Final Project Checkpoint 2
Building and Training a Neural Network
100 Points

Help Policy:

AUTHORIZED RESOURCES: Any material from the CS 471 course site and online sources regarding Python syntax only. This does not include any solutions or solution stubs for challenges similar to those asked in this assignment.
NOTE:

- Never copy another person's or group's work and submit it as your own.
- Do not jointly create a program or complete this assignment unless explicitly allowed.
- You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook).

Documentation Policy:

- You must document all help received from any source other than your instructor or instructor-provided materials, including your textbook (unless directly quoting or paraphrasing).
- The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance, and HOW it was used in completing the assignment.
- If no help was received on this assignment, the documentation statement must state "None."
- If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.
- Vague documentation statements must be corrected before the assignment will be graded and will result in a 5% deduction on the assignment.

Turn-in Policies:

- On-time turn-in is at the specific day and time listed on Canvas.
- Post the required solution files (as specified) to your Github repository. You will use the same Github repo as checkpoint 1. Ensure you add the following to your final commit comments: "FINAL TURN-IN CHECKPOINT 2"

Instructions

For this portion of the final project you will continue to build on your code you started in Checkpoint 1 (so using the same github repository). When you turn-in your completed code for Checkpoint 2, ensure you add “Checkpoint 2 Turn-in” to your commit comments.

Overview

In Checkpoint 1, you familiarized yourself with the pysc2 Application Programming Interface (API), which allows you to programmatically control and gain information about the current state of a StarCraft II game. You should have also built a start-up script that builds four supply depots, two barracks, as many marines as possible, and moves the marines to attack another quadrant in the Simple 64 map.
The goal of this checkpoint is to train a neural network to use reinforcement learning to beat the enemy. Figure 1 shows these steps:

1) Initialize agent and Starcraft II environment [done]
2) Run a scripted StarCraft II game to build barracks, supply depots, and marines [done] approximately 10 times. Move marines to random quadrants on the map.
3) For each game, record information about the game’s state, the location you sent the marines, and whether you won or lost this game.
4) Use the recorded information to train a neural network.
5) Run the scripted Starcraft II game to build barracks, supply depots, and marines. When the marines need to choose a location, use the neural network to predict where the marines should move. For each game (again, approximately 10 games), record the information about the game’s state, the location the neural network sent the marines, and whether you won or lost the game.
6) Re-train the neural network, including with this updated information.
7) Continue training the neural network (steps 5-6) until it regularly beats its opponents.
8) Record your results and save your neural network for comparison to other teams.

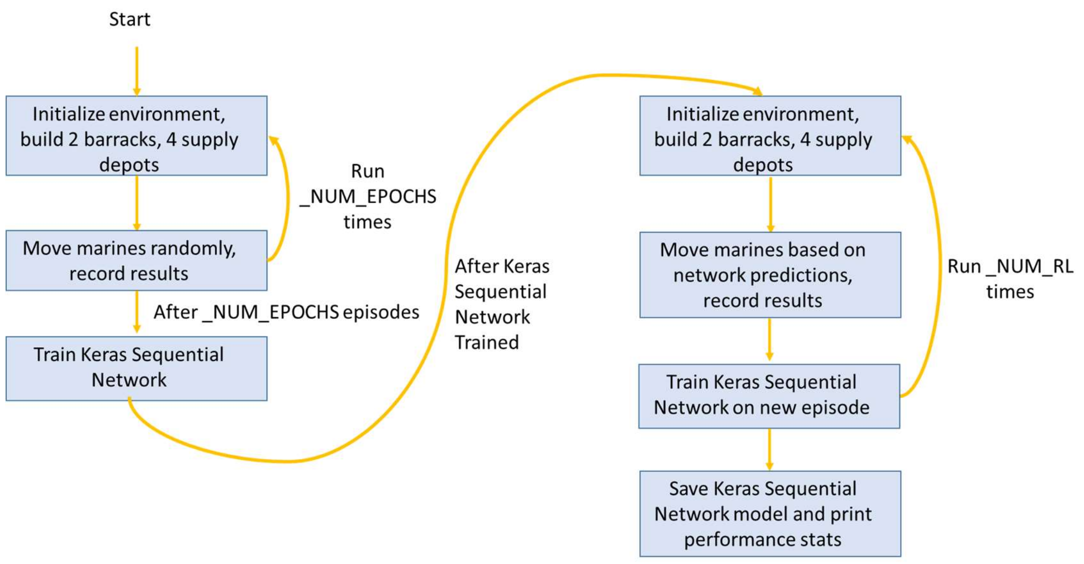
Start
Initialize environment, build 2 barracks, 4 supply depots
Run _NUM_EPOCHS times
Move marines randomly, record results
After _NUM_EPOCHS episodes
Train Keras Sequential Network
After Keras Sequential Network Trained
Initialize environment, build 2 barracks, 4 supply depots
Move marines based on network predictions, record results
Run _NUM_RL times
Train Keras Sequential Network on new episode
Save Keras Sequential Network model and print performance stats

