67750: Advanced Practical Machine Learning	2019/20

APML Project - Snake

Prof. Shai Shalev-Shwartz *TA: Omri Bloch*

1 The Task

In this exam you will implement two reinforcement learning agents that learns to play Snake (if you didn't have a Nokia phone and don't know Snake, look here). The first agent will be a **Q-learning agent with a linear function approximation**, and the second agent will be **an agent of your choice**.

Your agent will be a snake on a 2D grid of a certain size. The grid will have obstacles and different kinds of fruit (good ones and bad ones), along with possibly other snakes. Every turn, you will be able to choose whether to go left, right or straight. Your goal during the game will be to eat as many good fruit as you can while avoiding the obstacles, the bad fruits, other snakes and your own tail.

1.1 State Space

In every turn, you will receive a state which is a tuple containing the current board and the snake's head - (board, head).

The board is a NumPy array of the relevant shape, with every element containing the value which represents the objects in the board. The values will always be integers in the range [-1,9]. However, **don't assume that the values will mean the same thing when your code is evaluated** - you should build your code as to be able to learn with no prior knowledge of the meaning of the values.

The head is another tuple which contains two objects - the head position and the direction. The head position is a Position object - it has two coordinates which you can access with regular indexing like you would a tuple. The direction is one of the following strings: ["N","S","W", "E"] .

You can use all of the above information to figure out where your snake is on the board, along with the position of the head and the direction it is currently going.

1.2 Action Space

The action space of the game is choosing between moving left, right or forward: $\{L, R, F\}$.

1.3 Rewards

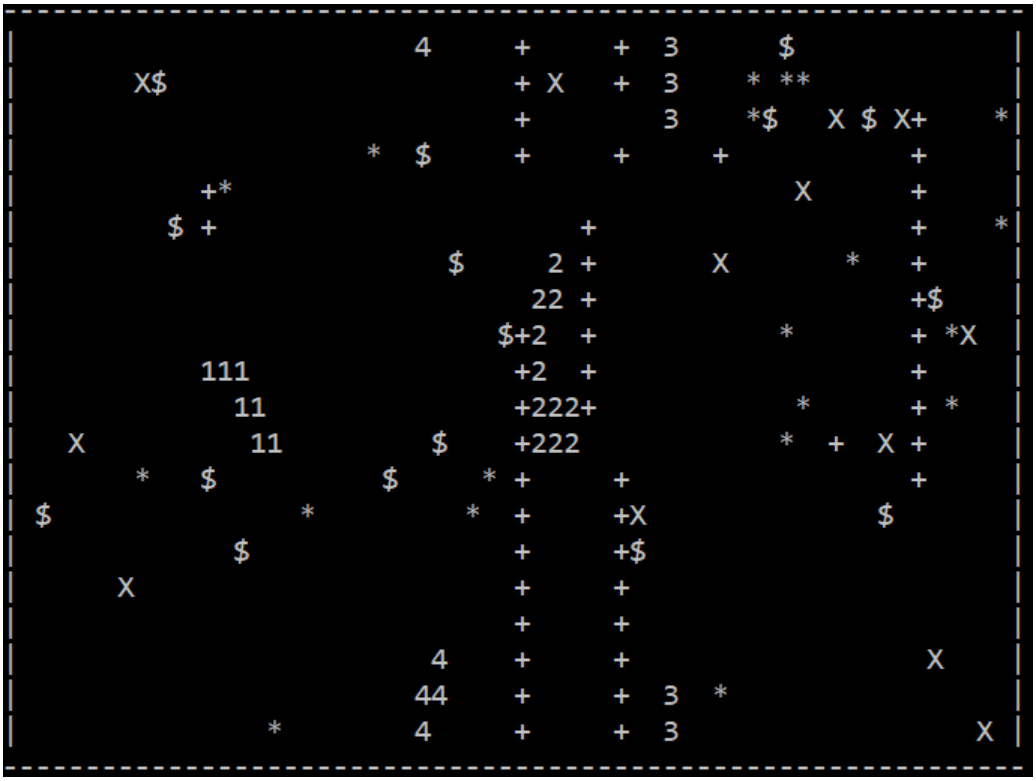
The base reward of the game is 0, for the times when the player moves into an empty space.

