Generative neural networks for the sciences
Exercise 02 solution

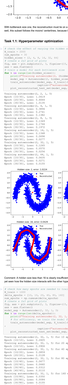
```
In [1]: %matplotlib inline
        %load_ext autoreload
        %autoreload 2
        import numpy as np
        import matplotlib as mp
        mp.rc('image', cmap='jet')

        from sklearn.datasets import make_moons
        from sklearn.metrics import roc_curve
        from sklearn.ensemble import GradientBoostingClassifier
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from sklearn.metrics import roc_auc_score
        from sklearn.metrics import confusion_matrix
        from sklearn.metrics import log_loss
        from sklearn.metrics import accuracy_score

        import numpy
        import numpy.random as npr
        import numpy.linalg as npla
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

        import sklearn.decomposition ipca1 ipca2
```

Create two moons data for tasks 1 and 2:

```
In [2]: # 2d-2 moons (source_dimensionality, noisiness)
        X_2d_2moons, y_2d_2moons = make_moons(10**5, noise=0.1, random_state=1)

        # 10-dim 2 moons (source_dimensionality, noisiness)
        # the first two dimensions are the same as the 2d-2 moons, the rest is random_state=2
        X_10d_2moons, y_10d_2moons = make_moons(10**5, noise=0.1, random_state=2)
        X_10d_2moons = X_10d_2moons[:,2:]

        # scale the coordinates
        X_2d_2moons = StandardScaler().fit_transform(X_2d_2moons)
        X_10d_2moons = StandardScaler().fit_transform(X_10d_2moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_2moons[:,0], X_2d_2moons[:,1], lw)
```

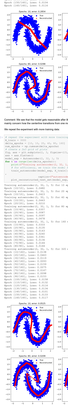
Create one more data for tasks 1 and 2:

```
In [3]: # 2d-5 moons (source_dimensionality, noisiness)
        X_2d_5moons, y_2d_5moons = make_moons(10**5, noise=0.1, random_state=3)

        # scale the coordinates
        X_2d_5moons = StandardScaler().fit_transform(X_2d_5moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_5moons[:,0], X_2d_5moons[:,1], lw)
```



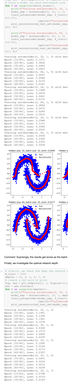
Task 1: Autoencoder with bottleneck size 1

```
In [4]: # keras backend stuff
        import keras, keras.models, keras.layers, keras.backend, keras, keras.utils
        from keras.models import Sequential, Model
        ae = Sequential()
        encoder = Sequential() # the network used here is exact 2 layers
        decoder = Sequential()
        for dim in [512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 10]:
        ae.add(keras.layers.Dense(dim, activation='tanh'))
        if len(encoder.layers) == len(decoder.layers):
            continue
        encoder.add(ae.layers[-1])
        decoder.add(ae.layers[-1])

        ae.add(keras.layers.Dense(10, activation='linear', kernel_initializer='identity'))
        ae.compile(optimizer='adadelta', loss='mse')

        # training for this network
        ae.fit(X_10d_2moons, X_10d_2moons,
               epochs=10,
               batch_size=256,
               shuffle=True,
               verbose=0)

        # encode the training data
        X_2d_2moons_enc = encoder.predict(X_10d_2moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```

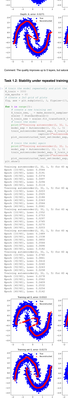


Task 1.1: Hyperparameter optimization

```
In [5]: # keras backend stuff
        import keras, keras.models, keras.layers, keras.backend, keras, keras.utils
        from keras.models import Sequential, Model
        ae = Sequential()
        encoder = Sequential() # the network used here is exact 2 layers
        decoder = Sequential()
        for dim in [512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 10]:
        ae.add(keras.layers.Dense(dim, activation='tanh'))
        if len(encoder.layers) == len(decoder.layers):
            continue
        encoder.add(ae.layers[-1])
        decoder.add(ae.layers[-1])

        ae.add(keras.layers.Dense(10, activation='linear', kernel_initializer='identity'))
        ae.compile(optimizer='adadelta', loss='mse')

        # training for this network
        ae.fit(X_10d_2moons, X_10d_2moons,
               epochs=10,
               batch_size=256,
               shuffle=True,
               verbose=0)

