Assignment: Reinforcement Learning and Deep Learning

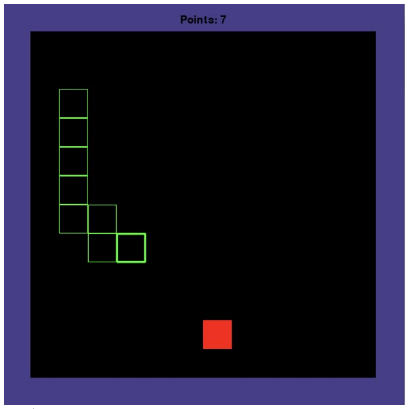
Contents

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.

Image from Wikipedia

Provided Snake Environment



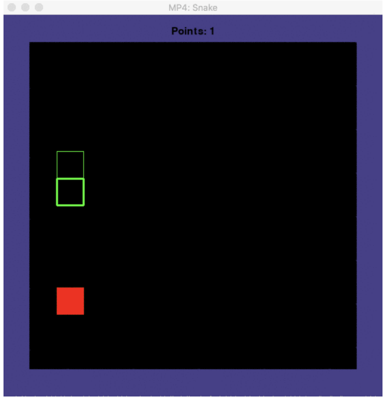
Snake

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).
    - **[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)
    - **[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)
    - **[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



TrainedAgent

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^+:

$$a = \underset{a' \in A(s)}{\arg\max}\, \underset{\substack{\uparrow \\ \text{exploration} \\ \text{function}}}{f}\left(Q(s,a'), \underset{\substack{\uparrow \\ \text{Number of times we've} \\ \text{taken action a' in state s}}}{N(s,a')}\right)$$

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

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 varaible 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 postion 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

- 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 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 for passing the autograder, 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

Note that for one part of the autograder, we will run your training process on different settings of parameters and compare the Q-table generated exactly when snake reaches 1 point first time in training, with ours. So making sure you can pass these debug examples will help you a lot for passing this part of the autograder.

In addition, **for the other part of autograder, we will test your performance using your q_agent.npy and agent.py. Average points over 20 points on 1000 test games** should be able to obatin full credit for this part.

Part 2: Deep Learning (MNIST Fashion)

Created by Austin Bae (Modified from Ryley Higa)

Background

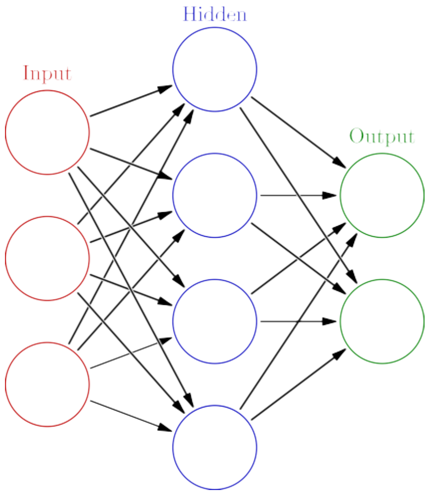
By now you should be familiar with the MNIST dataset from MP3. In MP3, we trained a model on this dataset using Linear Classifiers such as Naive Bayes and Perceptron. This time around, we will implement a fully connected neural network from scratch on the same dataset we used in MP3.

Your task is to build a 4-layer neural network with 256 hidden nodes per layer except the last layer which should have 10 (number of classes) layers. You are going to use a method called **Minibatch Gradient Descent**, which runs for a given number of iterations (epochs) and does the following per epoch:

1. Shuffle the training data
2. Split the data into batches (use batch size of 200)
3. For each batch (subset of data):
    - feed batch into the 4-layer neural network
    - Compute loss and update weights
4. Observe the total loss and go to next iteration

