Final Project

ME EN 7960

Scientific Machine Learning

Khalid Zobaid Adnan

u1447873

Project title:

Singular value decomposition and neural auto-encoded Dynamic mode decomposition technique to extract and visualize dominant phonon modes and predict positions and forces from MD simulations.

Introduction:

Evaluating the vibrational eigenmodes from molecular dynamics (MD) is of extreme importance in many areas of research, e.g., nano-scale thermal transport[1–3]. Phonon is a quantized lattice vibration, which can be resolved in the frequency or time domain. Heat conduction in materials depends on the phonon lifetime, phonon mean free path, and specific heat of each phonon mode. Understanding the spectral contribution to material properties is crucial in deepening the dominant features from a thermo-material science perspective[4,5]. Recent works have greatly advanced application-based design and understanding of device failure in modern electronics and thermal management. However, we are still limited to some of these methods due to several reasons. (1) Most of these spectral methods do not have any visual aid to properly understand the vibrational properties. (2) Some of these methods are derived from frequency or time-resolved methods, which cannot be easily implemented or replicated.

The key challenges are (1) creating a timestep matrix representation using code, (2) the accuracy of the MD simulations, which heavily rely on the interatomic potential employed. Thus, we have trained a deep learning interatomic potential to simulate the MD. After we are done, we can simply visualize our results but in terms of different dominating modes. SVD, DMD and autoencoder DMD will be utilized to see the accuracy in the prediction of positions as well as the forces in each

timestep. Thus, we will solve an issue of interpretability in MD simulations especially in the vibrational scheme of the atoms.

In this work, we have reported the utilization of singular value decomposition (SVD) and the with or without autoencoder dynamic mode decomposition (DMD) to find out the dominating modes of the atomic vibrations from molecular dynamics (MD) simulations. The rank of the data matrix presented in this work confirms the 3N eigenmodes of the simulation. Based on the rank of the matrix, we have utilized: (1) DMD to extrapolate the simulation data and (2) an autoencoder to work in the latent space to accelerate the predictions. Utilizing the DMD can accelerate the future of predicting thermal properties using MD simulations. The results from this investigation leads to a clear understanding of the compressed representation of the MD simulation data. Utilizing only a few dominating modes can predict the presentation of the final structure of the system.

Methods:

Singular value decomposition (SVD)[6,7] is clearly the single most important linear algebra method to reconstruct data consisting of three singular matrices, e.g., U, V^T and $\sum$, where U, V^T are left and right singular vectors and $\sum$ are the singular values.

$$A = U\sum V^T \tag{1}$$

The most dominant singular value is the first singular value in that vector, followed by other singular values in descending order. SVD is the most efficient in the representation of the data in terms of the dominant feature in the data. To do the analysis, we need to form a matrix format that can be represented as A with $n * m$ dimensions, where n is the size of the dataset and m is the

number of timesteps. For our MD simulations, n is the total number of configurations consisting of atoms, atom-ids, and coordinates of the atoms in one timestep, and m is the different timeframes.

To interpret the modes, we can use the dynamic mode decomposition (DMD)[8–11] , which is also a data-driven technique to obtain the dominant features in the data. Compared to SVD or PCA, where we take the mean from the timestep matrix, the governing equation has to be projected, which is very difficult to implement. Since the DMD heavily relies on the representation of the data in terms of a linear combination of dominant modes, it can be easily understood, and the eigenmodes of the vibrations can be visualized. Much work has been done in the fluid mechanics area to improve the DMD approach to interpret complex fluid behavior. The DMD relies on solving a pseudo-linear fit to the $X_2=AX_1$ equation. The equation comes from the very first dynamical system equation.

$$\frac{dx}{dt} = \dot{x} = f(x) \tag{2}$$

The finite difference representation with $t_{k+1} - \mathrm{t_k} = \Delta t$ will be,

$$x_{k+1} = F(x_k) \tag{3}$$

Our goal is to find a matrix (operator) A Such that we can easily write our data as a simple linear dynamical system.

$$x_{k+1} \sim A x_k \tag{4}$$

Which is equivalent to $x' = Ax$

From the matrix solution, we can represent A as

$$A = X' \tilde{V} \widetilde{\sum} U^*$$

The Autoencoder-DMD combines an autoencoder (a neural network for dimensionality reduction) with DMD to model and reconstruct the dynamics of a high-dimensional molecular dynamics (MD) dataset. The procedure assumes the input is a LAMMPS trajectory file and produces a reconstructed trajectory with minimized error, addressing frozen atoms, and comparing with direct DMD. This procedure leverages the strengths of autoencoders (nonlinear dimensionality reduction) and DMD (dynamic modeling) to reconstruct your MD trajectory with high fidelity.

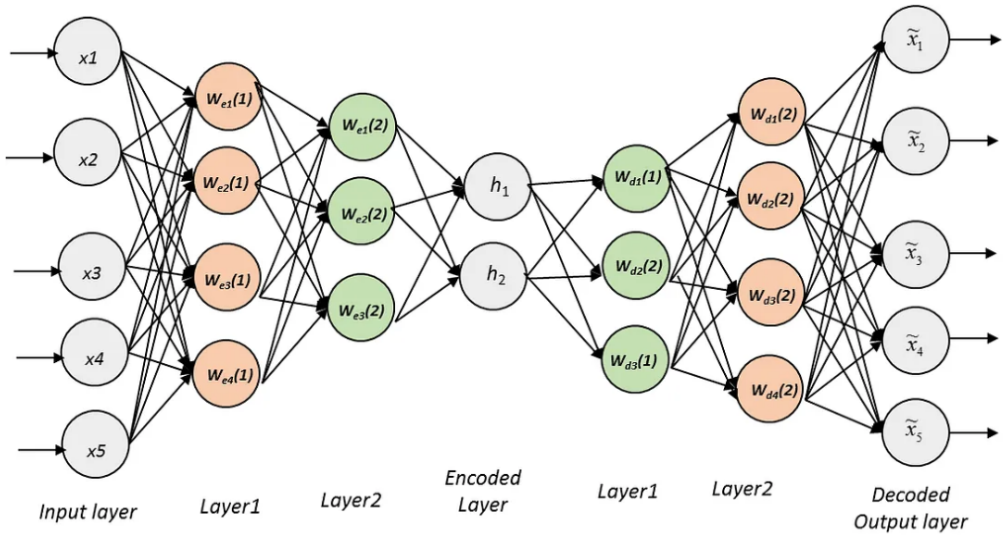


Figure: Autoencoder neural network

In our work, we have used a 256,128, encoded layer,128,256 network. We varied the encoded layer to see whether it can improve the prediction capability of the autoencoder.

Results:

Reconstruction from (SVD):

Figure 1 shows the actual atomic vibration of $Al_{1-x}Ga_xN/AlN$ from MD simulations. If we can represent this atomic vibration in a timestep matrix format A with $n * m$ dimensions. For our MD simulations, n is 8320 and m is 4273. The total number of atoms is 1664 for the system. With a total degree of freedom to be 4992. For our case, the frozen number of atoms is 64 + 62 =126, as shown in Figure 1 on the right and left sides. Since these atoms do not move, there are 3N eigenmodes of Phonons. The number of eigenvalues is then 1664-126= 1538, and the freedom of degree is 1538 *3= 4614.

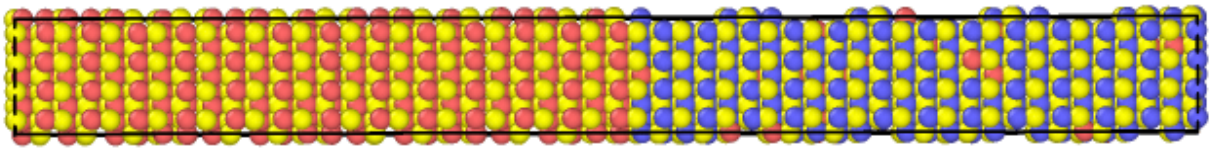


Figure 1: The actual atomic vibration of atoms in $Al_{1-x}Ga_xN/AlN$

The SVD rank of the timestep matrix is 4014. A natural question is whether the rank can be less than the total degree of freedom. Another question is whether including more timesteps can increase the rank of the matrix. To address this question, we have plotted the impact of a number of timesteps on the SVD rank for MD and non-equilibrium MD (NEMD). As shown in FIG.2, the calculated rank of the timestep matrix A Converges to 4614, which is represented by the total degrees of freedom in the MD system. One important thing to note is that since equilibrium MD (EMD) does not need a higher number of timesteps to extract the 3N phonon modes. However, for nonequilibrium MD (NEMD), we need more timesteps to converge to the dominant 3N modes.

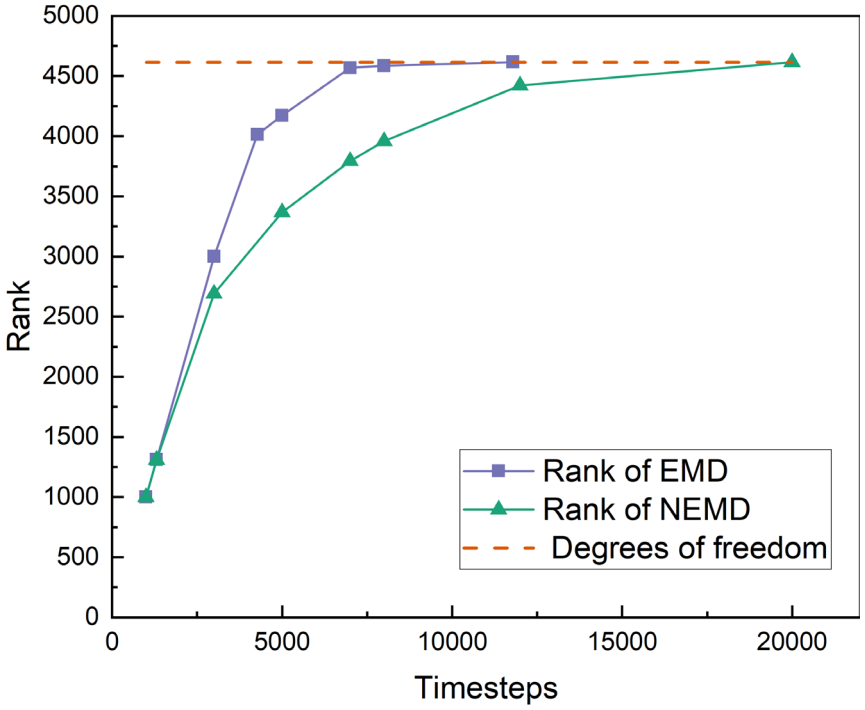


Figure 2: Impact of timesteps on the evaluated rank of the timestep matrix A

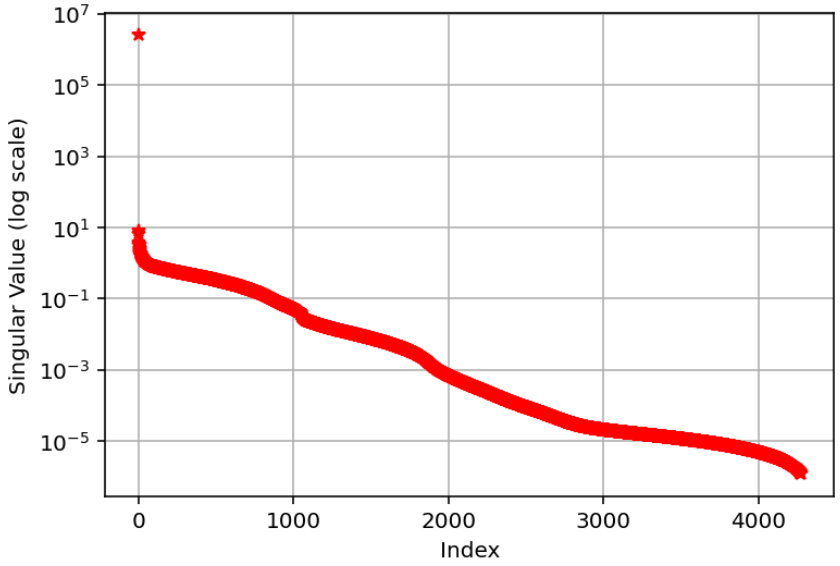
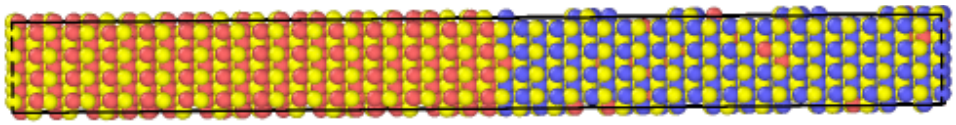


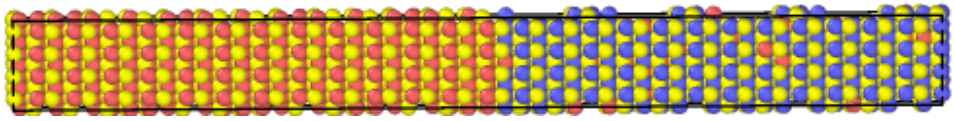
Figure 3: Singular values of the timestep matrix from MD simulations of $Al_{1-x}Ga_xN/AlN$ heterostructure.

