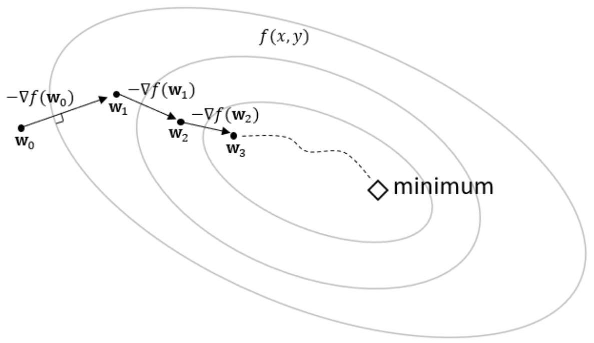
MA255, Application Day 1, Numerical Optimization

February 22, 2019

Sometimes we want to find the minimum of a function, but we can't apply the analytical techniques from class, and instead must resort to *numerical* techniques. The **gradient descent** method starts with an initial guess where the minimum is, and then takes small steps in the direction of the negative of the gradient until it finds a place where the gradient is zero.[1]


$f(x,y)$
$-\nabla f(\mathbf{w}_0)$
$-\nabla f(\mathbf{w}_1)$
$-\nabla f(\mathbf{w}_2)$
$\mathbf{w}_0$
$\mathbf{w}_1$
$\mathbf{w}_2$
$\mathbf{w}_3$
minimum

Gradient Descent is a simple, old technique, but it is surprisingly effective — as a result, it is at the heart of the way most artificial intelligence (AI) and machine learning algorithms are trained.

Consider the function[2]

$$f(x, y) = \ln\Big((1 - x)^2 + 20(y - 0.25x^2)^2 + 1\Big). \tag{1}$$

1. Define this as a function `f[x_,y_]` in *Mathematica* (recall the natural logarithm is `Log`). Using `ContourPlot`, **sketch** a plot of the level sets of this function over the domain $(x, y) \in [-1, 2] \times [-1, 2]$. Use `NMinimize` to **find the minimum** of this function, and **where** the minimum occurs. Depict this on your sketch below.

[1]This "stepping in the direction of the slope" may remind you of **Euler's method** from MA153 — you're right! We will make this connection explicit.

[2]This is called Rosenbrock's banana function. (Seriously.) It is a popular test function for numerical optimization methods.

Gradient Descent

Commands like `NMinimize` work by using numerical algorithms like gradient descent. Let's implement gradient descent ourselves to understand how it works.

Represent the initial guess with a position vector $\mathbf{w}_0 = \langle x_0, y_0 \rangle$. Now move a little in the direction of the steepest descent (i.e. the direction where the negative gradient points), and iterate:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - h\,\nabla f(\mathbf{w}_k) \qquad k = 0, 1, 2, ... \tag{2}$$

The parameter h controls how much we want to move in the direction of steepest descent, and is called the stepsize or **learning rate** (this is the "learning" in machine learning).

Let's implement this in *Mathematica* on the function $f(x, y)$ from the previous section, using an initial guess of $(-0.5, -0.5)$, a learning rate of $h = 0.1$, and taking 10 steps. Run the following commands (place these all in one cell):

```
gf[{x_,y_}] = Grad[f[x,y], {x,y}];
w[0] = {-0.5, -0.5};
h = 0.08;
nsteps = 20;
Do[
  w[k+1] = w[k] - h * gf[w[k]],
  {k, 0, nsteps-1}
]
pts = Table[w[i], {i, 0, nsteps}]
```

(You may recognize this from other programming classes as a loop. Note we also defined the gradient in such a way that we can pass it a vector input, like `w[k]`, instead of x and y separately.)

This stores the values of $\mathbf{w}_0, \mathbf{w}_1, ..., \mathbf{w}_{20}$ in `w[0]`, `w[1]`, ... `w[20]`, and then places all the points in a list called `pts`.

2. What is gradient descent's estimate of **where** the minimum occurs after 20 steps? What is the **value** of the function at this point? **Compare** this to the actual minimum you found in Question 1. Round answers to two decimal places.

