Lecture 6: Optimization

CS 182/282A (“Deep Learning”)

2022/02/07

Today's lecture

- So far in lecture, we have built up a simple neural network model, and we have defined our negative log likelihood (cross-entropy) loss function
- We saw last week two different ways to think about computing gradients of the loss function with respect to the model parameters: backprop and autodiff
- We have also seen the basic idea behind gradient based optimization
- Today, we will complete our story on optimization, flesh out gradient based optimization in detail, and describe how neural networks are trained in practice

A hand-wavy overview of optimization

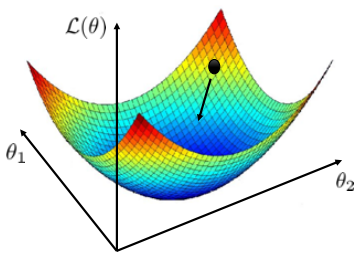
Remember: gradient based optimization

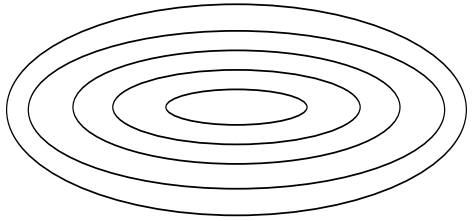
- Deep learning relies on **iterative optimization** to find good parameters
  - Starting from an initial “guess”, continually refine that guess until we are satisfied with our final answer
- By far the most commonly used set of iterative optimization techniques in deep learning is (first order) gradient based optimization and variants thereof
- Move the parameters in the direction of the *negative gradient* of the average loss:

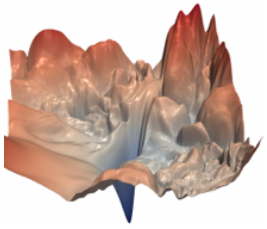
  $\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$ — we refer to α as a **step size** or **learning rate**

Visualizing losses and optimization

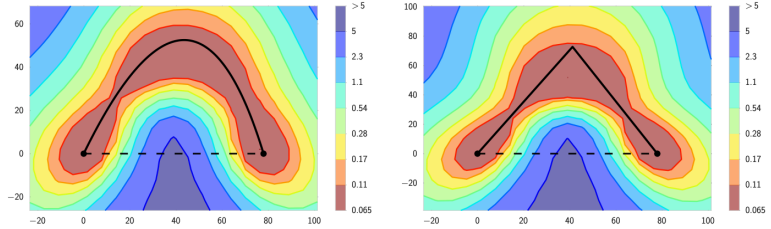
- Optimization is hard to visualize for any more than two parameters
  - But neural networks have thousands, millions, billions of parameters…
  - For visualization purposes, we will pretend they have two
- Some works have explored interesting ways to visualize loss “landscapes”







