CS 337, Fall 2023
Gradient Descent

Scribes: Karan Godara, Akshat Kumar Gupta, Doddy Subhash
Sankalan Baidya, Omm Agrawal, Gangisetty Krishna Sai Kusalts*
Edited by: Ashwin Ramachandran

August 10, 2023

1 Introduction

Gradient Descent is a **first-order iterative optimization algorithm** for finding local minimum points of a differentiable function. It's commonly used in machine learning and optimization problems (e.g., matrix factorization, neural networks), especially when dealing with non-convex cost functions that don't have a closed-form solution.

2 General Template of GD-Style Algorithms

Let the function being optimized be dependent on the weight vector $\mathbf{w}$. We now wish to find the optimal value of $\mathbf{w}$ so that the function in consideration is minimized (since its the loss function which we generally are trying to optimise in ML). GD-style algorithms helps us in finding that. The general flow of such algorithms is as follows:

- Initialize $\mathbf{w}$ (e.g. $\mathbf{w} = \vec{0}$)
- Repeat
  - Choose a descent direction (directions of fastest decrease)
  - Choose a step size
  - Update $\mathbf{w}$
- Exit repeat loop when certain stopping criterion is met. Some common stopping criterion include (where t represents a time step)
  - $||\nabla L(w_t)||_2 < \epsilon$
  - $||w_{t+1} - w_t||_2 < \epsilon$

*All scribes contributed towards the final notes.

3 Algorithm of Gradient Descent

The flow for the Gradient Descent algorithm keeping in the mind the template above is:

- $\mathbf{w} \leftarrow \mathbf{w}_0$: Initialisation can be done in various ways such as zero initialisation ($w_0 = \vec{0}$), randomly sampled Gaussian etc.
- For the iterative part, we use the following values :
  - Direction : $-\nabla L(\mathbf{w}_t)$
  - Step size (Learning Rate) : $\alpha > 0$ is a hyperparameter
  - Update : $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha \nabla L(\mathbf{w}_t)$
- Some common stopping criteria used among others are:
  - $||\nabla L(w_t)||_2 < \epsilon$
  - $||w_{t+1} - w_t||_2 < \epsilon$

Algorithm 1: Gradient Descent Algorithm with Epochs

Input : Initial weight vector w_0, step size $\alpha > 0$, tolerance $\epsilon > 0$, maximum number of epochs $N_{\max}$

Output: Optimal weight vector w^*

```
w ← w_0;
for t ← 1 to N_max do
    Compute gradient direction: ∇L(w);
    Update: w ← w − α∇L(w);
    if ||∇L(w)||_2^2 ≤ ε then
        break;
return w* = w
```

3.1 Why the name Gradient Descent

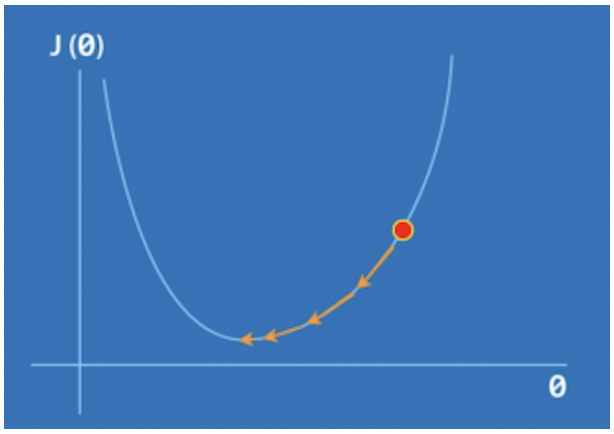
The name Gradient Descent is made of two parts:

- **Gradient** : Gives us the direction of fastest increase in function L

$$\nabla L(w) = \begin{bmatrix} \frac{\partial L(w)}{\partial w_1} \\ \frac{\partial L(w)}{\partial w_2} \\ \vdots \\ \frac{\partial L(w)}{\partial w_d} \end{bmatrix}$$

- **Descent** : Since we update in the direction of fastest decrease in L

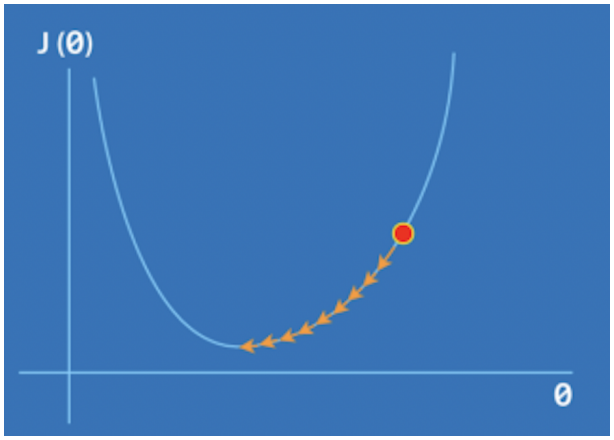
4 GD for the 1-D case and Linear Regression



The overall representation of Gradient Descent in linear regression would take on a similar form as given in the plot above, and given that the step size governs the speed and feasibility of convergence, it leads us to inquire about the following matters:

4.1 What if the step size is too small?

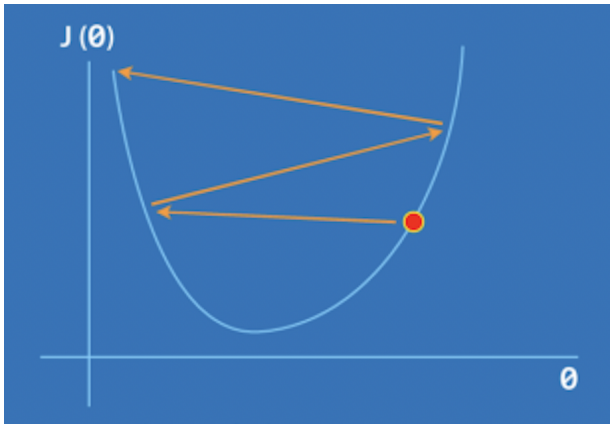
The plot obtained with very small step size would look something like the plot given below



From the plot, we can conclude that the loss will take very long to converge since we only update by a small value each time. This may also result in never reaching the optimal point if we train using a (fixed) maximum number of epochs.

