Deep Generative Learning

Department of Electrical Engineering
University of Notre Dame, USA

week 18, 19, 20, 21 (updated: April 11, 2024)

Introduction

- **Deep Generative Learning** has recently generated considerable attention due to applications such as DALLE-2 and ChatGPT.
- Both of these applications can take a natural language prompt and produce a meaningful response.

Prompt: "Write a 75 word summary of what ChatGPT is."
Response: *ChatGPT is a large language model developed by OpenAI. It is trained on a massive dataset of text and is able to generate human-like responses to a wide range of prompts. It can be used for a variety of tasks such as language translation, text summarization, and conversation generation. It has been trained on a diverse set of internet text and is capable of understanding and generating text in a variety of languages and styles.*

ChatGPT Prompt and Response

Prompt: "Teddy bears working on new AI research underwater with 1990's technology"

DALL-E 2 Prompt and Response

Introduction

- Deep generative learning is a significant departure from the earlier supervised learning problems considered in prior chapters.
- Supervised learning results in *discriminative models* since it trains models that are used to discriminate whether a given input lies in a given class or not.
- Generative learning, on the other hand builds models that generate new samples whose distribution match that of a given collection of training inputs.
- Recall that the system in our learning-by-example problem *input* samples, x, that are drawn from an unknown probability distribution $F_{\mathbf{x}}(x)$. An observer would then create a *target*, y, by drawing from the conditional distribution $Q(y|x)$.
- The learning by example problem built models for $Q(y|x)$. In generative learning, we are building models for $F_{\mathbf{x}}(x)$.

Introduction

- There are several reasons why learning the input distribution, $F_{\mathbf{x}}(x)$, might be useful.
- Since the generative model creates samples that were not in the original dataset, it can be used to augment the original dataset in a way that is more powerful than the earlier transformation-based data augmentation schemes for CNNs.
- Generative models can also be used to identify fundamental features in the input data, thereby providing reduced order representations for input samples. We can use these reduced order representations to explore the input data in a way that allows us to generate new datasets that are *biased* for or against certain features in the original dataset.
- This last part is particularly important for creating "deep fakes" using ML and also for addressing issues of fairness and privacy in surveillance, social and medical systems.
- Finally, generative models can also be used to more quickly identify inputs that are inconsistent with the dataset. This last part is useful in flagging outliers

Introduction

- Until a few years ago, discriminative models were the main driver in deep learning.
- This is changing due to recent advances in generative adversarial networks (GAN) (Goodfellow 2014), generative pre-trained transformers (GPT) (Radford 2018) and diffusion models (Ho 2020).
- These models architectures are the drivers behind applications that, at least to the untrained eye, appear to pass the Turing test.
- As of the writing of these lectures (2023), generative learning has come to be seen as the next major driver of deep learning technologies due not only to its technological breakthroughs but also due to the way it has penetrated the life of lay society.

Text Generation using GPT

- The NLP lectures showed how one can use RNN's or Transformers to predict the next words or next few words in a text fragment.
- This text fragment is called a *prompt*. For instance, if the prompt is "the cat is on the", and the model was trained on a database consisting of the name of Broadway musicals and plays, then the model would response might be "hot tin roof", followed with an attribution to the playright Tennessee Williams.
- Any model that can model that learns the probability of the "next word" in a text prompt is called a *language model*. A language model captures the statistical structure of the language's *latent space*.
- Once you have trained such a language model, you can *sample* from it to generate new sequences or sentences. This is done by feeding the model an initial string of text (the prompt) and asking it to generate the next character, word, or phase.

Text Generation using GPT

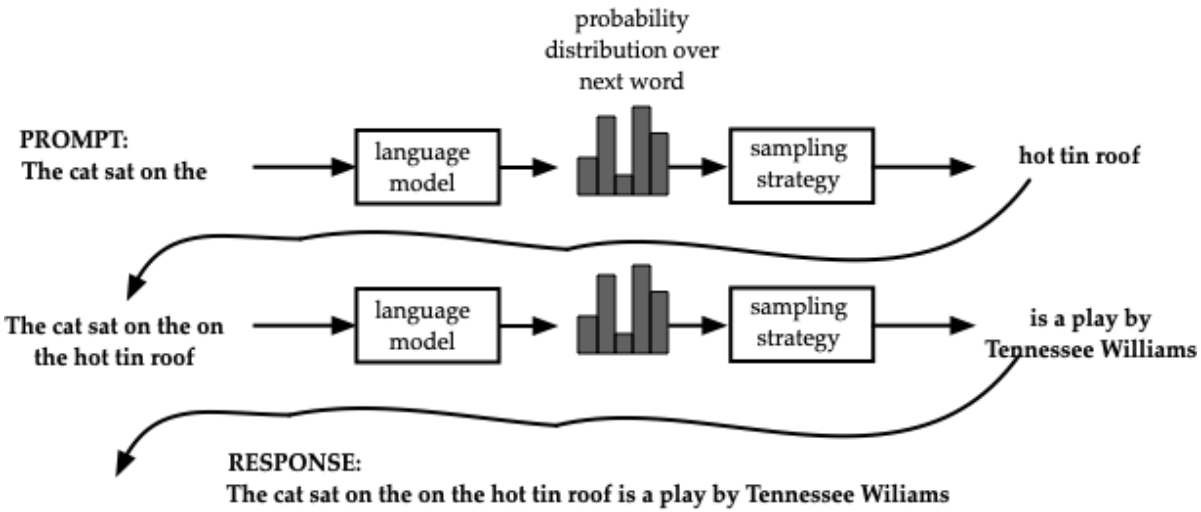


Figure: The process of word-by-word text generation using a language model

- That word or phrase is then added back to the model's input data and we repeat the process.
- This loop allows one to generate sentences (responses) of arbitrary length that reflect the structure of the data on which the model was trained.
- These sentences almost look "human-like" in their responses, since they were trained on natural language fragments.

Text Generation using GPT

- When generating text, the way one chooses the next token is critically important.
- A naive approach would use a *greedy sampling* strategy in which the model always selects the next most likely token. The greedy approach, however, often generates text that is predictable and may not look like coherent language.
- A more useful sampling strategy would select the next word in a probabilistic fashion. This is done by sampling from the probability distribution for the next word.
- Sampling probabilistically from the softmax output of the model allows for even unlikely words to be sampled from time to time, thereby generating more interesting reading sentences.

Text Generation using GPT

- In practice, we control the amount of randomness in the sampling process.
- If there is too much randomness, then the sentences become nonsensical. If there is too little, then the sentences because predictable.
- We control randomness by introducing a parameter called the *softmax temperature*.
- In particular, let $f(x)$ denote the original density and let T denote the softmax temperature. Then the new distribution, $g(x)$ would be

$$g(x) = e^{\log(f(x))/T}$$

- Higher temperatures result in sampling distributions of higher entropy producing more surprising and unstructured sentences, whereas lower temperature results in less randomness and more predictable generated data.

Text Generation with GPT

- We built a transformer model for sequence-to-sequence learning.
- We trained this model by feeding a source sequence (sentence) into a transformer encoder and then fed both the encoded sequence and target sequence into a transformer decoder.
- The decoder was trained to predict the next word in the input sentences.
- In training this model, we used a callback to generate text using a range of different softmax temperatures after every epoch.
- This allows us to see how the generated text evolves as the model begins to converge, as well as the impact in the sampling strategy. We will seed our training with the text prompt, "This movie" so that all of our generated reviews start with this phrase.

Text Generation with GPT

- **temperature = 0.2:** This movie film moved attempts far from between comedy situations central steve plays west his ultimately affect suit gives key the approach movie to filled progress life from adventures political humor tragedy humor violence pathos etc tolerable rookie personalities comedy cinematography ball original story music telling with how nice flooded a hollywood
- **temperature = 0.5:** This movie movie is at excellent truth funny [UNK] it is wasnt simply in boring fact history i i thought hated it it when was i over started [UNK] playing i it mean a everything tv like look this nothing movie can helicopters make and tricks better from then horrible on actors
- **temperature 1.0:** This movie movie sucks was well bad into the town movie while why one did cant you be get frustrated through when so you many start people drinking are is hiroshima attempting and laugh just at because how you you invest feel in how some terrible episodes movies of should kurosawa try

Feature Extraction using PCA

- One way of learning the input sample distribution would be to learn a parameterized model of the distribution.
- For example, let $\widehat{X} = \{\widehat{x}\}_{k=1}^{M}$ denote the collection of input samples, $x_k \in \mathbb{R}^n$, that were drawn in an i.i.d. manner from an unknown distribution $F_{\mathbf{x}}(x)$.
- We could use $\widehat{X}$ to construct an empirical distribution function

$$\widehat{F}_{\widehat{\mathbf{x}}}(x) = \frac{1}{M} \sum_{k=1}^{M} \sigma(x - \widehat{x}_k)$$

  where $\sigma : \mathbb{R}^n \to \mathbb{R}$ is a monotone increasing function from 0 to 1.
- We know that this empirical distribution converges almost surely to the true distribution as $M \to \infty$.

Feature Extraction using PCA

- The problem we have is that our input samples, x_k, have a high dimensionality, thereby meaning we would need a exponentially large number of input samples to obtain a good empirical estimate of the true distribution.
- Our dataset samples, however, will not usually distributed across uniformly across all of $\mathbb{R}^n$. In many real-life applications these samples are concentrated about a smooth lower dimensional surface in $\mathbf{R}^n$ called a *manifold*.
- These manifolds would have a lower dimensionality than n and the distribution could therefore be characterized with a smaller set of *latent variables*.
- We could think of these latent variables as fundamental features of the input samples.

Feature Extraction using PCA

- This leads to the following approach for generating samples matching the input sample's distribution.
- We would first identify an *encoder* that maps the input samples onto this lower dimensional latent space and have a decoder that that takes any vector in the latent space and maps it to an input sample.
- This approach is, essentially, a data compression scheme where the encoder compresses the "information" in the input data and the decoder decompresses the latent variable to recover that information.
- Note that the compression step is usually lossy in the sense that some information may be irretreivably lost and hence cannot be recovered when decoding.
- So the main goal is to find the "best" encoder/decoder pair from a given family that minimizes the reconstruction error.

Feature Extraction using PCA

- Principal component analysis (PCA) is one way for identifying the linear features used to encode a set of input samples.
- The goal of PCA is to identify the a basis of vectors to represent the data set so the number of basis vectors is less than the input sample dimension and to do so in a way that minimizes the reconstruction error.
- To describe this approach more precisely, let the input data samples be denoted as $X = \{x_k\}_{k=1}^{M}$ where $x_k \in \mathbb{R}^{n_d}$ is an n_d-dimensional real-valued vector for all k. We can concretely represent X as a matrix

$$\mathbf{X} = \begin{bmatrix} x_1 & x_2 & \cdots & x_M \end{bmatrix}$$

whose columns are the data samples, x_k.

