ECE2013 Fall 2022

Assignment 5: Reinforcement Learning and Deep Learning

# Contents

Part 1: Q-learning (Snake)

Snake is a famous video game originated in the 1976 arcade game Blockade. The player uses up, down, left and right to control the snake which grows in length (when it eats the food), with the snake body and walls around the environment being the primary obstacle. In this assignment, you will train AI agents using reinforcement learning to play a simple version of the game snake. You will implement a TD version of the Q-learning algorithm.

Provided Snake Environment
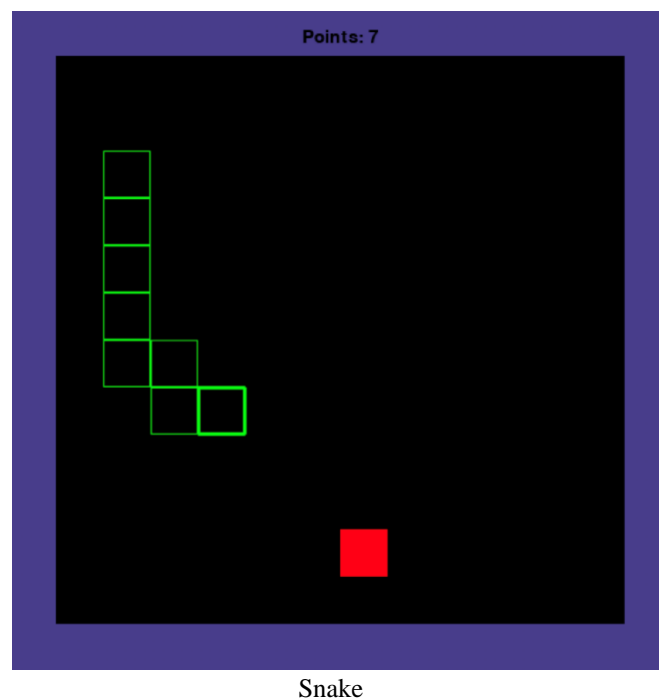
Snake

In this assignment, the size of the entire game board is 560x560. The green rectangle is the snake agent and the red rectangle is the food. Snake head is marked with a thicker boarder for easier recognition. Food is generated randomly on board once the initial food is eaten. The size for every side of wall

(filled with blue) is **40**. The snake head, body segment and food have the same size of 40x40. Snake moves with a speed of 40 per frame. In this setup, the entire board that our snake agent can move has a size of 480x480 and can be treated as a 12x12 grid. Every time it eats a food, the points increases 1 and its body grows one segment. Before implementing the Q-learning algorithm, we must first define Snake as a Markov Decision Process (MDP).

Note in Q-learning, state variables do not need to represent the whole board, it only needs to represent enough information to let the agent make decisions. (So once you get environment state, you need to convert it to the state space as defined below). Also, the smaller the state space, the more quickly the agent will be able to explore it all.

- **State:** A tuple (adjoining_wall_x, adjoining_wall_y, food_dir_x, food_dir_y, adjoining_body_top, adjoining_body_bottom, adjoining_body_left, adjoining_body_right).

  o **[adjoining_wall_x, adjoining_wall_y]** gives whether there is wall next to snake head. It has 9 states:
  adjoining_wall_x: **0** (no adjoining wall on x axis), **1** (wall on snake head left), **2** (wall on snake head right)
  adjoining_wall_y: **0** (no adjoining wall on y axis), **1** (wall on snake head top), **2** (wall on snake head bottom)
  (Note that [0, 0] is also the case when snake runs out of the 480x480 board)

  o **[food_dir_x, food_dir_y]** gives the direction of food to snake head. It has 9 states: **food_dir_x**: **0** (same coords on x axis), **1** (food on snake head left), **2** (food on snake head right) **food_dir_y**: **0** (same coords on y axis), **1** (food on snake head top), **2** (food on snake head bottom)

  o **[adjoining_body_top,   adjoining_body_bottom,   adjoining_body_left, adjoining_body_right]** checks if there is snake body in adjoining square of snake head. It has 8 states:
  adjoining_body_top: **1** (adjoining top square has snake body), **0** (otherwise)
  adjoining_body_bottom: **1** (adjoining bottom square has snake body), **0** (otherwise)
  adjoining_body_left: **1** (adjoining left square has snake body), **0** (otherwise)
  adjoining_body_right: **1** (adjoining right square has snake body), **0** (otherwise)

