3 Gradient Descent

3.1 Motivation

The explicit solution for linear regression with normal equation from Section 2.3.2.2 can compute the best fitting model for given datapoints. However, this method does not work in practice for larger input datasets, since the method involves matrix multiplication and inversion. The runtime for these operations is approximately cubic in size of the matrix. As a rule of thumb, the process is too time-consuming or breaks for more than 20'000 features or samples. In addition, the normal equation might become numerically instable if the input features are higly correlated.

If we cannot use the normal equation to solve an instance of linear regression explicitly, we can try to find an "as good as possible" solution, i.e. values for the parameters θ such that the corresponding model approximates an optimal model. Note that we now introduce a second level of error: First, any linear model for a given dataset will usually not fit exactly to the data, but rather have some error (the residuals). Now, in addition, we do not expect to find the optimum solution, but rather an approximation of the best solution. Hence the expected error will be even larger. For univariate linear regression, the direction of the resulting line might be "a little off", as shown in the figure below.

▶ Code

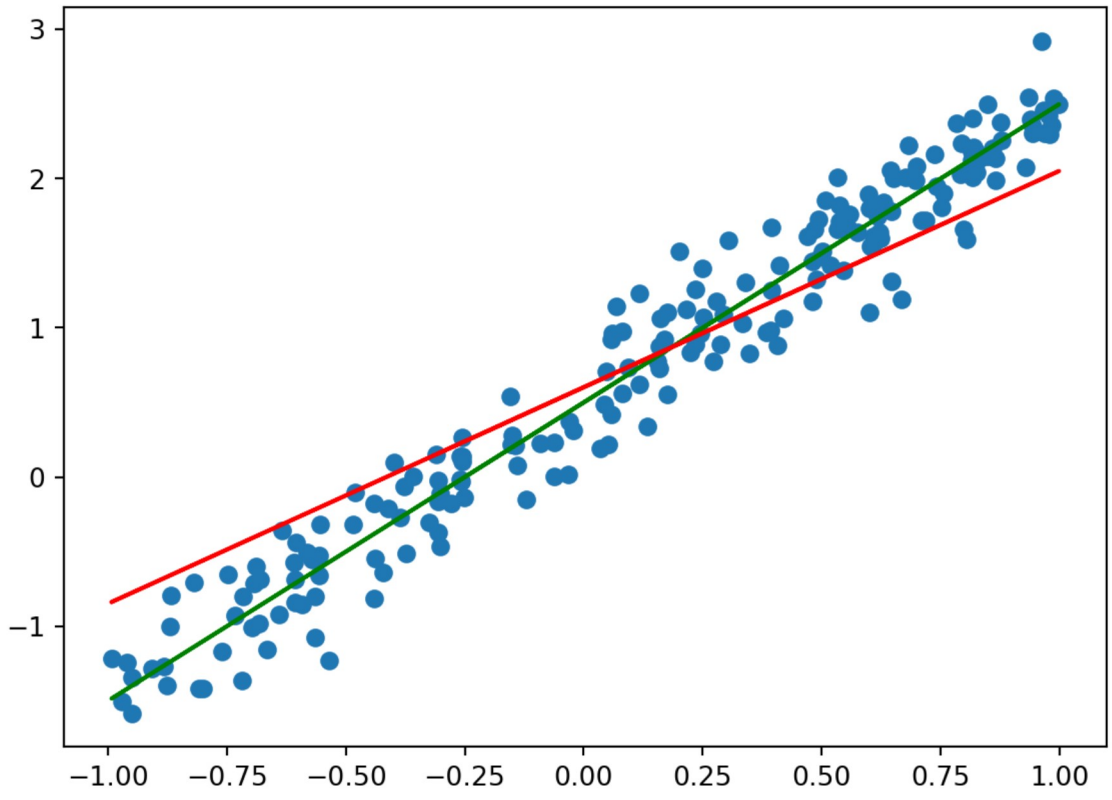

Figure 3.1: Application of gradient descent. Input: points randomly distributed around green line. Red line: Output of gradient descent.

A common approach is to start with some initial values for the parameters θ (e.g. random values), and then iteratively adapt θ "a little bit" such that the corresponding model better fits the given data. The update step is repeated until the process converges, e.g. if the parameters do not change anymore.

