Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

LAB 05 - WPROWADZENIE DO GŁĘBOKIEGO UCZENIA - SIECI GAN

Karol Działowski

nr albumu: 39259
przedmiot: Uczenie Maszynowe 2

Szczecin, 4 lutego 2021

Spis treści

1 Cel laboratorium

Celem laboratorium było zapoznanie się z funkcjonalnością pakietów *TensorFlow* i *Keras*. W tym celu należało zaimplementować sieć GAN.

Zadaniem głównym było zbudowanie modelu sieci GAN o dowolnej (autorskiej) architekturze, która będzie służyć do generowania obrazów twarzy. Należało przeprowadzić eksperymenty, badając wpływ rozmiaru obrazu wejściowego na jakość wynikowych obrazów oraz porównać wpływ kroku uczenia na jakość modelu.

Do uczenia modelu wykorzystano bazę Labeled Faces in the Wild [1]. Wybrano opcję z obrazami wyrównanymi (*deepfunneled*) i nauczono sieć na 530 obrazach George'a W. Busha.

2 Zaproponowana sieć

Poniżej przedstawiono zaproponowaną sieć dla obrazów o rozmiarze 64×64. Przy innych rozmiarach obrazu wejściowego analogicznie dodawano lub odejmowano warstwy dekonwolucyjne w generatorze.

2.1 Generator

- Warstwa gęsta o $8 \cdot 8 \cdot 256$ neuronach z wejściem 100 elementów
- Warstwa BatchNormalization
- Warstwa LeakyReLU
- Przekształcenie danych do rozmiaru $(8 \times 8 \times 256)$
- Warstwa dekonwolucyjna o głębokości 128 z rozmiarem kernela 5×5 o kroku 1
- Warstwa BatchNormalization
- Warstwa LeakyReLU
- Warstwa dekonwolucyjna o głębokości 64 z rozmiarem kernela 5×5 o kroku 2
- Warstwa BatchNormalization
- Warstwa LeakyReLU
- Warstwa dekonwolucyjna o głębokości 32 z rozmiarem kernela 5×5 o kroku 2
- Warstwa BatchNormalization
- Warstwa LeakyReLU
- Warstwa dekonwolucyjna o głębokości 1 z rozmiarem kernela 5×5 o kroku 2 z aktywacją tangens hiperboliczny

2.2 Dyskryminator

- Wejście o rozmiarze obrazka $64 \times 64 \times 1$
- Warstwa konwolucyjna o głębokości 64 z rozmiarem kernela 5×5 krokiem 2
- Warstwa LeakyReLU
- Warstwa Dropout 30%
- Warstwa konwolucyjna o głębokości 128 z rozmiarem kernela 5×5 krokiem 2
- Warstwa LeakyReLU
- Warstwa Dropout 30%
- Spłaszczenie danych do wektora i połączenie do neurona wynikowego

3 Przygotowanie zbioru uczącego

Zbiór uczący poddano przetwarzaniu wstępnemu. Obrazy poddano skalowaniu do rozmiarów 64×64, 32×32, 128×128 w zależności od wybranej architektury. Przekształcono obrazy do odcieni szarości, znormalizowano względem zera do wartości $(-1, 1)$ oraz dodano zdjęcia odbite w poziomie aby powiększyć zbiór uczący.

4 Wpływ rozmiaru próby uczącej na model

Przetestowano model ucząc go na pełnym zbiorze uczącym liczącym 530 obrazów z parametrem learning rate = 0.0002, $\beta_1 = 0.5$ oraz z analogicznymi parametrami na 365 (połowie zbioru uczącego) obrazach. Poniżej przedstawiono porównanie wyników generatora po 1000 epokach uczenia.

