Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Learning to Plan in Large Domains with Deep Neural Networks

Zhenkai Shou

Course of Study: INFOTECH

Examiner: Ph.D. Daniel Hennes

Supervisor: Hung Ngo

Commenced: November 8, 2017

Completed: May 8, 2018

Abstract

In the domain of artificial intelligence, effective and efficient planning is one key factor to developing an adaptive agent which can solve tasks in complex environments. However, traditional planning algorithms only work properly in small domains. Learning to plan, which requires an agent to apply the knowledge learned from past experience to planning, can scale planning to large domains. Recent advances in deep learning widen the access to better learning techniques. Combining traditional planning algorithms with modern learning techniques in a proper way enables an agent to extract useful knowledge and thus show good performance in large domains.

This thesis aims to explore learning to plan in large domains with deep neural networks. The main contributions of this thesis include: (1) a literature survey on learning to plan; (2) proposing a new network architecture that learns from planning, combining this network with a planner, implementing and testing this idea in the game Othello.

Contents

1 Introduction **11**

2 Background **13**
- 2.1 Deep Neural Networks 13
- 2.2 Reinforcement Learning 14
- 2.3 Imitation Learning 14
- 2.4 Planning 15
- 2.5 Two-Player Zero-Sum Board Games 15
- 2.6 Monte-Carlo Tree Search 16

3 Integrating Neural Networks with Planners **19**
- 3.1 Integrating with a Global Planner 19
- 3.2 Integrating with a Local Planner 20
- 3.3 Extracting Forecast Knowledge from a Planner 22

4 Neural Networks that Learn from Planning **25**

5 Experiment Setup **27**
- 5.1 Environment 27
- 5.2 Local Planner 28

6 Model Design **29**
- 6.1 Neural Network Architecture 29
- 6.2 Training 33

7 Experiment Result and Discussion **37**
- 7.1 General Statistics 37
- 7.2 Game-Play Performance 38
- 7.3 Comparison with *Alpha Zero* 39

8 Conclusion and Outlook **41**

A Implementation Details **43**
- A.1 Tournament and Self-Play 43
- A.2 Optimization 44

Bibliography **45**

List of Figures

2.1 Residual connection between layers. . . . 13
2.2 Four phases of Monte-Carlo Tree Search. . . . 16

3.1 Planning with Value Prediction Networks. . . . 21
3.2 Monte-Carlo Tree Search in AlphaGo Zero. . . . 22
3.3 Architecture of Imagination-Augmented Agents. . . . 23

5.1 Othello. . . . 27

6.1 Neural network architecture. . . . 29
6.2 Principal variation in Monte-Carlo Tree Search. . . . 31
6.3 Variable sharing among different heads. . . . 33

7.1 Average loss over the whole training process. . . . 37
7.2 Winning rate of challengers against the baseline player. . . . 38

A.1 Multiprocessing mechanism in tournament and self-play. . . . 43

List of Tables

7.1 Winning rate of the standard model against the *Alpha Zero* variant. 39

1 Introduction

In the domain of artificial intelligence, effective and efficient planning is one key factor to developing an adaptive agent which can solve tasks in complex environments. Planning is an operation that an agent samples the simulated experience from a given simulator or model in order to make good decisions. It allows an agent to look into the future without interacting with the real environment. Planning is very important for some agents like robotics because they need to complete tasks while avoiding wrong decisions which may lead them to irreversible or even dangerous situations. Planners used by the agents can be categorized into two classes: global planners and local planners. A global planner such as value iteration gives a global evaluation of the environment by enumerating all possible states; a local planner such as a tree search algorithm only evaluates a single state by sampling the most promising future from that state onwards.

However, in large domains, a global planner usually suffers the curse of dimensionality, while a local planner becomes less efficient. Therefore, an agent cannot solely rely on the planner in those domains. In order to scale planning in large domains, an agent has to learn from both real and simulated experience, then applies the learned knowledge to planning. This procedure is called learning to plan.

Recent advances in deep learning widen the access to better learning techniques. Many different approaches have been proposed to explore learning to plan in combination with deep neural networks. For example, Value Prediction Networks (Oh et al., 2017) embed a local planner into a deep neural network; *Alpha Zero* (Silver et al., 2017) maintains a local planner and a deep neural network separately. In those methods, a neural network, which maps the current state to the target value, learns prior knowledge from the agent's experience, and the agent uses the prior knowledge to bias the local planner towards more promising future. However, in those methods, the neural network cannot fully leverage the local planner since forecast information contained in the local planner is not served as input to the network. Here we raise the following question: can we design a neural network that can fully leverage the local planner to further improve the effectiveness and efficiency of planning?

In this thesis, we mainly contribute on two topics. First, we discuss the recent development of reinforcement learning approaches that integrate neural networks with planners. We classify those approaches into different categories and explain each category with one example. Second, we propose a deep neural network that can fully leverage a local planner. Our method extracts useful forecast information from the planner, and feeds it into the deep network. Our network gives evaluation based on both the agent's current state and forecast information derived from the planner. We implement and test our idea in the game Othello. The empirical result shows that this evaluation is much better than the one based solely on the current state. The actual performance of our method is better than *Alpha Zero*, a baseline algorithm in our experiment. The code for our implementation can be downloaded from the following repository: https://github.com/ZhenkaiShou/Deep-Networks-that-Learn-to-Plan.

2 Background

This section describes some background knowledge and algorithms that are related to this thesis.

2.1 Deep Neural Networks

A deep neural network is basically a neural network with many hidden layers. It maps an input value to some output value via non-linear functions. A deep neural network can approximate any functions due to its flexibility in the parameter space. This property is crucial in learning highly non-linear mappings which exist in most machine learning tasks.

A variety of network architectures have been proposed, each of which has its own application domain. In this section we only introduce the ones that are relevant to this thesis.

Convolutional Neural Networks A convolutional neural network is a special neural network that achieves huge success in image processing. It uses convolution operation to connect different layers so that each neuron is only connected to a small subset of neurons in the next layer. The convolutional kernel is shared by all neurons in the same layer, which dramatically reduces the number of parameters in the network, thus making the training of neural networks much more efficient.

Residual Networks A residual network (He et al., 2015) is a special neural network that has large impact on deep learning. It uses a simple but rather effective trick to connect different layers, that is, the input of a layer is connected to output of a later layer. Figure 2.1 illustrates this trick. Empirical result shows that it is easier to train deep neural networks with residual connection than the ones with plain connection.

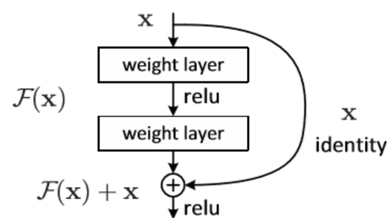

Figure 2.1: Residual connection between layers. This figure is taken from [HZRS15]. If we assume the input of a layer to be x, the output of a later layer to be y, the non-linear function between input and output to be F, then the output can be expressed as $y = F(x) + x$.

Long Short-Term Memory Networks A Long Short-Term Memory Network (LSTM) (Hochreiter and Schmidhuber, 1997) is a special neural network that is designed to remember information for a long term. It makes use of the forget gate, input gate and output gate to manipulate the cell state, which functions as the "memory" unit of the network. Both the network output and the cell state will be carried to the next input signal. Such memory mechanism makes a neural network learning from temporal-dependent data much easier.

2.2 Reinforcement Learning

Machine learning is a tool that generates algorithms. Those generated algorithms distinguish from other analytic algorithms by the fact that the former ones are learned and trained from example data rather than being explicitly programmed. Machine learning can be divided into three classes: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning occurs when an algorithm learns from labeled dataset to map the input data to a certain class or some values. Unsupervised learning occurs when an algorithm learns from unlabeled dataset to find the hidden structure of the input data. Reinforcement learning occurs when an agent learns how to take actions in a given environment where the optimal actions are not given but to yet to be discovered by the agent itself.

In the reinforcement learning category, we consider the Markov Decision Process (MDP). It is a tuple of 5 elements: $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is a set of states; $\mathcal{A}$ is a set of actions; $\mathcal{P} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is the transition that describes the probability of the next state s' after taking action a in state s; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward that describes the received reward r after taking action a in state s; γ is the discount factor. Policy $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$ is the probability distribution over the action space in state s. An episode starts with an initial state s_0. At each time step t the agent receives a state s_t, takes an action a_t based on the policy $\pi(s)$, and gets the reward $r_{t+1} = R(s_t, a_t)$. The next state s_{t+1} is sampled from the transition $P(s_t, a_t)$. Return $G_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1}$ is the discounted cumulative reward starting from time step t. The state-value function $V^\pi(s) = \mathbb{E}^\pi(G_t | s_t = s)$ is the expected return starting from state s, then following policy π. The action-value function $Q^\pi(s, a) = \mathbb{E}^\pi(G_t | s_t = s, a_t = a)$ is the expected return starting from state s, taking action a, then following policy π. There exists an optimal policy π^* such that $Q^{\pi^*}(s, a) \geq Q^\pi(s, a)$ for any state s, action a, and policy π. The agent's goal is to find out such optimal policy π^* that maximizes the expected return.