The Neural Network

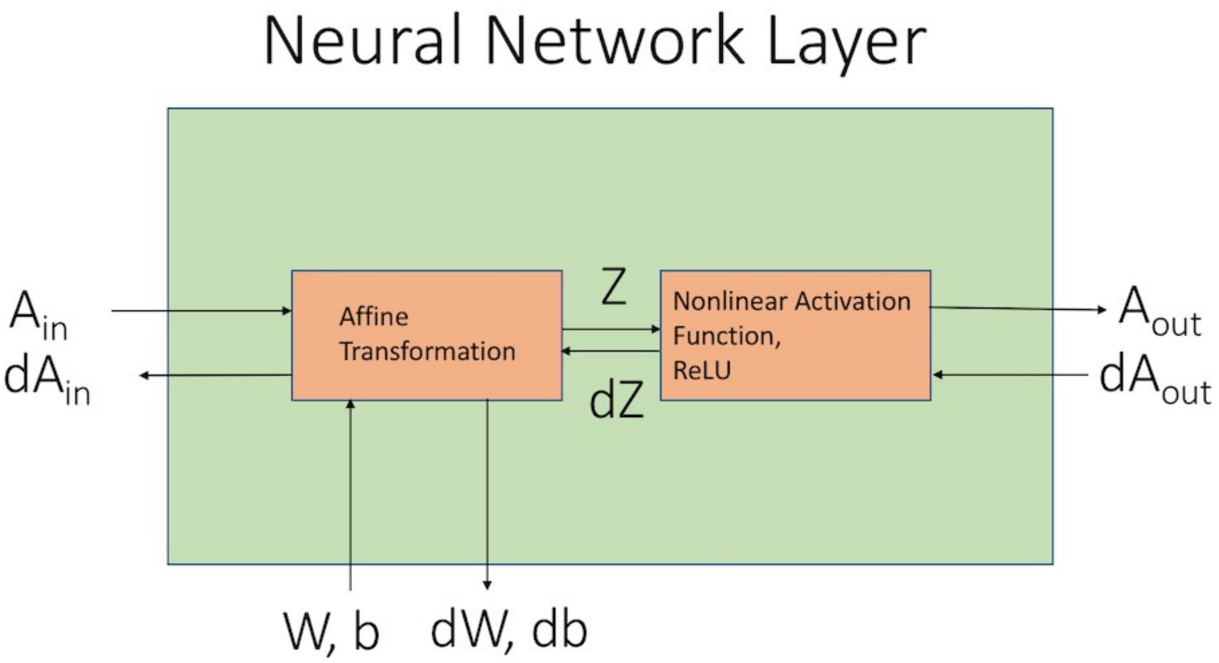
Neural Network Architecture



One Layer Neural Network

For this assignment, you will build a 4-layer, fully connected Neural Network. You can think about a fully-connected Neural Network as a series of multiple interconnected layers, in which are basically a group of nodes (See the diagram above). You can think of each layer as the perceptron model you implemented in MP3, but instead of one perceptron, you have multiple perceptrons that feed the output as the input to another.

Neural Network Layer



This is what each layer in a Neural Network looks like. Note that there are two directions: forward and backward propagation. **Forward Propagation** uses both the inputs (Ain) and the weights (W, b) specific to that layer to compute the output (Aout) to the next layer. **Backward Propagation** computes the gradient (derivative) of the loss function with respect to the weights at each layer.

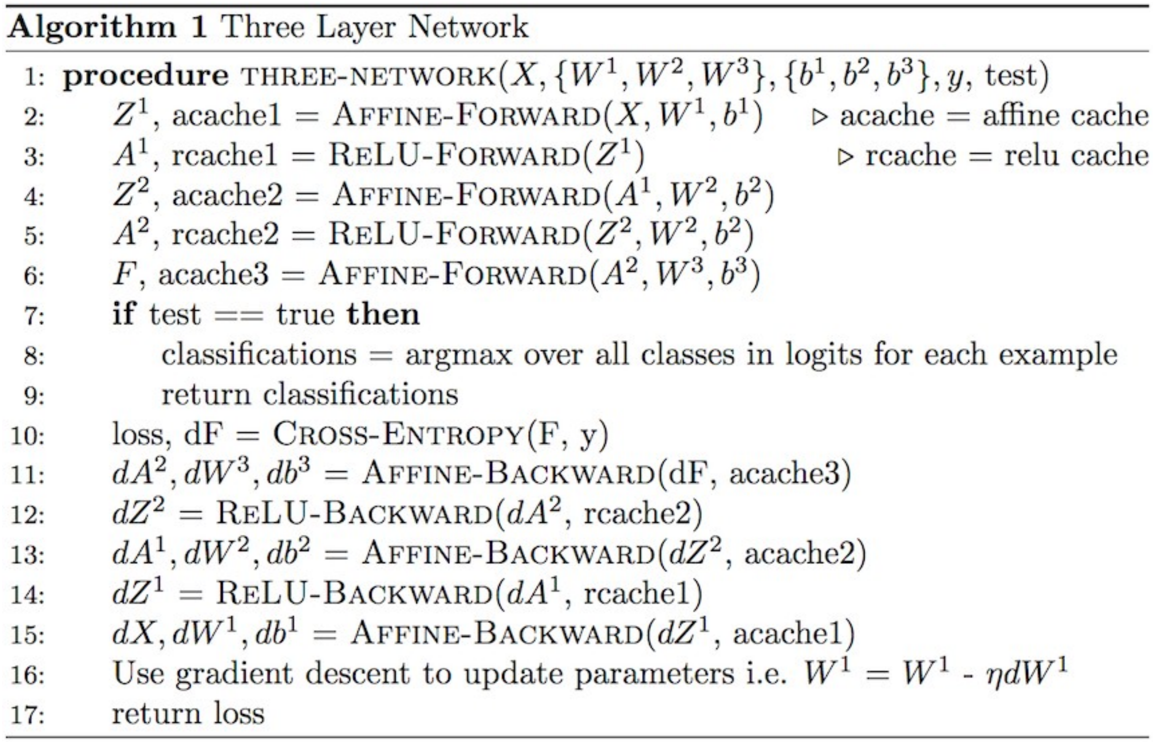
Inside of each propagation, there are two functions that both compute forward and backward. In general, Affine transformations compute the affine output (Z) by doing some sort of computation with the input and weights (like matrix multiply). Nonlinear Activation functions then take that affine output and transform it. There are numerous activation functions out there, but for our assignment we will be using **ReLU (Rectified Linear Units)** which is just simply R(x) = max(0, x). The backwards functions will compute the gradients of the inputs with respect to loss.

