Homework 7

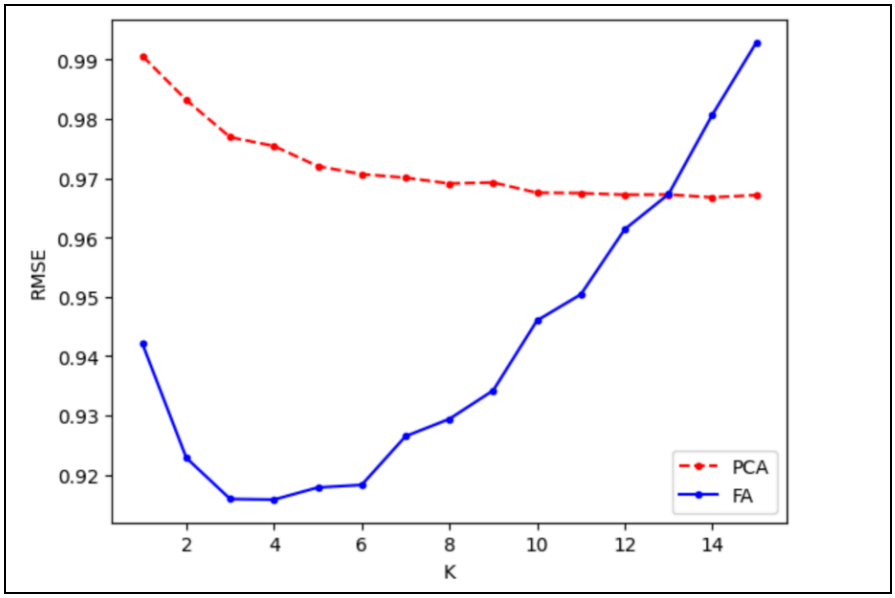
SHREYA CHETAN PAWASKAR
pawaskas@uci.edu
12041645

Question 1:

Code:

```
import numpy as np
import random
import matplotlib.pyplot as plt
from movie_utils import calc_rmse_for_movie_ratings
from sparse_fa_em import sparse_fa_em

use_multiprocessing = False

data = np.load("movielens500.npy", allow_pickle=True).item()
ratings_train = data["S_train"]
ratings_test = data["S_test"]
ratings_train = ratings_train.astype("float")
ratings_test = ratings_test.astype("float")

num_movies = ratings_train.shape[0]
num_users = ratings_train.shape[1]

ratings_filled_with_mean = np.copy(ratings_train)
for movie_idx in range(num_movies):
   missing_indices = ratings_train[movie_idx, :] == 0
   num_non_missing = np.sum(ratings_train[movie_idx, :] != 0).astype("float")
   sum_ratings = np.sum(ratings_train[movie_idx, :])
   ratings_filled_with_mean[movie_idx, missing_indices] = sum_ratings /
num_non_missing

rmse_fill_with_mean = calc_rmse_for_movie_ratings(ratings_test,
ratings_filled_with_mean)
print("Test RMSE", rmse_fill_with_mean_rating)

num_components = np.arange(1, 16)
```

```
pca_errors = np.zeros((num_components.shape[0],))
W, Z = pca(ratings_filled_with_mean.T)

for idx, K in enumerate(num_components):
   print(f"PCA Dimension: {K:3d}", end=" ")
   reconstructed_ratings_pca = Z[:, 0:K] @ W[:, 0:K].T
   reconstructed_ratings_pca += np.mean(ratings_filled_with_mean.T, 0)
   pca_errors[idx] = calc_rmse_for_movie_ratings(ratings_test.T,
reconstructed_ratings_pca)
   print(f"RMSE: {pca_errors[idx]:.3f}")

max_iters = 100
fa_errors = np.zeros((num_components.shape[0],))

def do_fa_for_k_value(K):
   Z, W, mu, psi, * = sparse_fa_em(ratings_train, K, max_iters)
   reconstructed_ratings_fa = Z @ W.T
   reconstructed_ratings_fa = reconstructed_ratings_fa + mu.T
   return calc_rmse_for_movie_ratings(ratings_test.T, reconstructed_ratings_fa)

if use_multiprocessing:
   from multiprocessing import Pool
   with Pool(5) as p:
       output = p.map(do_fa_for_k_value, list(num_components))
   for i in range(len(output)):
       fa_errors[i] = output[i]

   for idx, K in enumerate(num_components):
       print(f"FA Dimension: {K:3d} RMSE: {fa_errors[idx]:.3f}")
else:
   for idx, K in enumerate(num_components):
       print(f"FA Dimension: {K:3d}", end=" ")
       fa_errors[idx] = do_fa_for_k_value(K)
       print(f"RMSE: {fa_errors[idx]:.3f}")

plt.plot(num_components, pca_errors, "r.--", label="PCA")
plt.plot(num_components, fa_errors, label="FA")
plt.xlabel("K")
plt.ylabel("RMSE")
plt.legend()
plt.show()
```



a)

The test RMSE = **1.0117**.

