Ch 4: Advanced Optimization Algorithms for AI

Part I: ADAM and Its Variants

Dong-Young Lim
UNIST

AI51101, IE55101

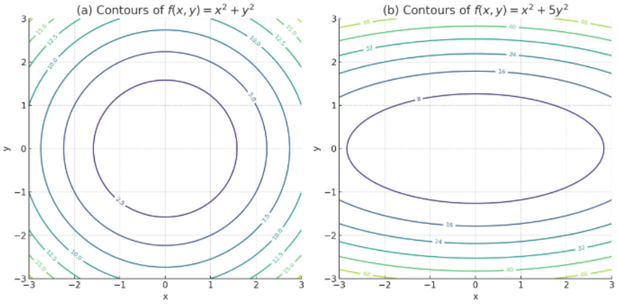
OBJECTIVE & ITS CHALLENGES

- **Objective:** Find a set of parameters θ that minimizes a high-dimensional and non-convex loss function $f(\theta)$.
- (Stochastic) Gradient Descent (GD) is a first-order iterative method that updates parameters based solely on the gradient at the current point (local information):

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

- However, the loss landscapes of complex models present significant geometric challenges where this local approach is insufficient. Key issues include:
  - **Differential Curvature:** Different parameters have vastly different sensitivities.
  - **Flat Surfaces and Saddle Points:** Regions where the gradient is near-zero, severely slowing down convergence.
  - **Steep Cliffs:** Regions of excessively large gradients, leading to unstable updates.

Challenge I: Differential Curvature



- In many problems, the loss function forms a "ravine" - steep in one direction but gentle in another. This is called **differential curvature**.
- The left contour ($f = x^2 + y^2$) is a simple bowl. Any step downhill points to the center.
- The right contour ($f = x^2 + 5y^2$) is an elliptical bowl.
  - The gradient (steepest descent direction) points almost perpendicular to the ravine's floor.
  - This leads to a **zigzag pattern**: overshooting across the narrow axis while making slow progress along the main axis. **A single learning rate** can't be both large enough for the gentle slope and small enough for the steep slope.

Challenge I: Differential Curvature

(a) Basin

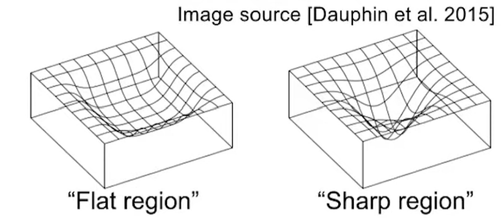
(b) Ravine

Challenge I: Differential Curvature

- **Question:** Can we solve the differential curvature problem by simply normalizing the input features?
- Normalization adjusts the scale of input features to be in a similar range. This can indeed help make the loss contours more uniform and is a recommended practice.
- **However, it is not a fundamental solution.**
  - The core problem of differential curvature is the varying sensitivity of the loss function with respect to each parameter θ.
  - Normalizing the inputs x is different from normalizing the parameter sensitivities θ. While the former can influence the latter, it does not directly solve the underlying issue.

CHALLENGE II: PLATEAUS & SADDLE POINTS

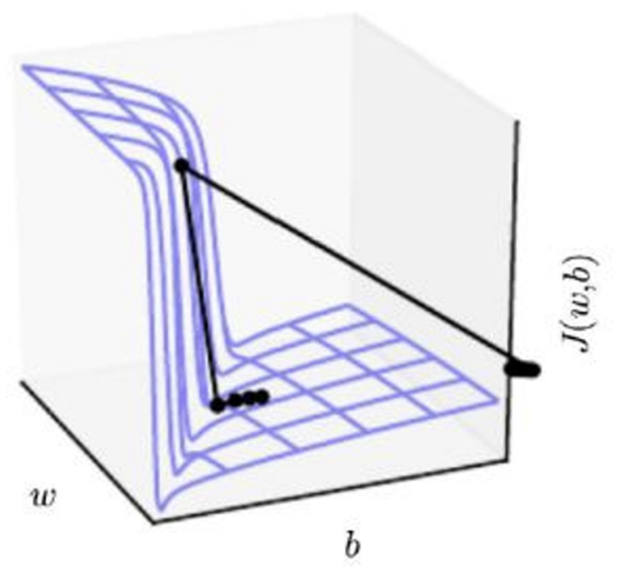
- **Saddle Points:** A point where the gradient is zero ($\nabla f(\theta) = 0$) but it is not a local extremum.
  - In high-dimensional functions, saddle points are exponentially more prevalent than local minima.
  - Standard GD algorithms are prone to get stuck near saddle points as the gradient diminishes.



