Metody rozpoznawania obrazów
Zadanie 1 – raport

Mateusz Kocot

23 października 2021

Spis treści

1 Wybór frameworka

Wybrałem opcję pierwszą, tj. TensorFlow. Kiedyś miałem z nim styczność, co prawda na krótko, ale wszystkiego nie zapomniałem. Poza tym, jakiś czas temu zakupiłem książkę o uczeniu maszynowym, w której wykorzystywany jest właśnie TensorFlow, więc liczę na to, że doświadczenie z zajęć ułatwi mi zrozumienie tej książki.

2 Przygotowanie zbioru danych

2.1 Wybór klas

Wybrałem klasy zalecane przez prowadzącego, tj. sukienki (klasa nr 3) i sandały (klasa nr 5).

2.2 Selekcja właściwych zdjęć

Binarną maskę utworzyłem z wykorzystaniem funkcji `tf.math.logical_or`, która jako argumenty przyjęła maski obu klas powstałe przy użyciu operatora porównania:

```
mask = tf.math.logical_or(y == 3, y == 5)
```

Zaaplikowanie maski było już proste:

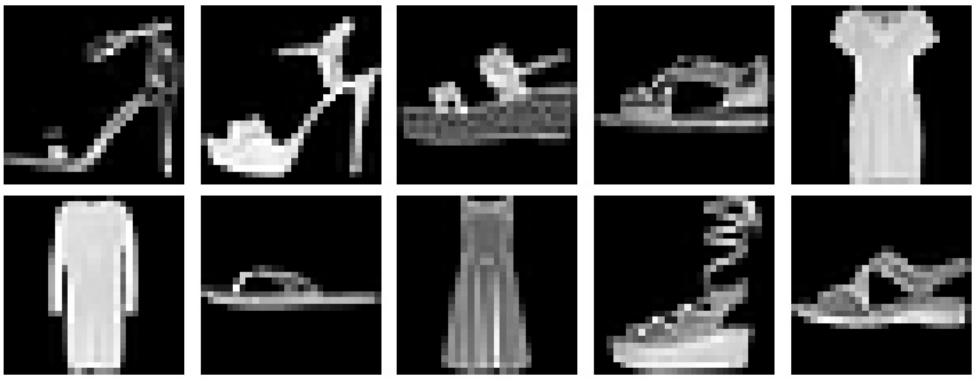
```
x, y = x[mask], y[mask]
```

Poza nałożeniem maski, zdecydowałem, że już w tym miejscu zamienię etykiety klas z 3 na 0 i z 5 na 1:

```
y = tf.where(y == 3, 0, 1)
```

Powód tej decyzji opiszę w kolejnej sekcji.

Po wykonaniu selekcji, podejrzałem pierwsze 10 elementów ze zbioru (rys. 1) i rzeczywiście były to same sukienki i sandały.



Rys. 1: Pierwsze 10 elementów ze zbioru po wykonaniu selekcji.

3 Model decyzyjny - regresja logistyczna

3.1 Konstrukcja modelu w TensorFlow

Najprostszym modelem w TensorFlow jest `tf.keras.Sequential`. Pozwala on na proste dodanie warstw sieci neuronowych, natomiast wydaje się być zbyt ubogi, gdy wymagane jest przetworzenie danych wejściowych jeszcze przed warstwą sieci. Dlatego skorzystałem z klasy `tf.keras.Model`. Jednym z dwóch sposobów na korzystanie z niej jest odziedziczenie jej w nowej klasie. Następnie należy nadpisać `__init__` oraz `call`. W skrócie, metoda `__init__` powinna wywołać konstruktor klasy nadrzędnej oraz stworzyć instancje potrzebnych warstw przetwarzania, a metoda `call` powinna wykorzystać te warstwy do przetworzenia danych wejściowych. Dzięki dziedziczeniu z `tf.keras.Model` otrzymujemy „za darmo” metody takie jak:

- `compile` - określenie metryki i funkcji straty,

- `fit` - wytrenowanie modelu,
- `predict` - predykcja.

3.2 Wstępna obróbka danych

Wstępną obróbkę zaimplementowano w nadpisywanej funkcji `call(self, inputs)`, gdzie `inputs` oznacza dane wejściowe, czyli w naszym przypadku - wektor obrazów. Najpierw obrazy zamieniane są na wektory:

```
x = tf.reshape(inputs, shape=[-1, 28 * 28])
```

Następnie wartości w wektorach są normalizowane. W tym celu po prostu podzieliłem wszystkie wartości przez 255 uprzednio zamieniając ich typ na zmiennopozycyjny:

```
x = tf.cast(x, dtype=float32) / 255.
```

Niestety, nie znalazłem ładnego sposobu na zmianę w tym miejscu etykiet na 0 i 1. Co prawda, mógłbym nadpisać metodę `fit` i tam dokonać zamiany, natomiast nie wydaje się to być dobrym pomysłem. Z tego powodu ten krok przesunąłem do poprzedniej sekcji.

