In the case of triangular elements in fig. 3.25 we notice that the agent never introduces a new vertex in the interior. In many cases (e.g. triangulating a hexagon) it is necessary to introduce interior vertices in order to obtain an optimal mesh. It is possible that the model has converged to a local minima due to lack of exploration. Further work is required to overcome this particular challenge.

3.7 Conclusion

In this chapter we presented a reinforcement learning method to generate meshes of user provided 2D shapes. We discussed in detail how we use the Doubly Connected Edge List (DCEL) to represent 2D geometries as collections of heterogeneous faces (elements). We illustrated a unified action space consisting of basic, elementary operations that can be used to generate a variety of mesh types. We phrased meshing as an optimization problem and introduced topological and geometric measures to construct a weighted objective function which was used to generate a reward signal for our agent.

We demonstrated that our agent can be trained on the meshing task and is successfully able to decompose random 2D shapes into the required element type. We showed that the exact same neural network architecture can be trained to produce triangles or quadrilaterals (or any other face degree) by simply changing the desired face degree parameter f^* from 3 to 4.

We explored automatic Hyperparameter Optimization (HPO) in detail. We showed the sensitivity of RL model performance to hyperparameter settings by running $\sim$100+ experiments in parallel and analyzing the results. While the number of hyperparameters can be large, typically a few hyperparameters have an outsized impact on model performance and are worth focusing on. We demonstrated the use of tools to extract hyperparameter importances from the results of HPO.

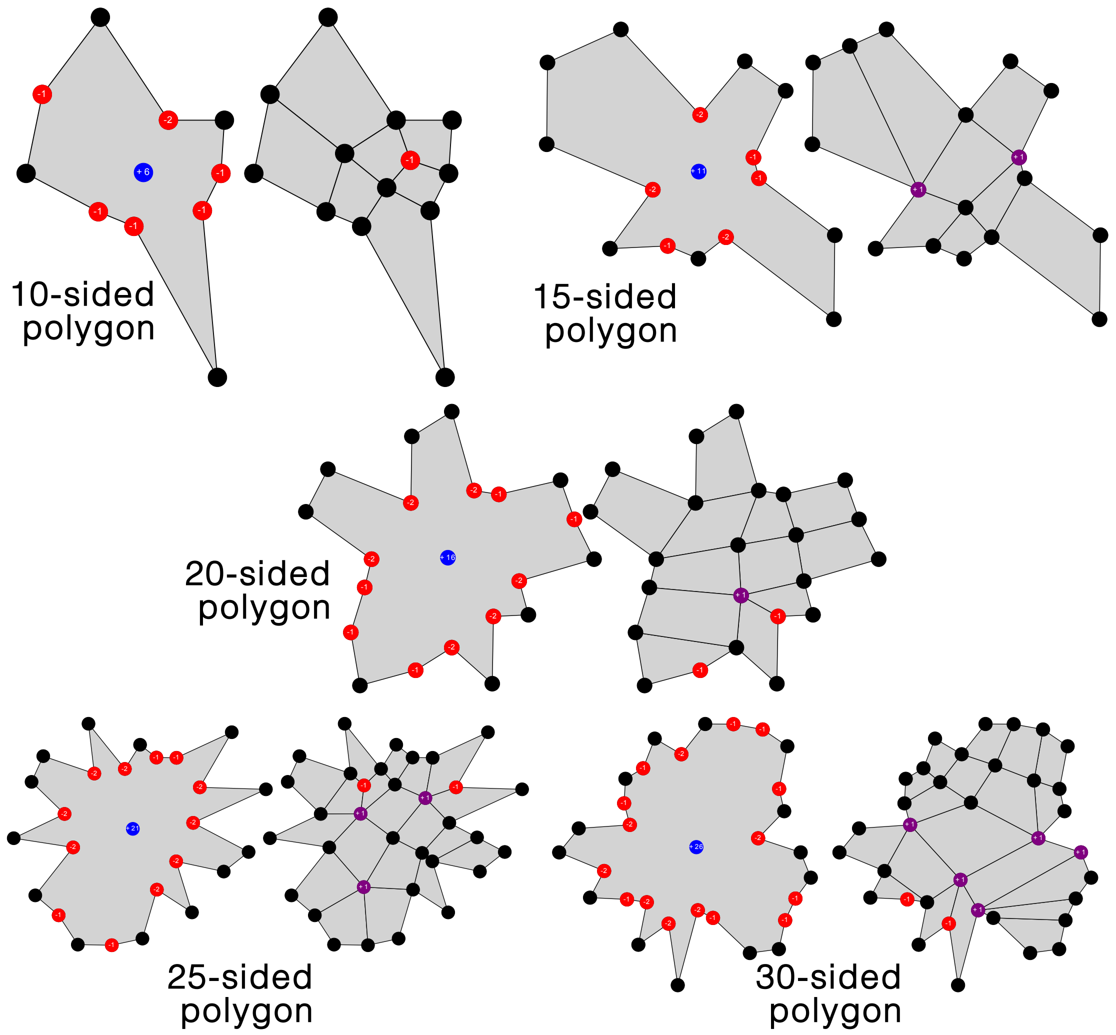


Figure 3.24: Example rollouts of the trained $f^* = 4$ policy for randomly generated polygonal shapes of different size. The final geometry was selected by running the policy 10 times from each initial state and picking the mesh with the smallest face score with ties broken by meshes with smallest total score. Note that in all the cases, the model consistently generates meshes with optimal face score. Irregular vertices are marked with their corresponding irregularity.

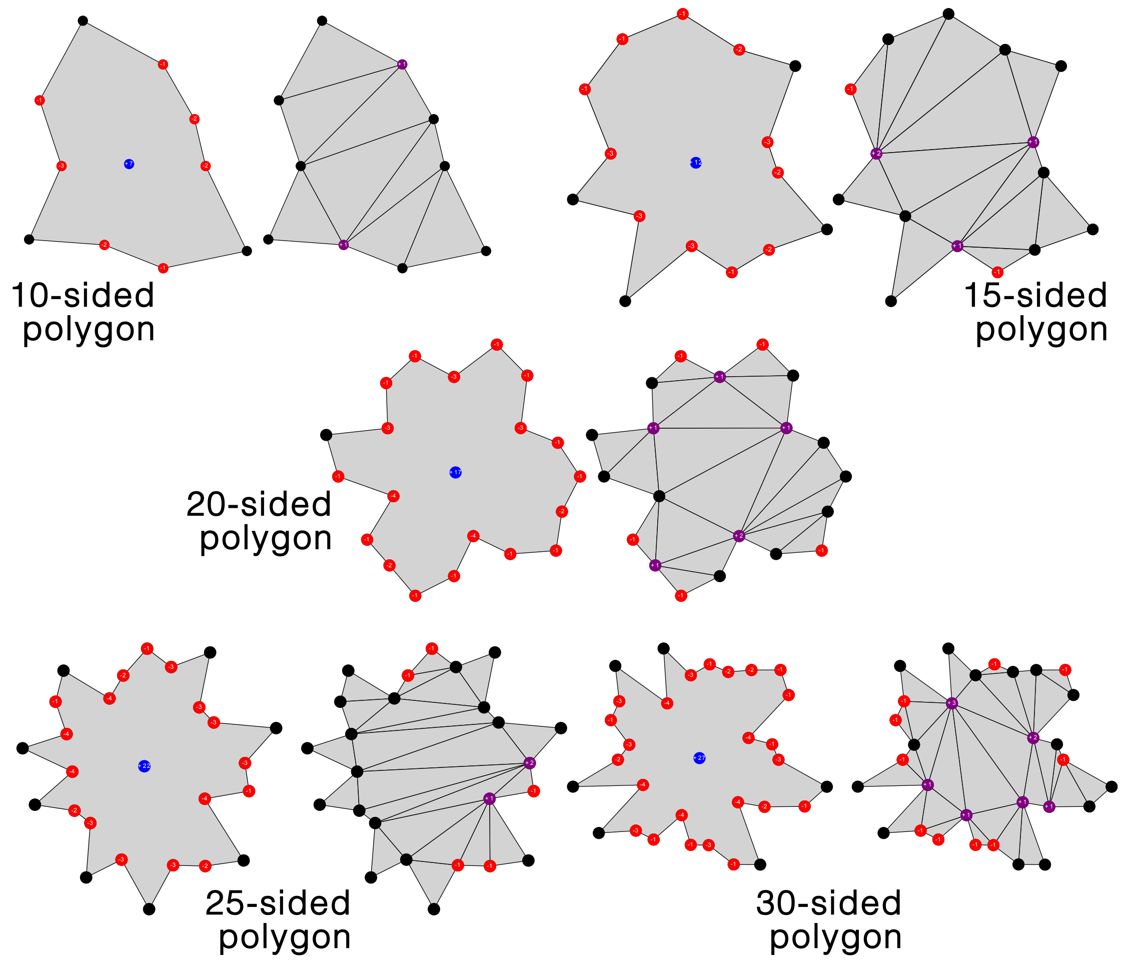


Figure 3.25: Example rollouts of the trained $f^* = 3$ policy for randomly generated polygonal shapes of different size. The final geometry was selected by running the policy 10 times from each initial state and picking the mesh with the smallest face score with ties broken by meshes with smallest total score. Note that in all the cases, the model consistently generates meshes with optimal face score. Irregular vertices are marked with their corresponding irregularity.

While the initial results look promising, there are certainly areas for improvement. One particular finding from observing figs. 3.24 and 3.25 is that the agent is converging upon local minima and not exploring the state space enough. The quadrilateral model has not learned the "global-split" operation. This is understandably challenging since the global-split requires a long-sequence of very specific moves to regularize the vertices. The triangle model never seems to insert a vertex which can be trivially shown to be sub-optimal in many scenarios (e.g. triangulating a regular hexagon.) Both of the above problems seem to do with a lack of exploration and convergence to local minima. Indeed, the relatively low values of the entropy weight in table 4.1 supports this view since the entropy loss encourages model exploration.

The exploration-exploitation dilemma is one of the most common challenges in RL. In our particular context, the following are promising areas of future work. Curriculum learning [6] aims to train the model on progressively challenging problems. Operations such as the "global-split" are easier to realize on smaller geometries, but this insight can be carried over to larger geometries. A more sophisticated approach is Never Give Up [5] wherein a single neural network is used to simultaneously learn multiple policies with different degrees of exploration/exploitation. They show that this approach results in transfer from exploratory policies to produce effective exploitative policies. We hope that our open-source implementation and integration with libraries such as `gymnasium` and `stable-baselines3` can simplify testing of different algorithms for future work.

3.8 Software

Our environment API and RL model have been made consistent with major open-source libraries to allow for ease-of-use and rapid experimentation. The majority of the code-base is unit-tested with over $\sim 600+$ hand-engineered test-cases. Source code for the models and results reported in this chapter can be found at
https://github.com/ArjunNarayanan/MeshRL.git

Chapter 4

LinFlo-Net: A two-stage deep learning method to generate simulation ready meshes of the heart

We present a deep learning model to automatically generate computer models of the human heart from patient imaging data with an emphasis on its capability to generate thin-walled cardiac structures. Our method works by deforming a template mesh to fit the cardiac structures to the given image. Compared with prior deep learning methods that adopted this approach, our framework is designed to minimize mesh self-penetration, which typically arises when deforming surface meshes separated by small distances. We achieve this by using a two-stage diffeomorphic deformation process along with a novel loss function derived from the kinematics of motion that penalizes surface contact and interpenetration. Our model demonstrates comparable accuracy with state-of-the-art methods while additionally producing meshes free of self-intersections. The resultant meshes are readily usable in physics based simulation, minimizing the need for post-processing and cleanup.

