Supervised and Reinforcement Learning for Fighting Game AIs using Deep Convolutional Neural Network

by

Nguyen Duc Tang Tri

A project report submitted in partial fulfillment for the degree of Master of Information Science

School of Information Science

Japan Advanced Institute of Science and Technology

March 2017

Abstract

AI has become important for human life since its application can help human in problem-solving. Imaging a world, when workers in dangerous environment are replaced by Robot, oldsters are taken care by automated and comfortable services, self-driving cars reduce the number of accidents etc. That wonderful world is a big dream but not impossible. Step by step, human improve AI and achieve many positive signals.

One of the early successes of human is creating AI that can defeat human player in some simple games. This success is meaningful because from the beginning of mankind history, game has been selected as a testbed of intelligence. For example in Japan, strong board game players are respected as intelligent people, about tens percent people may think "Habu Yoshiharu" is one of the most intelligent men in Japan. Furthermore, game is simple and easy to understand. In game, rules are clearly defined so we can evaluate human player or AI easily by matches. Therefore, when an AI can defeat human player, even in a very simple game, we can confirm this AI is quite "smart".

In 2016, Google has acquired DeepMind and tried to attack the hardest problem in board games: the Game of Go. Finally, an AI named AlphaGo was created based on Deep Q-network, self-playing method which allowed AlphaGo to improve itself, and Monte Carlo tree search. This powerful AI, which combined two cores of AI for games: tree search and machine learning, defeated human champion Lee Sedol in March 2016, opens a new era for Deep Learning.

Deep Learning has become most popular research topic because of its ability to learn from a huge amount of data. In recent research such as Atari 2600 games, they show that Deep Convolutional Neural Network (Deep CNN) can learn abstract information from pixel 2D data. After that, in VizDoom, we can also see the effect of pixel 3D data in learning to play games. But in all the cases above, the games are perfect-information games, and these images are available. For imperfect-information games, we do not have such bit-map and moreover, if we want to optimize our model by using only important features, then will Deep CNN still work?

In this report, a method has been described to successfully incorporate Deep CNN with optimized non-visual information. We investigated the allocation of features are important and valuable for improving its performance. By intentionally arranging features as an 2D grid, with some duplication of features and well-considered allocation, Deep CNN achieves 54.24% accuracy when predicting the next moves of AIs in the experiment. Meanwhile, the normal neural network can only reach 25.38% accuracy. With the promising result, we can expect Deep CNN to be applied in even more type of problems where visual or similar information is not available.

The network structure above was used as a policy of our agent in Fighting ICE environment. Thereby, our agent could get an average point 200 in matches against the AI champion of 2015. By applying reinforcement learning method to improve this policy, our agent could get an average point 250-300. By modifying the design of reward function, we increased the point to 350-400. This result was not enough to defeat the AI champion of 2015 ,since an agent can win when achieving 500 points, but it helped us have more knowledge about delay-reward in reinforcement learning.

Acknowledgements

I would like to thank my supervisor, Ikeda-sensei, who has been guiding and supporting me for the past two years. I really appreciate him for being patient with me. His understanding and patience make me have much motivation in research.

I also thank my seniors (Sato-san and Sila-san) and my friends at JAIST for helping me to adapt to daily life in Japan.

Contents

List of Figures

List of Tables

Abbreviations

CNN	**C**onvolutional **N**eural **N**etworks
SL	**S**upervised **L**earning
RL	**R**einforcement **L**earning
AI	**A**rtificial **I**ntelligence
ReLU	**Re**ctified **L**inear **U**nit
ICE	**I**ntelligent **C**omputer **E**ntertainment

Chapter 1

Introduction

AI has become important for human life since its application can help human in problem-solving. Imaging a world, when workers in dangerous environment are replaced by Robot, oldsters are taken care by automated and comfortable services, self-driving cars reduce the number of accidents etc. That wonderful world is a big dream but not impossible. Step by step, human improve AI and achieve many positive signals.

One of the early successes of human is creating AI that can defeat human player in some simple games. This success is meaningful because from the beginning of mankind history, game has been selected as a testbed of intelligence. For example in Japan, strong board game players are respected as intelligent people, about tens percent people think "Habu Yoshiharu" is one of the most intelligent men in Japan. Furthermore, game is simple and easy to understand. In game, rules are clearly defined so we can evaluate human player or AI easily by matches. Therefore, when an AI can defeat human player, even in a very simple game, we can confirm this AI is quite "smart".

As time passes, the strength of AI is increased, and it can achieve more difficult tasks. In 1997, when most of people believed that AI cannot defeat human player in chess, IBM's Deep Blue won against chess human champion Garry Kasparov [3]. Deep Blue is built based on tree search algorithm, with the computational power of computer, it can look deeper in the strategy and find a better move than human player's move.

This result is impressive because it confirms the abstract thinking of human player can be replaced by tree search.

In 2010, DeepMind, an artificial intelligence company, was founded to “solve intelligence” [4] based on machine learning. In 2013, they succeeded in training an AI to play Atari games [5] better than human player. From the beginning of the project, DeepMind thinks reinforcement learning is a good approach for AI in Atari games. Because in Atari games, every rule are fixed, the transitions between the states are unchanged. Then a game environment can be represented as a Markov Decision Process and a good agent might be made after many training step. The key of their approach is that reinforcement learning requires too much time to learn if the number of game states are huge, then some abstraction is needed to reduce the training time. They invented an algorithm named Deep Q-Network to use a deep convolutional neural network as a Q-function. And this network can represent the “good abstraction” from the game states. As the results, DeepMind’s AI can learn how to play the game based on given images from environment even it does not know the rules.

With that promising results, Google has acquired DeepMind and tried to attack the hardest problem in board games: the Game of Go. Finally, an AI named AlphaGo was created based on the same techniques of Atari AI, and self-playing method which allowed AlphaGo to improve itself, and Monte Carlo tree search. This powerful AI, which combined two cores of AI for games: tree search and machine learning, defeated human champion Lee Sedol in March 2016 [6], opens a new era for Deep Learning.

Nowadays with the explosion of data, Deep Learning has become one of the most popular research fields with its efficiency in modeling and learning from data. When we talk about Deep Learning, it means we talk about the neural network with many layers and their structure (how neurons in one layer connect to the ones in other layers). Depending on the problem, we have to choose the fittest structure to solve it. For example, in image processing, for MNIST dataset-digit number from ‘0’ to ‘9’ [1] and CIFAR 10-subset of tiny image dataset [7], Deep Convolutional Neural Network (Deep CNN) are chosen because of its ability in representing abstract feature. The convolutional layers and the connections are also designed to deal with images.

In recent years, the effect of DeepMind's impressive results makes many researchers try to apply Deep CNN in their supervised learning and reinforcement learning model. For example, in research with Doom game, Micha Kempka and his team confirm that Deep CNN also works well in 3D games [8]. In the cases above, the games are perfect-information games, and these images are available. Therefore, applying Deep CNN in such games is reasonable. But we want to apply Deep CNN to more and more target problems, so we have some research questions: In case that we don't have images input, can Deep CNN be used for all extracted-features? or just related-feature? In which way we should allocate features to achieve good performance? If we can answer all research questions above, it will be very useful to solve other problems.

In this report, we do our experiment with the fighting video game and we select Fighting ICE, an environment developed and maintained by Intelligent Computer Entertainment Lab, Ritsumeikan University [9]. In Fighting ICE environment, images of the game are not available. Instead, AIs will receive features such as hitpoint, energy level, or in-game locations as input. That information is necessary for our experimental purposes. In the first goal, we want to verify the effective of allocation of features when they are used as input in a convolutional neural network. The second goal is creating a strong AI that can win ICE competition and furthermore, defeat human champion in fighting video game.

This report has 7 chapters. In chapter 2, the basic knowledge of neural network and some related works when using supervised learning to modeling strong AI is introduced. In chapter 3, categories of Reinforcement Learning and its applications are summarized. After that, our target fighting video game and the competition ICE are shown in chapter 4. Chapter 5 introduces our approach and chapter 6 shows the detals of experimental setup and results. We also have the conclusion and plan to improve our work in chapter 7.

Chapter 2

Supervised Learning with Neural Networks

2.1 Classical Artificial Neural Network

The idea of classical artificial neural network was proposed in 1950s by Frank Rosenblatt. The first version had only feed-forward step and did not have so much meaningful. In 1980s, Paul Werbos introduced a method to train neural network with backpropagation step. Let's look more details:

2.1.1 Feed-forward

We start with the simplest example: a perceptron [10]. It takes a vector input and returns a result that help us to answer some questions or to predict something (Figure 2.1).

