Rozszerzenie *Variational Autoencoder (VAE)* do generowania bardziej złożonych danych

Mateusz Kacpura

17 listopada 2024

Streszczenie

Celem niniejszej pracy jest rozwinięcie wcześniej zbudowanego modelu *Variational Autoencoder* (VAE) w celu lepszego uchwycenia i generowania bardziej złożonych danych. Dokonano tego poprzez modyfikację architektury modelu, eksperymenty z różnymi parametrami oraz analizę wpływu tych zmian na jakość generowanych danych. Napotkano szereg problemów podczas implementacji modelu zdolnego do trenowania na różnych zbiorach danych, które zostały opisane wraz z zastosowanymi rozwiązaniami. Przeprowadzono także interpolacje w przestrzeni latentnej oraz wizualizację przestrzeni latentnej przy użyciu technik redukcji wymiarów.

Spis treści

1 Wstęp **3**
1.1 Motywacja . . . 3
1.2 Cel pracy . . . 3

2 Wcześniejszy model VAE **4**
2.1 Opis modelu . . . 4
2.2 Implementacja . . . 4
2.3 Ograniczenia modelu . . . 5

3 Rozwinięcie modelu VAE i napotkane problemy **6**
3.1 Wybór nowego zbioru danych . . . 6
3.1.1 Przygotowanie danych . . . 6
3.2 Zmiana architektury modelu . . . 8
3.2.1 Konwolucyjne VAE . . . 8
3.3 Napotkano liczne problemy, które rozwiązano . . . 13
3.3.1 Dodatkowy wymiar podczas treningu . . . 13
3.3.2 Użycie funkcji straty `MeanSquaredError` . . . 13
3.3.3 Niedopasowanie wymiarów wyjściowych dekodera . . . 13
3.3.4 Niewłaściwe użycie parametru `output_padding` . . . 13
3.3.5 Spójna obsługa danych w metodach `train_step` i `test_step` . . . 13
3.4 Eksperymenty z funkcją straty . . . 14
3.4.1 Wpływ wagi dywergencji KL . . . 14
3.4.2 Dodatkowy term w funkcji straty . . . 14
3.5 Interpolacje w przestrzeni latentnej . . . 14
3.6 Wizualizacja przestrzeni latentnej . . . 15

4 Eksperymenty i wyniki **16**
4.1 Wyniki dla różnych zbiorów danych . . . 16
4.1.1 Fashion MNIST . . . 16
4.1.2 CIFAR-10 . . . 17
4.1.3 CelebA . . . 19
4.2 Wpływ parametrów modelu . . . 20
4.2.1 Wymiar przestrzeni latentnej . . . 20
4.2.2 Waga dywergencji KL . . . 20
4.3 Interpolacje i wizualizacja przestrzeni latentnej . . . 20

5 Wnioski **22**

6 Prace przyszłe **22**

1 Wstęp

1.1 Motywacja

Modele generatywne, takie jak *Variational Autoencoder* (VAE), odgrywają kluczową rolę w dziedzinie uczenia maszynowego, umożliwiając modelowanie skomplikowanych rozkładów danych [1]. Dotychczasowe modele VAE były stosowane głównie do prostych zbiorów danych, takich jak MNIST, co ograniczało ich zdolność do generowania bardziej złożonych struktur.

1.2 Cel pracy

Celem pracy jest rozwinięcie wcześniejszego modelu VAE tak, aby lepiej uchwycić i generować bardziej złożone dane. W ramach pracy:

1. **Opis wcześniejszego modelu VAE** Prezentacja gęstych warstw i wcześniejszego kodu
2. **Wybór nowego zbioru danych**: Rozszerzenie modelu na bardziej złożone zbiory danych, takie jak **Fashion MNIST**, **CIFAR-10**, **CIFAR-100** oraz **CelebA**.
3. **Zmiana architektury modelu**:
   - Wprowadzenie *konwolucyjnych VAE* poprzez dodanie warstw konwolucyjnych do enkodera i dekodera.
   - Pogłębienie sieci przez dodanie większej liczby warstw i jednostek ukrytych.
   - Eksperymenty z różnymi wymiarami przestrzeni latentnej z.
4. **Eksperymenty z funkcją straty**:
   - Analiza wpływu zmiany wagi komponentu dywergencji Kullbacka-Leiblera (KL) na balans między rekonstrukcją a regularnością przestrzeni latentnej.
   - Dodanie dodatkowego termu w funkcji straty, kładącego większy nacisk na rekonstrukcję detali.
5. **Rozwiązanie napotkanych problemów**: Omówienie problemów pojawiających się podczas implementacji modelu zdolnego do trenowania na różnych zbiorach danych oraz przedstawienie zastosowanych rozwiązań.
6. **Interpolacje w przestrzeni latentnej**: Wygenerowanie próbek obrazów, które są interpolacją pomiędzy dwoma punktami w przestrzeni latentnej, aby obserwować przejścia między obiektami.
7. **Wizualizacja przestrzeni latentnej**: Redukcja wymiarów przestrzeni latentnej do 2D za pomocą metod takich jak PCA lub t-SNE, w celu zobaczenia, jak VAE rozdziela różne klasy obiektów.

2 Wcześniejszy model VAE

2.1 Opis modelu

Pierwotny model VAE został zbudowany i przetestowany na zbiorze danych MNIST [2], zawierającym ręcznie pisane cyfry. Model wykorzystywał jedynie warstwy gęste (*Dense Layers*) w enkoderze i dekoderze.

2.2 Implementacja

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Layer, Dense
from tensorflow.keras.models import Sequential, Model

# 1. Załadowanie i przygotowanie danych MNIST
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train_d = np.reshape(x_train, (x_train.shape[0], -1))
x_test_d = np.reshape(x_test, (x_test.shape[0], -1))

# 2. Definicja warstwy enkodera
class Encoder(Layer):
    def __init__(self, architecture):
        super(Encoder, self).__init__()
        self.model = Sequential([Dense(ni, activation='relu') for ni in
            architecture])

    def call(self, x):
        return self.model(x)

# 3. Definicja warstwy dekodera
class Decoder(Layer):
    def __init__(self, architecture, original_dim):
        super(Decoder, self).__init__()
        layers = [Dense(ni, activation='relu') for ni in architecture]
        layers.append(Dense(original_dim, activation='sigmoid'))
        self.model = Sequential(layers)

    def call(self, x):
        return self.model(x)

# 4. Definicja modelu Autoenkodera
class Autoencoder(Model):
    def __init__(self, encoder_architecture, decoder_architecture,
        original_dim):
        super(Autoencoder, self).__init__()
        self.encoder = Encoder(encoder_architecture)
        self.decoder = Decoder(decoder_architecture, original_dim)

    def call(self, x):
        z = self.encoder(x)
        y = self.decoder(z)
        return y

# 5. Inicjalizacja Autoenkodera
original_dim = x_train_d.shape[1]  # 784
autoencoder = Autoencoder(encoder_architecture=[64, 32, 16],
                          decoder_architecture=[16, 32, 64],
                          original_dim=original_dim)

# 6. Kompilacja modelu
```

```
autoencoder.compile(optimizer='adam',
                    loss='mse',
                    metrics=['mse'])

# 7. Trening Autoenkodera przy użyciu model.fit
history = autoencoder.fit(x_train_d, x_train_d,
                          epochs=20,
                          batch_size=256,
                          shuffle=True,
                          validation_data=(x_test_d, x_test_d))

# 8. Rekonstrukcja obrazów testowych
x_test_reconstructed = autoencoder.predict(x_test_d)

# 9. Wyświetlenie oryginalnych i zrekonstruowanych obrazów
def plot_reconstructions(x_original, x_reconstructed, num_images=10):
    plt.figure(figsize=(20, 4))
    for i in range(num_images):
        # Wyświetlenie oryginalnych obrazów
        ax = plt.subplot(2, num_images, i + 1)
        plt.imshow(x_original[i], cmap='gray')
        ax.axis('off')

        # Wyświetlenie zrekonstruowanych obrazów
        ax = plt.subplot(2, num_images, i + 1 + num_images)
        plt.imshow(x_reconstructed[i].reshape(28, 28), cmap='gray')
        ax.axis('off')
    plt.tight_layout()
    plt.show()

plot_reconstructions(x_test, x_test_reconstructed, num_images=10)
```

Listing 1: Implementacja podstawowego VAE na zbiorze MNIST

2.3 Ograniczenia modelu

Model ten sprawdza się dobrze dla prostych danych, takich jak ręcznie pisane cyfry z MNIST. Jednak jego możliwości są ograniczone w przypadku bardziej złożonych danych ze względu na:

- Brak warstw konwolucyjnych, co utrudnia uchwycenie lokalnych wzorców w danych obrazowych.
- Ograniczoną głębokość sieci, co wpływa na zdolność modelu do reprezentowania skomplikowanych zależności.
- Statyczną architekturę, która nie pozwala na łatwą adaptację do różnorodnych zbiorów danych.

3 Rozwinięcie modelu VAE i napotkane problemy

3.1 Wybór nowego zbioru danych

Aby zwiększyć złożoność zadania i przetestować zdolności modelu, wybrano trzy zbiory danych:

1. **Fashion MNIST** [3]: Zbiór zawierający obrazy ubrań, który jest bardziej złożony niż cyfry z MNIST.
2. **CIFAR-10** [4]: Zawierający kolorowe obrazy 32x32 pikseli z 10 klasami obiektów.
3. **CelebA** [5]: Duży zbiór danych zawierający obrazy twarzy z różnymi atrybutami.

3.1.1 Przygotowanie danych

Fashion MNIST

```python
from tensorflow.keras.datasets import fashion_mnist

# Załadowanie danych
(x_train_fashion, _), (x_test_fashion, _) = fashion_mnist.load_data()

# Normalizacja i zmiana kształtu
x_train_fashion = x_train_fashion.astype('float32') / 255.
x_test_fashion = x_test_fashion.astype('float32') / 255.
x_train_fashion = np.expand_dims(x_train_fashion, -1)
x_test_fashion = np.expand_dims(x_test_fashion, -1)
```

Listing 2: Ładowanie zbioru Fashion MNIST

Dane zostały wczytane w formacie 28x28 w skali szarości, znormalizowane do zakresu (0, 1) i podzielone na zbiór treningowy i testowy o kolejno liczebności 60000 i 10000 obrazów. Szczegółowy kod został przedstawiony w sekcji 2.

CIFAR-10

```python
from tensorflow.keras.datasets import cifar10

# Załadowanie danych
(x_train_cifar, _), (x_test_cifar, _) = cifar10.load_data()

# Normalizacja
x_train_cifar = x_train_cifar.astype('float32') / 255.
x_test_cifar = x_test_cifar.astype('float32') / 255.
```

Listing 3: Ładowanie zbioru CIFAR-10

Dane zostały wczytane w formacie 32x32 w kolorze, znormalizowane do zakresu (0, 1) i podzielone na zbiór treningowy i testowy o kolejno liczebności 50000 i 10000 obrazów. Szczegółowy kod został przedstawiony w sekcji 3.

CelebA

```python
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import glob
import os

# Bezpośredni link do pliku ZIP
file_id = '1A2dNWabg6_um-V3lhw1tyead5hCpjaW8'
zip_url = f'https://drive.google.com/uc?id={file_id}'

# Ścieżka do pobranego pliku ZIP i lokalizacja rozpakowania
zip_path = '/content/image.zip'
local_extract_path = '/content/images'

# Pobieranie pliku ZIP
if not os.path.exists(zip_path):  # Sprawdzanie, czy plik ZIP już istnieje
    print(f"Pobieranie pliku ZIP z Google Drive: {zip_url}...")
    gdown.download(zip_url, zip_path, quiet=False)
else:
    print(f"Plik ZIP już istnieje: {zip_path}")
```

```
19
20 # Rozpakowanie archiwum
21 if not os.path.exists(local_extract_path):  # Sprawdzanie, czy katalog już
       istnieje
22     os.makedirs(local_extract_path)  # Tworzenie katalogu, jeśli nie istnieje
23     print(f"Rozpakowywanie pliku ZIP: {zip_path} do {local_extract_path}...")
24     with zipfile.ZipFile(zip_path, 'r') as zip_ref:
25         zip_ref.extractall(local_extract_path)
26     print("Rozpakowywanie zakończone!")
27 else:
28     print(f"Katalog docelowy już istnieje: {local_extract_path}")
29
30 import zipfile  # Dodano moduł zipfile
31
32 celeba_path = '/content/images/image'
33
34 def load_celeba_images(path, img_size=(64, 64)):
35     image_paths = glob.glob(os.path.join(path, '*.jpg'))
36     images = []
37     for img_path in image_paths:
38         img = load_img(img_path, target_size=img_size)
39         img = img_to_array(img) / 255.
40         images.append(img)
41     return np.array(images)
42
43 # Load CelebA data
44 x_celeba = load_celeba_images(celeba_path)
45 x_celeba = (x_celeba - 0.5) * 2 # normalizacja danych
46 print(f"x_celeba shape: {x_celeba.shape}")  # e.g., (30000, 64, 64, 3)
47
48 # Split into training and validation sets
49 x_train_celeba, x_val_celeba = train_test_split(x_celeba, test_size=0.3,
       random_state=42)
```

Listing 4: Przygotowanie danych CelebA

Dane zostały wczytane w formacie 64x64 w kolorze. Proces przygotowania danych CelebA obejmował pobranie danych z Dysku Google w środowisku Colab, rozpakowanie archiwum z obrazami oraz wczytanie ścieżek do plików graficznych. Normalizację danych. Podział danych na zbiór treningowy i walidacyjny, gdzie kolejno zawierały one 21000 i 9000. Szczegółowy kod został przedstawiony w sekcji 4.

3.2 Zmiana architektury modelu

3.2.1 Konwolucyjne VAE

Aby lepiej uchwycić lokalne wzorce w danych obrazowych, wprowadzono warstwy konwolucyjne do enkodera i dekodera.

```
def build_conv_encoder(input_shape, latent_dim):
    inputs = Input(shape=input_shape)
    x = inputs

    # Dynamiczne obliczanie liczby warstw konwolucyjnych
    num_conv_layers = int(np.floor(np.log2(min(input_shape[0],
        input_shape[1])))) - 2)
    num_conv_layers = max(num_conv_layers, 1)
    conv_filters = [32, 64, 128, 256][:num_conv_layers]

    for filters in conv_filters:
        x = Conv2D(filters, kernel_size=3, strides=2, padding='same')(x)
        x = BatchNormalization()(x)
        x = LeakyReLU()(x)

    x_shape = tf.keras.backend.int_shape(x)[1:]
    x = Flatten()(x)
    x = Dense(512)(x)
    x = LeakyReLU()(x)

    z_mean = Dense(latent_dim, name='z_mean')(x)
    z_log_var = Dense(latent_dim, name='z_log_var')(x)
    z = Sampling()([z_mean, z_log_var])
    encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
    return encoder, x_shape, num_conv_layers
```

Listing 5: Definicja konwolucyjnego enkodera

Opis argumentów wejściowych

- **input_shape**: Krotka określająca kształt danych wejściowych, zazwyczaj w formacie `(wysokość, szerokość, kanały)`.
- **latent_dim**: Liczba wymiarów przestrzeni latentnej, do której dane wejściowe będą mapowane.

Szczegółowy opis działania

1. **Inicjalizacja wejść**:
   - Tworzenie warstwy wejściowej modelu z określonym kształtem danych.
   - Przypisanie tej warstwy do zmiennej `x`, która będzie modyfikowana w kolejnych krokach.
2. **Dynamiczne obliczanie liczby warstw konwolucyjnych**:
   - Obliczanie liczby warstw konwolucyjnych na podstawie mniejszego wymiaru obrazu wejściowego.
   - Użycie funkcji logarytmu dwójkowego i zaokrąglenie w dół, a następnie odjęcie 2, aby określić ilość warstw.
   - Zapewnienie, że liczba warstw jest co najmniej 1.
   - Przydzielenie odpowiedniej liczby filtrów dla każdej warstwy konwolucyjnej, korzystając z listy `[32, 64, 128, 256]`.
3. **Dodawanie warstw konwolucyjnych**:
   - Iteracyjne dodawanie warstw `Conv2D` z określoną liczbą filtrów, rozmiarem jądra 3, krokiem 2 i paddingiem 'same'.

- Dodawanie warstwy normalizacji wsadowej (`BatchNormalization`).
- Zastosowanie aktywacji `LeakyReLU` dla wprowadzenia nieliniowości.

4. **Przygotowanie warstwy gęstej**:
   - Uzyskanie kształtu danych po warstwach konwolucyjnych.
   - Spłaszczanie danych za pomocą warstwy `Flatten`.
   - Dodanie warstwy gęstej (`Dense`) z 512 jednostkami.
   - Zastosowanie aktywacji `LeakyReLU`.
5. **Tworzenie przestrzeni latentnej**:
   - Generowanie średniej (`z_mean`) dla przestrzeni latentnej za pomocą warstwy `Dense`.
   - Generowanie logarytmu wariancji (`z_log_var`) za pomocą warstwy `Dense`.
   - Próbkowanie wektora latentnego `z` przy użyciu warstwy `Sampling`, która wykorzystuje `z_mean` i `z_log_var`.
6. **Budowanie modelu enkodera**:
   - Utworzenie modelu `encoder` za pomocą funkcji `Model` Keras, który przyjmuje jako wejście warstwę `inputs` i zwraca `z_mean`, `z_log_var` oraz próbkę `z`.
   - Zwrócenie modelu enkodera wraz z kształtem danych po warstwach konwolucyjnych oraz liczbą warstw konwolucyjnych.

Główne elementy techniczne

- **Warstwy konwolucyjne (`Conv2D`)**: Służą do ekstrakcji cech z danych wejściowych poprzez zastosowanie filtrów konwolucyjnych.
- **Normalizacja wsadowa (`BatchNormalization`)**: Stabilizuje proces uczenia poprzez normalizację aktywacji warstw.
- **Aktywacja (`LeakyReLU`)**: Wprowadza nieliniowość, pozwalając na małe wartości gradientów dla ujemnych wejść, co pomaga w uniknięciu problemu znikających gradientów.
- **Warstwa (`Flatten`)**: Spłaszcza wielowymiarowe dane do jednowymiarowej postaci, umożliwiając przejście do warstw gęstych.
- **Warstwy gęste (`Dense`)**: Przekształcają dane do przestrzeni latentnej.
- **Warstwa (`Sampling`)**: Implementuje proces próbkowania w przestrzeni latentnej, umożliwiając generowanie nowych prób z rozkładu.
- **Model keras (`Model`)**: Tworzy całościowy model enkodera, łącząc wszystkie zdefiniowane warstwy.

Zwracane wartości

Funkcja `build_conv_encoder` zwraca trzy wartości:

1. **encoder**: Model enkodera konwolucyjnego.
2. **x_shape**: Kształt danych po przejściu przez warstwy konwolucyjne, przed spłaszczaniem.
3. **num_conv_layers**: Liczba dodanych warstw konwolucyjnych.

```
def build_conv_decoder(x_shape, latent_dim, output_shape, num_conv_layers):
    latent_inputs = Input(shape=(latent_dim,))
    x = Dense(512)(latent_inputs)
    x = LeakyReLU()(x)
    x = Dense(np.prod(x_shape))(x)
    x = LeakyReLU()(x)
    x = Reshape(x_shape)(x)
```

```
8
9       conv_filters = [256, 128, 64, 32][:num_conv_layers][::-1]
10
11      for filters in conv_filters:
12          x = Conv2DTranspose(filters, kernel_size=3, strides=2,
                padding='same')(x)
13          x = BatchNormalization()(x)
14          x = LeakyReLU()(x)
15
16      x = Conv2DTranspose(
17          output_shape[2],
18          kernel_size=3,
19          strides=1,
20          activation='sigmoid',
21          padding='same'
22      )(x)
23
24      target_height, target_width = output_shape[0], output_shape[1]
25      final_height, final_width = tf.keras.backend.int_shape(x)[1],
            tf.keras.backend.int_shape(x)[2]
26      diff_height = final_height - target_height
27      diff_width = final_width - target_width
28
29      if diff_height > 0 or diff_width > 0:
30          cropping = (
31              (diff_height // 2, diff_height - diff_height // 2),
32              (diff_width // 2, diff_width - diff_width // 2)
33          )
34          x = Cropping2D(cropping=cropping)(x)
35      elif diff_height < 0 or diff_width < 0:
36          padding = (
37              (-diff_height // 2, -diff_height - (-diff_height // 2)),
38              (-diff_width // 2, -diff_width - (-diff_width // 2))
39          )
40          x = ZeroPadding2D(padding=padding)(x)
41
42      decoder = Model(latent_inputs, x, name='decoder')
43      return decoder
```

Listing 6: Definicja konwolucyjnego dekodera

Opis argumentów wejściowych

- **x_shape**: Kształt danych wejściowych do dekodera przed przekształceniem przez warstwy gęste, zazwyczaj wynik warstw konwolucyjnych enkodera, w formacie `(wysokość, szerokość, kanały)`.
- **latent_dim**: Liczba wymiarów przestrzeni latentnej, z której dekoder będzie generować dane.
- **output_shape**: Kształt danych wyjściowych, do których dekoder będzie dążył, zazwyczaj w formacie `(wysokość, szerokość, kanały)`.
- **num_conv_layers**: Liczba warstw konwolucyjnych w dekoderze, która powinna odpowiadać liczbie warstw w enkoderze.

Szczegółowy opis działania

1. **Inicjalizacja wejść**:
   - Tworzenie warstwy wejściowej dla dekodera z określonym wymiarem latentnym `latent_dim`.
   - Przypisanie tej warstwy do zmiennej `latent_inputs`, która będzie modyfikowana w kolejnych krokach.
2. **Przekształcenia gęste**:

- Dodanie warstwy gęstej (`Dense`) z 512 jednostkami, przekształcającej wektor latentny.
- Zastosowanie aktywacji `LeakyReLU` w celu wprowadzenia nieliniowości.
- Dodanie kolejnej warstwy gęstej przekształcającej do liczby jednostek równej iloczynowi wymiarów `x_shape`, co przygotowuje dane do przekształcenia w odpowiedni kształt.
- Ponowne zastosowanie aktywacji `LeakyReLU`.
- Przekształcenie danych do oryginalnego kształtu `x_shape` za pomocą warstwy `Reshape`.

3. **Określenie filtrów konwolucyjnych**:
   - Definiowanie listy filtrów `conv_filters` dla warstw konwolucyjnych dekodera.
   - Wykorzystanie listy `[256, 128, 64, 32]` ograniczonej do `num_conv_layers` elementów i odwrócenie jej kolejności, aby filtrowanie odbywało się w odwrotnej kolejności niż w enkoderze.
4. **Dodawanie warstw konwolucyjnych transponowanych**:
   - Iteracyjne dodawanie warstw `Conv2DTranspose` z określoną liczbą filtrów, rozmiarem jądra 3, krokiem 2 i paddingiem 'same' w celu zwiększenia wymiarów przestrzennych danych.
   - Dodawanie warstwy normalizacji wsadowej (`BatchNormalization`).
   - Zastosowanie aktywacji `LeakyReLU` dla wprowadzenia nieliniowości.
5. **Warstwa wyjściowa**:
   - Dodanie ostatniej warstwy `Conv2DTranspose` z liczbą filtrów równą liczbie kanałów w `output_shape`, rozmiarem jądra 3, krokiem 1, aktywacją 'sigmoid' i paddingiem 'same' w celu uzyskania ostatecznego obrazu wyjściowego.
6. **Dopasowanie kształtu wyjściowego**:
   - Pobranie docelowej wysokości i szerokości z `output_shape`.
   - Uzyskanie aktualnej wysokości i szerokości po ostatnich warstwach konwolucyjnych (`final_height` i `final_width`).
   - Obliczenie różnic `diff_height` i `diff_width` między aktualnymi a docelowymi wymiarami.
   - Jeżeli różnica jest większa niż zero, zastosowanie warstwy `Cropping2D` w celu przycięcia nadmiarowych pikseli.
   - Jeżeli różnica jest mniejsza niż zero, zastosowanie warstwy `ZeroPadding2D` w celu dodania brakujących pikseli.
7. **Budowanie modelu dekodera**:
   - Utworzenie modelu `decoder` za pomocą funkcji `Model` Keras, który przyjmuje jako wejście warstwę `latent_inputs` i zwraca przekształcony obraz wyjściowy `x`.
   - Zwrócenie modelu dekodera.

Główne elementy techniczne

- **Warstwy gęste (`Dense`)**: Przekształcają dane z przestrzeni latentnej do wysokowymiarowej przestrzeni, która jest następnie przekształcana przez warstwy konwolucyjne.
- **Aktywacja (`LeakyReLU`)**: Wprowadza nieliniowość, pozwalając na małe wartości gradientów dla ujemnych wejść, co pomaga w uniknięciu problemu znikających gradientów.
- **Warstwa (`Reshape`)**: Zmienia kształt danych z jednowymiarowego wektora do wielowymiarowego tensora, przygotowując go do przekształceń konwolucyjnych.
- **Warstwy konwolucyjne transponowane (`Conv2DTranspose`)**: Służą do przekształcania danych w kierunku od przestrzeni latentnej do przestrzeni wyjściowej poprzez zwiększanie wymiarów przestrzennych.
- **Normalizacja wsadowa (`BatchNormalization`)**: Stabilizuje proces uczenia poprzez normalizację aktywacji warstw.

- **Warstwa wyjściowa (`Conv2DTranspose z aktywacją 'sigmoid'`)**: Generuje ostateczny obraz wyjściowy z wartościami pikseli w zakresie `[0, 1]`.
- **Warstwy dopasowujące kształt (`Cropping2D`, `ZeroPadding2D`)**: Zapewniają, że ostateczny obraz wyjściowy ma dokładnie pożądany rozmiar, poprzez odpowiednie przycinanie lub dodawanie pikseli.
- **Model keras (`Model`)**: Tworzy całościowy model dekodera, łącząc wszystkie zdefiniowane warstwy.

Zwracane wartości

Funkcja `build_conv_decoder` zwraca jeden element:

1. **decoder**: Model dekodera konwolucyjnego, który przekształca reprezentację latentną w dane wyjściowe o zadanym kształcie.

Kluczowe zmiany

1. **Głębsza sieć**: Dodano większą liczbę warstw i jednostek ukrytych w obu sieciach, co pozwala modelowi uczyć się bardziej złożonych reprezentacji.
2. **Eksperymenty z wymiarem przestrzeni latentnej** z Przeprowadzono eksperymenty z wymiarami $z \in \{16, 32, 64, 128, 256\}$, aby zbadać wpływ rozmiaru przestrzeni latentnej na jakość generacji.

3.3 Napotkano liczne problemy, które rozwiązano

Podczas implementacji modelu zdolnego do trenowania na różnych zbiorach danych napotkano szereg problemów:

3.3.1 Dodatkowy wymiar podczas treningu

Problem:
Wystąpił błąd związany z dodatkowym wymiarem podczas treningu, spowodowany przekazaniem danych walidacyjnych w formacie `(x_val, None)`.

Rozwiązanie:
Zmieniono przekazywanie danych walidacyjnych na `validation_data=(x_val,)` lub `validation_data=x_val`, aby uniknąć problemów z wymiarami.

3.3.2 Użycie funkcji straty `MeanSquaredError`

Problem:
Podczas korzystania z klasy `MeanSquaredError` w niestandardowej pętli treningowej pojawił się błąd `OperatorNotAllowedInGraphError`.

Rozwiązanie:
Zamiast korzystać z klasy `MeanSquaredError`, obliczono błąd średniokwadratowy bezpośrednio za pomocą operacji TensorFlow.

```
reconstruction_loss = tf.reduce_mean(
    tf.reduce_sum(
        tf.math.squared_difference(data, reconstruction),
        axis=[1, 2, 3]
    )
)
```

Listing 7: Bezpośrednie obliczanie straty rekonstrukcji

3.3.3 Niedopasowanie wymiarów wyjściowych dekodera

Problem:
Dekoder generował wyjścia o wymiarach niezgodnych z oryginalnymi wymiarami danych wejściowych, co prowadziło do błędów podczas obliczania straty.

Rozwiązanie:
Dostosowano architekturę dekodera, aby dynamicznie dostosowywała się do wymiarów danych wejściowych. Wykorzystano warstwy `Cropping2D` i `ZeroPadding2D` do precyzyjnego dopasowania wymiarów.

3.3.4 Niewłaściwe użycie parametru `output_padding`

Problem:
Użycie parametru `output_padding` z wartością zero lub ujemną w warstwie `Conv2DTranspose` powodowało błędy.

Rozwiązanie:
Usunięto parametr `output_padding` z warstw `Conv2DTranspose` i zamiast tego wykorzystano warstwy `Cropping2D` oraz `ZeroPadding2D` do regulacji wymiarów.

3.3.5 Spójna obsługa danych w metodach `train_step` i `test_step`

Problem:
Niespójna obsługa danych podczas treningu i testowania prowadziła do błędów.

Rozwiązanie:
W metodach `train_step` i `test_step` dodano sprawdzenie, czy wejście `data` jest krotką, i wyodrębniono dane wejściowe.

```python
def train_step(self, data):
    if isinstance(data, tuple):
        data = data[0]
    # Kontynuacja implementacji...
```

Listing 8: Metoda `train_step`

3.4 Eksperymenty z funkcją straty

3.4.1 Wpływ wagi dywergencji KL

Podczas treningu VAE kluczowe jest właściwe zbalansowanie wpływu dywergencji Kullbacka-Leiblera (KL) i straty rekonstrukcji. W tym celu przeprowadzono eksperymenty z różnymi wartościami wagi dywergencji KL (`kl_weight`), aby zbadać wpływ na balans między dokładnością rekonstrukcji a regularnością przestrzeni latentnej.

Testowano wartości `kl_weight` w przedziale od 0.0001 do 0.01. Zaobserwowano, że mniejsze wartości `kl_weight` skutkują lepszą jakością rekonstrukcji, jednak kosztem mniejszej regularności przestrzeni latentnej, co może prowadzić do gorszej jakości generowanych próbek spoza rozkładu treningowego. Większe wartości `kl_weight` promują bardziej ujednoliconą przestrzeń latentną, ale mogą powodować rozmycie rekonstrukcji.

3.4.2 Dodatkowy term w funkcji straty

Aby zwiększyć nacisk na rekonstrukcję detali w obrazach, wprowadzono dodatkowy komponent do funkcji straty oparty na miarze *Structural Similarity Index Measure* (SSIM). SSIM jest miarą podobieństwa między dwoma obrazami, uwzględniającą aspekty luminancji, kontrastu i struktury [6]. Dodanie termu SSIM do straty pozwala na uzyskanie rekonstrukcji o wyższej jakości percepcyjnej.

Całkowita funkcja straty została zdefiniowana jako:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruction}} + \beta \cdot \mathcal{L}_{\text{KL}} + \gamma \cdot \mathcal{L}_{\text{SSIM}} \tag{1}$$

gdzie:

- $\mathcal{L}_{\text{reconstruction}}$ - strata rekonstrukcji (np. błąd średniokwadratowy),
- $\mathcal{L}_{\text{KL}}$ - dywergencja KL,
- $\mathcal{L}_{\text{SSIM}}$ - komponent oparty na SSIM,
- β, γ - wagi poszczególnych komponentów.

3.5 Interpolacje w przestrzeni latentnej

Przeprowadzono interpolacje w przestrzeni latentnej między parami obrazów z różnych klas, co pozwoliło zaobserwować płynne przejścia między obiektami. Wykorzystano liniową interpolację wektora latentnego $\mathbf{z}$:

$$\mathbf{z}_{\text{interp}} = (1 - \alpha) \cdot \mathbf{z}_1 + \alpha \cdot \mathbf{z}_2, \quad \alpha \in [0, 1] \tag{2}$$

Dekodując $\mathbf{z}_{\text{interp}}$, uzyskano obrazy będące przejściami między dwoma początkowymi obrazami.

3.6 Wizualizacja przestrzeni latentnej

Aby zbadać strukturę przestrzeni latentnej, zastosowano metodę t-SNE [7] do redukcji wymiarowości wektorów latentnych do 2D. Pozwoliło to na wizualną ocenę, czy model grupuje podobne obrazy w bliskich obszarach przestrzeni latentnej.

```
def visualize_latent_space(encoder, x_data, dataset_name):
    z_mean, _, _ = encoder.predict(x_data)
    if z_mean.shape[1] > 2:
        # Use t-SNE to reduce dimensions to 2D
        tsne = TSNE(n_components=2, verbose=1)
        z_tsne = tsne.fit_transform(z_mean)
        plt.figure(figsize=(8,6))
        plt.scatter(z_tsne[:, 0], z_tsne[:, 1], s=2)
        plt.title(f'2D t-SNE of Latent Space on {dataset_name}')
        plt.xlabel('t-SNE Dim 1')
        plt.ylabel('t-SNE Dim 2')
        plt.show()
    else:
        plt.figure(figsize=(8,6))
        plt.scatter(z_mean[:, 0], z_mean[:, 1], s=2)
        plt.title(f'Latent Space on {dataset_name}')
        plt.xlabel('Latent Dim 1')
        plt.ylabel('Latent Dim 2')
        plt.show()
```

Listing 9: Funkcja wizualizacji przestrzeni latalnej `visualize_latent_space`

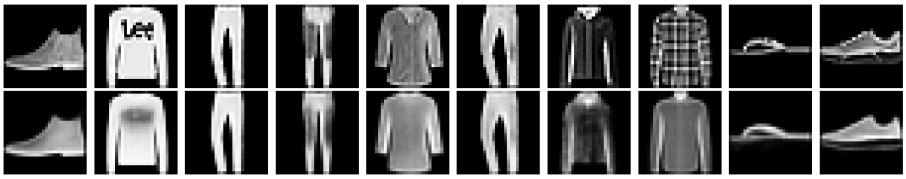
4 Eksperymenty i wyniki

4.1 Wyniki dla różnych zbiorów danych

Przeprowadzono trening modelu na wybranych zbiorach danych, analizując jakość rekonstrukcji oraz zdolności generacyjne.

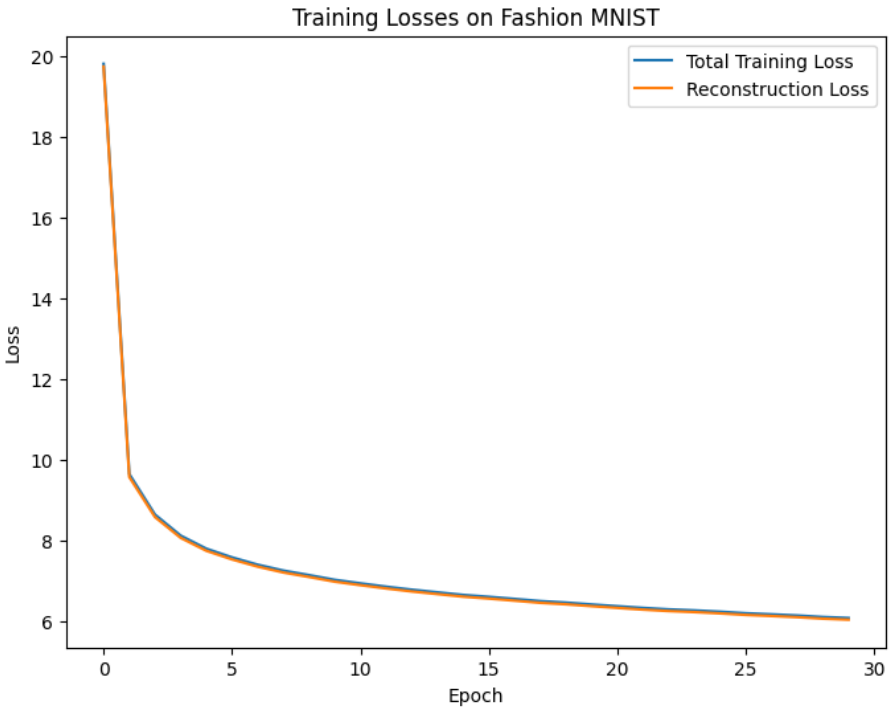
4.1.1 Fashion MNIST

Model osiągnął dobrą jakość rekonstrukcji, zachowując detale ubrań. Interpolacje między różnymi klasami pokazały płynne przejścia.



Rysunek 1: Porównanie oryginalnych i zrekonstruowanych obrazów Fashion MNIST. Górny rząd: oryginały; dolny rząd: rekonstrukcje.

Przebieg treningu modelu przedstawiono na Rysunku 2.



Rysunek 2: Wykres przebiegu treningu modelu Fashion MNIST.

Log treningu

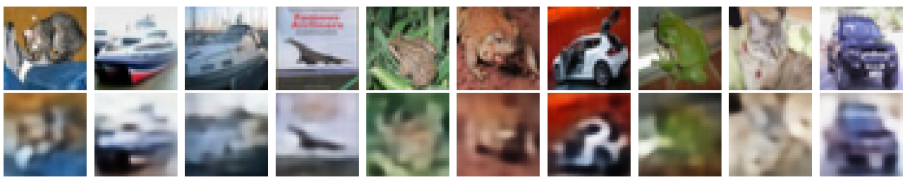
```
# Trening VAE na zbiorze Fashion MNIST
vae_fashion = train_vae(
    input_shape=(28, 28, 1),
    latent_dim=16,
    x_train=x_train_fashion,
    x_val=x_test_fashion,
    epochs=30,
    batch_size=128,
    dataset_name='Fashion MNIST'
)

Epoch 1/30
469/469 [==============================] - 36s 49ms/step - kl_loss: 98.9796 -
    loss: 35.9067 - reconstruction_loss: 35.8572 - val_kl_loss: 163.2720 -
    val_loss: 10.7505 - val_reconstruction_loss: 10.6688
...
Epoch 30/30
469/469 [==============================] - 14s 30ms/step - kl_loss: 84.7810 -
    loss: 6.0669 - reconstruction_loss: 6.0245 - val_kl_loss: 84.2304 -
    val_loss: 6.6343 - val_reconstruction_loss: 6.5922
```

Listing 10: Przebieg treningu VAE na zbiorze Fashion MNIST

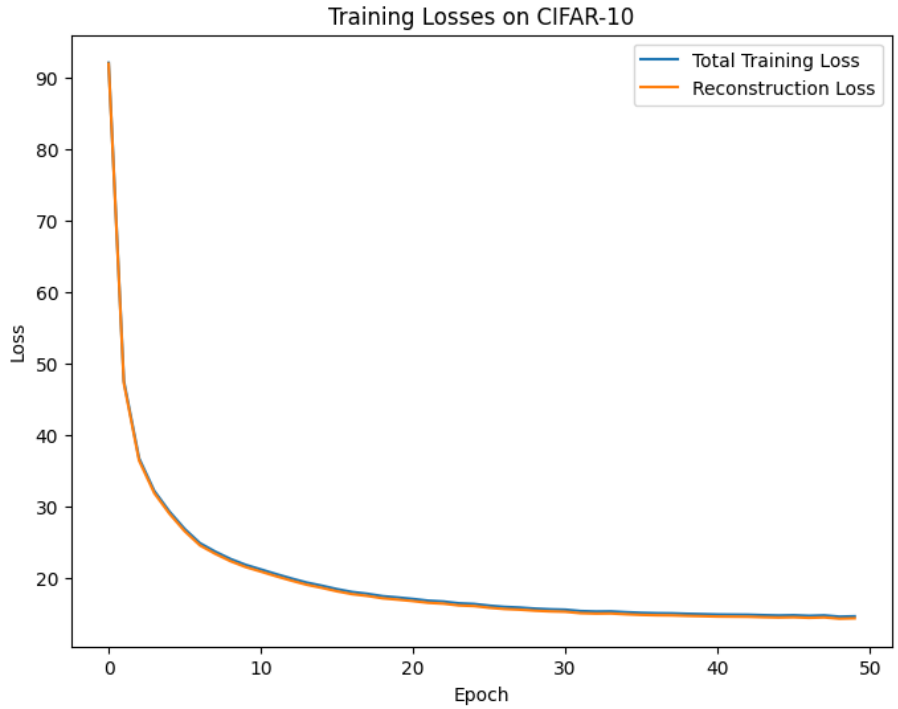
4.1.2 CIFAR-10

Ze względu na większą złożoność danych, model wymagał głębszej architektury i większego wymiaru przestrzeni latentnej. Rekonstrukcje były mniej dokładne niż dla Fashion MNIST.



Rysunek 3: Porównanie oryginalnych i zrekonstruowanych obrazów CIFAR-10. Górny rząd: oryginały; dolny rząd: rekonstrukcje.

Przebieg treningu modelu przedstawiono na Rysunku 4.



Rysunek 4: Wykres przebiegu treningu modelu CIFAR-10.

Log treningu

```
# Trening VAE na zbiorze CIFAR-10
vae_cifar = train_vae(
    input_shape=(32, 32, 3),
    latent_dim=128,
    x_train=x_train_cifar,
    x_val=x_test_cifar,
    epochs=50,
    batch_size=256,
    dataset_name='CIFAR-10'
)

Epoch 1/50
196/196 [==============================] - 40s 133ms/step - kl_loss: 130.9861 -
    loss: 128.5716 - reconstruction_loss: 128.5061 - val_kl_loss: 594.9145 -
    val_loss: 59.4751 - val_reconstruction_loss: 59.1777
...
Epoch 50/50
196/196 [==============================] - 21s 76ms/step - kl_loss: 568.1981 -
    loss: 14.7158 - reconstruction_loss: 14.4317 - val_kl_loss: 568.6900 -
    val_loss: 14.8200 - val_reconstruction_loss: 14.5357
```

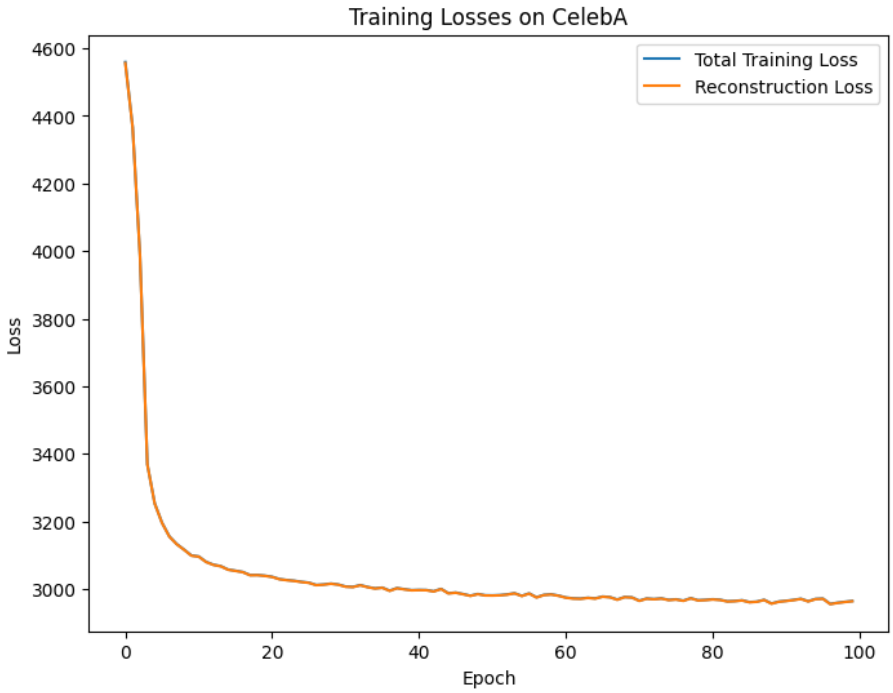
Listing 11: Przebieg treningu VAE na zbiorze CIFAR-10

4.1.3 CelebA

Model był w stanie generować realistyczne twarze, a manipulacje w przestrzeni latentnej pozwoliły na zmianę atrybutów.

Rysunek 5: Porównanie oryginalnych i zrekonstruowanych obrazów CelebA. Górny rząd: oryginały; dolny rząd: rekonstrukcje.

Przebieg treningu modelu przedstawiono na Rysunku 6.



Rysunek 6: Wykres przebiegu treningu modelu CelebA.

Log treningu

```
# Trening VAE na zbiorze CelebA
vae_celeba = train_vae(
    input_shape=(64, 64, 3),
    latent_dim=256,
    x_train=x_train_celeba,
    x_val=x_val_celeba,
    epochs=100,
    batch_size=128,
    dataset_name='CelebA'
)

Epoch 1/100
165/165 [==============================] - 60s 232ms/step - kl_loss: 15503.8574
    - loss: 5168.7778 - reconstruction_loss: 5161.0259 - val_kl_loss: 585.3336
    - val_loss: 4353.9053 - val_reconstruction_loss: 4353.6123
...
Epoch 100/100
165/165 [==============================] - 26s 157ms/step - kl_loss: 1284.3070
    - loss: 2963.7439 - reconstruction_loss: 2963.1018 - val_kl_loss: 1278.6896
    - val_loss: 3002.5999 - val_reconstruction_loss: 3001.9604
```

Listing 12: Przebieg treningu VAE na zbiorze CelebA

4.2 Wpływ parametrów modelu

4.2.1 Wymiar przestrzeni latentnej

Zwiększenie wymiaru z poprawiało jakość rekonstrukcji, ale po przekroczeniu pewnej wartości efekty były malejące, a czas treningu znacząco wzrastał. Dla zbiorów CIFAR-10 i CelebA optymalne okazały się wymiary odpowiednio 128 i 256.

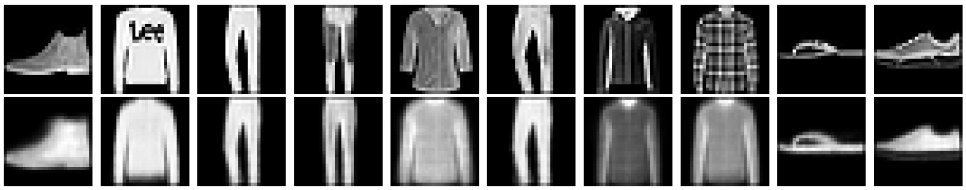
4.2.2 Waga dywergencji KL

Manipulacja wagą dywergencji KL miała istotny wpływ na balans między jakością rekonstrukcji a strukturą przestrzeni latentnej. Zbyt duża waga prowadziła do rozmytych rekonstrukcji, natomiast zbyt mała wpływała negatywnie na właściwości generacyjne modelu.

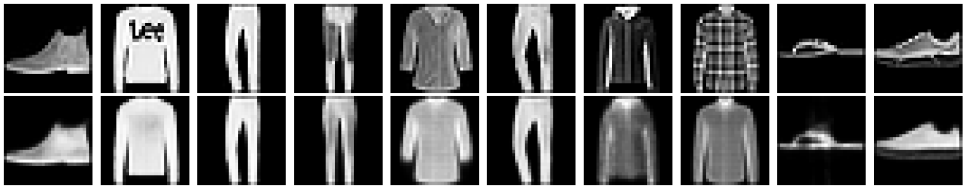
4.3 Interpolacje i wizualizacja przestrzeni latentnej

Interpolacje pokazały płynne przejścia między obiektami, potwierdzając ciągłość i strukturalność przestrzeni latentnej.

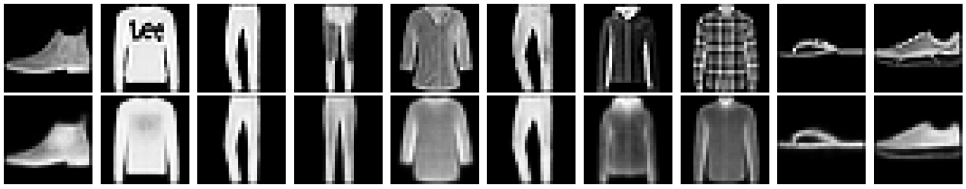
Wizualizacja przestrzeni latentnej ukazała, że model grupuje podobne klasy blisko siebie.



L=2

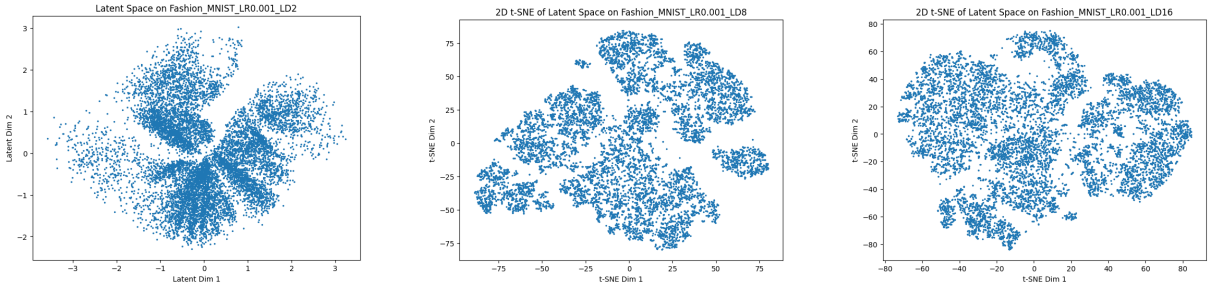


L=8



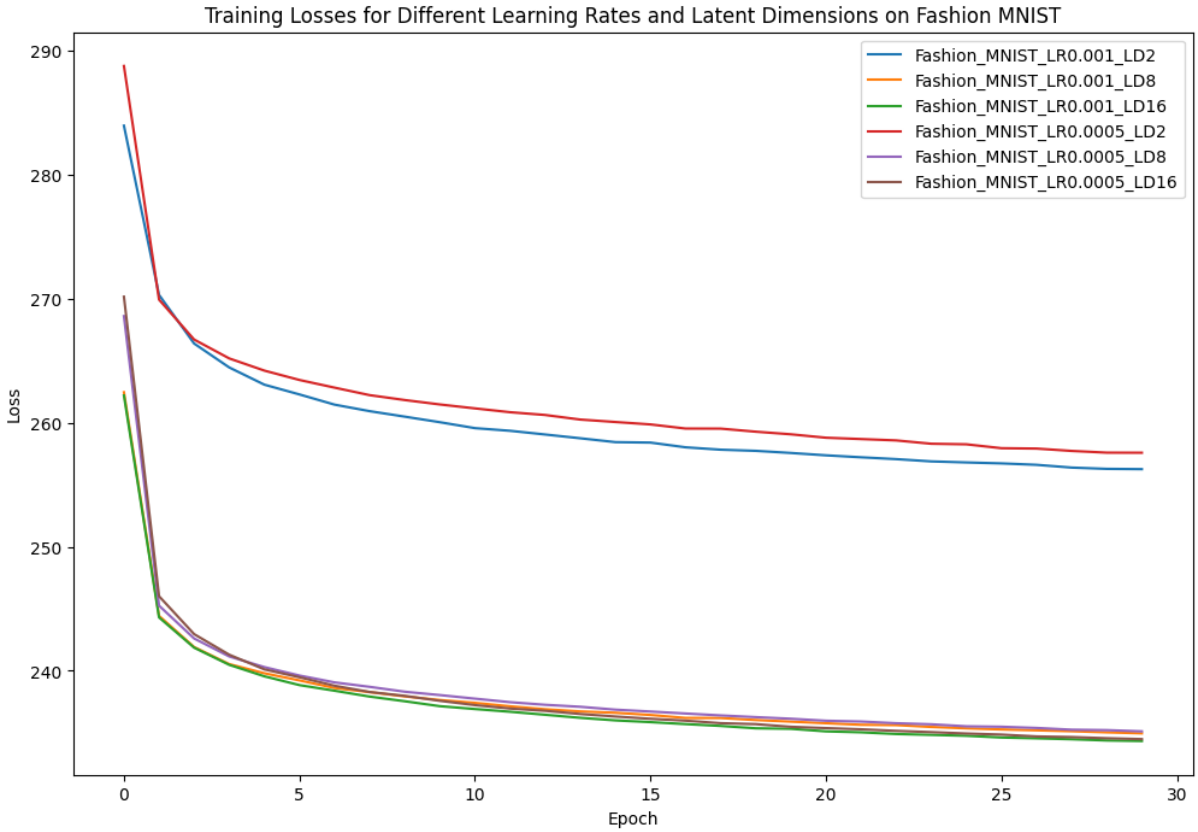
L=16

Rysunek 7: Interpolacje między obrazami w przestrzeni latentnej dla zbioru Fashion MNIST.



(a) L=2 (b) L=8 (c) L=16

Rysunek 8: Wizualizacja przestrzeni latentnej przy użyciu t-SNE dla zbioru Fashion MNIST.



Rysunek 9: Szkolenie Fashion MNIST z różnym tempem uczenia i parametrami przestrzeni latalnej

5 Wnioski

Przeprowadzone eksperymenty wykazały, że rozszerzenie modelu VAE poprzez zastosowanie warstw konwolucyjnych, pogłębienie sieci oraz dostosowanie parametrów, takich jak wymiar przestrzeni latentnej i waga dywergencji KL, pozwala na skuteczne modelowanie i generowanie bardziej złożonych danych. Rozwiązanie napotkanych problemów, takich jak niedopasowanie wymiarów czy błędy podczas treningu, było kluczowe dla osiągnięcia poprawnych wyników.

Uzyskane wyniki potwierdzają, że VAE jest potężnym narzędziem do modelowania skomplikowanych rozkładów danych, a odpowiednie dostosowanie architektury i funkcji straty umożliwia generowanie wysokiej jakości próbek.

6 Prace przyszłe

W przyszłości planuje się:

- Zastosowanie *Conditional VAE*, umożliwiających kontrolowane generowanie obrazów na podstawie zadanych atrybutów.
- Eksperymentowanie z innymi funkcjami straty, takimi jak *Perceptual Loss*, bazującymi na cechach wyższych rzędów z sieci neuronowych.
- Integrację modelu VAE z architekturami *Generative Adversarial Networks* (GAN), tworząc hybrydowe modele VAE-GAN w celu poprawy jakości generowanych obrazów.
- Badanie wpływu różnych rozkładów priory w przestrzeni latentnej oraz zastosowanie *Normalizing Flows* do modelowania bardziej skomplikowanych rozkładów latentnych.

Literatura

[1] Kingma, D. P., & Welling, M. (2014). *Auto-Encoding Variational Bayes*. Proceedings of the International Conference on Learning Representations (ICLR).

[2] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). *Gradient-Based Learning Applied to Document Recognition*. Proceedings of the IEEE, 86(11), 2278-2324.

[3] Xiao, H., Rasul, K., & Vollgraf, R. (2017). *Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv preprint arXiv:1708.07747.

[4] Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Technical report, University of Toronto.

[5] Liu, Z., Luo, P., Wang, X., & Tang, X. (2015). *Deep Learning Face Attributes in the Wild*. Proceedings of the IEEE International Conference on Computer Vision (ICCV).

[6] Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). *Image Quality Assessment: From Error Visibility to Structural Similarity*. IEEE Transactions on Image Processing, 13(4), 600-612.

[7] van der Maaten, L., & Hinton, G. (2008). *Visualizing Data using t-SNE*. Journal of Machine Learning Research, 9(Nov), 2579-2605.