Puzzle Reassembly using Model Based Reinforcement Learning

(It is advisable that you use the more up-to-date web-version: https://johan-gras.github.io/projects/puzzlereassembly/)

Johan Gras
University of Cergy-Pontoise
johan.gras@outlook.com

Supervised by David Picard
and Marie-Morgane Paumard
david.picard@ensea.fr
marie-morgane.paumard@ensea.fr

Abstract

In this paper, we are interested in solving visual jigsaw puzzles in a context where we cannot rely on boundary information. Thus, we train a deep reinforcement learning model that iteratively select a new piece from the set of unused pieces and place it on the current resolution. The pieces selection and placement is based on deep visual features that are trained end-to-end. Our contributions are twofold: First, we show that combining a Monte Carlo Tree Search with the reinforcement learning allows the system to converge more easily in the case of supervised learning. Second, we tackle the case where we want to reassemble puzzles without having a groundtruth to perform supervised training. In that case, we propose an adversarial training of the value estimator and we show it can obtain results comparable to supervised training.

1. Introduction

In this paper, we focus on jigsaw puzzle reassembly. Our puzzles are made from 2D images and divided in 9 same-sized squared fragments, while also considering an erosion between each fragment. The problem consists in finding the optimal absolute position of each fragment on the reassembly, as show in Figure x. Moreover, we want to find this set of optimal positions using, on the one hand, an iterative process, and one the other hand, a content-based pattern matching method.

During the last decade, deep convolutional architectures as become the norm in most pattern matching tasks involving 2D images. Therefore, we choose to use such methods in order to extract semantic information from fragments and the reassembly. We build on the method proposed by Paumard et al. [11] that proposes to predict the relative position of a fragment with respect to another one, using a deep learning extractor and a bilinear feature aggregator.

In order to solve this problem, we construct this one as a Markov Decision Process, while solving the latter with a reinforcement learning (RL) algorithm. We build on the ExIt framework proposed by Anthony et al. [3] and the AlphaZero algorithm [12], that proposes to use model based RL in conjunction with deep learning to guide the model exploration.

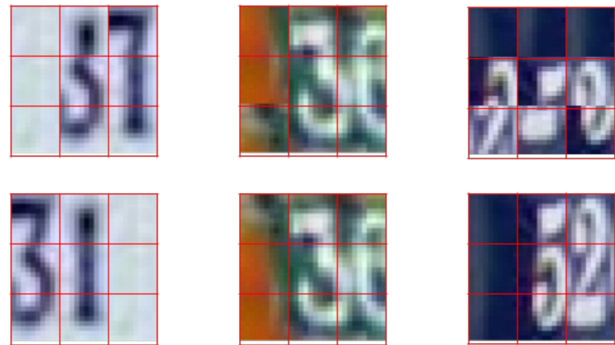

Figure 1. Pathological case puzzle reassembly on MNIST SVHN.Top row: reconstruction; bottom row: perfect image.

However, such methods enforce the presence of the ground truth to learn the semantic of our problem. Alternatively, we replace the state value approximator trained on the ground truth, by an adversarial discriminator trying to predict the realness of one puzzle reassembly. We show that using the discriminator's method not only provided similar accuracy compared to our value-based method, but also allow for faster convergence.

Our contributions are the following. First, a model based RL technique using a Monte Carlos Tree Search to compute simulations and two Neural Networks guiding the search and evaluating each final search's trajectory. Second, a discriminator based method using deep learning to predict each fragment's position probability. Third, the merging of the two methods, using the discriminator network instead of the initial state value approximator.

This paper is organized as follows: in section 2, we present related work on puzzle solving and fragment re-

assembly as well as relevant literature on iterative problem solving methods. Next, we detail the core of our value-based method : the optimization problem, the path simulation and the architecture of our neural networks. In section 4, we introduce the discriminator base method and we explain the merging of our methods. Then, we present our experimental setups and analyze the results obtained for different methods and datasets.

2. Literature review

Our project was inspired by several research articles which aim at solving this very problem using such techniques as deep learning. Other articles were unrelated to our purpose, but we adapted techniques described in there for our project purpose (such as Alpha Zero).

2.1. Review of puzzle solving methods

Most publications of this field [4, 1] rely on the border irregularities and aim for precise alignment. These methods perform well on a small dataset with only one source of fragments. However, they are fragile towards erosion and fragment loss.

Without being interested in jigsaw puzzle solving, Doersch et al. proposed a deep neural network to predict the relative position of two adjacent fragments in [6]. The authors show their proposed task outperforms all other unsupervised pretraining methods. Based on [6], Noroozi and Favaro [9] introduce a network that compares all the nine tiles at the same time. Those papers do not focus on solving a jigsaw puzzle but, on understanding the spatial structure of images by building generic images features.

