ECE 47300 Assignment 9 Exercise

Your Name: Owen Semeter

Exercise 1: Define classifier that extracts latent representations and visualize representations (**40 points**)

The latent (i.e., hidden) representations generated by a deep neural network are very important concept in deep learning since the latent space is where the most significant features of the dataset are learned and extracted. In this homework, we will explore the latent representations of a classifier using clustering and nearest neighbor methods.

We provide the code for a simple residual CNN with batchnorm and data loaders.

- Here, we define a neural network block architecture that does batch normalization after each convolution layer and has a skip connection. You can read more about batch normalization and skip connections in these papers[1,2], respectively.
- In this neural network, the block networks are designed so that the input dimension and the output dimension stay the same. This may not be an optimal design, but it will help us visualize the latent representation later.

[1]: https://arxiv.org/pdf/1502.03167.pdf

[2]: https://arxiv.org/pdf/1512.03385.pdf

In [65]:
```
import torch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

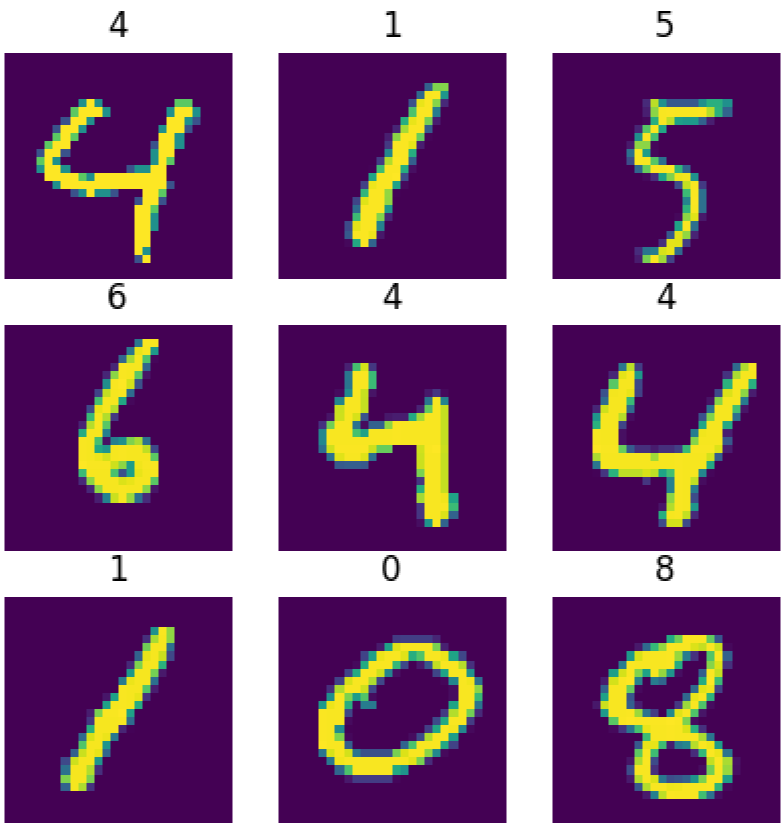
```
cuda
```

In [66]:
```
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np

class SimpleResidualBlock(nn.Module):
    def __init__(self, ch_in, mult=4):
        super().__init__()
        self.conv1 = nn.Conv2d(ch_in, mult * ch_in, kernel_size=3, stride=1, paddin
        self.bn1 = nn.BatchNorm2d(mult * ch_in)

        self.conv2 = nn.Conv2d(mult * ch_in, mult * ch_in, kernel_size=3, stride=1,
        self.bn2 = nn.BatchNorm2d(mult * ch_in)

        self.conv3 = nn.Conv2d(mult * ch_in, ch_in, kernel_size=3, stride=1, paddin
        self.bn3 = nn.BatchNorm2d(ch_in)

    def forward(self, x):
        x_ = x.clone()
        x_ = torch.relu(self.bn1(self.conv1(x_)))
        x_ = torch.relu(self.bn2(self.conv2(x_)))
        x_ = torch.relu(self.bn3(self.conv3(x_)))
        x = x + x_
        return x

class SimpleResNet(nn.Module):
```

```
    def __init__(self, ch_in, n_blocks=3):
        super().__init__()
        self.residual_layers = nn.ModuleList([SimpleResidualBlock(ch_in) for i in r
        self.maxpool = nn.MaxPool2d((2, 2))
        self.fc = nn.Linear(9, 10)

    def forward(self, x):
        for residual in self.residual_layers:
          x = residual(x)
          x = self.maxpool(x)
        x = x.view(x.shape[0], -1)  # Unravel tensor dimensions
        out = self.fc(x)
        return out
```

In [67]:

```
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Create MNIST datasets
classes = np.arange(10)
transform = torchvision.transforms.Compose(
  [torchvision.transforms.ToTensor(),
   torchvision.transforms.Normalize((0.1307,),(0.3081,))])
train_dataset = torchvision.datasets.MNIST('./data', train=True, download=True, tra
test_dataset = torchvision.datasets.MNIST('./data', train=False, download=True, tra

# Create dataloaders
batch_size_train, batch_size_test = 64, 128
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size_tra
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size_test,

# Show sample images
batch_idx, (images, targets) = next(enumerate(train_loader))
fig, ax = plt.subplots(3,3,figsize = (5,5))
for i in range(3):
  for j in range(3):
    image = images[i*3+j].permute(1,2,0)
    image = image/2 + 0.5
    ax[i,j].imshow(image.squeeze(2))
    ax[i,j].set_title(f'{classes[targets[i*3+j]]}')
    ax[i,j].axis('off')
fig.show()
```



Task 1: Inherit the original model class and define a new function returning the same output with the parent's forward function as well as intermediate representations (including the original input)

Specifically, the function below should return the original output from `forward` function of the parent's class and a list of intermediate representations `z_list`. `z_list` should be a Python list with 4 entries corresponding to the original batch and the batch **after each maxpool layer.**

Hints:

- Because of inheritance, you do not need to implement another `__init__` function. For those who are not familiar with inheritance, here is the link to get to know what Inheritance in Python is: https://www.geeksforgeeks.org/inheritance-in-python/.
- You will want to do the same computation as the original `forward` function but add some code to save intermediate representations, i.e., the code should output exactly the same thing as the original `forward` function but also return intermediate outputs.
- The output of this exercise should be:

```
Representation z0 batch shape = torch.Size([128, 1, 28, 28])
Representation z1 batch shape = torch.Size([128, 1, 14, 14])
Representation z2 batch shape = torch.Size([128, 1, 7, 7])
Representation z3 batch shape = torch.Size([128, 1, 3, 3])
```

- We provide a simple example for a linear model below.

In [68]:
```python
# Trivial example below
class AffineModel(nn.Module):
    def __init__(self, A, b):
        super().__init__()
        self.A, self.b = A, b
    def forward(self, x):
        x = torch.matmul(x, self.A)
        return x + self.b
class ExtractAffineModel(AffineModel):
    def compute_and_extract_representations(self, x):
        z_list = [x]
        x = torch.matmul(x, self.A)
        z_list.append(x)
        return x + self.b, z_list
```

In [69]:
```
class SimpleResNetWithRepresentations(SimpleResNet):
  def compute_and_extract_representations(self, x):
    # ---------- Your code ----------
    z_list = [x]
    for residual in self.residual_layers:
      x = residual(x)
      x = self.maxpool(x)
      z_list.append(x.clone())
    x = x.view(x.shape[0], -1)  # Unravel tensor dimensions
    out = self.fc(x)
    # ---------- End your code ----------
    return out, z_list

model = SimpleResNetWithRepresentations(ch_in=1)
model.to(device)
images, labels = next(iter(test_loader)) # get a batch
images = images.to(device)
# Check that outputs match
out, z_list = model.compute_and_extract_representations(images)
assert torch.all(model(images) == out), 'Outputs should be the same'
# Check shapes of representations
assert len(z_list) == 4, 'Should have length of 4'
assert torch.all(z_list[0] == images), 'First entry should be original data'
for zi, z in enumerate(z_list):
  print(f'Representation z{zi} batch shape = {z.shape}')
```

