Projekt kompetencyjny

Konwolucyjne sieci neuronowe

	Autor	**Album**
	Kateryna Tsarova	226451

	Prowadzący
	Piotr Duch, dr inż.

Ocena prowadzącego:

Łódź, 19.09.2020 r.

SPIS TREŚCI

1. WSTĘP

1.1 Cel projektu

Celem mojego projektu jest stworzenie konwolucyjnej sieci neuronowej, umożliwiającej klasyfikację sześciu typów ręcznie rysowanych obiektów, i wykorzystanie jej w aplikacji do rozpoznawania rysunków namalowanych przez użytkownika.

1.2 Czym jest uczenie głębokie

Uczenie głębokie (zwane też uczeniem hierarchicznym) jest klasą algorytmów uczenia maszynowego i strategii uczących takich, że:

- Rozwijają strukturę hierarchiczną i reprezentację podstawowych i wtórnych cech, reprezentujących różne poziomy abstrakcji.
- Wykorzystują kaskadę wielu warstw neuronów (lub innych jednostek obliczeniowych) różnych rodzajów w celu stopniowej ekstrakcji cech i ich transformacji w celu osiągnięcia hierarchii cech wtórnych/pochodnych, które prowadzą do lepszych wyników zbudowanych na ich podstawie sieci neuronowych. W ten sposób próbują określić bardziej skomplikowane cechy na podstawie prostszych cech.
- Stosują różne strategie uczenia nadzorowanego i nienadzorowanego dla różnych warstw.
- Stopniowo rozwijają i aktualizują strukturę sieci dopóki występuje znacząca poprawa wyników działania sieci.

Głębokie sieci neuronowe znajdują szerokie zastosowanie w rozpoznawaniu obrazów i kształtów. Przykładowe aplikacje obejmują rozpoznawanie twarzy, analizę obrazów w medycynie, klasyfikację pisma czy detekcję obiektów otoczenia. Specjalnym rodzajem sieci neuronowej, który wyjątkowo dobrze radzi sobie z przetwarzaniem obrazu, są konwolucyjne sieci neuronowe. Sieci konwolucyjne poprzez trening są w stanie nauczyć się, jakie cechy szczególne obrazu pomagają w jego klasyfikacji. Ich przewagą nad standardowymi sieciami głębokimi jest większa skuteczność w wykrywaniu zawiłych zależności w obrazach.

Przykłady wykorzystania sieci neuronowych w codziennym życiu:

- wykrywanie i rozpoznawanie twarzy na zdjęciach w smartfonie,
- rozpoznawanie komend głosowych przez wirtualnego asystenta,
- autonomiczne samochody [1].

1.3 Poznane techniki i narzędzia

Do napisania sieci neuronowej użyłam języka Python, przy napisaniu korzystałam z bibliotek Tensorflow, Keras, tf2onnx i z bazy danych «Qiuck, Draw!» [2].

Do napisania aplikacji użyłam języka C#, sama aplikacji była stworzona w środowisku Unity. Przy napisaniu korzystałam z bibliotek Unity.Barracuda, System.Collections.Generic, UnityEngine, System.IO, UnityEngine.UI.

1.4 Przebieg pracy

Projekt polega na zaimplementowaniu do aplikacji wytrenowanego modelu konwolucyjnej sieci neuronowej, która umożliwia rozpoznawanie obiektów narysowanych przez użytkownika.

Przed implementacją sieci neuronowej napisałam aplikację do malowania w środowisku Unity.

