Gradient Descent Tutorial

Table of Contents

1 Introduction to Gradient Descent

Optimization (maximizing or minimizing a particular function) is a common class of problems in many analytic paths of life (computer science, engineering, economics, business,…). Gradient descent is an extremely popular approach (algorithm) to optimization. Inexplicably, it is not necessarily taught in undergraduate CS and engineering programs.

So before I explain the theory and some specific applications of gradient descent, I want to present an analogy that should provide an intuitive understanding of how it behaves (by the way, this is a classic analogy that is commonly offered).

1.1 Analogy for Gradient Descent

Suppose you are the top of a mountain, and you have to reach the lowest point in the mountain range (the valley) in ideally the quickest amount of time. However, here's the catch, there is heavy fog limiting visibility.

What strategy could you take to get down the mountain as quickly as possible?

Without seeing the entire mountain and having all the paths mapped out, you can't be certain that the path you would take is optimal. However, if at each step, you use the information available to choose your next trajectory, you might come up with a reasonably good path (maybe the optimal path, or maybe one that arrives at the valley in near optimal time).

So how do you decide which direction to move at each step? Well, at each step you could choose to move in the direction of quickest descent (ie. always move in the direction of the steepest slope).

This sounds easy, right?

Let's add another measure of difficulty… What if it isn't entirely obvious which path (at any given point) is the quickest/steepest slope? Let's say for argument purposes, the quickest slope (at any given point) could only be determined by a special gauge. Unfortunately, that gauge takes time to operate, and each usage introduces another delay in arriving to the valley. It would make sense to minimize the use of the instrument. This presents a difficulty in choosing the frequency at which you should measure the steepness of the mountain.

This is analogous to obtaining a data set and attempting to determine the quickest path to the minima (ideally the global minima). At each iteration (like the steps down the mountain), local information can be used to find the local minima via calculating the gradient (ie. differentiation) (much like using the gauge to calculate the slope). The amount of time between measurement intervals on the gauge is the learning rate of the algorithm.

1.2 Description

So what exactly is gradient descent (more formally)? It's an iterative optimization algorithm to find a local minimum of a function. Each iteration is a small step proportional to the negative of the gradient of the function at the current point.

It is based on the observation that a multivariate function $F(x)$ defined and differentiable in a neighborhood of point a decreases fastest in the direction of the negative gradient of F at a that is to say $-\nabla F(a)$.

If $a_{n+1} = a_n - \gamma \nabla F(a_n)$ for γ small enough, then $F(a_n) \geq F(a_{n+1})$. That is saying that if we move in the direction of the negative gradient, then the subsequent step will be less than or equal to the current step (ie. moving downhill or level in our original analogy).

Thus the approach starts with an initial guess x_0 for a local minimum of F, and consider the sequence $x_0, x_1, x_2, \ldots$ such that

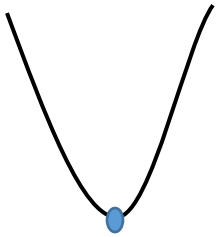
$$x_{n+1} = x_n - \gamma_n \nabla F(x_n)$$

We have $F(x_0) \geq F(x_1) \geq \cdots \geq F(x_n) \geq F(x_{n+1})$. It should be noted that γ is allowed to vary over iterations.

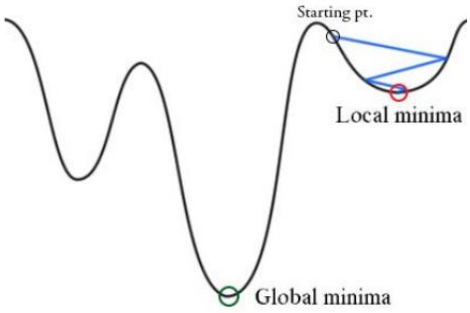
The sequence of x_n should converge to the local minimum. (When F is convex, all local minima are also global).

1.3 Challenges in Executing Gradient Descent: When does it fail?

- An ideal optimization problem would be **strongly convex** (as in the illustration below). The blue dot reflects the global minima (the absolute lowest point in our data set). We aim to find the global minima.



- Now consider a **non-convex data sets.** In these instance, the gradient descent algorithm will fail or return an incorrect solution. In the illustration, we can see that the gradient descent from the given starting point will yield a local minima, which is the incorrect solution.



- **Saddle points** also present a difficulty. A saddle point is a point where the gradient becomes zero, but it is not an extremes (local minima or maxima points).

