Metody Rozpoznawania Obrazów
Zadanie 05 - I want to ride my bicycle!*

Wojciech Michaluk

21.11.2025

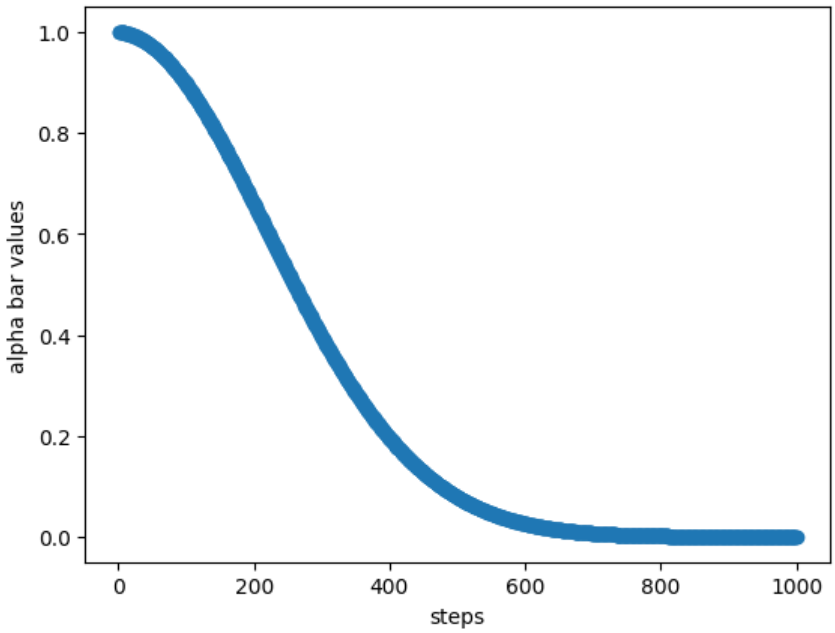
Opis kolejnych etapów zadania

Wczytuję dane z pliku **bicycle.txt**, zapisując je do list, które konwertuję na tablice *NumPy'owe*, a następnie na tensory (`tensorflow.Tensor`) - to ułatwia w późniejszej części przedstawianie rezultatów na wykresie oraz wykonywanie operacji na pełnych batchach danych, co przyda się podczas treningu sieci. Wszystkich punktów danych jest dokładnie 50468, natomiast losuję spośród nich podzbiór 1000 punktów i rysuję wynik z wykorzystaniem biblioteki `matplotlib`. Prezentuje się on następująco:

Rzeczywiście, przypomina to rower - powinno być w porządku. Teraz przystąpię do obliczenia sekwencji parametrów β_t, α_t oraz $\bar{\alpha}_t$ dla $t \in \{1, 2, \dots 1000\}$, używanych w dyfuzji oraz w procedurze generacji po wytrenowaniu sieci. Przyjmuję wartości β_t tak jak opisano w poleceniu i na ich podstawie obliczam wartości pozostałych parametrów.

*I want to ride my bike!

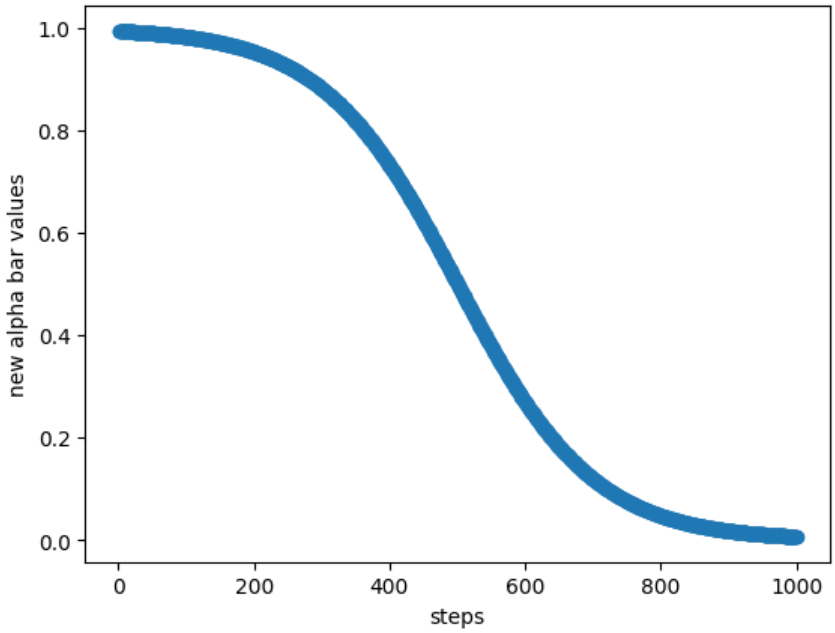
Uzyskany ciąg $\bar{\alpha}_t$ dla kolejnych kroków t wygląda następująco:



Następnie zmieniam sekwencję $\bar{\alpha}_t$ na taką, która stopniowo zanika od 1 do 0, przyjmując wartość 0.5 dla $t = \frac{T}{2} = 500$. W tym celu wykorzystuję funkcję sigmoidalną:

$$f(t) = \frac{1}{1 + e^{\frac{10}{T}t - 5}} \text{ dla } t \in \{1, 2, \dots 1000\},$$

gdzie $T = 1000$. Taka funkcja f zapewnia pożądany przebieg. Wykres jej wartości w zależności od kroku t przedstawiam poniżej.



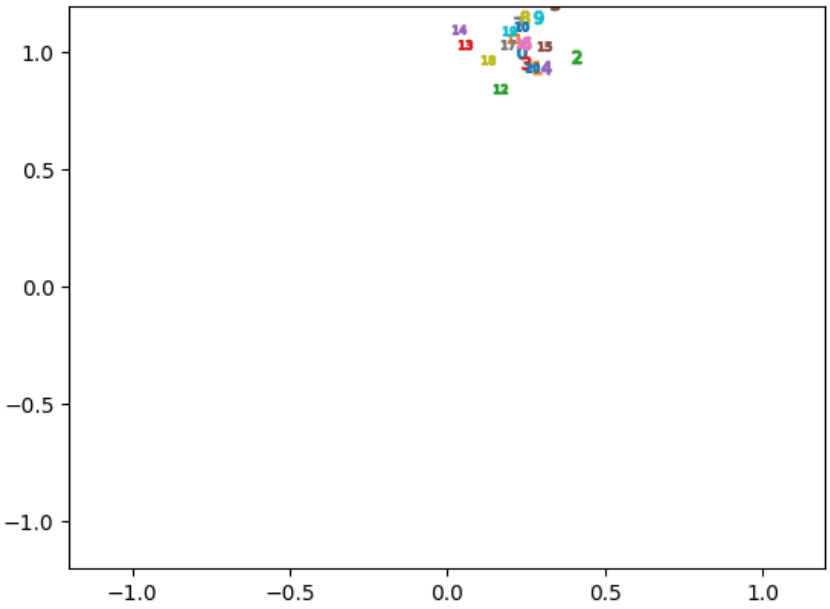
Teraz wyliczę pozostałe parametry na podstawie $\bar{\alpha}_t$, korzystając z zależności między nimi. Jest to konieczne, ponieważ te parametry są później wykorzystywane w procesie odszumiania. Przekształcając wzór:

$$\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i,$$

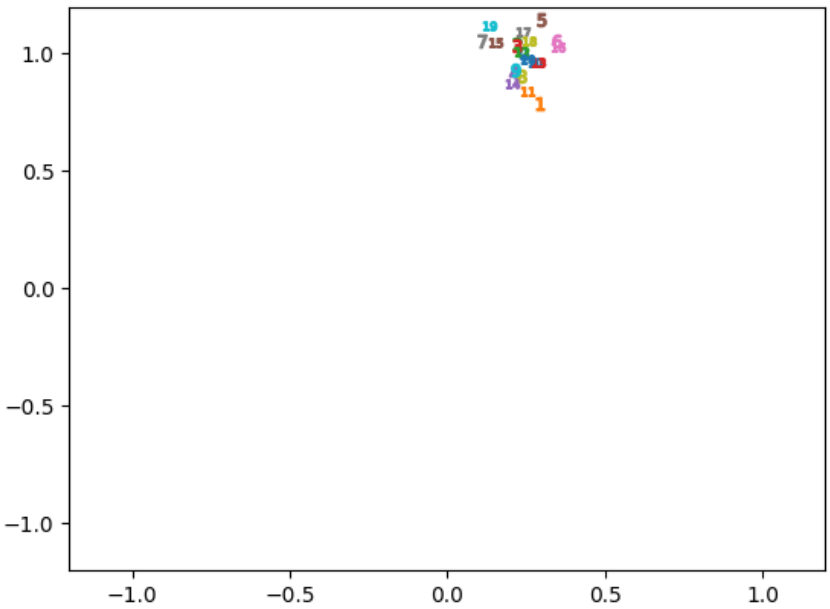
otrzymuję:

$$\alpha_1 = \bar{\alpha}_1, \ \alpha_t = \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} \text{ dla } t \in \{2, 3, \dots 1000\}.$$

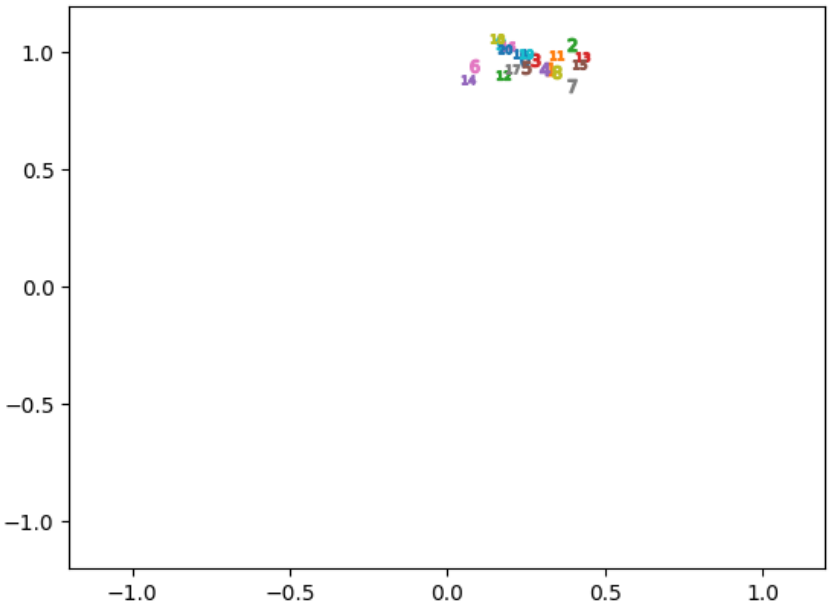
Z kolei mając wartości parametrów α_t, przekształcenie zależności $\alpha_t = 1 - \beta_t$ jest proste i prowadzi do wzoru $\beta_t = 1 - \alpha_t$. Mogę teraz przystąpić do przetestowania działania dyfuzji dla wybranych początkowych przykładów ze zbioru danych, obliczając dla nich położenie przez pierwsze 20 etapów dyfuzji. Położenie w danym etapie jest reprezentowane przez liczbę z numerem tego etapu. Przedstawiam wyniki dla arbitralnie wybranych 3 punktów.



