DeepProbLog Tasks

Davide Lusuardi, 223821, *davide.lusuardi@studenti.unitn.it*

***Abstract*—DeepProbLog is an extension of ProbLog that integrates Probabilistic Logic Programming with deep learning by means of neural predicates. The neural predicate represents probabilistic facts whose probabilities are parameterized by neural networks.**

DeepProbLog is a framework where general-purpose neural networks and expressive probabilistic-logical modeling and reasoning are integrated in a way that exploits the full expressiveness and strengths of both worlds and can be trained end-to-end based on examples.

The aim of this report is to show how to solve AI tasks that require the integration of high-level reasoning and low-level perception. We focus on the multi-digit MNIST octal-division task that consists in computing the division between two lists of MNIST digits representing multi-digit octal numbers. Using DeepProbLog we are able to solve the task given that supervision is only present at the output side of the probabilistic reasoner and considering that the approach can be extended to multi-digit numbers without being explicitly trained on them.

I. Introduction

THERE are many tasks in AI that require both low-level perception and high-level reasoning but the integration of the two is an open challenge in the field of artificial intelligence. Today, low-level perception is typically achieved by deep neural networks, while high-level reasoning is typically handled using logical and probabilistic representations and inference. Even if deep learning can create intelligent systems used to interpret images, text and speech with unprecedented accuracy, there is a growing awareness of its limitations: deep learning requires large amounts of data to train a network and the models are black-boxes that do not provide explanations and cannot be modified by domain experts.

The abilities of deep learning and probabilistic logic approaches are complementary: deep learning excels at low-level perception and probabilistic logic excels at high-level reasoning. Recently, there has been a lot of progress in both deep learning and high-level reasoning areas and today there exists approaches able to integrate logical and probabilistic reasoning with statistical learning.

DeepProbLog [1] is one possible approach. It is a neural probabilistic logic programming language that incorporates deep learning by means of neural predicates. With DeepProbLog, instead of integrating reasoning capabilities into a complex neural network architecture, the authors have decided to start from an existing probabilistic logic programming language, ProbLog [2], that has been extended with the neural predicates. In this way, the framework exploits the full expressiveness and strengths of general-purpose neural networks and expressive probabilistic-logical modeling and reasoning and can be trained end-to-end based on examples.

In this work, we introduce DeepProbLog explainig the basics of the language in Section II. Subsequently, in Section III we present the multi-digit MNIST octal-division task and how can be solved using DeepProbLog. Finally, in Section **??** and **??** we discuss the results obtained.

II. DeepProbLog

In this section, we explain briefly the basics of probabilistic logic programming using ProbLog and how to extend it to obtain DeepProbLog.

A ProbLog program is made of a set of ground probabilistic facts $\mathcal{F}$ of the form $p :: f$ where p is a probability and f is a ground atom, and a set of rules $\mathcal{R}$. One extension introduced for convenience that is nothing else than syntactic sugar are the annotated disjunctions (ADs). An annotated disjunction is an expression of the form

$$p_1 :: h_1; \ldots; p_n :: h_n \text{: } -b_1, \ldots, b_m. \tag{1}$$

where the p_i are probabilities that sum to at most one, the h_i are atoms and the b_j are literals. Given the AD of Eq. 1, when all b_i hold then one of the heads h_j will be true with probability p_j or none of them with probability $1 - \sum p_i$. Several of the h_i may be true at the same time if they also appear as heads of other ADs or rules. Since ADs are syntactic sugar, they do not change the expressivity power of ProbLog and can be alternatively modeled as facts and logical rules [3].

While in ProbLog probabilities are explicitly specified as part of probabilistic facts or ADs, in DeepProbLog probabilities are specified through neural networks. A DeepProbLog program is a ProbLog program that is extended with a set of ground neural annotated disjunctions (nADs) of the form

$$nn(m_q, \vec{t}, \vec{u}) :: q(\vec{t}, u_1); \ldots; q(\vec{t}, u_n) : -b_1, \ldots, b_m \tag{2}$$

where nn is a reserved keyword that stands for 'neural network' and m_q is the identifier of the neural network model, $\vec{t} = t_1, \ldots, t_k$ is a vector of ground terms representing the inputs of the neural network for predicate q, $\vec{u} = u_1, \ldots, u_n$ is the vector of the possible output values of the neural network and b_i are atoms. The NN defines a probability distribution over its output values $\vec{u}$ given the input $\vec{t}$. The neural network could be of any type, e.g. a recurrent or a convolutional network, but its output layer, which feeds the corresponding neural predicate, needs to be normalized. Note that a neural AD realizes a regular AD $p_1 :: q(\vec{t}; u_1); \ldots; p_n :: q(\vec{t}; u_n) : -b_1; \ldots; b_m$ after a forward pass on the neural network.

