2.4 Regularized Regression

In this exercise, you will investigate regularized regression.

2.4.1 Short Theory

While introducing more parameters will typically lead to a reduction in error between the predicted value and training data (training error), it doesn't necessarily lead to a model that better represents the real underlying data distribution. An example is a very complex hypothesis that allows to model the specific noise which is present on the training data. In contrast to the good results on the training data, it will lead to large errors on data which was not used during training (in this case: the parameter computation) due to the random nature of this noise. This behaviour is called 'overfitting' and shows the need for a partitioning of the available data into a training and test set. While the training set is used to computer the model parameters, the test set can be used to spot possible overfitting. To avoid the model to focus on details of the training dataset instead of the generic relation between input and output parameters that ought to be modeled, regularization techniques can be employed.

In case of linear regression, we can regularize the model by adding an extra penalty term to the cost function, controlled by the regularization parameter $\lambda \in \mathbb{R}$. Thus the cost function for simple linear regression in matrix notation,

$$\mathcal{J} = ||\underline{y} - \underline{\beta}X||_2^2, \tag{8}$$

with $X \in \mathbb{R}^{n\times(m+1)}$ the input features, $\underline{\beta} \in \mathbb{R}^{(m+1)\times 1}$ the model parameters and $\underline{y} \in \mathbb{R}^{n\times 1}$ the target values, becomes the following for regularized linear regression:

$$\mathcal{J} = ||\underline{y} - \underline{\beta}X||_2^2 + \lambda||\underline{\beta}||_2^2. \tag{9}$$

After minimization, the following closed-form solution is obtained:

$$\underline{\beta}_{reg} = \left(X^T X + \lambda * I\right)^{-1} X^T \underline{y}. \tag{10}$$

As the penalty term is only taken into account for the parameter computation, the predictions $\underline{\hat{y}}$ can still be computed as follows:

$$\underline{\hat{y}} = X\underline{\beta}_{reg} \tag{11}$$

2.4.2 Python Exercise

To solve this exercise, you can use the partly solved notebook *regularized_regression.ipynb*.

Step 1: Fitting the model

You will need to:

- Implement the function to calculate the pseudo inverse with the regularized term taken into account.
- Use the implemented function to estimate the parameter $\underline{\beta}_{reg}$

Step 2: Make prediction and evaluate

After estimating the parameter $\underline{\beta}_{reg}$, we can make predictions on the testing set. You will need to:

- Make predictions from X_{te} using the estimated parameters β_{reg}.
- Call the *Mean Square Error (MSE)* and *Mean Absolute Error (MAE)* functions to evaluate the learned model.

3 Gradient descent

3.1 Linear Regression using Gradient Descent

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

3.1.1 Short theory

Gradient Descent (GD) is an optimization algorithm that can be used to find a set of parameters that maximizes or minimizes a given (cost) function. Given a function defined by a set of parameters $\underline{\theta}$, GD starts with an initial value of $\underline{\theta}$ and iteratively updates $\underline{\theta}$ in a direction that reduces the value of the (cost) function. This update direction is the reverse of the one indicated 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:

$$\hat{y}_i = h_{\underline{\theta}}(\underline{x}_i) = \underline{\theta}^T \underline{x}_i = \theta_0 + \sum_{j=1}^{m} \theta_j x_{i,j}, \tag{12}$$

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 $\underline{\theta}$ is obtained by minimizing the following cost function:

$$\mathcal{J}_{\underline{\theta}} = \frac{1}{2n} \sum_{i=1}^{n} \left(h_{\underline{\theta}}\left(\underline{x}_i\right) - y_i \right)^2, \tag{13}$$

with n the number of training samples. $\mathcal{J}$ 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:

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

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

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

$$\frac{\partial \mathcal{J}}{\partial \underline{\theta}} = \frac{1}{n} X^T (X\underline{\theta} - \underline{y}). \tag{14}$$

This equation is used by the GD algorithm to iteratively update $\underline{\theta}$ during the training phase according to the following procedure:

- Step 1: Initialize $\underline{\theta}$ randomly (or according to an initialization technique)
- Step 2: Check if a certain criterion is met (e.g. no performance improvements during x cycles or just simply a predefined number of cycles)
  - Step 3: Compute the gradient of the cost function with respect to the parameters $\underline{\theta}$
  - Step 4: Update the parameters $\underline{\theta}$ using the following update rule:

    $$\underline{\theta} \longleftarrow \left(\underline{\theta} - \alpha \frac{\partial \mathcal{J}}{\partial \underline{\theta}}\right) \tag{15}$$

    with α the learning rate.
  - Step 5: Go to step 2

3.1.2 Python Exercise

In this exercise, we will implement the Gradient Descent algorithm to learn a linear regression model. As in the previous chapter, 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 $\underline{\theta}$ 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 $\underline{\theta}$.

