Implementing policy gradients to build a hyper-heuristic selection model.

Leonardo Clemente, Jose Carlos Ortiz-Bayliss, and Hugo-Terashima Marín
Tecnológico de Monterrey, Departamento de sistemas inteligentes, Campus Monterrey

A hyper-heuristic selection model was developed using the policy gradients technique. Model was trained twice to select between 2 and 4 heuristics. Model was trained to perform on a sub-domain of the one-dimensional bin packing problem. Policy function is approximated using a fully connected neural network. Environment, score function and heuristics are hand coded and model is implemented using python and tensorflow. Results for the model with a fully connected layer show that the model does learn to take better decisions within time based on arbitrary metrics (average space used per bin), yet it is not capable to beat the individual use of heuristics.

I. INTRODUCTION

Hyper-heuristics are methodologies that search the space generated by a finite set of low level heuristics.[1] Hyper heuristics are higher level algorithms that provide an alternative to other approaches to solve problems that do not necessarily have an exact solution. The scope of hyper-heuristics is broad, and ranges from methods designed to choose the most 'appropriate' heuristic depending on the instance of the problem or even the generation of new heuristics by the method itself [2]. Hyper-heuristics formally became a research topic around 2000 based on works from Cowling and Soubeiga and a journal publication in 2003 [10]. They originate from the main heuristic research. Heuristics are algorithms that provide approximated solution in a short time to problems that do not have an exact solution or that would take a huge amount of time using exact mathematical methods. There is a large quantity of real-life problems that do not have an exact solution and at the same time are relevant for day-to-day practical applications, a few examples are mentioned below:

- Knapsack problem given objects of various sizes and values and a knapsack with a fixed integer size, choose the objects that can fit inside with the biggest value.
- Traveling salesman problem : Finding the shortest route that allows you to visit every city exactly once (variations exist).
- Bin packing problem : Given a set of n pieces with different sizes, find a way to fit them in the least number of bins of fixed size.
- Minimal Vertex cover: In a computational graph, the challenge is to find the smallest set of vertices such that each edge contains at least one vertex.
- Clique : In a computational graph, finding subgraphs such that each of the vertices are completely connected.

All of these problems are considered NP-hard. This means that it would take a big amount of time for a computer to find an optimal solution. This is why investing computational resources looking for an optimal solution might not be the best approach to a practical solution and, instead, heuristics are used to find a feasible solution [15]. Hyper-heuristics are sometimes confused with other high-level heuristic algorithms called meta-heuristics. Meta-heuristics are considered methodologies that search a large solution-space with the similar objective of finding a good solution. TheDifference between meta-heuristics and hyper-heuristics the space where each of them looks for the solution. Hyper-heuristics ultimate goal is to solve a problem that does not have an exact solution. The difference between hyper-heuristics and other problem-solving techniques is that it does so by searching not through the solution space of the specific problem but a different space generated by heuristics. The idea behind is that the algorithm used to solve the problems is able to interact with, ideally, all the possible instances of a problem. A brief (non-canonical) explanation on how they work is described in Figure 1. A hyper-heuristic model is designed and it interacts with a portion of the problem domain to find, for example, the best combination of heuristics to generally solve the problem.

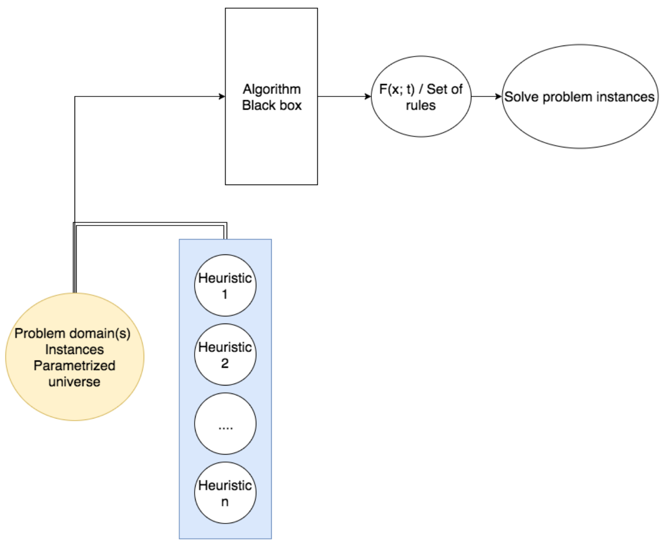

FIG. 1: A hyper-heuristic model is optimized based on instances of a problem domain and a set of heuristics.