After we are done calculating the rank of the timestep matrix A. We want to investigate how the singular value, $\sum$ changes. Figure 3 shows the singular values for MD simulations of a heterostructure consisting of AlN and $Al_{1-x}Ga_xN$. As shown, the most dominant singular values are the first few values. Thus, we can reconstruct the timestep matrix data from the first few modes of the singular values. Note that we have plotted the data up to the rank of our timestep matrix of MD simulations. If we want to reconstruct the data, we can include more modes to represent the actual ground truth of the data. For example, we can construct data by taking only 10,20, 50, 100 % of the rank. Figure 3 shows the reconstructed atomic vibrations. Using 100 percent of the rank provides us with the best data reconstruction using SVD.

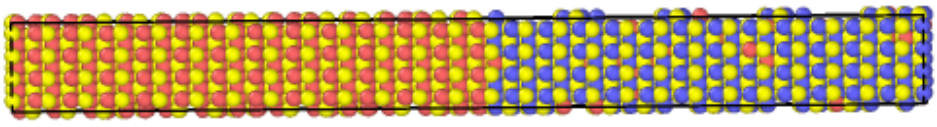
10%



20%



50%



100%

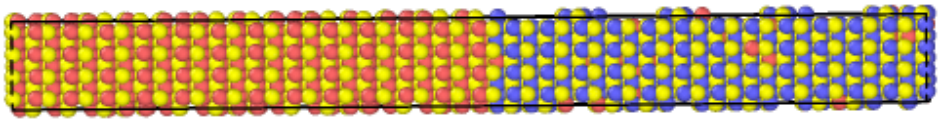


Figure 3: Reconstruction of timestep data from SVD rank as a function of percentage of rank utilized

To interpret and go forward with the timestep, we need to implement DMD with or without the neural autoencoder. Since we have done SVD on the timestep matrix. We have the U , V^T And the singular values. Based on this, we can extend our investigation into implementing the DMD. In DMD, we can utilize the singular values obtained from the SVD.

Reconstruction from DMD:

To illustrate the impact of adding forces to the MD simulation data, we have first calculated the SVD of the updated timestep matrix A. The rank of the matrix should be 9984 since we have added three more force components per atom. Our results show that the rank of the matrix is 9706, which should converge to 9984 if there is sufficient timestep data, as discussed in Figure 2. As shown in Figures 4 & 5, we have included the force and positions of the atoms from DMD calculations. Unlike SVD, DMD suffers from predicting outside the existing data, as shown in Figure 6. Note that for all the cases, we have only utilized 675 modes.

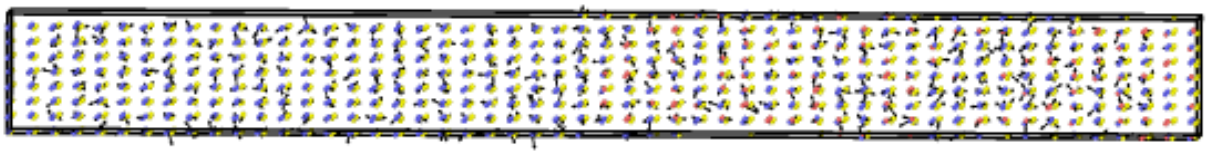


Figure 4: SVD reconstruction from force and position data

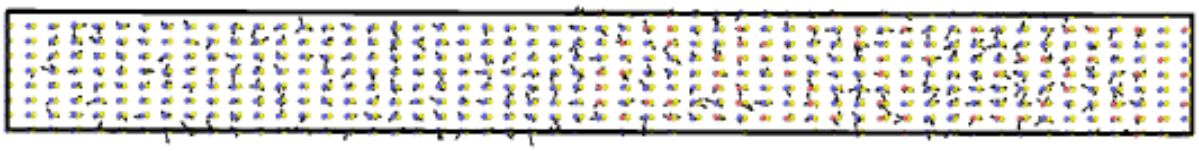


Figure 5: DMD reconstruction from force and position data

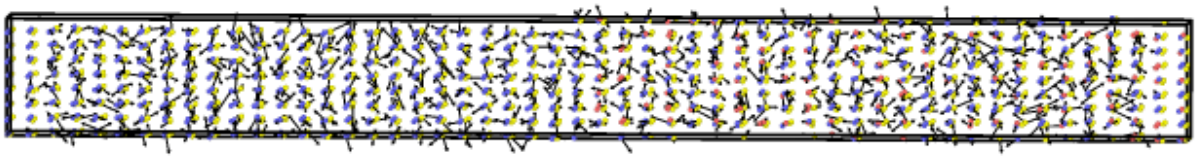


Figure 6: DMD prediction for timesteps not included in MD data

To overcome this limitation, we can (1) increase the number of timesteps in the MD data, (2) utilize physics-informed DMD to make sure there is no inconsistency in the forces, which can be a future project. Based on the results, we can conclude that compressed representation of the data is possible for the given data matrix. However, we might not be able to predict accurate data outside the given data, considering that we have some reduced-order modeling (ROM) scheme with SVD using intrusive ROM or Physics-informed DMD.

To illustrate the impact of different modes, we can visualize the data using only the first few modes to understand the underlying physics. The first mode represents the equilibrium position of the atoms as shown in Figure 7. If we add another mode, we can see all the forces acting on the atoms that decay over time. We think this is due to the initial velocity distribution applied during the MD simulation, as shown in Figure 8. For other numbers of modes (5), the forces are increasing or

decreasing and rotating around a fixed axis. The decaying part is slowing down the growth of the force vectors as shown in Figure 9.

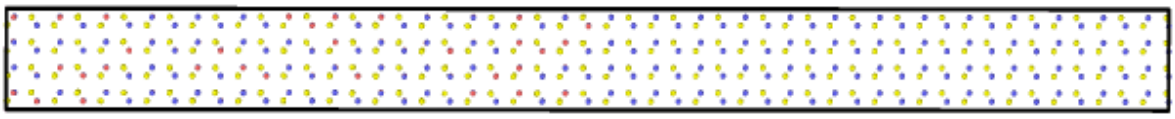


Figure 7: DMD (1 mode) reconstruction of the MD data

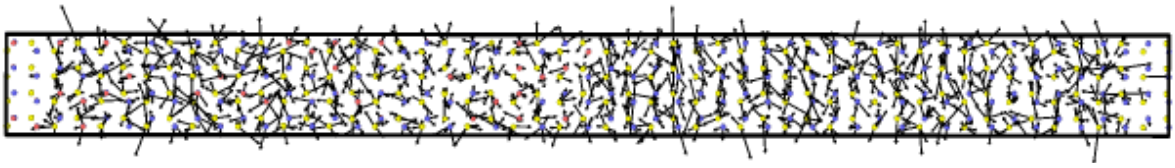


Figure 8: DMD (2 modes) reconstruction of the MD data

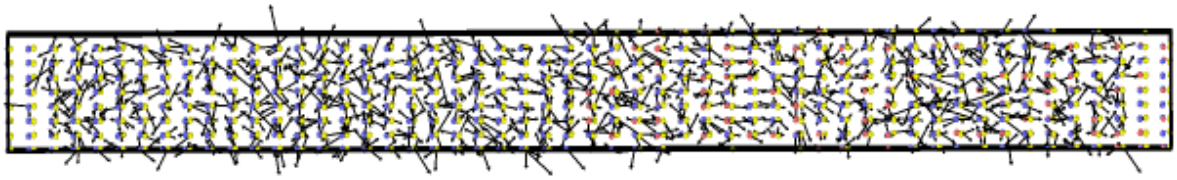


Figure 9: DMD (5 modes) reconstruction of the MD data

Reconstruction from neural Auto-encoded DMD:

To combine a traditional learning scheme with a modern neural network, we have included an autoencoder DMD. The autoencoder DMD learns through a non-linear compressed latent space compared to linear compression in DMD. Figure 10 shows the autoencoder DMD results. From Figure 10b shows the normalized mean squared error per timestep based on the latent 32 encoder layer representation. Similar representations are shown in Figure 10c-e. The training and validation errors are plotted against a number of epochs. All these figures represent a learning rate of 0.0001. At a higher learning rate of 0.01, the training errors shoot up during the training procedure as shown in Figure 11. Thus, we should pick a learning rate between.

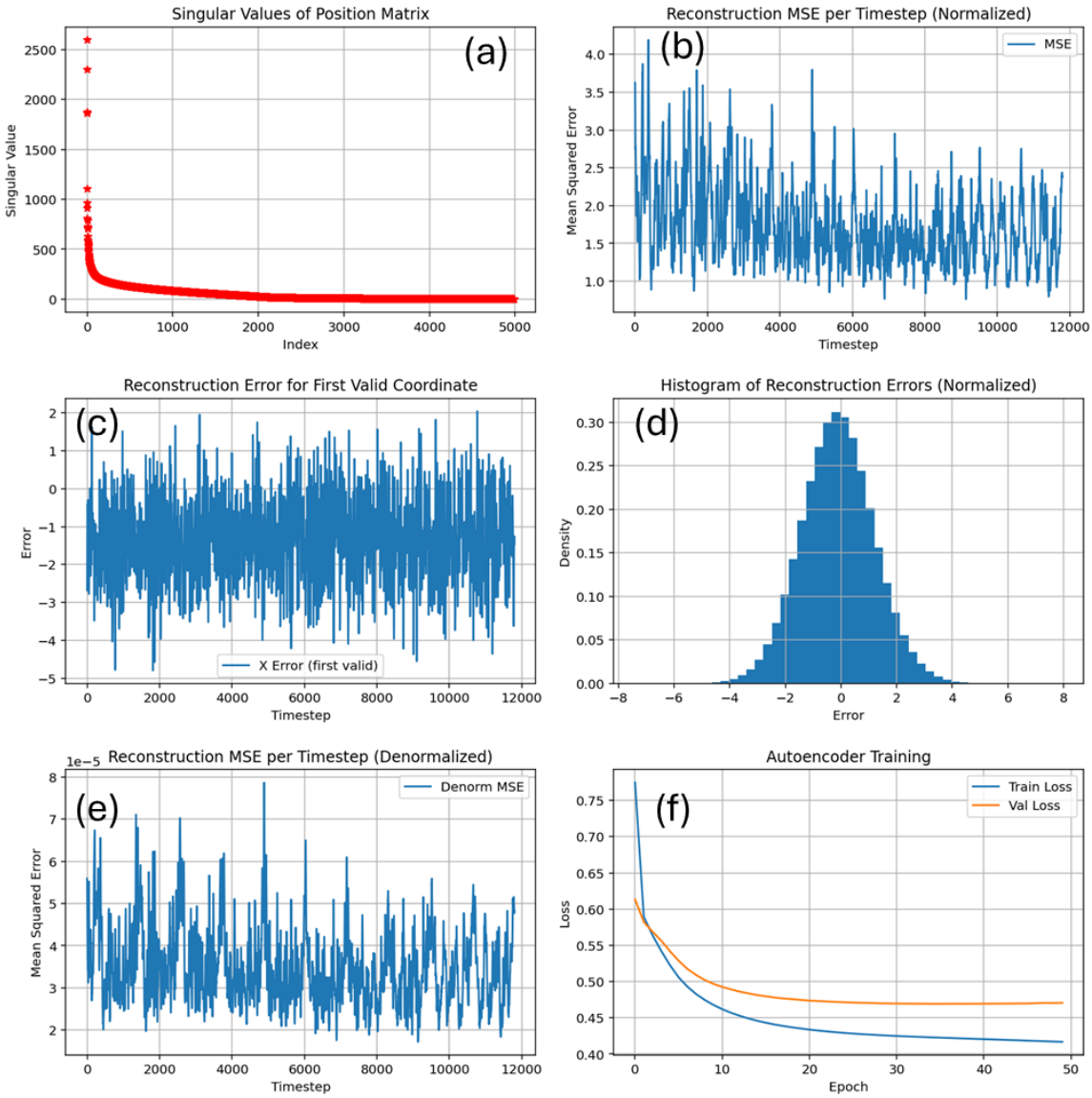


Figure 10: For a learning rate of 0.0001(a) Singular values (b) MSE for each timestep (c) X coordinate error normal to heat flux direction, (d) Histogram of normalized reconstruction error, (e) MSE for denormalized in each timestep (f) training and validation loss as a function of epoch.