Feature Extraction using PCA

- Now let $\mathbf{P}$ be a linear transformation from $\mathbb{R}^{n_d}$ to $\mathbb{R}^{n_e}$ where $n_e < n_d$. If we then look at

$$\mathbf{Y} = \mathbf{P}\mathbf{X}$$

- The columns of matrix $\mathbf{Y} \in \mathbb{R}^{n_e \times M}$ are the projection of the columns of $\mathbf{X}$ onto the lower n_e-dimensional *latent space*.
- In particular, we can view

$$\mathbf{P} = \begin{bmatrix} p_1^T \\ p_2^T \\ \vdots \\ p_{n_e}^T \end{bmatrix}$$

This is a stack of row vectors in which $p_k \in \mathbb{R}^{n_d}$ are seen as an alternative set of basis vectors for the original input data vectors in $\mathbf{X}$.

Feature Extraction using PCA

- Note that the projection of $\mathbf{X}$ through $\mathbf{P}$ may lose information because the dimensionality of the latent vectors is less than that of the original data vectors.
- The covariance matrix of $\mathbf{Y}$ is defined as

$$\mathbf{C_Y} = \frac{1}{M}\mathbf{Y}\mathbf{Y}^T = \mathbf{P}\left(\frac{1}{M}\mathbf{X}\mathbf{X}^T\right)\mathbf{P}^T = \mathbf{P}\mathbf{C_X}\mathbf{P}^T$$

  where $\mathbf{C_X} = \frac{1}{M}\mathbf{X}\mathbf{X}^T$ is the covariance matrix of the original data matrix, $\mathbf{X}$.
- Ideally, we want the rows of $\mathbf{P}$ to be orthogonal vectors. If this is the case then $\mathbf{C_Y}$ is a diagonal matrix and we say that the rows of $\mathbf{P}$ are *principal components* of $\mathbf{X}$.

Feature Extraction using PCA

- Note that $\mathbf{X}\mathbf{X}^T$ is a symmetric matrix that can de decomposed as $\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$ where Λ is a diagonal matrix consisting of the eigenvalues of $\mathbf{C_X}$ and $\mathbf{V}$ is a matrix of eigenvectors of $\mathbf{C_X}$ arranged as columns.
- We can, therefore, see that if we choose $\mathbf{P}$ to be a matrix whose rows are eigenvectors of $\mathbf{C_X}$ then

$$\mathbf{C_Y} = \mathbf{V}^T\mathbf{V}\mathbf{\Lambda}\mathbf{V}^T\mathbf{V} = \mathbf{\Lambda}$$

- We have just shown that the principal components of $\mathbf{X}$ are the eigenvectors of $\mathbf{C_X}$.
- It is common to use singular value decompositions (SVD) to compute the principal components. SVDs represent the most numerically stable way of computing such decompositions for large data matrices.

Feature Extraction using PCA

- For any $m \times p$ matrix, $\mathbf{Q}$, one can prove that there exist $m \times m$ and $p \times p$ unitary matrices $\mathbf{U}$ and $\mathbf{V}$ and a real $r \times r$ diagonal matrix $\mathbf{\Sigma}$ such that

$$\mathbf{Q} = \mathbf{U} \begin{bmatrix} \mathbf{\Sigma} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{V}^T$$

- The matrix $\mathbf{\Sigma}$ has the form

$$\mathbf{\Sigma} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$$

  where $\sigma_i \geq \sigma_{i+1}$ for $i = 1, \ldots, r-1$ and $r \leq \min(m, p)$ is the rank of matrix $\mathbf{Q}$.

- The triple, $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$ is called the *singular value decomposition* of $\mathbf{Q}$. This decomposition is unique and σ_1 to σ_r are called the non-zero singular values of $\mathbf{Q}$.

- These non-zero singular values are also the positive roots of the non-zero eigenvalues of $\mathbf{Q}^T\mathbf{Q}$.

Feature Extraction using PCA

- To see how this relates back to PCA, let us consider a data matrix $\mathbf{X}$ whose columns are the data sample vectors that have been centered with respect to the dataset's mean.
- Recall that $\mathbf{C} = \frac{1}{M}\mathbf{X}\mathbf{X}^T$ is the covariance matrix of the data matrix. We know the principal components are the eigenvectors of $\mathbf{C}_\mathbf{X}$.
- Now consider the SVD of the data matrix $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. Let us express the covariance matrix of $\mathbf{X}$ in terms of its SVD

$$\mathbf{X}\mathbf{X}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T$$

- We can therefore conclude that

$$\mathbf{C}_\mathbf{X} = \frac{1}{M}\mathbf{U}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$$

  where $\mathbf{\Lambda}$ is a diagonal matrix whose diagonal elements are $\lambda_i = \frac{\sigma_i^2}{M}$.
- Since $\mathbf{U}$ is a unitary matrix (i.e. $\mathbf{U}^T\mathbf{U} = \mathbf{I}$) we can readily see that

$$\mathbf{C}_\mathbf{X}\mathbf{U} = \mathbf{U}\mathbf{\Lambda}$$

Feature Extraction using PCA

- This means that the columns of $\mathbf{U}$ are the prinicipal components.
- Since we defined the PCA transformation $\mathbf{P}$ so its rows were the principal component vectors, we have $\mathbf{P} = \mathbf{U}^T$. If we then look at transforming all data points into the PCA coordinates we have

$$\mathbf{Y} = \mathbf{PX} = \mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{\Sigma}\mathbf{V}^T$$

- This last result is used in the following example where we use the SVD of the data matrix to do a PCA of the Fisher iris dataset.

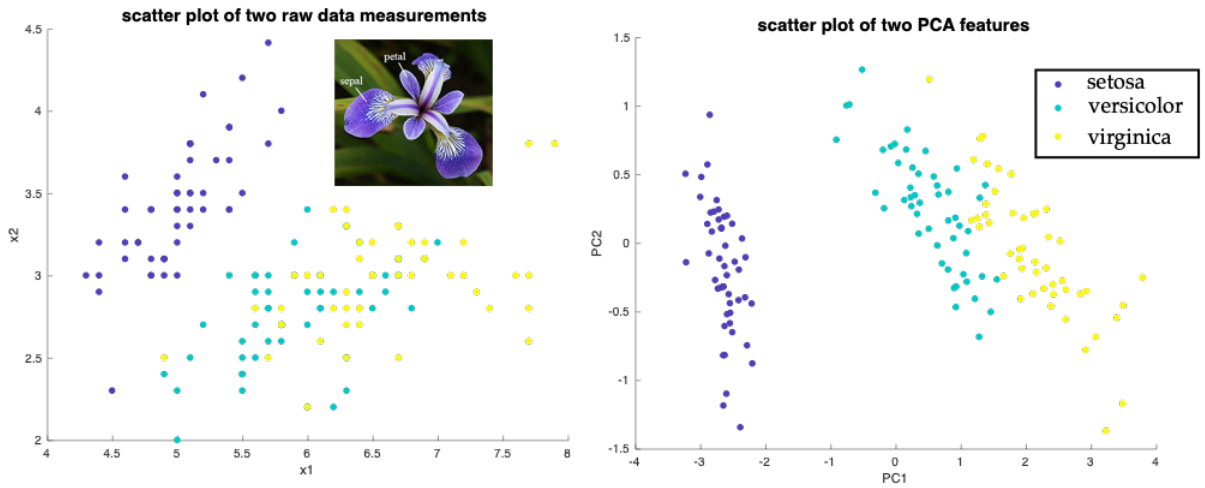


Figure: Fisher Iris Data set, PCA analysis

Feature Extraction using PCA

- We used the SVD to find the principal components of Fisher's iris dataset. This dataset has 150 length and width measurements on the petals and sepals of 3 species of irises.
- I used MATLAB to load the iris dataset, scatter plot the first two features, compute the SVD of the data matrix and then find the first right singular vectors with the largest singular values.
- The preceding figure shows that the data for the 3 classes are not well separated. The scatter plot for the two dominant principal components on the right show a much cleaner separation between the 3 classes.

```
load irisdata.txt
X = irisdata(:,1:4)';                  %150 measurements of length 4
spec = irisdata(:,5)';                %class labels
n = size(X,2);
figure(1)
scatter(X(1,:),X(2,:),30,spec,'filled')
xlabel("x1");ylabel("x2");
title("scatter plot of 2 raw measurements");

Xmean = mean(X,2);              %find mean
A = X - Xmean*ones(1,n);    %center the data
[U,S,V] = svd(A, 'econ')        %find SVD of centered data matrix
sigma = diag(S);
C = S(1:2,1:2)*V(:,1:2)';       %principal components of each data point
figure(2);
```

Autoencoders

- An autoencoder is a deep learning model with an encoder-decoder architecture that takes an input image or text, encodes it over a space of latent variables, and then decodes that latent vector into the original image.
- We can see the encoder as compressing the high dimensional input data onto the lower dimensional latent embedding vector.
- The decoder then decompresses that latent vector back into the original image.

