Deep Learning Course: Lab 2

Alex Stergiou, Esther Rodrigo, Fawaz Sammani

10th October 2023

Contents

1 Linear Regression via Gradient Descent **2**
1.1 Short theory 2
1.2 Python Exercise 3
1.3 Migration to Pytorch 5

2 Linear Regression via Stochastic Gradient Descent and Mini-batch Gradient Descent **5**
2.1 Short Theory 6
2.1.1 Stochastic Gradient Descent (SGD) 6
2.1.2 Stochastic Gradient Descent in mini-batches 6
2.2 Python Exercise 6
2.3 Migrate to Pytorch 8

3 Linear Regression via Gradient Descent with Regularization **8**
3.1 Short Theory 8
3.2 Python Exercise 9

1 Linear Regression via Gradient Descent

During the first lecture, you studied various machine learning paradigms such as regression, classification or clustering. In this first exercise, you will investigate (multivariate) linear regression using gradient descent (GD). We will first implement the machine learning algorithm using Numpy and later migrate to Pytorch.

1.1 Short theory

Given a function defined by a set of parameters, $\underline{\mathbf{w}}$, GD finds the direction of the greatest descent – reducing the score function error. Starting from an initial (random) set of parameter values, we iteratively move towards parameters that minimize the score function. This iterative minimization is achieved using calculus, taking steps in the negative direction of the function gradient.

We define the score function that can be applied over each example x_i part of the set of training examples $\underline{\mathbf{x}} = \{x_1 \ldots x_m\}$:

$$f(\underline{\mathbf{x}}; \underline{\mathbf{w}}) = \underline{\mathbf{w}}^T \underline{\mathbf{x}} = \underline{\mathbf{x}}^T \underline{\mathbf{w}} = w_0 + \sum_{j=1}^{m} w_j x_j,$$

where $\underline{\mathbf{w}} = (w_0, w_1, \ldots, w_m)$ is the parameters vector. Given the above score function, let us try to figure out the set of parameters $\underline{\mathbf{w}}$, which minimizes the mean Euclidean distance between the predicted value $f(\underline{\mathbf{x}})$ and the actual output $\mathbf{y}$. We call this loss function the *mean square error* and we can define it as:

$$\mathcal{L}(\underline{\mathbf{w}}) = \frac{1}{N} \sum_{i=1}^{N} \left(f\left(\underline{\mathbf{x}}_i; \underline{\mathbf{w}}\right) - \mathbf{y}_i\right)^2, \tag{1}$$

where N is the number of training examples. The loss function can also be written in the following form using a matrix notation,

$$\mathcal{L}(\underline{\mathbf{w}}) = \frac{1}{N} \left(\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}\right)^T \left(\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}\right)$$

where the $n \times 1$ response vector $\mathbf{y}$ is

$$\mathbf{y} = (y_1, y_2, \ldots, y_n)^T,$$

and the $n \times (m+1)$ design matrix $\mathbf{X}$ is given by

$$\mathbf{X} = (\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_n)^T,$$

or, equivalently,

$$\mathbf{X} = \left(\mathbf{1}_{n\times 1}, \mathbf{x}_1^T, \ldots, \mathbf{x}_j^T, \ldots, \mathbf{x}_m^T\right),$$

where

$$\mathbf{x}_j = (x_{1j}, x_{2j}, \ldots, x_{ij}, \ldots, x_{nj}), j = 1, 2, \ldots, m.$$

The matrix form of the equations is useful and efficient when you're working with numerical computing tools like MATLAB or numpy. If you are familiar with matrices, you can prove to yourself that the two forms are equivalent.

