Multi-output and Structured-output Learning with Chaining Methods (Agent-based inference)

Alicia Fortes Machado Aloysio Galvão Lopes
Igor Albuquerque Silva Iago Martinelli Lopes

March 2021

1 Abstract

In this work we propose two reinforcement learning approaches to tackle the inference task in Classifier Chains for the Multi-Label Classification problem. Finding the best path in the associated probability tree can take exponentially long times, and in order to tackle that we rely on how similar problems have been dealt with such as Go [15] and graph path [14]. Thus, we implemented two reinforcement learning algorithms, one that uses a Deep Q-Learning algorithm and another that uses a Monte Carlo Tree Search as the policy improvement method. We conclude that both are able to reach baseline methods results in small datasets, but require extensive exploration to do so.

2 Introduction

2.1 Multi-Label Classification Problem

Multi-Label Classification (MLC) has attracted attention over the past few years due to its capacity of improving the prediction of problems as such movie genre, textual data and protein function classification. The key difference when compared to other classification problems is that the input can have more than one relevant label, which means that it can be classified in multiple labels, which aren't necessarily independent.

There are different methods in the literature to tackle this problem, such as Binary Relevance (BR) [9], Recurrent Neural Networks [5] and Classifier Chains (CC) [12]. The last one is of particular interest, given that it's one of the state of the art options for MLC and that it has been researched for a couple of years and showed improvements along this period of time. It's a more refined technique when compared to the Binary Relevance one. We say that because BR predicts independently each label, transforming the multi-label classifications into a series of independent binary classifications, while the classifier chain technique considers that previously predicted variables might influence on the next ones.

2.2 Classifier Chain

A classifier chain consists of a chain of predictors. In the most common chain type, the input of each predictor is the concatenation of the input and the output of the previous one. This is shown in Figure 1 and the prediction of each label is done as shown in Equation 1:

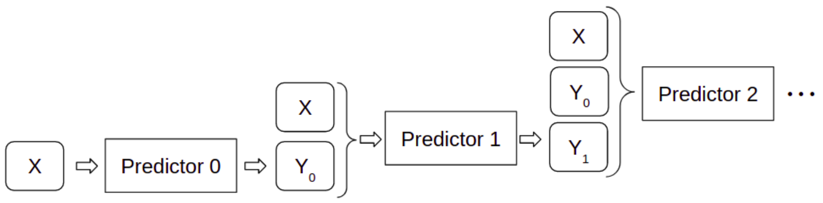

Figure 1: Example of how a classifier chain works. The X corresponds to the input and the Y_i corresponds to the output of predictor i.

$$\hat{y}_i = \underset{y_i \in \{0,1\}}{\text{argmax}} P(y_i|\mathbf{x}, y_1, ..., y_{i-1}) \tag{1}$$

Notice that we can have many different permutations of the order of labels. The discussion behind which order to use in our CC is not the focus of our work. However, we will briefly discuss this topic in Section 3 and use one of these techniques (Genetic Algorithm) to choose the order of labels for our work.

Nevertheless, notice that, in this first definition, each predictor greedily chooses to classify the label into true or false, when the actual final output should be the set of labels that maximizes the final joint probability. The approach that pursues the maximization of this final joint probability is called probabilistic classifier chains (PCC) and it is an extension of the previous definition. So, now, we want our prediction to be as it is in Equation 2.

$$\hat{\mathbf{y}} = \underset{y \in \{0,1\}^L}{\text{argmax}} P(y_1|\mathbf{x}) \prod_{i=2}^{L} P(y_i|\mathbf{x}, y_1, ..., y_{i-1}) \tag{2}$$

Once trained, the prediction step in the probabilistic classifier chain can be seen as a probability tree, where each path from root to leaf represents one combination of labels $\mathbf{y} \in \{0,1\}^L$. The probabilities in the non-leaf nodes are the predictions for the next label from the estimator considering the past actions. Figure 2 shows an example of this tree.

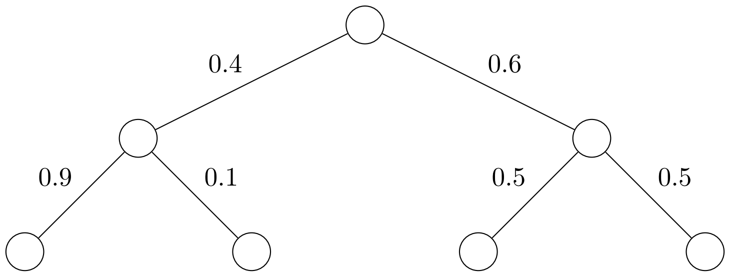

Figure 2: Example of inference tree given by a classifier chain. The numbers represent the estimator's outputs at each label of the sample, which are represented as levels on the tree.

However, even after considering an order for the labels and constructing the probability tree, we still need to find a way to go through the last one with a good quality-time trade-off. Maximizing the probabilities of the chosen path in this tree is important to maximize the model's performance. Some simple methods to do that like Greedy and Exhaustive Search are described in Section 3, but they fail to balance both factors. Hence the need to find better methods to go through these trees.