```
Representation z0 batch shape = torch.Size([128, 1, 28, 28])
Representation z1 batch shape = torch.Size([128, 1, 14, 14])
Representation z2 batch shape = torch.Size([128, 1, 7, 7])
Representation z3 batch shape = torch.Size([128, 1, 3, 3])
```

Task 2: Train the model

Using the train and test functions given above, train `model` for 4 epochs using the Adam optimizer with a learning rate of 0.01.

In [70]:
```
def train(epoch, model, optimizer):
    model.train() # we need to set the mode for our model
    for batch_idx, (images, targets) in enumerate(train_loader):
      images = images.to(device)
      targets = targets.to(device)
      optimizer.zero_grad()
      output = model(images)
      loss = F.cross_entropy(output, targets) # Here is a typical loss function (ne
      loss.backward()
      optimizer.step()
      if batch_idx % 100 == 0: # We visulize our output every 10 batches
        print(f'Epoch {epoch}: [{batch_idx*len(images)}/{len(train_loader.dataset)}

def test(epoch, model):
  model.eval() # we need to set the mode for our model
  test_loss = 0
  correct = 0
  with torch.no_grad():
```

```
    for images, targets in test_loader:
      images = images.to(device)
      targets = targets.to(device)
      output = model(images)
      test_loss += F.cross_entropy(output, targets, reduction='sum').item()
      pred = output.data.max(1, keepdim=True)[1] # we get the estimate of our resul
      correct += pred.eq(targets.data.view_as(pred)).sum() # sum up the corrected s
  test_loss /= len(test_loader.dataset)
  print(f'Test result on epoch {epoch}: Avg loss is {test_loss}, Accuracy: {100.*co

# ----------- <Your code> ---------------
optimizer = optim.Adam(model.parameters(), lr=0.01)
for x in range(4):
  train(x, model, optimizer)
  test(x, model)

# ----------- <End Your code> ---------------
```

```
Epoch 0: [0/60000] Loss: 3.6695616245269775
Epoch 0: [6400/60000] Loss: 1.1497571468353271
Epoch 0: [12800/60000] Loss: 0.5824035406112671
Epoch 0: [19200/60000] Loss: 0.5036520957946777
Epoch 0: [25600/60000] Loss: 0.5514126420021057
Epoch 0: [32000/60000] Loss: 0.4325895309448242
Epoch 0: [38400/60000] Loss: 0.3915180563926697
Epoch 0: [44800/60000] Loss: 0.3874974250793457
Epoch 0: [51200/60000] Loss: 0.1549413949251175
Epoch 0: [57600/60000] Loss: 0.39205214381217957
Test result on epoch 0: Avg loss is 0.3842582817792892, Accuracy: 88.0199966430664%
Epoch 1: [0/60000] Loss: 0.5150216817855835
Epoch 1: [6400/60000] Loss: 0.32011520862579346
Epoch 1: [12800/60000] Loss: 0.1276804357767105
Epoch 1: [19200/60000] Loss: 0.2583090662956238
Epoch 1: [25600/60000] Loss: 0.27325037121772766
Epoch 1: [32000/60000] Loss: 0.3006344735622406
Epoch 1: [38400/60000] Loss: 0.15581171214580536
Epoch 1: [44800/60000] Loss: 0.44465938210487366
Epoch 1: [51200/60000] Loss: 0.21590733528137207
Epoch 1: [57600/60000] Loss: 0.18032878637313843
Test result on epoch 1: Avg loss is 0.3043919304251671, Accuracy: 90.25999450683594%
Epoch 2: [0/60000] Loss: 0.21492154896259308
Epoch 2: [6400/60000] Loss: 0.5110961198806763
Epoch 2: [12800/60000] Loss: 0.112759068608284
Epoch 2: [19200/60000] Loss: 0.22185185551643372
Epoch 2: [25600/60000] Loss: 0.2940877377986908
Epoch 2: [32000/60000] Loss: 0.2301902323961258
Epoch 2: [38400/60000] Loss: 0.5253562927246094
Epoch 2: [44800/60000] Loss: 0.24128997325897217
Epoch 2: [51200/60000] Loss: 0.14765119552612305
Epoch 2: [57600/60000] Loss: 0.23065665364265442
Test result on epoch 2: Avg loss is 0.2078941892027855, Accuracy: 93.40999603271484%
Epoch 3: [0/60000] Loss: 0.2786734104156494
Epoch 3: [6400/60000] Loss: 0.2121688574552536
Epoch 3: [12800/60000] Loss: 0.3478432595729828
Epoch 3: [19200/60000] Loss: 0.08582475036382675
Epoch 3: [25600/60000] Loss: 0.21550384163856506
Epoch 3: [32000/60000] Loss: 0.23857267200946808
Epoch 3: [38400/60000] Loss: 0.25694605708122253
Epoch 3: [44800/60000] Loss: 0.24033263325691223
Epoch 3: [51200/60000] Loss: 0.1967705488204956
Epoch 3: [57600/60000] Loss: 0.25468340516090393
Test result on epoch 3: Avg loss is 0.17323281983733177, Accuracy: 94.6399993896484
4%
```

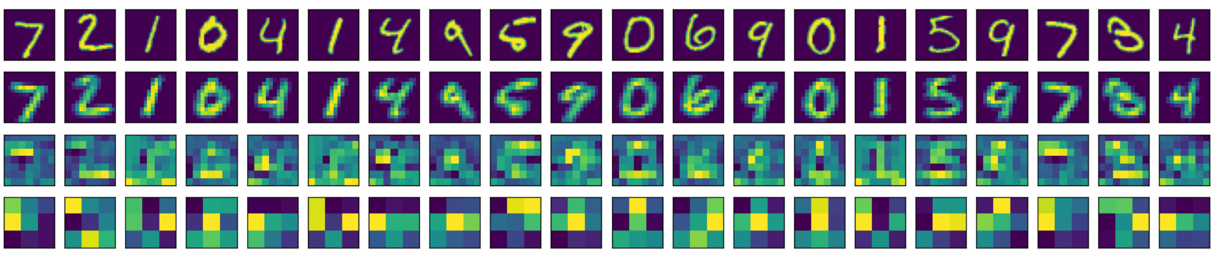
Task 3: Visualize the intermediate latent representations

- Plot the representations of 20 images from the test dataset in a supblots grid of shape (20, 4) (code already given for setting up these subplots) where the rows correspond to samples in the dataset and columns correspond to the representations produced by `compute_and_extract_representations`

Notes:

- We give code below for normalizing the image and plotting on an axis with a title.
- Make sure to set `model.eval()` when computing because of the batchnorm layers
- No title or ylabel is needed in this case.
- `z_list` is a list of 4 tensors of shape ([B, 1, 28, 28]). **Figure out how to pass through the assertion error and think why the batch dimension cannot be processed together.**

In [71]:
```
def plot_representation(z, ax):
  # Normalize image for visualization
  assert z.ndim == 3, 'Should be 3 dimensional tensor with C x H x W'
  z = (z - z.min())/(z.max() - z.min())
  if torch.is_tensor(z):  # Convert torch tensor to numpy if needed
    z = z.detach().cpu().numpy()
  ax.imshow(z.transpose((1,2,0)).squeeze(2))
  # Remove ticks and ticklabels to make plot clean
  ax.set_xticks([])
  ax.set_yticks([])
  ax.set_xticklabels([])
  ax.set_yticklabels([])

n_show = 20
fig, axes_mat = plt.subplots(4, n_show, figsize=[n_show, 4])
# ----------- <Your code> ---------------
model.eval()
out, z_list = model.compute_and_extract_representations(images)
for x in range(n_show):
  for y, z in enumerate(z_list):
    plot_representation(z[x], axes_mat[y][x])
# ----------- <End Your code> ---------------
```



Notice how the representations become more and more abstract as the depth increases.

Exercise 2: Clustering with different representations (35 points)

Task 1: Create simple numpy arrays of the representations

To perform further manipulations in numpy and scikit-learn, we will need to create simple numpy arrays for each representation. We provide the code for merging multiple batches. You will need to provide the code for extracting from the given data loader.

- Loop through the data loader and extract representations for each batch
- Append the labels and z_list to corresponding lists
- Break out of loop when the number extracted is n_extract or greater

The output of the merged lists should print the following for both train and test:

```
Types of merged lists
    [<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>]
Shapes of merged lists
    [(200, 1, 28, 28), (200, 1, 14, 14), (200, 1, 7, 7), (200, 1, 3, 3)]
Shape of merged labels
    (200,)
```

In [72]:
```
def extract_numpy_representations(model, data_loader, n_extract):
  extracted_z_lists = []
  labels_list = []
  # ------------- Your code ---------------
  for batch_idx, (data, target) in enumerate(data_loader):
    data = data.to(device)
```

```
    with torch.no_grad():
      _, z_list = model.compute_and_extract_representations(data)
    extracted_z_lists.append(z_list)
    labels_list.append(target)

    if len(labels_list) * len(data) >= n_extract:
      break
  # ------------- End your code ---------------
  # Check extracted_z_lists (type should be tensor)
  print(f'Types of first batch\n    {[type(z) for z in extracted_z_lists[0]]}')
  print(f'Shapes of first batch\n    {[z.shape for z in extracted_z_lists[0]]}')