If a player hits an obstacle or another snake, the reward will be “-5”.

Finally, if the player picks up some fruit, the reward will be according to the fruit type (either a positive or negative reward).

1.4 Visual Example

You will be able to visualize the snake game using the supplied code. The following is an example of a frame from a game between four players:



The numbers are the positions of the four players. The + signs are obstacles which are randomly drawn in the beginning of the game and remain fixed. The three signs $\{*, \$, X\}$ are different kinds of food, which randomly pop up on the board during the game and also pop up in place of snakes which die due to collisions.

When you pick up a good fruit, your snake will grow larger (as you can see from player 2 in the above image).

The edges of the board aren’t walls or obstacles - you can go through them and pop up from the other side (as you can see for players 3 and 4 in the above image).

1.5 The Policies You Should Implement

As explained, there are two policies which you are required to implement - a linear policy and a policy of your choice.

1.5.1 The Linear Policy

This policy is here to force you to start out simple when solving the project. You should implement Q-learning as described in class, where instead of a table for your Q-function, you will approximate it using a linear function of the state and the action. You may (and should) process the given state so that it has a lower dimensionality than the original board, as suggested in section 4.5.

Note that your policy isn't expected to get incredible results here - it is more of a sanity check to see that your learning algorithm works and that you are able to generally learn that obstacles are bad and fruit are good. This can be learned with a relatively small state representation and very simple parameters, so there's no need to try and do anything too fancy here.

The linear policy will be worth 50% of your project grade.

1.5.2 The Policy of Your Choice

This policy is where you can be more creative, and implement a learning algorithm of your choice to solve the problem. We saw many possible algorithms, and you can implement any of them (or something that wasn't learned in class but you feel is relevant).

The second policy will be worth the other 50% of your grade.

2 Code and Environment

In the provided zip file you will find the following:

1. Snake.py - the main script which runs the game
2. Policies Folder - the folder in which to place your policy file (along with an example policy)
3. sbatch_example.sh - an example of how to run your code on the sm-cluster to test your running time

2.1 Implementing a Policy

Each policy in the game is a stand-alone process (a multiprocessing.Process instance) that communicates with its agent via python Queue objects. All this is encapsulated and hopefully you won't need to worry about it. In order to implement a policy, all one has to do is implement a class which inherits from the "Policy" class (defined in the base_policy.py file), and override four simple methods:

1. cast_string_args - a function which allows you to accept arguments for your policy when running Snake.py (like a learning rate, for example).
2. init_run - a function which will run in the beginning of the session, which you can use for initialization (for your Keras model or weights vector, for example).

3. learn - a function which will be called every 5 rounds, where you can improve your policy.
4. act - a function which will be called before every round, where you return the desired action (given the current state and previous state-action-reward triplet).

The functions are all documented in the "base_policy.py" file, and an implementation example is given in the "policy_avoid.py" file. This policy simply avoids collisions with obstacles and snakes without paying any attention to fruit. It also acts randomly with a probability decided by the "epsilon" parameter that you can send to the policy in the beginning.

The python file which implements the policy should be placed in the "policies" folder.

2.2 Game Flow

The game starts with the creation of the processes for the different agents (up to 5 possible agents in a single game), followed by a default waiting time of 60 seconds (-pit option in the command line) to make sure they are all initialized correctly. During this waiting time, all agents run their "init_run" functions. This should be enough time for you to initialize your variables and anything else needed for your policy.