2.3 Method

In our final method we had two main inspirations. This first one was AlphaGo Zero [15] which describes an state of the art reinforcement learning method to solve the Go game. What caught our attention was that in their approach one of the main challenges is to discover how to traverse a search tree (given by the games possibilities), therefore we could adapt what they did to our needs. The second one was M-walk [14], which uses Monte Carlo Tree Search, one of the main strategies used in AlphaGo Zero [15], to explore graphs (in our case we should explore a tree which is also a graph by definition).

We decided to follow an approach that merges different techniques together: Deep Q-Learning, Policy Learning and Monte Carlo Search Tree Search (MCTS). For each state, we will use MCTS to search promising paths and then take the one with greatest expected value. We also created an agent based only on Deep Q-Learning, in order to compare with the method just explained. The estimation of the expected value on both approaches and the policy will be done using neural networks, as explained in Section 4.

Finally, in Section 5 we present our results, in which we compare our proposed methods with other baseline inference methods. We considered different methods such as Greedy, Exhaustive Search, Monte Carlo and ϵ-approximate inference. Our comparison was done by measuring subset 0/1 and Hamming losses, average number of explored nodes and average inference time for each approach.

3 Related Work

3.1 Inference Methods

Inference methods are algorithms applied in the prediction step of a CC. Due to the inherent necessity to determine an order of the estimators in the chain, the prediction step may be done in different manners. An example of this is if in the prediction of an instance of the data the first estimator predicts its label with equal probabilities of `True` and `False`. It may be interesting in this case to analyze both possibilities when feeding the label value for the next estimators, because they may give a high probability if the first label is `True`, for example, and low probability if it is `False`.

Many CC inference methods have been proposed before. In this subsection we'll go over some of those, more precisely the greedy and exhaustive-search methods [11], ϵ-approximation inference [1], beam-search [3] and two Monte Carlo approaches [1, 10]. An overview and comparison of these algorithms was done in [4], and in Section 5 we'll present a reproduction of their results, in comparison with our proposed methods.

Greedy and Exhaustive Search

The greedy method was first presented alone as CC, and was followed by the exhaustive search extension, the PCC. The former is the simplest and quickest way of doing an inference: each estimator will use the labels predicted by the previous ones in the chain to make their inference. The exhaustive search, however, will use every possibility of labels on each estimator and take the combination with the lowest loss to make its prediction.

A way to visualize this is by drawing a tree of labels, like the one in Figure 3. This tree represents a chain predicting only two labels, in which $\mathbf{x}$ is the test instance and the objective is to predict a vector $\mathbf{y}$. In a greedy search inference, the chosen path is always the ones following the highest conditional probability $\hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$, which are the probabilities given by each individual estimator, in this case 0.6 and 0.5. The final probability is $\hat{P}(\mathbf{y}|\mathbf{x}) = 0.3$. An exhaustive search would transverse all of the four paths one by one and find a better probability of 0.36.

The exhaustive search can also maximize another expression: $\sum_i \hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$, which is the sum of all of the conditional probabilities, instead of the multiplication. The difference in this is that it would minimize the Hamming loss, which is given by the ratio of incorrectly predicted labels over the total. In the example the exact-match (or subset 0/1) loss is minimized, which is 1 if all labels are correctly predicted and 0 otherwise [10].

ϵ-Approximation

The exhaustive search method's runtime is $\mathcal{O}(2^L)$, where L is the number of labels. Naturally, it is necessary to find alternatives to find better predictions than the greedy algorithm which can be applied to datasets with large number of labels. The first example of those is the ϵ-Approximation method.

The idea behind this algorithm is to start as an exhaustive search and stop looking through a path if its joint probability $\hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$ becomes less than a value ϵ. This way, the number of visited paths is limited by $\mathcal{O}(L/\epsilon)$ [1, 4].

Beam Search

Another way of limiting the number of total paths is by specifying a number of maximum paths to explore at each level of the tree. The Beam Search algorithm transverses the tree in a breadth-first-search, limiting the number of paths being explored by a value b, called the beam [3]. The paths with lowest score are discarded (as in the exhaustive search, the score can be the one that minimizes the exact-match or Hamming loss.) in each level. This algorithm is $\mathcal{O}(Lb)$ in time.

Monte Carlo Sampling

There are two different Monte Carlo techniques for inference, which are based on the same idea. In both of them a number k of paths are tried out, and they differ in the way the best path is chosen.

Each path will be transversed in a probabilistic way following the distributions of each estimator. This way, in each node of the tree a random number R is generated and the label will be `False` if $\hat{P}(y_i = \texttt{False}|y_{i-1},...,y_0,\mathbf{x}) < P$ and `True` otherwise. Figure 3 shows an example of a path explored in a CC with two estimators.

