CSC 2541: Neural Net Training Dynamics
Lecture 7 - Stochasticity and Parallelism

Roger Grosse

University of Toronto, Winter 2022
What’s different about SGD?

Empirical observations about stochasticity and parallelism
  - Evidence for small batch (noise-dominated) and large batch (curvature-dominated) regimes

Two models of stochastic optimization (which make very different predictions!)
  - Noisy quadratic model (NQM)
  - Interpolation regime / student-teacher
Stochastic Gradient Descent

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

\[ \theta \leftarrow \theta - \alpha \nabla J^{(i)}(\theta) \]

- 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:

\[ \mathbb{E}_i \left[ \nabla J^{(i)}(\theta) \right] = \frac{1}{N} \sum_{i=1}^{N} \nabla J^{(i)}(\theta) = \nabla J(\theta). \]
Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.
**Problem:** if we only look at one training example at a time, we can’t exploit efficient vectorized operations.

**Compromise approach:** compute the gradients on a medium-sized set of training examples, called a mini-batch.

Each entire pass over the dataset is called an epoch.

Stochastic gradients computed on larger mini-batches have smaller variance:

\[
\text{Var} \left[ \frac{1}{S} \sum_{i=1}^{S} \frac{\partial L^{(i)}}{\partial \theta_j} \right] = \frac{1}{S^2} \text{Var} \left[ \sum_{i=1}^{S} \frac{\partial L^{(i)}}{\partial \theta_j} \right] = \frac{1}{S} \text{Var} \left[ \frac{\partial L^{(i)}}{\partial \theta_j} \right]
\]
**SGD Learning Rate**

- In stochastic training, the learning rate also influences the amount of noise due to the stochasticity of the gradients.

![Diagram](image)

- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the noise

- This is in contrast to the deterministic setting, where learning rate decay might not be necessary
SGD Learning Rate

- It’s common to decay the learning rate by a large factor (e.g. 10x) at specific points during training.
- This results in large, sudden drops in the loss due to the reduction in gradient noise.

(He, 2015, “Deep residual learning for image recognition”)
Stochastic Gradient Descent: Batch Size

- The mini-batch size $S$ is a hyperparameter that needs to be set.
  - **Large batches**: converge in fewer weight updates because each stochastic gradient is less noisy.
  - **Small batches**: perform more weight updates per second because each one requires less computation.

**Claim**: If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal.

- 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$.
- Rewrite minibatch gradient descent as a for-loop:
  
  $\begin{align*}
  S &= 1 \\
  S &= 100 \\
  \text{For } k = 1, \ldots, 100: \\
  \theta_k &\leftarrow \theta_{k-1} - \alpha \nabla J^{(k)}(\theta_{k-1}) \\
  \theta_k &\leftarrow \theta_{k-1} - \frac{\alpha}{100} \nabla J^{(k)}(\theta_0)
  \end{align*}$

- All else being equal, you’d prefer to compute the gradient at a fresher value of $\theta$. So $S = 1$ is better.
Stochastic Gradient Descent: Batch Size

- The reason we don’t use $S = 1$ is that larger batches can take advantage of fast matrix operations and parallelism.

- **Small batches:** An update with $S = 10$ isn’t much more expensive than an update with $S = 1$.

- **Large batches:** Once $S$ is large enough to saturate the hardware efficiencies, the cost becomes linear in $S$.

- Cartoon figure, not drawn to scale:

- Since GPUs afford more parallelism, they saturate at a larger batch size. Hence, GPUs tend to favor larger batch sizes.
Distributed SGD:

- **Synchronous SGD**
  - Weights stored centrally on a parameter server, data divided between workers
  - Parameter server sends weights to workers
  - Workers separately compute gradients on a batch of data and send them to the parameter server
  - Parameter server aggregates the gradients and updates the weights

- Main advantage: efficiently compute gradients on larger batches
- There's also **asynchronous SGD** which improves data throughput by removing locking. I believe the fundamental tradeoffs are the same as for synchronous SGD.
The convergence benefits of larger batches see diminishing returns.

**Small batches**: large gradient noise, so large benefit from increased batch size

**Large batches**: SGD approximates the batch gradient descent update, so no further benefit from variance reduction.
Stochastic Gradient Descent: Batch Size

Shallue et al., 2019. “Measuring the effects of data parallelism on neural network training.”