Rysunek 1: Początkowe etapy dyfuzji dla 1. punktu ze zbioru danych



Rysunek 2: Początkowe etapy dyfuzji dla 2. punktu ze zbioru danych



Rysunek 3: Początkowe etapy dyfuzji dla 101. punktu ze zbioru danych

Punkty są oryginalnie dość blisko siebie, więc w każdym przypadku są skupione w podobnym obszarze. Proces wydaje się przebiegać prawidłowo. Przy implementacji zadbałem o wykorzystanie operacji na tensorach, aby uniknąć niepotrzebnych pętli. Przechodzę teraz do implementacji sieci neuronowej zgodnie z poleceniem zadania. Wykorzystuję do tego framework `keras`. Pozwala on na zdefiniowanie własnej warstwy w sieci oraz określenie warstw wejściowych i wyjściowych modelu, co ułatwia budowę niestandardowej sieci, jak w przypadku tego zadania. Pomocniczo definiuję blok **learnable sinusoidal embedding**, żeby móc go wykorzystać w architekturze sieci. Tworzę klasę, która dziedziczy po klasie `keras.layers.layer` i definiuję dla niej metodę `call`.

```
class LearnableSinusoidalEmbedding(layers.Layer):
    def __init__(self):
        super(LearnableSinusoidalEmbedding, self).__init__()

        self.d = 50
        self.dense1 = layers.Dense(128, activation="relu")
        self.dense2 = layers.Dense(128)


    def call(self, t):
        d_range = tf.range(self.d, dtype=tf.float32)
        div_term = tf.math.pow(10000.0, 2 * (d_range // 2) / self.d)
        t = tf.cast(t, tf.float32)

        # kodowanie pozycyjne
        pe = tf.expand_dims(t, -1) * tf.expand_dims(div_term, 0)
        sin_pe = tf.sin(pe[:, 0::2])
        cos_pe = tf.cos(pe[:, 1::2])
        pe = tf.concat([sin_pe, cos_pe], axis=-1)

        # przepuszczenie przez warstwy FC
        output = self.dense1(pe)
        output = self.dense2(output)

        return output
```

Listing 1: Zdefiniowanie bloku jako warstwa w sieci

Następnie przygotowuję funkcję budującą model DDPM, która wykorzystuje powyższy blok (przekazany jako parametr `embedding_layer`) oraz standardowe warstwy w sieci neuronowej, zgodnie z architekturą opisaną w poleceniu zadania. W tej funkcji również określam wejście i wyjście modelu. Zwraca ona zbudowany, ale niewytrenowany model.

```
def build_ddpm_model(embedding_layer):
    # warstwy wejściowe modelu
    x_input = layers.Input(shape=(2,))
    time_input = layers.Input(shape=())

    # conditional layer 0
    x = layers.Dense(128)(x_input)
    t_emb = embedding_layer(time_input)
    output = layers.Add()([x, t_emb])
    output = layers.ReLU()(output)

    # conditional layer 1
    x = layers.Dense(128)(output)
    output = layers.Add()([x, t_emb])
    output = layers.ReLU()(output)

    # conditional layer 2
    x = layers.Dense(128)(output)
    output = layers.Add()([x, t_emb])
    output = layers.ReLU()(output)

    # conditional layer 3
    x = layers.Dense(128)(output)
    output = layers.Add()([x, t_emb])

    # warstwa wyjściowa modelu
    epsilon_pred = layers.Dense(2)(output)

    # stworzenie modelu i zwrócenie go
    model = models.Model(inputs=[x_input, time_input], outputs=epsilon_pred)

    return model
```

Listing 2: Funkcja budująca model

Aby sprawdzić, czy model przyjmuje i zwraca dane w odpowiednim formacie, używam polecenia `model.summary()`.

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 2)	0	-
input_layer_1 (InputLayer)	(None)	0	-
dense_2 (Dense)	(None, 128)	384	input_layer[0][0]
learnable_sinusoid… (LearnableSinusoid…	(None, 128)	23,040	input_layer_1[0]…
add (Add)	(None, 128)	0	dense_2[0][0], learnable_sinuso…
re_lu (ReLU)	(None, 128)	0	add[0][0]
dense_3 (Dense)	(None, 128)	16,512	re_lu[0][0]
add_1 (Add)	(None, 128)	0	dense_3[0][0], learnable_sinuso…
re_lu_1 (ReLU)	(None, 128)	0	add_1[0][0]
dense_4 (Dense)	(None, 128)	16,512	re_lu_1[0][0]

