Solving Jigsaw Puzzles with Deep Learning for Heritage

Marie-Morgane Paumard

SOLVING JIGSAW PUZZLES WITH DEEP LEARNING

FOR HERITAGE

Marie-Morgane Paumard

December 14th, 2020

A dissertation submitted for the degree of
Doctor of Philosophy from CY Cergy Paris University

École doctorale n°405 EM2PSI
Économie, Management, Mathématiques, Physique & Sciences Informatiques

David Picard	Senior Research Scientist, École des Ponts ParisTech	Supervisor
Hedi Tabia	Full Professor, Université d'Évry	Co-Supervisor
Vivien Barrière	Associate Professor, CY Cergy Paris Université	Advisor
Aurélie Bugeau	Associate Professor, Université de Bordeaux	Reviewer
Vincent Lepetit	Senior Research Scientist, École des Ponts ParisTech	Reviewer
Vicky Kalogeiton	Associate Professor, École Polytechnique	Examiner
Blaise Hanczar	Full Professor, Université d'Évry	Examiner
Nicolas Thome	Full Professor, Conservatoire national des arts et métiers	Examiner

Colophon This document was typeset in LaTeX, using the beautiful `tufte-latex` class[1] and a hand-made overlay inspired by Aaron Turon's thesis, *Understanding and expressing scalable concurrency*. Firmin Didot's `GFS Didot` acts as the typeface. The bibliography is typeset using `biblatex`.

[1] `tufte-latex` is based on on the work of the famous statistician Edward Tufte.

First printing, October 2020

Acknowledgements

Avant toute chose, j'aimerai exprimer ma gratitude envers tous ceux qui ont contribué à cette thèse.

Je remercie les membres de mon jury : Aurélie Bugeau, Vincent Lepetit, Vicky Kalogeiton, Blaise Hanczar et Nicolas Thome, pour avoir généreusement offert leurs temps et leurs conseils. Je garde un excellent souvenir de ma soutenance et c'est en grande partie grâce à vous tous.

Ma seconde pensée va à David Picard : son soutien sans faille, son humilité, ses connaissances, sa sollicitude, sa manière de pousser ses doctorants à se surpasser, sa grandeur d'âme, font de lui un directeur de thèse exceptionnel. Dix pages ne suffiraient pas à dresser un portrait fidèle ni à exprimer ma reconnaissance pour tout ce qu'il m'a apporté.

Viennent ensuite mon co-directeur de thèse, Hedi Tabia, et mon encadrant, Vivien Barrière, qui ont toujours répondu présents quand j'ai eu besoin d'eux. D'autres, nombreux, ont contribué directement à cette thèse : pour leurs conseils bienveillants, Eduardo Valle et Vincent Lepetit ; pour m'avoir permis de découvrir tant de choses sur le patrimoine, la Fondation des Sciences du Patrimoine, et notamment Anne-Julie Etter et Emmanuel Poirault ; pour ce projet si enrichissant, l'équipe du projet ARCHEPUZ'3D, et en particulier Michel Jordan et Thomas Sagory ; pour leur assistance efficace, Laurent Protois, Annick Bertinotti et Isabelle Simunic. Je remercie également tous les permanents des équipes du laboratoire ETIS et d'IMAGINE, avec qui j'ai toujours eu des discussions plaisantes et intéressantes.

Durant ces trois années, j'ai été merveilleusement bien entourée par mes camarades de laboratoire. Je peine à mesurer cette chance que d'avoir pu rencontrer tant d'êtres exceptionnels, que ce soit par leurs nombreux talents, leur conversation plaisante ou leur admirable caractère : Pierre Jacob, M. Amine Khelif, Diogo C. Luvizon, Alexandre Marcastel, Juline Camps, Habiba Ladhiri, Louis Desportes, Marwa Dammak et Louis Annabi à ETIS, ainsi qu'Abderahmane Bedouhene, Thomas Belos, Victor Besnier, Robin Champenois, Philippe Chiberre, François Darmon, Théo Deprelle, Yuming Du, Rahima Djahel, Thibault Groueix, Shell Xu Hu, Timothée Lacroix, Pierre-Alain Langlois, Thomas Luka, Tom Monnier, Giorgia Pitteri, Xuchong Qiu, Michaël Ramamonjisoa, Clément Riu, Othman Sbai, Xi Shen, Yang Xiao à IMAGINE, ainsi que Thomas Robert, au LIP6.

Pendant mon parcours, j'ai également rencontré des femmes de sciences admirables. Par leur talent, leur passion et leur dévouement à la recherche, certaines sont pour moi un exemple à suivre : Gentiane Venture, E. Veronica Belmega, Iryna Andriyanova, Inbar Filjacob, Marwa Chafii et Geneviève Pinçon. D'autres m'ont offert une oreille attentive et un soutien inconditionnel : mes co-lauréates du Prix L'Oréal-UNESCO Jeune Talent 2020 m'aident à combler mon manque de confiance en moi et j'espère entretenir une amitié durable avec chacune de ces jeunes femmes merveilleuses.

Ensuite, je remercie les professeurs et encadrants qui m'ont encouragée à poursuivre dans la recherche, sans lesquels mon parcours aurait été bien différent. Par ordre chronologique : Laurent Guitard, David Pichardie, David Cachera, Luc Bougé, Clément Moulin-Frier, Pierre Rouanet, Pierre-Yves Oudeyer, Gentiane Venture, Olivier Sigaud, Jérôme Lesueur, Julie Iem et Claire Ripault-Blandin. Mes études m'ont permis de rencontrer beaucoup de futurs chercheurs et c'est également grâce à leur exemple que j'ai effectué mon doctorat : la promotion Info 2013 de l'ENS Rennes et la promotion #3 de PSL-ITI.

Enfin, sans le support de ma famille et de mes amis, il m'aurait été très difficile de compléter cette thèse. Je remercie mes parents pour leur éducation, leur amour et pour m'avoir offert un excellent cadre pour la fin de mon doctorat en confinement, ma mamie qui avait tant envie de me voir docteure, ainsi que mes grands-parents maternels, mon frère, ma sœur et ma belle-famille pour leur soutien ; mes élèves de la danse pour me donner la chance de partager ma passion et pour croire autant en moi ; mes meilleurs amis pour être toujours disponibles pour moi et mon fiancé, Jules Brochard, pour avoir pris si bien soin de moi tout en menant ses propres travaux de doctorat, pour être à mes côtés et m'aimer autant.

Résumé

L'objectif de cette thèse est de développer des méthodes sémantiques de réassemblage dans le cadre compliqué des collections patrimoniales, où certains blocs sont érodés ou manquants.

Le remontage de vestiges archéologiques est une tâche importante pour les sciences du patrimoine : il permet d'améliorer la compréhension et la conservation des vestiges et artefacts anciens. Certains ensembles de fragments ne peuvent être réassemblés grâce aux techniques utilisant les informations de contour et les continuités visuelles. Il est alors nécessaire d'extraire les informations sémantiques des fragments et de les interpréter. Ces tâches peuvent être accomplies automatiquement grâce aux techniques d'apprentissage profond couplées à un solveur, c'est-à-dire un algorithme de prise de décision sous contraintes.

Cette thèse propose deux méthodes de réassemblage sémantique pour fragments 2D avec érosion, ainsi qu'un jeu de données et des métriques d'évaluation.

La première méthode, Deepzzle, propose un réseau de neurones auquel succède un solveur. Le réseau de neurones est composé de deux réseaux convolutionnels siamois entraînés à prédire la position relative de deux fragments : il s'agit d'une classification à neuf classes. Le solveur utilise l'algorithme de Dijkstra pour maximiser la probabilité jointe. Deepzzle peut résoudre le cas de fragments manquants et surnuméraires, est capable de traiter une quinzaine de fragments par puzzle, et présente des performances supérieures à l'état de l'art de 25 %.

La deuxième méthode, Alphazzle, s'inspire d'AlphaZero et de recherche arborescente Monte Carlo (MCTS) à un joueur. Il s'agit d'une méthode itérative d'apprentissage profond par renforcement : à chaque étape, on place un fragment sur le réassemblage en cours. Deux réseaux de neurones guident le MCTS : un prédicteur d'action, qui utilise le fragment et le réassemblage en cours pour proposer une stratégie, et un évaluateur, qui est entraîné à prédire la qualité du résultat futur à partir du réassemblage en cours. Alphazzle prend en compte les relations entre tous les fragments et s'adapte à des puzzles de taille supérieure à ceux résolus par Deepzzle. Par ailleurs, Alphazzle se place dans le cadre patrimonial : en fin de réassemblage, le MCTS n'accède pas à la récompense, contrairement à AlphaZero. En effet, la récompense, qui indique si un puzzle est bien résolu ou non, ne peut être qu'estimée par l'algorithme, car seul un conservateur peut être certain de la qualité d'un réassemblage.

Mots-clés Apprentissage profond, apprentissage par renforcement, décision par parcours de graphe, recherche arborescente Monte Carlo, puzzles, sciences du patrimoine.

Abstract

This thesis aims to develop semantic methods of reassembly in the complicated framework of heritage collections, where some blocks are eroded or missing.

The reassembly of archaeological remains is an essential task for heritage sciences: it improves the understanding and conservation of ancient vestiges and artifacts. However, some sets of fragments cannot be reassembled with techniques using contour information or visual continuities. It is then necessary to extract semantic information from the fragments and to interpret them. These tasks can be performed automatically thanks to deep learning techniques coupled with a solver, i.e., a constrained decision-making algorithm.

This thesis proposes two semantic reassembly methods for 2D fragments with erosion, as well as a new dataset and evaluation metrics.

The first method, Deepzzle, proposes a neural network followed by a solver. The neural network is composed of two Siamese convolutional networks trained to predict the relative position of two fragments: it is a 9-class classification. The solver uses Dijkstra's algorithm to maximize the joint probability. Deepzzle can address the case of missing and supernumerary fragments. It can process about 15 fragments per puzzle and outperforms state of the art by 25%.

The second method, Alphazzle, is based on AlphaZero and single-player Monte Carlo Tree Search (MCTS). It is an iterative method that uses deep reinforcement learning: at each step, a fragment is placed on the current reassembly. Two neural networks guide MCTS: an action predictor, which uses the fragment and the current reassembly to propose a strategy, and an evaluator trained to predict the quality of the future result from the current reassembly. Alphazzle considers the relationships between all fragments and adapts to puzzles larger than those solved by Deepzzle. Moreover, Alphazzle is compatible with constraints imposed by a heritage framework: at the end of reassembly, MCTS does not access the reward, unlike AlphaZero. Indeed, the reward, which indicates if a puzzle is well solved or not, can only be estimated by the algorithm because only a conservator can be sure of a reassembly quality.

Keywords Deep learning, reinforcement learning, decision theory with graph traversal, single-player Monte Carlo Tree Search, jigsaw puzzles, heritage.

Publications

This dissertation draws heavily on earlier work and writing in the following papers:

Journals

- [PPT20] Marie-Morgane Paumard, David Picard, and Hedi Tabia (2019). Deepzzle: Solving Visual Jigsaw Puzzles with Deep Learning. In IEEE Transactions on Image Processing (TIP);
- Marie-Morgane Paumard (2020). Remonter un site archéologique à partir de fragments : cas du sanctuaire des Vaux de la Celle (fr). In Technè (forthcoming publications);

Refereed conferences

- Marie-Morgane Paumard, David Picard, and Hedi Tabia (2020). Solving Jigsaw Puzzle with Deep Monte-Carlo Tree Search. In submission;
- [PPT18a] Marie-Morgane Paumard, David Picard, and Hedi Tabia (2018). Image Reassembly Combining Deep Learning and Shortest Path Problem. In Proceedings of the European Conference on Computer Vision (ECCV);
- [PPT18b] Marie-Morgane Paumard, David Picard, and Hedi Tabia (2018). Jigsaw Puzzle Solving Using Local Feature Co-Occurrences in Deep Neural Networks. In Proceedings of the IEEE International Conference on Image Processing (ICIP);

Non-refereed conference

- Marie-Morgane Paumard, David Picard, and Hedi Tabia (2019). L'apprentissage profond pour le réassemblage d'images patrimoniales (fr). In proceedings of the *Colloque francophone de traitement du signal et des images* (GRETSI).

Artwork 1: *Oedipus and the Sphinx*, Gustave Moreau, 1864, from the MET Open Collections.

Contents

List of Artworks

List of Figures

List of Tables

List of Terms and Acronyms

GLOSSARY

Puzzle-solving A subtask of the reassembly task that aims to find the fragments' coarse positioning. 6

Reassembly (a ~) Any output of a puzzle-solving algorithm. 6, 7

Solution The intended reassembly. 6

ACRONYMS

ADI Autodidactic Iteration. 55

ExIt Expert Iteration. 54

FEN Feature Extraction Networks. 25, 26, 33, 34

MCTS Monte-Carlo Tree Search. 5, 52, 53
MET Metropolitan Museum of Art. 4, 14
MSE Mean squared error. 66

NMCS Nested Monte-Carlo Search. 54
NMCTS Nested Monte-Carlo Tree Search. 54
NRPA Nested Rollout Policy Adaptation. 54

PUCB Predictor + Upper Confidence Bound. 62
PUCT Predictor + Upper Confidence Bound for Tree. 62, 63, 67, 81

R2 Ranked-Rewards. 55
RANSAC Random Sample Consensus. 9

SGD Stochastic Gradient Descent. 33, 34
SP-MCTS Single-Player MCTS. 53

UCB Upper Confidence Bounds. 53, 62
UCT Upper Confidence Bounds for Trees. 62

WRN WideResNet. 66

Part I

PROLOGUE

1

Overview

Chapter 2 ➤

1.1 REASSEMBLY FOR HERITAGE

The problem of reassembly is shared by many archaeological sites. When many fragments are discovered, archaeologists face a gigantic 3D jigsaw puzzle with damaged or even missing pieces. For instance, the sculpted ceiling of the prehistoric rock-shelter of Roc-aux-Sorciers (Angles-sur-l'Anglin, Vienne, France) collapsed into a thousand pieces (Figure 1.1).

Roc-aux-Sorciers was occupied by the Magdalenians 16,000 years ago and is composed of two distinct sections: the Bourdois shelter, a classic rock-shelter site beneath a slight overhang, and the Taillebourg cave, a typical vestibule. Both are entirely sculpted, but only the Bourdois shelter's frieze has been preserved, while the cave's bas-reliefs are scattered in a multitude of pieces. The research directed by Geneviève Pinçon focuses on the parietal art and the material culture (mostly jewelry and tools)[1] and studies the normative system of animal representation, the cave as a habitat, the relationship to art, and the function and activities associated with the tools. If the Taillebourg cave were rebuilt, many questions would be answered. To this end, 3D geomorphological reconstructions and blocks digitization are in progress. Some reassembly tests have been carried out on joints identified by archaeologists, but the complexity of the calculations impedes the ceiling reconstruction.

Another famous archaeological monument to be rebuilt is the temple of the Vaux-de-la-Celle Gallo-Roman sanctuary (Genainville, Val d'Oise, France). The site consists of a two-*cella* temple's ruins (Figure 1.2) surrounded by a circulation gallery, a theatre that can accommodate up to 8,000 people, and four sacred basins. The first *fanum*[2] is dated to the middle of the 1st century CE, and the architectural ensemble was built in the second half of the 2nd century. During the 3rd century, the site was gradually abandoned. Therefore, the carved blocks of the temple served as a limestone quarry until the modern era. They are now gathered in the reserves of the *Musée archéologique du Val-d'Oise*, for their reassembly. About sixty blocks have been digitized through photogrammetry and 3D scan.

Many other reconstruction projects are conducted all around the world. To name a few, we mention Angkor Wat (Cambodia), Djoser funerary complex (Egypt), Huaca Pucllana (Peru), Notre-Dame-de-Paris (France), Parthenon (Greece), and Troy's Odeion (Turkey).

Anastylosis is the archaeological term that refers to the reconstruction of a monument using the original material. Learn about it on Wikipedia.

Figure 1.1: The lapidary of the Taillebourg cellar. © G. Pinçon, Ministry of Culture (France), C. Archambeau.

[1] *Le Roc-aux-Sorciers: art et parure du Magdalénien (fr)*, under the direction of G. Pinçon (catalog).

Figure 1.2: The ruins of the Vaux-de-la-Celle Gallo-Roman temple.

[2] A *fanum* is a Gallo-Roman temple.

In most cases, software has been developed to help the conservators find the match between blocks. Usually, the search for correspondences is done manually or is semi-automated[3]. Such software seeks to associate the contours or the blocks' visual continuities, which is sometimes enough to obtain a coherent reconstruction. However, some reassemblies depend on the painted or carved representations' semantics[4], and no software can manage them yet. Understanding semantics appears to be the next milestone in reassembly algorithms.

In recent years, deep-learning-based algorithms learned to use semantic features to perform different tasks, and they are now beginning to perform easy 2D reassemblies from square fragments [DGE15, NF16]. We, too, focus on solving a 2D square jigsaw puzzle. We motivate this choice in Section §4.1. The long-term goal is to develop a versatile method that can reassemble artifacts and monuments without relying on expert knowledge.

[3] Manually means that the conservators make the matches by themselves. Semi-automated means that the conservators pick plausible matches from the software-generated proposals.

[4] We can distinguish three levels of reassembly abilities. The first one is based solely on visual clues, such as patterns. The second one is based on the semantics, which means that the reassembly should be plausible, such as placing the sky above the ground. The last one uses logic to create pertinent stories, such as cooking before eating.

1.2 MAIN CONTRIBUTIONS

This dissertation aims to improve state-of-the-art reassembly methods along with two approaches: first, by *comparing* all the blocks to a significant block, ordering them by probable positions, and thus minimizing the joint probability to compose a reassembly. Second, by *iteratively placing* one block after another to use all the placed blocks to build a better reassembly. To that end, we design two algorithms:

1.2.1 *Pairwise comparison: Deepzzle*

While iterative solving is probably the closest to how we solve a puzzle, the pairwise comparison is what we would be doing if our executive memory was limited. It consists of comparing every fragment to every other fragment, so we only see two fragments simultaneously. Then, we assess all the pairs' relevance, and we deduce the reassembly from the pairs sorted with respect to this metric.

Doersch et al.[5] proposed the first evaluation of the fragments pairs with deep learning. We extended their work with a reassembly ability. Our contribution includes:

- A refined neural network architecture and merging function;
- Three reassembly solving methods (greedy, exact and heuristic);
- Various results such as solving 3×3 puzzles with missing or extra fragments, or fragments photographed under different lightning.

[5] [DGE15] C. Doersch, A. Gupta, and A.A. Efros, Unsupervised visual representation learning by context prediction.

In the rest of this dissertation, we refer to this method as Deepzzle.

1.2.2 *Iterative solving: Alphazzle*

Iterative solving consists of recursively comparing a fragment to the current partial reassembly, from which we place the fragment and

update the reassembly. It is what most people do when solving a jigsaw puzzle. We propose the first puzzle-solving method based on such iterative solving. We take our inspiration from AlphaZero[6], and we introduce some significant changes. Our contribution includes:

- A deep MCTS that proceeds visual features;
- A reassembly method that estimates the game reward, because it cannot be accessed directly;
- Extended results on bigger puzzles such as 5×5 puzzles.

[6] [SHS+17] D. Silver et al., Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.

In the rest of this dissertation, we refer to this method as Alphazzle.

1.2.3 *Other contributions*

Our other contribution includes:

- Few metrics assessing the pairs and the reassembly quality;
- A new dataset of 14,000 heritage images;
- An open-source code[7].

[7] GitHub repositories for Deepzzle and Alphazzle.

1.3 Organization of the dissertation

Chapter 2 gives an overview of the puzzle-solving task. After presenting the terminology, we list the jigsaw puzzle-solving tasks' variations, e.g., solving with missing fragments. We discuss the metrics used to evaluate the output of the algorithms. The last section deals with the applications of puzzle-solving across various research areas, especially in heritage.

Chapter 3 presents the computer-vision-based methods to make reassemblies from fragments without relying on deep learning.

Chapter 4 introduces the datasets we use. We start by explaining our choices on the fragments' shape and detailing the requirements for building the dataset. Then, we present the MET dataset, on which we trained our neural networks. Last, we introduce the datasets we used for fine-tuning and evaluating our models.

The rest of the dissertation is composed of the two proposed approaches:

Pairwise comparison with deep learning

Chapter 5 draws up a state of the art of puzzle-solving with deep learning. It introduces the two major methods for puzzle-solving: the pairwise comparison and the permutations. We highlight their strengths and weaknesses and justify our algorithm design.

Chapter 6 describes Deepzzle, our method for pairwise comparison puzzle-solving.

Chapter 7 walks through Deepzzle's major results, as well as the extensive results on some peculiar tasks and heritage datasets.

Iterative solving with deep reinforcement learning

Chapter 8 introduces Monte-Carlo Tree Search (MCTS) for (deep) reinforcement learning. We focus on single-player games.

Chapter 9 details Alphazzle, our method for iterative puzzle-solving relying on deep reinforcement learning.

Chapter 10 brings together our results.

Finally, the dissertation concludes with Chapter 11, which summarizes the contributions and suggests a few additional research ideas.

General appendices

Appendix A is an introduction to deep learning.

Appendix B provides basic knowledge on decision science.

Artwork 2: Detail of the Bourdois shelter's frieze, © G. Pinçon, A. Frich.

2

Introduction to the puzzle-solving task

◂ Chapter 1 Chapter 3 ▸

Synopsis This chapter describes what is the puzzle-solving task §2.1, draws up a task typology §2.2, and presents the metrics to evaluate the correctness of a reassembly §2.3. Last, it lists the applications of reassembly §2.4.

2.1 Terminology

First and foremost, we can divide the reassembly task into two stages: the coarse positioning, which we call puzzle-solving, and the precise reassembly of the blocks. The latter includes feature correspondence refinement, matching [OBA20] and object completion, such as extrapolation and inpainting [MRS10, TKAM15] (Figure 2.1). As this task does not require semantics understanding and can be performed more easily by human experts—provided that the coarse positioning is known—we decide to focus on puzzle-solving solely.

Figure 2.1: A virtual object completion. © D. Tsiafaki et al. [TKAM15].

In the remainder of this dissertation, we used puzzle-solving to describe the task and reassembly to describe the output of the puzzle solvers. We call solution the correct reassembly, i.e., the reassembly that we aim to compute.

2.2 Type of tasks

There is no standard puzzle-solving task, and all authors propose their variation. In this subsection, we present all the parameters and variations that we have read about[1]:

Fragments shape: The standard task for contour-based solving implies various size 2D or 3D fragments (Figure 2.2), while content-based solving tends to remove the fractures by cutting all pieces square to focus on the content solely. Gur and Ben-Shahar [GBS17] introduce a variation on the standard square shape and solve "brick wall" jigsaw puzzles (rectangles of the same height but various length), using a similar pipeline to Paikin and Tal [PT15].

Fragments quantity: The content-based methods can solve puzzles of several thousand pieces, while the contour-based methods use a few dozen fragments in 3D and a hundred or so in 2D.

Binding puzzle sizes: Sometimes, the puzzle size and shape are known, which implies that the fragments cannot be placed out of the puz-

[1] Most articles mentioned are presented in Chapter 3.

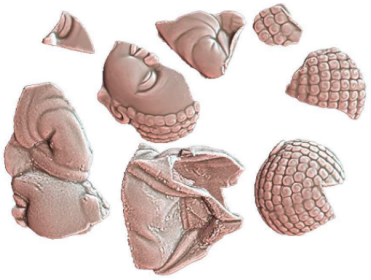


Figure 2.2: Some input fragments for a child Buddha reassembly from contours. © K. Zhang et al. [ZYM+15].

zle. The size can be strongly binding, which means that the first fragment's correct place is unique, or lightly binding, which means that the placed fragments can be moved within the puzzle borders. In contrast, when the puzzle sizes are unknown, each fragment can be placed anywhere.

Rotation: In the case of square fragments, they can be well-oriented [SDN13] or require rotation to solve the puzzle [Gal12, SHC14]. Some authors address the case of known position but unknown rotation, as pictured in Figure 2.3. When the fragments are not square, the rotation value has to be found by the algorithm.

Figure 2.3: Example of input in the case the position is known but the rotation is unknown. © A.C. Gallagher [Gal12].

Anchor piece: Anchor piece refers to the recommended first fragment. Usually, it comes with strongly binding puzzle sizes, and the solver is given the anchor piece position.

Erosion: Erosion refers to the reduction and smoothing of borders so that fractured regions cannot be of use anymore. It can be as light as a few pixels removal [DTS18] or very strong as half of the fragment size suppression.

Missing fragments: Sometimes, all the fragments to complete the puzzle are not available [PT15, SF17] (Figure 2.4). Inpainting and other objects completion techniques can be used to complete the obtained partial reassembly.

Figure 2.4: Example of reassembly with missing pieces, on a large 22k pieces puzzle. © G. Paikin and A. Tal [PT15].

Mixed fragments: Some work tackle mixed fragments originating from several objects. The algorithms have to exclude irrelevant fragments to solve the puzzle. Other algorithms can solve several puzzles from mixed input, such as [SHC14].

2.3 EVALUATION

We introduce three standard metrics to evaluate a reassembly quality:

Solved puzzles: This measure indicates the percentage of correctly solved puzzles. A puzzle is correctly solved when all its fragments are placed in the right position.

Well-placed fragments: This metric shows the average percentage of correctly placed fragments per puzzle. Note that in the case of no missing nor mixed fragmented, the number of mistakes must be null or strictly greater than 1.

Correct neighbors: This metric computes the fraction of correct pairwise adjacencies. It can work with cardinal neighbors only (up, down, right, left), full neighbors (cardinal neighbors and intercardinal neighbors), and extended neighbors (all fragments within a given range. The correct neighbors metric is usually higher than the well-placed fragments. As the only relative metric, it is very effective in the case where all fragments are shifted in one direction (Figure 2.5).

Figure 2.5: Example of a shifted solution: solved puzzle and well-placed fragments return 0% while correct neighbors metric is close to 100%. © D. Sholomon et al. [SDN13].

In some papers, other metrics have been introduced to counteract some data bias. For instance, the *almost-perfectly solved puzzles* metric is a solved puzzles metric that allows the inversion of almost identical fragments. Their similarity can be easy to compute (e.g., pixel-wise comparison) or rely on more advanced concepts (e.g., on the features).

2.4 APPLICATIONS OF REASSEMBLY

Reassembly is a very specific problem, and its impact is fairly limited to niche applications. These applications include, among other things:

- archaeology, such as the reconstruction of ancient artifacts;
- cryptography, such as attack encrypted images [CKK17];
- forensic medicine, such as skull assembly (Figure 2.6) [YWLM11];
- forensic science, such as shredded documents reconstruction;
- genome biology, such as assembly of DNA or RNA [HF18];
- medicine, such as reassembly of fractured bones for surgery.

Figure 2.6: Example of skull reassembly. © K. Zhang et al. [ZYM+15].

2.4.1 *The case of archaeology*

The case of automatic reassembly is extensively studied to restore cultural sites and objects, as Rasheed and Nordin highlight in their surveys [RN15a, RN15b]. It is indeed a crucial task for heritage sciences as it improves the understanding and conservation of these sites. For example, it enables to counter the effect of erosion or prevent material damages. Because of their size, some archaeological ensembles require automatic assembly.

The first pitfall encountered by conservators is the digitization of fragments. In some cases, the fragments are well-labeled in museum collections and light enough to be handled. In other cases, they remained on the archaeological site or are too big to be easily digitized. It is the case of Roc-aux-Sorciers ceiling blocks that are mostly still on site. Regarding the Vaux-de-la-Celle temple, some blocks are located in a museum and have been digitized, as blocks of Figure 2.7. In any case, scanning and cleaning the dataset takes a long time and should be cautiously planned.

Restoring archaeological vestiges requires to address a variety of tricky issues. Examples are the fragments' non-square shape, the very different sizes of the fragments, the fading of the fragments contours and colors, the missing fragments, the mixture of fragments from different objects, and the continuity of the space of the relative transformations between a couple of fragment.

Figure 2.7: A digitized pair of matching blocks from a statuary group preserved in the *Musée archéologique départemental du Val-d'Oise* (Guiry-en-Vexin), from the Vaux-de-la-Celle temple.

3

When puzzles meet computer vision

◂ Chapter 2 Chapter 4 ▸

Synopsis This chapter walks through state-of-the-art in automatic reassembly when no artificial intelligence is involved.

3.1 Introduction

The first jigsaw puzzles are credited to John Spilsbury, who invented them in 1767. They were used as an educative tool for children and only emerged for adults around 1900 (source).

In this chapter, we present many puzzle-solving methods that do not rely on artificial intelligence. They operate on either 3D blocks or 2D patches. Among puzzle-solvers, most state-of-the-art methods fall into one of these two categories: some exploit the content of each patch (such as colors or patterns) in §3.2, while other focus on the contours of the patches in §3.3. As one might guess, a few methods combine the two ways of extracting pertinent features in §3.4. Based on the features, the reassembly algorithm produces a coarse or precise reassembly. The algorithms include greedy algorithms, graph models, combinatorial solvers, iterative reassembly, and human expert advice. Appendix B provides an introduction to such algorithms.

We conclude this chapter with a short discussion on the reassembly methods presented above in §3.5, planting the seeds of the following chapters.