The first way of selecting the best path between the ones explored is by taking the one which occurred the most, the mode [16]. In the second approach the best path is selected by taking the one with the highest score [10]. The advantage of the latter is that it doesn't require the calculation of the mode, which requires storing all of the unique explored paths.

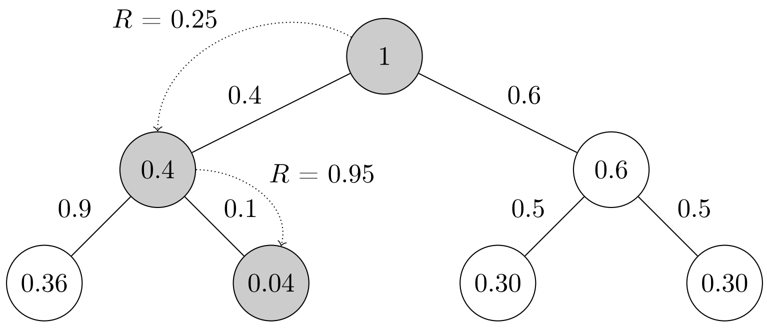

Figure 3: An example of a Monte Carlo inference path in a chain with two estimators. Darker nodes represent visited nodes. The number on the edges represent $\hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$ and the numbers inside the nodes represent $\prod_i \hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$. Example based on [4].

3.2 Chain Structure

In the context of chain classification, a part as crucial as going through the chain is to construct it in the most optimal way possible, because different chain structures produce different results, directly affecting the quality of predictions, performance and interpretability.

An early approach to this problem was using ensembles of randomly selected chain structures [9], and then combine prediction results. However, previous work has shown that this method is much more efficient to reduce the effect of bad chains than to value the outcome of good chains [12]. Being an NP-Hard problem, a brute force approach is far from being satisfactory, so many methods have been developed over the years in order to give an efficient order for the chain. Among them, we will present below the Evolutionary approach [2], which we've also implemented.

Genetic Algorithm

This method uses Genetic Algorithm in order to find the most optimal labels order. By using the fitness function from Equation 3 for any individual, where EM is the Exact Match, ACC is the Accuracy and HL is the Hamming Loss, it quickly converges to a label order. The fitness of an individual p is given by Equation 3.

$$Fitness(p) = \frac{EM + ACC + (1 - HL)}{3} \tag{3}$$

It is important to notice that this algorithm will start with k random individuals, and after calculating the fitness for everyone, a winner will be selected among them, which will be the one with highest Fitness value. Afterwards, this individual will suffer random mutations in order to create the next generation, and the same steps above are repeated. After N generations, we will have an order for the labels which is a local maximum in relation to its fitness function.

4 Proposed Method

4.1 General Idea

In simple terms, our objective is to learn how to traverse a binary tree. What makes learning algorithms interesting in this problem is that some complex patterns can be present in the best path, and the search space, in general, is too big to be completely explored. Learning how to traverse a search space is a frequent problem in reinforcement learning.

4.2 Problem Definition

As stated earlier, our problem can be abstracted as finding the path with the highest product in a binary tree. To formalize this problem, from now on we refer to our search tree Γ as a set of nodes $n \in \Gamma$ connected only to its parent p_n and exactly two children c_1^n and c_2^n or exactly no children, a node with no children is called leaf node or terminal node. The root well be defined as r and is the only node with no parent. A path P is a sequence $n_0 n_1 ... n_f$ such that for every pair $n_i \in P$ and $n_j \in P$, if n_j comes strictly after n_i, then n_i is the parent of n_j. To each non terminal node n we associate a number p_n that is the probability to get to c_1^n and its value is $\hat{P}(y_i|y_{i-1}, ..., y_0, \mathbf{x})$, in the same way, we can associate a probability to c_2^n as $1 - p_n$. We finally define the value of a path as the product of the probabilities along that path from parent node to child node (we could also define it as the sum, but we'll focus on the product for now).

This way our problem can be formally defined as to maximize the value $V(P)$ $\forall P \in \Gamma$ such that the first node $n_0 = r$ and the last node n_t is a terminal node. Our model should learn to maximize

$V(P)$ for every tree generated by the classifier chain method in a given dataset.

4.3 Reinforcement learning

In reinforcement learning it's important to establish two important concepts: the environment and the agent. In general terms, there's a state space S which defines the agent's state, the agent acts given a state $s_t \in S$ with an action $a_t \in A$ where A is the action space that defines the possible actions the agent can take. Then, the environment provides as a response to the action a_t: the next state s_{t+1} and a reward $r_{t+1} \in R$ where R is the reward space. Figure 4 illustrates the reinforcement learning framework.

