

Accuracy for perceptron (with flip probability 0.1):

0.7602

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

Solution: 0.75

Accuracy for averaged perceptron (with flip probability 0.1):

0.805

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

Solution: 0.8

Accuracy for perceptron (with flip probability 0.25):

0.6046

Submit

View Answer

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

Accuracy for averaged perceptron (with flip probability 0.25):

0.6375

Submit

View Answer

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

Modify your `eval_learning_alg` so that it tests hypothesis on the training data instead of generating a new test data set. Run enough trials that you can confidently predict this "training accuracy" for the two learning algorithms. Note the differences from your results above.

Accuracy for perceptron (with flip probability 0.1) on training data:

0.8298

Submit

View Answer

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

Accuracy for averaged perceptron (with flip probability 0.1) on training data:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: 0.87

Accuracy for perceptron (with flip probability 0.25) on training data:

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Accuracy for averaged perceptron (with flip probability 0.25) on training data:

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

In the first part of this assignment, we will consider several general issues in representing features and their impact on classification. In the second part of the assignment, we will experiment with these strategies in the context of realistic data sets. Please make sure you read the [lecture notes covering feature representations](#) for this assignment.

# Feature Transformations

A code file that is required for this assignment can be found [here](#), and a [colab notebook here](#).

## 1) Scaling

Consider a linearly separable dataset with two features:

```
data = ([[200, 800, 200, 800],
         [0.2, 0.2, 0.8, 0.8]])
labels = [[-1, -1, 1, 1]]
```

Consider the separator defined by  $\theta = (0, 1), \theta_0 = -0.5$ .

In order to apply the perceptron mistake bound ([see notes](#)), we transform our problem from  $\theta^T x + \theta_0 = 0$  to some  $\theta'^T x = 0$ . We do this by appending  $\theta_0$  to  $\theta$ , and appending 1 to  $x$ , as follows:

$$\theta'^T x = [\theta_1 \quad \theta_2 \quad \dots \quad \theta_0] \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ 1 \end{bmatrix} = 0$$

In this phrasing, our new " $\theta$ " is  $(0, 1, -0.5)$ . For a separator through the origin, recall that [the margin of the data set](#) is the minimum of  $\gamma = y^{(i)}(\theta^T x^{(i)})/\|\theta\|$  over all data points  $(x^{(i)}, y^{(i)})$ .

For the following questions, assume we are working in the transformed (3d) feature space, with perceptron through the origin, and where if the data has bounded magnitude  $R$ , then the theoretical upper bound on mistakes made by perceptron is  $(\frac{R}{\gamma})^2$ , for a separable data set.

You are free to use your perceptron algorithm implemented in the previous homework to answer the following questions. (Some parts require more runs of the perceptron algorithm than one could reasonably perform by hand.)

**1A)**

What is the margin  $\gamma$  of this data set with respect to that separator (up to 3 decimal places)?





100.00%

You have infinitely many submissions remaining.

**1B)**

What is the theoretical bound on the number of mistakes perceptron will make on this problem?

[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

Solution: 8910693

**Explanation:**

$R$  is approximately 800 from  $\sqrt{800^2 + 0.8^2 + 1^2}$ , then mistakes are bounded by  $(R/\gamma)^2$ . Recall that we found  $\gamma$  in the previous part to be 0.268 which we plug in here.  $(800/0.268)^2$  is approximately 8910700 (exact value is 8910693).

**1C)**

How many mistakes does perceptron through origin have to make in order to find a perfect separator on the data provided above, in the order given? (Try it on your computer, not by hand!)

[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

Solution: 666696

**Explanation:**

Running the code from the previous perceptron homework, we obtain 666696.  
A common pitfall is not running the code for enough iterations.

**1D)**

If we were to multiply both original features of all of the points by .001, and considered the separator through origin  $\theta = (0, 1, -0.0005)$ , what would the margin of the new dataset be?

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

**1E)**

How would the performance of the perceptron (as predicted by the mistake bound) change?

More mistakes ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1F)

If we multiplied just the first original feature (first row of the data) by .001, and used our original separator, what would the new margin be? 0.268328

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1G)

What would the mistake bound be in this case? 32

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution: 32

**Explanation:**

Now  $R = \sqrt{(1^2 + 0.8^2 + 0.8^2)} = 1.51$

While there was no effect on the margin  $\gamma$ ,  $R$  actually decreased significantly under this transformation, because the element that contributed the most to its value (first coordinate of  $x$ ) has been scaled down.

Hence the mistake bound dropped by a lot!

1H)

Run the perceptron algorithm on this data; how many mistakes does it make?

7

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 2) Encoding Discrete Values

Some data sets have features that take on discrete values drawn from a set. Examples might be:

- which section of a class a student is in (1, 2, 3, 4)
- manufacturer of a cell phone (Samsung, Xiaomi, Sony, Apple, LG, Nokia)
- which laboratory performed a particular medical test

Sometimes they already have an obvious encoding into integers; other times, they don't but it's easy to make one (e.g., Samsung = 1, Xiaomi = 2, Sony = 3, Apple = 4, LG = 5, Nokia = 6)

**2A)** Let's consider the case of the cell phones, using the encoding above, and imagine there is some prediction problem, such as predicting whether the phone will last three years, for which we have the data set:

```
data = [[2, 3, 4, 5]]
labels = [[1, 1, -1, -1]]
```

What value of  $\theta$  and  $\theta_0$  would we get when running perceptron on this data? You are free to use the perceptron implemented in homework 2.

Enter a Python list with two floats, one for  $\theta$  and one for  $\theta_0$ . [-2, 7]

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2B)** What prediction would we make about other phone types based on this classifier?

Enter a Python list with two labels (1 or -1), the first one for a Samsung phone and the second for a Nokia phone.

[1, -1]

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2C)**

Are these predictions meaningful given the training data we used? No ▾

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2D)** It is common to encode a feature which takes on a value from a set of discrete values, not as a single multi-valued feature, but using a *one hot* encoding.

Here, assume you have a feature  $f$  which can take on any value from the set  $\{1, 2, \dots, k\}$ . If  $f$  takes on value  $i$ , then we represent it as a vector of length  $k$  of all zeros, except for a +1 at the  $i$ th coordinate.

Write a function `one_hot` that takes as input  $x$ , a single feature value (between 1 and  $k$ ), and  $k$ , the total possible number of values this feature can take on, and transform it to a numpy column vector of  $k$  binary features using a one-hot encoding (remember vectors have zero-based indexing).

For example, `one_hot(3,7)` should return a column vector of length 7 with the entry at index 2 taking value 1 (indices start at 0) and other entries taking value 0.

```

1 import numpy as np
2
3 def one_hot(x, k):
4     out = [1 if i==(x-1) else 0 for i in range(k)]
5     return np.array([out]).T

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2E)** What happens if we use one-hot encoding on the data set part 2A) above, and put it into the perceptron? Recall that for a classifier  $h(x)$ , the prediction is  $+1$  if  $h(x) > 0$  and  $-1$  otherwise. Further note that the perceptron algorithm makes an update whenever  $y^{(i)}(\theta^T x^{(i)} + \theta_0) \leq 0$ .

**2E i)** What is the separator produced by the perceptron algorithm?

Enter a Python list with 7 floats, six for  $\theta$  and one for  $\theta_0$ .

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution:  $[0, 2, 1, -2, -1, 0, 0]$

#### Explanation:

We map each phone to a separate dimension and update the offset by one for either a positive or negative example to obtain that  $\theta = [0, 2, 1, -2, -1, 0]$  and  $\theta_0 = 0$

**2E ii)** What are the predictions for Samsung and Nokia?

Enter a Python list with two labels (1 or -1), the first one for a Samsung phone and the second for a Nokia phone.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2E iii)** What are the distances for the Samsung and Nokia data points from the separator?

Enter a Python list with two distances, the first one for a Samsung phone and the second for a Nokia phone.




**100.00%**

*You have infinitely many submissions remaining.*

**2F)** Now, what if we have this dataset:

```
data = [[1, 2, 3, 4, 5, 6]]
labels = [[1, 1, -1, -1, 1, 1]]
```

Is it linearly separable in the original encoding?



**100.00%**

*You have infinitely many submissions remaining.*

**2G)** Is it linearly separable in the one-hot encoding? If so, provide the separator found by the perceptron.

Enter a Python list with 7 floats, six for  $\theta$  and one for  $\theta_0$  or 'none'




**100.00%**

*You have infinitely many submissions remaining.*

**2H)** Enter an assignment of data values to labels (with distinct data points) that is not linearly separable using the one-hot encoding, or enter None if no such assignment exists.

Enter a Python list with 6 tuples (value, label) or 'none'



**100.00%**

*You have infinitely many submissions remaining.*

### 3) Polynomial Features

One systematic way of generating non-linear transformations of your input features is to consider the polynomials of increasing order. Given a feature vector  $x = [x_1, x_2, \dots, x_d]^T$ , we can map it into a new feature vector that contains all the factors in a polynomial of order  $d$ . For example, for  $x = [x_1, x_2]^T$  and order 2, we get

$$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]^T$$

and for order 3, we get

$$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1^2x_2, x_1x_2^2, x_1^3, x_2^3]^T.$$



In the code file, we have defined `make_polynomial_feature_fun` that, given the order, returns a feature transformation function (analogous to  $\phi$  in the description). You should use it in doing this problem.

**3A)**

Enter a list of 6 integers indicating the number of polynomial features of degrees [1, 10, 20, 30, 40, 50] for a 2-dimensional feature vector.

**100.00%**

*You have infinitely many submissions remaining.*

**3B)** Consider this data-set of four points in two-dimensional space:

```
data = ([[1, 1, 2, 2],
         [1, 2, 1, 2]])
labels = [[-1, 1, 1, -1]]
```

It is standardly called the "exclusive-or" or "xor" problem. These points are not linearly separable, and you could interpret each point as being a pair of truth values, with their label being the XOR of the values.

In the code file, we have defined 4 sample data sets, (1) `super_simple_separable_through_origin`, (2) `super_simple_separable`, (3) `xor`, and (4) `xor_more`. On your own machine, you should run the code we have provided (`test_with_features`) for various orders of polynomial features and enter below the order of the smallest feature that separates the data. Make sure that you have included your implementation of `perceptron` in that file or you can use the implementation we have provided. You may need to adjust the number of iterations that the perceptron runs.

The separators are displayed when the code runs; it's instructive to watch them to see the range of separators that these non-linear transformations produce. Note that the separators are drawn by evaluating the feature transformations on a grid of points in the feature space and using the separator to classify them. (Note: If you have issues with the graphic not moving forward, try pressing the keys within your terminal.)

Enter a Python list of integers indicating the smallest polynomial order for which a separator exists for each of the four datasets in the code file (in order).

**100.00%**

*You have infinitely many submissions remaining.*

Solution: [1, 1, 2, 3]

## Experiments

A code and data folder that will be necessary for doing this homework can be found at the top of the page. In the file [code\\_for\\_hw3\\_part2.py](#), include your learner code from HW 2. **You will want to modify the evaluation algorithms so that they take a  $T$  argument to pass to the learners.**

The rest of this assignment will require running the code on your computer; we will be asking only for the results of your runs.

## 4) Evaluating algorithmic and feature choices for AUTO data

We now want to build a classifier for the auto data, with a focus on the numeric data. In the `code_for_hw3_part2.py`, we have supplied you with the `load_auto_data` function, which can read the relevant `.tsv` file. It returns a list of dictionaries, one for each data item.

We then specify what feature function to use for each column in the data. The file `hw3_part2_main.py` has an example that constructs the data and label arrays using `raw` features for all the columns.

In the list `features` of `hw3_part2_main.py`, you will find a list of feature name, feature function tuples. There are three options for feature functions: `raw`, `standard` and `one_hot`. `raw` uses the original value; `standard` subtracts out the mean value and divides by the standard deviation; and `one_hot` will one-hot encode the input, as described in the notes.

The function `auto_data_and_labels` processes the dictionaries and return `data`, `labels`. `data` has dimension  $(d, 392)$ , where  $d$  is the total number of features specified, and `labels` has dimension  $(1, 392)$ . The data in the file is sorted by class, but it will be shuffled when loaded.

We have included staff implementations of `perceptron` and `average perceptron` in `code_for_hw3_part2.py`. Using the feature arrays and these implementations, you will be able to compute  $\theta$  and  $\theta_0$ .

We have also included staff implementations of `eval_classifier` and `xval_learning_alg` (in the same code file). You should use these functions to report accuracies.

### 4.1) Making choices

We know of two algorithm classes: `perceptron` and `averaged perceptron` (which we implemented in HW 1). We have a several parameters that specify the settings for these learning algorithms.

**A)** Which parameters should we use for the learning algorithm? In the `perceptron` and `averaged perceptron`, there is a single parameter,  $T$ , the number of iterations.

**B)** Which features should we use? We have lots of choices here: we can use any subset of the data columns and for each column we have choices of how to compute features.

**C)** We will use expected accuracy, estimated by 10-fold cross-validation (we have included the definition in the code file), to make these choices of parameters.

- We will try two types of algorithms: `perceptron` and `averaged perceptron`.
- We will try 3 values of  $T$ :  $T = 1$ ,  $T = 10$ ,  $T = 50$ .
- We will try 2 feature sets:
  1. `[cylinders=raw, displacement=raw, horsepower=raw, weight=raw, acceleration=raw, origin=raw]`
  2. `[cylinders=one_hot, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one_hot]`

Perform 10-fold cross-validation for all combinations of the two algorithms, three  $T$  values, and the two choices of feature sets. It will be worthwhile investing in a piece of code to carry out all of the evaluations, in case you need to do this more than once.

In general, you should shuffle the dataset before evaluating, but for this exercise, please use `hw3.xval_learning_alg`, which shuffles the dataset for you, so that your results match ours.

**4.1C i)**

Enter accuracies (perceptron, averaged perceptron) for  $T=1$ , feature set 1:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (0.653, 0.844)

**4.1C ii)**

Enter accuracies (perceptron, averaged perceptron) for  $T=1$ , feature set 2:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (0.791, 0.9)

**4.1C iii)**

Enter accuracies (perceptron, averaged perceptron) for  $T=10$ , feature set 1:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (0.742, 0.837)

**4.1C iv)**

Enter accuracies (perceptron, averaged perceptron) for T=10, feature set 2:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (0.806, 0.898)

#### 4.1C v)

Enter accuracies (perceptron, averaged perceptron) for T=50, feature set 1:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (0.691, 0.837)

#### 4.1C vi)

Enter accuracies (perceptron, averaged perceptron) for T=50, feature set 2:

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (0.806, 0.901)

Now we have the data we need to make rational choices.

#### 4.1D) Which algorithm class is typically more effective?

Pick one: **Averaged Perceptron** ▼

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

**4.1E)** For the better algorithm, which combination of  $T$  and feature would you use? Consider expected accuracy as of primary importance, take into account running time for near ties in accuracy.

Enter a tuple of two integers (T, feature\_set):

[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

Solution: '(1, 2) or (10, 2)'

#### Explanation:

(1, 2) is the better choice because it gets nearly as good accuracy as using more iterations, but does far less work. We accept (10, 2) as well because before we did staff revisions to this question, it was better, and we forgot to fix the answer!

## 4.2) Analysis

**4.2 A)** For the best algorithm type, best  $T$  and best feature set, construct your best classifier  $(\theta, \theta_0)$  using all the data. Based on the values of the coefficients, which feature has the most impact on the output predictions?

Choose the name of one feature:

[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

Solution: cylinders or weight, depending on how you compute this

**4.2 B) (Optional)** Is there any set of two features you can use to attain comparable results as your best accuracy? What are they?

## 5) Evaluating algorithmic and feature choices for review data

In [the code file](#) (and the [colab notebook](#)), we have supplied you with the `load_review_data` function, that can be used to read a .tsv file and return the labels and texts. We have also supplied you with the `bag_of_words` function, which takes the raw data and returns a dictionary of unigram words. The resulting dictionary is an input to `extract_bow_feature_vectors` which computes a feature matrix of ones and zeros that can be used as the input for the classification algorithms. The file `hw3_part2_main.py` has code for constructing the data and label arrays. Using these arrays and our implementation of the learning algorithms, you will be able to compute  $\theta$  and  $\theta_0$ . You will need to add your (or the one written by staff) implementation of perceptron and averaged perceptron.

### 5.1) Making choices

We have two algorithm classes: perceptron and averaged perceptron. We have a couple of choices of parameters to make to completely specify the learning algorithms.

**5.1A)** Which parameters should we use for the learning algorithm? In the perceptron and averaged perceptron, there is a single parameter,  $T$ , the number of iterations.

**5.1B)** Which features should we use? We could do variations of bag-of-words, for example, simply indicating whether a word is present or, alternatively, using a count of how many times it is present. We can also remove commonly used words with little information; the code distribution includes a file of those words: `stopwords.txt`. You're welcome to explore these on your own; we'll use only a binary indicator for all the words.

**5.1C)** Perform 10-fold cross-validation for all combinations of the two algorithms and three  $T$  values (1, 10, 50). Record the accuracies for each combination (there should be 6 values total).

**Note:** These tests may take a couple of minutes to run; don't expect instant response, but it shouldn't run for 10 minutes.

Now we have the data we need to make rational choices.

**5.1D)** Which algorithm class is typically more effective?

Pick one: Averaged Perceptron ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**5.1E)** For the better algorithm, which value of  $T$  would you use? Consider expected accuracy as of primary importance, take into account running time for near ties in accuracy.

Enter a value of T:

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution: 10

**5.1F)** For the better algorithm and best value of  $T$ , what is your accuracy?

Enter a number between 0 and 1:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 5.2) Analysis

For the best algorithm and best  $T$ , construct your best classifier  $(\theta, \theta_0)$  using all the data.

**Note:** We have included a function called `reverse_dict` in `code_for_hw3_part2.py` that you may find convenient. You are not required to use this function.

**5.2A)** What are the 10 most positive words in the dictionary, that is, the words that contribute most to a positive prediction?

Enter a Python list of ten strings: `['great', 'delicious', 'perfect', 'excellent', 'satisfied', 'yummy', 'easily']`

[Submit](#)[View Answer](#)[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

**5.2B)** What are the 10 most negative words in the dictionary, that is, the words that contribute most to a negative prediction.

Enter a Python list of ten strings: `['worst', 'awful', 'poor', 'horrible', 'unfortunately', 'formula', 'bland', '']`

[Submit](#)[View Answer](#)[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

**5.2C) (Optional)** You might find it amusing to find the most positive and most negative reviews. That is, ones with the most positive and negative signed distance to the hyperplane.

## 6) Evaluating features for MNIST data

This problem explores how well the perceptron algorithm works to [classify images of handwritten digits](#), from the well-known ("MNIST") dataset, building on your thoughts from lab about extracting features from images. This exercise will highlight how important feature extraction is, before linear classification is done, using algorithms such as the perceptron.

### Dataset setup

Often, it may be easier to work with a vector whose spatial orientation is preserved. In previous parts, we have represented features as one long feature vector. For images, however, we often represent a  $m$  by  $n$  image as a  $(m, n)$  array, rather than a  $(mn, 1)$  array (as the previous parts have done).

In the code file, we have supplied you with the `load_mnist_data` function, which will read from the provided image files and populate a dictionary, with image and label vectors for each numerical digit from 0 to 9. These images are already shaped as  $(m, n)$  arrays.

### 6.1) Feature extraction

In the real world, there may be complicated ways to extract meaningful features from images, but in this section we will explore several simple methods.

**6.1A)** You may notice that some numbers (like 8) take up more horizontal space than others (like 1). We can compute a feature based on the average value in each row. Write a Python function that takes in a  $(m, n)$  array and returns a  $(m, 1)$  array, where element  $i$  is the average value in row  $i$ .

