CS 229, Fall 2025
Problem Set #4

YOUR NAME HERE (`YOUR SUNET HERE`)

Due Wednesday, November 19 at 11:59 pm on Gradescope.

Notes: (1) These questions require thought, but do not require long answers. Please be as concise as possible.

(2) If you have a question about this homework, we encourage you to post your question on our Ed forum, at `https://edstem.org/us/courses/87361/discussion`.

(3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work.

(4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted.

(5) The due date is Wednesday, November 19 at 11:59 pm. If you submit after Wednesday, November 19 at 11:59 pm, you will begin consuming your late days. The late day policy can be found in the course website: Course Logistics and FAQ.

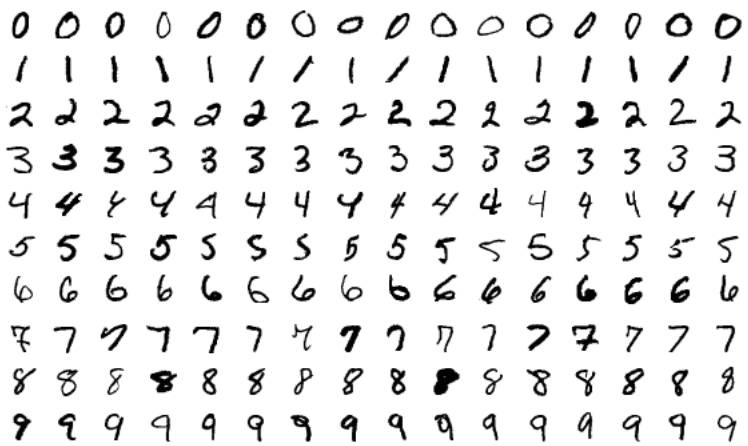
All students must submit an electronic PDF version of the written question including plots generated from the codes. We highly recommend typesetting your solutions via LaTeX. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup. Please make sure that your PDF file and zip file are submitted to the corresponding Gradescope assignments respectively. We reserve the right to not give any points to the written solutions if the associated code is not submitted.

Honor code: We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solution independently, and without referring to written notes from the joint session. Each student must understand the solution well enough in order to reconstruct it by him/herself. It is an honor code violation to copy, refer to, or look at written or code solutions from a previous year, including but not limited to: official solutions from a previous year, solutions posted online, and solutions you or someone else may have written up in a previous year. Furthermore, it is an honor code violation to post your assignment solutions online, such as on a public git repo. We run plagiarism-detection software on your code against past solutions as well as student submissions from previous years. Please take the time to familiarize yourself with the Stanford Honor Code[1] and the Stanford Honor Code[2] as it pertains to CS courses.

[1] https://communitystandards.stanford.edu/policies-and-guidance/honor-code
[2] https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1164/handouts/honor-code.pdf

1. **[30 points] Neural Networks: MNIST image classification**

In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images are shown below.



The data and starter code for this problem can be found in

- `src/mnist/nn.py`
- `src/mnist/images_train.csv`
- `src/mnist/labels_train.csv`
- `src/mnist/images_test.csv`
- `src/mnist/labels_test.csv`

The dataset files are initially provided in compressed `.gz` format. Be sure to unzip them before use so that the files are accessible as `.csv` files. To unzip the files:

- On Mac/Linux: `gunzip images_train.csv.gz` (repeat for each `.gz` file)
- On Windows: Use a tool like 7-Zip, WinRAR, or the built-in extraction feature

The starter code splits the set of 60,000 training images and labels into a set of 50,000 examples as the training set, and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. You will use the sigmoid function as activation for the hidden layer and use the cross-entropy loss for multi-class classification. For a single example (x, y), the cross-entropy loss is given by:

$$\ell_{\text{CE}}(\bar{h}_\theta(x), y) = -\log\left(\frac{\exp(\bar{h}_\theta(x)_y)}{\sum_{s=1}^{k}\exp(\bar{h}_\theta(x)_s)}\right),$$

where $\bar{h}_\theta(x) \in \mathbb{R}^k$ denotes the logits, i.e., the output of the model on a training example x, and $\bar{h}_\theta(x)_y$ is the y-th coordinate of the vector $\bar{h}_\theta(x)$ (with $y \in \{1, \ldots, k\}$ serving as an index).