2.3 Imitation Learning

Reinforcement learning is known to be data inefficient as the label of input data is not provided to the agent. For typical reinforcement learning algorithms such as Q-learning and policy gradient, the agent randomly explores in the environment and learns from data collected during self-exploration. Imitation learning is a technique in reinforcement learning that improves data efficiency. It allows the agent to learn from an expert, mimicking its behavior. The expert may exist in the form of human demonstration, or output result of other algorithms. However, it is worth noticing that the quality of the expert may influence the performance of the agent. Human demonstration is not optimal in some cases, which may lower the agent's final performance; output result of other algorithms needs

to be superior than the agent's current policy throughout the whole training progress in order to provide meaningful guidance. With imitation learning, there is no need for the agent to discover hidden knowledge of the environment by itself, which yields more efficient training as long as a proper expert can be found.

2.4 Planning

Planning is an operation that an agent samples the simulated experience from a given simulator or model in order to make good decisions. It helps an agent to find out the optimal policy in MDP problems. Planners can be categorized into two classes: global planners and local planners.

Global Planner A global planner gives a global evaluation of the environment by enumerating all possible states. It suffers the curse of dimensionality since it tries to evaluate all states at the same time. Therefore, a global planner is hardly used in any large domains.

A typical example of global planner is the value iteration (Bellman, 1957). It alternatively update the Bellman Optimality Equations for each state and action, starting with $V(s) = 0$ for any $s \in \mathcal{S}$:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s'), \tag{2.1}$$

$$V(s) = \max_{a} Q(s,a). \tag{2.2}$$

Local Planner A local planner evaluates a single state by sampling the most promising future from that state onwards. It is much more efficient than a global planner since only the given state and its successor states may be considered. However, a local planner still becomes less efficient in large domains due to the increased planning steps required as well as the increased uncertainty of future. In general, a local planner applies a tree search algorithm, e.g. Monte-Carlo Tree Search, $\alpha - \beta$ pruning. A detailed description of Monte-Carlo Tree Search is given in Section 2.6.

2.5 Two-Player Zero-Sum Board Games

One challenging research field for reinforcement learning is two-player zero-sum board games because they require deep reasoning. In these games, two participating players compete against each other and never cooperate. In most cases, a game ends up with one winner and one loser. In some games, it is also possible that the game reaches some state that no one can win the game, which is called draw. These games are usually fully observable: both parties can observe the whole state from the game board. The initial state of these games are usually fixed, e.g. for Go the game board is empty while for Chess each party has 16 pieces at predefined positions.

These games can often be formulated as follows. A game begins with initial state s_0. Each party takes a move alternately and the next state is generated according to the game rule until terminal step T. The player gets a reward $r_{t+1} = 0$ at non-terminal steps $t < T$, or a reward $r_T \in \{-1, 0, 1\}$ at terminal step T for losing/drawing/winning a game. Binary rewards are applied to these games

because it is very difficult to manually shape the reward function. Note that the total reward $G_0 = \sum_{i=0}^{T-1} \gamma^i r_{i+1}$ is equal to the reward r_T at terminal step T with discount factor $\gamma = 1$. So the total reward of both parties will always be zero. In theory, value iteration can be applied here to accurately evaluate all possible positions of the game because the environment is fully observable. However, this approach is infeasible due to the enormous state space of these games, e.g. state space of size 10^{170} for Go. Therefore, it is necessary to apply the "learning to plan" concept here in order to find out the optimal policy.

2.6 Monte-Carlo Tree Search

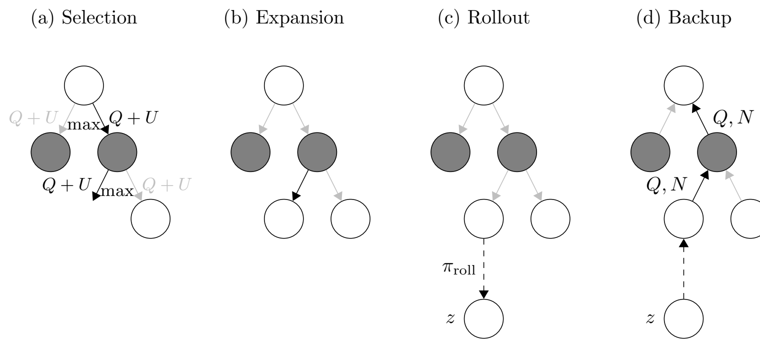

Figure 2.2: Four phases of Monte-Carlo Tree Search. (a) Selection. Starting from the root node s the most promising child node is selected according to the tree policy $\pi_{\text{tree}}(s)$ until a leaf node is reached. $\pi_{\text{tree}}(s)$ usually follows UCT algorithm to encourage exploration of less frequently visited child nodes. (b) Expansion. When the searching reaches a leaf node, a new child node will be created and added to the tree unless the leaf node is a terminal node. (c) Rollout. The simulation continues with a rollout policy π_{roll} (e.g. uniformly random) from the leaf node until the terminal state, receiving reward z. (d) Backup. The reward z is backpropagated from the leaf node to the root node.

Monte-Carlo Tree Search (MCTS) (Coulom, 2006) is a local planner which is often used in game play. It tries to keep the balance between exploration and exploitation when searching for the most promising future.

A search tree is composed of several nodes and edges. A node represents a state of the environment. An edge, which connects a parent node and a children node, represents an action that the parent node may take. A children node, which is connected to a parent node by an edge, represents a successor state of its parent node after taking the corresponding action.

The search tree begins with a single root node, which represents the current state s. MCTS executes k rounds of simulation. Each simulation contains following stages:

Selection Starting from the root node s the most promising child node is selected according to the tree policy $\pi_{\text{tree}}(s)$ until a leaf node is reached. $\pi_{\text{tree}}(s)$ often follows the Upper Confident Bounds for Trees (UCT) (Kocsis and Szepesvári, 2006) algorithm:

$$U(s,a) = c_{\text{uct}} \sqrt{\frac{\log N(s,a)}{N(s)}}, \tag{2.3}$$

$$Q(s,a) = \frac{W(s,a)}{N(s,a)}, \tag{2.4}$$

$$\pi_{\text{tree}}(s) = \arg\max_a Q(s,a) + U(s,a), \tag{2.5}$$

where $N(s,a)$ is the visit counts of edge a; $N(s) = \sum_a N(s,a)$ is the visit counts of node s; $W(s,a)$ is the sum of all rewards obtained from simulations that pass through the edge a. The action with maximum $Q + U$ value will be selected by the tree policy. UCT encourages exploration of less frequently visited child nodes.

Expansion When the searching reaches a leaf node, a new child node will be created and added to the tree unless the leaf node is a terminal node. This new child node is initialized with $W(s,a) = 0, N(s,a) = 0$ for each edge a.

Rollout The simulation continues with a rollout policy π_{roll} (e.g. uniformly random) from the leaf node until the terminal state, receiving reward z.

Backup The reward z is backpropagated from the leaf node to the root node. Update the action-value Q and visit counts N of all nodes along the trajectory:

$$W(s,a) = W(s,a) + z, \tag{2.6}$$

$$N(s,a) = N(s,a) + 1. \tag{2.7}$$

Figure 2.2 briefly illustrates these four stages. Once the search is complete, the planner selects the action with the most visit counts at root state s:

$$\pi(s) = \arg\max_a N(s,a). \tag{2.8}$$

3 Integrating Neural Networks with Planners

Reinforcement learning has been studied for several decades. Integrating neural networks with planners to solve MDP problems becomes popular in recent years. A neural network grants an agent the generalization ability to new scenarios. It reasons fast by mapping an input to some output value. On the other hand, a planner grants an agent the ability to look into the future. It reasons slow by simulating future. By integrating these two pieces together can the agent own the ability of both fast reasoning and slow reasoning, as illustrated by Anthony et al. (2017). Several different approaches have been proposed to integrate neural networks with planners.

