

Bab 6

Autoencoder untuk Representasi Fitur

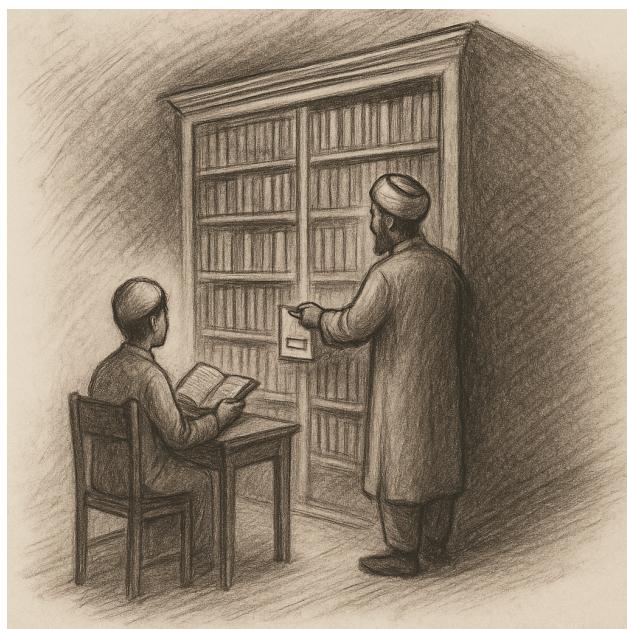
Setelah membahas Recurrent Neural Network (RNN) sebagai pendekatan yang efektif untuk menangani data sekuensial, kita telah melihat bagaimana model ini mampu mempelajari pola temporal dalam berbagai jenis data seperti teks, suara, atau sinyal waktu. Namun, tidak semua permasalahan dalam pembelajaran mesin bergantung pada prediksi berurutan atau keluaran yang terstruktur secara temporal. Pada banyak kasus, tantangan utama justru terletak pada bagaimana merepresentasikan data secara efisien dalam bentuk laten berdimensi lebih rendah tanpa kehilangan informasi penting. Untuk mengatasi kebutuhan ini, Bab 6 akan membahas Autoencoder, sebuah arsitektur jaringan saraf yang dirancang untuk melakukan representasi dan rekonstruksi data. Pendekatan ini memberikan landasan kuat dalam berbagai aplikasi seperti kompresi data, deteksi anomali, dan pralatih dalam pembelajaran mendalam.

6.1 Pendahuluan

Bayangkan di hadapanmu ada tumpukan catatan yang berisi semua pelajaran yang telah kamu pelajari selama ini—mulai dari ayat-ayat Al-Qur'an, hadis,

ilmu fiqih, bahasa Arab, hingga sejarah Islam. Catatan-catatan itu sangat beragam bentuknya: ada yang panjang, pendek, ditulis tangan, dicetak, atau bahkan hanya berupa coretan kecil. Ustaz Abdullah, gurumu yang bijak, mulai merasa kesulitan setiap kali kamu menanyakan ulang sebuah materi, karena ia harus mencari-cari di antara tumpukan itu semua.

Akhirnya, ia pun mengusulkan sebuah rencana cerdas. Ia memintamu untuk menyimpan semua catatan itu ke dalam lemari hafalan ajaib yang bisa memampatkan isi catatan ke dalam bentuk singkat berupa kode-kode khusus. Ketika kamu ingin mengulang pelajaran tertentu, kamu cukup menyebutkan kode atau lokasi di dalam lemari, dan Ustaz Abdullah akan menjelaskan ulang isi catatan tersebut dari awal, seolah-olah ia menuliskannya kembali hanya dari kodennya.



Gambar 6.1: Ustaz Abdullah dan rak buku tak terhingga

Lama-kelamaan, kamu dan Ustaz Abdullah mulai memahami tata letak lemari hafalan itu. Kamu menyadari bahwa agar penjelasan ustaz lebih akurat, kamu harus menyusun catatan yang serupa ke lokasi yang berdekatan. Misalnya, semua pelajaran fiqh disimpan di bagian kiri bawah, dan

hadis-hadis disimpan di kanan atas. Proses ini menjadi lebih mudah dan efisien.

Lalu muncul sebuah ide menarik: bagaimana jika kamu menyebutkan lokasi kosong yang belum pernah digunakan sebelumnya? Dengan penuh takjub, kamu mendapati bahwa Ustaz Abdullah bisa menyusun penjelasan baru yang masuk akal, seolah-olah itu memang materi baru yang pernah ada. Meskipun tidak sempurna, penjelasan itu tetap sesuai dengan gaya dan isi pelajaran-pelajaran sebelumnya.

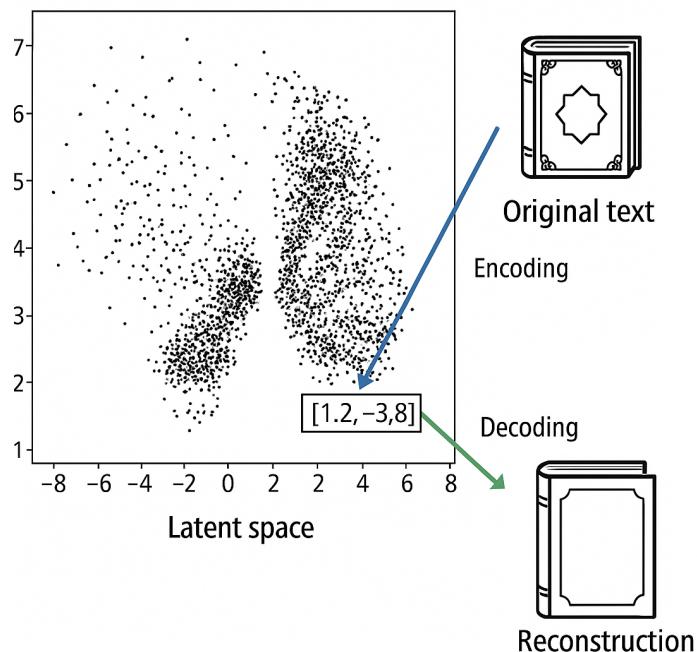
Cerita ini mengilustrasikan *autoencoder*: kamu sebagai *encoder*, menyusun catatan ke dalam lemari hafalan (**ruang laten**), dan Ustaz Abdullah sebagai *decoder*, membangun kembali isi dari kode atau lokasi tersebut. Bahkan saat diberi kode baru, ia mampu "menghasilkan" isi baru berdasarkan pemahaman dari seluruh pelajaran yang telah ia bantu susun sebelumnya—mirip dengan cara kerja *variational autoencoder*.

Dalam bab ini, ada beberapa istilah fundamental yang sangat penting, yaitu **ruang laten** dan **representasi laten**. Ruang laten adalah ruang imajiner tempat model menyimpan informasi penting dari data aslinya dalam bentuk yang lebih padat dan abstrak. Bayangkan, ada gambar wajah dengan ukuran 100×100 piksel. Untuk mengenali wajah itu, kita tidak perlu semua informasi pada setiap piksel. Kita bisa mengenali fitur wajah itu dengan cukup melihat dari bentuk wajah, ukuran mata, letak hidung, dan gaya rambut. Model akan menyaring informasi penting itu dan menyimpannya dalam bentuk vektor berdimensi kecil, misalnya 64 dimensi. Nah, tempat di mana kumpulan vektor tinggal disebut sebagai ruang laten. Adapun vektor pada ruang laten disebut sebagai representasi laten.

6.2 Konsep Autoencoder

Diagram proses *autoencoder* dari cerita sebelumnya disajikan pada Gambar 6.2. Teks asli (Original text) dikodekan ke dalam ruang laten sebagai titik koordinat $[1.2, -3.8]$. Titik ini mewakili representasi ringkas dari isi teks.

Proses dekoding kemudian mengubah representasi ini kembali menjadi bentuk rekonstruksi teks (Reconstruction) yang mirip dengan aslinya, layaknya Ustaz Abdullah yang menyusun ulang pelajaran hanya dari kode lokasi dalam lemari hafalan.