The session then runs for a predefined number of rounds. In each round, every agent receives the current state along with the previous state-action-reward triplet using the "act" function. The agent returns an action in $\{L, R, F\}$. The game waits for "player_action_time" seconds for a response. otherwise it will assume that the action is F. At the end of each round, the game checks which snakes died and which snakes ate fruit, and distributes the rewards accordingly. Every 5 rounds, the agents will also get their "learn" function called, with a longer waiting time which will allow the agents to run their learning algorithm (a couple of SGD steps, for example).

The score of a policy for a single session is the average reward in the last number of rounds, where the number of rounds is defined to using the "score_scope" variable.

2.3 Running a Session

To run a session of the game, simply run the "Snake.py" script with the desired parameters. A summary of the parameters can be accessed by typing:

```
>> python Snake.py -h
```

Important options include the game duration, score scope, the policies, the waiting times for the policies to respond to the "act" and "learn" functions, the board size and whether or not to render the game.

For example:

```
>> python Snake.py -D 10000 -P "Avoid(epsilon=0.3)" -bs "(20,60)" -plt 0.2 -pat 0.01
>> python Snake.py -D 20000 -P "Avoid(epsilon=1.0);Avoid(epsilon=0.0)" -bs="(30,30)"
>> python Snake.py -D 15000 -P "Avoid();MyPolicy(lr=0.01,eps=0.02)" -r 0
```

The first example runs a session with 10,000 rounds with one policy that avoids collisions, with a board shape of 20 by 60. The waiting times are 0.2 seconds for the "learn" function and 0.01 seconds for the "act" function.

The second example runs a session with 20,000 rounds between two policies that avoid collisions (the first one acts completely randomly while the second one always avoids collisions), with a board size of 30 by 30.

The third example runs a session with 15,000 rounds between the avoid collisions policy and a custom policy, with the game not being rendered in real time.

2.4 Additional Instructions and Features

2.4.1 Logging

Debugging in a multi-process environment is never fun, so to make things a bit easier you are supplied with a logging API which is shared between the game and the agents. The method which allows for logging is the Policy.log() function. An example of using the logging function can be seen in the implementation of the avoid collisions policy.

2.4.2 Recording

You can record a session and save it to a pickle file, so that later you can playback the games and see how your agent is learning. This can be done using the "-rt" option, which specifies the path where the recording will be saved. Looking at your agent playing can be mesmerizing, but don't spend too much time staring at the recordings...

2.4.3 Playback

If you want to playback a recorded game, you can do so using the options "-p" (path of recording) along with "-pir", "-pfr", and "-rr". For example:

```
>> python Snake.py -p MyRecording.pkl -ipr 2000 -fpr 4000 -pit 0
```

The above line runs the recording "MyRecording.pkl" from round 2000 up to round 4000 round, with 0 seconds of waiting time for the agents to initialize (since this is a playback session).

2.4.4 String Arguments

Another feature you can use is the "cast_string_args" function. This allows you to define arguments for your learner to receive from the command line when you are running the game. All you have to do is override the function and add your parameters like so:

```
Class MyPolicy(base_policy.Policy):

  def cast_string_args(self, policy_args):
    policy_args['r'] = float(policy_args['r']) if 'r' in policy_args else 0.01
    policy_args['b'] = int(policy_args['b']) if 'b' in policy_args else 512
    return policy_args
```

And start the game as follows:

```
>> python Snake.py -P "MyPolicy(r=0.01,b=32);MyPolicy(r=0.1,b=8)"
```

And you will have two instances of your policy against each other, with different learning rates and batch sizes.

Note that when you submit your code you won't be able to pass these arguments to your policy, so this is mainly for debugging and understanding what the best parameters should be. **You should hard code your parameters into your policy before submitting**.