We have labeled data $(x^{(i)}, y^{(i)})_{i=1}^n$, where $x^{(i)} \in \mathbb{R}^d$, and $y^{(i)} \in \{1, \ldots, k\}$ is the ground truth label. Note that, in this specific application, we pass in the *flattened* images as input. In other words, if each original input image is in $\mathbb{R}^{r \times r}$, we have $x^{(i)} \in \mathbb{R}^{r^2}$.

Additionally, let m be the number of hidden units in the neural network, so that weight matrices $W^{[1]} \in \mathbb{R}^{d \times m}$ and $W^{[2]} \in \mathbb{R}^{m \times k}$.[3] We also have biases $b^{[1]} \in \mathbb{R}^m$ and $b^{[2]} \in \mathbb{R}^k$. The parameters of the model θ is $(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]})$.

We define the following neural network with a single hidden layer, which we refer to as the **Linear Layer NN**. For a single input $x^{(i)}$, the forward propagation equations are:

$$\begin{aligned} a^{(i)} &= \sigma\left({W^{[1]}}^\top x^{(i)} + b^{[1]}\right) \in \mathbb{R}^m \\ \bar{h}_\theta(x^{(i)}) &= {W^{[2]}}^\top a^{(i)} + b^{[2]} \in \mathbb{R}^k \\ h_\theta(x^{(i)}) &= \text{softmax}(\bar{h}_\theta(x^{(i)})) \in \mathbb{R}^k \end{aligned}$$

where σ is the sigmoid function. The softmax function maps $\mathbb{R}^k \to \mathbb{R}^k$ and is defined as:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^k \exp(z_j)}$$

for each component $i \in \{1, \ldots, k\}$.

For n training examples, we average the cross entropy loss over the n examples.

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n}\sum_{i=1}^n \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)}) = -\frac{1}{n}\sum_{i=1}^n \log\left(\frac{\exp(\bar{h}_\theta(x^{(i)})_{y^{(i)}})}{\sum_{s=1}^k \exp(\bar{h}_\theta(x^{(i)})_s)}\right).$$

Suppose $e_y \in \mathbb{R}^k$ is the one-hot embedding/representation of the discrete label y, where the y-th entry is 1 and all other entries are zeros. We can also write the loss function in the following way:

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = -\frac{1}{n}\sum_{i=1}^n e_{y^{(i)}}^\top \log\left(h_\theta(x^{(i)})\right).$$

Here $\log(\cdot)$ is applied entry-wise to the vector $h_\theta(x^{(i)})$. The starter code already converts labels into one-hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, the common practice is to use mini-batch gradient descent for deep learning tasks. Concretely, we randomly sample B examples $(x^{(i_k)}, y^{(i_k)})_{k=1}^B$ from $(x^{(i)}, y^{(i)})_{i=1}^n$. In this case, the mini-batch cost function with batch-size B is defined as follows:

$$J_{MB} = \frac{1}{B}\sum_{k=1}^B \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_k)}), y^{(i_k)})$$

where B is the batch size, i.e., the number of training examples in each mini-batch.

[3] Please note that the dimension of the weight matrices is different from those in the lecture notes, but we also multiply ${W^{[1]}}^\top$ instead of $W^{[1]}$ in the matrix multiplication layer. Such a change of notation is mostly for some consistence with the convention in the code.

