Summer Projects

Update Presentation

Friday - July 25th, 2025

1. Corrections to Code (Post-Meeting)
2. Visualizing the Data Changing Through The Layers
3. Implementation with Synthetic Data
4. Hyperplane Separation
5. Questions and Remarks

1 Corrections to Code (Post-Meeting on July 2nd)

Previous

Cross-Entropy Loss

```
criterion = nn.CrossEntropyLoss()
```

I switched to MSE instead of cross-entropy because MSE directly pushes each class to a fixed target vector, making it possible to see how the ReLU layers shape the data step-by-step through explicit linear truncation maps.

To Align with Papers

Current

MSE Loss (w/ One-Hot Vectors)

```
def one_hot(labels, num_classes):
    return torch.nn.functional.one_hot(labels, num_classes=num_classes).float()

criterion = nn.MSELoss()  # squared L2 loss, matches the paper
```

I added one-hot encoding because MSE loss requires the targets to be probability vectors instead of class indices, unlike cross-entropy.

Small Epochs and High Learning Rate

```
epochs = 10
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01,
```

For Clearer Separation

Large Epochs and Low Learning Rate

```
epochs = 200
```

```
optimizer = optim.SGD(model.parameters(), lr=0.005,
```

Using more epochs with a lower learning rate helps the network gradually refine its weights, which can make the separation of data through layers and the resulting truncation maps clearer and more interpretable.

1 Corrections to Code (Post-Meeting on July 2nd)

Where I Left Off

- I did PCA on the training data and then used those principal components to project both the data and the cones, including their base points and edges, into the same 2D space.

2 Visualizing the Data Changing Through The Layers

Suggestions Before I Start

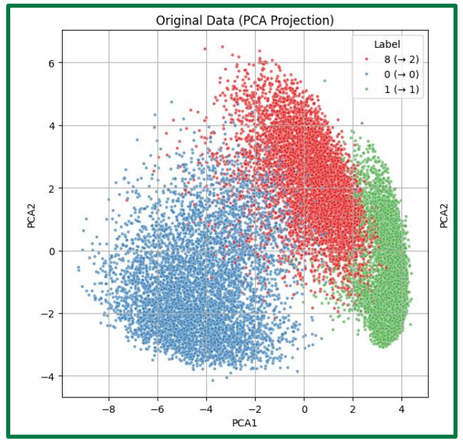
- No need to compute the a(i) components; compute truncated data points directly using the truncation map definition and cumulative parameters.
- For plots:
  - Do PCA on the original training data to get principal components.
  - Use those same principal components to project the truncated data.
- For Q=3 classes, make 4 plots:

  a. 1 plot of original data projected to 2D.
  b. 3 plots for truncated data using τ_{W^{(l)}, b^{(l)}} for ℓ=1,2,3, projected to 2D.

- Alternative projections could be tried (e.g., PCA on truncated data or projections to class means).
- Also plot the final network output in ℝ^Q; for Q=3, this requires no dimensionality reduction.

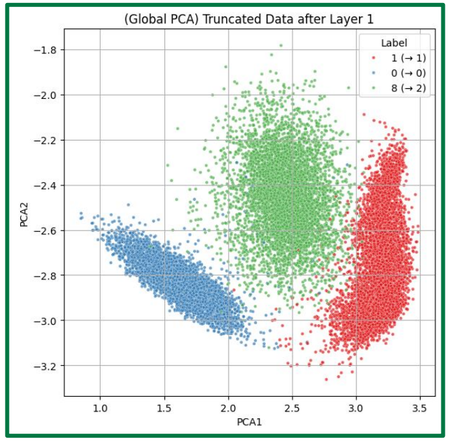
2 Visualizing the Data Changing Through The Layers

(1) Global PCA



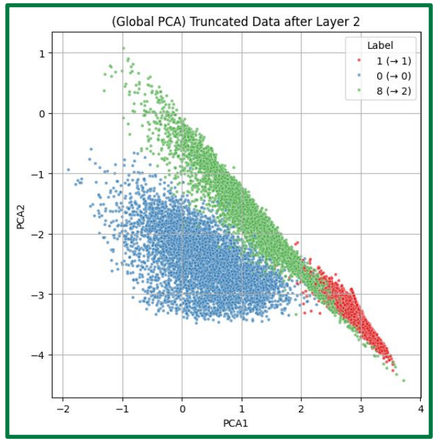
PCA on the Original Training Data

Shows how the raw input data looks when reduced to 2D using PCA.



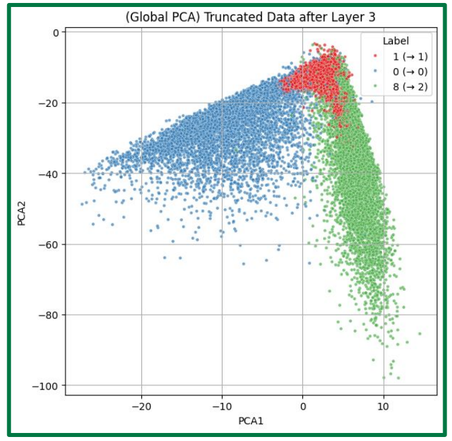
Truncation Map for Hidden Layer 1

Shows how the first hidden layer begins to reshape and separate the data.



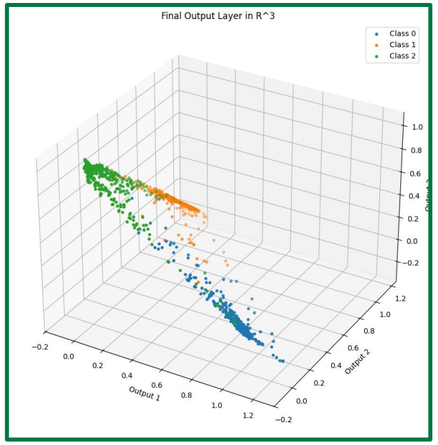
Truncation Map for Hidden Layer 2

Visualizes further separation and reshaping after the second hidden layer.



Truncation Map for Hidden Layer 3

Highlights the nearly linearly separable structure after the third hidden layer.



Final Network Output

Displays the final network output in output space R^3 with class clusters.

2 Visualizing the Data Changing Through The Layers

Global PCA – Main Steps

Step No.1 - PCA on Original Data

- Perform PCA on the Original Training Data (Global PCA) and prepare readable labels for visualization

```
# ------------------------------------------------------------
# Prepare full training data as one big matrix
# ------------------------------------------------------------
X_train = []
y_train = []

for batch, labels in train_loader:
    X_train.append(batch)
    y_train.append(labels)

X_train = torch.cat(X_train, dim=0).numpy()
y_train = torch.cat(y_train, dim=0).numpy()

# ------------------------------------------------------------
# Do PCA on original data
# ------------------------------------------------------------
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)

# For label coloring
inv_map = {v: k for k, v in label_map.items()}
label_names = [f"{inv_map[y]} (→ {y})" for y in y_train]
```

Step No.2 - Compute Truncated Data

- Compute the truncated datasets for each layer using the theoretical truncation map definition and store them for further analysis.

```
# ------------------------------------------------------------
# Compute truncated data using truncation map definition
# ------------------------------------------------------------

X_trunc_chain = []  # List to store truncated data for each layer

for ell in range(Q):
    # Get cumulative parameters for layer ell: W^(ell) and b^(ell)
    Wcum = W_cum_chain[ell].detach().cpu().numpy()
    bcum = b_cum_chain[ell].detach().cpu().numpy()

    # Compute pseudoinverse of W^(ell)
    W_pinv = np.linalg.pinv(Wcum)

    # Compute pre-activation: Z = X * W^T + b
    Z = X_train @ Wcum.T + bcum

    # Apply ReLU nonlinearity: max(0, Z)
    Z_relu = np.maximum(Z, 0)

    # Apply truncation map definition:
    # τ_{W,b}(x) = W^(-1) (ReLU(Wx + b) - b)
    X_trunc = (W_pinv @ (Z_relu.T - bcum[:, np.newaxis])).T

    # Store truncated data for this layer
    X_trunc_chain.append(X_trunc)

    # Print shape for confirmation
    print(f"Truncated data for layer {ell+1}: {X_trunc.shape}")
```

Step No.3 - Project Truncated Data on PCA of Original Data

- Project each truncated dataset onto the same Global PCA frame and plot them to visualize how the original variance directions evolve across layers.

```
# ------------------------------------------------------------
# Project truncated data in same PCA frame & plot (2x2 grid)
# ------------------------------------------------------------

for ell, X_trunc in enumerate(X_trunc_chain):
    X_pca = pca.transform(X_trunc)
    df_trunc = pd.DataFrame({
        "PCA1": X_pca[:, 0],
        "PCA2": X_pca[:, 1],
        "Label": label_names
    })
    sns.scatterplot(
        data=df_trunc, x="PCA1", y="PCA2",
        hue="Label", palette='Set1', s=10, alpha=0.7, ax=axes[ell+1]
    )
    axes[ell+1].set_title(f"(Global PCA) Truncated Data after Layer {ell+1}")
    axes[ell+1].grid(True)

plt.tight_layout()
plt.show()
```

Step No.4 - Plotting Final Output in R^Q

- Forward all training data through the full model to get the final output layer in its native output space R^Q, and plot it in 3D without PCA to visualize true output separation.

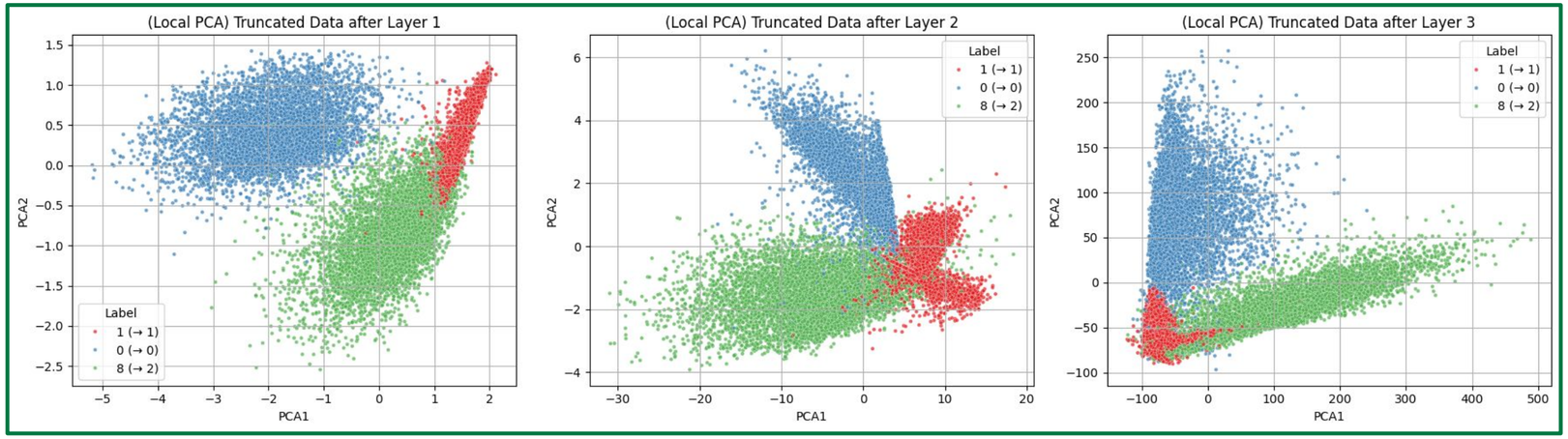
```
# ------------------------------------------------------------
# Plot final output in R^Q (no PCA needed)
# ------------------------------------------------------------

X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
model.eval() # Set model to evaluation mode and disable gradients
with torch.no_grad():
    X_out = model(X_train_tensor).numpy() # Forward pass to get final output

print(f"Output layer shape: {X_out.shape}")

from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(111, projection='3d')
for c in np.unique(y_train): # Loop through each class to plot its points
    idx = y_train == c
    ax.scatter(
        X_out[idx, 0], X_out[idx, 1], X_out[idx, 2],
        label=f"Class {c}", s=10
    )
```

2 Visualizing the Data Changing Through The Layers

(2) Local PCA

PCA on the Original Training Data

Shows the raw variance structure before any layer transformations



Truncation Map for Hidden Layer 1

Reveals how the first layer reorients the local data geometry

Truncation Map for Hidden Layer 2

Illustrates further separation along new local directions

Truncation Map for Hidden Layer 3

Highlights how the final hidden layer sharpens class boundaries

2 Visualizing the Data Changing Through The Layers

Local PCA – Main Steps

Step No.1 - PCA on Original Data

- Perform PCA on the Original Training Data (Global PCA) and prepare readable labels for visualization

```
# ------------------------------------------------------------
# Global PCA on original training data
# ------------------------------------------------------------

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)

plt.figure(figsize=(8, 6))
df_orig = pd.DataFrame({
    "PCA1": X_train_pca[:, 0],
    "PCA2": X_train_pca[:, 1],
    "Label": label_names
})
sns.scatterplot(
    data=df_orig, x="PCA1", y="PCA2",
    hue="Label", palette='Set1', s=10, alpha=0.7
)
plt.title("Original Training Data (PCA)")
plt.grid(True)
plt.show()
```