3.2 Solving from the content

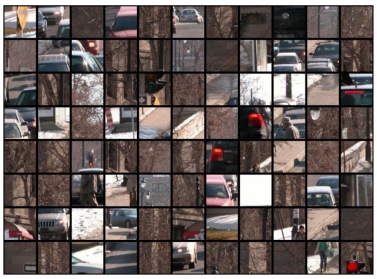


Figure 3.1: Example of input for a content-based solver. © D. Bridger et al. [BDT20].

The colors and patterns constitute the content of the 2D patches, plus depth in the case of carved 3D blocks, from which one can extract the salient curves. These characteristics make it possible to solve a puzzle thanks to the identification of visual continuities. Most content-based approaches use 2D square patches as input and colors as features, which is the case of the research presented below:

Son et al.[1] compute all the merged pair of patches' dissimilarity ratio for every possible configuration. They extended their work in [SHC+16], where they reduce the dependency on dissimilarity and instead exploit the consensus, as in Random Sample Consensus (RANSAC)[2] [FB81]. For the reassembly step, they present an algorithm that solves puzzles in a bottom-up fashion: from the initial pair of patches, it iteratively assembles them into pairs of pairs, and so forth until no pair can be merged. Then, the algorithm proceeds top-down and merge all the structures.

Similarly, Paikin and Tal[3] propose a compatibility metric based on

[1] [SHC14] K. Son, J. Hays, and D.B. Cooper, Solving Square Jigsaw Puzzles with Loop Constraints.

[2] RANSAC on Wikipedia.

[3] [PT15] G. Paikin and A. Tal, Solving Multiple Square Jigsaw Puzzles with Missing Pieces.

both the dissimilarity between the patches and the compatibility inspired by the "best buddies" metric, introduced in [PSBS11]. Paikin and Tal propose a greedy placement algorithm that iteratively selects and places the best candidate. They take special care in selecting the first piece, as it impacts all the following stages. Note that there is no constraint on the shape of the puzzle, so that the first piece is always well-placed.[4]

Gallagher[5] builds his solution on the Mahalanobis distance[6], which gives importance to the local gradients near a patch's borders. He casts his puzzle problem into a tree and uses Kruskal's algorithm[7] variant to find the minimal spanning tree.

Sholomon et al.[8]'s work is inspired by [TFSM02]. They both propose a genetic algorithm[9] that merges two wrongly-solved parents into a child while trying to minimize dissimilarity. Sholomon et al.'s fitness function is computed from the pairwise compatibility, i.e., two pieces' likelihood to be adjacent.

[4] More detail on this idea in Chapter 9.

[5] [Gal12] A.C. Gallagher, JigsawPuzzles with Pieces of Unknown Orientation.

[6] Mahalanobis distance on Wikipedia.

[7] Kruskal's algorithm on Wikipedia.

[8] [SDN13] D. Sholomon, O. David, and N.S. Netanyahu, A Genetic Algorithm-Based Solver for Very Large Jigsaw Puzzles.

[9] Genetic algorithms on Wikipedia.

3.3 SOLVING FROM THE CONTOUR

The contour refers to fractured surfaces: 3D broken objects exhibit fractured and intact surfaces (Figure 3.2), while 2D objects are supposed to have only fractures (Figure 3.3). When an object is fully reassembled, no fracture should remain, and only its original contours can be observed. The literature on contour-based solving appears to be much more abundant than on content-based solving. We carefully select what seems to be the most important papers and present them below:

Figure 3.2: An artifact reassembly. © Q.-X. Huang et al. [HFG+06].

Huang et al.[10] address 3D broken artifacts reassembly by segmenting the surfaces into a set of faces. They then extract their features and proceed to pairwise matching: they analyze all the potential correspondences to select the valid ones and represent the merging with sub-graphs. Finally, they merge the sub-graphs and obtain a reassembly.

[10] [HFG+06] Q.-X. Huang et al., Reassembling Fractured Objects by Geometric Matching.

Zhang et al.[11] present two approaches, which can be used together or separately depending on the context. In the first approach, a general template of the artifact to rebuild is provided, and their algorithm matches the features. The second approach identifies the fractured regions and extracts boundary curves and feature regions, rather than feature points. They use a graph model to find the best reassemblies. Note that other work features boundary curves matching for 2D puzzles, such as [ZZZH06].

[11] [ZYM+15] A. Zhang et al., 3D Fragment Reassembly using Integrated Template Guidance and Fracture-Region Matching.

Papaioannou et al.[12]'s pipeline starts by preprocessing all the 3D meshes to discriminate potentially fractured regions from intact surfaces. When some fragments have no or very small usable contact surface, they are submitted to users that find and extract features curves. They are used to find continuity, making this

[12] [PSA+17] G. Papaioannou et al., From Reassembly to Object Completion - A Complete Systems Pipeline.

combined approach a mixed-method (see §3.4). The next step is the computation of the pairwise scores of fractured regions. Last, a combinatorial solver selects the best matches and build the reassembly. Papaioannou et al.'s pipeline encompasses other steps, such as multi-part refinement and completion.

Sizikova and Funkhouser[13] propose a genetic algorithm. They provide pairwise matches and single fragments as input, which are called clusters. Their algorithm alternates between selection and recombination phases until convergence. The fitness function used for selection ranks the current clusters by size, diversity, and estimated correctness. The recombination creates new clusters from two-parent clusters. They either share a common fragment a set of spanning matches.

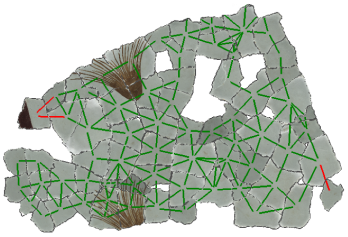


Figure 3.3: A contour-based puzzle. © E. Sizikova and T. Funkhouser [SF17].

[13] [SF17] E. Sizikova and T. Funkhouser, Wall Painting Reconstruction Using a Genetic Algorithm.

3.4 MIXED METHODS

Finally, some work exploits both the contour and the content.

Tsamoura and Pitas[14] identify the similarities between fragments colors with a content-based image retrieval system. It allows them to reduce the computational burden of the second step, which is the discovery of matching couples contour segments of adjacent image fragments based on the Smith-Waterman algorithm[15]. Then, they use the iterative closest point algorithm to align the image fragments[16]. Last, they reassemble all fragments, using an alignment angle to solve conflicts.

[14] [TP09] E. Tsamoura and I. Pitas, Automatic color based reassembly of fragmented images and paintings.

[15] Smith-Waterman algorithm on Wikipedia.

[16] Iterative closest point on Wikipedia.

Zhang and Li[17] introduce a method based on both fragment shapes and patterns. They approximate polygons from the fragments contour and match them in pairs. Each matching is associated with a score based on the contour and the colors of the patches. Last, they apply a graph optimization algorithm that selects the edges that maximize compatibility rather than the highest scores to reassemble the whole image.

[17] [ZL14] K. Zhang and X. Li, A graph-based optimization algorithm for fragmented image reassembly.

Derech et al.[18] propose to match overlapping fragments rather than searching valid continuities. To do so, they extrapolate the fragments and superpose the extrapolation, looking for a match. Then, they solve the puzzle one piece after another: they use the current reassembly to place the next fragment. They also consider a slight erosion of the fragments borders and tackle it by using inpainting techniques.

[18] [DTS18] N. Derech, A. Tal, and I. Shimshoni, Solving Archaeological Puzzles.

Savelonas et al.[19] propose a setup that separates fractures from intact surfaces. Then, salient curves are extracted from the intact facets. A heuristic is then used to compare each pair of fragments, looking for the number of features and the surface's geometric texture. After this initial test, they look for matches of the plausible pairs, using either fractures or curves. They construct a graph and use Kruskal's algorithm to obtain complete reassembly.

[19] [SAPM17] M.A. Savelonas et al., Exploiting Unbroken Surface Congruity for the Acceleration of Fragment Reassembly.

3.5 CONCLUSION

This chapter presented the most popular methods to solve puzzles with computer vision without deep learning. The main criticism of such approaches is that they rely solely on local information. As such, they do not achieve a global understanding of the images' content, which can lead to incorrect reassemblies.

However, these algorithms provide both goals and benchmarks to help us design deep learning-driven approaches.

Artwork 3: *Two Men Contemplating the Moon*, Caspar David Friedrich, ca. 1825-30, from the MET Open Collections.

4

On the datasets

◂ Chapter 3 Chapter 5 ▸

Synopsis This chapter lists the input data requirements in §4.1 and introduces the datasets we use to train (§4.2.1) and test (§4.2.2) our neural networks.

4.1 Requirements for the dataset

To reassemble an object using only the semantics, we need to prevent our neural networks from using visuals cues such as fractures regions and color continuities. Hence, all the fragments should be similar in shape and size and cropped enough to break continuities. To a certain extent, the erosion can conceal the original fragment shape, and so this square-patch hypothesis is consistent with the archaeological setting in which we operate.

Although we ought to reconstruct monuments from blocks, we do not need to feed our neural networks with 3D images. Indeed, the semantic information is often only present on one side of the fragment, so using 3D blocks turns out to be inefficient because the deep learning algorithm would need to extract the pertinent face's data. If we want to keep the depth information, it is better to project the block onto a 2.5D[1] picture. For example, Roc-au-Sorcier data benefits from 2.5D because the color contrast of the blocks is very low. Last, when significant information is sculpted on two faces, i.e., for an outside corner block of the Vaux-de-la-Celle's temple, placing the faces side by side (i.e., flattening) preserve the geometry of the sculpted representations.

[1] We use the adjective 2.5D to describe any 2D image that uses grey-level to depict the depth.

Inspired by previous work such as Doersch et al.[2], we opt for 3×3 jigsaw puzzles with square 2D fragments spaced by an important margin representing the erosion.

[2] [DGE15] C. Doersch, A. Gupta, and A.A. Efros, Unsupervised visual representation learning by context prediction.

However, we do not train our neural networks on the 2D images from Roc-aux-Sorciers and Vaux-de-la-Celle, for three reasons. First, Doersch et al. also highlight in [DGE15] an important issue shared by most standard datasets: the color deformations induced by the camera lens. They find out that the green channel is shrunk towards the center of the pictures, and their first neural network relied on that information to correctly reassemble the images. They recommend using pictures taken for a high-quality camera with no lens aberration, at least for the training set, and we do not have such a camera at our disposal.

Second, training a neural network requires a large amount of data, and the hundred of fragments from each site is not nearly enough to obtain accurate predictions from the network.

Third, we do not have enough examples of correct reassemblies to do supervised learning: among the blocks, only a few combinations have been identified by archaeologists and conservators. We would need thousands of validated combinations for the training and validation sets. As we cannot break artifacts to obtain more pairs of known reassemblies[3], using the Roc-aux-Sorciers and Vaux-de-la-Celle blocks' images for training is incompatible with the supervised learning we planned.

For these three reasons, we build our main dataset from pictures without lens-induced flaws. We find out that most of the museums' open-access photographs datasets are made through a high-quality scanning procedure and meet our quality and quantity criteria. Last, we can divide the images into parts virtually, which give us the labels.

[3] An alternative is to virtually break 3D-scans into pieces which allows augmenting our dataset, provided that we prefer to work on scanned data rather than on photographs.

Puzzle-solving can be seen as a self-supervised task: labels (i.e., the patches positions) are given to the neural network, but they are generated along with the patches.

4.2 THE DATASETS

4.2.1 *The MET dataset*

We introduce the met dataset in our first article[4]. This dataset is made of open-access photographs from the Metropolitan Museum of Art (MET). It provides images taken with ultra-high-resolution cameras[5] that avoid the lens bias mentioned above (§4.1) that comes with the popular computer vision datasets.

[4] [PPT18b] M.-M. Paumard, D. Picard, and H. Tabia, Jigsaw puzzle solving using local feature co-occurrences in deep neural networks.

[5] A museum without walls: How the Met is bringing its ancient collection online, Mashable.com, © R. Kraus, 2018.

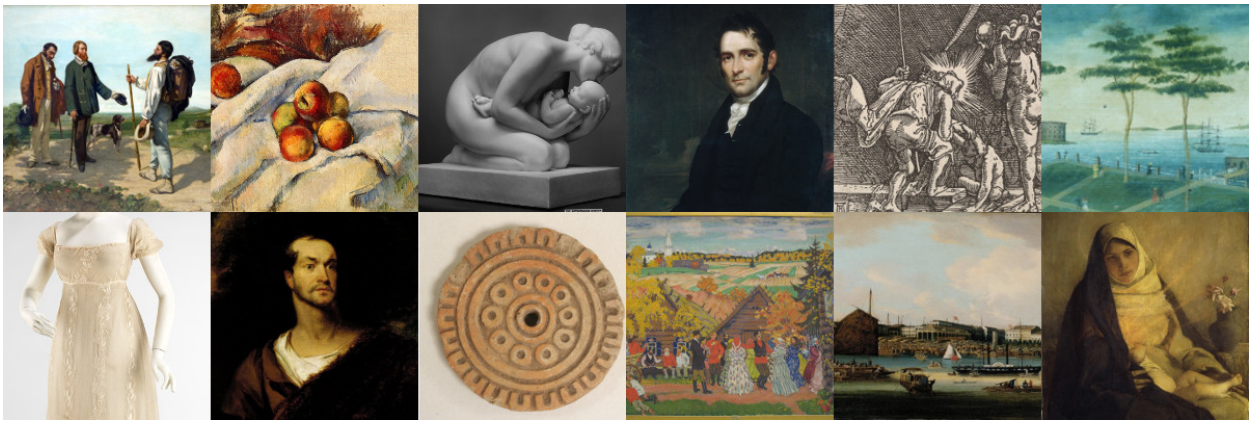


Figure 4.1: Images from the MET dataset.

The dataset pictures (Figure 4.1) fall into three main categories, similar in size: artifacts, engravings and texts, and paintings. An artifact may be a piece of clothing, a piece of tableware, a pottery plate, a carved flint, or a sculpture. As the artifact pictures display a uniform background, the background fragments are expected to be misplaced. The paintings are mostly portraits and landscapes. The engravings include of geometric engravings (around 20% of the dataset), illustrated engravings (13% of the dataset) and printed texts

(less than 1% of the dataset). A fifth of the dataset is composed of black and white images.

The dataset is made of 10000 training images and 2000 validation images. Patches coordinates are provided as well for replication purposes, but they can be randomly extracted from images at each step of the learning process if the user prefers. We prepare the patches following the procedure exposed by Doersch et al. [DGE15]. Each image from the training set is resized and square-cropped so that its size is 398×398 pixels. We divide it into 9 parts separated by a 48-pixels gap, mimicking the fragments' erosion. Each fragment is of size 96×96 pixels, and we randomly jitter its location by ±7pixels in each direction, as illustrated in Figure 4.2.

Figure 4.2: Example of puzzle preparation from an image of the MET dataset.

4.2.2 *Other datasets*

We apply our methods to four other datasets: the well-known ImageNet, a self-made bas-reliefs dataset, and two small validation set of pictures from our archaeological sites.

ImageNet ImageNet [DDS+09] is a large visual dataset used in visual object recognition. It has continued to evolve since 2009 and now contains millions of hand-annotated images [RDS+15]. The images have been collected through the web, and they depict real-life situations and objects in their environment (Figure 4.3).

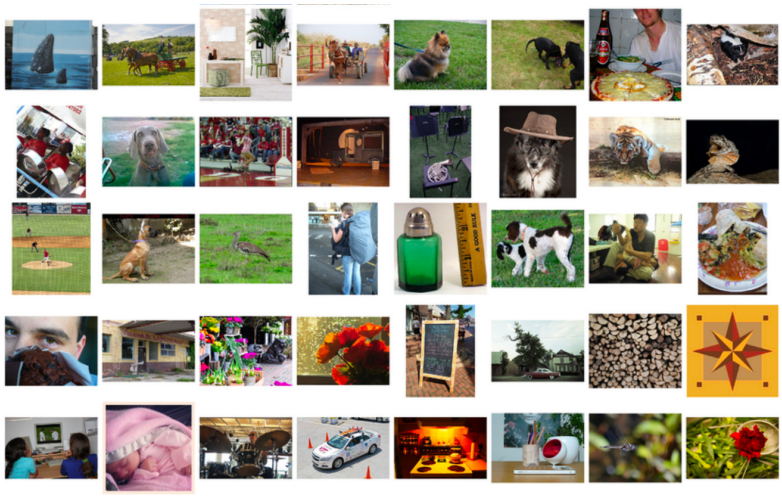


Figure 4.3: Images from ImageNet.

We use the data used for Large Scale Visual Recognition Challenge 2012 (ILSVRC2012), which contains 1,200,000 training images and 150,000 validation images.

Bas-relief We propose a dataset made of bas-reliefs found on the internet. It is made of 100 fine-tuning images and 87 validation images. Figure 4.4 shows some pictures from this dataset. Most of the images display low contrast and similar hue.

Figure 4.4: Images from the bas-reliefs dataset.

Roc-aux-Sorciers We crop 20 square puzzles from the Bourdois shelter's frieze of Roc-aux-Sorciers (Figure 4.5).

Figure 4.5: Images from Roc-aux-Sorciers.

Vaux-de-la-Celle From the 3D-scans of 60 Vaux-de-la-Celle's blocks, we extract the most important face and make 2D renderings under 14 different lighting. Figure 4.6 illustrates the light variation for three 2D-renders.

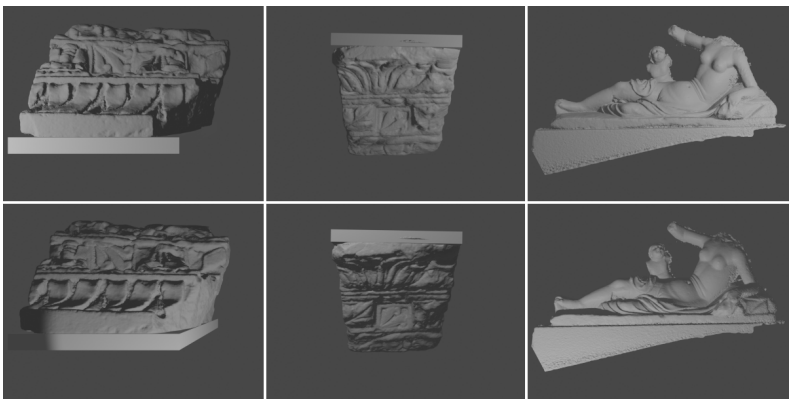


Figure 4.6: Images from Vaux-de-la-Celle.

Part II

PAIRWISE COMPARISON WITH DEEP LEARNING

5

Puzzle-solving with deep learning

◂ Chapter 4 Chapter 6 ▸

Synopsis This chapter introduces the methods that rely on deep learning to solve puzzles. After introducing the three types of methods in §5.1, we detail each of them in §5.2-5.4.

5.1 Introduction

In Chapter 3, we presented state-of-the-art methods to solve puzzles without relying on deep learning. We now present those that do. If the reader is unfamiliar with deep learning research, we recommend reading Appendix A first.

In 2015, Doersch et al.[1] introduced a pretext task[2] for classification and detection. They endow a neural network with a sense of spatial semantics by training it to solve 3×3 puzzles, i.e., square puzzles with one central fragment and eight lateral fragments. Their neural network is trained to predict the relative position of a lateral fragment compared to the central fragment (Figure 5.1). Because Doersch et al. only care about the tasks that follow the pretext task, they settle for classifying the positions rather than proceeding to complete reassembly.

Doersch et al. brought the puzzle-solving task to the deep learning community and inspired many works. Some also regard it as a pretext task, while others tackle the reassembly problem as a whole. Another line of comparison emerged through state of the art, concerning the objective of the neural networks, which can either be:

Pairwise comparison: The neural network predicts the position of a fragment in relation to another fragment (§5.2);

Global comparison: The neural network predicts the position of a fragment compared to the placed fragments (Chapter 9);

Permutation: Given all the fragments, the neural network predicts a permutation that gives a correct reassembly (§5.4).

[1] [DGE15] C. Doersch, A. Gupta, and A.A. Efros, Unsupervised visual representation learning by context prediction.

[2] A pretext task is what we use to pretrain a neural network. It allows it to discover visual features in a self-supervised learning setup, which helps it to perform better at another task. We detailed why puzzle-solving is self-supervised in §4.1.

Figure 5.1: Example of pairwise comparison task. © Doersch et al. [DGE15].

5.2 Pairwise comparison

The pairwise comparison[3] has been introduced by Doersch et al. in [DGE15] and strongly inspired our work [PPT18a, PPT18b, PPT20] as well as Ostertag and Beurton-Aimar's [OBA20] and Bridger et al.'s [BDT20]. In both cases, the reassembly is obtained from a

[3] See §5.1 for a brief explanation of Doersch et al.'s method.

decision algorithm[4]. The reassembly algorithm makes its choice based on each pair of fragments' relative position, which has been predicted by the neural network.

[4] Appendix B is an introduction to decision science.

Ostertag and Beurton-Aimar[5] study reassembly for ostraca[6]. Their dataset is composed of square ostraca images, which they cut into 9 same-sized fragments. As a continuation of their work in [PAJ19], they use a 2D Siamese neural network to evaluate the matching possibilities of each couple of fragments. They predict whether the second fragment goes up, down, left, right, or is not adjacent to the first one. The reassembly step constructs a graph through iterative addition of small fragments, which is very similar to our greedy algorithm (§6.5). They do not use erosion, but they create oblique fractures between the pair of fragments. In detail, they concatenate two adjacent fragments, draw a random oblique line between the fragments, and cut the fragments in such a way that each contain the pixels of "its side" of the oblique line (Figure 5.2). The pixels of the "other side" are replaced with fully transparent black pixels. They reach 96% on the pairing task; we suspect their neural network to look for matching based on the oblique rather than on the content or the semantics. They do not give any score on reassembly, describing their results as "poor."

[5] [OBA20] C. Ostertag and M. Beurton-Amar, Matching ostraca fragments using a siamese neural network.

[6] An *ostracon* is a piece of pottery.

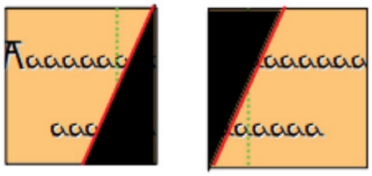


Figure 5.2: The oblique fracture mentioned in [OBA20]. © Ostertag and Beurton-Aimard [OBA20].

Bridger et al.[7] use a small erosion that allows them to inpaint the erosion area of each couple of pieces in order to classify their relation (up, down, right, left, not adjacent). The fragments are not compared to a central fragment. Then, they apply the greedy placement algorithm presented in [PT15][8].

[7] [BDT20] D. Bridger, D. Danon, and A. Tal, Solving jigsaw puzzles with eroded boundaries.

[8] Read about [PT15] in §3.2.

5.3 GLOBAL COMPARISON

We introduced iterative solving with deep learning in our article [PPT21]. Chapters 9 and 10 describe it in detail.

5.4 PERMUTATIONS

The permutation-based methods do not compare the lateral fragments to a central one. They solve puzzles by finding the correct permutation to perform on the fragments. Such methods are end-to-end: no reassembly algorithm is required to solve the puzzle. The standard setup introduced in [NF16, NVFP18] is a classification problem, were each permutation (usually 9!) is a class. The high number of classes induces a tremendous computation time. The authors avoid this issue by restricting the number of possible reassemblies, which causes most of them to be unattainable.

Noroozi and Favaro[9] solve 3×3 jigsaw puzzles and use the puzzle-solving as a pretext task. The neural network receives the 9 pieces as an input and predicts the correct fragments permutation among 10 to 1000 combinations. While preserving their architecture,

[9] [NF16] M. Noroozi and P. Favaro, Unsupervised learning of visual representations by solving jigsaw puzzles.

they complicate the resolution task in their extension article[10] by replacing one or two fragments of the puzzle by fragments extracted from a random image. This setup is not equivalent to solving puzzles with two missing and two outsider fragment, as the two outsider fragments cannot be labeled as outsiders.

[10] [NVFP18] M. Noroozi et al., Boosting Self-Supervised Learning via Knowledge Transfer.

Santa Cruz et al.[11] propose an architecture that order an image sequence, which serves as a pretext task. For instance, they order a set of faces by their expressions and solve 3×3 jigsaw puzzles. They introduce a Sinkhorn layer that transforms the predictions into a permutation matrix. Their architecture achieves better results than [DGE15, NF16] on the usual classification tasks. However, they do not evaluate their architecture on the puzzle-solving task.

[11] [SCFCG17] R. Santa Cruz et al., DeepPermNet: Visual Permutation Learning.

Kim et al.[12] tackle the case of discoloration with one missing fragment. Based on inpainting and colorization techniques, they restore the images (Figure 5.3). For the reassembly, they use a network similar to Noroozi et al.'s [NF16] in which they input a white fragment, representing the missing tile. They provide no insight into the effectiveness of their architecture for the reassembly.

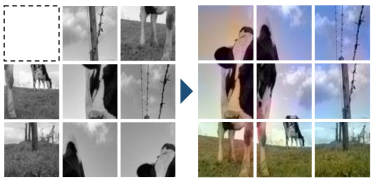


Figure 5.3: Example of permutation task with inpainting and colorization. © Kim et al. [KCYK18].

[12] [KCYK18] D. Kim et al., Learning Image Representations by Completing Damaged Jigsaw Puzzles.

Wei et al.[13] propose a permutation method that partially relies on pairwise comparisons. Their neural network outputs a cost made of two terms: one that gives the relative position (among 9) of every couple of fragment, and one that measures how likely a fragment is located in each position. This last term is obtained from the concatenation of all the features that feed a fully-connected layer that outputs the matrix of fragment-position scores. Then, a permutation is obtained from the cost function. It is applied, and the new reassembly is re-evaluated, iteratively until a stop criterion is reached. This method is still limited in terms of puzzle size, and solving 4×4 is too resource-costly. Nonetheless, Wei et al. have been able to solve 3×3×3 puzzles.

[13] [WXR+19] C. Wei et al., Iterative Reorganization with Weak Spatial Constraints: Solving Arbitrary Jigsaw Puzzles for Unsupervised Representation Learning.

5.5 COMPARISON

Table 5.1 shows the reassembly tasks[14] addressed by each mentioned article. We detail our work from [PPT18a, PPT18b, PPT20, PPT21] in the next chapters and only present other author's work. The Table details the maximum quantity of fragments used as input, the puzzle's binding size (if any), the available permutations[15], the percentage of erosion, and the maximum number of missing and extra fragments. Note that all methods except Doersch et al.'s compute a reassembly; some aim to improve the reassembly quality while others only focus on the tasks for which the puzzle-solving is a pretext task.

The pairwise comparison-based methods exhibit strengths and weaknesses as opposed to permutation-based methods. In the first case, it is possible to work with missing or extra fragments. In the case of permutations, the neural network makes its predictions based on all the fragments, rather than a couple of fragments.

[14] See §2.2 to review the reassembly tasks.

[15] The available permutations column indicates the number of puzzles that can be solved and is explained in §5.4.

Article	Type	Quantity of fragments	Binding puzzle size	Available permutations	Erosion	Missing fragments	Extra fragments
[DGE15]	Pairwise	9	3×3	All (9!)	50%	-	-
[PPT18b]	Pairwise	9	3×3	All (9!)	50%	No	No
[PPT18a]	Pairwise	9	3×3	All (9!)	50%	7	No
[PPT20]	Pairwise	9 (12 seen)	3×3	All (9!)	50%	7	3
[OBA20]	Pairwise	9	3×3	All (9!)	No*	No	No
[BDT20]	Pairwise	150	No	All	7-14%	No	No
[PPT21]	Global	9	5×5	All (25!)	10-30%	No	No
[NF16]	Permutation	9	3×3	1000	17%	No	No
[NVFP18]	Permutation	9	3×3	1000	17%	2	2
[SCFCG17]	Permutation	4	2×2	All (4!)	No	No	No
[KCYK18]	Permutation	8+1	3×3	All (9!)	No	1	No
[WXR+19]	Permutation	9	3×3(×3)	All	30%	No	No

Table 5.1: The reassembly task addressed by each presented article. (*) indicates the oblique fracture with no erosion; see the in-text explanation.

6

Deepzzle

◂ Chapter 5 Chapter 7 ▸

Synopsis This chapter explains Deepzzle, our jigsaw puzzles solver relying on the pairwise comparison. It starts with an overview §6.2 and a mathematical formulation §6.3 of the puzzle-solving task, and then it details the pairwise comparison §6.4 and the reassembly §6.5 steps. The last section gives a description of our experiments and metrics §9.5.

6.1 Introduction

In this chapter, we present Deepzzle, a method to solve 3×3 jigsaw puzzles (Figure 6.1) with the pairwise comparison. We place our work in an archaeological context, with heavily eroded fragments.

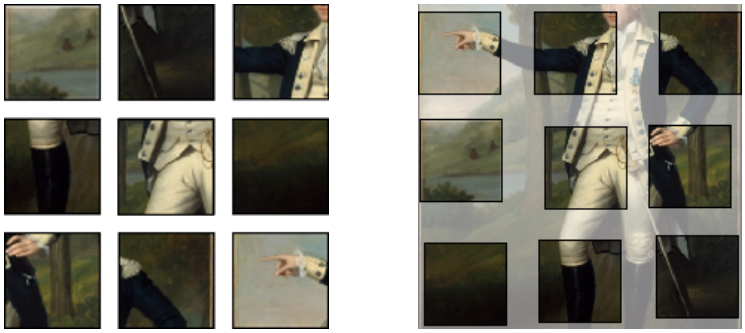


Figure 6.1: A task submitted to Deepzzle.