(a)

dense_4 (Dense)	(None, 128)	16,512	re_lu_1[0][0]
add_2 (Add)	(None, 128)	0	dense_4[0][0], learnable_sinuso…
re_lu_2 (ReLU)	(None, 128)	0	add_2[0][0]
dense_5 (Dense)	(None, 128)	16,512	re_lu_2[0][0]
add_3 (Add)	(None, 128)	0	dense_5[0][0], learnable_sinuso…
dense_6 (Dense)	(None, 2)	258	add_3[0][0]

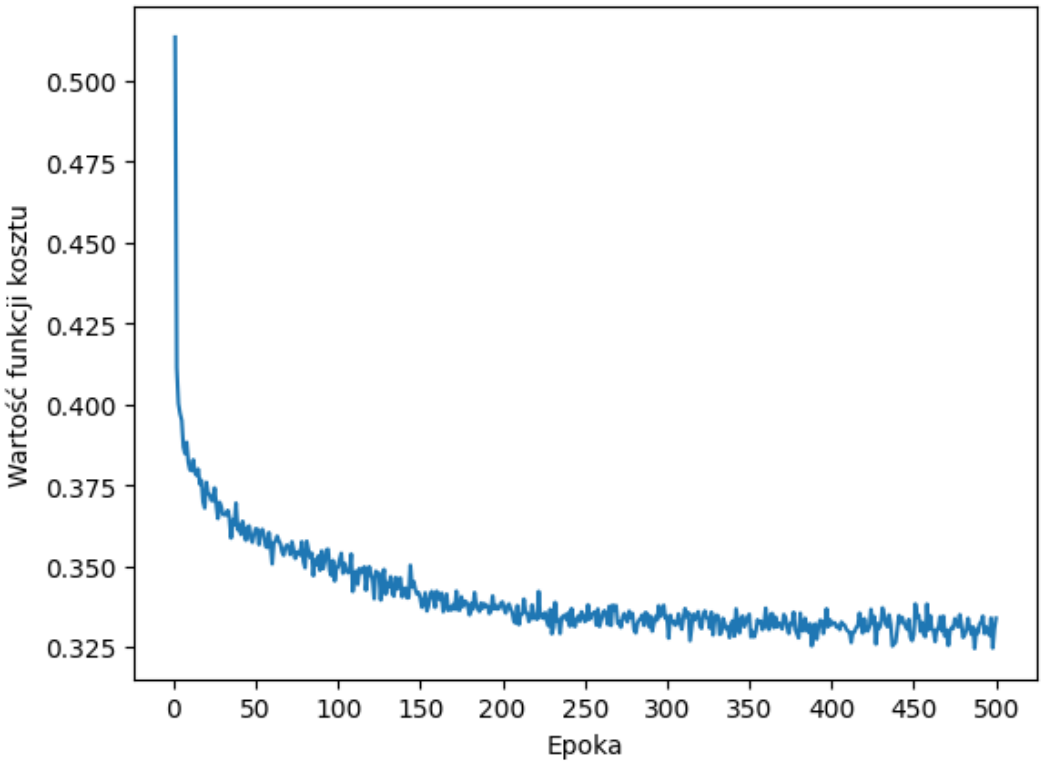
Total params: 73,218 (286.01 KB)

Trainable params: 73,218 (286.01 KB)

Non-trainable params: 0 (0.00 B)

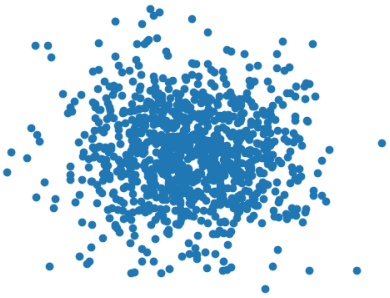
(b)

Dzielę wypisany wynik na konsoli na 2 części – (a) i (b) – bowiem jest on dość długi i nie mieści się w całości na moim ekranie. Analizując kolumny **Output Shape** oraz **Connected to**, widzimy, że wymiary i struktura sieci powinny odpowiadać pożądanym. Mogę zatem przystąpić do treningu modelu. Zgodnie z poleceniem używam batchów o rozmiarze 64, optymalizatora Adam ze współczynnikiem uczenia 0.0001, a jako funkcję kosztu przyjmuję MSE (Mean Squared Error). Drobne odstępstwo od polecenia występuje w ustalonej liczbie epok treningu. Przy początkowych próbach, okazało się, że jedna epoka treningu trwa niecałą minutę na moim laptopie. Zdecydowałem, że przyjmę 500 epok treningu zamiast pierwotnie planowanych 1000, żeby zdążyć z terminem oddania zadania oraz nieco ograniczyć (i tak długi) czas treningu. Mam jednak nadzieję, że mimo tej redukcji, uzyskane rezultaty będą zadowalającej jakości. Oba algorytmy, zarówno treningu sieci jak i późniejszego odszumiania, przepisuję wiernie z załączonego pseudokodu, zachowując odpowiednie rozkłady dla występujących zmiennych. Na potrzeby zapisywania różnych wersji modelu przyjmuję, że zapiszę stan modelu na koniec epok: 25., 100., 300. i ostatniej, 500. Z kolei w każdej epoce obliczam na koniec uśrednioną wartość funkcji kosztu (per batch) i zapisuję ją. Po wykonaniu treningu uzyskuję następujący wykres funkcji kosztu:

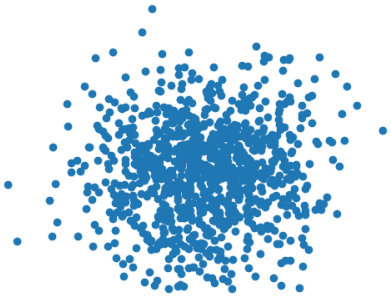


Rysunek 4: Zależność wartości funkcji kosztu od epoki

Jak widać, wartość funkcji kosztu dość szybko się stabilizuje, już po kilkudziesięciu epokach, natomiast od około 200-250. epoki występują minimalne wahania, wartość utrzymuje się w zakresie od 0.32 do 0.33. Patrząc na wybrane epoki, w 25. epoce wartość funkcji kosztu wynosi około 0.37, w 100. epoce jest to mniej więcej 0.35, natomiast 300. i 500. epoka przypadają na wspomniany powyżej przedział. Zatem pomiędzy tymi epokami nie widać dużych różnic w wartościach funkcji kosztu. Jak się to przełoży na wyniki? O tym się przekonamy, wykonując procedurę odszumiania. Nie przedstawiam tutaj stricte animacji kolejnych kroków, ale co 100 epok zapisuję aktualny stan operacji. Prezentuję kolejne etapy (zaczynając od kroku $t = 1000$) na rysunkach poniżej.

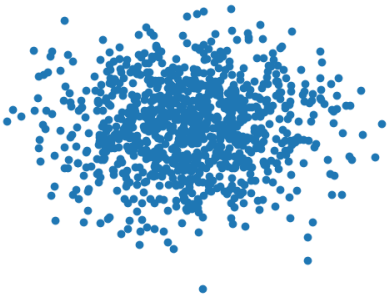


(a)

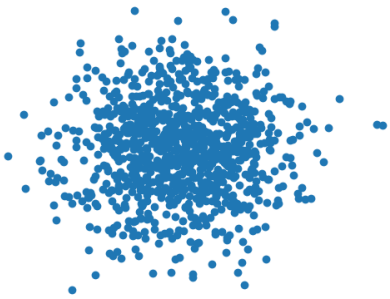


(b)

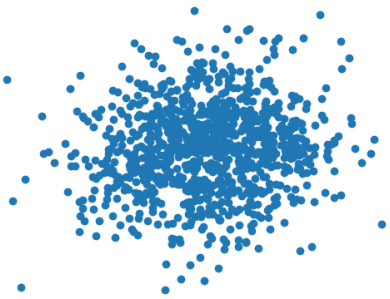
(c)



