USC CSCI 467
Intro to Machine Learning

Assignment 4

Due: May 1, 2025, 11:59pm

Spring 2025
Instructor: Robin Jia

This assignment has 3 questions, for a total of 100 points. Make sure you also download the `hw4.zip` file from the course website, and install the required libraries by running `pip install -r requirements.txt`.

When submitting on Gradescope, note that **you must make a submission both for the written portion and programming portion.** For the programming portion, upload the files `wordvec.py` and `qlearning.py` with your completed solutions. (There is an "autograder" which will not actually grade your code but it will run it, and should return 0 if it encountered an error and 100 otherwise.) **Please still include the output of your code in the PDF report when requested in the problems.**

Question 1: Word Vectors and PCA (40 points)

In this question, you will get some experience working with word vectors and implement PCA to visualize word vectors in a low-dimensional space.

The starter code includes a file `vocab.txt` containing a list of common English words, and an associated file `vectors.npy` containing 300-dimensional word vectors for each vocabulary word. These word vectors were taken from a much larger list of word vectors trained with word2vec on Google News articles (we extracted a subset to reduce runtime and disk usage; the full word vector file is 3GB).

(a) (10 points) Similarity in word vector space captures various aspects of semantic similarity (that is, similarity in word meaning). Let's first write a function that takes in a given word vector and finds the k most similar words. To define similarity between two words, we will use the **cosine similarity** of their word vectors. Given two vectors u and v, the cosine similarity is defined as

$$\text{cossim}(u, v) = \frac{u^\top v}{\|u\|\|v\|}.$$

Note that this is exactly equal to the cosine of the angle between vectors u and v. The cosine similarity is thus always between -1 and 1; two vectors have similarity of 1 if and only if they point in exactly the same direction.

Now implement `find_neighbors()`. This function takes as input a list `vocabulary` of n words, a matrix `word_vectors` of shape $n \times d$ where the i-th row is the word vector for the i-th word in `vocabulary`, a d-dimensional vector `query`, and an integer k. The function should return a list of k 2-tuples where each tuple is `(neighbor_word, cosine_similarity)`, and the words in the list are the k most similar words to `query` sorted from most similar to least similar. **You should not use a for-loop over the entire vocabulary size.** You may use a for loop that runs for k iterations, to generate the final output list. Hint: The function `numpy.argsort` may be helpful.

When you're ready, you can find the 10 nearest neighbors to a given word with

```
python3 wordvec.py neighbors -k 10 -q [word]
```

This will run your code using the word vector for the provided word as the query. If you query the word *cat*, the top three neighbors should be *cat, cats, dog*, and *cat* should have similarity of 1 with itself (since the cosine similarity between a vector and itself is always 1).

(b) Now let's take a look at some nearest neighbors for a few more words.

i. (2 points) `query=disappointed`: Some of the neighbors words have a similar meaning to the *disappointed*, but others mean the opposite. Why do you think these other words show up on the list? Hint: Try to think about some phrases where either *disappointed* or a word with the opposite meaning could both fit.

ii. (2 points) `query=bug`: Bug is a *polysemous* word—it can have very different meanings depending on the context. Use this fact to explain the nearest neighbors for *bug*. You should notice three different senses of the word.

iii. (2 points) `query=plate`: Plate is also a polysemous word. In everyday conversation, the most common meaning is something that holds food, so you might expect to find neighbors like *bowl* or *cup*. However, the neighbors your code should output are actually mostly related to baseball (e.g., *mound*, *batter*, *catcher*, *bunt*, *infield*). Provide an explanation of this observation. Hint: Think about the dataset used to train these word vectors.

iv. (2 points) Explore on your own and pick another word where you notice something interesting or surprising about the nearest neighbors. Describe what you find.

(c) (5 points) Let v_w denote the vector for word w. Another interesting thing you can do with word vectors is define relationships between words. For example, the difference $v_{man} - v_{king}$ is similar to the difference $v_{woman} - v_{queen}$, i.e.

$$v_{man} - v_{king} \approx v_{woman} - v_{queen}$$

because these pairs of words have a similar relationship between them. Because of this fact, we can do vector arithmetic to complete the analogy "man is to king as woman is to ..." First compute

$$v_{king} - v_{man} + v_{woman}$$

We know this should approximately equal the vector for the female equivalent to *king*. So, we can search our set of word vectors for the word whose vector is closest to this vector (in terms of cosine similarity).

Fill in the missing code in the function `query_relation`. This function takes in three words: `head1`, `tail1`, and `head2`. Your code will compute the query vector that can be passed to `find_neighbors` to find the best candidates to complete the analogy "`head1` is to `tail1` as `head2` is to...".

