

Appendix

A Extended results

In Section 4, we compare MEMENTO to popular methods from the literature in numerous settings. In particular, Table 1 compares using stochastic sampling, MEMENTO and EAS to adapt POMO on the standard benchmark; and Figure 6 compares the relative performances of MEMENTO and EAS on larger instances of TSP and CVRP for several values of batch sizes and budget. In this section, we report additional values and results of other methods for those experiments.

A.1 Standard benchmark

We evaluate our method, MEMENTO, against leading RL methods and industrial solvers. For RL specific methods, we provide results for POMO with greedy action selection and stochastic sampling, and EAS; an active search RL method built on top of POMO, that fine-tunes a policy on each problem instance. We also report population-based method COMPASS and the zero-shot combination of COMPASS with MEMENTO.

We compare to the heuristic solver LKH3 (Helsgaun, 2017); the current leading industrial solver of both TSP and CVRP, as well as an exact solver Concorde (Applegate et al., 2006) which is a TSP-specific industrial solver, and the CVRP-specific solver (Vidal et al., 2012).

We use datasets of 10,000 instances with 100 cities/customer nodes drawn from the training distribution, and three generalization datasets of 1,000 instances of sizes 125, 150, and 200, all from benchmark sets frequently used in the literature (Kool et al., 2019; Kwon et al., 2020; Hottung et al., 2022; Grinsztajn et al., 2023; Chalumeau et al., 2023b). Table 5 displays results for TSP and CVRP on the standard benchmark. The columns provide the absolute tour length, the optimality gap, and the total inference time that each method takes to solve one instance within the attempts budget.

Standard benchmark with augmentation trick Augmentation with symmetries is a problem-specific trick that can only be used for a few CO problems. Most prior work assumes that this additional x8 batching can be achieved seamlessly, which is unlikely in practice, when simulating complex real-world scenarios. Using this trick means decreasing the room for search and adaptation, since 87.5% of the budget is consumed to squeeze performance through uncontrolled network variance, rather than letting methods use principled strategies. Nevertheless, we report the standard benchmark with the "augmentation trick" in Table 6. It can be observed that MEMENTO outperforms EAS in distribution (CVRP 100), and for the large instances task (CVRP 200). EAS leads on the two other tasks. MEMENTO leads the augmented benchmark overall: it consistently leads in distribution, and both methods are competing closely out-of-distribution. It is to be noted that we have not tuned MEMENTO's hyper-parameters for these runs.

Times reported When possible, we decide to report the time to solve one instance rather the entire dataset following four main observations: (i) First, the literature use datasets of varying sizes, e.g. 10k for TSP100, 1k for TSP200, 128 for TSP500, hence time reported can be confusing, and do not enable to clearly see how methods' solving time scale with instance size. (ii) Second, the default real-world application consist in solving one instance at a time, or solving several instances at the same time but on separated hardware. (iii) Additionally, measuring on the entire dataset mixes several aspects, i.e. the instance solving time is mixed with the batch scalability of the method. We do think that this property is very interesting to know, but should be considered on the side, rather than mixed with the instance solving time. (iv) Moreover, this property will express differently depending on the hardware available. For instance, on a small hardware POMO sampling with n attempts will be n times slower than POMO greedy; but given a large enough GPU, POMO sampling can be parallelised n times and hence take exactly the same amount of time as POMO greedy.

We were able to do so on the standard benchmark since we have implementations of most methods we were comparing in a single framework. Since we are reporting several external methods in Table 8, we could only report time taken for the full dataset in that case.

Performance with runtime as budget Expressing budget as time taken to solve one instance is subject to high biases but it gives interesting perspectives on the time analysis and effects of

parallelism. In Table 7, we provide results of using runtime in stead of attempts as budget given to each methods to solve TSP and CVRP instances. We use EAS runtime as the reference time. These results may only be considered moderately since they are subject to a number of limitations: (i) different labs use different implementations and frameworks (ii) labs have access to different hardwares (iii) time is sensitive to parallelism, whereas number of attempts is not. Hence, comparing methods with time is subject to a higher number of biases, and would make it almost impossible to compare papers without a common codebase and hardware. We observe that MEMENTO still leads the benchmark, with 6 out of 8 top results; and both methods are very close on the two tasks where EAS leads.

Additional comments about the results On Table 5, we can see that in the single-agent setting, MEMENTO leads the entire benchmark, and in the population-based setting, MEMENTO (COMPASS) is leading the whole benchmark. MEMENTO is able to give significant improvement to COMPASS on CVRP, pushing significantly the state-of-the-art on this benchmark. On TSP, the improvement is not significant enough to be visible on the rounded results. To finish with, we can observe that the time cost associated with MEMENTO is reasonable and is worth the performance improvement (except maybe for TSP results of MEMENTO (COMPASS)).

