AI 1104: Programming for AI

Final Project

Due on: 06 May 2024, 10:59 AM IST.

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Read each question carefully, and answer all parts. Write well-documented codes with contextual variable names. Put the codes, plots, and calculations into one single folder, compress (.zip) this folder, and upload the .zip file. You should name the .zip file as `AI1104_final_project_<your_roll_number>.zip`.

If you are participating in a team, submit a .pdf file listing the detailed contributions of each of the team members. Submit one .zip file per team with the name as `AI1104_final_project_<roll_numbers_of_team_mates.zip`.

This project comprises the following three tasks, each of which pertains to a key technical concept in machine learning.

Contents

1 Training an Artificial Neural Network

In this task, you will learn how to train an artificial neural network (or simply neural network), a key ingredient in the implementation of many deep learning tasks in modern machine learning applications.

1.1 Dataset Preparation

We shall first prepare our dataset.

1. Using pandas, load the given dataset `data_Q1.csv`.
2. Using the `.to_numpy()` method, convert each column of the dataframe to a numpy array and store them in arrays named `training_x1`, `training_x2`, and `training_y` respectively. Let N denote the length of each array.
3. Create a numpy array named `training_x` using the following code snippet:

```
training_x = np.array([[x1,x2,1] for x1,x2 in zip(training_x1,training_x2)])
```

1.2 Neural Network Preliminaries

Our neural network will consist of one *input* layer with 2 nodes (excluding bias), one *hidden* layer with 3 nodes (excluding bias), and an *output* layer. Below is the model for our neural network.

$$X \xrightarrow{W^{(1)}} H \xrightarrow{\sigma} Z \xrightarrow{W^{(2)}} O \xrightarrow{\sigma} \widehat{y}.$$

Here:

- X is the input data (coming from `training_data`) of size $N \times 3$.
- $W^{(1)} = [W^{(1)}_{k,l} : k, l \in \{1, 2, 3\}]$ is a weight matrix of size 3×3, connecting the input layer and hidden layer.
- $W^{(2)} = [W^{(2)}_{k,1} : k \in \{1, 2, 3, 4\}]^\top$ is a weight matrix of size 4×1 between the hidden layer and the output layer.
- σ is the *sigmoid* activation function, defined by $\sigma(x) = \frac{1}{1+e^{-x}}$.
- $H = X \times W^{(1)}$ is a matrix of size $N \times 3$ obtained at the hidden layer, before the activation function σ is applied.
- Z is a matrix of size $N \times 4$ obtained at the hidden layer after the activation function is applied to each entry of H, followed by appending the bias column at the end.
- $O = Z \times W^{(2)}$ is a matrix of size $N \times 1$ obtained at the output layer before the activation function is applied.
- $\widehat{y} = \sigma(O)$ is a matrix of size $N \times 1$ containing the predicted values.

To measure the performance of our neural network, we will use the so-called mean-squared-error loss, defined as

$$\mathcal{L}(Y, \widehat{y}) \coloneqq \frac{1}{2N} \sum_{i=1}^{N} (Y_i - \widehat{y}_i)^2,$$

where N is the number of data points, Y_i represents the true value for the i^{th} data point (coming from the array `training_y`), and $\widehat{y}_i$ represents the predicted value for the i^{th} data point.

1.3 Definitions and Initialisations

1. Add the following line to your code:

   ```
   np.random.seed(123)
   ```

   This will lead to reproducible variable values when working with random data.

2. Define a function called `sigmoid` which takes one single input x and returns $\frac{1}{1+e^{-x}}$. You will be calling this function on each entry of a matrix.

3. Define a function called `grad_sigmoid` which takes a single input x and returns the derivative of $\sigma(x)$. Again, you will be calling this function on each entry of a matrix.

4. Initialisation of weights matrices $W^{(1)}$ and $W^{(2)}$:

   (a) Given a random floating point number $\epsilon \in [0, 1]$, we can generate a random floating point in $[a, b]$ as follows: $a + \epsilon(b - a)$.

   (b) The weight matrix $W^{(1)}$ is represented by `weights1`. Initialize this 3×3 matrix with random floating point numbers lying between $[-1, 1]$.