3.1 Integrating with a Global Planner

Embedding a global planner into a neural network is a promising approach for specific environments. Tamar et al. (2016) discovered that value iteration can be formed as a special type of convolutional neural network, the Value Iteration Networks (VIN). It tries to solve true MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ by converting it into a latent MDP $\bar{\mathcal{M}} = (\bar{\mathcal{S}}, \bar{\mathcal{A}}, \bar{\mathcal{P}}, \bar{\mathcal{R}}, \bar{\gamma})$. In general, $\mathcal{M} \neq \bar{\mathcal{M}}$. VIN learns the latent reward $\bar{r}$ from the environment:

$$\bar{r} = f^r(s, g), \tag{3.1}$$

where s is the global state, g is the goal, f^r is a convolutional neural network. Then it performs value iteration on the latent MDP for N iterations, starting with $\bar{V}_0$ the zero tensor:

$$\bar{Q}_{t+1} = \bar{r} + f^p(\bar{V}_t), \tag{3.2}$$

$$\bar{V}_t = \max_{\bar{a}}(\bar{Q}_t), \tag{3.3}$$

where f^p is a convolutional layer whose kernel represents the latent transition $\bar{\mathcal{P}}$; $\max_{\bar{a}}(\cdot)$ performs max-pooling over the channels. Afterwards, it maps the action-value in the latent MDP to the one in the true MDP, and selects the action that maximizes the action-value in the true MDP:

$$Q = f^{\text{map}}(\bar{Q}_{N+1}), \tag{3.4}$$

$$a = \arg\max_{a}(Q), \tag{3.5}$$

where f^{map} is a mapping function, e.g. a fully connected layer.

VIN integrates a neural network with a global planner at the network architecture level. It shares the same shortcoming as value iteration that it suffers the curse of dimensionality. It also requires that the environment could be represented as a grid structure, which limits its usage mainly in the field of navigation.

Niu et al. (2017) extends VIN to Generalized VIN (GVIN) so that the environment could be represented as any graph structure. The extension to Partially Observable MDP (POMDP) has also been well explored. QMDP-Net (Karkus et al., 2017) and Cognitive Mapping and Planning (CMP) (Gupta et al., 2017) both use VIN as planning module under partial observability.

3.2 Integrating with a Local Planner

Integrating a local planner with a neural network is an alternative choice for large domains. The integration can be categorized into different classes, based on how the neural network and local planner are integrated together.

3.2.1 Implicit Integration

For implicit integration, we only maintain a special neural network where the local planner is encoded by the network architecture. The perfect environment is often not given to the agent. Value Prediction Networks (VPN) (Oh et al., 2017) and TreeQN (Farquhar et al., 2017) fall under this category. Both of them can be viewed as an extension of Deep Q-Networks (DQN) (Mnih et al., 2015), a simple model-free network which is capable of playing Atari games at human level.

In VPN, the state s is first encoded into some feature x:

$$x = f^{\text{enc}}(s). \tag{3.6}$$

From feature x it learns a model (reward, discount and transition) of the environment:

$$r, \gamma = f^{\text{out}}(x, a), \tag{3.7}$$

$$x' = f^{\text{trans}}(x, a), \tag{3.8}$$

where a is the action taken at feature x. The state-value of feature x is estimated by a value network:

$$V = f^{\text{value}}(x). \tag{3.9}$$

Then VPN performs tree search planning from the root node x. The planner simulates b-best actions up to a certain depth d using the learned model. The action-value of the root node for each action a is averaged over the value of the most promising child nodes:

$$Q^d(x, a) = r + \gamma V^d(x'), \tag{3.10}$$

$$V^d(x) = \begin{cases} V(x) & \text{if } d = 1 \\ \frac{1}{d}V(x) + \frac{d-1}{d} \max_a Q^{d-1}(x, a) & \text{if } d > 1. \end{cases} \tag{3.11}$$

Figure 3.1 illustrates this planning part. Finally VPN selects the action that maximizes the action-value of the root node.

The architecture of TreeQN is basically the same as VPN, except that TreeQN uses a different local planner and training procedure.

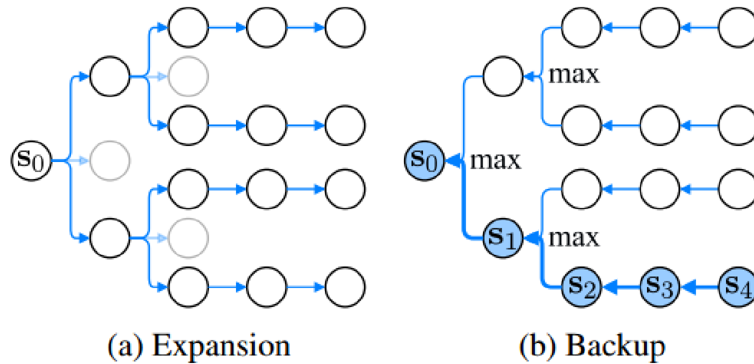

Figure 3.1: Planning with Value Prediction Networks. This figure is taken from [OSL17]. (a) Simulate b-best actions up to a certain depth d. (b) Estimate the action-value by averaging over the value of the most promising child nodes.

3.2.2 Explicit Integration

For explicit integration, we maintain a neural network and a local planner separately. The perfect environment is often assumed to be known so that there is no need to learn such an environment model. *AlphaGo Zero* (Silver et al., 2017) and Expert Iteration (ExIt) (Anthony et al., 2017) fall under this category. Both of them use MCTS as the local planner to simulate the future and guide the neural network.

AlphaGo Zero is the state-of-the-art algorithm that masters the game of Go. In *AlphaGo Zero*, the neural network encodes a game state s to some feature $x = f^{\text{res}}(s)$, followed by a value head $V = f^{v}(x)$ and a policy head $P = f^{p}(x)$ that estimates the value and policy of state s, respectively. This network provides a coarse estimation of any game state, which is used to bias the local planner towards more promising future. The MCTS implemented in *AlphaGo Zero* also differs from vanilla MCTS in the following aspects. First, the tree policy π_{tree} uses a variant of the PUCT (Rosin, 2011) algorithm:

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{N(s)}}{N(s, a) + 1}. \tag{3.12}$$

PUCT prefers child nodes with not only fewer visit counts $N(s, a)$ but also higher prior probability $P(s, a)$. Second, *AlphaGo Zero* estimates the value of a leaf node via a more accurate value function $V(s)$ rather than the game result z simulated from a rollout policy. Note that both the prior probability $P(s, a)$ and the value function $V(s)$ can be obtained from the neural network. Once the search is complete, MCTS outputs a searching probability π, which corresponds to the visit frequency of each edge at the root node, as a more accurate policy. *AlphaGo Zero* uses MCTS to play each move. Figure 3.2 illustrates MCTS in *AlphaGo Zero*.

AlphaGo Zero is trained with imitation learning technique that the neural network learns from a better expert, the MCTS. Starting from scratch, the neural network continuously improves its value and policy estimation, so does the expert since MCTS depends on these estimations. The training data (s, π, z) are sampled from the most recent self-play games. *AlphaGo Zero* is trained with minibatch gradient descent on the loss $l = (z - V)^2 - \pi^{\top} \log P + c\|\theta\|^2$. By minimizing loss l the neural network is guided towards the expert.

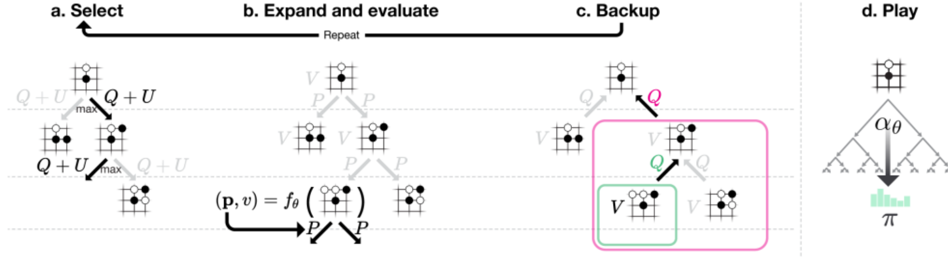

Figure 3.2: Monte-Carlo Tree Search in *AlphaGo Zero*. This figure is taken from [SSS+17]. (a) The tree policy follows a variant of PUCT algorithm, selecting child nodes with fewer visit counts $N(s, a)$ and higher prior probability $P(s, a)$. (b) When the simulation reaches a new leaf node, it will be added to the tree. The value of the leaf node is evaluated by the value network $V(s)$; the prior probability of all edges is evaluated by the policy network $P(s, a)$. (c) The value of the leaf node is backpropagated up to the root node. (d) Once the search is complete, *AlphaGo Zero* plays the move with the most visit counts.

ExIt was developed independently at the same period as *AlphaGo Zero*. It shares the same idea as *AlphaGo Zero* and succeeds in the board game Hex. *AlphaGo Zero* is later evolved to *Alpha Zero* (Silver et al., 2017), which masters multiple kinds of board games.

