1 Linear Regression using Gradient Descent

In the previous lab session, you used the normal equations to solve the univariate and multivariate linear regression problems. In this exercise, you will investigate the use of gradient descent optimization algorithm for solving the linear regression problems.

1.1 Short theory

Gradient Descent (GD) is an optimization algorithm. It can be use to find a set of parameters that maximizes or minimizes a given function. Given a function defined by a set of parameters θ, GD starts with an initial value of θ and iteratively update θ toward a direction that reduces the value of the function. This update direction is the reverse of the one represented by the function's gradient.

In linear regression, we want to find a linear target function (also call a hypothesis function), h that map input features to target values. This function can be expressed by:

$$\hat{y}^i = h(x^i) = \theta^T x^i = \theta_0 + \sum_{j=1}^{m} \theta_j x_j^i, \tag{1}$$

with x^i and $\hat{y}^i$ the features and the predicted target value of the $i-$th sample, m the number of features (i.e., the dimension of the inputs) and θ_0 the bias.

The optimal value of θ by minimizing the following cost function:

$$\mathcal{J}_\theta = \frac{1}{2n} \sum_{i=1}^{n} \left(h\left(x^i\right) - y^i \right)^2, \tag{2}$$

with n the number of training samples. J is equal to the sum of square of the prediction errors, divided by $\frac{1}{2n}$. This cost function can also be written in the matrix form as follows:

$$\mathcal{J}_\theta = \frac{1}{2n} \left(\mathbf{X}\theta - \mathbf{y}\right)^T \left(\mathbf{X}\theta - \mathbf{y}\right)$$

where $\mathbf{X} \in \mathbb{R}^{n \times (m+1)}$ is the data matrix and $\mathbf{y} \in \mathbb{R}^{n \times 1}$ is the vector containing target values.

The gradient of the cost function $\mathcal{J}$ with respect to its parameters, θ, is:

$$\frac{\partial \mathcal{J}}{\partial \theta} = \frac{1}{n}\mathbf{X}^{\mathsf{T}}(\mathbf{X}\theta - \mathbf{y}). \tag{3}$$

This quantity is used by the GD algorithm to to iteratively update θ during the training phase as follows:

- Step 1: start with a random initialization of θ
- Step 2: update θ according to:

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \frac{\partial \mathcal{J}}{\partial \theta^{\text{old}}} \tag{4}$$

- Step 3: repeat Step 3 for a pre-defined number of times

1.2 Python Exercise

In this exercise, we will implement the Gradient Descent algorithm to learn a linear regression model. As in the previous session, we will use the Boston dataset from `sklearn`. The skeleton code for this exercise is provided in the `simple_gradient_descent.ipynb` notebook.

Step 1: Preprocessing the data. In this step, you will need to load the Boston dataset from `sklearn` and split it as you did in the previous lab session. Specifically, keep 80 percent of the samples to form the training set and the rest to form the test set. Then, scale all the features to a similar value range, by means of subtracting the mean and dividing the results by the standard deviations. Note that you should only use the training date to calculate the means and standard deviations of the features.

Step 2: Add intercept term and initialize parameters. In this step, you need to add the intercept term to the training and the test data. You can add the intercept term as the first column of your data matrices. After that, initialize θ using the normal distribution with $\mu = 0$ and $\sigma = 0.5$.

Step 3: Implement the gradient calculation and cost functions. In this step, you have to implement functions to calculate the cost $\mathcal{J}$ and its gradient with respect to θ.

Step 4: Verify that your gradient calculation is correct. It is important to make sure that your implementation of the gradient calculation is correct. To do so, you can estimate the gradient using numerical method and compare this estimation with the results of your implementation in Step 3.

The gradient estimation can be obtained following the definition of gradient, that is:

$$\frac{\partial \mathcal{J}_\theta}{\partial \theta_i} \approx \frac{\mathcal{J}(\theta_1, \theta_2, \theta_i + \epsilon, \cdots, \theta_m) - \mathcal{J}(\theta_1, \theta_2, \theta_i - \epsilon, \cdots, \theta_m)}{2\epsilon}, \tag{5}$$

with ϵ a small value.

Correct implementations of the two methods should result in gradients with very small sum of squared errors (around $\sim 10^{-18}$).

Step 5: Selecting a learning rate. In the GD algorithm, selecting a suitable learning rate α is highly important. Changing α can strongly affect the performance of the final model after training. In this step, we will do an experiment to see the effects of α on the GD algorithm.

Specifically, we experiment with difference values of α, between 0.001 and 0.3. With each value of α, we run the GD algorithm for 100 iteration. We record the values of the cost function $\mathcal{J}$ at each iteration plot them to see how $\mathcal{J}$ decreases over time.

Step 6: Make prediction Select a value of α that you think is the best. We it to learn θ using the GD algorithm. After that, use the θ that you find to make predictions.