Among our contributions, we propose two extensions of the standard 3×3 problem introduced by Doersch et al. [DGE15]. First, we deal with missing fragments and outsider fragments, which are frequent in archaeology. In this case, we allow fragments to be unused and positions to be unfilled. Second, we consider the case where the central fragment is unknown. In this case, we compute the relative positions supposing that each fragment is the central one.

Sources We walk through the following articles, presenting most of our contributions[1]:

- *Jigsaw Puzzle Solving Using Local Feature Co-Occurrences in Deep Neural Networks*;
- *Image Reassembly Combining Deep Learning and Shortest Path Problem*;
- *Deepzzle: Solving Visual Jigsaw Puzzles with Deep Learning*.

[1] The results are discussed in Chapter 7.

6.2 METHOD OVERVIEW

Figure 6.2 illustrates our puzzle-solving process, inspired by Doersch et al. [DGE15]. A neural network predicts the relative position of each couple of central-lateral fragments. The probabilities are used to build a graph that serves to identify the best reassembly.

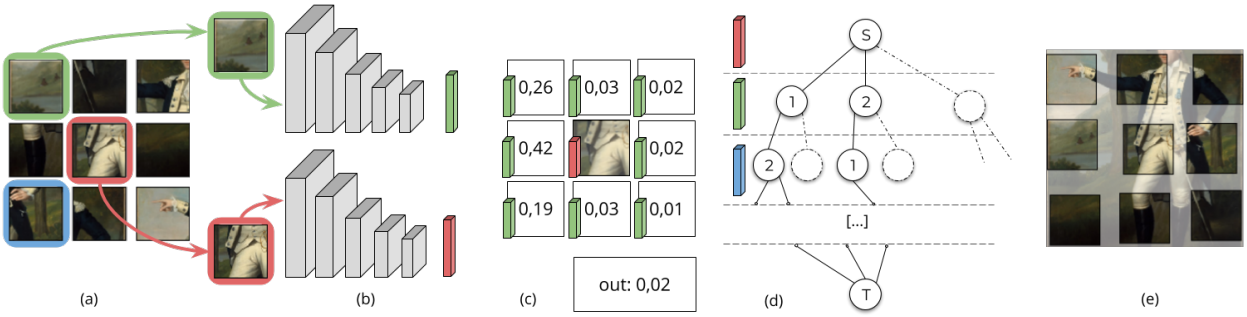


Figure 6.2: Outline of Deepzzle. From a set of pieces (a) made of a central fragment (in red) and lateral fragments, we pick a lateral fragment (in green). We extract its features (b) and predict its place among the eight lateral positions and the outsider class (c). Then, we build the graph of the prediction (d) in which each line matches a fragment. The reassembly (e) is computed from the shortest path in the graph.

Our method addresses the following settings:

- The puzzle size is 3×3;
- The algorithm is explicitly given a central fragment;
- The algorithm solves complete puzzles (9 fragments) and puzzles with missing fragments;
- The algorithm solves puzzle with no or several added fragments. In this last case, the available relative positions (i.e. classes) represent the 4 cardinal position, the 4 intercardinal[2] position, and the non-adjacency of the fragments.

We also solve puzzles with unknown central fragment: we predict all the combinations for each fragment made central, and we select the best reassembly among them.

[2] The intercardinal positions are: left-top, right-top, left-bottom, and right-bottom.

6.3 PROBLEM FORMULATION

Puzzle-solving identifies the most probable reassembly, i.e., the reassembly that satisfies as many relative position predictions as possible. In this section, we present the optimization problem we use to solve 3×3 jigsaw puzzles. We also justify we can use the pairwise comparison to the central fragment to solve a puzzle.

NOTATIONS We introduce P_r a probability and $x_{i,j} \in [0,1]$ the affectation of the fragment $i \in [0\,..\,f]$ at the position $j \in [0\,..\,p]$, where f is the number of lateral fragments and p the number of available position, i.e., classes. Therefore, $p \in [8,9]$, where 9 applies to the case when added fragments are permitted.

We define position 0 as the central position and fragment 0 as the central fragment. We then introduce $x_c = x_{0,0}$, the placement of the central fragment at the central position.

We want to find the maximum joint probability of placing all fragments:

$$\max P_r(x_c, x_{1,1}, x_{1,2}, \dots, x_{2,j_1}, \dots, x_{f,p}).$$

Because each fragment can occupy only one position, we simplify the latter equations and introduce $x_i \in [1\,..\,p]$ the chosen affectation of the fragment i:

$$\max P_r(x_c, x_1, x_2, \dots, x_f). \tag{6.1}$$

The constraint on the single fragment per position is then:

$$\forall i, j \in [1\,..\,f]^2 \text{s.t.} x_i \neq 9, x_i \neq x_j.$$

As the predictions of the positions of the lateral fragments depend on the central fragment, we extract the central fragment x_c from P_r. We use Bayes rule:

$$P_r(x_c, x_1, \dots, x_f) = P_r(x_1 \dots x_f | x_c) \times P_r(x_c).$$

We assume $P_r(x_c) = 1$. To ease the notation, we drop the term $|x_c$ in the further equations while keeping in mind that x_c conditions all probabilities.

We now restate the previous equation with Bayes rule, to expose that assembling the puzzle is an iterative process where fragments are selected and placed sequentially. As such, the probability of a reassembly depends on the probabilities of placing the last fragment, knowing that all previous fragments are placed:

$$P_r(x_f \dots x_1) = P_r(x_f | x_{f-1} \dots x_1) \times P_r(x_{f-1} \dots x_1). \tag{6.2}$$

To obtain a tractable approximation, we suppose that x_i follows the Markov Chain:

$$P_r(x_f | x_{f-1} \dots x_1) = P_r(x_f | x_{f-1}). \tag{6.3}$$

Unrolling the recursion of Equation 6.2 leads to:

$$P_r(x_1 \dots x_f) = \prod_{i \in [2\,..\,f]} P_r(x_i | x_{i-1}) \times P_r(x_1).$$

To further simplify the problem, we make the approximation that x_i and x_{i-1} are independent:

$$P_r(x_i | x_{i-1}) = P_r(x_i), \tag{6.4}$$

which leads to:

$$P_r(x_1 \dots x_f) = \prod_{i \in [1\,..\,f]} (P_r(x_i)).$$

This approximation allows using pairwise relationships to solve a puzzle. Without this approximation, the neural network architecture would be significantly more complex as it would require to compare all the fragments. Such architecture would be less adaptable to missing and outsider fragments.

In turns, it means we want to solve the following optimization problem:

$$\max P_r(x_1, \dots x_f) = \max \prod_i P_r(x_i), \tag{6.5}$$

which is equivalent to:

$$\max \log P_r(x_1, \dots x_f) = \max \sum_i \log P_r(x_i). \tag{6.6}$$

6.4 PAIRWISE COMPARISON STEP

In order to solve the optimization problem of Equation 6.6, we need an estimator of $P_r(x_i|x_c)$. We cast the problem of estimating $P_r(x_i|x_c)$ as a classification problem that can easily be solved by a deep convolutional neural network. The neural network has two inputs, corresponding to the central fragment and the lateral fragment, and its outputs correspond to the possible positions of the fragment i. To optimize this network, we use a categorical cross-entropy. The architecture we propose is directly derived from the independence approximation made in Equation 6.4.

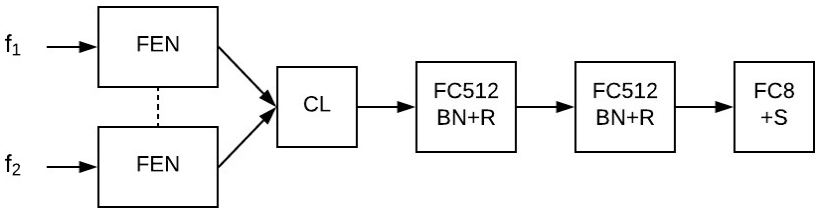


Figure 6.3: Full network architecture for 8 classes. FEN: Feature Extractor Network. CL Combination Layer. FC: Fully Connected. BN: Batch-Normalization. R: ReLU activation. S: Softmax activation.

More specifically, each fragment goes through a Siamese network (Figure 6.3) called the Feature Extraction Networks (FEN). It performs the same features extraction, thanks to shared weights. Then, the features of the fragments are merged in the Combination Layer (CL). Finally, three fully-connected (FC) layers followed by a batch-normalization and an activation (ReLU for the first two and softmax to ensure probabilities for the last layer) predicts the relative position.

The neural network is trained at once using stochastic gradient descent on batches of fragments pairs. Its output consists of a fully connected layer with p neurons followed by a softmax activation, corresponding to the p possible relative locations' probabilities. We propose three output sizes:

- $p = 8$, the number of lateral positions;
- $p = 9$, the number of lateral positions plus the outsider class;
- $p = 1$, a Boolean that is true when the fragments are adjacent.

6.4.1 *Feature extractor*

We use a convolutional neural network to compute the features associated with the fragments. Our architecture takes inspiration

Layer	Shape	# parameters
Input	$96 \times 96 \times 3$	0
Conv+BN+ReLU	$96 \times 96 \times 32$	1k
Maxpooling	$48 \times 48 \times 32$	-
Conv+BN+ReLU	$48 \times 48 \times 64$	19k
Maxpooling	$24 \times 24 \times 64$	-
Conv+BN+ReLU	$24 \times 24 \times 128$	74k
Maxpooling	$12 \times 12 \times 128$	-
Conv+BN+ReLU	$12 \times 12 \times 256$	296k
Maxpooling	$6 \times 6 \times 256$	-
Conv+BN+ReLU	$6 \times 6 \times 512$	1.2M
Maxpooling	$3 \times 3 \times 512$	-
Fully Connected+BN	512	2.4M

Table 6.1: Architecture of the Feature Extraction Network with 8 output classes. Conv: 3×3 convolution, BN: Batch-Normalization, ReLU: ReLU activation.

from VGG [SZ14] and is composed of a sequence of 3×3 convolutions followed by batch-normalizations [IS15], ReLU activations [NH10] and max-poolings. The full architecture is shown on Table 6.1.

The feature extraction network is inspired by VGG [SZ14] with a fully connected layer that allows preserving the spatial layout of the input fragment. We did not append a global pooling layer [LCY13] to the FEN, and thus spatial information is preserved, which we believe is essential for the relative position prediction.

We also tried other models based on more recent architectures such as Resnet [HZRS16], but we empirically found that they were underperforming compared to the simpler architecture. It can be explained by the fact that, contrary to full images, fragments do not contain as much semantic information and thus require less involved features.

6.4.2 *Combination layer*

These features obtained from the FEN are combined through a combination layer. In [DGE15], the authors propose to concatenate both features in the combination layer. The output of the concatenation layer is, therefore:

$$y_{CL} = x_1 + x_2,$$

where y_{CL} is the output of the concatenation layer and x_1 and x_2 the outputs of the FEN.

In such a formulation, the cross-covariance between the features of both fragments is neglected. Indeed, the output of the convolutional neural network can be viewed as localized pattern activations. The relative position's prediction depends on the conjunction of specific patterns occurring at specific positions in the first fragment and specific patterns occurring at specific positions in the second fragment. It can be argued that a sufficiently deep multi-layer perceptron can model these cross-covariances, but it seems easier to model them directly.

In [LRM15], the authors suggest modeling these co-occurrences

of patterns using a bilinear model, which can be computed using the Kronecker product of the feature vectors. They report improved accuracy on fine-grained classification. Therefore, we implemented an alternative combination layer with the Kronecker product $\otimes$:

$$y_{CL} = x_1 \otimes x_2.$$

However, using the Kronecker product leads to high dimensional features that are intractable in practice. To overcome this burden, the authors of [GBZD16] propose to use random projections combined with the Hadamard product[3] to approximate the bilinear model. This strategy is further extended in [KOL+16], where the projections are trained in true deep learning fashion. We implemented another possible combination layer:

$$y_{CL} = x_1 \circ x_2,$$

where $\circ$ is the Hadamard product. As we said, it is an approximation of the Kronecker product; therefore, we expect to get lower scores but better computing time.

Alternatively, a factorization based on the Tucker decomposition is also proposed in [BYCCT17], which allows controlling the rank of the considered co-occurrences.

[3] The Hadamard product is also known as "element-wise product." Hadamard product on Wikipedia.

6.5 REASSEMBLY STEP

We present three reassembly methods to solve Equation 6.6 with the probabilities predicted by the deep neural network: a greedy solver, an exact solver relying on Dijkstra's algorithm, and a heuristic.

We consider the case where the central fragment is known, with no missing or added fragment. We feed the reassembly algorithm with Y, the 8×8 predictions matrix obtained through the last fully-connected layer of the neural network. Each cell contains $x_{i,j}$, i.e., the odds that the corresponding lateral fragment (the row of the matrix) has a specific relation (the columns) with the central fragment. Henceforth, the sum of each row values is 1.

Appendix B gives an introduction to the assignment problem, standard algorithms, performance, and complexity.

6.5.1 *Greedy solver*

The puzzle-solving problem is an assignment problem where we have to pick 8 values from the matrix (only one per row/column) such that their sum is maximized. We iteratively pick the maximum value and remove its corresponding row and column until all the positions are filled. In the rare case that two cells contain the exact same maximum float, we select the one from the first row. Therefore, the order of the fragments' impact on the solution we obtain can be approximated as null. We prove later that it is not the case for the heuristic.

Algorithm 1 presents the outline of the greedy solver:

Algorithm 1: Greedy algorithm outline.

```
1: procedure GREEDY(Y)
2:     reassembly ← [0] × 8
3:     while 0 ∈ reassembly or Y ≠ ∅ do
4:         max_frag, max_pos ← argmax(Y)
5:         reassembly[max_pos] ← max_frag
6:         Y.pop_row(max_frag)
7:         Y.pop_column(max_pos)
8:     end while
9:     return reassembly
10: end procedure
```

where Y is the predicted values matrix containing $x_{i,j}$. We store the solution in the size-8 array $reassembly$. Each of its cell corresponds to a position, starting from the upper-left corner. Figure 4.2 shows the correspondence between positions and $reassembly$ cells.

COMPLEXITY The complexity of the greedy solver is $\mathcal{O}(f \times p^2)$: the while loop repeats p times or less ($\mathcal{O}(p)$), the argmax operation requires to browse Y entirely (as Y is of size $f \times p$, we have $\mathcal{O}(n \leq f \times p)$ and the two pop operations are in constant time ($\mathcal{O}(1)$).

PERFORMANCE The greedy algorithm amplifies the neural network's mistakes. It offers no performance guarantee if it makes at least one mistake, i.e., if it assigns a fragment's best score to a wrong position. In the worst case, no fragment is well placed (except for the central fragment). However, if the neural network correctly predicts each fragment's class, the greedy solver always obtains the correct reassembly.

CENTRAL FRAGMENT If the central fragment is unknown, we compare all the pairs of fragments and apply 9 times the reassembly algorithm, for different central fragment each time. We score to all the reassemblies with the sum of $x_{i,j}$ for the chosen (i, j), and we keep the reassembly with the maximal score.

MISSING AND EXTRA FRAGMENTS The greedy solver can manage missing or extra fragments[4], but not both of them at the same time. Indeed, if we have 4 lateral fragments (i.e., 4 missing fragments) and 4 extra fragments, the greedy solver will place the 8 fragments and not the 4 correct ones. We add a preliminary step to the greedy algorithm to circumvent this problem: we remove all the rows where the maximal probability is in the 9th "not related" column.

[4] Reminder: with added fragments, we predict the relation among 9 classes.

6.5.2 *Graph-based reassembly*

Dijkstra's[5,6] shortest path algorithm [D+59] is an alternative to the greedy solver that comes with performance guarantee. It finds the

[5] Wikipedia article on Pr. Edsger Wybe Dijkstra.

[6] Pronounce d**ay**-kstra, like m**ay**.

best path between two nodes in a graph, and it is a precursor of the A* algorithm. Among other topics, Appendix B gives an introduction to graph traversal algorithms, also known as graph search algorithms.

We present the two steps of our graph-based reassembly algorithm:

1. We first build a graph that contains all the reassembly path, i.e., all the consecutive actions that can lead to a complete reassembly. The nodes of the graph contain the action of placing a fragment in a position. The graph is similar to a tree of consecutive actions, where all the leaves are linked to the endgame state T[7] (Figure 6.4).
2. From the graph, we apply Dijkstra's algorithm to find the best path from the empty puzzle situation to the endgame state.

[7] T stands for target, and S designates the source of the shortest path.

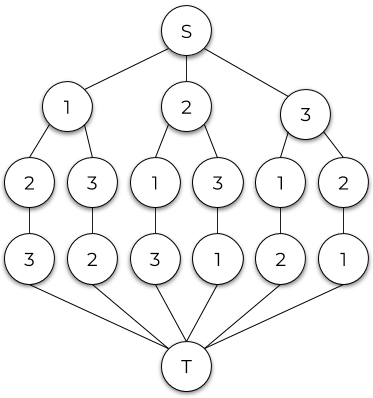


Figure 6.4: Graph obtained with 3 fragments and 3 positions.

Graph building Given a set of fragments and empty positions, there are many ways to reach a chosen solution, which is uselessly time-consuming for Dijkstra's algorithm. To avoid such redundancy, we set the order of the fragments: we place the first fragment first, then the second fragment, and so forth. The fragments' order does not matter because the predictions of the position only depend on the central fragment, not on the already placed fragments. We could have set the order of the positions, but it would have been more complicated in the case of extra fragments.

To build the graph of the reassembly paths, we design a recursive algorithm. Starting from an empty puzzle S, we decide where to place the first fragment i. We model this decision by p nodes connected to S. The negative logarithm of the classification scores weights the edges. Then, each node is connected to the remaining positions that can be attributed to the second fragment, and so on. The last fragment is placed in the last remaining position, and it is connected to the end of the graph T. These last edges are given a null weight. In other words, the depth of the graph corresponds to fragments, and the width is the available positions.

Algorithms 2 and 3 detail how the graph G is built. The first one introduces the initialization step that launches the recurrence:

Algorithm 2: Initialization of the graph building.

```
1: procedure CONSTRUCT_EDGES(Y)
2:     empty_pos ← [1 .. p]
3:     used_pos ← [S]
4:     frag ← 1
5:     G ← ADD_EDGE(Y, empty_pos, used_pos, next)
6:     return G
7: end procedure
```

where G is a list (dictionary) of edges. Each edge is described by the current fragment, the position of the previous fragment, the position of the current fragment, and the cost of the edge:

- The current fragment refers to the fragment added to the graph, which corresponds to its row number. We use None to describe the last row's current fragment.
- The position of the previous fragment is the last item of the $used\_pos$ array that tracks down all the positions filled from the current path. We note it: $used\_pos[-1]$.
- The position of the current fragment is either part of $empty\_pos$ array or T.
- The cost of the edge is the negative logarithm of the classification scores stored in Y. The cost of those that go to T is null.

For instance, the first edge that is appended to G with ADD_CHILDREN is S—1, which means the first fragment is placed on the empty state S, in position 1 (Figure 6.4). We note it: $(1, S, 1, Y[0, 0])$.

Algorithm 3 presents a recursive method that append edges to G.

Algorithm 3: Adding edges in the graph.

```
1: procedure ADD_EDGE(Y, empty_pos, used_pos, frag)
2:     G ← [ ]
3:     if empty_pos is empty then
4:         G.append(None, used_pos[−1], T, 0)
5:         return G
6:     end if
7:     for pos in empty_pos do
8:         G.append(frag, used_pos[−1], pos, Y[frag − 1, pos − 1])
9:         empty_pos.pop(pos)
10:        used_pos.append(pos)
11:        G.append(ADD_EDGE(Y, empty_pos, used_pos, frag + 1))
12:    end for
13:    return G
14: end procedure
```

DIJKSTRA In order to find the most likely reassembly, we compute the shortest path from S to T to minimize the sum of the weights between visited nodes, which corresponds to the solution of Equation 6.6.

The path's length equals the sum of the its edges weights. In brief, Dijkstra's algorithm always explores the edge that minimizes the path's weight, as described in Algorithm 4:

where:

- P is a list of the explored nodes, i.e., those whose distance to S is shorter than the distance between S and T;
- $predecessors$ is a dictionary that indicates for each node of P its best parent;
- $scores$ is a list that groups all the shortest distance to S for each

Algorithm 4: Shortest path.

```
1: procedure DIJKSTRA(G, S, T)
2:     P ← [ ]
3:     predecessors ← [None]×nb_nodes(G)
4:     scores ← [+∞]×nb_nodes(G)
5:     scores[S] ← 0
6:     while T not in P do
7:         n ← FIND_NODES(G, P, scores)
8:         P.append(n)
9:         UPDATE(n, scores, predecessors)
10:    end while
11:    shortest_path ← GET_PATH(P, predecessors)
12:    return shortest_path
13: end procedure
```

node. Therefore:

$$\begin{aligned} scores[i] =& scores[predecessors[i]] + \\ & scores[predecessors[predecessors[i]]] + \\ & ... + scores[S]; \end{aligned}$$

- FIND_NODE is a method that find the best node n to add to P, which has the lowest $scores$ while not being part of G;
- UPDATE is an in-place method that update the $scores$ and $predecessors$ of each child node of n, only if the new $scores$ are better.
- GET_PATH is a method that goes up the list of $predecessors$ to find the shortest path.

COMPLEXITY The size of the resulting graph is $|N| = 2 + \sum_{l=0}^{f-1} \prod_{k=l+1}^{f} k$ for the number of nodes and $|E| = |N| - 2 + nodes(f-1)$ for the number of edges, with f the number of fragments and p the number of positions. See details in Appendix C. With 8 fragments and positions, this corresponds to $|E| \approx 1.5 \times 10^5$ and $|N| \approx 10^5$.

The complexity of graph building algorithm is $\mathcal{O}(|E|)$, because each time the for loop is called, an edge is added.

The complexity of Dijkstra's algorithm is $\mathcal{O}((|E| + |N|) * |N| + p)$. In the worst case, the while loop is applied $|N|$ times; the FIND_NODE loops through the elements of $G \backslash P$ (i.e., $\mathcal{O}(|N|)$; the UPDATE edits the score for all the children of the current node, which is $|E|$ in the worst case; the GET_PATH selects p fragments. We use a more effective version of Dijkstra's algorithm, whose complexity is $\mathcal{O}(|E| + |N| log(|N|) + p) \simeq \mathcal{O}(|E| + |N| log(|N|))$. Because we use a tree-shaped graph, our worst case complexity is lower that Dijkstra's.

PERFORMANCE Dijkstra's algorithm ensures we will find the best path, which was not the case with the greedy algorithm. It corrects the neural network's local mistakes and obtains the best average score, as illustrated in Table 6.2.

	Positions		
Fragments	#1	#2	#3
#1	0.4	0.4	**0.2**
#2	**0.7**	0.2	0.1
#3	0.1	**0.5**	0.4
Greedy score			0.466

	Positions		
Fragments	#1	#2	#3
#1	0.4	**0.4**	0.2
#2	**0.7**	0.2	0.1
#3	0.1	0.5	**0.4**
Dijkstra score			0.5

Table 6.2: Toy example that illustrates the performances of the greedy algorithm versus Dijkstra's.

Central fragment If we do not know the central fragment, we perform the central fragment selection as a first step. The first expansion from S consists of all the possible cases where each fragment is used as the central fragment. The corresponding sub-graphs are built using Algorithm 2. The resulting graph's size is unchanged, except we have $n+1$ fragments, with n the number of the fragment to be assigned to a relative position. With $n = 8$, we obtain $|E| \approx 1.3 \times 10^6$ and $|N| \approx 10^6$. We show in Figure 6.5 a simplified example with 3 fragments and 2 relative positions.

Missing fragments Our standard algorithm can build graphs with missing fragments without requiring any change. The only difference is that the number of rows will be inferior to the number of positions. Details on $|N|$ and $|E|$ are given in Appendix C.

Extra fragments We now consider the case where we have more than 8 fragments, coming from various sources. It implies that any fragment can be discarded (i.e., placed at Ø position). We construct a graph allowing such configurations. A simplified example of the graph is shown in Figure 6.6.

$|N|$ and $|E|$ are described in Appendix C. In the case of 10 fragments and 8 relative positions, the size of the graph is $|E| = 5 \cdot 10^9$ and $|N| = 4 \cdot 10^8$.

The graph building algorithm is similar to the Algorithm 3, as detailed in Algorithm 5:

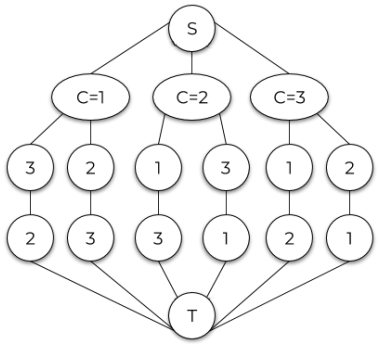


Figure 6.5: Graph with unknown central fragment.

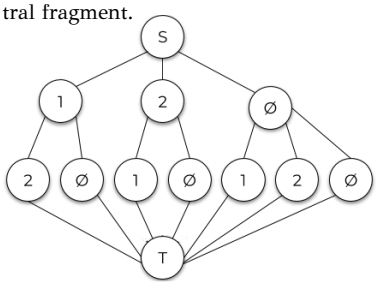


Figure 6.6: Graph allowing empty positions.

6.5.3 *Cuts in the graph*

To tackle the graph method's complexity, we cut the branches that display a weight lower than a threshold θ. Such branches correspond to a low placement probability, which in turn produces a low reassembly probability due to the multiplicative property of Equation 6.5. Cutting improves our computation time significantly and the number of fragments we can take into account. If the value of a relative position prediction comes under a specific threshold, the branch is not connected to the trunk T (see Figure 6.7).

```
1: procedure ADD_EDGE(Y, empty_pos, used_pos, frag)
2:     G ← [ ]
3:     if empty_pos is empty or frag > p then
4:         G.append(None, used_pos[−1], T, 0)
5:         return G
6:     end if
7:     for pos in empty_pos ∪ Ø do
8:         if pos in empty_pos then
9:             G.append(frag, used_pos[−1], pos, Y[...])
10:            empty_pos.pop(pos)
11:        else
12:            G.append(frag, used_pos[−1], Ø, 0)
13:        end if
14:        used_pos ← used_pos ∪ pos
15:        G.append(ADD_EDGE(Y, empty_pos, used_pos, frag + 1))
16:    end for
17:    return G
18: end procedure
```

Algorithm 5: Graph building with empty positions.

As the shortest path starts from the trunk T and not from S, the graphs on Figures 6.7 and 6.8 are equivalent. However, the latest is quicker to build, as it is smaller than the others. Thus, the sooner the cuts occur, the better it is. This observation leads to a reordering of the graph rows: the first fragments we place are these that allow the most of cuts.

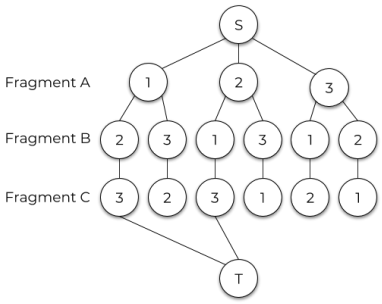


Figure 6.7: Graph with a cut of the fragment C for positions 1 and 2, without reordering.

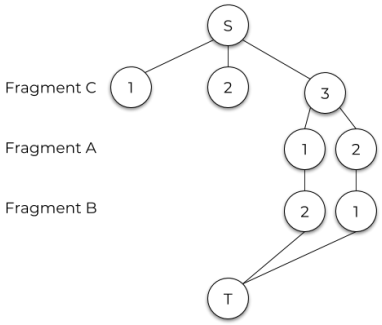


Figure 6.8: Graph with a cut of the fragment C for positions 1 and 2, with reordering.

6.6 EXPERIMENTS

6.6.1 *Training procedure*

We program the neural network with Keras and TensorFlow libraries.

We train it before proceeding to the reassembly stage, which relies on the trained classifiers with either 2, 8, and 9 outputs. We run a grid search on the following hyper-parameters: optimizer (SGD versus Adam), learning rate, momentum, and FEN output size. The loss function is the categorical cross-entropy.

We introduce the following training setups:

- MET setup: We train and evaluate the network on MET;
- ImageNet setup: We train and evaluate the network on ImageNet;
- Transfer setup: We train the network ImageNet, then we fine-tune and evaluate it on MET.

We compare the accuracy of our FEN standard architecture presented in Table 6.1 with Resnet [HZRS16] and our implementation of Doersch et al.'s [DGE15], which has fewer parameters (we reduced the fully connected layers from 4,000 to 512 neurons).

Table 6.3 shows the standard parameters for our experiments.

FEN output size	512
Optimizer	SGD
Learning rate	0.1
Momentum	0.9

Table 6.3: Summary of the experiments parameters.

6.6.2 *Reassembly metrics*

We introduced the evaluation metrics in §2.3. For Deepzzle, we use the *solved puzzles* and the *well-placed fragments* metrics. We do not count the *correct neighbors* because all our fragments are placed in relation to the central fragment only.