4.1 Introduction

Image-based computer modeling is playing an increasing role in understanding the mechanisms of cardiac disease and personalized care [53]. Broadly, this paradigm uses medical imaging, such as computed tomography (CT) or magnetic resonance (MR), to construct an anatomically accurate computer model of the heart in order to mathematically model physiological processes and probe functional information [15]. Reconstructing an accurate, personalized computer model of the heart is challenging because of imaging artifacts, limited resolution and difficultly differentiating between cardiac and surrounding tissues. Further, generating these models manually can take on the order of 6-10 hours [82] for an expert human. This is one of the major hurdles in the adoption of such technologies on a larger scale, motivating the development of automatic and scalable methods.

Segmentation is the process of identifying structures of interest in an image. Recent advances in machine learning and computer vision have demonstrated considerable success in the field of medical image segmentation. Numerous methods have been proposed that can achieve human-level performance on a large variety of structures of interest for the medical community [75]. However, segmentation often generates artifacts that are unfit for simulation-based modeling. Recently, we have developed alternative template based deep-learning methods that are able to generate simulation-ready computer models of cardiac structures automatically from images [37, 36]. These methods use machine learning to deform mesh templates to create a personalized geometry that best matches the image data. However, these methods do not guarantee a bijective mapping between the template and the deformed meshes. Thus, self-intersections and unphysiological distortions are possible, requiring significant post-processing steps to correct these artefacts.

Notably, prior methods have focused primarily on generating surface models of the blood pool boundaries since, except for the left ventricular (LV) myocardium, only blood pools are discerned from clinical imaging (cf. Fig. 4.1). However, many applications of cardiac modeling require modeling of the cardiac and vascular tissue. These tissues (with the exception of the LV wall) have modest thickness, and thus deformation of templates that contain such tissue structures are highly susceptible to self-intersections.

To address those limitations, we present here a deep learning method to produce whole-heart meshes with thin-walled structures from medical image (CT or MR) data. We employ a template-based method to ensure accurate and simulation compatible models free of self-intersections. Briefly, we use deep learning to deform a template mesh using a combination of linear-transformations and diffeomorphic flow deformations. We present a novel physics-based loss term that penalizes vanishing volumes thereby preventing self-penetrations when mapping thin walled structures. The model can be trained on data containing no thickness information and subsequently evaluated on a template with thickness. We demonstrate that this approach is able to successfully deform template meshes in a realistic manner without interpenetration. The predictions of our model can be readily used to generate volumetric grids for computational simulations.

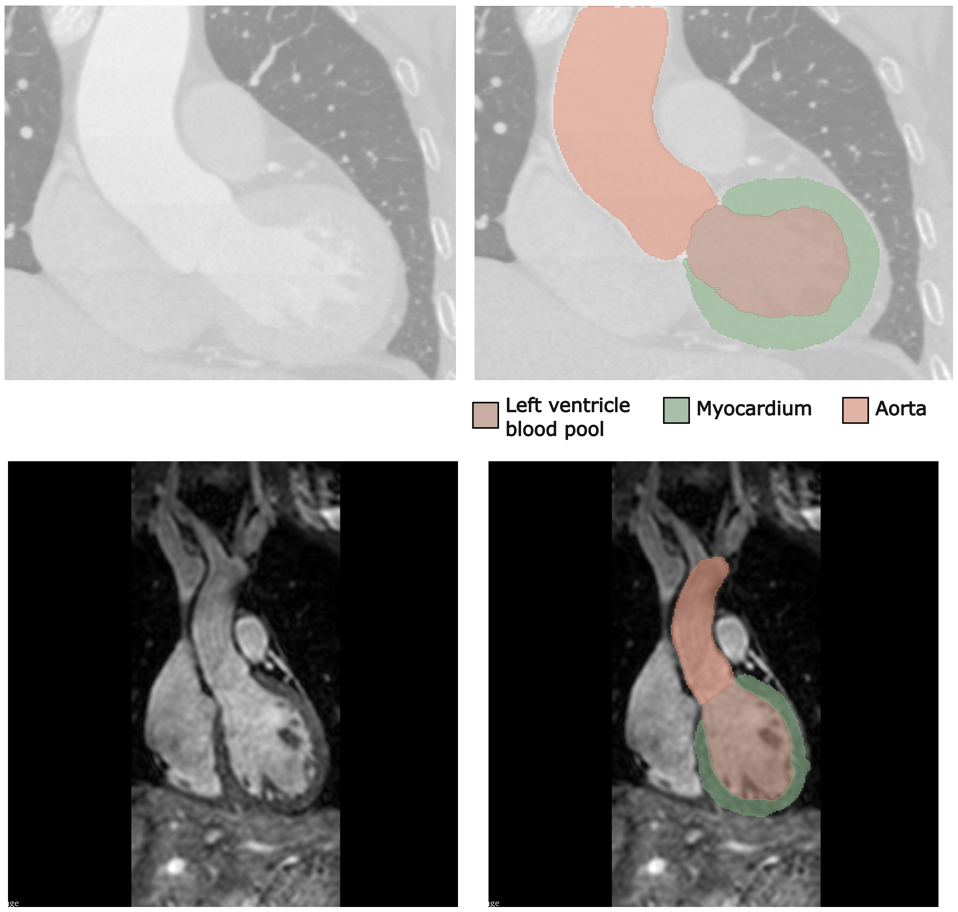


Figure 4.1: Illustrative examples of CT (top row) and MRI (bottom row) images of the cardiac region. The segmentation of some cardiac structures of interest are overlaid on the figures on the right. Since the myocardium is a thick muscular structure, it is clearly visible in the image. However, the tissue thickness of other structures like the aorta are not visible in these images, and only the blood pool within these structures is visible.

4.2 Related Work

Automatically generating 3D meshes from images is a challenging problem that has attracted considerable interest recently. In Pixel2Mesh [74], the authors deform an initial ellipsoidal template mesh to a target shape. Voxel2Mesh [76] extended these ideas to 3D medical images wherein the resultant mesh was dynamically refined in areas to capture geometric details. MeshDeformNet [37] and HeartDeformNet [36] demonstrated state-of-the-art accuracy in whole-heart mesh generation while preserving anatomical accuracy. However, these methods are not specifically designed to avoid self-intersections, which hinders their use for physics-based simulations. For example, standard meshing softwares like TetGen [29] fail to produce volumetric tetrahedral meshes from surface meshes that contain self-intersections.

To overcome these challenges, there has been interest in incorporating diffeomorphic constraints to deep learning algorithms, either explicitly through regularization losses or implicitly through inherent network designs, to produce deformation fields that are diffeomorphic. Pak et al. [47] generated 3D volumetric meshes of the aortic valve by training a neural network to predict the displacement field of an initial template. They proposed a novel distortion energy based on the singular value decomposition of the deformation gradient that penalizes deformations that are not diffeomorphic. The NeuralMeshFlow framework [27] used a neural network to predict the 3-dimensional vector field that could be used to integrate a point cloud (e.g. the vertices of a mesh) to a target geometry. Under regularity assumptions, this deformation is diffeomorphic. Similar ideas have been extended to applications in 3D medical imaging. CorticalFlow [38] and CortexODE [43] leveraged diffeomorphic flow fields for cortical surface reconstruction. UNetFlow [10] used a similar approach to generate meshes of abdominal structures such as the liver and pancreas. In practice, the meshes generated by such methods are not strictly intersection free due to various factors like numerical integration errors, but nevertheless, these methods typically show a significant reduction in the number of self-intersecting faces.

The method presented herein was inspired by ideas from the approaches described above. However, our method is unique in several key ways highlighted below:

1. We employ a two-stage deformation process consisting of an initial linear transformation followed by diffeomorphic flow-based deformation to capture finer details.

2. Along with the clinical image, we provide a representation of the template mesh's current position as input to our model. We use an unsigned distance function as the representation since computer vision models are well suited to this format. We observed empirically that this significantly improves model performance.

3. We demonstrate a simple and effective strategy to minimize integration errors by constraining the magnitude of the flow field, drawing connections to the

Courant–Friedrichs–Lewy condition in numerical analysis.

4. We propose a physics-based loss function derived from the kinematics of motion of continua that penalizes flow fields that generate collapsing volumes, thus preserving structure thickness even when thickness is not visible in the original image.

4.3 Methods

Neural network architecture

We aim to create a machine learning method that can deform a template heart model to match with a patient's image data. There can be significant variability in heart geometry. Figure 4.2 shows the variation in scale for two hearts in the dataset we will consider. The differences in scale are due to different field of view between scans and inter-patient variation in heart-size. We expect such variations to exist in real-world applications and our model is designed to handle these cases. Namely, our method first performs a linear transformation (cf. sec. 4.3) via scaling, rotation, and translation. A linear transformation is well behaved and guaranteed to be diffeomorphic. Further, it is highly interpretable. Thus, we aim to train our linear deformation module to maximally capture large-scale deformations. A subsequent (nonlinear) mesh deformation module is used to deform the linearly transformed mesh by integrating the mesh vertices along a learned vector field. This approach was designed with the requirement that the deformed template has minimal self-intersections. Since integration errors accumulate over larger deformations, by utilizing the linear transformation module in the first step, we reduce errors and self-intersections. Namely, the flow module is only required to capture finer details. Indeed, we have verified that the quality of meshes produced by the flow module alone (without any linear transformation) is poor.

Linear transformation

A linear transformation is a global operation applied to all mesh vertices. The transformation can be defined by 9 parameters: 3 scaling, 3 rotation, and 3 translation. We observed empirically that scaling and translation have a bigger impact on accuracy. Rotation only provides a small benefit, but we include it nonetheless since there is no significant overhead to do so. Our linear transformation module consists of a 3D convolutional neural network (CNN) encoder and a multi-layer perceptron (MLP) decoder. The CNN consists of multiple layers of convolution followed by

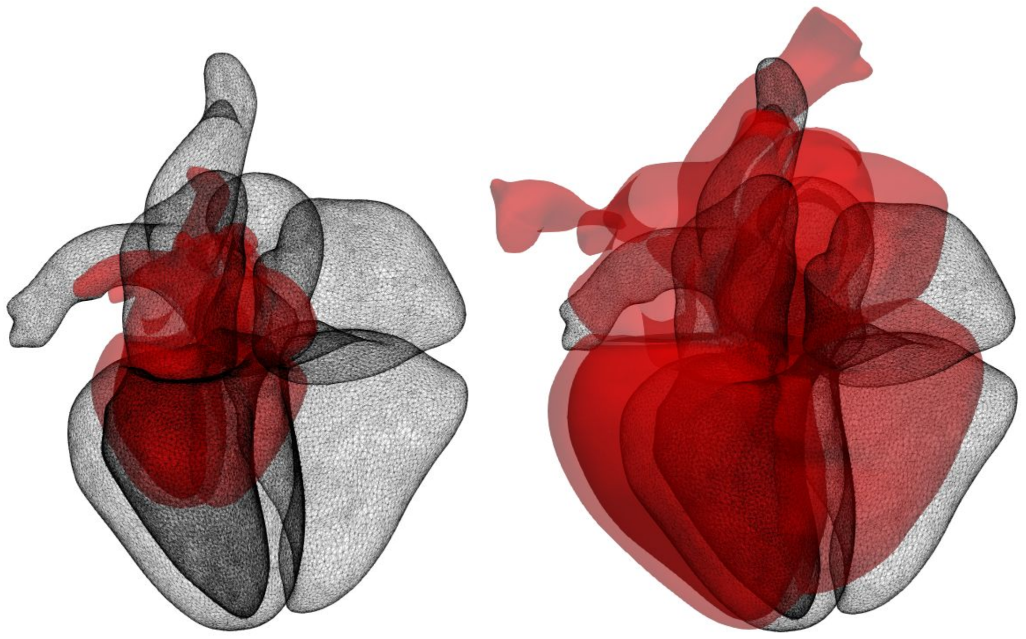


Figure 4.2: Two samples from our dataset that demonstrate the variation in scale across samples. Black wireframe is the template mesh, and red surface is the ground-truth mesh.

downsampling. The CNN processes a normalized input image of size 128^3 (see sec. 4.3 for details on the dataset and normalization protocol) and produces an encoding of dimension 4^3 with 512 channels. This encoding is flattened and processed by an MLP with a single hidden layer to produce the 9 parameters defining a linear transformation. The model is initialized to produce the identity transformation.

