Machine Learning at Berkeley: Machine Learning Decal

Homework Three: Unsupervised Learning and Autoencoders

Release Date: February 27th, 2019 Due Date: March 11th, 2019 Contributing Authors: Brandon Trabucco

The goal of this homework is to familiarize you with various unsupervised learning and dimensionality reduction algorithms that are commonly used when handling large datasets. In particular, you will implement:

- Extracting The Dataset
- Principal Component Analysis
- A Linear Autoencoder
- A Convolutional Autoencoder
- (Optional) A Variational Autoencoder (VAE for short)

In addition to implementing these algorithms, you will use these algorithms to interpolate between existing data points, and extrapolate to new data points. Since images have nice visualizations, this homework shall use a miniature version of the CelebA (S. Yang et al. 2015) dataset that contains 5000 cropped images of celebrity faces. Feel free to download the full dataset after finishing the homework and tinkering with your models.

S. Yang, P. Luo, C. C. Loy, and X. Tang, "From Facial Parts Responses to Face Detection: A Deep Learning Approach", in IEEE International Conference on Computer Vision (ICCV), 2015

In [56]:
```
%%capture
# IMPORTANT: you must have all of these repositories properly installed on you
r machine to complete this homework.
# you must also have ffmpeg installed. You may find the binaries at https://ww
w.ffmpeg.org/download.html
# Make sure you add the directories that contain the ffmpeg binaries to your p
ath, reinstall matplotlib afterwards
import torch
import torchvision
import torch.nn.functional as F
import glob
import os
from PIL import Image
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.animation as animation
if not "ffmpeg" in matplotlib.animation.writers.list():
    print("WARNING!!! You must add FFMPEG to your path before you can use the
 animations in this homework.")
from IPython.display import HTML, display
```

Section One: Extracting The Dataset

In this section, you will extract the folder of images into a matrix $X \in \mathcal{R}^{N \times D}$ where the number of rows N corresponds to the number of images in the dataset (5000 in total), and the number of features D corresponds to the RGB values of every pixel in every image (32 *32* 3 = 3072 in this case).

In [57]:
```
def extract_dataset(path_to_images, output_height, output_width, path_to_matri
x):
    """Loads each image into memory, processes each image, and saves a matrix
 to the disk.
    Args:
        path_to_images: string, the path to the directory containing image fil
es.
        output_height: integer, the height to scale each image to.
        output_width: integer, the width to scale each image to.
        path_to_matrix: string, the path where the matrix will be saved.
    """
    all_matching_files = glob.glob(os.path.join(path_to_images, "*.jpg"))
    X = np.zeros([len(all_matching_files), output_height * output_width * 3])
    for i, file in enumerate(all_matching_files):
        # TODO: fill in this section to accomplish the following.
        # 1) load the image with Image.open specified by its file path from th
e disk
        # 2) resize that image to be a [output_width, output_height] numpy arr
ay
        # 3) perform a row-major flatten of the array
        # 4) scale the elements of the array to be in the range [-1, 1]
        # 5) assign the array to the ith column of data matrix X
        # BEGIN YOUR CODE
        im = Image.open(file)
        arr = np.array(im.resize((output_width, output_height), resample=0))
        arr = arr.flatten()
        arr = np.interp(arr, (arr.min(), arr.max()), (-1, +1))
        X[i,:] = arr
        # END YOUR CODE
    np.save(os.path.join(path_to_matrix, "dataset.npy"), X)
```

In [58]:
```
def load_dataset(path_to_matrix):
    """Loads a matrix containing processed images into the memory.
    Args:
        path_to_matrix: string, the path where the matrix was saved.
    Returns:
        a numpy matrix with 5000 rows (one per image) and 3072 columns.
    """
    return np.load(os.path.join(path_to_matrix, "dataset.npy"))
```

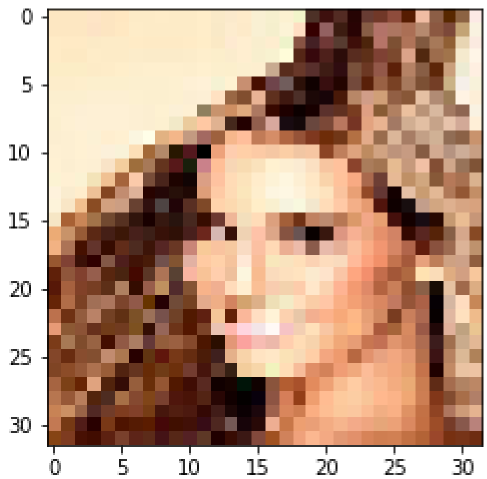
In [59]:
```
def show_image(flat_image_vector, output_height, output_width):
    """Displays an image on jupyter notebook using matplotlib imshow.
    Args:
        flat_image_vector: a np.float32 vector with D = 3072 elements.
        output_height: integer, the height to reshape the image to.
        output_width: integer, the width to reshape the image to.
    """
    # TODO: fill in this section to accomplish the following.
    # 1) perform a row-major reshape from a flattened array to a [output_heigh
t, output_width, 3] tensor
    # 2) scale the elements of the array to be in the range [0, 1]
    # 3) render the image using matplotlib imshow(...)
    # 4) show() and close() the plot
    # BEGIN YOUR CODE
    flat_image_vector = flat_image_vector.reshape(output_height, output_width,
3)
    flat_image_vector = np.interp(flat_image_vector, (flat_image_vector.min(),
flat_image_vector.max()), (0, +1))
    plt.imshow(flat_image_vector)
    plt.show()
    plt.close()
    # END YOUR CODE
```

In [60]:
```
# TODO: fill in this section to accomplish the following.
# 1) call the extract_dataset function with the appropriate paths
# 2) assign the height 32 and the width 32
# BEGIN YOUR CODE
extract_dataset("C:/Users/anika/Downloads/celeba/celeba", 32, 32, "C:/Users/an
ika/Downloads")
# END YOUR CODE
```

In [61]:
```
# TODO: fill in this section to accomplish the following.
# 1) call the load_dataset function with the appropriate path
# 2) assign the result to a data matrix named X
# BEGIN YOUR CODE
X = load_dataset(r"C:\Users\anika\Downloads")
# END YOUR CODE
print("A matrix with {0} images and {1} features per image was loaded.".format
(*X.shape))
```

```
A matrix with 5000 images and 3072 features per image was loaded.
```

In [62]:
```
# TODO: fill in this section to accomplish the following.
# 1) call the show_image function using a single row from the matrix X
# BEGIN YOUR CODE
show_image(X[0, :], 32, 32)
# END YOUR CODE
```



Section Two: Principal Component Analysis

In this section, you will learn about Principal Component Analysis from an optimization perspective. You will then implement PCA to learn the K principal components from the data matrix X. You will then use these principal components to interpolate between random rows of X. Finally, you will sample points in a lower dimensional subspace and invert PCA to generate new images of faces.

The rows of X lives in the space of $\mathcal{R}^D$. We define D to be 3072 for the remainder of this homework. The principal components of X provide a sequence of the best linear approximations to X in a lower dimensional subspace $\mathcal{R}^Q$ where the rank of the subspace $Q \leq D$ is no larger than the rank of the space that contains X. Consider a function of a vector λ in $\mathcal{R}^Q$.

$$f(\lambda) = \mu + V_Q \lambda$$

This function defines a linear transformation from the space of $\mathcal{R}^Q$ to the space of $\mathcal{R}^D$. There are two important parameters in this formulation: namely μ and V_Q. The vector μ is a position in the space of $\mathcal{R}^D$. The matrix $V_Q \in \mathcal{R}^{D \times Q}$ is a unitary matrix that maps the vector λ from the subspace $\mathcal{R}^Q$ to the space of the data $\mathcal{R}^D$. The goal of PCA is to minimize the following reconstruction error.

$$\min_{\mu, V_Q, \{\lambda_i\}} \sum_{i=1}^{N} ||x_i - \mu - V_Q \lambda_i||_2^2$$

Where the vector x_i is the row in position i from the data matrix X, and the vector λ_i represent the best approximation of the vector x_i in the column space of the matrix V_Q. The other parameters have been previously defined, and are the same. We take this objective, and we optimize for μ and $\{\lambda_i\}$.

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$
$$\lambda_i = V_Q^T (x_i - \mu)$$

The optimization objective now amounts to solving for the optimal orthonormal matrix V_Q that minimizes reconstruction error.

$$\min_{V_Q} \sum_{i=1}^{N} ||x_i - \frac{1}{N} \sum_{i=1}^{N} x_i - V_Q V_Q^T (x_i - \frac{1}{N} \sum_{i=1}^{N} x_i)||_2^2$$

The matrix resulting from $V_Q V_Q^T$ can be imagined a projection that maps each data point x_i onto the best rank Q approximation. See that we are subtracting the mean from each data point. If we assume each data point already has zero mean, the objective simplifies.

$$\frac{1}{N} \sum_{i=1}^{N} x_i = 0 \implies \min_{V_Q} \sum_{i=1}^{N} ||x_i - V_Q V_Q^T x_i||_2^2$$

The solution may be obtained using Singular Value Decomposition. In particular, we can express the data matrix by its SVD $X = U \Sigma V^T$. Here, U is an $N \times D$ orthogonal matrix. The matrix $U\Sigma$ represents the principal components of X, the directions with highest variance. The solution for V_Q is simply to take the first Q columns of the matrix V. This is left as an exercise for the reader and is not required.

In [63]:
```
# TODO: fill in this section to accomplish the following.
# 1) using the function np.linalg.svd, calculate the singular value decomposit
ion of the data matrix X
# 2) assign the SVD results to three matrices: U, S, V_T
# BEGIN YOUR CODE
A = np.linalg.svd(X, compute_uv=1)
U, S, V_T = A[0], A[1], A[2]
# END YOUR CODE
```

In [64]:
```
# TODO: fill in this section to accomplish the following.
# 1) define Q = 256 to be the rank of the lower dimensional subspace in which
 you shall embed the data points
# 2) define variances to be the first Q singular values
# 3) define principal_components to be the basis vectors corresponding to the
 first Q singular values
# 4) define V_Q to be the matrix consisting of the first Q columns of V
# BEGIN YOUR CODE
Q = 256
variances = S[:Q]
principal_components = np.matmul(U[:, :Q], variances)
V_Q = np.transpose(V_T)[:, :Q]
# END YOUR CODE

print("The variances along the first {0} principal components are: {1}".format
(Q, variances))
print("The first {0} principal components are: {1}".format(Q, principal_compon
ents))
print("The first {0} right singular vectors are: {1}".format(Q, V_Q))
```