   (c) The weight matrix $W^{(2)}$ is represented by `weights2`. Initialize this 4×1 matrix with random floating point numbers lying between $[-1, 1]$.

5. The learning rate, say γ, is a hyperparameter which controls the size of the steps you take while descending along the gradient. Initialize the learning rate to 0.05.

6. An epoch is one complete pass of the data through the network. Initialize `max_epochs` to 100.

7. Create an empty array to store the training errors after each epoch. Name the array `training_error`.

1.4 Training the Neural Network

Repeat the following over `max_epochs` many epochs.

1. **Forward pass.**

   (a) Initialise X to `training_x`.

   (b) Set H as the matrix multiplication of X and $W^{(1)}$.

   (c) Set Z as $\sigma(H)$, where $\sigma(H)$ denotes the matrix obtained by applying the sigmoid function on each element of H.

   (d) Append an all-ones column to Z. Store the result in Z again.

   (e) Set O as the product of Z and $W^{(2)}$.

   (f) Set $\widehat{y}$ as $\sigma(O)$.

   (g) Compute the loss between the true values in `training_y` and the predicted values in $\widehat{y}$. Append the loss to the array `training_error`.

2. **Back propagation.**

   (a) Calculating the gradients for `weights2`.

      i. Calculate the partial derivative of the error function with respect to $W^{(2)}_{k,1}$ as follows:

      $$\frac{\partial \mathcal{L}}{\partial W^{(2)}_{k,1}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \widehat{y}_i} \times \frac{\partial \widehat{y}_i}{\partial O_i} \times \frac{\partial O_i}{\partial W^{(2)}_{k,1}}, \quad k \in \{1, 2, 3, 4\}. \tag{1}$$

      Attach your calculations (in a .pdf file).

      ii. Implement the gradient you calculated and store it in a matrix `gradient2` which will have the same shape as `weights2`.

   (b) Calculating the gradients for `weights1`:

i. Calculate the partial derivative of the loss function with respect to $W_{k,l}^{(1)}$ as follows:

$$\frac{\partial \mathcal{L}}{\partial W_{k,l}^{(1)}} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}}{\partial \widehat{y}_i} \times \frac{\partial \widehat{y}_i}{\partial O_i} \times \frac{\partial O_i}{\partial Z_{i,l}} \times \frac{\partial Z_{i,l}}{\partial H_{i,l}} \times \frac{\partial H_{i,l}}{W_{k,l}^{(1)}}, \quad k, l \in \{1, 2, 3\}. \tag{2}$$

Attach your calculations (in a .pdf file).
You may use the below snippet of code for calculating the gradients in (2):

```
temp = -(training_y - y_hat) * grad_sigmoid(O) * grad_sigmoid(H)
temp = temp @ np.diag(weights2[:-1].reshape(-1))
gradient2 = training_x.T @ temp
gradient2 /= len(training_y)
```

Following the above snippet, generate your own snippet of code for implementing the gradients in (1).

ii. Implement the gradient you calculated and store it in a matrix `gradient1` which will have the same shape as `weights1`.

(c) Gradient descent:

i. Update each entry of `weights2` matrix (i.e., the weights matrix $W^{(2)}$) using the rule

$$W_{k,1}^{(2)} \leftarrow W_{k,1}^{(2)} - \gamma \frac{\partial \mathcal{L}}{\partial W_{k,1}^{(2)}}, \quad k \in \{1, 2, 3, 4\}.$$

ii. Update each entry of `weights1` matrix (i.e., the weights matrix $W^{(1)}$) using the rule

$$W_{k,l}^{(1)} \leftarrow W_{k,l}^{(1)} - \gamma \frac{\partial \mathcal{L}}{\partial W_{k,l}^{(1)}}, \quad k, l \in \{1, 2, 3\}.$$

In both of the above equations, γ denotes the learning rate parameter.

1.5 Plotting the Training Loss

Plot the training loss on the y-axis and the epochs on the x-axis. Label the axes and provide a title to the plot.

2 K-Means Clustering

For this task, we will generate a synthetic dataset, and implement a popular clustering technique called *K-means clustering* to cluster the points in the dataset.