- For each batch size, measure the number of steps required to reach a target validation loss.
- Hyperparameters (learning rates, etc.) tuned separately for each batch size (very expensive!)
- Clear separation into two regimes
  - a small batch regime which is noise dominated and achieves linear scaling
  - a large batch regime which is curvature dominated and gets no further benefit from parallelism
Stochastic Gradient Descent: Batch Size

(d) ResNet-50 on ImageNet
(e) ResNet-50 on Open Images
(f) Transformer on LM1B

(g) Transformer on Common Crawl
(h) VGG-11 on ImageNet
(i) LSTM on LM1B
Hypothesis: second order optimization will help more for large batch sizes:

First-order, small batch

Second-order, small batch

First-order, large batch

Second-order, large batch
Stochasticity and Curvature

Empirical scaling of different optimizers:

(b) Simple CNN on Fashion MNIST

(c) ResNet8 on CIFAR10

(e) ResNet32 on CIFAR10

(f) Transformer on LM1B
In Lecture 1, we got a lot of insight from analyzing the dynamics of convex quadratics, or equivalently, linear regression.

Two natural stochastic generalizations:
- Noisy quadratic model (NQM): convex quadratic with noisy gradient observations
- Linear regression with mini-batches

These two problems are not equivalent, and lead to interestingly different predictions.
Somewhat orthogonally: the field of optimization distinguishes several settings

- **Incremental optimization**: finite dataset, compute gradients on mini-batches for efficiency, minimize training cost
- **Stochastic optimization**: cost is stochastic, want to minimize its expectation (think dropout, VAEs, etc.)
- **Online optimization**: cost function sampled in each iteration (not necessarily i.i.d.), want to minimize regret, i.e. total loss compared to the best parameters in retrospect

We often blur these together in ML

- We use “stochastic” to refer to the incremental setting as well
- Unclear if there’s any advantage (on the validation set) to exploiting the finiteness of the dataset
- Algorithms like Adagrad, Adam, Shampoo, etc. come from the online setting

The noisy quadratic model uses the stochastic setting, or online learning with an i.i.d. assumption

- “infinite data”
Noisy Quadratic Model
Noisy Quadratic Model

Noisy Quadratic Model (NQM):

- Quadratic loss function:
  \[ \mathcal{L}(\theta) = \frac{1}{2} \theta^\top H \theta \]

- WLOG, we assume the optimum is at 0
- Each gradient query is noisy:
  \[ g = H\theta + \varepsilon \quad \mathbb{E}[\varepsilon] = 0 \quad \text{Cov}(\varepsilon) = C \]

- We measure the loss on the deterministic objective (so that the minimum is 0)
- The gradient noise is independent between steps (so this is essentially the infinite data setting)
Noisy Quadratic Model

Three optimization runs with different learning rates:
Noisy Quadratic Model

Taking the expectation (with respect to the gradient noise)
Noisy Quadratic Model

- To simplify the derivations, we assume $H$ and $C$ are codiagonalizable
  - Then we can assume WLOG that they are diagonal
- Consider the SGD dynamics. Since $H$ and $C$ are assumed diagonal, each coordinate evolves independently:

\[
\theta_i^{(k+1)} = \theta_i^{(k)} - \alpha g_i = (1 - \alpha h_i) \theta_i^{(k)} + \alpha \sqrt{c_i} \varepsilon_i
\]

- The risk (expected loss) can be decomposed as a sum of terms for each individual dimension
- These terms satisfy a bias-variance decomposition:

\[
\mathbb{E}[\mathcal{L}_i(\theta_i)] = \frac{1}{2} [\mathbb{E}[\theta_i]^2 + \text{Var}(\theta_i)]
\]
The system is linear, so we can compute the first and second moments of $\theta_i^{(k)}$ for all time steps using dynamic programming.

Assuming each $\alpha h_i < 2$, the loss for each dimension converges exponentially to a steady state risk, or noise floor:

$$E[L_i(\theta_i^{(k)})] = (1 - \alpha h_i)^{2k} \cdot E[L_i(\theta_i^{(0)})] + (1 - (1 - \alpha h_i)^{2k}) \cdot \frac{\alpha c_i}{2(2 - \alpha h_i)}$$

Tradeoff: Increasing $\alpha$ speeds up convergence, but increases the noise floor.
Noisy Quadratic Model

Visualizing the training dynamics, and the effect of decaying $\alpha$, along a high-curvature and a low-curvature eigendirection.
How can we choose the curvature \( H \) and the gradient covariance \( C \)?

This makes a difference to the qualitative behavior!