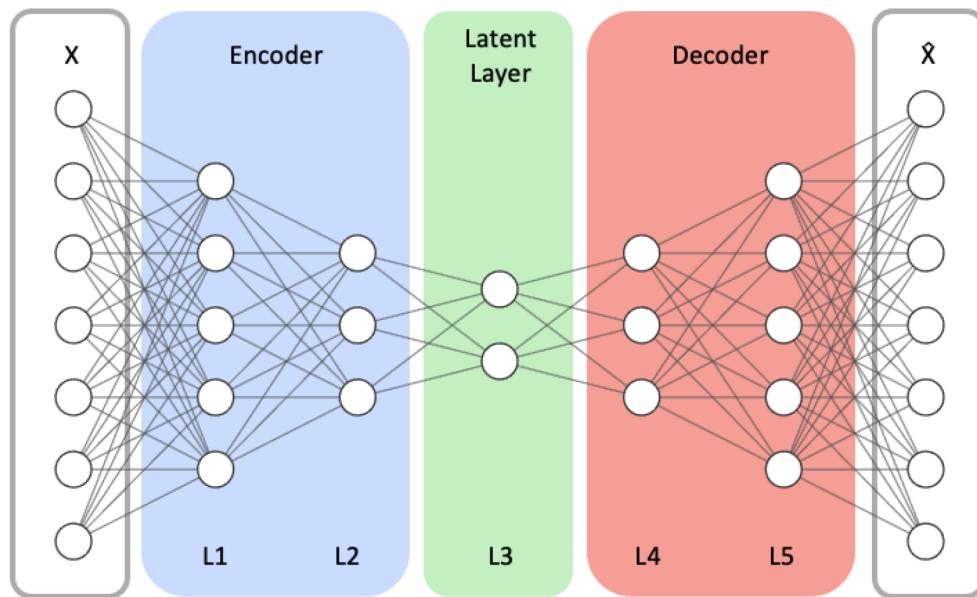
Gambar 6.2: Ilustrasi proses autoencoder berdasarkan analogi ustaz Abdullah.

6.2.1 Definisi dan Tujuan

Autoencoder (AE) adalah merupakan jaringan saraf (neural network) yang dilatih untuk melakukan proses *encoding* dan *decoding* pada suatu objek, untuk menghasilkan luaran dari proses atau model ini bisa sangat mirip dengan objek aslinya. Sebenarnya, model AE ini bisa digunakan untuk model generatif karena kita bisa *decode* titik apapun pada ruang 2D untuk menghasilkan objek baru.

6.2.2 Arsitektur Autoencoder

Arsitektur dasar autoencoder terdiri dari dua komponen utama, yaitu encoder dan decoder.



Gambar 6.3: Contoh arsitektur model autoencoder dengan jaringan encoder dan decoder yang simetris. X dan \tilde{X} masing-masing mewakili input model dan output rekonstruksinya.

(Song, Hyun, & Cheong, 2021)

6.2.3 Encoder

Encoder adalah bagian pertama dari autoencoder yang bertugas untuk mengubah data input menjadi representasi yang lebih ringkas, digambarkan pada bagian L1 dan L2 pada Gambar 6.3. Proses ini melibatkan pengurangan dimensi data, di mana informasi penting dari data asli dipertahankan. Encoder biasanya terdiri dari beberapa lapisan jaringan syaraf yang bertugas untuk mengekstrak fitur-fitur penting dari data input.

6.2.4 Decoder

Decoder adalah bagian kedua dari autoencoder yang bertugas untuk merekonstruksi data asli dari representasi laten yang dihasilkan oleh encoder, digambarkan pada bagian L4 dan L5 pada Gambar 6.3. Proses ini memungkinkan autoencoder untuk belajar bagaimana mengkodekan dan mendekodekan data dengan cara yang efisien, sehingga menghasilkan representasi yang dapat digunakan untuk berbagai tujuan analitis.

6.2.5 Fungsi Loss

Fungsi loss adalah komponen penting dalam pelatihan autoencoder, karena mengukur seberapa baik model dapat merekonstruksi data input dari representasi laten. Salah satu fungsi loss yang umum digunakan adalah Mean Squared Error (MSE), yang menghitung rata-rata kuadrat perbedaan antara data asli dan data yang direkonstruksi. Fungsi ini berperan dalam mengarahkan proses pembelajaran, sehingga autoencoder dapat meminimalkan kesalahan rekonstruksi dan belajar representasi yang lebih akurat. Dengan meminimalkan fungsi loss, autoencoder dapat meningkatkan kemampuannya dalam menangkap pola dan fitur yang relevan dari data.

6.2.6 Latent Layer/Space

Konsep ruang laten adalah inti dari cara kerja autoencoder. Ruang laten adalah representasi terkompresi dari data input yang dihasilkan oleh encoder digambarkan bagian L3 pada Gambar 6.3. Dalam ruang ini, data dipetakan ke dalam dimensi yang lebih rendah, yang memungkinkan model untuk menangkap informasi penting sambil mengabaikan noise dan detail yang tidak relevan. Dengan memanfaatkan ruang laten, autoencoder dapat menghasilkan representasi yang lebih ringkas dan informatif, yang sangat berguna dalam berbagai aplikasi, termasuk klasifikasi, clustering, dan anomaly detection. Dengan demikian, pemahaman yang mendalam tentang ruang laten

dan cara autoencoder memetakan data ke dalamnya adalah kunci untuk memanfaatkan potensi penuh dari teknik ini dalam analisis data.

6.2.7 Dataset MNIST untuk Busana

Fashion-MNIST adalah dataset yang berisi gambar-gambar artikel dari Zalando—terdiri dari 60.000 data pelatihan dan 10.000 data pengujian. Setiap data berupa gambar grayscale berukuran 28x28 piksel, yang dikaitkan dengan label dari 10 kelas. Zalando merancang Fashion-MNIST sebagai pengganti langsung untuk dataset MNIST asli guna keperluan benchmarking algoritma machine learning. Dataset ini memiliki ukuran gambar dan struktur pembagian data pelatihan serta pengujian yang sama dengan MNIST.

Contoh Dataset class untuk MNIST busana

```
1 import torch
2 from torch.utils.data import Dataset
3 import pandas as pd
4 import numpy as np
5 import torchvision.transforms as transforms
6
7 class FashionMNISTDataset(Dataset):
8     def __init__(self, csv_file, transform=None):
9         self.data = pd.read_csv(csv_file)
10        self.transform = transform or transforms.Compose([
11            transforms.ToPILImage(),
12            transforms.Resize((32, 32)),
13            transforms.ToTensor()
14        ])
15
16    def __len__(self):
17        return len(self.data)
18
19    def __getitem__(self, idx):
20        row = self.data.iloc[idx].values
```

```

21     image = row[1:].astype(np.uint8).reshape(28, 28)
22     image = np.expand_dims(image, axis=2)
23     image = self.transform(image)
24
25     return image, image

```

Listing 6.1: Kode untuk FashionMNISTDataset

Encoder

Berikutnya, mari kita lihat arsitektur dari Autoencoder kita.

```

1  class Encoder(nn.Module):
2
3      def __init__(self):
4          super(Encoder, self).__init__()
5          self.encoder = nn.Sequential(
6              nn.Conv2d(1, 32, kernel_size=3, stride=2, padding
7                  =1),
8              nn.ReLU(),
9              nn.Conv2d(32, 64, kernel_size=3, stride=2, padding
10                 =1),
11              nn.ReLU(),
12              nn.Conv2d(64, 128, kernel_size=3, stride=2,
13                  padding=1),
14              nn.ReLU(),
15              nn.Flatten(),
16              nn.Linear(2048, 2)
17
18
19      def forward(self, x):
20          return self.encoder(x)

```

Listing 6.2: Kode untuk Encoder

Model Encoder pada Listing 6.2 terdiri atas fitur input 32 x 32 piksel dan 128 fitur map. Untuk membangun model ini, kita mulai dengan membuat

Tabel 6.1: Model summary of the encoder

Layer (type)	Output Shape	Param #
InputLayer	(None, 1, 32, 32)	0
Conv2D (1 → 32)	(None, 32, 16, 16)	320
ReLU	(None, 32, 16, 16)	0
Conv2D (32 → 64)	(None, 64, 8, 8)	18,496
ReLU	(None, 64, 8, 8)	0
Conv2D (64 → 128)	(None, 128, 4, 4)	73,856
ReLU	(None, 128, 4, 4)	0
Flatten	(None, 2048)	0
Dense (2048 → 2)	(None, 2)	4,098

layer input untuk gambar. Gambar tersebut kemudian diproses melalui tiga layer Conv2D secara berurutan, yang masing-masing bertugas menangkap pola-pola dari yang sederhana hingga yang lebih kompleks.

