Connect4 Using Reinforcement Learning Deployed on Google Coral Development Board

Jonah O'Brien Weiss
Dept. of Electrical and Computer Engineering
University of Massachusetts Amherst
Amherst, MA
jobrienweiss@umass.edu

Abhinaba Das
Dept. of Electrical and Computer Engineering
University of Massachusetts Amherst
Amherst, MA
abhinabadas@umass.edu

Subhankar Chowdhury
Dept. of Electrical and Computer Engineering
University of Massachusetts Amherst
Amherst, MA
subhankarcho@umass.edu

***Abstract*— This paper provides an overview of our project to develop a ML model using Reinforcement Learning to play the game of Connect4 on the Google Coral. The model learns iteratively about the best possible move to make in response to a players move and beats the player irrespective of whether it makes the first move or not.**

Keywords—reinforcement learning, connect four

I. Introduction

Connect4 is a game where two players take alternate turns at dropping colored discs into a vertical grid. Each player uses a different color (usually red or blue), and the objective of the game is to be the first player to get four discs in a row.

Training an agent to learn to play board games is difficult due to the large state space of possible game states. Deep Reinforcement learning is one method to address the problem. It is the process via which an agent learns to maximize the reward for its actions taken over a period of time while interacting with its environment. Rather than encoding an action for each possible board state, a neural network learns from a representative set of board states and extrapolates or generalizes to unseen states.

Monte Carlo Tree Search(MCTS) provides a way to search a tree while balancing exploration and exploitation. Since some sequential board games like Connect4 can be represented as a tree, MCTS works well to find game states where a model trained on these states generalizes well.

The combination of MCTS and Reinforcement Learning has been shown to be an effective method to find optimal strategies in board games such as Connect4 and Go. AlphaGo is the first computer program to defeat a professional human Go player, the first to defeat a Go world champion, and is arguably the strongest Go player in history [1].

To deploy these algorithms, there have been several edge devices designed to handle machine learning intensive workloads such as the Google Coral. The Coral is a development board centered around Google's TPU (Tensor Processing Unit) which speeds up ML computations substantially as compared to a traditional CPU. Edge devices such as these allow us to deploy ML models in a at the edge in a cost effective manner while consuming low power as well.

II. Problem Statement

Our problem statement consists of the following:

- Training a Reinforcement Learning agent to play Connect4
- Deploying the model and the application on the Google Coral
- Constructing a Graphical User Interface to enable a more user-friendly version of the game

III. Edge Device: Google Coral Dev Board

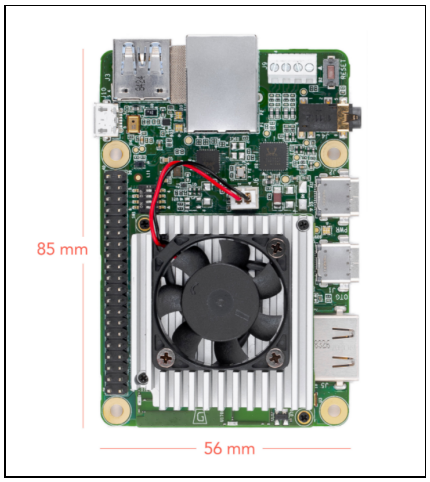

Fig. 1: Google Coral development board.

Google Coral is a single board computer that has been designed for fast on-board machine learning (ML) applications and embedded system design. It features a removable system-on-module (SOM) featuring the Edge TPU coprocessor, a small ASIC designed by Google that provides high performance ML inferencing with a low power cost [8]. It is a full computer, with a CPU, GPU and memory with Linux OS. It features a removable system-on-module (SOM) that contains eMMC, SOC, wireless radios and Coral Edge TPU on board.

The Coral Dev Board is ideal for prototyping internet-of-things (IOT) devices and other embedded systems that demand fast on-device ML inferencing. The TPU is capable of performing 4 trillion operations (tera-operations) per second (TOPS), using 0.5 watts for each TOPS (2 TOPS per watt). The board also integrates a GC7000 lite GPU, 1 GB of LPDDR4, and 8 GB of eMMC flash memory[9].

