Clustering and Dimensionality Reduction : Course Project

Exploring Dimensionality Reduction Techniques on MNIST Handwritten Digit Dataset

Rhichard Koh

In [ ]:
```
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from sklearn.decomposition import PCA
from sklearn.metrics import mean_squared_error
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from matplotlib import pyplot as plt
```

Importing Data

In [ ]:
```
# Transform to convert images to PyTorch tensors and normalize them
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize around the mean 0.5
])

# Download and load the training data
trainset = datasets.MNIST(root='./data', download=True, train=True, transform=transfor
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# Download and load the test data
testset = datasets.MNIST(root='./data', download=True, train=False, transform=transfor
testloader = DataLoader(testset, batch_size=64, shuffle=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNI
ST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 219483760.03it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNI
ST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 49503757.18it/s]
```

```
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIS
T/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:00<00:00, 78595926.91it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIS
T/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 16638016.39it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

PCA

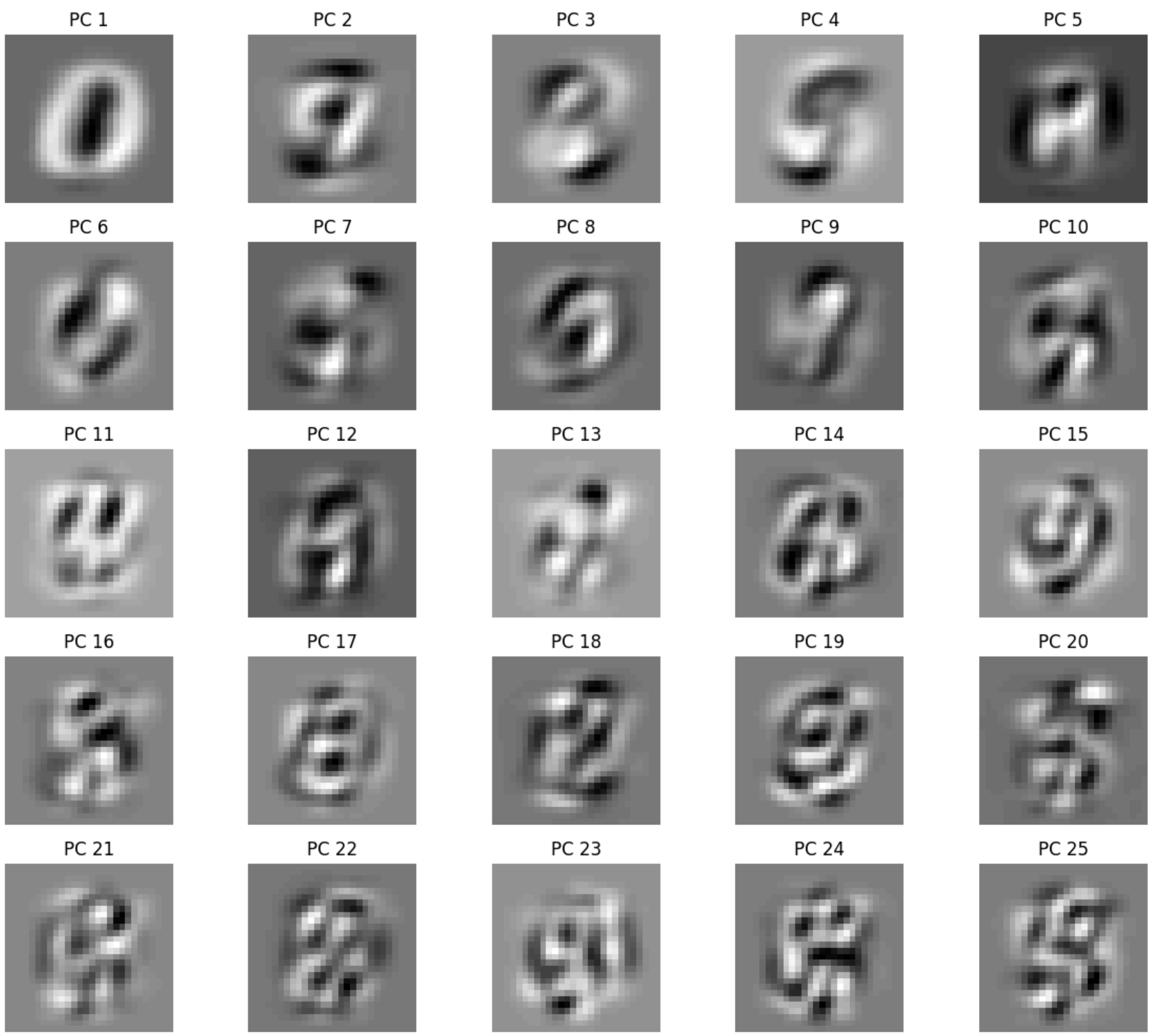
In [ ]:

```python
# Flatten the images for PCA
X_train_flattened = trainset.data.view(len(trainset), -1).numpy()
X_test_flattened = testset.data.view(len(testset), -1).numpy()

# Normalize data
X_train_flattened = X_train_flattened / 255.0
X_test_flattened = X_test_flattened / 255.0

# PCA with different number of components
components = [50, 100, 150, 200]
for n_components in components:
    pca = PCA(n_components=n_components)
    pca.fit(X_train_flattened)
    X_train_pca = pca.transform(X_train_flattened)
    X_test_pca = pca.transform(X_test_flattened)
    X_reconstructed_pca = pca.inverse_transform(X_train_pca)
    reconstruction_loss = mean_squared_error(X_train_flattened, X_reconstructed_pca)