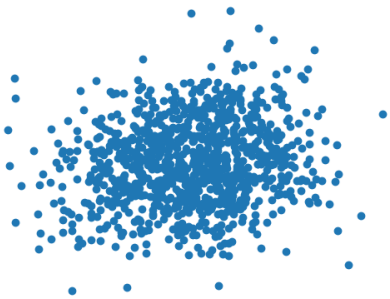
(d)



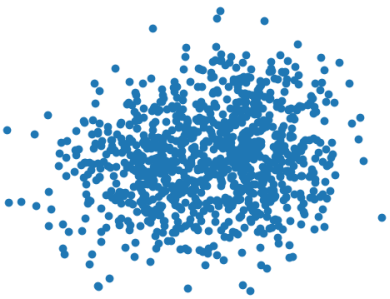
(e)



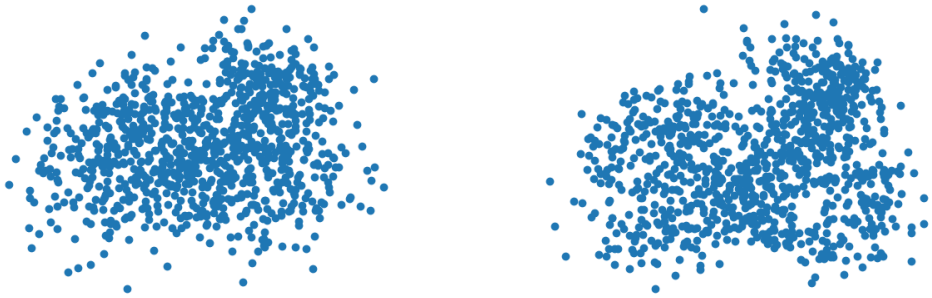
(f)



(g)



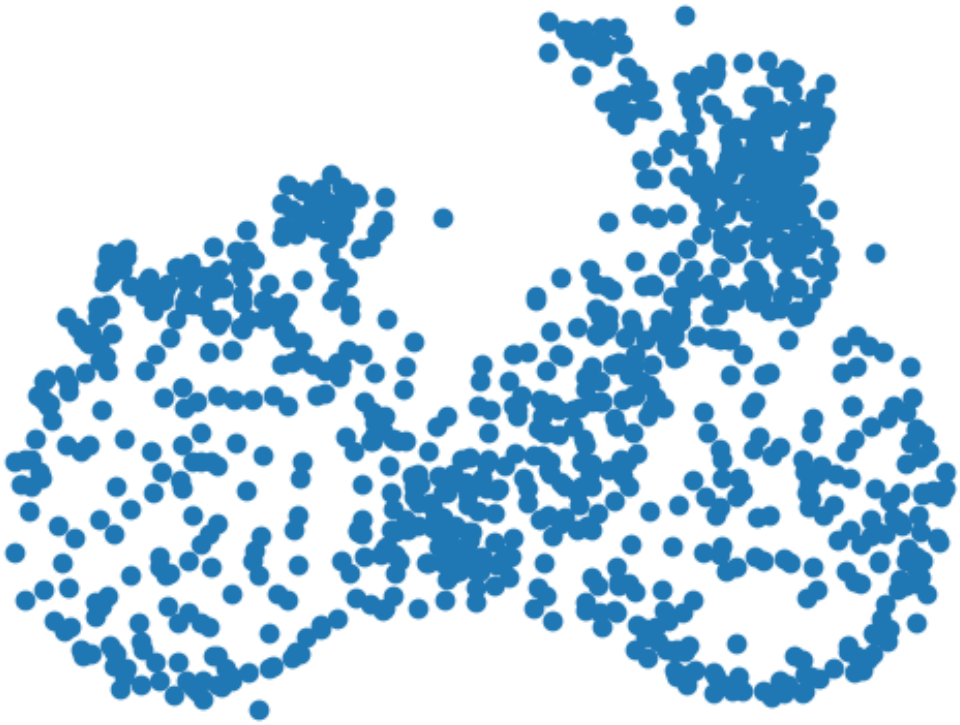
(h)



(i) (j)

