CS 4700 - Foundations of Artificial Intelligence **Homework 4**
Fall 2019 **Due Wednesday, November 20th (2:40pm)**

1. In this question, we will be taking a look at *Bernoulli Bandits*. A Bernoulli bandit has n arms where the i'th arm produces a reward of 1 with probability μ_i and a reward of 0 with probability $1 - \mu_i$. For this question suppose we have 3 arms: M_1 with $\mu_1 = 0.2$, M_2 with $\mu_2 = 0.4$, and M_3 with $\mu_3 = 0.6$.

   (a) A very simple optimal policy to the multi armed bandits problem is to always pull the arm with the largest *Gittens Index*. The Gittens Index for an arm M is defined as:

   $$\lambda(M) = \max_{T>0} \frac{\mathbb{E}\left(\sum_{t=0}^{T-1} \gamma^t R_t\right)}{\mathbb{E}\left(\sum_{t=0}^{T-1} \gamma^t\right)}$$

   where R_t is the random variable denoting the reward at time t and γ is our usual discount factor. What is the Gittens Index for each of the arms M_1, M_2, M_3? What is the optimum policy?

   (b) For more complex problems, calculating the Gittens Index may prove to be difficult. In class we've seen the Upper Confidence Bound (UCB) heuristic, which is an approximately optimal policy with provable bounds on the expected regret. Recall that the UCB heuristic pulls the arm with the largest $UCB(M_i)$:

   $$UCB(M_i) = \hat{R}_i + \frac{g(N)}{\sqrt{N_i}}$$

   where $\hat{R}_i$ is the average reward for M_i so far, N is the total number of pulls made so far, and N_i is the total number of pulls of M_i so far. For this question, let us use

   $$g(N) = \sqrt{2 \ln N}$$

   Suppose the three arms, when pulled, will produce the following sequence of rewards:

M_1	0_3	1_4	0_5	0	$0 \cdots$
M_2	1_2	0	0	1	$0 \cdots$
M_3	1_1	1	0	1	$0 \cdots$

   For example, if a gambler chooses to pull arms M_3, M_2, M_1, M_1, M_1, we'd get the sequence $1_1, 1_2, 0_3, 1_4, 0_5$. (The subscripts are not part of the reward; they are only there to clarify the explanation.)
   Simulate the UCB algorithm on this sequence of rewards for 5 timesteps (every time an arm is pulled counts as a timestep; this includes the initialization steps). If there is a tie between two arms's UCB heuristics, choose the arm with the smaller index e.g. choose M_1 over M_2 if $UCB(M_1) = UCB(M_2)$.

   (c) In this next question, we'll get some intuition on why this heuristic is called the Upper Confidence Bound, and why we chose $g(N)$ the way we did.

   First, we introduce *Hoeffding's Inequality* which states that for independent and identically distributed (iid) random variables $R_1 \ldots R_n$ that are either zero or one we have:

   $$Pr(\hat{R} + a > \mu) \geq 1 - e^{-2na^2}$$

   where $\hat{R} = \frac{1}{n} \sum_{i=1}^{n} R_i$ and μ is the expected value of the random variables. Notice that since they are iid all the random variables have the same expected value.

   Suppose we want to guarantee with high probability that the true expected payoff μ is less than $\hat{R} + a$. This makes $\hat{R} + a$ an upper bound. In other words, what is a value for a such that:

   $$Pr(\hat{R} + a > \mu) \geq 1 - N^{-4}$$

   where N is the number of rounds played in total? a can, and should depend on n and N

2. **AlphaZero and MCTS**

In 2016, AlphaGo defeated former Go World Chamption Lee Sedol in a historic game. Follow-up work simplified to the AlphaGo algorithm to result in AlphaGo Zero and AlphaZero[1] (basically the same as AlphaGo Zero, but removing Go-specific tweaks). Monte Carlo Tree Search is a key component of these algorithms. Because we're talking about games other than Go, we will talk about AlphaZero.