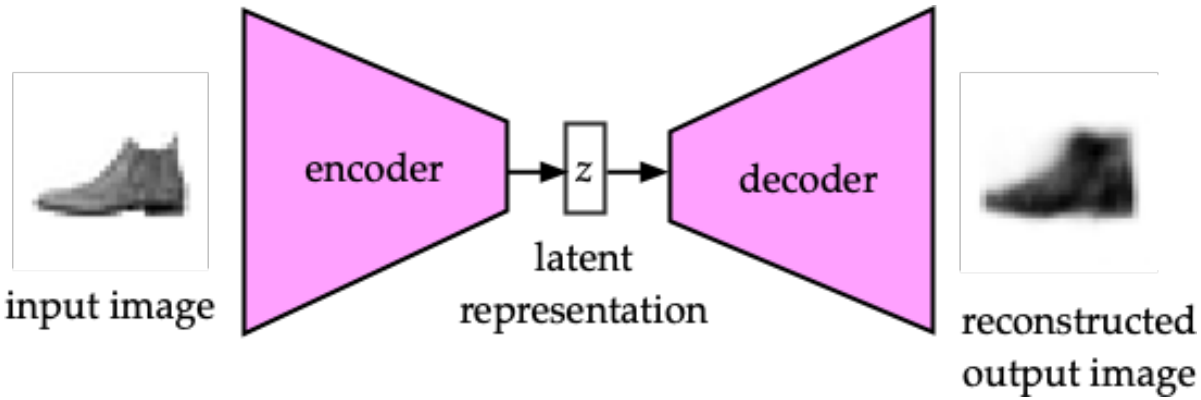
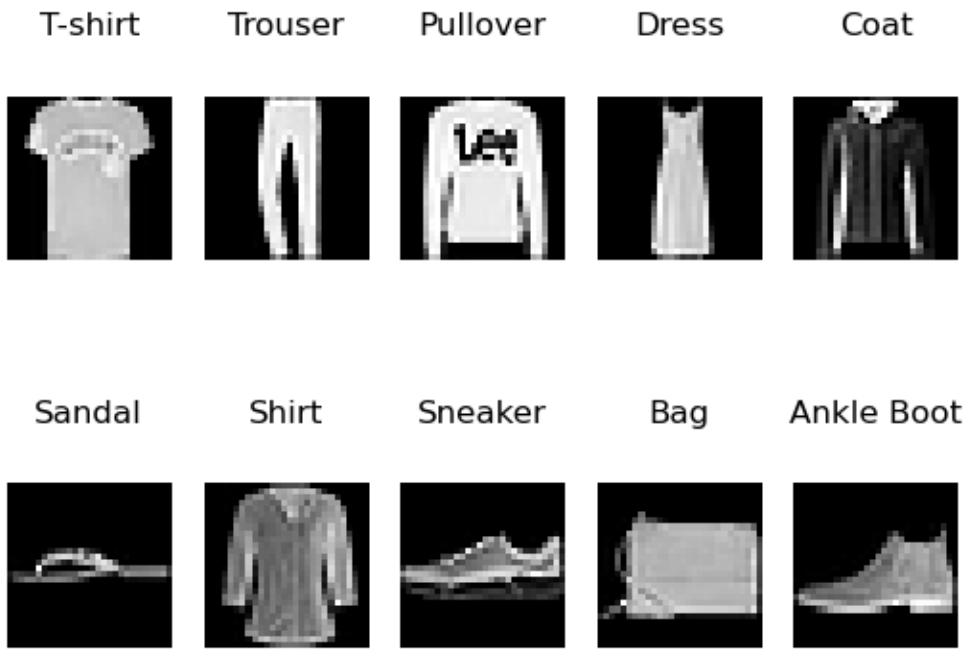


Figure: Autoencoder architecture

Autoencoders

- the autoencoder is trained to *reconstruct* the input image, so that we train this with the input and target sample being the same image.
- The latent vector z is a vector in a low dimensional vector space, $\mathbb{R}^{n_e}$.
- If the embedding dimension, n_e, is 2 or 3, we can easily visualize the points in the dataset and see how various images are "close" or "far" apart. Let us do this for the fashion MNIST database .



Autoencoder

- We are now going to load the Fashion MNIST dataset and then construct an autoencoder based on CNNs. The dataset is included in TensorFlow, so we can load it as follows.
- We are going to retype the pixel data from `uint8` to `float32` and normalize it so it takes values between 0 and 1. We will then zero pad and expand the shape of the input images so they are all $(32, 32, 1)$. This is done because our model expects a rank-3 tensor.
- We declare the encoder model formed from three 2D convolutional layers, that use strides to down sample the image to a shape of $(4, 4, 128)$.
- We will select a latent space, n_e, of 2. So the encoder's output is simply a two dimensional vector.

Autoencoder

```
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]  # the decoder will need this!

x = layers.Flatten()(x)
encoder_output = layers.Dense(EMBEDDING_DIM, name="encoder_output")(x)

encoder = models.Model(encoder_input, encoder_output)
encoder.summary()
```

Autoencoder

The decoder is a mirror image of the encoder. It upsamples the latent vector from a shape of (2) to a shape of $(32, 32, 1)$. The 2-d latent vector is expanded through a dense layer to a shape of (2048) and then reshaped to $(4, 4, 128)$. We then use three transposed 2D convolutional layers to upsample the spatial dimensions until we get to the desired output shape.

```
decoder_input = layers.Input(shape=(EMBEDDING_DIM,), name="decoder_input")
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input)
x = layers.Reshape(shape_before_flattening)(x)
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation="relu", padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation="relu", padding="same"
)(x)
decoder_output = layers.Conv2D(
    CHANNELS,
    (3, 3),
    strides=1,
    activation="sigmoid",
    padding="same",
    name="decoder_output",
)(x)

decoder = models.Model(decoder_input, decoder_output)
decoder.summary
```

Autoencoder

We now build the autoencoder, compile it with Aam, and train for epochs. We visualize how the autoencoder maps the dataset images to the latent space.

Autoencoder

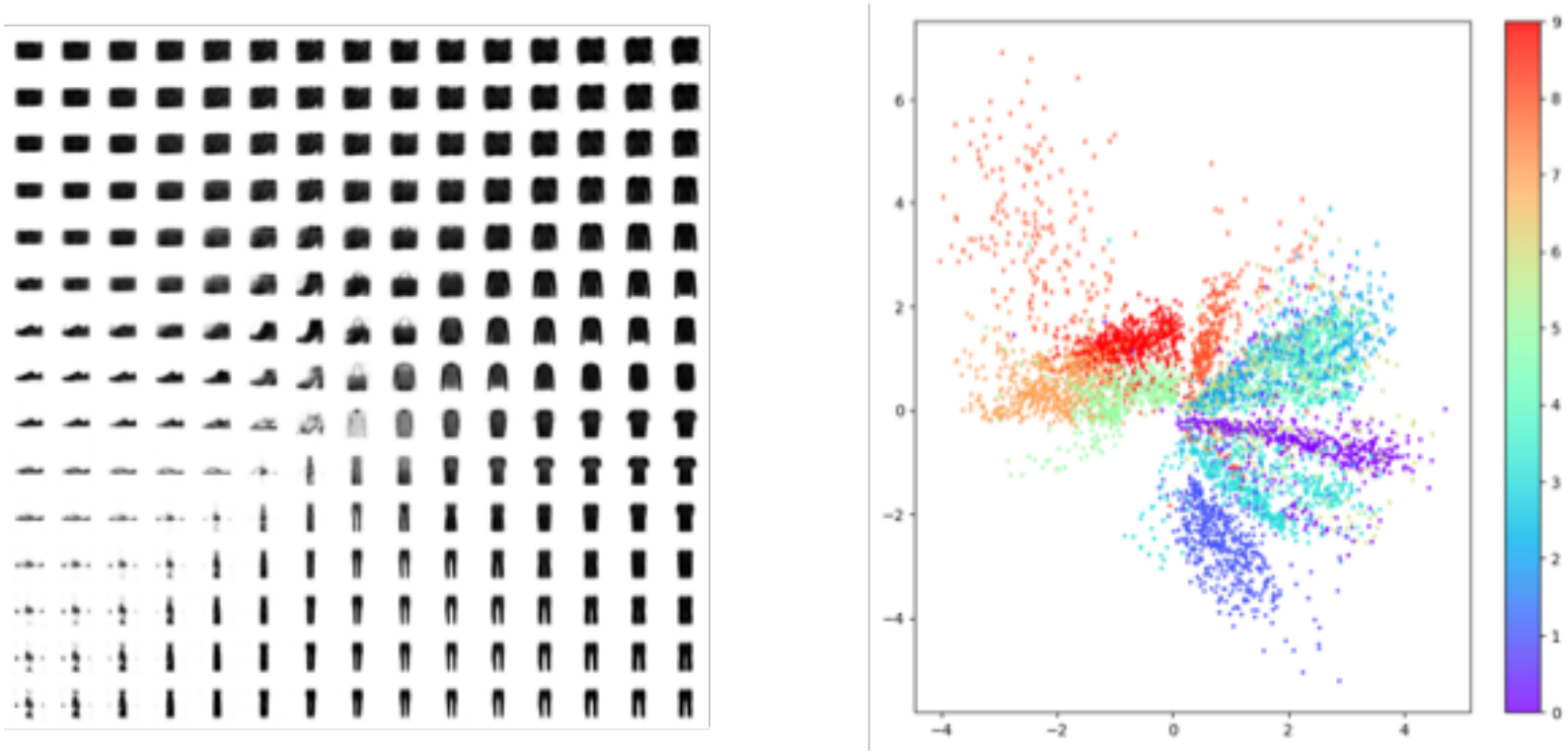


Figure: Fashion-MNIST Latent Space (right) embedding of all dataset samples (left) reconstructed images whose latent variables are regularly sampled in the 2-d latent space

Autoencoder

- The reconstructions were obtained by regularly sampling the latent space and then running the latent vector through the decoder.
- What we see is that as we move across the space, that the images "morph" in a somewhat predictable manner.
- In practice, this aspect of the autoencoder can be used to create a biased set of samples that ignore some features in the original dataset and accentuate others.
- Notice that some clothing items are represented over rather small areas of the latent space, whereas other clothing items cover a larger area.
- notice is that there are large empty areas in the latent space that do not represent any actual clothing item.
- These are well known problems with autoencoders that we address through the varational autoencoder (VAE).

Variational Autoencoder

- Varational autoencoders (Kingma 2013) map the inputs to a normal distribution centered at a point in the latent space.
- Because a normal distribution is completely characterized by its first two moments, we really only need to map the encoder's output to two different outputs; the mean vector and the variance matrix.
- In general, however, we assume the normal distribution has a diagonal covariance matrix, so that we really only need to have the encoder output two vectors, one for the mean of the distribution and the other for the diagonal of the covariance matrix.
- Since variance values are non-negative, however, we will find it more convenient to map to the log of the variance since this has values between $-\infty$ and ∞.
- This range of mappings fits more nicely with our neural network models.

Variational Autoencoder

As a result the variational autoencoder architecture changes so the latent variables between the encoder and decoder now are the mean, `z_mean` and the log of the variance, `z_log_var` of the distribution.

