CSC321 Lecture 7: Optimization

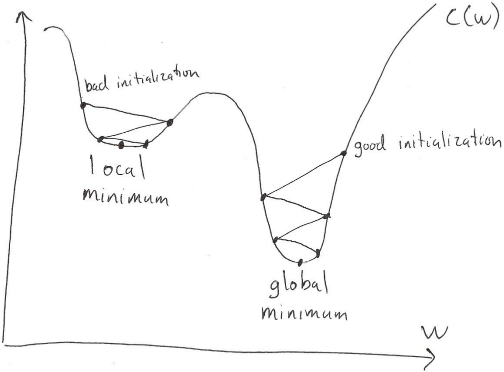
Roger Grosse

Overview

- We've talked a lot about how to compute gradients. What do we actually do with them?
- Today's lecture: various things that can go wrong in gradient descent, and what to do about them.
- Let's take a break from equations and think intuitively.
- Let's group all the parameters (weights and biases) of our network into a single vector $\boldsymbol{\theta}$.

Optimization

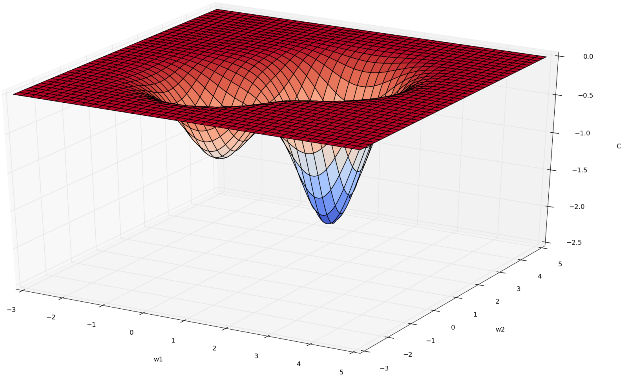
Visualizing gradient descent in one dimension: $w \leftarrow w - \epsilon \frac{\mathrm{d}C}{\mathrm{d}w}$



- The regions where gradient descent converges to a particular local minimum are called basins of attraction.

Optimization

Visualizing two-dimensional optimization problems is trickier. Surface plots can be hard to interpret:



Optimization

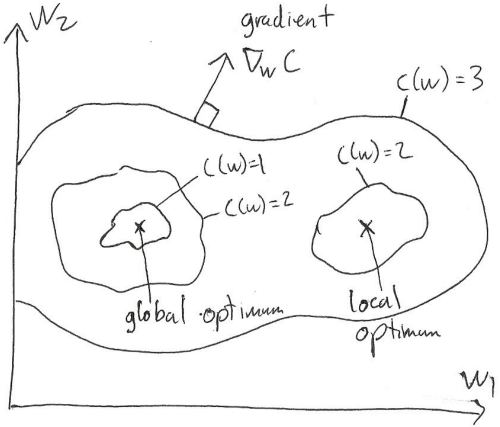
Recall:

- Level sets (or contours): sets of points on which $C(\boldsymbol{\theta})$ is constant
- Gradient: the vector of partial derivatives

$$\nabla_{\boldsymbol{\theta}} \mathcal{C} = \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}} = \left(\frac{\partial \mathcal{C}}{\partial \theta_1}, \frac{\partial \mathcal{C}}{\partial \theta_2} \right)$$

  - points in the direction of maximum increase
  - orthogonal to the level set
- The gradient descent updates are opposite the gradient direction.

Optimization



Local Minima

- Recall: convex functions don't have local minima. This includes linear regression and logistic regression.
- But neural net training is not convex!
  - Reason: if a function f is convex, then for any set of points $\mathbf{x}_1, \ldots, \mathbf{x}_N$ in its domain ,

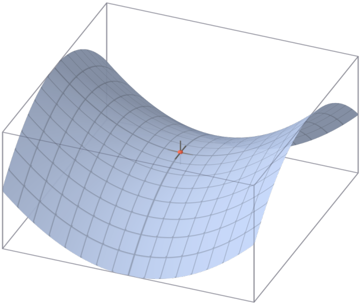
$$f(\lambda_1 \mathbf{x}_1 + \cdots + \lambda_N \mathbf{x}_N) \leq \lambda_1 f(\mathbf{x}_1) + \cdots + \lambda_N f(\mathbf{x}_N) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1.$$

  - Neural nets have a weight space symmetry: we can permute all the hidden units in a given layer and obtain an equivalent solution.
  - Suppose we average the parameters for all $K!$ permutations. Then we get a degenerate network where all the hidden units are identical.
  - If the cost function were convex, this solution would have to be better than the original one, which is ridiculous!

Local Minima

- Since the optimization problem is non-convex, it probably has local minima.
- This kept people from using neural nets for a long time, because they wanted guarantees they were getting the optimal solution.
- But are local minima really a problem?
  - Common view among practitioners: yes, there are local minima, but they're probably still pretty good.
    - Maybe your network wastes some hidden units, but then you can just make it larger.
  - It's very hard to demonstrate the existence of local minima in practice.
  - In any case, other optimization-related issues are much more important.