A compact description of documented algorithms used in hyper-heuristics is described by Ender et al. [1]. In that work the author's mentioned algorithms include randomization schemes such as simple random, random descent (pick a heuristic randomly and keep using it until the solution worsens then pick another one), computationally intensive such as greedy (given a problem, evaluate the whole sets of heuristics then choose the one that gives the best solution) and peckish (analyze a subset of heuristics and choose the one that gives the best solution) and reinforcement learning algorithms, which involve the use of a learning agent which learns based on positive and negative feedback of the heuristic it choses everytime.

More recently, Documented work on reinforcement learning (RL) and Neural nets (NN) demonstrates the potential and capacity to tackle large-domain problems (see [1],[2] and [3]).

Reinforcement learning models have recently shown big success in areas where it is neccessary to tackle problems with big search spaces and large-scale tasks. Deepmind's team was able to demonstrate how a computer could learn to beat ATARI games and more recently developed a Go playing model. (Go is a classical strategy game considered the most complex. It is stated that the number of plays that exist in Go is bigger than the number of atoms in the 'observable universe') [13]. Success is attributed to new techniques for function approximation (Deep learning, specifically) and also being able to process large amounts of data in shorter amounts of times due to GPU or more efficient cloud-computing implementations. Based on this, the idea of developing a hyper-heuristic model arises. It is possible to see an NP-hard problem as a game, where the the RL agent has access to information via hand-coded observation vectors and it plays using the heuristics as the possible actions within the game.

For this project we explore the implementation of policy gradients, a technique that combines reinforcement learning and neural networks, on heuristic selection models. The main scope of the project is to explore the possibility of implementing neural networks and reinforcement learning techniques to hyper-heuristics. Policy gradients technique to solve NP hard problems using an hyper-heuristic approach. The initial test problem used for these models is the bin packing problem in one dimension.

II. JUSTIFICATION

Reinforcement learning is an AI technique used to train agents for tasks where you have completed or even limited access to information. It is used for sequential data where a notion of "time" is necessary and that actions taken at the current time affect the future [14]. RL is classified as an unsupervised learning algorithm. Basic unsupervised learning involves observing several examples of a random vector x, and attempting to implicitly or explicitly learn the probability distribution $p(x)$. [6] Heuristic selection models create a sequence of decisions while solving a problem's instance. The model's learning is based on these sequences. In RL you do not have access to instant and direct feedback, only to a reward signal that gives a (usually scalar) number. A RL agent is not particularly interested in the immediate reward of a state but the reward from a sequence. When possible, reward functions are arbitrarily designed to meet long term or short term sequences.

Reinforcement learning systems interact with the environment and change its state via a selected action in such a way to increase some notion of long term reward [14]. The main question in this algorithm is What's the best sequence of steps (states) I should take to maximize my reward? A basic diagram of RL is shown in Fig. 2.

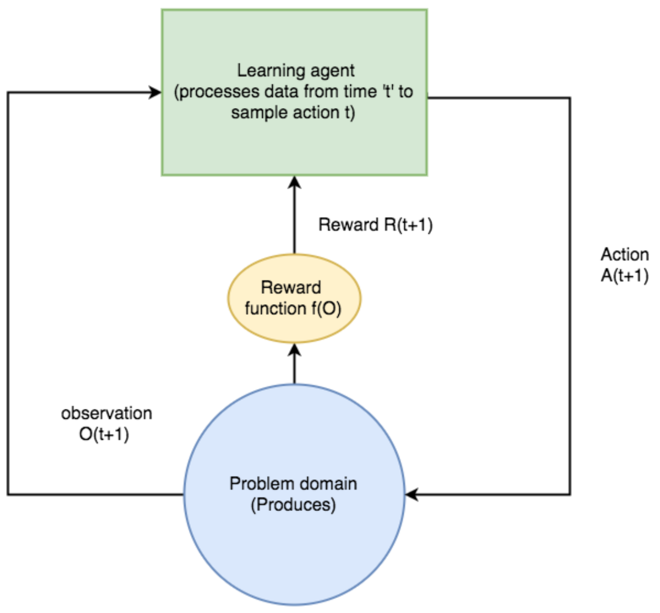
Learning agent (processes data from time 't' to sample action t)
Reward R(t+1)
Reward function f(O)
Action A(t+1)
observation O(t+1)
Problem domain (Produces)