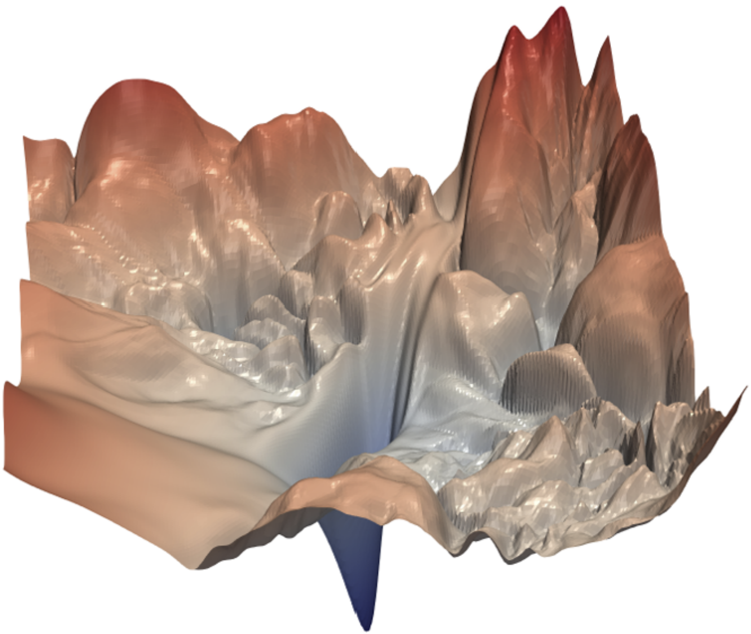
- **Flat regions (Plateaus):** Nearly flat regions in the loss landscape.
  - In these regions, the gradient is consistently close to zero, which can halt the learning process. This is known as the **vanishing gradient problem**.

Challenge III: Steep Cliffs

- The loss landscape can contain extremely steep regions, often called "cliffs".
- Descending a cliff can cause an excessively large update, leading to the **exploding gradient** problem and unstable training.



Loss Landscape in Deep Learning



Beyond Standard Gradient Descent

- As we have seen, the complex loss landscapes in deep learning present significant challenges for standard Gradient Descent.
- To overcome these challenges, various strategies that go **beyond the standard Gradient Descent** algorithm have been developed.
- We will explore two principal approaches:
  1. **Acceleration using Momentum**
  2. **Adaptive Learning Rates (ADAM and its variants)**

Gradient Descent with Momentum

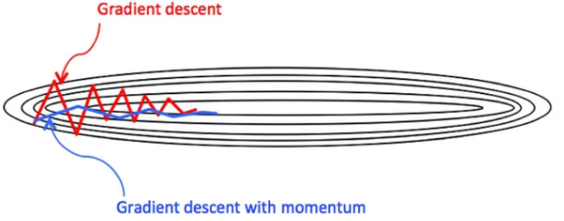
- **Key Idea:** Don't just rely on the local information, i.e., the current gradient. Accumulate past gradients to create "velocity" and accelerate the descent.
- **Physical Analogy:** A heavy ball rolling down a hill.
  - **Standard GD** is like a massless ball. It follows the steepest path at each point but stops instantly (No inertia).
  - **GD with Momentum** is like a rolling ball. It builds up speed (momentum) as it rolls, making it less affected by noisy gradients.
  - That's why GD with momentum is sometimes called "heavy-ball method"
- This accumulated velocity helps to:
  - Accelerate through flat plateaus where the gradient is small.
  - Dampen oscillations in narrow ravines.

Gradient Descent with Momentum

- Momentum tracks an **exponential moving average (EMA)** of the gradients, which is treated as the current velocity.
- **Update Rules:** Let $g_t = \nabla f(\theta_t)$ (or, represent the stochastic gradient such that $\mathbb{E}[g_t] = \nabla f(\theta_t)$ where the randomness of the data is omitted.)
  1. Update velocity v_t using the previous velocity and current gradient:
  $$v_t = \beta v_{t-1} + (1 - \beta) g_t$$
  2. Update parameters θ_t:
  $$\theta_{t+1} = \theta_t - \gamma v_t$$
- **Key Components:**
  - v_t: The "velocity" vector, representing the direction and speed of the update.
  - β: The momentum coefficient (e.g., 0.9). It controls how much of the past velocity is retained.
    - A high β means past gradients have a strong, lasting influence (less friction).
  - γ: The learning rate.

Gradient Descent with Momentum

- **1. How it Navigates Ravines (Differential Curvature):**
  - Oscillating gradients (across the ravine) are in opposite directions and **cancel each other out** in the moving average.
  - Consistent gradients (along the ravine floor) point in the same direction and **accumulate**, building up velocity.
  - As a result, it reduces the zigzag motion and accelerates progress.
- **2. How it Escapes Plateaus & Saddle Points:**
  - When the optimizer encounters a flat region, the current gradient g_t becomes nearly zero.
  - However, the stored velocity v_t allows the optimizer to go through the flat area and push past the saddle point, where standard GD would stop.



From Momentum to Adaptive Rates

- **Recap:** Momentum helps accelerate through flat regions and reduces oscillations.
- However, momentum still uses a **single & fixed learning rate** for all parameters.
- In a ravine, we ideally want to move slowly across the steep axis but quickly along the flat axis. This requires different step sizes for different directions.

$\Rightarrow$ What if we could adapt the learning rate for each parameter individually? This leads us to our second strategy: **Preconditioning**.

From Momentum to Adaptive Rates

- **Recap:** Momentum helps accelerate through flat regions and reduces oscillations.
- However, momentum still uses a **single & fixed learning rate** for all parameters.
- In a ravine, we ideally want to move slowly across the steep axis but quickly along the flat axis. This requires different step sizes for different directions.

$\Rightarrow$ This brings us to our second principal strategy (adaptive learning rate):

  - **Adaptive learning rate** scheme uses a different learning rate for each parameter at each iteration.
  - This is formally achieved through a mechanism known as **Preconditioning**.