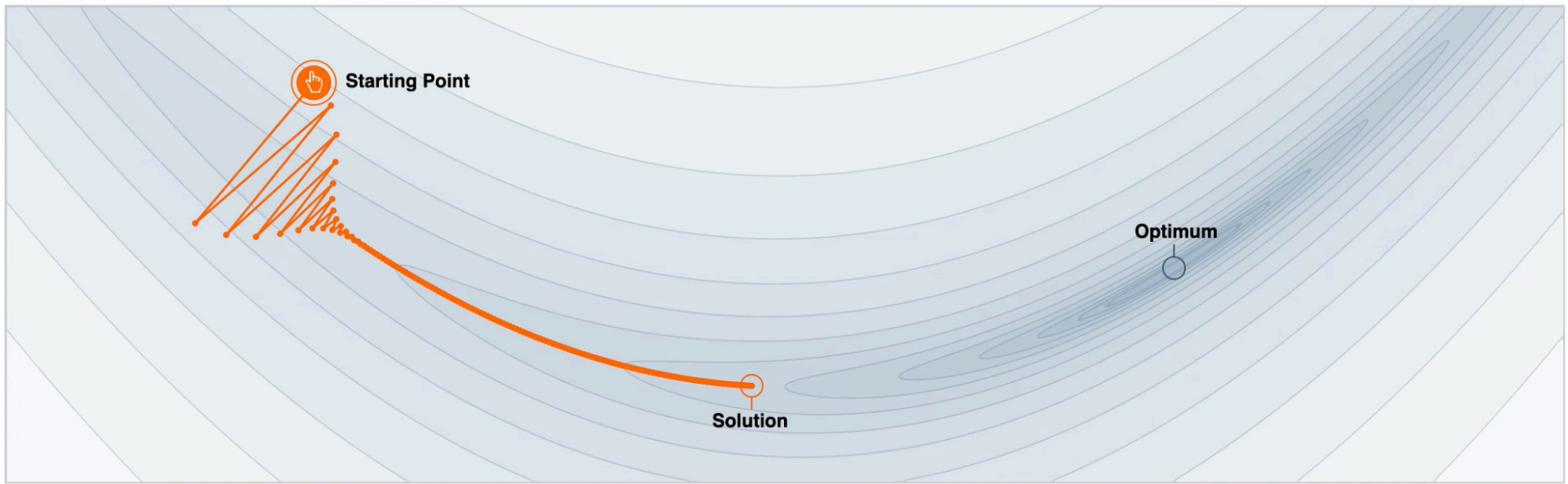
Li et al, NIPS ’18



Garipov et al, NIPS ‘18

Visualizing gradient descent

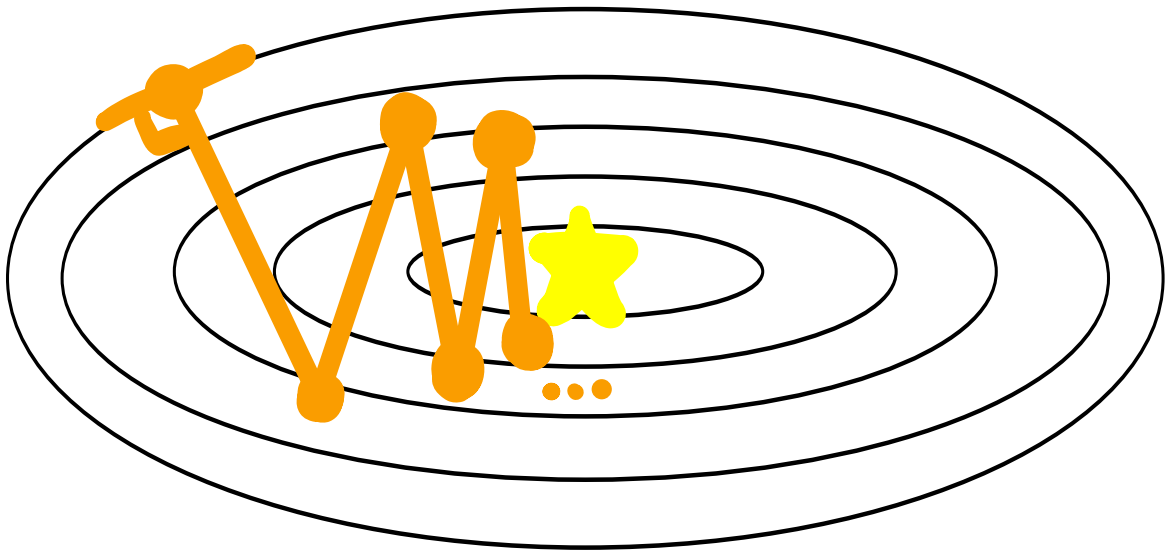
https://distill.pub/2017/momentum/



Step-size α = 0.0030

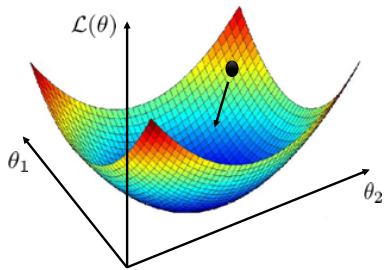
0 0.003 0.006

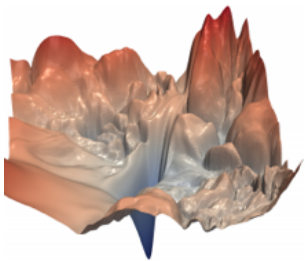
What's going on with gradient descent?



So… optimization is really hard?

- Even for the previous *convex, well conditioned* optimization problem, we are not blown away by gradient descent's performance
- Do we really have any hope of applying this to train neural networks?

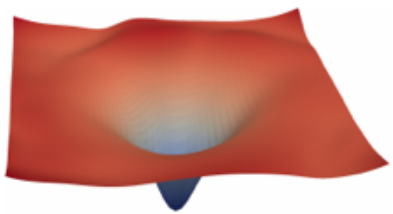




Li et al, NIPS '18

What makes neural network training possible?