However, Paumard et al. in [11] and [10] directly attempt to solve those kind of problems with the use of deep learning and based on the method proposed in [6]. Moreover, they bring two significant innovations with their method. First, they consider the correlations between localized parts of the fragments using the Kronecker Product of the feature vectors. Thus, directly using the spacial correlation allow for faster convergence than the concatenation of the feature vectors used in [6]. Additionally, they look for a complete fragment reassembly which they performed using the neural network predictions to build a shortest path graph problem. Therefore, achieving a significant accuracy increase over a pure greedy policy. Taken together, they obtained results that outperformed the previous state of the art in jigsaw puzzle solving.

Their method works in the following way (see 2) :

- First, they use deep learning in order to extract important features of each image fragments.
- Then, they compare each couple of fragments features to predict the relative position of the two tiles, through a classifier.

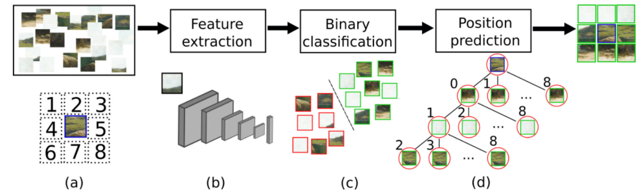

Figure 2. Overview of the method. Knowing a central fragment, we are looking for the correct arrangement to reassemble the image.

- Finally, they solve the best reassembly by computing the shortest path problem given the relative positions of each couples provided.

More specifically, the model perform quite well when we give it only the 9 accurate fragments. However, if we try to delete or to add outsider fragments to the puzzle, the accuracy decrease strongly. Furthermore, the increase of computation time is reasonable as long as the puzzle still contains 9 pieces, but any increment of the number of pieces leads to an factorial increase of the number of solution. Thus, the reassembly problem is NP-hard and the problem become quickly intractable.

2.2. MCTS based methods

As we seen, STOA methods like [10] suffer indeed from serious scaling issues. In this research project, we are looking to solve the scaling problem of the jigsaw puzzle reassembly. In particular, we are looking to scale both in terms of the size of the puzzle and the size of the fragments ensemble.

More specifically, we are looking to use a similar approch as [6] and [11] in order to extract vector features from our images and then to compute the position likelihood of each fragments. On the other hand, instead of solving the shortest path problem using an expensive optimal algorithm like Disjtra in [10], we want to tackle this by using an heuristic search based algorithm, thus speeding up the process of evaluating the shortest path in our graph. Thus, we will be able to compute larger graphs in a reasonable time frame.

Here we propose to used a deep reinforcement learning algorithm such as Alpha Zero [12] in order to learn the heuristic function only from self-play. AlphaZero is a more generalized variant of the AlphaGo Zero [2] algorithm, that accommodates, without special casing, a broader class of game rules (chess and shogi, as well as Go). Their results demonstrate that a general-purpose reinforcement learning algorithm can learn, tabula rasa (without domain-specific human knowledge or data) superhuman performance across multiple challenging games.

Instead of a handcrafted evaluation function and move-

ordering heuristics, AlphaZero uses a deep neural network. This neural network takes the board position as an input and outputs a vector of move probabilities and a scalar value estimating the expected outcome of the game from position. AlphaZero learns these move probabilities and value estimates entirely from self-play; these are then used to guide its search in future games. Additionally, AlphaZero uses a general purpose Monte Carlo tree search (MCTS) algorithm [5]. Each search consists of a series of simulated games of self-play that traverse a tree from root state until a leaf state is reached.

3. Puzzle reassembly using model based RL

In this section, we detailed our first proposed method. This is a model based reinforcement learning technique, which uses a Monte Carlos Tree Search to compute simulations and two Deep Neural Networks that work as Value Function and Advantage Function approximators.

3.1. Problem formulation and formalism

From a set of fragments coming from the same image, we want to solve their best reassembly by assigning each fragment to its optimal position.

3.1.1 Data representation

We want to introduce the notation of our data. n is the number of fragments of a puzzle. f_i is a flatten representation of the ith fragment given. f_{empty} is a special fragment filled with zeros. F is a puzzle reassembly (a n-sized vector of fragments). F^* is the optimal puzzle reassembly.

3.1.2 Goal, MDP and objective maximisation

Our goal being to find the optimal puzzle reassembly F^*. We want to consider this problem as a Markov Decision Process (MDP) in order to solve it with a reinforcement learning algorithm.

Starting from an initial state s_0, we want to assign each fragment f_i to its optimal position p by taking the n optimal actions a_i^*. More specifically, we want to find one optimal policy π^* (7) in order to find those optimal actions.

We define the notation part of the MDP formalism.

s_t is the state at step t defined by the tuple (F_t, R_t). With F_t the puzzle reassembly at step t. R_t an n-sized vector, f_{empty} padded, of every fragment not used in F_t.

The initial state s_0 from the start-state distribution (1) is composed of F_0 an empty puzzle reassembly and R_0 n-sized vector of fragments f_i. In fact, this start-state distribution is due to the uniform sample of a puzzle from the dataset.

$$s_0 \sim \rho_0(.) \tag{1}$$

An action is characterized by a **fragment-position pair** $a_t = (f_i, p)$. With f_i a fragment from R_t and p a unassigned position of the reassembly F_t.

