Game Applications of Deep Neural Networks

Savanna Endicott Carleton University
savannaendicott@gmail.com

December 15, 2017

Abstract

In the past ten years, neural networks have made a significant impact on machine learning by enabling computers to make decisions previously difficult for them to accomplish. This predominantly began as recognizing photographic content, and is now being applied in a variety of contexts. Recently, there have been significant achievements for neural networks that select best moves to play in games. DeepMind's AlphaGo defeated 18-time Go world champion Lee Sedol in 2016, and since then neural networks have been given serious consideration in this field. This study analyzes which games are best suited to a neural network solution, and how to design a game specific neural network to successfully select best moves.

I. Introduction

Neural networks are systems inspired by neurons in the human brain, with the goal of achieving tasks previously too difficult for computers.

Neural networks are currently in popular use for speech and image recognition, and are becoming an important research development in the field of autonomous vehicles. With the success of DeepMind's AlphaGo against 18-time Go world champion Lee Sedol, they have also broken into the game playing industry. How applicable this style of machine learning is to general gameplay, however, is still unknown. The research discussed in this paper will is intended to help identify which games make for good applications of neural networks and how to design them.

This research is intended to aid in the design of neural networks used to select moves in games, particularly strategic board games. This involves strategies for deciding whether or not deep learning will apply well to a given game, as well as guidance for design of the system and important decisions that arise. The research is supported by the design of a player that uses deep learning strategies to select moves in the game of Blokus.

Developing a neural network approach to a game is no easy task, due to computational capabilities of the average developer, understanding of calculus and neural networks, and limited learning resources. This paper is meant to reduce the learning curve, by describing the types of problems commonly faced, and the factors that contribute to an approach's success. As successful implementations of neural networks in game play is a recent development, background knowledge about important events in this field will also be imparted, along with predictions of future research.

II. Concept Overview

Before getting into specific applications and implementation details, it is important to describe some of the major concepts of machine learning and define terms that will appear in this paper.

In general, artificial intelligence refers to the ability of computers to perform tasks that would normally require human intelligence. In the context of this research, it refers to decision making. Machine learning is a method of solving these tasks that does not directly compute the solution, but learns how to solve tasks over time using knowledge gained from previous attempts. This is an approach intended to solve problems in the same manner that a human would, while still taking advantage of computational capabilities beyond that of a human. The terms *deep learning* and *deep neural networks* are used interchangeably. They refer to neural networks that are unsupervised and learn from unlabeled data. This means the desired output isn't necessarily known and the network doesn't depend on the initial sample pairs of input and output values. To achieve success without knowing the desired output we use reinforcement learning, which generally put is the ability for a system to behave in a way that is learned from the environment's feedback.

The general structure of a Neural Network appears as follows:

As seen in *Figure 1*, neural networks tradi-

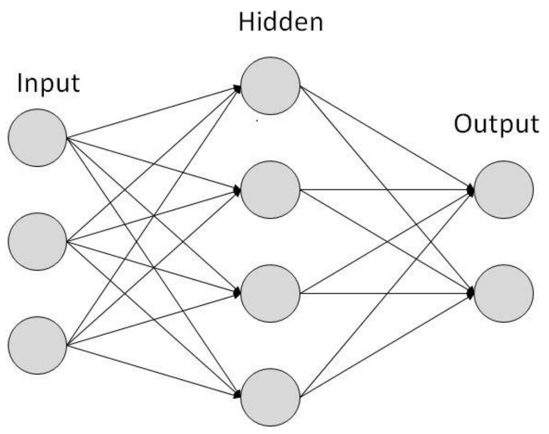

Figure 1: *Neural Network with a Single Hidden Layer from tutorialspoint.com/artificial_intelligence*

tionally have one input layer and one output layer, in addition to 1-to-many hidden layers. The input layer neurons are what we call *features* of an input variable. For example, in the case of images, the input variable could be a image represented as 64 pixels. Possible input features would be red qualities of the pixels, green quantities of the pixels, and blue quantities of the pixels, each also being an array of equal dimension to the photo. The output value would then be 1: cat, 0: not a cat.

With these concepts and terms defined, we can now proceed into a background of applications of neural networks, and discuss finer details.

III. Background

Two of the most successful and common uses of deep learning are speech and image recognition. One of the first and arguably most widely known achievements of neural networks was presented by Google in 2012 in the form of identifying images of cats (Le et. al, 2012). Although a relatively trivial task, the implications are very important. This provided evidence for the use of neural networks to complete tasks that humans are able to do with ease, but previously could not be achieved reliably by computers. This work launched a series of developments in image recognition using neural networks which is still highly active today. Though an important achievement for deep learning, one could say Google was only scratching the surface.

