HW3: Principal Component Analysis

Due Oct 5, 2021 by 10:59am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Sep 27, 2021 at 8pm - Oct 5, 2021 at 10:59am

This assignment was locked Oct 5, 2021 at 10:59am.

Edit Sep 30: added an example for "Find Covariance Matrix".

Assignment Goals

- Explore Principal Component Analysis (PCA) and the related Python packages
- Make pretty pictures :)

Summary

In this project, you'll be implementing a facial analysis program using Principal Component Analysis (PCA), using the skills you learned from the **linear algebra + PCA** lecture. You'll also continue to build your Python skills.

We'll walk you through the process step-by-step (at a high level).

Packages Needed for this Project

In this project, you'll need the following packages:

```
>>> from scipy.linalg import eigh
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

Dataset

You will be using part of Yale face dataset (processed). You can download the dataset here: Yale8_32x32.npy

The dataset contains *2414* sample images, each of size *32x32*. We will use *n* to refer to number of images, **n=2414** and *d* to refer to number of features for each sample image, **d=1024** (32x32). We will test your code only using the provided data set. Note, we'll use x_i to refer to the ith sample image which would be a d-dimensional feature vector.

Program Specification

Implement these **six** Python functions to do PCA on our provided dataset, in a file called pca.py :

1. **load_and_center_dataset(filename)** — load the dataset from a provided .npy file, re-center it around the origin and **return** it as a NumPy array of floats
2. **get_covariance(dataset)** — calculate and **return** the covariance matrix of the dataset as a NumPy matrix (d x d array)
3. **get_eig(S, m)** — perform eigen decomposition on the covariance matrix S and **return** a diagonal matrix (NumPy array) with the largest m eigenvalues on the diagonal, and a matrix (NumPy array) with the corresponding eigenvectors as columns
4. **get_eig_perc(S, perc)** — similar to get_eig, but instead of returning the first m, return all eigenvalues and corresponding eigenvectors in similar format as get_eig that explain more than *perc* % of variance
5. **project_image(image, U)** — project each image into your m-dimensional space and **return** the new representation as a one-dimensional NumPy array with *d* elements
6. **display_image(orig, proj)** — use matplotlib to display a visual representation of the original image and the projected image side-by-side

Load and Center the Dataset

First, if you haven't, download our sample dataset to the machine you're working on: Yale8_32x32.npy .

Once you have it, you'll want to use the numpy function `load()` to load the file into Python. (You may need to install NumPy first.)

```
>>> x = np.load(filename)
```

This should give you an *n x d* dataset (*n*: the number of images in the dataset and *d*: the dimensions of each image)

Each row represents an image feature vector. For this particular dataset, we have *n* = **2414** (no. of images) and *d*=32 x 32=**1024** (32 by 32 pixels).

Your next step is to center this dataset around the origin. Recall the purpose of this step from lecture---it is a technical condition that makes it easier to perform PCA---but does not lose any information.

To center the dataset is simply to subtract the mean μ_x from each data point x_i (image in our case), i.e. $x_i^{cent} = x_i - \mu_x$ where mean $\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i$ (n is the no. of images)

You can take advantage of the fact that `x` (as defined above) is a NumPy array and as such, has this convenient behavior:

```
>>> x = np.array([[1,2,5],[3,4,7]])
>>> np.mean(x, axis=0)
=> array([2., 3., 6.])
>>> x - np.mean(x, axis=0)
=> array([[-1., -1., -1.],
          [ 1.,  1.,  1.]])
```

After you've implemented this function, it should work like this:

```
>>> X = load_and_center_dataset("Yale8_32x32.npy")
>>> len(X)
=> 2414
>>> len(X[0])
=> 1024
>>> np.average(X)
=> -8.315174931741023e-17
```

(Its center isn't *exactly* zero, but taking into account precision errors over 2414 arrays of 1024 floats, it's what we call Close Enough.)

Note, From now on, we will use x_i to refer to x_i^{cent}.

Find Covariance Matrix

Recall, from lecture, that one of the interpretations of PCA is that it is the eigendecomposition of the sample covariance matrix. We will rely on this interpretation in this assignment, with all of the information you need below.

The covariance matrix is defined as

$$S = \frac{1}{n-1}\sum_{i=1}^{n} x_i x_i^T$$

Note that x_i is one of the *n* images in the dataset (centered) and it is considered as a column vector (size **d x 1**) in this formula.

If you consider it as a row vector (size **1 x d**), then the formula would like below.

$$S = \frac{1}{n-1}\sum_{i=1}^{n} x_i^T x_i$$

To calculate this, you'll need a couple of tools from NumPy again:

```
>>> x = np.array([[1,2,5],[3,4,7]])
>>> np.transpose(x)
=> array([[1, 3],
          [2, 4],
          [5, 7]])
>>> np.dot(x, np.transpose(x))
=> array([[30, 46],
          [46, 74]])
>>> np.dot(np.transpose(x), x)
=> array([[10, 14, 26],
          [14, 20, 38],
          [26, 38, 74]])
```

The result of this function for our sample dataset should be a **d x d** (1024 x 1024) matrix. An example:

```
>>> X = np.array([[1,2,3], [2,3,4]])
>>> print(get_covariance(X))
[[ 5.  8. 11.]
 [ 8. 13. 18.]
 [11. 18. 25.]]
```

Get m Largest Eigenvalues/Eigenvectors

Again, recall from lecture that eigenvalues and eigenvectors are useful objects that characterize matrices. Better yet, PCA can be performed by doing an eigendecomposition and taking the eigenvectors corresponding to the largest eigenvalues. This replaces the recursive deflation step we discussed in class.

You'll never have to do eigen decomposition by hand again ! Use `scipy.linalg.eigh()` to help, particularly the optional `eigvals` argument.

We want the *largest m eigenvalues* of S.

Return the eigenvalues as a diagonal matrix, in descending order, and the corresponding eigenvectors as columns in a matrix.

To return more than one thing from a function in Python:

```
def multi_return():
    return "a string", 5
mystring, myint = multi_return()
```

Make sure to return the diagonal matrix of eigenvalues FIRST, then the eigenvectors in corresponding columns. You may have to rearrange the output of eigh() to get the eigenvalues in decreasing order and *make sure to keep the eigenvectors in the corresponding columns* after that rearrangement.

```
>>> Lambda, U = get_eig(S, 2)
>>> print(Lambda)
[[1369142.41612494          0.        ]
 [      0.        1341168.50476773]]
>>> print(U)
[[-0.01304065 -0.04342441  0.0129248 ]
 [-0.01177219 -0.04342345 -0.00978305]
 [-0.00905278 -0.04095089 -0.00601955]
 ...
 [ 0.00146631  0.03622013  0.0465019 ]
 [ 0.00329216  0.0348809  -0.04643187]
 [ 0.00305951  0.03330786]]
>>> Lambda, U = get_eig(S, 0.07)
>>> print(Lambda)
[[1369142.41612494          0.        ]
 [      0.        1341168.50476773]]
>>> print(U)
[[-0.01304065 -0.04342441]
 [-0.01177219 -0.04342345]
 [-0.00905278 -0.04095089]
 ...
 [ 0.00146631  0.03622013]
 [ 0.00329216  0.0348809 ]
 [ 0.00305951  0.03330786]]
```

Get all Eigenvalues/Eigenvectors that Explain More than Certain % of Variance

We want all the *eigenvalues* that explain more than a certain percentage of variance.

Let λ_i be an eigenvalue of the covariance matrix **S**. Then the percentage of variance explained is calculated as:

$$\frac{\lambda_i}{\sum_{i=1}^{n}\lambda}$$

Return the eigenvalues as a diagonal matrix, in descending order, and the corresponding eigenvectors as columns in a matrix.

Make sure to return the diagonal matrix of eigenvalues FIRST, then the eigenvectors in corresponding columns. You may have to rearrange the output of eigh() to get the eigenvalues in decreasing order and *make sure to keep the eigenvectors in the corresponding columns* after that rearrangement.

```
>>> Lambda, U = get_eig_perc(S, 0.04)
>>> print(Lambda)
[[1369142.41612494          0.        ]
 [      0.        1341168.50476773]]
>>> print(U)
[[-0.01304065 -0.04342441]
 [-0.01177219 -0.04342345]
 [-0.00905278 -0.04095089]
 ...
 [ 0.00146631  0.03622013]
 [ 0.00329216  0.0348809 ]
 [ 0.00305951  0.03330786]]
>>> Lambda, U = get_eig_perc(S, 0.07)
>>> print(Lambda)
[[1369142.41612494          0.        ]
 [      0.        1341168.50476773]]
>>> print(U)
[[-0.01304065 -0.04342441]
 [-0.01177219 -0.04342345]
 [-0.00905278 -0.04095089]
 ...
 [ 0.00146631  0.03622013]
 [ 0.00329216  0.0348809 ]
 [ 0.00305951  0.03330786]]
```

Project the Images

Given one of the images from your dataset and the results of your `get_eig()` function, create and return the *projection* of that image.

For any image x_i, we project it into the *m* dimensional space as

$$x_i^{proj} = \sum_{j=1}^{m} \alpha_{ij} u_j \text{ where } \alpha_{ij} = u_j^T x_i \text{ , } \alpha_i \in \mathbb{R}^m \text{ and } x_i^{proj} \in \mathbb{R}^d.$$

u_j is an eigenvector column (size **d x 1**) of U from the previous function.

Find the alphas for your image, then use them together with the eigenvectors to create your projection.

```
>>> projection = project_image(X[0], U)
>>> print(projection)
[6.8412225 4.83981287 1.41736694 ... 8.75796534 7.45916035 5.4548656 ]
>>> projection = project_image(X[1], U)
>>> print(projection)
[ 6.45209766 4.3081919  0.69973235 ... 10.08336124  8.6668639
 6.4944833 ]
```

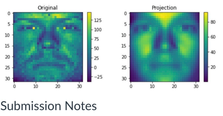
Visualize

We'll be using matplotlib's imshow , First, make sure you have the matplotlib library , for creating figures.

Follow these steps to visualize your images:

1. Reshape the images to be 32x32 (you should have calculated them as 1d vectors of 1024 numbers).
2. Create a figure with one row of two subplots .
3. The first subplot (on the left) should be titled ". "Original", and the second (on the right) should be titled "Projection".
4. Use `imshow()` with optional argument `aspect='equal'` to render the original image in the first subplot and the projection in the second subplot.
5. Use the return value of `imshow()` to create a colorbar , for each image.
6. Render , your plots!

