

# CSCI-GA.2565 — Homework 1

*your name and NetID here*

Version 1.0

## Instructions.

- **Due date.** Homework is due **Tuesday, September 30, at noon ET**.
- **Gradescope submission.** Everyone must submit individually at gradescope under **hw1** and **hw1code**: **hw1code** is just python code, whereas **hw1** contains everything else. For clarity, problem parts are annotated with where the corresponding submissions go.
  - **Submitting hw1.** **hw1** must be submitted as a single PDF file, and typeset in some way, for instance using L<sup>A</sup>T<sub>E</sub>X, Markdown, Google Docs, MS Word; you can even use an OCR package (or a modern multi-modal LLM) to convert handwriting to L<sup>A</sup>T<sub>E</sub>X and then clean it up for submission. Graders reserve the right to award zero points for solutions they consider illegible.
  - **Submitting hw1code.** Only upload the two python files **hw1.py** and **hw1\_utils.py**; don't upload a zip file or additional files.
- **Consulting LLMs and friends.** You may discuss with your peers and you may use LLMs. *However*, you are strongly advised to make a serious attempt on all problems alone, and if you consult anyone, make a serious attempt to understand the solution alone afterwards. You must document credit assignment in a special final question in the homework.
- **Evaluation.** We reserve the right to give a 0 to a submission which violates the intent of the assignment and is morally equivalent to a blank response.
  - **hw1code:** your grade is what the autograder gives you; note that you may re-submit as many times as you like until the deadline. However, we may reduce your auto-graded score if your solution simply hard-codes answers.
  - **hw1:** you can receive 0 points for a blank solution, an illegible solution, or a solution which does not correctly mark problem parts with boxes in the gradescope interface (equivalent to illegibility). All other solutions receive full points, *however* the graders do leave feedback so please check afterwards even if you received a perfect score.
- **Regrades.** Use the gradescope interface.
- **Late days.** We track 3 late days across the semester per student.
- **Library routines.** Coding problems come with suggested “library routines”; we include these to reduce your time fishing around APIs, but you are free to use other APIs.

## Version history.

1.0. Initial version.

# 1. Linear Regression/SVD.

Throughout this problem let  $\mathbf{X} \in \mathbb{R}^{n \times d}$  with  $i$ -th row  $\mathbf{x}_i \in \mathbb{R}^d$ . Suppose  $\mathbf{X} = \sum_{i=1}^r s_i \mathbf{u}_i \mathbf{v}_i^\top$  is its singular value decomposition, where  $\mathbf{u}_i \in \mathbb{R}^n$  are left singular vectors,  $\mathbf{v}_i \in \mathbb{R}^d$  are right singular vectors,  $s_i > 0$  are singular values, and  $r = \text{rank}(\mathbf{X})$ .

- (a) [hw1] Let the dataset consist of  $n_i > 0$  copies of the standard basis vector  $\mathbf{e}_i \in \mathbb{R}^d$  for each  $i \in \{1, \dots, d\}$ , with labels  $(y_{ij})_{j=1}^{n_i}$  and  $\sum_{i=1}^d n_i = n$ . Equivalently, the training set is

$$\left\{ (\mathbf{e}_i, y_{ij}) : i \in \{1, \dots, d\}, j \in \{1, \dots, n_i\} \right\}.$$

Show that for a vector  $\mathbf{w}$  that minimizes the empirical risk under squared loss, the components  $w_i$  of  $\mathbf{w}$  are the averages of the labels  $(y_{ij})_{j=1}^{n_i}$ :  $w_i = \frac{1}{n_i} \sum_{j=1}^{n_i} y_{ij}$ .

**Hint:** Write out the expression for the empirical risk with the squared loss and set the gradient equal to zero.

**Remark:** This illustrates the classic “regression toward the mean” phenomenon.

- (b) [hw1] Returning to a general matrix  $\mathbf{X}$ , show that if the label vector  $\mathbf{y}$  is a linear combination of  $\{\mathbf{u}_i\}_{i=1}^r$  then there exists a  $\mathbf{w}$  for which the empirical risk is zero (meaning  $\mathbf{X}\mathbf{w} = \mathbf{y}$ ).

