Optimizers in deep learning

CPE 727 - Deep Learning

Ana Clara Loureiro Cruz **Bruno Coelho Martins** **Emre Aslan** **Felipe Barreto Andrade** (anaclaralcruz@poli.ufrj.br
bruno.martins@smt.ufrj.br
emre@gta.ufrj.br

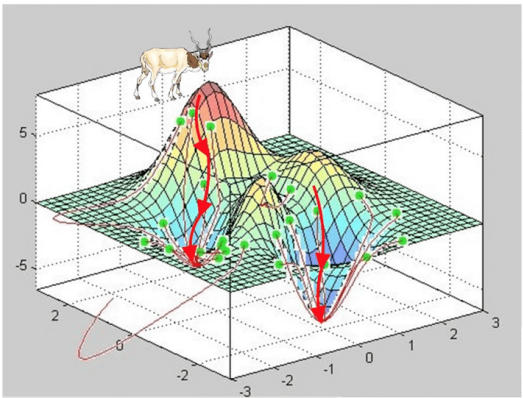
felipebarretoandrade@poli.ufrj.br)

November 10, 2025

Table of Contents

Donny Loves Optimization!

1 Introduction



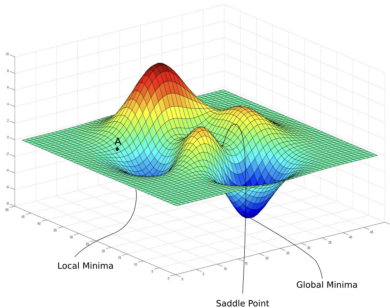
Meet Donny the donkey! He's on a mountain, trying to find the lowest valley. His adventure is like optimization in deep learning—finding the best settings for a model.

What is Optimization?

1 Introduction

- Donny's goal: Walk down the mountain to the lowest point (fewest mistakes).
- Optimization: Finding the best weights (settings) for a model to make the smallest mistakes.
- A loss function is like Donny's map—it shows how far he is from the bottom.
- The loss function measures mistakes (how wrong the model's guesses are). Example: Mean Squared Error (compares guesses to correct answers).

Why We Optimize?



- **Goal**: Find settings (weights) that make the *fewest mistakes* for accurate predictions.
- Ensures the model works great on *new, unseen data* (generalization).
- The map is tricky—wrong dips and flat spots can slow us down, but we aim for the *best spot*!

Table of Contents

2 Gradient Descent

What exactly is a gradient?

- Gradient is just a bunch of partial derivatives. This means that if we have some function of multiple variables, its gradient is just a vector of the function's derivatives with respect to each of its variables.
- The gradient has a fascinating property: it points in the direction where the function grows fastest! Correspondingly, the opposite direction is where the function decreases fastest!
- Since our goal is to minimize the loss function, the best course of action is to take a "step" in the direction where the function decreases fastest, which, as we saw, is the direction opposite to where the gradient points. [1]

Gradient Descent (GD) Optimization

2 Gradient Descent

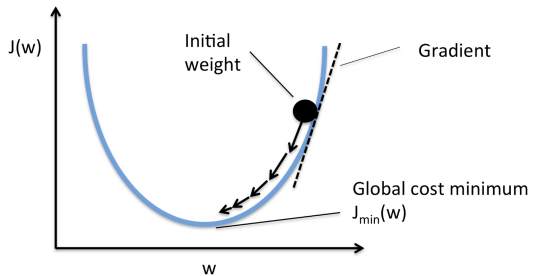
Donny the donkey navigates the loss landscape by taking steps opposite the steepest slope! The update rule for model parameters is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

Where:

- θ_t: Model parameters at iteration t (e.g., weights).
- η: Learning rate (step size for Donny's moves).
- $\nabla_\theta J(\theta_t)$: Gradient of the loss function, computed over the entire dataset, pointing to the steepest increase.
- θ_{t+1}: Updated parameters after each epoch (full pass over data). [2]