- **Actions:** Your agent's actions are chosen from the set {up, down, left, right}.
- **Rewards:** +1 when your action results in getting the food (snake head position is the same as the food position), -1 when the snake dies, that is when snake head hits the wall, its body segment or the head tries to move towards its adjacent body segment (moving backwards). -0.1 otherwise (does not die nor get food).

Q-learning Agent

In this part of the assignment, you will create a snake agent to learn how to get food as many as possible without dying. In order to do this, you must use Q-learning. Implement the TD Q-learning algorithm and train it on the MDP outlined above.

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Also, use the exploration policy mentioned in class and use **1** for R$^+$:

$$a = \underset{a' \in A(s)}{\arg\max} \, f\left(Q(s,a'), N(s,a')\right)$$

exploration   Number of times we've
function   taken action a' in state s

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \text{ (optimistic reward estimate)} \\ u & \text{otherwise} \end{cases}$$

During training, your agent needs to update your Q-table first (this step is skipped when the initial state and action are None), get the next action using the above exploration policy, and then update N-table with that action. If the game is over, that is when the dead variable becomes true, you only need to update your Q table and reset the game. During testing, your agent only needs to give the best action using Q-table. Train it for as long as you deem necessary, counting the average number of points your agent can get. Your average over 1000 test games should be at least 20.

For grading purposes, please submit code with the above exploration policy, state configurations and reward model. We will initialize your agent class with different parameters (Ne, C, gamma), initialize environment with different initial snake and food position and compare the Q-table result at the point when the first food is eaten during training (see snake_main.py for Q-table generation detail).

Once you have this working, you will need to adjust the learning rate, α (how about a fixed learning rate or other C value?), the discount factor, γ, and the settings that you use to trade off exploration vs. exploitation.

In your report, please include the values of α, γ, and any parameters for your exploration settings that you used and discuss how you obtained these values. What changes happen in the game when you adjust any of these variables? How many games does your agent need to simulate before it learns an optimal policy? After your Q-learning seems to have converged to a good policy, run your algorithm on a large number of test games (≥1000) and report the average number of points.

In addition to discussing these things, try adjusting the state configurations that were defined above. If you think it would be beneficial, you may also change the reward model to provide more informative feedback to the agent. Try to find modifications that allow the agent to learn a better policy than the one you found before. In your report, describe the changes you made and the new number of points the agent was able to get. What effect did this have on the time it takes to train your agent? Include any other interesting observations.

Tips

- To get a better understanding of the Q learning algorithm, read section 21.3 of the textbook.
- Initially, all the Q value estimates should be 0.
- The learning rate should decay as $C/(C+N(s,a))$, where $N(s,a)$ is the number of times you have seen the given the state-action pair.
- When adjusting state configurations, try to make state numbers as small as possible to make the training easier. If the state numbers are too large. The snake may be stuck in an infinite loop.
- In a reasonable implementation, you should see your average points increase in seconds.

- You can run python snake_main.py --human to play the game yourself.

Debug Convenience

For debug convenience, we provide three debug examples of Q-table for you. **Each Q-table is generated exactly after snake eats the first food in training process. More specifically, it's the first time when snake reaches exactly 1 point in training**, see how the Q-table is generated and saved during training in **snake_main.py**. For example, you can run diff checkpoint.npy checkpoint1.npy to see whether there is a difference. The only difference of these three debug examples is the setting of parameters (initialized position of snake head and food, Ne, C and gamma).
Notice that if the scores of actions from exploration function are equal, the priority should be right > left > down > up.