**Hint:** Either consider the range of  $\mathbf{X}$  and use the SVD, or compute the empirical risk explicitly with  $\mathbf{y} = \sum_{i=1}^r a_i \mathbf{u}_i$  for some constants  $a_i$  and  $\hat{\mathbf{w}}_{\text{ols}} = \mathbf{X}^+ \mathbf{y}$ .

**Remark:** It’s also not hard to show that if  $\mathbf{y}$  is not a linear combination of the  $\{\mathbf{u}_i\}_{i=1}^r$ , then the empirical risk must be nonzero.

- (c) [hw1] Show that  $\mathbf{X}^\top \mathbf{X}$  is invertible if and only if  $(\mathbf{x}_i)_{i=1}^n$  spans  $\mathbb{R}^d$ .

**Hint:** Recall that the squares of the singular values of  $\mathbf{X}$  are eigenvalues of  $\mathbf{X}^\top \mathbf{X}$ .

**Remark:** This characterizes when linear regression has a unique solution due to the normal equation (note that we always have at least one solution obtained by the pseudoinverse). We would not have had a unique solution for part (a) if some  $n_i = 0$ .

- (d) [hw1] Provide a matrix  $\mathbf{X}$  such that  $\mathbf{X}^\top \mathbf{X}$  is invertible and  $\mathbf{X}\mathbf{X}^\top$  is not. Include a formal verification of this for full points.

**Hint:** Use part (c). It may be helpful to think about conditions under which a matrix is not invertible.

**Solution.**

## 2. Linear Regression.

Recall that the empirical risk in the linear regression method is defined as  $\hat{\mathcal{R}}(\mathbf{w}) := \frac{1}{2n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i - y_i)^2$ , where  $\mathbf{x}_i \in \mathbb{R}^d$  is a data point and  $y_i$  is an associated label.

- (a) [hw1code] Implement linear regression using gradient descent in the `linear_gd(X, Y, lrate, num_iter)` function of `hw1.py`. You are given as input a training set  $\mathbf{X}$  as an  $n \times d$  tensor, training labels  $\mathbf{Y}$  as an  $n \times 1$  tensor, a learning rate `lrate`, and the number of iterations of gradient descent to run `num_iter`. Using gradient descent, find parameters  $\mathbf{w}$  that minimize the empirical risk  $\hat{\mathcal{R}}(\mathbf{w})$ . Use  $\mathbf{w} = 0$  as your initial parameters, and return your final  $\mathbf{w}$  as output. Prepend a column of ones to  $\mathbf{X}$  in order to accommodate a bias term in  $\mathbf{w}$ . Return  $\mathbf{w} \in \mathbb{R}^{(d+1) \times 1}$  with the bias as the first entry.  
**Library routines:** `torch.matmul (@)`, `X.shape`, `X.t()`, `torch.cat`, `torch.ones`, `torch.zeros`, `torch.reshape`.
- (b) [hw1code] Implement linear regression by using the pseudoinverse to solve for  $\mathbf{w}$  in the `linear_normal(X, Y)` function of `hw1.py`. You are given a training set  $\mathbf{X}$  as an  $n \times d$  tensor and training labels  $\mathbf{Y}$  as an  $n \times 1$  tensor. Return your parameters  $\mathbf{w}$  as output. As before, make sure to accommodate a bias term by prepending ones to the training examples  $\mathbf{X}$ . Return  $\mathbf{w} \in \mathbb{R}^{(d+1) \times 1}$  with the bias as the first entry.  
**Library routines:** `torch.matmul (@)`, `torch.cat`, `torch.ones`, `torch.pinverse`.
- (c) [hw1] Implement the `plot_linear()` function in `hw1.py`. Use the provided function `hw1_utils.load_reg_data()` to generate a training set  $\mathbf{X}$  and training labels  $\mathbf{Y}$ . Plot the curve generated by `linear_normal()` along with the points from the data set. Return the plot as output. Include the plot in your written submission.  
**Library routines:** `torch.matmul (@)`, `torch.cat`, `torch.ones`, `plt.plot`, `plt.scatter`, `plt.show`, `plt.gcf` where `plt` refers to the `matplotlib.pyplot` library.