Then, at the last layer, the output should be the classification computed by your Neural Network. You have to then calculate the **loss**, which would represent how well your network did in classifying correctly. The model's job is to minimize this loss, since the better it does, the lower loss it will have. Again, we can use many different types of loss functions, but we will use **Cross-entropy** for this assignment.

Implementation Details

You are to implement these 8 different functions inside of `neural_network.py`. **You will only have to edit and submit this file. Do not import any non-standard Python libraries other than numpy, or you will result in getting a 0 on the autograder.**

- `affine_forward(A, W, b)`
    - Inputs: A (data with size n,d), W (weights with size d,d'), b (bias with size d')
    - $$Z_{ij} = \sum_{k=0}^{d-1} A_{ik} W_{kj} + b_j$$
    - Outputs: Z (affine output with size n,d'), cache (tuple of the original inputs)
- `affine_backward(dZ, cache)`
    - Inputs: dZ (gradient of Z), cache (of the forward operation)
    - $$dA_{ik} = \sum_{j=0}^{d'-1} dZ_{ij} W_{kj} \quad dW_{kj} = \sum_{i=0}^{n-1} A_{ik} dZ_{ij} \quad db_j = \sum_{i=0}^{n-1}$$
    - Outputs: dA, dW, db (gradients with respect to loss)
- `relu_forward(Z)`
    - Inputs: Z (affine output with size n,d')
    - R(Z) = max(0,Z) : basically sets all negative values in matrix to 0
    - Outputs: A (Relu Output with size n,d'), cache object (Z)
- `relu_backward(dA, cache)`
    - Inputs: dA (gradient of A), cache
    - $dZ_{ij} = 0$ if $Z_{ij} = 0$. else $dZ_{ij} = dA_{ij}$. Basically if Z was zeroed out at a point, then dZ should also be zeroed out since it shouldn't contribute to the gradient at that location.
    - Outputs: dZ (gradient of Z)
- `cross_entropy(F, y)`
    - Inputs: F (logits with size n, num_classes), y (actual class label of data with size n)
    - $$L = -\frac{1}{n}\left(\sum_{i=0}^{n-1} F_{iy_i} - \log\left(\sum_{k=0}^{C-1} \exp(F_{ik})\right)\right) \quad dF_{ij} = \frac{\partial L}{\partial F_{ij}} = -\frac{1}{n}\left(\mathbb{1}\{j = y_i\} - \frac{\exp(F_{ij})}{\sum_{k=0}^{C-1} \exp(F_{ik})}\right)$$
    - F_{ik} refers to the score of classifying row i as class k. F_{iy_i} refers to the score of F classifying row i as the actual class given by y_i. So if the actual label for row i was 7, then $F_{iy_i} = F_{i7}$. The `1{j = yi}` function in the gradient calculation is just a binary function that outputs either a 0 or a 1 if the condition is met.
    - Output: loss, dF (gradient of the logits)
- `four_nn()` - 4 layer neural network function
    - This function's inputs and outputs are up to you, and it won't be autograded
    - The Neural Network must have 4 layers, with (256, 256, 256, and num_classes) nodes per layers
    - In this function, you should use all of your helper functions above
    - This should be called inside of both `minibatch_gd()` and `test_nn()`
    - Here is the pseudocode for a 3 layer neural network, as reference.