A. DeepProbLog inference

Inference in DeepProbLog closely follows that in ProbLog. ProbLog inference proceeds in four steps, the first step is the grounding step, in which the logic program is grounded with respect to the query, generating all ground instances of clauses the query depends on. This step uses backward reasoning

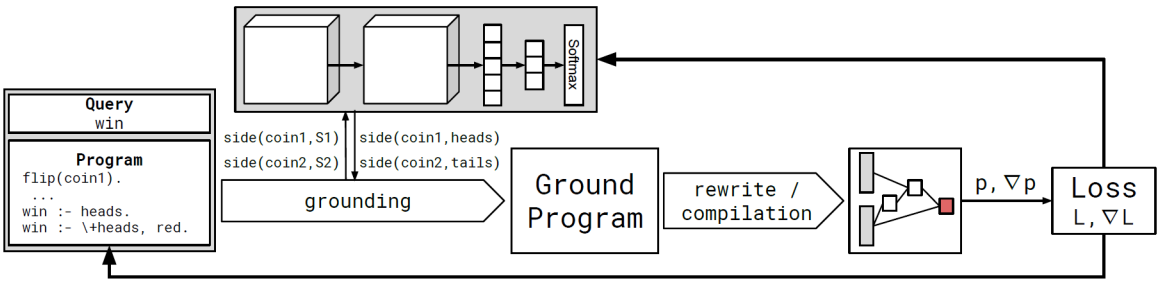


Fig. 1: The learning pipeline [2].

to determine which ground rules are relevant to derive the truth value of the query, and may perform additional logical simplifications that do not affect the query's probability.

The second step rewrites the ground logic program into a formula in propositional logic that defines the truth value of the query in terms of the truth values of probabilistic facts. We can calculate the query success probability by performing weighted model counting (WMC) on this logic formula even if it is not efficient.

The third step is knowledge compilation that compiles the logic formula into a Sentential Decision Diagram (SDD, [4]), a form that allows for efficient weighted model counting. SDDs are a subset of deterministic decomposable negational normal forms (d-DNNFs) and as these ones allow for polytime model counting.

The fourth step transforms the SDD into an arithmetic circuit (AC). The AC has the probabilities of the probabilistic facts on the leaves and is obtained replacing the OR nodes with addition and the AND nodes by multiplication. The AC is then evaluated to calculate the WMC.

Inference in DeepProbLog works as in ProbLog, the only difference is that we need to instantiate nADs and neural facts respectively to regular ADs and probabilistic facts. During the grounding step, we obtain ground nADs and ground neural facts. The concrete parameters are determined subsequently by making a forward pass on the relevant neural network with the ground input.

B. Learning in DeepProbLog

Learning in DeepProbLog consists in jointly train the learnable parameters in the logic program, i.e. the parameters of probabilistic facts, and learnable parameters of the neural network in DeepProbLog programs. DeepProbLog programs are trained in a discriminative training setting, which is called *learning from entailment* [5]. This approach proceeds as follows: given a DeepProbLog program with parameters X, a set Q of pairs (q, p) with q a query and p its desired success probability, and a loss function L, compute:

$$\arg\min_{\vec{x}} \frac{1}{|Q|} \sum_{(q,p)\in Q} L(P_{X=\vec{x}}(q), p) \tag{3}$$

In contrast to the approach proposed by Gutmann et al. [6], gradient descent is used rather than EM, as this method allows for seamless integration with neural network training. It is worth noting that we can use the same AC that ProbLog uses for inference for gradient computations as well: this AC is a differentiable structure, as it is composed of addition and multiplication operations only. The framework relies on the automatic differentiation capabilities already available in ProbLog to derive the gradients. More specifically, to compute the gradient with respect to the probabilistic logic program part, it relies on Algebraic ProbLog (aProbLog [7]), a generalization of the ProbLog language, and inference to arbitrary commutative semirings, including the gradient semiring [8]. An overview of the approach is shown in Figure 1. Given a DeepProbLog program, its neural network models, and a query used as training example, we first ground the program with respect to the query, getting the current parameters of nADs from the external models, then use the ProbLog framework to compute the loss and its gradient, and finally use these to update the parameters in the neural networks and the probabilistic program.

In the following section, we present and discuss the multi-digit MNIST octal-division task and how can be solved using DeepProbLog.

III. Multi-digit MNIST Octal-division Task

The multi-digit MNIST octal-division task can be formulated as follows: given two lists of MNIST images representing multi-digit octal numbers, we want to perform the division between the two numbers. We constrain the problem assuming that the second number is an integer divisor of the first. Moreover, we generate the training set in such a way that it contains only single-digit numbers in order to show how the program can be extended to multi-digit numbers without being explicitly trained on them.

The main goal is to define the predicate `division(X,Y,Z)`, where X and Y are lists of images of handwritten digits from the MNIST dataset that represent two base-8 numbers and Z is the base-8 number corresponding to the result of the division between X and Y. While such a predicate can be learned directly by a standard neural classifier, such an approach cannot incorporate background knowledge such as the definition of the octal division between two numbers.