    # Logistic regression classifier to classify our reconstructed images
    lr = LogisticRegression(max_iter=1000, solver='sag', multi_class='multinomial')
    lr.fit(X_train_pca, trainset.targets.numpy())

    # Evaluating on a subset of the test set
    validation_index = np.random.choice(len(testset), 1000, replace=False)
    validation_images_pca = X_test_pca[validation_index]
    validation_labels = testset.targets.numpy()[validation_index]

    validation_predictions = lr.predict(validation_images_pca)
    accuracy = accuracy_score(validation_labels, validation_predictions)
    print(f'Reconstruction loss for PCA with {n_components} components: {reconstructic
```

```
Reconstruction loss for PCA with 50 components: 0.011795084610364161 Classification A
ccuracy: 0.924
Reconstruction loss for PCA with 100 components: 0.005756633405031867 Classification
Accuracy: 0.93
Reconstruction loss for PCA with 150 components: 0.0034891502018314425 Classification
Accuracy: 0.924
Reconstruction loss for PCA with 200 components: 0.0022767954728689765 Classification
Accuracy: 0.926
```

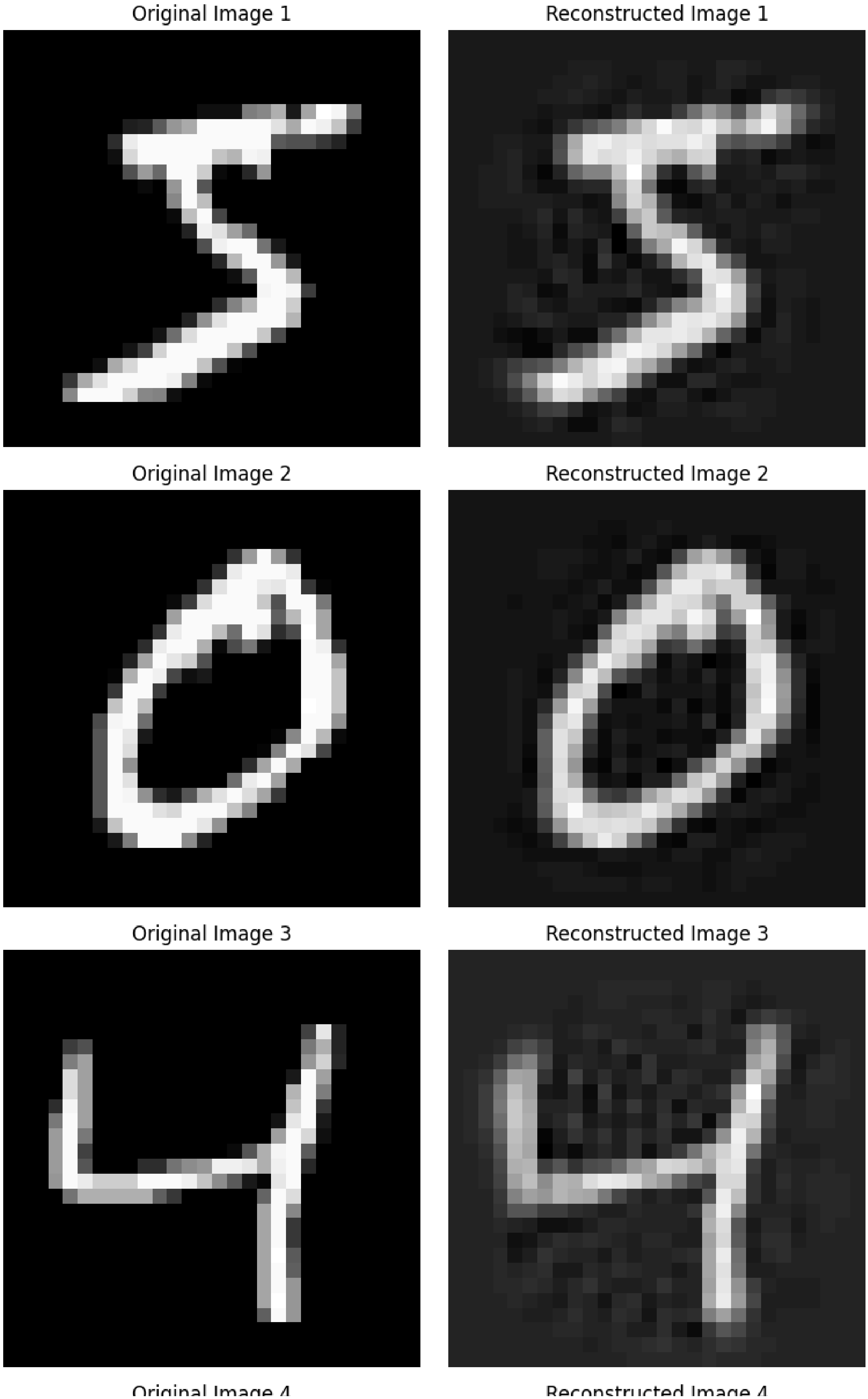
The best scoring minimal components is 100 as it has the highest accuracy and a decent reconstruction loss.

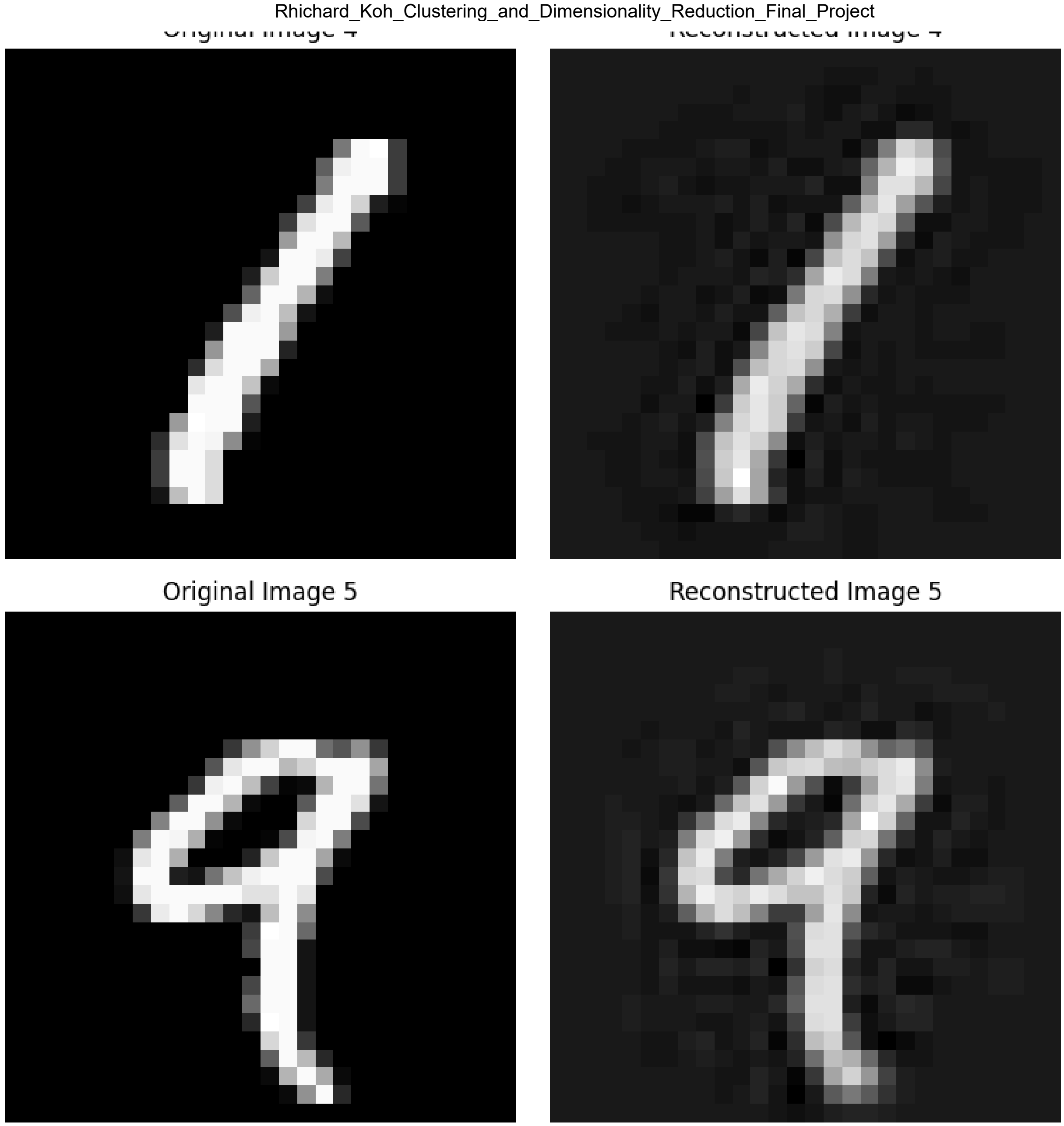
In [ ]:
```
# Visualizing the first 25 PCA'S
fig, axes = plt.subplots(5, 5, figsize=(12, 10))

