Solving Pong with DeepProblog

Ravoet Yannou
Computer Science
KU Leuven
Belgium, Leuven
yannou.ravoet@student.kuleuven.be

***Abstract*—This report discusses the implementation of a Pong agent using DeepProblog. We compare a DeepProblog trained agent where the perception and decision part of the agent are split, to the classical end-to-end approach. Specifically, we highlight the fast convergence and efficient use of data by DeepProblog as well as the possibilities to adapt the decision logic of the agent, without having to retrain all neural networks. Finally, we experiment on the agents robustness to noise in the game screen to simulate the influence particle effects would have on the agent.**

***Index Terms*—DeepProblog, CNN, FOL**

I. Introduction

This report discusses the implementation of a DeepProblog agent capable of playing Pong. The first section discusses the literature study that inspired the implementation. Next, a brief overview of the structure of the program is given. The third section discusses the idea behind and the execution of several experiments, meant to show the advantages of using DeepProblog over classical end-to-end training. We compare the speed at which accuracy converges, the final accuracy as well as how much data is needed to train the agent. We implement two DeepProblog agents with overlapping perception capabilities yet different decision making logic. We also look at the robustness of the agents to noise. Finally, the conclusion summarizes the report and mentions the time spent on each part of the project.

II. Literature Study

Since this project is focused on the implementation of a DeepProblog program, the literature study mainly revolved around studying the original DeepProblog paper [1]. This paper was the inspiration to comparing the training process of an agent in DeepProblog and an agent trained end-to-end in PyTorch. The MNIST examples provided by the DeepProblog repository [2] were highly helpful in the implementation of both the DeepProblog and PyTorch models.

III. Software Architecture

This section provides a brief overview of how the project software is structured. The implemented software has three main functionalities:

- generating data to train the agents,
- training the agents and
- running a game of Pong between two players.

Fig. 1. A game state with coordinates for the left-paddle (0,8), the ball (4,7) and the right-paddle (29,9).

A. Pong Game

The game of Pong consists of two players bouncing a ball back-and-forth. The goal of each player is to bounce the ball past the opponent. The players are modelled as 1x3 (x=1, y=3) rectangles that can move up or down along the y-axis but are fixed to a side of the screen along the x-axis. The ball is a 1x1 square that bounces against the top edges of the map and against the players. The map itself is a 30x16 pixels rectangle. In total, there are 87.808 possible game states, one of which is shown in Figure 1.

Both players and the ball are modelled as GameObjects with a position, speed and direction on the x- and y-axes. A Game object stores these GameObjects as well as game related statistics such as the score of a match. Each frame, the Game updates all GameObjects to determine their new positions and redraws the updated game screen.

B. Functionalities

There are three main functionalities of the project. Generating data consists of creating images representing a random legal state of the game as well as the corresponding meta-data containing the positions of the agents and the ball. To train the agents, the data is later sorted based on the target class. For example, when training a network where the loss is calculated based on the action that was chosen, the data will be sorted in the directory structure shown in Figure 2. For a classical end-to-end agent, this would be because the neural network directly outputs one of the possible actions. For a

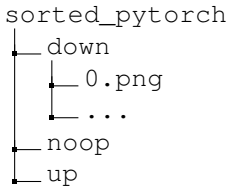

Fig. 2. Example of how a dataset will be sorted to ease the training of networks with different outputs. Datasets are loaded as a torchvision.datasets.ImageFolder.

DeepProblog agent, this is because the logic used to train the network returns an action, not necessarily the network itself.

The training process allows the user to choose a train- and test-dataset as well as how often to log the training loss and testing accuracy whilst training a player. After training, a model is saved for later use. Both a DeepProblog and a PyTorch training process are available.

Finally, running a game matches two opponents against each other and tracks the wins of each opponent. The user can change the speed of the ball, the speed of the players and the amount of noise in the game screen. These parameters will be used in the experiments to highlight the strengths and weaknesses of different agents.