An action a_t **may be applied** to a state s_t using a deterministic transition function (2), creating the **new state** s_{t+1}. Thus, this transition function f dictate the law of our synthetic environment.

$$s_{t+1} = f(s_t, a_t) \tag{2}$$

$\{a_i^*\}$ is the set of the n optimal actions to take in order to solve the puzzle. In fact, this is due to the particularity of of our problem. No matter in what order the actions are applied the final reassembly F_n will be identical.

A trajectory τ, is a sequence of states and actions $(s_0, a_0, s_1, a_1, ..., a_{n-1}, s_n)$.

r_t (3), the reward for taking the action a_t at state s_t.

$$r_t = R(s_t, a_t) \tag{3}$$

$$r_t = \begin{cases} \frac{1}{n} & \text{if } a_t \in \{a_i^*\} \\ 0 & \text{if } a_t \notin \{a_i^*\} \end{cases} \tag{4}$$

$R(\tau)$ (5), the finite-horizon undiscounted return of a trajectory τ.

$$R(\tau) = \sum_{t=0}^{n} r_t \tag{5}$$

$\pi(s_t)$ (6) is a policy that dictate witch action an agent should take in a given state.

$$a_t = \pi(s_t) \tag{6}$$

Our central optimization problem (7) is to optimize the expected return of our policy π (8). With π^* being the optimal policy.

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{7}$$

$$J(\pi) = \sum_{\tau} P(\tau|\pi) R(\tau) = \mathop{\mathbf{E}}_{\tau \sim \pi}[R(\tau)] \tag{8}$$

3.2. The ExIt framework

Human reasoning consists of two different kinds of thinking. When learning to complete a challenging planning task, such as solving a puzzle, humans exploit both processes: strong intuitions allow for more effective analytic reasoning by rapidly testing promising actions. Repeated deep study gradually improves intuitions. Stronger intuitions feedback to stronger analysis, creating a closed

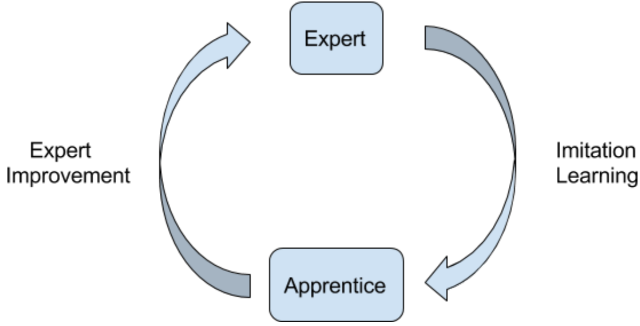

Figure 3. Expert Iteration framework.

learning loop. In other words, humans learn by thinking fast and slow.

Expert Iteration (ExIt) 3 is a general framework for learning that can result in powerful AI machines, without needing to mimic human strategies. ExIt can be viewed as an extension of Imitation Learning methods to domains where the best known experts are unable to achieve satisfactory performance. In standard IL an apprentice is trained to imitate the behaviour of an expert. In ExIt, between each iteration, an Expert Improvement step is perform, where the apprentice policy is bootstrap to increase the performance of the expert.

3.3. Overview

The goal of this method is to find the optimal policy (7) able to solve any puzzles 4 from a specific dataset. In order, to achieve this goal we need to train a Reinforcement Learning (RL) algorithm able to find such policy.

We use a model based RL algorithm instead of less sample efficient algorithm such as Policy Gradients or DQN. In particular, we use a Monte Carlos Tree Search to compute simulation in our synthetic environment and those simulations allow our algorithm for a slow-thinking phase instead of pure intuition. The MCTS is a strong playing strategy and is acting as the expert in the ExIt framework, with π_{MCTS} its policy.

A first neural network, the Policy Network (PN), act as the apprentice and try to mimic the expert policy π_{MCTS}. In other words, we can consider that this PN is acting as an Advantage Function (9) approximator, with π_{PN} being the apprentice policy.

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{9}$$

A second neural network, the Value Network (VN), act as a Final Value Function (10) approximator. This VN approximate the finite-horizon return of the trajectory τ used to be in the final state s_f. Therefore, it evaluates the quality of a complete reassembly.