We introduce one last metric, which is the *almost-perfect solved puzzles*. In numerous images of the dataset, we have few indistinguishable background fragments (see Figure 6.9), which lead to a random prediction that scores poorly with the previous metrics. However, we look for a visually plausible solution rather than the exact one: some archaeological puzzles contain similar fragments that can often be swapped, e.g., the limestone blocks of a Roman temple.

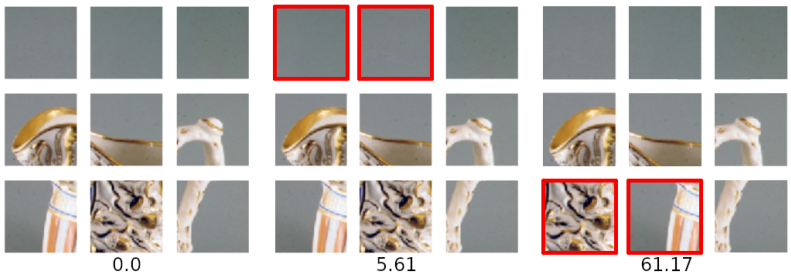


Figure 6.9: Selection of the best threshold for the almost-perfect metric. The red outline shows the fragments that are misplaced. The case described by the third image is typical: the upper fragments are so similar that they are swapped. The values below the reassemblies are the difference between the prediction and the solution.

We consider as successful any reassembly where similar fragments are swapped. Then, we introduce a metric that reflects this objective of visually acceptable reassembly. It evaluates the number of almost correct reassemblies by measuring the similarity between fragments i_1 and i_2. We use the Frobenius norm $||\cdot||_F$ and introduce a threshold θ, so that if $||i_1 - i_2||_F < \theta$, we consider the fragments similar. Therefore, when two similar fragments are swapped, the puzzle is still considered correctly reassembled if the norm of the difference between the fragment of the solution and the fragment of the predicted reassembly is below a threshold.

In Figure 6.9, we show an example of the threshold values based on the fragments that are misplaced. We performed a statistical analysis and set the threshold to $\theta = 20$, as this value confuses most of the similar fragments without allowing wrong switches.

6.6.3 *Dataset*

We train our neural network on ImageNet [DDS+09] (1,181,167 training images and 100,000 validation images) or MET (10,000 training images and 4,000 validation images). At each epoch, we use different

crop within the images, making unique puzzles, and the provided coordinates for the reassembly phase. We consider a single pair of fragments per image. We normalize the values between -1 and 1.

Artwork 4: *The Princesse de Broglie*, Jean Auguste Dominique Ingres, 1851–1853, from the MET Open Collections.

7

Deepzzle's results

◂ Chapter 6 Chapter 8 ▸

Synopsis This chapter presents the results we obtained with Deepzzle. We start by analyzing the neural network and the reassembly algorithms §7.1, and we continue with the reassembly score on the major tasks §7.2. Section §7.3 examines the scores we obtain for various data, notably heritage datasets.

7.1 Benchmarks and comparisons

This section compares the effects of parameters on the neural network (architectures, setups, merging function, and classes quantity) and on the reassembly (algorithms, branch-cut).

7.1.1 *Neural network scores*

Doersch versus ours Figure 7.1 shows the evolution of the validation accuracy while training on ImageNet, for our implementation of Doersch et al.'s architecture and ours, based on VGG. It outperforms the 40% reported in [DGE15], achieving 57% accuracy on ImageNet. Our own architecture reaches 64.6%, which significantly outruns [DGE15]'s scores by a 25% margin.

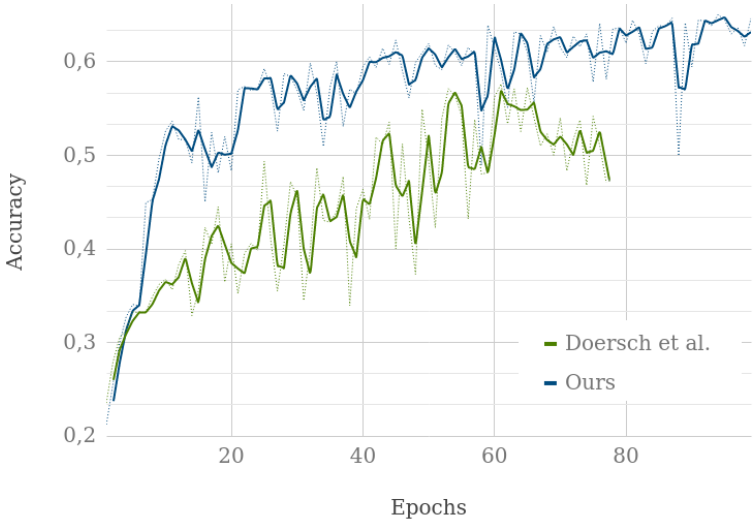


Figure 7.1: Validation accuracy scores — Comparison of our architecture and Doersch et al.'s.

Setups Table 7.1 compares the validation accuracy of the three setups on the 8-classes task with concatenation. It shows that learning to predict the relative position is easier on ImageNet dataset. We assume it is mostly due to the solid background of many MET images (Figure 6.9), which makes impossible to correctly place some fragments. Another reason may be that the network trained on ImageNet learns the lens aberrations, making the puzzles easier to solve, but the high score on the Transfer setup discards this hypothesis: the MET puzzles do not contain lens aberrations, so if the network was relying on these, it could not have reached the obtained accuracy.

MET	ImageNet	Transfer
48.9%	64.6%	59.7%

Table 7.1: Validation accuracy scores — Comparison between the setups.

Merging function Table 7.2 reports the validation accuracy for different merging functions on the 8-classes problem on the ImageNet setup. The Kronecker product obtains slightly better results than the concatenation. In comparison, the low-rank approximation of Hadamard product [KOL+16] yields lower results, which implies that the full covariances are needed to obtain the best performances.

Fusion	Accuracy
Concatenation	64.6%
Kronecker product	66.4%
Hadamard product	59.2%

Table 7.2: Validation accuracy scores — Comparison between the three fusion strategies.

Classes number Last, Table 7.3 compares the validation accuracy with extra-fragments for different neural network. In this experiment, we use the Kronecker product.

In Deepzzle, enabling extra-fragments implies 9 classes, which is equivalent to filter the extra-fragments (with a binary classifier) before applying the 8-classes classifier. For the binary classification problem, we set the proportion of extra-fragments to 50%. We obtain 92.5% accuracy, which means that deciding whether two fragments belong to the same image seems an easy problem. For the 8-classes problem, we obtain 66.4% accuracy. Therefore, the combination of the filter and the 8-classes neural networks leads to an accuracy of 61.4%. Finally, the joint classification problem achieves 64.2% (the proportion of fragment belonging to the same image was set to 70%), which indicates that solving the joint problem is slightly easier than solving the sequence of simpler problems.

Neural network	Accuracy
Binary classifier	92.5%
8-classes	66.4%
9-classes	64.2%

Table 7.3: Validation accuracy scores — Comparison of the 2-classes, 8-classes and 9-classes problems on ImageNet.

7.1.2 *Reassembly scores*

On the standard 3×3 task, we solved perfectly 44.4% of the puzzles, for 89.9% of well-placed fragments. Those scores are our reassembly baseline for the 8-classes neural network.

The fragment-wise score is much better than the 66% accuracy of the neural network, so we argue that the reassembly step removes some classifier's uncertainty. This hypothesis is corroborated by the scores of the task with missing fragments (see below, Table 7.4): because we have less fragments, we cannot rely on already placed fragments to determine the positions of the fragments for which we hesitate.

Figure 7.2 is a perfect illustration of the kind of results we achieved. We made a thorough analysis of it to expose the "reasoning" of Deepzzle: Most of the fragments share color and shape continuity with respect to the central fragment, except the two top corner fragments that display similar probabilities to be in any of the top position. Indeed, the middle-top fragment shares a part of the left green curtain. The top-right fragment is placed last: it displays a uniform probability for every top position, as its primary color is not part of the central fragment. It is placed correctly because other fragments have been assigned to their correct location before, thanks to their higher probability.

We also illustrate the effects of the almost-perfect metric in Figure 7.3. In this puzzle, most of the painting fragments are neutral background fragments: finding which fragment goes where is a random guess. Thanks to the metric, the reassembly is correct.

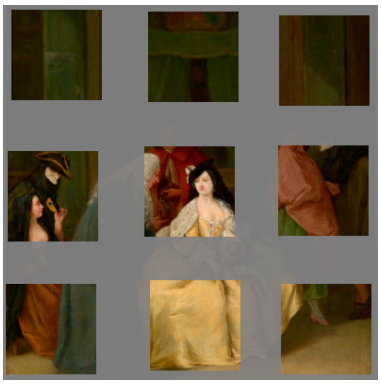


Figure 7.2: A typical reassembly.

Figure 7.3: An almost-perfect reassembly. The yellow outline indicates almost-perfectly placed fragments.

Greedy We first compare the greedy algorithm to the graph-based algorithm on the MET dataset with the 8-classes classifier. The scores are presented in Table 7.4 and show that, on average, only 2 fragments are swapped per image. Dijkstra's algorithm leads to a general 3% puzzle-wise improvement over the greedy-algorithm.

Algorithm	Task	Puzzle-wise	Fragment-wise
Greedy	Standard	41.0%	87.7%
Graph-based	Standard	44.4%	89.9%
Greedy	Missing fragments	26.5%	80.5%
Graph-based	Missing fragments	29.5%	82.4%
Greedy	Unknown center	36.2%	69.5%
Graph-based	Unknown center	39.2%	71.1%

Table 7.4: Reassembly scores — Comparison with the greedy algorithm.

It appears that missing fragments increase the difference between well-placed fragments and well-solved puzzles: some easy fragments that help remove uncertainties may have been missing. Last, the scores of the task with an unknown center display a bigger difference between the fragment-wise and puzzle-wise scores, which indicates that the average number of misplaced fragments is higher, probably

because some puzzles are shifted. We detail those scores in §7.2.

End-to-end We perform a comparison between our method and the first permutation-based method, from Noroozi and Favaro [NF16]. We reproduce their setup and apply it to the MET dataset for 10, 100, and 1000 permutations. We use our architecture to extract the features of each fragment, i.e., before the feature merging. To compare with our method, we cut the tree so that the authorized paths correspond to the allowed permutations. We use our 8-classes network, and we use graph solving for an unknown central fragment. The results are exposed in Table 7.5.

	Available permutations			
	10	100	1000	9!
Noroozi and Favaro	86.6%	69.3%	51.6%	-
Ours with unknown center	91.5%	81.7%	64.8%	39.2%

Table 7.5: Puzzle-wise reassembly scores — Comparison with Noroozi and Favaro [NF16].

We observe that our process greatly surpasses Favaro and Noroozi's in reassembly scores. As we apply pairwise comparison on the input fragments, we can see it as a subtask of the permutation classification. We better guide the learning process. Moreover, we recall that our method offers two other benefits over Favaro and Noroozi's: it covers all the possible permutations and handles outsider fragments.

Branch-cut evaluation We evaluate the trade-off between accuracy and computational time for different threshold values in our branch cut strategy in Figure 7.4. As a baseline, solving a full 3×3 puzzle takes about 20,000 s. Setting the threshold θ to 0.01 allows us to gain an order of magnitude without any loss of accuracy. Setting θ to 0.05 leads to a gain of 3 orders of magnitude, or about 20s per reassembly, with a marginal loss of accuracy. We consequently use a threshold of 0.05 in the following experiments.

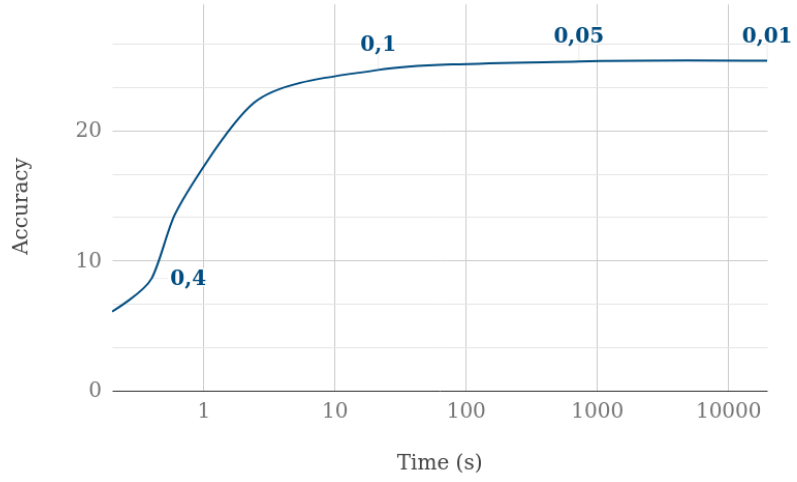


Figure 7.4: Reassembly scores — Comparison of the reassembly time for various cut values.

7.2 ADVANCED REASSEMBLY TASKS

This section analyzes our results on the advanced reassembly tasks our method can address, which are the problems of an unknown central fragment, missing fragments, and extra fragments. We start with the unknown center.

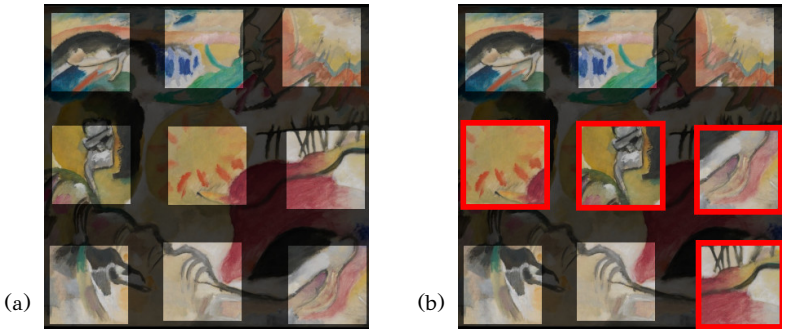
7.2.1 *Reassembly with unknown center*

Table 7.6 compares the reassembly score with and without known central fragment. In the second case, the algorithm has to perform many reassemblies for each fragment being assumed center; then it has to select the best reassembly based on the reassembly score. We use the same 8-classes architecture, with a Kronecker product. We observe a 5% drop of the reassembly accuracy.

	Puzzle-wise	Fragment-wise
Center known	44.4%	89.9%
Center unknown	39.2%	71.1%

Table 7.6: Reassembly scores — Comparison of for known and unknown central fragment.

As mentioned before, some reassemblies obtained from an unknown central fragment are shifted, which means that most neighbors are correct while the fragments' positions are all wrong. Another type of result is illustrated on Figure 7.5. It shows that some fragments of the reassembly are well-placed despite a wrongly-placed center.



(a) (b)

Figure 7.5: Example of a wrong reassembly with unknown center. The red outline shows the fragments that are misplaced — Fig. (a) shows the expected outcome, and Fig. (b) the predicted result.

7.2.2 *Reassembly with missing and additional fragments*

QUANTITATIVE SCORES Table 7.7 presents all the reassembly scores for 0 to 7 missing fragments and 0 to 3 extra fragments. We use the almost-perfect metric rather than the puzzle-wise metric, except on the first line, and a 9-classes neural network, trained with a 10% probability to sample an additional fragment.

The middle section of the table indicates how many puzzles turn to (almost-)perfect reassemblies. The bottom section displays the number of well-placed fragments and empty tiles. We draw the

		Number of extra fragments			
		0	1	2	3
Puzzle-wise	0	22.1%	18.4%	16.8%	15.4%
Almost perfect puzzle-wise Number of missing fragments	0	24.7%	19.9%	18.3%	16.9%
	1	20.8%	12.9%	11.3%	11.0%
	2	21.1%	10.6%	8.8%	8.3%
	3	22.6%	12.0%	9.8%	6.5%
	4	24.9%	12.2%	8.4%	6.8%
	5	31.1%	16.6%	10.9%	8.3%
	6	43.4%	22.7%	13.9%	10.6%
	7	64.0%	33.7%	21.0%	13.0%
Fragment-wise Number of missing fragments	0	64.6%	62.8%	60.9%	60.3%
	1	61.6%	59.4%	57.9%	57.0%
	2	61.1%	57.8%	55.7%	55.4%
	3	63.0%	59.6%	57.3%	54.6%
	4	66.9%	62.0%	58.3%	56.0%
	5	72.4%	66.9%	62.2%	59.3%
	6	80.0%	73.5%	67.5%	62.0%
	7	82.0%	81.1%	73.6%	67.6%

Table 7.7: Reassembly scores with missing and outsider fragments.

following conclusions:

- On the standard task: We obtain 64.6% of well-placed fragments for only 24.7% of almost-correctly solved puzzles. It means that we often make a few errors in the reassemblies. We evaluated the correctly-placed fragment among the not solved puzzle to strengthen our observation and obtained an average score of 55%.
- On the task difficulty: According to the tables, we obtain the best scores when no fragment is missing or when many fragments are missing. First, when we have all the pieces, we can discriminate similar fragments and select the best one for each location by optimizing the full reassembly. When there are several missing fragments, there is much less information available to assess which one goes where. On the other end of the spectrum, when almost every fragment is missing, the odds we sample the most ambiguous image fragment are low.
- On the almost-perfect puzzle-wise metric: The almost-perfect metric improves the score by 2.1% on the standard task (no missing nor extra fragment). Overall, we observe a gain of at least 1.5% over the standard puzzle-wise metric, which indicates that our dataset contains at least 2.1% of highly similar fragments[1].
- On the effect of extra-fragments: The results indicate that the more we consider external fragments, the lower the number of almost-perfect reassemblies is, as the number of possible solutions increases.

[1] An approximation of 4% seems reasonable because two equivalent fragments have half chances to be well-placed.

- On the 7-missing fragments puzzles: We observe that, when we only have to place one fragment, we obtain roughly 64% images solved for 82% correctly-placed fragments. It is because the central fragment is always well-placed, which raises the fragment-wise score.
- On the number of classes: The results obtained by the 9-classes classifier are less precise by 20% than the reassemblies given by the 8-classes classifier (Table 7.6). However, the accuracies of the networks are similar. We come with two ideas that may explain the loss of precision during the 9-classes reassembly: Dijkstra may exclude correct fragments or the neural network give a similar score to the lateral positions classes, and so Dijkstra does not place them well.

Reassemblies with missing fragments Figure 7.6 shows puzzles with four or five missing fragments. Most of the time, the remaining fragments are placed correctly. When a mistake occurs, it usually respects the picture's semantics, as we can see in the central puzzle of Figure 7.6.

Figure 7.6: Reassemblies with missing fragments.

Reassemblies with extra-fragments Figure 7.7 exhibits five puzzles with extra-fragments, their reassemblies and their solutions. We made a qualitative analysis on about twenty puzzles, and picked a correct reassembly (a) and four wrong reassemblies to illustrate our point. We conclude the following:

- Overview: At first glance, the algorithm tends to replace missing fragments by outsider fragments (puzzles (b) and (d)). This observation fits with our analysis of Table 7.7. The switch of missing fragments by extra fragments is especially common for background fragments from clothing, shards, and sculptures photographs (puzzle (b)). Two other categories of images are prone to be reassembled with outsiders fragments: texts (puzzle (d)) and engravings. Conversely, paintings are less exposed to this effect (puzzles (a) and (e)), especially when the additional fragments come from non-painting images. When there are no missing fragments (puzzle (e)), most of the reassemblies errors are due to

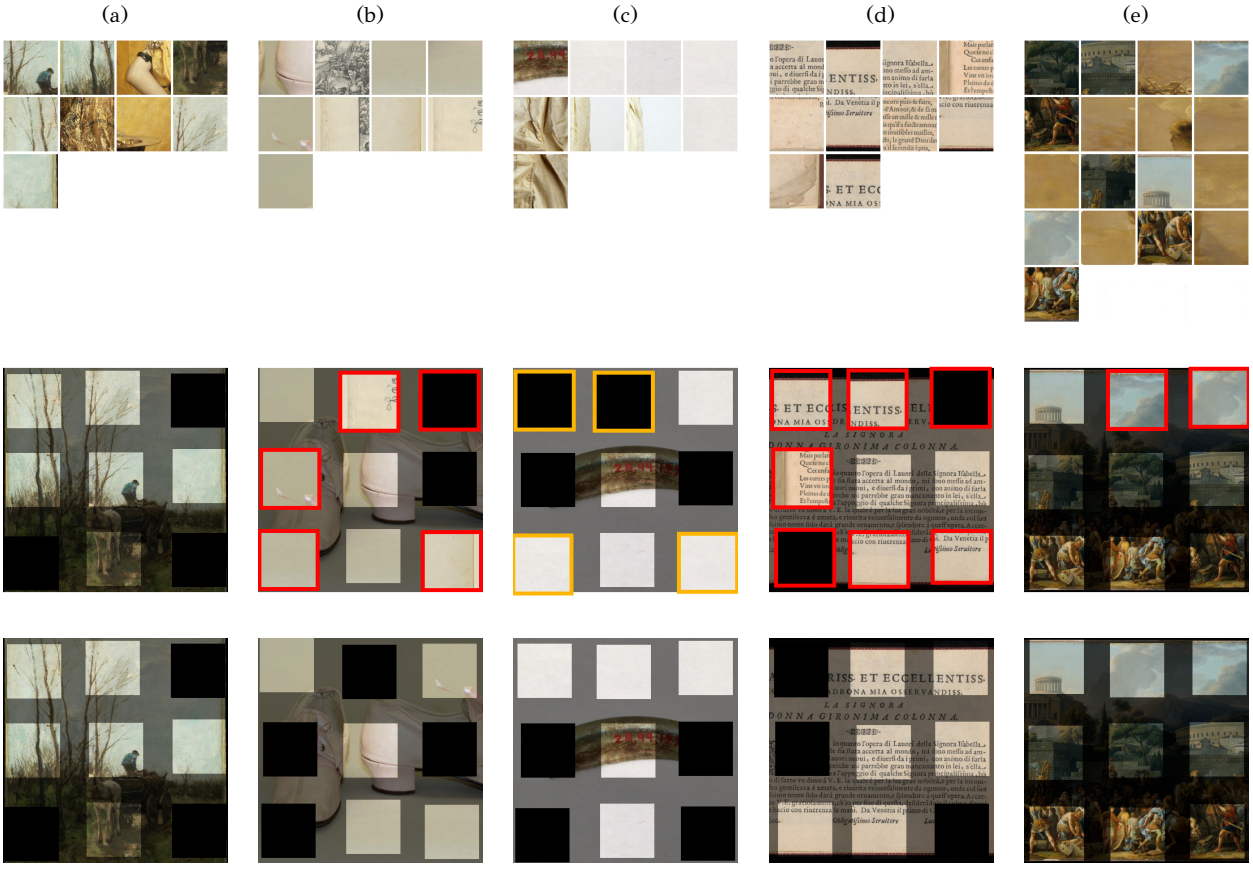


Figure 7.7: Various reassemblies with outsider fragments. The first row contains the input fragments. The first (top left) emplacement is reserved for the central fragment. The second row shows the predicted reassemblies. The last row displays the solutions. The red outline indicates wrongly placed fragments. The yellow outline shows the almost-perfectly placed fragments.

misplacing the image's fragments rather than the replacement of a correct fragment by an additional fragment.

- Puzzle (b) is a typical example of what wrong reassemblies look like: two missing fragments were replaced by similar outsider fragments that contain a mostly-beige background.
- The shard of puzzle (c) is almost-perfectly reassembled, as only background fragments were to be placed.
- Puzzle (d) illustrates the reassembly of a text when another text is the source of the outsider fragments. We obtain poor results (only one fragment is correctly placed), but the text's spatial coherence is respected. The title is positioned on the top of the image. The fragment that contains the end of the subtitle is at the right of the other title fragment. The end of the text is also placed on the bottom. The italic closing formula is on the right of the other bottom fragment. Finally, the algorithm uses the outsider fragment that contains a left margin at the left of the central fragment. However, the algorithm cannot distinguish between the French and Italian languages, which suggests the convolutional architecture cannot learn fine-grain details. It illustrates a limitation of Deepzzle: the input resolution is too small to allow the neural network to capture such details and produce precise alignment.

Deepzzle is intended to solve coarse alignments and thus works best for puzzles with large visual features and sufficient image resolution.

- Puzzle (e) is an example of reassembly with a relatively high number of fragments. The algorithm swapped the cloudy sky fragments. As they are too different pixel-wise, even the almost-perfect metric does not grant the correct reassembly label. Note that to a human eye, the computed reassembly looks realistic with the cloudy sky reversal.

NOTE ON THE COMPUTING TIME Thanks to the cutting strategy, we were able to compute reassemblies from a set of 17 fragments quickly. We spend approximately one hour on constructing the graph and applying the shortest path algorithm. Without it, processing more than 3 outsiders fragment could take several months.

7.3 IMPACT OF DATA ON REASSEMBLIES

In this last section of Deepzzle's result, we present our work applied to some specific datasets. We apply Deepzzle to the datasets we mentioned in Chapter 4. Then, we deepen our research on MET, dividing it into three categories (paintings, artifacts, and engravings, including texts) and detailing our results on MET's texts and MET-based patchworks datasets.

7.3.1 *Other datasets*

Table 7.8 shows the reassembly scores we obtained on other dataset and Figure 7.8 shows two reassemblies for each dataset. We run the experiment under the 8-classes architecture, with fine-tuning for ImageNet and Bas-reliefs datasets, as they contain enough data to allow it.

Dataset	Puzzle-wise	Fragment-wise
ImageNet	48%	78%
Bas-reliefs	40%	61%
Vaux-de-la-Celle (3D scans)	28%	63%
Roc-aux-Sorciers	0%	31%

Table 7.8: Reassembly scores on various datasets.

Without surprise, the scores on ImageNet are higher than the scores on MET. Puzzles (a) and (b) illustrate the type of reassemblies Deepzzle achieves on ImageNet.

Next puzzles, (c) and (d) are from the bas-relief dataset. According to Table 7.8, they are usually well solved, and their scores are close to the MET dataset's scores. We note that the main issue of puzzle (d) is due to similar patterns. It suggests we should improve the almost-

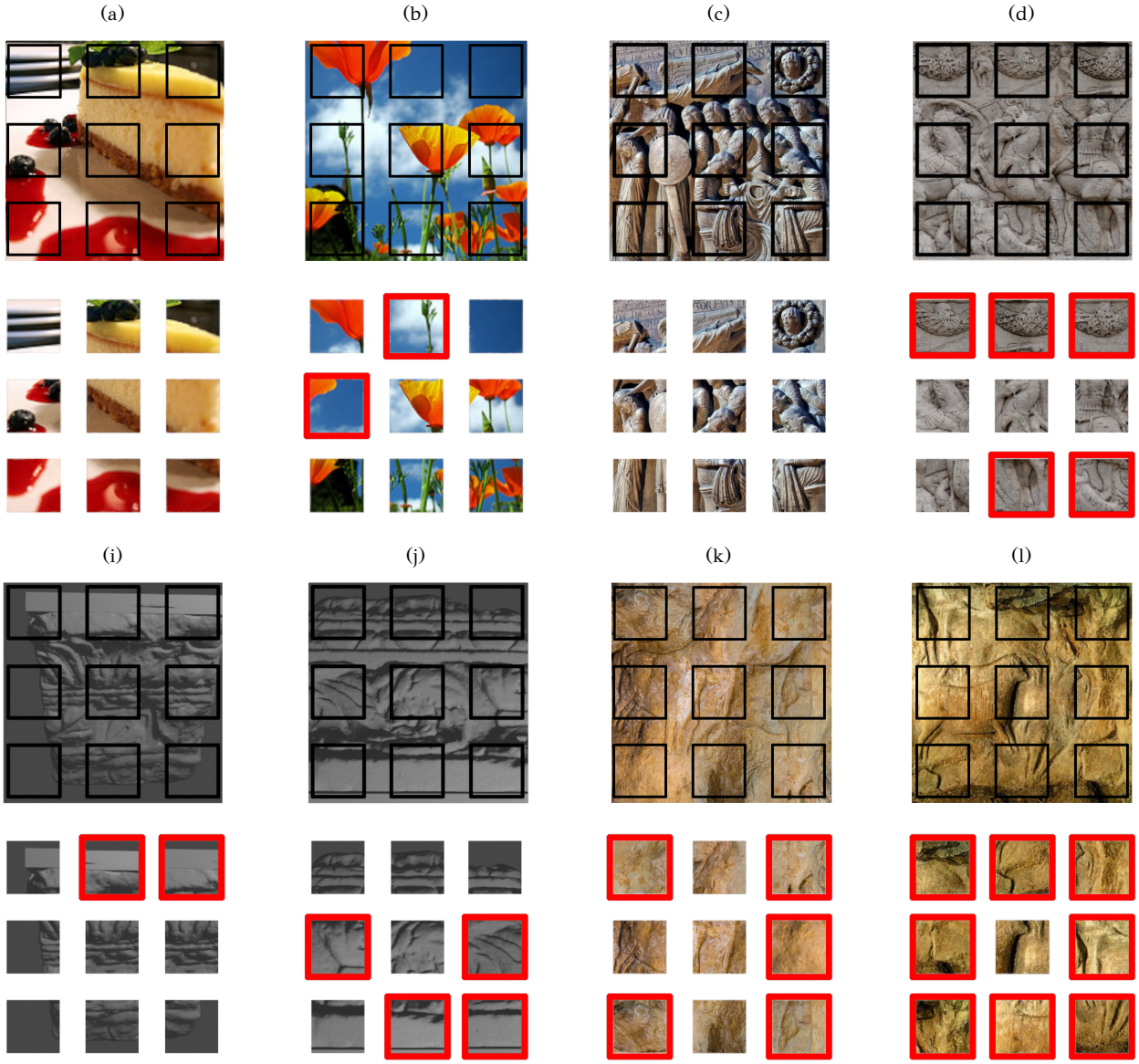


Figure 7.8: Predicted reassemblies (odd rows) and their solutions (even rows) for various datasets. The red outline shows the fragments that are misplaced.

perfect metric to be able to accept the current top-row reassembly, for example, by using a deep classifier rather than a distance.