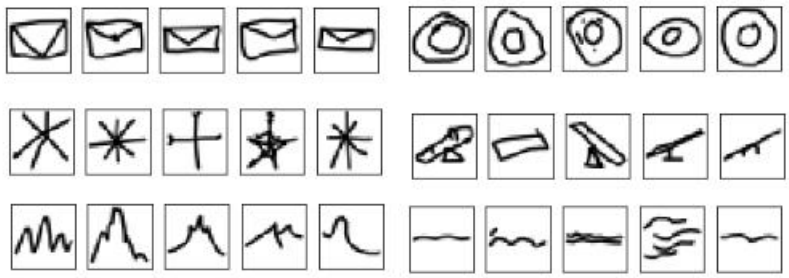
2. DZIAŁANIE KONWOLUCYJNYCH SIECI NEURONOWYCH

2.1 Przygotowanie danych wejściowych

Do wytrenowania sieci neuronowej była wykorzystana baza danych «Quick, draw!», obejmująca zbiór 50 milionów rysunków w 345 kategoriach, nadesłanych przez graczy gry «Quick, Draw!» [3]. Z tego zbioru były wybrane następne kategorię:

- envelope (koperta),
- donut (pączek),
- snowflake (śnieżynka),
- see-saw (huśtawka),
- mountain (góra),
- ocean.

Wszystkie obrazki są skalowane do rozmiaru 28x28 i są reprezentowane w trybie monochromatycznym. Na rys. 1 zaprezentowane przykłady obrazków, wykorzystanych do wytrenowania sieci.



Rys. 1: Przykłady obrazków z każdej z sześciu kategorii

Do uczenia sieci neuronowej były wybrane 60 000 obrazków z każdej kategorii, rozdzielonych na dwa zbiory: treningowy (90% obrazków) i testowy (10% obrazków).

2.2 Model CNN

Model konwolucyjnej sieci neuronowej był stworzony w *Tensorflow* (open-source'owy framework stworzony przez Google'a do obliczeń numerycznych). Oferuje on zestaw narzędzi służących do projektowania, trenowania oraz douczania sieci neuronowych [4].

Każda warstwa modelu ma dokładnie jeden tensor wejściowy i jeden tensor wyjściowy - jest to model sekwencyjny (rys. 2). Sieć neuronowa składa się z kilku rodzajów warstw, oferowanych przez *Tensorflow*:

- *Conv2D* – konwolucyjnej, po polsku splotu (od funkcji splotu),
- *MaxPooling* – zmieniającej rozdzielczość obrazka,
- *Dense* – warstwa, dla której wszystkie jednostki poprzedniej warstwy są połączone ze wszystkimi w następnej,
- *Dropout* – przepuszczających tylko fragment danych (aby zapobiec przeuczeniu sieci),
- *Flatten* – spłaszczającej macierze do wektorów.

Layer (type)	Output Shape	Param #
conv2d_97 (Conv2D)	(None, 26, 26, 256)	2560
max_pooling2d_81 (MaxPooling	(None, 13, 13, 256)	0
dropout_101 (Dropout)	(None, 13, 13, 256)	0
conv2d_98 (Conv2D)	(None, 11, 11, 128)	295040
max_pooling2d_82 (MaxPooling	(None, 5, 5, 128)	0
dropout_102 (Dropout)	(None, 5, 5, 128)	0
conv2d_99 (Conv2D)	(None, 3, 3, 64)	73792
max_pooling2d_83 (MaxPooling	(None, 1, 1, 64)	0
dropout_103 (Dropout)	(None, 1, 1, 64)	0
flatten_33 (Flatten)	(None, 64)	0
dense_99 (Dense)	(None, 512)	33280
dense_100 (Dense)	(None, 128)	65664
dense_101 (Dense)	(None, 6)	774

Total params: 471,110
Trainable params: 471,110
Non-trainable params: 0

Rys. 2: Architektura sieci neuronowej

Aby ocenić wydajność sieci neuronowej, do wytrenowania modelu była wykorzystana kategoryczna entropia krzyżowa (*categorical crossentropy*), która jest optymalnym wyborem do klasyfikacji wieloklasowej. Był także wykorzystany algorytm optymalizacji ADAM (*ang. adaptive moment estimation*).