Figure 5: Reinforcement Learning Game Loop for StarCraft II

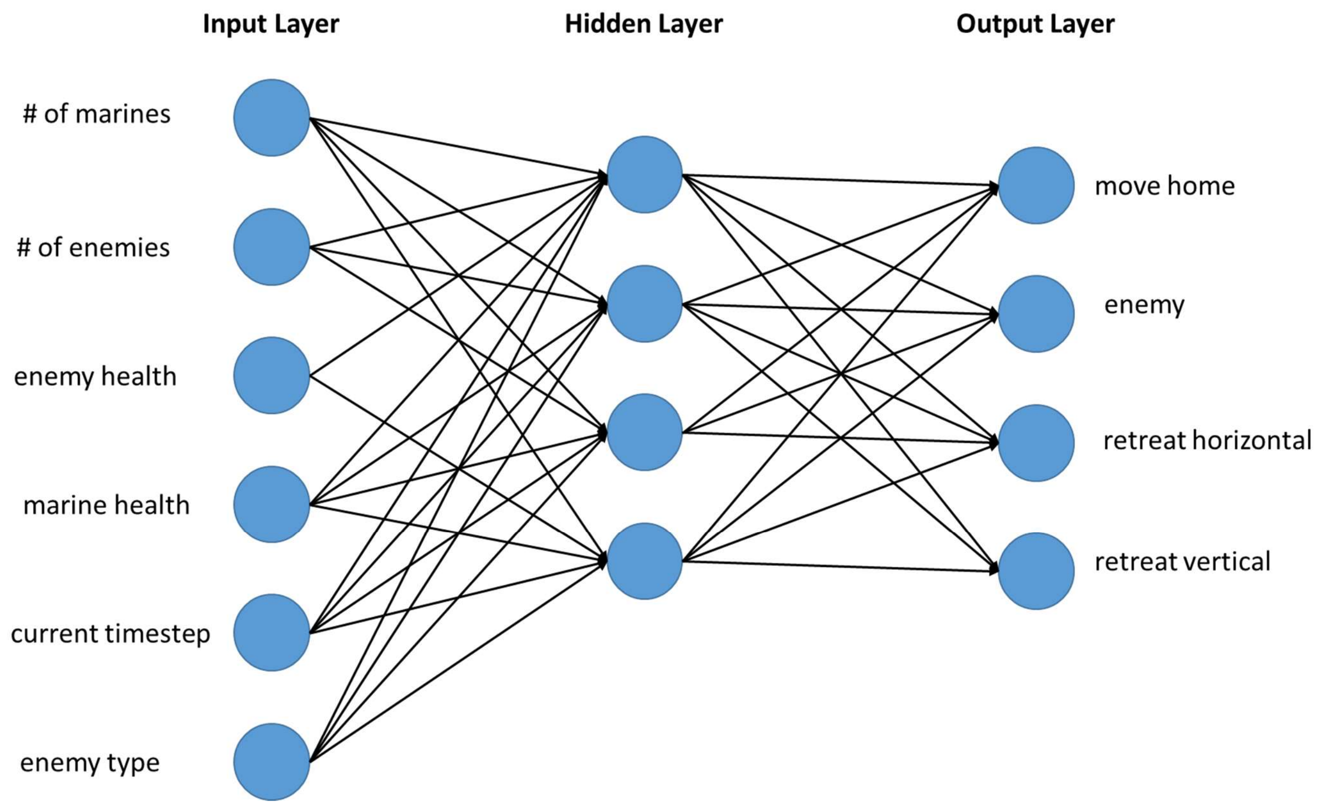

Figure 1: Example architecture of shallow neural network for Playing StarCraft II

Figure 1 provides a high-level overview of the sequential neural network you will build and train in this checkpoint to play StarCraft II.

Starting from the left, you can collect these **input values** each time you issue a command for marines to move. These input values will represent a greatly-simplified version of the current state of the game. While there are exponentially-many unique game states, these six features are plausible predictors of success against an easy opponent; additionally, they take less memory to store than every Q-state and allow Q-learning to