The predicted parameters are used to perform a linear transformation on a template mesh. We use the center of the image as the origin of the transformation. We apply the operation in the following order: scale–rotate–translate. We use the same initial template, and always initialize it at the same location in normalized image coordinates. The model is trained to minimize the chamfer distance in the L1-norm between the transformed template and ground truth mesh vertices given by eq. 4.3. We compute the Chamfer loss for each cardiac structure separately and take its average. The linear transformation and loss function are implemented in PyTorch3D [56]. We show a schematic of this module in fig. 4.3.

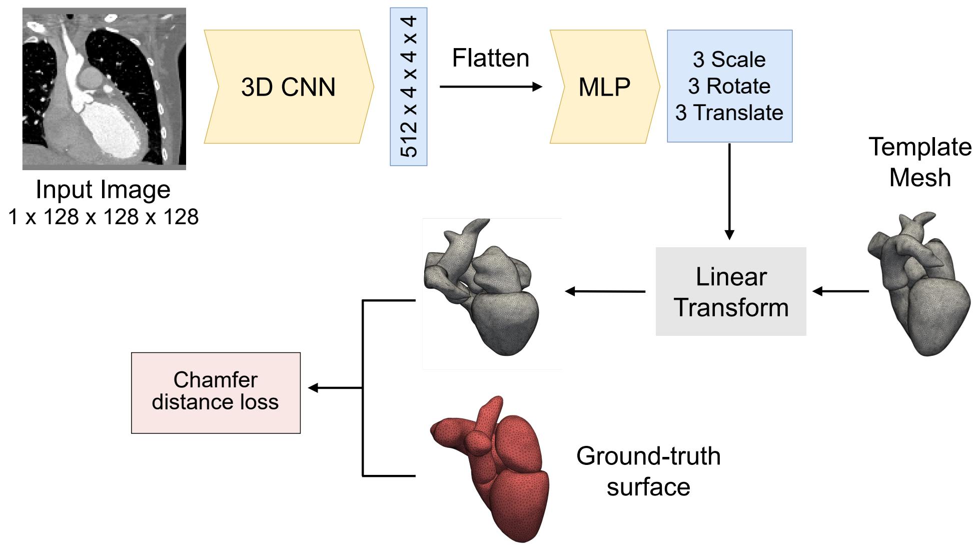


Figure 4.3: Workflow describing the training pipeline for the linear transformation module

Flow deformation

The linear transformation is a global operation and is unable to adjust the template to the finer features of the target geometry. We instead require a spatially varying deformation field. While this deformation field can be directly learned, it will generally not be diffeomorphic. Instead, we can learn a flow vector field that can be used to integrate the vertices of the linearly transformed template, resulting in a diffeomophic deformation. The learned vector field is further constrained using the loss function eq. 4.5 to prevent collapsing volume. A schematic of this module is shown in fig. 4.4.

We trained a U-Net architecture [58] to produce a dense flow vector field in the image space. U-Net has emerged as a mature technology particularly in the field of medical image segmentation. The U-Net model is able to produce an output that is at the original resolution of the image space. Further, the use of skip connections between the encoder and decoder arm equip the model with local and global context in the image. This makes the U-Net architecture ideally suited to our application since we wish to predict a dense flow vector field at the same resolution as the input image. Prior flow based methods [38, 10] have also successfully employed the U-Net in their model design.

Since the position of the template is no longer fixed but depends on the result of the linear transformation, the model requires a representation of the template mesh's current position. We achieve this by providing an unsigned distance map, $d(x)$, that encodes the template mesh's position in the image space,

$$d(x) = \min_{y \in T} |x - y|_2 \tag{4.1}$$

where T is the surface of the template mesh and $|\cdot|_2$ is the L2 distance norm. This distance map is concatenated to the image as a second channel and serves as the input to the U-Net.

To avoid excessive compute at training time, we construct the distance map for the initial template ahead of time and simultaneously apply the same learned linear transformation to the template and the distance map. The distance map is computed from the center of each voxel to the nearest point on the surface of the initial template mesh. We use routines available in PyTorch3d to compute the distance to surface. We empirically observed that the distance map significantly improves the performance of the model. This approach can be an effective strategy to improve the performance of any multi-stage mesh deformation process. Namely, a distance function can effectively encode the location of the template mesh to standard computer vision models that expect inputs to be represented as dense arrays.

We performed numerical integration using an explicit 4th order Runge-Kutta scheme. Numerical integration is an approximation to the "true" integral and accumulates error over time. Prior flow-based approaches [38] aim to minimize integration errors by controlling the time-step size based on the Lipschitz constant of the flow vector field. However, by doing this, the time-step size of integration is controlled by the region with the largest Lipschitz constant, possibly resulting in undesirably small time steps globally. We propose here an alternative strategy to mitigate the accumulation of integration error. We enforce the condition that a vertex cannot travel more than a distance of 1 voxel per time step by upper-bounding the L2 norm of the flow vector field. This condition is similar to the Courant-Friedrichs-Lewy (CFL) condition in numerical analysis. We refer the reader to a standard resource on numerical analysis such as [40] for a more detailed discussion on the CFL condition. If v is the predicted flow vector field before clipping, we compute the clipped flow vector field v_{clip} as,

$$v_{\text{clip}} = \alpha \frac{v}{\max(|v|_2, \alpha)} \tag{4.2}$$

where α is a parameter that is set to approximately 1-voxel spacing. Notice that v_{clip} has an L2 norm upper-bounded by α. Since the normalized image coordinates is a cube ranging across $[0,1]^3$ and is discretized into 128^3 uniform voxels, the voxel-spacing in normalized image coordinates is $1/128 \approx 0.0078$. Thus we took $\alpha = 0.0075 < 1/128$. By clipping the flow-field, we are able to use a uniform step-size throughout the training process. We empirically observed that clipping the flow vector field resulted in a reduction in the number of self-intersecting faces suggesting that this simple strategy can be effective at controlling error due to numerical integration.

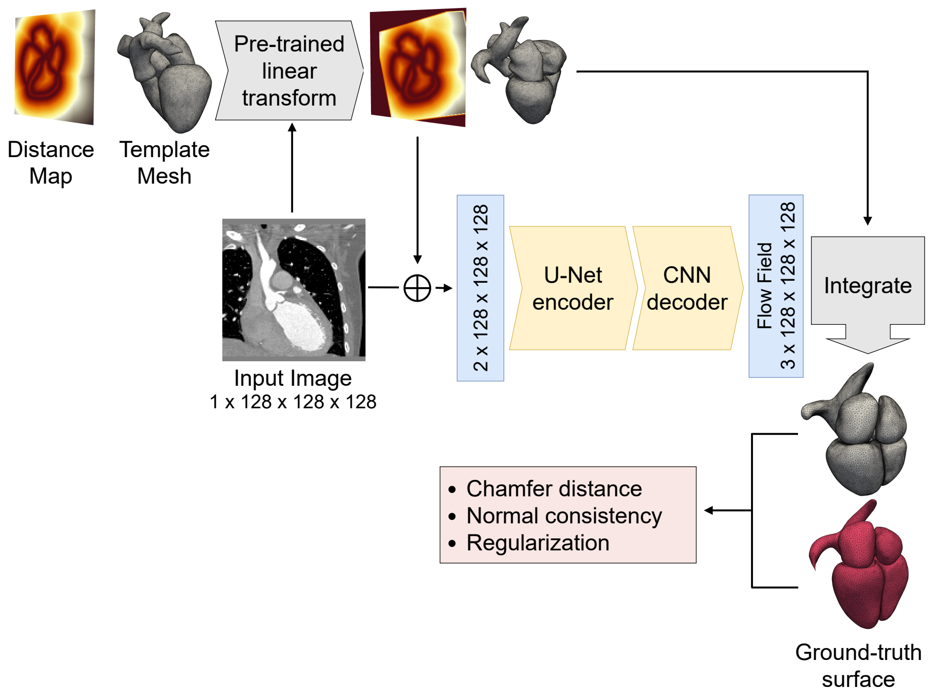


Figure 4.4: Workflow illustrating the training pipeline for the flow deformation module

Loss functions

The model is trained using a weighted sum of the following losses:

1. Chamfer distance in the L1 norm as defined in eq. 4.3.
2. Normal consistency between template and ground truth meshes.
3. Volume loss as defined in eq. 4.5.
4. Mesh regularization, which includes edge length, normal consistency across faces, and Laplacian smoothing loss.

Loss terms (1) and (4) are directly available in PyTorch3D [56].

Chamfer distance

The chamfer distance between two point clouds P_1 and P_2 can be computed as follows,

$$\begin{aligned}\text{chamfer}(P_1, P_2) = \frac{1}{P_1} \sum_{x \in P_1} \min_{y \in P_2} |x - y|_1 + \\ \frac{1}{P_2} \sum_{y \in P_2} \min_{x \in P_1} |x - y|_1\end{aligned} \tag{4.3}$$

Minimizing this loss leads to the model trying to increase the overlap between the point clouds. This has the effect of improving the accuracy of the model in terms of overlap with the grounds-truth geometry.

Normal consistency between template and ground truth

The normal consistency loss helps to associate surfaces with the correct orientation. This is typically achieved by computing the cosine similarity between the normal vectors from the template mesh with the normal vectors at the closest point in the ground-truth mesh and vice versa. This loss function is particularly important for the myocardium which is a cup-like structure. Since its inner and outer surfaces are separated by a small length scale, incorrectly associating these two surfaces will result in the two surfaces collapsing into each other. The normal consistency loss helps to prevent this. Note that the PyTorch3D implementation does not distinguish between the orientation of the two surfaces i.e. the sign of the normals does not affect the loss. This distinction is important for our application. For structures with thickness, not making this distinction can result in incorrectly associating surfaces together resulting in the collapse of this thickness. Instead, we use the following form of the normal consistency loss, which is a slight modification of the loss implemented in PyTorch3D. Given two point-sets P, Q and a normal function $n(\cdot)$ which gives the normal at any point, the normal consistency loss L_{NC} is,

$$\begin{aligned} L_{NC}(P, Q) &= \frac{1}{|P|} \sum_{x \in P} 1 - n(x)^T n(y) \\ \text{s.t.} \quad y &= \text{argmin}_{z \in Q} |x - z|_2 \end{aligned} \tag{4.4}$$

We compute the average of $L_{NC}(T, G)$ and $L_{NC}(G, T)$ as the final loss where T and G are respectively the template mesh and the ground truth mesh.