**Solution.**

### 3. Polynomial Regression.

In Problem 2 you constructed a linear model  $\mathbf{w}^\top \mathbf{x} = \sum_{i=1}^d x_i w_i$ . In this problem you will use the same setup as in the previous problem, but enhance your linear model by doing a quadratic expansion of the features. Namely, you will construct a new linear model  $f_{\mathbf{w}}$  with parameters

$$(w_0, w_{01}, \dots, w_{0d}, w_{11}, w_{12}, \dots, w_{1d}, w_{22}, w_{23}, \dots, w_{2d}, \dots, w_{dd})^\top,$$

defined by

$$f_{\mathbf{w}}(x) = \mathbf{w}^\top \phi(\mathbf{x}) = w_0 + \sum_{i=1}^d w_{0i} x_i + \sum_{i \leq j}^d w_{ij} x_i x_j.$$

**Warning:** If the computational complexity of your implementation is high, it may crash the autograder (try to optimize your algorithm if it does)!

- [hw1] Given a 3-dimensional feature vector  $\mathbf{x} = (x_1, x_2, x_3)$ , completely write out the quadratic expanded feature vector  $\phi(\mathbf{x})$ .
- [hw1code] Implement the `poly_gd()` function in `hw1.py`. The input is in the same format as it was in Problem 2. Implement gradient descent on this training set with  $\mathbf{w}$  initialized to 0. Return  $\mathbf{w}$  as the output with terms in this exact order: bias, linear, then quadratic. For example, if  $d = 3$  then you would return  $(w_0, w_{01}, w_{02}, w_{03}, w_{11}, w_{12}, w_{13}, w_{22}, w_{23}, w_{33})$ .

**Library routines:** `torch.cat`, `torch.ones`, `torch.zeros`, `torch.stack`.

**Hint (feature order, important for autograder):** Prepend a column of ones to  $\mathbf{X}$ , then append the *quadratic* features (including cross terms). Use the following exact order:

$$\phi(\mathbf{x}) = \left[ 1, x_1, \dots, x_d, x_1^2, x_1 x_2, \dots, x_1 x_d, x_2^2, x_2 x_3, \dots, x_2 x_d, \dots, x_d^2 \right]^\top.$$

Equivalently, generate terms in lexicographic order over index pairs  $(i, j)$  with  $1 \leq i \leq j \leq d$ :  $(1, 1), (1, 2), \dots, (1, d), (2, 2), (2, 3), \dots, (d, d)$ . Do not include duplicates (e.g., include  $x_1 x_2$  but not  $x_2 x_1$ ). The autograder expects this exact ordering.

- [hw1code] Implement the `poly_normal` function in `hw1.py`. You are given the same data set as from part (b), but this time determine  $\mathbf{w}$  by using the pseudoinverse. Return  $\mathbf{w}$  in the same order as in part (b). Return  $\mathbf{w} \in \mathbb{R}^{(1+d+\frac{d(d+1)}{2}) \times 1}$  with the bias first, then linear, then quadratic terms.

**Library routines:** `torch.pinverse`.

**Hint:** You will still need to transform the matrix  $\mathbf{X}$  in the same way as in part (b).

- [hw1] Implement the `plot_poly()` function in `hw1.py`. Use the provided function `hw1_utils.load_reg_data()` to generate a training set  $\mathbf{X}$  and training labels  $\mathbf{Y}$ . Plot the curve generated by `poly_normal()` along with the points from the data set. Return the plot as output and include it in your written submission. Compare and contrast this plot with the plot from Problem 2. Which model appears to approximate the data better? Justify your answer.

