Module 6 — Optimization Dynamics

Variational Quantum Algorithms

hbar.university

Gradient Descent in VQA

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.1

This is the **opening node** of Module 6 — the module on optimization dynamics. With cost landscapes characterized (Module 3) and barren plateaus understood (Module 4), the question now becomes: **how do we actually move parameters to reduce the cost?**

The simplest answer, and the historical starting point of VQA optimization, is **gradient descent**: at each step, take a small move in the direction of steepest descent. This node walks through what gradient descent looks like in the VQA context, what makes it work, what makes it fail, and how it sets the baseline that all other optimizers are compared against.

The Update Rule

Vanilla gradient descent updates parameters according to

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \, \nabla C(\boldsymbol{\theta}_k),$$

where $\eta > 0$ is the **learning rate** (or step size).

For VQA, the gradient is estimated by parameter-shift (Section 3.8), so each step requires $2P$ cost evaluations for a P-parameter ansatz. Each cost evaluation is itself a noisy quantity (Section 3.7), so what the optimizer actually computes is

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \, \widehat{\nabla C}(\boldsymbol{\theta}_k),$$

where $\widehat{\nabla C}$ is a stochastic estimator with mean equal to the true gradient and variance set by the shot budget.

This is **stochastic gradient descent (SGD)** in the strict sense — the gradient is unbiased but noisy. The convergence theory is the textbook stochastic-approximation theory (Robbins-Monro 1951), which Section 6.5 will treat in more detail.

The Three Hyperparameters

A naive gradient descent on a VQA has three knobs:

1. Learning rate η. Too small: convergence is slow. Too large: the optimizer overshoots minima and oscillates or diverges. Optimal η depends on the local Hessian (Section 3.5) — specifically, you need $\eta < 2/\lambda_{\max}$ where $\lambda_{\max}$ is the largest Hessian eigenvalue.

2. Shot budget per evaluation N. Too small: gradient noise overwhelms signal, optimizer does a random walk. Too large: each step costs more than necessary, total budget is wasted on precision that wasn't needed.

3. Stopping criterion. When to halt. Common choices: cost stops decreasing, gradient norm below a threshold, fixed budget exhausted.

These three are interrelated in non-obvious ways. For example, doubling N reduces noise by $\sqrt{2}$, which lets you take larger steps (effectively double η) with the same robustness. So "shots per step" and "step size" trade off against each other in a way that affects the overall budget allocation.

The thesis's "stochastic objective generator" framing (Module 1.8) makes this explicit: the optimizer is making decisions under noise, and N sets the noise level it sees. Tuning N is part of the optimizer's job, not just a fixed hyperparameter.

Why Gradient Descent Is The Default

Gradient descent is the default first-attempt optimizer for VQAs for several reasons:

1. It's local. You only need information at the current point — no global landscape knowledge required. This matches the VQA context, where global landscape information is unavailable.

2. It's cheap per step. $2P$ cost evaluations per step, no Hessian, no extra structure. This is the minimum information you can use and still be doing first-order optimization.

3. It has well-understood theory. For convex landscapes, convergence rate is $O(1/k)$. For L-smooth landscapes, GD with $\eta < 1/L$ is guaranteed to monotonically decrease cost (in expectation).

4. It's a clean baseline. Every fancier optimizer is "GD plus something." You can ask whether the "plus something" is worth the cost. This makes GD the reference point.

5. It works on small problems. For $n < 10$ qubits and shallow ansätze, gradient descent on a VQE problem typically converges in $O(100 - 1000)$ iterations to reasonable accuracy. It's not the fastest possible optimizer, but it's adequate.

Why Gradient Descent Often Fails On VQA

The list of GD's pathologies in VQA is long:

1. Barren plateaus (Module 4). At random initialization in a deep ansatz, the gradient is exponentially small in qubit count. GD makes essentially zero progress per step — the noise dominates the signal. No amount of patience helps; GD does a random walk.

2. Step size sensitivity. The optimal step size depends on the local Hessian, which varies wildly across parameter space. A single global η is rarely optimal everywhere. Too large: oscillations. Too small: glacial progress in the flat regions.

3. Curvature mismatch. GD uses the same step size in all directions. If the landscape has very different curvature in different directions (long, narrow valleys), GD zig-zags inefficiently. This is the "ill-conditioned Hessian" problem from classical optimization.

4. Coordinate-system dependence. GD's behavior depends on how you parameterize the ansatz. If you reparameterize (e.g., $\theta \to 2\theta$), the trajectory changes in a non-trivial way. This is "lack of reparameterization invariance" and it is the central problem the **natural gradient** (Section 6.2) addresses.

5. No escape from local minima. GD converges to whatever critical point is closest to the initialization. For non-convex VQA landscapes (which are most of them), this means GD is at the mercy of the starting point.

6. Stagnation in noise. Once the gradient drops below the shot noise floor, GD effectively stops. Even if there is a meaningful descent direction further away, GD can't get there because it has lost the signal.

These failure modes motivate every other optimizer in Module 6. Quantum natural gradient (6.2) addresses #4. Adam (6.3) addresses #2 and #3. Stochastic optimization theory (6.4, 6.5) addresses #6. Population methods (6.7-6.8) address #5.

A Cautionary Worked Example

Consider the simplest possible VQA: a single qubit, $|\psi(\theta)\rangle = R_Y(\theta)|0\rangle$, $H = Z$, $C(\theta) = \cos(\theta)$, with global minimum at $\theta = \pi$.

Setup. Start at $\theta_0 = 0.1$. Learning rate $\eta = 0.5$. $N = 100$ shots per evaluation.

Run. - $k = 0$: $C(0.1) \approx 0.995$. Gradient $\approx -\sin(0.1) = -0.0998$. Update: $\theta_1 = 0.1 + 0.5 \cdot 0.0998 = 0.150$. - $k = 1$: $C(0.15) \approx 0.989$. Gradient ≈ -0.149. Update: $\theta_2 = 0.15 + 0.075 = 0.225$. - $k = 2$: $C(0.225) \approx 0.975$. Gradient ≈ -0.223. Update: $\theta_3 = 0.225 + 0.111 = 0.336$.

The optimizer is making progress, but slowly. The gradient grows as it moves away from the starting point (because $C''(\theta) = -\cos(\theta)$ is positive in this region, i.e., the landscape is convex), so successive steps get larger.

Eventually it crosses $\theta = \pi/2$ where the gradient peaks at magnitude 1 (the inflection point), and then enters the descending side of the cosine. At $\theta \approx \pi$, the gradient is again ~ 0 and the optimizer slows down — this is approaching the minimum. With $\eta = 0.5$, the optimizer converges to within 10^{-3} of π in about 30 iterations.

Now repeat with shot noise. $N = 100$ gives gradient noise of order $\sin(\theta)/\sqrt{100} = 0.1\sin(\theta)$. When the gradient is ~ 0.1 (early or late in the trajectory), the noise is comparable to the signal and the optimizer's progress becomes erratic. At the minimum, the gradient and noise are both ~ 0.001, so the optimizer wanders around the minimum forever rather than converging exactly.

Now scale to 10 qubits with random init in a 5-layer HEA. Initial gradient is $\sim 10^{-2}$. Shot noise at $N = 100$ is ~ 0.1. Signal is below noise. **GD does a random walk and never converges.**

This progression — works in 1D, breaks at 10 qubits — is the canonical reason VQA optimizers need to do something more sophisticated than vanilla gradient descent.

What Gradient Descent Looks Like To The Stochastic Objective Generator

In the thesis's framing (Module 1.8), the optimizer interacts with a stochastic objective generator:

- The generator takes a parameter vector $\boldsymbol{\theta}$ as input.
- It returns a noisy scalar $\widehat{C}(\boldsymbol{\theta})$.
- The generator's properties (variance structure, gradient signal, plateau regions) are determined upstream by the ansatz, the Hamiltonian, the encoding, the noise model, and the shot budget.

Gradient descent's job, in this framing, is to **make decisions about where to query next** based on a sequence of past noisy evaluations. Vanilla GD is a particularly simple decision rule: query at $\boldsymbol{\theta}_k - \eta\widehat{\nabla C}$.

This framing makes clear that GD is **agnostic to the structure of the noise** and to the specific properties of the VQA. It treats the cost function as a black box. Optimizers that exploit VQA-specific structure (parameter-shift, periodicity, sinusoidal cost dependence, QFIM) can do better — and that is what the rest of Module 6 develops.

Structural Role

This is the **baseline** node of Module 6. It defines the simplest first-order optimizer, identifies its failure modes, and sets up the rest of the module as a sequence of refinements that address those failures.

It also forwards directly to Module 5 (regularization), where the same gradient descent dynamics are augmented with regularization terms that encode prior knowledge or shape the optimizer's path.

Relations to Other Concepts

- **Robbins-Monro 1951** — the foundational paper on stochastic approximation, the parent theory.
- **Cauchy 1847** — the original gradient descent paper.
- **Convex optimization** — the regime where GD has clean convergence guarantees.
- **L-smoothness** — the standard assumption needed for GD convergence proofs.
- **Module 5 regularization** — the dynamics-side framework that builds on top of GD.

Worked Example — A Two-Parameter VQE

Consider a 2-qubit VQE with ansatz $R_Y(\theta_1) \otimes R_Y(\theta_2) \cdot \mathrm{CNOT}_{0,1} \cdot |00\rangle$, Hamiltonian $H = Z_0 Z_1 + 0.5 X_0 + 0.5 X_1$.

Cost function (after a tedious but standard expansion):

$$C(\theta_1, \theta_2) = \cos(\theta_1)\cos(\theta_2) + 0.5\sin(\theta_1) + 0.5\sin(\theta_2).$$

Gradients:

$$\frac{\partial C}{\partial \theta_1} = -\sin(\theta_1)\cos(\theta_2) + 0.5\cos(\theta_1).$$
$$\frac{\partial C}{\partial \theta_2} = -\cos(\theta_1)\sin(\theta_2) + 0.5\cos(\theta_2).$$

Setting both gradients to zero: critical points at the solutions of $\tan(\theta_1)\cos(\theta_2) = 0.5$ and $\cos(\theta_1)\tan(\theta_2) = 0.5$.

Numerically, one minimum is at approximately $(\theta_1, \theta_2) \approx (-0.46, -0.46)$ with cost ≈ -1.12.

Run GD from $\boldsymbol{\theta}_0 = (0.5, 0.5)$ with $\eta = 0.1$, no noise: - Step 0: gradient $\approx (0.0, 0.0)$ — wait, let me reconsider. At $(0.5, 0.5)$, $-\sin(0.5)\cos(0.5) + 0.5\cos(0.5) \approx -0.421 + 0.439 = 0.018$. So gradient is small but nonzero. - The optimizer slowly drifts toward $(-0.46, -0.46)$ over ~ 50 steps.

Now with shot noise at $N = 1000$ per cost evaluation: gradient noise is ~ 0.03, comparable to the gradient near the start. The optimizer makes erratic progress for the first ~ 100 steps, then settles into a slow descent once the gradient is large enough.

This 2-parameter example shows the basic GD behavior in miniature: convergence works but is sensitive to noise, particularly when the gradient is small compared to the noise level.

Visualization

Pending. A 2D contour plot of $C(\theta_1, \theta_2)$ with a GD trajectory overlaid. Two trajectories shown: one noiseless (smooth curve to the minimum) and one noisy (erratic walk that eventually finds the minimum). Caption: "GD: simple, slow, noise-limited."

Exercises

1. For a 1D cost $C(\theta) = \theta^2$, show that GD with $\eta < 1$ converges, with $\eta = 1$ reaches the minimum in one step, and with $\eta > 2$ diverges. What is the equivalent statement for a 2D quadratic?
2. For shot-noisy GD on $C(\theta) = \cos(\theta)$ starting at $\theta_0 = 0.1$, estimate the shot budget needed to reach $\theta = \pi$ within 10^{-2} precision. How does this scale with N?
3. (VQA) For a 5-qubit HEA with 3 layers and a global $H = \sum_i Z_i$, estimate the typical gradient magnitude and the shot budget needed for GD to make meaningful progress. At what qubit count does GD become infeasible?

Research Extensions

- **Adaptive shot scheduling for GD** — protocols that allocate more shots when the gradient is small and fewer when it is large.
- **Line-search GD for VQA** — using a backtracking line search to choose η per step.
- **Restart strategies** — when to restart GD with a different initialization to avoid local minima.
- **GD with momentum vs vanilla GD** — quantitative comparison on standard VQA benchmarks.

Cross-links to Other Courses

- **Numerical Optimization** — gradient descent, line search, convergence theory.
- **Stochastic Approximation** — Robbins-Monro, Kiefer-Wolfowitz.
- **Module 3 (Cost Landscapes)** — the geometry GD navigates.
- **Module 4 (Barren Plateaus)** — the failure mode that breaks GD on large problems.
- **Module 5 (Regularization)** — the dynamics-side framework GD lives inside.

Quantum Natural Gradient

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.2

The vanilla gradient descent of node 6.1 has a glaring conceptual problem: **it depends on how you parameterize the ansatz**.

If you reparameterize $\theta \to 2\theta$ for some parameter, the gradient component scales by $1/2$ (chain rule), but the optimal step doesn't. Vanilla GD takes the wrong-sized step under reparameterization. For physically equivalent ansätze with different parameterizations, GD produces different trajectories.