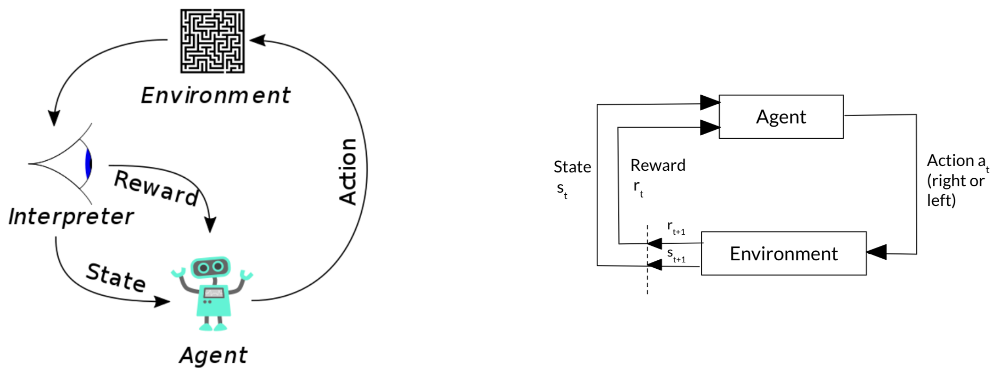

Figure 4: The image on the left [13] shows the general idea of training a reinforcement learning where the agent is rewarded for it's actions and the environment provides the state transitions given actions. The image on the left places the definitions above in a agent-environment chart.

4.4 The environment

We represented each node in Γ using a vector of size equal to two times the height of the tree minus one (or two times the number of labels in the chain). The first half of the vector contains the path from the root to the current node, which is represented as a sequence of values from the set $-1, 0, 1$, where -1 and 1 represent the direction it has taken (left or right, respectively), and 0 represents that it hasn't taken any direction yet. The second half of the vector is the probability of choosing left at that level of the tree.

For example, in Figure 3 the node with identified as 0.4 would have the state vector $[-1, 0, 0.4, 0]$, while the node 0.04 would be $[-1, 1, 0.4, 0.9]$.

Therefore, the first state of the environment is the root, and as the agent chooses actions it moves on the tree, receiving rewards. Those will be zero for all of the nodes except the ones that are leaves. The rewards for those are given by the value $V(P)$ defined before.

4.5 The agent

Given the state previously defined, which encodes the position that the agent is in the tree Γ, our agent should be able to choose the best action taking into consideration that he should end up with the best solution for our problem. Considering that our problem should benefit from a reinforcement learning method that is good at exploring trees, our agent should benefit from the ideas present in the AlphaGo Zero framework [16] explained briefly in Figure 5. However, to start and develop a proof of concept we implemented a reinforcement learning agent that explores a tree based on deep Q-Learning.

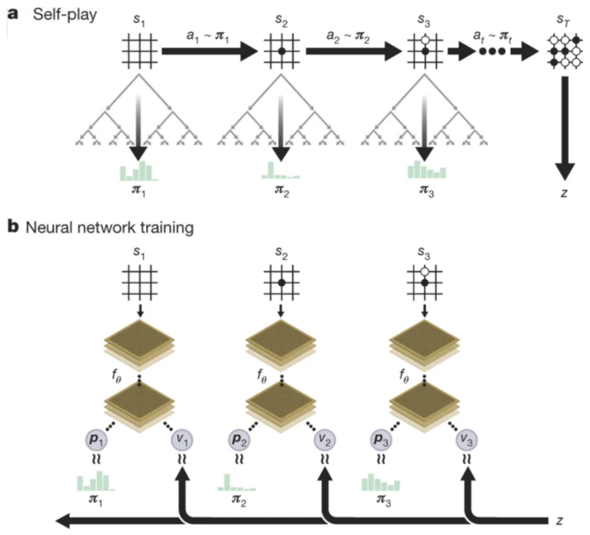

Figure 5: General idea of AlphaGo Zero, (a) shows an evaluation stage that uses Monte Carlo Tree Search and (b) shows the train stage which train a policy and a value networks.

4.5.1 Agent based on Deep Q-Learning

This was the first agent that we implemented and our main objective with it was to set up the code framework for our final agent based on AlphaGo Zero [16].

In this agent, we train a single neural network that takes as inputs a state s_t where t is the depth in the search tree, defined in the environment section, and an action $a_t \in \{-1, 1\}$ which defines the direction to take left -1 or right 1 and outputs the best final value that the network estimates the agent will achieve from s_t and a_t.

It's important now to define how the agent is trained. For the training, the agent first go down the tree several times following an ϵ-greedy, strategy. This means that with probability $\epsilon \in [0, 1]$ the agent takes a random action and with probability $1 - \epsilon$ the agent takes the action that the network estimates to have the best value. In the traversals, the final value achieved is recorded for each path and then, for each state, the best final value it has achieved in the roll-outs is recorded. This way, we have a set of pairs $\{\{s_1, z_1\}, \{s_2, z_2\}...\{s_n, z_n\}\}$ giving states and final values. We finally use these recorded states and final values to train our network. Figure 6 shows the architecture of our deep Q-learning network.