- Do we really have any hope of using gradient descent to train neural networks?
- Yes! Because of a few reasons:
  - We have methods that work better than vanilla gradient descent
  - Some neural network architectures result in easier optimization — a topic for future lectures
  - In practice, we don't really care about reaching the global optimum
  - Actually, we don't really care about reaching *any* optimum…



Li et al, NIPS '18

An aside: critical points

- Critical points, in our setting, occur when $\nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i) = 0$
- The global optimum is a critical point, but critical points could also be:
  - A local optimum! Turns out, though, that these are often quite good too
  - A plateau or saddle point! Turns out that we don't really worry about these
- For neural network training, we have bigger practical concerns than what type of critical point we have reached — we don't usually reach one in the first place!

Practical neural network training

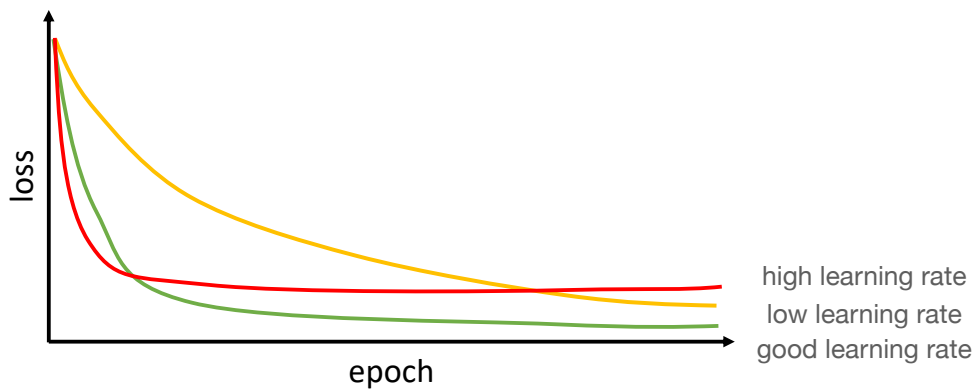
Stochastic optimization

Or “stochastic gradient descent (SGD)”, colloquially

- Computing $\nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$ every iteration for large N (think 1 million) is a bad idea
  - Instead, we pick a **batch size** (or **mini batch size**) $B \ll N$, we randomly sample $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_B, y_B)\}$ from the training data, and we compute $\nabla_\theta \frac{1}{B} \sum_{i=1}^{B} \ell(\theta; \mathbf{x}_i, y_i)$
- Sampling the mini batch i.i.d. is rather slow due to random memory accesses
  - Instead, we *shuffle* the dataset and construct mini batches from consecutive data points
  - After each pass on the training data (called an **epoch**), we reshuffle

Learning rates and learning curves

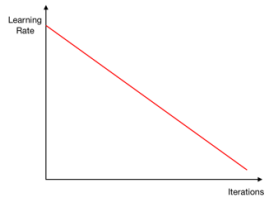
- Learning curves plot loss values (or something related) over the course of training
- What might our learning curves look like for different learning rates α?
- Too low may “stop learning” too early, too high may cause oscillation/divergence



Does the learning rate have to be constant?

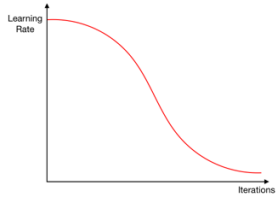
- Commonly, a learning rate **schedule** will be used rather than a constant
- **Linear decay** decreases the learning rate a constant amount each iteration:

$$\alpha_i = \alpha_{\text{initial}} \cdot \left(1 - \frac{i}{\text{max\_steps}}\right)$$



- **Cosine annealing** decays the learning rate according to:

$$\alpha_i = \alpha_{\text{initial}} \cdot 0.5 \cdot \left[1 + \cos\left(\pi \cdot \frac{i}{\text{max\_steps}}\right)\right]$$