Setiap layer konvolusi menggunakan stride 2 agar ukuran citra keluarannya menjadi setengah dari sebelumnya, sementara jumlah fitur (channel) ditingkatkan agar model bisa belajar lebih banyak informasi.

Hasil dari layer konvolusi terakhir diubah menjadi bentuk satu dimensi (flatten) lalu diteruskan ke layer dense (fully connected) berukuran 2, yang berfungsi sebagai representasi laten berdimensi dua dari citra tersebut. Lihat struktur Encoder padat Tabel 6.1. Cobalah untuk mengubah jumlah layer konvolusi dan jumlah filter guna memahami bagaimana arsitektur model memengaruhi jumlah parameter, kinerja model, dan waktu eksekusi.

Decoder

Decoder adalah seperti cermin dari encoder. Di sini, model decoder tidak menggunakan Conv2D melainkan ConvTranspose2d, sebagaimana pada Tabel 6.2. Layer konvolusi biasa itu mengurangi ukuran tensor menjadi setengah, baik tinggi maupun lebar, dengan cara mengatur nilai *stride* = 2.

Adapun *Convolutional Transpose Layer*(CTL) bekerja dengan prinsip

yang mirip seperti konvolusi biasa, yaitu menggeser filter pada citra. Tapi, perbedaannya terletak pada saat kita mengatur *stride*=2, maka ukuran tensor input akan menjadi dua kali lipat di kedua sisi dimensi (tinggi dan lebar).

Pada CTL, parameter *stride* menentukan jarak antar piksel hasil di dalam citra. Misalnya, apabila ada citra dengan ukuran filter $3 \times 3 \times 1$ dikonvolusikan pada citra dengan ukuran $3 \times 3 \times 1$ dengan *stride* =2, maka menghasilkan tensor output dengan ukuran $6 \times 6 \times 1$.

```
1 class Decoder(nn.Module):
2     def __init__(self):
3         super(Decoder, self).__init__()
4         self.decoder = nn.Sequential(
5             nn.Linear(2, 2048),
6             nn.Unflatten(1, (128, 4, 4)),
7             nn.ConvTranspose2d(128, 128, kernel_size=3,
8                               stride=2, padding=1, output_padding=1),
9             nn.ReLU(),
10            nn.ConvTranspose2d(128, 64, kernel_size=3,
11                               stride=2, padding=1, output_padding=1),
12            nn.ReLU(),
13            nn.ConvTranspose2d(64, 32, kernel_size=3,
14                               stride=2, padding=1, output_padding=1),
15            nn.ReLU(),
16            nn.Conv2d(32, 1, kernel_size=3, padding=1),
17            nn.Sigmoid()
18        )
19
20    def forward(self, x):
21        return self.decoder(x)
```

Listing 6.3: Kode untuk Decoder

Tabel 6.2: Model summary of the decoder

Layer (type)	Output Shape	Param #
InputLayer	(None, 2)	0
Linear ($2 \rightarrow 2048$)	(None, 2048)	6,144
Unflatten to (128, 4, 4)	(None, 128, 4, 4)	0
ConvTranspose2D (128 \rightarrow 128)	(None, 128, 8, 8)	147,584
ReLU	(None, 128, 8, 8)	0
ConvTranspose2D (128 \rightarrow 64)	(None, 64, 16, 16)	73,792
ReLU	(None, 64, 16, 16)	0
ConvTranspose2D (64 \rightarrow 32)	(None, 32, 32, 32)	18,464
ReLU	(None, 32, 32, 32)	0
Conv2D (32 \rightarrow 1)	(None, 1, 32, 32)	289
Sigmoid	(None, 1, 32, 32)	0

Menggabungkan Encoder dan Decoder

Untuk melatih encoder dan decoder secara bersamaan, kita perlu membuat satu model yang merepresentasikan alur data dari gambar masuk ke encoder, lalu hasilnya diteruskan ke decoder.

Di PyTorch, kita cukup membuat satu kelas Autoencoder yang berisi dua bagian utama: `self.encoder` dan `self.decoder`. Kemudian, di dalam fungsi `forward()`, data akan mengalir dari encoder ke decoder. Adapun kode Autoencoder bisa dilihat pada Listing 6.4.

```

1 class Autoencoder(nn.Module):
2     def __init__(self):
3         super(Autoencoder, self).__init__()
4         self.encoder = Encoder()
5         self.decoder = Decoder()
6
7     def forward(self, x):
8         z = self.encoder(x)
9         out = self.decoder(z)

```

```
10  
11     return out
```

Listing 6.4: Kode untuk Autoencoder

Melatih Autoencoder

Setelah membangun model, kita bisa melakukan training. Sebelum itu, kita perlu load dataset Fashion-MNIST. Dataset ini yang diload adalah data training dulu. Untuk batch size, kita bisa memakai ukuran 64 atau 128. Jangan lupa untuk mengacak data dengan shuffle.