When you're done, you should be able to answer relation queries by running:

```
python3 wordvec.py relation -q man,king,woman
```

If you have implemented this correctly, this should return `queen`.

(d) (3 points) Do your own exploration with other types of analogies. Find one other example where the word vectors give you a good answer, and another example where they give you a bad answer.

(e) (10 points) Now let's visualize word vectors with PCA. Implement the function `project_2d`, which takes in a matrix X of shape $n \times d$, and returns a matrix of shape $n \times 2$ by using the top two principal components.

More specifically, to implement PCA you will need to do the following:

1. Mean-center the data. This means you compute μ, the average vector of all the rows in X, and then subtract μ from each row in X. This makes the resulting matrix have a mean row of 0.

2. Compute the $d \times d$ covariance matrix $\Sigma = X^\top X$.
3. Use `np.linalg.eig` to compute the eigendecomposition of Σ. This function returns the d eigenvalues and d eigenvectors of Σ; in particular, the i-th **column** of the eigenvectors matrix is the eigenvector corresponding to the i-th eigenvalue. Note that the eigenvalues are not guaranteed to be sorted in any order.
4. Locate the largest eigenvalue, which we will call λ_1, and second-largest eigenvalue, which we will call λ_2. Let the corresponding eigenvectors be v_1 and v_2. Hint: `np.argsort` will again be useful here.
5. For each row x of the mean-centered matrix X, compute the 2-dimensional projection $[x^\top v_1, x^\top v_2]$. This projects x onto v_1 and v_2, respectively. Return the $n \times 2$ matrix where the i-th row is the projection of the i-th row of X computed in this way.

(f) (4 points) Finally, let's visualize some word vectors. Run

```
python3 wordvec.py visualize -f adjectives.txt
```

This will run your PCA code with the word vectors for the words in `adjectives.txt`. It will plot the 2-dimensional projection of each word vector, where the x-axis will show the first principal component (i.e., the projection onto v_1) and the y-axis will show the second principal component (i.e., the projection onto v_2). It will also draw a line connecting every pair of words, which in this file are an adjective in base form followed by the corresponding superlative. Finally, it will write this figure to `visual.png`

Paste the resulting figure and answer the following questions:

- Which principal component primarily captures the difference between the base and superlative form of each adjective?
- Which word vectors are close to each other in this visualization? Why does that make sense?

Question 2: Q-Learning (20 points)

In this problem, you will implement Q-learning with a discrete state space. We will use the Cartpole environment, in which a pole that rotates at its base is placed on top of a car. The RL agent can choose to nudge the car left or right at each timestep, and the goal is to keep the pole upright while keeping the car within the bounds of the environment for as long as possible.

We will be relying on the OpenAI Gym implementation of Cart Pole.[1] There is an *environment* (called `env` in the code) that the agent interacts with. At each timestep, the agent chooses an action, and the environment tells it the new state, reward, and whether the current episode has terminated (in the case of Cartpole, the episode terminates when the pole falls by at most 24° in either direction or the car moves too far to the left or right). In Cartpole, the agent gets a reward of $+1$ for each timestep, which means the total (un-discounted) reward is equal to the total number of timesteps the episode lasts. At the start of each episode, we call `env.reset()` to reset back to a starting state.

The actual state space is continuous, defined by four variables: cart position, cart velocity, pole angle, and pole angular velocity. To run tabular Q-learning, we discretize each variable into a number of bins to get a discrete state space. We will then learn the Q-value for each pair of discrete state and action via Q-learning. This code is already provided for you.

(a) (2 points) Run the baseline policy that just chooses a random action at each timestep:

[1] https://gymnasium.farama.org/environments/classic_control/cart_pole/

```
python3 qlearning.py cartpole -a random
```

This should create an animation of 20 episodes using this random policy. How well does the random policy do? What usually causes the episodes to end?

(b) (12 points) Now implement Q-learning. You will need to fill in code both in `run_qlearning()` to actually do Q-learning, as well as test-time code in `run_test` to use the Q-values you've learned to choose actions at test time. There is a total of three missing code blocks:

- While running Q-learning, you need to use ϵ-greedy exploration: with probability ϵ you should choose a random action, (there is a helpful function `env.action_space.sample()` that samples a random action in the environment's action space), and with probability $1 - \epsilon$ you should choose the optimal action based on the current Q-values. There are many ways to implement this; one way is to use the `random.random()` function, which returns a uniformly random real number between 0 and 1.
- Then, after taking the action, you should update the Q-values for the previous state. Remember to use the provided discount factor when doing the Q-learning update.
- Finally, in `run_test`, you should not use ϵ-greedy, but instead always choose the best action according to the learned Q-values.