The DeepProbLog program implemented to solve the task is shown in Fig. TODO. The program specifies the neural annotated disjunction

$$nn(mnist\_net, [X], Y, [0, 1, 2, 3, 4, 5, 6, 7]) :: digit(X, Y). \tag{4}$$

where nn is a reserved functor, $mnist\_net$ is a neural network that classifies MNIST digits defining a probability distribution over the domain $1, 2, 3, 4, 5, 6, 7$ and $digit$ is the corresponding neural predicate. The output layer of the network that feeds the $digit$ neural predicate should be normalized in order to get proper probability values. In general, the neural network, apart from the output layer, could be of any kind, e.g., a recurrent network for sequence processing or a convolutional network for image processing. We decided to implement the standard convolutional network used for MNIST images since it is the more suitable NN for MNIST images.

The program contains also some rules, as shown in Fig. TODO, in order to obtain the base-8 number from the input list of MNIST digits, convert it to a decimal number, apply the standard Prolog operator for the integer division between decimal numbers, and finally convert back the result to an octal number.

A. Training and test sets

Starting from the MNIST dataset we constructed the training and test sets as follows. The training set is composed of pair of images that represent single-digit base-8 numbers and the test set is composed of two lists of images that represent three-digit base-8 numbers. In both cases, the second number is a divisor of the first one. After removing the images representing digit 8 or 9 from the MNIST training and test sets, the algorithm proceeds constructing a random number n_1. Then, a random divisor n_2 is constructed: we select a random number in the range $[1, n_1]$ and we select the nearest number that is a divisor of n_1. The images used to construct the two numbers are removed from the MNIST dataset. Proceeding in this way until no more pairs can be constructed, we managed to generate a training set made of 22958 pairs and a test set made of 1462 pairs. We fixed the random seed to obtain always the same dataset. In this way, there are no repeated images in the training and test sets but not all the MNIST images will be used: we used 45916 of the available 48200 training images of MNIST and 7835 of the available 8017 test images of MNIST. Note that the divisors in the test set can have up to three digits, whereas the dividends are forced to have exactly three digits. Finally, the train and test queries are generated based respectively on the train and test sets to be subsequently used by DeepProbLog during training.

B. Neural network model

In our implementation, the neural network model used to classify the MNIST digit images has a total of 44256 parameters and is a basic architecture based on the one discussed in [1]. The model architecture can be described as follows: it consists of 2 2D-convolutional layers both with a kernel size of 5 and with respectively input channels of 1 and 6 and output channels of 6 and 16; the convolutional layers are stacked with a 2D max pooling layer, with a kernel size of 2 and stride of 2, and a rectified linear unit layer. After these layers, the model has 3 fully connected layers of sizes 120, 84 and 8 with a rectified linear unit layer between them. The last layer is followed by a softmax layer in order to get a probability value. The learning process optimizes the cross-entropy loss between the predicted and desired query probabilities as implemented by the function $train\_model$ that is part of the DeepProbLog framework [**TODO**], performing gradient accumulation instead of mini-batching. As optimizer we used Adam with a learning rate of 0.001 for the network parameters and SGD for the logic parameters.

C. Results

DeepProbLog managed to achieve TODO 97% of accuracy on the validation set after TODO iterations as shown in Fig. 2a. The learning curve is shown in Fig. 2b.

IV. Methodologies and Implementation

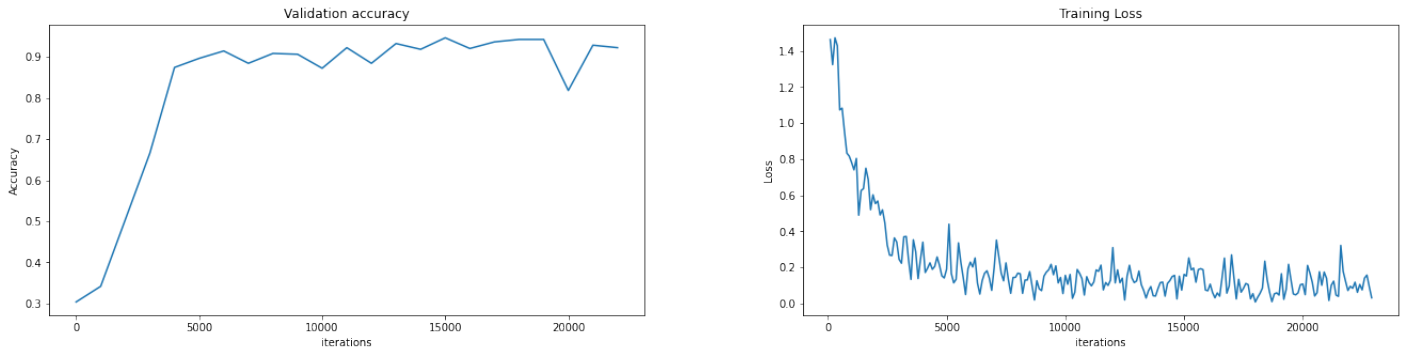
Starting from the Yellow-Spaceship game [**Yellow-Spaceship**], we have adapted the code to our needs. In particular, we have modified the code that reads the action from the user keyboard in order to be able to substitute the user input with an action computed by a program and apply it to the game. The controlling action can be computed by an ANN evolved using NEAT, or by a tree-based program evolved using GP, passing them the appropriate information about the current state of the game. Moreover, the possibility to not show the game has been introduced in order to speed up the execution.