2.1 Dataset Generation

1. We will generate two semi-circles of radius 1 each. One of the semi-circles faces downwards, centered at $(0,0)$, and the other faces upward, centered at $(1,0)$.
2. Recall that the parameterized coordinates of any point on the circumference of a circle with center (a,b) and radius r are of the form $(a + r \cos\theta, b + r \sin\theta)$ for some $\theta \in [0, 2\pi]$.
3. Using the functions `np.linspace` or `np.arange`, generate an array `theta` which comprises 500 equidistant points between 0 and π.
4. Add the following line of code: `np.random.seed(123)`. A seed helps to reproduce results, especially if there is same randomness present in the code.
5. The coordinates of any point on the circumference of the first semicircle are of the form $(\cos\theta, \sin\theta)$. Store the x-coordinates in an array called `semicircle1_x`. Generate an array of noise using `np.random.normal` with mean 0 and standard deviation 0.1 and add it to `semicircle1_x`. Similarly, store the y-coordinates in an array called `semicircle1_y`. Generate an array of noise using `np.random.normal` with mean 0 and standard deviation 0.1 and add it to `semicircle1_y`.
6. The coordinates of any point on the circumference of the second semicircle are of the form $(1+\cos\theta, -\sin\theta)$. Repeat the previous step to store the points for the second semicircle and store the x- and y-coordinates in `semicircle2_x` and `semicircle2_y` respectively.
7. Plot the points for the first and the second semicircle on the same graph using the same color. We use the same color because these points comprise the original dataset, and given the original dataset, we do not yet know the clusters. Our aim now is to use a clustering algorithm to identify clusters within the dataset.
8. Store all the data points in one single array as tuples for convenience. Tuples are a hashable data type, whereas lists are not. This becomes important when we use dictionaries for clustering. Create an array `data_points`, and you can use the following code-sketch for adding all the data points as tuples to it:

```
d1 = [(x,y) for x,y in zip(s1_x,s1_y)]

d2 = [(x,y) for x,y in zip(s2_x,s2_y)]

        data_points = d1 + d2
```

2.2 K-Means Clustering

K-Means clustering is an iterative algorithm comprising three parts: initialization, iteration, and termination. The idea behind the algorithm is that each cluster has a representative centroid. The centroids are initialised randomly, and then updated sequentially with each iteration over the data. This process continues until a certain stopping criterion is met. Several stopping criteria such as checking for the number of points updated during an iteration, checking for the average distance of points from centroids, etc., are employed in practice.

1. Initialisation:
   Note that all of our data lies in the region $[-2,3] \times [-2,2]$. Pick two points uniformly at random from this region, and set these as the initial values for the centroids. Let the first centroid be given by `C_1` while the second centroid be given by `C_2`.
2. Iteration:
   (a) Initialize a dictionary called `mappings`to keep track of each point and the cluster associated with it. Your dictionary `mappings`should be as follows (only a template):

$$\{(x_1, y_1) : 1, (x_2, y_2) : 2, \ldots (x_{1000}, y_{1000}) : 2\}.$$

(b) Initialize a variable called `num_points_updated` to 0. This will keep track of the number of points whose cluster was updated during an iteration. Also, initialize `max_iters` to $10,000$ and `current_iters` to 0. Your algorithm should stop if no point gets updated during an iteration, or if the maximum number of iterations is reached.

(c) Your algorithm should execute the following steps until stoppage:

i. Assign each point in the dictionary to one of the two clusters. For this, check the point's Euclidean distance from each of the two centroids, and assign the cluster corresponding to the nearest centroid. Keep track of the number of points whose clusters were updated.

ii. Update the centroid values. For this, compute the average of the coordinates of all the data points mapped to a particular cluster. For instance, if A is the set of all points that have been mapped to cluster 1, then

$$\texttt{C\_1} = \frac{1}{|A|} \sum_{\vec{a} \in A} \vec{a},$$

where $\vec{a}$ denotes the coordinates of a generic point in A, and $|A|$ denotes the number of points in A.

