

Case Study: Anomaly Detection in High-Energy Particle Physics Using Energy-Based Models and Score-Guided Sampling

Section 1: Industry Context and Business Problem

Industry: Particle Physics and High-Energy Physics Research

Company Profile: NovaCERN Analytics is a computational physics startup that provides anomaly detection services to particle physics research laboratories worldwide. Founded by former CERN data scientists, the company specializes in identifying rare particle interaction events from the massive data streams produced by particle collider experiments.

Business Challenge

Modern particle colliders like the Large Hadron Collider (LHC) produce approximately 40 million collision events per second. Each event generates a complex multi-dimensional data record describing particle trajectories, energies, and interaction signatures. The vast majority of these events are well-understood "background" processes predicted by the Standard Model of physics.

The scientific goal is to find **anomalous events** -- interactions that deviate from known physics and may signal new particles, new forces, or other discoveries beyond the Standard Model. These events are extremely rare: perhaps 1 in every 10 billion collisions.

Stakes

- **Scientific:** Missing a genuine anomaly could mean overlooking the next major physics discovery -- the equivalent of missing the Higgs boson signal.
- **Financial:** Each year of LHC operation costs approximately \$1 billion. If the analysis pipeline cannot efficiently flag anomalies, this investment is partially wasted.
- **Computational:** The data volume is approximately 1 petabyte per second at the raw detector level. The anomaly detection system must operate within strict latency and throughput constraints.

Constraints

- **Unsupervised:** We do not know what anomalies look like (by definition, they are unknown physics). The system must detect deviations from the known background without labeled examples of anomalies.
- **High-dimensional:** Each collision event is described by 50-200 features (particle momenta, energies, angles, multiplicities).
- **Real-time adjacent:** While full offline analysis can take hours, the trigger system needs initial anomaly scoring within milliseconds.
- **False positive rate:** The system must maintain an extremely low false positive rate ($< 0.01\%$) to avoid overwhelming physicists with false alarms.

Why Energy-Based Models?

EBMs are a natural fit for this problem because:

1. **Density estimation without normalization:** EBMs assign an energy score to every event. Known physics events get low energy (high probability); anomalies get high energy (low probability). The partition function is never needed for ranking.
2. **Flexible architecture:** The energy function can be any neural network, allowing it to capture the complex correlations in particle physics data.
3. **Score-based training:** Using score matching, we can train the energy function from data samples alone, without labels and without computing the intractable partition function.
4. **Anomaly score is built in:** The energy value itself IS the anomaly score. No separate anomaly detection head is needed.

Section 2: Technical Problem Formulation

Problem Type: Unsupervised Anomaly Detection via Density Estimation

Justification: We frame anomaly detection as identifying low-density regions in the data distribution. EBMs model the density implicitly through the energy function, and the score function enables tractable training. Events with high energy (low probability under the learned model) are flagged as anomalies.

Input/Output Specifications

Input: A collision event represented as a feature vector $x \in \mathbb{R}^D$ where $D = 64$, containing: - Particle 4-momenta (energy, px, py, pz) for up to 10 leading particles - Missing transverse energy (MET) - Event-level summary features (total transverse momentum, sphericity, aplanarity) - Jet-level features (number of jets, leading jet mass, b-tag scores)

Output: - Anomaly score $a(x) = E_\theta(x)$ (higher energy = more anomalous) - Binary anomaly flag based on a threshold calibrated to the desired false positive rate - Score vector $s_\theta(x) = -\nabla_x E_\theta(x)$ for interpretability (shows which features push the event toward or away from normality)

Mathematical Foundation from First Principles

Energy-Based Density Model:

$$p_\theta(x) = \frac{\exp(-E_\theta(x))}{Z(\theta)}$$

where $E_\theta(x)$ is parameterized by a neural network and $Z(\theta) = \int \exp(-E_\theta(x)) dx$ is the intractable partition function.