The Google Coral supports only TensorFlow Lite as a deep learning framework.

IV. Methodology

We have investigated 2 machine learning models (one with TensorFlow and one with PyTorch) on top of existing work [2, 3, 4]. The PyTorch model was pretrained using Deep Q-Learning against a random agent, and we trained the

TensorFlow model using Deep Q-Learning with MCTS against a previous version of itself. Thereafter, we convert the original PyTorch model to TensorFlow Lite (via ONNX) for compatibility with the Google Coral. We also convert the TensorFlow model to TensorFlow Lite. ONNX is a library for converting ML models to a lighter version fit to be deployed on edge devices.

We find that the TensorFlow model beats the PyTorch model consistently, so we design a GUI to integrate with the TensorFlow model.

A. PyTorch Model

The PyTorch model consists of a single 6*7 tensor as input and 7 convolutional layers (kernel size of 5) with 32 filters followed by 4 fully connected layers. The activation layer we implemented is the Leaky ReLU(Rectified Linear Unit). The model was pre trained using Deep Q-Learning for 20000 games and is able to beat a random opponent 94% of the time. The game state is fed into the neural network to get action probabilities.

The PyTorch code contains:

- The game environment.
- The deep CNN as a function approximator of state action values.
- Application of Bellman optimality equation in training a Deep-Q network.
- Ability for the agent to learn from scratch and win Connect4 with no human designed rules.

The agent has been achieved effective and stable performance and demonstrated high win rate and decreasing steps needed for a win [3].

1) Game Environment: The game state is stored as a 1D array. The players are represented by 'O' and 'X' respectively (in any order). It contains the class methods-render (for showing the board state), reset (to clear the board), get_available_actions (for scanning the board state and return the list of legal moves available), check_game_done (to check whether any player has won), and make_move (to make a move on behalf of the current player).

2) Deep Q Network: Our model uses a CNN[3] that takes into the state as an image of board_state and output the state action function value for all states. In effect, the network is trying to predict the expected return of taking each action given the current input. The output of the model is given to the get_available_actions() module that returns the set of legal moves the current player can make.

The network uses a single image map as 6 * 7 * 1 encoded as 1 as the moves made by player 1 and 2 as the moves made by player 2 and 0 for a blank space.

3) Agent: The pretrained model that we used was trained using the free GPU provided by Google Colab.

While training, the agent uses an exploitation/ exploration trade-off. The ϵ (epsilon) (that determines the chances of exploring new options versus taking game paths already seen before) was set to 0.9 at first but decayed exponentially to 0.05). The agent plays non-deterministically during training and deterministically after training completes. The rewards us are 1 for a win, 0.5 for a tie and -1 for a loss. The batch size we used during training is 256 states.

In [3], the agent plays against random agent. It learns how to play and win the game at an elementary level. The agent is expected to perform better than the random player at the end of training episodes.

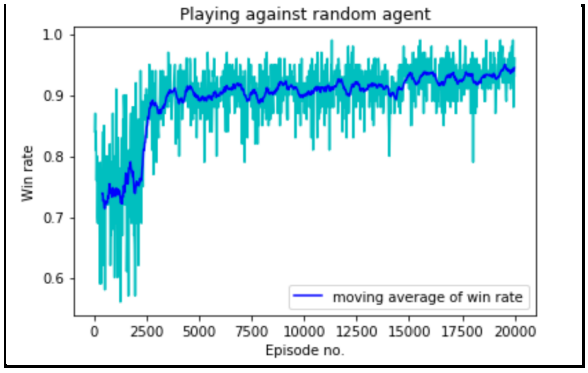

Fig. 2: PyTorch Deep Q-learning agent vs random player. From [3].

As one can see from the figures above and below [3]:

- The variance of the win rate is decreasing and the moving average for the win rate is increasing.
- The average number of steps to win has decreased so the agent learns to win faster vs a random player.