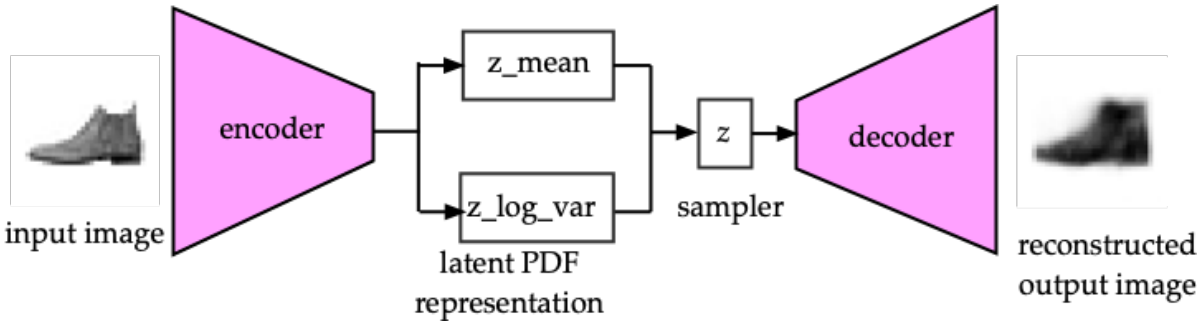


Figure: VAE Model Architecture

Variational Autoencoder

- We can build the VAE model in much the same way we did for the autoencoder. There are, however, some significant differences.
- The first major difference is seen in the encoder where the encoder now has three possible outputs, `z_mean`, `z_log_var`, and `z`.
- The extra output `z` is a randomly drawn sample from the distribution defined by `z_mean` and `z_log_var`.
- This sampled output, `z`, is generated by a new `Sampling` layer class that we define at the top of the following script.
- Decoder is similar to that in Autoencoder

Variational Autoencoder - Encoder

```
IMAGE_SIZE = 32
EMBEDDING_DIM = 2

#Sampling Layer
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon

# Encoder
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]  # the decoder will need this!

x = layers.Flatten()(x)
z_mean = layers.Dense(EMBEDDING_DIM, name="z_mean")(x)
z_log_var = layers.Dense(EMBEDDING_DIM, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])

encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder")
```

Variational Autoencoder

- The other big change from our earlier encoder is our choice of loss function. The autoencoder used MSE or binary crossentropy for the loss.
- The VAE, however, needs to use a loss function that helps the latent layer learn the mean and variance of the normal distributions that each input maps to.
- This is accomplished by regularizing the loss function so it penalizes means and variance that are not close to a unit variance normal distribution.
- This is accomplished by taking our loss to be the sum of the *reconstruction loss* (measured as before by the MSE between the input and the reconstruction) and an additional weighted loss term that measures the "distance" of the current distribution's mean/variance against a zero-mean unit variance normal distribution.

Variational Autoencoder

- This second measure is usually taken to be the Kullback-Leibler (KL) divergence. The KL divergence for two densities p and q is defined as

$$\text{KL divergence} = D_{KL}(p, q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

- This measure equals zero when $p(x) = q(x)$. In our case, p is the normal distribution $N(\texttt{z\_mean}, \texttt{z\_log\_var})$ and q is the normal distribution $N(0, 1)$.
- So the KL divergence is

$$\texttt{kl\_loss} = -\frac{1}{2} \sum_i (1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2)$$

  where $\texttt{z\_mean} = \mu$ and σ is the diagonal of the covariance matrix.
- The actual loss function used to train the VAE adds the KL divergence to the reconstruction loss

$$\text{VAE Loss} = \text{reconstruction loss} + \beta \times \text{KL divergence}$$

Variational Autoencoder

```
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def call(self, inputs):
        """Call the model on a particular input."""
        z_mean, z_log_var, z = encoder(inputs)
        reconstruction = decoder(z)
        return z_mean, z_log_var, reconstruction

    def train_step(self, data):
        """Step run during training."""
        with tf.GradientTape() as tape:
            z_mean, z_log_var, reconstruction = self(data)
            reconstruction_loss = tf.reduce_mean(
                BETA
                * losses.binary_crossentropy(
                    data, reconstruction, axis=(1, 2, 3)
                )
            )
            kl_loss = tf.reduce_mean(
                tf.reduce_sum(
                    -0.5
                    * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                    axis=1,
                )
            )
            total_loss = reconstruction_loss + kl_loss

        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
```

```
def test_step(self, data):
        """Step run during validation."""
        if isinstance(data, tuple):
            data = data[0]

        z_mean, z_log_var, reconstruction = self(data)
        reconstruction_loss = tf.reduce_mean(
            BETA
            * losses.binary_crossentropy(data, reconstruction, axis=(1, 2, 3))
        )
        kl_loss = tf.reduce_mean(
            tf.reduce_sum(
                -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                axis=1,
            )
        )
        total_loss = reconstruction_loss + kl_loss

        return {
            "loss": total_loss,
            "reconstruction_loss": reconstruction_loss,
            "kl_loss": kl_loss,
        }
```

Variational Autoencoder

We are now in a position to create the VAE model using our preceding VAE class object.

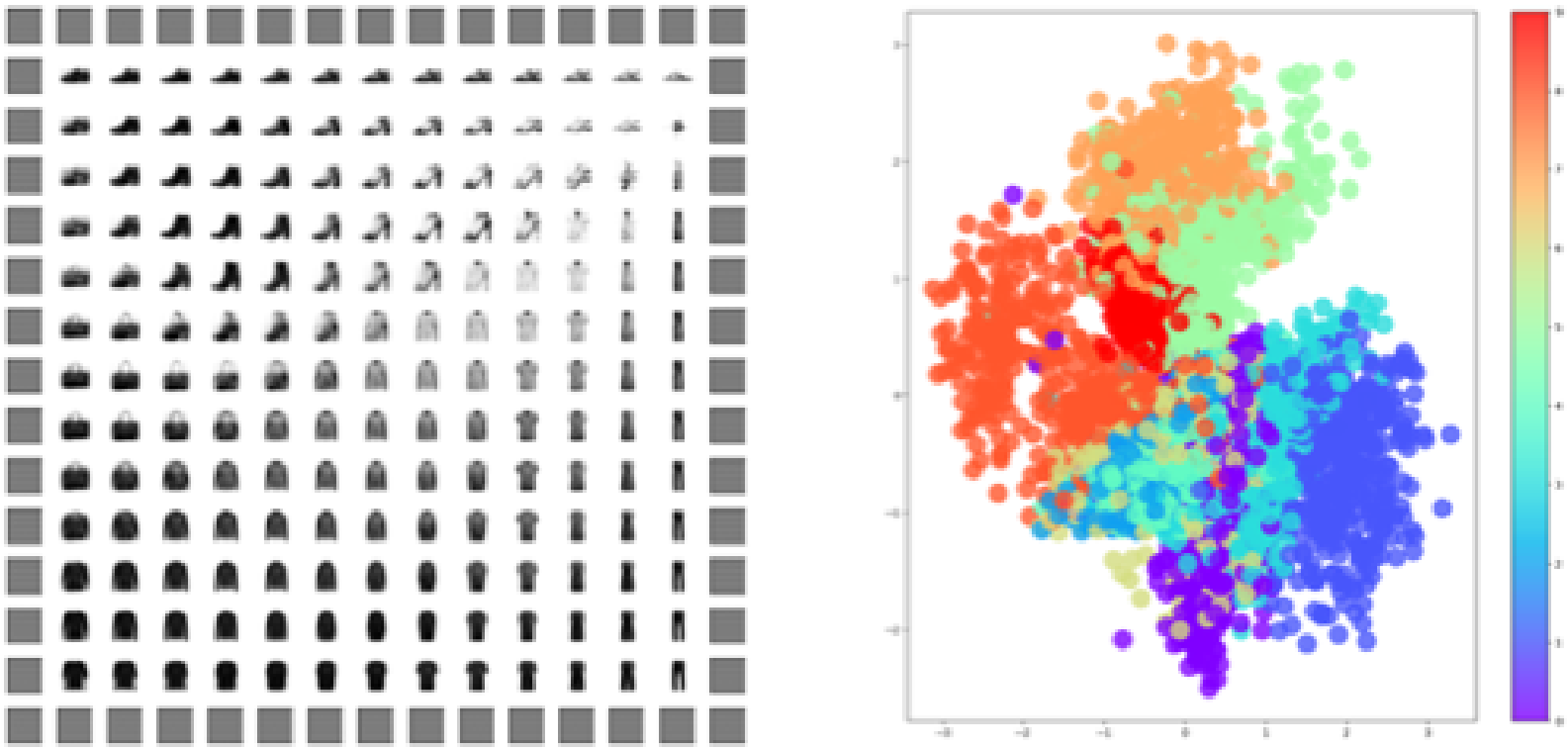


Figure: VAE latent space for fashion MNIST (left) reconstructions (right) location of sampled points

Variational Autoencoder

- The preceding examples used a low dimensional latent space to help you visualize how vectors in the latent space give rise to image reconstructions.
- We now turn to an example using a higher dimensional latent space to demonstrate how we can "explore" the latent space and create or modify how a given input image.
- This example uses the CelebFaces Attributes (CelebA) dataset with over 200,000 color images of celebrity faces, each annotated with various label (e.g. smiling, blonde,...).
- The images in this dataset are RGB images of shape (64,64,3). We will consider a latent space of 200, rather than 2.

Variational Autoencoder

- We construct a variational autoencoder in much the same way as we did for the Fashion MNIST dataset.
- Because the inputs are larger, however, we will need a more deeper model. In particular, we will have 5 convolutional layers with batch normalization that take the input shape (64,64,3) down to $(2, 2, 64)$.
- We then map this to two layers of shape $(200,)$ for `z_mean` and `z_log_var` that are passed through a sampling layer producing the actual output latent vector `z`.
- We then train the VAE model for 10 epochs. This training takes significantly longer because the model is deeper with over 500,000 weights.

Variation Autoencoder

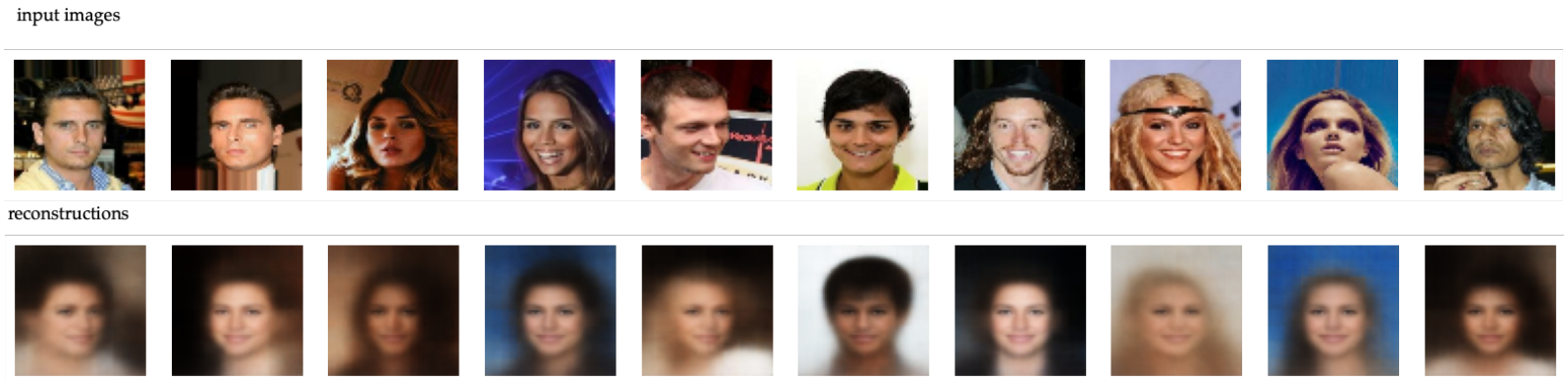


Figure: (Top) input images from CelebA dataset (Bottom) reconstructed images generated by trained VAE.