This is unsatisfying. The **natural gradient** is the fix: rescale the gradient by a metric on parameter space, chosen so that the resulting update is invariant under reparameterization. For VQAs, that metric is the **quantum Fisher information matrix (QFIM)**, also known as the Fubini-Study metric, also discussed in node 2.9 as a property of the ansatz manifold.

This node walks through what natural gradient means, why it's the "right" geometry for VQAs, and what it costs to compute.

The Update Rule

Natural gradient descent uses the update

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta\, g^{-1}(\boldsymbol{\theta}_k)\, \nabla C(\boldsymbol{\theta}_k),$$

where $g(\boldsymbol{\theta})$ is the QFIM at point $\boldsymbol{\theta}$.

Compare to vanilla GD:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta\, I\, \nabla C(\boldsymbol{\theta}_k).$$

The only difference is replacing the identity I with g^{-1}. But this difference matters: it converts the update from a step in *parameter coordinates* to a step in *state space distance*.

What The QFIM Is

The QFIM is the Riemannian metric on the manifold of pure states, pulled back through the parameterization:

$$g_{ij}(\boldsymbol{\theta}) = 4\,\mathrm{Re}\left[\langle \partial_i\psi|\partial_j\psi\rangle - \langle \partial_i\psi|\psi\rangle\langle\psi|\partial_j\psi\rangle\right].$$

Equivalently, it is the Hessian of the **quantum infidelity** $1 - |\langle\psi(\boldsymbol{\theta})|\psi(\boldsymbol{\theta}+\boldsymbol{\delta})\rangle|^2$ at $\boldsymbol{\delta} = 0$, divided by 2.

Key properties:

- **Positive semidefinite.** $g \geq 0$.
- **Reparameterization-covariant.** If $\boldsymbol{\theta} \to \boldsymbol{\phi}(\boldsymbol{\theta})$, then $g \to J^T g J$ where J is the Jacobian. The combination $g^{-1}\nabla C$ is invariant.
- **Captures state distinguishability.** Two parameter directions with small g_{ii} produce nearly indistinguishable states; large g_{ii} means easily distinguishable.
- **Has a kernel for gauge-redundant ansätze.** If the parameterization has continuous gauge directions (Section 3.4), the QFIM has zero eigenvalues in those directions, and g^{-1} doesn't exist.

The kernel issue is technically inconvenient and is usually handled by replacing g^{-1} with the **Moore-Penrose pseudoinverse** g^{+}.

Why Natural Gradient Is "Natural"

The intuition is that **vanilla GD treats parameter space as flat**, but the parameterization of the ansatz manifold doesn't preserve flatness in any meaningful way.

For example, if θ_1 and θ_2 are two parameters, vanilla GD assumes that "moving by δ in θ_1" and "moving by δ in θ_2" produce equal-sized state-space changes. This is rarely true. θ_1 might be a parameter that strongly affects the state (a single-qubit rotation on a superposition state), while θ_2 is a parameter with weak effect (a rotation on an eigenstate).

Natural gradient corrects for this by **measuring step size in state space, not parameter space**. The QFIM is the conversion factor.

The result: natural gradient takes uniform steps in the *physically meaningful* metric — Fubini-Study distance on the manifold of states. Two trajectories that produce the same sequence of states but use different parameter labels look identical to natural gradient.

This is the same idea as Fisher's natural gradient in statistics (Amari 1998), which uses the classical Fisher information matrix to make gradient descent on probability distributions invariant under reparameterization. The QFIM is the quantum version.

How To Compute The QFIM

For a parameterized circuit, the QFIM has $P \times P$ entries. Computing one entry requires evaluating two quantum-state inner products:

$$g_{ij} = 4\,\mathrm{Re}\left[\langle\partial_i\psi|\partial_j\psi\rangle - \langle\partial_i\psi|\psi\rangle\langle\psi|\partial_j\psi\rangle\right].$$

Each inner product can be evaluated with a **Hadamard test** circuit, which takes 1 ancilla qubit and a controlled version of the ansatz. So computing the full QFIM takes $O(P^2)$ circuit evaluations, each $O(N)$ shots — comparable to computing a Hessian.

For $P = 50$ and $N = 1000$, the QFIM takes $\sim 10^6$ shots per iteration. This is **5-10x more expensive** than computing the gradient itself, which limits how often you can afford to update the QFIM in practice.

Common cost-saving approximations:

1. **Block-diagonal QFIM.** Only compute the diagonal blocks corresponding to layers or qubits, ignoring cross-block correlations. Cost drops from $O(P^2)$ to $O(P \cdot \text{block-size})$.
2. **Diagonal QFIM.** Only compute g_{ii}, no off-diagonal entries. Cost drops to $O(P)$, the same as the gradient. The resulting "natural gradient" is no longer fully invariant but captures the per-coordinate scaling.
3. **Cached QFIM.** Compute the QFIM every K iterations rather than every iteration. Useful when the QFIM doesn't change much between steps.
4. **K-FAC-style approximations.** Borrowed from classical natural gradient literature: factor the QFIM into smaller, cheaper-to-invert pieces.

In practice, block-diagonal or diagonal QFIM is what most VQA implementations use. The fully invariant natural gradient is theoretically appealing but operationally expensive.

What Natural Gradient Buys You

The empirical case for natural gradient:

1. Faster convergence near critical points. At a minimum, the QFIM rescales the gradient so that all directions are equivalent. This eliminates the "ill-conditioning" problem of vanilla GD on stretched-out minima.

2. Better behavior in the barren plateau regime. In a plateau, the QFIM eigenvalues are typically very small (the state barely changes with parameter changes). Inverting the QFIM amplifies these directions. This can, in principle, recover signal from regions where vanilla GD has none — though in practice, the noise on the QFIM estimation is *also* large, so the amplification mostly amplifies noise.

3. Reparameterization invariance. Two ansätze with different parameter labels but the same physics give identical natural-gradient trajectories. This is a clean conceptual property and useful for fair comparison of ansatz designs.

4. Better step-size selection. With the QFIM, the appropriate step size is in *state-space units* (e.g., "move 0.01 in Fubini-Study distance"), which is more interpretable than parameter-space units.

5. Smoother optimization paths. Natural gradient produces trajectories that are smoother in state space — fewer zig-zags, more direct routes to the minimum. This reduces the number of steps needed.

What Natural Gradient Doesn't Buy You

It is important to be honest about what natural gradient *doesn't* fix:

1. Barren plateaus. Natural gradient doesn't eliminate them. If the gradient is below the shot noise floor, multiplying by g^{-1} (whose entries are also exponentially small) doesn't recover signal. The ratio $\nabla C/g$ might be more useful conceptually, but estimating both numerator and denominator in the noise floor regime is hopeless.

2. Local minima. Natural gradient is still a local first-order optimizer. It finds the critical point closest to the initialization. No global escape.

3. Cost. The QFIM is expensive. For $P > 100$ parameters, computing the full QFIM each iteration is prohibitive. The savings from "fewer iterations" don't always outweigh the "more shots per iteration."

In practice, natural gradient is most useful for **medium-sized VQAs** (10-50 parameters) where the shot cost of the QFIM is tolerable and the convergence speedup is meaningful.

The Stoudenmire-Schwab Variant

A specific natural gradient variant worth knowing about: **Stoudenmire-Schwab quantum natural gradient (Stokes et al. 2020)**.

This computes the QFIM efficiently by using the **block structure** of the parameterized ansatz: each ansatz layer depends on a small number of parameters, and the QFIM block for that layer can be computed by Hadamard tests within the layer alone, using the *previous* layer's output as the input state.

For a layered ansatz with L layers and q parameters per layer, the per-layer QFIM block is $q \times q$ and costs $O(q^2)$ circuit evaluations. The total QFIM is approximately block-diagonal with L blocks, costing $O(Lq^2) = O(Pq)$ total — linear in the number of parameters times the layer width.

For typical VQAs with q small (say 5-10 parameters per layer), this brings the QFIM cost down from quadratic to nearly linear in P, making natural gradient practical for ansätze of ~ 100 parameters.

Stokes et al. (2020) showed that this block-diagonal natural gradient gives most of the convergence benefit of the full QFIM at a fraction of the cost. In modern VQA practice, "natural gradient" usually refers to this block-diagonal version unless otherwise specified.

Structural Role

This node is the **geometric** entry point of Module 6. It introduces the QFIM (formally; node 2.9 introduced it informally) and motivates the use of intrinsic geometry rather than ambient parameter coordinates.

It forwards to Section 6.3 (Adam and adaptive methods, which use diagonal-curvature-like rescalings as a poor-man's natural gradient) and Section 6.6 (optimal transport, which generalizes the natural gradient idea to even richer geometries).

Relations to Other Concepts

- **Fisher information (classical)** — Amari's natural gradient.
- **QFIM / Fubini-Study metric** — the quantum analogue.
- **Riemannian optimization** — the general framework for optimization on curved spaces.
- **Newton's method** — second-order optimization that uses the cost Hessian (different from QFIM).
- **Stokes et al. 2020** — the block-diagonal practical QFIM construction.

Worked Example — QFIM For A Single-Qubit Ansatz

Consider $|\psi(\theta_1, \theta_2)\rangle = R_Y(\theta_2)R_X(\theta_1)|0\rangle$.

Compute the partial derivatives:

$$|\partial_1\psi\rangle = -\frac{i}{2}R_Y(\theta_2)XR_X(\theta_1)|0\rangle.$$
$$|\partial_2\psi\rangle = -\frac{i}{2}YR_Y(\theta_2)R_X(\theta_1)|0\rangle.$$

Compute the inner products:

$$\langle\partial_1\psi|\partial_1\psi\rangle = \frac{1}{4}\langle 0|R_X^\dagger X^2 R_X|0\rangle = \frac{1}{4}.$$
$$\langle\partial_2\psi|\partial_2\psi\rangle = \frac{1}{4}\langle\psi(0)|Y^2|\psi(0)\rangle = \frac{1}{4}.$$
$$\langle\partial_1\psi|\partial_2\psi\rangle = -\frac{1}{4}\langle 0|R_X^\dagger X R_Y^\dagger Y R_Y R_X|0\rangle = ?$$

Working out the third quantity gives a θ_1, θ_2-dependent value.

The QFIM is

$$g(\theta_1, \theta_2) = 4\begin{pmatrix} \frac{1}{4} - |\langle\partial_1\psi|\psi\rangle|^2 & \cdots \\ \cdots & \frac{1}{4} - |\langle\partial_2\psi|\psi\rangle|^2 \end{pmatrix}.$$

For $\theta_1 = 0$, the state is $R_Y(\theta_2)|0\rangle = \cos(\theta_2/2)|0\rangle + \sin(\theta_2/2)|1\rangle$, and the QFIM evaluates to

$$g(0, \theta_2) = \begin{pmatrix} \cos^2(\theta_2/2) & 0 \\ 0 & 1 \end{pmatrix}.$$

Note: at $\theta_2 = \pi$, the (1,1) entry vanishes — the QFIM has a kernel here, and natural gradient is undefined in the θ_1 direction. The vanishing happens because at this point, the state $R_Y(\pi)|0\rangle = |1\rangle$ is an eigenstate of X (up to phase), so an infinitesimal rotation around the X-axis doesn't change the state. There is no direction in state space corresponding to motion in θ_1, so natural gradient pseudo-inverse must handle this gracefully.

This is the kind of degeneracy you see in QFIMs in practice; pseudoinverse handles it cleanly.

Visualization

Pending. A 2D parameter-space plot showing two trajectories on the same cost landscape: one from vanilla GD (zig-zag) and one from natural gradient (smoother). Annotations highlight the QFIM rescaling at each step. Caption: "Vanilla GD walks in coordinates. Natural gradient walks in state space."

Exercises

1. For $|\psi(\theta)\rangle = R_Y(\theta)|0\rangle$, compute the QFIM (which is just a scalar). Show that it equals 1 for all θ, and explain why.
2. For a 2-parameter ansatz with QFIM $g = \begin{pmatrix} 4 & 1 \\ 1 & 1 \end{pmatrix}$, compare the vanilla and natural gradient updates for a fixed gradient $\nabla C = (1, 1)^T$. Which direction does each take?
3. (VQA) Estimate the cost of a single natural gradient step (in shot count) for a 30-parameter ansatz, using the block-diagonal QFIM with 5 parameters per block. Compare to a vanilla gradient step.

Research Extensions

- **K-FAC and other approximate natural gradients** — borrowed from classical deep learning.
- **Online QFIM estimation** — updating the QFIM incrementally rather than recomputing from scratch.
- **Gauge-invariant natural gradient** — formulations that handle continuous gauge redundancies cleanly.
- **Hardware-aware QFIM** — accounting for noise in the QFIM computation itself.

Cross-links to Other Courses

- **Information Geometry** — Amari's natural gradient and Fisher information.
- **Differential Geometry** — Riemannian metrics, geodesics, parallel transport.
- **Module 2 (Ansatz as Manifold, Section 2.9)** — informal introduction to the QFIM.
- **Module 3 (Local Geometry, Section 3.5)** — gradient and Hessian alongside the QFIM.

Adam and Adaptive Methods

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.3

Vanilla gradient descent (6.1) uses one global learning rate. Natural gradient (6.2) rescales the gradient by the QFIM, which is geometrically principled but expensive. **Adaptive methods** sit in between: they rescale the gradient using *cheap, online estimates* of curvature derived from the gradient history itself, without ever computing the QFIM.