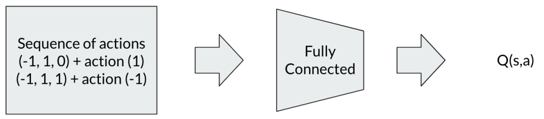

Figure 6: Architecture of our deep Q-Network.

4.5.2 Agent based on the AlphaGo Zero [16] framework

The base model

Our tree traversal is very similar to the one use used by AlphaGo Zero [16], we take the current state s_t, where t is the index in the current traversed path from the root node (therefore the number of nodes traversed). Then, using this state as base, it calculates two values, $v^\theta(s_t)$ and $p^\theta(s_t)$. This values represent, respectively, the value and the policy calculated by the network in the following way: $v^\theta(s_t) = f_{\theta_v}(s_t)$ and $p^\theta(s_t) = f_{\theta_p}(s_t)$. The layer $f_\theta(\cdot)$ is a fully connected layer with rectified linear unit activation. The values θ are the model parameters. Figure 7 shows the general outline of our network.

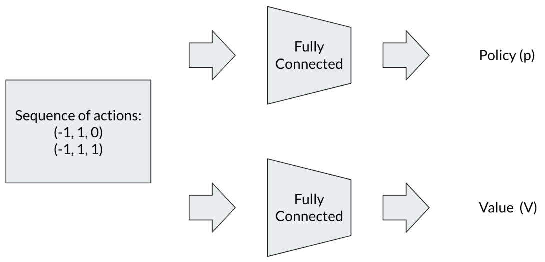

Figure 7: Architecture of the network used in our final approach.

The value $v^\theta(s_t)$ represents a prediction for the network of the maximum value that can be achieved from the current state s_t until a terminal node. The policy $p^\theta(s_t)$ represents an approximation of the probability that it should select each child for the next step to maximize the final $V(P)$, the value of the final path after the traversal.

Training the model

When training and evaluating the model, it doesn't sample an action (which represents choosing the child node to go next) based on the predicted policy. It uses MCTS [16] to calculate a new improved policy that we'll call $\pi(s_t)$. Finally, to train the model it runs simulated traversals from root to leaf on the tree and use the improved policy $\pi(s_t)$ to decide at each step which direction to go. In this simulation it records the values of the tuples $(s_t, \pi(s_t), z_t)$, respectively the value of the state, the improved policy and the final value achieved after a path is transversed completely.

The loss used in training is given by Equation 4.

$$l = \sum_t \left(\left(v^\theta(s_t) - z_t \right)^2 - \pi_t \log(p^\theta(s_t)) \right) \tag{4}$$

The model uses this loss with gradient descent to train our model. To understand this loss, let's divide it in two parts. The first one is the $\left(v^\theta(s_t) - z_t\right)^2$ term, which represents that the value predictions should be updated to predict accurately the final value (the value $v^\theta(s_t)$ is obtained directly from the network placing the current state in the network). The second term, $\pi_t \log(p^\theta(s_t))$ represents that the predicted policy should converge to the distribution of the improved policy calculated by MCTS (in the same way, $p^\theta(s_t)$ can be obtained from the network). Observe that we're recording the states and training the network based on the recorded states.

The Monte Carlo Tree Search

An important part of the described model is the improved policy, it's what is used to make actual path decisions and also to define the loss for the predicted policy. We proceed, again, by using the same idea presented in the AlphaGo Zero [15] paper. For each state s we define the state-action value function $Q(s, a)$, which defines the benefit from taking action a (equivalent to choose a child to go along the path) at state s (which defines the current position on the tree traversal). During the simulation to calculate the improved policy $\pi(s)$ of a given state, we expand a tree with s as root and choose child nodes that maximize the state-action value plus a bonus $u(s, a)$ defined in Equation 5, this way we're discouraged to explore nodes that already have been explored.

$$u(s, a) = cte \frac{P(s, a)}{1 + N(s, a)} \tag{5}$$

At each state we choose a_t as $argmax(Q(s_t, a) + u(s_t, a))$ for all of the possible actions at s_t. $P(s, a)$ defines the policy calculated by the network and can be directly derived from $p^{\theta}(s)$. We initialize the action values Q and the counts N as zero and as we traverse the tree, we increase the count for each pair state action that is traversed. We also backpropagate the values calculated by the value network so that $Q(s, a)$ gets updated with the average between all of the states that it reaches during the simulation.

Finally, once some paths were simulated, the new improved policy can be calculated using the counts for each action that has been taken from the current state s. The formal definition of the new policy is presented on Equation 6.

$$\pi(s) = \frac{N(s, \cdot)}{\sum_i N(s, i)} \tag{6}$$

5 Results

We have implemented the methods presented until now, as well as the baseline methods introduced in Section 3.1. In this section we'll show a comparison of their performances in a few datasets often used in multi-label classification studies.

We created a Gym [6] environment to make the interface between the datasets and the two agents, which were implemented using Pytorch [8]. Figure 8 contains the visualization of the environment, which we implemented using Pygame [7], for debugging purposes. This tool was very useful for testing our algorithms, and see clearly which decisions they were taking at each level of the tree.