---

b)

Output:

Dimension	PCA RMSE	FA RMSE
1	0.991	0.942
2	0.983	0.923
3	0.977	0.916
4	0.975	0.915
5	0.972	0.918
6	0.971	0.918
7	0.97	0.926
8	0.969	0.932
9	0.969	0.935
10	0.968	0.942
11	0.967	0.947
12	0.967	0.958
13	0.967	0.965
14	0.967	0.981
15	0.967	0.993

- A 11-dimensional model yields the best performance, with a RMSE of approximately 0.967.
- Increasing the number of dimensions beyond 11 does not improve the results further.
- In some cases, it even degrades the performance.

- The optimal 11-dimensional model outperforms the baseline method, as evidenced by its lower RMSE, which represents a smaller squared error.

---

c)

- When you increase K in PCA, it can recreate the original data matrix more accurately.
- If K =M, PCA can perfectly reconstruct the filled-in ratings matrix.
- So here, the performance of PCA will be the same as simply using the mean rating for each movie, which was the baseline approach.

---

d)

- FA performs much better than PCA.
- For larger K, test performance for factor analysis rapidly deteriorates, indicating overfitting.
- This overfitting can be reduced by placing prior distributions on the model parameters and doing MAP estimation.
- A prior which discourages estimated variances from becoming overly small is esp. useful.

Question 2:

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats
from tqdm import tqdm
from vae_utils import *
from ppca import *
import torch
import torch.optim as optim
import torch.nn as nn

do_train_on_gpu = True
```

```
# Function to train a VAE model
def train_vae(vae_model, dataset, batch_size=128, num_epochs=50, train_on_gpu=False):
  device = "cuda" if train_on_gpu else "cpu"
  train_loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=batch_size,
shuffle=True, num_workers=6)
  vae_model = vae_model.to(device)
  optimizer = optim.Adam(vae_model.parameters(), lr=0.001)
  training_losses = []
  for epoch in tqdm(range(num_epochs)):
      epoch_losses = []
      t = tqdm(iter(train_loader), leave=False, total=len(train_loader))
      for step, data in enumerate(t):
          imgs, _ = data
          imgs = torch.reshape(imgs, (imgs.shape[0], -1))
          imgs = imgs.to(device)
          vae_model.zero_grad()
          recon_imgs, mu, sigma = vae_model(imgs)
          loss = elbo_loss_function(recon_imgs, imgs, mu, sigma)
          epoch_losses.append(loss.detach().item())
          loss.backward()
          optimizer.step()
      avg_epoch_loss = sum(epoch_losses) / len(epoch_losses)
      training_losses.append(avg_epoch_loss)
  return vae_model, np.asarray(training_losses)

# Function to create reconstructions of images using a model
```

```
def create_reconstructions(model, dataset, title, num_images=7,
num_reconstructions=1):
  loader = torch.utils.data.DataLoader(dataset=dataset, batch_size=num_images,
shuffle=False, num_workers=6)
  orig_imgs, _ = next(iter(loader))
  flat_orig_imgs = torch.reshape(orig_imgs, (-1, 28 * 28))
  if isinstance(model, nn.Module):
      model_internal = model.cpu()
      with torch.no_grad():
          model_internal.eval()
          all_recon_imgs = []
          for _ in range(num_reconstructions):
              recon_imgs, _, _ = model_internal(flat_orig_imgs)
              recon_imgs = torch.reshape(recon_imgs, (-1, 28, 28))
              all_recon_imgs.append(recon_imgs.numpy())
  else:
      flat_orig_imgs = flat_orig_imgs.numpy()
      all_recon_imgs = []
      for _ in range(num_reconstructions):
          recon_imgs = model.recover(flat_orig_imgs)
          recon_imgs = np.reshape(recon_imgs, (-1, 28, 28))
          all_recon_imgs.append(recon_imgs)
  fig = plt.figure(constrained_layout=True)
  rows = num_reconstructions + 1
  cols = num_images
  subfigs = fig.subfigures(nrows=rows, ncols=1, wspace=0.07)
  axes = []
  for i in range(rows):
      if i == 0:
          subfigs[i].suptitle("Original")
      elif i == 1:
          subfigs[i].suptitle("Reconstructions")
      axs = subfigs[i].subplots(nrows=1, ncols=cols)
      axes.append(axs)
  for img_idx in range(num_images):
      orig_img = orig_imgs[img_idx]
      axes[0][img_idx].imshow(orig_img.numpy(), cmap="Greys")
      axes[0][img_idx].axis("off")
      for r_idx in range(num_reconstructions):
          recon_img = all_recon_imgs[r_idx][img_idx]
          axes[r_idx + 1][img_idx].imshow(recon_img, cmap="Greys")
```