3. Plot the clusters you have obtained with different colors on the same plot.

Note: You may observe that K-means clustering may not give the expected clustering of data points. Thus, we have other methods such as DBSCAN which also takes into account the densities of the points, i.e number of points in the neighbourhood of a point to assign clusters.

3 Classification using the k-Nearest Neighbors Technique

In this task, you will use the popular k-nearest neighbors method to learn the classes in a labeled dataset and predict the class of a new data point. Classification is an ubiquitous *supervised learning* technique that finds use in several day-to-day machine learning applications.

3.1 The Abalone Dataset

We will work with the abalone dataset. Add the following snippet of code to load the data into a pandas dataframe.

```
import pandas as pd
url = ``https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data''
abalone = pd.read_csv(url, header=None)
```

The age of an abalone can be found by cutting its shell and counting the number of rings on the shell. In the abalone dataset, you can find the age measurements of a large number of abalones along with a lot of other physical measurements. The goal of this task is to learn a model that can predict the age of an abalone as a function of its physical measurements.

3.2 Data Pre-Processing and Cleaning

Notice that we have removed the column names when importing the data from the above url. Append the column names in the below array to the `abalone` dataframe created earlier.

```
column_names = [“Sex”, “Length”, “Diameter”, “Height”, “Whole weight”,
                “Shucked weight”, “Viscera weight”, “Shell weight”, “Rings”]
```

One of the columns in the dataset is `Sex` of the abalone, which does not directly influence its age. Remove this column from the dataset.

3.3 Training and Test Data

Now, we shall use Python's `scikit-learn` to split our dataset into two parts. One part, called the training dataset, will be used to learn the classes in the data, and the other part, called the test dataset, will be used to evaluate the performance of the learning task.
Use the below snippet of code to create training and test datasets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
                                       X, y, test_size=0.3, random_state=333 )
```

In the above snippet, you should substitute `y` with the “`Rings`” column and `X` with the remaining columns in `abalone`. Use the `.values()` method to remove the column names before inputting the data values to `X` and `y` in the above snippet.
Note: The above snippet creates a test dataset with 30% of the total data points and a training dataset with 70% of the total data points.

3.4 Implementing the k-Nearest Neighbors Algorithm

Consider a new abalone with physical measurements as given in Table 1. Our task is to predict its age using the k-nearest neighbors technique.

1. Using NumPy's `linalg.norm()` method, compute the Euclidean distance between the each row of `X_train` and that of the new data point (viewed as a row with same column names as `X_train`). Let the distances be stored in a variable called `distances`.

2. Sort the values in `distances` in increasing order of their values, and get the indices of the smallest 3 values using the `.argsort()` method. These indices correspond to the indices of the 3 nearest neighbors in the abalone dataset to the new data point.

Variable	Value
Length	0.569552
Diameter	0.446407
Height	0.154437
Whole weight	1.016849
Shucked weight	0.439051
Viscera weight	0.222526
Shell weight	0.291208

Table 1: Physical measurements for a new abalone.

3. Corresponding to the indices of the 3 nearest neighbors obtained above, extract the number of rings (ages) from `y_train`. Among the 3 ages thus obtained, pick the age which appears most frequently (i.e., the "mode"). If multiple ages appear equal number of times, pick one among them uniformly at random. The predicted age is the age we will be assigning to new abalone data point. Report the predicted age.

4. Treating each row in `X_test` as a new abalone data point, repeat steps 1-3 above to generate a prediction for the age. Store the predicted ages in an array called `predicted_ages_test`. Compute and report the *mean squared error* (MSE) in prediction using the formula

$$\text{MSE} = \frac{1}{M}\sum_{j=0}^{M-1}\left(\texttt{predicted\_ages\_test}[j] - \texttt{y\_test}[j]\right)^2, \tag{3}$$

   where M is the length of `y_test` column.

3.5 Tuning k to Achieve Optimal Performance

Vary k from 1 to 50, and compute the MSE value for each k. Store the MSE values in an array named `mse_values`. Generate a plot of k values on the x-axis and `mse_values` on the y-axis. From the plot, report an optimal value of k at which MSE is the smallest. If there are multiple optimal values for k, you may pick any one at random. Also report the MSE corresponding to this optimal value of k.