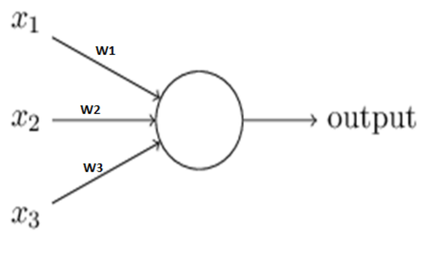


FIGURE 2.1: A perceptron with 3 inputs [1]

TABLE 2.1: Information of houses

Name	Size	Near hospital	Near school	Value
House A	500	0	0	5000
House B	400	0	1	4004
House C	200	1	1	2011
House D	300	1	0	3007

For example, we have the information of houses such as size of the house, near hospital or not, near school or not, and we want to predict the values of those houses. To do that, we have to estimates the effect of each information to the value. Assume that the size has strong effect about '10', near hospital has less effect about '7' and near school has '4'. From Table 2.1, in column "Near hospital" and "Near school", 1 means yes and 0 means no, so we can calculate the value of house B as: $(400 \times 10) + (0 \times 7) + (1 \times 4) = 4004$. Similarity, the value of house A is 5000, the value of house C is 2011 and the value of house D is 3007. With those results, we can compare the values between many houses easily.

To answer a binary question: the house is comfortable or not, a threshold value is used to determine. If the house's value is greater than the threshold, output will be 1, it means that the house is comfortable. Otherwise, the house is uncomfortable. This threshold is a parameter and can be tuned. For the middle class people, they can accept the value at 3000, then 3 of 4 houses are comfortable. But for some choosy people, they can only accept the value at 7000, then none of 4 houses are comfortable.

$$\text{Output} = \begin{cases} 1, & \text{if } \sum_i W_i \times x_i \ > \ \text{threshold} \\ 0, & \text{if } \sum_i W_i \times x_i \ \leq \ \text{threshold} \end{cases} \tag{2.1}$$

Suppose that some people have children and they want to get comfortable health care and good education for them. So, "Near hospital" and "Near school" attributes will effect so much to their decisions. Thereby, the value of weights: W_2 W_3 should be increased. By changing these parameters, many different decision-making models can be made [1]. In other words, in feed-forward step, a perceptron learns nothing, the weights are decided by users.

In the case of a single perceptron, only weighted-sum can be represented. For more complicated example, a XOR function cannot be represented. But by combining many perceptrons, it can be. Therefore, it seems better to use a network of perceptrons to build a decision-making model. This network has 3 columns of perceptrons (Figure 2.2). The first column takes information from input features and return 3 decisions, so we call it input layer. The second columns takes information from the results of input layer, therefore it can make more abstract decisions than the input layer. And we call this layer hidden layer. The last column has one output perceptron is called output layer.

Some people will be confused because in the example of perceptron in Figure 2.1, one perceptron returns only one decision - one output. But in Figure 2.2, each perceptron in input layer look like it returns many output. And the correct answer is that one perceptron returns one output, but this output will be used as an input of many perceptrons in the next layer [1].

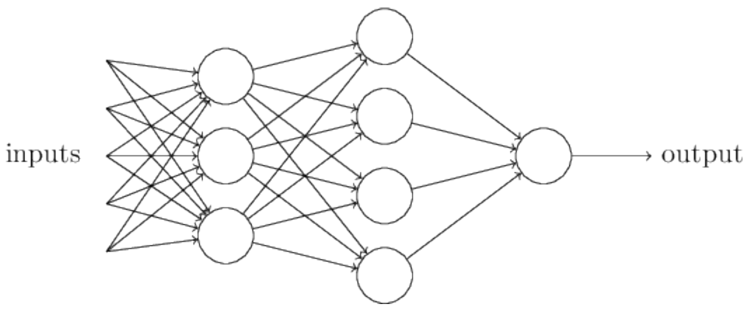


FIGURE 2.2: A simple network of perceptrons [1]

Now, we understand the concept of network of perceptrons. And the next important point is network of perceptrons with sigmoid activation function (we also call it neural network). Firstly, we start with some simple mathematics, move the threshold from right-hand side of equation 2.1 to left-hand side and names it bias: $b = -threshold$. This bias allows us to move the decision line to make a better decision. The equation 2.1 becomes:

$$\text{Output} = \begin{cases} 1, & \text{if } \sum_i W_i \times x_i + b > 0 \\ 0, & \text{if } \sum_i W_i \times x_i + b \leq 0 \end{cases} \tag{2.2}$$

From the equation 2.2, by changing the weights just a little, the final decision may be change or may be not. It's up to the value of the bias, if bias's value is too large it's very hard to change the final decision. So, we add a sigmoid function to make the network can be changed smoothly. The equation 2.2 becomes:

$$Output = \sigma(\sum_i W_i \times x_i + b) \tag{2.3}$$

In the equation 2.3, σ is the sigmoid function, which is defined by equation 2.4. By using a sigmoid activation function, when we make a small change in the weights, it will cause a small change in the output too. When $\sum_i W_i \times x_i + b$ is positive and large, the output will be approximately 1. When $\sum_i W_i \times x_i + b$ is negative and too large, the output will be approximately 0. Therefore, a sigmoid neuron has decision approximately a perceptron [1].

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.4}$$

2.1.2 Backpropagation

As we discussed in section 2.1.1, a neural network is useful for decision-making. But it has limitation: the weights can not be tuned automatically. The solution for this problem is "backpropagation" which allow the network can be updated to improve itself [11].

The whole system (neural network with "feed-forward" and "backpropagation") can be viewed as a black box, doing trial and error. It takes an input, makes a decision, and receives the feedback from a supervisor (who has already known the correct answer) to tell it whether its decision is true or not. The supervisor also forces the system update to get the correct answer. That circle continues for a while, finally the system can make a same decision as its supervisor. This framework is called supervised learning.

From the viewpoint of math, we build a cost function which measures the difference between the current decision and the expected decision. After that, we try to minimize the difference by calculating the derivative of cost function at the current weight's value and updating the weights with a small part of the derivative. This method is called

Gradient Descent. The weights are updated as follow:

$$W_i = W_i - \alpha \cdot \frac{\partial C}{\partial W_i} \tag{2.5}$$

In the formula 2.5, C is the cost function that we want to minimize, α is a parameter (is usually called learning rate), $\frac{\partial C}{\partial W_i}$ is the gradient. If the gradient is positive, it tries to tell us decrease the weight to get the minimum. If the gradient is negative, it tries to tell us increase the weight to get the minimum [12]. Therefore, in the update formula, we subtract the current weights with a small part of the gradient to expect out cost function is decreasing in each step.

In summary, a dataset with input features and "true output" (we usually call it label) are given. With feed-forward step, a network can create an output based on the input features. To make the output of network look like the "true output" we have to tune the parameter by hand. The backpropagation technique is created to make a neural network update its weights automatically by calculating the error between its current output and "true output" then tuning its parameters to minimize this error.

2.2 Modern Artificial Neural Network

Training a classical artificial neural network has many problems. As time passes, we have many knowledge and improve neural network a lot. During practices, one of the early part was improved is activation function.

2.2.1 Activation Function

As we introduced in section “feed-forward”, sigmoid function (Figure 2.3) is a standard of activation function for classical neural network. But it has weakness which is called “kill gradient” [13]. If the input value is too large or the weights are initialized too large or after some learning step, the weights become too large etc... then the output is approximately 0 or 1 and the current gradient is very small. There is almost no useful information for updating and the neurons will become “saturated”.

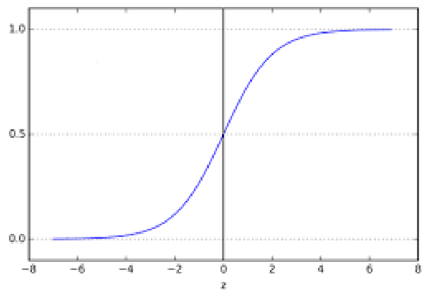


FIGURE 2.3: Sigmoid activation function

The second activation function is tanh function (Figure 2.4) which is defined by: $\tanh(x) = 2 \times sigmoid(2x) - 1$. This activation function also has “kill gradient” problem when the weights are too large, the neuron will saturate to 1 or -1. In practice, people usually prefer tanh function to sigmoid function.

The third activation function is ReLU function (Figure 2.5) which is defined as: $ReLU(x) = max(0,x)$. It involves cheap operations when compare with sigmoid or tanh. Furthermore, this function does not saturate to 0, 1 or -1. In recent years, ReLU is a popular choice for hidden layers because many researchers tried it and get good results. For example, Andrew Ng and his teammates [14] compare the performance of

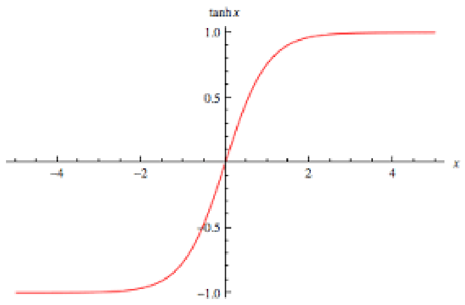


FIGURE 2.4: Tanh activation function

tanh and ReLU in a speech recognition task. They try many experiments, and in all of the experiments, the accuracy of the model with ReLU function is always better than the accuracy of the model with tanh function.

