Optimization for Training Deep Models

Chapter 8

Why to optimize log likelihood instead of 0-1 loss

“For example, the test set 0-1 loss often continues to decrease for a long time after the training set 0-1 loss has reached zero, when training using the log-likelihood surrogate. This is because even when the expected 0-1 loss is zero,one can improve the robustness of the classifier by further pushing the classes apart from each other, obtaining a more confident and reliable classifier, thus extracting more information from the training data than would have been possible by simply minimizing the average 0-1 loss on the training set.”

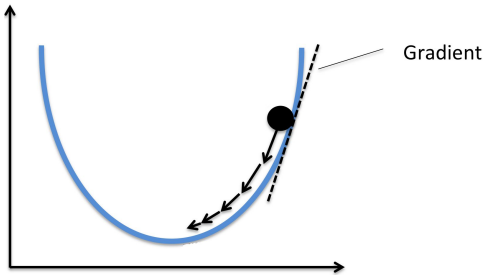
Minibatches

We want to minimize the loss over the whole training set

But the gradient based on a random subset of the training data is the same (on average) as the gradient based on the whole training set

Using a minibatch:

- The computation per step is smaller, so you can take more (smaller) steps
- The direction of each step is almost as good as the full training set



Minibatches

How to set the batch size?

- Sub-linear returns with larger batches [sqrt(n)]
- However, CPUs (and especially GPUs) are optimized to do the same calculation in parallel on multiple bits of data
- Some optimization methods (for example in second order methods where the inverse hessian is computed) require larger batches to avoid instability

Very important that the batches are chosen randomly

- Becomes an issue in online learning (e.g., a robot)
- AlphaGo solution: draw randomly from all past experience, not just the most recent experience
- Analogy to replay?

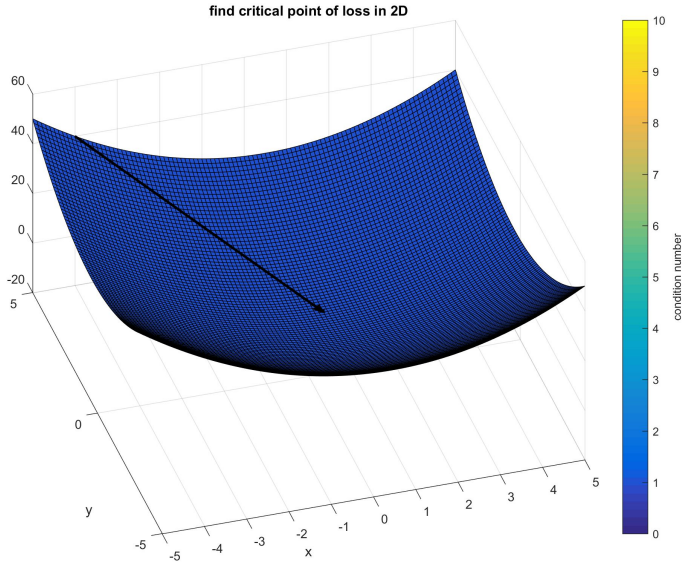
2nd-order optimization and ill-conditioned hessians

As we've discussed before, including information about the 2nd derivative can hugely speed up optimization

For example, here the surface is a quadratic, so a single step from anywhere in the world jumps directly to the global minimum.

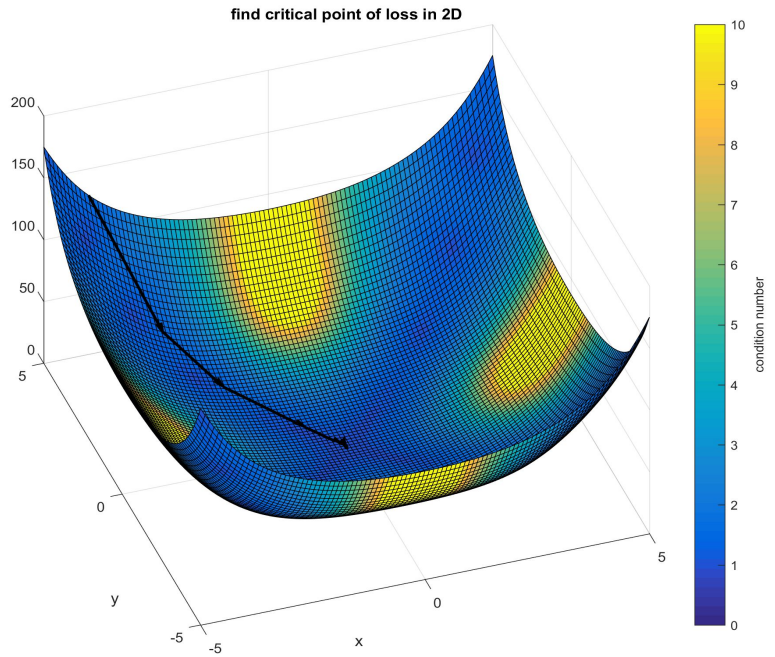
The weights are updated by this amount:

- gradient * inv(hessian)



2nd-order optimization and ill-conditioned hessians

Even if the surface isn’t quadratic, a few steps are often enough to reach the solution:

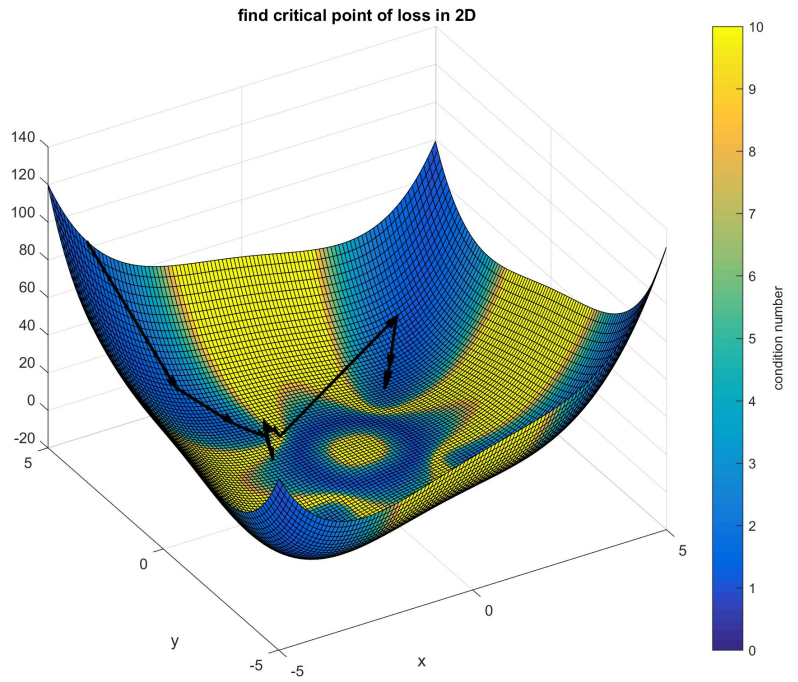


2nd-order optimization and ill-conditioned hessians

However, watch out for when one or more eigenvalues of the hessian are near-zero

(Which means the surface is very flat in one or more cardinal directions)

Then, the inverse of the hessian will be wacky big, and the update can go anywhere



Local minima

Does everyone know the difference between convex and non-convex problems?

An “identifiable” model is one with a unique solution

- But in a neural network, there are lots of ways to get an equivalent solution
- For example, you can scale some weights up, and others down
- So, there can be a huge number of local minima corresponding to these equivalent solutions, but we’re happy to find any of them

If you think your optimization is getting stuck in a local minimum, try plotting the gradient over time

- You may find that it’s not decreasing → not a local minimum

Local minima

Actual local minima in some kinds of functions tend to be very good solutions, because the goodness of the solution is related to how many positive eigenvalues there are

I guess it’s not completely understood if neural networks are in this class of function

Saddle points

More common in high dimensional problems

This may explain why second order methods have not worked well in neural networks

Cliffs

Narrow regions with very steep gradients can arise from multiplication of large weights, combined with a nonlinear activation function (e.g. rectification)

If you compute the gradient here, you'll jump a million miles away, even though the best solution might be right at the bottom of the cliff

One solution is to put a hard cap on step size that you can ever take

- Remember the gradient doesn't specify how far you should do, just the direction

Vanishing and exploding gradients in recurrent nets

Say you have a weight matrix W with an eigendecomposition $V \cdot \lambda \cdot V^{-1}$

You multiply by this matrix n times, so you've multiplied by W^n

But, $W^n = V \cdot \lambda^n \cdot V^{-1}$

So as n becomes very large, any eigenvalues less than 1 approach 0, and any eigenvalues greater than 1 approach infinity

Decreasing the learning rate over time

Stochastic gradient descent (using minibatches) will keep jumping around even if it gets to a minimum

- Because the “local minima” defined by each minibatch could actually be in very different places

So, it’s a good idea to start taking very small steps as you get close

- Unfortunately, it’s hard to know when this is
- The book suggests some rules of thumb

Another idea is to gradually increase the minibatch size over time

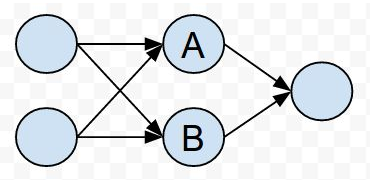
Momentum

The noisiness of stochastic gradient descent can also be combated by adding “momentum”, where each minibatch changes your velocity like a “force”

But I don’t understand how this is basically different from using larger minibatches…?

Initialization - breaking symmetry

Imagine a small network like this:



If A and B start with the same values, they may stay “stuck together”

Good solution: random initialization

- Interestingly, if you have early stopping, this is like a prior that the solution is near the initialization. Thus, initializing near zero is like regularization.

Often good to initialize biases near zero, unless you know something about the output distribution

Adaptive learning rate

Commonly used algorithms like RMSProp

- Weights that have recently (in terms of minibatches) had small partial derivatives get relatively faster learning rates
- The idea is to go more quickly through flat areas
- I didn’t understand this, though. In the limit, would this reduce to moving around in a grid-like way? Does this defeat the purpose of the gradient?

No consensus on best algorithm

Approximate second-order methods

Newton’s method jumps right to the minimum if the problem is convex and quadratic

- However, if there are saddle points, Newton’s method can move in the wrong direction
- Also, the Hessian is very expensive to calculate and invert (computation grows with the 3rd power of the number of weights!)

Conjugate gradient and BFGS are methods for getting some 2nd-order information without calculating the inverse Hessian

- BFGS is used in fminunc
- BFGS uses too much memory for large neural networks, so a variant is used

Batch normalization

Gradient for each weight assumes the other weights won’t change

But distribution of activations can move a lot over multiple layers

Z-score the activations of each unit, over data points within a minibatch

Also avoids getting stuck in a sigmoidal nonlinearity

Design choices

Just like in the 1980s, we still use stochastic gradient descent with momentum

So what changed?

Seemingly less-important design details (like ReLU and others) have often shifted more of the architecture to be linear, which means the local gradients tend to point toward a distant solution

Curriculum learning

Start by teaching the network something “easier”, and progress to learning harder data (e.g. start by learning about basic shapes before learning compositions of them)

Related to “continuation methods”, where you first try to learn a simpler version of the loss function (e.g., a smoothed version)