The inputs from the game environment for both NEAT and GP individuals are the following: the x coordinate of the battleship; the velocity of the battleship; the x coordinate of the first and second alien (if any, otherwise they are set to 0); the x and y coordinates of the first and second closest lasers (if any, otherwise they are set to 0); the x coordinate of the enemy spaceship (if any, otherwise it is set to 0). In this way, there are a total of 9 arguments of type float passed to the individuals as input that can be used to decide the action to perform. The performance of an individual can be evaluated based on how much it moves on in the game.

The fitness function for the individuals of both the algorithms that we decided to use can be formulated as follows:

$$\begin{aligned} fitness = & alien\_kills * 10 + \\ & enemy\_spaceship\_kills * 50 + \\ & \sum_{h \in battleships\_healths} \frac{h}{50} + \\ & \sum_{a \in aliens} \frac{cur\_alien\_health - a.health}{cur\_alien\_health} * 10 + \\ & \sum_{s \in enemy\_spaceships} 50 - s.health \end{aligned} \tag{5}$$

where $alien\_kills$ is the number of aliens killed, $enemy\_spaceship\_kills$ is the number of enemy spaceships



(a) Validation accuracy

(b) Training loss

Fig. 2: Multi-digit MNIST octal-division task learning curves: accuracy on the multi-digit test set on the left 2a, training loss on the single-digit training set on the right 2b.

killed, $battleship\_healths$ is a vector of battleship health at the beginning of each level and cur_alien_health is the initial alien health at the last level reached.

The fitness function should be maximized and takes into account the number of aliens and enemy spaceships killed, the battleship health at each level and the health of aliens and enemy spaceships at the last level reached. There is no intermediate reward and the overall fitness is returned at the end of simulation.

In order to handle cases in which the simulation takes too much time because the battleship is not killed, we introduced a maximum threshold on the number of frames: the game will be stopped after 1 million frames. The threshold permits to prevent the execution from getting stuck and also incentivizes the NEAT and GP algorithms to learn how to reach higher levels with the same number of frames. When the best individual execution is shown, this limit is released in order to obtain the actual fitness without interrupting the game.

To speed up the evolution process, individuals are evaluated only every 10 frames to obtain the action to perform: the previous action is applied for the rest of 9 frames. This value should be sufficient to permit the individual to have enough control over the battleship actions and not be killed by enemies. In this way the evaluation process proceeds about 10 times faster and the battleship movements look smoother.

A. NEAT

The implementation is based on the NEAT-Python library [**NEAT-Python**], which is a pure Python implementation of NEAT, with no dependencies other than the Python standard library. We decided to evolve a feed forward neural network with the possibility of learning skip connections instead of a recurrent neural network since the actual state of the game is sufficient to evolve the neural network and no more information is required. The network has 6 output nodes and each of them encodes a particular action that the agent has to perform: go left; go left and fire; stay still; stay still and fire; go right; go right and fire. The output node with the maximum value is taken to decide the action to perform.

We decided to adopt the following algorithm parameters:

- $num\_runs = 10$, $num\_generations = 200$ and $no\_fitness\_termination = True$
- $pop\_size = 100$ and $reset\_on\_extinction = True$
- $activation = sigmoid$ and $aggregation = sum$
- $conn\_add\_prob = 0.5$ and $conn\_delete\_prob = 0.5$
- $enabled\_default = True$ and $enabled\_mutate\_rate = 0.01$
- $feed\_forward = True$ and $initial\_connection = full\_nodirect$
- $node\_add\_prob = 0.2$ and $node\_delete\_prob = 0.2$
- $num\_inputs = 9$, $num\_hidden = 9$ and $num\_outputs = 6$
- $species\_fitness\_func = max$, $max\_stagnation = 20$ and $species\_elitism = 2$
- $elitism = 2$, $survival\_threshold = 0.2$ and $min\_species\_size = 2$

All the configuration parameters used by the algorithm can be modified in the configuration file named 'configNEAT.txt' present in the root directory of the project.

B. GP

The implementation is based on the DEAP library [**DEAP**], which is a Python evolutionary computation framework which provides the main blocks for building Genetic Programming algorithms. Strongly typed genetic programming (STGP) is an enhanced version of genetic programming which enforces data type constraints.

