THE UNIVERSITY OF MELBOURNE

# Learning From Game Screens Before Playing Atari Games

by
Hangyu Xia 802971

A thesis submitted for the degree of Master of Computer Science
in
The University of Melbourne

October 2018

# Declaration of Authorship

I, AUTHOR Hangyu Xia, declare that this thesis titled, 'Learning From Game Screens BeforePlaying Atari Games' and the work presented in it are my own. I certify that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- Where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the School.
- The thesis is 15502 words in length (excluding text in images, table, bibliographies and appendices).

Signed: HANGYU XIA

Date: 29/10/2018

# Contents

# *Abstract*

This thesis focuses on the ways of applying game screen features to an algorithm named Iterated Width (IW) search on the Atari 2600 game. The IW algorithm is first introduced by Nir et. al. in 2015 and performs at the level of the UCT algorithm which is one of the state-of-the-art planning algorithms on the Atari 2600 game with the game RAMs. However, the game RAMs are highly structured data of the Atari game that capture all the important game information. It is not easy to obtain such data for other problems similar to the Atari game. Therefore we hope to find a way to apply screen pixels, which is more general than the game RAMs, to the IW algorithm so that it can be used for more purposes. We have tried two general ways to process screen pixels, the first one is based on deep learning method and the second one is based on general image processing algorithm. Unfortunately, only the second one finally worked on our problem.

# List of Figures

# Chapter 1

# Introduction

*Atari 2600 Video Games* were first released on September 11, 1977 in North America. Now there are hundreds of games varying from sport games like Pong and Tennis to shooting games like Alien, Seaquest and so on. In order to get high scores in these games, players need to prevent making actions that will end the game early and try to score as much as they can. Human players without practice will find these games hard to play. In the domain of game planning, a game agent is a program that can control the game and try to get scores comparative to or even better than human players. For this purpose, game agents are expected to know what to do for given situations or know the best actions to take for given game states. Their performances are measured by the final scores achieved in games.

The most common way of designing a game agent is called domain-specific game planning [1]. Domain-specific here means having some pre-knowledge of a game and a specific strategy is designed based on this knowledge to instruct the agent to play the game. This kind of agents are often called domain-specific agents. It is very straightforward to design a domain-specific agent. The basic idea is simply listing all the cases the agent will encounter and hard-coding the actions to be taken to the agent. As we can image it is very efficient and effective if the game rules are clear and simple, but if the games rules are too complex to be characterized it can be quite hard to explore all the action plans beforehand. What is more, in domain-specific game planning an agent is exclusive to one specific game. We need to design different algorithms for different games, which is quite troublesome.

An alternative way is to develop algorithms or methods that have some generalities in solving multiple tasks without domain-specific designing. Agents of this property are called domain-independent/general agents [2]. General game agents can be applied to different games with very few changes and only one algorithm is needed for a bunch of games. Currently, the **Reinforcement Learning (RL)** [3] is the most popular method in designing general game agents. **Deep Q Learning (DQN)** [4], for example, is a deep learning based RL method. Agents based on RL methods can improve their performance by playing a game again and again. The more they play the better performance they will have.

Apart from the RL methods, tree search algorithms like the **Upper Confidence Bound for Trees (UCT)** [5] a member of the family of **Monte Carlo Tree Search (MCTS)** [6] and the **Iterated Width (IW)** Search [7] also have good generality and performance on planning on Atari games.

IW is a breath-first search algorithm but combines the scope of the blind search and the heuristic search. The IW agent designed by Nir, Miquel, and Geffner in 2015 had been only tested with the **RAM features** that contain game information like remaining lives, object locations, scores and so on. Even if IW is agnostic about which Byte contains this information. It has been shown that a variation of IW can work with game screen pixels [Bandres, Bonet and Geffner AAAI -18]. In this work, we explore further simple image processing methods that also work very well.

The image processing methods can be divided into two groups based on the AI technologies used. Methods in the first group are deep learning models like the **Autoencoder (AE)** [8], the **Convolutional Neural Network (CNN)** [9] and the **Recurrent Neural Network (RNN)** [10]. Methods in the second group are based on screen feature construction algorithms like the **Basic** method [11] and the **Basic Pairwise Relative Offsets in Space (B-PROS)** method [12]. The problem of using the B-PROS method is the computation is quite inefficient because the size of the B-PROS feature set can be extremely large. To make it easier, we simplified the B-PROS features by replacing the basic pairwise relative offsets between screen objects with their Manhattan distances. We call this simplified B-PROS method the **Manhattan B-PROS** method. The price of using the Manhattan B-PROS method is the lack of the relative direction between objects and the overlapping between objects with the same distances. This makes the Manhattan

B-PROS features extracted from a game screen no longer unique to the screen but representative to multiple screens. To tackle this problem a little bit, we adopted the idea of using the **e-features** that consider node depth in the IW algorithm from an algorithm named **Rollout IW** algorithm [13].

# Chapter 2

# Literature Review

## 2.1 Reinforcement Learning (RL)

### 2.1.1 Concepts

Reinforcement learning is simultaneously a problem, a class of solution methods that work well on the problem and the field that studies this problem and its solution methods [14]. To understand what is RL, the distinction between RL problems and RL solution methods is critical.

In reinforcement learning problems, a learning agent interacts over time with an unknown environment to maximize a "reward" signal which represents its performance. The agent must be able to sense the state of the environment and take actions to affect the state. Take the Atari game planning problem as an example, the game agent can sense the environment states like the game screens, game RAMs and the rewards. Actions like Fire, Up, Down etc. can be applied to games to affect game screens and game RAMs. Each time an action is applied to a game state, which is an interaction with the environment, a numeric reward will be observed by the agent to indicate how good this action is in an immediate sense. The goal of the agent is to maximize a reward signal which is the total reward it received over the long run.

These problems are often formalized as a **Markov Decision Process (MDP)** [15]. MDP is a sequential decision making process where each state is only depended on its previous

state and the action been taken by this previous state. Solution of such problems can be a look-up table or a complex strategy that can be used to find a sequence of actions to maximize the reward signal.

Reinforcement learning solution methods are algorithms or systems where learning happens. If we think about the nature of learning, we can easily find that interacting with our environment and knowing how the environment (the state) changes are always the basic. We need to have the idea of how good it is now and how promising it is in the future. Based on this information, we can try to find a good action to take in any situation. This is how learning is conducted through our lives. Reinforcement learning just has the same idea of learning through experience. It is learning what to do, how to map situations to actions so as to maximize the numerical reward signal [14] in the long run. The key thing we need to do to the the learner is acting as a teacher and telling apart the good and bad signal.

To achieve this, the reinforcement learning characterizes the learning process as three main components: a **state-action transition function**, a **state-action reward function**, a **value function**. The state-action transition function takes a state and an action to generate the next state. The state-action reward function calculates the reward when an action is taken by a state. The value function indicates how good a state is in the long run with a numeric value. For example, the value can be the maximum accumulated reward it can get in the rest of the game. Given a problem, these components are what a reinforcement learning method tries to learn. More specifically, in model-based RL the first two components are already known while in model-free RL we have no access to the transition and the reward function and we can learn the value function only through experience interacting with the world.

### 2.1.2 Planning on Atari Games

Planning on Atari games with a game emulator can be either model based or model free. In the Atari game planning problem, we treat it as a model free RL problem and use a game emulator to generate states and rewards. The emulator itself encodes a state-action transition function and a state-action reward function. The only thing needs to be considered is the value function. Thus in this work, we treated it as a model-free RL problem.

Since we are working with a game emulator, the state-action transition $F$ and the state-action reward function $R$ are already known. At time t, the game agent has a game state $s_t \in S$ where $S$ is the state space of the game. An action $a_t$ is selected from the action set $A$ available to this game and is applied to $s_t$ to generate the next game state $s_{t+1} \in S$. Mention that actions in Atari games are all deterministic, so $s_{t+1}$ is simply equal to $F(s_t, a_t)$ with $Pr(s_{t+1}|s_t, a_t) = 1$. The state-action reward $r_{t+1}$ which is the game reward at time $t+1$ is calculated from $R(s_t, a_t)$. If we have unlimited computational space and time budget, simply using a breadth-first search algorithm can actually work very well and finds the optimal solution by looking into all the possible cases into the future. Unfortunately, the truth is that the heavy burden on the computation and space makes it impossible to solve the problem in this way.

Instead of looking into all future states to find an optimal solution, reinforcement learning methods first learns a value function $Q(s, a)$ to map each state-action pair $(s, a)$ to a value $v$ where $s \in S, a \in A$ and $v$ is $Q(s, a)$. The learner plays the game hundreds of times or even more to update the value function based on the game experiences. To be more specific, the value of a $(s, a)$ pair is often the highest accumulated reward observed. This is basically what we referred as the learning process. There are many ways to learn a value function. For example, the **Temporal Difference (TD)** [16] used in **Q-learning** [17].

Note that the state-action value function is often called the **Q function**. Sometimes the dimensionality of a Q function can be very large if the action space is big. This can make it quite hard to get enough $(s, a)$ samples to learn a reliable Q function. So very commonly we use the **V function** rather than the Q function in the learning process. The V function is merely on the level of state but not the pair of state and action. For example, a $V_s$ can be the average of all the $Q_{s,a}$ where $a \in A(s)$. The best action chosen by the V function is the one leading to the next state with the highest value.

## 2.2 The Arcade Learning Environment

**Arcade Learning Environment (ALE)**: both a challenge problem and a platform and methodology for evaluating the development of general, domain-independent AI technology [11]. ALE is built on top of Stella, an emulator of the Atari 2600 games written in

C++ programming language. By given the emulator a state and an action, the emulator generates the following state and a reward.

ALE goes further and wraps around Stella core to provide its users a class named alestate (the game state) that can be saved and loaded by the emulator to control the game environment. Apart from this, the users can also get access to the game RAMs (highly structured game data) and the game screens in ALE. Available actions are grouped in a vector with each element a type of action. To make it more clear, ALE defined a class named TreeNode to store all information at a game step. The two most important members of a Treenode are the available action set and the ALE game state. The action set is a vector with at most 18 actions (Up, Down, Left, Right, Fire etc.) and differs from game to game as some actions are not used in some games. The ALE game state is an instance of alestate that stores all information including game screen and game RAM of a game step. Together with an action and an alestate, the users can build their state value function.

In Atari games, game RAM is a highly structured vector of 128 bytes (1024-bit) in the game memory. Important game features like lives, current score, object locations are contained in the game RAM. In fact the state value is more directly related to the RAM-action pair than the alestate-action pair. Although RAM data is more powerful in planning on Atari games, it is less representative in other tasks thus is still not a good way to evaluate the generality of our agent. As a result, we turned to the more general screen data to develop IW algorithm.

A single game screen of Atari 2600 games is a 2D array of 210 pixels high by 160 pixels width with each pixel a 8-bit value. In fact, ALE has two formats of screen pixels for its users. The first one is RGB, and the second one is ALE palette. RGB screens has tree channels for each pixel and each channel ranges from 0 to 128. Palette screens has just one channel, and the channel ranges from 0 to 255. In this paper, we use the palette screens to represent game screens which is different other papers. We do not want to do gray scaling or other transitions. All we need is just one value for each pixel. So the palette screen is enough for us. Figure 2.1 shows two screenshots of Atari game Alien and Space Invaders.

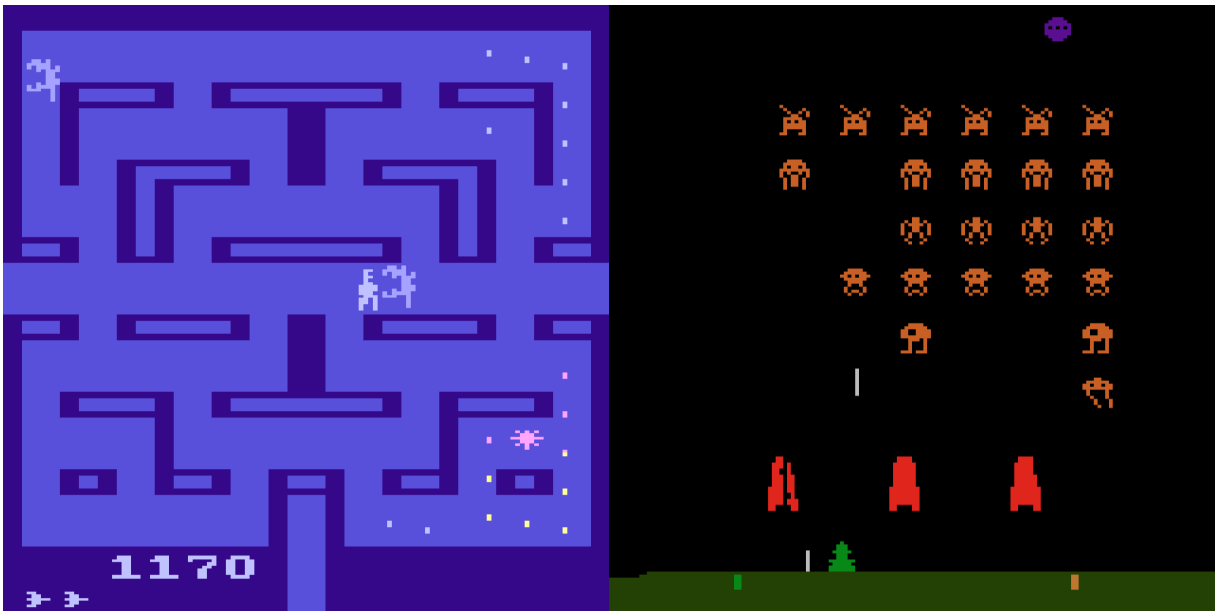



FIGURE 2.1: Screenshots of Alien and Space Invaders

The code of Arcade Learning Environment can be found in
https://github.com/mgbellemare/Arcade-Learning-Environment

## 2.3 Tree Search Algorithms

In this section, we are going to talk about how game agents plan on Atari games using tree search algorithms like MCTS and IW. MCTS is a set of search algorithms that are efficient in searching in large space. IW is a completely blind search algorithm where planning decisions are made by merely looking into future states that are meaningful to IW algorithm. Different from breath-first search algorithm, IW algorithm can search very deep into a search tree with a limited computational budget. Both MCTS and IW can provide quite good solutions on playing Atari games.