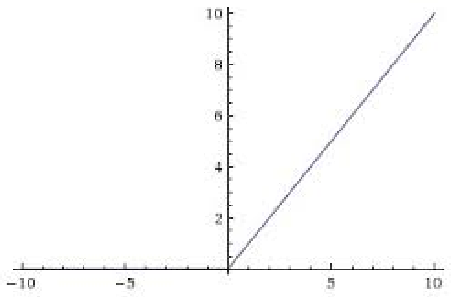


FIGURE 2.5: ReLU activation function

2.2.2 Avoid Overfitting

It is called overfitting when the model is quite fitted with training data but it can not predict well because the representation ability is too rich, the parameters are too many or the number of training data is too few etc... In other words, the model try to "memorize training data, rather than learn from it" [12]. To solve this problem, a technique called "regularization" is used. It's a penalty part added to cost function. There are 2 kinds of regularization term: L1 and L2.

L1 is defined by: $\lambda \cdot \sum_{W_i} |W_i|$ and L2 is defined by: $\lambda \cdot \sum_{W_i} W_i^2$. In those formulas, λ is a hyper parameter representing how important the regularization part means to the cost function. When we look at the formulas, we can see that those terms are not so different, L1 is the sum of absolute value of the weights and L2 is the sum of square value of the weights. But the meaning of each term is so different. When the representation ability is too rich and the parameters are too many, a L1 term tries to sparse our model thereby it reduces overfitting. In the case of L2, it tries to use small weights to optimize our network. For example, we have an input $x = [2,2,2,2,2]$ and two sets of weight $W_1 = [1,0,0,0,0]$, $W_2 = [0.2,0.2,0.2,0.2,0.2]$. The output of 2 models are equal: ${W_1}^T x = {W_2}^T x = 2$, but L2 value of them are different. The L2 value of W_1 is 1 and the L2 value of W_2 is 0.2, it mean the first weights will be penalized more than the second weights.

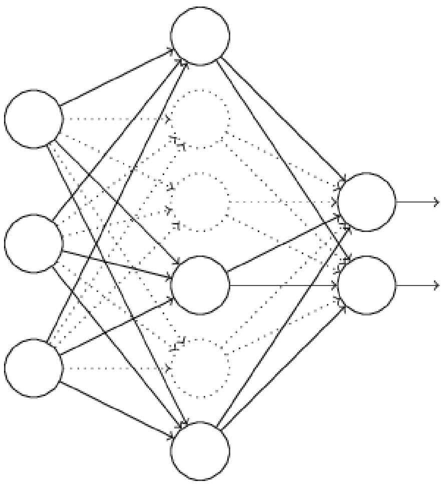


FIGURE 2.6: A network with dropout rate 50% at hidden layer [1]

One recent popular technique is "dropout". The idea of dropout is quite simple: in every step (feed-forward and backpropagation), some neurons are randomly selected and temporarily removed from the network (Figure2.6). Suppose that we have a network with 3 layers: input layers with 3 neurons, hidden layer with 6 neurons, output layer with 2 neurons. We train this network with dropout rate = 0.5 to avoid overfitting. It means that, during training process, 50% of neurons in hidden layer is randomly selected and temporarily removed. Finally, when we finish training, we have a big network contained $\frac{6!}{3!(6-3)!} = 20$ sub-networks. When making a decision, our big network does a voting task: collecting decisions from 20 sub-networks and choosing the most popular decision. This process reduces overfitting because if some sub-networks are overfitting, their decisions might be refused in voting step. For example, we train a network to classify a digit number from '0' to '9'. The big network have 5 sub-networks and 2 of them are overfitting with dataset. We test this big network with a new input, 2 overfitting sub-networks return ‘1’ but the other sub-networks return ‘7’. Thereby, our big network return a correct answer ‘7’.

2.2.3 Deep Convolutional Neural Networks

One of the disadvantages of classical neural network is that we have to represent input information as a vector. For some characteristic datasets such as image, text etc. this disadvantage is a big problem. Fukushima (1980) introduced a concept of convolutional neural network (CNN) and many researchers in Computer Vision have improved it. In this section, we introduce some basic background of CNN:

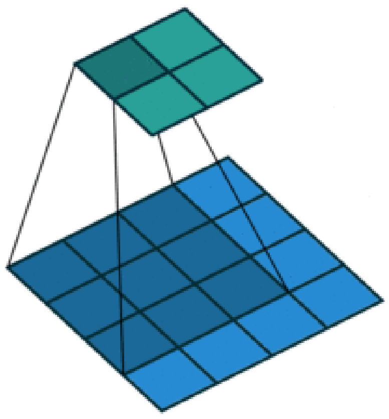


FIGURE 2.7: A window size 3x3, stride length 1, sliding through input grid (blue grid) and connect to next feature map (green grid) [2]

Firstly, we represent input layer as a 2D grid input not a vector input. This structure is useful to accept a whole image as an input without any extract feature step. Using a window slide from left to right, from top to down, with a fix stride length, we can group a number of neurons in grid input to another neurons in the next layer (Figure 2.7). The most characteristic of this window is that it uses the same weights and bias while sliding through the grid. So, all the neurons in next layer will get the same information from the previous grid, just at different locations. For this reason, the next grid is usually called a feature map. We also call the weights and bias in one window is shared-weights and shared-bias. And a pair of (shared-weights, shared-bias, stride length) is called a filter. When we change the filter, we will get a different feature map. A set of filters will create a set of feature maps. The mapping from grids to a set of feature maps using a set of filter is the structure of convolutional layer. This design (using share-weights and share-bias) reduces the number of parameters and makes a convolutional layer has ability to avoid overfitting itself.

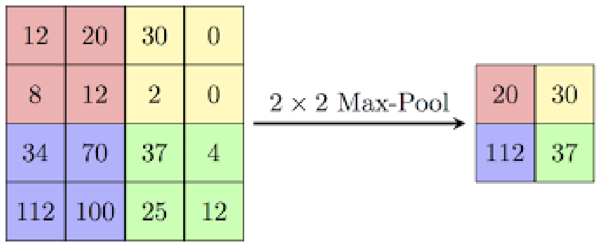


FIGURE 2.8: A max pool example with window size 2x2, stride length 2

The output of convolutional layer contains so much abstract information. If we want to simplify it, a pooling layer will be used. A pooling layer is usually put right after a convolutional layer, it does not have weights or bias. It takes a feature map as an input and simplifies this feature map by pooling technique likes max-pooling or average-pooling (Figure 2.8).

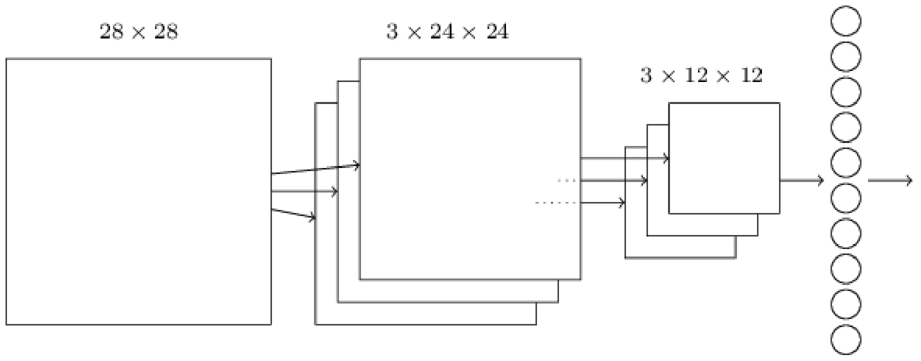


FIGURE 2.9: A complete CNN, take an image of digit number size 28x28 as an input, use 3 filters size 5x5 stride length 1, follow by one max-pooling layer and one fully-connected layer [1]

To have a complete convolutional neural networks (Figure 2.9), we put a fully-connected layer at the last to learn all abstract information from previous layer. This fully-connected layer is exactly the same as the hidden layer in classical neural network. To train a Deep CNN, the same process is used: feed-forward and then backpropagation.

2.3 Related Works

In this section, we introduce some promising results that humans have achieved by using CNN model. We start with 2 famous datasets in Computer Vision: MNIST and CIFAR-10, and continue with some related works in modeling strong AI.

2.3.1 Convolutional Neural Networks in Image Processing

The MNIST (database of handwritten digits number, has a training set of 60,000 examples, and a test set of 10,000 examples [15]) was created to call for competition from other researchers in Computer Vision. Many approaches were given: support vector machine (SVM), K-Nearest Neighbors, classical neural network etc. But CNN defeated them all when achieving a very good error rate 0.21%. And this result might be the best result because it's impossible even for human to recognize image from this datasets at 100% accuracy (a handwritten digits number from a style can be recognized as '7' might be recognized as '1' in other styles). But the success in MNIST is not the significant result of CNN in Computer Vision while other approaches can achieve an error rate 0.56%.

The CIFAR-10 dataset consists of 60,000 32×32 colour images, it is divided into training set with 50,000 images and test set with 10,000 images [16]. There are 10 classes, with 6000 images per class, and we have to build a model to achieve a good accuracy in classifying the test set. This task is similar to MNIST's task but using the same structure in MNIST cannot get a good accuracy with CIFAR-10. Because images from MNIST are black-and-white images which contained less information than colour images in CIFAR-10. Therefore, researchers reach an accuracy about 75.86% when applying naive Deep CNN model [17]. By optimizing the structure (using more hidden layers) and pre-processing the input images, they can improve the model and get a good accuracy 96.53% [17].