Liczba neuronów w trzech warstwach konwolucyjnych jest równa odpowiednio 256, 128 i 64, liczba neuronów w trzech warstwach *Dense* – 512, 128, 6. Rozmiar filtra w warstwach konwolucyjnych jest 3x3, rozmiar filtra warstwy pooling - 2x2. Warstwy *Conv2D* i *Dense* korzystają z funkcji optymizacji ReLU (*ang. rectified linear unit*) - jest to najbardziej znana i najczęściej stosowana funkcja w przypadku warstw ukrytych. Warstwa wyjściowa korzysta z funkcji optymizacji softmax, dzięki czemu można otrzymać na wyjściu tablicę składającą się z 6 wartości sumujących się do 1.

2.3 Wyniki nauczania

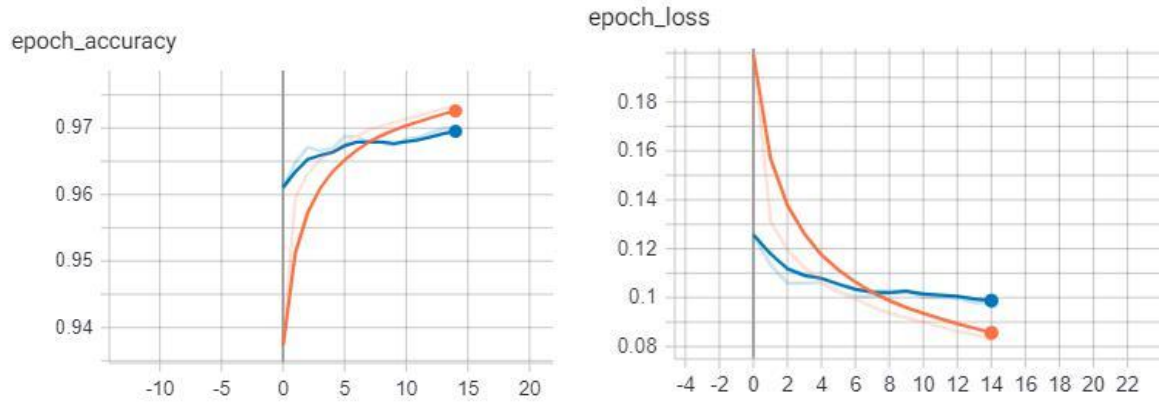
Moim celem było uzyskanie znacznie lepszej dokładności niż przypadkowe zgadywanie, dokładność którego jest tylko 16.67%. Wytrenowana sieć osiągnęła zamierzony cel: końcowa dokładność sieci neuronowej jest równa 97%.

Do trenowania modelu były wybrane takie parametry:

- *epoch* – 15
- *batch_size* – 200

Epoch odpowiada za liczbę epok użytych do trenowania modelu – jedna epoka oznacza przejście całego zbioru przez sieć. *Batch_size* odpowiada za zdefiniowanie ile rekordów (obserwacji) przechodzi na raz podczas pojedynczego przebiegu zanim nastąpi pierwsza aktualizacja wag parametrów.

Wyniki nauczania można zaobserwować na rys. 3.



Rys. 3: Wykresy zależności dokładności/straty od liczby epok
(czerwony - trening, niebieski - validacja)

Wykres na lewo prezentuje jak wraz z kolejnymi epokami wygląda metryka *accuracy* w podziale na zbiorze treningowym i testowym. Widoczne, że na obu wykresach jest wzrost, zatem teoretycznie można byłoby zwiększyć liczbę epok.

Drugi wykres przedstawia stratę w podziale na zbiór uczący oraz testowy. Widoczne, że strata dla obu spada, co jeszcze raz pokazuje, że można byłoby zwiększyć liczbę epok dla tej architektury.

Zwiększenie liczby epok zwykle zapewnia wyższą dokładność, ale czas obliczeń również znacznie się wydłuża. Do osiągnięcia zamierzonego celu wystarczy 15 epok, ponieważ poza tym nie ma prawie żadnego wzrostu dokładności sieci, ale jej trening zajmuje o kilka razy więcej czasu.

