Autoencoders and Image Manipulation

In the previous chapters, we have learned about classifying images, detecting objects in an image, and segmenting the pixels corresponding to objects in images. In this chapter, we will learn about representing an image in a lower dimension using autoencoders and leveraging the lower-dimensional representation of an image to generate new images by using variational autoencoders. Learning to represent images in a lower number of dimensions helps us manipulate (modify) the images to a considerable degree. We will learn about leveraging lower-dimensional representations to generate new images as well as novel images that are based on the content and style of two different images. Next, we will also learn about modifying images in such a way that the image is visually unaltered, however, the class corresponding to the image is changed from one to another. Finally, we will learn about generating deep fakes: given a source image of person A, we generate a target image of person B with a similar facial expression as that of person A.

Overall, we will go through the following topics in this chapter:

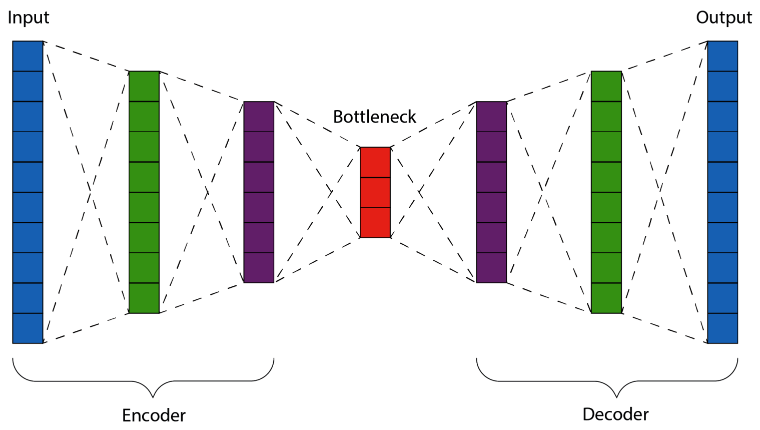
- Understanding and implementing autoencoders
- Understanding convolutional autoencoders
- Understanding variational autoencoders
- Performing an adversarial attack on images
- Performing neural style transfer
- Generating deep fakes

Understanding autoencoders

So far, in the previous chapters, we have learned about classifying images by training a model based on the input image and its corresponding label. Now let's imagine a scenario where we need to cluster images based on their similarity and with the constraint of not having their corresponding labels. Autoencoders come in handy to identify and group similar images.

An autoencoder takes an image as input, stores it in a lower dimension, and tries to reproduce the same image as output, hence the term **auto** (which stands for being able to reproduce the input). However, if we just reproduce the input in the output, we would not need a network, but a simple multiplication of the input by 1 would do. The differentiating aspect of an autoencoder is that it encodes the information present in an image in a lower dimension and then reproduces the image, hence the term **encoder** (which stands for representing the information of an image in a lower dimension). This way, images that are similar will have similar encoding. Further, the **decoder** works towards reconstructing the original image from the encoded vector.

In order to further understand autoencoders, let's take a look at the following diagram:



Let's say the input image is a flattened version of the MNIST handwritten digits and the output image is the same as what is provided as input. The middlemost layer is the layer of encoding called the **bottleneck** layer. The operations happening between the input and the bottleneck layer represent the **encoder** and the operations between the bottleneck layer and output represent the **decoder**.

Through the bottleneck layer, we can represent an image in a much lower dimension. Furthermore, with the bottleneck layer, we can reconstruct the original image. We leverage the bottleneck layer to solve the problems of identifying similar images as well as generating new images, which we will learn how to do in subsequent sections.

The bottleneck layer helps in the following ways:

- Images that have similar bottleneck layer values (encoded representations) are likely to be similar to each other.
- By changing the node values of the bottleneck layer, we can change the output image.

With the preceding understanding, let's do the following:

- Implement autoencoders from scratch
- Visualize the similarity of images based on bottleneck layer values

In the next section, we will learn about how autoencoders are built and also will learn about the impact of different units in the bottleneck layer on the decoder's output.

Implementing vanilla autoencoders

To understand how to build an autoencoder, let's implement one on the MNIST dataset, which contains images of handwritten digits:

The following code is available as `simple_auto_encoder_with_different_latent_size.ipynb` in the `chapter11` folder of this book's GitHub repository - https://tinyurl.com/mcvp-packt The code is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Import the relevant packages and define the device:

```
!pip install -q torch_snippets
from torch_snippets import *
from torchvision.datasets import MNIST
from torchvision import transforms
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Specify the transformation that we want our images to pass through:

```
img_transform = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize([0.5], [0.5]),
                    transforms.Lambda(lambda x: x.to(device))
                ])
```

In the preceding code, we see that we are converting an image into a tensor, normalizing it, and then passing it to the device.

3. Create the train and validation datasets:

```
trn_ds = MNIST('/content/', transform=img_transform, \
               train=True, download=True)
val_ds = MNIST('/content/', transform=img_transform, \
               train=False, download=True)
```

4. Define the dataloaders:

```
batch_size = 256
trn_dl = DataLoader(trn_ds, batch_size=batch_size, \
                    shuffle=True)
val_dl = DataLoader(val_ds, batch_size=batch_size, \
                    shuffle=False)
```

5. Define the network architecture. We define the `AutoEncoder` class constituting the encoder and decoder in the `__init__` method, along with the dimension of the bottleneck layer, `latent_dim`, and the `for-ward` method, and visualize the model summary:

- Define the `AutoEncoder` class and the `__init__` method containing the encoder, decoder, and the dimension of the bottleneck layer:

```
class AutoEncoder(nn.Module):
    def __init__(self, latent_dim):
        super().__init__()
        self.latend_dim = latent_dim
        self.encoder = nn.Sequential(
                            nn.Linear(28 * 28, 128),
                            nn.ReLU(True),
                            nn.Linear(128, 64),
                            nn.ReLU(True),
                            nn.Linear(64, latent_dim))
        self.decoder = nn.Sequential(
                            nn.Linear(latent_dim, 64),
                            nn.ReLU(True),
                            nn.Linear(64, 128),
                            nn.ReLU(True),
                            nn.Linear(128, 28 * 28),
                            nn.Tanh())
```

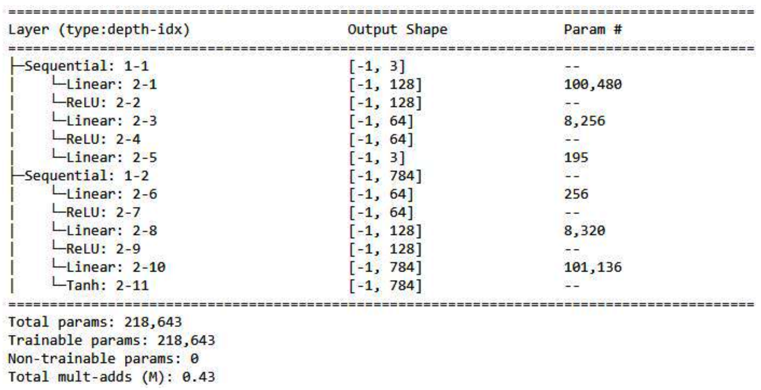
- Define the `forward` method:

```
    def forward(self, x):
        x = x.view(len(x), -1)
        x = self.encoder(x)
        x = self.decoder(x)
        x = x.view(len(x), 1, 28, 28)
        return x
```

- Visualize the preceding model:

```
!pip install torch_summary
from torchsummary import summary
model = AutoEncoder(3).to(device)
summary(model, torch.zeros(2,1,28,28))
```

This results in the following output:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 3]                   --
|    └─Linear: 2-1                       [-1, 128]                 100,480
|    └─ReLU: 2-2                         [-1, 128]                 --
|    └─Linear: 2-3                       [-1, 64]                  8,256
|    └─ReLU: 2-4                         [-1, 64]                  --
|    └─Linear: 2-5                       [-1, 3]                   195
├─Sequential: 1-2                        [-1, 784]                 --
|    └─Linear: 2-6                       [-1, 64]                  256
|    └─ReLU: 2-7                         [-1, 64]                  --
|    └─Linear: 2-8                       [-1, 128]                 8,320
|    └─ReLU: 2-9                         [-1, 128]                 --
|    └─Linear: 2-10                      [-1, 784]                 101,136
|    └─Tanh: 2-11                        [-1, 784]                 --
==========================================================================================
Total params: 218,643
Trainable params: 218,643
Non-trainable params: 0
Total mult-adds (M): 0.43
```

From the preceding output, we can see that the `Linear: 2-5 layer` is the bottleneck layer, where each image is represented as a 3-dimensional vector. Furthermore, the decoder layer reconstructs the original image using the three values in the bottleneck layer.

6. Define a function to train on a batch of data (`train_batch`), just like we did in the previous chapters:

```
def train_batch(input, model, criterion, optimizer):
    model.train()
    optimizer.zero_grad()
    output = model(input)
    loss = criterion(output, input)
    loss.backward()
    optimizer.step()
    return loss
```

7. Define the function to validate on the batch of data (`validate_batch`):

```
@torch.no_grad()
def validate_batch(input, model, criterion):
    model.eval()
    output = model(input)
    loss = criterion(output, input)
    return loss
```

8. Define the model, loss criterion, and optimizer:

```
model = AutoEncoder(3).to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.AdamW(model.parameters(), \
                              lr=0.001, weight_decay=1e-5)
```

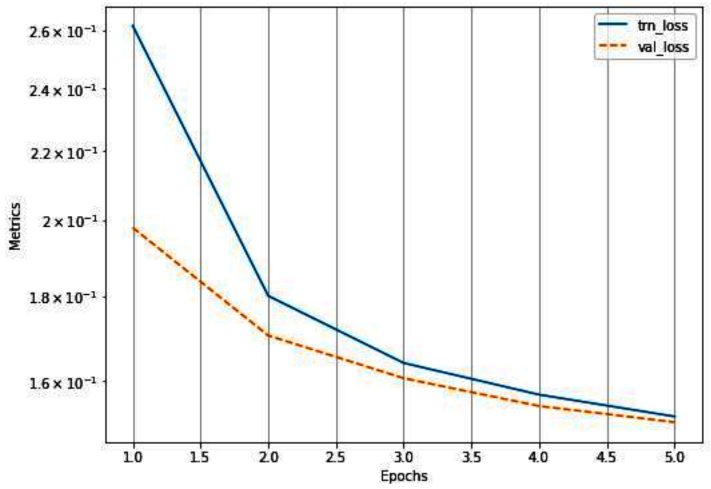
9. Train the model over increasing epochs:

```
num_epochs = 5
log = Report(num_epochs)

for epoch in range(num_epochs):
    N = len(trn_dl)
    for ix, (data, _) in enumerate(trn_dl):
        loss = train_batch(data, model, criterion, optimizer)
        log.record(pos=(epoch + (ix+1)/N), \
                   trn_loss=loss, end='\r')

    N = len(val_dl)
    for ix, (data, _) in enumerate(val_dl):
        loss = validate_batch(data, model, criterion)
        log.record(pos=(epoch + (ix+1)/N), \
                        val_loss=loss, end='\r')

        log.report_avgs(epoch+1)
```

