Course Materials for Deep Generative AI

Northeastern University

These materials have been prepared and sourced for the course **Deep Generative AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

If you believe any material has been inadequately cited or requires correction, please contact me at:

Instructor: Ramin Mohammadi
r.mohammadi@northeastern.edu

Thank you for your understanding and collaboration.

Content sourced from multiple references - see Reference section for details

Optimization Techniques

Jacobian Matrix

In optimization and root-finding problems involving systems of nonlinear equations, the Jacobian matrix $J(x)$ plays an important role.

> The Jacobian matrix is composed of first-order partial derivatives and provides information about the sensitivity of the function's outputs to changes in the inputs.

For a system of m functions $f_1(x), f_2(x), \ldots, f_m(x)$ in n variables $x_1, x_2, \ldots, x_n$, the Jacobian matrix is an $m \times n$ matrix and is defined as:

$$J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Each element $\frac{\partial f_i}{\partial x_j}$ represents the partial derivative of the i-th function with respect to the j-th variable, capturing how sensitive the function $f_i(x)$ is to changes in x_j.

Hessian Matrix

For optimization problems where we are trying to minimize a scalar-valued function $f(x)$, the Hessian matrix $H(x)$

> contains the second-order partial derivatives of the function and describes the curvature of the function. It plays a crucial role in Newton's method for optimization.

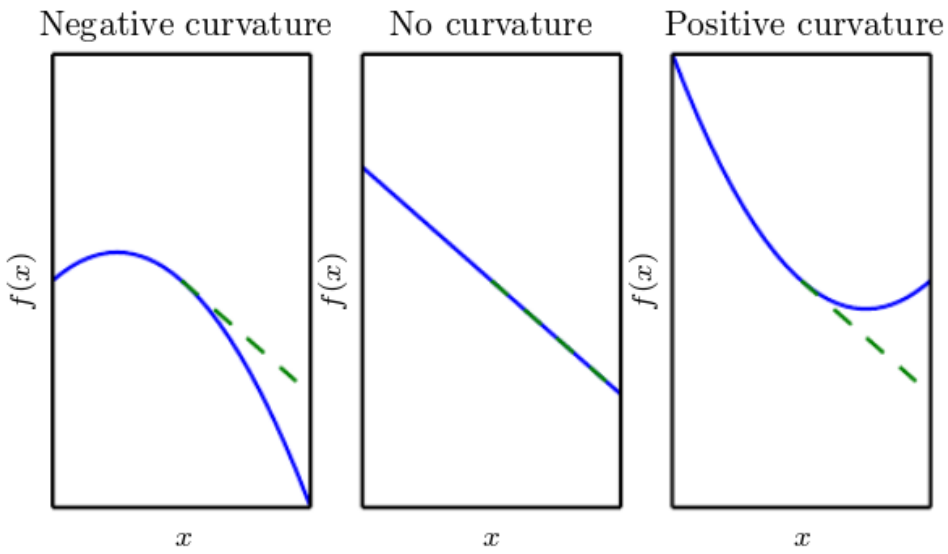
However, when the Hessian is unavailable or too costly to compute, Quasi-Newton methods approximate it using only gradient information.

The Hessian matrix for a function $f(x)$ of n variables is defined as an $n \times n$ matrix:

$$H(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Understanding the Second Derivative and the Hessian in Machine Learning

- Intuitively, the second derivative of a function is the rate of change of the slope of the function. This is analogous to its *acceleration*.
- The Hessian matrix of a function is the rate at which different input dimensions accelerate with respect to each other. If $H_{1,1}$ is high, it means there is a high acceleration in dimension 1. If $H_{1,2}$ is high, then we accelerate simultaneously in both dimensions. If $H_{1,2}$ is negative, then as we accelerate in the first dimension, we decelerate in the second dimension.
- In the context of gradient descent in Machine Learning, the second derivative measures the curvature of the loss function, as opposed to the slope (gradient) at a single coordinate. Information about the Hessian can therefore help us take appropriate gradient steps towards the minima.



- Suppose we have a function f, where the gradient $f'(x)$ is negative. We know that the function is sloping downwards, however we do not know whether it is:
  1. sloping downwards more steeply,
  2. sloping downwards at the same rate, or
  3. sloping downwards less and potentially going upwards.

  These can be informed by the second derivative, which is the gradient of the gradient.