2.4.5 Policy Member Variables

Aside from the variables that you can pass as strings when starting the game, your policy has four additional variables that you can use:

1. self.id - this is your player ID in the session. This is also your snake value on the game board, if you wish to use it to get more information for your state representation.
2. self.game_duration - this is the amount of rounds the session was started with. You may use this to schedule your parameters according to the amount of rounds you expect to play (for instance, if you want to schedule your learning rate to decrease as a function of the round).
3. self.score_scope - the number of rounds at the end of the session which will be used to calculate your score. The score will be the average reward during this number of rounds at the end of the session. You may use this to schedule your hyper parameters as well (for instance, stop exploring during this time in order to maximize your reward).
4. self.too_slow - this is a boolean variable which is True if the game didn't get an action from your policy in time for a few consecutive rounds. Assuming your "act" implementation isn't problematic, this is probably due to a learning implementation which exceeds the given time for learning (your policy is still learning while the game is waiting for new actions). **If this variable is True, you should adapt and learn/act faster** (this means lowering your batch size, for example).

2.4.6 Running Your Code on the SM Cluster

Your code will be evaluated on the sm cluster, to make the evaluation process faster and to make sure that you all get the same computational resources. This means that you need to make sure your policy is able to act and learn on time **on the sm cluster** (this could be different from your laptop).

You have been given access to the sm cluster in order to test your code's running time. To do this, modify the sbatch example script you've been given to have the correct parameters (along with paths to where the log will be saved). Please do not change the rendering option (keep the "-r 0" option). Next, to run the code on the cluster you will need to connect to it using ssh. Run the following command from any CS computer:

```
>> ssh phoenix-gw
```

You will be asked to insert your CSE password.

Once you are logged in, you should run the sbatch script in the following way (after editing the script to have your relevant parameters):

```
>> sbatch <folder_path>/sbatch_example.sh
```

This will send your requested job to the queue, where it will run. If you wish to see the status of your job, you can use the "squeue" command. To see only your jobs, you can run the following command:

```
>> squeue | grep <YOUR_USER>
```

Once your job is running, you can look at the log and out files to see if you are indeed acting and learning in the appropriate time.

If you accidentally sent too many jobs and want to cancel some of them, you can cancel them by their job ID or simply cancel all of your user's jobs in the following ways:

```
>> scancel <JOB_ID>
>> scancel -u <USER>
```

2.4.7 The Position Class

Since the board size isn't constant, the Position class is there in order to make things easier for you when it comes to accessing certain parts of the board in relation to your snake. This way, you don't need to worry about the edges of the map (where we need to take a modulo of the position) or worry about the current orientation of the head of your snake.

In the avoid policy you can find an example of how to use the Position class in a way which avoids dealing with these issues - the avoid policy looks at the positions to the right, left and in front of your snake and acts according to what's in those positions.

You can look at a position in one step in a certain direction using the "move" method of the Policy class, and you can find out what direction is relevant given your current direction by using the "TURNS" dictionary, as seen in the avoid policy example.

2.4.8 Using Different Python Packages

You can use any python library, but it's your responsibility to make sure it's installed in the environment in which the interpreter will be executed. The environment is located in: **/cs/labs/dshahaf/omribloch/env/snake/snake/.** To install packages, you can connect to the environment via the "source" command, and then install any package as you would on a regular python environment. Make sure to make all files completely public using "chmod -R 777".

```
>> source /cs/labs/dshahaf/omribloch/env/snake/snake/bin/activate.csh
(snake_env)>> pip install important_package
(snake_env)>> deactivate
>> chmod -R 777 /cs/labs/dshahaf/omribloch/env/snake/snake
```

Tensorflow, keras and sklearn are already installed. If you plan on installing something, please let the TA know in advance. To see if your code works in the virtualenv, you can just use the source command as seen above, and run your code (or run your code on the sm cluster as instructed). Please do not change the versions of existing packages in the virtual environment.

