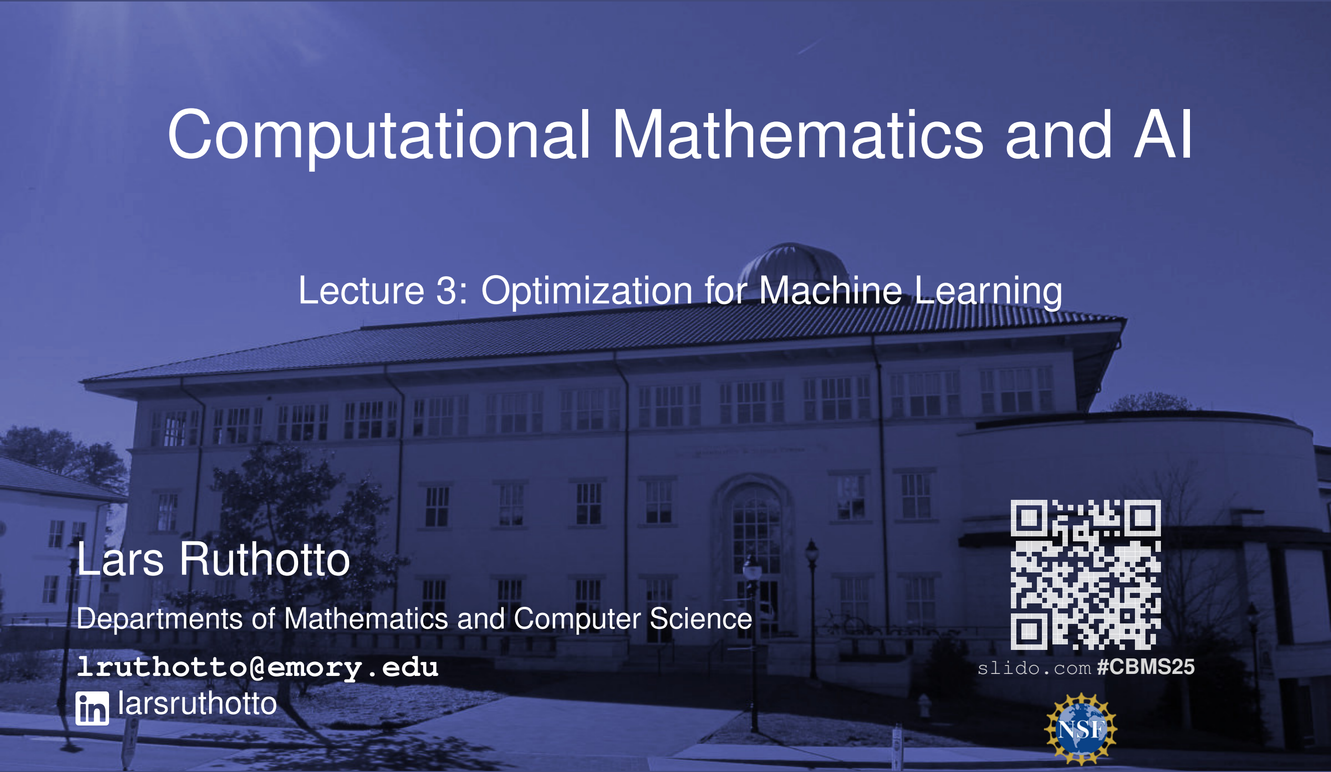


Computational Mathematics and AI

Lecture 3: Optimization for Machine Learning

Lars Ruthotto
Departments of Mathematics and Computer Science
lruthotto@emory.edu
larsruthotto

slido.com **#CBMS25**

Reading List

Historical Context: Stochastic optimization starts in 50s, backpropagation enabled ML in the 80s, acceleration/ implicit regularization are focus today.

Key Readings:

1. Robbins and Monro (1951) – Stochastic Approximation. *Ann. Math. Stat.*

   Foundational convergence theory for SGD.
2. Rumelhart, Hinton, and Williams (1986) – Back-Propagating Errors. *Nature*

   Classic paper that enabled neural network training.
3. Nocedal and Wright (2006) – *Numerical Optimization*, Springer.

   Classical optimization theory and second-order methods.
4. Bottou, Curtis, and Nocedal (2018) – Optimization Methods for Large-Scale ML. *SIAM Review*

   Comprehensive survey of SA vs SAA framework.
5. Baydin et al. (2018) – Automatic Differentiation in ML: A Survey. *JMLR*
   Forward/reverse mode AD for backpropagation.