```

2
3 def row_average_features(x):
4     """
5     @param x (m,n) array with values in (0,1)
6     @return (m,1) array where each entry is the average of a row
7     """
8     m, n = x.shape
9     row_avgs = np.zeros((m, 1))
10    for row in range(m):
11        row_sum = 0
12        for col in range(n):
13            row_sum += x[row, col]
14        row_avgs[row, 0] = row_sum/n
15    return row_avgs
16
17

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**6.1B)** We can also compute a feature based on the average value in each column. Write a Python function that takes in a  $(m,n)$  array and returns a  $(n,1)$  array, where element  $j$  is the average value in column  $j$ .

```

3 def col_average_features(x):
4     """
5     @param x (m,n) array with values in (0,1)
6     @return (n,1) array where each entry is the average of a column
7     """
8     x = x.T
9     m, n = x.shape
10    row_avgs = np.zeros((m, 1))
11    for row in range(m):
12        row_sum = 0
13        for col in range(n):
14            row_sum += x[row, col]
15        row_avgs[row, 0] = row_sum/n
16    return row_avgs
17
18

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**6.1C)** Finally, you may notice that some features are more "top heavy" while others are more "bottom heavy." Write a function that takes in a  $(m,n)$  array and returns a  $(2,1)$  array, where the first element is the average value in the top half of the image, and the second element is the average value in the bottom half of the image.

You may use the `cv` function from homework 1.



```

7   top half of the image = rows 0 to floor(m/2) [exclusive]
8   and the second entry is the average of the bottom half of the image
9   = rows floor(m/2) [inclusive] to m
10  """
11  m, n = x.shape
12  mid_row = m//2
13  top_sum = 0
14  bot_sum = 0
15  for col in range(n):
16      for row in range(mid_row):
17          top_sum += x[row, col]
18      for row in range(mid_row, m):
19          bot_sum += x[row, col]
20  return np.array([[top_sum/(mid_row*n)], [bot_sum/((m-mid_row)*n)]])
21

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**Important:** In `hw3_part2_main.py`, you may need to **modify** these functions to accept a data matrix of size  $(n, 28, 28)$ , or you may use these functions in other ways (loops are allowed).

## 6.2) Feature evaluation

We can use these features to distinguish between numbers.

**Important:** For this section, we will be using  $T=50$  with the base perceptron algorithm to train our classifier and 10-fold cross validation to evaluate our classifier. A function called `get_classification_accuracy` has already been implemented for you in `code_for_hw3_part2` to compute the accuracy, given your selected data and labels.

You *must* use our implementation of cross validation to report accuracies (you may call `hw3.xval_learning_alg`).

**6.2A)** First we will find baseline accuracies using the raw 0-1 features.

Convert each image into a  $(28*28, 1)$  vector for input into the perceptron algorithm. **Hint:** `np.reshape` may be helpful here.

Run the perceptron on four tasks: 0 vs. 1, 2 vs. 4, 6 vs. 8, and 9 vs. 0.

Enter a list of accuracies [0 vs. 1, 2 vs. 4, 6 vs. 8, 9 vs. 0]:

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution: `[0.975, 0.8641666666666665, 0.9479166666666667, 0.6470833333333333]`

Now let's evaluate the features we extracted.

**6.2B)** Using the extracted features from above, run the perceptron algorithm on the set of 0 vs. 1 images.

Enter a list of accuracies [row, column, top/bottom]:

**100.00%**

*You have infinitely many submissions remaining.*

**6.2C)** Using the extracted features from above, run the perceptron algorithm on the set of 2 vs. 4 images.

Enter a list of accuracies [row, column, top/bottom]:

**100.00%**

*You have infinitely many submissions remaining.*

**6.2D)** Using the extracted features from above, run the perceptron algorithm on the set of 6 vs. 8 images.

Enter a list of accuracies [row, column, top/bottom]:

**100.00%**

*You have infinitely many submissions remaining.*

Solution: [0.92125, 0.52125, 0.5650000000000001]

**6.2E)** Using the extracted features from above, run the perceptron algorithm on the set of 9 vs. 0 images.

Enter a list of accuracies [row, column, top/bottom]:

**100.00%**

*You have infinitely many submissions remaining.*

Solution: [0.49749999999999994, 0.5041666666666667, 0.49749999999999994]

**6.2F) (Optional)** What does it mean if a binary classification accuracy is below 0.5, if your dataset is balanced (same number from each class)? Are these datasets balanced?

**6.2G) (Optional)** Feel free to classify other images from each other. Which combinations perform the best, and which perform the worst? Do these make sense? Other than row and column average, are there any other features you could think of that would preserve some spatial information?

This homework does not provide Python code. Instead, we encourage you to write your own code to help you answer some of these problems, and/or to test and debug the code components we do ask for. Some of the problems below are simple enough that hand calculation should be possible; your hand solutions can serve as test cases for your code. You may also find that including utilities written in previous labs (like an `sd` or signed distance function) will be helpful, as you build up additional functions and utilities for calculation of margins, different loss functions, gradients, and other functions needed for margin maximization and gradient descent here.

For your convenience, we have copied the hands-on section into a colab notebook, [which may be found here](#).

## 1) Margin

When we train a classifier, it is desirable for the classifier to have a large margin with regard to the points in our data set, in the hope that this will make the classifier more robust to any new points we might see.

We have previously defined the margin of a single example (a single data point) with respect to a separator, but that does not directly indicate whether a separator will perform well on a large data set. Thus, we would like to find a score function  $S$  for a separator  $(\theta, \theta_0)$ , such that maximizing  $S$  leads to a better separator.

Marge Inovera suggests that because big margins are good, we should maximize the sum of the margins. So, she defines:

$$S_{sum}(\theta, \theta_0) = \sum_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

Minnie Malle suggests that it would be better to just worry about the points closest to the margin, and defines:

$$S_{min}(\theta, \theta_0) = \min_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

Maxim Argent suggests:

$$S_{max}(\theta, \theta_0) = \max_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

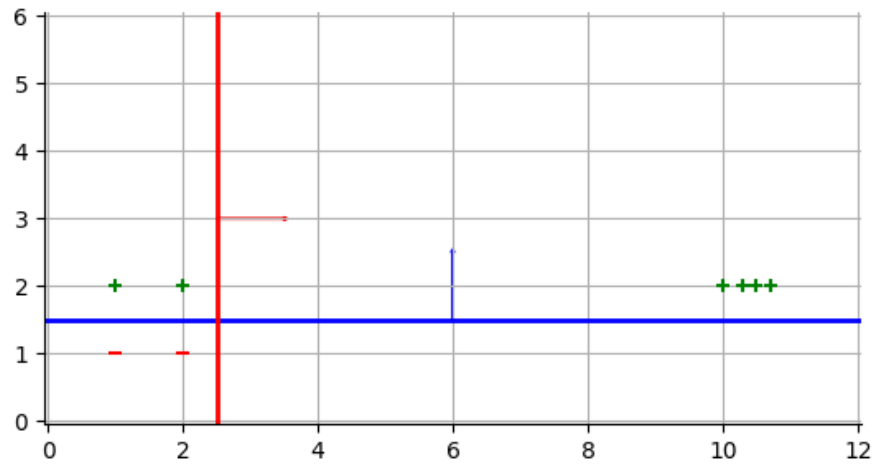
Recall that the margin of a given point is defined as

$$\gamma(x, y, \theta, \theta_0) = \frac{y(\theta \cdot x + \theta_0)}{\|\theta\|}.$$

Consider the following data, and two potential separators (red and blue).

```
data = np.array([[1, 2, 1, 2, 10, 10.3, 10.5, 10.7],
                 [1, 1, 2, 2, 2, 2, 2, 2]])
labels = np.array([[-1, -1, 1, 1, 1, 1, 1, 1]])
blue_th = np.array([[0, 1]]).T
blue_th0 = -1.5
red_th = np.array([[1, 0]]).T
red_th0 = -2.5
```

The situation is illustrated in the figure below.



**1A)** What are the values of each score ( $S_{sum}$ ,  $S_{min}$ ,  $S_{max}$ ) on the red separator?

Enter a Python list of three numbers.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1B)** What are the values of each score ( $S_{sum}$ ,  $S_{min}$ ,  $S_{max}$ ) on the blue separator?

Enter a Python list of three numbers.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1C)** Which of these separators maximizes  $S_{sum}$ ?

Choose one:  ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1D)** Which separator maximizes  $S_{min}$ ?

Choose one:  ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1E)** Which separator maximizes  $S_{max}$ ?

Choose one: red ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1F)** Which score function should we prefer if our goal is to find a separator that generalizes better to new data?

Choose one: S\_min ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 2) What a loss

Based on the previous part, we've decided to try to find a linear separator  $\theta, \theta_0$  that **maximizes** the **minimum margin** (the distance between the separator and the points that come closest to it.) We define the margin of a data set  $(X, Y)$ , with respect to a separator as

$$\gamma(X, Y, \theta, \theta_0) = \min_i \gamma(x^{(i)}, y^{(i)}, \theta, \theta_0).$$

As discussed in the [notes](#), an approach to this problem is to specify a value  $\gamma_{ref}$  for the margin of the data set, and then seek to find a linear separator that maximizes  $\gamma_{ref}$ .

**2A)** We can think about a (not necessarily maximal) margin  $\gamma_{ref}$  for the data set as a value such that:

- ☐ for at least one point  $x^{(i)}, y^{(i)}$ , we have  $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \geq \gamma_{ref}$
- ☒ for every point  $x^{(i)}, y^{(i)}$ , we have  $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \geq \gamma_{ref}$
- ☐ for at least one point  $x^{(i)}, y^{(i)}$ , we have  $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \leq \gamma_{ref}$
- ☐ for every point  $x^{(i)}, y^{(i)}$ , we have  $\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0) \leq \gamma_{ref}$

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2B)** Suppose for our data set we find that the maximum  $\gamma_{ref}$  across all linear separators is 0.

Is our data linearly separable? No ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2C)** For this subproblem, assume that  $\gamma_{ref} > 0$  (i.e., the data is linearly separable). Note that in this case, the Perceptron algorithm is guaranteed to find a separator that correctly classifies all of the data points.

What is the **largest minimum margin** guaranteed by running the Perceptron algorithm on a data set that has a maximum margin equal to  $\gamma_{ref} > 0$  ?

- ☐ 0  
☐  $\infty$   
☐  $\gamma_{ref}$   
☒ some  $\epsilon$  where  $\epsilon > 0$

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Now we want to improve on the (infinitesimally small) guaranteed margin of the Perceptron algorithm. We saw in the lecture that a powerful way of designing learning algorithms is to **describe them as optimization problems**, then use relatively general-purpose optimization strategies to solve them.

A typical form of the optimization problem is to minimize an objective that has the form

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0)$$

where  $L$  is a per-point *loss function* that characterizes how much error was made by the hypothesis  $(\theta, \theta_0)$  on the point, and  $R$  is a *regularizer* that describes some prior knowledge or general preference over hypotheses.

We first consider the objective of finding a maximum-margin separator using the format above, using the so-called "zero-infinity" loss,  $L_{0,\infty}$ :

$$L_{0,\infty}(\gamma(x, y, \theta, \theta_0), \gamma_{ref}) = \begin{cases} \infty & \text{if } \gamma(x, y, \theta, \theta_0) < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

and

$$J_{0,\infty}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n L_{0,\infty}(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0).$$

**2D)** For a linearly separable data set, positive  $\lambda$ , and positive  $R$  given nonzero  $\theta$ , what is true about the **minimal** value of  $J_{0,\infty}$  ?

Which of the following is true: It is always finite and positive ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2E)** For a **non** linearly separable data set, and positive  $\lambda$  and  $\gamma_{ref}$ , what is true about the **minimal** value of  $J_{0,\infty}$ :

Which of the following is true:



**100.00%**

*You have infinitely many submissions remaining.*

### 3) Simply inseparable

We would prefer a loss function that helps steer optimization toward a solution, in the case when the data is linearly separable. Furthermore, in real data sets it is relatively rare that the data is linearly separable, so our algorithm should be able to handle this case also and still work toward an optimal, though imperfect, linear separator. Instead of using  $(0, \infty)$  loss, we should design a loss function that will let us "relax" the constraint that all of the points have margin bigger than  $\gamma_{ref}$ , while still encouraging large margins.

The [hinge loss](#) is one such more relaxed loss function; we will define it in a way that makes a connection to the problem we are facing:

$$L_h \left( \frac{\gamma(x, y, \theta, \theta_0)}{\gamma_{ref}} \right) = \begin{cases} 1 - \frac{\gamma(x, y, \theta, \theta_0)}{\gamma_{ref}} & \text{if } \gamma(x, y, \theta, \theta_0) < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

When the margin of the point is greater than or equal to  $\gamma_{ref}$ , we are happy and the loss is 0; while when the margin is less than  $\gamma_{ref}$ , we have a positive loss that increases the further away the margin is from  $\gamma_{ref}$ .

**3A)** Given this definition, if  $\gamma_{ref}$  is positive what can we say about  $L_h(\gamma(x, y, \theta, \theta_0)/\gamma_{ref})$ , no matter what finite values  $\theta$  and  $\theta_0$  take on?

Which of the following is true about  $L_h$ :

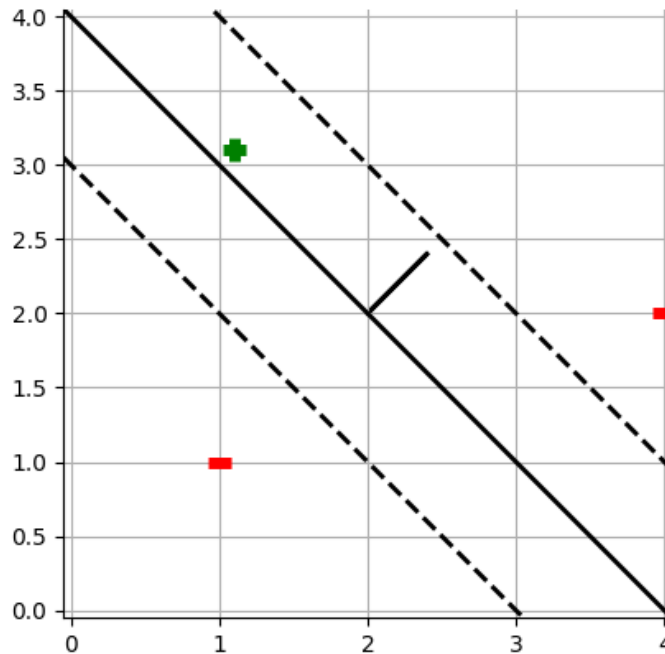


**100.00%**

*You have infinitely many submissions remaining.*

Here is a separator and three points. The dotted lines represent the margins determined by  $\gamma_{ref}$ .

```
data = np.array([[1.1, 1, 4],[3.1, 1, 2]])
labels = np.array([[1, -1, -1]])
th = np.array([[1, 1]]).T
th0 = -4
```



**3B)** What is  $L_h(\gamma(x, y, \theta, \theta_0)/\gamma_{ref})$  for each point, where  $\gamma_{ref} = \sqrt{2}/2$ ? Enter the values in the same order as the respective points are listed in `data`. You can do this computationally or by hand.

