

Marcin Wardyński

wtorek, 9:45

1. AutoEnkoder (AE)

Zadanie 1.1 - Dlaczego sigmoid jest odpowiednią funkcją aktywacji w ostatniej warstwie dekodera w tym przypadku? (0.25pkt)

Ponieważ chcemy generować przypadki wyjściowe w sposób podobny do danych wejściowych. Na wejściu autoenkodera podawaliśmy obrazki w skali szarości, gdzie intensywność bieli wyrażona jest w sposób znormalizowany w przedziale [0, 1], chcąc otrzymać taki sam przedział wartości na wyjściu aplikujemy sigmoidę jako funkcję aktywacji ostatniej warstwy, która sprowadza generowane wartości do pożądanego przedziału [0, 1].

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. (0.25 pkt)

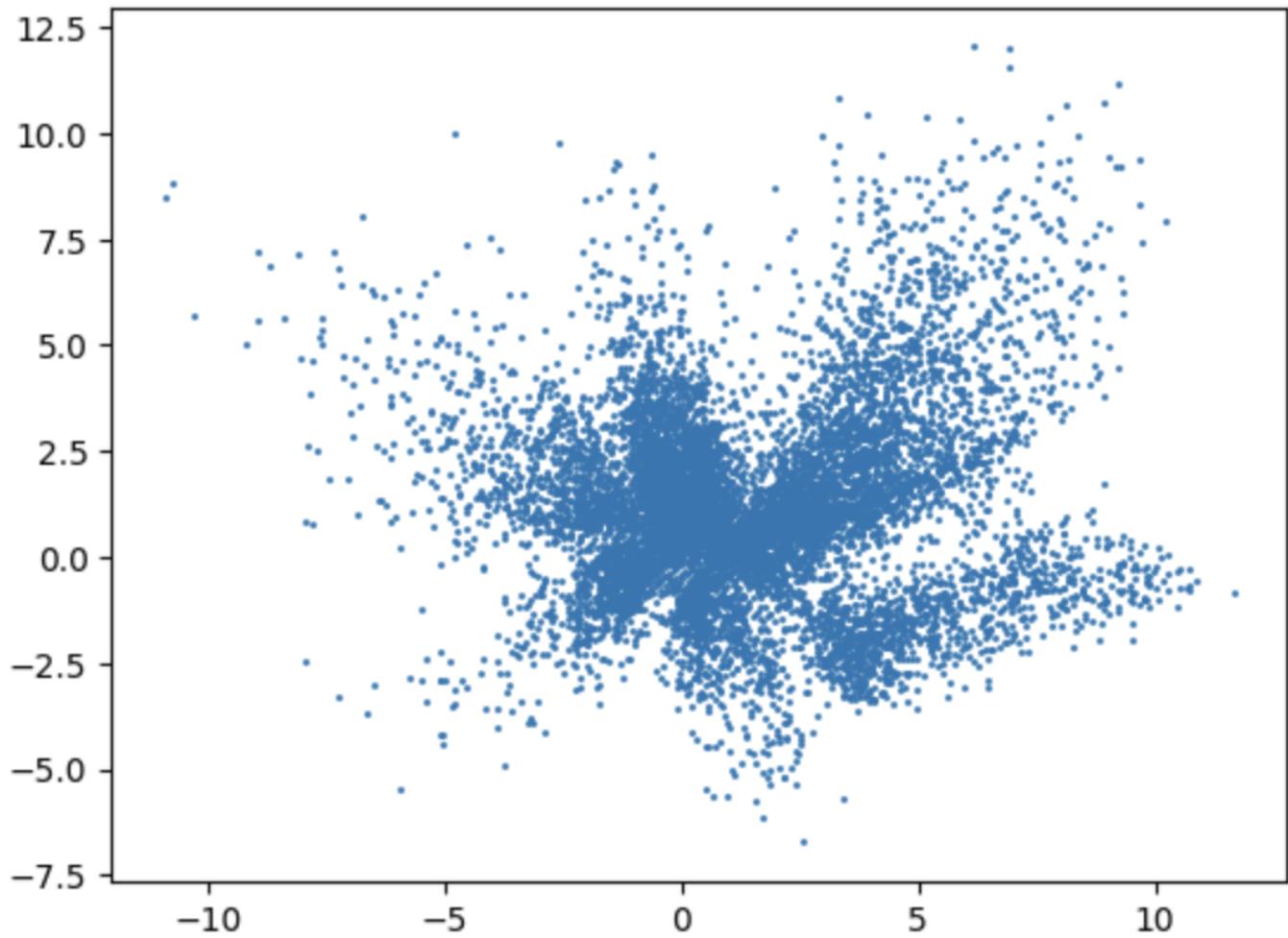
```
autoencoder.compile(optimizer='adam', loss=keras.losses.MeanSquaredError)
```

Wartości wejściowymi są wartościami ciągłymi na przedziale [0, 1], a poprzez zastosowanie sigmoidy jako funkcji aktywacji w ostatniej warstwie również wartości wyjściowe przyjmą taki sam przedział. Ponieważ nie klasyfikujemy danych do dwóch osobnych grup, 0 lub 1, lecz obliczamy funkcję kosztu dla dwóch wartości z przedziału ciągłego, dobrze nada się do tego podejście typu MSE, lub jemu podobne np.: RMSE lub MAE.

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.

