

# Homework 1 – Deep Learning (CS/DS541, Whitehill, Fall 2024)

**Collaboration policy:** You may complete this assignment with a partner, if you choose. In that case, both partners should sign up on Canvas as a “team”, and only one of you should submit the assignment. You are permitted to use ChatGPT to help you with the plotting aspects of this assignment – i.e., how to call `matplotlib` to render the requested scatter plots, contour plots, etc. No other use of ChatGPT is permitted.

1. **Python and Numpy Warm-up Exercises [20 pts]:** This part of the homework is intended to help you practice your linear algebra and how to implement linear algebraic and statistical operations in Python using `numpy` (to which we refer in the code below as `np`). For each of the problems below, write a method (e.g., `problem_1a`) that returns the answer for the corresponding problem.

In all problems, you may assume that the dimensions of the matrices and/or vectors that are given as input are compatible for the requested mathematical operations. You do not need to perform error-checking.

**Note 1:** In mathematical notation we usually start indices with  $j = 1$ . However, in `numpy` (and many other programming settings), it is more natural to use 0-based array indexing. When answering the questions below, do not worry about “translating” from 1-based to 0-based indexes. For example, if the  $(i, j)$ th element of some matrix is requested, you can simply write `A[i, j]`.

**Note 2:** To represent and manipulate vectors and matrices, please use `numpy`’s `array` class (*not* the `matrix` class).

**Note 3:** While the difference between a row vector and a column vector is important when doing math, `numpy` does not care about this difference *as long as the array is 1-D*. This means, for example, that if you want to compute the inner product between two vectors `x` and `y`, you can just write `x.dot(y)` without needing to transpose the `x`. If `x` and `y` are 2-D arrays, however, then it *does* matter whether they are row-vectors or column-vectors, and hence you might need to transpose accordingly.

- (a) Given two matrices `A` and `B`, compute and return an expression for `A + B`. [ 0 pts ]  
*Answer:* While it is completely valid to use `np.add(A, B)`, this is unnecessarily verbose; you really should make use of the “syntactic sugar” provided by Python’s/`numpy`’s operator overloading and just write: `A + B`. Similarly, you should use the more compact (and arguably more elegant) notation for the rest of the questions as well.
- (b) Given matrices `A`, `B`, and `C`, compute and return `AB - C` (i.e., right-multiply matrix `A` by matrix `B`, and then subtract `C`). Use `dot` or `np.dot`. [ 1 pts ]
- (c) Given matrices `A`, `B`, and `C`, return `A ⊙ B + CT`, where  $\odot$  represents the element-wise (Hadamard) product and  $\top$  represents matrix transpose. In `numpy`, the element-wise product is obtained simply with `*`. [ 1 pts ]
- (d) Given column vectors `x` and `y`, compute the inner product of `x` and `y` (i.e., `xTy`). [ 1 pts ]
- (e) Given matrix `A` and integer `i`, return the sum of all the entries in the  $i$ th row *whose column index is even*, i.e.,  $\sum_{j:j \text{ is even}} A_{ij}$ . Do **not** use a loop, which in Python can be very slow. Instead use the `np.sum` function. [ 2 pts ]
- (f) Given matrix `A` and scalars `c, d`, compute the arithmetic mean over all entries of `A` that are between `c` and `d` (inclusive). In other words, if  $S = \{(i, j) : c \leq A_{ij} \leq d\}$ , then compute  $\frac{1}{|S|} \sum_{(i,j) \in S} A_{ij}$ . Use `np.nonzero` along with `np.mean`. [ 2 pts ]
- (g) Given an  $(n \times n)$  matrix `A` and integer `k`, return an  $(n \times k)$  matrix containing the right-eigenvectors of `A` corresponding to the  $k$  eigenvalues of `A` with the largest absolute value. Use `np.linalg.eig`. [ 2 pts ]