Preconditioning

- An optimization problem is **ill-conditioned** if the loss function's curvature dramatically varies across different directions.
- **Condition Number (κ):**
  - The degree of ill-conditioning is measured by the **condition number** of the Hessian:

    $$\kappa(H) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}$$

    where λ_{max} and λ_{min} are the largest and smallest eigenvalues of H.
  - Eigenvalues represent curvature. A large κ means a steep and narrow ravine, which makes gradient descent less effective. On the other hand, a perfect circle has $\kappa = 1$.
- The goal of preconditioning is to transform an ill-conditioned problem into a well-conditioned one.

Preconditioning

- **Key Idea:** Reshape the parameter space to make the loss contours more spherical (circular).
- This is achieved by multiplying the gradient by a matrix P_t^{-1}, called the **preconditioner**.
- **General Update Rule:**

  $$\theta_{t+1} = \theta_t - \gamma P_t^{-1} g_t$$

  The preconditioner P_t approximates the curvature of the loss function. Its inverse, P_t^{-1}, scales the gradient components: it shrinks steps in high-curvature directions and enlarges them in low-curvature directions.
- With preconditioning, parameter updates become balanced according to the local curvature of the loss surface.

PRECONDITIONING

- Consider the objective function:

$$f(x, y) = \frac{1}{2}(10x^2 + y^2)$$

  with $\theta_0 = (1, 10)$.
- **1. Standard Gradient Descent Step**
  - The gradient at this point is: $g_0 = \nabla f(1, 10) = [10x, y]|_{(1,10)} = [10, 10]$.
- **2. Preconditioned Step**
  - Choose the Hessian as the preconditioner: $P = H = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}$.
  - The preconditioned gradient is:

$$P^{-1}g_0 = \begin{bmatrix} 1/10 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \end{bmatrix} = \begin{bmatrix} 1 \\ 10 \end{bmatrix}$$

- **Effect of Preconditioning:** preconditioning creates an adaptive learning rate- small for sensitive directions (high curvature) and large for stable directions (low curvature).

Preconditioning

- In the previous toy example, we used the true Hessian matrix as the preconditioner. This represents the **ideal** case.
- **Ideal Preconditioner:** The Hessian (H_t)
  - The Hessian fully captures the curvature of the loss function, and using it ($P_t = H_t$) perfectly reshapes the problem space. This is known as **Newton's Method**.
  - **The Problem:** For a model with d parameters, computing and inverting the Hessian is computationally expensive and this is infeasible for large-scale neural networks.
- **Practical Approach: Diagonal Approximation**
  - In deep learning, we use cheap approximations. The most common is a **diagonal matrix**, which assumes zero correlation between parameters. This is the core idea behind:
    - **AdaGrad, RMSProp, ADAM, and their variants**

AdaGrad

- AdaGrad (2011, JMLR) implements the diagonal preconditioning idea by the accumulated magnitude of past gradients.
- **Mechanism:**
  - Let $g_t = \nabla f(\theta_t)$.

  1. Accumulate squared gradients (element-wise):

$$s_t = s_{t-1} + g_t^2$$

  2. Update parameters:

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\sqrt{s_t + \epsilon}} g_t$$

- It performs well on problems with sparse features, as infrequent parameters receive larger updates.
- **Limitation:** The accumulator s_t grows monotonically. As a result, the effective learning rate for all parameters steadily decreases and eventually becomes very small, causing the model to stop learning prematurely.

RMSProp

- RMSProp (2012, unpublished) addresses AdaGrad's critical weakness by preventing the learning rate from monotonically decreasing.
- **Mechanism:**
  - Instead of summing all past squared gradients, it uses an **exponential moving average (EMA)**, which allows old gradients to be "forgotten".

  1. Update the EMA of squared gradients (element-wise):

$$s_t = \rho s_{t-1} + (1-\rho) g_t^2, \quad s_{-1} = 0$$

  2. Update parameters:

$$\theta_{t+1} = \theta_t - \frac{\gamma}{\sqrt{s_t + \epsilon}} g_t$$

- Here, ρ is a decay rate, typically set to 0.9 or 0.99. By keeping a moving average, the denominator does not grow indefinitely.
- **Limitation:** The accumulator s_t is initialized to zero, making it biased towards zero in the early stages of training. This can cause unintentionally large parameter updates initially.

Adam

- Adam (2015, ICLR) is arguably the most popular adaptive optimizer, effectively combining the ideas of **Momentum** and **RMSProp**.
- It computes and maintains two separate exponential moving averages:
  1. **First Moment (m_t):** EMA of the gradients like Momentum.
  2. **Second Moment (v_t):** EMA of the squared gradients (like RMSProp).
- **The Full Adam Algorithm:**
  1. Compute gradient: $g_t = \nabla f(\theta_{t-1})$
  2. Update 1st moment: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  3. Update 2nd moment: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  4. Correct bias in 1st moment: $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
  5. Correct bias in 2nd moment: $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
  6. Update parameters: $\theta_t = \theta_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$

Adam Property I: Bias Correction

- **The Problem:** The moment vectors, m_t and v_t, are initialized as zeros. This causes them to be biased towards zero, especially during the initial steps of training.
- **The Solution:** Adam introduces bias-correction terms that analytically compute the expected value of the moments and scale them appropriately.