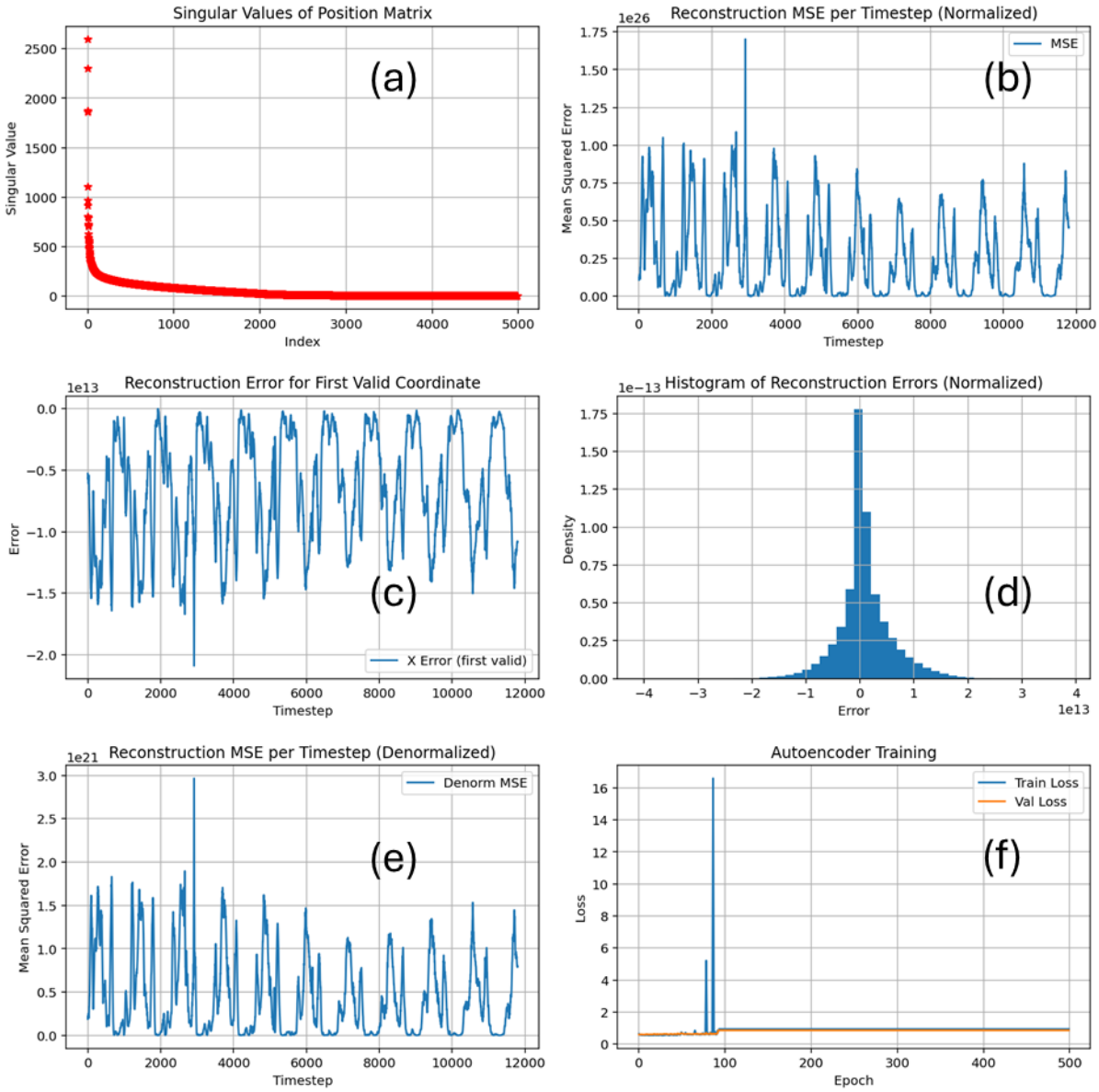


Figure 11: For a learning rate of 0.01(a) Singular values (b) MSE for each timestep (c) X coordinate error normal to heat flux direction, (d) Histogram of normalized reconstruction error, (e) MSE for denormalized in each timestep (f) training and validation loss as a function of epoch.

The impact of the size of the latent space is a crucial factor in determining the training accuracy. Figure 13 shows the results when the encoded layer is 64. If we decrease the encoder size, the validation error increases after some epoch steps. This is reasonable since it does not have enough compressed representation using an encoder size of 12. Based on the results, we can see that DMD results are still better than the autoencoder-DMD results. The reason might be that the latent representation is not yet fully discovered by the encoded layer of size 64.

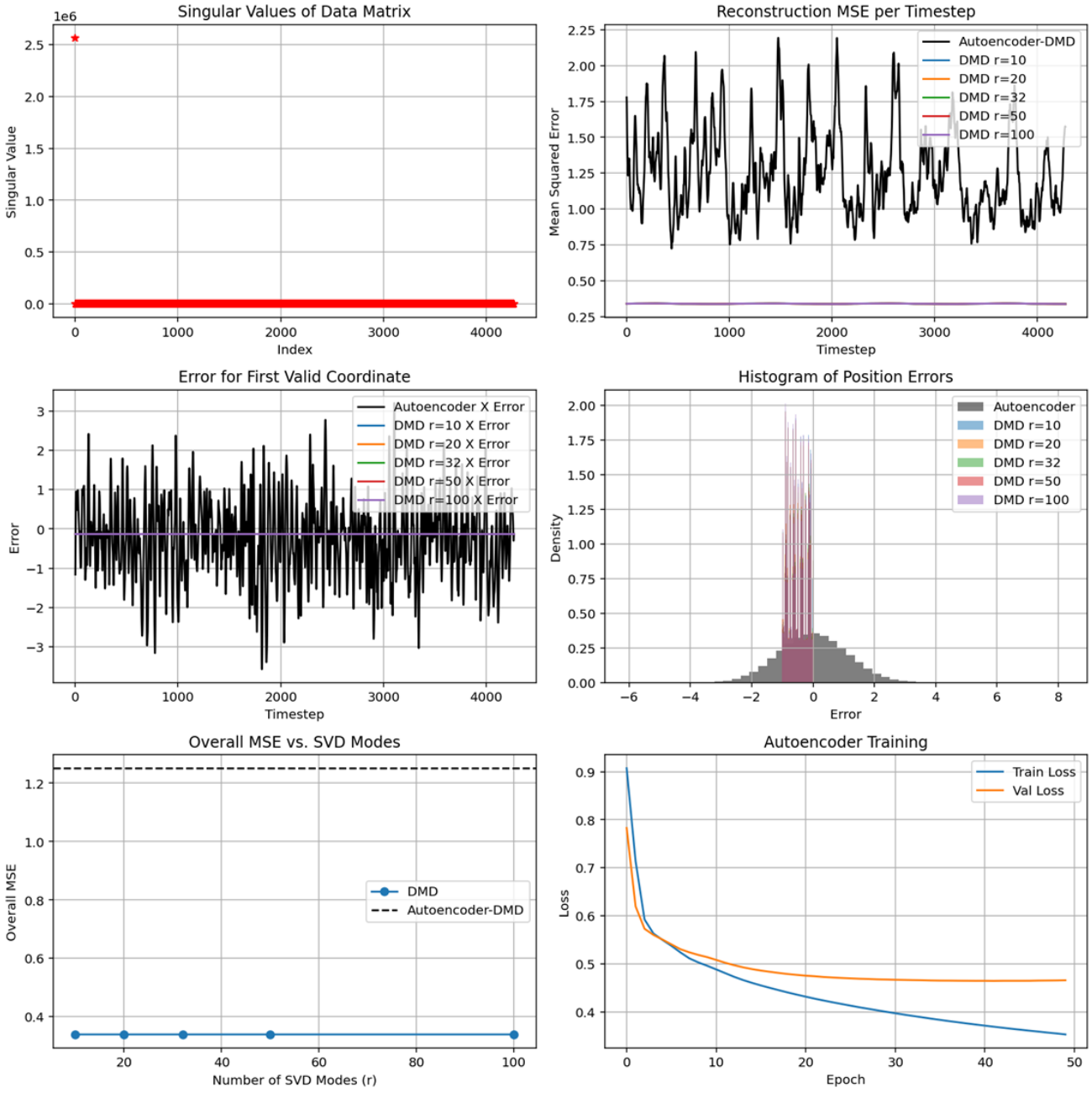


Figure 12: Comparison between autoencoder DMD with traditional DMD results with encoded layer size 64