Enter the three hinge loss values in order as a Python list of three numbers:





100.00%

You have infinitely many submissions remaining.

## 4) It hinges on the loss

Putting hinge loss and regularization together, we can look at regularized average hinge loss:

$$\frac{1}{n} \sum_{i=1}^n L_h \left( \frac{\gamma(x^{(i)}, y^{(i)}, \theta, \theta_0)}{\gamma_{ref}} \right) + \lambda \frac{1}{\gamma_{ref}^2} .$$

We only need to minimize this over two parameters  $\theta, \theta_0$ , since the third parameter  $\gamma_{ref}$  can be expressed as  $\frac{1}{\|\theta\|}$ , as they both represent the distance from the decision boundary to the margin boundary. Plugging in  $\gamma_{ref} = \frac{1}{\|\theta\|}$  and also expanding  $\gamma$ , we arrive at the [SVM \(support vector machine\) objective](#):

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n L_h(y^{(i)}(\theta^T x^{(i)} + \theta_0)) + \lambda \|\theta\|^2 .$$

**4A)** If the data is linearly separable and we use the SVM objective, if we now let  $\lambda = 0$  and find the minimizing values of  $\theta, \theta_0$ , what will happen?

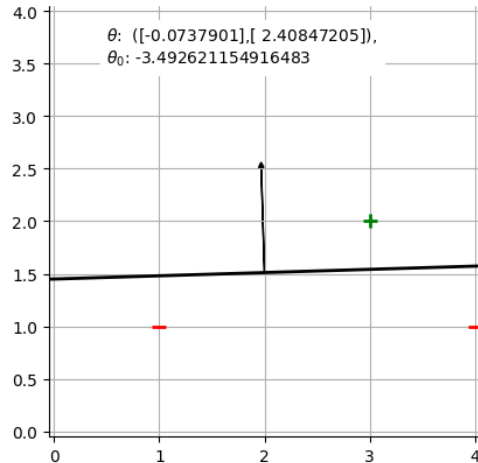


Which of the following is true: The minimal objective value will be 0 ▼

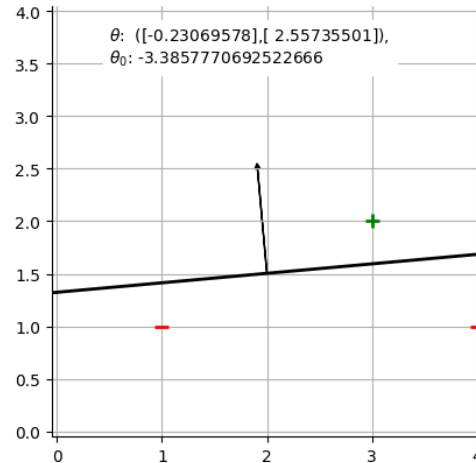
Submit
View Answer
Ask for Help
100.00%

*You have infinitely many submissions remaining.*

**4B)** Consider the following plots of separators. They are for  $\lambda$  values of 0 and 0.001. Match each  $\lambda$  to a plot.



A



B

Enter a Python list with the values of  $\lambda$  for the two graphs above (i.e.,  $[\lambda_{\text{A}}, \lambda_{\text{B}}]$ ).

[0.001, 0]
Ask for Help
100.00%

*You have infinitely many submissions remaining.*

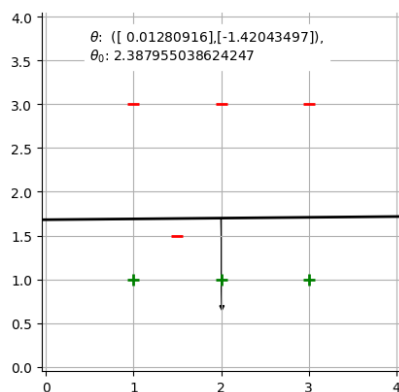
Solution:  $[0.001, 0.0]$

### Explanation:

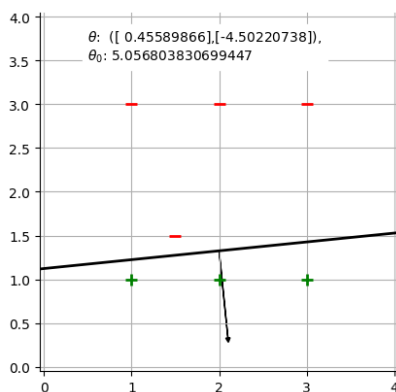
The separators in both plot A and plot B achieve zero hinge loss (for some  $\theta$ ). (Note: this is not immediately obvious; one needs to plug in the numbers to confirm at least for non-zero  $\lambda$ ). Remember that even if a separator has no errors, there may still be non-zero hinge loss if some of the correctly classified points are "too close" to the separator (within the margin).

With zero hinge loss in these plots, the operative term in the cost function becomes  $\lambda \|\theta\|^2$ . Considering the norm of the  $\theta$  vector in the two cases, for plot A the norm of  $\theta$  is 2.4096, while for plot B the norm of  $\theta$  is 2.5677. So plot A has non-zero  $\lambda$  operating to shrink  $\theta$ , while plot B has zero  $\lambda$  allowing the norm of  $\theta$  in plot B to be bigger.

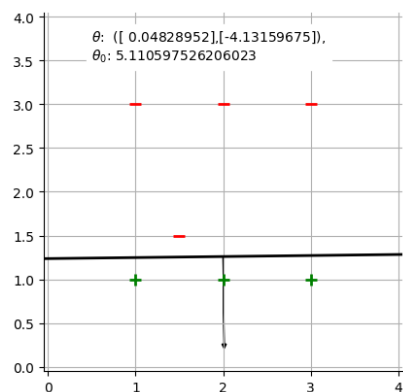
**4C)** Consider the following three plots of separators. They are for  $\lambda$  values of 0, 0.001, and 0.03. Match to the plot.



A



B



C

Enter a Python list with the values of  $\lambda$  for the three graphs above (i.e., `[lambdaA, lambdaB, lambdaC]`).

[Ask for Help](#)
**100.00%**

You have infinitely many submissions remaining.

Solution: `[0.03, 0.0, 0.001]`

### Explanation:

When we increase  $\lambda$ , we penalize larger values of  $\theta$ . In some cases, this may mean we incur non-zero or larger hinge loss (where points are closer to the separator, or even where some points are misclassified).

Plot A has minimized  $\theta$  to the degree that a point is misclassified; this corresponds to the largest value of  $\lambda$ , 0.3. (Note that for Plot A,  $\|\theta\| = 1.4205$ , average hinge loss is 0.1861, and the margin is negative, -2.0735).

Plot B, in contrast, has the largest  $\|\theta\|$ , corresponding to the smallest  $\lambda$ , or  $\lambda = 0$ . (Note that for Plot B,  $\|\theta\| = 4.525$ , average hinge loss is 0, and the margin is positive, 0.2233.)

Finally, for Plot C we see that again zero hinge loss can be achieved, but here a non-zero  $\lambda = 0.001$  has acted to reduce  $\|\theta\|$  or equivalent increase the margin. (Note that for Plot C,  $\|\theta\| = 4.1319$ , average hinge loss is 0, and the margin is positive, 0.2455, slightly larger than in Plot B.)

Another observation we may make is that when  $\lambda = 0$ , only the hinge loss term remains, so we only care to linearly separate the data. Since the data are separable, the optimization process will find a separator that perfectly separates the data. This gives us either B or C could correspond to  $\lambda = 0$  from visual inspection. To choose between them for which corresponds to  $\lambda = 0$  or  $\lambda$  very small, we have to look more closely at the corresponding margin or norm of  $\theta$ .

## 5) Linear Support Vector Machines

The training objective for the Support Vector Machine (with slack) can be seen as optimizing a balance between the average hinge loss over the examples and a regularization term that tries to keep  $\theta$  small (or equivalently, increase the margin). This balance is set by the regularization parameter  $\lambda$ . Here we only consider the case without the offset parameter  $\theta_0$  (setting it to zero) and rewrite the training objective as an average so that it is given by

$$\left[ \frac{1}{n} \sum_{i=1}^n L_h(y^{(i)} \theta \cdot x^{(i)}) \right] + \frac{\lambda}{2} \|\theta\|^2 = \frac{1}{n} \sum_{i=1}^n \left[ L_h(y^{(i)} \theta \cdot x^{(i)}) + \frac{\lambda}{2} \|\theta\|^2 \right]$$

where  $L_h(y(\theta \cdot x)) = \max\{0, 1 - y(\theta \cdot x)\}$  is the hinge loss. (Note that we will also sometimes write the hinge loss as  $L_h(v) = \max(0, 1 - v)$ .) Now we can minimize the above overall objective function with the Pegasos algorithm that iteratively selects a training point at random and applies a gradient descent update rule based on the corresponding term inside the brackets on the right hand side.

In this problem we will optimize the training objective using a single training example, so that we can gain a better understanding of how the regularization parameter,  $\lambda$ , affects the result. To this end, we refer to the single training example as the feature vector and label pair,  $(x, y)$ . We will then try to find a  $\theta$  that minimizes

$$J_{\lambda}^1(\theta) \equiv L_h(y(\theta \cdot x)) + \frac{\lambda}{2} \|\theta\|^2.$$

In the next subparts, we will try to show that the  $\theta$  minimizing  $J_{\lambda}^1$ , denoted  $\hat{\theta}$ , is necessarily of the form

$$\hat{\theta} = \eta y x$$

for some real  $\eta > 0$ .

In the expressions below, you can use `lambda` to stand for  $\lambda$ , `x` to stand for  $x$ , `transpose(x)` for transpose of an array, `norm(x)` for the length (norm) of a vector, `x@y` to indicate a matrix product of two arrays, and `x*y` for elementwise (or scalar) multiply.

**5A)** Consider first the case where the loss is positive:  $L_h(y(\theta \cdot x)) > 0$ . We can minimize  $J_{\lambda}^1$  with respect to  $\theta$  by computing a formula for its gradient with respect to  $\theta$ , and then solving for the  $\theta$  for which the gradient is equal to 0. Let us denote that value as  $\hat{\theta}$ .

Enter an expression for  $\hat{\theta}$ :

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**5B)** Now find the smallest (in the norm sense)  $\hat{\theta}$  for which  $L_h(y(\theta \cdot x)) = 0$ .

Note: Be careful -- you cannot simply divide by a vector!

Enter your answer as a Python expression:  $\hat{\theta} =$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Note that if the hinge loss is zero, the point is correctly classified.

**5C)** Let  $\hat{\theta} = \hat{\theta}(\lambda)$  be the minimizer of  $J_{\lambda}^1(\theta)$ . Is it possible to pick a value for  $\lambda$  so that the training example  $x, y$  will be misclassified by  $\hat{\theta}(\lambda)$ ?

To answer this question, recall that a point is misclassified when  $y(\theta \cdot x) \leq 0$ . Use your result from part 5A where you found the  $\hat{\theta}$  that minimizes  $J_{\lambda}^1(\theta)$  to write an expression for  $y(\hat{\theta} \cdot x)$  in terms of  $x, y$  and  $\lambda$ .

For  $\theta$  that minimizes  $J_{\lambda}^1(\theta)$ , enter an expression for  $y(\hat{\theta} \cdot x)$ :






100.00%

You have infinitely many submissions remaining.

**5D)**

Under what conditions is  $y(\hat{\theta} \cdot x) \leq 0$ ? Select all that are true.

☒  $x = 0$

☐  $y < 0$

☐  $y > 0$

☐  $\lambda = 0$

☒  $\lambda = \infty$




100.00%

You have infinitely many submissions remaining.

**5E)** You will notice that if  $y(\hat{\theta} \cdot x) \leq 0$ , then  $\hat{\theta}$  will misclassify  $x$ . The above result shows that our optimal classifier  $\hat{\theta}$  won't misclassify (except in edge cases); however, we might still be concerned about correctly classified points that are "too close" to the separator, and thereby increase our regularized loss function.

Suppose we have a linear classifier described by  $\theta$ . We say a correctly classified datapoint  $\hat{x}, \hat{y}$  is on the margin boundary of the classifier if

$$\hat{y}(\theta \cdot \hat{x}) = 1.$$

When a classifier is determined by minimizing a regularized loss function with a single training example, like  $J_{\lambda}^1$  above, too much regularization can result in a classifier that puts a correctly classified training point *inside* the margin, and thus incur hinge loss. That is, if we have a single training example  $(x, y)$  and regularize with a  $\lambda$  that is too large, we may discover that  $y(\hat{\theta} \cdot x) < 1$ . Fortunately, for this single training example case, we can ensure that  $\lambda$  is not too large.

Write an expression for the maximum value of  $\lambda$ , in terms of  $x$  and  $y$ , that ensures that the  $(x, y)$  example is NOT inside the margin:






100.00%

You have infinitely many submissions remaining.

So, where are we? We now have a good objective function, the SVM objective, that will strive to correctly classify data points, but also seek to maximize the margin (minimize the norm of  $\theta$ ), given a judicious choice of  $\lambda$ . This objective function can be used in either batch optimization (calculating average losses across the whole data set), or on a data point by data point basis. This gives us powerful flexibility in optimizing (minimizing) this objective function using gradient descent, which we will consider next.

## 6) Implementing gradient descent

In this section we will implement generic versions of gradient descent and apply these to the SVM objective.

**Reminder:** For your convenience, we have copied the hands-on section into a colab notebook, [which may be found here](#).

### 6.1) Gradient descent

**Note:** If you need a refresher on gradient descent, you may want to reference [this week's notes](#).

We want to find the  $x$  that minimizes the value of the *objective function*  $f(x)$ , for an arbitrary scalar function  $f$ . The function  $f$  will be implemented as a Python function of one argument, that will be a numpy column vector. For efficiency, we will work with Python functions that return not just the value of  $f$  at  $f(x)$  but also return the gradient vector at  $x$ , that is,  $\nabla_x f(x)$ .

We will now implement a generic gradient descent function, `gd`, that has the following input arguments:

- `f`: a function whose input is an  $x$ , a column vector, and returns a scalar.
- `df`: a function whose input is an  $x$ , a column vector, and returns a column vector representing the gradient of  $f$  at  $x$ .
- `x0`: an initial value of  $x$ ,  $x_0$ , which is a column vector.
- `step_size_fn`: a function that is given the iteration index (an integer) and returns a step size.
- `max_iter`: the number of iterations to perform

Our function `gd` returns a tuple:

- `x`: the value at the final step
- `fs`: the list of values of  $f$  found during all the iterations (including  $f(x_0)$ )
- `xs`: the list of values of  $x$  found during all the iterations (including  $x_0$ )

**Hint:** This is a short function!

**Hint 2:** If you do `temp_x = x` where  $x$  is a vector (numpy array), then `temp_x` is just another name for the same vector as  $x$  and changing an entry in one will change an entry in the other. You should either use `x.copy()` or remember to change entries back after modification.

Some test or example functions that you may find useful are included below. You may also find `rv` and `cv` (from previous weeks) useful, though not necessary.

```
def f1(x):
    return float((2 * x + 3)**2)
```

```
def df1(x):
    return 2 * 2 * (2 * x + 3)
```

```
def f2(v):
    x = float(v[0]); y = float(v[1])
    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y - 1)**2
```

```
def df2(v):
    x = float(v[0]); y = float(v[1])
    return cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
               2 * (-1. + x + y),
               2 * (-1. + x + y)])
```

To evaluate results, we also use a simple `package_ans` function, which checks the final `x`, as well as the first and last values in `fs`, `xs`.

```
def package_ans(gd_vals):
    x, fs, xs = gd_vals
    return [x.tolist(), [fs[0], fs[-1]], [xs[0].tolist(), xs[-1].tolist()]]
```

The test cases are provided below, but you should feel free (and are encouraged!) to write more of your own.

```
# Test case 1
ans=package_ans(gd(f1, df1, cv([0.]), lambda i: 0.1, 1000))

# Test case 2
ans=package_ans(gd(f2, df2, cv([0., 0.]), lambda i: 0.01, 1000))
```

```

1 def gd(f, df, x0, step_size_fn, max_iter):
2
3     #initialize some parameters
4     num_iter = 0
5     list_x = []
6     list_f = []
7     #copy the input so we don't modify it on each iter
8     x = x0.copy()
9     #execute updates until we reach out max num of updates
10    while num_iter < max_iter:
11        #calculate the current gradient
12        grad = df(x)
13        list_x.append(x)
14        list_f.append(f(x))
15        #next selection is the current - the gradient times a step size

```

[Ask for Help](#)

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```

def gd(f, df, x0, step_size_fn, max_iter):
    prev_x = x0
    fs = []; xs = []
    for i in range(max_iter):
        prev_f, prev_grad = f(prev_x), df(prev_x)
        fs.append(prev_f); xs.append(prev_x)
        if i == max_iter-1:
            return prev_x, fs, xs
        step = step_size_fn(i)
        prev_x = prev_x - step * prev_grad

```

## 6.2) Numerical Gradient

Getting the analytic gradient correct for complicated functions is tricky. A very handy method of verifying the analytic gradient or even substituting for it is to estimate the gradient at a point by means of *finite differences*.

Assume that we are given a function  $f(x)$  that takes a column vector as its argument and returns a scalar value. In gradient descent, we will want to estimate the gradient of  $f$  at a particular  $x_0$ .

The  $i^{th}$  component of  $\nabla_x f(x_0)$  can be estimated as

$$\frac{f(x_0 + \delta^i) - f(x_0 - \delta^i)}{2\delta}$$

where  $\delta^i$  is a column vector whose  $i^{th}$  coordinate is  $\delta$ , a small constant such as 0.001, and whose other components are zero. Note that adding or subtracting  $\delta^i$  is the same as incrementing or decrementing the  $i^{th}$  component of  $x_0$  by  $\delta$ , leaving the other components of  $x_0$  unchanged. Using these results, we can estimate the  $i^{th}$  component of the gradient.

For example, if  $x_0 = (1, 1, \dots, 1)^T$  and  $\delta = 0.01$ , we may approximate the first component of  $\nabla_x f(x_0)$  as

$$\frac{f((1, 1, 1, \dots)^T + (0.01, 0, 0, \dots)^T) - f((1, 1, 1, \dots)^T - (0.01, 0, 0, \dots)^T)}{2 \cdot 0.01}.$$

(We add the transpose so that these are column vectors.) **This process should be done for each dimension independently, and together the results of each computation are compiled to give the estimated gradient, which is  $d$  dimensional.**

Implement this as a function `num_grad` that takes as arguments the objective function `f` and a value of `delta`, and returns a new **function** that takes an `x` (a column vector of parameters) and returns a gradient column vector.

**Note:** As in the previous part, make sure you do not modify your input vector.

The test cases are shown below; these use the functions defined in the previous exercise.