2 Details of Traditional Gradient Descent

There many variants of gradient descent algorithms. We can foremost categorize them based of two criteria: data integration and differentiation technique. That is to say, some approaches (full batch) use the entire data at once, while others (stochastic) sample while computing the gradient. Likewise, some approaches use first order differentiation, while others use second order differentiation.

2.1 Cost Function

Up this point we really haven't discussed what we are trying to optimize or minimize. Generally, the optimization is applied to a cost function. So what is this cost function?

In a business/economics model, this cost function would be something like a supply/demand, price/demand, or profit model. We would be aiming to optimize something like sales or profit, or minimize costs.

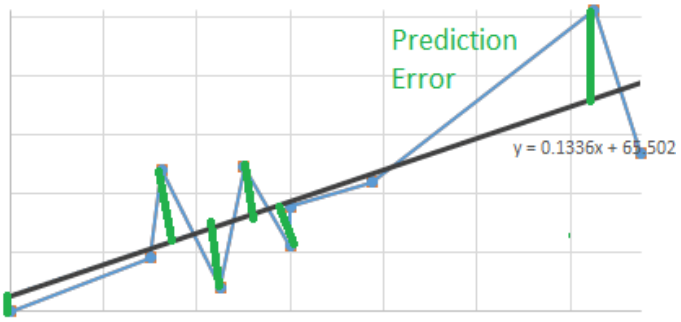
In computing, this cost model can be less intuitive. While most contemporary implementations are focused on data machine learning algorithms, the general objective of gradient descent is to optimize a data fitting model. This has implications beyond machine learning, some non-linear (ie. deformable) registration algorithms use gradient descent to fit one data set/model to another (in circumstances where the relation between the two models is non-linear).

EXAMPLE 1 PART A

At this point, we will take a brief detour to illustrate one such usage. Suppose you were given price data from a local housing market, and we wanted to predict the price a new house on the market given its size.

House size (sqft)	1400	1600	1700	1875	1100	1550	2350	2450	1425	1700
House Price (k$)	245	322	288	309	199	220	455	334	320	255

Now a very basic model would be to apply a linear model. A linear model is of the form $y = mx + b$, where the x variable is the size and the y variable is the price, the two coefficients are m and b, which are the intercept and slope.



In the above chart, we get a linear model that says the price of a house $Price = \$65{,}000 + \$1{,}330 * size$. The difference between the predicted model and the actual data is our prediction error. So in this case, (still sticking to a linear model), we are trying to find an m and b that minimize the error between actual and predicted values.

So our model cost function in this case could be the **Sum of Squared Errors (SSE)**

$$SSE = \frac{1}{2}\sum(actual - predicted)^2$$

(note: the $\frac{1}{2}$ is introduced for mathematical convenience, and that the SSE is just one of many possible metrics for error)

With this in mind, we can look at some of the most common gradient descent algorithms and their implementation.

2.2 How to Calculate a Gradient?

The gradient of a scalar valued multivariate function $f(x, y, \dots.)$ is denoted by ∇f, packages all the partial derivative information into a vector

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \vdots \end{bmatrix}$$

The physical meaning of the gradient is the direction the function f increases or decreases most rapidly.

2.3 Conventional/Traditional (Batch) Gradient Descent

In its simplest form (sometimes informally stated as vanilla/conventional/traditional, but more properly called "batch"), gradient descent works by taking small steps in the direction of the minima by taking gradients of the cost function with respect to the parameter θ for the entire data set. This can be expressed in equation form as

$$\theta_i = \theta_{i-1} - \eta \cdot \nabla_\theta J(\theta_{i-1})$$

Where θ_i is the i[th] iteration of the parameters, and η is the learning rate (iteration step size).

Input: data, cost function, termination criteria, learning rate

Step 0: pre-process data (example: normalize)

Step 1: initialize parameters (arguments of cost function)

Step 2: calculate the gradient of the cost function and update the parameters to those that optimize the cost function.