Step No.2 - Compute Truncated Data

- Compute the truncated datasets for each layer using the theoretical truncation map definition and store them for further analysis.

```
# ------------------------------------------------------------
# Compute truncated data using truncation map definition
# ------------------------------------------------------------

X_trunc_chain = []  # List to store truncated data for each layer

for ell in range(Q):
    # Get cumulative parameters for layer ell: W^(ell) and b^(ell)
    Wcum = W_cum_chain[ell].detach().cpu().numpy()
    bcum = b_cum_chain[ell].detach().cpu().numpy()

    # Compute pseudoinverse of W^(ell)
    W_pinv = np.linalg.pinv(Wcum)

    # Compute pre-activation: Z = X * W^T + b
    Z = X_train @ Wcum.T + bcum

    # Apply ReLU nonlinearity: max(0, Z)
    Z_relu = np.maximum(Z, 0)

    # Apply truncation map definition:
    # τ_{W,b}(x) = W^(-1) (ReLU(Wx + b) - b)
    X_trunc = (W_pinv @ (Z_relu.T - bcum[:, np.newaxis])).T

    # Store truncated data for this layer
    X_trunc_chain.append(X_trunc)

    # Print shape for confirmation
    print(f"Truncated data for layer {ell+1}: {X_trunc.shape}")
```

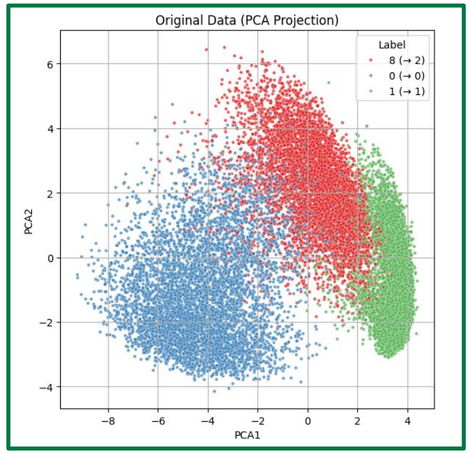
Step No.3 - Project Each Truncated Dataset onto Its Own Local PCA

- Project each layer's truncated dataset using its own Local PCA to visualize how the local variance structure evolves across layers.

```
# ------------------------------------------------------------
# Local PCA on each truncated dataset
# ------------------------------------------------------------
fig, axes = plt.subplots(1, Q, figsize=(6 * Q, 5))
if Q == 1:
    axes = [axes]

for ell, X_trunc in enumerate(X_trunc_chain):
    local_pca = PCA(n_components=2)
    X_trunc_pca = local_pca.fit_transform(X_trunc)

    df_trunc_local = pd.DataFrame({
        "PCA1": X_trunc_pca[:, 0],
        "PCA2": X_trunc_pca[:, 1],
        "Label": label_names
    })

    sns.scatterplot(
        data=df_trunc_local, x="PCA1", y="PCA2",
        hue="Label", palette='Set1', s=10, alpha=0.7, ax=axes[ell]
    )
    axes[ell].set_title(f"(Local PCA) Truncated Data after Layer {ell+1}")
    axes[ell].grid(True)

plt.tight_layout()
plt.show()
```

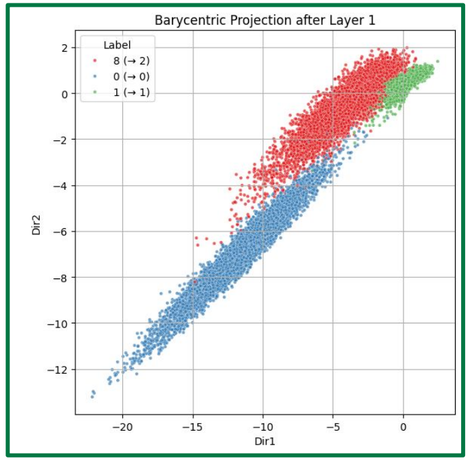
2 Visualizing the Data Changing Through The Layers

(3) Barycentric Projection



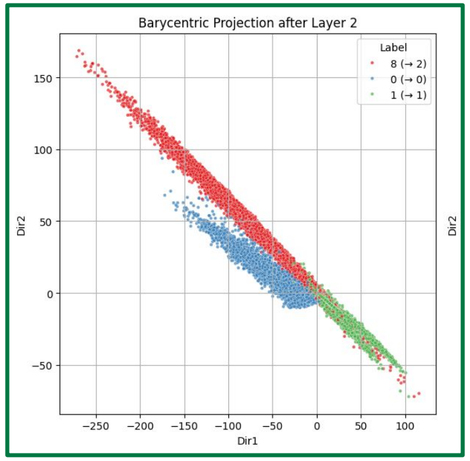
PCA on the Original Training Data

The raw training data is projected into 2D using PCA to show its initial spread.



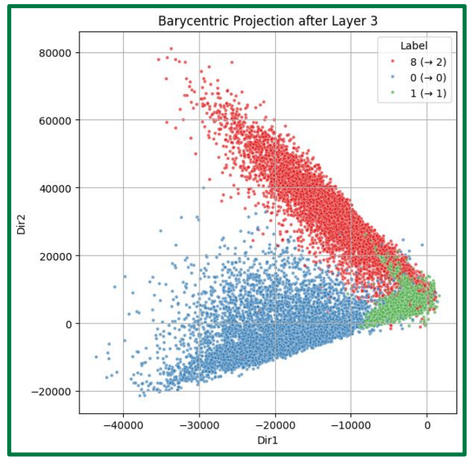
Barycentric Projection for Hidden Layer 1

The first hidden layer begins shaping the data into cone-like regions toward separation.



Barycentric Projection for Hidden Layer 2

The second layer pushes the data further apart along directions defined by class means.



Barycentric Projection for Hidden Layer 3

The final layer's cones clearly separate classes, showing near-linear separability.

2 Visualizing the Data Changing Through The Layers

(3) Barycentric Projection – Main Steps

Step No.1 - PCA on Original Data

- Perform PCA on the Original Training Data (Global PCA) and prepare readable labels for visualization

```
# ------------------------------------------------------------
# Prepare full training data as one big matrix
# ------------------------------------------------------------
X_train = []
y_train = []

for batch, labels in train_loader:
    X_train.append(batch)
    y_train.append(labels)

X_train = torch.cat(X_train, dim=0).numpy()
y_train = torch.cat(y_train, dim=0).numpy()

# For label coloring
inv_map = {v: k for k, v in label_map.items()}
label_names = [f"{inv_map[y]} (→ {y})" for y in y_train]


# ------------------------------------------------------------
# Global PCA on original data (optional baseline for comparison)
# ------------------------------------------------------------
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train)
```

Step No.2 - Compute Truncated Data

- Compute the truncated datasets for each layer using the theoretical truncation map definition and store them for further analysis.

```
# ------------------------------------------------------------
# Compute truncated data using truncation map definition
# ------------------------------------------------------------

X_trunc_chain = []  # List to store truncated data for each layer

for ell in range(Q):
    # Get cumulative parameters for layer ell: W^(ell) and b^(ell)
    Wcum = W_cum_chain[ell].detach().cpu().numpy()
    bcum = b_cum_chain[ell].detach().cpu().numpy()

    # Compute pseudoinverse of W^(ell)
    W_pinv = np.linalg.pinv(Wcum)

    # Compute pre-activation: Z = X * W^T + b
    Z = X_train @ Wcum.T + bcum

    # Apply ReLU nonlinearity: max(0, Z)
    Z_relu = np.maximum(Z, 0)

    # Apply truncation map definition:
    # τ_{W,b}(x) = W^(-1) (ReLU(Wx + b) - b)
    X_trunc = (W_pinv @ (Z_relu.T - bcum[:, np.newaxis])).T

    # Store truncated data for this layer
    X_trunc_chain.append(X_trunc)

    # Print shape for confirmation
    print(f"Truncated data for layer {ell+1}: {X_trunc.shape}")
```

Step No.3 - Compute Barycentric Projection

- This loop computes class means for each truncated dataset, forms mean-difference directions, and projects all data points onto these directions to visualize how well each layer separates class centers.

```
# ------------------------------------------------------------
# Project each truncated dataset using barycentric projection
# ------------------------------------------------------------

class_labels = np.unique(y_train)

for ell, X_trunc in enumerate(X_trunc_chain):
    # 1) Compute mean vector for each class in truncated data
    means = []
    for c in class_labels:
        means.append(X_trunc[y_train == c].mean(axis=0))  # shape: (d,)
    means = np.vstack(means)  # shape: (C, d)

    # 2) Choose reference mean (e.g., class 0)
    ref = means[0]

    # 3) Form mean-difference directions: w_c = μ_c - μ_ref
    W_proj = []
    for i in range(1, len(means)):
        W_proj.append(means[i] - ref)
    W_proj = np.vstack(W_proj).T  # shape: (d, C-1)

    # 4) Project each data point: z_i = x_i · W_proj
    X_proj = X_trunc @ W_proj  # shape: (N, C-1)
```

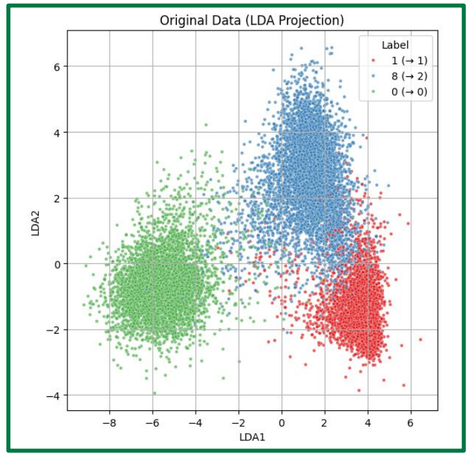
Step No. 4 - Pad, Project, and Plot Barycentric

- If there's only one mean-difference direction, it pads the projection to 2D and then plots all points in mean-difference space to show class separation.

```
    # 5) If only 1 direction, pad to 2D for plotting
    if X_proj.shape[1] == 1:
        X_proj = np.hstack([X_proj, np.zeros_like(X_proj)])

    # 6) Plot projected data in mean-difference coordinates
    df_bary = pd.DataFrame({
        "Dir1": X_proj[:, 0],
        "Dir2": X_proj[:, 1],
        "Label": label_names
    })
    sns.scatterplot(
        data=df_bary, x="Dir1", y="Dir2",
        hue="Label", palette='Set1', s=10, alpha=0.7, ax=axes[ell+1]
    )
    axes[ell+1].set_title(f"Barycentric Projection after Layer {ell+1}")
    axes[ell+1].grid(True)

plt.tight_layout()
plt.show()
```

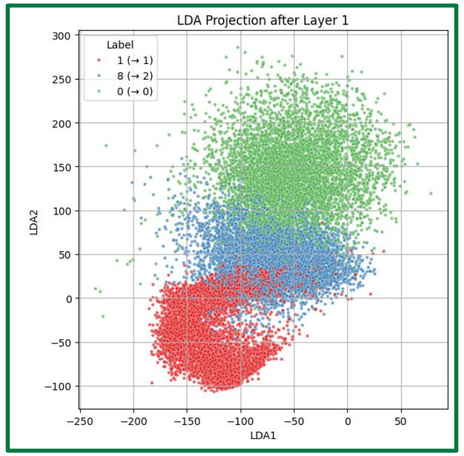
2 Visualizing the Data Changing Through The Layers

(4) Global LDA Projection



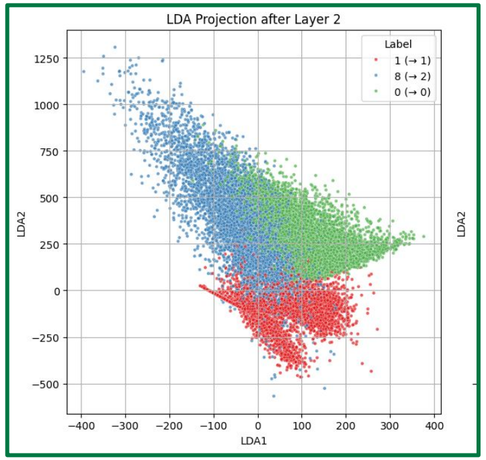
LDA on the Original Training Data

Shows the best possible linear separation of the raw data using LDA directions.



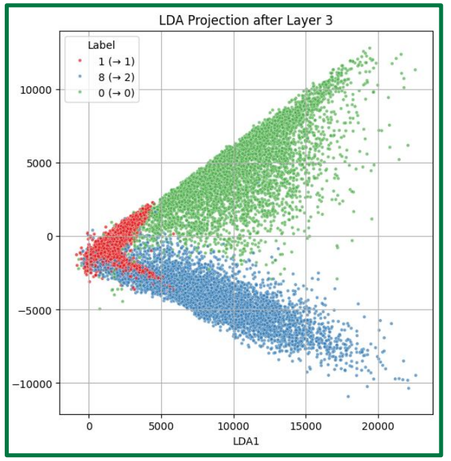
LDA Projection for Hidden Layer 1

Visualizes how the first layer begins to push data toward the learned cone structure.



LDA Projection for Hidden Layer 2

Highlights further unfolding and stretching along directions that improve separation.



LDA Projection for Hidden Layer 3

Reveals the final piecewise linear cones that make the classes linearly separable.