```
x = cv([0.])
ans=(num_grad(f1)(x).tolist(), x.tolist())
```

```
x = cv([0.1])
ans=(num_grad(f1)(x).tolist(), x.tolist())
```

```
x = cv([0., 0.])
ans=(num_grad(f2)(x).tolist(), x.tolist())
```

```
x = cv([0.1, -0.1])
ans=(num_grad(f2)(x).tolist(), x.tolist())
```

```
1 def num_grad(f, delta=0.001):
2     def df(x):
3         x_temp = x.copy()
4         d, n = x.shape
5         grad = np.zeros((d, 1))
6
7         for row in range(d):
8             increment = np.zeros((d,1))
9             increment[row, 0] = delta
10            grad_row = (f(x + increment) - f(x - increment))/(2*delta)
11            grad[row, 0] = grad_row
12        return grad
13    return df
14 |
```

[Run Code](#)
[Submit](#)
[View Answer](#)
[Ask for Help](#)
**100.00%**

You have infinitely many submissions remaining.

A faster (one function evaluation per entry), though sometimes less accurate, estimate is to use:

$$\frac{f(x_0 + \delta^i) - f(x_0)}{\delta}$$

for the  $i^{th}$  component of  $\nabla_x f(x_0)$ .

## 6.3) Using the Numerical Gradient



Recall that our generic gradient descent function takes both a function  $f$  that returns the value of our function at a given point, and  $df$ , a function that returns a gradient at a given point. Write a function `minimize` that takes only a function  $f$  and uses this function and numerical gradient descent to return the local minimum. We have provided you with our implementations of `num_grad` and `gd`, so you should not redefine them in the code box below. You may use the default of `delta=0.001` for `num_grad`.

**Hint:** Your definition of `minimize` should call `num_grad` exactly once, to return a function that is called many times. You should return the same outputs as `gd`.

The test cases are:

```
ans = package_ans(minimize(f1, cv([0.]), lambda i: 0.1, 1000))
```

```
ans = package_ans(minimize(f2, cv([0., 0.]), lambda i: 0.01, 1000))
```

```
1 def minimize(f, x0, step_size_fn, max_iter):
2     #initialize some parameters
3     num_iter = 0
4     list_x = []
5     list_f = []
6     #copy the input so we dont modify it on each iter
7     x = x0.copy()
8     num_grad_func = num_grad(f)
9     #execute updates until we reach out max num of updates
10    while num_iter < max_iter:
11        #calculate the current gradient
12        grad = num_grad_func(x)
13        list_x.append(x)
14        list_f.append(f(x))
15        #next selection is the current - the gradient times a step size
```

[Run Code](#)
[Submit](#)
[View Answer](#)
[Ask for Help](#)
**100.00%**

You have infinitely many submissions remaining.

## 7) Applying gradient descent to SVM objective

Now that we've implemented gradient descent in the general case, let's go back to hinge loss and the SVM objective. Our goal in this section will be to derive and implement appropriate gradient calculations that we can use with `gd` for optimization of the SVM objective. In the derivations below, we'll consider linear classifiers *with* offset, i.e.,  $\theta, \theta_0$ .

Recall that hinge loss is defined as:

$$L_h(v) = \begin{cases} 1 - v & \text{if } v < 1 \\ 0 & \text{otherwise} \end{cases}.$$

This is usually implemented as:

$$\text{hinge}(v) = \max(0, 1 - v).$$

The hinge loss function, in the context of our problem, takes in the distance from a point to the separator as  $x$ . This loss function helps us penalize a model for leaving points within a distance to our separator:

$$L_h(y(\theta \cdot x + \theta_0)) = \begin{cases} 1 - y(\theta \cdot x + \theta_0) & \text{if } y(\theta \cdot x + \theta_0) < 1 \\ 0 & \text{otherwise} \end{cases}$$

The SVM objective function incorporates the mean of the hinge loss over all points and introduces a regularization term to this equation to make sure that the magnitude of  $\theta$  stays small (and keeps the margin large). Note that we have used  $\lambda$  instead of  $\frac{\lambda}{2}$  for simplicity (and without loss of generality).

$$J(\theta, \theta_0) = \left[ \frac{1}{n} \sum_{i=1}^n L_h(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)) \right] + \lambda \|\theta\|^2$$

We're interested in applying our gradient descent procedure to this function in order to find the 'best' separator for our data, where 'best' is measured by the lowest possible SVM objective.

## 7.1) Calculating the SVM objective

Implement the single-argument function `hinge`, which computes  $L_h$ , and use that to implement hinge loss for a data point and separator. Using the latter function, implement the SVM objective. Note that these functions should work for matrix/vector arguments, so that we can compute the objective for a whole dataset with one call.

Note that  $x$  is  $d \times n$ ,  $y$  is  $1 \times n$ ,  $th$  is  $d \times 1$ ,  $th_0$  is  $1 \times 1$ ,  $\lambda$  is a scalar.

Hint: Look at `np.where` for implementing `hinge`.

In the test cases for this problem, we'll use the following `super_simple_separable` test dataset and test separator for some of the tests. A couple of the test cases are also shown below.

```
def super_simple_separable():
    X = np.array([[2, 3, 9, 12],
                  [5, 2, 6, 5]])
    y = np.array([[1, -1, 1, -1]])
    return X, y

sep_e_separator = np.array([[-0.40338351], [1.1849563]]), np.array([[-2.26910091]])

# Test case 1
x_1, y_1 = super_simple_separable()
th1, th1_0 = sep_e_separator
ans = svm_obj(x_1, y_1, th1, th1_0, .1)

# Test case 2
ans = svm_obj(x_1, y_1, th1, th1_0, 0.0)
```

**Note:** In this section, you will code many individual functions, each of which depends on previous ones. We **strongly recommend** that you test each of the components on your own to debug.

```

1 def hinge(v):
2     #hinge loss calculate for each data point so v is 1xn
3     return np.where(v>0, v, 0)
4
5 # x is dxn, y is 1xn, th is dx1, th0 is 1x1
6 def hinge_loss(x, y, th, th0):
7     return 1 - y * (np.dot(th.T, x) + th0)
8
9 # x is dxn, y is 1xn, th is dx1, th0 is 1x1, lam is a scalar
10 def svm_obj(x, y, th, th0, lam):
11     d, n = x.shape
12     loss_list = np.zeros((1, n))
13     for col in range(n):
14         h_loss = hinge_loss(x[:, col:col+1], y[0, col], th, th0)
15         loss_list[0, col] = h_loss
16
17     loss_list = hinge(loss_list)
18     reg_loss = lam * (np.linalg.norm(th)**2)
19
20     return np.sum(loss_list)/n + reg_loss
21

```

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```

def hinge(v):
    return np.where(v >= 1, 0, 1 - v)

def hinge_loss(x, y, th, th0):
    return hinge(y * (np.dot(th.T, x) + th0))

def svm_obj(X, y, th, th0, lam):
    return np.mean(hinge_loss(X, y, th, th0)) + lam * np.linalg.norm(th) ** 2

```

## 7.2) Calculating the SVM gradient

Define a function `svm_obj_grad` that returns the gradient of the SVM objective function with respect to  $\theta$  and  $\theta_0$  in a single column vector. The last component of the gradient vector should be the partial derivative with respect to  $\theta_0$ . Look at `np.vstack` as a simple way of stacking two matrices/vectors vertically. We have broken it down into pieces that mimic steps in the chain rule; this leads to code that is a bit inefficient but easier to write and debug. We can worry about efficiency later.

Some test cases that may be of use are shown below:

```

X1 = np.array([[1, 2, 3, 9, 10]])
y1 = np.array([[1, 1, 1, -1, -1]])
th1, th10 = np.array([[ -0.31202807]]), np.array([[1.834    ]])
X2 = np.array([[2, 3, 9, 12],
               [5, 2, 6, 5]])
y2 = np.array([[1, -1, 1, -1]])
th2, th20 = np.array([[ -3., 15.]]) .T, np.array([[ 2.]])

```

```

d_hinge(np.array([[ 71.]])).tolist()
d_hinge(np.array([[ -23.]])).tolist()
d_hinge(np.array([[ 71, -23.]])).tolist()

d_hinge_loss_th(X2[:,0:1], y2[:,0:1], th2, th20).tolist()
d_hinge_loss_th(X2, y2, th2, th20).tolist()
d_hinge_loss_th0(X2[:,0:1], y2[:,0:1], th2, th20).tolist()
d_hinge_loss_th0(X2, y2, th2, th20).tolist()

d_svm_obj_th(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
d_svm_obj_th(X2, y2, th2, th20, 0.01).tolist()
d_svm_obj_th0(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
d_svm_obj_th0(X2, y2, th2, th20, 0.01).tolist()

svm_obj_grad(X2, y2, th2, th20, 0.01).tolist()
svm_obj_grad(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()

```

**Note:** In this section, you will code many individual functions, each of which depends on previous ones. We **strongly recommend** that you test each of the components on your own to debug.

```

1 # Returns the gradient of hinge(v) with respect to v.
2 def d_hinge(v):
3     return np.where(v<1, -1, 0)
4
5 # Returns the gradient of hinge_loss(x, y, th, th0) with respect to th
6 def d_hinge_loss_th(x, y, th, th0):
7     return y * x * d_hinge(y * (np.dot(th.T, x) + th0))
8
9 # Returns the gradient of hinge_loss(x, y, th, th0) with respect to th0
10 def d_hinge_loss_th0(x, y, th, th0):
11     return y * d_hinge(y * (np.dot(th.T, x) + th0))
12
13 # Returns the gradient of svm_obj(x, y, th, th0) with respect to th
14 def d_svm_obj_th(x, y, th, th0, lam):
15     return np.mean(d_hinge_loss_th(x, y, th, th0), axis = 1, keepdims = True) + 2*lam*
16 # Returns the gradient of svm_obj(x, y, th, th0) with respect to th0
17 def d_svm_obj_th0(x, y, th, th0, lam):
18     return np.mean(d_hinge_loss_th0(x, y, th, th0), axis = 1, keepdims = True)
19
20 # Returns the full gradient as a single vector (which includes both th, th0)
21 def svm_obj_grad(x, y, th, th0, lam):
22     return np.vstack((d_svm_obj_th(x, y, th, th0, lam), d_svm_obj_th0(x, y, th, th0, 1
23 |

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 7.3) Batch SVM minimize

Putting it all together, use the functions you built earlier to write a gradient descent minimizer for the SVM objective. You do not need to paste in your previous definitions; you can just call the ones defined by the staff. You will need to call `gd`, which is already defined for you as well; your function `batch_svm_min` should return the values that `gd` does.

- Initialize all the separator parameters to zero,
- use the step size function provided below, and
- specify 10 iterations.

Test cases are shown below, where an additional separable test data set has been specified.

```
def separable_medium():
    X = np.array([[2, -1, 1, 1],
                  [-2, 2, 2, -1]])
    y = np.array([[1, -1, 1, -1]])
    return X, y
sep_m_separator = np.array([[ 2.69231855], [ 0.67624906]]), np.array([[ -3.02402521]])

x_1, y_1 = super_simple_separable()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))

x_1, y_1 = separable_medium()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))
```