succeed. (Remember approximate Q-learning from an earlier assignment!) These features are all accessible from the pysc2 `environment` variable. You will need to research how to extract these. The FAQ section at the end of this document contains some hints.

The output values represent the four quadrants in StarCraft II. These output values are named so that the same effect will be achieved regardless of whether your command center is located in the top-left or bottom-right quadrant of the game. For example, "retreat-vertical" will always send a marine to the quadrant vertically aligned with the enemy.

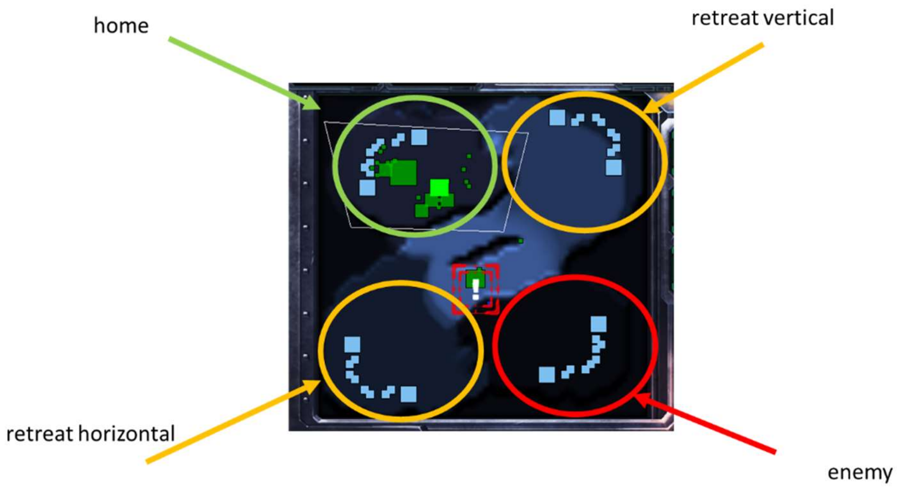

Figure 2: Illustration of the 4 positions a marine can be moved, when the player's base is in the top-left corner

