Survey of Deep Neural Networks Handling Plan Development Tasks Using Simulations of Real-World Environments

by

Kaleb Askren

A thesis presented to the Honors College of Middle Tennessee State University in partial fulfillment of the requirements for graduation from the University Honors College

(Fall 2019)

Survey of Deep Neural Networks Handling Plan Development Tasks Using Simulations of Real-World Environments

By
Kaleb Askren

APPROVED:

Dr. Sal Barbosa
Dept. of Computer Science

Dr. Medha Sarkar
Dept. of Computer Science

Dr. John Vile, Dean
University Honors College

Special thanks to Dr. Sal Barbosa for advising me throughout this project and to the University Honors College of MTSU for providing this opportunity.

Abstract

Neural networks are computational models that demonstrate the capability of advanced computing applications in plan development tasks. Five influential projects that independently demonstrate various applications of neural network models include AlphaGO/AlphaGO Zero, Playing Atari with Deep Reinforcement Learning, NeuroChess, OpenAI Five, and Playing Checkers without Human Expertise. Each of these projects includes different approaches to plan development tasks and are surveyed in this thesis using four criteria: the efficiency of the system, the form of their input based on their target environment, the structure of the neural network(s), and the processes through which they are trained. Each project approaches the plan development problem with strict regard to the environment in which they are targeting and thus vary in implementation. This survey is a collection of the details regarding each project and how the research teams approached their development.

Table of Contents

Introduction

Throughout the evolution of computing, research has aimed to create a computational model that is capable of handling complex analytical tasks similar to what humans do. One design that has been implemented in thousands of applications to solve this problem is the Artificial Neural Network model. Neural Networks are composed of model neurons, also known as nodes, that are connected by arcs. An arc is a connection from one node to the next that may contain information including weights. Weights are modified after each iteration in which new input values are presented. By repeatedly modifying weights, the network can begin to accurately identify patterns in input. This phase is known as training the network. Examples of the most commonly used input forms include pixel values and audio information. An agent is a program or source that utilizes the neural network. As neural network research continues to grow and the utility of these computational models become more applicable in more environments, advancements continue to be made in the effectiveness of their application. Although the range of neural network applications is broad, the structures and designs they use share similar characteristics.

The structure of a neural network is split into three layers: input, hidden, and output (Hinton 145). The input layer can be composed of one or more input nodes depending on the type of input that the network is expecting to receive. The input layer is also responsible for passing data into the hidden layer(s). The hidden layers handle the majority of the processing of the input from the input layer. The data that these layers receive may or may not be modified from the original input depending on the application of the neural network. The processing defined in the hidden layers is usually kept private and protected. This processing is done by the activation function. Also referred to as input/output functions, activation functions are typically classified as linear, threshold, or sigmoid (Hinton 145). The names refer to the type of processing that the network performs on input data which may vary depending on the implementation.   The results of this function are then passed from the hidden layer to the output layer (Hinton 145). The output layer is similar to the input layer in that it is tuned to represent the desired output format (Hinton 146). The structure of a neural network allows for a versatile and maintainable framework that is capable of continuous modification and change. By using a process to continually provide input and subsequently alter the weights, the output can change over time to be more accurate in comparison to known data sets (Krogh 195). While the structure of most neural networks is similar, the way they process information can vary depending on the application.

Some of the most common types of artificial neural networks include feedforward, recurrent, and convolutional. A feedforward neural network is one of the simplest forms of neural networks. This processing model linearly maps an input value to an output value. This means that feedforward neural networks do not allow cycles between nodes (Schmidhuber 4). This form is one of the most classic forms of neural networks and typically is used with a threshold-based processing technique. A threshold is a value in a function used for comparison with incoming input within the network (Schmidhuber 6). The output from the node depends on whether the input exceeds the threshold value and, if not, the node typically passes on a negative value such as a -1. However, not all problems can be solved using fixed-length input and output: the primary drawback of feedforward neural networks. In this case, a modified type of feedforward neural network referred to as a recurrent neural network is used.

Recurrent neural networks solve the fixed-length input problem by taking previous output or hidden states as input (Mallya 4). This is useful because intermediate states can store information on past input over several iterations (Mallya 11). Recurrent neural networks are most often used with time-series data. These models can also create copies of their cells over time, also known as unfolding, with different inputs at different time steps (Mallya 11). Sometimes, the process of training a recurrent neural network with many duplicated cells can be difficult. In these cases, the network is treated as a single large feed-forward neural network. Recurrent neural networks are also different than most basic feedforward neural networks in that classification algorithms can also be integrated to use previous strings of input to affect output (Venkatachalam). Instead of only using the input vector, recurrent neural networks can also create influence based on a hidden vector that contains data on prior input/output (Venkatachalam). This is in contrast to a basic neural network that is trained on a data set and is fixed in application. This also allows for the possibility for output to change after strings of input have been passed between two identical input values (Venkatachalam). Another commonly used type of neural network is the convolutional neural network.

Convolutional neural networks are most commonly used in image classification solutions and utilize a structure in which every node is fully connected. This refers to the connections of nodes in one layer in the network to the nodes of the next layer. Nodes within each layer have connections to every node within the next layer. An issue that is common with this structure is that these networks commonly overfit data (Saha). This means that analysis corresponds exactly or very closely to the data set. In this event, the network may perform well or even exceptionally well on training data but is unable to reliably produce accurate output on test data. In this case, a regularization process is used. Each node in a convolutional neural network responds to information from a specific region of the data set. This is known as a receptive field. A collection of receptive fields represents the entire data set (Saha). Because an image is simply a matrix of pixel values, image classification is a popular use for this computational model. Within the structure of a convolutional neural network are convolution layers. In these layers, the convolution operation is performed by the kernel (Saha). The kernel is a process that selects a region in the image and after the convolution operation is completed on the selected region, it shifts across by a degree that is known as a stride length (Saha). The kernel continuously performs the operation over the entire data set. After the convolutional layer is completed the network shifts to the pooling layer. This layer is responsible for reducing the size of the resultant data from the convolutional layer and increase efficiency (Saha). Once the data has been reduced or flattened, it becomes manageable enough to be processed through a normal neural network for classification (Saha).

While neural networks have the potential to be effective at classifying data, they must first be conditioned on sample data sets in order to learn the patterns within them. This is referred to as training. In the study of this process, there are two major types of learning: supervised and unsupervised. Both types of training have benefits specific to the type of neural network that is being trained. Two important terms for the study of these processes is variance and bias within the data set. The variance refers to the difference between individual items while the bias the error from false assumptions (Webb 100).

Supervised learning is the process of using filtered input data for the neural network to process. This is most commonly done by selecting values individually in the data set. For use of this process, the user must define the input as well as the desired output. While the data set must also be representative of the target environment in which the network will operate, the critical characteristic of supervised learning data is that it must be labeled for outcome-comparison. An example of this in practice could be labeled photos for use in training a convolutional network. The accuracy of the output of the network is determined by comparing it to the predefined label attached to the input. However, there are many issues with this approach. Some of these issues include the amount of training data, the dimensionality of the input space, and bias/variance tradeoffs (Mehryar et al. 1).

The issue of the amount of training data hinges on the type of training algorithm used. If the classification function is too simple, then a learning algorithm with a high level of bias and low variance in the data will only need a small data set. However, if the classification is too complex, then the amount of training data required by the same training algorithm will be far too large and take too much time to be considered effective (Mehryar et al. 1). Another issue with this design is the dimensionality of the input space. This is prevalent in models using many dimensions within the input data set for problems such as image classification. In this example, there is a matrix for every pixel value for all color values: red, blue, yellow. With many dimensions within the data set, the training algorithm that will perform best is one with a high variance and low bias. A third issue is the bias-variance tradeoff dilemma. In order to effectively train a neural network, the algorithm must be appropriately chosen based on the variance of the data set and the bias error of the algorithm (Gemen et al. 2). This is important because if the algorithm has a low bias, then it must be able to appropriately bend to the variance within the data set. This isn’t a problem for data sets that have a low variance. However, if the data does have a high variance, then the network will classify the data differently each time (Gemen et al. 2). Despite these concerns, supervised learning is highly effective when a sample data set with a labeled desired outcome that is also closely representative of the target environments data set is used. However, applications in which the goal is to find patterns that were unknown before are better suited to an unsupervised learning approach.

