Project 1

Due Sunday, September 17 by 11:59pm

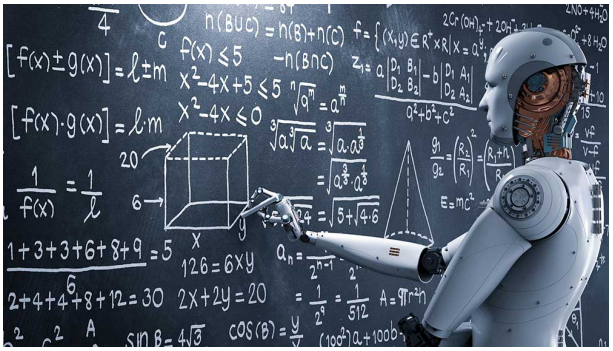
INSTRUCTIONS

For this first project you will create a single file that contains several function definitions. Your file, in other words, should be a module which theoretically could be imported into other Python files in order to access those functions.

Below you will find a project description including a description of the functions your code should define. **Be prepared to re-read this project multiple times and ask questions if needed.** If you get stuck on one function definition please do your best and get as far as you can. Partial code will receive partial credit. Any code that doesn't work or isn't finished should preferably be commented out so that the rest of your code will run, and please do include comments explaining your attempt and the challenges you encountered.

INTRODUCTION

Artificial intelligence (AI) has advanced immensely in recent years. The basic idea of using a neural network as part of a machine learning algorithm to process and answer text-based questions has been around for a while. However, these algorithms have generally required a lot of "training", where questions and answers are fed into the algorithm so the neural network can learn what answers are likely to be correct. Needless to say, this training needs to occur before the AI is put to the test in order for the AI to perform well. Depending on the quality of the training, the AI may need to be trained a long time before it gives reasonable answers consistently. What's made AI so much more powerful in recent years is the quantity (and in some cases the quality) of training that computer scientists have been able to put into it. By scouring the internet and making use of massive text databases, algorithms like ChatGPT are able to learn how to replicate human-like text and can also learn how to give the correct answer to test questions on various subjects.



Obviously, creating an AI would be beyond the scope of this course. However, we can try to create a simulation to model the connection between training an AI and its performance on a test. In this project, you will create a function that allows us to see the effect that the quality and quantity of training has on a hypothetical AI.

FUN FACT

In the language of mathematical probability, a Markov chain is a stochastic (i.e., random) process in which the next state of the system depends on the previous state. This important concept has all sorts of applications. It forms the basis of modern computerized random number generation (the so-called

Markov chain Monte Carlo algorithm), as well as Google's famous PageRank algorithm for ranking the relevance of webpages. In this project, our AI model will also make use of this concept insofar as the training effect will be randomized and will change depending on previous training outcomes.

PROJECT DESCRIPTION:

Ok, so we're going to create a model for training and then testing an AI. Let's suppose there are 20 questions we want the AI to be able to answer. Since the AI isn't perfect we will assume there is a probability of getting each question right or wrong, and as the AI undergoes training these probabilites may change for each question separately. So, when you're coding you will need to keep track of these 20 different probabilites. Let me be clear: in this simplified situation it doesn't actually matter exactly how the AI works or what the specific questions or answers are. We only care about the 20 probabilities that the AI will correctly answer each question. I will denote these probabilities as

$$P_1, P_2, P_3, ..., P_{20}$$

where P_1 is the probability of answering the first question correctly, P_2 is the probability of answering the second question correctly, etc.

To start with, let's assume that all 20 probabilites are 50%. In other words, before any training the AI is 50% likely to answer each of the 20 questions correctly. Training will change these probabilities, hopefully for the better. Training a neural network is imperfect, but it gets better with quantity and quality. To model this, let's allow for two variables the user can control: `t_num` and `t_quality`. The first variable `t_num` will be the number of steps of training before we stop (we're assuming that training occurs in steps). The second variable `t_quality` should be a number between 0 and 1 and will represent the quality of the training.

Each step of training should work like this. First, randomly pick a specific question and assume n is the question number we picked. Then, generate a random float from 0 to 1. If this random float is **less** than `t_quality` then the training was successful and the probability the AI can answer the n^{th} question correctly goes up by some amount x (in other words, add x to the value of P_n). Otherwise the training was a failure and the probability goes down by x for that question. By default $x = 1$.

If we always use $x = 1$ then this is a pretty slow process. To speed things up, let's assume the AI adapts quickly when trained successfully (or unsuccessfully) multiple times in a row. For a specific

question, your code should DOUBLE the value of x every time that question is trained successfully (or unsuccessfully) two or more times in a row, and only reset x back to 1% when a training success is followed by a failure or vice versa. Note that this will result in values that are powers of 2. Two successes in a row adds $x = 2$ instead of $x = 1$, three successes in a row adds $x = 4$, four successes in a row adds $x = 8$, etc., and similarly for two or more failures in a row.