### 2.3.1 UCT: A Member of MCTS

**Monte Carlo Tree Search (MCTS).** A heuristic search algorithm for decision making processes. It is commonly used in playing games like computer Go and Atari video games.

Searching in MCTS is based on the analysis of the most promising game state, a trade-off of exploitation and exploration. The exploitation means choosing an action leading to the state with the highest value which is thought to be the best action, and the exploration means choosing an action in random. In MCTS, a game will be played out many times to expand the search tree where tree nodes have information like state values and edges correspond to actions. In each playout round, the game starts from the current game state and played out to the end by taking actions randomly. There are four steps in each playout round:

- *Selection*: Starting from the root node which is the current game state, a *tree policy* based on state values is applied to traverse the search tree until a leaf node $L$ is reached. Here a leaf node is any node that has not been visited or played out yet. It is a completely new node in the search tree.

- *Expansion*: If the node $L$ ends the game, nothing to do in this step. If not, create one or more child nodes for $L$ and randomly choose one child $C$ from them.

- *Simulation*: This step is called playout or rollout. Actions are selected from a *rollout policy* to simulate a complete game episode from node C. The result of this episode trail can be a game score or whether win, lose or draw in games like chess.

- *Backpropagation*: Result of the playout is used to update node values back from $C$ to the root node. After this, all nodes below $C$ which are created in the playout will be discarded.

The details of solving a planning problem with MCTS methods is like this. From the very start where no playout has been tried, the search tree has just one node , the root node which is the initial game state. Then the four steps in the playout are continuously executed from the root node until some computational budgets are reached, the number of playout rounds for example. Finally, an action from the root node is selected to be the best action according to some mechanism like the tree policy. After the game transitions to a new state, the root node will move to the subtree corresponding to the best action. At the same time the other subtrees will be discarded. MCTS keeps doing this for every root node until the root node reaches a goal or the end of a game. The set of the best actions chosen by MCTS is the finally solution of the planning problem.

The main difficulty of MCTS lies in the selection of child node which is the tree policy in the Selection step. It is the balance of the exploitation and the exploration also known as the exploration-exploitation dilemma in multi-armed bandit allocation. As what we have mentioned above, the exploitation tends to choose the best action which leads to a state with the highest value, and the exploration tries to randomly choose an action from available actions. As we can imagine, the exploitation will prefer a fixed action path in the Selection step because the Backpropagation will only make state values increasing through time. Imagine that if we always expand the subtree whose root node has the highest state value, this subtree will definitely be the best subtree forever as it improves each time after expansion. This can make our searching get trapped in a local optimal solution. The exploration however makes it possible to choose the action that is currently suboptimal or even bad but turn out to be optimal in the long run. This basically means the exploration has the ability to prevent us from getting trapped in the local optimal solution. Despite of this, the exploration is not a reliable way to select actions because the randomization makes it even hard to find a local optimal solution most of the time.

One way to tackle the exploration-exploitation dilemma is using the $\epsilon - Greedy$ algorithm [18]. Each time the tree policy selects an action, the best action will be selected with a probability of $1 - \epsilon$ where $\epsilon$ is a predefined constant, and a random action will be selected with a probability of $\epsilon$.

Though the $\epsilon - Greedy$ algorithm has been widely used in multi-armed bandit problems, but in the Atari game planning problem it is too simple to have good performances. Fortunately, an algorithm named UCT, a member of the MCTS, has been proved to work quite well on our problem.

**Upper Confidence Bound for Trees (UCT).** UCT is a member of the MCTS family [5]. Searching in the UCT algorithm is exactly the same as what we have discussed in the MCTS. What make the UCT algorithm special are the rollout policy and the tree policy it used in each playout round. In the UCT algorithm, the rollout policy simply chooses an action in random. The tree policy addresses the exploration-exploitation dilemma with a balanced exploitation-exploration strategy which is a function similar to the V function we talked in 2.1.2. Actions are not considered here because we just need an evaluation of the general performance of a node (a game state) in the future. According to the UCT

tree policy, the best child node j is selected to maximize:

$$V(j) = \bar{X}_j + C\sqrt{\frac{2\ln n}{n_j}} \tag{2.1}$$

$V(j)$ is the value of node $j$ where $j$ is the child node of the selected node (start from the root node). $\bar{X}_j$ is the average value of node $j$ discovered till now which guides the exploitation. $C\sqrt{\frac{2\ln n}{n_j}}$ is used to guide the exploration where $C$ is a constant and $n$ is the number of times the parent of node $j$ has been visited. $n_j$ is the number of times node $j$ has been visited. It is easy to see that UCT tree policy prefers nodes with a higher expected value (larger $\bar{X}_j$) or have rarely been seen now (larger $C\sqrt{\frac{2\ln n}{n_j}}$). Here are the details of a single playout round in UCT:

- *Selection*: Start from the root node, recursively select a node from child node with the UCT tree policy until a leaf node is reached. A leaf node is a node not in the terminal state and has unvisited child nodes according to untried actions.
- *Expansion*: An action is randomly selected from untried actions of the selected leaf node to create a new child node. The search tree is expanded with this new child node and the chosen action.
- *Simulation*: This step is where the agent interacts with game emulator, or we can say the environment, to play out from the game. The game emulator takes a state and an action to generate a new state with a reward. This is what we referred as the "state-action transition" and the "state-action reward function" in 2.1.1. Start from the newly expanded child node, random actions are taken by the emulator to descend the search tree until a terminal state is reached.
- *Backpropagation*: In this step, the final reward for the terminal state will propagate back from the new child node to the root node to update the search tree.

Pseudo code of the UCT algorithm is showed in Algorithm 1.

**Algorithm 1** The UCT Algorithm

**function** UCTSEARCH($v_{root}$)
  **while** within computational budget **do**
    $v_{leaf} \leftarrow Select(v_{root})$
    $v_{child} \leftarrow Expand(v_{leaf})$
    $r_{final} \leftarrow Emulate(v_{child})$
    $Backpropagation(v_{leaf},\ r_{final})$
  **return** $BestChild(v_{root},\ C)$

**function** SELECT($v$)
  **while** $v$ not terminal **do**
    **if** $v$ has unvisited child **then**
      **return** $v$
    **else**
      $v \leftarrow BestChild(v,\ C)$
  **return** $v$

**function** EXPAND($v_{parent}$)
  randomly choose $a \in v_{parent}.untried\_actions$
  create a new tree node $v_{child}$ with:
    $v_{child}.parent = v_{parent}$
    $v_{child}.N = 0$
    $v_{child}.V = 0$
    $v_{parent}.child[a] = v_{child}$
  remove $a$ from $v_{parent}.untried\_actions$
  **return** $v_{child}$

**function** BESTCHILD($v,\ C$)
  **return** $\text{argmax}_{v' \in v.child} \frac{v'.V}{v'.N} + C\sqrt{\frac{2 \ln v.N}{v'.N}}$ ▷ $v'.V$ is the total reward played out from $v'$, $v'.N$ is the number of times $v'$ has been visited

**function** EMULATE($v_{parent}$)
  $s \leftarrow v_{parent}.s$
  **while** $s$ is non-ternimal **do**
    Randomly choose $a \leftarrow s.actions$
    $s,\ r \leftarrow CallEmulator(s,\ a)$ ▷ CallEmulator is a function offered by ALE
  **return** $r$

36: **function** BACKPROPAGATION($v$, $r_{final}$)
37: **while** $v$ not null **do**
38: $v.N \leftarrow v.N + 1$
39: $v.V \leftarrow v.V + r_{final}$
40: $v \leftarrow v.parent$

Figure 2.2 shows how the UCT search tree is updated in one single playout round. Each node $v$ has a tuple of values where the first value is $V(v)$ and the second value is $N(v)$.

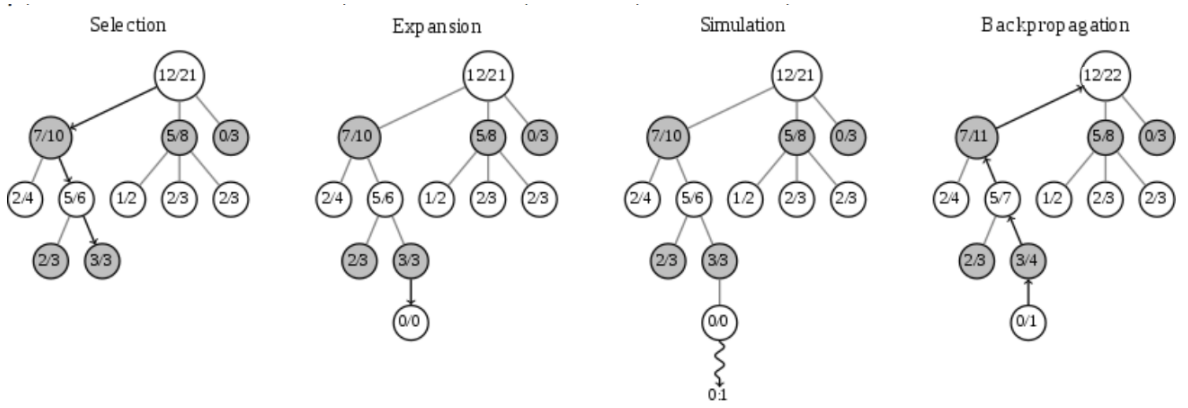



FIGURE 2.2: UCT Example [1]

### 2.3.2 Iterated Width (IW) Search

In Atari game planning problems there are 18 actions applicable in each state, which requires a computational budget of more than 1 million nodes to look just 5 steps ahead. What is more, it is often the case that non-zero rewards can only be observed after a look ahead of over 20 steps. This makes the problem nearly impossible to be solved with basic blind search methods in a reasonable amount of time. As a result, we turned to *Iterated Width* (IW) [19] algorithm that is anticipated to run in time exponential in the problem width. Like blind search algorithms, IW does not require any prior knowledge like the state transitions or goals of the planning problems. At the same time, IW is also like heuristic algorithms that can search in large problem space effectively. What is more, as

[1] Proposed in https://en.wikipedia.org/wiki/MonteCarlotreesearch

IW has been described as a classical planning algorithm rather than a learning algorithm, there is no need for IW to learn a state-action value function like what UCT does.

When planning on Atari games, IW states are represented with game RAMs where each state $s$ is associated with 128 multi-valued atoms $X_{i,\ i\in\{1,..,128\}}$ with a domain of 0 to 255 (8-bit). A little different from UCT algorithm, there are just 3 steps in each search iteration in IW: Selection, Expansion and Simulation, and the Backpropagation step is applied only after the tree has been expanded. Recall that in the UCT algorithm, we balance the exploitation-exploration with a state value function $V(s)$ to select an expandable node in the selection step. In the expansion step, one action is randomly selected from unvisited actions of the selected node to generate a new child node. However, in the IW algorithm the selection step selects a node simply by popping out a node from a breadth-first queue managed by the expansion step. All observed expandable nodes during the search are pushed to this queue following the fifo (first in first out) policy. The expansion step expands all actions that are available to the expandable node instead of just one and adds child nodes to the queue according to a notion named state novelty.

**Novelty.** The *novelty* of a state is a value that evaluates how novel an observed state is from current search experience. It is calculated right after a state is generated and will not change in the future. The novelty of a state should not be influenced by the future experience. To understand the idea of using novelty in a search algorithm, there are two notions we should know.

- *state novelty*: the novelty of a state $s$ is the minimum number of its atom values that have been observed together for the first time in the current searching step. It is calculated right after a node is emulated. Each atom value of a state is labelled as $x_{i,\ i\in\{1,..,128\}}$. Namely, if a state has one single atom of which the value we have not seen before, the novelty of this state is 1. If not but the state has a pair of atoms that we have not seen before, then the novelty is 2. It is similar for calculating the novelty 3, 4 and more.
- *novelty test*: a step responsible for calculating the novelty of states. Novelty test maintains a structure named novelty table to store all visited single atoms, atom pairs and so on. An example of novelty test will be given in IW(1) algorithm later.

**IW(1).** A notion of IW(k) is used to define the level of the novelty test. Only states with exactly novelty k will be kept in the tree or else will be pruned. It is hard to say that which IW(k) is the best because each of them treats the state space in a different way. In this work, we use IW(1) as our searching algorithm because the performance of IW(1) has been shown comparative to UCT algorithm in Atari game planning problem [7].

In IW(1), the *noveltytable* is a set of 128 vectors with each one of 256 boolean values. Each vector is related to an atom of the RAM state, and the 256 boolean values stand for the occurrence of the value of this atom (from 0 to 255). Basically, the novelty table can describe any game state by setting each atom vector to a one-hot vector with only the position of the atom value true. So it can be seen as a description of the state space. More importantly, the novelty table is mainly used for storing search experience. It will be updated each time a new state is observed. Despite that IW manages to gain some experience with the help of the novelty table, it is an exploitation method because node rewards are not considered during the search.

IW(1) algorithm first initializes the novelty table by setting all atom booleans to zero before the first search iteration starts. This means our agent has no search experience when no state has been observed. In the searching part, IW simply uses the breadth-first search to expand the tree with three steps in each search iteration.

- *Selection*: different from the UCT algorithm where an expandable node is selected by recursively by descending through the tree, the expandable node in the IW algorithm is the first node in the breath-first queue $q$ which is managed in the Expansion step. What should be mentioned is that in the first iteration there is just one root node in the queue so the expandable node is the root node.
- *Expansion*: after an expandable node has been selected, it will be fully expanded with all child nodes generated and added to the tree. Simulation step is then called for each action to generate a state and a reward for corresponding child node. Child nodes with novelty at least 1 will be added to the breath-first queue to wait for expansion while the rest are pruned.
- *Simulation*: takes a state and an action as inputs to generate a new state and a reward.

The last step of IW algorithm is the *Backpropagation* step. It is called iteratively to update the accumulated reward for each subtree of the root node. The best action for the root node is the one leading to the subtree with the highest accumulated reward.