        # encode the training data
        X_2d_2moons_enc = encoder.predict(X_10d_2moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```



```
In [6]: # 10-dim 2 moons (source_dimensionality, noisiness)
        X_10d_2moons, y_10d_2moons = make_moons(10**5, noise=0.1, random_state=2)
        X_10d_2moons = X_10d_2moons[:,2:]

        # scale the coordinates
        X_10d_2moons = StandardScaler().fit_transform(X_10d_2moons)

        # encode the training data
        X_2d_2moons_enc = encoder.predict(X_10d_2moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```



```
In [7]: # 2-dim 5 moons (source_dimensionality, noisiness)
        X_2d_5moons, y_2d_5moons = make_moons(10**5, noise=0.1, random_state=3)

        # scale the coordinates
        X_2d_5moons = StandardScaler().fit_transform(X_2d_5moons)

        # encode the training data
        X_2d_5moons_enc = encoder.predict(X_2d_5moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_5moons_enc[:,0], X_2d_5moons_enc[:,1], lw)
```


Task 1.2: Reconstruction error

```
In [8]: # 2d-2 moons (source_dimensionality, noisiness)
        X_2d_2moons, y_2d_2moons = make_moons(10**5, noise=0.1, random_state=1)

        # scale the coordinates
        X_2d_2moons = StandardScaler().fit_transform(X_2d_2moons)

        # encode the training data
        X_2d_2moons_enc = encoder.predict(X_2d_2moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```



```
In [9]: # 10-dim 2 moons (source_dimensionality, noisiness)
        X_10d_2moons, y_10d_2moons = make_moons(10**5, noise=0.1, random_state=2)
        X_10d_2moons = X_10d_2moons[:,2:]

        # scale the coordinates
        X_10d_2moons = StandardScaler().fit_transform(X_10d_2moons)

        # encode the training data
        X_2d_2moons_enc = encoder.predict(X_10d_2moons)

        # plot the training data
        f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
        ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```



```
In [10]: # 2-dim 5 moons (source_dimensionality, noisiness)
         X_2d_5moons, y_2d_5moons = make_moons(10**5, noise=0.1, random_state=3)

         # scale the coordinates
         X_2d_5moons = StandardScaler().fit_transform(X_2d_5moons)

         # encode the training data
         X_2d_5moons_enc = encoder.predict(X_2d_5moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_5moons_enc[:,0], X_2d_5moons_enc[:,1], lw)
```


Task 1.3: Reproducibility

```
In [11]: # keras backend stuff
         import keras, keras.models, keras.layers, keras.backend, keras, keras.utils
         from keras.models import Sequential, Model
         ae = Sequential()
         encoder = Sequential() # the network used here is exact 2 layers
         decoder = Sequential()
         for dim in [512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 10]:
         ae.add(keras.layers.Dense(dim, activation='tanh'))
         if len(encoder.layers) == len(decoder.layers):
             continue
         encoder.add(ae.layers[-1])
         decoder.add(ae.layers[-1])

         ae.add(keras.layers.Dense(10, activation='linear', kernel_initializer='identity'))
         ae.compile(optimizer='adadelta', loss='mse')

         # training for this network
         ae.fit(X_10d_2moons, X_10d_2moons,
                epochs=10,
                batch_size=256,
                shuffle=True,
                verbose=0)

         # encode the training data
         X_2d_2moons_enc = encoder.predict(X_10d_2moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```


Comment: Seems like the reproducibility is not the case. The network converges to different solutions.

Task 1.4: Other architectures

```
In [12]: # keras backend stuff
         import keras, keras.models, keras.layers, keras.backend, keras, keras.utils
         from keras.models import Sequential, Model
         ae = Sequential()
         encoder = Sequential() # the network used here is exact 2 layers
         decoder = Sequential()
         for dim in [512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 10]:
         ae.add(keras.layers.Dense(dim, activation='tanh'))
         if len(encoder.layers) == len(decoder.layers):
             continue
         encoder.add(ae.layers[-1])
         decoder.add(ae.layers[-1])

         ae.add(keras.layers.Dense(10, activation='linear', kernel_initializer='identity'))
         ae.compile(optimizer='adadelta', loss='mse')

         # training for this network
         ae.fit(X_10d_2moons, X_10d_2moons,
                epochs=10,
                batch_size=256,
                shuffle=True,
                verbose=0)

         # encode the training data
         X_2d_2moons_enc = encoder.predict(X_10d_2moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```


Comment: Seems like the network gets bounded after 64 epochs, but keeps improving after 128 and 512 epochs. The final improvements looking acceptable. The reconstruction error seems to be fine.

We repeat the experiment with more training data.