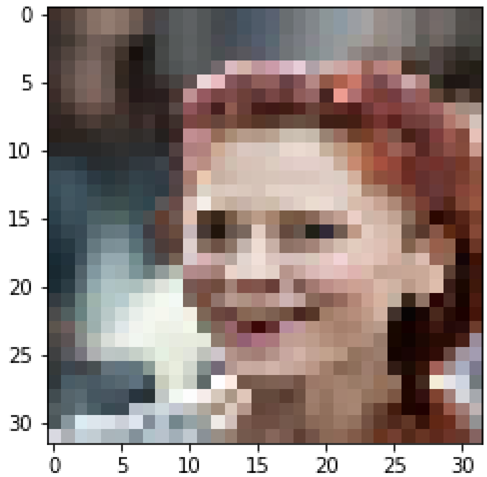
```
The variances along the first 256 principal components are: [1385.04217087  9
12.64752404  567.69063325  503.60281896  444.02342026
  387.70277861  370.30681225  353.95860315  293.88451622  284.72437915
  274.90384158  258.8822779   250.99486015  225.31653829  214.66430889
  210.65833985  195.50016485  187.48634671  181.75741715  176.44608263
  172.50442187  157.23604971  154.23229432  150.80668455  144.52491069
  139.43884054  137.47533452  134.3856944   130.95337227  130.02354359
  127.79035675  126.44931528  125.50775747  124.65768218  120.48078051
  119.45877059  114.92585548  113.42735303  111.58544088  109.57947975
  107.12094155  105.72623075  103.25813773  102.11646637  101.2658764
   97.81924383   96.7548743    95.2855847    94.42577194   92.56881158
   91.86844494   91.22472529   90.28176138   89.6405042    88.26636286
   87.16335853   86.32683381   83.93040446   83.46319741   81.84591969
   81.49903119   80.62701048   79.48054618   78.84522164   77.9764179
   77.56393324   76.58291043   76.31287013   76.12557083   75.19344507
   74.27231875   72.8071794    72.38443169   71.6637436    71.24967257
   70.42612194   70.00424298   68.66081929   68.34678099   67.76972067
   67.19887492   66.8847789    66.12105139   65.74328156   65.57563094
   65.15260562   64.61605503   63.91784248   63.71556236   62.93932132
   62.58128788   62.08538371   61.72863171   61.18914957   60.82931864
   60.74507873   60.34347231   60.1692592    59.99210031   59.42563669
   59.30393085   58.61324605   58.26491913   58.10602925   57.67145984
   57.35258194   56.97801301   56.68671662   56.18989629   55.93298407
   55.50797874   54.8606683    54.67960284   54.55705325   54.4609478
   53.70331126   53.44975189   53.13394879   52.86314593   52.57499841
   52.37905008   51.82648226   51.5715215    51.22721322   50.89385592
   50.84194737   50.59156207   50.3240394    50.09748318   49.76956839
   49.57248477   49.1383492    48.94597597   48.93454314   48.75579428
   48.63868358   48.42315446   47.95729276   47.88460693   47.57712162
   47.36299492   47.24477453   46.96996438   46.7856405    46.76134431
   46.48754899   46.30455196   46.16482844   46.12919398   45.93761292
   45.88284158   45.44755838   45.37918156   45.22682896   44.79889007
   44.68555521   44.47551674   44.31094814   44.22194067   44.08368538
   44.06886664   43.83396118   43.63900785   43.4400458    43.21157629
   43.06042465   42.94578595   42.64871282   42.51036536   42.36546878
   42.2425418    42.16019808   41.96102593   41.82099739   41.64934914
   41.51592745   41.33720813   41.25875627   41.1041806    41.04890355
   40.89935667   40.84160503   40.57296295   40.31288266   40.28298697
   40.18755991   40.00963312   39.76509449   39.6858947    39.56619536
   39.50450543   39.41980077   39.34819893   39.27537153   39.08846232
   38.95172029   38.86626531   38.71672445   38.60613251   38.35843668
   38.25468772   38.19117323   38.13521091   37.99245175   37.83006042
   37.78342784   37.70350533   37.64828826   37.49683263   37.42174931
   37.20170023   37.14619875   37.03627974   36.97490515   36.81900688
   36.65695401   36.63822502   36.50518131   36.44721074   36.3610514
   36.24671244   36.15522477   36.09392859   35.95650002   35.894469
   35.82969955   35.66515175   35.60202772   35.46429931   35.27881205
   35.26503398   35.16422317   34.99774963   34.91891427   34.88647464
   34.75147497   34.67995872   34.6476579    34.54061316   34.48621129
   34.43176474   34.3328586    34.22796789   34.1475179    34.10922787
   34.00490572   33.85432015   33.79833628   33.78544487   33.66658465
   33.55764738   33.49060062   33.42750184   33.35188785   33.3336392
   33.22094569]
The first 256 principal components are: [ 25.54231205  36.85310352 -57.495212
88 ...  58.23494038  -7.18564589
  15.17088645]
The first 256 right singular vectors are: [[-0.02390227  0.02451178  0.007726
```

```
63 ...  0.0110737   0.02193316
  -0.01134639]
 [-0.02600947  0.02171382  0.00788852 ... -0.00943443  0.01354154
  -0.01524858]
 [-0.02670861  0.01939525  0.00840332 ...  0.00508976 -0.000343
  -0.00992379]
 ...
 [-0.01152804 -0.00435087 -0.01443639 ... -0.00926149 -0.01049446
  -0.0136619 ]
 [-0.01400566 -0.01258904 -0.01494839 ... -0.01818521  0.01011941
  -0.01128622]
 [-0.01476598 -0.01633043 -0.01464998 ... -0.01614891  0.01554927
  -0.00429167]]
```

In [65]:

```
# TODO: fill in this section to accomplish the following.
# 1) select a single row of the data matrix X
# 2) project that row onto the rank-Q lower dimension subspace in R^D specifie
d by the projection matrix (V_Q V_Q^T)
# BEGIN YOUR CODE
A = np.matmul(np.matmul(V_Q, np.transpose(V_Q)), X[1, :])
# END YOUR CODE
```

In [66]:
```
# TODO: fill in this section to accomplish the following.
# 1) display the original image using show_image with height 32 and width 32
# 1) display the projected image using show_image with height 32 and width 32
# BEGIN YOUR CODE
show_image(X[1, :], 32, 32)
show_image(A, 32, 32)
# END YOUR CODE
```



Comment on how well the best Q principal components reconstruct the image:

The best Q principal components for me did not reconstruct the image very well.

In [67]:
```
def latent_interpolation(z_one, z_two, reconstruction_function, output_height, output_width):
    """This function draws an interpolating animation from one image to another image in the latent space.
    Args:
        z_one: an np.float32 vector with Q elements.
        z_two: an np.float32 vector with Q elements.
        reconstruction_function: a function that takes in z_one or z_two and returns an np.float32 vector with D elements.
        output_height: integer, the height to reshape each image to.
        output_width: integer, the width to reshape each image to.
    """
    fig = plt.figure()
    im = None
    im = plt.imshow(reconstruction_function(z_one).reshape([output_height, output_width, 3]) / 2.0 + 0.5, animated=True)
    def updatefig(t):
        alpha = (0.5 * np.cos(t) + 0.5)
        im.set_array(reconstruction_function(z_one * alpha + z_two * (1.0 - alpha)).reshape([output_height, output_width, 3]) / 2.0 + 0.5)
        return im,
    display(HTML(animation.FuncAnimation(fig, updatefig, frames=np.linspace(0, 2*np.pi, 64), blit=True).to_html5_video()))
    plt.close()
```

In [68]:
```
# TODO: fill in this section to accomplish the following.
# 1) select two different rows from the data matrix X
# 2) project each row onto the best Q principal components using V_Q^T
# 3) define reconstruction_function that reconstructs a data point x from its
 projection onto the best Q principal components using V_Q
# 4) call the function latent_interpolation and generate a visualization with
 height 32 and width 32
# BEGIN YOUR CODE
z_one = np.matmul(np.transpose(V_Q), X[0, :])
z_two = np.matmul(np.transpose(V_Q), X[1, :])
def reconstruction_function(x):
    return np.matmul(V_Q, x)
latent_interpolation(z_one, z_two, reconstruction_function, 32, 32)
# END YOUR CODE
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
```

```
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
```

0:00 / 0:12

Comment on what happens during the interpolation process:

Interpolation process finds points in common between the two images and the animation plays through formulating a sequence whereby the image changes to find points in-common between the two photos and combines them into the original image.

In [69]:
```
def latent_generation(z_mean, z_stddev, reconstruction_function, output_height
, output_width):
    """This function samples from the latent space of the model and shows the
 resulting image.
    Args:
        z_mean: an np.float32 vector with Q elements.
        z_stddev: an np.float32 matrix with Q by Q elements.
        reconstruction_function: a function that takes in z_one or z_two and r
eturns an np.float32 vector with D elements.
        output_height: integer, the height to reshape each image to.
        output_width: integer, the width to reshape each image to.
    """
    sampled_point = z_mean + z_stddev.dot(np.random.normal(0, 1, z_mean.shape
))
    show_image(reconstruction_function(sampled_point), output_height, output_w
idth)
```

In [73]:
```
# TODO: fill in this section to accomplish the following.
# 1) project the data matrix onto best Q principal components using V_Q^T
# 2) compute z_mean as the average of the projected matrix along the 0th axis
# 3) define z_stddev to be the rank Q identity matrix for now
# 4) generate an new image by calling latent_generation with height 32 and wid
th 32
# BEGIN YOUR CODE
A = np.matmul(X, V_Q)
z_mean = np.mean(A, axis=0)
z_stddev = np.eye(Q)
latent_generation(z_mean, z_stddev, reconstruction_function, 32, 32)
# END YOUR CODE
```

Comment on how real the generated face looks:

The generated face does not look very real. It looks very pixelated and varies too much in color to be accurate. (And it looks pretty creepy.)

Section Three: Linear Autoencoder

In this section, you will learn about the linear autoencoder, and you will also implement the linear autoencoder using pytorch. We shall use your linear autoencoder to interpolate between data points from X and to also generate new samples of faces.

In [74]:
```
class LinearEncoder(torch.nn.Module):

    def __init__(self, image_height, image_width, hidden_size):
        """Creates a single layer neural network.
        Args:
            image_height: an integer, the height of each image
            image_width: an integer, the width of each image
            hidden_size: an integer, the number of neurons in the hidden layer
of this network
        """
        super(LinearEncoder, self).__init__()
        # TODO: fill in this section to accomplish the following.
        # 1) create a single layer neural network that performs a linear trans
formation from a vector with
        #    image_height * image_width * 3 dimensions to a vector with hidden
_size dimensions.
        #    HINT: consider the class torch.nn.Linear
        # BEGIN YOUR CODE
        self.vector = torch.nn.Linear(image_height * image_width * 3, hidden_s
ize, bias=True)
        # END YOUR CODE

    def forward(self, x):
        """Computes a single forward pass of this network.
        Args:
            x: a float32 tensor with shape [batch_size, D]
        Returns:
            a float32 tensor with shape [batch_size, hidden_size]
        """
        # TODO: fill in this section to accomplish the following.
        # 1) perform a forward pass using the hidden layer you defined
        # 2) return the resulting vector
        # BEGIN YOUR CODE
        return self.vector(x)
        # END YOUR CODE
```

In [75]:
```
class LinearDecoder(torch.nn.Module):

    def __init__(self, image_height, image_width, hidden_size):
        """Creates a single layer neural network.
        Args:
            image_height: an integer, the height of each image
            image_width: an integer, the width of each image
            hidden_size: an integer, the number of neurons in the hidden layer
of this network
        """
        super(LinearDecoder, self).__init__()
        # TODO: fill in this section to accomplish the following.
        # 1) create a single layer neural network that performs a linear trans
formation from a vector with
        #    hidden_size dimensions to a vector with image_height * image_widt
h * 3 dimensions.
        #    HINT: consider the class torch.nn.Linear
        # BEGIN YOUR CODE
        self.vector = torch.nn.Linear(hidden_size, image_height * image_width
* 3, bias=True)
        # END YOUR CODE

    def forward(self, x):
        """Computes a single forward pass of this network.
        Args:
            x: a float32 tensor with shape [batch_size, hidden_size]
        Returns:
            a float32 tensor with shape [batch_size, D]
        """
        # TODO: fill in this section to accomplish the following.
        # 1) perform a forward pass using the hidden layer you defined
        # 2) return the resulting vector
        # BEGIN YOUR CODE
        return self.vector(x)
        # END YOUR CODE
```

In [130]:
```
# TODO: fill in this section to accomplish the following.
# 1) create an instance of LinearEncoder named linear_encoder with height 32,
 width 32, and hidden size Q
# 1) create an instance of LinearDecoder named linear_decoder with height 32,
 width 32, and hidden size Q
# 2) assign linear_autoencoder_loss to be an instance of torch.nn.MSELoss
# 2) create an optimizer named linear_autoencoder_optimizer of your choosing w
ith a learning rate of your choosing.
#    HINT: consider the torch.optim.Adam object
# BEGIN YOUR CODE
linear_encoder = LinearEncoder(32, 32, Q)
linear_decoder = LinearDecoder(32, 32, Q)
linear_autoencoder_loss = torch.nn.MSELoss()
linear_autoencoder_optimizer = torch.optim.Adam(linear_encoder.parameters(), l
r=0.001)
linear_autoencoder_optimizer = torch.optim.Adam(linear_decoder.parameters(), l
r=0.001)
# END YOUR CODE
```