Score Function:

$$s_\theta(x) = \nabla_x \log p_\theta(x) = -\nabla_x E_\theta(x)$$

The partition function vanishes because $\nabla_x \log Z = 0$ (Z does not depend on x).

Denoising Score Matching Loss:

$$\mathcal{L}(\theta) = \mathbb{E}_{x \sim p_{\text{data}}} \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} \left[\left\| s_\theta(x + \sigma \epsilon) + \frac{\epsilon}{\sigma} \right\|^2 \right]$$

Loss Function with Per-Term Justification

The total training loss combines three terms:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{DSM}} + \lambda_1 \mathcal{L}_{\text{multi-scale}} + \lambda_2 \mathcal{L}_{\text{reg}}$$

Term 1: Denoising Score Matching Loss (\mathcal{L}_{DSM})

$$\mathcal{L}_{\text{DSM}} = \mathbb{E}_{\sigma \sim p(\sigma)} \mathbb{E}_{x, \epsilon} \left[\sigma^2 \left\| s_\theta(x + \sigma \epsilon, \sigma) + \frac{\epsilon}{\sigma} \right\|^2 \right]$$

Justification: This is the core training objective. It trains the score network to predict the direction from a noisy point back to the clean data. The σ^2 weighting ensures balanced learning across noise scales. Without this term, the model cannot learn the data distribution.

Term 2: Multi-Scale Consistency ($\mathcal{L}_{\text{multi-scale}}$)

$$\mathcal{L}_{\text{multi-scale}} = \sum_{i=1}^L w_i \cdot \mathcal{L}_{\text{DSM}}(\sigma_i)$$

Justification: Using multiple noise levels ($\sigma_1 > \sigma_2 > \dots > \sigma_L$) ensures the score network captures both global structure (large σ : broad background distributions) and local anomalies (small σ : fine deviations from background). This is critical because anomalies may manifest as subtle local deviations.

Term 3: Weight Regularization (\mathcal{L}_{reg})

$$\mathcal{L}_{\text{reg}} = \|\theta\|_2^2$$

Justification: L2 regularization prevents the energy function from becoming overly complex and overfitting to training data noise. This ensures smooth energy landscapes where anomaly scores generalize to unseen events.

Evaluation Metrics

1. **AUROC** (Area Under Receiver Operating Characteristic): Measures the model's ability to rank anomalies above normal events across all thresholds.
2. **AUPRC** (Area Under Precision-Recall Curve): More informative than AUROC for highly imbalanced data (anomalies are rare).
3. **Signal Efficiency at Fixed Background Rejection:** At a 99.99% background rejection rate, what fraction of true anomalies are detected? This is the standard metric in particle physics.
4. **Latency:** Inference time per event (target: $< 5\text{ms}$ for online scoring).

Baseline

- **Autoencoder reconstruction error:** Train an autoencoder on background events; anomalies should have higher reconstruction error. This is the current industry standard.
- **Isolation Forest:** A classical anomaly detection baseline.

Why This Concept (EBMs + Score Matching)?

Compared to autoencoders, EBMs with score matching offer several advantages for this problem: - **No bottleneck assumption:** Autoencoders force data through a low-dimensional bottleneck, which may discard information relevant to anomaly detection. EBMs model the full distribution. - **Principled density estimation:** The energy function directly quantifies how likely an event is under the learned model, providing a theoretically grounded anomaly score. - **Interpretability via the score:** The score vector $s_\theta(x)$ shows which features push an event toward normality, providing physicists with interpretable explanations.