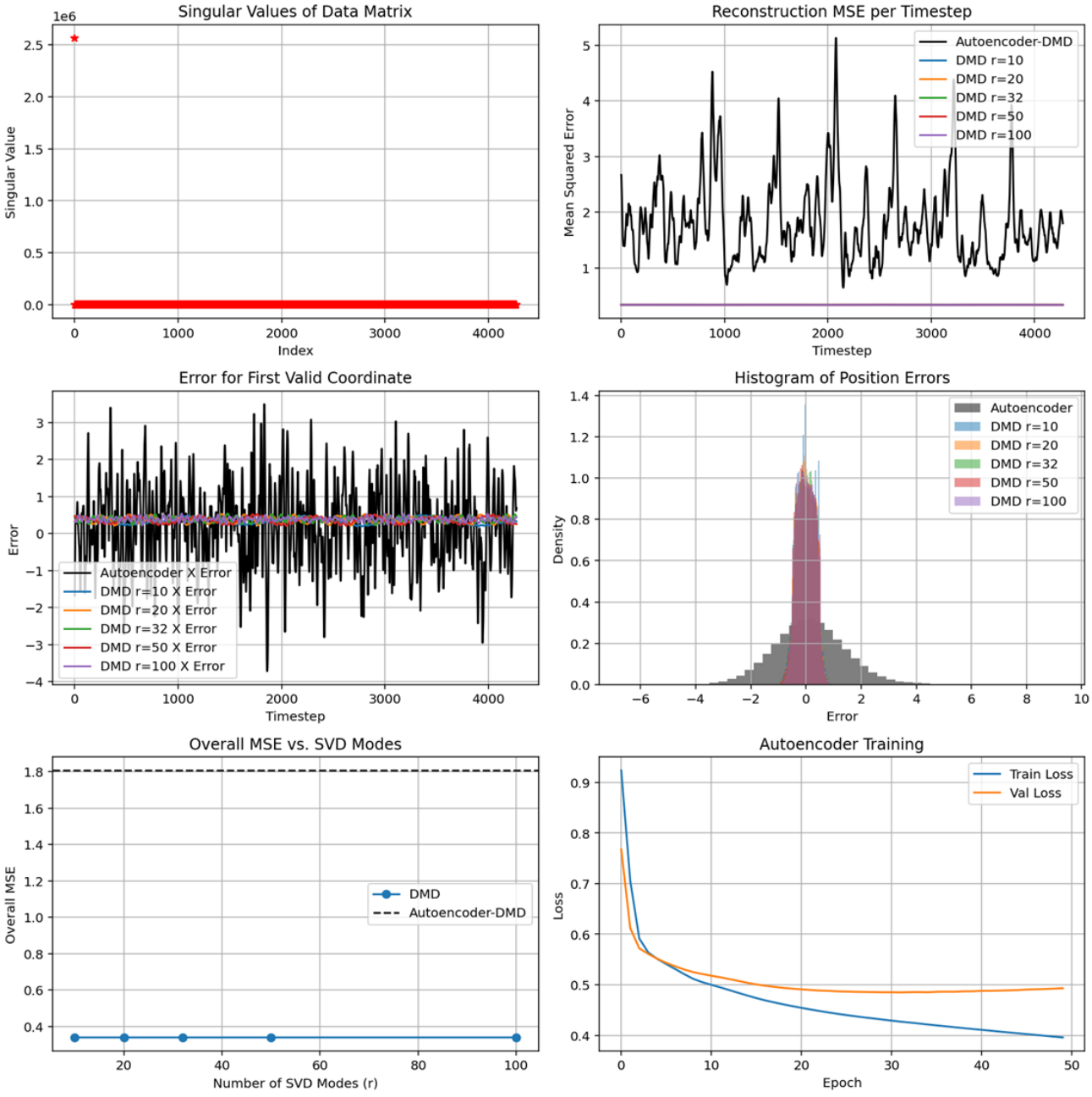


Figure 12: Comparison between autoencoder DMD with traditional DMD results with encoded layer size 12

To investigate the impact of different activation functions, we train the autoencoder using five different activation functions. As shown in Figure 13, the worst-performing activation function was leaky Relu. However, the DMD is still the best representation in terms of the error. This is understandable since we have only used a compressed representation of data using a 32-64 encoded layer. Figure 13 shows the reconstruction error for each activation function.

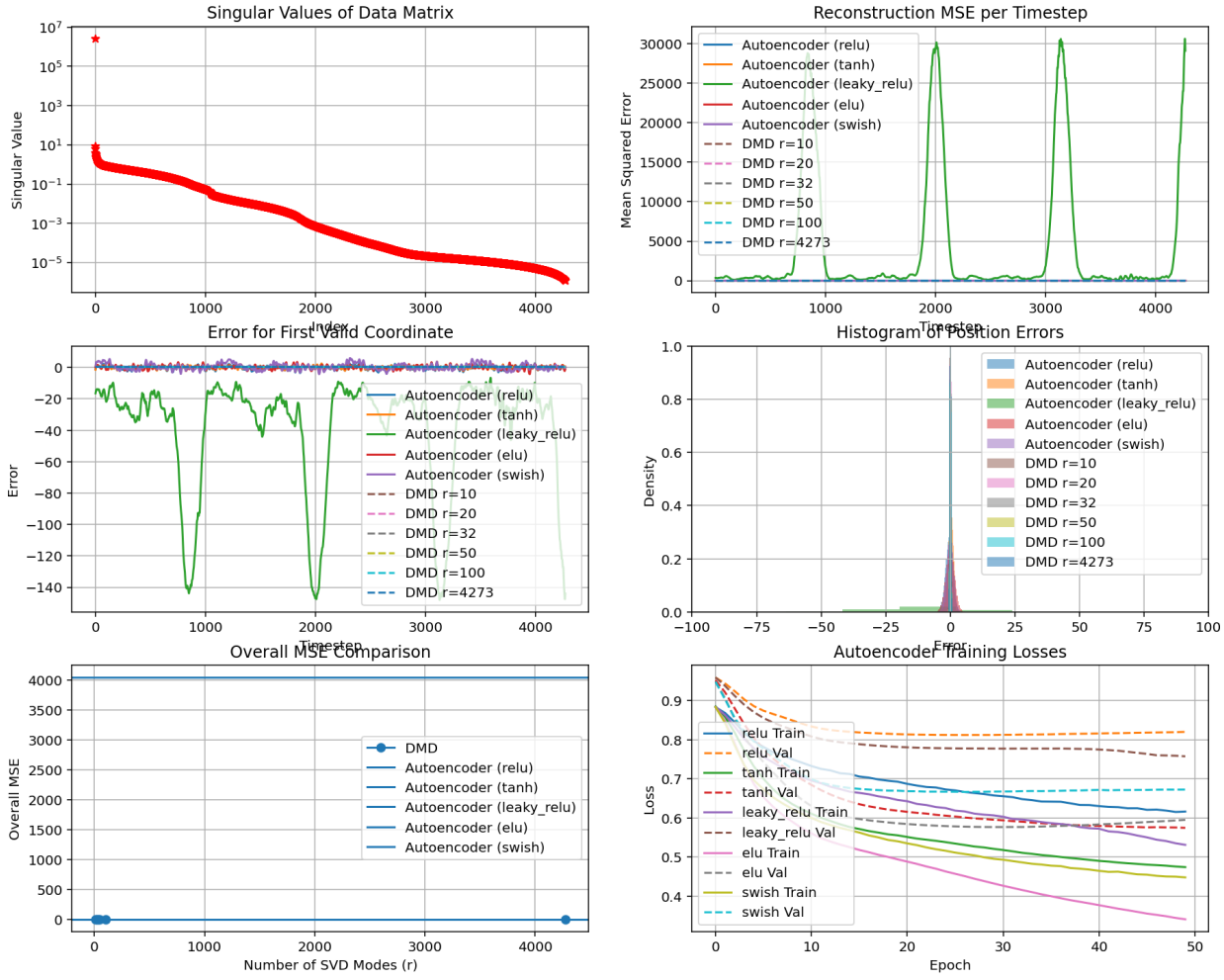


Figure 13: Reconstruction error for each activation function (Relu, leaky Relu, tanh, elu, swish)

Since encountering higher errors, we have increased the hidden latent dimension to 256, and the encoding hidden layer is 1028,512. The increasing latent dimension could not increase the accuracy of the encoded layer. On the contrary, we have a higher validation error. Figure 14 shows the results based on the activation functions. The tanh is working better than all the autoencoder-DMD. Note that in all our autoencoder training, we have only included positions. Including the forces might worsen the performance of the autoencoder-DMD.

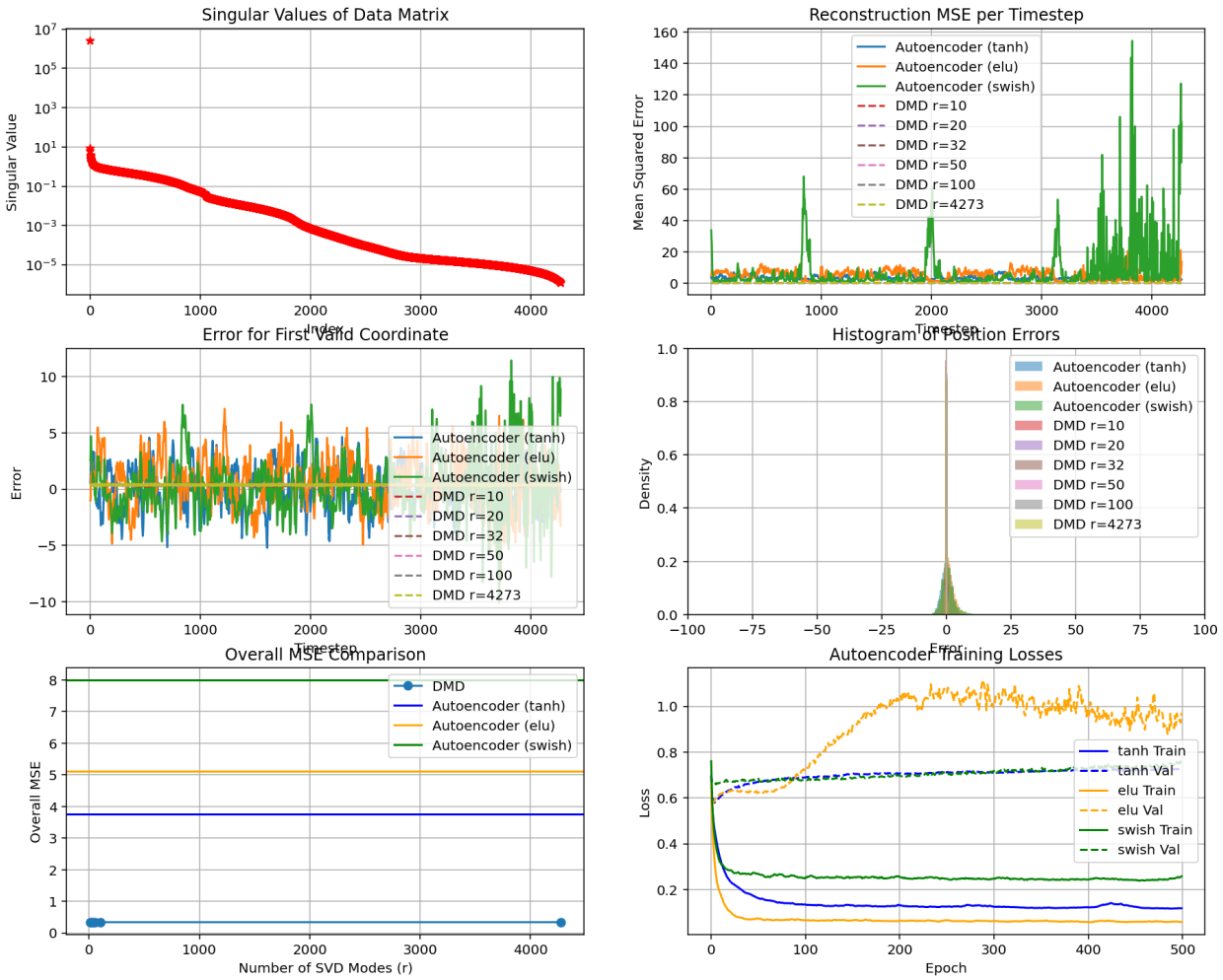
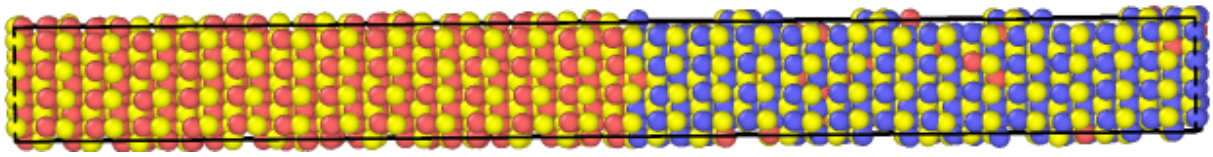


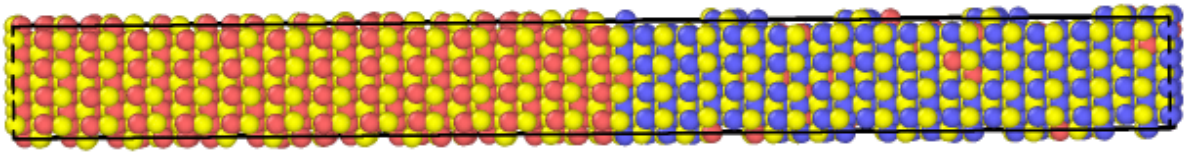
Figure 14: Results from 256 encoder layer with 1028, 512 hidden layers using different activation functions

To visualize some of the reconstructions, we have selected elu, tanh, and swish, all of which gave better results among other activation functions. The results show that the dynamics of the MD simulations are not shown in the reconstruction. Overall, the system behaves very differently from the ground truth. We need to find better representations in terms of latent space and activation functions to see whether we can reconstruct the data properly.