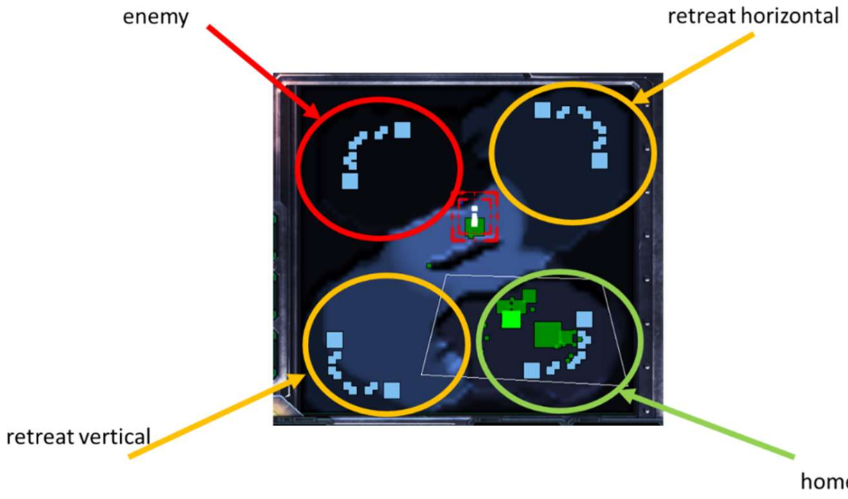

Figure 3: Illustration of the 4 positions a marine can be moved, when the player's base is in the bottom-right corner

Figure 2 and Figure 3 show both possible starting configurations for the Simple 64 minimap.

Figure 1 also shows a hidden layer with four neurons, but you can put any number of hidden layers and neurons per hidden layer in your model. Changing hyperparameters like this to make your predictions more effective will be the main goal of checkpoint 3.

Data Collection (25 points)

To train a network, you need to collect state information about the game. We recommend collecting this information each time you issue an **`attack_quadrant`** command (our name) to a set of marines. Our game is set up so that we issue individual move commands to each marine. Once each marine has moved, the “move” is complete, and we might let the marines go on attacking that quadrant for some number of steps – if we moved too quickly, they couldn't do any real damage. We can even keep training marines at the barracks while waiting to issue another move command. But we record the game state whenever we tell the first marine to move.

Now is a good time to talk about how you'll expand your checkpoint 1 code to include this checkpoint's functionality.

If you had trouble with checkpoint 1, please come for assistance.

A good place to start on this checkpoint is to refactor your code and make some small changes. You might want to place all of your commands to build and train marines into a single method, perhaps called **`build(self, obs)`** or **`scripted_start(self, obs)`**. The reason is that these things will only happen once per game, while the attack command that moves the marines will happen multiple times. Be sure to re-run your game to make sure it still works.

A second goal will be to change your code so that the marines attack a random quadrant, perhaps called **`attack_quadrant(self, obs)`**. The attack command doesn't change, only the destination (x,y) coordinates. You might want to generate a random command (enemy, retreat-horizontal, etc.), and together with the starting location of your base, you can calculate the (x,y) location. Good software design practice encourages you to put this in a helper method. The more focused each method is, the better. (We used 3 for this functionality (, and ~24 methods altogether for this checkpoint!) In fact, looking ahead to the game loop, I'll soon need some code to track which step I am on, perhaps using a finite state machine pattern (see FAQ for a few more details).

At this point, when you run your program, the marines should repeatedly run to a different quadrant to attack it, until the episode ends with either a win, loss, or tie. Using the standard game loop, it will then start a new game and repeat the process.

Now you are ready to store the game state data needed to train a neural network. You'll want to structure this in some way that is clear and easy to work with, since you'll be saving data of various types (features, actions, game results) thoughout the game. One way is to create a couple of new classes:

For example, you could create a `ScenarioRecord` (see Figure 4) for all the data for a single game episode. The `ScenarioRecord` just holds a list of data that was collected each time the group of marines moved to a new quadrant. Each entry in the list can be another list or tuple that contains all of the inputs to the neural network and the action that was selected (or another class to hold these things). An example of potential inputs are listed in Figure 1, although you can add or remove inputs to make your network work better. It will also need the result (win/loss/tie) of the game.

If you want, you can also create a second class called `HistoryRecord`. `HistoryRecord` just contains a list of `ScenarioRecords`, one per episode. You can build additional, helper methods in `HistoryRecord` and `ScenarioRecord` to access and manipulate the data from the game. For example, one could be generateNumpy(), which outputs your data in the numpy format expected by **keras**, which we'll discuss next.