Section 3: Implementation Notebook Structure

3.1 Data Loading and Preprocessing

```
def load_particle_data(filepath: str, n_features: int = 64) -> tuple:
    """
    Load and preprocess particle physics collision event data.

    Args:
        filepath: Path to the dataset (CSV or HDF5)
        n_features: Number of features per event
```

```

Returns:
    X_train: Background events for training (n_train, n_features)
    X_val: Validation set with some injected anomalies
    X_test: Test set with labeled anomalies
    y_test: Binary labels (0 = background, 1 = anomaly)

Steps:
    1. Load raw event data
    2. Apply feature selection (keep the 64 most discriminating features)
    3. Standardize features to zero mean, unit variance
    4. Split into train (background only), val, and test sets
    5. For val/test, inject synthetic anomalies at a controlled rate

Hint: Use sklearn.preprocessing.StandardScaler for normalization.
"""
# ===== TODO =====
pass
# =====

```

```

# Verification: check data shapes and distributions
# assert X_train.shape[1] == 64, "Expected 64 features"
# assert y_test.sum() > 0, "Test set should contain anomalies"
# print(f"Training: {X_train.shape[0]} events (background only)")
# print(f"Test: {X_test.shape[0]} events ({y_test.sum()} anomalies)")

```

3.2 Exploratory Data Analysis

```

def explore_data(X_train, X_test, y_test, feature_names=None):
    """
    TODO: Visualize the training data distribution and compare
    normal vs anomalous events in the test set.

    Create:
    1. Histogram of each feature for normal vs anomalous events
    2. 2D scatter plot of top-2 PCA components, colored by label
    3. Correlation heatmap of features
    4. Distribution of pairwise distances (normal-normal vs normal-anomaly)

    Hint: Use sklearn.decomposition.PCA for dimensionality reduction.
    """
    # ===== TODO =====
    pass
    # =====

```

3.3 Baseline Model (Autoencoder)

```

def build_autoencoder(input_dim: int = 64, latent_dim: int = 8):
    """
    TODO: Build an autoencoder baseline for anomaly detection.

    Architecture:
        Encoder: input_dim -> 128 -> 64 -> latent_dim
        Decoder: latent_dim -> 64 -> 128 -> input_dim

    The anomaly score is the reconstruction error (MSE).

    Hint: Use nn.Sequential with nn.Linear and nn.ReLU layers.
    """
    # ===== TODO =====
    pass
    # =====

```

```

def train_autoencoder(model, X_train, n_epochs=100, lr=1e-3, batch_size=256):
    """
    TODO: Train the autoencoder on background events only.
    Use MSE reconstruction loss.

    Return: trained model, list of epoch losses
    """
    # ===== TODO =====

```

```
pass
# =====
```

3.4 Energy-Based Model Architecture

```
class EnergyNetwork(nn.Module):
    """
    TODO: Implement the energy network for particle physics data.

    The network takes a collision event (64-dim vector) and an optional
    noise level index, and outputs a scalar energy value.

    Architecture:
    - Input projection: 64 -> 256
    - Noise level embedding: n_sigmas -> 256 (added to input projection)
    - Hidden layers: 256 -> 256 -> 256 (with SiLU activations and residual connections)
    - Output: 256 -> 1 (scalar energy)

    For anomaly scoring, we use the energy directly: higher energy = more anomalous.
    For score computation, we use autograd: s(x) = -grad_x E(x).

    Hint: Use nn.Embedding for noise level conditioning.
    """
    def __init__(self, input_dim=64, hidden_dim=256, n_sigmas=10):
        super().__init__()
        # ===== TODO =====
        pass
        # =====

    def forward(self, x, sigma_idx=None):
        # ===== TODO =====
        pass
        # =====
```

```
class ScoreNetwork(nn.Module):
    """
    TODO: Implement the score network that directly predicts s(x, sigma).

    Same architecture as EnergyNetwork but outputs a D-dimensional score vector
    instead of a scalar energy.

    Architecture:
    - Input projection: 64 -> 256
    - Noise level embedding: n_sigmas -> 256
    - Hidden layers: 256 -> 256 -> 256 (SiLU + residual)
    - Output: 256 -> 64 (score vector)
    """
    def __init__(self, input_dim=64, hidden_dim=256, n_sigmas=10):
        super().__init__()
        # ===== TODO =====
        pass
        # =====

    def forward(self, x, sigma_idx):
        # ===== TODO =====
        pass
        # =====
```