2 Visualizing the Data Changing Through The Layers

(4) Global LDA Projection – Main Steps

Step No.1 - PCA on Original Data

- Perform LDA on the Original Training Data (Global LDA).

```
# ------------------------------------------------------------
# Fit LDA on original training data
# ------------------------------------------------------------
lda = LDA(n_components=2)
X_train_lda = lda.fit_transform(X_train, y_train)

# ------------------------------------------------------------
# Set up plot grid: Original + each layer in same LDA basis
# ------------------------------------------------------------
fig, axes = plt.subplots(2, 2, figsize=(12, 12))
axes = axes.flatten()

# ------------------------------------------------------------
# Plot original data in LDA space
# ------------------------------------------------------------
df_orig_lda = pd.DataFrame({
    "LDA1": X_train_lda[:, 0],
    "LDA2": X_train_lda[:, 1],
    "Label": label_names
})
sns.scatterplot(
    data=df_orig_lda, x="LDA1", y="LDA2",
    hue="Label", palette='Set1', s=10, alpha=0.7, ax=axes[0]
)
axes[0].set_title("Original Data (LDA Projection)")
axes[0].grid(True)
```

Step No.2 - Compute Truncated Data

- Compute the truncated datasets for each layer using the theoretical truncation map definition and store them for further analysis.

```
# ------------------------------------------------------------
# Compute truncated data using truncation map definition
# ------------------------------------------------------------

X_trunc_chain = []  # List to store truncated data for each layer

for ell in range(Q):
    # Get cumulative parameters for layer ell: W^(ell) and b^(ell)
    Wcum = W_cum_chain[ell].detach().cpu().numpy()
    bcum = b_cum_chain[ell].detach().cpu().numpy()

    # Compute pseudoinverse of W^(ell)
    W_pinv = np.linalg.pinv(Wcum)

    # Compute pre-activation: Z = X * W^T + b
    Z = X_train @ Wcum.T + bcum

    # Apply ReLU nonlinearity: max(0, Z)
    Z_relu = np.maximum(Z, 0)

    # Apply truncation map definition:
    # τ_{W,b}(x) = W^(-1) (ReLU(Wx + b) - b)
    X_trunc = (W_pinv @ (Z_relu.T - bcum[:, np.newaxis])).T

    # Store truncated data for this layer
    X_trunc_chain.append(X_trunc)

    # Print shape for confirmation
    print(f"Truncated data for layer {ell+1}: {X_trunc.shape}")
```

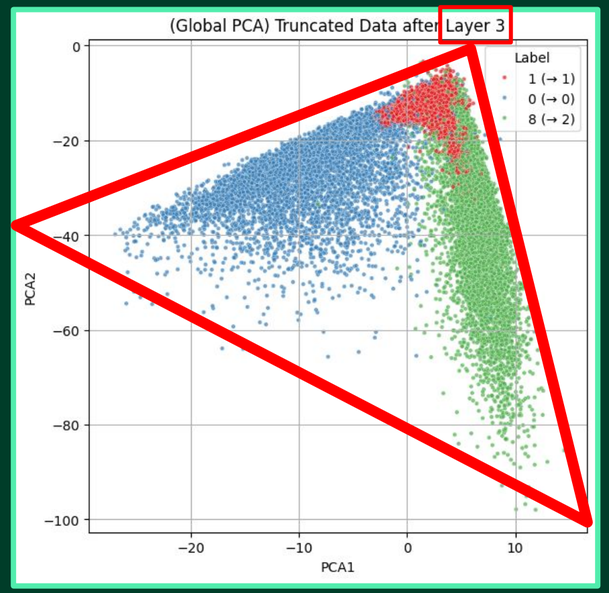
Step No.3 - Project Truncated Data on LDA of Original Data

- Project each truncated dataset onto the LDA directions computed from the original training data to visualize how the hidden layers reshape the data toward linear separability.

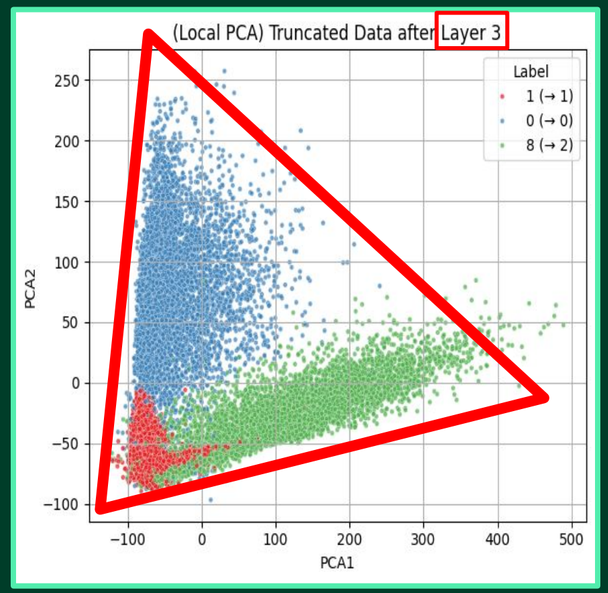
```
# ------------------------------------------------------------
# Project each truncated dataset using same LDA transform
# ------------------------------------------------------------
for ell, X_trunc in enumerate(X_trunc_chain):
    X_trunc_lda = lda.transform(X_trunc)

    # If only 1 direction, pad to 2D for consistent plotting
    if X_trunc_lda.shape[1] == 1:
        X_trunc_lda = np.hstack([X_trunc_lda, np.zeros_like(X_trunc_lda)])

    df_trunc_lda = pd.DataFrame({
        "LDA1": X_trunc_lda[:, 0],
        "LDA2": X_trunc_lda[:, 1],
        "Label": label_names
    })
    sns.scatterplot(
        data=df_trunc_lda, x="LDA1", y="LDA2",
        hue="Label", palette='Set1', s=10, alpha=0.7, ax=axes[ell+1]
    )
    axes[ell+1].set_title(f"LDA Projection after Layer {ell+1}")
    axes[ell+1].grid(True)

plt.tight_layout()
plt.show()
```

2 Visualizing the Data Changing Through The Layers

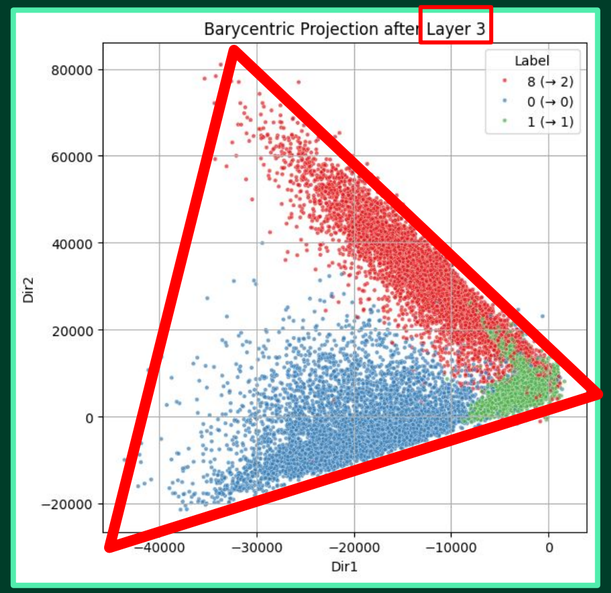
Note: I wanted to quickly expand upon an observation I noticed through the various types of projections.



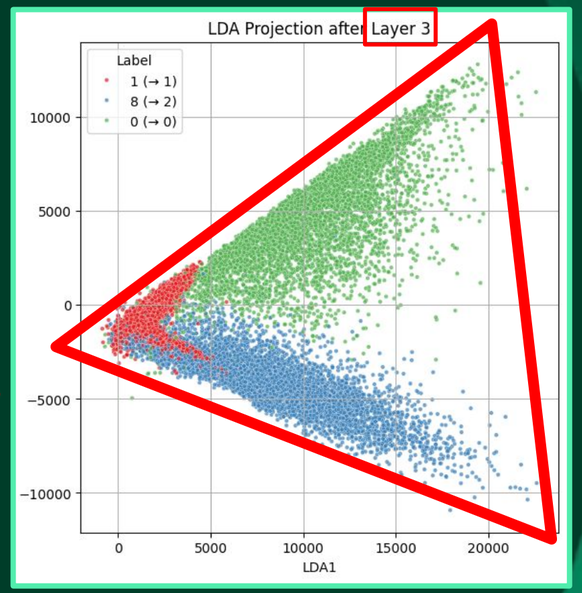
Global PCA



Local PCA



Barycentric Projection (Class Means)



Global LDA

- **Although I can't say with full certainty that I get these shapes every single time I rerun the code with a clean kernel, I usually see this wedge-shaped (cone-like ?) structure appear after Layer 3 for the truncated data. I've checked this across all projections, and in each one the hidden layers push the data into piecewise linear cones. Each cone corresponds to an activation region of the ReLU network, and the widening fans in deeper layers show how the network achieves linearly separable regions by Layer 3. The truncation maps make these cones visible in the original input coordinates.**

- **I could be completely wrong about this, but I noticed this recurring pattern in the third layer, so I wanted to highlight this observation.**

3 Implementation with Synthetic Data

(So it's more of a study on cones and truncation maps rather than MNIST)

- **Goal:** Generate simple synthetic data in **R^3** to train a ReLU network and **visualize cones & truncation maps** clearly.
- **Why:** Low dimension means **no projection loss**, so geometry is easy to see.
- **Where Do I Start?:**
  - Use **Q=3 classes**, each cluster well-separated.
  - Example:
    - 3 spheres/balls centered at **(1,0,0)**, **(0,1,0)**, **(0,0,1)**.
    - Or use **Gaussian mixture model** with small standard deviation.
- **Optional variations:**
  - 3 classes that are **SLS but not clustered**.
  - 2 classes **not linearly separable**, but **not concentric** (like Fig 1a in [Ewa25]).
  - 2 classes that are **concentric** (like Fig 1b in [Ewa25]).
- **Next step:** Train, extract **cumulative parameters**, plot **cones & truncations** just like for MNIST.

3 Implementation with Synthetic Data

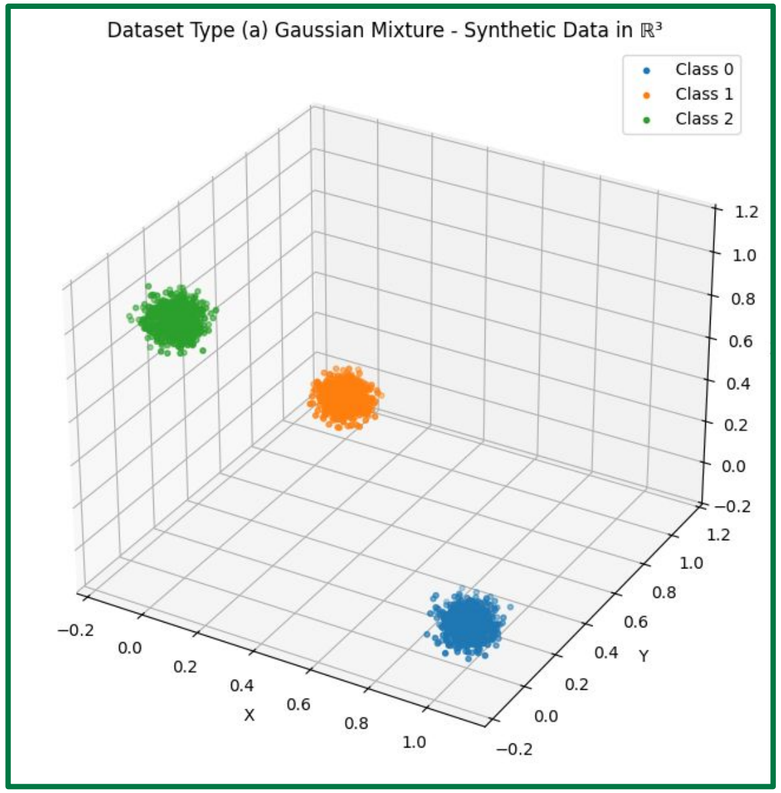
Step No.1 – Generate Different Types of Synthetic Data

(a) Gaussian Mixture Dataset

This sets the means for each Gaussian, each center creates one "bell-shaped" cluster in the mixture.

```python
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader

# Create synthetic data: 3 well-separated 3D Gaussian blobs
X, y = make_blobs(
    n_samples=3000,
    centers=[
        [1.0, 0.0, 0.0],
        [0.0, 1.0, 0.0],
        [0.0, 0.0, 1.0]
    ],
    cluster_std=0.05,
    n_features=3,
    random_state=42
)
```

This controls the spread of each Gaussian, making each blob tight or wide around its center.



Data was generated with cluster standard deviation (cluster_std) of 0.05

3 Implementation with Synthetic Data

Step No.1 – Generate Different Types of Synthetic Data

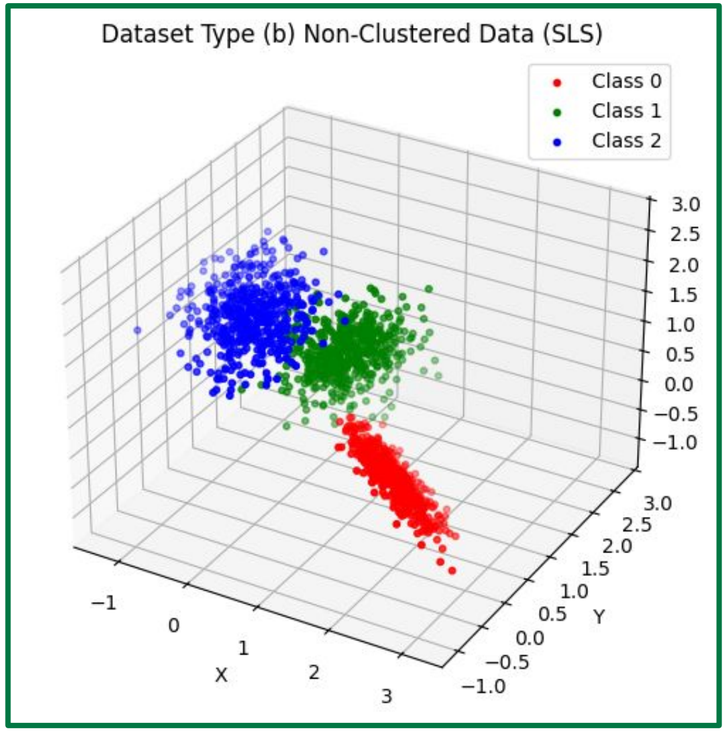
(b) SLS Non-Clustered Dataset

Defines three independent directions in 3D space, each corresponding to a distinct plane orientation.

