Programming Assignment 3: Pokemon & Q-Learning

Due: Wednesday 12/10/2025 @ 11:59pm EST

The purpose of programming assignments is to use the concepts that we learn in class to solve an actual real-world task. We will **not** be using Sepia: I have developed a custom game engine for us to use. In this assignment you will be imlementing and training a Reinforcement Learning agent to play pokemon battles!

WARNING: There are **NO EXTENSIONS** for this assignment. This assignment has a **firm** due date of the last day of class!

Task 1. Reinforcement Learning via Sampling

Reinforcement Learning is a style of artificial intelligence where we aren't given input data and we aren't given ground truth. Instead, we are given a world in which our agent will act. Our agent, in order to learn, must either employ supervised or unsupervised learning techniques, so we have to cast our problem into either a Supervised or Unsupervised learning context. The trouble is that both Supervised and Unsupervised learning require data to train on which we don't have. To solve this problem we interact with the world and record the trajectories that follow. This requires some engineering decisions:

1. What senses should the agent have? The agent must perceive the state of the world in order to select an action.
2. How should the agent choose actions?
3. What should the agent use to adjust its behaviour? Initially, the decisions the agent makes will seem random. That's ok, we haven't learned anything yet! In order to adjust the behaviour, we need to engineer some way for the model to percieve "good" or "bad." This is the reward function and is something you need to engineer. There are three flavors of reward functions and you get to choose in this assignment:
    - $R(s)$. Pick this option if you personally like reasoning about states in isolation.
    - $R(s, a)$. Pick this option if you personally like reasoning about the choice of action in specific states.
    - $R(s, a, s')$. Pick this option if you personally like reasoning about how good specific actions in specific states are knowing how that action resolved.

With these engineering decisions made, we are going to employ supervised learning techniques to learn what actions to make in what states. The goal here is to learn a **policy**: a function that maps states to the (optimal) action that should be taken. We will do this with the following paradigm:

1. Play a bunch of training games. In these games, the agent is "frozen" (i.e. unable to learn anything). The point of these games are to generate training data in the form of (state, reward, next_state) triples.

    An important detail during this step is to sometimes ignore the policy. This is obviously a bad idea if we happen to know that our policy is optimal. This is less of a bad idea if we happen to know that our policy sucks. The tricky part is that we don't know whether our policy is optimal or not. We can at least make an educated guess: when our agent is brand new (e.g. just "born"), the policy is randomly initialized, so it probably sucks. The older the agent gets, the more experience it has been trained on, so the chances the policy is good improves (terms

and conditions may apply like overfitting, etc). So, we want to engineer some randomness into whether we listen to the policy when choosing an action. Many people engineer this randomness to be a function of time and make the probability we ignore the policy decrease as a function of age.

2. Convert this training data into a supervised learning dataset. How we do this depends on what our model is trying to learn, which we'll talk more about in the next section. However we choose to generate ground truth values, once we have this supervised learning dataset, we train our policy a little bit on the dataset.

3. After our model is trained, we want to see how well it performs. So we play a bunch of evaluation games where the model is frozen. The purpose of these games is not to record and build a dataset, but just to evaluate how good the policy performs.

   An important detail during this step is to always listen to the policy (unlike in the training games where we could sometimes ignore the policy). The point is to evaluate the policy so we shouldn't ignore it during these games.

Solving a Reinforcement learning problem like this is nice because it is guaranteed to learn the optimal model in the limit of infinite data. The downside to solving the problem like this is because of the practicality of "learning in the limit of infinite data." Sure, the model is guaranteed to get better eventually, the question is practically how long does it take? Training a model like this can take **ages**, its up to the quality of our engineering decisions and luck!

