ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DEGREE COURSE IN ARTIFICIAL INTELLIGENCE

THESIS

in

Decision-Focused Learning

SMART SAMPLING APPROACHES FOR DECISION-FOCUSED LEARNING

Candidate:
Marco Foschini

Supervisors:
Prof. Michele Lombardi
Prof. Tias Guns

Academic year 2021/2022

Contents

List of Figures

Abstract

Many real-word decision- making problems are defined based on forecast parameters: for example, one may plan an urban route by relying on traffic predictions. In these cases, the conventional approach consists in training a predictor and then solving an optimization problem. This may be problematic since mistakes made by the predictor may trick the optimizer into taking dramatically wrong decisions. Recently, the field of Decision-Focused Learning overcomes this limitation by merging the two stages at training time, so that predictions are rewarded and penalized based on their outcome in the optimization problem. There are however still significant challenges toward a widespread adoption of the method, mostly related to the limitation in terms of generality and scalability. One possible solution for dealing with the second problem is introducing a caching-based approach, to speed up the training process. This project aims to investigate these techniques, in order to reduce even more, the solver calls. For each considered method, we designed a particular smart sampling approach, based on their characteristics. In the case of the SPO method, we ended up discovering that it is only necessary to initialize the cache with only several solutions; those needed to filter the elements that we still need to properly learn. For the Blackbox method, we designed a smart sampling approach, based on inferred solutions.

1 Introduction

It is not uncommon to find real-world applications where both machine learning and optimization are used jointly. In these cases, a model is trained in order to predict parameters for the optimization model. One of the most recurring strategy for this type of tasks is the predict-then-optimize paradigm.

Let's imagine a travelling salesman problem, where the distances are unknown but, it is possible to predict them from some features (traffic, weather, condition of the road). If we apply the predict-then-optimize paradigm, we are going to train a model and then we will use it for computing distances between edges. The solver will use this information to compute the shortest path.

Common sense would make us think that better predicting models will result in solutions closer to the real ones, but in reality this is not the case. As a matter of fact, the ML model will always make some errors and they will impact in some ways the solution obtained. Think again about the travelling salesman problem, if the model underestimates the shortest distance, it is more likely to get a better solution with respect to a model that is overestimating it.

As a consequence, one way to obtain better predicting models would be, during training, to take into consideration the effect of the errors on the optimization model. This has been done in several studies; for example Elmachtoub and Grigas (2017) propose a custom loss which takes into account the results of the optimization model, while Vlastelica et al. (2019) presents a custom layer, also here the error of the predicted solution will be taken into account.

By introducing these new approaches, also new challenges show up. During the training process, now it will be necessary to compute the output of the optimization model, but if we are dealing with an NP-hard problem, it would be impossible to obtain a properly trained model, due to high training time.

One possible solution is to introduce a caching scheme, as proposed by Mulamba et al. (2020). Instead of always calling the optimization model, the solution will be obtained by searching inside a cache. However in order to obtain performances comparable to those where the solver is always used, it is not possible to rely only on the cache; sometimes the optimization model must be called.

The thesis tries to discover if it is possible to develop a smart sampling scheme capable of understanding if it is worth using the optimization model or the cache. We are going to propose two different smart sampling approach, each designed for a particular method. The first one, created specifically for SPO, will basically filter the element that still needs to be learnt, while for Blackbox, it is possible to add new solutions, without calling the solver and by inferring them from the predicted values.

We also noticed, that for particular problems, it is not necessary to sample new solutions. This is caused by how the data for predicting the values are generated.

The thesis is organized as follow:

- Section 3 presents the considered problems and the used methods for the training process.
- Section 4 is about the way in which the balance between the solver calls and the cache calls, influences the model.
- Section 5 focuses on finding which solutions are the most important for learning, considering an SPO model, for a knapsack problem.
- Section 6 is about exploiting the knowledge of the previous section, in order to develop a particular initialization for an SPO model, the filter initialization. In this way sampling new solution is not needed.
- Section 7 is about using the cosine similarity and additional information, in order to infer the solutions from the predicted values, by using a Blackbox model.
- Section 9 tries to check if both the approaches designed for a particular method can be used altogether with the other method.

2 Related work

Predict-and-optimize problems are becoming more and more important in many applications. Right now, the standard technique is the Two-stage approach, where a model is trained, without taking into account the optimization task.

However new techniques have been invented. For example Elmachtoub and Grigas (2017) propose a new framework in order to train a ML model. It will take into consideration the optimization problem and as a consequence, the model is able to make more optimal decisions. Another possible approach is presented by Vlastelica et al. (2019), where a layer with a custom backward pass is used, in order to consider the optimization problem. Last but not least, Mulamba et al. (2020) propose another possible loss, based on viewing non-optimal solutions as negative examples and deal with the bottleneck of all these approaches, which consists of the frequent computation of the optimal solutions, by introducing a caching scheme. All these techniques can be applied for linear optimization problem with a convex feasible region.

In the end we can say that optimizing over prediction is as valid as stochastic optimisation over learnt distributions, as proven by Demirović et al. (2019). They have also classified the possible learning approaches, into three groups: indirect, which do not rely on the optimization problem, semi-direct, which rely on knowledge of the optimization problem, like the ranking and, direct approaches, which use the optimization problem in the learning process.

3 Background Knowledge

3.1 Problems

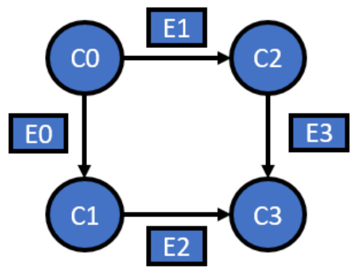
3.1.1 Shortest path problem

Given a directed graph represented by a 5 x 5 grid where each edge i is associated to a cost y_i. The objective is to reach the Southeast corner from the Northwest corner, by taking the path with the minimum cost.

First of all the optimization model will generate an incidence matrix from the corresponding directed graph; it consists of an n x m matrix A, where n is the number of vertices, while m is the number of edges. The matrix is defined as:

$$A_{ij} = \begin{cases} -1 & \text{if the edge } j \text{ leaves the vertex } i \\ 1 & \text{if the edge } j \text{ enters the vertex } i \\ 0 & \text{else} \end{cases} \tag{1}$$

Here is an example:



	E0	E1	E2	E3
C0	-1	-1	0	0
C1	1	0	-1	0
C2	0	1	0	-1
C3	0	0	1	1

Figure 1: Example of incidence matrix.

Given also $Sol \in \mathrm{R}^m$, a vector of binary decision variable where:

$$Sol_i = \begin{cases} 1 & \text{if the edge } i \text{ is traversed} \\ 0 & \text{if the edge } i \text{ is not traversed} \end{cases} \tag{2}$$

And by defining a binary vector B of length n, where the starting element is set to -1 and the last one is set to 1, it is possible to specify the necessary constraint for our model, which is simply:

$$A \cdot Sol = B \tag{3}$$

Last but not least, the objective function is defined as:

$$\min(y \cdot Sol) \tag{4}$$

Since we are in the branch of decision-focused learning, the costs y must be predicted from some features x.

3.1.1.1 SPO data

In order to generate the costs of the edges for the Shortest path problem, we applied the generation process introduced by Elmachtoub and Grigas (2017). Each set of costs c_i can be predicted by a feature vector $x_i \in \mathrm{R}^5$ sampled from a multivariate Gaussian distribution with i.i.d. standard normal entries, which means $x \sim N(0, I_5)$. It is also necessary to compute the matrix B* $\in \mathrm{R}^{dx5}$, (d = 40 since it corresponds to the total amount of edges in a 5 x 5 directed graph). Each element of the matrix corresponds to a Bernoulli random variable which is equal to 1 with a probability of 0.5. Given x_i and the matrix B^*, it is possible to compute the costs vector c_i with the following formula:

$$c_{ij} = \left[\left(\frac{1}{\sqrt{5}} (B^* x_i)_j + 3 \right)^{deg} + 1 \right] \cdot \epsilon_i^j \qquad \forall j = 1..d \tag{5}$$

c_{ij} denotes the j^{th} component of c_i, while $(B^*x_i)_j$ the j^{th} component of B^*x_i. Deg is a fixed positive integer number, while ϵ^j is a noise term; it is generated by sampling from a random distribution between [1-$\bar{\epsilon}$,1+$\bar{\epsilon}$] where $\bar{\epsilon}$ is a parameter greater than zero.

3.1.2 Knapsack problem

A knapsack problem consists of a maximum capacity C and a set of m items, each of them associated to a value y_i and a cost w_i. The task consists in selecting n elements in order to maximize the total value, without exceeding the capacity. The optimization model will receive all this information and it will return as output the solution, consisting of a binary array $Sol \in \mathrm{R}^m$ where:

$$Sol_i = \begin{cases} 1 & \text{if the item } i \text{ is picked} \\ 0 & \text{if the item } i \text{ is not picked} \end{cases} \tag{6}$$

In order to properly define the optimization model, it is necessary to correctly set the objective function and the constraints, which will be:

$$\max \sum_{i=1}^{n} y_i sol_i \tag{7}$$

Such that:

$$\sum_{i=1}^{n} w_i sol_i \leqslant capacity \tag{8}$$

Since we are dealing with a decision-focused learning problem, the values y need to be predicted from some features x.

3.1.2.1 Energy Price dataset

In this case the features are obtained by considering the data-set from Ifrim et al. (2012). It consists of 30 minutes slots of historical energy price data from 2011 to 2013. Each of them is associated to features like: day-ahead estimates of weather characteristics, SEMO day-ahead forecasted energyload and wind-energy production; but also to a target represented by the energy price.

Given its characteristics, the data-set can be used to build knapsack problems, as presented in the paper by Mandi et al. (2019). In our settings, each knapsack problem will consist of 48 time slots (corresponding to one day), where each of them has a value y, represented by the energy price and a cost w, which is randomly assigned. The possible values for it are: 3,5,7.

In order to increase the difficulty of solving these problems, the same approach presented by Pisinger (2005) has been applied, which means that a correlation between the costs w and the values y is added. Basically, each value y_i is obtained by multiplying themselves with their correspondent cost $\tilde{w_i}$, where $\tilde{w_i} = w_i + \xi$. Where ξ is a Gaussian noise $\sim \mathbb{N}(0, 25)$.

3.2 Methods

All the presented methods will follow the predict-then-optimize paradigm. A model is trained in order to predict costs from some features and then, these costs are used by the solver in order to compute the solution.

3.2.1 Two-stage approach

The approach that is used widely in the industrial sector is the Two-stage approach. A model m, represented by its weights w, is trained accordingly to the stochastic gradient descent, in order to minimize a suitable loss, which in our case is the mean squared error.

$$\mathcal{L}_{mse}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2 \tag{9}$$

Algorithm 1 Stochastic batch gradient descent for the Two-stage approach

```
for each epochs do
    Sample N elements from the dataset
    $\mathcal{L} = 0$
    for $i = 1, \ldots, N$ do
        $\hat{y_i} = m(x_i, w)$
        $\mathcal{L} = \mathcal{L} + \mathcal{L}_{mse}(y_i, \hat{y_i})$
    end for
    $w = w - lr \cdot \nabla\mathcal{L}$
end for
```

Since the MSE does not give relevant information about the solution obtained from the predicted target $\hat{y}$, it is necessary to use an additional metric which, in our case, will be the regret, defined as:

$$regret(y, \hat{y}) = |f(v^*(\hat{y}), y) - f(v^*(y), y)| \tag{10}$$

where:

$$v^*(y) = \begin{cases} \arg\max_{s \in S} f(s, y) & \text{if maximization problem} \\ \arg\min_{s \in S} f(s, y) & \text{if minimization problem} \end{cases} \tag{11}$$

$v^*(\hat{y})$ denotes the optimal solution $\hat{s}$ obtained by calling the solver using the predicted target $\hat{y}$, while $v^*(y)$ denotes the optimal solution s for the real target y. $f(\hat{s}, y)$ is the final cost linked to the predicted solution $\hat{s}$ by using the real costs y, while $f(s, y)$ is the final cost for the real solution.

One of the advantages of the Two-stage approach, is that we do not rely on the solution returned by the solver, as a consequence, it is not necessary to call it during the training process, leading to a faster training with respect to the other presented options. However, at the same time, by doing that, the model is not considering the impact of the predicted target, over the returned solution, leading to sub-optimal decisions.

Think about the Shortest path problem, there can be a high difference between a model which overestimates the shortest distance between two edges, with respect to one which underestimate it. In the first case, the computed solution can be totally different with respect to the real one, even if all the other edges are predicted correctly, while for the second case, it is more likely to obtain a solution closer to the real one; and for this reason, it would be necessary if not essential to consider the effect of the predictions over the predicted solutions.

3.2.2 SPO approach

As already mentioned before, one of the major drawbacks of the Two-stage approach, is that it does not take into account the impact of the predicted target, over the returned solution, leading in most of the cases to sub-optimal choices. Our final goal would be to learn the model parameters w in order to minimize the regret of the predicted costs, however we can not use it directly as loss, since it is non-continuous and it would involve in differentiating over an argmin or an argmax, based on the problem that we are considering, as depicted in the equation 11.

In order to deal with this problem, Elmachtoub and Grigas (2017) derive a new loss from the regret formula. In fact if the optimization problem can be described as:

$$v^*(y) = \arg\min_{s \in S}(y^T w) \tag{12}$$

Then we can define the regret as:

$$regret(y, \hat{y}) = y^T(v^*(\hat{y}) - v^*(y)) \tag{13}$$

However, the formula is still non differentiable, but it is possible to derive an upper bound for it, that will be our loss, in fact:

$$L_{SPO+}(y, \hat{y}) = \max_{s \in S}(y^T s - 2\hat{y}^T s) + 2\hat{y}^T v^*(y) - y^T v^*(y) \tag{14}$$

Given that, the authors discovered the formula:

$$g(y, \hat{y}) = v^*(y) - v^*(2\hat{y} - y) \in \nabla\mathcal{L}_{SPO+}(y, \hat{y}) \tag{15}$$

It can be easily plugged in our algorithm as sub-gradient of the SPO+ loss; we can apply the same approach also for maximization problem, by simply switching the sign of the subgradient.

Algorithm 2 Stochastic batch gradient descent for the SPO approach

```
for each epochs do
    Sample N elements from the dataset
    $\nabla\mathcal{L} = 0$
    for $i = 1, \ldots, N$ do
        $\hat{y}_i = m(x_i, w)$
        $\nabla\mathcal{L} = \nabla\mathcal{L} + v^*(y_i) - v^*(2\hat{y}_i - y_i)$
    end for
    $w = w - lr * \nabla\mathcal{L}$
end for
```

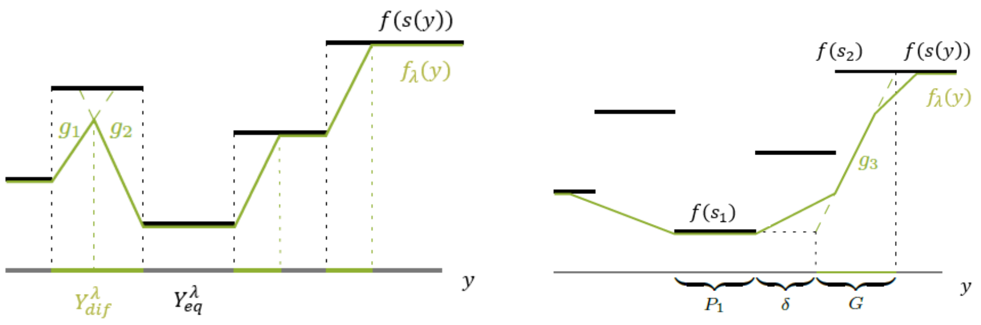
In the end, we are going to update the weights of our network, based only on the sub-gradient of the SPO+ loss, which is simply the difference in terms of solutions between the one obtained using the real costs y_i and the one obtained by combining y_i with the predicted costs $\hat{y}_i$. By applying this approach, we are able to take into consideration the optimization problem during the training process. However, one of the biggest issues that appears is that it will be necessary to solve the optimization problem for each instance, since $v^*(2\hat{y}_i - y_i)$ is needed, while $v^*(y_i)$ can be stored in a cache during the initialization, since we suppose it is known. As you can imagine, solving all these optimization problems can be computationally expensive, leading to a high training time.