3.3 Extracting Forecast Knowledge from a Planner

All the algorithms in the above sections share a common issue, that is, the neural network does not fully leverage the planner. Weber et al. (2017) explored how to integrate neural networks with planners in a different way. They proposed Imagination-Augmented Agent (I2A), which learns a contextual feature from the planner and uses the contextual feature as additional input to the neural network. More specifically, I2A maintains a deep neural network consisting of a model-free path f^{mf} and a model-based path f^{mb}. The model-free path f^{mf} is a multi-layer neural network but it outputs a feature x:

$$x = f^{\mathrm{mf}}(s). \tag{3.13}$$

The model-based path f^{mb} builds a model (reward and transition) of the environment:

$$r, s' = f^{\mathrm{model}}(s, a), \tag{3.14}$$

and learns a contextual feature ϕ from a local planner. The local planner is implicitly embedded into the model-based path f^{mb}, just like VPN and TreeQN. For each action a in the current state s, the planner simulates the future up to certain depth d by a rollout policy π_{roll}. Each simulated future trajectory $\tau_i = \{s_1, r_1, \cdots, s_d, r_d | r_1, s_1 = f^{\mathrm{out}}(s, a_i)\}$ is processed by an LSTM module to extract a slice of contextual feature ϕ:

$$\phi_i = f^{\mathrm{lstm}}(\tau_i). \tag{3.15}$$

The contextual feature ϕ encodes all useful information about the future. The deep network outputs the policy π and value V based on both x and ϕ:

$$P, V = f^{\mathrm{out}}(x, \phi). \tag{3.16}$$

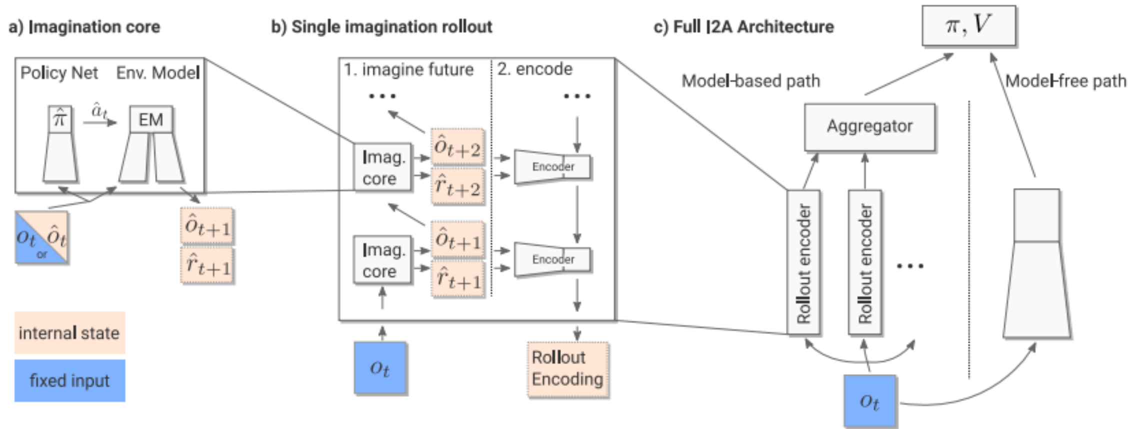

Figure 3.3: Architecture of Imagination-Augmented Agents. This figure is taken from [WRR+17]. (a) I2A builds a model of the environment and simulates with the rollout policy on the environment model. (b) The local planner simulates a future trajectory up to a certain depth and extracts contextual feature from the trajectory via LSTM. (c) The policy and value depends on the output feature from both the model-free path and the model-based path.

Figure 3.3 illustrates the whole I2A architecture.

I2A leverages both current knowledge and forecast knowledge. Thus it surpasses both model-free and model-based reinforcement learning approaches in data efficiency, performance and robustness. Our work is mostly close to I2A that we use such contextual feature to improve the network evaluation.

4 Neural Networks that Learn from Planning

In this section we discuss how to design a neural network so that it can fully leverage the local planner. We assume that the environment is deterministic, and the perfect environment is also given to the agent.

The input to the network is the agent's current state s. The state s itself is not ideal for the evaluation task, so the network learns a feature x from state s via a residual tower f^{res}:

$$x = f^{\text{res}}(s). \tag{4.1}$$

The feature x encodes all useful information of the current state. From feature x the network is then split into two heads, namely, the value head f^v and the policy head f^p, estimating the value logits u_v and policy logits u_p of the current state s, respectively:

$$u_v = f^v(x), \tag{4.2}$$

$$u_p = f^p(x). \tag{4.3}$$

The logits u_v and u_p are then rescaled to the desired range by an activation function h^u and h^v, respectively, forming the output value V and policy P:

$$V = h^v(u_v), \tag{4.4}$$

$$P = h^p(u_p). \tag{4.5}$$

The agent can evaluate its current state s by using a standard model-free network architecture. Below we will discuss how to combine our idea, learning from forecast knowledge, with the above network.

We believe that the local planner can tell the agent not only the future prediction such as the value or policy of future states, but also the future itself. Therefore the local planner in our method plays the role of both future prediction and forecast information extraction. We need to obtain useful forecast information from the planner so that the deep network could learn from forecast information. Since a local planner in general applies a search tree algorithm, we can trace the best trajectory s_{seq} (e.g. the most frequently visited trajectory) in the search tree. The best trajectory s_{seq} tells the agent the most promising future if the agent follows its current policy. For the same reason that states in s_{seq} are not suitable for evaluation tasks, The best trajectory s_{seq} has to be converted to a feature sequence x_{seq} via the residual tower f^{res}. The feature sequence x_{seq} is a list of future features with unfixed length, as a result it cannot be directly processed by the deep network which requires inputs to have fixed size. Instead we apply an LSTM f^{lstm} to learn contextual feature ϕ from the feature sequence x_{seq}:

$$\phi = f^{\text{lstm}}(x_{\text{seq}}). \tag{4.6}$$

LSTM is good at remembering a long input sequence, so the resulting contextual feature ϕ encodes all the useful information of the feature sequence x_{seq}.

Now we can equip the deep network with both current knowledge x and forecast knowledge ϕ. We wish that the deep network could learn how to correct its original evaluation of V and P by using both x and ϕ. Therefore, the network outputs the delta logits Δu_v and Δu_p via the corresponding calibration heads f_{cal}^{v} and f_{cal}^{p}:

$$\Delta u_v = f_{\text{cal}}^{v}(x, \phi), \tag{4.7}$$

$$\Delta u_p = f_{\text{cal}}^{p}(x, \phi). \tag{4.8}$$

Δu_v and Δu_p serve as a local correction of the original logits u_v and u_p. The calibrated value V' and policy P' can be computed as follows:

$$V' = h^{v}(u_v + \Delta u_v), \tag{4.9}$$

$$P' = h^{p}(u_p + \Delta u_p). \tag{4.10}$$

Now the agent can better evaluate its current state s by leveraging the forecast knowledge ϕ. However, the calibrated estimation V' and P' cannot be directly used in the local planner because the feature sequence x_{seq} can only be obtained after planning is complete. To compensate for this shortcoming, we introduce an imitation module f^{imi} that learns an imitated contextual feature $\hat{\phi}$ without using the feature sequence x_{seq}:

$$\hat{\phi} = f^{\text{imi}}(x). \tag{4.11}$$

The imitation module f^{imi} functions as a fast reasoning module of the agent so that the agent can obtain forecast knowledge by fast reasoning rather than slow planning. The imitated contextual feature $\hat{\phi}$ plays the same role as the real contextual feature ϕ except that $\hat{\phi}$ is directly generated from feature x. Now the agent can better evaluate its current state s in a model-free approach, without the help of a planner:

$$\Delta \hat{u}_v = \hat{f}_{\text{cal}}^{v}(x, \hat{\phi}), \tag{4.12}$$

$$\hat{V}' = h^{v}(u_v + \Delta \hat{u}_v), \tag{4.13}$$

$$\Delta \hat{u}_p = \hat{f}_{\text{cal}}^{p}(x, \hat{\phi}), \tag{4.14}$$

$$\hat{P}' = h^{p}(u_p + \Delta \hat{u}_p). \tag{4.15}$$

We have developed a deep neural network that can fully leverage a local planner. In our model, the agent is enhanced in two different aspects. First, the deep network improves its evaluation by using both current knowledge and forecast knowledge. Second, the evaluation of the network is used to bias a local planner. A detailed description of our model, including the neural network architecture and the training procedure, is given in Section 6.

5 Experiment Setup

In this section we briefly describe the environment and the local planner used in our experiment.