```
>>> X = load_and_center_dataset("Yale8_32x32.npy")
>>> S = get_covariance(X)
>>> Lambda, U = get_eig(S, 2)
>>> projection = project_image(X[0], U)
>>> display_image(X[0], projection)
```



Submission Notes

Please submit your files in a zip file named **hw3_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **pca.py**. Do not submit a Jupyter notebook .ipynb file.

Be sure to **remove all debugging output** before submission; your functions should run silently (except for the image rendering window). Failure to remove debugging output will be **penalized (10pts)**.

Precision: For grading, the floating-point output needs to match with the correct answer up to 6 decimal places, i.e., the difference is smaller than 10^-6.

Regrading: Any regrading request needs to be submitted within 72 hours from the grade release. The regrading requests should be about more than 10 point deduction due to output format mismatch that can be fixed by modifying one or two lines in the code, and 10 points will be deducted for such output format mismatch. Those requests due to other mistakes (e.g., logic flow of the code) are not accepted.

This assignment is due on **10/05/2021 10:59am**. It is preferred to first submit a version well before the deadline and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

PCA Rubric

Criteria	Ratings		Pts
load_and_center_dataset() returns correct output	20 pts Full Marks	0 pts No Marks	20 pts
get_covariance() returns correct result for the provided dataset	15 pts Full Marks	0 pts No Marks	15 pts
get_eig() & get_eig_perc() return correct eigenvalue and eigenvector matrix	25 pts Full Marks	0 pts No Marks	25 pts
project_image() correctly projects image vector onto eigenvectors	15 pts Full Marks	0 pts No Marks	15 pts
display_image() creates correctly formatted plot (titles/subplots/colorbars appear as specified)	10 pts Full Marks	0 pts No Marks	10 pts
display_image() displays images correctly for given input	15 pts Full Marks	0 pts No Marks	15 pts
		Total Points:	100

HW4: Clustering

Due Oct 12, 2021 by 10:59am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Oct 4, 2021 at 11:59pm - Oct 12, 2021 at 11:15am

This assignment was locked Oct 12, 2021 at 11:15am.

Assignment Goals

- Process real-world data
- Implement hierarchical clustering
- Randomly generate your own data
- Imshow your linkage process

Summary

In this assignment, you'll be performing clustering using the publicly available Pokemon stats. Each Pokemon is defined by a row in the data set. Because there are various ways to characterize how strong a Pokemon is, it is often desirable to convert a raw stats sheet into a shorter feature vector. For this assignment, you will represent a Pokemon's strength by two numbers: "x" and "y". "x" will represent the Pokemon's total offensive strength, which is defined by Attack + Sp. Atk + Speed. Similarly, "y" will represent the Pokemon's total defensive strength, which is defined by Defense + Sp. Def + HP. After each Pokemon becomes that two-dimensional feature vector, you will cluster the first 20 Pokemon with hierarchical agglomerative clustering (HAC).

Packages Needed for this Project

In this project, you'll need the following packages:

```
import csv
import numpy as np
import matplotlib.pyplot as plt
```

Program Specification

Download the data in CSV format: Pokemon.csv

Write the following Python functions:

1. **load_data(filepath)** — takes in a string with a path to a CSV file formatted as in the link above, and returns the first 20 data points (**without** the Generation and Legendary columns but retaining all other columns) in a single structure.
2. **calculate_x_y(stats)** — takes in one row from the data loaded from the previous function, calculates the corresponding x, y values for that Pokemon as specified above, and returns them in a single structure.
3. **hac(dataset)** — performs single linkage hierarchical agglomerative clustering on the Pokemon with the (x,y) feature representation, and returns a data structure representing the clustering.
4. **random_x_y(m)** — takes in the number of samples we want to randomly generate, and returns these samples in a single structure.
5. **imshow_hac(dataset)** — performs single linkage hierarchical agglomerative clustering on the Pokemon with the (x,y) feature representation, and imshow the clustering process.

You may implement other helper functions as necessary, but these are the functions we will be testing.

Load Data

Read in the file specified in the argument (the DictReader from Python's csv module, will be of use) and return a list of dictionaries, where each row in the dataset is a dictionary with the column headers as keys and the row elements as values. These dictionaries should **not** include the Generation and Legendary columns, as we will not be using them in this program. The integer strength of Pokemons (6 columns) and the 2 other columns (#, Total) should be converted to integers. Also, you only should have the **first 20** Pokemon in this structure. Note the first 20 refers to row 1 through row 20 (From Bulbasaur to BeedrillMega Beedrill excluding the headers).

You may assume the file exists and is a properly formatted CSV.

```
>>> pokemons = load_data("Pokemon.csv")
>>> len(pokemons)
20
>>> pokemons
[{'#': 1, 'Name': 'Bulbasaur', 'Type 1': 'Grass', 'Type 2': 'Poison', 'Total': 318, 'HP': 45, 'Attack': 49, 'Defense': 49, 'Sp. Atk': 65, 'Sp. Def': 65, 'Speed': 45},
..., {'#': 15, 'Name': 'BeedrillMega Beedrill', 'Type 1': 'Bug', 'Type 2': 'Poison', 'Total': 495, 'HP': 65, 'Attack': 150, 'Defense': 40, 'Sp. Atk': 15, 'Sp. Def': 80,
'Speed': 145}]
```

Calculate Feature Values

This function takes in the data from a single row of the raw dataset as read in the previous function (i.e. a single dictionary, **without** the Generation and Legendary values but retaining all other columns). This function should return the x, y values in a tuple, formatted as (x, y).

```
>>> pokemons[1]
{'#': 2, 'Name': 'Ivysaur', 'Type 1': 'Grass', 'Type 2': 'Poison', 'Total': 405, 'HP': 60, 'Attack': 62, 'Defense': 63, 'Sp. Atk': 80, 'Sp. Def': 80, 'Speed': 60}
>>> calculate_x_y(pokemons[1])
(202, 203)
```

Perform HAC

For this function, we would like you to mimic the behavior of SciPy's HAC function, linkage(). You may not use this function in your implementation, but we strongly recommend using it to verify your results!

Input: A collection of *m* observation vectors in *n* dimensions may be passed as an *m* by *n* array (for us, this will be a list of tuples, not a numpy array like for linkage()!). All elements of the condensed distance matrix must be finite, i.e. no NaNs or infs. In our case, *m* could be the number of Pokemon (20) and *n* is 2 (the x and y features for each Pokemon). You may hardcode n = 2, but do not hardcode m = 20! You need to check for invalid inputs. If there is any, pop them out.

Using **single linkage**, perform the hierarchical agglomerative clustering algorithm as detailed on slides. Use a standard Euclidean distance function for calculating the distance between two points.

Output: An (*m*-1) by 4 matrix Z. At the *i*-th iteration (i starts from 0), clusters with indices Z[i, 0] and Z[i, 1] are combined to form cluster ***m + i***. A cluster with an index less than *m* corresponds to one of the *m* original observations. The distance between clusters Z[i, 0] and Z[i, 1] is given by Z[i, 2]. The fourth value Z[i, 3] represents the number of original observations in the newly formed cluster.

That is:

- Number each of your starting data points from 0 to *m*-1. These are their original cluster numbers.
- Create an (*m*-1) x 4 array or list. Iterate through the list row by row.
- For each row of the output matrix Z, determine which two clusters you will merge and put their numbers into the first and second elements of the row. The first point listed should be the smaller of the two cluster indexes. The single-linkage distance between the two clusters goes into the third element of the row. The total number of points in the cluster goes into the fourth element.
- Before returning the data structure, convert it into a NumPy matrix.

If you follow these guidelines for input and output, your result should match the result of scipy.cluster.hierarchy.linkage() and you can use that function to verify your results.

Tie-Breaking

In the event that there are multiple pairs of points with equal distance for the next cluster:

Given a set of pairs with equal distance {(xi, xj)} where i < j, we prefer the pair with the smallest first cluster index *i*. If there are still ties (xi, xj), ... (xi, xk) where *i* is that smallest first index, we prefer the pair with the smallest second cluster index.

Be aware that this tie-breaking strategy may not always produce identical results to scipy.cluster.hierarchy.linkage().

```
>>> pokemons_x_y = []
>>> for row in pokemons:
...     pokemons_x_y.append(calculate_x_y(row))
...

>>> hac(pokemons_x_y)

matrix([[ 7.        ,  8.        ,  4.24264069,  2.        ],
        [15.        , 18.        ,  7.07106781,  2.        ],
        [ 1.        , 21.        , 13.34166406,  3.        ],
        [13.        , 16.        , 14.14213562,  2.        ],
        [14.        , 17.        , 14.14213562,  2.        ],
        [ 5.        , 22.        , 14.56021978,  4.        ],
        [ 3.        , 12.        , 16.64331698,  2.        ],
        [23.        , 24.        , 22.36067977,  4.        ],
        [10.        , 25.        , 22.627417  ,  5.        ],
        [ 0.        ,  9.        , 22.8035085 ,  2.        ],
        [ 2.        , 11.        , 26.40075756,  2.        ],
        [ 4.        , 29.        , 32.44996148,  3.        ],
        [ 6.        , 30.        , 38.01315562,  3.        ],
        [28.        , 31.        , 55.47071299,  8.        ],
        [19.        , 32.        , 58.52349955,  4.        ],
        [20.        , 26.        , 59.46427499,  4.        ],
        [34.        , 35.        , 68.24221567,  8.        ],
        [27.        , 33.        , 77.1038261 , 12.        ],
        [36.        , 37.        , 84.85281374, 20.        ]])
```

For example, in row 3, column 2 of the output, 21 is the cluster index that represents the cluster formed at the 2nd iteration (i = 1).

Generate Random Data

Given a positive integer m, your aim is to uniformly randomly generate m Pokemons' "x" and "y", which satisfy 0<x<360, 0<y<360, and where x,y are both integers. The output of this **random_x_y(m)** function will further be used as the input to **hac(dataset).**

```
>>> random_x_y(20)
[[350, 144], [187, 76], [3, 149], [319, 281], [305, 249], [179, 213], [199, 47], [45, 80], [273, 242], [16, 89], [172, 337], [292, 353], [328, 2], [297, 171], [163,
197], [230, 211], [70, 343], [311, 250], [349, 282], [133, 44]]
```

Note: your output does not need to match the given output, since the data are randomly generated, but be aware of the bound and the distribution from which these data are generated.

Imshow HAC Process

You need to imshow the HAC process by including the start status (m points) and the results of (m-1) linkage processes. You should **only have one** plt.figure(), so in total you will need to plot m times on the **same** figure. Pause 0.1 seconds after each plotting, and do not close or clean the figure at the end of your function. An example of the start and the end status are shown as below. (**Hint**: consider using plt.scatter(), plt.plot(), plt.pause(); you can reuse/revise the code of **hac(dataset)** by embedding the calls to appropriate plotting functions into the HAC process code.)

A useful intro to plt.pause().

Submission Notes

Please submit your files in a zip file named **hw4_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **pokemon_stats.py**. Do not submit a Jupyter notebook .ipynb file.

All code should be contained in functions or under a