FIG. 2

A RL model is graphically described by an environment, a learning agent, and all the intermediate information transactions. The learning agent interacts with the environment via actions A and receives a reward R and an observation O (a vector with information about the new state of the environment). Mathematically, a RL process is modeled as a markov decision process with the following tuple. $\langle S, A, P, R, \gamma \rangle$ where :

- S if a finite set of states
- A is a finite set of actions
- P is the transition probability (usually a matrix)
- R is a reward function
- γ discount factor (between 0 and 1)

And a series of functions :

- The value function V, which describes in average the value of each possible state in the environment to analyze.
- The state-value function Q, which gives in average the value of each possible action given the fact that the model is in a specific state.
- The reward function that gives an score based on the action(s) taken by the model or the state of the environment.
- The Transfer probability function, which gives the probability of going to a state s_f from a state s_o

The RL agent is able to complete the desired task when we find optimal functions for V and Q. In practice this functions are estimated or approximated and the point is to numerically find such optimal functions [16].

$$V_*(s) = \underset{\pi}{max}\ V_\pi(s)$$

$$q_*(s,a) = \underset{\pi}{max}\ q_\pi(s,a)$$

III. POLICY GRADIENTS

Policy gradients is a practical approach to RL. In this case, the approximated function is not the value of state-value function but the policy function. The policy function fully describes how the model should behave depending on the state of the environment. Policy gradients mathematical justification is based on the 'Score function gradients estimator theory'. This theory tells us that the gradient of the expected value of an score function $f(x)$ under some probability distribution $p(x;\theta)$ is equal to the expected value of our score function times the gradient of the logarithm of the probability function $p(x)$ [8].

$$\nabla_\theta E_x[f(x)] = E_x[f(x)\nabla_\theta log(p(x))]$$

The actions order for the policy gradients technique is as follows:

1. Define a stochastic policy function (in this case a neural network) π that implements agent's behavior with actions.
2. Gather up samples (sequences of actions) by making the policy and the environment interact.
3. Calculate the respective reward for each of the actions take within each sample.
4. Define the discounted reward.
5. Find the gradient of the score function for each episode until the desired number of episodes is met.
6. Backpropagate (optimize) through policy.
7. Repeat steps 2 to 6 until the policy fits a correct behavior

Based on Karpathy's work and deepmind, policy gradients have been more successful than other RL approaches and less computationally intense [16][144]

IV. POLICY GRADIENTS AND HYPER-HEURISTICS: THE MODEL ARCHITECTURE

The policy gradients technique was chosen to generate a heuristic-selection model. In this case, the model interacts with a series of problem instances from an specific domain and has to decide the most appropriate heuristic for that particular state. The policy function is approximated using a neural network and it is used to sample actions which in this case are a set of hand-coded heuristics. The observation and reward are also hand-coded functions.

A. The environment: The bin packing problem

The domain selected to test the proposed technique was the bin packing problem. The bin packing problem is defined as: 'given a list of objects and their sizes, and a collection of bins of fixed capacity, find the smallest number of bins so that all the objects are assigned to a bin'. This problem is considered an NP-hard combinatorial optimization problem.

As a first approach, the sub-domain chosen is that of all pieces with integer size values from 1 to 9 and bins with fixed size capacity 10. The limit size of pieces is suspected to play an important role in the efficiency of the heuristics. Different piece size limits for a fixed bin capacity need to be explored.

For each instance, pieces are generated based on a uniform probability distribution [1,9] in python. A python seed of 2 is used for repeatability.

The bin packing problem is simple and can be extended to 2D and 3D, providing even bigger challenges for the model.

B. The policy

The policy function is approximated using a fully connected neural network. The neural network architecture is composed of the input vector x (a vector of 10 features) a hidden layer of 100 neurons and an output layer of either 1 neuron (binary classification) or 4 neurons (4 class classification).

