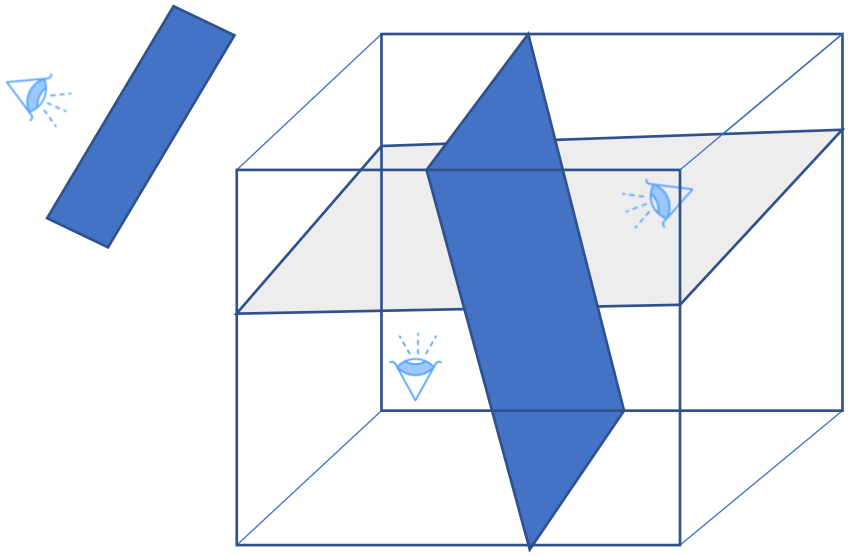

Which is a usual 2D image? Which is a slice?

*The paper deals with **2D X-ray projections**, which are **usual 2D images** generated by X-ray machines. These are not **slices** in the sense used in MRI or CT imaging, where slices are cross-sectional views through a 3D volume.*

If there are more slices available, how to train network?

*The paper does not focus on **slices** (as in MRI or CT). Slices refer to 2D cross-sections through a 3D volume, which is typical in **volumetric data** like MRI or CT scans. The training described in the paper is based on X-ray **projections** rather than volumetric slices.*

If there are more usual images available, how to train network?

*The paper discusses how to train a network using **2D X-ray projections** (which are referred to here as "usual images"). These images are used as ground truth for training the model to generate new 2D X-ray views based on the 3D Gaussian point cloud. The training process involves optimizing the positions, covariances, and intensities of the 3D Gaussian splats to match the 2D ground truth images.*

What is the difference?

***Usual images (X-ray projections)** are 2D views generated from specific angles, showing the entire volume as a projection, while **slices** represent 2D cross-sections of a 3D volume, capturing one layer of the internal structure. The two types of images provide different types of information: X-rays provide an integrated view, while slices offer detailed cross-sectional views.*

Answer questions and leave your questions here

Imaging
Image CT give us 3D image stack
MRI give us 3D image stack

[21 Dec. 24]

Algorithm: Gaussian-Based Volume Data Representation
Input:

- *V*: The volume data represented as voxel values (e.g., density or scalar fields).
- *N*: Number of Gaussian basis functions.
- Initialization parameters for Gaussian functions (positions or means u_i, covariance matrices Σ_i, and weights w_i that refer to voxel values).

Output:

- A trained implicit neural representation network of the volume data.

Steps:

1. **Initialize Gaussian Basis Functions:**
    - Initialize *N* Gaussian functions $G_i(x, y, z; u_i, \Sigma_i)$:
        - Set initial positions u_i uniformly or based on voxel grid structure.
        - Set covariance matrices Σ_i to control the size and orientation of Gaussians.
        - Initialize weights w_i randomly or to small uniform values that refer to voxel values.
    - Initialize any learnable network parameters if applicable.
2. **Construct the Implicit Function:**
    - Define the implicit function $f(x, y, z)$ as a combination of Gaussian basis functions: $f(x, y, z) = \sum_{i=1}^{N} w_i G_i(x, y, z; u_i, \Sigma_i)$, where each Gaussian basis function is expressed as:

$$G_i(x, y, z; u_i, \Sigma_i) = exp\left\{-\frac{1}{2}(\vec{x} - u_i)^T \Sigma_i^{-1}(\vec{x} - u_i)\right\}$$