Lecture Outline: SA vs SAA $\rightarrow$ Backprop & AD $\rightarrow$ Gauss Newton $\rightarrow$ SGD Basics

Roadmap: Optimization for Machine Learning

Question: How to train neural networks with millions to billions of parameters?

Four foundational pillars:

1. **SA vs SAA:** Two paradigms for stochastic optimization

   Robbins-Monro (1951) vs Vapnik (1998)

2. **Efficient gradients:** The enabling technology

   Backpropagation makes $O(p)$ gradient computation possible

3. **Gauss Newton methods:** Appeal and computational challenges

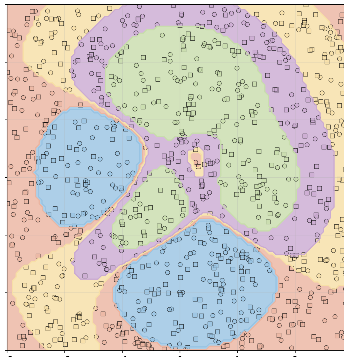
   $O(p^2)$ memory, $O(p^3)$ computation, and SAA/SA tension

4. **SGD as prototype SA algorithm:** Simple, scalable, surprisingly effective

   Convergence theory + sampling perspective: noise as feature, not bug

Running Example: Peaks Classification (Data + Model)

Dataset:



- 2D Peaks function
- 5 classes (level sets)
- 1000 i.i.d samples $\mathcal{U}([-3,3]^2)$

Two-Layer MLP Architecture:

$$\mathbf{h}^{(0)} = \mathbf{x} \in \mathbb{R}^2$$

$$\mathbf{h}^{(1)} = \text{ReLU}\left(\mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)}\right), \quad \mathbf{W}^{(1)} \in \mathbb{R}^{32\times 2}$$

$$\hat{\mathbf{y}} = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{5\times 32}$$

$$\hat{\mathbf{p}} = \text{softmax}(\hat{\mathbf{y}}) \in \mathbb{R}^5$$

Cross-entropy loss:

$$\ell(F_{\boldsymbol{\theta}}(\mathbf{x}), \mathbf{y}) = -\mathbf{y}^\top F_{\boldsymbol{\theta}}(\mathbf{x}) + \log\left(\mathbf{e}^\top \exp\left(F_{\boldsymbol{\theta}}(\mathbf{x})\right)\right)$$

Parameters: $\theta = (\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)})$

$$p = (2\times 32 + 32) + (32 \times 5 + 5) = 261$$

Goal: Find $\theta^* \in \mathbb{R}^{261}$ minimizing $\mathcal{L}(\theta)$

Stochastic Approximation vs Sample Average Approximation

Two Ways to Minimize Expected Loss

Sample Average Approximation (SAA)
Vapnik & Chervonenkis, 1998

Setup: Fix dataset and min empirical loss

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(F_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Algorithm: Any deterministic optimization method:

$$\theta_{t+1} = \theta_t - \eta \mathbf{H}^{-1} \nabla_\theta L_N(\theta_t)$$

Theory: Rich statistical learning theory (VC dimension, generalization bounds)
Advantage: Well-understood methods, no hyperparameters, parallel computation

Stochastic Approximation (SA)
Robbins & Monro, 1951

Setup: Minimize expected loss

$$\theta^* = \arg\min_{\theta} \mathbb{E}_{(\mathbf{x},\mathbf{y})}[\ell(F_\theta(\mathbf{x}), \mathbf{y})]$$

Algorithm: Stochastic gradient descent

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta[\ell(F_\theta(\mathbf{x}), \mathbf{y})]$$

for i.i.d. samples $(\mathbf{x}, \mathbf{y})$.