The input vector features are related to either information about the pieces left to fit and information about the space within the current opened bins:

1. The size of the current piece to fit in a bin.
2. Number of pieces left.
3. Total space from pieces left.
4. Average size from pieces left.
5. Minimum space left from current bin list.
6. Maximum space left from current bin list.
7. The percentage of space used from all the space in the current bin list.
8. Average space left from a bin
9. Number of opened bins

Observation was designed to be compact and contain statistical information about the problem instance to approximate an exact information state to prevent overfitting of the neural since no regularization or dropout is being implemented [5]. The policy's parameters are initialized so the probability sampled at the output layers are relatively equal at the start of the training. This, along a stochastic selection during training encourages exploration of different heuristic combinations.The model is trained using a gradient descent technique with an RMSprop normalization. RMSprop is used based on information available on [8] and [11]. Other optimization algorithms are yet to be explored.

C. The actions

The policy function samples a set of probabilities for each of the possible heuristics to select. . During the experiments, the model is optimized to choose between either 2 or 4 heuristics. For the one-dimensional bin packing problem, the selected heuristics were:

- Next fit : Directly places the piece on the last open bin.
- First fit: Fits the piece on the first bin that has enough space.
- Best fit: Places the piece inside the bin that will leave the least space left.
- Worst fit: Places the piece inside the bin that will leave the most space left.

Heuristics are hand-coded and implemented as functions in python. If any of them is not able to fit a piece inside the current bins, a new bin is opened.

D. The reward

The reward function is hand-coded to give the algorithm a scalar score based on its decisions. It must return a positive score when it fits pieces correctly, and a negative score when it performs poorly (below an arbitrary used space threshold). The importance of correctly assigning rewards is crucial because it affects the final direction of the gradient for each of the decisions taken.

Reward is generated the following way:

For a set of bins with pieces inside, the reward function gives 2 points for each full bin. For each bin that's on 70 percent of its capacity or above, the function gives the value of the current space used in percent (a value from .7 to .99) and for each value that's below the 70 percent threshold, the function assigns a value of $v = -(1 - c)$ where c is the current space used (a value of -1 to -.3).

Filling a bin completely has more than the double of reward than filling it between .7-.99. This encouragea priority of full bins over almost full bins. Values below .7 are penalized to encourage filling bins rather than opening up new bins. Another reason for these values is extending the reward function's domain to negative values. The idea behind is that samples that gave out too many opened and non-filled bins have a big negative score and, while optimizing, the gradient will move in the opposite direction to prevent doing those same decisions. On the other hand, Samples with the most full bins will have a big positive score and the gradient will be scaled to increase the probability of taking that set of actions. For this project the reward function is called only at the end of the decision taking process. Intermediate rewards are yet to be explored.

E. Model optimization

Data obtained from running various problem instances and generating the samples is organized in sets of samples called episodes. For each episode, the algorithm calculates the gradient of the parameters and sums them up with the gradients from other episodes until it reaches an arbitrary episode limit. The model is tuned using stochastic gradient descent along with RMSprop normalization. It is important to mention that the reward for each of the decisions take is also normalized prior to multiplication with the log probabilities of the policy function [10].

For this model, backpropagation occurs every 10 episodes. Each episode contains 10 samples from randomly generated instances. And each instance is composed of 50 different decisions of forward propagations through the policy function. This accounts to 100 sequences or 500 decisions taken in different problem instances of the same bin-packing sub-domain.

V. EXPERIMENT AND RESULTS

All models were trained using 300,000 randomly generated instances with the same seed (2).Number of instances used was determined experimentally based on the speed of convergence of the model. Using a seed allows reproducing results and testing other models based on the same values for the pieces used on every instance. A list of the pieces for each instance can be found on `https://github.com/LeonardoClemente/binPackingPolicyGradients`.

Instances were used to train the 4-heuristic model and six binary-classification models (one for each pair combination for the four chosen heuristics).

Algorithm's results were measured based on average bin space used (ASU). This is basically the sum of every space used within each bin divided by the total number of bins. Average spaced used is measured locally and globally. Local ASU is an average of all the instances in 1 set of 10 episodes prior to backpropagation. Global ASU accounts for all the 300, 000 instances during training. The individual ASU results for each heuristic (this means using the same heuristic over and over for all the instances) are shown in table 1.

TABLE I: My caption

Heuristic Name	**Average space used per bin.**
Next fit	.695
First fit	.816
Best fit	.466
Worst fit	.772

Next fit and worst fit have the two highest scores while the best fit heuristic performs poorly at only .772. It is relevant to know the performance of individual heuristics to have comparison values and see how well the algorithm performs in case it finds a good heuristic combination. It is also a threshold for which the model has to beat.

