Using Monte Carlo Tree Search Methods for Real-Time Champion Recommendations in League of Legends

COMP 400 Research Project Report – James Ting

Abstract

League of Legends is the most popular e-sports game in the world. With many champions to choose from, all with complex non-linear relationships, players can have difficulty choosing a champion that will have the highest probability of winning. Any potential champion recommendation system would have to provide accurate recommendations before the champion selection period for the player has ended. Since this period is approximately 30 seconds, the algorithm must be able to return a result before the time has ended. This project explores the viability of using Monte Carlo Tree Search (MCTS) for real-time recommendation systems of champions to users. This project considers several techniques and reward functions and demonstrates MCTS with a neural network reward function provides the best balance between time complexity, speed, and prediction performance.

1. Introduction

League of Legends [1] is the most popular multiplayer online battle arena (MOBA) in the world with 8 million daily players [2]. In 2019, the League of Legends World Championship tournament brought in 44 million peak concurrent viewers [3], while the next most popular e-sports title, Defense of The Ancients 2 [4] had 1.9 million peak viewers during its world championship [5].

In each game of League of Legends (LoL), two teams of five players each fight to destroy a core structure on the enemy base, called the Nexus, while defending their own. With 154 unique playable characters [6] (called "champions") to choose from, all of which have complex and non-linear relationships, players can easily struggle to choose an appropriate champion for any situation. Due to the high number of champions, there are approximately 3.86×10^{17} unique team combinations. Furthermore, each player has at most 30 seconds to select a champion, leaving limited time to make a choice. Currently, there is no such tool that assists players in selecting champions depending on team composition and current selection. Players must rely on their understanding of game mechanics, information available online, and champion performance statistics to select an appropriate champion. However, players' understanding of game mechanics may be incomplete or incorrect, players have limited time to find information online, and individual champion performance statistics aggregated by third party sites only provide average champion performance across all encountered team compositions. The burden of selecting a champion can be especially heavy on new or casual players, who lack the sufficient game knowledge to make an appropriate decision. New players may not even be aware of the many third-party sites that aggregate information and help provide information to players.

Monte Carlo Tree Search (MCTS) is a heuristic tree search method used to find optimal decisions for a given domain. The algorithm takes random samples in the decision space and builds a search space based on the results of the sample. MCTS has been a popular area of AI research, due in no small part to the success of MCTS in the game *Go*, where human players had traditionally been far more advanced than computer players. MCTS methods are useful in domains where the search space is very large, as it will balance the exploration of new nodes and exploitation of existing nodes. Further, since MCTS converges to *minimax* [7], MCTS is a suitable alternative for combinatorial games.

This project aims to demonstrate that MCTS is a viable algorithm to provide real-time champion recommendations to players, based on the champions that have been selected thus far, that maximize the probability of winning. The project will demonstrate that champion selection is a two-player zero-sum game with perfect information. Any useful champion recommendation system must satisfy the following requirements:

1) Recommend a champion that maximizes the probability of winning.
2) Provide a recommendation within the allotted time for players to select a champion.

Building such a recommendation system will benefit many players at all skill levels. Choosing the correct champion will help experienced players progress to the higher ranks faster, professional players gain a competitive edge over their opponents, and new players overcome the steep learning curve of the game.

First, we obtained a dataset of matches, including data about champions and winning teams. As a baseline technique, the filtering technique counted win rates of each encountered team combination and searched the dataset for similar teams to the provided team. To provide recommendations, the filtering technique returns the combinations with the highest win rate. We then developed a MCTS algorithm that searches through the tree representing the champion selection process and considered several reward functions for the MCTS. We compare the latency, accuracy, and memory usage of each technique, and consider the scalability of each technique. We also demonstrate that MCTS is more suitable for providing recommendations for champions selection in a real-time system, with latencies in the order of a few seconds.

2. Background

2.1 League of Legends

In this section, we provide an overview of the LoL gameplay and mechanics. Figure 1 presents a high-level view of Summoner's Rift, the main map in the ranked mode of LoL at the beginning of each match. At the beginning of each match each player selects a champion, a playable in-game character, to play for the duration of each match. In the map, there are three lanes, and each lane has three towers, destructible structures that incurs large amounts of damage to enemies in the radius. To reach the Nexus, a team must destroy the all the structures in at least one lane, while defending their own structures. The team that starts in the bottom-left corner of the map is the blue team, and the team that starts in the top-right corner of the map is the red team. Typically, four of the five players in each team will be in lanes, with the fifth player being in the jungle out of the lanes. Players will have to face their opponent for the beginning of the game and choosing the correct champion could help players gain a lead to win. Therefore, the selection of the champion is crucial to gaining an edge over opponents.

Figure 1: A bird's eye view of Summoner's Rift, the main map in the Ranked Mode of League of Legends. [8]

At the beginning of each match, each team can discuss strategy and see what the other players have selected, to build teams that synergize[1] well. Before champion selection, each player can ban a single champion, removing that champion from the option space. Players then choose champions in an alternating in a 1-2-2-2-2-1 order, where the first team selects one champion, the second team selects two champions, and so on until all players have selected, with the second team selecting the last champion. As a result of this pick order, the search space of choices consists of pairs of champions. This adds complexity to the search and requires more time to generate choices.

The importance of selecting champions in the game is a crucial phase of the game. Teams that select the appropriate champions are more able to exploit the weaknesses of their opponents and leverage their skill or knowledge of a certain champion to gain an advantage over the opponents. The importance of this portion of the game is compounded by the large possibility space for team combinations. Given that each player can choose from a pool of 154 champions [6], we can estimate the number of unique team combinations to be approximately $|T| = \binom{154}{5} \times \binom{149}{5} = 3.86 \times 10^{17}$. Further, since champions have complex non-linear synergies and counters[2] with other champions, situation-appropriate champion selection can be a challenging task for many players.

2.2 Methods

2.2.1 Recommendation systems

Recommending algorithms are well established in academic and industry applications, with various techniques such as data mining, content-based, context-aware methods and more [9]. However, most algorithms cannot not efficiently search such a large search space, and/or provide recommendations in real-time, while taking into account input data that is also changing in real time. A common recommendation algorithm is Apriori associative rules. Apriori association rules are common in e-commerce and multimedia systems to recommend relevant items to users. The algorithm will search through the dataset and find the

[1] "Synergy" is term used by players to refer to how well each champion works with other champions on their team. Teams where champions enhance the strengths of other teammates, while also covering weaknesses synergize well, whereas teams that do not synergize poorly.