**Library routines:** `plt.plot`, `plt.scatter`, `plt.show`, `plt.gcf`.

- [hw1] The Minsky-Papert XOR problem is a classification problem with data set:

$$X = \{(-1, +1), (+1, -1), (-1, -1), (+1, +1)\}$$

where the label for a given point  $(x_1, x_2)$  is given by its product  $x_1 x_2$ . For example, the point  $(-1, +1)$  would be given label  $y = (-1)(1) = -1$ . Implement the `poly_xor()` function in `hw1.py`. In this function you will load the XOR data set by calling the `hw1_utils.load_xor_data()` function, and then apply the `linear_normal()` and `poly_normal()` functions to generate predictions for the XOR points. Include a plot of contour lines that show how each model classifies points in your written submission. Return the predictions for both the linear model and the polynomial model and use `contour_plot()` in `hw1_utils.py` to help with the plot. Do both models correctly classify all points?

(Note that red corresponds to larger values and blue to smaller values when using `contour_plot` with the “coolwarm” colormap).

**Using `contour_plot()` effectively:** Define a small prediction function that maps an  $(n \times d)$  tensor of grid points to an  $(n \times 1)$  tensor of scores, then pass it to `contour_plot`. For a linear model with parameters  $\mathbf{w} \in \mathbb{R}^{(d+1) \times 1}$ , one convenient choice is

$$\text{pred\_lin}(Z) = [\mathbf{1} \ Z] \mathbf{w}, \quad \text{where } Z \in \mathbb{R}^{n \times d} \text{ and } \mathbf{1} \in \mathbb{R}^{n \times 1}.$$

For the quadratic model, use your feature map  $\phi(Z)$  constructed in the order specified above and compute  $\text{pred\_poly}(Z) = \phi(Z) \mathbf{w}$ . The decision boundary corresponds to the 0-level contour; you can emphasize it by supplying a `levels` argument (e.g., `levels=[0]`) to draw just that contour.

**Hint:** A “Contour plot” is a way to represent a 3-dimensional surface in a 2-D figure. In this example, the data points are pinned to the figure with their features  $(x_1, x_2)$  as the coordinates in 2-D space (e.g., x and y axis); the third dimension (e.g., the predictions of the data points) is labeled on the points in the figure. The lines or curves that link the grid points with the same predictions together are called the “contours”. See `contour_plot()` in `hw1_utils.py` for details.

**Solution.**

## 4. Logistic Regression.

Recall the empirical risk  $\hat{\mathcal{R}}$  for logistic regression (as presented in lecture 3). Throughout this problem, assume labels  $y_i \in \{-1, +1\}$ :

$$\hat{\mathcal{R}}_{\log}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \ln(1 + \exp(-y_i \mathbf{w}^\top \mathbf{x}_i)).$$

Here you will minimize this risk using gradient descent.

- (a) [hw1] In your written submission, derive the gradient descent update rule for this empirical risk by taking the gradient. Write your answer in terms of the learning rate  $\eta$ , previous parameters  $\mathbf{w}$ , new parameters  $\mathbf{w}'$ , number of examples  $n$ , and training examples  $(\mathbf{x}_i, y_i)$ . Show all of your steps.
- (b) [hw1code] Implement the `logistic()` function in `hw1.py`. You are given as input a training set  $\mathbf{X}$ , training labels  $\mathbf{Y}$ , a learning rate `lr`, and number of gradient updates `num_iter`. Implement gradient descent to find parameters  $\mathbf{w}$  that minimize the empirical risk  $\hat{\mathcal{R}}_{\log}(\mathbf{w})$ . Perform gradient descent for `num_iter` updates with a learning rate of `lr`, initializing  $\mathbf{w} = \mathbf{0}$  and returning  $\mathbf{w}$  as output. Don't forget to prepend  $\mathbf{X}$  with a column of ones. Return  $\mathbf{w} \in \mathbb{R}^{(d+1) \times 1}$  with the bias as the first entry.