IW(1) can also use a strategy called $stop\_on\_first\_reward$ in the Selection step to make the tree expansion more balanced in the perspective of scoring. The basic idea is that nodes with positive reward will not be expanded right away but added to a queue named *pivots* and wait for expansion until no node passes the novelty test. Pseudocode of the IW(1) algorithm has been shown in Algorithm 2.

**Algorithm 2** The IW(1) Algorithm

```
1:  function IW1Search(v_root)
2:      initialize novelty table
3:      pivots.push(v_root)
4:      while within computational budget and pivots not empty do
5:          curr_node ← pivots.pop()
6:          q.push(curr_node)
7:          while q not empty do
8:              v_expandable ← Select()
9:              Expand(v_expanable)
10:     Backpropagation(v_root)
11:     return BestChild(v_root)
12:
13: function Select
14:     return q.pop()
15:
16: function Expand(v_parent)
17:     for action a ∈ v_parent.available_actions do
18:         create a new tree node v_child with:
19:             v_child.parent = v_parent
20:             v_parent.child[a] = v_child
21:         Emulate(v_parent, a)
22:         child_novelty ← Novelty_test(v_child.state)
23:         if child_novelty >= 1 then
24:             q.push(v_child)
        return
25:
26: function Emulate(v_parent, a)
27:     v_chlid ← v_parent.child[a]
28:     v_child.state, v_child.r ← CallEmulator(v_parent.s, a)
29:     return
```

```
30: function NOVELTY_TEST(state)
31:     novelty ← 0
32:     for i in range(0, 128) do
33:         atom_value ← state.atom(i)
34:         if novelty_table.atom(i)[atom_value] is false then
35:             novelty_table.atom(i)[atom_value] ← true
36:             novelty ← novelty + 1
37:     return novelty
38:
39: function BACKPROPAGATION(v)
40:     if v is leaf node then
41:         v.accumulated_reward ← v.reward
42:         return
43:     for v_child ∈ v.child do
44:         Backpropagation(v_child)
45:     v.bestchild ← BestChild(v)
46:     v.accumulated_reward ← v.reward + v.bestchild.accumulated_reward
47:     return
48:
49: function BESTCHILD(v)
50:     return v_child ∈ v.child with the highest accumulated reward
```

### Problem

Sometimes there are several nodes that have the same highest accumulated rewards. What IW does in this case is to choose the node with the greatest branch depth (the deepest subtree) as the best child. As we can image, this will result in a search that prefers to look ahead as far as possible which sounds reasonable. However, when we tried to test the IW agent with a computational budget of 10000 nodes on games like Alien, Pacman and Space Invaders where a positive reward can be observed in about 5 steps look ahead, the object controlled by the agent tended to wander forward and backward several times before it achieved a score even when the score is right next to it. Another problem is that the agent can not prevent its object from death. Take the Space Invaders as an

example, our object in control shows no response when the monsters are approaching but stands still or heads to the monster most of the time. One reason of this is that the computational budget is too small. A computational budget of more than 1 million nodes will work much better but takes much longer time. Right now with a budget of 10000 nodes, our agent takes about 5 seconds to make one decision.

**Solution**

To solve the first problem that the object in control wanders so often in the games, we use a different but quite similar policy to select the best child. If the highest accumulated rewards are larger than 0, the branch with the shallowest depth will be chosen. This can make our agent more eager in scoring and thus less wandering before it can score. If the highest accumulated rewards are 0, the branch with the greatest depth will be chosen instead. The reason is that we hope to look far into the future when no score has been discovered so that we are closer to a state that can score as much as possible. With these two changes, our agent can easily find a path to score more while paying fewer efforts. Remind that this all depends on the efficiency and effectiveness of IW algorithm in the tree expansion. IW is efficient and effective in pruning nodes that are not meaningful and thus makes our agent more intelligent in finding a path promising in the future. Though this is a good solution for the first problem, it can make our object more easy to die and end the game very early.

To solve the second problem that the agent can not protect its object from death, we first need to know how ALE formats a negative reward. In ALE, a negative reward of -1 is used to represent the death of our object or losing a score to our opponent in sport games like Pong and Tennis. After knowing this, we prune the nodes that return a negative reward in the expansion step to avoid our agent choosing an action leading to these nodes. This does make things better but still not perfect. The reason is that in the ALE games a negative reward is not often observed right at the time our object dies but several frames later. To make it more clear, let's take the game Alien as an example. In the game Alien, we lose a life when our object the spaceman is caught by the monster. A negative reward of -1 will not be returned immediately the time our object encounters a monster. Several frames will be displayed then to show how the spaceman gets destroyed by the monster, and at this time the game is out of control which means the current round has already ended for the players. Only then will a negative reward been observed and a new round starts. This makes it significantly difficult to detect the actual death state and

thus difficult to perfectly protect our object from death. There is nothing we can do for this apart from changing the ALE source code to provide an accurate timing for negative rewards. In our work, there is no much need to do so as the solution mentioned above has already done a good job.

### 2.3.3 E-Feature Based IW(1) Algorithm

Different from the IW(1) algorithm where the state novelty is computed merely based on the novelty table, the e-feature based IW(1) algorithm maintains a depth table in addition to the novelty table to calculate the novelty of a state. The depth table has the same size with the novelty table. The number of state atoms to be specific. Basically, the depth table captures the minimum node depth that an atom value that has been observed. If a state has an atom of which the value is observed the first time, the novelty of this state is 1 and both the novelty table and the depth table will be updated. If a state has any atom value true in the novelty table but its depth is smaller than the minimum depth of the atom value kept by the depth table, we will say that this state also has the novelty 1 and only the depth table will be updated. This allows us to better use some of the states that are not novel for the IW(1) algorithm but are actually special in some sense. This is extremely useful when the tree is not balanced, which is exactly the case in the searching stopped on the first reward.

## 2.4 Image Preprocessing: Background subtraction

Planning directly on $210 \times 160$ pixel game screens with a 255 color palette can be computationally demanding. A total number of $210 \times 160 \times 255 = 8,568,000$ states are there in the state space, which is quite big. To make things easier, we subtract the background from the screens so that they are more sparse and only non-zero pixels representing game objects are kept. All feature extraction methods referred in this thesis are based on the game screens with background subtracted. Figure 2.3 has shown the screens of Freeway and Tennis after their background has been subtracted.

To compute the background of a game, we first run IW algorithm with RAM features 5000 frames on the game to get 5000 game screens. After this, the frequencies of each

pixel values will be computed from these screens. Lastly, each pixel of our background is represented with the value that has the highest frequency. For example, if 0 has appeared 4000 times in the first pixel of the 5000 game screens, and 127 appeared 1000 times. Then the first pixel of the background should be 0.

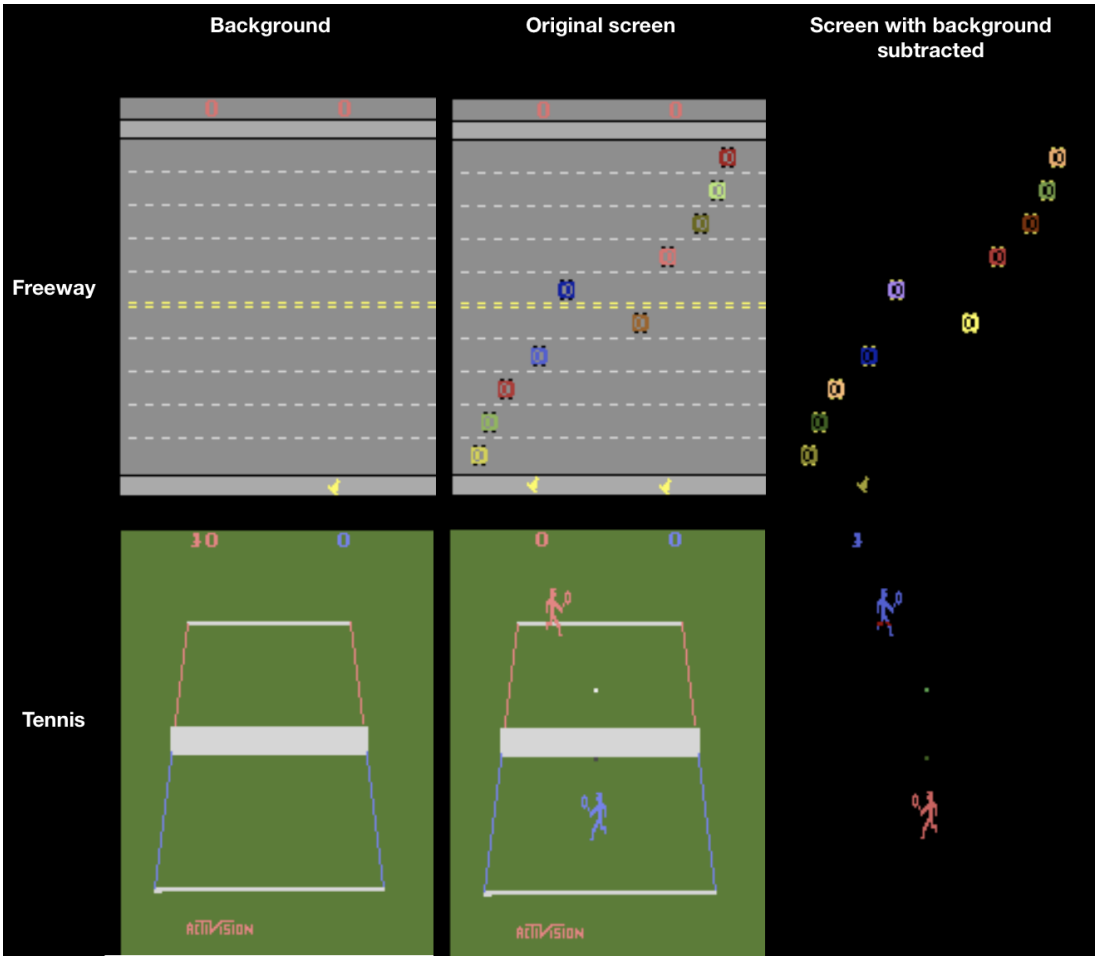



FIGURE 2.3: Background subtracted screenshots of Freeway and Tennis

# Chapter 3

# Learning Features From Game Screens

Till now, we have introduced the RAM features based IW algorithm in great details. From this chapter on, we are going to apply IW algorithm to screens features to see how it works on tasks that are more abstract. Instead of viewing each variable in the game RAM an atom, what we are going to do is characterizing the features learned from game screens as the atom set.

In this chapter, we will talk about four deep learning models we tried to learn features from game screens. Basically, all the models are Autoencoders (AE). Two of them are trained with action transformations and the other two are trained without action transformations. Since all of these models did not improve the agent performance, we will just talk about how we tried to build these models and why they do not work in this thesis. Our implementation is based on Tensorflow. All models are trained in python and loaded in C++ where the ALE environment and our agent are implemented.

## 3.1 Deep Learning Methods

### 3.1.1 Autoencoder (AE)

An autoencoder is a type of Artificial Neural Network that can be used to learn a representation for a set of data or reduce their dimensionality. This means apart from applying game screens directly to the IW algorithm, we can also try to learn a game screen representation that contains screen information more than just pixel values. In this sense, we trained several AE models on game screens to see if they work well with the IW algorithm.

Typically, an autoencoder is composed of two parts where each part is a Neural Network. The first Neural Network is often called the encoder which takes a set of data as its inputs and returns the representations for these data. The second Neural Network is called the decoder where the data representations got from the encoder are transformed back to its original data. In other words, we can view the encoder as a compressor and the decoder as a decompressor. A data can be compressed to a new format and then decompressed back to what it is originally. The new format of data can be seen as a representation of the original data as they can be transformed from one to the other. The general form of an autoencoder is shown in Figure 3.1.

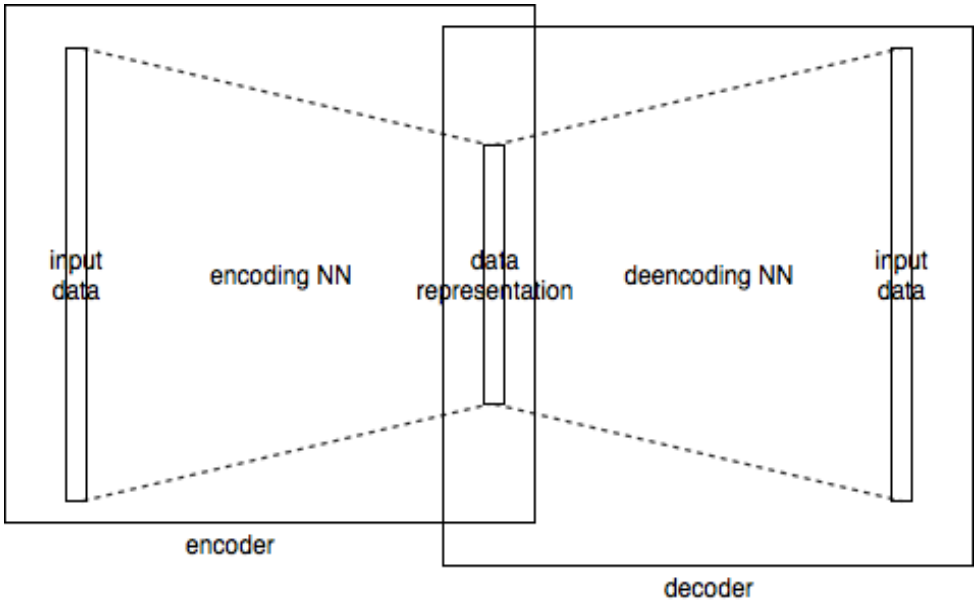



FIGURE 3.1: General form of an AE

We have tried to build four AE models based on this framework with two models a group and two groups in total. Models in the first group are trained without action transformation where only game screens and no actions are taken into consideration. Models in the second group are trained with action transformation where actions are also taken as part of the inputs. A little different from general AE model, models in the second group are anticipated to predict the next game screen with a sequence of previous game screens and an action. In each of the group, one model is based on the Convolutional Neural Network (CNN). The other one is based on the Recurrently Neural Network (RNN), the Long Short Term Memory (LSTM) to be specific.