Kilka printów umożliwiło sprawdzenie, że obróbka wykonywana jest poprawnie. Sprawdziłem też, czy TensorFlow wspiera debugowanie na żywo. W tym celu przesiadłem się z domyślnego dla mnie Jupyter Notebook'a na JupyterLab'a, który w przeciwieństwie do poprzedniego wspiera ustawianie break pointów i zatrzymywanie wykonywania programu w tych punktach. Z pomocą tego mechanizmu sprawdziłem zawartości tensorów w odpowiednich miejscach i o ile wymiary mogłem zobaczyć bez problemu, to miałem problem z zobaczeniem zawartości; nie dlatego, że tensory były bardzo duże, ale dlatego, że z jakiegoś powodu nie było możliwości podejrzenia zawartości atrybutu `numpy` klasy `Tensor`, w którym przechowywane są wartości liczbowe tensora. Być może korzystając z bardziej zaawansowanego IDE jak Pycharm nie byłoby takiego problemu.

3.3 Część decyzyjna

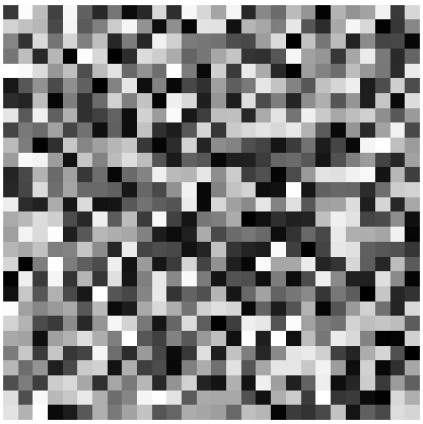
Model regresji linowej zaimplementowałem zgodnie ze wskazówkami z treści zadania. W metodzie `__init__` stworzyłem odpowiednią instancję gęsto połączonej warstwy sieci neuronowej:

```
self.dense = tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)
```

Tak stworzony obiekt wystarczyło już wykorzystać w metodzie `call` przy okazji kończąc przetwarzanie:

```
return self.dense(x)
```

Wyniki otrzymane przed wytrenowaniem matematycznie mają sens, tj. mieszczą się między 0 i 1. Pobrałem także domyślne wagi warstwy gęstej (rys. 2) i rzeczywiście po ich zobrazowaniu otrzymałem losowy szum.



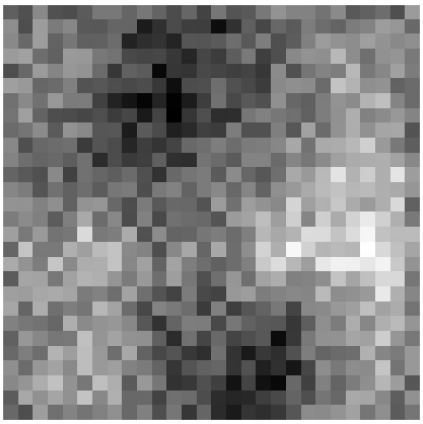
Rys. 2: Domyślne wagi warstwy gęstej.

3.4 Trening

Klasa `tf.keras.Model` pozwala na łatwe wybranie funkcji straty oraz metryki wykorzystywanej do oceny jakości dopasowania:

```
model.compile(optimizer='SGD', loss='binary_crossentropy')
```

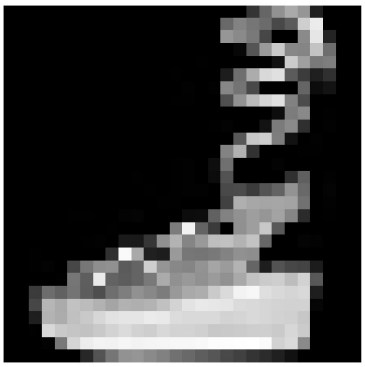
Strojenie danych przeprowadziłem w 15 epokach. Po strojeniu wagi modelu nabrały sensu (rys. 3). Ciemniejsze pola oznaczają małe (ujemne) wagi. Oprócz tego, odpowiadają one za miejsca, w których przeważnie znajdują się części sukienek (które są klasą 0), ale nie sandałów. Analogiczna sytuacja dotyczy pól jasnych, tj. wag dużych. Model nauczył się więc znajdować miejsca na obrazkach, a właściwie indeksy wektorów odpowiadające jednej z klas i w ten sposób, gdy większość elementów wektora z wagami małymi jest aktywnych, wartość funkcji liniowej w regresji logistycznej jest ujemna, co po przejściu przez sigmax daje prawdopodobieństwo bliskie zeru, czyli właściwe dla klasy 0. Odwrotnie działa to dla sandałów



Rys. 3: Wagi warstwy gęstej po wytrenowaniu modelu.

Wytrenowany model w zdecydowanej większości przypadków podejmuje poprawne decyzje.

Tyczy się to zbioru treningowego, jak i testowego. Problemem są jednak ubrania odstające od reszty, np. jeden z butów pokazany na rys. 4. Przez to, że jest on wysoki, podeszwa znajduje się w „czarnej strefie", czyli miejscu, które model uznał za wyróżniające dla sukienek. Dołożenie kilkunastu epok więcej eliminuje problem w przypadku tego buta.



Rys. 4: Jeden z trudniejszych do sklasyfikowania sandałów.