Mayank Konduri

July 02, 2025

Summer 2025 Projects (Update Meeting #1)

Experiment 1: Testing for Sequential Linear Separability (SLS)

Experiment 2: Visualizing The Cones

Experiment #1:

1. Testing for sequential linear separability

In [CE24], we define what it means for a set of training data to be sequentially linearly separable (SLS). We are interested in finding out if common benchmark datasets, like MNIST and CIFAR10, are SLS.

It might be easier to start with MNIST, and perhaps choose a subset of the classes, say $N = 3$ or 4 classes.

Suggested initial plan:

- Find mean $\overline{x_{0,j}}$ for each class.
- Find barycenter $\overline{x}$.
- Pick an order
- For n in range(N):
  (1) Use SVM to do one-vs-all classification and find a hyperplane that separates class $\mathcal{X}_{0,n}$ from all other ones.
  (2) Find intersection p_n of this hyperplane with the line connecting the class mean $\overline{x_{0,n}}$ to $\overline{x}$.
  (3) Send all points in $\mathcal{X}_{0,n}$ to p_n.

It's possible that, even if MNIST **is** SLS, the choice of hyperplanes will be suboptimal and at some point the separation will not be perfect, even for the correct order. If none of the experiments result in perfect separation, report back with some measure of the errors, and we can go from there.

Step 0 - Import Necessary Libraries

```
[ ] import numpy as np
    import tensorflow as tf
    from sklearn.svm import LinearSVC, SVC
    import matplotlib.pyplot as plt
```

Step 1 - Load and Filter MNIST Dataset

```
[ ] import tensorflow as tf
    import numpy as np

    #Load MNIST from TensorFlow
    (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()

    #Select classes 0, 3, 8
    selected_classes = [8, 3, 0]

    #Create filters
    train_mask = np.isin(y_train, selected_classes)

    #Apply the filter to the training and testing datasets
    x_train_subset = x_train[train_mask]
    y_train_subset = y_train[train_mask]

    print("Train images shape:", x_train_subset.shape)
    print("Train labels shape:", y_train_subset.shape)
    print("Unique train labels:", np.unique(y_train_subset))
    print("")
    print("")

    fig, axes = plt.subplots(nrows=1, ncols=6, figsize=(12, 3))
    fig.suptitle("Noiseless MNIST Samples")

    plot_idx = 0

    for digit in selected_classes:
        indices = np.where(y_train_subset == digit)[0][:2]

        for idx in indices:
            ax = axes[plot_idx]
            ax.imshow(x_train_subset[idx], cmap='gray')
            ax.axis('off')
            ax.set_title(f"Digit {digit}")
            plot_idx += 1

    plt.tight_layout(rect=[0, 0.03, 1, 0.9])
    plt.show()
```

```
Train images shape: (17905, 28, 28)
Train labels shape: (17905,)
Unique train labels: [0 3 8]
```

Step 0 & Step 1

Getting the Data Ready

- **Load & filter:** Loads MNIST, keeps only digits **0, 3, 8** → reduces experiment to a smaller class subset.
- **Subset:** Extracts images/labels for these digits → shapes stay *(samples, 28, 28)*.
- **Visualization:** Plots **2 samples per digit** → check that filtering worked and data is noiseless.

Step 2, Step 3, & Step 4

Computing SLS Prereqs.

Step 2 - Flatten Images from MNIST

```
#Flatten each 28x28 image to a 784-dimensional vector
x_train_flat = x_train_subset.reshape(x_train_subset.shape[0], -1)
```