In [131]:
```
# TODO: run the following section of code in order to train the model
# Construct a tensor from the dataset
image_tensor = torch.FloatTensor(X)
for i in range(1000):
    # Clear the previous gradient from the optimizer by calling .zero_grad()
    linear_autoencoder_optimizer.zero_grad()
    # Compute a full encoding and decoding step
    reconstructed_image = linear_decoder(linear_encoder(image_tensor))
    # Compute the mean squared reconstruction loss
    loss = linear_autoencoder_loss(reconstructed_image, image_tensor)
    # Pass the loss backward throgh the network, and compute the gradients
    loss.backward()
    # Update the optimizer by calling .step()
    linear_autoencoder_optimizer.step()
    # Return a detatched value of the loss for logging purposes
    print("On iteration {0} the loss was {1}.".format(i, loss.detach()))
```

```
On iteration 0 the loss was 0.43444961309432983.
On iteration 1 the loss was 0.40115466713905334.
On iteration 2 the loss was 0.37087303400039673.
On iteration 3 the loss was 0.3435925543308258.
On iteration 4 the loss was 0.3192284405231476.
On iteration 5 the loss was 0.29764971137046814.
On iteration 6 the loss was 0.2786896824836731.
On iteration 7 the loss was 0.26215413212776184.
On iteration 8 the loss was 0.24782854318618774.
On iteration 9 the loss was 0.23548570275306702.
On iteration 10 the loss was 0.2248927354812622.
On iteration 11 the loss was 0.2158181369304657.
On iteration 12 the loss was 0.2080378383398056.
On iteration 13 the loss was 0.20134077966213226.
On iteration 14 the loss was 0.19553311169147491.
On iteration 15 the loss was 0.19044150412082672.
On iteration 16 the loss was 0.18591497838497162.
On iteration 17 the loss was 0.18182571232318878.
On iteration 18 the loss was 0.1780686378479004.
On iteration 19 the loss was 0.17456023395061493.
On iteration 20 the loss was 0.17123663425445557.
On iteration 21 the loss was 0.16805119812488556.
On iteration 22 the loss was 0.1649719476699829.
On iteration 23 the loss was 0.1619787961244583.
On iteration 24 the loss was 0.159060999751091.
On iteration 25 the loss was 0.15621469914913177.
On iteration 26 the loss was 0.15344083309173584.
On iteration 27 the loss was 0.1507432758808136.
On iteration 28 the loss was 0.14812737703323364.
On iteration 29 the loss was 0.14559882879257202.
On iteration 30 the loss was 0.1431628316640854.
On iteration 31 the loss was 0.14082351326942444.
On iteration 32 the loss was 0.13858360052108765.
On iteration 33 the loss was 0.1364443004131317.
On iteration 34 the loss was 0.13440531492233276.
On iteration 35 the loss was 0.13246497511863708.
On iteration 36 the loss was 0.13062037527561188.
On iteration 37 the loss was 0.12886767089366913.
On iteration 38 the loss was 0.12720227241516113.
On iteration 39 the loss was 0.1256190538406372.
On iteration 40 the loss was 0.12411264330148697.
On iteration 41 the loss was 0.1226775273680687.
On iteration 42 the loss was 0.12130825221538544.
On iteration 43 the loss was 0.11999955773353577.
On iteration 44 the loss was 0.11874645203351974.
On iteration 45 the loss was 0.1175442710518837.
On iteration 46 the loss was 0.1163887307047844.
On iteration 47 the loss was 0.11527593433856964.
On iteration 48 the loss was 0.11420238018035889.
On iteration 49 the loss was 0.113164983689785.
On iteration 50 the loss was 0.11216098815202713.
On iteration 51 the loss was 0.11118802428245544.
On iteration 52 the loss was 0.110244020819664.
On iteration 53 the loss was 0.10932720452547073.
On iteration 54 the loss was 0.10843604803085327.
On iteration 55 the loss was 0.10756924003362656.
On iteration 56 the loss was 0.10672564804553986.
```

```
On iteration 57 the loss was 0.10590428858995438.
On iteration 58 the loss was 0.10510428249835968.
On iteration 59 the loss was 0.10432484745979309.
On iteration 60 the loss was 0.1035652607679367.
On iteration 61 the loss was 0.1028248518705368.
On iteration 62 the loss was 0.10210297256708145.
On iteration 63 the loss was 0.1013989970088005.
On iteration 64 the loss was 0.10071231424808502.
On iteration 65 the loss was 0.10004232823848724.
On iteration 66 the loss was 0.09938843548297882.
On iteration 67 the loss was 0.098750039935112.
On iteration 68 the loss was 0.09812657535076141.
On iteration 69 the loss was 0.0975174754858017.
On iteration 70 the loss was 0.09692218154668808.
On iteration 71 the loss was 0.09634017944335938.
On iteration 72 the loss was 0.0957709401845932.
On iteration 73 the loss was 0.09521398693323135.
On iteration 74 the loss was 0.09466885030269623.
On iteration 75 the loss was 0.0941350981593132.
On iteration 76 the loss was 0.09361230581998825.
On iteration 77 the loss was 0.09310009330511093.
On iteration 78 the loss was 0.0925980806350708.
On iteration 79 the loss was 0.092105932533741.
On iteration 80 the loss was 0.09162332117557526.
On iteration 81 the loss was 0.0911499410867691.
On iteration 82 the loss was 0.09068550914525986.
On iteration 83 the loss was 0.09022975713014603.
On iteration 84 the loss was 0.08978241682052612.
On iteration 85 the loss was 0.08934324234724045.
On iteration 86 the loss was 0.0889119952917099.
On iteration 87 the loss was 0.08848845958709717.
On iteration 88 the loss was 0.08807240426540375.
On iteration 89 the loss was 0.08766362816095352.
On iteration 90 the loss was 0.08726193010807037.
On iteration 91 the loss was 0.08686710149049759.
On iteration 92 the loss was 0.08647896349430084.
On iteration 93 the loss was 0.08609732985496521.
On iteration 94 the loss was 0.08572202175855637.
On iteration 95 the loss was 0.08535286784172058.
On iteration 96 the loss was 0.08498968929052353.
On iteration 97 the loss was 0.08463233709335327.
On iteration 98 the loss was 0.08428063988685608.
On iteration 99 the loss was 0.0839344710111618.
On iteration 100 the loss was 0.08359366655349731.
On iteration 101 the loss was 0.08325810730457306.
On iteration 102 the loss was 0.0829276591539383.
On iteration 103 the loss was 0.08260219544172287.
On iteration 104 the loss was 0.08228158950805664.
On iteration 105 the loss was 0.08196573704481125.
On iteration 106 the loss was 0.08165451884269714.
On iteration 107 the loss was 0.08134783059358597.
On iteration 108 the loss was 0.08104556053876877.
On iteration 109 the loss was 0.08074760437011719.
On iteration 110 the loss was 0.08045387268066406.
On iteration 111 the loss was 0.08016426116228104.
On iteration 112 the loss was 0.07987868040800095.
On iteration 113 the loss was 0.07959703356027603.
```

```
On iteration 114 the loss was 0.07931923866271973.
On iteration 115 the loss was 0.07904520630836487.
On iteration 116 the loss was 0.07877486199140549.
On iteration 117 the loss was 0.07850811630487442.
On iteration 118 the loss was 0.0782449021935463.
On iteration 119 the loss was 0.07798514515161514.
On iteration 120 the loss was 0.0777287632226944.
On iteration 121 the loss was 0.07747568935155869.
On iteration 122 the loss was 0.07722586393356323.
On iteration 123 the loss was 0.07697920501232147.
On iteration 124 the loss was 0.07673566788434982.
On iteration 125 the loss was 0.07649517059326172.
On iteration 126 the loss was 0.07625766843557358.
On iteration 127 the loss was 0.07602309435606003.
On iteration 128 the loss was 0.0757913887500763.
On iteration 129 the loss was 0.0755624994635582.
On iteration 130 the loss was 0.07533637434244156.
On iteration 131 the loss was 0.0751129537820816.
On iteration 132 the loss was 0.07489219307899475.
On iteration 133 the loss was 0.07467404007911682.
On iteration 134 the loss was 0.07445844262838364.
On iteration 135 the loss was 0.07424536347389221.
On iteration 136 the loss was 0.07403474301099777.
On iteration 137 the loss was 0.07382655143737793.
On iteration 138 the loss was 0.07362072914838791.
On iteration 139 the loss was 0.07341723889112473.
On iteration 140 the loss was 0.07321605086326599.
On iteration 141 the loss was 0.07301710546016693.
On iteration 142 the loss was 0.07282038033008575.
On iteration 143 the loss was 0.07262582331895828.
On iteration 144 the loss was 0.07243340462446213.
On iteration 145 the loss was 0.0722430869936943.
On iteration 146 the loss was 0.07205483317375183.
On iteration 147 the loss was 0.07186860591173172.
On iteration 148 the loss was 0.07168437540531158.
On iteration 149 the loss was 0.07150211185216904.
On iteration 150 the loss was 0.0713217705488205.
On iteration 151 the loss was 0.07114332914352417.
On iteration 152 the loss was 0.07096675038337708.
On iteration 153 the loss was 0.07079201191663742.
On iteration 154 the loss was 0.07061908394098282.
On iteration 155 the loss was 0.07044792920351028.
On iteration 156 the loss was 0.07027851790189743.
On iteration 157 the loss was 0.07011083513498306.
On iteration 158 the loss was 0.0699448511004448.
On iteration 159 the loss was 0.06978052854537964.
On iteration 160 the loss was 0.0696178525686264.
On iteration 161 the loss was 0.0694567933678627.
On iteration 162 the loss was 0.06929732114076614.
On iteration 163 the loss was 0.06913942098617554.
On iteration 164 the loss was 0.0689830631017685.
On iteration 165 the loss was 0.06882823258638382.
On iteration 166 the loss was 0.06867489218711853.
On iteration 167 the loss was 0.06852303445339203.
On iteration 168 the loss was 0.06837262958288193.
On iteration 169 the loss was 0.06822365522384644.
On iteration 170 the loss was 0.06807609647512436.
```

```
On iteration 171 the loss was 0.06792993098497391.
On iteration 172 the loss was 0.06778512895107269.
On iteration 173 the loss was 0.06764168292284012.
On iteration 174 the loss was 0.0674995705485344.
On iteration 175 the loss was 0.06735877692699432.
On iteration 176 the loss was 0.06721927225589752.
On iteration 177 the loss was 0.0670810416340828.
On iteration 178 the loss was 0.06694407761096954.
On iteration 179 the loss was 0.06680835038423538.
On iteration 180 the loss was 0.06667385250329971.
On iteration 181 the loss was 0.06654055416584015.
On iteration 182 the loss was 0.06640845537185669.
On iteration 183 the loss was 0.06627752631902695.
On iteration 184 the loss was 0.06614775955677032.
On iteration 185 the loss was 0.06601913273334503.
On iteration 186 the loss was 0.06589163839817047.
On iteration 187 the loss was 0.06576525419950485.
On iteration 188 the loss was 0.06563997268676758.
On iteration 189 the loss was 0.06551577150821686.
On iteration 190 the loss was 0.0653926432132721.
On iteration 191 the loss was 0.0652705654501915.
On iteration 192 the loss was 0.06514953076839447.
On iteration 193 the loss was 0.06502953171730042.
On iteration 194 the loss was 0.06491053849458694.
On iteration 195 the loss was 0.06479255855083466.
On iteration 196 the loss was 0.06467556208372116.
On iteration 197 the loss was 0.06455954164266586.
On iteration 198 the loss was 0.06444448977708817.
On iteration 199 the loss was 0.06433038413524628.
On iteration 200 the loss was 0.0642172247171402.
On iteration 201 the loss was 0.06410499662160873.
On iteration 202 the loss was 0.0639936774969101.
On iteration 203 the loss was 0.06388327479362488.
On iteration 204 the loss was 0.0637737587094307.
On iteration 205 the loss was 0.06366513669490814.
On iteration 206 the loss was 0.06355737894773483.
On iteration 207 the loss was 0.06345048546791077.
On iteration 208 the loss was 0.06334444880485535.
On iteration 209 the loss was 0.06323925405740738.
On iteration 210 the loss was 0.06313489377498627.
On iteration 211 the loss was 0.06303134560585022.
On iteration 212 the loss was 0.06292861700057983.
On iteration 213 the loss was 0.06282669305801392.
On iteration 214 the loss was 0.06272556632757187.
On iteration 215 the loss was 0.0626252144575119.
On iteration 216 the loss was 0.06252564489841461.
On iteration 217 the loss was 0.062426839023828506.
On iteration 218 the loss was 0.06232878938317299.
On iteration 219 the loss was 0.06223148852586746.
On iteration 220 the loss was 0.06213492900133133.
On iteration 221 the loss was 0.062039103358983994.
On iteration 222 the loss was 0.06194399669766426.
On iteration 223 the loss was 0.06184960901737213.
On iteration 224 the loss was 0.06175592541694641.
On iteration 225 the loss was 0.0616629458963871.
On iteration 226 the loss was 0.061570651829242706.
On iteration 227 the loss was 0.06147904694080353.
```

```
On iteration 228 the loss was 0.06138811632990837.
On iteration 229 the loss was 0.06129785254597664.
On iteration 230 the loss was 0.06120825186371803.
On iteration 231 the loss was 0.061119306832551956.
On iteration 232 the loss was 0.06103101000189781.
On iteration 233 the loss was 0.060943350195884705.
On iteration 234 the loss was 0.060856323689222336.
On iteration 235 the loss was 0.06076992303133011.
On iteration 236 the loss was 0.060684144496917725.
On iteration 237 the loss was 0.06059897691011429.
On iteration 238 the loss was 0.0605144165456295.
On iteration 239 the loss was 0.060430459678173065.
On iteration 240 the loss was 0.060347091406583786.
On iteration 241 the loss was 0.060264311730861664.
On iteration 242 the loss was 0.0601821169257164.
On iteration 243 the loss was 0.0601004920899868.
On iteration 244 the loss was 0.060019440948963165.
On iteration 245 the loss was 0.0599389523267746.
On iteration 246 the loss was 0.0598590187728405.
On iteration 247 the loss was 0.05977964028716087.
On iteration 248 the loss was 0.05970080569386482.
On iteration 249 the loss was 0.0596225126766205.
On iteration 250 the loss was 0.05954475328326225.
On iteration 251 the loss was 0.05946752801537514.
On iteration 252 the loss was 0.059390824288129807.
On iteration 253 the loss was 0.05931463837623596.
On iteration 254 the loss was 0.059238966554403305.
On iteration 255 the loss was 0.05916380509734154.
On iteration 256 the loss was 0.05908914655447006.
On iteration 257 the loss was 0.05901498720049858.
On iteration 258 the loss was 0.058941323310136795.
On iteration 259 the loss was 0.05886814370751381.
On iteration 260 the loss was 0.05879545211791992.
On iteration 261 the loss was 0.058723241090774536.
On iteration 262 the loss was 0.05865149945020676.
On iteration 263 the loss was 0.05858023464679718.
On iteration 264 the loss was 0.05850943177938461.
On iteration 265 the loss was 0.05843908712267876.
On iteration 266 the loss was 0.05836920440196991.
On iteration 267 the loss was 0.058299772441387177.
On iteration 268 the loss was 0.05823078751564026.
On iteration 269 the loss was 0.05816224589943886.
On iteration 270 the loss was 0.058094143867492676.
On iteration 271 the loss was 0.058026477694511414.
On iteration 272 the loss was 0.05795924365520477.
On iteration 273 the loss was 0.057892438024282455.
On iteration 274 the loss was 0.057826053351163864.
On iteration 275 the loss was 0.057760089635849.
On iteration 276 the loss was 0.05769453942775726.
On iteration 277 the loss was 0.05762940272688866.
On iteration 278 the loss was 0.05756467208266258.
On iteration 279 the loss was 0.05750034749507904.
On iteration 280 the loss was 0.057436421513557434.
On iteration 281 the loss was 0.05737289413809776.
On iteration 282 the loss was 0.05730975791811943.
On iteration 283 the loss was 0.057247012853622437.
On iteration 284 the loss was 0.05718465521931648.
```

```
On iteration 285 the loss was 0.05712267756462097.
On iteration 286 the loss was 0.057061079889535904.
On iteration 287 the loss was 0.05699985846877098.
On iteration 288 the loss was 0.056939005851745605.
On iteration 289 the loss was 0.056878525763750076.
On iteration 290 the loss was 0.056818410754203796.
On iteration 291 the loss was 0.056758660823106766.
On iteration 292 the loss was 0.05669926479458809.
On iteration 293 the loss was 0.05664023011922836.
On iteration 294 the loss was 0.05658154562115669.
On iteration 295 the loss was 0.05652321130037308.
On iteration 296 the loss was 0.05646522715687752.
On iteration 297 the loss was 0.056407585740089417.
On iteration 298 the loss was 0.056350283324718475.
On iteration 299 the loss was 0.056293319910764694.
On iteration 300 the loss was 0.056236691772937775.
On iteration 301 the loss was 0.05618039518594742.
On iteration 302 the loss was 0.056124430149793625.
On iteration 303 the loss was 0.056068792939186096.
On iteration 304 the loss was 0.056013476103544235.
On iteration 305 the loss was 0.05595847964286804.
On iteration 306 the loss was 0.055903807282447815.
On iteration 307 the loss was 0.05584944412112236.
On iteration 308 the loss was 0.055795397609472275.
On iteration 309 the loss was 0.05574166402220726.
On iteration 310 the loss was 0.05568823590874672.
On iteration 311 the loss was 0.05563511326909065.
On iteration 312 the loss was 0.05558229237794876.
On iteration 313 the loss was 0.055529773235321045.
On iteration 314 the loss was 0.055477552115917206.
On iteration 315 the loss was 0.055425625294446945.
On iteration 316 the loss was 0.05537399277091026.
On iteration 317 the loss was 0.05532265082001686.
On iteration 318 the loss was 0.05527159571647644.
On iteration 319 the loss was 0.055220827460289.
On iteration 320 the loss was 0.055170346051454544.
On iteration 321 the loss was 0.05512014403939247.
On iteration 322 the loss was 0.055070217698812485.
On iteration 323 the loss was 0.05502057075500488.
On iteration 324 the loss was 0.05497119948267937.