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

- By calculating the expectation, we see that

$$\mathbb{E}[\hat{m}_t] = \mathbb{E}[g_t]$$

and

$$\mathbb{E}[\hat{v}_t] = \mathbb{E}[g_t^2]$$

- This correction ensures that the update step has a proper magnitude from the very beginning of training.

Adam Property I: Bias Correction

- Assume a stationary process with $\mathbb{E}[g_t] = \mu$ and $\mathbb{E}[g_t^2] = \nu$ in the early phase.

$$\mathbb{E}[m_t] = (1-\beta_1)\sum_{k=1}^{t} \beta_1^{t-k}\mu = (1-\beta_1^t)\mu \;\Rightarrow\; \mathbb{E}[\hat{m}_t] = \mu.$$

$$\mathbb{E}[v_t] = (1-\beta_2)\sum_{k=1}^{t} \beta_2^{t-k}\nu = (1-\beta_2^t)\nu \;\Rightarrow\; \mathbb{E}[\hat{v}_t] = \nu.$$

- Thus $\hat{m}_t$ and $\hat{v}_t$ are nearly unbiased in the early phase of training.

Adam Property II: Scale Invariance

- The magnitude of Adam's update step is largely invariant to the scale of the gradients.
- If we rescale the gradient by a constant factor c (i.e., $g'_t = c \cdot g_t$), how does the update change?
  - The first moment estimate scales by c: $\hat{m}'_t = c \cdot \hat{m}_t$
  - The second moment estimate scales by c^2: $\hat{v}'_t = c^2 \cdot \hat{v}_t$
- The final update term becomes:

$$\frac{\hat{m}'_t}{\sqrt{\hat{v}'_t + \epsilon}} \approx \frac{c \cdot \hat{m}_t}{\sqrt{c^2 \cdot \hat{v}_t}} = \frac{c \cdot \hat{m}_t}{|c| \cdot \sqrt{\hat{v}_t}} = \operatorname{sign}(c)\frac{\hat{m}_t}{\sqrt{\hat{v}_t}}$$

- **Implication:** The magnitude of the update is unaffected by the scaling factor c. This makes Adam **robust** to objective functions that produce very large or very small gradients.

Adam Property III: Bounded Step Size

- Each coordinate's step is effectively capped by γ in stationary regimes.
- The per-coordinate update is:
$$|\Delta\theta_{t,i}| = \gamma \frac{|\hat{m}_{t,i}|}{\sqrt{\hat{v}_{t,i}} + \epsilon}$$
  By Cauchy–Schwarz on EMAs (stationary regime),
$$|\hat{m}_{t,i}| \leq \sqrt{\hat{v}_{t,i}} \quad \Rightarrow \quad |\Delta\theta_{t,i}| \leq \gamma \frac{\sqrt{\hat{v}_{t,i}}}{\sqrt{\hat{v}_{t,i}} + \varepsilon} \leq \gamma.$$
- **Intuition:** Adam has a built-in trust region property. It prevents the optimizer from taking excessively large steps, which greatly improves training stability, especially in regions with noisy or large gradients.

ADAM Property IV: Noise-to-Signal Ratio

- We can gain a deeper intuition for Adam's behavior by viewing it through the lens of the Noise-to-Signal Ratio (NSR):
  - **Signal:** The consistent direction of descent, estimated by the mean of the gradients ($m_t \approx \mathbb{E}[g_t]$).
  - **Noise:** The uncertainty or fluctuation of the gradients, measured by their standard deviation ($\sqrt{\mathrm{Var}(g_t)}$).
  - NSR measures the relative uncertainty of the gradient for a given parameter:

$$\mathrm{NSR} = \frac{\sqrt{\mathrm{Var}(g_t)}}{|\mathbb{E}[g_t]|}$$

  - A **high NSR** indicates a noisy, unreliable gradient, while a **low NSR** indicates a consistent, reliable gradient.
- How Adam Responds to NSR:
  - When **NSR is high** (high noise), the variance term dominates $\hat{v}_t$. This increases the denominator, which in turn **reduces the step size**.
  - When **NSR is low** (high signal), the mean term dominates, leading to a relatively **larger step size**.

ADAM: Default Setting

- Adam:
    1. Compute gradient: $g_t = \nabla f(\theta_{t-1})$
    2. Update 1st moment: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    3. Update 2nd moment: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    4. Correct bias in 1st moment: $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
    5. Correct bias in 2nd moment: $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
    6. Update parameters: $\theta_t = \theta_{t-1} - \gamma \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$
- Default Setting:

$$\gamma = 10^{-3},\ \beta_1 = 0.9,\ \beta_2 = 0.999,\ \varepsilon = 10^{-8}.$$

ADAM: THE CONVERGENCE

- The original Adam paper provided a proof of convergence for online convex optimization, showing a favorable $O(\sqrt{T})$ regret bound.
- However, subsequent research identified a flaw in the proof. It was demonstrated through counterexamples that **Adam can fail to converge** even on simple convex problems in the paper of AMSGrad.
- Since this flaw, analyzing the (non-)convergence of Adam has become a interesting topic of research in the optimization community.