C. Modelling a DeepProblog Agent

The AIPlayer class models an agent trained with DeepProblog. It consists of two parts: a neural network for perception and a logic program for decision logic.

The perception part consists of a Convolutional Neural Network, structured as shown in Figure 3. The CNN takes as input a 3 channel 30x16 image of the game screen passed by the Game object. Depending on the type of information we want to perceive the outputs of the network change. For a first version of a Pong agent, we use a network that outputs a probability distribution over the possible positions of the ball along the y-axis. As such, the output layer of Figure 3 shows 16 nodes.

The logic part of the agent consists of a First-Order Logic (FOL) program that uses the output of the CNN to determine the best course of action. This logic can be changed independently from the perception part.

IV. Experiments

This section discusses the experiments that were run during the project. We first briefly overview the datasets that were used. Next, we compare the training process of an end-to-end CNN to a DeepProblog trained agent. Thirdly, we try to improve the agents decision making to make it more robust to faster game speeds. Finally, we measure the performance of the agent with respect to the amount of noise in the game screen.

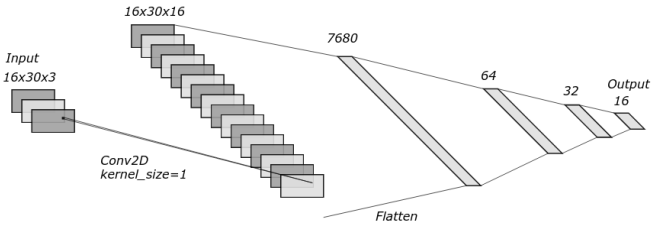

Fig. 3. Structure of the CNN used to interpret the game screen.

A. Generated Data

The data used to train the agents neural networks consists of randomly generated valid game states. The data is shuffled and split into training and testing data set with an 85-15% distribution. The split is stratified on the target directories (as shown in Figure 2). During the first experiment, we generated five datasets of different sizes: 1000, 2500, 5000, 10.000 and 20.000 game states.

B. Training in DeepProblog vs PyTorch

The goal of our first experiment is to provide a measurement of how quickly agent training converges as well as how much data is needed to train an agent to a high accuracy. We compare a DeepProblog agent against a classically trained end-to-end CNN. The DeepProblog agent uses First-Order Logic as an extra layer of feedback during training. Specifically, we introduce a *distance_y(Img, AIY, D)* term that measures the distance, *D*, between the center of the agents paddle, *AI*, and the ball along the y-axis as shown in Figure 5. In the classically trained CNN, the decision logic is part of the network. The only difference in the networks used by each model, is thus the number of outputs. Since the DeepProblog model uses FOL to determine the action of the player, our first perception network returns the balls y-positioning. For the PyTorch model, the output of the network directly corresponds to the action of the player, either up, down or no action. For both models, we measure the training loss and test accuracy over training iterations. We compare these measurements on five datasets of varying sizes.

Figure 4 shows the results of our first experiment. Since the PyTorch network only has three outputs, its initial accuracy is much higher. However, we can see that for datasets with 2500 images or more, the DeepProblog model quickly converges to a final near perfect accuracy. The PyTorch model needs slightly more training iterations to converge to a lower final accuracy. With a dataset larger than 5000 images, the DeepProblog model can achieve a 100% accuracy on the test set. However, even if we increase the size of the datasets fourfold, the PyTorch model still converges to a slightly lower final accuracy of roughly 97%. This shows us the first advantages of using FOL feedback during training: *slightly less iterations*

of training are needed to achieve a higher accuracy. We also need much less data to achieve this accuracy.

C. Improving the agent logic

However, the biggest advantage of the DeepProblog approach could be that we can change the decision logic of the agent without having to retrain the CNN as long as we use the same percepts.