Our choices (which I’ll now justify):

- \( H = C \)
- \( h_i = 1/i \) (scale-free Hessian)

Note: the second assumption means there are a lot of low-curvature directions that collectively contribute a lot to the risk.
Noisy Quadratic Model

- First assumption: $\mathbf{H} = \mathbf{C}$. This is motivated by how the “empirical Fisher matrix” is often used as a proxy for the Hessian.
- The following scatterplots plot the Rayleigh quotients $\mathbf{v}^\top \mathbf{H} \mathbf{v}$ vs. $\mathbf{v}^\top \mathbf{C} \mathbf{v}$ for different vectors $\mathbf{v}$.
- To get both high and low curvature directions, $\mathbf{v}$ is chosen using the eigenvectors of the K-FAC approximation to $\mathbf{H}$.
Noisy Quadratic Model

- We assume a power law eigenspectrum for $\mathbf{H}$ (and hence also for $\mathbf{C}$). In particular, $h_i = 1/i$
- While there are only a handful of high-curvature directions, the many low-curvature directions are still important in aggregate
- This is a reasonably good match to the eigenspectrum of conv net Hessians as estimated by K-FAC.
- I don’t know whether or not the true Hessian has the same eigenspectrum, and we don’t currently have good tools to find out
- Consistent with recent estimates using generalized trace estimation (e.g. Ghorbani et al., 2019)
By simulating the exact dynamics, we can determine the number of SGD steps required to reach a target risk threshold.

Each point on these curves is the min over all choices of (fixed) learning rate, analogous to Shallue et al.’s experiments.

This plot can be generated in seconds.

We can clearly distinguish a “large batch” and a “small batch” regime.

**Note:** the training dynamics do not change between these regimes!
NQM: Preconditioning

- Preconditioned SGD update:
  \[ \theta^{(k+1)} = \theta^{(k)} - \alpha P^{-1}[H\theta + \varepsilon] \]

- Consider curvature-based preconditioners: \( P = H^\gamma \) with \( 0 \leq \gamma \leq 1 \)
  - \( \gamma = 0 \): ordinary SGD
  - \( \gamma = 1 \): stochastic Newton

- Effect on convergence in each dimension:
  \[
  \mathbb{E}[\mathcal{L}_i(\theta_i^{(k)})] = \underbrace{(1 - \alpha h_i^{1-\gamma})^{2t}}_{\text{convergence rate}} \mathbb{E}[\mathcal{L}_i(\theta_i^{(0)})] + \underbrace{(1 - (1 - \alpha h_i^{1-\gamma})^{2t})}_{\text{noise floor}} \frac{\alpha c_i h_i^{-\gamma}}{2(w - \alpha h_i^{1-\gamma})}
  \]

- **Tradeoff**: If \( h_i < 1 \), then preconditioning speeds up convergence in dimension \( i \), but increases the noise floor. Whether or not this is favorable depends on the specifics of \( H \) and \( C \).
Noisy Quadratic Model

- Large benefit of preconditioning in the large batch (curvature-dominated) regime
- Modest benefit, if any, in the small batch regime
- Dotted line = information theoretic lower bound (exact optimum for training data so far)
Noisy Quadratic Model

Optimal learning rate by batch size

Linear scaling in the noise dominated regime (agrees with a well-known heuristic)
Noisy Quadratic Model

Fitting optimal learning rate schedules at different batch sizes

![Graph showing learning rate and steps to threshold for different batch sizes.](image-url)
Two superficially similar algorithms:

- **Heavy ball momentum** (equivalent to an exponential moving average of the stochastic gradients)

  \[ v^{(k)} = \beta v^{(k-1)} - \alpha g^{(k)} \]
  \[ \theta^{(k)} = \theta^{(k-1)} + v^{(k)} \]

- **Iterate averaging** (in this case, exponential moving average of the parameters)

  \[ \theta^{(k)} = \theta^{(k-1)} - \alpha g^{(k)} \]
  \[ \tilde{\theta}^{(k)} = \mu \tilde{\theta}^{(k-1)} + (1 - \mu) \theta^{(k)} \]
NQM: Momentum and Iterate Averaging

But they have very different benefits:

![Graphs comparing HB Momentum and Iterate Averaging](image-url)
Empirical results (recap)

(b) Simple CNN on Fashion MNIST

(e) ResNet32 on CIFAR10

(f) Transformer on LM1B
Stochastic Linear Regression
I said the NQM was one of two natural generalizations of Lecture 1 to the stochastic setting.

For the NQM, we made the simplifying assumption that the gradient covariance $C$ is independent of $\theta$.

In the NQM, this leads to an information theoretic lower bound of $O(1/k)$ on the risk.

Without this assumption, it may be possible to achieve exponential convergence.
Stochastic Linear Regression

- Consider a linear regression problem where, under the data generating distribution, the labels are sampled with Gaussian noise

\[ t \mid x \sim \mathcal{N}(f_\star(x), \sigma_n^2), \]

where \( f_\star(x) = w_\star^\top x \) is the true function, or teacher, and \( w_\star \) are the true weights.