10. Visualize the training and validation loss over increasing epochs:

```
log.plot_epochs(log=True)
```

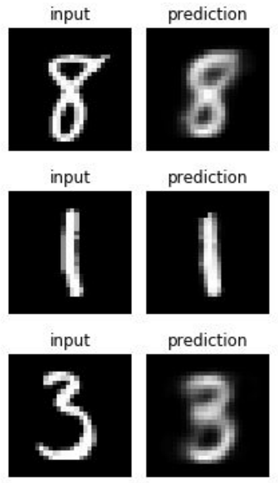
The preceding snippet returns the following output:



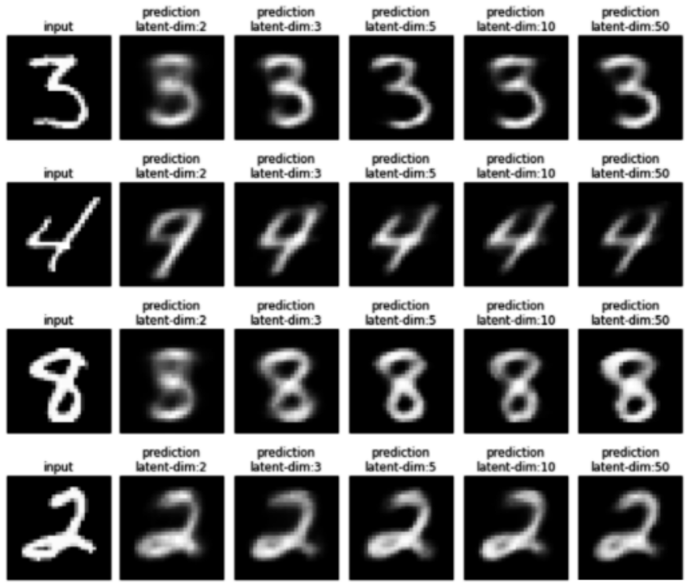
11. Validate the model on the `val_ds` dataset, which was not provided during training:

```
for _ in range(3):
    ix = np.random.randint(len(val_ds))
    im, _ = val_ds[ix]
    _im = model(im[None])[0]
    fig, ax = plt.subplots(1, 2, figsize=(3,3))
    show(im[0], ax=ax[0], title='input')
    show(_im[0], ax=ax[1], title='prediction')
    plt.tight_layout()
    plt.show()
```

The output of the preceding code is as follows:



We can see that the network can reproduce input with a very high level of accuracy even though the bottleneck layer is only three dimensions in size. However, the images are not as clear as we expect them to be. This is primarily because of the small number of nodes in the bottleneck layer. In the following image, we will visualize the reconstructed images after training networks with different bottleneck layer sizes - 2, 3, 5, 10, and 50:



It is clear that as the number of vectors in the bottleneck layer increased, the clarity of the reconstructed image improved.

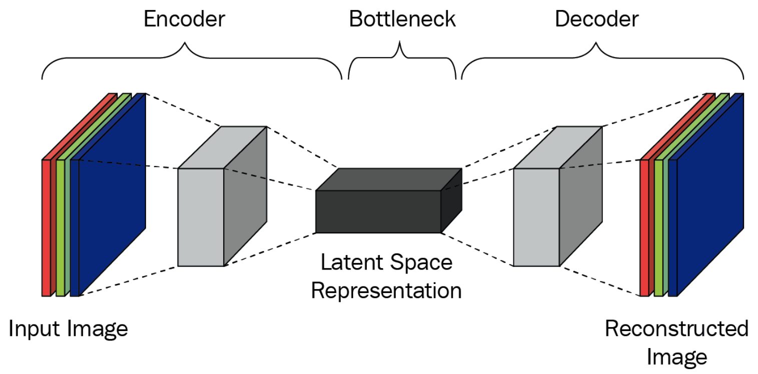
In the next section, we will learn about generating more clear images using a **convolutional neural network** (**CNN**) and we will learn about grouping similar images.

Understanding convolutional autoencoders

In the previous section, we learned about autoencoders and implemented them in PyTorch. While we have implemented them, one convenience that we had through the dataset was that each image has only 1 channel (each image was represented as a black and white image) and the images are relatively small (28 x 28). Hence the network flattened the input and was able to train on 784 (28*28) input values to predict 784 output values. However, in reality, we will encounter images that have 3 channels and are much bigger than a 28 x 28 image.

In this section, we will learn about implementing a convolutional autoencoder that is able to work on multi-dimensional input images. However, for the purpose of comparison with vanilla autoencoders, we will work on the same MNIST dataset that we worked on in the previous section, but modify the network in such a way that we now build a convolutional autoencoder and not a vanilla autoencoder.

A convolutional autoencoder is represented as follows:



From the preceding image, we can see that the input image is represented as a block in the bottleneck layer that is used to reconstruct the image. The image goes through multiple convolutions to fetch the bottleneck representation (which is the **Bottleneck layer** that is obtained by passing through **Encoder**) and the bottleneck representation is up-scaled to fetch the original image (the original image is reconstructed by passing through the **decoder**).

Now that we know how a convolutional autoencoder is represented, let's implement it in the following code:

Given that the majority of the code is similar to the code in the previous section, we have only provided the additional code for brevity. The following code is available as `conv_auto_encoder.ipynb` in `Chapter11` folder of this book's GitHub repository. We encourage you to go through the notebook in GitHub if you want to see the complete code.

1. Steps 1 to 4, which are exactly the same as in the vanilla autoencoder section, are as follows:

```
!pip install -q torch_snippets
from torch_snippets import *
from torchvision.datasets import MNIST
from torchvision import transforms
device = 'cuda' if torch.cuda.is_available() else 'cpu'
img_transform = transforms.Compose([
                    transforms.ToTensor(),
                    transforms.Normalize([0.5], [0.5]),
                    transforms.Lambda(lambda x: x.to(device))
                                    ])

trn_ds = MNIST('/content/', transform=img_transform, \
               train=True, download=True)
val_ds = MNIST('/content/', transform=img_transform, \
               train=False, download=True)

batch_size = 128
trn_dl = DataLoader(trn_ds, batch_size=batch_size, \
                    shuffle=True)
val_dl = DataLoader(val_ds, batch_size=batch_size, \
                    shuffle=False)
```

2. Define the class of neural network, `ConvAutoEncoder`:

- Define the class and the `__init__` method:

```
class ConvAutoEncoder(nn.Module):
    def __init__(self):
        super().__init__()
```

- Define the `encoder` architecture:

```
        self.encoder = nn.Sequential(
                            nn.Conv2d(1, 32, 3, stride=3, \
                                      padding=1),
                            nn.ReLU(True),
                            nn.MaxPool2d(2, stride=2),
                            nn.Conv2d(32, 64, 3, stride=2, \
                                      padding=1),
                            nn.ReLU(True),
                            nn.MaxPool2d(2, stride=1)
                        )
```

Note that in the preceding code, we started with the initial number of channels, which is `1`, and increased it to `32`, and then further increased it to `64` while reducing the size of the output values by performing `nn.MaxPool2d` and `nn.Conv2d` operations.

- Define the `decoder` architecture:

```
        self.decoder = nn.Sequential(
                        nn.ConvTranspose2d(64, 32, 3, \
                                           stride=2),
                        nn.ReLU(True),
                        nn.ConvTranspose2d(32, 16, 5, \
                                         stride=3,padding=1),
                        nn.ReLU(True),
                        nn.ConvTranspose2d(16, 1, 2, \
                                         stride=2,padding=1),
                        nn.Tanh()
                    )
```

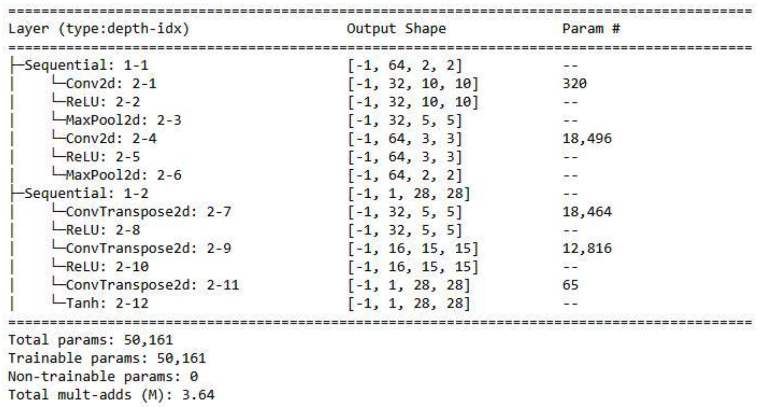
- Define the `forward` method:

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

3. Get the summary of the model using the `summary` method:

```
model = ConvAutoEncoder().to(device)
!pip install torch_summary
from torchsummary import summary
summary(model, torch.zeros(2,1,28,28));
```