Step 3 - Compute Class Means

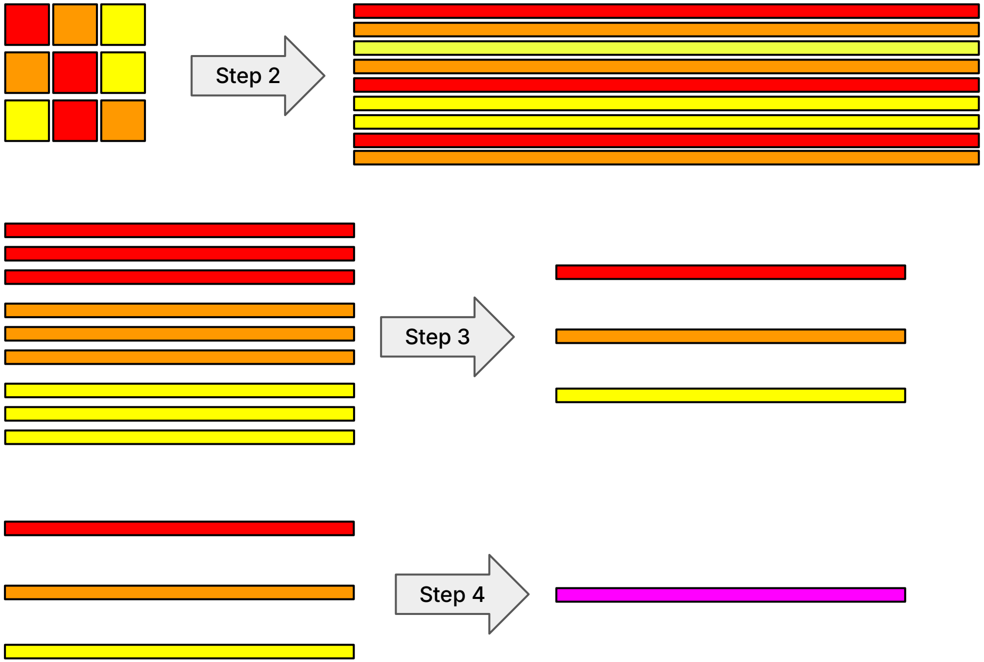
```
class_means = {}

for label in np.unique(y_train_subset):
    class_images = x_train_flat[y_train_subset == label]

    #Compute mean vector
    mean_vector = np.mean(class_images, axis=0)
    class_means[label] = mean_vector
    print(f"Class {label} mean computed. Shape: {mean_vector.shape}")
```

```
Class 0 mean computed. Shape: (784,)
Class 3 mean computed. Shape: (784,)
Class 8 mean computed. Shape: (784,)
```

Step 4 - Compute the Barycenter

```
#Combine class mean vectors into a matrix
means_stack = np.stack([class_means[cls] for cls in selected_classes])

#Compute the barycenter as the average of class means
barycenter = np.mean(means_stack, axis=0)

print("Barycenter shape:", barycenter.shape)
```

```
Barycenter shape: (784,)
```



```python
#Binary labels: 1 for class n, -1 for all other classes
y_binary = np.where(y_mod == n, 1, -1)

#If only one class label exists in y_binary (all 1 or all -1), skip
# if np.unique(y_binary).size < 2:
#     print(f"Skipping class {n} - only one class present in binary labels.")
#     separation_results[n] = True
#     misclassified_counts[n] = 0
#     break

#Training a linear SVM classifier (LinearSVC) to separate class n vs. the rest
clf = LinearSVC(C=1.0, max_iter=10000)
#clf = SVC(kernel='linear', C=1e6)
clf.fit(x_mod, y_binary)

#Evaluate performance through predictions
preds = clf.predict(x_mod)

#Error rate calculation for class n
error_rate = np.mean(preds[y_binary == 1] != 1)

#True if no misclassifications
is_perfect = error_rate == 0.0

#Results recorded for this class
separation_results[n] = is_perfect
misclassified_counts[n] = np.sum(preds != y_binary)
error_rate_dict[n] = error_rate

print(f"Perfect separation: {is_perfect}")
print(f"Misclassified points: {misclassified_counts[n]}")
print(f"Error rate: {error_rate:.4f}")

#Condition for moving forward through the order
if not is_perfect:
    if strict or error_rate > error_threshold:
        print("Stopping due to separation failure.")
        break
```