Notes concerning SGBS All neural methods reported in Table 5 are our own runs with standardised checkpoints and rollout strategies, except for the results of SGBS, which were taken from Choo et al. (2022). This introduces three biases: (i) the base POMO checkpoint used by SGBS is not exactly the same as our re-trained POMO checkpoint (ii) SGBS uses the domain-specific augmentation trick that we do not use (iii) SGBS pre-selects starting points in CVRP, which we do not do.

Overall, the results reported in SGBS show that SGBS alone is always outperformed by EAS; hence, it should be expected that in all the settings where we outperform EAS, we would significantly outperform SGBS if both methods were evaluated exactly in the same way. We hence expect the gap between MEMENTO and SGBS to be larger than the one reported here. Additionally, SGBS has yet never been validated on larger instances. Nevertheless, we think that SGBS is a very efficient method from the NCO toolbox and appreciate that MEMENTO and SGBS are orthogonal, and could be combined for further improvement. We leave this for future work.

A.2 Performance over different number of parallel and sequential attempts

In Section 4.3, we show performance improvement of MEMENTO over EAS on instances of size 500 on TSP and CVRP for increasing number of sequential attempts and size of attempt batch. We report extended results with an additional competitor, POMO. We compare the online adaptation of the methods over four different sizes of batched solutions across increasing sequential attempts for each CO problem. Results from Table 8 show results of the three methods on TSP and CVRP. The columns show the best tour length performance for various values of sequential attempts expressed as budget, and solution batches of size N . Performance is averaged over a set of 128 instances.

A.3 Evaluation over larger instances

In Section 4.3, we evaluate MEMENTO and baselines on instances of size 500. For TSP, we use the dataset from Fu et al. (2021). For CVRP, we use the dataset from Luo et al. (2023). We do not include LEHD in our results since we focus on methods trained with Reinforcement Learning, and LEHD can only be successfully trained with supervised learning at the time of writing. Nevertheless, the good performance achieved by LEHD has motivated us to study the scaling law of MEMENTO as the number of layers in the decoder increases, reported on Appendix A.4.

In order to run COMPASS and MEMENTO (COMPASS) with the same batch sizes as other methods on those large instances, we reduced the number of starting points used by COMPASS to ensure that this number multiplied by the number of latent vector sampled at the same time is equal to the number of starting point used by other methods (POMO, EAS, MEMENTO). In practice, we used a latent vector batch of size 10, and hence divided the number of starting points by 10. This is slightly disadvantaging COMPASS and MEMENTO (COMPASS) but enables to respect the constraint of number of parallel batches that can be achieved at once. Note that this slightly impacts the time performance reported since JAX jitting process will not fuse the operations in the same way, additionally, when combined with MEMENTO, this impact the size of the memory (we keep one per starting point to

Table 5: Results of MEMENTO against the baseline algorithms for (a) TSP and (b) CVRP. The methods are evaluated on instances from training distribution ($n = 100$) as well as on larger instance sizes to test generalization. We use the same dataset as the rest of the literature, those contain 10 000 instances for $n = 100$ and 1000 instances for $n = 125, 150, 200$. Gaps are computed relative to LKH3. We report time needed to solve one instance. SGBS * results are reported from Choo et al. (2022), and do not use the same POMO checkpoint as other reported results. Additionally, they rely on problem-specific tricks that were not used by other methods. Details in Appendix A.1.

(a) TSP

Method	Training distr. $n = 100$			$n = 125$			Generalization $n = 150$			$n = 200$		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
Concorde	7.765	0.000%	0.49S	8.583	0.000%	0.72S	9.346	0.000%	1S	10.687	0.000%	1.86S
LKH3	7.765	0.000%	2.9S	8.583	0.000%	4.4S	9.346	0.000%	6S	10.687	0.000%	11S
POMO (greedy)	7.796	0.404%	0.16S	8.635	0.607%	0.2S	9.440	1.001%	0.29S	10.933	2.300%	0.45S
POMO (sampling)	7.779	0.185%	16S	8.609	0.299%	20S	9.401	0.585%	29S	10.956	2.513%	45S
SGBS*	7.769*	0.058%*	-	-	-	-	9.367*	0.220%*	-	10.753*	0.619%*	-
EAS	7.778	0.161%	39S	8.604	0.238%	46S	9.380	0.363%	64S	10.759	0.672%	91S
MEMENTO (POMO)	7.768	0.045%	43S	8.592	0.109%	52S	9.365	0.202%	77S	10.758	0.664%	115S
COMPASS	7.765	0.008%	20S	8.586	0.036%	24S	9.354	0.078%	33S	10.724	0.349%	49S
MEMENTO (COMPASS)	7.765	0.008%	32S	8.586	0.035%	39S	9.354	0.077%	58S	10.724	0.348%	88S