(a) **[5 points]** Let $t \in \mathbb{R}^k, y \in \{1, \ldots, k\}$ and $p = \text{softmax}(t)$. Prove that

$$\frac{\partial \ell_{\text{CE}}(t, y)}{\partial t} = p - e_y \in \mathbb{R}^k, \tag{1}$$

As a direct consequence,

$$\frac{\partial \ell_{\text{CE}}(\bar{h}_\theta(x^{(i)}), y^{(i)})}{\partial \bar{h}_\theta(x^{(i)})} = \text{softmax}(\bar{h}_\theta(x^{(i)})) - e_{y^{(i)}} = h_\theta(x^{(i)}) - e_{y^{(i)}} \in \mathbb{R}^k \tag{2}$$

where $\bar{h}_\theta(x^{(i)}) \in \mathbb{R}^k$ is the input to the softmax function, i.e.

$$h_\theta(x^{(i)}) = \text{softmax}(\bar{h}_\theta(x^{(i)}))$$

(Note: in deep learning, $\bar{h}_\theta(x^{(i)})$ is sometimes referred to as the "logits".)

Answer:

(b) **[15 points] Coding problem.**

In `src/mnist/nn.py`, implement both forward-propagation and back-propagation for the above loss function $J_{MB} = \frac{1}{B} \sum_{k=1}^{B} \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_k)}), y^{(i_k)})$. Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set $B = 1,000$ (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch, we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the development (validation) set, and plot it against the number of epochs.

Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the development (validation) set versus number of epochs.

Submit the two plots (one for loss vs epoch, another for accuracy vs epoch) in your writeup.

At the end of training, the program will automatically print the test accuracy for your model. Be sure to record this printed accuracy, as you will compare it with the regularized model in later parts.

Hint: Be sure to vectorize your code as much as possible! Training can be very slow otherwise. For better vectorization, use one-hot label encodings in the code (e_y in part (a)).

Answer:

(c) **[7 points] Coding problem.**

Now add a regularization term to your cross entropy loss. The loss function will become

$$J_{MB} = \left(\frac{1}{B} \sum_{k=1}^{B} \ell_{\text{CE}}(\bar{h}_\theta(x^{(i_k)}), y^{(i_k)}) \right) + \lambda \left(||W^{[1]}||^2 + ||W^{[2]}||^2 \right)$$

Be careful not to regularize the bias/intercept term. Set λ to be 0.0001. Implement the regularized version and plot the same figures as part (b). Be careful NOT to include the regularization term when plotting the loss values (i.e., regularization should only be used for gradient calculation for the purpose of training).

Submit the two new plots obtained with regularized training (i.e loss (without regularization term) vs epoch, and accuracy vs epoch) in your writeup.

Compare the plots obtained from the regularized model with the plots obtained from the non-regularized model, and summarize your observations in a couple of sentences.

As in the previous part, the program will automatically print the test accuracy for your regularized model. Record this printed accuracy so you can compare it with the non-regularized model in the next part.

Answer:

(d) **[3 points]**

Now that you have convinced yourself that the model is working as expected (i.e., the observations you made in the previous part match what you learnt in class about regularization), it is time to report the test performance. Once the test performance is recorded, we report it as is and do not go back to refine the model.

The program in the previous parts already evaluates both models on the test set and prints their accuracies. Now, using the recorded test accuracies, compare the performance of the non-regularized and regularized models, and briefly (in one sentence) explain why the regularized model performs better on the test set.

You should have accuracy close to 0.92870 without regularization, and 0.96760 with regularization. Note: these accuracies assume you implement the code with the matrix dimensions as specified in the comments. Even if you do not match these numbers exactly, you should observe good accuracy and better test accuracy with regularization.