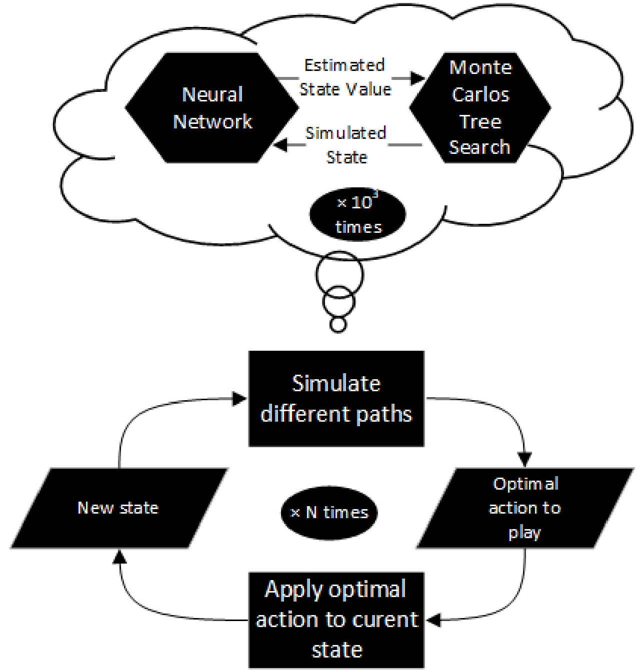

Figure 4. Workflow of a puzzle solving. Knowing all the fragments of a puzzle, we are looking for the optimal assignment of each fragment. First, those fragments constitute the initial state. Second, a "dreaming phase" occurs with the MCTS simulating multiple trajectories (of paths) and the networks estimating their action-state values. After its dream, the MCTS should give us a strong estimated of the optimal action. Third, the transition function apply the optimal action on the actual state, giving us a new state. Again, this process is repeated until all fragments are assigned.

$$V_f(s_f) = R(\tau) \tag{10}$$

During the Expert Improvement phase 3 we use the PN to direct the MCTS toward promising moves, while effectively reducing the branching factor. In this way, we bootstrap the knowledge acquired by Imitation Learning back into the planning algorithm. In addition, after each simulation we backpropagate finals state value estimated by the VN. Indeed, this let our algorithm solve puzzles without having access to the ground truth after training.

3.4. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an any-time best-first tree-search algorithm. It uses repeated game simulations to estimate the value of states, and expands the tree further in more promising lines. When all simulations are complete, the most explored move is taken.

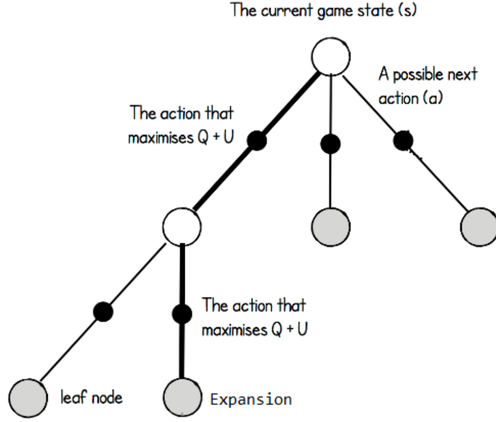

Figure 5. Monte Carlo Tree Search structure. The simulation start in the current game state, then select the action maximising $Q+U$ until a leaf node is reach, to finally expand the leaf node.

3.4.1 The search

Each search, consist of a series of simulated games by the MCTS that traverse a tree from root state until a final state is reached. Each simulation proceeds in four parts.

First, a tree phase where the tree is traversed by taking actions according to a tree policy 5. Second, an expansion phase where child nodes are created from legal moves. Third, a rollout phase, where some default policy is followed until the simulation reaches a final state. Finally, the backpropagation of the final state value on its final parents nodes. We describe here the MCTS algorithm used in our method; further details can be found in the pseudocode 1.

The MCTS is a tree search where each node corresponds to a state-action pair (s, a) composed of a set of statistics, $\{N(s,a), W(s,a), Q(s,a), P(s,a)\}$, with $N(s,a)$ the visit count, $W(s,a)$ the total action-value, $Q(s,a)$ the mean action-value, and $P(s,a)$ the prior probability of selecting a in s.

During the tree phase, actions are selected by choosing in in each state s the action a that maximise the UCB formula (11). $Q(s,a)$ is the mean value of the next state and $U(s,a)$ (12) a function that increases if an action hasn't been explored much or if the prior probability of the action is high.

$$UCB(s,a) = Q(s,a) + U(s,a) \tag{11}$$

$$U(s,a) = C_{PUCT} P(s,a) \frac{\sqrt{N(s)}}{1 + N(s,a)} \tag{12}$$

Once a leaf node s_L is reach we enter in the expansion phase. We use the PN (13) to obtain a prior prediction P over the action space by evaluating the leaf node. Then, we expand the tree by initialising each legal child node with a softmax normalisation (14), such that a and i are legals actions. After, we choose once again the action maximising the UCB formula (11).

$$P = PN_{\theta}(s_L) \tag{13}$$

$$p(s_L, a) = \frac{\exp(P_a)}{\sum_i exp(P_i)} \tag{14}$$

Then, the rollout phase begin, where the default policy used is simply to choose actions uniformly from the legal ones. This random policy is followed until we reach a terminal state.

Finally, when the rollout is complete, we compute $\hat{v}$ an approximation of the Final Value Function (10) with the VN (15). This estimated reward signal is propagated through the tree. Thus, all the parents visits counts and total action-value are updated in a backward pass.

$$\hat{v} = VN_{\theta}(F_f) \tag{15}$$

At the end of the search, the action is return in function of its exploration during the search N. Two policies are used by the MCTS, one for competitive play and one for exploratory play.