(b) CVRP

Method	Training distr. $n = 100$			$n = 125$			Generalization $n = 150$			$n = 200$		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
HGS	15.563	-0.536%	19S	-	-	-	19.055	-0.884%	32S	21.766	-1.096%	61S
LKH3	15.646	0.000%	52S	17.50	0.000%	-	19.222	0.000%	72S	22.003	0.000%	90S
POMO (greedy)	15.874	1.430%	24S	17.818	1.818%	34S	19.750	2.757%	52S	23.318	5.992%	87S
POMO (sampling)	15.713	0.399%	24S	17.612	0.642%	34S	19.488	1.393%	52S	23.378	6.264%	87S
SGBS*	15.659*	0.08%*	-	-	-	-	19.426*	1.08%*	-	22.567*	2.59%*	-
EAS	15.663	0.081%	66S	17.536	0.146%	82S	19.321	0.528%	123S	22.541	2.460%	179S
MEMENTO (POMO)	15.657	0.066%	118S	17.521	0.095%	150S	19.317	0.478%	169S	22.492	2.205%	392S
COMPASS	15.644	-0.019%	29S	17.511	0.064%	39S	19.313	0.485%	56S	22.462	2.098%	85S
MEMENTO (COMPASS)	15.634	-0.082%	82S	17.497	-0.041%	100S	19.290	0.336%	118S	22.405	1.808%	272S

Table 6: Results of MEMENTO and the baseline algorithms with instance augmentation for (a) TSP and (b) CVRP.

(a) TSP

Method	Training distr. $n = 100$			$n = 125$			Generalization $n = 150$			$n = 200$		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
LKH3	7.765	0.000%	2.9S	8.583	0.000%	4.4S	9.346	0.000%	6S	10.687	0.000%	11S
SGBS	7.769	0.058%	-	-	-	-	9.367	0.220%	-	10.753	0.619%	-
POMO (sampling)	7.767	0.026%	16S	8.594	0.128%	20S	9.376	0.321%	29S	10.916	2.14%	45S
EAS	7.768	0.038%	39S	8.590	0.080%	46S	9.361	0.159%	64S	10.730	0.403%	91S
MEMENTO (POMO)	7.765	0.008%	43S	8.586	0.035%	52S	9.355	0.091%	77S	10.743	0.526%	115S

(b) CVRP

Method	Training distr. $n = 100$			$n = 125$			Generalization $n = 150$			$n = 200$		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
HGS	15.563	-0.536%	-	-	-	-	19.055	-0.884%	-	21.766	-1.096%	-
LKH3	15.646	0.000%	-	17.50	0.000%	-	19.222	0.000%	-	22.003	0.000%	-
SGBS	15.659	0.08%	-	-	-	-	19.426	1.08%	-	22.567	2.59%	-
POMO (sampling)	15.67	0.18%	24S	17.56	0.33%	34S	19.43	1.08%	52S	23.24	5.64%	87S
EAS	15.623	-0.175%	66S	17.473	-0.153%	82S	19.261	0.213%	123S	22.556	2.49%	179S
MEMENTO (POMO)	15.616	-0.196%	118S	17.511	0.040%	150S	19.316	0.477%	169S	22.515	2.308%	392S

adapt to POMO, although this is completely agnostic to MEMENTO’s method in itself). Since those factors impacts TSP and CVRP performance in different ways, this explains why their relative speed differ, i.e. why MEMENTO (COMPASS) is slower on TSP but faster on CVRP.

A.4 Time complexity analysis

To get the time curves reported in Fig. 7, we used CVRP, since it is the environment were MEMENTO was the slowest compared to EAS. We hence expect curves on TSP to be even better for MEMENTO.

Table 7: Results of MEMENTO and the baseline algorithms with budget expressed as runtime for (a) TSP and (b) CVRP.

(a) TSP

Method	Training distr. $n = 100$			$n = 125$			Generalization $n = 150$			$n = 200$		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
LKH3	7.765	0.000%	-	8.583	0.000%	-	9.346	0.000%	-	10.687	0.000%	-
POMO (sampling)	7.779	0.216%	39S	8.609	0.299%	46S	9.401	0.585%	64S	10.956	2.513%	91S
EAS	7.778	0.161%	39S	8.604	0.238%	46S	9.380	0.363%	64S	10.759	0.672%	91S
MEMENTO	7.768	0.046%	39S	8.592	0.110%	46S	9.365	0.203%	64S	10.760	0.681%	91S

(b) CVRP

Method	Training distr. $n = 100$			$n = 125$			Generalization $n = 150$			$n = 200$		
	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
LKH3	15.647	0.000%	-	17.504	0.000%	-	19.225	0.000%	-	22.007	0.000%	-
POMO (sampling)	15.713	0.399%	66S	17.612	0.642%	82S	19.488	1.393%	123S	23.378	6.264%	179S
EAS	15.663	0.081%	66S	17.536	0.146%	82S	19.321	0.528%	123S	22.541	2.460%	179S
MEMENTO	15.660	0.086%	66S	17.526	0.127%	82S	19.321	0.502%	123S	22.546	2.450%	179S

Table 8: Results of MEMENTO, POMO and EAS on instances of size 500 of (a) TSP and (b) CVRP for increasing number of sequential attempts and sizes of attempt batch.