Google now uses neural networks in language processing, another large application area of deep learning. Google Assistant speech recognition AI uses deep neural networks to better understand the user's verbal requests and commands. Apple's Siri uses deep neural networks to recognize when she is being spoken to (Siri Team, 2017). Apple also used deep neural networks to create on on-device face detection, whose debut came with iOS 10 (Computer Vision Machine Learning Team, 2017).

Additionally, neural networks are currently generating a lot of development in the field of autonomous vehicles (Tian et al., 2017). These networks are relatively more complex than the methods described in this research, as they generally use hybrid/custom neural network architectures.

IV. Developments in Go

In 1997, IBM's Deep Blue defeated a reigning grand master in the game of chess. It used brute force deterministic tree search methods along with the aid of human knowledge. This was one of the first breakthroughs in the field of game playing for artificial intelligence. This method cannot apply to Go, because the search space for Go is much larger. The amount of moves a player has to consider at any given time is much larger than chess, and each move has a large impact on the rest of the game.

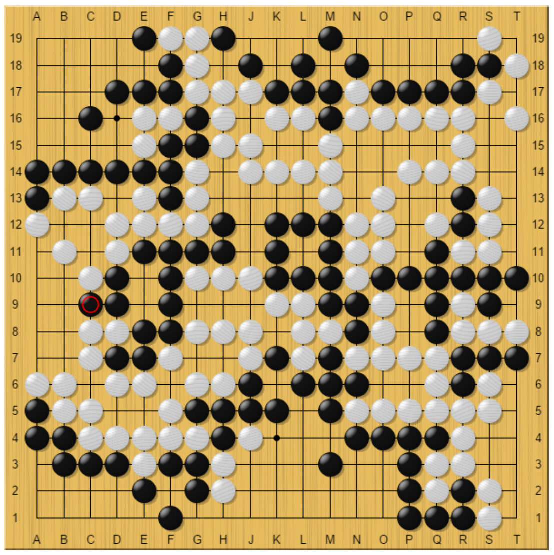

Figure 2: *Example Board State of a game of Go from gobeginnings.files.wordpress.com*

The first big advancement in learning to play go was using Monte Carlo Search Trees. The basic structure of these are trees that select random moves from root to tree until they reach the result of a game, then propagate back up through the tree and update the weights of each move. *Backpropagation* is an important concept used in supervised learning whose purpose is to correct itself from error over time. In 2008, a system called MoGo achieved dan level in 9x9 Go, and in 2012, the Zen program defeated an amateur 2 dan player on a standard 19x19 board. Monte Carlo search trees were gaining some traction, but the programs could not compete with the best Go players in the world. This didn't happen until DeepMind's AlphaGo came on the seen in 2016.

DeepMind Technologies is a British company acquired by Google in 2014. DeepMind published results of using deep learning to play 7 Atari games in 2013 (Mnih, 2013), including Pong and SpaceInvaders. Although this was a significant step in the right direction, the major influence that sparked interest of neural networks for games was AlphaGo, which uses deep learning to select moves in the game of Go (Silver et al., 2016). AlphaGo beat Lee Sedol in a five game match, with a score of 4:1. AlphaGo was awarded an honorary 9-dan (master) level for this achievement. This was a huge breakthrough in machine learning because it forced the world to take game playing neural network machines seriously.

AlphaGo uses policy and value networks and a Monte Carlo tree search. The policy networks include a supervised learning policy network trained on expert move data, a fast policy used to quickly sample actions during rollouts, and a policy that performs reinforcement learning in order to optimize outcomes. The value network predicts the winner of games played by the reinforcement learning policy against itself. These policies are combined with Monte Carlo tree searches, discussed earlier in this section. More details of the implementation will follow as we discuss implementing our own neural network approach to the game of Blokus.

When implementing Blokus, I used AlphaGo to gather inspiration and ideas of how to design the various components. It is important to note that when I discuss "AlphaGo's" details, I am referring to the version published in January 2016, *Mastering the game of Go with deep neural networks and tree search*(Silver et al., 2016)

V. Designing the Input for the Neural Network

Input design is an essential part of implementing a game specific neural network. It has the ability to make or break a project, and in my opinion directly determines how effective a system will be. Before creating a model of the input, it is important to determine if we *can* create a model that will result in an effective system. In this section we will discuss what aspects of a game make neural networks a good or bad choice, the first steps to take, and important design decisions and considerations.

i. Model Complexity

An important question to ask in order to determine whether or not to use a deep learning approach is to compare the representation of states for the game to the structure of common representations used in successful implementations. For example, an image. If your game state is very similar to an image, it is likely a good application of neural networks. But what does it mean to be similar to an image? As mentioned before, an image would be represented as a grid, where each position in the grid can be represented by a set of attributes with identical structure, and this accumulation of states gives a full representation of the game's state.

This is the case in the example of Go. Each grid position has a state of black, white, or empty. Then additionally we can use attributes like the liberties available and legality of positions to add context. We will discuss the design of Go further as we go along as it is not important to fully understand what this means right now. The key point is that simple attributes of each square in the grid can create a full image of the state, and any empty space should be considered as a possible move.