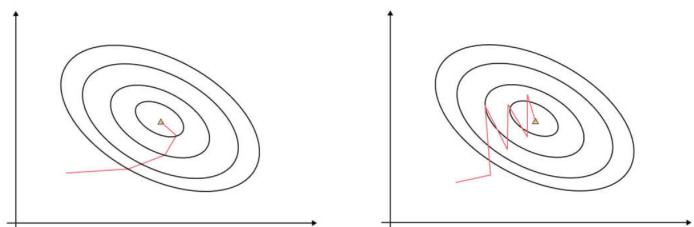
- Gradient Descent updates weights by moving opposite the gradient of the cost function to reach the minimum.
- Each step's size is determined by the learning rate.[3]



Scaling Gradient Descent

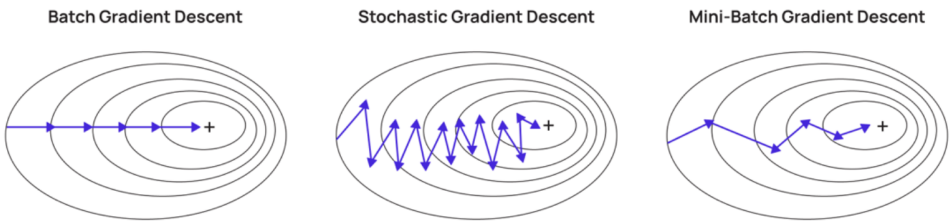
2 Gradient Descent

- Gradient Descent Challenge: Computing gradients on all data is computationally expensive, especially for large neural network datasets.
- Stochastic Gradient Descent (SGD): Updates weights using one data point, taking steps based on an estimated steepest descent, ideal for big datasets.[4]



Scaling Gradient Descent

- SGD's Strength: By the law of large numbers, single-data-point gradients average close to the true gradient, despite some zigzagging. [5]
- Minibatch Gradient Descent: Uses a small batch of examples for gradient computation, balancing speed and accuracy as a middle ground. [6]



Stochastic Gradient Descent (SGD) Formulation

Given a loss function $L(\theta, x_i, y_i)$ for a model with parameters θ, a single data point (x_i, y_i), and a learning rate η, the SGD update rule for the parameters at iteration t is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t, x_i, y_i)$$

Where:

- θ_t: Model parameters at iteration t.
- η: Learning rate (step size).
- $\nabla_\theta L(\theta_t, x_i, y_i)$: Gradient of the loss function with respect to θ, computed using a single randomly selected data point (x_i, y_i).
- θ_{t+1}: Updated parameters after the step. [3]

Why Stochastic Gradient Descent (SGD)?

- **Large Datasets**: SGD updates parameters using single data points, making it faster and more efficient than traditional gradient descent for massive datasets.[7]
- **Non-Convex Problems**: SGD's random updates help escape local minima, increasing the chance of finding the global minimum in non-convex optimization. [8]
- **Online Learning**: SGD supports real-time updates with new data, ideal for applications where data arrives continuously.[9]

Table of Contents

3 Momentum Variants

From SGD to Momentum: Why We Need It

3 Momentum Variants

- In standard Gradient Descent or SGD, parameters are updated by:

$$\theta_{t+1} = \theta_t - \eta\, g_t$$

- Where:
  - θ_t: model parameters at iteration t
  - $g_t = \nabla_\theta f_t(\theta_t)$: gradient of the loss
  - η: learning rate (step size)
- This simple rule moves directly opposite to the gradient.
- **Problem:** In narrow or curved valleys, the gradient direction changes quickly, causing **zig-zag motion**.
- We need a way to **smooth** these updates and keep consistent direction across steps.

Momentum: Adding Memory to SGD

3 Momentum Variants

Idea: Add "inertia", accumulate a moving average of past gradients:

$$v_t = \beta v_{t-1} + (1-\beta) g_t, \qquad \theta_{t+1} = \theta_t - \eta v_t$$

Where:

- v_t: velocity (smoothed gradient direction)
- β: momentum coefficient (typically 0.9)
- η: learning rate

Effect:

- Reduces oscillations across steep axes.
- Accelerates along consistent descent directions.
- Leads to faster and more stable convergence.

Visualization: Like rolling a ball downhill — it gains speed in the main slope direction and ignores small bumps.