  # Merge extracted z_lists and labels and make numpy arrays
  z_list_merge_np = [
    np.vstack([
      z_list[i].detach().cpu().numpy()
      for z_list in extracted_z_lists
    ])[:n_extract]  # Extract up to n_extract
    for i in range(len(extracted_z_lists[0]))
  ]
  print(f'Types of merged lists\n    {[type(z) for z in z_list_merge_np]}')
  print(f'Shapes of merged lists\n    {[z.shape for z in z_list_merge_np]}')
  labels_merged_np = np.concatenate([
    labels.detach().cpu().numpy()
    for labels in labels_list
  ])[:n_extract]  # Extract up to n_extract
  print(f'Shape of merged labels\n    {labels_merged_np.shape}')
  return z_list_merge_np, labels_merged_np

# Extract train and test samples
z_list_train, labels_train = extract_numpy_representations(model, train_loader, n_e
z_list_test, labels_test = extract_numpy_representations(model, test_loader, n_extr
# Extract regular train and test
x_test = z_list_test[0]
x_train = z_list_train[0]
```

```
Types of first batch
    [<class 'torch.Tensor'>, <class 'torch.Tensor'>, <class 'torch.Tensor'>, <class
'torch.Tensor'>]
Shapes of first batch
    [torch.Size([64, 1, 28, 28]), torch.Size([64, 1, 14, 14]), torch.Size([64, 1, 7,
7]), torch.Size([64, 1, 3, 3])]
Types of merged lists
    [<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <cla
ss 'numpy.ndarray'>]
Shapes of merged lists
    [(200, 1, 28, 28), (200, 1, 14, 14), (200, 1, 7, 7), (200, 1, 3, 3)]
Shape of merged labels
    (200,)
Types of first batch
    [<class 'torch.Tensor'>, <class 'torch.Tensor'>, <class 'torch.Tensor'>, <class
'torch.Tensor'>]
Shapes of first batch
    [torch.Size([128, 1, 28, 28]), torch.Size([128, 1, 14, 14]), torch.Size([128, 1,
7, 7]), torch.Size([128, 1, 3, 3])]
Types of merged lists
    [<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <cla
ss 'numpy.ndarray'>]
Shapes of merged lists
    [(200, 1, 28, 28), (200, 1, 14, 14), (200, 1, 7, 7), (200, 1, 3, 3)]
Shape of merged labels
    (200,)
```

Task 2: Perform K-means clustering on different representations

In this task, we will perform kmeans clustering on each of the latent representations of the test set and then evaluate the clustering based on the true class labels. A good discussion of clustering metrics can be found in scikit-learn's documentation on clustering metrics.

- Using scikit-learn's `sklearn.cluster.KMeans` estimator, perform kmeans with $k = 10$ and `random_state=0` on the latent representations and extract the cluster labels.
- Use `sklearn.metrics.adjusted_rand_score` to compute a score to evaluate the clustering based on the true class labels.

Notes:

- You will need to reshape the tensors into matrices immediately before passing into sklearn functions (you should keep the original data as is so that the images can be plotted, but just reshape immediately before passing into scikit-learn functions). Specifically, the arrays will have shape (B, C, H, W) and you should reshape to (B, C*HW*) before passing to scikit-learn functions.
- We provide code for plotting and evaluating your clustering.
- Note that clustering is unsupervised. What we're plotting here is the ten different clusters, not the ten different categories of true labels. Thus, the cluster index in the plotted image is not necessarily matched to the true label.
- Sometimes the `plot_cluster` will have white boxes if there are less than 5 samples in that cluster. Generally, if you use `n_clusters=10` for the clustering tasks, you will have none or only a few white boxes, which is okay.

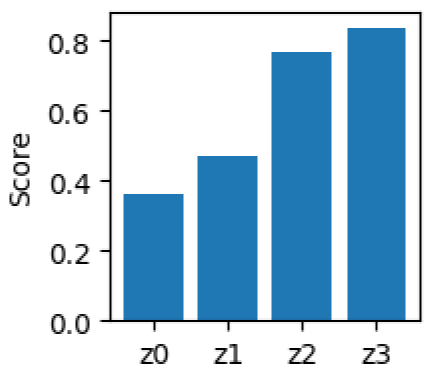
scikit-learn's documentation on clustering metrics

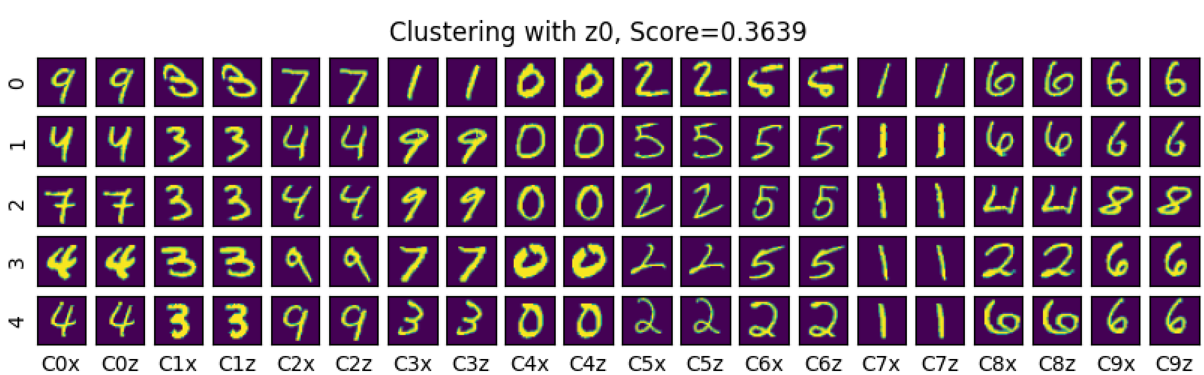
sklearn.cluster.KMeans

In [73]:
```python
def plot_cluster(cluster_labels, z_test, title):
  # Plot the top images in each cluster both in original space and latent represent
  n_samples_show, n_clusters = 5, 10
  nr, nc = n_samples_show, 2*n_clusters
  fig, axes_mat = plt.subplots(nr, nc, figsize=np.array([nc, nr])/2)
  axes_mat_list = np.split(axes_mat, n_clusters, axis=1)
  for ci, axes_mat in enumerate(axes_mat_list): # Loop over clusters
    sel = cluster_labels==ci
    z_cluster = z_test[sel][:n_samples_show]
    x_cluster = x_test[sel][:n_samples_show]
    for test_i, (z, x, axes) in enumerate(zip(z_cluster, x_cluster, axes_mat)):
      plot_representation(x, axes[0])
```

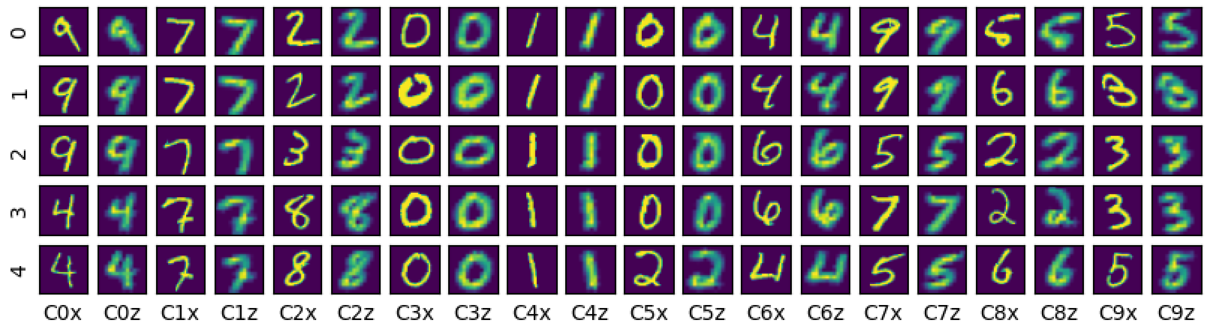
```
      plot_representation(z, axes[1])
      if ci == 0:
        axes[0].set_ylabel(test_i)
      if test_i == len(axes_mat)-1:
        axes[0].set_xlabel(f'C{ci}x')
        axes[1].set_xlabel(f'C{ci}z')
  fig.suptitle(title)
  plt.show()
```

In [74]:
```
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score

def cluster_and_score(z, true_labels):
  # ----------- <Your code> ---------------
  cluster_labels = KMeans(n_clusters=10, random_state=0).fit_predict(z.reshape(z.sh
  score = adjusted_rand_score(true_labels, cluster_labels)
  # ----------- <End your code> ---------------
  return cluster_labels, score

