Deep Learning - Homework ♯2

Gradient descent

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

Introduction

Consider the 3-dimensional graph below in the context of a cost function. Our goal is to move from the mountain in the top right corner (high cost) to the dark blue sea in the bottom left (low cost). The arrows represent the direction of steepest descent (negative gradient) from any given point–the direction that decreases the cost function as quickly as possible.

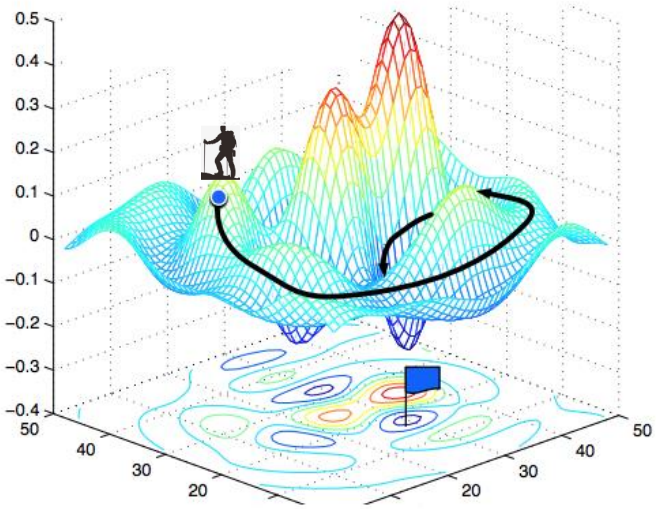
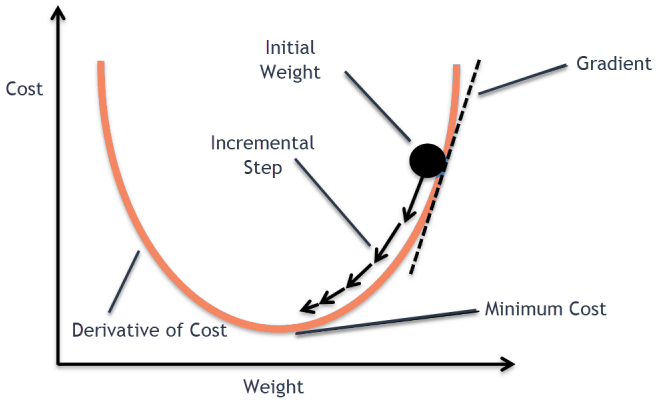


Figure 1: Examples for the steepest descent.

Starting at the top of the mountain, we take our first step downhill in the direction specified by the negative gradient. Next we recalculate the negative gradient (passing in the coordinates of our new point) and take another step in the direction it specifies. We continue this process iteratively until we get to the bottom of our graph, or to a point where we can no longer move downhill–a local minimum.

Learning rate

The size of these steps is called the learning rate. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing.



With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

Cost function

A Loss Functions tells us "how good" our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate.

Step-by-step

Now let's run gradient descent using our new cost function. There are two parameters in our cost function we can control: θ_1 (weight) and θ_0 (bias). Since we need to consider the impact each one has on the final prediction, we need to use partial derivatives. We calculate the partial derivatives of the cost function with respect to each parameter and store the results in a gradient.

Given the cost function:

$$f(\theta_1, \theta_0) = \frac{1}{N}\sum_{i=1}^{n}(y_i - (\theta_1 x_i + \theta_0))^2$$

The gradient can be calculated as:

$$f'(\theta_1, \theta_0) = \begin{bmatrix} \frac{df}{d\theta_1} \\ \frac{df}{d\theta_0} \end{bmatrix} = \begin{bmatrix} \frac{1}{N}\sum -2x_i(y_i - (\theta_1 x_i + \theta_0)) \\ \frac{1}{N}\sum -2(y_i - (\theta_1 x_i + \theta_0)) \end{bmatrix}$$

To solve for the gradient, we iterate through our data points using our new m and n values and compute the partial derivatives. This new gradient tells us the slope of our cost function at our current position (current parameter values) and the direction we should move to update our parameters. The size of our update is controlled by the learning rate.