3. Plot the function's level sets (`ContourPlot`) overlaid with the path of your gradient descent results (`ListPlot`). Use the following code as a foundation, and **sketch** the result.

```
Show[
  ContourPlot[ ... , Contours->20],
  ListPlot[pts, Joined->True, Mesh->All, PlotRange->All, PlotStyle->Black]
]
```

4. Experiment with different learning rates (h) and total steps (`nsteps`). Comment on the following:

   (a) Compare using a small learning rate ($h < 0.08$) and lots of steps to using a large learning rate and few steps.

   (b) What happens to your steps as you get closer to the minimum? Why?

   (c) Select a learning rate and total steps that gets as close to the actual minimum as you can. Record your results for h, `nsteps`, the estimated (x, y) where the optimum occurs, and the function value at this point.

Accelerated Gradient Descent

As you probably found in the previous section, if we make our stepsize big, we will take too big of steps and "overshoot" the optimum. But if we make our stepsize small, since the gradient also gets very small near the optimum, our steps get extremely short and it takes too long to reach our goal! We can **accelerate** the method by adding another term, called a **momentum term**:

$$\mathbf{w}_{k+1} = \underbrace{\mathbf{w}_k - h\nabla f(\mathbf{w}_k)}_{\text{gradient descent}} + \underbrace{\beta(\mathbf{w}_k - \mathbf{w}_{k-1})}_{\text{momentum term}} \tag{3}$$

The momentum term is based on the amount we moved in the **previous step**. So if we changed a lot last time, then even if the gradient term is small, the momentum term ensures we change a lot again. The **momentum parameter** β controls how much of this momentum we want.

To implement accelerated gradient descent, run the following commands (place these all in one cell):

```
h = 0.075;
b = 0.5;
nsteps = 10;
w[0] = {-.5, -.5};
w[1] = w[0] - h*gf[w[0]];
Do[
 w[k + 1] = w[k] - h*gf[w[k]] + b*(w[k] - w[k-1]),
 {k, 1, nsteps}
 ]
pts = Table[w[i], {i, 0, nsteps}]
```

5. Run the accelerated gradient descent algorithm from initial condition $\mathbf{w}_0 = \langle -0.5, -0.5 \rangle$, with learning rate $h = 0.08$ and momentum parameter $\beta = 0.5$, for 10 steps. **Plot** your results on a contour plot of $f(x, y)$. Notice that some of the steps don't appear to move strictly in the direction of steepest descent: give an explanation for **why**.

6. Increase the learning rate h and momentum β. What happens? **Sketch** your result.

7. One way to improve both gradient descent and the accelerated version is to have the learning rate h start big, and get smaller as the iterations continue. This lets the method take big, bold steps early on, but take smaller, more conservative steps when it is (hopefully) close to the optimum. Fill in the blank below with a suggestion for how to modify the gradient descent step in this way. (*Hint:* use the iteration number k.)

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \underline{\hspace{4cm}} \nabla f(\mathbf{w}_k)$$

Bonus: Implement this by modifying your code from a previous section and comment on the results.

Follow-on (Bonus): A Physical Analogy

Let's get some physical intuition about the gradient descent method.

Imagine our algorithm trying to find the minimum as a particle rolling down a hill into a valley. Represent our particle's (x, y) position with the vector function $\mathbf{w}(t) = \langle x(t), y(t)\rangle$. Represent the force pulling the particle toward the center of the valley as a vector $\mathbf{F}(x, y)$ which depends on the (x, y) position of the particle.

Let's further imagine this vector force field happens to correspond to the negative of the gradient of some function $f(x, y)$, i.e. $\mathbf{F} = -\nabla f$. This is hopefully a natural assumption, since we can imagine the force being large when the particle is high on the mountain, but getting smaller as it reaches flat terrain near the bottom of the valley.[3]

Finally, imagine the particle is subject to friction proportional to its velocity, with coefficient α.

Newton's Second Law of Motion in 2-dimensions states $\mathbf{F} = m\mathbf{a}$, where here $\mathbf{F}$ represents all forces acting on our particle, and $\mathbf{a} = \mathbf{w}''$, the acceleration vector of our particle. This gives

$$\underbrace{-\nabla f(\mathbf{w})}_{\text{gravity}} \underbrace{-\alpha\, \mathbf{w}'(t)}_{\text{friction}} = m\, \mathbf{w}'' \tag{4}$$