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:

$$\frac{\partial \mathcal{J}_{\underline{\theta}}}{\partial \theta_i} \approx \frac{\mathcal{J}(\theta_1, \theta_2, \theta_i + \epsilon, \cdots, \theta_m) - \mathcal{J}(\theta_1, \theta_2, \theta_i - \epsilon, \cdots, \theta_m)}{2\epsilon}, \tag{16}$$

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: Prediction Select a value of α that you think is the best. Use it to learn $\underline{\theta}$ with the GD algorithm. After that, use the $\underline{\theta}$ that you find to make predictions.

3.2 Regularized Linear Regression using Gradient Descent

3.2.1 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 these regularization techniques, we learn a $\underline{\theta}$ that minimizes the following cost function:

$$\mathcal{J}_{\underline{\theta}} = \frac{1}{2n} \sum_{i=1}^{n} \left(h\left(\underline{x}_i\right) - y_i\right)^2 + \lambda \sum_{j=1}^{m} \theta_j^2, \tag{17}$$

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 $\underline{\theta}$. 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 $\underline{\theta}$ is:

$$\frac{\partial \mathcal{J}}{\partial \underline{\theta}} = \frac{1}{n} X^T (X\underline{\theta} - \underline{y}) + \lambda \underline{\theta}. \tag{18}$$

3.2.2 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.

3.3 Linear Regression using Stochastic and Mini-batch Gradient Descent

In the previous two exercises, we employed the (full) 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.

3.3.1 Short Theory

Gradient descent is a powerful optimization method and can be applied to a wide range of loss functions. Nevertheless, when dealing with big datasets, as often is the case in machine learning, two main issues arise with the gradient descent (GD) algorithm we have investigated above, from now on called 'batch gradient descent':

- Memory limitations may arise when trying to load an entire dataset and compute the gradients and associated parameters for an iteration.
- Batch gradient descent can easily be stuck in a local minimum when a suboptimal learning rate is chosen.

Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an effective alternative for the batch gradient descent. The only difference between the two methods is that SGD updates the parameters $\underline{\theta}$ using the gradient of the cost function calculated from a single training sample per iteration, instead of whole training set at once. This is visualized in Figure 2. The updating of parameters is repeated for each sample of the dataset (the processing of the whole dataset is called one 'epoch'). As a consequence, the memory requirements for SGD are much less strict as for batch GD. On the other hand, many iterations are necessary for convergence.

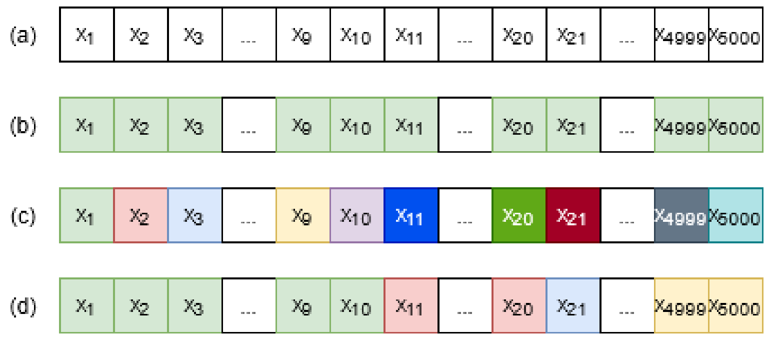

Figure 2: Visualizations of the different possible batch structures for gradient descent (GD): (a) Dataset consisting of 5000 data samples (b) Batch GD (c) Pure SGD (d) Minibatch GD

As each optimization iteration is done for only one sample, the resulting parameter update could lead to a deterioration of the cost value with respect of the whole dataset. While this leads to more unstable behaviour, it also enables the possibility to escape local minima.

Mini-batch Gradient Descent

As batch GD puts heavy memory requirements for large datasets and SGD requires too much time and could possibly be too unstable, a good compromise has to be made. This middle ground is mini-batch gradient Descent. Instead of using only one sample, it employs a 'mini-batch' consisting of multiple samples. The number of samples in such a mini-batch is referred to as 'batch-size'. Using this definition, one could define batch GD as a set-up with batch-size equal to the dataset size. For 'pure' SGD, the batch-size is 1. This is visualized in Figure 2.

- Similar to SGD, it mitigates the problem of local minima
- It converges faster than SGD, as the number of require iterations is smaller

- It endures less fluctuations compared to SGD, as the gradient is averaged over multiple samples per iteration.

3.3.2 Python Exercise

In this exercise, you will implement a simple SGD algorithm to estimate the parameters $\underline{\theta}$ 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 $\underline{\theta}$ learned via Stochastic Gradient Descent. In this step, you need to evaluate the parameter $\underline{\theta}$ 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 $\underline{\theta}$ learned via Mini-batch Gradient Descent. In this step, you need to evaluate the parameter $\underline{\theta}$ that you trained via Mini-batch Gradient Descent algorithm in step 6.