In the case of Chess, we also have a grid where we can store attributes. However, these attributes are significantly more complex. For example, there are 6 different pieces that can be in any given square. Each of these pieces can be played differently, compared to Go where each piece has the same capabilities. Additionally, the number of spaces a player can move to on the board at any given time is more limited than Go.

What should be noted here is the difference between move complexities. In Go, the results of each action are quite different from each other and relatively complex in their dynamics, but the complexity of the actual action is very low. There are 19x19 places to move on the board. Each has the same attributes as the others, with different values for those attributes. Each of the values is also relatively simple, with 2-3 possible values. This can be represented in a very similar fashion to an image. In Chess, on the other hand, a piece can only move to certain locations and each possible piece/location action has a unique result. The problem is that there is no continuity in the board's representation. What is meant by this is that the value of positions are not closely related, even if they only have one changed value of input (e.g. piece used). Representing this in a way that will be easily interpreted by a neural network is very difficult, and has not been found to play better than various human-knowledge based AIs.

Not only should we strongly consider this state-action model before we decide to implement the neural network, we should also put a large emphasis on the model's design throughout the project. A mistake I made during the development of a neural network for Blokus within a small amount of time, (approximately 100 hours including learning about machine learning and AlphaGo) was to quickly move forward with a model that made sense to me at that time. I would have achieved much more success in my project if I had taken the time early on to understand the implications of my problem to the model, and what aspects of the model were more challenging in the context of Blokus.

Another factor to consider would be the number of possible moves in the entire game. For example, in Go we wouldn't expect to play 10 games before seeing a given move twice. There are games, such as Blokus, as we will see, where some moves are very rare, and so comparing the probability of that move winning compared to frequently seen moves is most likely not representative.

In my opinion, designing the input model is the most important aspect of designing a neural network. To do this well, one needs to have an understanding of how neural networks manipulate this model in order to develop an understanding of move quality. The ability to design a strong representational model of the input facilitates and speeds up the process drastically. In my case, with a very brief learning period before beginning, I took a trial and error approach to the design of the model, which had negative impacts on the project. In order to avoid this in the future, important design components of the input will be discussed, and examples from AlphaGo and my implementation of Blokus will be given.

ii. State representation

First we need to decide how to model our state. The state is actually simply the variable we will be using to create our features from. In most board games, we can use a basic board state that would be maintained over the course of the game anyways. *Figure 2* is an example of a board state in Go. In this research, these are the only types of games that are discussed. The state should be a simple 2 dimensional array with the state of each box in that array. For checkers a possible state used to create features from would be an 8x8 array where each position would have a value of 0 for empty, 1 if a white piece was there, or 2 if a black piece was there.

Each time we go to make a move, we use our state as input to the neural network. We can also use the result of the game in the case of a supervised learning policy. Then when selecting moves using our network, we can send the state that would result from each possible move, and that will return the move with the highest probability to win by comparing with past moves.

iii. Feature Design

Designing features for games is relatively complex. It is important to select features wisely, as they will affect how your network perceives the game state. They should include all basic features of the state, as well as any attributes that apply to each position of the board that are not biased. A trick I use to determine which features to use is to ask myself if it provides knowledge about the state. Anything that provides knowledge about the state that would otherwise be unknown should be included in the features. Anything that does not, should not. I was inclined to add to the feature details that would perhaps encourage the network to select actions that I would play, given my strategy; however this does not make sense for input features to a neural network.

It is a bit hard to explain exactly how to select features, but it will be more clear as we look at the examples of AlphaGo and Blokus. The features used for input to AlphaGo's networks can be examined in *Figure 3*, a table I made based off *Extended Data Table 2* from *Mastering the game of Go with deep neural networks and tree search* (Silver et al., 2017).

Figure 3 states a number of planes asso-

Feature	# of Planes
Stone colour	3
Ones	1
Turns since	8
Liberties	8
Capture size	8
Self-atari size	8
Liberties after move	8
Ladder capture	1
Ladder escape	1
Sensibleness	1
Zeros	1

Figure 3: *Input features for neural networks*

ciated with each feature. These planes are N_1xN_2 matrices, where N_1 and N_2 are the dimensions of the game's board. In the case of AlphaGo, N_1 and N_2 are both 19. So a feature for some state of the board would be represented by 48 matrices with dimensions 19x19, which is a lot of data. When trying to understand this, I had difficulty at first as I do not understand the game of Go very well, and have never played. To clarify, the reason *Liberties* have 8 planes is because 8 binary planes are used to represent whether any given position is a corner with x liberties. So each plane is for a specific number of liberties from 1 to 8. Almost every plane is a direct inference of the state, and includes rules from the game. Looking at the features, it can perhaps start to become clear why AlphaGo was so successful.