[2] A "counter pick" or "counter" is a term commonly used by players to refer to when an opponent selects a champion that is particularly strong against a champion picked by the player or their teammates, typically due to the design of the champion and the opponent champion.

most frequent item set. The algorithm relies on the fact that any item set that occurs frequently must also have any subset occur at least as frequently [10]. Given a threshold C, the algorithm identifies the set of items which are subsets of at least C sets in the database [11]. However, the Apriori algorithm is expensive in terms of time and space complexity, with a time and space complexity of $O(2^{d+1})$, where d is the total number of items in the dataset [12]. With such a large search space, this algorithm would be unlikely to converge in time, and is therefore unsuited for this task. Furthermore, the algorithm can only find items that tend to appear together, but not necessarily items (champions) that are appearing together and winning. Such a model has the possibility of recommending a set of champions that appear frequently and have poor probability of winning, resulting in popularity bias.

Two algorithms that are worth considering for a real-time recommendation system are collaborative filtering, and Monte Carlo Tree Search. Collaborative filtering techniques are common in e-commerce as well as multimedia for recommendation systems, and are able to provide recommendations to players that maximize enjoyment [13]. Monte Carlo Tree Search, however, is an algorithm primarily concerned with finding optimal decisions in a decision space and is able to provide updated decisions as more data is provided. Since the champion selection is a decision, this technique is useful to find the optimal champion choice in a given selection state.

2.2.2 Collaborative filtering

The principal goal of collaborative filtering is to suggest items that a user will rate highly, based on past items that the user rated highly and the ratings of other users. In essence the algorithm recommends items that are typically purchased or highly rated by similar users. There are two principal types of collaborative filtering techniques, user-based and item-based algorithms [14]. Since the aim of the project is to build a user-agnostic tool, the focus will be put on item-based models. This family of algorithms consists of two main steps [15]:

- Similarity computation: Compute the similarity of two items using a similarity function.
- Prediction computation: Using the most similar items, look at the target user ratings and use a technique to obtain predictions.

To compute the similarity between two items, several techniques are available. Methods such as cosine-based similarity, correlation-based similarity and adjusted-cosine similarity are common techniques [15]. Since cosine similarity is a relatively simple calculation, it has low latency and memory usage, which are crucial in a real-time system. Given two items, they can be thought of vectors in a m-dimensional space, $i, j \in \mathbb{R}^m$. The cosine similarity function defined as:

$$\text{sim}(i, j) = \frac{i \cdot j}{||i|| * ||j||}$$

where the numerator is the dot product of i, j. Items which are identical will have a similarity score of one, and orthogonal items will have a similarity score of 0.

Next, for the prediction computation, the algorithm can then take the set of most similar items, and then use techniques such as a weighted sum or regression to output items that are similar to the user's past choices, and the choices of other likeminded users.

Using this technique for a champion recommendation system, the algorithm will select the champion combinations in the dataset that are most similar to the provided champion state using the cosine-based similarity. The algorithm will return the top k recommendations, sorted by win rate.

2.2.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a family of heuristic search algorithms that employ multiple Monte Carlo simulations within decision trees. In general, all MCTS build decision trees in an incremental and asymmetric manner, and then on each iteration of the algorithm, a Tree Policy estimates which currently leads to the most optimal decision. The tree will be built incrementally by adding a single child node at each iteration of the MCTS algorithm. The tree is built asymmetrically because the algorithm selects a node that is worth exploring exclusively using the Tree Policy. A good Tree Policy should balance exploration of new nodes, and the exploitation of explored nodes. The algorithm then runs a domain-specific reward function and updates the tree based on the result of the reward function [16]. A domain specific reward function for champion selection would be a reward function that takes in a game state and then determines which team would win.

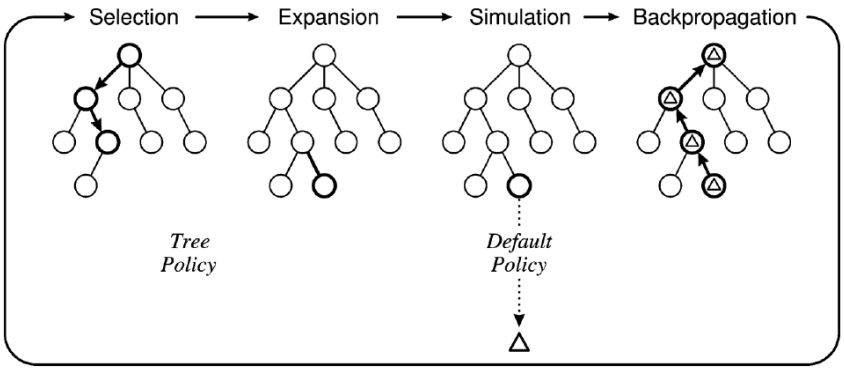

Figure 2: One iteration of the general MCTS technique [16].

Any MCTS algorithm will, at a high-level, follow these steps, illustrated in Figure 2:

1. Selection: The algorithm selects an interesting and expandable node. A node is expandable if it is not a terminal state, and at least one child node has not been expanded. The Tree Policy will determine which nodes are interesting and worth exploring further.
2. Expansion: The algorithm selects a random choice or action from the selected node, and then adds a new child node to the tree.
3. Simulation: Run a reward function on the expanded node using the Default Policy to estimate the outcome of the decision.
4. Backpropagation: Backpropagate the outcome (reward) up the tree to update the values of parent nodes.

Traditionally, MCTS have excelled in the domain of games. Go has an exceptionally difficult challenge for artificial intelligence. Due to the high branching factor, deep decision tree, and a lack of a known reliable heuristic for nonterminal positions, Go has a domain where human players have remained well ahead. In October 2015, Google DeepMind used a MCTS algorithm to play an unhandicapped version of 19x19 Go, beating the European world champion [17]. It is because of this that MCTS methods have seen a growing level of interest, with applications in other domains such as Settlers of Catan, automated complex material design, poker, and many more [18-20].