Do polepszenia dokładności sieci zostały wypróbowane dwie metody: powiększenie liczby neuronów w warstwach konwolucyjnych i zwiększenie liczby obrazków testowych.

Powiększenie liczby neuronów w dwóch warstwach konwolucyjnych nieznacznie zwiększyło dokładność kosztem znacznie dłuższego czasu obliczeniowego. Jako kompromis między dokładnością a zużyciem czasu była dodana do modelu trzecia warstwa konwolucyjna o liczbie neuronów 64 (dodanie trzeciej warstwy nie zmieniło dokładności, ale skróciło czas treningu o połowę).

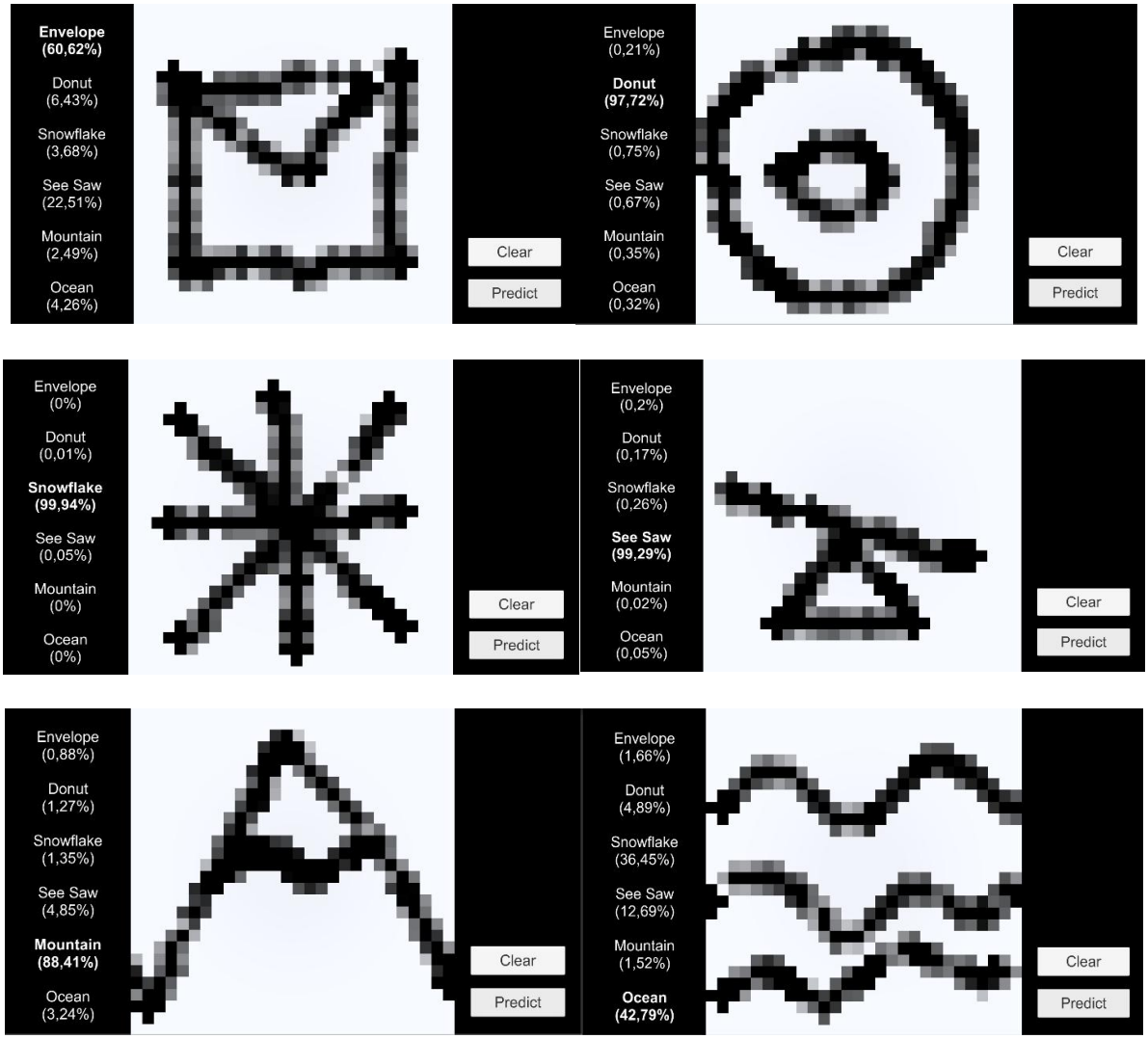
Zwiększenie liczby obrazków testowych z 20 000 do 60 000 zwiększyło dokładność sieci od 94.9% do 97%, ale czas treningu był również zwiększony. Teoretycznie można zwiększyć liczbę obrazków testowych do 100 000 i tym samym zwiększyć dokładność sieci do 98-99% (kosztem zwiększenia czasu treningu pięciokrotnie), ale nie uważam, że to jest koniecznie dla tego projektu: istniejąca sieć skutecznie spełnia swoje zadanie i potrafi efektywnie rozpoznać rysunki użytkownika (co zostało opisane w szczegółach w rozdziale 2.4).