The most famous adaptive method is **Adam** (Kingma and Ba, 2014), which has become the default optimizer in classical deep learning. Variants include RMSProp, AdaGrad, AdaDelta, AMSGrad, and RAdam — all with similar structure but different details.

This node describes the Adam family, explains why they often outperform vanilla GD on VQAs, and identifies their failure modes.

The Adam Update Rule

Adam maintains two moving averages alongside the parameters:

- $\mathbf{m}_k$: exponential moving average of gradients (first moment).
- $\mathbf{v}_k$: exponential moving average of squared gradients (second moment).

At each step:

$$\begin{aligned}
\mathbf{m}_k &= \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\nabla C(\boldsymbol{\theta}_{k-1}), \\
\mathbf{v}_k &= \beta_2 \mathbf{v}_{k-1} + (1-\beta_2)[\nabla C(\boldsymbol{\theta}_{k-1})]^2, \\
\widehat{\mathbf{m}}_k &= \frac{\mathbf{m}_k}{1-\beta_1^k}, \quad \widehat{\mathbf{v}}_k = \frac{\mathbf{v}_k}{1-\beta_2^k} \quad \text{(bias correction)}, \\
\boldsymbol{\theta}_{k+1} &= \boldsymbol{\theta}_k - \eta \frac{\widehat{\mathbf{m}}_k}{\sqrt{\widehat{\mathbf{v}}_k} + \epsilon}.
\end{aligned}$$

The default hyperparameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, $\eta = 10^{-3}$.

The squared-gradient division is the key. **It rescales each component of the update by an estimate of the local gradient variance**, which is roughly an estimate of the local curvature in that direction. Components with consistently large gradients get smaller effective steps; components with consistently small gradients get larger effective steps.

Why The Squared-Gradient Rescaling Works

The intuition is: $\mathbf{v}_k$ is an exponential moving average of $|\nabla C|^2$. For a smooth landscape near a minimum, $|\nabla C|^2$ is approximately related to the local curvature: high-curvature directions have rapidly-changing gradients, so $\mathbf{v}$ is large there; low-curvature directions have slowly-changing gradients, so $\mathbf{v}$ is small.

Dividing by $\sqrt{\mathbf{v}}$ thus produces an update where steps are normalized by $\sqrt{\text{curvature}}$, which is similar to what natural gradient does — except the "curvature" estimate is *empirical*, not derived from the QFIM.

This is much cheaper than computing the QFIM (no extra circuit evaluations), and it's adaptive (the estimate updates as the optimizer moves). The cost is that the curvature estimate is *noisy* and *biased* — $\mathbf{v}$ reflects past gradients, which may not reflect current curvature, and it's contaminated by shot noise.

In classical deep learning, this rescaling is usually a net win. In VQAs, the picture is more nuanced (see "Failure Modes" below).

Why Adam Works Well For VQA

1. Per-coordinate learning rates. Different parameters in a VQA ansatz often have wildly different sensitivities. A single global learning rate either underutilizes the small-sensitivity parameters or overshoots on the large-sensitivity ones. Adam's per-coordinate rescaling handles this automatically.

2. Robustness to gradient noise. The first-moment averaging $\mathbf{m}$ acts like momentum, smoothing out noisy gradient estimates over many iterations. This is helpful when shot noise is comparable to the gradient signal.

3. Adaptive step size near minima. As the optimizer approaches a minimum, the gradient shrinks but $\mathbf{v}$ remains large (it's a moving average of past, larger gradients), so the effective step size shrinks gracefully — no manual learning-rate decay needed.

4. No QFIM. Adam approximates curvature information without ever computing the QFIM, saving the $O(P^2)$ cost.

5. Robust default hyperparameters. The defaults ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 10^{-3}$) work reasonably well across a wide range of problems. This makes Adam a "fire and forget" choice for an unfamiliar VQA instance.

For these reasons, Adam is the **most common practical optimizer** for VQAs in 2026 software (Pennylane, Qiskit, Yao). It is the default for "I just want to optimize this thing without thinking about it."

Failure Modes Of Adam On VQAs

Adam's strengths come with corresponding weaknesses:

1. Bias from noisy gradients. $\mathbf{v}$ accumulates squared *noisy* gradients, not squared true gradients. This biases the estimate of curvature upward (because variance contributes to expectation of squared values), which biases the step size *downward*. In the high-noise regime, Adam can be more conservative than necessary, taking smaller steps than vanilla GD with the same learning rate.

2. No barren plateau immunity. Like vanilla GD and natural gradient, Adam doesn't escape barren plateaus. If the gradient is below the noise floor, no rescaling can recover signal.

3. Hyperparameter sensitivity in the noisy regime. The default $\beta_2 = 0.999$ (essentially infinite memory) was chosen for classical deep learning where the gradient is computed once per minibatch on a deterministic loss. For VQAs with shot noise, the optimal β_2 is typically smaller (more weight on recent gradients), and the optimal η depends strongly on the shot budget.

4. Coordinate-system dependence. Adam is *less* coordinate-dependent than vanilla GD (the per-coordinate rescaling helps), but it is *not* fully reparameterization-invariant the way natural gradient is. Reparameterizing the ansatz changes the Adam trajectory.

5. Convergence to non-stationary points (Reddi et al. 2018). Adam was shown to fail to converge in some convex examples — the squared-gradient bias can lead the optimizer to oscillate around a non-minimum point. **AMSGrad** is a variant that fixes this by using the maximum of past $\mathbf{v}$ values rather than an exponential average.

These failure modes are usually mild in practice but worth knowing about.

Variants

RMSProp (Hinton, 2012): Adam without the first-moment averaging. Just $\theta_{k+1} = \theta_k - \eta \nabla C / \sqrt{\mathbf{v}_k}$. Older, simpler, sometimes preferable when you don't want momentum dynamics.

AdaGrad (Duchi et al., 2011): Uses the cumulative sum of squared gradients instead of an exponential average. Step sizes monotonically decrease over time. Good for problems with sparse updates; converges very slowly on dense problems.

AdaDelta (Zeiler, 2012): Like RMSProp but with adaptive learning rate. Largely superseded by Adam.

AMSGrad (Reddi et al., 2018): Fixes Adam's non-convergence by using max over past $\mathbf{v}$ values.

RAdam (Liu et al., 2020): Rectified Adam, addresses the warm-up period instability of Adam.

LAMB / LARS: Layer-wise adaptive methods designed for large-batch training in deep learning. Less relevant for VQAs.

For VQA work, **Adam** and **AMSGrad** are the most commonly used. Other variants are situational.

Tuning Adam For VQA

Learning rate η: typically 10^{-2} to 10^{-1} for VQAs (larger than the classical default of 10^{-3}), because VQA gradients are usually smaller than classical NN gradients due to the cosine-bounded nature of expectation values.

β_2: consider lowering from 0.999 to 0.99 or 0.9 for noisy VQA optimization. The lower β_2 gives more weight to recent gradients, which is helpful when the noise level is high.

Shot budget: Adam's smoothing means you can use fewer shots per evaluation than vanilla GD with the same stability — the moving average does some of the noise reduction for you. N can often be reduced by a factor of 4-10 compared to a vanilla GD baseline with the same learning rate.

Restarts: if Adam stalls, restart with the current parameters but reset $\mathbf{m}$ and $\mathbf{v}$ to zero. This sometimes breaks out of bad regions where the moving averages have accumulated stale information.

These tuning adjustments are not from the original Adam paper but are folk wisdom from VQA practice.

Adam vs Natural Gradient

A natural question: does Adam approximate natural gradient?

Sort of, but not really.

Natural gradient rescales by g^{-1}, the inverse QFIM. Adam rescales by $1/\sqrt{\mathbf{v}}$, the inverse-sqrt of moving-average squared gradients. These are *related* — the squared-gradient moving average is a (noisy, biased) estimate of the diagonal of the cost Hessian, and the cost Hessian is related to the QFIM through the cost function. But they are not the same.

Specifically: - Natural gradient uses **state-space curvature** (independent of cost function). - Adam uses **cost-space curvature estimate** (depends on cost function and gradient history).

In some regimes they agree; in others they differ significantly. Empirically, **Adam is usually faster per step (no QFIM cost) but Natural Gradient is usually fewer steps to convergence** (more accurate descent direction). The total budget winner depends on the problem.

A reasonable rule of thumb: use Adam by default; switch to natural gradient if you're spending too many iterations to converge and the per-iteration QFIM cost is affordable.

Structural Role

This node is the **practical default** of Module 6. Adam is what most VQA implementations use, and understanding its behavior — strengths, failure modes, hyperparameters — is essential for any practitioner.

It also bridges between vanilla GD (6.1) and natural gradient (6.2), occupying a middle ground in the cost-vs-quality tradeoff.

Relations to Other Concepts

- **Kingma and Ba 2014** — the original Adam paper.
- **Reddi et al. 2018** — AMSGrad and the convergence analysis.
- **Momentum methods (Polyak, Nesterov)** — Adam's first-moment averaging is essentially momentum.
- **AdaGrad / RMSProp** — predecessors that motivated Adam.
- **Natural gradient** — the geometric ideal that Adam approximates cheaply.

Worked Example — Adam Vs GD On A Two-Parameter Cost

Consider $C(\theta_1, \theta_2) = \theta_1^2 + 100\theta_2^2$ — a stretched quadratic with very different curvature in the two directions.

Vanilla GD: the optimal step size is $\eta = 1/100 = 0.01$ (limited by the worst direction). At $\boldsymbol{\theta}_0 = (1, 1)$, the gradient is $(2, 200)$. The update is $(1, 1) - 0.01 \cdot (2, 200) = (0.98, -1)$. Notice it overshot $\theta_2 = 0$ and went to -1. The next step will overshoot back. Vanilla GD oscillates in θ_2 while crawling in θ_1.

Adam (with default $\eta = 0.001$): at iteration 1, $\mathbf{m} = (0.2, 20)$ and $\mathbf{v} = (0.004, 40)$. The update is $-0.001 \cdot (0.2/\sqrt{0.004}, 20/\sqrt{40}) = -0.001 \cdot (3.16, 3.16) = (-0.003, -0.003)$. **Both components have the same step size**, despite the very different gradient magnitudes. This is the per-coordinate rescaling at work.

After many iterations, Adam approaches the minimum at roughly equal rates in both directions, while vanilla GD oscillates wildly. Adam reaches a precision of 10^{-3} in ~ 1000 iterations; vanilla GD with the same number of iterations is still oscillating.

For the analogous problem in VQA — a parameter that strongly affects cost (θ_1-like) and one that weakly affects cost (θ_2-like) — Adam handles both gracefully where vanilla GD struggles.

Visualization

Pending. A 2D contour plot of a stretched-quadratic loss with two trajectories: vanilla GD (zig-zag, slow descent in long direction) and Adam (smooth, balanced descent in both directions). Caption: "Adam handles ill-conditioning that GD cannot."

Exercises

1. For the stretched quadratic $C(\theta_1, \theta_2) = a\theta_1^2 + b\theta_2^2$, derive the optimal step size for vanilla GD and compare to Adam's effective step size at each component.
2. The bias correction in Adam (the $1 - \beta^k$ divisions) only matters in early iterations. Show that without bias correction, Adam under-estimates $\mathbf{m}$ and $\mathbf{v}$ for the first $\sim 1/(1 - \beta)$ iterations.
3. (VQA) For a 20-parameter HEA with shot budget $N = 1000$, estimate the relative iteration count for vanilla GD vs Adam to converge to a fixed cost precision. Which uses fewer total shots?

Research Extensions

- **Adam variants for shot-noisy gradients** — versions of Adam that explicitly model shot noise in the moment estimates.
- **Adam + natural gradient hybrids** — using Adam as a fast preconditioner and natural gradient as a refinement.
- **Hyperparameter auto-tuning for Adam on VQA** — meta-learning the optimal β_2 for a given problem class.
- **Catalyst-style acceleration of Adam** — combining Adam with classical acceleration techniques.

Cross-links to Other Courses

- **Deep Learning** — Adam is the workhorse optimizer of classical neural network training.
- **Stochastic Optimization** — convergence theory for noisy gradient methods.
- **Module 6 Section 6.5 (Convergence Theory)** — formal convergence analysis of Adam.
- **Module 5 (Regularization)** — Adam is often combined with regularization terms in the objective.

Stochastic Optimization

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.4

Every VQA optimizer is, by necessity, a **stochastic** optimizer. The cost function it sees is corrupted by shot noise (Section 3.7), the gradient it computes is corrupted by parameter-shift estimation noise, and on real hardware, the gates are corrupted by physical noise. The optimizer's job is not to optimize a deterministic function — it's to make good decisions under noise.

This node introduces the **stochastic optimization** framework that the rest of Module 6 lives inside. It covers the foundational theory (Robbins-Monro), the special class of zero-order methods (Kiefer-Wolfowitz, SPSA), and the practical implications for VQA optimizer design.

The thesis's "stochastic objective generator" abstraction (Module 1.8) is most explicitly cashed out here.

The Stochastic Optimization Setting

A stochastic optimization problem is:

$$\arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\xi}[f(\boldsymbol{\theta}, \xi)],$$

where ξ is a random variable representing noise, and the optimizer can only access noisy samples $f(\boldsymbol{\theta}, \xi)$ — not the true expectation.