```
if __name__=="__main__":
```

check so that it will not run if your code is imported to another program.

Be sure to **remove all debugging output** before submission. Failure to remove debugging output will be **penalized (10pts)**.

Cheating results in -100 pts and further punishment.

Note: From homework 4 and on, we will deduct 5 points if the submitted zip file contains subfolder when the grader unzips the zip file

This assignment is due on 10/12/2021 at 10:59 am. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

Some Rubric

Criteria	Ratings		Pts
load_data(filepath)	20 pts Full Marks	0 pts No Marks	20 pts
calculate_x_y(stats)	20 pts Full Marks	0 pts No Marks	20 pts
hac(dataset)	20 pts Full Marks	0 pts No Marks	20 pts
random_x_y(m)	20 pts Full Marks	0 pts No Marks	20 pts
imshow_hac(dataset)	20 pts Full Marks	0 pts No Marks	20 pts
			Total Points: 100

HW5: Linear Regression

Due Oct 19, 2021 by 10:59am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Oct 11, 2021 at 11:59pm - Oct 19, 2021 at 11:15am

This assignment was locked Oct 19, 2021 at 11:15am.

Update (Oct 13th): It is fine to have small differences from the outputs in the writeup. We will be checking out to 6 digits.

Assignment Goals

- Implement a linear regression calculation
- Examine the trends in real (messy) data

Summary

Percentage of body fat, age, weight, height, and ten body circumference measurements (e.g., abdomen) are recorded for 252 people. Fitting body fat to the other measurements using regression provides a convenient way of estimating body fat using only a scale and a measuring tape. In this assignment, you will be looking at the bodyfat dataset ↗, and build several models on top of it.

Program Specification

You will be using the bodyfat dataset (bodyfat.csv ↓) for this assignment. Complete the following Python functions in this template regression.py ↓ :

1. **get_dataset(filename)** -- takes a filename and **returns** the data as described below in a n-by-(m+1) array
2. **print_stats(dataset, col)** -- takes the dataset as produced by the previous function and **prints** several statistics about a column of the dataset; does not return anything
3. **regression(dataset, cols, beta)** -- calculates and **returns** the mean squared error on the dataset given fixed betas
4. **gradient_descent(dataset, cols, beta)** -- performs a single step of gradient descent on the MSE and **returns** the derivative values as a 1D array
5. **iterate_gradient(dataset, cols, beta, T, eta)** -- performs T iterations of gradient descent starting at the given betas and **prints** the results; does not return anything
6. **compute_betas(dataset, cols)** -- using the closed-form solution, calculates and **returns** the values of betas and the corresponding MSE as a tuple
7. **predict(dataset, cols, features)** -- using the closed-form solution betas, **return** the predicted body fat percentage of the give features.
8. **synthetic_datasets(betas, alphas, X, sigma)** -- generates two synthetic datasets, one using a linear model and the other using a quadratic model.
9. **plot_mse()** -- fits the synthetic datasets, and **plots** a figure depicting the MSEs under different situations.

Get Dataset

The `get_dataset()` function should return an n-by-(m+1) array of data, where n is the number of data points, and m is the number of features, plus an additional column of labels. The first column should be bodyfat percentage, which is the **target** that our regression model aims for. We denote it as **y** in the rest of the write-up. Starting from the second column, there goes a list of **features**, including density, age, weight, and more. We use index 1 to represent density, 2 to represent age, and so on... You should ignore the "IDNO" column as it merely represents the individual id of each participant.

```
>>> get_dataset("bodyfat.csv")
>>> array([[12.6, 1.0708, 23. , ..., 32. , 27.4 , 17.1 ],
           [ 6.9 , 1.0853, 22. , ..., 30.5 , 28.9 , 18.2 ],
           [24.6 , 1.0414, 22. , ..., 28.8 , 25.2 , 16.6 ],
           ...,
           [29.1 , 1.0328, 72. , ..., 31.3 , 27.2 , 18. ],
           [26.1 , 1.0309, 72. , ..., 30.5 , 29.4 , 19.8 ],
           [30.7 , 1.0275, 74. , ..., 29.7 , 30. , 20.9 ]])
>>> dataset = get_dataset("bodyfat.csv")
>>> dataset.shape
(252, 15)
```

Dataset Statistics

This is just a quick summary function on one **feature**, given in the parameter col, from the above dataset. When called, you should **print**:

1. the number of data points
2. the sample mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
3. the sample standard deviation $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$

on three lines. Please format your output to include only **TWO digits** after the decimal point. For example:

```
>>> print_stats(dataset, 1) # summary of density
```

```
252
1.06
0.02
```

You might find this guide ↗ to python print formatting useful.

Note: Please implement the formula yourself, instead of using `np.mean()` or `np.std()`.

Linear Regression

This function will perform linear regression with the model

$$f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m$$

We first define the mean squared error (MSE) as the sum of squared errors divided by # data points:

$$MSE(\beta_0, \beta_1, \cdots, \beta_m) = \frac{1}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)^2$$

The first argument refers to the dataset, and the second argument is a list of features that we wish to learn on. For example, if we would like to study the relationship of body fat vs age and weight, cols should be [2, 3]. In this case, the model should be $f(x) = \beta_0 + \beta_2 x_2 + \beta_3 x_3$. The last argument (betas) of this function represents three betas, β_0, β_2, β_3. **Return** the corresponding MSE as calculated on your dataset.

```
>>> regression(dataset, cols=[2,3], betas=[0,0,0])
>>> 418.5034902063452
>>> regression(dataset, cols=[2,3,4], betas=[0,-1.1,-2.3])
>>> 11859.174086111111
```

Gradient Descent

This function will perform gradient descent on the MSE. At the current parameter $(\beta_0, \beta_1, \ldots, \beta_m)$, the gradient is defined by the vector of partial derivatives:

$$\frac{\partial MSE(\beta_0, \beta_1, \ldots, \beta_m)}{\partial \beta_0} = \frac{2}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)$$

$$\frac{\partial MSE(\beta_0, \beta_1, \ldots, \beta_m)}{\partial \beta_1} = \frac{2}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)x_{i1}$$

$$\cdots$$

This function **returns** the corresponding gradient as a 1-D NumPy array, with the partial derivative with respect to β_0 as the first value.

```
>>> gradient_descent(dataset, cols=[2,3], betas=[0,0,0])
>>> array([-37.87698413, -1756.37222222, -7053.95138889])  # order: [partial derivative of beta_0, beta_2, beta_3]
>>> gradient_descent(dataset, cols=[1,4], betas=[0,0,0])
>>> array([-57.87698413, -59.69160405, -265.98393127 ])  # order: [partial derivative of beta_0, beta_1, beta_4]
```

Iterate Gradient

Gradient descent starts from the initial parameter $(\beta_0^{(0)}, \beta_1^{(0)}, \ldots, \beta_m^{(0)})$ and iterates the following updates at time t = 1, 2, ..., T:

$$\beta_0^{(t)} = \beta_0^{(t-1)} - \eta\frac{\partial MSE(\beta_0^{(t-1)}, \beta_1^{(t-1)}, \ldots, \beta_m^{(t-1)})}{\partial \beta_0}$$

$$\beta_1^{(t)} = \beta_1^{(t-1)} - \eta\frac{\partial MSE(\beta_0^{(t-1)}, \beta_1^{(t-1)}, \ldots, \beta_m^{(t-1)})}{\partial \beta_1}$$

and so on for the rest.

The parameters to this function are the dataset and a selection of features, the initial values for parameter betas, the T number of iterations to perform, and η (eta), the parameter for the above calculations.

Print the following for each iteration on one line, separated by spaces:

1. the current iteration number beginning at 1 and ending at T
2. the current MSE
3. the current value of beta_0
4. the current value of other betas

All floating-point values should be **rounded** to **two digits** for output.

```
>>> iterate_gradient(dataset, cols=[1,2], betas=[400,-400,300], T=10, eta=1e-4)
1 423805132.40 394.65 -405.84 -220.18  # order: T, mse, beta0, beta1, beta0
2 127353746.01 -251.73 -517.94 -301.14
3 124764241.68 101.59 -484.74 -311.03
4 67765350.84 197.75 -402.37 88.82
5 36787281.39 196.11 -404.09 -44.57
6 19957268.50 197.32 -402.82 48.47
7 10847555.87 196.43 -403.76 -34.83
8 5898470.68 197.09 -403.07 26.55
9 3210000.00 196.61 -403.57 -19.41
10 1746058.59 196.96 -403.20 14.65

>>> iterate_gradient(dataset, cols=[1,4], betas=[400,-400,10], T=5, eta=1e-4)
1 76.76 399.87 -400.14 0.71 # order: T, mse, beta0, beta1, beta4
2 5.84 399.87 -400.14 0.59
3 5.84 399.87 -400.14 0.59
4 5.84 399.87 -400.14 0.59
5 5.84 399.87 -400.14 0.59
```

Try different values for eta and a much larger T, and see how small you can make MSE (optional).

Compute Betas

Instead of using gradient descent, we can compute the closed-form solution for the parameters directly. For ordinary least-squares, this is

$$\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$$

This function returns the calculated betas (selected by cols) and their corresponding MSE in a **tuple**, as (MSE, beta_0, beta_1, and so on).

```
>>> compute_betas(dataset, cols=[1,2])
>>> (1.4829395080144443, 441.35293435552249, -400.595495368588, 0.009892284263461639)

>>> compute_betas(dataset, cols=[1,2,8,9])
>>> (1.3684583814974934, 414.36152718826951, -379.585226834994907, 0.010655149680356892, 0.042176954357878826, 0.008180045341715684)
```

Predict Body Fat

Using your closed-form betas, predict the body fat percentage for a given number of features. Return that value.

For example:

```
>>> predict(dataset, cols=[1,2], features=[1.0708, 23])
>>> 12.622458295781113
```

Synthetic Datasets

Now, let us create a few new datasets to study some characteristics of Linear Regression models. The function `synthetic_datasets(betas, alphas, X, sigma)` should return two synthetic datasets, a linear one and a quadratic one.

The linear dataset is defined as an n x 2 array, where n is the size of the input array of data points X. You may assume that X is a 2D array with shape (n,1). The second column of the linear dataset should be a copy of X (the original data points). The first column is defined as follows:

$y_i = \beta_0 + \beta_1 x_i + \epsilon_i, \ \epsilon_i \sim N(0, \sigma)$

β_0 and β_1 are provided by the argument betas. ϵ_i is an artificial error term to 'jitter' the labels so that they don't look too perfect. It is used to create a noise in the label. You should sample ϵ's from a normal distribution with mean 0 and standard deviation `sigma`, which is provided in the argument, too. You may use `numpy.random.normal()` detailed here ↗.

Similarly, the quadratic dataset is also defined as an n x 2 array. The second column remains the same as above, while the first column is defined as follows:

$y_i = \alpha_0 + \alpha_1 x_i^2 + \epsilon_i, \ \epsilon_i \sim N(0, \sigma)$

The alphas and ϵ's are from the argument `alphas`, and a normal distribution with mean 0 and standard deviation `sigma`, respectively.

The function should return a tuple, with the first being the linear dataset, followed by the quadratic dataset array.