  - In **scenario (1)**, if the second derivative is negative, then the function is accelerating downwards, and the cost function will end up decreasing more than the gradient multiplied by step-size.
  - In **scenario (2)**, if the second derivative $f''(x)$ is 0, the function is a straight line with no curvature because acceleration/deceleration is 0. Then a step-size of α along the negative gradient will decrease the $f(x)$ by $c \cdot \alpha$.
  - In **scenario (3)**, if the second derivative is positive, then the function is decelerating and eventually accelerates upward. Thus, if α is too large, gradient descent might end up with coordinates that result in greater cost. If we are able to calculate the second derivative, then we can control the α to reduce oscillation around the local minima.

Hessian Matrix: Eigenvalues, Convexity, and Saddle Points

- Eigenvectors/eigenvalues of the Hessian describe the directions of principal curvature and the amount of curvature in each direction. Intuitively, the local geometry of curvature is measured by the Hessian.
- In the context of Machine Learning optimization, after we have converged to a critical point using gradient descent, we need to examine the eigenvalues of the Hessian to determine whether it is a minimum, maximum, or saddle point. Examining the properties of the eigenvalues tells us something about the convexity of the function. For all $\mathbf{x} \in \mathbb{R}^n$:
  - If H is positive definite $H \succ 0$ (all eigenvalues are > 0), the quadratic problem takes the shape of a "bowl" in higher dimensions and is strictly convex (has only one global minimum). If the gradient at coordinates $\mathbf{x}$ is 0, $\mathbf{x}$ is at the global minimum.
  - If H is positive semi-definite $H \succeq 0$ (all eigenvalues are ≥ 0), then the function is convex. If the gradient at coordinates $\mathbf{x}$ is 0, $\mathbf{x}$ is at a local minimum.
  - If H is negative definite $H \prec 0$ (all eigenvalues are < 0), the quadratic problem takes the shape of an inverted bowl in higher dimensions and is strictly concave (has only one global maximum). If the gradient at coordinates $\mathbf{x}$ is 0, $\mathbf{x}$ is at the global maximum.
  - If H is negative semi-definite $H \preceq 0$ (all eigenvalues are ≤ 0), then the function is concave. If the gradient at coordinates $\mathbf{x}$ is 0, $\mathbf{x}$ is at a local maximum.
- Note that the opposite doesn't hold in the above conditions. E.g., strict convexity does not imply that the Hessian everywhere is positive definite.
- If H is indefinite (has both positive and negative eigenvalues at $\mathbf{x}$), this implies that $\mathbf{x}$ is both a local minimum and a local maximum. Thus, $\mathbf{x}$ is a saddle point for f.

Momentum

Momentum is a technique used to accelerate convergence, especially in regions with flat or oscillatory gradients. It draws inspiration from physical momentum, helping the algorithm maintain its direction and avoid getting stuck in local minima or saddle points.

The velocity V term is introduced to smooth out the updates and accumulate gradients over time:

$$V_{t+1} = \beta V_t + (1 - \beta)\nabla Q(w_t)$$

Here, β is the momentum coefficient that controls how much of the past gradients are retained. The weight update then becomes:

$$w_{t+1} = w_t - \alpha V_{t+1}$$

This approach can be interpreted as applying a running average of all gradients observed so far, with a discount factor applied to older gradients.

The key steps for momentum-based updates are:

- At $t = 0$, the initial velocity is zero: $V_0 = 0$.
- At the first iteration, the velocity is directly proportional to the gradient: $V_1 = (1 - \beta)\nabla Q(w_0)$, leading to the weight update:
$$w_1 := w_0 - \alpha(1 - \beta)\nabla Q(w_0)$$
- At subsequent iterations, the velocity combines the current gradient with the discounted previous gradients, leading to smoother updates:
$$V_2 = \beta V_1 + (1 - \beta)\nabla Q(w_1)$$
This gives the weight update rule:
$$w_2 := w_1 - \alpha(1 - \beta)[\beta\nabla Q(w_0) + \nabla Q(w_1)]$$

Thus, momentum introduces a running average of previous gradients, effectively smoothing out updates and accelerating convergence.

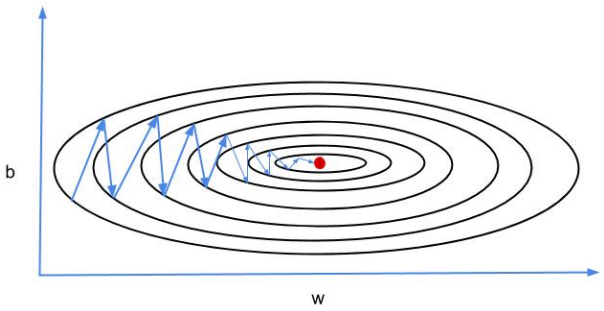


Figure 1: Momentum with SGD

Understanding Velocity Terms

The velocity term provides several benefits:

- It acts as the running average of past gradients, accumulating information about the optimization landscape.
- It smooths out noisy updates, making the learning process more stable.
- It accelerates the search in flat regions of the loss function, allowing the algorithm to make faster progress.
- It provides stability in regions where gradients oscillate, particularly near saddle points or local minima.