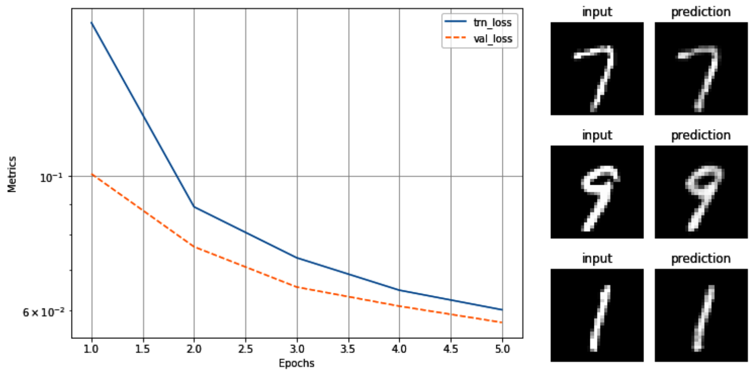
The preceding code results in the following output:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 64, 2, 2]            --
|    └─Conv2d: 2-1                       [-1, 32, 10, 10]          320
|    └─ReLU: 2-2                         [-1, 32, 10, 10]          --
|    └─MaxPool2d: 2-3                    [-1, 32, 5, 5]            --
|    └─Conv2d: 2-4                       [-1, 64, 3, 3]            18,496
|    └─ReLU: 2-5                         [-1, 64, 3, 3]            --
|    └─MaxPool2d: 2-6                    [-1, 64, 2, 2]            --
├─Sequential: 1-2                        [-1, 1, 28, 28]           --
|    └─ConvTranspose2d: 2-7              [-1, 32, 5, 5]            18,464
|    └─ReLU: 2-8                         [-1, 32, 5, 5]            --
|    └─ConvTranspose2d: 2-9              [-1, 16, 15, 15]          12,816
|    └─ReLU: 2-10                        [-1, 16, 15, 15]          --
|    └─ConvTranspose2d: 2-11             [-1, 1, 28, 28]           65
|    └─Tanh: 2-12                        [-1, 1, 28, 28]           --
==========================================================================================
Total params: 50,161
Trainable params: 50,161
Non-trainable params: 0
Total mult-adds (M): 3.64
```

From the preceding summary, we can see that the `MaxPool2d-6` layer with a shape of batch size x 64 x 2 x 2 acts as the bottleneck layer.

Once we train the model, just like we did in the previous section (in steps 6, 7, 8, and 9), the variation of training and validation loss over increasing epochs and the predictions on input images is as follows:



From the preceding image, we can see that a convolutional autoencoder is able to make much clearer predictions of the image than the vanilla autoencoder. As an exercise, we suggest you vary the number of channels in the encoder and decoder and then analyze the variation in results.

In the next section, we will address the question of grouping similar images based on bottleneck layer values when the labels of images are not present.

Grouping similar images using t-SNE

In the previous sections, we represented each image in a much lower dimension with the assumption that similar images will have similar embeddings, and images that are not similar will have dissimilar embeddings. However, we have not yet looked at the image similarity measure or examined embedding representations in detail.

In this section, we will plot embedding (bottleneck) vectors in a 2-dimensional space. We can reduce the 64-dimensional vector of convolutional autoencoder to a 2-dimensional space by using a technique called **t-SNE**. (More about t-SNE is available here: http://www.jmlr.org/papers/v9/vandermaaten08a.html.)

This way, our understanding that similar images will have similar embeddings can be proved, as similar images should be clustered together in the two-dimensional plane. In the following code, we will represent embeddings of all the test images in a two-dimensional plane:

The following code is a continuation of the code built in the previous section, Understanding convolutional autoencoders, and is available as `conv_auto_encoder.ipynb` *in the* `Chapter 11` *folder of this book's GitHub repository -* https://tinyurl.com/mcvp-packt

1. Initialize lists so that we store the latent vectors (`latent_vectors`) and the corresponding `classes` of images (note that we store the class of each image only to verify if images of the same class, which are expected to have a very high similarity with each other, are indeed close to each other in terms of representation):

```
latent_vectors = []
```

```
classes = []
```

2. Loop through the images in the validation dataloader (`val_dl`) and store the output of the encoder layer (`model.encoder(im).view(len(im),-1)` and the class (`clss`) corresponding to each image (`im`):

```
for im,clss in val_dl:
    latent_vectors.append(model.encoder(im).view(len(im),-1))
    classes.extend(clss)
```

3. Concatenate the NumPy array of `latent_vectors`:

```
latent_vectors = torch.cat(latent_vectors).cpu()\
                      .detach().numpy()
```

4. Import t-SNE (`TSNE`) and specify that each vector is to be converted into a 2-dimensional vector (`TSNE(2)`) so that we can plot it:

```
from sklearn.manifold import TSNE
tsne = TSNE(2)
```

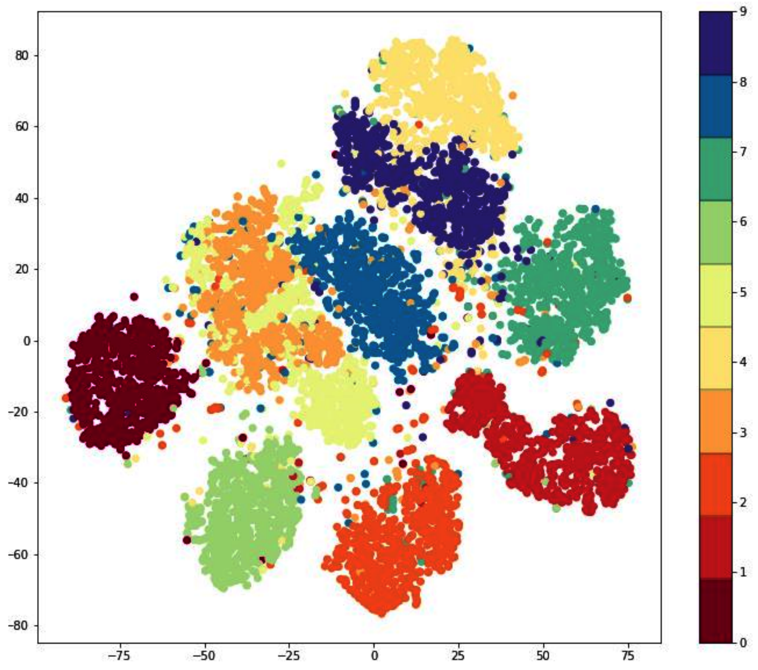
5. Fit t-SNE by running the `fit_transform` method on image embeddings (`latent_vectors`):

```
clustered = tsne.fit_transform(latent_vectors)
```

6. Plot the data points after fitting t-SNE:

```
fig = plt.figure(figsize=(12,10))
cmap = plt.get_cmap('Spectral', 10)
plt.scatter(*zip(*clustered), c=classes, cmap=cmap)
plt.colorbar(drawedges=True)
```

The preceding code provides the following output:



We can see that images of the same class are clustered together, which reinforces our understanding that the bottleneck layer has values in such a way that images that look similar will have similar values.

So far, we have learned about using autoencoders to group similar images together. In the next section, we will learn about using autoencoders to generate new images.

Understanding variational autoencoders

So far, we have seen a scenario where we can group similar images into clusters. Furthermore, we have learned that when we take embeddings of images that fall in a given cluster, we can re-construct (decode) them. However, what if an embedding (a latent vector) falls in between two clusters? There is no guarantee that we would generate realistic images. Variational autoencoders come in handy in such a scenario.

Before we dive into building a variational autoencoder, let's explore the limitations of generating images from embeddings that do not fall into a cluster (or in the middle of different clusters). First, we generate images by sampling vectors:

The following code is a continuation of the code built in the previous section, Understanding convolutional autoencoders, and is available as `conv_auto_encoder.ipynb` *in the* `chapter11` *folder of this book's GitHub repository* - https://tinyurl.com/mcvp-packt

1. Calculate the latent vectors (embeddings) of the validation images in the previous section:

```
latent_vectors = []
classes = []
for im,clss in val_dl:
    latent_vectors.append(model.encoder(im))
    classes.extend(clss)
latent_vectors = torch.cat(latent_vectors).cpu()\
                      .detach().numpy().reshape(10000, -1)
```

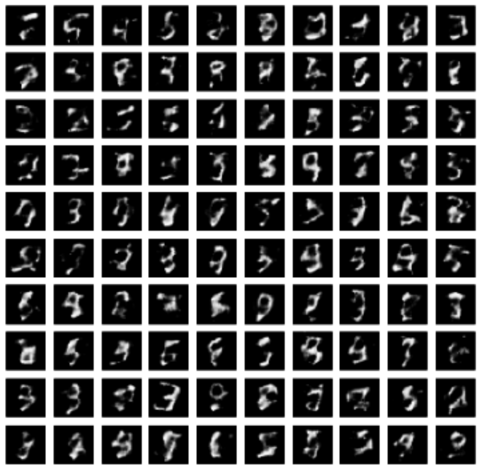
2. Generate random vectors with a column-level mean (`mu`) and a standard deviation (`sigma`) and add slight noise to the standard deviation (`torch.randn(1,100)`) before creating a vector from the mean and standard deviation. Finally, save them in a list (`rand_vectors`):

```
rand_vectors = []
for col in latent_vectors.transpose(1,0):
    mu, sigma = col.mean(), col.std()
    rand_vectors.append(sigma*torch.randn(1,100) + mu)
```

3. Plot the images reconstructed from the vectors obtained in step 2 and the model trained in the previous section:

```
rand_vectors=torch.cat(rand_vectors).transpose(1,0).to(device)
fig,ax = plt.subplots(10,10,figsize=(7,7)); ax = iter(ax.flat)
for p in rand_vectors:
    img = model.decoder(p.reshape(1,64,2,2)).view(28,28)
    show(img, ax=next(ax))
```

The preceding code results in the following output:



We can see from the preceding output that when we plot images that were generated from the mean and the noise-added standard deviation of columns of known vectors, we got images that are less clear than before. This is a realistic scenario, as we would not know beforehand about the range of embedding vectors that would generate realistic pictures.

Variational Autoencoders (**VAE**) help us resolve this problem by generating vectors that have a mean of 0 and a standard deviation of 1, thereby ensuring that we generate images that have a mean of 0 and a standard deviation of 1.

In essence, in VAE, we are specifying that the bottleneck layer should follow a certain distribution. In the next sections, we will learn about the strategy we adopt with VAE, and we will also learn about KL divergence loss, which helps us fetch bottleneck features that follow a certain distribution.

Working of VAE

In a VAE, we are building the network in such a way that a random vector that is generated from a pre-defined distribution can generate a realistic image. This was not possible with a simple autoencoder, as we did not specify the distribution of data that generates an image in the network. We enable that with a VAE by adopting the following strategy:

1. The output of the encoder is two vectors for each image:
    1. One vector represents the mean.

   2. The other represents the standard deviation.
2. From these two vectors, we fetch a modified vector that is the sum of the mean and standard deviation (which is multiplied by a random small number). The modified vector will be of the same number of dimensions as each vector.
3. The modified vector obtained in the previous step is passed as input to the decoder to fetch the image.
4. The loss value that we optimize for is a combination of the mean squared error and the KL divergence loss:
   1. KL divergence loss measures the deviation of the distribution of the mean vector and the standard deviation vector from 0 and 1, respectively.
   2. Mean squared loss is the optimization we use to re-construct (decode) an image.

By specifying that the mean vector should have a distribution centered around 0 and the standard deviation vector should be centered around 1, we are training the network in such a way that when we generate random noise with a mean of 0 and standard deviation of 1, the decoder will be able to generate a realistic image.

Further, note that, had we only minimized KL divergence, the encoder would have predicted a value of 0 for the mean vector and a standard deviation of 1 for every input. Thus, it is important to minimize KL divergence loss and mean squared loss together.

In the next section, let's learn about KL divergence so that we can incorporate it in the model's loss value calculation.

KL divergence

KL divergence helps explain the difference between two distributions of data. In our specific case, we want our bottleneck feature values to be following a normal distribution with a mean of 0 and a standard deviation of 1.

Thus, we use KL divergence loss to understand how different our bottleneck feature values are with respect to the expected distribution of values having a mean of 0 and a standard deviation of 1.

Let's take a look at how KL divergence loss helps by going through how it is calculated:

$$\sum_{i=1}^{n} \sigma_i^2 \; + \; \mu_i^2 \; - \; log(\sigma_i) \; - \; 1$$

In the preceding equation, σ and μ stand for the mean and standard deviation values of each input image.

Let's understand the intuition behind the preceding equation:

- Ensure that the mean vector is distributed around 0:
  - Minimizing mean squared error (μ_i^2) in the preceding equation ensures that $\mu$is as close to 0 as possible.
- Ensure that the standard deviation vector is distributed around 1:
  - The terms in the rest of the equation (except μ_i^2) ensure that sigma (the standard deviation vector) is distributed around 1.

The preceding loss function is minimized when the mean (μ) is 0 and the standard deviation is 1. Further, by specifying that we are considering the logarithm of standard deviation, we are ensuring that sigma values cannot be negative.

Now that we understand the high-level strategy of building a VAE and the loss function to minimize in order to obtain a pre-defined distribution of encoder output, let's implement a VAE in the next section.

Building a VAE

In this section, we will code up a VAE to generate new images of handwritten digits.

The following code is available as `VAE.ipynb` in the `Chapter11` folder of this book's GitHub repository - https://tinyurl.com/mcvp-packt

Since we have the same data, all the steps in the *Implementing vanilla autoencoders* section remain the same except steps 5 and 6, where we define the network architecture and train model respectively, which we define in the following code:

1. Step 1 to step 4, which are exactly the same as in the vanilla autoencoder section, are as follows:

```
!pip install -q torch_snippets
from torch_snippets import *
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision.utils import make_grid
device = 'cuda' if torch.cuda.is_available() else 'cpu'
train_dataset = datasets.MNIST(root='MNIST/', train=True, \
                        transform=transforms.ToTensor(), \
                               download=True)
test_dataset = datasets.MNIST(root='MNIST/', train=False, \
                        transform=transforms.ToTensor(), \
                              download=True)

train_loader = torch.utils.data.DataLoader(dataset = \
                train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset= \
                test_dataset, batch_size=64, shuffle=False)
```

2. Define the neural network class, `VAE`:

- Define the layers in the `__init__` method that will be used in the other methods:

```
class VAE(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim):
        super(VAE, self).__init__()
        self.d1 = nn.Linear(x_dim, h_dim1)
        self.d2 = nn.Linear(h_dim1, h_dim2)
        self.d31 = nn.Linear(h_dim2, z_dim)
        self.d32 = nn.Linear(h_dim2, z_dim)
        self.d4 = nn.Linear(z_dim, h_dim2)
        self.d5 = nn.Linear(h_dim2, h_dim1)
        self.d6 = nn.Linear(h_dim1, x_dim)
```

Note that the `d1` and `d2` layers will correspond to the encoder section, and `d5` and `d6` will correspond to the decoder section. The `d31` and `d32` layers are the layers that correspond to mean and standard deviation vectors respectively. However, for convenience, one assumption we will make is that we will use the `d32` layer as a representation of the log of the variance vectors.

- Define the `encoder` method:

```
def encoder(self, x):
    h = F.relu(self.d1(x))
    h = F.relu(self.d2(h))
    return self.d31(h), self.d32(h)
```

Note that the encoder returns two vectors: one vector for the mean `(self.d31(h))` and the other for the log of variance values `(self.d32(h))`.

- Define the method to sample (`sampling`) from the encoder's outputs:

```
def sampling(self, mean, log_var):
    std = torch.exp(0.5*log_var)
    eps = torch.randn_like(std)
    return eps.mul(std).add_(mean)
```

Note that exponential of *0.5*log_var* (`torch.exp(0.5*log_var)`) represents the standard deviation (`std`). Also, we are returning the addition of the mean and the standard deviation multiplied by noise generated by a random normal distribution. By multiplying by `eps`, we ensure that even with a slight change in the encoder vector, we can generate an image.

- Define the `decoder` method:

```
def decoder(self, z):
    h = F.relu(self.d4(z))
    h = F.relu(self.d5(h))
    return F.sigmoid(self.d6(h))
```

- Define the `forward` method:

```
    def forward(self, x):
        mean, log_var = self.encoder(x.view(-1, 784))
        z = self.sampling(mean, log_var)
        return self.decoder(z), mean, log_var
```

In the preceding method, we are ensuring that the encoder returns the mean and log of the variance values. Next, we are sampling with the addition of mean with epsilon multiplied by the log of the variance and returning the values after passing through the decoder.

3. Define functions to train on a batch and validate on a batch:

```
def train_batch(data, model, optimizer, loss_function):
    model.train()
    data = data.to(device)
    optimizer.zero_grad()
    recon_batch, mean, log_var = model(data)
    loss, mse, kld = loss_function(recon_batch, data, \
                                   mean, log_var)
    loss.backward()
    optimizer.step()
    return loss, mse, kld, log_var.mean(), mean.mean()

@torch.no_grad()
def validate_batch(data, model, loss_function):
    model.eval()
    data = data.to(device)
    recon, mean, log_var = model(data)
    loss, mse, kld = loss_function(recon, data, mean, \
                                   log_var)
    return loss, mse, kld, log_var.mean(), mean.mean()
```

4. Define the loss function:

```
def loss_function(recon_x, x, mean, log_var):
    RECON = F.mse_loss(recon_x, x.view(-1, 784), \
                       reduction='sum')
    KLD = -0.5 * torch.sum(1 + log_var - mean.pow(2) - \
                           log_var.exp())
    return RECON + KLD, RECON, KLD
```

In the preceding code, we are fetching the MSE loss (`RECON`) between the original image (`x`) and the reconstructed image (`recon_x`). Next, we are calculating the KL divergence loss (`KLD`) based on the formula we defined in the previous section. Note that the exponential of the log of the variance is the variance value.

5. Define the model object (`vae`) and the `optimizer` function:

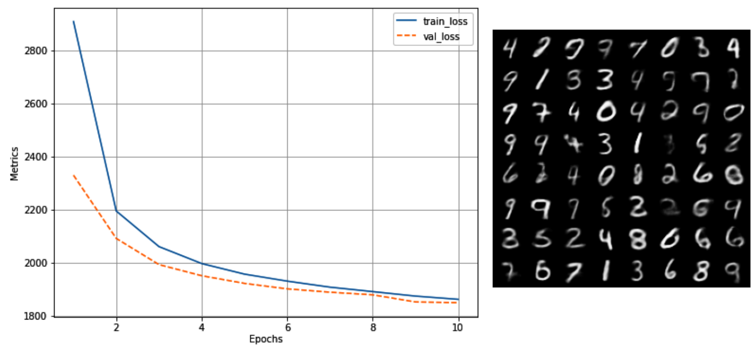
```
vae = VAE(x_dim=784, h_dim1=512, h_dim2=256, \
          z_dim=50).to(device)
optimizer = optim.AdamW(vae.parameters(), lr=1e-3)
```

6. Train the model over increasing epochs:

```
n_epochs = 10
log = Report(n_epochs)

for epoch in range(n_epochs):
    N = len(train_loader)
    for batch_idx, (data, _) in enumerate(train_loader):
        loss, recon, kld, log_var, mean = train_batch(data, \
                                            vae, optimizer, \
                                               loss_function)
        pos = epoch + (1+batch_idx)/N
        log.record(pos, train_loss=loss, train_kld=kld, \
                   train_recon=recon,train_log_var=log_var, \
                   train_mean=mean, end='\r')

    N = len(test_loader)
    for batch_idx, (data, _) in enumerate(test_loader):
        loss, recon, kld,log_var,mean = validate_batch(data, \
                                           vae, loss_function)
        pos = epoch + (1+batch_idx)/N
        log.record(pos, val_loss=loss, val_kld=kld, \
                   val_recon=recon, val_log_var=log_var, \
                   val_mean=mean, end='\r')

    log.report_avgs(epoch+1)
    with torch.no_grad():
        z = torch.randn(64, 50).to(device)
        sample = vae.decoder(z).to(device)
        images = make_grid(sample.view(64, 1, 28, 28))\
                                          .permute(1,2,0)
        show(images)
log.plot_epochs(['train_loss','val_loss'])
```

While the majority of the preceding code is familiar, let's understand the grid image generation process. We are first generating a random vector (`z`) and passing it through the decoder (`vae.decoder`) to fetch a sample of images. The `make_grid` function plots images (and denormalizes them automatically, if required, before plotting).

The output of loss value variations and a sample of images generated is as follows:



We can see that we are able to generate realistic new images that were not present in the original image.

So far, we have learned about generating new images using VAEs. However, what if we want to modify images in such a way that a model cannot identify the right class? We will learn about the technique leveraged to address this in the next section.

Performing an adversarial attack on images

In the previous section, we learned about generating an image from random noise using a VAE. However, it was an unsupervised exercise. What if we want to modify an image in such a way that the change in image is so minimal that it is indistinguishable from the original image for a human, but still the neural network model perceives the object as belonging to a different class? Adversarial attacks on images come in handy in such a scenario.

Adversarial attacks refer to the changes that we make to input image values (pixels) so that we meet a certain objective.

In this section, we will learn about modifying an image slightly in such a way that the pre-trained models now predict them as a different class (specified by the user) and not the original class. The strategy we will adopt is as follows:

1. Provide an image of an elephant.
2. Specify the target class corresponding to the image.
3. Import a pre-trained model where the parameters of the model are set so that they are not updated (`gradients = False`).
4. Specify that we calculate gradients on input image pixel values and not on the weight values of the network. This is because while training to fool a network, we do not have control over the model, but have control only over the image we send to the model.
5. Calculate the loss corresponding to the model predictions and the target class.
6. Perform backpropagation on the model. This step helps us understand the gradient associated with each input pixel value.
7. Update the input image pixel values based on the direction of the gradient corresponding to each input pixel value.
8. Repeat *steps 5*, *6*, and *7* over multiple epochs.

Let's do this with code:

The following code is available as `adversarial_attack.ipynb` in the `Chapter11` folder of this book's GitHub repository - https://tinyurl.com/mcvp-packt The code contains URLs to download data from. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.

1. Import the relevant packages, the image that we work on for this use case, and the pre-trained ResNet50 model. Also, specify that we want to freeze parameters:

```
!pip install torch_snippets
from torch_snippets import inspect, show, np, torch, nn
from torchvision.models import resnet50
model = resnet50(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
model = model.eval()
import requests
from PIL import Image
url = 'https://lionsvalley.co.za/wp-content/uploads/2015/11/african-elephant-square.jpg'
original_image = Image.open(requests.get(url, stream=True)\
                            .raw).convert('RGB')
original_image = np.array(original_image)
original_image = torch.Tensor(original_image)
```

2. Import Imagenet classes and assign IDs to each class:

```
image_net_classes = 'https://gist.githubusercontent.com/yrevar/942d3a0ac09ec9e5eb3a/raw/238f720ff059c1f82f3
image_net_classes = requests.get(image_net_classes).text
image_net_ids = eval(image_net_classes)
image_net_classes = {i:j for j,i in image_net_ids.items()}
```

3. Specify a function to normalize (`image2tensor`) and denormalize (`tensor2image`) the image:

```
from torchvision import transforms as T
from torch.nn import functional as F
normalize = T.Normalize([0.485, 0.456, 0.406],
                        [0.229, 0.224, 0.225])
denormalize=T.Normalize( \
                [-0.485/0.229,-0.456/0.224,-0.406/0.225],
                [1/0.229, 1/0.224, 1/0.225])
def image2tensor(input):
    x = normalize(input.clone().permute(2,0,1)/255.)[None]
    return x
def tensor2image(input):
    x = (denormalize(input[0].clone()).permute(1,2,0)*255.)\
                                      .type(torch.uint8)
    return x
```

4. Define a function to predict on a given image (`predict_on_image`):

```
def predict_on_image(input):
    model.eval()
    show(input)
    input = image2tensor(input)
    pred = model(input)
    pred = F.softmax(pred, dim=-1)[0]
    prob, clss = torch.max(pred, 0)
    clss = image_net_ids[clss.item()]
    print(f'PREDICTION: `{clss}` @ {prob.item()}')
```

In the preceding code, we are converting an input image into a tensor (which is a function to normalize using the `image2tensor` method defined earlier) and passing through a `model` to fetch the class (`clss`) of the object in the image and probability (`prob`) of prediction.

5. Define the `attack` function:

- The `attack` function takes `image`, `model`, and `target` as input:

```
from tqdm import trange
losses = []
def attack(image, model, target, epsilon=1e-6):
```

- Convert the image into a tensor and specify that the input requires gradients to be calculated:

```
    input = image2tensor(image)
    input.requires_grad = True
```

- Calculate the prediction by passing the normalized input (`input`) through the model, and then calculate the loss value corresponding to the specified target class:

```
    pred = model(input)
    loss = nn.CrossEntropyLoss()(pred, target)
```

- Perform backpropagation to reduce the loss:

```
    loss.backward()
    losses.append(loss.mean().item())
```

- Update the image very slightly based on the gradient direction:

```
        output = input - epsilon * input.grad.sign()
```

  In the preceding code, we are updating input values by a very small amount (multiplying by `epsilon`). Also, we are not updating the image by the magnitude of the gradient, but the direction of gradient only (`input.grad.sign()`) after multiplying it by a very small value (`epsilon`).

- Return the output after converting the tensor to an image (`tensor2image`), which denormalizes the image:

```
        output = tensor2image(output)
        del input
        return output.detach()
```

6. Modify the image to belong to a different class:

- Specify the targets (`desired_targets`) that we want to convert the image to:

```
modified_images = []
desired_targets = ['lemon', 'comic book', 'sax, saxophone']
```

- Loop through the targets and specify the target class in each iteration:

```
for target in desired_targets:
    target = torch.tensor([image_net_classes[target]])
```

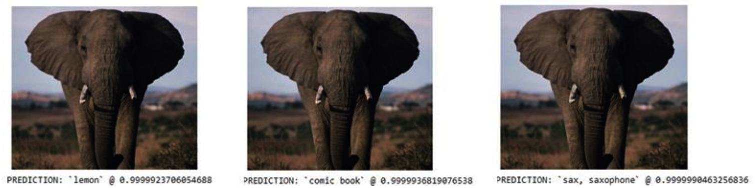
- Modify the image to attack over increasing epochs and collect them in a list:

```
    image_to_attack = original_image.clone()
    for _ in trange(10):
        image_to_attack = attack(image_to_attack,model,target)
    modified_images.append(image_to_attack)
```

- The following code results in modified images and the corresponding classes:

```
for image in [original_image, *modified_images]:
    predict_on_image(image)
    inspect(image)
```

The preceding code generates the following:



```
PREDICTION: `lemon` @ 0.9999923706054688    PREDICTION: `comic book` @ 0.9999936819076538    PREDICTION: `sax, saxophone` @ 0.9999990463256836
```

We can see that as we modify the image very slightly, the prediction class is completely different but with very high confidence.

Now that we understand how to modify images to so that they are classed as we wish, in the next section, we will learn about modifying an image (a content image) in the style of our choice. We must provide a content image and a style image.

Performing neural style transfer

In neural style transfer, we have a content image and a style image, and we combine these two images in such a way that the combined image preserves the content of the content image while maintaining the style of the style image.

An example style image and content image are as follows:

In the preceding picture, we want to retain the content in the picture on right (the content image), but overlay it with the color and texture in the picture on the left (the style image).

The process of performing neural style transfer is as follows. We try to modify the original image in a way that the loss value is split into **content loss** and **style loss**. Content loss refers to how **different** the generated image is from the content image. Style loss refers to how **correlated** the style image is to the generated image.

While we mentioned that the loss is calculated based on the difference in images, in practice, we modify it slightly by ensuring that the loss is calculated using the feature layer activations of images and not the original images. For example, the content loss at layer 2 will be the squared difference between the *activations of the content image and the generated image* when passed through the second layer.

Loss is calculated on feature layers and not the original image, as the feature layers capture certain attributes of the original image (for example, the outline of the foreground corresponding to the original image in the higher layers and the details of fine-grained objects in the lower layers).

While calculating the content loss seems straightforward, let's try to understand how to calculate the similarity between the generated image and the style image. A technique called **gram matrix** comes in handy. Gram matrix calculates the similarity between a generated image and a style image, and is calculated as follows:

$$L_{GM}(S,G,l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (GM[l](S)_{ij} - GM[l](G)_{ij})^2$$

GM[l] is the gram matrix value at layer *l* for the style image, *S*, and the generated image, *G*.
A gram matrix results from multiplying a matrix by the transpose of itself. Let's understand the use of this operation.

Imagine that you are working on a layer that has a feature output of 32 x 32 x 256. The gram matrix is calculated as the correlation of each of the 32 x 32 values in a channel with respect to the values across all channels. Thus, the gram matrix calculation results in a matrix that is 256 x 256 in shape. We now compare the 256 x 256 values of the style image and the generated image to calculate the style loss.

Let's understand why GramMatrix is important for style transfer.

In a successful scenario, say we transferred Picasso's style to the Mona Lisa. Let's call the Picasso style *St* (for style), the original Mona Lisa *So* (for source), and the final image *Ta* (for target). Note that in an ideal scenario, the local features in image *Ta* are the same as the local features in *St*. Even though the content might not be the same, getting similar colors, shapes, and textures as the style image into the target image is what is important in style transfer.

By extension, if we were to send *So* and extract its features from an intermediate layer of VGG19, they will vary from the features obtained by sending *Ta*. However, within each feature set, the corresponding vectors will vary relatively with each other in a similar fashion. Say, for example, the ratio of the mean of the first channel to the mean of the second channel in both the feature sets will be similar. This is why we are trying to compute using Gram Loss.

Content loss is calculated by comparing the difference in feature activations of the content image with respect to the generated image. Style loss is calculated by first calculating the gram matrix in the pre-defined layers and then comparing the gram matrices of the generated image and the style image.
Now that we are able to calculate the style loss and the content loss, the final modified input image is the image that minimizes the overall loss, that is, a weighted average of the style and content loss.
The high-level strategy we adopt to implement neural style transfer is as follows:

1. Pass the input image through a pre-trained model.

2. Extract the layer values at pre-defined layers.
3. Pass the generated image through the model and extract its values at the same pre-defined layers.
4. Calculate the content loss at each layer corresponding to the content image and generated image.
5. Pass the style image through multiple layers of the model and calculate the gram matrix values of the style image.
6. Pass the generated image through the same layers that the style image is passed through and calculate its corresponding gram matrix values.
7. Extract the squared difference of the gram matrix values of the two images. This will be the style loss.
8. The overall loss will be the weighted average of the style loss and content loss.
9. The input image that minimizes the overall loss will be the final image of interest.

Let's now code up the preceding strategy:

The following code is available as `neural_style_transfer.ipynb` *in the* `chapter11` *folder of this book's GitHub repository -* https://tinyurl.com/mcvp-packt *The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce results while you understand the steps to perform and explanation of various code components in text.*

1. Import the relevant packages:

```
!pip install torch_snippets
from torch_snippets import *
from torchvision import transforms as T
from torch.nn import functional as F
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

2. Define the functions to preprocess and postprocess the data:

```
from torchvision.models import vgg19
preprocess = T.Compose([
                T.ToTensor(),
                T.Normalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225]),
                T.Lambda(lambda x: x.mul_(255))
                ])
postprocess = T.Compose([
                T.Lambda(lambda x: x.mul_(1./255)),
                T.Normalize(\
                mean=[-0.485/0.229,-0.456/0.224,-0.406/0.225],
                            std=[1/0.229, 1/0.224, 1/0.225]),
                ])
```

3. Define the `GramMatrix` module:

```
class GramMatrix(nn.Module):
    def forward(self, input):
        b,c,h,w = input.size()
        feat = input.view(b, c, h*w)
        G = feat@feat.transpose(1,2)
        G.div_(h*w)
        return G
```

In the preceding code, we are computing all the possible inner products of the features with themselves, which is basically asking how all the vectors relate to each other.