Answer:

2. **[15 points] Markov decision processes**

Consider an MDP with finite state and action spaces, and discount factor $0 < \gamma < 1$. Let B be the Bellman update operator with V a vector of values for each state. I.e., if $V' = B(V)$, then

$$V'(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V(s').$$

(a) **[10 points]** Prove that, for any two finite-valued vectors V_1, V_2, it holds true that

$$||B(V_1) - B(V_2)||_\infty \leq \gamma ||V_1 - V_2||_\infty.$$

where

$$||V||_\infty = \max_{s \in S} |V(s)|.$$

(This shows that the Bellman update operator is a "γ-contraction in the max-norm.")
Remark: The result you proved in part (a) implies that value iteration converges geometrically (i.e. exponentially) to the optimal value function V^*.
Hint: You can start by proving the following useful fact: for any functions f and g,

$$|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|.$$

Answer:
Let $f(a_f) = \max_a f(a)$, and $g(a_g) = \max_a g(a)$.
Thus,

$$\max_a f(a) - \max_a g(a) = f(a_f) - g(a_g)$$

Since $f(a_f) \geq f(a_g)$,

$$f(a_f) - g(a_g) \leq f(a_g) - g(a_g)$$

$$B(V_1) - B(V_2) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V_1(s') - R(s) - \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V_2(s')$$

$$= \gamma \left[\max_{a \in A} \sum_{s' \in S} P_{sa}(s')V_1(s') - \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V_2(s') \right]$$

$$f(a) = \sum_{s' \in S} P_{sa}(s')V_1(s'), \qquad f(a) = \sum_{s' \in S} P_{sa}(s')V_2(s')$$

V a vector of values for each state. I.e., if $V' = B(V)$, then

$$V'(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_{sa}(s')V(s').$$

$$||B(V_1) - B(V_2)||_\infty \leq \gamma ||V_1 - V_2||_\infty.$$

where

$$||V||_\infty = \max_{s \in S} |V(s)|.$$

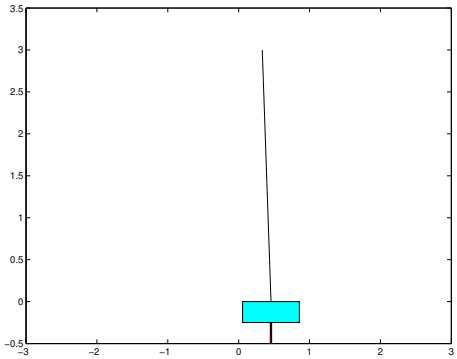
(b) **[5 points]** We say that V is a **fixed point** of B if $B(V) = V$. Using the fact that the Bellman update operator is a γ-contraction in the max-norm, prove that B has at most one fixed point—i.e., that there is at most one solution to the Bellman equations. You may assume that B has at least one fixed point.
Answer:

3. **[25 points] Reinforcement Learning: The inverted pendulum**

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.[4]

Consider the figure shown. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.



We have written a simple simulator for this problem. The simulation proceeds in discrete time cycles (steps). The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position x, the cart velocity $\dot{x}$, the angle of the pole θ measured as its deviation from the vertical position, and the angular velocity of the pole $\dot{\theta}$. Since it would be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector $(x, \dot{x}, \theta, \dot{\theta})$ into a number from 0 to `NUM_STATES-1`. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no *do-nothing* action.) These are represented as actions 0 and 1 respectively in the code. When the action choice is made, the simulator updates the state parameters according to the underlying dynamics, and provides a new discretized state.

We will assume that the reward $R(s)$ is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards observed.

To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

[4]The dynamics are adapted from `http://www-anw.cs.umass.edu/rlr/domains.html`

Briefly, you will maintain a current model of the MDP and a current estimate of the value function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state s_i to state s_j using action a has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabilities and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, assume that the whole learning procedure has converged once several consecutive attempts (defined by the parameter `NO_LEARNING_THRESHOLD`) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

(a) [20 points] **Coding problem.**

Follow the instructions in `src/cartpole/cartpole.py` to implement the algorithm. Use a discount factor of $\gamma = 0.995$. **In particular, implement the following functions:**

i. `choose_action(state, mdp_data)`

ii. `update_mdp_transition_counts_reward_counts(mdp_data, state, action, new_state, reward)`

iii. `update_mdp_transition_probs_reward(mdp_data)`

iv. `update_mdp_value(mdp_data, tolerance, gamma)`

Once you've finished implementing the above functions, run the experiment via `python cartpole.py`. Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial (the starter code already includes the code to plot). **Include this plot in your writeup.**

Note: With a correct implementation, you should expect to see the log of steps to failure reach around 5-7 and be converging by the end of the experiment. The number of failures before convergence can vary depending on implementation details, but the algorithm should converge well before reaching the maximum of 500 failures.