Upper Confidence applied to Trees (UCT) is a common MCTS algorithm [16] that approximates the game-theoretic value of possible actions from the current state. The pseudocode in Figure 3 shows a variation of

UCT for two-player, zero-sum games with alternating moves [16]. The general UCT algorithm is similar, however this variation for this project uses a different backpropagation function that negates the reward at each step. The general UCT simply propagates the reward throughout the entire tree.

Algorithm 1 The UCT Algorithm for two players

```
function UCTSEARCH(s_0)
    v_0 ← root node with state s_0
    while have computational budget do
        v ← TreePolicy(v_0)
        Δ ← DefaultPolicy(s(v))
        Backpropagate(v, Δ)
    end while
    return a(BestChild(v_0, 0))
end function
function TREEPOLICY(v)
    while v is non-terminal do
        if v not fully expanded then
            return Expand(v)
        else
            v ← BestChild(v, C_P)
        end if
    end while
    return v
end function
function EXPAND(v)
    choose a ∈ untried actions from A(s(v))
    add new child v' to v with s(v') = f(s(v), a) and a(v') = a
    return v'
end function
function BESTCHILD(v, c)
    return max_{v' ∈ children of v} Q(v')/N(v') + c·sqrt(2 ln N(v)/N(v'))
end function
function DEFAULTPOLICY(s)
    while s is non-terminal do
        choose a ∈ A(s) uniformly at random
        s ← f(s, a)
    end while
    return reward for state s
end function
function BACKPROPAGATE(v, Δ)
    while v is not null do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + Δ
        Δ ← −Δ
        v ← parent of v
    end while
end function
```

Figure 3: Pseudocode for a UCT algorithm for a combinatorial game. Adapted from [16]

Figure 3 shows the pseudocode for a UCT algorithm. Each node n of the decision tree for any MCTS algorithm contains four attributes: $\mathrm{N}(n)$ is the number of times that a node has been visited by the algorithm, $\mathrm{Q}(n)$ is the total reward of all playouts containing the action represented by that node, $S(n)$ is the state of the current node, and $\mathrm{A}(n)$ is the set of possible actions from the current node's state. In general, $\mathrm{N}(n)$ is a non-negative integer and $\mathrm{Q}(n)$ is a real-valued vector. In the case for this project, $\mathrm{Q}(n)$ will be a non-negative integer and will be the number of times that this state resulted in a blue team victory.

The return value of the entire algorithm will be the best child of the root node, which is the action from the current state that will give the highest reward. In general, the algorithm could also be made to return the

most visited child node [16], however in this project's implementation, it recommends champions that maximize the probability of winning, and so it selects the action that will give the highest reward.

UCT is a Tree Policy that aims to balance the exploration-exploitation dilemma. This Tree Policy has several advantages, which make it well suited for this task. It is simple and efficient to calculate and guaranteed to be within a constant factor of the best possible bound on the growth of regret [16]. The UCT equation is given by:

$$\text{UCT} = \bar{X}_j + 2C_P\sqrt{\frac{2\ln n}{n_j}}$$

where n is the number of times the current node has been visited and n_j is the number of times the j-th child has been visited, $C_P > 0$ is a constant and $\bar{X}_j$ is a value in the range $[0,1]$. For this project, $\bar{X}_j$ will be the estimated blue team win rate of the state represented by that node, or more formally: $\bar{X}_j = \frac{Q(n_j)}{N(n_j)}$. For example, consider three nodes, one parent node and two children and suppose we are making recommendations for the blue team. Let each node be a tuple $N_i = (q_i, n_i, A_i, S_i)$ where q_i is the number of times that node led to a blue team victory, n_i is the number of times each node has been visited, A_i is the set of actions, and S_i is the state of the node. Suppose the parent node has values $N_p = (4,6, A_p, S_p)$ and the children nodes have values $N_1 = (2,3, A_1, S_1)$ and $N_1 = (1,3, A_1, S_1)$. Suppose that $C_P = \frac{1}{\sqrt{2}}$. The UCT for N_1 would be $\frac{2}{3} + 2\left(\frac{1}{\sqrt{2}}\right)\sqrt{\frac{2\ln 6}{3}} = 2.212$ and the UCT for N_2 would be $\frac{1}{3} + 2\left(\frac{1}{\sqrt{2}}\right)\sqrt{\frac{2\ln 6}{3}} = 1.878$. We can see that based on the UCT, N_1 is currently more worth exploring. Intuitively, this is correct as we can see that N_1 has a higher estimated blue team win rate than N_2 and is therefore more worth exploring.

Adjusting the C_P exploration term affects the amount of exploration by the algorithm. For reward function rewards of between [0, 1], a value of $C_P = \frac{1}{\sqrt{2}}$ has been shown to satisfy the Hoeffding inequality [21]. Therefore, $C_P = \frac{1}{\sqrt{2}}$ was selected for the implementation of the UCT in this project.

MCTS methods have found their niche in domains where the search space is large, and where algorithms like *minimax* would take too long to converge [7]. With the space of champions being so large, MCTS is a possible strategy to algorithmically make champion recommendations in real time.

3. Contribution

3.1 Data Collection

The data for this project is collected from the Riot Games API, which provides data about matches and players in League of Legends. We built a NodeJS script to repeatedly issue requests to the API. We selected NodeJS for the scrip due to its ease of use when working with asynchronous tasks, such as issuing requests. On startup, the scraper reads the match that was last added to the dataset. Then the script selects a random participant in that match, enqueues all ranked matches played by that player. The script only collected data from ranked matches to ensure players would take the game seriously and reduce the effects of "trolling". The players that were targeted were players of average rank, with the assumption that a player's rank is correlated with their skill, to build a more generalizable model for the algorithm to be able to predict the outcomes of the most matches. The script will issue a request once every 2.5 seconds by default, which would reduce the chance of a "Too Many Requests" errors. The scraper collected data from a total of 1,357,379 matches from January 25th, 2021 to April 5th, 2021.

