Using Deep Autoencoders to reduce state space of RTS games

Leonardo Rosa Amado
Pontificia Universadade Catolica do Rio grande do Sul (PUCRS)
Email: leonardo.amado@acad.pucrs.br

Abstract

Real-time Strategy (RTS) games are a well known interesting case of study for the application of intelligent agents. However, applying artificial intelligence techniques on RTS games can be a challenging task, due to the immense state space and the immense branch factor. In this paper, we present a possible solution for the state space problem, using Neural Networks. We developed a Deep Neural Network, that work as an Autoencoder capable of encoding the state space. We apply this solution to states of the MicroRTS, an open-source abstraction of a full fledged RTS game.

Introduction

Games are well known for being an interesting case of study for the application of intelligent agents. RTS (Real-time Strategy) games are the most used type of game to research AI due to their complexion, and some them have a real strong competitive scene. However RTS games prove to be too complex. Research shows that (Ontan et al. 2013.) the branching factor of possible actions, for the popular RTS game, Starcraft, can reach 10^{50}.

Most modern games have an AI (Artificial Intelligence) built in. However, in most games, AI is purely scripted, making its actions predictable. If a player can easily predict the AI actions, it can *exploit* the AI. For exploits, we mean the ability of a player to predict an behavior of the AI, and use this knowledge to his advantage. To avoid exploits, an AI should be able to *learn from experience*, like a human player would do. Reinforcement Learning is an AI technique that enforces this idea of learning from experience.

However, reinforcement learning algorithms, such as Q-Learning, requires exploration of the entire state space. In RTS games, the state space tend to be gigantic due to the massive number of information contained. Therefore, exploring the entire state space can take too long, resulting in slow convergence for the algorithm. There is a type of Deep Neural Network capable of encoding a sequence of bytes to a smaller sequence, with little loss of representation. This type of network is called Autoencoders.

In this paper, we demonstrate the configuration of a Deep Autoencoder to reduce the state space, using the Java library *deeplearning4j*[1]. First we explain the basics of a RTS game. Then, we explain the how the game we choose, MicroRTS, works. Then, we provide a brief background on Neural networks and Deep Autoencoders. Finally, we present our Deep Autoencoder configuration and the experiments we evaluated. We conclude our paper discussing future work and possible improvements to our approach.

Real-Time Strategy Games

Real-time Strategy (RTS) games are complex domains that tries to simulate a scenario of war between two factions. Normally played by two players, one against the other, RTS games are complex because the huge amount of units to control and large amount of actions to take at each moment. As the name suggest, RTS games are not based on turns, which means that both players can perform actions simultaneously. Also, it is possible to issue an order to multiple units at the same time, increasing even more the complexity of those domains. Due to those characteristics, building AIs for those games are a well known challenge. One option is to create a purely scripted AI player, which only follows the actions scripted to each state. This type of approach makes the AI susceptible to *exploits*. Modern games, mainly Starcraft, are being subject of study to create complex AI implementations.

The main objective of a RTS game is to destroy the enemy base, leaving the enemy with no way to rebuilt its base. In most RTS games, the player starts with a base, a supply of resources to be harvested (minerals, trees, gas), and workers that can harvest resources and build structures. Starting with only those workers, the objective is to build an army. Workers can build structures, like barracks, that can train military units to fight. When a player is ready to fight, he sends his units to find the enemy base. In Starcraft, for example, there are a vast number of buildings for each of 3 factions that can build different units, and allow workers to create other builds. The possibility of armies are vast, and there are multiple ways of playing the game.

Due to the complexity of Starcraft, a strong competitive scene was built around it. Starcraft was launched on March 31, 1998, and its competitive scene survived through 12 years, when a sequel was released. This game is still one of

[1] http://deeplearning4j.org

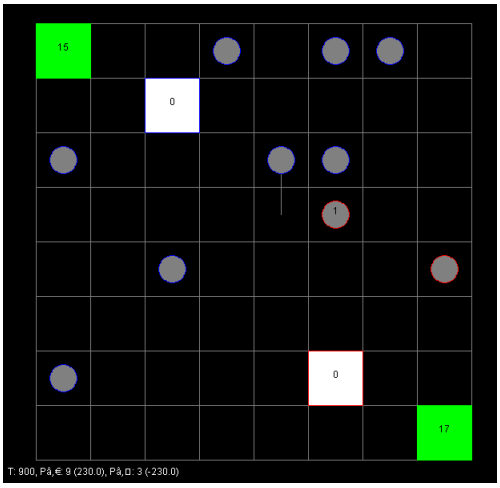

Figure 1: MicroRTS game state.

the most used subjects of study for developing AI for RTS games, and is still a remarkable challenge. However, due to the complexity of the game, testing theoretical ideas can be troublesome and time consuming.