### 3.1.2 Convolutional Neural Network (CNN)

CNN is primarily used to solve difficult image-driven pattern recognition tasks [20]. It has a different architecture from regular Neural Networks where every layer is made up of a set of neurons and fully connected to the layer before and after. The main building block of a CNN is called the convolutional layer.

In each convolutional layer, the convolution is applied on the input data with the use of multiple filters/kernels. The input and output data are organized in 3-dimensions in the format of (height, width, channels). The number of channels of the input data is the number of filters in the previous convolutional layer. The input data is the output of the previous convolutional layer. In the first convolution layer where there is no previous layers, images are taken directly as the inputs. The number of the input channels in this case can be 3 if the images are represented in RGB way: the red channel, the green channel and the blue channel. It can also be 1 if the images are gray scaled with each pixel a single value. The filters are 2-dimensional matrices and have the same size. Each of them is a mapping from the input data to the output data.

To make the convolution process more clear, let's first look at how a single mapping with just one filter is applied to the input data. First of all, all channels of the input data are convolved with the filter one by one to get filtered features for each channel. After this, all filtered channels are merged to one single channel by averaging or summing over the values in each position of the channels. By doing this, one filter can transform the input data with several channels to a new one with just one channel. Suppose we have 128

filters in a convolutional layer, it is easy to see that we can get 128 filtered channels that are all mapped from the input data at last. This is where the output data comes from with a shape of (height, width, 128) in this case.

A CNN can be simply viewed as a stack of convolutional layers. More layers are stacked, more general the features extracted will be. This does not mean that deeper CNN is always better because features that are too general will fail to keep some important information of the original images.

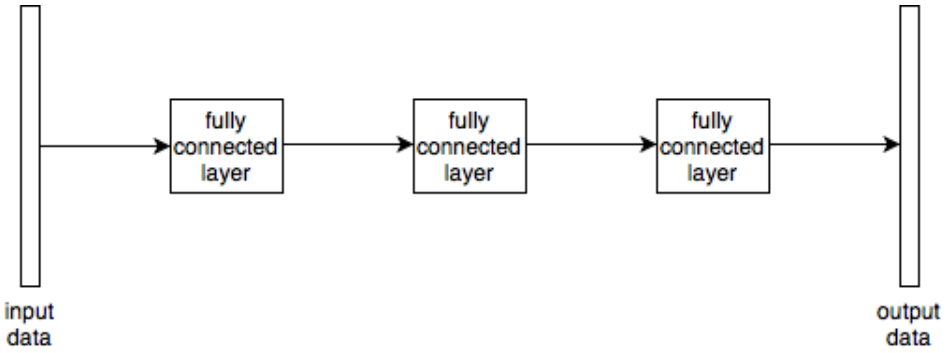
### 3.1.3 Recurrent Neural Network (RNN) : LSTM

In traditional Neural Networks, image processing methods like downsampling, denoising and feature extraction are based on single image. This basically means all of these methods take just one image as the input, and the features extracted are static to this image. However, if we are in a scenario like classifying the kind of event happening in a movie screenshot where the answer does not completely come from the current screenshot but is also related to previous screenshots, it can be extremely hard for an ordinary Artificial Neural Network to get the right answer. In such case, a Recurrent Neural Network is what can make things happen [21].

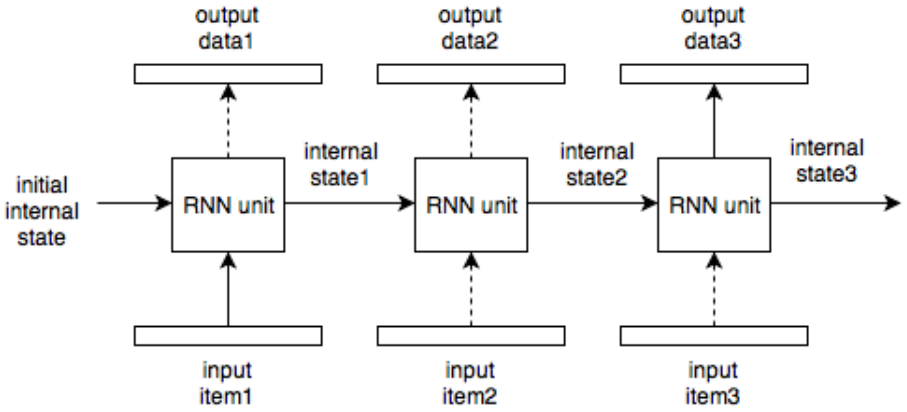
RNNs can exhibit dynamic behavior of a sequence of inputs. The main building block of a RNN is a structure called RNN unit, a general name of all kinds of recurrent unit used in RNNs. For example, the **Gated Recurrent Unit (GRU)** [22] cell and the **Long Short Term Memory (LSTM)** cell are two instances of the RNN unit. A RNN can be simply viewed as a stack of RNN units which looks very similar to a regular Neural Networks where fully connected layers are often stacked one on the top of another. However, RNNs are totally different from the regular Neural Networks because of the way RNN units perform. An intuition of the frameworks of Regular NNs and RNNs have been showed in Figure 3.2.

In RNNs, the input and the output can be either a single data or a sequence of data. For example, in the task of classifying the kind of event happening in a movie, the input can be a sequence of movie screenshots which are arranged by time and the output can be a one-hot event vector that indicates what the event is. Also, if we look at the task of speech recognition, the input can be a sequence of audio words arranged in the order of a

sentence. The output can be each of the corresponding word in characters. Each item in the input sequence is an input of the RNN unit. The RNN unit takes one item at a time and returns a result after all the items have been processed.




(a) Framework of regula NN




(b) Framework of RNN

FIGURE 3.2: Frameworks of Regular NN and RNN

What should be mentioned here is that there are three RNN units showed in Figure 3.2 but they are not three different RNN units but one single RNN unit been used three times with each one stands for a time step (3 time steps in total in this case). The internal state (often known as the hidden states) is some features calculated from the input item and accumulated through time to obtain dynamic behavior of these items for a time sequence. To be more precise, the internal state at a time step depends on the item of the current step and the internal state of the last step which depends on the items further before. This basically means the internal state at a time step is actually a representation of all the items earlier before, and thus somehow stands for the dynamic behavior of these items

through time. The output at each time step is computed based on the internal state and the current input item, which is kind of a summation of the previous knowledge.

The most well-known RNN is the LSTM which uses the LSTM cell as the RNN unit. The benefit of using LSTM is to avoid the vanishing gradient problem in training Artificial Neural Networks with with gradient-based learning methods like **Stochastic Gradient Descent (SGD)** [23], **Root Mean Square Propagation (RMSProp)** [24], **Adaptive Moment Estimation (Adam)** [25] and so on.

**Long Short Term Memory (LSTM).** A special kind of RNN composed of LSTM cells [26]. The internal state of a LSTM cell is constructed of two variables, one is called the **cell state** $C$ and the other is called the **hidden state** $h$. Figure 3.3 shows how a LSTM cell looks like. There are three inputs and three outputs in a LSTM cell. $X_t$ is the sequence item at time step $t$, $C_t$ and $h_t$ are the cell state and the hidden state at time step $t$ respectively.

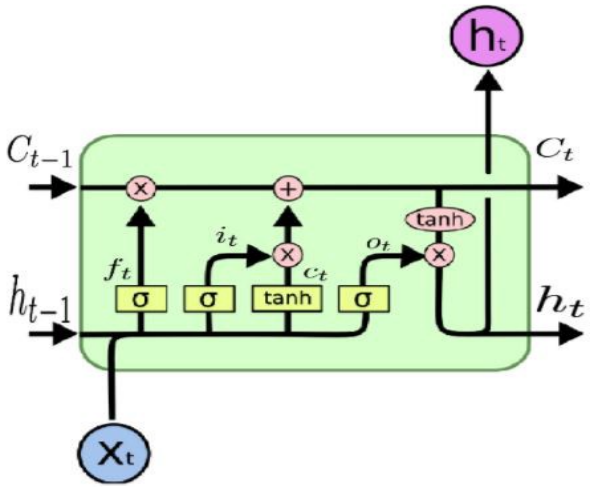
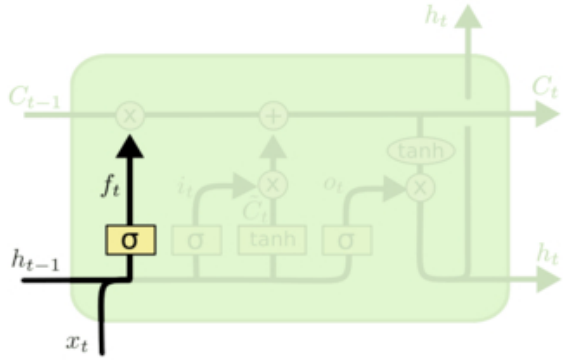



FIGURE 3.3: LSTM cell [1]

The cell state is the key to the LSTM and stores the information over arbitrary time intervals. The LSTM cells can remove or add information to regulate the cell state with structures called gate: a **forget gate**, an **input gate**, and an **output gate**.

### Forget Gate

[1] Proposed in http://colah.github.io/posts/2015-08-Understanding-LSTMs/



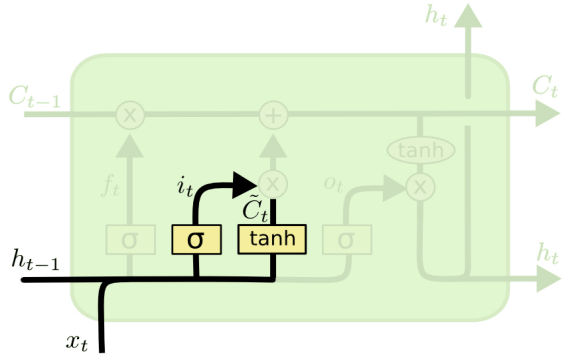

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

FIGURE 3.4: Forget gate [2]

The forget gate, showed in Figure 3.4, controls the information to be removed from the last cell state $C_{t-1}$. This decision is represented with the $f_t$ which is the result of a sigmoid function and has the same dimensions with the cell state $C$. Values in $f_t$ range from 0 to 1 and each of them is the weight for a number in $C_{t-1}$. Numbers in $C_{t-1}$ with weights equal to 1 will be completely kept while those with weights equal to 0 will be discarded.

$W_f \cdot [h_{t-1}, x_t]$ is a representation of $(W_{fh} \cdot h_{t-1} + W_{fx} \cdot x_t)$ where $W_f = [W_{fh},\ W_{fx}]$. $W_{fh}$ fully connects $h_{t-1}$ to the size of $C_{t-1}$ and $W_{fx}$ fully connects $x_t$ to the size of $C_{t-1}$. Then the results of these two fully connections are added together to generate $f_t$. We can see from here that the forget decision making is based on an analysis of the last hidden state $h_{t-1}$ and the current input $x_t$.

### Input Gate




$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

FIGURE 3.5: Input gate [3]

[2]Proposed in http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[3]Proposed in http://colah.github.io/posts/2015-08-Understanding-LSTMs/

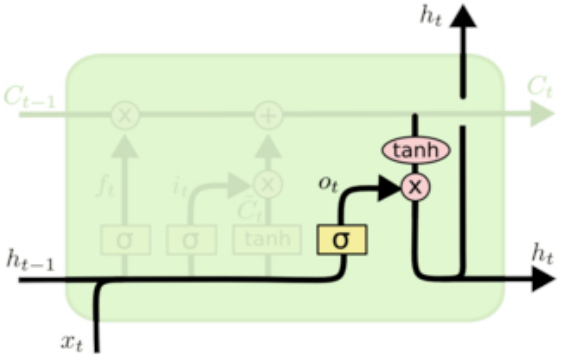
The input gate controls the information to be updated on the last cell state. This is done with the calculation of $i_t$ and $\tilde{C}_t$ showed in Figure 3.5. $\tilde{C}_t$ is the result of a tanh function where values range from -1 to 1. It is a candidate of calculating the $C_t$ as well as an estimate of the $C_t$. The idea of using $\tilde{C}_t$ to update $C_t$ is similar to the idea of using the Temporal Difference algorithm to update the value function in Q-learning. $i_t$ is a vector of weights for numbers in $\tilde{C}_t$ which is similar to $f_t$ but has a totally different meaning. In the forget gate, $f_t$ is used to throw away information from the previous cell state while in the input gate $i_t$ is used to decide the information kept in $\tilde{C}_t$. What finally got in this step is the product of $i_t$ and $\tilde{C}_t$ which is the information used to update $C_{t-1}$.

### Cell State updating

New cell state $C_t$ is computed with the results of the input gate and the output gate.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{3.1}$$

### Update Gate




$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

FIGURE 3.6: Update gate [4]

Till now, we have already updated the cell state. The only thing remained to do is updating the hidden state which is the output as well as the information passing to the next time step. This is showed in Figure 3.6.

$o_t$ is just like $f_t$ and $i_t$, a vector of weights range from 0 to 1. The updated hidden state $h_t$ is a filtered version of $C_t$. For example, for a language model the updated hidden state can be something like the possible tags of words coming next when only part of a sentence

[4]Proposed in http://colah.github.io/posts/2015-08-Understanding-LSTMs/

has been observed. It is obvious that the $h_t$ here is highly related to the $C_t$ which is the knowledge learned from the observed part of sentence. This is basically where the idea of using an update gate to control the outputs comes from.

## 3.2 Training Without Action Transformation

In this group, there are a CNN based AE model and a LSTM based AE model. Actions are not considered in both models. We can call them action-free models. Features are extracted by finding a screen representation that can be transformed back to its original game screen.