Once the NodeJS script had collected initial dataset, a Python script removed duplicate matches and filtered the remaining matches to extract the champion and outcome information. Since there is no upper bound on the length of the game and each match data object contains a timeline of player performance, each match response had a varying amount of information [22]. The NodeJS flattens each response to store the data in a row of a .csv file. We then selected most frequent match data length, to ensure commonality between all datapoints. The selected row length is 1150.

To build the dataset about the champions, another Python script selected the columns that contained the participant champion data, with the first five champions being the champion keys of the blue team, and the next five champions being the champion keys for the red team. For example, a row in the dataset is: $(57, 44, 124, 3, 21, 72, 151, 1, 59, 132, 1)$. For this combination, the blue team selected champions with keys $(57, 44, 124, 3, 21)$ and the red team selected champions with keys $(72, 151, 1, 59,132)$ and this game resulted in a blue team victory. Within the scope of this paper, champion key refers to an integer in the range $[1, N]$ where N is the number of distinct champions in the game, and champion ID refers to a non-negative integer the Riot API uses to identify champions. The Riot API docs contains a JSON file that is used to convert between the champion key and champion ID [23]. The final column contains a binary value that records whether blue team won the game. A value of 1 means a blue victory, and a value of 0 means red victory. The final dataset contained champion data for 984,976 unique matches.

3.2 Filtering for Team Combinations

For this task, since players cannot change their champions after selection, any useful recommendation must have the current state of the selection as a subset. The key difference between traditional collaborative filtering methods and the method implemented for this project is that traditional methods do not guarantee this constraint [15].

In the model implemented for this project, on startup, each combination of teams is recorded in a hash map, mapping to the win-rate of that combination. For the similarity calculation, the model searches the datasets for all possible combinations that could result from the provided team. More formally, let x be a set of length less than 5, where each element in the set is a champion key. The algorithm then searches the dataset to find all combinations $\mathcal{C}$ such that $x \subseteq \mathcal{C}$.

For the prediction calculation, the combinations that satisfied the subset condition were sorted by win rate. Then the algorithm returns the top k combinations with the highest win rate, where k is the number of recommendations requested by the user. To find the top recommendation, a linear scan is sufficient.

Figure 4 gives the pseudocode for the filtering technique. The algorithm creates a HashMap for each team's recommendations, then for each combination in the dataset, it determines if the input team is a subset of the combination. It then sorts each recommendation HashMap and returns the top k recommendations.

Algorithm 2 Filtering Recommendation system

```
function PREDICT(blueTeam, redTeam, k)
    blueTeamRecs ← new HashMap()
    redTeamRecs ← new HashMap()
    for combination in dataset do
        if isSubTeam(blueTeam, combination) then
            winRate ← dataset.get(combination)
            blueTeamRecs.insert(combination,winRate)
        end if
        if isSubTeam(redTeam, combination) then
            winRate ← dataset.get(combination)
            redTeamRecs.insert(combination, winRate)
        end if
    end for
    return sortHashmap(blueTeamRecs, k), sortHashmap(redTeamRecs, k)
end function
function ISSUBTEAM(subTeam, team)
    teamSet ← new Set(team)
    teamSubset ← new Set(subTeam)
    return teamSubset.isSubset(teamSet)
end function
function SORTHASHMAP(map, k)
    list ← list of keys ∈ map sorted by value in descending order
    result ← new Hashmap()
    for i ∈ [0, k) do
        result.insert(list[i], map.get(list[i])
    end for
    return result
end function
```

Figure 4: Pseudocode for filtering technique

3.3 Applying UCT to Champion Selection

For the UCT algorithm to be applicable, champion selection must be defined as a zero-sum two player combinatorial game. Chen et al. demonstrated that in DOTA 2, hero selection satisfies the requirements to be a combinatorial game [24]. This requires considering each team, composed of multiple players, as a single game-theoretic player and each game-theoretic player will always know what the other has selected and therefore has perfect information. Further, since only one team can win, the game is a zero-sum game. Since the champion selection phase of DOTA and League are nearly identical and differ only in the champions themselves, champion selection in League of Legends also satisfies the requirements to be a zero-sum two player combinatorial game. This will assume that all players on both teams share a goal: selecting champions with the highest probability of winning. Further, while different individuals collectively form each team, we can view the players in each team as a collective single game-theoretic player. Without loss of generality, let blue team be the team that selects first. Formally, champion selection is a combinatorial game with the following components [16]:

- $S \subset \mathbb{Z}^n$: the set of game states, where n is the number of unique champions. Each $s \in S$ is a n-dimensional tuple where each i-th index encodes the selection of i-th champion. The possible selection values are:

$$s_i = \begin{cases} 1 \ i\text{-th champion selected by blue team} \\ -1 \ i\text{-th champion selected by red team} \\ 0 \text{ otherwise} \end{cases}$$

The start state s_0 is a zero vector.

- $S_T \subseteq S$: the set of terminal states. A terminal state for champion selection is a state where each team has selected 10 champions. Each final state should contain exactly five indexes that are equal to 1 and exactly five indexes that are equal to -1. The rest of the indexes will be zero.
- $n = 2$: the number of players in the game.
- A: the set of actions. Each action is the player selecting a champion.
- $f : S \times A \to S$: the state transition function. This function, given a champion selection and a starting state, will give the next state as a result.
- $R: S \to \{0,1\}$: the reward function. This function, given a state, returns the outcome of that state. 0 is a red victory and 1 is a blue victory. The reward function is only defined on the set of terminal states.
- $\rho : S \to$ (Blue, Red) the player about to act in each state.

With these components, champion selection satisfies the conditions of a two-player, zero-sum, deterministic, sequential discrete game with perfect information. In other words, champion selection is a combinatorial game. Using these components, champion selection can be viewed as a combinatorial game, UCT can be applied to find optimal strategies. The pseudocode implemented for this project is given in Figure 3. No large modifications are necessary to apply UCT to champion selection, with the components described above.

3.4 Reward function selection.

A crucial component of the MCTS methods is the reward function. For any application of MCTS, a domain specific reward function simulates the playouts of the game and determines the rewards. A good reward function is crucial to the performance of MCTS because it provides the updates for the heuristic to search the tree. This project considers the following reward functions: random choice, majority class, cosine similarity, and neural networks. For all reward functions, the function takes a state and reward function metadata as input and will return 1 if the blue team should win, and 0 if the red won the simulation.