From Vaux-de-la-Celle's blocks dataset, we picked the puzzles (i) and (j) that are representative of the average problem. Well-solved puzzles are rare, but most fragments are usually well-placed.

Last, puzzle (k) is from our dataset made of Roc-aux-Sorciers's photographs. It contains 20 images, and Deepzzle did not solve any of them. We suspected the low contrast and uniform shades, so we try some more contrasted puzzles made from the bas-relief of Roc-aux-Sorciers, like puzzle (l). In this last puzzle, all the fragments are misplaced; we obtain similar reassemblies from the few high-contrast parietal bas-reliefs we tried, which invalidates our hypothesis.

7.3.2 *MET: Reassembly depending on the type of object*

In Table 7.9, we compare the reassembly scores for the three major types of images of our dataset (artifacts, engraving and texts, and painting), on the standard 9-fragments task. The types of images are almost homogeneously distributed.

Type of image	Puzzle-wise	Fragment-wise
Artifact	38.2%	70.6%
Engraving and texts	25.5%	68.0%
Painting	12.1%	56.2%
Dataset	24.7%	64.6%

Table 7.9: Reassembly scores depending on image type.

The puzzle-wise score of paintings is surprisingly low. It means it is harder to reassemble painting puzzles despite their semantic consistency. As the fragments-wise score is not as low as we can expect based on the image reassembly score, we conclude that most of the paintings' reassemblies only had very few misplaced fragments.

On the contrary, the artifacts score well, primarily because of the almost-perfect metric: the artifacts always have a neutral background (see puzzles (b) and (c) from Figure 7.7).

Table 7.10 describes the scores obtained with two additional fragments extracted from various types of images. In this experiment, there are no missing fragments.

Image	Extra fragments	Puzzle-wise	Fragment-wise
Artifact	Artifact	33.0%	69.2%
Artifact	Engraving	32.7%	69.9%
Artifact	Painting	31.9%	69.5%
Engraving	Artifact	22.2%	67.2%
Engraving	Engraving	14.1%	63.3%
Engraving	Painting	21.4%	67.0%
Painting	Artifact	11.5%	54.9%
Painting	Engraving	12.5%	56.7%
Painting	Painting	11.1%	54.6%
Dataset		17.3%	60.9%

Table 7.10: Reassembly scores depending on image type, with extra-fragments.

When the image is of an artifact, we obtain the best results on the puzzle-wise score compared to another artifact image. Interestingly, the best fragment-wise score is obtained when adding two fragments from an engraving (or texts): we suppose Deepzzle can easily discard them, while it "doubts" more when the extra fragments come from another artifact (probably with similar background).

When trying to reassemble an engraving (or a text), the best score goes to the artifact additional fragments, closely followed by the painting fragments. The main reason is that engraving or text are monochromatic images, while photographs of paintings and artifacts usually come in various colors. Thus, it is more difficult to discriminate against the outsider fragments when they come from another engraving or text. It is also why the additional fragments of the engraving score well for the artifacts and the paintings.

Last, the results for paints are homogeneous regardless of the type of fragment added.

7.3.3 *MET: Reassembly from texts*

In this section, we analyze in detail the reassemblies of texts shown in Figure 7.9. We aim to gain insights into the patterns used by the neural network to make its text-based predictions. We select thirty text pictures from the MET dataset. In this sample, we obtained 24% of perfect reassemblies and 68% of well-placed fragments, which is consistent with Table 7.9. We made the following observations:

- Puzzle (a) is a perfect example of confident reassembly: most fragments positions are predicted with a confidence score superior to 70%. In this image, the only fragments whose correct class is not the most confident are the upper right fragments (24% for the upper right position, against 36% for the bottom left position).

- The central fragments of puzzles (b) and (c) contains clues about how to solve the puzzle. Looking at the central fragment of puzzle (b), we have an image on top that probably stretches out over the top fragments. We also have text at the bottom left and at the bottom right of the central fragment, with a space between them. By extending all of these structures, one can easily solve the puzzle. Each relative prediction is correctly predicted with confidence over 50%.

- Puzzles (d), (i), (j), (k) and (l) shows a central text fragment. In puzzles (d) and (i), the lateral fragments contains text and margin in the four directions, and are well reassembled. Puzzle (j) is perfectly reassembled by chance, as the left and right fragments vertical position display very close classifications scores. Puzzles (k) and (l) contains the same puzzle, with a vertical shift.

- Puzzle (k) is interesting, as most title fragments were placed at the bottom of the puzzle. These two fragments are similar text fragments because there is no space between the top of the fragments and the horizontal ornamentation. We suppose this similarity is the cause of the misplacement. On the contrary, the upper left fragment contains space before the frieze: then, it cannot continue the text. The correct position of the title in puzzle (l) supports this idea. Looking to first predicted class scores in puzzles

Figure 7.9: Predicted reassemblies (odd rows) and their solutions (even rows) for texts. The red outline shows the fragments that are misplaced.

(k) and (l), we observe a strong vertical arrangement with close position scores (with a difference lower than 5% between the vertical positions scores).

- In puzzle (q), the reassembly display mistakes on similar fragments. In puzzle (r), the position of the fragments that display the bookbinding are correctly predicted (the first classes are at 75% and 77% respectively). The fragments are placed correctly in the horizontal axis, but the right and left upper fragment are unluckily swapped (their scores for the various top positions classes are around 30%).
- Puzzles (s) and (t) illustrates that texts and ornaments are not distinguished by the neural network. When the fragments are well placed, it is because of its white space.

In summary, text reassembly primarily uses borders, margins, and frames. They often identify the fragments being part of the same column (and, more rarely, the fragments from the same row).

7.3.4 *Reassembly from patchworks*

In archaeology, the fragments are photographed independently. The puzzles to solve are made of several tiles coming from different cameras and shooting angles. Merged into one 2D-puzzle, they show slight variations of colors and proportions. We produce 30 patchwork puzzle made from different photographs of some MET paintings, and we solve them (Figure 7.10). We observed a decrease of 1% on the number of well-placed fragments, compared to the corresponding MET images. It means that our neural network is not biased by overfitting on the camera parameters.

Figure 7.10: Reassemblies from patchwork images. The first row shows the patchwork images from which the fragments were extracted. The second row displays the reassemblies for the patchwork fragments. The third row contains the reassemblies of the MET image (without patchwork). The red outline shows the fragments that are misplaced.

Part III

ITERATIVE SOLVING WITH DEEP REINFORCEMENT LEARNING

8

On AlphaZero

◂ Chapter 7 Chapter 9 ▸

Synopsis This chapter presents Monte-Carlo Tree Search (MCTS) coupled with reinforcement learning §8.2 and deep reinforcement learning §8.3.

8.1 Introduction

In Chapter 5, we presented methods for puzzle-solving with deep learning. We introduced three paradigms: pairwise comparison, global comparison, and permutation, and we described their strengths and weaknesses. Briefly, pairwise comparisons obtain better scores than "one-shot" permutation methods, which are limited in the number of classes (Table 7.5). Repeated permutation methods like Wei et al.'s [WXR+19][1] combine the best of both worlds, but are much slower. Last, global comparison methods are very similar to human's puzzle-solving techniques. They consist of placing the fragments iteratively on a grid where the previous fragments are already placed. However, to our knowledge, there are no publications on this subject. We make the hypothesis that they are very efficient, and we develop a model to confirm or invalidate this assumption in Chapter 9.

[1] [WXR+19] has been introduced in §5.4.

Our preliminary experiments with a convolutional neural network were unsuccessful because the global comparison task requires forecasting to make the right decision. Rather than opting for a recursive model such as RNN [RM87] or LSTM [HS97], we cast this task as a planning problem, and thus apply a model-based reinforcement learning[2] framework.

[2] We introduced reinforcement learning in Appendix A.

One of the most famous model-based algorithms is AlphaZero [SHS+17]. It combines neural networks with Monte-Carlo Tree Search (MCTS) algorithms and brilliantly defeated human players on complex board games. In this section, we present AlphaZero, Monte-Carlo Tree Search, and the research on those topics.

8.2 Monte Carlo Tree Search

8.2.1 *MCTS and the game of go*

The game of Go has long been a challenge for artificial intelligence because of its 10^{170} legal configurations. It is a two-player, perfect-information, deterministic board game with two basic rules. At the

end of the game, the score r is deduced from the state s of the board. It can be a tie ($r = 0$), a win for the first player ($r = +1$) or a loss for the first player ($r = -1$).

Since 2006, the programming of Go has made significant progress, in particular thanks to the MCTS method. It is a heuristic search algorithm in a tree, like Dijkstra's algorithm[3]. It was introduced by [Cou06] for Go game [GW06] and proven to be guaranteed to converge [KS06]. Shortly after, algorithms using both MCTS and reinforcement learning emerged:

[3] Dijkstra's algorithm has been introduced in §6.5

Silver[4], the future lead researcher on AlphaGo, worked on MCTS and reinforcement learning. He proposed a 9×9 Go solver.

[4] [Sil09] D. Silver, Reinforcement learning and simulation-based search.

Browne et al.[5], followed by Vodopivec et al.[6] a few years later, propose a survey of the relations between MCTS and reinforcement learning, and some enhancements on the method.

[5] [BPW+12] C.B. Browne et al., A survey of Monte Carlo tree search methods.

[6] [VSS17] T. Vodopivec, S. Samothrakis, and B. Šter, On Monte Carlo Tree Search and Reinforcement Learning.

MCTS, coupled with reinforcement learning, is now widely adopted in various fields.

8.2.2 *Advances in MCTS*

To depict the trending lines of research on improving MCTS, we select a few papers among recent work. Efroni et al. [EDSM19] show that some standard tree search implementations are not guaranteed to converge; they proposed an enhancement of MCTS that uses the optimal tree path values. Kartal et al. [KHLT19] focus on sample inefficiency of MCTS. Takada et al. [TIY19] propose an algorithm where the policy function is trained directly from the game results without the search probabilities. Wu et al. [WWL+19] filters the low-quality moves, which echoes our work on Deepzzle branch-cut §6.5.

8.2.3 *Single-player MCTS*

Puzzle-solving is a single-player game, but MCTS is designed for two agents: it is legitimate to question whether MCTS can be applied in this case.

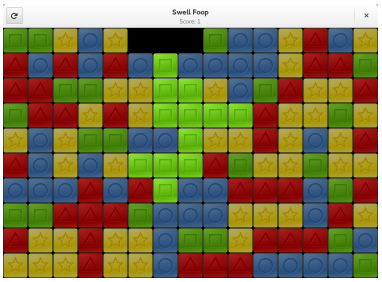


Figure 8.1: Example of SameGame board. © Swell-Foop.

As Seify pointed out in his master thesis[7], deterministic single-player games are equivalent to a two-player game where the second player always pass. He lists some differences that must be addressed when designing a single-player MCTS. Taking the example of SameGame[8] (Figure 8.1), he argues single-player games reward are not bounded (or even non-losable), while MCTS is adapted for rewards among $\{-1, 0, 1\}$. He also explains why standard MCTS parallelization is unsuitable for one-player games. Consequently, some authors optimized MCTS to address single-agent problems:

[7] [Sei20] A. Seify, Single-agent optimization with Monte-Carlo Tree Search and deep reinforcement learning.

[8] SameGame is a tile-matching puzzle game where the player tries to remove every tile.

Schadd et al.[9] introduced Single-Player MCTS (SP-MCTS), which improves among other things the performances of UCB[10]. They create a tree per move rather than a tree per game and change MCTS selection formula to compensate for the non-adverse aspect.

[9] [SWTU12] M.P.D. Schadd et al., Single-Player Monte-Carlo Tree Search for SameGame.

[10] We present UCB it in Chapter 9.

Baier and Winands propose a recursive MCTS that solves single-players games such as Bubble Breaker, NMCTS algorithm [BW12]. It is based on NMCS [Caz09], which also inspired Rosin NRPA [Ros11b] that was able to solve Morpion Solitaire and construct crossword puzzles.

Orseau et al.[11] introduce two tree search algorithms for single-player games. The first one is adapted for "needle-in-a-haystack" problems, i.e., problems for which the number of correct solutions is very limited. It derivates from Levin's search [Lev73]. The second one is well-suited for problems where many paths lead to a goal.

[11] [OLLW18] L. Orseau et al., Single-agent policy tree search with guarantees.

Seify and Buro[12] proposes a variant of MCTS which is adapted with games with unbounded rewards. Like Schadd [SWTU12], they propose a variant of the MCTS selection formula. Their algorithm also parallelizes well.

[12] [SB20] A. Seify and M. Buro, Single-Agent Optimization Through Policy Iteration Using Monte-Carlo TreeSearch.

8.3 DEEP REINFORCEMENT LEARNING AND MCTS

8.3.1 *Two-player games*

Figure 8.2: The match between grandmaster Lee Sedol, right, and AlphaGo. © Lee Jin-man/AP.

Model-based deep reinforcement learning has boomed since 2016. In most research, the model is learned [RWR+17, NKFL18, BHT+18], but some authors propose methods where the model, usually MCTS, is given to the algorithm:

Silver et al. introduced AlphaGo[13] in 2015. It managed to beat a professional human player for the first time (Figure 8.2. AlphaGo combines MCTS with neural networks, which predict a policy and a value estimate of the state, i.e., an estimate of the future reward.

[13] [SHM+16] D. Silver et al., Mastering the game of Go with deep neural networks and tree search.

Silver et al. improved their algorithm and named it AlphaGo Zero [SSS+17]. They were able to remove prior knowledge of the rules and data from human games, so their algorithm explored original ideas and ultimately beat AlphaGo. Moreover, AlphaGo Zero uses a more efficient search algorithm and merges the redundant part of AlphaGo's neural networks into a two-headed network.

AlphaZero[14] follows AlphaGo Zero and tackles many two-player turn-based board games such as chess and shogi. The main difference between the two versions is that AlphaZero's neural network is updated continually.

[14] [SHS+17] D. Silver et al., Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

Anthony et al.[15] developed ExIt algorithm, which is a similar alternative to AlphaZero. They start with a policy-only neural network, which they replace with a two-headed network when the data generated through games is of sufficient quality.

[15] [ATB17] T. Anthony, Z. Tian, and D. Barber, Thinking fast and slow with deep learning and tree search.

MuZero[16] is the last version of AlphaGo. It surpasses AlphaZero's performance on Go and atari games and matches it on the easiest games like chess and shogi. Briefly, MuZero is compatible with

[16] [SAH+19] J. Schrittwieser et al., Mastering atari, go, chess and shogi by planning with a learned model.

single-agent games, has not a perfect knowledge of the ruleset, and separates the representation of the current step from its dynamics and the predictions.

8.3.2 *Single-player games*

Applying AlphaZero or ExIt to one-player games with sparse-reward environments such as jigsaw puzzle or Rubik's cube is challenging: a randomly initialized policy will be unlikely to encounter the only rewarding state. In the worst case, the state's value estimate is biased or divergent, and the policy will not converge to the optimal policy. We present some single-player games solvers that use deep reinforcement learning and tree search:

Arfaee et al.[17] proposed in 2011 to combine a neural network with the tree search algorithm IDA*[18].

Laterre et al.[19] proposed R2, an algorithm for single-player games. It addresses the issue of unbounded reward with a relative performance metric.

McAleer et al.[20] proposed DeepCube, a solver for the Rubik's cube. They pre-train a two-headed neural network and call this procedure ADI. After the network is trained, it is combined with MCTS to effectively solve the Rubik's cube[21].

Agostinelli et al.[22] continued the work of McAleer et al. and proposed DeepCubeA. In brief, they replaced MCTS with a weighted A* search. Their algorithm solves Rubik's cube, but also 8-puzzle (also known as gem puzzle and mystic square) (Figure 8.3).

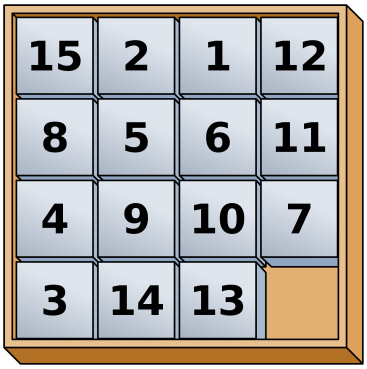


Figure 8.3: Example of 15-puzzle.

[17] [AZH11] S.J. Arfaee, S. Zilles and R.C. Holte, Learning heuristic functions for large state spaces.

[18] IDA* is a depth-limited version of depth-first search algorithms. Read more about it on Wikipedia

[19] [LFJ+18] A. Laterre, Y. Fu, M.K. Jabril, et al., Ranked reward: enabling self-play reinforcement learning for combinatorial optimization.

[20] [MASB18] S. McAleer, F. Agostinelli, A. Shmakov and P. Baldi, Solving the Rubik's cube with Approximate Policy Iteration.

[21] Rubik's cube is close to our problem because there are only one correct solution and many ways to reach it; it differs because it knows the solution

[22] [AMSB19] F. Agostinelli, S. McAleer, A. Shmakov and P. Baldi, Solving the Rubik's cube with deep reinforcement learning and search.

Artwork 5: *Old Plum*, Kano Sansetsu, 1646, from the MET Open Collections.

9

Alphazzle

◂ Chapter 8 Chapter 10 ▸

Synopsis This chapter walks through Alphazzle, our jigsaw puzzle solver relying on a global comparison. We present the interaction between MCTS and the neural networks in §9.2, and we examine each component in more detail in §9.3 and §9.4.

9.1 Prologue

In this chapter, we present Alphazzle, a method to solve jigsaw puzzles (Figure 9.1) with a global comparison. Its design meets two purposes. First, to compensate for the weaknesses of Deepzzle: the comparison to the central fragment only and the puzzle size limited to 3×3. Second, we build on AlphaZero because it is widely known and serves as an excellent baseline to prove the interest of global comparison.

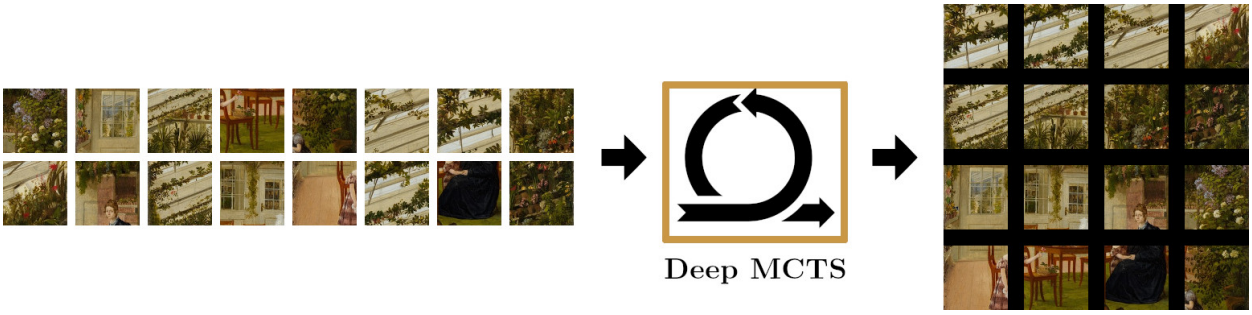


Figure 9.1: Example of a 4×4 jigsaw puzzle with iterative solving.

Among our contributions, we introduce a new deep reinforcement learning-based method to reassemble numerous fragments. The major challenge is that the ground-truth reward is not available to MCTS. We show how to estimate it from the visual input with neural networks. This constraint is induced by the puzzle-solving task and dramatically adds to the task complexity (and interest!). We perform an in-deep ablation study that shows the importance of MCTS and the neural networks working together, and we solve up to 25-fragments puzzles, which significantly outperforms state of the art. We achieve excellent results and get exciting insights into the combination of search algorithms and visual feature learning.

Sources The work presented in this chapter is under review:

- Solving Jigsaw Puzzle with Deep Monte-Carlo Tree Search.

9.2 OVERVIEW

In this section, we focus on the interaction between MCTS and the neural network. We start by presenting two-player games with deep reinforcement learning, which allows us to detail the notations, and AlphaZero, which gives an overview of the interactions between the tree search and the deep learning. Then, we introduce the rules of the jigsaw puzzle game. These two pieces of knowledge allow us to explain how puzzle-solver works by comparing it with AlphaZero.

9.2.1 *Simplified framework for two-player games with deep reinforcement learning*

Two agents (g_1, g_2) play a turn-based game. The game is defined by a set of hard-coded rules, which includes the initial board state s_0, the available actions given a current state s_t, the end game criteria, and the scoring function $r(s_{t\_max}) \in \{-1, 0, 1\}$ (i.e., the reward), that is computed at the end of the game t_max. If the score is null, there is a tie; if $r(s_{t\_max}) = +1$, agent g_1 won the game, and vice versa. Note that t_max varies between two games.

At each turn t, one of the agents chooses the available action a_t that is presumed to minimize the opponent's final gain: g_1 aims to maximize $r(s_{t\_max})$ and g_2 wants to minimize it. As one agent does not control the other's actions, it has to plan what can happen next and predict the reward.

The next action choice is based on policy $\pi_\theta(a_t|s_t)$, where θ are the parameters of the neural network P that returns the policy. As some games' duration can be very long, getting samples of games and associated rewards is not feasible in a reasonable amount of time. Therefore, having an estimator of the value function is a must. The value function is the sum of expected rewards values, given a current state s_t: $v(s_t) = \mathbb{E}(r(s_{t\_max})|s_t)$. Consequently, there is a neural network V in charge of learning $v(s_t)$, and guiding the optimization of θ. Such an architecture is reminiscent of actor-critic models. In practice, P and V share many layers and parameters, and the earlier AlphaGo has even 3 distinct policy networks for the different stages of the game (opening, middle-game, and endgame).

9.2.2 *AlphaZero algorithm*

AlphaZero adds to the framework presented above a planning algorithm, MCTS, which explores many compelling actions' further effects. The planning helps to foresee the changes in the environment induced by the other agent's actions. Instead of selecting the action a_t that maximizes $v(s_t)$ according to V and P, the agent performs simulations of what could happen according to its action. Therefore, it can select its action based on the value of the most promising explored state.

MCTS returns the policy $\pi_{MCTS}(a_t|s_t)$. During inference, the agent

selects the best action; then, the second agent applies MCTS from the state s_{t+1}. During the learning phase, an exploration trade off is provided to the agent, and the neural networks are engaged in reinforcement learning, playing until the accuracy is acceptable.

9.2.3 *Interaction between MCTS and the neural network*

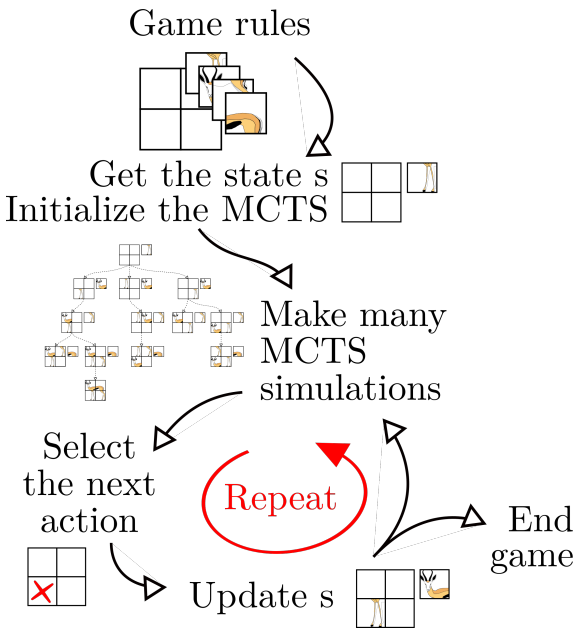


Figure 9.2: Alphazzle outline.

Our core algorithm (Figure 9.2) reproduces AlphaZero. After initializing the state, MCTS explores many partial reassemblies and returns a policy $\pi_{MCTS}(a_t|s_t)$. Then, the agent chooses (one of) the most promising action. The state is updated, i.e., the fragment is added to the current reassembly. We start again from MCTS exploration, except that the root node is s_{t+1}. When one of the endgame criteria is validated, the game ends.

9.2.4 *Jigsaw puzzle rules and formalization*

A single agent plays the game.

States The current (board) state s_t is described by the ordered set of fragments to place and the current partial reassembly, obtained from the already placed fragments. The observable state only contains the next fragment to place $x_{f,t}$ and the partial reassembly $x_{r,t}$.

Actions The available actions A_t are those that assign the next fragment to any empty position. When the agent performs action $a_t \in A_t$, the fragment is added to the partial reassembly and the state is updated to s_{t+1}. Note that the environment is deterministic, so we know exactly what s_{t+1} is from s_t and a_t.

It is equivalent to having an unordered set of fragments in s_t and letting MCTS selects which fragments to place where. The width of the tree increases strongly, but the ratio of correct paths is unchanged. We choose to fix the fragments' order to reduce the tree size. The downside is that, as some orders as easier to solve than others, finding a correct reassembly may be more difficult with our setup. To compensate for this, we solve several times the same puzzle with different fragments' orders.

Initialization At first, $x_{r,t=0}$ is the zero matrix. Its size is fixed and depend on the number of fragments, on their size, and on the gap between fragments size. To differentiate black fragments from empty locations, we also initialize to -1 a dictionary d_t which matches the positions $j \in [0\,..\,p]$ in the reassembly to the indexes of the fragments $i \in [0\,..\,f]$, where f is the number of lateral fragments and p the number of position.

Endgame The game ends when all the positions are filled up:

$$\forall j \in [0\,..\,p], d_{t_{max}}[j] \neq -1,$$

or as soon as all the fragments are placed:

$$A_{t_{max}} = \emptyset, \text{ i.e., } x_{f,t_{max}} = None.$$

Consequently, the depth of the tree spanning the action space is bounded by $\min(p, f)$, which is not the case for AlphaZero's games, as games of variable length can be generated.

Reward We have several choices for the reward. As mentioned in §2.3, we can use three metrics to evaluate how correct the game is: the percentage of *correct neighbors*, the percentage of *well-placed fragments* and the *solved puzzle*. All of them are bounded by one, so we do not face unbounded reward described in §8.2.3, like in SameGame. Therefore, most of the suggested MCTS single-player optimizations in §8.2 are no longer required.

We opt for the binary *solved puzzle* reward: we expect this reward encourages MCTS to focus on the solution and discards all wrong reassemblies, even those with a high percentage of correct neighbors. The downside is the wrong reassemblies are considered as equivalent, i.e., they are not ordered. Therefore, if we want MCTS to return the three best reassemblies, we will get only the best one and return two other reassemblies, which probably will not be the second and third best.

We endow Alphazzle's MCTS with two reward modes:

- The first one is based on ground-truth: if the final reassembly dictionary equals the solution dictionary, then $r(s_{t_{max}}) = 1$, else $r(s_{t_{max}}) = 0$.
- The second one is based on an automatic assessment of the realism of the reassembly. In the real world, archaeological puzzles do not

come with their solution, so experts must evaluate if the reassembly is correct. An alternative would be to broke artifacts: in this way, we would know the ground-truth solution and could evaluate our reassembly quality. Unsurprisingly, archaeologists did not choose this option. Compared to AlphaZero and other deep reinforcement learning algorithms, this is new. We describe this reward mode as "the (ground-truth) reward is not available to MCTS."

To assess whether a puzzle is correctly reassembled (second mode), we need an evaluator who has learned how to classify the correct reassembly. Fortunately, this is the goal of V, as we will discuss in §9.4. Therefore, we use the neural network V to predict the value function and compute the reward, which brings to mind inverse reinforcement learning [AN04].

9.3 MONTE CARLO TREE SEARCH

9.3.1 *Two-player MCTS algorithm*

As we already know, MCTS visits N_{visits} nodes from the current state s_t, and returns a_t, the most promising action for the step t. The current player applies it. Then, MCTS explores N_{visits} nodes, starting from $s_t + 1$ and returns a_{t+1}, and so on until the end of the game.

The possible states are represented with a tree, whose branches are the actions. At the beginning of MCTS, there is only one node, s_t.

Each node is associated with an agent, the number of visits, and the number of simulations from that node that led to the agent's victory. Each row of the tree represents a turn, and so is associated with a different agent. The number of visits to a node is the sum of the visits of its children. The number of wins of a node is the number of visits minus the sum of its children's wins.

After N_{visits} nodes visited, MCTS returns the policy $\pi_{MCTS}(a_t|s_t)$.

MCTS applies the four following steps N_{visits} times:

1. *Selection* MCTS selects a node with potential children (or an endgame). The selection is based on a strategy $U(a_t|s_t)$, derived from $\pi_\theta(a_t|s_t)$, and introduces a trade-off between exploitation and exploration.