```
>>> synthetic_datasets(np.array([0,2]), np.array([0,1]), np.array([[4]]), 1)
>> (array([[13.42085793],  4.        ]), array([[15.5334939,  4.        ]]))
```

For example, the output above contains 2 n x 2 arrays (n = 1), the second column of both arrays are the same as the input X (note X is a 2D array with shape (n, 1)).

Compare and Plot

Using the two synthetic datasets, let us study the performance of linear regression under linear data and quadratic data respectively. In `plot_mse()`, do the following:

1. Create an input array X containing 1000 numbers within range [-100, 100].
2. Create couples of betas and alphas with non-zero values.
3. Set sigmas to be $10^{-4}, 10^{-3}, \ldots, 1, 10, \ldots, 10^5$.
4. Under each setting of sigmas, generate two synthetic datasets.
5. Fit both datasets using compute_betas(), obtain the corresponding MSEs.
6. Plot a figure showing how MSE changes while sigma is increasing:
   - You should use the plotting format `-o` (line with circle markers)
   - The x-axis should be different settings of sigma.
   - The y-axis should be the MSEs calculated from the linear and quadratic datasets.
   - Make both axes **log scale**.
   - Label your x- and y-axes.
   - Make a legend.
   - Save the figure as `mse.pdf`. Do NOT display it (i.e., no `plt.show()`).

Hint: You may expect your graph to contain two nearly straight lines (with a little curve).

Hint: You should be getting something that looks like this:

Discovery

Experiment on different combinations of features, and learning methods (closed-form solution, gradient descent). Try to answer these questions:

- What is the best feature that predicts well?
- What are the best set of features that predict well?
- Which learning method is the most effective in terms of training error (MSE)?
- Which learning method is the most efficient (takes the least time)?
- What will happen if our dataset has more than 1 million entries?

Observe the figure you plotted, and try to answer these questions:

- What do you see happening in the linear case as the noise variance goes up?
- What do you see happening in the quadratic case as the noise variance goes up?
- What happens if the noise is extremely small?
- What happens if the noise is extremely large?

We won't grade these questions, but feel free to share your thoughts on Piazza!

Submission

Please submit your code zipped in a file called `hw5_onetid6.zip`. Inside your zip file, there should be **only** one file named: `regression.py`. Do NOT submit a Jupyter notebook .ipynb file.

Be sure to **remove all debugging output** before submission. Failure to remove debugging output will be **penalized (10pts)**.

Cheating results in -100 pts and further punishment

This assignment is due on 10/19/2021 at 10:59 am. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

HW5

Criteria	Ratings		Pts
get_dataset	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
print_stats	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
regression	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
gradient_descent	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
iterate_gradient	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
compute_betas	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
predict	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
synthetic_datasets	10 to >0.0 pts Full Marks	0 pts No Marks	10 pts
plot_mse (requires manual grading)	20 to >0.0 pts Full Marks	0 pts No Marks	20 pts
		Total Points: 100	

HW6: Neural Networks

Due Nov 4, 2021 by 10:59am **Points** 100 **Submitting** a file upload **Available** Oct 18, 2021 at 9pm - Nov 4, 2021 at 10:59am

This assignment was locked Nov 4, 2021 at 10:59am.

Update Nov 3: For evaluate_model function output, the average loss should be around 0.15. This is updated in the example.

Update Oct. 31: It is OK to have small difference between your training or evaluation results and the example on this page, which is caused by the randomness in the initialization of weights and DataLoader's shuffling. However, if your evaluation accuracy differs more than 5 percents, it may indicate your code has some problem.

Update Oct. 27: An example grading script file(s) provided to help you avoid mistakes before submission. Usage: simply put it in the same folder as your intro_pytorch.py, then run it. Your results will be printed out. The total points we test in this script is 30. Note that the test cases in the script are different from the actual grading and it only contains test cases for some (not all) functions. It is intended for a basic sanity check. You still need to pay attention to all functions and all the formats.

Assignment Goals

- Get Pytorch set up for your environment
- Familiarize yourself with the tools
- Perform some basic neural network tasks using Pytorch's utilities
- Happy deep learning! :)

Summary

Home-brewing every machine learning solution is not only time-consuming but potentially error-prone (as you may have discovered). One of the reasons we're using Python in this course is because it has some very powerful machine learning tools. Besides common scientific computing packages such as SciPy and NumPy, It's very helpful in practice to use frameworks such as Scikit-Learn, TensorFlow, Pytorch, and MXNet to support your projects, as these utilities have been developed by a team of professionals and undergo rigorous testing and verification.

In this homework, we'll be exploring the the Pytorch framework. Please complete the functions in this template intro_pytorch.py .

Part 1: Setting up the Python Virtual Environment

In this assignment, you will familiarize yourself with the Python Virtual Environment. Working in a virtual environment is an important part of working with modern ML platforms, and we want you to get a flavor of that through this assignment. Why do we prefer virtual environments? Virtual environments allow us to install packages within the virtual environment without affecting the host system setup. So you can maintain project-specific packages in respective virtual environments.

We suggest that you use the CS lab computers for this homework. You can also work on your personal system for the initial development, but finally, you will have to test your model on the CSL lab computers. Find more instructions: How to access CSL Machines Remotely

The following are the installation steps for Linux (CSL machines are recommended):

You will be working on Python 3 (instead of Python 2 which is no longer supported). Read more about Pytorch and Python version here . To check your Python version use:

```
python -V or python3 -V
```

If you have an alias set for python=python3 then both should show the same version (3.x.x)

Step1: For simplicity, we use the venv module (feel free to use other virtual envs such as Conda). To set up a Python Virtual Environment named Pytorch:

```
python3 -m venv /path/to/new/virtual/environment
```

For example, if you want to put the virtual environment in your home directory:

```
python3 -m venv ~/system-site-packages ~/Pytorch
```

If you want to learn more about Python virtual environments, a very good tutorial can be found here .

Here the name of our virtual environment is Pytorch (you can use any other name if you want).

Step2: Activate the environment:

```
source ~/Pytorch/bin/activate
```

Step3: From your virtual environment shell, run the following commands to upgrade pip and install the CPU version of Pytorch 1.8 (this homework only uses the basic functionality of Pytorch, so feel free to use older versions such as 1.6):

```
pip install --upgrade pip
```

```
pip install torchvi=1.8.0+cpu torchvision=0.9.0+cpu torchaudio=0.8.0 -f https://download.pytorch.org/whl/torch_stable.html
```

You can check the version of the packages installed using the following command:

```
pip freeze
```

Note: to deactivate the virtual environment, just type

```
deactivate
```

Part 2: Build Your First Neural Network

In this section, we will guide you step by step to build a simple deep learning model for predicting labels of handwritten images. You will learn how to build, train, evaluate models, and make predictions on test data using this model. We expect you to implement the following functions in Python.

- **part_one_loader(training=True)**
  - Input: an optional boolean argument (default value is True for training dataset)
  - Return: Dataloader for the training set (if training = True) or the test set (if training = False)
- **build_model()**
  - Input: none.
  - Return: an untrained neural network model
- **train_model(model, train_loader, criterion, T)**
  - Input: the model produced by the previous function, the train DataLoader produced by the first function, the criterion, and the number of epochs T for training.
  - Return: none
- **evaluate_model(model, test_loader, criterion, show_loss=True)**
  - Input: the trained model produced by the previous function, the test DataLoader, and the criterion.
  - It **prints** the evaluation statistics as described below (displaying the loss metric value if and only if the optional parameter has not been set to False).
  - Return: none
- **predict_label(model, test_images, index)**
  - Input: the trained model and test images.
  - It **prints** the top 3 most likely labels for the image at the given index, along with their probabilities.
  - Return: none

You are free to implement any other utility function. But we will only be testing the functionality using the above 5 APIs, so make sure that each of them follows the exact function signature and returns. You can also use helper methods to visualize the images from the MNIST dataset for a better understanding of the dataset and the labels. But it is totally optional and does not carry any points.

Import necessary packages

Here are some of the useful modules that may help us save a ton of efforts in the project:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np, matplotlib
from torchvision import datasets, transforms
```

The following 5 sections explain the details for each of the above functions you are required to implement respectively.

Get the DataLoader

We will use the "hello world" dataset MNIST , which consists of handwritten digits from 0 to 9.

Hint1: note that Pytorch already contains various datasets for you to use, so there is no need to manually download from the Internet. Specifically, `torchvision.datasets.MNIST()` can be used to retrieve and return a Dataset object `torchvision.datasets.mnist.MNIST` which is a wrapper that contains image inputs (as 2D arrays) and labels (0 to 9) representing handwritten numbers:

```
train_set = datasets.MNIST('./data', train=True, download=True,
                           transform=transform)
test_set = datasets.MNIST('./data', train=False,
                          transform=transform)
```

The train set contains images and labels we'll be using to train our neural network; the test set contains images and labels for model evaluation. Here we set the location where the dataset is downloaded as the `./data` folder in the current directory. If you encounter HTTP Error 403, check here .

Note that input preprocessing can be done by specifying `transform` as our `custom_transform`:

```
custom_transform= transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
    ])
```

- `transforms.ToTensor()` converts a PIL Image or `numpy.ndarray` to tensor.
- `transforms.Normalize()` normalizes the tensor with a mean and standard deviation which goes as the two parameters respectively. Feel free to check the official doc for more details.

Hint2: After obtaining the dataset object, you may wonder how to retrieve images and labels during training and testing. Luckily, Pytorch provides such a class called `torch.utils.data.DataLoader` that implements the iterator protocol. It also provides useful features such as:

- Batching the data
- Shuffling the data
- Load the data in parallel using `multiprocessing`.
- ...

Below is the full signature (for more details, check here):

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
           batch_sampler=None, num_workers=0, collate_fn=None,
           pin_memory=False, drop_last=False, timeout=0,
           worker_init_fn=None, *, prefetch_factor=2,
           persistent_workers=False)
```

As an introductory project, we won't use complicated features. We ask you to set the `batch_size` = 50 for both train loader and test loader. Besides, set `shuffle=False` for the test loader. Given a Dataset object data, we can obtain its DataLoader as follows:

```
loader = torch.utils.data.DataLoader(data_set, batch_size = 50)
```

Putting it all together, you should be ready to implement the `get_data_loader()` function. Note that when the optional argument is unspecified, the function should return the **Dataloader for the training set**. If the optional argument is set to false, the **Dataloader for the test set** is returned. The expected output is as follows:

```
>>> train_loader = get_data_loader()
>>> type(train_loader)
<class 'torch.utils.data.dataloader.DataLoader'>

>>> print(train_loader.dataset)
Dataset MNIST
    Number of datapoints: 60000
    Split: train
    Root location: ./data
    Standard Transform
Transforms (if any): ToTensor()
                     Normalize(mean=(0.1307,), std=(0.3081,))

>>> test_loader = get_data_loader(False)
```

Build Your Model

After setting up the data loaders, let's build the model we're going to use with the datasets. Neural networks in Pytorch are composed of layers. You've heard about these in the lecture, but take a minute to look through this simple example (It's nice and short!) to get an idea of what the implementation logistics will look like.

We use the following layers in this order:

1. A Flatten layer to convert the 2D pixel array to a 1D array.
2. A Dense layer with 128 nodes and a ReLU activation.
3. A Dense layer with 64 nodes and a ReLU activation.
4. A Dense layer with 10 nodes.

In this assignment, you are expected to use a Sequential container to hold these layers:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(*, *),
    nn.ReLU(),
    nn.Linear(*, *),
    nn.ReLU(),
    )
```

After building the model, the expected output should be like this:

```
>>> model = build_model()
>>> print(model)
Sequential(
  (0): Flatten(),
  (1): Linear(in_features=?, out_features=?, bias=True)
  (2): ReLU()
  (3): Linear(in_features=?, out_features=?, bias=True)
  (4): ReLU()
)
```

Note that the Flatten layer just serves to reformat the data.

Train Your Model

After building the model, now we are ready to implement the training procedure.