```
In [13]: # keras backend stuff
         import keras, keras.models, keras.layers, keras.backend, keras, keras.utils
         from keras.models import Sequential, Model
         ae = Sequential()
         encoder = Sequential() # the network used here is exact 2 layers
         decoder = Sequential()
         for dim in [512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 10]:
         ae.add(keras.layers.Dense(dim, activation='tanh'))
         if len(encoder.layers) == len(decoder.layers):
             continue
         encoder.add(ae.layers[-1])
         decoder.add(ae.layers[-1])

         ae.add(keras.layers.Dense(10, activation='linear', kernel_initializer='identity'))
         ae.compile(optimizer='adadelta', loss='mse')

         # training for this network
         ae.fit(X_10d_2moons, X_10d_2moons,
                epochs=10,
                batch_size=256,
                shuffle=True,
                verbose=0)

         # encode the training data
         X_2d_2moons_enc = encoder.predict(X_10d_2moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)
```


Comment: We see that the model gets bounded after 64 epochs, but keeps improving after 128 and 512 epochs.

Task 1.5: Generative model

We now train a generative model (specifically a GMM) for the code distribution to enable sampling from the autoencoder.

```
In [14]: # 10-dim 2 moons (source_dimensionality, noisiness)
         X_10d_2moons, y_10d_2moons = make_moons(10**5, noise=0.1, random_state=2)
         X_10d_2moons = X_10d_2moons[:,2:]

         # scale the coordinates
         X_10d_2moons = StandardScaler().fit_transform(X_10d_2moons)

         # encode the training data
         X_2d_2moons_enc = encoder.predict(X_10d_2moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)

         # fitting a GMM to the encoded vectors
         from sklearn.mixture.gmm import GMM
         gmm = GMM(n_components=1, covariance_type='full')
         gmm.fit(X_2d_2moons_enc)

         # plotting the samples from the GMM and with their density characteristics
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(gmm.sample(10**5)[0][:,0], gmm.sample(10**5)[0][:,1], lw)
```

Comment: We code histogram consists of two clearly separated modes, one for each moon. The generated data are mainly indistinguishable from real examples. However, there are also some outliers. Unfortunately we can not quantify the goodness of the samples.

Task 1.6: Noisy moons without retraining

```
In [15]: # 10-dim 2 moons (source_dimensionality, noisiness)
         X_10d_2moons, y_10d_2moons = make_moons(10**5, noise=0.1, random_state=2)
         X_10d_2moons = X_10d_2moons[:,2:]

         # scale the coordinates
         X_10d_2moons = StandardScaler().fit_transform(X_10d_2moons)

         # encode the training data
         X_2d_2moons_enc = encoder.predict(X_10d_2moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)

         # fitting a GMM to the encoded vectors
         from sklearn.mixture.gmm import GMM
         gmm = GMM(n_components=1, covariance_type='full')
         gmm.fit(X_2d_2moons_enc)

         # plotting the samples from the GMM and with their density characteristics
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(gmm.sample(10**5)[0][:,0], gmm.sample(10**5)[0][:,1], lw)
```


Comment: Training is surprisingly often unsatisfactory. It seems that GMM training gets ran up quite enough. However, the initializa are interesting. The distribution mean and variance are fine because the two moons.

Task 1.7: Generative model

We now train a generative model (specifically a GMM) for the code distribution to enable sampling from the autoencoder.