COPPE 60 anos UFRJ

Table of Contents

4 Adaptive First-Order

Adaptive Learning Rates: Intuition (Adagrad vs. RMSProp)

- **Motivation:** Different parameters experience very different gradient magnitudes.
- **Adagrad:**
  - Accumulates the sum of squared gradients $\Rightarrow$ per-parameter step sizes shrink over time.
  - **Pros:** Excellent for sparse features.
  - **Cons:** Accumulator grows without bound $\Rightarrow$ steps can "die".
- **RMSProp:**
  - Uses an *exponential moving average* of squared gradients $\Rightarrow$ finite memory of the past.
  - **Pros:** Stabilizes step scale throughout training.
  - **Cons:** Sensitive to the EMA coefficient ρ.
- **Bridge to Adam:** Combine momentum (first moment) + RMSProp-like scaling (second moment) + bias correction.

Adam: Motivation and Intuition

- Adaptive Moment Estimation was proposed in 2014 by Kingma and Ba .
- Combines **momentum** (smooths noisy gradients) with **adaptive learning rates** (per-parameter step scaling).
- Inspired by ideas from *RMSProp* (adaptive second moment) and *Momentum* (EMA of gradients).
- Works well out-of-the-box; minimal tuning, fast convergence.
- Default optimizer for many architectures: CNNs, Transformers, LLMs.

Adam: Adaptive Moment Estimation

4 Adaptive First-Order

Formal equations:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t \odot g_t$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \qquad \theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Where:

- θ_t: model parameters at iteration t; ε: small constant for numerical stability (10^{-8}).
- m_t: first moment (EMA of gradients); β_1: decay factor for the mean (0.9).
- v_t: second moment (EMA of squared gradients); β_2: decay factor for variance (0.999).
- $\hat{m}_t$, $\hat{v}_t$: bias-corrected versions to remove initialization bias.
- $\odot$: element-wise multiplication (Hadamard product).

Introduced in **Adam** [10]. Combines momentum (first moment) and adaptive scaling (second moment).

Common variants: AMSGrad [11], RAdam [12].

AdamW: Decoupled Weight Decay

4 Adaptive First-Order

Update rule:

$$\theta_{t+1} = (1 - \eta\lambda)\,\theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Key idea:

- In the original Adam, weight decay (L2 regularization) was included *inside* the adaptive term, interfering with per-parameter learning rates.
- AdamW **decouples** weight decay from the gradient-based update.
- This preserves clean regularization and improves generalization, especially in large models (Transformers, ViTs).

Where:

- λ: weight decay coefficient. $(1 - \eta\lambda)$ applies pure L2 regularization.

Proposed in **AdamW** [13]. Later analyses on convergence and decay scaling: [14, 15].

Adam: Step-by-Step

4 Adaptive First-Order

1. Compute gradient $g_t = \nabla_\theta f_t(\theta_t)$.
2. Update first moment (mean): $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$.
3. Update second moment (variance): $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$.
4. Bias-correct: $\hat{m}_t = m_t/(1-\beta_1^t)$, $\hat{v}_t = v_t/(1-\beta_2^t)$.
5. Parameter update:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

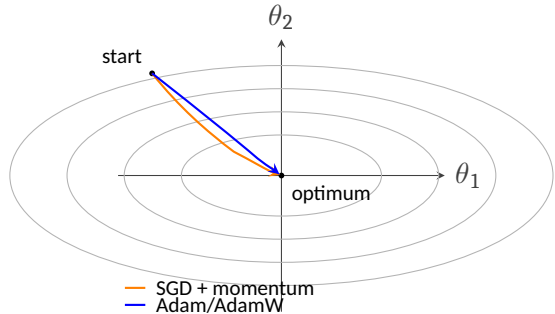
Typical defaults: $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \varepsilon = 10^{-8}$

Algorithm details follow [10]; bias corrections mitigate initialization effects in early iterations.

Adam: Geometric Interpretation

4 Adaptive First-Order

- **Anisotropy:** curvature differs across coordinates.
- **SGD (momentum):** faster than vanilla SGD, but still zig-zags along the steep axis.
- **Adam/AdamW:** per-parameter scaling (via $\sqrt{\hat{v}_t}$) damps steps where gradients are large and allows larger steps where they are small.
- **Effect:** straighter trajectory toward the minimum, fewer oscillations.