Both the 4-way classification and binary classification models (figures 4 and 6) showed learning but did not surpass the score set by the individual heuristics. In most cases the model converged to using only one heuristic (the heuristic with the highest score). Figure 4 Shows how the 4 class model starts at a value approximate to .74 bin capacity. This value corresponds to the average space occupied per bin if we were choosing heuristics using an uniform probability distribution. After some time has elapsed, the model asymptotically converges to the highest individual heuristic (next fit heuristic) value. figure 5 shows a bar chart corresponding to the use of each heuristic during the model training(in %). Figure 6 shows the evolution of the use of each heuristic during training for the 4-class model.

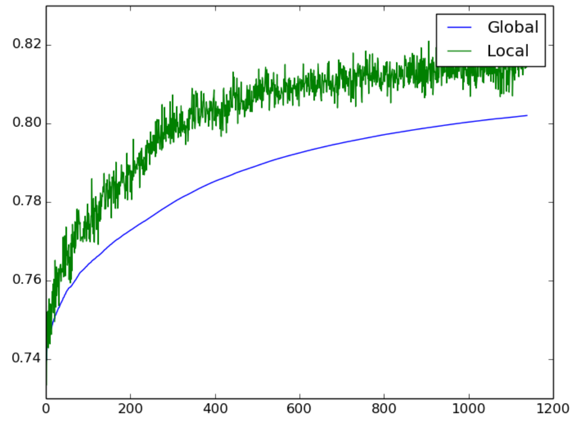

FIG. 3: Graphical evolution for the 4-way classification model. Y coordinates correspond to average bin space in percent whereas x coordinates correspond to the number of times backpropagation has been applied to the model.

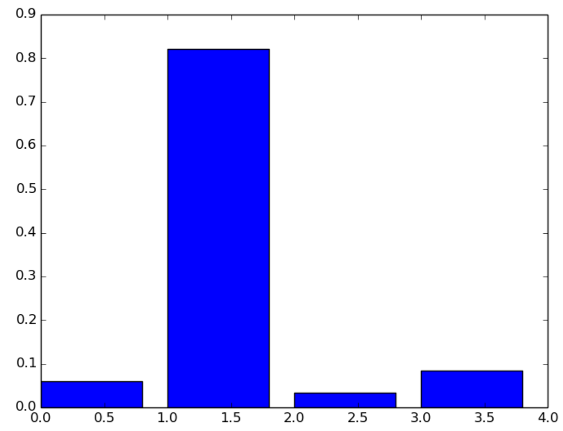

FIG. 4: Normalized proportion in use of all the decisions taken within the learning in the 4-class model.

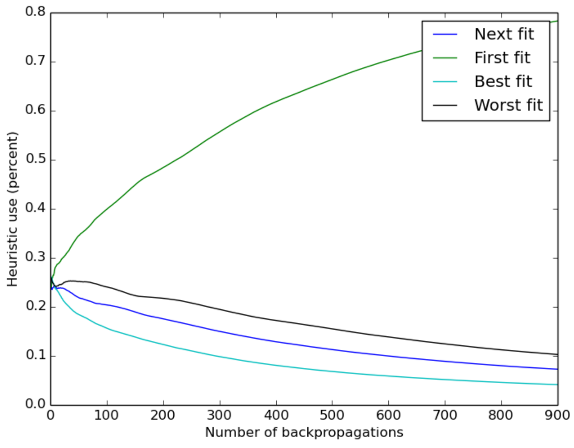

FIG. 5: Evolution of the use of each heuristic in the 4-class model.

Fig. 3 and 5 show the evolution of the 4-class model. Fig.3 shows how the ASU global average converges to a value of approximately 81.4l. This value is close to the ASU for the first fit heuristic but does not beat it. Fig. 5 shows how the first fit heuristic becomes the most used heuristic of the model while the other 3 become less used. There's a possibility that the function is quickly increasing the probability of choosing the first fit heuristic given it has an score that in average results in positive rewards, thus reducing the exploration capacity of the model in an early stage.Other reason could that the decision possibilities are limited for the sub-domain used, More exploring must be done before further conclusions. Fig. 4 is an histogram that shows, by the end of training the model, the use of each heuristic in percentage.