5.1 Environment

We test our idea in Othello (also known as Reversi), a two-player competitive board game. The two participants, black and white, alternatively place a stone of their color in the empty positions of a 8×8 game board. A stone can only be placed in the position where the current player can flip at least one opponent stones. A player can only pass its turn (do not place any stone on the game board) if no other moves are available. A game begins with 2 black stones and 2 white stones diagonally placed in the center of the game board. A participant wins the game if it has more stones than the opponent at the end of the game. A game may end up with a draw if both players have the same amount of stones at the end of the game. Figure 5.1 shows an example of the game play.

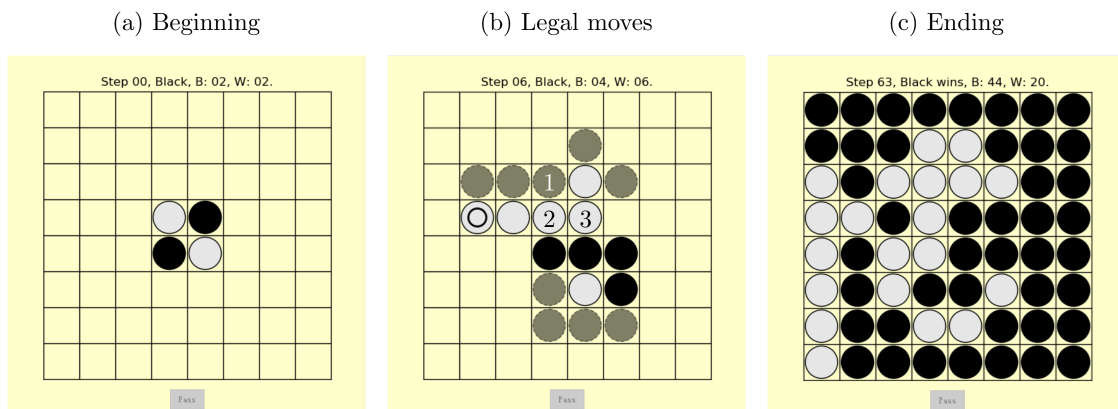

Figure 5.1: Othello. (a) The game begins with 4 stones placed in the center of the game board. (b) Gray circles represent the legal moves for the current player (black). A move is legal if the player can flip at least one opponent stones. For example, if black plays at position 1, then it can flip the white stones to black at position 2 and 3. (c) Black wins the game with score 44:20.

Othello has simple rules, which makes it easy to implement. But it also requires deep strategy, which makes it a challenging task for intelligent agents. Furthermore, it has large state space, which makes it an ideal test-bed for learning to plan in large domains.

5.2 Local Planner

We choose MCTS as the local planner since it is a powerful and widely used local planner for board games. In particular, we use the PUCT variant of MCTS, which is the planner implemented in *Alpha Zero*. See Section 2.6 and Section 3.2.2 for more details. However, we use the model-free calibrated estimation $\hat{V}'$ and $\hat{P}'$ to evaluate the leaf nodes in the search tree, instead of using the basic estimation V and P as in *Alpha Zero*.

6 Model Design

In this section, we discuss the neural network architecture and the training procedure for training an Othello player.

6.1 Neural Network Architecture

Our neural network is based on *AlphaGo Zero*, but with some extension. The network architecture is shown in Figure 6.1.

Figure 6.1: Neural network architecture. The value and policy estimation follows the red arrows; the calibrated value and policy estimation follows the green arrows; the calibrated value and policy estimation without using MCTS follows the blue arrows.

Input State The first input to the network is the state s, which can be obtained from the game board. It contains 4 channels, each of which is a binary image of size 8×8. The first channel describes the position of the current player's stones; the second channel describes the position of the opponent's stones; the third channel is either filled with ones if it is black to play, or filled with zeros if it is white to play; the last channel is either filled with ones if it is white to play, or filled with zeros if it is black to play.

Feature Extraction Feature x is extracted from state s by the residual tower f^{res}, see Equation 4.1. The residual tower f^{res} is composed of a convolutional block and 9 residual blocks. The convolutional block contains the following operations:

(1) a convolution with 64 filters of kernel size 3×3;

(2) a batch normalization;

(3) a rectified linear unit (ReLU).

Each residual block contains the following operations:

(1) a convolution with 64 filters of kernel size 3×3;

(2) a batch normalization;

(3) a ReLU;

(4) a convolution with 64 filters of kernel size 3×3;

(5) a batch normalization;

(6) a skip connection between the input of step 1 and the output of step 5;

(7) a ReLU.

Value Estimation The value V can be estimated by following Equation 4.2 and 4.4, with the activation function $h^v = \tanh$ rescaling the value logits u_v to range $(-1, 1)$. The value head f^v contains the following operations:

(1) a convolution with 2 filters of kernel size 1×1;

(2) a batch normalization;

(3) a ReLU;

(4) a fully connected layer with 64 output units;

(5) a ReLU;

(6) a fully connected layer with a scalar output.

Policy Estimation The policy P can be estimated by following Equation 4.3 and 4.5, with the activation function $h^p = \text{softmax}$ rescaling the policy logits u_p to a probability distribution. The policy head f^p contains the following operations:

(1) a convolution with 1 filter of kernel size 1×1;

(2) a batch normalization;

(3) a ReLU;

(4) a fully connected layer with 65 output units that corresponds to all 64 positions and the pass move.

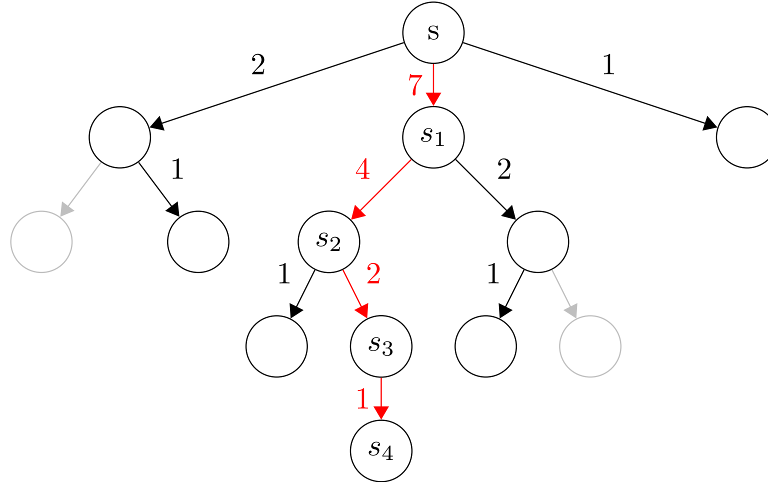

Figure 6.2: Principal variation in Monte-Carlo Tree Search. The principal variation is the most frequently visited trajectory in the search tree. In this figure, we execute MCTS with 10 simulations. The number near the arrow indicates the total visit counts for the corresponding action. The principal variation $s_{\text{seq}} = [s, s_1, s_2, s_3, s_4]$ is shown with red arrows.

Input Feature Sequence The second input to the network is the feature sequence x_{seq}, which can be obtained from the best trajectory in MCTS. We choose the principal variation as the best trajectory. The principal variation $s_{\text{seq}} = [s, s_1, \cdots, s_{\text{leaf}}]$ is the most frequently visited trajectory in the search tree from root node s until leaf node s_{leaf}. Figure 6.2 illustrates this concept. The feature sequence $x_{\text{seq}} = [x_{\text{leaf}}, \cdots, x_1, x]$ can be computed by feeding all states in s_{seq} to the residual tower f^{res} in the reversed order.

Contextual Feature Extraction Contextual feature ϕ can be extracted from feature sequence x_{seq} by the LSTM module f^{lstm}, see Equation 4.6. The LSTM module f^{lstm} contains the following operations:

(1) a Convolutional LSTM (Shi et al., 2015);

(2) a filter that only keeps the last output;

(3) a batch normalization;

(4) a ReLU.

Note that the feature sequence x_{seq} is sorted in the reversed order so that the last output of LSTM corresponds to the contextual feature of the current state s.

Calibrated Value Estimation The calibrated value V' can be estimated by following Equation 4.7 and 4.9. The value calibration head f^{v}_{cal} contains the following operations:

(1) a concatenation of feature x and contextual feature ϕ along the last dimension;

(2) a convolution with 2 filters of kernel size 1×1;

(3) a batch normalization;

(4) a ReLU;

(5) a fully connected layer with 64 output units;

(6) a ReLU;

(7) a fully connected layer with a scalar output.

