# CIS 520, Machine Learning, Fall 2020
# Homework 8
Due: Monday, December 7th, 11:59pm

Submit to Gradescope

**Instructions.** Please write up your responses to the following problems clearly and concisely. We require you to write up your responses using LaTeX; we have provided a LaTeX template, available on Canvas, to make this easier. **Submit your answers in PDF form to Gradescope. We will not accept paper copies of the homework.**

**Collaboration.** You are allowed and encouraged to work together. You may discuss the **written homework** to understand the problem and reach a solution in groups. However, **it is recommended that each student also write down the solution independently and without referring to written notes from the joint session.** You must understand the solution well enough to reconstruct it by yourself. (This is for your own benefit: you have to take the exams alone.)

## Learning Objectives

After completing this assignment, you will be able to:

- Derive optimal behavior policies using Markov Decision Processes
- Understand and implement Q-learning
- Understand the relationship between Autoencoders and PCA

### Deliverables

This homework can be completed **individually or in group of 2**. You need to make one submission per group. Make sure to add your team member's name on Gradescope when submitting the homeword's written and coding parts.

1. **A PDF compilation of `hw8_template.tex` with your solutions**
2. **A `hw8.ipynb` with the functions implemented**

## 1 Reinforcement Learning: MDPs [35 points]

Recall that in a reinforcement learning problem, an agent tries to learn an optimal behavior policy by interacting with an environment. Such a problem is usually formulated as a Markov decision process (MDP), given by a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$; here $\mathcal{S}$ denotes the set of states in the environment, $\mathcal{A}$ denotes the set of actions available to the agent, $p$ is the transition probability function, $r$ is the reward function, and $\gamma$ is the discount factor. When the MDP is not fully specified, i.e. when the transition probabilities $p$ or the reward function $r$ are unknown, one needs to use reinforcement learning techniques (such as Q-learning). However, if the MDP is fully specified, an optimal policy can be found using dynamic programming.

In this problem, you will consider the `Chain` environment shown in Figure 1. Specifically, in this environment, there are 5 states arranged in a chain, and 2 possible actions available to the agent in each state: `f` ("forward") and `b` ("backward"). Once an agent takes an action, there is some stochasticity in how the environment responds, and correspondingly, which state the agent ends up in/what reward it receives. Each action in a state is depicted by a small black circle; the arrows from actions to states depict possible transitions, labeled by the probability with which that transition occurs given the action and previous state, and the associated reward. As can be seen, when the agent takes the `f` action, it usually (with probability 0.9) moves one step forward, and receives zero reward (unless it is in state 4, in which case it usually stays in state 4 and receives a reward of 10), but it sometimes (with probability 0.1) falls back all the way to state 0 and receives a reward of 2. When the agent takes the `b` action, the reverse happens: it usually (with probability 0.9) falls back to state 0 and receives a reward of 2, but sometimes (with probability 0.1) moves a step forward/stays in state 4 and receives a different reward.

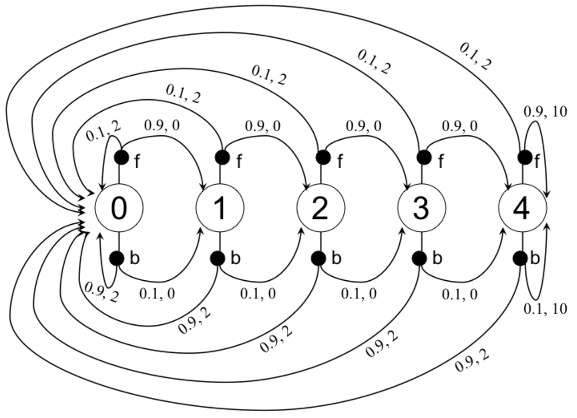

Figure 1: `Chain` environment

Assume a discount factor of 0.9. The MDP here is then fully specified, and you will use dynamic programming to find an optimal deterministic policy for the agent in this environment.

1. **[10 points]** Formulate an MDP for the above `Chain` environment by specifying each component of the tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$. For the components $p$ and $r$, write down $p(s'|s,a)$ and $r(s,a,s')$ for each setting of $s, a, s'$ for which the probability/reward is non-zero.

2. **[15 points]** Find the optimal state-value function $V^*$, i.e. find the optimal value $V^*(s)$ for each state $s$.

   *(Hint: You will need to solve the Bellman optimality equation for $V^*$:*

$$V^*(s) = \max_a \sum_{s'} p(s'|s,a)\Big(r(s,a,s') + \gamma V^*(s')\Big) .$$

*You may write a small piece of code to solve this using value iteration. However, you do not need to turn in the code, and we will not grade your code. You can start with any initial value function $V^0$, iteratively compute $V^{t+1}$ by evaluating the RHS above on $V^t$, and repeat until convergence (until $V^{t+1}$ becomes indistinguishable from $V^t$.)*

3. [**10 points**] Using your solution to the second part above, find an optimal deterministic policy $\pi^*$, i.e. find an optimal action $\pi^*(s)$ for each state $s$.

   *Hint: Recall that an optimal policy is given by* $\pi^*(s) \in \arg\max_a \underbrace{\sum_{s'} p(s'|s,a)\big(r(s,a,s') + \gamma V^*(s')\big)}_{Q^*(s,a)}$. *You may write a small piece of code to find such an optimal policy. However, you do not need to turn in the code, and we will not grade your code.*

## 2 Reinforcement Learning: Q-Learning [35 points]

In this question, we will implement Q-learning for a classic reinforcement learning problem, MountainCar. See the provided notebook for more details.