Volume loss

To address the issue of collapsing volumes, we introduce a physics-based loss function derived from the kinematics of continuous media. Consider a mesh vertex located at $x(t)$ where t is the time-like parameter of integration. We can take $t \in [0, 1]$ without loss of generality with $x(0)$ being the initial location of the mesh vertex and $x(1)$ being the location of the vertex after integration along the flow vector field v. Let $V(0)$ and $V(1)$ represent the volume of a small neighborhood around x in the initial and final states. Since the flow vector field transforms the entire space, it also transforms the volume $V(0)$ to $V(1)$. A standard result in continuum mechanics (see chapter 3 in [1]) relates the rate of change of V to the divergence of the flow vector field,

$$\frac{\mathrm{d}V}{\mathrm{d}t} = V \ \mathrm{div}(v)$$

Integrating this quantity from time $t = 0$ to 1 we get,

$$L_{vol} := \frac{V_0}{V_1} = \exp\left(-\int_0^1 \mathrm{div}(v) \ \mathrm{dt}\right)$$

Suppose the flow vector field v is such that a region with finite initial volume V_0 collapses to an infinitesimally small volume V_1 (i.e. $V_1 << V_0$) then L_{vol} evaluates to a large value. We call L_{vol} the volume loss and demonstrate that including this quantity during training helps to prevent surfaces separated by a small distance from collapsing into each other. L_{vol} is computed by computing the integral in the above equation along the trajectory of every mesh vertex for which this loss is applied. Even though v is taken to be a stationary vector field (i.e. it is not time-dependent), for a given mesh vertex $\text{div}(v)$ varies along its integration trajectory. We compute $\text{div}(v)$ using a central finite-difference scheme in the image space. Subsequently, we simultaneously integrate the positions and accumulate the above integral for all mesh vertices to which this loss is applied.

Note that the exponential term in the above equation can cause this loss and its associated gradients to blow up which could cause instability in training. To avoid this, we clip the upper bound of the integral point-wise prior to computing the exponential. The equation below is our final expression for the Volume Loss,

$$L_{vol} = \exp\left(\min\left(3, -\int_0^1 \text{div}(v)\ \text{dt}\right)\right) \tag{4.5}$$

This simple strategy resulted in stable training in our experiments. Alternatively, gradient clipping available in machine learning libraries such as PyTorch may be directly employed.

Dataset

For training and testing, we use the same dataset as [37]. The training data consists of data from four public datasets including the multi-modality whole heart segmentation challenge (MMWHS) [82], orCaScore challenge [77], left atrial wall thickness challenge (SLAWT) [33], and left-atrial segmentation challenge (LASC) [71]. In total we had 101 CT samples and 47 MR samples in our dataset. We split this into a training dataset (86 CT, 41 MR) and a validation dataset (15 CT, 6 MR) for hyper-parameter tuning. The MMWHS challenge further provides a held-out test dataset consisting of 40 CT and 40 MR samples for which no ground-truth is available. Instead, the challenge organizers provide encrypted scripts that can be used to evaluate predicted segmentations on a variety of accuracy metrics. We use this held-out test dataset to evaluate the final performance of our models and report the same in the subsequent section. The input images are available in the NIfTI file format. Other image formats can be supported so long as the appropriate backend is available to load these files and convert them into image arrays.

We augment the training dataset with small perturbations including random scaling, translation, rotation, shear, and local b-spline deformations. We produce 20 random augmentations per image. Figure 4.5 shows typical image samples and

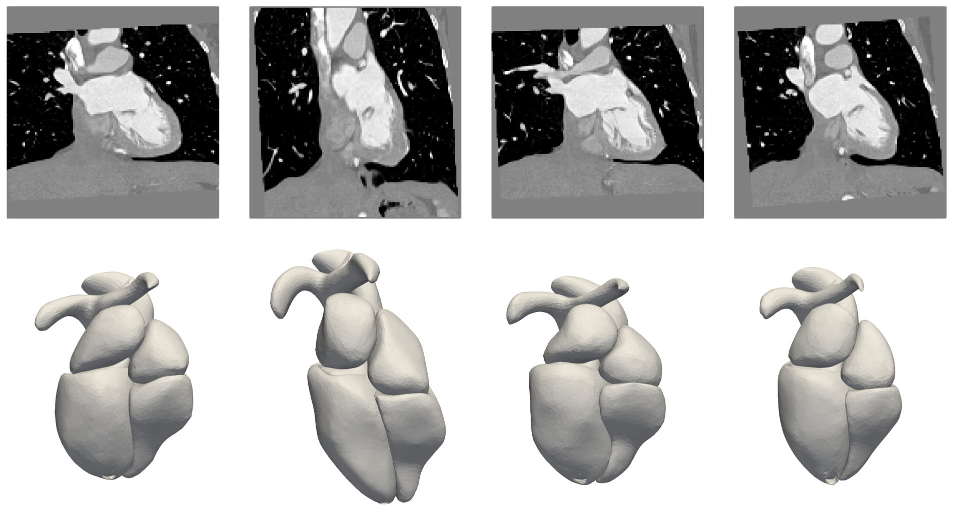


Figure 4.5: An illustration of samples generated by the data-augmentation process from a given input image and associated ground-truth segmentation. We generate these samples by applying small perturbations including random scaling, translation, rotation, shear, and local b-spline deformations to the input image and segmentation. Meshes are produced from the generated segmentations via the marching cubes algorithm.

associated ground-truth surfaces generated by the data-augmentation procedure for a single input image and segmentation. Subsequently, the dataset is pre-processed and normalized prior to training following a procedure similar to [35]. The input images are resampled to a standard dimension of 128^3. Further, the voxel intensities are thresholded and normalized to lie between $[-1, 1]$. We apply different normalization procedures to CT and MR samples since the Hounsfeld intensity values for CT images is standardized. For CT samples we threshold voxel intensities to lie between $[-750, 750]$ and then scale the values to $[-1, 1]$. We threshold MR voxel intensities at the 20^{th} percentile of values (lower) and 99^{th} percentile of values (upper) and scale the resulting values to $[-1, 1]$. We apply thresholding to ensure that the voxel intensities in the range expressed by the cardiac structures of interest are captured, while largely removing intensity variations due to bones or imaging artefacts. Ground-truth meshes were generated using the marching cube algorithm on the ground-truth segmentations, followed by mesh smoothing.

Since the original MR volumes had a significantly larger field-of-view, we cropped

Table 4.1: Hyperparameter settings used to train the model

Hyperparameter	Value
Batch size	1
Initial learning rate	5×10^{-5}
Flow norm threshold	0.0075

Loss Function	Weight
Chamfer distance	1
Chamfer normal consistency	0.20
Volume loss	0.005
Edge loss	50
Normal consistency loss	1
Laplace smoothing loss	30

the images such that the cardiac structures occupied a similar proportion of the image space as in the CT data. This was automated in the training data by using the ground truth segmentations to determine the cropping dimensions. The test data was manually cropped, however automatic methods to generate a bounding box [51] may be used in the future to avoid manual cropping. We evaluated our models on the held-out test dataset of the MMWHS Challenge [82].

4.4 Results

We trained a single model on both CT and MR data using the hyperparameters listed in table 4.1. This corresponds to model LT-FL-V. For model LT-FL we set the weight of the volume loss to zero. We used a learning rate scheduler to reduce the learning rate when the validation loss plateaus for a fixed number of iterations. The model was trained on a single Nvidia 2080TI GPU for about 24 hours.

Model accuracy

To measure our model's accuracy, we converted our generated meshes into segmentations. The VTK [59] library provides routines to convert a mesh into an image mask. We used these routines to obtain an image mask consisting of the region enclosed by the mesh in the image space. We obtained one mask per cardiac structure and consider these as the segmentations corresponding to our predicted meshes. We then evaluated the accuracy of these segmentations using the scripts provided by the MMWHS challenge organizers. We consider HeartDeformNet [36] as the benchmark for this problem and compare our accuracy and mesh quality against the results reported in that work. Table 4.2 compares accuracy metrics including Dice Score, Jaccard Index, Average Symmetric Surface Distance (ASSD) and Hausdorff Distance (HD). Similarly Table 4.3 compares these accuracy metrics for the MR test dataset. The reported results are the average over the 40 test samples in each dataset with the standard deviation reported in parantheses.

We observe that the performance of our model is commensurate similar with HeartDeformNet on average being slightly more accurate. We note that HeartDeformNet deforms the mesh over 3 successive stages each of which is a graph convolutional neural network operating on the mesh. We anticipate that by incorporating a second deformation block, or fine-tuning a pre-trained network, we can substantially improve our performance.

Mesh quality

Next we compare the quality of the generated meshes in terms of the percentage self-intersecting faces (SIF). We measure SIF using the PyMesh library [81]. SIF measures intersections due to surface inter-penetration and intersections due to element inversion. The usefulness of our method is clearly seen in Table 4.4 and Table 4.5. These tables report the percentage SIF for each cardiac structure for CT and MR respectively measured on the meshes generated from the test dataset. Notice that for CT samples each of the 40 generated meshes for the CT test dataset had *zero* self-intersecting faces across all cardiac structures. Similarly, 38 out of 40 generated meshes for the MR test dataset showed *zero* self-intersecting faces across all cardiac structures. Of the two samples that had non-zero SIF, one had 0.85% SIF in the Aorta and another sample had 0.92% SIF in the pulmonary artery. We note that our template mesh had a higher resolution compared to the template used by HeartDeformNet. Our template mesh consists of 110K triangular faces and the template used by HeartDeformNet consists of 45K faces. Generally speaking, higher resolution meshes are more prone to self-intersections due to element collapse. Despite having a significantly higher number of elements, our model is able to consistently generate meshes with far fewer self-intersecting faces.

The ability to reliably generate meshes with zero self-intersections is a very useful property because the presence of even a few self-intersecting faces can cause a mesh generation software like TetGen [29] to abort. Thus, our model makes significant progress towards robustly producing simulation-ready meshes that do not require any post-processing before being usable in a simulation environment.

Table 4.2: Comparing mean (standard deviation) of accuracy metrics on CT test data. ↑ (↓) indicates higher (lower) value is better.

Metric	Model	Epi	LA	LV	RA	RV	Ao	PA	WH
Dice (↑)	Ours	0.86 (0.04)	0.92 (0.03)	0.91 (0.06)	0.85 (0.08)	0.88 (0.04)	0.89 (0.04)	0.82 (0.09)	0.89 (0.03)
	HeartDeformNet	0.88 (0.03)	0.93 (0.03)	0.92 (0.04)	0.89 (0.05)	0.91 (0.03)	0.91 (0.04)	0.85 (0.09)	0.91 (0.02)
Jaccard (↑)	Ours	0.76 (0.06)	0.85 (0.05)	0.84 (0.09)	0.75 (0.11)	0.79 (0.07)	0.81 (0.06)	0.70 (0.12)	0.80 (0.04)
	HeartDeformNet	0.79 (0.05)	0.86 (0.05)	0.85 (0.06)	0.80 (0.07)	0.83 (0.05)	0.84 (0.06)	0.74 (0.13)	0.83 (0.04)
ASSD (↓)	Ours	1.40 (0.36)	1.20 (0.33)	1.16 (0.47)	1.82 (0.72)	1.34 (0.39)	1.13 (0.35)	1.53 (0.75)	1.36 (0.26)
	HeartDeformNet	1.38 (0.24)	1.14 (0.38)	1.10 (0.33)	1.63 (0.72)	1.14 (0.27)	0.93 (0.38)	1.20 (0.74)	1.25 (0.24)
HD (↓)	Ours	13.74 (2.97)	12.21 (2.20)	12.55 (2.05)	12.52 (4.83)	10.80 (4.74)	5.79 (1.70)	7.84 (3.01)	16.32 (4.00)
	HeartDeformNet	14.40 (2.75)	8.18 (3.08)	6.87 (2.51)	12.46 (5.99)	9.55 (2.01)	5.54 (1.91)	8.45 (2.96)	16.63 (4.37)

Table 4.3: Comparing mean (standard deviation) of accuracy metrics on MR test data. ↑ (↓) indicates higher (lower) value is better.