On the convergence of **adam** and beyond
SJ Reddi, S Kale, S Kumar - arXiv preprint arXiv:1904.09237, 2019 - arxiv.org
… only prove **non-convergence** of **ADAM** in the … of **ADAM** and AMSGRAD on synthetic example on a simple one dimensional convex problem inspired by our examples of **non-convergence**…
☆ 저장 99 인용 3666회 인용 관련 학술자료 전체 9개의 버전

Non-convergence and limit cycles in the **adam** optimizer
S Bock, M Weiß - International Conference on Artificial Neural Networks, 2019 - Springer
… This is done for the **Adam** algorithm in batch mode only, but … to clarify the global **non-convergence** of **Adam** even for strictly … 2 we define our variant of the **Adam** algorithm and explain …
☆ 저장 99 인용 18회 인용 관련 학술자료 전체 7개의 버전

Non-convergence of **Adam** and other adaptive stochastic gradient descent optimization methods for non-vanishing learning rates
S Dereich, R Graeber, A Jentzen - arXiv preprint arXiv:2407.08100, 2024 - arxiv.org
… and the **Adam** … **Adam** optimizer fail to converge to any possible random limit point if the learning rates are asymptotically bounded away from zero. In our proof of this **non-convergence** …
☆ 저장 99 인용 12회 인용 관련 학술자료 전체 4개의 버전

On the convergence of a class of **adam**-type algorithms for non-convex optimization

AMSGrad

- We will examine a simple 1D **convex** problem where Adam is shown to converge to a suboptimal solution.
- Problem Setup: Online Convex Optimization
  - **Domain:** $\mathcal{F} = [-1, 1]$.
  - **Function Sequence:** A repeating 3-step sequence of functions.

$$f_t(x) = \begin{cases} Cx & \text{if } t \mod 3 = 1 \\ -x & \text{otherwise} \end{cases} \quad \text{(for a large constant } C > 2\text{)}$$

  - **Gradients:** g_t is C once, then a small negative value -1 twice.
- **Goal:** Find the point that minimizes the cumulative loss:

$$F_{cycle}(x) = Cx + (-x) + (-x) = (C - 2)x$$

Since $C > 2$, the term $(C - 2)$ is positive. To minimize this on the domain $[-1, 1]$, we must choose the smallest possible value -1 for x. (You need to study online learning for full details.)

DETOUR: A REGRET ANALYSIS

- **The Goal of Online Learning:** The algorithm doesn't know the future loss functions. Its goal is to make a sequence of choices $(x_1, x_2, \dots)$ that, in total, performs nearly as well as a hypothetical "expert" who knew everything in advance.
- We define this expert's performance as the best possible outcome using a single and fixed point x^* over the entire history. This provides a stable and consistent **benchmark** to measure our algorithm against. The ultimate goal of training a model is often to find one final set of parameters, so this benchmark is very practical.
- We measure performance by **Regret**, R_T, which is the difference between our algorithm's total loss and the expert's total loss.

$$R_T = \underbrace{\sum_{t=1}^{T} f_t(x_t)}_{\text{Algorithm's Total Loss}} - \underbrace{\min_{x^* \in \mathcal{F}} \sum_{t=1}^{T} f_t(x^*)}_{\text{Best Fixed Point's Total Loss}}$$

- A good algorithm ensures that its average regret, R_T/T, approaches zero as $T \to \infty$. This means our algorithm, on average, performs as well as the best fixed point in hindsight.

AMSGrad

- To demonstrate the failure, we uses specific parameters that simplify the analysis: $\beta_1 = 0$ (so $m_t = g_t$) and a small β_2.
- Case 1: The Noraml Step ($g_t = -1$)
  - Most of the time, the gradient is -1. The second moment estimate v_t (the EMA of squared gradients) will therefore converge to $(-1)^2 = 1$ when β_2 is very small.
  - The parameter update is:

$$\Delta\theta = -\gamma \frac{g_t}{\sqrt{v_t + \epsilon}} \approx -\gamma \frac{-1}{\sqrt{1}} = +\gamma$$

  - **Observation:** During these iterations, small-gradient steps, Adam moves towards +1.

AMSGrad

- Case 2: The "Spike" Step ($g_t = C$)
  - When the rare, large gradient C appears, v_t spikes.
  - For a small β_2, the denominator $\sqrt{v_t}$ becomes approximately C.
  - The parameter update is:
  $$\Delta\theta = -\gamma \frac{g_t}{\sqrt{v_t + \epsilon}} \approx -\gamma \frac{C}{C} = -\gamma$$
  - **Observation:** The large and informative gradient is scaled down so much that it only produces a small step of size γ in the correct direction.

- Then, the net change over one 3-step cycle:
  - Step 1 (Spike): $\Delta\theta \approx -\gamma$
  - Step 2 (Normal): $\Delta\theta \approx +\gamma$
  - Step 3 (Normal): $\Delta\theta \approx +\gamma$
- Total Change over 3 steps $\approx (-\gamma) + (+\gamma) + (+\gamma) = +\gamma$
- Adam consistently drifts in the wrong direction (towards $+1$), away from the true minimum at -1.
- This shows a fundamental flaw of ADAM under a rare and large gradients environment.