```
            axes[r_idx + 1][img_idx].axis("off")
    fig.suptitle(title)
    return fig

# Function to sample images from a model
def sample_from(model, title, num_images=25):
    assert num_images >= 4
    if isinstance(model, nn.Module):
        model_internal = model.cpu()
        with torch.no_grad():
            model_internal.eval()
            z = torch.randn((num_images, model_internal.latent_dim))
            imgs = model_internal.decoder(z)
            imgs = torch.sigmoid(imgs)
            imgs = torch.reshape(imgs, (-1, 28, 28))
            imgs = imgs.numpy()
    else:
        z = np.random.randn(num_images, model.latent_dim)
        imgs = model.decode(z)
        imgs = np.reshape(imgs, (-1, 28, 28))
    rows = int(np.sqrt(num_images))
    cols = int(np.sqrt(num_images))
    if (rows * cols) != num_images:
        rows += 1
    fig, axes = plt.subplots(rows, cols)
    counter = 0
    for r in range(rows):
        if counter >= imgs.shape[0]:
            break
        for c in range(cols):
            if counter >= imgs.shape[0]:
                break
            axes[r, c].imshow(imgs[counter], cmap="Greys")
            axes[r, c].axis("off")
            counter += 1
    fig.suptitle(title)
    fig.tight_layout()
    return fig

# Load the dataset
train_dataset = BinaryMNIST()
```

```
# Load all images and labels from the dataset
loader = torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=len(train_dataset), shuffle=True, num_workers=6)
all_imgs, all_labels = next(iter(loader))
all_imgs = torch.reshape(all_imgs, (all_imgs.shape[0], -1))
all_imgs = all_imgs.numpy()
```

```
#PPCA
# Fit a PPCA model with 2-dimensional latent space
print("Fitting PPCA model with 2-dimensional latent space...")
ppca_2d = PPCA(latent_dim=2, verbose=False)
ppca_2d.fit(all_imgs)

# Generate plots for the 2D PPCA model
fig_2d_clustering_ppca = plot_2d_clusterings(ppca_2d, train_dataset, "PPCA 2D Latent
Space Clustering")
fig_2d_reconstructions_ppca = create_reconstructions(ppca_2d, train_dataset, "PPCA 2D
Latent Space Reconstructions")
fig_2d_samples_ppca = sample_from(ppca_2d, "PPCA 2D Latent Space Samples")
```

```
#PPCA
# Fit a PPCA model with 50-dimensional latent space
print("Fitting PPCA model with 50-dimensional latent space...")
ppca_50d = PPCA(latent_dim=50, verbose=False)
ppca_50d.fit(all_imgs)

# Generate plots for the 50D PPCA model
fig_50d_reconstructions_ppca = create_reconstructions(ppca_50d, train_dataset, "PPCA
50D Latent Space Reconstructions")
fig_50d_samples_ppca = sample_from(ppca_50d, "PPCA 50D Latent Space Samples")
```

```
#VAEE
# VAE with 2-dimensional latent space
print("Creating and training VAE model with 2-dimensional latent space...")
vae_2d = VAE(latent_dim=2)
vae_2d, losses_2d = train_vae(vae_2d, train_dataset, train_on_gpu=do_train_on_gpu)

#Training losses for 2D VAE
fig_losses_2d = plt.figure()
plt.plot(losses_2d)
plt.xlabel("Epoch")
```

```
plt.ylabel("ELBO Loss")
plt.title("Training Losses for VAE with 2D Latent Space")

#Pplots for the 2D VAE
fig_clustering_2d = plot_2d_clusterings(vae_2d, train_dataset, "VAE 2D Latent Space Clustering")
fig_reconstructions_2d = create_reconstructions(vae_2d, train_dataset, "VAE 2D Latent Space Reconstructions")
fig_samples_2d = sample_from(vae_2d, "VAE 2D Latent Space Samples")

# VAE with 50-dimensional latent space
print("Creating and training VAE model with 50-dimensional latent space...")
vae_50d = VAE(latent_dim=50)
vae_50d, losses_50d = train_vae(vae_50d, train_dataset, train_on_gpu=do_train_on_gpu)

#Training losses for 50D VAE
fig_losses_50d = plt.figure()
plt.plot(losses_50d)
plt.xlabel("Epoch")
plt.ylabel("ELBO Loss")
plt.title("Training Losses for VAE with 50D Latent Space")