### 3.2.1 CNN Based AE

#### Network Architecture

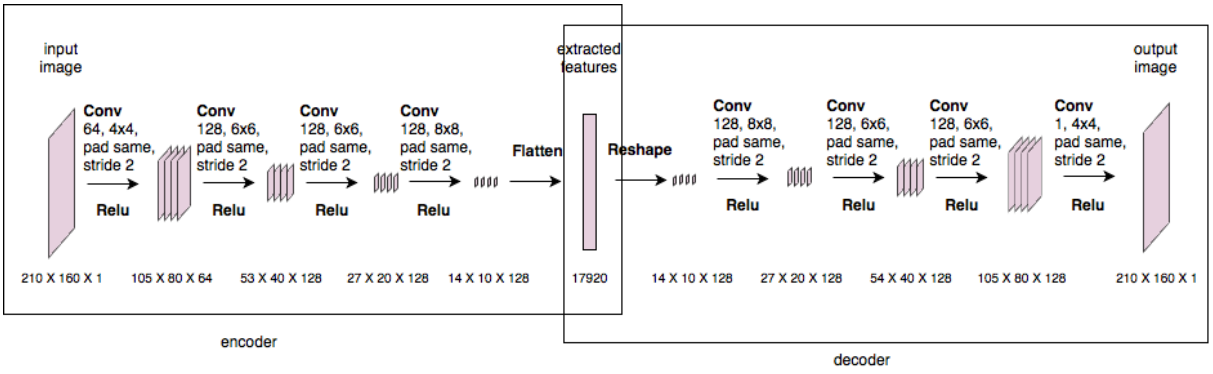



FIGURE 3.7: Architecture of the action-free CNN based AE

**The encoder** is a 4 layers convolutional network. The first layer has 64 filters with the filter size of $8 \times 8$ and the other three layers all have 128 filters of size $6 \times 6$, $6 \times 6$ and $4 \times 4$ respectively. Filtering strides for each layer are set to 2 to confine the size of the data representation to certain extent. The input of the encoder which is also the input of the AE model is a $210 \times 160 \times 1$ pixel matrix with just 1 tunnel. The output is a $14 \times 10 \times 128$ matrix with 128 tunnels. In this model, the output is the features we extracted from the input image.

**The decoder** is a 4 layers deconvolutional network, a reverse version of the convolutional network. It can turn a $14 \times 10 \times 128$ matrix back to $210 \times 160 \times 1$. The first two layers all have 128 filters with the filter size of $4 \times 4$, $6 \times 6$ respectively. The third layer has 64 filters of size $6 \times 6$ and the last layer has only 1 filter of size $8 \times 8$. The input of the decoder is the output of the encoder, which is the $14 \times 10 \times 128$ image features. Filtering strides for each deconvolutional layer are set to 2 to consist with the layers in the encoder. The output is a restoring of the orignal image, so it is a $210 \times 160 \times 1$ pixel matrix with 1 tunnel.

It is quite obvious that the size of the features (17,920 values) extracted from the encoder is much smaller than the size of the original image (33600 values). Every layer in the encoder and the decoder is followed by a rectified linear function [27] to remove negative values.

#### Training Details

For each game, the data set consists of 4480 training screens and 512 test screens sampled from running IW(1) algorithm with the RAM feature. All these screen data are preprocessed in two ways by subtracting the background pixels and dividing each pixel by 255. The training loss is the mean square error of the original image and the restored image. Adam is used with learning rate of 0.001 as the optimizer. The batch size for each iteration is 64. The number of iterations trained is 200.

### 3.2.2 LSTM Based AE Model

#### Network Architecture

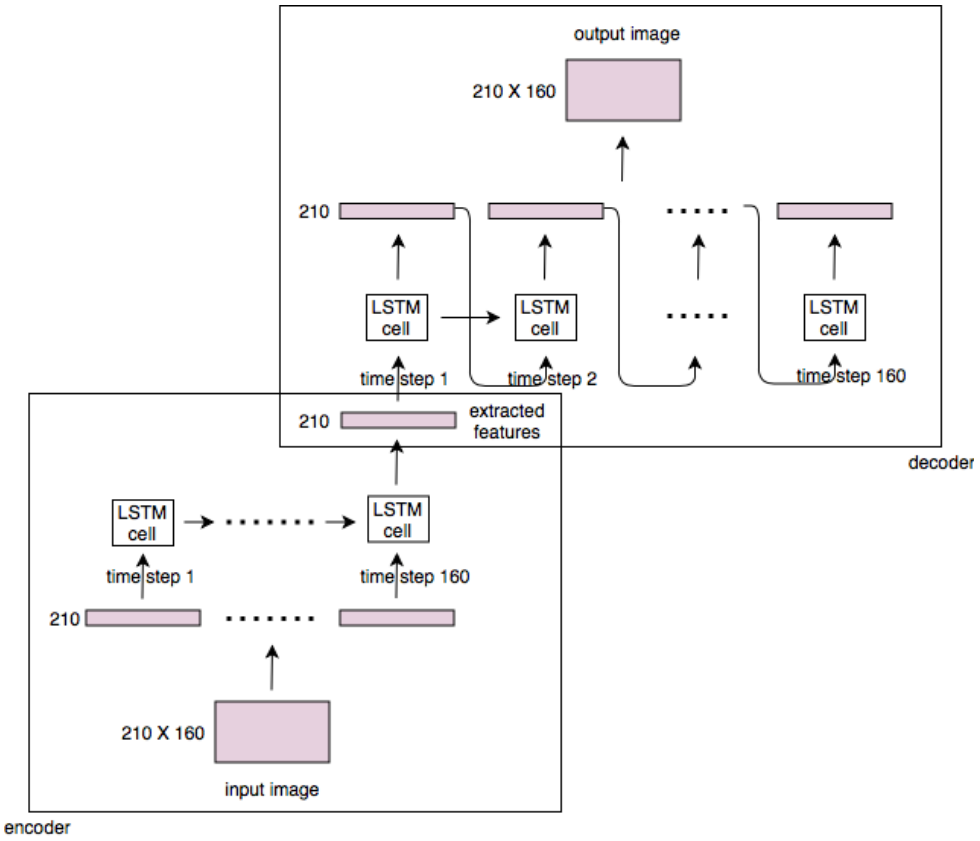



FIGURE 3.8: Architecture of the action-free RNN based AE

**The encoder** is a LSTM network with 160 time steps. The input image which is a $210 \times 160$ pixel matrix is treated as 210 vectors of size 160 with each one represents a line of the image. The LSTM cell takes one line at each time step and returns an output of size 160. In this model, only the output at the last time step is treated as the features of the input image.

**The decoder** is also a LSTM network with 210 time steps. The initial internal state of the LSTM cell is set to the final internal state of the encoding LSTM cell. Different from the encoder, the decoder does not take a full image but just the vector of features as its input. What is more, this feature vector is only used at the first time step. Inputs for the other time steps are generated by previous time step. This means, at the first time step the feature vector is fed to to the LSTM cell and generates the output vector and the internal states. At the second time step, the LSTM cell takes the output from the first time step as its input and generates the output for this time step. The same things go for the rest of the time steps. When the last time step is finished, the decoder has generated

210 outputs of size 160 in total. Hopefully, this $210 \times 160$ matrix can be used to restore the input image.

In this LSTM Based AE Model the feature size is limited to just 160, which looks quite unreliable. Since the size of the RAM features is just 128, it is still worth trying cause it might work.

**Training Details**

For each game, we train the RNN AE model with the same data set of the CNN AE model. Training for the LSTM AE model is the same with the training for CNN AE model. Training loss is the mean square error of the original image and the restored image. Adam is used with learning rate of 0.001 as the optimizer. The batch size for each iteration is 64. The number of iterations trained is 200.

## 3.3 Training With Action Transformation

Recall that in the models trained without action transformation, only one image is taken as the input and the output image is anticipated to be exactly the same or highly similar to the input image. This is how an ordinary AE does to extract important information or find a new representation of the input data. However, in this section the AE models we are going to talk about is totally different from ordinary AEs. It is more like a predictor but not a compressor-decompressor.

The idea of building action transformation based AEs comes from the action-conditional spatio-temporal prediction architectures on ALE screens introduced by Junhyuk, Guoxiao, Honglak, Rickl and Baveja in 2015 [28]. They have introduced two architectures, one CNN architecture and one RNN architecture. This is where our two AE models going to be discussed in this section originate from. First, let us focus on the notion of action transformation.

In reinforcement learning, action transformation is the transformation from a state to a new state when an action is applied. In our models, the meaning of action transformation is the same. It is a function that can map a game image into a new image with an action factor. The action transformation function can be written in the format of $f : s_t, a_t \rightarrow$

$s_t + 1$. $s_t$ and $a_t$ are the game screen and the action taken at time t. Our models are anticipated to learn such a function to predict the next game screen when the current game screen and an action are given.

In both models, the current game screen which is the input of the models will be encoded into a representation of unknown features. We call these features the hidden states and label them as $h_t^{enc}$ for time $t$. An action is then applied to the hidden states to generate the new hidden states $h_t^{dec}$ for the next time $t + 1$. At last, the newly generated hidden states representing the features for the next game screen will be decoded to the predicted next screen $s_{t+1}$.

The key to the action transformation is a method called multiplicative interactions between the encoded hidden states and the chosen action:

$$h_{t,i}^{dec} = \sum_{j,l} W_{ijl} h_{t,j}^{enc} a_{t,l} + b_i, \tag{3.2}$$

where $h_t^{enc}, h_t^{dec} \in R^n$, $a_t \in R^a$, and $W \in R^{n \times n \times a}$. Here $n$ is the length of the vector $h_t^{dec}$, and a is the length of $a_t$ which is 18 in our model. $b \in R^n$ is bias. $a_t$ is an one-hot vector of size 18. $h_{t,i}^{dec}$ is the $i$th value of the $h_t^{enc}$. $a_{t,l}$ is the $j$th value of the $a_t$ which is either 0 or 1. This enables our model to have different transformations for different actions. This is a pretty good way to model the way of action transformations, but in practice it is quite hard to scale up because of its large amount of parameters. What is more, a 3-way tensor computation like this is hard to be implemented in Tensorflow. Fortunately, an alternative way that approximates the multiplicative interactions is also introduced by Junhyuk, Guoxiao, Honglak, Rickl and Baveja [28]. They approximate the 3-way tensor by factorizing it into three matrices:

$$h_t^{dec} = W^{dec}(W^{enc} h_t^{enc} \odot W^a a_t) + b, \tag{3.3}$$

where $h_t^{enc}, h_t^{dec} \in R^n$, $a_t \in R^a, W^{dec} \in R^{n \times f}$, $W^{enc} \in R^{f \times n}$ and $W^a \in R^{f \times a}$. Here $f$ is the factor number which is equal to the number of extracted features from the input screen. Remind that when we use the equation 3.3 to approximate them equation 3.2, we no longer use the hidden states $h_t^{enc}$ but the $W^{enc} h_t^{enc}$ (a vector of size $f$) to represent the screen features. This is because $W^{enc} h_t^{enc}$ is where the action factor $W^a a_t$ directly works on.

### 3.3.1 CNN Based AE Model

In this part, we try to build an action-conditional CNN AE model to learn the function $f : s_t, a_t \rightarrow s_{t+1}$.

#### Network Architecture

The action-conditional CNN AE is constructed of three components: an encoder, an action transformer and a decoder.

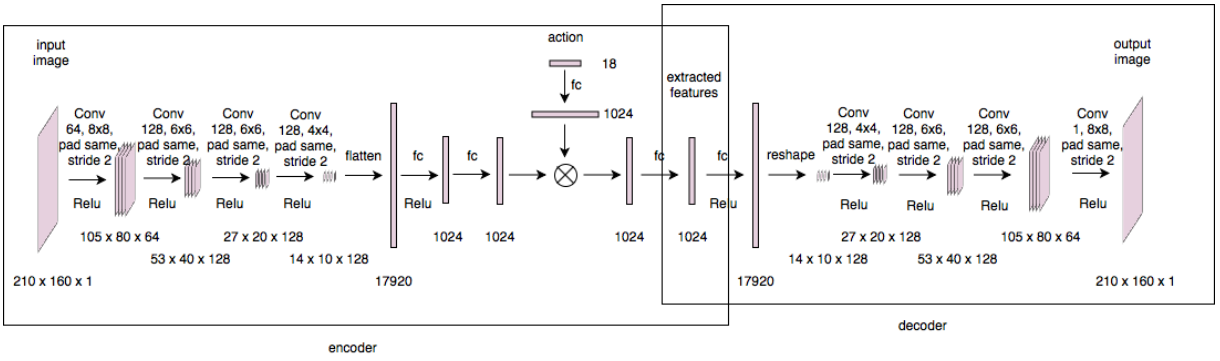



FIGURE 3.9: Architecture of the action-conditional CNN based AE

**The encoder** is where a state $s_t$ is encoded to $h_t^{enc}$. It is exactly the same with the encoder used in the action-free CNN AE model in 3.2.1. There are 4 convolutional layers with 64, 128, 128, 128 filters respectively in the encoder. Filter sizes are set to $4 \times 4$, $6 \times 6$, $6 \times 6$ and $8 \times 8$. Filtering strides are 2. The input is a $210 \times 160 \times 1$ pixel matrix with just 1 tunnel and is then transferred to a $14 \times 10 \times 128$ matrix with 128 tunnels after it is convolved. The output $h_t^{enc}$ is the flattened version of the convolved matrix.

**The action transformer** transfers $h_t^{enc}$ and $a_t$ to $h_t^{dec}$ according to the equation 3.3. It has two inputs: the $h_t^{enc}$ and an one-hot action factor $a_t$. Each of the inputs is handled by a fully connected layers with 1024 (factor number $f$) neurons to compute $W^{enc}h_t^{enc}$ and $W^{enc}h_t^{enc}$. Then the $W^{enc}h_t^{enc}$ and $W^{enc}h_t^{enc}$ are fed to a matrix multiplication layer which returns a tensor representing $W^{enc}h_t^{enc} \odot W^a a_t$. The last layer is a fully connected layer with $17,920$ $(14 \times 10 \times 128)$ neurons followed by a rectified linear function. This is where $W^{dec}(W^{enc}h_t^{enc} \odot W^a a_t) + b/h_t^{dec}$ is computed with $W^{enc}h_t^{enc} \odot W^a a_t$. In this way, we can finally get a vector $h_t^{dec}$ that can be used to predicted the next game screen $s_{t+1}$ with the decoder.