One of the parameters of `train_model(..., criterion, ...)` is the criterion, which can be specified as:

```
criterion = nn.CrossEntropyLoss()
```

Here we use the cross-entropy loss nn.CrossEntropyLoss() (this criterion combines nn.LogSoftmax() and nn.NLLLoss())

Inside the function `train_model()`, you may need to pick your favorite optimization algorithm by setting up an optimizer first: here we use stochastic gradient descent (SGD) with a learning rate of 0.001 and momentum of 0.9:

```
opt = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

A note on the major difference between gradient descent (GD) and SGD: In GD, all samples in the training set are used to update parameters in a particular iteration; while in SGD, only a random subset of training samples are used to update parameters in a particular iteration. SGD often converges much faster than GD for large datasets.

The standard training procedure contains 2 `for` loops: the outer `for` loop iterates over epochs, while the inner `for` loop iterates over batches of (images, labels) pairs from the train DataLoader. Feel free to check the `fmnist` example in the official tutorial to for more details (please pay attention to the order of `zero_grad()`, `backward()` and `step()`).

A kind reminder: please set your model to `train` mode before iterating over the dataset. This can be done with the following call:

```
model.train()
```

We ask you to print the training status after every epoch of training in the following format (it should have 3 components per line):

```
Train Epoch: 7 Accuracy: 7/7(77.77%) Loss: 7.777
```

Then the training process (for 5 epochs) will be similar to the following (numbers can be different):

```
>>> train_model(model, train_loader, criterion, T = 5)
Train Epoch: 0   Accuracy: 47771/60000(79.62%) Loss: 0.748
Train Epoch: 1   Accuracy: 51308/60000(85.51%) Loss: 0.479
Train Epoch: 2   Accuracy: 51854/60000(86.42%) Loss: 0.436
Train Epoch: 3   Accuracy: 52557/60000(87.59%) Loss: 0.396
Train Epoch: 4   Accuracy: 53142/60000(88.57%) Loss: 0.366
```

Here are a few specific requirements for the format:

- We count the first epoch as Epoch 0
- All the information should be summarized in one line for each epoch. (e.g. in total you should print 5 lines if you train for 5 epochs)
- Accuracy in percentage should be put inside parentheses
- Accuracy should be printed before Loss
- Loss denotes the average loss per epoch (accumulated loss in an epoch / length of the dataset)

You should be able to reach at least 90% accuracy after 5 epochs of training.

Evaluate Your Model

After the model is trained, we need to evaluate how good it is on the test set. The process is very similar to that of training, except that you need to turn the model into evaluation mode:

```
model.eval()
```

Besides, there is no need to track gradients during testing, which can be disabled with the context-manager:

```
with torch.no_grad():
    for data, labels in test_loader:
```

You are expected to print both the test Loss and the test Accuracy if `show_loss` is set to True (print Accuracy only otherwise) in the following format:

```
>>> evaluate_model(model, test_loader, criterion, show_loss = False)
Accuracy: 91.40%

>>> evaluate_model(model, test_loader, criterion, show_loss = True)
Average Loss: 0.2539
Accuracy: 91.40%
```

Format the Accuracy with **two** decimal places and the accuracy should be shown as a percentage. Format the Loss with **four** decimal places. The loss should be printed in a separate line before Accuracy (as shown above).

Predict the Labels

Instead of testing on a whole dataset, sometimes it's more convenient to examine the model's output on a single image.

As it's easier for humans to read and interpret probabilities, we need to use a Softmax function to convert the output of your final Dense layer into probabilities (note that by default your model outputs logits). Generally, Softmax is often used as the activation for the last layer of a classification network because the result can be interpreted as a categorical distribution. Specifically, once we obtain the logits, we can use:

```
prob = F.softmax(logits, dim=1)
```

You can assume the input `test_images` in `predict_label(model, test_images, index)` is a torch tensor with the shape Nx1x28x28. Your implementation should display the top **three** most likely class labels, **in descending order of predicted probability**, three lines in total, for the image **at the given index** (assumed to be valid i.e. $0 \leq i \leq N-1$) along with their respective probabilities in percentage (again, your output will vary in its exact numbers but should follow the format below):

```
>>> predict_label(model, prod_set, 1)
bag: 96.42%
three: 0.42%
one: 0.09%
```

where we assume the class names are:

```
class_names = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine']
```

Deliverable:

A single file named **intro_pytorch.py** containing the methods mentioned in the program specification section. Please pay close attention to the format of the print statements. Incorrect format will lead to point deductions.

Submission

Please submit your files in a zip file named **hw6_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be one single file named **intro_pytorch.py**. Do not submit a Jupyter notebook .ipynb file.

All code should be contained in functions OR under a

```
if __name__== "__main__":
```

check so that it will not run if your code is imported to another program.

Be sure to **remove all debugging/testing output** before submission. Failure to remove them will be **penalized (10pts)**.

Regrading: Any regrading request needs to be submitted within 72 hours from the grade release. The regrading requests should be about more than 10 point deduction due to output format mismatch that can be fixed by modifying one or two lines in the code; and 10 points will be deducted for such output format mismatch. Those requests due to other mistakes (e.g., logic flaw of the code) are not accepted. If a regrading request isn't justifiable (the initial grade is correct and clear, subject to the instructors' judgment), the request for regrading will be penalized (10 pts).

This assignment is due on Nov. 4, 10:59AM. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

Rubric

Criteria	Ratings	Pts
get_data_loader() successfully returns the desired train and test loader	20 pts Full Marks / 0 pts No Marks	20 pts
build_model() returns an untrained, Sequential model with the specified components	20 pts Full Marks / 0 pts No Marks	20 pts
train_model() trains the model for the specified epochs and the print format is correct	25 pts Full Marks / 0 pts No Marks	25 pts
evaluate_model() displays different output based on its optional argument	5 pts Full Marks / 0 pts No Marks	5 pts
predict_label() displays correctly formatted output. predict_label() displays expected top-3 categories; predict_label() displays probabilities, not logits	30 pts Full Marks / 0 pts No Marks	30 pts

Total Points: 100

HW7: Convolutional Neural Networks

Due Nov 16, 2021 by 10:59am **Points** 100 **Submitting** a file upload **Available** Nov 4, 2021 at 11am - Nov 16, 2021 at 11am

This assignment was locked Nov 16, 2021 at 11am.

Update

November 13 11:00AM: Relaxing the range in part 3 from 1% to 1.5%. The sample testing script should change its range to 12.35 to 15.35.

November 8 9:30PM: a testing script is now available here: sample_testing.py. Run the script using "python3 sample_testing.py".

November 8 2PM: there are issues with downloading the train/validation splits from the miniplaces github. After unpacking the backup tarball from @477, you will likely see an SSL Certificate error. Download the train.txt and val.txt files into the data/miniplaces folder. The dataloader should work now.

November 4, 2:00 PM: due date extended to November 16 at 10:59am

Assignment Goals

- Compare creating a CNN to a basic neural network as in HW6
- Design your own deep network for scene recognition

Summary

Your implementation in this assignment might take one or two hours to run. **We highly recommend to start working on this assignment early!** In this homework, we will explore building deep neural networks, including Convolutional Neural Networks (CNNs), using PyTorch. Helper code is provided in this assignment and can be downloaded from this zip file code_hw7-2.zip ↓ .

In this HW, you are still expected to use the Pytorch backend and virtual environment for programming where you can find the tutorials in HW6.

Design a CNN model for MiniPlaces Dataset

In this part, you will design a simple CNN model along with your own convolutional network for a more realistic dataset -- MiniPlaces, again using PyTorch.

Dataset

MiniPlaces is a scene recognition dataset developed by MIT. This dataset has 120K images from 100 scene categories. The categories are mutually exclusive. The dataset is split into 100K images for training, 10K images for validation, and 10K for testing.

Helper Code

We provide helper functions in **train_miniplaces.py** and **dataloader.py**, and skeleton code in **student_code.py**. See the comments in these files for more implementation details.

The original image resolution for images in MiniPlaces is 128x128. To make the training feasible, our data loader reduces the image resolution to 32x32. You can always assume this input resolution. Our data loader will also download the full dataset the first time you run train_miniplaces.py

Before the training procedure, we define the dataloader, model, optimizer, image transform and criterion. We execute the training and testing in function **train_model** and **test_model**, which is similar to what we have for HW6.

Part I Creating LeNet-5

Background: LeNet was one of the first CNNs and its success was foundational for further research into deep learning. We are just implementing an existing architecture in the assignment, but it might be interesting to think about why early researchers chose certain kernel sizes, padding, and strides which we learned about conceptually in class.

In this part, you have to implement a classic CNN model, called LeNet-5 in Pytorch for the MiniPlaces dataset. We use the following layers in this order:

1. One convolutional layer with the number of output channels to be 6, kernel size to be 5, stride to be 1, padding to be 0 followed by a relu activation layer and then a 2D max pooling layer (kernel size to be 2, padding to be 0 and stride to be 2).
2. One convolutional layer with the number of output channels to be 16, kernel size to be 5, stride to be 1, padding to be 0 followed by a relu activation layer and then a 2D max pooling layer (kernel size to be 2, padding to be 0 and stride to be 2).
3. A Flatten layer to convert the 3D tensor to a 1D tensor.
4. A Linear layer with output dimension to be 256, followed by a relu activation function.
5. A Linear layer with output dimension to be 128, followed by a relu activation function.
6. A Linear layer with output dimension to be the number of classes (in our case, 100).

You have to fill in the **LeNet** class in **student_code.py**. You are expected to create the model following this tutorial, which is different from using nn.Sequential() in the last HW.

In addition, given a batch of inputs with shape [N, C, W, H], where N is the batch size, C is the input channel and W, H are the width and height of the image (both 32 in our case), you are expected to return both the output of the model along with the shape of the intermediate outputs for the above 6 stages. The shape should be a dictionary with the keys to be 1,2,3,4,5,6 (integers) denoting each stage, and the corresponding value to be a list that denotes the shape of the intermediate outputs.

Hint: The expected model has the following form:

```
class LeNet(nn.Module):

    def __init__(self, input_shape=(32, 32), num_classes=100):

        super(LeNet, self).__init__()

        # certain definitions

    def forward(self, x):

        shape_dict = {}

        # certain operations

        return out, shape_dict
```

Shape_dict should have the following form: {1: [a,b,c,d], 2:[e,f,g,h], .., 6: [x,y]}

The linear layer and the convolutional layer have bias terms.

Part II Count the number of trainable parameters of LeNet-5

Background: As discussed in lecture, fully connected models (like what we made in the previous homework) are dense with many parameters we need to train. After finishing this part, it might be helpful to think about the number of parameters in this model compared to the number of parameters in a fully connected model of similar depth (similar number of layers). Especially, how does the difference in size impact performance?

In this part, you are expected to return the number of trainable parameters of your created LeNet model in the previous part. You have to fill in the function **count_model_params** in **student_code.py**.

The function output is in the unit of Million(1e6). **Please do not use any external library which directly calculates the number of parameters (other libraries, such as NumPy can be used)!**

Hint: You can use the model.named_parameters() to gain the name and the corresponding parameters of a model. Please do not do any rounding to the result.

Part III Training LeNet-5 under different configurations

Background: Part of creating neural networks is designing the architecture (part 1). However, there are still ways of tweaking the neural net to change its performance. In this section, we can see how batch size, learning rate, and number of epochs impact how well the model learns. As you get your results, it might be helpful to think about how the changes have impacted the training (hint: think back to gradient descent from the neural net lectures).

Based on the LeNet-5 model created in the previous parts, in this section, you are expected to train the LeNet-5 model under different configurations.

You will use similar implementations of **train_model** and **test_model** as you did for HW6 (which we provide in student_code.py). When you run train_miniplaces.py, the python script will save two files in the "outputs" folder.

- **checkpoint.pth.tar** is the model checkpoint at the latest epoch.
- **model_best.pth.tar** is the model weights that has highest accuracy on the validation set.

Our code supports resuming from a previous checkpoint, such that you can pause the training and resume later. This can be achieved by running

```
python train_miniplaces.py --resume ./outputs/checkpoint.pth.tar
```

After training, we provide **eval_miniplaces.py** to help you evaluate the model on the validation set and also help you in timing your model. This script will grab a pre-trained model and evaluate it on the validation set of 10K images. For example, you can run

```
python eval_miniplaces.py --load ./outputs/model_best.pth.tar
```

The output shows the validation accuracy and also the model evaluation time in seconds (see an example below).