MicroRTS was created with the intend to be a simple tool to test theoretical ideas before implementing them um full-fledged RTS games, like Starcraft. Programmed in Java, MicroRTS is simple, but preserves the main components of a RTS game.

MicroRTS

MicroRTS is a simple implementation of a Real-time Strategy game, design for the solely purpose of AI research. Developed by Santiago Ontan, MicroRTS is a well structured implementation of an RTS game in Java. The advantage with respect to using a full-fledged game like Starcraft, and the main reason we choose MicroRTS, is the fact that MicroRTS is much simpler, becoming a useful tool to quickly test theoretical ideas, before trying on to full-fledged RTS games.

MicroRTS consists of two players trying to eliminate every single structure and unit of the enemy. Figure 1 demonstrates a game state of the MicroRTS. There are 4 types of units in MicroRTS:

1. **Worker**. This unit is responsible for harvesting minerals and constructing structures. This unit can also fight, but does very little damage.
2. **Light**. Light units does little damage but are extremely fast. This type of unit can only attack.
3. **Ranged**. Ranged units are capable of ranged attacks. They have moderate damage and moderate speed. This unit can only attack.
4. **Hard**. Hard units are heavy attack based units. They do high damage but are extremely slow. This unit can only attack.

Units requires resources to be produced, but more than that, they require structures. There are 3 type of structures in the MicroRTS. Those are:

- **Base**. The main structure. This structure is responsible for producing workers. This structure is also were the workers return the minerals they harvested. The game starts with a base for each player. Can be attacked.
- **Barracks**. Auxiliary structure. This structure is responsible for producing light, ranged, and heavy units. It can be built by the workers by using resources. Can be attacked.
- **Minerals**. Minerals can be harvested by the workers to obtain resources. Not an exactly structure, can not be attacked. Minerals are finite, and each player starts with one source.

The game is totally observable, so you can see every enemy action. With those components, MicroRTS provides a good simplification of a full-fledged RTS games.

Deep Auto Encoder

A Deep Autoencoder is one of the multiple structures of the Deep Neural Networks. In this section, first we provide a briefly background of Artificial Neural Netwokrs (ANN). Then, we explain with more depth the architecture of a Deep Autoencoder.

Artificial Neural Networks

Artificial Neural Networks are a group of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. ANNs are composed of multiple nodes, organized in layers. A node is where the computation happens, it combines input from the data with a set of coefficients (or weights) that amplifies our suppress a input. An ANN is composed three types of layers:

- An **Input layer** responsible for receiving the signal (data) that feeds the neural network.
- A **Hidden layer** responsible for receiving the signal from the input layer. This layer is not always needed.
- A **Output layer** responsible for receiving the signal from the hidden layer (or input if there is no hidden layer in the network) and producing the output of the network. For example, in a classification problem, this layer will output which class the data belongs too.

A Deep Neural Network (DNN) is an ANN with multiple hidden layers between the input and the output layers. There are several types of DNNs. Those types are separated based on the architecture of those networks. One of those types, are the Autoencoders.

Deep Autoencoder architecture

A Deep Autoencoder is composed of two symmetrical neural networks connected. Usually, those networks are Deep-believe networks (DBN), using as layers restricted Boltzmann machines (RBM). RBM is a generative stochastic artificial neural network that, using a set of inputs, can learn a probability distribution.