#Plots for 50D VAE
fig_reconstructions_50d = create_reconstructions(vae_50d, train_dataset, "VAE 50D Latent Space Reconstructions")
fig_samples_50d = sample_from(vae_50d, "VAE 50D Latent Space Samples")

plt.show()
```

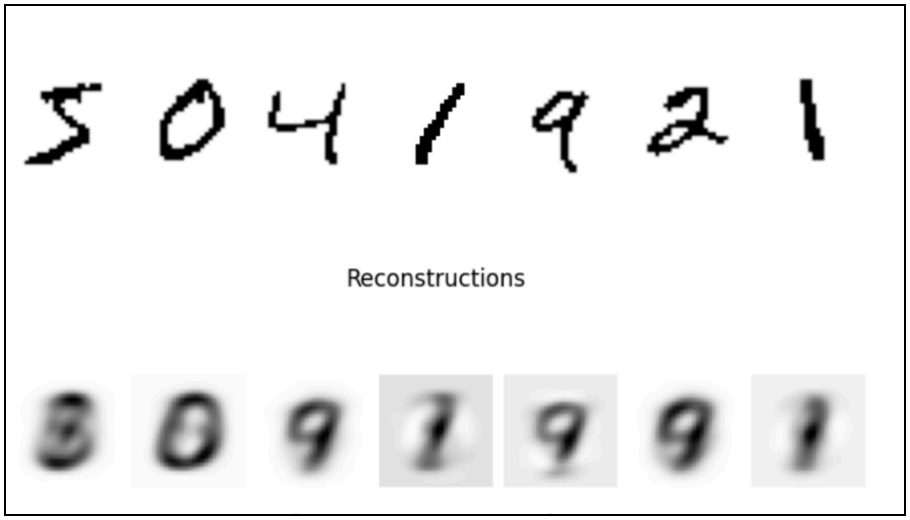
A) This code snippet gives output for PPCA:

```
#PPCA
# Fit a PPCA model with 2-dimensional latent space
print("Fitting PPCA model with 2-dimensional latent space...")
ppca_2d = PPCA(latent_dim=2, verbose=False)
ppca_2d.fit(all_imgs)

# Generate plots for the 2D PPCA model
fig_2d_clustering_ppca = plot_2d_clusterings(ppca_2d, train_dataset, "PPCA 2D Latent Space Clustering")
```

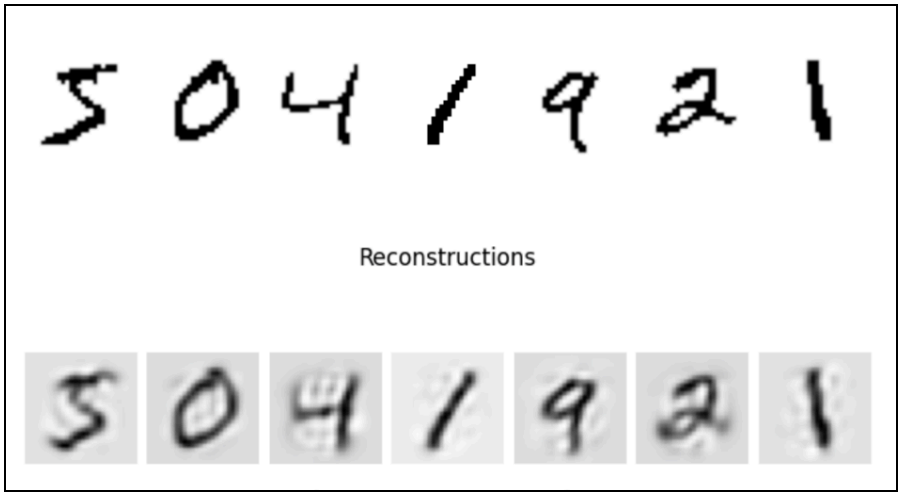
```
fig_2d_reconstructions_ppca = create_reconstructions(ppca_2d, train_dataset, "PPCA 2D Latent Space Reconstructions")
fig_2d_samples_ppca = sample_from(ppca_2d, "PPCA 2D Latent Space Samples")

# Fit a PPCA model with 50-dimensional latent space
print("Fitting PPCA model with 50-dimensional latent space...")
ppca_50d = PPCA(latent_dim=50, verbose=False)
ppca_50d.fit(all_imgs)