3.4.1 Random choice reward function.

The random choice reward function is quick and simple reward function. It is considered for this project as a lower bound on the latency and memory usage for a reward function. It works by simply using the built-in random function from standard Python module random. Random choice will return a 1 if the result from the call to random.random() is greater than 0.5, and 0 otherwise. It represents a random guess, where for each match, the function guesses which team will win with equal probability and then returns the result without considering the current game state. This computation is fast and runs in constant time relative to the size of the dataset. Further, it requires relatively little memory, the reward function is potentially scalable for a real-time system. While this calculation is very fast, it is relatively meaningless when it comes to making recommendations, as it does not recommendations based on the state. Furthermore, when the recommendation is made, the recommendation would be the result of the action that happened to win the most times in over most simulations, but not necessarily the recommendation with the highest probability of winning.

3.4.2 Majority class reward function.

The majority class reward function is another quick and simple reward function. Majority class methods are a useful baseline for classification systems, since they guarantee an accuracy equal to the proportion of elements in the majority class. Since each team is equally likely to win, we would expect the majority class to have an accuracy no greater than 50%. When configured to use this reward function, the UCT program will first load the win rate file, and then compute the win rate. The algorithm passes the computed win rate to the reward function as an argument. Then, if the computed win rate is greater than 0.5, it will return 1, representing a blue team victory. It will return a 0 otherwise, representing a red team victory. This reward function also runs in constant time and space with respect to the size of the dataset. It would also be highly scalable for a real-time system. However, it also suffers from the similar issues as the random choice, where the function does not consider the current state. This may result in the MCTS not searching a path that would have a higher probability of winning.

3.4.3 Cosine similarity reward function.

For a function that takes considerations based on the state, a cosine similarity function was used, similar to the kind described by [15]. This function takes as input the state of the game, and a HashMap mapping to team combinations and whether that game state led to a win or a loss. It checks for an identical copy of the state, and then if such a state exists, it will return the outcome of that game based on the dataset. If no such state exists, then the function will search the dataset and select the most similar state. It will then return the outcome of that state. The cosine function does consider the current state of the game and is able to make predictions on states that are similar combinations in the dataset. However, this algorithm is linear with respect to the size of the dataset and takes significantly longer than the random choice and majority class reward function.

3.4.4 Neural network reward function.

For this reward function, UCT uses a neural network trained on the dataset to predict the outcomes of the state. In a similar project aimed at recommending heroes in DOTA 2, Chen et al., considered several reward functions, including the majority class, logistic regression, gradient boosted decision tree, and neural networks [24], and found the neural network had the highest accuracy. Therefore, we selected a neural network reward function for the implementation of MCTS for this project.

At the start of the recommendation system, the program loads the neural network from the specified location. The program then initializes a new instance of the model using the neural network binary file. The UCT program then passes the network object as an argument for the Tree Policy for the program, and then the Tree Policy performs a forward pass on the network object. Initially, had a 2-node output layer, returning a vector $x \in \mathbb{R}^2$ and then the program would return the index of the maximum element of x. However, due to poor validation accuracy, we changed the final output layer to a single node with the sigmoid activation function, representing the probability that the blue team will win. We can see that an output layer with a sigmoid activation function is intuitively correct since, without loss of generality, the probability that blue team wins is the complement of the probability that red team wins. In game-theoretic terms, the game of champion selection is a zero-sum game.

Design and training of the neural network developed using PyTorch 1.7.1. The network has one hidden layer, with a configurable number of hidden units. The activation function of the nodes in the hidden layer that is ReLU, and dropout was used to reduce overfitting. We selected the Adam optimizer [25] and the Binary Cross Entropy with Logistic Loss, for better numerical stability [26]. This loss function is a categorical loss function for binary classification and is defined as:

$$L(y) = \sigma(-(y \log(p(y)) + (1 - y) \log(1 - p(y)))$$

where σ is the sigmoid function, y is the class predicted by the network, and $p(y)$ is the class prior of class y in the training set. The network takes as input a vector $s \in S$ representing the game state, and outputs a real number $x \in [0,1]$. This Tree Policy then interprets this real number as a blue team victory if greater than 0.5, and a red team victory otherwise.

4. Experimental Evaluation

A real-time recommendation system should satisfy the following requirements:

1) Provide accurate recommendations with latencies on the order of seconds, to ensure real-time usability and usefulness.
2) Have constant time complexity with respect to the size of a dataset, to ensure scalability for a large system.

These experiments will evaluate the mean recommendation win rate, latencies and memory usages for each method and reward function. The time and space complexity of each is also considered. The code for all the techniques in this project is available at: https://github.com/jamesbting/COMP-400-Research-Project.

4.1 Neural Network Performance

We empirically selected the hyperparameters to optimize the validation accuracy without overfitting. The training script includes options to configure the hyperparameters. It was found that with a hidden layer size of 256, learning rate of 0.0005, batch size of 75000, and dropout rate of 0.5. The dataset was randomly split with an 8:2 ratio for the training and validation set. The accuracy, loss, validation accuracy, and validation loss graphs are included in Figure 4.