Calibrated Policy Estimation The calibrated policy P' can be estimated by following Equation 4.8 and 4.10. The policy calibration head f^p_{cal} contains the following operations:

(1) a concatenation of feature x and contextual feature ϕ along the last dimension;

(2) a convolution with 1 filter of kernel size 1×1;

(3) a batch normalization;

(4) a ReLU;

(5) a fully connected layer with 65 output units that corresponds to all 64 positions and the pass move.

Contextual Feature Imitation Imitated contextual feature $\hat{\phi}$ can be learned from feature x by the imitation module f^{imi}, see Equation 4.11. The imitation module f^{imi} is composed of 3 residual blocks. Each residual block contains the following operations:

(1) a convolution with 64 filters of kernel size 3×3;

(2) a batch normalization;

(3) a ReLU;

(4) a convolution with 64 filters of kernel size 3×3;

(5) a batch normalization;

(6) a skip connection between the input of step 1 and the output of step 5;

(7) a ReLU.

Model-Free Calibrated Estimation The model-free calibrated value $\hat{V}'$ can be estimated by following Equation 4.12 and 4.13; the model-free calibrated policy $\hat{P}'$ can be estimated by following Equation 4.14 and 4.15. The model-free value calibration head $\hat{f}^v_{\text{cal}}$ has the same structure as its model-based counterpart f^v_{cal}, except that in step (1) we concatenate feature x and imitated feature $\hat{\phi}$. Note that both f^v_{cal} and $\hat{f}^v_{\text{cal}}$ share the same set of variables, except for the variables in the convolution kernel of step (2). Similar structure and variable sharing mechanism applies to the policy calibration head $\hat{f}^p_{\text{cal}}$. Figure 6.3 illustrates the variable sharing mechanism.

(a) Variable sharing among value calibration heads

Head	Input	Convolution kernel
f^v_{cali}	$x \| \phi$	$k^v_1 \| k^v_2$
$\hat{f}^v_{\text{cali}}$	$x \| \hat{\phi}$	$k^v_1 \| k^v_3$

(b) Variable sharing among policy calibration heads

Head	Input	Convolution kernel
f^p_{cali}	$x \| \phi$	$k^p_1 \| k^p_2$
$\hat{f}^p_{\text{cali}}$	$x \| \hat{\phi}$	$k^p_1 \| k^p_3$

Figure 6.3: Variable sharing among different heads. (a) Both f^v_{cal} and $\hat{f}^v_{\text{cal}}$ share the same set of variables, except for the variables in the convolution kernel. Specifically, we assume the convolution kernel of f^v_{cal} to be $k^v_1 \| k^v_2$, the convolution kernel of $\hat{f}^v_{\text{cal}}$ to be $k^v_1 \| k^v_3$, where $\|$ is the concatenation operator. k^v_1 is the common kernel in both heads due to the same input component x. k^v_2 and k^v_3 correspond to the kernel of the remaining input component ϕ and $\hat{\phi}$, respectively. (b) The same variable sharing mechanism applies to the policy calibration heads.

6.2 Training

The training procedure of our model is similar to *Alpha Zero*. The network is trained with imitation learning technique to continuously learn from a better expert, the MCTS. The training pipeline is composed of three modules: (1) tournament, where the latest player α_{θ_i} competes with the current best player α_{θ_*}; (2) self-play, where the current best player α_{θ_*} generates new self-play data; (3) optimization, where the latest network θ_i is optimized. All these three modules are implemented using multiprocessing, see Appendix A for more details. For each iteration these three modules are executed sequentially, except that for the first iteration $i = 0$ the tournament module is replaced by the initialization module to generate the initial network θ_0. The whole training process is shown in Algorithm 6.1.

6.2.1 Initialization

The network is initialized with uniformly random weights θ_0 sampled from the interval $[-0.05, 0.05]$. The current best network θ_* is also initialized to be the same as the initial network θ_0.

6.2.2 Tournament

The latest player α_{θ_i} competes with the current best player α_{θ_*} for a total of 400 tournament games. For a fair competition, the latest player plays black for 200 games and plays white for the remaining 200 games. Both players use MCTS with 100 simulations to select each move. To encourage variations of the tournament games, we add a light Dirichlet noise to the prior probabilities in the root node, $P'(s, a) = (1 - \epsilon_1)P(s, a) + \epsilon_1\eta$, where $\eta \sim \text{Dir}(0.03)$ and $\epsilon_1 = 0.05$. The behavior policy is alway greedy with respect to the searching probability π, that is, the player always plays the best move throughout the tournament game. If the latest player α_{θ_i} wins at least 55% of the games, then it becomes the current best player α_{θ_*} and it will be used to generate new self-play games.

Algorithm 6.1 Training Process.

```
i = 0
while True do
    if i = 0 then
        # Initialization
        initialize network θ_0
        initialize the current best network θ_* = θ_0
    else
        # Tournament
        hold 400 tournament games between α_{θ_*} and α_{θ_i}
        if α_{θ_i} wins at least 55% of the games then
            update the current best network θ_* = θ_i
        end if
    end if
    # Self-play
    generate 2500 self-play games from the current best player α_{θ_*}
    # Optimization
    for j in range(500) do
        sample a mini-batch size of 256 training data from the most recent 25000 self-play games
    end for
    for j in range(500) do
        update θ_main over loss l_1
    end for
    for j in range(500) do
        update θ_enh over loss l_2
    end for
    for j in range(500) do
        update θ_{imi_1} over loss l_{3_1}
        update θ_{imi_2} over loss l_{3_2}
    end for
    save the network θ_{i+1}
    i = i + 1
end while
```

6.2.3 Self-Play

2500 self-play games are generated from the current best player α_{θ_*}. The best player α_{θ_*} uses MCTS with 100 simulations to select each move for both black and white. To encourage exploration in self-play games, we add a heavy Dirichlet noise to the prior probabilities in the root node, $P'(s,a) = (1-\epsilon_2)P(s,a) + \epsilon_2\eta$, where $\eta \sim \text{Dir}(0.03)$ and $\epsilon_2 = 0.25$. In order to explore all kinds of openings, the behavior policy follows the searching probability π for the first 6 moves, and becomes greedy afterwards. All self-play games are stored to files in the form of $(s, \pi, z, a_{\text{seq}})$, where a_{seq} is the action sequence that follows the principle variation. To save space, we store the action sequence a_{seq} rather than the principal variation s_{seq} itself.

6.2.4 Optimization

The training data $(s, \pi, z, a_{\text{seq}})$ are randomly sampled from the most recent 25000 self-play games. The latest network θ_i is optimized using TensorFlow with a mini-batch size of 256. We use stochastic gradient descent optimizer with momentum = 0.9. The learning rate lr is annealed throughout the whole training process: $lr = 0.02$ for the first 100 iterations; $lr = 0.002$ for the next 100 iterations; $lr = 0.0002$ for the remaining iterations. The optimization is composed of three phases. In each phase, a subset of the network parameters is optimized over some loss for 500 training steps. Once the optimization is complete, the network parameters are saved and a new player $\alpha_{\theta_{i+1}}$ will be generated.

In the first phase we optimize the variables in θ_{main} over the loss l_1:

$$l_1 = (z - V)^2 - \pi \log P + c\|\theta_{\text{main}}\|^2, \tag{6.1}$$

where $c = 10^{-4}$ is a scaling factor for L2 weight regularization; θ_{main} is the set of all variables in f^{res}, f^v, f^p. Loss l_1 is the same loss used in *Alpha Zero*, which enforces the value V and the policy P to be close to z and π respectively.

In the second phase we optimize the variables in θ_{cal} over the loss l_2:

$$l_2 = (z - V')^2 - \pi \log P' + c\|\theta_{\text{cal}}\|^2, \tag{6.2}$$

where θ_{cal} is the set of all variables in f^{lstm}, f^v_{cal} and f^p_{cal}. Loss l_2 enforces the calibrated value V' and policy P' to be close to z and π respectively.

In the third phase we simultaneously optimize the variables in θ_{imi_1} over the loss l_{3_1}:

$$l_{3_1} = c_{\text{f}}\|\phi - \hat{\phi}\|^2 + c\|\theta_{\text{imi}_1}\|^2, \tag{6.3}$$

and the variables in θ_{imi_2} over the loss l_{3_2}:

$$l_{3_2} = (z - \hat{V}')^2 - \pi \log \hat{P}' + c\|\theta_{\text{imi}_2}\|^2, \tag{6.4}$$

where $c_{\text{f}} = 10$ is a scaling factor for the feature loss; θ_{imi_1} is the set of all variables in f^{imi}; θ_{imi_2} is the set of non-shared variables in $\hat{f}^v_{\text{cal}}$ and $\hat{f}^p_{\text{cal}}$ (the convolution kernel k^v_3 and k^p_3 in Figure 6.3). Variables in θ_{imi_1} are updated to make the imitation block f^{imi} mimic the contextual feature ϕ; variables in θ_{imi_2} are updated to make the model-free calibrated value $\hat{V}'$ and policy $\hat{P}'$ to be close to z and π respectively.