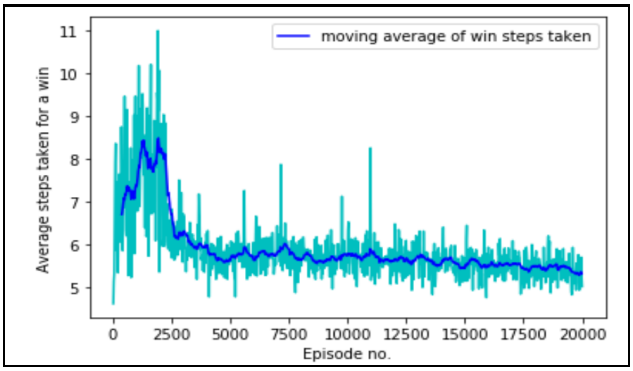

Fig. 3: PyTorch Deep Q-learning agent vs random player. From [3].

B. TensorFlow Model

The TensorFlow model consist of two 6*7 tensors as input and 6 convolutional layers (kernel size 4) with 75 filters and residual connections to the previous layers. We have used Leaky ReLU activation and Batch Normalization in this model and have trained it on the GPU for 96 hours. The TensorFlow model makes use of the Monte Carlo Tree Search to select the best possible mov to beat the opponent. In this model, we have set the number of MCTS simulations to 50 (starting from the current state) so that the model can calculate 50 moves ahead and select the move with the highest probability of winning. The model moves to an unseen state and then adds it to the list of seen states. It then passes the state to the neural network to evaluate its value. The calculated value returned by the neural network is propagated back to the current state from the unseen state. After the value of possible

states has been computed through simulations by the model , it chooses the action which brings the highest value to the child state of the current state.

On each move the agent has a multinomial distribution of actions over the columns. It plays non deterministically (the model samples from action distribution instead of choosing highest value) for 10 moves and then switches to a deterministic mode of play.

An overview of the modules is below:

1) game.py: provides the same functionality as the game environment in the PyTorch model except that it stores the game environment is 2-D array.

2) MCTS.py: contains the code for implementing the MCTS algorithm which includes modules such as *movetoLeaf(), addNode() and backfill().*

3) agent.py: defines 2 classes – one for the user and the other for the Reinforcement Learning Agent. The Reinforcement Learning Agent begins at the current game state and conducts the configured number of MCTS simulations to determine the next best possible move. During each MCTS simulation, the model moves to an unseen game state, evaluates the game state using the Deep Neural Network, and propagates the value to all the ancestor states. Then if it is playing deterministically, it selects the child state with the highest reward. Similar to the PyTorch model, illegal moves are discarded by the agent.

4) model_tournament.py: contains the code for making the two models play between the PyTorch and Tensorflow models in the functions tourney() and tourney2().

5) funcs.py: a generic module which causes the games to be played between any two agents.

6) main.py: runs the training algorithm . It can either start from scratch or from the most recent training result. In an infinite loop the best previous model has been fixed . It plays against a copy of itself except that the copy is allowed to be trained. During this, the trained model plays non deterministically. They play against each other 25 times. Afterwards, the previous model and new models play against each other 20 times. If the new mode wins more than 1.3 times than the previous model then the new model is set as the best model and is checkpointed.

7) memory.py: It provides a way to cache or store the no of game state nodes (that increases as the no of games played increases). This cache allows the agent to skip past the game state to the neural network.

8) config.py: This module contains hyperparameters for training.

V. Comparison Between Models

We set up experiments to compare the agents produced from the PyTorch and TensorFlow Models. The results we obtained are: TensorFlow Model (Monte Carlo Tree Search) wins 100% (20/20) of games when going first OR second, playing deterministically or non-deterministically. Results from deterministic play for different turn orders are shown in Figures 4 and 5. The PyTorch model is much weaker because it was trained against a random opponent, whereas the TensorFlow model was trained against the best version of itself.