- (h) Given a column vector (with  $n$  components)  $\mathbf{x}$ , an integer  $k$ , and positive scalars  $m, s$ , return an  $(n \times k)$  matrix, each of whose columns is a sample from multidimensional Gaussian distribution  $\mathcal{N}(\mathbf{x} + m\mathbf{z}, s\mathbf{I})$ , where  $\mathbf{z}$  is column vector (with  $n$  components) containing all ones and  $\mathbf{I}$  is the identity matrix. Use either `np.random.multivariate_normal` or `np.random.randn`. [ 2 pts ]
- (i) Given a matrix  $\mathbf{A}$  with  $n$  rows, return a matrix that results from **randomly permuting** the columns (but not the rows) in  $\mathbf{A}$ . [ 2 pts ]
- (j) Z-scoring: Given a vector  $\mathbf{x}$ , return a vector  $\mathbf{y}$  such that each  $y_i = (x_i - \bar{x})/\sigma$ , where  $\bar{x}$  is the mean (use `np.mean`) of the elements of  $\mathbf{x}$  and  $\sigma$  is the standard deviation (use `np.std`). [ 2 pts ]
- (k) Given an  $n$ -vector  $\mathbf{x}$  and a non-negative integer  $k$ , return a  $n \times k$  matrix consisting of  $k$  copies of  $\mathbf{x}$ . You can use numpy methods such as `np.newaxis`, `np.atleast_2d`, and/or `np.repeat`. [ 2 pts ]
- (l) Given a  $k \times n$  matrix  $\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} & \dots & \mathbf{x}^{(n)} \end{bmatrix}$  and a  $k \times m$  matrix  $\mathbf{Y} = \begin{bmatrix} \mathbf{y}^{(1)} & \dots & \mathbf{y}^{(m)} \end{bmatrix}$ , compute an  $n \times m$  matrix  $\mathbf{D} = \begin{bmatrix} d_{11} & \dots & d_{1m} \\ & \ddots & \\ d_{n1} & \dots & d_{nm} \end{bmatrix}$  consisting of all pairwise  $L_2$  distances  $d_{ij} = \|\mathbf{x}^{(i)} - \mathbf{y}^{(j)}\|_2$ . In this problem you may **not** use loops. Instead, you can avail yourself of numpy objects & methods such as `np.newaxis`, `np.atleast_3d`, `np.repeat`, `np.swapaxes`, etc. (There are various ways of solving it.) **Hint:** from  $\mathbf{X}$  (resp.  $\mathbf{Y}$ ), construct a 3-d matrix that contains multiple copies of each of the vectors in  $\mathbf{X}$  (resp.  $\mathbf{Y}$ ); then subtract these 3-d matrices. [ 3 pts ]

## 2. Training 2-Layer Linear Neural Networks with Stochastic Gradient Descent [25 pts]:

- (a) Train an age regressor that analyzes a  $(48 \times 48 = 2304)$ -pixel grayscale face image and outputs a real number  $\hat{y}$  that estimates how old the person is (in years). The training and testing data are available here:

- [https://s3.amazonaws.com/jrwprojects/age\\_regression\\_Xtr.npy](https://s3.amazonaws.com/jrwprojects/age_regression_Xtr.npy)
- [https://s3.amazonaws.com/jrwprojects/age\\_regression\\_ytr.npy](https://s3.amazonaws.com/jrwprojects/age_regression_ytr.npy)
- [https://s3.amazonaws.com/jrwprojects/age\\_regression\\_Xte.npy](https://s3.amazonaws.com/jrwprojects/age_regression_Xte.npy)
- [https://s3.amazonaws.com/jrwprojects/age\\_regression\\_yte.npy](https://s3.amazonaws.com/jrwprojects/age_regression_yte.npy)

Your prediction model  $g$  should be a 2-layer linear neural network that computes  $\hat{y} = g(\mathbf{x}; \mathbf{w}) = \mathbf{x}^\top \mathbf{w} + b$ , where  $\mathbf{w}$  is the vector of weights and  $b$  is the bias term. The cost function you should optimize is

$$f_{\text{MSE}}(\mathbf{w}, b) = \frac{1}{2n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2$$

where  $n$  is the number of examples in the training set  $\mathcal{D}_{\text{tr}} = \{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$ , each  $\mathbf{x}^{(i)} \in \mathbb{R}^{2304}$  and each  $y^{(i)} \in \mathbb{R}$ . To optimize the weights, you should implement stochastic gradient descent (SGD).