7 Experiment Result and Discussion

We demonstrate the performance of our model in the Othello environment. First, we show the general statistics over the whole training process. Then we evaluate the influence of forecast knowledge on the game play. Finally we compare our model with *Alpha Zero*.

7.1 General Statistics

We measure the average loss for each training iteration by randomly sampling an additional 200 mini-batches of training data, which are not used during parameter optimization. Figure 7.1 shows the average value and policy loss over the whole training process. The basic estimation V and P has the highest error (red curves); the calibrated estimation V' and P' has the lowest error (green curves); the model-free calibrated estimation $\hat{V}'$ and $\hat{P}'$ lies in the middle (blue curves). The result implies that the neural network can significantly benefit from forecast knowledge ϕ. Furthermore, using imitated forecast knowledge $\hat{\phi}$ also leads to lower losses, even though it is not as good as using the real forecast knowledge ϕ. In particular, it is rather difficult for the network to learn a better policy from $\hat{\phi}$, which we believe is due to the scaling issue that the network has to learn 65 output units simultaneously.

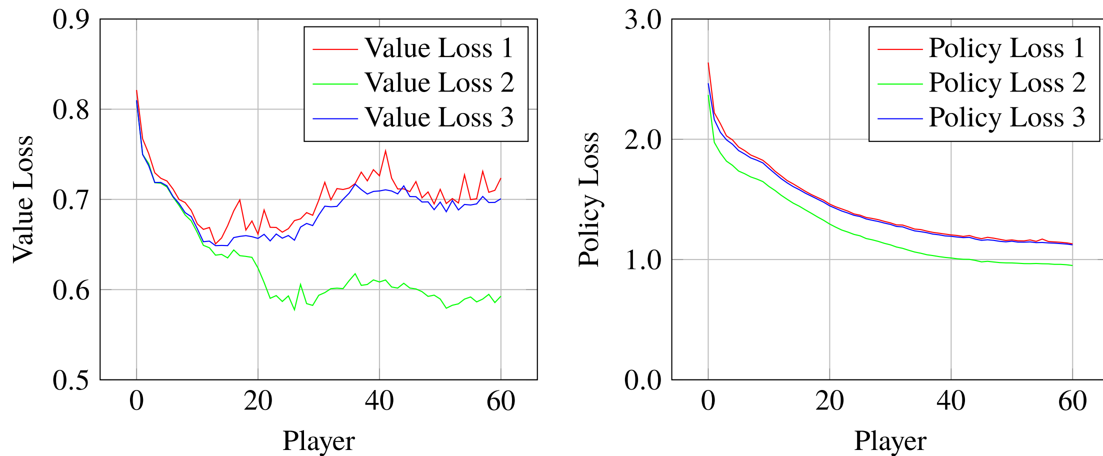

Figure 7.1: Average loss over the whole training process. The red curves correspond to the basic estimation loss l_1; the green curves correspond to the calibrated estimation loss l_2; the blue curves correspond to the model-free calibrated estimation loss l_{3_2}.

7.2 Game-Play Performance

General statistics alone is not enough to evaluate the agent's playing strength. We want to know how forecast knowledge can influence the game play. So we hold 10 groups of tournament to measure the game-play performance. For each group, a different challenger plays against the same baseline player for 400 games. All challengers use the model-free calibrated estimation $\hat{V}'$ and $\hat{P}'$ to evaluate the leaf nodes in the search tree, while the baseline player uses the basic estimation V and P to evaluate the leaf nodes. Those 10 challengers only vary in the number of simulations executed by MCTS. Specifically, challenger in group $i = 1, \cdots, 10$ runs $10 \times i$ simulations during tree search. The baseline player, in contrast, runs 100 simulations throughout all 10 tournaments.

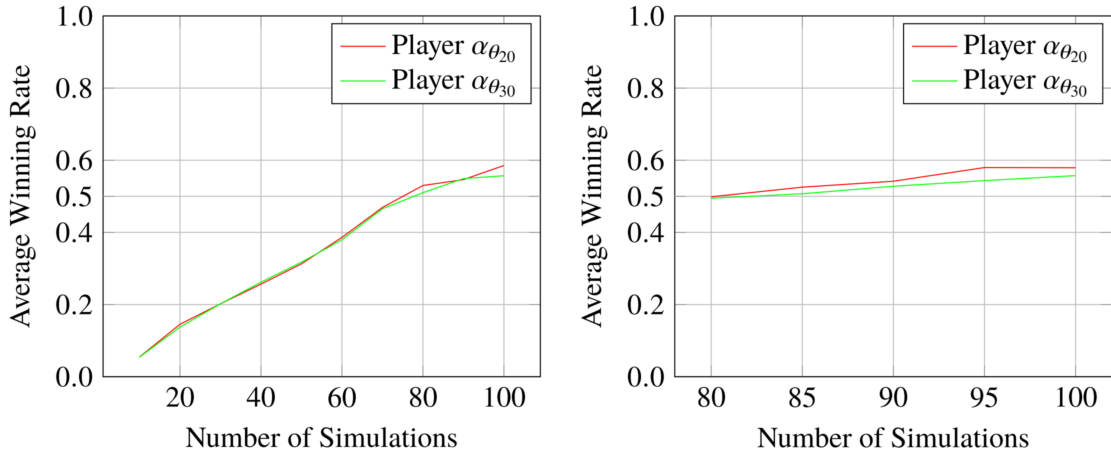

Figure 7.2: Winning rate of challengers against the baseline player. The horizontal axis represents the number of simulations used by the challenger. Winning rate over 0.5 means that the challenger is stronger than the baseline player. *left*: The average winning rate of each challenger is calculated from 3 different trials. *right*: The average winning rate of each challenger is calculated from 10 different trials to reduce the variance of the tournament result.

We conduct this experiment based on player $\alpha_{\theta_{20}}$ and $\alpha_{\theta_{30}}$, and report the average winning rate of each challenger from 3 different trials in Figure 7.2 *left*. The challenger defeats the baseline player when both sides use 100 simulations for the search tree. The challenger has approximately equal playing strength as the baseline player when the challenger only runs 80 simulations per tree search. In order to further validate our experiment result, we conduct an additional experiment with the challenger using 80, 85, 90, 95, 100 simulations for the tree search, and report the average winning rate of each challenger from 10 different trials in Figure 7.2 *right*. We observe similar result from the additional experiment. This concludes that using forecast knowledge, even if it is generated by the agent's intuition rather than the real planning, gives the agent a performance boost in the game play.

7.3 Comparison with *Alpha Zero*

We know that using forecast knowledge can improve the agent's playing strength. However, we still need to verify whether our method can outperform the *Alpha Zero* algorithm. Therefore we trained an *Alpha Zero* variant as a baseline for our comparison. The *Alpha Zero* variant has the same neural network architecture as our standard model, but it uses the basic estimation V and P to evaluate the leaf nodes when generating self-play games. In contrast, our standard model uses $\hat{V}'$ and $\hat{P}'$ to evaluate the leaf nodes so that we can always generate self-play games with potentially higher quality. We hold 3 groups of tournament to compare our model with *Alpha Zero*. Each tournament lasts 400 games. For the first group, both sides use V and P to evaluate the leaf nodes; for the second group, both sides use $\hat{V}'$ and $\hat{P}'$ to evaluate the leaf nodes; for the third group, the *Alpha Zero* variant uses V and P to evaluate the leaf nodes while the standard model uses $\hat{V}'$ and $\hat{P}'$ to evaluate the leaf nodes. For a fair comparison, both sides use their respective network parameters $\theta_i$at the same training step i, and simulate 100 times per tree search.

Player	Alpha Zero	Standard Model	Average Winning Rate
20	0	0	0.5705
20	1	1	0.5939
20	0	1	0.6389
30	0	0	0.5211
30	1	1	0.5955
30	0	1	0.6046

Table 7.1: Winning rate of the standard model against the *Alpha Zero* variant. The value under the column "Alpha Zero" and "Standard Model" indicates whether forecast knowledge $\hat{\phi}$ is used in the network for the corresponding model. Specifically, "0" means the model uses V and P to evaluate the leaf nodes; "1" means the model uses $\hat{V}'$ and $\hat{P}'$ to evaluate the leaf nodes. The average winning rate in each row is calculated from 3 different trials.