```
1 def batch_svm_min(data, labels, lam):
2     def svm_min_step_size_fn(i):
3         return 2/(i+1)**0.5
4     d, n = data.shape
5     th = np.zeros((d+1, 1))
6
7     def f(th):
8         return svm_obj(data, labels, th[:-1, :], th[-1:, :], lam)
9
10    def df(th):
11        return svm_obj_grad(data, labels, th[:-1, :], th[-1:, :], lam)
12
13
14    return gd(f, df, th, svm_min_step_size_fn, 10)
```

[Run Code](#)
[Submit](#)
[View Answer](#)
[Ask for Help](#)
**100.00%**

You have infinitely many submissions remaining.

## 7.4) Numerical SVM objective (Optional)

Recall from the previous question that we were able to closely approximate gradients with numerical estimates. We may apply the same technique to optimize the SVM objective.

Using your definition of `minimize` and `num_grad` from the previous problem, implement a function that optimizes the SVM objective through numeric approximations.

How well does this function perform, compared to the analytical result? Consider both accuracy and runtime.

You may find the lecture notes on [regression](#) helpful as you do this homework.

## 1) Intro to linear regression

So far, we have been looking at classification, where predictors are of the form

$$\hat{y}^{(i)} = \text{sign}(\theta^T x^{(i)} + \theta_0)$$

where  $\hat{y}^{(i)}$  is our prediction of the corresponding label  $y^{(i)}$  making a binary classification as to whether example  $x^{(i)}$  belongs to the positive or negative class of examples.

In many problems, we want to predict a real value, such as the actual gas mileage of a car, or the concentration of some chemical. Luckily, we can use much of the mechanism we have already developed, and make predictors of the form:

$$\hat{y}^{(i)} = \theta^T x^{(i)} + \theta_0 .$$

This is called a *linear regression* model.

We would like to learn a linear regression model from examples. Assume  $X$  is a  $d$  by  $n$  array (as before) but that  $Y$  is a  $1$  by  $n$  array of floating-point numbers (rather than  $+1$  or  $-1$ ). Given data  $(X, Y)$  we need to find  $\theta, \theta_0$  that does a good job of making predictions on new data drawn from the same source.

We will approach this problem by formulating an objective function. There are many possible reasonable objective functions that implicitly make slightly different assumptions about the data, but they all typically have the form:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n L(x^{(i)}, y^{(i)}, \theta, \theta_0) + \lambda R(\theta, \theta_0)$$

For regression, we most frequently use *squared loss*, in which

$$L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) = (\hat{y}^{(i)} - y^{(i)})^2 = (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2 .$$

We might start by simply trying to minimize the average squared loss on the training data; this is called the *empirical risk* or *mean square error*:

$$J_{\text{emp}}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) .$$

Later, we will add in a regularization term.

We will see later in this assignment that we can find a closed form matrix formula (requiring a matrix inverse) for the optimal  $\theta$  in a linear regression formula. Being able to solve a machine-learning problem in closed form is very awesome! But inverting a matrix is computationally expensive (a bit less than  $O(m^3)$  where  $m$  is the dimension of our matrix), and so, as our data sets get larger, we will need to find some more efficient or approximate ways to approach the problem.

For your convenience, we have copied the hands-on section into a colab notebook, [which may be found here](#). You can alternatively fill in these functions in `code_for_hw5.py`, which is part of [this set of files](#) (the other files will be useful for the last part of the homework).

Let's start by thinking about [gradient descent](#) to attack this problem:

**1A)** What is the gradient of the empirical risk with respect to  $\theta$ ? We can see that it is of the form:

$$\nabla_{\theta} J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n g(x^{(i)}, y^{(i)})$$

where  $x^{(i)}, y^{(i)}$  are the  $i^{th}$  data point and its label.

Write an expression for  $g(x^{(i)}, y^{(i)})$  using the symbols:  $x\_i$ ,  $y\_i$ ,  $\theta$  and  $\theta\_0$ , where  $g(x^{(i)}, y^{(i)})$  is the derivative of the  $L_s$  function described above with respect to  $\theta$ . Remember that you can use `@` for matrix product, and you can use `transpose(v)` to transpose a vector. Note that this  $g(\cdot)$  function is *just* the derivative with respect to a single data point  $x^{(i)}, y^{(i)}$ . We'll build up to the gradient of  $J_{emp}$  in a moment.

$g(x^{(i)}, y^{(i)}) =$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1B)** What is the gradient of the empirical risk **now with respect to  $\theta_0$** ? We can see that it is of a similar form:

$$\nabla_{\theta_0} J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n g_0(x^{(i)}, y^{(i)}) .$$

Write an expression for  $g_0(x^{(i)}, y^{(i)})$  using the symbols:  $x\_i$ ,  $y\_i$ ,  $\theta$  and  $\theta\_0$ .

$g_0(x^{(i)}, y^{(i)}) =$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1C)** Next we're interested in the gradient of the empirical loss with respect to  $\theta$ ,  $\nabla_{\theta} J_{emp}$ , but now for a whole data set  $X$  (of dimensions  $d$  by  $n$ ).

Write an expression for  $\nabla_{\theta} J_{emp}$  using the symbols:  $x$  and  $y$  for the full set of data,  $\theta$ ,  $\theta_0$ , and  $n$ . Here  $x$  has dimensions  $d$  by  $n$ , and  $y$  has dimensions 1 by  $n$ . Remember that you can use `@` for matrix product, and you can use `transpose(a)` to transpose a vector or array.

$\nabla_{\theta} J_{emp} =$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 2) Sources of Error



Recall that *structural* error arises when the hypothesis class cannot represent a hypothesis that performs well on the test data and *estimation* error arises when the parameters of a hypothesis cannot be estimated well based on the training data. (You can also refer to [here](#) in the notes.)

Following is a collection of potential cures for a situation in which your learning algorithm generates a hypothesis with a high test error.

For each one, indicate whether it can **can reduce** structural error, estimation error, both, or neither.

**2A)** Penalize  $\|\theta\|^2$  during training.

Can reduce:

- ☐ structural error
- ☒ estimation error
- ☐ both
- ☐ neither

Submit

View Answer

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

**2B)** Penalize  $\|\theta\|^2$  during testing.

Can reduce:

- ☐ structural error
- ☐ estimation error
- ☐ both
- ☒ neither

Submit

View Answer

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

**2C)** Increase the amount of training data.

Can reduce:

- ☐ structural error
- ☒ estimation error
- ☐ both
- ☐ neither

Submit

View Answer

Ask for Help

100.00%

*You have infinitely many submissions remaining.*

**2D)** Increase the order of a fixed polynomial basis.

Can reduce:

- ☒ structural error
- ☐ estimation error
- ☐ both
- ☐ neither

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2E)** Decrease the order of a fixed polynomial basis.

Can reduce:

- ☐ structural error
- ☒ estimation error
- ☐ both
- ☐ neither

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

### 3) Minimizing empirical risk

We can also solve regression problems analytically.

Remember the definition of *squared loss*,

$$L_s(x^{(i)}, y^{(i)}, \theta, \theta_0) = (\theta^T x^{(i)} + \theta_0 - y^{(i)})^2$$

and *empirical risk*:

$$J_{emp}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n L_s(x^{(i)}, y^{(i)}, \theta, \theta_0)$$

Later, we will add in a regularization term.

For simplicity in this section, assume that we are handling the constant term,  $\theta_0$ , by adding a dimension to the input feature vector that always has the value 1. To review why this works, take a look at the introduction to problem 1 in HW2.

Let data matrix  $Z = X^T$  be  $n$  by  $d$ , let target output vector  $T = Y^T$  be  $n$  by 1, and recall that  $\theta$  is  $d$  by 1. Then we can write the whole linear regression prediction as  $Z\theta$ .

**3A)**  $T$  is the  $n$  by 1 vector of target output values. Write an equation expressing the mean squared loss of  $\theta$  in terms of  $Z$ ,  $T$ ,  $n$ , and  $\theta$ . Hint: note that this loss  $J(\theta)$  is a scalar, a sum of squared terms divided by  $n$ ; we can write it as  $(W^T W)/n$  for a column vector  $W$ .

Enter your answer as a Python expression. You can use symbols  $z$ ,  $T$ ,  $n$  and  $\theta$ . Recall that our expression syntax includes `transpose(x)` for transpose of an array, `inverse(x)` for the inverse of an array, and `x@y` to indicate a matrix product of two arrays.

$$J(\theta) = \text{transpose}(Z @ \theta - T) @ (Z @ \theta - T) / n$$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Now, how can we find the minimizing  $\theta$ , given  $Z$  and  $T$ ? Take the gradient (yes, even with a matrix expression), set it to zero(s) and solve for  $\theta$ .

**3B)** What is  $\nabla_{\theta} J(\theta)$  in terms of  $Z$ ,  $T$ ,  $\theta$ , and  $n$ ? You can use matrix derivatives or, compute the answer for some individual elements and deduce the matrix form.

$$\nabla_{\theta} J(\theta) = 2/n * \text{transpose}(Z) @ (Z @ \theta - T)$$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3C)** What if you set this equation to 0 and solve for  $\theta^*$ , the optimal  $\theta$ ? Hint: It's ok to ignore the constant scaling factor.

$$\theta^* =$$

- ☐  $(Z^T T)^{-1} (Z^T Z)$   
☒  $(Z^T Z)^{-1} Z^T T$   
☐  $(Z Z^T)^{-1} Z T^T$

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution:  $(Z^T Z)^{-1} Z^T T$

#### Explanation:

We solve for  $\theta$  in  $\frac{2}{n} \cdot Z^T (Z\theta - T) = 0$ . We first expand this to obtain  $(Z^T Z) \theta - Z^T T = 0$ . This can then be rewritten  $(Z^T Z) \theta = Z^T T$ . This is then a system of linear equations that can be solved as  $\theta = (Z^T Z)^{-1} Z^T T$ , as desired.

**3D)** Just converting back to the data matrix format we have been using (not transposed), we have

$\theta^* =$ 

- ☐  $(XY^T)^{-1}(XX^T)$   
☐  $(X^T X)^{-1}X^T Y$   
☒  $(XX^T)^{-1}XY^T$

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3E)** Now implement  $\theta^*$  as found in **3D)**, using symbols  $x$  and  $y$  for the data matrix and outputs, and `np.dot`, `np.transpose`, `np.linalg.inv`.

```
1 # Enter an expression to compute and set th to the optimal theta
2 th = np.linalg.inv(X@X.T) @ X @ Y.T
```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 4) Adding regularization

Although we don't have the same notion of margin maximization as with the SVM formulation for classification, there is still a good reason to *regularize* or put pressure on the coefficient vector  $\theta$  to prevent the model from fitting the training data too closely, especially in cases where we have few data points and many features.

And, as it happens, this same regularization will help address a problem that you might have anticipated when finding the analytical solution for  $\theta$ , which is that  $XX^T$  might not be invertible (where we are using the definition of  $X$  as in problem 3, where each column of  $X$  is a  $d$ -length vector representing a  $d$ -feature sample point and there are  $n$  columns in  $X$  (or equivalently, there are  $n$  training examples)).

Consider this matrix:

```
X = np.array([[1, 2], [2, 3], [3, 5], [1, 4]])
```

Is  $XX^T$  invertible? If not, what's the problem? Mark all that are true.







- ☐ It is invertible
- ☐ It is not invertible because  $X$  is not square
- ☐ It is not invertible because two columns of  $X$  are linearly dependent
- ☒ It is not invertible because the rows of  $XX^T$  are linearly dependent
- ☒ It is not invertible because  $n$  is smaller than  $d$
- ☐ We cannot compute the transpose of  $X$

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution:

-  It is invertible
-  It is not invertible because  $X$  is not square
-  It is not invertible because two columns of  $X$  are linearly dependent
-  It is not invertible because the rows of  $XX^T$  are linearly dependent
-  It is not invertible because  $n$  is smaller than  $d$
-  We cannot compute the transpose of  $X$

**Explanation:**

We can multiply out  $XX^T$  to obtain

$$XX^T = \begin{bmatrix} 5 & 8 & 13 & 9 \\ 8 & 13 & 21 & 14 \\ 13 & 21 & 34 & 23 \\ 9 & 14 & 23 & 17 \end{bmatrix}$$

Immediately, we notice that rows 1 and 2 add to row 3, so the rows of  $X$  are not linearly independent (4 is true). Therefore, we conclude that  $XX^T$  is not full rank, and thus not invertible (1 is false).

For generic  $X$ ,  $XX^T$  may be invertible, even if  $X$  is not square; simply consider  $X' = X^T$ , where  $X'X'^T$  is a full rank 2 by 2 matrix (2 is false). Similarly,  $XX^T$  may be invertible even if the columns of  $X$  are linearly dependent; consider matrix

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}.$$

We see that  $AA^T$  is an invertible 2 by 2 matrix, even though columns 1 and 3 are the same (3 is false).

If  $n$  is smaller than  $d$ , then  $X$  has maximum rank  $n$ , and so  $XX^T$  has maximum rank  $n$ . Since  $XX^T$  is a  $d$  by  $d$  matrix, where  $d > n$ ,  $XX^T$  is not invertible (5 is true).

Finally, we can calculate  $X^T$  for any  $X$ , regardless of its rank (6 is false).

## 5) Evaluation

We have been hired by Buy 'n' Large to deliver a predictor of change in sales volume from last year, for each of their stores. We have a machine-learning algorithm that can be used with regularization parameter  $\lambda$ . Our overall objective is to deliver a predictor that minimizes squared loss on predictions when actually used by the company. We have three data sets:  $D_{train}$ ,  $D_{test}$  and  $D_{real}$ , each of size  $n$ . The  $D_{real}$  is owned by the company.

We will focus on a linear predictor with parameters  $\theta$  without the offset parameter  $\theta_0$  for simplicity and use regularizer  $\lambda\|\theta\|^2$ , where  $\lambda$  is the regularization parameter. There are several phases of the training process (as represented in problems 5A through 5D below), and we need to select the appropriate objective for each of those tasks. In the problems below, we will have different expressions with different "slots" (A, B, C, D) to fill from, picking among the following options available to us related to

- what the minimization is over,
- the dataset used,
- the predictor used, and
- whether regularization is added.

Fill in the slots (A,B,C,D) for each phase by choosing the expressions for the indicated slots. The available expressions are shown below; please enter the index for the expression for each of the slots.

1. 0
2.  $\theta$
3.  $\theta_{best}(\lambda)$ ,
4.  $\theta^*$ ,
5.  $\lambda$ ,
6.  $\lambda^*$ ,
7.  $\lambda\|\theta\|^2$ ,
8.  $\lambda^*\|\theta\|^2$ ,
9.  $D_{train}$ ,
10.  $D_{test}$ ,
11.  $D_{train} \cup D_{test}$ ,
12.  $D_{real}$

Note that  $\theta_{best}(\lambda)$  is a value of  $\theta$  that is a function of  $\lambda$ ;  $\theta^*$ ,  $\lambda^*$  are specific values found as described below;  $\theta$  and  $\lambda$  are variables that range over  $d$ -dimensional column vectors and positive reals, respectively.

**5A)** Selecting the best hypothesis (parameters  $\theta$ ) for some fixed value of the regularization parameter  $\lambda$ . Call this  $\theta_{best}(\lambda)$ .

$$\theta_{best}(\lambda) = \arg \min_A \frac{1}{|B|} \sum_{(x,y) \in B} (C^T x - y)^2 / 2 + D$$

Enter a list of 4 indices for [A, B, C, D]:



**100.00%**

*You have infinitely many submissions remaining.*

**5B)** Selecting the best value of the regularization parameter  $\lambda$ . We will call this best value  $\lambda^*$ .

$$\lambda^* = \arg \min_A \frac{1}{|B|} \sum_{(x,y) \in B} (C^T x - y)^2 / 2 + D$$

Enter a list of 4 indices for [A, B, C, D]:



**100.00%**

*You have infinitely many submissions remaining.*

**5C)** Selecting the hypothesis (parameters  $\theta$ ) to deliver to the company. Call this  $\theta^*$ .

$$\theta^* = \arg \min_A \frac{1}{|B|} \sum_{(x,y) \in B} (C^T x - y)^2 / 2 + D$$

Enter a list of 4 indices for [A, B, C, D]:



**100.00%**

*You have infinitely many submissions remaining.*

**5D)** Evaluating the actual on-the-job performance  $\epsilon^*$  of the selected hypothesis  $\theta^*$ .

$$\epsilon^* = \frac{1}{|B|} \sum_{(x,y) \in B} (C^T x - y)^2 / 2 + D$$

Enter a list of 3 indices for [B, C, D]:



**100.00%**

*You have infinitely many submissions remaining.*

## 6) Linear regression - going downhill

We will now write some general Python code to compute the gradient of the squared-loss objective, following the structure of the expression and the rules of calculus. Note that this style of writing the gradient functions maps directly into the chain-rule steps required to compute the gradient, but produces code that is inefficient, because of duplicated computations. It is straightforward to implement more efficient versions if you want to use them for larger problems.

**Reminder:** For your convenience, we have copied the hands-on section into a colab notebook, [which may be found here..](#) Alternatively, you can work with these functions on your own computer in `code_for_hw5.py`, contained in [this zip file](#). That file has somewhat longer docstrings and doctests for many of these functions and other basic utilities, that may be useful to you in debugging your implementations. (The other files therein will be useful in the last part of this homework).

We start by defining some basic functions for computing the mean squared loss. Note that we want these to work for any value of  $n$ , that is,  $x$  could be a single feature vector (of dimension  $d$  by 1) or a full data matrix (of dimension  $d$  by  $n$ ), and similarly for  $y$ .

```
# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar

def lin_reg(x, th, th0):
    return np.dot(th.T, x) + th0

def square_loss(x, y, th, th0):
    return (y - lin_reg(x, th, th0))**2

def mean_square_loss(x, y, th, th0):
    # the axis=1 and keepdims=True are important when x is a full matrix
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True)
```

These functions will already be defined when you are answering the questions below.

Warm up:

6A)

If  $X$  is  $d$  by  $n$  and  $Y$  is 1 by  $n$ , what is the dimension of  $\theta$ ?

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

6B)

If  $X$  is  $d$  by  $n$  and  $Y$  is 1 by  $n$ , what is the dimension of  $\nabla_{\theta} J_{emp}(\theta, \theta_0)$ ?

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**6C)** Now let's compute the gradients with respect to  $\theta$ , make sure that they work for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently.



In the code below, the following values are used in the test cases:

```
X = np.array([[1., 2., 3., 4.], [1., 1., 1., 1.]])
Y = np.array([[1., 2.2, 2.8, 4.1]])
th = np.array([[1.0],[0.05]])
th0 = np.array([[0.]])
```

```
1 # Write a function that returns the gradient of lin_reg(x, th, th0)
2 # with respect to th
3 def d_lin_reg_th(x, th, th0):
4     return x
5
6 # Write a function that returns the gradient of square_loss(x, y, th, th0) with
7 # respect to th. It should be a one-line expression that uses lin_reg and
8 # d_lin_reg_th.
9 def d_square_loss_th(x, y, th, th0):
10     return 2* (y - lin_reg(x, th, th0)) * - d_lin_reg_th(x, th, th0)
11
12 # Write a function that returns the gradient of mean_square_loss(x, y, th, th0) with
13 # respect to th. It should be a one-line expression that uses d_square_loss_th.
14 def d_mean_square_loss_th(x, y, th, th0):
15     return np.mean(d_square_loss_th(x, y, th, th0), axis = 1, keepdims = True)
16 |
```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**6D)** Now let's compute the gradients with respect to  $\theta_0$ , make sure that they work for data matrices and label vectors. You can write one function at a time, some of the checks will apply to each function independently. The test cases will include example variables for  $x$ ,  $y$ ,  $th$ , and  $th0$  from 6C above.

```

1 # Write a function that returns the gradient of lin_reg(x, th, th0)
2 # with respect to th0. Hint: Think carefully about what the dimensions of the returned
3 def d_lin_reg_th0(x, th, th0):
4     d, n = x.shape
5     return np.ones((1,n))
6
7 # Write a function that returns the gradient of square_loss(x, y, th, th0) with
8 # respect to th0. It should be a one-line expression that uses lin_reg and
9 # d_lin_reg_th0.
10 def d_square_loss_th0(x, y, th, th0):
11     return 2 * (y - lin_reg(x, th, th0)) * - d_lin_reg_th0(x, th, th0)
12
13 # Write a function that returns the gradient of mean_square_loss(x, y, th, th0) with
14 # respect to th0. It should be a one-line expression that uses d_square_loss_th0.
15 def d_mean_square_loss_th0(x, y, th, th0):
16     return np.mean(d_square_loss_th0(x, y, th, th0), axis = 1, keepdims = True)
17 |

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 7) Going down the ridge

Now, let's add a regularizer. The ridge objective can be implemented as follows:

```

# In all the following definitions:
# x is d by n : input data
# y is 1 by n : output regression values
# th is d by 1 : weights
# th0 is 1 by 1 or scalar
def ridge_obj(x, y, th, th0, lam):
    return np.mean(square_loss(x, y, th, th0), axis = 1, keepdims = True) + lam * np.linalg.norm(th)**2

```

Let's extend our previous code for the gradient of the mean square loss to compute the gradient of the ridge objective with respect to  $\theta$ . Our previous solutions for the non-ridge case: `d_mean_square_loss_th` and `d_mean_square_loss_th0` are defined for you and you can call them. The test cases will include example variables for `x`, `y`, `th`, and `th0` from 6C above.

```

1 def d_ridge_obj_th(x, y, th, th0, lam):
2     return d_mean_square_loss_th(x, y, th, th0) + 2 * lam * th
3
4 def d_ridge_obj_th0(x, y, th, th0, lam):
5     return d_mean_square_loss_th0(x, y, th, th0)
6

```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 8) Stochastic gradient

We will now implement [stochastic gradient descent](#) in a general way, similar to what we did with gradient descent (`gd`).

`sgd` takes the following as input: (Recall that the *stochastic* part refers to using a randomly selected point and corresponding label from the given dataset to perform an update. Therefore, your objective function for a given step will need to take this into account.)