Setelah load dataset, model perlu kita kompile dengan fungsi loss, misalnya MSELoss, dan optimizer Adam. Untuk tahapan ini, kita cukup menggunakan jumlah epoch sebanyak 10. Adapun contoh bisa dilihat pada Listing 6.5.

```
1 train_dataset = FashionMNISTDataset('./data/fashion-  
2   mnist/fashion-mnist_train.csv')  
3  
3 train_loader = DataLoader(train_dataset, batch_size=128,  
4   shuffle=True)  
4  
5 device = torch.device("cuda" if torch.cuda.is_available  
6   () else "cpu")  
6 model = Autoencoder().to(device)  
7  
8 criterion = nn.MSELoss()  
9 optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

Listing 6.5: Kode untuk Load dataset

Rekonstruksi Citra

Kita dapat menguji kemampuan autoencoder dalam merekonstruksi citra dengan cara mengalirkan citra dari data uji (test set) ke dalam autoencoder,

lalu membandingkan hasil keluarannya dengan citra aslinya. Kode untuk melakukan ini ditunjukkan pada Listing 6.6.

Perhatikan bahwa hasil rekonstruksinya tidak sempurna — masih ada beberapa detail dari gambar asli yang tidak berhasil ditangkap oleh proses decoding, seperti misalnya logo. Hal ini wajar, karena saat setiap gambar direduksi menjadi hanya dua angka, secara alami ada informasi yang hilang.

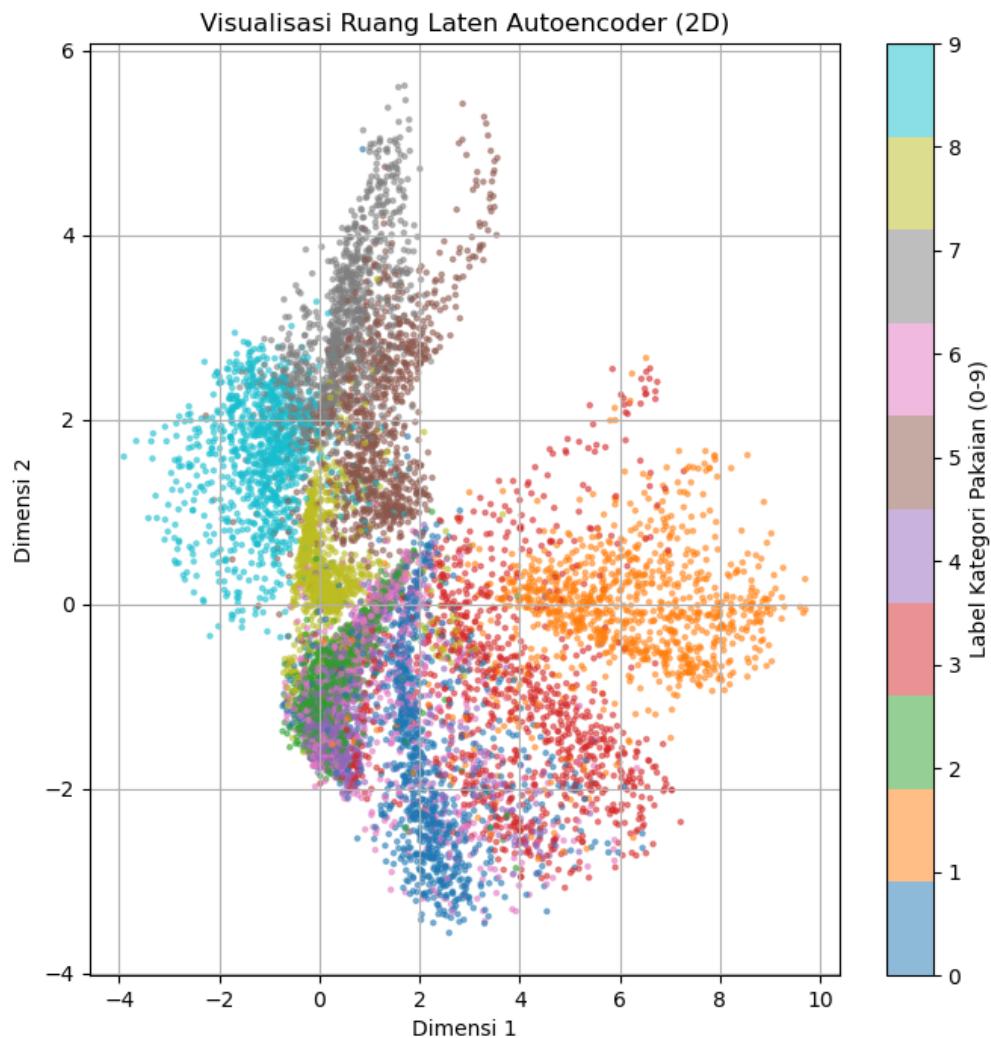
Sekarang, mari kita telusuri bagaimana encoder merepresentasikan citra-citra tersebut di dalam ruang laten (latent space).