We conduct this experiment based on player $\alpha_{\theta_{20}}$ and $\alpha_{\theta_{30}}$, and report the average winning rate of the standard model from 3 different trials in Table 7.1. The standard model defeats the *Alpha Zero* variant in all cases. Note that the standard model only differs from the *Alpha Zero* variant in the self-play stage. This concludes that using forecast knowledge can indeed speed up the training process since better training data can be generated with forecast knowledge.

8 Conclusion and Outlook

In this thesis, we listed some of the recent reinforcement learning approaches that integrate neural networks with planners. We pointed out a common issue among those approaches, that is, the neural network does not fully leverage the planner. We presented a neural network architecture which can correct its initial evaluation by using forecast information derived from a local planner. We tested our idea in Othello games in combination with the *Alpha Zero* algorithm. The empirical result shows that the neural network can provide much more accurate estimation when equipped with both current knowledge and forecast knowledge. However, such accurate estimation cannot be achieved when we replace the real forecast knowledge with an imitated one, since we cannot use the real forecast knowledge during planning. As a result, the actual performance of our method cannot achieve the ideal level where the real forecast information is already provided to the agent before planning, but our method still performs better than the *Alpha Zero* baseline. We believe there is still a large improvement space for our method. Extracting better information from the planner, as well as imitating such forecast information with better final performance, is an interesting direction for future research.

A Implementation Details

The whole training process is implemented using multiprocessing. In this section we explain how this is achieved.

A.1 Tournament and Self-Play

In tournament and self-play, we need to generate a large amount of games. Generating those games sequentially is infeasible for the training. Therefore we use multiprocessing to generate games in parallel. More specifically, we spawn 25 processes and run them on the server in parallel. Each process plays up to 50 games simultaneously using multi-threading. Each game spawns 2 threads to execute MCTS. Once all threads finish one simulation, all the newly expanded leaf nodes will be evaluated by the deep network. Evaluating multiple entries at one time is far more efficient than evaluating one entry at each time. Figure A.1 illustrates this mechanism.

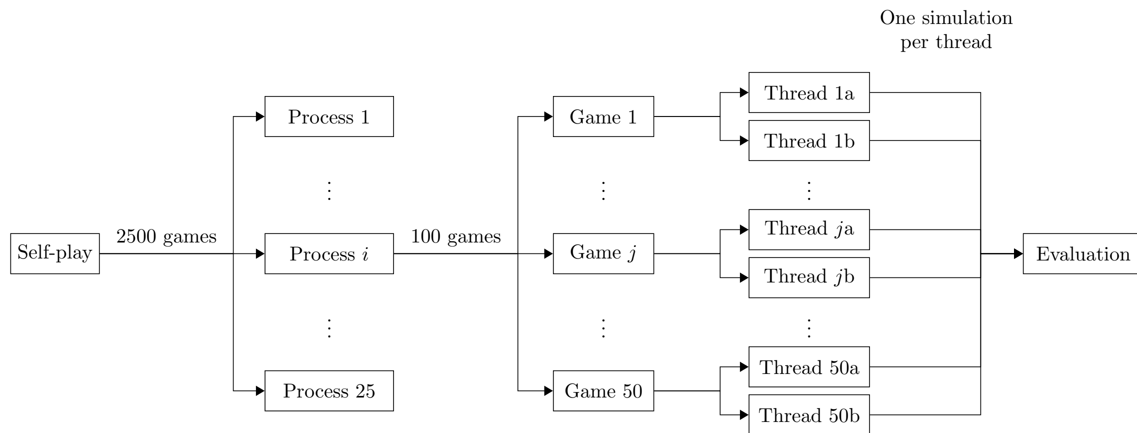

Figure A.1: Multiprocessing mechanism in tournament and self-play. We use self-play in this example. The workload of generating 2500 games is distributed over 25 processes, each of which only needs to generate 100 games. Each process plays 50 games simultaneously, which means it takes the process 2 rounds to generate 100 games. Each game spawns 2 threads to execute MCTS. Once all the $50 \times 2 = 100$ threads finish one simulation, all the newly expanded leaf nodes will be evaluated by the deep network.

A.2 Optimization

The training data $(s, \pi, z, a_{\text{seq}})$ has to be preprocessed before they can be used for the optimization. This preprocessing step actually takes much longer time than the parameter optimization step. To reduce preprocessing time, we sample all 500 mini-batches of training data before parameter optimization. We spawn 25 processes and divide the workload among them. These processes run on the server in parallel. Once all processes complete their tasks, we continue with parameter optimization.

Bibliography

[ATB17] T. Anthony, Z. Tian, D. Barber. "Thinking Fast and Slow with Deep Learning and Tree Search". In: *CoRR* abs/1705.08439 (2017). arXiv: 1705.08439. URL: http://arxiv.org/abs/1705.08439 (cit. on pp. 19, 21).

[Bel57] R. Bellman. *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press, 1957. URL: http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false (cit. on p. 15).

[Cou06] R. Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *5th International Conference on Computer and Games*. Ed. by P. Ciancarini, H. J. van den Herik. Turin, Italy, May 2006. URL: https://hal.inria.fr/inria-00116992 (cit. on p. 16).

[FRIW17] G. Farquhar, T. Rocktäschel, M. Igl, S. Whiteson. "TreeQN and ATreeC: Differentiable Tree Planning for Deep Reinforcement Learning". In: *CoRR* abs/1710.11417 (2017). arXiv: 1710.11417. URL: http://arxiv.org/abs/1710.11417 (cit. on p. 20).

[GDL+17] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, J. Malik. "Cognitive Mapping and Planning for Visual Navigation". In: *CoRR* abs/1702.03920 (2017). arXiv: 1702.03920. URL: http://arxiv.org/abs/1702.03920 (cit. on p. 19).

[HS97] S. Hochreiter, J. Schmidhuber. "Long Short-term Memory". In: *Neural Comput.* 9.9 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735 (cit. on p. 14).

[HZRS15] K. He, X. Zhang, S. Ren, J. Sun. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (cit. on p. 13).

[KHL17] P. Karkus, D. Hsu, W. S. Lee. "QMDP-Net: Deep Learning for Planning under Partial Observability". In: *CoRR* abs/1703.06692 (2017). arXiv: 1703.06692. URL: http://arxiv.org/abs/1703.06692 (cit. on p. 19).

[KS06] L. Kocsis, C. Szepesvári. "Bandit Based Monte-carlo Planning". In: *Proceedings of the 17th European Conference on Machine Learning*. ECML'06. Berlin, Germany: Springer-Verlag, 2006, pp. 282–293. ISBN: 3-540-45375-X, 978-3-540-45375-8. DOI: 10.1007/11871842_29. URL: http://dx.doi.org/10.1007/11871842_29 (cit. on p. 17).

[MKS+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236 (cit. on p. 20).

[NCG+17] S. Niu, S. Chen, H. Guo, C. Targonski, M. C. Smith, J. Kovacevic. "Generalized Value Iteration Networks: Life Beyond Lattices". In: *CoRR* abs/1706.02416 (2017). arXiv: 1706.02416. URL: http://arxiv.org/abs/1706.02416 (cit. on p. 19).

[OSL17] J. Oh, S. Singh, H. Lee. "Value Prediction Network". In: *CoRR* abs/1707.03497 (2017). arXiv: 1707.03497. URL: http://arxiv.org/abs/1707.03497 (cit. on pp. 11, 20, 21).

[Ros11] C. D. Rosin. "Multi-armed bandits with episode context." In: *Ann. Math. Artif. Intell.* 61.3 (2011), pp. 203–230. URL: http://dblp.uni-trier.de/db/journals/amai/amai61.html#Rosin11 (cit. on p. 21).

[SCW+15] X. Shi, Z. Chen, H. Wang, D. Yeung, W. Wong, W. Woo. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: *CoRR* abs/1506.04214 (2015). arXiv: 1506.04214. URL: http://arxiv.org/abs/1506.04214 (cit. on p. 31).

[SHS+17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, D. Hassabis. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815 (cit. on p. 22).

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–. URL: http://dx.doi.org/10.1038/nature24270 (cit. on pp. 11, 21, 22).

[TLA16] A. Tamar, S. Levine, P. Abbeel. "Value Iteration Networks". In: *CoRR* abs/1602.02867 (2016). arXiv: 1602.02867. URL: http://arxiv.org/abs/1602.02867 (cit. on p. 19).

[WRR+17] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Silver, D. Wierstra. "Imagination-Augmented Agents for Deep Reinforcement Learning". In: *CoRR* abs/1707.06203 (2017). arXiv: 1707.06203. URL: http://arxiv.org/abs/1707.06203 (cit. on pp. 22, 23).

All links were last followed on May 7, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature