CS 440 Assignment 4 Report

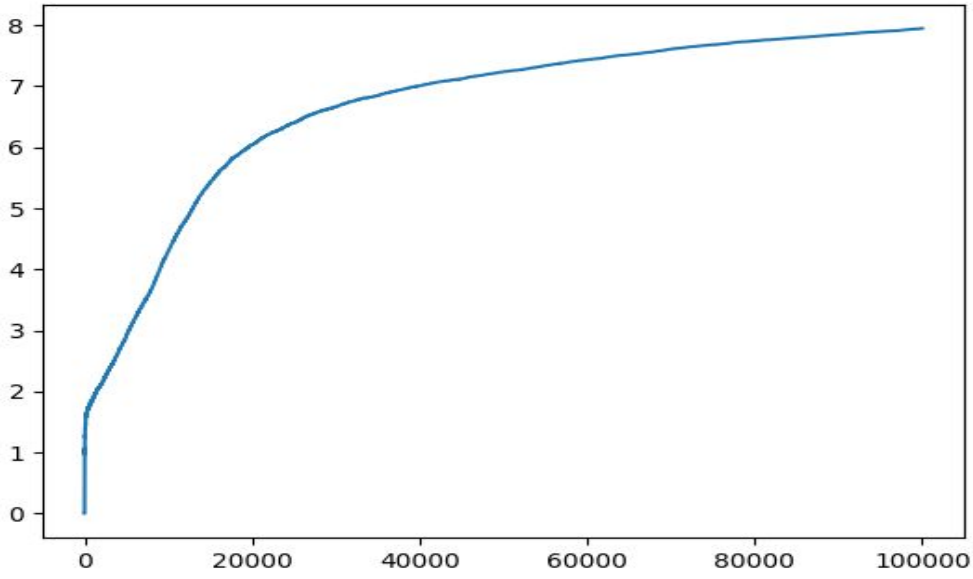
Edbert Linardi, YoungBin Jo, Xiao Tang (3 credit)

Part 1.1

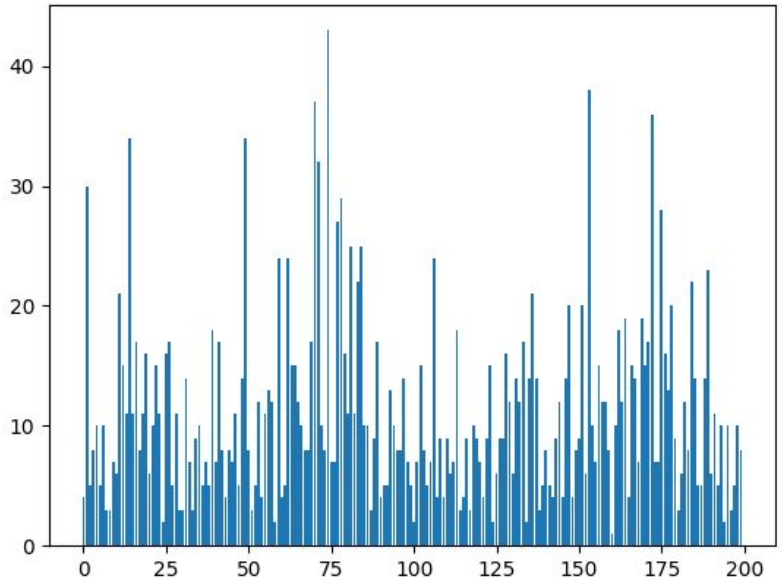
In both TD-Learning and SARSA, we trained on 100K epochs.