```
1 model.eval()
2 with torch.no_grad():
3     test_imgs, _ = next(iter(train_loader))
4     test_imgs = test_imgs.to(device)
5     outputs = model(test_imgs)
6
7
8     n = 10
9     plt.figure(figsize=(20, 4))
10    for i in tqdm(range(n), desc="showing reconstruction"):
11        ax = plt.subplot(2, n, i + 1)
12        plt.imshow(test_imgs[i].cpu().squeeze(), cmap='gray')
13        )
14        ax.axis("off")
15        ax = plt.subplot(2, n, i + 1 + n)
16        plt.imshow(outputs[i].cpu().squeeze(), cmap='gray')
17        ax.axis("off")
18    plt.show()
```

Listing 6.6: Kode untuk Rekonstruksi Citra

Visualisasi Ruang Laten

Kita bisa memvisualisasikan bagaimana citra di-embed pada ruang laten dengan memasukkan data test pada model encoder dan plot hasilnya.



Gambar 6.4: Visualisasi Ruang Laten dari Fashion-MNIST

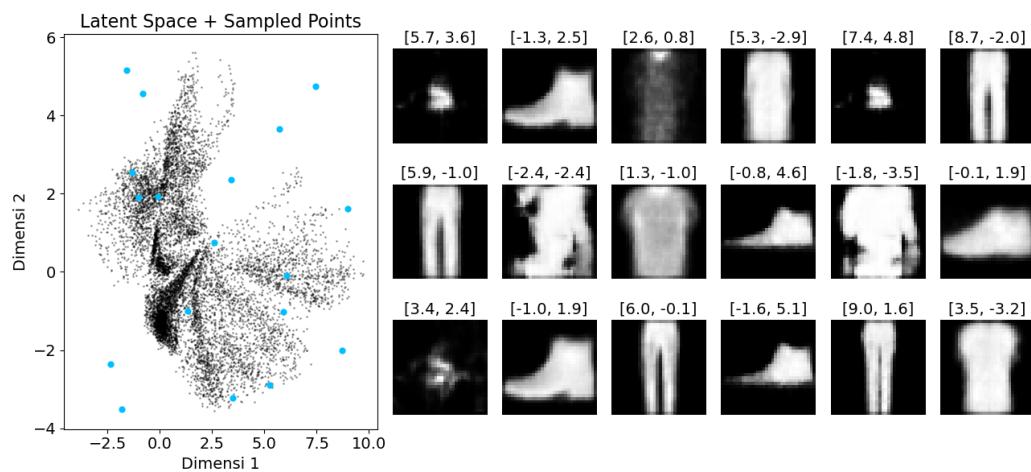
Kita dapat memberi warna pada setiap titik berdasarkan label pakaian dari gambar aslinya untuk menghasilkan plot seperti pada Gambar 6.4. Hasilnya, struktur ruang laten menjadi sangat jelas!

Meskipun label pakaian tidak pernah diberikan ke model saat pelatihan, autoencoder secara alami berhasil mengelompokkan gambar-gambar yang mirip ke bagian yang sama di ruang laten.

Sebagai contoh, gugus titik berwarna biru tua di pojok kanan bawah me-

rupakan gambar-gambar celana (trousers), sedangkan gugus titik berwarna merah di bagian tengah merupakan citra-citra sepatu boot (ankle boots).

Membuat Citra Baru (Image Generation)



Gambar 6.5: Hasil Generasi Image dari Pakaian dan Busana

Kita dapat membuat gambar-gambar baru dengan cara mengambil titik-titik acak dari ruang laten, lalu mengubahnya kembali menjadi gambar menggunakan bagian decoder dari autoencoder. Proses ini disebut generasi atau sampling dari ruang laten, dan hasilnya bisa divisualisasikan seperti pada gambar.

Beberapa contoh gambar yang dihasilkan dari titik-titik acak tersebut dapat dilihat di sebelah kanan Gambar 6.5. Titik-titik biru di diagram kiri menunjukkan posisi titik-titik yang diambil dari ruang laten, sedangkan gambar di sebelah kanannya menunjukkan hasil dekode dari masing-masing titik. Di bawah setiap gambar, terdapat vektor koordinat dari titik tersebut di ruang laten.

Menariknya, sebagian gambar yang dihasilkan terlihat cukup masuk akal dan menyerupai gambar pakaian asli, namun sebagian lainnya tampak kurang realistik. Mengapa hal ini bisa terjadi?

Untuk menjawab pertanyaan tersebut, mari kita perhatikan lebih lanjut

bagaimana distribusi titik-titik di ruang laten. Beberapa jenis pakaian hanya diwakili oleh area yang sangat kecil di ruang laten, sedangkan jenis lainnya menempati area yang jauh lebih luas.

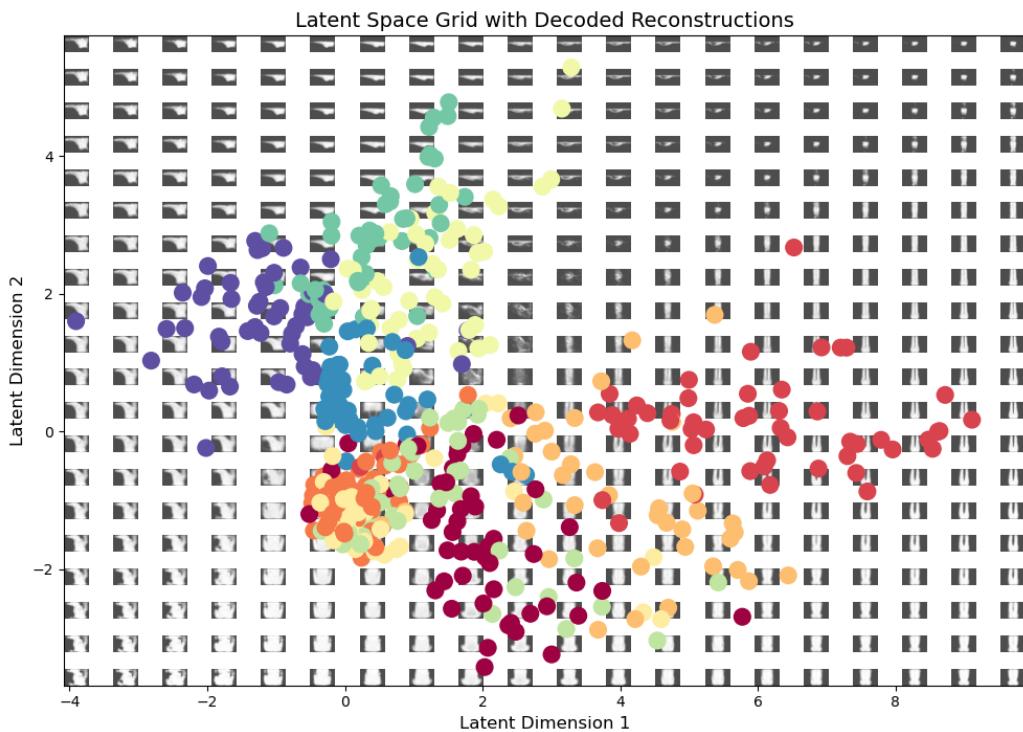
Distribusi titik tidak simetris terhadap titik pusat (0,0), dan juga tidak terbatas dalam batas tertentu. Sebagai contoh, banyak titik yang memiliki nilai sumbu-y positif lebih besar dibandingkan sumbu-y negatif. Bahkan, ada titik-titik yang mencapai nilai di atas 8 pada sumbu-y. Terdapat celah-celah besar di antara kelompok titik dengan warna berbeda, yang mewakili jenis pakaian yang berbeda. Karena hal-hal tersebut, **mengambil titik acak dari ruang laten menjadi tantangan tersendiri**. Titik-titik yang dipilih secara sembarangan bisa saja berada di area kosong atau tidak relevan, sehingga gambar yang dihasilkan oleh decoder menjadi tidak realistik atau bahkan tidak dapat dikenali.

Sebenarnya, observasi ini menjadikan sampling dari ruang laten menjadi sulit. Jika melapisi ruang laten dengan citra hasil dekode pada grid, sebagaimana pada Gambar 6.6, kita bisa memahami kenapa dekoder tidak selalu menhasilkan citra yang bagus.

Pertama, kita dapat melihat bahwa jika kita mengambil titik-titik secara acak dalam ruang laten yang dibatasi, maka kita lebih mungkin mendapatkan citra yang mirip seperti tas (ID 8) daripada sepatu bot (ID 9). Hal ini karena area pada ruang laten yang digunakan untuk mengenali tas (berwarna oranye) lebih luas dibandingkan area untuk sepatu bot (berwarna merah). Artinya, beberapa kategori memiliki area representasi yang lebih besar di ruang laten dibanding kategori lain.

Kedua, sebenarnya tidak ada aturan jelas bagaimana cara memilih titik acak di ruang laten, karena distribusi titik-titik ini tidak terdefinisi secara pasti. Secara teknis, kita bisa memilih titik mana saja di bidang 2D. Bahkan tidak ada jaminan bahwa titik-titik akan terpusat di sekitar titik (0, 0). Hal ini membuat proses sampling dari ruang laten menjadi menantang.

Ketiga, kita bisa menemukan area kosong di ruang laten, yaitu area yang tidak diisi oleh citra-citra hasil encoding dari data pelatihan. Misalnya,



Gambar 6.6: Grid dari embedding dekode yang ditumpuk pada citra hasil dari dekoder

terdapat ruang putih besar di bagian tepi domain — ini berarti autoencoder tidak pernah menerima data dari area tersebut, sehingga tidak dilatih untuk menghasilkan citra yang baik dari sana.

Bahkan, titik-titik yang dekat dengan pusat ruang laten belum tentu menghasilkan citra yang baik. Ini karena autoencoder tidak dirancang untuk menjamin bahwa ruang laten bersifat kontinu. Contohnya, meskipun titik $(-1, -1)$ mungkin bisa menghasilkan citra sandal yang memuaskan, titik yang sangat dekat seperti $(-1.1, -1.1)$ bisa saja menghasilkan citra yang tidak dapat dikenali. Tidak ada mekanisme di dalam model untuk memastikan bahwa area yang berdekatan menghasilkan output yang serupa.

Masalah ini memang tidak begitu terlihat ketika ruang laten hanya dua dimensi, karena autoencoder harus "menjejalkan" berbagai kategori pakaian ke dalam ruang terbatas. Namun saat kita ingin menghasilkan citra