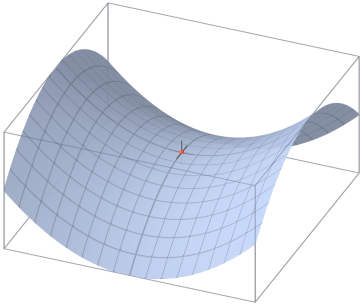
Saddle points



At a saddle point $\frac{\partial C}{\partial \boldsymbol{\theta}} = 0$, even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

When would saddle points be a problem?

Saddle points



At a saddle point $\frac{\partial C}{\partial \boldsymbol{\theta}} = 0$, even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

When would saddle points be a problem?

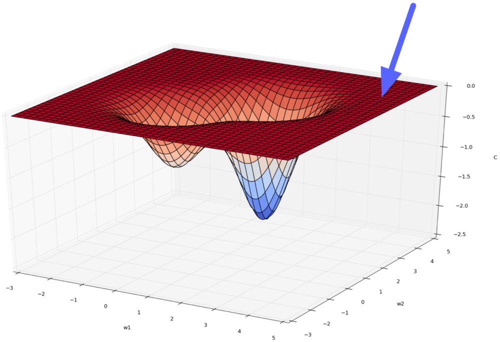
- If we're exactly on the saddle point, then we're stuck.
- If we're slightly to the side, then we can get unstuck.

Saddle points

- Suppose you have two hidden units with identical incoming and outgoing weights.
- After a gradient descent update, they will still have identical weights. By induction, they'll always remain identical.
- But if you perturbed them slightly, they can start to move apart.
- Important special case: don't initialize all your weights to zero!
    - Instead, use small random values.

Plateaux

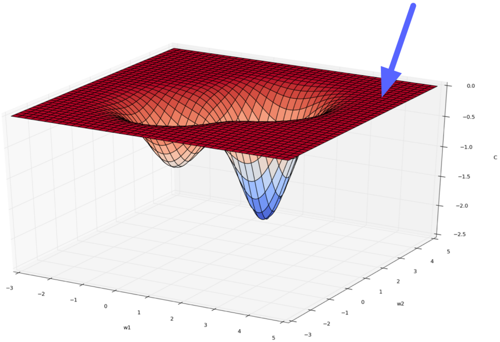
A flat region is called a plateau. (Plural: plateaux)



Can you think of examples?

Plateaux

A flat region is called a plateau. (Plural: plateaux)



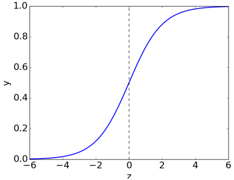
Can you think of examples?

- 0–1 loss
- hard threshold activations
- logistic activations & least squares

Plateaux

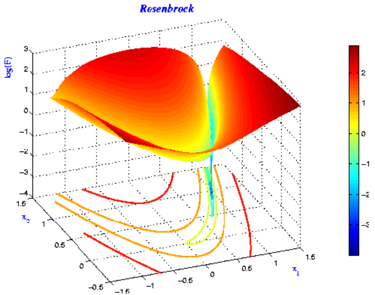
- An important example of a plateau is a saturated unit. This is when it is in the flat region of its activation function. Recall the backprop equation for the weight derivative:

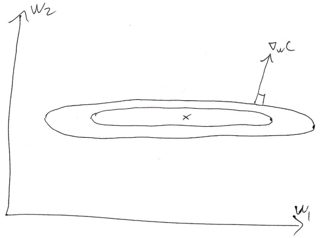
$$\overline{z_i} = \overline{h_i}\,\phi'(z)$$
$$\overline{w_{ij}} = \overline{z_i}\,x_j$$



- If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be *exactly* 0. We call this a dead unit.

Long, narrow ravines:





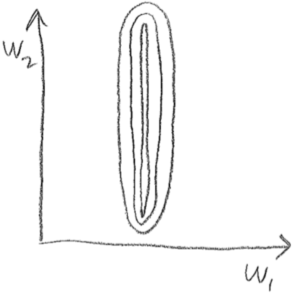
Lots of sloshing around the walls, only a small derivative along the slope of the ravine's floor.

Ravines

- Suppose we have the following dataset for linear regression.

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
$\vdots$	$\vdots$	$\vdots$

$$\overline{w_i} = \overline{y}\, x_i$$

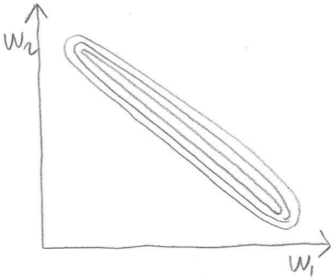


- Which weight, w_1 or x_1, will receive a larger gradient descent update?
- Which one do you want to receive a larger update?
- Note: the figure vastly *understates* the narrowness of the ravine!