for i in range(25):
    row = i // 5
    col = i % 5
    pc_image = pca.components_[i].reshape(28, 28)  # Reshape the principal component t
    axes[row, col].imshow(pc_image, cmap='gray')
    axes[row, col].set_title(f'PC {i+1}')
    axes[row, col].axis('off')

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



In [ ]:
```
# Comparing the reconstructed images to the original images
fig, axes = plt.subplots(5, 2, figsize=(8, 4 * 5))

for i in range(5):
    # Original image
    original_image = X_train_flattened[i].reshape(28, 28)
```

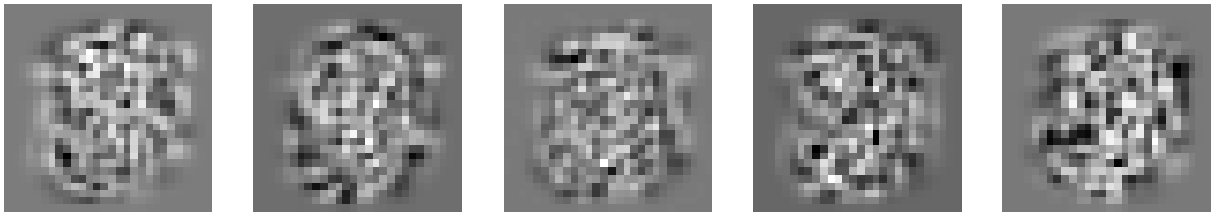
```
    # Reconstructed image after PCA
    reconstructed_image = pca.inverse_transform(X_train_pca[i]).reshape(28, 28)

    # Plot original image
    axes[i, 0].imshow(original_image, cmap='gray')
    axes[i, 0].set_title(f'Original Image {i+1}')
    axes[i, 0].axis('off')

    # Plot reconstructed image
    axes[i, 1].imshow(reconstructed_image, cmap='gray')
    axes[i, 1].set_title(f'Reconstructed Image {i+1}')
    axes[i, 1].axis('off')

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


Original Image 1
Reconstructed Image 1
Original Image 2
Reconstructed Image 2
Original Image 3
Reconstructed Image 3
Original Image 4
Reconstructed Image 4



In [ ]:
```
def generate_digits(n_digits):
    fig, axes = plt.subplots(1, n_digits, figsize=(10, 2))
    for i in range(n_digits):
        # Randomly sample from the distribution of the PCA coefficients
        random_sample = np.random.normal(0, 1, pca.n_components)

