Final Project

Parsa Samadnejad, Amir Mirzaei

In this project you practice concepts such as Principal Component Analysis, Singular Value Decomposition, differentiation, and gradient descent through the implementation of neural networks and a dimensionality reduction technique from scratch. It comprises four tasks described below.

1. Logistic Regression

In this task, you should implement a logistic regression model for a binary (two-class) classification problem. You cannot use machine learning libraries such as Scikit-learn, Tensorflow, Keras, or PyTorch.

Having a set of feature points $X_1, X_2, \ldots, X_N \in R^d$, and a set of labels $y_1, y_2, \ldots, y_N \in \{0, 1\}$, a logistic regression model is a mapping $\Phi: R^d \to (0, 1)$, defined as

$$\Phi(X) = \sigma(w^T X + b),$$

where $w \in R^d$ and $b \in R$, are the model parameters and σ is the sigmoid function

$$\sigma(x) = 1/(1 + e^{-x}).$$

To obtain the full score of this task, you must follow these steps:

1.0 Look at the data

Two data files are provided to you: **`data2D.npz`** contains 2D data and **`data5d.npz`** contains 5D data. Open the data files, look at the data and see what they look like.

```
raw_data = np.load('data5d.npz')
X = raw_data['X']
y = raw_data['y']
```

- Plot the 2D data points for **`data2D.npz`**. Use different colors for class 0 and class 1.
- In the same manner, plot the first two coordinates of **`X`** in **`data5D.npz`** (**`X[:,1]`** and **`X[:,2]`**).

1.1 Calculate the Gradients

Before implementing *gradient descent*, you need to compute the gradient of a cost function with respect to the parameters $w \in R^d$ and $b \in R$. As a cost function you may either use the sum of squared errors

$$C(w,b) = \sum_{i=1}^{N} (\Phi(X_i) - y_i)^2,$$

or the cross-entropy cost

$$C(w,b) = -\sum y_i log(\Phi(X_i)) + (1-y_i) log(1-\Phi(X_i)).$$

Notice that as $y_i \in \{0,1\}$, each term in the cross-entropy loss is either equal to $log(\Phi(X_i))$ or $log(1-\Phi(X_i))$ depending on whether $y_i = 1$, or $y_i = 0$.

Calculate the gradient of $C(w,b)$ with respect to w and b. Write down your derivations on a piece of paper or using LaTeX and explain your solution for the TA at the time of your presentation.

1.2 Implement the Model and its gradients

Implement the model $\Phi(X)$, the cost function $C(w,b)$ and the gradients of $C(w,b)$ with respect to w and b (the formulae you obtained in section 1.1).

```
def Phi(X):
    ....

def C(w,b, data):
    ....

def compute_dC_dw(w,b, data):
    ....

def compute_dC_db(w,b, data):
    ....
```

1.3 Numeric Gradient Checking

First, write python functions that compute the gradients *numerically* as discussed in the class.

```
def compute_dC_dw_numeric(w,b, data):
    ....

def compute_dC_db_numeric(w,b, data):
    ....
```

Then compare the actual gradient vectors

```
dC_dw = compute_dC_dw(w,b, data)
dC_db = compute_dC_db(w,b, data)
```

and the numeric gradient vectors

```
dC_dw_n = compute_dC_dw_numeric(w,b, data)
dC_db_n = compute_dC_db_numeric(w,b, data)
```

To do this you may check the length of the difference between the actual and numeric gradients

```
np.linalg.norm(dC_dw-dC_dw_n) # absolute error
np.linalg.norm(dC_dw-dC_dw_n)/np.linalg.norm(dC_dw) # relative error
```

Test for different random values of w and b.

```
raw_data = np.load('data5d.npz')
X = raw_data['X']
y = raw_data['y']
data = (X,y)
```

Your functions must work for any dimension d. Test on both **data2d.npz** and **data5d.npz**.

1.4 Implement gradient descent

Start from a random w and b, choose a learning rate λ, and keep moving in the direction of negative gradient.

```
loop:
   w = w - lambda * dC_dW
   b = b - lambda * dC_db
```

- Stop the loop if the difference between two consecutive w, b is smaller than some threshold, or when the length of the gradient vectors are below some threshold.
- After every iteration print the cost $C(w, b)$. Ideally, the cost must go down after every iteration.
- Test different values of `lambda`. What happens if you choose a very large learning rate? What if you choose it too small? Try to find an appropriate learning rate.

1.5 Compute the training error

After each print the training error. First, compute $\Phi(X_i)$ for each X_i. The i-th sample is classified as $\hat{y}_i = 0$ if $\Phi(X_i) < 0.5$ and $\hat{y}_i = 1$ otherwise. The sample is misclassified if $\hat{y}_i \neq y_i$. The training error is the ratio between the number of misclassified samples to the number of all samples (**N**).