For VQAs, $\boldsymbol{\theta}$ is the parameter vector, ξ is the randomness from quantum measurement (shot noise) plus hardware noise, and f is the cost-function-with-noise. The expected value $\mathbb{E}_{\xi}[f]$ is the noiseless cost $C(\boldsymbol{\theta})$ (assuming unbiased estimation).

The question is: **how do you find** $\arg\min C$ **when you can only see noisy estimates of** C **and its derivatives?**

This is the territory of stochastic optimization theory, which has been developed since the 1950s and is now a mature field.

Robbins-Monro: The Foundational Result

Robbins and Monro (1951) proved that a simple recursion can find roots of a function from noisy estimates of the function's value:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - a_k \widehat{\nabla C}(\boldsymbol{\theta}_k),$$

where a_k is a sequence of step sizes (the "learning rate schedule") satisfying:

1. $\sum_k a_k = \infty$ — the steps don't decrease too fast.
2. $\sum_k a_k^2 < \infty$ — the steps eventually decrease.

A canonical choice is $a_k = a/k$.

Theorem: Under these conditions, plus mild regularity on C (smoothness, bounded variance of the gradient estimator, unique minimum), the iterates $\boldsymbol{\theta}_k$ converge almost surely to the minimum of C.

Why this matters: Robbins-Monro gives the first formal guarantee that stochastic gradient descent works *despite the noise*, provided the learning rate is decreased appropriately. The decreasing learning rate is the key — it averages out noise asymptotically.

For VQA, the Robbins-Monro condition translates to: - Use a decreasing learning rate, e.g., $\eta_k = \eta_0/k$ or $\eta_0/\sqrt{k}$. - Allocate enough shots that the gradient estimate is unbiased and has finite variance. - Be patient — convergence is asymptotic, and the rate is typically $O(1/\sqrt{k})$ for noisy problems.

Convergence Rates In Stochastic Optimization

For a smooth, strongly convex C with bounded gradient variance σ^2, the convergence rate of stochastic GD is

$$\mathbb{E}[C(\boldsymbol{\theta}_k) - C^*] \leq \frac{\sigma^2}{\mu k} + O\left(\frac{1}{k^2}\right),$$

where μ is the strong-convexity parameter.

For VQAs (which are typically *not* convex), the right notion of convergence is to a critical point, not the global minimum:

$$\mathbb{E}[\|\nabla C(\boldsymbol{\theta}_k)\|^2] \leq \frac{C}{\sqrt{k}}.$$

The $1/\sqrt{k}$ rate is the canonical "stochastic optimization rate" — slower than the $1/k$ of deterministic GD on smooth-convex problems, because the noise contributes a floor on accuracy.

The implication: if you want to halve the gradient norm in stochastic optimization, you need *4x* more iterations, not just 2x. This is why VQA optimizers often need many hundreds or thousands of iterations to converge — the noise scaling is unfavorable.

Mini-Batch SGD And Its Quantum Analogue

In classical deep learning, **mini-batch SGD** estimates the gradient from a subset (mini-batch) of training data, which gives a noisy but unbiased gradient estimate. The mini-batch size trades off computation per step against gradient noise.

The VQA analogue is **shot allocation**: the number of shots per cost evaluation determines the gradient noise level. More shots = lower noise = larger possible step size = fewer iterations. Fewer shots = higher noise = smaller possible step size = more iterations.

The **total shot budget** is the right thing to optimize. Given a fixed total of S shots, you can spend them as: - Many cheap iterations: S/(cost per iter) iterations, each with high noise per iter. - Few expensive iterations: S/(cost per iter) iterations, each with low noise.

The optimal allocation depends on the noise structure. As a rule of thumb, **noisy gradients with many cheap iterations** wins for problems where the optimizer can recover from noise (e.g., convex problems or near a minimum), while **clean gradients with few expensive iterations** wins for problems where noise causes instability (e.g., near a saddle point or in a barren plateau).

This is one of the genuine VQA-specific design problems: choosing the shot budget per evaluation, which has no clean classical analogue.

Zero-Order Stochastic Optimization

What if you don't have gradients at all — only noisy cost evaluations?

This is the **zero-order** setting, and it has its own theory.

Kiefer-Wolfowitz (1952): the original zero-order analogue of Robbins-Monro. Estimates each gradient component by finite differences:

$$\widehat{\partial_i C} = \frac{C(\boldsymbol{\theta} + h\mathbf{e}_i) - C(\boldsymbol{\theta} - h\mathbf{e}_i)}{2h}.$$

Each component requires 2 cost evaluations, so the full gradient costs $2P$ — same as parameter-shift, but with finite-difference error.

SPSA (Spall, 1992): Simultaneous Perturbation Stochastic Approximation. Estimates the *full gradient* from just **2 cost evaluations**, regardless of P. The trick: perturb all parameters simultaneously by a random sign vector $\boldsymbol{\Delta}$, and use the difference $C(\boldsymbol{\theta} + h\boldsymbol{\Delta}) - C(\boldsymbol{\theta} - h\boldsymbol{\Delta})$ to estimate the gradient projected onto the random direction.

The SPSA gradient estimate is

$$\widehat{\partial_i C} = \frac{C(\boldsymbol{\theta} + h\boldsymbol{\Delta}) - C(\boldsymbol{\theta} - h\boldsymbol{\Delta})}{2h\Delta_i}.$$

This is unbiased (in expectation over $\boldsymbol{\Delta}$) and gives a noisy but useful descent direction.

Convergence: SPSA converges at the same asymptotic rate as full-gradient stochastic descent, but with $P/2$ **times fewer cost evaluations per iteration**. For large P, this is a huge saving.

The cost is that the per-iteration gradient is much noisier, so SPSA needs more iterations to reach the same accuracy. The **total cost** can still be lower because the per-iteration cost drops from $O(P)$ to $O(1)$.

For VQAs with $P \sim 100$, SPSA can be 10-50x faster than vanilla parameter-shift gradient descent, especially in the early iterations where high gradient precision is unnecessary.

SPSA In VQA Practice

SPSA was popularized in the VQA community by **Kandala et al. (2017)** ("Hardware-efficient variational quantum eigensolver for small molecules"), which used it to optimize VQE on real superconducting hardware. Since then it has been a default choice for hardware experiments where shot budgets are tight.

Practical SPSA workflow:

1. Choose perturbation size h (typically $h = c/k^{0.101}$, the canonical Spall schedule).
2. Choose learning rate $a_k = a/(k + A)^{0.602}$, with A a small constant.
3. At each iteration, draw $\boldsymbol{\Delta}$ as a vector of ± 1 (Bernoulli rademacher).
4. Evaluate $C(\boldsymbol{\theta} + h\boldsymbol{\Delta})$ and $C(\boldsymbol{\theta} - h\boldsymbol{\Delta})$.
5. Compute the SPSA gradient estimate.
6. Update parameters with $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - a_k \widehat{\nabla C}$.
7. Repeat.

The schedules $h \sim 1/k^{0.101}$ and $a \sim 1/k^{0.602}$ come from Spall's analysis as the asymptotically optimal choices. In practice they are tuned per problem.

When To Use SPSA Vs Parameter-Shift

Use SPSA when:

- You have many parameters ($P > 50$) and the per-iteration cost of parameter-shift is prohibitive.
- The problem is well-conditioned enough that noisy gradients are tolerable.
- You're early in the optimization and want to make fast progress before fine-tuning.
- You're in a regime where parameter-shift's precision is wasted (e.g., early iterations where you just need a descent direction, not an accurate one).

Use parameter-shift when:

- You have few parameters ($P < 30$) and per-iteration cost is manageable.
- You're late in the optimization and need accurate gradients for fine-tuning.
- The problem is ill-conditioned and noisy gradients lead to instability.
- You need exact gradients for theoretical reasons (e.g., natural gradient with QFIM).

A common workflow: use SPSA for the first 100-1000 iterations to get into a good basin, then switch to parameter-shift for fine-tuning.

Other Zero-Order Methods

Coordinate descent: update one parameter at a time using its 1D gradient. Good for problems with structure.

Random search: sample random parameter perturbations, accept those that decrease the cost. Slow but very robust to noise.

Bayesian optimization: model the cost function as a Gaussian process, query points to maximize information gain. Expensive per iteration but data-efficient.

Trust-region methods: maintain a region within which the local approximation is trusted, only step within the region. Good for noisy problems.

These all have niches in VQA optimization, but parameter-shift gradient descent (with some adaptive variant) and SPSA are the dominant choices.

Structural Role

This node is the **theoretical foundation** for everything stochastic in Module 6. It justifies why noisy optimization is possible, gives the convergence rates that subsequent sections use, and introduces SPSA as the principal alternative to parameter-shift gradients.

It also connects directly to the thesis's "stochastic objective generator" abstraction (Module 1.8), which is the conceptual frame for treating the VQA optimizer as a stochastic-decision-maker rather than as a smooth-function-minimizer.

Relations to Other Concepts

- **Robbins-Monro 1951** — the foundational stochastic approximation theorem.
- **Kiefer-Wolfowitz 1952** — the zero-order analogue.
- **Spall 1992** — SPSA, the efficient zero-order method.
- **Stochastic gradient descent (SGD)** — the classical deep learning workhorse.
- **Mini-batch optimization** — the classical analogue of shot allocation.

Worked Example — SPSA On A 4-Parameter VQE

Consider a 4-parameter ansatz with cost $C(\boldsymbol{\theta}) = \sum_{i=1}^{4}(\theta_i - i)^2$, so the minimum is at $\boldsymbol{\theta}^* = (1, 2, 3, 4)$ with $C^* = 0$.

Parameter-shift gradient: at any point, the gradient is exactly $2(\boldsymbol{\theta} - (1, 2, 3, 4))$, and computing it requires 8 cost evaluations.

SPSA gradient at $\boldsymbol{\theta}_0 = (0, 0, 0, 0)$:

Draw $\boldsymbol{\Delta} = (+1, +1, -1, +1)$. Evaluate

$$C(\boldsymbol{\theta}_0 + h\boldsymbol{\Delta}) - C(\boldsymbol{\theta}_0 - h\boldsymbol{\Delta}) = ?$$

With $h = 0.1$: - $C(0.1, 0.1, -0.1, 0.1) = 0.81 + 3.61 + 9.61 + 15.21 = 29.24$. - $C(-0.1, -0.1, 0.1, -0.1) = 1.21 + 4.41 + 8.41 + 16.81 = 30.84$. - Difference: $29.24 - 30.84 = -1.60$.

SPSA gradient estimate:

$$\widehat{\partial_i C} = \frac{-1.60}{2 \cdot 0.1 \cdot \Delta_i} = -8/\Delta_i.$$

- For $i = 1$: $-8/(+1) = -8$. True gradient: $2(0 - 1) = -2$. SPSA estimate is biased high in magnitude but correct in sign. - For $i = 2$: $-8/(+1) = -8$. True: -4. - For $i = 3$: $-8/(-1) = +8$. True: -6. **Wrong sign!** - For $i = 4$: $-8/(+1) = -8$. True: -8. Correct by luck.

The single-iteration SPSA estimate is noisy and can be wrong on individual components. But **averaged over many iterations** (each with a fresh $\boldsymbol{\Delta}$), the estimate converges to the true gradient. In the long run, SPSA produces a valid descent direction with much less per-iteration cost than parameter-shift.

For this 4-parameter problem, SPSA converges in roughly the same number of iterations as parameter-shift, but each iteration costs 2 cost evaluations instead of 8 — a 4x saving. For a 100-parameter problem, the saving is 100x.

Visualization

Pending. Two trajectories on a 2D cost landscape: parameter-shift gradient descent (smooth, slow) and SPSA (jagged, fast). The SPSA trajectory is noisier per step but reaches the minimum in fewer total cost evaluations. Caption: "SPSA: noisy but cheap."

Exercises

1. Verify Robbins-Monro conditions for the schedule $a_k = 1/k$. Verify that they fail for $a_k = 1/k^2$ (sums to a finite value).
2. For the same 4-parameter cost function above, run SPSA for 100 iterations (analytically or in code) and observe how fast the parameters converge to the minimum.
3. (VQA) For a 50-parameter HEA with shot budget $N = 1000$ per evaluation, compare the total shot cost of (a) 1000 parameter-shift iterations and (b) 10000 SPSA iterations. Which uses fewer total shots? Which is likely to give a better final cost?

Research Extensions

- **Variance-reduced SPSA** — combining SPSA with control variates to reduce per-iteration noise.
- **Adaptive perturbation sizes** — choosing h_k based on the local landscape.
- **SPSA + Adam hybrids** — using SPSA gradients in an Adam-like update rule.
- **Quantum-aware SPSA** — exploiting the periodicity and sinusoidal structure of VQA cost functions to improve SPSA bias.

Cross-links to Other Courses

- **Stochastic Approximation Theory** — the broader field.
- **Reinforcement Learning** — uses similar zero-order techniques (REINFORCE, evolution strategies).
- **Module 3 (Shot Noise)** — the noise structure SPSA navigates.
- **Module 6 Section 6.1 (Gradient Descent in VQA)** — the parameter-shift baseline SPSA competes with.

Convergence Theory

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.5

This node is the **theoretical** node of Module 6. It collects the convergence guarantees for VQA optimizers — what we can prove about how fast they converge, under what conditions, and to what.