- [Debug Example 1] snake_head_x=200, snake_head_y=200, food_x=80, food_y=80, Ne=40, C=40, gamma=0.7 checkpoint1.npy
- [Debug Example 2] snake_head_x=200, snake_head_y=200, food_x=80, food_y=80, Ne=20, C=60, gamma=0.5 checkpoint2.npy
- [Debug Example 3] snake_head_x=80, snake_head_y=80, food_x=200, food_y=200, Ne=40, C=40, gamma=0.7 checkpoint3.npy

Making sure you can pass these debug examples will help you a lot. In addition, **Average points over 20 points on 1000 test games** should be able to obtain full credit for this part.

Part 2: Classical Neural Network

In this part, you will be using the PyTorch and NumPy libraries to implement neural net. The PyTorch library will do most of the heavy lifting for you, but it is still up to you to implement the right high-level instructions to train the model.

The basic neural network model consists of a sequence of hidden layers sandwiched by an input and output layer. Input is fed into it from the input layer and the data is passed through the hidden layers and out to the output layer. Induced by every neural network is a function F_W which is given by propagating the data through the layers.

To make things more precise, in lecture you learned of a function $f_w(x) = \sum_{i=1}^{n} w_i x_i + b$. In this assignment, given weight matrices W_1, W_2 with $W_1 \in R^{h \times d}$, $W_2 \in R^{h \times 2}$ and bias vectors $b_1 \in R^h$ and $b_2 \in R^2$, you will learn a function F_W defined as

$$F_W(x) = W_2 \sigma(W_1 x + b_1) + b_2$$

where σ is your activation function. In part 2, you should use either of the sigmoid or ReLU activation functions. You will use 32 hidden units (h=32) and 3072 input units, one for each channel of each pixel in an image ($d = (32)^2(3) = 3072$).

Dataset

The dataset consists of 10000 32x32 colored images (a subset of the CIFAR-10 dataset, provided by Alex Krizhevsky), split for you into 7500 training examples (of which 2999 are negative and

4501 are positive) and 2500 development examples.

The data set is included within the coding template. When you uncompress this you'll find a binary object that our reader code will unpack for you.

Training and Development

- Training: To train the neural network you are going to need to minimize the empirical risk R(W)R(W) which is defined as the mean loss determined by some loss function. For this assignment you can use cross entropy for that loss function. In the case of binary classification, the empirical risk is given by

$$\mathcal{R}(W) = \frac{1}{n}\sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i)\log (1 - \hat{y}_i)$$

where y_i are the labels and $\hat{y}_i$ are determined by $\hat{y}_i = \sigma\big(F_W(x_i)\big)$ where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. For this assignment, you won't have to implement these functions yourself; you can use the built-in PyTorch functions.

Notice that because PyTorch's **CrossEntropyLoss** incorporates a sigmoid function, you do not need to explicitly include an activation function in the last layer of your network.

- **Development:** After you have trained your neural network model, you will have your model decide whether or not images in the development set contain animals in them. This is done by evaluating your network F_W on each example in the development set, and then taking the index of the maximum of the two outputs (argmax).
- **Data Standardization:** Convergence speed and accuracies can be improved greatly by simply centralizing your input data by subtracting the sample mean and dividing by the sample standard deviation. More precisely, you can alter your data matrix X by simply setting $X := (X - \mu)/\sigma$.

With the aforementioned model design and tips, you should expect around 0.84 dev-set accuracy.

Extra credit: CIFAR-100 superclasses

For an extra 10% worth of the points on this MP, your task will be to pick any *two* superclasses from the CIFAR-100 dataset (described in the same place as CIFAR-10) and rework your neural net from part 2, if necessary, to distinguish between those two superclasses. A superclass contains 2500 training images and 500 testing images, so between two superclasses you will be working with 3/5 the amount of data in total (6000 total images here versus 10000 total in the main MP).

To begin working on the extra credit, we recommend that you make a copy of the entire directory containing your solution to the second part of the main MP. Then replace the data directory and the file **reader.py**, and modify your file **neuralnet.py**, as described in the next two paragraphs.

- Replace CIFAR-10 with CIFAR-100, and download the new reader

You can download the CIFAR-100 data here and extract it to the same place where you've placed the data for the main MP. A custom reader for it is provided here; to use it with the CIFAR-100 data, you should rename this to **reader.py** and replace the existing file of that name in your working directory.