fig = plt.figure(figsize=(2,2))
plt.bar([f'z{zi}' for zi in range(len(z_list_test))],
        [cluster_and_score(z_test, labels_test)[1] for z_test in z_list_test])
plt.ylabel('Score')
for zi, z_test in enumerate(z_list_test):
  cluster_labels, score = cluster_and_score(z_test, labels_test)
  plot_cluster(cluster_labels, z_test, f'Clustering with z{zi}, Score={score:.4f}')
```

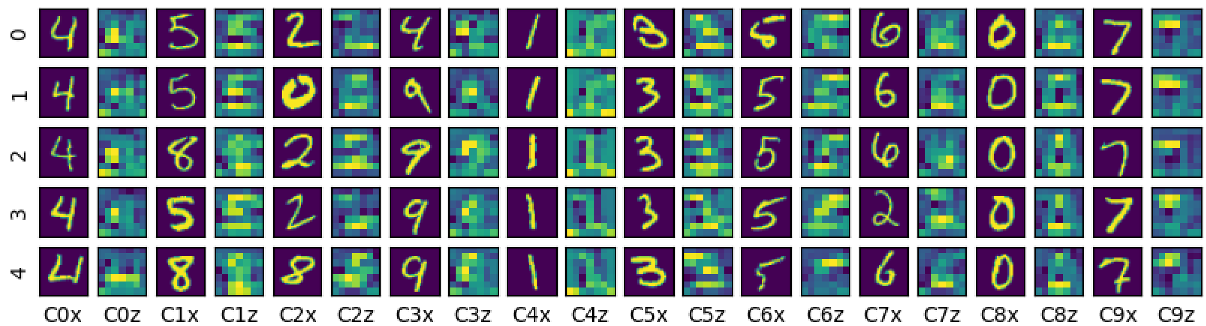




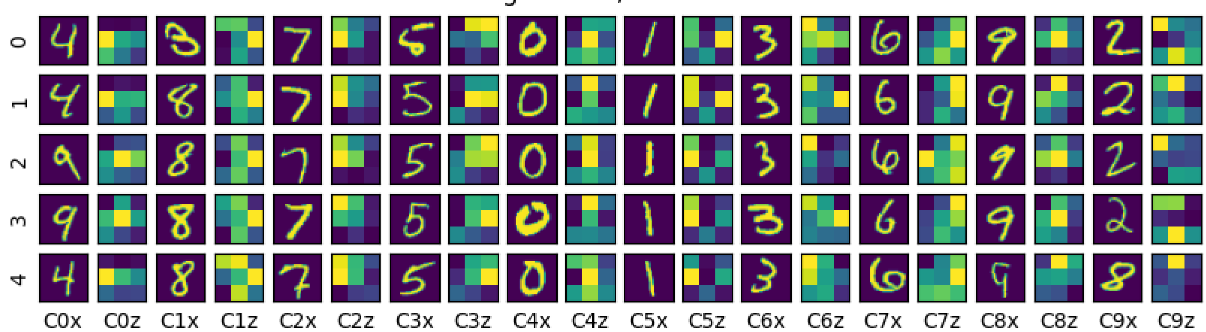
Clustering with z1, Score=0.4711



Clustering with z2, Score=0.7659



Clustering with z3, Score=0.8382



Notice how the 3x3 pattern for the last representation looks similar across the samples.

Exercise 3: Nearest neigbhors methods using representations (10 points)

Task 1: Compute and plot nearest neigbhors in different representations

We will now compute the 1 nearest neighbor (i.e., `n_neighbors=1`) of test points compared to train points in different representations.

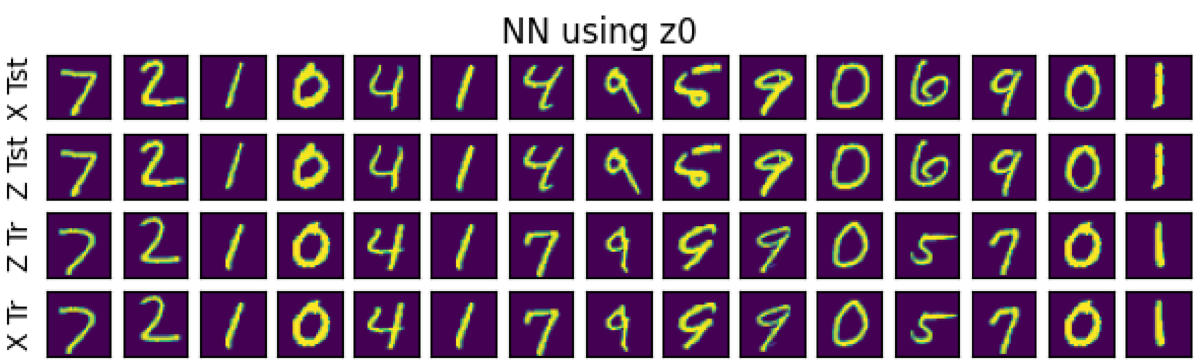
- Loop through the representations for the train and test numpy arrays (i.e., `z_list_train` and `z_list_test`).
- For each representaiton from the different layers, compute the *training* indices corresponding to the nearest neighbor of first 15 *testing* indices.
- Plot the neighbors by passing the test indices and corresponding nearest neighbor training indices along with the corresponding train and test representations and a title that describes which representation into `plot_neighbor`.

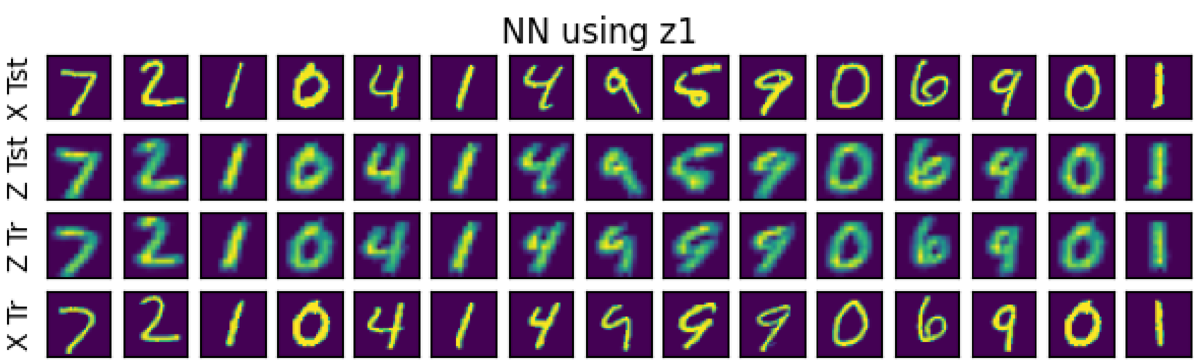
Notes:

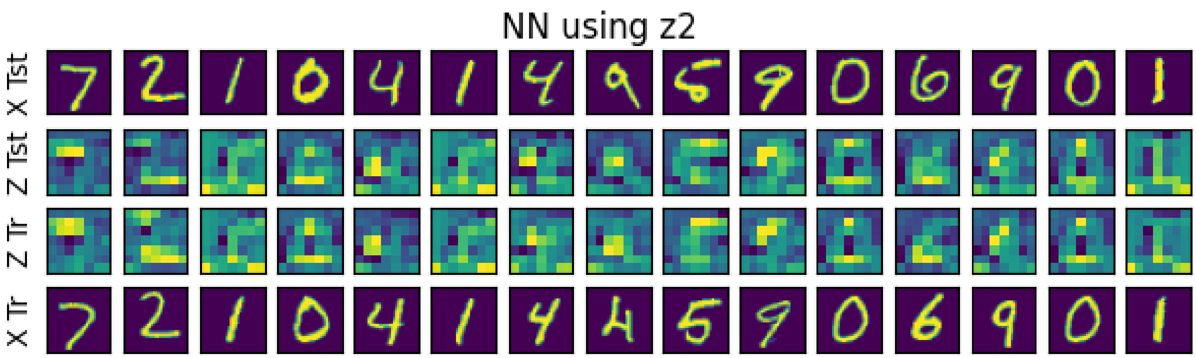
- See note above about reshaping tensors immediately before passing to scikit-learn functions which expect a matrix.
- The `sklearn.neighbors.NearestNeighbors` class and the `kneighbors` method may be very helpful. The data that is passed to `fit` will be the training data and the data passed to `kneighbors` should be the new test data.
- The test indices should just be `np.arange(15)` assuming that you find the nearest training points for the first 15 points in the test dataset.

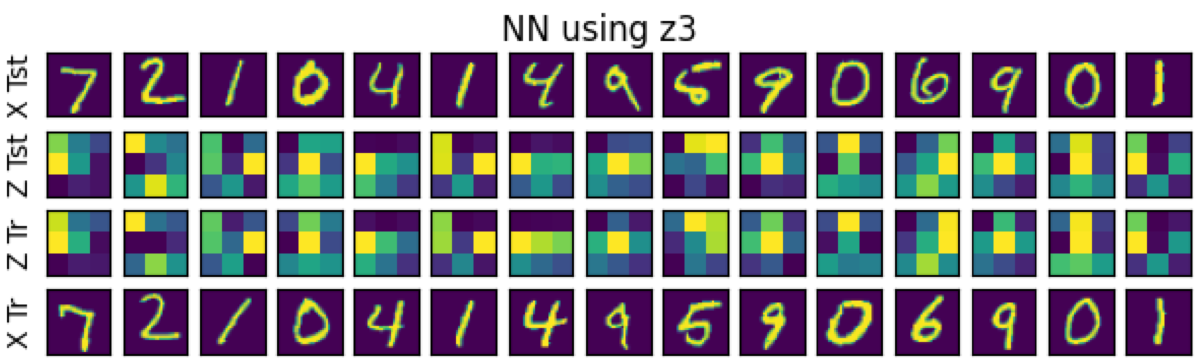
In [75]:
```
def plot_neigbhor(test_ind, nearest_train_ind, z_test, z_train, title):
  '''
  Plots the original test image, the test image representation,
    the nearest train image representation, the nearest original train image.
```

