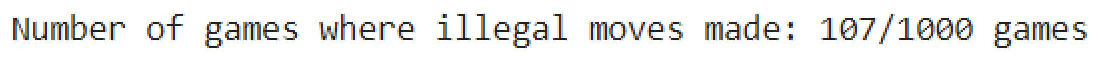
Outcome analysis of our own Policy Gradient RL model

This is our analysis for a policy gradient reinforcement learning model using numpy, which we designed. First, we designed an uno environment, and created a basic agent model. For the basic agent, we chose to represent the agent's hand as an array, and select the first available and legal move to play if there is one via a linear search through the hand. Then, the agent would choose to draw if it had no legal moves.

Our goal was to initially have a reinforcement learning based agent that didn't even know what legal moves were, and to eventually learn which moves were legal and illegal by incentivizing legal moves (positive rewards) and punishing illegal moves (negative rewards). Additionally, since drawing cards is detrimental to winning, we punished the model slightly for choosing this option (small negative reward).

For the implementation of the policy gradient, we tried various methods, but were unsuccessful at implementing a true policy gradient learning strategy. We initialized a simple neural network as the policy gradient, with one hidden layer. The input was 124 data points, which were one hot encoded to represent the current game state and the agent's hand, and the hidden layer was 80 perceptrons. We eventually settled on just using ReLU as the first activation method. Our output layer was 53 perceptrons, which were representative of the 53 different possible cards to play (52 cards and 1 draw card move). The final activation layer was softmax, which was used to transform the output layer into probabilities. We also initialized the weights and biases to zero, since we wanted the neural network to begin with no biases whatsoever in learning. The selection choice for which move to make was simply the highest probability, which in hindsight limited the creativity of the agent. Although we tried implementing a sampling strategy based on the probabilities calculated by the agent, it didn't work out as intended so we will present the results of our model using a deterministic move choice strategy.

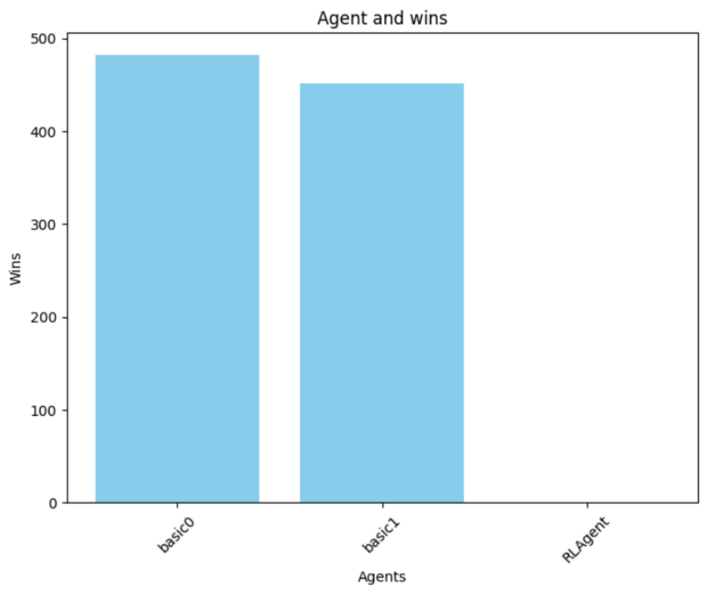
Initially, the reinforcement learning agent (RLAgent) performed terribly as expected. We made it play 1000 games to train, and our agent started by making illegal moves for the first ~100-150 games:



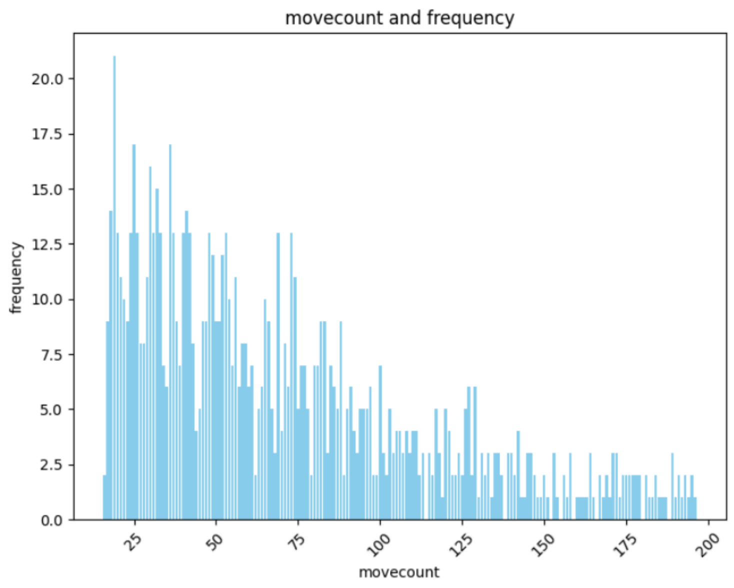
```
Number of games where illegal moves made: 107/1000 games
```

Then, we ran it on another sample set of 1000 games, and observed the results. While the agent no longer made any illegal moves, we found that given our chosen reward and loss function, the neural network for the probability output converged to incentivize selecting the draw card option. Additionally, the deterministic quality of the model caused it to always select this option, further exacerbating the problem.

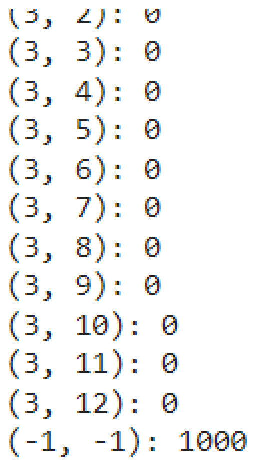
This caused the agent to never win, and instead continue drawing cards on every move (even if it was not the ideal move):



Compared to the somewhat uniform distribution from our previous milestone where we compared three basic agent strategies, it's pretty obvious that the model doesn't win often (at all). We also kept track of the number of moves each game would last, and found that the average move count per game was substantially higher than the distribution of the original basic agents.

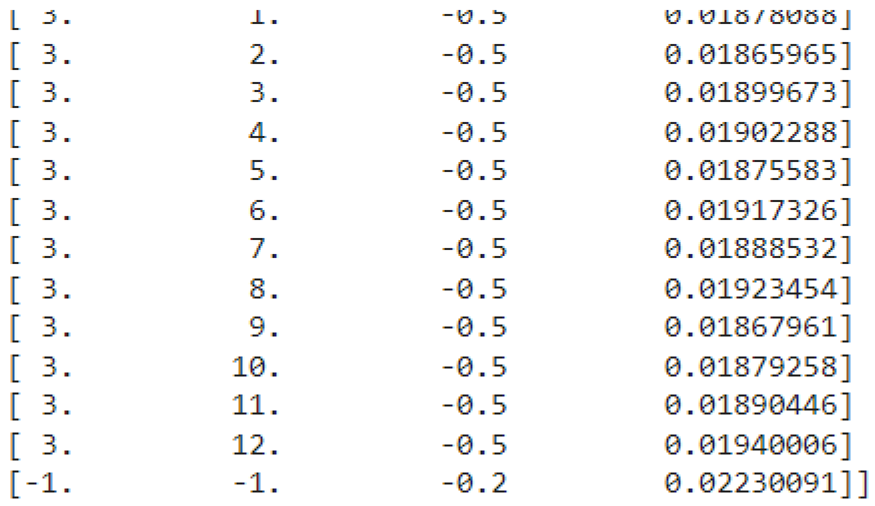


Furthermore, given any game state, we can further show that the model converges to select a slightly negative move. For the dictionary, each tuple is a card move, and (-1, -1) corresponds to the draw card move. It selects it 1000 out of 1000 times:



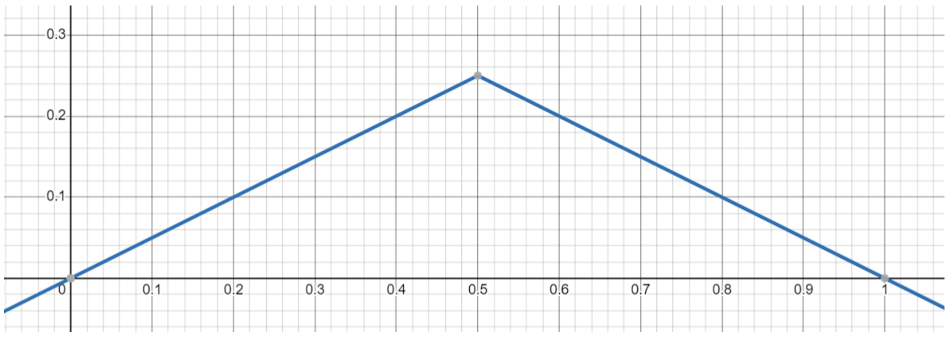
```
(3, 2): 0
(3, 3): 0
(3, 4): 0
(3, 5): 0
(3, 6): 0
(3, 7): 0
(3, 8): 0
(3, 9): 0
(3, 10): 0
(3, 11): 0
(3, 12): 0
(-1, -1): 1000
```

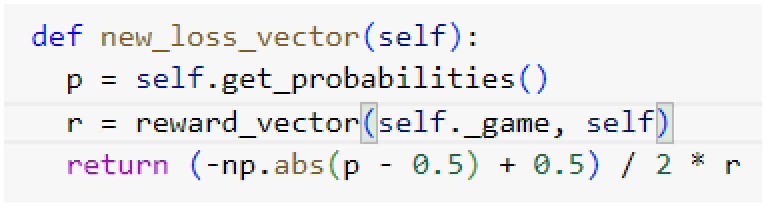
Additionally, taking a look at the softmax output for the agent's neural network given some random game state shows us why the agent selects this move. Since there are 53 possible moves, and the agent uses a deterministic policy to select it's next move, the draw card option has a probability slightly higher than 2%, which is greater than any of the other moves, which generally are learned to be just less than 2%. In the below image, the first two columns are the corresponding move, the third column is the reward for making that move, and the rightmost column is the corresponding probability distribution of making said move. Due to this issue, we found that our model was not designed to balance exploration with its exploitative strategy:



```
[ 3.          1.          -0.5         0.01878088]
[ 3.          2.          -0.5         0.01865965]
[ 3.          3.          -0.5         0.01899673]
[ 3.          4.          -0.5         0.01902288]
[ 3.          5.          -0.5         0.01875583]
[ 3.          6.          -0.5         0.01917326]
[ 3.          7.          -0.5         0.01888532]
[ 3.          8.          -0.5         0.01923454]
[ 3.          9.          -0.5         0.01867961]
[ 3.         10.          -0.5         0.01879258]
[ 3.         11.          -0.5         0.01890446]
[ 3.         12.          -0.5         0.01940006]
[-1.         -1.          -0.2         0.02230091]]
```

The biggest issue was likely our loss function. We had some trouble understanding how to calculate the policy gradient itself, and used our own function to decide how to modify the probability distributions themselves, which didn’t account for the actual move selected by the agent. We designed this as the function to base the policy gradient on. The idea was that given some probability p where 0 < p < 1, moves would increase or decrease the probability by some amount that would allow the new amount to approach 0 or 1 but never reach it. The main problem with this is that at a very low probability, selecting a good move actually didn’t increase the probability significantly to change the gradient. For this reason, we realized too late that this function would cause our agent to require extensive training to overcome the initially uniform probability distribution where each move’s probability of being selected was around 2%.





```
def new_loss_vector(self):
  p = self.get_probabilities()
  r = reward_vector(self._game, self)
  return (-np.abs(p - 0.5) + 0.5) / 2 * r
```

This was the gradient function itself, which was just multiplied by the reward (which was guaranteed to be between -1 and 1. While we did attempt to implement a loss vector and policy gradient method using log probability, we were unable to correctly implement this strategy.

Ultimately, we were at least able to design an agent that was capable of learning and assigning higher probabilities to legal moves rather than illegal moves. Unfortunately, our design did not account for a number of important factors that significantly influenced the outcome: a deterministic move selection policy, a gradient algorithm that didn’t properly affect probabilities as intended, and a lack of systematically tuning neural network hyperparameters. Given the loss vector we used, we think that given enough training time and a non-deterministic sampling method for choosing moves, the agent would be able to eventually learn which moves are better, although it would take an inordinate amount of time to do it.