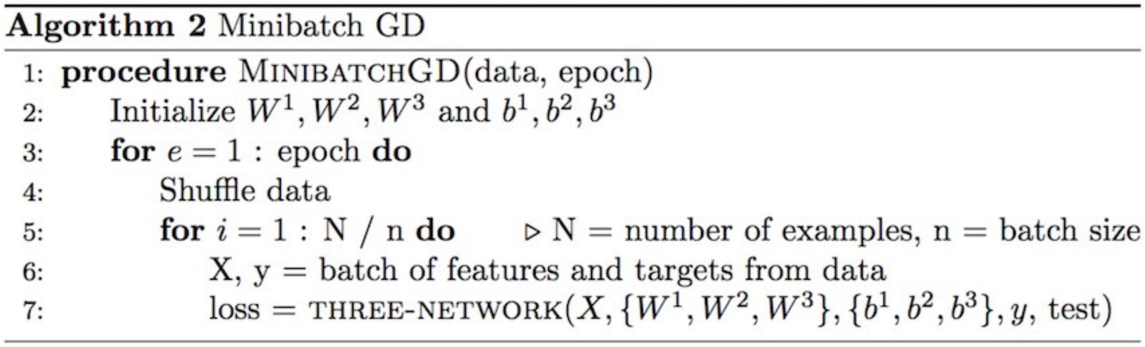


Algorithm 1 Three Layer Network

```
procedure THREE-NETWORK($X, \{W^1, W^2, W^3\}, \{b^1, b^2, b^3\}, y$, test)
    $Z^1$, acache1 = AFFINE-FORWARD($X, W^1, b^1$)    ▷ acache = affine cache
    $A^1$, rcache1 = RELU-FORWARD($Z^1$)    ▷ rcache = relu cache
    $Z^2$, acache2 = AFFINE-FORWARD($A^1, W^2, b^2$)
    $A^2$, rcache2 = RELU-FORWARD($Z^2, W^2, b^2$)
    $F$, acache3 = AFFINE-FORWARD($A^2, W^3, b^3$)
    if test == true then
        classifications = argmax over all classes in logits for each example
        return classifications
    loss, dF = CROSS-ENTROPY(F, y)
    $dA^2, dW^3, db^3$ = AFFINE-BACKWARD(dF, acache3)
    $dZ^2$ = RELU-BACKWARD($dA^2$, rcache2)
    $dA^1, dW^2, db^2$ = AFFINE-BACKWARD($dZ^2$, acache2)
    $dZ^1$ = RELU-BACKWARD($dA^1$, rcache1)
    $dX, dW^1, db^1$ = AFFINE-BACKWARD($dZ^1$, acache1)
    Use gradient descent to update parameters i.e. $W^1 = W^1 - \eta dW^1$
    return loss
```

- **You should use a learning rate (eta) of 0.1**

- `minibatch_gd(epoch, w1, w2, w3, w4, b1, b2, b3, b4, x_train, y_train, num_classes, shuffle=True)`
  - This function will implement your minibatch gradient descent (model training).
  - **Use batch size of 200.** You can assume that len(x_train) will always be divisible by 200.
  - Inputs:
    - epoch: number of iterations
    - w1, w2, w3, w4: Your weights corresponding to each of the layers
    - b1, b2, b3, b3: Your biases corresponding to each of the layers
    - x_train, y_train: Numpy arrays of the features and labels
    - num_classes: number of classes. This should be 10 for our dataset. Reason we pass this as a parameter is that your model **should be able to run for datasets of any size.**
    - shuffle: boolean flag indicating whether you should shuffle your data at each epoch. By default this is set to True. This will be set to false during testing/autograding, so adjust your code so that it will shuffle only if this boolean flag is set to True.
  - Here is a pseudocode for reference:



Algorithm 2 Minibatch GD

```
procedure MINIBATCHGD(data, epoch)
    Initialize $W^1, W^2, W^3$ and $b^1, b^2, b^3$
    for $e = 1$ : epoch do
        Shuffle data
        for $i = 1$ : N / n do    ▷ N = number of examples, n = batch size
            X, y = batch of features and targets from data
            loss = THREE-NETWORK($X, \{W^1, W^2, W^3\}, \{b^1, b^2, b^3\}, y$, test)
```