By accelerating in directions where gradients remain consistent and damping oscillations in directions where they vary, momentum helps the algorithm efficiently navigate the optimization landscape, leading to faster convergence.

Newton's Methods

Newton's method is an optimization technique used to find the local minima of a twice-differentiable function $f(x)$. It is particularly effective when the function is positive quadratic or locally quadratic at a given point. The update rule for Newton's method is derived from using second-order information (the Hessian matrix) to take a more direct route toward the minima. The general update rule is given by:

$$x_{t+1} := x_t - \frac{f'(x_t)}{f''(x_t)}$$

or, more generally in higher dimensions:

$$x_t := x_t - \alpha \nabla^2 f(x)^{-1} \nabla f(x)$$

where $\nabla f(x)$ is the gradient of the function, and $\nabla^2 f(x)$ is the Hessian matrix, representing the second derivative (curvature information) of the function.

Taylor Series Approximation

Newton's method leverages a second-order Taylor series approximation of the function $f(x)$.

The Taylor series expansion approximates $f(x)$ by polynomials of increasing powers, capturing the local behavior of the function near a given point x_p.

For a second-order Taylor expansion around the point x_p, the function can be approximated as:

$$f(x_p + \Delta x) = f(x_p) + f'(x_p)\Delta x + \frac{1}{2} f''(x_p)\Delta x^2 + \dots$$

Here, $f'(x_p)$ is the first derivative (gradient) at x_p, and $f''(x_p)$ is the second derivative (curvature) at x_p.

The goal of Newton's method is to find Δx such that $x_p + \Delta x$ minimizes the function $f(x)$, i.e., we want $x_p + \Delta x$ to be the stationary point, where the derivative is zero.

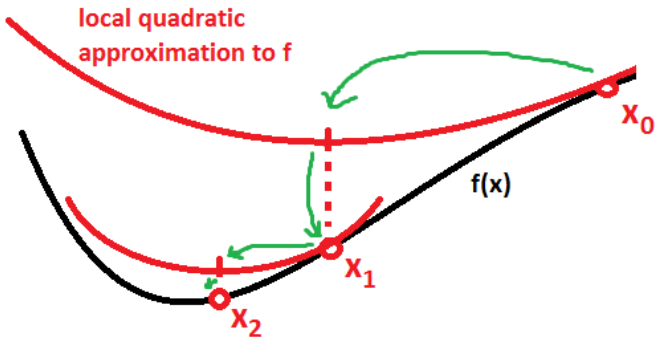


Figure 2: Taylor Series Approximation

Finding Δx

To estimate the value of x that minimizes $f(x)$, we can truncate the Taylor series to the second order and solve for Δx. The truncated series is:

$$f(x_p + \Delta x) \approx f(x_p) + f'(x_p)\Delta x + \frac{1}{2} f''(x_p)\Delta x^2$$

We minimize this function by setting the derivative with respect to Δx to zero:

$$\frac{\partial}{\partial \Delta x}\left(f(x_p) + f'(x_p)\Delta x + \frac{1}{2}f''(x_p)\Delta x^2\right) = 0$$

Simplifying this expression, we get:

$$f'(x_p) + f''(x_p)\Delta x = 0$$

Solving for Δx, we obtain:

$$\Delta x = -\frac{f'(x_p)}{f''(x_p)}$$

This gives the update rule for Newton's method. By iteratively updating the current estimate x_t using this rule, the algorithm converges toward the local minima.