Generates 3D class points on a plane orthogonal to a direction, centered at a given point, with labels.

Generates SLS data with 3 class clouds on orthogonal planes, ideal for layer-wise separability.

```python
#------------------------------------------------------------------------------
# Create Synthetic Data
#------------------------------------------------------------------------------

n_points = 500  # Number of data points per class

# Three direction vectors in ℝ³, which will correspond to unique separating hyperplanes
dir1 = np.array([1, 1, 1])
dir2 = np.array([-1, 1, 0])
dir3 = np.array([0, -1, 1])

# Normalize the direction vectors to ensure they have unit length
dir1, dir2, dir3 = [d / np.linalg.norm(d) for d in [dir1, dir2, dir3]]

# Function to generate data points lying in a plane orthogonal to the given direction
def generate_points(center, direction, label):
    # Generate random noise vectors in ℝ³
    noise = np.random.randn(n_points, 3)

    # Remove the component of noise in the direction vector to make it orthogonal
    noise -= (noise @ direction)[:, None] * direction

    # Add the (scaled) noise to a given center to spread it out in the orthogonal plane
    points = center + 0.4 * noise

    # Return the generated points and their associated class label
    return points, np.full(n_points, label)

# Generate data for each class centered at different locations in ℝ³
X1, y1 = generate_points(np.array([2, 0, 0]), dir1, 0)
X2, y2 = generate_points(np.array([0, 2, 0]), dir2, 1)
X3, y3 = generate_points(np.array([0, 0, 2]), dir3, 2)

# Stack the data points from all classes into a single feature matrix
X = np.vstack([X1, X2, X3])

# Concatenate all labels into a single label vector
y = np.concatenate([y1, y2, y3])
```

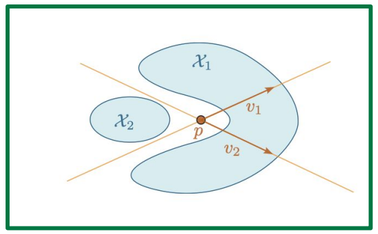


Data was generated with noise of 0.04

3 Implementation with Synthetic Data

Step No.1 – Generate Different Types of Synthetic Data

(c) 2 Non-SLS + Non-Concentric Classes (like Figure 1a in [Ewa25]).

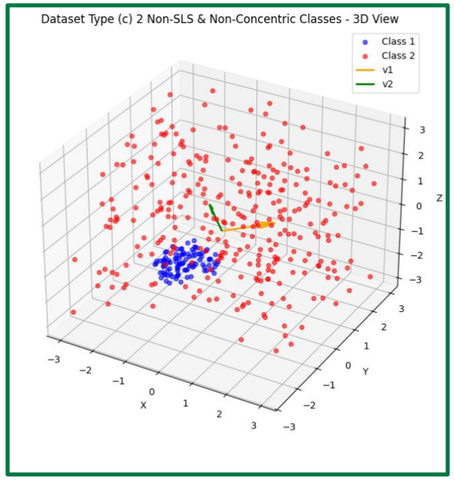


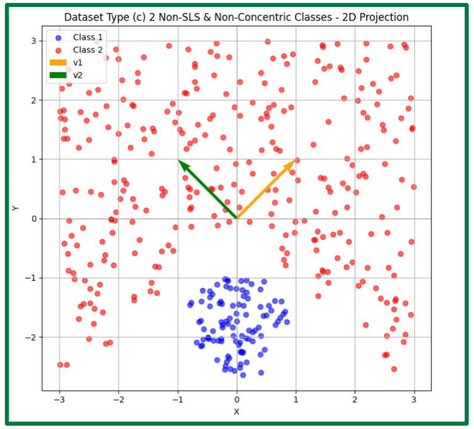
Sets up the cone geometry using a base point and two directional vectors.

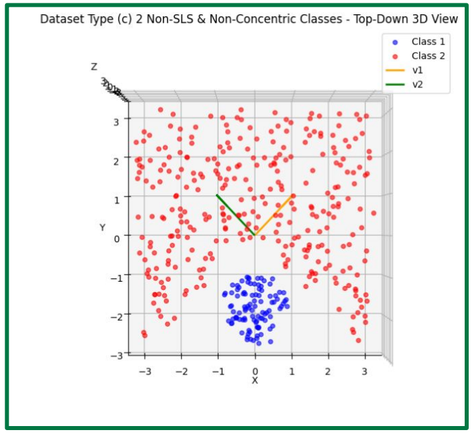
Creates Class 1 inside the cone and Class 2 outside the cone using structured sampling.

Labels and merges both classes into a unified dataset for classification or visualization.

```python
#-----------------------------------------------------------------------------
# Create Synthetic Data
#-----------------------------------------------------------------------------

# Define the cone apex and two generating vectors
p = np.array([0, 0, 0])
v1 = np.array([1, 1, 0])
v2 = np.array([-1, 1, 0])

# Generate Class 1 points: inside the cone, with negative combinations of v1 and v2 + small noise
def generate_class1(n=100):
    coeffs = -np.abs(np.random.rand(n, 2)) - 0.35  # Ensure negative direction
    return p + coeffs[:, 0][:, None]*v1 + coeffs[:, 1][:, None]*v2 + 0.1*np.random.randn(n, 3)

# Generate Class 2 points: outside the cone by rejecting points with all-negative coefficients
def generate_class2(n=100):
    pts = []
    while len(pts) < n:
        x = np.random.uniform(-3, 3, 3)  # Sample randomly from a cube
        A = np.column_stack((v1, v2))
        coeffs = np.linalg.lstsq(A, x - p, rcond=None)[0]  # Solve for cone coords
        if not np.all(coeffs <= 0):  # Accept only points not inside the cone
            pts.append(x)
    return np.array(pts)

# Generate data
X1 = generate_class1(100)     # Class 1: inside cone
X2 = generate_class2(300)     # Class 2: outside cone

# Assign class labels
y1 = np.zeros(X1.shape[0])    # Label 0 for Class 1
y2 = np.ones(X2.shape[0])     # Label 1 for Class 2

# Combine into a full dataset
X = np.vstack([X1, X2])       # Stack features
y = np.concatenate([y1, y2])  # Stack labels
```



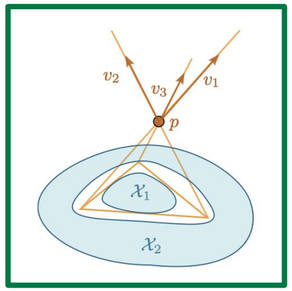




3 Implementation with Synthetic Data

Step No.1 – Generate Different Types of Synthetic Data

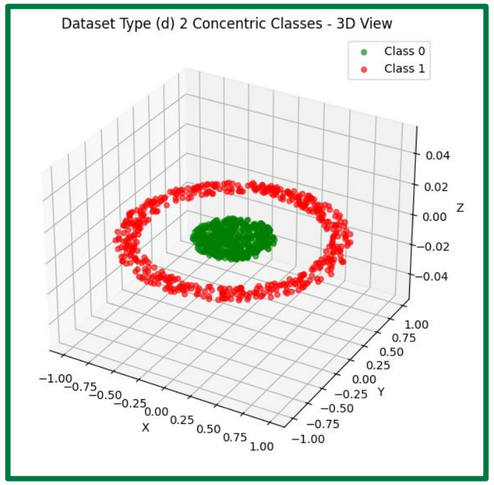
(d) 2 Concentric Classes (like Figure 1b in [Ewa25]).

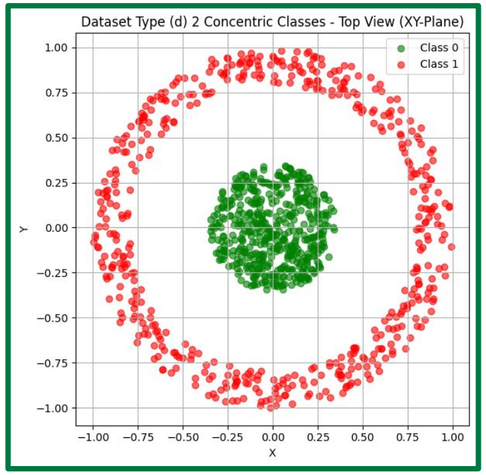


This function samples points uniformly within a circular disk using polar coordinates.

This function samples points uniformly from a ring-shaped region between two radii, ensuring area-uniformity.

```python
#-----------------------------------------------------------------------------
# Create Synthetic Data (Filled Disk + Ring)
#-----------------------------------------------------------------------------

# Function to generate points uniformly inside a filled disk of a given radius
def generate_filled_disk(radius=0.35, n_samples=500):
    # Sample radius with area-uniformity (r^2 scaling)
    r = np.sqrt(np.random.uniform(0, radius**2, n_samples))
    # Sample angle uniformly from 0 to 2π
    theta = np.random.uniform(0, 2 * np.pi, n_samples)
    # Convert polar to Cartesian coordinates
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return np.column_stack([x, y])  # Shape: (n_samples, 2)

# Function to generate points uniformly in a ring between inner and outer radii
def generate_ring(inner=0.8, outer=1.0, n_samples=500):
    # Sample radius uniformly between inner and outer radii (area-corrected)
    r = np.sqrt(np.random.uniform(inner**2, outer**2, n_samples))
    theta = np.random.uniform(0, 2 * np.pi, n_samples)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return np.column_stack([x, y])  # Shape: (n_samples, 2)
```





3 Implementation with Synthetic Data

Step No.2 - Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.1) Designing a Pipeline

Configure Based On Which Dataset Was Chosen

- This block selects and loads the appropriate synthetic dataset (Gaussian, SLS, non-concentric, or concentric) based on the chosen dataset_type, extracting its data, labels, and train/test loaders.

```
if dataset_type == 'a':
    print('You have chosen: (a) Gaussian Mixture - Synthetic Data in R³')
    data = generate_gaussian_mixture_data()
    Q = data['Q']
    X = data['X']
    y = data['y']
    X_train = data['X_train']
    y_train = data['y_train']
    train_loader = data['train_loader']
    test_loader = data['test_loader']
elif dataset_type == 'b':
    print('You have chosen: (b) Non-Clustered Data (SLS)')
    data = generate_non_clustered_sls_data()
    Q = data['Q']
    X = data['X']
    y = data['y']
    X_train = data['X_train']
    y_train = data['y_train']
    train_loader = data['train_loader']
    test_loader = data['test_loader']
elif dataset_type == 'c':
    print('You have chosen: (c) 2 Non-SLS & Non-Concentric Classes')
    data = generate_no_SLS_no_concentric_data()
    Q = data['Q']
    X = data['X']
    y = data['y']
    X_train = data['X_train']
    y_train = data['y_train']
    train_loader = data['train_loader']
    test_loader = data['test_loader']
elif dataset_type == 'd':
    print('You have chosen: (d) 2 Concentric Classes')
    data = generate_concentric_data()
    Q = data['Q']
    X = data['X']
    y = data['y']
    X_train = data['X_train']
    y_train = data['y_train']
    train_loader = data['train_loader']
    test_loader = data['test_loader']
else:
    raise ValueError("Invalid dataset_type. Choose from {'a', 'b', 'c', 'd'}")
```

Train a Neural Network [based on CE24 - Part 1 -

- This code defines a ReLU feedforward neural network with input dimension 3, Q hidden layers of size 3, and an output layer of size Q, where Q is the number of classes in the selected dataset.

```
input_dim = 3  # Always 3
hidden_dims = [3] * Q  # Q hidden layers
output_dim = Q  # Either 2 or 3 depending on dataset

# ReLU Feedforward Network
class TruncationNet(nn.Module):
    def __init__(self, input_dim=input_dim, hidden_dims=hidden_dims, output_dim=output_dim):
        super().__init__()
        dims = [input_dim] + hidden_dims + [output_dim]
        self.layers = nn.ModuleList([
            nn.Linear(dims[i], dims[i+1]) for i in range(len(dims)-1)
        ])

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = F.relu(layer(x))
        return self.layers[-1](x)

model = TruncationNet()
#print(model)
```

Train a Neural Network [based on CE24 - Part 2 -

- This code trains the neural network for 200 epochs using mean squared error (MSE) loss between the model's output and one-hot encoded labels, optimizing with stochastic gradient descent (SGD) and printing the loss every 10 epochs.

```
def one_hot(labels, num_classes):
    return torch.nn.functional.one_hot(labels, num_classes=num_classes).float()

criterion = nn.MSELoss()  # squared L2 loss, matches the paper
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

epochs = 200
for epoch in range(epochs):
    model.train()
    for data, target in train_loader:
        optimizer.zero_grad()

        # One-hot encode the labels
        target_onehot = one_hot(target, num_classes=Q)

        output = model(data)
        loss = criterion(output, target_onehot)

        loss.backward()
        optimizer.step()

    if (epoch+1) % 10 == 0 or epoch == 0:
        print(f'Epoch {epoch+1}, MSE Loss: {loss.item():.4f}')

#print("Training done with MSELoss!")
print(f"Final MSE Loss: {loss.item():.6f}")
print('\n---------- END TRAINING MODEL ----------')
```

3 Implementation with Synthetic Data

Step No.2 – Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.1) Designing a Pipeline – continued...