        # Generate new data by inverse transforming the random sample
        new_digit = pca.inverse_transform(random_sample)

        # Rescale to original range and reshape to 28x28
        new_digit_image = new_digit.reshape(28, 28)

        # Plot
        ax = axes[i]
        ax.imshow(new_digit_image, cmap='gray')
        ax.axis('off')
    plt.show()
```

```
# Generate and display 5 new digits
generate_digits(5)
```



The images were generated by selecting random points within a space transformed by Principal Component Analysis (PCA) and then converting these points back into the format of the original images. As a result, these images might not look exactly like any specific digits from the MNIST dataset. Instead, they display variations based on the major trends or features identified by the PCA.

Autoencoder

In [ ]:

```
import torch.nn as nn
import torch.optim as optim

class Autoencoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Linear(input_size, hidden_size)
        self.decoder = nn.Linear(hidden_size, input_size)

    def forward(self, x):
        x = torch.flatten(x, start_dim=1)
        encoded = torch.relu(self.encoder(x))
        decoded = torch.sigmoid(self.decoder(encoded))
        return decoded

# Train autoencoder
def train_autoencoder(hidden_size, results={}):

    model = Autoencoder(784, hidden_size)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-3)

    # Training Loop
    num_epochs = 8
    for epoch in range(num_epochs):
        total_loss = 0
        for images, _ in trainloader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, torch.flatten(images, start_dim=1))
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        average_loss = total_loss / len(trainloader)

    results[hidden_size] = {'reconstruction_loss': average_loss}
```

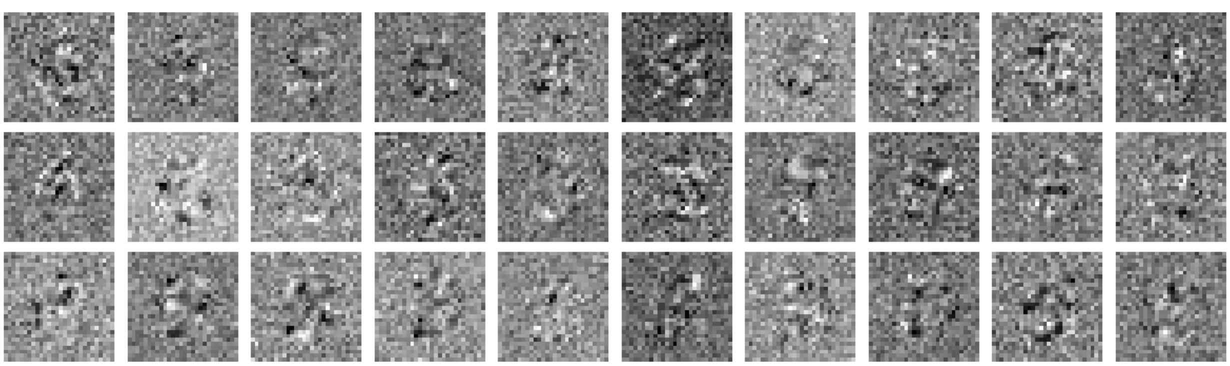
```
    encoded_train = []
    train_labels = []
    for data, target in trainloader:
        encoded = model(data)
        encoded_train.append(encoded.detach().cpu().numpy())
        train_labels.append(target.numpy())
    encoded_train = np.concatenate(encoded_train, axis=0)
    train_labels = np.concatenate(train_labels, axis=0)

    clf = LogisticRegression(max_iter=1000, solver='sag', multi_class='multinomial')
    clf.fit(encoded_train, train_labels)

    encoded_val = []
    val_labels = []
    for data, target in testloader:
        encoded = model(data)
        encoded_val.append(encoded.detach().cpu().numpy())
        val_labels.append(target.numpy())
    encoded_val = np.concatenate(encoded_val, axis=0)
    val_labels = np.concatenate(val_labels, axis=0)
    val_predictions = clf.predict(encoded_val)
    accuracy = accuracy_score(val_labels, val_predictions)

    results[hidden_size]['classification_accuracy'] = accuracy
    print(f'Encoding Dim: {hidden_size}, Classification Accuracy: {accuracy}')

    return model, loss.item(), results

hidden_sizes = [50, 100, 150, 200]
results = {}
for hidden_size in hidden_sizes:
    model, loss, results = train_autoencoder(hidden_size,results)
    print(f'Reconstruction loss for Autoencoder with {hidden_size} hidden units:', los
for encoding_dim, metrics in results.items():
    print(f'{encoding_dim}: Reconstruction Loss: {metrics["reconstruction_loss"]}, Cla
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_sag.py:350: Convergence
Warning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
Encoding Dim: 50, Classification Accuracy: 0.8038
Reconstruction loss for Autoencoder with 50 hidden units: 0.9075490236282349
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_sag.py:350: Convergence
Warning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
Encoding Dim: 100, Classification Accuracy: 0.8621
Reconstruction loss for Autoencoder with 100 hidden units: 0.8906221985816956
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_sag.py:350: Convergence
Warning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
Encoding Dim: 150, Classification Accuracy: 0.8531
Reconstruction loss for Autoencoder with 150 hidden units: 0.8868787884712219
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_sag.py:350: Convergence
Warning: The max_iter was reached which means the coef_ did not converge
  warnings.warn(
```