If the particle has **zero mass**, $m = 0$, this reduces to the autonomous system of differential equations

$$\mathbf{w}'(t) = -\frac{1}{\alpha}\nabla f(\mathbf{w}(t)) \tag{5}$$

Instead of trying to solve this differential equation for $\mathbf{w}(t)$, let's approximate (or *discretize*) by applying a *finite difference approximation*

$$\mathbf{w}'(t) \approx \frac{\mathbf{w}_{k+1} - \mathbf{w}_k}{\Delta t}. \tag{6}$$

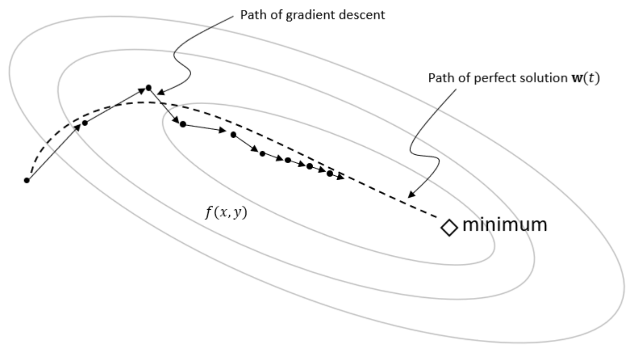
As $\Delta t \to 0$, this approximation is exact (it's the definition of the derivative!).

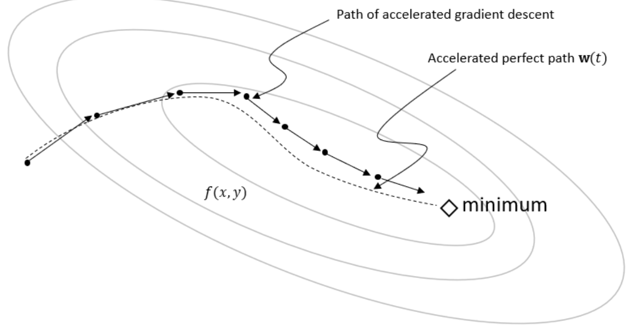
1. Substitute the forward difference approximation in Equation (6) into our differential equation in Equation (5) and rearrange terms to show we obtain the relation:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - h\nabla f(\mathbf{w}_k). \tag{7}$$

   where h is in terms of Δt and α.

[3] In Block 4, we will learn more about *vector fields*. Gravity and electromagnetism are examples of vector force fields which are *conservative*, meaning there is a function f such that $\mathbf{F} = \nabla f$.




Path of accelerated gradient descent
Accelerated perfect path $\mathbf{w}(t)$
$f(x,y)$
minimum

You have now demonstrated that gradient descent is approximating the path of a massless particle moving through a force field, and our stepsize h captures both how big of steps we are willing to take (Δt) and how much "friction" we are submitting our particle to (α).

You should also notice that Equation (7) is an **Euler's Method** step! Gradient descent is equivalent to an Euler's Method approximation of the ODE in Equation (5).

Now let's give our particle a **nonzero mass**. The intuition here is that this will give our particle some *momentum* and (hopefully) prevent it from creeping slowly through the almost-flat areas near the optimum, like you observed in the App Day.

Recall our full differential equation for the particle's motion, Equation (4), and now assume a mass $m > 0$. As before, we can "discretize" this differential equation by using finite difference approximations for both $\mathbf{w}'$ and $\mathbf{w}''$,

$$\mathbf{w}' \approx \frac{\mathbf{w}_{k+1} - \mathbf{w}_k}{\Delta t} \qquad \text{and} \qquad \mathbf{w}'' \approx \frac{\mathbf{w}_{k+1} - 2\mathbf{w}_k + \mathbf{w}_{k-1}}{(\Delta t)^2} \tag{8}$$

2. Substitute the approximations in Equation (8) into the differential equation Equation (4) and rearrange terms to show we get the following modified, **accelerated** version of the gradient descent step:

$$\mathbf{w}_{k+1} = \underbrace{\mathbf{w}_k - h\nabla f(\mathbf{w}_k)}_{\text{gradient descent}} + \underbrace{\beta(\mathbf{w}_k - \mathbf{w}_{k-1})}_{\text{momentum term}} \tag{9}$$

   where now h and β should be in terms of Δt, α, and m.

So if gradient descent approximates the path of a massless particle moving down a hill toward the bottom of a valley, *accelerated* gradient is like a large metal ball rolling down the hill, with its own mass creating momentum that propels it even when the terrain flattens out.