1 Linear Regression via Stochastic and Mini-batch Gradient Descent

In this exercise, you will investigate multivariate linear regression using gradient descent.

1.1 Short Theory

1.1.1 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 dataset, 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.

1.1.2 Mini-batch Gradient Descent

The vanilla gradient descent (GD) and stochastic gradient descent (SGD) both have their own disadvantages. 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 required iterations. 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 heaviliy compared to SGD, as the gradient is estimated from a number of training samples at a time

1.2 Python Exercise

In this exercise, you will implement a simple SGD algorithm to estimate the parameters θ for the linear regression problem. Steps 1-3 have been done during previous exercise on GD so you can reuse your codes.

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.

2 Linear Regression via Gradient Descent with Regularization

2.1 Short Theory

Selecting a good model is challenging. From the lecture, you know that assuming too many coefficients with respect to the number of observations results in:

- Too many predictors
- Complicated relations

These effects lead to overfitting. There are many techniques to control overfitting, and in this exercise we focus on l_2 regularization. Concretely, we add an regularization term to the cost function and let the optimization algorithm penalize the model's parameters

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n}\Big[\sum_{i=1}^{n}\left(h\left(\underline{x}_i\right) - y_i\right)^2 + \lambda\sum_{j=1}^{m}\theta_j^2\Big] \tag{1}$$

where λ is the regularization coefficient. The matrix form of this regularized cost is as follows:

$$\mathcal{J}(\underline{\theta}) = \frac{1}{2n}\Big[\left(\mathbf{X}\underline{\theta} - \mathbf{y}\right)^T\left(\mathbf{X}\underline{\theta} - \mathbf{y}\right) + \lambda\underline{\theta}^T\underline{\theta}\Big] \tag{2}$$

Due to this modification, the gradient for the cost function changes to

$$\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \frac{1}{n}\left[\mathbf{X}^{\mathsf{T}}(\mathbf{X}\underline{\theta} - \mathbf{y}) + \lambda\underline{\theta}\right] \tag{3}$$

and the update rule becomes:

$$\underline{\theta}^{\text{new}} = \underline{\theta}^{\text{old}} - \alpha\frac{\partial \mathcal{J}(\underline{\theta})}{\partial \underline{\theta}} = \underline{\theta}^{\text{old}} - \frac{1}{n}\left[\mathbf{X}^{\mathsf{T}}(\mathbf{X}\underline{\theta} - \mathbf{y}) + \lambda\underline{\theta}\right] \tag{4}$$

2.2 Python Exercise

In this exercise, you will modify the functions for calculating the gradient and the cost by adding regularization term. You will have a chance to monitor how the training process changes with different regularization coefficient. Because this exercise is based on the previous exercise of gradient descent for linear regression, the first part of loading and pre-processing data has been done for you.

Step 1: Modify the cost and the gradient functions. In this step, you have to modify the cost and the gradient functions using the provided fomulas 1, 2, 3 and 4. At the end of this step, you can verify your implementation using the previously-implemented `approximate_gradient` function.

Step 2: Train your model with different regularization coefficients. In this step, you train your linear regression model with different coefficients and plot the cost on the training and testing sets on the same figure. By doing it, you will notice which coefficient is appropriate for your model

Step 3: Select regularization coefficient and visualize your prediction. You have to 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 model with regularization sometimes brings a smaller cost than the model without regularization.