Task 2. Reinforcement Learning with Q-learning

Q-learning, as we will see in lecture, is a specific flavor of Reinforcement learning. In Q-learning, the model learns whats called a *Q-function*, which guesses how good an action is in a state $Q(s, a)$. If our model was optimal, then whenever our agent is in a state, we could calculate the best action to do via

$$a^* = \underset{a \in Actions(s)}{\arg\max} \; Q(s, a)$$

Q-learning tells us how to convert the training data generated from samples into a supervised learning dataset. Most of the work occurs in generating the ground truth. If we have a bunch of samples (s, a, s') from playing the game (s was a state, a was the action taken in state s, and s' was the resulting state of applying a in s), then we generate ground truth using whats called the *Bellman* equation (for Q-functions):

$$Q(s, a) = R(s, a) + \gamma \max_{a' \in Actions(s')} Q(s', a')$$

Reminder that there are multiple flavors of rewards: you don't have to choose $R(s, a)$ like we've done here. The Q-Bellman equation just tells us what Q-value our model *should have* predicted, so we can use it as ground truth to train our model against. This is technically called **temporal difference learning** and we'll talk about it a bunch in class. The tl;dr is that we can apply this equation to generate our ground truth and then we learn a model to minimize the following loss function:

$$\mathcal{L}(\hat{\vec{q}}, \vec{q}_{gt}) = \frac{1}{N} \sum_{i=1}^{N} \left(\hat{q}^{(i)} - q_{gt}^{(i)} \right)^2$$

Task 3. Reinforcement Learning with Neural Networks

Reinforcement learning requires us to learn, in the case of q-learning, a single Q-value per (state, action) pair. This mapping is not feasible to store as a table for all but the simplest toy problems. For most problems (Pokemon included), there are simply too many states for us to allocate a single float per entry. When we encounter a scenario like this, we often use a neural network in place of the tabular function. Neural networks learn functions, so we can use them in this setting to learn the function while skipping the memory requirements for an exhaustive table.

Like all ML models, neural networks learn a kernel function $K(\vec{x}, \vec{y})$: a similarity function. This kernel is both a blessing and a curse. A kernel is a blessing because, if you see an example (that you've never seen before), you can leverage the knowledge that you've learned from similar examples (the kernel tells you how similar things are). A Kernel is a curse because of how it is constructed. A Neural Network learns a kernel through its parameters. If we change a single parameter (i.e. tweak one of the many parameters inside the network), we change the **entire** kernel and therefore we change how the network views **every** example it will ever see. This means that if we try to improve performance on a single example, we can potentially destroy the knowledge that the network currently has contained within it. It is for this reason that when we train neural networks, we don't optimize for a single example: we take the average gradient across a batch of examples.

While we technically should update our network as soon as a transition occurs, we will instead aggregate them in a buffer called a *replay buffer*. The purpose of the replay buffer is to account for the curse of the kernel: if we updated our network with a transition the second we observe it, we would be optimizing for a single example and run the risk of our network forgetting all the knowledge it has already learned. So, instead of updating the network as soon as transitions occur, we will collect them into the replay buffer, and wait until the network has finished playing a bunch of games before using the replay buffer to update the network. To be clear, a replay buffer is a hack: this is a bandaid solution that does not solve the underlying problem. A more meaningful approach would be to address neural networks treating each example as i.i.d, but thats a topic for another time and is an active area of research.

Therefore, to be more accurate, we should update the training paradigm from the earlier sections. Training is organized into cycles. Each cycle has the following sections, which are executed in order:

1. Play a bunch of training games, the sole purpose of which is to observe (and record) trajectories to populate the replay buffer.
2. Convert the replay buffer into a supervised learning dataset (using our Bellman equation from earlier to calculate ground truth). Train the neural network on this dataset.
3. Play a bunch of evaluation games with the updated neural network, the sole purpose of which is to evaluate the performance of the network.

This cycle is repeated an enormous amount of times. Eventually, the network will get really good, and therefore our agent will get really good.