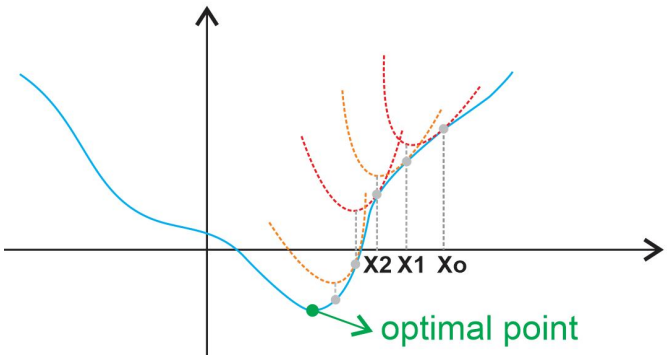


Figure 3: Newton Method Optimization

Curvature and Its Importance

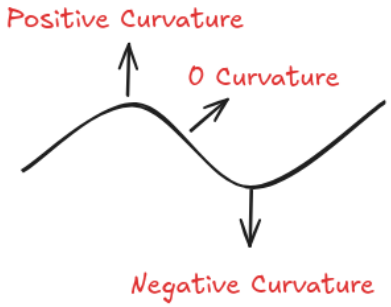


Figure 4: Curvatures

Newton's method uses second-order derivative information (the curvature) to adjust the step size more effectively compared to first-order methods like gradient descent. Curvature refers to the amount by which a curve deviates from being a straight line or a surface deviates from being a plane.

In the context of optimization, the curvature (second derivative) provides important information about the shape of the function around the current point.

A larger curvature indicates a sharper slope, while a smaller curvature indicates a flatter region.

Quasi-Newton Methods

Quasi-Newton methods are a class of optimization algorithms used to find the zeros or local maxima/minima of functions, particularly when the Jacobian (the matrix of first-order partial derivatives) or the Hessian (the matrix of second-order partial derivatives) is unavailable or computationally expensive to calculate. The update is based on gradients of the objective function, and over time, the approximation becomes more accurate, leading to faster convergence.

Quasi-Newton methods iteratively approximate the Hessian B_k and its inverse H_k to efficiently update the weights $\mathbf{w}$ during optimization. These methods rely on gradient differences and avoid direct computation of second derivatives.

1. Secant Condition

The secant condition ensures that the curvature of the objective function $f(\mathbf{w})$ is captured:

$$B_k \mathbf{s}_k = \mathbf{y}_k$$

where:

- B_k: Approximation of the Hessian matrix at iteration k,
- $\mathbf{s}_k = \mathbf{w}_{k+1} - \mathbf{w}_k$: Change in weights,
- $\mathbf{y}_k = \nabla f(\mathbf{w}_{k+1}) - \nabla f(\mathbf{w}_k)$: Change in gradients,
- $\nabla f(\mathbf{w}_k)$: Gradient of the objective function at $\mathbf{w}_k$.

2. Approximation of B_k (Hessian)

The Hessian B_k is updated iteratively as:

$$B_{k+1} = B_k + \Delta B_k$$

where ΔB_k is a correction term derived from $\mathbf{s}_k$ and $\mathbf{y}_k$. This ensures that the updated B_{k+1} satisfies the secant condition.

3. Approximation of H_k (Inverse Hessian)

The inverse Hessian $H_k = B_k^{-1}$ is updated using:

$$H_{k+1} = H_k + \Delta H_k$$

where ΔH_k satisfies:

$$H_k \mathbf{y}_k = \mathbf{s}_k$$

4. Weight Update Rule

Using the inverse Hessian approximation H_k, the weights are updated as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - H_k \nabla f(\mathbf{w}_k)$$

BFGS Method

One of the most popular Quasi-Newton methods is the BFGS algorithm.

The BFGS method updates the Hessian approximation iteratively using gradient information.

It determines the direction with curvature information, by gradually improving an estimation of Hessian matrix of

loss funciton, low complexity(no Hessian and inverse)

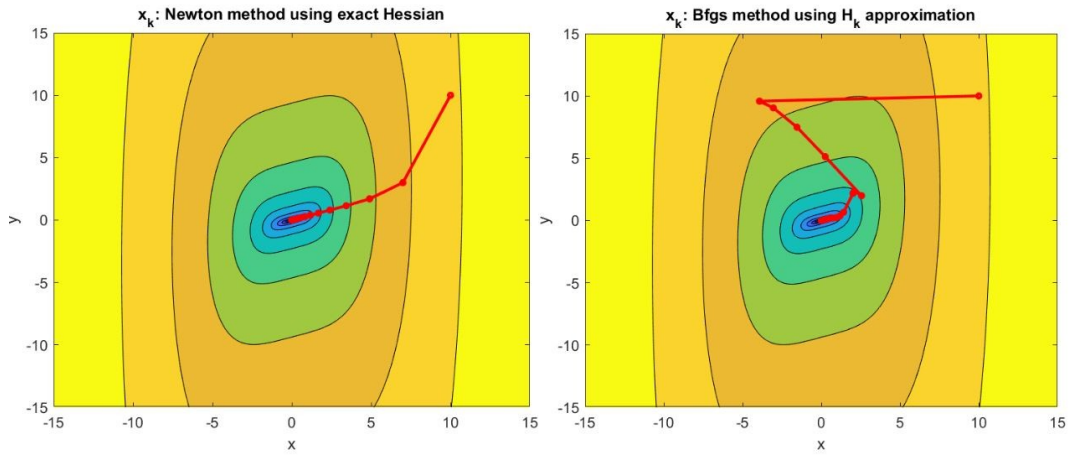


Figure 5: Newton method vs BFGS

BFGS Method for Weights w

The BFGS method iteratively approximates the Hessian B_k and its inverse H_k during optimization. It uses gradient and weight differences to update these matrices without directly computing second derivatives.