```
In [16]: # 10-dim 2 moons (source_dimensionality, noisiness)
         X_10d_2moons, y_10d_2moons = make_moons(10**5, noise=0.1, random_state=2)
         X_10d_2moons = X_10d_2moons[:,2:]

         # scale the coordinates
         X_10d_2moons = StandardScaler().fit_transform(X_10d_2moons)

         # encode the training data
         X_2d_2moons_enc = encoder.predict(X_10d_2moons)

         # plot the training data
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(X_2d_2moons_enc[:,0], X_2d_2moons_enc[:,1], lw)

         # fitting a GMM to the encoded vectors
         from sklearn.mixture.gmm import GMM
         gmm = GMM(n_components=1, covariance_type='full')
         gmm.fit(X_2d_2moons_enc)

         # plotting the samples from the GMM and with their density characteristics
         f, ax = plt.subplots(1, figsize=(4, 4),frameon=False, lw)
         ax.scatter(gmm.sample(10**5)[0][:,0], gmm.sample(10**5)[0][:,1], lw)
```


Comment: Training is surprisingly often unsatisfactory. It seems that GMM training gets ran up quite enough. However, the initializa are interesting. The distribution mean and variance are fine because the two moons.

```
In [48]: # Generate test data for visualization
         test_data = torch.randn(1000, 784).to(device)
         with torch.no_grad():
             test_recon = model(test_data)
             test_loss = criterion(test_recon, test_data)
         print(f"Reconstruction test loss: {test_loss.item():.4f}")
         plot_reconstructions(test_data, test_recon, num_samples=10, save_path='reconstructions.
         plot_latent_space(model, test_data, save_path='latent_space.
         plot_loss_curves(train_losses, save_path='loss_curves.
         # Save model
         torch.save(model.state_dict(), 'mnist_autoencoder.
         print("Saved model to mnist_autoencoder.
```

```
Reconstruction test loss: 0.2149
```

Comment: The pre-trained autoencoder works well on the noisy dataset and still finds the centroids.

Task 1.5: Noisy mouse with retraining

```
In [49]: # Load pre-trained model
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))

         Epoch [1/20], Loss: 0.3148
         Epoch [2/20], Loss: 0.2346
         Epoch [3/20], Loss: 0.2264
         Epoch [4/20], Loss: 0.2227
         Epoch [5/20], Loss: 0.2207
         Epoch [6/20], Loss: 0.2187
         Epoch [7/20], Loss: 0.2174
         Epoch [8/20], Loss: 0.2164
         Epoch [9/20], Loss: 0.2154
         Epoch [10/20], Loss: 0.2147
         Epoch [11/20], Loss: 0.2140
         Epoch [12/20], Loss: 0.2134
         Epoch [13/20], Loss: 0.2129
         Epoch [14/20], Loss: 0.2124
         Epoch [15/20], Loss: 0.2120
         Epoch [16/20], Loss: 0.2116
         Epoch [17/20], Loss: 0.2112
         Epoch [18/20], Loss: 0.2109
         Epoch [19/20], Loss: 0.2106
         Epoch [20/20], Loss: 0.2103
```

```
In [50]: # Generate test data for visualization
         test_data = torch.randn(1000, 784).to(device)
         with torch.no_grad():
             test_recon = model(test_data)
             test_loss = criterion(test_recon, test_data)
         print(f"Reconstruction test loss: {test_loss.item():.4f}")
         plot_reconstructions(test_data, test_recon, num_samples=10, save_path='reconstructions.
         plot_latent_space(model, test_data, save_path='latent_space.
         plot_loss_curves(train_losses, save_path='loss_curves.
         # Save model
         torch.save(model.state_dict(), 'mnist_autoencoder.
         print("Saved model to mnist_autoencoder.
```

```
Reconstruction test loss: 0.2149
```

Comment: Training reconstructions on noisy data leads to a spacing out. It does not always follow the centroids, but measures cost the data to stance the reconstruction cost.

Task 2: Autoencoder without bottleneck

```
In [51]: class DenseAutoencoder(nn.Module):
             def __init__(self, hidden_dim=128, latent_dim=784):
                 super(DenseAutoencoder, self).__init__()
                 self.hidden_dim = hidden_dim
                 self.latent_dim = latent_dim

                 # Encoder: 784 -> hidden_dim -> latent_dim
                 self.encoder = nn.Sequential(
                     nn.Linear(784, hidden_dim),
                     nn.ReLU(),
                     nn.Linear(hidden_dim, latent_dim)
                 )

                 # Decoder: latent_dim -> hidden_dim -> 784
                 self.decoder = nn.Sequential(
                     nn.Linear(latent_dim, hidden_dim),
                     nn.ReLU(),
                     nn.Linear(hidden_dim, 784),
                     nn.Sigmoid()
                 )

             def forward(self, x):
                 # Flatten if needed
                 if x.shape[-1] != 784:
                     x = x.view(x.shape[0], -1)
                 z = self.encoder(x)
                 recon = self.decoder(z)
                 return recon, z

         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         print(f"Dense autoencoder parameters: {sum(p.numel() for p in DenseModel.parameters())}")

         Dense autoencoder parameters: 203264
```

In addition to the reconstruction error, we need an MSE loss to pull the code distribution towards the desired standard normal. We use a cosine annealing schedule as a stable transitioning.