**The decoder** is exactly the same with the decoder of the action-free CNN AE model in 3.2.1 apart from one thing. The input of the decoder is the vector $h_t^{dec}$ got from the action transformer. Before we decode it with the deconvolutional layers, we should first reshape it back to a $14 \times 10 \times 128$ matrix. This is the shape of the output of the encoding convolutional layers. Then a 4 layers deconvolutional network with 128, 128, 128, 64 filters, $8 \times 8$, $6 \times 6$, $6 \times 6$, $4 \times 4$ filter sizes and filtering stride 2 takes the matrix to generate the prediction (a $210 \times 160 \times 1$ matrix) of the next game screen.

In this model, we treat $W^{enc}h_t^{enc}$ as the features extracted from the input game screen. The feature set size is 1024 in this case.

**Training Details** The screen set we used to train this model is the same with the screen set we mentioned above. The training data set has 4480 training item and the test data set has 512 test data. Each data consists of a pair of screens and an action. The first screen in each pair is a game screenshot at some time $t$, and the second one is the following screenshot of the first one after an action has been taken. The training loss is the mean square loss of the second screen and the predicted screen. Batch size of each iteration is 32, and our model is trained for 200 iterations in total. Adam is used with the learning rate of 0.0001 as the optimizer.

### 3.3.2 LSTM Based AE Model

Again, the overall task is still learning the action transformation function $f$ in this model. A little different from mapping a state-action pair $s_t, a_t$ to a new state $s_{t+1}$, this action-conditional LSTM AE is trying to mapping a sequence of states and an action to a new state. This can be formalized with a new action transformation function $f : s_{t-k+1:t}, a_t \rightarrow s_{t+1}$ where k is the number of screens to look back. Also, apart from just making one step prediction we can instead make multi-step prediction. For example, a 3-step prediction means an action transformation function $f : s_{t-k+1:t}, a_t \rightarrow s_{t+1:t+3}$. $s_{t+1}$ will be used to predict $s_{t+2}$, $s_{t+1}$ and $s_{t+2}$ will be used to predict $s_{t+3}$

In the ALE, each action is applied 5 times to the Atari games but not just once. This is just like we keep pressing the button for certain amount of time to make our object move certain distance. In the Atari games, one single action performance can just change the location of game objects a little bit. From the perspective of our human players, repeatedly

applying each action 5 times at each planning step is what sounds more reasonable. What is more, if our agent makes an action decision every 5 frames but not every single frame, 4 times more efficiency will be acquired in our planning process.

To make our predicting model consistent with this ALE feature, a k value of 5 is applied to our action transformation function. Also, our model is made to make just one-step prediction cause we want to make it as simple as possible. As a result, the action transformation we hope to learned with this model is $f : s_{t-4:t}, a_t \rightarrow s_{t+1}$.

### Network Architecture

The action-conditional LSTM AE has four parts: a CNN encoder, a LSTM encoder, an action transformer and a CNN decoder.

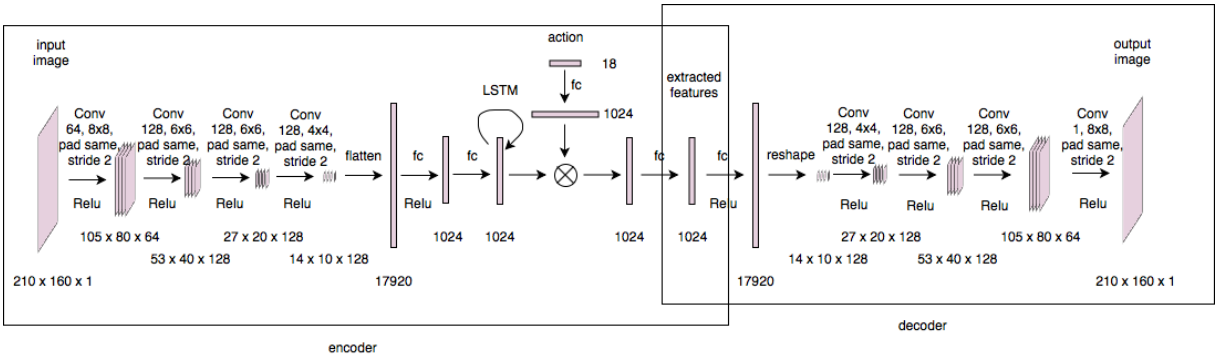



FIGURE 3.10: Architecture of the action-conditional RNN based AE

**The CNN encoder** is where a state $s_t$ is encoded to a feature vector of size 1024. This is exactly the same with what we did in the action-conditional CNN AE to generate the $W^{enc}h_t^{enc}$ features for a screen. 4 convolutional layers with 64, 128, 128, 128 filters comes first in the CNN encoder with filter sizes and filtering strides the same with the convolutional layers we talked above. After this, a fully connected layer with 1024 (factor number $f$) neurons followed by a rectified linear function comes next. The inputs are five $210 \times 160 \times 1$ screen matrices and the outputs are five encoded feature vectors of size 1024 with each one corresponds to an input screen.

**The LSTM encoder** is a LSTM network with 5 time steps. It takes the outputs from the CNN encoder and treats each of them an input item. The output of the last time step is the features $h^{enc}$ extracted from all the five screens $s_{t-4:t}$ and is also the output of the

LSTM encoder. Hopefully this final feature set could catch some knowledge of the game from a sequence of game screens.

**The action transformer** has two inputs: the feature set $h^{enc}$ got from the LSTM encoder and an one-hot action factor $a_t$. Each of the inputs is handled by a fully connected layers with 1024 (factor number $f$) neurons. This is then followed by a matrix multiplication layer that takes the two outputs as inputs and returns a tensor which is equal to $W^{enc}h^{enc} \odot W^a a_t$. This is the predicted features for the next game screen. The last layer is a fully connected layer with $17,920$ $(14 \times 10 \times 128)$ neurons followed by a rectified linear function. Finally, we can get a vector $h^{dec}$ of length $17,920$ that can be used to predicted the next game screen $s_{t+1}$ in this part.

**The CNN decoder** is exactly the same with the decoder of the action-conditional CNN AE model in 3.3.1. The input of the decoder is the vector $h^{dec}$ got from the action transformer. Before we decode it we should first reshape it to a $14 \times 10 \times 128$ matrix. Then the 4 layers deconvolutional network takes the matrix to generate the prediction (a $210 \times 160 \times 1$ matrix) of the next game screen.

In this model, the $h^{enc}$ from the LSTM encoder is the knowledge learned from the input screen sequence. We hope that it can capture some features of the current game state at time $t$.

### Training Details

The same screen set is also used in training this model. The data set has 4464 training data and 496 test data. Each data consists of 6 game screens arranged in time sequence and an action to be taken on the $5th$ screen $s_t$. The first 5 game screens are used to predict the $6th$ screen. The training loss is defined to be the mean square loss of the $6th$ screen and the predicted $6th$ screen $s_{t+1}$. Batch size of each iteration is 8, and our model is trained for 200 iterations in total. Adam is used with the learning rate of 0.0001 as the optimizer.

# Chapter 4

# Feature Construction From Game Screens

In Chapter 3, we have talked about the ways we tried of using deep learning methods to learn features from game screens. One problem of learning with these methods is that we can not label each of the learned features, which is to say that we have no idea what these features stand for. What is more, all these learning methods are trained completely independently from our planning methods. From Chapter 3, we can know that all the learning models are neither trained with any knowledge of the game nor trained with any feedbacks from the ALE or the IW agent. As a result, the feature sets learned from these models can be random representations of the game screens. We can not guarantee that the learned feature sets are reasonable enough for the IW algorithm to tell different screen states apart from each other.

Constructing features simply means choosing features ourselves from game states to use with our planning algorithms. It now has been a key concern for reinforcement learning, and how to select a good representation of these states is the most important design issue. Most of the time, the representation is tuned by the designer and is special for individual games. This sounds like a domain-specific way to manage our planning tasks as we still need to tune the screen representations for each Atari games when there is no need to change the planning algorithms. It is obvious not a good way to tackle our planning problem. Fortunately, it can still work because there are already some methods specially

designed for constructing features from images in general ways like **BASS**, **DISCO** originally introduced by Naddaf 2010 [29], **Basic** introduced by Bellemare et al. 2013 and B-PROS introduced by Yitao et al. 2015. All these feature types are constructed without any domain-specific knowledge and can capture image features at pixel levels. In this section, we will focus on the last two feature types: the **Basic** features and the **B-PROS** features.

## 4.1 BASIC Feature

The Basic method is derived from Naddaf's BASS (2010) method. It encodes the presence of colors which range from 0 to 255 on the Atari game screens. What should be mentioned here is that the Basic method also requires to remove the game background first.

The Basic method divides the $210 \times 160$ pixel Atari game screens into blocks of size $15 \times 10$. A basic feature $\phi_{(c,r),k}$ captures the presence of color $k \in (0, 255)$ in the block $(c, r)$ where $c \in (0, 13)$ and $r \in (0, 15)$. $\phi_{(c,r),k}$ is 1 if the color $k$ is observed in the block $(c, r)$ and is 0 if not. It is easy to compute that the size of the Basic feature set is **57,120** $= (14 \times 16 \times 255)$. If we apply the Basic features with block size of $15 \times 10$ to the IW(1) algorithm, we can get at most $57,120$ nodes to expand. Note that since we have removed the background from the game screens beforehand, the number of the basic features of a game screen is much smaller than $57,120$ actually because most of the pixels are 0 after the background is subtracted. More precisely, the number is around **7,168** $= 14 \times 16 \times 32$ as there are no more than 32 different colors in most of the Atari 2600 game.

The Basic features have a crude idea of the object types and the object positions in Atari game screens. After the game background has been removed, all colors in a game screen with non-zero value are key objects. The reason is that most of the objects in the Atari game are single colored. 80 percent of the objects have no more than two colors. As a result, if a Basic feature $\phi_{(c,r),k}$ is 1 we can roughly say that the object $k$ is at position $(c, r)$.

However, despite that the Basic method can tell every objects apart and locate their positions it is still not enough for our agents to make relatively good decisions in playing games. To this end, we turn to another way of feature extraction called B-PROS method.

## 4.2 Basic Pairwise Relative Offset in Space (B-PROS) Feature

The B-PROS method is built on top of the Basic method. It captures pairwise relative distances between objects and their locations in a single game screen. The B-PROS feature is a combination of Basic feature and **Basic Pairwise Relative Offset (B-PRO)** feature [30].

### The Basic Pairwise Relative Offset (B-PRO) method

Like the Basic method, the B-PRO method also divides the $210 \times 160$ pixel game screens into blocks of $15 \times 10$. The B-PRO feature is a binary value $\phi_{(i,j),k1,k2}$ with $i \in (-13, 13)$, $j \in (-15, 15)$ and $k1, k2 \in (0, 255)$. If $\phi_{(i,j),k1,k2}$ is 1, it means there is a color $k1$ in a block $(c, r)$ somewhere and a color $k2$ contained in the block $(c + i, r + j)$. Basically, this is to say that there is at least one pair of pixels of color $k1$ and $k2$ where pixel $k1$ is $i$ blocks vertically and $j$ blocks horizontally from the pixel $k2$. Recall that we have mentioned that objects in Atari games are often of single color and the Basic feature captures the locations of every objects by capturing the location of pixels. The B-PRO feature has adopted the same idea and captures the offset of each pair of objects in a game screen by indicating the offset of each pair of Basic features.

It is not difficult to compute that the size of the B-PRO feature set is **54,425,925** $= (27 \times 31 \times 255 \times 255)$. This is a quite big number and when we plan with the IW(1) algorithm the novelty table will be extremely huge.

### The Basic Pairwise Relative Offset in Space (B-PROS) method

The B-PROS method is not a new way to construct screen features but simply a way that exploits the idea of Basic method and B-PRO method and combines their features together. The Basic feature in the B-PROS method is constructed in the same way that we discussed above while the B-PRO feature in the B-PROS method is slightly different. First, let's look at the case of a screen with just two non-zero pixels $k1$ and $k2$ with $k1$ contained in the block $(2, 2)$ and $k2$ in the block $(2, 3)$. The B-PRO feature set got from this screen will contain two features: $\phi_{(0,1),k1,k2} = 1$ and $\phi_{(0,-1),k2,k1} = 1$. Quite obvious, one of these two features is redundant and we only need one of them. To this end, in the

B-PROS feature all the redundant features are removed. This can be simply implemented by adding an condition $k2 > k1$ to the feature $\phi_{(i,j),k1,k2}$.

The complete B-PROS feature set is a combination of the Basic feature set and the B-PRO feature set with redundancy removed. Intuitively, what the B-PROS feature is trying to encode is the information like "there is a blue pixel in the block (3,3) and a green pixel in the block (2,2). The blue pixel is (1,1) blocks away from the green pixel", or even like "there is an object of blue in block (3,3) and an object of green in block (2,2). The blue one is (1,1) blocks away from the green one". The size of the B-PROS feature set is **27,213,090**. This is computed by summing the sizes of the Basic feature set and the B-PRO feature set with redundant features removed, which is $((14 \times 16 \times 255) + (27 \times 31 \times 255 \times 255 - 255)/2 + 255)$. For games with just 32 colors, the size of the feature set is about **435,840** which is much smaller.

## 4.3 Manhattan B-PROS method

The Manhattan B-PROS method is a simplified version of the B-PROS method. Instead of encoding the pairwise Basic feature offset in the form of a pair of block distances in horizontal and in vertical, it uses the Manhattan distance between blocks to represent the pairwise offset. To be more specific, a B-PRO feature in Manhattan B-PROS method is a binary feature $\phi_{d,k1,k2}$ where $d$ is the Manhattan distance between the blocks that $k1$ and $k2$ are contained. For example, if a color $k1$ is found in block (2,2) and $k2$ is found in block (3,3), then the feature $\phi_{2,k1,k2}$ is set to 1 to indicate that $k1$ and $k2$ are 2 blocks away from each other. Note that $d$ is equal to $\mid i \mid + \mid j \mid$ and is always a positive value ranging from 0 to 28. In this case, the maximum number of features we can generate is **476,672** $= (14 \times 16 \times 255) + ((13 + 15) \times 255 \times 255/2)$ and **14,336** for games with at most 32 colors.