```
Encoding Dim: 200, Classification Accuracy: 0.8583
Reconstruction loss for Autoencoder with 200 hidden units: 0.883091390132904
50: Reconstruction Loss: 0.9358660034152236, Classification Accuracy: 0.8038
100: Reconstruction Loss: 0.9164198400623509, Classification Accuracy: 0.8621
150: Reconstruction Loss: 0.9233703469035468, Classification Accuracy: 0.8531
200: Reconstruction Loss: 0.9157706576623896, Classification Accuracy: 0.8583
```

The best minimum hidden encoding dimmension was 100 because it had the highest classification accuracy while still having the 2nd lowest reconstruction loss.

In [ ]:
```
hidden_size = 100
autoencoder = Autoencoder(784, hidden_size)
criterion = nn.MSELoss()
optimizer = optim.Adam(autoencoder.parameters(), lr=0.001)

for epoch in range(8):
    for data, _ in trainloader:
        optimizer.zero_grad()
        output = autoencoder(data)
        loss = criterion(output, torch.flatten(data, start_dim=1))
        loss.backward()
        optimizer.step()


# Generate new handwritten digits using the trained autoencoder
np.random.seed(42)
random_noise = np.random.normal(0,1,(30, hidden_size))
random_noise = torch.tensor(random_noise, dtype=torch.float32)
generated_images = autoencoder.decoder(random_noise).detach().cpu().numpy()

# Plot the generated images
fig, axes = plt.subplots(3, 10, figsize=(15, 4.5))  # Adjusted for 3 rows and 10 colum
for i, ax in enumerate(axes.flatten()):
    ax.imshow(generated_images[i].reshape(28, 28), cmap='gray')
    ax.axis('off')
plt.tight_layout()
plt.show()
```



RBN

In [ ]:
```
# Load the MNIST dataset
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])
train = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

```
trainloader = torch.utils.data.DataLoader(train, batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(test, batch_size=64, shuffle=False)
```

In [ ]:

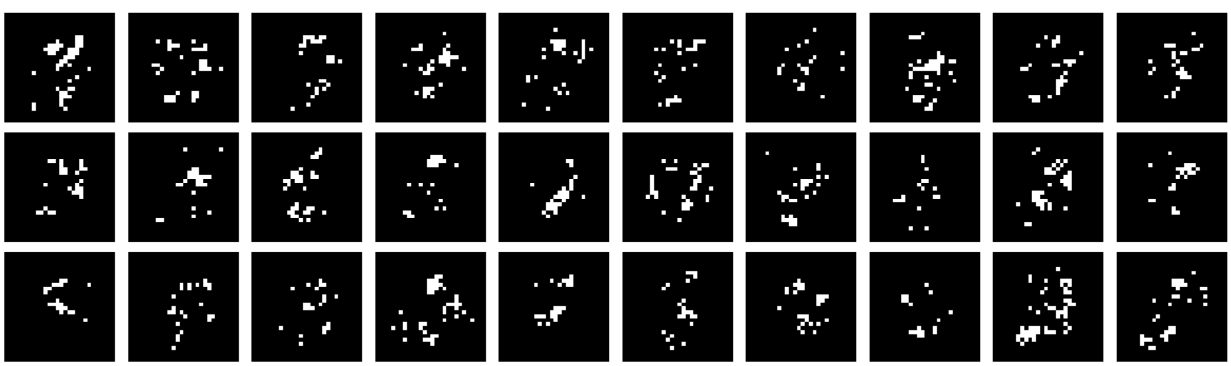
```
# Define the RBM class
class RBM(torch.nn.Module):
    def __init__(self, input, hiddensize):
        super(RBM, self).__init__()
        self.W = torch.nn.Parameter(torch.randn(hiddensize, input) * 0.01)
        self.v_bias = torch.nn.Parameter(torch.zeros(input))
        self.h_bias = torch.nn.Parameter(torch.zeros(hiddensize))

    def sample_h(self, v):
        h_prob = torch.sigmoid(torch.matmul(v, self.W.t()) + self.h_bias)
        h_sample = torch.bernoulli(h_prob)
        return h_sample

    def sample_v(self, h):
        v_prob = torch.sigmoid(torch.matmul(h, self.W) + self.v_bias)
        v_sample = torch.bernoulli(v_prob)
        return v_sample

    def free_energy(self, v):
        vbias_term = torch.matmul(v, self.v_bias)
        wx_b = torch.matmul(v, self.W.t()) + self.h_bias
        hidden_term = torch.sum(torch.log(1 + torch.exp(wx_b)), dim=1)
        return -vbias_term - hidden_term

    def forward(self, v):
        h = self.sample_h(v)
        v_recon = self.sample_v(h)
        return v_recon

# List of encoding dimensions to test
hidden_size = [50, 100, 150, 200]
results_dict = {}

for size in hidden_size:
    rbm = RBM(input=784, hiddensize=size)
    optimizer = torch.optim.SGD(rbm.parameters(), lr=0.01)

    # Training Loop for RBM
    for epoch in range(8):  # Adjust the number of epochs if necessary
        loss_sum = 0
        for data, _ in trainloader:
            data = data.view(-1, 784)  # Flatten the images
            data = torch.clamp(data, 0.0, 1.0)
            data = (data > 0.5).float()  # Convert to binary using thresholding
            v_recon = rbm(data)
            loss = torch.mean(rbm.free_energy(data)) - torch.mean(rbm.free_energy(v_re
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_sum += loss.item()
        average_loss = loss_sum / len(trainloader)
    results_dict[size] = {'reconstruction_loss': average_loss}

    # Extract features from the training set
    encoded_train = []
    train_labels = []
```

```
    for data, target in trainloader:
        data = data.view(-1, 784)
        data = torch.clamp(data, 0.0, 1.0)
        data = (data > 0.5).float()  # Convert to binary using thresholding
        encoded = rbm.sample_h(data)  # Get encoded (hidden) representations
        encoded_train.append(encoded.detach().numpy())
        train_labels.append(target.numpy())
    encoded_train = np.concatenate(encoded_train, axis=0)
    train_labels = np.concatenate(train_labels, axis=0)

    # Train a softmax classifier
    lr = LogisticRegression(max_iter=1000, solver='lbfgs', multi_class='multinomial')
    lr.fit(encoded_train, train_labels)

    # Evaluate classification accuracy on the test set
    encoded_test = []
    test_labels = []
    for data, target in testloader:
        data = data.view(-1, 784)
        data = torch.clamp(data, 0.0, 1.0)
        data = (data > 0.5).float()  # Convert to binary using thresholding
        encoded = rbm.sample_h(data)
        encoded_test.append(encoded.detach().numpy())
        test_labels.append(target.numpy())
    encoded_test = np.concatenate(encoded_test, axis=0)
    test_labels = np.concatenate(test_labels, axis=0)
    test_predictions = lr.predict(encoded_test)
    accuracy = accuracy_score(test_labels, test_predictions)

    # Record the classification accuracy
    results_dict[size]['classification_accuracy'] = accuracy
    print(f'Hidden Size: {size}, Reconstruction Loss: {average_loss}, Classification A

# Print the results for all encoding dimensions
print("Results for different Hidden Size:")
for size, metrics in results_dict.items():
    print(f'{size}: Reconstruction Loss: {metrics["reconstruction_loss"]}, Classificat
```

```
Encoding Dim: 50, Reconstruction Loss: -19.923210117608498, Classification Accuracy:
0.8439
Encoding Dim: 75, Reconstruction Loss: -18.55157110787658, Classification Accuracy:
0.8666
Encoding Dim: 100, Reconstruction Loss: -17.291857076860442, Classification Accuracy:
0.8758
Encoding Dim: 150, Reconstruction Loss: -15.531131760652132, Classification Accuracy:
0.8845
Encoding Dim: 200, Reconstruction Loss: -14.382853977715792, Classification Accuracy:
0.8889
Results for different Encoding Dimensions:
50: Reconstruction Loss: -19.923210117608498, Classification Accuracy: 0.8439
75: Reconstruction Loss: -18.55157110787658, Classification Accuracy: 0.8666
100: Reconstruction Loss: -17.291857076860442, Classification Accuracy: 0.8758
150: Reconstruction Loss: -15.531131760652132, Classification Accuracy: 0.8845
200: Reconstruction Loss: -14.382853977715792, Classification Accuracy: 0.8889
```