3.5 Training Loop (DSM)

```
def train_score_model(model, X_train, sigmas, n_epochs=500, lr=1e-3, batch_size=512):
    """
    TODO: Train the score network using multi-scale denoising score matching.

    For each batch:
    1. Sample a batch of training events
    2. For each event, randomly select a noise level sigma_i
    3. Add Gaussian noise: x_noisy = x + sigma_i * epsilon
    4. Compute target score: target = -epsilon / sigma_i
    5. Predict score: pred = model(x_noisy, sigma_idx)
```

```

6. Compute weighted MSE: loss = sigma_i^2 * ||pred - target||^2
7. Backprop and update

Return: trained model, loss history

Hint: Use torch.randint to randomly select noise levels per sample.
"""
# ===== TODO =====
pass
# =====

```

3.6 Evaluation

```

def compute_anomaly_scores(model, X, sigmas, method='energy'):
    """
    TODO: Compute anomaly scores for a batch of events.

    Two methods:
    1. 'energy': Use the energy network output directly as anomaly score
    2. 'score_norm': Use the norm of the score vector at the lowest noise level

    Higher scores = more anomalous.

    Return: array of anomaly scores (n_events,)
    """
    # ===== TODO =====
    pass
    # =====

```

```

def evaluate_anomaly_detection(y_true, scores):
    """
    TODO: Compute AUROC, AUPRC, and signal efficiency at 99.99% background rejection.

    Use sklearn.metrics for AUROC and AUPRC.
    For signal efficiency: find the threshold that rejects 99.99% of background,
    then compute the fraction of anomalies above that threshold.

    Return: dict with 'auROC', 'auprc', 'signal_efficiency'
    """
    # ===== TODO =====
    pass
    # =====

```

3.7 Error Analysis

```

def error_analysis(model, X_test, y_test, scores, sigmas):
    """
    TODO: Analyze model failures.

    1. Find false positives (normal events with high anomaly scores)
       - Visualize their feature distributions
       - Are they genuinely unusual background events?

    2. Find false negatives (anomalies with low anomaly scores)
       - Visualize their feature distributions
       - Why did the model miss them?

    3. Score interpretability using the score vector:
       - For the top-10 anomalies, compute s(x) and identify
         which features have the largest score magnitude
       - These features are most "surprising" to the model

    Hint: The score vector provides per-feature anomaly attribution.
    """
    # ===== TODO =====
    pass
    # =====

```

3.8 Deployment Considerations

```
def optimize_for_inference(model, input_dim=64):
    """
    TODO: Optimize the model for deployment.

    Steps:
    1. Convert to TorchScript via torch.jit.trace
    2. Measure inference latency on CPU and GPU
    3. Verify that traced model produces same outputs

    Target: < 5ms per event on GPU, < 50ms on CPU

    Hint: Use torch.jit.trace(model, example_input)
    """
    # ===== TODO =====
    pass
    # =====
```

3.9 Ethics and Responsible AI

```
def ethical_considerations():
    """
    Discussion points for responsible deployment:

    1. False discovery risk: Publishing a physics anomaly based on a model's output
       requires rigorous statistical validation. A 5-sigma significance threshold
       is standard in particle physics. The model should be used as a FILTER,
       not as the final arbiter.

    2. Reproducibility: All training procedures, hyperparameters, and random seeds
       must be documented. Results should be reproducible by independent groups.

    3. Bias in training data: If the training set does not perfectly represent
       the true background, the model may systematically flag certain known
       physics processes as anomalous. Regular calibration against simulation
       is essential.

    4. Dual use: While designed for physics, the anomaly detection pipeline
       could be applied to surveillance or profiling. Clear use-case restrictions
       should be established.

    5. Environmental cost: Training on petabytes of data requires significant
       compute. The energy cost should be weighed against the scientific benefit.
    """
    pass
```

Section 4: Production and System Design Extension

System Architecture

The production anomaly detection system consists of three tiers:

Tier 1: Real-Time Trigger (Latency: $< 1\text{ms}$) - Lightweight rule-based filters applied at the hardware level - Reduces data rate from 40 MHz to $\sim 1\text{ kHz}$ - Not ML-based; uses physics-motivated thresholds

