Gradient Descent

1 Introduction and Basic Idea

In *optimization* we have some type of objective, which is a function of a set of parameters, and our goal is to choose the parameters that optimize (minimize or maximize) the objective function. We'll suppose that we want to minimize the objective function. Training a neural network is a kind of optimization; we are trying to find the weights of the network that minimize the loss on the training data.

The most basic approach to optimization is *gradient descent*. The idea is that we begin somehow with some choice of parameters. Then we compute the gradient of the objective function with respect to the parameters. This tells us how we can change the parameters to have the biggest effect on the objective function. Then we update the parameters by changing them in the negative direction of the gradient. This will be the small, local change of the parameters that will do the most to reduce the objective function. We keep doing this until we find that we are no longer significantly reducing the objective function.

We can summarize this algorithm as follows. Let $\mathbf{x}$ denote the parameters, and let $f(\mathbf{x})$ be a function that produces a real value that is the *cost*, or the value of the objective function. Then we have:

1. Initialize $\mathbf{x_c}$.
2. Set $\mathbf{x_n} \leftarrow \mathbf{x_c} - \eta \nabla f_{\mathbf{x}}(\mathbf{x_c})$
3. If $\|f(\mathbf{x_n}) - f(\mathbf{x_c})\| < \epsilon$ stop.
4. Else $\mathbf{x_c} \leftarrow \mathbf{x_n}$ and return to step 2.

In step 1 we need to choose some initial values for the parameters. We call these $\mathbf{x_c}$, for the current parameters. This might be done by just picking some random values. This is how matconvnet initializes a neural network. Usually, we have to be a bit careful to choose these random values in an appropriate range, so we don't start somewhere ridiculous Sometimes, we can use some prior knowledge to choose the initial values so that they are likely to be near a good solution. This can make the algorithm run faster (since we don't have to go so far to get to the optimal solution) and more accurate (since we are less likely to converge to a bad local minimum. More about that later).

In step 2 we update the parameters so they move in the direction of the negative gradient. We call these $\mathbf{x_n}$, for the new parameters. Remember the gradient is the direction in which the cost function increases most rapidly, so the negative of this is the direction where the cost function decreases most rapidly. η is a parameter of the algorithm that tells us how big a step to take. This is called the *learning rate*. Often this is chosen in an ad-hoc way. Figuring out good learning rates, and perhaps adapting the learning rate as the algorithm progresses, is an interesting research area, and there is lots of work on how to do this well.

Then in step 3 we check whether this latest step is really making any progress in providing a significant improvement in the cost function. When we reach a point where

we are no longer improving, we stop. Notice that when we have found the parameters that minimize the cost function, the gradient will be zero, so we would not be moving any more. We could make step 3 more complicated in a realistic algorithm.

Let's see how this works on an example. Suppose we want to minimize the objective function:

$$f(x, y) = (x - 7)^2 + (y - 3)^2$$

Just looking at this function, we can see that it is zero when $(x, y) = (7, 3)$ and positive for any other values of x and y. So lets see how gradient descent would help us to find these values for (x, y) that minimize the cost function.

First, note that the gradient is:

$$\nabla f = (2x - 14, 2y - 6)$$

Suppose we initialize (x, y) at $(0, 0)$. Note that $f(0, 0) = 49 + 9 = 58$. Then the gradient is $\nabla f(0, 0) = (-14, -6)$. Let's suppose we use a learning rate, η of .1. Then we would set $(x, y) \leftarrow (0, 0) + .1(14, 6) = (1.4, .6)$. Since $f(1.4, .6) = 37.12$ we have reduced the cost a lot. We then continue this process.

Let's use matlab to repeat this:

```
>> xy=[0;0]; for i = 1:30 gra = [2*xy(1)-14; 2*xy(2)-6]; xy = xy - .1*gra, end

xy =

    1.4000
    0.6000


xy =

    2.5200
    1.0800


xy =

    3.4160
    1.4640


xy =

    4.1328
    1.7712


xy =
```

```
    4.7062
    2.0170


xy =

    5.1650
    2.2136


xy =

    5.5320
    2.3709


xy =

    5.8256
    2.4967


xy =

    6.0605
    2.5973


xy =

    6.2484
    2.6779


xy =

    6.3987
    2.7423


xy =

    6.5190
    2.7938
```

```
xy =

    6.6152
    2.8351


xy =

    6.6921
    2.8681


xy =

    6.7537
    2.8944


xy =

    6.8030
    2.9156


xy =

    6.8424
    2.9324


xy =

    6.8739
    2.9460


xy =

    6.8991
    2.9568


xy =

    6.9193
```

```
2.9654
```

...