Unsupervised learning is the process of training a neural network on data that is not labeled. This process is used to identify patterns in data that were not made apparent before. This is used to find an underlying structure within the data set and can be used for accurate data set collection for supervised learning systems. However, because there is no way to validate the output of a neural network trained with unlabeled data, there is no feasible way to determine the network’s accuracy. There are four primary uses for unsupervised learning: clustering, anomaly detection, association mining, and latent variable models. Clustering is used to divide the data set into groups that share characteristics (McGregor 15). There are two types of clusters known as hard and soft clusters (McGregor 15). A hard cluster is a cluster whose contents belong to one of many mutually exclusive classifications (McGregor 15). Soft clusters contain data points that belong to multiple different groups (McGregor 15). Anomaly detection is an application of unsupervised models that determine outliers in the dataset. In this model, the data set is considered normal and then any values that lie outside of the normal pattern are labeled (Eskin et. al. 3). This has an application in error detection such as fraud, user error, and others (Eskin et. al. 3). Associating mining is used to identify items in a data set that occur together. This is similar to clustering, but instead of putting members into groups, it identifies groups that are natively in the data set. Latent variable models are commonly used to reduce the number of features in the data set. Unsupervised learning is similar to supervised learning models in that they recognize patterns, however, the application of unsupervised models is more advantageous in analytical studies of data sets. While not able to determine the accuracy of output, these models are capable of efficiently classifying groups within the data when applied properly. There are many forms of conditioning and many aspects of training to produce a network that provides accurate output.

A common and effective form of training in neural network research is known as reinforcement learning. Reinforcement learning is a process of machine learning based on the prospect of maximizing the reward for correct and/or optimal decisions of the neural network (Doya, Kenji 30). Reinforcement learning, like most influences on neural network design, is similar to the way that the human brain learns effectively. Reinforcement learning in the application of neural networks can be integrated to work with many popular learning algorithms. There are two primary forms of reinforcement learning that pertain to machine learning and neural networks in particular: positive and negative reinforcement.

Positive reinforcement learning engages in rewarding the network when the output from a given data set is similar or the same as the desired output (Doya, Kenji 30). The more similar that output during the training process is to the desired output for the given data value, the more frequently that occurrences similar to the previous occurrence will be applied. This means that the more correct the output is the more likely that future input of similar data will also be correct.

Negative reinforcement learning engages in rewarding the network when negative conditions are avoided/prevented. This is different from positive reinforcement learning because the network needs to be able to recognize that it avoided an error. This means that when incorrect output is avoided that future input with similar data will also be avoided (Doya, Kenji 30).

Both forms of reinforcement learning can be more or less effective than its counterpart based on the form of input, application, and other factors. However, each is capable of effectively training each network given that the data set presented as input is appropriate for the type of reinforcement learning applied. Also, the primary training algorithm used with this style of learning must be effective with the given data set to avoid hindering the effectivity of the learning process.

Neural networks have countless applications in various environments. They can be used to identify the contents of an image, process human language, and identify patterns in datasets that are too large for a human to effectively analyze. More recently in neural network research, however, neural network applications have become more advanced and broader in application. These new applications can be seen in examples such as military planning frameworks, artificial intelligence in virtual services, and in the research of discovering new strategies for games that are thousands of years old. This research shows the promise of neural networks in complex environments. This survey will discuss five research projects in which neural networks have been used to demonstrate the capability of this computational model in planning architecture and development.

Section 1: Work Summary

AlphaGo and AlphaGo Zero

Introduction

In March of 2016, Google's DeepMind research group demonstrated the power of deep neural networks and the capability that they have to solve complex strategic problems by defeating 18-time world champion Go player Lee Se-dol with their artificial intelligence system AlphaGo (Granter et al. 619). This is not the first time the world has seen an artificial intelligence system that has bested champion board game players, however. In 1997, IBM produced a chess-playing AI named Deep Blue that beat world chess champion Garry Kasparov (Granter et al. 619). However, AlphaGo is unique in comparison to Deep Blue in that the game that it has been trained to play is known as the most complex board game in the world. Go itself is over 2000 years old with a very simple set of rules. However, the environment of the game is capable of 10^360 different configurations (Granter et al. 619). This immense level of complexity is what sets AlphaGo apart from other artificial intelligence systems and is the reason AlphaGo exhibits a newly found problem-solving potential of deep neural networks.

Overview

The rules of Go are relatively simple yet create a complex and large number of combinations of moves. The board is composed of a 19 by 19 grid in which two competitors alternate placing black and white stones, respectively. Once a stone is placed on the board it is no longer allowed to move. The goal of the game is to use the stones to occupy as much of the grid as possible. A player may form a group, a collection of adjacent cells, around the opponent's stones to capture them. If this occurs, the whole group is removed from the board. While this rule set is simple, it is difficult to implement a computerized system to choose moves. This is because it is difficult to determine which player is winning. While one player may occupy more of the board, the other player may be at a positioning advantage that will prove to be successful later in the game. Because there is no adequate method of identifying who is winning in the game, traditional reward factor-based machine learning techniques prove inadequate. This is one of the challenges that proved most difficult to overcome during the conceptualization of AlphaGo.

Most artificial intelligence applications designed to tackle problems similar to that of AlphaGo, like Deep Blue, are designed using specific sets of protocols that define the training process. For example, Deep Blue approached the game of chess using a, as Dr. Scott Granter describes, "brute force algorithm" (Granter et al. 620). Due to the complexity of Go, the design of AlphaGo followed a different play-style. This was also because the research team at DeepMind wanted to create a system that would later be implemented in broader ranges of use outside of Go. Many researchers have suggested that the AlphaGo framework could continue to be applied in medical research and diagnosis (Granter et al. 620).

AlphaGo utilizes a deep neural network structure that is different than other similar systems. The structure of the AlphaGo system is referred to as a Parallel Intelligence System that uses separate software-defined systems (SDS) (Wang et al. 116). An SDS can refer to deep neural networks or other machine learning systems. The use of SDS in the case of AlphaGo is that simulated environments can be utilized to perform actions of the SDS against itself (Wang et al. 116). This was utilized in the design of AlphaGo for one primary philosophy. This philosophy focuses on the fact that using human level play to train a deep neural network is highly effective, however, the level of accuracy of that network can only escalate to the level of the human examples it is given. Using an SDS within a Parallel Intelligence System, the neural network is able to train on input from a copy of itself within a simulated environment (Wang et al. 117). In this method, the system is able to experience settings in which typical human-based environments do not typically form. This is critical within Go because of the extent of complexity within the game. AlphaGo performed strategies that had never been experienced before during its matches with Lee Se-dol (Silver et al). The strategic examples presented by AlphaGo have changed the way that the game is typically played – a traditional play style that has been present for thousands of years. Dr. Scott Granter describes this with, "It makes moves that no human, including the team that made it, understands," and continues with "AlphaGo is the creation of humans, but the way it plays is not" (Granter et al. 619). The strength of AlphaGo, in this aspect, lies within its parallel design.

A parallel design, in this sense, is effective because of the difference between the representation of the physical system and an artificial system (Wang et al. 118). In the setting of AlphaGO, this describes the setting of the game environment and the virtual system within the network. The parallel design of AlphaGo is able to be used with three modes of operation: learning and training, experiment and evaluation, and control and management (Wang et al. 118). During the learning and training of the system, the two environments, the physical system, and the artificial system are different, and the goal of this stage is to bring the virtual environment closer in relation to the physical environment (Wang et al. 118). Because this is in the early stages of the process, not much action is required to generate large differences between the two systems. The experiment and evaluation stage describe the process of generating, or simulating, environments and conducting computational experiments within the system. During this process, the artificial and physical environments communicate with each other as solutions are generated (Wang et al. 118). In the third stage, control and management, the artificial and physical systems continue to communicate in real-time while control and management are attempted over the entire system (Wang et al. 118). AlphaGo is not famous because of the implementations of new methods or techniques. Instead, it demonstrates the potential of current concepts within the artificial intelligence research community.

Playing Atari with Deep Reinforcement Learning

Introduction

The DeepMind research programs commonly aim to improve the ease-of-use of reinforcement learning (RL) in applications of neural network architectures. Demonstrating the possibility of using neural networks in processing digital visual data effectively while creating a multi-purpose strategy game player was the focus of DeepMind's Atari project. Primarily, this project was aimed to create a convolutional neural network that is successfully able to learn control agents through high-dimensional sensory inputs such as vision and speech. While the model to accomplish this goal is built on a convolutional model, its inspiration comes from breakthroughs in other architectural techniques such as multilayer perceptrons, restricted Boltzmann machines, and recurrent neural networks, each of which have demonstrated their capabilities in deducing control agents through both supervised and unsupervised learning (Mnith, Volodymyr, et al. 5).

Overview

The largest challenge of this project was in the unsupervised reinforcement learning approach. Reinforcement learning is a computational framework for an entity to learn by receiving rewards (Doya, Kenji 30). The reward reinforces the action in which it was caused. In the form of a neural network, rewards ultimately determine changes in weights. However, these rewards are only received on occasion and are unreliable in timing. This means that training could take exponentially long periods of time (Mnith, Volodymyr, et al. 5). Another challenge that this approach encountered was that in deducing control agents, different states could and often would be very closely related. This relation would need to be considered during processing in order to ensure accurate results (Mnith, Volodymyr, et al. 5). In order to overcome these challenges, the DeepMind research team decided to apply a convolutional neural network model to process video data throughout changing and complex reinforcement learning environments.