The features used clearly give a strong representation of a state and it's implications on a board, in such a way that the features of planes can be compared to each other. The high number of features as well as the standard shape work well together in the hidden layers, where they are represented in various forms. However, the issue is that this is a very complex set of features, which requires a lot of computational power to use. I have come across many relatively professional and advanced open source imitations of AlphaGo, and none of them were able to use this set of features due to the required time and space complexities. That said, some were able to achieve the abilities of an amateur dan player using a subset of the features.

iv. Action Representation

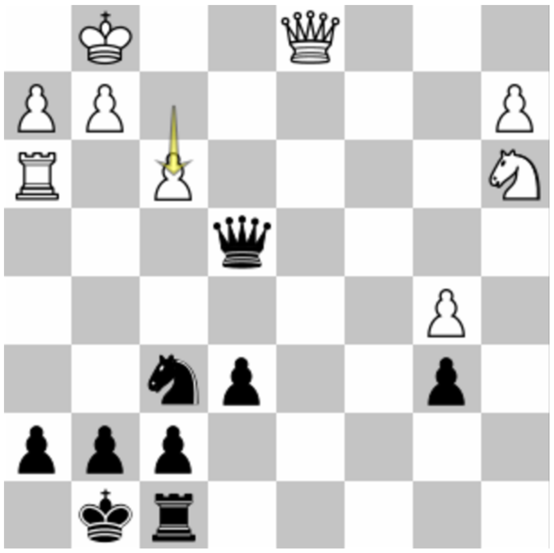

Figure 4: *Example Chess Move from chessfox.com*

Action representation is also important, and more difficult in some scenarios. For example, in Go, it is quite simple, as it can be represented by a position on the board given by x and y between 0 and 18. Actions that cannot be represented this way, however, need some sort of alternative. I will give Chess as example, because despite not being a good application of neural networks, it is a widely understood board game that I can use to give examples of game features. Additionally, using it as an example will expose some of the reasons why deep learning doesn't apply as well. In chess, an action could be represented as two positions on the board: the starting position and the position the piece moved to. For example, in *Figure 4* the move could be represented as ([2,1],[2,2]). However, that only works if you include some information regarding piece type inside your features. The features of that state would need to result in quite different planes in order for us to use them to determine which piece to use (and not only which position). Consequently, we introduce a barrier between each possible action, which again explains the challenges of chess and other games that produce more complex action states. The more we can reduce the number of possible actions and increase their relation to each other, the better our network will be able to interpret this information.

VI. Designing the Network

When designing a network, we first need to choose an architecture. Convolutional Neural Networks (CNN) are commonly used in game contexts. I have limited knowledge of the details of other forms of neural networks. However, the type of input produced by a board game is very similar to that of an image. CNNs are the architecture of choice for image recognition, so intuitively, they would apply best to our problem. This would be in comparison to Recurrent Neural Networks(RNN), generally more useful for areas such as speech recognition, which does not provide obvious overlaps with the representation of game states. A third option would be to create a custom architecture, perhaps a hybrid CNNs and RNNs as is sometimes used for autonomous vehicle applications. Having limited experience with deep learning, I decided not to take this approach.

In AlphaGo, the reinforcement learning policy neural network could be interpreted as the "brains" of the operation, despite the "expert knowledge" used by the supervised learning network. The end goal is for the system to *beat* the expert players, and not to play similarly to them, by learning in ways that leverage computational abilities beyond that of a human brain. Reinforcement learning also helps us to recover from flaws in the system. An important example of a flaw mended by reinforcement learning is overfitting to data. A metaphor for this could be a child learning from only one person their whole life, or a small group of people with similar opinions. The child's brain may become overfitted to its peers limited way of thinking. Another flaw it helps avoid is learning from incorrect or damaging actions.

Gradient Descent iteratively updates a set of parameters to minimize these types of errors. The reinforcement learning policy of AlphaGo uses Stochastic Gradient Descent (SGD) to update its weights at every step through the tree that does not result in the game ending. The difference between Gradient Descent and SGD is that the former requires you to run through all samples in the training set to update one parameter, where as the latter only uses one training sample or a small subset. Using regular Gradient Descent with a large amount of training samples would take significantly longer. The trade-off is that the error is slightly less minimized in SGD; however, SGD offers a significant advantage in that it makes progress right away with each new sample.

The structure used by AlphaGo for both the Supervised Learning and Reinforcement Learning policy networks alternated between convolutional layers and rectifier nonlinearities, and its last layer is a softmax layer that outputs a probability distribution over all legal moves. The design of convolutional layers is critical to designing a convolutional network. To get an idea of how to approach this, it is important to understand the different attributes of the layers. One of the important features of each layer is the activation function used.

i. Activation Functions

AlphaGo uses the Rectifier Linear Unit (ReLU) activation function for all of its hidden layers. There are several choices to use for activation functions, and it can be beneficial to try a couple of different ones, if not for the network, then for learning. Commonly used activation functions and their parabolas can be seen in

Figure 5.