```
=> Loading from cached file ./data/miniplaces/cached_val.pkl ...
=> loading checkpoint './outputs/model_best.pth.tar'
=> loaded checkpoint './outputs/model_best.pth.tar' (epoch x)
Evaluting the model ...
[Test set] Epoch: xx, Accuracy: xx.xx%
Evaluation took 2.26 sec
```

You can run this script a few times to see the average runtime of your model.

Please train the model under the following different configurations:

1. The default configuration provided in the code, which means you do not have to make modifications.
2. Set the batch size to 8, the remaining configurations are kept the same.
3. Set the batch size to 16, the remaining configurations are kept the same.
4. Set the learning rate to 0.05, the remaining configurations are kept the same (revert batch size to default of 32).
5. Set the learning rate to 0.01, the remaining configurations are kept the same.
6. Set the epochs to 20, the remaining configurations are kept the same (revert learning rate to default of 0.001).
7. Set the epochs to 5, the remaining configurations are kept the same.

After training, you are expected to get the validation accuracy for each configuration using the best model (**model_best.pth.tar**), then save these accuracies into a **results.txt** file, where the accuracy of each configuration is placed in one line in order. Your .txt file will end up looking like this:

```
18.00
17.39
12.31
...
```

The exact accuracy may not align well with your results. They are just for illustration purposes.

You have to submit the **results.txt** file together with your **student_code.py**.

Optional: Profiling Your Model

You might find that the training or evaluation of your model is a bit slower than expected. Fortunately, PyTorch has its own profiling tool. Here is a quick tutorial of using PyTorch profiler. You can easily inject the profiler into **train_miniplaces.py** to inspect the runtime and memory consumption of different parts of your model. A general principle is that a deep (many layers) and wide (many feature channels) network will train much slower. It is your design choice to balance between the efficiency and the accuracy.

Optional: Training on CSL

We recommend training the model on your local machine or CSL machine and time your model (using eval_miniplaces.py) on CSL machines. If you decide to train the model on a CSL machine, you will need to find a way to allow your remote session to remain active when you are disconnected from CSL. In this case, we recommend using tmux, a terminal multiplexer for Unix-like systems. tmux is already installed on CSL. To use tmux, simply type **tmux** in the terminal. Now you can run your code in a tmux session. And the session will remain active even if you are disconnected.

- If you want to detach a tmux session without closing it, press "ctrl + b" then "d" (detach) within a tmux session. This will exist to the terminal while keeping the session active. Later you can re-attach the session.
- If you want to enter an active tmux session, type "tmux a" to attach to the last session in the terminal (outside of tmux).
- If you want to close a tmux session, press "ctrl + b" then "x" (exit) within a tmux session. You won't be able to enter this session again. Please make sure that you close your tmux sessions after this assignment.
- See here for a brief tutorial on the powerful tool of tmux.

Deliverable

You will need to submit **student_code.py** together with your **results.txt** for Part III. The validation accuracy for different configurations must be in **.txt** format and named as **results.txt**.

Submission Notes

You will need to submit a single zip file named **hw7_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). This file will include student_code.py and the validation accuracy file with the proper name as mentioned above. Do not submit a Jupyter notebook .ipynb file.

Be sure to **remove all debugging output** before submission. Failure to remove debugging output will be **penalized (10pts)**.

If a regrading request isn't justifiable (the initial grade is correct and clear, subject to the instructors' judgment), the request for regrading will be **penalized (10 pts)**.

This assignment is due on Tuesday November 16, 10:59 AM. We highly recommend to start early. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

Scoring Rubric

Criteria	Ratings		Pts
Correctly create the LeNet model with the right output shape.	30 pts Full Marks	0 pts No Marks	30 pts
Correctly calculate the number of parameters of LeNet.	20 pts Full Marks	0 pts No Marks	20 pts
Correctly train the model under different configurations and report the right accuracy.	50 pts Full Marks	0 pts No Marks	50 pts

HW8: A* Search

Due Nov 23, 2021 by 10:59am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Nov 15, 2021 at 10pm - Nov 23, 2021 at 10:59am

This assignment was locked Nov 23, 2021 at 10:59am.

Update Nov 19: Max queue length output will not be checked when grading. The provided student testing script is updated to reflect this.

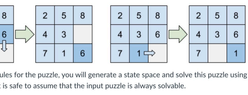
Assignment Goals

- Deepen understanding of state space generation
- Practice implementation of an efficient search algorithm

Summary

This assignment is about solving the 8-tile puzzle we have discussed in class. The 8-tile puzzle was invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3x3 grid with 8 tiles labeled 1 through 8 and an empty grid. The goal is to rearrange the tiles so that they are in order.

You solve the puzzle by moving the tiles around. For each step, you can only move one of the neighbor tiles (left, right, top, bottom) into an empty grid. And all tiles must stay in the 3x3 grid. An example is shown in the picture below. In this example, there are only 2 valid moves, i.e., either moving 6 down or moving 1 right.



Given these rules for the puzzle, you will generate a state space and solve this puzzle using the **A* search algorithm**. Note that not all 8-tile puzzles are solvable. For this assignment, it is safe to assume that the input puzzle is always solvable.

Program Specification

The code for this program should be written in Python, in a file called **funny_puzzle.py**.

You may represent your states internally however you wish; a two-dimensional list or numpy matrix may be useful. We will provide states in a one-dimensional list of integers, with the empty space represented as 0. We will represent the initial state above in our input as `[2,5,8,4,3,6,7,1,0]` and the successors pictured are, in order, `[2,5,8,4,3,0,7,1,6]`, `[2,5,8,4,3,6,7,0,1]`.

In this assignment, you will need to implement a priority queue. We highly recommend to use package `heapq` for the implementation. You should refer to heapq if you are not familiar with it.

Any code that you do not personally write **should be cited** asyou would cite a quotation in an essay; that is, with its author and source in as complete a format as possible. Code used without citation will be considered plagiarism and violates our policy of appropriate academic conduct. For example:

```
''' Original author: ecto
    Source: https://codepen.io/ectochange/pen/dZepPB

    The following function was translated from the original Javascript for the current program
'''
def addRandomTile():
```

Even if you modify the code, you should still cite your original inspiration(and any assisting TAs or peer mentors!).

Goal State

The goal state of the puzzle is `[1, 2, 3, 4, 5, 6, 7, 8, 0]`, or visually:

1	2	3
4	5	6
7	8	

Heuristic

Since we are using the A* search algorithm, we need a heuristic function. Recall the Manhattan distance mentioned in lecture (the l1-norm). **We will use the sum of Manhattan distance of each tile to its goal position as our heuristic function.**

The Manhattan distance in this case is the sum of the absolute difference of their x co-ordinates and y co-ordinates.

In our first example puzzle `{2, 5, 8, 4, 3, 6, 7, 1, 0}`, the h() value for the original state is 10. This is computed by calculating the Manhattan distance of each title and summing them. Specifically, tiles 4/6/7 are already in place, thus they have 0 distances.

Tile 1 has a Manhattan distance of 3 `{manhattan((2,1), (0,0)) = abs(2-0) + abs(1-0) = 3}`, tiles 2/3/5/8 have distances of 1/2/1/3, respectively.

Caution: do not count the distance of title '0' since it is actually not a tile but an empty grid.

Functions

For this program you should write at least **two (2)** Python functions:

1. **print_succ(state)** — given a state of the puzzle, represented as a single list of integers with a 0 in the empty space, print to the console all of the possible successor states
2. **solve(state)** — given a state of the puzzle, perform the A* search algorithm and print the path from the current state to the goal state

You may, of course, add any other functions you see fit, but these two functions must be present and work as described here.

Print Successors

This function should print out the successor states of the initial state, as well as their heuristic value according to the function described above. The number of successor states depends on the current state. There could be 2 (empty grid is at corner), 3 (empty grid is at middle of a boundary), or 4 (empty grid is at the center of a boundary) successors.

```
>>> print_succ([1,2,3,4,5,0,6,7,8])
[1, 2, 0, 4, 5, 3, 6, 7, 8] h=6
[1, 2, 3, 4, 0, 5, 6, 7, 8] h=6
[1, 2, 3, 4, 5, 8, 6, 7, 0] h=6
```

We **do** require that these be printed in a specific order: if you consider the state to be a nine-digit integer, the states should be sorted in *ascending* order. Conveniently, if you ask Python to sort one-dimensional arrays, it will adhere to this order by default; don't do more work than you have to:

```
>>> lists = [[1,2,3,4,5,8,6,7,0],
... [1,2,3,4,0,5,6,7,8],
... [1,2,0,4,5,3,6,7,8]]
>>> sorted(lists)
=> [[1, 2, 0, 4, 5, 3, 6, 7, 8],
    [1, 2, 3, 4, 0, 5, 6, 7, 8],
    [1, 2, 3, 4, 5, 8, 6, 7, 0]]
```

Priority Queue

Now is a good time to implement the priority queue in your code.

We recommend you use the python library **heapq** to create your priority queue. Here is a quick example:

```
>>> import heapq
>>> pq = []
>>> heapq.heappush(pq,(5, [1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1)))
>>> print(pq)
[(5, [1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1))]
```

The code will push an item ([1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1)) with priority 5 into the queue. This format follows **(g+h, state, (g, h, parent_index))** representing both the cost, state and the parent index in A* search (a parent index of -1 denotes the initial state, without any parent). For more details, please refer to the documentation of `heapq`.

.

To get the final path, for each element in the priority queue we need to remember its parent state. Remember to store the state when you pop it from the priority queue, so you could refer to it later when you generate the final path.

Here is how you can pop from a priority queue,

```
>>> b = heapq.heappop(pq)
>>> print(b)
(5, [1, 2, 3, 4, 5, 0, 6, 7, 8], (0, 5, -1))
>>> print(pq)
[]
```

The priority queue is stored in a list and you can print its items (with the associated priority) by using

```
>>> # assume that you have generated and enqueued the successors
>>> print(*pq, sep='\n')
(7, [1, 2, 0, 4, 5, 3, 6, 7, 8], (1, 6, 0))
(7, [1, 2, 3, 4, 0, 5, 6, 7, 8], (1, 6, 0))
(7, [1, 2, 3, 4, 5, 8, 6, 7, 0], (1, 6, 0))
```

Note that the heappush maintains the priority in ascending order, i.e., heappop will always pop the element with the smallest priority.

We require that the states **with the same cost (priority)** to be popped in a specific order: if you consider the state to be a nine-digit integer, the states should be sorted in *ascending* order - just like we mentioned above. If you follow the format in pq as shown above, heapq will automatically take care of this and you do not need more work.

Solve the Puzzle

This function should print the solution path from the provided initial state to the goal state, along with the heuristic values of each intermediate state according to the function described above, and total moves taken to reach the state. Recall that our cost function g(n) is the total number of moves so far, and every valid successor has an additional cost of 1.