3.4.2 Exploration versus exploitation trade-off

On-line decision making involves a fundamental choice; exploration, where we gather more information that might lead us to better decisions in the future or exploitation, where we make the best decision given current information in order to optimize our present reward. This exploration-exploitation trade-off comes up because we're learning online. We're gathering data as we go, and the actions that we take affects the data that we see, and so sometimes it's worth to take different actions to get new data.

In order to explore our synthetic environment and not only acting greedily, we use three kinds of exploration processes : an Upper Confident Bound (UCB), a Dirichlet noise onto the root's prior and a softmax sample over the action's visit count to select a final action. The first one is applied both during the training and the evaluation phase, while the other two, are only applied during the exploratory phase.

To select a move during the simulation phase, we use the UCB score (11). Thus, we do not only act greedily over the mean estimation of an action Q, but we also estimate an upper confidence U, of what we think the mean could be. The UCB allow to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how their estimates are to being maximal and the uncertainties in those estimates.

At the start of each search, we add a Dirichlet noise (16) onto the root's prior (17), to encourage the search to explore new actions. α is the parameter of the symmetric Dirichlet distribution and β the parameter of the linear interpolation between the original prior value and a sample of the distribution.

$$X \sim Dir(\alpha) \tag{16}$$

$$p(s_{root}, a) = p(s_{root}, a)(1 - \beta) + \beta X \tag{17}$$

The search return the visit-count vector of the root's moves, $N_a = N(s_{root}, a)$. The move to play is either select stochastically (19), for exploration or deterministically (20), for competitive play. During training, the move is sample from the probability distribution over moves (18) and in exploitation, greedily in respect to the visit counts.

$$\pi_a = Pr(a) = \frac{\exp(N_a)}{\sum_i exp(N_i)} \tag{18}$$

$$a \sim \pi \tag{19}$$

$$a = \arg\max_a N_a \tag{20}$$

3.4.3 The MCTS pseudocode

Listing 1. Modified MCTS algorithm.

```
def mcts(game, policy_net, value_net):
  # Initialize root node
  root = Node()
  policy_net.expand(root, game)
  add_exploration_noise(root)

  for s in range(num_simulations):
    # Start a new simulation
    simu_game = game.clone()
    search_path = [root]
    action, node = select_child(root)
    simu_game.apply(action)
    search_path.append(node)

    # Tree policy
    while not simu_game.finished()
          and node.expanded():
      action, node = select_child(node)
      simu_game.apply(action)
      search_path.append(node)

    # Expansion
    if not simu_game.finished():
      policy_net.expand(node, simu_game)
      action, node = select_child(node)
      simu_game.apply(action)
      search_path.append(node)

    # Rollout phase
    while not simu_game.finished():
      action = random_action(simu_game)
      simu_game.apply(action)

    # Backpropagate the terminal value
    value = value_net.eval(simu_game)
    backpropagate(search_path, value)

  # Return the MCTS policy's action
  action = policy_mcts(game, root)
  return action
```

3.5. Functions approximators

In this method we use two Neural Networks working as functions approximators, the Policy Network (PN) and the Value Network (VN). In particular, the PN compute an estimation of the Advantage Function (9), learning the expert policy π_{MCTS}. Whereas, the VN estimate the Final Value Function (10), learning from the ground truth.

Value Network architecture The non-domain specific architecture of the VN is described in Figure 6. We use a problem dependent neural network architecture $\hat{v} = VN_\theta(F_f)$ with parameters θ. This neural network takes the final reassembly F_f as input and output a scalar value $\hat{v}$ estimating the finite-horizon return of the state's trajectory.

Policy Network architecture The global PN architecture is described in Figure 7. The neural network $P = PN_\theta(s_L)$ takes a state s as input and outputs a move probability P with components $P_a = Pr(a|s)$ for each action a.

More specifically, the PN takes F and R the couple composing the state s as separated inputs. On the one hand, the reassembly representation is extracted using a first Feature Extraction Network (FEN1). On the other hand, each individual fragment representation of R is extracted using a second Feature Extraction Network (FEN2), with shared weights. Both FEN architecture is either, depending on the

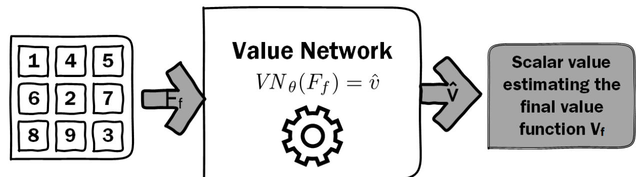

Figure 6. Value network architecture.

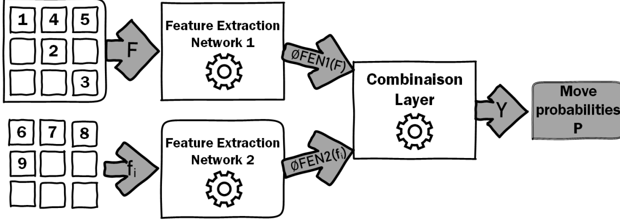

Figure 7. Policy network architecture.

image complexity: a fully connected network or a deep convolutional architecture followed by a fully connected classifier.