Task 4. Copy Files

Please copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/pa3/lib/pokePA-<VERSION>.jar` to `cs440/lib/pokePA-<VERSION>.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/pa3/lib/argparse4j-<VERSION>.jar` to `cs440/lib/argparse-4j-<VERSION>.jar`
  This is a java implementation of python's `argparse` module. It makes it easier for programs to define and parse command line arguments.
- Copy `Downloads/pa3/junit-<VERSION>.jar` to `cs440/lib/junit-<VERSION>.jar`
  This library provided unit testing functionality in Java.
- Copy `Downloads/pa3/hamcrst-<VERSION.jar` to `cs440/lib/hamcrest-<VERSION>.jar`
  This library is needed for the `junit` library to function correctly.
- Copy `Downloads/pa3/src/pas` to `cs440/src/pas`.
  This directory contains our source code `.java` files.
- Copy `Downloads/pa3/pokePA.srcs` to `cs440/pokePA.srcs`.
  This file contains the paths to the `.java` files we are working with in this lab. Just like last lab, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- Copy `Downloads/pa3/doc/pas` to `cs440/doc/pas`. This is the documentation generated from `pokePA-<VERSION>.jar` and will be useful in this assignment. After copying, if you double click on `cs440/doc/pas/pokemon/index.html`, the documentation should open in your browser.

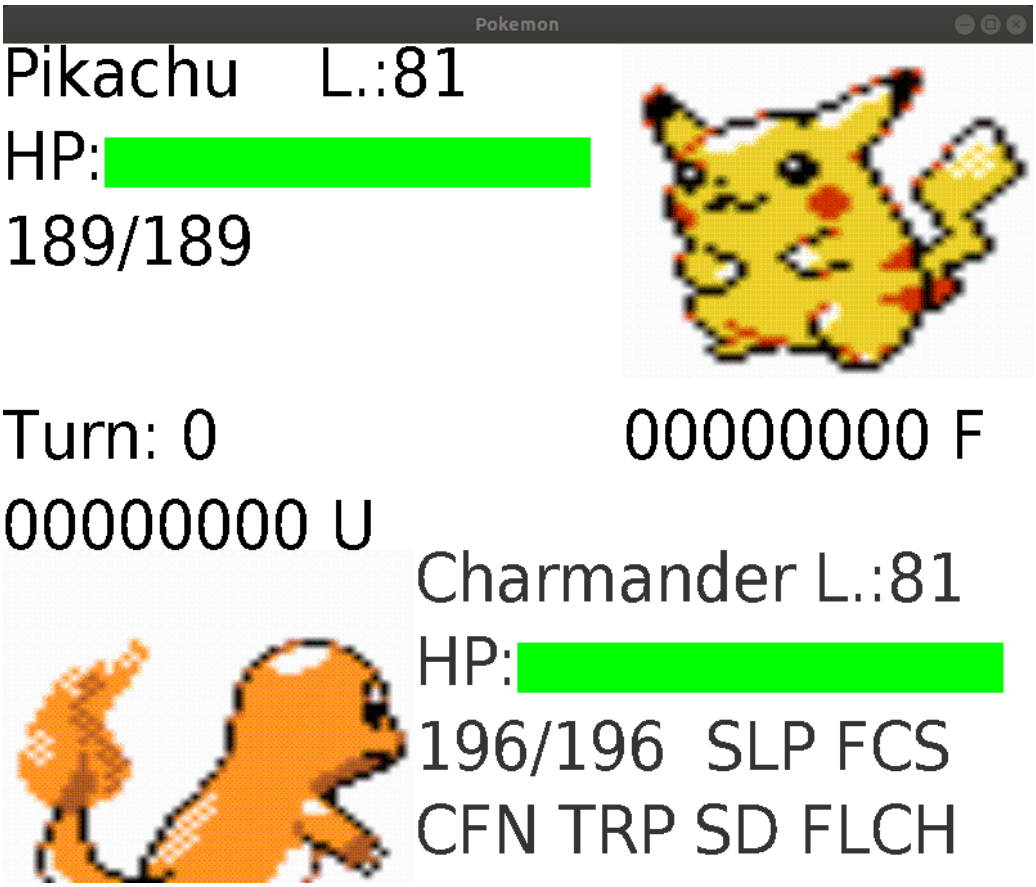
Task 5. Test run

If your setup is correct, you should be able to compile and execute the given template code. You should see a window appear showing two pokemon teams battling against each other.

