Deep Learning Blackjack Bot

Jack Snyder, Noah Bernot, Finn Jackson

April 2025

1 Introduction

Our initial idea for a Blackjack bot was to incorporate card counting, as it seemed like the simplest way to make the model play similarly to how a human would. While this was a logical approach to building the bot, it had significant limitations when playing a large number of games, especially since most casinos use multiple decks in a single game, making the bot's strategy less effective. StockFish, a pretrained chess bot that relied on a database of previous games played by humans was just recently dethroned by a reinforcement learning bot named AlphaZero. The latter bot was able to consistently beat a bot that relied on human knowledge while it learned completely on its own over millions of games. Our AI Blackjack Bot is designed to learn and master the game of Blackjack using deep learning techniques like AlphaZero did. Built with PyTorch, it leverages a neural network that analyzes game states, predicts optimal actions, and continuously improves its strategy through training. By simulating thousands of games, the bot refines its decision-making process, demonstrating how reinforcement learning and neural networks can be applied to classic card games in an intelligent, dynamic and "human-like" way.

2 Process

Our goal was to create a bot that played Blackjack at a success rate similar to that of a human. We define this vague goal as playing near but not perfectly at the probable success rate. A normal blackjack game when played perfectly by the book is around a 0.5% house edge, however, our model does not include any splitting or doubling which brings the house edge to around 2.5%. We decided to avoid adding splitting, doubling, and units because we believed that this would make our goal of training a model to moderate success unachievable in our time frame. First, we built a working Blackjack game that can be played manually with all of the rules. Admittedly, we used the deck of cards made by the GitHub repository cited at the bottom of this paper to avoid excessive and tedious work. This work can be found in our source code (game.py, blackjack.py, dealer.py, and deck.py). Once we had a game up and running, we initialized a generic PyTorch model and built support classes to allow it to play the game. This portion

of work can be found in our source code (model.py, bot.py, run_bot_game.py). This was an interesting step in our process because it allowed us to get a baseline for the success of our future training. Though we did not evaluate game results thoroughly, it was clear that the generic weights prompted the model to make poor decisions that would seldom lead to successful games, i.e., hitting on a 20, or standing on anything less than 10. From here, we built our training loop. This can be found in our source code (train_bot.py and evaluate_bot.py). The latter of the two files is unimportant to the results of the model, but it allowed for us to visualize our results thoroughly. Additional files include .pth files holding model data, .png files that contain graphical results from our evaluations, as well as beat_the_bot_backend.py and beat_the_bot_gui.py. These last files are not integral to our project, and were used for model evaluation as well as demos for our presentation.

Model Structure and Objectives The model is built using PyTorch and implements a Q-learning approach for playing blackjack. It takes 3 key inputs: the player's current score, the dealer's visible card score, and whether the player has a usable ace (an ace that can be counted as 11 without busting). The neural network processes these inputs through multiple layers ($3 \rightarrow 128 \rightarrow 64 \rightarrow 2$) to output two Q-values representing the expected future rewards for hitting or standing. The model's objective is to maximize these Q-values through reinforcement learning, where the highest Q-value determines the bot's action (hit or stand). The model adheres to standard blackjack rules where the goal is to beat the dealer's hand without exceeding 21, and aces can be flexibly counted as 1 or 11. The bot makes binary decisions (hit/stand) based purely on the numerical state of the game, without considering advanced strategies like splitting or doubling down.

Algorithm Analysis

We used two main algorithms in our project, along with some sub-algorithms. The core of our learning is **Q-learning**, which is a reinforcement learning algorithm that estimates future reward and adjusts weights based on success. Since we are using deep learning instead of building a Q-table, we do not update Q-values using the Bellman equation, instead, we predict Q-values and update our model weights based on the returned reinforcement weights and our loss values. Because of this, there is no probability theory behind our code, at least not explicitly. This also means that we do not have a true Q-learning algorithm, but we have some smaller algorithms that allow us to optimize our Q-value estimates, leading to success in testing. A relatively simple yet theoretically complicated algorithm we used in our training is called the **Epsilon-Greedy Algorithm**, which allows the model to stray away from its current estimations and explore other possibilities. The way this is implemented is by first defining an epsilon value early in our training loop: in our case, we defined epsilon as the maximum of 0.2 and 1 - epoch / 10,000, which means the value starts