**Library routines:** `torch.matmul` (`@`), `X.t()`, `torch.sigmoid`, `torch.special.softplus` (or `torch.logaddexp`), `torch.exp`.

- (c) [hw1] Implement the `logistic_vs_ols()` function in `hw1.py`. Use `hw1_utils.load_logistic_data()` to generate a training set  $\mathbf{X}$  and training labels  $\mathbf{Y}$ . Run `logistic(X,Y)` from part (b) taking  $\mathbf{X}$  and  $\mathbf{Y}$  as input to obtain parameters  $\mathbf{w}$ . Also run `linear_gd(X,Y)` from Problem 2 to obtain parameters  $\mathbf{w}$ . Plot the decision boundaries for your logistic regression and least squares models along with the data  $\mathbf{X}$ . Which model appears to classify the data better? Explain why you believe your choice is the better classifier for this problem.

**Library routines:** `torch.linspace`, `plt.scatter`, `plt.plot`, `plt.show`, `plt.gcf`.

**Hints:**

- The positive and negative points are guaranteed to be linearly separable (though an algorithm may or may not find the optimal line to separate them).
- Decision boundary with bias: let  $\mathbf{z} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}$  denote the augmented feature and  $\mathbf{w}$  include the bias; the boundary is  $\{\mathbf{z} : \mathbf{w}^\top \mathbf{z} = 0\}$ . When plotting in 2D with  $\mathbf{w} = (w_0, w_1, w_2)^\top$ , draw the line  $x_2 = -(w_0 + w_1 x_1)/w_2$  for  $|w_2| > 0$ , or a vertical line at  $x_1 = -w_0/w_1$  if  $|w_2|$  is tiny.
- Average the gradient over examples (divide by  $n$ ); do not sum.
- In order to make the two models significantly different, we recommend that you train the logistic regression with a large `num_iter` (e.g., 1,000,000 or even larger).

**Solution.**

## 5. N-Gram Next Token Prediction.

Recall the empirical risk  $\hat{\mathcal{R}}$  for cross entropy (as presented in lecture 2). In this problem, let  $\mathbf{X} \in \mathbb{R}^{n \times d}$ ,  $\mathbf{W} \in \mathbb{R}^{d \times k}$ , and labels  $Y \in \{0, \dots, k-1\}^{n \times 1}$ . We consider a linear predictor  $f: \mathbb{R}^d \rightarrow \mathbb{R}^k$  given by  $f(\mathbf{x}) = \mathbf{W}^\top \mathbf{x}$  (no bias term):

$$\hat{\mathcal{R}}_{\text{CE}}(\mathbf{W}) = -\frac{1}{n} \sum_{i=1}^n \ln \left( \frac{\exp(f_{y_i}(\mathbf{x}_i))}{\sum_{j=1}^k \exp(f_j(\mathbf{x}_i))} \right).$$

You will minimize this risk using gradient descent, and apply it to next token prediction.