```
In [52]: def train_dense_autoencoder(model, lr=0.001, epochs=20):
             """Train dense autoencoder with MSE loss"""
             optimizer = torch.optim.Adam(model.parameters(), lr=lr)
             criterion = nn.MSE(reduction='mean')
             scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

             losses = []
             for epoch in range(epochs):
                 epoch_loss = 0.0
                 for batch_idx, (data, _) in enumerate(train_loader):
                     data = data.view(data.shape[0], -1).to(device)
                     optimizer.zero_grad()
                     recon, z = model(data)
                     loss = criterion(recon, data)
                     loss.backward()
                     optimizer.step()
                     epoch_loss += loss.item()
                 avg_loss = epoch_loss / len(train_loader)
                 losses.append(avg_loss)
                 scheduler.step()
                 if (epoch + 1) % 5 == 0:
                     print(f"Epoch [{epoch+1}/{epoch}], Loss: {avg_loss:.4f}, LR: {optimizer.
             return losses

         print("Training Dense Autoencoder...")
         dense_losses = train_dense_autoencoder(DenseModel, epochs=20)

         Training Dense Autoencoder...
         Epoch [5/20], Loss: 0.0000, LR: 0.0009
         Epoch [10/20], Loss: 0.0000, LR: 0.0005
         Epoch [15/20], Loss: 0.0000, LR: 0.0002
         Epoch [20/20], Loss: 0.0000, LR: 0.0000
```

The training function is similar to the plain autoencoder. Only the loss is changed into a weighted sum of reconstruction error and MMD.

```
In [53]: def train_dense_autoencoder(model, lr=0.001, epochs=20):
             """Train dense autoencoder with MSE loss"""
             optimizer = torch.optim.Adam(model.parameters(), lr=lr)
             criterion = nn.MSE(reduction='mean')
             scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

             losses = []
             for epoch in range(epochs):
                 epoch_loss = 0.0
                 for batch_idx, (data, _) in enumerate(train_loader):
                     data = data.view(data.shape[0], -1).to(device)
                     optimizer.zero_grad()
                     recon, z = model(data)
                     loss = criterion(recon, data)
                     loss.backward()
                     optimizer.step()
                     epoch_loss += loss.item()
                 avg_loss = epoch_loss / len(train_loader)
                 losses.append(avg_loss)
                 scheduler.step()
                 if (epoch + 1) % 5 == 0:
                     print(f"Epoch [{epoch+1}/{epoch}], Loss: {avg_loss:.4f}, LR: {optimizer.
             return losses

         print("Training Dense Autoencoder...")
         dense_losses = train_dense_autoencoder(DenseModel, epochs=20)

         Training Dense Autoencoder...
         Epoch [5/20], Loss: 0.0000, LR: 0.0009
         Epoch [10/20], Loss: 0.0000, LR: 0.0005
         Epoch [15/20], Loss: 0.0000, LR: 0.0002
         Epoch [20/20], Loss: 0.0000, LR: 0.0000
```

Hence function is plot results.

```
In [54]: def plot_dense_comparison(dense_losses, save_path=None):
             """Compare training losses"""
             plt.figure(figure_size=(10, 6))
             plot_loss_curves(dense_losses, save_path=None, figure_size=(10, 6))
             plt.title('Dense Autoencoder Training Loss', fontsize=14)
             plt.text(0.0, 0.0, f'Final loss: {dense_losses[-1]:.4f}', transform=plt.gca().transf
             if save_path:
                 plt.savefig(save_path, dpi=300, bbox_inches='tight')
             plt.show()

         plot_dense_comparison(dense_losses)
         print("Dense autoencoder training complete!")
         print(f"Final loss: {dense_losses[-1]:.4f}")