Matric	Model	Epi	LA	LV	RA	RV	Ao	PA	WH
Dice (↑)	Ours	0.76 (0.09)	0.85 (0.05)	0.90 (0.03)	0.86 (0.04)	0.85 (0.09)	0.84 (0.06)	0.69 (0.14)	0.84 (0.05)
	HeartDeformNet	0.79 (0.10)	0.86 (0.08)	0.89 (0.06)	0.88 (0.04)	0.87 (0.06)	0.83 (0.07)	0.78 (0.12)	0.86 (0.05)
Jaccard (↑)	Ours	0.62 (0.11)	0.74 (0.08)	0.82 (0.05)	0.76 (0.06)	0.74 (0.12)	0.73 (0.08)	0.54 (0.15)	0.73 (0.07)
	HeartDeformNet	0.66 (0.11)	0.77 (0.10)	0.81 (0.09)	0.78 (0.06)	0.78 (0.09)	0.72 (0.10)	0.65 (0.14)	0.76 (0.07)
ASSD (↓)	Ours	2.22 (1.14)	1.72 (0.67)	1.46 (0.51)	1.79 (0.44)	1.97 (1.36)	1.48 (0.54)	2.37 (0.92)	1.86 (0.69)
	HeartDeformNet	2.10 (1.24)	1.57 (0.66)	1.54 (0.77)	1.58 (0.49)	1.56 (0.72)	1.53 (0.66)	1.66 (0.73)	1.66 (0.59)
HD (↓)	Ours	16.90 (3.48)	12.76 (3.30)	13.86 (6.23)	11.80 (3.15)	12.99 (9.69)	7.71 (4.72)	10.29 (3.70)	20.59 (8.96)
	HeartDeformNet	15.96 (3.33)	10.16 (3.77)	8.97 (6.74)	12.56 (4.51)	12.46 (9.12)	7.39 (3.39)	9.14 (3.43)	18.91 (9.24)

Table 4.4: Comparing mean (max) percentage self-intersecting faces on CT test dataset

Model	Epi	LA	LV	RA	RV	Ao	PA
Ours	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
HeartDeformNet	0 (0)	0 (0)	0 (0)	0.01 (0.25)	0 (0)	0 (0)	0.18 (1.14)

Ablation study of model architecture

We investigated the different elements of our proposed model architecture and compared the accuracy and quality of the generated meshes. We compare the following model design choices here,

1. LT - A model that *only* employs the linear transformation.

Table 4.5: Comparing mean (max) percentage self-intersecting faces on MR test dataset

Model	Epi	LA	LV	RA	RV	Ao	PA
Ours	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0.02 (0.85)	0.023 (0.92)
HeartDeformNet	0.019 (0.32)	0 (0)	0.013 (0.38)	0.03 (0.07)	0.012 (0.31)	0.138 (2.96)	0.625 (4.76)

2. FL - A model that *only* employs the flow deformation *without* the linear transformation.
3. LT-FL - A model combining the linear transformation and flow deformation but trained without the volume loss eq. 4.5.
4. LT-FL-v - The LT-FL model that is trained additionally with the volume loss eq. 4.5.

To assess the performance of these models, we generated meshes on the held-out test dataset. Figures 4.6 and 4.7 compare the dice scores of the different models. Detailed comparison of all accuracy metrics are provided in tables 4.6 and 4.7. Further, detailed comparison of mesh quality are provided in tables 4.8 and 4.9. We observed that combining the linear transformation with the flow deformation substantially improves the accuracy and quality of the generated meshes in almost all cases. As expected, the linear transformation introduces zero self-intersections in all cases, which benefits the LT-FL model as well. Further, the volume loss has a powerful regularization effect on the generated flow fields, which helps to almost completely eliminate self-intersections.

Patient specific meshes of left ventricle with tissue thickness

We now showcase an application of our method in generating meshes of cardiac structures wherein thickness information is not available in the ground truth. Figure 4.8 shows a template mesh of the left-ventricle myocardium that was generated by combining meshes of the myocardium, aorta, and left atrium. As shown in fig. 4.1, the thickness of the aorta and left atrium is not visible in the original image and must be added based on knowledge of cardiac tissue.

It is important to note that since the ground truth surfaces of the aorta and left atrium do not contain thickness, the vector fields describing the deformation of the template in a neighborhood around these surfaces point towards these surfaces. In

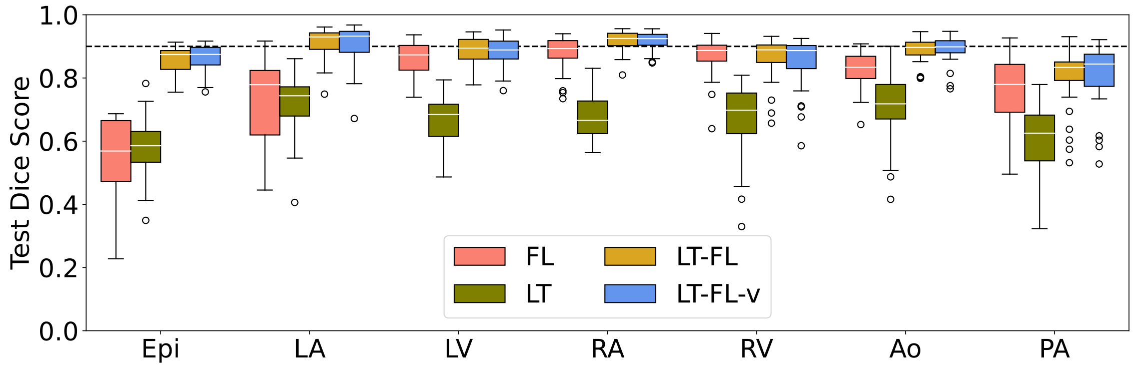


Figure 4.6: Comparison of dice score on CT test data for different model choices

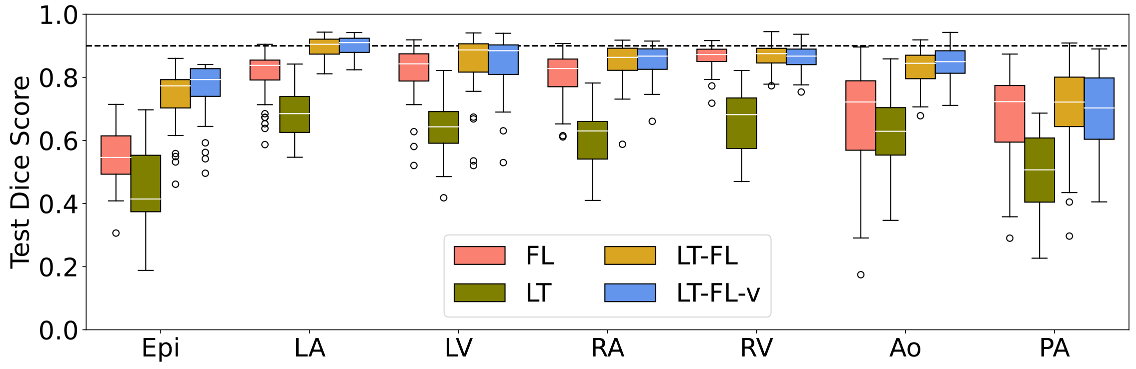


Figure 4.7: Comparison of dice score on MR test data for different model choices

other words, these surfaces act like sinks in the deformation field. The model thus learns to collapse any wall thicknesses added in the template. This is undesirable for the following reasons:

1. We will not be able to generate a volumetric mesh of the myocardium from these collapsed surfaces. Repairing these collapsed surfaces is non-trivial and requires significant manual effort. Further, the mechanical response of a collapsed surface will vastly over-exaggerate the true deformation of these tissues when subjected to mechanical load.

2. We require the red surfaces in fig. 4.8 to apply boundary conditions for electromechanics simulation of the LV. If these surfaces are collapsed, we will not

Table 4.6: Ablation study – comparing mean (standard deviation) of accuracy metrics on CT test data. ↑ (↓) indicates higher (lower) value is better.

Metric	Model	Epi	LA	LV	RA	RV	Ao	PA	WH
Dice (↑)	Fl	0.56 (0.12)	0.88 (0.05)	0.73 (0.14)	0.87 (0.06)	0.86 (0.05)	0.83 (0.06)	0.76 (0.10)	0.79 (0.05)
	LT	0.58 (0.09)	0.68 (0.07)	0.72 (0.09)	0.67 (0.11)	0.66 (0.08)	0.71 (0.11)	0.61 (0.11)	0.66 (0.06)
	LT-Fl	0.86 (0.04)	0.92 (0.03)	0.91 (0.05)	0.87 (0.06)	0.89 (0.04)	0.89 (0.03)	0.81 (0.09)	0.89 (0.02)
	LT-Fl-v	0.86 (0.04)	0.92 (0.03)	0.91 (0.06)	0.85 (0.08)	0.88 (0.04)	0.89 (0.04)	0.81 (0.09)	0.89 (0.03)
Jaccard (↑)	Fl	0.39 (0.11)	0.79 (0.08)	0.59 (0.16)	0.77 (0.09)	0.76 (0.08)	0.71 (0.08)	0.63 (0.13)	0.66 (0.07)
	LT	0.41 (0.09)	0.51 (0.08)	0.57 (0.10)	0.51 (0.12)	0.50 (0.09)	0.56 (0.13)	0.44 (0.11)	0.49 (0.06)
	LT-Fl	0.75 (0.06)	0.85 (0.05)	0.84 (0.07)	0.77 (0.09)	0.80 (0.06)	0.80 (0.05)	0.69 (0.12)	0.80 (0.04)
	LT-Fl-v	0.76 (0.06)	0.85 (0.05)	0.84 (0.09)	0.75 (0.11)	0.79 (0.07)	0.81 (0.06)	0.69 (0.12)	0.80 (0.04)
ASSD (↓)	Fl	1.97 (1.10)	1.54 (0.41)	4.40 (1.85)	1.86 (0.85)	1.75 (0.51)	2.02 (0.84)	2.00 (0.89)	2.24 (0.52)
	LT	3.63 (0.97)	4.28 (1.05)	4.01 (1.04)	4.44 (1.21)	4.59 (1.05)	2.94 (1.09)	3.71 (1.23)	4.07 (0.64)
	LT-Fl	1.33 (0.38)	1.22 (0.36)	1.16 (0.37)	1.70 (0.69)	1.36 (0.43)	1.17 (0.30)	1.64 (0.77)	1.35 (0.24)
	LT-Fl-v	1.43 (0.36)	1.25 (0.33)	1.17 (0.47)	1.82 (0.72)	1.33 (0.39)	1.13 (0.35)	1.55 (0.75)	1.37 (0.26)
HD (↓)	Fl	13.90 (9.09)	10.81 (3.13)	12.68 (2.84)	12.01 (5.26)	12.10 (5.11)	10.40 (5.37)	10.61 (4.56)	18.86 (9.09)
	LT	16.80 (2.46)	14.75 (3.47)	14.88 (3.37)	17.82 (4.38)	16.71 (3.30)	9.79 (2.75)	12.69 (3.99)	20.34 (3.80)
	LT-Fl	12.59 (2.95)	9.79 (3.08)	8.88 (3.08)	11.73 (4.99)	10.65 (4.71)	5.96 (1.82)	8.08 (2.92)	15.86 (4.19)
	LT-Fl-v	13.60 (2.97)	12.15 (2.20)	12.30 (2.05)	12.55 (4.83)	11.28 (4.74)	5.74 (1.70)	7.92 (3.01)	16.21 (4.00)

Table 4.7: Ablation study – comparing mean (standard deviation) of accuracy metrics on MR test data. ↑ (↓) indicates higher (lower) value is better.