On iteration 325 the loss was 0.05492210015654564.
On iteration 326 the loss was 0.054873272776603.
On iteration 327 the loss was 0.05482470989227295.
On iteration 328 the loss was 0.05477641522884369.
On iteration 329 the loss was 0.05472838506102562.
On iteration 330 the loss was 0.05468061938881874.
On iteration 331 the loss was 0.05463310703635216.
On iteration 332 the loss was 0.054585859179496765.
On iteration 333 the loss was 0.05453886464238167.
On iteration 334 the loss was 0.054492123425006866.
On iteration 335 the loss was 0.05444563180208206.
On iteration 336 the loss was 0.05439939349889755.
On iteration 337 the loss was 0.05435340106487274.
On iteration 338 the loss was 0.05430765822529793.
On iteration 339 the loss was 0.054262157529592514.
On iteration 340 the loss was 0.0542168989777565.
On iteration 341 the loss was 0.05417187884449959.
```

```
On iteration 342 the loss was 0.054127100855112076.
On iteration 343 the loss was 0.05408255755901337.
On iteration 344 the loss was 0.05403825268149376.
On iteration 345 the loss was 0.053994178771972656.
On iteration 346 the loss was 0.05395033583045006.
On iteration 347 the loss was 0.053906723856925964.
On iteration 348 the loss was 0.05386333912611008.
On iteration 349 the loss was 0.053820181638002396.
On iteration 350 the loss was 0.05377725139260292.
On iteration 351 the loss was 0.053734540939331055.
On iteration 352 the loss was 0.0536920540034771.
On iteration 353 the loss was 0.05364978313446045.
On iteration 354 the loss was 0.05360773578286171.
On iteration 355 the loss was 0.05356590077280998.
On iteration 356 the loss was 0.053524285554885864.
On iteration 357 the loss was 0.05348287895321846.
On iteration 358 the loss was 0.05344168841838837.
On iteration 359 the loss was 0.05340070277452469.
On iteration 360 the loss was 0.05335992947220802.
On iteration 361 the loss was 0.05331936478614807.
On iteration 362 the loss was 0.053279004991054535.
On iteration 363 the loss was 0.053238850086927414.
On iteration 364 the loss was 0.05319889634847641.
On iteration 365 the loss was 0.05315914750099182.
On iteration 366 the loss was 0.05311959609389305.
On iteration 367 the loss was 0.0530802421271801.
On iteration 368 the loss was 0.053041085600852966.
On iteration 369 the loss was 0.05300212651491165.
On iteration 370 the loss was 0.052963364869356155.
On iteration 371 the loss was 0.05292479321360588.
On iteration 372 the loss was 0.05288641154766083.
On iteration 373 the loss was 0.052848223596811295.
On iteration 374 the loss was 0.052810221910476685.
On iteration 375 the loss was 0.052772410213947296.
On iteration 376 the loss was 0.05273478478193283.
On iteration 377 the loss was 0.05269734188914299.
On iteration 378 the loss was 0.05266008526086807.
On iteration 379 the loss was 0.05262301117181778.
On iteration 380 the loss was 0.05258611962199211.
On iteration 381 the loss was 0.05254940688610077.
On iteration 382 the loss was 0.05251287296414375.
On iteration 383 the loss was 0.05247651785612106.
On iteration 384 the loss was 0.0524403378367424.
On iteration 385 the loss was 0.05240433290600777.
On iteration 386 the loss was 0.05236850306391716.
On iteration 387 the loss was 0.05233284458518028.
On iteration 388 the loss was 0.05229736119508743.
On iteration 389 the loss was 0.052262045443058014.
On iteration 390 the loss was 0.052226901054382324.
On iteration 391 the loss was 0.052191924303770065.
On iteration 392 the loss was 0.05215711519122124.
On iteration 393 the loss was 0.05212246999144554.
On iteration 394 the loss was 0.052087992429733276.
On iteration 395 the loss was 0.052053675055503845.
On iteration 396 the loss was 0.052019525319337845.
On iteration 397 the loss was 0.05198553577065468.
On iteration 398 the loss was 0.051951706409454346.
```

```
On iteration 399 the loss was 0.05191803723573685.
On iteration 400 the loss was 0.051884524524211884.
On iteration 401 the loss was 0.051851172000169754.
On iteration 402 the loss was 0.05181797221302986.
On iteration 403 the loss was 0.0517849326133728.
On iteration 404 the loss was 0.05175204575061798.
On iteration 405 the loss was 0.051719311624765396.
On iteration 406 the loss was 0.05168673023581505.
On iteration 407 the loss was 0.05165430158376694.
On iteration 408 the loss was 0.051622021943330765.
On iteration 409 the loss was 0.05158989503979683.
On iteration 410 the loss was 0.051557913422584534.
On iteration 411 the loss was 0.05152608081698418.
On iteration 412 the loss was 0.05149439349770546.
On iteration 413 the loss was 0.05146285519003868.
On iteration 414 the loss was 0.051431458443403244.
On iteration 415 the loss was 0.05140020698308945.
On iteration 416 the loss was 0.05136909708380699.
On iteration 417 the loss was 0.051338132470846176.
On iteration 418 the loss was 0.051307305693626404.
On iteration 419 the loss was 0.05127662420272827.
On iteration 420 the loss was 0.051246076822280884.
On iteration 421 the loss was 0.05121567100286484.
On iteration 422 the loss was 0.051185403019189835.
On iteration 423 the loss was 0.051155272871255875.
On iteration 424 the loss was 0.05112527683377266.
On iteration 425 the loss was 0.05109541490674019.
On iteration 426 the loss was 0.05106569081544876.
On iteration 427 the loss was 0.05103609710931778.
On iteration 428 the loss was 0.05100663751363754.
On iteration 429 the loss was 0.05097731202840805.
On iteration 430 the loss was 0.050948113203048706.
On iteration 431 the loss was 0.050919048488140106.
On iteration 432 the loss was 0.050890110433101654.
On iteration 433 the loss was 0.05086130276322365.
On iteration 434 the loss was 0.05083262547850609.
On iteration 435 the loss was 0.05080407112836838.
On iteration 436 the loss was 0.05077564716339111.
On iteration 437 the loss was 0.0507473461329937.
On iteration 438 the loss was 0.05071917176246643.
On iteration 439 the loss was 0.05069112032651901.
On iteration 440 the loss was 0.050663191825151443.
On iteration 441 the loss was 0.050635386258363724.
On iteration 442 the loss was 0.05060770362615585.
On iteration 443 the loss was 0.050580140203237534.
On iteration 444 the loss was 0.05055269971489906.
On iteration 445 the loss was 0.05052537843585014.
On iteration 446 the loss was 0.050498172640800476.
On iteration 447 the loss was 0.05047108978033066.
On iteration 448 the loss was 0.05044412240386009.
On iteration 449 the loss was 0.05041727423667908.
On iteration 450 the loss was 0.050390541553497314.
On iteration 451 the loss was 0.050363924354314804.
On iteration 452 the loss was 0.05033741891384125.
On iteration 453 the loss was 0.05031103268265724.
On iteration 454 the loss was 0.05028475821018219.
On iteration 455 the loss was 0.05025859549641609.
```

```
On iteration 456 the loss was 0.05023254454135895.
On iteration 457 the loss was 0.050206609070301056.
On iteration 458 the loss was 0.05018078163266182.
On iteration 459 the loss was 0.05015506595373154.
On iteration 460 the loss was 0.05012945830821991.
On iteration 461 the loss was 0.050103962421417236.
On iteration 462 the loss was 0.05007857084274292.
On iteration 463 the loss was 0.05005329102277756.
On iteration 464 the loss was 0.05002811551094055.
On iteration 465 the loss was 0.0500030480325222.
On iteration 466 the loss was 0.04997808858752251.
On iteration 467 the loss was 0.04995323345065117.
On iteration 468 the loss was 0.04992848262190819.
On iteration 469 the loss was 0.049903836101293564.
On iteration 470 the loss was 0.049879290163517.
On iteration 471 the loss was 0.04985485225915909.
On iteration 472 the loss was 0.049830514937639236.
On iteration 473 the loss was 0.04980627819895744.
On iteration 474 the loss was 0.04978214576840401.
On iteration 475 the loss was 0.04975811019539833.
On iteration 476 the loss was 0.04973417893052101.
On iteration 477 the loss was 0.04971034452319145.
On iteration 478 the loss was 0.04968661069869995.
On iteration 479 the loss was 0.04966297373175621.
On iteration 480 the loss was 0.04963943734765053.
On iteration 481 the loss was 0.04961599409580231.
On iteration 482 the loss was 0.049592651426792145.
On iteration 483 the loss was 0.04956940561532974.
On iteration 484 the loss was 0.0495462529361248.
On iteration 485 the loss was 0.04952319711446762.
On iteration 486 the loss was 0.0495002381503582.
On iteration 487 the loss was 0.04947737231850624.
On iteration 488 the loss was 0.04945459961891174.
On iteration 489 the loss was 0.04943192005157471.
On iteration 490 the loss was 0.04940933361649513.
On iteration 491 the loss was 0.04938683658838272.
On iteration 492 the loss was 0.04936443641781807.
On iteration 493 the loss was 0.04934212565422058.
On iteration 494 the loss was 0.049319904297590256.
On iteration 495 the loss was 0.049297772347927094.
On iteration 496 the loss was 0.04927573353052139.
On iteration 497 the loss was 0.049253784120082855.
On iteration 498 the loss was 0.04923192039132118.
On iteration 499 the loss was 0.04921014606952667.
On iteration 500 the loss was 0.049188461154699326.
On iteration 501 the loss was 0.04916686192154884.
On iteration 502 the loss was 0.049145352095365524.
On iteration 503 the loss was 0.04912392795085907.
On iteration 504 the loss was 0.04910258948802948.
On iteration 505 the loss was 0.049081336706876755.
On iteration 506 the loss was 0.049060165882110596.
On iteration 507 the loss was 0.0490390844643116.
On iteration 508 the loss was 0.04901808500289917.
On iteration 509 the loss was 0.048997171223163605.
On iteration 510 the loss was 0.048976339399814606.
On iteration 511 the loss was 0.04895558953285217.
On iteration 512 the loss was 0.048934925347566605.
```

```
On iteration 513 the loss was 0.048914339393377304.
On iteration 514 the loss was 0.04889383912086487.
On iteration 515 the loss was 0.0488734170794487.
On iteration 516 the loss was 0.0488530769944191.
On iteration 517 the loss was 0.048832815140485764.
On iteration 518 the loss was 0.048812635242938995.
On iteration 519 the loss was 0.04879253730177879.
On iteration 520 the loss was 0.04877251386642456.
On iteration 521 the loss was 0.048752572387456894.
On iteration 522 the loss was 0.048732709139585495.
On iteration 523 the loss was 0.048712924122810364.
On iteration 524 the loss was 0.0486932136118412.
On iteration 525 the loss was 0.04867358133196831.
On iteration 526 the loss was 0.04865402728319168.
On iteration 527 the loss was 0.04863455146551132.
On iteration 528 the loss was 0.048615146428346634.
On iteration 529 the loss was 0.04859582334756851.
On iteration 530 the loss was 0.04857657104730606.
On iteration 531 the loss was 0.04855739325284958.
On iteration 532 the loss was 0.048538293689489365.
On iteration 533 the loss was 0.0485192649066448.
On iteration 534 the loss was 0.0485003106296062 5.
On iteration 535 the loss was 0.04848143085837364.
On iteration 536 the loss was 0.04846262186765671.
On iteration 537 the loss was 0.04844388738274574.
On iteration 538 the loss was 0.04842522367835045.
On iteration 539 the loss was 0.048406634479761124.
On iteration 540 the loss was 0.04838811233639717.
On iteration 541 the loss was 0.04836966469883919.
On iteration 542 the loss was 0.048351287841796875.
On iteration 543 the loss was 0.04833298176527023.
On iteration 544 the loss was 0.048314742743968964.
On iteration 545 the loss was 0.04829657822847366.
On iteration 546 the loss was 0.048278480768203735.
On iteration 547 the loss was 0.04826045036315918.
On iteration 548 the loss was 0.048242490738630295.
On iteration 549 the loss was 0.04822460189461708.
On iteration 550 the loss was 0.04820677638053894.
On iteration 551 the loss was 0.04818902164697647.
On iteration 552 the loss was 0.048171333968639374.
On iteration 553 the loss was 0.04815371334552765.
On iteration 554 the loss was 0.048136159777641296.
On iteration 555 the loss was 0.048118673264980316.
On iteration 556 the loss was 0.04810125008225441.
On iteration 557 the loss was 0.048083893954753876.
On iteration 558 the loss was 0.048066604882478714.
On iteration 559 the loss was 0.048049379140138626.
On iteration 560 the loss was 0.04803222045302391.
On iteration 561 the loss was 0.04801512509584427.
On iteration 562 the loss was 0.0479980930685997.
On iteration 563 the loss was 0.04798112437129021.
On iteration 564 the loss was 0.047964222729206085.
On iteration 565 the loss was 0.04794738069176674.
On iteration 566 the loss was 0.047930605709552765.
On iteration 567 the loss was 0.047913890331983566.
On iteration 568 the loss was 0.04789723828434944.
On iteration 569 the loss was 0.04788064584136009.
```

```
On iteration 570 the loss was 0.047864120453596115.
On iteration 571 the loss was 0.047847650945186615.
On iteration 572 the loss was 0.04783124476671219.
On iteration 573 the loss was 0.047814901918172836.
On iteration 574 the loss was 0.04779861867427826.
On iteration 575 the loss was 0.04778239503502846.
On iteration 576 the loss was 0.04776623100042343.
On iteration 577 the loss was 0.04775012657046318.
On iteration 578 the loss was 0.047734085470438004.
On iteration 579 the loss was 0.0477181002497673.
On iteration 580 the loss was 0.04770217463374138.
On iteration 581 the loss was 0.04768630489706993.
On iteration 582 the loss was 0.04767049849033356.
On iteration 583 the loss was 0.04765474796295166.
On iteration 584 the loss was 0.04763905331492424.
On iteration 585 the loss was 0.047623418271541595.
On iteration 586 the loss was 0.047607842832803726.
On iteration 587 the loss was 0.047592319548130035.
On iteration 588 the loss was 0.04757685586810112.
On iteration 589 the loss was 0.04756144806742668.
On iteration 590 the loss was 0.04754609987139702.
On iteration 591 the loss was 0.047530803829431534.
On iteration 592 the loss was 0.047515563666820526.
On iteration 593 the loss was 0.047500379383563995.
On iteration 594 the loss was 0.04748525470495224.
On iteration 595 the loss was 0.047470178455114365.
On iteration 596 the loss was 0.047455161809921265.
On iteration 597 the loss was 0.04744019731879234.
On iteration 598 the loss was 0.0474252887070179.
On iteration 599 the loss was 0.04741043224930763.
On iteration 600 the loss was 0.047395627945661545.
On iteration 601 the loss was 0.047380879521369934.
On iteration 602 the loss was 0.0473661869764328.
On iteration 603 the loss was 0.047351542860269547.
On iteration 604 the loss was 0.04733695462346077.
On iteration 605 the loss was 0.04732241854071617.
On iteration 606 the loss was 0.04730793461203575.
On iteration 607 the loss was 0.04729350283741951.
On iteration 608 the loss was 0.04727911949157715.
On iteration 609 the loss was 0.047264792025089264.
On iteration 610 the loss was 0.04725051298737526.
On iteration 611 the loss was 0.04723628982901573.
On iteration 612 the loss was 0.047222115099430084.
On iteration 613 the loss was 0.04720798879861832.
On iteration 614 the loss was 0.04719391465187073.
On iteration 615 the loss was 0.04717989265918732.
On iteration 616 the loss was 0.04716591536998749.
On iteration 617 the loss was 0.047151993960142136.
On iteration 618 the loss was 0.047138117253780365.
On iteration 619 the loss was 0.04712429270148277.
On iteration 620 the loss was 0.04711051657795906.
On iteration 621 the loss was 0.04709679260849953.
On iteration 622 the loss was 0.047083113342523575.
On iteration 623 the loss was 0.0470694825053215.
On iteration 624 the loss was 0.04705590009689331.
On iteration 625 the loss was 0.0470423698425293.
On iteration 626 the loss was 0.047028884291648865.
```

```
On iteration 627 the loss was 0.047015443444252014.
On iteration 628 the loss was 0.04700205475091934.
On iteration 629 the loss was 0.04698871076107025.
On iteration 630 the loss was 0.04697541519999504.
On iteration 631 the loss was 0.04696216806769371.
On iteration 632 the loss was 0.04694896563887596.
On iteration 633 the loss was 0.046935807913541794.
On iteration 634 the loss was 0.046922698616981506.
On iteration 635 the loss was 0.0469096340239048.
On iteration 636 the loss was 0.046896614134311676.
On iteration 637 the loss was 0.04688364267349243.
On iteration 638 the loss was 0.04687071591615677.
On iteration 639 the loss was 0.04685783386230469.
On iteration 640 the loss was 0.04684499651193619.
On iteration 641 the loss was 0.04683220759034157.
On iteration 642 the loss was 0.04681945964694023.
On iteration 643 the loss was 0.046806756407022476.
On iteration 644 the loss was 0.0467940978705883.
On iteration 645 the loss was 0.04678148403763771.
On iteration 646 the loss was 0.0467689111828804.
On iteration 647 the loss was 0.04675638675689697.
On iteration 648 the loss was 0.04674390330910683.
On iteration 649 the loss was 0.046731460839509964.
On iteration 650 the loss was 0.04671906679868698.
On iteration 651 the loss was 0.04670671001076698.
On iteration 652 the loss was 0.046694401651620865.
On iteration 653 the loss was 0.04668213054537773.
On iteration 654 the loss was 0.04666990414261818.
On iteration 655 the loss was 0.04665772244334221.
On iteration 656 the loss was 0.04664557799696922.
On iteration 657 the loss was 0.04663347825407982.
On iteration 658 the loss was 0.0466214194893837.
On iteration 659 the loss was 0.04660940542817116.
On iteration 660 the loss was 0.0465974286198616.
On iteration 661 the loss was 0.04658549651503563.