3.2.3 Blackbox approach

In order to consider the optimization problem during the training process, Vlastelica et al. (2019) introduce a new layer with a custom backward pass. The settings are the same of the optimization problem described in the section 3.2.2

$$v^*(y) = \arg\min_{w \in S}(y^T w) \tag{16}$$

The authors pointed out that, the fundamental problem of differentiating through a combinatorial solver, is not actually the lack of differentiability, but the fact that the gradient is a constant zero or does not exists at the point of jumps. This is due to the fact that we are considering combinatorial optimization problems, as a consequence if the predicted costs are infinitesimally increased, we will have two scenarios: in the first one the computed solution will be the same as before, as a consequence the gradient will be zero, while in the other case, the predicted solution will change and for that we will have a point of jump. In order to deal with this problem, they simplify the situation by considering the linearization f of $\mathcal{L}$ at the point $v^*(\hat{y})$.



(a) Small value of λ

(b) High value of λ

Figure 2: Continuous interpolation of a piece-wise constant function. 2a has only two incomplete interpolators: g_1 and g_2. 2b has most of the interpolators incomplete, but also a δ-interpolator g_3 which attains the value f(s$_1$) δ-aways from the set P_1. Despite losing some local structure for high λ, the gradient f_λ is still informative. Source Vlastelica et al. (2019)

$$f(s) = \mathcal{L}(\hat{s}) + \frac{d\mathcal{L}}{ds}(\hat{s})(s - \hat{s}) \tag{17}$$

As a consequence, we are going to have:

$$\frac{df(s(y))}{dy} = \frac{d\mathcal{L}}{dy} \tag{18}$$

Then, we need $s_\lambda(y)$, which is a solution of a perturbed optimization problem.

$$s_\lambda(y) = \arg\min_{s \in S}(s \cdot y + \lambda f(s)) \tag{19}$$

From that, we are able to compute $f_\lambda(y)$, a continuous interpolation of $f(s)$.

$$f_\lambda(y) = f(s_\lambda(y)) - \frac{1}{\lambda}[y \cdot s - y \cdot s_\lambda(y)] \tag{20}$$

Now, we are able to compute $\nabla f_\lambda(y)$, since f_λ is differentiable and it can be formulated as:

$$\nabla f_\lambda(y) = -\frac{1}{\lambda}[s - s_\lambda(y)] \tag{21}$$

Finally, we can use it as gradient of the selected loss, as shown below.

Algorithm 3 Forward pass for Blackbox

```
function ForwardPass($\hat{y}$)
    $\hat{s} = v^*(\hat{y})$
    save $\hat{y}$ and $\hat{s}$ for backward pass
    return $\hat{s}$
end function
```

Algorithm 4 Backward pass for Blackbox

```
function BackwardPass($\frac{d\mathcal{L}}{d\hat{s}}(\hat{s}), \lambda$)
    load $\hat{y}$ and $\hat{s}$ from forward pass
    $y' = \hat{y} + \lambda \cdot \frac{d\mathcal{L}}{d\hat{s}}(\hat{s})$
    $s_\lambda = v^*(y')$
    $\nabla_w f_\lambda(\hat{y}) = -\frac{1}{\lambda}(\hat{s} - s_\lambda)$
    return $\nabla_w f_\lambda(\hat{y})$
end function
```

By adding, only one additional hyper-parameter λ, which controls the trade-off between the "informativeness of the gradient" and "faithfulness to the original function", we are able to consider the optimization problem during the learning process, however the same issue of the SPO approach is also here. For each element of our dataset, it will be necessary to call the solver twice and, as before, this will lead to a high training time.

The main difference between the SPO approach and the Blackbox approach, is that SPO introduces the optimization problem, by using a custom loss, while Blackbox uses only a custom backward pass, as a consequence different losses can be used with the second one. In our case, we used the Hamming distance or the regret.

3.2.3.1 Hamming distance

One possible suitable loss is the Hamming distance between the predicted solution $\hat{s}$ and the target s. It basically computes the number of positions at which the elements are different. We will use a linear relaxation of it, as Vlastelica et al. (2019) do in their paper, since it would be not differentiable.

$$H(s, \hat{s}) = \frac{\sum_{i=1}^{n}(\hat{s}_i(1 - s_i) + (1 - \hat{s}_i)s_i)}{n} \tag{22}$$

3.2.3.2 Regret

Another possible candidate to use, is the regret. It is also possible to simplify the forward and backward pass for it. In fact given the regret formula:

$$regret(y, \hat{y}) = y^T(v^*(\hat{y}) - v^*(y)) \tag{23}$$

We need to compute $\frac{d\mathcal{L}}{ds}(\hat{s})$ which is simply y. As a consequence, the forward and backward pass can be rewritten as:

Algorithm 5 Forward pass for Blackbox with regret as loss

```
function ForwardPass($\hat{y}$)
    $\hat{s} = v^*(\hat{y})$
    $s_\lambda = v^*(\hat{y} + \lambda y)$
    save $\hat{s}$ and $s_\lambda$ for backward pass
    return $\hat{s}$
end function
```

Algorithm 6 Backward pass for Blackbox with regret as loss

```
function BackwardPass($\frac{d\mathcal{L}}{d\hat{s}}(\hat{s}), \lambda$)
    load $\hat{s}$ and $s_\lambda$ from forward pass
    $\nabla_w f_\lambda(\hat{y}) = -\frac{1}{\lambda}(\hat{s} - s_\lambda)$
    return $\nabla_w f_\lambda(\hat{y})$
end function
```

3.3 Cache

As already mentioned before, the biggest concern of the SPO and Blackbox method, is that, in order to consider the optimization problem, during the learning process, it is necessary to call the solver for each element, leading to higher training time with respect to the Two-stage approach.

The paper written by Mulamba et al. (2020) deals with this issue, by introducing a caching scheme. If we suppose to have available a cache filled with feasible solutions, we can actually replace the solver with it. Instead of solving the optimization problem, it will be only necessary to find the solution in the cache, which maximizes or minimizes the objective function. As a consequence, this approach will lower the training time.

This technique can be applied for both SPO and Blackbox but, instead of always calling the solver, we are going to have an additional parameter p_{solve}, indicating the probability of calling the optimization model or the cache. For each element of our data-set, we are going to sample a number between 0 and 100, if is it below p_{solve} then the solver is called and a new solution will be generated and stored in the cache. In the other case, we are going to simply do an argmin/argmax over the list of solutions. The cache is initialized with the optimal solutions of the data-set.

As explained in the paper, the results in terms of regret for $p_{solve} = 100$ and $p_{solve} = 5$ are similar, however for $p_{solve} = 5$ there is a significant reduction of the computational time, since the solver is called less times.

By introducing the caching scheme, new questions rise, for example, if we want to additionally decrease the amount of solver calls, it would be necessary to decrease even more the p_{solve} parameter, but this can effect the performances in terms of regret. So it is important to understand when we should sample from the cache and, in particular, finding how we should initialize the cache.

In order to do that, first of all, it is necessary to understand how the growth parameter p_{solve} affects the learning process; then by observing in depth the evolution of the cache, it would be possible to understand which new solutions are added and how relevant they are. From this knowledge we should be able to develop a smart sampling approach or a particular initialization, which ideally would lead us to have a model which does not rely on the solver.

4 Effect of the growth parameter

4.1 Introduction

To deal with the high training time, the authors Mulamba et al. (2020) proposed a caching scheme for the solutions. As a consequence, during the training process, instead of solving the real optimization problem, we can use the inner approximation associated to the caching scheme, leading to lower training time. We can say that there will be a trade-off between exploitation (when we are going to use the cache) and exploration (when we are going to call the solver). This trade-off will be dictated by the growth parameter p_{solve}.

What we want to understand in this section is how this hyper-parameter affects network performances and cache evolution in order to understand if is possible to balance the exploitation steps with the exploration steps. The final aim is to reduce even more the solver calls. Two problems have been considered: the knapsack problem and the shortest path problem, together with the methods explained previously.

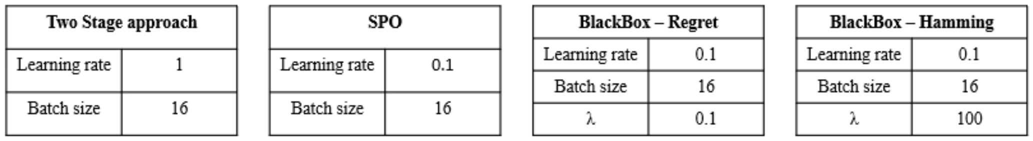
4.2 Shortest path problem

4.2.1 Experimental setup

The considered problem is the one explained in section 3.1.1. The data-set is generated by following the method explained in section 3.1.1.1. In particular, as explained by Elmachtoub and Grigas (2017), in order to observe a difference between the Two-stage approach and the other approaches, it is necessary to set the degree parameter to a high value. In fact, our network will consist only of a linear layer and, since the generation process is non-linear, it will be difficult for the Two-stage approach to find the proper weights. By increasing deg, we are going to increase the model misspecification; as a consequence the Blackbox and SPO approaches will obtain better results with respect to the Two-stage approach since they are more capable of handling non-linear signals. The noise term ϵ was set to 0.5, while n, the amount of generated data to 100.

After generating a training set of 100 elements, a validation set of 25 elements and a test set of 50 elements, we did an hyper-parameter search over the validation set, in order to obtain the best performances.

The network will consist only of a linear layer, made up of five neurons (as the number of feature) and forty output neurons (as the number of arcs). ADAM (Kingma and Ba (2014)) was the selected optimizer. All the methods were implemented with Pytorch (Paszke et al. (2019)) and Gurobi (Gurobi Optimization, LLC (2022)).



Two Stage approach	
Learning rate	1
Batch size	16

SPO	
Learning rate	0.1
Batch size	16

BlackBox – Regret	
Learning rate	0.1
Batch size	16
λ	0.1

BlackBox – Hamming	
Learning rate	0.1
Batch size	16
λ	100

.

Figure 3: Hyper-parameters for shortest path methods

4.2.2 Experiment

A set of values for p_{solve} was considered in order to understand how it affects the training process and the performances, in particular $p_{solve} \in [0, 5, 50, 90, 100]$. For each of these values, we trained a model five times and we computed the average and the standard error of the test regret for each epoch; the cache was initialized by using the target solutions. The considered methods are all the one shown before.

Since we also want to uncover how p_{solve} affects the evolution of the caching scheme, we generated the caching history. It is a file showing some information of each solution inside the cache, like: in which epoch it was generated and how many times it was picked. For SPO we are going to have only one file, containing this information since, the solver is called only once. While for both the Blackbox methods, two different caching histories are needed, since the solver/cache will be called twice, one time for $\hat{s}$ and one time for s_λ. By doing this we expect to have more insight about what is happening, internally, during the training process.

Down below, the final results are shown.

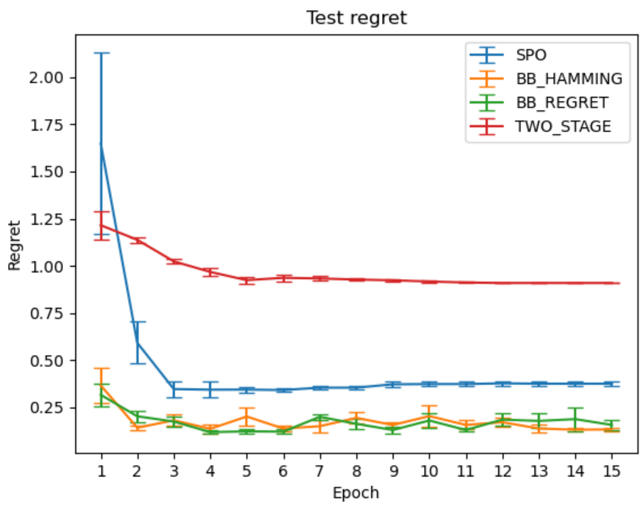


Figure 4: Comparison of the methods for the shortest path problem

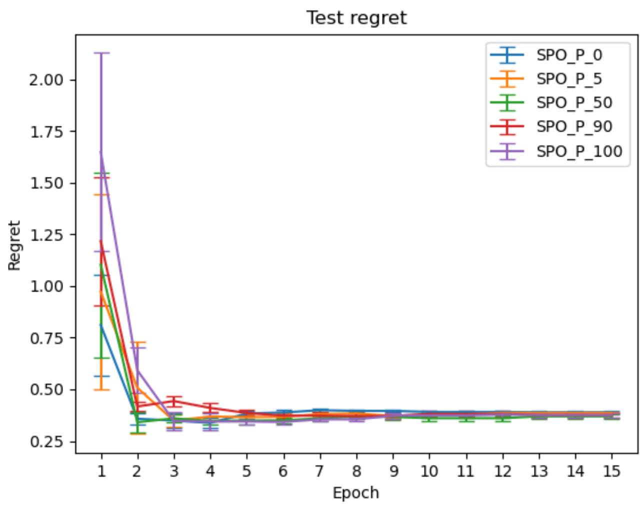


Figure 5: Comparison of SPO with different p_{solve}

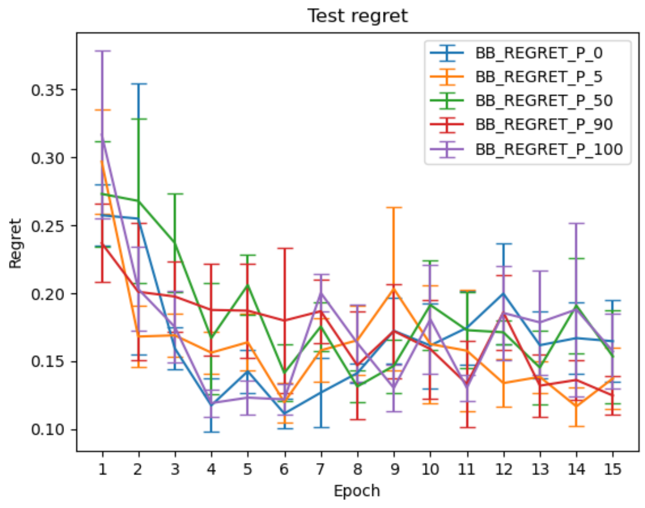


Figure 6: Comparison of Blackbox regret with different p_{solve}

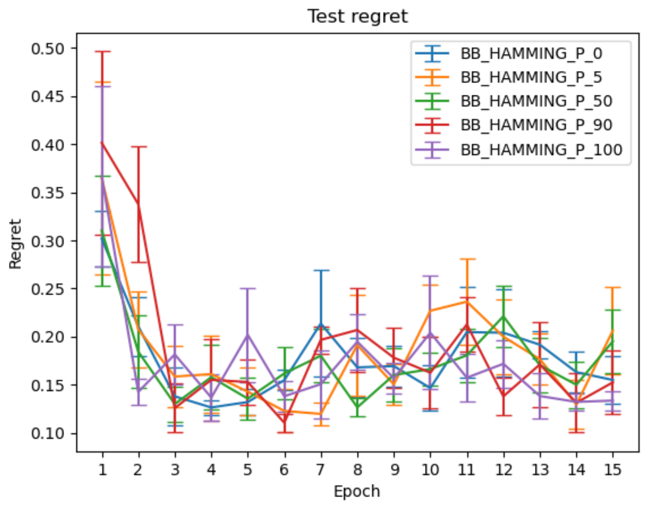


Figure 7: Comparison of Blackbox hamming with different p_{solve}

4.2.3 Discussion

As you can see from the figure 4, SPO and Blackbox methods are able to outperform the Two-stage approach, as expected. However one interesting fact, is that there is no significant difference in terms of regret for different p_{solve}, consequentially we can actually use only $p_{solve} = 0$ and we would still get a suitably trained model. For each of these methods there is a proper explanation.

4.2.3.1 SPO

By checking the evolution of the cache, it is possible to understand the reason for this behaviour. The image 8 represents the caching history for all the epochs for $p_{solve} = 100$. This means that the obtained solutions by calling the solver, are always the same for different epochs. This is due to the fact that, in order to compute the sub-gradient, it is needed to compute $v^*(2\hat{y} - y)$. Since we set deg, an hyper-parameter of the data generation, equals to 8, the real costs will lay in a big interval, as a consequence the prediction y will act as noise most of time, since the model is not able to reach the real costs. However the model will still learn, since $v^*(2\hat{y} - y)$ will find the shortest path for $-y$, so the highest costs will become the lowest, leading to a different problem with respect to the original one.

One would be inclined to think that the model with $p_{solve} = 100$ would obtain better results, since the optimal solutions are used. However the main problem with this setting is that, we can not expect to obtain very different solutions from those stored in the cache (in terms of number of ones), due to how the problem is defined. In fact only 70 solutions are possible, as a consequence it is most likely to have already inside the cache most of them. This can also explain the fact that, even if with $p_{solve} = 100$, new solutions are added only during the first epoch, meaning that most of the time, the cache has already the desired solution. By also knowing that 75% of the time the solution picked is one from the initialization, we can understand why it is not needed to obtain new solutions. However, things will be different for the knapsack problem.

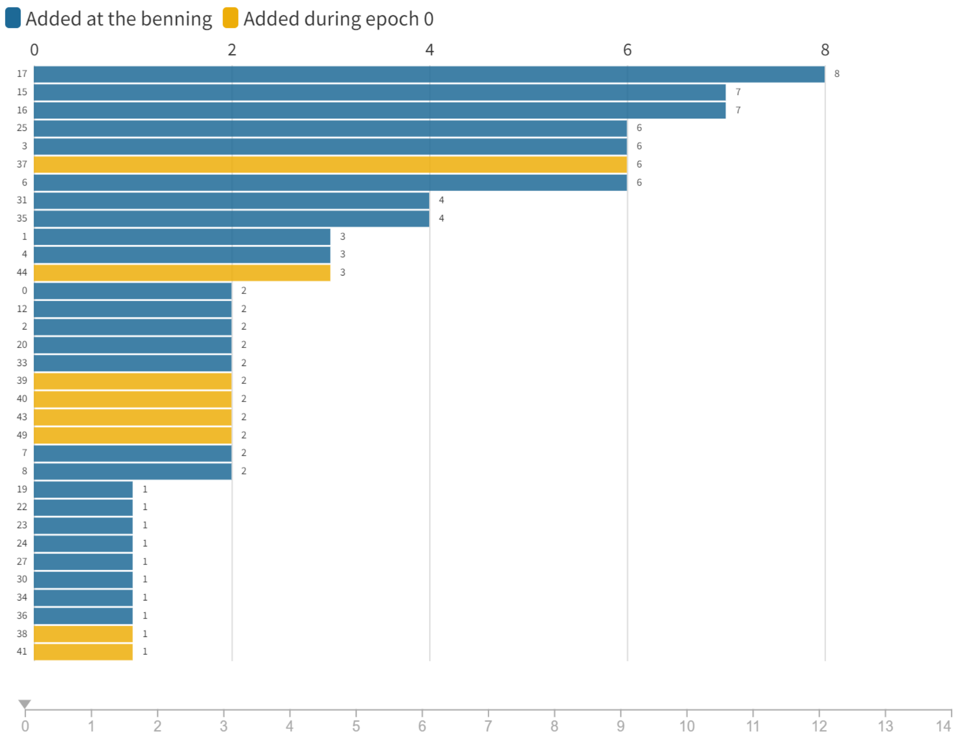


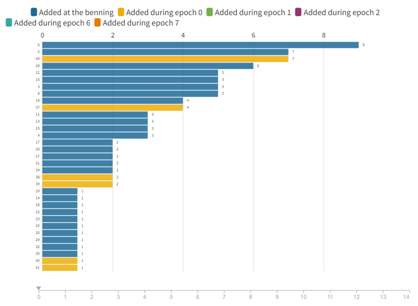
Figure 8: Cache evolution for all the epochs for $p_{solve} = 100$, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

4.2.3.2 Blackbox - Regret

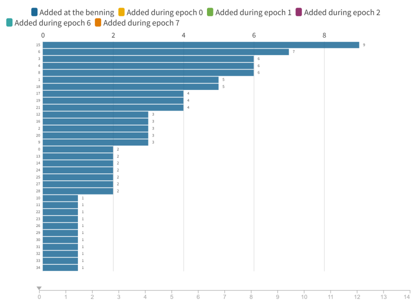
As already done for the SPO method in the previous section, we can check the caching history of the model in order to understand why p_{solve} does not affect the learning process in these settings. Since the Blackbox methods require to call the solver two times, we generated the caching history for both the calls, for $\hat{s}$ and s_λ.

As you can see from the images 9, 10, 11, 12, if we focus our attention on s_λ, we can see that the distribution is always the same, which means that for each epoch the same solutions are used. There is an explanation: s_λ is computed as $v^*(\hat{y} + \lambda y)$, however as already explained in the section 4.2.3.1, y will be incomparable in terms of magnitude most of time, since we set deg = 8, for generating the data. Even if we have λ that decreases the values of y, it is not enough. As a consequence we will have a similar behaviour to that of the SPO method, since $\hat{y}$ will act as noise with respect to y, unable to reach those values. In the end, by knowing this, we can immediately understand that $v^*(\hat{y} + \lambda y)$ will return the real solution s, so our prediction $\hat{y}$ does not have any influence.

The only part where $\hat{y}$ influences the solutions, is for $\hat{s}$, since $\hat{s} = v^*(\hat{y})$, however as you can see down below, during the training process most of the time the real solutions are picked, as a consequence we can actually rely only on them and obtaining the same result of a model with $p_{solve} = 100$.

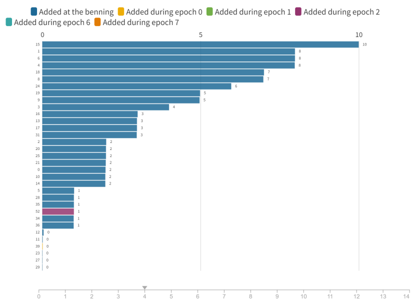


(a) Cache evolution for $\hat{s}$

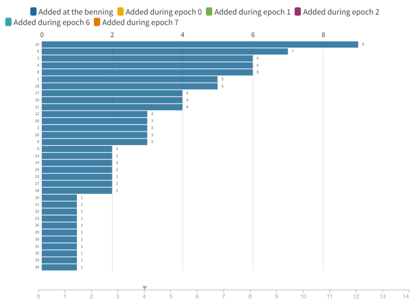


(b) Cache evolution for s_λ

Figure 9: Cache evolution for $\hat{s}$ and s_λ for $p_{solve} = 100$ for the first epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

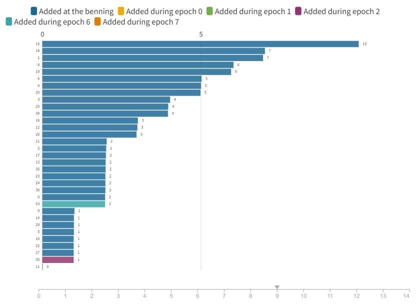


(a) Cache evolution for $\hat{s}$

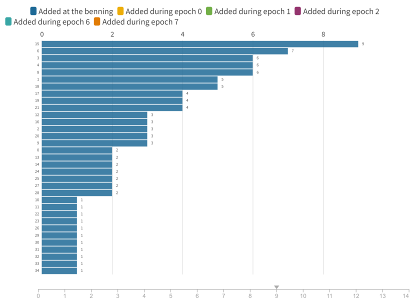


(b) Cache evolution for s_{λ}

Figure 10: Cache evolution for $\hat{s}$ and s_{λ} for $p_{solve} = 100$ for the fifth epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

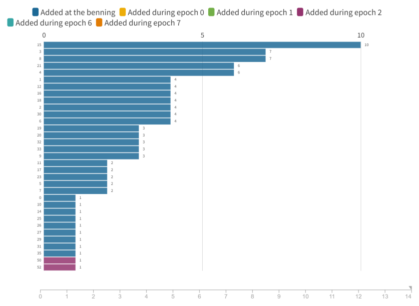


(a) Cache evolution for $\hat{s}$

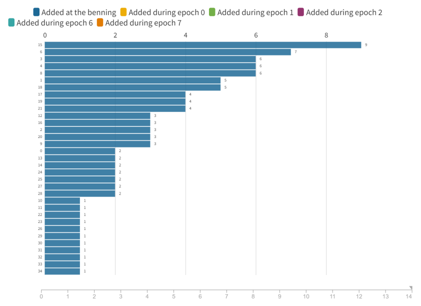


(b) Cache evolution for s_{λ}

Figure 11: Cache evolution for $\hat{s}$ and s_{λ} for $p_{solve} = 100$ for the tenth epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked



(a) Cache evolution for $\hat{s}$



(b) Cache evolution for s_λ

Figure 12: Cache evolution for $\hat{s}$ and s_λ for $p_{solve} = 100$ for the fifteenth epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

4.2.3.3 Blackbox - Hamming

Also here, in order to understand why p_{solve} does not affect the training process, the caching history was checked. Since we are using a Blackbox method, two caching history are used, one for s_λ and one for $\hat{s}$. If we take a look at the figures 13, 15 and 16, we can see that we are obtaining the same behaviour of the Blackbox with the regret as loss, in fact the distribution for s_λ does not change through the epochs.

s_λ is computed as $v^*(y')$, where $y' = \hat{y} + \lambda \cdot \frac{d\mathcal{L}}{d\hat{s}}(\hat{s})$. In order to compute $\frac{d\mathcal{L}}{d\hat{s}}$ it is necessary to compute the sub-gradient of the linear Hamming loss (shown in the equation 22) for each $\hat{s}_i$, the elements of the predicted solution. It is possible to notice that:

$$\frac{d\mathcal{L}}{d\hat{s}_i} = \frac{d\left(\frac{(\hat{s}_i(1-s_i)+(1-\hat{s}_i)s_i)}{n}\right)}{d\hat{s}_i} = \frac{1-2s_i}{n} \tag{24}$$

As a consequence $\frac{d\mathcal{L}}{d\hat{s}}$ will consist of a vector dependent only on s_i, in fact, we can simplify the formula without considering n and we will obtain:

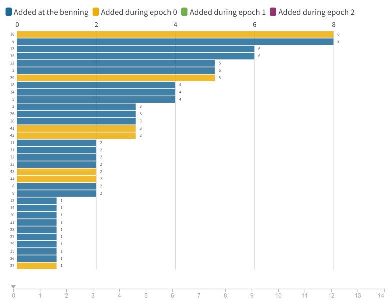
$$\frac{d\mathcal{L}}{d\hat{s}_i} = \begin{cases} 1 & \text{if the edge i is not traversed} \\ -1 & \text{if the edge i is traversed} \end{cases} \tag{25}$$

Now, if we plug it in y' we can immediately notice, that we will obtain the same behaviour of the Blackbox model with regret as loss, shown in the section 4.2.3.2. In fact given the formula $y' = \hat{y} + \lambda \cdot \frac{d\mathcal{L}}{d\hat{s}}(\hat{s})$ and by knowing that $\lambda = 100$ (obtained by running a grid search), we can notice that $\hat{y}$ will act as noise, since it is not comparable to $\lambda \cdot \frac{d\mathcal{L}}{d\hat{s}}(\hat{s})$. We can also deduct that s_λ will consist of the real solution s. Given that $s_\lambda = v^*(\lambda \cdot \frac{d\mathcal{L}}{d\hat{s}}(\hat{s}))$ and $\lambda = 100$, the solver has to solve a problem where:

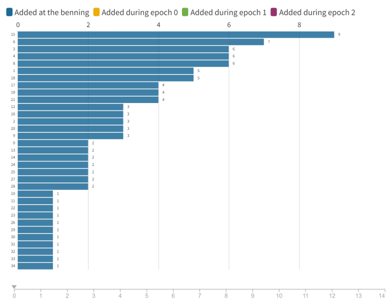
$$\lambda \cdot \frac{d\mathcal{L}}{d\hat{s}}(\hat{s}) = \begin{cases} 100 & \text{if the edge } i \text{ is not traversed in the solution } s \\ -100 & \text{if the edge } i \text{ is traversed in the solution } s \end{cases} \tag{26}$$

Which means that the traversed edges in the real solution will have a negative cost, while the not traversed ones will have a positive cost. It is obvious that the returned solution s_λ it is in reality s.

The only part where $\hat{y}$ influences the solutions, is for $\hat{s}$, since $\hat{s} = v^*(\hat{y})$, however as you can see down below, during the training process most of the time the real solutions are picked, as a consequence we can actually rely only on them and obtaining the same result of a model with $p_{solve} = 100$.

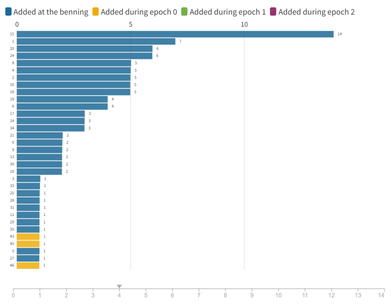


(a) Cache evolution for $\hat{s}$

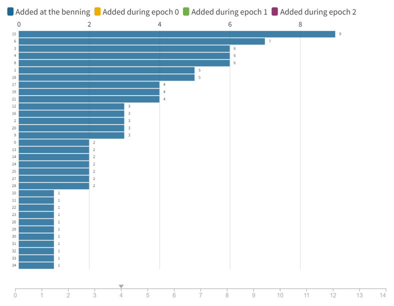


(b) Cache evolution for s_λ

Figure 13: Cache evolution for $\hat{s}$ and s_λ for $p_{solve} = 100$ for the first epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

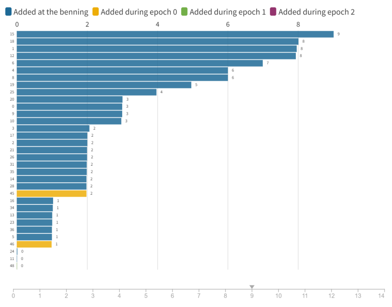


(a) Cache evolution for $\hat{s}$

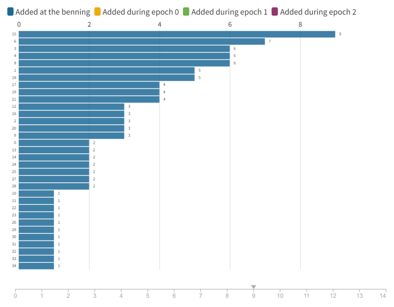


(b) Cache evolution for s_λ

Figure 14: Cache evolution for $\hat{s}$ and s_λ for $p_{solve} = 100$ for the fifth epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

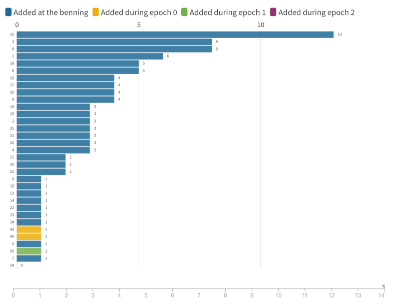


(a) Cache evolution for $\hat{s}$

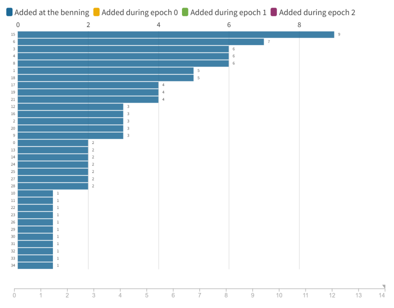


(b) Cache evolution for s_λ

Figure 15: Cache evolution for $\hat{s}$ and s_λ for $p_{solve} = 100$ for the tenth epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked



(a) Cache evolution for $\hat{s}$



(b) Cache evolution for s_λ

Figure 16: Cache evolution for $\hat{s}$ and s_λ for $p_{solve} = 100$ for the fifteenth epoch, the numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

4.3 Knapsack problem

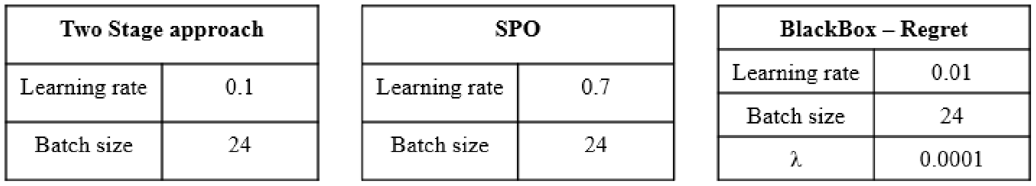
4.3.1 Experimental setup

The problem is the one explained in the section 3.1.2. Several capacities have been considered, in particular: 60, 120 and 180.

Speaking about the data-set, for this experiment, we decided to consider the Energy-price dataset, explained in section 3.1.2.1. It was split into three sets: training, validation and test. In order to obtain the best performances, an hyper-parameter search over the validation set was needed.

About the Blackbox approaches, we decided to apply only the one that uses the regret as loss, since, as you can see from the section 4.2.3.3 and 4.2.3.2, both of them have a similar behaviour, as a consequence we expect to obtain similar results.

The network will consist only of a linear layer, made up of eight input neurons (as the number of features) and one output neuron (the predicted cost). ADAM (Kingma and Ba (2014)) was the selected optimizer. All the methods were implemented with Pytorch (Paszke et al. (2019)) and Gurobi (Gurobi Optimization, LLC (2022)).



Two Stage approach	
Learning rate	0.1
Batch size	24

SPO	
Learning rate	0.7
Batch size	24

BlackBox – Regret	
Learning rate	0.01
Batch size	24
λ	0.0001

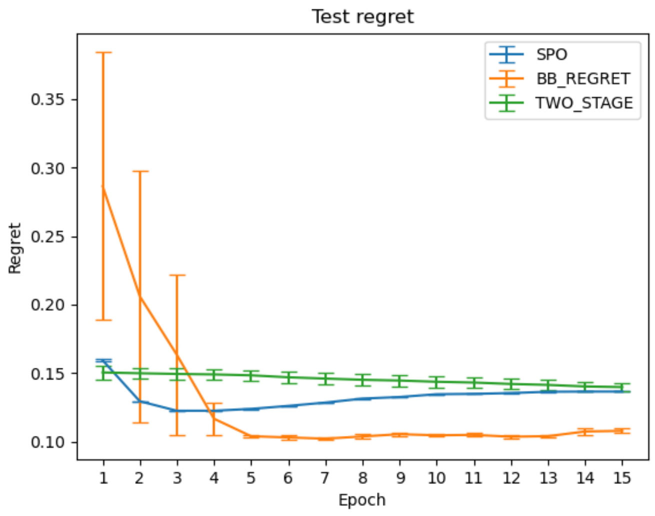
.

Figure 17: Hyper-parameters for knapsack methods

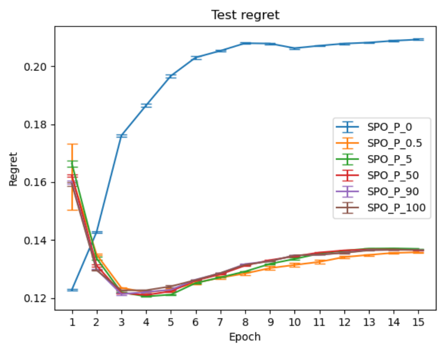
4.3.2 Experiment

In order to understand how p_{solve} affects the training and the performances, we decided to consider a set of values for it, in particular: $p_{solve} \in [0, 0.5, 1, 5, 90, 100]$. For each of these values, we trained a model five times and we computed the average and the standard error of the test regret for each epoch. The cache was initialized by using the target solutions. The results are also compared with the one of Two-stage approach, in order to show the potential of SPO and Blackbox approaches.

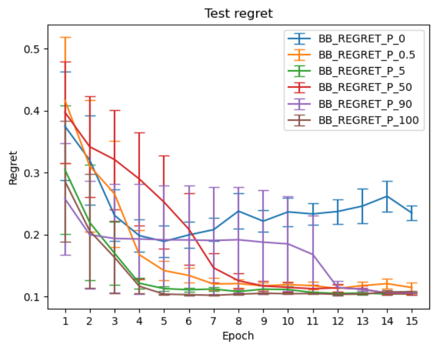
Since we also want to discover, how p_{solve} influences the caching scheme and its evolution, we also generated the caching history, a file containing for each solution, when it was generated and, how many times it was picked given an epoch.



(a) Comparison of the methods

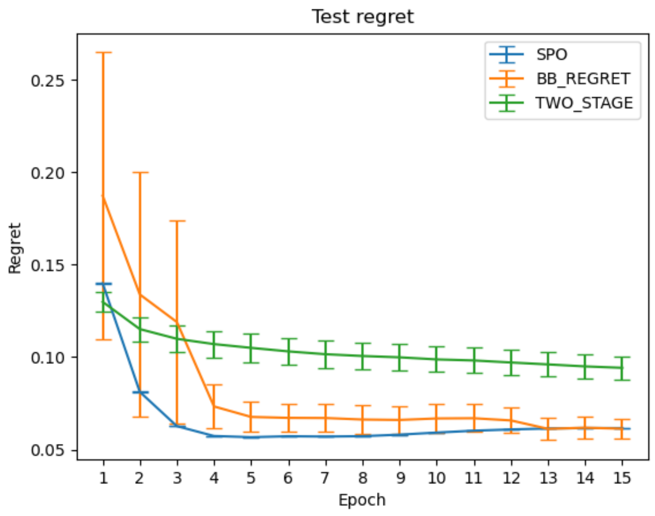


(b) Comparison of SPO with different p_{solve}

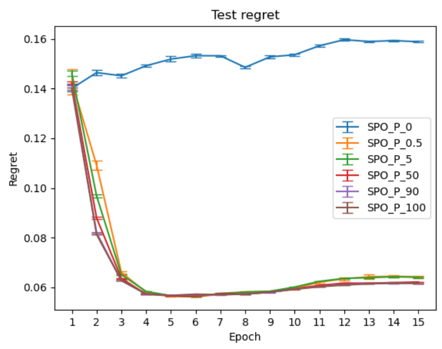


(c) Comparison of Blackbox regret with different p_{solve}

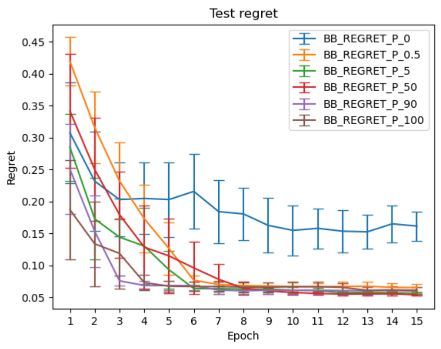
Figure 18: Comparison of methods for the knapsack problem with capacity = 60



(a) Comparison of methods

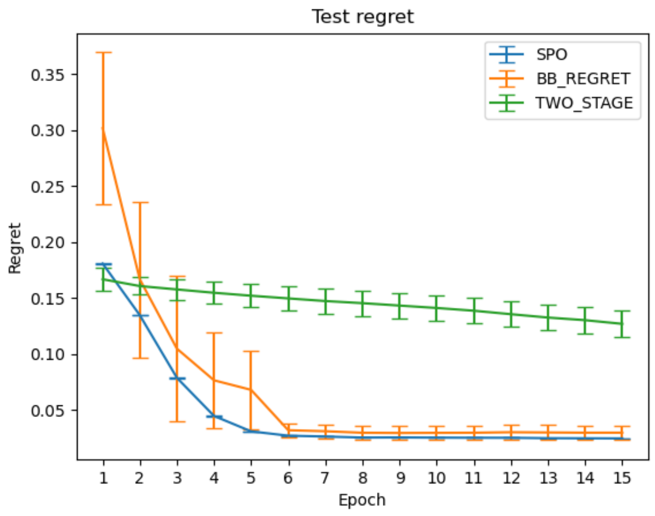


(b) Comparison of SPO with different p_{solve}

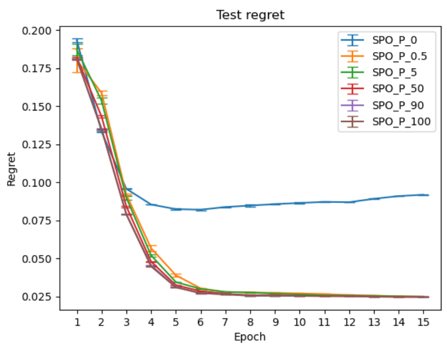


(c) Comparison of Blackbox regret with different p_{solve}

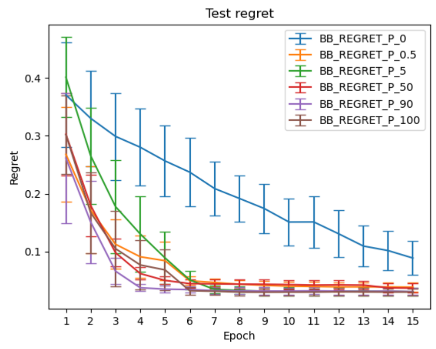
Figure 19: Comparison of methods for the knapsack problem with capacity = 120



(a) Comparison of methods



(b) Comparison of SPO with different p_{solve}



(c) Comparison of Blackbox regret with different p_{solve}

Figure 20: Comparison of methods for the knapsack problem with capacity = 180

4.3.3 Discussion

As you can see from the figure 18a,19a and 20a, SPO and Blackbox methods are able to generate a better model with respect to the Two-stage approach. Also another interesting fact is that even if we set p_{solve} to 0.5, which means that only 0.5 % of the times, the solver is called, we are able to obtain a model which is comparable to the one obtained with $p_{solve} = 100$. However if we set it to 0, the performances become worse. In the next chapter we are going to uncover these causes.

4.3.3.1 SPO

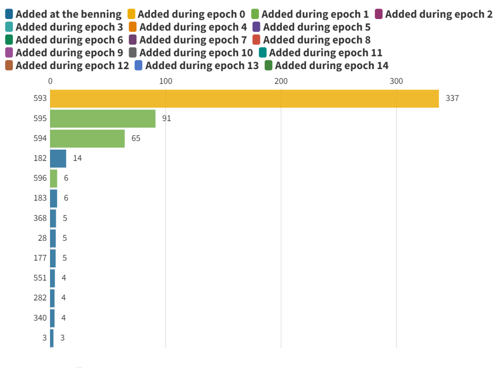
In order to understand why there is no significant difference between $p_{solve} = 0.5$ and $p_{solve} = 100$ the caching history must be checked. Figure 21 and 22, represent respectively the caching history during the first, third and fourth epoch for $p_{solve} = 0.5$ and $p_{solve} = 100$. We show only the graph related to capacity = 60, since also the other models, related to the different capacities, share the same behaviour.

As you can see until the third epoch, only one solution is picked most of the time, this solution is the one composed only of zeros. This is due to the fact that, we need to compute $v^*(2\hat{y}_i - y_i)$, as depicted in the equation 15, however we encounter a situation similar to the one of the shortest path problem. In this case, as before, our predictions are incomparable in terms of magnitude to the target, as a consequence $2\hat{y}_i - y_i$ will turn out to be $-y_i$. It is obvious that passing these values to the solver will return a solution of only zeros, since maximizing the value of a knapsack with negative objects, means to not pick any of them.

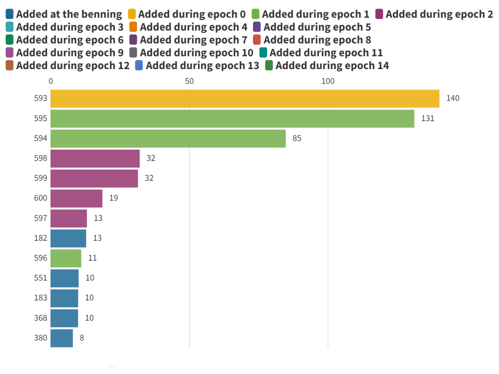
Even if, the probability of calling the solver is very low, on the contrary, the probability of obtaining a solution of only 0 is really high during the firsts epochs. After being obtained, it will basically be used most of the time, as a consequence this leads to have a similar behaviour between $p_{solve} = 0.5$ and $p_{solve} = 100$.

It is also explainable why the model related to $p_{solve} = 0$ gives worse results. The main problem here is that, in the initialized cache, we have only the solutions for the targets, which means that a solution of only 0 is not present. Since the solutions in the cache are very different from it (in terms of number of ones), it is obvious that the model is not able to learn properly.

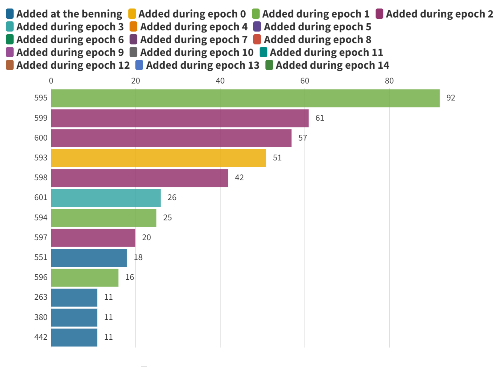
Also the last interesting fact is that, the learning process seems to converge around the $4^{th} - 5^{th}$ epoch, as shown in the figure 18b, 19b and 20a. Which means that, the solution of only zeros takes an important role, in the learning process for the SPO method. In the chapter 5 we are going to verify this statement.



(a) Cache evolution for the second epoch

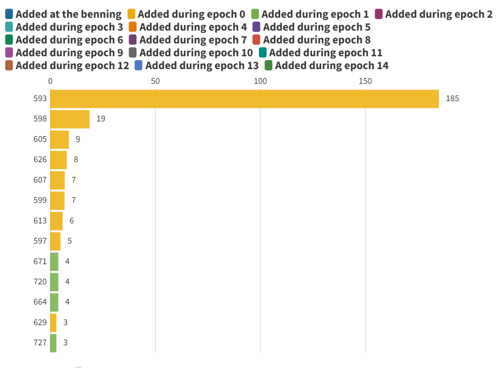


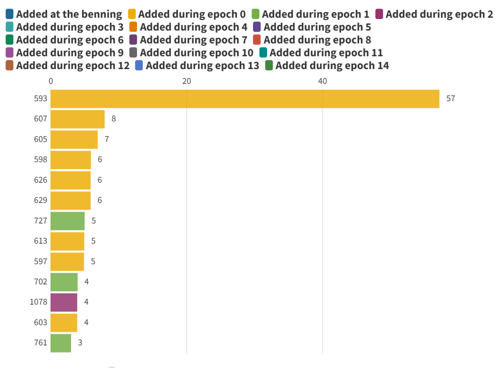
(b) Cache evolution for the third epoch



(c) Cache evolution for the fourth epoch epoch

Figure 21: Cache evolution of SPO with $p_{solve} = 0.5$ and capacity = 60 for the second, third and fourth epoch. The numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked


(a) Cache evolution for the second epoch


(b) Cache evolution for the third epoch

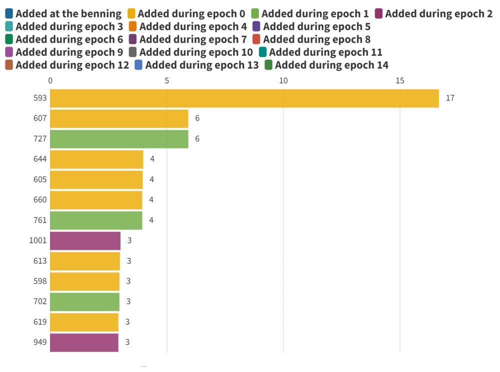

(c) Cache evolution for the fourth epoch epoch

Figure 22: Cache evolution of SPO with $p_{solve} = 100$ and capacity = 60 for the second, third and fourth epoch. The numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

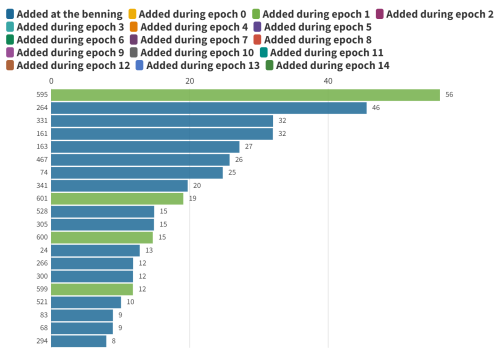
4.3.3.2 Blackbox - Regret

As you can see from the graphs below, the Blackbox method with regret as loss has a very different behaviour with respect to the SPO method. This can be noticed, from the figure 23 and 24, since in neither of them, the first picked solution is the one of only zeros, meaning that it does not have a lot of relevance.

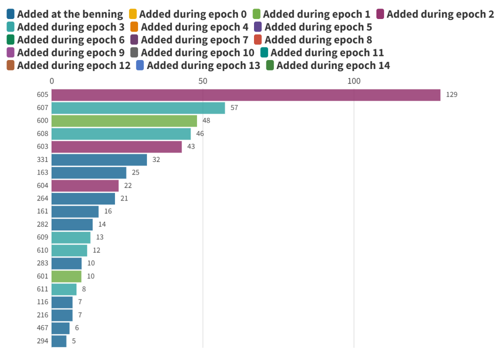
It has also a very different behaviour with respect to the Blackbox method for the shortest path problem, discussed in the section 4.2.3.2. Here the λ hyper-parameter is set to 10^{-5}, remember that is used for computing $s_\lambda = v^*(\hat{y} + \lambda y)$. Since it is very low, but at the the same time y » $\hat{y}$, λ balances the difference in terms of magnitude of the two terms.

Last but not least, it is important to notice that, even if we set $p_{solve} = 0.5$, the computed solutions are sufficient to obtain a properly trained model, as you can see from the figure 18c, 19c and 20c. Also, for both $\hat{s}$ and s_λ, some solutions obtained by the solver, are composed only of a small amount of ones. This is due to the fact that the predicted values can be negative, and if they are the majority, the solution will consists of all the objects with a positive cost, since the knapsack will not reach the maximum capacity.

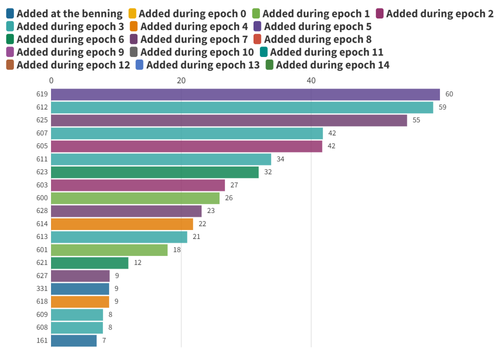
In the end, we were not able to find any indication of a possible relation between $\hat{s}$ and s_λ, therefore in the section 7, we are going to discuss if it is possible to find a connection between the predicted values and the solutions inside the cache; reasoning even more about why, with $p_{solve} = 0.5$, we were able obtains those promising results.



(a) Cache evolution for the second epoch

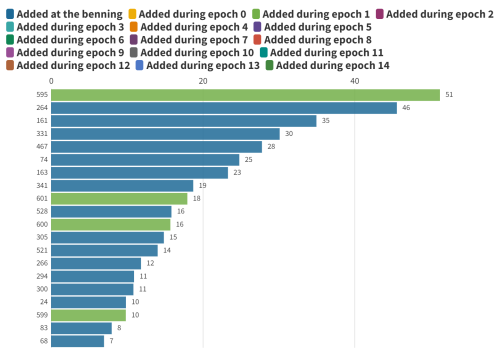


(b) Cache evolution for the fourth epoch

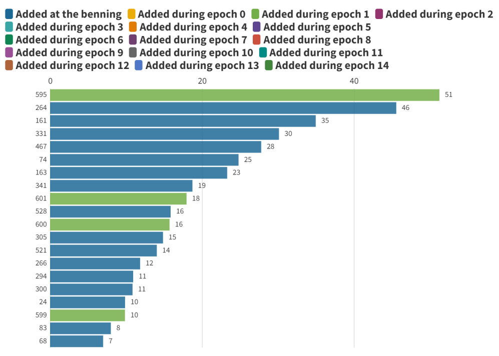


(c) Cache evolution for the eighth epoch

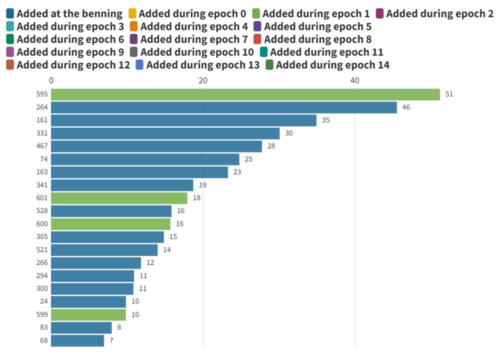
Figure 23: Cache evolution of $\hat{s}$ with $p_{solve} = 0.5$ and capacity = 60 for the second, third and eights epoch. The numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked



(a) Cache evolution for the second epoch

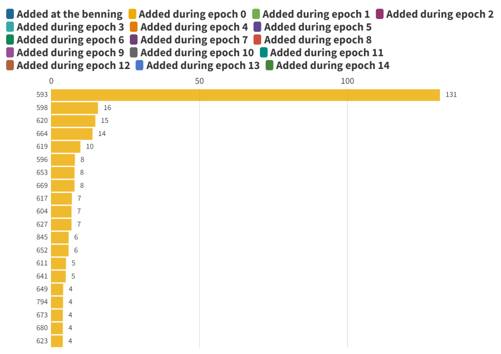


(b) Cache evolution for the fourth epoch

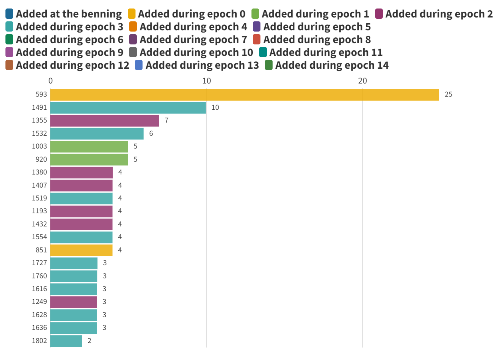


(c) Cache evolution for the eighth epoch

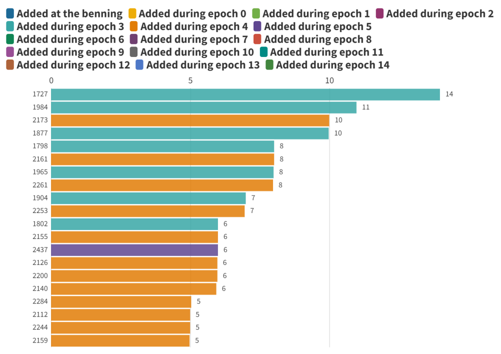
Figure 24: Cache evolution of s_λ with $p_{solve} = 0.5$ and capacity = 60 for the second, third and eighth epoch. The numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked



(a) Cache evolution for the second epoch

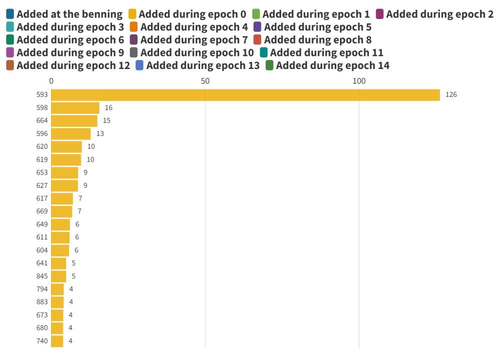


(b) Cache evolution for the fourth epoch

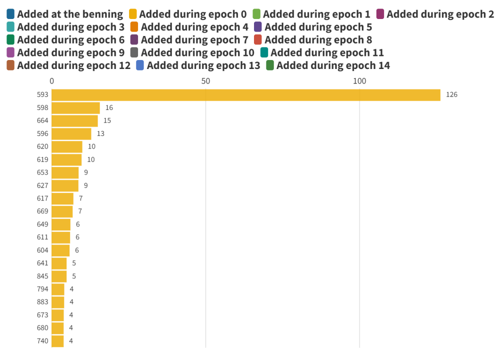


(c) Cache evolution for the eighth epoch

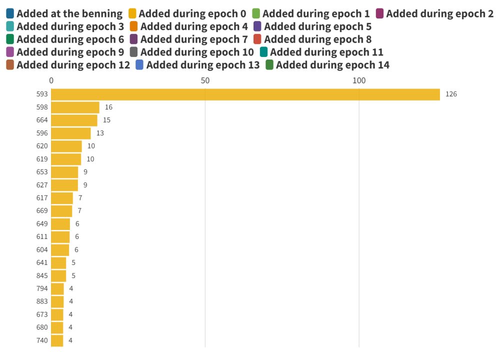
Figure 25: Cache evolution of $\hat{s}$ with $p_{solve} = 100$ and capacity = 60 for the second, third and eights epoch. The numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked



(a) Cache evolution for the second epoch



(b) Cache evolution for the fourth epoch



(c) Cache evolution for the eighth epoch

Figure 26: Cache evolution of s_λ with $p_{solve} = 100$ and capacity = 60 for the second, third and eighth epoch. The numbers on the y-axis are the id of the solutions, while the numbers at the end of the bars indicates how many times that solution is picked

5 Importance of new solutions for SPO

5.1 Introduction

As shown in the previous section, there is a high difference, for the SPO method, between $p_{solve} = 0$ and $p_{solve} = 100$. This means that it is necessary to not rely only on the target solutions from the cache. However, even if we set p_{solve} to 0.5, the obtained model results similar to the one with $p_{solve} = 100$. We even noticed that, in each model, the solution that is used mostly during the firsts epochs, is the one with only zeros. In this section we want to understand how important this solution is for the learning process and if also the others, have a strong effect like it.

5.2 Experimental setup

The experimental setup is the same explained in the section 4.3.1.

5.3 Experiment

With the aim of understating how important the solution with only zeros is, we examined an additional initialization of the cache. We filled it with the target solutions, together with one made only by zeros. With this initialization (INIT$_1$), we considered two SPO models, with $p_{solve} \in [0, 100]$. The final performances are obtained by training each model five times; the average and the standard error of the test regret is computed for each epoch.

The final results are compared with the ones obtained for the SPO models of the previous experiments. So, the only difference is in the initialization, before, it was filled only with the target solutions (INIT$_0$).

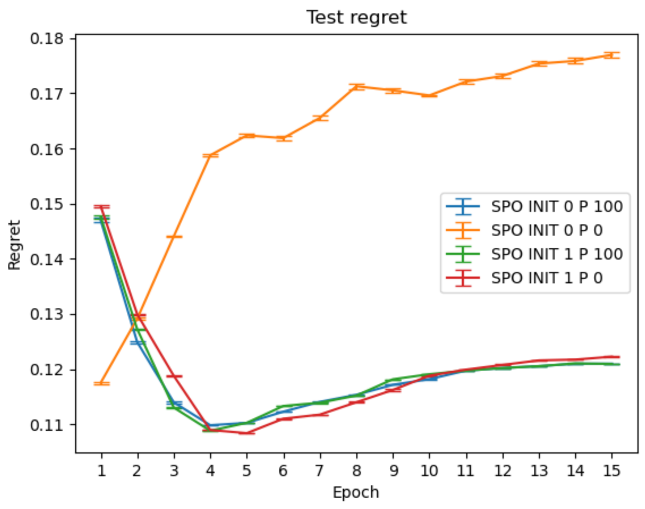


Figure 27: Comparison for SPO, with INIT$_0$ and INIT$_1$ and capacity = 60

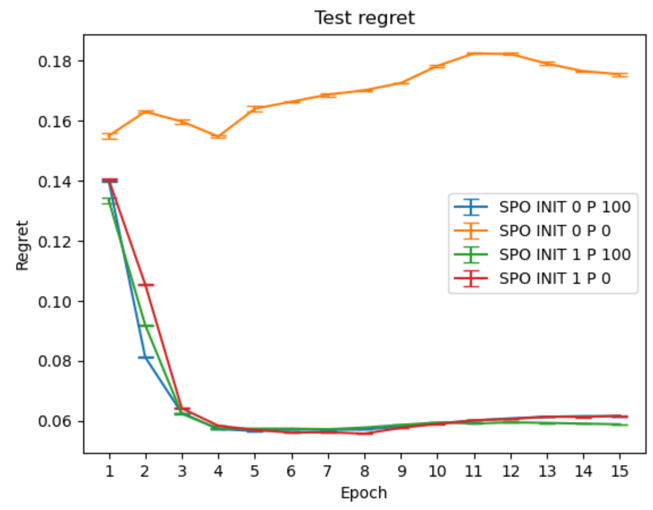


Figure 28: Comparison for SPO, with INIT_0 and INIT_1 and capacity = 120

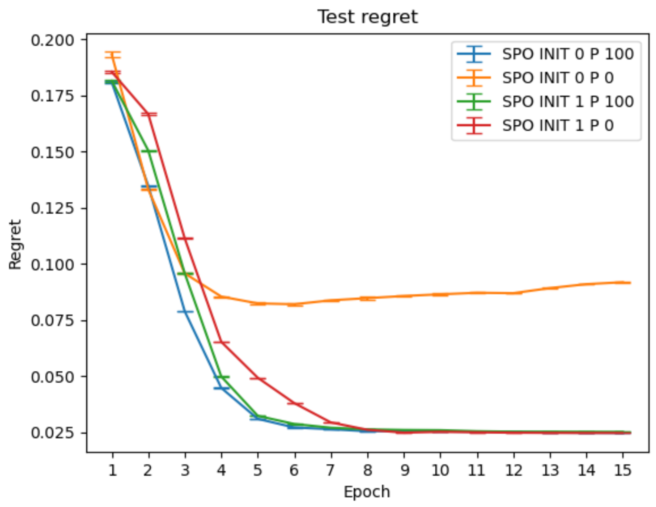


Figure 29: Comparison for SPO, with INIT_0 and INIT_1 and capacity = 180

5.4 Discussion

As you can see from the figures 27,28 and 29, the results in terms of regret are very similar for all the models, except for the one with INIT_0 and $p_{solve} = 0$. This means that it is not necessary to sample new solutions, as the one made only by zeros, is used most of the time.

One interesting fact, is that in the figure 29, the model with INIT_1 and $p_{solve} = 0$ reaches the minimum regret later with respect to the other models. Probably sampling new and different solutions leads to faster convergence. However we can clearly say that in these settings they are not important as the one of only zeros.

In section 6, we are going to bring these settings to the extreme, in order to understand if the solution from the initialization are actually needed, in order to obtain a properly trained model.

6 Exploitation of new solutions for SPO and Filter initialization

6.1 Introduction

In the previous section, we were able to prove that the solution, composed of only zeros, have a strong relevance in the learning process, since most of the time is the one picked from the cache or that the solver would return.

In this chapter, we want to investigate further the importance of this solution, by bringing the settings of the cache to an extreme condition: initializing it with only that solution.

In the end from the results obtained, we will be able to propose a new cache initialization where only two solutions are needed; last but not least it will be not necessary to call the solver. We define this initialization as the filter initialization.

6.2 Exploitation of new solutions

6.2.1 Introduction

The goal of this experiment is to understand if the cache initialization can be brought to extreme conditions and even returning a regret comparable to models with $p_{solve} = 100$, this will also help in understating if it exists a ”key element” whose absence, will lead to the computation of sub-optimal decisions and of course to higher regret.

For now, one of the candidates for being this ”key element” appears to be the solution made of only zeros. In fact if we consider two SPO model, one with $p_{solve} = 0$ and the other with $p_{solve} = 0.5$, there will be a high difference in terms of regret, as shown in figures 18b, 19b and 20b. Also in section 5, we were able to discover that, if the cache is initialized with the target solutions and one made only of zeros, it is not necessary to use the solver.

6.2.2 Experimental setup

The experimental setup is the same explained in the section 4.3.1.

6.2.3 Experiment

We considered an additional type of initialization defined as INIT_2. The cache is simply initialized by a solution made of only zeros. For this experiment we are going to consider one model, initialized with INIT_2, but with $p_{solve} = 0$. The final performances are obtained by training each model five times. The average and the standard error of the test regret are computed for each epoch.

The final results are also compared with those obtained from an SPO model, with INIT_1 (the cache is initialized with the target solutions and one made only of zeros) and $p_{solve} = 0$.