The features of each FEN are then combined through a Combination Layer (CL). We use a bilinear product in order to optimally capture the spacial covariances among the features. In particular, we use this bilinear layer (21), with $\varnothing FEN1(f_i)$ and $\varnothing FEN2(F)$ the output of the first and second FEN, respectively inside the ensemble $\mathbb{R}^D$ and $\mathbb{R}^{D'}$. W is a learnable tensor, such that $W = (w_{p,d,d'})_{p \in P, d \in D, d' \in D'}$, with P the number of positions in the puzzle. Y is the output of the CL, such that $Y = (y_{i,p})_{i \in N, p \in P}$.

$$Y_{i,p} = \varnothing FEN2(f_i)^T W_p \varnothing FEN1(F) \tag{21}$$

Finally, we flatten the output Y of the CL such as $Y'_a = Y_{i,p}$, with $a = (i,p)$ an action and we apply a softmax normalization (22) to get the move probabilities P.

$$P_a = Pr(a) = \frac{\exp(Y'_a)}{\sum_i exp(Y'_i)} \tag{22}$$

Training Our algorithm learns these move probabilities and value estimates entirely from self-play; these are then used to guide its search for future reassembly.

At the end of each reassembly, the final reassembly is scored according to the cumulative reward of the environment $R(\tau)$. The VN parameters are updated to minimize the error between the predicted outcome $VN_\theta(F_f)$ and the environment outcome. While the PN parameters are updated to maximize the similarity of the move probabilities vectors P to the search probabilities π (18). Specifically, the parameters are adjusted by gradient descent with a mean-squared error for the VN loss and a cross-entropy for the PN loss.

4. Puzzle reassembly using RL with discriminative value estimation

In this section, we introduce briefly the discriminator based method, before focusing on our merged method, combining this last method and the RL's describes in section 3.

4.1. Discriminative value estimation

This method can be seen as a mixture between an Actor-Critic [8] and a Generative Adversarial Network [7] methods. The actor is also a generator and try to reassemble the puzzle. The critic is also a discriminator and try to predict the reassembly realness.

The discriminator is trying to make the difference between real images and reassembly images coming from the generator. Then, the prediction of the discriminator is used to train the generator. If the discriminator's prediction is good, actions chosen by the generator are reinforced. Otherwise, if the prediction is low actions chosen are negated.

This discriminative-based technique is very efficient because it can be trained without having access to the ground truth. For more details on this method, refer to Loïc's paper: Puzzle reassembly using deep reinforcement learning with an adversarial model.

4.2. Merged method

We merged our two iterative puzzle reassembly methods into one. We still use the model base RL algorithm, but instead of using the original Value Network (15) from the first method, we use the discriminator from the last subsection. Therefore, instead of estimating the cumulative reward of the trajectory, we try to estimate the realness of the reassembly. The network architecture of the discriminator is the same as the VN architecture.

The goal of this merged version is to keep the best of both worlds. On the one hand, the reassembly look-ahead of the Monte Carlo Tree Search, on the other hand, the "unsupervised" setup of the discriminator value.

5. Experiments

We test our methods for different kinds of datasets with different setup. First, we present the results of the MNIST dataset, second we consider an erosion of the MNIST images and finally on the MNIST SVHN dataset.

Two metrics are used in order to evaluate our methods. On the one hand, the fragments accuracy is the percentage of perfectly position fragments, on the other hand, the perfect reconstruction is a pixel-wise difference with the original image with a 10-pixel threshold.

5.1. MNIST

We tested on the MNIST dataset 8 because the images are small enough for us to train the complete network and search for the best hyper-parameters.

In figure 9 we can see that the increase in the number of simulation increase logarithmically the accuracy. Also, both the classic method and the merged method give similar results, however, if the number of simulations is too low the merged method's accuracy is dropping. It may seem

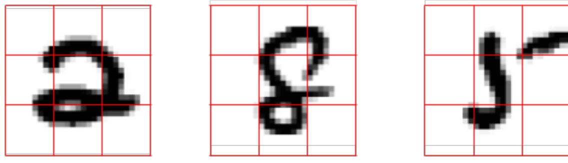

Figure 8. Mnist Puzzles examples. Our puzzles are made from 2D images and divided in 9 same-sized squared fragments.