The honest answer is: **less than we'd like.** Most classical convergence theory assumes convexity or strong-convexity, which VQA cost landscapes do not satisfy. Most stochastic convergence theory assumes bounded-variance noise, which is violated by hardware noise (which can be biased) and is borderline for shot noise. The result is that VQA optimizer behavior is mostly understood empirically, with theory providing a (loose) skeleton.

That said, some theoretical results do apply, and they are worth knowing because they tell you what's *possible* and what's *not* in principle.

The Convergence Hierarchy

VQA convergence theory roughly stratifies into the following hierarchy, from strongest to weakest:

1. **Convex strongly-convex theory:** $O(1/k)$ or $O(\rho^k)$ (linear). Requires convex C. **Does not apply to VQAs in general.**
2. **Smooth non-convex theory:** $O(1/\sqrt{k})$ for $\mathbb{E}[\|\nabla C\|^2]$ under SGD. Requires L-smooth C and bounded gradient variance. **Applies to VQAs in shallow regimes.**
3. **PL-condition theory:** $O(\rho^k)$ under the Polyak-Łojasiewicz condition (a weakened convexity replacement). **Sometimes applies to VQAs near minima.**
4. **No-guarantee theory:** convergence is empirical, no formal rates. **The default assumption for VQAs in practice.**

The challenge of VQA convergence theory is figuring out which level of the hierarchy applies to which VQA instance, and what the relevant constants are.

L-Smoothness And Why It Holds For VQAs

A cost function is L**-smooth** if its gradient is L-Lipschitz:

$$\|\nabla C(\boldsymbol{\theta}_1) - \nabla C(\boldsymbol{\theta}_2)\| \leq L\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|.$$

For VQAs, this is **always satisfied** with a calculable L, because the cost is built from cosines and sines of parameters, which have bounded second derivatives.

Specifically: for an ansatz built from Pauli rotations, each parameter contributes at most $\|H\|$ to the Lipschitz constant of the gradient (this is a consequence of the parameter-shift identity and the boundedness of H). For a P-parameter ansatz, $L \leq \|H\| \cdot P$ at most, often much less.

L-smoothness is what enables the standard SGD convergence rate $O(1/\sqrt{k})$ for non-convex objectives. This is the **only** clean convergence rate that applies to VQAs in general — it gives you a guarantee that, asymptotically, you're heading toward a critical point.

The catch: $O(1/\sqrt{k})$ for the *gradient norm*, not the cost. To convert "small gradient" into "small cost minus optimum" requires additional structure (convexity or PL).

The Polyak-Łojasiewicz (PL) Condition

The PL condition is a weakened form of convexity that often holds locally:

$$\frac{1}{2}\|\nabla C(\boldsymbol{\theta})\|^2 \geq \mu(C(\boldsymbol{\theta}) - C^*),$$

for some $\mu > 0$ and the global minimum C^*.

This says: the gradient is at least *some* fraction of the cost gap. Equivalently, you can't have a small gradient with a large cost gap.

Under PL: - SGD converges at the *deterministic* rate, not the slower stochastic rate (you can think of PL as saying "the noise floor doesn't matter as long as you're far from the minimum"). - The rate is $O(\rho^k)$ (linear) for the cost, not just for the gradient.

Does PL hold for VQAs? - Globally: very rarely. - Locally near a minimum: often, with μ related to the local Hessian eigenvalues. - In the barren plateau regime: definitely not (small gradient, large cost gap).

So PL is a **regional** property that you can sometimes invoke for late-stage convergence proofs.

SGD Convergence Under L-Smooth Non-Convex

The cleanest result that applies to VQAs:

Theorem (smooth non-convex SGD): For L-smooth C with bounded gradient variance σ^2, the iterates of stochastic GD with step size $\eta = c/\sqrt{k}$ satisfy

$$\frac{1}{K}\sum_{k=1}^{K}\mathbb{E}[\|\nabla C(\boldsymbol{\theta}_k)\|^2] \leq \frac{C(\boldsymbol{\theta}_0) - C^*}{c\sqrt{K}} + \frac{cL\sigma^2}{\sqrt{K}}.$$

In words: the *average squared gradient norm* over the trajectory is $O(1/\sqrt{K})$.

Implications:

- Average gradient eventually decreases — we are converging toward a critical point.
- The rate $O(1/\sqrt{K})$ is slow but nontrivial.
- Doubling K only reduces the bound by $\sqrt{2}$, not 2. So 4x more iterations halves the bound.
- The constant depends on σ^2 — the *gradient variance* — which for VQAs is set by shot noise plus barren plateau effects.

For shallow VQAs without barren plateau, this gives a useful (if loose) guarantee. For deep VQAs in the plateau, σ^2 explodes and the bound is vacuous.

Convergence Rates For Different Optimizers

A summary table for the optimizers covered in Module 6:

Optimizer	Rate (cost)	Rate (grad norm)	Cost per iteration	Where it applies
Vanilla GD (deterministic)	$O(1/k)$	$O(1/k)$	$2P$	Smooth convex
Vanilla SGD	$O(1/\sqrt{k})$	$O(1/\sqrt{k})$	$2P$	Smooth non-convex

Optimizer	Rate (cost)	Rate (grad norm)	Cost per iteration	Where it applies
Natural Gradient	$O(1/k)$ (better constants)	$O(1/k)$	$2P + O(P^2)$	Smooth convex
Adam	$O(1/\sqrt{k})$	$O(1/\sqrt{k})$	$2P$	Smooth non-convex
SPSA	$O(1/\sqrt{k})$	$O(1/\sqrt{k})$	2	Smooth non-convex
Population (CMA-ES)	Empirically $O(1/k)$	N/A (zero-order)	O(pop size)	Multi-modal

The **rates** are roughly the same across optimizers (for non-convex problems, everyone gets $O(1/\sqrt{k})$ at best). The differences are in:

1. **Constants** (Natural Gradient and Adam have better constants in practice).
2. **Per-iteration cost** (SPSA has $O(1)$, Natural Gradient has $O(P^2)$).
3. **Robustness regime** (Adam handles ill-conditioning, SPSA handles many parameters cheaply).

The **total cost** to reach a fixed accuracy is a product of (rate) and (per-iteration cost), and the best optimizer depends on which factor dominates.

What The Theory Doesn't Cover

Things that VQA convergence theory does *not* address well:

1. Barren plateaus. No clean convergence rate when the gradient is below the noise floor. The theorems either give vacuous bounds or assume away the plateau.

2. Non-stationary noise. Hardware noise drifts over hours/days, but stochastic-optimization theory assumes i.i.d. noise. Drift breaks the analysis.

3. Biased gradients. Hardware noise can bias the gradient estimator (the noisy expected gradient differs from the true gradient). Standard theory assumes unbiased estimators.

4. Discrete decisions. Many VQA optimizers make discrete choices — which Pauli grouping to measure, which parameter to update next, etc. — that aren't covered by smooth-optimization theory.

5. Population dynamics. Population-based optimizers (HOPSO, CMA-ES) have their own convergence theories but they don't fit the SGD framework.

For these reasons, VQA optimizer practice usually relies on **empirical** convergence behavior with theory as a sanity check, not the other way around.

A Useful Theoretical Frame: Implicit Regularization

A more recent theoretical perspective: **the choice of optimizer implicitly regularizes the solution.**

In classical deep learning, it has been shown that SGD has an implicit bias toward "flat minima" (minima with small Hessian eigenvalues), which generalize better. This implicit bias is partly responsible for why SGD outperforms full-batch GD on overparameterized neural networks, even though both should converge to the same minima.

The same idea applies to VQAs: different optimizers find different minima, even when starting from the same initialization. Adam tends to find flatter minima than vanilla GD; natural gradient finds minima that are flat in the Fubini-Study metric; SPSA finds minima robust to per-coordinate perturbations.

This is the bridge to **regularization** (Module 5): the optimizer is not just a tool for finding the minimum — it's a mechanism for *selecting* among the minima a given cost landscape contains. This is the thesis's "dynamics-side regularization" view in another guise.

Stochastic Convergence Theorems For VQA-Specific Settings

A few results worth knowing that are tailored to VQA:

Sweke et al. (2020): "Stochastic gradient descent for hybrid quantum-classical optimization." Showed that even with single-shot gradient estimates ($N = 1$ per parameter-shift evaluation), SGD on VQAs converges in the smooth non-convex sense, provided the learning rate decays appropriately. This is theoretically reassuring — you don't need many shots per estimate, you can use single shots and rely on the optimizer to average.

Harrow and Napp (2021): "Low-depth gradient measurements can improve convergence in variational hybrid quantum-classical algorithms." Showed that more efficient gradient measurements (jointly measuring multiple gradient components) can reduce variance and accelerate convergence.

Skolik et al. (2021): "Layerwise learning for quantum neural networks." Provided the theoretical justification for layerwise training as a way to avoid the barren plateau regime, with explicit convergence guarantees in the layerwise setting.

These are examples of theory tailored to VQA structure rather than borrowed wholesale from classical optimization.

Structural Role

This node is the **theoretical scaffolding** of Module 6. It provides the convergence rate framework that subsequent sections use, identifies the regimes where theory does and doesn't apply, and connects to the implicit-regularization view that bridges to Module 5.

Relations to Other Concepts

- **Robbins-Monro 1951** — foundational convergence theorem for stochastic approximation.
- **L-smoothness / Lipschitz gradients** — the standard structural assumption.
- **PL condition** — weakened convexity sufficient for fast rates.
- **Implicit regularization (deep learning)** — the optimizer-as-regularizer view.
- **Sweke et al. 2020** — single-shot SGD convergence for VQA.

Worked Example — Convergence Rate For A Smooth Convex VQE Toy

Consider $C(\theta) = (1 - \cos(\theta))^2/4$, a smooth convex cost on a single parameter with minimum at $\theta = 0$ and $C^* = 0$.

Compute the smoothness constant. $C'(\theta) = (1 - \cos\theta)\sin\theta/2$. $|C''(\theta)| \leq 1$ (verifiable by differentiating). So $L = 1$.

Vanilla GD with $\eta = 0.5$: the convergence rate for smooth convex is $O(1/k)$, so $C(\theta_k) - C^* \leq C(\theta_0)/(1+k\eta)$.

Starting from $\theta_0 = \pi/2$: $C(\pi/2) = 1/4$. Predicted bound at $k = 100$: $(1/4)/51 \approx 0.005$. The actual error after 100 steps is similar (the bound is loose but in the right ballpark).

SGD with shot noise $\sigma = 0.05$: the bound becomes $O(\sigma^2/\sqrt{k})$ instead of $O(1/k)$, because the noise floor dominates after enough iterations. At $k = 100$, the bound is $0.05^2/10 = 0.00025$ — actually *better* than the deterministic bound! This is because the $O(1/k)$ deterministic bound is loose, and the SGD bound only "pays" for the noise asymptotically.

In practice, deterministic GD reaches 10^{-4} precision in ~100 iterations. SGD with the same step size and $\sigma = 0.05$ reaches 10^{-3} precision in ~100 iterations and stagnates near the noise floor. To reach 10^{-4}, you'd need either much more shots (lower σ) or many more iterations.

This is the canonical "rate vs noise floor" tradeoff that shapes all stochastic optimization.

Visualization

Pending. A log-log plot of $C(\boldsymbol{\theta}_k) - C^*$ vs iteration k for several optimizers (deterministic GD, stochastic GD, Adam, SPSA, natural gradient). Each shows a different rate and a different noise floor. Caption: "Rates differ; noise floors differ; total budget differs."

Exercises

1. Verify that $C(\theta) = \cos(\theta)$ is L-smooth with $L = 1$. Compute the convergence rate of GD with $\eta = 1$ starting at $\theta_0 = 0.1$, targeting the maximum at $\theta = 0$. (Note: this is a maximum, not a minimum — but the same theorem applies.)
2. The PL condition is $\frac{1}{2}\|\nabla C\|^2 \geq \mu(C - C^*)$. For $C(\theta) = \cos(\theta) - \cos(\theta^*)$ near $\theta = \theta^*$, what is μ?
3. (VQA) For a 30-parameter HEA at a fixed shot budget of 10^6 total shots, estimate which optimizer (vanilla GD, Adam, SPSA, natural gradient) will give the lowest final cost. State your assumptions.

Research Extensions

- **VQA-specific PL conditions** — finding regions of VQA cost landscapes where PL holds and quantifying the constant.
- **Convergence under hardware noise drift** — extending stochastic optimization theory to non-stationary noise.
- **Convergence under biased gradients** — VQA with biased estimators.
- **Implicit regularization theory for VQA optimizers** — formalizing which optimizers prefer which kinds of minima.

Cross-links to Other Courses

- **Convex Optimization (Boyd-Vandenberghe)** — foundational convergence results.
- **Stochastic Approximation (Kushner-Yin)** — asymptotic theory of stochastic recursions.
- **Module 4 (Barren Plateaus)** — the regime where convergence theory fails to give useful bounds.
- **Module 5 (Regularization)** — the implicit regularization perspective on optimizer choice.

Optimal Transport and Optimization

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.6

This is the most **abstract** node in Module 6, and one of the most thesis-aligned.

The standard view of optimization is: pick a starting point in parameter space, follow a descent direction, end up at a local minimum. The **optimal transport** view is different: the optimizer is moving a *probability distribution* over parameter space, not a single point. The "trajectory" is a path of distributions, evolving according to a flow that decreases the expected cost.