Ravines

- Or consider the following dataset:

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
⋮	⋮	⋮



Ravines

- To avoid these problems, it's a good idea to center your inputs to zero mean and unit variance, especially when they're in arbitrary units (feet, seconds, etc.).
- Hidden units may have non-centered activations, and this is harder to deal with.
  - One trick: replace logistic units (which range from 0 to 1) with tanh units (which range from -1 to 1)
  - A recent method called **batch normalization** explicitly centers each hidden activation. It often speeds up training by 1.5-2x, and it's available in all the major neural net frameworks.

Momentum

- Unfortunately, even with these normalization tricks, narrow ravines will be a fact of life. We need algorithms that are able to deal with them.
- Momentum is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$\mathbf{p} \leftarrow \mu \mathbf{p} - \alpha \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?

Momentum

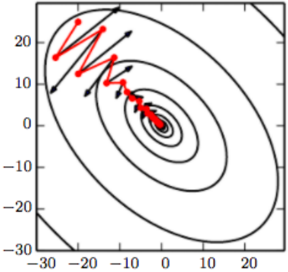
- Unfortunately, even with these normalization tricks, narrow ravines will be a fact of life. We need algorithms that are able to deal with them.
- Momentum is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$\mathbf{p} \leftarrow \mu \mathbf{p} - \alpha \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?
  - If $\mu = 1$, conservation of energy implies it will never settle down.

Momentum

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.
- In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.



- If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of

$$\frac{\alpha}{1-\mu} \cdot \frac{\partial \mathcal{C}}{\partial \boldsymbol{\theta}}$$

This suggests if you increase μ, you should lower α to compensate.

- Momentum sometimes helps a lot, and almost never hurts.

Ravines

- Even with momentum and normalization tricks, narrow ravines are still one of the biggest obstacles in optimizing neural networks.
- Empirically, the curvature can be many orders of magnitude larger in some directions than others!
- An area of research known as **second-order optimization** develops algorithms which explicitly use curvature information, but these are complicated and difficult to scale to large neural nets and large datasets.
- There is an optimization procedure called **Adam** which uses just a little bit of curvature information and often works much better than gradient descent. It's available in all the major neural net frameworks.

Stochastic Gradient Descent

- So far, the cost function $\mathcal{E}$ has been the average loss over the training examples:

$$\mathcal{E}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}^{(i)} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y(\mathbf{x}^{(i)}, \boldsymbol{\theta}), t^{(i)}).$$

- By linearity,

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}.$$

- Computing the gradient requires summing over *all* of the training examples. This is known as batch training.
- Batch training is impractical if you have a large dataset (e.g. millions of training examples)!

Stochastic Gradient Descent

- Stochastic gradient descent (SGD): update the parameters based on the gradient for a single training example:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}$$

- SGD can make significant progress before it has even looked at all the data!
- Mathematical justification: if you sample a training example at random, the stochastic gradient is an unbiased estimate of the batch gradient:

$$\mathbb{E}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}\right] = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}}.$$

- Problem: if we only look at one training example at a time, we can't exploit efficient vectorized operations.

Stochastic Gradient Descent

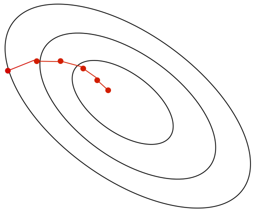
- Compromise approach: compute the gradients on a medium-sized set of training examples, called a mini-batch.
- Stochastic gradients computed on larger mini-batches have smaller variance:

$$\mathrm{Var}\left[\frac{1}{S}\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S^2}\,\mathrm{Var}\left[\sum_{i=1}^{S}\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right] = \frac{1}{S}\,\mathrm{Var}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}\right]$$

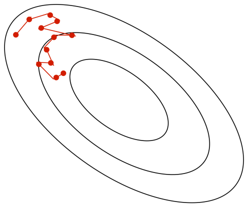
- The mini-batch size S is a hyperparameter that needs to be set.
  - Too large: takes more memory to store the activations, and longer to compute each gradient update
  - Too small: can't exploit vectorization
  - A reasonable value might be $S = 100$.

Stochastic Gradient Descent

- Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.



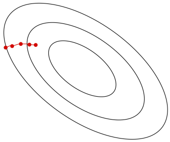
batch gradient descent



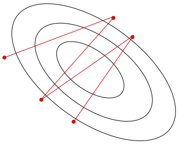
stochastic gradient descent

Learning Rate

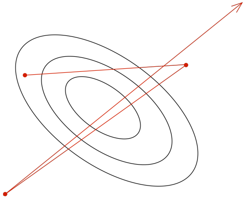
- The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong in batch mode:



α too small: slow progress



α too large: oscillations



α much too large: instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).

Learning Rate

- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.
- By reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly. But this can come at the expense of long-run performance.