1. Secant Condition

The BFGS method satisfies the secant condition:

$$B_k \mathbf{s}_k = \mathbf{y}_k$$

2. Hessian Update Formula for B_k

The BFGS update for the Hessian approximation B_k is given by:

$$B_{k+1} = B_k - \frac{B_k \mathbf{s}_k \mathbf{s}_k^T B_k}{\mathbf{s}_k^T B_k \mathbf{s}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}$$

This ensures that B_{k+1} remains symmetric and positive definite, provided that $\mathbf{y}_k^T \mathbf{s}_k > 0$.

3. Inverse Hessian Update Formula for H_k

Instead of directly updating B_k, it is often more practical to update its inverse H_k. The BFGS update for H_k is:

$$H_{k+1} = \left(I - \frac{\mathbf{s}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) H_k \left(I - \frac{\mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\right) + \frac{\mathbf{s}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}$$

This formula ensures that H_{k+1} also remains symmetric and positive definite.

4. Weight Update Rule

The weights are updated using the inverse Hessian approximation H_k:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - H_k \nabla f(\mathbf{w}_k)$$

L-BFGS

Similar to BFGS but, instead of string a dense $d \times d$ approximation of inverse of Hessian (BFGS), it only stores few vectors that represent the approximation.

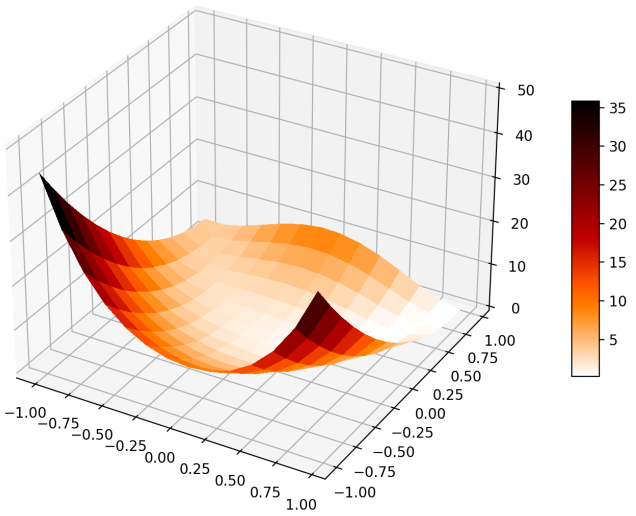


Figure 6: L-BFGS

Root Mean Square Propagation (RMSProp)

RMSProp is an adaptive learning rate optimization algorithm, designed to address some of the limitations encountered with stochastic gradient descent (SGD). Specifically, it adjusts the learning rate for each weight dynamically, allowing for smoother and more efficient training, particularly when the gradient is noisy or the optimization landscape has steep gradients in certain directions and flat regions in others.

> RMSProp maintains a moving average of the squared gradients for each weight and scales the learning rate by this average.

This helps to dampen oscillations in directions with steep gradients while allowing for faster movement in flat regions.

Steps of RMSProp

The key steps of RMSProp can be outlined as follows:

- **Compute the gradient of the cost function:**

$$\nabla J(\theta)$$

Here, θ represents the model parameters, and $J(\theta)$ is the objective function or cost function.

- **Accumulate the squared gradients:** RMSProp keeps an exponentially decaying average of past squared gradients. The accumulated squared gradient is updated as:

$$E[\nabla^2 J(\theta)]_t = \beta E[\nabla^2 J(\theta)]_{t-1} + (1-\beta)\nabla^2 J(\theta)$$

where $E[\nabla^2 J(\theta)]_t$ is the moving average of the squared gradients at time step t, and β is a decay rate (typically set to 0.9).

- **Compute the adaptive learning rate:** The learning rate for each parameter is adapted based on the accumulated squared gradients:

$$\alpha_t = \frac{\alpha}{\sqrt{E[\nabla^2 J(\theta)]_t + \epsilon}}$$

where α is the base learning rate, $E[\nabla^2 J(\theta)]_t$ is the moving average of squared gradients, and ϵ is a small constant (e.g., 10^{-8}) added to prevent division by zero.