We can see that we get closer and closer to the optimal values, $(7, 3)$. Do get a better idea of this, let's look at what happens when we get close, say when $(x, y) = (6, 2)$. In that case, $\nabla f(6, 2) = (-2, -2)$. So we update (x, y) to equal $(6, 2) + .1(2, 2) = (6.2, 2.2)$. Notice that as we got closer to the optimal solution, the gradient became smaller. This makes sense since at the minimum, the gradient must be zero. If it weren't there'd be a direction we could move in that would decrease the function still further. This is the same reason that for 1D functions, a maximum or minimum value only occurs when the derivative is 0.

We can also see why it's important to select a good value for η. If we had $\eta = 1$, our update would have been $(x, y) = (6, 2) + (2, 2) = (8, 4)$, and we would not have gotten any closer to the minimum. By continuiing the updates, you can see that we would just oscillate and never converge to the minimum.

2 Convexity and Local Minima

We see above that gradient descent can reduce the cost function, and can converge when it reaches a point where the gradient of the cost function is zero. An important question is when will this result in the optimal solution to the optimization function? Basically, when the cost function is *convex*, there will be a single local minimum, and any time the gradient is zero, we will be at the *global minimum* of the cost function.

One way to visualize this convexity is to think about the 3D space $(x, y, f(x, y))$ (the same ideas will hold for higher dimensions). We can think of the volume of space containing all points (x, y, z) such that $z \geq f(x, y)$. So in the example above, this volume looks like a 3D parabola. If this volume is convex, we say the problem is convex.

Why is convexity important? We will show that if a problem is convex it will only have a single local minima (by *show* we mean some very hand-wavy, intuitive explanation).

Let's consider two points, (x_1, y_1) and (x_2, y_2). We can define a line segment of points between them as all points of the form: $(x_t, y_t) = t(x_1, y_1) + (1 - t(x_2, y_2), 0 \leq t \leq 1$. Convexity implies that $f(x_t, y_t) \leq tf(x_1, y_1) + (1 - t)f(x_2, y_2)$. So if $f(x_1, y_1) > f(x_2, y_2)$ then the cost must be monotonically decreasing as we go from (x_1, y_1) to (x_2, y_2).

But suppose there are two local minima. For a point to be a local minimum of the cost function, the gradient at that point must be zero, and all points in a small neighborhood around that point must have a larger cost. So imagine that (x_1, y_1) and (x_2, y_2) are both local minima. Anywhere in the neighborhood of (x_1, y_1) the cost must be non-decreasing. This contradicts the last paragraph. Similar reasoning applies if the second point has a larger cost, or if both points have the same cost.

This means that if gradient descent converges and the cost function is convex, we have found the globally optimal solution to the problem.

However, many optimization problems are not convex. In particular, learning the weights of a neural network is a non-convex optimization problem (learning a perceptron can be formulated as a convex optimization problem, but that's another story). When a problem is nonconvex, it can have many local minima. And depending on where we initialize gradient descent, we might wind up in any of those local minima, since they are all fixed points. Furthermore, there is no general guarantee about what the value of the cost function will be at different local minima. The cost could be very low at one of them, and very high at others. So we might converge to a solution that is much worse than the best possible solution.

3 Stochastic Gradient Descent

When there are multiple local minima, gradient descent will fail to find the global minimum if we fail to initialize it sufficiently close to the global minimum. We are not exactly sure what the *energy landscape* looks like for deep networks, because the space is very high-dimensional, making it impossible to map it out numerically and hard to analyze. Not only that, but training a neural network tends to be slow, so typically we do not even run it to convergence. So we never wind up at even a local minimum, we just hope that we are approaching a good minimum when we decide to stop. But there are several approaches for avoiding bad local minima in optimization.