Let us start with some parameter vector $\underline{\mathbf{w}} = \underline{0}$, and keep changing the $\underline{\mathbf{w}}$ to reduce the loss $\mathcal{L}$, i.e.,

$$\underline{\mathbf{w}}_j^{\text{new}} = \underline{\mathbf{w}}_j^{\text{old}} - \frac{1}{N}\sum_{i=1}^{N} \left(f\left(\underline{x}_i; \underline{\mathbf{w}}\right) - \mathbf{y}_i\right)^2 x_{ij}, \forall j \in \{1, 2, \ldots, m\}. \tag{2}$$

The update rule of the parameters can be written in a matrix form as

$$\underline{\mathbf{w}}^{\text{new}} = \underline{\mathbf{w}}^{\text{old}} - \frac{\partial \mathcal{L}(\underline{\mathbf{w}})}{\partial \underline{\mathbf{w}}} = \underline{\mathbf{w}}^{\text{old}} - \frac{1}{N}\mathbf{X}^T(\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}). \tag{3}$$

The parameter vector after the convergence of the algorithm can be used for prediction. Note that for each update of the parameter vector, the algorithm processes the full training set. This algorithm is called *Batch Gradient Descent*.

1.2 Python Exercise

Step 1: Load and split dataset then scale features. In this step, you will need to load the California dataset from `sklearn` and split it. Concretely, 80 percent of the examples are used for the training set and the rest are for the test set. Then, you have to scale the features to similar value ranges. The standard way to do it is by 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 $\underline{\mathbf{w}}$ for your model using the normal distribution with $\mu = 0$ and $\sigma = 0.5$.

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

Step 4: Selecting a learning rate using $\mathcal{L}(\underline{\mathbf{w}})$. You can adjust the step with which you update the weights by reformulating Equation 3:

$$\underline{\mathbf{w}}^{\text{new}} = \underline{\mathbf{w}}^{\text{old}} - \alpha \frac{\partial \mathcal{L}(\underline{\mathbf{w}})}{\partial \underline{\mathbf{w}}} = \underline{\mathbf{w}}^{\text{old}} - \alpha \frac{1}{N} \mathbf{X}^{\mathsf{T}} (\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}). \tag{4}$$

We call α the *learning rate*. We will see the effects of it later on in the course. For now, see the effect of the learning rate to the loss by selecting a value from:

$$0.001 \leq \alpha \leq 10$$

You will need to make an initial selection, run gradient descent, observe the loss, and adjust the learning rate accordingly. You will calculate $\mathcal{L}(\underline{\mathbf{w}})$ using the $\underline{\theta}$ of the current step of the gradient descent algorithm. After several steps of iterations, you will see how $\mathcal{L}(\underline{\mathbf{w}})$ changes at each iteration. **Now,** build your code by following the next steps:

- Run gradient descent for about 50 iterations using your initial learning rate. At each iteration, calculate $\mathcal{L}(\underline{\mathbf{w}})$ and store the result in vector $\mathbf{L}$.
- Test the following values of α: 0.001, 0.003, 0.01, 0.03, 0.1 and 0.3. You may also want to adjust the number of iterations in order to better observe the overall trend of the curve.

- Plot $\mathcal{L}(\underline{\mathbf{w}})$ for several learning rates on the same graph so as to compare how different learning rates affect convergence. In Python, this can be done by using function `plot` from `matplotlib`. Observe the changes in the loss as the learning rate changes. *What happens when the learning rate is too small? Too large?*

Step 5: Using the best learning rate that you found, run gradient descent until convergence to find

1. The final values of $\underline{\mathbf{w}}$.
2. The predicted price of houses from the test set.

1.3 Migration to Pytorch

Now, re-implement the previous code with pytorch following the skeleton code provided in pytorch_ex1.ipynb.

Step 1. Transform the normalized input and target matrices to pytorch using *.from_numpy()* function.

Step 2. Create the weight and bias matrices (torch matrices from normal distribution).

Step 3. Define the regression model function and the loss function (RMSE) from *torch.F* library.

Step 4. Define training loop to compute loss and update weights.

Step 5. Compare the learning procedure by computing RMSE before and after training.