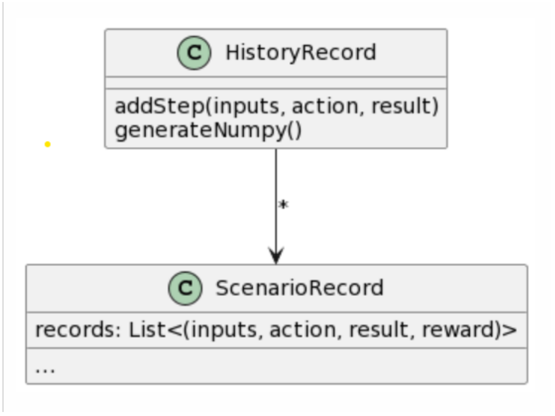
C HistoryRecord
addStep(inputs, action, result)
generateNumpy()
*
C ScenarioRecord
records: List<(inputs, action, result, reward)>
...

Figure 4: Class Diagram for objects to store game play data.

You don't have to use this exact set of classes, but it is likely to be easier to work with objects during data collection, and then only use **numpy** arrays when it's time to train the net.

Keras Sequential Network Configuration (15 points)

Keras makes building Neural Networks easy in Python. Use the lessons from class and the Keras Sequential tutorial to get started: The Sequential model (keras.io). Your model should accept an input describing the hidden network structure and then create that structure.

You may want to create a `NeuralNetwork` class, or at least a separate method. that will hold the model. You can instantiate the class to create an object representing your model. After you train your model, this class will hold the trained weights so you can easily perform predictions based on new data.

NOTE: Pay close attention to the Setup portion of the Keras tutorial. Keras requires you to install and import the Python keras and tensorflow packages (like we imported pysc2 in checkpoint 1), and will generate an error if you don't.

Focus on building a Neural Network that is generalizable, in case you want to change options while you are training the network. For example, an input to initialize the Neural Network can be a list of hidden layers, e.g., `[5 10 5]` represents three hidden layers of size 5 neurons, 10 neurons, and 5 neurons.

Data Preprocessing (15 points)

Data pre-processing will involve two major steps. First, you need to format the data you collected in ScenarioRecords and HistoryRecords into `numpy` arrays, since Keras models require that format. NumPy is a Python package for matrix and scientific calculations (think, adding MATLAB or Octave capabilities to Python). For this step, we will only utilize the NumPy arrays data structure.

NumPy arrays, like C arrays, are *fixed size*: when the array is initialized, either all of the array elements must be set, or the array size must be specified. Easy-to-digest information on how to initialize and access NumPy arrays can be found at: NumPy Creating Arrays (w3schools.com). We suggest also reading the pages linked in the sidebar to learn NumPy array functionality beyond just creating one. You will also install and import the `numpy` package to your Python environment like you imported `pysc2` in checkpoint 1.

Your final numpy array will be a 2D array with one row for each record, and one column for each feature you chose to use. For example, we used 6 features to train the network shown in Figure 1. The number of rows will vary depending on the actions taken. If your marines moved an average of 5 times per game, and we played 10 games, then would expect 50 rows. But some games will be quicker (like if it randomly chose the *enemy* action and your marines immediately defeated the enemy) and some will be longer (like those where your marines only stay home or retreat, and it ends in a tie).

The second data pre-processing step is to normalize your data (make all of the values between 0.0 and 1.0), so it can be properly interpreted by the neural network. As described in class, the process of normalizing input data prevents one input attribute with large magnitude from overwhelming other input attributes. For example, consider how large the timestep can be compared to the number of Marines! You can do this outside of keras, or by adding a Keras built-in normalization layer between the input layer and the hidden layers. A description of the layer can be found at BatchNormalization layer (keras.io). For the final turn-in of the Final Project, you will report on how you have experimented with your model to improve its performance – normalizing is just one thing you could discuss. We expect most of the discussion will center on choice of input features and experiments with various hyperparameters.

We are doing *supervised* machine learning, that is, the outputs are known and we train on input, output pairs. So you will also need to create a second NumPy array for the output values (i.e., the selected actions). We recommend that you create a NumPy array that uses one-hot encoding to represent the actions. One hot encoding means you will have a column in your array for each possible action. The selected action will be represented with a 1 and non-selected actions will be represented with a 0. For example, if our columns were in the order: move-home, enemy, retreat-horizontal, retreat-vertical, then a retreat-horizontal action would be represented by [0,0,1,0], while a retreat-vertical would be [0,0,0,1].