yang lebih kompleks (seperti wajah manusia) dan menggunakan ruang laten berdimensi tinggi, masalah ini menjadi jauh lebih serius. Jika autoencoder dibiarkan secara bebas menentukan cara menggunakan ruang laten, maka akan muncul celah besar di antara kelompok-kelompok data, dan model tidak punya alasan untuk "mengisi" bagian tengah agar menghasilkan citra yang wajar.

Untuk mengatasi ketiga permasalahan tersebut, kita perlu mengubah autoencoder menjadi variational autoencoder.

6.3 Variational Autoencoder (VAE)

Mari kita ingat kembali cerita ustaz Abdullah dengan muridnya

Setelah sekian lama menyusun semua catatan pelajaran ke dalam lemari hafalan berdasarkan satu titik lokasi tertentu, kamu dan Ustaz Abdullah mulai menyadari adanya masalah. Beberapa materi yang mirip bisa tersimpan terlalu jauh, dan ketika kamu mencoba mengambil titik di antara dua pelajaran, hasilnya menjadi penjelasan yang aneh atau bahkan tidak masuk akal.

Kali ini, kamu dan Ustaz mengubah pendekatan. Alih-alih menyimpan satu catatan di satu titik pasti, kamu memutuskan untuk menyimpan area kemungkinan—semacam zona kabur—yang mewakili di mana suatu catatan biasanya berada. Misalnya, pelajaran hadis disimpan dalam bentuk ”awan” di sisi kanan atas, dengan titik pusat dan sebaran yang menggambarkan variasi cara penulisan dan pengajarannya.

Namun, agar kamu tidak sembarangan menaruh awan-awan ini, Ustaz Abdullah memberlakukan aturan baru: semua awan harus dipusatkan sedekat mungkin dengan tengah lemari, dan setiap awan harus memiliki sebaran standar. Jika kamu menyimpang terlalu jauh dari aturan ini—misalnya meletakkan awan di tepi lemari atau membuatnya terlalu sempit atau terlalu lebar—kamu akan ”membayar lebih” dalam bentuk penjelasan ustaz yang makin tidak akurat.

Setelah berbulan-bulan menggunakan sistem baru ini, kamu menyadari hasilnya jauh lebih baik. Lemari hafalan menjadi lebih teratur. Bahkan saat kamu meminta penjelasan dari titik acak di dalam lemari, Ustaz Abdullah dapat menyusun pelajaran baru yang tetap masuk akal dan bergaya Islami. Kali ini tidak ada lagi penjelasan aneh atau materi yang tidak wajar. Ternyata, menyimpan catatan dalam bentuk distribusi dan menerapkan aturan penyebaran telah membuat perbedaan besar.

Cerita ini adalah ilustrasi bagaimana kita mengubah persepsi atas lemari tak terhingga. Hal ini juga berlaku dalam konteks autoencoder yang

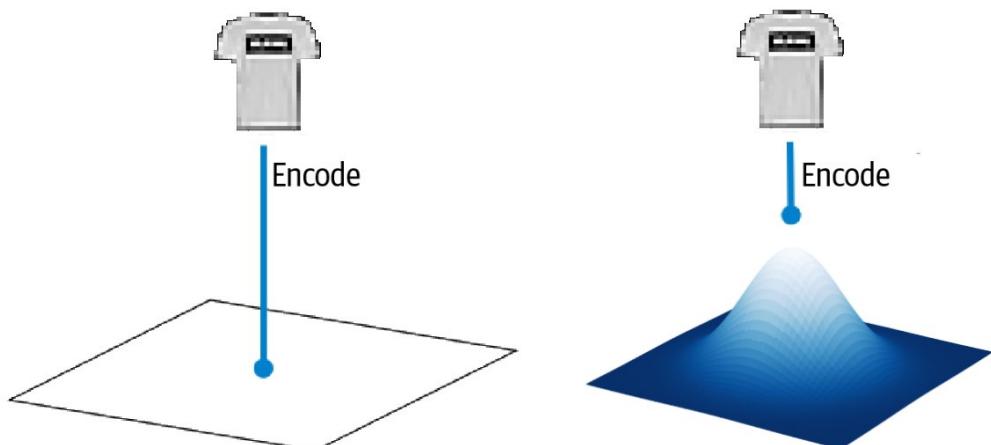
dikonversikan ke variasional autoencoder (VAE). Adapun yang perlu diubah dari sisi VAE adalah *encoder* dan fungsi loss.

VAE tidak menyimpan data sebagai satu titik pasti, tapi sebagai distribusi—with rata-rata (μ) dan simpangan baku (σ). Dari distribusi ini, model mengambil sampel acak menggunakan rumus $z = \mu + \sigma \cdot \epsilon$.

Agar distribusi tetap teratur, VAE menambahkan aturan regularisasi agar mendekati distribusi normal standar. Hasilnya, selain bisa merekonstruksi data, VAE juga mampu menghasilkan data baru yang masuk akal dan bervariasi.

6.3.1 Encoder

Pada autoencoder, setiap image dipetakan pada satu titik koordinat pada ruang laten. Alih-alih menggunakan titik, VAE menggunakan distribusi normal multivariabel di sekitar titik pada ruang laten, sebagaimana pada Gambar 6.7.



Gambar 6.7: Perbedaan antara encoder pada Autoencoder dan VAE

Distribusi Normal Multivariabel Distribusi normal (atau *distribusi Gaussian*) $\mathcal{N}(\mu, \sigma)$ adalah distribusi probabilitas yang ditandai dengan bentuk *bell curve* yang khas, didefinisikan oleh dua variabel: **mean** (μ) dan **varians** (σ^2). **Simpangan baku** (σ) adalah akar kuadrat dari varians.

Fungsi kepadatan probabilitas (PDF) dari distribusi normal satu dimensi adalah:

$$f(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gambar 3-11 menunjukkan beberapa distribusi normal dalam satu dimensi, untuk berbagai nilai mean dan varians. Kurva merah adalah *standard normal* (atau *unit normal*) $\mathcal{N}(0, 1)$ — distribusi normal dengan rata-rata 0 dan varians 1.

Kita dapat mengambil sampel titik z dari distribusi normal dengan mean μ dan simpangan baku σ menggunakan rumus berikut:

$$z = \mu + \sigma\epsilon$$

di mana $\epsilon \sim \mathcal{N}(0, 1)$ adalah variabel acak dari distribusi normal standar. Konsep distribusi normal dapat diperluas ke lebih dari satu dimensi. Fungsi kepadatan probabilitas untuk *distribusi normal multivariat* (atau *distribusi Gaussian multivariat*) $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ dalam k dimensi dengan vektor rata-rata $\boldsymbol{\mu}$ dan matriks kovarians simetris $\boldsymbol{\Sigma}$ adalah sebagai berikut:

$$f(x_1, \dots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}}$$

Tugas dari encoder cukup memetakan tiap-tiap input ke nilai rata-rata vektor dan vektor varians. Kita bisa menggunakan neural network pada encoder untuk memetakan input pada nilai rata-rata dan varians dari vektor. Tabel 6.3 adalah isi dari struktur model encoder dari VAE yang dibuat.

Tabel 6.3: Struktur Arsitektur Encoder untuk VAE

Layer	Output Shape	Keterangan
Conv2d ($1 \rightarrow 32$)	$32 \times 16 \times 16$	Kernel 3x3, stride 2, padding 1
Conv2d ($32 \rightarrow 64$)	$64 \times 8 \times 8$	Kernel 3x3, stride 2, padding 1
Conv2d ($64 \rightarrow 128$)	$128 \times 4 \times 4$	Kernel 3x3, stride 2, padding 1
ReLU	sama seperti input	Fungsi aktivasi non-linear
Flatten	2048	$128 \times 4 \times 4$
Linear $\rightarrow \mu$	2	Output mean dari distribusi laten
Linear $\rightarrow \log\sigma$	2	Output log varians dari distribusi laten