TD-Learning

200 epoch testing Hit Average: 11.155



Mean Episode Rewards vs. Episodes in 100K training epochs



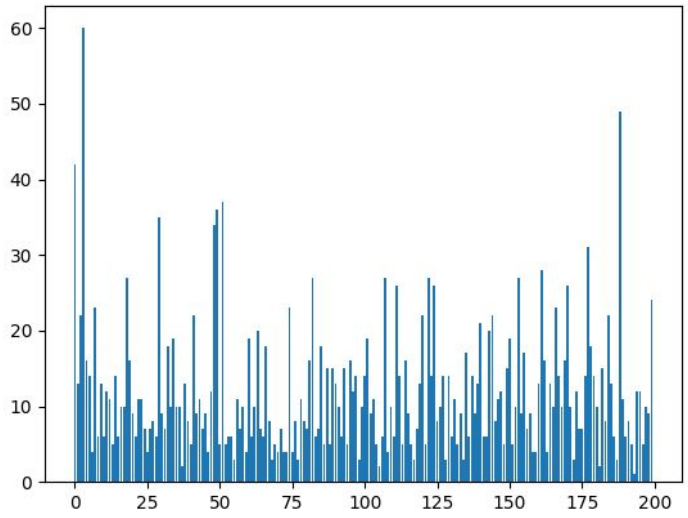
Hits vs Index in testing

In our implementation, we use $\varepsilon - greedy$ approach for our exploration function. We want to explore other actions than the ones with the highest value in Q-table. The ε value used is 0.08, meaning it has 8/100 probability to explore a new action. We don't want to set the ε value to be bigger, to prevent the AI from exploring too much, which could decrease the accuracy because it doesn't choose the best actions (the ones with the biggest value in Q-table).

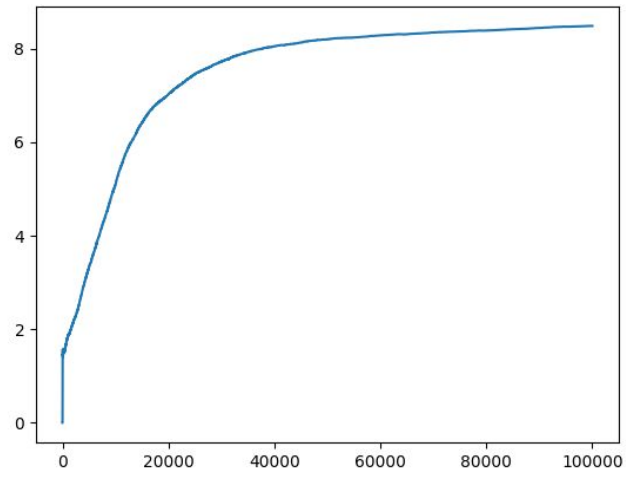
For our learning rate, we use $C = 4$, since the learning rate $\alpha(N) = C/(C + N)$ should decay as the number of times we see the state-action increases, which means that we need to start at a slightly bigger number, to prevent it decays too early at the beginning. With smaller values, such as ~ 0.9, the average hit in testing was only around 8, which we believe it is caused by the learning rate decaying too early.

We choose our γ value, the discount factor, to be 0.8. It is the best value that we have tried, because the reward from the previous hit has been mostly discounted. Therefore, the reward from the previous hit has only little effect on the next hit. With smaller value, such as 0.2, the result was bad, because previous hit's reward was not discounted a lot.

SARSA
200 epoch testing Hit Average: 11.95



Hits vs Index in 200 testing epochs



Mean Episode Rewards vs. Episodes in 100K training epochs

For SARSA, we also use $\varepsilon - greedy$ approach for our exploration function. However, the parameters are different. The C, γ and ε values are bigger than the ones in the TD-learning. For SARSA, we need to explore more frequently, and learn more quickly, which we believe it is caused by the update rule in SARSA considers the next step taken, which was different in TD-learning, which the action considered in the update rule is not always the one taken in the next step.
The C value used is 5, while the ε value used is 0.1 and γ value is 0.9..

Part 2.1

- What is the benefit of using a deep network policy instead of a Q-table (from part 1)?

Q-table deals with limited state spaces while deep network policy deals with continuous state spaces. Additionally, if the game reaches a new state which haven't been encountered before, Q-table agent will take an action based on a randomly initialized value. On the other hand, when making decision using a deep network trained with expert policy, the agent will take an action based on the function which it interpolated from the expert policy instead of at random, resulting in a more logical decision.

- Implement the forward and backwards functions of a neural network and give a brief explanation of implementation and architecture (number of layers and number of units per layer).

Affine-Forward:

This function computes affine transformation Z = AW + b where W is 2d array of layer weight and b is 1d array of bias. It is implemented by numpy matrix multiplication and addition.

Affine-Backward:

This function computes the gradients of A (2d array of data), W (2d array of layer weight) and b (1d array of bias). dA is computed by dZ@W.T, where @ is matrix multiplication and T is the transpose function. Similarly, dW = A.T@dZ, and db is the column sum of dZ.

ReLU-Forward:

This function creates an 2d array, A with same size of batch Z. If element value of batch Z is negative, value of element is zero in array A. Otherwise, copy the value of batch Z to A.

ReLU-Backward:

This function computes gradient of Z. If Zij is negative, dZij is zero. Otherwise, dZij is same as dAij

Cross Entropy:

This function is implemented using logsumexp function from SciPy, and a softmax function in order to be numerically stable.