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @pokePA.srcs
java -cp "./lib/*:." edu.bu.pas.pokemon.RandomBattle -t 1000

# Windows. Run from the cs440 directory.
javac -cp ./lib/*;. @pokePA.srcs
java -cp ./lib/*;. edu.bu.pas.pokemon.RandomBattle -t 1000
```

By default, the agent controlling each team selects moves uniformly at random. The game rendering will look something like this:



There is a lot of information about the game being conveyed in the rendering. Besides describing the turn number, let us focus on the Charmander's rendering as it shows more settings than the Pikachu:

- `Turn: XYZ` shows the current turn number of the game. This number will increment as the game progresses.
- `0000000 U`. This string either appears above or below the sprite of a pokemon. The integer numbers show the stage multipliers for the seven stats a pokemon keeps track of. During battle, a pokemon's stats can either be strengthened or weakened, and that is reflected in these numbers. If the number is positive, that pokemon's stat has been strenghtened. If the number is negative, that pokemon's stat has been weakened. Please click here to learn about how these values are used to change a pokemon's stat. We are using GenI-II rules when they are combined, and Gen II rules when they are separate. The stats these numbers modify are in the following order:

1. ATK: The strength of a pokemon when dealing physical attacks.
2. DEF: The defense of a pokemon against physical attacks.
3. SPD: The speed of a pokemon.
4. SPATK: The strength of a pokemon when dealing special attacks.
5. SPDEF: The defense of a pokemon against special attacks.
6. ACC: The accuracy of a pokemon. This affects how likely a move this pokemon uses is to hit its target.
7. EVASIVE: The evasiveness of a pokemon. This affects how likely a move is to hit this pokemon.

Each stage multiplier is bounded in the interval [-6, +6].

The `U` shows the height of a pokemon. If a pokemon is underground (i.e. using the move `Dig`), you will see `U`. If a pokemon is flying in the air (i.e. using the move `Fly`), you will see `F` (as in the case of the Pikachu). If a pokemon is neither flying nor underground, you will not see a character in this location.

- `Charmander L.:81`. This string shows the name of the pokemon as well as its level. Generally, the higher the level, the more powerful a pokemon is.
- `HP: <BAR>`. This shows how much health a pokemon has as a percentage of its initial health. Over 50% health and the bar is green, less than 50% but over 20% and the bar is yellow. Below 20% the bar will be red.
- `196/196`. This is a numerical format of the HP shown in the HP bar.
- `SLP`. This is where the persistent status (called a `NonVolatileStatus` in the code) of a pokemon will be shown. Currently the Charmander is asleep, but there are other persistent statuses that a pokemon can have. Note that a pokemon can only have a single persistent status, so if they are afflicted with one already, they cannot be afflicted with another until the existing one is removed. Here are the persistent statuses that a pokemon can be afflicted with:
  - PARALYSIS: If paralyzed, a pokemon has a 25% of losing their turn due to being fully paralyzed. Additionally, a pokemon's SPEED is rounded down to 75% of its value when determining the move order. The string `PAR` will appear when a pokemon is paralyzed.
  - POISON: When poisoned, a pokemon will take 1/8 of its maximum HP as damage at the end of each turn. The string `PSN` will appear if a pokemon is poisoned.
  - TOXIC: This is like poison but on steroids. When affliced by toxic, a pokemon will take $N/16$ of its maximum HP as damage at the end of every turn, where N counts the number of turns since being afflicted with toxic. The string `TOX` will appear if a pokemon is affliced with toxic. The count N will reset when the pokemon is no longer afflicted with toxic.
  - BURN: Like poison, a pokemon will take 1/8 of its maximum HP as damage at the end of every turn. However, if a pokemon is frozen and a move applies a burn, that pokemon will thaw and will no longer be frozen (and it will also not be burned). The string `BRN` will appear if a pokemon is burned.
  - FREEZE: When a pokemon is frozen, it will stay frozen (and unable to make a move) for at maximum 7 turns. At the beginning of that pokemon's move, it has a 9.8% chance of thawing on its own. As mentioned, a frozen pokemon can be thawed early by being burned when frozen. The string `FRZ` will appear if a pokemon is frozen.

  - SLEEP: When a pokemon is asleep, it will sleep for a maximum of 7 turns. Like being frozen, it will have a 9.8% chance of waking up at the beginning of its move. This deviates from the traditional sleep rules, but I found it was easier to implement. The string `SLP` will appear is a pokemon is asleep.
  - NONE: A pokemon that does is not afflicted with anything gets this value. No string will appear.

- `FCS`. This string appears if a pokemon has used a move called `Focus Energy`. This move increases the chances that a pokemon will inflict a critical hit the next turn. Unlike Gen-1 pokemon, where this is bugged (in Gen-1, focus energy actually **halves** your critical hit chance instead of **doubling** it), I have made sure the chances of a critical hit have been **doubled**. This is considered a "volatile" stat, so if you use `Focus Energy` and then swap the pokemon out, this stat will be lost.

- `CFN`. This will appear when a pokemon is confused. A pokemon will remain confused for anywhere between 2 to 5 turns maximum. At the beginning of using a move, a pokemon has a 50% chance of doing damage to itself instead. This is cured when the confused pokemon is swapped out or the confusion duration ends.

- `TRP`. This will appear when a pokemon is trapped. Trapped pokemon cannot be swapped out until the trap ends. The duration of a trap are set by moves designed to trap pokemon (e.g. `Fire Spin`, etc.).

- `SD`. This will appear if a pokemon is seeded. The move `Leech Seed` plants seeds on the opponent pokemon, which at the end of every turn, sap health from the seeded pokemon and deliver that health to the non-seeded one (who recovers a little bit of health). This is cured when the seeded pokemon is swapped out.

- `FLCH`. This will appear if a pokemon has flinched. Some moves may cause the target of a move to flinch when struck by that move. When a pokemon flinches, it will not execute the move it had inteded to during that turn (if the flinched pokemon goes after it has flinched). This is reset at the end of every turn.

Task 6. Command Line Arguments

If you wish to change the agent controlling team1, you can do so by specifying the `--t1Agent` command line argument when running the program (here is the Linux/Mac version of how you would do this):