With this observation in mind, one starts with a guess $[\theta_1^0, \theta_0^0]$ for a local minimum of $f(\theta_1^0, \theta_0^0)$, and considers the sequence $[\theta_1^0, \theta_0^0]$, $[\theta_1^1, \theta_0^1]$, $[\theta_1^2, \theta_0^2]$, $\cdots$ such that

$$\begin{bmatrix} \theta_1^{t+1} \\ \theta_0^{t+1} \end{bmatrix} = \begin{bmatrix} \theta_1^t \\ \theta_0^t \end{bmatrix} - \lambda \left. \begin{bmatrix} \frac{df}{d\theta_1} \\ \frac{df}{d\theta_0} \end{bmatrix} \right|_{(\theta_1,\theta_0)=(\theta_1^t,\theta_0^t)}$$

Programming

1 Linear regression with one variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities.

You would like to use this data to help you select which city to expand to next.

The file `ex1data1.txt` contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the first of a food truck in that city. A negative value for profit indicates a loss.

1.1 Plotting the Data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). (Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.)

Let's start by importing some libraries and examining the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import os
path = os.getcwd() + '\data\ex1data1.txt'
data = pd.read_csv(path, header=None, names=['Population', 'Profit'])
data.head()

data.describe()
```

Let's plot it to get a better idea of what the data looks like (Fig. 2).

```
data.plot(kind='scatter', x='Population', y='Profit', figsize=(12,8))
```

1.2 Gradient Descent

In this part, you will find the linear regression parameters θ to our dataset using gradient descent.

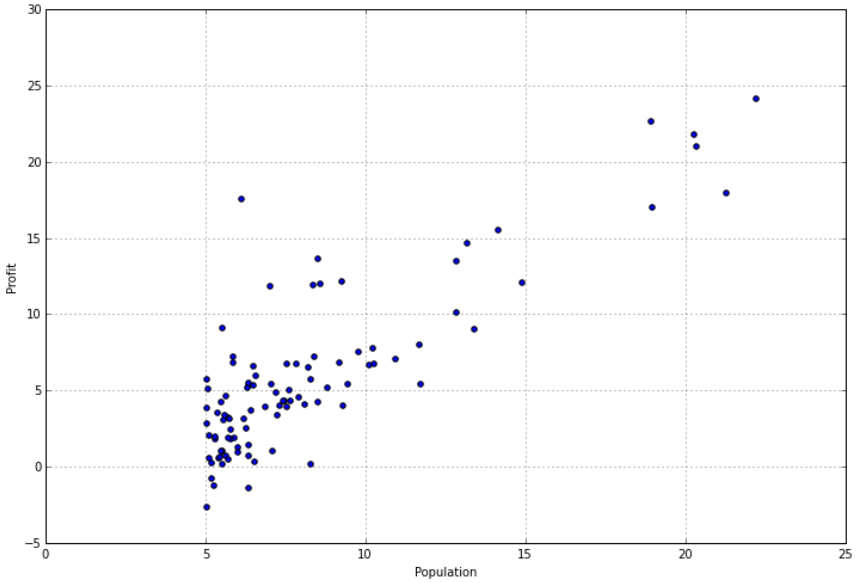


Figure 2: Scatter plot of training data.

1.2.1 Update Equations

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1.$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j).$$

With each step of gradient descent, your parameters θ_j come closer to the optimal values that will achieve the lowest cost $J(\theta)$.

1.2.2 Implementation

First we'll create a function to compute the cost of a given solution (characterized by the parameters theta).

```
def computeCost(X, y, theta):
    inner = np.power(((X * theta.T) - y), 2)
    return np.sum(inner) / (2 * len(X))
```

Let's add a column of ones to the training set so we can use a vectorized solution to computing the cost and gradients.

```
data.insert(0, 'Ones', 1)
```

Now let's do some variable initialization.

```
# set X (training data) and y (target variable)
cols = data.shape[1]
X = data.iloc[:,0:cols-1]
y = data.iloc[:,cols-1:cols]
```

Let's take a look to make sure X (training set) and y (target variable) look correct.

```
In: X.head()

Out:
Ones    Population
0   1   6.1101
1   1   5.5277
2   1   8.5186
3   1   7.0032
4   1   5.8598

In: y.head()

Out:
    Profit
0   17.5920
1   9.1302
2   13.6620
3   11.8540
4   6.8233
```

The cost function is expecting numpy matrices so we need to convert X and y before we can use them. We also need to initialize theta.