2. *Expansion* MCTS initializes the child(ren) node(s) of the selected node and selects one of them.

3. *Simulation* MCTS executes a random game from the child's state, to the end game. It obtains a reward.

4. *Backpropagation* MCTS backpropagates the (inverse) reward and updates all the upstream nodes expectation.

These steps are shown in Figure 9.3, which illustrates the process with values.

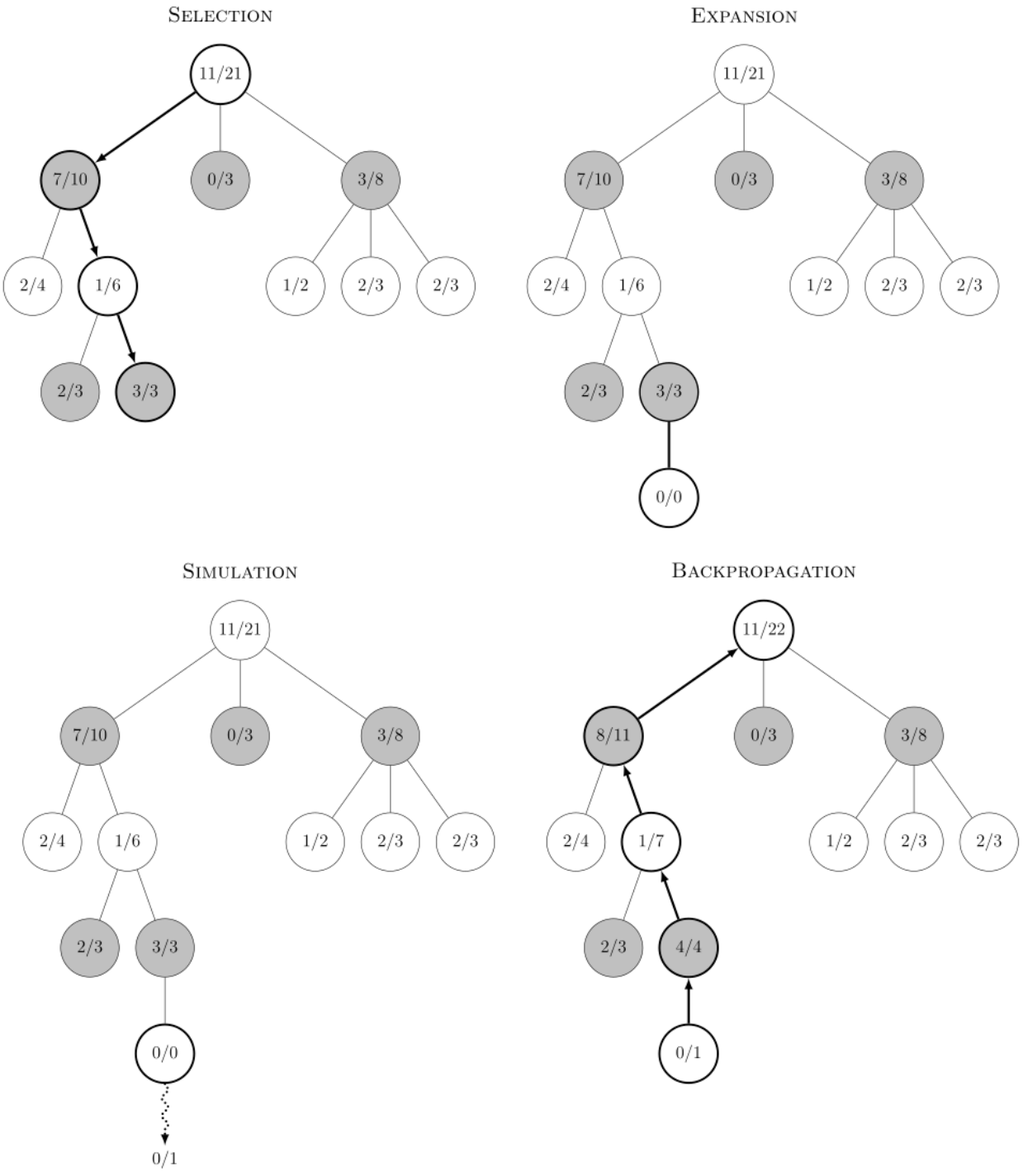


Figure 9.3: Steps of Monte Carlo tree search. In this example, the reward obtained after the simulation phase is 0. Each line corresponds to an agent. Each node is associated with two numbers: the first one indicates the sum of winning simulations for the current agent, and the second, the total visits.

9.3.2 *Selection*

The Selection phase enables picking a node among the leaves[1] according to a vector $U(a|s_t)$, named from Upper Confidence Bounds (UCB). It assigns values to each available action from the state s_t. We apply the best action according to $U(a|s_t)$ recursively until a leaf state is reached. Note if the leaf has children already, the Selection algorithm can either stop or selects a child.

[1] By leaves, we mean the nodes that have at least one unvisited child.

In 2006, Kocsis and Szepesvári proposed Upper Confidence Bounds for Trees (UCT) [KS06], a selection strategy derived from UCB [ACBF02]. It states:

$$\forall a \in A_t, U(a|s_t) = Q(a|s_t) + C \cdot \sqrt{\frac{\log N(s_t)}{N(a|s_t)}}, \tag{9.1}$$

where A_t is the set of available actions at step t, $Q(a|s_t)$ is the expected value of the available actions (Equation in §9.3.5), C is the exploration trade-off constant, $N(s_t)$ is the number of visits to the node associated with s_t, $N(a|s_t)$ is the number of times the available actions have been taken from the state s_t. Note that $\forall a \in A_t, N(a|s_t) = N(s_{t+1|a})$.

This strategy U has been adapted to single-player games by Schadd et al. [SWTU12]:

$$\forall a \in A_t, U_{SP}(a|s_t) = U(a|s_t) + W \cdot Q_{max}(a|s_t) + \sigma(a_t|s_t).$$

They made two modifications on 9.1:

- $W \cdot Q_{max}(a|s_t)$ is a fraction of the maximum value obtainable from the action a applied from s_t. This term indicates that it is relevant to focus not only on $Q(a|s_t)$, the average value that can be obtained from a, but also on the maximum value that can be derived from it. This is possible because there are no opponents[2]. Schadd et al. set $W = 0.02$. Jacobsen et al. [JGT14] recommend using $(1-\lambda) \cdot Q(a|s_t) + \lambda \cdot Q_{max}(a|s_t)$ rather than $Q(a|s_t) + W \cdot Q_{max}(a|s_t)$.
- $\sigma(a_t|s_t)$ is the standard deviation estimate. This term is pertinent in case of game with unbounded rewards.

[2] In two-player games, if the first player chooses the action that has a bad average but a maximum value, the second player will steer the game so that the maximum value is not reached.

AlphaGo [SHM+16] introduced Predictor + Upper Confidence Bound for Tree (PUCT), derived from PUCB [Ros11a]:

$$\forall a \in A_t, U(a|s_t) = Q(a|s_t) + C \cdot \pi_\theta(a|s_t) \cdot \frac{\sqrt{N(s_t)}}{1 + N(a|s_t)}, \tag{9.2}$$

where $\pi_\theta(a|s_t)$ is the policy returned by the neural network P that predicts the actions.

In Alphazzle, we mostly use Equation 9.2. We also implemented this equation in which we replaced $Q(a|s_t)$ by Jacobsen et al.'s term.

9.3.3 *Expansion*

The Expansion phase occurs after the Selection and enables appending one or several nodes to the tree. An example of an Expansion strategy is to select an unexplored node randomly.

In AlphaZero, one node is expanded at each iteration. The expanded node is obtained from the best $U(a|s_t)$. Indeed, in PUCT, the predictors allows $U(a|s_t)$ to select an unexplored node. Therefore it is used for both the Selection and Expansion phases.

In Alphazzle, we use AlphaZero Expansion.

9.3.4 *Simulation*

The Simulation phase aims to find a possible value obtained from the expanded node. The objective is to make (more) accurate this node's value and its predecessors during the Backpropagation phase. An example of a Simulation strategy is to select actions randomly until an endgame is reached. In most cases, however, a handmade policy guides the Simulation. For example, Schadd et al. [SWTU12] propose two policies for SameGame, which promotes the creation of large groups of color.

In AlphaZero, Silver et al. replace the Simulation phase with a neural network V, which predicts the expected value from the expanded node. When their MCTS reach an endgame node, the ground-truth reward is returned.

In Alphazzle, we use AlphaZero Simulation, except that the endgame ground-truth reward may be replaced by the predicted reward[3], depending on our experimental settings.

[3] We introduced the predicted reward in §9.2.4.

9.3.5 *Backpropagation*

The Backpropagation phase updates $Q(a|s_t)$, $N(a|s_t)$ and $N(s_t)$ of each node visited.

Given $v(s_t)$ the value of the leaf node, we initialize $Q(a|s_t) = v(s_t)$, $N(a|s_t) = 1$ at the first visit of the node. At each next visit, we perform the update:

$$
\begin{aligned}
Q(a|s_t) &\leftarrow \frac{N(a|s_t) \cdot Q(a|s_t) + v(s_t)}{N(a|s_t) + 1}, \\
N(a|s_t) &\leftarrow N(a|s_t) + 1, \\
N(s_t) &\leftarrow N(s_t) + 1.
\end{aligned}
$$

9.3.6 *Solving a puzzle with our MCTS*

In Algorithm 6, we present the algorithm that initializes MCTS, updates it and returns the policy $\pi_{MCTS}(a_t|s_t)$. We detail Update_mcts below. The other functions are:

- Init_mcts creates an object *tree* with a single node which corresponds to s_t, and initializes dictionaries indexed by s_t that contains the node-related values: $Q(a|s_t)$, $N(a|s_t)$, $N(s_t)$, $P(a|s_t)$, and $scores(s_t)$.

- Zero_if_not_max is called during the inference phase (i.e., when

training is False) and sets all values of *actions_scores* to zero, except its maximum.

- DIVIDE_BY_SUM takes a list of values and returns the list of values divided by the sum of the values.

Algorithm 6: MCTS launcher.

```
1: procedure GET_Π_MCTS(s_t)
2:     tree ← INIT_MCTS(s_t)
3:     for N_visits do
4:         tree, _ ← UPDATE_MCTS(tree, s_t)
5:     end for
6:     next_actions_scores ← tree.scores[s_t]
7:     if training is False then
8:         next_actions_scores ← ZERO_IF_NOT_MAX(next_actions_scores)
9:     end if
10:    policy ← DIVIDE_BY_SUM(next_actions_scores)
11:    return policy
12: end procedure
```

We propose two ways of computing $tree.scores$, the list of scores for each action available for s_t:

- We use $tree.scores(s_t) = Q(a|s_t)$, because $Q(a|s_t)$ is the list of expected values of actions a. Therefore, the optimal action should be the the one with the highest Q-value.
- As in AlphaZero, we use $N(a|s_t)$, because the most visited nodes[4] are more trusted than a newly visited node with higher Q-value.

[4] The number of visits depends on $Q(a|s_t)$.

In Algorithm 7, we explain UPDATE_MCTS, in the case where we use the predicted reward rather than the ground-truth reward.

Algorithm 7: MCTS updater.

```
1: procedure UPDATE_MCTS(tree, s_t)
2:     if IS_ENDGAMES(s_t) then
3:         return tree, V.PREDICT(s_t)
4:     end if
5:     if IS_NOT_VISITED(s_t) then
6:         tree.P[s_t] ←P.PREDICT(s_t)
7:         return tree, V.PREDICT(s_t)
8:     end if
9:     a ← max_a GET_U(s_t, tree)
10:    s_{t+1} ← UPDATE_STATE(s_t, a)
11:    tree, v ← UPDATE_MCTS(tree, s_{t+1})
12:    tree ← BACKPROPAGATION(tree, v)
13:    return tree, v
14: end procedure
```

We detail the functions introduced in Algorithm 7:

- IS_ENDGAMES returns True if s_t validates an endgame criteria.

- Is_not_visited returns True if s_t is not in the tree.
- V.predict and P.predict returns the predictions of the neural networks V and P.
- Get_U computes $U(a|s_t)$.
- Update_state executes the actions a from state s_t and returns the obtained state s_{t+1}.
- Backpropagation applies the backpropagation equations and updates the tree values.

In Figure 9.4, we shows our MCTS applied to a 2×2 jigsaw puzzle. We display the value of $v(s_t)$, the expectation of finding the solution from each state s_t.

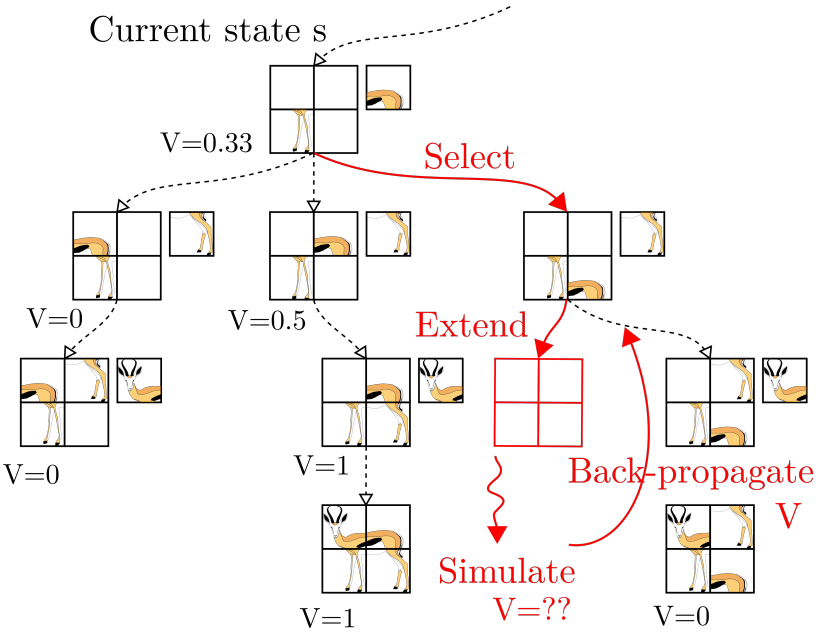


Figure 9.4: Example of MCTS simulations applied to puzzles. In this example, the states are clearly shown, as well as the state of the value function, which is the expectation to find the correct reassembly from the current node.

9.4 Deep Reinforcement Learning

Like McAleer et al. suggested in [MASB18], we pre-train our neural networks on handcrafted tasks. We expect such pre-training enables them to be more accurate. After the pre-training, we integrate them into our MCTS algorithm. Then, we may fine-tune them after MCTS solved several puzzles.

Note that another difference with AlphaZero is that our P and V use visual inputs and thus have to extract meaningful information from these noisy signals. In contrast, AlphaZero directly uses the board state, which is a clean semantic input. For that reason, we use two image datasets: one for training P and V and one for evaluating our algorithm.

9.4.1 *Pre-training P*

The neural networks P predicts the best actions from a fragment's image and a partial reassembly image. We generate inputs from our puzzle dataset: we make correct partial reassemblies and select fragments to place among the remaining fragments. Note that P is not trained on wrong partial reassemblies, because there may be no correct answer for the fragment to place (i.e., its position is already taken).

The neural network P is trained using categorical cross-entropy to predict the fragment position, outputting an estimate $\vec{p}_s$ of the policy.

Architecture We use two architectures for the features extractor part of P: a WideResNet (WRN) [ZK16] initialized with random weights and a ResNet [HZRS16] pre-trained on ImageNet. Most of our experiments ran on WRN, because it was what we implemented first[5].

Each input goes through the same feature extractor, thanks to shared weights. Then, each goes to a different multi-layer perceptron made of 6 (2 for ResNet) successive fully-connected layers of size 512, alternating with ReLU functions. The two outputs are concatenated and given to a shallower perceptron that predicts the fragment position (2 fully-connected layers, a ReLU between them, and a softmax at the end). The classes correspond to the puzzle positions, and P is trained by sampling random partial reassemblies.

[5] We implemented WRN to be able to evaluate our trained neural network on few-shot tasks and compare it easily with other WRN trained on different tasks.

9.4.2 *Pre-training V*

The neural networks V predicts the expected reward value that can be reached from the state s_t. Therefore, its input is a reassembly, either partial or complete. When the reassembly is complete, or when only one fragment is remaining, V predicts the reward: 0 if there is at least a mistake[6] and 1 otherwise. When the reassembly is partial, V is trained to predict 0 if there is at least a mistake. If the partial reassembly is correct, it can lead to wrong and correct reassemblies; therefore we train V to predict:

[6] If $p = f$, the minimal number of mistake is 2.

$$v(s_t) = 0.5 + 0.5 \cdot \frac{i}{f-1}, \tag{9.3}$$

where f is the number of fragments, and i is the number of well-placed fragments in state s_t. We choose this value over the expectation because it indicates the confidence in the prediction: an empty puzzle predicted value is 0.5, because we have no information.

The neural network V is trained using MSE loss. Its architecture borrows the WRN (or ResNet) from P, followed by an MLP.

9.4.3 *MCTS-based fine-tuning*

Finally, we introduce the optional fine-tuning of P and V that could occur while solving puzzles.

This mining of training examples is akin to active learning, where the learning focuses on more important examples. Indeed, during the first training, reassemblies are sampled uniformly, while it is not the case of the nodes explored by MCTS. Thus, we suggest that fine-tuning the networks on the nodes that are likely to be visited. This process differs from AlphaZero: in a two-player game, there is always a set of actions that led to victory and thus can be used to reinforce the PV network, especially because V has access to the ground truth at terminal nodes.

We use the states obtained from all the choices made after the policy $\pi_{MCTS}(a_t|s_t)$. In AlphaZero, the neural networks learn to reproduce actions that lead to the victory of an agent. In our case, we cannot deploy such learning from the opponent; therefore, we learn the ground-truth for P (even if the position is already occupied) and the value from Equation 9.3 for V.

9.5 EXPERIMENTS

9.5.1 *Training procedure*

We program the neural networks with PyTorch library. Table 9.1 shows the standard parameters for our experiments.

Feature extraction	WRN
Optimizer	Adam
Learning rate	0.001
Fragment size	40
Fragment per size	3
Space size	4
Selection	PUCT
Reward	Predicted
Action choice	$N(a\|s_t)$

Table 9.1: Summary of the experiments parameters.

9.5.2 *Reassembly metrics*

We introduced the metrics in §2.3. For Alphazzle, we use the *solved puzzles*, the *well-placed fragments* and the *correct neighbors* metrics.

9.5.3 *Dataset*

We train our neural network on MET (10,000 training images and 2,000 validation images). At each epoch, we use different crops within the images. We consider a single pair of input per image for

the pre-training, and all chosen pairs per image for the fine-tuning. We normalize the values between -1 and 1 for WRN and accordingly to PyTorch documentation for ResNet.

Artwork 6: *The Dance Class*, Edgar Degas, 1874, from the MET Open Collections.

10

Alphazzle results

◂ Chapter 9 Chapter 11 ▸

Synopsis This chapter presents the results obtained from training P and V §10.1, and optimizing MCTS §10.2. Last, we discuss the results on the reassembly §10.3 and introduce some optimization §10.4.

10.1 Pre-training results

This section presents our pre-training results for P and V, for ResNet and WideResNet. Then, we compare the settings, i.e., the number of fragments, their size, the space between them, and the impact of some fragments placed on the partial reassemblies.

10.1.1 *Architectures comparison*

Table 10.3 shows the performance of our two architectures:

Network	P (%)	V (%)
WideResNet	69.91	88.46
ResNet	50.71	78.73

Table 10.1: Validation accuracy scores — Comparison between ResNet and WideResNet, on 100 epochs.

We observe that ResNet displays lower performance than WideResNet; therefore, we rely on WideResNet for our next experiments.

10.1.2 *Settings comparison*

Table 10.2[1] shows the impact of settings on neural networks:

[1] Table 10.2 is an excerpt of Table D.1.

Fragment per side	Space size (px)	Hints	P (%)	V (%)
3	0	0	69.56	90.07
3	4	0	69.91	88.46
3	10	0	67.39	87.15
3	20	0	61.34	85.79
4	4	0	37.65	92.64
4	4	8	52.02	99.09
5	4	0	19.15	94.25

Table 10.2: Validation accuracy scores — Comparison between the settings.

First of all, we note it is easier for V to scout mistakes in the reassembly than for P to predict the action.

Fragment per side Increasing the number of fragments in a puzzle does not profoundly impact the performance of V but leads to a drop in performance for the network P, which is not compensated when training with already placed fragments. In Table D.1, 6×6 puzzles reach a validation accuracy of 3.43% for P and of 64.47% for V.

Space size Increasing the size of the space between the fragments makes the puzzles slightly more complicated, by a few percent for P and V. It means our neural networks successfully learn to solve puzzles without relying on continuities.

Fragment size We ran very few experiments on the fragment size. Note that on 96×96 fragments, the space size is half the fragment size. Therefore, we should compare the results to the 40×40 fragments spaced by 20 pixels. According to Table D.1, bigger fragments lead to -5% accuracy for P and V, probably because our architecture is not well suited for larger fragments, or at least not well tuned for them.

Placed fragments The placed fragments, or "hints", are the minimal number of placed fragments in the reassembly when training or evaluating the networks. For P, they are the well-placed fragments; for V, correct reassemblies alternates with reassemblies that can lead to a wrong reassembly. The impact of hints on results allows studying how our neural networks behave on easier tasks. For example, when we give 8 hints to a neural network that solves the 4×4 puzzle, we theoretically approximate the 3×3 puzzle difficulty, but the performance is lower on P.

We also consider a central hint to compare with Deepzzle. When there are 8 hints, it means that there is only one fragment to place in the case of 3×3 puzzles. The results show that the neural network P successfully learned to place the fragments in empty positions. However, it did not reach 100% because some background fragments are as black as empty spaces. Inspired by AlphaGo's data structure, we proposed to append a 4th channel to the images that allow differentiating empty spaces and black fragments. We did not see any visible improvement of the validation accuracy, but the computing time has become slightly longer.

We ran more experiments to analyze the impact on hints. We present the results in Table D.2. In this Table, contrarily to the previous one, the validation hints does not indicate the minimal number of hints, but their exact number.

Giving hints makes the neural networks perform better in solving easy puzzles (with as many or more hints). However, the validation accuracy we obtain dropped significantly on puzzles with fewer hints than during the training: we obtain 11.34% $\simeq$ 1/9 on empty reassembly if we train P to place only the last fragment. Note that the standard P network trained with no hint can correctly predict the position of the first fragment 46.97% of the time.

If we study the impact of hints on specific partial reassembly, for every amount of fragments placed, we see that performance on easy puzzles (i.e., when most fragments are placed) are equivalent for various minimal numbers of training hints. It means that using specialized neural networks[2] for the last steps brings no substantial gain. A neural network which has not been trained on difficult partial reassemblies performs worse than a neural network trained on all type of partial reassemblies. However, it will still be better than random on puzzles slightly more difficult than those on which it learned. Last but not least, V displays bad accuracies on the most challenging reassemblies. It is because it learns to express its uncertainty about the puzzles: it makes soft guesses over the class.

[2] As suggested in earlier versions of AlphaGo, with their different P networks for opening, middle-game, and endgame.

Accuracy of V on complete puzzles Table 10.3 shows the results obtained for complete reassemblies, i.e., the predicted reward for the endgame, under various errors distribution:

Well-placed fragments	D*	9	<7	7	6
V (%)	96.77	92.87	99.74	58.51	85.83

Table 10.3: Validation accuracy scores of V — Comparison on complete puzzles. D* is the usual distribution: half correct reassemblies, half with mistake.

On average, on our pre-training distribution, we obtain 96.77%. Half of this distribution are perfect reassemblies (i.e., 9 well-placed fragments), the other half are reassemblies with any number of mistakes (i.e., <7 well-placed fragments).

Interestingly, it is difficult for V to evaluate puzzles with only one inversion, but as soon as three fragments are inverted, the precision goes back up.

Figure 10.1 shows an example of a two-fragments inversion:

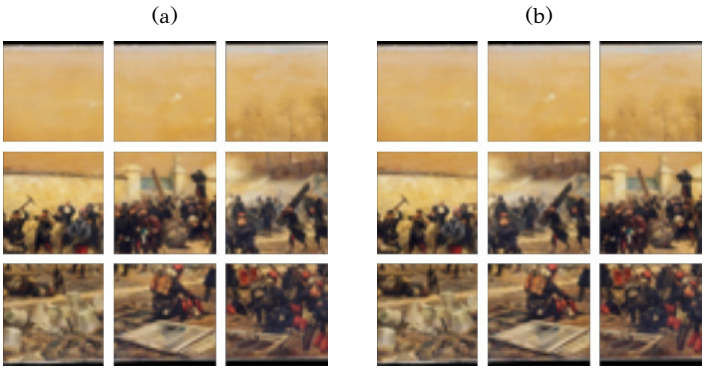


Figure 10.1: Comparison of a correct reassembly and a incorrect one with one inversion — Fig. (b) has two misplaced fragment, the central one and the right one.

Figure 10.1 illustrates the difficulty of the task: (b) may seem correct. Moreover, many images from MET have a plain background and interchangeable fragments, but our metric does not take it into account.

10.2 MCTS PERFORMANCE

In this section, we compare various settings, and we study the impact of neural networks, especially of the endgame reward, on MCTS.

10.2.1 *MCTS meta-parameter optimization*

We analyze the number of visits N_{visits}[3] to run before selecting the action, and the trade-off C between exploration and exploitation. We want to select the best configuration for the reassembly task.

Table 10.4[4] presents some results on meta-parameters:

[3] For 3×3 puzzles, the algorithms need at least 10^5 simulations to explore all the branches if no exploitation occurs.

[4] Table 10.4 is an excerpt of Table D.3.

N_{visits}	10	10^2	10^3	10^3	10^3	10^3	10^4	10^5	10^6
C	1	1	0.01	0.1	1	10	1	1	1
Fragment accuracy (%)	49.0	54.5	48.8	55.0	55.6	52.2	57.9	58.0	65.6
Puzzle accuracy (%)	12	15	10	14	15	13	17	22	30
Solving time (s/puzzle)	1	4	8	9	16	25	84	685	7200

Table 10.4: Reassembly scores — Comparison of MCTS meta-parameters.

We obtain the best score with a high number of simulations and $C = 1$. Running many simulations increases the computation time drastically, although results are always better with more simulations. Note that high exploration (when $C > 1$) also has a non-negligible computational cost as more new states have to be analyzed by the neural networks. For that reason, we use 10^3 simulations and $C = 1$ in the following experiments.

10.2.2 *Influence of P and V on MCTS*

We compare the endgame reward type, the behavior of MCTS deprived of P or V, and the behavior of P and V without MCTS.

PREDICTED REWARD VERSUS GROUND-TRUTH REWARD We analyze the results obtained for the two types of endgame reward in Table 10.5:

Reward	Fragment-wise (%)	Neighbor-wise (%)	Puzzle-wise (%)
Ground-truth	78.14	79.92	70.55
Predicted	55.63	58.93	14.55

Table 10.5: Reassembly scores — Comparison between endgame reward.

Scores drop by 20% for fragment-wise and neighbor-wise metrics, and by 55% for the reassemblies, which is a significant difference.

We assume that this is due to the accuracy issue of V on complete puzzles. On the one hand, with ground-truth reward, if MCTS finds a path with $r(a, s) = 1$, it is the correct reassembly, and so MCTS will select it. On the other hand, with predicted reward, MCTS may find paths with $r(a, s) = 1$, but that leads to wrong reassemblies.

This experience allows us to better grasp the difficulties related to

the use of a predicted reward.

Deactivation of P and V To measure the importance of P and V for MCTS, we compare the impact of deactivating them, i.e., replacing P by a unit vector or V by a constant. Table 10.6 displays the results:

P	✓	✓	✓	✓				
V (endgame)	✓	✓			✓	✓		
V (during game)	✓		✓		✓		✓	
Fragment accuracy (%)	55.63	53.64	49.00	45.01	54.17	14.47	51.47	11.23
Puzzle accuracy (%)	14.55	14.65	11.10	9.20	12.25	0.00	10.50	0.00

Table 10.6: Reassembly scores — Comparison of MCTS with and without P or V.

It appears that MCTS can cope with the absence of either P or V, but not the lack of both. We note that deactivating P has more impact than replacing V by 1 during the game.

If we keep P and V but stop predicting the reward from the middle-game (second column), the results are close to the baseline. However, if we remove P or V and the middle-game predictions, the results drop. Especially if we remove P, we are not able to reassembly any puzzle. On the contrary, if we only deactivate V for the endgame reward and predict 1, the neural network still picks pertinent reassemblies, although it considers them all equivalent.

Greedy neural networks Last, we compare our results with a baseline that uses a greedy exploitation by taking the argmax of P or V at each step, without MCTS. We show the results in Table 10.7:

Reward	Fragment-wise (%)	Puzzle-wise (%)
MCTS	55.63	14.55
Greedy P	42.0	6.0
Greedy V	44.0	8.6

Table 10.7: Reassembly scores — Comparison with and without MCTS.

When P solves a puzzle by itself, it is shown pairs of fragments and partial reassemblies, starting from the empty reassembly. Then, the fragments are placed according to P's predictions, updating the reassembly. Without MCTS, the results' quality drops, which shows the importance of exploring alternative reassemblies with MCTS.