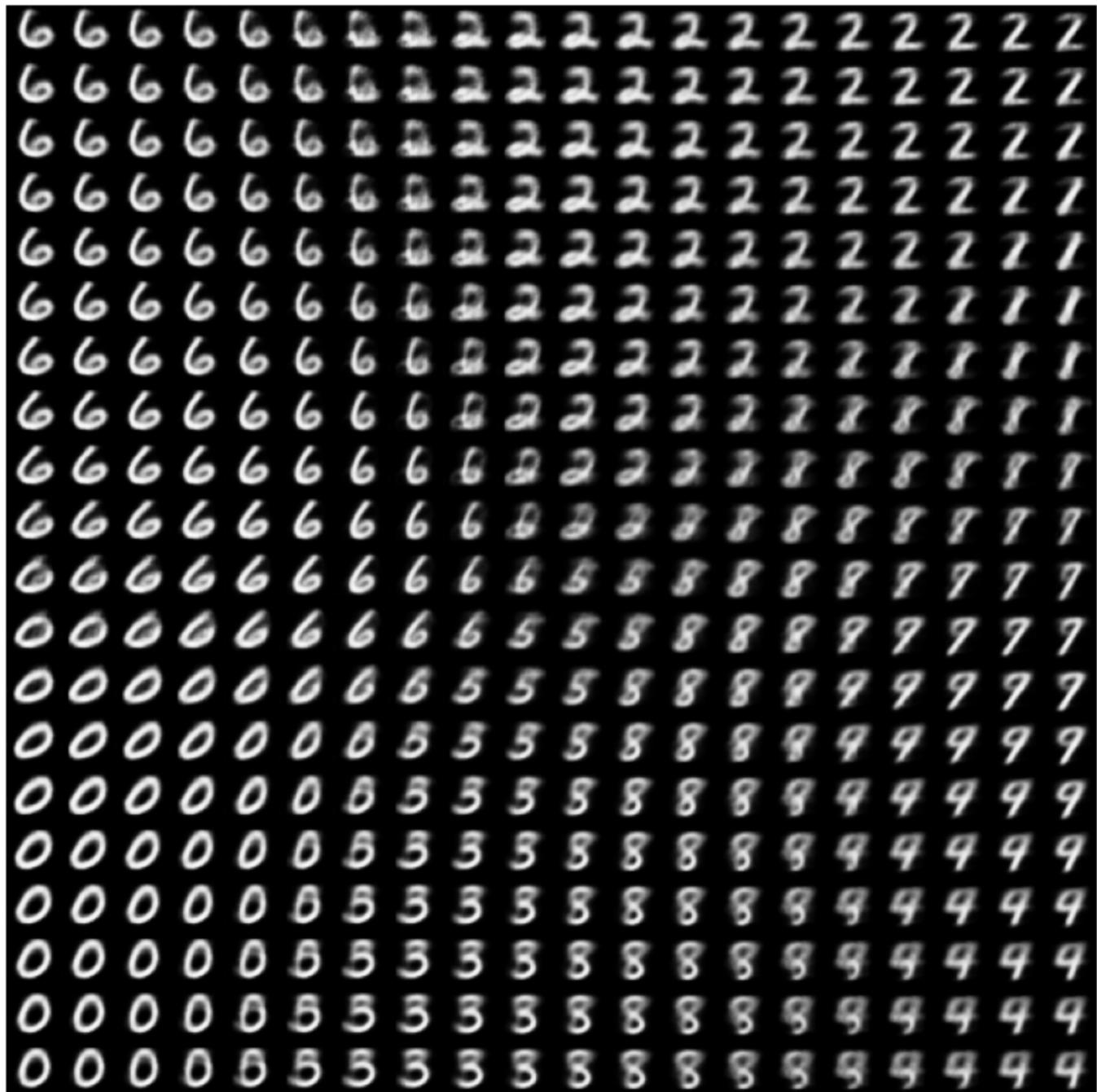
Przygotowanie:

```
def plot_latent_space(model, data):
    coords = model.encoder(data).numpy()
    plt.scatter(coords[:, 0], coords[:, 1], s=1)
    plt.show()
```



```
def plot_latent_images(model, n, digit_size=28):  
    grid_x = np.linspace(-2, 2, n)  
    grid_y = np.linspace(-2.5, 2.5, n)  
  
    ...
```

Do wygenerowania tablicy liczb wybrałem obszar o największej gęstości występowania zakodowanych elementów, tj. x i y pomiędzy -2.5 i 2.5.



Wybrałem pary: 6 i 7 oraz 4 i 0

```
def combine_pairs(model, pairs):
    plt.figure(figsize=(3, 3))
    N = 3
    for i, pair in enumerate(pairs):
        enc_pair = []
        for j, num in enumerate(pair):
            ax = plt.subplot(2, N, j + 1 + i*N)
            plt.imshow(num)
            plt.title(f"Pair: {i+1}")
            plt.gray()
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)

        enc_num = model.encoder(np.array([num])).numpy()
```

```

        enc_pair.append(enc_num)
mean_enc_pair = np.mean(np.array(enc_pair), axis=0)
comb_pair = model.decoder(mean_enc_pair).numpy()
ax = plt.subplot(2, N, 3 + i*N)
plt.imshow(comb_pair[0])
plt.title(f"Comb: {i+1}")
plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

pair_6_7 = [x_test[21], x_test[0]]
pair_4_0 = [x_test[4], x_test[3]]
pairs = [pair_6_7, pair_4_0]

combine_pairs(autoencoder, pairs)

```

Dla których powyższy kod produkuje następujący wynik:

Pair: 1



Pair: 1



Comb: 1



Pair: 2



Pair: 2



Comb: 2



Pierwsza para wypadła całkiem nieźle, gdyż faktycznie uśrednienie liczb 6 i 7 wskazywało na powstanie czegoś na kształt 5 lub 8. Wygenerowany kształt bardzo przypomina 8, choć dość rozmazane.

Druga para, 4 i 0, wygenerowała coś, co jest bardzo podobne do 3 albo 9. W tym wypadku oczekiwałem 5 lub 3, czyli właściwie udało się trafić, o ile bardzo obniżyliśmy swoje oczekiwania co do czytelności wygenerowanego znaku.

Jak widać nie zawsze udaje się wygenerować jednoznaczne liczby, natomiast obydwa rezultaty mają część wspólną: są mocno rozmazane.

2. AutoEnkoder wariacyjny (VAE)

Zadanie 2.1. Dlaczego powyższa implementacje CVAE nie stosuje żadnej aktywacji w ostatniej warstwie enkodera? Czy jakaś funkcja by się tutaj nadawała? (0.25pkt)

Warstwa latentna zwaraca parametry rozkładu Gaussa, takie jak średnia i odchylenie standardowe, czy też jak w naszym przypadku logarytm z wariancji. Obydwie te wartości należą do zbioru liczb rzeczywistych i dla zapewnienia lepszych efektów uczenia się enkodera, wartości te powinny pozostać bez zmian na wyjściu z niego.

Przyjrzyjmy się kilku funkcjom aktywacji:

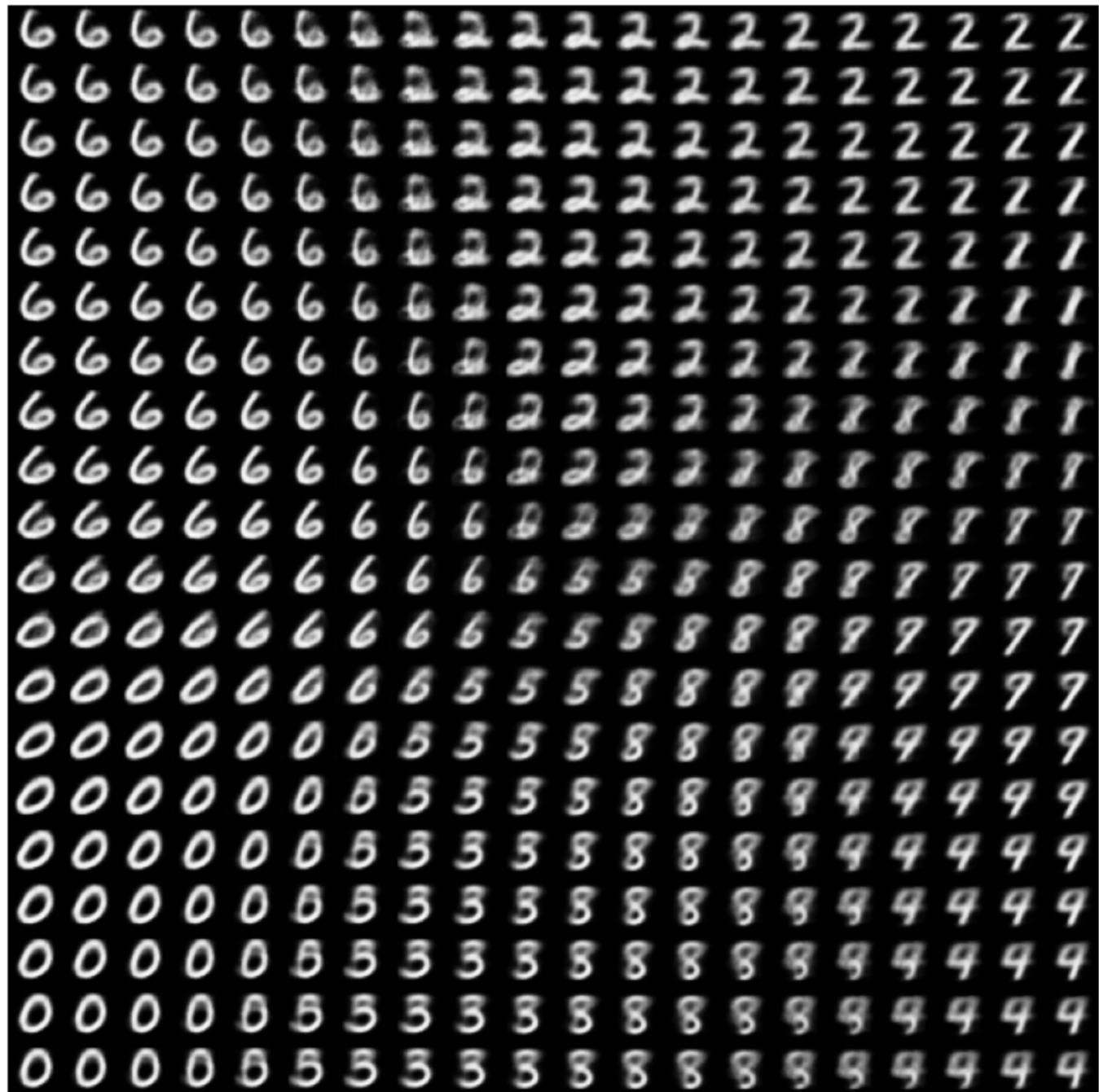
- ReLU - zeruje wartości ujemne
- Sigmoida - ma dużą zmienność dla dziedziny w okolicach 0, a w pozostałych przedziałach dziedziny zmienia się znacznie wolniej
- tanh - podobnie do sigmoidy

Zastosowanie funkcji aktywacji zaburzyłoby tylko wartości rozkładu, którego specyfiki próbuje się nauczyć enkoder, uwaga ta dotyczy się wszystkich funkcji nieliniowych. Funkcje liniowe natomiast tylko przeniosą przedział wartości wyjściowych, więc ma z nich w tym przypadku żadnego porzytku.

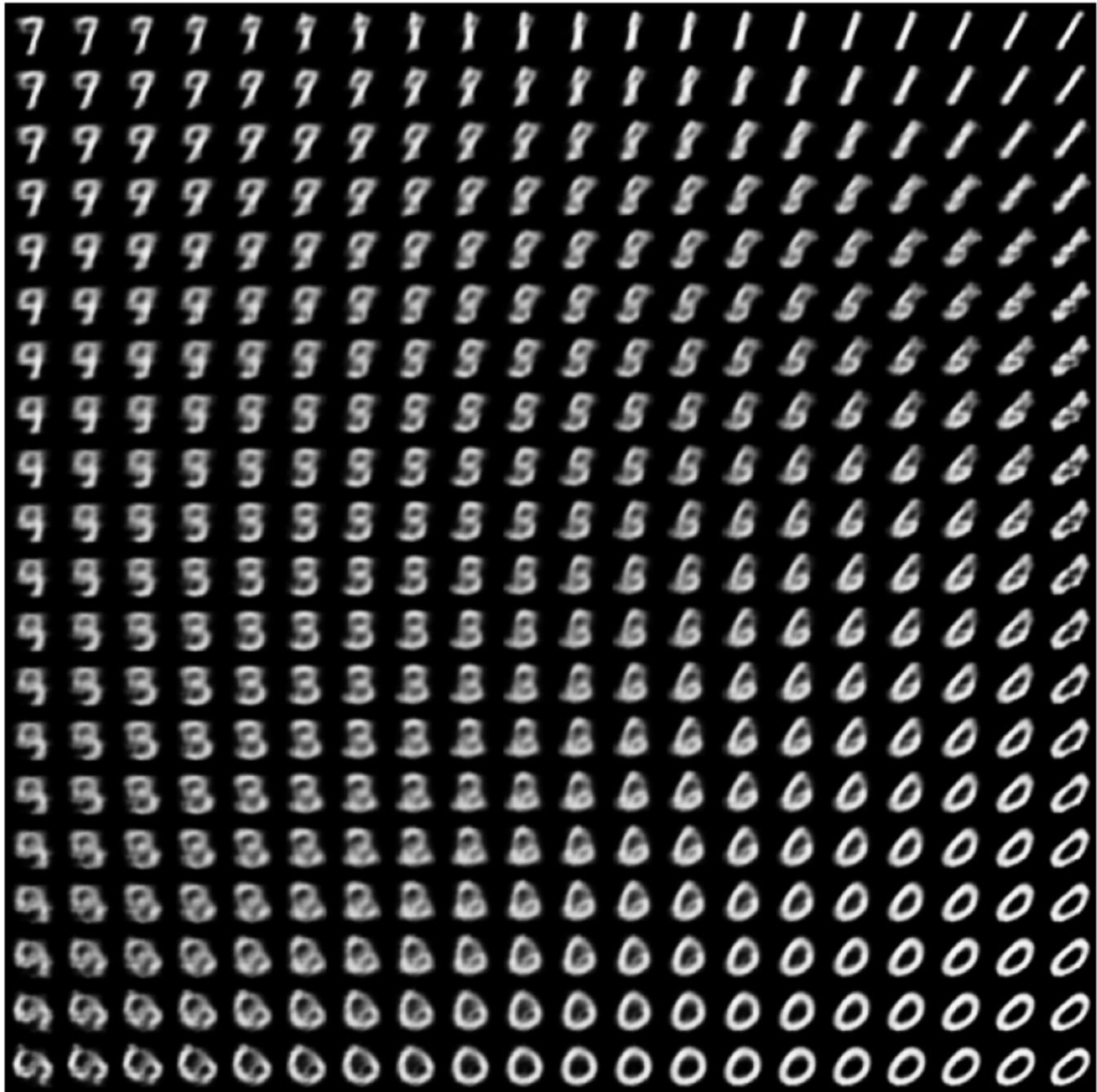
Na podstawie powyższej argumentacji stwierdzam, że w tym przypadku do poprawy jakości modelu nie nadaje się żadna funkcja aktywacji.

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. (0.25pkt)

Najpierw powtórzę tablicę liczb dla AE:



A poniżej wkleję tablicę liczb dla VAE:



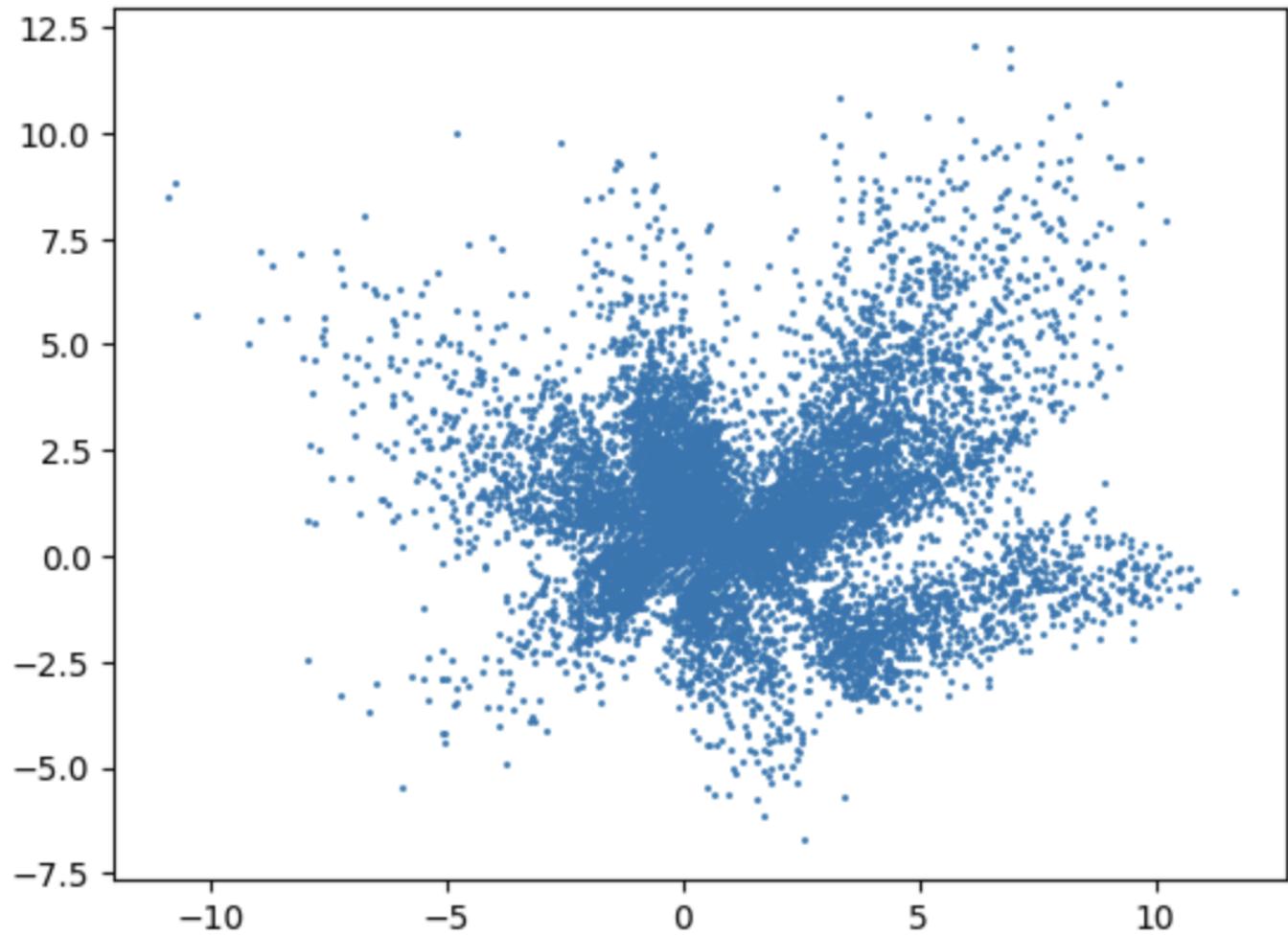
Czego należało się spodziewać, rozmieszczenie liczb w tablicy jest różne, co wiąże się z różnicami, jakie przyjmują zmienne latentne dla obydwu modeli.

W obydwu przypadkach udało się zaprezentować większość występujących cyfr i chociaż obszar dla VAE w przestrzeni kartezańskiej jest znacznie mniejszy, ilość uwzględnionych punktów z wybranego zakresu jest dość podobna.

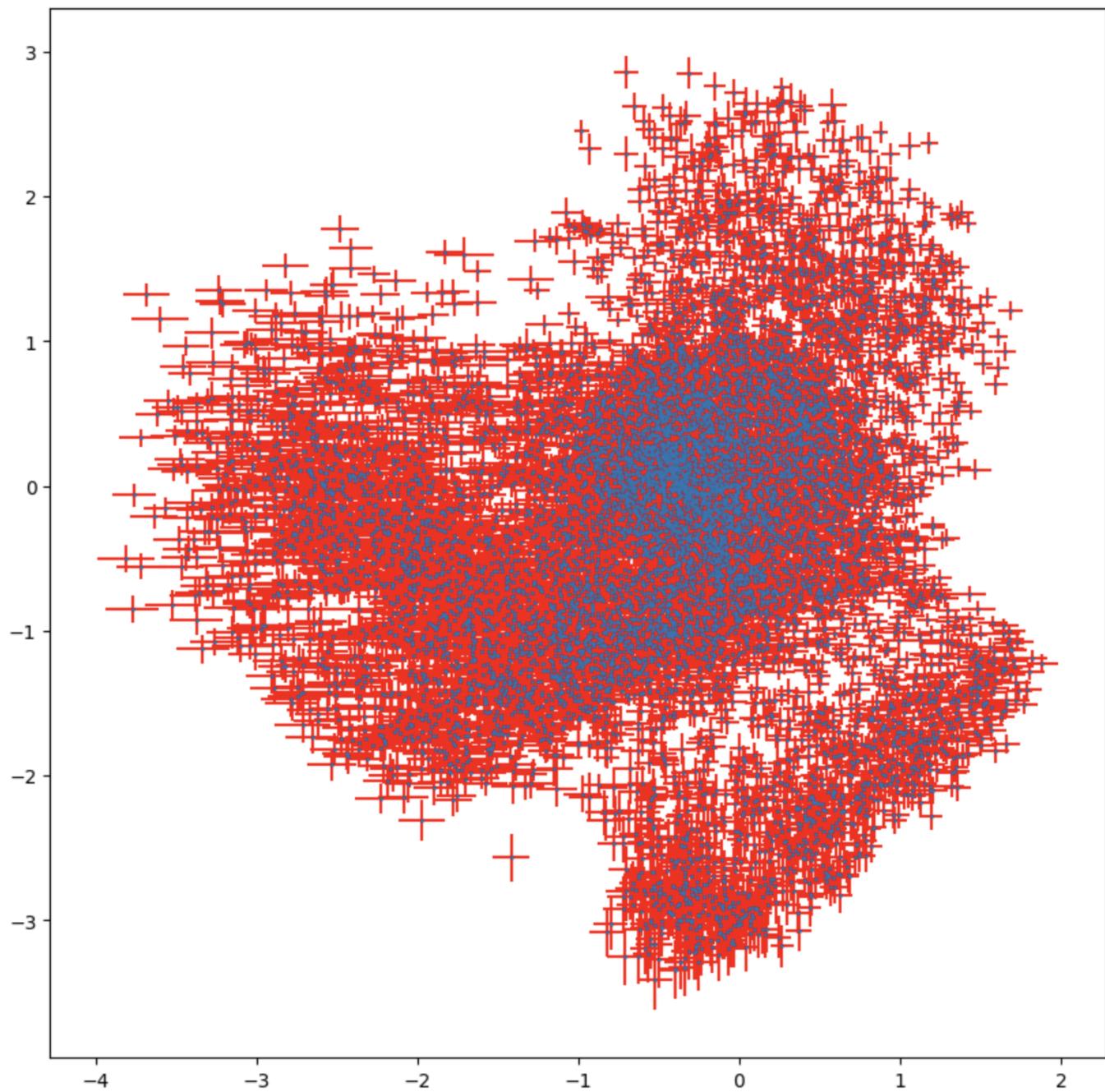
Jakościowo wyniki nie odbiegają znacząco jedne od drugich. Zarówno wśród wyników AE, jak i VAE jasno widać przenikanie się pomiędzy cyframi i użytych modeli, których trening trwał 10 epok, nie jestem w stanie wskazać faworyta.

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.

Z poprzedniej sekcji uzyskaliśmy następujący rozkład zmiennych latentnych w AE:



Rozmieszczenie zmiennych latentnych w przestrzeni kartezjańskiej odpowiadających za rozkład prawdopodobieństw użytych sampli w VAE:



Funkcja użyta do generacji powyższego wykresu dla VAE:

```
def plot_latent_space(model, data):
    mean, logvar = model.encode(data)
    std = tf.exp(logvar / 2)

    plt.figure(figsize=(10, 10))
    plt.errorbar(mean[:, 0], mean[:, 1], xerr=std[:, 0], yerr=std[:, 1],
    ecolor='red', fmt='o', markersize=1, )
    plt.show()
```

Jak już pokrótce zauważyłem w poprzednim zadaniu, punkty w wykresie dla VAE są gęściej upakowane, a ich przedział wartości jest mniejszy, niż dla AE.

Uwzględniając odchylenia standardowe, wciąż widać znaczne przeciecia pomiędzy sąsiadującymi punktami, które mogą reprezentować różne liczby, czego efektem jest przenikanie się reprezentacji cyfr tworzonych przez enkoder - patrz wyniki z zadania 2.2.

Ciekawym elementem wykresu dla VAE jest grupa w prawej-dolnej części wykresu, która się znacznie odłączała od reszty. Porównując wykres z tablicą liczb dla VAE można uznać, że jest to grupa generująca zera.

Zadanie 2.4. Dla tych samych par obrazów, na których pracowałaś/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.

Funkcja generująca reprezentacje dla uśrednionych sampli:

```
def combine_pairs(model, pairs):
    plt.figure(figsize=(3, 3))
    N = 3
    for i, pair in enumerate(pairs):
        enc_pair = []
        for j, num in enumerate(pair):
            ax = plt.subplot(2, N, j + 1 + i*N)
            plt.imshow(num)
            plt.title(f"Pair: {i+1}")
            plt.gray()
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)

            mean, logvar = model.encode(np.array([num]))
            z = model.reparameterize(mean, logvar).numpy()
            enc_pair.append(z)

        mean_enc_pair = np.mean(np.array(enc_pair), axis=0)
        comb_pair = model.sample(mean_enc_pair)
        ax = plt.subplot(2, N, 3 + i*N)
        plt.imshow(comb_pair[0])
        plt.title(f"Comb: {i+1}")
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

    pair_6_7 = [x_test[21], x_test[0]]
    pair_4_0 = [x_test[4], x_test[3]]
    pairs = [pair_6_7, pair_4_0]

    combine_pairs(model, pairs)
```

wygenerowała następujący wynik:



Dla pary 6 i 7 spodziwałam się zobaczyć 5 albo 9 i faktycznie uzyskana cyfra jest dziewiątką.

Druga para liczb, mianowicie 4 i 0, wygenerowały liczbę 3. Porównanie wyniku z tablicą liczb VAE jest trudniejsze, gdyż o ile 0 jest łatwe do odnalezienia, 4 właściwie nie występuje w tablicy. Natomiast wiedząc, że para 4 i 0 generuje 3, a 3 jest na lewo od 0, to 4 musi znajdować się całkiem po lewej stronie, ale zostało wycięte przy doborze obszaru do stworzenia tablicy liczb. Faktycznie kształty po lewej stronie tablicy mogą przypominać 4.

Wygenerowane liczby nie są zupełnie czytelne, ale wg mnie są czytelniejsze, od tych wygenerowanych przez AE.

Oczywiście, skoro rozlokowanie liczb w przestrzeni 2D jest odmienne dla AE i VAE, wygenerowane liczby z uśrednienia tych samych liczb wejściowych powinny być różne dla każdego z modeli i tak też jest.

Zadanie 2.5. Wróć do funkcji `compute_loss` ... wykorzystaj wzór na KL-divergence dla dwóch rozkładów gaussowskich. Zamieść w raporcie przygotowaną formułę. Wytrenuj model ponownie, porównaj wyniki z poprzednią implementacją `compute_loss`.

Formuła KL-divergence dla dwóch rozkładów gaussowskich wygląda następująco:

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

któłą można zapisać w postaci funkcji python w następujący sposób:

```
def kl_div(mean, logvar):
    var = tf.exp(logvar)
```

```

trace_var = tf.reduce_sum(var, axis=1)
mean_squared_norm = tf.reduce_sum(tf.square(mean), axis=1)
log_det_sigma = tf.reduce_sum(logvar, axis=1)
k = tf.cast(tf.shape(mean)[1], tf.float32)

return 0.5 * (trace_var + mean_squared_norm - k - log_det_sigma)

```

Objaśnienie co do wartości ze wzoru:

- na podstawie logarytmu wariancji wyciągamy wariancję
- ślad macierzy Σ to suma wartości wekora wariancji
- średnia to wektor dwuliczbowy, z którego obliczamy kwadrat normy euklidesowej
- dla stabilności obliczeń w ostatnim elemencie, zamiast obliczać logarytm pomnożonych wartości wektora wariancji, dodaję do siebie wartości `logvar`, co sprowadza się do tego samego wyniku
- k to liczba długość wektorów wejściowych, czyli 2

Dodatkowo, funkcja została zapisana w taki sposób, aby operowała na minibatchach.

Funkcja `compute_loss_gauss_kl` wykorzystująca `kl_div` przyjmuje następującą postać:

```

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])

kl_div_val = kl_div(mean, logvar)

return -tf.reduce_mean(logpx_z - kl_div_val)

```

Wyniki uzyskane przy użyciu `compute_loss_gauss_kl` nie różnią się niczym od tych uzyskanych stosując bazową funkcję straty `compute_loss`, a wartość `ELBO` dla obydwu podejść po ukończeniu treningu jest prawie taka sama i wynosi ok 158.

3. Conditioned VAE

Uzupełniona funkcja do generowania siatek liczb i wyboru podanej liczby z nich:

```

def conditioned_mnist(x, y, num_imgs=2):
    x_res = np.empty(shape=(x.shape[0]*num_imgs, x.shape[1]*3,
    x.shape[2]*3), dtype='float32') # pusta macierz z wynikami - obrazy x
    y_res = np.empty(shape=(y.shape[0]*num_imgs, 12), dtype='float32') # pusta macierz z wynikami - wektor y: etykieta (10 liczb), pozycja x, pozycja y
    empty_res = np.zeros(shape=(x.shape[1]*3, x.shape[2]*3)) # 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(np.arange(0, 9), size=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

```

Budowa sieci neuronowych enkodera i dekodera:

W poprzednim zadaniu zarówno enkoder, jak i dekoder od pewnego poziomu operują na obrazach 14x14. Tą część możemy zostawić, natomiast dla enkodera dołączymy dodatkową warstwę konwolucyjną, która zmniejsza wejściowy obraz 42x42, do pożądanych 14x14. Odpowiednio dla dekodera dołączamy warstwę dekonwolucyjną zwiększającą obraz z 14x14 do 42x42.

Warstwy VAE wyglądają w następujący sposób:

```

def prepare_encoder(self):
    input_img = tf.keras.layers.Input(shape=(42, 42, 1))
    input_cond = tf.keras.layers.Input(shape=(12, ))
    #convolution 42x42 to 14x14, kernel overlapping
    x = tf.keras.layers.Conv2D(filters=32, kernel_size=4, strides=(3, 3),
activation='relu')(input_img)
    x = tf.keras.layers.Conv2D(filters=64, kernel_size=3, strides=(2, 2),
activation='relu')(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(units=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)

```

```
#deconvolution 14x14 to 42x42, kernel overlapping
x = tf.keras.layers.Conv2DTranspose(filters=32, kernel_size=4,
strides=3, padding='same', activation='relu')(x)
# No activation
x = tf.keras.layers.Conv2DTranspose(filters=1, kernel_size=4,
strides=1, padding='same')(x)
return tf.keras.Model([input_latent, input_cond], [x])
```

W funkcji koszta liczymy logity w następujący sposób:

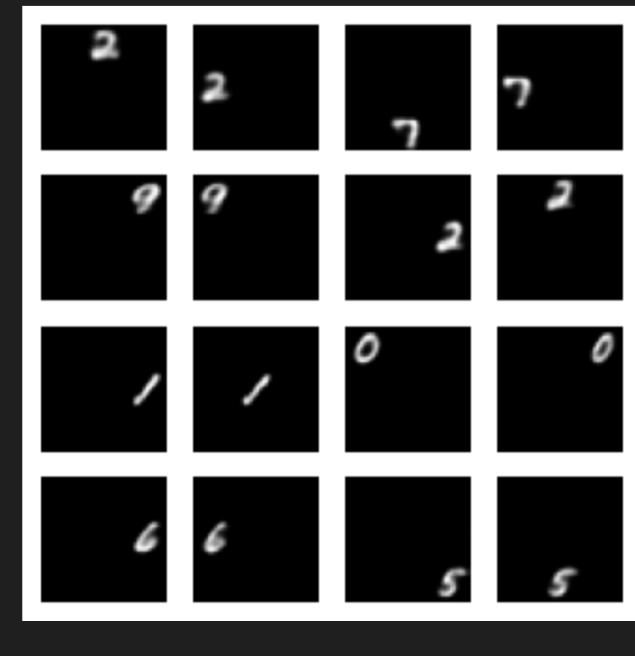
```
x_logit = model.decode(z, cond)
```

Zadanie 3.1. Sprawdź jakość modelu dla 3 różnych wartości latent_dim

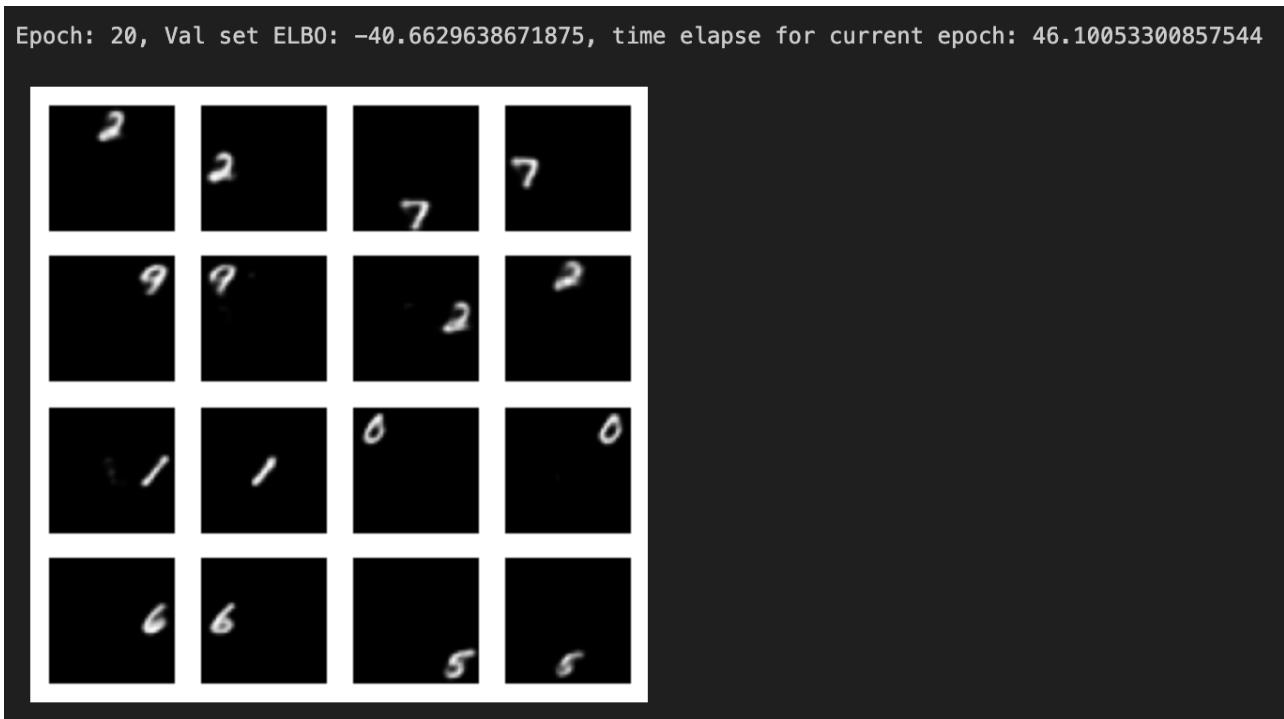
Dla tak przygotowanego CVAE uruchomiłem trening w zaleconych wariantach:

- Neurony warstwy latentnej: 2, liczba epok: 10

```
Epoch: 10, Val set ELBO: -41.28705978393555, time elapse for current epoch: 48.87895107269287
```



- Neurony warstwy latentnej: 25, liczba epok: 20



- Neurony warstwy latentnej: 100, liczba epok: 50



Wizualnie wszystkie wyniki wyglądają bardzo podobnie i nie sposób powiedzieć, który z nich jest najlepszy. Generalnie wszystkie uważam za wystarczająco dobre jakościowo. Natomiast wartość ELBO wyszła najlepsza dla największych badanych hiperparametrów, więc przy tym modelu zostaję w dalszych zadaniach: `latent_dim = 100, epoch = 50`