By using the Manhattan B-PROS method, we can encode the information like the distance between any two objects regardless of the relative direction from one to the other. One reason of making this change is that we hope the generated feature set is not too big so that the size of the novelty table created by IW algorithm is reasonable. If the novelty table is too big, it will take enormous time to update the table and check the state novelty. The other reason is that the performance of IW algorithm relies on the state features quite

a bit, especially in the Atari 2600 games where distances between game objects are more important than their relative offsets along each coordinate axes.

# Chapter 5

# Experimental Results

Note that the IW(1) algorithm we mentioned in the results is the one we improved to solve the problems related in 2.3.2. We tested IW(1) with RAM features, features learned from deep learning models and features constructed by B-PROS method and Manhattan B-PROS method. We have only tested e-feature IW(1) with features constructed by the Manhattan B-PROS method. As what we have said earlier in Chapter 1, the e-featured IW algorithm is mainly used for dealing with the ambiguity of the Manhattan B-PROS features. This is done by reusing the nodes that have no novelty but can improve the depth of the atoms in the novelty table. In other words, the e-feature IW prefers not only the nodes that are novel but also the nodes whose atom values are easier to be reached. All the experiments are tested over 7 Atari 2600 games (Ms Pacman, Alien, Tennis, Pong, Seaquest, Space Invaders and Freeway) of 5 different types. The Ms Pacman and the Alien have similar game rules, but their game screens are totally different. The shapes and colors of the objects in the Alien are more complex than they are in the MS Pacman. The Pong and the Tennis also have similar game rules. The main difference between these two games is that the Tennis has more available actions than the Pong. Rewards in the Pong and the Tennis are delayed, which is also the case in the game Freeway. This makes these three games quite hard to score with an AI agent. The Seaquest and the Space Invaders are totally different from the other five games, especially the Seaquest where the game rules are much more complex than the others.

Results of these experiments are arranged in two parts. In the first part we will talk about the results of applying learned features to our IW(1) agent, and in the second part we will

talk about the results of applying constructed features to our IW(1) agent and e-featured IW(1) agent.

Before we go further into the experiments, let us first look at the results of the IW(1) agent, the improved IW(1) agent and the UCT agent with the RAM feature to have an idea of the performances of the IW(1) algorithms.

## 5.1 IW(1), improved IW(1), UCT with the RAM feature

Each game is evaluated over 10 runs with at most **150000** simulation nodes per action step and 18000 frames in total for IW(1) and UCT, and at most **50000** simulation nodes per action step and 18000 frames in total for the improved IW(1). The results of these three methods are showed in Figure 5.1.

| | ***UCT*** | ***IW(1)*** | | ***Improved IW(1)*** | |
|---|---|---|---|---|---|
| | Score | Score | Time | Score | Time |
| *Ms Pacman* | 22336 | 21695 | 21 | **24847** | 8 |
| *Alien* | 7785 | 25634 | 81 | **26300** | 11 |
| *Tennis* | 3 | **24** | 21 | **24** | 8 |
| *Pong* | **21** | **21** | 17 | **21** | 6 |
| *Seaquest* | 5132 | 14272 | 25 | **25500** | 8 |
| *Space Invaders* | 2718 | 2877 | 21 | **3165** | 12 |
| *Freeway* | 0 | **31** | 32 | **31** | 9 |

FIGURE 5.1: Game results of the original IW(1) algorithm, the improved IW(1) algorithm and the UCT algorithm with the RAM feature. Scores in bold are the highest scores achieved among these agents. Time is shown in the average CPU times per action in seconds. The results of the original IW(1) algorithm are proposed in [7]. The results of the UCT algorithm are proposed in [11].

We can see that on the seven games we tested the improved IW(1) algorithm outperforms the original IW(1) algorithm and the UCT algorithm quite a lot. Especially in the Seaquest, the improved IW(1) algorithm scored nearly twice as much as the other two algorithms did. For the other six games, the improved IW(1) algorithm is also slightly better than the other two algorithms. The most important is that the computational

budget for the improved IW(1) algorithm is just 1/3 of the budget for the IW(1) and the UCT algorithm. This is the reason why the improved IW(1) agent takes much less CPU time to make an action than the IW(1) agent. For example, in the Alien the IW(1) agent takes 81 seconds averagely to take an action while the improved IW(1) agent just takes 11 seconds.

## 5.2 Experiments with Learned Features

### 5.2.1 Training results

Recall that we have built four models in total: one action-free CNN model, one action-free RNN mode, one action-conditional CNN model and one action-conditional RNN model. However, it turns out that only one of these models converged finally, the action-free CNN model. The other three models all failed to restore the original input image or predict the next game screen. Training loss of these models trained on the Freeway can be found in Figure 5.2. The reason why we tested our models on the Freeway first is that the next game screen of the Freeway is totally predictable according to the current game screen while in most other games only part of their screens can be predicted. For example, in the Tennis the movements of our opponent are kind of random sometimes and in the Ms Pacman the movements of the ghosts are unpredictable even for human players sometimes. The action-conditional CNN model and the action-conditional RNN model are all built in a similar way introduced by Junhyuk et al. in 2015. They have made their models successfully predict game screens over 30-500 steps ahead depending on games with very few errors. Instead of setting the factor number to 1024, they set it to 2048. This asks for more than 4GB GPU memories, but we just have a NVDIA GeForce GTX 850M with 2GB memory. Due to this reason, we finally decided to limit the factor number to 1024 for both models. In fact, we have also tried to limit the filter numbers of the convolutional layers to 32, 64, 64, 64 for both models, and increased the number of factors from 1024 to 2048. Unfortunately, our action-conditional models still failed to converge because the training loss barely dropped. According to Junhyuk, their data set consists of about 500,000 training screens and 50,000 test screens. Compared with their data set, the data set we used is just about 1 percent of its size. 5000 screens to be specific. This might be the reason why we have totally different results. As they have generated

such a huge data set, let us first think if it reasonable for us to train a model with a data set of this size to reach our goal. In Atari 2600 game, there are no more than 32 colors in most games after we subtract the background. Nearly 1/3 or more of the game screen pixels are the background and never change. Intuitively, the size of the screen space with background subtracted is roughly around or much less than 716,800 ($210 \times 160 \times 32 \times 2 \div 3$). 500,000 screens nearly cover most of the possible game screens or the most frequent game screens can be observed in a game. Remind that the purpose of our model is trying to extract features from game screens where these features are representative enough to be transformed with an action vector to new features that can be used to predict the next game screen. If we train our model with 500,000 training data, it is very likely that what we will get in the end is a model that has memorized all the scenarios but not a model learned some features. When sampling screens from Atari games, either with a IW agent which is what we did in this research or with a DQN gent which is what Junhyuk et al. did in their research, it is very hard to make sure that the test data is completely exclusive to the training data. As a result, it is very hard to guarantee that our models did learn some knowledge from the game screens with a large data set even if they converged. What is more, Junhyuk and his fellows trained their models for 1,500,000 iterations. This makes it quite possible that their models are over-estimated because the development loss is not meaningful when we can not guarantee our test data is different from our training data.

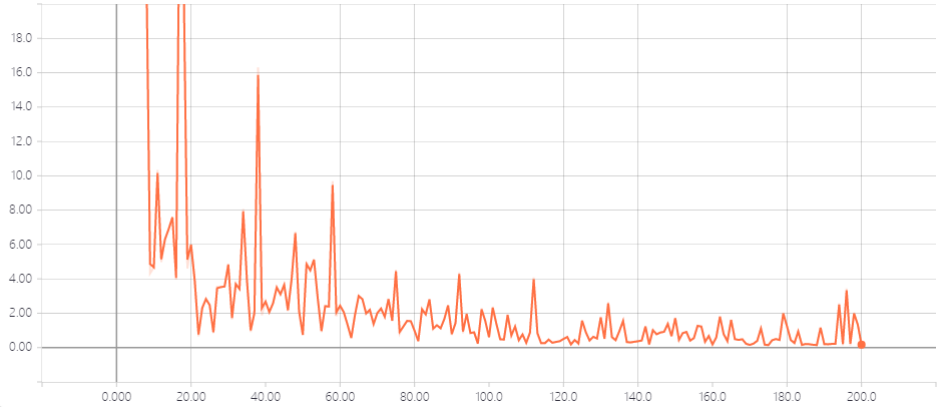
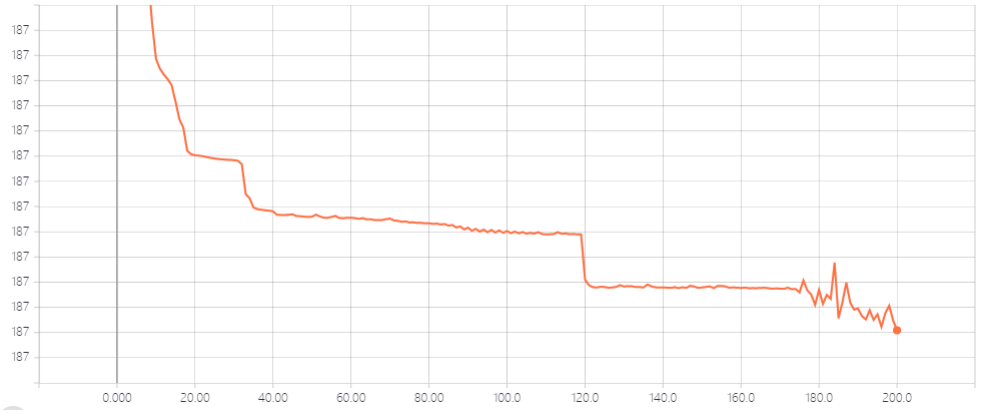
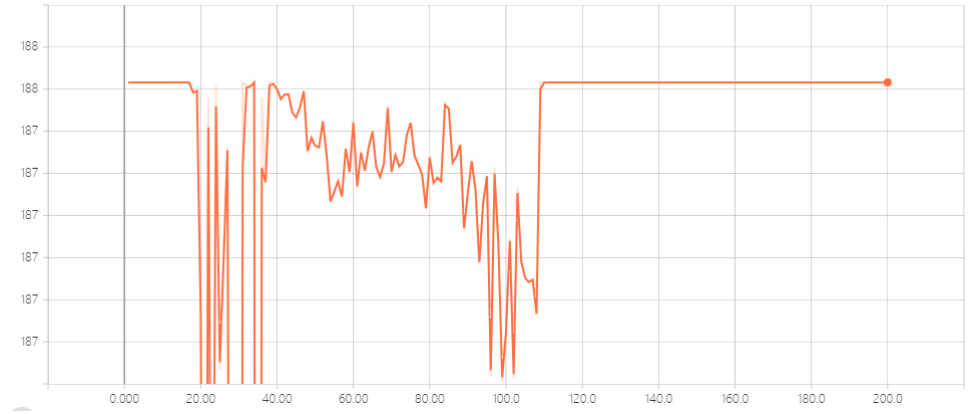



FIGURE 5.2: Training losses. X-axis is the number of training iteration. Y-axis is the training loss.




(a) Training loss of the Action-free RNN AE model




(b) Training losses of the Action-conditional CNN AE model


(c) Training losses of the Action-conditional RNN AE model

FIGURE 5.2: Training loss (cont). X-axis is the number of training iteration. Y-axis is the training loss.

Anyway, from Figure 5.2 we can know that only the action-free CNN based AE model

finally learned something from the Freeway screens. Figure 5.3 shows two Freeway screens and the corresponding restored screens generated by the model. For each input screen, features extracted with the action-free CNN AE model as a whole can be seen as a matrix of shape $14 \times 10 \times 128$. This $14 \times 0 \times 128$ matrix is basically composed of 128 $14 \times 10$ screen transformations of the input screen. Part of the transformations of the two Freeway screens can be found in the Figure 5.3. From it we can know that what our action-free CNN AE model actually did is something like down-sampling from the input screens. In our action-conditional CNN AE model, what we basically did is to compress these transformations to a factor vector first then apply an action transformation to it to generate a new factor vector. After this, this new factor vector will be decompressed to the transformations of the next game screen which will be used to predict the next game screen. This seems like a reasonable way to make game screen predictions, but the fact is that the screens generated by our other models are full of noise.

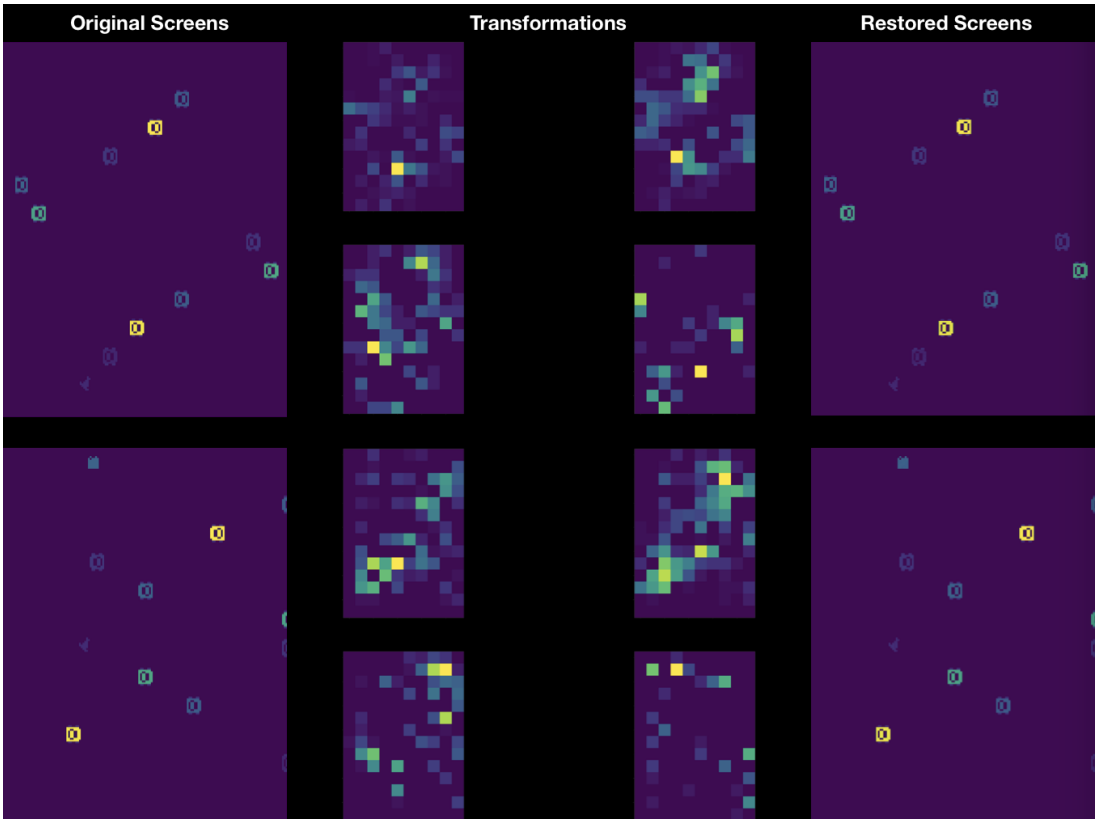



FIGURE 5.3: Original screens are the inputs of the action-free CNN AE model. Transformations are extracted features of the input screens. Here, we only displayed 4 out of 128 transformations of each screen. Restored screens are the outputs of the model which are the restored screens of the original screens