closer to 1, but eventually evens out to 0.2. These values are arbitrary, but it is important to base them on the current epoch to allow the model to settle into its predictions later in the loop. These values are only used in a conditional that decides whether to use the estimated Q-value or a random one, with the random one being helpful for avoiding overfitting and allowing exploration of the vector space. This implementation can be found in train_bot.py, but it essentially allows our model to explore for the early stages of training, and eventually settle into its predictions. We believe this algorithm is important in our Blackjack model because it is for the model to get caught up in results early on, given that one bad hand can return negative reinforcement weights even if the model makes a good decision, and overall, this may have played a large role in our model's success. Later, we implement a slightly less rigorous version of the Bellman equation, adding gamma to our target value as well as the reward weight calculated based on game results. This implementation leads to questions about whether our model is considered supervised or unsupervised. The answer is neither, as reinforcement learning is based on reward values; it does not rely on correct answers to check validity, which is like unsupervised learning, but it also uses a target value to compute loss, which is like supervised learning. The important distinction here is that the target values are computed based on model results and rewards, meaning they aren't probabilistically accurate, but ideally, this leads to adjustment of the weights that eventually allows for successful results. Now the most uniform and important part of our project is the **Backpropagation Algorithm - Gradient Descent (ADAM Optimizer)**, which is where the model utilizes the loss function, which in this case is Mean Squared Error, to improve weights as the training continues. This algorithm first cancels any previous gradient calculations by calling torch.nograd, allowing new gradients to be calculated based on the most recent epoch. Once these new gradients are calculated based on loss, the algorithm then backpropagates through the neural network to determine what parameters caused the most error. Then the optimizer.step line allows the algorithm to update model weights based on the gradient, the problem parameters, and the learning rate, allowing for the model to have more accurate weights for future epochs. Much of this process is implemented through PyTorch, specifically through the ADAM optimizer method. As for why we chose ADAM instead of normal stochastic gradient descent or any other common optimizers, it is adaptive in its gradient calculations, allowing for circumstantial updates instead of uniform ones, which is extremely important given the variability in Blackjack. Additionally, it works well with simple learning rates, i.e., powers of 10, which makes it much easier to fine-tune later in the process. Overall, all of these algorithms were defined either through a library (ADAM), or probabilistically (Epsilon-Greedy, Q-Values, etc.), which is vastly different than what we have become accustomed to in our coursework (binary search, Dijkstra's, etc.), which are more strictly defined and consistent. This was a learning experience, and it is fascinating to see how we can achieve successful results without defining q-values explicitly.

As for the Neural Network Structure, we had a fair amount of options, but

most of what we ended up with were commonly used choices or guesses, which is standard given the probabilistic complexity of a model this size. We chose to use an input layer with three inputs: user's hand, dealer's showing card, and usable ace (a Boolean that is true when you have an ace that counts as 11 without busting and false otherwise), we believed this scope made sense because it provides enough information to make a reasonable decision based on the current game state. Additionally, we chose to use linear layers because it is standard, and we didn't believe we had constraints that would require a different layer structure. Then we added two hidden layers, one with 128 neurons and one with 64 neurons. This was the guess we were previously referring to, though a model with one hidden layer may have been satisfactory, we figured that adding another would resolve unforeseen problems regarding oversimplification of the problem space. The number of nodes was also a guess, but it is standard to have more neurons in the earlier hidden layers than the later ones, so we chose a number for the first hidden layer (128) and halved it for the second layer (64). Next, we defined the output layer to have two values (one Q-value for hitting, one for standing), which allows us to use a maximum function to choose these values in game. Lastly, we had to choose an activation function. Our obvious choice was ReLU because it tends to be effective for linear, feed-forward networks, and it suited us well. One problem that we considered was the fact that ReLU changes negative values to 0, therefore killing negative neurons, stopping them from contributing to future network updates. This seems particularly pertinent because Blackjack is so variable, meaning that we thought any extra neurons could help even out our results, but this was not the case. We tried to use LeakyReLU instead, but this yielded results notably worse than ReLU (as shown in the results section), leaving us with the ever-reliable ReLU as our activation function of choice.