4. Define the gram matrix's corresponding MSE loss, `GramMSELoss`:

```
class GramMSELoss(nn.Module):
    def forward(self, input, target):
        out = F.mse_loss(GramMatrix()(input), target)
        return(out)
```

Once we have the gram vectors for both feature sets, it is important that they match as closely as possible, and hence the `mse_loss`.

5. Define the model class, `vgg19_modified`:

- Initialize the class:

```
class vgg19_modified(nn.Module):
    def __init__(self):
        super().__init__()
```

- Extract the features:

```
        features = list(vgg19(pretrained = True).features)
        self.features = nn.ModuleList(features).eval()
```

- Define the `forward` method, which takes the list of layers and returns the features corresponding to each layer:

```
    def forward(self, x, layers=[]):
        order = np.argsort(layers)
        _results, results = [], []
        for ix,model in enumerate(self.features):
            x = model(x)
            if ix in layers: _results.append(x)
        for o in order: results.append(_results[o])
        return results if layers is not [] else x
```

- Define the model object:

```
vgg = vgg19_modified().to(device)
```

6. Import the content and style images:

```
!wget https://www.dropbox.com/s/z1y0fy2r6z6m6py/60.jpg
!wget https://www.dropbox.com/s/1svdliljyo0a98v/style_image.png
```

- Make sure that the images are resized to be of the same shape, 512 x 512 x 3:

```
imgs = [Image.open(path).resize((512,512)).convert('RGB') \
        for path in ['style_image.png', '60.jpg']]
style_image,content_image=[preprocess(img).to(device)[None] \
                              for img in imgs]
```

7. Specify that the content image is to modified with `requires_grad = True`:

```
opt_img = content_image.data.clone()
```

```
opt_img.requires_grad = True
```

8. Specify the layers that define content loss and style loss, that is, which intermediate VGG layers we are using, to compare gram matrices for style and raw feature vectors for content:

```
style_layers = [0, 5, 10, 19, 28]
content_layers = [21]
loss_layers = style_layers + content_layers
```

9. Define the loss function for content and style loss values:

```
loss_fns = [GramMSELoss()] * len(style_layers) + \
            [nn.MSELoss()] * len(content_layers)
loss_fns = [loss_fn.to(device) for loss_fn in loss_fns]
```

10. Define the weightage associated with content and style loss:

```
style_weights = [1000/n**2 for n in [64,128,256,512,512]]
content_weights = [1]
weights = style_weights + content_weights
```

11. We need to manipulate our image such that the style of the target image resembles `style_image` as much as possible. Hence we compute the `style_targets` values of `style_image` by computing `GramMatrix` of features obtained from a few chosen layers of VGG. Since the overall content should be preserved, we choose the `content_layer` variable at which we compute the raw features from VGG:

```
style_targets = [GramMatrix()(A).detach() for A in \
                 vgg(style_image, style_layers)]
content_targets = [A.detach() for A in \
                   vgg(content_image, content_layers)]
targets = style_targets + content_targets
```

12. Define the `optimizer` and the number of iterations (`max_iters`). Even though we could have used Adam or any other optimizer, LBFGS is an optimizer that has been observed to work best in deterministic

scenarios. Additionally, since we are dealing with exactly one image, there is nothing random. Many experiments have revealed that LBFGS converges faster and to lower losses in neural transfer settings, so we will use this optimizer:

```
max_iters = 500
optimizer = optim.LBFGS([opt_img])
log = Report(max_iters)
```

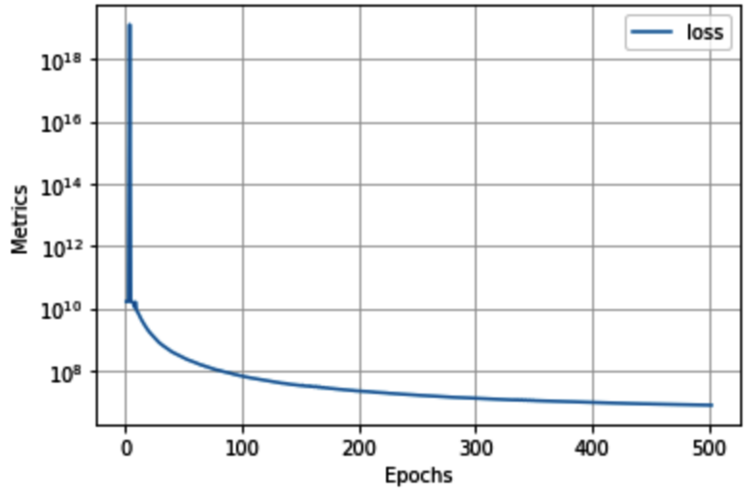
13. Perform the optimization. In deterministic scenarios where we are iterating on the same tensor again and again, we can wrap the optimizer step as a function with zero arguments and repeatedly call it, as shown here:

```
iters = 0
while iters < max_iters:
    def closure():
        global iters
        iters += 1
        optimizer.zero_grad()
        out = vgg(opt_img, loss_layers)
        layer_losses = [weights[a]*loss_fns[a](A,targets[a]) \
                        for a,A in enumerate(out)]
        loss = sum(layer_losses)
        loss.backward()
        log.record(pos=iters, loss=loss, end='\r')
        return loss
    optimizer.step(closure)
```

14. Plot the variation in the loss:

```
log.plot(log=True)
```

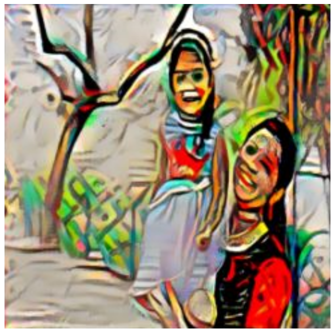
This results in the following output:



15. Plot the image with the combination of content and style images:

```
out_img = postprocess(opt_img[0]).permute(1,2,0)
show(out_img)
```

The output is as follows:



From the preceding picture, we can see that the image is a combination of content and style images.

With this, we have seen two ways of manipulating an image: an adversarial attack to modify the class of an image, and style transfer to combine the style of one image with the content of another image. In the next section, we will learn about generating deep fakes, which transfer an expression from one face to another.

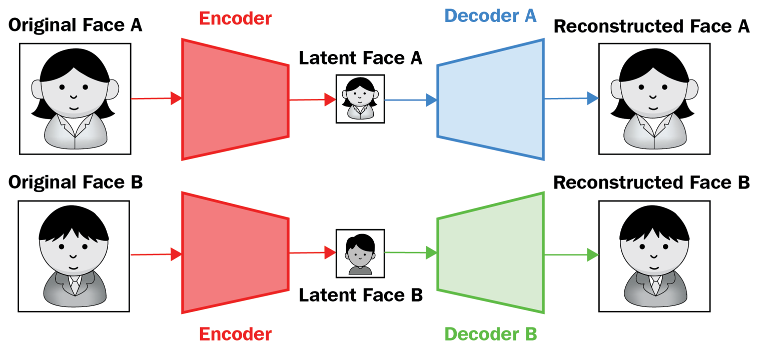
Generating deep fakes

We have learned about two different image-to-image tasks so far: semantic segmentation with UNet and image reconstruction with autoencoders. Deep fakery is an image-to-image task that has a very similar underlying theory.

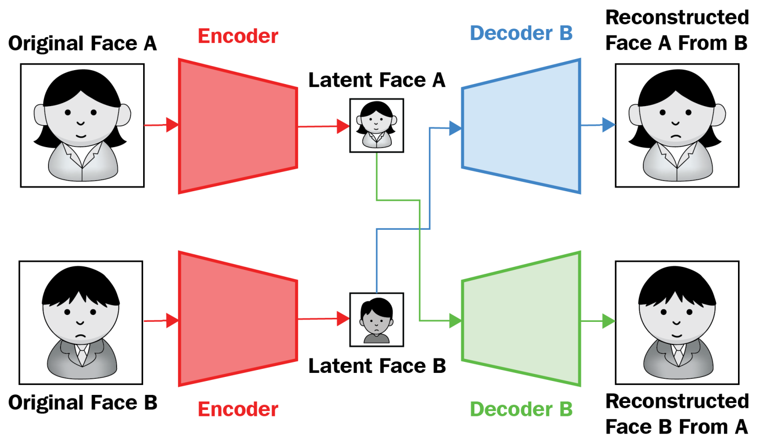
Imagine a scenario where you want to create an application that takes a given image of a face and changes the facial expression in a way that you want. Deep fakes come in handy in this scenario. While we will not discuss the very latest in deep fakes in this book, techniques such as few-shot adversarial learning are developed to generate realistic images with the facial expression of interest. Knowledge of how deep fakes work and GANs (which you will learn about in the next chapters) will help you identify videos that are fake videos.

In the task of deep fakery, we would have a few hundred pictures of person A and a few hundred pictures of person B. The objective is to reconstruct person B's face with the facial expression of person A and vice versa.

The following diagram explains how the deep fake image generation process works:



In the preceding picture, we are passing images of person A and person B through an encoder (**Encoder**). Once we get the latent vectors corresponding to person A (**Latent Face A**) and person B (**Latent Face B**), we pass the latent vectors through their corresponding decoders (**Decoder A** and **Decoder B**) to fetch the corresponding original images (**Reconstructed Face A** and **Reconstructed Face B**). So far, the concept of encoder and decoder is very similar to what we learned in the *Autoencoders* section. However, in this scenario, *we have only one encoder, but two decoders* (each decoder corresponding to a different person). The expectation is that the latent vectors obtained from the encoder represent the information about the facial expression present within the image, while the decoder fetches the image corresponding to the person. Once the encoder and the two decoders are trained, while performing deep fake image generation, we switch the connection within our architecture as follows:



When the latent vector of person A is passed through decoder B, the reconstructed face of person B will have the characteristics of person A (a smiling face) and vice versa for person B when passed through decoder A (a sad face).