         Dense autoencoder training complete!
         Final loss: 0.0000
```

We use sufficient hyperparameters (hidden_dim = 128, latent_dim = 784) to demonstrate that convergence of the code distribution is slow. After 20 epochs, the loss still shows significant variation. This demonstrates that without a bottleneck, the model struggles to learn the standard normal distribution. Dense layers of the code distribution tend to generalize rows pixels between the rooms, where they don't belong. Even after 100 epochs, the reconstruction quality improves but the code distribution remains far from standard normal. This shows that the reconstruction error is dominating the gradient of magnitude smaller, and the regularization is being ignored by MMD scores.

In contrast, reconstruction without a bottleneck seems to be easy — it is already good after 10 epochs and doesn't change much upon further training.

```
In [55]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```

Check if training reconstruction from MMD reconstruction. Results are not better, but not catastrophic. so

```
In [56]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```

Finally, we compare latent statistics and better. Improvements seem to be small, so that it is questionable whether the additional training is worthwhile.

```
In [57]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```

Task 3: Autoencoder for the digits dataset

```
In [58]: # Load CIFAR-10 dataset
         print("Loading CIFAR-10 dataset...")
         # Define transforms for CIFAR-10
         transform = transforms.ToTensor()
         # Download and load CIFAR-10
         cifar_train = datasets.CIFAR10('./data', train=True, download=True, transform=transform
         cifar_test = datasets.CIFAR10('./data', train=False, transform=transform)
         # Create data loaders
         cifar_train_loader = DataLoader(cifar_train, batch_size=128, shuffle=True)
         cifar_test_loader = DataLoader(cifar_test, batch_size=128, shuffle=False)
         print(f"CIFAR-10 training samples: {len(cifar_train)}")
         print(f"CIFAR-10 test samples: {len(cifar_test)}")
         # Visualize some CIFAR-10 images
         fig, axes = plt.subplots(2, 5, figure_size=(12, 5))
         for i, ax in enumerate(axes.flatten()):
             image, label = cifar_train[i]
             ax.imshow(image.permute(1, 2, 0))
             ax.set_title(f'{cifar_train.classes[label]}')
             ax.axis('off')
         plt.suptitle('CIFAR-10 Training Samples', fontsize=14)
         plt.tight_layout()
         plt.savefig('cifar10_samples.
         plt.show()
         print("Saved cifar10_samples.")

         Loading CIFAR-10 dataset...
         Downloading https://www.cs.toronto.edu/kriz/cifar-10-python.tar.gz
         Extracting ./data/cifar-10-python.tar.gz to ./data
         Files already downloaded and verified
         CIFAR-10 training samples: 50000
         CIFAR-10 test samples: 10000
         Saved cifar10_samples.
```

We do the same thing for the digits dataset. Since the digits are smaller than the digits, we need to pad the first conv layer to

```
In [59]: class ConvAutoencoderCIFAR(nn.Module):
             def __init__(self, latent_dim=2):
                 super(ConvAutoencoderCIFAR, self).__init__()
                 self.latent_dim = latent_dim

                 # Encoder: 3x32x32 -> latent_dim
                 self.encoder = nn.Sequential(
                     nn.Conv2d(3, 32, 4, stride=2, padding=1),  # 32x16x16
                     nn.ReLU(),
                     nn.Conv2d(32, 64, 4, stride=2, padding=1),  # 64x8x8
                     nn.ReLU(),
                     nn.Conv2d(64, 128, 4, stride=2, padding=1),  # 128x4x4
                     nn.ReLU(),
                     nn.Conv2d(128, 256, 4, stride=2, padding=1),  # 256x2x2
                     nn.ReLU(),
                     nn.Flatten(),
                     nn.Linear(256*2*2, 512),
                     nn.ReLU(),
                     nn.Linear(512, latent_dim)
                 )

                 # Decoder: latent_dim -> 3x32x32
                 self.decoder = nn.Sequential(
                     nn.Linear(latent_dim, 512),
                     nn.ReLU(),
                     nn.Linear(512, 256*2*2),
                     nn.ReLU(),
                     nn.Unflatten(1, (256, 2, 2)),
                     nn.Conv2d(256, 128, 4, stride=2, padding=1),  # 128x4x4
                     nn.ReLU(),
                     nn.Conv2d(128, 64, 4, stride=2, padding=1),  # 64x8x8
                     nn.ReLU(),
                     nn.Conv2d(64, 32, 4, stride=2, padding=1),  # 32x16x16
                     nn.ReLU(),
                     nn.Conv2d(32, 3, 4, stride=2, padding=1),  # 3x32x32
                     nn.Sigmoid()
                 )

             def forward(self, x):
                 z = self.encoder(x)
                 recon = self.decoder(z)
                 return recon, z

         CIFARModel = ConvAutoencoderCIFAR(latent_dim=2).to(device)
         print(f"CIFAR autoencoder parameters: {sum(p.numel() for p in CIFARModel.parameters())}")

         CIFAR autoencoder parameters: 1,180,163
```