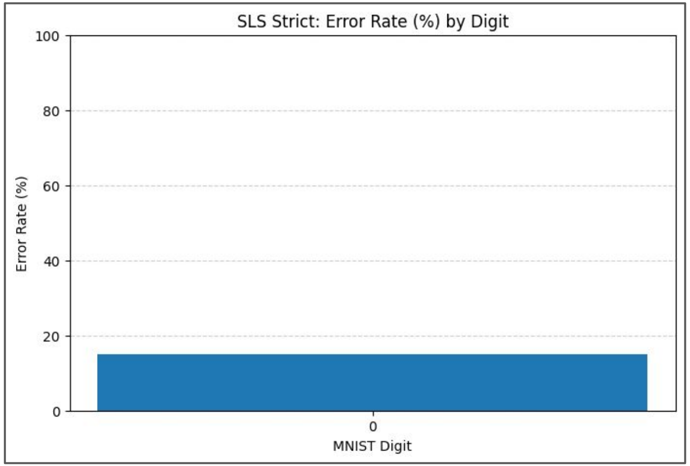
Step 6 (Part 1)
The SVM!

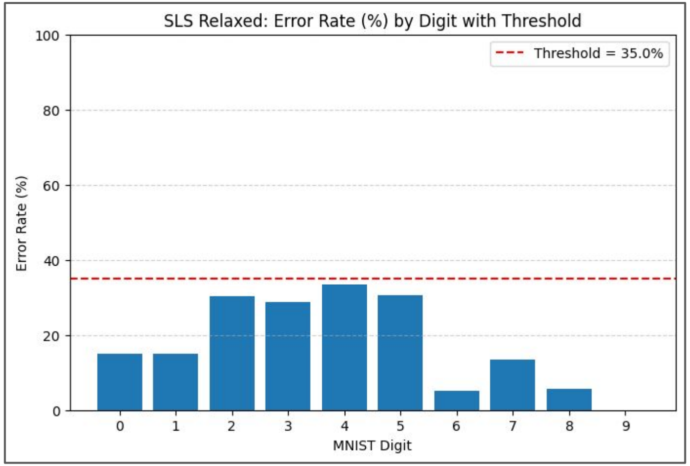
- Creates **binary labels** for one-vs-all classification: target class is **+1**, all others **-1**.
- **Trains a linear SVM** to find a hyperplane that separates the target class from the rest, then checks predictions to compute the **error rate**.
- **Logs whether separation is perfect**, counts misclassifications, and **stops early** if the error is too high (depending on strictness).

Step 7
Plotting the Results

Step 7 - Execute Function (Takes about 10 minutes to execute entirely)

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

    #Run SLS experiments
    results_strict, errors_strict, error_rate_dict_strict, _, _ = sls_experiment(strict=True)
    results_relaxed, errors_relaxed, error_rate_dict_relaxed, error_threshold, pn_points = sls_experiment(strict=False)

    processed_classes_strict = list(error_rate_dict_strict.keys())
    errors_strict_pct = [error_rate_dict_strict[d] * 100 for d in processed_classes_strict]

    processed_classes_relaxed = list(error_rate_dict_relaxed.keys())
    errors_relaxed_pct = [error_rate_dict_relaxed[d] * 100 for d in processed_classes_relaxed]

    error_threshold_pct = error_threshold * 100

    plt.figure(figsize=(8, 5))
    plt.bar(processed_classes_strict, errors_strict_pct, tick_label=processed_classes_strict)
    plt.title('SLS Strict: Error Rate (%) by Digit')
    plt.xlabel('MNIST Digit')
    plt.ylabel('Error Rate (%)')
    plt.ylim(0, 100)
    plt.grid(True, axis='y', linestyle='--', alpha=0.6)
    plt.show()