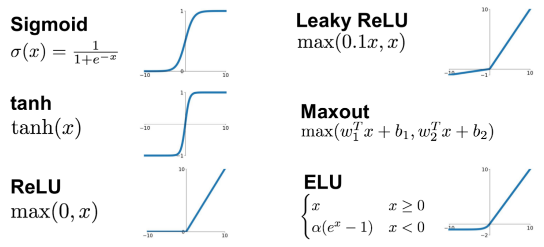

Figure 5: *Activation functions from cdn-images-1.medium.com*

In Andrew Ng's course *Neural Networks and Deep Learning* (Coursera inc., 2017), he explains that the sigmoid function is a good choice only when working with binary classification. When the input to the activation function is either very large or very small, the slope of the function is close to 0, slowing gradient descent. ReLU is currently the most successful and widely-used activation function (Ramachandran et. al., 2017). The reason it is such a popular option is because it lessens the effect of this Gradient Descent slowing, speeding up the entire learning process significantly. However, there are other options to consider. A successful imitation of AlphaGo that I found claims that using Elu greatly sped up the learning (McGlynn, 2016). Additionally, the *Leaky ReLU* activation function as seen in *Figure 5* has been claimed to improve speeds and fault handling by a group from Stanford University (Maas et al., 2013).

ii. Convolutional Layers

TensorFlow simplifies the creation of convolutional neural network layers (Google, 2017). To see which input parameters work best, a solution could be to simply try all the classic example inputs and compare their results. However, it is perhaps better to actually understand the parameters that go into these layers, in order to get a better understanding of how the entire architecture functions.

TensorFlow has a function tf.layers.conv2d (Google, 2017) which, among other things, takes in inputs, the number of filters, the dimensions of the filters, padding specification, and an activation function. Input is straightforward, feed it the input layer constructed (see section V. Designing the Input for the Neural Network). Understanding what the filters are used for is important to understanding how these convolutional layers work and what they achieve.

The idea behind convolutional layers is that each filter is an area that will be looked at and checked for some sort of signal activation. For example in images it could be a curve or edge. The math behind this is a dot product of each filter with a small piece of the input matrix. The more filters we have, the more preservation we have of the image. However, they also correspond to the output dimension of our features, so using a higher number of filters will not necessarily provide any additional information and will drastically grow our computational requirements. For the dimensions of the filter, this is a general question of how large of an area contains highly dependent pixels. For example, if we said 1X1 for the filter, we would be stating that pixels are *completely independent* of their neighbors, where as if we go bigger, we are saying large areas have pixels that are related to each other in a significant way. In images, if we were looking for, say, a cat, we would want to use a filter big enough that it could recognize a curve without the entire paw. Of course we don't know how big the paw will be in the photo, which is a reason to use a variety of filter dimensions in different layers of the network.

Another commonly specified attribute is the *stride*. This is simply the number of positions you shift horizontally or vertically before taking a new filter. For example if you had a 3x3 filter size and took your first filter from $x = 0...2$ and $y = 0...2$, the next filter would jump x and y to a certain new number.

The default stride is 1, so in this case we would have $x = 1...3$, $y = 1...3$. The TensorFlow input for this is an array of length 4. The first and last items in the array are the samples and the features, which have to be set to 1. The other two items in the area are these width and height strides that can be specified. In the case of games, it is most likely better to keep the default, however there is no harm in trying things out.

I will not go into the reasoning behind each of the selected parameters for AlphaGo, as these values are implementation-specific; however, *Figure 6* below gives a brief breakdown of the parameters used in AlphaGo's 12 hidden layers.

Layer	Details
1	• zero pads the input into a 23x23 matrix • Applies 192 5x5 filters with stride 1 • Applies ReLu
2-12	• pads the previous hidden layer into a 21x21 matrix • Applies 192 3x3 filters with stride 1 • Applies ReLu
Output	• Applies 1 1x1 filter with stride 1 • Applies position bias 1 • Applies softmax function

Figure 6: *Details of AlphaGo's Policy Neural Network*

VII. Implementation of Blokus

As part of this research, I began the implementation of a neural network player for Blokus, with a model inspired by AlphaGo. I designed the input to the neural network and its features, implemented the game and neural network. However, I ran out of time before I was able to determine how to feed the input properly into the reinforcement learning network. This section will include a description of the game, the design of input for the neural networks, challenges faced during the project, and future work.

i. Blokus Rules

Blokus is traditionally a 4-player game played on a 20x20 board. Each player has 21 pieces, each of which has a unique shape. The shapes are based on free polyominoes between 1 and 5 squares. Each turn, a player has to play a piece if they are able. On their first turn, they must play a piece such that a tile of the piece lays in their designated corner tile of the board (e.g. [0,0], [19,19], ...). They can only play a piece in a given location if one of its corners will be attached to the corner of another of its pieces, and if none of its side's lengths are against the length of another of its pieces (see *Figure 7*).