On iteration 662 the loss was 0.04657360166311264.
On iteration 663 the loss was 0.046561747789382935.
On iteration 664 the loss was 0.04654993861913681.
On iteration 665 the loss was 0.04653816670179367.
On iteration 666 the loss was 0.046526435762643814.
On iteration 667 the loss was 0.04651474580168724.
On iteration 668 the loss was 0.04650309309363365.
On iteration 669 the loss was 0.046491481363773346.
On iteration 670 the loss was 0.04647991061210632.
On iteration 671 the loss was 0.046468380838632584.
On iteration 672 the loss was 0.04645688831806183.
On iteration 673 the loss was 0.04644543305039406.
On iteration 674 the loss was 0.04643401876091957.
On iteration 675 the loss was 0.04642264544963837.
On iteration 676 the loss was 0.04641130939126015.
On iteration 677 the loss was 0.04640001058578491.
On iteration 678 the loss was 0.04638874903321266.
On iteration 679 the loss was 0.046377528458833694.
On iteration 680 the loss was 0.04636634513735771.
On iteration 681 the loss was 0.046355199068784714.
On iteration 682 the loss was 0.0463440902531147.
On iteration 683 the loss was 0.04633302241563797.
```

```
On iteration 684 the loss was 0.046321988105773926.
On iteration 685 the loss was 0.046310991048812866.
On iteration 686 the loss was 0.04630003497004509.
On iteration 687 the loss was 0.04628911241889.
On iteration 688 the loss was 0.046278227120637894.
On iteration 689 the loss was 0.04626737907528877.
On iteration 690 the loss was 0.046256568282842636.
On iteration 691 the loss was 0.046245791018009186.
On iteration 692 the loss was 0.04623505473136902.
On iteration 693 the loss was 0.04622434824705124.
On iteration 694 the loss was 0.04621368274092674.
On iteration 695 the loss was 0.04620305076241493.
On iteration 696 the loss was 0.04619245603680611.
On iteration 697 the loss was 0.04618189483880997.
On iteration 698 the loss was 0.04617137089371681.
On iteration 699 the loss was 0.04616088047623634.
On iteration 700 the loss was 0.04615042731165886.
On iteration 701 the loss was 0.04614000767469406.
On iteration 702 the loss was 0.04612962156534195.
On iteration 703 the loss was 0.04611927270889282.
On iteration 704 the loss was 0.04610895738005638.
On iteration 705 the loss was 0.046098675578832626.
On iteration 706 the loss was 0.046088431030511856.
On iteration 707 the loss was 0.046078216284513474.
On iteration 708 the loss was 0.046068038791418076.
On iteration 709 the loss was 0.046057891100645065.
On iteration 710 the loss was 0.04604778066277504.
On iteration 711 the loss was 0.0460377037525177.
On iteration 712 the loss was 0.04602766036987305.
On iteration 713 the loss was 0.04601764678955078.
On iteration 714 the loss was 0.0460076704621315.
On iteration 715 the loss was 0.04599772393703461.
On iteration 716 the loss was 0.0459878146648407.
On iteration 717 the loss was 0.04597793519496918.
On iteration 718 the loss was 0.04596808925271034.
On iteration 719 the loss was 0.045958273112773895.
On iteration 720 the loss was 0.045948490500450134.
On iteration 721 the loss was 0.04593874141573906.
On iteration 722 the loss was 0.04592902585864067.
On iteration 723 the loss was 0.04591934010386467.
On iteration 724 the loss was 0.045909687876701355.
On iteration 725 the loss was 0.04590006545186043.
On iteration 726 the loss was 0.04589047655463219.
On iteration 727 the loss was 0.045880917459726334.
On iteration 728 the loss was 0.04587138816714287.
On iteration 729 the loss was 0.04586189240217209.
On iteration 730 the loss was 0.045852430164813995.
On iteration 731 the loss was 0.04584299400448799.
On iteration 732 the loss was 0.04583359137177467.
On iteration 733 the loss was 0.04582421854138374.
On iteration 734 the loss was 0.0458148792386055.
On iteration 735 the loss was 0.045805566012859344.
On iteration 736 the loss was 0.045796286314725876.
On iteration 737 the loss was 0.045787036418914795.
On iteration 738 the loss was 0.0457778163254261.
On iteration 739 the loss was 0.045768626034259796.
On iteration 740 the loss was 0.04575946554541588.
```

```
On iteration 741 the loss was 0.04575033858418465.
On iteration 742 the loss was 0.045741237699985504.
On iteration 743 the loss was 0.04573216661810875.
On iteration 744 the loss was 0.04572312533855438.
On iteration 745 the loss was 0.0457141138613224.
On iteration 746 the loss was 0.04570512846112251.
On iteration 747 the loss was 0.04569617658853531.
On iteration 748 the loss was 0.045687250792980194.
On iteration 749 the loss was 0.04567835479974747.
On iteration 750 the loss was 0.04566948860883713.
On iteration 751 the loss was 0.045660652220249176.
On iteration 752 the loss was 0.045651841908693314.
On iteration 753 the loss was 0.04564306139945984.
On iteration 754 the loss was 0.04563430696725845.
On iteration 755 the loss was 0.045625582337379456.
On iteration 756 the loss was 0.045616887509822845.
On iteration 757 the loss was 0.045608218759298325.
On iteration 758 the loss was 0.04559957981109619.
On iteration 759 the loss was 0.04559096693992615.
On iteration 760 the loss was 0.04558238387107849.
On iteration 761 the loss was 0.045573826879262924.
On iteration 762 the loss was 0.045565295964479446.
On iteration 763 the loss was 0.045556794852018356.
On iteration 764 the loss was 0.045548319816589355.
On iteration 765 the loss was 0.04553987458348274.
On iteration 766 the loss was 0.04553145170211792.
On iteration 767 the loss was 0.045523058623075485.
On iteration 768 the loss was 0.04551469162106514.
On iteration 769 the loss was 0.04550635442137718.
On iteration 770 the loss was 0.045498039573431015.
On iteration 771 the loss was 0.045489754527807236.
On iteration 772 the loss was 0.045481495559215546.
On iteration 773 the loss was 0.045473262667655945.
On iteration 774 the loss was 0.04546505585312843.
On iteration 775 the loss was 0.04545687511563301.
On iteration 776 the loss was 0.04544872045516968.
On iteration 777 the loss was 0.045440591871738434.
On iteration 778 the loss was 0.04543248936533928.
On iteration 779 the loss was 0.045424412935972214.
On iteration 780 the loss was 0.04541635885834694.
On iteration 781 the loss was 0.04540833458304405.
On iteration 782 the loss was 0.045400336384773254.
On iteration 783 the loss was 0.04539236053824425.
On iteration 784 the loss was 0.04538441076874733.
On iteration 785 the loss was 0.0453764870762825.
On iteration 786 the loss was 0.04536858946084976.
On iteration 787 the loss was 0.045360714197158813.
On iteration 788 the loss was 0.045352865010499954.
On iteration 789 the loss was 0.045345041900873184.
On iteration 790 the loss was 0.0453372448682785.
On iteration 791 the loss was 0.04532947018742561.
On iteration 792 the loss was 0.045321717858314514.
On iteration 793 the loss was 0.0453139953315258.
On iteration 794 the loss was 0.04530629515647888.
On iteration 795 the loss was 0.04529861733317375.
On iteration 796 the loss was 0.04529096558690071.
On iteration 797 the loss was 0.04528333619236946.
```

```
On iteration 798 the loss was 0.0452757328748703.
On iteration 799 the loss was 0.04526815190911293.
On iteration 800 the loss was 0.04526059702038765.
On iteration 801 the loss was 0.04525306448340416.
On iteration 802 the loss was 0.04524555802345276.
On iteration 803 the loss was 0.04523807018995285.
On iteration 804 the loss was 0.04523061215877533.
On iteration 805 the loss was 0.0452231727540493.
On iteration 806 the loss was 0.04521575942635536.
On iteration 807 the loss was 0.045208368450403214.
On iteration 808 the loss was 0.045200999826192856.
On iteration 809 the loss was 0.04519365355372429.
On iteration 810 the loss was 0.04518633335828781.
On iteration 811 the loss was 0.045179035514593124.
On iteration 812 the loss was 0.04517175629734993.
On iteration 813 the loss was 0.045164503157138824.
On iteration 814 the loss was 0.04515727609395981.
On iteration 815 the loss was 0.045150067657232285.
On iteration 816 the loss was 0.04514288157224655.
On iteration 817 the loss was 0.04513571783900261.
On iteration 818 the loss was 0.04512857645750046.
On iteration 819 the loss was 0.045121461153030396.
On iteration 820 the loss was 0.045114364475011826.
On iteration 821 the loss was 0.045107290148735046.
On iteration 822 the loss was 0.04510023817420006.
On iteration 823 the loss was 0.04509320855140686.
On iteration 824 the loss was 0.045086201280355453.
On iteration 825 the loss was 0.04507921636104584.
On iteration 826 the loss was 0.045072250068187714.
On iteration 827 the loss was 0.04506530985236168.
On iteration 828 the loss was 0.04505838826298714.
On iteration 829 the loss was 0.045051489025354385.
On iteration 830 the loss was 0.045044608414173126.
On iteration 831 the loss was 0.045037753880023956.
On iteration 832 the loss was 0.04503091797232628.
On iteration 833 the loss was 0.04502410441637039.
On iteration 834 the loss was 0.045017309486866.
On iteration 835 the loss was 0.04501054063439369.
On iteration 836 the loss was 0.04500378668308258.
On iteration 837 the loss was 0.04499705880880356.
On iteration 838 the loss was 0.04499034956097603.
On iteration 839 the loss was 0.04498365893959999.
On iteration 840 the loss was 0.04497699439525604.
On iteration 841 the loss was 0.04497034475207329.
On iteration 842 the loss was 0.044963717460632324.
On iteration 843 the loss was 0.04495711252093315.
On iteration 844 the loss was 0.04495052620768547.
On iteration 845 the loss was 0.04494396224617958.
On iteration 846 the loss was 0.04493741691112518.
On iteration 847 the loss was 0.04493089020252228.
On iteration 848 the loss was 0.04492438584566116.
On iteration 849 the loss was 0.04491790384054184.
On iteration 850 the loss was 0.04491143673658371.
On iteration 851 the loss was 0.04490499198436737.
On iteration 852 the loss was 0.044898565858602524.
On iteration 853 the loss was 0.04489216208457947.
On iteration 854 the loss was 0.044885776937007904.
```

```
On iteration 855 the loss was 0.04487941041588783.
On iteration 856 the loss was 0.044873062521219254.
On iteration 857 the loss was 0.044866736978292465.
On iteration 858 the loss was 0.04486043006181717.
On iteration 859 the loss was 0.044854141771793365.
On iteration 860 the loss was 0.044847872108221054.
On iteration 861 the loss was 0.044841621071100235.
On iteration 862 the loss was 0.04483539238572121.
On iteration 863 the loss was 0.04482918232679367.
On iteration 864 the loss was 0.04482298716902733.
On iteration 865 the loss was 0.04481681436300278.
On iteration 866 the loss was 0.04481066018342972.
On iteration 867 the loss was 0.04480452463030815.
On iteration 868 the loss was 0.04479840770363808.
On iteration 869 the loss was 0.044792309403419495.
On iteration 870 the loss was 0.044786229729652405.
On iteration 871 the loss was 0.04478016868233681.
On iteration 872 the loss was 0.0447741262614727.
On iteration 873 the loss was 0.04476810246706009.
On iteration 874 the loss was 0.04476209357380867.
On iteration 875 the loss was 0.04475610703229904.
On iteration 876 the loss was 0.044750139117240906.
On iteration 877 the loss was 0.044744186103343964.
On iteration 878 the loss was 0.04473825544118881.
On iteration 879 the loss was 0.044732339680194855.
On iteration 880 the loss was 0.04472644254565239.
On iteration 881 the loss was 0.04472056403756142.
On iteration 882 the loss was 0.04471470043063164.
On iteration 883 the loss was 0.04470885917544365.
On iteration 884 the loss was 0.044703032821416855.
On iteration 885 the loss was 0.04469722509384155.
On iteration 886 the loss was 0.04469143599271774.
On iteration 887 the loss was 0.04468566179275513.
On iteration 888 the loss was 0.044679906219244.
On iteration 889 the loss was 0.04467416927218437.
On iteration 890 the loss was 0.044668447226285934.
On iteration 891 the loss was 0.04466274380683899.
On iteration 892 the loss was 0.044657059013843536.
On iteration 893 the loss was 0.04465138912200928.
On iteration 894 the loss was 0.04464573785662651.
On iteration 895 the loss was 0.044640105217695236.
On iteration 896 the loss was 0.044634487479925156.
On iteration 897 the loss was 0.04462888464331627.
On iteration 898 the loss was 0.04462330415844917.
On iteration 899 the loss was 0.04461773484945297.
On iteration 900 the loss was 0.04461218789219856.
On iteration 901 the loss was 0.04460665211081505.
On iteration 902 the loss was 0.044601134955883026.
On iteration 903 the loss was 0.044595636427402496.
On iteration 904 the loss was 0.04459015280008316.
On iteration 905 the loss was 0.04458468779921532.
On iteration 906 the loss was 0.04457923769950867.
On iteration 907 the loss was 0.04457380250096321.
On iteration 908 the loss was 0.04456838592886925.
On iteration 909 the loss was 0.04456298425793648.
On iteration 910 the loss was 0.0445575974881649.
On iteration 911 the loss was 0.04455222934484482.
```

```
On iteration 912 the loss was 0.04454687610268593.
On iteration 913 the loss was 0.04454154148697853.
On iteration 914 the loss was 0.04453622177243233.
On iteration 915 the loss was 0.04453091695904732.
On iteration 916 the loss was 0.0445256270468235.
On iteration 917 the loss was 0.04452035576105118.
On iteration 918 the loss was 0.04451509937644005.
On iteration 919 the loss was 0.04450985789299011.
On iteration 920 the loss was 0.04450463131070137.
On iteration 921 the loss was 0.04449941962957382.
On iteration 922 the loss was 0.044494226574897766.
On iteration 923 the loss was 0.044489048421382904.
On iteration 924 the loss was 0.044483885169029236.
On iteration 925 the loss was 0.04447873681783676.
On iteration 926 the loss was 0.04447360336780548.
On iteration 927 the loss was 0.04446848854422569.
On iteration 928 the loss was 0.0444633848965168.
On iteration 929 the loss was 0.0444582998752594.
On iteration 930 the loss was 0.044453226029872894.
On iteration 931 the loss was 0.04444817081093788.
On iteration 932 the loss was 0.04444313049316406.
On iteration 933 the loss was 0.04443810507655144.
On iteration 934 the loss was 0.044433094561100006.
On iteration 935 the loss was 0.04442809522151947.
On iteration 936 the loss was 0.04442311450839043.
On iteration 937 the loss was 0.04441814869642258.
On iteration 938 the loss was 0.04441319778561592.
On iteration 939 the loss was 0.04440826177597046.
On iteration 940 the loss was 0.04440333694219589.
On iteration 941 the loss was 0.04439843073487282.
On iteration 942 the loss was 0.04439353570342064.
On iteration 943 the loss was 0.04438865929841995.
On iteration 944 the loss was 0.04438379406929016.
On iteration 945 the loss was 0.044378943741321564.
On iteration 946 the loss was 0.04437410831451416.
On iteration 947 the loss was 0.04436928778886795.
On iteration 948 the loss was 0.044364482164382935.
On iteration 949 the loss was 0.04435969144105911.
On iteration 950 the loss was 0.044354911893606186.
On iteration 951 the loss was 0.04435014724731445.
On iteration 952 the loss was 0.044345397502183914.
On iteration 953 the loss was 0.04434066265821457.
On iteration 954 the loss was 0.04433594271540642.
On iteration 955 the loss was 0.04433123394846916.
On iteration 956 the loss was 0.0443265400826931.
On iteration 957 the loss was 0.04432186111807823.
On iteration 958 the loss was 0.04431719332933426.
On iteration 959 the loss was 0.04431254044175148.
On iteration 960 the loss was 0.044307902455329895.
On iteration 961 the loss was 0.044303279370069504.
On iteration 962 the loss was 0.04429866746068001.
On iteration 963 the loss was 0.044294070452451706.
On iteration 964 the loss was 0.0442894846200943.
On iteration 965 the loss was 0.044284917414188385.
On iteration 966 the loss was 0.04428035765886307.
On iteration 967 the loss was 0.04427581652998924.
On iteration 968 the loss was 0.04427128657698631.
```

```
On iteration 969 the loss was 0.04426676779985428.
On iteration 970 the loss was 0.04426226392388344.
On iteration 971 the loss was 0.04425777494907379.
On iteration 972 the loss was 0.04425329715013504.
On iteration 973 the loss was 0.04424883425235748.
On iteration 974 the loss was 0.04424438625574112.
On iteration 975 the loss was 0.04423994570970535.
On iteration 976 the loss was 0.04423552379012108.
On iteration 977 the loss was 0.0442311130464077.
On iteration 978 the loss was 0.044226713478565216.
On iteration 979 the loss was 0.044222328811883926.
On iteration 980 the loss was 0.04421795532107353.
On iteration 981 the loss was 0.04421359673142433.
On iteration 982 the loss was 0.04420924931764603.
On iteration 983 the loss was 0.04420491307973862.
On iteration 984 the loss was 0.0442005917429924.
On iteration 985 the loss was 0.04419628530740738.
On iteration 986 the loss was 0.044191986322402954.
On iteration 987 the loss was 0.04418770223855972.
On iteration 988 the loss was 0.044183433055877686.
On iteration 989 the loss was 0.044179175049066544.
On iteration 990 the loss was 0.0441749282181263.
On iteration 991 the loss was 0.044170696288347244.
On iteration 992 the loss was 0.04416647180914879.
On iteration 993 the loss was 0.044162265956401825.
On iteration 994 the loss was 0.04415806755423546.
On iteration 995 the loss was 0.044153884053230286.
On iteration 996 the loss was 0.04414971172809601.
On iteration 997 the loss was 0.044145550578832626.
On iteration 998 the loss was 0.04414140433073044.
On iteration 999 the loss was 0.044137269258499146.
```

