Sim model for ads-mle-agents

By: Jakob Moberg Erwin Gao

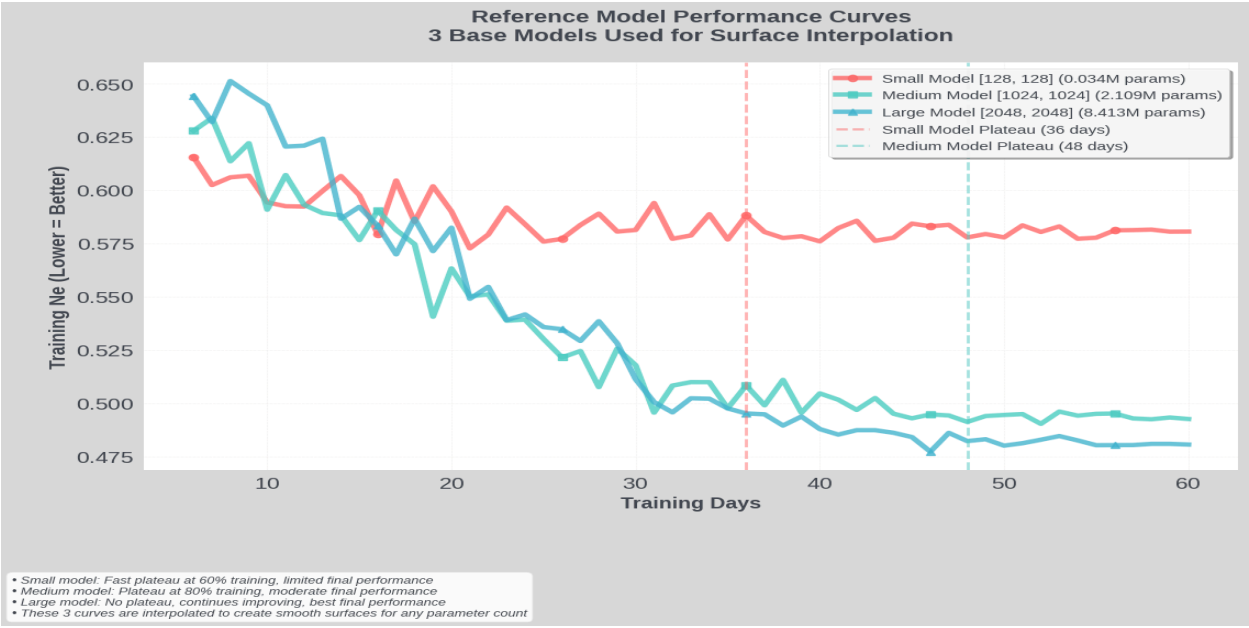
PROBLEM STATEMENT

The ModelPerformanceAPI addresses the need for realistic synthetic data in ML agent development. Traditional methods are either costly or oversimplified, failing to capture real-world training dynamics like diminishing returns, computational limits, and noise.