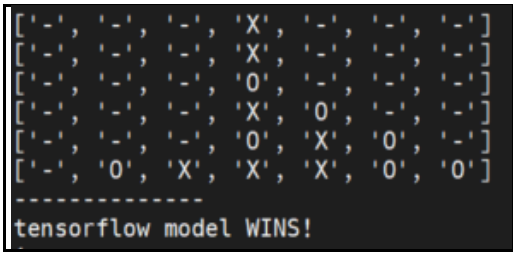

Fig. 4: PyTorch Deep Q-learning agent vs TensorFlow MCTS agent when playing deterministically and when PyTorch model moves first.

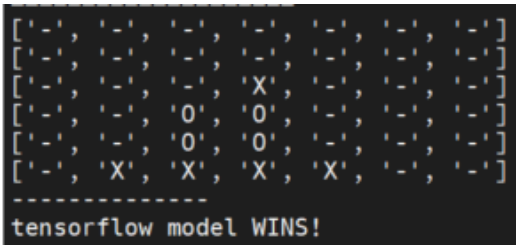

Fig. 5: PyTorch Deep Q-learning agent vs TensorFlow MCTS agent when playing deterministically and when PyTorch model moves second.

VI. Running the Code

The GitHub repository for this project [6] allows the user to play against a trained RL agent using the GUI. It also includes the PyTorch and TensorFlow models and the GUI code.

ML training was run on Ubuntu Linux with Python 3.9 and an NVIDIA Quadro RTX 8000 GPU. The TensorFlow Lite version of the TensorFlow model can be on systems with a CPU, GPU, and/or TPU. The requirements are installed with the command:

```
pip3 install -r requirements.txt
```

There are several options to run the GUI depending on the type of system. In all cases, the command to run is:

```
python3 ./GUI/gui_against_ai.py
```

Configuration of the game is in the *./GUI/tensorflow_model/config.py* file. Increasing the MCTS_SIMS will result in a model that is harder to beat but will take longer to make decisions. Other configurations based on the system being used are detailed below.

A. Run on CPU

This option runs using TensorFlow Lite on a CPU. This will allow the user to play against a trained version of the TensorFlow agent. The model version is taken from the function *setup_ai()* in *./GUI/tensorflow_model/ai_player.py* file. The variable TFLITE in the *./GUI/tensorflow_model/config.py* file must be True and TPU must be false.

B. *Run on GPU*

If the system has a GPU, the variable TFLITE can be set in *./GUI/tensorflow_model/config.py* to be false to run with TensorFlow, or you can set it to true to run with TensorFlow Lite (which is faster). TPU must be false.

C. *Run on Google Coral (TPU)*

The steps to set up the Google Coral are outlined in [10]. This takes a few hours. Then, ensuring that the Google Coral has internet connection (through ethernet or Wi-Fi), the requirements can be installed using the command stated above. The TFLITE variable is to be set in *./GUI/tensorflow_model/config.py* to True and the TPU variable is also set to True. We then connect the Google Coral device to a computer monitor via HDMI cable.

VII. GUI

We developed the GUI (Graphical User Interface) using Python. We have used the colors red and blue to depict the players (Red in this case represents the AI agent). We have made use of *Tkinter*- a python binding of and the standard python interface of the Tk GUI toolkit. Implemented as a Python wrapper around a complete *Tcl* interpreter embedded in the Python interpreter, *Tkinter* calls are translated into *Tcl* commands, which are fed to this embedded interpreter, thus making it possible to mix Python and *Tcl* in a single application.