Adam rescales each coordinate by $\frac{1}{\sqrt{\hat{v}_t}+\varepsilon}$, so directions with persistently large gradients get smaller steps, reducing oscillations across the high-curvature axis and yielding a more direct path.

Practical Notes on Adam

4 Adaptive First-Order

- Default LR often 1e-3 for small models; 3e-4 for large ones.
- Combine with learning rate schedules (warmup + cosine decay).
- Works well with mixed-precision (FP16/BF16) training.
- Common in NLP/ViTs; sometimes replaced by SGD for vision due to generalization.
- Always prefer **AdamW** over Adam (decoupled weight decay).

Recent analyses such as [14, 15] revisit weight decay scaling for modern large-scale training.

COPPE 60 anos UFRJ

Table of Contents

5 Learning-rate (LR) Schedulers

Why use an LR scheduler?

- Schedulers dynamically change the learning rate during training to improve convergence and generalization.
- They can warm up, decay, oscillate, or respond to validation performance.
- Choosing the right scheduler depends on model size, batch size, dataset, and training stability.

[16]

LambdaLR — Flexible functional schedules

5 Learning-rate (LR) Schedulers

What it does: Applies a user-defined function that scales the base learning rate for each epoch or step.

When to use:

- When you need a fully customized learning rate function.
- Useful in research or experiments that require nonstandard decay shapes.

MultiplicativeLR — repeated multiplicative updates

What it does: Multiplies the current learning rate by a specified factor each iteration.
When to use:

- When you want a simple multiplicative decay or growth pattern.
- Ideal for smooth scaling without abrupt changes.

StepLR & MultiStepLR — discrete drops

What they do: StepLR reduces the learning rate by a fixed factor every few epochs. MultiStepLR applies drops at specific milestone epochs.

When to use:

- When you have prior knowledge about when to slow down learning.
- Suitable for traditional deep learning setups like ResNet training.
- StepLR for regular intervals; MultiStepLR for manually chosen epochs.

ConstantLR & LinearLR

5 Learning-rate (LR) Schedulers

What they do: ConstantLR maintains a fixed multiplier for a few iterations. LinearLR gradually increases or decreases the rate linearly between two factors.
When to use:

- Ideal for warmup phases before the main schedule begins.
- LinearLR provides a smooth transition from a small to a normal learning rate.

ExponentialLR & PolynomialLR

5 Learning-rate (LR) Schedulers

What they do: ExponentialLR decays the learning rate exponentially over epochs. PolynomialLR uses a polynomial decay curve.

When to use:

- ExponentialLR for steady and continuous decay over long training runs.
- PolynomialLR when you want slower decay near the end of training or more control over the tail behavior.

CosineAnnealingLR — smooth annealing

5 Learning-rate (LR) Schedulers

What it does: Follows a cosine-shaped curve from a maximum to a minimum learning rate.

$$\eta_{t+1} = \eta_{\min} + (\eta_t - \eta_{\min}) \cdot \frac{1 + \cos\left(\frac{T_{\text{cur}}}{T_{\max}}\pi\right)}{1 + \cos\left(\frac{T_{\text{cur}}+1}{T_{\max}}\pi\right)}$$

Where:

- η_t — learning rate at step t
- T_{cur} — number of epochs since the last restart
- $T_{\max}$ — maximum number of epochs in a cycle

When to use:

- Recommended for long training sessions where smooth convergence is desired.
- Commonly paired with warmup for large-scale models.
- Provides a graceful reduction toward the end of training.

ChainedScheduler & SequentialLR — combining schedules

What they do: ChainedScheduler applies multiple schedulers together; SequentialLR runs them in sequence over predefined intervals.

When to use:

- When combining warmup, decay, or cyclic phases.
- SequentialLR is ideal for warmup followed by a main schedule.
- ChainedScheduler is used for composite effects applied at the same time.

ReduceLROnPlateau — metric-driven reduction

5 Learning-rate (LR) Schedulers

What it does: Reduces the learning rate when a monitored validation metric stops improving.

When to use:

- When model performance does not improve consistently.
- Common in fine-tuning or transfer learning scenarios.
- Useful when the ideal decay timing is unknown.