- Assume for simplicity batches of size 1 and \( x \sim \mathcal{N}(0, \Sigma_x) \)

- Applying the Law of Total Covariance,

\[
\text{Cov}(g) = \underbrace{\mathbb{E}[\text{Cov}(g \mid x)]}_{\text{label noise}} + \underbrace{\text{Cov}(\mathbb{E}[g \mid x])}_{\text{batch noise}}
\]

\[ = \mathbb{E} [\text{Var}(t - y \mid x) xx^\top] + \text{Cov}(x(f_\star(x) - f(x, w))) \]

\[ = \sigma_n^2 \Sigma_x + \text{Cov}(xx^\top (w_\star - w)) \]

- With no label noise, the first term vanishes.

- The second term vanishes when \( w = w_\star \)
Stochastic Linear Regression

- This implies that, with clean labels (or an overparameterized model!) the noise vanishes as you approach the optimum.

- Consider the proximal update which, in each iteration, computes:

\[
\mathbf{w}^{(k)} \leftarrow \arg\min_{\mathbf{w} : f(x^{(k)}, \mathbf{w}) = t^{(k)}} \| \mathbf{w} - \mathbf{w}^{(k-1)} \|^2
\]

- This is equivalent to the randomized Kaczmarz method for solving linear systems, which has long been known to converge exponentially.

- See Mark Schmidt’s “Notes on Randomized Kaczmarz” (https://www.cs.ubc.ca/~nickhar/W15/Lecture21Notes.pdf)
Ma et al., 2018, “The power of interpolation: Understanding the effectiveness of SGD in modern over-parameterized learning”
- In the interpolation regime (i.e. overparameterized), there exists a $\mathbf{w}_*$ which correctly predicts the training labels
- Exponential convergence bound for SGD on convex losses

Kidambi et al., 2018, “On the insufficiency of existing momentum schemes for stochastic optimization”
- Shows that Nesterov Accelerated Gradient (Lecture 9) doesn’t achieve acceleration in the stochastic setting, but provides an alternative that does
To summarize, we’ve seen two distinct models of stochastic optimization.

- **Online learning**, e.g. Noisy Quadratic Model
  - Infinite data, each update is independent
  - Exponential convergence is impossible

- **Interpolation regime**, e.g. stochastic linear regression
  - Finite dataset, possible to fit exactly
  - Noise comes only from the choice of batch
  - Exponential convergence is possible

- Which one best describes neural net optimization?
Two Models of Stochastic Optimization

- The interpolation regime would seem to be a better match to how neural nets are actually trained.
- On the other hand:
  - Online learning has motivated lots of algorithms that work really well for neural nets (natural gradient descent, adaptive gradient methods).
  - Optimization methods specifically designed to exploit the finiteness of the dataset (e.g. Stochastic Variance Reduced Gradients) aren’t used much for training neural nets.
  - It’s often easy to minimize the training loss much faster than current algorithms, and this rarely translates into better generalization error.
  - Arguably, the ability to fit a finite dataset faster will just help you overfit.
Deep Bootstrap
Deep Bootstrap

Nakkiran et al., ICLR 2021. “The deep bootstrap framework: Good online learners are good offline generalizers.”

- Intriguing claim: generalization error from training on a finite dataset closely follows the loss in the online (infinite data) regime until the point where you’ve fit the training set.
- The following experiment uses a synthetic CIFAR-like dataset generated by a deep generative model.
- Real world = train on 50K samples for 100 epochs. Ideal world = train on 5M samples for 1 epoch.
Deep Bootstrap

- Classical model of generalization:
  \[
  \text{TestError}(f_t) = \text{TrainError}(f_t) + \left[\text{TestError}(f_t) - \text{TrainError}(f_t)\right]
  \]
  Generalization Gap

- Since the training error can be very close to 0, this puts a lot of burden on the “generalization gap” to exactly predict the test error.

- Their alternative decomposition:
  \[
  \text{TestError}(f_t) = \text{TestError}(f_{t}^{iid}) + \left[\text{TestError}(f_t) - \text{TestError}(f_{t}^{iid})\right]
  \]
  Online Learning
  Bootstrap Error

- Hypothesis: the bootstrap error is typically small, until the point where the training loss is close to 0. Therefore, generalization error is well modeled by online learning.
Deep Bootstrap

- Left: online error vs. generalization from finite dataset
- Right: effect of dataset size. The dots indicate the point where the training error reaches 1%.
The Deep Bootstrap Hypothesis is still an empirical conjecture. No proof yet.

If it’s correct, then what appear to be generalization phenomena may actually be optimization phenomena, even when training error is small.

Some surprising interpretations/consequences

- Increasing the dataset size helps generalization by making it harder to reach low (e.g. 1%) training error, thereby keeping the bootstrap error low for more epochs.
- Dropout, data augmentation, etc. help for the same reason. One trades off the increased online training loss (due to stochasticity) with smaller bootstrap error.