Step 3: use the new parameters to update the prediction and calculate the cost function

Step 4: repeat steps 2 and 3 until termination criteria is achieve

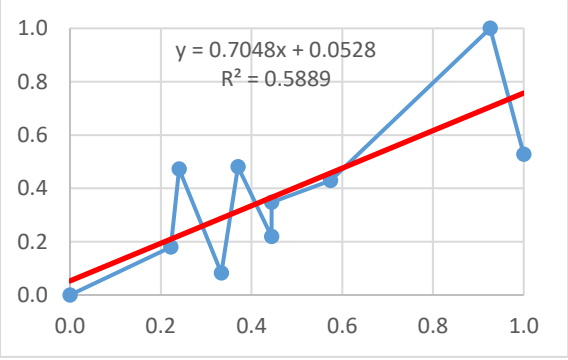
To elucidate how conventional gradient descent works, let's continue our example, going step-by-step:

EXAMPLE 1 PART B

Step 0: Normalize the data

House size	0	0.2	0.2	0.3	0.4	0.4	0.4	0.6	0.9	1
House Price	0	0.2	0.5	0.1	0.5	0.2	0.3	0.4	1	0.5
Prediction	0.08	0.21	0.21	0.28	0.34	0.34	0.34	0.48	0.67	0.74
Prediction Error	0.08	0.01	0.29	0.18	0.16	0.14	0.04	0.08	0.33	0.24

This gives a total SSE of 1.54



Step 1: Initialize the parameters

So we have a cost function of

$$SSE = \frac{1}{2}\sum(actual - predicted)^2 = \frac{1}{2}\sum\left(Y_{Actual} - Y_{predicted}\right)^2 = \frac{1}{2}\sum\left(Y_{Act} - (mx + b)\right)^2$$

Where Y_{act} is the data (constant), $m\ and\ b$ are the parameters, and x is the sqft of the property.

So in this case we could initialize m = 0.7 and b = 0.05

Step 2: Calculate the gradient with respect to the parameters

$$\frac{\partial SSE}{\partial m} = -(Y_{actual} - Y_{predicted})$$

$$\frac{\partial SSE}{\partial b} = -(Y_{actual} - Y_{predicted})x$$

m	**b**	**X**	**Y**	$\boldsymbol{Y_{predicted}}$	**SSE**		$\boldsymbol{\frac{\partial SSE}{\partial m}}$	$\boldsymbol{\frac{\partial SSE}{\partial b}}$
0.60	0.00	0.00	0.00	0.00	0.00		0.00	0.00
		0.20	0.20	0.12	0.08		0.08	0.02
		0.20	0.50	0.12	0.38		0.38	0.08
		0.30	0.10	0.18	0.08		-0.08	-0.02
		0.40	0.50	0.24	0.26		0.26	0.10
		0.40	0.20	0.24	0.04		-0.04	-0.02
		0.40	0.30	0.24	0.06		0.06	0.02
		0.60	0.40	0.36	0.04		0.04	0.02
		0.90	1.00	0.54	0.46		0.46	0.41
		1.00	0.50	0.60	0.10		-0.10	-0.10
				total SSE	1.50	**Sum**	1.06	0.52

Step 3: Adjust the weights with the gradients to reach the optimal values where SSE is minimized

We need to update the random values of m,b so that we move in the direction of optimal m, b.

Update rules:

$$m_{new} = m - learning\ rate * \sum \frac{\partial SSE}{\partial m}$$

$$b_{new} = b - learning\ rate * \sum \frac{\partial SSE}{\partial b}$$

We would repeat the calculation in the tables using our new m and b, and recalculate the SSE

Step 4: We would repeat the calculations until a stop condition (such as the error being 0, or approaching 0, or an iteration stop limit)

The last table would look something like this:

m	b	X	Y	$Y_{predicted}$	SSE		$\frac{\partial SSE}{\partial m}$	$\frac{\partial SSE}{\partial b}$
0.66	0.08	0.00	0.00	0.08	0.08		-0.08	0.00
		0.20	0.20	0.21	0.01		-0.01	0.00
		0.20	0.50	0.21	0.29		0.29	0.06
		0.30	0.10	0.28	0.18		-0.18	-0.05
		0.40	0.50	0.34	0.16		0.16	0.06
		0.40	0.20	0.34	0.14		-0.14	-0.06
		0.40	0.30	0.34	0.04		-0.04	-0.02
		0.60	0.40	0.48	0.08		-0.08	-0.05
		0.90	1.00	0.67	0.33		0.33	0.29
		1.00	0.50	0.74	0.24		-0.24	-0.24
				total SSE	1.54	**Sum**	0.00	0.00

Note: This data is a small sample set with simplistic models (clearly, the linear model isn't a good fit). Real data could be (almost certainly will be) multivariate.