AMSGRAD

- AMSGrad (2018) fixes this by adding a **"long-term memory"** to the second moment, guaranteeing a non-increasing adaptive learning rate.
- **The Key Modification:**
  - AMSGrad maintains the maximum of the second moment estimates seen so far:

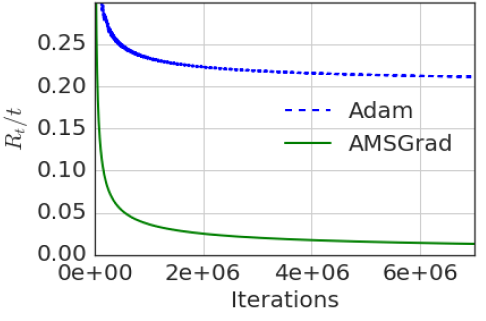
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

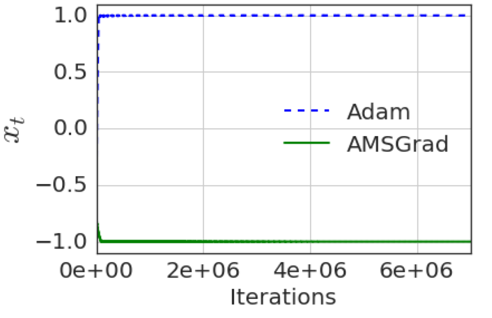
.

- **How this solves the non-convergence issue:**
  - By using the running maximum, AMSGrad ensures the denominator used for scaling never shrinks.
  - Once it encounters a large squared gradient (like C^2 from our example), the denominator will remain large for all future steps. This stabilizes the adaptive learning rate, preventing it from becoming inappropriately large.

AMSGrad



R_t/t
0.25
0.20
0.15
0.10
0.05
0.00
0e+00
2e+06
4e+06
6e+06
Iterations
Adam
AMSGrad



x_t
1.0
0.5
0.0
−0.5
−1.0
0e+00
2e+06
4e+06
6e+06
Iterations
Adam
AMSGrad

AMSGrad

- There exist convex problems where Adam has non-zero average regret under $\beta_1 < \sqrt{\beta_2}$.

  Theorem 2. *For any constant $\beta_1, \beta_2 \in [0,1)$ such that $\beta_1 < \sqrt{\beta_2}$, there is an online convex optimization problem where* Adam *has non-zero average regret i.e., $R_T/T \nrightarrow 0$ as $T \rightarrow \infty$.*

- AMSGrad achieves $R(T) = O(\sqrt{T})$ in convex settings.

  Theorem 4. *Let $\{x_t\}$ and $\{v_t\}$ be the sequences obtained from Algorithm 2, $\alpha_t = \alpha/\sqrt{t}$, $\beta_1 = \beta_{11}$, $\beta_{1t} \leq \beta_1$ for all $t \in [T]$ and $\gamma = \beta_1/\sqrt{\beta_2} < 1$. Assume that $\mathcal{F}$ has bounded diameter D_∞ and $\|\nabla f_t(x)\|_\infty \leq G_\infty$ for all $t \in [T]$ and $x \in \mathcal{F}$. For x_t generated using the* AMSGrad *(Algorithm 2), we have the following bound on the regret*

$$R_T \leq \frac{D_\infty^2 \sqrt{T}}{\alpha(1-\beta_1)} \sum_{i=1}^{d} \hat{v}_{T,i}^{1/2} + \frac{D_\infty^2}{(1-\beta_1)^2} \sum_{t=1}^{T} \sum_{i=1}^{d} \frac{\beta_{1t} \hat{v}_{t,i}^{1/2}}{\alpha_t} + \frac{\alpha\sqrt{1+\log T}}{(1-\beta_1)^2(1-\gamma)\sqrt{(1-\beta_2)}} \sum_{i=1}^{d} \|g_{1:T,i}\|_2.$$

ADAMW

- In standard SGD, there are two common ways to implement regularization:
  - L_2 **Regularization:** Add the L_2 penalty term to the objective.

    $$\min_{\theta} f(\theta) + \lambda\|\theta\|_2^2$$

    Therefore, the gradient and the update rule become

    $$g_t = \nabla f(\theta_t) + \lambda\theta_t$$

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

  - **Weight Decay:** Directly subtract a fraction of the weights from the parameters after the gradient step.

    $$\theta_{t+1} = (1 - \gamma\lambda)\theta_t - \gamma\nabla f(\theta_t)$$

- **For SGD, these two forms are identical.** This equivalence has led many to use the terms interchangeably. However, we will see this is not true for adaptive optimizers.

ADAMW

```
optimizer = torch.optim.Adam(lr=0.001, weight_decay=1e-5)
```

ADAMW

- Adam uses a diagonal preconditioner, $D_t = \text{diag}((\hat{v}_t + \epsilon)^{-1/2})$, to scale the gradients.
- **Naive L_2-regularization to ADAM**: When L2 regularization is naively added to the gradient *before* preconditioning, the update becomes:

$$\theta_{t+1} = \theta_t - \gamma D_t(\hat{m}_t + \lambda\theta_t)$$

- **Critical Flaw:** The regularization term $\lambda\theta_t$ is now also scaled by the adaptive matrix D_t.
  - This is no longer equivalent to minimizing a standard L2 penalty, $\frac{\lambda}{2}\|\theta\|_2^2$.
  - Instead, it's equivalent to minimizing a complex, time-varying weighted penalty: $\frac{\lambda}{2}\theta^\top D_t\theta$.
- Parameters with large historical gradients will have a large $\hat{v}_t$, making their corresponding entry in D_t small. As a result, they receive **less regularization**, which is the opposite of the desired regularization effect.