Elu activation function



tanh activation function



swish activation function

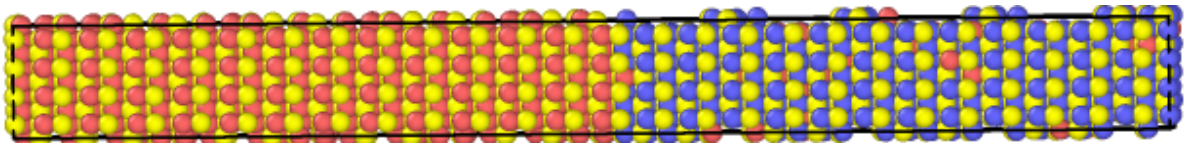


Figure 15: Reconstruction of MD simulation using activation function elu, tanh and swish, respectively.

Table 1: Comparison between the implementation of the original DMD code vs the autoencoder-DMD code.

Aspect	**Original Code**	**Extended Code with Autoencoder-DMD**
Data Used	Full matrix (x, y, z, fx, fy, fz)	Half matrix (x, y, z)
Dimensionality	High (5×num_atoms)	Reduced (Z≈64)
DMD Application	Direct on high-dimensional data	In latent space after autoencoder
Computational Cost	High due to large matrix operations	Lower due to reduced dimensionality
Prediction Accuracy	May struggle with nonlinearity	Improved if latent space is more linear
Implementation Tools	NumPy for SVD and DMD	PyTorch for autoencoder, pyDMD for DMD

Future works:

The research can be headed in directions where we can see (1) the impact of using intrusive ROM using SVD to predict future states in MD system. The interpretation might be more physics based compared to SVD alone. (2) the accuracy of autoencoder relied on the neural network fitting, if we can extend it using deep neural networks to have an efficient compressed representation at the

encoded layer. (3) Physics informed DMD can deepen our understanding of the MD simulations, which might predict accurate forces and positions. (4) An extension to the data of MD comprising velocities, forces and positions. The velocities and forces can be used to see whether we can predict the heat fluxes through the system.

Appendix:

SVD and DMD code:

```
import numpy as np
import matplotlib.pyplot as plt

def create_timestep_matrix(file_path, output_file="timestep_matrix.txt"):
    """
    Reads atomic positions from a LAMMPS trajectory file and converts them into a matrix.

    Stores:
    - Atom ID, Type, x, y, z fx fy fz in rows
    - Timesteps as columns
    - Box bounds separately

    Returns:
    - matrix: NumPy array (1664×5, timesteps)
    - sorted_timesteps: List of timesteps
    - box_bounds: Dictionary of box bounds for each timestep
    - num_atoms: Number of atoms detected in the file
    """

    timestep_data = {}
    box_bounds = {}
    current_timestep = None
    current_positions = []
    num_atoms = None  # To store detected number of atoms

    with open(file_path, 'r') as file:
        lines = file.readlines()
        i = 0
        while i < len(lines):
            line = lines[i].strip()

            # Detect timestep
            if "ITEM: TIMESTEP" in line:
                if current_positions and current_timestep is not None:
                    # Store previous timestep data
                    timestep_data[current_timestep] = np.array(current_positions).flatten()
                    current_positions = []  # Reset for next timestep

                i += 1
                current_timestep = int(lines[i].strip())

            # Read number of atoms
            elif "ITEM: NUMBER OF ATOMS" in line:
                i += 1
                num_atoms = int(lines[i].strip())

            # Read box bounds
            elif "ITEM: BOX BOUNDS" in line:
                i += 1
                x_bounds = list(map(float, lines[i].strip().split()))
                i += 1
                y_bounds = list(map(float, lines[i].strip().split()))
                i += 1
                z_bounds = list(map(float, lines[i].strip().split()))
                box_bounds[current_timestep] = (x_bounds, y_bounds, z_bounds)
```

```
            # Read atomic positions
            elif "ITEM: ATOMS" in line:
                i += 1
                for _ in range(num_atoms):
                    values = list(map(float, lines[i].strip().split()))
                    current_positions.extend(values)  # Flatten atom data into one long row
                    i += 1
                continue  # Prevent extra increment

            i += 1

    # Store last timestep
    if current_positions and current_timestep is not None:
        timestep_data[current_timestep] = np.array(current_positions).flatten()

    # Ensure timesteps are ordered correctly

    sorted_timesteps = sorted(timestep_data.keys())


    # Convert to matrix (1664 × 5, timesteps)
    num_rows = num_atoms * 8  # Each atom has (ID, Type, x, y, z)
    num_cols = len(sorted_timesteps)  # Number of time snapshots
    # cutoff=20000
    # sorted_timesteps=sorted_timesteps[:cutoff]
    matrix = np.zeros((num_rows, num_cols))
    for j, timestep in enumerate(sorted_timesteps):
        matrix[:, j] = timestep_data[timestep]

    print(f"Matrix shape: {matrix.shape} (Rows: {num_rows}, Columns: {num_cols})")

    rank_x = np.linalg.matrix_rank(matrix)
    print(f"Rank of the data matrix is {rank_x}")

    return matrix, sorted_timesteps, box_bounds, num_atoms

# Load trajectory
matrix, sorted_timesteps, box_bounds, num_atoms = create_timestep_matrix("nvt.tj")


#%%

# SVD calculation
U, S, VT = np.linalg.svd(matrix, full_matrices=False)

plt.figure()
plt.xlabel("Index")
plt.ylabel("Singular Value")
plt.plot(S, "r*")
plt.grid(True)
plt.show()

rank_x = np.linalg.matrix_rank(matrix)
print(f"Rank of the data matrix is {rank_x}")
```

```
[r,c]=np.shape(matrix)
S_filtered = np.zeros_like(S)



#%%

MODE_CUTOFF=7
U_r = U[:, :MODE_CUTOFF]

x_tilde =U_r.T @ matrix
# Step 1: Construct X and X'
X1 = x_tilde[:,:-1]  # All columns except the last
X2 = x_tilde[:,1:]   # All columns except the first

# Step 2: Compute A using least squares (Moore-Penrose pseudoinverse)
A = X2 @ np.linalg.pinv(X1)

# Step 3: Compute eigenvalues and eigenvectors of A
eigvals, W = np.linalg.eig(A)  # A W = W Λ

#%% plotting eigenvalues

plt.figure(1)
plt.tight_layout()
plt.plot(np.imag(eigvals),np.real(eigvals),'r*' )
plt.show()


#%%
# Assume matrix has shape (num_features, num_timesteps)
# Start with the last snapshot i    n the existing data

x_last = x_tilde[:, -1]

#%%
# Number of additional timesteps you want to predict
num_new_steps = 674

# Initialize an array to store new predictions
new_matrix = np.zeros((x_tilde.shape[0], num_new_steps))

# Iteratively apply A to generate future states
for i in range(num_new_steps):
    x_last = A @ x_last
    new_matrix[:, i] = x_last

# Now, you can concatenate the new data to your original matrix
new_matrix=U_r@new_matrix
augmented_matrix = np.hstack((matrix, new_matrix))

print(f"Original matrix shape: {matrix.shape}")
print(f"Augmented matrix shape (after adding new DMD-predicted steps): {augmented_matrix.shape}")
#%
```

```python
# Fixing sorted_timesteps to include new ones correctly
last_timestep = (sorted_timesteps[-1])
last=int(last_timestep)

for i in range(num_new_steps):
    sorted_timesteps.append(last + (i + 1))  # Ensures correct timestep numbering

# Update box bounds for new predicted timesteps
last_box_bounds = box_bounds[last_timestep]
for timestep in sorted_timesteps[-num_new_steps:]:
    box_bounds[timestep] = last_box_bounds  # Assume no box deformation for prediction

#%
def reconstruct_nvt_file(augmented_matrix, sorted_timesteps, box_bounds, num_atoms, output_file="n\
    """
    Reconstructs a new LAMMPS trajectory file with only dominant SVD/DMD modes.

    Ensures:
    - Correct number of atoms
    - Box bounds are preserved

    Parameters:
    - matrix: (1664×5, timesteps)
    - sorted_timesteps: Ordered timestep list
    - box_bounds: Dict of box dimensions per timestep
    - num_atoms: Number of atoms in the system
    - output_file: Name of output trajectory file
    - mode_cutoff: Number of dominant modes to retain
    """

    num_timesteps = augmented_matrix.shape[1]  # Number of time snapshot

    # Step 4: Write new LAMMPS trajectory file

    with open(output_file, "w") as f:
        for t_idx, timestep in enumerate(sorted_timesteps):
            f.write("ITEM: TIMESTEP\n")
            f.write(f"{timestep}\n")
            f.write("ITEM: NUMBER OF ATOMS\n")
            f.write(f"{num_atoms}\n")

            # Retrieve box bounds
            x_bounds, y_bounds, z_bounds = box_bounds[timestep]
            f.write("ITEM: BOX BOUNDS pp pp ss\n")
            f.write(f"{x_bounds[0]} {x_bounds[1]}\n")
            f.write(f"{y_bounds[0]} {y_bounds[1]}\n")
            f.write(f"{z_bounds[0]} {z_bounds[1]}\n")

            f.write("ITEM: ATOMS id type x y z fx fy fz\n")

            # Write atom data for the timestep
            for i in range(num_atoms):
                idx = i * 8  # Position in flattened array
                atom_id = int(augmented_matrix[idx, t_idx])  # ID
                atom_type = round(augmented_matrix[idx + 1, t_idx])  # Type
```

```
                x, y, z, fx,fy,fz = augmented_matrix[idx + 2, t_idx], augmented_matrix[idx + 3, t_i
                f.write(f"{atom_id} {atom_type} {x:.6f} {y:.6f} {z:.6f} {fx:.6f} {fy:.6f} {fz:.6f}\

    print(f"Filtered trajectory saved as {output_file} with  dominant modes.")


# Example usage

#%
reconstruct_nvt_file(augmented_matrix, sorted_timesteps, box_bounds, num_atoms, output_file=f"Recor
```

Activation function code:

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 24 23:25:37 2025

@author: Khalid
"""
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
from pydmd import DMD

def create_timestep_matrix(file_path, output_file="timestep_matrix.txt"):
    """
    Reads atomic positions from a LAMMPS trajectory file and converts them into a matrix.

    Returns:
    - matrix: NumPy array (num_atoms * 5, timesteps)
    - pos_matrix: NumPy array (num_atoms * 3, timesteps) for x, y, z only
    - sorted_timesteps: List of timesteps
    - box_bounds: Dictionary of box bounds for each timestep
    - num_atoms: Number of atoms detected
    """
    timestep_data = {}
    box_bounds = {}
    current_timestep = None
    current_positions = []
    num_atoms = None

    with open(file_path, 'r') as file:
        lines = file.readlines()
        i = 0
        while i < len(lines):
            line = lines[i].strip()
            if "ITEM: TIMESTEP" in line:
                if current_positions and current_timestep is not None:
                    timestep_data[current_timestep] = np.array(current_positions).flatten()
                    current_positions = []
                i += 1
                current_timestep = int(lines[i].strip())
            elif "ITEM: NUMBER OF ATOMS" in line:
                i += 1
                num_atoms = int(lines[i].strip())
            elif "ITEM: BOX BOUNDS" in line:
                i += 1
                x_bounds = list(map(float, lines[i].strip().split()))
                i += 1
                y_bounds = list(map(float, lines[i].strip().split()))
                i += 1
                z_bounds = list(map(float, lines[i].strip().split()))
                box_bounds[current_timestep] = (x_bounds, y_bounds, z_bounds)
            elif "ITEM: ATOMS id type xs ys zs" in line:
                i += 1
                for _ in range(num_atoms):
                    values = list(map(float, lines[i].strip().split()))
```

```
                current_positions.extend(values)
                i += 1
            continue
        i += 1

    if current_positions and current_timestep is not None:
        timestep_data[current_timestep] = np.array(current_positions).flatten()

    sorted_timesteps = sorted(timestep_data.keys())
    num_rows = num_atoms * 5
    num_cols = len(sorted_timesteps)
    matrix = np.zeros((num_rows, num_cols))
    for j, timestep in enumerate(sorted_timesteps):
        matrix[:, j] = timestep_data[timestep]

    indices = []
    for i in range(num_atoms):
        indices.extend([5*i + 2, 5*i + 3, 5*i + 4])
    pos_matrix = matrix[indices, :]

    print(f"Matrix shape: {matrix.shape}, Position matrix shape: {pos_matrix.shape}, Timesteps: {le
    print(f"Original xs range: {pos_matrix[0::3, :].min():.3f} to {pos_matrix[0::3, :].max():.3f}")
    return matrix, pos_matrix, sorted_timesteps, box_bounds, num_atoms

def robust_normalize(matrix):
    """
    Normalize matrix, preserving constant coordinates.