2.3.2 Convolutional Neural Networks in Modeling Strong AI Game Player

Modeling a strong AI game player is a hard task because it requires a huge dataset and careful optimization of many parameters. The more actions that AI can perform the more difficulty we get. In particular, in the game Go, it is very hard for researchers to model and predict the next move of the opponent. But, with Deep CNN, finally, researchers are succeeded in that hard task. Christopher Clark used 81,000 professional games dataset (about 16.5 million samples) and an average network structure (four convolutional layers, one fully-connected layer) and he gets a good accuracy 41% in predict the next move [18]. After that, DeepMind did a better work when creating the strong AI AlphaGo that defeat the human champion. In supervised learning step, AlphaGo used 13 layers, 30 million samples, training by 200 GPUs in 2 weeks and get a very good accuracy 57% [6].

Chapter 3

Reinforcement Learning

3.1 Introduction to Reinforcement Learning

Reinforcement Learning is a framework of learning policy for decision making, through trials and errors in dynamic environment. In reinforcement learning, "the learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them" [19]. In other words, an agent has to interact with environment and learns from its own experience (Figure 3.1).

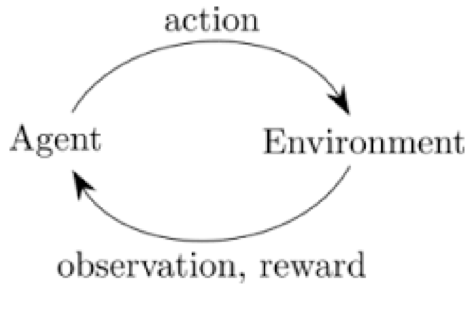


FIGURE 3.1: The agent makes a decision based on the observation that it received from the environment. After that, the environment calculates reward and feedback to the agent

3.1.1 Categorization of Reinforcement Learning Agents

A Reinforcement Learning agent may include one or more of these components: policy, value function and predictive model of transition function and reward function (we simply call it a model). A policy is the agent's behaviour function, mapping from state to action. For example, in simple maze game size 6×5 (Figure 3.2), when the agent start at the allocation $maze_{[5,0]}$, a good policy will tell it go Right to get closed to the Goal. This kind of policy is called deterministic policy. It takes a state as input and returns an output action: $a = \pi(s)$. In more difficult maze game, people can design a wind which blow from right to left. When the agent decides to go Right, it has 70% possibility to stay in next location and 30% to stay in the same location. In this case, the wind is a factor of environment: stochastic transition, and the policy is still deterministic.

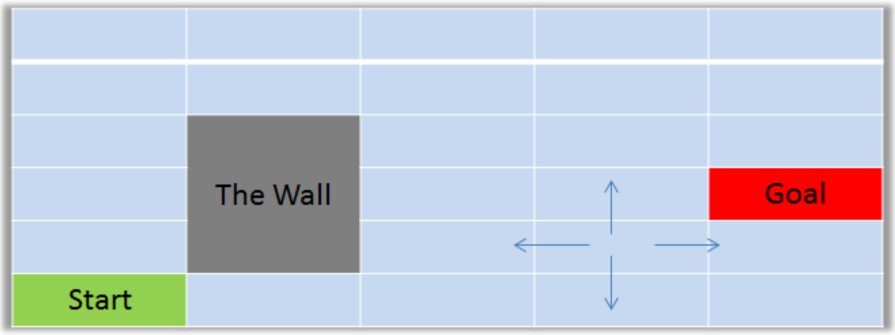
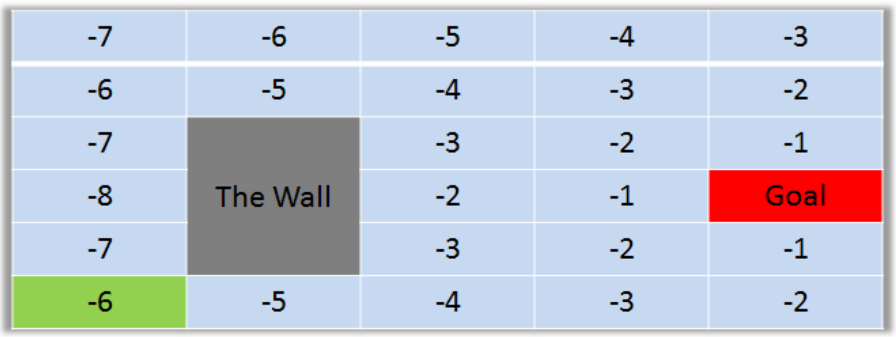


FIGURE 3.2: A simple maze game size 6×5, the agent can perform 4 actions: Up, Down, Left and Right. It start from allocation $maze_{[5,0]}$ (coloured by green) and try to go to the Goal (coloured by red). Inside the maze, there are some allocations that the agent cannot go through, which is called Wall (coloured by grey).

Value function is a prediction of future reward. It is used to evaluate the goodness/badness of the state [20]. In maze game, if we design a reward is -1 for every action we take, then the nearest allocations to the Goal will have highest value, and the farthest allocations will have lowest value (Figure 3.3).

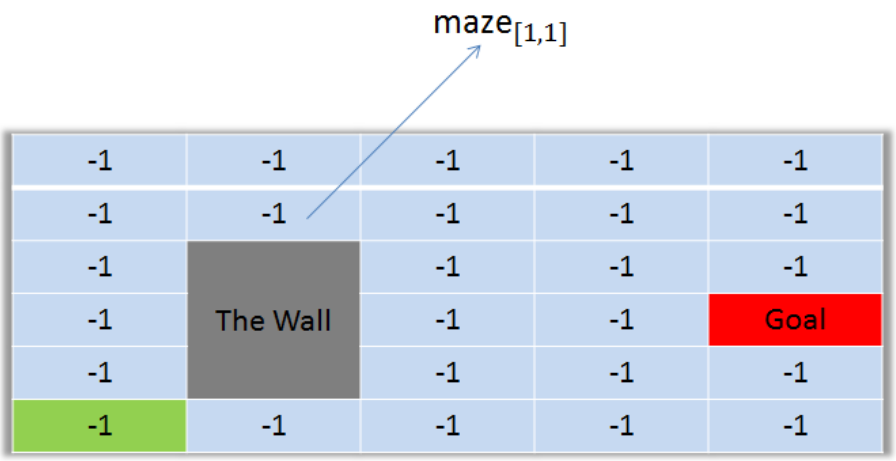
A model predicts what the environment will do next [20]. It can be used to predict the next state or the next (immediate) reward. In maze game example, if the agent stay in the state $maze_{[1,1]}$, the model can predict the next state is $maze_{[0,1]}$ or $maze_{[1,0]}$ or $maze_{[1,2]}$, and the agent cannot stay in state $maze_{[2,1]}$ because this is a Wall. As we design reward



-7	-6	-5	-4	-3
-6	-5	-4	-3	-2
-7		-3	-2	-1
-8	The Wall	-2	-1	Goal
-7		-3	-2	-1
-6	-5	-4	-3	-2

FIGURE 3.3: Designing a reward -1 per action, then the value function can evaluate the distance from each state to the Goal.

-1 for every action we take, the model can easily predict the next immediate reward is always -1 (Figure 3.4).



-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1		-1	-1	-1
-1	The Wall	-1	-1	Goal
-1		-1	-1	-1
-1	-1	-1	-1	-1

FIGURE 3.4: Designing a reward -1 per action, then the model can predict the next immediate reward is always -1. If the agent stay at $maze_{[1,1]}$ the model can predict the next state is $maze_{[0,1]}$ or $maze_{[1,0]}$ or $maze_{[1,2]}$
.

Based on 3 components: policy, value function and model, a RL agent can be categorized as [20]:

- Model-free: policy and/or value function
- Model-based: policy and/or value function and model
- Value-based: value function
- Policy-based: policy

- Actor-Critic: policy and value function

3.1.2 Markov Decision Processes

In some environments, a state, which is responded at time t+1, depends on everything that happen before. Therefore the probability of that state can be defined by the complete probability distribution in formula 3.1.

$$Pr[S_{t+1} = s'|S_0, S_1 ... S_{t-1}, S_t] \tag{3.1}$$

In other environments, when “the future is independent of the past given the present” [20] (we call it Markov property), the states capture all relevant information from the history. Therefore the probability of a future state can be defined by the give present state as formula 3.2

$$Pr[S_{t+1} = s'|S_t = s] \tag{3.2}$$

For a Markov state s and a successful transition state s', the transition probability is defined by formula 3.3. And a Markov Process (or Markov Chain) is a memoryless process, consists a set of states S and a state transition $T_{ss'}$.

$$T_{ss'} = Pr[S_{t+1} = s'|S_t = s] \tag{3.3}$$

In reinforcement learning, the agent makes its decisions as a function of the environment's state: $A(s)$ [19]. If all states in the environment are Markov, we call this environment Markov Decision Process, and we have a tuple $< S, A, T, R, \gamma >$ like this:

- S: finite set of states
- A: finite set of actions
- T: state transition probability $P^a_{ss'} = Pr[S_{t+1} = s'|S_t = s, A_t = a]$
- R: reward function $R^a_s = E[R_{t+1}|S_t = s, A_t = a]$

- γ: discount factor $\gamma \in [0,1]$. This value represent for the fact that future is unpredictable.