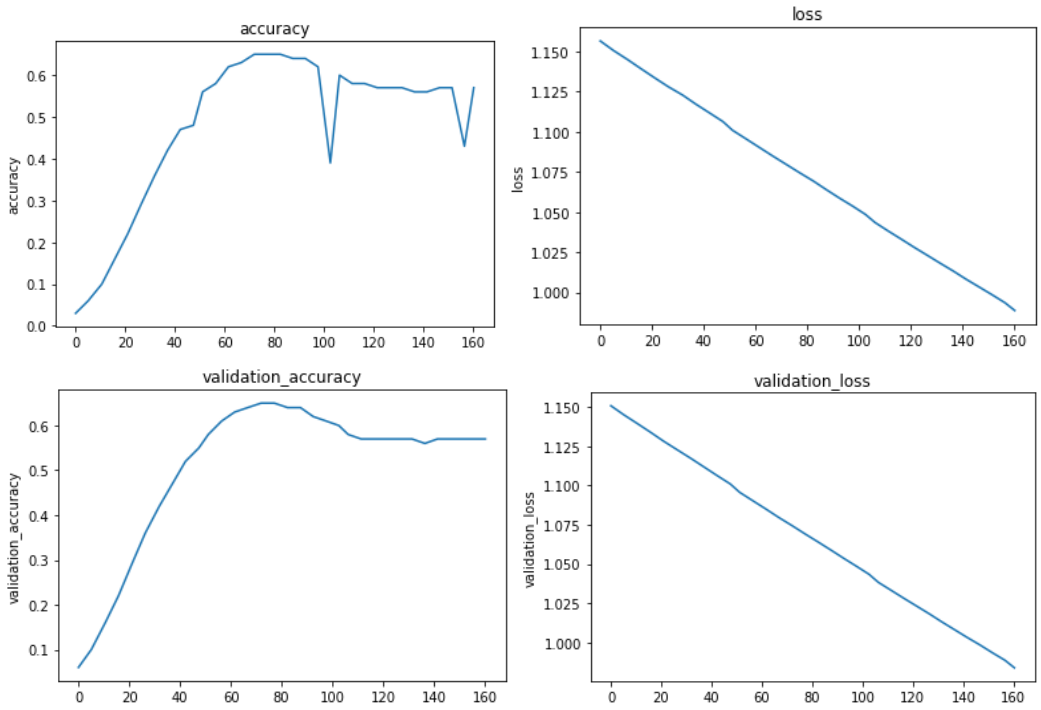

Figure 5: Graphs for accuracy, loss, validation accuracy and validation loss. The x-axis for all graphs is the time since training began.

While the validation accuracy is relatively low, the model can outperform a naïve model that simply guesses the blue team will win each time. Furthermore, while training the network, we observed that as the size of the training set increased, the validation accuracy tended increased. Therefore, it is possible that the dataset does not contain enough datapoints to learn the true distribution of champions and will likely benefit from a larger dataset. This is likely due to the large search space and popularity bias among players. A million unique team combinations would be approximately 2.52×10^{-12} % of the total search space, and therefore cannot capture the entire breadth of the relationships between champions. Adding information about champions that have been banned could slightly reduce the search space and improve recommendations. Further, since players do not select champions uniformly, the dataset is more likely to contain champions that players select more frequently. As a result, the neural network has the potential to have a higher validation accuracy.

4.2 Model performance

To evaluate the different methods, prediction performance, memory usage and latency are compared. Each system used the time and psuitl module to measure and record the time to make a prediction and the peak memory usage, respectively. For each algorithm and every reward function, except cosine similarity, the algorithm made 100 predictions, and recorded the time and peak memory usage. For cosine similarity, tests with only 15 predictions with 10 iterations of UCT for the sake of time. The computer conducting all experiments is equipped with an AMD Ryzen 5 3600 3.6 GHz CPU. The input for all tests is [121, 24, 18] for the blue team and [11, 26] for the red team. Since red team currently has less picks, the recommendation system would be making recommendations for the red team. Without loss of generality, all systems can make recommendations for either team.

To measure the accuracy of each model, the trained neural network computes the probability that, for a given recommendation, the blue team will win. Then, a Python script reads each recommendation, uses the neural network computes the win rate for each recommendation, and computes the mean win rate for each system.

System	Mean Blue Team Win Rate
Filtering	0.5047
MCTS – Random Choice	0.5130
MCTS – Majority Class	0.5122
MCTS – Cosine Similarity	0.5108
MCTS – Neural Network	0.5067

Table 1: Mean predicted win rate for the blue team for each recommendation system – lower is better.

Since the experiment aims to maximize the red team win-rate, we can see that the models with the best performance are the filtering and the MCTS using the Neural network reward function. Those systems will be more likely to produce favorable results for the users. The random choice, majority class and cosine similarity reward functions for the neural network preformed the worst. For the random choice and majority class, this would be easy to see since both reward functions do not consider the game state. The cosine similarity however, while improves on the random and majority class, still performs worse than the filtering and neural network. A possible reason for this could be the lack of iterations for MCTS, and therefore the algorithm is unable to sufficiently search the possibilities for a winning combination. For a real-time system to be able to provide recommendations to many users at a large scale, it must also be able to provide recommendations quickly without consuming too much memory.

The time and memory averages for the filtering technique are shown below:

Test	Average Latency (s)	99th Percentile Latency (s)	Average Peak Memory Usage (MB)	Mean Blue Team Win Rate
1	1.584	1.600	1018.85	0.5047
2	1.586	1.608	1018.98	0.5047
3	1.581	1.599	1018.89	0.5407

Table 2: Results from filtering, 1 recommendation, 100 experiments

We can see that the mean blue team win rate is consistent across the tests is because, for a fixed dataset and request, the algorithm is deterministic, and will always give the same recommendations for each request. While this filtering technique is relatively fast, and able to deliver a prediction in under the five second benchmark, algorithm depends on searching the dataset for suitable matches. Further, the algorithm can recommend multiple options, something that MCTS will require more iterations to match. However, this comes at the cost. The filtering technique has time and space complexity $O(n)$, where n is the number of matches in the dataset and is therefore unsuited for deployment in a real time system designed for end-users. In such a system, it will likely require on the order of billions of matches for a representative subset of the sample space. Therefore, collaborative filtering will likely struggle on a larger dataset, and is poorly suited for a real-time system.

Within the context of the MCTS tests, an iteration is one cycle of selection, expansion, simulation, and backpropagation. The time and memory averages for each reward function (random choice, majority class, cosine similarity, and neural network) of the MCTS are shown below:

Test	Average Latency (s)	99th Percentile Latency (s)	Average Peak Memory Usage (MB)	Number of nodes created per experiment	Mean Blue Team Win Rate
1	0.1054	0.1349	235.21	1002	0.5131
2	0.1054	0.1440	235.15	1002	0.5135
3	0.1062	0.1399	234.93	1002	0.5125

Table 3: Results from UCT with random winner, 1000 iterations, 100 experiments

Test	Average Latency (s)	99th Percentile Latency (s)	Average Peak Memory Usage (MB)	Number of nodes created per experiment	Mean Blue Team Win Rate
1	0.1050	0.1359	235.16	1002	0.5120
2	0.1044	0.1359	238.16	1002	0.5123
3	0.0940	0.1359	237.66	1002	0.5123