Convolutional neural networks are an alternative type of deep neural network that is especially and most commonly used in image and speech recognition (Sainath, Tara, et al 1). In convolutional neural networks, each hidden value is computed by multiplying small local inputs against the weight sets. Those weights are then shared across the entire input space (Sainath, Tara, et al. 1). After hidden units are computed, another layer, the max-pooling layer, helps remove possible discrepancies due to variations in the same input (Sainath, Tara, et al. 1). These include different speaking styles or variations in video quality. A convolutional neural network is made up of convolutional layers where each node only processes data for a receptive field, which refers to the space in which input will modify the contents of that node (Sainath, Tara, et al. 1).

This network was trained with a variant of the Q-learning algorithm. Q learning is, in Dr. Christopher Watkin's words, "a form of model-free reinforcement learning" (Watkins, Dayan 55). The agent of this project is used within an Atari emulator for experimentation with different simulated digital environments. The process in which the agent acts is broken into time-steps. However, it not feasible to process efficiently and accurately one frame per time step. To overcome this issue, consecutive observations are separated and processed at the same time (Watkins, Dayan 56). There is a standard assumption that rewards are discounted over time (Watkins, Dayan 56). The future discounted return time is calculated with respect to the time step in which the game ends.

The games that this network was trained to be implemented with offer several different sets of rules, controls, and reward systems. Because the network processes input from an array of pixels, training the network is the same independent of the game platform. This modularity allows for the potential of implementing this architecture in other environments as well. However, encoding rules for each game is different per application. Nonetheless, the potential that this technique demonstrates through its success in the Atari platform pushes the possibility of the future of using convolutional models in reinforcement learning projects.

NeuroChess

Introduction

The use of artificial intelligence to play chess has been a computing research topic for decades; however, typical techniques have proven ineffective and inefficient due to the variability in the game environment. These techniques mostly include generators that evaluate every move on the board possible for each decision. While this has proven effective in small instances, processing and maintenance require copious amounts of time and resources. In order to mitigate these fallbacks, fast evaluation functions are used. However, these functions are typically made by hand and are difficult to maintain. Overall, these classical techniques are highly inefficient and ineffective. NueroChess is a research program that challenges typical approaches by using an inductive deep neural network learning process, temporal differencing, and a variant of explanation-based learning (Thrun, 1069).

Overview

The beginning of this process starts at the end of the game of chess. The goal of NeuroChess is to learn from the outcome of chess games. The NeuroChess architecture is built around the concept of temporal difference learning or TD learning. In author Sebastian Thrun's words, "Temporal difference learning comprises a family of approaches to prediction in cases where the event to be predicted may be delayed by an unknown number of time steps" (Thrun, Sebastian 1070). This approach is ideal for implementation in chess because of the variability in the length of potential strategies. The use of TD in NeuroChess is to find a function that evaluates the current chessboard on the likelihood that it is winning. TD finds this function by denoting each game board, being each setting between moves, into training patterns. This evaluation function is denoted as V (Tesauro 58). V is evaluated by a series of three rules depending on whether the current setting is winning, losing, or a draw (Tesauro 58). The evaluation function is constructed recursively over the course of evaluations and is controlled by a decay constant γ so that the network favors quick success over lengthy or drawn-out conclusions (Thrun, Sebastian 1070).

Inductive learning techniques such as back-propagation suffer from immense training times in settings like chess (Thrun, Sebastian 1071). This is because some events have several different components that determine whether or not a specific action is good or bad, and aspects like the location of other components need to be considered as well. Explanation-based learning, however, generalizes more accurately from less training information and relies on the availability of domain knowledge instead (Thrun, Sebastian 1071). This way the program can consider the rules and format of the game to infer whether decisions are good or bad. This is an immense advantage in the case NeuroChess where the environment is completely reliant on these factors.

Chess, like many board games, is played by two players who engage in each other's game pieces in order to eventually defeat their opponent by trapping their king in a state known as checkmate. However, this game model features many different varying components. These include different game pieces and special cases where moves may be made depending on the setting. Each different type of piece has different sets of rules on how that piece may move. Additionally, each piece has a large tree of potential moves and paths. In order to develop an artificial intelligence system that can perform at an expert level, the system must be able to consider every possible move that is available. As stated previously, however, the immense amount of combinations makes evaluating each move non-feasible. The NeuroChess project implemented several different techniques to overcome these issues including parallel architecture, a knowledge base, and several optimization operations. The success of this project demonstrates the ability for current machine learning techniques to handle complex learning tasks that have otherwise been seen as overly difficult.

OpenAI Five

Introduction

OpenAI Five is a team of five independent artificial neural networks that have been designed to handle continuously changing environments by using the environment within the strategy game Dota 2. OpenAI Five is the second generation in research by OpenAI after its resounding success in a one-versus-one exhibition setting in the 2017 Dota 2 championship. The primary goal of OpenAI Five is to further the research into the OpenAI platform OpenAI Gym. OpenAI Five uses a new training system developed by the OpenAI team. This learning algorithm is known as Proximal Policy Optimization and aims to make training artificial neural networks simpler, more effective, safer, and more efficient than traditional training processes.

Overview

The decision to use the Dota 2 game environment in this research project was to exhibit the potential of deep neural network's capabilities to perform in continuously changing environments. Dota 2 is an online game where thousands of variables are constantly changing the game environment. OpenAI aimed to use this in reference to the future of the application of the system in real-world environments. OpenAI Five is a team of five independent neural networks that do not communicate with each other directly (Pachoki 1). Instead, the research team decided that it would be more effective to only have each agent-driven purely by incentive. While this seems counterintuitive at first, this system provides for much longer applicability and has proven to be effective in team-based environments, nonetheless. Because there are thousands of changing variables in the environment, the OpenAI team had to find a way to systemize and quantify each aspect of the game. This was implemented successfully as a list of 20,000 values that represent the aspects of the game that humans are able to observe; this is to ensure that the networks are not receiving any hidden information to give them an advantage (Pachoki 1). Using the information stored in this list, the system can be trained using the general-purpose training system Rapid.

Rapid is the reinforcement learning system that OpenAI has developed to run on any Gym environment. Gym is a framework for reinforcement learning for training a neural network (Pachoki 1). An agent of the Gym framework learns on a system in which an agent's experience is divided into episodes (Brockman, Greg et al. 1). The initial state of an agent was randomly sampled from a distribution and the interactions in the environment continue until the agent reaches a terminal state (Brockman, Greg et al. 1). Each episode is divided into timesteps in order to keep simulations linear. Interactions are generated when the agent makes a decision. With each decision, the agent receives an observation and a reward (Brockman, Greg et al. 1). The goal of reinforcement learning is to maximize the reward an agent receives for each decision.

The effectivity of the agent is determined by the cumulative reward by each agent (Brockman, Greg et al. 1). Thus, Gym's approach to reinforcement is heavily reliant on proprietary mathematical theory. This process generates a gradient on the batch which is averaged with the gradients generated from other batches (Pachoki 1). After each iteration, the agent syncs with the other workers. What sets Rapid apart, however, is that the Gym framework focuses on the abstraction of the environment as opposed to the agent (Brockman, Greg et al. 2).

OpenAI Five uses the advantages of Gym and Rapid to effectively and efficiently generate agents that are able to challenge professional players in Dota 2 to exhibit the advantages and the future of using reinforcement learning in complex environments. As opposed to other neural network projects such as AlphaGO, the work by OpenAI focuses on the environment rather than the agent. Because games like Chess and Go have very simple environments with very complex forms of strategies, their frameworks are limited in environments that are rapidly changing. However, the agents in games like Go have to consider multiple game pieces at once while OpenAI five has separate agents for each artificial player. The amount variability in the Dota environment comes from the continuous list of factors including, but not limited to, other players, non-player characters, character actions, enemy actions, item building, and obstacles.

OpenAI uses Dota 2 as the environment for its research projects because of the game's complexity and popularity. Dota 2 has been an extremely popular multiplayer game for several years and hosts annual competitions that draw millions of viewers. To demonstrate the power of the Dota 2 neural network project, OpenAI developed a timeline that displays the progress that the project has made. The project began on

November 5, 2016, and in January agents successfully defeated a scripted game against bots. After this first proof of concept was a success, the project continued to grow all the way until on April 13, 2019, OpenAI Five becomes the first AI to beat a team of world champions in an esports game at national finals. As OpenAI Five continues to grow, it continues to set new records on the potential that neural networks have in handling exceedingly complex learning and planning tasks.

Playing Checkers Without Human Expertise

Introduction

Checkers is a classic board game played on an eight-by-eight board. Each player controls 12 game pieces and makes one move per turn by choosing to move any of their remaining pieces in diagonals by one space. However, if a player's next move is obstructed by one of the other player's pieces, that player may "jump" over the opposing piece and remove it from the game. If more than one possible opportunity to jump is available, the player must decide which one to move. The goal of this project was to exhibit the potential for AI to learn without the input of human expertise. This model, shared by other projects like OpenAI Five, aims to challenge traditional training practices that supply human input as a learning basis.