    Returns:
    - normalized_matrix: Normalized data
    - valid_indices: Indices of rows with non-zero std
    - mean: Mean used for normalization
    - std: Standard deviation used for normalization
    """
    mean = matrix.mean(axis=1, keepdims=True)
    std = matrix.std(axis=1, keepdims=True)
    valid_indices = np.where(std.flatten() > 1e-10)[0]
    normalized_matrix = matrix.copy()
    normalized_matrix[valid_indices, :] = (matrix[valid_indices, :] - mean[valid_indices, :]) / std
    invalid_indices = np.where(std.flatten() <= 1e-10)[0]
    normalized_matrix[invalid_indices, :] = matrix[invalid_indices, :]
    if np.any(np.isnan(normalized_matrix)):
        print("Warning: NaN values detected after normalization")
        normalized_matrix = np.nan_to_num(normalized_matrix, nan=0.0)
    print(f"Valid indices shape: {valid_indices.shape}, Invalid indices: {len(invalid_indices)}")
    return normalized_matrix, valid_indices, mean, std

def denormalize_matrix(normalized_matrix, valid_indices, mean, std, original_shape, original_matri
    """
    Denormalize matrix, restoring constant coordinates.

    Returns:
    - denorm_matrix: Denormalized data (D, T)
    """
    print(f"Denormalizing: Input shape {normalized_matrix.shape}, Target shape {original_shape}")
```

```
    if valid_indices.max() >= original_shape[0]:
        raise ValueError(f"Valid indices exceed dimension: max {valid_indices.max()} >= {original_s
    denorm_matrix = np.zeros(original_shape)
    denorm_matrix[valid_indices, :] = (normalized_matrix[:, valid_indices].T * std[valid_indices, :
    invalid_indices = np.where(std.flatten() <= 1e-10)[0]
    denorm_matrix[invalid_indices, :] = original_matrix[invalid_indices, :]
    print(f"Denormalized xs range: {denorm_matrix[0::3, :].min():.3f} to {denorm_matrix[0::3, :].ma
    return denorm_matrix

def write_lammps_trajectory(recon_data, matrix, num_atoms, sorted_timesteps, box_bounds, valid_indi
    """
    Write reconstructed data to a LAMMPS trajectory file.

    Returns:
    - None (writes file)
    """
    print(f"Writing trajectory to {output_file}, Recon data shape: {recon_data.shape}")
    if is_autoencoder:
        recon_pos = denormalize_matrix(recon_data, valid_indices, mean, std, (num_atoms * 3, len(so
        augmented_matrix = matrix.copy()
        pos_indices = []
        for i in range(num_atoms):
            pos_indices.extend([5*i + 2, 5*i + 3, 5*i + 4])
        augmented_matrix[pos_indices, :] = recon_pos
    else:
        augmented_matrix = recon_data
        pos_indices = [5*i + 2 for i in range(num_atoms)]
        xs_range = augmented_matrix[pos_indices, :].min(), augmented_matrix[pos_indices, :].max()
        print(f"DMD trajectory xs range: {xs_range[0]:.3f} to {xs_range[1]:.3f}")
        if not (0.0 <= xs_range[0] <= xs_range[1] <= 1.0):
            print("Warning: DMD coordinates out of scaled range [0, 1]. Scaling...")
            for i in range(num_atoms):
                idx = 5*i + 2
                augmented_matrix[idx:idx+3, :] = (augmented_matrix[idx:idx+3, :] - augmented_matrix

    with open(output_file, "w") as f:
        for t_idx, timestep in enumerate(sorted_timesteps):
            f.write("ITEM: TIMESTEP\n")
            f.write(f"{timestep}\n")
            f.write("ITEM: NUMBER OF ATOMS\n")
            f.write(f"{num_atoms}\n")
            x_bounds, y_bounds, z_bounds = box_bounds[timestep]
            f.write("ITEM: BOX BOUNDS pp pp pp\n")
            f.write(f"{x_bounds[0]} {x_bounds[1]}\n")
            f.write(f"{y_bounds[0]} {y_bounds[1]}\n")
            f.write(f"{z_bounds[0]} {z_bounds[1]}\n")
            f.write("ITEM: ATOMS id type xs ys zs\n")
            for i in range(num_atoms):
                idx = i * 5
                atom_id = int(augmented_matrix[idx, t_idx])
                atom_type = int(np.round(augmented_matrix[idx + 1, t_idx]))
                x = augmented_matrix[idx + 2, t_idx]
                y = augmented_matrix[idx + 3, t_idx]
                z = augmented_matrix[idx + 4, t_idx]
                f.write(f"{atom_id} {atom_type} {x:.6f} {y:.6f} {z:.6f}\n")
```

```
class Autoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dims=[1028, 512], latent_dim=256, activation='relu'):
        super(Autoencoder, self).__init__()
        if activation == 'relu':
            act_fn = nn.ReLU()
        elif activation == 'tanh':
            act_fn = nn.Tanh()
        elif activation == 'leaky_relu':
            act_fn = nn.LeakyReLU(0.01)
        elif activation == 'elu':
            act_fn = nn.ELU()
        elif activation == 'swish':
            act_fn = lambda x: x * torch.sigmoid(x)  # Custom Swish
        else:
            raise ValueError(f"Unsupported activation: {activation}")

        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dims[0]),
            act_fn if activation != 'swish' else nn.Identity(),
            nn.Linear(hidden_dims[0], hidden_dims[1]),
            act_fn if activation != 'swish' else nn.Identity(),
            nn.Linear(hidden_dims[1], latent_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dims[1]),
            act_fn if activation != 'swish' else nn.Identity(),
            nn.Linear(hidden_dims[1], hidden_dims[0]),
            act_fn if activation != 'swish' else nn.Identity(),
            nn.Linear(hidden_dims[0], input_dim)
        )
        # Apply Swish manually if selected
        self.act_fn = act_fn if activation != 'swish' else lambda x: x * torch.sigmoid(x)

    def forward(self, x):
        z = self.encoder(x)
        if hasattr(self.act_fn, '__call__'):
            z = self.act_fn(z)
        x_recon = self.decoder(z)
        if hasattr(self.act_fn, '__call__'):
            x_recon = self.act_fn(x_recon)
        return x_recon, z

def train_autoencoder(model, X_train, X_val, epochs=500, lr=0.001, batch_size=64):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    train_loader = torch.utils.data.DataLoader(X_train, batch_size=batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(X_val, batch_size=batch_size)

    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        model.train()
        train_loss = 0
```

```
        for batch in train_loader:
            optimizer.zero_grad()
            recon, _ = model(batch)
            loss = criterion(recon, batch)
            if torch.isnan(loss):
                print(f"Warning: NaN loss detected at epoch {epoch}")
                continue
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            train_loss += loss.item()
        train_losses.append(train_loss / len(train_loader))

        model.eval()
        val_loss = 0
        with torch.no_grad():
            for batch in val_loader:
                recon, _ = model(batch)
                loss = criterion(recon, batch)
                val_loss += loss.item()
            val_losses.append(val_loss / len(val_loader))

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Train Loss: {train_losses[-1]:.4f}, Val Loss: {val_losses[-1]:.4

    return train_losses, val_losses

def perform_svd_dmd(matrix, r, pos_indices):
    """
    Perform DMD on SVD-reduced data with r modes.

    Returns:
    - recon_data: Reconstructed data (num_atoms * 5, T)
    - mse_per_timestep: MSE per timestep for positions
    """
    U, S, VT = np.linalg.svd(matrix, full_matrices=False)
    U_r = U[:, :r]
    X_tilde = U_r.T @ matrix
    dmd = DMD(svd_rank=r, exact=True)
    dmd.fit(X_tilde.T)
    recon_tilde = np.real(dmd.reconstructed_data.T)
    recon_data = U_r @ recon_tilde
    error = recon_data[pos_indices, :] - matrix[pos_indices, :]
    mse_per_timestep = np.mean(error**2, axis=0)
    print(f"DMD r={r} reconstructed xs range: {recon_data[pos_indices[0::3], :].min():.3f} to {recc
    return recon_data, mse_per_timestep

# Load data
matrix, pos_matrix, sorted_timesteps, box_bounds, num_atoms = create_timestep_matrix("nvt.tj")

# Check for NaN in raw data
if np.any(np.isnan(pos_matrix)):
    print("Error: NaN values in pos_matrix")
    pos_matrix = np.nan_to_num(pos_matrix, nan=0.0)
```

```
# Robust normalization for autoencoder
pos_matrix, valid_indices, mean, std = robust_normalize(pos_matrix)

# Prepare for autoencoder
X = pos_matrix.T
X_tensor = torch.FloatTensor(X)
train_size = int(0.8 * len(X_tensor))
X_train, X_val = X_tensor[train_size:], X_tensor[:train_size]

# Test different activation functions
activations = ['leaky_relu', 'tanh', 'relu', 'elu', 'swish']
autoencoder_results = {}

for activation in activations:
    print(f"\nTraining autoencoder with {activation} activation")
    # Train autoencoder
    input_dim = pos_matrix.shape[0]
    autoencoder = Autoencoder(input_dim=input_dim, activation=activation)
    train_losses, val_losses = train_autoencoder(autoencoder, X_train, X_val)

    # Get latent representations
    autoencoder.eval()
    with torch.no_grad():
        _, Z = autoencoder(X_tensor)
    Z = Z.numpy().T

    # Check latent representations
    if np.any(np.isnan(Z)):
        print(f"Error: NaN values in latent representations for {activation}")
        Z = np.nan_to_num(Z, nan=0.0)

    # Compute rank of Z for DMD
    z_rank = np.linalg.matrix_rank(Z, tol=1e-10)
    print(f"Rank of latent matrix Z ({activation}): {z_rank}")

    # Apply DMD
    dmd = DMD(svd_rank=z_rank, exact=True)
    try:
        dmd.fit(Z.T)
    except np.linalg.LinAlgError as e:
        print(f"DMD failed for {activation}: {e}")
        continue

    # Reconstruct data
    recon_data = np.zeros((X.shape[0], X.shape[1]))
    with torch.no_grad():
        latent_recon = np.real(dmd.reconstructed_data)
        recon_tensor = autoencoder.decoder(torch.FloatTensor(latent_recon)).detach().numpy()
        recon_data[:, :] = recon_tensor[:, :]

    # Compute error
    auto_error = recon_data - X
    auto_mse_per_timestep = np.mean(auto_error[:, valid_indices]**2, axis=1)
    auto_overall_mse = np.mean(auto_mse_per_timestep)
    print(f"Autoencoder-DMD ({activation}) Overall MSE: {auto_overall_mse:.4f}")
```

```
    autoencoder_results[activation] = {
        'recon_data': recon_data,
        'mse_per_timestep': auto_mse_per_timestep,
        'overall_mse': auto_overall_mse,
        'train_losses': train_losses,
        'val_losses': val_losses
    }

    # Write trajectory
    write_lammps_trajectory(recon_data, matrix, num_atoms, sorted_timesteps, box_bounds, valid_indi

# SVD on matrix for direct DMD
U, S, VT = np.linalg.svd(matrix, full_matrices=False)
matrix_rank = np.linalg.matrix_rank(matrix, tol=1e-10)
print(f"Rank of data matrix: {matrix_rank}")