- Modify **neuralnet.py** in order to choose the two CIFAR-100 classes you want to classify

Define two global variables *class1* and *class2* at the top level of the file **neuralnet.py** (that is, outside of the NeuralNet class). Set the values of these variables to integers, in order to choose the two classes that

you want to classify for extra credit. The order of the superclasses listed on the CIFAR description page hints at the index for each superclass; for example, "aquatic mammals" is 0 and "vehicles 2" is 19.

- Now that you have the new classification task, the first thing you should do is to try running the same code that you used in part 2 of the regular MP, to re-train your neural net on these new data, then test it to see how well it performs. This can be your baseline; your goal for extra credit will be to find new algorithms that give you better accuracy. Your new algorithm can be anything you like (a network with more layers, or with more nodes per hidden layer, or a convolutional neural network, or any other algorithm of your choice). Note that your revised neural net must still have fewer than 500,000 total parameters.

Provided Code Skeleton

We have provided skeleton.zip with the descriptions below. For Part 1, **do not import any non-standard libraries except pygame (pygame version 1.9.4) and numpy**.

Part 1

- **snake.py** - This is the file that defines the snake environment and creates the GUI for the game.
- **utils.py** - This is the file that defines some of the discretization constants as defined above and contains the functions to save and load models.
- **agent.py** This is the file where you will be doing all of your work. This file contains the Agent class. This is the agent you will implement to act in the snake environment. Below is the list of instance variables and functions in the Agent class.
    - **self._train:** This is a boolean flag variable that you should use to determine if the agent is in train or test mode. In train mode, the agent should explore (based on exploration function) and exploit based on the Q table. In test mode, the agent should purely exploit and always take the best action.
    - **train():** This function sets the **self._train** to be True. This is called before the training loop is run in **snake_main.py**
    - **test():** This function sets the **self._train** to be False. This is called before the testing loop is run in **snake_main.py**
    - **save_model():** This function saves the **self.Q** table. This is called after the training loop in **snake_main.py**.
    - **load_model():** This function loads the **self.Q** table. This is called before the testing loop in **snake_main.py**.
    - **act(state, points, dead):** This is the main function you will implement and is called repeatedly by **snake_main.py** while games are being run. "state" is the state from the snake environment and is a list of [snake_head_x, snake_head_y, snake_body, food_x, food_y] **(Notice that in act function, you first need to discretize this into the state configuration we defined above)**. "points" is the number of foods the snake has eaten. "dead" is a boolean indicating if the snake is dead. "points", "dead" should be used to define your reward function. **act** should return a number from the set of {0,1,2,3}. Returning 0 will move the snake agent up, returning 1 will move the snake agent down, and returning 2 will move the agent left, returning 3 will move the agent right. If **self._train** is True, this function should update the Q table and return an action(**Notice that if the scores of actions from exploration function are equal, the priority should be right > left > down > up**). If **self._train** is False, the agent should simply return the best action based on the Q table.

- **snake_main.py** - This is the main file that starts the program. This file runs the snake game with your implemented agent acting in it. The code runs a number of training games, then a number of testing games, and then displays example games at the end.

You will only have to modify agent.py.

Part 2

- **reader.py** - This file is responsible for reading in the data set. It creates a giant NumPy array of feature vectors corresponding to each image.
- **mp5_part2.py** - This is the main file that starts the program, and computes the accuracy, precision, recall, and F1-score using your implementation
- **neuralnet.py** is file where you will be doing all of your work for part 2. You are given a **NeuralNet** class which implements a **torch.nn.module**. This class consists of **__init__()**, **forward()**, and **step()** functions. (Beyond the important details below, more on what each of these methods in the NeuralNet class should do is given in the skeleton code.)
  - ○ **__init__()** is where you will need to construct the network architecture. There are multiple ways to do this.
    - One way is to use the **Linear** and **Sequential** objects. Keep in mind that `Linear` uses a Kaiming He uniform initialization to initialize the weight matrices and sets the bias terms to all zeros.
    - Another way you could do things is by explicitly defining weight matrices W_1, W_2, ... and bias terms b_1, b_2, ... by defining them as **Tensors**. This approach is more hands on and will allow you to choose your own initialization. For this assignment, however, Kaiming He uniform initialization should suffice and should be a good choice.