Time Complexity is something we did not struggle with much, simply because the algorithms imported from PyTorch are fully optimized already, but we will include the results anyway. One forward pass through the model is O(n*m), where n is the input size (3) and m is the largest layer size (128). Decision making is O(1) because the Q-values are determined in the model; in-game decisions only require the decision of which value is the maximum out of the two (argmax). Thus, the total time complexity per game is O(k), where k is the number of decisions per game.

The **Space Complexity** of this model is daunting, which is why we chose to use PyTorch instead of defining our neural network from scratch, as well as why we chose to ignore splitting and doubling to focus on hitting and standing. There are 8,832 model parameters, 3*128 from the input to the first hidden layer, 64*128 from the first hidden layer to the second, and 64*2 from the second hidden layer to the output layer.

The **Limitations** of our model are clear-cut, but it would be a simple process to account for them using our process. First, as mentioned before, We are

making binary decisions only (hit or stand); otherwise, we would use splitting and doubling which would complicate the problem space drastically, more output nodes would need to be added as well as input nodes representing units for betting, then in-game decisions would be more complex, as well as the complications with training and success. Additionally, we don't use card counting, but that would be a simpler fix with a better-defined deck and a HashMap representing visited cards. Additionally, there are some training limitations due to the randomness of Blackjack. We ran our initial model for 50,000 epochs (1 game each), and it yielded similar results to running it through another 150,000 epochs, meaning the training scope is optimal, but there is still a margin of a few percentage points of wins that keep our model from playing perfectly. The variance in our evaluations shows that the precise success is largely reliant on the quality of hands given over 1000 games, yet we are still able to make general conclusions about its success.

We were approached with a question during our presentation about how practical our model is. Admittedly, it is not practical at all; it doesn't play with a specified number of decks, it doesn't count cards, it doesn't compute strict probabilities, and it doesn't even play with all of the rules of Blackjack, meaning it really has no use other than learning and research deep-learning. With this, we will include some **alternatives** that are more practical than our project. The obvious choice, which has been around for a long time, is using a Rule-Based System, which defines a strategy table or probability table as a tree, and uses modern search algorithms to make decisions based on the game state and the given results. Though practical, this is boring, but it requires no training, and it is easy to interpret. Another alternative is to train a model to learn policy, which would be a more technically interesting approach than the previous one. This essentially means using a model to develop a table to make decisions on later, which would be similarly effective to our model. Another approach, which changes the game structure but not necessarily the model, is a CNN, which would process image information to make decisions. Given all of this, we do believe that a deep learning model has its benefits and wanted to show these benefits through our project.

3 Human Parallels

The mechanics of Blackjack are intuitive for most people; simply get as close to 21 as possible. For a neural network, this is not as simple, which is why reinforcement learning is so important to build a model that can play the game well. By looking at the details of our reinforcement learning, the human learning parallels become clear. The model essentially guesses a Q-value to play, leading to a hit or a stand, which is similar to how a human would choose hit or stand immediately. Then, the training loop takes the results of the game, defines reinforcement values, decided by the programmer, and then lets the gradient descent algorithm use these reinforcement values to make a target value, and

compute the difference between the chosen Q-value and the target value, or loss, which the model can then hold onto as the problem with its previous action. This is similar to how a human would see the result of their guess and troubleshoot what they did wrong. They would decide if they should have stood instead or if they should have hit again, for instance. The model then takes this loss value and uses it to adjust its weights for future decisions. Though this weight adjustment is not necessarily a perfect solution, it accounts for similar hands and hopefully leads to better results later on. This is similar to how a human would hypothesize a solution for the next hand and apply it to future hands. Both the human and the model only gained a small amount of information from this one hand, but they will use it to perfect their weights (for the model), or their policy (for the human), and eventually, they should be winning hands often. From a distant view, it is obvious that reinforcement learning is similar to human behavioral learning, but it is not so obvious that even the events occurring in the gradient descent and weight adjustment are similar to human troubleshooting and solving. Even so, they are quite similar, which is why it makes sense that after 50,000 training epochs, a model can reach a win rate similar to that of a human who is learning based on playing Blackjack, as opposed to reading probabilities and learning policy.