We use the same function as the digits dataset to analyze the quality of our reconstructions and synthetic images.

```
In [60]: # Train CIFAR autoencoder
         print("Training CIFAR autoencoder...")
         cifar_losses = train_autoencoder(CIFARModel, cifar_train_loader, epochs=20, lr=0.001)

         Training CIFAR autoencoder...
         Epoch [1/20], Loss: 0.0000
         Epoch [2/20], Loss: 0.0000
         Epoch [3/20], Loss: 0.0000
         Epoch [4/20], Loss: 0.0000
         Epoch [5/20], Loss: 0.0000
         Epoch [6/20], Loss: 0.0000
         Epoch [7/20], Loss: 0.0000
         Epoch [8/20], Loss: 0.0000
         Epoch [9/20], Loss: 0.0000
         Epoch [10/20], Loss: 0.0000
         Epoch [11/20], Loss: 0.0000
         Epoch [12/20], Loss: 0.0000
         Epoch [13/20], Loss: 0.0000
         Epoch [14/20], Loss: 0.0000
         Epoch [15/20], Loss: 0.0000
         Epoch [16/20], Loss: 0.0000
         Epoch [17/20], Loss: 0.0000
         Epoch [18/20], Loss: 0.0000
         Epoch [19/20], Loss: 0.0000
         Epoch [20/20], Loss: 0.0000
```

```
In [61]: # Analyze CIFAR autoencoder
         print("Analyzing CIFAR autoencoder...")
         # Create test data
         test_cifar = torch.randn(1000, 3, 32, 32).to(device)
         # Compute reconstructions
         with torch.no_grad():
             cifar_recon, cifar_z = CIFARModel(test_cifar)
         # Plot reconstruction comparison
         plot_reconstruction_comparison(test_cifar, cifar_recon, num_samples=10, save_path='cifa
         # Plot latent space
         plot_latent_space(CIFARModel, test_cifar, save_path='cifar_latent_space.
         # Plot loss curves
         plot_loss_curves(cifar_losses, save_path='cifar_loss_curves.

         Analyzing CIFAR autoencoder...
         Saved cifar_reconstruction_comparison.
         Saved cifar_latent_space.
         Saved cifar_loss_curves.
```

We run the Autoencoder trained models to the three code spaces.

```
In [62]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))
         CIFARModel = ConvAutoencoderCIFAR(latent_dim=2).to(device)
         CIFARModel.load_state_dict(torch.load('cifar_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)
         test_cifar = torch.randn(1000, 3, 32, 32).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)
             cifar_recon, cifar_z = CIFARModel(test_cifar)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```


We run the Autoencoder trained models to the three code spaces.

```
In [63]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))
         CIFARModel = ConvAutoencoderCIFAR(latent_dim=2).to(device)
         CIFARModel.load_state_dict(torch.load('cifar_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)
         test_cifar = torch.randn(1000, 3, 32, 32).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)
             cifar_recon, cifar_z = CIFARModel(test_cifar)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```


We run the Autoencoder trained models to the three code spaces.

```
In [64]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))
         CIFARModel = ConvAutoencoderCIFAR(latent_dim=2).to(device)
         CIFARModel.load_state_dict(torch.load('cifar_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)
         test_cifar = torch.randn(1000, 3, 32, 32).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)
             cifar_recon, cifar_z = CIFARModel(test_cifar)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```


We run the Autoencoder trained models to the three code spaces.