```

1 class EncoderVAE(nn.Module):
2     def __init__(self):
3         ..
4     def _reparameterize(self, mu, logvar):
5         ..
6     def forward(self, x):
7         ..

```

Listing 6.7: Kode untuk model encoder VAE

Reparameterisasi Jika diperhatikan pada Listing 6.7, ada fungsi dengan nama `reparameterize()`.

Daripada melakukan sampling langsung dari distribusi normal dengan parameter `z_mean` dan `z_log_var`, kita dapat melakukan sampling `epsilon` dari distribusi normal standar, lalu menyesuaikan hasilnya secara manual agar memiliki nilai rata-rata dan varians yang benar.

Hal ini dikenal sebagai *trik reparameterisasi*, dan penting karena memungkinkan gradien dapat melakukan backpropagation secara bebas melalui lapisan tersebut. Dengan menjaga seluruh elemen acak hanya dalam variabel `epsilon`, turunan parsial dari output lapisan terhadap inputnya bisa menjadi deterministik (yaitu, tidak bergantung pada acakan `epsilon`), yang sangat penting agar proses backpropagation melalui lapisan tersebut tetap memungkinkan.

6.3.2 Fungsi Loss untuk VAE

Dalam VAE, kita tidak hanya ingin gambar hasil rekonstruksi mirip dengan input aslinya (melalui *reconstruction loss*), tetapi juga ingin memastikan bahwa distribusi laten mendekati distribusi normal standar. Untuk itu, kita tambahkan fungsi loss *Kullback-Leibler*(KL) loss:

$$\text{KL_loss} = -0.5 \times \sum (1 + z_log_var - z_mean^2 - \exp(z_log_var))$$

atau secara matematis:

$$D_{\text{KL}}[\mathcal{N}(\mu, \sigma) \parallel \mathcal{N}(0, 1)] = -\frac{1}{2} \sum (1 + \log(\sigma^2) - \mu^2 - \sigma^2) \quad (6.1)$$

Loss ini akan bernilai minimum jika $\mu = 0$ dan $\log(\sigma^2) = 0$. Artinya, distribusi encoder menyerupai distribusi Gaussian standar.

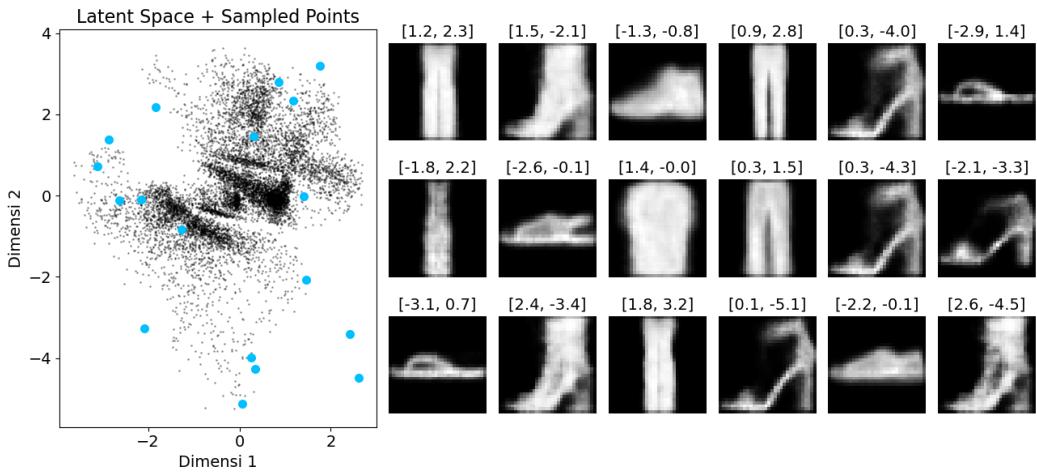
Mengapa penting? KL divergence berperan penting dalam menjaga agar distribusi di ruang laten tetap rapih dan terstruktur. Dengan memaksa hasil encoder untuk mendekati distribusi Gaussian standar, kita mencegah terjadinya ruang laten yang kosong atau tidak termanfaatkan. Selain itu, hal ini juga memungkinkan kita untuk melakukan sampling titik-titik baru dari distribusi standar tersebut untuk menghasilkan data baru yang bervariasi namun tetap konsisten dengan distribusi pelatihan.

Untuk mengatur seberapa kuat efek KL divergence, kita bisa mengalikan loss ini dengan faktor β dalam β -VAE.

Analisis Hasil VAE

Setelah melatih model VAE, kita bisa menggunakan enkoder VAE untuk mengencode citra dari kumpulan data tes dan memetakaan nilai z_{mean} dari ruang laten. Gambar 6.8 menunjukkan struktur dari ruang laten dari hasil VAE dan kumpulan hasil citra generaasi dari decoder. Kita bisa melihat perbandingan distribusi atau sebaran koordinat representasi laten (warna biru) yang mulai tersebar pada titik-titik hitam pada ruang laten.

Perubahan sebaran distribusi dari hasil dekoder VAE ini bisa terjadi karena beberapa hal. Pertama, fungsi loss KL sangat membantu menjamin



Gambar 6.8: Hasil generasi citra dari VAE

bahwa rata-rata dan log varians dari citra hasil enkode tidak menyebar terlalu jauh dari distribusi normal. Kedua, kualitas citra hasil dekode menjadi lebih baik dibanding sebelumnya karena ruang laten kini lebih kontinyu, berkat encoder yang bersifat stokastik, bukan deterministik seperti pada autoencoder biasa.

6.3.3 Eksplorasi pada Ruang Laten

Sejauh ini, semua yang telah kita lakukan pada autoencoder dan VAE itu terbatas pada jumlah dimensi dari ruang laten, yaitu dua dimensi. Tentunya, dimensi yang kecil ingin sangat membantu kita dalam memahami ruang laten dari autoencoder maupun VAE, terlebih pada cara kerja enkoder dan dekoder, dan bagaimana pengaruh perubahan kecil pada model berdampak pada hasil akhirnya.

Mari kita masuk ke dalam kasus dataset yang lebih kompleks dan bagaimana VAE bekerja pada dimensi ruang laten yang lebih tinggi.

Dataset wajah selebriti CelebA

Untuk mencoba ruang laten yang lebih kompleks, kita menggunakan dataset CelebFaces Attributes (CelebA) (Liu, Luo, Wang, & Tang, 2015) untuk melatih model VAE kita. CelebA adalah kumpulan dataset wajah beragam jenis selebriti dunia dengan beragam jenis label seperti senyum, berkumis, berkacamata, dan sebagainya. Dataset ini berjumlah lebih dari 200.000 citra berwarna. Contoh sampelnya seperti pada Gambar 6.9.



Gambar 6.9: Hasil generasi citra dari VAE

Untuk saat ini, kita belum memerlukan label dari dataset CelebA untuk melatih VAE. Label akan berguna ketika kita ingin memvisualisasikan ruang laten yang multidimensi.

6.3.4 Melatih VAE untuk dataset CelebA

Arsitektur jaringan untuk model CelebA mirip dengan arsitektur pada Fashion-MNIST. Tapi, ada sedikit perbedaan pada arsitekturnya. Dikarenakan dataset CelebA adalah RGB, maka kita perlu mengubah jumlah kanal input dari arsitektur VAE dari satu menjadi tiga. Pada model encoder Fashion-MNIST, ukuran dimensi d ruang laten $d = 2$, adapun untuk kasus CelebA, kita akan

Tabel 6.4: Encoder Architecture

Layer	Konfigurasi
Conv2d	c=3, f=128, k=3, s=2, p=1
BatchNorm2d (bn1)	f=128
Conv2d	c=128, f=128, k=3, s=2, p=1
BatchNorm2d (bn2)	f=128
Conv2d	c=128, f=128, k=3, s=2, p=1
BatchNorm2d (bn3)	f=128
Conv2d	c=128, f=128, k=3, s=2, p=1
BatchNorm2d (bn4)	f=128
LeakyReLU	neg=0.01
Flatten	dim=1
Linear (fc_mu)	in=512, out=200
Linear (fc_logvar)	in=512, out=200

menggunakan $d = 200$. Dengan ukuran dimensi yang lebih besar, kita bisa menyimpan informasi lebih banyak dari citra. Ada layer *batch normalization* (BN) pada setiap konvolusi. BN berfungsi untuk stabilisasi ketika pelatihan model. Di sini, kita menaikkan faktor β untuk KL loss menjadi 2.000.

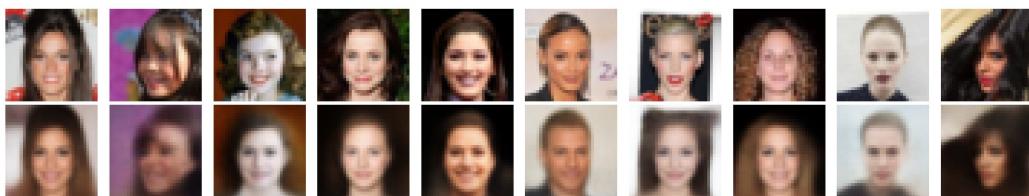
Tabel 6.4 adalah arsitektur dari enkoder VAE untuk kasus CelebA. Enkoder ini terdiri atas empat layer konvolusi yang diikuti dengan BN di mana setiap akhir dari pasangan Conv2d dan BN diaktivasi dengan fungsi LeakyReLU. Di layer FC, terdapat fc_mu dan fc_logvar yang berfungsi untuk variasionalnya. Adapun Tabel 6.5 adalah arsitektur untuk dekodernya yang terdiri atas empat layer dekonvolusi (ConvTranspose2d). Pada Tabel 6.4 dan Tabel 6.5c adalah jumlah input kanal, f adalah jumlah output fitur map, k adalah ukuran kernel, s adalah ukuran *stride*, dan p adalah ukuran *padding*.

6.3.5 Analisis Hasil Rekonstruksi

Mari kita amati hasil rekonstruksi wajah artis. Sisi atas pada Gambar 6.10 menunjukkan foto asli wajah artis, sedangkan yang bawah menunjukkan hasil

Tabel 6.5: Decoder Architecture

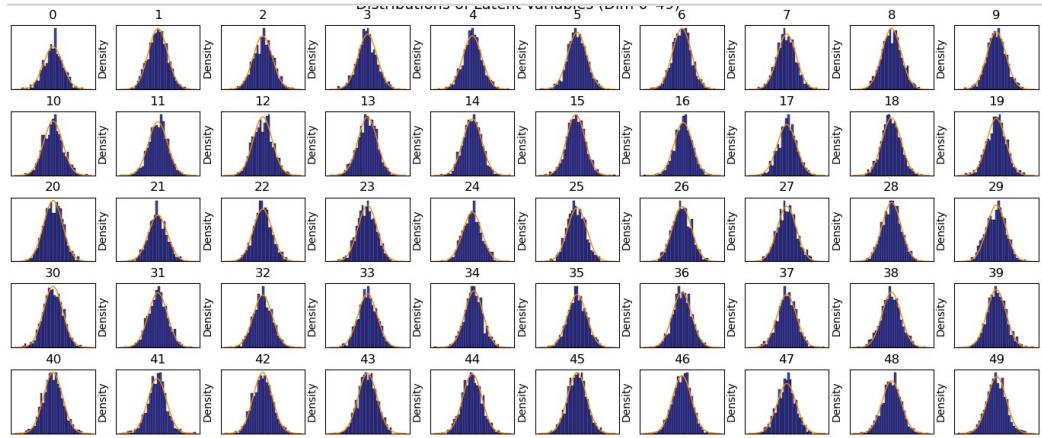
Layer	Konfigurasi
Linear (fc)	c=200, f=512
ConvTranspose2d	c=128, f=128, k=4, s=2, p=1
BatchNorm2d (bn1)	f=128
ConvTranspose2d	c=128, f=128, k=4, s=2, p=1
BatchNorm2d (bn2)	f=128
ConvTranspose2d	c=128, f=128, k=4, s=2, p=1
BatchNorm2d (bn3)	f=128
ConvTranspose2d	c=128, f=3, k=4, s=2, p=1
LeakyReLU	neg=0.2
Sigmoid	-



Gambar 6.10: Hasil rekonstruksi dari foto wajah artis

rekonstruksi melalui model enkoder dan dekoder.

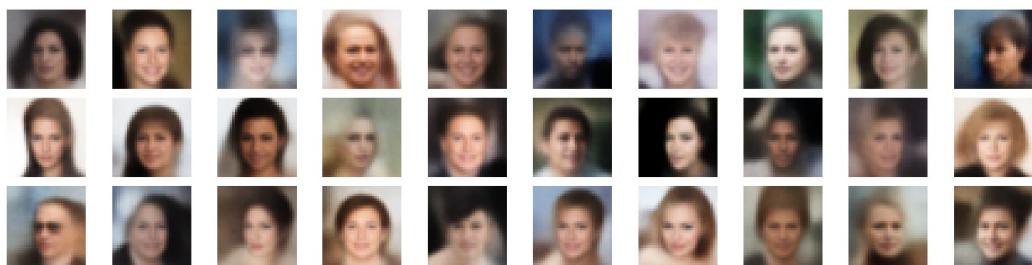
Kita bisa melihat bahwa VAE berhasil menangkap fitur-fitur kunci dari setiap wajah, seperti sudut wajah, gaya rambut, ekspresi, dan sebagainya. Memang, beberapa informasi detil masih kurang. Tapi, ini penting untuk diingat bahwa tujuan dari membuat VAE tidaklah untuk mencapai hasil rekonstruksi paling sempurna. Tujuan akhir kita adalah membuat sampel dari ruang laten untuk menghasilkan citra wajah baru artis. Sebagaimana yang ditunjukkan pada Gambar 6.11 bahwa sampel dimensi ruang laten menunjukkan adanya 'kenormalan' pada distribusinya. Ini menunjukkan bahwa kita bisa melanjutkan ke proses berikutnya, yaitu menghasilkan wajah baru.



Gambar 6.11: Distribusi titik pada 50 sampel dimensi dari ruang laten

6.3.6 Membuat Wajah Baru

Untuk membuat wajah baru, kita bisa menggunakan kode pada Listing 6.8. Kita lihat bagaimana VAE bisa menghasilkan wajah-wajah baru. VAE bisa mengambil titik-titik acak dari distribusi normal standar, lalu mengubahnya menjadi gambar wajah manusia yang terlihat nyata. Ini adalah contoh awal dari kekuatan model generatif.



Gambar 6.12: Contoh Hasil Wajah Baru dari dekoder VAE