Theory: Converges if gradient is unbiased, learning rate suitable

Advantage: simple, scalable, good for streaming/online data, generalization

Mini-Batch SGD: The Practical Hybrid

Modern practice: combine benefits of both formulations

$$\theta_{t+1} = \theta_t - \eta_t \frac{1}{b} \sum_{i=1}^{b} \nabla_\theta \ell(F_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

where $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_b, \mathbf{y}_b)\}$ are i.i.d. samples from dataset

Interpretation ambiguity:

- SA view: improved gradient estimates using b samples (variance reduction)
- SAA view: stochastic approximation to batch gradient descent

Key terminology:

- **Batch size** b: number of samples per gradient computation
- **Epoch:** complete pass through training dataset (SAA concept)
- **Iteration:** single parameter update step
- **Learning rate** η: step size controlling update magnitude

mini-batch SGD balances variance reduction with computational efficiency

Computing Gradients: The Enabling Technology

The Gradient Computation Challenge

Requirement: all optimization algorithms need $\nabla L(\theta)$

- network with p parameters $\Rightarrow$ gradient $\nabla L(\theta) \in \mathbb{R}^p$
- modern networks: $p \sim 10^6$ to 10^{11} parameters
- **Example:** BERT-base has 110M parameters

Why not finite differences? Naive approach $\partial L/\partial \theta_i \approx [L(\theta + h\mathbf{e}_i) - L(\theta)]/h$ requires p forward passes – prohibitive for $p \sim 10^{11}$!

The solution: Backpropagation

- computes *exact* gradient in $O(p)$ operations
- same asymptotic cost as single forward pass
- exploits network structure via chain rule
- enables training of deep networks

efficient gradient computation via backpropagation enables deep learning

Backpropagation: The Chain Rule in Action

Idea: compute gradient in backward pass using chain rule

Computational graph: network is simple DAG

- nodes: variables (inputs, activations, outputs, loss)
- edges: operations (linear transforms, nonlinearities)

Two-pass algorithm:

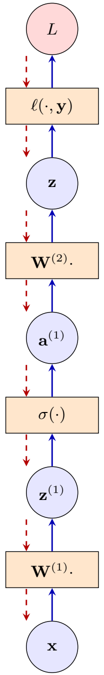
1. Forward pass: compute and store activations

$$\mathbf{a}^{(0)} = \mathbf{x}, \quad \mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{a}^{(\ell-1)}, \quad \mathbf{a}^{(\ell)} = \sigma(\mathbf{z}^{(\ell)})$$

2. Backward pass: propagate gradients (reverse order)

$$\frac{\partial L}{\partial \mathbf{a}^{(\ell)}} = \left(\frac{\partial \mathbf{a}^{(\ell+1)}}{\partial \mathbf{a}^{(\ell)}}\right)^T \frac{\partial L}{\partial \mathbf{a}^{(\ell+1)}}$$

cost $O(p)$ – same order as forward pass



Modern AD Frameworks: PyTorch, JAX, TensorFlow

- **automatic graph construction** (dynamic or static)
  - PyTorch: dynamic computation graphs
  - TensorFlow: static graphs with eager execution
  - JAX: functional transformations
- **built-in reverse-mode AD**
  - PyTorch: `.backward()`, `torch.autograd.grad()`
  - JAX: `grad()`, `value_and_grad()`
  - TensorFlow: `GradientTape`
- **handle complex control flow**
  - conditionals, loops, recursion
  - dynamic architectures
- **optimizations**
  - operator fusion for efficiency
  - memory management and checkpointing
  - graph compilation (XLA, TorchScript)

modern AD frameworks make backpropagation automatic and efficient

Computing Gradients with JAX: Value and Grad

Automatic Differentiation in Action

```python
import jax
import jax.numpy as jnp

# Define loss function
def loss_fn(params, X, y):
    """Compute softmax cross-entropy loss."""
    pred = model(params, X)
    return softmax_cross_entropy(pred, y)

# Get both loss value and gradient
loss_value, grad = jax.value_and_grad(loss_fn)(
    params, X_batch, y_batch
)

# Update parameters (vanilla SGD)
params_new = params - learning_rate * grad
```

Key Features:

- `jax.grad` returns gradient function
- `jax.value_and_grad` returns both
- Complexity: O(p) time, same as forward pass!
- Works via reverse-mode AD (backprop)

Mathematical View

Given $L : \mathbb{R}^p \to \mathbb{R}$:

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L}{\partial \theta_1} \\ \vdots \\ \frac{\partial L}{\partial \theta_p} \end{bmatrix}$$

Computational Cost:

- Forward pass: O(p)
- Backward pass: O(p)
- Total: O(p)

Enabling technology for deep learning!