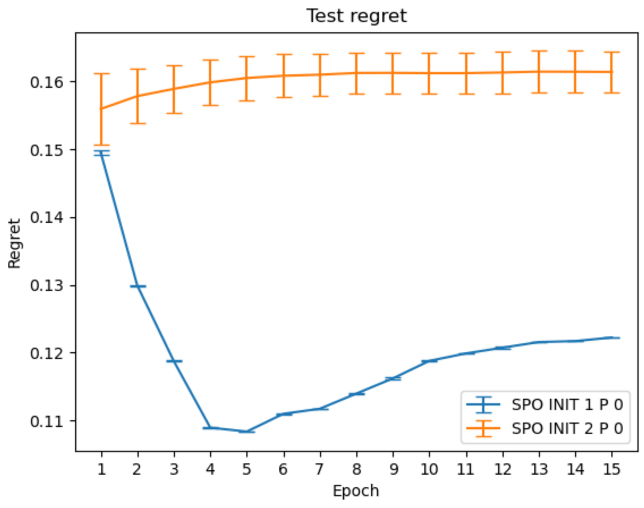


Figure 30: Comparison for SPO, with INIT_1 and INIT_2 and capacity = 60

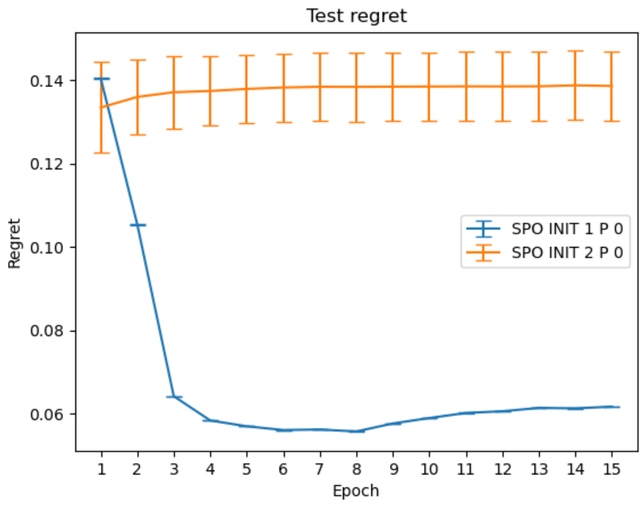


Figure 31: Comparison for SPO, with INIT_1 and INIT_2 and capacity = 120

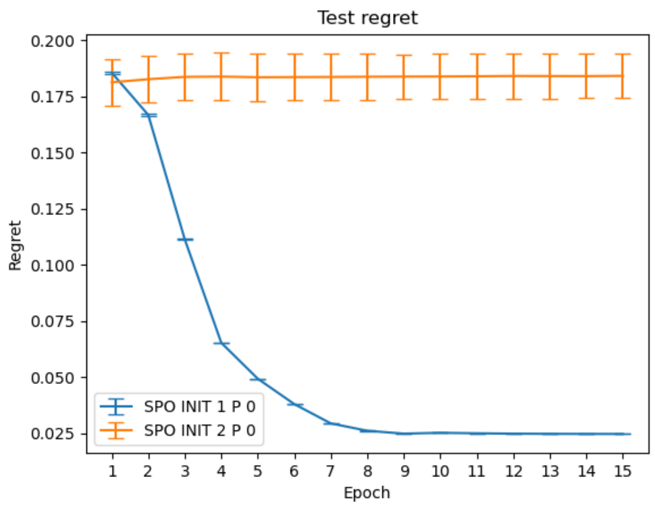
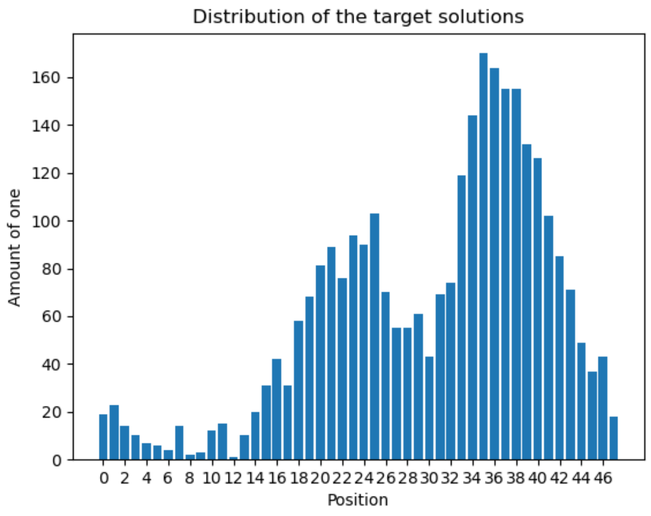


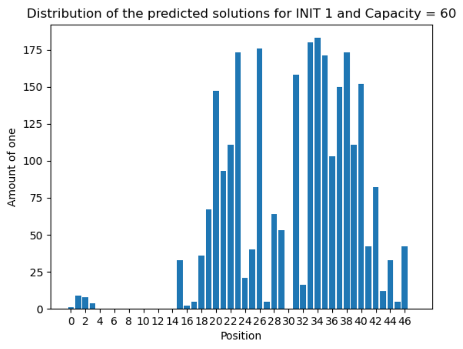
Figure 32: Comparison for SPO, with INIT$_1$ and INIT$_2$ and capacity = 180

6.2.4 Discussion

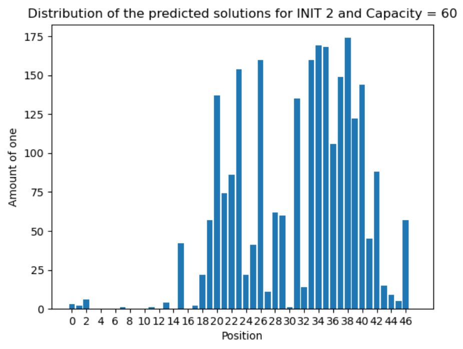
As you can see from the graphs 30, 31 and 32, the results are not the ones expected, since the model with INIT$_2$ and $p_{solve} = 0$ does not get even close, the performances of the other model. As a consequence we have to discard the idea that the solution made only of zeros is the key factor for making optimal decisions. In order to understand properly what is the difference between these models, we decided to focus our attention on the distribution of the solutions.



(a) Distribution of target solutions

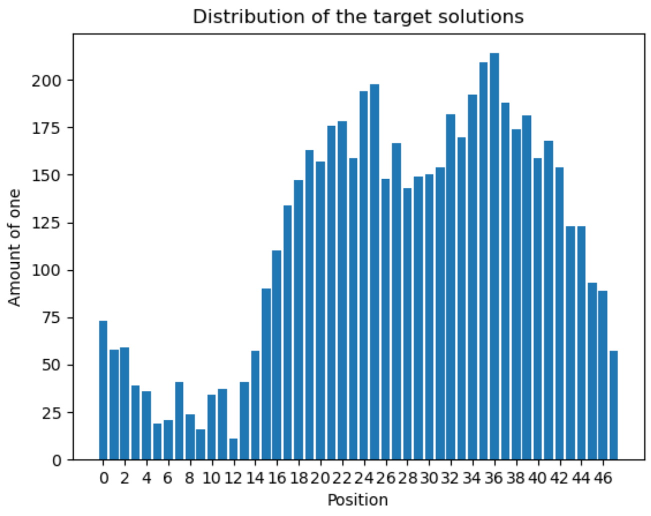


(b) Distribution of predicted solutions with INIT_1

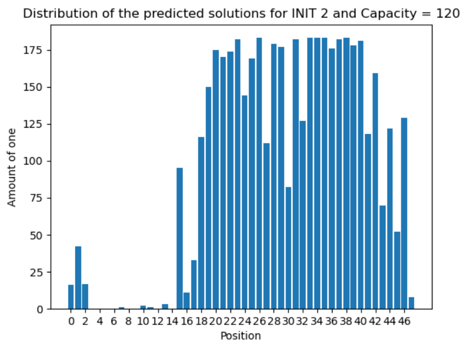


(c) Distribution of predicted solutions with INIT_2

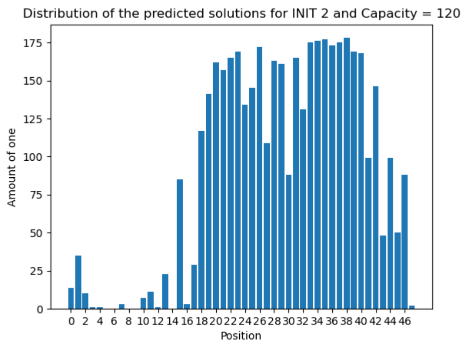
Figure 33: Distribution of solutions for capacity = 60. On top the distribution of target solutions, on the left the distribution of predicted solutions for INIT_1 and on the right the distribution of predicted solutions for INIT_2



(a) Distribution of target solutions

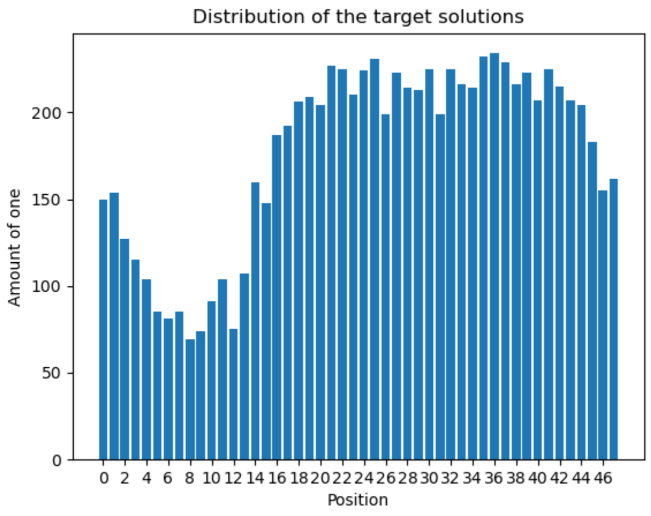


(b) Distribution of predicted solutions with INIT_1

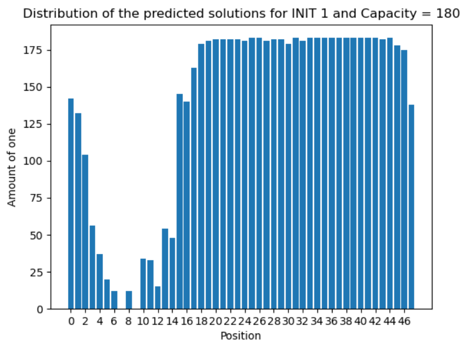


(c) Distribution of predicted solutions with INIT_2

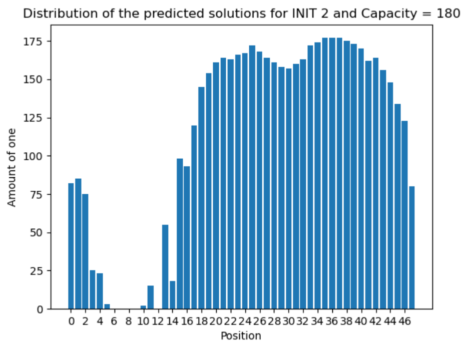
Figure 34: Distribution of solutions for capacity = 120. On top the distribution of target solutions, on the left the distribution of predicted solutions for INIT_1 and on the right the distribution of predicted solutions for INIT_2



(a) Distribution of target solutions



(b) Distribution of predicted solutions with $INIT_1$



(c) Distribution of predicted solutions with $INIT_2$

Figure 35: Distribution of solutions for capacity = 180. On top the distribution of target solutions, on the left the distribution of predicted solutions for $INIT_1$ and on the right the distribution of predicted solutions for $INIT_2$

First of all, it is possible to notice that, the distribution of the target solutions, for all the capacities, is not homogeneous, in fact after approximately the 14^{th} element, the amount of one tends to increase, probably due to a similar distribution of the energy costs.

Between the two initialization INIT_1 and INIT_2, it is not possible to notice a great difference in the distribution of the predicted solutions, for capacity = 60, 120; as shown in figure 33 and 34. However, things start to change for capacity = 180, as depicted in the figure 35. In fact it seems that the INIT_1 model is more capable of mimicking the target distribution between the 1^{st} and the 14^{th} element.

The cache, if initialized with INIT_2, will consist only of a solution: the one made only of zeros. If we recall the training algorithm for the SPO approach, depicted in figure 2, we can actually simplify the formula for computing the gradient. In fact, given that $v^*(y_i)$ is the target solution and, $v^*(2\hat{y}_i - y_i)$ is the solution contained in the cache (the one of only zeros). The sub-gradient can be rewritten as:

$$\nabla \mathcal{L} = -v^*(y_i) \tag{27}$$

A minus is added, since we are dealing with a maximization problem. As you can see, the gradient will not depend on our prediction, but only on the target. This is why, the model with INIT_2 is not able to deal with the distribution between the 1^{st} and the 14^{th} element.

If we suppose, that the features from the same slots, are similar, we have the key to understand this behaviour. In fact, since the sub-gradient will be the real solution, it means that, the network will change the weights in order to increase the values, associated to the one in the real solution. The problem here, is that, since we do not have an homogeneous distribution, the times that the values between the 1^{st} and the 14^{th} slot are increased, are less than those from the 15^{th} onwards. As a consequence, during the evaluation, the predicted values, for the first interval, will be incomparable, with respect to the second one.

Things change, if we consider a model with $p_{solve} = 100$, in fact, given the formula for the computation of the sub-gradient for SPO.

$$\nabla \mathcal{L} = -v^*(y_i) + v^*(2\hat{y}_i - y_i) \tag{28}$$

Here, the updates of the weights are dependent on the predicted solution obtained from $v^*(2\hat{y}_i - y_i)$. As a consequence, if we consider the slot i from the solutions, defined as $v^*(y_i)_i$ and $v^*(2\hat{y}_i - y_i)_i$, if both of them, are one, then $\nabla\mathcal{L}_i$ will turn out to be zero, which will not change the weights for optimizing the prediction for that particular slot. Basically, the model will stop increasing the value of that slot, when the prediction gives the same solution of the target. Leading to comparable values, even if the distribution of the solutions is not homogeneous.

By understanding the differences between the two initialization, we understood that, in these settings, it is necessary to put inside the cache, solutions, that can negate the update of the weights. In the next sub-section, we are going to use this knowledge, in order to develop a cache initialization, which can rely on few solutions, without even sampling.

6.3 Filter initialization

6.3.1 Introduction

In the previous sub-section we were able to discover why the model initialized with INIT$_2$ was not able to reach the same results of INIT$_1$. The main problem is that, since in the cache, it is present only a solution which is composed only of zeros, the network will always increase the predicted values, even if the associated object would still be picked. In order to deal with this problem, it is necessary to put inside the cache, solutions that will act as filters. Basically they will select the predicted values that need to be updated, in order to obtain the optimal solution. In this section we are going to develop an initialization that considers this fact.

6.3.2 Experimental setup

The experimental setup is the same explained in the section 4.3.1.

6.3.3 Experiment

For this experiment, we are going to define an additional initialization called, INIT$_{filter}$. Given the distribution of the target solutions, depicted in figure 33, 34 and 35; since we know that there is a disparity between the first fourteen elements and the rest, it is necessary to add solutions that prevent the unbalance in the predictions. In order to deal with it, the cache, will consist of two solutions: the first one, will consist of only zeros, while the second one, will have the first fourteen elements set as zero, while the other set as one.

We trained the model with this initialization and with $p_{solve} = 0$. The final performances are obtained by training it five times, by computing the average and the standard error of the test regret, for each epoch. The results are compared with those obtained from a model with INIT$_1$ and $p_{solve} = 0$.