    plt.figure(figsize=(8, 5))
    plt.bar(processed_classes_relaxed, errors_relaxed_pct, tick_label=processed_classes_relaxed)
    plt.axhline(y=error_threshold_pct, color='red', linestyle='--', label=f'Threshold = {error_threshold_pct:.1f}%')
    plt.title('SLS Relaxed: Error Rate (%) by Digit with Threshold')
    plt.xlabel('MNIST Digit')
    plt.ylabel('Error Rate (%)')
    plt.ylim(0, 100)
    plt.legend()
    plt.grid(True, axis='y', linestyle='--', alpha=0.6)
    plt.show()
```





Experiment #2:

2. Visualizing the cones

In [CE23, CE24, Ewa25] we define certain cones which are used to explicitly construct ReLU neural networks that classify data. We are interested in seeing how such cones might arise when training neural networks with gradient descent and its variants.

Suggested initial plan:

- Train a neural network to classify MNIST (perhaps only a subset of the classes), using the architecture suggested in the hyperplanes paper [CE24].
- Determine cumulative parameters $W^{(\ell)}, b^{(\ell)}$.
- Determine cones (can use polyhedral cones from [Ewa25] and determine the base point and edges).
- Try to visualize the cones by some form of dimensional reduction. For instance: Do principal component analysis (PCA) on training data, then use these coordinates and project cones (base point and edges) as well.

There may be other ways of gaining information about these cones that would be interesting.

Step 2 & Step 3

Getting the Data Ready

Step II - Select Digits

```
[2] selected_digits = [5, 6, 7]
    Q = len(selected_digits)     # number of classes
    L = Q + 1                    # total layers

    print(f"Using digits: {selected_digits} | Q = {Q} | L = {L}")
```

```
Using digits: [5, 6, 7] | Q = 3 | L = 4
```

Step III - Preprocess Digits

```
[3] transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.view(-1))  # flatten 28x28 to 784
    ])

    train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
    test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)


    train_mask = torch.zeros_like(train_dataset.targets, dtype=torch.bool)
    test_mask = torch.zeros_like(test_dataset.targets, dtype=torch.bool)


    for digit in selected_digits:
        train_mask |= (train_dataset.targets == digit)
        test_mask |= (test_dataset.targets == digit)

    # create filters
    train_dataset.targets = train_dataset.targets[train_mask]
    train_dataset.data    = train_dataset.data[train_mask]
    test_dataset.targets = test_dataset.targets[test_mask]
    test_dataset.data    = test_dataset.data[test_mask]

    # remap labels to {0,...,Q-1}
    label_map = {digit: idx for idx, digit in enumerate(selected_digits)}
    train_dataset.targets = torch.tensor([label_map[label.item()] for label in train_dataset.targets])
    test_dataset.targets = torch.tensor([label_map[label.item()] for label in test_dataset.targets])


    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
    test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)

    print(f"Train samples: {len(train_dataset)}")
    print(f"Test samples: {len(test_dataset)}")

    # confirm size and flattened shape
    images, labels = next(iter(train_loader))
    print(f"Batch images shape: {images.shape}")
    print(f"Batch labels shape: {labels.shape}")
    images, labels = next(iter(test_loader))
    print(f"Test batch images shape: {images.shape}")
    print(f"Test batch labels shape: {labels.shape}")
```

Select digits and compute layer counts

- selected_digits = [5, 6, 7] → only keep MNIST samples for digits 5, 6, 7
- Q = number of classes (3), L = total layers (Q + 1)

Load and flatten MNIST

- Uses transforms.ToTensor() to convert each image to a tensor in [0,1]
- transforms.Lambda flattens each 28×28 image to a 784-dim vector

Mask to keep only selected digits

- Creates boolean masks to filter training & test sets to digits in selected_digits

Remap class labels

- Re-labels selected digits to [0, 1, 2] instead of [5, 6, 7] for easier training

Wrap in PyTorch DataLoader

- Creates train_loader and test_loader with batches for training/testing
- Confirms the shapes: each batch has images of shape [batch_size, 784]

Task I - Training the NN

