# analyzing optimizer space

* Reconstruction MSE for different signal types
* Constraint violation rates
* Convergence stability (oscillations, plateaus)
* Gradient signal-to-noise ratio
* Weight sensitivity distributions

### **Phase 1: Foundation Analysis**

1. **Activation function comparison** - Start here as it's fundamental
2. **Basic optimizer comparison** - the existing SGD/Adam/etc. framework
3. **Learning rate sensitivity** - Critical for understanding convergence

Pick 3-4 activation functions (ReLU, Tanh, Linear, Sigmoid), **ReLU vs no activation**: exploring the "activation function dimension" of optimizer space

Train your autoencoder with each one using your existing code

Compare final reconstruction errors - pick the winner

Test 3-4 learning rates (0.0001, 0.001, 0.01, 0.1) with the best activation

Compare 2-3 optimizers (Adam, SGD, Momentum) with best activation + learning rate

### **Phase 2: Constraint Analysis**

1. **Implement lossless constraints** - This is unique to your application
2. **Measure constraint satisfaction rates**
3. **Analyze trade-offs between compression ratio and reconstruction quality**

### **Phase 3: Deep Landscape Analysis**

1. **Selective weight variation** - Understand which weights matter most
2. **Gradient noise analysis** - Critical for neural signal processing
3. **Multi-scale perturbation testing**

### Step 1: Use Best Weights

After training your model with a certain optimizer (like SGD or Adam), save the model’s weights at the point where it performs best on the validation data.

This gives you a “baseline” — the known good configuration to compare with.

### Step 2: Add Small Random Noise

Take those good weights and **add small random noise** to them (e.g., ±0.01 or ±0.001). Try this layer-by-layer.

* To see **how sensitive** each layer is to changes.
* If a small change drastically worsens performance, that layer is **very sensitive**.
* Robust models shouldn’t collapse from tiny noise.

**You can then compare:**

* “This layer dropped performance by 10%”
* “This one barely changed”

→ Tells you which layers matter most.

### Step 3: Identify Sensitive Weights

After doing noise tests on different parts (layers or parameter groups), measure how much performance drops.

* Sensitive weights are the ones optimizers need to treat carefully (smaller step sizes).
* Some optimizers (like Adam) are better at handling such cases.

🎯 **Result:** You get a **map of which weights are fragile** and influence overall performance the most.

### Step 4: Test Gradient Consistency

Take **two different mini-batches of data**, and compute the gradient (direction of steepest descent) for each.

Then, compare how similar the gradients are (using cosine similarity, for example).

A screenshot of a white background

AI-generated content may be incorrect.

* If gradients vary wildly between batches, your optimization landscape is **noisy or unstable**.
* Optimizers like Adam or RMSprop are designed to work better in noisy environments.

**Insight:** You understand how “reliable” the gradients are for training.

### Step 5: Map Landscape Roughness

Now, you test how the **loss changes when you move slightly** in random directions around your current weight values.

You perturb weights a bit in random directions and plot the loss. This gives you a **1D curve (or 2D surface)** of the loss landscape.

A screen shot of a graph

AI-generated content may be incorrect.

* If the loss increases smoothly → the landscape is **smooth** → SGD works well.
* If it jumps erratically → the landscape is **rough** → adaptive optimizers (Adam, RMSprop) help more.

📈 **Plotting the curve** helps you visualize if you’re in a sharp valley or a flat bowl.

### 🎯 Final Outcome:

After all of this, you’ll have a clearer sense of:

* Which weights are critical,
* How noisy your gradients are,
* What your local landscape looks like,
* And why a specific optimizer might work better than another.

### **Phase 4: Advanced Optimization**

1. **Implement cosine annealing** (easy win)
2. **Test simulated annealing** if needed
3. **Custom hybrid approaches** based on your findings

## more optimizers to add: Cosine Annealing (Easier to implement)

This would be particularly straightforward. Cosine annealing adjusts the learning rate during training using a cosine decay schedule:

% Add this to your training loop

initial\_lr = learning\_rate;

current\_lr = initial\_lr \* 0.5 \* (1 + cos(pi \* epoch / num\_epochs));

**Benefits for your autoencoder:**

* Helps escape local minima by periodically increasing the learning rate
* Often leads to better final convergence
* Works well with Adam, Momentum, etc. (you'd just modify their learning rates)
* Simple to implement with your existing optimizers

## Simulated Annealing (More complex but potentially powerful)

% Simulated annealing parameters

initial\_temp = 1.0;

cooling\_rate = 0.95;

temperature = initial\_temp;

% In your training loop, accept/reject weight updates based on temperature

if new\_loss < current\_loss || rand() < exp(-(new\_loss - current\_loss)/temperature)

% Accept the update

weights = new\_weights;

end

temperature = temperature \* cooling\_rate;

**Benefits for your signal processing application:**

* Excellent for avoiding local minima (crucial for neural signal reconstruction)
* Can help find globally better solutions
* Particularly good when the loss landscape is complex (common with autoencoders)

Evaluation

* Does the optimizer navigate the optimization space well?

### ✅ Signs of Good Navigation:

* **Loss decreases consistently** (no wild oscillations)
* **Training converges in reasonable steps**
* **Final solution has low loss and high task performance**
* **Trajectory follows smooth descent, not stuck early**
  + Does the optimizer converge (finds the true reconstruction of the signal)
    - plot loss vs Epoch
  + Does it find the global maximum/minimum or only the local one
* Why do we use gradient based optimization?
* Why are Neural networks not simple Optimizers
* Evaluate whether the Optimizer is good
  + Is the optimization space noisy
  + Does the optimizer converge (finds the true reconstruction of the signal)
  + Does it find the global maximum/minimum
  + use a SSIM metric in our loss function, build a feature extractor