Overview

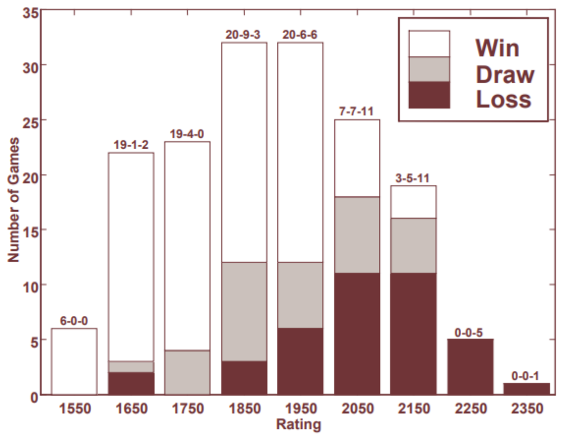
In the design of this project, there were many restrictions that had to be followed. This is because the platform that the final agents would be performing on was an online checker tournament application. Therefore, limitations, like time available for decision making, were set to lie within the rules of the online platform. Also, within this platform draws were permitted. This was decided if both players mutually agreed to end the game. With restrictions set, the design of the agents was considered.

Each game field was represented by a vector with a length of 32 (Chellapilla et al. 4). This represented the total possible moves for the game piece’s potential moves. Because there is no limit to the number of moves that may be made during the game and there was no time limit to the game, the value of each move made by the agent was determined by the quality of the resulting positions (Chellapilla et al. 4). This served as an effective method to ensure that throughout the game the intention of each play was to be able to make better moves in the future. Each position was evaluated on quality based on several different factors including safety, jumping potential, and the number of moves required to reach the back of the board. The structure of the network followed a standard layout with only one output node that represented the move decision.

Compared to other board games like chess and go, checkers is much less complex in terms of variability between boards. This means that many typical brute force approaches might be an effective method to tackle this learning task. However, one goal of this project was to determine if using the extra computational resources to use a fully evolved network over networks that rely solely on the piece differential was valuable. In order to test this, two control experiments were performed. The first experiment used a fully evolved network playing a total of fourteen games, two games were played to completion with the network winning both. In the other twelve games, the network held a significant advantage in ten games while two of the games were held at a disadvantage. The second experiment consisted of the same limitations as the first experiment, but a limit of two minutes was implemented for search time. In this case, the network won two of the fourteen games, the same as the first experiment, but only held an advantage for six of the remaining twelve games. These two experiments display the value of using the extra computational resources to implement a fully evolved network in the setting of checkers (Chellapilla et al. 6).

The human opponents were not made aware that they were playing against a computer to keep play bias at a minimum. A total of 165 games were played against human opponents. Each move was modified to take anywhere between 0 and 60 seconds to be executed. Opponents were chosen based on availability with diverse ranges of skill levels. Each game was rated based on a score approaching 3000. The higher the score typically meant the game lasted longer and was more complex than average games. In a distribution of scores, it is apparent that as the score approaches 3000 the win/loss ratio becomes smaller. This is opposed to the large win/loss ratio from games with lower scores. This is demonstrated in figure 1.

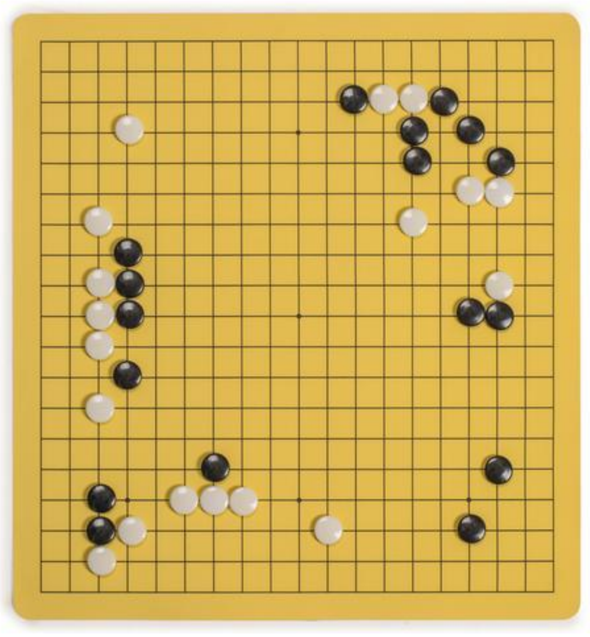
Fig. 1



Chellapilla, Kumar, and David B. Fogel. "Evolving an expert checkers playing program without using human expertise." *IEEE Transactions on Evolutionary Computation* 5.4 (2001): 422-428.

Section 2: Analysis

AlphaGo/AlphaGo Zero



Standard Go! Board by Yellow Mountain Imports

Efficiency

The reinforcement learning algorithm named the “lookahead search” is used inside the training loop to increase precision, speed, and stability during training. The lookahead search is a combinatorial search component that identifies the limit in which the search explores (Wang, et al. 115). This is critical in AlphaGo because the search tree that is processed each time is extremely large. The use of a traditional search algorithm without the lookahead component could hinder the utility of the network because most traditional searches would consume the majority of available memory. The neural network is represented by a function and is optimized on Google Cloud using TensorFlow. The program is aided with the help of 64 dedicated graphics processing units as well as a collection of 19 CPU parameter servers. The parameters within the neural network are optimized by stochastic gradient descent and the learning rate is annealed (Wang, et al. 116).

Stochastic gradient descent is a powerful, yet simple training algorithm based on the gradient descent learning algorithm. The gradient descent algorithm focuses on minimizing empirical risk. Empirical risk measures the training set performance while the expected risk measures the expected performance on future samples (Bottou 2). Each iteration of training using gradient descent updates weight values using the gradient of the empirical risk (Bottou 2). Stochastic gradient descent drastically simplifies the gradient descent model by removing the need to compute the empirical risk exactly (Wang, et al. 118). Instead, each iteration makes a general estimation using a single randomly picked sample. However, it is important to note that because each updated weight value is calculated using an estimate that, in many instances, significant noise can change the behavior resulting in values that aren't as accurate as they would have been using the classic gradient descent model (Bottou 2). This model is extremely powerful because the stochastic algorithm can be deployed whenever needed in whatever application. This is because it does not need to remember which examples it has seen before and thus optimizes the expected risk (Bottou 2). This is highly effective in the environment of Go! because of the extremely large number of possibilities in the game environment.

Input

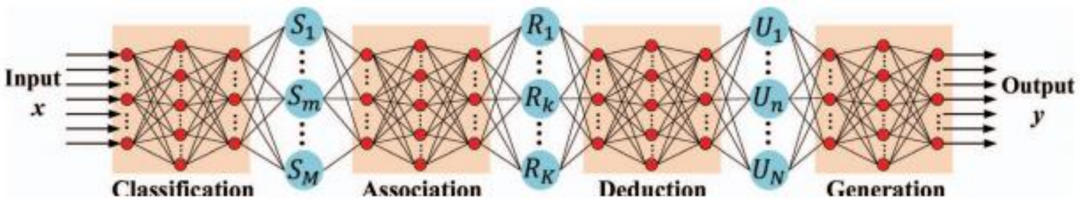
AlphaGo and AlphaGo Zero are both different in comparison to other neural network projects with similar goals in the simplicity of the input types. Because the setting of Go!

is simply black and white game pieces on a checkerboard, the development team behind the AlphaGo projects decided the most efficient and effective method of organizing input would be by using a simple vector with values representing the black and white game pieces. This also contains the history data of the game environment for use in calculating values such as the probability of moves.

Structure

The AlphaGo projects were designed based on deep rule-based networks that were implemented for decision and evaluation using several layers that control different stages of the output generation process. The four layers of the AlphaGo DNN networks are classification, association, deduction, and generation with given input vector and subsequent output vector. These layers are vastly different from each other and can be utilized independently with separate networks. This structure allows for greater flexibility and allows for individual implementation based on the user's preferences. This means that all four layers are not restricted to one type of network such as convolutional or recursive (Wang, et al. 116). This can be visualized by figure 2.

Fig 2.



Source: Wang, Fei-Yue, et al. "Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond." *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016): 113-120.

Training

The single neural network used in the AlphaGo projects is trained with a unique training algorithm. For each active position, a Monte Carlo Tree Search (MCTS) is executed. The Monte Carlo Tree Search was implemented in the AlphaGo projects because of its capability to evaluate the quality of different members of the tree which, in the case of the AlphaGo projects, are the possible moves. In this search tree, entire games are completed starting with random choices for moves. Once a game is completed, the nodes that represent the tree's members are weighted, and, typically, the nodes with larger weights represent more promising options. The MCTS is separated into four successive processes: selection, expansion, simulation, and finally backpropagation (Enzenberger et al. 259). During the selection phase, the tree is traversed starting with the root of the tree until a leaf node is reached. However, a leaf node can only be reached once due to the actions in the expansion phase (Enzenberger et al. 259). During the expansion phase, children are generated for the leaf node chosen during the selection phase. One of the children created during this stage, labeled as X, is selected and used during the simulation phase. During the simulation phase, an entire playout starting at node X until the game is completed. Finally, the backpropagation phase is completed in which the weights of the nodes traversed from the root to node X are updated based on the performance of the simulation phase. The algorithm used in this phase is common amongst many neural network training algorithms seen in other projects.