```python
[4] # ReLU Feedforward Network
class TruncationNet(nn.Module):
    def __init__(self, input_dim=784, hidden_dims=[784, 784, 784], output_dim=3): # 3 hidden layers (Q) + 1 output layer, should output_dim be 3 or 10?
        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)
```

```
TruncationNet(
  (layers): ModuleList(
    (0-2): 3 x Linear(in_features=784, out_features=784, bias=True)
    (3): Linear(in_features=784, out_features=3, bias=True)
  )
)
```

```python
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # gradient descent
criterion = nn.CrossEntropyLoss()

epochs = 10
for epoch in range(epochs):
    model.train()
    for data, target in train_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward() # backpropogation
        optimizer.step()
    print(f'Epoch {epoch+1}, Loss: {loss.item():.4f}')

print("Training done!")
```

```
Test Accuracy: 99.20%
```

In [CE23, CE24, Ewa25] we define certain cones which are used to explicitly construct ReLU neural networks that classify data. We are interested in seeing how such cones might arise when training neural networks with gradient descent and its variants.

Suggested initial plan:

- Train a neural network to classify MNIST (perhaps only a subset of the classes), using the architecture suggested in the hyperplanes paper [CE24].

Theorem 5.2. *Consider a set of training data $\mathcal{X}_0 = \bigcup_{j=1}^{Q} \mathcal{X}_{0,j} \subset \mathbb{R}^M$ separated into Q classes corresponding to linearly independent labels $\{y_j\}_{j=1}^{Q} \subset \mathbb{R}^Q$. If the data is sequentially linearly separable, then a neural network with ReLU activation function defined as in (2.1), with $L = Q+1$ layers, $d_0 = M$, $d_{Q+1} = Q$, and $d_0 = d_\ell \geq Q$ for all hidden layers $\ell = 1, \cdots, Q$, attains*

$$\min_{(W_i,b_i)_{i=1}^{L}} \mathcal{C}[(W_i,b_i)_{i=1}^{L}] = 0, \tag{5.2}$$

Based on Theorem 5.2 in the hyperplanes paper [CE24], the suggested architecture is:

Layers: L = Q + 1 *(ex: 3+1)*

Input dim: d_0 = M *(ex: 784)*

Hidden layers: all same width M ≥ Q *(ex: 784 = 784 = 784 > 3)*

Last layer: output dim = Q *(ex: 3)*

Weight matrices: recursively defined

Bias vectors: recursively defined

Task II – Determine cumulative parameters $W(\ell), b(\ell)$

```
[7] W_cum_chain = []
    b_cum_chain = []

    for ell in range(L):  # loop through each layer
        if ell == 0:
            # for the first layer, cumulative weight and bias are just themselves
            Wcum = Ws[0]
            bcum = bs[0]
        else:
            # multiply the current layer's weight by the total weight so far
            Wcum = Ws[ell] @ W_cum_chain[-1]

            # for the bias:
            # start at zero, and for each previous bias, multiply it by all the weights that come after it (so they affect how the bias carries forward)
            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]

            # finally add the current layer's bias
            bcum = btemp + bs[ell]

        print(f"Layer {ell+1}: W_cum shape = {Wcum.shape}, b_cum shape = {bcum.shape}")
        W_cum_chain.append(Wcum) # adds the current layer's cumulative weight to the final array
        b_cum_chain.append(bcum) # adds the current layer's cumulative bias to the final array
```

Proposition 2. *Assume $M = d_0 \geq d_1 \geq \cdots \geq d_L = Q$, $X^{(\ell)} \in \mathbb{R}^{d_\ell \times N}$ corresponds to the output of a hidden layer of a neural network defined as in (1) on a data matrix $X_0 \in \mathbb{R}^{M \times N}$, and all of the associated weight matrices $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ are full rank. Then the truncation map defined satisfies*

$$X^{(\ell)} = W_\ell \tau_{W_\ell, b_\ell}(X^{(\ell-1)}) + B_\ell. \tag{4}$$

Moreover, defining the cumulative parameters

$$\boxed{W^{(\ell)} := W_\ell \cdots W_1 \in \mathbb{R}^{d_\ell \times d_0}, \quad \text{for } \ell = 1, \cdots, L,} \tag{5}$$

and

$$\boxed{b^{(\ell)} := \begin{cases} W_\ell \cdots W_2 b_1 + W_\ell \cdots W_3 b_2 + \cdots + W_\ell b_{\ell-1} + b_\ell, & \text{if } \ell \geq 2, \\ b_1, & \text{if } \ell = 1, \end{cases}} \tag{6}$$