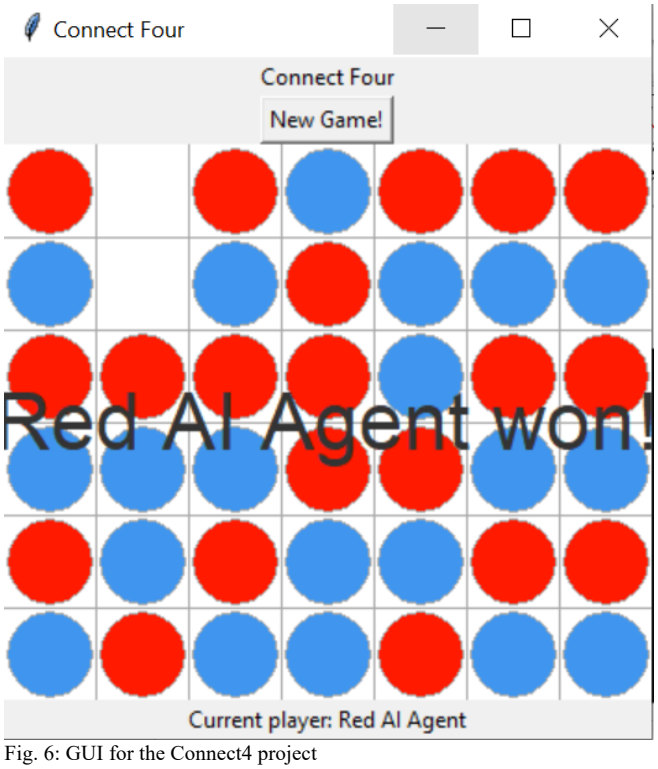

Fig. 6: GUI for the Connect4 project

In order to connect the GUI class to the AI model , we wrote a different module – *ai_player.py* with 3 functions, namely, *setup_ai(), convert_game_state(),* and *model_column()*. The *setup_ai()* function sets up the Reinforcement Learning agent. The *convert_game_state()* function transforms the game state to format compatible with the GUI and the *model_column()* function gets the RL agent's next move for a given game state.

VIII. Conclusion

In this work, we have managed to show that the Reinforcement Learning based AI agent is able to play and beat a random human opponent all of the time regardless of whether it makes the first or second move. We constructed two models- PyTorch and TensorFlow, converted them into lightweight models and deployed them on the Google Coral edge device.

References

[1] M. Ashhad. "AlphaGo: The AI that defeated a World Champion". tessalatedscience.com. https://tessellatedscience.com/alphago-the-ai-that-defeated-a-world-champion/.

[2] D. Foster. "How to build your own AlphaZro using Python and Keras". Medium.com. https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188.
github.com. https://github.com/AppliedDataSciencePartners/DeepReinforcementLearning (TensorFlow model).

[3] N. Yung. "A Reinforcement Learning agent trained without prior Human Knowledge".github.com. https://github.com/neoyung/connect-4 (PyTorch model).

[4] U. Hekić. "Connect Four game written in python". Github.com. https://github.com/uroshekic/connect-four (GUI).

[5] G. Wisney. "Deep Reinforcement Learning and Monte Carlo Tree Search with Connect4". towardsdatascience.com. https://towardsdatascience.com/deep-reinforcement-learning-and-monte-carlo-tree-search-with-connect-4-ba22a4713e7a.

[6] J.O.B. Weiss. "Reinforcement Learning Connect4 Agent". Github.com. https://github.com/jonahobw/ece697/blob/main/PyTorch_model/README.md.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, "Playing Atari with Deep Reinforcement Learning". DeepMind Technologies. NIPS Deep Learning Workshop, Dec. 2013. Available: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[8] "Everything You Need to Know about Google Coral". Okdo.com. https://www.okdo.com/blog/everything-you-need-to-know-about-google-coral/.

[9] "Google Coral Range of Products". arrow.com. https://www.arrow.com/en/research-and-events/articles/the-new-google-coral#:~:text=The%20Coral%20Dev%20Board%20is%20a%20powerful%20single-board,LPDDR4%2C%20and%208%20GB%20of%20eMMC%20flash%20memory.

[10] "Get Started with the Dev Board". Coral.ai. https://www.coral.ai/docs/dev-board/get-started/#requirements.

[11] "Tkinter". en.wikipedia.org. https://en.wikipedia.org/wiki/Tkinter.

[12] "Python GUI with Tkinter". CodersLegacy.com. https://coderslegacy.com/python/python-gui/.