Variational Autoencoder

- One benefit of the latent space is that because it is a vector space we can do basic vector arithmetic on it.
- For example, suppose we want to take an image of somebody who looks sad and given them a smile. To do this, we first need to find a vector in the latent space that points in the direction of increasing smile.
- Adding this vector to the encoding of the original image in the latent space will give a new latent vector which when decoded should generate a smiling image.
- To find this "smile" vector, we return to the CelebA dataset and look at the labels.
- Each image in the dataset is labeled with 40 different attributes such as "wearing hat", "wearing lipstick", "young", "smiling", etc.

Variational Autoencoder

- To get the "smiling" direction, for example, we would first compute the average position of encoded images in the latent space with the "smiling" attribute and subtract the average position of encoded images that do not have the attribute "smiling".
- The resulting vector can then be taken as the "smiling" director, what we will call its `feature_vector`.
- We can then generate new latent vectors from the original one, `z` as

$$\texttt{z\_new} = \texttt{z} + \texttt{alpha} * \texttt{feature\_vector}$$

where `alpha` controls how much we want to shift our original vector, `z`, along the feature direction.

Variational Autoencoder

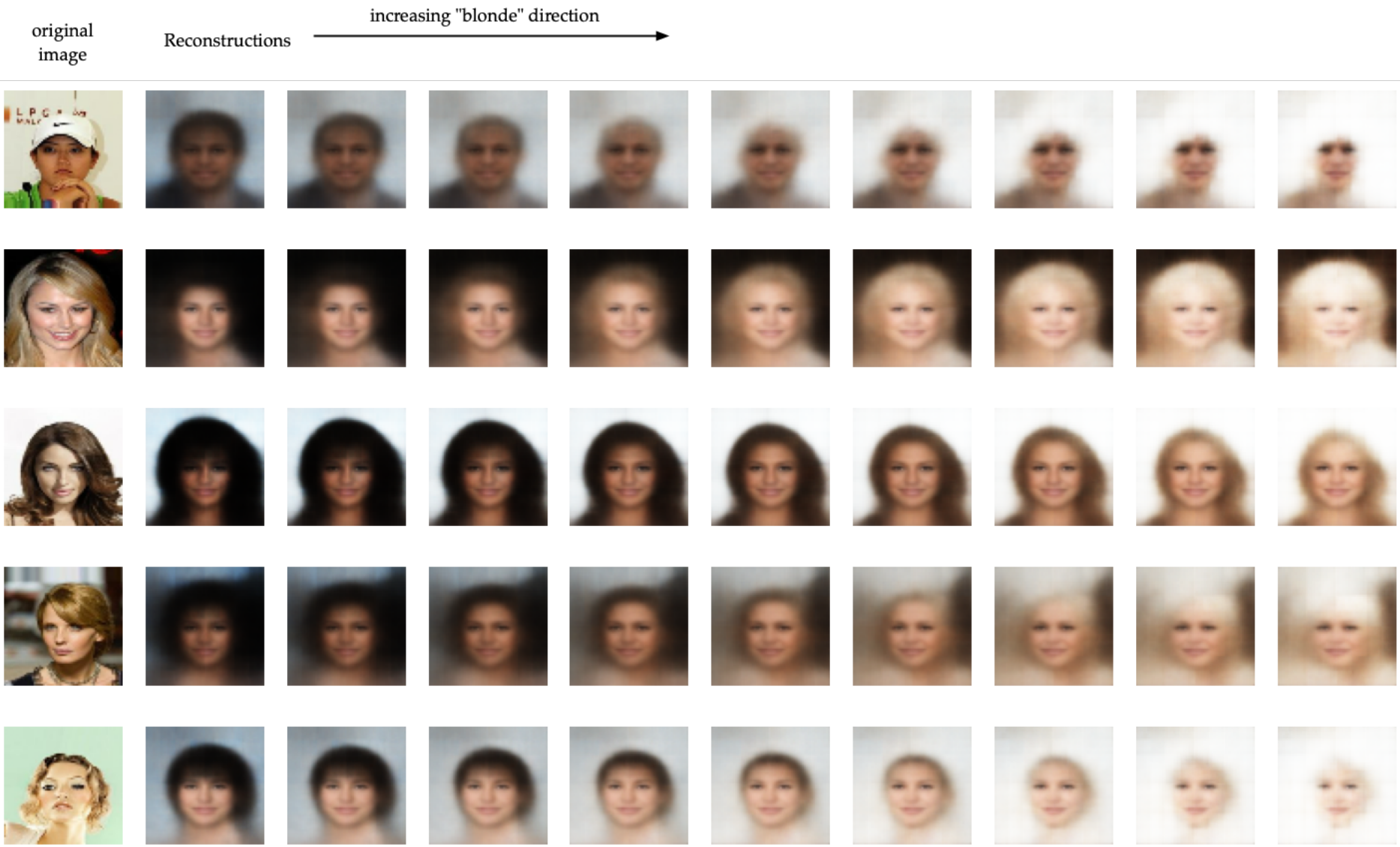


Figure: Transformation of various input images into blondes

Generative Adversarial Networks

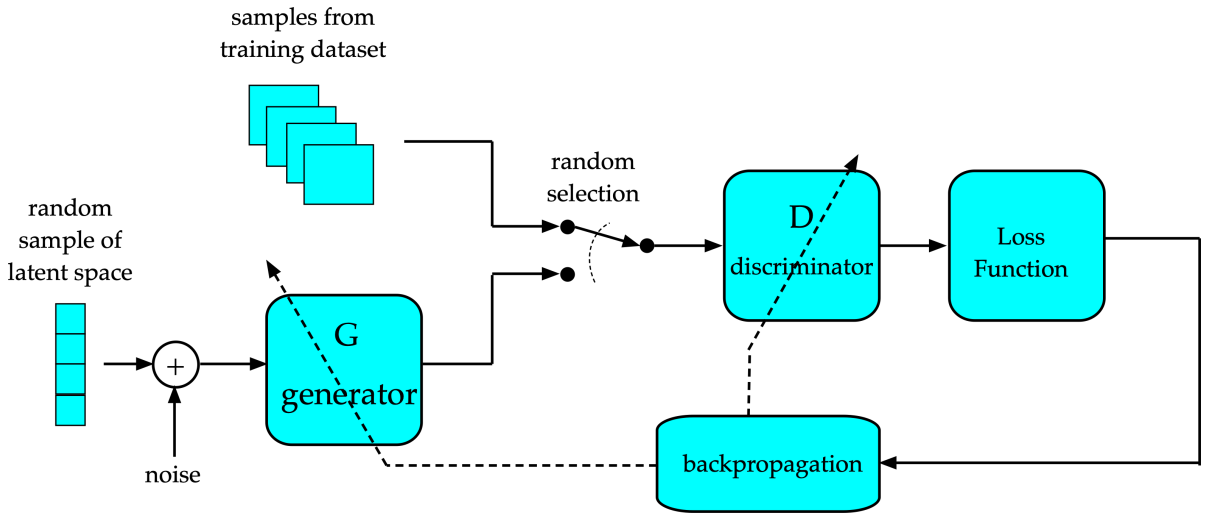
- Generative adversarial networks (GAN) (Goodfellow 2014) were first proposed in 2014.
- Their introduction stimulated a great deal of interest in generative learning and led to some of generative learning's most impressive accomplishments.
- The GAN takes a *game theory* approach to learning how to generate new samples from the system's input distribution, $F_{\mathbf{x}}(x)$.
- Game theory envisions two players who take actions that further their own self interests while having the additional impact of interfering with the competing player's game performance.
- The basic idea is that these two players struggle until they reach a point from which neither player can gain an advantage over the other. Such points are called *Nash equilibria*.

Generative Adversarial Networks

- The GAN's two players are a *generator* and *discriminator* model.
- The generator generates samples that may be seen as "fake" copiesof inputs in the original dataset.
- The generator trains itself so its distribution of "fake" samples matches the distribution, $F_{\mathbf{x}}(x)$, of inputs in the original dataset.
- The other player is a *discriminator* who takes an input from either the generator or dataset and updates itself so it can correctly discriminate between "fake" samples from the generator and "true" samples from the dataset.
- Another way of thinking about this game is that the "generator" learns to create samples that can "fool" the discriminator while the discriminator is learning how to distinguish the generator's fake samples from the true ones.

Generative Adversarial Networks

- At each training step, we randomly select an input to the discriminator from either the training dataset or the Generator, G. The generator's output is obtained from a random input. That random input is obtained by randomly sampling the latent space and then adding noise to the resulting latent vector.
- The discriminator, D, then classifies that input as being either "fake" or "true". We then compute the loss for the discriminator's classification and use that loss to drive backpropagation's update of both the Generator and Discriminator's weights.



Generative Adversarial Networks

- Let us now describe the training process in a more formal manner. Let $P_{\mathbf{z}}(z)$ denote the distribution of a latent vector drawn from the latent space.
- Let $F_{\widehat{\mathbf{x}}}(\widehat{x})$ denote the distribution of samples, $\widehat{x}$, created by the generator. Let $F_{\mathbf{x}}(x)$ denote the true distribution of input samples, x in the dataset.
- We will train the discriminator, D to maximize the accuracy over "real" decision points, x, in the training data by maximizing

$$\mathbb{E}_{x \sim F_{\mathbf{x}}} [\log D(x)]$$

- Meanwhile for a *fake* sample, $\widehat{x}$, created by the generator G using input, z, drawn from $\mathbf{P_z}$ over the latent space, the discriminator will be trained so its outputs a probability $D(G(z))$ that is small for "fake" samples and close to 1 for "true" samples. This suggests we should train our discriminator to maximize

$$\text{Discriminator Accuracy on Fake Data} == \mathbb{E}_{z \sim P_{\mathbf{z}}} [\log(1 - D(G(z)))] \quad (1)$$

Generative Adversarial Network

- So the loss function for the discriminator is our usual binary cross-entropy function

$$L(D, G) = \mathbb{E}_{x \sim F_x} [\log D(x)] + \mathbb{E}_{z \in P_z} [\log(1 - D(G(z)))]$$

  Our training of the discriminator seeks to select discriminator weights that maximize $L_D(D, G)$ for a fixed generator, G.

- During the generator's training, we adjust G's weights to increase the likelihood of D being wrong in its classification of fake samples. This means that we are trying G to minimize equation (1), exactly opposite of what we did when training the discriminator.