ii. Input Design

When starting the design and implementation of Blokus, I did not have the level of understanding displayed in this document. It did not stand out to me that Blokus could have similar issues to Chess, and did not contain some of the important features that made Go applicable for a neural network approach. As discussed in section V. Designing the Input for the Neural Network, this game contains a significantly complex action state that does not offer a simple solution for highly related input features. As in Chess, the piece one uses results in a

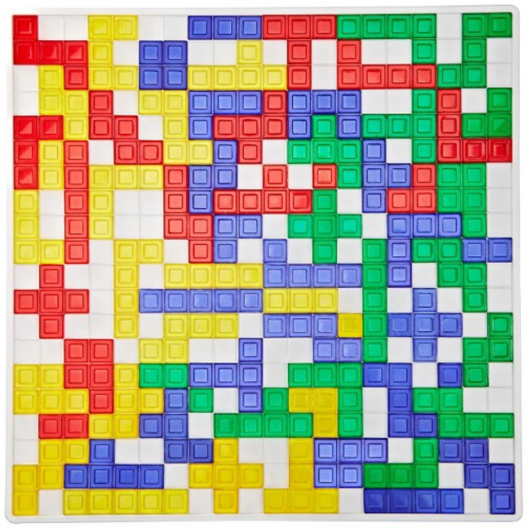

Figure 7: *Example Blokus board layout from https://target.scene7.com*

unique action and, consequently, has a unique effect on the value of the move. Unfortunately, this applies to Blokus more dramatically. Not only are there 21 different pieces as opposed to 6, but they have different states. The different states are unique rotations and flips of the piece, where each unique state forms independent features as in chess. Fortunately, not all of them have 8 unique states, one for each of the four rotations before and after flipping. *Figure 8* displays the number of shapes that have x unique sets of indices for each shape, meaning all flips and rotations that result in a unique set of tile positions and indices, where x is between 1 and 8. This means there are consequently 102 unique actions for *each* position on the board (where the move fits in the boundaries).

# of unique forms	# of shapes
8	7
4	10
2	2
1	2

Figure 8: *The number of unique tile positions for each shape*

Compared to AlphaGo's 1 possible state per position and even chess' 6 states per position, this already presents doubt that Blokus will be a good application for neural networks. Additionally, the number of moves possible at a given state in the board is more similar to chess than to Go, as there are several options for pieces and only a limited number of locations they can be placed. The number of pieces/shapes you can place is greater than chess; however, the number of locations where a corner will be available is significantly less. Generally, the features of the variety of actions will not be continuous.

Selecting features took a significant amount of time, as I did not visualize the problem in a way that was appropriate for neural networks. It took a lot of analyzing and trial and error to understand how the feature planes of AlphaGo are formed and how they are used, as it is not explained. Once I had a better understanding of deep learning, I was able to understand AlphaGo's approach, and finally design the feature planes for Blokus. *Figure 9* displays the feature planes I selected for Blokus.

Feature	# of Planes
Tile colour	5
Liberties	4
Sensibleness	1
Zeros	1

Figure 9: *Input features for neural networks*

The main reason that there are only four types of feature planes is that there are fewer rules in Blokus. At first, I had wanted to make biases according to strategies I support; however, this would be incorrectly implementing a neural network. There are a total of 11 features planes here. The colour planes are the same as AlphaGo except that there are 4 players instead of 2, so there are 2 extra planes. The liberties in this case refer to tiles that are attached to corners, are empty, and do not share side lengths with another piece of that player's colour. Each player's planes are used to enabled a blocking strategy. It is a binary plane, where 1 means the piece is a liberty and 0 means it is not. The planes will represent the possible *locations* for each player to go, but does not acknowledge piece options. For example, if there is an empty liberty where a player can go but only with their single-square piece and they have already played it, it will count as a liberty even though they cannot play there. Sensibleness is the legality plane, that shows places the player could place a tile, without acknowledging attachments. A tile is legal if it is empty and none of its edges is adjacent to a tile with that player's colour.