# Generate plots for the 50D PPCA model
fig_50d_reconstructions_ppca = create_reconstructions(ppca_50d, train_dataset, "PPCA 50D Latent Space Reconstructions")
fig_50d_samples_ppca = sample_from(ppca_50d, "PPCA 50D Latent Space Samples")
```

B)

PPCA 2:

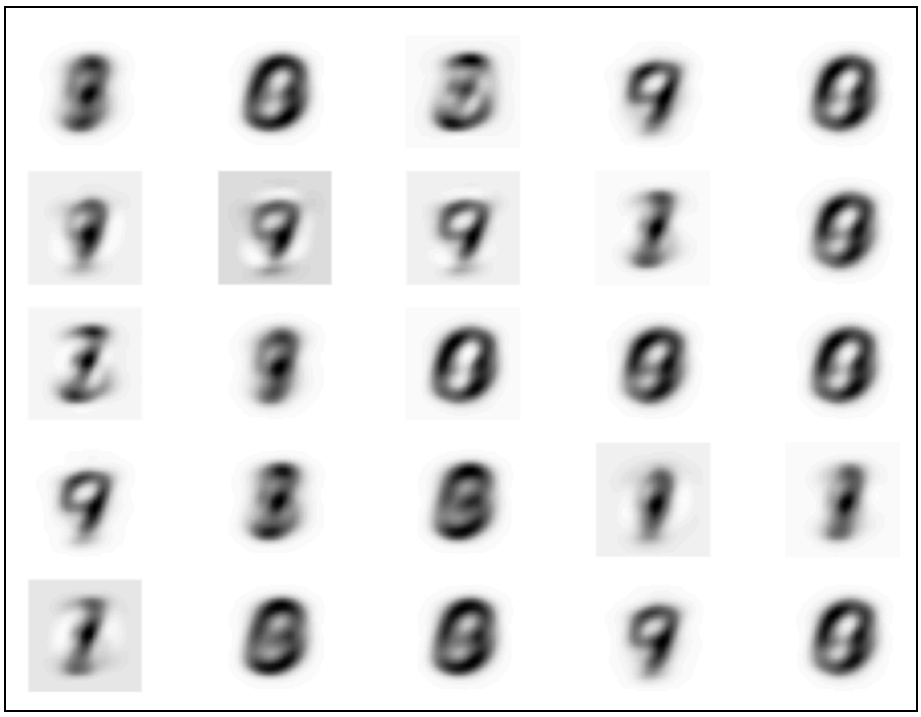


PPCA 50:



C)

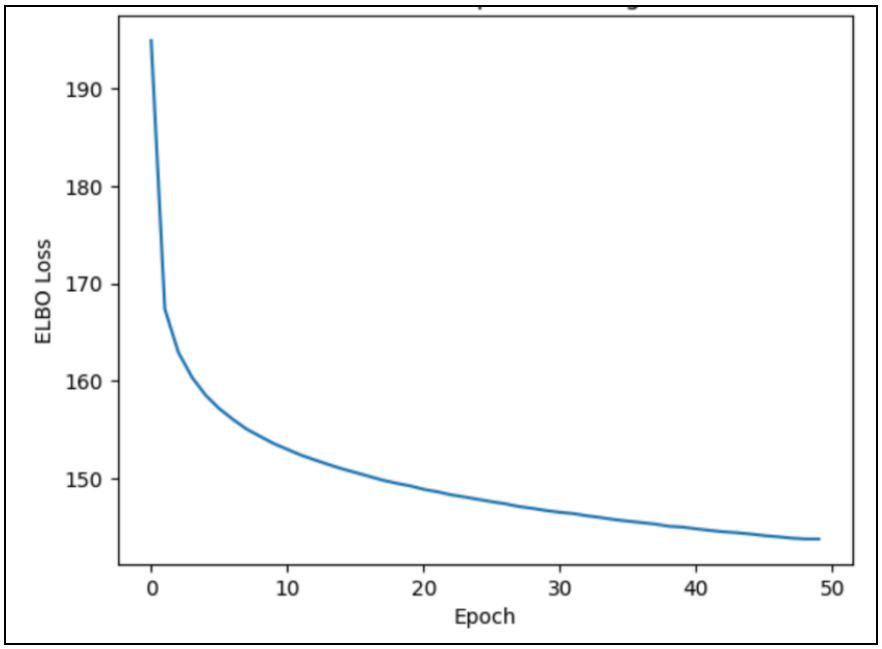
PPCA 2:



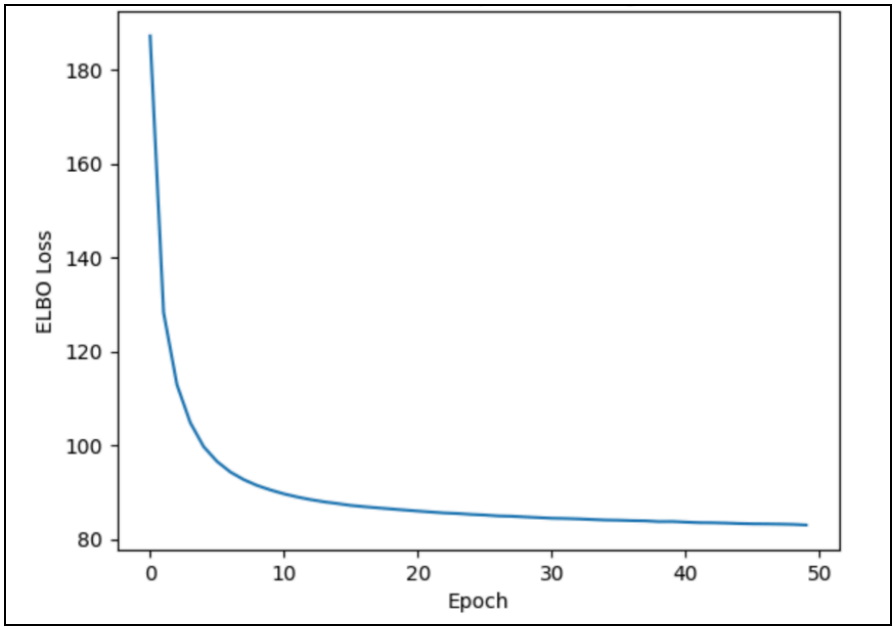
PPCA 50:

D)

VAE: k=2