The best minimum hidden encoding dimmension was 150 because it had the 2nd highest classification and the 2nd lowest reconstruction loss but it is about the same as our max value which is 200. When trying to go for efficiency and the smallest size we can go, 150 is better than 200.

In [ ]:
```
# Set the number of hidden dimensions
hiddensize = 150

# Train an RBM with 150 hidden dimensions (assuming train_loader is defined)
rbm = RBM(n_vis=784, n_hid=hiddensize)
optimizer = torch.optim.SGD(rbm.parameters(), lr=0.1)

for epoch in range(8):  # Adjust the number of epochs if necessary
    for data, _ in trainloader:
        data = data.view(-1, 784)
        data = torch.clamp(data, 0.0, 1.0)
        data = (data > 0.5).float()  # Convert to binary using thresholding
        v_recon = rbm(data)
        loss = torch.mean(rbm.free_energy(data)) - torch.mean(rbm.free_energy(v_recon.
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Generate 30 new images using the trained RBM
np.random.seed(42)  # Set a seed for reproducibility
random_hidden = torch.bernoulli(torch.rand(30, hiddensize))  # Randomly sample the hic
generated_images = rbm.sample_v(random_hidden).detach().numpy()  # Reconstruct the vis

# Plot the generated images
fig, axes = plt.subplots(3, 10, figsize=(15, 4.5))  # Adjusted for 3 rows and 10 colum
for i, ax in enumerate(axes.flatten()):
    ax.imshow(generated_images[i].reshape(28, 28), cmap='gray')
    ax.axis('off')
plt.tight_layout()
plt.show()
```



DBN

In [1]:
```
!pip install keras-tuner
```

```
Collecting keras-tuner
  Downloading keras_tuner-1.4.7-py3-none-any.whl (129 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 129.1/129.1 kB 1.5 MB/s eta 0:00:00
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from
keras-tuner) (2.15.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages
(from keras-tuner) (24.0)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (f
rom keras-tuner) (2.31.0)
Collecting kt-legacy (from keras-tuner)
  Downloading kt_legacy-1.0.5-py3-none-any.whl (9.6 kB)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/
dist-packages (from requests->keras-tuner) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-package
s (from requests->keras-tuner) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-p
ackages (from requests->keras-tuner) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-p
ackages (from requests->keras-tuner) (2024.2.2)
Installing collected packages: kt-legacy, keras-tuner
Successfully installed keras-tuner-1.4.7 kt-legacy-1.0.5
```