For example, the maze game with random wind blows from right to left, is a Markov Decision Process:

- S: set of allocations $\{maze_{[0,0]}, maze_{[0,1]}$..., wall, goal$\}$
- A: Up, Down, Left, Right
- T: from $maze_{[1,1]}$, the agent decides to go Right and it has 70% possibility to stay at $maze_{[1,2]}$, 30% possibility to stay at $maze_{[1,1]}$...etc.
- R: -1 for every action
- γ: 1

3.1.3 Q-learning

Q-learning is a model-free technique. It is used to find an optimal action-value function for a given Markov Decision Process. From definition of [20] "the action-value function $q_\pi(s,a)$ is the expected value return starting from the state s, taking action a, and then following policy π" (Formula 3.4).

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma R_{t+2} + ... + \gamma^k R_{t+k+1} | S_t = s, A_t = a] \tag{3.4}$$

And this function can be decomposed into immediate reward plus discounted of successor state as formula 3.5. This formula is called Bellman Expectation Equation.

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma\, q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \tag{3.5}$$

To solve the Bellman Expectation Equation and find the optimal action-value function $q_*(s,a)$, the following method (Q-learning) in formula 3.6 is used. The α value is called learning rate and $\alpha \in [0,1]$. In practices, small α (about 0.1 or 0.01) is usually used.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha \times (R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)) \tag{3.6}$$

For any Markov Decision Process, all optimal policies achieve the optimal action-value function $q_{\pi*}(s,a) = q_*(s,a)$ (theorem from [20]). Using Q-learning, the optimal action-value function might be found, and the agent just selects the action with the optimal q-value to achieve the expected goal.

The most important point in Q-learning algorithm is the convergence of the action-value function. To solve the formula 3.6, the agent must try some trials and errors for all pairs (state,action). If the environment is a finite and discrete Markov Decision Process, the optimal action-value function can be found. In case the environment is a continuous Markov Decision Process, Q-learning cannot guarantee action-value function to converge. To solve this problem, a new algorithm named Deep Q-Network is invented. Using the ability of CNN to learn some abstraction, action-value function can converge in some specific games. In general, Deep Q-Network also cannot guarantee the convergence.

3.2 Related Works

3.2.1 Deep Q-network in Atari and Doom Games

In 2013, DeepMind used a Deep CNN with weights w to represent an action-value function: $q(s,a,w) \approx q_\pi(s,a)$ to train their agent [5]. The results in Atari 2600 games show that Deep CNN works quite well in 2D games. Taking an $84 \times 84 \times 4$ color image as an input, the agent can learn to distinguish many different states. It also has the ability to evaluate how good a state is. As it is successful, DeepMind copyrights the algorithm as the name Deep Q-Network.

After that, in research with Doom game, Micha Kempka and his team confirm that Deep CNN also works well in 3D games. Using the same algorithm, the agent can be trained from an input image that contains depth information, and it can clear many scenarios from easy to hard smoothly [8].

Chapter 4

Fighting ICE environment

Our target is fighting video game, and we select Fighting ICE, an environment developed and maintained by Intelligent Computer Entertainment Lab, Ritsumeikan University [9].

4.1 Game Details

The competition in Fighting ICE is held annually from 2013 and attracts many competitors from other laboratories [21]. Like other fighting video games, Fighting ICE has 2 players making close-combat in a 2D arena. Each player tries to attack, avoids the skill or blocks the hit from the opponent. The game starts with player 1 (P1) from the left and player 2 (P2) from the right. In the top-left and top-right corner are their hit point (HP) and energy information. In the top-middle, there is a countdown clock from 60000 (the maximum frame) to 0 (Figure 4.1)

The fighting's rules are quite simple. Each game consists of 3 rounds, each round last 60 seconds. At the end of one round, the score will be calculated as formula 4.1. This formula is quite strange because the hit point of character is a negative number. Anyways, the organizers chose it. Winner of the round is the one who has higher score, and the player who wins more than 2 rounds is the winner of the match.

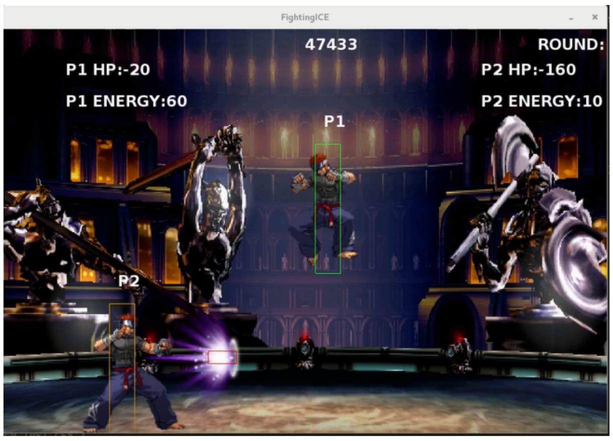


FIGURE 4.1: A screenshot taken from a match between 2 characters in a game of Fighting ICE

There are 3 kinds of character in Fighting ICE: Zen, Garnet and Lud. Each character has different interface, size and skills. But they have the same kinds of action: move, attack and guard. Each kind of actions has many sub-classes, for example: attack-air, attack-ground, guard-kick, guard-punch etc... Therefore, the total number of actions that one character can perform is 56. Some actions take short time to perform (about 3-5 frames), a few special actions take long time (about 10 frames). This setting makes the Fighting ICE more interesting and diverse.

$$Score = \frac{opponent's\ HP}{player's\ HP + opponent's\ HP} \times 1000 \tag{4.1}$$

To simulate a real time environment, Fighting ICE use 60 frames per second. In other words, one frame lasts 16.67ms. And both players will receive a delay 15-frame information. Because the organizers want to avoid the case that the player uses simple counter actions to opponent's move in previous frame. This design is also helpful for evaluating the strength of the AI in matches with human player. Because human players can receive only 24 frames per second. During the process of receiving and feedback information, human players spend more time than AIs and they have to press the button. So, delaying 15-frame makes the AI more human-like and help balancing the game play.

4.2 Recent Approaches for Competition

The first popular approach is rule-based AI. The advantage of this approach is easily for programming. With some simple conditions, you can make your fighting AI (as formula 4.2). Based on the experience of the programmer, the rules can be strong or not. In the case the programmer is also a good player in fighting game, s/he understands well about the game, and s/he knows some ’tricks’ to get an advantage state. Her/his rule-based AI may be very strong. Machete, a rule-based AI and the champion of competition in 2015, is an impressive example [21].

$$\begin{cases} \text{throw energy}, & \text{if distance} > 300 \ \& \ \text{energy} > 60 \\ \text{normal kick}, & \text{if distance} < 300 \ \& \ \text{energy} < 60 \\ \text{hard kick}, & \text{if distance} < 300 \ \& \ \text{energy} > 60 \\ \text{jump}, & \text{otherwise} \end{cases} \tag{4.2}$$

But rule-based AI has a very serious weakness. It’s easy to be countered if its rules are shown. For example, if Machete’s opponent is closed to him, he always performs a hard kick. And we know that information, so when our AI is closed to Machete, it can counter Machete’s hard kick by jumping and kick.

The second popular approach is online-learning. This kind of AI has ability to update after each game to adapt with new opponents. So, it is hardly countered. But the most disadvantage is that the number of games in a competition is limited. Then online-learning AI does not has enough time to learn and adapt. Therefore, all of the online-learning AIs are not strong enough to win Fighting ICE competition since it was held in 2013.

The third approach is tree search algorithm. This approach is very promising, by looking deeper in the strategy, a tree search AI can get a good move and a good strategy. This kind of AI are usually strong and hardly countered. But in Fighting ICE competition, the organizers want to balance the game by providing limited memory and limited computational time. So, a tree search AI cannot look deeper in the strategy anymore. However, in 2016 competition, a group of researchers tries to apply heuristics to build a Monte Carlo tree search AI [21]. Those heuristics help they cut some unnecessary branches and their AI can searches deeper in the tree. Finally, this AI is ranked 2 in the competition.

For our research, we choose the approach offline-learning based on supervised learning and reinforcement learning. We plan to use a Deep Convolutional Neural Network as a policy of the agent. If we optimize this network carefully, we could have a good AI which is more powerful than rule-based AI. This kind of AI is also hardly to be countered, and can adapt with new opponents. Comparing with a tree search AI, this offline-learning AI can be created with less memory. For all above reasons, we think this approach is very promising.

Chapter 5

Approach

Our approach is offline-learning based on machine learning. It has 2 steps: in the first step, a Deep CNN is used to modeling the actions of strong AIs player. By changing the way to represent input features and the structure of the network, a best network may be found. In the second step, the best network in previous step is used to initialize for the policy network. After that, a policy optimization method is used to update the policy to get the goal: win the game.

This approach is employed because of many reasons. Firstly, an offline-learning AI can be stronger than other AIs if we optimize the model carefully and this kind of AI requires an acceptable memory. Secondly, among many machine learning algorithm, reinforcement learning is a conventional way to create a strong game AI. We plan to use a policy-based method because the number of actions in Fighting ICE is high. It makes a value function hard to converge. So it's easier to optimize a policy than a value function in Fighting ICE. Thirdly, the best way to represent a policy is a neural network. But the performance of a classical neural network is not high enough, then we try to use Deep CNN to modeling the actions of strong AI.