Note that once the process has finished and we found our solution θ, we can as usual compute for every input x a corresponding output $\hat{y} = h_{\hat{\theta}}(x)$. To determine how good the data "fits the given data", we can use the cost function $J(\theta) = \frac{1}{2M} \sum_{m=1}^{M} (y^{(m)} - \hat{y}^{(m)})^2$ from Equation 2.1.

3.2 Gradient Descent Algorithm

One of the most well-known algorithms to solve this task is **gradient descent**. Figure 3.2 shows the core idea for a function where we have two paramaters that we need to find, θ_0 and θ_1. The image shows for each parameter combination the corresponding cost $J(\theta_0, \theta_1)$ (note that the image shows the cost function of a non-linear model; for a linear model, the cost function would be convex).

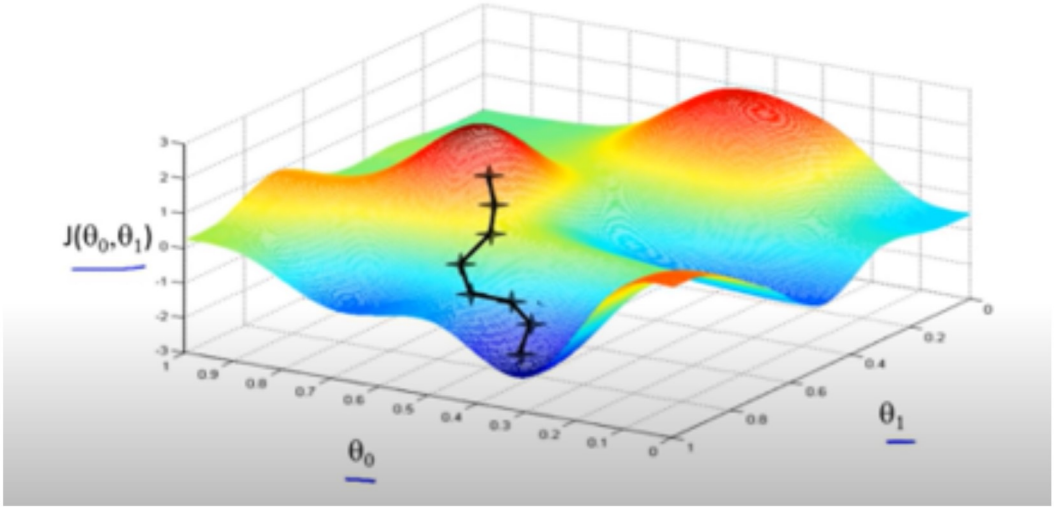

Figure 3.2: Illustration of Gradient Descent: x-axis and y-axis are values of parameters θ_0 and θ_1, respectively, and the z-axis shows for each combination of θ_0 and θ_1 the cost $J(\theta_0, \theta_1)$ of the corresponding model. Starting with initial values for θ_0 and θ_1, it follows a decending path until it ends up with good values for θ_0 and θ_1.

For instance, for parameters $\theta_0 = 0.7$ and $\theta_1 = 0.5$, we have very large costs J (indicated in red), which means that the corresponding model for these parameters does not fit well with the data, whereas $\theta_0 = 0.4$ and $\theta_1 = 0.1$ has a lower cost (indicated in blue).

Note that gradient descent is a very generic algorithm that can be used for optimizing many machine learning models. Moreover, it can be used not just for convex cost functions (like in linear regression), but also for the non-convex cost functions that arise, for instance, in training neural networks. The image above shows a non-convex example.

Gradient Descent uses the gradient (i.e., partial derivaties) to determine in which direction the parameters should be adapted in order to reduce the cost. The gradient gives the direction of steepest *increase*, and by moving opposite the gradient, we cause the function to *decrease* in value.

The generic algorithm for gradient descent for n parameters $\theta_0, \ldots \theta_n$ is as follows:

1. Initialize: $\theta_0, \ldots \theta_n$
2. Repeat until convergence:

$$\text{Update } \theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad \forall j = 0, \ldots n \tag{3.1}$$

$\partial\theta_j$

Here, α is the *learning rate*, which determines how far we step each time. (We will discuss learning rate optimization in Section 3.4.) Since we want to *reduce* the cost, we step *opposite* the direction of the gradient (i.e., $-\alpha \frac{\partial}{\partial \theta_j} J(\theta)$).