Over the three days of training of the AlphaGo Zero program 4.9 million games were completed with 1600 simulation cycles for each MCTS that was performed. Approximately, it required roughly 0.4 seconds of processing for each move.

Playing Atari with Deep Reinforcement Learning

Efficiency

DeepMind's research team was presented with several challenges while designing the system to control and maintain an efficient agent for playing digital Atari games. Arguably the most troublesome component in their project regarding the efficiency of the neural network was processing the visual data from the game environment and formatting it to be recognized by the neural network. The raw components of visual input consist of a 210 x 160-pixel array containing color data from a pallet of 128 values. Such a large set of values would take far too much time to be effective. In order to create a data set that is much more consumable by the processing of the neural network, a series of preprocessing techniques was implemented to optimize the input vector (Mnith, Volodymyr, et al. 5). First in this process was a system of converting each frame into a grayscale. Second, the images were sampled down to a 110x84 pixel array. Finally, the new array is cropped into an 84x84 pixel rough representation of the play screen (Mnith, Volodymyr, et al. 5). This process shrinks the amount of data being collected each frame significantly and is enough to produce an input vector that is much more realistic to be processed by the main processing function. This series of optimizations is unique to digital format applications. The final cropping to an 84 x 84 image is required by the GPU implementations of convolutions, which expects square input dimensions (Mnith, Volodymyr, et al. 5). Also, another key optimization made by DeepMind is that only the final four frames of each designated time step are recorded (Mnith, Volodymyr, et al. 5). This is important because not all frames are crucial for input.

Input

The input for this neural network consists of seven different games made for the original Atari console. Each game is processed by the preprocessing function described above. The games used in this research project include Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders. Each game environment is vastly different from the others in order to demonstrate the flexibility of this particular neural network framework without needing game-specific information. However, each different application rewards success differently. This was taken into account to ensure that training would be consistent independent of the game with which it was training. The primary reward system used by the network used the reward systems for each game. For example, the most well-known game included in the game set, Pong's reward system uses the number of times that the player scored against the opponent and whether the game resulted in a victory for the player. However, in games like space-invaders, this system does not translate over directly. For example, the definition of scoring is destroying one of the many enemy ships. Because there are thousands of potential points in space invaders, the reward system must be adapted proportionally to match the availability of potential points for the other games.

Structure

The Atari neural network model faced an issue with the way the activation function parameterized the input vector. The design of this neural network model was considered by two different approaches in response to this issue. The first consideration used both the input vector as well as the history of the frame data as input. However, this architecture requires a full second pass to compute the expected output for each input pass. Considering the necessity of efficiency of this environment model, this approach was not appropriate for Atari game applications. The second consideration uses only the input vector as input and a separate output unit for each action or pass. Each output corresponds directly to the expected value of each input pass. This structure allows for only a single pass through the activation function for each input set providing for more efficient and quick processing time – an important aspect for many of the fast-passed target game environments (Mnith, Volodymyr, et al. 6).

The neural network design for processing most pixel data representations primarily uses a convolutional model. This network is no different and uses several layers with different convolutional operations. The layer structure of the Atari neural network consists of one input layer, three individual hidden layers, and a single output layer (Mnith, Volodymyr, et al. 6). The input layer takes the 84 x 84-pixel input vector and passes it through to the first hidden layer. The first hidden layer convolves 16 8 x 8 filters (Mnith, Volodymyr, et al. 6) This convolution includes a stride of four. The second hidden layer differs with convolutions of 32 4 x 4 filters and with a stride of only two. Both hidden layers one and two use a rectifier nonlinearity, a mathematical operation that reduces the prominence of noise, after each convolution cycle (Mnith, Volodymyr, et al. 6). Finally, the third hidden layer is fully connected and consists of 256 rectifier units (Mnith, Volodymyr, et al. 6). Fully connected to the third hidden layer, the output layer that produces a single action. The structure of this neural network model focuses primarily on efficiently processing the input vector and condenses into a single output that represents one action.

Training

The Atari neural network system focuses on deep reinforcement learning techniques within the training process using a stochastic descent gradient. In most supervised learning studies, tracking the success of the network is fairly straight-forward and easily analyzed. However, because the reward system for the game environments targeted by the Atari model needs to be adjusted before the end of processing, issues arise in deducing the success of a training set because of inconsistencies in the reward data. These inconsistencies can be confusing and depict less correlation between run time and success rates. This can cause longer training times and can create less consistent results in the final application. However, because this model is trained using deep reinforcement learning, a more accurate statistical representation of training success is in the estimated value plotted against training run time. This representation demonstrates less noise and more appropriate distributions.

NeuroChess

Efficiency

Similar to other neural network applications that feature game board environments, NeuroChess's performance experiences issues with the more commonly used training algorithms such as back-propagation. In order to maintain effective training times and still produce accurate results, NeuroChess uses an explanation-based learning technique. Explanation based learning is different from most learning algorithms because it uses a set of knowledge of the domain of the environment in evaluations (Thrun, Sebastian 1070). This technique is not typically used for neural network applications, however, due to the complexity of the Chess environment, it serves well to reduce the time in the evaluation steps. The format of this knowledge is a set of rules for game pieces. Because the network does not have to evaluate every set of moves during each time step, the network can train more accurately with much less time. However, the same database can be used to train other systems increasing the modularity of the project.

The work of the primary neural network is divided between a series of workstations (Thrun, Sebastian 1073). There is one workstation that acts as a primary server that contains the most recent weight values and diagnostic information (Thrun, Sebastian 1073). Each subsequent workstation remotely accesses the primary server to read weight values in between iterations as well as send its current weight values. This system increases training time to a much more manageable level.

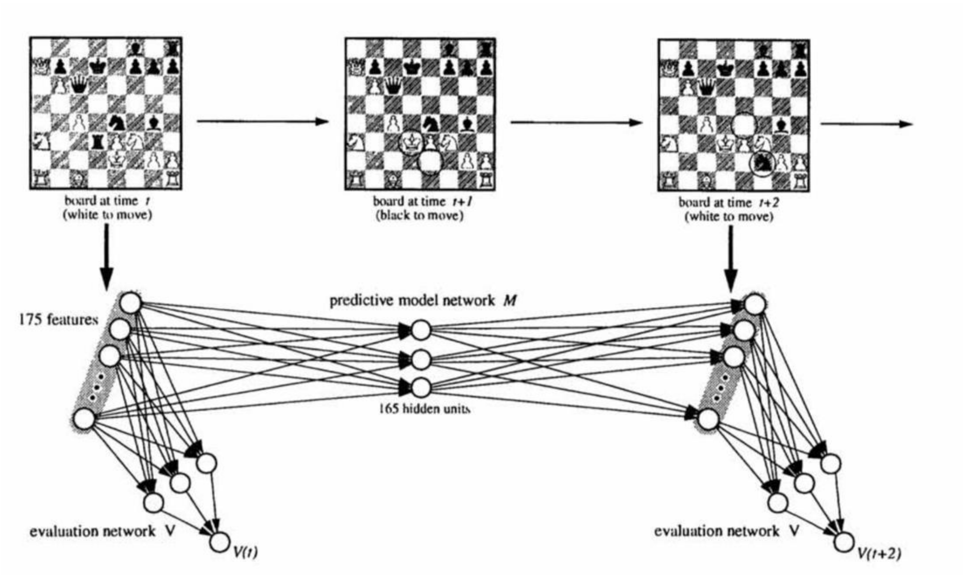
Input

The chess pieces are represented by individual identification numbers and are stored into an input vector. Empty spaces are represented by their identification numbers. The input vector is sent to the first hidden layer. Unlike most neural networks with only a single network structure, the input is also sent to the secondary network. The secondary network handles the input by using a series of processing functions to determine likely input values in the next two moves.

Structure

NeuroChess uses a set of two parallel neural networks to host the processing of the game environment (Thrun, Sebastian 1072). The first network takes the input vector and passes it into a series of hidden layers. The evaluation function considers the expected environment produced by the secondary network along with the input vector. The output of this layer is the final move decision for the current time step. The secondary network is constructed with 165 hidden layers that produce a set of expected values that represent the expected environment two moves ahead of the current environment (Thrun, Sebastian 1072). This network is trained prior to the training of the primary network using known chess games and is used to create a new input vector. The new input vector is passed into a series of processing layers that are very similar to the primary evaluation system. Finally, the secondary network maps the expected game piece locations using a large database of grand-master chess games (Thrun, Sebastian 1072). This way, the secondary network is able to determine differences between expert-level chess play and implement the same strategies in the training of the primary network. Together, both neural network models work together to determine the best move regarding several different factors including but not limited to the likelihood of reaching checkmate and advantage between each piece. This structure is visualized in figure 3.