5.1 Creating a Good Model by Supervised Learning

In the fighting game, we have stand-alone features which contain fixed information. For example, 'hit point' indicates the health of characters, 'energy' indicates the ability to use special skills, 'distance' may indicate safety level (the further the distance between our character and opponent's, the safer our character gets) etc. When we combine some of these features, they can be described as a strong effective in decision making. For example, if we combine 'distance' and 'energy' we can get some rules like this:

$$\begin{cases} \text{throw energy}, & \text{if distance} > 300 \ \& \ \text{energy} > 60 \\ \text{normal kick}, & \text{if distance} < 300 \ \& \ \text{energy} < 60 \\ \text{hard kick}, & \text{if distance} < 300 \ \& \ \text{energy} > 60 \\ \text{jump}, & \text{otherwise} \end{cases}$$

Such combinations as the example in Figure 5.1 are very popular in rule-based AIs. In order to make strong rules, the relationship of two (or more) features is important. If the employed rules are not so related, like "energy" and "size of character", they would make the agent act clumsily.

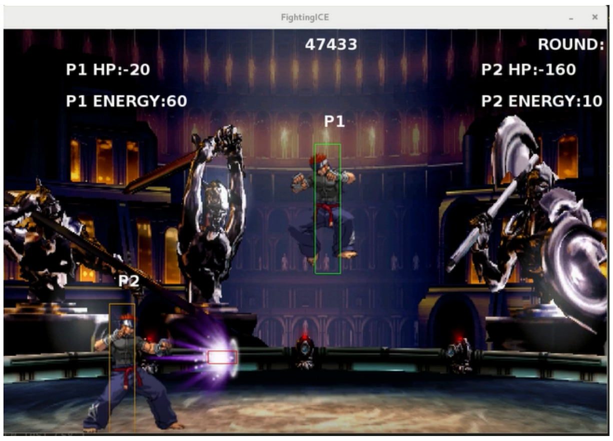


FIGURE 5.1: Player 2 (P2) is a rule-based AI, when all conditions are satisfied, it throw a energy ball to player 1 (P1).

We hypothesize that the connection between relative features in Fighting game is similar to the relation of neighboring pixels in an image. The nearer neighboring pixels have strong connections, while the further neighboring pixels have weaker or no connection to others. If grouping pixels help to improve the performance of modeling strong AI then grouping relative features could also work. Although usual neural network has the ability to combine these features if we use multiple layers, we can not control which feature would combine with others because all neurons in one layer are fully-connected with next layer. In those networks, strange connections like ’energy’ and ’size of character’ in the previous example are redundant.

By using CNN, we can reduce the numbers of ineffective connections, and put features that we need to a group. This idea can be implemented simply by:

- Represent input feature as a 2D grid
- Find some important features and put them in a group by using a window
- When sliding the window, make sure the window always contain such important feature. This leads us to duplicate the important feature.

Many experiments has been tried (in section 6.1), and an optimal network structure is found. This network can model the next moves of strong AIs with a good accuracy 54.24% and it is used to initialize a policy for next step.

5.2 Improving the Policy by Reinforcement Learning

When a good network structure is created, it can be used as a policy of the agent. But this policy is not good enough to make the agent win the fighting ICE competition. Therefore, we have to improve it to find an optimal policy by Reinforcement Learning.

The policy samples actions from a initialized distribution, and then actions taken that lead to good outcomes get encouraged, actions taken that lead to bad outcomes get discouraged. To evaluate the outcomes, a reward function f is used, it takes a input sample and returns a scalar value. Finally, we update the policy network to maximize the expected of good outcomes via Gradient Descent. This method is a policy-based method named Policy Gradients.

From the view point of math, we have sampling actions x taken from a distribution $p(x|\theta)$ parameterized by the weights θ and we want to maximize the expected of the reward function $E[f(x)]$. So, we have to "shift the distribution (through its parameters θ) to increase the scores of its samples" [22]. To do that, the derivative of the expected reward is calculated by the following process [23]:

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by p(x)} \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}$$

Once the gradient is calculated, the parameters (the weights) can be updated by Gradient Descent (Formula 5.1). In this formula, we do not subtract the current weights as formula 2.5, instead we add a small part of the gradient to the current weights. This gradient forces the distribution shift to a better region.

$$\theta = \theta + \alpha \cdot \nabla_\theta E_x[f(x)] \tag{5.1}$$

For example, we have a distribution $p(x)$ and a reward function $f(x)$ in the Figure 5.2. When x is increased, the score of $f(x)$ is also increased. Then the gradient and the reward function try to tell us shift the distribution $p(x)$ to the right-hand side to have higher possibility to get good x when sampling.

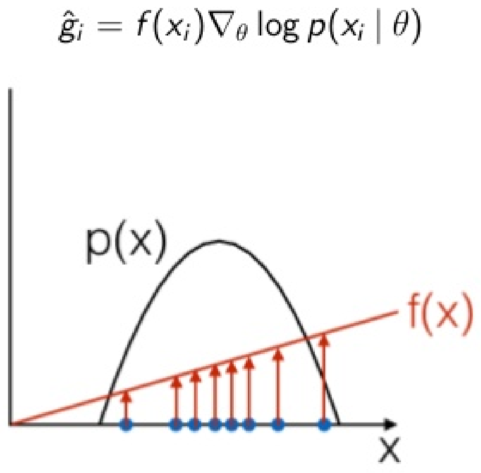


FIGURE 5.2: the gradient and reward function try to tell us shift the distribution $p(x)$ to right-hand side to have higher possibility to get good x.

The most important part is the design of reward function. In board games and Atari games, a reward function returns a scalar value $+1$ if the agent win, otherwise, it returns -1. But that design might not work well in fighting video game because the connections between states of fighting game are not strong. For example, every pair (state,action) at time t in board game effects a lot to the end-game result. If you play badly at time t, it's hard to win the game. But in a fighting game, in the first half of the match, the AI plays badly and in the second half, it plays well, so in the end-game it win. In other words, the reward in board game has long delayed-time and the one in fighting game has short delayed-time. Therefore, if we wait until the end-game to evaluate a action, this evaluation might be not good. Instead we evaluate actions after a short period of time (about 7 second). If the agent's points are increased by r in a short period of time, it means the agent plays well in this period, reward function will return positive scalar $+r$, otherwise it will return negative scalar $-r$.

Chapter 6

Experiments

6.1 Experiments with Supervised Learning

The contents of this section has been published in our paper “Optimized Non-Visual Information for Deep Neural Network in Fighting Game” [24].

6.1.1 Dataset

We collect 560 games between top 3 players of Fighting AI Competition in 2015. Each game contains 3 rounds, each round last 60 seconds, and there are 60 frames per second. In other words, we have $560 \times 3 \times 60 \times 60 = 6{,}048{,}000$ pairs of state-action. We use 70% for training and 30% for validating. From FightingICE environment, we get information from our character and the opponent's character such as hitpoint, energy, the location of characters, size of characters, etc. Totally, we have 15 features from our character and 15 features from the opponent, then we compute 5 more important relative features such as distance, difference in hitpoint, difference in energy and 2 relative positions. Using this dataset, we try to model the next move of the top 3 strong AIs.

TABLE 6.1: Summary of our experimental results. The optimized CNN structure is clearly better than others and significantly better than a usual neural network.

	input	structure	training time (min)	accuracy
usual NN one-layer	35 features	1 fully connected-100 neurons	26.61	19.45%
	35 features	1 fully connected-200 neurons	30.91	18.80%
	35 features	1 fully connected-400 neurons	45.24	20.86%
	35 features	1 fully connected-1000 neurons	83.36	22.42%
usual NN two-layer	35 features	2 fully connected-100 neurons	36.92	20.97%
	35 features	2 fully connected-200 neurons	48.68	22.08%
	35 features	2 fully connected-400 neurons	51.32	23.38%
	35 features	2 fully connected-1000 neurons	167.84	25.38%
naive CNN	5x7 grid	5 filters size 2x2-1 fully connected	54.08	29.39%
	5x7 grid	10 filters size 2x2-1 fully connected	81.66	33.93%
	5x7 grid	20 filters size 2x2-1 fully connected	123.65	33.59%
CNN with 2x35 grid	2x35 grid	5 filters size 2x10-1 fully connected	43.62	33.70%
	2x35 grid	10 filters size 2x10-1 fully connected	54.76	38.32%
	2x35 grid	20 filters size 2x10-1 fully connected	77.12	37.98%
optimized CNN	4x15 grid	5 filters size 2x5-1 fully connected	77.10	49.20%
	4x15 grid	10 filters size 2x5-1 fully connected	116.16	54.14%
	4x15 grid	20 filters size 2x5-1 fully connected	182.22	54.24%
CNN with 3x5 filter	3x15 grid	5 filters size 3x5-1 fully connected	44.58	46.29%
	3x15 grid	10 filters size 3x5-1 fully connected	55.38	49.09%
	4x15 grid	5 filters size 3x5-1 fully connected	61.5	46.22%
	4x15 grid	10 filters size 3x5-1 fully connected	86.64	49.26%
	4x15 grid	20 filters size 3x5-1 fully connected	150.16	49.95%

6.1.2 Experimental Setup and Results

Since FightingICE is written in Java, we have to write a short description in Java to get the dataset and save it in CSV format. After that, we build our neural networks in Python with supported from two famous libraries: Numpy and Theano [25]. We also use GPU GTX970 to run experiments. Table 6.1 summarizes the result of our experiments.