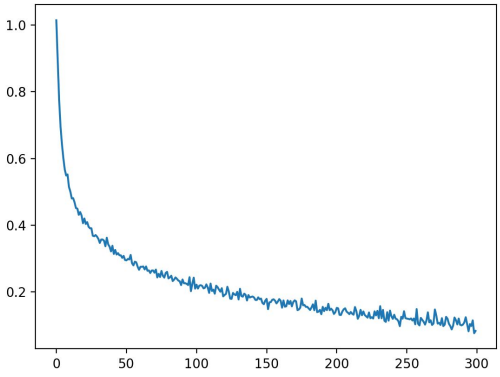
We implemented a **4-layer neural net, with 256 hidden nodes** at each level, following the pseudocode in the MP instruction. We let our neural net take an additional current epoch parameter so that we can construct a decaying learning rate. The gradient descent procedure is identical to the pseudocode in the instruction, and we used a **batch size of 128**.

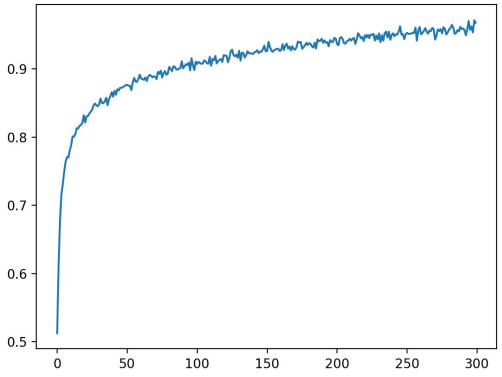
- Train your neural network using minibatch gradient descent. Report the confusion matrix and misclassification error. You should be able to get an accuracy of at least 85% and probably 95% if you train long enough.

We experimented with different parameters to train the neural network. With a **constant learning rate of 0.1**, and weight being randomly initialized in range [-0.1, 0.1], we achieved an **accuracy of 92%** on the training data after 300 epochs. With **a decaying learning rate of 0.25 * (0.99^epoch)**, and weight being initialized in range [-0.5, 0.5], we achieved **99.95% accuracy** on the training data after 300 epochs. The confusion matrix for decaying learning rate training is at the end of the report.

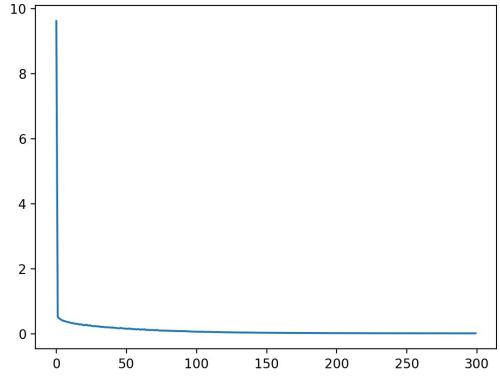
- Plot loss and accuracy as a function of the number of training epochs. **You do not need to do a train-validation split on the data**, but in practice this would be helpful.

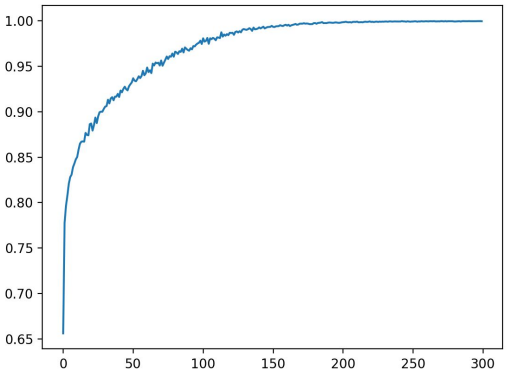
Loss vs. Epoch and Accuracy vs. Epoch for const. learning rate





Loss vs. Epoch and Accuracy vs. Epoch for decaying learning rate





Since our weight initialization range for the decaying learning rate training is very wide, the loss value of our first epoch is very big (~9.4). After the second epoch, loss value already decreased to 1.4. This caused the graph to be less useful in representing the data.

Confusion matrix of decaying learning rate training result:

```
1.0000000000000e+00 0.0000000000000e+00 0.0000000000000e+00
1.2804097311139e-03 9.9871959026888e-01 0.0000000000000e+00
2.3132084200786e-04 2.3132084200786e-04 9.9953735831598e-01
```

The row number is the true label of the data, and the column number is the label our neural network classified.

When running our neural net agent in *Pong*, it achieved an average of **15.65 bounces** across 200 games.