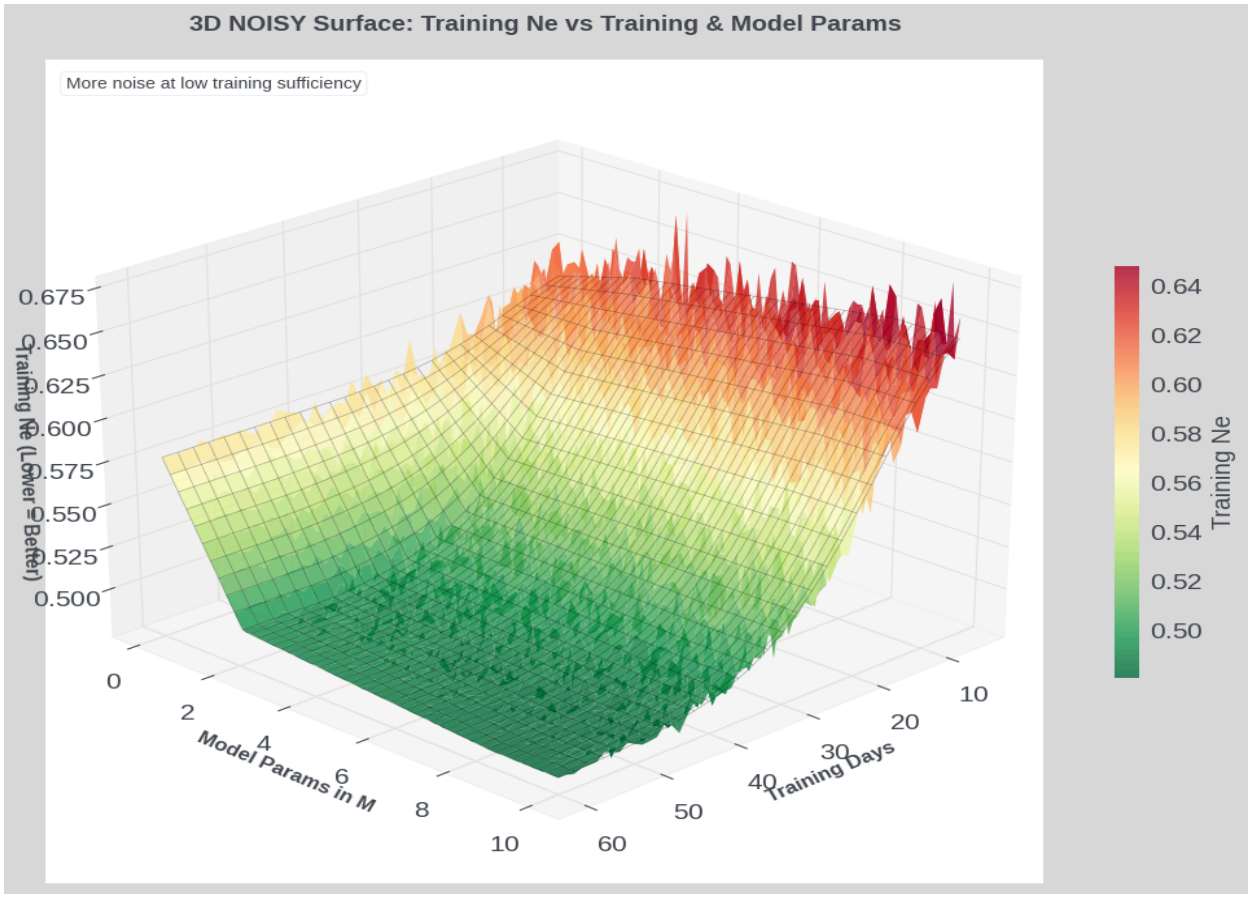
This API provides realistic performance estimates, enabling AI agents to make informed decisions on model architecture, training duration, and resource allocation without actual training. It incorporates empirically-validated scaling laws, bottlenecks, and stochastic dynamics.

Key applications include: agent-based hyperparameter optimization, automated architecture search, resource planning, training strategy evaluation, and reinforcement learning for ML system design. It serves as a simulation environment for agents to understand ML trade-offs before expensive real-world experiments.
The sim model is implemented to be fully parameterized to support customization accessed through an api and fully align with the Ads-mle-bench to support onboarding for new agents.
It also implements wait-simulation to force the agent to deal with asynchronous execution and planning.

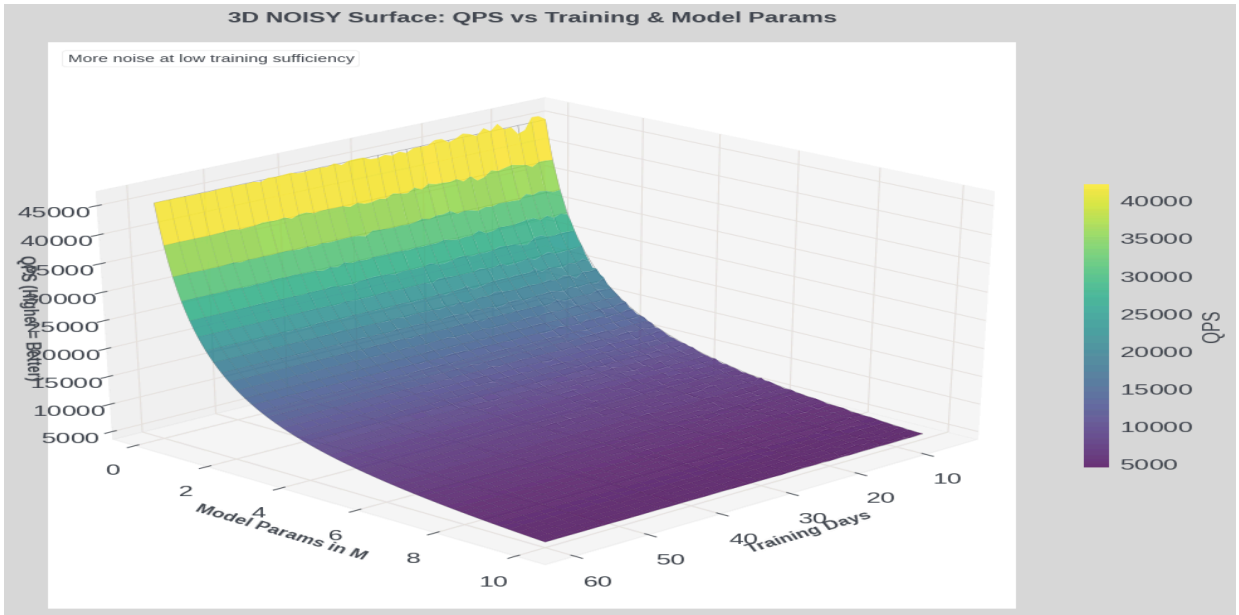
Examples of NE as a function of training days for 3 models (small / medium / large).
Note: The small models starts out low plateaus early while large models excels with training