The first version of the AI agent is shown in Figure 5. The model simply checks whether the player's y-axis positioning is higher than, lower than or equal to the ball's y-axis positioning predicted by the network. Based on this comparison, the AIPlayer respectively moves up, down or not at all. This decision making is entirely reactive to the current state of the game and does no planning based on the rules of the game or any previous states of the game. As a result, it works well for a game were the players speed is at least as high as the ball speed and both speeds are low enough such that initial differences can be compensated. For a speed of 1 pixel per frame, it even achieves a 100% win rate against a random player. However, when we increase the speed and especially when we make the ball faster than the players, the agents win rate falls all the way to around 50%. The exact win rates over 1000 rounds for variable ball and player speeds are shown in Table I.

To handle higher speeds and especially a ball that is faster than the agent, the agent must be able to predict where the ball will be. The second version of the AIPlayer is therefore no longer simply a reactive player. To predict where the ball will be when it reaches the agents side of the screen, we need to extend the system with two parts. Firstly, the agents needs to be able to measure the x- and y-coordinates of the ball in the current and the previous frame. Secondly, the decision making logic needs to include the game logic that determines how the ball bounces against the side of the screen.

As stated before, the perception of the y-coordinate of the ball does not need to be retrained, we can simply reuse the same network. For the x-coordinate, we define a new predicate *distance_x(Img, D)* that returns the distance, *D*, between the ball coordinate retrieved from the image and the player coordinate (fixed at 29) along the x-axis. We use the same network architecture as before, with the sole difference that the outputs are now the x-coordinates (there are thus 30 outputs).

For the decision logic, we first check whether the ball is traveling towards or away from the player. If it is travelling away, we simply go back to the center of the y-axis. Otherwise, we calculate the y-coordinate the ball will have when it reaches the x-position of the agent. We do this by calculating the speed of the ball and the distance to the player. We can then find the number of frames between the current frame and the frame where the agent will have to intercept the ball. We then simply simulate the movement of the ball by continuing in the direction it is going and taking into account that it bounces against the top and bottom edges of the screen. The logic for this is shown in Figure 6.

TABLE I
WIN RATES OF THE FIRST DEEPPROBLOG AGENT VERSION AGAINST A RANDOM PLAYER WITH VARYING BALL AND PLAYER SPEEDS.

Player	Ball	win rate	Player	Ball	win rate
1	1	100%	1	2	46.6%
2	2	63.5%	1	3	48.5%
3	3	62.3%	2	3	47.2%
4	4	55.5%	2	4	56.4%
			3	4	51.6%

TABLE II
WIN RATES OF THE SECOND DEEPPROBLOG AGENT VERSION AGAINST A RANDOM PLAYER WITH VARYING BALL AND PLAYER SPEEDS.

Player	Ball	win rate	Player	Ball	win rate
1	1	100%	1	3	95.9%
2	2	100%	1	4	73.7%
3	3	79.8%	2	3	79.8%
4	4	77.9%	2	4	69.7%
1	2	100%	3	4	65.6%

The results of playing this second agents against a random agent over 1000 rounds are shown in Table II. We can see that the agent now handles higher speeds better and balls that are faster than the agent much better since it can predict the future position of the ball and is thus no longer completely reactive to the current state of the game.

D. Testing agent robustness

We perform a final experiment to test how robust the second version agent is against noise. Although noise in the game screen due to challenges like packet loss are becoming less common, particle effects have become more and more common in games. Since particle effects usually don't influence gameplay logic, they can be considered as a form of noise. In our experiment, we add a percentage of *x%* noise to the game screen by choosing *x%* of the pixels on screen and giving them a random color. An example of a game screen with varying levels of noise is shown in Figure 7.

We measure the win rates of the second version agent with a player speed of 1 and a ball speed of 2 pixels per frame for 0%, 1%, 2.5% and 5% noise in the game screen over 1000 games. The results are shown in Table III. We can clearly see the accuracy drop as a function of the noise, but with low to intermediate levels of noise, we can still retain a decent enough accuracy to be challenging to play against. We can see in Figure 7 that 5% noise is already a large amount that is unlikely to happen often as the game becomes too chaotic.