ADAMW

- AdamW solves this by **decoupling** the weight decay from the adaptive gradient update.
- **AdamW Update Rule:**

$$\begin{aligned}\theta_{t+1} &= (1-\gamma\lambda)\theta_t - \gamma D_t \hat{m}_t \\ &= \theta_t - \gamma D_t \hat{m}_t - \gamma\lambda\theta_t.\end{aligned}$$

  1. First, the adaptive gradient step is calculated based on the moments: $-\gamma D_t \hat{m}_t$.
  2. Separately, the weight decay is applied uniformly to all weights: $-\gamma\lambda\theta_t$.
- This restores the intended effect of weight decay. The amount of weight shrinkage is now independent of the gradient history, leading to more stable and predictable regularization.
- Default Settings: $\gamma = 0.001$, $\lambda = 0.01$.

ADAMW

- **Proximal veiw**: The AdamW update rule can be derived as the solution to the following optimization problem at each step:

$$\theta_{t+1} = \arg\min_{\theta} \underbrace{\langle \hat{m}_t, \theta - \theta_t \rangle + \frac{1}{2\gamma}\|\theta - \theta_t\|^2_{D_t^{-1}}}_{\text{Adaptive Gradient Step}} - \underbrace{\gamma\lambda\theta_t}_{\text{L2 Penalty}}$$

  where $\|\theta\|_{D_t^{-1}} := \theta^\top D_t^{-1}\theta$.
  - The first two terms aim to follow the momentum ($\hat{m}_t$) but penalize moving too far from the current point (θ_t).
  - The third term is the standard L_2 penalty.
- Taking the first-order condition of this objective and solving for θ_{t+1} gives us the exact AdamW update rule:

$$\theta_{t+1} = \theta_t - \gamma D_t \hat{m}_t - \gamma\lambda\theta_t$$

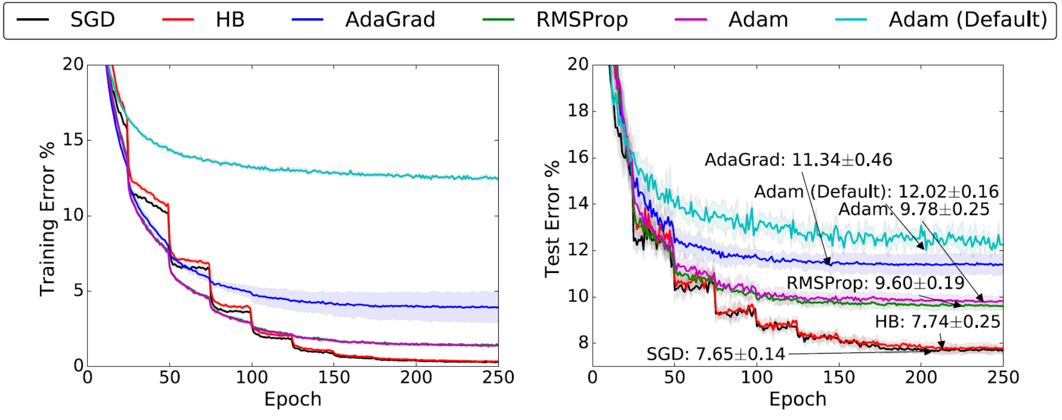
A New Question: Do Faster Optimizers Generalize Better?

- By the late 2010s, Adam and its variants became the mainstream choice for training deep neural networks due to their remarkably fast training speed.
- **Emerging Problems:**
  - However, models trained with Adam, despite reaching low training error quickly, often showed worse performance on the **test set** compared to models trained with simple, slower SGD.
  - The "best" optimizer for training loss was not always the "best" for test accuracy.
- **Critical Questions:**
  - Does the choice of optimizer influence not just *how fast* we find a solution, but ***which*** solution we find?
  - And do different optimizers converge to solutions with fundamentally different **generalization abilities**?

SGD VS ADAM-LIKE OPTIMIZERS

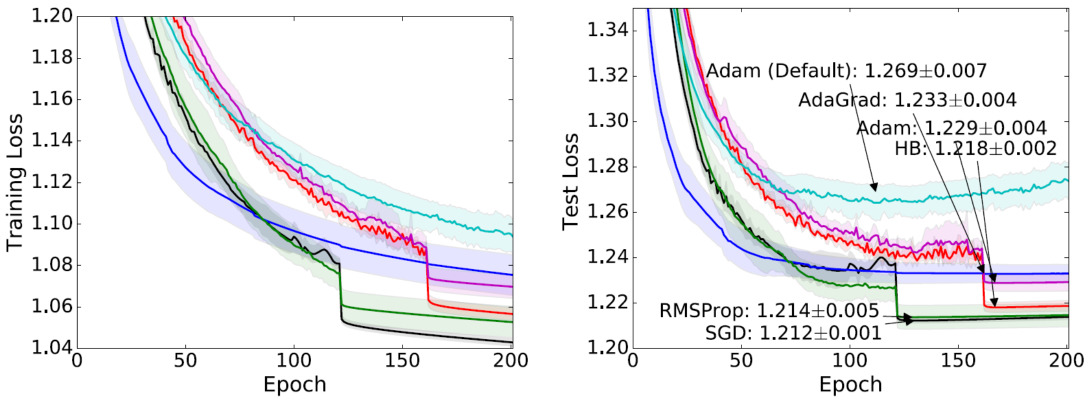
- A seminal paper by Wilson et al. (2017) provided a striking answer to this question.
- **Main Claim:** Adaptive methods like Adam can find fundamentally different solutions from SGD, and these solutions often **generalize significantly worse**.
- **Key Findings:**
  - They constructed a simple convex problem where SGD achieves perfect generalization (0% test error), while Adam fails completely (50% test error).
  - They showed empirically across multiple deep learning tasks that well-tuned SGD outperforms well-tuned Adam on test performance, even when Adam's training loss is lower.