```
>>> solve([4,3,8,5,1,6,7,2,0])
[4, 3, 8, 5, 1, 6, 7, 2, 0] h=10 moves: 0
[4, 3, 8, 5, 1, 0, 7, 2, 6] h=11 moves: 1
[4, 3, 0, 5, 1, 8, 7, 2, 6] h=10 moves: 2
[4, 0, 3, 5, 1, 8, 7, 2, 6] h=9 moves: 3
[4, 1, 3, 5, 0, 8, 7, 2, 6] h=8 moves: 4
[4, 1, 3, 5, 8, 0, 7, 2, 6] h=7 moves: 5
[4, 1, 3, 5, 8, 6, 7, 2, 0] h=6 moves: 6
[4, 1, 3, 5, 8, 6, 7, 0, 2] h=7 moves: 7
[4, 1, 3, 5, 0, 6, 7, 8, 2] h=6 moves: 8
[4, 1, 3, 0, 5, 6, 7, 8, 2] h=5 moves: 9
[0, 1, 3, 4, 5, 6, 7, 8, 2] h=4 moves: 10
[1, 0, 3, 4, 5, 6, 7, 8, 2] h=3 moves: 11
[1, 3, 0, 4, 5, 6, 7, 8, 2] h=4 moves: 12
[1, 3, 6, 4, 5, 0, 7, 8, 2] h=5 moves: 13
[1, 3, 6, 4, 5, 2, 7, 8, 0] h=4 moves: 14
[1, 3, 6, 4, 5, 2, 7, 0, 8] h=5 moves: 15
[1, 3, 6, 4, 0, 2, 7, 5, 8] h=6 moves: 16
[1, 3, 6, 4, 2, 0, 7, 5, 8] h=5 moves: 17
[1, 3, 0, 4, 2, 6, 7, 5, 8] h=4 moves: 18
[1, 0, 3, 4, 2, 6, 7, 5, 8] h=3 moves: 19
[1, 2, 3, 4, 0, 6, 7, 5, 8] h=2 moves: 20
[1, 2, 3, 4, 5, 6, 7, 0, 8] h=1 moves: 21
[1, 2, 3, 4, 5, 6, 7, 8, 0] h=0 moves: 22
Max queue length: 394
```

The max queue length is tracked by our priority queue implementation (as the variable max_len), and displaying it may be useful for debugging purposes, but it is **not** required output.

Note: please do not use `exit()` or `quit()` to end your function when you find a path, as this will cause Python to close entirely.

Additional Examples

For your successor function:

```
>>> print_succ([8,7,6,5,4,3,2,1,0])
[8, 7, 6, 5, 4, 0, 2, 1, 3] h=17
[8, 7, 6, 5, 4, 3, 2, 0, 1] h=17
```

For your solution function:

```
>>> solve([1,2,3,4,5,6,7,0,8])
[1, 2, 3, 4, 5, 6, 7, 0, 8] h=1 moves: 0
[1, 2, 3, 4, 5, 6, 7, 8, 0] h=0 moves: 1
Max queue length: 3
```

Additional hints

For some complicated cases, your solve function could take a very long time to get an answer, or maybe even the entire program breaks. It is recommended that you don't explore any state that has been visited before (justified because the Manhattan distance heuristic here is consistent.). To do this, you can use a set to track which states have been visited already.

Submission Notes

Please submit your files in a zip file named **hw8_<netid>.zip** , where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **funny_puzzle.py** . Do not submit a Jupyter notebook .ipynb file. Failure to adhere to this will be **penalized (10 pts)**.

Be sure to **remove all testing/debugging output** before submission; your functions should run silently except required I/O. Failure to adhere to this will be **penalized (10pts)** .

We provide a testing script for you to check the output format. Unzip the file and put your funny_puzzle.py in the same directory as test.py; then run "python3 test.py". If you pass all the tests, you will get 15 points. Note that the actual grading test cases are different and this is only intended for a sanity check.

Make sure to test your submission on the **CS Linux** machines!

Regrading: Any regrading request needs to be submitted within 72 hours from the grade release. The regrading requests should be about more than 10 point deduction due to output format mismatch that can be fixed by modifying one or two lines in the code, and 10 points will be deducted for such output format mismatch. Those requests due to other mistakes (e.g., logic flow of the code) are not accepted. If a regrading request isn't justifiable (the initial grade is correct and clear, subject to the instructors' judgment), the request for regrading will be penalized (10 pts).

This assignment is due on **Nov 23 at 10:59 AM** . It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

HW8 Rubric (1)

Criteria	Ratings		Pts
print_succ 1	5 pts Full Marks	0 pts No Marks	5 pts
print_succ 2	5 pts Full Marks	0 pts No Marks	5 pts
print_succ 3	5 pts Full Marks	0 pts No Marks	5 pts
print_succ 4	5 pts Full Marks	0 pts No Marks	5 pts
solve 1	10 pts Full Marks	0 pts No Marks	10 pts
solve 2	10 pts Full Marks	0 pts No Marks	10 pts
solve 3	10 pts Full Marks	0 pts No Marks	10 pts
solve 4	10 pts Full Marks	0 pts No Marks	10 pts
solve complicated 1	20 pts Full Marks	0 pts No Marks	20 pts
solve complicated 2	20 pts Full Marks	0 pts No Marks	20 pts

Total Points: 100

HW9: Game AI

Due Dec 7, 2021 by 11am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Nov 23, 2021 at 11am - Dec 7, 2021 at 11am

This assignment was locked Dec 7, 2021 at 11am.

Update

Update Dec 6 9:00PM: Fixed a win condition in test script.

Update Dec 4 2:30PM: New test file which includes scores here.

Update Dec 1 12:30PM: MakeMove formatting should be a list of 1 tuple in drop phase and a list of 2 tuples in move phase.

Update Nov 29 6:30PM: HW9 due date extended to next Tuesday (12/7). We are still planning to release HW10 this Thursday though.

Update Nov 29: To clarify, the helper functions you need to implement are not directly tested, so they do not need to have the same parameters as the writeup.

Update Nov 27: Sample testing script is available here. This is just a simulation of a game against a random opponent; beating the random opponent does not guarantee full credit on the assignment. See the HW9 rubric for the grading breakdown.

Assignment Goals

- Familiarize yourself with solved games in AI.
- Practice implementing a minimax algorithm.
- Develop an internal state representation in Python and get familiarized with classes in Python.

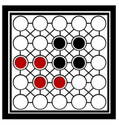
Summary

In this assignment, you'll be developing an AI game player for a modified version of the game called Teeko. We call this modified version Teeko2.

As you're probably aware, there are certain kinds of games that computers are very good at, and others where even the best computers will routinely lose to the best human players. The class of games for which we can predict the best move from any given position (with enough computing power) is called Solved Games. Teeko is an example of such a game, and this week you'll be implementing a computer player for a modified version of it.

How to play Teeko

Teeko is very simple:



It is a game between two players on a 5x5 board. Each player has four markers of either **red** or **black**. Beginning with black, they take turns placing markers (the "drop phase") until all markers are on the board, with the goal of getting four in a row horizontally, vertically, or diagonally, or in a 2x2 box as shown above. If after the drop phase neither player has won, they continue taking turns moving one marker at a time -- to an adjacent space only! (Note this includes diagonals, not just left, right, up, and down one space.) -- until one player wins.

How to play Teeko2

The Teeko2 rules are almost identical to those of Teeko but we will exchange a rule. Specifically, we remove the 2x2 box winning condition and replace it with a 3x3 box winning condition -- same colored markers at the four corners of a 3x3 square. Mathematically, if (i,j) is the center of a 3x3 board, then it must be that (i,j) is empty and that there is a marker of the appropriate color on each of (i-1,j-1), (i-1,j+1), (i+1,j-1), and (i+1,j+1).

Win conditions summarized for Teeko2:

- Four same colored markers in a row horizontally, vertically, or diagonally.
- Four same colored markers at the corners of a 3x3 square and the center of the square is empty.

Program Specification

This week we're providing a basic Python class and some driver code, and it's up to you to finish it so that your player is actually intelligent.

Here is our partially-implemented game: game.py (game.py)

(If your computer doesn't like downloading .py files, grab game.py.txt(game.py.txt) and remove the .txt extension.)

If you run the game as it stands, you can play as a human player against a very stupid AI. This sample game currently works through the drop phase, and the AI player only plays randomly.

First, familiarize yourself with the comments in the code and classes in Python (if you are new to this, then you can refer to this tutorial.). There are several TODOs that you will complete to make a more "intelligent" player. You are allowed to implement helper functions but please do not change the signature (parameters/name etc) of the functions given in the starter code.

Make Move

The `make_move(self, state)` method begins with the current state of the board. It is up to you to generate the subtree of depth *d* under this state, create a heuristic scoring function to evaluate the "leaves" at depth *d* (as you may not make it all the way to a terminal state by depth *d* so these may still be internal nodes) and propagate those scores back up to the current state, and select and return the best possible next move using the minimax algorithm.

The following section will provide you with the steps that you should be implementing as a part of this exercise. You will be implementing several helper functions for your make_move method to work (for the helper functions, the parameters do not have to be the exact same as the write-up shows because none of those functions are directly tested).

You may assume that your program is always the **max** player.

1. Generate Successors

Define a successor function (e.g. `succ(self, state)`) that takes in a board state and returns a list of the legal successors. During the drop phase, this simply means adding a new piece of the current player's type to the board; during continued gameplay, this means moving any one of the current player's pieces to an unoccupied location on the board, adjacent to that piece.

Note: wrapping around the edge is NOT allowed when determining "adjacent" positions.

2. Evaluate Successors

Using `game_value(self, state)` as a starting point, create a function to score each of the successor states. A terminal state where your AI player wins should have the maximal positive score (1), and a terminal state where the opponent wins should have the minimal negative score (-1).

Finish coding the diagonal and 3x3 win condition checks for the game_value method.

Define a `heuristic_game_value(self, state)` function to evaluate non-terminal states. For some hints, check out Slides 56 through 59 of the Games I Lecture Slides (You should call the game_value method from this function to determine whether the **state** is a terminal state before you start evaluating it heuristically.) This function should return some floating-point value between 1 and -1.

3. Implement Minimax

Follow the pseudocode recursive functions on slide #58 of our Game I lecture, incorporating the depth cutoff to ensure you terminate in under 5 seconds.

- Define a `max_value(self, state, depth)` function where your first call will be `max_value(self, curr_state, 0)` and every subsequent recursive call will increase the value of **depth**.
- When the depth counter reaches your tested depth limit OR you find a terminal state, terminate the recursion.

We recommend timing your make_move() method (use Python's time library), to see how deep in the minimax tree you can explore in under five seconds. Time your function with different values for your depth and pick one that will safely terminate in under 5 seconds.

Testing Your Code

We will be testing your implementation of **make_move()** under the following criteria:

1. Your AI must follow the rules of Teeko2 as described above, including the drop phase and continued gameplay.
2. Your AI must return its move as described in the comments, without modifying the current state.
3. Your AI must select each move it makes in **five seconds or less**.
4. Your AI must be able to beat a random player in 2 out of 3 matches.

We will be timing your make_move() remotely on the CS Linux machines, to be fair in terms of processing power.

Submission Notes

Please submit your files in a zip file named **hw9_<netid>.zip**, where you replace <netid> with your netID (your wisc.edu login). Inside your zip file, there should be **only** one file named: **game.py**. Do not submit a Jupyter notebook .ipynb file.

Be sure to **remove all debugging output** before submission. Failure to remove debugging output will be **penalized (10pts)**.

Make sure to test your submission on the **CS Linux** machines!

This assignment is due on **December 2nd at 11:00 AM**. It is preferable to first submit a version well before the deadline (at least one hour before) and check the content/format of the submission to make sure it's the right version. Then, later update the submission until the deadline if needed.

HW9 Rubric

Criteria	Ratings				Pts
AI follows all rules of Teeko2, both during and after the drop phase.	**25 pts** **Full Marks** AI only makes legal moves during all game phases.	**15 pts** **Partial Marks** AI makes illegal moves in the drop phase.	**10 pts** **Partial Marks** AI makes illegal moves in the continued gameplay phase.	**0 pts** **No Marks** AI makes illegal moves in both the phases.	25 pts
AI correctly returns move without modifying state in make_move	**10 pts** **Full Marks** make_move does not modify the state of its Teeko2Player instance.	**5 pts** **Partial Marks** AI modifies the state in either drop or continued gameplay phases.	**0 pts** **No Marks** make_move does modify the state of its Teeko2Player instance.		10 pts
AI correctly returns move in make_move which is formatted in both the drop phase and continued gameplay	**10 pts** **Full Marks** make_move returns the move in a correct format in both drop and continued gameplay phases	**5 pts** **Partial Marks** AI returns an incorrect move format in either of the phases.	**0 pts** **No Marks** make_move returns a move in incorrect format for both the phases.		10 pts
AI selects a move in 5 or fewer seconds	**25 pts** **Full Marks** AI selects a move in 5 or fewer seconds.	**13 pts** **Partial Marks** AI selects a move in greater than 5 seconds and lesser than 10 seconds.	**0 pts** **No Marks** AI takes more than 10 seconds to select a move.		25 pts
AI wins at least 2 out of 3 games against a random player.	**30 pts** **Full Marks** AI wins at least 2 out of 3 games against a random player.	**15 pts** **Partial Marks** AI wins 1 out of 3 games against a random player.	**0 pts** **No Marks** AI wins no games against a random player.		30 pts
					Total Points: 100

HW10: Reinforcement Learning

Due Dec 14, 2021 by 10:59am **Points** 100 **Submitting** a file upload **File Types** zip **Available** Dec 1, 2021 at 11:59pm - Dec 14, 2021 at 11:15am

This assignment was locked Dec 14, 2021 at 11:15am.

Getting started

The starter code is available here: starter.zip

You can create and activate your virtual environment with the following commands:

```
python3 -m venv /path/to/new/virtual/environment
```

```
source /path/to/new/virtual/environment/bin/activate
```

once you have sourced your environment, you can run the following commands to install the necessary dependencies:

```
pip install --upgrade pip
```

```
pip install torch==1.8.0+cpu torchvision==0.9.0+cpu torchaudio==0.8.0 -f https://download.pytorch.org/whl/torch_stable.html
```

```
pip install gym==0.17.1 numpy==1.19.5
```

You should now have a virtual environment that is fully compatible with the skeleton code. You should set up this virtual environment on an instructional machine (CSL machines) to do your final testing. More detail about python virtual environments is available in the assignment 6 prompt: HW6: Neural Networks .

Q-Learning

For the Q-learning and SARSA portion of HW10, we will be using the environment FrozenLake-v0 from OpenAI gym. This is a discrete environment where the agent can move in the cardinal directions (up, down, left, right), but is not guaranteed to move in the direction it chooses (more in the FrozenLake-v0 doc). The agent gets a reward of 1 when it reaches the tile marked G, and a reward of 0 in all other settings. You can read more about FrozenLake-v0 here: https://gym.openai.com/envs/FrozenLake-v0/

You will not need to change any code outside of the area marked TODO, but you are free to change the hyper-parameters if you want to (but keep the random seed unchanged). The update rule for Q-learning is:

$$Q(s,a) = \begin{cases} Q(s,a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)) & \text{if !done} \\ Q(s,a) + \alpha(r - Q(s,a)) & \text{if done} \end{cases}$$ for each sampled $(s, a, r, s', \text{done})$ tuple

In this equation, alpha is the learning rate hyper-parameter, and gamma is the discount factor hyper-parameter.

The agent should act according to an epsilon-greedy policy as defined in the Reinforcement Learning 2 slides.

Epsilon-greedy policy: link

Pseudo Code

```
p = random()

if p < ε:
  pull random action
else:
  pull current-best action
```

HINT: tests.py is worth looking at to gain an understanding of how to use the OpenAI gym env.

For this section, you should submit the files Q_learning.py and Q_TABLE.pkl. The Q_TABLE.pkl will be generated automatically when you run your Q_learning.py.

A sample run of *python Q_learning.py*

```
LAST 100 EPISODE AVERAGE REWARD: 0.01
EPSILON: 0.903687356965959
LAST 100 EPISODE AVERAGE REWARD: 0.0
EPSILON: 0.817630186649157
LAST 100 EPISODE AVERAGE REWARD: 0.03
EPSILON: 0.739966025123943E
E[LAST 100 EPISODE AVERAGE REWARD: 0.02
EPSILON: 0.669515720180730
LAST 100 EPISODE AVERAGE REWARD: 0.03
EPSILON: 0.605772659163237
LAST 100 EPISODE AVERAGE REWARD: 0.04
EPSILON: 0.548089260578011
LAST 100 EPISODE AVERAGE REWARD: 0.03
EPSILON: 0.495915802017667E
LAST 100 EPISODE AVERAGE REWARD: 0.02
EPSILON: 0.448699994614647
LAST 100 EPISODE AVERAGE REWARD: 0.04
EPSILON: 0.405980235926587
LAST 100 EPISODE AVERAGE REWARD: 0.12
EPSILON: 0.367277225461537
LAST 100 EPISODE AVERAGE REWARD: 0.12
EPSILON: 0.332353442554527
LAST 100 EPISODE AVERAGE REWARD: 0.12
EPSILON: 0.300712415664385E
E[LAST 100 EPISODE AVERAGE REWARD: 0.09
EPSILON: 0.272082222336578
LAST 100 EPISODE AVERAGE REWARD: 0.16
EPSILON: 0.246178678932771
```

The first few print outputs are shown above. Two things are worth pointing out:

1. The value of EPISODE AVERAGE REWARD is obtained from the deque variable *episode_reward_record*. So you need to update this variable in your code to get appending the **accumulated** reward obtained in this episode into the deque.

2. EPSILON is decreasing, which means you need to update (decay) the hyperparameter variable EPSILON in your code (e.g. using EPSILON_DECAY).

Note: matching the sample output above is not guaranteed for the correctness of your program, please use the tests.py to test your code.

SARSA

SARSA is very similar to Q-learning, although it differs in the fact that it is on-policy. The name stands for "State, Action, Reward, State, Action," and refers to the tuples that are used by the update rule. You will not need to change any code outside of the area marked TODO, although, you are again welcome to change any hyperparameters you want.

The update rules for SARSA is:
$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$ always for each sampled $(s, a, r, s', a', \text{done})$ tuple
$Q(s',a') = Q(s',a') + \alpha(r' - Q(s',a'))$ if done

Similarly to Q-learning, the agent should act according to an epsilon-greedy policy as defined in the Reinforcement Learning 2 slides.

HINT: tests.py is worth looking at to gain an understanding of how to use the OpenAI gym env.

HINT: make sure that the SARSA update is performed for the last action-reward pair for each episode, because for FrozenLake-v0 this is the only possible time for a non-zero reward to be received. For the last State action pair, the expected cumulative reward is equal to the reward for that state and action, since there will be no further rewards.

For this section, you should submit the files SARSA.py and SARSA_Q_TABLE.pkl

A sample run of *python SARSA.py*

```
LAST 100 EPISODE AVERAGE REWARD: 0.01
EPSILON: 0.903687356965959
LAST 100 EPISODE AVERAGE REWARD: 0.0
EPSILON: 0.817630186649157
LAST 100 EPISODE AVERAGE REWARD: 0.03
EPSILON: 0.739966025123943E
LAST 100 EPISODE AVERAGE REWARD: 0.01
EPSILON: 0.669515720180730
LAST 100 EPISODE AVERAGE REWARD: 0.05
EPSILON: 0.605772659163237
LAST 100 EPISODE AVERAGE REWARD: 0.02
EPSILON: 0.455915802017667E
LAST 100 EPISODE AVERAGE REWARD: 0.03
EPSILON: 0.548089260578011
LAST 100 EPISODE AVERAGE REWARD: 0.08
EPSILON: 0.495980235926587
LAST 100 EPISODE AVERAGE REWARD: 0.11
EPSILON: 0.367327722346192S7
LAST 100 EPISODE AVERAGE REWARD: 0.1
```

DQN (Extra Credit)

Deep Q-learning has produced some exciting results in the past 5 years. First introduced in 2015 in Mnih et al., DQN allows the application of Q-learning to domains with continuous observation spaces.

For the experience replay buffer, you should add tuples of the form $(s, a, r, s', \text{done})$. The sample_minibatch function should return a list containing a random sample of MINI_BATCH_SIZE tuples from the experience replay buffer

For the first TODO in the main while loop, you should use the current policy_net policy to determine the best action to take then add the resulting $(s, a, r, s', \text{done})$ tuple you encounter from the environment to the experience replay buffer.

HINT: remember to use "with torch.no_grad():" when you do not need to calculate gradients.

HINT: Mnih et al. has the full pseudo-code, and is useful to look at for reference when implementing DQN.

For the second TODO, you will need to sample MINI_BATCH_SIZE tuples from the experience replay buffer, then calculate the appropriate estimates using the policy_net, and y values using the observed rewards, second states, and the target_policy_net.

OpenAI gym Environment

You will need to use several OpenAI gym functions in order to operate your gym environment for reinforcement learning. As stated in a previous hint, tests.py has a lot of the function calls you need. Several important functions are as follows:

env.step(action)
Given that the environment is in state s, step takes an integer specifying the chosen action, and returns a tuple of the form $(s', r, \text{done}, \text{info})$. 'Done' specifies whether or not s' is the final state for that particular episode, and 'info' is unused in this assignment.

env.reset()

Resets the environment to its initial state, and returns that state.

env.action_space.sample()
Samples an integer corresponding to a random choice of action in the environment's action space.

env.action_space.n

In the setting of the environments we will be working with for these assignments, this is an integer corresponding to the number of possible actions in the environment's action space.

You can read more about OpenAI gym here: https://gym.openai.com/docs/ .

Submission Format

You can test your learned policies, by calling python3 ./tests.py. Make sure to test your saved Q-tables using tests.py on the instructional machines with a virtual environment set up as specified above. This is the same program, which we will be using to test your Q-tables and Q-network, so you will have a good idea about how many points you will receive for the automated tests portion of the grade. Your submission should contain the following files:

Required: Q_learning.py, Q_TABLE.pkl, SARSA.py, SARSA_Q_TABLE.pkl
Optional: DQN.py, DQN.mdl, DQN_DATA.pkl

Please submit these files in a zipped folder title <yournetid>.zip , where 'yournetid' is your net ID. Please make sure that there is not a folder inside the zipped folder, and that the submitted files are at the top level of the zipped folder.

The assignment is due December 14th at 10:59 am central time. We are not accepting late submissions for this assignment. Regrades will only be accepted if they are due to an error in tests.py.

Grading

- 60 points (out of 100) will be given if you passed the tests.py (by running python tests.py after you generate the pkls). You can ensure that by testing on your own.

- 40 points will be determined via manual inspection of the code. We will inspect whether you implement the learning algorithms correctly.

- 10 points (extra credits) will be rewarded if your DQN passes the tests.py.

HW10: Reinforcement Learning

Criteria	Ratings			Pts
Q-Learning: Trained Q table 100 episode average score	**30 pts** **Full Marks** 100 episode average >= 0.6	**15 pts** **Partial marks** 100 episode average >= 0.4	**0 pts** **No Marks** 100 episode average <0.4	30 pts
Q-Learning: Implementation correctness	**20 pts** **Full Marks** Correct implementation of Q-learning	**10 pts** **Partial marks** Partially correct implementation of Q-learning	**0 pts** **No Marks** Incorrect implementation of Q-learning	20 pts
SARSA: Trained Q table 100 episode average score	**30 pts** **Full Marks** 100 episode average >= 0.6	**15 pts** **Partial marks** 100 episode average >= 0.4	**0 pts** **No Marks** 100 episode average < 0.4	30 pts
SARSA: Implementation correctness	**20 pts** **Full Marks** Correct implementation of SARSA	**10 pts** **Partial marks** Partially correct implementation of SARSA	**0 pts** **No Marks** Incorrect implementation of SARSA	20 pts
DQN: Trained Q network 100 episode average score	**0 pts** **0 points extra credit**	**0 pts** **10 points extra credit** 100 episode average >= 70		0 pts
			Total Points: 100	