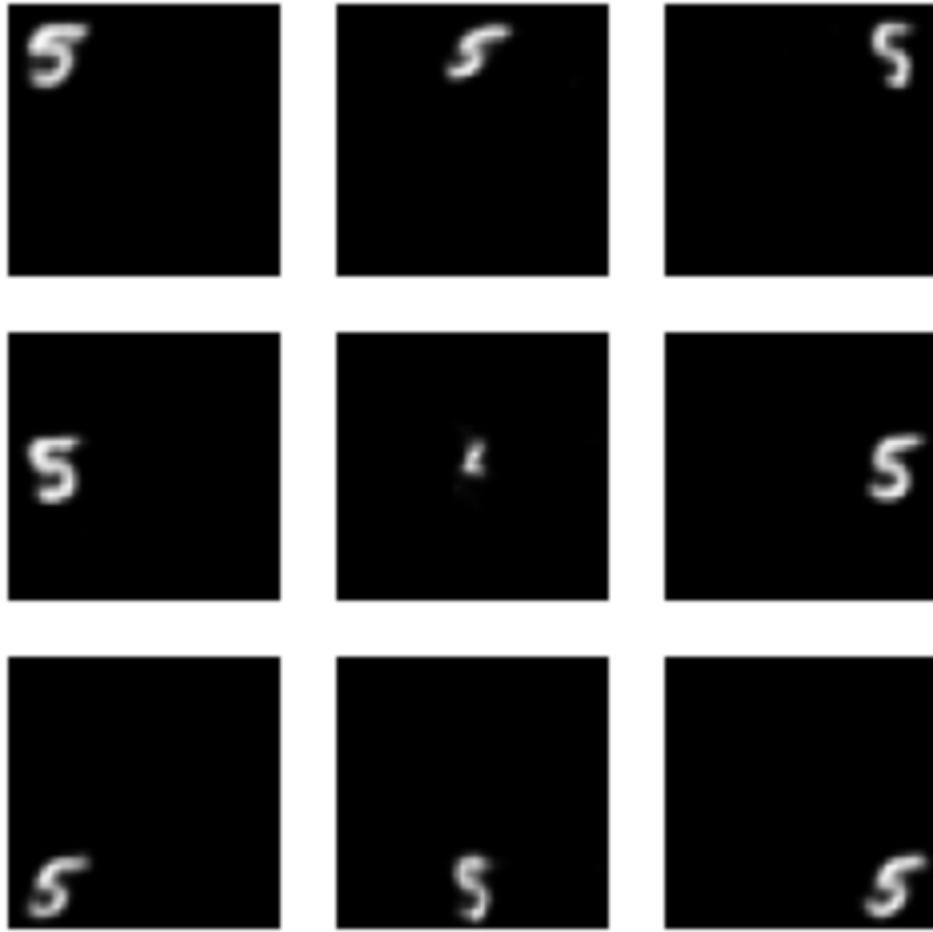
Zadanie 3.2. Dla każdego z 9 możliwych wektorów przepuść przez dekoder reprezentację z wybranego przypadku testowego

Poniższy kod został przygotowany do zrealizowania zadań 3.2 i 3.3. W zależności od użycia wygeneruje siatkę liczb na podstawie przypadku testowego, lub dla podanej etykiety i szumu wygenerowanego z

rozkładu gaussowskiego. W zależności od przypadku należy uruchomić `generate_test_cases` z wartością `None`, żeby użyć kolejnego przypadku testowego, lub podać porządaną wartość liczbową.

Dla tego zadania uruchomienie wygląda w następujący sposób:

```
test_cases = generate_test_cases(None)
sample_and_viz(test_cases)
```



Generacja wyszła bardzo dobrze dla wybranego przypadku testowego. Jedynie cyfra wygenerowana w środkowej komórce nie bardzo przypomina "5", ale w pozostałych komórkach wynik można opisać za bez zarzutu.

Jednak nie zawsze udaje się otrzymać tak dobre wyniki, a zwłaszcza korzystając z modeli o mniejszej liczbie neuronów w warstwie latentnej, które również sprawdziłem, lecz wyników nie zamieściłem w sprawozdaniu. Zbyt mała liczba neuronów w warstwie latentnej sprzyja niewystarczającej separacji poszczególnych przypadków, przez co możemy obserwować przenikanie się elementów z różnych grup - dokładnie tak samo, jak to miało miejsce w dwóch poprzednich sekcjach ćwiczenia, z tym że tym razem to przenikanie przybiera trochę inny wygląd. Cyfry są rzadziej rozmazane, ale potrafią pojawić się w nieodpowiednich komórkach.

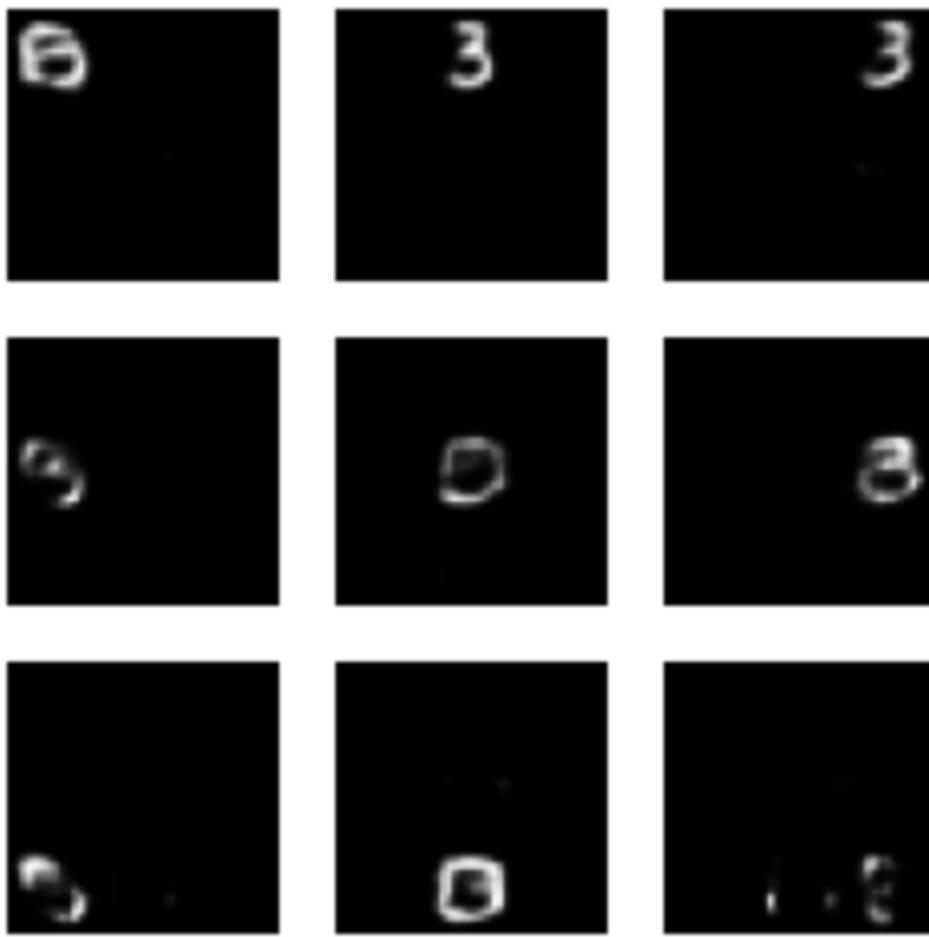
Do sprawozdania dołączam kilka plików gif, które prezentują przebieg uczenia się poszczególnych modeli. Przyglądając się CVAE.gif można zauważyc etapy, gdy faktycznie niektóre cyfry były wyświetlane w komórkach różnych, od tych zadanych przez nasze warunki.

Zadanie 3.3. Dla każdego z 9 możliwych wektorów przepuść przez dekoder szum z rozkładu normalnego i wybraną etykietę

Wywołanie funkcji dla tego zadania:

```
test_cases = generate_test_cases(3)
sample_and_viz(test_cases)
```

I otrzymany wynik:



Tym razem jakość generowanych wyników jest słabsza, niż w poprzednim zadaniu. Wiąże się to z tym, że zadanie zlecone dekoderowi jest trudniejsze, a to gdyż na wejściu nie dostaje on wartości z wychodzących z enkodera, a szum z rozkładu gaussowskiego.

Właściwym jest oczekiwany wobec dekodera możliwości wygenerowania zadanego przypadku na podstawie szumu i dostarczonych warunków, jednakże najwyraźniej nasz model wciąż nie jest w stanie zawrzeć w swoich strukturach pełni informacji potrzebnej do właściwej generacji wyłącznie na podstawie dostarczonych warunków.