When you're ready, run Q-learning with 10,000 episodes and an ϵ of 0.1 for ϵ-greedy exploration.

```
python3 qlearning.py cartpole -a qlearning -n 10000 -e 0.1
```

The average reward across Q-learning will be written to the file `reward_cartpole_eps0.10.png`. Include this figure in your report. You should find that reward increases and by the end you are consistently getting above 100 reward.

After running Q-learning, the code will once again show 20 episodes using the learned policy (and turning off ϵ-greedy). How well does your learned policy do? Contrast its behavior with the random policy.

(c) (3 points) You probably noticed that Q-learning got slower for the Cartpole problem (i.e., 100 episodes took not very much time in the beginning, and took more time later on). Provide an explanation of this.

(d) (3 points) Let's test whether ϵ-greedy was necessary. Try again without ϵ-greedy, i.e. $\epsilon = 0$:

```
python3 qlearning.py cartpole -a qlearning -n 10000 -e 0.0
```

What happens to the learned policy and final reward?

Question 3: Adversarial Examples and Linear Classifiers (40 points)

In class we talked about how adversarial perturbations can fool state-of-the-art neural image classifiers. This problem is not unique to neural networks—linear classifiers can also be attacked! In this problem, we will work out a closed-form attack and defense strategy for linear classifiers.

Suppose we have the following setting:

- We are doing a binary classification task of mapping an image x, represented as a d-dimensional vector of pixels, to a label $y \in \{-1, 1\}$. (For example, if the images are 28×28, $d = 28^2 = 784$.)

- Our model is a linear classifier parameterized by a weight vector $w \in \mathbb{R}^d$ and bias term $b \in \mathbb{R}$. To make a prediction on input x, we compute $w^\top x + b$ and predict $+1$ if it is > 0, and -1 otherwise.
- The loss we care about is the logistic loss. Given an example (x, y), the logistic loss for parameters w and b is defined as $\ell(x, y; w, b) = -\log \sigma(y \cdot (w^\top x + b))$.
- The adversary is allowed to perturb each pixel of x by at most ϵ, for some $\epsilon > 0$.

(a) (8 points) Suppose that the attacker uses the fast gradient sign method (FGSM) to attack our model. Recall that in FGSM, the attacker first computes

$$g = \nabla_x \ell(x, y; w, b),$$

the gradient of the loss with respect to the **input** x (not the parameters). Derive the formula for g in terms of x, y, w, and b. Hint: If you are stuck on differentiating $\log \sigma(\cdot)$, you can refer back to the lecture on logistic regression.

(b) (6 points) The next step for the attacker is to perturb x. Let $z \in \mathbb{R}^d$ denote the vector that the attacker adds to x. For each component x_j of x, they will add ϵ to x_j if $g_j > 0$, and subtract ϵ from x_j if $g_j < 0$, and leaves x_j unchanged if $g_j = 0$. Based on your work from the previous part, show that

$$z_j = -\epsilon y \cdot \operatorname{sgn}(w_j),$$

where $\operatorname{sgn}(a)$ denotes the "sign" of a, which is $+1$ if $a > 0$, -1 if $a < 0$, and 0 if $a = 0$. Hint: You may use the fact that $\operatorname{sgn}(ab) = \operatorname{sgn}(a) \cdot \operatorname{sgn}(b)$.

(c) (10 points) The adversary modifies the original input x to the perturbed input $x + z$. This perturbed input is then fed to the model. Let $\ell_{adv}(x, y; w, b)$ denote the loss of the model on the perturbed input $x + z$. Show that this is equal to

$$\ell_{adv}(x, y; w, b) = -\log \sigma(y \cdot (w^\top x + b) - \epsilon \|w\|_1),$$

where $\|w\|_1$ denotes the L_1 norm of w, i.e.

$$\|w\|_1 = \sum_{j=1}^{d} |w_j|.$$

(d) (6 points) By comparing the formula for $\ell(x + z, y; w, b)$ with the original loss $\ell(x, y; w, b)$, show that the loss on the adversarially perturbed example is always higher than the loss on the original example, as long as $\|w\|_1 > 0$. You may use the fact that $\log(a)$ and $\sigma(a)$ are both monotonically increasing functions in a.

(e) (10 points) Finally, we can use the formula in part (c) to train our model to do well on adversarial perturbations. We can do this simply by running gradient descent on $\ell_{adv}(x, y; w, b)$.

Derive the formulas for the gradient of $\ell_{adv}(w, b)$ with respect to both w of b. (Note that for simplicity, we will just calculate the gradient of the loss on a single example, and not an entire training dataset.) You can use $\operatorname{sgn}(v)$ for a vector v to denote the vector whose i-th element is $\operatorname{sgn}(v_i)$.