Similarly, V evaluates all the partial reassemblies that can be obtained from the current state. Then, it selects the action leading to the best reassembly according to its predictions. It is interesting to see that V is a better action predictor than P alone, which already appeared in Table 10.6.

10.3 Reassembly results

This section presents our results for the standard experiment, as described in Table 9.1.

10.3.1 *Quantitative analysis*

Table 10.8, an excerpt of Table D.4, presents some reassembly scores:

Configuration					Reassembly scores		
Fragment size (px)	Fragment per side	Space size (px)	Hints P – V	Hints reassembly	Fragment-wise (%)	Puzzle-wise (%)	Reassemblies done in 24h
40	3	4	0-0	0	55.63	14.55	2000
40	3	4	0-0	1	60.94	20.65	2000
40	3	4	0-0	1 (central)	62.33	22.45	2000
40	3	10	0-0	0	48.59	7.15	2000
40	3	20	0-0	0	45.91	6.80	2000
40	4	4	0-0	0	29.08	0.00	407
40	5	4	0-0	0	15.68	0.00	203

Table 10.8: Reassembly scores with 1,000 simulations and $C = 1$.

It covers three comparisons. First, we compare the baseline performance to a one-hint puzzle. The central hint is similar to Deepzzle configuration. We note that when the hint is central, the puzzle is slightly easier to solve (2%) than when the hint is lateral—in both cases, having a hint improves the performance by +5%.

Second, we see that space has a great influence on our reassembly (-10%), while it has a limited impact on the neural networks' validation scores (Table 10.2). In Table D.5, we present our results for 40×40 fragments and 96×96 fragments, with large space and more simulations. Interestingly, the reassemblies with bigger fragments are more accurate than the one with 40×40, whereas the neural networks' scores are lower.

Last, the reassembly of 4×4 and 5×5 puzzles takes time (in a day, we were able to compute 407 4×4 puzzles and 203 5×5 puzzles).

Table 10.9 presents the errors distribution, depending on how many fragments are well-placed. Note that having a single mistake (i.e., 8 well-placed fragments for a 3×3 puzzle) is impossible.

Number of well-placed fragments									
0	1	2	3	4	5	6	7	8	9
2.98	6.53	10.53	9.98	11.48	18.08	7.43	18.38	(impossible)	14.55

Table 10.9: Reassembly scores — Distribution of errors.

It is rare to have wrongly placed all fragments. Most of the time, two or four fragments are swapped.

10.3.2 *Qualitative analysis*

Figure 10.2 is made of typical reassemblies and their solutions (except for Puzzles (a) and (g) that are correct reassemblies):

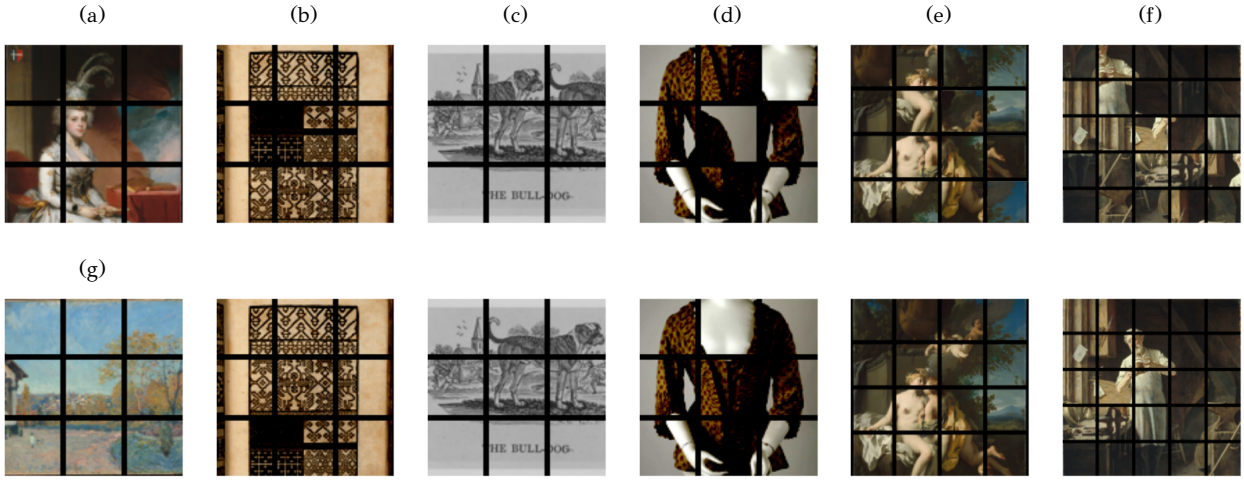


Figure 10.2: The two left images are examples of correct reassemblies. On the right, the first row shows some failure reassemblies made MCTS; their solution is below.

At first glance, we observe that most of the 3×3 reassemblies we made feature many well-placed fragments. Indeed, on such a setup, puzzles with at most 2 mistakes represent more than 30% of the results (Table 10.9).

- Puzzles (a) and (g) are correct reassemblies. While (g) is more difficult than (a), they both cannot be wrong but visually correct.
- In contrast, Puzzle (b) is visually correct, but the rows are inverted.
- Puzzle (c) features an error of *V*: the head and the tail are swapped, despite being placed during the last steps.
- Puzzle (d) illustrates how a misplaced first fragment (the cleavage) alters the rest of the puzzle.
- Puzzles (e) and (f) illustrate how complex the task is for bigger puzzles and show some well-assembled background or bodies fragments. Corrects fragments are grouped, and we obtain better scores with the neighbor-wise metric rather than the fragment-wise metric.

10.4 Results optimization

In this section, we present some optimization we made on Alphazzle: the strategy for selecting an action from MCTS output, the impact of changing the order of the fragments given to MCTS, and the impact of fine-tuning. Last, we present results in comparison with Noroozi and Favaro [NF16] and Deepzzle.

10.4.1 *Order of the fragments*

We expect the order of the fragments fed to MCTS to have a consequent impact on the reassembly score, and run an experiment to validate our hypothesis.

Table 10.10 details the impact of making several attempts to solve a puzzle, with different input fragments reordering:

	Best attempt		Worst attempt		
Number of attempt	Fragment-wise (%)	Puzzle-wise (%)	Fragment-wise (%)	Puzzle-wise (%)	Reassemblies done in 24h
1	55.63	14.55	-	-	2000
5	64.49	42.33	24.20	0.90	2000
10	68.98	44.59	33.48	3.43	466
20	70.62	41.69	36.04	1.80	222

Table 10.10: Reassembly scores — Impact of the order of fragments.

We find out that solving 10 times the same puzzle while reordering the fragments and letting V selecting its favorite leads to a gain of 14% on fragments reassembly score and 30% on puzzle reassembly score. To compare, if we always select the worst reassembly on 10 attempts, we obtain 33.48% and 3.43%, which was lucky compared to the worst attempt we made on the 5 attempts test.

However, the possibility of changing the order of the fragments goes hand in hand with an increase in computing time, limiting the number of reassemblies we made.

When we use multiple fragments order during the inference, we select the best solution according to V rather than the true best solution. Therefore, the final scores may be lower than those displayed in Table 10.10.

10.4.2 *Action choice from MCTS output*

MCTS returns the policy $\pi_{MCTS}(a_t|s_t)$. To select the action to perform, we use $N(a|s_t)$. If we replace it by $Q(a|s_t)$, we observe a slight improvement from 55.63% to 58.11% for fragments accuracy, and from 14.55% to 16.75% for puzzles accuracy.

Table 10.11 shows the impact of using $Q_s(\cdot)$ rather than $N_s(\cdot)$ to compute the policy vector $\vec{\pi}_s$. The other parameters are the number of hints in the reassembly and the number of attempts.

			$N(a\|s_t)$		$Q(a\|s_t)$	
Fragment per side	Hints reassembly	Number of attempts	Fragment-wise (%)	Puzzle-wise (%)	Fragment-wise (%)	Puzzle-wise (%)
3	0	1	55.63	14.55	58.11	16.75
3	0	10	64.49	42.33	72.28	39.56
3	1 (central)	10	80.66	45.95	82.47	51.50
4	0	10	35.32	0.74	32.49	0.76
5	0	10	15.92	0.00	14.04	0.00

Table 10.11: Reassembly scores — Impact of $Q(a|s_t)$.

We observe that on 3×3 puzzles, $Q(a|s_t)$ is better than $N(a|s_t)$.

Note that the puzzles are not identical from one generation to another, which explains why $Q(a|s_t)$ may have a lower puzzle-wise score. On bigger puzzles, it is better to prefer $N(a|s_t)$.

10.4.3 *Comparison with other methods*

Table 10.12 displays the puzzle-wise score for different reassembly algorithms:

	Number of available permutations, i.e., terminal nodes.					
Algorithm	10	10^2	10^3	$9! \simeq 10^6$	$16! \simeq 10^{13}$	$25! \simeq 10^{25}$
Noroozi and Favaro [NF16]	86.6	69.3	51.6	overflow	overflow	overflow
Deepzzle, without central fragment	91.5	81.7	64.8	39.2	overflow	overflow
Alphazzle, without central fragment	n/a	n/a	n/a	39.56	0.76	0.0
Deepzzle, with central fragment	n/a	n/a	n/a	44.4	overflow	overflow
Alphazzle, with central fragment	n/a	n/a	n/a	51.50	0.0	0.0

Table 10.12: Reassembly scores — Comparison with the literature, with one central fragment already placed, 1000 simulations and 10 attempts.

We make 10 attempts before selecting which reassembly is correct, and we use $Q(a|s_t)$. We did not implement a way to limit the available permutations; therefore, we do not compare to the lowest numbers of available permutations.

With a central fragment, we outperform [PPT20] by 7% on the puzzle accuracy.

In terms of computational cost, our method greatly outperforms [CDB+19, NF16, PPT20]. Not only do we address all possible $n!$ permutation, but we can propose reassemblies to 4×4 and 5×5 puzzles in a few hours, whereas state of the art cannot rely on deep learning to solve such puzzles.

10.4.4 *Impact of fine-tuning*

Table 10.13 shows the fine-tuning results on various epochs sizes:

Fine-tuning epoch	0	1	3	5	7	10	15	20
Fragment accuracy (%)	55.63	52.79	53.63	54.36	53.92	54.63	54.49	56.72
Puzzle accuracy (%)	14.55	17.35	18.25	20.05	20.40	20.00	20.30	23.10

Table 10.13: Reassembly scores — Detail on fine-tuning accuracy for 3×3 puzzles with 500 puzzles generated per iteration.

Note that we make 100 simulations rather than 1000, leading MCTS to make more mistakes, which should provide more relevant training samples. We use $N(a|s_t)$ and only make one attempt to generate 500 training puzzles and evaluate the validation images' results.

After a few epochs, we significantly improve the puzzle reassembly scores by almost 9%. Strangely, the fragment accuracy stays the same. It means that on average, fine-tuning improves easy puzzle up to the point where the reassembly is correct while it has a negative impact

on challenging puzzles.

As we build the fine-tuning dataset on the previous reassemblies done by Alphazzle, we can choose to pick the best action from the MCTS output ("best"), or select an action randomly, using π_{MCTS} as a probability distribution ("softmax"), to add some noise and make our method more robust. We call the choice the "action choice".

Table 10.14 shows the reassembly scores obtained with fine-tuning, with a learning rate of 0.0001 and 100 simulations for MCTS:

Fragment per side	Fine-tuning batch size	Qt of reassemblies in solving	Action choice	Fragment-wise (%)	Puzzle-wise (%)	Qt of puzzles seen
3	100×3^2	100	Best	56.44	33.00	28 000
3	100×3^2	100	Softmax	61.11	30.00	10 800
3	100×3^2	2000	Best	54.07	20.05	2 500
3	200×3^2	2000	Best	55.11	20.06	4 800
3	500×3^2	2000	Best	56.72	23.10	10 000
3	500×3^2	2000	Softmax	53.96	18.80	5 000
4	100×4^2	100	Best	23.94	0.10	2 200
4	500×4^2	2000	Best	24.51	0.35	1 000
5	100×5^2	100	Best	15.60	0.00	1 000
5	500×5^2	2000	Best	14.32	0.00	500

Table 10.14: Reassembly scores with fine-tuning, with 100 simulations and lr = 0.0001.

Thanks to fine-tuning, we have been able to reach 33% of well-solved puzzles.

The last column detail how puzzles have been seen during training. It allows mitigating the results for bigger batches size and bigger puzzles.

10.4.5 *Combination of the best parameters*

To conclude, we combine the standard experiment with fine-tuning, $Q(a|s_t)$ and 10 attempts. Table 10.15 shows the scores we obtain:

Fragment-wise (%)	Neighbor-wise (%)	Puzzle-wise (%)
75.12	77.54	51.49

Table 10.15: Reassembly scores — Combination of the best parameters.

Part IV

EPILOGUE

11

Conclusion

◂ Chapter 10 Appendix A ▸

11.1 LOOKING BACK

Puzzle-solving with neural networks started as a simple pretext task aiming to improve state-of-art neural networks [DGE15, NVFP18, CDB+19, KCYK18], but is now a standalone goal, which we believe to be very interesting for the community. Indeed, it is one of the very few tasks that combine two traditionally opposed aspects of artificial intelligence: visual understanding and choice space exploration. Robotics exhibits a similar positioning between these two domains, with the differences that the choice space is much larger and that there are many optimal action paths. In contrast, puzzle-solving occurs in a controlled environment characterized by a narrow choice space and a unique correct path—given an ordered set of patches. Thus, it exhibits excellent properties to study the combination of visual understanding with choice space exploration.

Hybrid approaches combining search algorithms and deep learning are potent and could be applied to a wide range of domains, such as autonomous driving. They are important globally since they offer several advantages over using an empirical predictor alone (neural networks or other machine learning algorithms). Namely, they offer some level of interpretability by showing the path leading to the selected solution. Being able to explain how the solution was obtained is crucial in decision processes that impact people (e.g., healthcare admission). Successive decisions can also be constrained by explicit rules that could prevent the system from choosing an unfair solution based on biased empirical evidence.

In this dissertation, we obtained results that improve state of the art in puzzle-solving thought two hybrid approaches:

- Deepzzle, which performs pairwise comparisons and minimizes the joint probabilities with a graph-based heuristic;
- Alphazzle, which performs a global comparison and uses MCTS to explore the consequences of its actions.

These contributions make a significant step forward in our ability to solve puzzle automatically. Our results suggest that solving complex decision processes by leveraging help from a deep neural network requires more research to be used for practical applications effectively.

11.2 LOOKING AHEAD

11.2.1 *On heritage*

Probably the thing we are most eagerly looking forward to is to train our algorithms with new heritage data.

For the Taillebourg cave, our algorithms need a lot of similar data to be able to assemble fragments, which is challenging to obtain given the small number of known Magdalenian carved caves. Besides, carvers of Roc-aux-Sorciers used the asperities of the wall to carve their art; therefore, the semantics data are hard to detect on the stones. Annotated data that highlight the semantics may be helpful, as well as 3D scans. If no semantic annotation for fragments can be provided, we recommend to digitize the fragments and apply a contour-based algorithm, hoping that the erosion is low. If no pertinent reassembly is found, a more robust solution that combines contours, patterns, and semantics should be considered.

For Roc-aux-Sorciers, Deepzzle obtains almost state-of-the-art results on the bas-relief dataset, which is similar to the temple's carved blocks. On the 3D scans, the results are lower. We think that digitizing (or generating) more 3D blocks is enough to obtain correct reassemblies.

11.2.2 *Short-term projects*

We implemented rectangle puzzles to compare Alphazzle with Bridger et al. [BDT20], who obtained great results on 64+ pieces jigsaw puzzles. We also allow our algorithm to resize the partial reassemblies to be able to process them, and we are currently training the neural networks with such configuration. Our goal is to see if Alphazzle can compete with the tedious greedy solver of [BDT20]. We also believe that our algorithm should obtain better results because Bridger et al. make a 4-classes pairwise comparison.

We also recently implemented neighbor-wise metric and updated PUCT with [SWTU12] and [JGT14]. We are running experiments on these two topics.

11.2.3 *Optimizing Deepzzle*

We have several suggestions for improving Deepzzle's solver:

- Comparing Dijkstra's algorithms with other tree traversal algorithms[1] and the greedy algorithm with the Hungarian algorithm;
- Applying the greedy algorithm to bigger problems with many extra-fragments;
- Designing a mega-solver that can merge several reassemblies, based on the classifier outputs: we use the neural network to evaluate

[1] Tree traversal on Wikipedia.

many pairs of fragments and use a solver that can solve the puzzles we obtained.

11.2.4 *Improving Alphazzle*

We have several suggestions for improving Deepzzle's solver:

- Replacing the convolutionals networks by transformers[2]; we expect transformers to be more effective to process the fragments than P and V. Moreover, they allow unlimited number of fragments per puzzle.
- Gaining more insight on MCTS choices by running a step by step analysis of the tree expansion;
- Implementing an almost-perfect metric to compete more fairly with Deepzzle, as well as missing and extra-fragments.
- Comparing the type of reward: currently, the predicted reward is based on the solved-puzzle metric; we wonder if a reward based on the number of well-placed fragments or well-placed neighbors can outperform the current reward.

[2] Transformers [VSP+17] are build on an attention mechanism and are very effective for performing sequential tasks based on what happened before, such as completing a sentence.

11.2.5 *New horizons*

Ultimately, the methods we have developed in this dissertation are not the only ones with potential. Alphazzle may benefit from different rules, such as the ability to skip (or replace) a fragment that is challenging to place. Another pertinent rule would be to authorize for shifting the placed fragments within the puzzle size: if we place the central fragment on top, we want to shift it rather than have to go back up the tree.

A second method worth exploring is building a regressor that predicts the relative position in terms of coordinates rather than classes. It would allow us to place fragments at their exact position and make it easier to dispense with the fragments' fixed size. However, our toy implementation with triplet loss was unsuccessful; we expect making such method work would require a lot of research and effort.

"Thence we came forth to rebehold the stars."
– Dante Alighieri, *Divine Comedy, Inferno, Canto XXXIV*

Artwork 7: *Landscape with Stars*, Henri-Edmond Delacroix, ca. 1905, from the MET Open Collections.

References

[ACBF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[AMSB19] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

[AN04] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[ATB17] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[AZH11] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[BDT20] Dov Bridger, Dov Danon, and Ayellet Tal. Solving jigsaw puzzles with eroded boundaries. 2020.

[BHT+18] Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. Sample-efficient reinforcement learning with stochastic ensemble value expansion. In *Advances in Neural Information Processing Systems*, pages 8224–8234, 2018.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[BW12] Hendrik Baier and Mark HM Winands. Nested monte-carlo tree search for online planning in large mdps. In *ECAI*, volume 242, pages 109–114, 2012.

[BYCCT17] Hedi Ben-Younes, Rémi Cadene, Matthieu Cord, and Nicolas Thome. Mutan: Multimodal tucker fusion for visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2612–2620, 2017.

[Caz09] Tristan Cazenave. Nested monte-carlo search. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[CDB+19] Fabio M Carlucci, Antonio D'Innocente, Silvia Bucci, Barbara Caputo, and Tatiana Tommasi. Domain generalization by solving jigsaw puzzles. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2229–2238, 2019.

[CKK17] Tatsuya Chuman, Kenta Kurihara, and Hitoshi Kiya. Security evaluation for block scrambling-based etc systems against extended jigsaw puzzle solver attacks. In *2017 IEEE International Conference on Multimedia and Expo (ICME)*, pages 229–234. IEEE, 2017.

[Cou06] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[D+59] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[DGE15] Carl Doersch, Abhinav Gupta, and Alexei A Efros. Unsupervised visual representation learning by context prediction. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1422–1430, 2015.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[DTS18] Niv Derech, Ayellet Tal, and Ilan Shimshoni. Solving archaeological puzzles. *arXiv preprint arXiv:1812.10553*, 2018.

[EDSM19] Yonathan Efroni, Gal Dalal, Bruno Scherrer, and Shie Mannor. How to combine tree-search methods in reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3494–3501, 2019.

[FB81] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[Fuk80] Kunihiko Fukushima. Neocognitron: A self-organizing neural network for a mechanism of pattern recognition unaffected by shift in position. *BiologicalCybernetics*, 1980.

[Gal12] Andrew C Gallagher. Jigsaw puzzles with pieces of unknown orientation. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 382–389. IEEE, 2012.

[GBS17] Shir Gur and Ohad Ben-Shahar. From square pieces to brick walls: The next challenge in solving jigsaw puzzles. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4029–4037, 2017.

[GBZD16] Yang Gao, Oscar Beijbom, Ning Zhang, and Trevor Darrell. Compact bilinear pooling. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 317–326, 2016.

[GW06] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. 2006.

[HF18] Yaser Hashem and Joachim Frank. The jigsaw puzzle of mrna translation initiation in eukaryotes: A decade of structures unraveling the mechanics of the process. *Annual review of biophysics*, 47:125–151, 2018.

[HFG+06] Qi-Xing Huang, Simon Flöry, Natasha Gelfand, Michael Hofer, and Helmut Pottmann. Reassembling fractured objects by geometric matching. In *ACM SIGGRAPH 2006 Papers*, pages 569–578. Association for Computing Machinery, 2006.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HSS12] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), 2012.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IL67] Alexey. G. Ivakhnenko and Valentin G. Lapa. *Cybernetics and Forecasting Techniques*. 1967.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JGT14] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. Monte mario: platforming with mcts. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 293–300, 2014.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KCYK18] Dahun Kim, Donghyeon Cho, Donggeun Yoo, and In So Kweon. Learning image representations by completing damaged jigsaw puzzles. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 793–802. IEEE, 2018.

[KHLT19] Bilal Kartal, Pablo Hernandez-Leal, and Matthew E Taylor. Action guidance with mcts for deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, pages 153–159, 2019.

[KOL+16] Jin-Hwa Kim, Kyoung-Woon On, Woosang Lim, Jeonghee Kim, Jung-Woo Ha, and Byoung-Tak Zhang. Hadamard product for low-rank bilinear pooling. *arXiv preprint arXiv:1610.04325*, 2016.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBD+89] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[Lev73] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[LFJ+18] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:1807.01672*, 2018.

[LRM15] Tsung-Yu Lin, Aruni RoyChowdhury, and Subhransu Maji. Bilinear cnn models for fine-grained visual recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 1449–1457, 2015.

[MASB18] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with approximate policy iteration. In *International Conference on Learning Representations*, 2018.

[MRS10] Nicolas Mellado, Patrick Reuter, and Christophe Schlick. Semi-automatic geometry-driven reassembly of fractured archeological objects. 2010.

[NF16] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In *European Conference on Computer Vision*, pages 69–84. Springer, 2016.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[NKFL18] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[NVFP18] Mehdi Noroozi, Ananth Vinjimoor, Paolo Favaro, and Hamed Pirsiavash. Boosting self-supervised learning via knowledge transfer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9359–9367, 2018.

[OBA20] Cecilia Ostertag and Marie Beurton-Aimar. Matching ostraca fragments using a siamese neural network. *Pattern Recognition Letters*, 2020.

[OLLW18] Laurent Orseau, Levi Lelis, Tor Lattimore, and Théophane Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pages 3201–3211, 2018.

[PAJ19] Antoine Pirrone, Marie Beurton Aimar, and Nicholas Journet. Papy-s-net: A siamese network to match papyrus fragments. In *Proceedings of the 5th International Workshop on Historical Document Imaging and Processing*, pages 78–83, 2019.

[PPT18a] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Image reassembly combining deep learning and shortest path problem. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 153–167, 2018.

[PPT18b] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Jigsaw puzzle solving using local feature co-occurrences in deep neural networks. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 1018–1022. IEEE, 2018.

[PPT20] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Deepzzle: Solving visual jigsaw puzzles with deep learning and shortest path optimization. *IEEE Transactions on Image Processing*, 29:3569–3581, 2020.

[PPT21] Marie-Morgane Paumard, David Picard, and Hedi Tabia. Solving jigsaw puzzle with deep monte-carlo treesearch. 2021.

[PSA+17] Georgios Papaioannou, Tobias Schreck, Anthousis Andreadis, Pavlos Mavridis, Robert Gregor, Ivan Sipiran, and Konstantinos Vardis. From reassembly to object completion: A complete systems pipeline. *Journal on Computing and Cultural Heritage (JOCCH)*, 10(2):1–22, 2017.

[PSBS11] Dolev Pomeranz, Michal Shemesh, and Ohad Ben-Shahar. A fully automated greedy square jigsaw puzzle solver. In *CVPR 2011*, pages 9–16. IEEE, 2011.

[PT15] Genady Paikin and Ayellet Tal. Solving multiple square jigsaw puzzles with missing pieces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4832–4839, 2015.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[RM87] David E. Rumelhart and James L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1987.

[RN15a] Nada A Rasheed and Md Jan Nordin. A survey of classification and reconstruction methods for the 2d archaeological objects. In *2015 International Symposium on Technology Management and Emerging Technologies (ISTMET)*, pages 142–147. IEEE, 2015.

[RN15b] Nada A Rasheed and Md Jan Nordin. A survey of computer methods in reconstruction of 3d archaeological pottery objects. *International Journal of Advanced Research*, 3(3):712–714, 2015.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Ros11a] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[Ros11b] Christopher D Rosin. Nested rollout policy adaptation for monte carlo tree search. In *Ijcai*, pages 649–654, 2011.

[RWR+17] Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in neural information processing systems*, pages 5690–5701, 2017.

[SAH+19] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

[SAPM17] Michalis A. Savelonas, Anthousis Andreadis, Georgios Papaioannou, and Pavlos Mavridis. Exploiting unbroken surface congruity for the acceleration of fragment reassembly. In *GCH*, pages 137–144, 2017.

[SB20] Arta Seify and Michael Buro. Single-agent optimization through policy iteration using monte-carlo tree search. *arXiv preprint arXiv:2005.11335*, 2020.

[SCFCG17] Rodrigo Santa Cruz, Basura Fernando, Anoop Cherian, and Stephen Gould. Deeppermnet: Visual permutation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3949–3957, 2017.

[SDN13] Dror Sholomon, Omid David, and Nathan Netanyahu. A genetic algorithm-based solver for very large jigsaw puzzles. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2013.

[Sei20] Arta Seify. Single-agent optimization with monte-carlo tree search and deep reinforcement learning. Master's thesis, 2020.

[SF17] Elena Sizikova and Thomas Funkhouser. Wall painting reconstruction using a genetic algorithm. *Journal on Computing and Cultural Heritage (JOCCH)*, 11(1):1–17, 2017.

[SHC14] Kilho Son, James Hays, and David B Cooper. Solving square jigsaw puzzles with loop constraints. In *European Conference on Computer Vision*, pages 32–46. Springer, 2014.

[SHC+16] Kilho Son, James Hays, David B Cooper, et al. Solving small-piece jigsaw puzzles by growing consensus. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1193–1201, 2016.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[Sil09] David Silver. *Reinforcement Learning and Simulation-Based Search*. PhD thesis, 2009.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[SWTU12] Maarten PD Schadd, Mark HM Winands, Mandy JW Tak, and Jos WHM Uiterwijk. Single-player monte-carlo tree search for samegame. *Knowledge-Based Systems*, 34:3–11, 2012.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TFSM02] Fubito Toyama, Yukihiro Fujiki, Kenji Shoji, and Juichi Miyamichi. Assembly of puzzles using a genetic algorithm. In *Object recognition supported by user interaction for service robots*, volume 4, pages 389–392. IEEE, 2002.

[TIY19] Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning to create value and policy functions using minimax tree search in hex. *IEEE Transactions on Games*, 12(1):63–73, 2019.

[TKAM15] Despoina Tsiafaki, Anestis Koutsoudis, Fotis Arnaoutoglou, and Natasa Michailidou. Virtual reassembly and completion of a fragmentary drinking vessel. *Virtual Archaeology Review*, 7(15):67–76, 2015.

[TP09] Efthymia Tsamoura and Ioannis Pitas. Automatic color based reassembly of fragmented images and paintings. *IEEE Transactions on Image Processing*, 19(3):680–690, 2009.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[VSS17] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. On monte carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, 2017.

[WWL+19] I-Chen Wu, Ti-Rong Wu, An-Jen Liu, Hung Guei, and Tinghan Wei. On strength adjustment for mcts-based programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1222–1229, 2019.

[WXR+19] Chen Wei, Lingxi Xie, Xutong Ren, Yingda Xia, Chi Su, Jiaying Liu, Qi Tian, and Alan L Yuille. Iterative reorganization with weak spatial constraints: Solving arbitrary jigsaw puzzles for unsupervised representation learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1910–1919, 2019.

[YWLM11] Zhao Yin, Li Wei, Xin Li, and Mary Manhein. An automatic assembly and completion framework for fragmented skulls. In *2011 International Conference on Computer Vision*, pages 2532–2539. IEEE, 2011.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[ZL14] Kang Zhang and Xin Li. A graph-based optimization algorithm for fragmented image reassembly. *Graphical Models*, 76(5):484–495, 2014.

[ZYM+15] Kang Zhang, Wuyi Yu, Mary Manhein, Warren Waggenspack, and Xin Li. 3d fragment reassembly using integrated template guidance and fracture-region matching. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2138–2146, 2015.