(a) TSP

Method	$N = 20$					$N = 40$				
	200	400	600	800	1000	200	400	600	800	1000
POMO (sampling)	16.9810	16.9707	16.9638	16.9619	16.9574	16.9717	16.96	16.9549	16.9523	16.9493
EAS	16.9223	16.8923	16.8754	16.864	16.8556	16.8777	16.8468	16.83	16.8208	16.8143
MEMENTO	16.8312	16.81	16.799	16.7939	16.7902	16.8187	16.8018	16.7926	16.7855	16.7815

Method	$N = 60$					$N = 80$				
	200	400	600	800	1000	200	400	600	800	1000
POMO (sampling)	16.9678	16.9587	16.9539	16.952	16.9503	16.9634	16.9547	16.9513	16.9495	16.9474
EAS	16.8616	16.8353	16.8211	16.8134	16.8084	16.8521	16.8234	16.8106	16.8014	16.7941
MEMENTO	16.8129	16.7966	16.7867	16.7816	16.7780	16.8087	16.7935	16.7854	16.7811	16.7770

(b) CVRP

Method	$N = 20$					$N = 40$				
	200	400	600	800	1000	200	400	600	800	1000
POMO (sampling)	37.5256	37.4917	37.4763	37.4639	37.4570	37.4858	37.4599	37.4475	37.4372	37.4307
EAS	37.6107	37.5988	37.5914	37.582	37.5769	37.5022	37.4546	37.422	37.4017	37.3849
MEMENTO	37.3398	37.2992	37.2731	37.2589	37.2527	37.3172	37.2776	37.2515	37.2357	37.2260

Method	$N = 60$					$N = 80$				
	200	400	600	800	1000	200	400	600	800	1000
POMO (sampling)	37.4598	37.436	37.4228	37.4191	37.4135	37.4588	37.4335	37.4198	37.4113	37.4050
EAS	37.4569	37.3679	37.3248	37.2892	37.2624	37.3588	37.2804	37.2346	37.2021	37.1747
MEMENTO	37.292	37.2607	37.2305	37.2171	37.2092	37.2887	37.2556	37.2382	37.2268	37.2142

For the instance size scaling, we use 50 starting points and 100 sequential attempts, and 1 layer in the decoder, and evaluate several instance sizes going from 100 to 1000. For the decoder layer depth scaling, we use CVRP100, 100 starting points and 160 sequential attempts. We then evaluate methods for a depth going from 1 to 10.

B Training procedure

This section give a detailed description of how an existing pre-trained model can be augmented with MEMENTO, and how we train MEMENTO to acquire adaptation capacities.

Firstly, we take an existing single-agent model that was trained using the REINFORCE algorithm and reuse it as a base model. In our case, the base model is POMO. We augment POMO with a memory module and begin the training procedure which aims to create a policy that is able to use past experiences to make decisions in a multi-shot setting. We initialize the memory module weights with

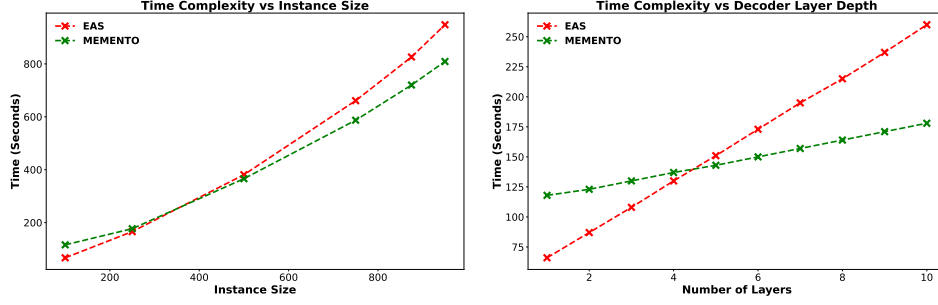


Figure 7: Time complexity of MEMENTO and EAS for increasing values of decoder size, and instance size. MEMENTO is shown to scale better in time than EAS.

small values such that they barely affect the initial output. Hence, the initial MEMENTO checkpoint maintains the same performance as the pre-trained POMO checkpoint.