SGD VS ADAM-LIKE OPTIMIZERS



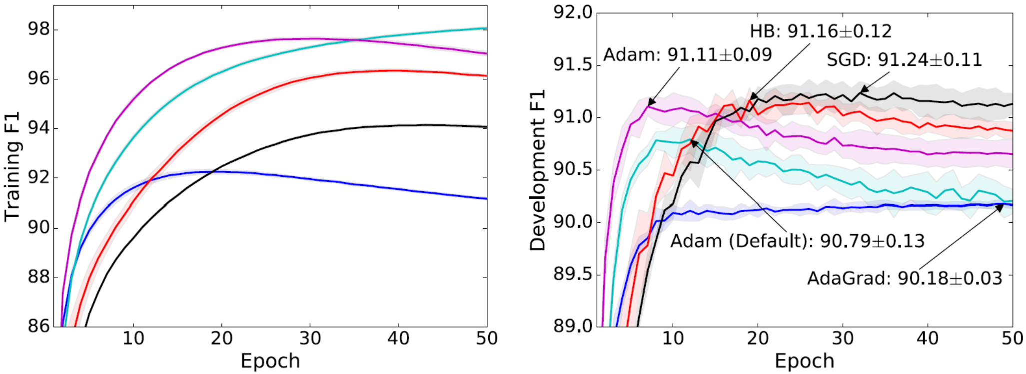
(a) CIFAR-10 (Train) **(b)** CIFAR-10 (Test)

SGD VS ADAM-LIKE OPTIMIZERS



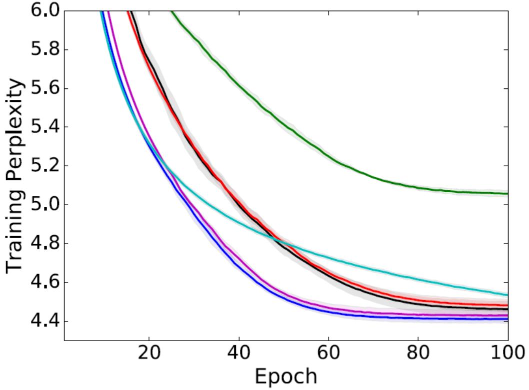
(a) War and Peace (Training Set) **(b)** War and Peace (Test Set)

SGD VS ADAM-LIKE OPTIMIZERS

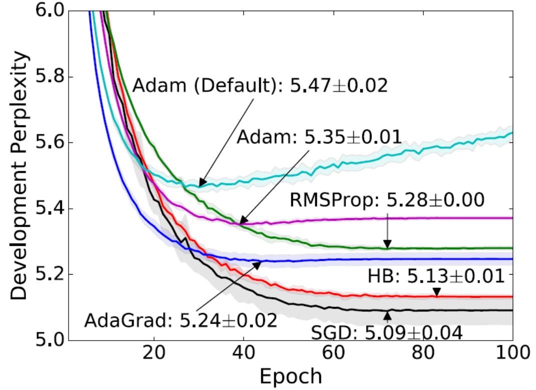


(c) Discriminative Parsing (Training Set) **(d)** Discriminative Parsing (Development Set)

SGD VS ADAM-LIKE OPTIMIZERS



(e) Generative Parsing (Training Set)



(f) Generative Parsing (Development Set)

SGD vs ADAM-like optimizers

- **Why?** In overparameterized models, there are many possible solutions that achieve zero training error.
- The optimization algorithm's design gives it an **"implicit bias"** that makes it prefer certain types of solutions over others:
  - **SGD's Implicit Bias:**
    - When started from zero, SGD has an implicit bias towards finding the solution with the **minimum Euclidean (L_2) norm**.
    - This minimum norm solution is often associated with large margins and good generalization.
  - **Adaptive Methods' Implicit Bias:**
    - Their preconditioner gives them a different bias. They don't converge to the min L_2 norm solution.
    - They sometimes tend to converge to solutions that rely on non-generalizable features.

SGD VS ADAM-LIKE OPTIMIZERS

- The concept of **"implicit bias"** provides one compelling explanation for the generalization gap observed between SGD and adaptive methods.
- **However, this is just one possible explanation.**
- The deep and complex relationship between the choice of optimizer and a model's final generalization ability is still not fully understood. It remains a somewhat **mysterious and open problem**.
- Understanding exactly why different algorithms lead to solutions with different test performances is one of the most significant and active areas of research in deep learning theory today.