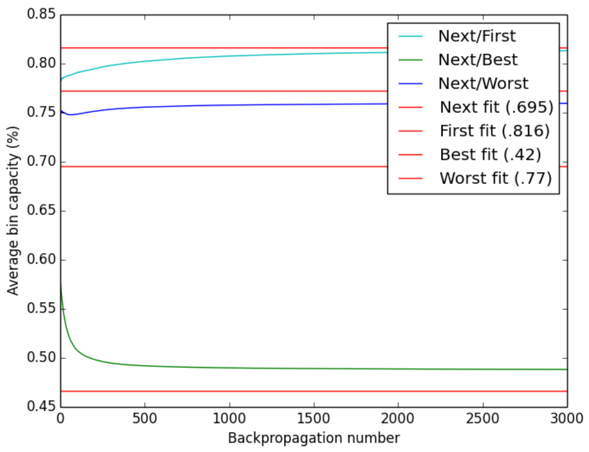

FIG. 6: Average bin space used evolution for model permutations containing the next fit heuristic

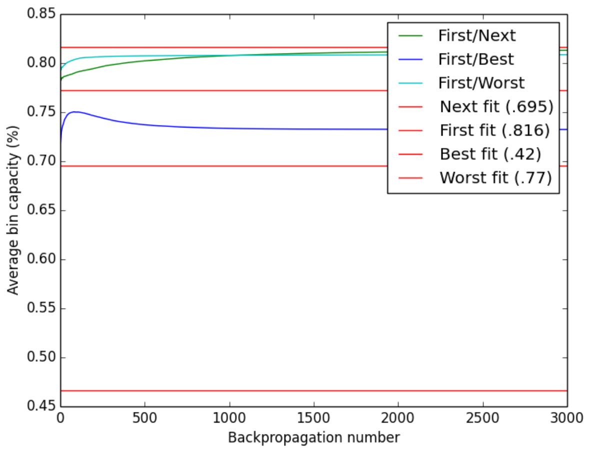

FIG. 7: Average bin space used evolution for each of the first fit heuristic

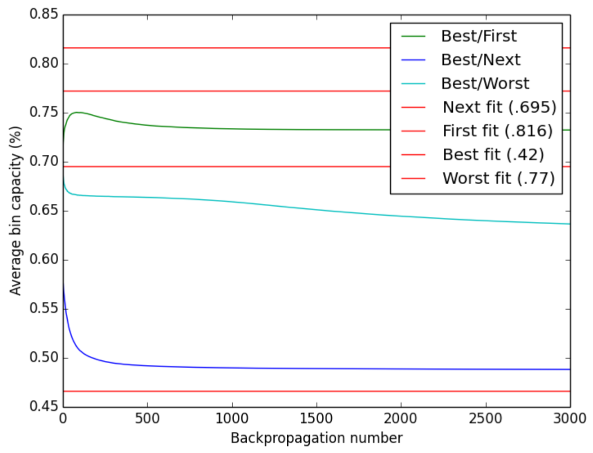

FIG. 8: Average bin space used evolution for each of the best fit heuristic

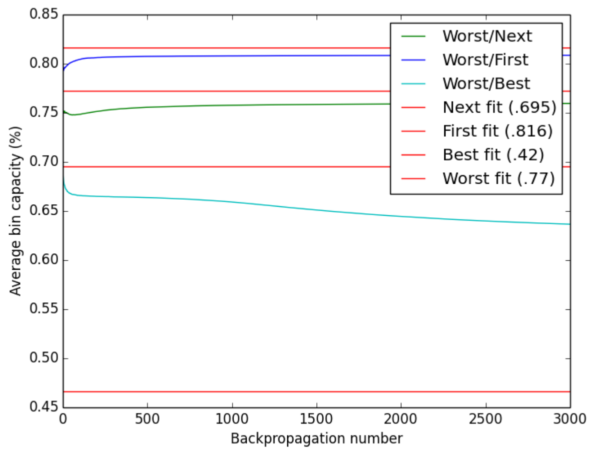

FIG. 9: Average bin space used evolution for each of the best fit heuristic

Binary models global ASU evolution is shown on Figs. 6 through 9 (local ASU is not included to avoid cluttering). There's one figure for heuristic and its possible combination with the rest. The red horizontal lines in all figures correspond to the ASU value of using heuristics individually. All model combinations which started with an ASU below .7 showed erroneous behavior. This is accounted to backpropagation only working with negative feedback from the reward function all the time and the model's gradient not being able to point in the right direction. Models that started with an ASU above .7 showed positive learning with exception to the first heuristic and best heuristic combination (Fig.7), which starts at an ASU value of .70 to reach a maximum value of .75 and then slowly plateaus to a value of .73. Reasons for this behavior could be the model is stuck in a situation where it can't discern the samples that improve the model from the one that don't because, on average, the best fit heuristic's poor performance is being covered by the first fit's best performance. Models that showed overall positive learning slowly converged to the value of the heuristic with the highest individual ASU. On these cases the reward function had only positive rewards, thus samples where the model chose the heuristic with the best individual score more frequently also had the bigger rewards (on average) and the gradient was scaled accordingly, encouraging the use of the individual heuristic more and more.

VI. CONCLUSION

Preliminary results in the use of policy gradients for the design of an hyper-heuristic model were shown. In most cases, the model showed improvement over its decision taking and converged to the individual heuristic with the highest score. The model might not be able to surpass the individual scores because it converges too fast to only one decision. Increasing the number of instances before each backpropagation might account for that. Other possible reason is that the selected sub-domain is not suitable. Having pieces that take the most space of a bin might reduce the combination possibilities. More sub domains must be explored before further conclusions. The model is yet to beat the highest score made by individual heuristics. A new exploration policy must be devised to encourage search for alternatives to only using the same heuristic in most cases. Regarding Model's current input, it does not give full information about the current