```
  '''
  assert len(test_ind) == len(nearest_train_ind), 'Test and train indices should be
  n_test = len(test_ind)
  fig, axes_mat = plt.subplots(4, n_test, figsize=np.array([n_test, 4])/2)
  for test_i, nearest_train_i, axes in zip(test_ind, nearest_train_ind, axes_mat.T)
    plot_representation(x_test[test_i], axes[0])
    plot_representation(z_test[test_i], axes[1])
    plot_representation(z_train[nearest_train_i], axes[2])
    plot_representation(x_train[nearest_train_i], axes[3])
    if test_i == 0:
      for lab, ax in zip(['X Tst', 'Z Tst', 'Z Tr', 'X Tr'], axes):
        ax.set_ylabel(lab)
  fig.suptitle(title)

# --------- Your code ---------
from sklearn.neighbors import NearestNeighbors

for i, (z_train, z_test) in enumerate(zip(z_list_train, z_list_test)):
  z_train_new = z_train.reshape(z_train.shape[0], -1)
  z_test_new = z_test.reshape(z_test.shape[0], -1)

  nn = NearestNeighbors(n_neighbors=1)
  nn.fit(z_train_new)
  dist, idx = nn.kneighbors(z_test_new[:15])

  plot_neigbhor(np.arange(15), idx.flatten(), z_test, z_train, f"NN using z{i}")
# --------- End your code ---------
```









Exercise 4: Further Understanding the Latent Representations and the Domain Shift (15 points)

In this exercise, we'll explore how latent representations change with different output layers and examine the concept of **domain shift**. Domain shift refers to the difference in data distribution between training and test datasets. In this case, we'll compare the MNIST dataset (greyscale digits) with the SVHN dataset (color street view house numbers) to observe how models trained on MNIST handle data from a different domain like SVHN.

Task 1: Load the SVHN Dataset

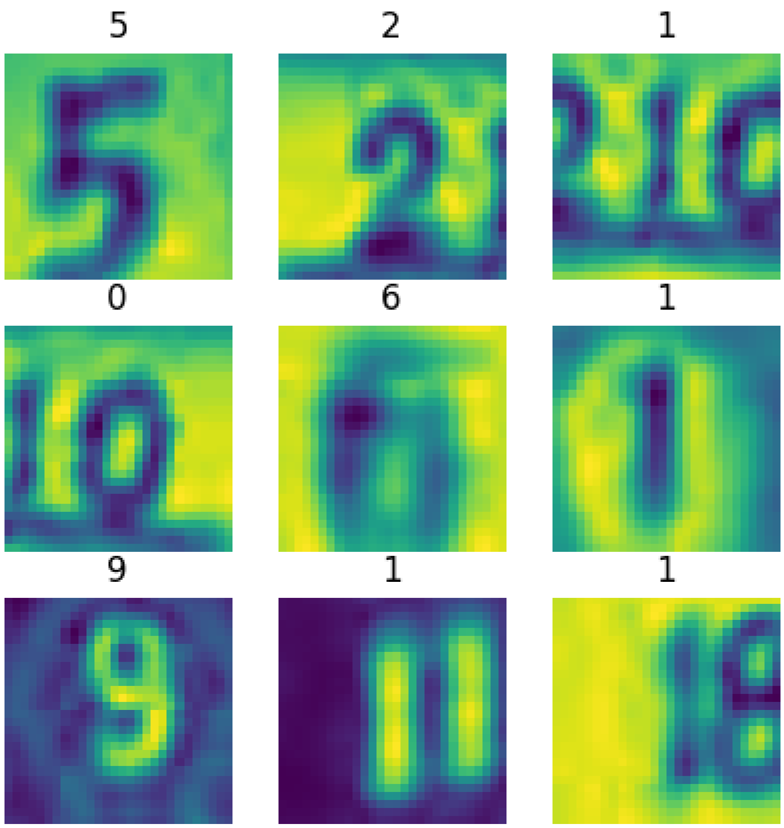
In this task, you'll load the **SVHN** dataset so that it structurally resembles the **MNIST** dataset. SVHN is originally composed of 32×32 RGB images, so we must preprocess the dataset accordingly:

- Resize images to **28×28 pixels** to match MNIST's dimensions.
- Convert images from **RGB to grayscale** (i.e., from 3 channels to 1).
- Normalize the images using the same mean and standard deviation as used for MNIST.

Use appropriate transforms to achieve the above, and then create DataLoaders for training and testing data with appropriate batch sizes.

In [76]:
```
###### Your code starts here ######
transform = torchvision.transforms.Compose(
  [torchvision.transforms.ToTensor(),
   torchvision.transforms.Resize((28, 28)),
   torchvision.transforms.Grayscale(1),
   torchvision.transforms.Normalize((0.1307,),(0.3081,))])
train_dataset = torchvision.datasets.SVHN('./data', split='train', download=True, t
test_dataset = torchvision.datasets.SVHN('./data', split='test', download=True, tra
batch_size_train, batch_size_test = 64, 128
svhn_train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_siz
svhn_test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size_
####### Your code ends here ######

# plot 3x3 samples from the SVHN dataset
classes = np.arange(10)
batch_idx, (images, targets) = next(enumerate(svhn_test_loader))
fig, ax = plt.subplots(3, 3, figsize=(5, 5))
```