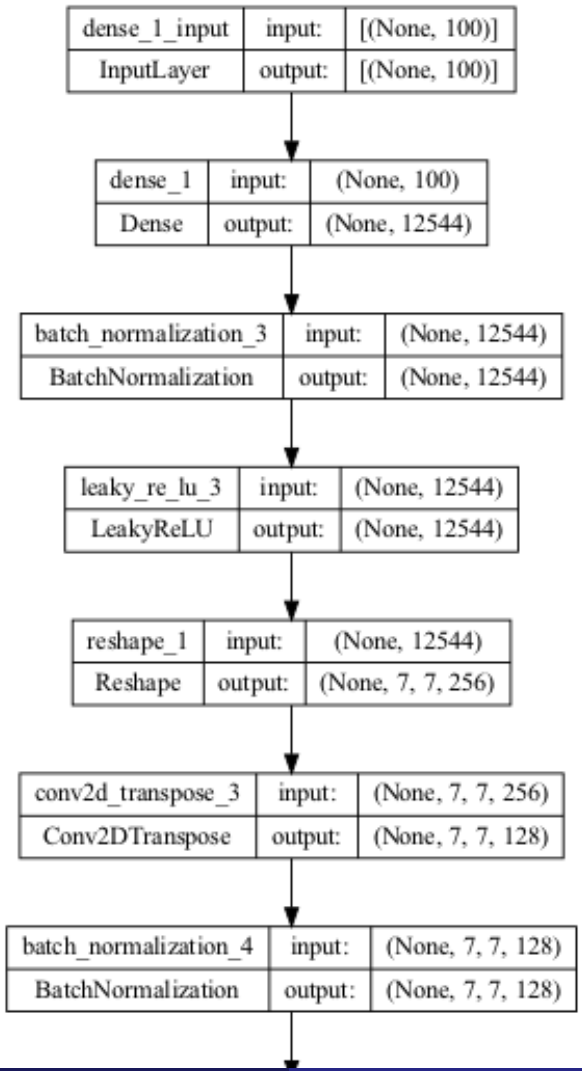
- So training the generator seeks to *minimize* $L(D, G)$ for a given D by selecting the weights of G, whereas training the discriminator seeks to *maximize*, $L(D, G)$ for a given G by selecting the weights of D.

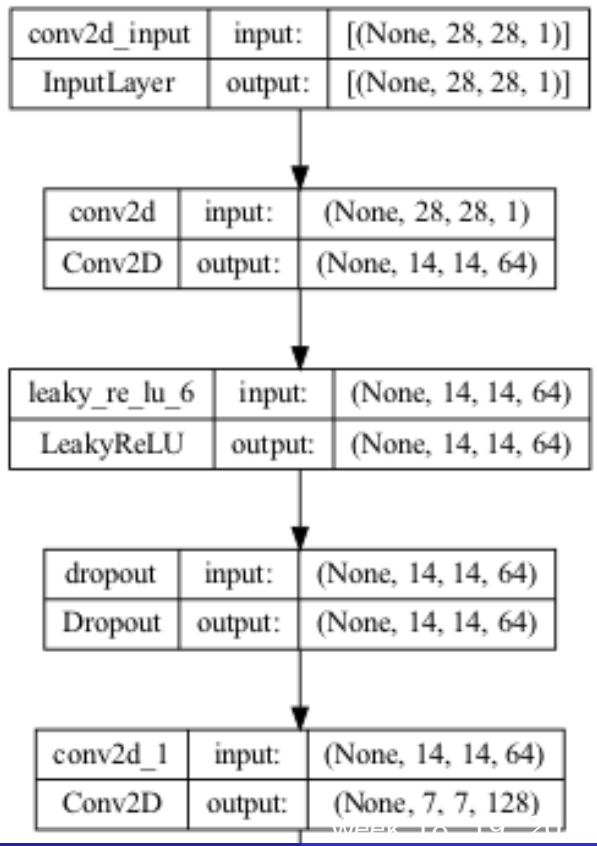
- This training is done in a back and forth manner between D and G until we achieve a *Nash equilibrium*, (D^*, G^*), where

$$L(D, G^*) \leq L(D^*, G^*) \leq L(D^*, G)$$

Generative Adversarial Network

We now demonstrate the construction and training of a deep convolution GAN (DC-GAN) used to generate "fake" images.





Generative Adversarial Network

- We define a separate loss functional and optimizer for each model, since they are trained separately.
- The discriminator's loss function quantifies how well the discriminator is able to distinguish real images from fake images. It compares the discriminator's prediction on the real image to an array of 1's and the prediction on the fake input to zeros. So the discriminator loss function is a binary cross entropy function,
- The generator's loss quantifies how well it is able to trick the discriminator. This loss function is simply the cross entropy function
- We also declare two separate optimizers because we are training two different models. The both use the ADAM optimizer, rather than the RMSprop. The learning rate is usually set very small to help stabilize learning and in some cases the learning rate is changed adaptively.

Generative Adversarial Network

- Training the GAN follows the game theoretic flow and so we have to write a new `train` method for the GAN.
- This method will call the following `train_step` function which uses the GradientTape object to separately compute gradients used in updating the weights of the two models.
- Note that the training-step in this implementation trains the discriminator for one step, and then the generator for one step.
- In many applications, however, it has proven to be better to train the generator for several steps before updating the discriminator.

Generative Adversarial Network

```
# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_varia

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_vari
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.tr
```

Generative Adversarial Network

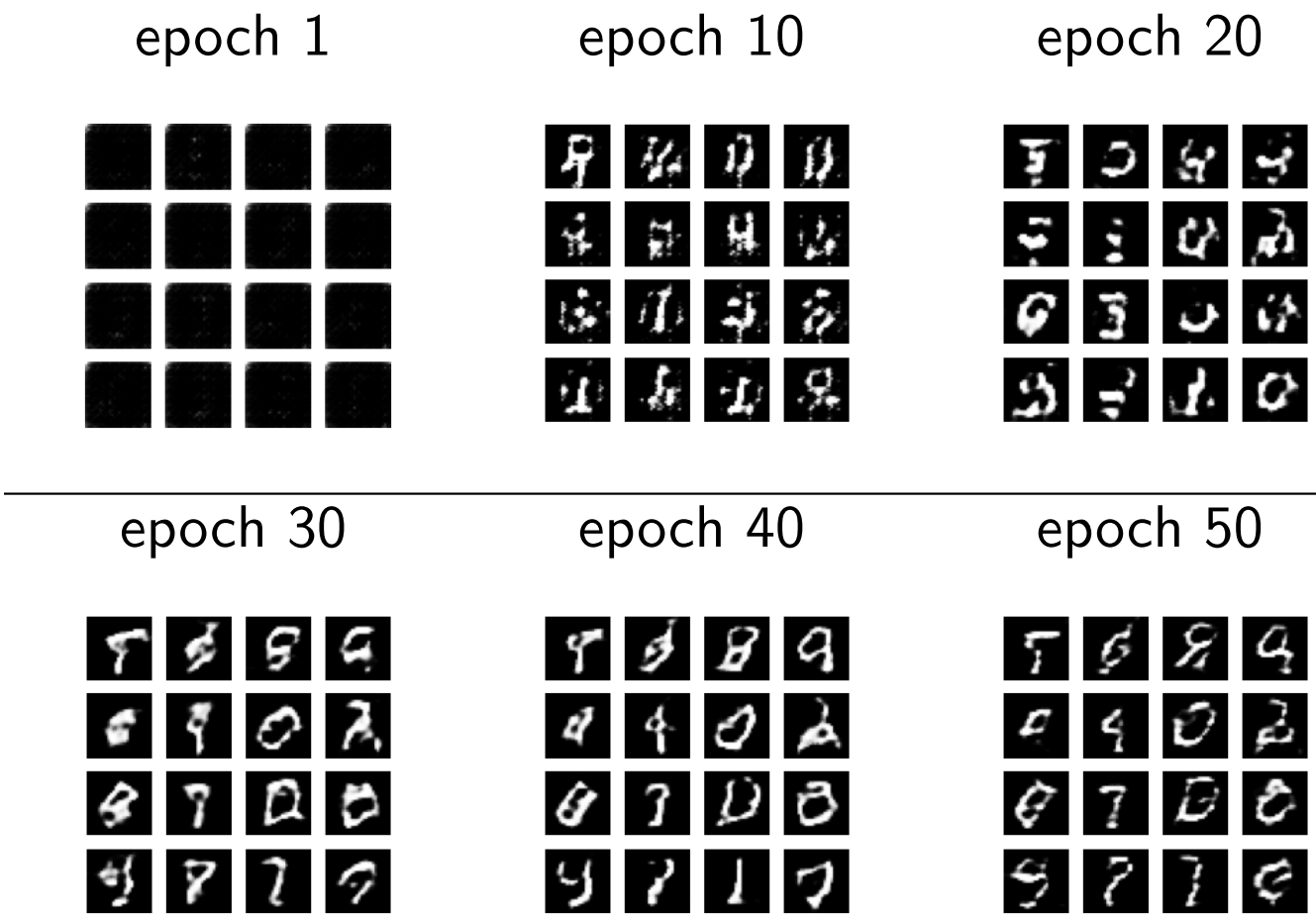
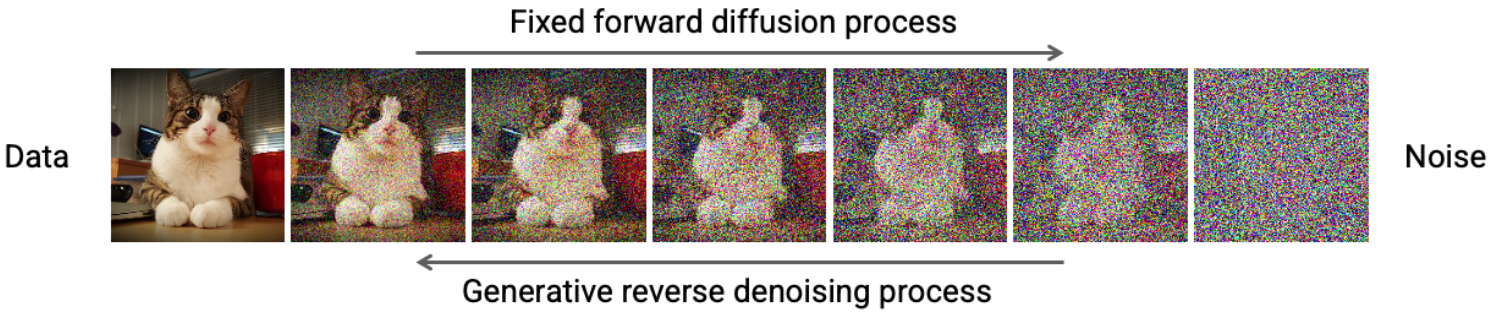


Figure: GAN generated images

Diffusion Models

- Alongside GANs, diffusion models are one of the most influential generative modeling techniques for images.
- Diffusion models now outperform previous state-of-the-art GANs have become the go-to choice for generative modeling engineers in the visual domain.
- For example, OpenAI's DALLE-2 and Google's ImageGen applications for text-to-image generation all use diffusion models to generate the output image after a transformer has decoded the users prompting text.
- The breakthough diffusion model appeared in 2020 trained a diffusion model that rivaled GANs across several datasets.