```
java -cp "./lib/*:." edu.bu.pas.pokemon.RandomBattle --t1Agent <CLASSPATH_OF_AGENT_TO_USE>
```

There are a few other command line arguments you can provide to this program. To see the list of what arguments you can provide, please trigger the help message by doing the following (here is the Linux/Mac version):

```
java -cp "./lib/*:." edu.bu.pas.pokemon.RandomBattle -h
```

There are multiple executables within `pokePA-X.X.X.jar` all within the package `edu.bu.pas.pokemon`. The executable mentioned so far `RandomBattle` does exactly that, it generates two randomly chosen teams (random pokemon with random moves) and plays a single game. If you wish to design your own teams you can run the `TeamCreator` executable. This executable will provide a GUI for you to build a team and save that team to a user-specified file on disk. Given two pre-chosen teams, the

executable `CustomBattle` is for pitting those specific teams against each other.

The main executable you will use the most is `Train`. This executable is how you will train your agent. While the other executables render by default (the rendering can be turned off), there is no rendering available for `Train`. As discussed in lecture and detailed in the earlier sections, the purpose of `Train` is to launch a massive "training run" where you iteratively fit a neural network by collecting lots of experience playing the game, coverting that experience into a supervised learning dataset, and using that dataset to fit the model. This executable has quite a few arguments: some have to do with training hyperparameters, some let you choose how you train the network, some manage files, etc. Trigger the help string to see what is available!

Task 7. The Transition Model

One of the things that makes Pokemon such an interesting RL application is that we have direct access to the transition model: moves in Pokemon have probabilites of landing, status effects and non-volatile statuses have random chances of resolving, etc. I have provided part of the transition model for you, and that is the method `getPotentialEffects` in the `Move` and `MoveView` class. This method will return a `List` of all potential outcomes of a move along with their corresponding probabilities. If you choose to, you can make your Q-agent use the transition model. Q-learning (specifically learned via temporal difference learning) assumes that the network will "absorb" the transition model (e.g. the network will hopefully learn that a certain action in a certain state has low value because that action is unlikely to resolve in the desired way in that state). This is nice because as long as the network has the capacity to learn this behavior its not unreasonable to ask it to. The downside of doing so is that we're asking the network to learn a more complicated problem which has its own risk of overfitting, etc.

One way to make this problem easier to learn is to use the transition model directly but can only be done if the transition model is either directly accessible or you dedicate some other model to learning transition probabilities (which is hard). In Pokemon we have the transition model, so you can choose to incorporate it if you want! If you do, you need to finish the implementation of the transition model. My `getPotentialEffects` only produces the ways that a move can resolve *if that move actually attempts to resolve*. There are several layers of randomness in Pokemon (listed in order) that happen every turn of the game:

1. The ordering of which pokemon's moves will occur happens first. Each move has a priority stat to it, and pokemon have speed stats. If one pokemon submits a move with a higher priority (i.e. larger priority value) than the other, the move with the higher priority will occur first. Moves where a user decides to switch out their pokemon (called `SwitchMoves`) have high priority. If both pokemon submit moves with the same priority, then the speed stat of the two pokemon deicide the order (i.e. the faster pokemon goes first). Remember that being paralyzed reduces you speed to 75% of its original value here! Finally, if both pokemon are equally fast, then its a pure 50/50 chance you go first. The ordering of moves certainly can affect the game in different ways (for instance if we go first we may put our opponent to sleep so their move **doesn't** get a chance to resolve). You will have to account for this!

2. Once the move order is decided, the moves submitted are attempted to be executed in that order. However, if a pokemon is asleep/frozen/paralyzed, then there is a chance that the submitted move will not occur. Provided that the submitted move makes it through this check, if the pokemon is confused, there is a 50% chance that the submitted move will not occur and the pokemon will hurt itself instead. You will have to account for this!. When dealing with hurting yourself due to confusion, I would recommend generating a synthetic move and use the `getPotentialEffects` api. You will have to do something similar to this to generate the synthetic move:

```
hurtYourselfMove = new Move(
    "SelfDamage",           // move name
    Type.NORMAL,            // damage type (should be typeless but we'll tell the
                            // to ignore STAB and type terms in damage calculation
    Category.PHYSICAL,      // move category
    40,                     // base power for hurting yourself from confusion is 40
    null,                   // infinite accuracy
    Integer.MAX_VALUE,      // number of uses
    1,                      // critical hit ratio
    0                       // priority
).addCallback(
    new MultiCallbackCallback(
        new ResetLastDamageDealtCallback(), // new damage so reset old value
        new DoDamageCallback(
            Target.CASTER,                  // hurt yourself
            false,                          // dont include STAB term in damage calc
            false,                          // ignore type terms in damage calculation
            true                            // damage ignores substitutes
        )
    )
);

List<Pair<Double, BattleView> > confusionDamageOutcomes = hurtYourselfMove
    .getView().getPotentialEffects(...); // get all the potential outcomes
```

3. If the move you submit has made it this far, it can resolve in a bunch of different ways. For instance, moves can miss, or the damage they cause can be random, or they might cause additional side effects, etc. I have handled getting the potential ways the move can resolve for you with the `getPotentialEffects(...)` api.

4. Finally, once the turn is over, there are some post-conditions that can occur. For instance, any damage caused by POISON/TOXIC/BURN/SEEDING needs to be calculated and applied. Flags need to be reset (like FLINCHING), counters for FREEZE/SLEEP need to be decremented, etc. If a pokemon has fainted (i.e. its HP is now zero), then that pokemon must be swapped out for a new, unfainted one. Since you don't know which pokemon you want to choose if your pokemon faints, or if its your opponents, you don't know which one they'll choose, you have to consider all available options. You will need to account for this in your transition model! **Note** that this only occurs at the end of every **turn**, **not** after every Move.

Task 8. `class CustomSensorArray`

Please take a look at the file `src/pas/pokemon/senses/CustomSensorArray.java`. The purpose of this class is to convert a `BattleView` into a row vector `Matrix` object which you can feed to your neural network. When implementing the `getSensorValues(final BattleView state, final MoveView action)` method you will need to answer the following questions:

1. "how should my neural network perceive the battle?"

2. "how should my neural network perceive the action?"

Whatever sensing choices you make, you will need to implement that functionality in this file.

Task 9. `class PolicyAgent`

Please take a look at the file `src/pas/pokemon/agents/PolicyAgent.java`. The purpose of this class is to wrap around your neural network instance. You will need to finish the following methods:

1. `initializeSenses(Namespace args)`. This method should create your `CustomSensorArray` and store it as a field so that we can use it when predicting Q-values.
2. `initialize(Namespace args)`. This method initializes your neural network as well as calls `initializeSenses`. If you engineer anything else you need to initialize it can go here.
3. `initModel()`. This method instantiates a neural network. See the starter code for an example. We are only using feed-forward neural networks in this class. The number of input features should be the same as the number of sensor measurements your `CustomSensorArray` produces, and the number of output is 1 (a single Q-value).
4. `chooseNextPokemon(BattleView view)`. This method is responsible for choosing a replacement pokemon when the pokemon you are controlling faints.