```
[8] #manually checking if the calculated cumulative parameters are correct

    # layer 1
    W1_manual = Ws[0]
    b1_manual = bs[0]

    print("\nLayer 1:")
    print("W_cum correct?", torch.allclose(W_cum_chain[0], W1_manual, atol=1e-6))
    print("b_cum correct?", torch.allclose(b_cum_chain[0], b1_manual, atol=1e-6))

    # layer 2
    W2_manual = Ws[1] @ Ws[0]
    b2_manual = Ws[1] @ bs[0] + bs[1]

    print("\nLayer 2:")
    print("W_cum correct?", torch.allclose(W_cum_chain[1], W2_manual, atol=1e-6))
    print("b_cum correct?", torch.allclose(b_cum_chain[1], b2_manual, atol=1e-6))

    # layer 3
    W3_manual = Ws[2] @ Ws[1] @ Ws[0]
    b3_manual = Ws[2] @ Ws[1] @ bs[0] + Ws[2] @ bs[1] + bs[2]

    print("\nLayer 3:")
    print("W_cum correct?", torch.allclose(W_cum_chain[2], W3_manual, atol=1e-6))
    print("b_cum correct?", torch.allclose(b_cum_chain[2], b3_manual, atol=1e-6))

    # layer 4
    W4_manual = Ws[3] @ Ws[2] @ Ws[1] @ Ws[0]
    b4_manual = Ws[3] @ Ws[2] @ Ws[1] @ bs[0] + Ws[3] @ Ws[2] @ bs[1] + Ws[3] @ bs[2] + bs[3]

    print("\nLayer 4:")
    print("W_cum correct?", torch.allclose(W_cum_chain[3], W4_manual, atol=1e-6))
    print("b_cum correct?", torch.allclose(b_cum_chain[3], b4_manual, atol=1e-6))
```

```
Layer 1:
W_cum correct? True
b_cum correct? True

Layer 2:
W_cum correct? True
b_cum correct? True
```

```
Layer 3:
W_cum correct? True
b_cum correct? True

Layer 4:
W_cum correct? True
b_cum correct? True
```

Task III (Part 1) – Determine cones

```python
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})
        print(f"\n=== Layer {ell+1} Cone Checks ===")