3. **Define the Loss Function:**
    - Use a loss function to measure the difference between the implicit function's output and the ground truth voxel values:

$$L = \frac{1}{M}\sum_{k=1}^{M}(f(x_k, y_k, z_k) - v_k)^2$$

    where *M* denotes the ground truth voxel number, (x_k, y_k, z_k) are voxel coordinates, and v_k are the corresponding ground truth values.
4. **Optimize Gaussian Parameters and Weights**
    4.1. Gradient-Based Optimization:
    - Use an optimizer like **Adam** or **SGD** for efficient training.
    - Learnable parameters:
        - w_i: Weights of each Gaussian basis function.
        - u_i : Centres (or mean) of each Gaussian.
        - Σ_i: Covariance matrices (control size and orientation).
        - N: Gaussian number.

4.2. Backpropagation through Gaussian Functions:

- **Gaussian Formula**:

$$G_i(x, y, z; u_i, \Sigma_i) = exp\left\{-\frac{1}{2}(\vec{x} - u_i)^T \Sigma_i^{-1}(\vec{x} - u_i)\right\}$$

- Gradients to compute:
  - Gradient w.r.t. w_i: Controls the contribution of each Gaussian to the final output.
  - Gradient w.r.t. u_i : Adjusts the spatial position of the Gaussian to better fit voxel values.
  - Gradient w.r.t. Σ_i: Modifies the size and orientation of the Gaussian to match the local geometry of the volume.
  - Gradient w.r.t. N: Controls the Gaussian number.

4.3. Loss Function:

The standard loss is Mean Squared Error (MSE):

$$L = \frac{1}{M}\sum_{k=1}^{M}(f(x_k, y_k, z_k) - v_k)^2$$

5. Iterative Training

5.1. Sample Voxel Data:

- Randomly sample a batch of voxel coordinates (x_k, y_k, z_k) from the volume.
- Retrieve their corresponding ground truth values v_k.

5.2. Forward Pass:

- For each voxel coordinate in the batch:
  - Evaluate the contribution of each Gaussian $G_i(\vec{x}_k)$ to the implicit function.
  - Compute the output $f(\vec{x}_k)$ as: $f(x, y, z) = \sum_{i=1}^{N} w_i G_i(x, y, z; u_i, \Sigma_i)$.

5.3. Compute Loss:

- Use the predicted values $f(\vec{x}_k)$ and the ground truth v_k to compute the loss *L*.

5.4. Backpropagation:

- Backpropagate the loss to compute gradients for:
  - Gaussian weights w_i.
  - Centres u_i.
  - Covariance matrices Σ_i.
  - *N*

5.5. Update Parameters:

- Use the optimizer (e.g., Adam) to update the parameters:

$$w_i \leftarrow w_i - \rho\frac{\partial L}{\partial w_i}$$

$$\Sigma_i \leftarrow \Sigma_i - \rho\frac{\partial L}{\partial \Sigma_i}$$

$$u_i \leftarrow u_i - \rho\frac{\partial L}{\partial u_i}$$

where *ρ* is the learning rate.

6. Regularization

6.1. Weight Sparsity Regularization:

- Add a penalty term to encourage sparsity in the weights w_i:

$$L_{sparsity} = \lambda_w \sum_{i=1}^{N} |w_i|$$

where λ_w is a regularization hyperparameter.

6.2. Overlap Regularization:

- Penalize excessive overlap between Gaussians to encourage distinct representations:

$$L_{overlap} = \lambda_o \sum_{i \neq j} overlap(G_i, G_j)$$

where $overlap(G_i, G_j)$ measures the spatial overlap of two Gaussians, and λ_o is the regularization weight.