# Position indices for error computation
pos_indices = []
for i in range(num_atoms):
    pos_indices.extend([5*i + 2, 5*i + 3, 5*i + 4])

# Vary r for direct DMD
r_values = [10, 20, 32, 50, 100,matrix_rank]
dmd_reconstructions = {}
dmd_mse = {}
for r in r_values:
    print(f"Performing DMD with r={r}")
    recon_r, mse_r = perform_svd_dmd(matrix, r, pos_indices)
    dmd_reconstructions[r] = recon_r
    dmd_mse[r] = mse_r
    print(f"DMD (r={r}) Overall MSE: {np.mean(mse_r):.4f}")
    write_lammps_trajectory(recon_r, matrix, num_atoms, sorted_timesteps, box_bounds, output_file=f
#%%
# Plotting
activations=['tanh','elu','swish']
plt.figure(figsize=(15, 12))

# Singular Values
plt.subplot(3, 2, 1)
plt.semilogy(S, "r*")
plt.xlabel("Index")
plt.ylabel("Singular Value")
plt.title("Singular Values of Data Matrix")
plt.grid(True)

# MSE per Timestep
plt.subplot(3, 2, 2)
for activation in activations:
    plt.plot(autoencoder_results[activation]['mse_per_timestep'], label=f"Autoencoder ({activation}
for r in r_values:
    plt.plot(dmd_mse[r], label=f"DMD r={r}", linestyle='--')
plt.xlabel("Timestep")
plt.ylabel("Mean Squared Error")
plt.title("Reconstruction MSE per Timestep")
```

```
plt.legend()
plt.grid(True)

# Error for First Valid Coordinate
plt.subplot(3, 2, 3)

for activation in activations:
    auto_error = autoencoder_results[activation]['recon_data'] - X
    plt.plot(auto_error[:, valid_indices[0]], label=f"Autoencoder ({activation})")
for r in r_values:
    r_error = dmd_reconstructions[r][pos_indices[0], :] - matrix[pos_indices[0], :]
    plt.plot(r_error, label=f"DMD r={r}", linestyle='--')
plt.xlabel("Timestep")
plt.ylabel("Error")
plt.title("Error for First Valid Coordinate")
plt.legend()
plt.grid(True)

# Error Histogram
plt.subplot(3, 2, 4)

for activation in activations:
    auto_error = autoencoder_results[activation]['recon_data'] - X
    plt.hist(auto_error[:, valid_indices].flatten(), bins=50, density=True, alpha=0.5, label=f"Autc
for r in r_values:
    r_error = dmd_reconstructions[r][pos_indices, :] - matrix[pos_indices, :]
    plt.hist(r_error.flatten(), bins=50, density=True, alpha=0.5, label=f"DMD r={r}")
plt.xlabel("Error")
plt.ylabel("Density")
plt.title("Histogram of Position Errors")
plt.xlim([-100,100])
plt.ylim([0,1])

plt.legend()
plt.grid(True)

# Overall MSE vs. r
plt.subplot(3, 2, 5)
r_mse_values = [np.mean(dmd_mse[r]) for r in r_values]
plt.plot(r_values, r_mse_values, marker='o', label="DMD")
colors = ['blue', 'orange', 'green']
for idx, activation in enumerate(activations):
    color=colors[idx%len(colors)]
    mse = autoencoder_results[activation]['overall_mse']
    plt.axhline(mse, label=f"Autoencoder ({activation})",color=color)
plt.xlabel("Number of SVD Modes (r)")
plt.ylabel("Overall MSE")
plt.title("Overall MSE Comparison")
plt.legend()
plt.grid(True)

# Autoencoder Training Losses
plt.subplot(3, 2, 6)
colors = ['blue', 'orange', 'green']
for idx, activation in enumerate(activations):
```

```
    color=colors[idx%len(colors)]
    plt.plot(autoencoder_results[activation]['train_losses'], color=color,label=f"{activation} Trai
    plt.plot(autoencoder_results[activation]['val_losses'], color=color, linestyle='--', label=f"{a
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Autoencoder Training Losses")
plt.legend()
plt.grid(True)

plt.show()
plt.tight_layout()
plt.savefig('reconstruction_error_comparison_activations.png')
```

Autoencoder-DMD with DMD comparison:

```python
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
from pydmd import DMD

def create_timestep_matrix(file_path, output_file="timestep_matrix.txt"):
    """
    Reads atomic positions from a LAMMPS trajectory file and converts them into a matrix.

    Returns:
    - matrix: NumPy array (num_atoms * 5, timesteps)
    - pos_matrix: NumPy array (num_atoms * 3, timesteps) for x, y, z only
    - sorted_timesteps: List of timesteps
    - box_bounds: Dictionary of box bounds for each timestep
    - num_atoms: Number of atoms detected
    """
    timestep_data = {}
    box_bounds = {}
    current_timestep = None
    current_positions = []
    num_atoms = None

    with open(file_path, 'r') as file:
        lines = file.readlines()
        i = 0
        while i < len(lines):
            line = lines[i].strip()
            if "ITEM: TIMESTEP" in line:
                if current_positions and current_timestep is not None:
                    timestep_data[current_timestep] = np.array(current_positions).flatten()
                    current_positions = []
                i += 1
                current_timestep = int(lines[i].strip())
            elif "ITEM: NUMBER OF ATOMS" in line:
                i += 1
                num_atoms = int(lines[i].strip())
            elif "ITEM: BOX BOUNDS" in line:
                i += 1
                x_bounds = list(map(float, lines[i].strip().split()))
                i += 1
                y_bounds = list(map(float, lines[i].strip().split()))
                i += 1
                z_bounds = list(map(float, lines[i].strip().split()))
                box_bounds[current_timestep] = (x_bounds, y_bounds, z_bounds)
            elif "ITEM: ATOMS id type xs ys zs" in line:
                i += 1
                for _ in range(num_atoms):
                    values = list(map(float, lines[i].strip().split()))
                    current_positions.extend(values)
                    i += 1
                continue
            i += 1

    if current_positions and current_timestep is not None:
```

```
            timestep_data[current_timestep] = np.array(current_positions).flatten()

        sorted_timesteps = sorted(timestep_data.keys())
        num_rows = num_atoms * 5
        num_cols = len(sorted_timesteps)
        matrix = np.zeros((num_rows, num_cols))
        for j, timestep in enumerate(sorted_timesteps):
            matrix[:, j] = timestep_data[timestep]

        indices = []
        for i in range(num_atoms):
            indices.extend([5*i + 2, 5*i + 3, 5*i + 4])
        pos_matrix = matrix[indices, :]

        print(f"Matrix shape: {matrix.shape}, Position matrix shape: {pos_matrix.shape}, Timesteps: {le
        print(f"Original xs range: {pos_matrix[0::3, :].min():.3f} to {pos_matrix[0::3, :].max():.3f}")
        return matrix, pos_matrix, sorted_timesteps, box_bounds, num_atoms

    def robust_normalize(matrix):
        """
        Normalize matrix, preserving constant coordinates.

        Returns:
        - normalized_matrix: Normalized data
        - valid_indices: Indices of rows with non-zero std
        - mean: Mean used for normalization
        - std: Standard deviation used for normalization
        """
        mean = matrix.mean(axis=1, keepdims=True)
        std = matrix.std(axis=1, keepdims=True)
        valid_indices = np.where(std.flatten() > 1e-10)[0]
        normalized_matrix = matrix.copy()
        normalized_matrix[valid_indices, :] = (matrix[valid_indices, :] - mean[valid_indices, :]) / std
        # Preserve constant coordinates using original values
        invalid_indices = np.where(std.flatten() <= 1e-10)[0]
        normalized_matrix[invalid_indices, :] = matrix[invalid_indices, :]
        if np.any(np.isnan(normalized_matrix)):
            print("Warning: NaN values detected after normalization")
            normalized_matrix = np.nan_to_num(normalized_matrix, nan=0.0)
        print(f"Valid indices shape: {valid_indices.shape}, Invalid indices: {len(invalid_indices)}")
        return normalized_matrix, valid_indices, mean, std

    def denormalize_matrix(normalized_matrix, valid_indices, mean, std, original_shape, original_matrix
        """
        Denormalize matrix, restoring constant coordinates.

        Args:
        - normalized_matrix: Normalized data (T, D)
        - valid_indices: Indices of rows with non-zero std
        - mean: Original mean (D, 1)
        - std: Original std (D, 1)
        - original_shape: Shape of output matrix (D, T)
        - original_matrix: Original position matrix (D, T)

        Returns:
```

```
    - denorm_matrix: Denormalized data (D, T)
    """
    print(f"Denormalizing: Input shape {normalized_matrix.shape}, Target shape {original_shape}")
    if valid_indices.max() >= original_shape[0]:
        raise ValueError(f"Valid indices exceed dimension: max {valid_indices.max()} >= {original_s
    denorm_matrix = np.zeros(original_shape)
    denorm_matrix[valid_indices, :] = (normalized_matrix[:, valid_indices].T * std[valid_indices, :
    # Restore constant coordinates from original matrix
    invalid_indices = np.where(std.flatten() <= 1e-10)[0]
    denorm_matrix[invalid_indices, :] = original_matrix[invalid_indices, :]
    print(f"Denormalized xs range: {denorm_matrix[0::3, :].min():.3f} to {denorm_matrix[0::3, :].ma
    return denorm_matrix

def write_lammps_trajectory(recon_data, matrix, num_atoms, sorted_timesteps, box_bounds, valid_indi
    """
    Write reconstructed data to a LAMMPS trajectory file, matching nvt.tj format.

    Args:
    - recon_data: Reconstructed data (num_atoms * 5, T) or (T, num_atoms * 3)
    - matrix: Original matrix (num_atoms * 5, T)
    - num_atoms: Number of atoms
    - sorted_timesteps: List of timesteps
    - box_bounds: Dictionary of box bounds
    - valid_indices: Indices of valid coordinates (for autoencoder)
    - mean: Mean from normalization (for autoencoder)
    - std: Std from normalization (for autoencoder)
    - pos_matrix: Original position matrix (num_atoms * 3, T) (for autoencoder)
    - output_file: Output filename
    - is_autoencoder: Flag to handle autoencoder vs. direct DMD data
    """
    print(f"Writing trajectory to {output_file}, Recon data shape: {recon_data.shape}")
    if is_autoencoder:
        # Denormalize position data, preserving frozen atoms
        recon_pos = denormalize_matrix(recon_data, valid_indices, mean, std, (num_atoms * 3, len(sc
        # Create augmented matrix with IDs and types from original
        augmented_matrix = matrix.copy()
        pos_indices = []
        for i in range(num_atoms):
            pos_indices.extend([5*i + 2, 5*i + 3, 5*i + 4])
        augmented_matrix[pos_indices, :] = recon_pos
    else:
        # Direct DMD: use recon_data as augmented_matrix
        augmented_matrix = recon_data
        # Check scaling
        pos_indices = [5*i + 2 for i in range(num_atoms)]
        xs_range = augmented_matrix[pos_indices, :].min(), augmented_matrix[pos_indices, :].max()
        print(f"DMD trajectory xs range: {xs_range[0]:.3f} to {xs_range[1]:.3f}")
        if not (0.0 <= xs_range[0] <= xs_range[1] <= 1.0):
            print("Warning: DMD coordinates out of scaled range [0, 1]. Scaling...")
            for i in range(num_atoms):
                idx = 5*i + 2
                augmented_matrix[idx:idx+3, :] = (augmented_matrix[idx:idx+3, :] - augmented_matri>

    with open(output_file, "w") as f:
        for t_idx, timestep in enumerate(sorted_timesteps):
```