Obviously, it would help to see a full example. Let's suppose `t_num = 5` and `t_quality = 0.7`. Then, depending on how the random numbers turn out, our AI training might look like this:

- BEGIN TRAINING: set $P_1 = 50\%$, $P_2 = 50\%$, ..., $P_{20} = 50\%$.
- STEP 1: we randomly pick question #12 and generate a random float 0.41253 which is **less** than `t_quality`, so now we add 1 to the value of P_{12} and get $P_{12} = 51\%$. Note that all other probability values are unaffected, we only change one probability value at a time.
- STEP 2: we randomly pick question #3 and generate a random float 0.89771 which is **more** than `t_quality`, so now we subtract 1 from the value of P_3 to get $P_3 = 49\%$.
- STEP 3: we randomly pick question #12 (by coincidence, the same question as step 1) and generate a random float 0.27602 which is again less than `t_quality`. This is the second "training success" in a row for question #12, so for question #12 we double $x = 1$ to $x = 2$. Adding 2 to the value of P_{12} now gives us $P_{12} = 53\%$.
- STEP 4: we randomly pick question #3 and generate a random float 0.00013 which is less than `t_quality`, so now we add 1 to the value of P_3 to get $P_3 = 50\%$ (back where we started).
- STEP 5: we randomly pick question #12 yet again and generate a random float 0.70599 which is more than `t_quality`. This "training failure" breaks the success streak for question #12, so we reset $x = 1$ for question #12. Subtracting 1 from the value of P_{12} gives us $P_{12} = 52\%$.
- END TRAINING: We now have $P_{12} = 52\%$ and all other probability values are still 50%.

Don't forget that probability can't be more than 100% or less than 0%. Any time you add or subtract, you should make sure your probability values are within this range.

Once training is over, its time to test the AI! Simply "ask" it all 20 questions and use the probabilities $P_1, ..., P_{20}$ to randomly determine how many it gets correct. We will say the AI has "passed" the test if it gets at least 15 questions correct. We can then repeat the training and testing multiple times to get a better sense of how likely this AI is to pass the test. Specifically, let's repeat the training and testing 1000 times, and however many times the AI passes the test will be used to determine the "pass rate". For instance, if the AI passes the test 832 times out of 1000, we will say the pass rate is 83.2%.

MINIMUM REQUIREMENTS: At minimum, for full credit your project should define the following three functions. Feel free to add more helper functions if needed. These two functions, however, are the main points you will be graded on.

- Define a function called `train` that executes the AI training procedure described above. It should have two inputs, `t_num` and `t_quality`. This function should run all steps of training (use loops and conditionals as needed) and then **return** the resulting probabilities as a list $[P_1, P_2, ..., P_{20}]$.
- Define a function called `test` that takes one input `prob_list` which will be the list of probabilities $[P_1, P_2, ..., P_{20}]$. This function should then use those probabilities to randomly determine how many questions the AI gets "correct" when tested. The function should **return** 1 if the AI passed (by getting at least 15 out of 20 correct) and should return 0 otherwise.
- Define a function called `pass_rate` that takes two inputs: `t_num`, `t_quality`. Use the `train` and `test` functions above to train and test the AI 1000 times in succession (make sure the training is

repeated from scratch each time). This function should **print** the pass rate as a percentage. Try to make a complete sentence like "The pass rate for this AI is 83.2%". This function doesn't have to return anything.

- Special note: Make sure to test as you go! Use "print" to see the value of your variables at every step and make sure things are behaving as they should. Debugging is part of the work involved.
- Another special note: your functions can assume the user will give appropriate input, no need to create a special message for bad input (although you certainly can if you want to).

THE RANDOM MODULE:

You will need to use the random module for this project. Make sure to import at (or near) the top of your code. The random module contained helpful functions like:

- `random.random()` generates a random float from 0 to 1.
- `random.randint(a, b)` generates a random integer from `a` to `b` (inclusive)
- `random.choice(L)` makes a random choice from the list `L` (or any non-list sequence).
- You can find more on the documentation page for the random module.

A FEW RULES THAT APPLY TO ALL PROJECTS:

- Your project should be submitted to Gradescope as a `.py` file (for instance, `Proj1.py`).
- **Name, UIC email, and Signature:** Please include your name and UIC email at the top of your submission as a comment. You must also copy and paste the following statement as a comment at the top of your file (make sure to put your name at the end):

  > I hereby attest that I have adhered to the rules for quizzes and projects as well as UIC's Academic Integrity standards. Signed: [[your full official name here]]

- ADD COMMENTS and documentation to clarify your choice of variables and to indicate what is happening at different steps of the program. The rule of thumb you should use in this course is to have **as many lines of comments as lines of code**. Your comments will be checked to see that you understand how your code is structured and what it is doing in the big picture. Code lacking in comments will be penalized! Make sure to add docstrings!
- Even without comments, your code should be readable and clear. Use variable and function names that are descriptive. Avoid unneeded or slow or overly convoluted calculations. For instance, in Python it is often unnecessary to use `range(len(...))`.
- While this won't be strictly enforced, the PEP 8 style guide for Python code gives really really good advice for following proper coding conventions. Take a look at it!
- It is always recommended to test your code every few lines. If you write 20 lines of code and an error is thrown, it can take longer to debug than to start over. Go slow and double check everything.
- If you're not sure where to start, try making a flowchart or try writing "pseudocode" to help visualize what you're trying to do.