In [4]:

```python
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from kerastuner.tuners import RandomSearch

def load_data():
    # Load and preprocess the MNIST dataset
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = x_train.reshape(-1, 784).astype('float32') / 255
    x_test = x_test.reshape(-1, 784).astype('float32') / 255
    y_train = to_categorical(y_train, 10)
    y_test = to_categorical(y_test, 10)
    return x_train, y_train, x_test, y_test

def build_model(hp):
    model = Sequential()
    # Hyperparameters to tune
    units = hp.Choice('units', values=[50, 100, 150, 200])
    dropout_rate = hp.Float('dropout_rate', min_value=0.0, max_value=0.5, step=0.1)

    # Model architecture
    model.add(Dense(units=units, activation='relu', input_shape=(784,)))
    model.add(Dropout(dropout_rate))
    model.add(Dense(units=units, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(Dense(10, activation='softmax'))

    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

x_train, y_train, x_test, y_test = load_data()
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
```

```
    max_trials=10,  # Adjust the number of trials to cover the grid
    executions_per_trial=1,
    directory='tuner_dir',
    project_name='DBN_grid_search')

tuner.search(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# Get the optimal hyperparameters
best_hps = tuner.get_best_hyperparameters()[0]
print(f"Best number of units: {best_hps.get('units')}")
print(f"Best dropout rate: {best_hps.get('dropout_rate')}")

# Build the model with the best hyperparameters and train it
model = tuner.hypermodel.build(best_hps)
model.fit(x_train, y_train, epochs=50, validation_data=(x_test, y_test))
```

```
Reloading Tuner from tuner_dir/DBN_grid_search/tuner0.json
Best number of units: 200
Best dropout rate: 0.30000000000000004
Epoch 1/50
1875/1875 [==============================] - 18s 9ms/step - loss: 0.2960 - accuracy: 0.9098 - val_loss: 0.1286 - val_accuracy: 0.9607
Epoch 2/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.1493 - accuracy: 0.9553 - val_loss: 0.0959 - val_accuracy: 0.9679
Epoch 3/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.1188 - accuracy: 0.9638 - val_loss: 0.0737 - val_accuracy: 0.9767
Epoch 4/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1014 - accuracy: 0.9685 - val_loss: 0.0719 - val_accuracy: 0.9768
Epoch 5/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0892 - accuracy: 0.9724 - val_loss: 0.0780 - val_accuracy: 0.9769
Epoch 6/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0801 - accuracy: 0.9753 - val_loss: 0.0732 - val_accuracy: 0.9785
Epoch 7/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0770 - accuracy: 0.9761 - val_loss: 0.0739 - val_accuracy: 0.9794
Epoch 8/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0696 - accuracy: 0.9781 - val_loss: 0.0733 - val_accuracy: 0.9807
Epoch 9/50
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0677 - accuracy: 0.9782 - val_loss: 0.0692 - val_accuracy: 0.9805
Epoch 10/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0641 - accuracy: 0.9798 - val_loss: 0.0660 - val_accuracy: 0.9821
Epoch 11/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0580 - accuracy: 0.9814 - val_loss: 0.0671 - val_accuracy: 0.9809
Epoch 12/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0608 - accuracy: 0.9808 - val_loss: 0.0707 - val_accuracy: 0.9830
Epoch 13/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0526 - accuracy: 0.9833 - val_loss: 0.0670 - val_accuracy: 0.9824
Epoch 14/50
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0550 - accuracy: 0.9822 - val_loss: 0.0670 - val_accuracy: 0.9816
Epoch 15/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0534 - accuracy: 0.9830 - val_loss: 0.0650 - val_accuracy: 0.9835
Epoch 16/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0477 - accuracy: 0.9849 - val_loss: 0.0726 - val_accuracy: 0.9813
Epoch 17/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0452 - accuracy: 0.9855 - val_loss: 0.0767 - val_accuracy: 0.9810
Epoch 18/50
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0471 - accuracy: 0.9854 - val_loss: 0.0741 - val_accuracy: 0.9817
Epoch 19/50
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0499 - accuracy: 0.9845 - val_loss: 0.0693 - val_accuracy: 0.9822
```

```
Epoch 20/50
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0446 - accuracy:
0.9857 - val_loss: 0.0722 - val_accuracy: 0.9810
Epoch 21/50
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0455 - accuracy:
0.9859 - val_loss: 0.0764 - val_accuracy: 0.9801
Epoch 22/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0432 - accuracy:
0.9865 - val_loss: 0.0738 - val_accuracy: 0.9839
Epoch 23/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0422 - accuracy:
0.9872 - val_loss: 0.0806 - val_accuracy: 0.9798
Epoch 24/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0432 - accuracy:
0.9864 - val_loss: 0.0745 - val_accuracy: 0.9828
Epoch 25/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0429 - accuracy:
0.9868 - val_loss: 0.0769 - val_accuracy: 0.9819
Epoch 26/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0423 - accuracy:
0.9874 - val_loss: 0.0683 - val_accuracy: 0.9848
Epoch 27/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0395 - accuracy:
0.9877 - val_loss: 0.0738 - val_accuracy: 0.9840
Epoch 28/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0398 - accuracy:
0.9871 - val_loss: 0.0723 - val_accuracy: 0.9833
Epoch 29/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0393 - accuracy:
0.9881 - val_loss: 0.0695 - val_accuracy: 0.9825
Epoch 30/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0371 - accuracy:
0.9882 - val_loss: 0.0771 - val_accuracy: 0.9832
Epoch 31/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0334 - accuracy:
0.9897 - val_loss: 0.0795 - val_accuracy: 0.9838
Epoch 32/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0385 - accuracy:
0.9883 - val_loss: 0.0783 - val_accuracy: 0.9817
Epoch 33/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0362 - accuracy:
0.9886 - val_loss: 0.0723 - val_accuracy: 0.9821
Epoch 34/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0352 - accuracy:
0.9893 - val_loss: 0.0778 - val_accuracy: 0.9830
Epoch 35/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0351 - accuracy:
0.9890 - val_loss: 0.0877 - val_accuracy: 0.9826
Epoch 36/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0341 - accuracy:
0.9895 - val_loss: 0.0854 - val_accuracy: 0.9818
Epoch 37/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0339 - accuracy:
0.9897 - val_loss: 0.0906 - val_accuracy: 0.9824
Epoch 38/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0355 - accuracy:
0.9893 - val_loss: 0.0835 - val_accuracy: 0.9813
Epoch 39/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0360 - accuracy:
0.9893 - val_loss: 0.0747 - val_accuracy: 0.9854
```

```
Epoch 40/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0339 - accuracy:
0.9904 - val_loss: 0.0849 - val_accuracy: 0.9825
Epoch 41/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0345 - accuracy:
0.9900 - val_loss: 0.0775 - val_accuracy: 0.9834
Epoch 42/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0360 - accuracy:
0.9887 - val_loss: 0.0903 - val_accuracy: 0.9840
Epoch 43/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0292 - accuracy:
0.9911 - val_loss: 0.1028 - val_accuracy: 0.9832
Epoch 44/50
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0336 - accuracy:
0.9902 - val_loss: 0.0864 - val_accuracy: 0.9832
Epoch 45/50
1875/1875 [==============================] - 13s 7ms/step - loss: 0.0312 - accuracy:
0.9907 - val_loss: 0.0897 - val_accuracy: 0.9827
Epoch 46/50
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0318 - accuracy:
0.9905 - val_loss: 0.0930 - val_accuracy: 0.9827
Epoch 47/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0325 - accuracy:
0.9906 - val_loss: 0.0858 - val_accuracy: 0.9834
Epoch 48/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0298 - accuracy:
0.9907 - val_loss: 0.0991 - val_accuracy: 0.9829
Epoch 49/50
1875/1875 [==============================] - 11s 6ms/step - loss: 0.0311 - accuracy:
0.9912 - val_loss: 0.0994 - val_accuracy: 0.9821
Epoch 50/50
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0332 - accuracy:
0.9903 - val_loss: 0.1109 - val_accuracy: 0.9808
```