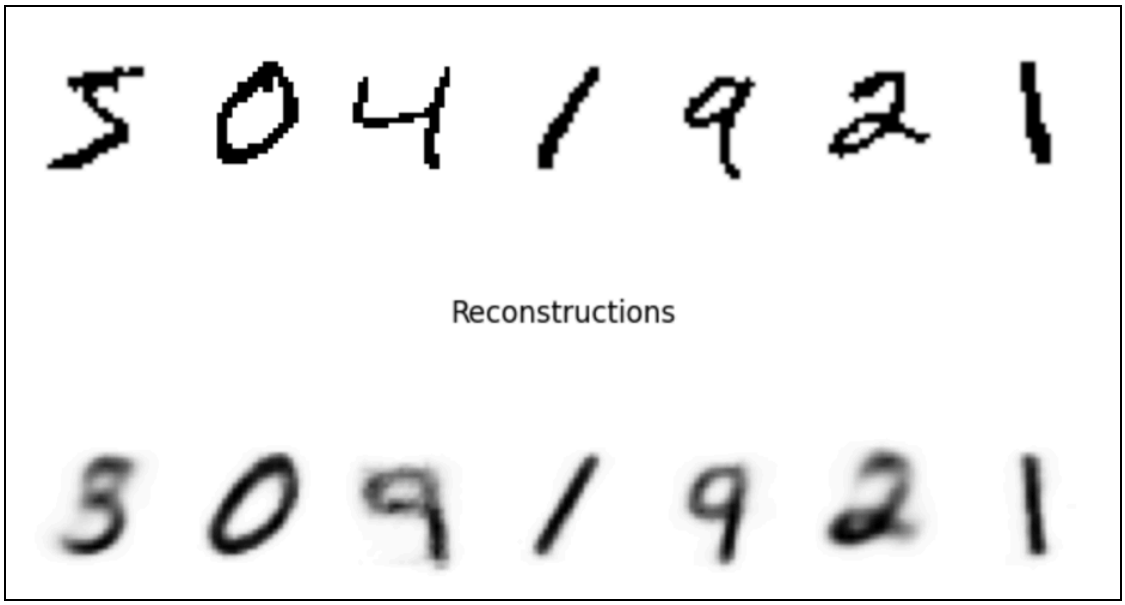


VAE: k=50

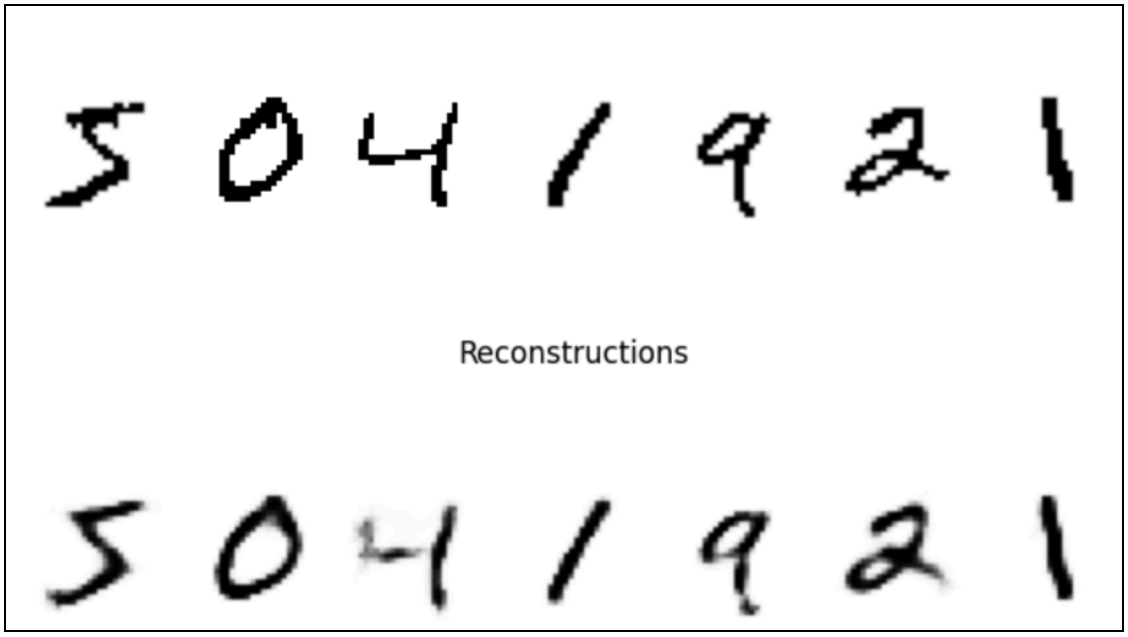


E)

VAE: k=2



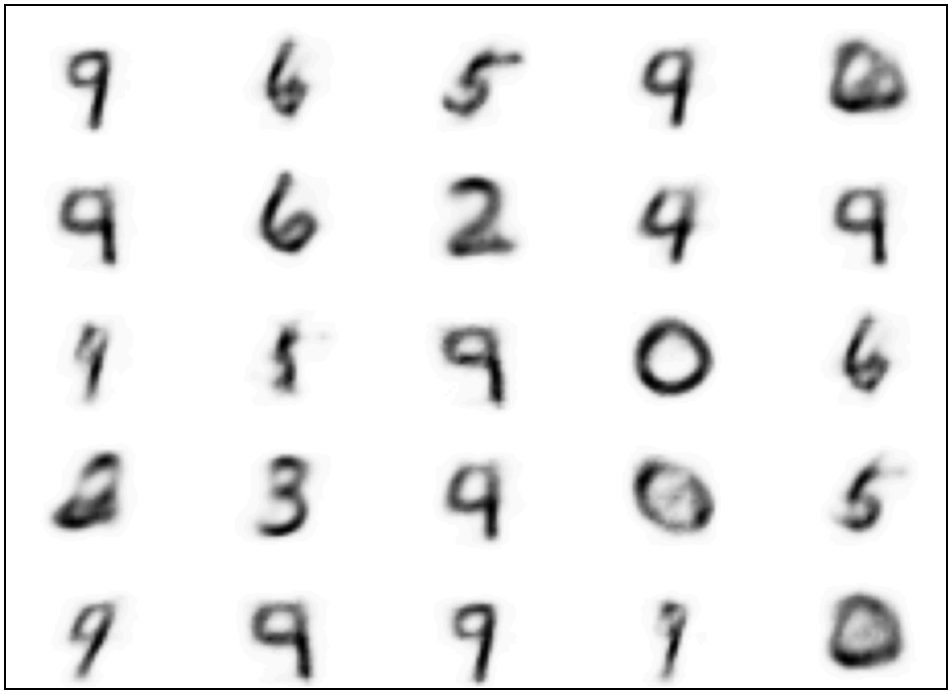
VAE: k=50



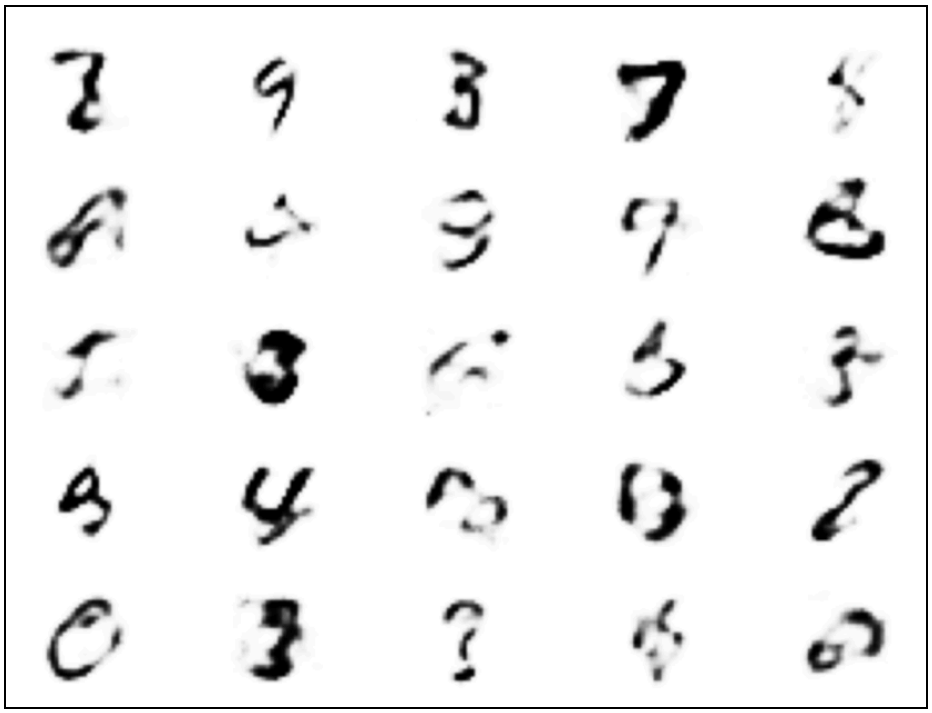
- VAE with K=2 reconstructs images as good as or better than PPCA with K=50.
- VAE with K=50 can almost perfectly reconstruct images.
- PPCA with K=50 has a slight "haze" on non-zero background during reconstructions.