In the first experiment, we use a usual neural network setting with 3 layers: one input layer, one hidden layer, and one output layer. The input layer has 35 neurons (because we have 35 features), hidden layer has 100 neurons and output layer has 56 neurons (because our agent can perform 56 actions). Training this network, we get a model which can archive an accuracy of 19.45% when predicting the next move. Tuning the number of neuron in hidden layer, we get the highest accuracy is 22.42% with 1000 neurons.

In the second experiment, we use a usual neural network setting with 4 layers: one input layer, two hidden layers, and one output layer. This setting based on the well-known fact that usual two hidden layer networks have higher expressiveness than one hidden layer. Training this network, we get a model which can archive an accuracy of 25.38% (with 1000 neurons per hidden layer).

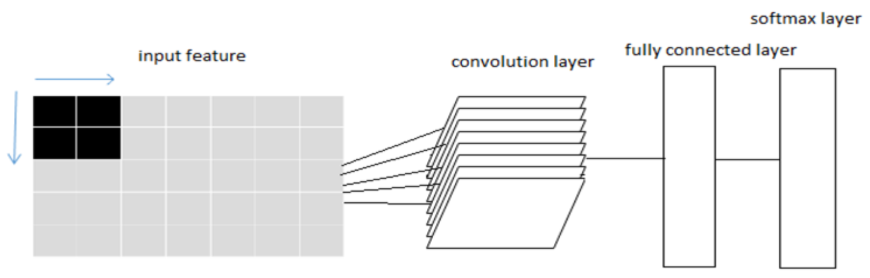


FIGURE 6.1: Naive CNN structure with 5x7 grid input, follow by one convolutional layer, one fully-connected layer and one softmax layer

In the third experiment, we use a naive Deep CNN setting: represent 35 features as a 5x7 grid input, one convolutional layer with 20 filters size 2x2 stride length 1, one fully-connected layer and one softmax layer (Figure 6.1). Training this network, we get a model which can archive an accuracy of 33.59%. It is reasonable because the location of features are select at random. Some features which are grouped together may have strong relationships. As a result, our model increased performance slightly compare with usual NN in Table 6.1

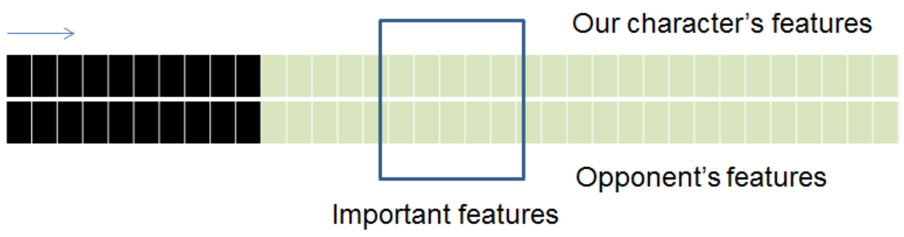


FIGURE 6.2: The structure of CNN in experiment 4. Information of our character and opponent's are separated: our information is at the top row, opponent's information is at the bottom row.

In the fourth experiment, we improve our model by separating information of our character and opponent's character (Figure 6.2). We duplicate all the features, so we have total 70 features and represent them as 2x35 grid input. The first row of the grid has 15 features from our character, 5 important features and 15 features from our character again, in this order. The second row has 15 features from opponent's character, 5 important features and 15 features from opponent's character again. We also use one convolutional layer with 20 filters size 2x10 and stride length 1, one fully-connected layer and one softmax layer. This setting lets the network compare the relationship between our information and opponent's information. It also shows that the location of features in the input grid has positive effects. When we sliding the window from left to right, the same features of both characters are accessible, allows the network to make comparisons between the states of the two characters. This structure improves the performance of our model to 37.98%.

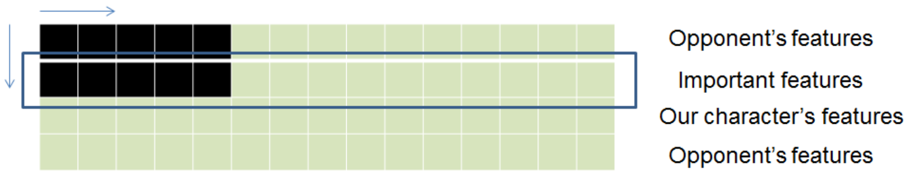


FIGURE 6.3: The input grid of the optimized CNN. Important features are duplicated 2 times and put between information of the 2 players. Opponent information is also duplicated and put at the bottom of the input grid.

In the fifth experiment, the input grid is expanded to preserve characteristic of the previous experiment while adding new potential combination. We duplicate 5 important features 2 times and the opponent's features 1 time. Totally, we have 60 features, which were represented as a 4x15 grid input. Information of our character and opponent's character are also separated in different row and the duplicated opponent's features are put at the bottom of the input grid, so that information of both characters can be combined similar to previous experiment (Figure 6.3). We use one convolutional layer with 20 filters size 2x5 and stride length 1, one fully-connected layer and one softmax layer. By using a 2x5 filter and duplicating the important features, it is possible for every filter in our CNN to have access to all 5 important features in any position in the input grid. This structure improves the performance of our model significantly to 54.24%.

In the last experiment, we try to confirm that if a bigger window can improve accuracy. First, we try a 3x15 grid input which contains the same information as the 4x15 grid input in previous experiment without the last duplicated row. Then, we paired it with a 3x5 filter which will capture information of our character, opponent and mutual information at the same time. However, the accuracy is reduced to 49.09%. Even when we duplicate the opponent's feature again, the result is only slightly improved to 49.95%.

Many allocations are tested, especially about the relationship between characters and important features, in other words about Y-axis. However, no test is shown about the relationship between features of a character, in other words about X-axis. Because changing the locations in Y-axis can increase the performance, so changing the locations in X-axis can increase the performance too. But the number of cases in Y-axis is much smaller than the number of cases in X-axis. If we focus in X-axis, it's hard to found an optimal structure. Therefore, we fixed the locations in X-axis and did many experiments in Y-axis.

6.2 Experiments with Reinforcement Learning

6.2.1 Sharing Full-Reward

As we discussed in section 5.2, the reward function evaluate actions after a short period of time, if the agent's points are increased, it will return positive scalar, otherwise, it will return negative scalar. All the actions in the sequence will be given a same reward (Figure 6.4). This reward is categorized as "full-reward" to distinguish with the one in section 6.2.2

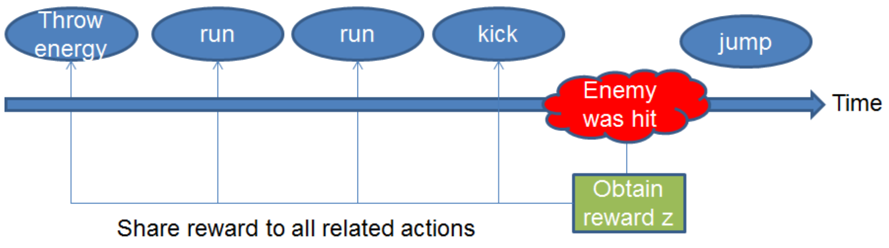


FIGURE 6.4: All the actions in a short period of time: 'throw energy', 'run', 'run' and 'kick' are received a same reward.

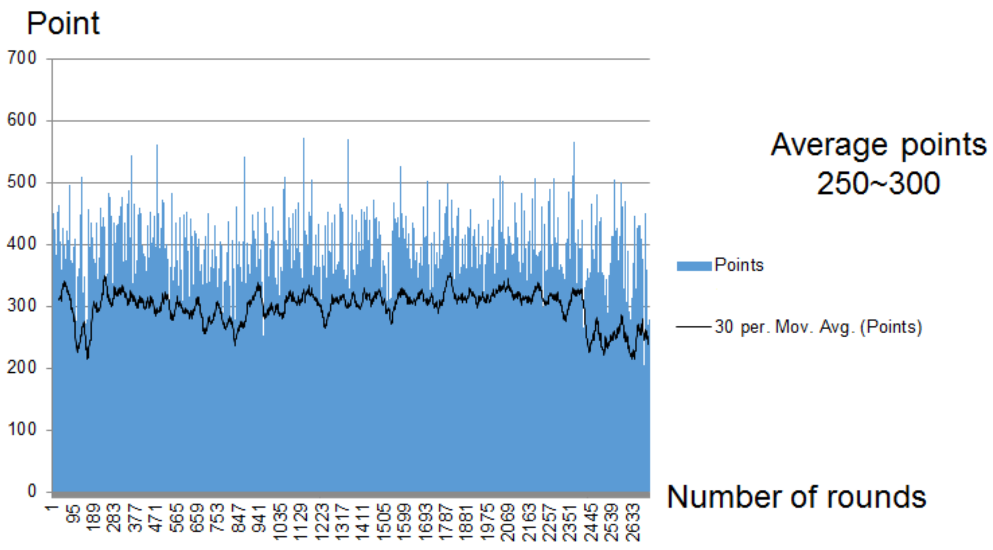


FIGURE 6.5: The x-axis shows the number of training rounds, the y-axis shows the points of our AI at the end of each round (if our points are greater than 500, we win the round). The agent learns almost nothing, and keeps running or jumping during the game. It get average points about 250 300 against the champion AI of 2015.