### 5.2.2 Planning details

Since only the action-free CNN model has converged at last, what we did in our experiment is planning with the IW(1) algorithm where state novelty is evaluated with learned screen features from the action-free CNN model. Remind that our model is written in python Tensorflow and our IW(1) agent is written in C++. To make our C++ agent able to use the model trained in python, one way to do is first exporting the graph of our model as text that defines the proto of the model graph and saving the whole model to the format of checkpoint after our model has been trained. Then a Tensorflow script named freeze_graph.py is used to convert the graph text and the checkpoint files into constant options in a standalone graph definition file that can be loaded in C++ Tensorflow. The C++ model is called every time after the IW(1) agent has emulated a node to update the node state with features learned from the node screen. Note that early in the 2.3.2, the novelty table in the IW(1) algorithm with RAM features is a set of 128 vectors with each one of 256 boolean values while in this case the novelty table is a set of 17920 ($14 \times 10$ features) vectors with each one of 1024 boolean values. 1024 is the max value among all the features. Each feature value is discretized by pruning the decimal value. This is basically how we combine our IW(1) agent with learned features.

### 5.2.3 Game results

We have only tested these models described above on three games: the Ms Pacman, the Tennis and the Freeway. These games are relatively simpler in both game rules and game screens than other Atari 2600 games, so we tested our method on these games first to see what can we get. In each search iteration, our computational budget is 50,000 nodes. This means, each time we can emulate up to 50,000 nodes ahead to make an action decision for the current node.

The method is evaluated over 10 runs for each of these games, which is the same with the experiment conducted by Nir Lipovetzky et al. 2015 using IW(1) algorithm with RAM features and the one conducted by Marc G Bellemare et al. 2013 using UCT algorithm with RAM features. Game results of our IW(1) agent with learned features are listed below.

- **Ms Pacman**: Our agent finally got a total score of **638** and ended the game at frame **2090** with losing all lives. Each action is made in at least 100 seconds. This is a very bad result compared with the result got from the UCT algorithm with RAM features which is **22336** in average.

- **Tennis**: The score our agent got in this game is **-18**, which is the lowest score can be reached in this game. Each action is made in at least 100 seconds. This is also a very bad result compared with the result got from the UCT algorithm with RAM features which is **2.8** in average.

- **Freeway**: In this game, our agent got a score of **0**, which means it failed to score at all. Each action is made in at least 100 seconds. The UCT algorithm got a score of **0.4** in average.

From the game results we can know that our IW(1) agent with features learned from screens with an action-free CNN based AE model finally failed to control the 3 simplest games out of the 7 Atari 2600 games we chose. This basically means the way our model compressing the information of game screens is neither a way similar to game RAMs nor a way reasonable for humans or game rules. What we are sure is merely that the features learned from a screen are truly a representation of this screen. This is a problem of using AE models to find a mapping from the input data to representative features that you can not control the meaning of the mapping at all. What is more, as what we have said before, the performance of the IW algorithm highly relies on the state features because the power of IW algorithm in searching depends on the node pruning which is related to the novelty of the state features. If the state features are not capturing the structure of the game, it is very likely that the IW algorithm will fail to make good action decisions.

In conclusion for the deep learning methods that we have tried in this research work, it is actually very hard to use the action transformations to guide the learning process. Even if we can finally make these models converge, it will still be difficult to guarantee that the features learned are game related and can be applied to the IW algorithm. This is not what we have anticipated at the very start that the AE models can at least capture the object locations or relative offsets between different objects.

## 5.3 Experiments with Constructed Features

### 5.3.1 Constructed screen features

Recall that the B-PRO method (mentioned in the 4.2) originally divides the $210 \times 160$ game screens into blocks of size $15 \times 10$. This does not work very well on the Atari game, because many objects in the games are much smaller than $15 \times 10$ and move just few pixels per action, which means the B-PRO features will not change at all if the screen has just changed a little bit. This is how we applied the B-PROS features to the IW(1) at the very start. For this reason, in some of the games like the Space Invaders, the Alien and the Ms Pacman, our agent can not even prevent itself from losing lives and ends the games very early. As a result, we change the block size from $15 \times 10$ to $5 \times 5$ so that the B-PRO features can capture the slight movement of the objects. Fortunately, this actually works pretty well.

As for the size of the feature sets, there are at most $170,350,710 = ((42 \times 32 \times 255) + (83 \times 63 \times 255 \times 255 - 255)/2 + 255)$ and most commonly **2,720,272** $= ((42 \times 32 \times 32) + (83 \times 63 \times 32 \times 32 - 32)/2 + 32)$ features in the B-PROS feature set, and at most $2,683,748 = ((42 \times 32 \times 255) + ((41 + 31) \times 255 \times 255 - 255)/2 + 255)$ and most commonly **79,888** $= ((42 \times 32 \times 32) + ((41 + 31) \times 32 \times 32 - 32)/2 + 32)$ features in the Manhattan B-PROS feature set. Though the sizes of these feature sets have increased to about 4 times of they are with the block size of $5 \times 5$, it is still acceptable that most games have no more than 32 different colors.

### 5.3.2 Game results

The results are arranged in four groups. The first one comes from the IW(1) agent with the RAM feature which is treated as the baseline. The second and the third groups come from the IW(1) agent with the B-PROS feature and the Manhattan B-PROS feature respectively. The last one is from the E-feature IW(1) agent with Manhattan B-PROS feature. The e-feature IW(1) algorithm is another kind of improvement made on the IW(1) algorithm special for the Manhattan B-PROS feature. The idea of using e-features in the IW(1) algorithm has been explained in 2.3.3.

Each game is evaluated over 10 runs with at most 50000 simulation nodes per action step and 18000 frames in total apart from the Ms Pacman. This is because the background of the Ms Pacman changes in the third stage and in the ALE environment our agent can not identify the stage it is in, which means we can not remove the screen background if the game background has changed. In practice, the B-PROS method without removing the screen background still works kind of well with the IW(1) algorithm. The main problem is that the search process will be incredibly slow because the size of the screen feature set can be extremely large, and the performance of our agent will also be influenced. As a result, instead of running our agents for 18000 frames we finally only ran all of our agents on the first two stages of the Ms Pacman. Game results are showed in the Figure 5.4.

| | *IW(1)* | | | | | | *E-feature IW(1)* | |
|---|---|---|---|---|---|---|---|---|
| | ***RAM feature*** | | ***B-PROS feature*** | | ***Manhattan B-PROS feature*** | | ***Manhattan B-PROS feature*** | |
| | Score | Time | Score | Time | Score | Time | Score | Time |
| *Ms Pacman* | **20400** | 6.8 | 12360 | 46 | 9300 | 7.5 | 11100 | 8 |
| *Alien* | 26300 | 9.9 | 12090 | 23 | 28710 | 7.5 | **33680** | 8 |
| *Tennis* | **24** | 5.2 | -21 | 12.4 | **24** | 6.3 | **24** | 6.7 |
| *Pong* | **21** | 4 | **21** | 6.7 | **21** | 1.8 | **21** | 2.3 |
| *Seaquest* | **25500** | 8.3 | 297 | 8 | 339 | 6.3 | 359 | 6.7 |
| *Space Invaders* | **3165** | 8.3 | 1863 | 26.55 | 1449 | 4 | 2470 | 4.5 |
| *Freeway* | **31** | 9 | 8 | 17.3 | **31** | 11 | **31** | 12.7 |

FIGURE 5.4: Game results of the IW(1) and the e-feature IW(1) with constructed features. Time is shown in the average CPU times per action in seconds. Numbers in bold show the best performer among all the agents. Scores shaded in light gray show the best performer among the screen based agents

Let us first look at the figures in the red box, which is the results from the screen based agents. We can see that the agent of IW(1) algorithm with B-PROS feature is better at the Ms Pacman, Pong and Space Invaders while the one of IW(1) with Manhattan B-PROS feature is better at the Alien, Tennis and Freeway. None of them is good at Seaquest. It is hard to say that which one of the methods is absolutely the best because neither of them dominates the other one. However, in terms of the generality the IW(1) agent with Manhattan B-PROS feature outperforms the IW(1) agent with B-PROS feature. The scores achieved by the Manhattan B-PROS based IW(1) agent are actually comparative to the scores can be achieved by ordinary human players in the listed games apart from the Seaquest. We will explain why our agent can not play the Seaquest at all later. In addition, the IW(1) agent with the Manhattan B-PROS feature works much faster than the one with the B-PROS feature. This is simply due to the smaller size of the B-PROS feature set.

The agent of the e-feature IW(1) with Manhattan B-PROS feature has the best performance among all the screen based agents. It outperforms the other two agents in nearly all of the games apart from the MS Pacman. However, if we compare it with the baseline agent it is still not good enough but pretty close. All of our screen feature based agents died in the Seaquest at a very early stage apart from the RAM based IW(1) agent. This is because there is no punishment on the reward for making actions that lose lives in the Seaquest. In fact, our agent will receive a higher reward (40) for crashing its submarine than scoring in an ordinary way like hitting the creatures in the sea which is just 20. As a result, our screen based agents are more likely to crash its submarine as the reward is doubled. However, the RAM feature can capture the state of death somehow so that the RAM based IW(1) agent can prevent its submarine from being destroyed.

Anyway, despite the performances of all the screen based agents are not comparative to the performance of the RAM based agent, they have still done a good job on the several games we listed. Especially the agent of the e-feature IW(1) with the Manhattan B-PROS feature. This has showed us a possible way of applying the screen features to the IW algorithm on the problem of planning on the Atari games.

The performance of all these agents can be played following this link:

[**https://www.youtube.com/channel/UC9QnDPExehDjGwc428EXnyg/playlists**].

Code can be found:

[**https://github.com/xiahangyu/ALE-Atari-Width**]

# Chapter 6

# Future Work

In this work, the deep learning models we have tried are completely based on the screen pixels. This might not be a reasonable way of extracting screen features that are meaningful on planing games. This is to say, the features extracted with the Autoencoders are not likely to be representative enough for the game state. The model introduced by Sascha Lange and Martin Riedmiller in 2010 [31] may be useful in solving this problem. Generally, they combine deep Autoencoder neural networks with batch-mode RL algorithms [32] to learn compact screen features with game feedbacks. This could be one way to improve the learning models.

Since the B-PROS method captures not only the distance but also the direction between objects, this is the main reason why it has better performances on the Ms Pacman than the Manhattan B-PROS method, we can also encode some direction information in the Manhattan B-PROS method. For example, instead of characterize the B-PROS feature as $\phi_{d,k1,k2}$, we can format it as $\phi_{d,direction,k1,k2}$ where the direction represents top left, top right, bottom left or bottom right with 0, 1, 2, 3 respectively.

# *Acknowledgements*

Thanks a lot to my supervisor Nir Lipovetzky who has helped me quite a lot and informed me of the fascinating world of AI. Thanks Nir, it is a great honor and fortune of me to have this wonderful journey.

# Bibliography


[1] Elly Zoe Winner. *Learning domain-specific planners from example plans*. PhD thesis, United States Air Force, 2008.

[2] David E Wilkins. Representation in a domain-independent planner. In *IJCAI*, volume 8, pages 733–740, 1983.

[3] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810.

[7] Nir Lipovetzky, Miquel Ramírez, and Hector Geffner. Classical planning with simulators: Results on the atari video games. In *IJCAI*, 2015.

[8] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 37–49, 2012.

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[10] Stephen Grossberg. Recurrent neural networks. *Scholarpedia*, 8(2):1888, 2013.

[11] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[12] Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael H. Bowling. State of the art control of atari games using shallow reinforcement learning. *CoRR*, abs/1512.01563, 2015.

[13] Wilmer Bandres, Blai Bonet, and Hector Geffner. Planning with pixels in (almost) real time. *arXiv preprint arXiv:1801.03354*, 2018.

[14] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[15] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[16] Adithya M Devraj and Sean P Meyn. Differential td learning for value function approximation. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 6347–6354. IEEE, 2016.

[17] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992.

[18] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[19] Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *Proceedings of the 20th European Conference on Artificial Intelligence*, ECAI'12, pages 540–545, Amsterdam, The Netherlands, The Netherlands, 2012. IOS Press. ISBN 978-1-61499-097-0. doi: 10.3233/978-1-61499-098-7-540. URL https://doi.org/10.3233/978-1-61499-098-7-540.

[20] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[21] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[22] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, pages 2067–2075, 2015.

[23] Jack Kiefer, Jacob Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[24] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, page 14, 2012.

[25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[27] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[28] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in neural information processing systems*, pages 2863–2871, 2015.

[29] Yavar Naddaf et al. *Game-independent ai agents for playing atari 2600 console games*. PhD thesis, University of Alberta, 2010.

[30] Erik Talvitie and Michael H Bowling. Pairwise relative offset features for atari 2600 games. In *AAAI Workshop: Learning for General Competency in Video Games*, 2015.

[31] Sascha Lange and Martin A Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *IJCNN*, pages 1–8, 2010.

[32] Dirk Ormoneit and Śaunak Sen. Kernel-based reinforcement learning. *Machine learning*, 49(2-3):161–178, 2002.