2 Regularized Linear Regression using Gradient Descent

2.1 Short Theory

As we saw in the lectures, ℓ_2 regularization is often used to control overfitting when training a linear regression model. By applying this regularization technique, we learn θ that minimizes the following cost function:

$$\mathcal{J}_\theta = \frac{1}{2n}\sum_{i=1}^{n}\left(h\left(x_i\right) - y_i\right)^2 + \lambda\sum_{j=1}^{m}\theta_j^2, \tag{6}$$

with λ a hyperparameter controlling the effect of the regularization term.

Similar to the normal linear regression model, we can employ the GD algorithm to learn θ. The only difference is that we need to account for the regularization term when calculating the gradient. Specifically, in this case, the gradient of $\mathcal{J}$ with respect to θ is:

$$\frac{\partial\mathcal{J}}{\partial\theta} = \frac{1}{n}\mathbf{X}^\top(\mathbf{X}\theta - \mathbf{y}) + \lambda\theta. \tag{7}$$

2.2 Python Exercise

In this exercise, you will modify the functions for calculating the gradient and the cost to account for the regularization term. You will have a chance to monitor how the training process changes with different regularization coefficient (λ). The skeleton code is provided for you in the `gradient_descent_with_regularization` notebook. You can re-use parts of the code that you wrote for the previous exercise.

Step 1: Modify the cost and the gradient functions. In this step, you need to modify your implementations of the cost function and the gradient calculation. Do not forget to verify your implementation of the gradient calculation using the numerical method as done in the previous exercise.

Step 2: Train your model with different regularization coefficients. In this step, you need to train your model with different values of λ and plot

the cost on the training and testing sets in the same figure. By doing so, you will notice which coefficient is most appropriate for your model

Step 3: Select regularization coefficient and visualize your prediction. In this step, select the best regularization coefficient and visualize your prediction and the ground truth in the same graph. Notice when calculating the cost here, we set `lambda` to zero. The regularized model you obtain at this step should produce smaller cost than the one you obtained at the end of the previous exercise.

3 Linear Regression using Stochastic and Mini-batch Gradient Descent

While in the previous two exercises, we employed the batch-based gradient descent algorithm. In this exercise, we will investigate the use of other variants of gradient descent, namely, the stochastic and mini-batch variants.

3.1 Short Theory

3.1.1 Stochastic Gradient Descent (SGD)

Gradient descent is a powerful optimization method and can be applied to a wide range of loss functions . Nevertheless, when dealing with big datasets, two problems arise with the vanilla batch gradient descent (GD) algorithm:

- It might be not possible to load and fit the whole training dataset at one iteration
- The optimization might get stuck at local minima

Stochastic Gradient Descent (SGD) is an effective alternative for the vanilla gradient descent. SGD is very simillar to GD, except that at each iteration, the parameter θ is updated using gradient of the cost function calculated from a single training sample, instead of all the training samples.

3.1.2 Mini-batch Gradient Descent

The vanilla gradient descent (GD) and stochastic gradient descent (SGD) both have their own pros and cons. For example, for big dataset, the former cannot be applied due to memory constraint, the latter takes too long to finish due to the large number of iterations required. As a result, an algorithm which is in-between the two will be of great significance. Mini-batch Gradient Descent is a good alternative for both GD and SGD algorithms. The Mini-batch Gradient Descent algorithm is similar to SGD, but instead of learning on a single sample at each iteration, it learns from a batch consisting of a small number of samples at a time. It has several advantages compared to SGD:

- Similar to SGD, it mitigates the problem of local minima
- It converges faster than SGD, as the number of require iterations is smaller
- It fluctuates less heavily compared to SGD, as the gradient is estimated from a number of training samples at a time

3.2 Python Exercise

In this exercise, you will implement a simple SGD algorithm to estimate the parameters θ for the linear regression problem. The skeleton code is provided for you in the `stochastic_and_minibatch_gradient_descent` notebook. You can reuse parts of the code that you wrote for the first exercise.

Step 1: Load and split dataset then scale features. In this step, you will need to load the Boston dataset from `sklearn` and split it as you do in previous exercise. Concretely, 80 percent of the examples is used for the training set and the rest is for the test set. Then, you have to scale the features to similar value ranges. The standard way to do it is removing the mean and dividing by the standard deviation.

Step 2: Add intercept term and initialize parameters. In this step, you need to add the intercept term to the training and the test matrices. Normally, the intercept term is the first column of your data matrices. Also,

you have to initialize the parameters $\underline{\theta}$ for your model using the normal distribution with $\mu = 0$ and $\sigma = 0.5$.

Step 3: Implement the gradient and cost functions. In this step, you have to implement functions to calculate the cost and its gradient. You can calculate using for loop but vectorizing your calculation is recommended.