In the training procedure, the policy is trained to use data stored in the memory to take decisions and solve a problem instance. This is achieved by training the policy in a budgeted multi-shot setting on a problem instance where past experiences are collected and stored in a memory. The memory is organised by nodes such that only information about a specific node is found in a data row that corresponds to that node. The policy retrieves data from the memory at each budget attempt and learns how to use the data to decide on its next action.

The details of the MEMENTO training procedure are presented in Algorithm 1 and can be understood as follows. At each iteration, we sample a batch B of instances from a problem distribution \mathcal{D} . Then, for each instance ρ_i where $i \in 1, \dots, B$, for K budget attempts, we retrieve data from the memory. This is done as follows; given that the current selected node is a_j , we only retrieve data $M(a_j)$ associated with node a_j and its starting point. However, the method is agnostic to starting point sampling and would work without it. The features associated with the retrieved action are then normalised. We add remaining budget as an additional feature and process the data by a Multilayer Perceptron (MLP) which outputs a scalar weight for each action. We compute correction logits by averaging the actions based on their respective weights. The correction logits are added to the logits of the base policy. We then rollout the resulting policy $\pi_{\tilde{\theta}}$ on the problem instance (i.e., generate a trajectory which represents a solution to the instance). After every policy rollout attempt, the memory is updated with transitions data such as the action taken, the obtained return, the log-probability of the action and the log-probability of the trajectory. These data are stored in the row that corresponds to the current selected node.

Details about the loss For each problem instance, we want to optimise for the best return. At each attempt, we apply the rectified linear unit (ReLU) function to the difference between the last return and the best return ever obtained. We use the rectified difference to compute the REINFORCE loss at each attempt to avoid having a reward that is too sparse and perform back-propagation through the network parameters of our model (including the encoder, the decoder and the memory networks). The sum of the rectified differences is equal to the best return ever obtained over the budget. As the budget is used, it becomes harder to improve over the previous best, the loss terms hence getting smaller. We found that adding a weight to the terms, with logarithmic increase, helped ensuring that the last terms would not vanish, and thus improved performance. We provide the mathematical formulation below.

Given a problem instance, we unroll B trajectories that we store iteratively in the memory. Each trajectory τ_i generates a return $R(\tau_i)$. The advantage for each trajectory is defined as $\tilde{R}(\tau_i) = \max(R(\tau_i) - R_{best}, 0)$, where R_{best} is the highest return found so far. The total loss for updating the policy is calculated using the REINFORCE algorithm: $\mathcal{L} = -\sum_{i=1}^B \log(1 + \epsilon + i) \tilde{R}(\tau_i) \sum_t \log \pi_M(a_t | s_t, M_t)$, where π_M is the policy enriched with the memory M_t , using the logits l_M defined in eq.1 in the paper. ϵ is a small number ensuring that the first term is not zero.

1017 To keep the computations tractable, we still compute a loss at each step, estimate the gradient, and
 1018 average them sequentially until the budget is reached, at which point we take a gradient update step
 1019 and consider a new batch of instances.

1020 In practice, we also observe that we can improve performance further by adding an optional refining
 1021 phase where the base model is frozen, and only the memory module is trained, with a reduced learning
 1022 rate (multiplied by 0.1), for a few hours.

Algorithm 1 MEMENTO Training

```

1: Input: problem distribution  $\mathcal{D}$ , problem size  $N$ , memory  $M$ , batch size  $B$ , budget  $K$ , number of
   training steps  $T$ , policy  $\pi_\theta$  with pre-trained parameters  $\theta$ .
2: initialize memory network parameters  $\phi$ 
3: combine pre-trained policy parameters and memory network parameters  $\tilde{\theta} = (\theta, \phi)$ 
4: for step 1 to  $T$  do
5:    $\rho_i \leftarrow \text{Sample}(\mathcal{D}) \forall i \in 1, \dots, B$ 
6:   for attempt 1 to  $K$  do
7:     for node 1 to  $N$  do
8:        $m_j \leftarrow \text{Retrieve}(M) \forall j \in 1, \dots, B$  {Retrieve data from the memory}
9:        $\tau_i^j \leftarrow \text{Rollout}(\rho_i, \pi_{\tilde{\theta}}(\cdot|m_j)) \forall i, j \in 1, \dots, B$ 
10:       $m_j \leftarrow f(m_j, \tau_i^j)$  {Update the memory with transition data}
11:       $R_i^* \leftarrow \max(R_i^*, \mathcal{R}(\tau_i^j)) \forall i \in 1, \dots, B$  {Update best solution found so far}
12:       $\nabla L(\tilde{\theta}) \leftarrow \frac{1}{B} \sum_{i \leq B} \text{REINFORCE}(\text{ReLU}(\tau_i^j - R_i^*))$  {Estimate gradient}
13:       $\nabla L(\tilde{\theta}) \leftarrow \frac{1}{K} \sum_{i=1}^K \nabla L(\tilde{\theta})$  {Accumulate gradients}
14:       $\tilde{\theta} \leftarrow \tilde{\theta} - \alpha \nabla L(\tilde{\theta})$  {Update parameters}

```

1023 C Hyper-parameters

1024 We report all the hyper-parameters used during train and inference time. For our method MEMENTO,
 1025 there is no training hyper-parameters to report for instance sizes 125, 150, and 200 as the model
 1026 used was trained on instances of size 100. The hyper-parameters used for MEMENTO are reported
 1027 in Table 9. Since we also trained POMO and COMPASS on larger instances, we report hyper-parameters
 1028 used for POMO and COMPASS in Table 10 and Table 11, respectively.