```
X = np.matrix(X.values)
y = np.matrix(y.values)
```

```
theta = np.matrix(np.array([0,0]))
```

Here's what theta looks like.

```
In : X.shape, theta.shape, y.shape

Out: ((97L, 2L), (1L, 2L), (97L, 1L))
```

Now let's compute the cost for our initial solution (0 values for theta).

```
In : computeCost(X, y, theta)

Out: 32.072733877455676
```

Now we need to define a function to perform gradient descent on the parameters theta using the update rules defined in the text.

```
def gradientDescent(X, y, theta, alpha, iters):
    temp = np.matrix(np.zeros(theta.shape))
    parameters = int(theta.ravel().shape[1])
    cost = np.zeros(iters)

    for i in range(iters):
        error = (X * theta.T) - y

        for j in range(parameters):
            term = np.multiply(error, X[:,j])
            temp[0,j] = theta[0,j] - ((alpha / len(X)) * np.sum(term))

        theta = temp
        cost[i] = computeCost(X, y, theta)

    return theta, cost
```

Initialize some additional variables - the learning rate alpha, and the number of iterations to perform.

```
alpha = 0.01
iters = 1000
```

Now let's run the gradient descent algorithm to fit our parameters theta to the training set.

```
g, cost = gradientDescent(X, y, theta, alpha, iters)
```

```
In: g

Out: matrix([[-3.24140214,  1.1272942 ]])
```

Finally we can compute the cost (error) of the trained model using our fitted parameters.

```
In: computeCost(X, y, g)

Out: 4.5159555030789118
```

Now let's plot the linear model along with the data to visually see how well it fits.

```
x = np.linspace(data.Population.min(), data.Population.max(), 100)
f = g[0, 0] + (g[0, 1] * x)

fig, ax = plt.subplots(figsize=(12,8))
ax.plot(x, f, 'r', label='Prediction')
ax.scatter(data.Population, data.Profit, label='Traning Data')
ax.legend(loc=2)
ax.set_xlabel('Population')
ax.set_ylabel('Profit')
ax.set_title('Predicted Profit vs. Population Size')
```

Since the gradient decent function also outputs a vector with the cost at each training iteration, we can plot that as well. Notice that the cost always decreases - this is an example of a convex optimization problem.

```
fig, ax = plt.subplots(figsize=(12,8))
ax.plot(np.arange(iters), cost, 'r')
ax.set_xlabel('Iterations')
ax.set_ylabel('Cost')
ax.set_title('Error vs. Training Epoch')
```

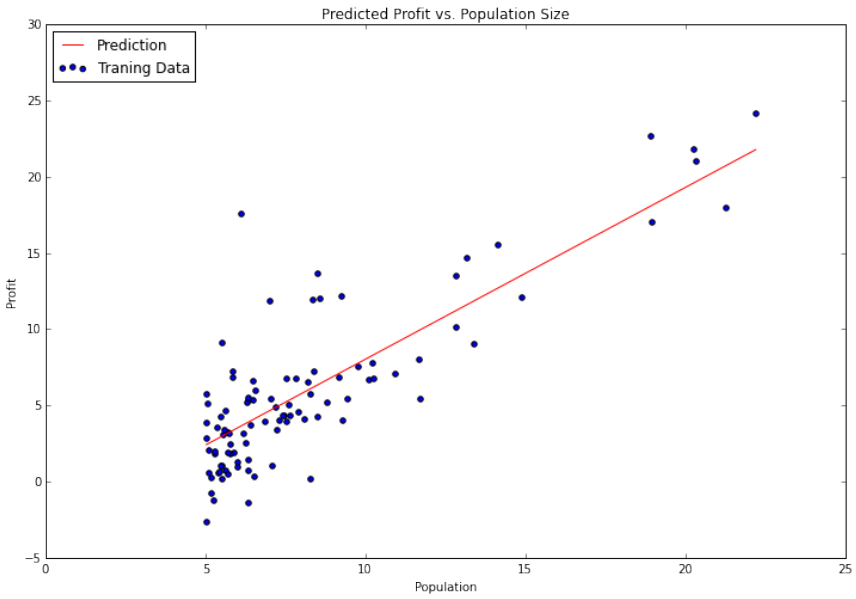


Figure 3: Training data with linear regression fit.

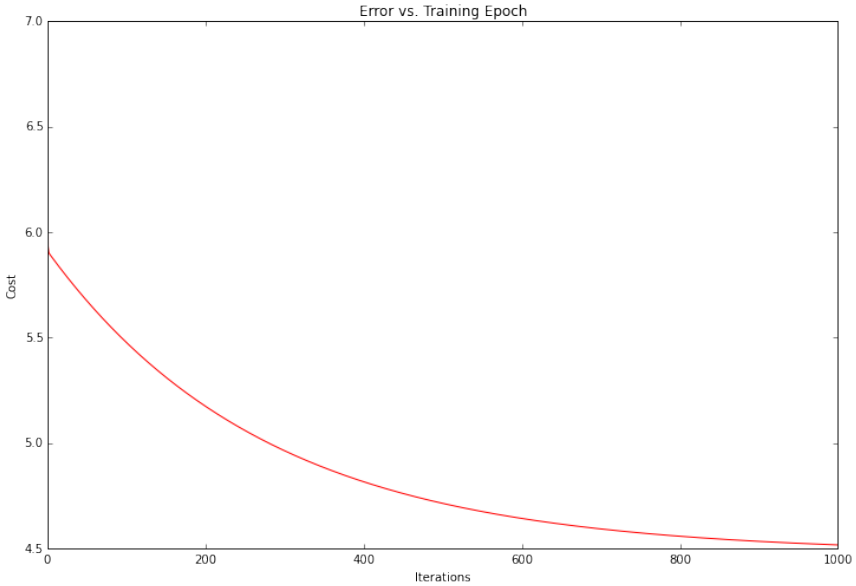


Figure 4: Convergence of gradient descent with an appropriate learning rate.

Instead of implementing these algorithms from scratch, we could also use scikit-learn's linear regression function. Let's apply scikit-learn's linear regressio algorithm to the data from part 1 and see what it comes up with.