5. `getMove(BattleView view)`. This method is responsible for choosing which `MoveView` you want to submit to the game (to then attempt to resolve). The returned move must be a **legal** move. If you return an illegal move (or `null`), it will be counted as a timeout against you. Too many timeouts and you auto-lose the game!
6. `afterGameEnds(BattleView view)`. This method is called at the end of every game. It is useful for diagnostics/statistics calculations. Be aware that I will disable printouts on the autograders (printouts are currently disabled for `edu.bu.pas.pokemon.Train`)

If you want to include the transition model into your agent (as discussed in an earlier section), you will likely want to override two more methods:

1. `argmax(final BattleView state)`. This method returns the action that argmaxes the Q-function. It currently does not use the transition model. If you choose to incorporate the transition model you will likely want to argmax the Q-Bellman equation which uses transition probabilities.
2. `makeGroundTruth(...)`. This method calculates supervised learning ground truth from a `ReplayBuffer` object (see the documentation for the full signature). It currently implements the Q-learning temporal difference ground truth discussed in an earlier section. If you want to include the transition model, you will likely want to implement this using the Q-Bellman equation which uses transition probabilities.

Task 10. `class CustomRewardFunction`

Please take a look at the file `src/pas/pokemon/rewards/CustomRewardFunction.java`. The purpose of this class is to isolate your reward function. As mentioned in an earlier section, you have a choice of which flavor of reward function you want to implement:

- $R(s)$. Pick this option if you personally like reasoning about states in isolation.
- $R(s, a)$. Pick this option if you personally like reasoning about the choice of action in specific states.
- $R(s, a, s')$. Pick this option if you personally like reasoning about how good specific actions in specific states are knowing how that action resolved.

Each of these flavors is implemented with a different method inside `CustomRewardFunction`. You are welcome to design multiple kids of rewards and then combine them together: just like heuristics there are typically more than one way of measuring the "quality" of decisions/states. If you do this, the vocab term for this is called reward *shaping*.

Task 11. Training

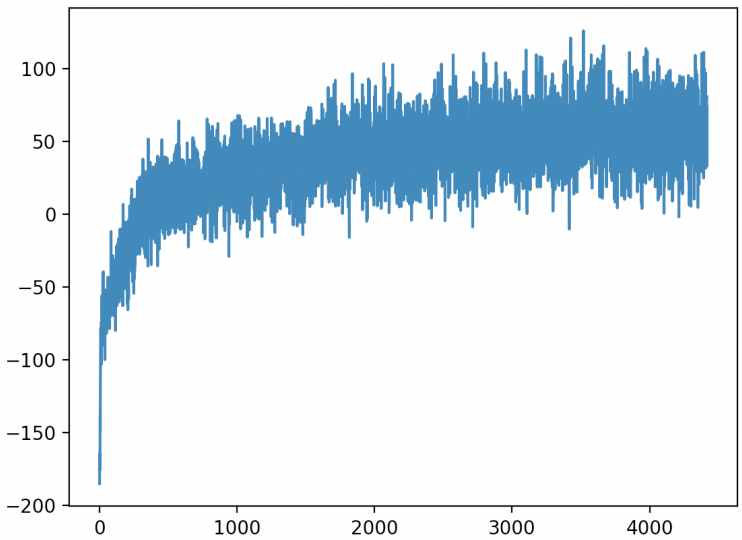
To train your agent, you should develop your code locally and make sure that it doesn't error, and that it measures quantities the way you intend them to be measured. Once you have that working, you will want to train your agent on tens of thousands, if not hundreds of thousands, if not millions of games. To do this, we are giving you access to the SCC. We will provide you a tutorial of how to use the SCC in lecture.

Task 12. Checking on Performance

When training your agent (which will take days), you will want to check in on it periodically. My code will, after every cycle, save your neural network to disk in a directory called `params`. You are free to change the destination of the saved models, and on the SCC I encourage you to save your models to the `projectnb` directory. When should you stop training, and which version of your network is the best? These are questions answered by the evaluation section of every cycle. The average trajectory utility is printed out after every cycle: this is information you will want to keep and to plot. To do so, when it is time for you to launch your agent's massive training run, you should run it like so (only shown for linux since I am assuming you'll be running this on a linux machine):