        # 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")

    return cones

cones = compute_cones(W_cum_chain, b_cum_chain)
```

Task III – Determine cones

> "can use polyhedral cones from [Ewa25] and determine the base point and edges."

- Determine cones (can use polyhedral cones from [Ewa25] and determine the base point and edges).

Note that given W, b, where W is surjective, we can find $\boxed{p := -(W)^+ b}$ and $\boxed{v_i := (W)^+ e_i^m \in \mathbb{R}^n,}$ for $i = 1, \cdots, m$. Conversely, given $p \in \mathbb{R}^n$ and $(v_i)_{i=1}^m \subset \mathbb{R}^n$ linearly independent, we can define $(W)^+ := [v_1 \cdots v_n]$, which is injective and so W is surjective, and $b := -Wp$. □

Consider $W \in GL(n)$ and $b \in \mathbb{R}^n$. Then $p \in \mathbb{R}^n$ and $\underline{v} := (v_1, \cdots, v_n)$ given by Lemma 2.1 define two polyhedral cones

$$\mathcal{S}_+(p, \underline{v}) := \left\{ p + \sum_{i=1}^n a_i v_i : a_i \geq 0, i = 1, \cdots, n \right\} \tag{2.14}$$

and

$$\mathcal{S}_-(p, \underline{v}) := \left\{ p + \sum_{i=1}^n a_i v_i : a_i \leq 0, i = 1, \cdots, n \right\}. \tag{2.15}$$

As it was stated in **Lemma 2.1** in [Ewa25], a truncation map for a ReLU neural network can be described in terms of **polyhedral cones** defined by a base point and edges.

In my implementation, I use the **cumulative weights and biases** for each layer to construct these cones exactly as in the paper:

$$p^{(\ell)} = -(W^{(\ell)})^+ b^{(\ell)}, \quad v_i^{(\ell)} = (W^{(\ell)})^+ e_i.$$

This gives two polyhedral cones for each layer:

$$S_+ = \left\{ p + \sum a_i v_i : a_i \geq 0 \right\}, \quad S_- = \left\{ p + \sum a_i v_i : a_i \leq 0 \right\}.$$

My implementation fully matches **Lemma 2.1 and Equations (2.14–2.15)** in the paper.

Task III (Part 2) – Verifying accuracy of cones

```
# 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")
```

```
=== Layer 1 Cone Checks ===
Rank(W) = 784 | Should be full row rank = 784
✅ Surjective check passed, it is full row rank
✅ Base point residual ||Wp + b|| = 2.20e-05 (OK)
✅ All edge checks passed | Failing: 0/784 | Mean residual: 4.54e-06
------------------
✅ Lemma 2.1 conditions are met

=== Layer 2 Cone Checks ===
Rank(W) = 777 | Should be full row rank = 784
⚠ Not surjective
⚠ Base point residual ||Wp + b|| = 1.28e-03 exceeds tolerance
⚠ Some edge residuals exceed tolerance | Failing: 619/784 | Mean residual: 2.80e-04
------------------
⚠ Lemma 2.1 conditions are not met

=== Layer 3 Cone Checks ===
Rank(W) = 743 | Should be full row rank = 784
⚠ Not surjective
⚠ Base point residual ||Wp + b|| = 4.42e-02 exceeds tolerance
⚠ Some edge residuals exceed tolerance | Failing: 784/784 | Mean residual: 7.44e-03
------------------
⚠ 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|| = 1.79e-07 (OK)
✅ All edge checks passed | Failing: 0/3 | Mean residual: 6.97e-08
------------------
✅ Lemma 2.1 conditions are met
```

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

        if hasattr(W, 'detach'): W = W.detach().cpu().numpy()
        if hasattr(b, 'detach'): b = b.detach().cpu().numpy()

        rank = np.linalg.matrix_rank(W)
        rows, cols = W.shape

        print(f"\n=== Layer {ell+1} ===")
        print(f"Rank(W) = {rank} | Rows = {rows} | Columns = {cols}")

        if rank == rows:
            print("✅ Surjective: full row rank")
        else:
            print("⚠ Not surjective")
            if rank == cols:
                print("✅ Injective: full column rank, can apply Lemma 2.2 to extend cone in higher dimensions.")
            else:
                print("⚠ Not injective either, cone only valid in lower-dimensional subspace.")

    return results_injective

results_injective = check_injective(W_cum_chain, b_cum_chain)
```

```
=== Layer 1 ===
Rank(W) = 784 | Rows = 784 | Columns = 784
✅ Surjective: full row rank

=== Layer 2 ===
Rank(W) = 777 | Rows = 784 | Columns = 784
⚠ Not surjective
⚠ Not injective either, cone only valid in lower-dimensional subspace.

=== Layer 3 ===
Rank(W) = 743 | Rows = 784 | Columns = 784
⚠ Not surjective
⚠ Not injective either, cone only valid in lower-dimensional subspace.

=== Layer 4 ===
Rank(W) = 3 | Rows = 3 | Columns = 784
✅ Surjective: full row rank
```

Analysing What Happened...