2.4.9 Restrictions You Must Work Under

Finally, there are a couple of restrictions you should uphold:

- Your policy should not exceed 0.5 GB of memory - you can test this using the sbatch example. Generally, this means you shouldn't keep a replay buffer of the entire game - you should have a maximum-size replay buffer.
- You need to act and learn in the appropriate times for the game to accept your actions. It is OK to miss an action here and there, but you should try and avoid missing several actions every minute... This probably means not using too big of a neural network in your model.

3 Evaluation

To make sure everything is fair and no one has an advantage of more training time, your policies will be trained from scratch during evaluation.

Each of your agents will be tested under different conditions.

3.1 Evaluating The Linear Agent

Your linear agent should be able to learn quickly, since the Q function is approximated by a simple, linear function.

We will run 10 sessions of your agent against two additional avoid collision policies for **5000 rounds with a score scope of 1000**. Your rewards will be averaged across the 5 sessions. Your action time will be **0.005** seconds and your learning time will be **0.01** seconds.

Your goal here isn't to beat the best avoid collisions policy, but rather to successfully learn to avoid enough collisions to get a reasonable score (a positive score, showing you eat more fruit than you hit obstacles).

3.2 Evaluating The Second Agent

Your second policy will be evaluated for a longer game duration, to allow for a more complicated learning algorithm and model. It will be tested in two different ways.

3.2.1 Evaluation Against Avoid Agents

We will run 10 sessions of your agent against 4 avoid policies, with different values of epsilon (between 0 and 0.5) for **50,000 rounds with a score scope of 5000**. Your action time will be **0.01** seconds and your learning time will be **0.05** seconds.

Your goal here is to outperform the avoid policy with the smallest possible epsilon value. Note that if you can't beat the avoid policy with an epsilon value of 0, that's OK and you shouldn't worry - you may still get a full grade for this part. However, you should definitely hope to consistently beat the avoid policy with an epsilon value of 0.1.

3.2.2 APML Tournament

The final evaluation will be in a tournament with all of the other agents handed in by your fellow students.

We will run sessions with 5 agents, with every agent participating in 10 sessions in total. Each session will have randomly chosen players, so you will not know in advance who you'll be playing against. The sessions will have the same parameters as the evaluation against the avoid agents.

Your scores will be averaged across your 10 sessions and this will give us a final tournament ranking. The results of this ranking will count towards 5% of your project grade.

4 Guidelines and Tips

4.1 Start Simple!

We ask you to hand in a linear model because reinforcement learning has many parameters and many options of things to try, and it is easy to get lost. So, we strongly recommend that you **start with the linear agent**.

The linear agent is simple and allows you to quickly test your code to see that you are actually learning. Only once you are sure that you implemented a correct Q-learning algorithm and that you agent is actually learning should you move on to implementing a more complicated model.

4.2 Choosing Parameters

There are quite a few parameters in reinforcement learning, and it can be hard to find the best set of parameters for your model... However, you should be able to find the right range for your parameters by thinking about their meaning.

4.2.1 The Discount Rate - γ

The discount rate decides how much you value immediate rewards versus rewards which come later. It is highly linked to how much you believe your learner is able to think ahead.

If, for example, your learner can only see what is in front of it (like the avoid policy), then γ should be very small since there is hardly any way of evaluating how good the next state is. Your agent shouldn't be able to plan ahead very well in this situation, and so you shouldn't put a large emphasis on future rewards in this situation either.

Generally, γ gives you a way of deciding how much you should value a reward that is n steps in the future (γ^n). If you can estimate how many steps into the future are relevant for your agent, then you can estimate a reasonable range of values for γ this way.

4.2.2 ϵ-Greedy Exploration

If you choose to use ϵ-greedy exploration in your agent, then choosing ϵ is a very important thing which will have a large effect on the performance of your model. In this case, we recommend looking into changing ϵ according to the current round, to make sure you start with a large epsilon and slowly reach a low epsilon in the final stages of the session.