4 Results

Through our continued trial and error with our model structure, we were able to choose a model that performs the best under the given constraints. Our initial model had the following specifications as referenced previously; Adam gradient descent optimizer, ReLU activation function, two hidden layers, a learning rate of 0.001, and a gamma value of 0.99. Though this structure was satisfactory, by plotting our loss functions by averaging the loss every 1000 epochs, we were able to draw informed conclusions about which structures performed best.

Our Initial model had a win rate ranging from 38% to 42%, which is an arguably human win percentage, though not perfect.

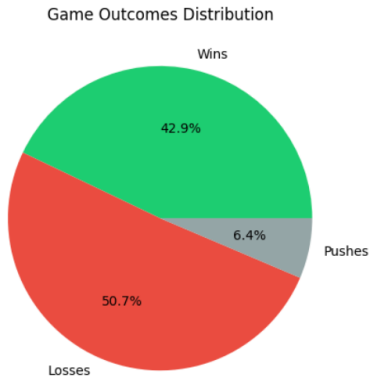

Figure 1: Pie Chart dividing results over 1000 games

Additionally, the loss by epoch graph included below shows the gradual descent of the loss values

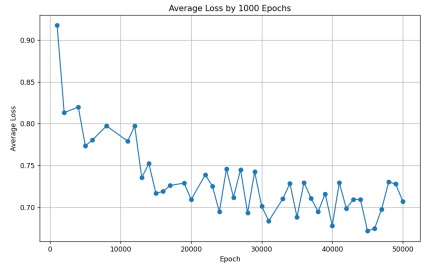

Figure 2: Loss Progression through training for original model specifications

Given this loss graph, it is challenging to draw immediate conclusions about potential changes to the model. Usually, the learning rate can be changed to manage the erratic descent behavior, but given the probabilistic nature of Blackjack, such a loss graph is not achievable. This is because false positives and negatives occur often during the training, all of which can be associated with the luck of the hand being dealt. Given this constraint, we decided to approach our model optimization with trial and error.
The following results come from a model that is identical to our initial one, but the learning rate has been lowered by a power of 10, making it 0.0001. This approach gave us more consistent success in our model, winning from 42% to 46% of games, a small but clear improvement on our initial model.

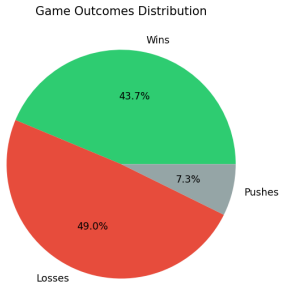

Figure 3: Game results from lower learning rate

These results are not obvious from the loss graph, as the erratic behavior of the loss descent is similar to that of the previous model, but their are slight differences.

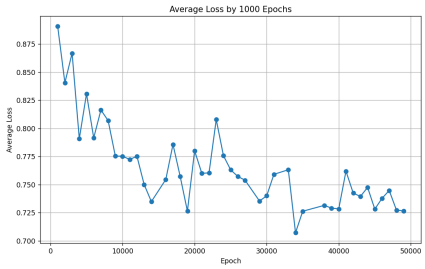

Figure 4: Loss vs Epochs for model with learning rate of 0.0001

We made other changes to our model, but no results were better than shown above. Regardless, I will describe each of our other tests below.
Another obvious choice was to see if an even lower learning rate (0.00001) improved the model's performance. It did not, but it led to a similar result to our initial model.