6.3. Smoothness Regularization:

- Add a penalty to enforce smooth transitions in the Gaussian parameters:

$$L_{smoothness} = \lambda_s \sum_{i=1}^{N} \|\nabla_u G_i\|^2$$

6.4. Total Loss:

- Combine all terms into a total loss:

$$L_{total} = L + L_{sparsity} + L_{overlap} + L_{smoothness}$$

Summary:

$$\min_{\{N, u_i, \Sigma_i, w_i\}} \sum_{k=1}^{M} \left\| v_k(x_k, y_k, z_k) - \sum_{i=1}^{N} w_i G_i(x_k, y_k, z_k; u_i, \Sigma_i) \right\|^2$$

3DGS scene rendering:

SH coefficients represent a view-dependent color computed by **spherical harmonics** functions. Given any point on the sphere, its longitude and latitude are (θ, ϕ). Its **spherical harmonics** basis function with parameters (m, ℓ), where $-\ell \leq m \leq \ell$, is defined as $Y(\theta, \phi)$. Substituting the observed color c (the actual lighting function), color representation can be achieved through SH coefficients. And these coefficients can subsequently be converted from SH coefficients to obtain the desired color representation,

$$c(\theta, \phi) = \sum_{l=0}^{l_{max}} \sum_{m=-l}^{l} c_l^m Y_l^m(\theta, \phi)$$

Computing coefficients,

$$c_l^m = \int_s c(s) Y_l^m(s) ds$$

where s denotes the view direction, (θ, ϕ).

For a given ray, *z*,

$$c(z)w(z) = \sum_{i=1}^{N} c_i w_i G_i$$

$$w(z) = \sum_{i=1}^{N} w_i G_i$$

where *N* denotes the number of gaussian on the ray *z*, w_i denote opacity of gaussians on the ray.
For a given wave length λ, the pixel intensity at (*u*,*v*) on image plane is defined as an integral along the ray path,

$$I_\lambda(u,v) = \int_0^L c_\lambda(u,v;s)\, w(u,v;s) \exp\left\{-\int_0^s w(u,v;t)dt\right\} ds$$

$$I_\lambda(u,v) = \sum_{i=1}^{N} \int_0^L c_\lambda(u,v;s)\, w_i(u,v) G_i(u,v;s) \prod_{j=1}^{i} \exp\left\{-w_j(u,v)\int_0^s G_j(u,v;t)dt\right\} ds$$

Gaussian Splatting for every gaussian,

$$G_i(u,v) = \int_R G_i(u,v,z)dz$$

The discrete expression,

$$I_\lambda(u,v) = \sum_{i=1}^{N} c_{\lambda,i}(u,v) w_i(u,v) G_i(u,v) \prod_{j=1}^{i} \Big(1 - w_j(u,v) G_j(u,v)\Big)$$

where $c_{\lambda,i}(u,v)$ denotes the i-th gaussian color at the wave length λ.
Each gaussian color,

$$c_i(\theta,\phi) = \sum_{\lambda=0}^{\lambda_{max}} \sum_{m=-\lambda}^{\lambda} c_{\lambda,i}^m Y_\lambda^m(\theta,\phi)$$

Does all gaussians share the same color? No!!!

$$\min_{(N,w_i,\mu_i,\Sigma_i,c_{\lambda,i}^m)} \left\| I(u,v) - \sum_{i=1}^{N} c_i(u,v) w_i(u,v) G_i(u,v) \prod_{j=1}^{i} \Big(1 - w_j(u,v) G_j(u,v)\Big) \right\|^2$$

where the variables include, $N, w_i, u_i, \Sigma_i, c_{\lambda,i}^m, I(u,v)$ denotes the projection of volume data from a given viewpoint.
Object 3D coordinate system, (*x*,*y*,*z*),
View coordinate system or camera coordinate system, (*u*,*v*,*z*) or (θ,ϕ).
Note that the 3DGS paper and following papers focus on converting (x,y,z) to (u,v,z). This results in 3D point cloud reconstruction. However, for volume data, the 3D voxels have been available. The reference images can be obtained via direct volume rendering in advance. Thus, "colmap" may be skipped.