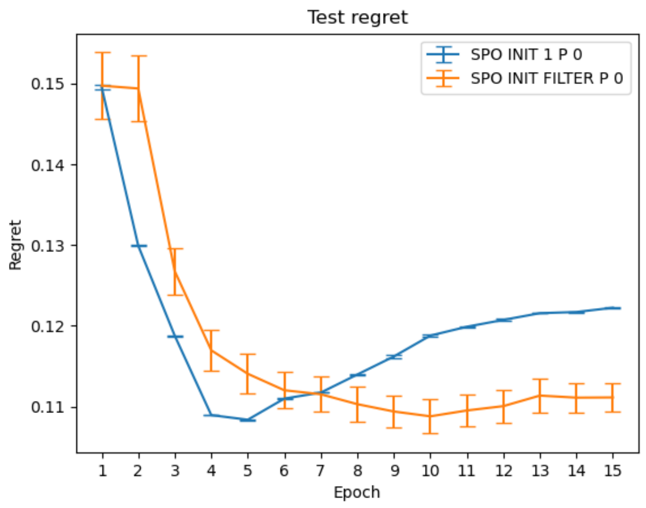


Figure 36: Comparison for SPO with INIT$_1$ and INIT$_{filter}$, capacity = 60

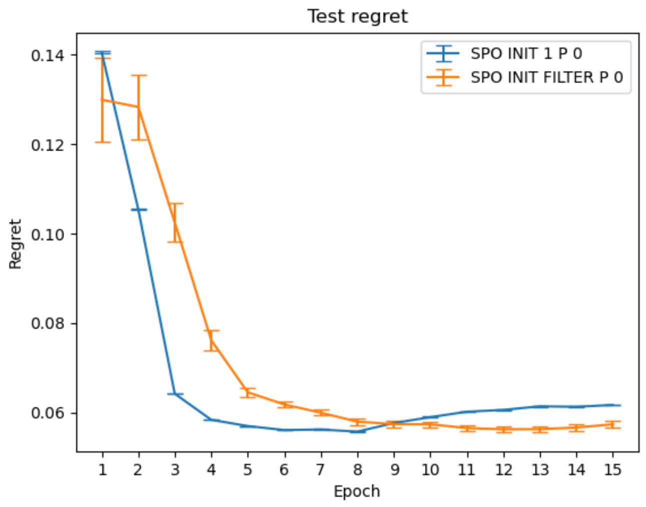


Figure 37: Comparison for SPO with INIT$_1$ and INIT$_{filter}$, capacity = 120

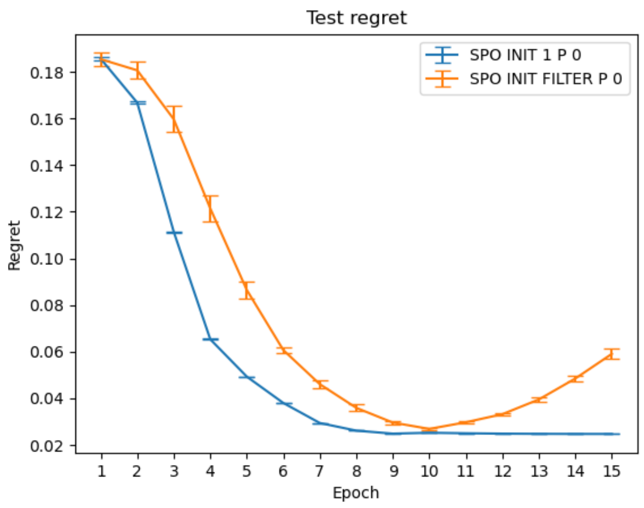
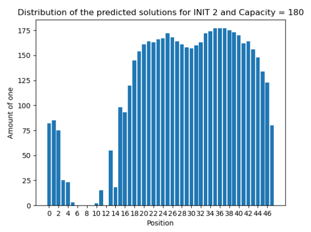


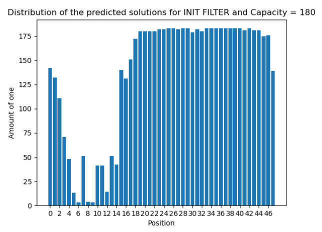
Figure 38: Comparison for SPO with INIT_1 and INIT_{filter}, capacity = 180

6.3.4 Discussion

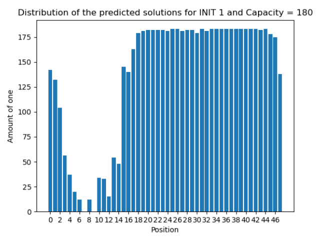
As you can see from figure 36, 37 and 38, the results are really promising. In fact with only two solutions inside the cache, we are able to reach the regret of the INIT_1 model. Also by taking a look at the distribution of the predicted solutions (fig. 39), we can confirm that the lower performances of the INIT_2 model were caused by the fact that, it was not able to mimic correctly the distribution between the 1^{st} and the the 14^{th}. While with INIT_{filter} it is not the case, leading to more optimal solutions.



(a) INIT_2



(b) INIT_{filter}



(c) INIT_1

Figure 39: Distribution of solutions for INIT_2, INIT_{filter} and INIT_1, capacity = 180

7 Cosine similarity for finding relations between predicted and real values

7.1 Introduction

In section 4, we were not able to find a relation between $\hat{s}$ and s_λ. As a consequence a smart sampling approach is needed in order to understand if it is better calling the solver or picking the solution from the cache, given a knapsack problem.

In order to do that, we need to compare the predicted values, with those associated to the solution stored in the cache. If they are very similar, we can infer that is not worth calling the solver.

In our case, we decided to measure the similarity between values with the cosine similarity. Given two vectors A and B $\in$ R^n, it is possible to compute it as:

$$sim(A, B) = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{29}$$

In this section, we are going to uncover if it is a suitable similarity metric or not.

7.2 Experimental setup

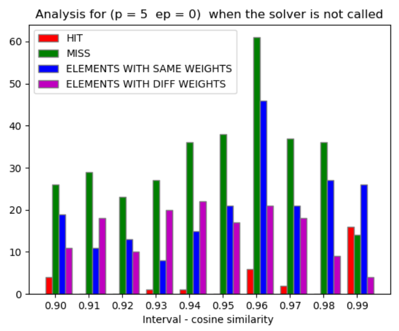
The experimental setup is the same explained in section 4.3.1, the only considered capacity is 60, since we expect to obtain similar results for all them.

7.3 Experiment

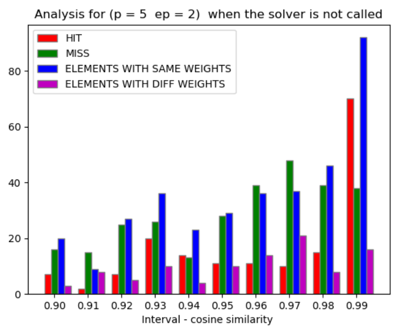
A Blackbox regret model with $p_{solve} = 5$, was trained for 25 epochs. The only difference between this one and the models, from the previous section, is that each solution in the cache, is linked to the predicted values returning that particular solution.

When the solver is not called, the cosine similarity between the predicted values and those stored in the cache is calculated and only the maximum value being considered. Then the real solution is computed, in order to check if it would be the same to sample from the cache (HIT) or not (MISS). It is also checked if the two solutions share the elements with the same weights.

In case the solver is called, the computed solution is stored in the cache as usual, without computing the cosine similarity.



(a) Analysis for the first epoch



(b) Analysis for the third epoch

Figure 40: Analysis of the similarity for the first and third epoch

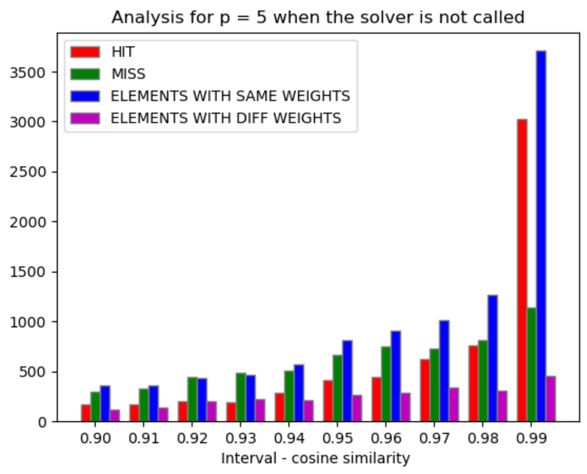


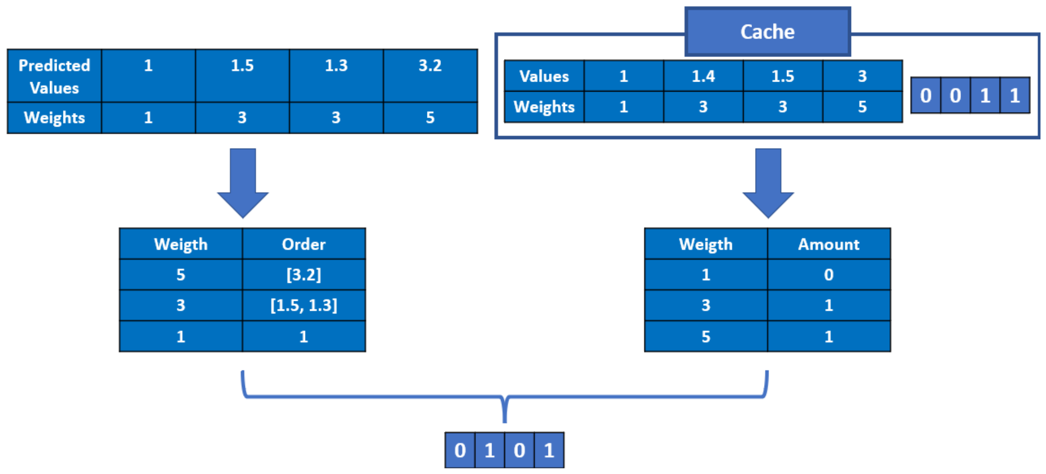
Figure 41: Analysis of the similarity, for all the epochs. HIT indicates the case where the solution from the cache is the same that the solver would return, MISS indicates the opposite case. ELEMENTS WITH SAME WEIGHTS indicates the situation where the elements of the solution picked from the cache shares the weights from the one that the solver would return

7.4 Discussion

As you can already tell from the figure 41, the cosine similarity can be a good indicator of when sampling new solutions. In fact with a value higher than 0.99 there is a great difference between HIT = TRUE and HIT = FALSE, which means that it is less likely to take from the cache a different solution from the real one. Even if it can seem a high number to reach, the graphs speak for themselves, in fact as shown in the figure 40, around 70 predicted values lay in that interval, already from the third epoch.

We can also notice from the figure 40a, that during the first epoch, this measure is less reliable, since, for a cosine similarity between 0.99 and 1, the difference between HIT and MISS is smaller. This is probably due to the fact that the solutions that would be returned by the solver would be very different from those stored during initialization. This is also confirmed by the figure 25 and 26, since with $p_{solve} = 100$ most of the time, during the first epochs, new solutions are being used. And it would also explain why, by considering later epochs, this measure, becomes more reliable, since new solutions are stored and it is more likely to have already the solution in the cache.

Last but least, it is important to notice that most of the time, with a high cosine similarity, the two solutions (the one from the cache and the one that the solver would return) share the weights of the elements. By knowing that instead of picking directly the solution, we can infer it, as depicted in the figure 42. By doing this, it is more likely to pick the correct solution, even if using a lower threshold. This is due to the fact, that the cosine similarity, does not take into consideration that these values will be used for solving an optimization problem, as a consequence, even small changes in the prediction, will change drastically the computed solution.



Predicted Values	1	1.5	1.3	3.2
Weights	1	3	3	5

Values	1	1.4	1.5	3
Weights	1	3	3	5

Weigth	Order
5	[3.2]
3	[1.5, 1.3]
1	1

Weigth	Amount
1	0
3	1
5	1

Figure 42: Creation of the solution: first of all the most similar values are found by using the cosine similarity. Then it is counted the amount of picked elements with a given weight for the solution in the cache. Based on that, we are going to consider the elements with the same weight but with the highest value in the predicted vector. In this example the maximum capacity is 8 and the cosine similarity between the two vector is 0.99

8 Smart sampling approach based on cosine similarity and inferred solutions

8.1 Introduction

In section 4, we were not able to find a relation between $\hat{s}$ and s_λ for the Black-box method. As a consequence a smart sampling approach is needed in order to understand if it is worth sampling a solution from the cache given the predicted values and those stored in the cache.

We even noticed that the solver, for some problem, will return a solution made by a small amount of one, due to the fact that, the predicted values are mostly negative for that particular problem. As a consequence, in these situations, we can immediately infer the solution from the predicted values, without calling the solver or sampling from the cache.

In section 7, we found out that we can actually rely on the cosine similarity, in order to understand if it is worth solving the knapsack problem or using the cache.

In this section we are going to develop a smart sampling approach based on cosine similarity and inferred solutions.

8.2 Experimental setup

The experimental setup is the same explained in section 4.3.1.

8.3 Experiment

First of all, it is necessary to define, the smart sampling approach, based on cosine similarity and inferred solutions. The first big difference, is in the structure of the cache, rather than having a simple list of solutions, it is necessary to store them, along with the values of the associated problem .

The first checks consist of summing the weights of the predicted positive values, if this sum does not surpass the total capacity, we can immediately infer the solution. In fact it will consists only of objects with a positive value. The solution is then stored in the cache, linked to the associated values, ready for further uses.

In the opposite case, where the sum of the weights exceeds the total capacity, we need to rely on the cosine similarity. Basically it is computed between the predicted values and those linked to the solutions in the cache; only the maximum value is considered. If it does not reach a threshold thr, then the solver is called and the new solution is used and stored in the cache. If it does exceed thr, we are going to follow the process depicted in figure 42, in order to obtain a proper solution.

For this experiment, we applied this smart sampling approach. In order to understand how the threshold thr influences the training process, a set of values was considered for it, in particular $thr \in [0, 0.3, 0.7, 0.9, 1]$.

For each of these values, we trained a model 5 times and we computed the average and standard deviation of the regret for each epoch. The cache was initialized with the target solutions, each of them linked to their values.

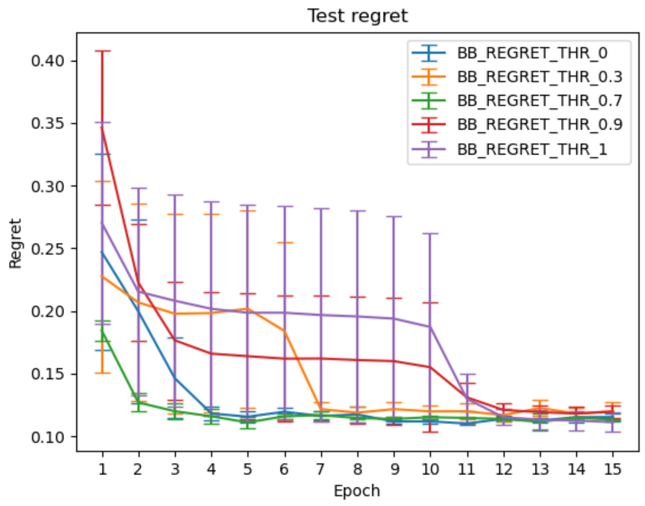


Figure 43: Comparison of Blackbox regret with different thr, capacity = 60

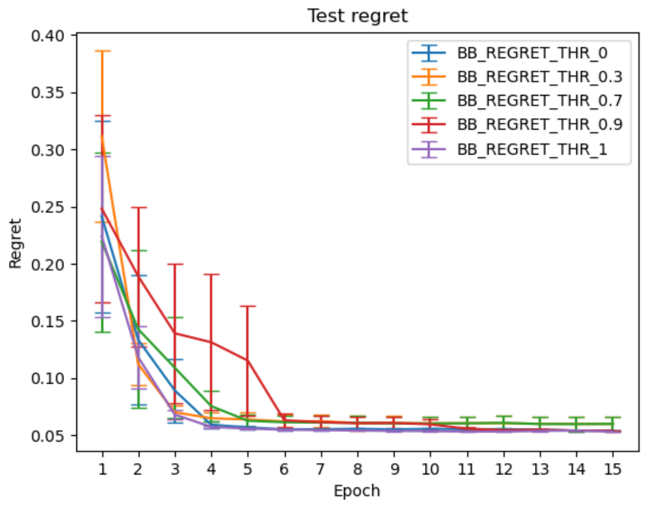


Figure 44: Comparison of Blackbox regret with different thr, capacity = 120

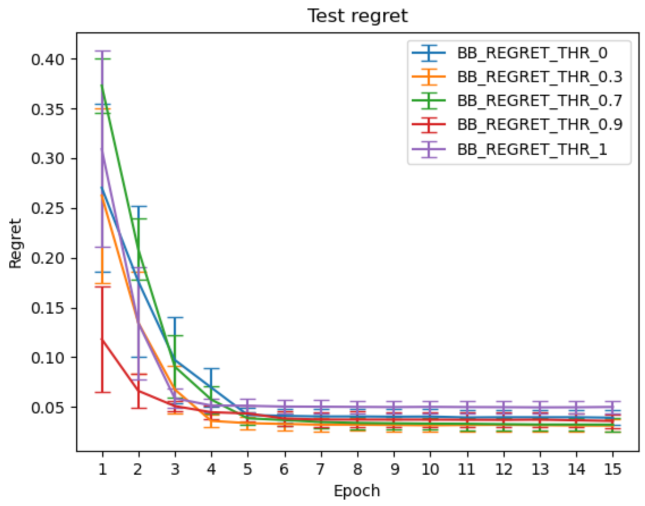


Figure 45: Comparison of Blackbox regret with different thr, capacity = 180

8.4 Discussion

As you can see from figure 43,44 and 45, the results, in terms of regret are almost similar for all the models.