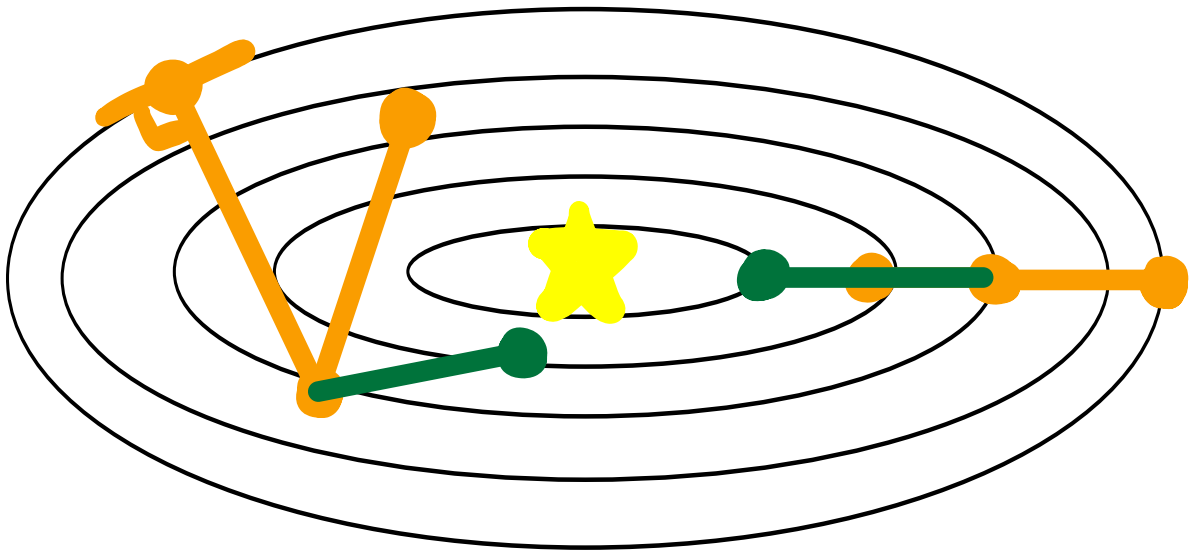
- For large α_{initial}, there may also be a **linear warmup** for the first few epochs

Summary

- For practical neural network training, we:
  - Pick a mini batch size B — this is usually limited by memory
  - Pick a learning rate α_{initial} and a learning rate schedule (and maybe a warmup)
  - Pick a maximum number of iterations to train (though we may stop early)
- How do we pick all of these things?
  - Training loss (empirical risk) can diagnose underfitting (poor optimization), validation loss (true risk estimate) can diagnose overfitting (poor generalization)

Beyond vanilla gradient descent

What's going on with gradient descent?