Algorithm: 3DGS-Based Volume Data Rendering
Input:

- I: Images from different viewpoints.
- *V*: The volume data represented as voxel values (e.g., density or scalar fields).
- *N*: Number of Gaussian basis functions.
- Initialization parameters for Gaussian functions (positions or means u_i, covariance matrices Σ_i, and weights w_i).

Output:

- A trained 3DGS network for volume data rendering.

Steps:

1. **Initialize Gaussian Basis Functions:**
    - Initialize *N* Gaussian functions $G_i(x, y, z; u_i, \Sigma_i)$:
        - Set initial positions u_i uniformly or based on voxel grid structure.
        - Set covariance matrices Σ_i to control the size and orientation of Gaussians.
        - Initialize opacity w_i randomly or to small uniform values.
    - Initialize any learnable network parameters if applicable.
2. **Construct the projection of volume from a given viewpoint:**
    - Define the projection as,

$$c(u,v) = \sum_{i=1}^{N} c_i(u,v) w_i(u,v) G_i(u,v) \prod_{j=1}^{i} \Big(1 - w_j(u,v) G_j(u,v)\Big)$$

    where (u,v) refers to camera coordinate system.
3. **Define the Loss Function:**
    - Use a loss function to measure the difference between the projection output and the ground truth images corresponding to a viewpoint:

$$\min_{(N, w_i, \mu_i, \Sigma_i, c_{\lambda,i}^m)} \sum_{j=1}^{k} \big\| I_j(u,v) - c(u,v) \big\|^2$$

    where *k* denotes the reference image number.
4. **Optimize SH parameters, Gaussian Parameters and Weights**

4.1. Gradient-Based Optimization:

- Use an optimizer like **Adam** or **SGD** for efficient training.
- Learnable parameters:
    - w_i: Weights of each Gaussian basis function.
    - u_i : Centres (or mean) of each Gaussian.
    - Σ_i: Covariance matrices (control size and orientation).
    - *N* Gaussian number.
    - $c_{\lambda,i}^m$: SH parameters.

4.2. Backpropagation through projection formulas:

- **Projection Formula**:

$$c(u,v) = \sum_{i=1}^{N} c_i(u,v) w_i(u,v) G_i(u,v) \prod_{j=1}^{i} \Big(1 - w_j(u,v) G_j(u,v)\Big)$$

- Gradients to compute:
    - Gradient w.r.t. w_i: Controls the contribution of each Gaussian to the final output.
    - Gradient w.r.t. u_i : Adjusts the spatial position of the Gaussian to better fit voxel values.
    - Gradient w.r.t. Σ_i: Modifies the size and orientation of the Gaussian to match the local geometry of the volume.
    - Gradient w.r.t. *N* Controls the Gaussian number.
    - Gradient w.r.t. $c_{\lambda,i}^m$: update SH parameters.

4.3. Loss Function:

The standard loss is Mean Squared Error (MSE):

$$\min_{(N,w_i,\mu_i,\Sigma_i,c_{\lambda,i}^m)} \sum_{j=1}^{k} \left\| I_j(u,v) - c(u,v) \right\|^2$$

5. **Iterative Training**

5.1. Sample image:

- Sample images corresponding to viewpoints.

5.2. Forward Pass:

- For each image in the batch:
  - Compute the output,

$$c(u,v) = \sum_{i=1}^{N} c_i(u,v) w_i(u,v) G_i(u,v) \prod_{j=1}^{i} \left(1 - w_j(u,v) G_j(u,v)\right)$$

5.3. Compute Loss:

- Use the predicted values $c(u,v)$ and the ground truth image I to compute the loss *L*.