In order to incorporate a discounted reward value for winning or losing, you can take the reward at the end of an episode (1 = win, 0 = tie, -1 = loss) and divide it by the number of steps to play the game. Instead of representing the selected action with a "1," you can replace the "1" with the fractional reward value. Thus, a *retreat-horizontal* action that resulted in a win in 600 steps would actually be [0, 0, 0.0016667, 0] and an enemy action that resulted in a loss in 1000 steps would be [0, -0.001, 0, 0].

Run Model (10 points)

Once you have configured your model and pre-processed the input data, you will need to train your model and then run inference on the trained model to output a predicted action. The Keras tutorial: Training & evaluation with the built-in methods (keras.io) provides an excellent overview of how to train your model and use new data to conduct inference.

Ensure that your model runs and it produces an output. The output does not have to be accurate at this point (we'll tune the model in the final checkpoint), but it does need to run and produce an action. Ideally, it will produce different actions based on different states of the game.

Save and load the Model (15 points)

You'll want to save the model so you can pause training and continue training later. You will also want to save the model for the final turn-in. Your trained and tuned model's performance will be compared to other students on the Simple 64 map. Instructions for saving and loading your model can be found at: Save and load Keras models | TensorFlow Core

Configure Reinforcement Learning Game Loop (25 points)

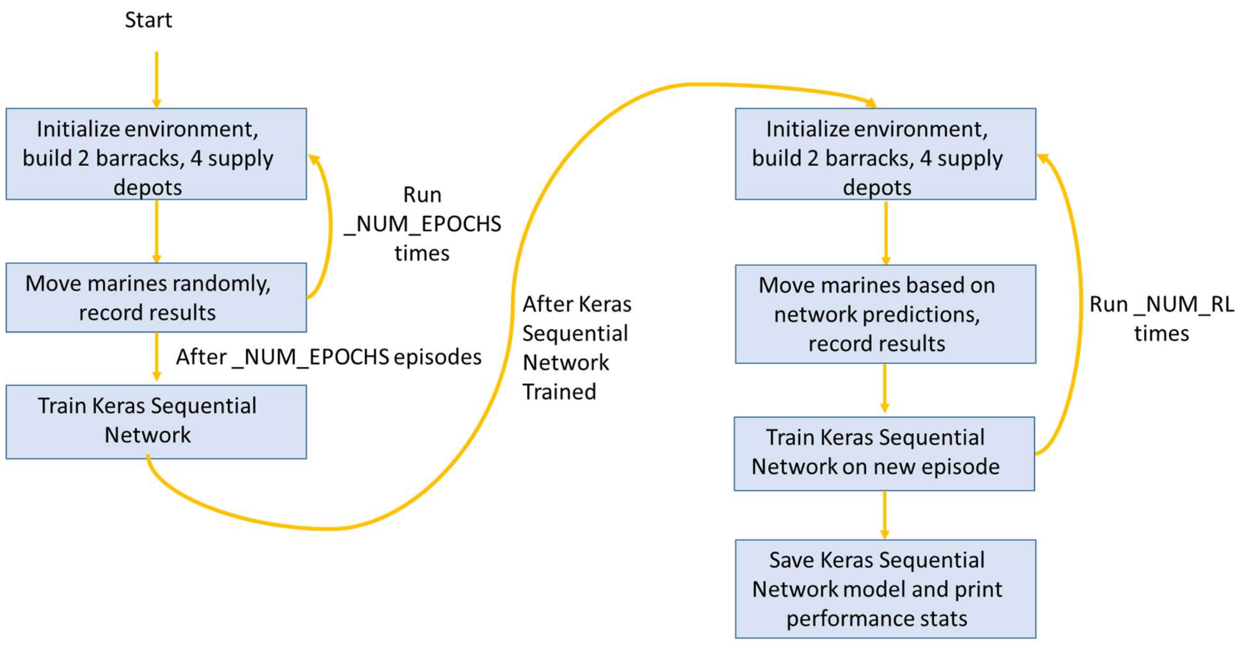
Start
Initialize environment, build 2 barracks, 4 supply depots
Move marines randomly, record results
Run _NUM_EPOCHS times
After _NUM_EPOCHS episodes
Train Keras Sequential Network
After Keras Sequential Network Trained
Initialize environment, build 2 barracks, 4 supply depots
Move marines based on network predictions, record results
Run _NUM_RL times
Train Keras Sequential Network on new episode
Save Keras Sequential Network model and print performance stats