Training the agent with the champion AI of 2015 (Machete) for 877 games takes us about 40 hours. The policy network converges to some bad behaviors, the agent keeps running or jumping most of the time. The results are shown in Figure 6.5.

6.2.2 Sharing Decreased-Reward

In a fighting match, the number of non-damage actions are usually greater than the number of damage actions. If we share the same "full-reward" to all actions, the policy will be updated by the effect of non-damage actions. To solve this problem, a penalty is given to non-damage actions (Figure 6.6). This kind of reward is categorized as "decreased-reward".

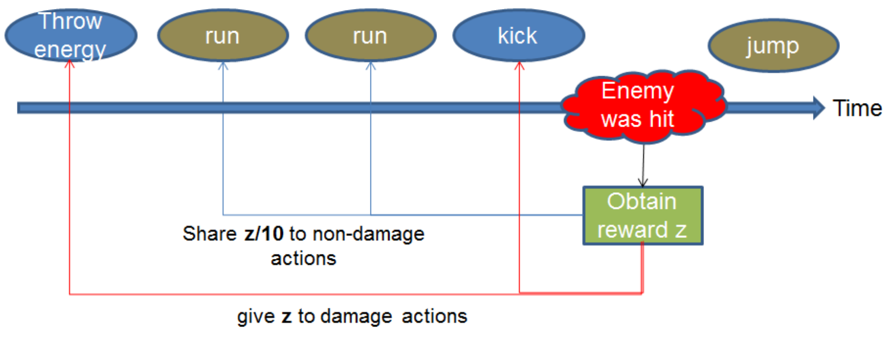

Throw energy
run
run
kick
jump
Enemy was hit
Time
Obtain reward z
Share **z/10** to non-damage actions
give **z** to damage actions

FIGURE 6.6: In a short period of time, all damage actions: 'throw' and 'kick' are received a full reward, all non-damage actions: 'run' and 'run' are received a decreased-reward.

Training the agent with the champion AI of 2015 for 1247 games takes us about 62.5 hours. The agent tried to learn many actions and get better results than previous experiment in section 6.2.1. The results are shown in Figure 6.7.

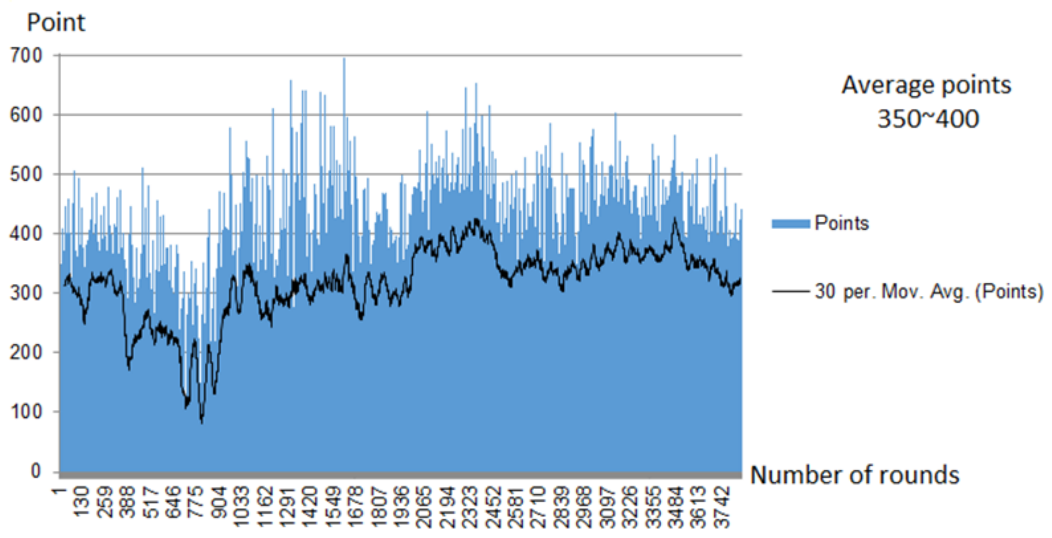


FIGURE 6.7: The x-axis shows the number of training rounds, the y-axis shows the points of our AI at the end of each round. The agent tries to learn many skill. It gets average points about 350 400 against the champion AI of 2015. This result is better than experiment with "full-reward" but still not sufficient to win the game.

Chapter 7

Conclusion

In this report, supervised and reinforcement learning with convolutional neural network (CNN) are applied to the domain of fighting game, where several features instead of image are given as input.

In supervised learning step, a method has been described to successfully incorporate Deep CNN with optimized non-visual information. We investigated the allocation of features are important and valuable for improving its performance. By intentionally arranging features as an 2D grid, with some duplication of features and well-considered allocation, Deep CNN achieves 54.24% accuracy when predicting the next moves of AIs in the experiment. Meanwhile, the normal neural network can only reach 25.38% accuracy. With the promising result, we can expect Deep CNN to be applied in even more type of problems where visual or similar information is not available.

In reinforcement learning step, some ideas have been tried to improve the design of reward function. The agent's point is increased but it's still not enough to win the champion of 2015. The first reason is that the number of training games is too small (877 games for "full-reward" experiment and 1247 games for "decreased-reward" experiment). Usually, for training an agent via reinforcement learning, thousand games are used. In case of Pong game, a simple game with 2 actions, the agent has been trained for 7,000 games [22]. In our situation, it's very hard to train our AI with a large number of games. Because the number of actions in Fighting ICE is 26 times compare with

Pong game, and each game in Fighting ICE last 3 minutes, so we need at least 145 days to make a good AI.

The second reason is that the organizers of ICE design the delay-frames (we discussed about it in section 4.1). At this time, we do not have much knowledge about the effect of delay information to the learning process of reinforcement learning. It might be a big problem.

For future work, we plan to create an similar environment with Fighting ICE and add a speed-up mode to the environment. This additional mode will help us decrease the learning time. We will also implement a new ruled-based AI which has the same strength with the champion of 2015. After training in new environment with new opponent, the agent will be evaluated in Fighting ICE competition.

Appendix A

Features are used in Fighting ICE

In this appendix section, we show more details of our experiments. Each character will be represented in our network by 15 features:

- Hitpoint
- Energy
- Character's size: width and height
- Character's hitbox coordinates: left, right, bottom and top
- Remaining frame of current action
- Type of current action
- Current speed of character: in x-axis and in y-axis
- Character's current facing direction
- Character's current state: can be controlled or not
- Information about projectiles in current state: quantity and relative position to the character (in front or behind)

The 5 important relative features:

- Distance between 2 characters
- Difference in hit point
- Difference in energy
- Difference of position in x-axis
- Difference of position in y-axis

Bibliography

[1] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL `http://neuralnetworksanddeeplearning.com/index.html`.

[2] Convolution arithmetic. `https://github.com/vdumoulin/conv_arithmetic`, 2017. Accessed: 2017-01-30.

[3] Murray Campbell, A Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[4] Deepmind website. `https://deepmind.com/`, 2017. Accessed: 2017-01-25.

[5] 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.

[6] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[8] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. *arXiv preprint arXiv:1605.02097*, 2016.

[9] Feiyu Lu, Kaito Yamamoto, Luis H Nomura, Syunsuke Mizuno, YoungMin Lee, and Ruck Thawonmas. Fighting game artificial intelligence competition platform. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, pages 320–323. IEEE, 2013.

[10] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[11] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.

[12] Jeff Heaton. *Artificial Intelligence for Humans, Volume 3: Neural Networks and Deep Learning*. CreateSpace Independent Publishing Platform, 2015. ISBN 978-1505714340.

[13] Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/`, 2017. Accessed: 2017-01-30.

[14] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[15] Mnist dataset. `http://yann.lecun.com/exdb/mnist/`, 2017. Accessed: 2017-01-30.

[16] Cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2017. Accessed: 2017-01-30.

[17] Classification datasets results. `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`, 2017. Accessed: 2017-01-30.

[18] Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*, 2014.

[19] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction, 2011.

[20] Ucl course on rl. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`, 2017. Accessed: 2017-01-30.

[21] Fighting ice competition website. `http://www.ice.ci.ritsumei.ac.jp/~ftgaic/index.htm`, 2017. Accessed: 2017-01-30.

[22] Andrej Karpathy. Deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl/`, 2016. Accessed: 2017-01-25.

[23] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

[24] Nguyen Duc Tang Tri, Vu Quang, and Kokolo Ikeda. Optimized non-visual information for deep neural network in fighting game. *ICAART*, 2017.

[25] Theano website. `http://deeplearning.net/software/theano/`, 2017. Accessed: 2017-01-30.