This is good news, because as discovered for the SPO method in section 5 and 6, we do not need to solve additional problems, we just need the initialized cache. In fact, if we consider the model with $thr = 0$ the solver will never be called, the only added solutions are those obtained from the inference pass. This has to do with the fact that, in these settings, an optimization problem is solved if the threshold is not exceed by the cosine similarity, however, since this metric belongs to the interval [0,1], due to the fact that the negative values are clipped to 0, it will always reach the threshold thr, which means that we are going to rely only on the cache.

In the end, we will not need to compute the cosine similarity, we will just need the inference pass. With high probability, we are in a situation very similar, to the one encountered in section 5 with the SPO model. The main problem of the initialized cache, is that it contains very different solutions from those that the solver would return, as a consequence, we just need few solutions similar to those that solve the optimization problem (in SPO we just need the solution full of zeros), in order to reach performances comparable to those of a model that always calls the solver.

9 Generalization of the discovered approaches

9.1 Introduction

In this chapter, we want to understand if it possible to apply the discovered approaches designed for a specific method, to another. In particular, in section 9.2 we are going to apply the smart sampling approach designed for Blackbox on SPO, while in section 9.3, we are going to initialize the cache as we did for SPO, but by using a Blackbox model.

9.2 Blackbox smart sampling approach on SPO

9.2.1 Experimental setup

The experimental setup is the same explained in section 4.3.1.

9.2.2 Experiment

A SPO model with $p_{solve} = 0$ and initialized with INIT_0 is created. In addition to that, in order to understand if it is possible to apply the approach described in section 8, it will be necessary, if possible, to infer the solutions from the predicted costs. The model is trained five times and the average and the standard error of the test regret are computed for each epoch.

The final results are compared with those obtained from a SPO model with $p_{solve} = 100$.

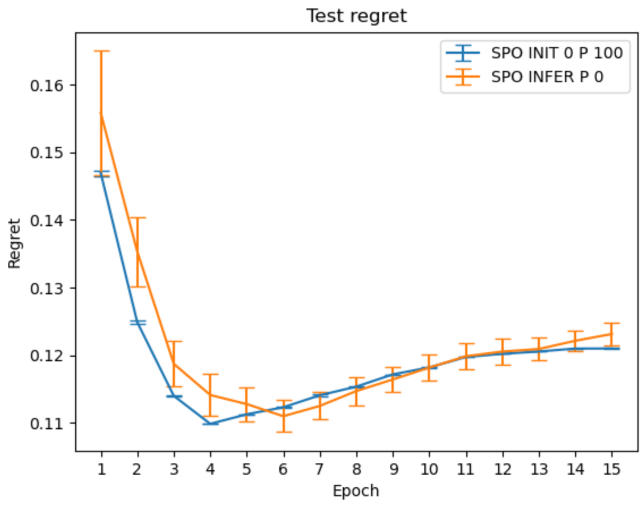


Figure 46: Comparison of SPO with inferred strategy and SPO with $p_{solve} = 100$, capacity = 60

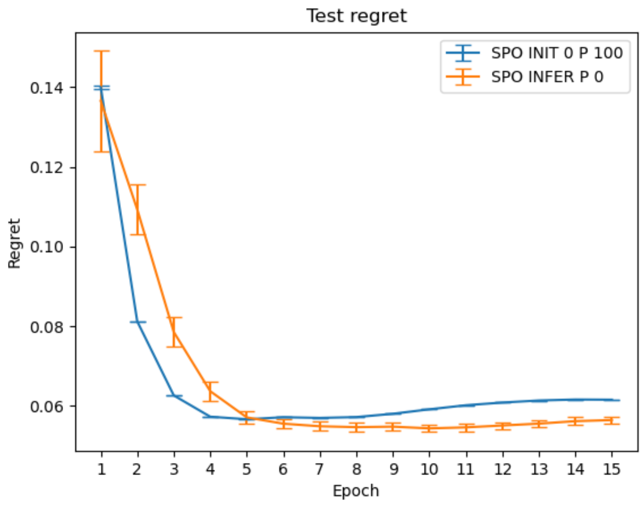


Figure 47: Comparison of SPO with inferred strategy and SPO with $p_{solve} = 100$, capacity = 120

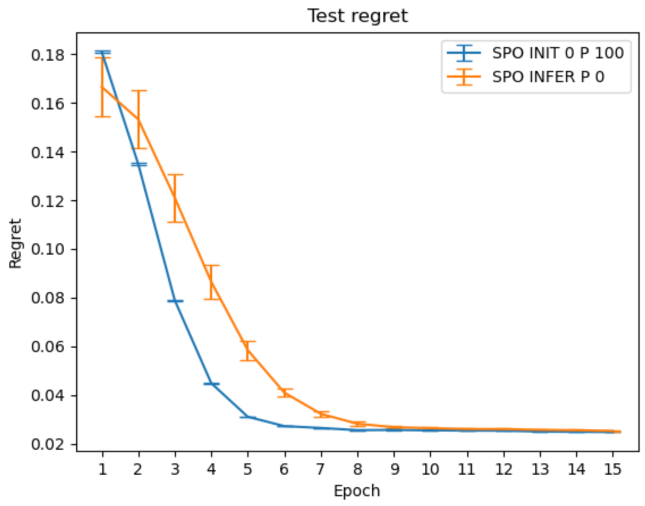


Figure 48: Comparison of SPO with inferred strategy and SPO with $p_{solve} = 100$, capacity = 180

9.2.3 Discussion

It is obvious that applying the smart sampling approach designed for Blackbox, on SPO, will give results similar to the optimal ones, as demonstrated by the figures 46, 47 and 48. In fact, as discovered in section 5, it is only necessary to have a cache filled with target solutions and one made only of zeros, to reach performances comparable to a model with p_{solve} = 100. Since we are going to initialize the cache with the target solutions and, one of the firsts inferred solutions will be only of zeros, we are going to obtain a model similar to those initialized with INIT$_1$.

9.3 Initialization of the cache on Blackbox

9.3.1 Experimental setup

The experimental setup is the same explained in section 4.3.1

9.3.2 Experiment

A Blackbox model with regret as loss, is created. During the initialization its cache, will contain the target solutions, plus one made only of zeros, as we did for

SPO models with INIT_1. The model is trained five times and the average and the standard error of the test regret are computed for each epoch.

The final results are compared with those obtained from a Blackbox model with $\text{p}_{solve} = 100$ and initialized with the target solutions (INIT_0).

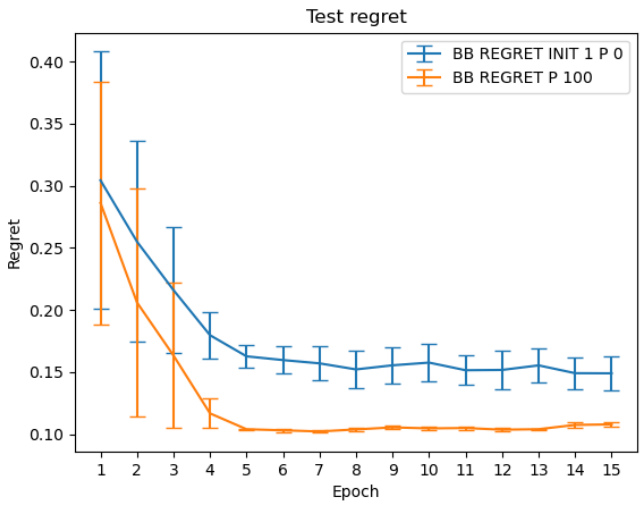


Figure 49: Comparison of Blackbox models with INIT_0 and INIT_1, capacity = 60

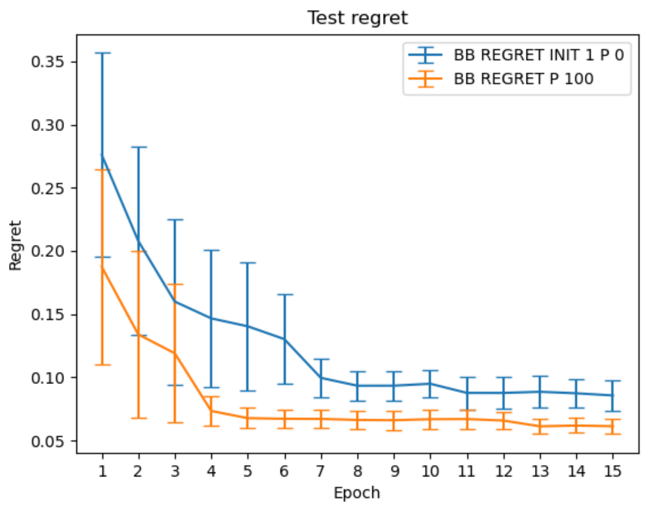


Figure 50: Comparison of Blackbox models with INIT_0 and INIT_1, capacity = 120

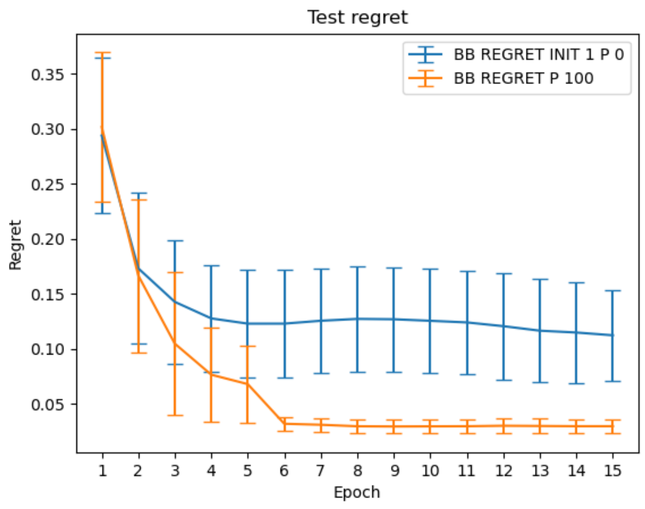


Figure 51: Comparison of Blackbox models with INIT_0 and INIT_1, capacity = 180

9.3.3 Discussion

As expected, the behaviour of a Blackbox model initialized with INIT_1, will not reach the results of a model with $p_{solve} = 100$. The main problem here, as noticed also in section 4.3.3.2, is that the solution with only zeros does not have that much importance for the learning process, while it is the opposite case for SPO. As a consequence we can also infer that adding only one new solution is not enough. However, in section 8, we noticed that by inferring them, we were able to obtain a better regret. As we know these solutions will consists of few picked object, as a consequence they must be one of the key elements for reaching optimal solution. At the same time, they are probably not enough, it is also necessary to use the target solutions. This is due to the fact, that for these models, it is necessary to check the cache or use the solver twice. For the first call, involving $\hat{s} = v^*(\hat{y})$, most of the time the inferred solutions will be used; while in the second case, for $s_\lambda = v^*(\hat{y} + \lambda y)$, the target solutions are those most likely to be picked.

10 Conclusions

The main objective of this thesis was to develop a smart sampling approach, designed for reducing the computational time due to the solver calls. We considered two problems: the shortest path problem and the knapsack problem. What we noticed is that in all the considered tasks, most of the time, the predicted values will not be comparable, in terms of magnitude, to the real ones. This will lead to particular behaviour during the learning process, in which the prediction does not have influence.

Considering SPO, for the shortest path problem, the optimization problem will involve basically $-y$, and the solution, most of the time, will be one from the target, as a consequence we do not actually need to sample new solutions; it is also important to point out that the solution space is not quite big. At the same time, for the knapsack problem, the returned solution from $-y$, will be the one made only of zeros. What we can do, is adding this solution, during initialization in order to obtain the same results of a model which would always call the solver. We can actually exploit this knowledge about the knapsack problem, to initialize the cache with only several solutions, in order to filter the elements which the model still needs in order to learn their values.

For the Blackbox model, if we consider the shortest path problem, we noticed that s_λ, will be the same for both the losses. In fact it will be $v^*(y)$, which will return one target solution. We also noticed that $\hat{s}$, is likely to be one of those. As a consequence we do not need to sample new solutions. For the knapsack problem, things are quite different, first of all, since the optimal value for λ is really low s_λ will not be the real solution, since the predicted value, influences the obtained result. As a consequence, here it is necessary, to use a smart sampling approach, based on inferred solutions. As we noticed by adding these solutions, made of few one. We will obtain results similar to a model which would always sample.

From this research it is possible to understand that sampling, is not needed and we can actually exploit some behaviour of our model, to create new solutions, that will help the learning process. The next research question would be to understand if we would find the same behaviour in other problems and if we can actually apply the designed smart sampling approaches to them.

References

Emir Demirović, Peter J. Stuckey, James Bailey, Jeffrey Chan, Chris Leckie, Kotagiri Ramamohanarao, and Tias Guns. An investigation into prediction + optimisation for the knapsack problem. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 241–257, Cham, 2019. Springer International Publishing. ISBN 978-3-030-19212-9.

Adam N. Elmachtoub and Paul Grigas. Smart "predict, then optimize", 2017. URL `https://arxiv.org/abs/1710.08005`.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL `https://www.gurobi.com`.

Georgiana Ifrim, Barry O'Sullivan, and Helmut Simonis. Properties of energy-price forecasts for scheduling. volume 7514, pages 957–972, 10 2012. ISBN 978-3-642-33557-0. doi: 10.1007/978-3-642-33558-7_68.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL `https://arxiv.org/abs/1412.6980`.

Jaynta Mandi, Emir Demirović, Peter. J Stuckey, and Tias Guns. Smart predict-and-optimize for hard combinatorial optimization problems, 2019. URL `https://arxiv.org/abs/1911.10092`.

Maxime Mulamba, Jayanta Mandi, Michelangelo Diligenti, Michele Lombardi, Victor Bucarey, and Tias Guns. Contrastive losses and solution caching for predict-and-optimize, 2020. URL `https://arxiv.org/abs/2011.05354`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL `https://arxiv.org/abs/1912.01703`.

David Pisinger. Where are the hard knapsack problems? *Computers Operations Research*, 32(9):2271–2284, 2005. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2004.03.002. URL `https://www.sciencedirect.com/science/article/pii/S030505480400036X`.

Marin Vlastelica, Anselm Paulus, Vít Musil, Georg Martius, and Michal Rolínek. Differentiation of blackbox combinatorial solvers, 2019. URL `https://arxiv.org/abs/1912.02175`.

Acknowledgements

First of all I would like to thank professor Lombardi and Professor Guns, for allowing me to do the thesis in Belgium, supporting me in this experience. Also the assistance provided by Mandi Jayanta, Mulamba Maxime and Berden Senne, for preparing the final project was essential. They too deserve to be thanked, especially Senne, for allowing me to discover the taste of the Gouden Carolus Tripel.

In Belgium I have met a lot of friends, which I hope I will meet soon, I would like to thank particularly Zancanaro Paola, Bravin Alice, Barone Paola, Di Francia Angela, Radomiak Hannah, Vermierdt Ruben and Zucchini Alessandro, for having improved this experience; without them it would have been totally different.

Also I would like to mention Biancacci Veronica, Aiezzo Agostino, Cena Carlo, Damiano Antonio and Spurio Federico for being excellent companions for all the exam projects. Of course I have also to say thank you to all the friends that I have met during these two years of master’s. I would also like to extend my thanks to Montebugnoli Sofia, Foschi Daniele, Crivellari Ilaria, Righini Elena and Varotto Riccardo, for being my friends since the bachelor’s degree.

Furthermore it is fundamental to express my deep gratitude to: Lobietti Enrico, Contarini Nicola, Ancarani Alessandro, Ravaglia Davide, Calderoni Filippo, Costa Francesca, Monti Giada, Capirossi Giorgia, Benini Ilaria, Loreti Alessandro and Mazzotti Lorenzo, for accompanying me on this adventure lasted ten years, from high school to this master’s.

Last but not least, I would like to thank one of the most important part of my life: my family. Thank you for supporting me thought all these years.