Table 4: Results from UCT with majority class, 1000 iterations, 100 experiments

Test	Average Latency (s)	99th Percentile Latency (s)	Average Peak Memory Usage (MB)	Number of nodes created per experiment	Mean Blue Team Win Rate
1	211.05	211.32	318.47	17	0.5108
2	209.08	210.67	318.22	17	0.5136
3	211.31	215.50	318.49	17	0.5148

Table 5: Results from UCT with cosine similarity, 15 iterations, 10 experiments

Test	Average Latency (s)	99th Percentile Latency (s)	Average Peak Memory Usage (MB)	Number of nodes created per experiment	Mean Blue Team Win Rate
1	0.2551	0.2900	238.34	1002	0.5069

2	0.2538	0.2889	238.18	1002	0.5068
3	0.2530	0.2829	238.26	1002	0.5064

Table 6: Results from UCT with neural network, 1000 iterations, 100 experiments

In the initial design of the UCT, each game state was a vector $s \in \mathbb{Z}^{154}$. Such a vector can be thought of as the "long game state". However, the memory usage and time to compute was significantly higher. Since such a vector would be mostly sparse, the game state can also be represented as a vector $s \in \mathbb{Z}^{10}$ where the k-th index is a positive integer representing the champion selected. The first five indexes are for the blue team and the next five indexes represent the champions for the red team. These two representations are equivalent, and therefore the implementation of UCT uses the lower dimensional game states, which greatly reduced the latency and memory usage of the program.

For the random winner and majority class, to make 100 predictions with 1000 iterations of UCT is, unsurprisingly, faster than the other reward functions. However, neither of these reward functions consider the provided state. Therefore, the recommendation is unlikely to be any better than a random choice of champion or a choice by a knowledgeable player. Furthermore, other systems have better accuracy of predictions, and therefore, despite the scalability of each reward function, other recommendation systems can provide more accurate recommendations.

The cosine similarity and the neural network are more interesting cases to examine. The cosine similarity, while lower than the neural network in memory usage, takes significantly longer than the neural network, even while running far fewer simulations. Since allowing UCT to perform more iterations tends to improve the result [16], this reward function is also unlikely to give results that are accurate. The high latency of the cosine reward function, which in turn reduces the number of iterations of UCT and hurts the accuracy, means this recommendation system is especially poorly suited for a real-time system. This should not be a surprise however, as the time complexity of UCT with the cosine reward function is $O(nk)$, where n is the size of the dataset and k is the number of iterations, where as a UCT with a constant time reward function would be $O(k)$. Furthermore, the cosine similarity also tends to have worse prediction accuracy, when compared to the other systems considered in this project. Therefore, the cosine similarity will be unfit for a real-time system.

The neural network, with a prediction time of approximately two seconds, strikes a balance of accuracy and latency. It had the second-best prediction and speed performance, while having a roughly constant time complexity. Further, because the reward function is a neural network, the validation accuracy of the network collecting more data and refining the model hyperparameters would improve the validation accuracy, and as a result, the UCT algorithm will be able to improve the recommendations. As a result, developing such a neural network and dataset, and then deploying such a system would be a research area worth exploring. A disadvantage of this model is that the neural network struggles to learn when the champion state is a vector $s \in \mathbb{Z}^{10}$. This is likely due to the non-linear relationship between the champion keys and their performance in the game. Therefore, before making a simulation with the neural network, the game state is to a long game state and then the long games state is the input for the neural network. The cause of the higher latency of the neural network when compared to the random and majority class is likely a mix of the state conversion cost and the added cost of performing a forward pass through the neural network. Training a neural network to learn the reduced game state would further improve the performance and memory usage of UCT. Future works could examine building a larger dataset and increasing the accuracy of the neural network.

A useful real-time recommendation system must provide accurate recommendations with low latency, while being scalable to a much larger dataset. As demonstrated by the experiment, we find that collaborative

filtering has the best mean recommendation win rates while having reasonable latencies and memory usage. However, collaborative filtering is linear with respect to the size of the dataset, and to improve the accuracy, a larger dataset is necessary. For the different reward functions used in the UCT algorithm, random choice and majority class were found to be highly scalable and fast but have lower accuracy. Cosine similarity reward function satisfied none of the requirements, and the neural network was fast, scalable, and accurate. The neural network also has the potential to improve the accuracy and provide better recommendations without increasing latency.

5. Conclusion

In this project, the viability of Monte Carlo Tree Search as a real-time recommendation system for champions in the game League of Legends. By viewing champion selection as a combinatorial game, and applying MCTS methods, the model can search an exceedingly large search space of possible team combinations and make recommendations about which champions to play. Of the systems and reward functions used, filtering and UCT with a neural network had the most accurate predictions, while being computationally in expensive. UCT with the random choice and majority class had worse predictions, and the UCT with cosine similarity is computationally very expensive. This project demonstrated that using a UCT algorithm with a neural network reward function satisfies the requirements for a large-scale, real-time recommendation system available to end users.

One drawback of such technique is the lack of player information data. Since most players only play a subset of champions, the algorithm may recommend to a player a champion they have never played before. While on average, selecting this champion may provide a better win rate, the player using such a recommender system may not be as skilled, and result in a loss. Expanding the game states to include player specific information is possible, however will come at the cost of greatly increasing the amount of data required to be stored at each game state and potentially increasing latency.

There are many future research directions for this project. Building a dataset on high-ranking players could potentially lead to better accuracy of the neural network, since these players typically have high skill on many champions and a far more advanced knowledge of the game. A neural network may be able to learn better on such a dataset and be able to deliver better predictions. However, that recommendation system would be less applicable to the general population of players. Growing the size of the dataset to contain billions of matches could also help the neural network train and allow the neural network to learn on a larger dataset. Modifying the MCTS to use player specific information will allow the network to accept that information to make predictions, however it will greatly increase the amount of data stored at each game state, since the tree will now need to contain information about each players' skill on each champion for all the players in the game, and any other data about the players that the neural network could potentially use to make predictions. Another improvement could be to parallelize UCT, thereby increasing the number of iterations, or reducing the latency.

Acknowledgments. I would like to thank my mentor, Oana Balmau, for providing feedback at every step of the project, especially during the early steps of the project and the suggestions on the final report.