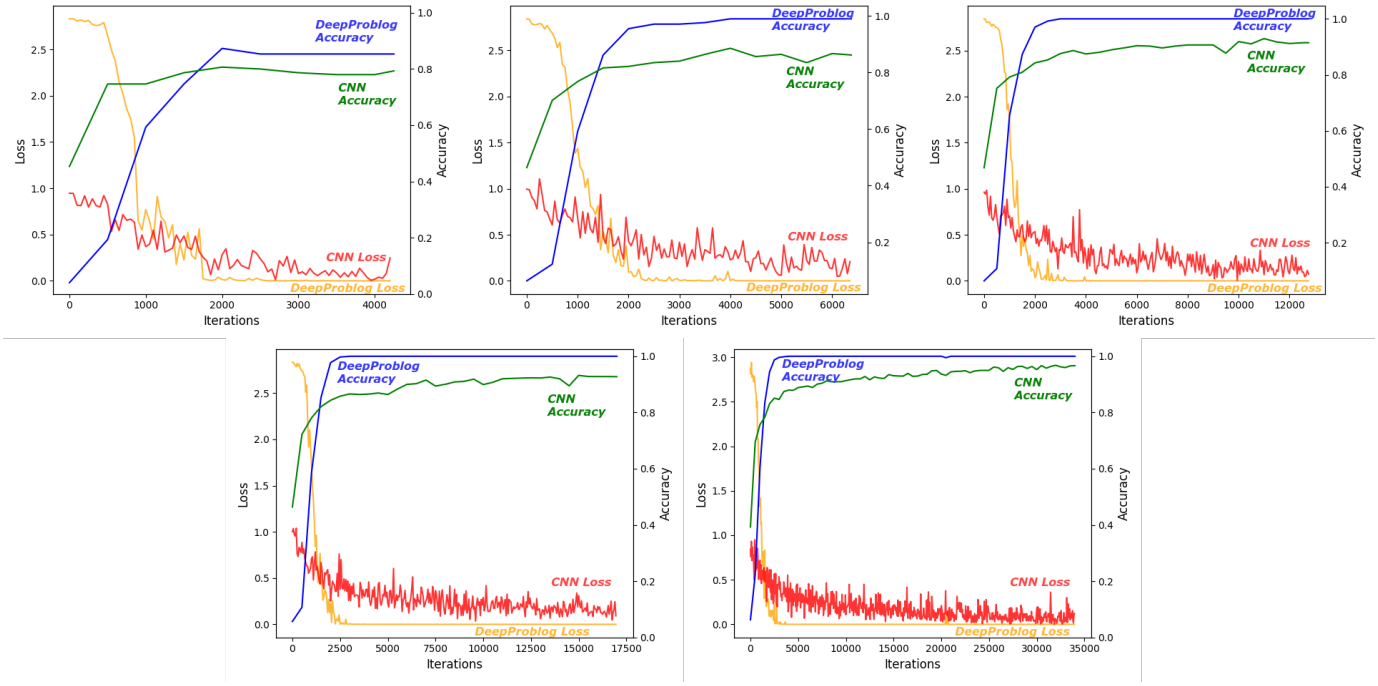

Fig. 4. Training loss and accuracy for agent models trained end-to-end with PyTorch compared to agent models trained with DeepProblog and FOL feedback. Each plot is trained on a different size dataset: 1000 (top-left), 2500 (top-mid), 5000 (top-right), 10.000 (bot-left) and 20.000 (bot-right) images.

```
nn(pong_net_y,[Img],BallY,[0,...,15]) :: y_coord(Img,BallY).

distance_y(Img, AIY, D):- y_coord(Img, BallY), D is AIY - BallY.

choose_action(Img, AIY, noop) :- distance_y(Img, AIY, D), D = 0.
choose_action(Img, AIY, up) :- distance_y(Img, AIY, D), D > 0.
choose_action(Img, AIY, down) :- distance_y(Img, AIY, D), D < 0.
```

Fig. 5. First-Order Logic used to model the first version of an AIPlayer.