- `x`: a standard data array ( $d$  by  $n$ )
- `y`: a standard labels row vector ( $1$  by  $n$ )
- `J`: a cost function whose input is a data point (a column vector), a label ( $1$  by  $1$ ) and a weight vector  $w$  (a column vector) (in that order), and which returns a scalar.
- `dJ`: a cost function gradient (corresponding to `J`) whose input is a data point (a column vector), a label ( $1$  by  $1$ ) and a weight vector  $w$  (a column vector) (also in that order), and which returns a column vector.
- `w0`: an initial value of weight vector  $w$ , which is a column vector.
- `step_size_fn`: a function that is given the (zero-indexed) iteration index (an integer) and returns a step size.
- `max_iter`: the number of iterations to perform

It returns a tuple (like `gd`):

- `w`: the value of the weight vector at the final step
- `fs`: the list of values of  $J$  found during all the iterations
- `ws`: the list of values of  $w$  found during all the iterations

**Note:** `w` should be the value one gets after applying stochastic gradient descent to `w0` for `max_iter-1` iterations (we call this the final step). The first element of `fs` should be the value of `J` calculated with `w0`, and `fs` should have length `max_iter`; similarly, the first element of `ws` should be `w0`, and `ws` should have length `max_iter`.

You might find the function `np.random.randint(n)` useful in your implementation.

**Hint:** This is a short function; our implementation is around 10 lines.

The test cases are:

```
def downwards_line():
    X = np.array([[0.0, 0.1, 0.2, 0.3, 0.42, 0.52, 0.72, 0.78, 0.84, 1.0],
                  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]])
    y = np.array([[0.4, 0.6, 1.2, 0.1, 0.22, -0.6, -1.5, -0.5, -0.5, 0.0]])
    return X, y

X, y = downwards_line()

def J(Xi, yi, w):
    # translate from (1-augmented X, y, theta) to (separated X, y, th, th0) format
    return float(ridge_obj(Xi[:-1,:], yi, w[:-1,:], w[-1:], 0))

def dJ(Xi, yi, w):
    def f(w): return J(Xi, yi, w)
    return num_grad(f)(w)
```

where num\_grad is taken from homework from the previous week:

```
def num_grad(f):
    def df(x):
        g = np.zeros(x.shape)
        delta = 0.001
        for i in range(x.shape[0]):
            xi = x[i,0]
            x[i,0] = xi - delta
            xm = f(x)
            x[i,0] = xi + delta
            xp = f(x)
            x[i,0] = xi
            g[i,0] = (xp - xm)/(2*delta)
        return g
    return df
```

```
17     Xi, yi = get_rnd_xiyi(X, y)
18     #append current data points to lists
19     w_list.append(w)
20     J_list.append(J(Xi, yi, w))
21     #apply the gradient on this data point
22     grad = dJ(Xi, yi, w)
23     #apply the step
24     w = w - step_size_fn(iter) * grad
25     if iter == max_iter-2:
26         #if this is the last iter, we want to add the
27         #w and j at this point without calculating a new w
28         #randomly select a data point
29         Xi, yi = get_rnd_xiyi(X, y)
30         #append current data points to lists
31         w_list.append(w)
```

Run Code

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 9) Predicting mpg values

We will now try to synthesize the functions we have written in order to perform ridge regression on the [auto-mpg dataset](#) from [lab03](#). Unlike in lab03, we will now try to predict the actual mpg values of the cars, instead of whether they are above or below the median mpg!

As a reminder, the dataset is as follows:

1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous
5. weight: continuous
6. acceleration: continuous
7. model year: multi-valued discrete
8. origin: multi-valued discrete
9. car name: string (many values)

For convenience, we will choose to not include `model year` and `car name` as features. For the remaining features, we again have the option to keep the raw values, standardize them, or use a one-hot encoding.

**9A)** What is true about leaving features as raw versus deciding to standardize them, in the context of linear regression without regularization?




- ☐ The set of all possible linear regression models learnable on standardized features is smaller than the set of linear regression models learnable on raw features
- ☒ The theoretical minimum value of the loss function is the same both with raw and standardized features
- ☒ SGD will typically perform better on standardized features than on raw features.

[Ask for Help](#)

100.00%

You have infinitely many submissions remaining.

Solution:

-  The set of all possible linear regression models learnable on standardized features is smaller than the set of linear regression models learnable on raw features
-  The theoretical minimum value of the loss function is the same both with raw and standardized features
-  SGD will typically perform better on standardized features than on raw features.

#### Explanation:

1. The set of linear regression models is the same, in both cases. Say that  $\theta$  and  $\theta_0$  are the parameters of a predictor operating on standardized features. Then, the prediction for the example  $x$  will be  $\hat{y} = \theta \cdot \left(\frac{x-\mu}{\sigma}\right) + \theta_0$ , where  $\mu$  and  $\sigma$  represent the standardization. Note that this is the same as  $\hat{y} = \tilde{\theta} \cdot x + \tilde{\theta}_0$ , where  $\tilde{\theta} = \frac{\theta}{\sigma}$  and  $\tilde{\theta}_0 = \theta_0 - \frac{\theta \cdot \mu}{\sigma}$ . For any fixed  $\mu, \sigma$ , we can vary  $\theta$  and  $\theta_0$  to make  $\tilde{\theta}$  and  $\tilde{\theta}_0$  equal to whatever values we want. Thus, we can represent any linear predictor with an appropriate choice of parameters in a linear predictor operating on standardized features.

2. From the explanation for 1, the set of possible models is the same with and without standardization. Thus, the predictor minimizing the error in both cases will be the same, causing the theoretical minimum of the loss function to also be the same.

3. At every iteration, SGD takes a step in the direction opposite to the (stochastic) gradient, with the same step size in each "direction" of parameter space. However, with raw features, you would naturally want larger step sizes to update parameters which have a larger range, and smaller ones for those which have a small range. Using a large step size can then often lead to divergence of SGD due to the parameters with a smaller range; using a small step size can lead to very slow convergence due to those parameters with a larger range. Standardizing features largely removes this problem, making it "more acceptable" to use a constant step size across all directions.

**9B)** What is true about leaving features as raw versus deciding to standardize them, in the context of ridge regression (i.e., we have a nonzero regularizer)?




- ☐ The set of all possible models learnable on standardized features is smaller than the set of models learnable on raw features
- ☐ The theoretical minimum value of the loss function is the same both with raw and standardized features
- ☒ SGD will typically perform better on standardized features than on raw features.

[Ask for Help](#)

100.00%

You have infinitely many submissions remaining.

Solution:

-  The set of all possible models learnable on standardized features is smaller than the set of models learnable on raw features
-  The theoretical minimum value of the loss function is the same both with raw and standardized features
-  SGD will typically perform better on standardized features than on raw features.

**Explanation:**

1. The set of learnable models is the set of all linear models, as in the previous question. Thus, the answer is the same.
2. While the set of learnable models is the same, for raw features, the regularizer penalizes certain feature weights much more than others. For instance, if the magnitude of a particular feature is very small in the raw data, the learned weight in the absence of a regularizer might be very large. When adding regularization, this feature weight will be driven to be closer to 0, so this feature is disproportionately affected by the regularizer. The optimal weights for both objectives will not then generically correspond to the same model, and the loss values will be different in both cases.
3. Same reason as in 9A.

With this considered, we decide to standardize or one-hot encode all features in this section (we encourage you, though, to try raw features on your own to see how their performance matches your expectations!).

One additional step we perform is to standardize the output values. Note that we did not have to worry about this in a classification context, as all outputs were  $\pm 1$ . In a regression context, standardizing the output values can have practical performance gains, again due to better numerical performance of learning algorithms on data that is in a good magnitude range.

The metric we will use to measure the quality of our learned predictors is **Root Mean Square Error (RMSE)**. This is useful metric because it gives a sense of the deviation in the nature units of the predictor. RMSE is defined as follows:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - f(x^{(i)}))^2}$$

where  $f$  is our learned predictor: in this case,  $f(x) = \theta \cdot x + \theta_0$ . This gives a measure of how far away the true values are from the predicted values; we are interested in this value, measured in units of mpg.

**Note:** One very important thing to keep in mind when employing standardization is that we need to reverse the standardization when we want to report results. If we standardize output values in the training set by subtracting  $\mu$  and dividing by  $\sigma$ , we need to take care to:

1. Perform standardization with the same values of  $\mu$  and  $\sigma$  on the test set (Why?) before predicting outputs using our learned predictor.
2. Multiply the RMSE calculated on the test set by a factor of  $\sigma$  to report test error. (Why?)

Given all of this, we now will try using:

- Two choices of feature set:
  1. [cylinders=standard, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one\_hot]
  2. [cylinders=one\_hot, displacement=standard, horsepower=standard, weight=standard, acceleration=standard, origin=one\_hot]
- Polynomial features (we will construct the polynomial features after having standardized the input data) of orders 1-3
- Different choices of the regularization parameter,  $\lambda$ . Although, ideally, you would run a grid search over a large range of  $\lambda$ , we will ask you to look at the choices  $\lambda = \{0.0, 0.01, 0.02, \dots, 0.1\}$  for polynomial features of orders 1 and 2, and the choices  $\lambda = \{0, 20, 40, \dots, 200\}$  for polynomial features of order 3 (as this is approximately where we found the optimal  $\lambda$  to lie).

We will use 10-fold cross-validation to try all possible combinations of these feature choices and test which is best. We have attached a code file with some predefined methods that will be useful to you [here](#). Alternatively, a google colab link [may be found here](#). If you choose to use the code file, a more detailed description of the roles of the files is below:

The file `code_for_hw5.py` contains functions, some of which will need to be filled in with your definitions from this homework. Your functions are then called by `ridge_min`, defined for you, which takes a dataset  $(X, y)$  and a hyperparameter,  $\lambda$  as input and returns  $\theta$  and  $\theta_0$  minimizing the ridge regression objective using SGD (this is the analogue of the `svm_min` function that you wrote for homework last week). The learning rate and number of iterations are fixed in this function, and should not be modified for the purpose of answering the below questions (although you should feel free to experiment with these if you are interested!). This function will then further be called by `xval_learning_alg` (also defined for you in the same file), which returns the average RMSE across all (here, 10) splits of your data when performing cross-validation. (Note that this RMSE is reported in standardized  $y$  units; to convert this to RMSE in mpg (miles per gallon), you should multiply this by the `sigma` returned by the `hw5.std_y` function call.)

The file `auto.py` will be used to implement the auto data regression. The file contains code for creating the two feature sets that you are asked to work with here. Transforming those features further with `make_polynomial_feature_fun`, and running the cross-validation function, which uses your implementations in `code_for_hw5.py` (both from `code_for_hw5.py`), you should be able to answer the following questions:

**9C)** What combination minimizes the average cross-validation RMSE?



Enter a tuple of three numbers (feature\_set, polynomial\_order, lambda):

[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

Solution: (2, 2, 0.0)

**Explanation:**

(2, 2, 0.0) gave us the lowest RMSE. We found that  $\lambda$  had a relatively modest impact on RMSE, with small (about zero) values of  $\lambda$  giving the best results, for feature set 2 with 2nd order polynomials.

**9D)** What is the cross-validation RMSE value (in mpg) that you obtain using the best combination?

Enter an accuracy value:

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

**9E)** Say that we really wanted to fit an order 3 polynomial model using the first feature set. What value of lambda minimizes the average cross-validation RMSE, and what is the RMSE value at that value of lambda?

Enter a python list of two numbers [lambda, RMSE\_value]:

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*

This homework builds on the material in the notes on [neural networks](#) up through and including section 6 on loss functions.

In particular, in this homework we consider neural networks with multiple layers. Each layer has multiple inputs and outputs, and can be broken down into two parts:

- A **linear** module that implements a linear transformation:  $z_j = (\sum_{i=1}^m x_i w_{i,j}) + w_{0,j}$  specified by a weight matrix  $W$  and a bias vector  $W_0$ . The output is  $[z_1, \dots, z_n]^T$ .
- An **activation** module that applies an activation function to the outputs of the linear module for some activation function  $f$ , such as Tanh or ReLU in the hidden layers or Softmax (see below) at the output layer. We write the output as:  $[f(z_1), \dots, f(z_m)]^T$ , although technically, for some activation functions such as softmax, each output will depend on all the  $z_i$ , not just one.

We will use the following notation for quantities in a network:

- Inputs to the network are  $x_1, \dots, x_d$ .
- Number of layers is  $L$
- There are  $m^l$  inputs to layer  $l$
- There are  $n^l = m^{l+1}$  outputs from layer  $l$
- The weight matrix for layer  $l$  is  $W^l$ , an  $m^l \times n^l$  matrix, and the bias vector (offset) is  $W_0^l$ , an  $n^l \times 1$  vector
- The outputs of the linear module for layer  $l$  are known as **pre-activation** values and denoted  $z^l$
- The activation function at layer  $l$  is  $f^l(\cdot)$
- Layer  $l$  activations are  $a^l = [f^l(z_1^l), \dots, f^l(z_{n^l}^l)]^T$
- The output of the network is the values  $a^L = [f^L(z_1^L), \dots, f^L(z_{n^L}^L)]^T$
- Loss function  $Loss(a, y)$  measures the loss of output values  $a$  when the target is  $y$

## 1) Loss functions and output activations: classification

When doing classification, it's natural to think of the output values as being discrete: +1 and -1. But it is generally difficult to use optimization-based methods without somehow thinking of the outputs as being continuous (even though you will have to discretize when it's time to make a prediction).

### 1.1) Hinge loss, linear activation

When we looked at the SVM objective for classification, we did this:

- Defined the output space to be  $\mathbb{R}$
- Developed the hinge loss function

$$Loss(a, y) = L_h(ya) = \begin{cases} 0 & \text{if } ya > 1 \\ 1 - ya & \text{otherwise} \end{cases}$$

where  $a$  is the continuous output (we're using  $a$  here to be consistent with the neural network terminology of *activation*) and  $y$  is the desired/target output

- Tried to find parameters  $\theta$  of our model to minimize loss summed over the training data

Consider a single "neuron" with a linear activation function; that is, where  $a_1^L = \sum_k w_{k,1}^L x_k + w_{0,1}^L$ . In this case, we have  $L = 1$  and  $f^L(z) = z$ .

**1.1.A)** Write a short program to compute the gradient of the loss function with respect to the weight vector (not the bias):  $\nabla_{w^L} \text{Loss}(a_1^L, y)$  when  $\text{Loss}(a, y) = L_h(ya)$ .

- $x$  is a column vector
- $y$  is a number, a label
- $a$  is a number, an activation

It should return a column vector.

```
1 def hinge_loss_grad(x, y, a):
2     d, n = x.shape
3     if a * y > 1:
```

Run Code

Submit

View Answer

Ask for Help

100.00%

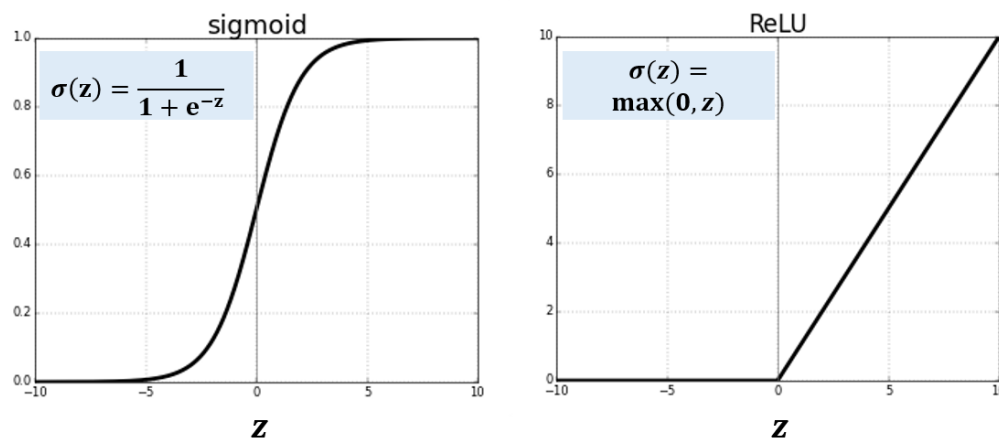
You have infinitely many submissions remaining.

## 1.2) Log loss, sigmoidal activation

Another way to make the output for a classifier continuous is to make it be in the range  $(0, 1)$ , which admits the interpretation of being the predicted **probability** that the example is positive. A convenient way to make the activation of a unit be in the range  $(0, 1)$  is to use a *sigmoid* function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

The figure below shows a sigmoid activation function on the left, with the rectified linear (ReLU) activation function on the right for comparison.



**1.2.A)** What is an expression for the derivative of the sigmoid with respect to  $z$ , expressed as a function of  $z$ , its input?

Enter a Python expression (use `**` for exponentiation) involving `e` and `z`:

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1.2.B)** What is an expression for the derivative of the sigmoid with respect to  $z$ , but this time expressed as a function of  $o = \sigma(z)$ , its output?

Hint: Think about the expression  $1 - \frac{1}{1+e^{-z}}$ . (Here is a [review](#) of computing derivatives.)

Enter a Python expression (use `**` for exponentiation) involving only `o` (note: `e` and `z` are not allowed, and remember  $o = \sigma(z)$ ):

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**In this model, we will consider positive points to have label +1, and negative points to have label 0.**

We need a loss function that works well when we are predicting probabilities. A good choice is to ask what probability is assigned to the correct label. We will interpret the value outputted by our classifier as the probability that the example is positive. So, if the output value is  $a$  and the true label is  $+1$ , then the probability assigned to the true label is  $a$ ; on the other hand, if the true label is  $0$ , then the probability assigned to the true label is  $1 - a$ . Because we actually will be interested in the probability of the predictions on the whole data set, we'd want to choose weights to **maximize**

$$\prod_t P(a^{(t)}, y^{(t)})$$

where  $P(a^{(t)}, y^{(t)})$  is the probability that the network predicts the correct label for data point  $(t)$ .