In its simplest form, we can express gradient descent as:

Step 1: Initialize initialize parameters (guess)
Step 2: Update calculate cost function and update = gradient of parameters update = learning rate x gradient of parameters
Step 3: Parameter Update Step parameters(new) = parameter(old) – update = parameters old – learning rate x gradient of parameters

In pseudocode, this looks like

```
for i = 0; i< max_num_iterations; i++  (*note)
        grad_paramenters = evaluate_gradient(loss_function, data, parameters(i))
        parameters(i+1) = parameters(i) – learning_rate * grad_paramenters
```

*** note: the for can be replaced by a while condition for equivalent effectiveness

Batch gradient is guaranteed to converge to a global minimum for convex functions and to a local minimum for non-convex functions. As we need to calculate the gradients for the entire dataset to perform just one update, batch gradient can be slow and is intractable for datasets that cannot be manipulated in memory.

In the next section and chapter, we will consider some other variants (flavors) of gradient descent.

2.3 Stochastic Gradient Descent (SGD)

Stochastic gradient descent performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$

$$\theta_i = \theta_{i-1} - \eta \cdot \nabla_\theta J(\theta_{i-1}; x^{(i)}; y^{(i)})$$

While batch gradient descent performs redundant computations for large datasets (as it recomputes the gradient for similar examples, before each parameter update), stochastic gradient descent does not. It performs one update at a time, and is therefore typically faster.

One shortcoming is that SGD will keep overshooting, however, if the learning rate is slowly decreased SGD shoes the same behavior as batch gradient descent (and tends to converge to a minima).

In pseudocode, this looks like

```
for i = 0; i< max_num_iterations; i++  (*note)
        random_shuffle(data)


        for example in data
                grad_paramenters = evaluate_gradient(loss_function, data, parameters(i))
                parameters(i+1) = parameters(i) – learning_rate * grad_paramenters
```

2.4 Mini-batch Gradient Descent

Mini-batch gradient descent attempts to merge between the two previous approaches by performing an update for every mini-batch of n training examples:

$$\theta_i = \theta_{i-1} - \eta \cdot \nabla_\theta J(\theta_{i-1}; x^{(i;i+n)}; y^{(i;i+n)})$$

This approach reduces the variance of the parameter updates, which can lead to more stable convergence. Likewise, it makes it computationally feasible to efficiently compute the gradient using matrix optimization libraries. In pseudocode, this looks like

```
for i = 0; i< max_num_iterations; i++  (*note)
        random_shuffle(data)

        for batch in batches(data, batch_size)
                grad_paramenters = evaluate_gradient(loss_function, data, parameters(i))
                parameters(i+1) = parameters(i) – learning_rate * grad_paramenters
```

2.5 Challenges

Conventional batch gradient descent presents some challenges, such as:

- Choosing a proper learning rate. Choosing a rate that is too small can lead to slow convergence, while a rate too large can hinder convergence and cause the loss function to oscillate around the minimum (or even diverge).
    - One approach is to adjust the learning rate. This can be achieved by "scheduling", which is to adjust the rate based change in the objective function. In other words, as the change in the objective function between iterations becomes less pronounce, the learning rate is made "smaller". These schedules, however have to be defined in advance and are thus unable to adapt to a dataset's characteristics.
    - Another issue with the learning rate is that it applies to all parameters. If our parameters have different rates of convergence, a learning rate suitable to one, may not be for the others.
- If the function is non-convex, the gradient descent may be trapped in suboptimal local minima or saddle points. Saddle points are usually surrounded by a plateau of the same objective function values, which makes it difficult for standard gradient descent to "escape" as the gradient is close to zero in all directions.

3 Variants of Gradient Descent

This section is a brief description of some alternate approaches to traditional gradient descent approach discussed in the previous section. By no means is this an in-depth investigation, but rather a primer to understand the approach and how it may impact your implementation.

3.1 Gradient Descent with Momentum

Stochastic gradient descent (SGD) can behave poorly in areas where the surface curves much more steeply in one dimension than in another (termed "ravines). Under such circumstances, SGD will oscillate slowing the progress to a global minima. By applying a momentum term, the convergence to the global minima can be accelerate.

$$v_i = \gamma v_{i-1} + \eta \cdot \nabla_\theta J(\theta_{i-1})$$

$$\theta_i = \theta_{i-1} - v_t$$

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation. This behavior is analogous to rolling a ball downhill. In that it may not take the optimal path at any given point, but will generally move quickly downhill.

3.2 Nesterov Accelerated Gradient (NAG)

Continuing the analogy of a ball rolling downhill. A ball rolling downhill following the slope may not be satisfactory. Particularly, due to that approaches nature to overshoot. We would hope that it could decelerate before the hill slopes up again.