A tree-based program has been evolved using STGP. The output of the program is one of the 6 encoded actions that the agent can perform. To encode the actions, we defined one class for each of them with the following encoding: A, go left; B, go left and fire; C, stay still; D, stay still and fire; E, go right; F, go right and fire.

The terminal set is composed of the action classes, some float values (5, 10, 15, 20, 25, 30) and the boolean values ("true", "false"). The function set is composed of the boolean operators "greater than", "equal", "and", "or" and "negation", the arithmetic operators "add", "subtract" and "multiply" and the "if_then_else" function that can outputs a float value or an action class. Since the "if-then-else" function is the only

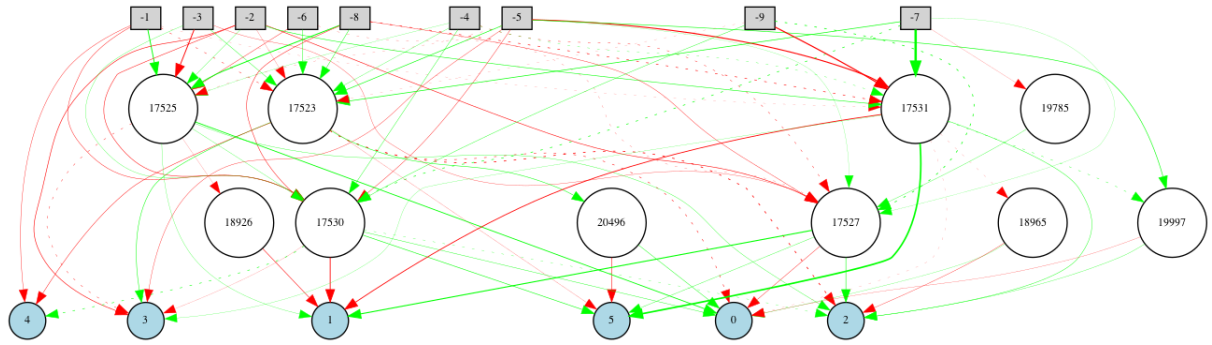


Fig. 3: Best ANN generated by NEAT.

function that outputs an action, it should always be at the root of the tree.

We decided to adopt the following algorithm parameters:

- $num\_runs = 10$ and $num\_generations = 200$
- $pop\_size = 100$
- $crossover\_prob = 0.5$
- $mutation\_prob = 0.5$
- $hof\_size = 4$
- $max\_tree\_size = 500$
- $max\_tree\_height = 10$
- As $expr\_init$ we used $gp.genFull$ with $min\_ = 1$ and $max\_ = 3$
- As $select$ we used $tools.selTournament$ with $tournsize = 7$
- As $mate$ we used $gp.cxOnePoint$
- As $expr\_mut$ we used $gp.genFull$ with $min\_ = 0$ and $max\_ = 2$
- As $mutate$ we used $gp.mutUniform$
- As $algorithm$ we used $deap.algorithms.eaSimple$

All the configuration parameters used by the algorithm can be modified in the configuration file named 'configGP.txt' present in the root directory of the project.

V. Results

In this section we will present and compare the results obtained with the two approaches. Some results obtained from a randomly piloted battleship are considered as baseline.

For both NEAT and GP, the evolution process stores at each run the best program generated with the relative statistical results and plots in the 'runs' folder. In this way, it is possible to launch the program later and observe through the game interface how the battleship behaves.

A. NEAT

The NEAT algorithm managed to evolve a ANN that can reach level 42 in the game with a fitness of 1114 after about 160 generations. The ANN is represented in Fig. 3. As we can see the network is quite simple, with a small number of nodes and connections: it has just 10 hidden nodes and 63 enabled connections (that are represented by the solid arrows).

In Fig. 4, we can observe that the best fitness increases significantly in just a few particular generations, whereas the average fitness does not grow significantly and stays under 200. With this fitness trend, we can also notice that the speciation decreases significantly after 20 generations due to the stagnation of a lot of species: after about 60 generations, just the 2 elitism species will survive as shown in Fig. 5.

A weak point of NEAT is that the generated ANN is not really interpretable and we cannot understand why the network produces certain output values.

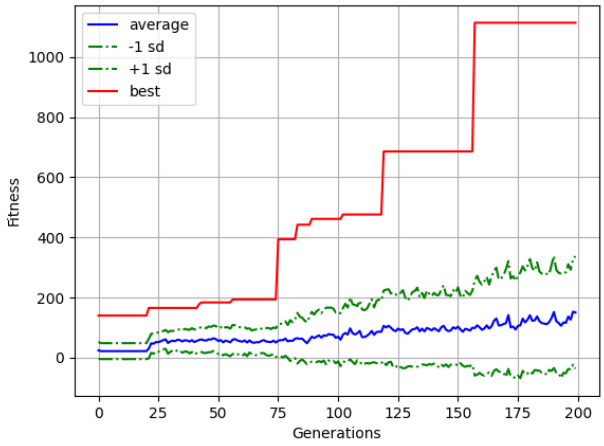


Fig. 4: Fitness trend of the NEAT run that has generated the best ANN.