[ZZZH06] Liangjia Zhu, Zongtan Zhou, Jingwei Zhang, and Dewen Hu. A partial curve matching method for automatic reassembly of 2d fragments. In *Intelligent Computing in Signal Processing and Pattern Recognition*, pages 645–650. Springer, 2006.

Part V

GENERAL APPENDIX

A

Introduction to deep learning

◂ Chapter 11 Appendix B ▸

A.1 PROLOGUE

Artificial intelligence's overall ambition is to imitate natural intelligence in terms of input understanding, knowledge reasoning, learning, planning, and decision-making. Examples of tasks include driving autonomous cars, retrieving faces in images, or successfully understanding human speech. In all these three tasks, state-of-the-art results have been achieved through deep learning methods.

In this chapter, we put deep learning for computer vision into context §A.2, we explain its mechanisms §A.3 and introduce reinforcement learning §A.4.

A.2 CONTEXT

A.2.1 *What is learning ?*

Learning is the process of acquiring new knowledge, whether in terms of skills, behaviors, or understandings. Humans, animals, some plants, and some machines exhibit learning abilities. Most of the time, learning occurs through repeated experiences.

When an algorithm improves automatically from experience, we refer to it as machine learning.

A.2.2 *Machine learning*

Machine learning proposes methods to create models that can learn how to perform a single task with varying degrees of success. Then, the task can be performed on new data (generalization).

The learning efficiency depends on many parameters, particularly the number of repetitions, the relevance of the dataset, and the model's descriptivity.

Machine learning approaches are traditionally divided into three broad categories:

Supervised learning The algorithm is given couples of inputs and their desired outputs. It learns the function mapping inputs to outputs. When the outputs are not labeled manually but generated by the algorithm prior to learning, it is often called *self-supervised learning*. Deepzzle, from Chapter 6, is self-supervised.

Unsupervised learning The algorithm is given inputs and learns to extract structure in the data. For instance, it can group similar data together.

Reinforcement learning The algorithm interacts with an environment with which it can interact. It is provided with an objective, and the interactions that contribute to its achievement are rewarded. It learns to maximize the reward. Alphazzle, from Chapter 9, uses reinforcement.

The machine learning toolbox includes, among others, decision trees, support vector machines, genetic algorithms, and of course, artificial neural networks.

Deep learning is part of machine learning methods: it is based on artificial neural networks aggregated into several layers.

A.2.3 *Deep learning*

Deep neural networks have been researched from 1967 [IL67]. First algorithms have been based on perceptron architecture [Ros58], and they have rapidly evolved thanks to a series of innovations. To name but a few: convolutional neural networks, their architectures [Fuk80, LBD+89, KSH12, HZRS16], backpropagation [RHW86, LBBH98], regularization techniques like dropout [SHK+14], batch-normalisation [IS15], or data augmentation, and large annotated datasets such as ImageNet ([RDS+15]).

Too computationally expensive, deep neural networks were not democratized until 2012. This year, they won ImageNet Large Scale Visual Recognition Challenge (ILSVRC), anchoring the start of a "deep learning revolution" that transformed the artificial intelligence field. One reason for this popularity is deep neural networks' ability to extract features without a human's need, unlike traditional machine learning.

Deep learning models can solve tasks on various types of data, such as images (computer vision), languages (natural language processing), signals, and structured data. As puzzle-solving depends on computer vision, we focus on it in the following.

A.2.4 *Computer Vision*

Computer vision is a research field at the crossroads of artificial intelligence, neurobiology, and signal (image) processing. It focuses on the automatic processing of images, 3D rendering, and videos. Our human brain is hardwired to process any visual information because we rely on semantics efficiently. We perceive objects and can even deduce events (the road is darker than usual; hence, it is wet; thus, it rained). In the "eyes" of a computer, images are only a sequence of color pixels[1], no more, no less. They must be interpreted in order to provide artificial intelligence with visual capacities similar

[1] A pixel is a tuple of three values: red, green, and blue.

to ours. To that end, computer vision relies on various techniques such as deep learning.

The jigsaw puzzle-solving task is an exotic example of a computer vision task. Some common tasks include image labeling, image retrieval, object tracking in video, face identification, 3D scene reconstruction, visual quality inspection, medical image interpretation, video annotation, species identification, people counting, pose estimation, robot control, autonomous driving, image restoration, photo editing, "in the style of" artwork generation, etc.

A.3 SUPERVISED DEEP LEARNING FOR COMPUTER VISION

A deep learning algorithm is made of many components, which we detail below. We focus on the classification task applied to images.

A.3.1 *Solving a task*

In the beginning, there is a task, i.e., a question that we want to solve automatically for each element of a dataset. For instance, we want to classify all images of a dataset in cat and dog categories. In this case, the images are the *inputs* of the algorithm, and the classification is the *task*. The algorithm will then learn to map each input to an answer: its answer is the *output*. Solving a task, i.e., applying an already-trained algorithm to data, is referred to as *inference*.

Once we have determined the task, we need to choose a simple data structure to represent the answer. For cats and dogs classification, the most straightforward output is a binary variable that is True when the picture is classified as a cat picture. To capture the incertitude, we can opt for an output that ranges from 0 to 1: a value of 0.2 means that the chances that the picture represents a dog are 80%. If we want to recognize more categories of animals, i.e., *classes*, the answers would be structured as a vector that sums to one. Some algorithms have many classes, and others have none (such as regression task or unsupervised-learning).

A.3.2 *Structuring of a neural network*

The universal approximation theorem states that we can approximate any continuous function with a deep neural network, for inputs within a specific range. In other words, it can solve complex tasks from raw data, such as pixels.

To complete a task, a deep neural network learns to regroup pixels and extract significant visual features that enable solving the task. For instance, studying the sky's color does not discriminate between dogs and cats, while comparing their ears' shape helps. Each feature is a non-linear combination of other features (or pixels).

In a neural network, each *neuron* computes a feature. They all apply a non-linear function $f_{w,b}()$ on their input, which is characterized by

weights w and a *bias* b. Together, all the weights and the biases are the *parameters* of the neural network.

From an input x_i, a neuron outputs a feature $f_{w,b}(w \cdot x_i + b)$. We organize neurons through layers and link them together. Among all the ways of connecting artificial neurons, i.e., *architectures*, some have been thoroughly tested and approved. Examples are fully-connected feed-forward networks, recurrent neural networks, auto-encoders, and convolutional networks. Figure A.1 presents a deep feed-forward network composed of 5 layers (one input layer for the pixels, three hidden layers and a 3-classes output layer).

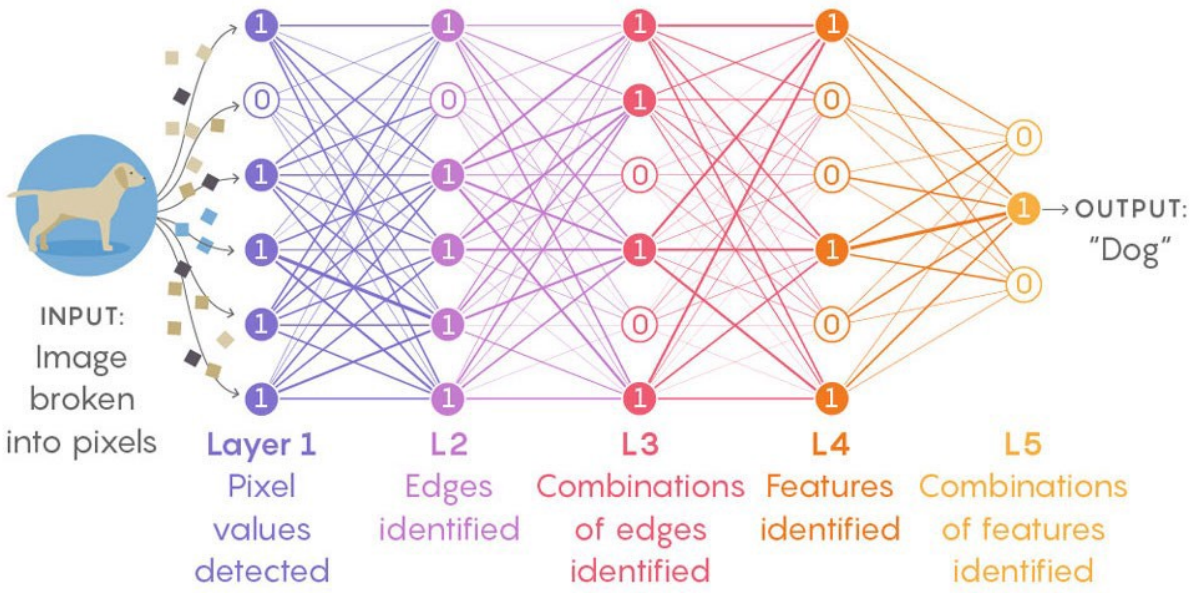


Figure A.1: A deep fully-connected neural network architecture. © Lucy Reading-Ikkanda, Quanta Magazine.

For the sake of illustration, the picture shows 6 neurons by layer, whereas there are usually hundreds or even thousands of neurons per layer. A binary value is attributed to each neuron: it indicates whether the neuron is activated.

Convolutional Neural Networks The most common architecture for extracting features from pixels is convolutional neural networks. Similarly to the visual cortex, a convolutional neural network naturally detects contours in the first layer, assembles them in textures in the second layer, and the next layers forms shapes and objects.

A *convolution* is an operation that applies a filter to all the patches of an image and retains spatial information. The filter is specific to a pertinent feature and returns a value indicating its intensity within a patch. In the case of a *convolutional layer*, the number of filters is the number of neurons, since each neuron performs a different convolution on the layer's input. The neurons' parameters form convolution filters.

Usually, convolutions layers alternate with *pooling layers*. Their

goal is to bring the features closer: if we use the same-sized filter, it can be perceived and be applied to "bigger" elements. To put it simply, if 10×10 pixels are seen from an 100×100 image, the filter only sees a tenth of it. If we apply a pooling that reduced the image size by a factor 10, the filter now sees all the images at once.

The standard architecture for visual object classification starts with several convolutional layers and pooling layers, to which we append a fully-connected network that proceeds to the classification. Figure A.2 shows some standard architectures [LBBH98, KSH12, SZ14]. VGG-16 inspires Deepzzle.

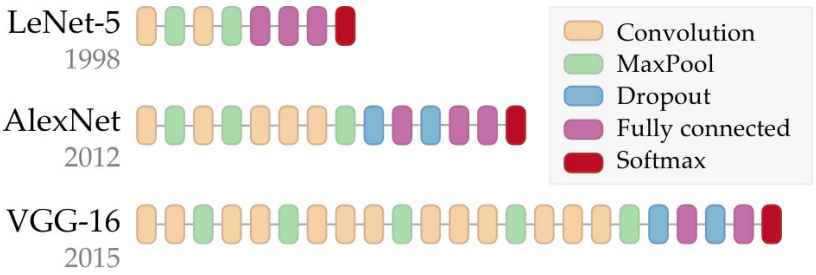


Figure A.2: Some standard architectures for computer vision. © T. Robert.

A.3.3 *Learning process*

We have seen that a neural network can extract features through its neurons. Their parameters determine which features are perceived. Therefore, we want to optimize the parameters so that the perceived features allow us to solve the task. However, at first, the parameters are randomly initialized. The parameters optimization is called *fitting*.

The principle of supervised learning is as follows: we have many examples of inputs x, and we know their desired outputs y, their classes, which are associated with them. If we feed the neural network with the inputs, we can observe the outputs $\hat{y}$ and compare it to y. We call $\hat{y}$ the prediction and y the (ground-truth) labels associated with the inputs.

We measure the error between y and $\hat{y}$ with a loss function $\mathcal{L}$. The objective is to find weights and bias that minimize the loss. We apply a gradient descent [RHW86, Bot10, HSS12, DHS11, KB14] and update them in a direction that decrease the loss value: $w \leftarrow w - \lambda \nabla_w \mathcal{L}(\hat{y}, y)$ and $b \leftarrow b - \lambda \nabla_b \mathcal{L}(\hat{y}, y)$.

When recursively applied to a neural network, this method is called (gradient) back-propagation [RHW86]. Based on the chain-rule, we iteratively compute the updated weights and biases of the whole network.

Progressively, over many repetitions of this fitting process, we converge towards a local minimum of the loss function.

A.4 DEEP REINFORCEMENT LEARNING

Deep reinforcement learning occurs when an algorithm learns to make a decision that affects its environment. It is not provided with labels but rewards, and it has to deduce how to maximize them. We note that this goal is much more abstract than the goal of supervised learning. In this last case, we know we should reduce the loss to 0; in reinforcement, the algorithm does not know how much reward it can obtain for its actions.

Another difference is in the way of apprehending the results. A supervised model produces outputs that do not affect the environment; most of the time, they even are independent. A deep reinforcement model makes sequential decisions that impact its environment: each decision depends on the current state of the environment, which is the consequence of the actions previously taken. Due to the changing environment, input data must be generated after each action.

The goal of reinforcement learning is to produce a policy $\pi_\theta(a|s)$, which is a probability distribution over the actions $a \in A$, given the current environment state $s \in S$. The policy can be represented in different ways, such as a Gaussian process or neural network. In this latter case, it is called deep reinforcement learning, and θ are the network's parameters.

Each action a_t is rewarded by the environment depending on the state, with t the step. The reward is $r(s_t, a_t)$. Therefore, a reinforcement learning algorithm is willing to find the optimal parameters $\theta^\star$ in such a way that $\theta^\star = \arg\max_\theta \mathbb{E} \sum_t r(s_t, a_t)$

Usually, the environment is not determined solely by the actions, and the future state s_{t+1} is obtain through a probability distribution $p(s_{t+1}|s_t, a_t)$. All these components define a Markovian decision process $M = \{S, A, p, r\}$.

Four types of algorithms optimize the reward: policy gradients methods, value-based methods, actor-critic methods, and model-based methods. As a planning algorithm, Monte-Carlo Tree Search is often considered as part of model-based methods.

B

Introduction to decision theory

◂ Appendix A Appendix C ▸

B.1 PROLOGUE

Decision theory, also known as decision science, is another branch of artificial intelligence that studies how to optimize decisions mathematically. This research field is structured in overlapping themes, such as game theory, operational research, systems engineering, business intelligence, financial engineering, management science, and applied mathematics. All of these themes have strong ties to analytics and computer science.

B.2 COMPLEX PROBLEMS

Operation research addresses most optimization and decision problems, as long as it is complex enough to benefit from operation research tools. A problem is complex when it is combinatorial, stochastic, or competitive:

Combinatorial problem The problem includes a large number of permissible solutions, among which an optimal or near-optimal solution is sought. If the number of inputs increases, the number of solutions faces a combinatorial explosion. Consequently, a combinatorial problem cannot be solved by a simple enumeration of possible solutions.

Stochastic problem The problem consists of finding an optimal solution to a problem that arises in uncertain terms.

Competitive problem The problem consists of finding an optimal solution to a problem whose terms depend on the interrelation between one's actions and those of other decision-makers.

Puzzle-solving is a combinatorial problem, like the traveling salesman problem and the knapsack problem.

B.3 MAIN CLASSES OF METHODS

Heuristics and metaheuristics A heuristic is a technique that finds an approximate solution by trading optimality for speed.

Tree and graph tranversal These classes of methods refers to the process of visiting the edges of a graph in order to find an optimal

solution to a problem, such as enumeration, shortest-path, minimal spanning tree, and coloring. Some algorithms examples are Branch and Bound, Dijkstra's algorithm, Kruskal's algorithm, and A*.

Dynamic programming It is a technique that solves overlapping recursive sub-problems, for example, with functional programming and memoization.

Constraint programming It is a programming paradigm that states the constraints and specifies the method to be used to solve them. Standard methods include chronological backtracking and constraint propagation.

Other methods draw upon polynomial algorithms, stochastic processes, linear and non-linear optimization, linear complementarity methods, and computer simulation.

B.4 PUZZLE-SOLVING NOTATIONS AND FORMULATION

We use the same notations as those introduced in §6.3.

A jigsaw puzzle is an assignment problem where each fragment $i \in [0\,..\,f]$ has to be associated with a position $j \in [0\,..\,p]$.

We note the binary assignment variable $x_{i,j}$. It is equal to 1 if fragment i is is placed at position j. We define position 0 as the central position and fragment 0 as the central fragment, and introduce $x_c = x_{0,0} = 1$, the placement of the central fragment at the central position.

Last, we introduce $Pr(i, j|0)$, the probability of placing fragment i in position j, given that fragment 0 is central. In Deepzzle, Pr is evaluated by the neural network.

The assignment problem objective is:

$$\max_{x_{i,j}} \sum_{i,j} P(i, j|x_c) \cdot x_{i,j} \tag{B.1}$$

under the constraints:

$$\forall j \geq 1, \sum_{i=1}^{f} x_{i,j} = 1\ , \tag{B.2}$$

$$\forall i \geq 1, \sum_{j=1}^{p} x_{i,j} = 1\ , \tag{B.3}$$

$$\forall i \geq 1, j > 1, x_{i,j} \in \{0, 1\}. \tag{B.4}$$

Only one fragment can occupy a position (Equation B.2) and a fragment can be placed only once (Equation B.3).

Then, if we allow the puzzle to be uncompleted (i.e. some positions are not used), we replace the constraint B.2 with:

$$\forall j \geq 1, \sum_{i=1}^{f} x_{i,j} \leq 1. \tag{B.5}$$

Similarly, if we have supernumerary fragments (i.e. some fragments are not used), we replace the constraint B.3 with:

$$\forall i \geq 1, \sum_{j=1}^{p} x_{i,j} \leq 1. \tag{B.6}$$

Finally, if we do not know which fragment is the central fragment, we have to solve the extended assignment problem where one fragment has to be assigned to the central position and the remaining fragment are assigned to the relative positions. This leads to the following problem:

$$\max_{c, x_{i,j}} \sum_{i \neq c, j} P(i, j|c) \cdot x_{i,j} \tag{B.7}$$

under the following constraints:

$$\begin{aligned}
\forall c, j, \sum_{i \neq c, i=0}^{f} x_{i,j} &\leq 1; \\
\forall i \neq c, \sum_{j=0}^{p} x_{i,j} &\leq 1; \\
\forall i, j, x_{i,j} &\in \{0, 1\}; \\
\forall c, j \geq 1, x_{c,0} = 1 \text{ and } x_{c,j} &= 0.
\end{aligned}$$

Part VI

TECHNICAL APPENDIX

C

On the graphs sizes

◂ Appendix B Appendix D ▸

Let G be the graph of the reassembly paths for a puzzle with f lateral fragments and p available positions. We define $|N|$ the number of node and $|E|$ the number of edges of G. G is similar to a tree of height $f+1$ (for the source S) whose all leaves are linked together to the target T, so the height of G is $f+2$. We number the lines from 0 for the source, so each intermediate lines number correspond to the fragment number and the last line number is $f+1$.

We start by calculating $nodes(l)$, the number of nodes for the l-th line of G, because $|N| = \sum_{l=0}^{f+1} nodes(l)$. $|E|$ can be easily deduced from $|N|$: for each couple of lines m that groups l and $l+1$, $edges(m) = nodes(l+1)$, with the exception of the last line where $edges(f+1) = nodes(f)$.

C.1 GRAPHS WITHOUT EXTRA-FRAGMENTS

The extra-fragments are disabled, which means we have either the same number of fragments and positions $f = p$ or missing fragments $f < p$. The tree is balanced, which means each node of line l has $p-l$ children, except for the last one line or two. Therefore, we have the following equations:

$$\boxed{f < p} \begin{cases} nodes(0) = 1; \\ \forall l \in [1\,..\,f], nodes(l) = (p-l+1) \times nodes(l-1); \\ nodes(f+1) = 1. \end{cases}$$

and:

$$\boxed{f = p} \begin{cases} nodes(0) = 1; \\ \forall l \in [1\,..\,f-1], nodes(l) = (p-l+1) \times nodes(l-1); \\ nodes(f) = nodes(f-1); \\ nodes(f+1) = 1. \end{cases}$$

We now calculate $|N|$:

$$\begin{cases} \boxed{f < p}\ |N| = 2 + \sum_{l=1}^{f} nodes(i); \\ \boxed{f = p}\ |N| = 2 + \sum_{l=1}^{f-1} nodes(i) + nodes(f-1). \end{cases}$$

In the case where $\boxed{f = p}$, we have:

$$|N| = 2 + \sum_{l=0}^{f-1} \prod_{k=l+1}^{f} k.$$

We calculate $|E|$:

$$|E| = |N| - 2 + nodes(f).$$

C.2 GRAPHS WITH EXTRA-FRAGMENTS

When the extra-fragments are enabled, the tree is no longer balanced and grows quickly at the right. Once again, we separate two cases: $\boxed{f \leq p}$ and $\boxed{f > p}$.

Once again, we calculate the number of node per line:

$$\boxed{f \leq p} \begin{cases} nodes(0) = 1; \\ \forall l \in [1\,..\,f], nodes(l) = \sum_{k=0}^{l-1} \binom{l-1}{k} \cdot A(k+1); \\ nodes(f+1) = 1; \end{cases}$$

and

$$\boxed{f > p} \begin{cases} nodes(0) = 1; \\ \forall l \in [1\,..\,p], nodes(l) = \sum_{k=0}^{l-1} \binom{l-1}{k} \cdot A(k+1); \\ \forall l \in [p+1\,..\,f], nodes(l) = \sum_{k=0}^{p} \binom{l-1}{k} \cdot A(k+1); \\ nodes(f+1) = 1; \end{cases}$$

where:

$$\begin{cases} A(1) = p+1; \\ \forall k \in [2\,..\,f], A(k) = (p-k+2) \cdot \prod_{i=p-k+2}^{p} i. \end{cases}$$

We calculate $|N|$:

$$\begin{cases} \boxed{f \leq p}\ |N| = 2 + \sum_{l=1}^{f} \sum_{k=0}^{l-1} \binom{l-1}{k} \cdot A(k+1); \\ \boxed{f > p}\ |N| = 2 + \sum_{l=1}^{p} \sum_{k=0}^{l-1} \binom{l-1}{k} \cdot A(k+1) + \sum_{l=p+1}^{f} \sum_{k=0}^{p} \binom{i}{k} \cdot A(k+1). \end{cases}$$

We calculate $|E|$:

$$|E| = |N| - 2 + nodes(f).$$

D

Alphazzle extended results

◂ Appendix C Appendix D ▸

Configuration					Validation accuracy	
fragment size (px)	fragment per side	Space size (px)	Hints training	Hints validation	P (%)	V (%)
40	3	0	0	0	69.56	90.07
40	3	4	0	0	69.91	88.46
40	3	10	0	0	67.39	87.15
40	3	20	0	0	61.34	85.79
40	3	4	0	4	79.43	95.51
40	3	4	0	8	99.29	97.93
40	3	4	1 (central)	1 (central)	73.99	92.34
40	3	4	2	2	71.82	93.35
40	3	4	4	0	65.52	86.36
40	3	4	4	4	79.98	95.77
40	3	4	4	8	99.74	95.48
40	3	4	6	6	87.50	97.53
40	3	4	8	0	29.35	70.96
40	3	4	8	4	42.59	82.11
40	3	4	8	8	99.49	88.45
40	4	4	0	0	37.65	92.64
40	4	4	4	4	48.54	98.24
40	4	4	8	8	52.02	99.09
40	4	4	12	12	72.53	99.04
40	4	20	0	0	39.67	90.47
40	5	4	0	0	19.15	94.25
40	5	4	10	10	23.79	99.50
40	6	4	0	0	3.43	64.47
96	3	48	0	0	56.55	80.54
96	3	48	1 (central)	1 (central)	60.89	73.08
96	3	48	2	2	67.24	88.00
96	3	48	4	4	75.30	90.02

Table D.1: Validation accuracy for P and V on various configurations.
▸ Return to §10.1.

Configuration					Validation accuracy	
Fragment size (px)	Fragment per side	Space size (px)	Hints training	Hints validation	P (%)	V (%)
40	3	4	0	0 fragment	46.97	50.00
40	3	4	0	1 fragment	51.96	69.35
40	3	4	0	2 fragments	56.25	85.78
40	3	4	0	3 fragments	57.71	88.91
40	3	4	0	4 fragments	63.45	90.57
40	3	4	0	5 fragments	69.10	92.64
40	3	4	0	6 fragments	77.37	96.02
40	3	4	0	7 fragments	85.33	96.97
40	3	4	0	8 fragments	99.49	99.29
40	3	4	0	9 fragments	—	96.77
40	3	4	4	0 fragment	39.61	50.00
40	3	4	4	1 fragment	46.98	70.97
40	3	4	4	2 fragments	52.77	86.49
40	3	4	4	3 fragments	58.62	88.71
40	3	4	4	4 fragments	63.61	91.33
40	3	4	4	5 fragments	70.46	93.19
40	3	4	4	6 fragments	76.46	96.37
40	3	4	4	7 fragments	84.53	97.88
40	3	4	4	8 fragments	99.65	98.94
40	3	4	4	9 fragments	—	98.23
40	3	4	8	0 fragment	11.34	50.00
40	3	4	8	1 fragment	11.99	50.00
40	3	4	8	2 fragments	11.74	52.47
40	3	4	8	3 fragments	16.78	57.86
40	3	4	8	4 fragments	17.79	66.73
40	3	4	8	5 fragments	23.18	75.30
40	3	4	8	6 fragments	30.79	79.99
40	3	4	8	7 fragments	49.54	82.66
40	3	4	8	8 fragments	99.39	86.59
40	3	4	8	9 fragments	—	91.23

Table D.2: Validation accuracy for all type of partial reassemblies.
▸ Return to §10.1.

Game meta-parameters		Reassembly scores		
NumSim	C	Fragment-wise (%)	Puzzle-wise (%)	Computation time (s per puzzle)
10	0.001	45.86	9	0.39
10	0.01	46.49	9	0.44
10	0.1	50.02	12	0.63
10	1	48.99	12	0.98
10	10	45.36	10	1.02
10	100	45.47	9	1.03
100	0.001	45.3	7	1.13
100	0.01	47.67	10	1.39
100	0.1	52.77	12	2.01
100	1	54.46	15	4.15
100	10	48.04	11	5.33
100	100	45.6	9	5.46
1000	0.001	47.11	10	7.30
1000	0.01	48.83	10	7.59
1000	0.1	55.03	14	8.97
1000	1	55.63	15	15.97
1000	10	52.23	13	25.11
1000	100	47.28	11	28.06
10000	0.001	49.43	9	63.11
10000	0.01	50.38	10	63.68
10000	0.1	58.09	17	69.78
10000	1	57.94	17	84.92
10000	10	55.07	15	116.92
10000	100	49.09	11	131.44
100000	0.001	50.57	12	620.69
100000	0.01	57.89	18	615.55
100000	0.1	61.01	21	643.01
100000	1	57.99	22	685.84
100000	10	56.97	17	766.22
100000	100	53.91	16	889.22
1000000	0.001	49.49	9	6545.73
1000000	0.01	67.68	27	6547.18
1000000	0.1	76.77	36	6547.00
1000000	1	65.56	30	7200.40
1000000	10	68.89	30	7202.50
1000000	100	53.33	10	7201.70

Table D.3: Grid search on the game meta-parameters.
▸ Return to §10.2.

Configuration					Reassembly scores		
Fragment size (px)	Fragment per side	Space size (px)	Hints P – V	Hints reassembly	Fragment-wise (%)	Puzzle-wise (%)	Qt of reassemblies
40	3	4	0-0	0	55.63	14.55	2000
40	3	4	0-0	1	60.94	20.65	2000
40	3	4	0-0	1 (central)	62.33	22.45	2000
40	3	10	0-0	0	48.59	7.15	2000
40	3	10	0-0	1	52.71	11.35	2000
40	3	10	0-0	1 (central)	52.99	11.15	2000
40	3	20	0-0	0	45.91	6.80	2000
40	3	20	0-0	1	50.24	10.85	2000
40	3	20	0-0	1 (central)	50.68	11.10	2000
40	4	4	0-0	0	29.08	0.00	407
40	4	4	0-0	1 (central)	32.47	0.16	528
40	4	4	0-0	8	50.01	5.40	639
40	4	20	0-0	0	23.42	0.00	440
40	4	20	0-0	8	41.60	0.00	2000
40	5	4	0-0	0	15.68	0.00	203
40	5	4	0-0	1 (central)	16.46	0.00	206
40	5	4	0-0	10	28.46	0.00	495
40	5	4	0-0	15	36.67	0.95	1363
40	5	4	10-10	10	32.10	0.00	493
40	5	4	10-10	15	42.67	1.20	1497
96	3	48	0-0	15	11.03	0.00	2000

Table D.4: Reassembly scores with 1,000 simulations and $C = 1$.
▸ Return to §10.3.

Configuration					Reassembly scores		
Fragment size (px)	Fragment per side	Space size (px)	Hints P – V	Hints reassembly	Fragment-wise (%)	Puzzle-wise (%)	Qt of reassemblies
40	3	20	0 0	0	48.38	7.69	52
40	3	20	0 0	1	51.35	4.62	65
40	3	20	0 0	1 (central)	44.62	4.60	65
96	3	48	0 0	0	51.36	11.32	53
96	3	48	0 0	1	50.56	14.93	67
96	3	48	0 0	1 (central)	58.40	19.40	67

Table D.5: Reassembly scores with 1,000,000 simulations and $C = 0.1$.
▸ Return to §10.3.