Table 1 contains the properties of the datasets we used in our analysis. The classifier chains orders were optimized using the genetic method described in Section 3.2. Moreover, the estimators are logistic regressions, which had their parameter C optimized between the values 10^{-3}, 10^{-2}, ..., 10^{3}, as was detailed in [4].

Table 1: Properties of the tested datasets.

Dataset	Type	Train size	Test size	Attributes	Labels	Cardinality
emotions	Music	391	202	72	6	1.87
flags	Image	129	65	19	7	3.39
image	Image	1800	200	135	5	1.24

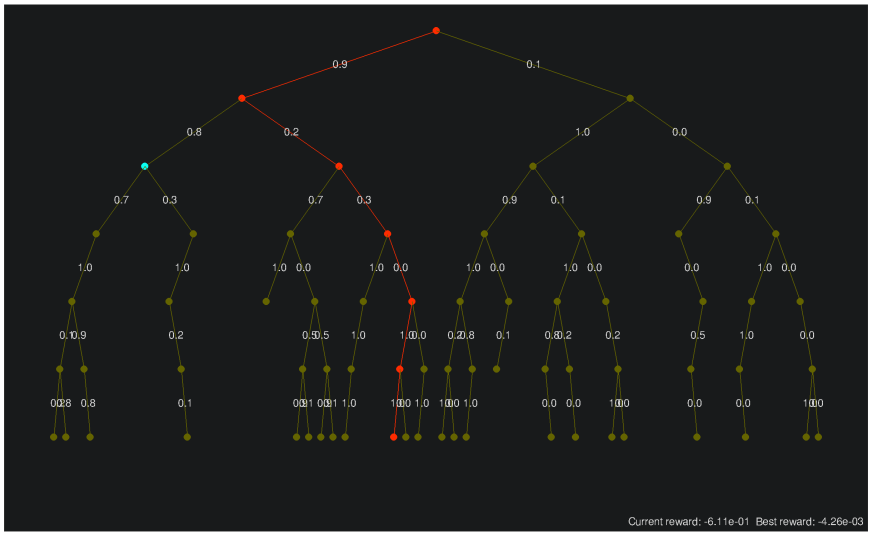

Figure 8: Visualization of the MCTS agent being trained in a sample of the "emotions" dataset. Each edge in the tree contains the probability given by the estimators in the chain. The red path contains the best path until now, and the blue node is where the algorithm is currently at.

A more thorough analysis was done in the "emotions" dataset. Figure 9 was generated in order to study the effect of the number of explored nodes in our proposed methods. The Exhaustive Search and Greedy Search algorithms are both present in the plot as horizontal lines, since they have a constant number of explored nodes (63 and 6, respectively, for this dataset). We also created a simple method inspired on Efficient Monte Carlo which, instead of throwing a biased coin to decide which path to take on the tree, it throws an unbiased one. This method in presented in Figure 9 as "Random".

The DQL agent's data was collected on each of 100 exploration-train steps, in which each exploration step collected data from 5 different paths, and 10 epochs were performed in each train step. The MCTS agent's data was collected on each of 100 exploration-train steps as well, in which each exploration step collected data from 1 path, which was found using 3 passes in each tree depth, and 10 epochs were performed in each train step.

We can conclude by this plot that our methods are better than the randomized approach. They are able to constantly achieve better rewards with the same number of nodes. This is a very important result, because it shows that both methods are able to successfully learn, without prior information, how to choose appropriate paths to achieve good rewards.

However, they still find worse paths than those from the baseline methods. To surpass the greedy reward, they both need around 1500 explored nodes, which is a very high number since each tree only has 63 distinct ones.

Tables 2, 3 and 4 contain the results of our experiments on other datasets. The following abbreviations were used: GS (greedy search), ES (exhaustive search), ϵ-A (ϵ-Approximation with $\epsilon = 0.25$), BE (Beam search with $b = 3$), MC (Monte Carlo on 100 paths), EMC (Efficient Monte Carlo on 100 paths), DQL (our Deep Q-Learning agent with 100 train steps, each with 3 different paths and 10 epochs) and MCTS (our Monte Carlo Tree Search method with 60 train steps, each with 1 path, 3 passes in each tree depth, and 20 epochs).