The first is to run gradient descent with many random starting points, and pick the solution that has the lowest energy. This way, if even one of these random starting points leads to the global minimum, we will choose that solution. This may not be too practical for neural networks, because training it even once takes a long time. But sometimes people will do this, or if they find a bunch of solutions that have similar energy, they combine the results of all of them into an ensemble. Some other approaches, such as dropout, are heuristically related to the idea of creating an ensemble of different networks.

A second approach is to use stochastic gradient descent. Here the idea is to not use the exact gradient, but use a noisy estimate of the gradient, a random gradient whose expected value is the true gradient. If we use this, after a while we are on, on average, following the gradient direction. But because we are using a noisy gradient, we can move in directions that are different from the gradient. This sometimes takes us away from a nearby local minimum, and can have the effect of preventing us from getting trapped in small local minimum.

3.1 Batch Size in Deep Networks

This fits very well with training deep networks, because the true gradient depends on all the training data, and is very expensive to compute. By computing a gradient estimate using just some of the training data, we can much more efficiently produce a noisy estimate of the gradient.

To see this, let's consider a regression cost function. Let x_i denote a vector that constitutes one example in the training set, and let y_i denote the corresponding label, which is a vector as well. For example, x_i might be a vectorized image of a face

and y_i might be a 2D vector that indicates the direction that the person is looking in (remember, direction in 3D has two degrees of freedom). Then if we have a neural network in which all the weights are described by the parameter vector θ, we can think of the network as a function that maps the input vectors to some output vectors, g_θ. Our cost is then:

$$f(\theta) = \sum_{i=1}^{n} \|g_\theta(x_i) - y_i\|^2$$

Our goal is to find the weights, θ, that minimize this cost. We will consider how to compute the gradient of this cost when we discuss backpropagation, but for now it is enough to note that to compute the gradient we must operate over the entire training set. When the training set might contain hundreds of thousands or millions of images, this much computation for each step in gradient descent is not practical.

Instead we can divide the training set into small batches (containing perhaps a hundred training examples) and compute an estimate of the gradient one batch at a time. This amounts to performing stochastic gradient descent, if the batches are randomly chosen. This leads to much more efficient optimization that also, in practice, seems to lead to better solutions.

4 Step Size and Convergence Rate

Now just a few more words on the learning rate. It can be quite difficult to determine a good learning rate, because this can be very problem dependent. Consider the cost function:

$$f(x, y) = \frac{x^2}{10} + 10y^2$$

Look what happens when we perform gradient descent on this with the same $\eta = .1$ that we used above:

```
>> xy=[5;1]; for i = 1:30 x=xy(1);y=xy(2); gra = [x/5; 20*y]; ...
xy = xy - .1*gra; [xy,gra], end

ans =

    4.9000    1.0000
   -1.0000   20.0000


ans =

    4.8020    0.9800
    1.0000  -20.0000


ans =
```

```
    4.7060    0.9604
   -1.0000   20.0000


ans =

    4.6118    0.9412
    1.0000  -20.0000


ans =

    4.5196    0.9224
   -1.0000   20.0000


ans =

    4.4292    0.9039
    1.0000  -20.0000


ans =

    4.3406    0.8858
   -1.0000   20.0000
```

The global minimum of this cost is $(0, 0)$. However, during gradient descent, the y values oscillate between 1 and -1. This is because the second derivative with respect to y is large, so as we take a step in the gradient direction, the y component of the gradient is changing very rapidly, and partway through that step we are no longer moving in the gradient direction. Meanwhile, the x component of the gradient is small, so we are not making very rapid progress in the x direction. We can solve the first problem by reducing the learning rate, but this won't necessarily help us to converge more quickly in the x direction. Try running the optimization given above with different learning rates. You can see that there can be quite a difference in performance with different learning rates, and these differences are not necessarily easy to predict. There is a good deal of research on adapting learning rates, or on searching at each step of gradient descent to determine what is the best step size. We won't consider this work right now.

We will mention that one common approach in training networks is to use *momentum*. This means that we take a step not in the current gradient direction, but in a direction that combines the current and previous recent gradient directions. You can think of this as smoothing out the gradient direction, so that we are not too influenced by our current estimate, which is very local.