Pełny zbiór uczący (530 zdjęć) Połowa zbioru uczącego (265 zdjęć

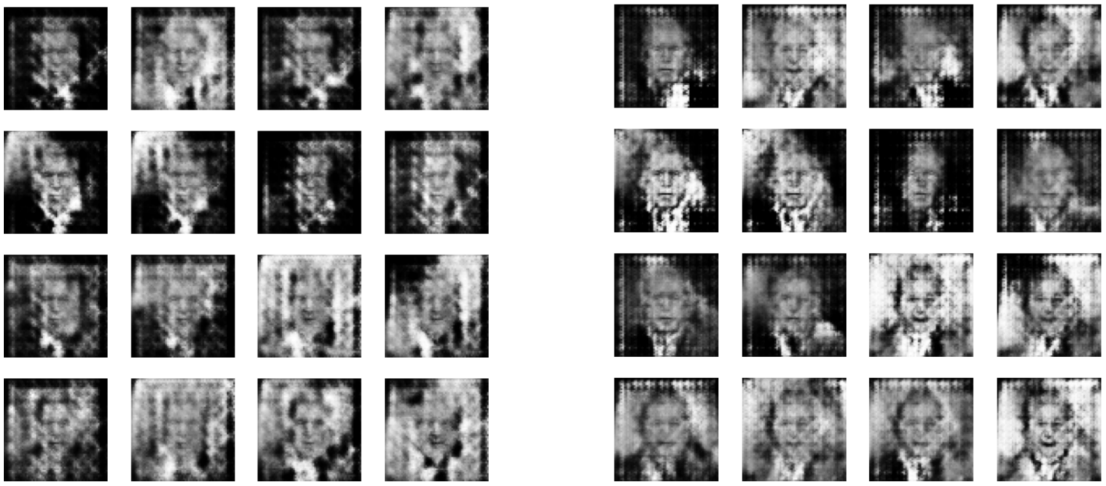
Rysunek 1: Porównanie wpływu jakości modelu na rozmiar zbioru uczącego uczonego na 1000 epokach

Na podstawie rysunku 1 widać, że rozmiar zbioru uczącego ma istotny wpływ na jakość wygenerowanych przez model obrazów wyjściowych.

5 Przebieg uczenia modelu

Na poniższych migawkach przedstawiono kolejne iteracje obrazów wyjściowych modelu w wybranych epokach. Badanie przeprowadzono na rozmiarze obrazów wejściowych 64×64 oraz przy learning rate = 0.0002.

Na podstawie rysunku 2 można stwierdzić, że liczba epok ma kluczowe znaczenie przy jakości modelu. Po 400 epokach model daje zadowalające wyniki, przypominające twarz człowieka. Model nie jest stabilny i od pewnego momentu przeprowadzanie dalszego uczenia nie poprawia rezultatów, wprowadzając zakłócenia. Wynika to z małej liczby danych wejściowych.