```python
for i in range(3):
  for j in range(3):
    image = images[i*3+j].permute(1, 2, 0)
    image = image/2 + 0.5
    ax[i, j].imshow(image.squeeze(2))
    ax[i, j].set_title(f'{classes[targets[i*3+j]]}')
    ax[i, j].axis('off')
fig.show()
```



Task 2: Extract Latent Representations for SVHN Test Samples

Similar to how you extracted latent representations (`z_list`) for MNIST in Exercise 3, repeat this for **SVHN test samples only**. These latent features will help us visualize how SVHN inputs are interpreted by the model trained on MNIST.

Once the representations are extracted:

- Select a few test samples (e.g., 15)
- For each intermediate representation `z_i`, use **Nearest Neighbors** to find the closest training sample from MNIST.
- Visualize the test image from SVHN along with its nearest neighbor in the MNIST dataset for each layer.

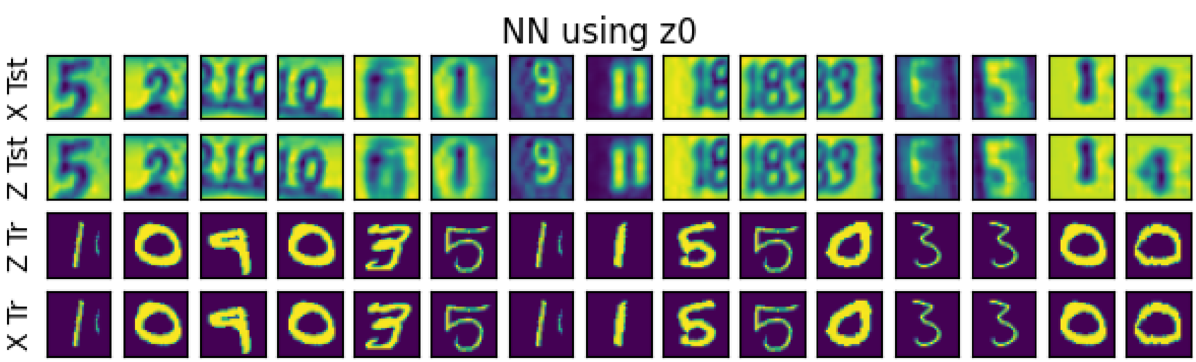
This helps in understanding how similar or different the latent space is for both datasets, and where the shift becomes most apparent.

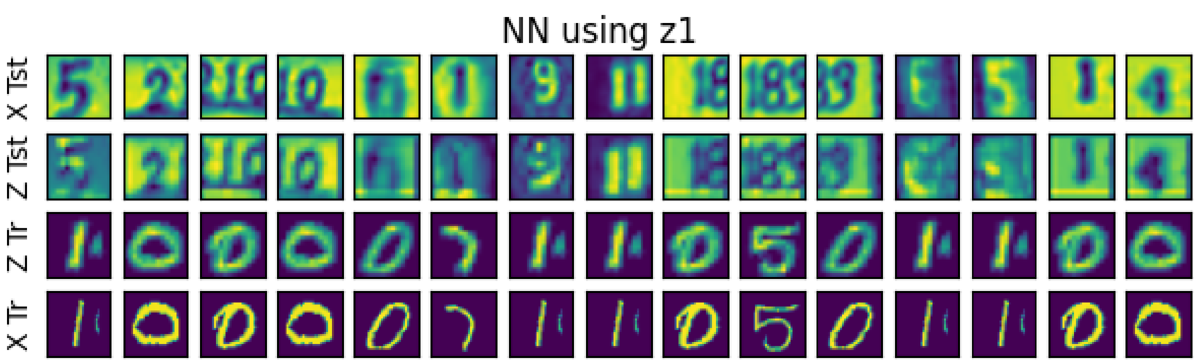
Note: Don't be surprised if the model fails to classify the SVHN images correctly. This is expected since the model was trained on MNIST, and the two datasets have different distributions. The goal here is to visualize the latent representations and understand how the model interprets data from a different domain.

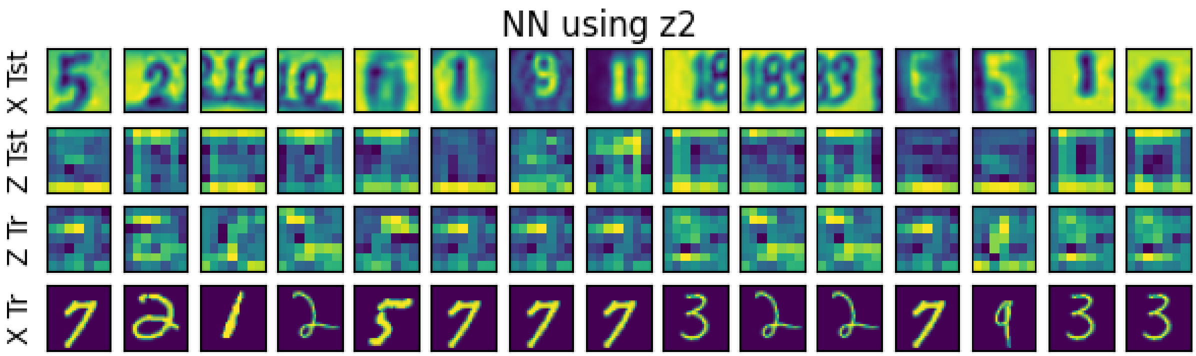
In [77]:
```python
###### Your code starts here ######
images, labels_svhn = next(iter(svhn_test_loader)) # get a batch
images = images.to(device)
z_list_test_svhn, svhn_labels = extract_numpy_representations(model, svhn_test_load
####### Your code ends here ######
```

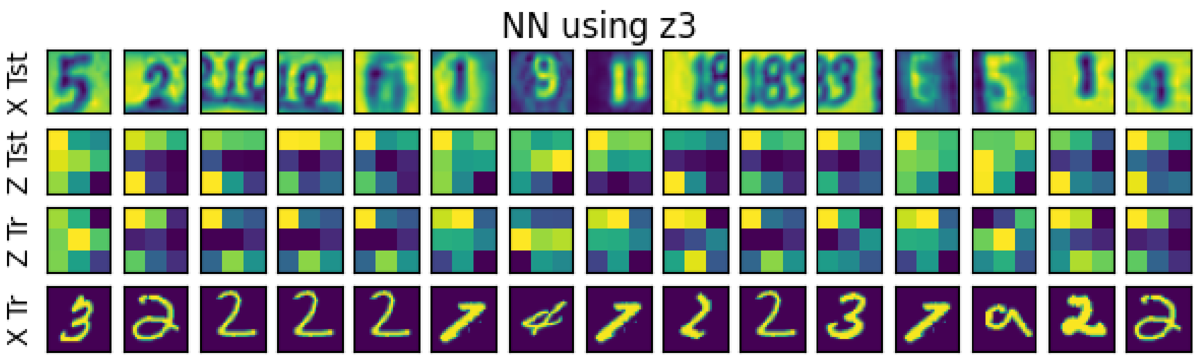
```
Types of first batch
    [<class 'torch.Tensor'>, <class 'torch.Tensor'>, <class 'torch.Tensor'>, <class
'torch.Tensor'>]
Shapes of first batch
    [torch.Size([128, 1, 28, 28]), torch.Size([128, 1, 14, 14]), torch.Size([128, 1,
7, 7]), torch.Size([128, 1, 3, 3])]
Types of merged lists
    [<class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <class 'numpy.ndarray'>, <cla
ss 'numpy.ndarray'>]
Shapes of merged lists
    [(200, 1, 28, 28), (200, 1, 14, 14), (200, 1, 7, 7), (200, 1, 3, 3)]
Shape of merged labels
    (200,)
```

In [78]:
```python
n_test = 15
x_test = z_list_test_svhn[0]
x_train = z_list_train[0]

###### Your code starts here ######
for zi, (z_train, z_test) in enumerate(zip(z_list_train, z_list_test_svhn)):
  z_train_new = z_train.reshape(z_train.shape[0], -1)
  z_test_new = z_test.reshape(z_test.shape[0], -1)

  nn = NearestNeighbors(n_neighbors=1)
  nn.fit(z_train_new)
  dist, idx = nn.kneighbors(z_test_new[:n_test])

  plot_neigbhor(np.arange(n_test), idx.flatten(), z_test, z_train, f"NN using z{zi}