Metric	Model	Epi	LA	LV	RA	RV	Ao	PA	WH
Dice (↑)	Fl	0.55 (0.09)	0.81 (0.07)	0.81 (0.08)	0.86 (0.04)	0.82 (0.09)	0.67 (0.18)	0.68 (0.15)	0.77 (0.05)
	LT	0.45 (0.12)	0.62 (0.09)	0.69 (0.08)	0.66 (0.10)	0.64 (0.08)	0.62 (0.12)	0.50 (0.13)	0.60 (0.06)
	LT-Fl	0.74 (0.09)	0.85 (0.06)	0.89 (0.04)	0.87 (0.04)	0.85 (0.10)	0.83 (0.06)	0.70 (0.14)	0.84 (0.05)
	LT-Fl-v	0.76 (0.09)	0.85 (0.05)	0.90 (0.03)	0.86 (0.04)	0.85 (0.09)	0.84 (0.06)	0.69 (0.14)	0.84 (0.05)
Jaccard (↑)	Fl	0.38 (0.08)	0.68 (0.10)	0.69 (0.10)	0.76 (0.06)	0.70 (0.12)	0.52 (0.19)	0.53 (0.16)	0.63 (0.07)
	LT	0.30 (0.11)	0.45 (0.09)	0.53 (0.09)	0.51 (0.11)	0.48 (0.09)	0.46 (0.12)	0.34 (0.11)	0.43 (0.06)
	LT-Fl	0.60 (0.11)	0.74 (0.09)	0.81 (0.06)	0.77 (0.06)	0.74 (0.13)	0.71 (0.08)	0.55 (0.16)	0.73 (0.07)
	LT-Fl-v	0.62 (0.11)	0.74 (0.08)	0.81 (0.05)	0.76 (0.06)	0.75 (0.12)	0.72 (0.08)	0.55 (0.15)	0.73 (0.07)
ASSD (↓)	Fl	2.89 (1.18)	2.00 (0.94)	3.00 (1.09)	1.89 (0.54)	2.20 (0.86)	3.14 (1.51)	2.46 (1.03)	2.49 (0.69)
	LT	5.20 (1.43)	4.47 (1.37)	4.94 (1.55)	4.85 (1.96)	5.20 (1.53)	3.60 (1.25)	4.61 (1.56)	4.97 (0.82)
	LT-Fl	2.46 (1.16)	1.70 (0.69)	1.51 (0.60)	1.65 (0.47)	1.92 (1.30)	1.62 (0.48)	2.26 (0.94)	1.88 (0.68)
	LT-Fl-v	2.26 (1.14)	1.72 (0.67)	1.48 (0.51)	1.78 (0.44)	1.96 (1.36)	1.52 (0.54)	2.34 (0.92)	1.86 (0.69)
HD (↓)	Fl	20.69 (7.37)	11.98 (4.69)	14.05 (6.35)	12.67 (4.20)	15.36 (9.26)	15.14 (6.16)	11.59 (3.95)	23.63 (9.94)
	LT	21.23 (5.19)	17.04 (4.26)	20.55 (7.41)	18.65 (6.01)	22.83 (8.65)	12.66 (4.20)	14.94 (3.87)	27.25 (8.00)
	LT-Fl	16.91 (4.52)	10.10 (3.53)	10.14 (7.23)	12.48 (4.51)	13.53 (9.82)	8.23 (4.45)	10.89 (4.57)	20.85 (9.29)
	LT-Fl-v	16.88 (3.48)	13.04 (3.30)	13.90 (6.23)	11.71 (3.15)	12.71 (9.69)	7.62 (4.72)	10.19 (3.70)	20.57 (8.96)

be able to apply the necessary boundary conditions.

A significant advantage of our method is that it is agnostic to the resolution of the template mesh and may be applied to any subset of the original template used for training. This enables us to deploy our model on the new template shown in fig. 4.8 even though this template mesh is different from the whole heart template mesh

Table 4.8: Ablation study – comparing mean (max) percentage self-intersecting faces on CT test data

Model	Epi	LA	LV	RA	RV	Ao	PA
Fl	0.07 (0.69)	0 (0)	0.007 (0.20)	0 (0)	0.17 (1.67)	0.36 (10.16)	4.97 (18.12)
LT	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
LT-Fl	0.0005 (0.01)	0 (0)	0 (0)	0.014 (0.58)	0 (0)	0.19 (2.64)	0.20 (5.41)
LT-Fl-v	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)

Table 4.9: Ablation study – comparing mean (max) percentage self-intersecting faces on MR test data

Model	Epi	LA	LV	RA	RV	Ao	PA
Fl	0.06 (1.58)	0.05 (1.35)	0.004 (0.14)	0.04 (0.49)	0.06 (1.95)	0.06 (1.78)	4.14 (21.8)
LT	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
LT-Fl	0.002 (0.06)	0.009 (0.37)	0 (0)	0.03 (1.12)	0.001 (0.04)	0.69 (4.89)	0.09 (0.82)
LT-Fl-v	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0.0008 (0.85)	0.02 (0.92)

used in training our model.

Table 4.10 shows a comparison of SIF % for the LT-FL and LT-FL-v models. Recall that the only difference between these two models is that the LT-FL-v additionally uses the volume loss eq. 4.5 during training. The LT-FL-v model on average reduces the number of SIF by over 90% for CT and 93% for MR compared to LT-FL with the median improvement being 100% for both modalities. We see that more than half the samples for CT (50%) and MR (62.5%) have *zero* SIF. This is clear evidence of the regularization effect provided by the volume loss and additionally helps prevent unphysical deformations as shown in fig. 4.12.

An important distinction between LT-FL and LT-FL-v is the kinds of SIFs they produce. Recall that SIF can result due to surface interpenetration or element inversion. Element inversion can often be fixed quite easily using standard surface remeshing techniques. Surface interpenetration is a significantly harder problem to fix and requires manual intervention or sophisticated contact detection algorithms. While the volume loss is effective at mitigating both of these types of SIF, it is particularly effective at mitigating surface interpenetration. Figure 4.9 illustrates a few test samples in which the LT-FL model collapses the thickness added to the surfaces of inlets and outlets of the LV resulting in significant surface interpenetration. The

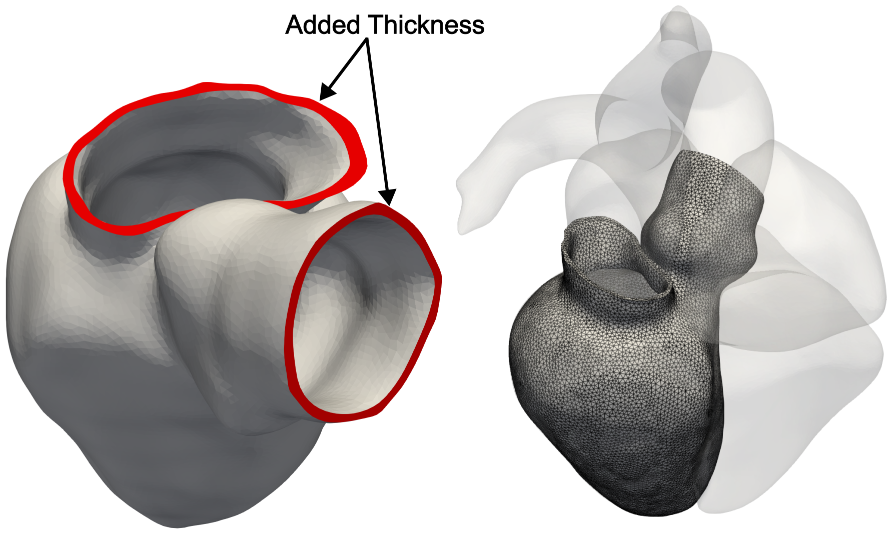


Figure 4.8: Template mesh of the left-ventricle myocardium with thickness added to inlets and outlets

Table 4.10: LV-template mesh quality on CT and MR test data

Modality	Metric	LT-FL	LT-FL-v	HeartDeformNet
CT	Avg. SIF (%)	0.109	0.003	0.134
	Median SIF (%)	0.078	0.002	0.112
	Max SIF (%)	0.731	0.013	0.54
	% Samples with 0% SIF	0	50	0
MR	Avg. SIF (%)	0.138	0.002	0.145
	Median SIF (%)	0.080	0	0.086
	Max SIF (%)	0.606	0.013	0.885
	% Samples with 0% SIF	2.5	62.5	0

LT-FL-v model is able to preserve this thickness. Figure 4.10 highlights the SIF in one of our test samples. We clearly see the different modalities of SIFs present in the meshes generated by the two models.

Since the SIF produced by the LT-FL-v model are due to element inversions, we are able to easily fix these issues in *all* of the cases using a simple isotropic remeshing

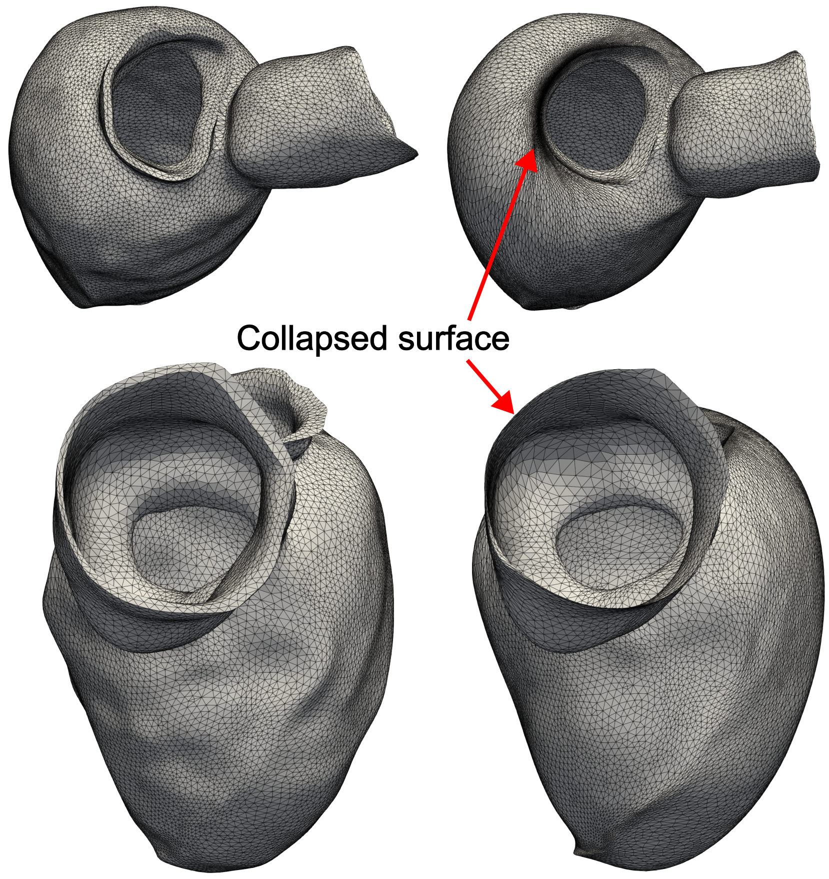


Figure 4.9: LT-FL (right) collapses the inlet/outlet surface. LT-FL-V (left) preserves the added thickness.

available in MeshLab [14]. We illustrate how remeshing can be used to fix element collapses in fig. 4.11. This strategy does not work for the meshes generated by the LT-FL model since the SIFs produced by this model are fundamentally different and not fixable by just remeshing. Thus, considering remeshing as a simple post-processing step, we are able to robustly generate simulation ready patient-specific meshes of the LV in *all* of our test samples.