- **Update the parameters:** The model parameters are updated using the computed adaptive learning rate:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla J(\theta)_t$$

where α_t is the adaptive learning rate and $\nabla J(\theta)_t$ is the gradient at time step t.

Benefits of RMSProp

RMSProp offers several advantages over standard SGD:

- **Faster convergence:** By dynamically adjusting the learning rate for each parameter, RMSProp can converge more quickly in many cases.
- **Stability:** RMSProp reduces oscillations in steep gradient directions, leading to more stable optimization, especially when the gradients vary significantly in magnitude.
- **Adaptation to non-stationarity:** RMSProp works well when the cost function $J(\theta)$ is non-stationary (i.e., changes over time), such as in cases of complex optimization landscapes.

Limitations of RMSProp

Despite its advantages, RMSProp has some limitations:

- **Hyperparameter tuning:** RMSProp relies on a decay rate β, a base learning rate α, and a smoothing term ϵ, all of which require careful tuning for the specific problem at hand.
- **Lack of theoretical support:** Unlike some other optimization algorithms, RMSProp does not have a formal theoretical foundation backed by research papers, though it is widely used in practice due to its empirical effectiveness.

Adaptive Moment Estimation (Adam)

Adam is an adaptive learning rate optimization algorithm that combines the benefits of both momentum and RMSProp. It is widely used for optimizing deep learning models due to its efficiency and effectiveness in handling

noisy and sparse gradients.

Adam calculates an adaptive learning rate for each parameter by using moving averages of both the gradient and the squared gradient, while also applying bias correction to counteract initial conditions.

Key Steps in Adam

- **Momentum update (First moment estimation):** Adam incorporates momentum by calculating an exponentially decaying average of past gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla J(\theta)_t$$

  where: - m_t is the first moment (the exponentially weighted average of the gradient) at time step t, - $\nabla J(\theta)_t$ is the gradient at time step t, - β_1 is the decay rate for the momentum term (typically around 0.9), which gives higher weight to recent gradients.

- **RMSProp-like update (Second moment estimation):** Adam also computes the exponentially decaying average of past squared gradients:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla^2 J(\theta)_t$$

  where: - v_t is the second moment (the exponentially weighted average of squared gradients) at time step t, - β_2 is the decay rate for the squared gradients (typically around 0.99) to capture longer-term trends in gradient magnitudes.

- **Bias correction:** Since both m_t and v_t are initialized at 0, they are biased toward 0 in the initial stages, especially when β_1 and β_2 are close to 1. Adam applies bias corrections to both moment estimates as follows:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

  These corrections ensure that the moment estimates are unbiased, particularly in the early iterations.

- **Parameter update:** The parameters θ are updated using the adaptive learning rate:

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon}$$

  where: - α is the base learning rate, - $\hat{m_t}$ is the bias-corrected first moment estimate, - $\hat{v_t}$ is the bias-corrected second moment estimate, - ϵ is a small constant (e.g., 10^{-8}) to prevent division by zero.

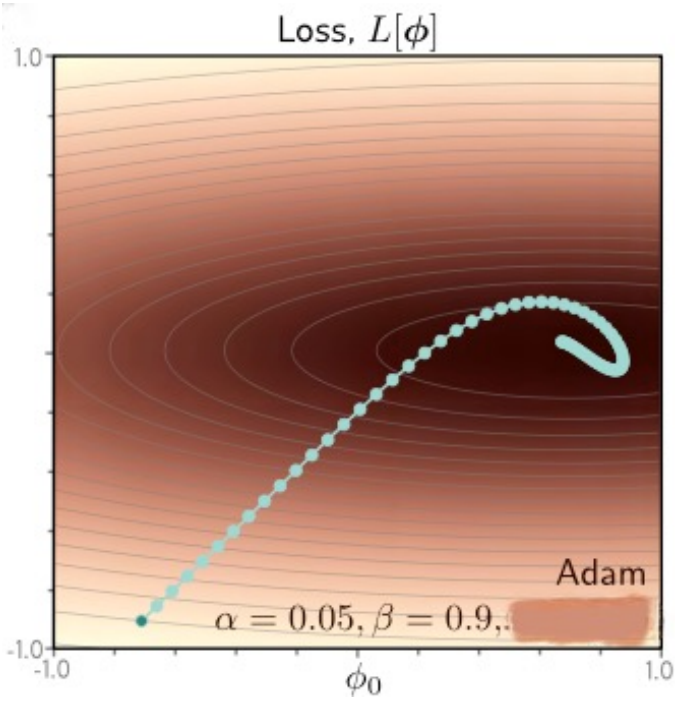


Figure 7: Adam

Advantages of Adam

- **Faster convergence:** The use of both the first and second moments helps achieve faster convergence by combining the stability of momentum with the adaptiveness of RMSProp.
- **Efficient for large datasets:** Adam works well with large datasets and is computationally efficient, making it suitable for deep learning models.
- **Robustness to noise:** The bias correction and adaptive learning rate provide robustness to noisy or sparse gradients, allowing for more stable optimization.