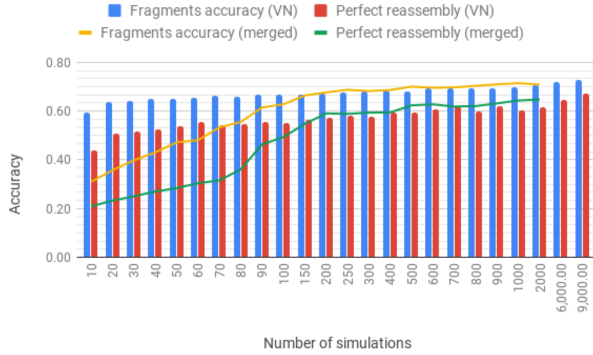

Figure 9. Accuracy of the two methods in test-time, with different numbers of simulations in the MCTS.

surprising that the merged method works so well without having access to the ground truth during the training phase.

In figure 10 and 11, we can observe the evolution of accuracy thought the training of both methods. The training time of the merged method is 5 times smaller than the classic method. However, the correlation of the Value Network to the fragments accuracy is a lot stronger than the Discriminator Network. We train the VN to directly estimate this value so this not a surprise.

With eroded fragments Test of the merged method on the MNIST dataset with erosion between the fragments 13. The test time result is really good 14, especially on the perfect reassembly metrics. During the training 15 we observe both accuracy being more or less equal, the correlation coefficient is not in the best shape ever.

5.2. MNIST SVHN

After playing with MNIST, we wanted to go further on the image complexity. Therefore we try our methods on MNIST SVHN , we show in 16 some promising reconstruction. On this dataset, with a really short training time 18, we achieve high fragments accuracy 17.

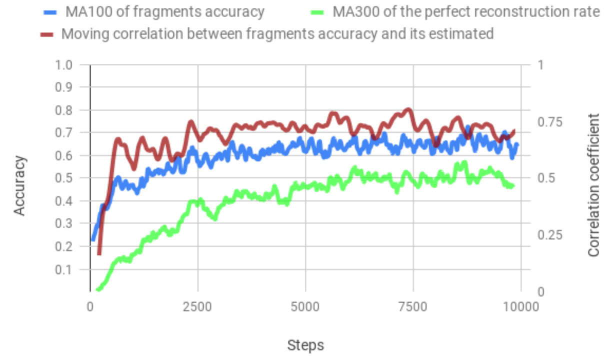

Figure 10. Evolution of puzzle reconstruction accuracy through training for the classic method. The coefficient correlation is calculated on the last 100 values, then a MA10 is used to smooth the curve. MA10 : Moving Average on 10 steps.

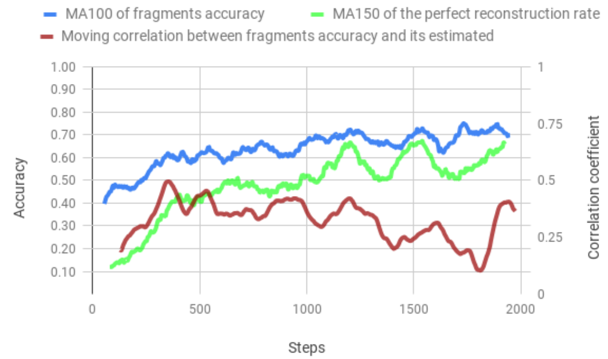

Figure 11. Evolution of puzzle reconstruction accuracy through training for the merged method.

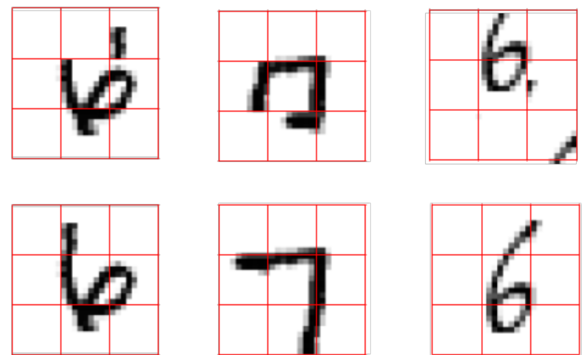

Figure 12. Pathological case puzzle reassembly on MNIST. Top row: reconstruction; bottom row: perfect image.

References

[1] *3D puzzle reconstruction for archeological fragments*, vol-

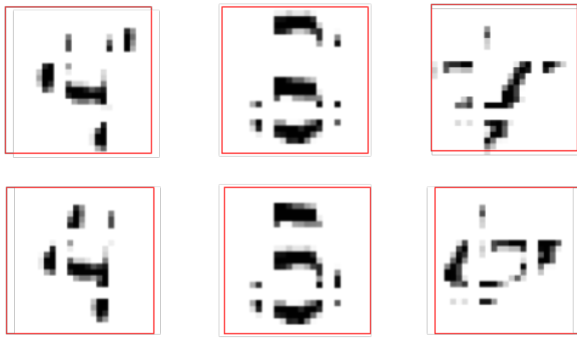

Figure 13. Pathological case puzzle reassembly on MNIST with erosion. Top row: reconstruction; bottom row: perfect image.