CyclicLR — cyclical policies

5 Learning-rate (LR) Schedulers

What it does: Cycles the learning rate between a minimum and maximum value, forming a triangular or exponential pattern.

When to use:

- When you want the optimizer to periodically explore new minima.
- Suitable for smaller models and datasets with high gradient noise.
- Encourages faster convergence and can escape plateaus.

OneCycleLR — single-cycle super-convergence

What it does: Increases the learning rate from a small value to a peak and then decreases it to near zero in one cycle.

When to use:

- When aiming for very fast convergence (super-convergence).
- Works well for vision and classification tasks with per-batch updates.
- Best used when the total number of iterations is known beforehand.

Practical recommendations (summary)

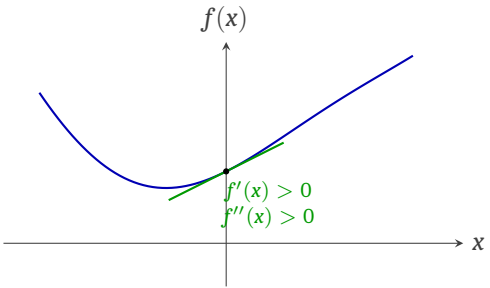
- **Warmup:** LinearLR or ConstantLR at the start of training.
- **Stable defaults:** StepLR or MultiStepLR for traditional pipelines.
- **Modern default:** Warmup followed by CosineAnnealingLR.
- **Fast training:** OneCycleLR or CyclicLR for rapid convergence.
- **Dynamic control:** ReduceLROnPlateau when validation loss drives the schedule.
- **Composition:** SequentialLR or ChainedScheduler for complex setups.

Table of Contents

6 Beyond First Order

First vs Second Derivative (1D intuition)

- The **first derivative** $f'(x)$ gives the *slope* (direction of change).
- The **second derivative** $f''(x)$ gives the *curvature* (how the slope bends).
- Where $f'(x) = 0$:
  - $f''(x) > 0 \rightarrow$ local minimum (curve opens upward)
  - $f''(x) < 0 \rightarrow$ local maximum (curve opens downward)
- For many dimensions:
  - $f'(x) \rightarrow \nabla f(\theta)$
  - $f''(x) \rightarrow H = \nabla^2 f(\theta)$



[17]

Why going beyond first order?

- First-order methods (SGD, Adam, etc.) rely only on the gradient $\nabla f(\theta)$.
- Second-order methods also use curvature information, via the Hessian $H = \nabla^2 f(\theta)$.
- They adapt the step direction and magnitude according to local curvature:

$$\theta_{t+1} = \theta_t - H_t^{-1} \nabla f(\theta_t)$$

- This can greatly accelerate convergence near the optimum (quadratic rate).

[17, 18].

Newton's Method in ML

6 Beyond First Order

Starting from a second-order Taylor expansion around the current point θ_t:

$$f(\theta_t + \Delta) \approx f(\theta_t) + \nabla f(\theta_t)^\top \Delta + \tfrac{1}{2}\Delta^\top H_t \Delta$$

- The **gradient** $\nabla f(\theta_t)$ gives the local slope (first-order info).
- The **Hessian** H_t gives the local curvature (second-order info).
- Minimizing this local quadratic model leads to:

$$H_t \Delta_t = -\nabla f(\theta_t) \quad \Rightarrow \quad \theta_{t+1} = \theta_t + \Delta_t$$

Interpretation: Instead of taking small steps like gradient descent, Newton's Method uses curvature to **jump directly to the minimum** of the local parabola approximation.

Newton's Method in ML [18]

Algorithm:

$$H_t \Delta_t = -\nabla f(\theta_t), \qquad \theta_{t+1} = \theta_t + \Delta_t$$

Characteristics:

- Converges in few iterations for convex, well-conditioned problems.
- Requires Hessian computation or its inverse.

Applications:

- Generalized Linear Models (as Iteratively Reweighted Least Squares).
- Gaussian process optimization, small neural nets, or fine-tuning.