Tier 2: Online Anomaly Scoring (Latency: $< 10\text{ms}$) - The trained score network runs on dedicated GPU servers - Each event receives an anomaly score - Events above threshold are

flagged for offline analysis - TorchScript-optimized model deployed on NVIDIA Triton Inference Server

Tier 3: Offline Deep Analysis (Latency: hours) - Full score-based analysis with multiple noise levels - Langevin-based sample generation to characterize the learned background - Statistical significance testing against simulated backgrounds - Human physicist review of top anomaly candidates

API Design

```
POST /api/v1/score
Body: {"events": [[f1, f2, ..., f64], ...]}
Response: {"scores": [0.42, 1.87, ...], "flags": [false, true, ...]}

POST /api/v1/score/batch
Body: {"events_file": "s3://bucket/events.parquet"}
Response: {"job_id": "abc123", "status": "processing"}

GET /api/v1/score/batch/{job_id}
Response: {"status": "complete", "results_file": "s3://bucket/results.parquet"}

POST /api/v1/explain
Body: {"event": [f1, f2, ..., f64]}
Response: {"score": 1.87, "feature_attribution": {"MET": 0.45, "jet1_mass": 0.32, ...}}
```

Serving Infrastructure

- **Model Serving:** NVIDIA Triton Inference Server with TorchScript backend
- **Scaling:** Kubernetes cluster with GPU node pools, autoscaling based on event queue depth
- **Caching:** Redis cache for recently scored events (useful for re-analysis workflows)
- **Storage:** Apache Parquet on S3 for scored event archives; PostgreSQL for metadata

Monitoring

- **Model Performance:** Track anomaly score distribution over time. A shift in the distribution indicates either a model issue or genuinely new physics (both require investigation).
- **Latency:** P50 and P99 inference latency tracked via Prometheus/Grafana. Alert if P99 exceeds 10ms.
- **Data Quality:** Monitor input feature distributions for sensor failures or data pipeline issues.
- **Calibration:** Weekly comparison of anomaly score percentiles against Monte Carlo simulated backgrounds.

Drift Detection

- **Input Drift:** Kolmogorov-Smirnov test on each feature distribution, comparing the last hour against the training set baseline. Alert if any feature drifts beyond a configured threshold.
- **Output Drift:** Monitor the fraction of events flagged as anomalous. A sudden increase may indicate detector issues rather than new physics.

- **Model Staleness:** Track the score matching loss on a held-out validation set. If loss increases, the model may need retraining on updated background simulations.

A/B Testing

- **Shadow Deployment:** New model versions run in shadow mode, scoring the same events as the production model without affecting downstream decisions.
- **Canary Releases:** 5% of event traffic routed to new model; compare anomaly score distributions.
- **Metric Comparison:** Compare AUROC and signal efficiency on injected synthetic anomalies between model versions.

CI/CD Pipeline

1. **Training Pipeline (Monthly):**
2. New background simulations and calibration data ingested
3. Model retrained with multi-scale DSM
4. Automated evaluation on standard benchmark anomalies
5. Human review of anomaly candidates before deployment
6. **Deployment Pipeline:**
7. Model exported to TorchScript
8. Integration tests: verify API responses, latency, score consistency
9. Canary deployment to 5% of traffic
10. Full rollout after 24-hour monitoring period
11. **Rollback:** If anomaly flag rate changes by $> 3x$ within 1 hour of deployment, automatic rollback to previous model version.

Cost Estimation

- **Training:** 4x NVIDIA A100 GPUs for 8 hours = ~\\$100 per training run (monthly)
- **Inference:** 2x NVIDIA T4 GPUs continuously = ~\\$1,500/month
- **Storage:** ~10 TB of scored events per month = ~\\$230/month on S3
- **Total estimated cost:** ~\\$2,000/month
- **Cost per event scored:** ~\\$0.000002 (approximately 1 billion events scored per month)