Step 4: Stochastic Gradient Descent. In this step, you have to implement the SGD algorithm. At the beginning of the training, or after passing the whole training dataset one time, you need to shuffle your training dataset. It is very important to shuffle the training data and target accordingly. Follow the pseudocode provided in the class and the comment helpers in the exercise.

Step 5: Evaluate θ learned via Stochastic Gradient Descent. In this step, you need to evaluate the parameter θ that you trained via SGD in step 4.

Step 6: Mini-batch Gradient Descent. In this step, you have to implement the Mini-batch Gradient Descent algorithm. The only difference to SGD is the sampling of training batch at each iteration.

Step 7: Evaluate θ learned via Mini-batch Gradient Descent. In this step, you need to evaluate the parameter θ that you trained via Mini-batch Gradient Descent algorithm in step 6.

4 Vectorized Implementation in Numpy

4.1 Short Theory

During the lab sessions so far, you have been extensively using the Numpy library, which allows efficient matrix and algebraic operations in Python, like you usually do in Matlab. Today we will look at an essential feature to further utilize the power of Numpy, namely, *vectorized implementation*. In fact, vectorization implementation is not unique to Numpy: we can apply it to Matlab as well. The basic principle is "*writing as few for loop as possible*", and make use of Numpy's vector/matrix operators as much as possible.

As a demonstration of vectorization implementation, we will consider the problem of pairwise Euclidean distance calculation between two matrices. Suppose we have two matrices, $A \in \mathbb{R}^{n\times d}$ and $B \in \mathbb{R}^{m\times d}$. We can think of A as containing n samples of d dimension, and of B in a similar way. The task is to compute a distance matrix $D \in \mathbb{R}^{n\times m}$, where an element D_{ij}, $i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,m\}$, is the Euclidean distance between $A_i \in \mathbb{R}^d$ (row i of A) and $B_j \in \mathbb{R}^d$ (row j of B). This distance calculation task is the center of many machine learning algorithm, such as *K-means clustering*, *K-nearest neighbor classification* and *ranking and retrieval*.

Recall that the Euclidean distance between vector $A_i \in \mathbb{R}^d$ and $B_j \in \mathbb{R}^d$ is calculated as follows:

$$\begin{aligned} D_{ij} &= \sqrt{(A_i^{(1)} - B_j^{(1)})^2 + (A_i^{(2)} - B_j^{(2)})^2 + \cdots + (A_i^{(d)} - B_j^{(d)})^2} \\ &= \sqrt{\sum_{k=1}^{d} \left(A_i^{(k)} - B_j^{(k)}\right)^2} \end{aligned} \tag{8}$$

4.1.1 Straightforward implementation with for loops

A straightforward way to implement a function performing this task is to loop over all pairs of row vectors between the two matrices A, B and calculate their distances using eq.(8). This approach is simple, but it is not efficient (in Numpy) as it does not utilize the highly optimized vector/matrix operations in Numpy.

4.1.2 Vectorized implementation

Expand eq. (8) further, we have:

$$\begin{aligned} D_{ij} &= \sqrt{\sum_{k=1}^{d} \left(A_i^k\right)^2 - 2A_i^k A_j^k + \left(B_j^k\right)^2} \\ &= \sqrt{\left(\sum_{k=1}^{d} \left(A_i^k\right)^2\right) + \left(\sum_{k=1}^{d} \left(B_j^k\right)^2\right) - 2\left(\sum_{k=1}^{d} A_i^k B_j^k\right)} \\ &= \sqrt{\|A_i\|_2^2 + \|B_j\|_2^2 - 2A_i^T B_j} \end{aligned} \tag{9}$$

As eq. (9) suggests, in order to calculate the distance between the row i in A to the row j in B, we need to calculate the length of A_i, the length of B_j and their inner product. A simple way to do it over all pairs of rows in A and B is, again, performing `for` loop over all i and j. However, a better way is to (i) first calculate the lengths of all rows in A, the length of all rows in B and the inner product between all pairs of rows in A and B; and (ii) sum them over and take the element-wise square root to get the distance matrix D.

4.2 Python Exercise

In this exercise, you will implement and experiment with the two approaches to compute the distance matrix D as presented above.

Step 1: Euclidean distance calculation with `for` loop. In this step, you will follow the straightforward approach, as presented in 4.1.1 to calculate the distance matrix D.

Step 2: Vectorized implementation of Euclidean distance calculation. In this step, you will follow the vectorized approach, as presented in 4.1.2 to calculate the distance matrix D. Note that a correct implementation for this step should not contain any `for` loop.

Step 3: Checking the correctness. Either approach you follow, you should get the same results for the distance matrix D (though this does not necessarily means you have correct results). Run the code cell to check this.

Step 4: Comparing the running time. Run the code cell and compare the difference in running time between the two implementations. The vectorized implementation should be significantly faster than the straightforward implementation. The speed up factor depends on the size of the matrices. With big matrices, you can have very big gain in speed using the vectorized implementation.