In [132]:

```
# TODO: run the following section of code to define the reconstruction functio
n
def linear_autoencoder_reconstruction_function(z):
    return linear_decoder(torch.FloatTensor(z[np.newaxis, :])).detach()[0, :]
```

In [133]:
```
# TODO: fill in this section to accomplish the following.
# 1) select two different rows from the data matrix X
# 2) compute the hidden representation for each row using your linear_encoder
# 3) call the function latent_interpolation and generate a visualization with
 height 32 and width 32
# BEGIN YOUR CODE
z_one = linear_encoder(torch.FloatTensor(X[0, :]))
z_two = linear_encoder(torch.FloatTensor(X[1, :]))
latent_interpolation(z_one, z_two, linear_autoencoder_reconstruction_function,
32, 32)
# END YOUR CODE
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
```

```
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
```

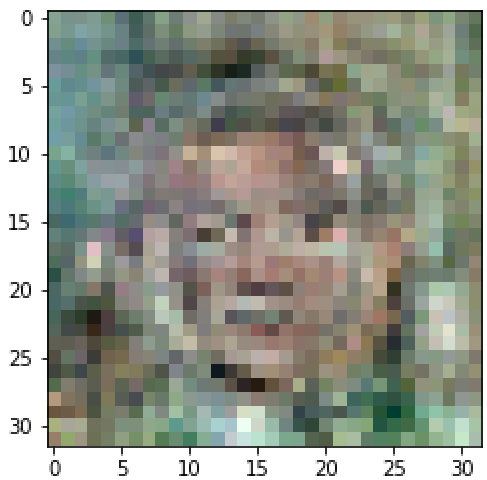
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for f
loats or [0..255] for integers).
```

0:00 / 0:12

Comment on what happens during the interpolation process:

This interpolation allows much more crossover for the two images than the previous interpolation process and can be seen in how the images morph into one another.

In [145]:
```
# TODO: fill in this section to accomplish the following.
# 1) compute the hidden representation of each row of the data matrix
# 2) compute z_mean as the average of the hidden representation matrix along t
he 0th axis
# 3) define z_stddev to be the rank Q identity matrix for now
# 4) generate an new image by calling latent_generation with height 32 and wid
th 32
# BEGIN YOUR CODE
for i in range(256):
    A[i, :] = linear_encoder(torch.FloatTensor(X[i, :])).detach().numpy()
z_mean = np.mean(A, axis=0)
z_stddev = np.eye(Q)
latent_generation(z_mean, z_stddev, linear_autoencoder_reconstruction_function
, 32, 32)
# END YOUR CODE
```



Comment on how real the generated face looks:

The generated face looks similar to the results rendered from PCA, but the images blend more together in this generated face.

Comment on how the linear autoencoder compares to PCA:

The effect is similar to that of PCA, so I looked into the background of linear autoencoders and saw that linear autocoders are the same as PCAs.

Section Four: Convolutional Autoencoder

In this section, you will learn about the convolutional autoencoder, and you will also implement the convolutional autoencoder using pytorch. We shall use your convolutional autoencoder to interpolate between data points from X and to also generate new samples of faces.

In [117]:
```
class ConvolutionalEncoder(torch.nn.Module):