- VAE models outperform PPCA models in reconstruction quality.
- Higher-dimensional VAE (K=50) achieves near-perfect reconstructions.

F)

VAE: k=2



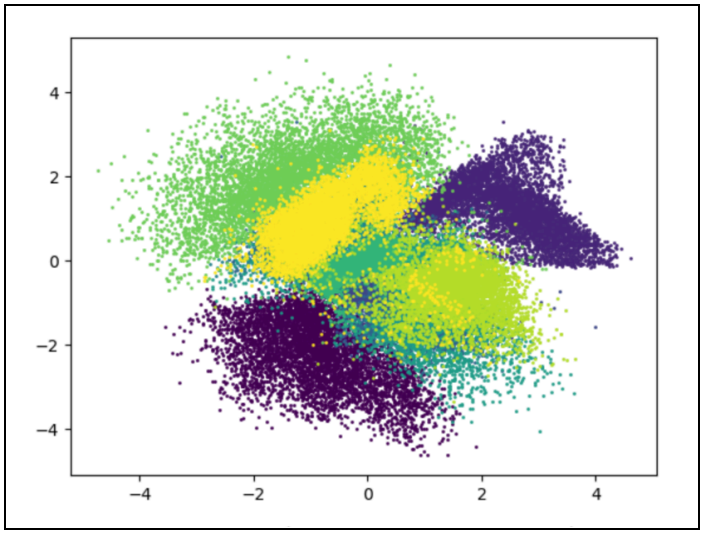
VAE: k=50



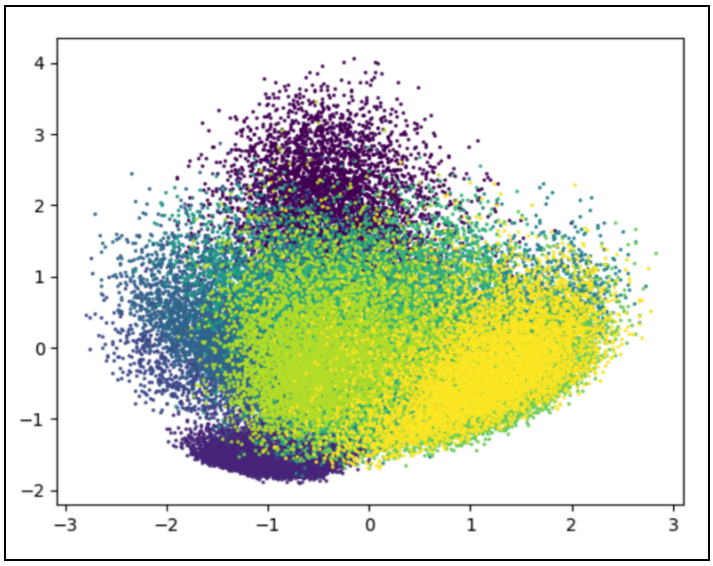
- VAE models look similar to the original than PPCA models.
- VAE samples look like real MNIST digits, even with low-dimensional latent space (K=2).
- PPCA samples do not resemble real digits, regardless of latent space dimension.
- VAEs capture data distribution better, generating more realistic samples.

---

g)
VAE: k=2



PPCA 2:



- The PPCA model exhibits significant overlap between images with different digit labels.
- The VAE model separates MNIST digits into more distinct clusters. It demonstrates clear grouping of images representing the same digit.
- It achieves better clustering of MNIST digits compared to PPCA.