```
In [65]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))
         CIFARModel = ConvAutoencoderCIFAR(latent_dim=2).to(device)
         CIFARModel.load_state_dict(torch.load('cifar_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)
         test_cifar = torch.randn(1000, 3, 32, 32).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)
             cifar_recon, cifar_z = CIFARModel(test_cifar)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```


We run the Autoencoder trained models to the three code spaces.

```
In [66]: # Load models
         print("Loading models...")
         model = Autoencoder(latent_dim=2).to(device)
         model.load_state_dict(torch.load('mnist_autoencoder.pth', map_location=device))
         DenseModel = DenseAutoencoder(hidden_dim=128, latent_dim=784).to(device)
         DenseModel.load_state_dict(torch.load('dense_autoencoder.pth', map_location=device))
         CIFARModel = ConvAutoencoderCIFAR(latent_dim=2).to(device)
         CIFARModel.load_state_dict(torch.load('cifar_autoencoder.pth', map_location=device))

         # Create test data
         test_data = torch.randn(1000, 784).to(device)
         test_cifar = torch.randn(1000, 3, 32, 32).to(device)

         # Compute reconstructions
         with torch.no_grad():
             conv_recon, conv_z = model(test_data)
             dense_recon, dense_z = DenseModel(test_data)
             cifar_recon, cifar_z = CIFARModel(test_cifar)

         print("Creating comparison plots...")
         plot_latent_comparison(conv_z, dense_z, save_path='latent_comparison.
         plot_reconstruction_comparison(test_data, conv_recon, dense_recon, num_samples=10, sav

         Loading models...
         Creating comparison plots...
         Latent space statistics:
         Conv - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Dense - Mean: [-0.0000, -0.0000], Std: [0.9999, 0.9999]
         Reconstruction quality:
         Conv - MSE: 0.0000, PSNR: 40.00 dB
         Dense - MSE: 0.0000, PSNR: 40.00 dB
         Saved latent_comparison.
         Saved reconstruction_comparison.
```


Finally, we COMPA the two code distributions and the different bottlenecks (conv and dense) and compare synthetic digit images. On top of each image, we print the digit predicted by the classifier trained. In most cases, it is the same that the net is forced to social design. The quality of reinforcement case is a lot worse than an ample bottleneck.

In [100]: display_predicts = 1

```
fig, ax = plt.subplots(1, display_predicts, sz)

for i in range(display_predicts):
    ax[i].imshow(np.reshape(x_test[i], (28, 28)))
    ax[i].set_xticks(np.arange(0, 28))
    ax[i].set_yticks(np.arange(0, 28))
    ax[i].grid(True)
    ax[i].set_xticklabels([])
    ax[i].set_yticklabels([])
    ax[i].set_title('')
    if y_test[i] == 1:
        ax[i].set_title('Cat')
    else:
        ax[i].set_title('Dog')
    ax[i].text(0.5, -0.1, 'predicted: {:.2f}'.format(y_predicts[i]), fontsize=12, transform=ax[i].transAxes)
    if y_test[i] == 1:
        ax[i].text(0.5, -0.2, 'Cat', fontsize=12, transform=ax[i].transAxes)
    else:
        ax[i].text(0.5, -0.2, 'Dog', fontsize=12, transform=ax[i].transAxes)
    ax[i].text(0.5, -0.3, 'Correct', fontsize=12, transform=ax[i].transAxes)
    ax[i].text(0.5, -0.4, '{:.2f}%'.format(100*y_predicts[i]), fontsize=12, transform=ax[i].transAxes)
plt.show()
```


The confidence of the cited predictions for cat/dog means a cat/dogs in among a lot/ignore them for net/data, but not distinguishes as. The shown first are generated some/some pets and for it/breed/ones and it, but is not/ones to it/breed/ones and it. The might be a consequence of the very small training set. We conclude that it/breed/ones and it is a bad for this task.

In [101]:

```
for i in range(display_predicts):
    print('Predicted: {:.2f}'.format(y_predicts[i]))
    print('Predicted: {:.2f%}'.format(100*y_predicts[i]))
    print('Correct: {:.2f}'.format(y_test[i]))
    print('Correct: {:.2f%}'.format(100*y_test[i]))
    print('Correct: {:.2f%}'.format(100*y_test[i]))
    print('')
```

```
Predicted: 0.01
Predicted: 0.96%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%

Predicted: 0.01
Predicted: 0.97%
Correct: 1.00
Correct: 100.00%
Correct: 100.00%
```