Bibliography

[1] I. Riot Games. "League of Legends." https://na.leagueoflegends.com/en-us/ (accessed.

[2] K. Webb, "More than 100 million people watched the 'League of Legends' World Championship, cementing its place as the most popular esport," in *Business Insider*, ed, 2019.

[3] L. Staff. "2019 World Championship Hits Record Viewership." Riot Games, Inc. https://nexus.leagueoflegends.com/en-us/2019/12/2019-world-championship-hits-record-viewership/ (accessed 08-04-2021.

[4] V. Corporation. "DOTA 2." https://www.dota2.com/home (accessed.

[5] S. Yakimenko. "Viewership results of The International 2019." Esports Charts. https://escharts.com/blog/results-the-international-2019 (accessed 04-08-2021.

[6] I. Riot Games. "Choose your Champion." https://na.leagueoflegends.com/en-us/champions/ (accessed March 27th 2021, 2021).

[7] B. Bouzy, "Old-fashioned Computer Go vs Monte-Carlo Go," in *IEEE Symposium on Computational Intelligence and Games*, Hilton Hawaiian Village, Honolulu, Hawaii, April 1-5 2007. [Online]. Available: https://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf. [Online]. Available: https://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf

[8] L. o. L. Wiki, "Summoner's Rift," ed: MOBAFire.

[9] A. G. Guy Shani, "Evaluating Recommender Systems," *Recommender Systems Handbook,* 2010, doi: https://doi.org/10.1007/978-0-387-85820-3_8.

[10] T. I. Rakesh Agrawal, Arun Swami, "Mining Association Rules between Sets of Items in Large Databas," *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD Interational Conference on Managment of Data,* 1993, doi: https://doi.org/10.1145/170035.170072.

[11] N. K. G. Abhishek Saxena, "Frequent Item Set Based Recommendation using Apriori," *International Journal Of Science, Engineering and Technology Research,* vol. 4, no. 5, pp. 1609 - 1612, 2015. [Online]. Available: http://ijsetr.org/wp-content/uploads/2015/05/IJSETR-VOL-4-ISSUE-5-1609-1612.pdf.

[12] H. H. Imam Tahyudin, Hidetaka Nanbo, "Time Complexity Of A Priori And Evolutionary Algorithm For Numerical Association Rule Mining Optimization," *International Journal Of Science, Engineering and Technology Research,* vol. 8, no. 11, pp. 483 - 485, 2019. [Online]. Available: https://repository.unmul.ac.id/bitstream/handle/123456789/3898/5.%20IJSTRvol8i11-Time-Complexity-Of-A-Priori-And-Evolutionary-Algorithm-For-Numerical-Association-Rule-Mining-Optimization.pdf?sequence=1&isAllowed=y.

[13] D. S. Y. Tiffany D. Do, Salaman Anwer, Seong Ioi Wang, "Using Collaborative Filtering to Recommend Champoins in League of Legends," in *2020 IEEE Conference on Games*, Osaka, Japan, 2020: IEEE, pp. 650-653, doi: 10.1109/CoG47356.2020.9231735. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9231735

[14] D. H. John S. Breese, Carl Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," *UAI'98: Proceedings of the Fourteenth conference on Uncertainty in Artifical Intelligence,* pp. 43 - 52, 1998. [Online]. Available: https://dl.acm.org/doi/10.5555/2074094.2074100.

[15] G. K. Badrul Sarwar, Joseph Konstan, John Riedl, "Item-Based Collaborative Filtering Recommendation Algorithms," *WWW '01: Proceedings of the 10th Interational Conferance on World Wide Web,* pp. 285 - 295, 2001. [Online]. Available: https://dl.acm.org/doi/10.1145/371920.372071.

[16] E. P. Cameron B. Brone, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton, "A Survey of

Monte Carlo Tree Search Methods," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES,* vol. 4, no. 1, 2012.

[17] A. H. David Silver, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature,* vol. 529, pp. 484-489, 2016, doi: https://doi.org/10.1038/nature16961.

[18] G. C. István Szita, Pieter Spronck, "Monte-Carlo Tree Search in Settlers of Catan," *ACG 2009: Advances in Computer Games,* pp. 21-32, 2010. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-12993-3_3.

[19] S. J. Thaer M. Dieb, Kazuki Yoshizoe, Zhufeng Hou, Junichiro Shiomi, Koji Tsuda, "MDTS: automatic complex materials design using Monte Carlo tree search," *Science and Technology of Advanced Materials,* vol. 18, no. 1, pp. 498 - 503, 2017, doi: 10.1080/14686996.2017.1344083.

[20] K. D. Guy Van den Broeck, Jan Ramon, "Monte-Carlo Tree Search in Poker Using Expected Reward Distributions," *ACML 2009: Advances in Machine Learning,* vol. 5828, pp. 367 - 381, 2009. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-05224-8_28.

[21] C. S. a. Levente Kocsis, Jan Willemson, "Improved Monte-Carlo Search," 2006. [Online]. Available: http://old.sztaki.hu/~szcsaba/papers/cg06-ext.pdf.

[22] I. Riot Games. "Riot Developer Portal - Match." https://developer.riotgames.com/apis#match-v4/GET_getMatch (accessed.

[23] I. Riot Games. "League of Legends API Documentation." https://developer.riotgames.com/docs/lol (accessed 2021).

[24] T.-H. D. N. Zhengxing Chen, Yuyu Xu, Christopher Amato, Seth Cooper, Yizhou Sun, Magy Seif El-Nasr, "The Art of Drafting: A Team-Oriented Hero Recommendation System for Multiplayer Online Battle Arena Games," *RecSys '18: Proceedings of the 12th ACM Conference on Recommender Systems,* pp. 200 - 208, 2018, doi: https://doi.org/10.1145/3240323.3240345.

[25] J. G. Sebastian Bock, Matin Weiß, "An improvement of the convergence proof of the ADAM-Optimizer," presented at the OTH CLUSTERKONFERENZ 27 April 2018, 2018. [Online]. Available: https://arxiv.org/abs/1804.10587.

[26] PyTorch. "BCEWITHLOGITSLOSS." https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html (accessed April 3rd, 2021.