What’s so hard about optimizing deep networks?

Benjamin Recht
University of California, Berkeley
What makes optimization of deep models hard?

Recent abstracts on arXiv:

“We prove that recovering the global minimum becomes harder as the network size increases.” arXiv:1412.0233

“Difficulty originates from the proliferation of saddle points, not local minima, especially in high dimensional problems of practical interest.” arXiv:1406.2572
It’s hard to hit a saddle

\[ f(x) = \frac{1}{2} \sum_{i=1}^{d} a_i x_i^2 \]

Gradient descent:

\[ x_i^{(k+1)} = (1 - t a_i) x_i^{(k)} \]

After k steps

\[ x_i^{(k)} = (1 - t a_i)^k x_i^{(0)} \]

If \( t |a_i| < 1 \) \{ converges to 0 if all \( a_i \) are positive

\[ \text{diverges almost surely if single } a_i \text{ is negative} \]
It’s hard to hit a saddle

\[ f(x, y) = xy \]

If you are not on the line \( \{x=-y\} \), you diverge at an exponential rate.

This picture fully generalizes to the nonconvex case.

Thm: [Lee et al, 2016] For the short-step gradient method, the basin of attraction of strong saddle points has measure zero.

Simple consequence of the Stable Manifold Theorem (Smale et al)
This is our fault, optimizers.

- Too many fragile examples in text books
- Minor perturbations in initial conditions always repel you from saddles.

\[ f(x, y) = x_1^4 - 2x_1^2 + x_2^2 \]
Flatness is what makes things hard

- In convex-land, flat directions (ill-conditioning) slow algorithms down.
- What happens in nonconvex-land?
Flatness is what makes things hard

- What happens in nonconvex-land?

\[ f(x) = \sum_{i,j=1}^{d} Q_{ij} x_i^2 x_j^2 \]

\[ \nabla f(0) = 0 \quad \text{Is 0 a global min, saddle, or global max?} \]

\[ \nabla^2 f(0) = 0 \quad \text{f is super flat at 0.} \]

Deciding if there is a descent direction at 0 is NP-complete
$f(x) = \sum_{i,j=1}^{d} Q_{ij} x_i^2 x_j^2$

$Q = I - A + s \cdot 11^T$

A = adjacency matrix of G

G has an clique of size larger than $1/(1-s)$ if and only if $0$ is not a local minimizer*.

Thm [Barak et al. 2016]: Finding a maximum clique is F-hard

Is deep learning as hard as maximum clique?

**Thm: [Soudry and Carmon 2016]**: For an $L$ layer neural network trained with dropout and with $n < d_{L-2}d_{L-1}$, any stable local minimum is a global minimum with loss 0 almost surely.
How can you get to zero with constant stepsize?

If there is a solution where all gradients vanish, constant stepwise converges linearly.

Is there something special about your initialization?

If you have more parameters than unknowns, why would you expect to converge to a saddle?

What does the test error look like?
Avoiding overfitting is hard.

- This is true in the convex case too!

\[ \text{minimize } \| y - \Phi x \|^2 \]

- \( \Phi \) \( n \times p, n<p \)

- Infinite number of global minima. Which one should we pick?

- Regularize to leverage structure.

---

Sparsity  Rank  Smoothness  Architecture

Can/should we directly minimize overfitting?
- penalty
- \[ \eta \in [10^{-3}, 10^{-1}] \]
- \( \ell_2 \)-penalty \( \lambda \in [10^{-6}, 10^{-1}] \)
- \# hidden nodes \( N_{hid} \in [10^1, 10^3] \)
Training set

$N_{\text{in}} = 784$
$N_{\text{out}} = 10$

$N_{hid} = 784$

Hyperparameters

$\eta \in [10^{-3}, 10^{-1}]$
$\lambda \in [10^{-6}, 10^{-1}]$

# hidden nodes $N_{hid} \in [10^1, 10^3]$

Eval-set

Eval-loss

0.0577
How do we choose hyperparameters to train and evaluate?

<table>
<thead>
<tr>
<th>Hyperparameters</th>
<th>Eval-loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-1.6}, 10^{-2.4}, 10^{1.7}$</td>
<td>0.0577</td>
</tr>
<tr>
<td>$10^{-1.0}, 10^{-1.2}, 10^{2.6}$</td>
<td>0.182</td>
</tr>
<tr>
<td>$10^{-1.2}, 10^{-5.7}, 10^{1.4}$</td>
<td>0.0436</td>
</tr>
<tr>
<td>$10^{-2.4}, 10^{-2.0}, 10^{2.9}$</td>
<td>0.0919</td>
</tr>
<tr>
<td>$10^{-2.6}, 10^{-2.9}, 10^{1.9}$</td>
<td>0.0575</td>
</tr>
<tr>
<td>$10^{-2.7}, 10^{-2.5}, 10^{2.4}$</td>
<td>0.0765</td>
</tr>
<tr>
<td>$10^{-1.8}, 10^{-1.4}, 10^{2.6}$</td>
<td>0.1196</td>
</tr>
<tr>
<td>$10^{-1.4}, 10^{-2.1}, 10^{1.5}$</td>
<td>0.0834</td>
</tr>
<tr>
<td>$10^{-1.9}, 10^{-5.8}, 10^{2.1}$</td>
<td>0.0242</td>
</tr>
<tr>
<td>$10^{-1.8}, 10^{-5.6}, 10^{1.7}$</td>
<td>0.029</td>
</tr>
</tbody>
</table>
Bayesian Optimization