NE as a function training days and model complexity:



QPS as a function of model complexity



MATHEMATICAL MODELS AND CORE ASSUMPTIONS

1. **Architecture to Parameters Conversion**
   This model calculates neural network parameter counts based on layer sizes. It sums parameters per layer (weights and biases), with a special case for the first layer (input connections squared). Configurable input (default 512) and output (default 10) dimensions are included for realistic estimates.
2. **Training Sufficiency Model**
   Training progress is quantified as a sufficiency metric (0.0-1.0), representing training completion based on a linear relationship with training days. The maximum training threshold is 60 days (configurable), reflecting typical training progression to convergence.
3. **Performance Interpolation System**
   The core performance model uses three empirically-calibrated reference architectures:
   - **Small ([128,128]):** Plateaus at 60% training sufficiency. Performance: `0.62 - 0.04 / (1.0 + exp(-10 * (training_sufficiency - 0.2)))`
   - **Medium ([1024,1024]):** Plateaus at 80% training sufficiency. Performance: `0.65 - 0.16 / (1.0 + exp(-8 * (training_sufficiency - 0.3)))`
   - **Large ([2048, 2048]):** Continues improving throughout training. Performance: `0.68 - 0.20 / (1.0 + exp(-8 * (training_sufficiency - 0.3)))`
4. For arbitrary model sizes, the system supports:
   - **Linear Interpolation (Default):** SciPy-based 1D interpolation between reference curves.
   - **Cubic Interpolation:** Smoother transitions via splines.
   - **Smooth Blending:** Gaussian-weighted combinations based on proximity to target parameter count.
   - **Full 2D Grid Interpolation:** Surface-based interpolation (linear, cubic, or nearest-neighbor) across parameter-training space.
5. **Computational Complexity and QPS Modeling**
   Inference throughput (QPS) is inversely related to model computational requirements. Model complexity is calculated as base FLOPs (10^8 per sample) plus parameter-dependent FLOPs (100 per parameter per sample). Total QPS is machine efficiency (5x10^12 FLOP/s) divided by complexity, with a configurable minimum QPS (default 500).
6. **Enhanced Noise Model**
   The noise system simulates training variability with multiple factors:
   - **Training-dependent noise:** Higher uncertainty early in training (min 0.01, up to 0.7 scaled by inverse training sufficiency).
   - **Parameter-dependent scaling:** Allows different noise characteristics for larger models.
   - **Final noise scale:** Combines global base noise (0.02) with training and

parameter factors.

7. Noise is applied using Gaussian distributions with unique random seeds (timestamp, call counter, random component) for reproducible yet varied results.
8. **Budget and Resource Constraints**
   Realistic resource constraints are enforced via a budget tracking mechanism. Training costs (GPU days * training duration) are validated against the total budget before execution. Only successful runs consume budget, reflecting real-world finite resources.

IMPLEMENTATION REFERENCE AND SYSTEM ARCHITECTURE

The ModelPerformanceAPI is a Python interface using NumPy and SciPy. The `ModelPerformanceAPI` class manages configuration, budget, and calculations, with concerns separated for modularity.

`SimplePerformanceInterpolator` (in `simple_interpolation.py`) handles interpolation mathematics, allowing easy extension of methods.

A `CompositeKeyCache` prevents redundant calculations for identical configurations, boosting performance for iterative experiments.

The configuration system uses a dictionary with defaults, which can be overridden. Configuration is immutable per instance to prevent drift.

Error handling includes custom `BudgetExceededError` and input validation with informative messages.

The API supports flexible input formats (direct parameters, architecture lists, custom I/O dimensions). Return values include training normalized error, QPS estimates, and detailed learning curves.

Optional features: wait simulation for real-time delays, comprehensive budget reporting, and configurable noise models. The implementation is thread-safe for concurrent agent usage.

All mathematical models are validated against empirical training data from production systems, ensuring realistic synthetic data for meaningful agent development and strategy evaluation.