B. GP

The GP algorithm managed to evolve two tree-based programs that reach over 4000 fitness in the simulation.

The first program reaches level 151 in the game with a fitness of 4061 during the evolution process and level 194 with a fitness of 5212 without limiting the number of frames. The tree of the program is represented in Fig. 6a. As we can see the tree is quite complex, with 259 nodes and a depth of 10 (the maximum depth allowed). In Fig. 7a, we can observe that the best fitness improves a lot in the first 60 generations reaching the frame threshold.

The second program reaches level 162 in the game with a fitness of 4342 during the evolution process and level 189 with a fitness of 5076 without limiting the number of frames. The tree of the program is represented in Fig. 6b. As we can see

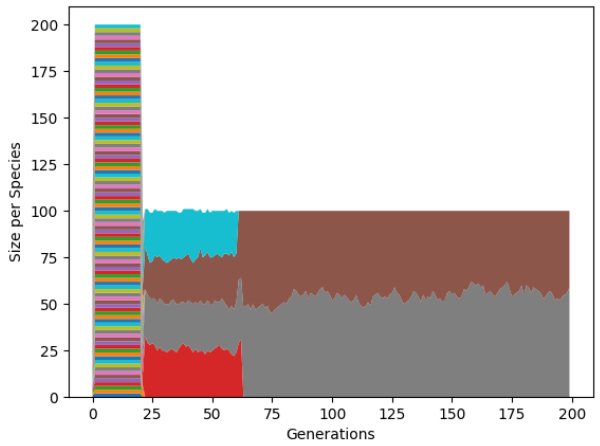


Fig. 5: Speciation trend of the NEAT run that has generated the best ANN.

the tree is a bit less complex than the previous one, with 177 nodes and a depth of 10 (the maximum depth allowed). As shown in Fig. 7b, the best fitness improves significantly only after 100 generations, reaching the frame threshold.

In both cases, for the remaining generations, the best fitness trend increases slightly, probably also due to this limitation on the number of frames. The average fitness trend is smoother and follows the best trend with a similar increase, reaching about 3000 fitness with a large variance (over 1000).

A weak point of GP is that the generated tree is a bit complex and can be simplified a lot, e.g. simplifying some "if_then_else" statements that have as condition pure boolean values as can be seen in Fig. 6.

C. Comparison

With both techniques, we managed to find a program that is able to play the game well, learning to fire almost always to hit enemies while dodging their lasers.

Considering our scenario, GP demonstrated to have a bigger potential: thanks to the tree-structure and the functions provided, it is able to learn more complex strategies reaching level 194, much more than the best evolved ANN. NEAT, instead, struggles to evolve the network reaching at most level 42. NEAT is probably more suitable for tasks where a program would not be the best option or even it would be impossible to build.

Moreover, a tree with a limited size can be more interpretable than a ANN and we can easily provide an explanation for what the agent is doing.

Observing the agent in action we can say that both NEAT and GP agents move quite smoother, even though neither of them resemble humans. GP agent, in the end, looks smarter and tends to stay in the middle of the playing area dodging all the lasers, whereas NEAT tends to remain in the corners.

Across multiple runs, NEAT seems to obtain more stable results with respect to GP that manages to reach very good results only few times, as shown in Fig. 8. Both the techniques find a program which performs much better than a randomly piloted battleship that on average is able to reach only level 3 with a fitness of about 45 as shown in Fig. 8c.

VI. Conclusions

As demonstrated, NEAT and GP are very good approaches able to find a way to play the game well. Even though, across multiple runs, NEAT seems to obtain more stable results, GP has proved to have a bigger potential thanks to the tree-structured individuals and overall manages to reach very good results.

One of the issues that we encountered using GP is how to manage the different input types: one may implement functions taking into account inputs of different types or, alternatively, can use a strongly typed primitive set as we have done.

Another issue is that the execution of the algorithms takes a lot of time and computational resources, especially when the frame threshold is reached. Due to this, the number of runs and individuals should be limited.

In the end, with this project we learned a lot about this field, in particular how it is possible to apply bio-inspired techniques to real-world problems and benefit from them. They have proved to be a valid alternative to classic back-propagation algorithms for ANN and we have learned how to apply them to Reinforcement Learning tasks using the fitness as a reward.