Fig. 3 : Representation of parallel network structure:



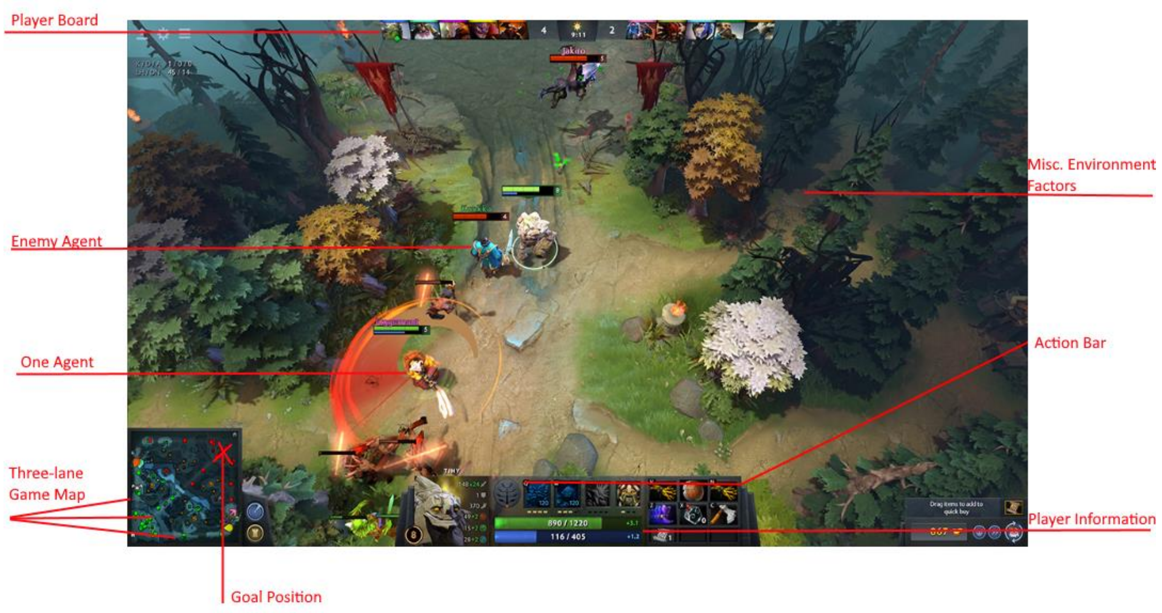
Source: Thrun, Sebastian. "Learning to play the game of chess." *Advances in neural information processing systems*. 1995.

Training

The NeuroChess project employs the application of a process known as temporal-difference learning – a collection of processes deployed in applications using delayed prediction (Thrun, Sebastian 1070). Because NeuroChess uses the parallel network to find an expected input vector for two game-moves ahead, this approach is used effectively in the evaluation functions to rank the chessboard input vector on the likelihood to win. The application of this technique is used to convert complete chessboard environments into one of three values: 1, 0, -1 where 1 represents a board that is likely to result in a win, a 0 represents a board that is likely to result in a draw, and -1 represents a board that is likely to result in a loss (Thrun, Sebastian 1070). 0 rankings are more common to appear in later game scenarios rather than early game settings. The algorithm handling this ranking procedure is implemented using recursion to pass through each intermediate board between stages (Thrun, Sebastian 1070). The evaluation function is also used at the end of the game to generate an overall rating of the game's outcome. However, the application of temporal-difference learning makes this procedure different than most standard outcome rating algorithms because the evaluation function is trained using the intermediate results generated by the parallel network for moves two steps in advance.

There were three major concerns that the NeuroChess team encountered throughout the development of this project concerning the training and application of the network. The first was the issue of sampling. Because most individual snapshots of board settings are insignificant among thousands of moves, a special system to assist the network in determining which move has the highest likelihood of leading to a victory is needed (Thrun, Sebastian 1073). To remedy this, earlier stages of training rely more heavily on the grand-master game database until the system is more capable of making more moves completely on its own. The second concern is quiescence, a criterion used to evaluate the depth of a search. This applies to the evaluation of a board setting and determining the goodness of that board. The primary issue here is that not all boards are equally complex and, while training, the evaluation of a board of higher complexity will not produce an adequate assessment (Thrun, Sebastian 1073). To fix this, the evaluation is based on a simple measure of threat from opponent pieces. The depth of the search for potential moves is based on this quiescence. The third major issue is discounting. Discounting is a process of adjusting the sums of payoffs so that, in cases where the pay-off values are continuously compounding, the future rewards will not become absurdly large in comparison to early rewards (Thrun, Sebastian 1073). Initially, the research team did not think that a discount would be needed for chess, however, they discovered that due to a high level of random anomalies in the evaluation functions, that the network completely failed without a discount. With appropriate adjustments to these issues, the neural network managed to perform exceedingly well against high-level opponents.

OpenAI Five



Efficiency

The complexity of the OpenAI Five project is regarded as one of the most intensive neural network research studies to date. Because the setting of the OpenAI Five agents is within the Dota 2 environment, there are thousands of factors influencing the environment every game tick. Typical approaches towards categorizing environmental factors are virtually useless in the scenario of Dota 2. Dota 2 is a multiplayer online game that focuses on controlling a predefined map. The game sets two teams of five players each against each other to continuously fight to gain control of the enemy team's home base. Each player selects a character to control before the game begins, and each character has its own set of actions that is different than the other characters. An action is an ability that the player can use to attack enemy players, defend themselves or teammates, or traverse the map. Because of the extremely large sets of characters, actions, and items that change the effect of the character's actions, Dota 2 has immensely complex environments that are vastly different between games. This also means that search trees for each factor or option for each time step can be infinitely large. Obviously, this has an immense impact on the performance of any application to handle the environment because, from the perspective of the network, restrictions are virtually non-existent. Putting this observation numerically, the environment of chess hosts an average of 35 potential moves per turn while Go hosts an average of 350 potential moves per turn (Pachoki 2). Dota 2, however, hosts an average of 1000 potential moves/actions per game tick. Furthermore, the average length of a game of Dota 2 lasts much longer than the average game of Chess or Go. Chess usually ends in less than 40 moves while Go usually lasts around 150 moves (Pachoki 2). OpenAI Five pulls game data every four game-ticks resulting in a game averaging 20,000 moves for each agent with an average of 80,000 ticks per game. Traditional approaches for training neural networks to efficiently observe and process environmental data is not suited for an environment for which the complexity that Dota 2 is renowned. In an attempt to approach the environmental optimization issue, OpenAI Five uses a massively scaled implementation of Proximal Policy Optimization.

Proximal Policy Optimization (PPO) is an approach to deep reinforcement learning approximation functions using a focus on first-order optimizations. PPO has two primary applications of policy optimization: policy gradient methods and trust-region methods (Shulman 1). Policy gradient methods implement a combination of estimating a policy gradient and then using it in a stochastic gradient ascent algorithm (Shulman 1). Stochastic gradient ascent is a close alternative to gradient ascent, which is a weight updating algorithm that uses the entire available data set for each update (Harrington 1). However, with large data sets such as the environment of Dota 2, this is not an effective approach. Instead, stochastic gradient ascent takes classifier data in batches as they are available instead of taking all of it at once (Harrington 1). This makes handling the immense amount of data easier by processing it in batches and updating the weights when needed. This allows the process of updating classifiers to occur throughout the game while minimizing the processing time of the evaluation functions.

One disadvantage that this concept has is that it is nearly impossible to draw information on what gameplay is considered better by pulling data from a set of previous known high-level games. Instead, OpenAI Five relies solely on training data from games it plays against itself. While this results in extremely long training times, it has proven to be the most effective method of implementing reinforcement learning approaches without using datasets from previous games. The implementation of such a large concept requires massive amounts of computational resources. OpenAI Five utilizes 128,000 preemptible CPU cores and 256 P100 Graphics processing units (Pachoki 2). The networks take one observation every four game-ticks with a total of 1,048,576 observations per day (Pachoki 2). Additionally, the agents play an accelerated gameplay system that performs the equivalent of 180 years of gameplay every day for each agent. While the number of resources required to effectively implement OpenAI Five, it is optimized to be as high performing and as efficient as possible within the limits of the Dota 2 environment.

Input

The world of Dota 2 is full of variations and different concentrations of potential actions throughout the symmetrical and lane-based map. OpenAI Five represents this environment by using a list of 20,000 numerical values for each factor (Pachoki 2). Additionally, because the game allows players, or in this case agents, to utilize a list of 8 different actions based on the character, OpenAI Five represents each action by its own enumeration values. Among the environment, values are sampled ranging from fellow teammates, nearby enemies, enemies' actions, non-player characters, structure and objects in the game, and affect areas of enemy attacks.