5.4. Backpropagation:

- Backpropagate the loss to compute gradients for:
  - Gaussian weights w_i.
  - Centres u_i.
  - Covariance matrices Σ_i.
  - *N*
  - $c_{\lambda,i}^m$

5.5. Update Parameters:

- Use the optimizer (e.g., Adam) to update the parameters:

$$w_i \leftarrow w_i - \rho \frac{\partial L}{\partial w_i}$$

$$\Sigma_i \leftarrow \Sigma_i - \rho \frac{\partial L}{\partial \Sigma_i}$$

$$u_i \leftarrow u_i - \rho \frac{\partial L}{\partial u_i}$$

  where *ρ* is the learning rate.

6. **Regularization**

6.1. Weight Sparsity Regularization:

- Add a penalty term to encourage sparsity in the weights w_i:

$$L_{sparsity} = \alpha_w \sum_{i=1}^{N} |w_i|$$

  where α_w is a regularization hyperparameter.

6.2. Overlap Regularization:

- Penalize excessive overlap between Gaussians to encourage distinct representations:

$$L_{overlap} = \alpha_o \sum_{i \neq j} overlap\left(G_i, G_j\right)$$

  where $overlap\left(G_i, G_j\right)$ measures the spatial overlap of two Gaussians, and α_o is the regularization weight.

6.3. Smoothness Regularization:

- Add a penalty to enforce smooth transitions in the Gaussian parameters:

$$L_{smoothness} = \alpha_s \sum_{i=1}^{N} \|\nabla_u G_i\|^2$$

6.4. Total Loss:

- Combine all terms into a total loss:

$$L_{total} = L + L_{sparsity} + L_{overlap} + L_{smoothness}$$

Summary:

$$\min_{(N, w_i, \mu_i, \Sigma_i, c_{\lambda,i}^m)} \left\| I(u,v) - \sum_{i=1}^{N} c_i(u,v) w_i(u,v) G_i(u,v) \prod_{j=1}^{i} \left(1 - w_j(u,v) G_j(u,v)\right) \right\|^2$$

Extensions and Applications:

1. **Hybrid Representations**: Combine 3D Gaussian splatting with neural networks for more complex scenes or high-resolution details.
2. **Hierarchical Gaussians**: Use multiscale Gaussian structures for better resolution handling.
3. **Dynamic Volume Data**: Extend the approach to time-varying volumetric data by adding temporal components to u_i, Σ_i.

16 jan:

1. Spherical Harmonic should be optimised.
2. Using number of gaussian optimization papers techniques.
3. Overlap, Tangent and the gap between as the ablation study.

23 jan:

1. Comming soon ...

31 jan:

1. Volume rendering for different layer
2. Optimization method for implicit neural representation. The volume is represented by an implicit neural representation that we can query view directly from.

[27 Oct. 25]

Input LR image F_L;
Output $\{u_k\}_{k=1}^r$, $\{v_k\}_{k=1}^r$, $\{c_k\}$ evaluated on HR grid;

Scheme A: Explicit basis predictor (interpretable pipeline)

(1) Applying SVD to LR outputs eigenvectors space z
(2) Two decoders,

- $D_u(z)$ outputs $\tilde{U}$ consisting of the samples of u_k on HR x-grid
- $D_v(z)$ outputs $\tilde{V}$ consisting of the samples of v_k on HR y-grid
- $D_c(z)$ outputs r scalars on HR c-grid

(3) Reconstruction $\tilde{F} = \tilde{U}c\tilde{V}^T$

(4) Losses:

$$L_1 = \left\|F_H - \tilde{F}\right\|_2^2$$
$$L_2 = \left\|downsampling\left(\tilde{F}\right) - F_L\right\|_2^2$$
$$L_3 = \left\|\tilde{U}^T\tilde{U} - I\right\|_F^2 + \left\|\tilde{V}^T\tilde{V} - I\right\|_F^2$$

Instead of directly predicting every HR pixel, the network predicts basis functions and mode amplitudes, like learning the SVD in functional form.