TABLE III
WIN RATES OF THE SECOND DEEPPROBLOG AGENT VERSION AGAINST A RANDOM PLAYER WITH VARYING NOISE LEVELS.

Noise	win rate	Noise	winrate
0%	100%	2.5%	85.2%
1%	89.2%	5%	62.5%

V. CONCLUSION

We showed the implementation of a Pong agent implemented using the DeepProblog framework where a Convolutional Neural Network was used for the agents perception. Using DeepProblog allowed us to change the decision logic written in First-Order Logic without needing to retrain the whole perception part of the agent. We compared the speed of convergence and the efficiency of data usage between training an agent with DeepProblog and the classical end-to-end approach. We noted that the DeepProblog agent converges faster and to a higher accuracy. The DeepProblog agent also needed much less training data to achieve this accuracy. We successfully extended both the perception and decision making logic of our first version of the agent to be able to better handle variable ball and player speeds. Finally, we tested the robustness of our agent to noise by setting it up in games with varying levels of noise. We showed that for low to intermediate levels of noise, the agent still retains enough accuracy to be challenging.

As this was mostly an implementation project, the literature study was limited to around 3-4 hours of reading the original DeepProblog paper and researching ideas for the project. The initial software architecture: the design and debugging of the game and data generation logic took roughly 35 hours, where a lot of time was spent getting to know the DeepProblog framework and understanding the examples. Running the experiments required me to generate datasets of varying size and retrain the agents on each dataset as well as running a large number of games to have stable winrates. Around 10 hours was also spent on testing different CNN architectures for the agents. The experimentation side of the project thus took the largest part of the time spent and is estimated to have taken around 50-60 hours. Finally, writing the report took around 15-20 hours. The total time spent on the project was roughly 14 days of work throughout the year or around 110 hours.

REFERENCES

[1] R. Manhaeve, S. Dumančić, A. Kimmig, T. Demeester, and L. De Raedt, "DeepProbLog: Neural Probabilistic Logic Programming," arXiv:1805.10872 [cs], Dec. 2018.