Limitations of Adam

- **Hyperparameter tuning:** Despite its automatic adjustment of learning rates, Adam still requires careful tuning of hyperparameters like α, β_1, and β_2 for optimal performance.
- **Non-convexity:** In non-convex optimization landscapes, Adam may still encounter saddle points or local minima, especially in deep neural networks.

Overal Challenges of Second-Order Optimization Techniques

Second-order methods are often computationally expensive due to:

- **Computation complexity:** Calculating the Hessian matrix H is $O(n^2)$, and inverting it is $O(n^3)$, which makes it impractical for large-scale problems.
- **Memory constraints:** Storing the Hessian matrix requires $O(n^2)$ memory, making it infeasible for high-dimensional models.
- **Non-convexity:** In non-convex optimization problems, second-order methods can get stuck in saddle points or suboptimal local minima, similar to first-order methods like Adam.

Optimizers specialized for Deep Generative AI scale

AdamW (Decoupled Weight Decay). AdamW is the default optimizer for LLMs. It extends Adam by *decoupling* the weight decay term from the gradient-based update, improving generalization.
Mechanics: For gradient g_t at step t,

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

Then parameters are updated as

$$\theta \leftarrow \theta - \alpha\lambda\theta \quad \text{(weight decay)}, \qquad \theta \leftarrow \theta - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

Why it matters: Classical Adam applied L2 regularization through gradients, which interacted poorly with adaptive learning rates. AdamW instead decays parameters directly, yielding more stable convergence.
Pros: Strong generalization, stable across architectures. **Cons:** Requires storing first/second moments (2× parameter memory).

Adafactor. Adafactor was introduced to train massive models (e.g., T5) without the $O(d)$ memory overhead of Adam's second-moment estimates.
Mechanics: For matrix-shaped parameter $X \in \mathbb{R}^{n\times m}$ and gradient g_t,

$$r_t = \rho r_{t-1} + (1-\rho)\,\mathrm{mean}_{\mathrm{cols}}(g_t^2), \qquad c_t = \rho c_{t-1} + (1-\rho)\,\mathrm{mean}_{\mathrm{rows}}(g_t^2),$$

$$V_t \approx \frac{r_t\, c_t^\top}{\mathrm{mean}(r_t)}, \qquad \Delta_t = \frac{g_t}{\sqrt{V_t} + \varepsilon}, \qquad \theta \leftarrow \theta - \alpha\Delta_t.$$

Why it matters: By factorizing V_t into row and column averages, memory drops from $O(nm)$ to $O(n+m)$. This enables training 11B+ parameter models on limited hardware.
Pros: Huge memory savings, competitive accuracy. **Cons:** More hyperparameters, approximation may bias updates.

Lion (Evolved Sign Momentum). Lion replaces the variance-tracking of Adam with sign-based updates, reducing memory and simplifying computation.
Mechanics (Optax form):

$$c_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad u_t = -\alpha\big(\mathrm{sign}(c_t) + \lambda\theta_t\big), \quad m_t = \beta_2 m_{t-1} + (1-\beta_2) g_t,$$

$$\theta_{t+1} = \theta_t + u_t.$$

Why it matters: Instead of scaling gradients by variance estimates, Lion takes the sign of a momentum blend. This yields robust, quantized updates with far less memory.
Pros: Extremely memory-light, often faster than AdamW on vision/LLM tasks. **Cons:** Updates can be noisy; requires careful learning-rate and weight-decay tuning.

Sophia / Sophia-G. Sophia introduces *second-order information* into large-scale training by combining gradient EMAs with diagonal Hessian approximations.
Mechanics: Maintain an EMA of the gradient:

$$m_t = \mu m_{t-1} + (1-\mu) g_t.$$

Estimate the diagonal Hessian $\hat{H}_t$ periodically via stochastic estimators (e.g., Hutchinson or Gauss-Newton). Then:

$$\tilde{u}_t = \frac{m_t}{\hat{H}_t + \varepsilon}, \qquad u_t = \mathrm{clip}(\tilde{u}_t, c), \qquad \theta \leftarrow \theta - \alpha u_t \ (+ \text{ weight decay}).$$

Why it matters: Standard Adam is purely first-order. Sophia's Hessian-aware updates take larger steps in flat directions and smaller ones in sharp regions, improving sample efficiency.
Pros: Reported $\sim$ 2× training speedup vs AdamW for GPT-style pretraining. **Cons:** Requires additional Hessian estimation and clipping logic.