Diffusion Models

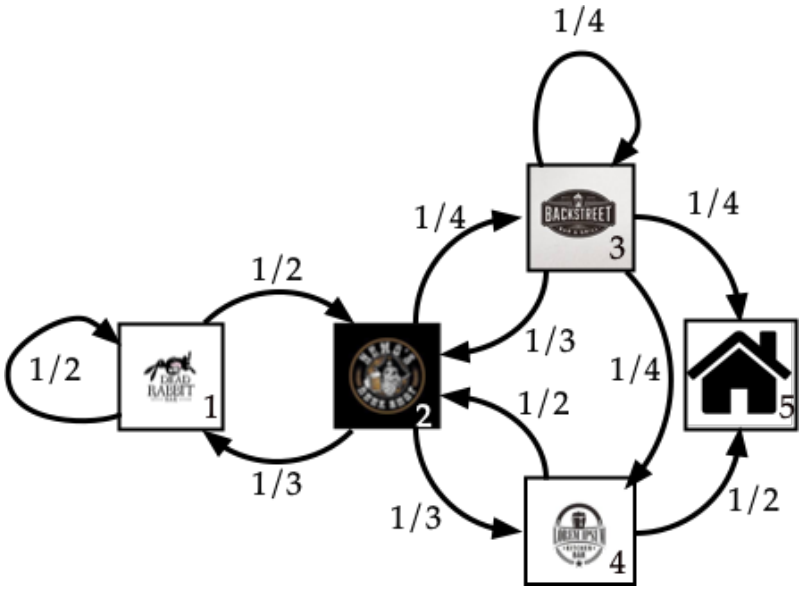
- The core idea behind DDPMs is relatively simple - we train a deep learning model to denoise an image over a series of very small steps.
- If we start from pure random noise, in theory we should be able to keep applying the model until will obtain an image that looks as if it were drawn from the training set.
- The DDPM makes use of two Markov chains to achieve this. There is a *forward chain* that perturbs data to noise and a reverse chain that converts noise back to data.
- The forward chain is usually hand-designed with the goal of transforming any data distribution into a standard Gaussian image.
- The reverse chain undoes the forward chain by learning transition kernels parameterized by deep neural networks. New data points are then generated by first sampling a random vector from the standard Gaussian, followed by ancestral sampling through the reverse Markov chain.

Digression - Markov Chains

- Markov chain is a probabilistic model for trials of a random experiment.
- Let $\{\mathbf{x}_k : k \geq 0\}$ be a sequence of random variables taking values from set $X = \{1, 2, \ldots, n\}$.
- **Markov Property:** For all $k \geq 0$ and any $x, y \in X$ we have

$$\Pr\{\mathbf{x}_{k+1} = y \,|\, k = x\} = \Pr\{\mathbf{x}_{k+1} = y \,|\, \mathbf{x}_k = x\} = p_{yx}$$

- We can represent p_{yx} as a matrix $q_k = \mathbf{P} q_{k-1}$. Markov chain is the ordered tuple $(X, \mathbf{P}, X_0, X_a)$.



Diffusion Models

- We can describe this more formally as follows. Let assume that we have an input $x_0 \in F_{\mathbf{x}}(x)$ that was selected from the training dataset's input distribution.
- The forward Markov chain generates a sequence of random variables, $\{\mathbf{x}k\}_{k=0}^{T}$, with transition kernel $Q(x_t|x_{t-1})$.
- Using the chain rule of probability and the Markov property, we can factor the joint distribution of $x_1, x_2, \ldots, x_T$ conditioned on the initial input, x_0, which we denote as $Q(x_1, \ldots, x_T \,|\, x_0)$ into

$$Q(x_1, \ldots, x_t \,|\, x_0) = \prod_{t=1}^{T} Q(x_t \,|\, x_{t-1}) \tag{2}$$

- In DDPMs, the transition kernel is handcrafted to incrementally transform the data distribution $x_0 \sim F_{\mathbf{x}}(x)$ into a tractable prior distribution which is usually taken to be a standard Gaussian distribution.

Diffusion Models

- The most common choice for the transition kernel is

$$Q(x_t \mid x_{t-1}) = N(\sqrt{1-\beta_t}x_{t-1}, \beta_t \mathbf{I}) \tag{3}$$

where $\beta_t \in (0,1)$ is a hyperparameter chosen ahead of time by the designer.

- Note that this transition kernel allows us to marginalize the joint distribution to obtain an analytic form for $Q(x_t \mid x_0$ for all $t \in \{0,1,\ldots,T\}$. Specifically if we let $\alpha_t \stackrel{\text{def}}{=} 1-\beta_t$ and $\overline{\alpha}_t \stackrel{\text{def}}{=} \prod_{k=0}^{t} \alpha_k$, then we have

$$\begin{aligned} x_t &= \sqrt{\alpha_t}x_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1} \\ &= \sqrt{\alpha_t\alpha_{t-1}}x_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\epsilon \\ &\vdots \\ &= \sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon \end{aligned}$$

So we can conclude that

$$Q(x_t|x_0) = N(\sqrt{\overline{\alpha}_t}x_0, (1-\overline{\alpha}_t)\mathbf{I})$$

Diffusion Models

- This means that given the initial input x_0, we can easily obtain a sample x_t by sampling a Gaussian vector $\epsilon \sim N(0, \mathbf{I})$ and applying the above transformation $x_t = \sqrt{\overline{\alpha}_t} x_0 + \sqrt{1 - \overline{\alpha}_t}\epsilon$.
- When $\overline{\alpha}_T \approx 0$, then x_T is nearly a Gaussian distribution and so $Q(x_T) \approx N(0, \mathbf{I})$.
- To generate new data samples, DDPMs start by first generating an unstructured noise vector from the prior distribution and then gradually removing noise through a learned Markov chain running in the reverse direction.
- In particular, the reverse Markov chain is parameterized by a prior distribution $P(x_T) = N(0, \mathbf{I})$ because the forward process was constructed so that $Q(x_T) \approx N(0, \mathbf{I})$.

Diffusion Models

- The learnable transition kernel $P_w(x_{t-1} \mid x_t)$ with weights w takes the form

$$P_w(x_{t-1} \mid x_t) = N(\mu_w(x_t, t), \Sigma_w(x_t, t))$$

  where w denotes the model parameters and the man $\mu_w(x_t, t)$ and variance $\Sigma_w(x_t, t)$ are parameterized by deep neural networks.

- With this reverse Markov chain in hand, we can generate a data sample x_0 by first sampling a noise vector $x_T \sim p(x_T)$ and then iteratively sampling from the learnable transition kernel $x_{t-1} \sim P_w(x_{t-1} \mid x_t)$ until $t = 1$.

Diffusion Models

- The key to the success of this sampling process is training the reverse Markov chain to match the actual time reversal of the forward Markov chain. We adjust the parameter w so the joint distribution of the reverse Markov chain, $p_w(x_0, x_1, \ldots, x_T) \stackrel{\text{def}}{=} P(x_T) \prod k = 1^T P_w(x_{t-1} \mid x_t)$ closely approximates that of the forward process

$$Q(x_0, x_1, \ldots, x_T) \stackrel{\text{def}}{=} Q(x_0) \prod_{k=1}^{T} Q(x_t \mid x_{t-1}).$$

- This is achieved by minimizing the Kullback-Leibler (KL) divergence of these two distributions.

$$\begin{aligned} KL(Q(x_0, \ldots, x_T), & \quad P_w(x_0, \ldots, x_T)) \\ &= -\mathbb{E}_{Q(x_0,\ldots,x_T)} [\log P_W(x_0, \ldots, x_T)] + \text{const} \\ &= \mathbb{E}_{Q(x_0,\ldots,x_T)} \left[-\log P(x_T) - \sum_{k=1}^{T} \log \frac{P_w(x_{t-1} \mid x_t)}{Q(x_t \mid x_{t-1})} \right] + \text{co} \\ &\geq \mathbb{E} [-\log P_w(x_0)] + \text{const} \end{aligned}$$

Diffusion Models

- The "const" term contains terms that are independent of the weights w and hence don't impact training.
- The objective of DDPM training is to maxmize the VLB, which is relatively easy because it is a sum of independent terms and can therefore be estimated efficiently through Monte Carlo sampling and optimized using stochastic gradient descent.
- Notice that we are free to choose a different β_t at each time step. The original DDPM paper used a linear schedule where β_t increased by a fixed increment at each time step.
- It was later found that a sinusoidal schedule worked better where

$$\overline{\alpha}_t = \cos^2\left(\frac{\pi t}{2T}\right)$$

Diffusion Models

We now illustrate the DDPM model following an example in. We'll be using the Oxford 102 flower dataset on Kaggle. After downloading the dataset we create the training dataset.