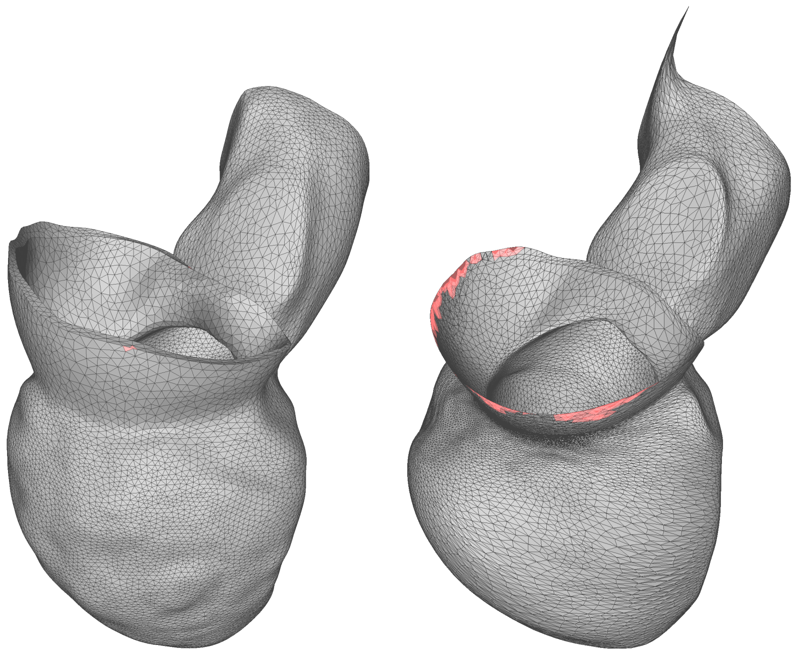


Figure 4.10: LT-FL-V (left) produces SIF primarily due to element collapse. LT-FL produces SIF due to surface interpenetration and element collapse. SIFs are highlighted in both images.

4.5 Discussion

We observe that HeartDeformNet [37] produces more accurate meshes when measured by the similarity metrics in Tables 4.2 and 4.3. However, as Tables 4.4 and 4.5 show, these meshes contain non-zero self-intersections that need to be handled by post-processing. In contrast, our proposed method demonstrates comparable accuracy while robustly producing meshes with *zero* self-intersections. The generated surface meshes can be readily used to produce full 3D volumetric meshes. Note that HeartDeformNet uses up to 3 mesh deformation blocks using graph convolutional networks. Similar to the results in [38], we anticipate that using a second flow deformation module in our workflow can help to improve our accuracy.

We demonstrated that our model trained on Whole-Heart mesh generation, may be deployed on a completely new template mesh of the myocardium along with its inlets and outlets that was not seen during training. By training our model with the volume loss eq. 4.5, we are able to generate meshes that maintain the thickness of

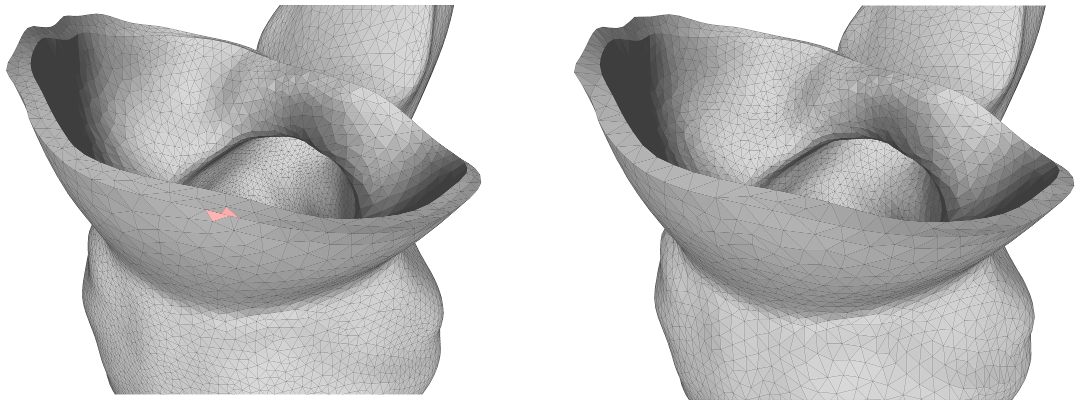


Figure 4.11: SIF due to element collapse (left) can be easily fixed by standard surface remeshing algorithms. Here we demonstrate the result of using the isotropic remeshing available in MeshLab.

cardiovascular tissue for which no ground-truth data is available. Our model robustly generates meshes with a very small number of SIF. Further, these SIFs are primarily due to element inversion/collapse as opposed to surface interpenetration, and may be easily fixed using standard surface remeshing algorithms. The volume loss is effective at regularizing unphysical deformations of the template mesh.

For cardiac structures where tissue thickness is not discernible, the model cannot be trained to match tissue thickness. However, the model is trained to match the lumen surface (inner boundary of the tissue) to the image data. For such structures, a tissue thickness can be prescribed in the template, which will deform according to the flow field that deforms the lumen to the image. While our volume loss constraint prevents collapse of the tissue thickness, hence preventing surface interpenetration, it does not directly guarantee a particular tissue thickness value in the final deformed configuration. Future work may explore further constraints during the deformation process so that tissue thickness in the final configuration matches a prescribed distribution.

The kinematics of continua is a mature discipline with a rigorous and rich theoretical base. We have seen that deep learning models can be constrained by our insights from continuum mechanics. The volume loss eq. 4.5 acts as a powerful physics-based regularization for the neural network. It is a robust metric to determine if a deformation field is physically realistic. By optimizing for this measure, our model is encouraged to learn deformations that are physically realistic resulting in meshes that are more immediately usable. Numerous relations similar to eq. 4.5

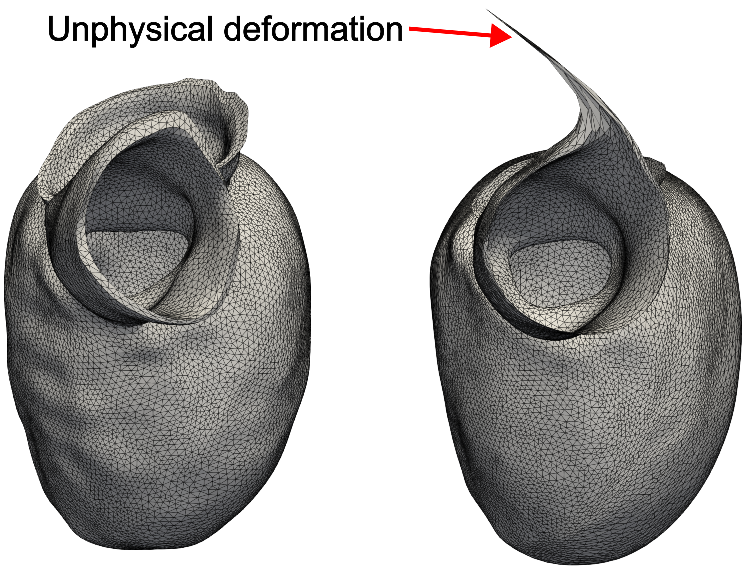


Figure 4.12: Volume loss is effective at regularizing unphysical deformations

can be found in the continuum mechanics literature, e.g. rate of change of lengths, areas, and normal vectors. For instance, the rate of change of an area element $A(t)$ with normal vector n due to deformation by a flow vector field v is,

$$\frac{\mathrm{d}A}{\mathrm{d}t} = A(\mathrm{div}(v) - n^T \nabla v \; n) \tag{4.6}$$

We believe that a loss function based on 4.6 can be used to prevent SIF due to element collapse as it effectively penalizes collapse modes in the plane of an element. This approach would be useful in situations where repeated remeshing is undesirable, e.g. time dependent motion of a mesh.

Template based methods have shown great promise in whole-heart mesh generation. However, there are some limitations. Firstly, templates enforce a specific topology onto the mesh. A given template is thus restricted in applicability to a given class of cardiac morphologies. When the morphology differs significantly e.g. due to congenital heart defect, we can no longer use the same template mesh. There are different solution approaches in these cases. We could consider the use of a library of template meshes depending on the target morphology. Alternatively, we could consider geometric representations, such as signed distance functions, that are able to handle changes in topology. Further, template based methods may not be effective in generating vascular meshes wherein there can be significant differences in geometry and topology due to branching and it is likely that a different strategy is required for vascular mesh generation. Our future work is focused on combining cardiac and vascular mesh generation to unify these two components into a single model.

4.6 Software

Source code for LinFlo-Net can be found at:
github.com/ArjunNarayanan/LinFlo-Net

Bibliography

[1] Rohan Abeyaratne. "Continuum mechanics". In: *Lecture Notes on The Mechanics of Elastic Solids* (1998).

[2] Takuya Akiba et al. "Optuna: A next-generation hyperparameter optimization framework". In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining.* 2019, pp. 2623–2631.

[3] Muhammad Naeem Akram, Kaoji Xu, and Guoning Chen. "Structure simplification of planar quadrilateral meshes". In: *Computers & Graphics* 109 (2022), pp. 1–14.

[4] S Alfonzetti. "A finite element mesh generator based on an adaptive neural network". In: *IEEE transactions on magnetics* 34.5 (1998), pp. 3363–3366.

[5] Adrià Puigdomènech Badia et al. "Never give up: Learning directed exploration strategies". In: *arXiv preprint arXiv:2002.06038* (2020).

[6] Yoshua Bengio et al. "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning.* 2009, pp. 41–48.

[7] James Bergstra et al. "Algorithms for hyper-parameter optimization". In: *Advances in neural information processing systems* 24 (2011).

[8] Ted D Blacker and Michael B Stephenson. "Paving: A new approach to automated quadrilateral mesh generation". In: *International journal for numerical methods in engineering* 32.4 (1991), pp. 811–847.

[9] David Bommes et al. "Quad-mesh generation and processing: A survey". In: *Computer graphics forum.* Vol. 32. Wiley Online Library. 2013, pp. 51–76.

[10] Fabian Bongratz, Anne-Marie Rickmann, and Christian Wachinger. "Meshes Meet Voxels: Abdominal Organ Segmentation via Diffeomorphic Deformations". In: *arXiv preprint arXiv:2306.15515* (2023).

[11] Marcel Campen, David Bommes, and Leif Kobbelt. "Quantized global parametrization". In: *Acm Transactions On Graphics (tog)* 34.6 (2015), pp. 1–12.

[12] Scott A Canann, SN Muthukrishnan, and RK Phillips. “Topological improvement procedures for quadrilateral finite element meshes”. In: *Engineering with Computers* 14.2 (1998), pp. 168–177.