> *One additional trick that helps in generating a realistic image is warping face images and feeding them to the network in such a way that the warped face is the input and the original image is expected as the output.*

Now that we understand how it works, let's implement the generation of fake images of one person with the expression of another person using autoencoders with the following code:

The following code is available as `Generating_Deep_Fakes.ipynb` *in the* `Chapter11` *folder of this book's GitHub repository -* https://tinyurl.com/mcvp-packt *The code contains URLs to download data from and is moderately lengthy. We strongly recommend you to execute the notebook in GitHub to reproduce* *results while you understand the steps to perform and explanation of various code components in text.*

1. Let's download the data and the source code as follows:

```
import os
if not os.path.exists('Faceswap-Deepfake-Pytorch'):
    !wget -q https://www.dropbox.com/s/5ji7jl7httso9ny/person_images.zip
    !wget -q https://raw.githubusercontent.com/sizhky/deep-fake-util/main/random_warp.py
    !unzip -q person_images.zip
!pip install -q torch_snippets torch_summary
from torch_snippets import *
from random_warp import get_training_data
```

2. Fetch face crops from the images:

- Define the face cascade, which draws a bounding box around the face in an image. There's more on cascades in *Chapter 18, OpenCV Utilities for Image Analysis*. However, for now, it suffices to say that the face cascade draws a tight bounding box around the face present in the image:

```
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + \
                        'haarcascade_frontalface_default.xml')
```

- Define a function (`crop_face`) for cropping faces from an image:

```
def crop_face(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    if(len(faces)>0):
        for (x,y,w,h) in faces:
            img2 = img[y:(y+h),x:(x+w),:]
        img2 = cv2.resize(img2,(256,256))
        return img2, True
    else:
        return img, False
```

In the preceding function, we are passing the grayscaled image (`gray`) through face cascade and cropping the rectangle that contains the face. Next, we are returning a re-sized image (`img2`).

Further, to account for a scenario where there is no face detected in the image, we are passing a flag to show whether a face is detected.

- Crop the images of `personA` and `personB` and place them in separate folders:

```
!mkdir cropped_faces_personA
!mkdir cropped_faces_personB

def crop_images(folder):
    images = Glob(folder+'/*.jpg')
    for i in range(len(images)):
        img = read(images[i],1)
        img2, face_detected = crop_face(img)
        if(face_detected==False):
            continue
        else:
            cv2.imwrite('cropped_faces_'+folder+'/'+str(i)+ \
                '.jpg',cv2.cvtColor(img2, cv2.COLOR_RGB2BGR))
crop_images('personA')
crop_images('personB')
```

3. Create a dataloader and inspect the data:

```
class ImageDataset(Dataset):
    def __init__(self, items_A, items_B):
        self.items_A = np.concatenate([read(f,1)[None] \
                                       for f in items_A])/255.
        self.items_B = np.concatenate([read(f,1)[None] \
                                       for f in items_B])/255.
        self.items_A += self.items_B.mean(axis=(0, 1, 2)) \
                        - self.items_A.mean(axis=(0, 1, 2))

    def __len__(self):
        return min(len(self.items_A), len(self.items_B))
    def __getitem__(self, ix):
        a, b = choose(self.items_A), choose(self.items_B)
        return a, b

    def collate_fn(self, batch):
        imsA, imsB = list(zip(*batch))
        imsA, targetA = get_training_data(imsA, len(imsA))
        imsB, targetB = get_training_data(imsB, len(imsB))
        imsA, imsB, targetA, targetB = [torch.Tensor(i)\
                                        .permute(0,3,1,2)\
                                        .to(device) \
                                        for i in [imsA, imsB,\
                                        targetA, targetB]]
        return imsA, imsB, targetA, targetB

a = ImageDataset(Glob('cropped_faces_personA'), \
                 Glob('cropped_faces_personB'))
x = DataLoader(a, batch_size=32, collate_fn=a.collate_fn)
```

The dataloader is returning four tensors, `imsA`, `imsB`, `targetA`, and `targetB`. The first tensor (`imsA`) is a distorted (warped) version of the third tensor (`targetA`) and the second (`imsB`) is a distorted (warped) version of the fourth tensor (`targetB`).

Also, as you can see in the line `a =ImageDataset(Glob('cropped_faces_personA'), Glob('cropped_faces_personB'))`, we have two folders of images, one for each person. There is no relation between any of the faces, and in the `__iteritems__` dataset, we are randomly fetching two faces every time.

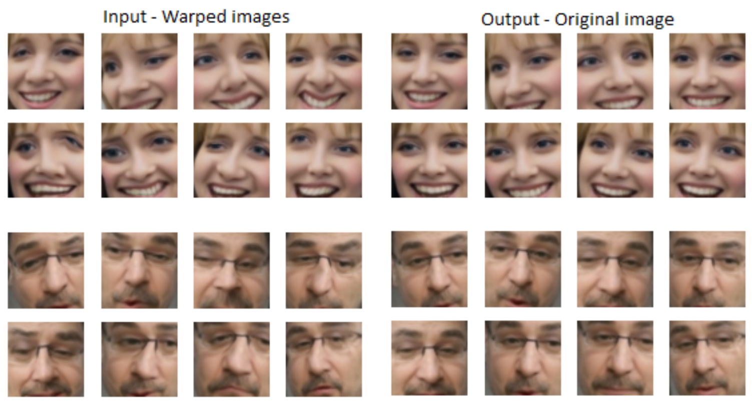
The key function in this step is `get_training_data`, present in `collate_fn`. This is an augmentation function for warping (distorting) faces. We are giving distorted faces as input to the autoencoder and trying to predict regular faces.

> *The advantage of warping is that not only does it increase our training data size but also acts as a regularizer to the network, which is forced to understand key facial features despite being given a distorted face.*

- Let's inspect a few images:

```
inspect(*next(iter(x)))

for i in next(iter(x)):
    subplots(i[:8], nc=4, sz=(4,2))
```

The preceding code results in the following output:



Note that the input images are warped, while the output images are not, and the input to output images now have a one-to-one correspondence.

4. Build the model and inspect it:

- Define the convolution (`_ConvLayer`) and upscaling (`_UpScale`) functions as well as the `Reshape` class that will be leveraged while building model:

```
def _ConvLayer(input_features, output_features):
    return nn.Sequential(
        nn.Conv2d(input_features, output_features,
                  kernel_size=5, stride=2, padding=2),
        nn.LeakyReLU(0.1, inplace=True)
    )

def _UpScale(input_features, output_features):
    return nn.Sequential(
        nn.ConvTranspose2d(input_features, output_features,
                         kernel_size=2, stride=2, padding=0),
        nn.LeakyReLU(0.1, inplace=True)
    )

class Reshape(nn.Module):
    def forward(self, input):
        output = input.view(-1, 1024, 4, 4) # channel * 4 * 4
        return output
```

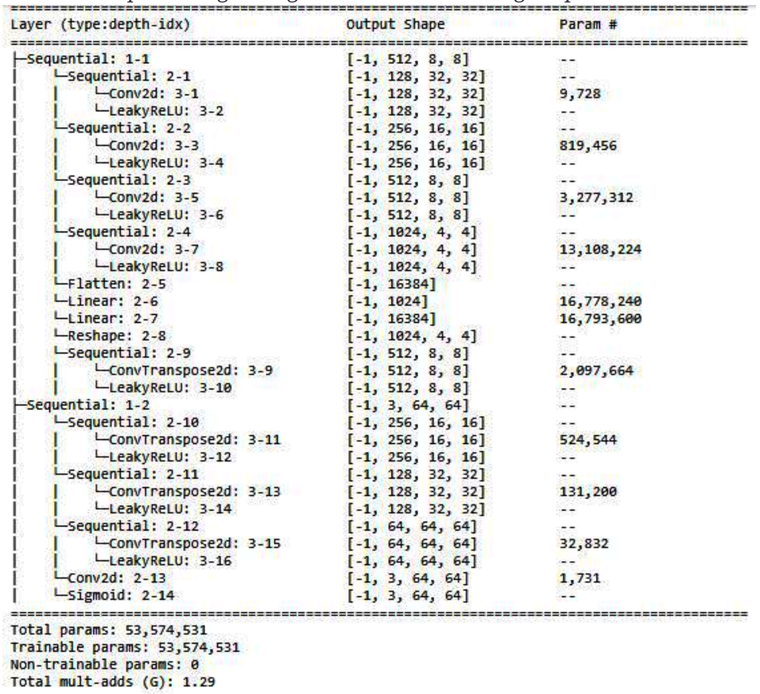
- Define the `Autoencoder` model class, which has a single `encoder` and two decoders (`decoder_A` and `decoder_B`):

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()

        self.encoder = nn.Sequential(
                        _ConvLayer(3, 128),
                        _ConvLayer(128, 256),
                        _ConvLayer(256, 512),
                        _ConvLayer(512, 1024),
                        nn.Flatten(),
                        nn.Linear(1024 * 4 * 4, 1024),
                        nn.Linear(1024, 1024 * 4 * 4),
                        Reshape(),
                        _UpScale(1024, 512),
                    )

        self.decoder_A = nn.Sequential(
                        _UpScale(512, 256),
                        _UpScale(256, 128),
                        _UpScale(128, 64),
                        nn.Conv2d(64, 3, kernel_size=3, \
                                  padding=1),
                        nn.Sigmoid(),
                    )

        self.decoder_B = nn.Sequential(
                        _UpScale(512, 256),
                        _UpScale(256, 128),
                        _UpScale(128, 64),
                        nn.Conv2d(64, 3, kernel_size=3, \
                                  padding=1),
                        nn.Sigmoid(),
                    )

    def forward(self, x, select='A'):
        if select == 'A':
            out = self.encoder(x)
            out = self.decoder_A(out)
        else:
            out = self.encoder(x)
            out = self.decoder_B(out)
        return out
```

- Generate a summary of the model:

```
from torchsummary import summary
model = Autoencoder()
summary(model, torch.zeros(32,3,64,64), 'A');
```

The preceding code generates the following output:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 512, 8, 8]           --
|    └─Sequential: 2-1                   [-1, 128, 32, 32]         --
|    |    └─Conv2d: 3-1                  [-1, 128, 32, 32]         9,728
|    |    └─LeakyReLU: 3-2               [-1, 128, 32, 32]         --
|    └─Sequential: 2-2                   [-1, 256, 16, 16]         --
|    |    └─Conv2d: 3-3                  [-1, 256, 16, 16]         819,456
|    |    └─LeakyReLU: 3-4               [-1, 256, 16, 16]         --
|    └─Sequential: 2-3                   [-1, 512, 8, 8]           --
|    |    └─Conv2d: 3-5                  [-1, 512, 8, 8]           3,277,312
|    |    └─LeakyReLU: 3-6               [-1, 512, 8, 8]           --
|    └─Sequential: 2-4                   [-1, 1024, 4, 4]          --
|    |    └─Conv2d: 3-7                  [-1, 1024, 4, 4]          13,108,224
|    |    └─LeakyReLU: 3-8               [-1, 1024, 4, 4]          --
|    └─Flatten: 2-5                      [-1, 16384]               --
|    └─Linear: 2-6                       [-1, 1024]                16,778,240
|    └─Linear: 2-7                       [-1, 16384]               16,793,600
|    └─Reshape: 2-8                      [-1, 1024, 4, 4]          --
|    └─Sequential: 2-9                   [-1, 512, 8, 8]           --
|    |    └─ConvTranspose2d: 3-9         [-1, 512, 8, 8]           2,097,664
|    |    └─LeakyReLU: 3-10              [-1, 512, 8, 8]           --
├─Sequential: 1-2                        [-1, 3, 64, 64]           --
|    └─Sequential: 2-10                  [-1, 256, 16, 16]         --
|    |    └─ConvTranspose2d: 3-11        [-1, 256, 16, 16]         524,544
|    |    └─LeakyReLU: 3-12              [-1, 256, 16, 16]         --
|    └─Sequential: 2-11                  [-1, 128, 32, 32]         --
|    |    └─ConvTranspose2d: 3-13        [-1, 128, 32, 32]         131,200
|    |    └─LeakyReLU: 3-14              [-1, 128, 32, 32]         --
|    └─Sequential: 2-12                  [-1, 64, 64, 64]          --
|    |    └─ConvTranspose2d: 3-15        [-1, 64, 64, 64]          32,832
|    |    └─LeakyReLU: 3-16              [-1, 64, 64, 64]          --
|    └─Conv2d: 2-13                      [-1, 3, 64, 64]           1,731
|    └─Sigmoid: 2-14                     [-1, 3, 64, 64]           --
==========================================================================================
Total params: 53,574,531
Trainable params: 53,574,531
Non-trainable params: 0
Total mult-adds (G): 1.29
```

5. Define the `train_batch` logic:

```
def train_batch(model, data, criterion, optimizes):
    optA, optB = optimizers
    optA.zero_grad()
    optB.zero_grad()
    imgA, imgB, targetA, targetB = data
    _imgA, _imgB = model(imgA, 'A'), model(imgB, 'B')

    lossA = criterion(_imgA, targetA)
    lossB = criterion(_imgB, targetB)
    lossA.backward()
    lossB.backward()

    optA.step()
    optB.step()

    return lossA.item(), lossB.item()
```

What we are interested in is running `model(imgA, 'B')` (which would return an image of class B using an input image from class A), but we do not have a ground truth to compare it against. So instead, what we are doing is predicting `_imgA` from `imgA` (where `imgA` is a distorted version of `targetA`) and comparing `_imgA` with `targetA` using `nn.L1Loss`.

We do not need `validate_batch` as there is no validation dataset. We will predict new images during training and qualitatively see the progress.

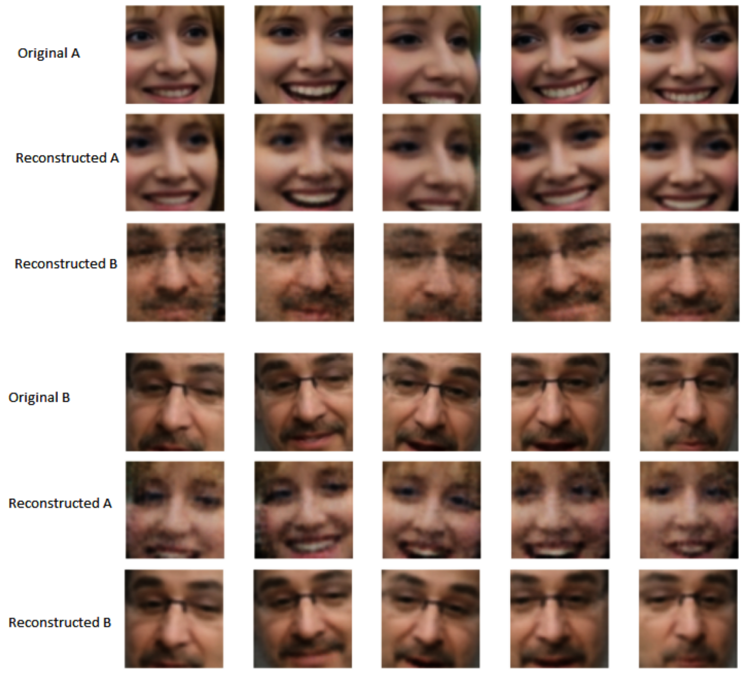
6. Create all the required components to train the model:

```
model = Autoencoder().to(device)

dataset = ImageDataset(Glob('cropped_faces_personA'), \
                       Glob('cropped_faces_personB'))
dataloader = DataLoader(dataset, 32, \
                        collate_fn=dataset.collate_fn)

optimizers=optim.Adam( \
                [{'params': model.encoder.parameters()}, \
                 {'params': model.decoder_A.parameters()}], \
                 lr=5e-5, betas=(0.5, 0.999)), \
        optim.Adam([{'params': model.encoder.parameters()}, \
                 {'params': model.decoder_B.parameters()}], \
                        lr=5e-5, betas=(0.5, 0.999))

criterion = nn.L1Loss()
```

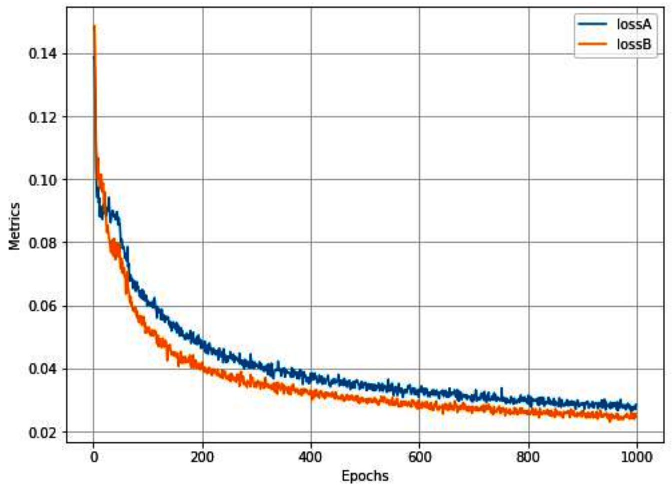
7. Train the model:

```
n_epochs = 1000
log = Report(n_epochs)
!mkdir checkpoint
for ex in range(n_epochs):
    N = len(dataloader)
    for bx,data in enumerate(dataloader):
        lossA, lossB = train_batch(model, data,
                                   criterion, optimizers)
        log.record(ex+(1+bx)/N, lossA=lossA,
                   lossB=lossB, end='\r')

    log.report_avgs(ex+1)
    if (ex+1)%100 == 0:
        state = {
                'state': model.state_dict(),
                'epoch': ex
            }
        torch.save(state, './checkpoint/autoencoder.pth')

    if (ex+1)%100 == 0:
        bs = 5
        a,b,A,B = data
        line('A to B')
        _a = model(a[:bs], 'A')
        _b = model(a[:bs], 'B')
        x = torch.cat([A[:bs],_a,_b])
        subplots(x, nc=bs, figsize=(bs*2, 5))

        line('B to A')
        _a = model(b[:bs], 'A')
        _b = model(b[:bs], 'B')
        x = torch.cat([B[:bs],_a,_b])
        subplots(x, nc=bs, figsize=(bs*2, 5))

log.plot_epochs()
```

The preceding code results in reconstructed images, as follows:



The variation in loss values is as follows:



As you can see, we can swap expressions from one face to another by tweaking an autoencoder to have two decoders instead of one. Furthermore, with a higher number of epochs, the reconstructed image gets more realistic.

Summary

In this chapter, we have learned about the different variants of autoencoders: vanilla, convolutional, and variational. We also learned about how the number of units in the bottleneck layer influences the reconstructed image. Next, we learned about identifying images that are similar to a given image using the t-SNE technique. We learned that when we sample vectors, we cannot get realistic images, and by using variational autoencoders, we learned about generating new images by using a combination of reconstruction loss and KL divergence loss. Next, we learned how to perform an adversarial attack on images to modify the class of an image while not changing the perceptive content of the image. Finally, we learned about leveraging the combination of content loss and gram matrix-based style loss to optimize for content and style loss of images to come up with an image that is a combination of two input images. Finally, we learned about tweaking an autoencoder to swap two faces without any supervision.

Now that we have learned about generating novel images from a given set of images, in the next chapter, we will build upon this topic to generate completely new images using variants of a network called the Generative Adversarial Network.

Questions

1. What is an encoder in an autoencoder?
2. What loss function does an autoencoder optimize for?
3. How do autoencoders help in grouping similar images?
4. When is a convolutional autoencoder useful?
5. Why do we get non-intuitive images if we randomly sample from vector space of embeddings obtained from vanilla/convolutional autoencoders?
6. What are the loss functions that VAEs optimize for?
7. How do VAEs overcome the limitation of vanilla/convolutional autoencoders to generate new images?
8. During an adversarial attack, why do we modify the input image pixels and not the weight values?

9. In a neural style transfer, what are the losses that we optimize for?
10. Why do we consider the activation of different layers and not the original image when calculating style and content loss?
11. Why do we consider gram matrix loss and not the difference between images when calculating style loss?
12. Why do we warp images while building a model to generate deep fakes?