```
from sklearn import linear_model
model = linear_model.LinearRegression()
model.fit(X, y)
```

Here's what the scikit-learn model's predictions look like.

```
x = np.array(X[:, 1].A1)
f = model.predict(X).flatten()

fig, ax = plt.subplots(figsize=(12,8))
ax.plot(x, f, 'r', label='Prediction')
ax.scatter(data.Population, data.Profit, label='Traning Data')
ax.legend(loc=2)
ax.set_xlabel('Population')
ax.set_ylabel('Profit')
ax.set_title('Predicted Profit vs. Population Size')
```

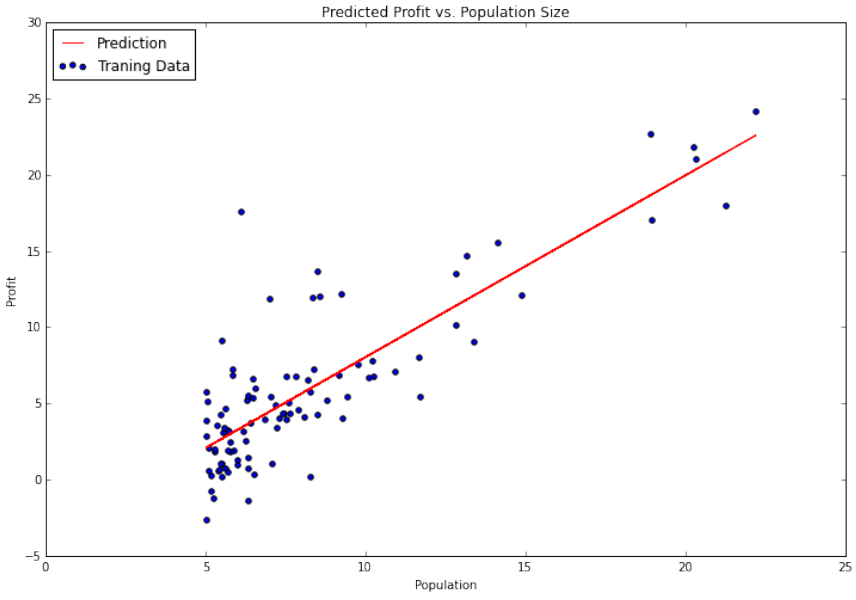


Figure 5: Training data with linear regression fit.