4.2 What if the step size is very large?

The plot obtained with very large step size would look something like the plot given below



Clearly in this case the curve will take longer to converge or in the worst case may diverge as seen above.

5 Weight update rule in GD for linear regression

The formula for update of $\mathbf{w}$ is,

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L(\mathbf{w}) \tag{1}$$

Unregularized Loss:

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \tag{2}$$

$$L(\mathbf{w}) = \frac{1}{N} \sum_{i}^{N} (y_i - \hat{y}_i)^2 \tag{3}$$

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = -\frac{2}{N} \sum_{i}^{N} (y_i - \hat{y}_i)\mathbf{x}_i = \frac{2}{N} \sum_{i}^{N} (\hat{y}_i - y_i)\mathbf{x}_i \tag{4}$$

$$\boxed{\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \alpha \frac{2}{N} \sum_{i}^{N} ([\mathbf{w}^t]^T \mathbf{x}_i - y_i)\mathbf{x}_i}$$

6 Different Variants of Gradient Descent

6.1 (Full) Gradient Descent

Here, we find the gradient of loss function over the entire training dataset. Our gradient update formula in this case is :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L(\mathbf{w}, \mathcal{D}_{train})$$

The training data can be very large in many applications of ML with millions of data points. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. Hence, we usually use other variations of GD which helps us in avoiding this computationally expensive step of finding gradient of loss over entire data every time.

6.2 Stochastic Gradient Descent(SGD)

In this modification of GD, rather than finding gradient of loss over entire training set, we find loss over only single instance of training data that is picked randomly and we update $\mathbf{w}$ based on the gradient for this loss,

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L(\mathbf{w}, \mathcal{D}_{random})$$

where $\mathcal{D}_{random} = \{(x_i, y_i)\}$, $(x_i, y_i) \rightarrow$ randomly sampled point in $\mathcal{D}_{train}$
Now the issue with this version is that there is a lot of noise in the convergence path of the loss function, as a single data point is not representative of entire data set and this leads to lot of fluctuations in computed gradient. Outliers can lead to training instability in this case. For example, training may stop prematurely, if for a certain point gradient becomes zero.

6.3 Mini Batch Gradient Descent

In this version of GD, we try to find the best of both the worlds of GD and SGD by using the gradient of loss computed over only a batch (small subset) of original data set. This helps not only in faster gradient calculation but also is less noisier as batch is still a better representative than a single instance used in SGD.

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L(\mathbf{w}, \mathcal{D}_{batch})$$

where $\mathcal{D}_{batch} = \{(x_i, y_i)\}_{i=1}^{\mathbb{B}}$, $\mathbb{B}$ =batch size
Here $\mathbb{B}$ is a hyper parameter. We should know that larger $\mathbb{B}$ is more stable as we approach close to GD and smaller $\mathbb{B}$ is less stable as we move closer to SGD, so the onus lies on us to find that optimal $\mathbb{B}$ that can help us gain the benefit of both the worlds. Also, note that it is a common convention to use $\mathbb{B} = 16, 32, 64...$ usually a power of 2. This method is an improvement over SGD, as it enhances training stability. Batches are randomly sampled during each epoch.

7 Probabilistic View of Linear Regression

(Elaborated on further in the next lecture)

For training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{\mathrm{n}}$. Let the target y_i have some noise defined as:

$$y_i = f(x_i) + \epsilon_i, \qquad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

$\epsilon_{i's}$ are independent and identically distributed (**i.i.d.**), 0 mean Gaussians with the some variance σ^2.($\mathrm{Cov}(\epsilon_i, \epsilon_j) = 0$ for all $j \neq i$). So basically now we can get the following interpretation of data,

$$\begin{aligned}
& y_i = f(x_i) + \epsilon_i \\
\implies & y_i = \mathbf{w}^T x_i + \epsilon_i \\
\implies & y_i \sim \mathcal{N}(\mathbf{w}^T x_i, \sigma^2) \\
\implies & \mathrm{P}(y_i \mid x_i, \mathbf{w}) \sim \mathcal{N}(\mathbf{w}^T x_i \sigma^2), \qquad \text{[where } \mathrm{P}(y_i \mid x_i, \mathbf{w}) \text{ is likelihood function]}
\end{aligned}$$

For training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{\mathrm{n}}$,

$\mathrm{P}(y_1, y_2, \ldots, y_n \mid x_1, x_2, \ldots, x_n, \mathbf{w}) = \prod_i \mathrm{P}(y_i \mid x_i, \mathbf{w})$ [y_i's are conditionally independent given x_i's]

$\implies \log \mathrm{P}(y_1, y_2, \ldots, y_n \mid x_1, x_2, \ldots, x_n, \mathbf{w}) = \sum_i \log \mathrm{P}(y_i \mid x_i, \mathbf{W})$ [Taking log on both sides]

Note, here we can use log likelihood instead of likelihood because of following reasons,

- Does not change the maxima or minima of function as log is monotonic function
- Mathematical convenience
- Computational convenience as it is easier to add many small values than to multiply them as they might underflow if multiplied