Gauss-Newton Methods

Gauss-Newton Derivation for General Convex Loss

Starting point: Per-sample loss $\ell(F_\theta(\mathbf{x}), \mathbf{y})$

Step 1: Linearize network output around θ_0:

$$F_\theta(\mathbf{x}) \approx F_{\theta_0}(\mathbf{x}) + \mathbf{J}(\theta_0)(\mathbf{x})(\theta - \theta_0)$$

where $\mathbf{J}(\theta_0)(\mathbf{x}) = \frac{\partial F_\theta(\mathbf{x})}{\partial \theta}\Big|_{\theta_0} \in \mathbb{R}^{m \times p}$ is the **Jacobian**

Step 2: Quadratic Taylor expansion of total loss: Substitute linearization into $\mathcal{L}(\theta) = \sum_i \ell(F_\theta(\mathbf{x}_i), \mathbf{y}_i)$ and expand to second order.

$$\nabla^2 \mathcal{L} \approx \sum_i \mathbf{J}_i^T H_i \mathbf{J}_i$$

where $H_i = \nabla^2_{\hat{y}} \ell(\hat{y}, \mathbf{y}_i)|_{\hat{y} = F_{\theta_0}(\mathbf{x}_i)}$ is the **Hessian of per-sample loss w.r.t. predictions**

Different losses have different H structure

- **Least-squares:** $H = I \Rightarrow \mathbf{J}^T \mathbf{J}$ (classical GN)
- **Softmax cross-entropy:** $H_i = \operatorname{diag}(\mathbf{p}_i) - \mathbf{p}_i \mathbf{p}_i^\top$ for logits $\mathbf{p}_i$

Gauss-Newton naturally emerges from linearization + quadratic approximation

Computing the Jacobian: Small Output Dimension

Key observation: For classification with m classes, output $F_\theta(\mathbf{x}) \in \mathbb{R}^m$ is small!

Jacobian via reverse-mode AD (backpropagation):

- $\mathbf{J} = \frac{\partial F_\theta}{\partial \theta} \in \mathbb{R}^{m \times p}$ requires m backward passes
- **CIFAR-10:** 10 classes $\Rightarrow$ 10 backprops per sample
- **Memory:** can use less storage if computed batchwise and hidden features are large

JAX implementation: parallelize across samples

`J = vmap(jacrev(F_fn))(X)` gives $\mathbf{J}_i \in \mathbb{R}^{m \times p}$ for $i = 1, \ldots, n$

Gauss-Newton as dense linear algebra:

$$\mathbf{G} = \sum_{i=1}^{n} \mathbf{J}_i^\top \mathbf{H}_i \mathbf{J}_i \in \mathbb{R}^{p \times p}, \quad \text{solve } (\mathbf{G} + \lambda \mathbf{I})\delta = -\nabla \mathcal{L}$$

For small networks: GN = Jacobian stacking + dense solve

The Appeal of Gauss-Newton Methods

$$\theta_{t+1} = \theta_t - \mathbf{G}^{-1}\nabla L_t \quad \text{where } \mathbf{G} = \sum_{i=1}^{n} \mathbf{J}_i^\top \mathbf{H}_i \mathbf{J}_i$$

Theoretical advantages:

- **fast convergence** – when fitting training data (low residual)
- **curvature adaptation** – automatically adjusts step size based on local geometry
- **affine invariance** – robust to parameter scaling
- **best-case:** single step for quadratic problems
- **no hyperparameters** – well-understood linesearch and trust region methods

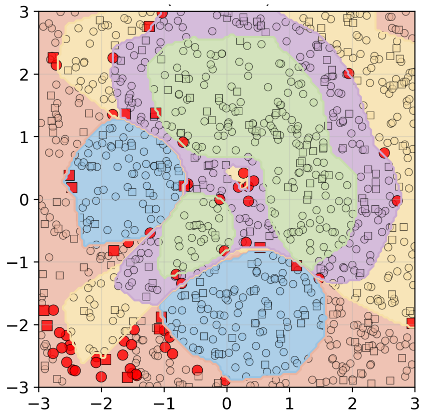
Practical advantages:

- **Parallelism** – Jacobian computations can be done in large batches

question: if so good theoretically, why limited use in deep learning?