- **Layer 1 & 4:** Full row rank → W is surjective → Moore-Penrose pseudoinverse works well → base point & edges match theory → residuals are small.
- **Layer 2 & 3:** Rank deficient (rank < rows) → W not surjective → so W does not cover all output space → pseudoinverse tries to approximate but can’t recover exact solution.
- When W is not full row rank, its pseudoinverse maps into a lower-dimensional subspace → so $Wp + b \neq 0$ exactly → base residuals are larger.
- Edges show large residuals because once again the approximate pseudoinverse for W is used.
- For valid cones (Lemma 2.1), I need W full row rank; otherwise, the pseudoinverse gives approximate results and residual checks fail.

What the Paper [Ewa25] Suggests....

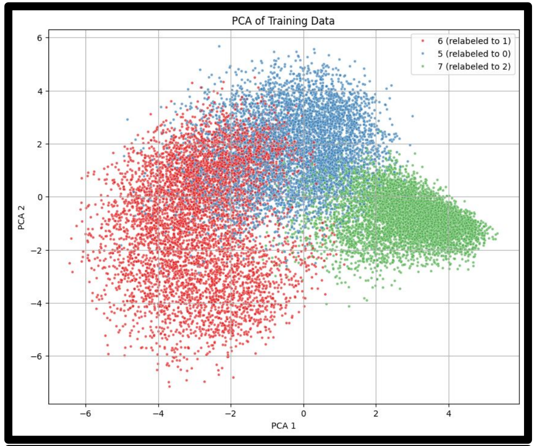
Lemma 2.2. *Let* $W : \mathbb{R}^n \to \mathbb{R}^m$ *be a linear map, for* $n < m$. *There exists* $\tilde{W} \in \mathbb{R}^{m \times m}$ *such that* $W = \tilde{W}\iota(n, m)$. *If* W *is injective, then* $\tilde{W}$ *can be made invertible. In that case,*

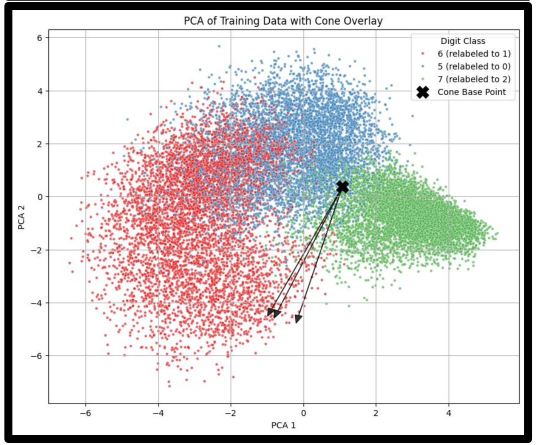
- Lemma 2.2 states that if W is injective (columns full rank but rows not full rank), then the cone defined by W can be lifted into a higher-dimensional space.

2.2. **Increased width.** Next, we study what happens if W is not surjective. We leave the case where W is not full rank for future work, and consider here the situation where $W \in \mathbb{R}^{m \times n}$ for $n < m$, and W

- If W is not injective, there is no action to do for now, **I can only interpret the cone inside the active subspace (the image of W).**

Task IV – PCA Visualization for Cones and Data





- **Batch collection:** Loads all training batches, flattens images to 784-D vectors, and stacks them into X_train; labels are also collected in y_train.
- **PCA fit:** Applies PCA to reduce high-dimensional data to 2D for visualization, fitting on the entire training set.
- **Label mapping & scatter plot:** Creates readable labels (original digit → relabeled) and plots a 2D scatter of training data in PCA space, colored by digit.
- **Cone projection:** Takes the computed cone base p and edges v_i in original space, projects them to 2D PCA space to overlay with the data.
- **Superimposed Data:** Plots the projected cone base as a big black X and edges as arrows, showing how the learned cone aligns with the training data distribution.