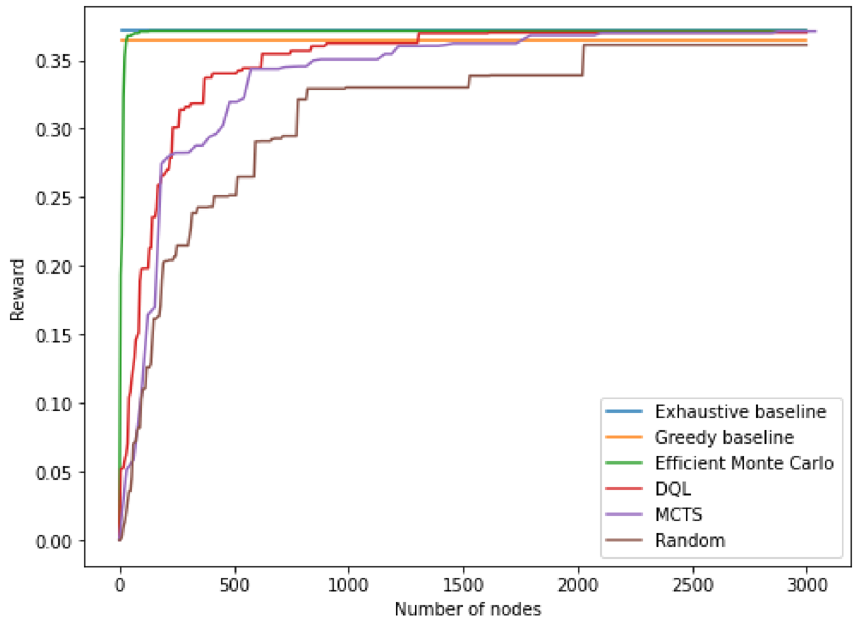

Figure 9: Comparison of baseline methods and our methods on the "emotions" dataset, with an increasing number of explored tree nodes. The plot shows the average rewards on the test samples.

The reward used when minimizing the 0/1 loss is given by $\prod_i \hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$, as described in Section 3.1 and [10]. To minimize Hamming loss, the reward is given by $\sum_i \hat{P}(y_i|y_{i-1},...,y_0,\mathbf{x})$.

Table 2: Average reward and subset 0/1 loss for tested datasets, multiplied by 100. The best result(s) in each row is in bold.

Dataset	Metric	GS	ES	ϵ-A	BE	MC	EMC	DQL	MCTS
emotions	Reward ↑	36.39	**37.13**	37.05	**37.13**	36.84	**37.13**	37.06	36.12
	0/1 Loss ↓	**65.35**	66.83	66.34	66.83	69.31	66.83	67.33	69.31
flags	Reward ↑	4.10	**5.92**	5.34	5.87	4.81	**5.92**	5.90	5.69
	0/1 Loss ↓	**73.85**	95.38	87.69	95.38	89.23	95.38	95.38	96.92
image	Reward ↑	24.88	**26.93**	26.89	26.92	25.97	**26.93**	**26.93**	**26.93**
	0/1 Loss ↓	71.50	**63.00**	63.50	**63.00**	**63.00**	**63.00**	**63.00**	**63.00**

Table 3: Average reward and hamming loss for tested datasets, multiplied by 100. The best result(s) in each row is in bold.

Dataset	Metric	GS	ES	ϵ-A	BE	MC	EMC	DQL	MCTS
emotions	Reward ↑	505.0	**511.1**	**511.1**	510.9	507.9	**511.1**	509.8	508.1
	Hamming loss ↓	**18.89**	21.45	21.45	21.20	20.79	21.45	21.04	21.53
flags	Reward ↑	450.0	**485.1**	**485.1**	484.8	469.7	**485.1**	484.8	480.9
	Hamming loss ↓	**23.08**	37.80	37.80	37.58	38.68	37.80	35.82	33.63
image	Reward ↑	379.7	**392.6**	**392.6**	**392.6**	387.3	**392.6**	391.7	391.7
	Hamming loss ↓	**20.20**	24.20	24.20	24.20	22.30	24.20	23.40	23.40

Table 4: Average number of explored nodes and average inference time in milliseconds for tested datasets.

Dataset	Metric	GS	ES	ϵ-A	BE	MC	EMC	DQL	MCTS
emotions	Nb of nodes ↓	6	63	63	15	600	600	1800	1836
	Time ↓	0.025	0.065	0.066	0.025	0.873	0.676	25233	3571
flags	Nb of nodes ↓	7	127	127	18	700	700	2100	2229
	Time ↓	0.019	0.169	0.152	0.040	1.135	1.041	29886	4159
image	Nb of nodes ↓	5	31	31	12	500	500	1500	1441
	Time ↓	0.027	0.039	0.042	0.022	0.835	0.611	21533	3077

As explained in [4], some of the loss results are inconsistent with what is theoretically expected. This happens because the estimators are not perfect, and probabilities predicted by them are not the true probabilities. We added the rewards information to confirm the correctness of the algorithms and improve the comparisons, since they are less dependent on the chain's estimators.

The first result that stands out is the time taken for the DQL and MCTS calculations. On our implementation we didn't focus on optimizing their processing time, and therefore they were expected to take much longer in comparison with the other algorithms, which were all implemented using vectorization techniques in Python. For this reason we weren't able to test our methods in larger datasets, which could possibly give us better results, since the baselines are already very good and extremely fast in small ones.

The number of nodes is a more important measurement and, as discussed in Figure 9, the methods need a lot of exploration to achieve good results. Therefore we decided to make them explore approximately three times as the Efficient Monte Carlo for these comparisons.