Figure: Sample Images from Oxford 102 Flower Dataset

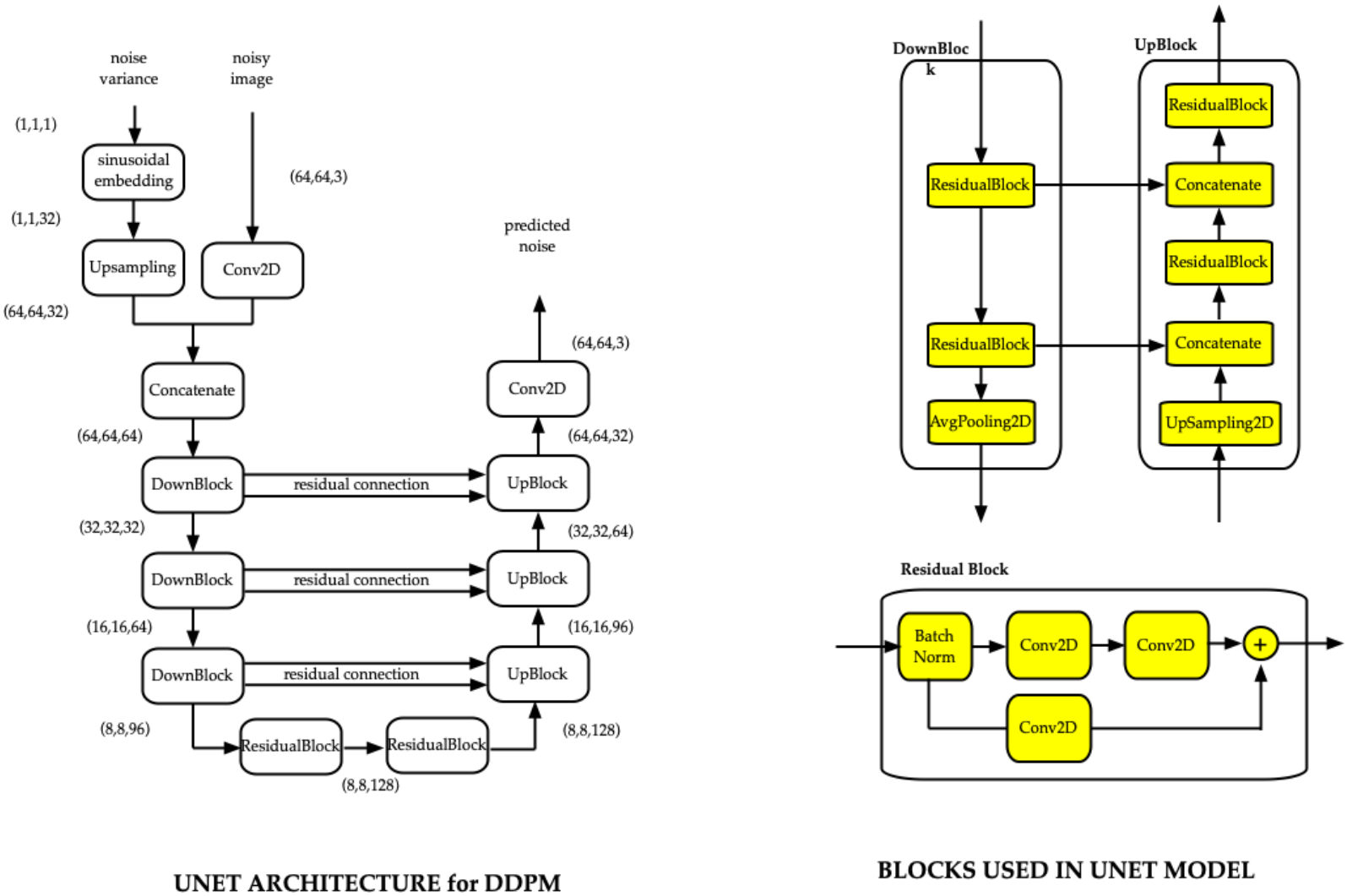
Diffusion Models

- Our denoising diffusion model will be based on a U-net architecture. Before discussing the U-net model in depth, let us look at how it is trained.
- Our model will be trained in a custom way so we need to build it as a subclass of Keras `Model` class.
- It is important to note that our `DiffusionModel` actually keeps two copies of the U-net model in it.
- One is trained using stochastic gradient descent, the other uses an exponential moving average (EMA) of the weights of the other copy. This is done because the EMA network is not as susceptible to short-term fluctuations and spikes in the trianing process, therefore making it more robust for generation of images than the actively trained network.
- We therefore use the EMA network when we produce an output from the network.

Diffusion Models

- The model we will use is a U-net model. This model consists of an encoder and decoder with residual connections between the convolutional layers.
- This model takes the noise variance β_t and an image x with shape $(64, 64, 3)$ as an input.
- The output is the model's prediction of the noise added to the image also with shape $(64, 64, 3)$.
- The encoder generates a latent space of size $(8, 8, 128)$ through a series of `DownBlocks`.
- The decoder predicts the noise added to the image during the forward pass.
- The decoder consists of a sequence of upsampling `UpBlocks`. The layers in the `DownBlock`, `UpBlock`, and `ResidualBlock` are shown on the right of the next figure.

Diffusion Models



UNET ARCHITECTURE for DDPM

BLOCKS USED IN UNET MODEL

Figure: (Left) Unet model (Right) blocks used in Unet model

Diffusion Model

This model was trained on the Oxford Flower Dataset using an Adam optimizer for 50 epochs. We used a batch size of 64 samples.

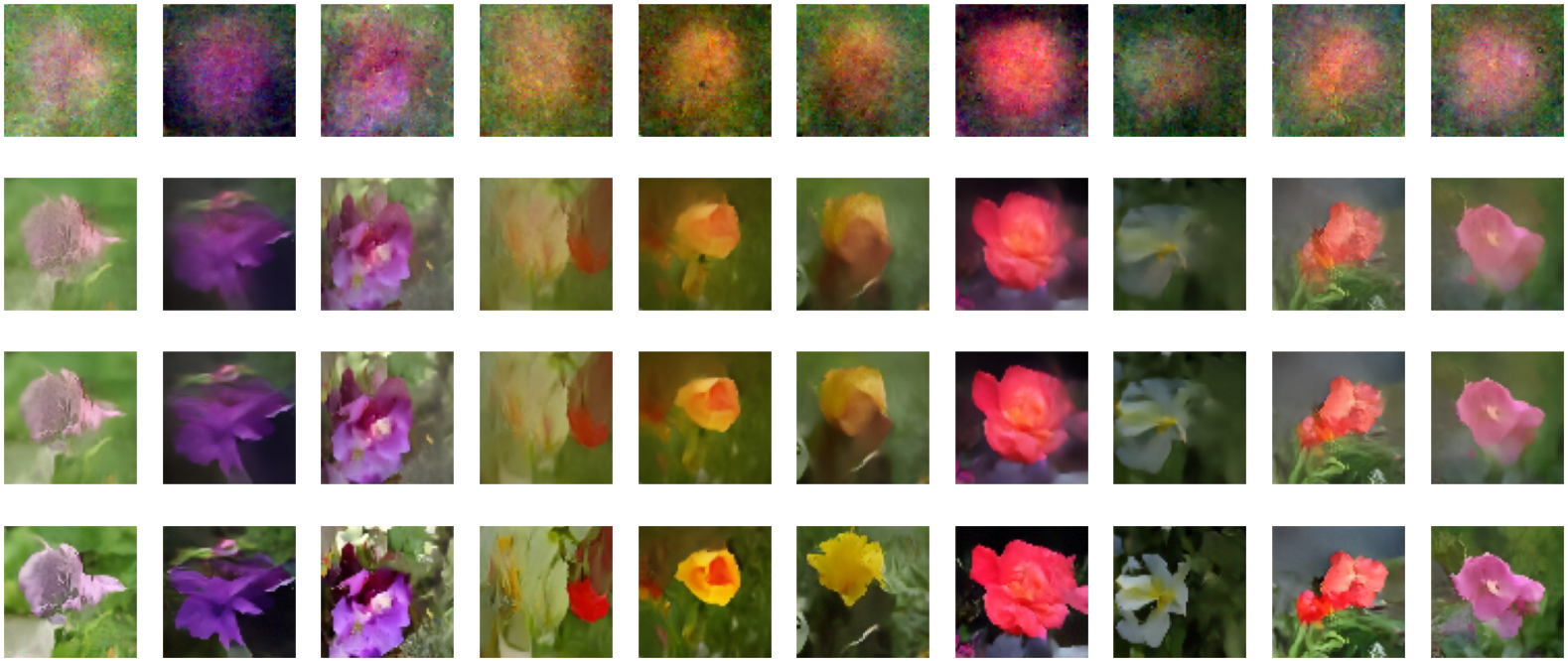


Figure: Reconstructions generated by DDPM model

Diffusion Models

Note that we can also interpolate between two points in the latent space. The results in Fig. ?? show how this allows us to smoothly morph one generated into another image.

Figure: Interpolating between points in the latent space to morph between images

Summary

- Generative learning builds models that generate new sample whose distribution match tat of the trainning inputs.
- Generative learning can be used to augment the original dataset, it can be used to obtained reduced order (latent space) representations of the inputs, we can use explore these latent spaces that allow us to control how we generate new samples, generative models can also be used to more easily identify outliers.
- As of 2023, generative learning has become the next "big" thing in deep learning with significant impact on lay society through applications such as ChatGPT and DALL-E 2.
- Generation of natural language using either RNN's or transformers provide one example of generative learning. These take a learned NLP models and feed the output back into the input to generate entire essays. Generative pre-trained transformer models (GPT) represent the current generation of large language models lying at the heart of many of the recent generative tools used by the lay public.

Summary

- Principal component analysis may be seen as the basis for early generative learning models. PCA seeks to transform a data matrix through a transformation the reduces the dimensionality of the representation. The rows of the transformation matrix are called the principal components of the data set when the transform the covariance of the original data to a diagonal form.
- In general the principal components of the data matrix are the eigenvalues of the original data set's covariance matrix.
- It is common to use singular value decompositions (SVD) to compute a dataset's principal components.
- Autoencoders are deep learning models with an encoder-decoder structure that takes an input vector, compresses it onto a lower dimensional latent space, and then decompresses the latent representation into a recosntruction of the original input.

Summary

- The main point of autoencoders is to find a compact latent space representation of the dataset, which can then be explored using vector arithmetic. We can use these operations to morph one image into another, to create biases in our data that favor one can of input, or to remove biases in a way that assures greater fairness.
- Variational autoencoders are autoencoders that map each input onto a probability distribution in the latent space. This latent distribution is usually Gaussian. The decoder then samples from these Gaussian distributions and uses the sample to reconstruct the input. A key part of training VAEs requires one to regularize training using the KL divergence. This has the effect of spreading out the representations over the entire latent space in a way that ensures all classes get more equal coverage than customarily occurs with unregulated autoencoders.

Summary

- Generative adversarial networks (GAN) were one of the first generative models that was highly successful in creating image samples.
- GANs use a game theory approach whose two players are a generator model G and a discriminator model, D.
- One trains a GAN by fixing D and then selecting G to minimize the loss. The next step involves fixing G and then training D to maximize the loss. The loss function is a binary cross entropy function that adjusts D to maximize the discriminator's likelihood of detecting fake samples and adjusts G to minimize the same objective for the discriminator. When it works, the training process converges to a Nash equilibrium where neither player has an advantage over the other.

Summary

- The latest generative model is the denoising diffusion probabilistic model (DDPM). This model is based on two Markov chains. The forward chain adds noise to an image in small increments until the image is a Gaussian distribution. The reverse chain undoes the addition of noise in small increments, taking a sampling of a multi-variate Gaussian distribution and then incrementally removing the noise to generate an image.
- DDPM models are based on a Unet architecture that takes an input image, adds noise to it and then predicts the noise that was added. The model can then be used to denoise a sample Gaussian to construct complex images.
- Diffusion models such as DDPM are used in text-to-image generators like DALL-E 2. These models use a GPT to transform a text string to a latent space representation and then use a diffusion model to generate the image.