### 2.1 "Traditional" Q-Learning

Implement the choose_action, update_epsilon, update_Q, and Qlearning functions in the notebook. We will implement Q-learning as discussed in lecture, with an $\epsilon$-greedy strategy for choosing actions. We will use a simple annealing strategy (update_epsilon) by decaying $\epsilon$ by a constant rate in each episode.

[**16 points**] After running Q-learning using the parameters provided in the notebook, use the code provided to plot the **Average Reward, Success Rate, and Car Final Position vs Episodes**. Also give an image of your car reaching the flag in the final episode.

### 2.2 Deep Q-Learning

Now we will take a deep learning approach by using a neural network to represent the Q values. The network will be as follows:

| **Layers** | **Size** |
|---|---|
| Fully Connected Layer 1 | Input size: 2 (size of state space), Output Size: 100 |
| Tanh 1 | — |
| Fully Connected (Hidden) Layer | Input size: 100, Output size: 100 |
| Tanh 2 | — |
| Fully Connected Layer 2 | Input size: 100, Output size: 3 (size of action space) |

Table 1: Architecture for the QNetwork

For this architecture, do not use bias terms for the fully connected layers.

Implement the QNetwork class, and the choose_action, reward_shaping, DeepQlearning functions.

With respect to the reward_shaping function, implement a new reward function such that: The reward is always 0.5 + position of the car in the next state, and if the position of the car in the next state is greater than 0.5 (a successful episode), add +1 on top of the reward you just computed. Return this reward.

[**12 points**] After running deep Q-learning using the parameters provided in the notebook, plot the **Success Rate, and Car Final Position vs Episodes**.

## 2.3 Questions

1. [**3 points**] As part of the Deep QLearning implementation, you implemented a reward_shaping function to aid in the learning process. Compare this to the original reward structure in part 2.1 – why do you think this modification of the reward is helpful?
2. [**4 points**] Compare the **Success Rate** and **Car Final Position** plots between your two implementations. Which algorithm is learning a successful policy more quickly? Briefly comment on potential reasons for any differences in performance.

# 3 Autoencoders [30 points]

Recall from Homework 5 that we examined reconstruction using PCA. We will now look at the same problem of reconstruction from a neural networks perspective. In this section, you will be working with the Fashion MNIST dataset and look at fitting a standard autoencoder to it, and look at the reconstruction properties. See the notebook template for a primer on autoencoders.

You will be using PyTorch for this on Google Colab. Template code is provided along with the notebook. This section will not be auto-graded. You will submit your code to Gradescope, along with the other programming sections.

## 3.1 Part 1: Constructing the Autoencoder

1. [**10 points**] You will first be constructing the autoencoder class and training it with a fixed architecture. Let $n$ be the size of the bottleneck layer. The autoencoder architecture is as follows:

| **Layers** | **Size** |
|---|---|
| Fully Connected Layer 1 | Input size: 28*28*1, Output Size: 256 |
| ReLU 1 | — |
| Fully Connected Layer 2 | Input size: 256, Output size: 64 |
| ReLU 2 | — |
| Fully Connected Layer 3 | Input size: 64, Output size: $n$ |
| ReLU 3 | — |
| Fully Connected Layer 4 | Input size: $n$, Output size: 64 |
| ReLU 4 | — |
| Fully Connected Layer 5 | Input size: 64, Output size: 256 |
| ReLU 5 | — |
| Fully Connected Layer 6 | Input size: 256, Output size: 28*28 |
| Tanh 1 | — |

Table 2: Architecture for a vanilla autoencoder

Note that the first three layers and ReLUs are part of the "encoder," while the last three layers plus the final Tanh output are part of the "decoder."

For 3.1 and 3.2, let the bottleneck layer $n = 2$ for this question. The Optimizer to use will be Adam, with a learning rate of $1e^{-3}$, and the loss is the Mean Squared Error. Use a batch size of 128, and train for 25 epochs. Normalize the input dataset with a mean and standard deviation of 0.5. **It will take up to 5 minutes to train a single model, so please plan accordingly.** Look for the code marked `#TODO` in the notebook and complete the code between the segments `#Begin Your Code` and `#End Your Code`.

Please report the reconstructed images from Epoch 0, Epoch 10 and Epoch 20. These reconstructions are the output of your autoencoder for the last minibatch of the epoch. Record these outputs for 0, 10, 20 Epochs respectively. Also report the training curve of the autoencoder (mean epoch loss vs epochs).

## 3.2 Part 2: Latent Space Decomposition

1. [**5 points**] You will now plot the latent space of this autoencoder, which is essentially the encoding that is obtained for any given input. In this problem, the latent space is the output of the ReLU 3, after the Fully connected layer 3. Use your learned model from the previous section, and plot the images with the class label 0 and for the images with the class label 7 in the 2D latent space.

2. [**5 points**] From the latent space plot, explain what the encoding has done to the inputs. How is this effect related to what PCA does? Why is this useful?

## 3.3 Part 3: Reconstruction Error vs Bottleneck Layer

1. [**5 points**] In this section, you will extend your code in Part 1 to a variable size bottleneck layer. Use your existing code in Part 1 and run the same for the bottleneck layer sizes $[2, 8, 32, 64]$. Record the training curves and the final reconstruction errors for the input images belonging to the class label 0 for each of the sizes. Report the combined training curves (mean epoch loss vs epochs) for all the configurations. Also report the reconstructed images for Epoch 20 for each of the configuration.

2. [**5 points**] Plot the mean reconstruction error of the 4 different trained models when the input images belong to the class label 0, with respect to the size of the bottleneck layer. What are you observing? How does what you see relate to PCA? What does this tell you about how you can potentially work with a high-dimensional data-space?