```
            f.write("ITEM: TIMESTEP\n")
            f.write(f"{timestep}\n")
            f.write("ITEM: NUMBER OF ATOMS\n")
            f.write(f"{num_atoms}\n")
            x_bounds, y_bounds, z_bounds = box_bounds[timestep]
            f.write("ITEM: BOX BOUNDS pp pp pp\n")
            f.write(f"{x_bounds[0]} {x_bounds[1]}\n")
            f.write(f"{y_bounds[0]} {y_bounds[1]}\n")
            f.write(f"{z_bounds[0]} {z_bounds[1]}\n")
            f.write("ITEM: ATOMS id type xs ys zs\n")
            for i in range(num_atoms):
                idx = i * 5
                atom_id = int(augmented_matrix[idx, t_idx])
                atom_type = int(np.round(augmented_matrix[idx + 1, t_idx]))
                x = augmented_matrix[idx + 2, t_idx]
                y = augmented_matrix[idx + 3, t_idx]
                z = augmented_matrix[idx + 4, t_idx]
                f.write(f"{atom_id} {atom_type} {x:.6f} {y:.6f} {z:.6f}\n")

class Autoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dims=[256, 128], latent_dim=12):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dims[0]),
            nn.ReLU(),
            nn.Linear(hidden_dims[0], hidden_dims[1]),
            nn.ReLU(),
            nn.Linear(hidden_dims[1], latent_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dims[1]),
            nn.ReLU(),
            nn.Linear(hidden_dims[1], hidden_dims[0]),
            nn.ReLU(),
            nn.Linear(hidden_dims[0], input_dim)
        )

    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon, z

def train_autoencoder(model, X_train, X_val, epochs=50, lr=0.0001, batch_size=64):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    train_loader = torch.utils.data.DataLoader(X_train, batch_size=batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(X_val, batch_size=batch_size)

    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        model.train()
        train_loss = 0
        for batch in train_loader:
```

```
            optimizer.zero_grad()
            recon, _ = model(batch)
            loss = criterion(recon, batch)
            if torch.isnan(loss):
                print(f"Warning: NaN loss detected at epoch {epoch}")
                continue
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            train_loss += loss.item()
        train_losses.append(train_loss / len(train_loader))

        model.eval()
        val_loss = 0
        with torch.no_grad():
            for batch in val_loader:
                recon, _ = model(batch)
                loss = criterion(recon, batch)
                val_loss += loss.item()
            val_losses.append(val_loss / len(val_loader))

        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Train Loss: {train_losses[-1]:.4f}, Val Loss: {val_losses[-1]:.4

    return train_losses, val_losses

def perform_svd_dmd(matrix, r, pos_indices):
    """
    Perform DMD on SVD-reduced data with r modes.

    Returns:
    - recon_data: Reconstructed data (num_atoms * 5, T)
    - mse_per_timestep: MSE per timestep for positions
    """
    U, S, VT = np.linalg.svd(matrix, full_matrices=False)
    U_r = U[:, :r]
    X_tilde = U_r.T @ matrix
    dmd = DMD(svd_rank=r, exact=True)
    dmd.fit(X_tilde.T)
    recon_tilde = np.real(dmd.reconstructed_data.T)
    recon_data = U_r @ recon_tilde
    error = recon_data[pos_indices, :] - matrix[pos_indices, :]
    mse_per_timestep = np.mean(error**2, axis=0)
    print(f"DMD r={r} reconstructed xs range: {recon_data[pos_indices[0::3], :].min():.3f} to {recc
    return recon_data, mse_per_timestep

# Load data
matrix, pos_matrix, sorted_timesteps, box_bounds, num_atoms = create_timestep_matrix("nvt.tj")

# Check for NaN in raw data
if np.any(np.isnan(pos_matrix)):
    print("Error: NaN values in pos_matrix")
    pos_matrix = np.nan_to_num(pos_matrix, nan=0.0)

# Robust normalization for autoencoder
```

```
pos_matrix, valid_indices, mean, std = robust_normalize(pos_matrix)

# Prepare for autoencoder
X = pos_matrix.T
X_tensor = torch.FloatTensor(X)
train_size = int(0.8 * len(X_tensor))
X_train, X_val = X_tensor[:train_size], X_tensor[train_size:]

# Train autoencoder
input_dim = pos_matrix.shape[0]
autoencoder = Autoencoder(input_dim=input_dim)
train_losses, val_losses = train_autoencoder(autoencoder, X_train, X_val)

# Get latent representations
autoencoder.eval()
with torch.no_grad():
    _, Z = autoencoder(X_tensor)
Z = Z.numpy().T

# Check latent representations
if np.any(np.isnan(Z)):
    print("Error: NaN values in latent representations")
    Z = np.nan_to_num(Z, nan=0.0)

# Compute rank of Z for DMD
z_rank = np.linalg.matrix_rank(Z, tol=1e-10)
print(f"Rank of latent matrix Z: {z_rank}")

# Apply DMD (autoencoder)
dmd = DMD(svd_rank=z_rank, exact=True)
try:
    dmd.fit(Z.T)
except np.linalg.LinAlgError as e:
    print(f"DMD failed: {e}")
    exit()

# Reconstruct data (autoencoder)
recon_data = np.zeros((X.shape[0], X.shape[1]))
with torch.no_grad():
    latent_recon = np.real(dmd.reconstructed_data)
    recon_tensor = autoencoder.decoder(torch.FloatTensor(latent_recon)).detach().numpy()
    recon_data[:, :] = recon_tensor[:, :]

# Compute autoencoder-DMD error
auto_error = recon_data - X
auto_mse_per_timestep = np.mean(auto_error[:, valid_indices]**2, axis=1)
auto_overall_mse = np.mean(auto_mse_per_timestep)
print(f"Autoencoder-DMD Overall MSE: {auto_overall_mse:.4f}")

# SVD on matrix for direct DMD
U, S, VT = np.linalg.svd(matrix, full_matrices=False)
matrix_rank = np.linalg.matrix_rank(matrix, tol=1e-10)
print(f"Rank of data matrix: {matrix_rank}")

# Position indices for error computation
```

```
pos_indices = []
for i in range(num_atoms):
    pos_indices.extend([5*i + 2, 5*i + 3, 5*i + 4])

# Vary r for direct DMD
r_values = [10, 20, 32, 50, 100]
dmd_reconstructions = {}
dmd_mse = {}
for r in r_values:
    print(f"Performing DMD with r={r}")
    recon_r, mse_r = perform_svd_dmd(matrix, r, pos_indices)
    dmd_reconstructions[r] = recon_r
    dmd_mse[r] = mse_r
    print(f"DMD (r={r}) Overall MSE: {np.mean(mse_r):.4f}")
    write_lammps_trajectory(recon_r, matrix, num_atoms, sorted_timesteps, box_bounds, output_file=f

# Write autoencoder-DMD trajectory
write_lammps_trajectory(recon_data, matrix, num_atoms, sorted_timesteps, box_bounds, valid_indices,

# Plotting
plt.figure(figsize=(12, 12))
plt.subplot(3, 2, 1)
plt.plot(S, "r*")
plt.xlabel("Index")
plt.ylabel("Singular Value")
plt.title("Singular Values of Data Matrix")
plt.grid(True)

plt.subplot(3, 2, 2)
plt.plot(auto_mse_per_timestep, label="Autoencoder-DMD", color='black')
for r in r_values:
    plt.plot(dmd_mse[r], label=f"DMD r={r}")
plt.xlabel("Timestep")
plt.ylabel("Mean Squared Error")
plt.title("Reconstruction MSE per Timestep")
plt.legend()
plt.grid(True)

plt.subplot(3, 2, 3)
plt.plot(auto_error[:, valid_indices[0]], label="Autoencoder X Error", color='black')
for r in r_values:
    r_error = dmd_reconstructions[r][pos_indices[0], :] - matrix[pos_indices[0], :]
    plt.plot(r_error, label=f"DMD r={r} X Error")
plt.xlabel("Timestep")
plt.ylabel("Error")
plt.title("Error for First Valid Coordinate")
plt.legend()
plt.grid(True)

plt.subplot(3, 2, 4)
plt.hist(auto_error[:, valid_indices].flatten(), bins=50, density=True, alpha=0.5, label="Autoencod
for r in r_values:
    r_error = dmd_reconstructions[r][pos_indices, :] - matrix[pos_indices, :]
    plt.hist(r_error.flatten(), bins=50, density=True, alpha=0.5, label=f"DMD r={r}")
plt.xlabel("Error")
```

```
plt.ylabel("Density")
plt.title("Histogram of Position Errors")
plt.legend()
plt.grid(True)

plt.subplot(3, 2, 5)
r_mse_values = [np.mean(dmd_mse[r]) for r in r_values]
plt.plot(r_values, r_mse_values, marker='o', label="DMD")
plt.axhline(auto_overall_mse, color='black', linestyle='--', label="Autoencoder-DMD")
plt.xlabel("Number of SVD Modes (r)")
plt.ylabel("Overall MSE")
plt.title("Overall MSE vs. SVD Modes")
plt.legend()
plt.grid(True)

plt.subplot(3, 2, 6)
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Val Loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Autoencoder Training")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.savefig('reconstruction_error_comparison.png')
plt.show()
```

References:

(1) Kaiser, J.; Feng, T.; Maassen, J.; Wang, X.; Ruan, X.; Lundstrom, M. Thermal Transport at the Nanoscale: A Fourier's Law vs. Phonon Boltzmann Equation Study. *J Appl Phys* **2017**, *121* (4). https://doi.org/10.1063/1.4974872.

(2) Guo, Y.; Zhang, Z.; Bescond, M.; Xiong, S.; Wang, M.; Nomura, M.; Volz, S. Size Effect on Phonon Hydrodynamics in Graphite Microstructures and Nanostructures. *Phys Rev B* **2021**, *104* (7), 075450. https://doi.org/10.1103/PHYSREVB.104.075450/FIGURES/16/THUMBNAIL.

(3) Xie, R.; Tiwari, J.; Feng, T. Impacts of Various Interfacial Nanostructures on Spectral Phonon Thermal Boundary Conductance. *J Appl Phys* **2022**, *132* (11). https://doi.org/10.1063/5.0106685.

(4) Sääskilahti, K.; Oksanen, J.; Tulkki, J.; Volz, S. Role of Anharmonic Phonon Scattering in the Spectrally Decomposed Thermal Conductance at Planar Interfaces. *Phys Rev B Condens Matter Mater Phys* **2014**, *90* (13). https://doi.org/10.1103/PhysRevB.90.134312.

(5) Feng, T.; Yao, W.; Wang, Z.; Shi, J.; Li, C.; Cao, B.; Ruan, X. Spectral Analysis of Nonequilibrium Molecular Dynamics: Spectral Phonon Temperature and Local Nonequilibrium in Thin Films and across Interfaces. *Phys Rev B* **2017**, *95* (19), 1–13. https://doi.org/10.1103/PhysRevB.95.195202.

(6) Ford, W. The Singular Value Decomposition. *Numer Linear Algebra Appl* **2015**, 299–320. https://doi.org/10.1016/B978-0-12-394435-1.00015-6.

(7) Chengwang, L. Singular Value Decomposition in Active Monitoring Data Analysis. *Handbook of Geophysical Exploration: Seismic Exploration* **2010**, *40* (C), 421–430. https://doi.org/10.1016/S0950-1401(10)04027-9.

(8) Kutz, J. N.; Brunton, S. L.; Brunton, B. W.; Proctor, J. L. Dynamic Mode Decomposition. *Dynamic Mode Decomposition* **2016**. https://doi.org/10.1137/1.9781611974508.

(9) Curtis, C. W.; Alford-Lago, D. J. Dynamic-Mode Decomposition and Optimal Prediction. *Phys Rev E* **2021**, *103* (1), 012201. https://doi.org/10.1103/PHYSREVE.103.012201/FIGURES/5/MEDIUM.

(10) Colbrook, M. J. The Multiverse of Dynamic Mode Decomposition Algorithms. **2023**.

(11) Maliyov, I.; Yin, J.; Yao, J.; Yang, C.; Bernardi, M. Dynamic Mode Decomposition of Nonequilibrium Electron-Phonon Dynamics: Accelerating the First-Principles Real-Time Boltzmann Equation. *NPJ Comput Mater* **2024**, *10* (1). https://doi.org/10.1038/s41524-024-01308-4.