  - Outputs:
    - all the modified weights, biases (w1 ~ w4, b1 ~ b4)
    - losses: a list of total loss at each epoch. Note that this will be a list with length = epoch
- `test_nn(w1, w2, w3, w4, b1, b2, b3, b4, x_test, y_test, num_classes)`
  - This function will evaluate how well your trained model performs in classifying test data.
  - Inputs:
    - w1 ~ w4, b1 ~ b4: trained weights/biases at each layer
    - x_test, y_test: Numpy arrays of the test features and labels
    - num_classes: Number of classes (10)
  - Outputs: Average Classification Rate, Average Classification Rate per Class (List with size=num_classes)

Testing

We have provided you with a unit testing script `nn_test.py` that will check if your individual functions produce the right output. Passing these unit tests are a good but **incomplete** measure of whether or not your functions are correct. This test script also checks if your minibatch gradient descent works for a smaller dataset, so you should confirm that you pass these tests before running `nn_main.py` on the actual MNIST data. Feel free to modify this file as you need, this file won't be submitted nor graded.

What to Report

Run your minibatch gradient for 10, 30, and 50 epochs. You should start with clean generated weights/biases for each run. For each run, report:

1. Confusion Matrix
2. Average Classification Rate
3. Runtime of your minibatch gradient function

At the end of 50 epochs, include a graph that plots epochs vs losses.
Also report any interesting observations, and possible explanations for those observations.

Extra Credit Opportunity

Convolutional Neural Network (CNN) is a different type of Neural Network that is very popular, especially in image processing. Implement a CNN using the same MNIST fashion dataset using any deep learning frameworks of your choice (Tensorflow, Pytorch, Keras, etc). You may reference online materials, but make sure to **cite them** as necessary!

Write a paragraph on how CNNs work, what you implemented, and the results of your CNN. Compare it to theperformance of your fully-connected 4 layer network. Report the confusion matrix, average classification rate, and average classification rate per class of the test data once your model converges. The extra credit will be capped to 10% of the entire MP grade.

Submit your file as nn_extracredit.ext, ext being the extension of the file. You do not have to submit any runtime information (modules/metadata) but make sure to describe your algorithm in the report.

Warning/Hints

One disadvantage of Neural Networks is that they may take a long time to compute. Given that fact, correctness of your algorithms may not be sufficient to successfully complete this assignment. Use numpy functions (matrix multiply, broadcasting) as much as possible as opposed to native python loops, as this will significantly decrease your runtime.

With decently optimized code, we were able to get under 10 seconds per epoch on a 2015 Macbook Pro, and roughly under 80 seconds on EWS. We highly recommend you running the code on a machine faster than EWS, but EWS should give you the highest-bound on computation time. We will cut off computation for autograding at around 120 seconds per epoch on EWS. Also, running your model for 50 epochs will take somewhere around 10 mins ~ 1 hour. If you do this last minute you may not be able to run your computations on time. There will be no excuses for missing the deadline.

We will also grade your code on a different dataset that will have different number of rows (but divisible by 200), features, and target classes. So it is very important that you don't hardcode any numbers into your algorithms, and it should be able to support data of any size. Provided hyperparameters (number of layers, number of nodes per layer, learning rate, batch size) should remain the same.

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**. For Part 2, **do not import any non-standard library except numpy**. Failure to do so **will** result in a 0 in the autograder.

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 food 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.

Do not modify the provided code. You will only have to modify agent.py

Part 2

- **neural_network.py** - The only file that you will have to modify. Descriptions of the functions are in the descriptions above.
- **nn_test.py** - Unit test module for you to use. Simply run it using `python nn_test.py`.
- **nn_main.py** - Main function that will run your code written in nueral_network.py.
- **data** - folder that contains the train and test numpy files, exactly the same from MP3
- **tests** - You don't have to touch this since all the text parsing has been done for you in nn_test.py

You can modify the test/main functions as long as your inputs/outputs for all functions inside of neural_network.py is consistent with the instructions. Note that only neural_network.py will be submitted & graded.

Deliverables

Please submit only the following files.

- **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.**
- **neural_network.py** with all the functions for part 2.
- **report.pdf**

Report Checklist

Part 1

1. Briefly describe the
   - implementation of your agent snake. How does the agent act during train phase?
   - 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.
   - Give the value of Ne, C (or fixed alpha) you believe to be the best.
   - Report the training convergence time.
   - 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. Briefly describe any optimizations you performed on your functions for faster runtime.
2. Report Confusion Matrix, Average Classification Rate, and Runtime of Minibatch gradient for 10, 30, 50 epochs. This means that you should have **3x3=9** total items in this section.
3. Add a graph that plots epochs vs losses at that epoch. (For 50 epochs)
4. Describe any trends that you see. Is this expected or surprising? What do you think is the explanation for the observations?
5. Report Extra Credit section, if any.