[2] ProbLog team, DeepProblog, (2020), Bitbucket repository, https://bitbucket.org/problog/deepproblog.

```
nn(pong_net_y,[Img],BallY,[0,...,15]) :: y_coord(Img,BallY).
nn(pong_net_x,[Img],BallX,[1,...,28]) :: x_coord(Img,BallX).

%used to train the pong_net_x network
distance_x(Img, D) :- x_coord(Img, X), D is 29 - X.
coordinates(Img, X, Y):- x_coord(Img,X), y_coord(Img, Y).

choose_action(Img0, Img1, AIY, Action):-
    coordinates(Img0, X0, Y0),
    coordinates(Img1, X1, Y1),
    choose_action(AIY, X0, Y0, X1, Y1, Action).

%going away from agent
choose_action(AIY, X0, _, X1, _, noop):-
    direction_x(BallPrevX, BallCurX, -1),
    AIY = 8.
choose_action(AIY, X0, _, X1, _, up):-
    direction_x(BallPrevX, BallCurX, -1),
    AIY > 8.
choose_action(AIY, X0, _, X1, _, down):-
    direction_x(BallPrevX, BallCurX, -1),
    AIY < 8.

%going towards agent
choose_action(AIY, X0, Y0, X1, Y1, noop):-
    direction_x(X0, X1, 1),
    intersect_y(X0, Y0, X1, Y1, YFuture),
    AIY = YFuture.
choose_action(AIY, X0, Y0, X1, Y1, up):-
    direction_x(X0, X1, 1),
    intersect_y(X0, Y0, X1, Y1, YFuture),
    AIY > YFuture.
choose_action(AIY, X0, Y0, X1, Y1, down):-
    direction_x(X0, X1, 1),
    intersect_y(X0, Y0, X1, Y1, YFuture),
    AIY < YFuture.

%to be safe: if x doesn't change (wrong perception)
% => assume towards agent
direction_x(X0, X1, 1):- PrevX =< CurX.
direction_x(X0, X1, -1):- PrevX > CurX.
```

```
speed(Prev, Cur, Speed):-
    (Cur > Prev; Cur < Prev),
    Speed is Cur - Prev.

intersect_y(X0, Y0, X1, Y1, YFuture):-
    speed(X0, X1, SpeedX),
    speed(Y0, Y1, SpeedY),
    DistX is 29 - X1,
    FramesLeft is DistX / SpeedX,
    intersect_y_iter(SpeedY, FramesLeft, Y1, YFuture).

intersect_y_iter(_, FramesLeft, Y, Y):-
    FramesLeft =< 0.
intersect_y_iter(SpeedY, FramesLeft, TempY, Y):-
    FramesLeft > 0,
    NewFramesLeft is FramesLeft - 1,
    bounded_y_calc(TempY, SpeedY, NewTempY, NewSpeedY),
    intersect_y_iter(NewSpeedY, NewFramesLeft, NewTempY, Y).

%calculates the next y position and updates yspeed if there was a bounce
bounded_y_calc(CurY, SpeedY, NewY, NewSpeedY):-
    Steps is abs(SpeedY),
    DirY is sign(SpeedY),
    bounded_y_calc_iter(CurY, DirY, NewY, NewDirY, Steps),
    NewSpeedY is Steps * NewDirY.

bounded_y_calc_iter(CurY, DirY, CurY, DirY, 0).
bounded_y_calc_iter(CurY, DirY, FinalY, FinalDirY, Steps):-
    Steps > 0,
    (CurY = 0; CurY = 15), %if next step would hit top or bottom of screen
    NewDirY is -1*DirY,
    TargetY is CurY + NewDirY,
    NewSteps is Steps - 1,
    bounded_y_calc_iter(TargetY, NewDirY, FinalY, FinalDirY, NewSteps).
bounded_y_calc_iter(CurY, DirY, FinalY, FinalDirY, Steps):-
    Steps > 0,
    CurY > 0,
    CurY < 15,
    TargetY is CurY + DirY,
    NewSteps is Steps - 1,
    bounded_y_calc_iter(TargetY, DirY, FinalY, FinalDirY, NewSteps).
```

Fig. 6. First-Order Logic used to model the second version of an AIPlayer.

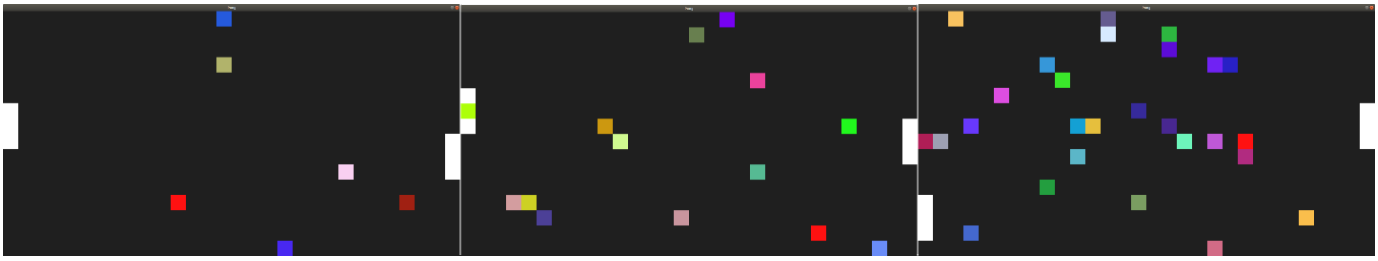

Fig. 7. Varying levels of noise in the game screen from left to right: 1%, 2.5%, 5%.