Structure

The structure of OpenAI Five consists of five separate networks that each contain independent, single layer Long Short-Term Memory networks (LSTM) with each containing 1024 units (Pachoki 2). An LSTM is a specially designed recurrent neural network that is capable of learning long-term dependencies (Olah). Typical recurrent neural network design involves keeping track of all past behaviors. This approach was implemented because it remedies this issue while maintaining the effectivity of a recurrent neural network structure. The core component of an LSTM is known as the cell state. The cell state is a single pipeline that runs through each cell in the network with very little linear interactions (Olah). This allows information to pass through the pipeline virtually unchanged (Olah). The LSTM is able to remove or add information into the cell state, however, this process is heavily regulated by a separate structure known as gates (Olah). Each gate is constructed with a sigmoid neural network layer and hosts a predefined operation to determine what information makes it into the cell state. Each network extracts data from the current game state, using a bot API developed by Valve corporation, which feeds into several input locations referred to as action heads (Olah). Each head is computed individually and holds specific semantic meaning representing possible actions – such as choosing an ability to use or a coordinate to move to in the game environment. This structure allows for the ability to recognize missing information or inconsistencies in the game environment very similar to that of an unsupervised learning neural network structure. An example of this in practice was observed when OpenAI Five was able to recognize when an area-of-effect attack was used by an enemy player. This is when damage is applied to anything within a specific area. The significance of this is that this attack style was not predefined in any way and the agent was able to avoid or escape the attack with no outside intervention (Pachoki 4).

Training

Because OpenAI Five learns solely from self-play, the training process begins with a sample of random values for weights. However, this poses a problem in which the strategic planning of the network can fall into a system in which it gains an increased bias towards strategies that it has performed in the past. To overcome this learning issue, OpenAI Five uses eighty percent of its gameplay against itself and twenty percent against playthroughs of its past selves (Pachoki 4). This overall process results in initially hopeless agents wandering aimlessly around the map. However, over thousands of hours of training, the agents begin to exhibit strategy planning. After several thousand hours of gameplay, the five agents began to exhibit advanced level strategy making including teammate swapping and resource farming to gain an advantage. However, once this network was used against human opponents, it failed miserably due to the inconsistency in loadout, health, speed, start level and several other variables (Pachoki 4). The network was retrained by playing against opponents with randomly selected loadouts to fix the issue in human play. This forced the network to explore various new strategies due to the network not having an implicit expectation of these variables. It is also important to note that these changes were applied only during the training and were not applied against final play. After these adjustments were made and several more training hours had completed, OpenAI Five began consistently beating human opponents. These training adjustments allowed for great variability and flexibility in the final application against varying skill leveled human opponents.

Because OpenAI Five utilizes deep reinforcement learning, one of the most important factors in training and effective network is measuring reward. OpenAI Five uses a collection of metrics that are defined by what human players use to evaluate how well they are performing in the game such as net worth, kills, deaths, assists, last hits and various other values (Pachoki 4). However, each agent has a hardcoded item and skill builds (Pachoki 4). This was implemented to simplify the training procedure and save time. The system that controls the reinforcement learning process is called Rapid. This divides the training system into a collection of workers that each runs a copy of the game with its own agent that collects experience and optimizer nodes. Optimizer nodes perform independent synchronous gradient descent utilizing the system of GPUs for faster processing times (Pachoki 5).

Playing Checkers without Human Expertise

Efficiency

The input format for the primary evaluation function is very lightweight compared to many other neural network input vectors and is represented using a single input vector of the game board. The simplicity of the input allows for fast processing time while using very few computing resources. In addition to the simplicity of the input vector's format, the rules for Checkers is very simple and very limiting for individual pieces. Because the vast majority of pieces are regular pieces that can only move in forward diagonal movements, the number of potential moves for each piece could be purely represented numerically (Chellapilla, and Fogel 4). Also, because each piece is equivalent, with an exception for King pieces, the same evaluation could be used for each piece. Each of these components together enabled the network to process each board incredibly fast.

Input

Using a representation of the eight-by-eight board, preparing the game setting to be of the appropriate form for processing by the evaluation function consisted of several different representations for each game piece. These representations include differentiating between which side a game piece belongs and King pieces. Each square in the checker grid is represented using a unique data structure with an index representing the individual square and a variable that represents the piece that is currently using that square during that time step (Chellapilla, and Fogel 4). Each of the representative variables was only one of five values: -K, -1, 0, +1, +K (Chellapilla, and Fogel 4). A 0 represented an empty state, a 1 represented a normal checker piece, and a K represented a king piece. Each value's sign represented which side, player or opponent, whom the piece belongs (Chellapilla, and Fogel 4).

Structure

The neural network is broken into five total layers. The first is the input layer that takes the input vector of length 32 and separates into a series of sub-vectors that sequentially represent overlapping sub-squares from the game board (see fig 4.) Each new sub-vector is passed into the first hidden layer referred to as the Spatial Preprocessing Layer (see fig 4.). This layer consists of 91 nodes. After evaluation by the first hidden layer, the modified values are passed into a second hidden layer with only 40 nodes. The new values are then passed into the third and final hidden layers consisting of only 10 nodes. Finally, the sum of the output from the third layer is passed into a single output node that represents the final move decision (see fig 4).

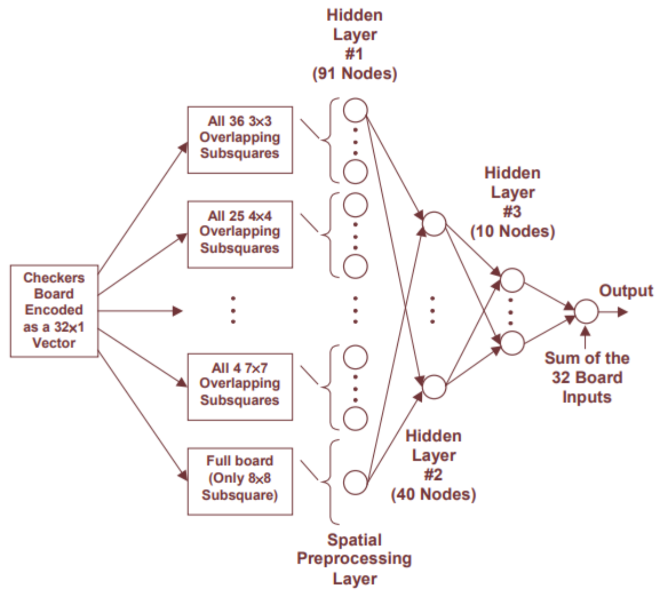

Hidden Layer #1 (91 Nodes)
All 36 3x3 Overlapping Subsquares
All 25 4x4 Overlapping Subsquares
Checkers Board Encoded as a 32x1 Vector
All 4 7x7 Overlapping Subsquares
Full board (Only 8x8 Subsquare)
Spatial Preprocessing Layer
Hidden Layer #2 (40 Nodes)
Hidden Layer #3 (10 Nodes)
Output
Sum of the 32 Board Inputs

Fig 4.

Source: Chellapilla, Kumar, and David B. Fogel. "Evolving an expert checkers playing program without using human expertise." *IEEE Transactions on Evolutionary Computation* 5.4 (2001): 422-428.

Training

The primary processes for evaluating weight modification were housed in an evolutionary algorithm that produced a scalar output of the board representing the worth from the perspective of the player (Chellapilla, and Fogel 4). This output ranged from 0 to 1 with 1 representing a perfect board with the most possibility for victory. This algorithm begins by creating a population of 15 strategies that are defined by weights and biases. These weights and biases are generated using a uniform sample over a range of 0.4 (Chellapilla, and Fogel 4). The size of this range determines the variability between strategies with the purpose of creating a useable variation for the initial stages of the training process (Chellapilla, and Fogel 5). The network also includes a value known as a self-adaptive parameter (Chellapilla, and Fogel 5). This parameter was equivalent to the variability range and was used to control the step size for each search process during evaluation. Each iteration created offspring with new weights and biases and received a point value representing the success of their performance. Each child performed in five randomly selected opponents (Chellapilla, and Fogel 5). In order to prevent children from playing games that lasted too long or reaching a never-ending stalemate, games were ended as a draw if they exceeded a total of 100 moves. If a game were lost, it would receive -2 points and if the network won, it would receive 1 point. If a game exceeded 100 moves and resulted in a draw the point value would be unchanged. In total, each generation played a total of 150 games with each generated strategy being implemented in 10 games (Chellapilla, and Fogel 5). Each game utilized a search algorithm known as a minimax alpha-beta search with a search depth of four. The minimax alpha-beta search algorithm utilizes a process of alpha-beta pruning that focuses on decreasing the number of subsequent node evaluations (Knuth, and Moore 1). The decision to use this algorithm in the setting of competitive checkers was made because of the efficiency of determining which subsequent moves to evaluate. If the algorithm determines that one move possibility proves that the current move is worse than the previous, it stops evaluating that sub-tree altogether (Knuth, and Moore 1). The process of evolving an expert level checker-playing neural network was completed over the course of six months through 840 total generations.