```

1 import torch
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from model_celeb import VAE
5

```

```

6 device = torch.device("cuda" if torch.cuda.is_available()
7     () else "cpu")
8 model = VAE().to(device)
9 model.load_state_dict(torch.load("model_celeba_VAE.pth",
10     map_location=device, weights_only=True))
11 model.eval()
12
13 grid_width, grid_height = 10, 3
14 z_dim = 200
15 num_samples = grid_width * grid_height
16 shape_before_flattening = (128, 2, 2) # dari encoder
17
18 z_sample = torch.randn(num_samples, z_dim).to(device)
19
20 with torch.no_grad():
21     generated = model.decoder(z_sample,
22         shape_before_flattening)
23
24 fig = plt.figure(figsize=(18, 5))
25 plt.subplots_adjust(hspace=0.4, wspace=0.4)
26
27 for i in range(num_samples):
28     ax = fig.add_subplot(grid_height, grid_width, i + 1)
29     img = generated[i].cpu().permute(1, 2, 0).numpy()
30     ax.imshow(img)
31     ax.axis("off")
32
33 plt.suptitle("Hasil Wajah Baru dari Ruang Laten",
34     fontsize=16)
35 plt.tight_layout()
36 plt.show()

```

Listing 6.8: Kode untuk menghasilkan wajah baru

6.4 Aplikasi Autoencoder dalam Deteksi Anomali

Anomaly detection, atau deteksi anomali, adalah proses identifikasi pola atau kejadian yang tidak biasa dalam data yang dapat menunjukkan masalah atau perilaku yang tidak diinginkan. Proses ini sangat penting dalam berbagai bidang, termasuk keamanan siber, di mana deteksi anomali dapat membantu mengidentifikasi serangan atau intrusi yang tidak biasa. Dalam pemeliharaan prediktif, deteksi anomali dapat digunakan untuk mendeteksi kerusakan pada mesin sebelum terjadi kegagalan, sehingga mengurangi biaya perbaikan dan waktu henti. Selain itu, dalam konteks deteksi penipuan, seperti dalam transaksi keuangan, anomaly detection dapat membantu mengidentifikasi aktivitas yang mencurigakan yang mungkin menunjukkan penipuan.

Penggunaan autoencoder dalam anomaly detection memiliki beberapa kelebihan, termasuk kemampuannya untuk belajar representasi yang efisien dari data dan fleksibilitas dalam menangani berbagai jenis data. Autoencoder juga dapat beradaptasi dengan perubahan pola dalam data seiring waktu, yang membuatnya sangat berguna dalam aplikasi yang dinamis. Namun, ada juga tantangan yang perlu diperhatikan, seperti sensitivitas terhadap noise dalam data, yang dapat mempengaruhi akurasi deteksi anomali. Selain itu, pemilihan threshold yang tepat untuk menentukan apakah suatu data dianggap anomali juga dapat menjadi tantangan, karena ambang batas yang terlalu ketat dapat menyebabkan banyak false negatives, sementara ambang batas yang terlalu longgar dapat menghasilkan false positives.

Salah satu contoh konkret penggunaan Variational Autoencoder (VAE) dalam deteksi anomali adalah dalam pendekripsi penipuan kartu kredit (Tingfei, Guangquan, & Kuihua, 2020). Dalam konteks ini, VAE dilatih menggunakan data transaksi yang dianggap normal, seperti pola pembelian yang umum dan perilaku pengguna yang wajar. Dengan memanfaatkan pendekatan probabilistik, VAE dapat memetakan data transaksi ke dalam ruang laten yang lebih terstruktur, memungkinkan model untuk memaha-

mi distribusi normal dari transaksi yang sah. Ketika transaksi baru terjadi, VAE dapat mendeteksi anomali dengan membandingkan rekonstruksi transaksi tersebut dengan data asli. Jika transaksi baru menunjukkan perbedaan signifikan dari pola yang telah dipelajari, seperti jumlah yang sangat besar atau lokasi yang tidak biasa, VAE dapat mengidentifikasi transaksi tersebut sebagai potensi penipuan, sehingga memberikan peringatan kepada pihak berwenang untuk melakukan pemeriksaan lebih lanjut.

Untuk melihat contoh aplikasi autoencoder dalam anomaly detection, Anda dapat mengunjungi <https://s.foxecho.id/PML-I-AE>.