####### Your code ends here ######
```









Task 3: t-SNE Visualization of Latent Space

Use **t-SNE** to reduce the dimensionality of each intermediate representation (`z_list`) for both MNIST and SVHN and visualize them in 2D space.

Complete the code that will use `plot_combined_tsne` function to visualize the t-SNE plots for both datasets and all four representations.

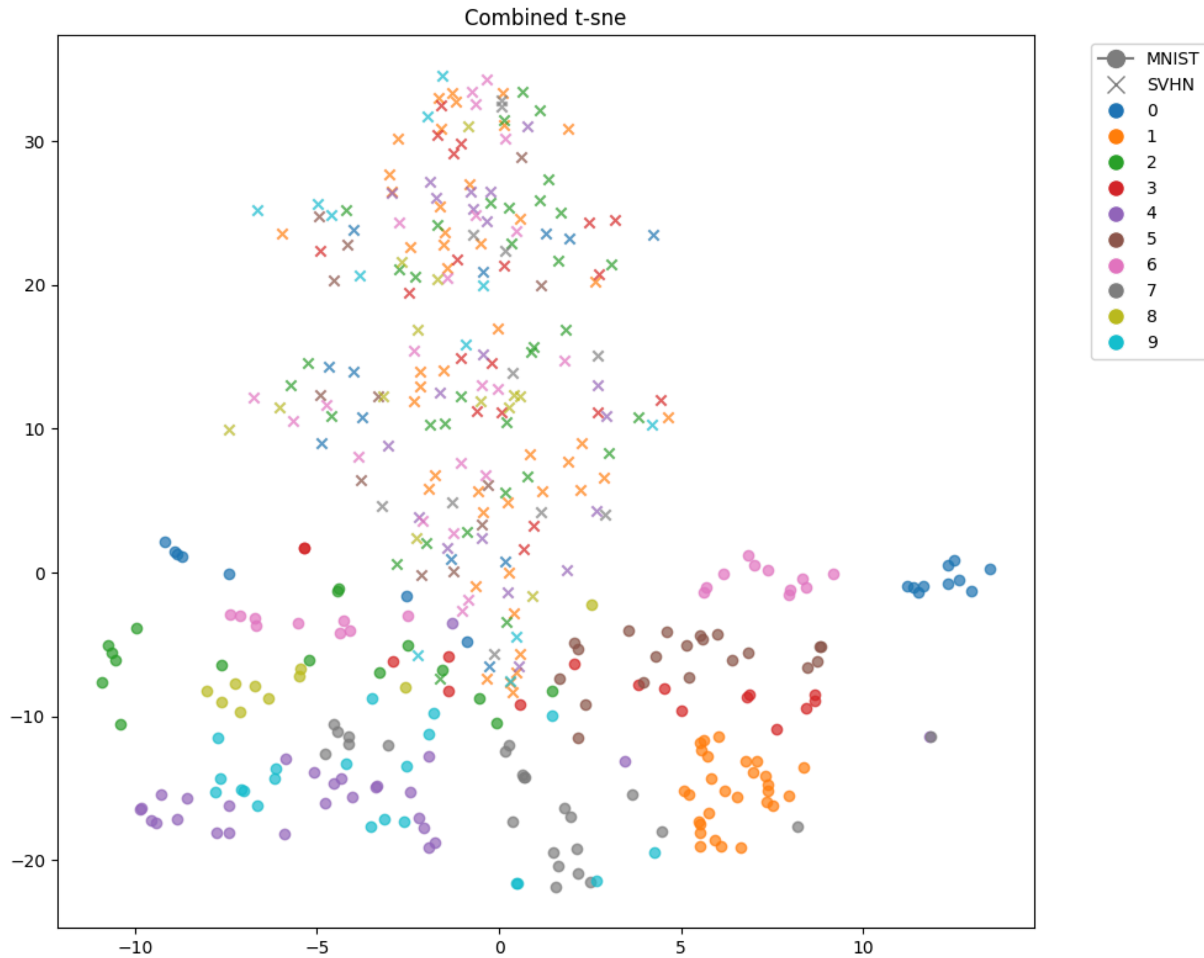
Important:

1. Understand how similar classes (1 and 7, 4 and 9) are clustered together in the latent space. This is a crucial aspect of understanding how the model generalizes across different datasets.
2. Also, note how the clusters are getting disentangled as we go deeper into the network. This is a key insight into how the model learns to represent data at different levels of abstraction.

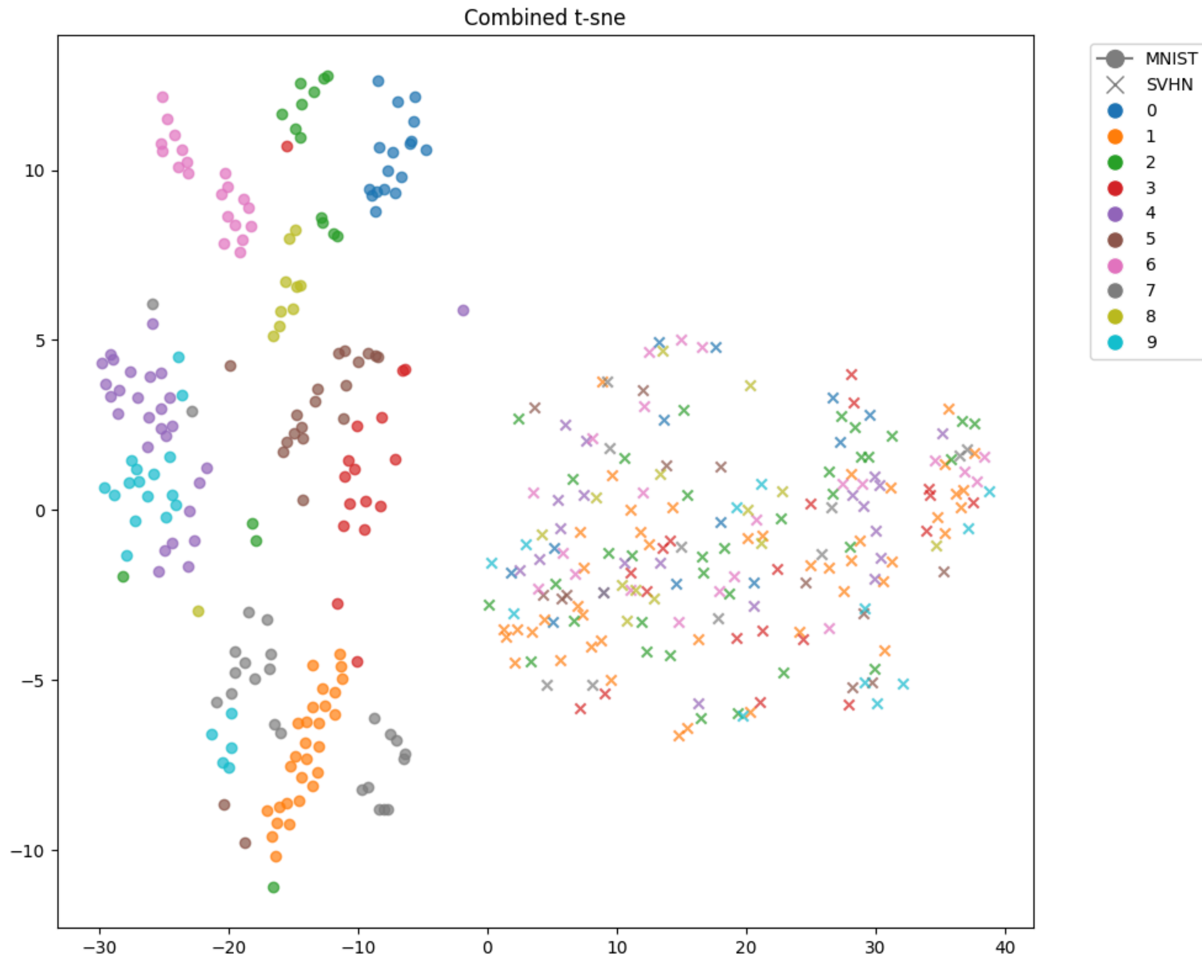
In [79]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from matplotlib.lines import Line2D

def plot_combined_tsne(z_mnist, labels_mnist, z_svhn, labels_svhn, title):
    # Flatten and combine
    z_combined = np.vstack([z_mnist.reshape(z_mnist.shape[0], -1),
                            z_svhn.reshape(z_svhn.shape[0], -1)])
    labels_combined = np.concatenate([labels_mnist, labels_svhn])
    markers = np.array(['o'] * len(labels_mnist) + ['x'] * len(labels_svhn))

    # t-SNE projection
    z_2d = TSNE(n_components=2, random_state=0).fit_transform(z_combined)

    # Plotting
    fig, ax = plt.subplots(figsize=(10, 8))
    cmap = plt.cm.get_cmap('tab10', 10)

    for i in range(10):
        for marker in ['o', 'x']:
            mask = (labels_combined == i) & (markers == marker)
```

```
            ax.scatter(z_2d[mask, 0], z_2d[mask, 1], marker=marker, color=cmap(i),

    # Legends
    dataset_legend = [Line2D([0], [0], marker=m, color='gray', label=l,
                             linestyle='None' if m == 'x' else '-', markerfacecolor
                      for m, l in zip(['o', 'x'], ['MNIST', 'SVHN'])]

    class_legend = [Line2D([0], [0], marker='o', color='w', label=str(i),
                           markerfacecolor=cmap(i), markersize=10) for i in range(1

    ax.legend(handles=dataset_legend + class_legend, loc='upper left', bbox_to_anch
    ax.set_title(title)
    plt.tight_layout()
    plt.show()


####### Your code starts here ######
for x in range(4):
  plot_combined_tsne(z_list_test[x], labels_test, z_list_test_svhn[x], svhn_labels,




####### Your code ends here ######
```