Small Network: Gauss-Newton Performs Well

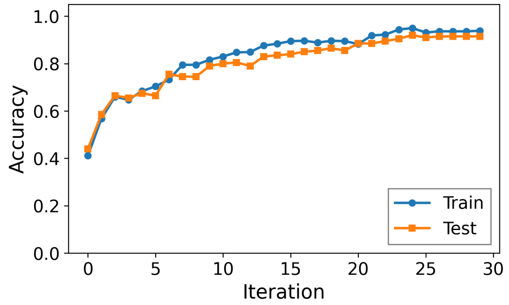
Predicted decision boundary:

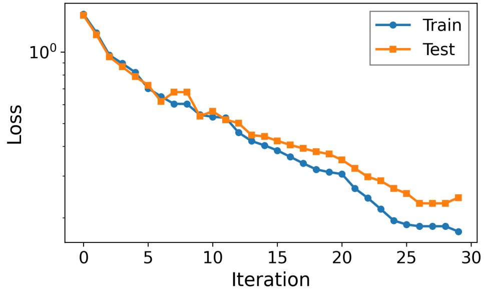


Final accuracy:

Train: 93.87% — Test: 91.50%

Convergence dynamics:





Computational Limitations: Memory and Storage Options

1. **Storage choices for curvature information:**
   - **Full Hessian:** $O(p^2)$ memory
     - Storage: p^2 elements (25M$\times$25M = 625 trillion) for ResNet-50
     - Our example: small network $p = 261 \Rightarrow$ feasible (68KB)
   - **Jacobian:** $O(n \times m)$ memory (per-sample outputs $\times$ parameters)
     - For classification: typically $m \ll p$, but n can be large
     - Need low-rank, quantization, or other compression techniques
2. **Computational cost: building the Hessian**
   - **Per-sample Jacobian computation:** $O(m)$ operations each (via AD)
   - **Hessian construction:** $O(n \times m^2)$ to form $\mathbf{J}^T H \mathbf{J}$
3. **Matrix-free approach and its limitations:**
   - **Preconditioning challenge:** Without Hessian matrix, preconditioner hard to construct
   - **Typical CG cost:** each iteration $\approx$ 2 SGD epochs of stochastic gradient computation
   - **Result:** Competition with SGD is lost in wall-clock time!

Statistical and Geometric Limitations

4. Stochastic optimization challenges:

- accurate Hessian estimation requires large batches
- tension: SAA ($b \gg 1$ for accurate curvature) vs SA (b small)
- small-batch Hessian too noisy for reliable updates
- implicit regularization benefits of SGD noise lost with large batches

5. Non-convexity in neural networks:

- Hessian may be indefinite (negative eigenvalues)
- Newton direction may not be descent direction
- requires modifications: trust regions, line search, damping
- further increases computational overhead

Path forward:

- structured approximations exploiting network architecture
- **Lecture 5:** modern approaches (K-FAC Martens and Grosse 2015, Shampoo) with tractable curvature

structured approximations enable practical adaptive methods

SGD: Convergence and Basic Properties

SGD Convergence in a Nutshell

$$\min_\theta \mathcal{L}(\theta) = \min_\theta \mathbb{E}_{(\mathbf{x},\mathbf{y})}[\ell(F_\theta(\mathbf{x}),\mathbf{y})]$$

SGD Update: $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \ell(F_{\theta_t}(\mathbf{x}_t),\mathbf{y}_t)$, where $(\mathbf{x}_t,\mathbf{y}_t) \sim P$ i.i.d.

Requirement 1 - Unbiasedness:

$$\mathbb{E}[\nabla_\theta \ell(F_\theta(\mathbf{x}),\mathbf{y})] = \nabla \mathcal{L}(\theta)$$

Requirement 2 - Robbins-Monro Conditions:

$$\sum_{t=1}^{\infty} \eta_t = \infty \quad \text{(reach optimum)} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty \quad \text{(control noise)}$$

Classical Convergence Results (assume $\sigma^2 = \mathbb{E}[\|\nabla_\theta \ell - \nabla \mathcal{L}\|^2] < \infty$):

- If problem is convex: $\mathbb{E}[\mathcal{L}(\theta_t) - \mathcal{L}(\theta^*)] \le \frac{C_1}{\sqrt{t}} + C_2\sigma^2\eta_t$
  - C_1 depends on: initial distance/suboptimality, smoothness and gradient bound of loss
  - Optimization error $O(1/\sqrt{t})$ + noise-induced error $O(\sigma^2\eta_t)$
  - Robbins-Monro: $\eta_t \to 0$ makes noise term vanish
- If problem is non-convex: $\min_{s\le t} \mathbb{E}[\|\nabla \mathcal{L}(\theta_s)\|^2] \le \frac{C}{\sqrt{t}} + \frac{\sigma^2}{t}$