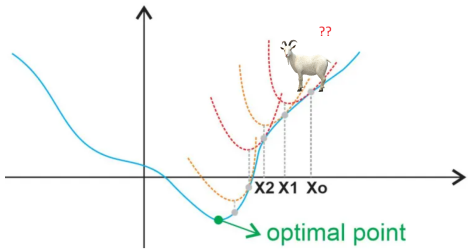


Figure: Visual representation of Newton algorithm in 1D [19].

BFGS [17]

Idea: Approximate the inverse Hessian $B_t \approx H_t^{-1}$ using only gradients and parameter steps.

Secant condition:

$$B_{t+1}\gamma_t = s_t, \quad s_t = \theta_{t+1} - \theta_t, \quad \gamma_t = \nabla f(\theta_{t+1}) - \nabla f(\theta_t)$$

BFGS update:

$$B_{t+1} = B_t + \frac{(s_t^\top \gamma_t + \gamma_t^\top B_t \gamma_t)(s_t s_t^\top)}{(s_t^\top \gamma_t)^2} - \frac{B_t \gamma_t s_t^\top + s_t \gamma_t^\top B_t}{s_t^\top \gamma_t}$$

Properties:

- Curvature-aware steps without computing H_t.
- Converges superlinearly for smooth convex functions.

Limitations:

- Requires storing full $B_t \in \mathbb{R}^{d\times d} \to O(d^2)$ memory.
- Sensitive to noisy gradients and non-convexity.

Limited BFGS [20]

Idea: Keeping only the last m correction pairs (s_i, y_i) to build B_t^{-1} implicitly.

Two-loop recursion:

1. Compute $\alpha_i = \rho_i \, s_i^\top q$, where $\rho_i = 1/(y_i^\top s_i)$.
2. Recursively apply curvature corrections backward, then forward:

$$p_t = -H_t \nabla f_t \approx -\hat{B}_t \nabla f_t$$

(without storing full B_t).

Application:

- Classical ML and small neural net fine-tuning.

Limitations:

- Still needs line search; not suited to noisy minibatches.
- Slower wall-time per iteration than SGD/Adam for large-scale deep nets.

Modern Successors: Curvature Approximations

6 Beyond First Order

- **Hessian-Free Optimization** [21]: uses matrix-vector products Hv without forming H.
- **K-FAC** [22]: Kronecker-factored preconditioning for layerwise curvature.
- **Shampoo** [23]: factored 2nd-order preconditioners for large tensors.

Table of Contents

7 Summary

Optimizer comparison

7 Summary

SGD + Momentum (Polyak/Nesterov)

Behaviour: low memory.
Cons: sensible to LR.
Use: CNN/vision with cosine+warmup.

Adam / AdamW

Behaviour: momentum + adaptative; fast.
Cons: can generalize worst than SGD.
Use: Transformers/ViTs/LLMs, mixed precision.

RMSProp / Adagrad

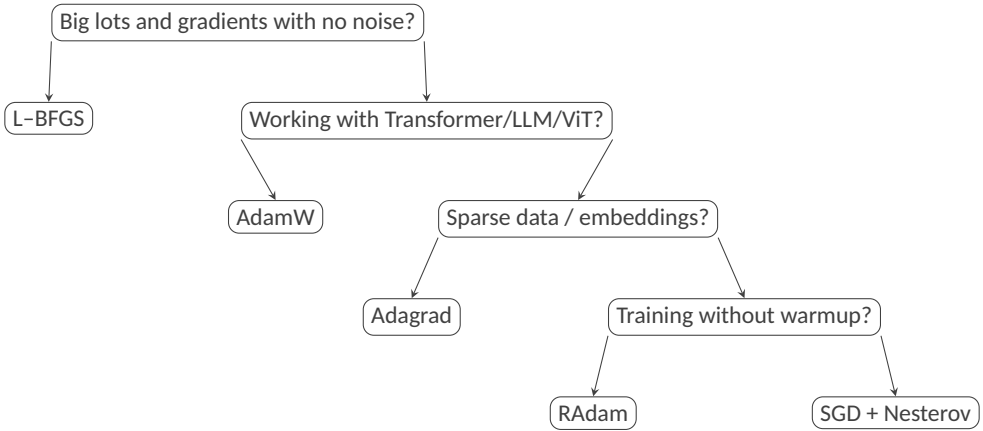
Behaviour: stable in scales; sparse.
Cons: Adagrad saturates.
Use: RNNs/online (RMSProp), NLP esparse (Adagrad).