Train a Neural Network [based on CE24 – Part 3 –]

- This code evaluates the trained model on the test set by comparing predicted class labels to true labels and reports the overall test accuracy.

```
# --- Testing Section ---

print('---------- BEGIN TESTING MODEL ----------\n')
model.eval()

correct = 0
total = 0

with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        pred = output.argmax(dim=1)
        correct += (pred == target).sum().item()
        total += target.size(0)

test_acc = correct / total
print(f"Test Accuracy: {test_acc * 100:.2f}%")
print('\n---------- END TESTING MODEL ----------\n')
```

Determine Cumulative Parameters $W(l), b(l)$

- This block computes the cumulative weight and bias parameters for each layer of the neural network by recursively chaining linear transformations.

```
Ws = []
bs = []
for layer in model.layers:
    Ws.append(layer.weight.detach().clone())
    bs.append(layer.bias.detach().clone())

L = len(Ws)
L1 = len(bs)

W_cum_chain = []
b_cum_chain = []

for ell in range(L):
    if ell == 0:
        Wcum = Ws[0]
        bcum = bs[0]
    else:
        Wcum = Ws[ell] @ W_cum_chain[-1]

        btemp = 0
        for k in range(ell):
            chain = Ws[ell]
            for j in range(ell-1, k, -1):
                chain = chain @ Ws[j]
            btemp += chain @ bs[k]

        bcum = btemp + bs[ell]

    W_cum_chain.append(Wcum)
    b_cum_chain.append(bcum)

print('- NOTE: FINISHED WITH CALCULATING CUMULATIVE PARAMETERS \n')
```

Determine Cones – Part 1 –

- This function computes the base point and edge vectors of the cone for each layer using the pseudoinverse of the cumulative weight matrix.

```
def compute_cones(W_cum, b_cum, residual_tol=1e-4):
    cones = []
    for ell, (W, b) in enumerate(zip(W_cum, b_cum)):

        # making sure everything is pure NumPy (got warning before)
        if hasattr(W, 'detach'): W = W.detach().cpu().numpy()
        if hasattr(b, 'detach'): b = b.detach().cpu().numpy()

        # compute the pseudoinverse of W
        W_pinv = np.linalg.pinv(W)

        # compute base point: p = - W^+ b [Ewa25]
        p = -W_pinv @ b

        # compute edges: v_i = W^+ e_i for standard basis vectors e_i [Ewa25]
        m = W.shape[0]
        edges = []
        edge_residuals = []

        for i in range(m):
            ei = np.zeros(m)
            ei[i] = 1
            vi = W_pinv @ ei
            edges.append(vi)

            residual = np.linalg.norm(W @ vi - ei)
            edge_residuals.append(residual)

        cones.append({'base': p, 'edges': edges})
```

3 Implementation with Synthetic Data

Step No.2 – Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.1) Designing a Pipeline – continued...

Determine Cones
– Part 2 –

- This block verifies Lemma 2.1 by checking whether the weight matrix is surjective, the base point satisfies $Wp+b\approx 0$, and all edge vectors accurately invert standard basis directions.

```
    # Surjectivity check (Lemma 2.1)
    rank = np.linalg.matrix_rank(W)
    rows, cols = W.shape
    print(f"Rank(W) = {rank} | Should be full row rank = {rows}")
    if rank == rows:
        print("✅ Surjective check passed, it is full row rank")
    else:
        print("⚠️ Not surjective")

    # Base point residual check
    base_residual = np.linalg.norm(W @ p + b)
    if base_residual <= residual_tol:
        print(f"✅ Base point residual ||Wp + b|| = {base_residual:.2e} (OK)")
    else:
        print(f"⚠️ Base point residual ||Wp + b|| = {base_residual:.2e} exceeds tolerance")

    # Edge residuals check
    edge_residuals = np.array(edge_residuals)
    num_failed_edges = np.sum(edge_residuals > residual_tol)
    mean_edge_residual = np.mean(edge_residuals)

    if num_failed_edges == 0:
      print(f"✅ All edge checks passed | Failing: {num_failed_edges}/{m} | Mean residual: {mean_edge_residual:.2e}")
    else:
      print(f"⚠️ Some edge residuals exceed tolerance | Failing: {num_failed_edges}/{m} | Mean residual: {mean_edge_residual:.2e}")

    print(f"--------------------")

    # Overall pass/fail summary
    if rank == rows and base_residual <= residual_tol and num_failed_edges == 0:
        print("✅ Lemma 2.1 conditions are met")
    else:
        print("⚠️ Lemma 2.1 conditions are not met")

print('\n---------- FINISHED COMPUTING CONES ----------\n')
return cones
```

Compute Truncated Data

- This block computes the truncated representation of the input data at each layer by applying the truncation map using the cumulative weights and biases.

```
X_trunc_chain = []  # List to store truncated data for each layer

for ell in range(Q):
    # Get cumulative parameters for layer ell: W^(ell) and b^(ell)
    Wcum = W_cum_chain[ell].detach().cpu().numpy()
    bcum = b_cum_chain[ell].detach().cpu().numpy()

    # Compute pseudoinverse of W^(ell)
    W_pinv = np.linalg.pinv(Wcum)

    # Compute pre-activation: Z = X * W^T + b
    Z = X_train @ Wcum.T + bcum

    # Apply ReLU nonlinearity: max(0, Z)
    Z_relu = np.maximum(Z, 0)

    # Apply truncation map definition:
    # τ_{W,b}(x) = W^(-1) (ReLU(Wx + b) - b)
    X_trunc = (W_pinv @ (Z_relu.T - bcum[:, np.newaxis])).T

    # Store truncated data for this layer
    X_trunc_chain.append(X_trunc)

    # Print shape for confirmation
    #print(f"Truncated data for layer {ell+1}: {X_trunc.shape}")

print('– NOTE: FINISHED COMPUTING TRUNCATED DATA FOR EACH LAYER \n')
```

Visualize Original, Truncated Layers, and Final Output Side by Side

- This part of the code shows how the input data changes after each layer by plotting the original data, the truncated versions at each layer, and the final output. It automatically adjusts between 2D and 3D plots depending on the data's dimensions.

3 Implementation with Synthetic Data

Step No.2 – Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.2) Running The Code – Dataset Type a

With hidden_dims = [3,3,3]

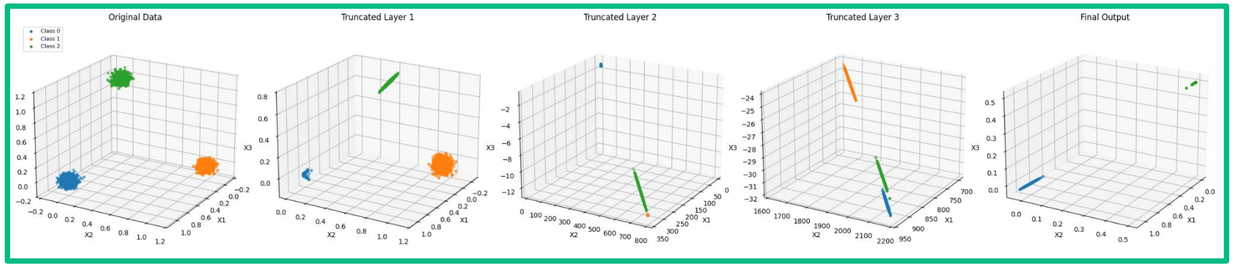
```
run_main_pipeline('a') #Gaussian Mixture Model
```

With hidden_dims = [10,10,10]

Testing Accuracy: ← But... → **Surjectivity Checks:**

```
---------- BEGIN TESTING MODEL ----------
Test Accuracy: 66.67%
---------- END TESTING MODEL ----------
```

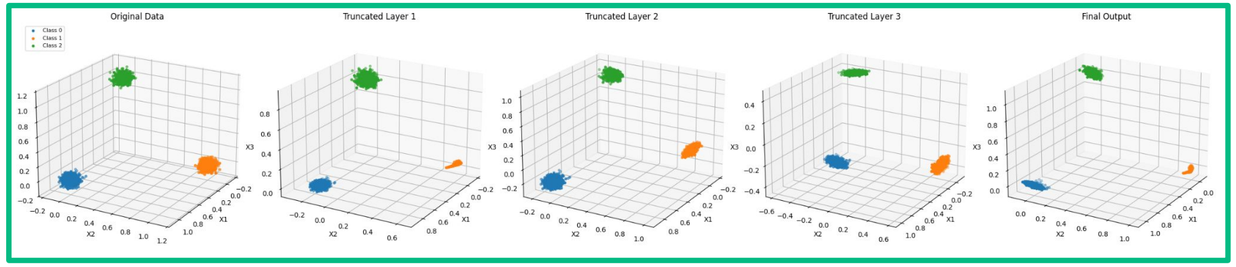
```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 8.43e-08 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 6.73e-08
----------
✅ Lemma 2.1 conditions are met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 3.13e-07 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 2.56e-07
----------
✅ Lemma 2.1 conditions are met

=== Layer 3 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 9.11e-06 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 2.63e-06
----------
✅ Lemma 2.1 conditions are met

=== Layer 4 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 2.63e-05 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 4.31e-06
----------
✅ Lemma 2.1 conditions are met
```



Testing Accuracy: → But... → **Surjectivity Checks:**

```
---------- BEGIN TESTING MODEL ----------
Test Accuracy: 100.00%
---------- END TESTING MODEL ----------
```

```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 1.04e+00 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.16e-01
----------
⚠️ Lemma 2.1 conditions are not met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 7.85e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.30e-01
----------
⚠️ Lemma 2.1 conditions are not met

=== Layer 3 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 9.11e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.21e-01
----------
⚠️ Lemma 2.1 conditions are not met

=== Layer 4 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.05e-06 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 3.88e-07
----------
✅ Lemma 2.1 conditions are met
```



3 Implementation with Synthetic Data

Step No.2 – Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.2) Running The Code – Dataset Type b

```
run_main_pipeline('b') #Non-Clustered Data (SLS)
```

With hidden_dims = [3,3,3]

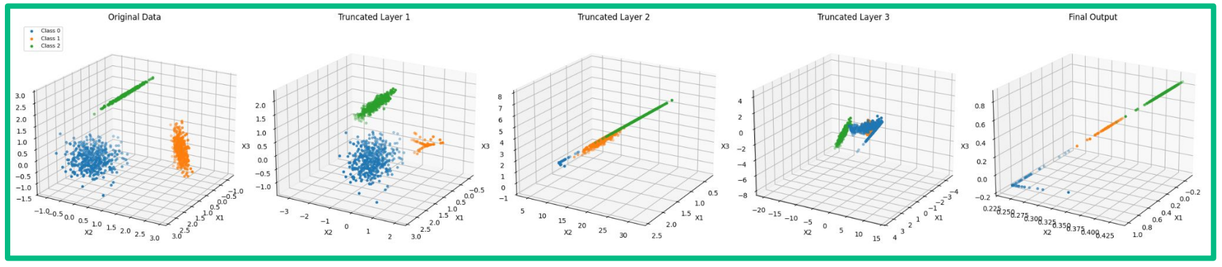
Testing Accuracy:

```
---------- BEGIN TESTING MODEL ----------
Test Accuracy: 66.67%
---------- END TESTING MODEL ----------
```

But...

Surjectivity Checks:

```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.67e-08 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 8.68e-08
--------------------
✅ Lemma 2.1 conditions are met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.33e-07 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 1.32e-07
--------------------
✅ Lemma 2.1 conditions are met

=== Layer 3 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.64e-06 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 2.67e-06
--------------------
✅ Lemma 2.1 conditions are met

=== Layer 4 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.11e-05 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 9.53e-06
--------------------
✅ Lemma 2.1 conditions are met
```



With hidden_dims = [10,10,10]

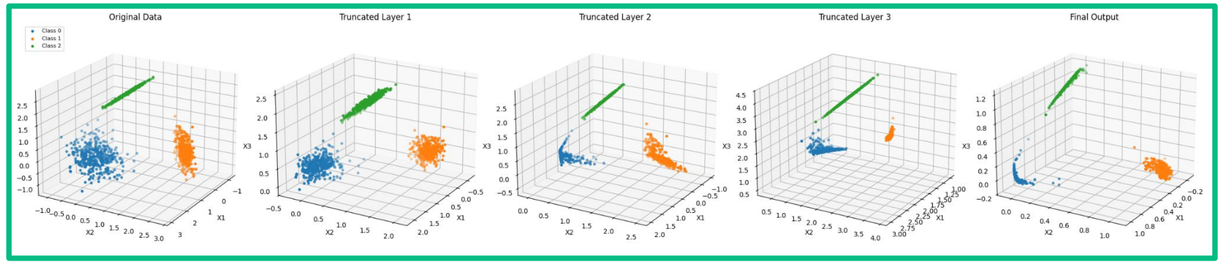
Testing Accuracy:

```
---------- BEGIN TESTING MODEL ----------
Test Accuracy: 100.00%
---------- END TESTING MODEL ----------
```

But...