This view is more general than the point-based view. It makes population-based optimizers (Section 6.7) look like a natural special case rather than an exotic alternative. It connects to the **gradient flow** view of differential equations, the **Wasserstein metric** on distributions, and the thesis's regularization framework.

This node introduces the optimal transport view of optimization, the technical machinery (Wasserstein gradient flow, JKO scheme), and explains why this perspective matters for VQA.

From Points To Distributions

In standard optimization, the optimizer's state is a single point $\boldsymbol{\theta}_k \in \mathbb{R}^P$.

In the distributional view, the optimizer's state is a probability distribution μ_k over $\mathbb{R}^P$. The state evolves over time according to a rule that depends on the cost function C.

A point-based optimizer is a special case: $\mu_k = \delta_{\boldsymbol{\theta}_k}$, a delta function at the current parameter. The distributional view collapses to the point view when the distribution has no spread.

A population-based optimizer is also a special case: $\mu_k = \frac{1}{N}\sum_{i=1}^{N} \delta_{\boldsymbol{\theta}_k^{(i)}}$, an empirical distribution from N particles. The distributional view captures the population's collective behavior.

A Bayesian optimizer is yet another case: μ_k is the posterior over parameters given the observed costs.

The distributional view unifies these.

Wasserstein Gradient Flow

The principal mathematical tool for distributional optimization is **Wasserstein gradient flow**.

The Wasserstein-2 distance between two distributions μ and ν is

$$W_2(\mu, \nu) = \left(\inf_{\pi \in \Pi(\mu,\nu)} \int \|x - y\|^2 \, d\pi(x, y) \right)^{1/2},$$

where $\Pi(\mu, \nu)$ is the set of joint distributions with marginals μ and ν. Intuitively, W_2 measures the "cost of transporting mass" from μ to ν, where the cost is squared Euclidean distance.

The space of distributions equipped with W_2 is a *metric space* (in fact, a Riemannian-like manifold called Wasserstein space).

A **Wasserstein gradient flow** is a curve μ_t in Wasserstein space that decreases a functional $\mathcal{F}[\mu]$ at each instant. The defining equation is

$$\partial_t \mu_t = -\nabla_{W_2} \mathcal{F}[\mu_t].$$

For $\mathcal{F}[\mu] = \int C(\boldsymbol{\theta})\, d\mu(\boldsymbol{\theta})$ (the expected cost under μ), the Wasserstein gradient flow is

$$\partial_t \mu_t = \nabla \cdot (\mu_t \nabla C),$$

which is a transport equation: each "particle" of mass moves with velocity $-\nabla C$, and the distribution as a whole flows downhill.

This is the continuous-time, distributional analogue of gradient descent.

Discretization: The JKO Scheme

To turn the continuous flow into an algorithm, you discretize in time. The canonical discretization is the **Jordan-Kinderlehrer-Otto (JKO) scheme**:

$$\mu_{k+1} = \arg\min_{\nu} \left[\mathcal{F}[\nu] + \frac{1}{2h} W_2^2(\nu, \mu_k) \right].$$

In words: move from μ_k to a new distribution ν that decreases $\mathcal{F}$ but doesn't move too far in Wasserstein distance.

This is a variational formulation: each step solves a small optimization problem (minimize cost plus transport penalty). The transport penalty acts as a regularizer that limits how much the distribution can change in one step — analogous to the trust-region constraint in classical optimization.

In the limit $h \to 0$, the JKO sequence converges to the continuous Wasserstein gradient flow.

For VQA: the JKO scheme can be implemented by an optimizer that maintains an empirical distribution (a population of points), updates it via cost-decreasing moves, and ensures the distribution doesn't change too much per step. Population methods like CMA-ES and particle swarm can be viewed as approximate JKO schemes.

Why This Matters For VQA

The optimal transport view brings several benefits to VQA optimization:

1. Natural framework for population methods. Population optimizers (HOPSO, CMA-ES, genetic algorithms) are point-based at the algorithm level but distributional at the conceptual level. The Wasserstein gradient flow gives them a clean theoretical foundation.

2. Better handling of multi-modal landscapes. A distributional optimizer can have mass at multiple local minima simultaneously, and the relative weights can shift over time as the optimizer learns which minimum is best. A point-based optimizer is committed to one location.

3. Natural connection to regularization. Many regularization terms (entropy regularization, trust regions, KL penalties) are naturally distributional. Adding them to the JKO objective gives a clean way to design **regularized population optimizers**, which is one of the threads of the thesis.

4. Plateau-aware exploration. A distributional optimizer with high entropy (broad distribution) explores plateau regions more efficiently than a point optimizer, because it samples multiple parameter values simultaneously and can discover gorges or non-plateau regions without making sequential decisions.

5. Connection to physics. Wasserstein gradient flow is the natural continuous-time limit of Brownian motion in a potential, which is a model from statistical physics. This gives a physical intuition for the optimizer dynamics — the optimizer is a "particle" moving in a "potential landscape" with "thermal noise."

Entropy Regularization And Exploration

A particularly important variant: **entropy-regularized JKO**.

Modify the JKO objective to include an entropy term:

$$\mu_{k+1} = \arg\min_{\nu} \left[\int C \, d\nu - \tau H[\nu] + \frac{1}{2h} W_2^2(\nu, \mu_k) \right],$$

where $H[\nu] = -\int \nu \log \nu \, d\nu$ is the differential entropy and $\tau > 0$ is a temperature parameter.

The continuous-time limit is the **Fokker-Planck equation**:

$$\partial_t \mu_t = \nabla \cdot (\mu_t \nabla C) + \tau \Delta \mu_t.$$

This is the equation of a Brownian particle in the potential C at temperature τ. The optimizer **explores the landscape** with thermal-like fluctuations.

Properties:

- High τ: broad distribution, lots of exploration, slow descent.
- Low τ: narrow distribution, fast descent, prone to local minima.
- $\tau \to 0$: recovers the deterministic Wasserstein gradient flow.

Annealing τ over time (start high, decrease) is analogous to **simulated annealing**, and gives a principled exploration-exploitation schedule.

For VQAs, entropy-regularized population optimizers can escape barren plateaus by maintaining enough diversity (high entropy) to keep some particles outside the plateau region, where they continue to make progress and pull the rest of the population along.

Sinkhorn And Practical Wasserstein

In practice, computing Wasserstein distances exactly is expensive. The standard practical tool is **Sinkhorn divergence** (Cuturi 2013), which is an entropy-regularized approximation that can be computed in $O(n^2)$ for n particles via fixed-point iteration.

Sinkhorn-based Wasserstein gradient flows have become a workhorse in machine learning (generative models, domain adaptation, etc.) and are starting to appear in VQA as well.

For population-based VQA optimizers, Sinkhorn-based updates allow you to maintain the distributional view computationally.

The Thesis Connection

The thesis argues that **regularization should be viewed as dynamical modification rather than objective modification**. The Wasserstein gradient flow framework makes this explicit:

- **Objective-side regularization:** modify $\mathcal{F}[\mu] = \int C \, d\mu$ by adding a penalty term, e.g., $\mathcal{F}[\mu] = \int C \, d\mu + \lambda R[\mu]$.
- **Dynamics-side regularization:** modify the *flow equation* directly, e.g., add diffusion: $\partial_t \mu = \nabla \cdot (\mu \nabla C) + \tau \Delta \mu$.

These are *not* the same. Adding entropy penalty to the objective is not the same as adding diffusion to the flow — they have different stationary distributions and different transient behavior. The thesis dwells on this distinction at length, and Module 6 (this node specifically) is where it gets the cleanest statement.

The optimal transport view is **the natural language** in which to make this distinction precise.

Limitations And Practical Considerations

The Wasserstein view is conceptually beautiful but has practical limitations:

1. Cost. Maintaining a distribution requires more state (a population of particles, or a density estimate) than maintaining a single point. Each iteration is more expensive.

2. Convergence rates. Wasserstein gradient flow has its own convergence theory, but the rates are typically slower than point-based optimizers in the deterministic regime. The benefit shows up in non-convex / multi-modal / noisy settings, not in clean convex problems.

3. Discretization artifacts. JKO discretization introduces its own errors, and the Sinkhorn approximation introduces more. The distributional view is mathematically clean but algorithmically messier than point-based methods.

4. Limited adoption. Most VQA software packages don't have first-class support for distributional optimizers. You'd typically implement them by hand or use a population optimizer (Section 6.7-6.8) as the practical proxy.

For these reasons, optimal-transport-based VQA optimization is more of a theoretical viewpoint than a practical tool in 2026. But it's an important viewpoint, because it grounds population methods and dynamics-side regularization in a coherent framework.

Structural Role

This node is the **distributional viewpoint** of Module 6. It abstracts the point-based optimizers (6.1-6.5) into a distributional framework, motivates population methods (6.7-6.8) as natural special cases, and provides the cleanest setting for the thesis's distinction between objective-side and dynamics-side regularization.

Relations to Other Concepts

- **Wasserstein metric** — the distance on the space of distributions.
- **JKO scheme (Jordan-Kinderlehrer-Otto 1998)** — the variational discretization.
- **Fokker-Planck equation** — the entropy-regularized continuous flow.
- **Sinkhorn algorithm (Cuturi 2013)** — practical Wasserstein computation.
- **Simulated annealing** — the analogue of entropy-regularized flow with annealed temperature.

Worked Example — Wasserstein Gradient Flow For A 1D Cost

Consider $C(\theta) = \cos(\theta)$ on $[-\pi, \pi]$, with global minimum at $\theta = \pi$ (or equivalently $-\pi$).

A point optimizer starting at $\theta_0 = 0.5$ follows GD: $\theta_{k+1} = \theta_k - \eta \cdot (-\sin(\theta_k))$, reaching the minimum.

A distributional optimizer starting at $\mu_0 = \mathcal{N}(0.5, 0.5^2)$ (a Gaussian centered at 0.5 with std 0.5) follows the JKO scheme. Each iteration: - Compute the expected cost $\int C \, d\mu_k$ and its gradient. - Move the distribution in the descent direction, with a Wasserstein penalty limiting the step.

The distribution slowly drifts toward $\theta = \pi$, simultaneously narrowing as the entropy decreases (in unregularized JKO) or staying broad (in entropy-regularized JKO).

After many steps, the unregularized JKO converges to a delta at π, while entropy-regularized JKO converges to a Boltzmann distribution $\exp(-C/\tau)$ concentrated near π but with some spread.

For a multi-modal landscape (e.g., $C(\theta) = \cos(2\theta)$ with minima at $\pi/2$ and $-\pi/2$), the distributional optimizer can have mass at both minima simultaneously, while the point optimizer is committed to one.

This is the conceptual benefit: distributional optimizers handle multi-modal landscapes natively.

Visualization

Pending. A 1D plot showing the evolution of μ_t over time on a non-convex $C(\theta)$ landscape. Initial Gaussian, then progressively concentrated at the global minimum. Side-by-side with a point-based GD trajectory. Caption: "The distribution flows downhill. The point only moves once."

Exercises

1. For $C(\theta) = \theta^2$ and $\mu_0 = \mathcal{N}(2, 1)$, compute the JKO update with $h = 0.1$. What is μ_1?
2. The Fokker-Planck equation has stationary distribution $\mu^* \propto \exp(-C/\tau)$. Verify this for $C(\theta) = \theta^2$. What is the limit as $\tau \to 0$?
3. (VQA) Sketch how an entropy-regularized population optimizer would behave on a 2D barren plateau landscape with a small "gorge" containing the global minimum. Compare to a point-based optimizer with random init.

Research Extensions

- **Sinkhorn-based VQA optimizers** — practical population optimizers with Wasserstein structure.
- **Wasserstein natural gradient** — combining the Fubini-Study metric with the Wasserstein metric.
- **Entropy regularization for barren plateaus** — using temperature annealing to escape plateaus.
- **JKO schemes for VQA** — algorithmic implementations of the JKO discretization tailored to quantum costs.

Cross-links to Other Courses

- **Optimal Transport (Villani, Peyré-Cuturi)** — the underlying mathematical theory.
- **Statistical Mechanics** — Brownian motion, Fokker-Planck, thermal exploration.
- **Generative Models in ML** — Wasserstein GANs and related distributional methods.
- **Module 5 (Regularization)** — the dynamics-side framework.
- **Module 6 Section 6.7 (Population-Based Optimizers)** — the practical embodiment.

Population-Based Optimizers

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.7

A point-based optimizer (everything in 6.1-6.5) maintains a single parameter vector and updates it based on local information. A **population-based optimizer** maintains a *set* of parameter vectors — a population — and updates them collectively.

This is a different optimization paradigm. Instead of "follow the gradient," it's "explore via diversity." Instead of "find a critical point," it's "concentrate the population on good regions of parameter space."

Population methods are the natural fit for VQA in two specific situations:

1. **Multi-modal landscapes** where a single optimizer would get stuck in a local minimum.
2. **Plateau regions** where individual optimizers can't make progress, but a population can collectively discover non-plateau regions.

This node introduces the population-based framework, walks through the major algorithm families (genetic algorithms, evolution strategies, particle swarm, CMA-ES), and sets up node 6.8 (HOPSO, the thesis's preferred hybrid).

Why Populations?