1029 D Model checkpoints

1030 Our experiments focus on two CO routing problems, TSP and CVRP, with methods being trained on
 1031 two distinct instance sizes: 100 and 500. Whenever possible, we re-use existing checkpoints from
 1032 the literature; in the remaining cases, we release all our newly trained checkpoints in the repository
 1033 *anonymised for the review process*.

1034 We evaluate MEMENTO on two CO problems, TSP and CVRP, and compare the performance
 1035 to that of two main baselines: POMO (Kwon et al., 2020) and EAS (Hottung et al., 2022). The
 1036 checkpoints used to evaluate POMO on TSP and CVRP are the same as the one used in Grinsztajn
 1037 et al. (2023) and Chalumeau et al. (2023b), and the EAS baseline is executed using the same POMO
 1038 checkpoint. These checkpoints were taken in the publicly available repository <https://github.com/instadeepai/poppy>. The POMO checkpoint is used in the initialisation step of MEMENTO
 1039 (as described in Appendix B). To combine MEMENTO and COMPASS on TSP100, we re-use the
 1040 COMPASS checkpoint available at <https://github.com/instadeepai/compass> and add the
 1041 memory processing layers from the MEMENTO checkpoint trained on TSP100. This checkpoint is
 1042 also released at *anonymised for the review process*.

1044 For larger instances, we compare our MEMENTO method to three baselines: POMO, EAS and COMPASS.
 1045 Since no checkpoint of POMO and COMPASS existed, we trained them with the tricks explained
 1046 in Section 4. The process to generate the MEMENTO checkpoint and the MEMENTO (COMPASS)
 1047 checkpoint is then exactly the same. All those checkpoints are available, for both TSP and CVRP.

Table 9: The hyper-parameters used in MEMENTO

(a) TSP

Phase	Hyper-parameters	TSP100	TSP(125, 150)	TSP200	TSP500
Train time	budget	200	-	-	200
	instances batch size	64	-	-	32
	starting points	100	-	-	30
	gradient accumulation steps	200	-	-	400
	memory size	40	-	-	80
	number of layers	2	-	-	2
	hidden layers	8	-	-	8
	activation	GELU	-	-	GELU
	learning rate (memory)	0.004	-	-	0.004
	learning rate (encoder)	0.0001	-	-	0.0001
	learning rate (decoder)	0.0001	-	-	0.0001
Inference time	policy noise	1	0.2	0.1	0.8
	memory size	40	40	40	40

(b) CVRP

Phase	Hyper-parameters	CVRP100	CVRP(125, 150)	CVRP200	CVRP500
Train time	budget	200	-	-	200
	instances batch size	64	-	-	8
	starting points	100	-	-	100
	gradient accumulation steps	200	-	-	800
	memory size	40	-	-	40
	number of layers	2	-	-	2
	hidden units	8	-	-	8
	activation	GELU	-	-	GELU
	learning rate (memory)	0.004	-	-	0.004
	learning rate (encoder)	0.0001	-	-	0.0001
	learning rate (decoder)	0.0001	-	-	0.0001
Inference time	policy noise	0.1	0.1	0.1	0.3
	memory size	40	40	40	40

Table 10: The hyper-parameters used in POMO

Phase	Hyper-parameters	TSP500	CVRP500
Train time	starting points	200	200
	instances batch size	64	32
	gradient accumulation steps	1	2
Inference time	policy noise	1	1
	sampling batch size	8	8

E Implementation details

The code-base is written in JAX (Bradbury et al., 2018), and is mostly compatible with recent repositories of neural solvers written in JAX, i.e. Poppy and COMPASS. The problems’ implementation are also written in JAX and fully jittable. Those come from the package Jumanji (Bonnet et al., 2023). CMA-ES implementation to mix MEMENTO and COMPASS is taken from the research package QDax (Chalumeau et al., 2023a). Neural networks, optimizers, and many utilities are implemented using the DeepMind JAX ecosystem (Babuschkin et al., 2020).

F Can MEMENTO discover the REINFORCE update?