**Note:** you must complete this problem using only linear algebraic operations in **numpy** – you may **not** use any off-the-shelf linear regression or neural network software, as that would defeat the purpose.

There are several different hyperparameters that you will need to optimize:

- Mini-batch size  $\tilde{n}$ .
- Learning rate  $\epsilon$ .
- Number of epochs.

In order not to cheat (in the machine learning sense) – and thus overestimate the performance of the network – it is crucial to optimize the hyperparameters **only** on a *validation set*. (The training set would also be acceptable but typically leads to worse performance.) To create

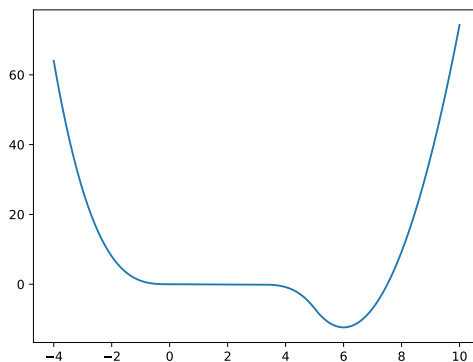
a validation set, simply set aside a fraction (e.g., 20%) of the `age_regression.Xtr.npy` and `age_regression.ytr.npy` to be the validation set; the remainder (80%) of these data files will constitute the “actual” training data. While there are fancier strategies (e.g., Bayesian optimization) that can be used for hyperparameter optimization, it’s often effective to just use a grid search over a few values for each hyperparameter. In this problem, you are required to explore systematically (e.g., using nested `for` loops) at least 2 different values for each hyperparameter.

**Performance evaluation:** Once you have tuned the hyperparameters and optimized the weights and bias term so as to minimize the cost on the validation set, then: (1) **stop** training the network and (2) evaluate the network on the **test** set. Report both the training and the test  $f_{\text{MSE}}$  in the PDF document you submit, as well as the training cost values for the last 10 iterations of gradient descent (just to show that you actually executed your code).

3. **Gradient descent: what can go wrong?** [30 pts] Please enter your code in a file named `homework1_problem3.py`, with one Python function (e.g., `problem3a`) for each subproblem.

(a) [10 pts]: The graph below plots the function  $f(x)$  that is defined piece-wise as:

$$f(x) = \begin{cases} -x^3 & : x < -0.1 \\ -3x/100 - 1/500 & : -0.1 \leq x < 3 \\ -(x - 31/10)^3 - 23/250 & : 3 \leq x < 5 \\ \frac{1083}{200}(x - 6)^2 - 6183/500 & : x \geq 5 \end{cases}$$

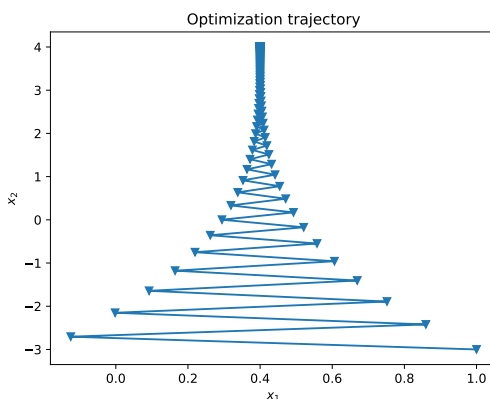


As you can see, the function has a long nearly flat section (sometimes known as a *plateau*) just before the minimum.<sup>1</sup> Plateaux can cause big problems during optimization. To show this:

- i. Derive by hand the (piece-wise) function  $\nabla f$  and implement it in Python/numpy.
  - ii. Use your implementation of  $\nabla f$  to conduct gradient descent for  $T = 100$  iterations. Always start from an initial  $x = -3$ . Try using various learning rates: `1e-3`, `1e-2`, `1e-1`, `1e0`, `1e1`. Plot  $f$ ,  $\nabla f$ , as well as superimposed dots that show the sequence  $((x^{(1)}, y^{(1)}), \dots, (x^{(T)}, y^{(T)}))$  of gradient descent. Use `plt.legend` to indicate which scatter plot corresponds to which learning rate.
  - iii. Describe in 1-2 sentences what you observe during gradient descent for the set of learning rates listed above.
  - iv. Find a learning rate  $\epsilon$  for which gradient descent successfully converges to  $\min f(x)$ , and report  $\epsilon$  in the PDF file.
- (b) [8 pts]: Even a convex paraboloid – i.e., a parabola in multiple dimensions that has only one local minimum and no plateaux – can cause problems for “vanilla” SGD (i.e., the kind we’ve learned in

<sup>1</sup>The ugly constants in  $f$  were chosen to give rise to these characteristics while ensuring that it remains differentiable.

class so far). Examine the scatter-plot below which shows the sequence  $((\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(T)}, y^{(T)}))$  of gradient descent on a convex paraboloid  $f$ , starting at  $\mathbf{x}^{(1)} = [1, -3]^\top$ , where each  $\mathbf{x} \in \mathbb{R}^2$ . The descent produces a zig-zag pattern that takes a long time to converge to the local minimum.



- i. Speculate how the SGD trajectory would look if the learning rate were made to be very small (e.g., 100x smaller than in the figure above).
  - ii. Let  $f(x_1, x_2) = a_1(x_1 - c_1)^2 + a_2(x_2 - c_2)^2$ . Pick values for  $a_1, a_2, c_1, c_2$  so that – when the scatter-plot shown above is superimposed onto it – the gradient descent is realistic for  $f$ . Rather than just guess randomly, consider: why would the zig-zag be stronger in one dimension than another, and how would this be reflected in the function’s constants? Plot a contour graph using `plt.contour` and superimpose the scatter-plot using `plt.scatter`. You can find the gradient descent sequence in `gradient_descent_sequence.txt`. Note that you are *not* required to find the exactly correct constants or even to estimate them algorithmically. Rather, you can combine mathematical intuition with some trial-and-error. Your solution should just look visually “pretty close” to get full credit.
- (c) [6 pts]: This problem is inspired by this paper. Consider the function  $f(x) = \frac{2}{3}|x|^{3/2}$ . Derive  $\nabla f$ , implement gradient descent and plot the descent trajectories  $((x^{(1)}, y^{(1)}), \dots, (x^{(T)}, y^{(T)}))$  for a variety of learning rates `1e-3`, `1e-2`, `1e-1` and a variety of starting points. See what trend emerges, and report it in the PDF.
- (d) [6 pts]: While very (!) unlikely, it is theoretically possible for gradient *descent* to converge to a local *maximum*. Give the formula of a function (in my own solution, I used a degree-4 polynomial), a starting point, and a learning rate such that gradient descent converges to a local maximum after 1 descent iteration (i.e., after 1 iteration, it reaches the local maximum, and the gradient is exactly 0). Prove (by deriving the exact values for the descent trajectory) that this is true. You do *not* need to implement this in code (and, in fact, due to finite-precision floating-point arithmetic, it might not actually converge as intended).

**Submission:** Create a Zip file containing both your Python and PDF files, and then submit on Canvas. If you are working as part of a group, then only **one** member of your group should submit (but make sure you have already signed up in a pre-allocated team for the homework on Canvas).