The DBNs are connected as Figure 2 shows. The first DBN work as a encoder, encoding the input signal to a

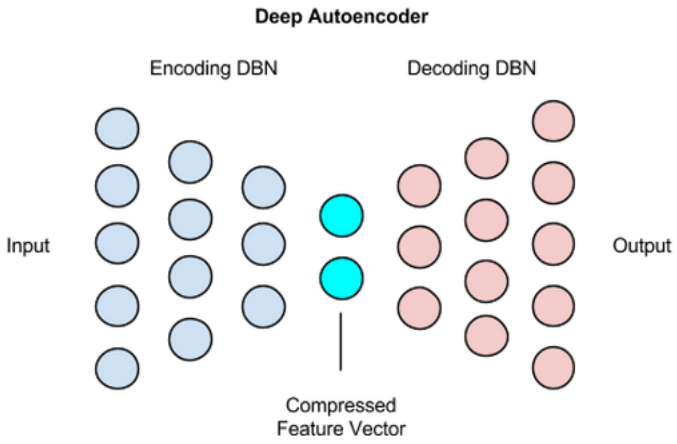

Figure 2: Deep Autoencoder architecture.

smaller representation. The second DBN receives the output of the first DBN as a input. From the smaller representation, the decoder DBN has to incrementally transform the encoded data back to the original form.

Suppose the encoding of an image that has 784 pixels (a 28x28 image). On this architecture of Deep Autoencoders, we are only considering one input signal, so the image has to be on grayscale. The image is fed to the neural network as an array of binary values, where each pixel is fed to one of the input nodes. A sketch of the encoder is determined as follows:

$$784(input) \rightarrow 1000 \rightarrow 500 \rightarrow 250 \rightarrow 100 \rightarrow 30$$

In this sketch, the first hidden layer has more nodes than the actual image input. This is necessary because normal RBMs works with only binary values, creating the necessity of increasing the representation in the case of real values. To represent the decoder, we provide the following sketch:

$$30 \rightarrow 100 \rightarrow 250 \rightarrow 500 \rightarrow 1000 \rightarrow 784(output)$$

The decoder is not needed to our implementation, we aim only to encode the state representation, with no necessity of decoding it after. However, to train the Deep Autoencoder, it is necessary to train the neural network with both the encoder and the decoder network. This is necessary because to train the network, you have to pass something as input and the expected output (in a classification problem, the data and the class that the data belongs). Since it is impossible to know the expected output of a encoder (we want the encoder to solve this problem), we train the network to return the exact input as output. After training the Autoencoder, we can retrieve the compressed feature vector from the hidden layer in the middle of the network, as shown in Figure 2.

State Encoding

In Section , we discussed about the limitations of representation when using Restricted Boltzmann Machines. Non-Gaussian RBMs can only operate with binary values, leading to the necessity of either converting the input to binary, or creating a binary representation of the input. To solve this problem, we present a naive representation of the MicroRTS using a byte array.

As shown in Figure 1, the grid of the MicroRTS game is usually 8x8. In each position, only one unit can stand at time. As discussed before, there are multiple types of units, and each unit belongs to one player. Being able to only use zeros and ones, we must encode the state of the MicroRTS game. To simplify this task, we explain how we managed to encode the grid using only binary values.

To encode the grid, we transform it in a 64 byte array. We use the value 1 to represent that there is an unit in the position represented in the array, and 0 to represent the absence of an unit. However, with this representation we are not considering the player that the unit belongs to. To solve this problem, we propose seeding the neural network with more than just one grid. The following matrix represents the input given to represent the blue player units in Figure 1, a similar matrix is also built to represent the red player units.

$$Blue = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Both matrix are merged to create the state representation. The merged matrix is converted to a byte array to be fed to the Deep Autoencoder. This representation however does not cover every aspect of the game, it is necessary a representation with a higher number of features. However, to test the Autoencoder, this representation is enough.

Experiments

To reduce the state representation, we fed the input obtained from executing the MicroRTS and encoding it to the neural network. The state representation ~~have~~ 128 features, so the input layer of the encoder must have 128 nodes to receive the signal. For the encoding output, we experiment with the transformation to a 20 feature representation. The sketch of the Deep Autoencoder can be described as follows:

$$125(input) \rightarrow 100 \rightarrow 60 \rightarrow 50 \rightarrow 20(output)$$

In this structure, we do not use a hidden layer bigger than the input layer, because our data is already binary, having no need to represent real valued data. Having structured our Deep Autoencoder, we train the network with the states generated by the MicroRTS. We first train the encoder with 20 states that have only units of the blue player. Those states are relatively similar, so we expected a high number of states to be equal when encoded. Then we trained the network with 20 states containing only units from the red player. We expected similar results for this training, since the only change was the position of in the signal array. Finally, we trained the network with 20 states containing units from both players, where we expect to have fewer repeated states. After training the network, we test the network by executing different

Table 1: Autoencoder Experiment

	Training	Test Enconding	Repeated States
Blue Units	50	100	52
Red Units	50	100	49
Mixed Units	50	100	18

states and verifying the encoded representation resulted. We encode 100 states in each experiment, verifying how many states where equals at the end of the encoding.