The basic argument for populations:

1. Implicit parallelism. A population of N particles samples N different parameter regions simultaneously. Even if any single particle gets stuck, the population as a whole has more information about the landscape.

2. Diversity preservation. Population methods can be designed to maintain diversity (spread among particles), which prevents premature convergence to a single minimum. This is the population analogue of entropy regularization.

3. No gradient required. Most population methods are zero-order — they only need cost evaluations, no gradients. This makes them robust to non-differentiable cost functions, discrete parameters, and simulation-based settings.

4. Robust to noise. Aggregating over a population reduces the influence of noise on any individual evaluation. The population's mean (or median) is a more robust statistic than any single particle's cost.

5. Embarrassingly parallel. The N cost evaluations per iteration are independent and can be run in parallel on N quantum devices (or sequentially with no synchronization).

The cost: each iteration evaluates N costs instead of 1, so per-iteration shot budget is N times higher. The benefit needs to outweigh this cost.

The Algorithm Families

Population methods come in several distinct families:

1. Genetic Algorithms (GA)

- Maintain a population of "individuals" (parameter vectors).
- Each generation: evaluate fitness, select the best, produce offspring via mutation and crossover.
- Inspired by biological evolution.

Strengths: very general, work on discrete and combinatorial problems. **Weaknesses:** slow, lots of hyperparameters, hard to tune.

2. Evolution Strategies (ES)

- Similar to GA but use Gaussian mutations (no crossover).
- The current "best" is updated by averaging successful mutations.
- More principled than GA, with cleaner convergence properties.

Strengths: simpler than GA, scales better, has theoretical foundations. **Weaknesses:** still slow compared to gradient methods on smooth problems.

3. CMA-ES (Covariance Matrix Adaptation Evolution Strategy)

- Hansen and Ostermeier (2001).
- The current "gold standard" of evolution strategies.
- Maintains a *covariance matrix* over parameters, adapted online to capture the local landscape geometry.
- Effectively learns a natural-gradient-like preconditioner from the population statistics.

Strengths: very robust, handles ill-conditioning, often competitive with gradient methods on noisy problems. **Weaknesses:** $O(P^2)$ covariance matrix, expensive for large P.

4. Particle Swarm Optimization (PSO)

- Eberhart and Kennedy (1995).
- Particles move through parameter space with velocities updated based on:
  - Inertia (continue current direction).
  - Personal best (return to best point seen).
  - Global best (move toward best point in population).
- Inspired by flocking behavior of birds.

Strengths: simple, intuitive, easy to implement. **Weaknesses:** can prematurely converge, requires careful inertia tuning.

5. Differential Evolution (DE)

- Storn and Price (1997).
- Update each particle by adding the difference between two random other particles.
- Simple and effective on rugged landscapes.

Strengths: simple, robust, few hyperparameters. **Weaknesses:** slow on smooth problems, no built-in noise handling.

6. Bayesian Optimization (BO)

- Maintain a Gaussian process posterior over the cost function.
- At each iteration, query the point that maximizes an acquisition function (e.g., expected improvement).
- Population is implicit (the GP uses all past observations).

Strengths: very data-efficient, principled handling of uncertainty. **Weaknesses:** $O(k^3)$ cost in number of past observations, doesn't scale to high P.

For VQA, **CMA-ES** and **PSO** (and the HOPSO variant in 6.8) are the most commonly used population methods. **BO** is used when shots are extremely expensive (real hardware experiments with limited time).

CMA-ES In Detail

CMA-ES is the canonical sophisticated evolution strategy. Here's the core update.

State: a mean vector $\mathbf{m}_k \in \mathbb{R}^P$, a covariance matrix $C_k \in \mathbb{R}^{P\times P}$, and a step size $\sigma_k > 0$.

At each generation:

1. **Sample.** Draw λ candidate solutions $\boldsymbol{\theta}_k^{(i)} \sim \mathcal{N}(\mathbf{m}_k, \sigma_k^2 C_k)$ for $i = 1, \ldots, \lambda$.
2. **Evaluate.** Compute $C(\boldsymbol{\theta}_k^{(i)})$ for each candidate.
3. **Select.** Sort by cost and keep the top $\mu < \lambda$ (the "elite").
4. **Update mean.** $\mathbf{m}_{k+1} = \sum_{i=1}^{\mu} w_i \boldsymbol{\theta}_k^{(i)}$, a weighted average of the elite.
5. **Update covariance.** Adapt C_k using the elite's distribution: increase variance in directions where the elite spread out, decrease in directions where it concentrated.
6. **Update step size.** Adapt σ_k based on the recent successful directions.

The covariance adaptation is the magic: over many generations, C_k converges to (a scalar multiple of) the local Hessian inverse, so the sampling distribution becomes aligned with the landscape geometry. CMA-ES is, in this sense, a population-based approximation to natural gradient — and it doesn't need any gradient computations.

For a detailed treatment, see Hansen's tutorial (2016).

CMA-ES For VQA

CMA-ES has been used for VQA in several papers, with mixed results:

When it works: - $P < 50$ (so the covariance matrix is manageable). - The cost landscape is smooth and the local structure is approximately quadratic. - Shot noise is moderate (CMA-ES handles it via population averaging).

When it doesn't: - Large P (covariance matrix becomes prohibitive). - Strong barren plateaus (CMA-ES gets confused when there's no signal). - Very rugged landscapes (the covariance adaptation lags behind the local structure).

For typical VQE problems with 10-30 parameters, CMA-ES is a competitive alternative to Adam, sometimes finding lower minima at the cost of more total cost evaluations. For larger problems, it's typically infeasible.

Particle Swarm For VQA

PSO is simpler and scales better but has its own issues:

Update rule for particle i:

$$\mathbf{v}_i^{k+1} = w\mathbf{v}_i^k + c_1 r_1(\mathbf{p}_i - \boldsymbol{\theta}_i^k) + c_2 r_2(\mathbf{g} - \boldsymbol{\theta}_i^k),$$
$$\boldsymbol{\theta}_i^{k+1} = \boldsymbol{\theta}_i^k + \mathbf{v}_i^{k+1},$$

where w is the inertia, c_1, c_2 are acceleration coefficients, r_1, r_2 are uniform random numbers, $\mathbf{p}_i$ is particle i's personal best, and $\mathbf{g}$ is the global best.

Strengths: - Very simple, no covariance matrix. - Scales linearly in P. - Easy to interpret and modify.

Weaknesses: - Can prematurely converge to the global best, losing diversity. - Sensitive to inertia/acceleration tuning. - Doesn't adapt to landscape geometry like CMA-ES does.

For VQA with many parameters, PSO is one of the few population methods that remains tractable. It is the foundation for HOPSO (Section 6.8), which is the thesis's specific PSO variant.

The Diversity Problem

A central tension in population methods: **how to maintain diversity without losing convergence speed**.

Too much diversity: the population explores uniformly but never converges. Too little diversity: the population collapses to a single point (effectively becoming a point optimizer) and loses the benefits of being a population.

Mechanisms to maintain diversity:

- **Niching:** explicitly assign different particles to different "niches" of parameter space.
- **Crowding:** penalize particles that get too close to each other.
- **Mutation rate:** keep mutation high enough to spread the population.
- **Restart:** periodically reset some particles to random points.
- **Multi-modal selection:** in CMA-ES, use multi-modal variants that maintain multiple covariance matrices.

For VQA, the right diversity strategy depends on whether the landscape is multi-modal (use diversity-preserving variants) or unimodal but plateau-prone (use restart-based variants).

Population Methods And Stochastic Optimization Theory

Population methods don't fit the classical stochastic optimization framework cleanly. The convergence theory for ES, CMA-ES, and PSO is its own subfield, with weaker guarantees than SGD theory.

That said, several recent results have shown that **CMA-ES is approximately equivalent to stochastic natural gradient on a Gaussian distribution** (Akimoto et al., Ollivier et al.), and that **PSO is approximately equivalent to a discrete-time approximation of the Wasserstein gradient flow** (Section 6.6).

This gives a theoretical foundation that connects population methods to the rest of the optimization literature, even though they look quite different at the algorithm level.

Structural Role

This node is the **distributional / population** entry point of Module 6. It introduces the population-based paradigm, surveys the major algorithms, and sets up node 6.8 (HOPSO) as the specific hybrid that the thesis advocates.

It also forwards directly to Module 5 (regularization), where many regularization techniques have natural population-based interpretations.

Relations to Other Concepts

- **Hansen 2016 (CMA-ES tutorial)** — the canonical reference for CMA-ES.
- **Eberhart-Kennedy 1995 (PSO)** — the original particle swarm paper.
- **Ollivier et al. 2017** — connecting CMA-ES to natural gradient.
- **Wasserstein gradient flow (6.6)** — the continuous limit of population dynamics.
- **Module 5 regularization** — population methods naturally implement many regularization ideas.

Worked Example — PSO On A 2D Cost

Consider $C(\theta_1, \theta_2) = (\theta_1 - 1)^2 + (\theta_2 - 2)^2$, with the minimum at $(1, 2)$.

Setup: 5 particles, initialized uniformly at random in $[-3, 3]^2$. Inertia $w = 0.7$, acceleration $c_1 = c_2 = 1.5$, velocities initialized to zero.

Iteration 1: - Particle positions: $(1.5, -2), (-2, 1), (0, 0), (3, 3), (-1, -3)$. - Costs: $16.25, 10, 5, 5, 29$. - Global best so far: $(0, 0)$ or $(3, 3)$, tie at cost 5. Pick $(0, 0)$. - Each particle's personal best is its current position.

Iteration 2: - Each particle accelerates toward $(0, 0)$ (global best) and toward its personal best (which is its current position, so no contribution). - New velocities: $w \cdot 0 + c_1 r_1 \cdot 0 + c_2 r_2(\mathbf{g} - \boldsymbol{\theta}_i)$ — random pull toward the global best. - Particles move toward $(0, 0)$, but with noise.

Iteration 3-10: - The particles cluster around the current global best, but explore the area. - One eventually finds $(1, 2)$ (the true minimum) and becomes the new global best. - The rest of the population follows.

Convergence: typically ~ 30 iterations for 2D problems, ~ 200 for 10D problems, $\sim 1000+$ for higher dimensions. The cost per iteration is 5 evaluations (for 5 particles), so the total cost is 5x the number of iterations.

For VQA, this scales linearly in the number of particles (which gives the population effect) and roughly as a polynomial in P (PSO doesn't suffer from the curse of dimensionality as badly as some methods).

Visualization

Pending. A 2D contour plot of a multi-modal landscape with 5 PSO particle trajectories overlaid. Initial random scatter, then gradual convergence on the global minimum, with some particles briefly visiting local minima. Caption: "Particles explore. Eventually, they vote with their feet."

Exercises

1. For PSO with 3 particles on $C(\theta) = \theta^2$, starting at $\theta = -2, 0, 1$ with zero velocities, run 5 iterations by hand and verify convergence toward $\theta = 0$.
2. CMA-ES adapts a covariance matrix. For a quadratic cost $C = \boldsymbol{\theta}^T A \boldsymbol{\theta}$, what does the converged covariance matrix look like in terms of A?
3. (VQA) For a 100-parameter VQA, estimate the per-iteration shot budget for: (a) a vanilla parameter-shift gradient; (b) CMA-ES with 50 particles; (c) PSO with 50 particles. Which is most expensive?

Research Extensions

- **Quantum-aware population methods** — variants that exploit VQA structure (periodicity, parameter-shift exact gradients).
- **Hybrid gradient + population methods** — using parameter-shift gradients to inform PSO or ES updates.
- **Diversity-aware CMA-ES for plateau escape** — variants that maintain multi-modal populations on barren plateau landscapes.
- **Bayesian optimization with quantum priors** — incorporating quantum-specific structure into the GP prior.

Cross-links to Other Courses

- **Evolutionary Computation** — the broader field of bio-inspired optimization.
- **Bayesian Optimization** — Gaussian process optimization, expected improvement.
- **Module 6 Section 6.6 (Optimal Transport)** — the continuous-limit framework.
- **Module 6 Section 6.8 (HOPSO)** — the specific PSO variant.

HOPSO: Hybrid Particle-Swarm Optimizer

Position in Knowledge Graph

System: Variational Quantum Algorithms **Structural Domain:** Optimization Dynamics **Node:** 6.8

This is the **closing node** of Module 6 and the most thesis-specific.

HOPSO — **Hybrid Optimizer Particle Swarm Optimization** — is the optimizer the thesis uses as its primary case study for the regularization-as-dynamical-modification framework. It is a hybrid of:

1. **Population-based exploration** (from PSO, Section 6.7), giving it the ability to escape local minima and barren plateau regions through population diversity.
2. **Per-particle gradient descent** (from parameter-shift, Section 6.1), giving each particle the ability to refine its position via gradient information rather than relying purely on swarm dynamics.

The result is an optimizer that combines the global-exploration benefit of populations with the local-precision benefit of gradients. It is the thesis's preferred answer to "what optimizer should you use for a difficult VQA," with the understanding that "difficult" means "barren-plateau-prone, multi-modal, or both."

This node walks through HOPSO's design, why each component is justified, and how it embodies the regularization-as-dynamics-modification thesis.

The HOPSO Update Rule

Each particle in the swarm has a position $\boldsymbol{\theta}_i^k$ and a velocity $\mathbf{v}_i^k$. At each iteration:

Step 1: Local gradient. Each particle computes its local cost gradient $\widehat{\nabla C}(\boldsymbol{\theta}_i^k)$ via parameter-shift, with whatever shot budget is allocated.