2 Linear Regression via Stochastic Gradient Descent and Mini-batch Gradient Descent

In this exercise, you will investigate multivariate linear regression using stochastic and mini-batch gradient descent.

2.1 Short Theory

2.1.1 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 a 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 to the vanilla gradient descent.

2.1.2 Stochastic Gradient Descent in mini-batches

The vanilla gradient descent (GD) has some disadvantages. For example, for large datasets, it cannot be applied due to memory constraints. Stochastic Gradient Descent can also be applied in mini-batches. Thus, instead of learning on a single sample at each iteration, a batch consisting of a small number of samples at a time is used. The advantages of doing SGD in batches are:

- It mitigates the problem of local minima.
- It converges faster, as the number of required iterations is smaller.
- It fluctuates less, as the gradient is estimated from a number of training samples at a time.

2.2 Python Exercise

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

Step 1: Load and split dataset then scale features. In this step, you will need to load the California dataset from `sklearn` and split it as you do in previous exercise. Concretely, 80 percent of the examples are used for the training set and the rest are for the test set. Then, you have to scale the features to similar value ranges. The standard way to do this is by 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 $\underline{\mathbf{w}}$ for your model using the normal distribution with $\mu = 0$ and $\sigma = 0.5$.

Step 3: Implement the gradient and loss functions. In this step, you have to implement functions to calculate the loss and its gradient. You can calculate this using a 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 $\underline{\mathbf{w}}$ learned via Stochastic Gradient Descent. In this step, you need to evaluate the parameter $\underline{\mathbf{w}}$ that you trained via SGD in step 4.

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

Step 7: Evaluate $\underline{\mathbf{w}}$ learned via Mini-batch Gradient Descent. In this step, you need to evaluate the parameter $\underline{\mathbf{w}}$ that you trained via Gradient Descent with mini-batches in step 6.

2.3 Migrate to Pytorch

Reuse the pytorch code from Section 1 and now implement SGD using the *torch.optim.SGD* function.
Do the same for mini-batch GD using DataLoaders from pytorch.

3 Linear Regression via Gradient Descent with Regularization

3.1 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 / features
- Complex model

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 a regularization term to the loss function and let the optimization algorithm penalize the model's parameters

$$\mathcal{L}(\underline{\mathbf{w}}) = \frac{1}{N}\Big[\sum_{i=1}^{N} \left(f\left(\underline{\mathbf{x}}_i; \underline{\mathbf{w}}\right) - \mathbf{y}_i\right)^2 + \lambda \sum_{j=1}^{m} w_j^2\Big] \tag{5}$$

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

$$\mathcal{L}(\underline{\mathbf{w}}) = \frac{1}{N}\left[\left(\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}\right)^T \left(\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}\right) + \lambda \underline{\mathbf{w}}^T \underline{\mathbf{w}}\right] \tag{6}$$

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

$$\frac{\partial \mathcal{L}(\underline{\mathbf{w}})}{\partial \underline{\mathbf{w}}} = \frac{1}{N}\left[\mathbf{X}^T(\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}) + \lambda \underline{\mathbf{w}}\right] \tag{7}$$

and the update rule becomes:

$$\underline{\mathbf{w}}^{\text{new}} = \underline{\mathbf{w}}^{\text{old}} - \alpha \frac{\partial \mathcal{L}(\underline{\mathbf{w}})}{\partial \underline{\mathbf{w}}} = \underline{\mathbf{w}}^{\text{old}} - \alpha \frac{1}{N} \left[\mathbf{X}^{T} (\mathbf{X}\underline{\mathbf{w}} - \mathbf{y}) + \lambda \underline{\mathbf{w}} \right] \tag{8}$$

3.2 Python Exercise

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

Step 1: Modify the loss and the gradient functions. In this step, you have to modify the loss and the gradient functions using the provided formulas 5, 6, 7 and 8.

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

Step 3: Select the 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 loss here, we set `lambda` to zero. The model with regularization sometimes brings a smaller loss than the model without regularization.