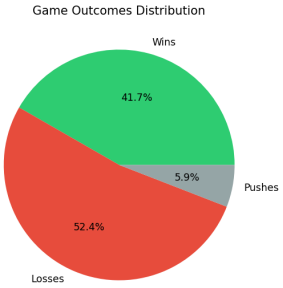

Figure 5: Very low learning rate game outcomes

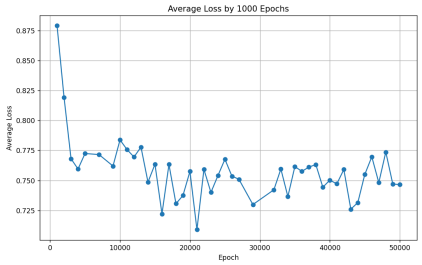

Figure 6: Very low learning rate loss vs epoch

Additionally, we attempted to change the gamma value, which we initially chose to be 0.99 (values future outcomes 99% as much as present outcomes), and changed to 0.5. This resulted in game outcomes similar to those from our initial model.

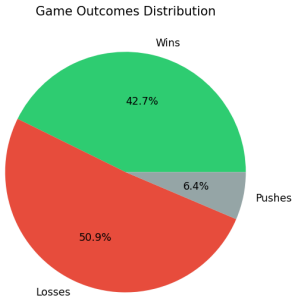

Figure 7: 0.5 Gamma game outcomes

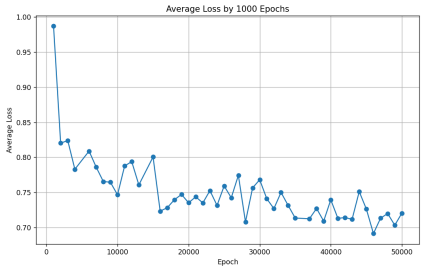

Figure 8: 0.5 Gamma loss vs epoch

Our last trial was with a new activation function, similar to ReLU, we used LeakyReLU, which instead of converting negative neurons to 0, converts them to a small number such as 0.01, allowing them to contribute to future weight adjustments and potentially making the training more accurate. Unfortunately, this did not yield better results, and instead was very similar to our initial model.

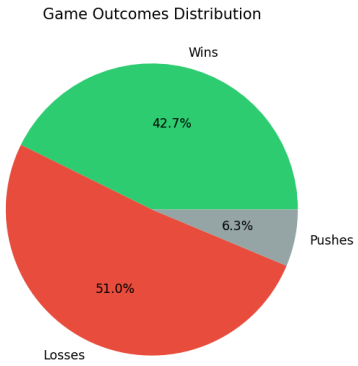

Figure 9: Leaky ReLU game outcomes

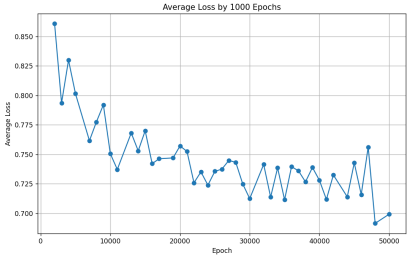

Figure 10: Leaky ReLU loss vs epochs

Though our method of randomly trying new model structures is not rigorous, it allowed us to test other options and make an educated decision regarding the most accurate model. We are particularly satisfied with this method because we were able to improve on our initial model even with the small scope of our testing. Given more time, we could continue this testing by testing model structures more systematically and averaging our game outcomes through an arbitrarily large number of evaluations, but for now, our bot plays Blackjack with great success.

As stated previously, a normal blackjack game when played perfectly has around a 2.5% house edge without splitting and doubling involved (which slightly decreases the edge for the house, *see link at bottom of paper). Our model was able to perform at a level very close to the house edge in some instances, while a little worse in others. Blackjack, being very random, would have a reasonable win percentage anywhere between 30-50% while not including pushes.

A 40% win rate in blackjack has several important implications about the problem space: While it falls short of perfect basic strategy (49.5% win rate), it significantly outperforms random play (42.42% house advantage), indicating successful learning. This win rate makes sense given the inherent house edge (0.5-1% with perfect play) and our model's limitations. The bot lacks access to advanced strategies like doubling down (1.5% EV loss) and splitting pairs (0.5% EV loss), plus the dealer's structural advantages (acting last, winning when both bust). To push beyond 40%, we'd need to implement full basic strategy options, add card counting capabilities, enhance the state representation beyond the current 3 inputs, and of course, get lucky hands. In the context of standard blackjack probabilities, our model's performance suggests it has learned reasonable strategy while leaving room for architectural improvements. We are very happy with our results.

5 Sources and Misc.