    Additionally, you can initialize an <u>optimizer</u> object in this function to use to optimize your network in the `step()` function.
  - ○ **forward()** should perform a forward pass through your network. This means it should explicitly evaluate $F_W(x)$. This can be done by simply calling your Sequential object defined in `__init__()` or (if you opted to define tensors explicitly) by multiplying through the weight matrices with your data.
  - ○ **step()** should perform one iteration of training. This means it should perform one gradient update through one batch of training data (not the entire set of training data). You can do this by either calling `loss_fn(yhat, y).backward()` then updating the weights directly yourself, or you can use an optimizer object that you may have initialized in `__init__()` to help you update the network. Be sure to call `zero_grad()` on your optimizer in order to clear the gradient buffer. When you return the loss_value from this function, make sure you return `loss_value.item()` (which works if it is just a single number) or `loss_value.detach().cpu().numpy()` (which separates the loss value from the computations that led up to it, moves it to the CPU—important if you decide to work locally on a GPU, bearing in mind that Gradescope won't be configured with a GPU—and then converts it to a NumPy array). This allows proper garbage collection to take place (lest your program possibly exceed the memory limits fixed on Gradescope).
  - ○ **fit()** takes as input the training data, training labels, development set, and the

maximum number of iterations. The training data provided is the output from **reader.py**. The training labels is a Tensor consisting of labels corresponding to each image in the training data. The development set is the Tensor of images that you are going to test your implementation on. The maximum number of iterations is the number you specified with `--max_iter` (it is 500 by default). `fit()` outputs the predicted labels. It should construct a NeuralNet object, and iteratively call the neural net's `step()` to train the network. This should be done by feeding in batches of data determined by batch size. You will use a batch size of 100 for this assignment. *max_iter* is the number of batches (not the number of epochs!) in your training process.

- **mp5_data** is the file of data set. See reader.py and mp5_part2.py for how to load and process the data.

To learn more about how to run the MP, run `python3 mp5_part2.py -h` in your terminal.

You should definitely use the PyTorch documentation, linked multiple times on this page, to help you with implementation details. You can also use [this PyTorch Tutorial](#) as a reference to help you with your implementation. There are also other guides out there such as [this one](#).

Deliverables

This MP will be submitted via Bb. Please upload only the following files to Bb.

- **agent.py** with the same exploration policy, state configurations and reward model mentioned above.
- **q_agent.npy** the best numpy array trained by you with the same state configurations mentioned above. (Can be saved by passing "--model_name q_agent.npy" to snake_main.py). **Note that this model above should work without modifying any code files other than agent.py.**
- **neuralnet.py** for part 2.
- **report.pdf**

Report Checklist
Part 1

1. Briefly describe the implementation of your agent snake.
   o How does the agent act during train phase?
   o How does the agent act during test phase?
2. Use Ne, C (or fixed alpha?), gamma that you believe to be the best. After training has converged, run your algorithm on 1000 test games and report the average point.
   o Give the value of Ne, C (or fixed alpha) you believe to be the best.
   o Report the training convergence time.
   o Report average point on 1000 test games.
3. Describe the changes you made to your MDP (state configuration, exploration policy and reward model), **at least make changes to state configuration**. Report the performance (the average points on 1000 test games). Notice that training your modified state space should give you at least 10 points in average for 1000 test games. Explain why these changes are reasonable, observe how snake acts after changes and analyze the positive and negative

effects they have. **Notice again, make sure your submitted agent.py and q_agent.npy are without these changes and your changed MDP should not be submitted.**

Part 2

1. Report the Average Classification Rate, Precision, Recall, and F1-score for your implementation.
2. Add a graph that plots epochs vs losses at that epoch.
3. Describe any trends that you see. Is this expected or surprising? What do you think is the explanation for the observations?
4. Report Extra Credit section, if any.