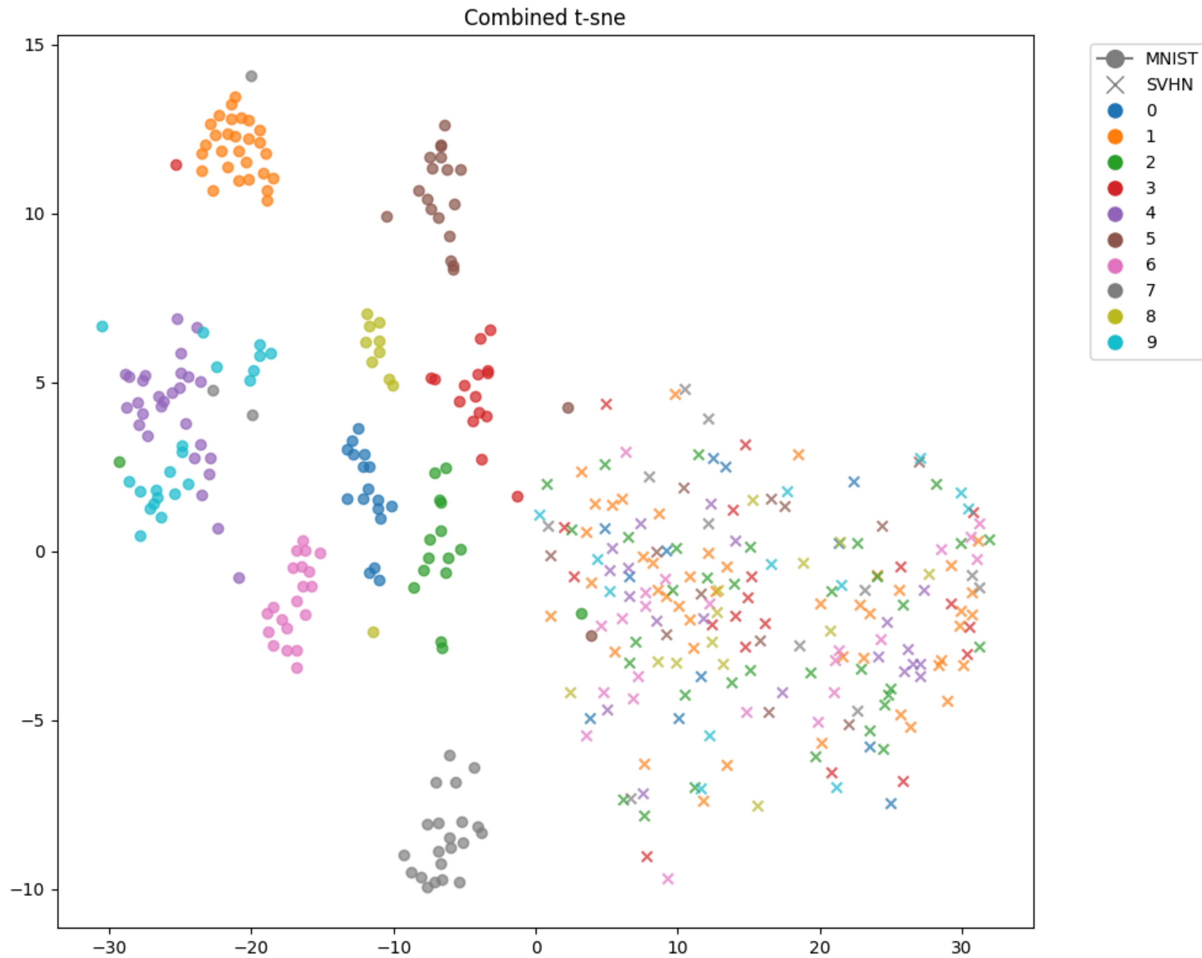
```
<ipython-input-79-59f753240139>:18: MatplotlibDeprecationWarning: The get_cmap funct
ion was deprecated in Matplotlib 3.7 and will be removed in 3.11. Use ``matplotlib.c
olormaps[name]`` or ``matplotlib.colormaps.get_cmap()`` or ``pyplot.get_cmap()`` ins
tead.
  cmap = plt.cm.get_cmap('tab10', 10)
```



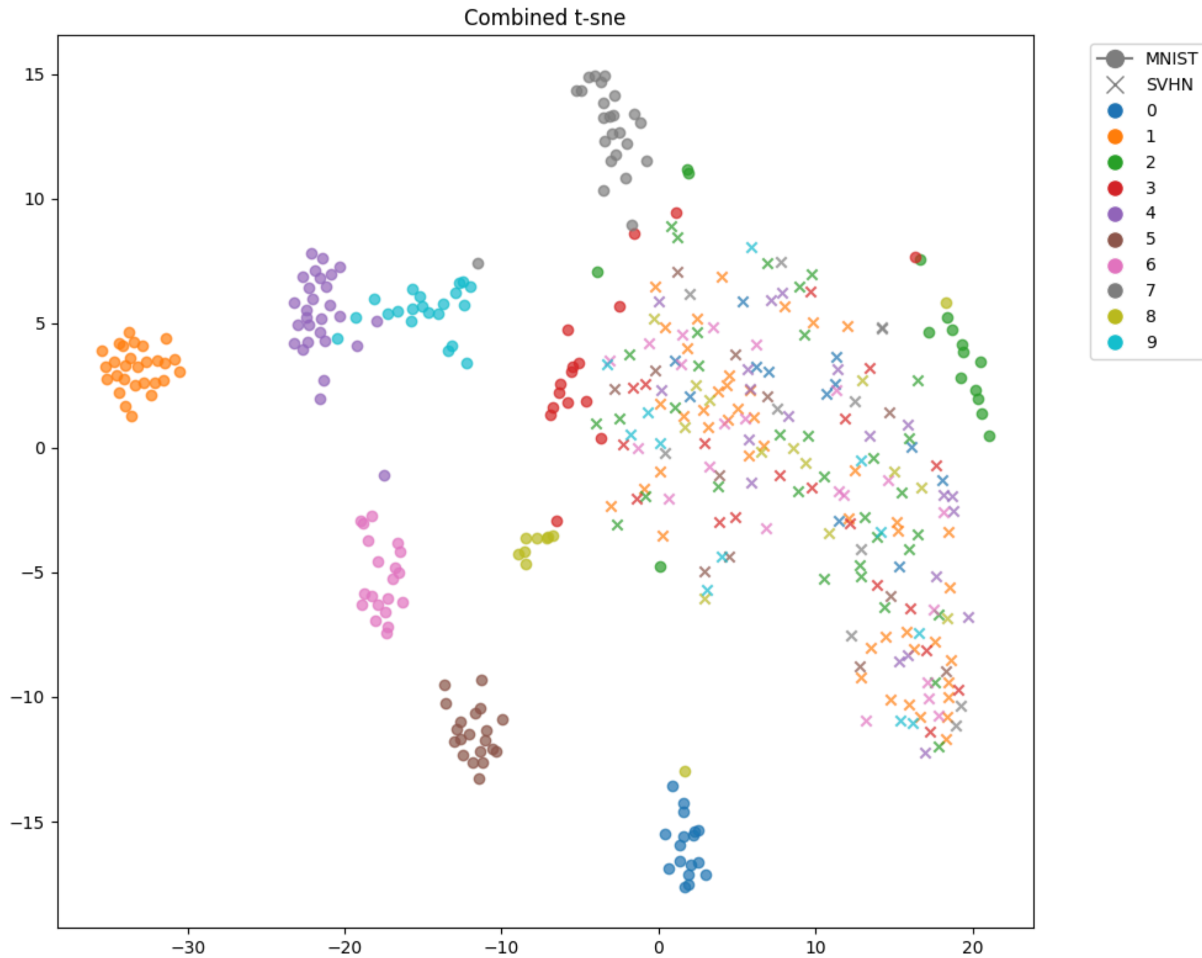

```
<ipython-input-79-59f753240139>:18: MatplotlibDeprecationWarning: The get_cmap funct
ion was deprecated in Matplotlib 3.7 and will be removed in 3.11. Use ``matplotlib.c
olormaps[name]`` or ``matplotlib.colormaps.get_cmap()`` or ``pyplot.get_cmap()`` ins
tead.
  cmap = plt.cm.get_cmap('tab10', 10)
```




```
<ipython-input-79-59f753240139>:18: MatplotlibDeprecationWarning: The get_cmap funct
ion was deprecated in Matplotlib 3.7 and will be removed in 3.11. Use ``matplotlib.c
olormaps[name]`` or ``matplotlib.colormaps.get_cmap()`` or ``pyplot.get_cmap()`` ins
tead.
  cmap = plt.cm.get_cmap('tab10', 10)
```




```
<ipython-input-79-59f753240139>:18: MatplotlibDeprecationWarning: The get_cmap funct
ion was deprecated in Matplotlib 3.7 and will be removed in 3.11. Use ``matplotlib.c
olormaps[name]`` or ``matplotlib.colormaps.get_cmap()`` or ``pyplot.get_cmap()`` ins
tead.
  cmap = plt.cm.get_cmap('tab10', 10)
```



Combined t-sne
MNIST
SVHN
0
1
2
3
4
5
6
7
8
9
15
10
5
0
−5
−10
−15
−30
−20
−10
0
10
20