References

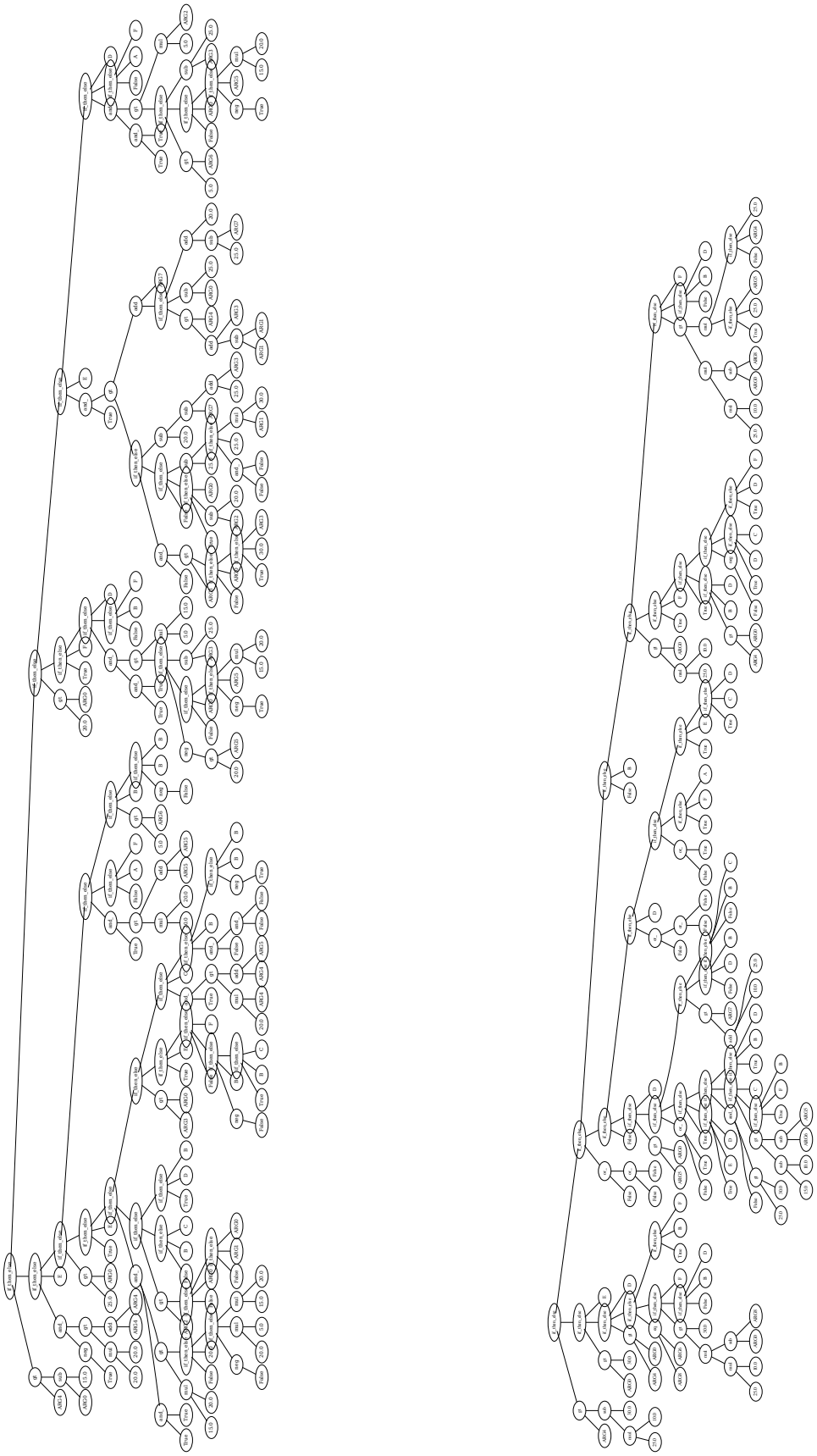
[1] Robin Manhaeve et al. "DeepProbLog: Neural Probabilistic Logic Programming". In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio et al. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/dc5d637ed5e62c36ecb73b654b05ba2a-Paper.pdf.

[2] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. "ProbLog: A Probabilistic Prolog and Its Application in Link Discovery". In: Jan. 2007, pp. 2462–2467.

[3] Luc De Raedt and Angelika Kimmig. "Probabilistic (logic) programming concepts". In: *Machine Learning* 100.1 (July 2015), pp. 5–47. ISSN: 1573-0565. DOI: 10.1007/s10994-015-5494-z. URL: https://doi.org/10.1007/s10994-015-5494-z.

