

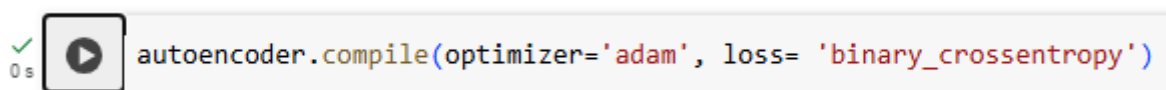
**Przedmiot: Uczenie Maszynowe**  
**Laboratorium: Modele Generatywne**  
**Kierunek: Informatyka – Data Science**  
**Autor: Bartłomiej Jamiolkowski**


## **Zadanie 1 - AutoEnkoder (AE)**

**Zadanie 1.1 - Dlaczego sigmoid jest odpowiednią funkcją aktywacji w ostatniej warstwie dekodera w tym przypadku?**

Sigmoid jest odpowiednią funkcją aktywacji w ostatniej warstwie dekodera w przypadku użycia zbioru MNIST, ponieważ zwraca wyniki w przedziale  $[0, 1]$ . Przedział ten pozwala modelować prawdopodobieństwo przynależności danego piksela do jasnego lub ciemnego koloru. Zapewnia to zgodność pomiędzy zakresem wartości wygenerowanego obrazu a znormalizowanymi danymi wejściowymi. Dodatkowo sigmoid stabilizuje gradienty podczas propagacji wstecznej, co ułatwia proces uczenia głębokiego modelu, minimalizując problemy eksplodujących lub zanikających gradientów.

**Zadanie 1.2. Skompiluj model. W tym celu najpierw zdefiniuj loss dla modelu. W przypadku autoenkodera jest to funkcja działająca na wejściach do enkodera oraz wyjściach z dekodera. Do wyboru są różne funkcje! Patrząc na reprezentację danych (wróć do funkcji definiującej preprocessing), wybierz odpowiednią. Uzasadnij swój wybór.**



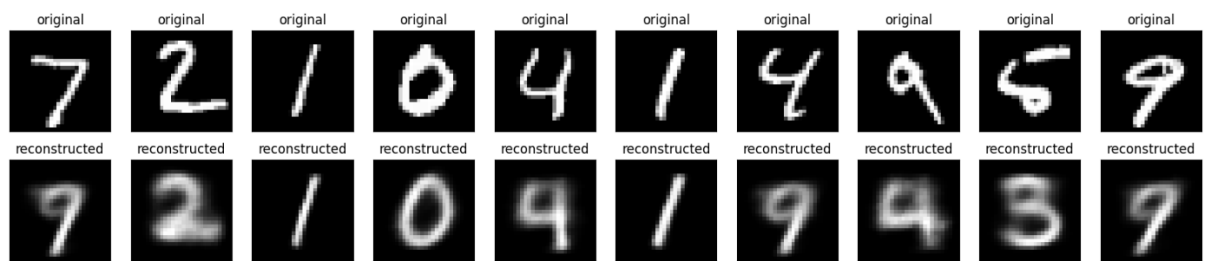
```
0s  autoencoder.compile(optimizer='adam', loss= 'binary_crossentropy')
```

**Rysunek 1 Zdefiniowana funkcja loss dla modelu AutoEnkoder**

Na podstawie obserwacji funkcji definiującej preprocessing, jako funkcję straty dla autoenkodera wybrałem Binary Cross Entropy (BCE). Moja decyzja umotywowana jest normalizacją wartości pikseli do przedziału  $[0, 1]$ , co czyni je podobnymi do wartości prawdopodobieństw. Na mój wybór miał również wpływ fakt, że w ostatniej warstwie autoenkodera jest używana funkcja sigmoid. Zastosowanie BCE (konkretnie jego logarytmu) odwraca działanie eksponenty w sigmoidzie. W ten sposób BCE poprawia propagację

gradientów, ponieważ logarytm niweluje efekt nasycania sigmoidy. Dzięki temu gradienty są bardziej stabilne i umożliwiają skuteczniejsze uczenie.

**Wyświetlmy przykładowe obrazy oraz ich rekonstrukcje:**



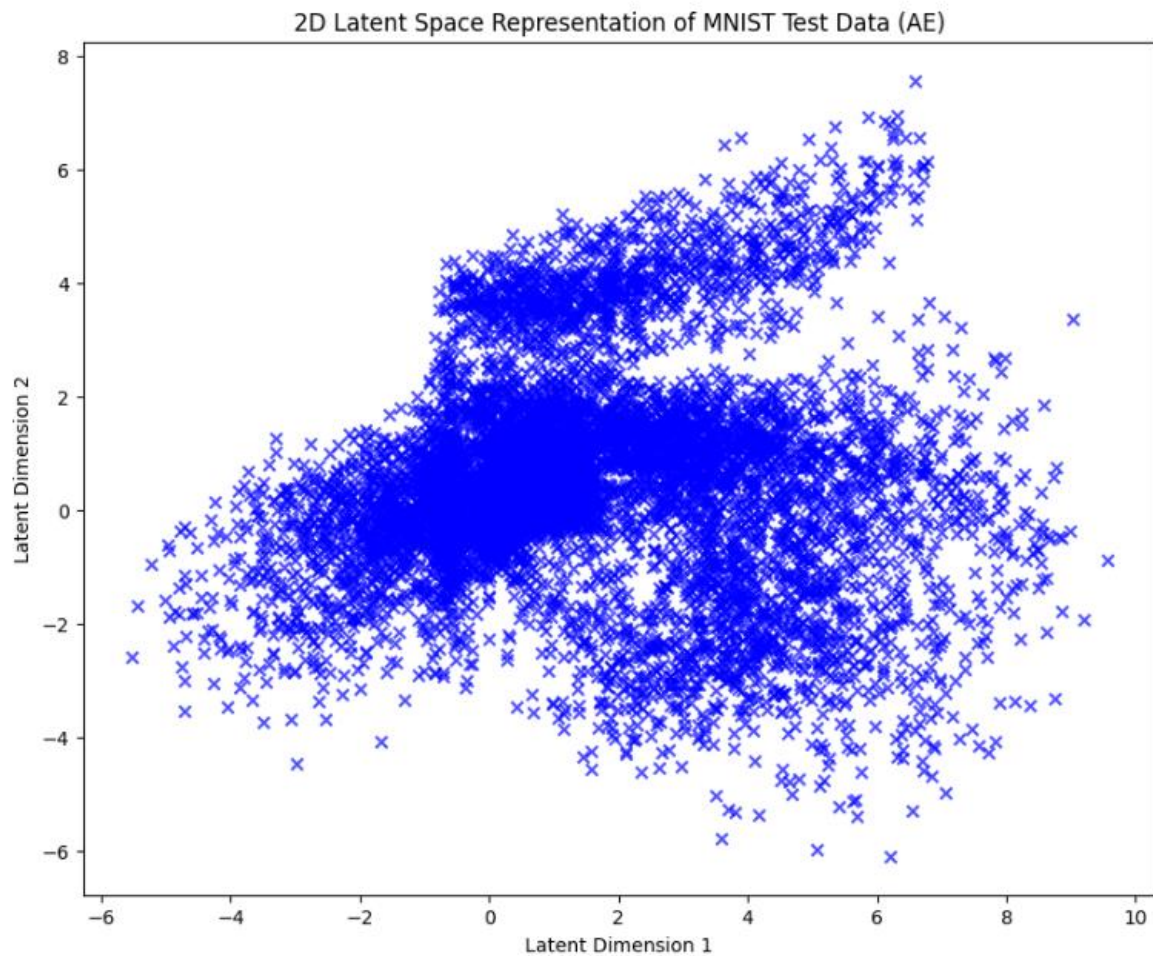
Rysunek 2 Przykładowe obrazy oraz ich rekonstrukcje

**Zaimplementuj funkcję do wizualizacji reprezentacji obrazów ze zbioru testowego w ukrytej przestrzeni 2-wymiarowej. Wyświetl wizualizację.**

```
✓ 0s ▶ def plot_latent_space(model, data):  
    latent_representations = model.encoder(data).numpy()  
  
    plt.figure(figsize = (10, 8))  
    plt.scatter(latent_representations[:, 0], latent_representations[:, 1], c = 'b', alpha = 0.7, marker = 'x')  
    plt.xlabel('Latent Dimension 1')  
    plt.ylabel('Latent Dimension 2')  
    plt.title('2D Latent Space Representation of MNIST Test Data')  
    plt.show()
```

Rysunek 3 Zaimplementowana funkcja do wizualizacji reprezentacji obrazów

```
plot_latent_space(autoencoder, x_test)
```



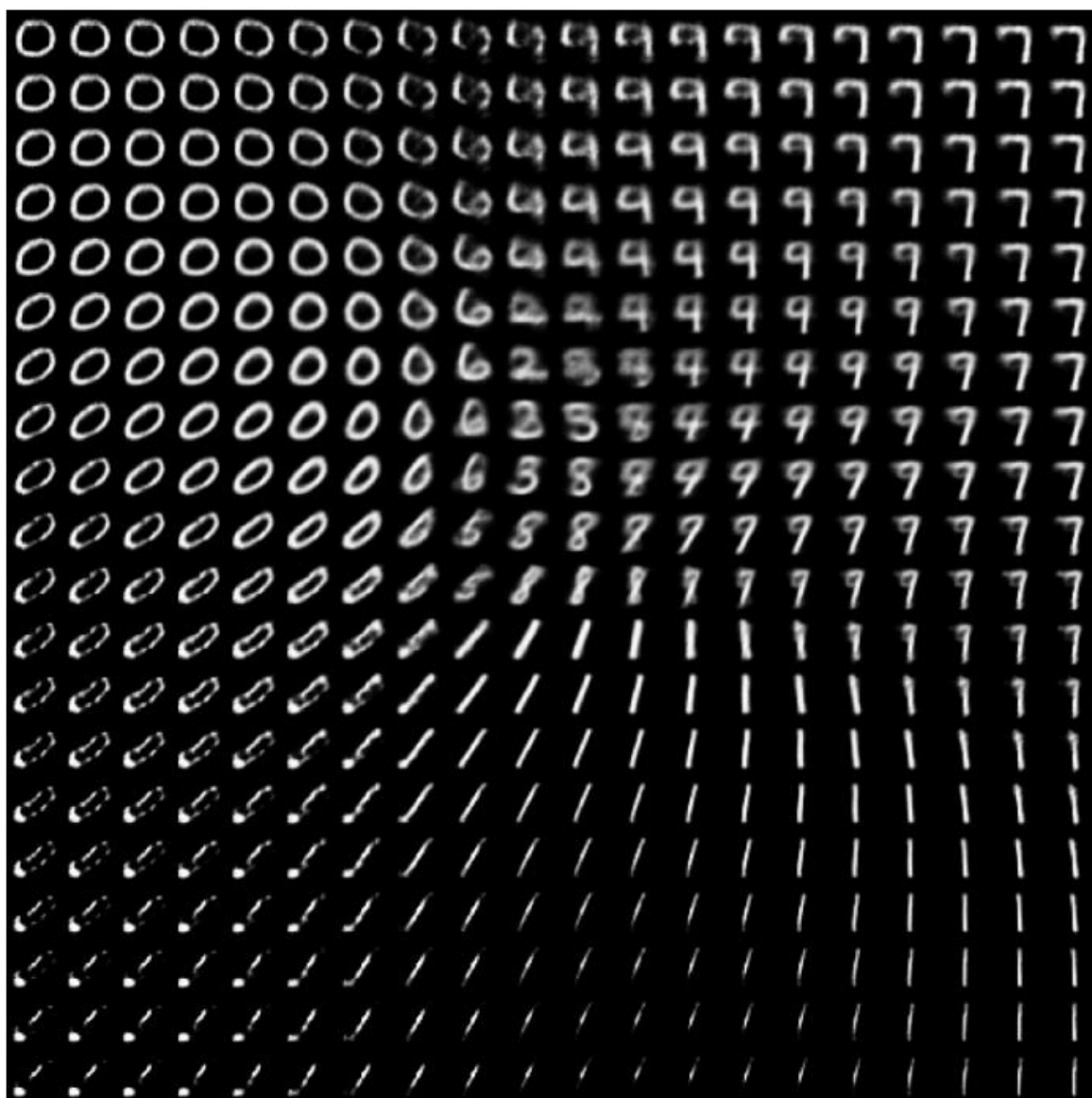
Rysunek 4 Zwizualizowana dwuwymiarowa przestrzeń latentna dla AE

Teraz wyświetlmy obrazy powstałe przez dekodowanie wartości z prostokątnej siatki. Dobierz granice siatki, analizując wyniki funkcji `plot_latent_space`.

```
def plot_latent_images(model, n, digit_size=28):  
    """Plots n x n digit images decoded from the latent space."""  
  
    grid_x = np.linspace(-6, 10, n)  
    grid_y = np.linspace(-6.5, 8, n)
```

Rysunek 5 Dobrane granice siatki w funkcji `plot_latent_images`

```
plot_latent_images(autoencoder, 20)
```



Rysunek 6 Obrazy powstałe przez dekodowane wartości z prostokątnej siatki dla AE

**Zadanie 1.3.** Wybierz ze zbioru testowego dwa obrazy z różnymi liczbami. Dobierz takie liczby, dla których spodziewasz się, że odkodowanie średniej z ich zenkodowanych reprezentacji będzie miało sens. Wybierz dwie takie pary. Dla każdej z par:

- Wyświetl wybrane liczby.
- Użyj enkodera do uzyskania 2-wymiarowych reprezentacji każdej liczby.
- Wylicz średnią z tych reprezentacji.
- Użyj dekodera na uzyskanej średniej.

- Wyświetl wynik.
- Skomentuj wynik - czy przypomina jakąś liczbę? Czy takiego wyniku się spodziewałeś?

```
def visualize_results_ae(first_number, second_number):
    fig, ax = plt.subplots(1, 3, figsize = (10, 6))
    ax[0].imshow(first_number, cmap = 'gray')
    ax[0].set_title('First number')
    ax[0].axis('off')
    ax[1].imshow(second_number, cmap = 'gray')
    ax[1].set_title('Second number')
    ax[1].axis('off')

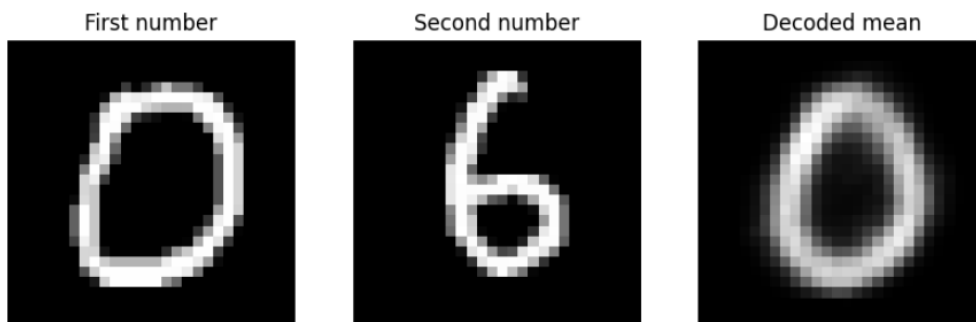
    latent_2d_first_number = autoencoder.encoder(tf.expand_dims(first_number, axis = 0)).numpy()
    latent_2d_second_number = autoencoder.encoder(tf.expand_dims(second_number, axis = 0)).numpy()
    latent_2d_mean = (latent_2d_first_number + latent_2d_second_number) / 2
    decoded_mean = autoencoder.decoder(latent_2d_mean).numpy().squeeze()

    ax[2].imshow(decoded_mean, cmap = 'gray')
    ax[2].set_title('Decoded mean')
    ax[2].axis('off')
    plt.show()
```

Rysunek 7 Funkcja wizualizująca rezultaty dla AE

```
number_zero = x_test[10]
number_six = x_test[21]

visualize_results_ae(number_zero, number_six)
```

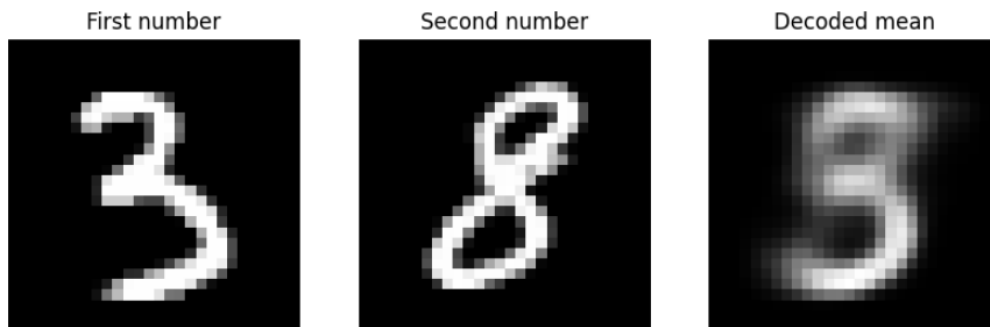


Rysunek 8 Wybrane liczby wraz z odkodowaną średnią z ich zenkodowanych reprezentacji dla AE

Uzyskany wynik przypomina liczbę zero. Mimo, że nie jest to idealna reprezentacja tej liczby (rozmyty obraz), to łączy ona w sobie kształty liczb zero (okrągły kształt) oraz sześć (okrąg podobny do zera z dodatkowym ogonem). Spodziewałem się takiego wyniku, ponieważ średnia w przestrzeni latentnej miesza cechy obu liczb, generując w ten sposób przejściowy kształt. Wskazuje to na zdolność autoenkodera do interpolacji w przestrzeni latentnej.

```
number_three = x_test[32]
number_eight = x_test[110]

visualize_results_ae(number_three, number_eight)
```



Rysunek 9 Wybrane liczby wraz z odkodowaną średnią z ich zenkodowanych reprezentacji dla AE

Uzyskany wynik łączy w sobie kształty liczb trzy oraz osiem. Spodziewałem się wyniku bardziej przypominającego liczbę trzy, ale na wizualizacji widać, że otrzymany wynik posiada również cechy liczby 8 tj. delikatne domknięcia pętli.

## Zadanie 2 - AutoEnkoder wariacyjny (VAE)

**Zadanie 2.1. Dlaczego powyższa implementacja CVAE nie stosuje żadnej aktywacji w ostatniej warstwie enkodera? Czy jakaś funkcja by się tutaj nadawała?**

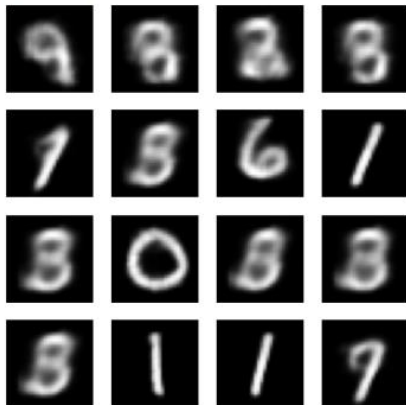
Powyższa implementacja CVAE nie stosuje żadnej aktywacji w ostatniej warstwie enkodera, ponieważ musi swobodnie modelować parametry rozkładu normalnego tj. średnią i logarytm z wariancji w pełnym zakresie liczb rzeczywistych. Średnia określa centrum rozkładu normalnego, dlatego może przyjmować dowolne wartości rzeczywiste. Logarytm wariancji może przyjmować dowolne wartości, w tym wartości ujemne, co pozwala na swobodne modelowanie rozkładu latentnego. W procesie reparametryzacji, zastosowanie funkcji wykładniczej gwarantuje dodatnią wariancję, ponieważ wykładnik z logarytmu daje wartość większą niż zero. To podejście gwarantuje dodatnią wariancję a co za tym idzie większą stabilność podczas uczenia. Ograniczenie obu parametrów funkcją aktywacji mogłoby zaburzyć modelowanie rozkładu latentnego.

W teorii nadawałaby się liniowa funkcja aktywacji, ponieważ nie zmienia ona wartości wejściowych i pozwala na pełną reprezentację liczb rzeczywistych. W praktyce jest ona już używana, ponieważ CVAE dla ostatniej warstwy enkodera nie stosuje żadnej funkcji aktywacji.



**Zainicjalizujemy model i przeprowadźmy trening:**

Epoch: 10, Val set ELBO: -158.04183959960938, time elapse for current epoch: 78.71325755119324



Rysunek 10 Wynik końcowy treningu modelu VAE

Narysujmy teraz, podobnie jak wcześniej dla AE, dwuwymiarową przestrzeń ukrytą. Zaimplementuj funkcję `plot_latent_space`, która zenkoduje zbiór danych, a następnie wyświetli każdy punkt wraz z odchyleniem standardowym.

```
def plot_latent_space(model, data):
    mean, logvar = model.encode(data)
    stddev = tf.exp(0.5 * logvar)
    mean = mean.numpy()
    stddev = stddev.numpy()

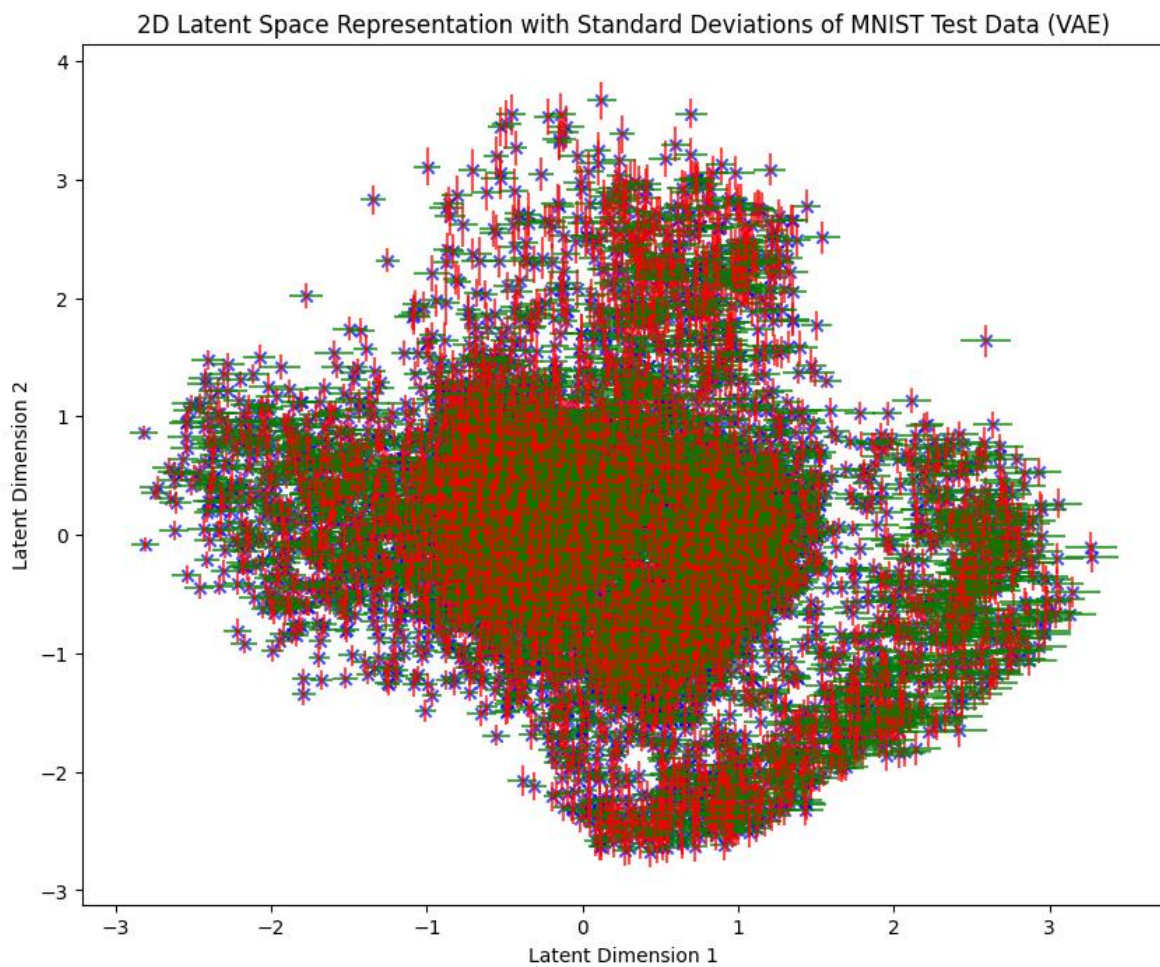
    plt.figure(figsize = (10, 8))
    plt.scatter(mean[:, 0], mean[:, 1], c = 'b', alpha = 0.7, marker = 'x')

    for i in range(len(mean)):
        plt.plot([mean[i, 0], mean[i, 0] + stddev[i, 0]], [mean[i, 1], mean[i, 1]], 'g-', alpha = 0.7)
        plt.plot([mean[i, 0], mean[i, 0] - stddev[i, 0]], [mean[i, 1], mean[i, 1]], 'g-', alpha = 0.7)
        plt.plot([mean[i, 0], mean[i, 0]], [mean[i, 1], mean[i, 1] + stddev[i, 1]], 'r-', alpha = 0.7)
        plt.plot([mean[i, 0], mean[i, 0]], [mean[i, 1], mean[i, 1] - stddev[i, 1]], 'r-', alpha = 0.7)

    plt.xlabel('Latent Dimension 1')
    plt.ylabel('Latent Dimension 2')
    plt.title('2D Latent Space Representation with Standard Deviations of MNIST Test Data (VAE)')
    plt.show()
```

Rysunek 11 Zaimplementowana funkcja `plot_latent_space`

```
plot_latent_space(model, x_test)
```



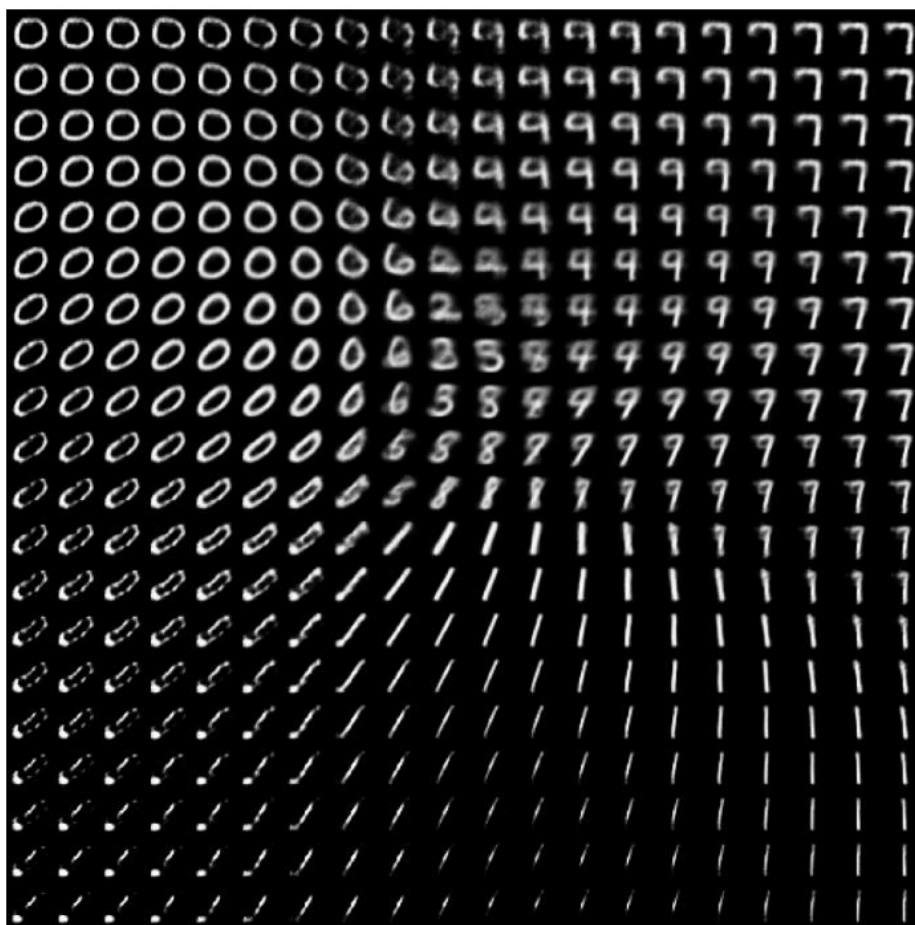
Rysunek 12 Zwizualizowana dwuwymiarowa przestrzeń latentna dla VAE

**Zadanie 2.2.** Skomentuj wynik uzyskany przy użyciu funkcji `plot_latent_images`. Zwróć uwagę na jakość/sensowność rysowanych liczb. Porównaj wykres do analogicznego wykresu dla modelu AE. Zamieść w raporcie wykresy.

Wynik uzyskany przy użyciu funkcji `plot_latent_images` dla modelu AE:



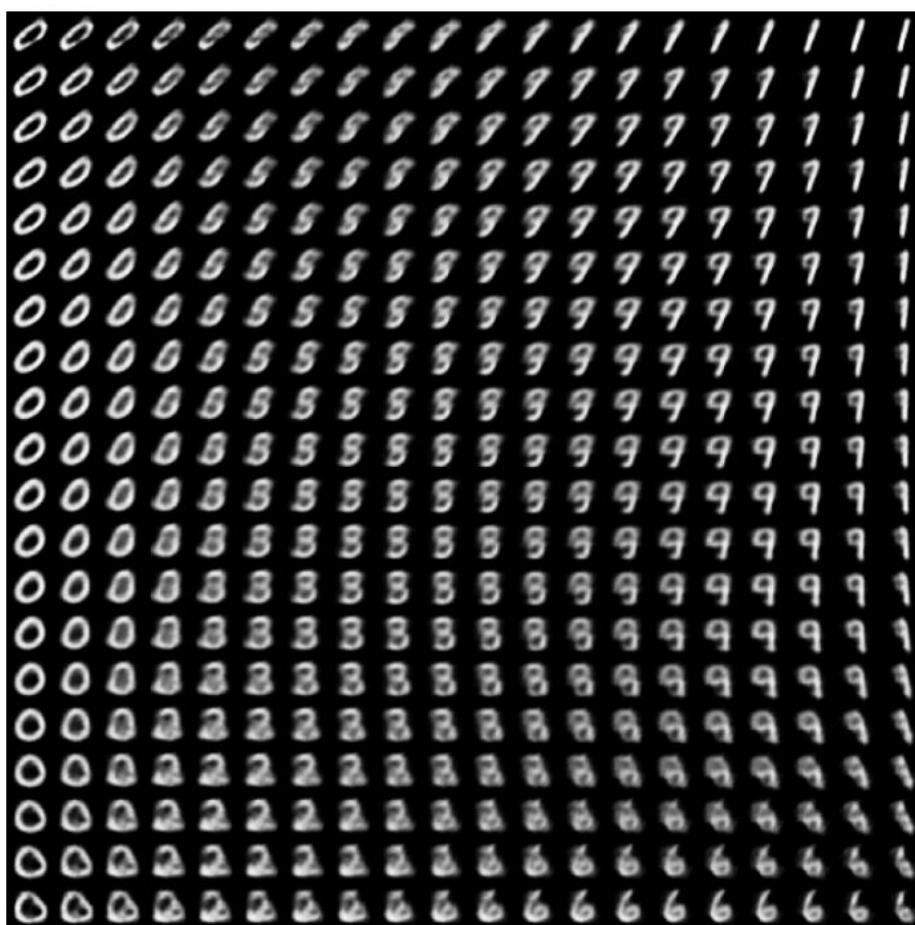
```
plot_latent_images(autoencoder, 20)
```



Rysunek 13 Obrazy powstałe przez dekodowane wartości z prostokątnej siatki dla AE

Wynik uzyskany przy użyciu funkcji `plot_latent_images` dla modelu VAE:

```
plot_latent_images(model, 20)
```



Rysunek 14 Obrazy powstałe przez dekodowane wartości z prostokątnej siatki dla VAE

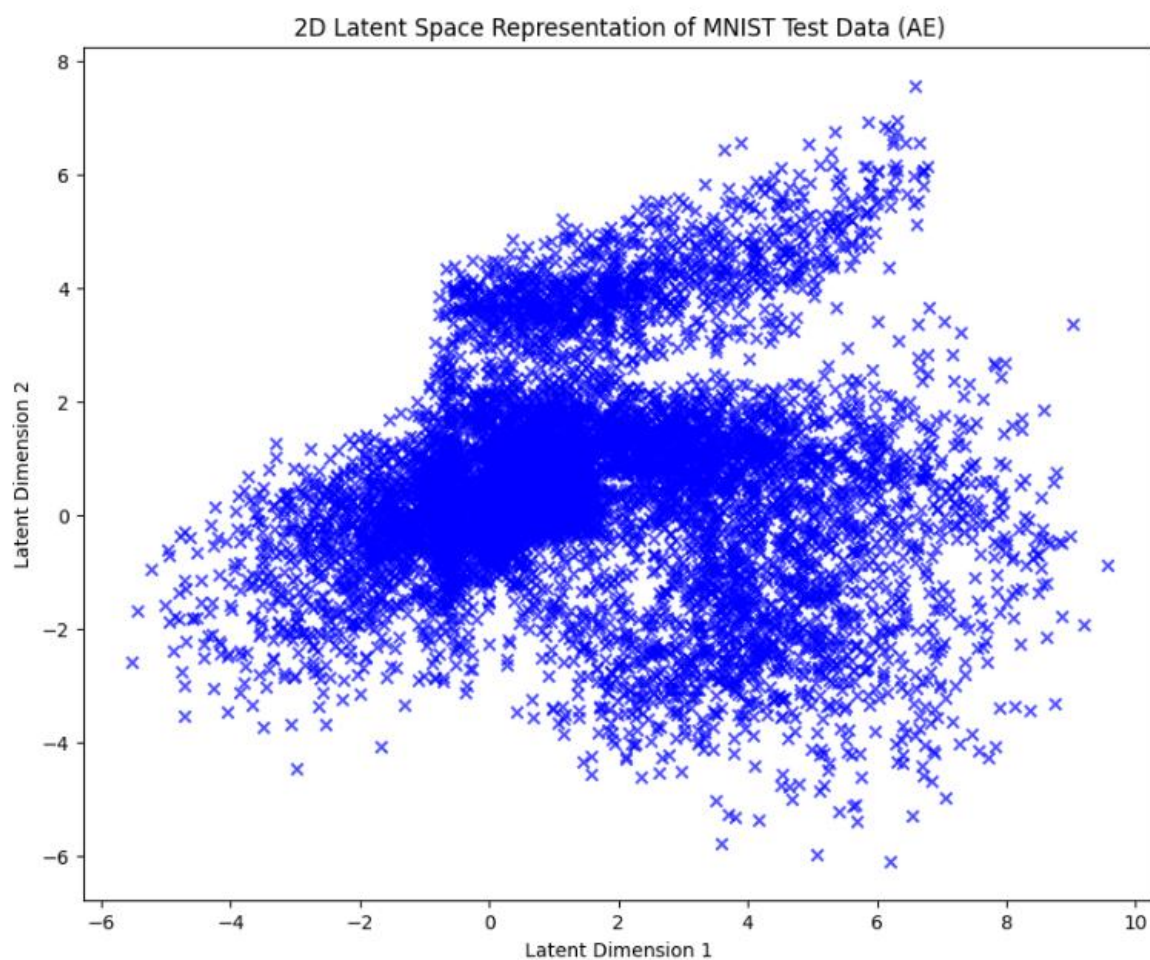
Wynik uzyskany przy użyciu funkcji `plot_latent_images` dla VAE przedstawia sensownie narysowane liczby. Choć jakość wygenerowanych obrazów nie jest idealna, to w większości przypadków obrazy są rozpoznawalne i można łatwo zidentyfikować reprezentowane liczby. Dzięki regularyzacji wprowadzanej przez VAE, obrazy są bardziej spójne i strukturalnie poprawne, mimo że nie są idealne pod względem szczegółowości.

W przeciwieństwie do wyników uzyskanych za pomocą modelu VAE, obrazy generowane przez AE są zdecydowanie gorszej jakości. Wiele z nich jest zdeformowanych lub całkowicie niezrozumiałych, a tylko część wygenerowanych liczb jest sensowna. AE nie korzysta z mechanizmu regularyzacji, co prowadzi do mniejszej spójności i dokładności w generowanych obrazach.

**Zadanie 2.3.** Porównaj wyniki funkcji `plot_latent_space` dla AE oraz VAE. Zwróć uwagę na "gęstość" punktów oraz zakres wartości. Zamieść w raporcie wykresy.

Wynik uzyskany przy użyciu funkcji `plot_latent_space` dla modelu AE:

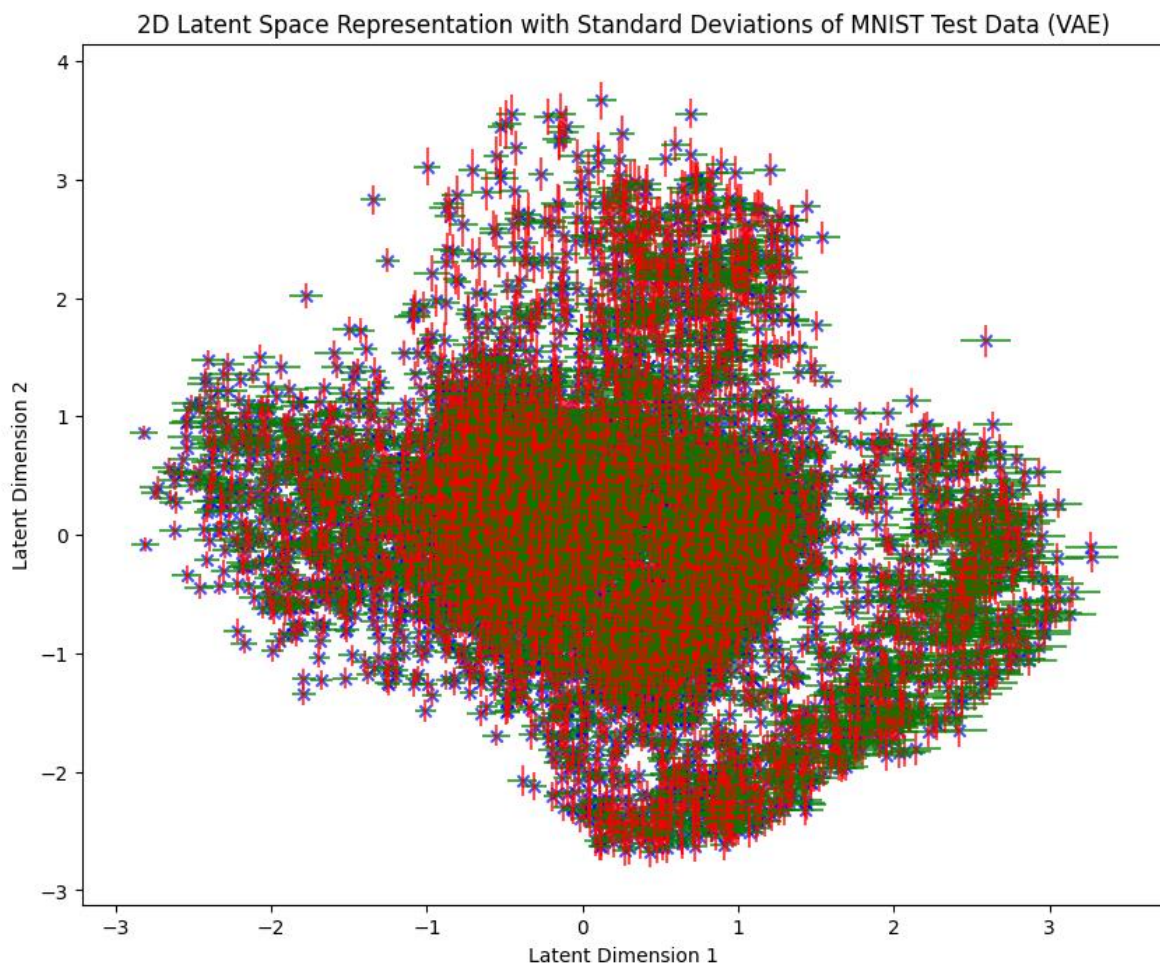
```
plot_latent_space(autoencoder, x_test)
```



Rysunek 15 Zwizualizowana dwuwymiarowa przestrzeń latentna dla AE

Wynik uzyskany przy użyciu funkcji `plot_latent_space` dla modelu VAE:

```
plot_latent_space(model, x_test)
```



Rysunek 16 Zwizualizowana dwuwymiarowa przestrzeń latentna dla VAE

Porównując wyniki funkcji `plot_latent_space` dla AE oraz VAE można zauważyć, że przestrzeń latentna VAE jest bardziej uporządkowana i spójna, co jest efektem wprowadzenia regularyzacji. Przeciwnieństwem jest wykres przestrzeni latentnej dla AE charakteryzujący się szerszym rozrzutem punktów i mniej skondensowaną strukturą.

**Zadanie 2.4.** Dla tych samych par obrazów, na których pracowałeś/eś w ostatnim zadaniu dot. AE, przygotuj reprezentacje ukryte z pomocą wytrenowanego VAE i odkoduj średnie z reprezentacji. Skomentuj wyniki, porównaj z wynikami z AE.

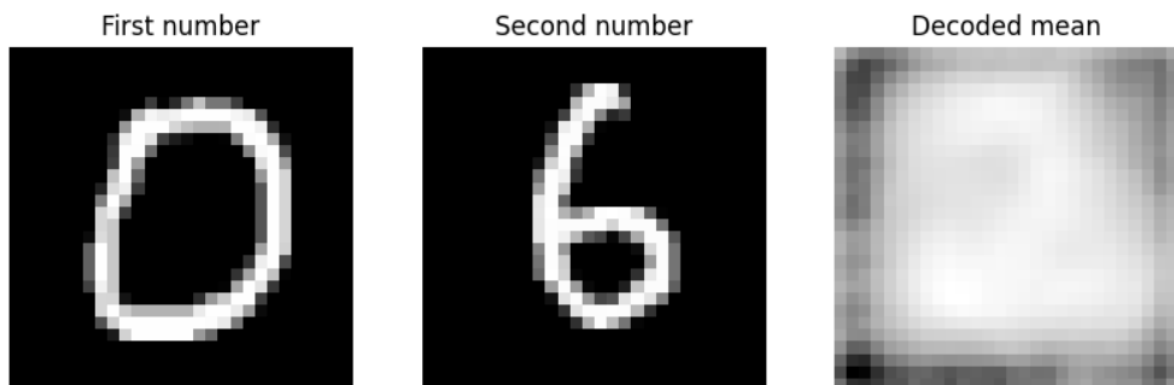
```
def visualize_results_vae(first_number, second_number):
    fig, ax = plt.subplots(1, 3, figsize = (10, 6))
    ax[0].imshow(first_number, cmap = 'gray')
    ax[0].set_title('First number')
    ax[0].axis('off')
    ax[1].imshow(second_number, cmap = 'gray')
    ax[1].set_title('Second number')
    ax[1].axis('off')

    mean_first_number, logvar_first_number = model.encode(tf.expand_dims(first_number, axis = 0))
    mean_second_number, logvar_second_number = model.encode(tf.expand_dims(second_number, axis = 0))
    mean_of_mean = (mean_first_number.numpy() + mean_second_number.numpy()) / 2
    mean_of_logvar = (logvar_first_number.numpy() + logvar_second_number.numpy()) / 2
    z = model.reparameterize(mean_of_mean, mean_of_logvar)
    decoded_mean = model.decode(z).numpy().squeeze()

    ax[2].imshow(decoded_mean, cmap = 'gray')
    ax[2].set_title('Decoded mean')
    ax[2].axis('off')
    plt.show()
```

Rysunek 17 Funkcja wizualizująca rezultaty dla VAE

```
visualize_results_vae(number_zero, number_six)
```

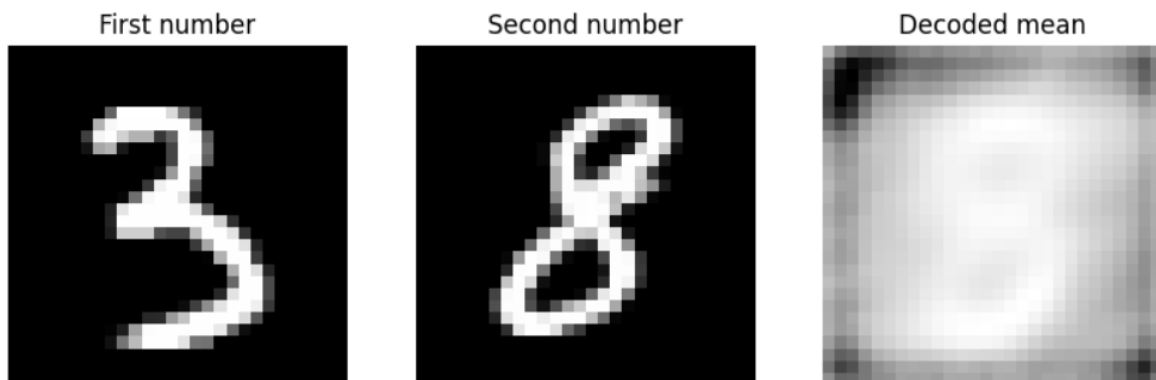


Rysunek 18 Wybrane liczby wraz z odkodowaną średnią z ich zenkodowanych reprezentacji dla VAE

Uzyskany wynik nie przypomina sensownej liczby. Porównując wynik dla VAE z wynikiem AE, nie ma wątpliwości, że ten pierwszy z wymienionych wyników jest gorszy.



```
visualize_results_vae(number_three, number_eight)
```



Rysunek 19 Wybrane liczby wraz z odkodowaną średnią z ich zenkodowanych reprezentacji dla VAE

Uzyskany wynik z użyciem VAE łączy w sobie kształty liczb trzy oraz osiem. Tak jak poprzedni rezultat jest on bardziej rozmyty w porównaniu z odpowiadającym mu rezultatem z AE.

**Zadanie 2.5.** Wróć do funkcji `compute_loss`. Człony `logpz` oraz `logqz_x` związane są z obliczaniem KL-divergence pomiędzy  $Q(z|X)$  oraz  $P(z)$ . Zakładamy, że oba te rozkłady są gaussowskie, stąd możemy wykorzystać wzór na KL-divergence dla dwóch rozkładów gaussowskich. Znajdź ten wzór oraz przepisz funkcję `compute_loss` z jego wykorzystaniem. Zamieść w raporcie przygotowaną formułę. Wytrenuj model ponownie, porównaj wyniki z poprzednią implementacją `compute_loss`.

$$\mathcal{D}[\mathcal{N}(\mu(X), \Sigma(X)) \| \mathcal{N}(0, I)] = \frac{1}{2} \left( \text{tr}(\Sigma(X)) + (\mu(X))^T (\mu(X)) - k - \log \det(\Sigma(X)) \right)$$

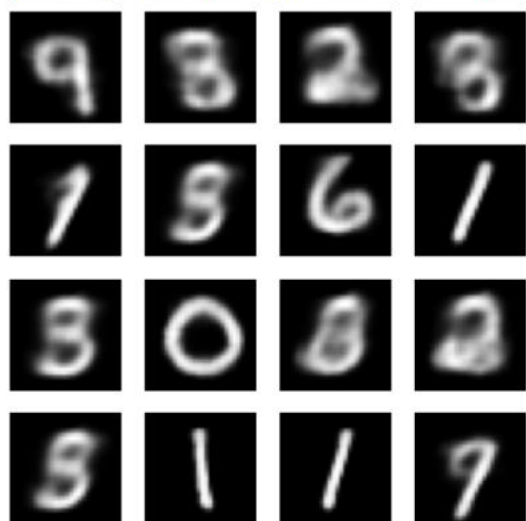
Rysunek 20 Wzór na KL-divergence

```
def compute_loss(model, x):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)
    kl_div = 0.5 * tf.reduce_sum(logvar + tf.square(mean) + tf.exp(logvar) - 1, axis=1)
    return -tf.reduce_mean(logpx_z + logpz - logqz_x + kl_div)
```

Rysunek 21 Funkcja `compute_loss` z wykorzystaniem KL-divergence



Epoch: 10, Val set ELBO: -160.7881622314453, time elapse for current epoch: 78.24462747573853



Rysunek 22 Wynik końcowy treningu z użyciem KL-divergence modelu VAE

Porównując wartości ELBO dla obu implementacji `compute_loss`, można zauważyć, że lepszym wynikiem dla VAE jest ten, gdzie w implementacji `compute_loss` użyto KL-divergence. Dzieje się tak, ponieważ `compute_loss` z KL-divergence uwzględnia zarówno błąd rekonstrukcji, jak i regularizację przestrzeni latentnej, co czyni model VAE bardziej odpowiednim do zadań generatywnych.

### Zadanie 3 - Conditioned VAE

Uzupełnij funkcję, która z każdego obrazu ze zbioru danych `x` tworzy 9 obrazów z cyfrą na każdej z 9 pozycji siatki 3x3, a także tworzy etykiety `y` w postaci wektora [cyfra-one-hot, pozycja\_x, pozycja\_y]. Dla każdej pary z oryginalnego zbioru danych (obraz, etykieta) wylosuj `num_imgs` par, które znajdą się w docelowym zbiorze danych (nie zapisujemy wszystkich 9 możliwości ze względu na ograniczenia RAM).

```
def conditioned_mnist(x, y, num_imgs=2):
    x_res = np.empty(shape=(x.shape[0] * num_imgs, 3 * x.shape[1], 3 * x.shape[2]), dtype='float32') # pusta macierz z wynikami - obrazy x
    y_res = np.empty(shape=(y.shape[0] * num_imgs, 2 + y.shape[1]), dtype='float32') # pusta macierz z wynikami - wektor y: etykieta (10 liczb),
    empty_res = np.zeros(shape=(3 * x.shape[1], 3 * x.shape[2])) # obraz wynikowy w docelowym rozmiarze, wypełniony zerami

    for el, (arr, label) in enumerate(zip(x, y)):
        to_sample_x = np.empty((9, x.shape[1]*3, x.shape[2]*3), dtype='float32') # macierz przechowująca 9 wersji obrazu
        to_sample_y = np.empty((9, 12), dtype='float32') # macierz przechowująca 9 wersji etykiet
        for i in range(3):
            for j in range(3):
                curr_x = empty_res.copy()
                curr_x[i*x.shape[1]: (i+1)*x.shape[1], j*x.shape[2]: (j+1)*x.shape[2]] = arr.reshape((x.shape[1], x.shape[2]))
                curr_y = [*label, i/2, j/2] # normalizacja
                to_sample_x[3*i+j] = curr_x
                to_sample_y[3*i+j] = curr_y
            idxs = np.random.choice(9, num_imgs, replace = False) # wylosuj num_imgs indeksów z zakresu [0; 8] jako wektor numpy
            x_res[el*num_imgs: (el+1)*num_imgs] = to_sample_x[idxs]
            y_res[el*num_imgs: (el+1)*num_imgs] = to_sample_y[idxs]
        x_res = x_res.reshape((-1, x.shape[1]*3, x.shape[2]*3, 1))
    return x_res, y_res
```

Rysunek 23 Uzupełniona funkcja conditioned\_mnist

**Uzupełnij klasę Cond\_CVAE na podstawie klasy CVAE. W tym celu:**

**Uzupełnij funkcję prepare\_encoder. Będziemy mieć dwa wejścia do modelu - jedno na obraz, jedno na wektor cech [etykieta, pos\_x, pos\_y]. Przeprocesuj obraz z pomocą warstw konwolucyjnych (możesz wykorzystać implementację z CVAE). Możesz też przygotować kilka warstw, które zajmą się wektorem cech. Użyj warstwy konkatenującej wyniki z przetwarzania obrazu i wektora cech. Za tą warstwą znajdzie się warstwa gęsta, wyliczająca średnią i logvar.**

**Uzupełnij funkcję prepare\_decoder. Tu również mamy do czynienia z dwoma wejściami - jedno przyjmuje szum, drugie wektor cech. Połącz oba wejścia i przygotuj dekodery. Możesz skorzystać z implementacji CVAE, ale będą potrzebne zmiany związane z innym rozmiarem obrazów.**

**Pozostałe funkcje są już zaimplementowane. Przyjrzyj się im. Co się zmieniło względem implementacji CVAE?**

```

def prepare_encoder(self):
    input_img = tf.keras.layers.Input(shape=(42, 42, 1))
    input_cond = tf.keras.layers.Input(shape=(12, ))

    x = tf.keras.layers.Conv2D(filters=32, kernel_size=3, strides=(2, 2), activation='relu', padding='same')(input_img)
    x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=(2, 2), activation='relu', padding='same')(x)
    x = tf.keras.layers.Flatten()(x)

    x = tf.keras.layers.Concatenate()([x, input_cond])
    # No activation
    x = tf.keras.layers.Dense(latent_dim + latent_dim)(x)
    return tf.keras.Model([input_img, input_cond], [x])

def prepare_decoder(self):
    input_latent = tf.keras.layers.Input(shape=(latent_dim,))
    input_cond = tf.keras.layers.Input(shape=(12, ))
    inputs = tf.keras.layers.Concatenate()([input_latent, input_cond])

    x = tf.keras.layers.Dense(7 * 7 * 32, activation=tf.nn.relu)(inputs)
    x = tf.keras.layers.Reshape(target_shape=(7, 7, 32))(x)

    x = tf.keras.layers.Conv2DTranspose(
        filters=64, kernel_size=3, strides=2, padding='same', activation='relu')(x)
    x = tf.keras.layers.Conv2DTranspose(
        filters=32, kernel_size=3, strides=3, padding='same', activation='relu')(x)
    x = tf.keras.layers.Conv2DTranspose(
        filters=1, kernel_size=3, strides=1, padding='same')(x)
    return tf.keras.Model([input_latent, input_cond], [x])

```

Rysunek 24 Uzupełnione funkcje prepare\_encoder oraz prepare\_decoder w klasie Cond\_CVAE

Porównując implementację Cond\_CVAE z implementacją CVAE, można zauważyć że do kodu Cond\_CVAE dodano wektor warunkowy, który jest uwzględniany w obu częściach modelu: enkoderze i dekoderze. Dzięki temu Cond\_CVAE stał się modelem warunkowym. Oprócz tego w Cond\_CVAE pojawił się dodatkowy krok łączenia warunkowej zmiennej z przestrzenią latentną.

### Uzupełnij funkcję kosztu:

```

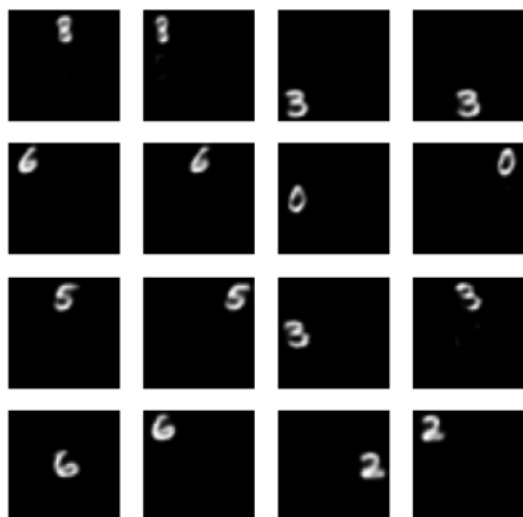
def compute_loss(model, x):
    data, cond = x
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logits = model.decode([z, cond])
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logits, labels=data)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)
    return -tf.reduce_mean(logpx_z + logpz - logqz_x)

```

Rysunek 25 Uzupełniona funkcja kosztu

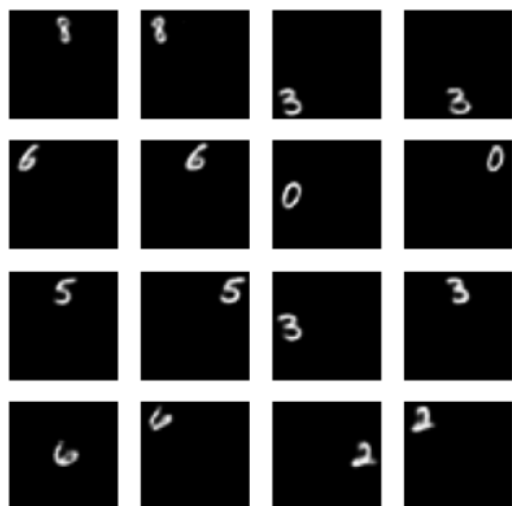
**Zadanie 3.1.** Sprawdź jakość modelu dla 3 różnych wartości `latent_dim` (trzeba dla każdej z nich osobno wytrenować model). Niech będą od siebie znacząco różne, np. 2, 25, 100. Przy większym `latent_dim` może być potrzebnych więcej epok.

Epoch: 10, Val set ELBO: -40.50996017456055, time elapse for current epoch: 15.58200716972351



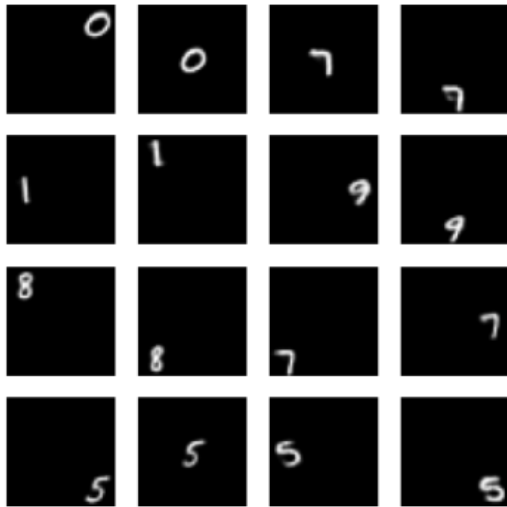
Rysunek 26 Wynik końcowy treningu z `latent_dim = 2` oraz epoki = 10 dla modelu `Cond_CVAE`

Epoch: 50, Val set ELBO: -37.065189361572266, time elapse for current epoch: 17.363019704818726



Rysunek 27 Wynik końcowy treningu z `latent_dim = 20` oraz epoki = 50 dla modelu `Cond_CVAE`

Epoch: 150, Val set ELBO: -36.51435089111328, time elapse for current epoch: 17.80129098892212



Rysunek 28 Wynik końcowy treningu z latent\_dim = 100 oraz epoki = 150 dla modelu Cond\_CVAE

Najlepszy wynik uzyskał ostatniego model dla parametrów latent\_dim = 100 oraz epoki = 150. Ten model był trenowany w innym czasie ze względu na przekroczenie limitu na google colab.

**Zadanie 3.2. Wykonaj dla najlepszego modelu z punktu 3.1.:**

**Wybierz przykład ze zbioru testowego (obraz + etykieta).**

**Przepuść próbkę przez enkoder, uzyskaj reprezentację z.**

**Dla każdego z 9 możliwych wektorów [poprawna\_etykieta, pos\_x, pos\_y] przepuść przez dekodery reprezentację z wraz z informacją o etykiecie i położeniu. Wyświetl uzyskany obraz. Skomentuj wyniki - czy za każdym razem uzyskano oczekiwaną liczbę w oczekiwanym miejscu? Jeśli nie, to co może być przyczyną?**

```
[92] assert batch_size >= 1
for test_batch in test_dataset_with_cond.take(1):
    test_sample_data = test_batch[0][0:1, :, :, :]
    test_sample_cond = test_batch[1][0:1, :]
    test_sample = [test_sample_data, test_sample_cond]

[93] data, cond = test_sample
mean, logvar = model.encode(test_sample)
z = model.reparameterize(mean, logvar)

vectors = []

for x_pos in [0, 0.5, 1]:
    for y_pos in [0, 0.5, 1]:
        label = cond[0, :10]
        new_cond = tf.concat([label, [x_pos, y_pos]], axis=0)
        vectors.append(new_cond.numpy())

generated_images = []

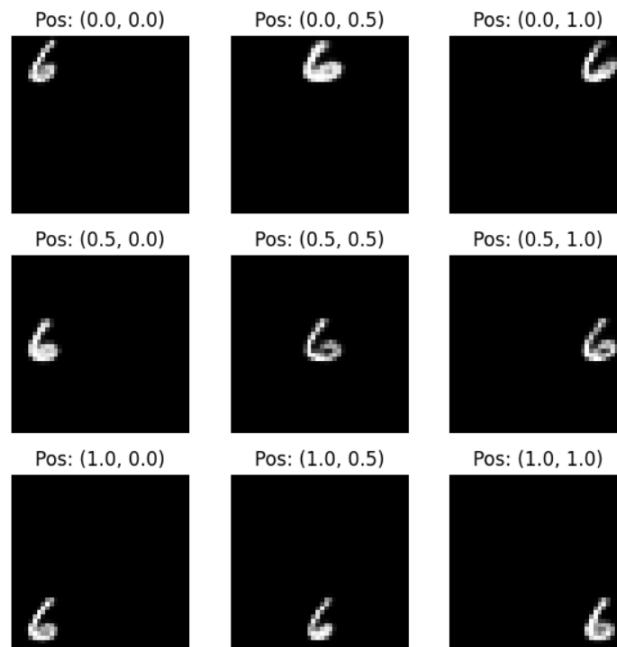
for vector in vectors:
    cond_tensor = tf.convert_to_tensor(vector.reshape(1, -1), dtype=tf.float32)
    generated_image = model.sample(cond_tensor, z)
    generated_images.append(generated_image.numpy().squeeze())

fig, axes = plt.subplots(3, 3, figsize=(6, 6))

for i, ax in enumerate(axes.flat):
    ax.imshow(generated_images[i], cmap='gray')
    ax.axis('off')
    ax.set_title(f'Pos: ({vectors[i][10]}, {vectors[i][11]})')

plt.tight_layout()
plt.show()
```

Rysunek 29 Kod stworzony na potrzeby zadania 3.2



Rysunek 30 Obrazy uzyskane po zdekodowaniu reprezentacji z wraz z informacją o etykiecie i położeniu

Otrzymane obrazy pokazują, że za każdym razem otrzymano liczbę w oczekiwanym miejscu, co oznacza, że model GAN dobrze nauczył się warunkowego generowania.



**Zadanie 3.3.** Powtórz zadanie 3.2, ale tym razem jako reprezentację z wykorzystaj wartości wylosowane z rozkładu normalnego oraz wybierz dowolną etykietę. Skomentuj wyniki - czy za każdym razem uzyskano oczekiwaną liczbę w oczekiwanym miejscu?

```
data, cond = next(iter(test_dataset_with_cond))
chosen_label = cond[0, :10]
vectors = []

for x_pos in [0, 0.5, 1]:
    for y_pos in [0, 0.5, 1]:
        new_cond = tf.concat([chosen_label, [x_pos, y_pos]], axis = 0)
        vectors.append(new_cond.numpy())

generated_images = []

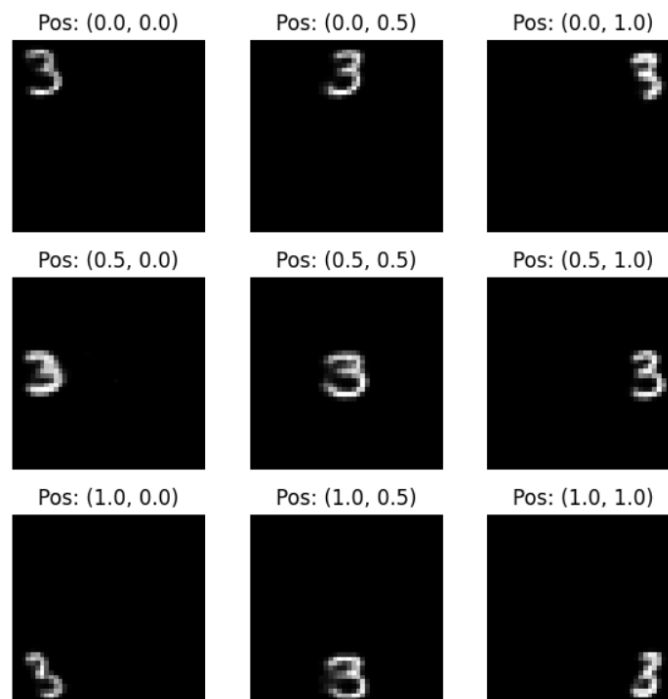
for vector in vectors:
    z = tf.random.normal(shape=(1, model.latent_dim))
    cond_tensor = tf.convert_to_tensor(vector.reshape(1, -1), dtype = tf.float32)
    generated_image = model.sample(cond_tensor, z)
    generated_images.append(generated_image.numpy().squeeze())

fig, axes = plt.subplots(3, 3, figsize=(6, 6))

for i, ax in enumerate(axes.flat):
    ax.imshow(generated_images[i], cmap = 'gray')
    ax.axis('off')
    ax.set_title(f'Pos: ({vectors[i][10]}, {vectors[i][11]})')

plt.tight_layout()
plt.show()
```

Rysunek 31 Kod stworzony na potrzeby zadania 3.3



Rysunek 32 Obrazy uzyskane po zdekodowaniu reprezentacji z wraz z informacją o etykiecie i położeniu

Otrzymane obrazy pokazują, że za każdym razem otrzymano liczbę w oczekiwanym miejscu, co oznacza, że model GAN dobrze nauczył się warunkowego generowania.

## Zadanie 4 - Conditioned GAN

Uzupełnij definicję generatora:

```
[27] def prepare_generator(latent_dim, cond_dim):
    input_img = tf.keras.layers.Input(shape=(latent_dim,))
    input_cond = tf.keras.layers.Input(shape=(cond_dim,))
    inputs = tf.keras.layers.Concatenate(axis=1)(input_img, input_cond)

    x1 = tf.keras.layers.Dense(7*7*256, use_bias=False)(inputs)
    x1 = tf.keras.layers.BatchNormalization()(x1)
    x1 = tf.keras.layers.LeakyReLU()(x1)
    x1 = tf.keras.layers.Reshape((7, 7, 256))(x1)

    x1 = tf.keras.layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False)(x1)
    x1 = tf.keras.layers.BatchNormalization()(x1)
    x1 = tf.keras.layers.LeakyReLU()(x1)

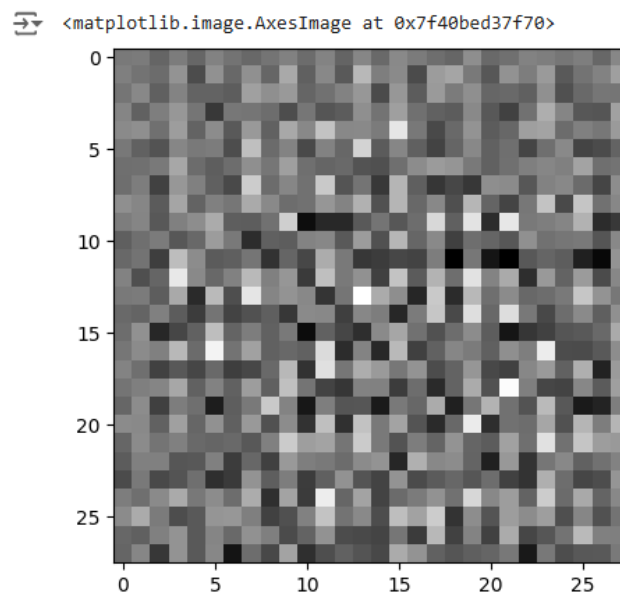
    x1 = tf.keras.layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False)(x1)
    x1 = tf.keras.layers.BatchNormalization()(x1)
    x1 = tf.keras.layers.LeakyReLU()(x1)

    x1 = tf.keras.layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh')(x1)

    return tf.keras.Model([input_img, input_cond], x1, name='generator')
```

Rysunek 33 Uzupełniona definicja generatora

Wyświetlmy obraz wygenerowany przez niewytrenowany generator przy podaniu na wejściu etykiety "1":



Rysunek 34 Obraz wygenerowany przez niewytrenowany generator

Uzupełnij definicję dyskryminatora:

```
✓ [30] def prepare_discriminator(img_shape, cond_dim):  
0s  
    input_img = tf.keras.layers.Input(shape=img_shape)  
    x = tf.keras.layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same')(input_img)  
    x = tf.keras.layers.LeakyReLU()(x)  
    x = tf.keras.layers.Dropout(0.3)(x)  
  
    x = tf.keras.layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same')(x)  
    x = tf.keras.layers.LeakyReLU()(x)  
    x = tf.keras.layers.Dropout(0.3)(x)  
  
    x = tf.keras.layers.Flatten()(x)  
  
    input_cond = tf.keras.layers.Input(shape=(cond_dim,))  
    x = tf.keras.layers.Concatenate(axis=1)([x, input_cond])  
  
    x = tf.keras.layers.Dense(256)(x)  
    x = tf.keras.layers.LeakyReLU()(x)  
    x = tf.keras.layers.Dropout(0.3)(x)  
  
    x = tf.keras.layers.Dense(128)(x)  
    x = tf.keras.layers.LeakyReLU()(x)  
    x = tf.keras.layers.Dropout(0.3)(x)  
    x = tf.keras.layers.Dense(1)(x)  
  
    return tf.keras.Model([input_img, input_cond], x, name='discriminator')
```

Rysunek 35 Uzupełniona definicja dyskryminatora

Zobaczmy predykcję niewytrenowanego dyskryminatora:

```
⇒ tf.Tensor([[0.00234074]], shape=(1, 1), dtype=float32)
```

Rysunek 36 Predykcja niewytrenowanego dyskryminatora

Uzupełnij funkcję kosztu:

```
✓ [34] def discriminator_loss(real_output, fake_output):  
0s  
    # real_output, fake_output - predykcje dyskryminatora  
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)  
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)  
    total_loss = real_loss + fake_loss  
    return total_loss
```

Rysunek 37 Uzupełniona funkcja kosztu

## Inicjalizacje

Zwiększyłem liczbę epok do 200, by poprawić jakość wyników

```
[37] EPOCHS = 200
      noise_dim = 100
      num_examples_to_generate = 16

      seed = tf.random.normal([num_examples_to_generate, noise_dim])
      seed_cond = tf.one_hot(tf.random.uniform([num_examples_to_generate], minval=0, maxval=10, dtype=tf.int32), depth=10)
```

Rysunek 38 Ustawione parametry treningu modelu GAN

## Uzupełnij funkcję train\_step:

```
[38] @tf.function
      def train_step(data):
          images, cond, noise_cond = data
          batch_size = tf.shape(images)[0]

          noise = tf.random.normal([batch_size, noise_dim])

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

              real_output = discriminator([images, cond], training=True)
              fake_output = discriminator([generated_images, cond], training=True)

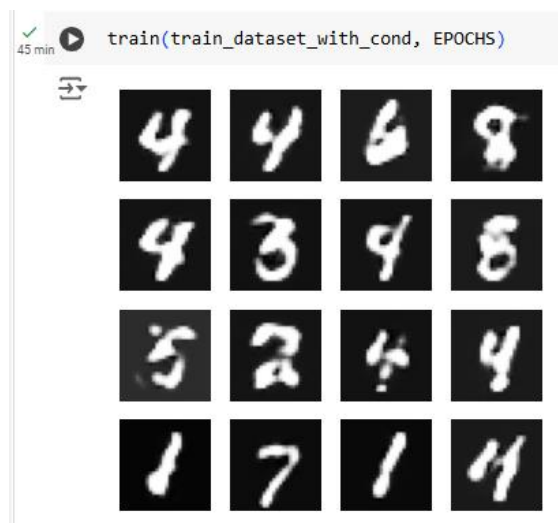
              gen_loss = generator_loss(fake_output)
              disc_loss = discriminator_loss(real_output, fake_output)

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

              generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
              discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

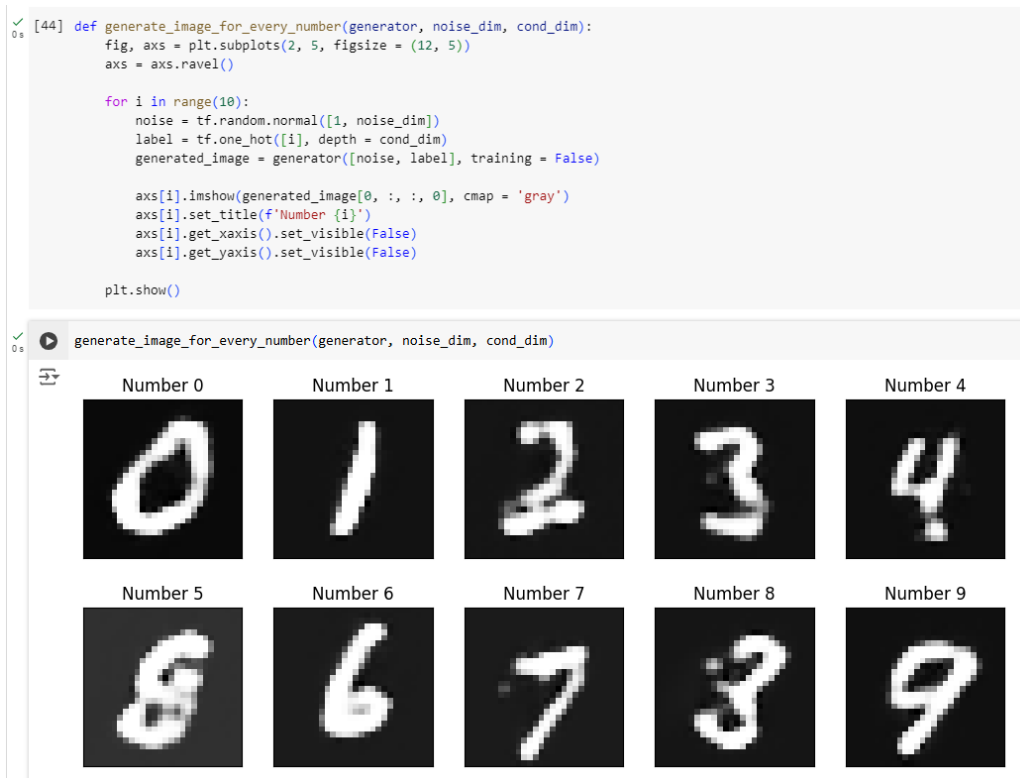
Rysunek 39 Uzupełniona funkcja train\_step

## Czas na trening!



Rysunek 40 Końcowy wynik treningu modelu GAN dla 200 epok

**Zadanie 4.1.** Wygeneruj po jednym obrazie z każdą liczbą z pomocą generatora. Oceń jakość wyników. Jeśli jakość modelu pozostawia wiele do życzenia, spróbuj go poprawić, np. zwiększając liczbę epok bądź zmieniając definicję generatora/dyskryminatora.



Rysunek 41 Wygenerowane obrazy z każdą liczbą z pomocą generatora

Wyniki uzyskane dla treningu z parametrem epok ustawionym na 200 są na tyle dobre, że pozwalają mi odczytać obrazy z każdą liczbą.