Step 2: Velocity update. The velocity is updated by combining gradient descent with PSO swarm terms:

$$\mathbf{v}_i^{k+1} = w\mathbf{v}_i^k + c_g(-\widehat{\nabla C}(\boldsymbol{\theta}_i^k)) + c_p r_p(\mathbf{p}_i - \boldsymbol{\theta}_i^k) + c_s r_s(\mathbf{g} - \boldsymbol{\theta}_i^k),$$

where: - w is the inertia (typically 0.5-0.9). - c_g is the gradient coefficient (the "gradient pull"). - c_p is the personal-best coefficient. - c_s is the swarm-best (global) coefficient. - r_p, r_s are uniform random numbers in $[0, 1]$. - $\mathbf{p}_i$ is particle i's personal-best position so far. - $\mathbf{g}$ is the swarm's global-best position so far.

Step 3: Position update. $\boldsymbol{\theta}_i^{k+1} = \boldsymbol{\theta}_i^k + \mathbf{v}_i^{k+1}$.

Step 4: Personal/global best update. If the new position has a better cost than $\mathbf{p}_i$, update $\mathbf{p}_i$. If better than $\mathbf{g}$, update $\mathbf{g}$.

Step 5: Diversity check. If the swarm has collapsed (all particles within some radius), inject mutation or restart some particles.

What Each Term Does

The four velocity terms each play a distinct role.

1. Inertia term $w\mathbf{v}_i^k$. Maintains the previous velocity, smoothing the trajectory. Like momentum in classical SGD.

2. Gradient term $c_g(-\widehat{\nabla C})$. Pulls each particle in its local descent direction. This is the "particle does GD" component. In a region where the gradient is reliable (above noise floor, no plateau), this makes the particle converge quickly.

3. Personal-best term $c_p r_p(\mathbf{p}_i - \boldsymbol{\theta}_i^k)$. Pulls the particle back toward its own best historical position. This term is *self-correcting*: if a particle wanders into a bad region, it gets pulled back to its best known location.

4. Global-best term $c_s r_s(\mathbf{g} - \boldsymbol{\theta}_i^k)$. Pulls the particle toward the swarm's global best. This is the *social* component, which lets information flow between particles. The swarm collectively concentrates on the best region found so far.

The mix of gradient (local information) and PSO terms (population information) is what makes HOPSO a *hybrid*.

Why HOPSO Works In The Plateau Regime

The key claim of the thesis is that HOPSO can find solutions in regions where vanilla gradient descent fails because of barren plateaus.

Mechanism 1: Population diversity beats single-point starting. A vanilla GD optimizer with random initialization has a probability of landing in a non-plateau region that decays exponentially with qubit count. A swarm of N particles has a probability $\sim N \times 2^{-n}$ of having at least one particle land in a non-plateau region — still exponentially small, but multiplied by N. For $N = 100$, this is a 100x improvement, which can be the difference between "no signal" and "some signal."

Mechanism 2: Particles in the plateau follow the leader. Once one particle finds a non-plateau region (via the global-best pull, or by random PSO motion), the rest of the swarm is pulled toward it. The plateau-trapped particles don't need to find the signal themselves — they just need to follow the leader.

Mechanism 3: Gradient terms become useful again outside the plateau. Once a particle is outside the plateau, its local gradient is meaningful and the gradient term in HOPSO accelerates convergence. The PSO terms get the particle out of the plateau; the gradient term takes it from "in a non-plateau region" to "at a minimum."

Mechanism 4: Diversity preservation. HOPSO's diversity-check step prevents the swarm from collapsing entirely, which would destroy the population benefit. By keeping some spread, the optimizer continues to explore even after concentrating most of its mass.

This is the core thesis claim: **plateau escape via population diversity, plateau exploitation via local gradients**.

HOPSO And The Regularization-As-Dynamics Thesis

HOPSO's connection to the thesis runs deeper than "it works on plateaus."

The thesis distinguishes: - **Objective-side regularization:** modify the cost function (add penalty terms). - **Dynamics-side regularization:** modify the optimizer dynamics (don't touch the cost).

HOPSO is **purely dynamics-side**. The cost function is unchanged. The regularization happens through the optimizer's own structure: the population maintains diversity, the swarm dynamics encourage exploration, the gradient terms encourage exploitation. There is no added term in $C(\boldsymbol{\theta})$.

This is operationally meaningful because:

1. **You don't need to know the right penalty term.** Adding a penalty to C requires designing the penalty, choosing its weight, tuning its hyperparameters. HOPSO has its own hyperparameters (w, c_g, c_p, c_s) but they parameterize the optimizer's behavior, not the problem.
2. **You don't change the global minimum.** A penalty in C shifts the minimum (intentionally — that's its purpose). HOPSO leaves the minimum where it is. If the optimizer converges, it converges to the *original* minimum, not a regularized one.

3. **The regularization is adaptive.** The swarm's exploration vs exploitation balance shifts over time as the population concentrates. This is a natural form of annealing without requiring an explicit schedule.
4. **The framework is composable.** You can add objective-side regularization on top of HOPSO if you want both. The two are not mutually exclusive — they are different layers of the optimization stack.

The thesis argues that **dynamics-side regularization deserves equal billing with objective-side**, and HOPSO is a clean example of why: it solves real problems (barren plateau escape) without requiring any modification to the cost function.

Hyperparameters And Tuning

HOPSO has more hyperparameters than vanilla GD or even Adam. The main ones:

- N: population size. Typical $20 - 100$.
- w: inertia. Typical $0.5 - 0.9$.
- c_g: gradient coefficient. Typical $0.1 - 1.0$ (depends on landscape gradient scale).
- c_p: personal best coefficient. Typical $0.5 - 2.0$.
- c_s: global best coefficient. Typical $0.5 - 2.0$.
- Restart threshold: when the swarm has collapsed below this radius, inject mutations.
- Shot budget per particle per iteration.

This is more knobs than most VQA optimizers, which is a real cost. But the defaults work reasonably well across a wide range of problems, and tuning is rarely needed for casual use.

A practical default: $N = 50$, $w = 0.7$, $c_g = 0.3$, $c_p = 1.0$, $c_s = 1.0$, restart threshold 0.1, shot budget per particle = same as vanilla GD baseline.

When HOPSO Is And Isn't The Right Choice

HOPSO is a good choice for:

- **Difficult VQAs with barren plateaus** that vanilla GD can't escape.
- **Multi-modal landscapes** where the optimizer needs to explore multiple basins.
- **Noisy hardware** where the population averaging helps stabilize the trajectory.
- **Problems with adequate compute** to support a large population per iteration.

HOPSO is **not** a good choice for:

- **Simple smooth VQAs** where vanilla GD or Adam works fine. The population overhead is wasted.
- **Very high-dimensional problems** ($P > 200$) where the population can't densely sample parameter space anyway.
- **Time-critical experiments** where the per-iteration cost is prohibitive.
- **Problems with very expensive cost evaluations** (e.g., real hardware with limited time slots) where shooting your shot budget across 50 particles is too costly.

For "small-easy" problems, use Adam. For "big-hard" problems, use HOPSO. For everything in between, it depends on your shot budget and your patience.

Empirical Performance Of HOPSO

The thesis reports empirical comparisons of HOPSO against vanilla GD, Adam, natural gradient, and other baselines on a suite of VQA problems. The summary findings (paraphrased):

1. **On smooth, plateau-free problems** ($n \leq 8$ qubits, shallow ansatz), HOPSO is comparable to or slightly slower than Adam in total cost. The population overhead is wasted on easy problems.
2. **On medium-difficulty problems** ($n = 10 - 15$, moderate plateau risk), HOPSO is competitive with Adam and significantly more reliable — it succeeds on a higher fraction of random initializations.
3. **On hard problems** ($n = 18 - 24$, strong plateau risk, multi-modal), HOPSO finds significantly better minima than Adam or natural gradient. The gap widens with problem difficulty.
4. **On very hard problems** ($n > 24$ with deep ansatz), all optimizers struggle, but HOPSO degrades more gracefully than gradient methods.

The thesis interprets this as evidence that **population-based exploration becomes more important as problem difficulty grows**, which is exactly what the regularization-as-dynamics framing predicts.

Closing Module 6

Module 6 has now traversed the full optimizer landscape:

- **Vanilla GD (6.1):** the baseline.
- **Natural Gradient (6.2):** geometric correctness at high cost.
- **Adam (6.3):** practical adaptive methods, the workhorse default.
- **Stochastic Optimization (6.4):** the foundational theory.
- **Convergence Theory (6.5):** what we can prove.
- **Optimal Transport (6.6):** the distributional view.
- **Population Methods (6.7):** the practical population framework.
- **HOPSO (6.8, this node):** the thesis's hybrid recommendation.

The arc of the module is from "simplest possible optimizer" to "most sophisticated thesis-aligned hybrid." Each step adds a layer that addresses a failure mode of the previous one.

The most important take-away: **there is no single best optimizer for VQA**. The right choice depends on the problem's difficulty, the noise level, the parameter count, and the available compute. The thesis's specific recommendation — HOPSO for difficult problems, Adam for easy ones — is one reasonable rule, but the broader framework (point vs population, gradient vs zero-order, objective-side vs dynamics-side regularization) is what matters.

Module 7 will look at all of this through a control-theoretic lens. Module 8 will use it to build the expressivity-trainability tradeoff. Modules 9 and 10 will tie it back to hardware and deployment.

Structural Role

This node is the **operational closure** of Module 6. It is also the most direct embodiment of the thesis's central optimization-side argument: that population-based, dynamics-side regularization is a viable and underexploited alternative to objective-side fixes.

Relations to Other Concepts

- **PSO (Eberhart-Kennedy 1995)** — the parent algorithm.
- **CMA-ES** — alternative population method with adaptive geometry.
- **Hybrid optimization** — the broader category of gradient + population mixes.
- **Module 5 regularization** — the framework HOPSO embodies.
- **Wasserstein gradient flow (6.6)** — the continuous limit of HOPSO's swarm dynamics.

Worked Example — HOPSO On A Plateau-Prone Toy

Consider a 6-qubit HEA with 3 layers and a global cost $C = \langle Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 \rangle$.

Vanilla GD with random init. Initial gradient is ~ 0.05, shot noise floor is ~ 0.03. The optimizer barely moves, and after 1000 iterations, the cost has decreased by < 0.01. **Failure.**

Adam with random init. Same regime. Adam's adaptive step size doesn't help — the gradient is too noisy. Same failure.

HOPSO with $N = 30$ particles. - Iteration 1: 30 particles initialized randomly. Global best particle has cost ~ 0.1 (better than typical). - Iteration 10: the swarm has concentrated around the global best. A few particles have wandered into less-plateau regions and discovered cost ~ -0.2. - Iteration 50: the global best is now -0.6. The gradient at this point is meaningful (~ 0.3), and the gradient terms in HOPSO start contributing significantly. - Iteration 200: the swarm has converged on the global minimum at cost -1.

Total cost comparison: - HOPSO: 200 iterations $\times 30$ particles $\times (2P + 1)$ cost evaluations per iteration $=\sim 200 \times 30 \times 37 \approx 220{,}000$ cost evaluations. - Vanilla GD that fails: 1000 iterations $\times (2P + 1) \approx 37{,}000$ cost evaluations, but doesn't reach the answer.

HOPSO uses 6x more cost evaluations but actually finds the minimum. Vanilla GD uses fewer but fails.

This is the canonical HOPSO benefit: **more total cost, but finds answers that gradient methods cannot.**

Visualization

Pending. A 2D plot showing two trajectories on a multi-modal cost landscape with a plateau region: vanilla GD (gets stuck in the plateau, never reaches the minimum) and HOPSO (population spreads, one particle finds the minimum, rest of the swarm converges). Caption: "HOPSO escapes plateaus that GD cannot."

Exercises

1. For a 4-parameter ansatz with cost $C = \theta_1^2 + \theta_2^2 + \theta_3^2 + \theta_4^2$, run HOPSO with 5 particles for 10 iterations from random initialization. How fast does the global best converge to zero?
2. For HOPSO on a 1D bimodal cost $C(\theta) = \cos(2\theta)$, starting with a uniform initial swarm, sketch how the population evolves and which minimum it converges to.
3. (VQA) For a 20-qubit barren-plateau-prone HEA, estimate the minimum population size N such that HOPSO has $> 50\%$ probability of having at least one particle outside the plateau region at initialization. (Use any reasonable assumption about the plateau coverage fraction.)

Research Extensions

- **Adaptive HOPSO hyperparameters** — auto-tuning w, c_g, c_p, c_s based on swarm behavior.
- **HOPSO with quantum-aware mutations** — using parameter-shift periodicity to design mutation operators.
- **Multi-population HOPSO** — running multiple independent swarms in parallel, with occasional cross-population exchange.
- **HOPSO + natural gradient** — combining the swarm dynamics with QFIM-rescaled gradient terms.

Cross-links to Other Courses

- **Evolutionary Computation** — the parent field of population methods.
- **Module 5 (Regularization)** — the theoretical framework HOPSO embodies.
- **Module 6 Section 6.7 (Population Methods)** — the broader class.
- **Module 7 (Control-Theoretic View)** — alternative framing of the same dynamics.