Very popular for hyperparameter tuning

Recent abstracts on arXiv:

“Bayesian optimization has become a successful tool for hyperparameter optimization of machine learning algorithms, such as support vector machines or deep neural networks.” arXiv:1605.07079

“Bayesian optimization is an elegant solution to the hyperparameter optimization problem in machine learning.” arXiv:1605.06170

“Bayesian optimization provides a principled way for searching optimal hyperparameters for a single algorithm.” arXiv:1602.06468

“finds global optima significantly faster than previous batch Bayesian optimization algorithms ... when tuning hyperparameters of practical machine learning algorithms” arXiv:1606.04414
Hyperparameters

- \((10^{-1.6}, 10^{-2.4}, 10^{1.7})\)
- \((10^{-1.0}, 10^{-1.2}, 10^{2.6})\)
- \((10^{-1.2}, 10^{-5.7}, 10^{1.4})\)
- \((10^{-2.4}, 10^{-2.0}, 10^{2.9})\)
- \((10^{-2.6}, 10^{-2.9}, 10^{1.9})\)
- \((10^{-2.7}, 10^{-2.5}, 10^{2.4})\)
- \((10^{-1.8}, 10^{-1.4}, 10^{2.6})\)
- \((10^{-1.4}, 10^{-2.1}, 10^{1.5})\)
- \((10^{-1.9}, 10^{-5.8}, 10^{2.1})\)
- \((10^{-1.8}, 10^{-5.6}, 10^{1.7})\)

Eval-loss

- 0.0577
- 0.182
- 0.0436
- 0.0919
- 0.0575
- 0.0765
- 0.1196
- 0.0834
- 0.0242
- 0.029

How computation time was spent?
Black-box solver stopped short due to lack of progress
Hyperparameters:
\((10^{-1.6}, 10^{-2.4}, 10^{1.7})\)
\((10^{-1.0}, 10^{-1.2}, 10^{2.6})\)
\((10^{-1.2}, 10^{-5.7}, 10^{1.4})\)
\((10^{-2.4}, 10^{-2.0}, 10^{2.9})\)
\((10^{-2.6}, 10^{-2.9}, 10^{1.9})\)
\((10^{-2.7}, 10^{-2.5}, 10^{2.4})\)
\((10^{-1.8}, 10^{-1.4}, 10^{2.6})\)
\((10^{-1.4}, 10^{-2.1}, 10^{1.5})\)
\((10^{-1.9}, 10^{-5.8}, 10^{2.1})\)
\((10^{-1.8}, 10^{-5.6}, 10^{1.7})\)

Eval-loss:
0.0577
0.182
0.0436
0.0919
0.0575
0.0765
0.1196
0.0834
0.0242
0.029

But many of iterations wasted on “losers”
Can we identify and kill “losers” early and focus on “winners”?

Hyperparameters

- $(10^{-1.6}, 10^{-2.4}, 10^{1.7})$
- $(10^{-1.0}, 10^{-1.2}, 10^{2.6})$
- $(10^{-1.2}, 10^{-5.7}, 10^{1.4})$
- $(10^{-2.4}, 10^{-2.0}, 10^{2.9})$
- $(10^{-2.6}, 10^{-2.9}, 10^{1.9})$
- $(10^{-2.7}, 10^{-2.5}, 10^{2.4})$
- $(10^{-1.8}, 10^{-1.4}, 10^{2.6})$
- $(10^{-1.4}, 10^{-2.1}, 10^{1.5})$
- $(10^{-1.9}, 10^{-5.8}, 10^{2.1})$
- $(10^{-1.8}, 10^{-5.6}, 10^{1.7})$

Eval-loss

- $0.0577$
- $0.182$
- $0.0436$
- $0.0919$
- $0.0575$
- $0.0765$
- $0.1196$
- $0.0834$
- $0.0242$
- $0.029$
Sparks and Talwalkar heuristic...
Sparks and Talwalkar heuristic...
Sparks and Talwalkar heuristic...
Sparks and Talwalkar heuristic...
**HyperBand**

Input: `max_iter`

for `s = log_3(max_iter), \ldots, 1, 0`:

\[ n = 3^s \frac{\log_3(n_{max_iter} + 1)}{s+1}, \quad r = \text{max}_\text{iter} 3^{-s} \]

batch = [ get_hyperparameter_config() for i=1, \ldots, n ]

for `i = 0, \ldots, log_3(max_iter/r)`:

\[ n_i = n 3^{-i}, \quad r_i = r 3^i \]

for params in batch:

run_and_get_val_loss(config=params, iters= r_i)

Throw out worst 2/3 `n_i` configurations

Return remaining configs in batch

---

**HyperBand**

\[ n > \frac{B}{(\log_3(max_iter)+1)^2} \]

**Uniform Allocation**

\[ n = \frac{B}{max_iter} \]
“The balance between theory and practice in nonlinear programming is particularly delicate, subjective, and problem dependent” - D. Bertsekas
References

- argmin.net