state of the instance. The model only has access to statistical information about the pieces and the space left within the bins but not the actual size of every piece or bin. Implementing full access to pieces and space left might improve learning given the fact it approaches more to the idea of the instance states being fully Markov. There was not observable difference in the performance between binary and 4-class models with the exception that binary models that started with rewards below .70 showed negative results. Reward function variations to adress that problem must be designed.

VII. REFERENCES

[1] Ozcan et al.(2010) "A reinforcement Learning - Great - Deluge - Hyper - heuristic for examination timetabling". Retrieved from http://www.maths.stir.ac.uk/~goc/papers/GD-RLHH_JMHC.pdf

[2] P.H. corr. (2006). A New Neural Network Based Construction Heuristic for the Examination Timetabling Problem. Volume 4193 of the series Lecture Notes in Computer Science pp 392-401.

[3] Bayliss et al. (2011). Neural Networks to Guide the Selection of Heuristics within Constraint Satisfaction Problems. Volume 6718 of the series Lecture Notes in Computer Science pp 250-259

[4] Alanazi, Fawaz. (2016). Limits to Learning in Reinforcement Learning Hyper-heuristics. Evolutionary Computation in Combinatorial Optimization Volume 9595 of the series Lecture Notes in Computer Science pp 170-185

[5] Goodfellow. (2016) Deep learning Book, part 1, chapter 5 : Machine learning basics. Retrieved from http://www.deeplearningbook.org/contents/ml.html

[6] Burke et al. (2003) Hyper-heuristics: An emerging direction in modern search technology. Retrieved from http://www.cs.nott.ac.uk/~pszgxk/papers/hhchap.pdf

[8] Fu, Michael.(2006) Stochastic Gradient Estimation. Retrieved from https://arxiv.org/pdf/1611.05763.pdf

[9] Silver, David. (2014) Playing Atari with Deep Reinforcement Learning. retrieved from https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[10] Burke et al. (2003).A Tabu-Search Hyperheuristic for Timetabling and Rostering. 1023 Volume 9 of Journal of heuristics, Issue 6, pp 451–470.

[11] University of nebraska, Lincoln. Summary of bin packing algorithms. Retrieved from http://www.math.unl.edu/~s-sjessie1/203Handouts/Bin%20Packing.pdf

[12] Hinton, Geoffrey. 2012. Lecture 6a overview of mini–batch gradient descent. Coursera Lecture slides Retrieved from https://class.coursera.org/neuralnets-2012-001/lecture,[Online.

[13] Silver et al.(2016) Mastering the game of go with deep neural networks and tree search. Nature, 529(7587):484–489.

[14] Silver et al. (2016) Learning to reinforcement learn. Retrieved from https://arxiv.org/abs/1611.05763 [15] Judea Pearl, [1984] Heuristics: Intelligent search strategies for computer problem solving. ISBN:0-201-05594-5

[16] Silver, David. (2015). Lecture 1 Introduction to Reinforcement Learning. UCL online lectures. Retrieved from http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/intro_RL.pdf