    def __init__(self, image_height, image_width, final_size):
        """Creates a deep convolutional neural network.
        Args:
            image_height: an integer, the height of each image
            image_width: an integer, the width of each image
            final_size: an integer, the depth of the final layer of this netwo
rk
        """
        super(ConvolutionalEncoder, self).__init__()
        self.image_height = image_height
        self.image_width = image_width
        self.final_size = max(16 * 3, final_size)
        # TODO: fill in this section to accomplish the following.
        # 1) create a 5 layer convolutional neural network that transforms an
 image with shape
        #    [image_height, image_width, 3] to a vector with final_size dimens
ions.
        #    HINT: consider the class torch.nn.Conv2d with stride=2
        # BEGIN YOUR CODE
        self.weight1 = torch.nn.ConvTranspose2d(image_height * image_width * 3
, final_size, 1, stride=2)
        self.weight2 = torch.nn.ConvTranspose2d(image_height * image_width * 3
, final_size, 2, stride=2)
        self.weight3 = torch.nn.ConvTranspose2d(image_height * image_width * 3
, final_size, 3, stride=2)
        self.weight4 = torch.nn.ConvTranspose2d(image_height * image_width * 3
, final_size, 4, stride=2)
        self.weight5 = torch.nn.ConvTranspose2d(image_height * image_width * 3
, final_size, 5, stride=2)
        self.Sigmoid = torch.nn.Sigmoid()
        # END YOUR CODE

    def forward(self, x):
        """Computes a single forward pass of this network.
        Args:
            x: a float32 tensor with shape [batch_size, D]
        Returns:
            a float32 tensor with shape [batch_size, final_size]
        """
        x = x.view(x.size()[0], self.image_height, self.image_width, 3)
        x = torch.transpose(x, 1, 3)
        # TODO: fill in this section to accomplish the following.
        # 1) perform a forward pass using the conv layers you defined
        # 2) apply any activation function you want
        # BEGIN YOUR CODE
        x = self.weight1(x)
        x = self.weight2(x)
        x = self.weight3(x)
        x = self.weight4(x)
        x = self.weight5(x)
        self.Sigmoid(x)
        # END YOUR CODE
        x = torch.transpose(x, 1, 3)
        x = x.contiguous()
```

```
        x = x.view(x.size()[0], self.final_size)
        return x
```

In [118]:
```
class ConvolutionalDecoder(torch.nn.Module):

    def __init__(self, image_height, image_width, final_size):
        """Creates a deep convolutional neural network.
        Args:
            image_height: an integer, the height of each image
            image_width: an integer, the width of each image
            final_size: an integer, the depth of the final layer of this netwo
rk
        """
        super(ConvolutionalDecoder, self).__init__()
        self.image_height = image_height
        self.image_width = image_width
        self.final_size = max(16 * 3, final_size)
        # TODO: fill in this section to accomplish the following.
        # 1) create a 5 layer transpose convolutional neural network that tran
sforms a vector with
        #    final_size dimensions to an image with shape [image_height, image
_width, 3]
        #    HINT: consider the class torch.nn.ConvTranspose2d with stride=2
        # BEGIN YOUR CODE
        self.weight1 = torch.nn.ConvTranspose2d(final_size, image_height * ima
ge_width * 3, 1, stride=2)
        self.weight2 = torch.nn.ConvTranspose2d(final_size, image_height * ima
ge_width * 3, 2, stride=2)
        self.weight3 = torch.nn.ConvTranspose2d(final_size, image_height * ima
ge_width * 3, 3, stride=2)
        self.weight4 = torch.nn.ConvTranspose2d(final_size, image_height * ima
ge_width * 3, 4, stride=2)
        self.weight5 = torch.nn.ConvTranspose2d(final_size, image_height * ima
ge_width * 3, 5, stride=2)
        self.Sigmoid = torch.nn.Sigmoid()
        # END YOUR CODE

    def forward(self, x):
        """Computes a single forward pass of this network.
        Args:
            x: a float32 tensor with shape [batch_size, final_size]
        Returns:
            a float32 tensor with shape [batch_size, D]
        """
        x = x.view(x.size()[0], 1, 1, self.final_size)
        x = torch.transpose(x, 1, 3)
        # TODO: fill in this section to accomplish the following.
        # 1) perform a forward pass using the conv layers you defined
        # 2) apply any activation function you want
        # BEGIN YOUR CODE
        x = self.weight1(x)
        x = self.weight2(x)
        x = self.weight3(x)
        x = self.weight4(x)
        x = self.weight5(x)
        self.Sigmoid(x)
        # END YOUR CODE
        x = torch.transpose(x, 1, 3)
        x = x.contiguous()
```

```
        x = x.view(x.size()[0], self.image_height * self.image_width * 3)
        return x
```

In [119]:

```
# TODO: fill in this section to accomplish the following.
# 1) create an instance of ConvolutionalEncoder named convolutional_encoder wi
th height 32, width 32, and hidden size Q
# 1) create an instance of ConvolutionalDecoder named convolutional_decoder wi
th height 32, width 32, and hidden size Q
# 2) assign convolutional_autoencoder_loss to be an instance of torch.nn.MSELo
ss
# 2) create an optimizer named convolutional_autoencoder_optimizer of your cho
osing with a learning rate of your choosing.
#    HINT: consider the torch.optim.Adam object
# BEGIN YOUR CODE
convolutional_encoder = ConvolutionalEncoder(32, 32, Q)
convolutional_decoder = ConvolutionalDecoder(32, 32, Q)
convolutional_autoencoder_loss = torch.nn.MSELoss()
convolutional_autoencoder_optimizer = torch.optim.Adam(convolutional_encoder.p
arameters(), lr=0.001)
# END YOUR CODE
```

In [120]:
```
# TODO: run the following section of code in order to train the model
# Construct a tensor from the dataset
image_tensor = torch.FloatTensor(X)
for i in range(1000):
    # Clear the previous gradient from the optimizer by calling .zero_grad()
    convolutional_autoencoder_optimizer.zero_grad()
    # Compute a full encoding and decoding step
    reconstructed_image = convolutional_decoder(convolutional_encoder(image_te
nsor))
    # Compute the mean squared reconstruction loss
    loss = convolutional_autoencoder_loss(reconstructed_image, image_tensor)
    # Pass the loss backward throgh the network, and compute the gradients
    loss.backward()
    # Update the optimizer by calling .step()
    convolutional_autoencoder_optimizer.step()
    # Return a detatched value of the loss for logging purposes
    print("On iteration {0} the loss was {1}.".format(i, loss.detach()))
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-120-b5225a39863c> in <module>
      6     convolutional_autoencoder_optimizer.zero_grad()
      7     # Compute a full encoding and decoding step
----> 8     reconstructed_image = convolutional_decoder(convolutional_encoder
(image_tensor))
      9     # Compute the mean squared reconstruction loss
     10     loss = convolutional_autoencoder_loss(reconstructed_image, image_
tensor)