Using a notational trick (which turns an *if* expression into a product) that might seem unmotivated now, but will be useful later, we can write the probability  $P(a, y)$  as

$$P(a, y) = a^y (1 - a)^{(1-y)}.$$

**1.2.C)** What is the value of  $P(a, y)$  when  $y = 0$ ?

Enter an expression for  $P(a, y)$  when  $y = 0$  in terms of  $a$ :

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1.2.D)** What is the value of  $P(a, y)$  when  $y = 1$ ?

Enter an expression for  $P(a, y)$  when  $y = 1$  in terms of  $a$ :

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1.2.E)** Find a simplified expression for  $\log P(a, y)$  that does not use exponentiation. Note that we refer to the natural logarithm  $\ln$  as  $\log$  throughout this assignment, consistent with the lecture notes.

Enter an expression in terms of  $y$  and  $a$ ; you can use  $\log(\cdot)$  to indicate natural log:

$y \cdot \log(a) + (1-y) \cdot \log(1-a)$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

In fact, because  $\log$  is a monotonic function, the same weights that maximize the product of the probabilities will minimize the *negative log likelihood* ("likelihood" is the same as probability; we just use that name here because the phrase is an idiom in machine learning, abbreviated NLL):

$$\text{Loss}(a, y) = \text{NLL}(a, y) = -y \log a - (1 - y) \log(1 - a).$$

Our objective function (over our  $n$  data points) will then be

$$\sum NLL(a^{(t)}, y^{(t)}) = - \sum_{t=1}^n \left[ y^{(t)} \log a^{(t)} + (1 - y^{(t)}) \log(1 - a^{(t)}) \right].$$

Remember that  $a^{(t)}$  is our model's output for training example  $t$ , and  $y^{(t)}$  is the true label (+1 or 0).

Now, we can think about a single unit with a sigmoidal activation function, trained to minimize NLL. So,  $a_1^L = \sigma(\sum_k w_{k,1}^L x_k + w_{0,1}^L)$ . In this case, we have  $L = 1$ .

**1.2.F)** Write a formula for the gradient of the NLL with respect to the first weight,  $\nabla_{w_{1,1}^L} NLL(a_1^L, y)$ , for a single training example. Hint: consider using the chain rule; the final answer (expression) is very short.

Write an expression in terms of  $x_1$ ,  $a_1$ , and  $y$ :  $(a_1 - y) * x_1$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**1.2.G)** Write a formula for the gradient of the NLL with respect to the full weight vector,  $\nabla_{w^L} NLL(a_1^L, y)$ , for a single training example.

Enter an expression in terms of  $x$ ,  $a_1$ , and  $y$ :  $(a_1 - y) * x$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

## 2) Multiclass classification

What if we needed to classify homework problems into three categories: enlightening, boring, impossible? We can do this by using a "one-hot" encoding on the output, and using three output units with what is called a "softmax" (SM) activation module. It's not a typical activation module, since it takes in all  $n_L$  pre-activation values  $z_j^L$  in  $\mathbb{R}$  and returns  $n_L$  output values  $a_j^L \in [0, 1]$  such that  $\sum_j a_j^L = 1$ . This can be interpreted as representing a probability distribution over the possible categories.

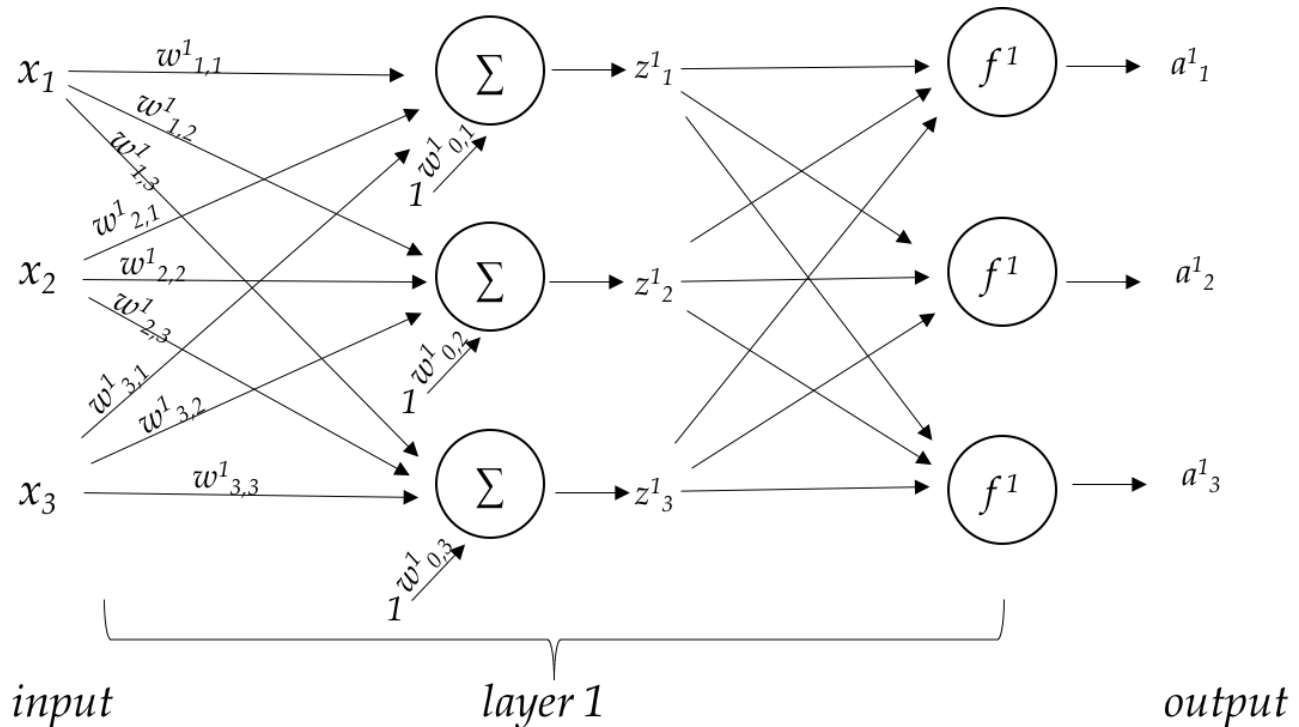
The individual entries are computed as

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{n^L} e^{z_k}}$$

We'll describe the relationship of the vector  $a$  on the vector  $z$  as

$$a = \text{SM}(z)$$

The network below shows a one-layer network with a linear module followed by a softmax activation module.



**2.A)** What probability distribution over the categories is represented by  $z^L = [-1, 0, 1]^T$ ?

Enter a distribution (a list of three numbers adding up to 1) for the three categories. Your answers should be numeric (please enter numbers, and do not use the symbol  $e$ ):





100.00%

You have infinitely many submissions remaining.

Now, we need a loss function  $Loss(a, y)$  where  $a$  is a discrete probability distribution and  $y$  is a one-hot vector encoding of a single output value. It makes sense to use negative log likelihood as a loss function for the same reasons as before. So, we'll just extend our definition of NLL from earlier:

$$NLL(a, y) = - \sum_{j=1}^{n^L} y_j \ln a_j^L.$$

Note that the above expression is for multi-classes (number of class  $> 2$ ). For two-classes, the expression reduce to what you saw after Problem 1.2.E.

**2.B)** If  $a = [.3, .5, .2]^T$  and  $y = [0, 0, 1]^T$ , what is  $NLL(a, y)$ ?

Enter an expression involving  $\log(\cdot)$  (for natural log) and constants:

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Now, we can think about a single layer with a softmax activation module, trained to minimize NLL. The pre-activation values (the output of the linear module) are:

$$z_j^L = \sum_k w_{k,j}^L x_k + w_{0,j}^L$$

and  $a^L = SM(z^L)$ .

To do gradient descent, we need to know  $\frac{\partial}{\partial w_{k,j}^L} NLL(a^L, y)$ . We'll reveal the secret (that you might guess from Problem 1) that it has an awesome form! (Please consider deriving this, for fun and satisfaction!)

$$\frac{\partial}{\partial w_{k,j}^L} NLL(a^L, y) = x_k(a_j^L - y_j)$$

And of course, it's easy to compute the whole matrix of these derivatives,  $\nabla_{W^L} NLL(a^L, y)$ , in one quick matrix computation.

**2.C)** Suppose we have two input units and three possible output values, and the weight matrix  $W^L$  is

$$W^L = \begin{bmatrix} 1 & -1 & -2 \\ -1 & 2 & 1 \end{bmatrix}$$

or in Python form: `w = np.array([[1, -1, -2], [-1, 2, 1]])`.

Assume the biases are zero, the input  $x = [1, 1]^T$  (e.g., `x = np.array([[1, 1]]).T`), and the target output  $y = [0, 1, 0]^T$  (e.g., `y = np.array([[0, 1, 0]]).T`). What is the matrix  $\nabla_{W^L} NLL(a^L, y)$ ? Hint: You might want to solve using Python and numpy, or using [colab](#) for calculation.

Enter the matrix as a list of lists, one list for each row of the matrix. Please enter values with a precision of three decimal points.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2.D)** What is the predicted probability that  $x$  is in class 1, before any gradient updates? (Assume we have classes 0, 1, and 2.)

Enter a number:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**2.E)** Using step size 0.5, what is  $W^L$  after one gradient update step?

Enter the matrix as a list of lists, one list for each row of the matrix. Please enter values with a precision of three decimal points.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

2.F) What is the predicted probability that  $x$  is in class 1, given the new weight matrix?

Enter a number:

Submit

View Answer

Ask for Help

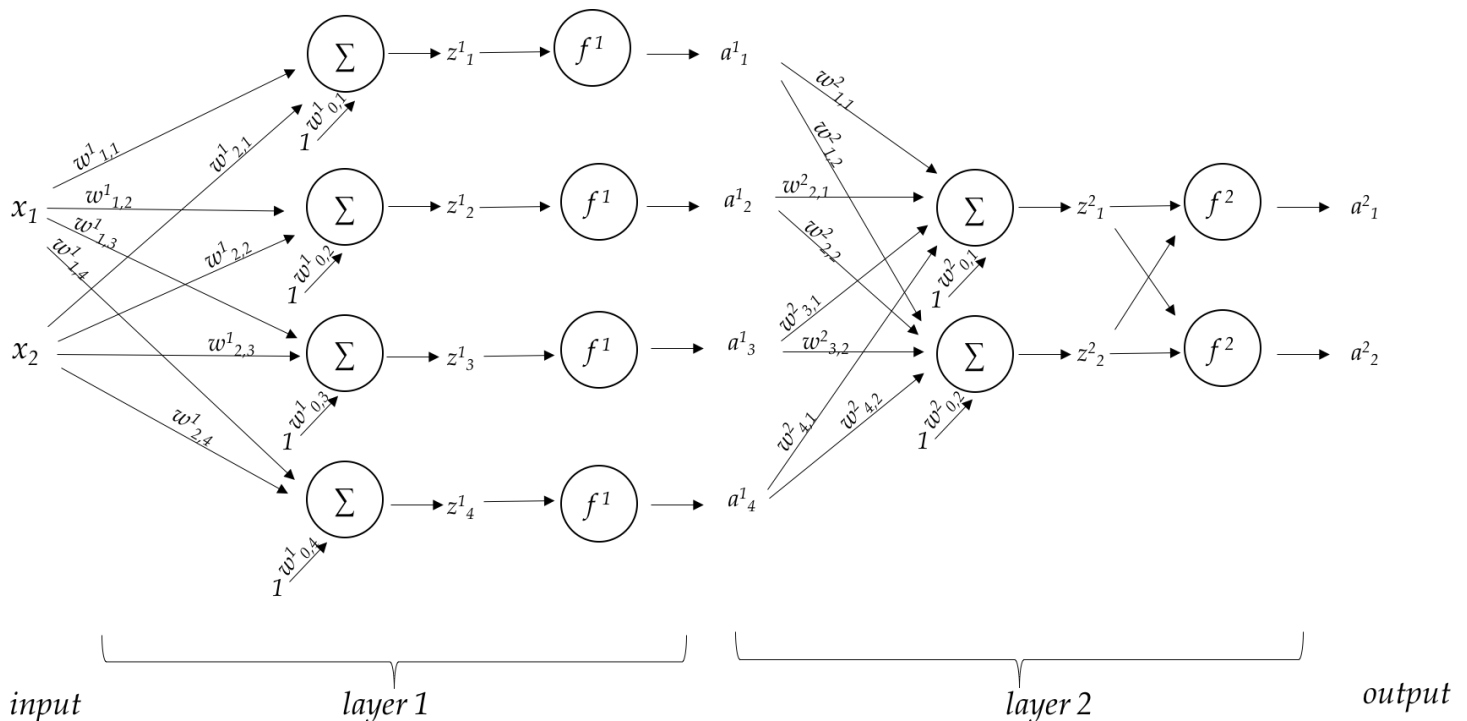
100.00%

You have infinitely many submissions remaining.

### 3) Neural Networks

In this problem we will analyze a simple neural network to understand its classification properties. You might find the [colab](#) file useful. However, we encourage you to go through all the calculation by hand once, which should be a good practice.

Consider the neural network given in the figure below, with ReLU activation functions ( $f^1$  in the figure) on all hidden neurons, and softmax activation ( $f^2$  in the figure) for the output layer, resulting in softmax outputs ( $a_1^2$  and  $a_2^2$  in the figure).



Given an input  $x = [x_1, x_2]^T$ , the hidden units in the network are activated in stages as described by the following equations:

$$\begin{aligned}
 z^1_1 &= x_1 w^1_{1,1} + x_2 w^1_{2,1} + w^1_{0,1} & a^1_1 &= \max\{z^1_1, 0\} \\
 z^1_2 &= x_1 w^1_{1,2} + x_2 w^1_{2,2} + w^1_{0,2} & a^1_2 &= \max\{z^1_2, 0\} \\
 z^1_3 &= x_1 w^1_{1,3} + x_2 w^1_{2,3} + w^1_{0,3} & a^1_3 &= \max\{z^1_3, 0\} \\
 z^1_4 &= x_1 w^1_{1,4} + x_2 w^1_{2,4} + w^1_{0,4} & a^1_4 &= \max\{z^1_4, 0\}
 \end{aligned}$$



$$z_1^2 = a_1^1 w_{1,1}^2 + a_2^1 w_{2,1}^2 + a_3^1 w_{3,1}^2 + a_4^1 w_{4,1}^2 + w_{0,1}^2$$

$$z_2^2 = a_1^1 w_{1,2}^2 + a_2^1 w_{2,2}^2 + a_3^1 w_{3,2}^2 + a_4^1 w_{4,2}^2 + w_{0,2}^2.$$

The final output of the network is obtained by applying the *softmax* function to the last hidden layer,

$$a_1^2 = \frac{e^{z_1^2}}{e^{z_1^2} + e^{z_2^2}}$$

$$a_2^2 = \frac{e^{z_2^2}}{e^{z_1^2} + e^{z_2^2}}.$$

In this problem, we will consider the following setting of parameters:

$$\begin{bmatrix} w_{1,1}^1 & w_{1,2}^1 & w_{1,3}^1 & w_{1,4}^1 \\ w_{2,1}^1 & w_{2,2}^1 & w_{2,3}^1 & w_{2,4}^1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad \begin{bmatrix} w_{0,1}^1 \\ w_{0,2}^1 \\ w_{0,3}^1 \\ w_{0,4}^1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix},$$

$$\begin{bmatrix} w_{1,1}^2 & w_{1,2}^2 \\ w_{2,1}^2 & w_{2,2}^2 \\ w_{3,1}^2 & w_{3,2}^2 \\ w_{4,1}^2 & w_{4,2}^2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}, \quad \begin{bmatrix} w_{0,1}^2 \\ w_{0,2}^2 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}.$$

### 3.1) Output

Consider the input  $x_1 = 3, x_2 = 14$ .

**3.1.A)** What are the outputs of the hidden units,  $(f^1(z_1^1), f^1(z_2^1), f^1(z_3^1), f^1(z_4^1))$ .

Enter a Python list of 4 numbers:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.1.B)** What is the final output  $(a_1^2, a_2^2)$  of the network?

Enter a Python list of 2 numbers:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

### 3.2) Unit decision boundaries

Let's characterize the *decision boundaries* in  $x$ -space, corresponding to the four hidden units. These are the regions where the input to the units  $z_1^1, z_2^1, z_3^1, z_4^1$  are exactly zero.

Hint: You should draw a diagram of the decision boundaries for each unit in the  $x$ -space and label the sides of the boundaries with 0 and + to indicate whether the unit's output would be exactly 0 or positive, respectively. (The diagram should be a 2D plot with  $x_1$  and  $x_2$  on each axis, with lines for  $z_1^1 = 0$ ,  $z_2^1 = 0$ ,  $z_3^1 = 0$ ,  $z_4^1 = 0$ .)

**3.2.A)** What is the shape of the decision boundary for a single unit?

Choose one:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.2.B)** Enter a 2 x 4 matrix where each column represents a (different) input vector  $[x_1, x_2]^T$  each of which is on the decision boundary for the first unit, that is, for which  $z_1^1 = 0$ . (There are multiple possible answers.)

Enter a Python list of lists where each list is a row of the matrix.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.2.C)** Consider the following input vectors:  $x^{(1)} = [0.5, 0.5]^T$ ,  $x^{(2)} = [0, 2]^T$ ,  $x^{(3)} = [-3, 0.5]^T$ . Enter a matrix where each column represents the outputs of the hidden units ( $f(z_1^1), \dots, f(z_4^1)$ ) for each of the input vectors. You can use your diagram of decision boundaries.