Stability

Gradient Clipping (Global Norm). Gradient clipping stabilizes training by rescaling gradients that exceed a threshold. This prevents exploding gradients in very deep or recurrent networks.
Mechanics: For gradient vector g and threshold τ,

$$\mathrm{clip}(g;\tau) = \begin{cases} g, & \|g\|_2 \le \tau, \\ \dfrac{\tau}{\|g\|_2}\, g, & \text{otherwise.} \end{cases}$$

If active ($\|g\|_2 > \tau$) and $s = \tau/\|g\|_2$,

$$\frac{\partial(sg)}{\partial g} = s\Big(I - \frac{gg^\top}{\|g\|_2^2}\Big).$$

Why it matters: Exploding gradients make optimization unstable, especially in long-sequence Transformers and RNNs. Clipping ensures stable step sizes while preserving direction.
Pros: Essential for stable training of large models; simple to implement. **Cons:** Threshold τ must be tuned; too low can under-train.

Learning-Rate Warmup + Cosine Decay. Large models often diverge if trained immediately with a high learning rate. Warmup schedules gradually increase the learning rate, while cosine decay smoothly reduces it later for better convergence.
Mechanics: For step $t \in [0, T]$, warmup length T_w:

$$\alpha(t) = \begin{cases} \alpha_{\max}\, \dfrac{t}{T_w}, & 0 \le t < T_w, \\ \alpha_{\min} + \frac{1}{2}(\alpha_{\max} - \alpha_{\min})\Big[1 + \cos\Big(\pi\, \frac{t - T_w}{T - T_w}\Big)\Big], & T_w \le t \le T. \end{cases}$$

Why it matters: Warmup prevents unstable parameter jumps in early training when weights are uninitialized. Cosine decay avoids abrupt drops, maintaining smooth convergence.
Pros: Robust schedule across architectures; prevents early divergence. **Cons:** Requires setting warmup length; may decay too aggressively for continual pretraining.

Pre-Norm Transformers & RMSNorm. Normalization strategies affect gradient flow in very deep networks. Pre-Norm and RMSNorm were introduced to stabilize LLM training.
Mechanics (Pre-Norm block):

$$x \leftarrow x + \mathrm{MSA}(\mathrm{LN}(x)), \qquad x \leftarrow x + \mathrm{MLP}(\mathrm{LN}(x)).$$

Here each residual branch normalizes input before the sublayer.
Mechanics (RMSNorm):

$$y = \gamma \frac{x}{r}, \quad r = \sqrt{\tfrac{1}{n}\sum_i x_i^2 + \varepsilon}.$$

Jacobian:

$$\frac{\partial y_i}{\partial x_j} = \gamma\Big(\frac{\delta_{ij}}{r} - \frac{x_i x_j}{n\, r^3}\Big), \qquad \frac{\partial y}{\partial \gamma} = \frac{x}{r}.$$

Why it matters: Pre-Norm improves gradient flow in deep Transformers vs Post-Norm. RMSNorm removes mean subtraction from LayerNorm, reducing compute and improving stability in half-precision training.
Pros: Enables deeper stacks; more efficient than LayerNorm. **Cons:** Slightly less expressive than LayerNorm; residual scaling may need tuning.

Label Smoothing. Label smoothing regularizes classification and language modeling by softening one-hot targets. This prevents the model from becoming overconfident.
Mechanics: For K classes, smoothing factor α:

$$\tilde{y} = (1-\alpha)\, y + \alpha\, \tfrac{\mathbf{1}}{K}.$$

Cross-entropy loss:

$$\mathcal{L} = -\sum_{k=1}^{K} \tilde{y}_k \log p_k, \qquad \frac{\partial \mathcal{L}}{\partial z_k} = p_k - \tilde{y}_k,$$

where $p = \mathrm{softmax}(z)$ are predicted probabilities.
Why it matters: Without smoothing, models tend to assign near-1.0 probabilities to predicted tokens, leading to poor calibration. Label smoothing encourages uncertainty estimation and improves generalization.
Pros: Better calibration; prevents overconfidence. **Cons:** Slight reduction in peak accuracy for easy tasks; requires choosing α.

References

[1] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023.