100 epok 200 epok

400 epok 600 epok

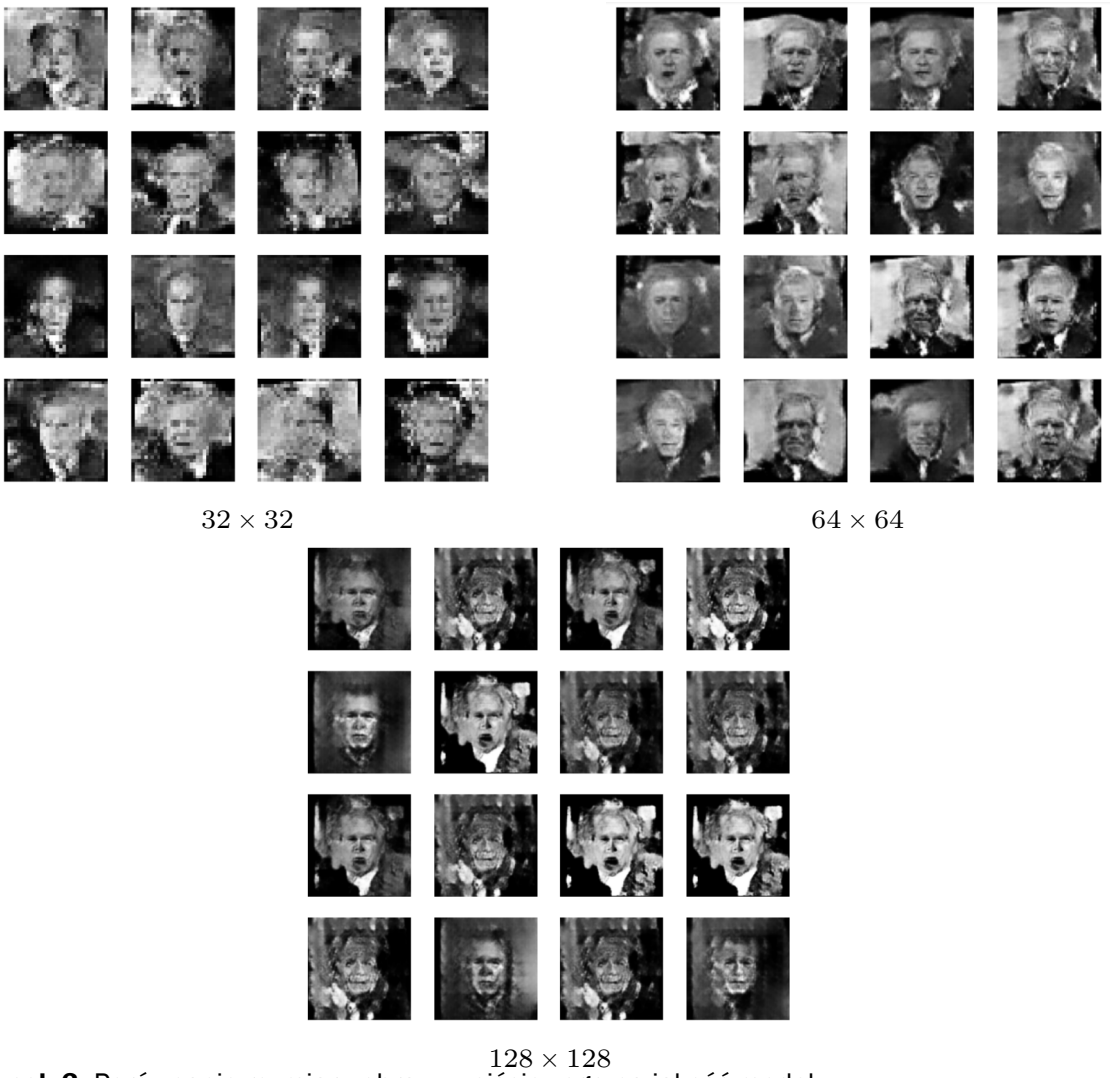
800 epok 1000 epok

Rysunek 2: Porównanie wpływu liczby epok na jakość modelu

6 Rozmiar obrazu wejściowe a jakość wyników

Kolejny eksperyment polegał na porównaniu jakości otrzymywanych wyników porównując architektury dla różnych rozmiarach obrazów wejściowych. Porównane rozmiary to 64×64, 32×32 oraz 128×128. Porównano czasy trwania pojedynczej epoki dla danej architektury.

Najlepsze wyniki po 1000 epokach uzyskiwano dla architektury o rozmiarze obrazu 64×64. Dla rozmiaru 128×128 występują zniekształcenia twarzy, szczególnie ust, oraz mocno wyróżniające się artefakty tła.



Rysunek 3: Porównanie rozmiaru obrazu wejściowego na jakość modelu

Rozmiar	Czas [s]
32×32	0.1288
64×64	0.2000
128×128	1.046

Tabela 1: Czas trwania jednej epoki dla rozmiaru obrazu

7 Wpływ współczynnika uczenia na jakoś modelu

Nauczono model porównując parametry learning rate ze zbioru $\{0.0002, 0.0005, 0.002, 0.02\}$. Uzyskane wyniki po 1000 epokach przedstawiono na rysunku 4. Subiektywnie najlepszy współczynnik uczenia to 0.0002. Wyniki jednak nie różnią się znacząco pomiędzy zbadanymi współczynnikami uczenia pomijając lr=0.02, dla którego model nie nauczył się generować kształtu twarzy.

Rysunek 4: Porównanie współczynnika uczenia na jakość modelu

8 Wnioski

Podczas laboratorium zbudowano model DCGAN, który nauczono na zdjęciach twarzy jednej osoby. Uzyskane wyniki nie są zadowalające i mają niską jakość. Można w wygenerowanych obrazach zauważyć cechy ludzkiej twarzy, podobnej do twarzy ze zbioru uczącego.

Aby stworzyć lepszy model należałoby nauczyć go większym zbiorem uczącym.

A Implementacja modelu

Kod źródłowy 1: Implementacja modelu
Źródło: Opracowanie własne

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(8*8*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((8, 8, 256)))
    assert model.output_shape == (None, 8, 8, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 8, 8, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 16, 16, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 32, 32, 32)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    # model.add(layers.Conv2DTranspose(16, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    # assert model.output_shape == (None, 64, 64, 16)
    # model.add(layers.BatchNormalization())
    # model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, BOX_SIZE, BOX_SIZE, 1)

    return model

def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                     input_shape=[BOX_SIZE, BOX_SIZE, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

```

```
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model

discriminator = make_discriminator_model()
generator = make_generator_model()
```