Enter a Python list of lists where each list is a row of the matrix.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

### 3.3) Network outputs

In our network above, the output layer with two softmax units is used to classify into one of two classes. For class 1, the first unit's output should be larger than the other unit's output, and for class 2, the second unit's output should be larger. This generalizes nicely to  $k$  classes by using  $k$  output units.

(We have previously examined addressing two-class classification problems using a single output unit with a sigmoid activation; this is another way to address them.)

Let's characterize the region in  $x$ -space where this network's output indicates the first class (that is,  $a_1^2$  is larger) or indicates the second class (that is,  $a_2^2$  is larger). Your diagram from the previous part will be useful here.

What is the output value of the neural network in each of the following cases? Write your answer for  $a_i^2$  as expressions, you can use powers of  $e$ , for example,  $e^{*2} + 1$ ; the exponents can be negative,  $e^{*(-2)} + 1$ .

Case 1) For  $f(z_1^1) + f(z_2^1) + f(z_3^1) + f(z_4^1) = 0$

**3.3.A)**

$$a_1^2 = \frac{e^{0}}{e^{0} + e^{2}}$$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.3.B)**

$$a_2^2 = \frac{e^{2}}{e^{0} + e^{2}}$$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.3.C)**

Which class is predicted? Class 2 ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Case 2) For  $f(z_1^1) + f(z_2^1) + f(z_3^1) + f(z_4^1) = 1$

**3.3.D)**

$$a_1^2 = \frac{1}{2}$$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.3.E)**

$$a_2^2 = \frac{1}{2}$$

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

**3.3.F)**

Which class is predicted? Boundary ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Case 3) For  $f(z_1^1) + f(z_2^1) + f(z_3^1) + f(z_4^1) = 3$

**3.3.G)**

$$a_1^2 = \text{e**3/(e**3+e**-1)}$$

[Check Syntax](#)[Submit](#)[View Answer](#)[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

**3.3.H)**

$$a_2^2 = \text{e**-1/(e**3+e**-1)}$$

[Check Syntax](#)[Submit](#)[View Answer](#)[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

**3.3.I)**

Which class is predicted? [Class 1](#) ▼

[Submit](#)[View Answer](#)[Ask for Help](#)

100.00%

*You have infinitely many submissions remaining.*

A code and data folder that will be useful for doing this lab can be found [here](#). Download this to your computer, or alternatively, use the [colab notebook](#).

This homework continues the exploration and implementation of [neural networks](#) as discussed in the notes.

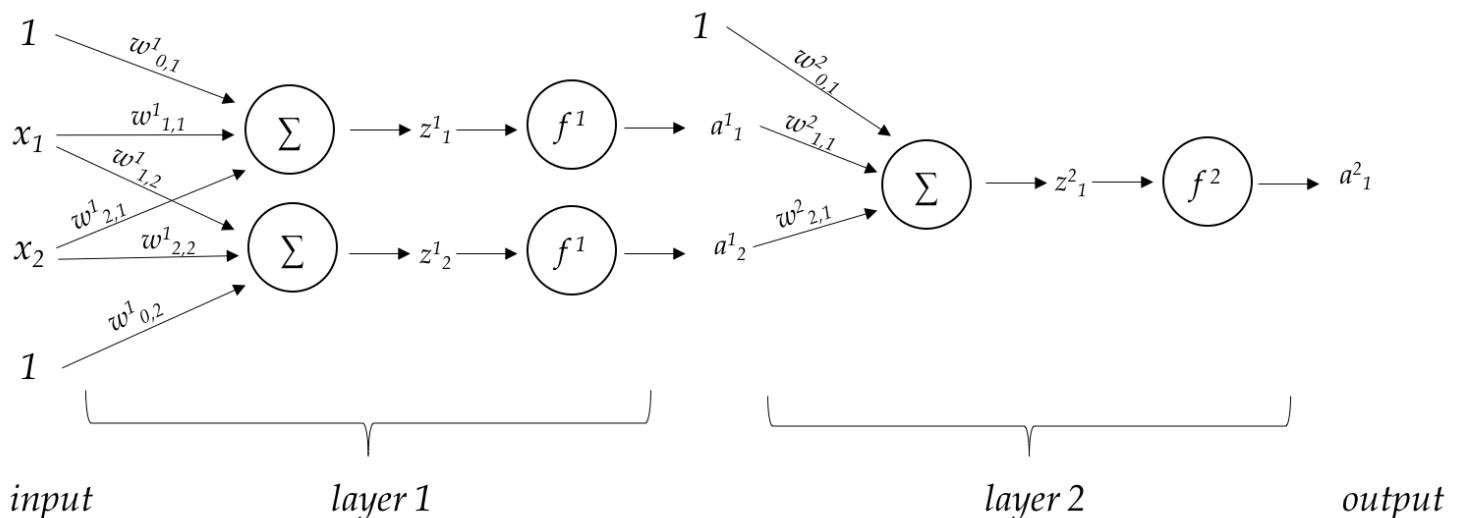
In particular, this homework considers neural networks with multiple layers. Each layer has multiple inputs and outputs, and can be broken down into two parts:

- A **linear** module that implements a linear transformation:  $z_j = (\sum_{i=1}^m x_i W_{i,j}) + W_{0,j}$  specified by a weight matrix  $W$  and a bias vector  $W_0$ . The output is  $[z_1, \dots, z_n]^T$ .
- An **activation** module that applies an activation function to the outputs of the linear module for some activation function  $f$ , such as Tanh or ReLU in the hidden layers or Softmax (see below) at the output layer. We write the output as:  $[f(z_1), \dots, f(z_m)]^T$ , although technically, for some activation functions such as softmax, each output will depend on all the  $z_i$ , not just one.

We will use the following notation for quantities in a network:

- Inputs to the network are  $x_1, \dots, x_d$ .
- Number of layers is  $L$
- There are  $m^l$  inputs to layer  $l$
- There are  $n^l = m^{l+1}$  outputs from layer  $l$
- The weight matrix for layer  $l$  is  $W^l$ , an  $m^l \times n^l$  matrix, and the bias vector (offset) is  $W_0^l$ , an  $n^l \times 1$  vector
- The outputs of the linear module for layer  $l$  are known as **pre-activation** values and denoted  $z^l$
- The activation function at layer  $l$  is  $f^l(\cdot)$
- Layer  $l$  activations are  $a^l = [f^l(z_1^l), \dots, f^l(z_{n^l}^l)]^T$
- The output of the network is the values  $a^L = [f^L(z_1^L), \dots, f^L(z_{n^L}^L)]^T$
- Loss function  $Loss(a, y)$  measures the loss of output values  $a$  when the target is  $y$

Here is an illustrative picture:



## 1) Backpropagation

The materials for week 6 and week 7 will be helpful here, including the [week6 lecture](#) and [week7 lecture](#).

We have seen in the [lecture notes](#) how to train multi-layer neural networks as classifiers using stochastic gradient descent (SGD). One of the key steps in the SGD method is the evaluation of the gradient of the loss function with respect to the model parameters. In this problem, you will derive the backpropagation method for a general  $L$ -layer neural network. We'll exploit the decomposition of the network into *linear* and *activation* modules that we introduced at the start of this homework. Remember that we've defined the shapes of the various quantities at the start of the homework.

- Each linear module has a `forward` method that takes in a column vector of activations  $A$  (from the previous layer) and returns a column vector  $Z$  of pre-activations; it can also store its input or output vectors for use by other methods (e.g., for subsequent backpropagation).
- Each activation module has a `forward` method that takes in a column vector of pre-activations  $Z$  and returns a column vector  $A$  of activations; it can also store its input or output vectors for use by other methods (e.g., for subsequent backpropagation).
- Each linear module has a `backward` method that takes in a column vector  $\frac{\partial Loss}{\partial Z}$  and returns a column vector  $\frac{\partial Loss}{\partial A}$ . This module also computes and stores  $\frac{\partial Loss}{\partial W}$  and  $\frac{\partial Loss}{\partial W_0}$ , the gradients with respect to the weights.
- Each activation module has a `backward` method that takes in a column vector  $\frac{\partial Loss}{\partial A}$  and returns a column vector  $\frac{\partial Loss}{\partial Z}$ .

The backpropagation algorithm will consist of:

- Calling the `forward` method of each module in turn, feeding the output of one module as the input to the next; starting with the input values of the network. After this pass, we have a predicted value for the final network output.
- Calling the `backward` method of each module in reverse order, using the returned value from one module as the input value of the previous one. The starting value for the backward method is  $\partial Loss(a^L, y) / \partial a^L$ , where  $a^L$  is the activation of the final layer (computed during the forward pass) and  $y$  is the desired output (the label).

## 1.1) Linear Module

The `forward` method, given  $A$  from the previous layer, implements:

$$Z = W^T A + W_0$$

and stores the input  $A$  to be used by the `backward` method.

Recall that there are  $n^l = m^{l+1}$  outputs from layer  $l$ . For layer  $l$ ,  $W$  is a  $m^l \times n^l$  matrix,  $W_0$  is a  $n^l \times 1$  vector, and  $A$  from the previous layer is a  $n^{l-1} \times 1$  (or  $m^l \times 1$ ) vector. Given these shapes, make sure that you understand why the forward equation has  $W^T$  and not  $W$ .

The following questions ask for a matrix expression involving any of  $A$ ,  $Z$ ,  $dLdA$ ,  $dLdZ$ ,  $W$  and  $W_0$ .

**Enter your answers as Python expressions. You can use `transpose(x)` for transpose of an array, and `x@y` to indicate a matrix product of two arrays. Remember that `x*y` denotes component-wise multiplication.**

The `backward` method, given  $dLdZ = \partial Loss / \partial Z$  (an  $n^l \times 1$  vector), returns  $dLdA = \partial Loss / \partial A$  (an  $m^l \times 1$  vector):

$$\frac{\partial Loss}{\partial A} = \frac{\partial Z}{\partial A} \frac{\partial Loss}{\partial Z}$$

### 1.1.A)

dLdA= W@dLdZ

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

The backward method, given  $\mathbf{dLdZ} = \partial \text{Loss} / \partial \mathbf{Z}$ , also computes  $\mathbf{dLdW}$  (an  $m^l \times n^l$  matrix) and  $\mathbf{dLdW0}$  (an  $n^l \times 1$  vector), and stores them in the module instance.

$$\mathbf{dLdW} = \frac{\partial \text{Loss}}{\partial \mathbf{W}} = \frac{\partial \mathbf{Z}}{\partial \mathbf{W}} \frac{\partial \text{Loss}}{\partial \mathbf{Z}}$$

**1.1.B)**

dLdW= A@transpose(dLdZ)

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

and

$$\mathbf{dLdW0} = \frac{\partial \text{Loss}}{\partial \mathbf{W}_0} = \frac{\partial \text{Loss}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{W}_0}$$

**1.1.C)**

dLdW0= dLdZ

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

We will use  $\mathbf{dLdW}$  and  $\mathbf{dLdW0}$  as the gradient values in SGD.

**1.2) Activation Module**

Activation modules don't have any weights and so they are simpler.

The forward method for functions like *tanh* or *sigmoid*, given  $\mathbf{Z}$ , return the function on the vector, componentwise. *Softmax* operates on the whole vector, as described earlier, and will need some special treatment.

The backward method, given  $\mathbf{dLdA} = \partial \text{Loss} / \partial \mathbf{A}$ , returns:

$$\mathbf{dLdZ} = \frac{\partial \text{Loss}}{\partial \mathbf{Z}} = \frac{\partial \text{Loss}}{\partial \mathbf{A}} \frac{\partial \mathbf{A}}{\partial \mathbf{Z}}$$

In this case,  $m^l = n^l$  and the quantities are column vectors of that size.

For  $\text{Softmax} = \text{SM}(\mathbf{Z})$  at the output layer and assuming that we are using *NLL* as the  $\text{Loss}(\mathbf{A}, \mathbf{Y})$  function, we have seen that there is a simple form for  $\mathbf{dLdZ} = \frac{\partial \text{Loss}}{\partial \mathbf{Z}}$ ; namely, it is the prediction error  $\mathbf{A} - \mathbf{Y}$ . A similar result holds when using *NLL*

with a sigmoid output activation or a quadratic loss with a linear output activation. Note that for hinge loss with a linear activation, the form of  $dLdz$  is different (see the lecture notes on the hinge loss).

## 2) Implementing Neural Networks

**Please watch the lecture videos for this week before attempting this problem.** Additionally, please review the [SGD notes](#).

Although for "real" applications you want to use one of the many packaged implementations of neural networks (we'll start using one of those soon), there is no substitute for implementing one yourself to get an in-depth understanding. Luckily, that is relatively easy to do if we're not too concerned with maximum efficiency.

We'll use the modular implementation that we guided you through in the previous problem, which leads to clean code. The basic framework for SGD training is given below. We can construct a network and train it as follows:

```
# build a 3-layer network
net = Sequential([Linear(2,3), Tanh(),
                  Linear(3,3), Tanh(),
                  Linear(3,2), SoftMax()])
# train the network on data and labels
net.sgd(X, Y)
```

You will need to fill in the missing code. We encourage you to test in your own Python environment and then paste your answer and verify the results. The test cases are provided in the code distribution linked at the top of the page. **The code distribution includes additional test methods that will test each of the methods in turn, so you can debug incrementally.** Below are some hints for some of the methods:

- `Sequential.sgd`: Implement SGD. Randomly pick a data point  $X_t, Y_t$  by using `np.random.randint` to choose a random index into the data. Compute the predicted output  $Y_{pred}$  for  $X_t$  with the forward method. Compute the loss for  $Y_{pred}$  relative to  $Y_t$ . Use the backward method to compute the gradients. Use the `sgd_step` method to change the weights. Repeat.
- `SoftMax.class_fun`: Given the column vector of class probabilities for each point (computed by Softmax), this returns a vector of the classes (integers) with the highest probability for each point.
- We will (later) be generalizing SGD to operate on a "mini-batch" of data points instead of a single point. You should strive for an implementation of the forward, backward, and `class_fun` methods that works with batches of data. Whenever  $b$  is mentioned as part of the shape of a matrix in the code, this  $b$  refers to the number of points.
- **A note on debugging.** We have provided you with a file `code_for_hw7.py` (as well as a colab) that has a copy of the template below and a detailed set of outputs to check your implementation. **Trying to debug directly on MITx will not be a good experience; intermediate tests for each method are available ONLY in the code file/colab.**



```

1 class Module:
2     def sgd_step(self, lrate): pass # For modules w/o weights
3
4
5 # Linear modules
6 #
7 # Each linear module has a forward method that takes in a batch of
8 # activations A (from the previous layer) and returns
9 # a batch of pre-activations Z.
10 #
11 # Each linear module has a backward method that takes in dLdZ and
12 # returns dLdA. This module also computes and stores dLdW and dLdW0,
13 # the gradients with respect to the weights.
14 class Linear(Module):
15     def __init__(self, m, n):
16         self.m, self.n = (m, n) # (in size, out size)
17         self.W0 = np.zeros([self.n, 1]) # (n x 1)
18         self.W = np.random.normal(0, 1.0 * m ** (-.5), [m, n]) # (m x n)
19
20     def forward(self, A):
21         self.A = A # (m x b) Hint: make sure you understand what b stands for #da
22         return self.W.T@self.A + self.W0 # Your code (n x b)
23
24     def backward(self, dLdZ): # dLdZ is (n x b), uses stored self.A
25         self.dLdW = self.A @ dLdZ.T # Your code
26         self.dLdW0 = np.sum(dLdZ, axis = 1, keepdims=True) # Your code
27         return self.W @ dLdZ # Your code: return dLdA (m x b)
28
29     def sgd_step(self, lrate): # Gradient descent step
30         self.W = self.W - lrate * self.dLdW # Your code
31         self.W0 = self.W0 - lrate * self.dLdW0 # Your code
32
33
34 # Activation modules
35 #
36 # Each activation module has a forward method that takes in a batch of
37 # pre-activations Z and returns a batch of activations A.
38 #
39 # Each activation module has a backward method that takes in dLdA and
40 # returns dLdZ, with the exception of SoftMax, where we assume dLdZ is
41 # passed in.
42 class Tanh(Module): # Layer activation
43     def forward(self, Z):
44         self.A = np.tanh(Z)
45         return self.A
46
47     def backward(self, dLdA): # Uses stored self.A
48         # print(f'dLdA : {dLdA}')
49         return dLdA * (1.0 - (self.A ** 2))
50
51
52 class ReLU(Module): # Layer activation
53     def forward(self, Z):
54         self.A = np.maximum(Z, np.zeros(Z.shape)) # Your code: (?, b)
55         return self.A
56
57     def backward(self, dLdA): # uses stored self.A
58         return dLdA * (self.A != 0) # Your code: return dLdZ (?, b)
59
60
61 class SoftMax(Module): # Output activation
62     def forward(self, Z):
63         z_exp = np.exp(Z)

```

```
64     sum = np.sum(z_exp, axis=0)
65     return z_exp/sum
66
67     def backward(self, dLdZ): # Assume that dLdZ is passed in
68         return dLdZ
69
70     def class_fun(self, Ypred): # Return class indices
71         return np.argmax(Ypred, axis=0) # Your code: (1, b)
72
73
74 # Loss modules
75 #
76
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

*You have infinitely many submissions remaining.*