(L-)BFGS

Behaviour: Uses Hessian; big steps.
Cons: sensible to stochastic noise.
Uso: smaller problems and last-layer.

Quick start guide for chosing optimizers

7 Summary



Areas of study

7 Summary

- **Optimization vs generalization:** [24] why does SGD often generalize better than AdamW in vision?
- **Sharpness & flatness:** [?] Is flatness truly a reliable indicator of generalization in deep networks?
- **Scaling laws interplay:** [25] how LR, schedule, and batch size co-vary optimally at trillion-token scale?

Key takeaways

7 Summary

- Start simple: SGD+momentum (vision) or AdamW (transformers), warmup + cosine.
- Schedules matter as much as the optimizer; tune LR first, then weight decay/momentum.
- Consider SAM/Lookahead when generalization/stability is a pain point.
- Large batches: add LARS/LAMB trust ratios and longer warmup.
- For small/smooth problems, L-BFGS/Newton can be very effective.

Table of Contents

8 References

Bibliography

8 References

[1] J. Lu, “Gradient descent, stochastic optimization, and other tales,” 2024.

[2] K. Chandra, A. Xie, J. Ragan-Kelley, and E. Meijer, “Gradient descent: The ultimate optimizer,” 2022.

[3] L. Bottou, “Online algorithms and stochastic approximations,” 1998.

[4] S. Ruder, “An overview of gradient descent optimization algorithms,” 2017.

[5] P. Erd, “On a new law of large numbers,” *J. Anal. Muth*, vol. 22, pp. 103–l, 1970.

[6] S. Khirirat, H. R. Feyzmahdavian, and M. Johansson, “Mini-batch gradient descent: Faster convergence under data sparsity,” 2017.

[7] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” 2010.

Bibliography

8 References

[8] Y. Lei, T. Hu, G. Li, and K. Tang, "Stochastic gradient descent for nonconvex learning without bounded gradient assumptions," *IEEE transactions on neural networks and learning systems*, vol. 31, no. 10, pp. 4394–4400, 2019.

[9] E. Jothimurugesan, A. Tahmasbi, P. Gibbons, and S. Tirthapura, "Variance-reduced stochastic gradient descent on streaming data," 2018.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[11] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," 2019.

[12] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," 2019.

[13] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2017.

Bibliography

8 References

[14] K. Ding, N. Xiao, and K.-C. Toh, “Adam-family methods with decoupled weight decay in deep learning,” 2023.

[15] X. Wang and L. Aitchison, “How to set adamw’s weight decay as you scale model and dataset size,” 2024.

[16] PyTorch Core Team, *PyTorch Documentation*, 2025. Accessed: October 2025.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[18] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

Bibliography

8 References

[19] A. Umam, “Newton’s method optimization: Derivation and how it works.” https://ardianumam.wordpress.com/2017/09/27/newtons-method-optimization-derivation-and-how-it-works/, 2017. Accessed: 21 October 2025.

[20] PyTorch Core Team, *LBFGS - PyTorch Documentation*, 2025. Accessed: October 2025.

[21] J. Martens *et al.*, “Deep learning via hessian-free optimization.,” in *Icml*, vol. 27, pp. 735–742, 2010.

[22] J. Ba, R. Grosse, and J. Martens, “Distributed second-order optimization using kronecker-factored approximations,” in *International conference on learning representations*, 2017.

Bibliography

8 References

[23] V. Gupta, T. Koren, and Y. Singer, “Shampoo: Preconditioned stochastic tensor optimization,” in *International Conference on Machine Learning*, pp. 1842–1850, PMLR, 2018.

[24] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, “On large-batch training for deep learning: Generalization gap and sharp minima,” *arXiv preprint arXiv:1609.04836*, 2016.

[25] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” *arXiv preprint arXiv:2001.08361*, 2020.

COPPE UFRJ 60 anos

Optimizers in deep learning

Obrigado pela Atenção!
Alguma Pergunta?