Surjectivity Checks:

```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 9.37e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.33e-01
--------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 6.89e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.26e-01
--------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 3 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 7.76e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.25e-01
--------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 4 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.49e-07 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 1.31e-07
--------------------
✅ Lemma 2.1 conditions are met
```



3 Implementation with Synthetic Data

Step No.2 – Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.2) Running The Code – Dataset Type c

With hidden_dims = [3,3,3]

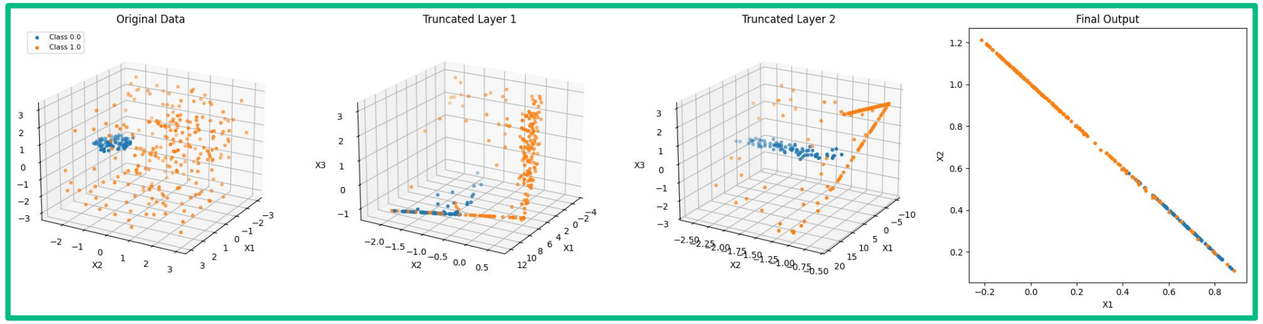
```
run_main_pipeline('c') #2 Non-SLS & Non-Concentric Classes
```

With hidden_dims = [10,10,10]

Testing Accuracy: ← But... → **Surjectivity Checks:**

```
---------- BEGIN TESTING MODEL ----------
Test Accuracy: 75.88%
---------- END TESTING MODEL ----------
```

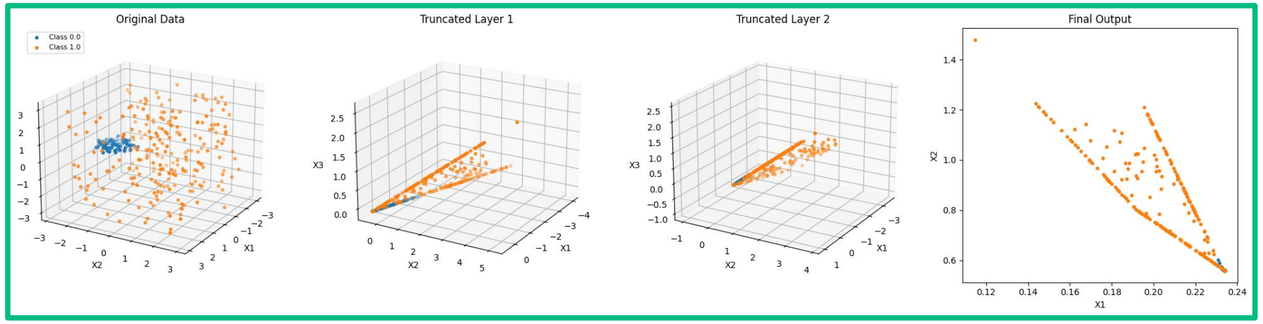
```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 2.11e-08 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 1.47e-07
------------------
✅ Lemma 2.1 conditions are met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 3.33e-08 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 8.79e-08
------------------
✅ Lemma 2.1 conditions are met

=== Layer 3 Cone Checks ===
Rank(W) = 2 | Should be full row rank = 2
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 4.69e-06 (OK)
✅ All edge checks passed | Failing: 0/2 | Mean residual: 5.08e-06
------------------
✅ Lemma 2.1 conditions are met
```



Testing Accuracy: ← But... → **Surjectivity Checks:**

```
---------- BEGIN TESTING MODEL ----------
Test Accuracy: 100.00%
---------- END TESTING MODEL ----------
```

```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 1.00e+00 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.33e-01
------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 8.31e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.27e-01
------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 3 Cone Checks ===
Rank(W) = 2 | Should be full row rank = 2
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 2.98e-08 (OK)
✅ All edge checks passed | Failing: 0/2 | Mean residual: 3.36e-08
------------------
✅ Lemma 2.1 conditions are met
```



3 Implementation with Synthetic Data

Step No.2 – Train, Extract Cumulative Parameters, Plot Cones & Truncation Maps

(2.2) Running The Code – Dataset Type d

```
run_main_pipeline('d') #2 Concentric Classes
```

With hidden_dims = [3,3,3]

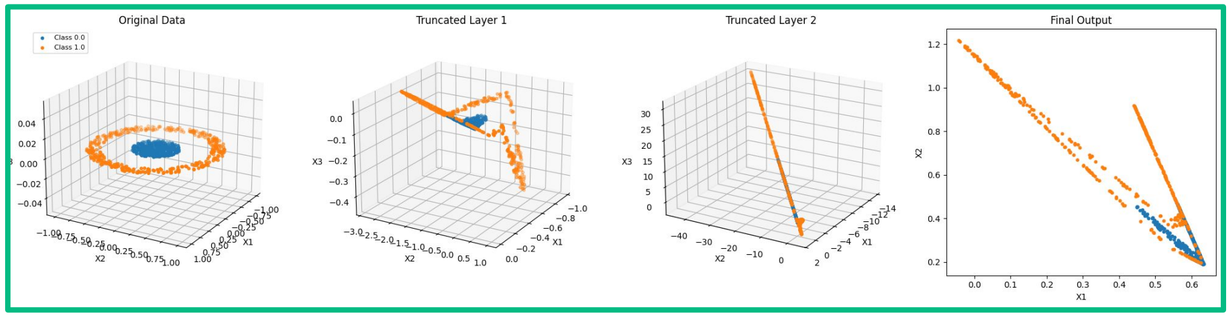
Testing Accuracy:

```
---------- BEGIN TESTING MODEL ----------

Test Accuracy: 81.50%

---------- END TESTING MODEL ------------
```

But...

Surjectivity Checks:

```
=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.34e-07 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 5.70e-08
------------------
✅ Lemma 2.1 conditions are met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 3
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 3.13e-07 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 7.73e-07
------------------
✅ Lemma 2.1 conditions are met

=== Layer 3 Cone Checks ===
Rank(W) = 2 | Should be full row rank = 2
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 5.96e-08 (OK)
✅ All edge checks passed | Failing: 0/2 | Mean residual: 9.15e-08
------------------
✅ Lemma 2.1 conditions are met
```



With hidden_dims = [10,10,10]

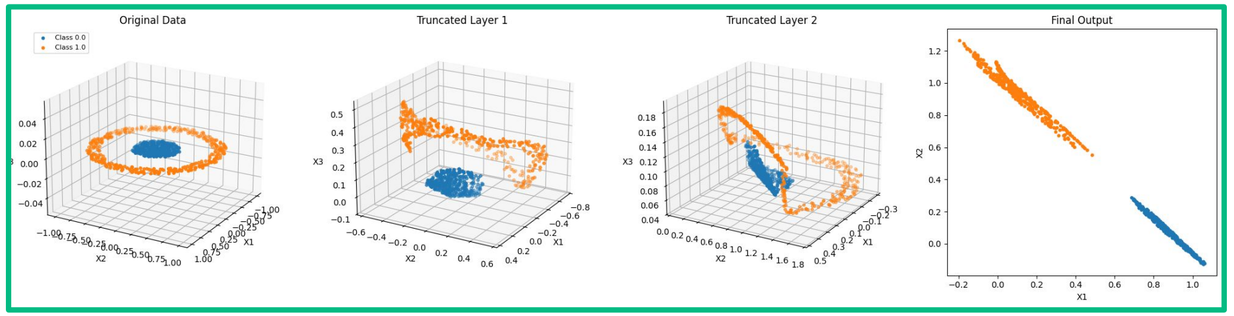
Testing Accuracy:

```
---------- BEGIN TESTING MODEL ----------

Test Accuracy: 100.00%

---------- END TESTING MODEL ------------
```

But...

Surjectivity Checks:

```
---------- END TRAINING MODEL ----------
---------- BEGIN TESTING MODEL ----------

Test Accuracy: 100.00%

---------- END TESTING MODEL ------------

- NOTE: FINISHED WITH CALCULATING CUMULATIVE PARAMETERS

---------- BEGIN COMPUTING CONES ----------

=== Layer 1 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 9.93e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.31e-01
------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 2 Cone Checks ===
Rank(W) = 3 | Should be full row rank = 10
⚠️ Not surjective
⚠️ Base point residual ||Wp + b|| = 8.54e-01 exceeds tolerance
⚠️ Some edge residuals exceed tolerance | Failing: 10/10 | Mean residual: 8.31e-01
------------------
⚠️ Lemma 2.1 conditions are not met

=== Layer 3 Cone Checks ===
Rank(W) = 2 | Should be full row rank = 2
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 1.49e-07 (OK)
✅ All edge checks passed | Failing: 0/2 | Mean residual: 4.42e-07
------------------
✅ Lemma 2.1 conditions are met

---------- FINISHED COMPUTING CONES ----------
```



3 Implementation with Synthetic Data

Step No.3 - Analyzing the Trends (What's Happening?)

- **Smaller network ([3,3,3])**:
  - **Lower accuracy**, but **clean cone structure** that **adheres** to **theoretical requirements (like surjectivity and residual tolerance)**
  - Good for geometric interpretability of learned features (as in CE24)
- **Larger network ([10,10,10])**:
  - **Perfect accuracy,** but **violates cone conditions**, base points and edge vectors fail residual checks due to rank-deficient mappings
  - Network overfits or learns too flexibly, breaking the strict geometric constraints

4 Hyperplane Separation

What's the Point?

Training networks has too little intuition

- Normally, we train a neural net and it spits out a result.
- But why it works and what it’s learning is unclear.

Goal:

To intuitively understand how a neural network separates digits by using simple geometric shapes (hyperplanes) in a high-dimensional space. This helps us "see" how digits like 0, 1, and 2 might be distinguishable just based on which parts of an image are black or white.

- Instead of using cones, use specific pixels to guess separating hyperplanes.
- Try constructing hyperplanes from filled pixels unique to one digit.
- Can be more intuitive and less restrictive than cones.

so… What are the hyperplanes doing inside the network? Can we see or guess them intuitively?

4 Hyperplane Separation

(1) Toy 5x5 Model (basic) – Main Steps

Step No.1 - Define Toy Digits

- Create 5 toy binary digit images (0–4) as numpy arrays to simulate MNIST-style patterns.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.metrics import accuracy_score
from matplotlib import colormaps


# Define toy digits (0–4)
digit_0 = np.array([
    [1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1],
    [1, 0, 0, 0, 1],
    [1, 0, 0, 0, 1],
    [1, 1, 1, 1, 1]
])

# ... (same for digits 1–4)
```

Step No.2 - Flatten Digits and Start Separation

- Flatten digits into vectors and initialize variables for separating them via unique pixel-based hyperplanes.

```
H, W = digit_0.shape
D = H * W
all_digits = [digit_0, digit_1, digit_2, digit_3, digit_4]
all_labels = np.array([0, 1, 2, 3, 4])
separation_steps = []

cmap = colormaps['tab10']
label_to_color = {label: cmap(label) for label in all_labels}
remaining_digits = list(all_digits)
remaining_labels = list(all_labels)
step = 1
```

Step No.3 - Loop Until One Digit Left

- Flatten digits into vectors and initialize variables for separating them via unique pixel-based hyperplanes.

```
while len(remaining_digits) > 1:
    print(f'\n---- STEP {step} ----\n')
    X = np.array([d.flatten() for d in remaining_digits])
    y = np.array(remaining_labels)

    # Unique pixel detection
    unique_pixels = {}
    for i in range(D):
        values = X[:, i]
        if list(values).count(1) == 1:
            digit = np.argmax(values)
            if digit not in unique_pixels:
                unique_pixels[digit] = []
            unique_pixels[digit].append(i)

    if not unique_pixels:
        break
```

Step No.4 - Visualize Remaining Digits and Pixel Indices

- Flatten digits into vectors, initialize variables and color mapping, and set up for separating digits using unique pixel-based hyperplanes while tracking actual classification accuracy.

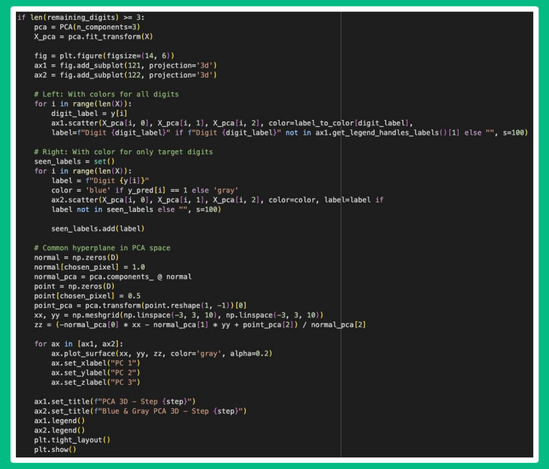
```
fig, axes = plt.subplots(1, len(remaining_digits), figsize=(2 * len(remaining_digits), 4))
titles = [f"Digit {label}" for label in remaining_labels]
for idx, ax in enumerate(axes):
    img = remaining_digits[idx]
    ax.imshow(1 - img, cmap='gray', vmin=0, vmax=1)
    ax.set_title(titles[idx])
    for i in range(H):
        for j in range(W):
            pixel_idx = i * W + j
            ax.text(j, i, str(pixel_idx), ha='center', va='center', color='cyan', fontsize=12, fontweight='bold')
    ax.axis('off')
plt.suptitle("Remaining Digits with Pixel Indices", fontsize=14)
plt.tight_layout()
plt.show()

# Print discovered hyperplanes
print("\nHyperplanes (x' = 0.5) that can uniquely separate each digit:")
for digit, indices in unique_pixels.items():
    digit_label = y[digit]
    rest_digits = [str(lbl) for lbl in y if lbl != digit_label]
    rest_str = f" (vs digits {', '.join(rest_digits)})" if rest_digits else ""
    hyperplanes = ", ".join(f"Hyperplane x{idx} = 0.5" for idx in indices)
    print(f"Digit {digit_label}{rest_str}: {hyperplanes}")

# Pick first digit and pixel
chosen_digit_idx = list(unique_pixels.keys())[0]
chosen_digit_label = y[chosen_digit_idx]
chosen_pixel = unique_pixels[chosen_digit_idx][0]

def hyperplane_classifier(X, pixel_idx):
    return (X[:, pixel_idx] > 0.5).astype(int)

y_binary = (y == chosen_digit_label).astype(int)
y_pred = hyperplane_classifier(X, chosen_pixel)
acc = accuracy_score(y_binary, y_pred)

print('\n------')
print(f"Classification Accuracy Using Pixel x{chosen_pixel} for Digit {chosen_digit_label}: {acc:.2f}\n")
print("y:       ", y)
print("y_binary:", y_binary)
print("y_pred:  ", y_pred)
print('------\n')
```

4 Hyperplane Separation

(1) Toy 5x5 Model (basic) – Main Steps

Step No.5 – PCA Visualization
Part A

- Use PCA to project digits into 3D for visualizing true labels, the target digit, and separating hyperplanes.