- (a) [hw1] In your written submission, derive the gradient descent update rule for this empirical risk by taking the gradient. Write your answer in terms of the learning rate  $\eta$ , previous parameters  $\mathbf{W}$ , new parameters  $\mathbf{W}'$ , number of examples  $n$ , training examples  $(\mathbf{x}_i, y_i)$ , and probabilities  $p(c|\mathbf{x}_i) = \frac{\exp(f_c(\mathbf{x}_i))}{\sum_{j=1}^k \exp(f_j(\mathbf{x}_i))}$ . Show all of your steps.
- (b) [hw1code] Implement the `cross_entropy()` function in `hw1.py`. You are given as input a training set  $\mathbf{X}$  ( $n \times d$ ), training labels  $\mathbf{Y}$  (integer class indices in  $\{0, \dots, k-1\}$ , shaped  $n \times 1$ ), number of classes  $k$ , a learning rate `lr`, and number of gradient updates `num_iter`. Implement gradient descent to find parameters  $\mathbf{W} \in \mathbb{R}^{d \times k}$  that minimize the empirical risk  $\hat{\mathcal{R}}_{\text{CE}}(\mathbf{W})$ . Perform gradient descent for `num_iter` updates with a learning rate of `lr`, initializing  $\mathbf{W} = 0$  and returning  $\mathbf{W}$  as output. Unlike previous problems, do *not* incorporate a bias term. Average the gradient over the batch (divide by  $n$ ), do not sum. Construct one-hot targets via either `torch.nn.functional.one_hot(Y.view(-1), num_classes=k).float()` or `torch.zeros(n,k).scatter_(1, Y, 1.0)`.  
**Library routines:** `torch.matmul` (`@`), `X.t()`, `torch.softmax`; optionally `torch.logsumexp`/`torch.log_softmax` for stability.
- (c) [hw1code] Implement the `get_ntp_weights()` function in `hw1.py`. You are given as input the context size `n` and the embedding dimension `embedding_dim`. We will use a small subset of the TinyStories dataset (Eldan and Li, 2023) as our text sample, which you can extract using `hw1_utils.load_ntp_data()`. This function will return text data split into tokens `tokenized_data` (our tokenizer treats each word as a token), the list of all tokens in order of their id `sorted_words`, and the inverse mapping of token to id `word_to_idx`. You will then need to create the appropriate N-gram training data from `tokenized_data`. To do so, for every list of words in `tokenized_data` (which is a list of lists of words), then create a training sample from every set of  $n+1$  consecutive words (the first  $n$  words are the context, and the last word is the target). Given a list of  $w$  words, you should create exactly  $w-n$  training samples (assuming  $w \geq n$ , of course); skip sentences shorter than  $n+1$ . Use `hw1_utils.load_random_embeddings()` to get random feature embeddings for each word in the vocabulary. Run `cross_entropy(X, Y, k)` from part (b) taking  $\mathbf{X}$ ,  $\mathbf{Y}$ , and vocabulary size  $k$  as input to obtain parameters  $\mathbf{W}$ . Use the default learning rate and number of iterations.  
**Library routines:** `torch.stack`.
- (d) [hw1code] Implement the `generate_text()` function in `hw1.py`. You are given as input the parameters  $\mathbf{W}$  from the `get_ntp_weights()` function in part (c), the context size `n`, the number of additional tokens to generate `num_tokens`, the embedding dimension `embedding_dim`, and an initial context string `context`. Use `hw1_utils.load_ntp_data()` and `hw1_utils.load_random_embeddings()` to get the rest of the necessary data. Greedy decoding: assume the context contains at least  $n$  tokens; if longer, use the last  $n$  tokens at each step. At each step, map tokens to indices, concatenate embeddings of the last  $n$  indices to form  $\mathbf{x} \in \mathbb{R}^d$  with  $d = n \cdot \text{embedding\_dim}$ , compute logits  $\mathbf{W}^\top \mathbf{x} \in \mathbb{R}^k$ , take `argmax` to pick the next token index, append it, and repeat. Return a string containing the initial context as well as all of the generated words, with each word separated by a space.  
**Library routines:** `torch.argmax`.
- (e) [hw1] Try generating at least 5 strings using different initial context strings using the `generate_text()` function in part (d) (use `n = 4`, `num_tokens`  $\geq 20$ , and `embedding_dim = 10`), and include the results here. Do you notice anything unusual about the generated strings? Why do you think this happens?

**Solution.**

## 6. LLM Use, collaboration, and other sources.

[hw1] Please document, in detail, all your sources, including include LLMs, friends, internet resources, etc. For example:

- 1a. I asked my friend, then I found a different way to derive the same solution.
- 1b. GPT-5 solved the problem in one shot, but then I rewrote it once on paper, and a few days later tried to re-derive an answer from scratch.
- 1c. I accidentally found this via a google search, and had trouble forgetting the answer I found, but still typed it from scratch without copy-paste.
- 1d. ...
- ⋮
- 6. I used my solution to problem 5 to write this answer.

**Answer.**



## References

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english?, 2023. URL <https://arxiv.org/abs/2305.07759>.