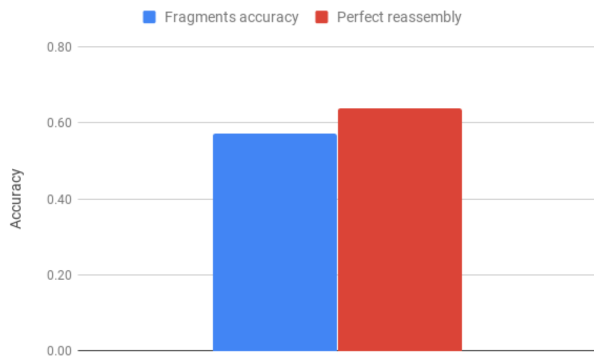

Figure 14. Accuracy of the merged method in test-time.

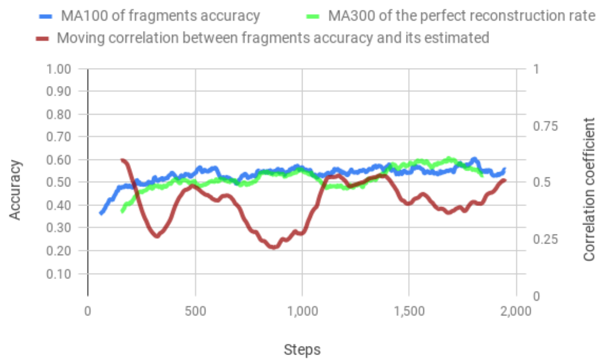

Figure 15. Evolution of puzzle reconstruction accuracy through training for the merged method on MNIST with erosion.

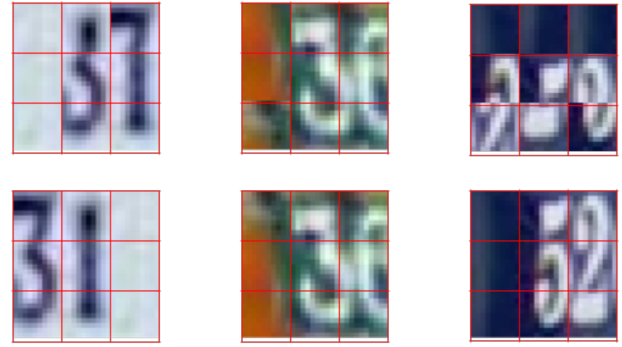

Figure 16. Pathological case puzzle reassembly on MNIST SVHN. Top row: reconstruction; bottom row: perfect image.

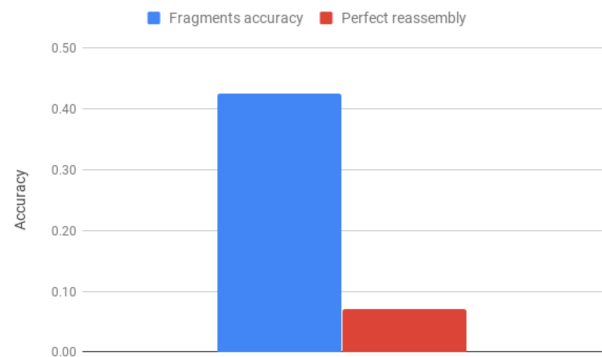

Figure 17. Accuracy of the classic method in test-time.

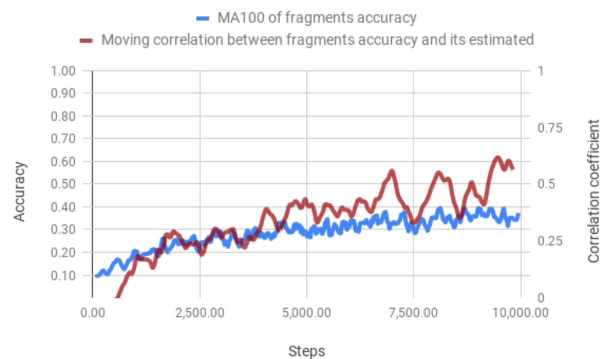

Figure 18. Evolution of puzzle reconstruction accuracy through training for the classic method on MNIST SVHN.

ume 9393, 2015. 2

[2] Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. 2

[3] T. Anthony, Z. Tian, and D. Barber. Thinking fast and slow with deep learning and tree search. *CoRR*, abs/1705.08439, 2017. 1

[4] J. C. McBride and B. B. Kimia. Archaeological fragment reconstruction using curve-matching. volume 1, pages 3 – 3, 07 2003. 2

[5] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008. 3

[6] C. Doersch, A. Gupta, and A. A. Efros. Unsupervised visual representation learning by context prediction. In *International Conference on Computer Vision (ICCV)*, 2015. 2

[7] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. 7

[8] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000. 7

[9] M. Noroozi and P. Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. *CoRR*, abs/1603.09246, 2016. 2

[10] M. Paumard, D. Picard, and H. Tabia. Image reassembly combining deep learning and shortest path problem. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VI*, volume 11210 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2018. 2

[11] M.-M. Paumard, D. Picard, and H. Tabia. Jigsaw Puzzle Solving Using Local Feature Co-occurrences In Deep Neural Networks. In *International Conference on Image Processing*, Athens, Greece, Oct. 2018. 1, 2

[12] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. 1, 2