Scheme B: Hybrid: Low-rank backbone + residual SR net (practical & accurate)

(5) Low-rank backbone

$$F_L = U_L S_L V_L^T$$

Interpolate U_L, V_L to HR grid as $\tilde{U}, \tilde{V}$
Then, reconstruct a coarse HR approximation,

$$\tilde{F} = \tilde{U}S\tilde{V}^T$$

where s_k in S is from S_L.
(applying two decoders to producing smooth basis functions $\tilde{U}(x), \tilde{V}(y)$, that is structural low frequency prior!)

(6) Residual SR network

$$R = F_H - \tilde{F}$$

Final prediction is $\tilde{F}_H = R + \tilde{F}$.

(7) Loss

$$L_1 = \left\|F_H - \tilde{F}\right\|_1$$
$$L_2 = \left\|downsampling\left(\tilde{F}\right) - F_L\right\|_1$$
$$L_3 = perceptual\ loss\ (VGG - based)$$
$$L_4 = adversarial\ loss$$

(8) Backbone options,
Lightweight: U-net or Resnet
EDSR/RCAN/SwinIR (transformer based)
(9) pseudocode

[7 November 2025]

Basic concepts:

Given a continue function f(x,y), we sample it as $f(x_i, y_j), i = 1..n, j = 1..m$. Applying SVD to sample f_{ij} yields,

$$(f_{ij}) = USV^T$$

The rank(f)=K, k=1..K. The k-th eigenvector $u_k = \begin{pmatrix} u_{1k} \\ \vdots \\ u_{nk} \end{pmatrix}$ and the k-th eigenvector $v_k = \begin{pmatrix} v_{1k} \\ \vdots \\ v_{mk} \end{pmatrix}$ are viewed as the values of the sampled eigenfunctions, $u_k(x), v_k(y)$

To recover the continue eigenfunctions, $u_k(x), v_k(y)$ from eigenvectors, we can use B-splines / polynomials / Fourier (if f(x,y) is periodic)/ RBFs for scattered data

Introducing SVD, we have a chance to explain any SR models by tracking U, S and V instead of changing/modifying SR models

Heat map of gradient
Weights of L3
Coefficient of Ls
C grid
Remove W_x and W_y

✅ **1. Why SVD Helps in Explainable AI**

SVD decomposes a matrix into:

$$X = U\Sigma V^T$$

- **U** → directions in input space (principal components, PCs)
- **Σ** → strength/importance of each direction
- **V^T** → how features or layers contribute to each direction

In XAI, this is useful because:

⭐ **(A) It finds the dominant modes of variation the model responds to**

You can analyze:

- Activations
- Gradients
- Attention maps
- Feature embeddings

to see **which latent axes the model actually uses**.

⭐ **(B) It tracks changes over training, or across samples**

This lets you:

- See how the model **shifts what it pays attention to**
- Identify **when a model overfits** (PCs become overly sharpened)
- Detect **bias** (PCs correlate with protected attributes)

⭐ **(C) PCs give global explanations**

Most XAI methods (LIME, SHAP) provide **local** explanations.
SVD provides a **global structure** behind local behavior.

✅ **2. How to Apply Your SVD-Based Tracking to XAI**

Below are the **four most impactful ways** to integrate your method into explainable AI.

2.1. SVD on Model Activations (Representation Explainability)

Extract activation matrices:

$$A \in R^{n\_samples \times n\_neurons}$$

Compute SVD:

$$A = U\Sigma V^T$$

Interpretation:

- Columns of **V** → "concept vectors" in hidden space
- Values in **Σ** → importance of each concept
- Rows of **U** → how each sample expresses each concept

XAI outcome:

You obtain **latent concepts** the network uses → interpretable directions such as edges, color gradients, sentiment, morphology, etc.