```
if len(remaining_digits) >= 3:
    pca = PCA(n_components=3)
    X_pca = pca.fit_transform(X)

    fig = plt.figure(figsize=(14, 6))
    ax1 = fig.add_subplot(121, projection='3d')
    ax2 = fig.add_subplot(122, projection='3d')

    # Left: With colors for all digits
    for i in range(len(X)):
        digit_label = y[i]
        ax1.scatter(X_pca[i, 0], X_pca[i, 1], X_pca[i, 2], color=label_to_color[digit_label],
        label=f"Digit {digit_label}" if f"Digit {digit_label}" not in ax1.get_legend_handles_labels()[1] else "", s=100)

    # Right: With color for only target digits
    seen_labels = set()
    for i in range(len(X)):
        label = f"Digit {y[i]}"
        color = 'blue' if y_pred[i] == 1 else 'gray'
        ax2.scatter(X_pca[i, 0], X_pca[i, 1], X_pca[i, 2], color=color, label=label if
        label not in seen_labels else "", s=100)

        seen_labels.add(label)

    # Common hyperplane in PCA space
    normal = np.zeros(D)
    normal[chosen_pixel] = 1.0
    normal_pca = pca.components_ @ normal
    point = np.zeros(D)
    point[chosen_pixel] = 0.5
    point_pca = pca.transform(point.reshape(1, -1))[0]
    xx, yy = np.meshgrid(np.linspace(-3, 3, 10), np.linspace(-3, 3, 10))
    zz = (-normal_pca[0] * xx - normal_pca[1] * yy + point_pca[2]) / normal_pca[2]

    for ax in [ax1, ax2]:
        ax.plot_surface(xx, yy, zz, color='gray', alpha=0.2)
        ax.set_xlabel("PC 1")
        ax.set_ylabel("PC 2")
        ax.set_zlabel("PC 3")

    ax1.set_title(f"PCA 3D – Step {step}")
    ax2.set_title(f"Blue & Gray PCA 3D – Step {step}")
    ax1.legend()
    ax2.legend()
    plt.tight_layout()
    plt.show()
```

Step No.5 – PCA Visualization
Part B

- Same as before… BUT
- Use PCA to project digits into **2D** for visualizing true labels, the target digit, and separating hyperplanes.

Step No.6 – Remove Separated Digits and Iterate
Step No.7 – Final Summary

- Remove the currently separated digit from the dataset and continue the loop until all are handled.
- Print a summary of all steps where digits were separated using simple hyperplanes based on unique pixels.

```
    #-----------------------------------------------------------------------------
    # Step #6 – Remove Separated Digit and Iterate
    #-----------------------------------------------------------------------------
    separation_steps.append(f"Step {step}: Separate Digit {chosen_digit_label} with pixel x{chosen_pixel} = 0.5")
    keep_indices = [i for i in range(len(y)) if y[i] != chosen_digit_label]
    remaining_digits = [remaining_digits[i] for i in keep_indices]
    remaining_labels = [remaining_labels[i] for i in keep_indices]
    step += 1

#-----------------------------------------------------------------------------
# Step #7 – Final Summary
#-----------------------------------------------------------------------------
print("\n" + "-" * 80)
print("\nFinal Separation Steps:")
for line in separation_steps:
    print(line)
```

4 Hyperplane Separation

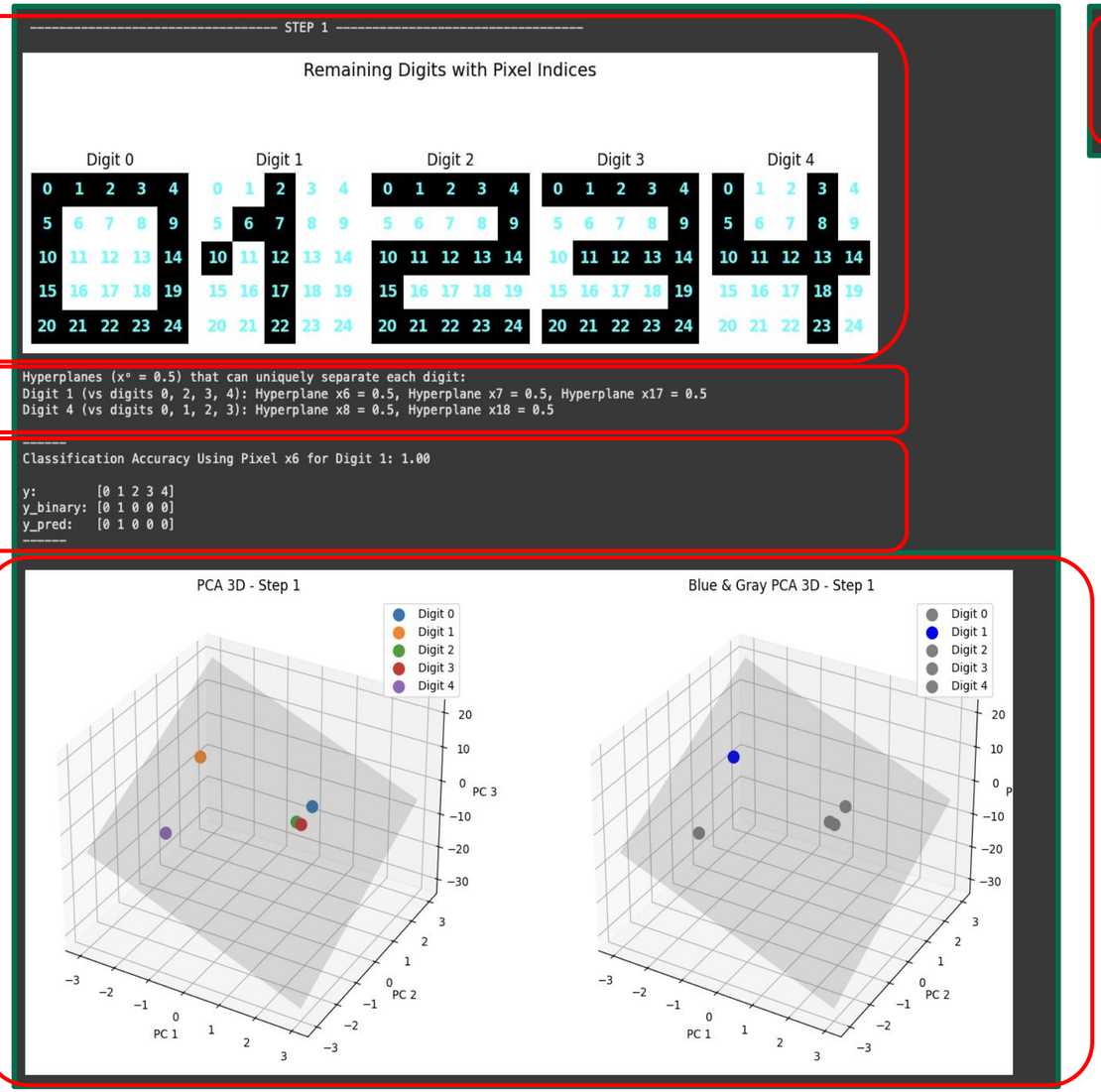
(1) Toy 5x5 Model (basic) – Results

In the first step, the remaining digits are displayed with their pixel indices overlaid on each pixel.

A list of hyperplanes (for each digit) that can uniquely separate each digit from the rest of the classes.

The classification accuracy which is determined by if the y_pred determined by the hyperplane is equal to the y_binary of that step.

PCA is used to project digits into 3D for visualizing true labels, the target digit, and separating hyperplanes.




```
Final Separation Steps:
Step 1: Separate Digit 1 with pixel x6 = 0.5
Step 2: Separate Digit 4 with pixel x8 = 0.5
Step 3: Separate Digit 0 with pixel x5 = 0.5
Step 4: Separate Digit 2 with pixel x10 = 0.5
```

At the end of the separation, all the digits are in order of separation and the hyperplanes that separated them are displayed.

4 Hyperplane Separation

(2) MNIST Data Model (basic) – Main Steps

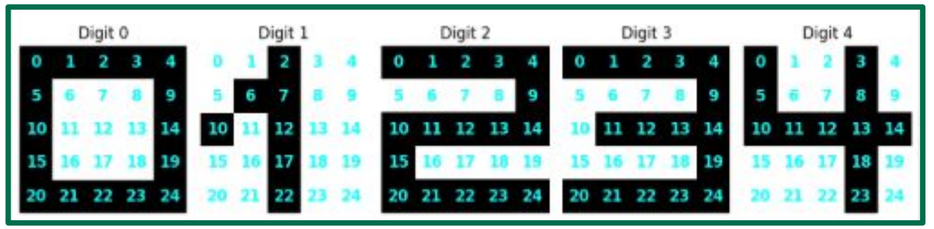
What's the Same?

- 95% of the code is the same, with the only difference coming in with how the output is displayed in the console and how the data is being generated.
- Toy 5x5 Digits -> MNIST Digits

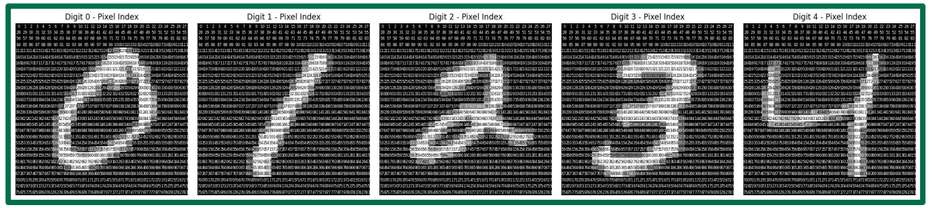
What's the Difference?

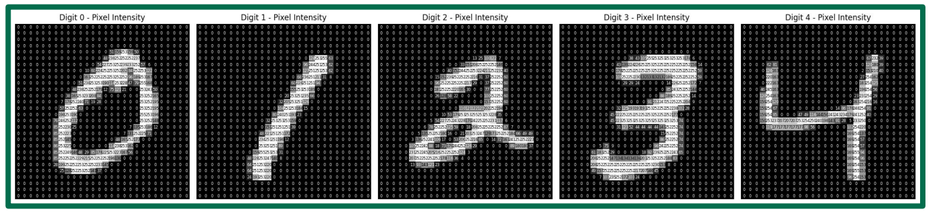
- Instead of generating toy 5x5 digits which lack a little complexity and curvature, this dataset takes the problem up another step with a significant increase in dimension (25D -> 784D)

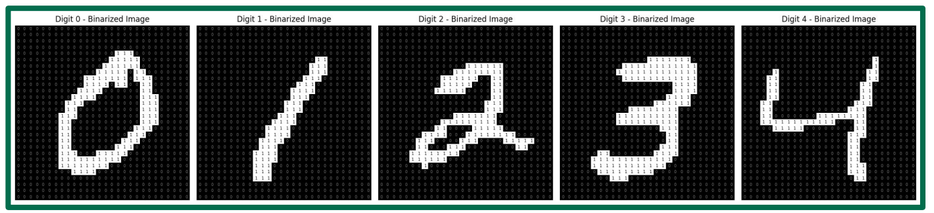
```
#---------------------------------------------------------------
# Step II – Select Digits
#---------------------------------------------------------------
selected_digits = [0, 1, 2, 3, 4]
digit_images = {}

#---------------------------------------------------------------
# Step III – Load MNIST Dataset (No Transform)
#---------------------------------------------------------------
raw_mnist = datasets.MNIST(root='./data', train=True, download=True, transform=None)

#---------------------------------------------------------------
# Step IV – Extract One Example per Digit
#---------------------------------------------------------------
for img, label in raw_mnist:
    if label in selected_digits and label not in digit_images:
        digit_images[label] = np.array(img)  # convert PIL to NumPy array
    if len(digit_images) == len(selected_digits):
        break
```



vs.







4 Hyperplane Separation

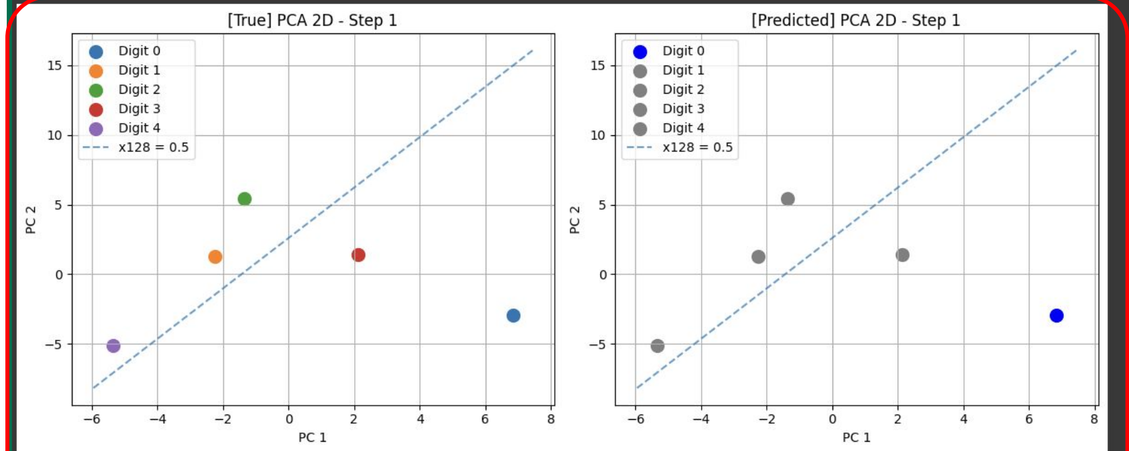
(2) MNIST Data Model (basic) – Results

A list of hyperplanes (for each digit) that can uniquely separate each digit from the rest of the classes.