In Section 3.2, we presented the architecture used by the auxiliary model that processes the memory data to derive the new action logits. The intuition behind this architecture choice is that it should be able to learn the REINFORCE update rule. Indeed the REINFORCE loss associated with a new transition is $R \log(\pi_\theta(a))$, such that $\frac{\partial R \log(\pi_\theta(a))}{\partial l_a} = R(1 - \pi_\theta(a))$. Therefore, simply having $H_{\theta_M}(f_a)$ match $R(1 - \pi_\theta(a))$ would recover a REINFORCE-like update. This is feasible, as R and $\log(\pi_\theta(a))$ are included in the features f_a .

On Fig. 4, we compare the rule learned by MEMENTO compared to REINFORCE. In Appendix G, we present an ablation study of the features used in the update rule of MEMENTO, showing how much performance can be gained from the use of more information to derive the update rule.

Table 11: The hyper-parameters used in COMPASS

Phase	Hyper-parameters	TSP500	CVRP500
Train time	latent space dimension	16	16
	training sample size	64	32
	instances batch size	8	8
	gradient accumulation steps	8	16
Inference time	policy noise	0.5	0.3
	num. CMA-ES components	1	1
	CMA-ES init. sigma	100	100
	sampling batch size	8	8

G Ablation Study: MEMENTO input features

In Section 3.2, we present MEMENTO, in particular, we present all the inputs that are used by the neural module to derive the new action logits from the memory data. These inputs, or features, are information associated to each past action taken, and that help decide whether those actions should be taken again or not. In Section 4.1, we compare the rule learned by MEMENTO compared to the policy-gradient update REINFORCE. This comparison is made on the features that both REINFORCE and MEMENTO use: i.e. the action log probability, and the return (or advantage if a baseline is used). Although REINFORCE only uses those two features, MEMENTO uses more, which enables to refine even more the update, and also to adapt it to the budget remaining in the search process. As a recall, in addition to the log probability and return, MEMENTO uses: the log probability of the full trajectory, the log probability of the rest of the trajectory after the action was taken, the budget at the time the action was taken, the action logit suggested by MEMENTO when the action was taken, and the budget currently remaining.

To validate the impact of all the features used in the memory, we provide an ablation study of those features. To highlight the interest of using all the additional features, we report the performance of MEMENTO using only the return and the log probability, against the performance of MEMENTO with all features, on Fig. 8. We also report on Fig. 9 a bigger ablation where components are added one after the others.

We can extract two main observations: (i) first, adding the remaining budget completely changes the strategy. We can see on the right panel of Fig. 8 that having access to this additional feature enables MEMENTO to explore much more, and then to focus on high-performing solutions when it gets closer to the end of the budget; (ii) then, Fig. 9 confirms empirically that all features contribute to improving the overall performance.

Last Sample Performance

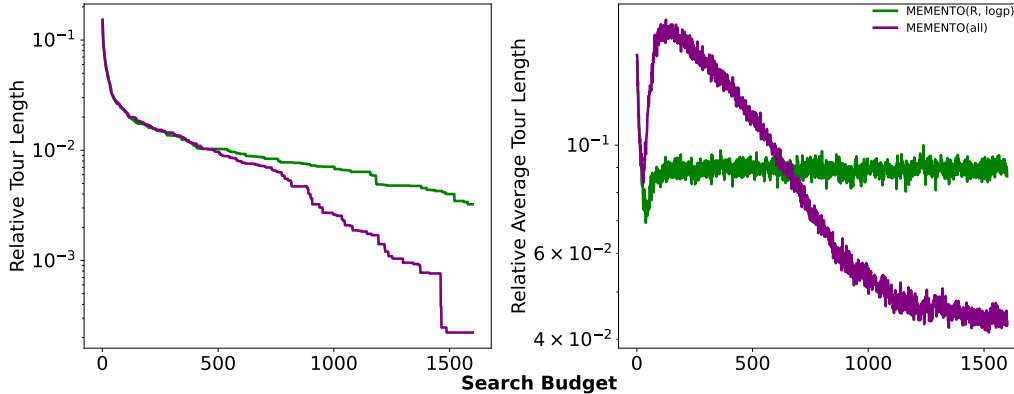


Figure 8: Ablation study of MEMENTO: comparing the impact of memory features that are not available in usual policy gradient estimations methods. The left plot reports the best solution found so far. The right plot shows the performance of the latest solution sampled. The plot illustrates how the additional features enable to achieve a complex exploration strategy, reaching a significantly more efficient adaptation mechanism.