Nesterov Accelerated Gradient (NAG) is a way to give out momentum term a degree of prescience or foresight [3]. Knowing that the momentum term γv_{i-1} is used to move the parameters θ_i, we can

obtain a rough idea of where our parameters are going to be in the subsequent step. Thus, we can look ahead by calculating the gradient not with respect to the current parameters, but rather with respect to those anticipated/approximate parameters (in our future position). We can express this as

$$v_i = \gamma v_{i-1} + \eta \cdot \nabla_\theta J(\theta_{i-1} - \gamma v_{i-1})$$

$$\theta_i = \theta_{i-1} - v_t$$

3.3 ADAGRAD

AdaGrad (Adaptive Gradient) is an approach that adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters [4]. It is suitable to sparse data.

In previously discussed approaches, we performed an update for all parameters θ_i at once as every parameter and all parameters used the same learning rate η. As ADAGRAD uses a different learning rate for every parameter θ_i at every iteration (or time step) *i*, we first show ADAGRAD's per parameter update, which we then formulate as a vector. For simplicity, we set $g_{i,k}$ to be the gradient of the object function with respect to the parameter θ_{i_k}(where k is the kth parameter) at iteration/time step i :

$$g_{i,k} = \nabla_\theta J(\theta_i)$$

The SGD update for every parameter θ_{i_k}at each iteration/time step i then becomes:

$$\theta_{i+1,k} = \theta_{i,k} - \eta\, g_{i,k}$$

In its update rule, ADAGRAD modifies the general learning rate η at each time step i for every parameter $\theta_{i+1,k}$ based on the past gradients that have been computed for $\theta_{i+1,k}$:

$$\theta_{i+1,k} = \theta_{i,k} - \frac{\eta\, g_{i,k}}{\sqrt{G_{i,k} + \varepsilon}}$$

$G_i \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element is the sum of the squares of the gradients with respect to θ_k up to interval/time-step i while ε is a smoothing term that avoids division by zero.

Adagrad's primary advantage is that it eliminates the need to manually tune the learning rate. Its primary weakness is the accumulation of the squared gradients in the denominator (since every added term is positive, the accumulated sum keeps growing during training). This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

3.4 ADADELTA

Adadelta is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate (see shortcoming in previous paragraph) [5]. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_i$ at time step i then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_i = \gamma E[g^2]_i + (1-\gamma) g_i^2$$

For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta\theta_i$:

$$\Delta\theta_i = -\eta \, g_{i,k}$$

$$\theta_{i+1} = \theta_i + \Delta\theta_i$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta\theta_i = -\frac{\eta}{\sqrt{G_{i,k} + \varepsilon}} \odot g_i$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_i$:

$$\Delta\theta_i = -\frac{\eta}{\sqrt{E[g^2]_i + \varepsilon}} g_i$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_i = -\frac{\eta}{\sqrt{RMS[g]_i}} g_i$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_i = \gamma E[\Delta\theta^2]_{i-1} + (1-\gamma) {\Delta\theta^2}_i^2$$

The root mean squared error of parameter updates is thus:

$$RMS[\Delta\theta]_i = \sqrt{E[\Delta\theta^2]_i + \varepsilon}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with finally yields $RMS[\Delta\theta]_{i-1}$ the Adadelta update rule:

$$\Delta\theta_i = -\frac{RMS[\Delta\theta]_{i-1}}{RMS[g]_i} \mathrm{g_i}$$

$$\theta_{i+1} = \theta_i + \Delta\theta_i$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

3.5 ADAM

ADAM is one more adaptive technique which builds on adagrad and further reduces it downside [6]. In other words, you can consider this as momentum + ADAGRAD. In addition to storing an exponentially decaying average of past squared gradients v_i like Adadelta, Adam also keeps an exponentially decaying average of past gradients m_i, similar to momentum:

$$m_i = \beta_1 m_{i-1} + (1-\beta_1) g_i$$

$$v_i = \beta_2 v_{i-1} + (1-\beta_2) g_i^2$$

$m_i \ and \ v_i$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As $m_i \ and \ v_i$ are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small.

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m_i} = \frac{m_i}{1 - \beta_1^i}$$

$$\widehat{v_i} = \frac{v_i}{1 - \beta_2^i}$$

which yields the Adam update rule:

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{\widehat{v_i}} + \varepsilon} \widehat{m_i}$$

Bibliography

[1] http://ruder.io/optimizing-gradient-descent/

[2] https://www.analyticsvidhya.com/blog/2017/03/introduction-to-gradient-descent-algorithm-along-its-variants/

[3] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

[4] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html

[5] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

[6] Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13