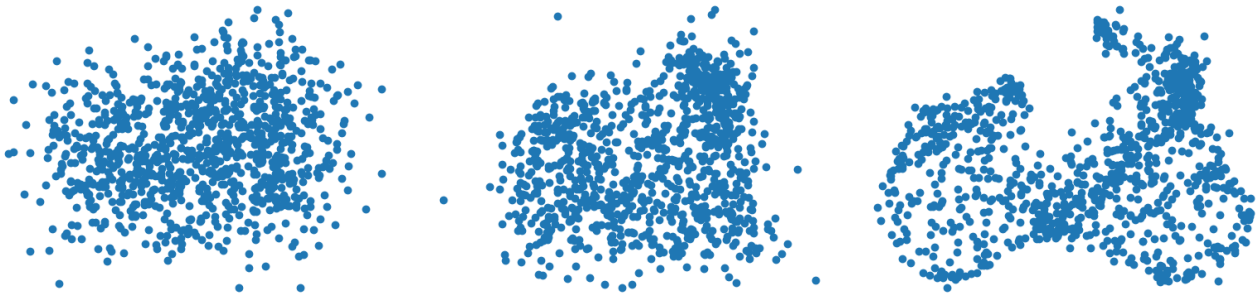
Rysunek 5: Rozkład próbek w odszumianiu dla kroku czasowego: (a) $t = 1000$, (b) $t = 900$, ... (j) $t = 100$

Pierwszy rysunek 5(a) przedstawia stan początkowy, jeszcze przed wykonaniem kroku algorytmu odszumiania - jest to wygenerowana losowo chmura 1000 punktów. W kolejnych etapach rozkład punktów się niewiele zmienia i tak naprawdę dopiero pod koniec widać coś, co kształtem jest bliższe rowerowi. Warto podkreślić, że rysunek 5(j) przedstawia efekt dla kroku czasowego $t = 100$, a nie finalny. Finalny efekt z kolei prezentuje się następująco:



Rysunek 6: Finalny efekt odszumiania losowej próbki przez wytrenowany model

Według mnie efekt końcowy wygląda dobrze. Wyraźnie zarysowany jest kształt roweru, mimo że oczywiście nie jest on idealny. Zobaczmy, czy wcześniejsze wersje modelu równie dobrze radzą sobie z tym zadaniem. Czy długi trening modelu przez 500 epok był potrzebny?



(a) (b) (c)

Rysunek 7: Finalny efekt odszumiania dla modeli zapisanych po: (a) 25 epokach, (b) 100 epokach, (c) 300 epokach

Odpowiedź na postawione wyżej pytanie jest raczej twierdząca. W moim przypadku akurat nie widać, żeby po kilkudziesięciu krokach efekt generacji dawał rozsądny rezultat. Nawet po 100 epokach wynik raczej kojarzy się dalej z losową chmurą punktów niż z rowerem. Dla 300. epoki jest już dużo lepiej - tutaj rysunek przypomina kształtem rower, efekt jest niewiele gorszej jakości niż dla 500 epok.
Zestawiając to z wcześniejszymi obserwacjami co do wartości funkcji kosztu, spadek jakości **można uznać** za proporcjonalny do *różnic* w wartościach straty, ale na pewno nie do bezwzględnych wartości. Bazując na znalezionych przeze mnie informacjach, dzieje się tak dlatego, że optymalna wartość funkcji kosztu zwykle nie jest zerowa i jest nieznana *a priori*. Ponadto wartość straty może pozostawać w wąskim zakresie i fluktuować (co się istotnie stało w moim przypadku), a jakość generowanych obrazów wtedy może nadal rosnąć.
W tym miejscu można się zastanowić, co dałoby trenowanie przez 1000 epok lub potencjalnie więcej. Wydaje mi się, że efekt generacji byłby jeszcze lepszy, dokładniejszy, ale odbyłoby się to niewspółmiernym kosztem (przynajmniej w moim przypadku) dodatkowych godzin treningu. Dlatego uważam, że decyzja o przyjęciu 500 epok treningu była dobrym wyborem i rozsądnym kompromisem.