Conclusion

AlphaGo/AlphaGo Zero, Playing Atari with Deep Reinforcement Learning, NeuroChess, OpenAI Five, and Playing Checkers without Human expertise are neural network projects that demonstrate the capability of neural networks in advanced plan development tasks. However, the approaches of each project vary between efficiency, input type, structure, and training techniques.

Each project utilized different forms of optimization to increase the efficiency of the network's performance. This is due to the variation between each project's target environment and the availability of physical resources. AlphaGo/AlphaGo Zero, OpenAI Five, and Playing Atari using Deep Reinforcement Learning had availability to larger amounts of processing power as compared to NeuroChess and Playing Checkers without Human Expertise. However, all networks used a system for optimizing search times and used a modification system suited appropriately to each network's individual needs. For example, OpenAI Five used proximal-policy optimization and only used game data pulled from previous iterations while NeuroChess was able to effectively pull game data from a database of grand-master level games. While NeuroChess did not have as great of a need for input optimization as OpenAI Five, both networks performed well using their respective approaches.

All five projects utilize a form of vector to hold the input values used by the network. AlphaGo/AlphaGo Zero utilizes a vector representing the pieces on the game board. This approach is shared by NeuroChess and Playing Checkers without Human Expertise because each environment is a game board grid. This is opposed to the input format of OpenAI Five and Playing Atari using Deep Reinforcement Learning because these projects focused on an environment that was much more dynamic. Although these two projects utilized vectors of input values, the input was drawn from a continuously changing system. For the Atari network, this was the pixel values of the screen and OpenAI Five used input pulled directly from the Dota 2 game environment.

One of the more varying aspects between each project are the structures that each use. Some networks, like AlphaGo/AlphaGo Zero and OpenAI Five, performed using parallel networks while Playing Checkers without Human Expertise utilized a single processing network instead. Parallel structures allow for communication between networks that increase efficiency and decrease training times while using large input data sets. However, not all environments produce large enough data sets and create large enough search trees that a parallel structure would be effective. However, when using such a structure is appropriate, the potential that the network has to handle extremely complex tasks is much greater.

Finally, the training techniques used by each network varies as well. While most neural network projects utilize forms of stochastic gradient descent algorithms for modifying weights, differences in how the network learns from the input can vary based on the network type, whether the network has continual access to sample data, whether the network maintains a memory of its previous iterations and other factors. NeuroChess is a great example of the potentially short training times when using a set of predefined data. However, networks like OpenAI Five, while training times are much longer, demonstrate the potential of using only sets of previous iterations by the network itself. The use of techniques like reinforcement learning is also effective is applied appropriately given the proper reinforcement type. Playing Atari Using Deep Reinforcement Learning demonstrates the need for appropriately evaluating the reward systems for individual reinforcement learning applications. The effectivity of different training techniques is dependent on the use of the network, type of input, and overall application of the neural network model.

Neural networks continuously demonstrate the future that the computational model has for advanced learning and planning tasks. The application of these networks in continuously changing environments continuously proves the capability of modern computing. AlphaGo/AlphaGo Zero, Playing Atari Using Deep Reinforcement Learning, NeuroChess, OpenAI Five, and Playing Checkers Without Human Expertise are five influential neural network projects that demonstrate the capability that this computational model has in complex plan development tasks. By observing these projects and the components that they share, it is shown that the complexity of the plan development problem has countless possible approaches using a neural network model.

Fig 5 – Findings	Efficiency Components	Input Components	Structure Components	Training Components
AlphaGo/AlphaGo Zero	-utilizes a lookahead search -64 GPUs and 19 CPUs -optimized with Stochastic Gradient Descent	-represented with a single input vector	-four processing layers -single pass for each iteration	-utilizes a four-phase Monte Carlo Tree Search -3 days training time -4.9 million games throughout training -deep reinforcement learning
Playing Atari with Deep Reinforcement Learning	-preprocessing for input flattening -84x84 pixel vector -utilizes final-frame capture for optimization -optimized with Stochastic Gradient Descent	-seven varying input vectors for each environment	- single pass for each iteration -convolutional	-reward proportion modification -deep reinforcement learning
NeuroChess	-explanation based learning for optimization -work divided across workstations	- represented with a single input vector	-parallel neural network structure	-temporal difference learning
OpenAI Five	-proximal policy optimization -128,000 CPU cores -256 P100 GPUs -1,048,576 observations per day - optimized with Stochastic Gradient Descent	-20,000 value representations and list of 8 actions for each agent -multiple input nodes	-five recurrent parallel networks	-deep reinforcement learning with Rapid
Playing Checkers Without HumanExpertise	-small input vector	-represented with a single input vector	-five layers with 3 hidden layers	-proprietary evolutionary algorithm -minimax alpha/beta search

References

Bottou, Léon. "Stochastic gradient descent tricks." *Neural networks: Tricks of the trade*. Springer, Berlin, Heidelberg, 2012. 421-436.

Brockman, Greg, et al. "Openai gym." *arXiv preprint arXiv:1606.01540* (2016).

Chellapilla, Kumar, and David B. Fogel. "Evolving an expert checkers playing program without using human expertise." *IEEE Transactions on Evolutionary Computation* 5.4 (2001): 422-428.

Domingos, Pedro. "A unified bias-variance decomposition for zero-one and squared loss." *AAAI/IAAI 2000* (2000): 564-569.

Doya, Kenji. "Reinforcement learning: Computational theory and biological mechanisms." *HFSP journal 1.1* (2007): 30.

Enzenberger, Markus, et al. "Fuego—an open-source framework for board games and Go engine based on Monte Carlo tree search." *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010): 259-270.

Eskin, Eleazar et al. "A Geometric Framework For Unsupervised Anomaly Detection: Detecting Intrusions In Unlabeled Data". *Columbia University Press*, 2014, Accessed 17 Aug 2019.

Granter, Scott R., Andrew H. Beck, and David J. Papke Jr. "AlphaGo, deep learning, and the future of the human microscopist." *Archives of pathology & laboratory medicine* 141.5 (2017): 619-621.

Harrington, Peter. "Stochastic Gradient Ascent". *Machine Learning In Action*, 2011

Hinton, Geoffrey E. "How neural networks learn from experience." *Scientific American* 267.3 (1992): 144-151.

Knuth, Donald E., and Ronald W. Moore. "An analysis of alpha-beta pruning." *Artificial intelligence* 6.4 (1975): 293-326.

Krogh, Anders. "What are artificial neural networks?." *Nature biotechnology* 26.2 (2008): 195.

Mallya, Arun. “Introduction To Recurrent Neural Networks”. Illinois State University, 2015,

McGregor, Anthony, et al. "Flow clustering using machine learning techniques." *International workshop on passive and active network measurement.* Springer, Berlin, Heidelberg, 2004.

Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar. *Foundations of Machine Learning.* MIT Press. (2012). Accessed 16 Aug 2019.

Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

Olah, Christopher. "Understanding LSTM Networks -- Colah's Blog". Colah.Github.Io, 2019, http://colah.github.io/posts/2015-08-Understanding-LSTMs/#lstm-networks.

Pachoki, Jacob. "Openai Five". Openai, 2019, https://openai.com/five/.

Pachoki, Jacob. "Openai Five". Openai, 2019, https://openai.com/blog/openai-five/.

Saha, Sumit. "A Comprehensive Guide To Convolutional Neural Networks". Medium, 2019, https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Sainath, Tara N., et al. "Deep convolutional neural networks for LVCSR." *2013 IEEE international conference on acoustics, speech and signal processing.* IEEE, 2013.

Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." *Neural networks 61* (2015): 85-117.

Geman, Stuart, Elie Bienenstock, and René Doursat. "Neural networks and the bias/variance dilemma." *Neural computation* 4.1 (1992): 1-58.

Schulman, John, et al. "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347* (2017).

Silver, David, et al. "Mastering the game of go without human knowledge." *Nature 550.7676* (2017): 354.

Silver, David et al. "Deterministic Policy Gradient Algorithms". *Deepmind Technologies* (2017)

Tesauro, Gerald. "Temporal difference learning and TD-Gammon*." Communications of the ACM* 38.3 (1995): 58-68.

Thrun, Sebastian. "Learning to play the game of chess." *Advances in neural information processing systems*. 1995.

Venkatachalam, Mahendran. "Recurrent Neural Networks". Medium, 2019, https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce.

Wang, Fei-Yue, et al. "Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond." *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016): 113-120.

Watkins, Christopher JCH, and Peter Dayan. "Q-learning." *Machine learning* 8.3-4 (1992): 279-292.

Webb, Geoffrey. *Bias–variance decomposition, In Encyclopedia of Machine Learning*. Springer 2011. pp. 100–101