[13] Xinhai Chen et al. “MGNet: a novel differential mesh generation method based on unsupervised neural networks”. In: *Engineering with Computers* 38.5 (2022), pp. 4409–4421.

[14] Paolo Cignoni et al. “MeshLab: an Open-Source Mesh Processing Tool”. In: *Eurographics Italian Chapter Conference*. Ed. by Vittorio Scarano, Rosario De Chiara, and Ugo Erra. The Eurographics Association, 2008. ISBN: 978-3-905673-68-5. DOI: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[15] Jorge Corral-Acero et al. “The ‘Digital Twin’to enable the vision of precision cardiology”. In: *European heart journal* 41.48 (2020), pp. 4556–4564.

[16] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[17] Joel Daniels et al. “Quadrilateral mesh simplification”. In: *ACM transactions on graphics (TOG)* 27.5 (2008), pp. 1–9.

[18] Zhi Deng et al. “Sketch2PQ: freeform planar quadrilateral mesh design via a single sketch”. In: *IEEE Transactions on Visualization and Computer Graphics* (2022).

[19] Alexander Dielen et al. “Learning direction fields for quad mesh generation”. In: *Computer Graphics Forum*. Vol. 40. 5. Wiley Online Library. 2021, pp. 181–191.

[20] Benjamin C DiPrete et al. “Reinforcement Learning for Block Decomposition of CAD Models”. In: *arXiv preprint arXiv:2302.11066* (2023).

[21] David P Dobkin and Michael J Laszlo. “Primitives for the manipulation of three-dimensional subdivisions”. In: *Proceedings of the third annual symposium on Computational geometry*. 1987, pp. 86–99.

[22] Julia Docampo-Sanchez and Robert Haimes. “Towards fully regular quad mesh generation”. In: *AIAA Scitech 2019 Forum*. 2019, p. 1988.

[23] Julia Docampo-Sánchez and Robert Haimes. “A regularization approach for automatic quad mesh generation”. In: *28th International Meshing Roundtable. Zenodo* (2020).

[24] Vladimir Dyedov et al. “AHF: Array-based half-facet data structure for mixed-dimensional and non-manifold meshes”. In: *Engineering with Computers* 31 (2015), pp. 389–404.

[25] Theresa Eimer, Marius Lindauer, and Roberta Raileanu. “Hyperparameters in reinforcement learning and how to tune them”. In: *International conference on machine learning*. PMLR. 2023, pp. 9104–9149.

[26] Lori A Freitag and Carl Ollivier-Gooch. “Tetrahedral mesh improvement using swapping and smoothing”. In: *International Journal for Numerical Methods in Engineering* 40.21 (1997), pp. 3979–4002.

[27] Kunal Gupta. *Neural mesh flow: 3d manifold mesh generation via diffeomorphic flows*. University of California, San Diego, 2020.

[28] Tuomas Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[29] Si Hang. “TetGen, a Delaunay-based quality tetrahedral mesh generator”. In: *ACM Trans. Math. Softw* 41.2 (2015), p. 11.

[30] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[31] Keefe Huang et al. “Machine learning-based optimal mesh generation in computational fluid dynamics”. In: *arXiv preprint arXiv:2102.12923* (2021).

[32] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. “An efficient approach for assessing hyperparameter importance”. In: *International conference on machine learning*. PMLR. 2014, pp. 754–762.

[33] Rashed Karim et al. “Algorithms for left atrial wall segmentation and thickness–Evaluation on an open-source CT and MRI image database”. In: *Medical image analysis* 50 (2018), pp. 36–53.

[34] Bryan Matthew Klingner and Jonathan Richard Shewchuk. “Aggressive tetrahedral mesh improvement”. In: *Proceedings of the 16th international meshing roundtable*. Springer. 2007, pp. 3–23.

[35] Fanwei Kong and Shawn C Shadden. “Automating model generation for image-based cardiac flow simulation”. In: *Journal of Biomechanical Engineering* 142.11 (2020), p. 111011.

[36] Fanwei Kong and Shawn C. Shadden. "Learning Whole Heart Mesh Generation From Patient Images for Computational Simulations". In: *IEEE Transactions on Medical Imaging* 42 (2022), pp. 533–545. URL: `https://api.semanticscholar.org/CorpusID:247593788`.

[37] Fanwei Kong, Nathan Wilson, and Shawn Shadden. "A deep-learning approach for direct whole-heart mesh reconstruction". In: *Medical image analysis* 74 (2021), p. 102222.

[38] Leo Lebrat et al. "CorticalFlow: a diffeomorphic mesh transformer network for cortical surface reconstruction". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 29491–29505.

[39] Franck Ledoux and Jason Shepherd. "Topological modifications of hexahedral meshes via sheet operations: a theoretical study". In: *Engineering with Computers* 26 (2010), pp. 433–447.

[40] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems.* SIAM, 2007.

[41] Richard Liaw et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018).

[42] Isaak Lim et al. "A simple approach to intrinsic correspondence learning on unstructured 3d meshes". In: *Proceedings of the European conference on computer vision (ECCV) workshops.* 2018.

[43] Qiang Ma et al. "CortexODE: Learning Cortical Surface Reconstruction by Neural ODEs". In: *IEEE Transactions on Medical Imaging* 42.2 (2022), pp. 430–443.

[44] de Berg Mark et al. *Computational geometry algorithms and applications.* Springer, 2008.

[45] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[46] Steven J Owen et al. "Q-Morph: an indirect approach to advancing front quad meshing". In: *International journal for numerical methods in engineering* 44.9 (1999), pp. 1317–1340.

[47] Daniel H Pak et al. "Distortion energy for deep learning-based volumetric finite element mesh generation for aortic valves". In: *Medical Image Computing and Computer Assisted Intervention–MICCAI 2021: 24th International Conference, Strasbourg, France, September 27–October 1, 2021, Proceedings, Part VI 24*. Springer. 2021, pp. 485–494.

[48] Jie Pan et al. “A self-learning finite element extraction system based on reinforcement learning”. In: *AI EDAM* 35.2 (2021), pp. 180–208.

[49] Jie Pan et al. “Reinforcement learning for automatic quadrilateral mesh generation: A soft actor–critic approach”. In: *Neural Networks* 157 (2023), pp. 288–304.

[50] Alexis Papagiannopoulos, Pascal Clausen, and Francois Avellan. “How to teach neural networks to mesh: Application on 2-D simplicial contours”. In: *Neural Networks* 136 (2021), pp. 152–179.

[51] Christian Payer et al. “Multi-label whole heart segmentation using CNNs and anatomical label configurations”. In: *International Workshop on Statistical Atlases and Computational Models of the Heart*. Springer. 2017, pp. 190–198.

[52] Jaime Peraire et al. “Adaptive remeshing for compressible flow computations”. In: *Journal of computational physics* 72.2 (1987), pp. 449–466.

[53] Adityo Prakosa et al. “Personalized virtual-heart technology for guiding the ablation of infarct-related ventricular tachycardia”. In: *Nature biomedical engineering* 2.10 (2018), pp. 732–740.

[54] Antonin Raffin et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: `http://jmlr.org/papers/v22/20-1364.html`.

[55] Marie-Julie Rakotosaona et al. “Differentiable surface triangulation”. In: *ACM Transactions on Graphics (TOG)* 40.6 (2021), pp. 1–13.

[56] Nikhila Ravi et al. “Accelerating 3D Deep Learning with PyTorch3D”. In: *arXiv:2007.08501* (2020).

[57] J.-F. Remacle et al. “Blossom-Quad: a non-uniform quadrilateral mesh generator using a minimum-cost perfect-matching algorithm”. In: *Internat. J. Numer. Methods Engrg.* 89.9 (2012), pp. 1102–1119. ISSN: 0029-5981,1097-0207. DOI: `10.1002/nme.3279`. URL: `https://doi.org/10.1002/nme.3279`.

[58] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer. 2015, pp. 234–241.

[59] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit (4th ed.)* Kitware, 2006. ISBN: 978-1-930934-19-1.

[60] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[61] Jonathan Richard Shewchuk. "Delaunay refinement algorithms for triangular mesh generation". In: *Computational geometry* 22.1-3 (2002), pp. 21–74.

[62] Jonathan Richard Shewchuk. "Two discrete optimization algorithms for the topological improvement of tetrahedral meshes". In: *Unpublished manuscript* 65 (2002), pp. 2–7.

[63] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[64] Jeffrey P Slotnick et al. *CFD vision 2030 study: a path to revolutionary computational aerosciences*. Tech. rep. 2014.

[65] Tomasz Służalec et al. "Quasi-optimal hp-finite element refinements towards singularities via deep neural network prediction". In: *Computers & Mathematics with Applications* 142 (2023), pp. 157–174.

[66] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[67] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (1999).

[68] Marco Tarini et al. "Practical quad mesh simplification". In: *Computer Graphics Forum*. Vol. 29. Wiley Online Library. 2010, pp. 407–418.

[69] Timothy J Tautgesa and Sarah E Knoopb. "Topology modification of hexahedral meshes using atomic dual-based operations". In: *algorithms* 11 (2003), p. 12.

[70] Wu Tingfan et al. "A mesh optimization method using machine learning technique and variational mesh adaptation". In: *Chinese journal of aeronautics* 35.3 (2022), pp. 27–41.

[71] Catalina Tobon-Gomez et al. "Benchmark for algorithms segmenting the left atrium from 3D CT and MRI datasets". In: *IEEE transactions on medical imaging* 34.7 (2015), pp. 1460–1473.

[72] Mark Towers et al. *Gymnasium*. Mar. 2023. DOI: `10.5281/zenodo.8127026`. URL: `https://zenodo.org/record/8127025` (visited on 07/08/2023).

[73] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". In: *Advances in neural information processing systems* 28 (2015).

[74] Nanyang Wang et al. "Pixel2Mesh: 3D mesh model generation via image guided deformation". In: *IEEE transactions on pattern analysis and machine intelligence* 43.10 (2020), pp. 3600–3613.

[75] Jakob Wasserthal et al. "TotalSegmentator: Robust segmentation of 104 anatomic structures in CT images". en. In: *Radiol. Artif. Intell.* 5.5 (Sept. 2023), e230024.

[76] Udaranga Wickramasinghe et al. "Voxel2mesh: 3d mesh model generation from volumetric data". In: *Medical Image Computing and Computer Assisted Intervention–MICCAI 2020: 23rd International Conference, Lima, Peru, October 4–8, 2020, Proceedings, Part IV 23*. Springer. 2020, pp. 299–308.

[77] Jelmer M Wolterink et al. "An evaluation of automatic coronary artery calcium scoring methods with cardiac CT using the orCaScore framework". In: *Medical physics* 43.5 (2016), pp. 2361–2373.

[78] Jiachen Yang et al. "Reinforcement learning for adaptive mesh refinement". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2023, pp. 5997–6014.

[79] S Yao et al. "An ANN-based element extraction method for automatic mesh generation". In: *Expert Systems with Applications* 29.1 (2005), pp. 193–206.

[80] Zheyan Zhang et al. "MeshingNet: A new mesh generation method based on deep learning". In: *International conference on computational science*. Springer. 2020, pp. 186–198.

[81] Qingnan Zhou. *PyMesh*. `https://github.com/PyMesh/PyMesh`. 2020.

[82] Xiahai Zhuang et al. "Evaluation of algorithms for multi-modality whole heart segmentation: an open-access grand challenge". In: *Medical image analysis* 58 (2019), p. 101537.