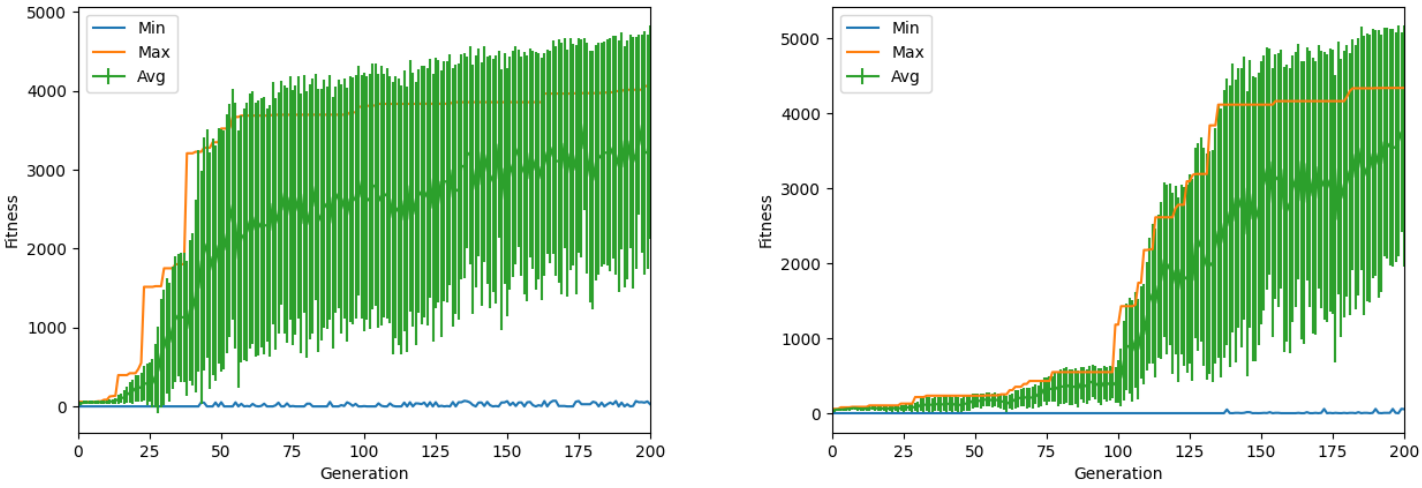
[4] Adnan Darwiche. "SDD: A New Canonical Representation of Propositional Knowledge Bases". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, 2011, pp. 819–826. ISBN: 9781577355144.

[5] Michael Frazier and Leonard Pitt. "Learning From Entailment: An Application to Propositional Horn Sentences". In: *Machine Learning Proceedings 1993*. San Francisco (CA): Morgan Kaufmann, 1993, pp. 120–127. ISBN: 978-1-55860-307-3. DOI: https://doi.org/10.1016/B978-1-55860-307-3.50022-8. URL: https://www.sciencedirect.com/science/article/pii/B9781558603073500228.



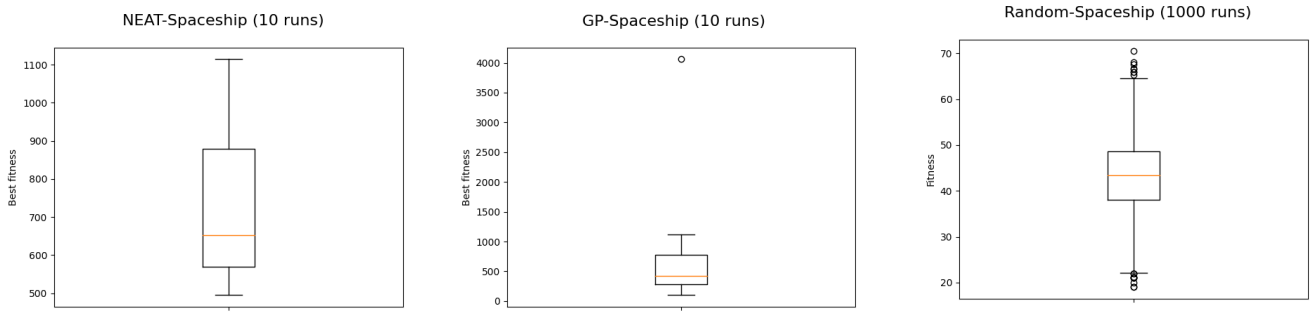
(a) (b)

Fig. 6: Best tree-based programs generated by GP.



(a) (b)

Fig. 7: Fitness trend of the GP runs that have generated the best programs.



(a) NEAT boxplot. (b) GP boxplot. (c) Random-piloted battleships boxplot.

Fig. 8: Comparison between the boxplots of NEAT, GP, and randomly piloted battleships.

[6] Bernd Gutmann et al. “Parameter Learning in Probabilistic Databases: A Least Squares Approach”. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Walter Daelemans, Bart Goethals, and Katharina Morik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 473–488. ISBN: 978-3-540-87479-9.

[7] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. “An Algebraic Prolog for Reasoning about Possible Worlds”. In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI’11. San Francisco, California: AAAI Press, 2011, pp. 209–214.

[8] Jason Eisner. “Parameter Estimation for Probabilistic Finite-State Transducers”. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL ’02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 1–8. DOI: 10.3115/1073083.1073085. URL: https://doi.org/10.3115/1073083.1073085.