The features are created from the player's colour and the board state, a 19x19 matrix with values -1, 0, 1, 2, or 3, where -1 means the location is empty and the other four are representative of each player having a tile there. The action is represented as a 5-tuple *(piece, x, y, rotation, flip)* where piece is a number 1-21 identifying the piece of choice from a list, x and y are the "center tile" positions for the piece, rotation is the number of spins clockwise between 0 and 3, and flip is 1 or 0 (flipped or not flipped).

iii. Pain Points and Major Blocking Sources

As discussed in section ii, the first major block I had in this project was trying to design a proper input model for the neural network, and generally trying to understand how AlphaGo's network functioned. Once I had a solid understanding, had implemented my input to the neural network including features, and had set up basic TensorFlow classes and methods, I was nearly out of time. The major stepping stone I did not overcome, primarily due to time, was simply feeding the input into a neural network. TensorFlow's data formatting examples for self-created input to their functions are exclusively images, and the process of creating persistent tensors and sessions is vaguely described. I found the documentation quite difficult to understand. The examples I used were image-based, with the exception of AlphaGo (McGlynn, 2016). AlphaGo feeds inputs to the network behind the scenes, as each move is simply a position (x,y) in the board state. The training function outputs a set of probabilities for each position in the board state, and McGlynn (2016) simply chooses the index with the maximum probability from that list. Having a 5-tuple action representation and over 40,000 possible moves, this is not feasible for Blokus. Moving forward, I would like to find a solution to this problem that is more efficient. My thinking was to simply run samples (possible actions) through the network one by one and keep track of probabilities produced for possible actions. However I hope to find a cleaner and more dynamic solution. A helpful strategy would be to use a smaller board, however this greatly changes the game, and does not actually aid in a representation of the possible moves - only lowers their quantity.

VIII. General Implementation Advice

Through the learning process of this project, I have developed a list of tips for implementing a neural network for games. This discusses high level decisions and strategies to use while programming as well as the research process used to gather knowledge required for this type of project. This advice comes from my own personal experience, and should therefore be considered along with the context of a project and resources available.

1. *Do not start by learning the low level math.* Read about convolutional neural networks, their layers, and their architectures first, before trying to understand the detailed math behind these concepts. Although the math is important to understanding how deep learning works, a higher level understanding of how neural networks function and interact should come first. This will allow you to develop a model of

how these concepts apply to your project early in the process. After these concepts are understood, more knowledge of their details will be required and more easily absorbed.

2. *Take the time to understand input model designs before implementing one*. Although designing an input model seem intuitive, having a full understanding of how your design will be interpreted by a neural network will greatly reduce implementation time. This will result in a better chance of moving into the network design earlier in the project's timeframe.

3. *Focus on one reinforcement learning policy network*. Although a set of neural networks identical to AlphaGo's seems appealing, these networks may not be necessary for a different project to be successful, and may require Google-level computational capabilities to achieve. Generally, the reinforcement learning policy network should be held at a higher priority than the supervised learning network. The supervised learning network by itself does not represent the capabilities of the neural network, it represents the capabilities of the players who created the data. Supervised networks should only be used as a tool to help kick-start the reinforcement learning policy's learning. The reinforcement learning policy is the policy that, well, does the machine learning! It is where the designs of input and the network itself can be tested and improved. Once content with a version of this policy, additional networks can be added to enhance the system or to compare with AlphaGo's abilities. Time should be allocated appropriately to all aspects of the project in order to prioritize the development of the reinforcement learning policy, which leads to the next point.

4. *Don't spend a lot of time on the game development or its output*. I spent at least half of my time implementing the game and producing output samples in a way that I thought would speed up the development process, and it had the reverse effect. I developed 3 AI players based on an alpha-beta mini-max strategy, with different heuristics. The purpose of the AI players was to provide a variety of input data for the supervised learning network. This is another example of time wasted due to a lack of understanding. I then formatted the sample game output in various ways to help determine desired input to the network, which took a very long time. Making matters worse, the AI players are quite slow, playing one game in over a minute. If I were to go back and redo the process, I would not develop the AI players at all. As mentioned in the last principle, I would have saved a lot of time focusing on the reinforcement learning network development and feeding it features, then modifying those features as I went. The changes needed for the game itself are discovered along the way, and hard to predict. Trying to implement them before hand (e.g. functions needed for features, sample output formatting and creation, parsing of that sample data) wasted a lot of time, and a lot of it ended up not being used.

5. *Learn a simple, similar open source implementation vigorously*. Find a project on GitHub that is *finished* and well documented, and go through the important files. These projects tend to include a ton of unused and possibly incorrect files, so understanding a full walkthrough of used code in a program similar to yours will help you know what to look for when analyzing other projects. Additionally, anytime they exclude something that would normally be included, even if there's an explanation, try including it first. What didn't work for them might still work for you, and if not then you can develop theories as to why it can have a harmful effect on the results.

6. *Try to keep the focus of your learning as di-*

rected as possible. Neural networks are becoming very popular, and reading about various projects and applications is very interesting. That said, all of this information can make your project much more confusing than it has to be. Try to research as much as possible inside the area of relevance of your project without straying into other interesting applications of deep learning. Of course, this only applies to significantly short-term projects. If the goal was to come up with unique ways of developing a neural network to play a game, researching a wide variety of successful architectures in unrelated topics would be important. This could also identify what specific qualities of an architecture apply in various contexts. Although focusing on relevant material is important, narrowing your research too quickly can also be problematic.