Mini Batches as Noise Reduction

Central Limit Theorem: For large batch size b,

$$\frac{1}{b}\sum_{j=1}^{b} \nabla_\theta \ell(F_\theta(\mathbf{x}_j), \mathbf{y}_j) \sim \mathcal{N}\left(\nabla \mathcal{L}(\theta), \frac{1}{b}\Sigma(\theta)\right)$$

where $(\mathbf{x}_j, \mathbf{y}_j)$ are i.i.d. samples and $\Sigma(\theta) = \text{Cov}[\nabla_\theta \ell(F_\theta(\mathbf{x}), \mathbf{y})]$.

Why Σ/b? For i.i.d. samples $g_j = \nabla_\theta \ell(F_\theta(\mathbf{x}_j), \mathbf{y}_j)$:

$$\text{Cov}\left[\frac{1}{b}\sum_{j=1}^{b} g_j\right] = \frac{1}{b^2}\sum_{j=1}^{b}\text{Cov}[g_j] = \frac{1}{b^2}\cdot b \cdot \Sigma = \frac{\Sigma}{b}$$

Key implications:

- Gradient noise variance $\propto 1/b$
- Noise structure determined by $\Sigma(\theta)$
- To halve noise, need $4\times$ larger batches

Consequence: We can think about SGD as Monte Carlo optimization

Beyond the Gaussian Approximation

The CLT approximation is convenient, but has limitations:

Empirical observation (Simsekli, Sagun, and Gurbuzbalaban 2019):

- Gradient noise in deep learning often exhibits **heavy tails**
- Better characterized by α-stable (SαS) distributions
- Tail decay: $p(x) \sim |x|^{-(1+\alpha)}$ where $\alpha \in (0, 2]$
- Gaussian is the **special case** $\alpha = 2$

Why this matters:

- Heavy tails $\Rightarrow$ occasional **large jumps** in parameter space
- Large jumps help escape sharp minima (not captured by Gaussian model)
- Different optimizers interact differently with heavy-tailed noise

Forward reference:

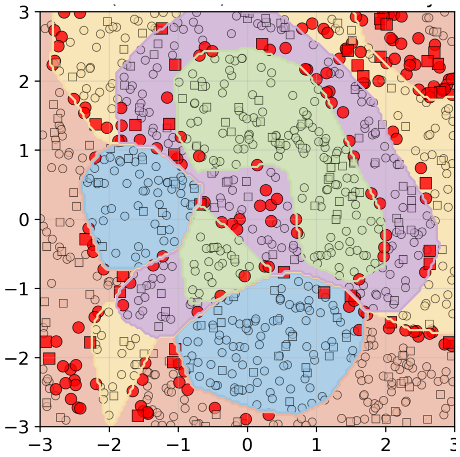
- **Lecture 5** develops this theory to explain Adam vs SGD generalization
- Adam *dampens* heavy-tailed noise $\Rightarrow$ different implicit bias

Reminder: A good learning algorithm converges, but a great one learns!!

SGD in Action

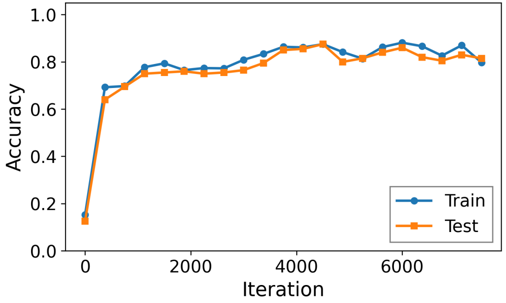
Small Network (width=32): Vanilla SGD

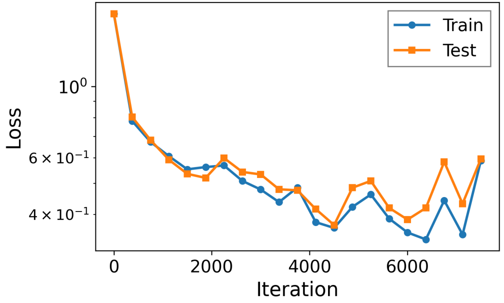
Decision boundary:



Final accuracy:
Train: 79.75% — Test: 81.50%

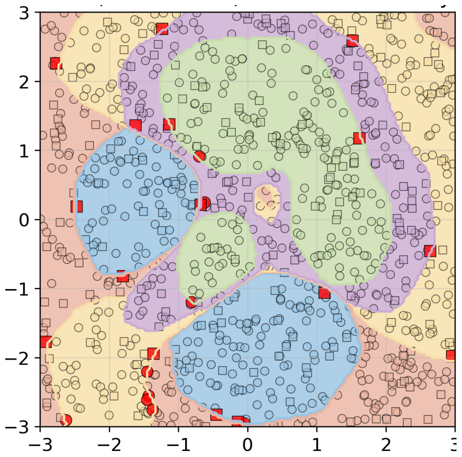
Convergence dynamics:





Large Network (width=8,192): The Regime Shift

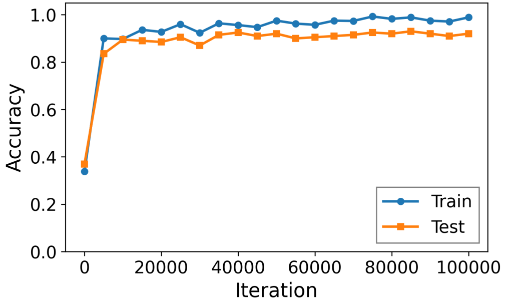
Decision boundary:

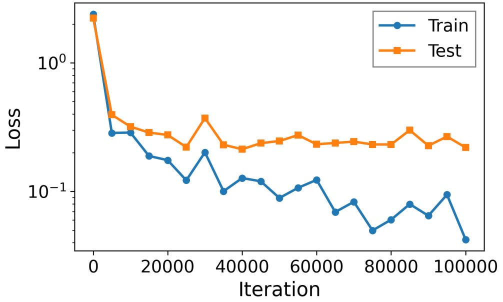


Final accuracy:

Train: 98.87% — Test: 92.00%

Convergence dynamics:





Σ: Optimization for Machine Learning

1. SA vs SAA formulations

- Robbins-Monro (1951): optimize expectations via stochastic gradients
- Vapnik (1998): optimize empirical risk on fixed datasets

2. Backpropagation as enabling technology

- Reverse-mode AD: exact gradient in $O(p)$ time (same as forward pass)
- Memory-storage tradeoff: forward stores activations OR recompute on backward

3. Gauss-Newton in small regime

- Linearize network, Hessian emerges naturally: $\nabla^2\mathcal{L} \approx J^T H J$
- Small networks: GN dominates; Large networks: $O(p^2)$ memory barrier

4. SGD in over-parameterized regime

- Small networks: hyperparameter tuning essential (Optuna helps)
- Large networks: SGD natural fit, ∼92% test accuracy with benign overfitting
- Modern practice: overparameterization makes optimization **easier**, not harder

Σ: Outlook

Open questions from this lecture:

- Why does SGD work so well in over-parameterized networks?
- Why does noise help optimization and generalization?

Where we're headed:

- **Lecture 4: SGD Deep Dive**
  - Implicit regularization mechanisms (early stopping, noise, flat minima)
  - Continuous-time perspective: gradient flow and Langevin dynamics
  - Edge of stability phenomenon in over-parameterized regime
  - Why over-parameterization makes optimization easier
- **Lecture 5: Modern Optimizers and Structured Methods**
  - Adaptive first-order: momentum, Adam, RMSprop and beyond
  - Modern structured second-order: K-FAC Martens and Grosse 2015, Shampoo

References I

- Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind (2018). "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Machine Learning Research* 18.153, pp. 1–43.
- Bottou, L., F. E. Curtis, and J. Nocedal (2018). "Optimization Methods for Large-Scale Machine Learning". In: *SIAM Review* 60.2, pp. 223–311.
- Martens, J. and R. Grosse (2015). "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature". In: *International Conference on Machine Learning (ICML)*, pp. 2408–2417.
- Nocedal, J. and S. J. Wright (2006). *Numerical Optimization*. 2nd. Springer.
- Robbins, H. and S. Monro (1951). "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22.3, pp. 400–407.
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). "Learning Representations by Back-Propagating Errors". In: *Nature* 323.6088, pp. 533–536.
- Simsekli, U., L. Sagun, and M. Gurbuzbalaban (2019). "A Tail-Index Analysis of Stochastic Gradient Noise in Deep Neural Networks". In: *International Conference on Machine Learning (ICML)*. PMLR, pp. 5827–5837.