Last Sample Performance

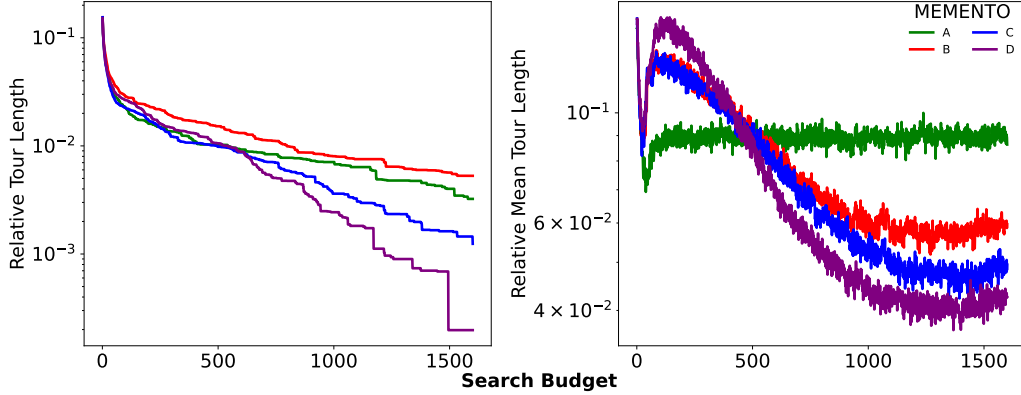


Figure 9: Full ablation study of MEMENTO: comparing the impact of memory features that are not available in usual policy gradient estimations methods. The left plot reports the best solution found so far. The right plot shows the performance of the latest solution sampled. A: return + logp; B: A + remaining budget; C: B + budget when action was taken; D: C + memory logp + trajectory logp + end trajectory logp.

H Evaluation metrics during training phase

In this section, we provide two plots of MEMENTO’s training phase. They show the evolution of performance over time during MEMENTO’s training on CVRP100. The left plot reports the evolution of the best tour length obtained during validation over time. The right plot reports the evolution of the improvement delta over time, i.e. the difference between the quality of the best solution generated minus the quality of the first solution generated. This metric shows well how the training phase results in MEMENTO learning an update rule that is able to significantly improve the base policy.

MEMENTO Training Phase

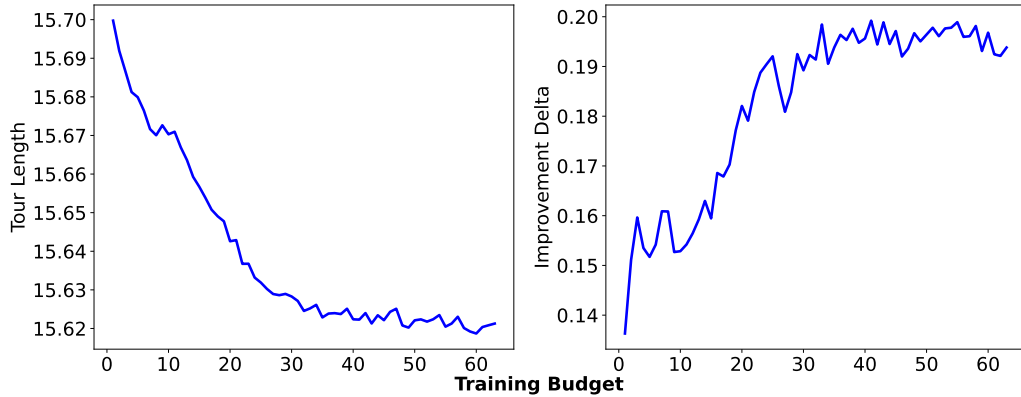


Figure 10: Evaluation metrics during the training phase of MEMENTO.

I Memory retrieval mechanism

In this section, we provide additional motivation for the choice of the node retrieval mechanism used in MEMENTO.

Ideally, the retrieval mechanism should retrieve data based on similarity to the current partial solution. But this comes with a computation cost since it requires getting a similarity measure and extracting the k most similar points in the entire memory. We observe that retrieving from the same node is an

1101 excellent proxy for similarity, and that the most similar points are very likely to come from the same
1102 node. This retrieval strategy hence provides a better trade-off between quality and computation cost,
1103 which is why it was selected as the final strategy for MEMENTO.

1104 Nevertheless, MEMENTO comes as a framework, and each practitioner is free to change part of
1105 the method to fit each specific problem. One can hence update the retrieval mechanism if desired.
1106 Below is an example of an alternative strategy for retrieval that does consider the partial solution
1107 constructed. Since the output of the multi-head attention of POMO 's decoder builds a low-dimensional
1108 representation of the partial solution, one can store this vector in the memory, and when building a
1109 new solution, retrieve only the k-nearest neighbors of the current partial solution's representation. One
1110 could even apply further dimensionality reduction to reduce the cost of the nearest neighbor search.
1111 This approach brings a significant cost increase, even when using state-of-the-art approximated
1112 nearest neighbor JAX implementation. This effect gets worse when batching the attempts, or the
1113 problem instances. In the problems and settings considered in this paper, our simplified retrieval
1114 approach maintains similar results, while significantly improving scaling.

1115