4.2.3 Neural Network Hyper-parameters

If you're implementing a neural network for your second agent, we recommend keeping things relatively simple, since you won't have too much time to play around with the many possible architectures and hyper-parameters that you are able to choose from. When it comes to parameters like the learning rate, batch size and so on - we would recommend staying around the default parameters of Keras and only changing them if you feel you've settled on the right γ and ϵ.

4.2.4 Running Cross Validation

Remember that you can use the string arguments to dynamically set these parameters for your model - this will allow you to test different parameters all at once by playing 5 agents with different parameters against each other for several times and seeing which parameters perform best.

4.3 Run Small, Fast Experiments

Reinforcement learning can take quite a bit of time, which you don't have too much of. This means that your experiments should be fast - don't train your model for 100,000 rounds only to find out

it isn't learning properly... As described above, learning to avoid obstacles reasonably can be done with a simple linear model in a relatively small amount of rounds.

Also, the game recordings can be a good way to see if your policy is learning well (has it learned to always go forward or is it actually moving around and eating fruit?). But still, don't spend too much time looking at the game recordings - draw fast conclusions and keep improving your algorithm.

4.4 Keeping a History of States For Learning

Each time your "act" and "learn" functions are called, the game only gives you the new board state along with the results of the previous action. To learn from all of the actions of the game, you should keep some history (replay buffer) of this in a data structure of your choice, so that you can access it during the learning phase.

4.5 The Importance of your Board Representation

One of the most important things that you will need to decide is what is the representation that your agent will see and act upon. Giving your agent the entire board will probably be too much information and will be hard to learn from, but giving it too little information is also problematic. Here, again, it is good to start with the linear policy which is able to learn with a very simple representation, and only then move on to more complicated representations.

Also, note that the board that is given to you has values which are categorical. If a fruit has a value of 4 on the board and an obstacle has a value of 8, it doesn't mean that the obstacle is twice as good/bad as the fruit. This means that the two values probably shouldn't share the same weight, and your representation should reflect that.

4.6 Using Symmetries To Your Advantage

If you're using a relatively large state representation, you may want to think of ways of learning faster by taking advantage of certain symmetries in the game. This way, you may turn a single state-action-reward triplet into more than one triplet, and in this way you could learn faster and "see more states" without actually visiting them.

4.7 Things To Watch Out For

The design of this game has some pitfalls and possible bugs you should avoid. Here are some things that might not be obvious but that you should look out for:

- Note that the first time the "act" function will be called in a session will be with "prev_state", "prev_action" and "reward" as None. If you save these in your history it could cause problems during learning... So you should probably start saving your history from the second state onward.

- If you're working on the CS computers and it seems like Tensorflow isn't installed on them, try running the following line before running Python: "module load tensorflow".

5 Submission Instructions

You are required to submit a tar named "project_<FIRST ID>_<SECOND ID>" (if you're submitting alone, just write one ID) with the following 3 files: an Answers.pdf file, a python3 code file named "linear<ID>" and a python3 code file named "custom_policy<ID>" (you only need to write one of the IDs in the code files). **If you are submitting in a pair, please only submit from one of your users, not both**.

The first code file should contain a single class named Linear<ID> that inherits from the base_policy.Policy class. The second one should contain a single class named Custom<ID>. **Do not rely on any argument-passing for the instantiation of your policies - hard code your parameters**. All policies will be placed in a folder and will be loaded as in the Snake.py code you are provided with. As usual, please write readable, well documented code.

Your Answers.pdf should contain the following:

- Both of your IDs and names
- An explanation of how you chose to represent the board state and why
- An explanation of the custom model you are learning and the reasons behind your choice of model
- An explanation of the learning algorithm you chose - which one did you use and why?
- An explanation of how you chose to handle the exploration-exploitation trade-off
- A short description of the tests and results you got when you trained your policy
- A detailing of other possible solutions that you tried and decided not to hand in and why (if there were any)
- Any other things you would like to note about your implementation

The Answers.pdf has implications on your grade so please make an effort to present your solution clearly.

Good Luck!