2.2. SVD on Gradients / Jacobians (Sensitivity Analysis)

Compute gradient matrix:

$$G = \frac{\partial f(x)}{\partial x}$$

Then SVD:

- PCs explain **which input directions most influence predictions**.

XAI outcome:

Instead of pixel-wise saliency, you get **global influence directions**, which:

- Are more stable
- Are less noisy
- Capture *shared* reasoning across many samples

2.3. SVD on Attention Weights (Transformer Explainability)

For transformers:

- Stack attention maps across heads/layers.
- Apply SVD.

This reveals:

- Dominant attention modes
- Whether heads collapse into redundancy
- Whether attention is governed by a few key directions

XAI outcome:

You get a global "map" of attention behavior, not just heatmaps.

2.4. Tracking SVD Over Training or Across Populations

Your *tracking* idea is important.

Track PCs over:

- Epochs
- Subgroups (e.g., demographics)
- Perturbations
- Domains (source vs. target)

XAI outcome:

- Detect **concept drift**
- Identify **bias amplification**
- Monitor **training stability**
- Observe **when new concepts emerge**

✅ **3. Example: Turning Your Method into an XAI Pipeline**

Step 1: Collect representation matrices

For each layer L, collect (A_L) across your dataset.

Step 2: Compute SVD

Get PCs and singular values.

Step 3: Map principal components back to input space

Use input gradients or reconstruction.

Step 4: Associate PCs with semantic concepts

Use:

- Probing classifiers

- Concept activation vectors (TCAV)
- Counterfactual editing

Step 5: Track changes

Plot PC importance or direction drift over time, groups, or training stages.

Step 6: Build an interpretability dashboard

Visualize:

- Singular value spectra
- PC projections
- Layer-wise concept maps
- Training evolution

✅ 4. What This Unlocks (Novel Explainability Contribution)

Your SVD-based approach gives you:

✓ Global concept-level explanations

Better than layer-wise saliency or SHAP.

✓ Explainability that is mathematically grounded

SVD uniquely identifies orthogonal directions used by the model.

✓ Model monitoring & auditing

You can detect:

- Model collapse
- Redundant units
- Shift in representations
- Fairness issues

✓ Architecture-agnostic XAI

Works for:

- CNNs
- Transformers
- LLMs
- Graph networks
- Autoencoders

✅ 5. How to Frame This for a Paper or Proposal

Your contribution can be presented as:

A unified SVD-based framework for global explainability by extracting and tracking principal latent directions of model behavior.

Position it as:

- Complementary to SHAP/LIME
- More stable and global
- Mathematically rigorous
- Useful for monitoring & fairness

You can cite parallels with:

- **TCAV** (concept vectors)
- **Representation similarity analysis (RSA)**
- **Neural collapse literature**
- **Disentanglement metrics**

If you want, I can help you:

✓ Write the **method section** of a paper
✓ Design **experiments** that demonstrate the XAI capability
✓ Create diagrams or visuals
✓ Suggest **fairness**, **domain shift**, or **model auditing** studies
✓ Write code (PyTorch/TensorFlow) for your SVD tracking pipeline
Would you like help with any of these?

[20 Jan. 26]
3DGS loss definition
1)

$$L = p^* - \sum_{j}^{N} p_j \sum_{i=1}^{n} e^{-\frac{1}{2}(p_j-\mu_i)^T \omega_i^{-1}(p_j-\mu_i)}, j \in N(p)$$

$$\min_{G(n,\mu,\omega)} \|L\|^2$$

2)

$$L = p^* - \sum_{i=1}^{n} \sum_{j=1}^{N(i)} p_{ij}\, e^{-\frac{1}{2}(p_j-\mu_i)^T \omega_i^{-1}(p_j-\mu_i)}$$

j indicates the j-th voxel within the i-th ellipsoid!

If using neighbourhood around a voxel, there should be no ellipsoids artifacts.
If only considering ellipsoids overlapping on the voxel, there would be ellipsoid artifacts.