Answer:

(b) [5 points] As the algorithm runs, it will print "[INFO] Failure number ###" each time the pole falls or cart goes out of bounds. When the algorithm converges and stops, the final printed failure number indicates how many trials it took to converge. Find the line of code that says `np.random.seed`, and rerun the code with the seed set to 1, 2, and 3. What do you observe?

Answer:

4. **[15 points] PCA**

In class, we showed that PCA finds the "variance maximizing" directions onto which to project the data. In this problem, we find another interpretation of PCA.

Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(n)}\}$. Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector u, let $f_u(x)$ be the projection of point x onto the direction given by u. I.e., if $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$, then

$$f_u(x) = \arg\min_{v \in \mathcal{V}} ||x - v||^2.$$

(a) [10 points] Show that the unit-length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

$$\arg\min_{u:u^Tu=1} \sum_{i=1}^{n} \|x^{(i)} - f_u(x^{(i)})\|_2^2$$

is equivalent to the optimization problem

$$\arg\max_{u:u^Tu=1} u^T \left(\sum_{i=1}^{n} x^{(i)} {x^{(i)}}^T \right) u,$$

which defines the first principal component (the eigenvector corresponding to the largest eigenvalue of $\sum_{i=1}^{n} x^{(i)} {x^{(i)}}^T$).

Remark. If we are asked to find a k-dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the k-dimensional subspace spanned by the first k principal components of the data. This problem shows that this result holds for the case of $k = 1$.

Answer:

(b) [5 points] Now we will explore the relationship between two of the most popular dimensionality reduction techniques, SVD and PCA, at a basic conceptual level. Before we proceed with the question itself, let us briefly recap the SVD and PCA techniques and a few important observations:

- **Eigenvalue Decomposition**: First, recall that the eigenvalue decomposition of a real, symmetric, and square matrix B (of size $d \times d$) can be written as the following product:

  $$B = Q \Lambda Q^\top$$

  where $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_d)$ contains the eigenvalues of B (which are always real) along its main diagonal, and Q is an orthogonal matrix containing the eigenvectors of B as its columns.
- **Principal Component Analysis (PCA)**: Given a data matrix M (of size $p \times q$), we showed in part (a) that PCA involves finding eigenvectors of the matrix $M^\top M$. The matrix of these eigenvectors can be thought of as a rigid rotation in a high-dimensional space. PCA then projects each row of M onto the top k principal components to produce a lower dimensional version of each data point (where $k << q$).

Now we turn to the question! Let us define a real matrix M (of size $p \times q$) and let us assume this matrix corresponds to a dataset with p data points and q dimensions.

i. [2 points] Prove that $M^\top M$ is real, symmetric, and square. Write its eigenvalue decomposition in terms of Q, Λ, Q^T.

ii. [2 points] SVD involves the decomposition of a data matrix $M \in \mathbb{R}^{p \times q}$ into a product $M = U\Sigma V^\top$ where $U \in \mathbb{R}^{p \times p}$ and $V \in \mathbb{R}^{q \times q}$ are column-orthonormal matrices and $\Sigma \in \mathbb{R}^{p \times q}$ is a diagonal matrix. The entries along the diagonal of Σ are referred to as the singular values of M. Write a simplified expression for $M^\top M$ in terms of V, V^T, and Σ.
Hint: A matrix A is column-orthonormal if and only if $A^T A = I$

iii. [1 points] What is the relationship (if any) between the eigenvalues of $M^\top M$ and the singular values of M?

Answer:

If you got here and finished all the above problems, you are done with the final PSet of CS 229! We know these assignments are not easy, so well done :)