7. *Don't jump right into TensorFlow*. TensorFlow is a great tool, and can save a lot of time, but trying to use it before having programmed smaller, more generic steps is difficult if a developer is unfamiliar with the library. Sometimes code that is the simplest is the hardest to use, because its simplification makes it difficult to read and understand. Additionally, using library methods that encompass many complex steps makes code difficult to test and debug, especially if those steps aren't understood. This is a case where I would recommend Andrew Ng's Neural Networks and Deep Learning course (Ng, 2017). The course includes programming assignments that break down each component of a neural network, and therefore imparts an understanding of the work being done by TensorFlow behind the scenes.

IX. Future Work

Any future trials of my project will include integrating the input into the neural network to test the system as a whole, trying out different activation functions and/or a combination of them, using different filter quantities and dimensions, specifying strides, and more. The substitution of smaller sized boards will need to be tested as well, as it may greatly reduce the difficulty of the problem.

Additionally, I would like to try other games that may be more applicable to neural networks. I think it would be interesting to investigate uses of neural networks in non-board based games other than Atari games or first-person-shooter games, where they are currently most popular. The project could also take a major turn and attempt to be completely unsupervised, as inspired by AlphaGo Zero.

AlphaGo Zero is a new version of AlphaGo that does not leverage any human expertise data (Silver et. al.,2017). AlphaGo Zero defeated the version of AlphaGo discussed in this paper by 100 games to 0, as well as defeating the world champion in Go, Ke Jie, in 2017 (Silver et. al., 2017). AlphaGo Zero is arguably the strongest Go player in history, human or computer. Only published this October, the research paper contains techniques that could potentially be used to facilitate the implementation of neural networks for games that do not have expert data available or are difficult to produce data for.

X. Conclusion

Deep learning is currently a very popular development, being experimented with in very interesting fields such as speech and image recognition, autonomous vehicles, and recommendation systems. I believe a large reason this is such an important area of development is due to the combination of computational abilities with human abilities. Learning is a very important characteristic of the human brain that, if leveraged correctly, could result in a variety of systems that behave much more dynamically than current artificial intelligence.

Theoretically, a player of a strategic board game that can attain the same knowledge of a human expert while leveraging the efficiency of move analysis for a machine, should defeat any human player. This type of machine would also avoid human-like mistakes by not "missing" anything, if it was correctly designed. AlphaGo is a huge step toward this. However, the field has a lot of developments to be made that can be applied to a larger variety of games. This research is meant to develop an understanding of AlphaGo's strategic strengths as well as its shortcomings when applying its techniques to other games. Despite the fact that Blokus is unfinished, I feel that I have learned a lot from applying AlphaGo's techniques to it, and hope to have more success in the future.

XI. References

1. Le, Q et. at. "Building High-level Features Using Large Scale Unsupervised Learning", *Proceedings of the 29th International Conference on Machine Learning, Edinburgh, Scotland, UK, 2012*, Google, 2012

2. Mnih, V, et al. "Playing Atari with Deep Reinforcement Learning." *DeepMind*, DeepMind Technologies, 1 Jan. 2013, deepmind.com/research/publications/playing-atari-deep-reinforcement-learning/

3. Maas, A, et al. "Rectifier Nonlinearities Improve Neural Network Acoustic Models",*Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013*, JMLR:W&CP, 2013

4. Silver, D, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature News*, Nature Publishing Group, 27 Jan. 2016, nature.com/articles/nature16961

5. McGlynn, G. "Go-playing neural network in Python using TensorFlow" GitHub, 10 May 2016, github.com/TheDuck314/go-NN

6. Ng, Andrew. "Neural Networks and Deep Learning" *Coursera*, Coursera inc, Jan. 2017, coursera.org

7. Tian, Y, et. al. "DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars" arXiv:1708.08559v1 [cs.SE], 28 Aug. 2017, https://arxiv.org/pdf/1708.08559.pdf

8. Siri Team. "Hey Siri: An On-device DNN-powered Voice Trigger for Apple?s Personal Assistant" *Machine Learning Journal*, Apple inc., 2017, machinelearning.apple.com/2017/10/01/hey-siri.html

9. Siri Team. "An On-device Deep Neural Network for Face Detection" *Machine Learning Journal*, Apple inc., 2017, machinelearning.apple.com/2017/10/01/hey-siri.html

10. Ramachandran, P, et. al. "Searching for Activation Functions" *Cornell University Library*, arXiv:1710.05941 [cs.NE], 16 Oct. 2017, arxiv.org/abs/1710.05941

11. Silver, D, et al. "Mastering the game of Go without human knowledge." *Nature News*, Nature Publishing Group, 18 Oct. 2017, nature.com/articles/nature24270

12. Google, "tf.layers.conv2d" *TensorFlow 1.4*, Creative Commons 3.0 Attribution License, 2 Nov. 2017, $tensorflow.org/api_docs/python/tf/layers/conv2d$