Przyjrzyjmy się bliżej otrzymanym wynikom, pola 1,1, 2,2 oraz cały trzeci wiersz nie stanowią ładnej reprezentacji zadanej cyfry "3". Dodatkowo w ostatniej komórce, 3,3, pojawił się jakiś artefakt obok wygenerowanej cyfry.

Jak i w poprzednim przypadku widać, że warstwa latentna nie wyizolowała w dostateczny sposób przypadków do wygenerowania czytelnego i pożądanej obrazu wyjściowego. Najpewniej można by uyskać lepsze wyniki zwiększając pojemność sieci i jednocześnie wydłużając ilość epok treningu. Może również pomóc dalsze zwiększenie wielkości warstwy latentnej, ale obecne 100 węzłów wydaje się wystarczającą liczbą, a wręcz dużą. Oczywiście otwartym pytaniem pozostaje, czy ilość danych treningowych wystarczy, żeby wykorzystać potencjał sieci przy jej zwiększonej pojemności.

Conditioned GAN

Definicja generatora została uzupełniona o specyfikację wejścia w następujący sposób:

```
inputs = tf.keras.layers.concatenate(axis=1)([input_img, input_cond])
```

Dyskryminator został uzupełniony następującymi warstwami dla wejścia `input_cond`:

```
x = tf.keras.layers.Dense(256, activation='relu')(x)
x = tf.keras.layers.Dropout(0.3)(x)
x = tf.keras.layers.Dense(128, activation='relu')(x)
x = tf.keras.layers.Dropout(0.3)(x)
```

Funkcja kosztu po uzupełnieniu wygląda w następujący sposób:

```
def discriminator_loss(real_output, fake_output):
    # 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
```

Poniżej przygotowana funkcja trenująca pojedyńczy krok:

```
@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, noise_cond], training=True)

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

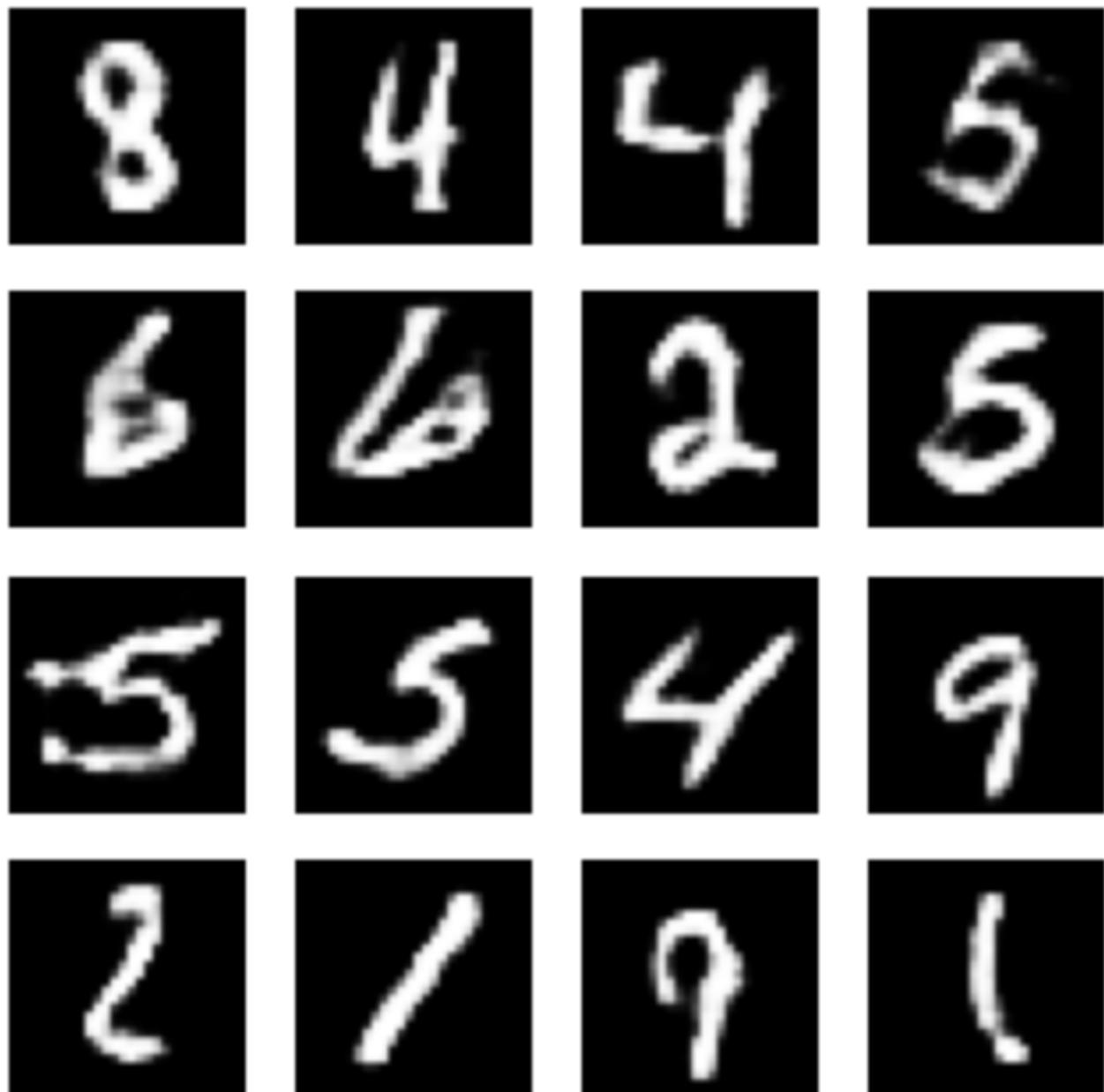
```
gen_loss = generator_loss(fake_output)
disc_loss = discriminator_loss(real_output, fake_output)

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))
```

Dla danych w tej części ćwiczenia, tak przygotowanym modelem, trening zajął ok 90 minut, a przykład wygenerowanych danych testowych wygląda obiecująco:



Zadanie 4.1. Wygeneruj po jednym obrazie z każdą liczbą z pomocą generatora. Oceń jakość wyników.

Przygotowałem funkcję generującą wytrenowanym modelem liczby z przedziału [0, 9]:

```
fig = plt.figure(figsize=(12, 5))
for i in range(10):
    init_label = np.zeros(10)
    init_label[i] = 1
    num_label = tf.constant(init_label.reshape(1, -1))

    noise = tf.random.normal([1, latent_dim])

    generated_image = generator([noise, num_label], training=False)
    plt.subplot(2, 5, i + 1)
    plt.imshow(generated_image[0, :, :, 0], cmap='gray')
    plt.axis('off')

plt.show()
```



Jak widać wyniki wyglądają dobrze, liczby są czytelne, nie rozmazane i trudne do pomylenia. Reprezentacja jest dużo czytelniejsza, niż w przypadku AE i VAE, a nawet CVAE. Pokazuje to wyższość modelu GAN, gdy został on odpowiednio wytrenowany, czyli gdy generator potrafi dobrze oszukać dyskryminator wygenerowanymi wartościami, a dyskryminator nie stał się "wszechwiedzący" i niemożliwy do oszukania - czyli gdy układ generator-dyskryminator pozostaje w równowadze.