~\Anaconda3\lib\site-packages\torch\nn\modules\module.py in __call__(self, *i
nput, **kwargs)
    487             result = self._slow_forward(*input, **kwargs)
    488         else:
--> 489             result = self.forward(*input, **kwargs)
    490         for hook in self._forward_hooks.values():
    491             hook_result = hook(self, input, result)

<ipython-input-117-76da2bbab851> in forward(self, x)
     38         # 2) apply any activation function you want
     39         # BEGIN YOUR CODE
---> 40         x = self.weight1(x)
     41         x = self.weight2(x)
     42         x = self.weight3(x)

~\Anaconda3\lib\site-packages\torch\nn\modules\module.py in __call__(self, *i
nput, **kwargs)
    487             result = self._slow_forward(*input, **kwargs)
    488         else:
--> 489             result = self.forward(*input, **kwargs)
    490         for hook in self._forward_hooks.values():
    491             hook_result = hook(self, input, result)

~\Anaconda3\lib\site-packages\torch\nn\modules\conv.py in forward(self, inpu
t, output_size)
    755         return F.conv_transpose2d(
    756             input, self.weight, self.bias, self.stride, self.padding,
--> 757             output_padding, self.groups, self.dilation)
    758 
    759 

RuntimeError: Given transposed=1, weight of size [3072, 256, 1, 1], expected
 input[5000, 3, 32, 32] to have 3072 channels, but got 3 channels instead
```

In [31]:

```
# TODO: run the following section of code to define the reconstruction functio
n
def convolutional_autoencoder_reconstruction_function(z):
    return np.asarray(convolutional_decoder(torch.FloatTensor(z[np.newaxis,
:])).detach()[0, :], np.float32)
```

In [121]:
```
# TODO: fill in this section to accomplish the following.
# 1) select two different rows from the data matrix X
# 2) compute the hidden representation for each row using your convolutional_e
ncoder
# 3) call the function latent_interpolation and generate a visualization with
 height 32 and width 32
# BEGIN YOUR CODE
z_one = np.asarray(convolutional_encoder(torch.FloatTensor(X[0, :])).detach(),
np.float32)
z_two = np.asarray(convolutional_encoder(torch.FloatTensor(X[1, :])).detach(),
np.float32)
latent_interpolation(z_one, z_two, convolutional_autoencoder_reconstruction_fu
nction, 32, 32)
# END YOUR CODE
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-121-efe94cd22fa0> in <module>
      4 # 3) call the function latent_interpolation and generate a visualizat
ion with height 32 and width 32
      5 # BEGIN YOUR CODE
----> 6 z_one = np.asarray(convolutional_encoder(torch.FloatTensor(X[0, :])).
detach(), np.float32)
      7 z_two = np.asarray(convolutional_encoder(torch.FloatTensor(X[1, :])).
detach(), np.float32)
      8 latent_interpolation(z_one, z_two, convolutional_autoencoder_reconstr
uction_function, 32, 32)

~\Anaconda3\lib\site-packages\torch\nn\modules\module.py in __call__(self, *i
nput, **kwargs)
    487             result = self._slow_forward(*input, **kwargs)
    488         else:
--> 489             result = self.forward(*input, **kwargs)
    490         for hook in self._forward_hooks.values():
    491             hook_result = hook(self, input, result)

<ipython-input-117-76da2bbab851> in forward(self, x)
     32             a float32 tensor with shape [batch_size, final_size]
     33         """
---> 34         x = x.view(x.size()[0], self.image_height, self.image_width,
3)
     35         x = torch.transpose(x, 1, 3)
     36         # TODO: fill in this section to accomplish the following.

RuntimeError: shape '[3072, 32, 32, 3]' is invalid for input of size 3072
```

Comment on what happens during the interpolation process:

[TODO: your response here]

In [123]:
```
# TODO: fill in this section to accomplish the following.
# 1) compute the hidden representation of each row of the data matrix using co
nvolutional_encoder
# 2) compute z_mean as the average of the hidden representation matrix along t
he 0th axis
# 3) define z_stddev to be the rank Q identity matrix for now
# 4) generate an new image by calling latent_generation with height 32 and wid
th 32
# BEGIN YOUR CODE
for i in range(3072):
    A[i, :] = convolutional_encoder(X[i, :])
z_mean = np.mean(A, axis=0)
z_stddev = np.eye(Q)
latent_generation(z_mean, z_stddev, convolutional_autoencoder_reconstruction_f
unction, 32, 32)
# END YOUR CODE
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-123-304324302455> in <module>
      6 # BEGIN YOUR CODE
      7 for i in range(3072):
----> 8     A[i, :] = convolutional_encoder(X[i, :])
      9 z_mean = np.mean(A, axis=0)
     10 z_stddev = np.eye(Q)

~\Anaconda3\lib\site-packages\torch\nn\modules\module.py in __call__(self, *i
nput, **kwargs)
    487             result = self._slow_forward(*input, **kwargs)
    488         else:
--> 489             result = self.forward(*input, **kwargs)
    490         for hook in self._forward_hooks.values():
    491             hook_result = hook(self, input, result)

<ipython-input-117-76da2bbab851> in forward(self, x)
     32             a float32 tensor with shape [batch_size, final_size]
     33         """
---> 34         x = x.view(x.size()[0], self.image_height, self.image_width,
3)
     35         x = torch.transpose(x, 1, 3)
     36         # TODO: fill in this section to accomplish the following.

TypeError: 'int' object is not callable
```

Comment on how real the generated face looks:

[TODO: your response here]

Comment on how the convolutional autoencoder compares to the linear autoencoder and PCA:

[TODO: your response here]

(Optional) Section Five: Variational Autoencoder

In this section, we implement the Variational Autoencoder, an extension for the traditional autoencoder that explicitly models the probability distribution of a latent variable. This section is optional, and so we fill in the code for you. If you have extra time after completing the rest of this homework, you should first read this tutorial on variational inference [https://arxiv.org/pdf/1606.05908.pdf (https://arxiv.org/pdf/1606.05908.pdf)](https://arxiv.org/pdf/1606.05908.pdf). Then, you may attempt to train the VAE given below.

In [41]:
```
# TODO: run the following section of code
class Sampler(torch.nn.Module):

    def __init__(self, hidden_size):
        """Creates a Variational sampling layer.
        Args:
            hidden_size: an integer, the number of neurons in the sampling lay
er.
        """
        super(Sampler, self).__init__()
        self.hidden_size = hidden_size
        self.log_scale = torch.nn.Linear(hidden_size, hidden_size)
        self.shift = torch.nn.Linear(hidden_size, hidden_size)

    def forward(self, x):
        """Computes a single forward pass of this network.
        Args:
            x: a float32 tensor with shape [batch_size, hidden_size]
        Returns:
            a float32 tensor with shape [batch_size, hidden_size]
        """
        scale = torch.exp(self.log_scale(x))
        shift = self.shift(x)
        sample = torch.randn([self.hidden_size]) * scale + shift
        return sample

    def kl_penalty(self, x):
        """Computes a single forward pass of this network.
        Args:
            x: a float32 tensor with shape [batch_size, hidden_size]
        Returns:
            a float32 scalar: KL divergence between this distribution and the
 standard normal distribution.
        """
        log_scale = self.log_scale(x)
        scale = torch.exp(log_scale)
        shift = self.shift(x)
        return torch.mean(log_scale + (1.0 + shift * shift) / (2.0 * scale * s
cale) - 0.5)
```

In [42]:
```
# TODO: run the following section of code
variational_encoder = ConvolutionalEncoder(32, 32, 256)
variational_decoder = ConvolutionalDecoder(32, 32, 256)
sampler = Sampler(256)
variational_autoencoder_loss = torch.nn.MSELoss()
variational_autoencoder_optimizer = torch.optim.Adam([
    {"params": variational_encoder.parameters()},
    {"params": variational_decoder.parameters()},
    {"params": sampler.parameters()}])
```

In [43]:
```
# TODO: run the following section of code in order to train the model
# Construct a tensor from the dataset
image_tensor = torch.FloatTensor(X)
for i in range(10000):
    # Clear the previous gradient from the optimizer by calling .zero_grad()
    variational_autoencoder_optimizer.zero_grad()
    # Compute a full encoding and decoding step
    hidden_variables = variational_encoder(image_tensor)
    reconstructed_image = variational_decoder(sampler(hidden_variables))
    # Compute the mean squared reconstruction loss
    loss = variational_autoencoder_loss(reconstructed_image, image_tensor) - s
ampler.kl_penalty(hidden_variables)
    # Pass the loss backward throgh the network, and compute the gradients
    loss.backward()
    # Update the optimizer by calling .step()
    variational_autoencoder_optimizer.step()
    # Return a detatched value of the loss for logging purposes
    print("On iteration {0} the loss was {1}.".format(i, loss.detach()))
```

```
On iteration 0 the loss was 0.3778478801250458.
On iteration 1 the loss was 0.3771253228187561.
On iteration 2 the loss was 0.37535417079925537.
On iteration 3 the loss was 0.36859118938446045.
On iteration 4 the loss was 0.33781716227531433.

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-43-532ebb744c62> in <module>
      7     # Compute a full encoding and decoding step
      8     hidden_variables = variational_encoder(image_tensor)
----> 9     reconstructed_image = variational_decoder(sampler(hidden_variable
s))
     10     # Compute the mean squared reconstruction loss
     11     loss = variational_autoencoder_loss(reconstructed_image, image_te
nsor) - sampler.kl_penalty(hidden_variables)

c:\program files\python36\lib\site-packages\torch\nn\modules\module.py in __c
all__(self, *input, **kwargs)
    487             result = self._slow_forward(*input, **kwargs)
    488         else:
--> 489             result = self.forward(*input, **kwargs)
    490         for hook in self._forward_hooks.values():
    491             hook_result = hook(self, input, result)

<ipython-input-28-f4a07bb67861> in forward(self, x)
     45         x = F.relu(self.conv2(x))
     46         x = F.relu(self.conv3(x))
---> 47         x = F.relu(self.conv4(x))
     48         x = F.relu(self.conv5(x))
     49         # END YOUR CODE

c:\program files\python36\lib\site-packages\torch\nn\modules\module.py in __c
all__(self, *input, **kwargs)
    487             result = self._slow_forward(*input, **kwargs)
    488         else:
--> 489             result = self.forward(*input, **kwargs)
    490         for hook in self._forward_hooks.values():
    491             hook_result = hook(self, input, result)

c:\program files\python36\lib\site-packages\torch\nn\modules\conv.py in forwa
rd(self, input, output_size)
    755         return F.conv_transpose2d(
    756             input, self.weight, self.bias, self.stride, self.padding,
--> 757             output_padding, self.groups, self.dilation)
    758 
    759 

KeyboardInterrupt:
```

In [44]:

```
# TODO: run the following section of code
def variational_autoencoder_reconstruction_function(z):
    return np.asarray(variational_decoder(torch.FloatTensor(z[np.newaxis, :]))
.detach()[0, :], np.float32)
```

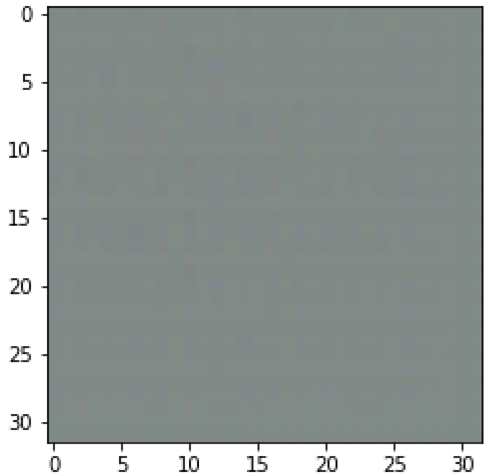
In [45]:
```
# TODO: run the following section of code
x_one = X[0, :]
x_two = X[1, :]
z_one = np.asarray(sampler(variational_encoder(torch.FloatTensor(x_one[np.newa
xis, :]))).detach(), np.float32)
z_two = np.asarray(sampler(variational_encoder(torch.FloatTensor(x_two[np.newa
xis, :]))).detach(), np.float32)
latent_interpolation(z_one, z_two, convolutional_autoencoder_reconstruction_fu
nction, 32, 32)
```

0:00 / 0:12

Comment on what happens during the interpolation process:

[TODO: your response here]

In [46]:
```
# TODO: run the following section of code
z_mean = np.asarray(torch.mean(sampler(variational_encoder(torch.FloatTensor(X
))), 0).detach(), np.float32)
z_stddev = np.identity(Q)
latent_generation(z_mean, z_stddev, convolutional_autoencoder_reconstruction_f
unction, 32, 32)
```



Comment on how real the generated face looks:

[TODO: your response here]

Comment on how the convolutional autoencoder compares to the linear autoencoder and PCA:

[TODO: your response here]

In [ ]:
```
```