AlphaZero uses a neural network $f_\theta(s)$ to output a probability distribution over moves $\mathbf{p}$ as well as an expected win value v (1 for winning with certainty, -1 for losing with certainty) for each state s.

$$f_\theta(s) = (\mathbf{p}, v)$$

AlphaZero is trained in two steps:

- Step 1 is to use MCTS (where rollouts are sampled with action probability according to $\mathbf{p}$ and evaluation values according to v) to get new observed win/loss labels z ($z = 1$ for winning, 0 for a draw, and -1 for losing), as well as action probabilities π.
- Step 2 is to use gradient descent to change the values of θ in order to minimize the following loss function:

$$l(\theta) = (z - v)^2 - \pi^T \log \mathbf{p} + c\|\theta\|$$

  - The $\mathbf{p}$ and v come from $f_\theta(s) = (\mathbf{p}, v)$.
  - $(z - v)^2$ penalizes the difference between the true win/draw/loss label z and the expected win value v.
  - $-\pi^T \log \mathbf{p}$ is the cross-entropy of π and $\mathbf{p}$. This term is minimized when $\pi = \mathbf{p}$, and is large when the policy $\mathbf{p}$ assigns low probability to a move which is actually played in π.
  - $c\|\theta\|$ is a regularization term to penalize large values in the vector θ.

(a) Why does improving the accuracy of the prediction for the expected win value v improve the move selections made by AlphaZero during MCTS?

(b) Why does updating towards the move probabilities found by MCTS improve the move probability $\mathbf{p}$?

(c) Why not use MCTS for Tic-Tac-Toe?

(d) If the we're playing tic-tac-toe, and the neural network always outputs a uniform probability distribution over all remaining legal moves, what is the probability of MCTS sampling the following game when starting from an empty board?

X	O	X
O	X	O
O	X	O

3. **Perceptrons and Logistic Regression**

Suppose we have the following tiny dataset with two input features x_1 and x_2:

x_1	x_2	y
0	0	0
1	1	1
1	-1	1
-1	0	1

Your colleague is attempting to use a perceptron to predict y, given x_1 and x_2 as input. However, even after thousands of iterations, the weights don't converge, and it still does not classify all four points correctly.

(a) Briefly explain why this is happening. (You don't need to use equations - an informal explanation is fine. Drawing a picture of the dataset may help.)

(b) If we use logistic regression instead of a perceptron, would it able to classify every point correctly? Why or why not? (Again, no math required.)

(c) Draw a picture of a different dataset (in 2 dimensions) where neither the perceptron algorithm nor logistic regression will be able to classify every point correctly.

(d) Now, suppose we add a third feature x_3 to the dataset. For each data point, x_3 is calculated to be $x_1^2 + x_2^2$. If the perceptron is trained using the Perceptron Learning Rule on this new dataset (which includes x_1, x_2, x_3), is it guaranteed to classify every point in our dataset correctly? Recall that the perceptron predicts

$$h_{\bar{w}} = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Justify your claim.

- If the perceptron will classify every point correctly, explain why. Also, give a set of weights w_0, w_1, w_2, w_3 that would cause all points to be correctly classified (and box the weights).
- If not, explain why the perceptron might not classify every point in this dataset correctly.

```
w_strings = input('Enter 4 weights separated by spaces.\n').split()
if len(w_strings) != 4:
    print('There should be exactly 4 weights (w0, w1, w2, w3)'
          'separated by spaces. You passed in', len(w_strings))
    exit(1)
w = [float(i) for i in w_strings]

errors = 0
if w[0] + w[1]*0 + w[2]*0 + w[3]*0 >= 0:
    print('Misclassified 1st data point.')
    errors += 1
if w[0] + w[1]*1 + w[2]*1 + w[3]*2 < 0:
    print('Misclassified 2nd data point.')
    errors += 1
if w[0] + w[1]*1 + w[2]*-1 + w[3]*2 < 0:
    print('Misclassified 3rd data point.')
    errors += 1
if w[0] + w[1]*-1 + w[2]*0 + w[3]*1 < 0:
    print('Misclassified 4th data point.')
    errors += 1

if errors == 0:
    print('Correct!')
else:
    print('Incorrect.', errors, 'out of 4 data points misclassified.')
```

(e) In the previous part, we manually came up with a new feature from existing ones. Name a different algorithm that would be able to correctly classify all of these points without having to manually come up with new features.

(f) One criticism of the perceptron algorithm is that its predictions are over-confident: it always announces a prediction of 1 or 0, even when the example is very close to the boundary or if there is a lot of noise. If I wanted to use a different linear classifier that can output the **probability** that the example is +1, what method should I use?

Historical sidenote: When Cornell professor Frank Rosenblatt invented the Perceptron in 1957, the New York Times reported: "The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." This quickly turned out to be an over-optimistic statement, especially when it was proven in 1969 that perceptrons were unable to model even fairly simple functions such as XOR, which caused funding to dry up. However, in the 1980s, the ideas behind the perceptron ended up being crucial as a building block for more complex neural networks, which were in fact able to model non-linear functions.

References

[1] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. URL: https://arxiv.org/abs/1712.01815.