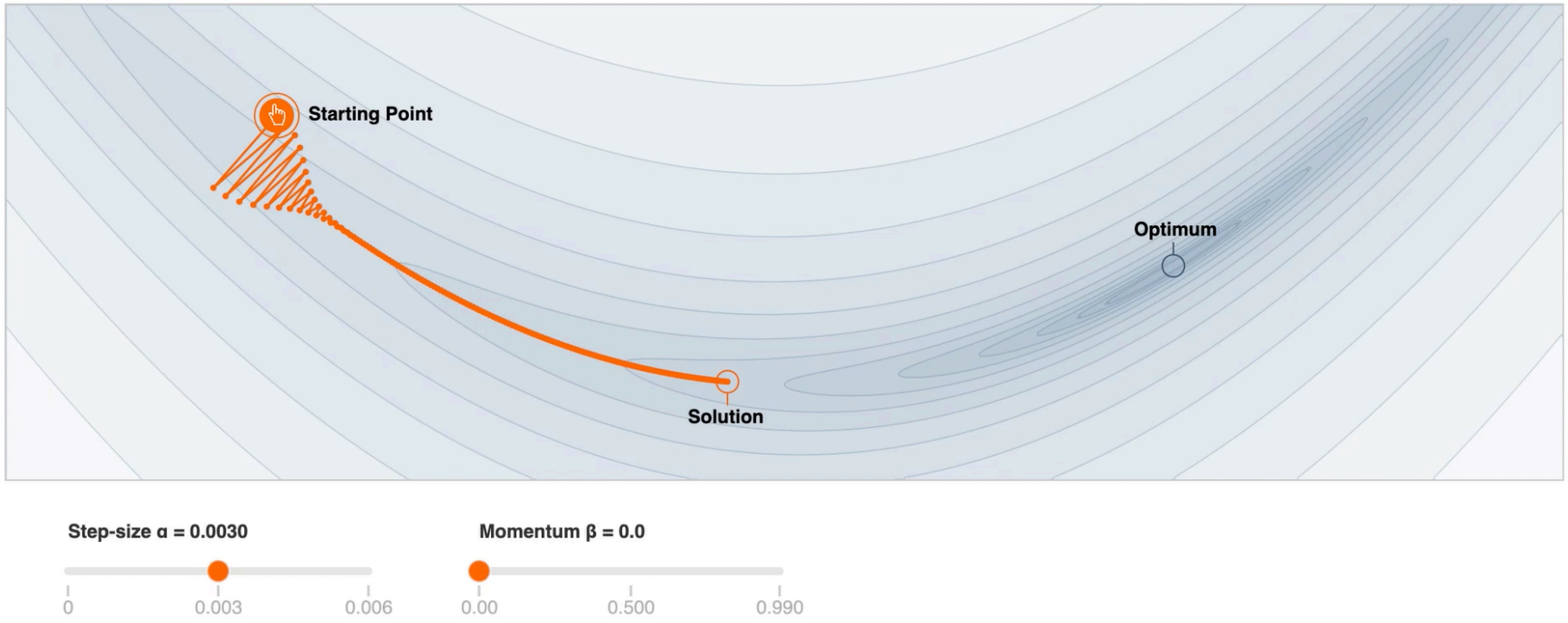


Momentum

- Intuitively, we want the optimization to “remember” the gradient steps it has taken
- We do so by modifying the update rule: $\theta \leftarrow \theta - \alpha \mathbf{g}$
  - Before, $\mathbf{g} = \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i)$; now, $\mathbf{g} \leftarrow \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i) + \mu \mathbf{g}$
- By “blending in” previous gradients, we avoid some of the aforementioned issues
- This is an example of an *exponential moving average* — gradients further in the past have exponentially less weight

Visualizing momentum

https://distill.pub/2017/momentum/



Nesterov's accelerated gradient

- Nesterov's accelerated gradient is another optimization approach which enjoys interesting theoretical guarantees on some problems
- It can be interpreted as a variant on the momentum approach we described
- We still have $\theta \leftarrow \theta - \alpha \mathbf{g}$; before, $\mathbf{g} \leftarrow \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta; \mathbf{x}_i, y_i) + \mu \mathbf{g}$
  - Now, $\mathbf{g} \leftarrow \nabla_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta - \alpha\mu \mathbf{g}; \mathbf{x}_i, y_i) + \mu \mathbf{g}$
- The common implementation does not look like this equation, but it is equivalent

Gradient directions vs. magnitudes

- The *sign* of the gradient is useful for telling us which direction to move in
- Oftentimes, however, the *magnitude* of the gradient is not as useful/trustworthy
  - We may have loss landscapes that are not sufficiently smooth
  - Gradient magnitudes also tend to start out large and end up very small
- As it turns out, “normalizing” the gradient magnitudes along each dimension (separately for each parameter) can lead to an effective optimization strategy

Adam

basic idea: combine momentum with a *second moment adjustment*

$\theta \leftarrow \theta - \alpha g$ what is g?

momentum: $m \leftarrow (1-\beta_1)\nabla_\theta \ell + \beta_1 m$

second moment estimate: $v \leftarrow (1-\beta_2)(\nabla_\theta \ell)^2 + \beta_2 v$

detail - bias correction: $\hat{m} = m/(1-\beta_1^t)$

$\hat{v} = v/(1-\beta_2^t)$

$g = \hat{m}/(\sqrt{\hat{v}} + \epsilon)$

What's so great about Adam?

- Empirically, Adam seems to work well "out of the box" for many neural networks
- It combines momentum with a cheap approximation of second order information — actual second order methods like *Newton's method* are far too expensive
  - There's also some relationship to methods which "adapt" the learning rate separately for each parameter — *AdaGrad* and *RMSProp*
- The important takeaway: when tackling a new deep learning problem, most people will try both stochastic gradients with momentum and Adam
  - Hopefully at least one of them does well…

Weight decay vs. ℓ_2-regularization

- Remember that adding $\lambda ||\theta||_2^2$ to the loss function is ℓ_2-regularization
- Sometimes (somewhat erroneously) referred to as **weight decay**
  - Weight decay is actually an extra step in the optimization: after taking a gradient step, we do $\theta \leftarrow (1 - \lambda)\theta$ (*shrinking* the parameters toward zero)
- For stochastic gradients, ℓ_2-regularization and weight decay are the same
- Not true for Adam! We can consider Adam either with ℓ_2-regularization or with weight decay (typically referred to as the **AdamW** optimizer)

Tuning the optimization

- What hyperparameters do we have? Already discussed: B, max # iterations, etc.
- α_{initial}: 0.001 is a good number to start from, but this usually requires tuning
  - A useful (and surprising!) rule-of-thumb: if some α_{initial} is good for some B, then $k\alpha_{\text{initial}}$ is often a good value for kB
  - These days, people are often *fine tuning* large *pretrained* models using small α_{initial}
- $\mu = 0.9$ is a good default value for momentum, often doesn't require tuning
- $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ for Adam *usually* don't require tuning!