```
java -cp "./lib/*:." edu.bu.pokemon.Train <OTHER_ARGS> | tee my_logfile.log
```

The `| tee my_logfile.log` part of this command will pipe stdout to a file called `my_logfile.log` which will exist in the same directory that the shell is in when the command was run (again I would recommend specifying a path to the `projectnb` directory). Periodically, you can download your `my_logfile.log` to your laptop, and then use `learning_curve.py` to plot the average trajectory utility as a function of time. The way you will know your model is learning is if you get something that looks like this



The x-axis is the number of cycles that the model has run for (this curve shows 4000 cycles), and the y-axis is the average trajectory utility.

Task 12. Performance (100 points)

There are three difficulty settings in this assignment. Each difficulty you will be facing different gym leaders from the Pokemon universe. Please click here for more information on the different gym leaders and their teams:

- EASY: You will face Brock's Gen-1 team (a Geodude and an Onix). You are given a Bulbasaur, who has two moves super-effective against Brock's pokemon. This battle should be easy to win.
- MEDIUM: You will face Sabrina's Gen-1 team (Kadabra, Mr. Mime, Venemoth, and Alakazam). You are given a Snorlax, a Charizard, a Parasect, and a Beedrill. This will be difficult match to win: even though you have pokemon that know moves super effective against Sabrina's, your pokemon are also partially weak to psychic damage.
- HARD: You will face Lance's Gen-1 team (Gyarados, Dragonair, Dragonair, Aerodactyl, and Dragonite). You are given a Snorlax, a Venusaur, a Dragonite, a Lapras, and a Machamp. Even though you have pokemon who know moves that are super effective against Lance's, his pokemon do tons of damage, and your pokemon are also partially weak against his. This should be a hard fight to win.

You will face off against each of these opponents 10 times on the autograder. To earn full credit, you will have to beat EASY 100% of the time, MEDIUM 75% of the time, and HARD 50% of the time.

Task 13. Extra Credit (50 points)

We are currently devising a custom agent to play as trainer **Red** (from the Pokemon gameboy games). Red will be **extremely** difficult to beat. We will release more information on this in the future. If you can beat this agent at all, we will award you full extra credit.

Task 14. What to Submit

Please submit all of the files you wrote. If you invented more files than this to help train or something please submit them too! As your model trains, my code will write the parameters of the model to files after every cycle. You need to pick one of these files (ideally the file that corresponds to the model that has the best performance), rename it to `params.model`, and submit it along with your code on gradescope. Reminder that this filename is case sensitive and must be **exactly** `params.model` for the autograder to recognize it.

Task 15. Tournament Eligibility

In order for your submission to be eligible in the tournament, your submission must satisfy all of the following requirements:

- Your submission must be on time.
- You do not get an extension for this assignment.
- Your agent compiles on the autograder.
- Your agent can beat the `EASY` difficulty more than 100% of the time, the `MEDIUM` difficulty more than 60% of the time, and the `HARD` difficulty more than 40% of the time. Since these agents are fast, I **will** be using the autograder percentages for the tournament qualifiers.