In Table 1, we present the results after training and testing the network. The results are similar to what we expected, however the number of similar states is still to high. This may be because of the low number of training samples, or the state representation provided being too simplistic. However, we believe that we managed to show that our Deep Autoencoder is capable of compressing the state space. To experiment further, we are still trying to implement a version of the network capable of executing in GPU, since it takes to long to train using a personal CPU.

Conclusion and Future work

In this work, we described how we built a Deep Neural Network capable of acting as a encoder for the states of the MicroRTS games. We trained the network with different training sets, and we were able to show that it is possible to reduce the state space by applying this technique. We tested the possibility of fully integrating the network with the MicroRTS, the training can be done offline and the encoding online, since after trained, executing the encode of the a state is extremely fast.

For future work, we present the following topics:

1. A full integration with MicroRTS.
2. Better state encoding.
3. Stacked Denoising Autoencoders.
4. Image feeding.

First, we would like to experiment the impacts of the encoding when executing the algorithm Q-Learning on the MicroRTS game. The *deeplearning4j* Java library provides the tools to export and load the parameters and configurations of a pre-built network. This would be ideal, however, this method of the library seems not to be working properly, returning all the weights of the network to be zero. We still need to study if there is something we can do about this bug, or we need to use other libraries for deep learning, such as *Theano*[2] and *TensorFlow*[3].

Second, we would like to create a better state representation to feed to the network. The actual representation is very simplistic, and there is only purpose to test if the autoencoder is capable of reducing the state space. For a better representation, we may need to use Gaussian Restricted Boltzmann Machines, so we can work with real valued number. This would help to represent numeric values, such as resources of each player. Still, we must take some precautions with the representation, since the numeric value must the mathematically relevant, so the network can produce good encoding.

Third, the *deeplearning4j* recently added tutorials to build a Stacked Denoising Autoencoders. This type of auto encoder is more expressive, and usually yield better results. This type of encoder use Gaussian transformation, enabling the use of real valued data. However, due to their complexity, they take even longer to train.

Finally, we would like to test the viability of image encoding to the network. With this, we would map the actual state of the MicroRTS game as the encoding of the actual game screen. We did some research on this topic, and already found some difficulties with this approach. First, the images fed to the Deep Autoencoder are all gray scale, so the data can be converted to binary. This means that the pixel of an image has only one value, the gray scale of the pixel. In the MicroRTS, the player controlling the unit is represented by the color (red or blue), making the use of gray scale unfeasible. To solve this we can feed the network using 3 signals, one for each color (Red, Green and Blue). However, it would be necessary to change the structure of the Autoencoder. Second, the screen capture is somewhat noise and the image is big. This could lead to poor training of the network, and slow training and encoding.

Even though there are still a lot of work to be done, we believe we managed to show the ability of a Deep Autoencoder, using two Deep Belief Networks, to reduce the state space of the MicroRTS game. This would significant aid the convergence of the implementation of a Distributed version of Q-Learning that we are working on. Deep Autoencoders may be complex and difficult to validate, but we believe they can be of great aid to combinatorial explosion of Real-Time Strategy games.

[2]http://deeplearning.net/software/theano/

[3]http://www.tensorflow.org

References

Santiago Ontan (2013) The Combinatorial Multi-Armed Bandit Problem and its Application to Real-Time Strategy Games, *In AIIDE 2013.* pp. 58 - 64.

Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Pearson Education, 2 edition (2003), Upper Saddle River, New Jersey: Prentice Hall, *ISBN 0-13-790395-2.*

Vincent, H. Larochelle Y. Bengio and P.A. Manzagol, Extracting and Composing Robust Features with Denoising Autoencoders, Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML08), pages 1096 - 1103, ACM, 2008.

Bengio, P. Lamblin, D. Popovici and H. Larochelle, Greedy Layer-Wise Training of Deep Networks, in Advances in Neural Information Processing Systems 19 (NIPS06), pages 153-160, MIT Press 2007.

Bengio, Learning deep architectures for AI, Foundations and Trends in Machine Learning 1(2) pages 1-127.