With this amount of exploration the methods are able to achieve competitive results, staying between the baseline methods. Both achieved the best result on the 0/1 Loss on the "image" dataset as well. Overall the DQL is better than the MCTS, but it is also 7-8 times slower.

6 Conclusion

We implemented two agents based on Deep Q-Learning and on the AlphaGo Zero framework, which were able to successfully learn how to explore the trees induced by our classifier chain without prior knowledge. They are superior than a random explorer, and with sufficient exploration they also achieve equal or better performances than the baseline methods.

However, they require a lot of exploration to reach these performances. Thus, a way to reduce this requirement would be very valid. In order to tackle this problem, possible extensions to this approach could explore transfer learning ideas, since our method trains a different agent for each new dataset sample. Simply extending our approach to training a single agent for the whole dataset isn't ideal, since two different samples of the same dataset can have the same state, but different policies for the neural networks to learn. Finally, creative solutions should be applied, like using an agent for each tree level, or other transfer learning related ideas.

An RNN could also be used to better encode the states. This could improve the model's learning capabilities, but it could also increase the processing time, which is already high.

References

[1] Krzysztof Dembczyński, Willem Waegeman, and Eyke Hüllermeier. "An Analysis of Chaining in Multi-Label Classification". In: *Proceedings of the 20th European Conference on Artificial Intelligence*. ECAI'12. Montpellier, France: IOS Press, 2012, pp. 294–299. ISBN: 9781614990970.

[2] E. C. Goncalves, A. Plastino, and A. A. Freitas. "A genetic algorithm for optimizing the label ordering in multi-label classifier chains". In: *IEEE 25th International Conference on Tools with Artificial Intelligence* (2013), pp. 469–476. URL: `https://www.cs.waikato.ac.nz/~eibe/pubs/chains.pdf`.

[3] Abhishek Kumar et al. "Beam search algorithms for multilabel learning". In: *Machine Learning* 92.1 (July 2013), pp. 65–89. ISSN: 1573-0565. DOI: `10.1007/s10994-013-5371-6`. URL: `https://doi.org/10.1007/s10994-013-5371-6`.

[4] Deiner Mena et al. "An overview of inference methods in probabilistic classifier chains for multilabel classification". In: *WIREs Data Mining and Knowledge Discovery* 6.6 (2016), pp. 215–230. DOI: `https://doi.org/10.1002/widm.1185`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1185`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1185`.

[5] Jinseok Nam et al. "Maximizing Subset Accuracy with Recurrent Neural Networks in Multi-label Classification". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 5413–5423. URL: `https://proceedings.neurips.cc/paper/2017/file/2eb5657d37f474e4c4cf01e4882b8962-Paper.pdf`.

[6] OpenAI. *Gym: A toolkit for developing and comparing reinforcement learning algorithms*. URL: `https://gym.openai.com` (visited on 03/22/2021).

[7] *Pygame*. URL: `https://www.pygame.org/news` (visited on 03/22/2021).

[8] *PyTorch*. URL: `https://www.pytorch.org` (visited on 03/22/2021).

[9] J. Read et al. "Classifier chains for multi-label classification". In: *Machine Learning* 85 (3) (2011). URL: `https://www.cs.waikato.ac.nz/~eibe/pubs/chains.pdf`.

[10] Jesse Read, Luca Martino, and David Luengo. "Efficient Monte Carlo Optimization for Multi-label Classifier Chains". In: *CoRR* abs/1211.2190 (2012). arXiv: `1211.2190`. URL: `http://arxiv.org/abs/1211.2190`.

[11] Jesse Read et al. "Classifier Chains for Multi-label Classification". In: vol. 85. Aug. 2009, pp. 254–269. ISBN: 978-3-642-04173-0. DOI: `10.1007/978-3-642-04174-7_17`.

[12] Jesse Read et al. *Classifier Chains: A Review and Perspectives*. 2020. arXiv: `1912.13405 [cs.LG]`.

[13] *Reinforcement learning*. In: *Wikipedia*. Page Version ID: 1013304333. Mar. 20, 2021. URL: `https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1013304333` (visited on 03/22/2021).

[14] Yelong Shen et al. *M-Walk: Learning to Walk over Graphs using Monte Carlo Tree Search*. 2018. arXiv: `1802.04394 [cs.AI]`.

[15] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017). Number: 7676 Publisher: Nature Publishing Group, pp. 354–359. ISSN: 1476-4687. DOI: `10.1038/nature24270`. URL: `https://www.nature.com/articles/nature24270` (visited on 12/13/2020).

[16] Tom Vodopivec, Spyridon Samothrakis, and Branko Ster. "On Monte Carlo Tree Search and Reinforcement Learning". In: *Journal of Artificial Intelligence Research* 60 (Dec. 20, 2017), pp. 881–936. ISSN: 1076-9757. DOI: `10.1613/jair.5507`. URL: `https://jair.org/index.php/jair/article/view/11099` (visited on 12/13/2020).