Figure 5: Reinforcement Learning Game Loop for StarCraft II

Figure 5 shows the program flow for initializing the Keras Sequential Neural Network and utilizing Reinforcement Learning to continue training the Neural Network. `_NUM_EPOCHS` and `_NUM_RL` are constants representing how many times the network will be initialized with random data and how many times it will be trained in a reinforcement learning configuration, respectively. While debugging, small values might be fine for each, but at some point, you'll want to run it with values of at least 10 for each so you have enough data to train the network. Note that there is only one network – in reinforcement learning mode, we don't reset the network each time we train it, so each time we train on a new episode, we are fine tuning the network. In each episode, a game will be played to completion. These games are intended to build a randomized starting point for the model to train from. The models trained by DeepMind utilized replays of expert players playing StarCraft; we are basically creating our own version of these replays using random play on the Simple 64 map. Note that the early training is pure exploration while the later training is exploitation. Once you implement this game loop in your code, your project will be ready for hyper-parameter tuning (the final checkpoint).

Grading Rubric

Expected Functionality	Points Allowed	Points Earned
Data Collection: Program collects game state data (features) and the selection action (retreat_horizontal, enemy, etc) each time a move marine action is issued.	20	
Keras Sequential Neural Network Configuration: Program creates a class or method to store a NeuralNetwork. Function to create the neural network allows the user to easily specify the hidden-layer configuration.	15	
Data Pre-processing: Data is copied into a NumPy array and accepted by the Neural Network. Data is normalized either before training or by a network layer.	15	
Run Model: Neural Network trains and runs without errors	10	
Save and load the model: Model is saved and persists after the program ends.	15	
Game Loop Configuration: Program is configured to run in the manner described in Figure 5	25	
Total	100	

FAQ

Some frequently-asked questions (or that we anticipate you may ask, at least)

Q: I liked the 6 game state features you mentioned in the intro. How do I get them in my code? A: With the exception of timesteps, you can probably get most of the info from the individual units. You already wrote code back in checkpoint 1 to extract a list of all units of a certain type. You can tweak that to get health of those units, for example. For a list of things available to you, see the pysc2 repo. Like lib/features.py has a FeatureUnit class that looks promising: all of its contents are fields that units have. Since your agent is a BaseAgent, you might want to look to see what data BaseAgents have too (in the pysc2/agents folder).

Q: How does my agent know when the game ends?
A: I didn't find it right away in the docs. But Steven Brown has a whole list of other tutorials (beyond the two you did in checkpoint 1) that might be useful. Step 7 in this one showed how to detect the end of a game (and it felt a little obvious once I read it).

Q: What is a Finite State Machine?
A: The CS majors probably have a little advantage here, since they must be discussed in the required Languages course. It's like the Markov Decision Processes we discussed without random actions. Basically, an action have different effects depending on which user-defined state the program is in. The game loop pictured above might translate to a few different states (like building, attacking randomly, attacking using the network, ...) The step(self, obs)

method shouldn’t try to do ALL of that in one method. But it could use the current user-defined state to call a helper method. If I’m building, call and return my build() method. Once build finishes, we change the state to attacking randomly. So when step() is next called, it detects that and calls my attack_random() method. And so on. It’s a commonly-used pattern in some kinds of software like this.