Note that the gradient descent algorithm in its generic form can be applied to any learning task where we have a differentiable cost function and need to find proper parameters θ. We will see later that it can be used, for instance, for optimizing neural networks.

3.3 Gradient Descent for Univariate Linear Regression

We now show how to use gradient descent for univariate linear regression. For this, we calculate partial derivatives of J for θ_0 and θ_1 and plug them into the algorithm shown in Equation 3.1.

Recall that in univariate linear regression, we are given M samples $(x^{(m)}, y^{(m)})$ that we use to fit our linear model, and that each input $x^{(m)}$ is a scalar value for which we are given an expected output $y^{(m)}$. Our goal is to find parameters θ_0 and θ_1 that minimize the cost function $J(\theta_0, \theta_1) = \frac{1}{2M} \sum_{m=1}^{M} (y^{(m)} - h_\theta(x^{(m)}))^2$.

In Section Section 2.7, we saw that

$$\frac{\partial}{\partial \theta_0} J = 2 \cdot (-1) \frac{1}{2M} \sum_{m=1}^{M} (y^{(m)} - \theta_0 - \theta_1 x^{(m)}) = \frac{1}{M} \sum_{m=1}^{M} (h(x^{(m)}) - y^{(m)})$$

Similarly [1], we obtain

$$\frac{\partial}{\partial \theta_1} J = \frac{1}{M} \sum_{m=1}^{M} (h_\theta(x^{(m)}) - y^{(m)}) x^{(m)}$$

Plugging the two formulas above into the gradient descent algorithm yields:

1. Initialize: $\theta_0, \ldots \theta_n$
2. Repeat until convergence:

$$\text{Update } \theta_0 = \theta_0 - \alpha \frac{1}{M} \sum_{m=1}^{M} (h_\theta(x^{(m)}) - y^{(m)})$$

$$\text{Update } \theta_1 = \theta_1 - \alpha \frac{1}{M} \sum_{m=1}^{M} (h_\theta(x^{(m)}) - y^{(m)}) x^{(m)} \tag{3.2}$$

Note that θ_0 and θ_1 are updated simultaneously. One common error when implementing gradient descent is to first update θ_0, and then use this updated value when computing $h_\theta(x^{(m)})$ for the update of θ_1. Instead, you have to use the *old* value of θ_0 here.

3.4 Learning Rate

The partial derivatives in gradient descent determine in which direction we need to adapt the parameters θ in order to reduce the cost function J. The larger the magnitude of the gradient, the steeper the descent,

and the more we change the values of θ in a single step. Unfortunately, we can sometimes "overstep" if the gradient is too large. This is shown in Figure 3.3 on the right hand side (note that this is NOT a true image for linear regression, but rather an illustrative example): Starting at the yellow point, the gradient might be so large that we "jump" over the minimum and end up on the left side of the curve and thus miss the minimum. In the worst case, we might even move further away from the minimum with each step, as illustrated in the Figure. To resolve this, we introduced in the gradient descent algorithm (see Equation 3.1) the **learning rate** α, which allows us to scale the step width up and down. Note that if we scale down the step width too much, the algorithm will converge very slowly, as shown in Figure Figure 3.3 on the left side.

Figure 3.3: Illustration of very small (left), good (center) and very large step width of the gradient descent algorithm

3.4.1 Learning Rate Optimization

As a rule of thumb, we want to have a larger learning rate at the beginning of the algorithm (the algorithm moves fast towards the minimum), while the learining rate should be smaller towards the end, where we want to approximate the minimum as good as possible. One way to achieve this is to use a fixed *decay rate* to reduce the learning rate in each step: Starting with an initial value $\alpha_0 = 0.1$, we compute the learning rate for each step as $\alpha_t = \frac{1}{1+decay\_rate*t}\alpha_0$ where $t = 1, 2, \ldots$ is the iteration of the gradient descent loop.

Other methods such as *Adam* or *Adagrad* are more involved and use, for instance, the values of the gradients from one or more previous steps to compute the update of α.

1. A detailed analysis of how to compute the partical derivatives is available at https://medium.com/swlh/the-math-of-machine-learning-i-gradient-descent-with-univariate-linear-regression-2afbfb556131↩