1.6 Visualize

For the 2D case plot all the samples on a 2D plane. Use different colors for different classes (e.g. blue for when $y_i = 0$ and red for when $y_i = 1$). Also, identify misclassified data (e.g. use hollow (unfilled) markers ○ for misclassified points and filled markers ● for correctly classified points). Then, draw the line that is supposed to separate samples from the two classes using w

and b. Notice that for the 2D case, we have $w = [w_0, w_1]^T$ and $X = [x, y]^T$. The separating line is then defined as $w_0 x + w_1 y + b = 0$. Plot the points and the line after every iteration of the gradient descent algorithm. Animate your results to show how the separating line and the classification result change throughout iterations.

2. Logistic Regression Using TensorFlow

Here, you must implement the task in section 1 using TensorFlow. Similarly, you are not allowed to use pre-defined NN layers provided by TensorFlow. The only difference is that you compute the gradients using the auto-differentiation capability of TensorFlow. To do this, all you have to do is define the parameters w, b as tensorflow "Variables" (**`tf.Variable`**) instead of numpy arrays, and use TensorFlow functions (like **`tf.math.exp`**) instead of python or numpy function (such as **`math.exp`** or **`np.exp`**). Then you can simply compute the gradient using a gradient tape (**`tf.GradientTape`**). For more details read the article "Introduction to gradients and automatic differentiation" in the TensorFlow website: https://www.tensorflow.org/guide/autodiff

- To get started from TensorFlow start from here https://www.tensorflow.org/guide/basics.
- You cannot use **`tf.math.sigmoid`** for the sigmoid function. Implement sigmoid yourself using the **`tf.math.exp`**.

3. Multi-Layer Neural Networks

In this task, extend the previous task by implementing a multi-layer neural network equipped with proper non-linear activations. Moreover, you should implement a Softmax layer manually and apply it to the output layer of your neural network. For example, a 3-layer network for classifying inputs into C classes looks like

$$h_1 = a_1(W_1 X + b_1),$$
$$h_2 = a_2(W_2 h_1 + b_2),$$
$$y = softmax(W_3 h_2 + b_3),$$

where $X \in R^d$ is the input, $h_1 \in R^p$, $h_2 \in R^q$ represent the hidden layers, $y \in R^C$ is the output, $W_1 \in R^{p \times d}$, $W_2 \in R^{q \times p}$, $W_3 \in R^{C \times q}$, $b_1 \in R^p$, $b_2 \in R^q$, $b_3 \in R^C$ are the weights and biases (network parameters), and $a_1(.)$, $a_2(.)$ are activation functions. The above is equivalent to

$$\Phi(X) = softmax(W_3\ a_2(W_2\ a_1(W_1 X + b_1) + b_2) + b_3).$$

You do not need to implement backpropagation. Use the automatic differentiation feature of Tensorflow to compute the gradients with respect to weights and biases. Remember from the class that the gradient with respect to a matrix has the shape of that matrix.

For this task, use "Iris Data Set". On the UCI Repository page, you can observe the dataset's description and a link to download the "iris.data" which contain a comma-separated values (csv) file.

Note:

- You must implement the neural net model yourself. You cannot use the built-in TensorFlow/Keras models or layers.
- The final layer must be a softmax layer. You have to implement softmax yourself and cannot use `tf.nn.softmax`. But you may use `tf.math.exp` for implementing softmax. (You can use `tf.nn.softmax` just to verify your implementation.)
- You can use any common ***nonlinear*** activation functions (such as sigmoid, ReLU, leaky ReLU, etc.), but you must implement them yourself. Thus, you cannot use `tf.nn.sigmoid` or `tf.nn.relu`. Use basic functions like `tf.math.sigmoid` or `tf.math.maximum` to implement them.

4. Visualization using PCA

We want to visualize the final classification results **like what you did in section 1.6**. To do this you need to reduce the dimensionality of your data samples (feature vectors) to **2D**. Use Principal Component Analysis (PCA) for dimensionality reduction. Then plot the data points (correctly and incorrectly classified) just like what you did in section 1.6 (you do not need to draw the separating lines though).

- There is no need to animate the training process. Just plot the final results after training.
- You cannot use the PCA functionality of the machine learning libraries like Scikit-Learn or TensorFlow. You need to implement PCA yourself as you learned in class.
- You need to implement PCA using ***Singular Value Decomposition (SVD)*** as described in the class. If you use Eigenvalue Decomposition instead of SVD, **you only receive 70% of the score of this part.**

5. Bonus: Visualize using LDA

You can receive 20% extra points if you also perform dimensionality reduction using Linear Discriminant Analysis (LDA) besides PCA. You need to learn about the LDA yourself. Search the internet for a good tutorial.

- Notice that you need to use a multi-class LDA algorithm.
- You need to implement LDA yourself. You are not allowed to use the LDA implemented in machine learning libraries.
- At the time of your presentation, you need to explain the LDA algorithm and how the result differs from that of the PCA, and why.