```
-------------------------------- STEP 1 --------------------------------


Hyperplanes (x₁ = 0.5) that can uniquely separate each digit:
Digit 0 (vs digits 1, 2, 3, 4): x128 = 0.5, x129 = 0.5, x130 = 0.5, x317 = 0.5, x318 = 0.5, and 28 more...
Digit 3 (vs digits 0, 1, 2, 4): x154 = 0.5, x178 = 0.5, x179 = 0.5, x180 = 0.5, x181 = 0.5, and 18 more...
Digit 4 (vs digits 0, 1, 2, 3): x161 = 0.5, x201 = 0.5, x228 = 0.5, x229 = 0.5, x256 = 0.5, and 28 more...
Digit 2 (vs digits 0, 1, 3, 4): x262 = 0.5, x289 = 0.5, x458 = 0.5, x463 = 0.5, x485 = 0.5, and 17 more...
Digit 1 (vs digits 0, 2, 3, 4): x295 = 0.5, x296 = 0.5, x297 = 0.5, x323 = 0.5, x324 = 0.5, and 13 more...

______
Classification Accuracy Using Pixel x128 for Digit 0: 1.00

y:        [0 1 2 3 4]
y_binary: [1 0 0 0 0]
y_pred:   [1 0 0 0 0]
______
```

The classification accuracy which is determined by if the y_pred determined by the hyperplane is equal to the y_binary of that step.

PCA is used to project digits into 2D for visualizing true labels, the target digit, and separating hyperplanes.




```
Final Separation Steps:
Step 1: Separate Digit 0 with pixel x128 = 0.5
Step 2: Separate Digit 3 with pixel x154 = 0.5
Step 3: Separate Digit 1 with pixel x159 = 0.5
Step 4: Separate Digit 4 with pixel x161 = 0.5
```

At the end of the separation, all the digits are in order of separation and the hyperplanes that separated them are displayed.

4 Hyperplane Separation

How to Make this More Applicable?

Main Goal: Instead of cherry-picking pixels visually, I want to extract pixel statistics across real data and validate them with proper test metrics.

First Change:

- **Instead of...** loading 1 example per digit which is hard to generalize.
- **I want to...** Use 100 train + 100 test samples per digit, which enables statistical learning.

Second Change:

- **Instead of...** using binary pixels directly.
- **I want to...** compute per-pixel activation frequencies across real data so the trends and unique pixels are generalized opposed to specific for single digit.

Third Change:

- **Instead of...** picking unique pixel locations per digit based on value.
- **I want to...** pick dominant pixels via thresholding (e.g., >0.65 in one class, <0.35 in others) so it is applicable to a larger set of number and won't look for a specific pixel that might have been accidentally on/off.

Final Change:

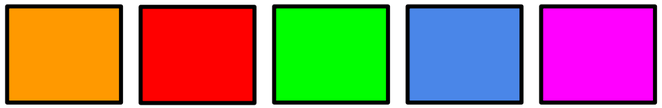
- **Instead of...** having no real-world testing.
- **I want to...** classify on test data using xi=0.5 hyperplanes & prints accuracy statistics.

4 Hyperplane Separation

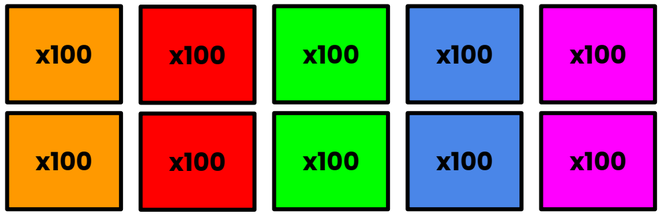
How to Make this More Applicable?

MNIST

Step I - III



Step IV

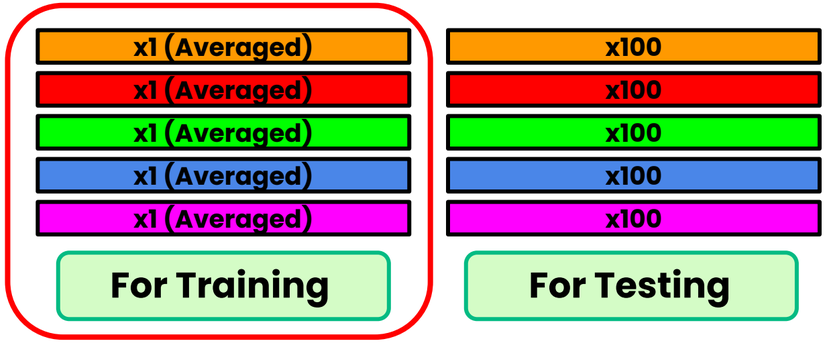


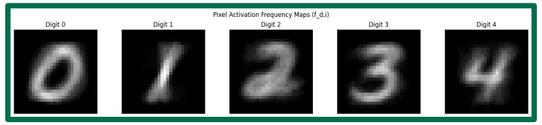
Step V - Step VI

For Training

For Testing

Step VII





Determine 'unique-pixels' by attaining pixels that are active over 65% in the target digit and less than 35% active in the non-target digits.

Step VIII

Important Step: Using the unique pixels found, design a **custom** method that tests whether a specific target digit has been accurately identified.

4 Hyperplane Separation

(1) Single-Pixel Hyperplane – Pixel-Wise Digit Separability Analysis

Single-Pixel Hyperplane

```
print(f"\n--- Per-Pixel Classification for All Digits with Dominant Pixels ---\n")
print(f"{'Digit':>5}  {'Pixel':>7}  {'Accuracy':>9}  {'F1 Score':>9}  {'Hyperplane Condition'}")
print("-" * 70)

for digit in selected_digits:
    if not dominant_pixels[digit]:
        continue  # Skip if no dominant pixels

    y_true = (y_test == digit).astype(int)
    results = []

    for pixel_index in dominant_pixels[digit]:
        y_pred = (X_test[:, pixel_index] > 0.5).astype(int)
        acc = accuracy_score(y_true, y_pred)
        f1 = f1_score(y_true, y_pred, zero_division=0)
        condition = f"x_{pixel_index} > 0.5"
        results.append((digit, pixel_index, acc, f1, condition))

    # Sort results for this digit by F1
    results.sort(key=lambda x: x[3], reverse=True)

    # Print each row
    for digit, pixel, acc, f1, condition in results:
        print(f"{digit:>5}  {pixel:>7}  {acc:>9.3f}  {f1:>9.3f}  {condition}")
```

For each digit with dominant pixels, this section:

- Converts the test labels into binary form (digit vs. not-digit),
- Loops through each dominant pixel,
- Applies a simple classifier (xi = 0.5),
- Computes Accuracy and F1 score,
- Stores the results along with the pixel's hyperplane condition.

```
--- Per-Pixel Classification for All Digits with Dominant Pixels ---
Digit    Pixel   Accuracy   F1 Score  Hyperplane Condition
----------------------------------------------------------------------
    0      329      0.878      0.681  x_329 > 0.5
    0      384      0.796      0.528  x_384 > 0.5
    0      265      0.760      0.528  x_265 > 0.5
    3      376      0.770      0.538  x_376 > 0.5
    3      404      0.680      0.416  x_404 > 0.5
```

For each digit, every unique pixel acts as a hyperplane xi >0.5, and is individually applied to the test dataset to evaluate its ability to classify the digit against all others.

4 Hyperplane Separation

(2) Logical Combination (AND/OR) – Pixel-Wise Digit Separability Analysis

Logical Combination (AND/OR)

```
results = []

for digit in selected_digits:
    pixels = dominant_pixels.get(digit, [])
    if not pixels:
        continue
    x_vals = X_test[:, pixels]
    y_true = (y_test == digit).astype(int)

    for rule in ["any", "all"]:
        if rule == "any":
            y_pred = np.any(x_vals > 0.5, axis=1).astype(int)
            condition = " OR ".join([f"x_{i} > 0.5" for i in pixels])
        else:
            y_pred = np.all(x_vals > 0.5, axis=1).astype(int)
            condition = " AND ".join([f"x_{i} > 0.5" for i in pixels])

        acc = accuracy_score(y_true, y_pred)
        f1 = f1_score(y_true, y_pred, zero_division=0)

        results.append((digit, rule, acc, f1, condition))

print(f"\n--- Logical Rule Classification using Dominant Pixels ---\n")
print(f"{'Digit':>5}  {'Rule':>5}  {'Accuracy':>9}  {'F1 Score':>9}       {'Hyperplane Condition'}")
print("-" * 80)
for digit, rule, acc, f1, condition in results:
    print(f"{digit:>5}  {rule:>5}  {acc:>9.3f}  {f1:>9.3f}       {condition}")
```

In this section, for each digit's unique pixels, two logical rules, **any** and **all**, are tested. Under the any rule, the digit is classified as the target if any of its corresponding hyperplane conditions xi >0.5 are satisfied. Under the all rule, the digit is classified as the target only if all of the hyperplane conditions are simultaneously satisfied.

```
--- Logical Rule Classification using Dominant Pixels ---

Digit   Rule   Accuracy   F1 Score       Hyperplane Condition
--------------------------------------------------------------------------------
    0    any      0.714      0.557       x_265 > 0.5 OR x_329 > 0.5 OR x_384 > 0.5
    0    all      0.860      0.500       x_265 > 0.5 AND x_329 > 0.5 AND x_384 > 0.5
    3    any      0.680      0.474       x_376 > 0.5 OR x_404 > 0.5
    3    all      0.770      0.475       x_376 > 0.5 AND x_404 > 0.5
```

For each digit, all of its dominant pixels are combined using a logical rule **(any or all)**, where each pixel defines a hyperplane xi >0.5; the resulting composite condition is applied to the test dataset to evaluate its effectiveness in separating the digit from all others.

4 Hyperplane Separation

(3) Weighted Average (Linear Hyperplane) – Pixel-Wise Digit Separability Analysis

Weighted Average (Linear Hyperplane)

```python
results = []

for digit in selected_digits:
    pixels = dominant_pixels.get(digit, [])
    if not pixels:
        continue
    avg_activation = np.mean(X_test[:, pixels], axis=1)
    y_pred = (avg_activation > 0.5).astype(int)
    y_true = (y_test == digit).astype(int)

    acc = accuracy_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred, zero_division=0)

    # Define hyperplane condition string
    k = len(pixels)
    condition = f"(1/{k}) * sum(x_i for i in S) > 0.5  ⇔  sum(x_i) > {k/2:.1f}"

    results.append((digit, acc, f1, condition))

print(f"\n--- Weighted Average Classification using Dominant Pixels ---\n")
print(f"{'Digit':>5}  {'Accuracy':>9}  {'F1 Score':>9}  {'Hyperplane Condition'}")
print("-" * 70)
for digit, acc, f1, condition in results:
    print(f"{digit:>5}  {acc:>9.3f}  {f1:>9.3f}  {condition}")
```

This code calculates the average value of the **preselected dominant pixels** for each test image. If the average is greater than 0.5, the image is classified as the target digit. The true labels are also converted to binary (1 for the target digit, 0 for all others).

```
--- Weighted Average Classification using Dominant Pixels ---

Digit   Accuracy   F1 Score  Hyperplane Condition
----------------------------------------------------------------------
    0      0.860      0.646  (1/3) * sum(x_i for i in S) > 0.5  ⇔  sum(x_i) > 1.5
    3      0.770      0.475  (1/2) * sum(x_i for i in S) > 0.5  ⇔  sum(x_i) > 1.0
```

For each digit, the hyperplane is defined by taking the average of its predetermined dominant pixel values in a test image. If this average (for example, 1 plus 1 plus 0 divided by 3) is greater than 0.5, then the test sample is classified as the target digit.

5 Questions and Remarks

Regarding "**Visualizing the Data Changing Through The Layers**":

- **No major questions, I followed your suggestion on how to order the plots, and how to bring the data down to 2D via PCA and I have provided some other types of data projections I thought would be insightful.**
- **One remark I would like to make is that the results captured don't always occur, but I tried to screenshot the most useful result after running the code multiple times. I would like help in better understanding these results, what they mean, and how to move forward.**

Regarding **"Implementation with Synthetic Data"**:

- **This section was relatively new for me, so could you please help with verifying that the different types of synthetic data I created are accurate?**
- **I ran the full code involving training and calculating the cumulative parameters, and there was a major tradeoff between accuracy and surjectivity. Could you please review this section and let me know what changes I have to make?**

Regarding **"Hyperplane Separation"**:

- **As this section was experimental, I first tried implementing what Professor Chen explained (the basic version) to make sure I understood the code and the structure. Then, I tried playing around with MNIST data with actual training and testing splits, and I was left wondering, how can I accurately project a hyperplane from a super-high dimension to 2D via PCA. I tried this out and there seems to be something wrong as shown in the images of slide 33; while the accuracy of the hyperplane is 1.00, the hyperplane is projected to separate digits incorrectly.**

Thank You So Much

for staying with me through this presentation, I am looking forward for your feedback!