Out[4]:

```
<keras.src.callbacks.History at 0x79e708e1e3b0>
```

In [17]:

```python
# Get all completed trials, sorted by their performance
sorted_trials = sorted(tuner.oracle.get_best_trials(num_trials=10), key=lambda x: x.sc

# Loop through the top 10 trials to display hyperparameters and corresponding metrics
for i, trial in enumerate(sorted_trials):
    print(f"Top {i+1} Trial:")
    print("Hyperparameters:", trial.hyperparameters.values)
    print("Objective (e.g., Loss):", trial.score)  # Score refers to the objective you

    # Accessing specific metrics like 'loss' and 'accuracy'
    # Make sure these names match how they were recorded during the search
    try:
        val_loss = trial.metrics.get_best_value('val_loss')
        val_accuracy = trial.metrics.get_best_value('val_accuracy')
        print(f"Best Validation Loss: {val_loss}")
        print(f"Best Validation Accuracy: {val_accuracy}")
    except KeyError as e:
        print(f"Error retrieving metric: {e}")

    print("\n")
```

```
Top 1 Trial:
Hyperparameters: {'units': 50, 'dropout_rate': 0.4}
Objective (e.g., Loss): 0.9639000296592712
Best Validation Loss: 0.12775953114032745
Best Validation Accuracy: 0.9639000296592712


Top 2 Trial:
Hyperparameters: {'units': 50, 'dropout_rate': 0.30000000000000004}
Objective (e.g., Loss): 0.9678999781608582
Best Validation Loss: 0.10606943070888519
Best Validation Accuracy: 0.9678999781608582


Top 3 Trial:
Hyperparameters: {'units': 50, 'dropout_rate': 0.2}
Objective (e.g., Loss): 0.97079998254776
Best Validation Loss: 0.09899251908063889
Best Validation Accuracy: 0.97079998254776


Top 4 Trial:
Hyperparameters: {'units': 100, 'dropout_rate': 0.4}
Objective (e.g., Loss): 0.9760000109672546
Best Validation Loss: 0.08561421930789948
Best Validation Accuracy: 0.9760000109672546


Top 5 Trial:
Hyperparameters: {'units': 100, 'dropout_rate': 0.30000000000000004}
Objective (e.g., Loss): 0.9763000011444092
Best Validation Loss: 0.08270733803510666
Best Validation Accuracy: 0.9763000011444092


Top 6 Trial:
Hyperparameters: {'units': 100, 'dropout_rate': 0.2}
Objective (e.g., Loss): 0.9778000116348267
Best Validation Loss: 0.07927803695201874
Best Validation Accuracy: 0.9778000116348267


Top 7 Trial:
Hyperparameters: {'units': 150, 'dropout_rate': 0.4}
Objective (e.g., Loss): 0.9781000018119812
Best Validation Loss: 0.07377926260232925
Best Validation Accuracy: 0.9781000018119812


Top 8 Trial:
Hyperparameters: {'units': 200, 'dropout_rate': 0.4}
Objective (e.g., Loss): 0.9786999821662903
Best Validation Loss: 0.07289712131023407
Best Validation Accuracy: 0.9786999821662903


Top 9 Trial:
Hyperparameters: {'units': 150, 'dropout_rate': 0.30000000000000004}
Objective (e.g., Loss): 0.9787999987602234
Best Validation Loss: 0.07296411693096161
```

```
Best Validation Accuracy: 0.9787999987602234


Top 10 Trial:
Hyperparameters: {'units': 200, 'dropout_rate': 0.30000000000000004}
Objective (e.g., Loss): 0.9815000295639038
Best Validation Loss: 0.0684111937880516
Best Validation Accuracy: 0.9815000295639038
```

After completing the grid search the best value for our hidden size is 200 has it gave us the lowest loss and highest accuracy.