Podczas treningu sieci miałam problem z przetrenowaniem modelu (*ang. overfitting*), który został rozwiązany dodaniem do modeli trzech warstw *Dropout* z parametrem *noise_shape* równym 0.2 (warstwy *Dropout* zostały umieszczone po każdej z trzech warstw konwolucyjnych). Zwiększenie liczby obrazków treningowych także pomogło w rozwiązaniu tego problemu.

Ustawienia	Czas treningu (sec)	Dokładność (%)
- dwie warstwy *Conv2D* (liczba neuronów - 30, 15) - trzy warstwy *Dense (*liczba neuronów - 128, 50) - liczba obrazków testowych - 20 000 - liczba epok - 15	501	92.65
- dwie warstwy *Conv2D* (liczba neuronów - 764, 512) - trzy warstwy *Dense (*liczba neuronów - 764, 512) - trzy warstwy Dropout - liczba obrazków testowych - 20 000 - liczba epok - 15	21 124	95
- trzy warstwy *Conv2D* (liczba neuronów - 256, 128, 64) - trzy warstwy *Dense (*liczba neuronów - 512, 128) - trzy warstwy Dropout - liczba obrazków testowych - 20 000 - liczba epok - 15	9420	94.9
- trzy warstwy *Conv2D* (liczba neuronów - 256, 128, 64) - trzy warstwy *Dense (*liczba neuronów - 512, 128) - trzy warstwy Dropout - liczba obrazków testowych - 60 000 - liczba epok - 15	27 626	97

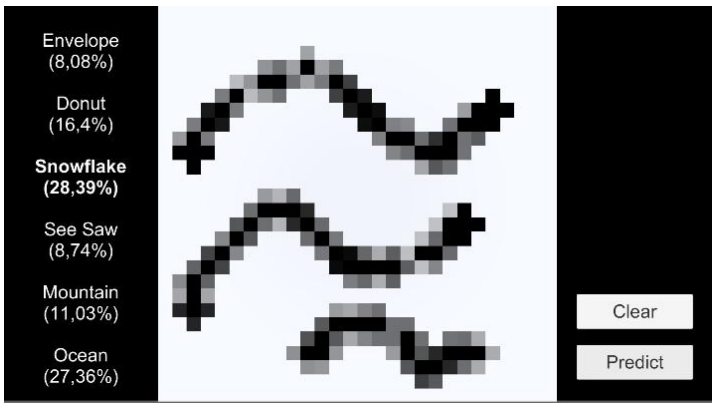
2.4 Działanie aplikacji

Testowanie sieci neuronowej było wykonane w aplikacji do malowania, stworzonej w Unity. Rozmiar planszy do malowania (28x28) i typ pędzla zostały specjalnie wybrane dla ułatwienia klasyfikacji rysunków użytkowników. Przykład działania aplikacji na rys.4.



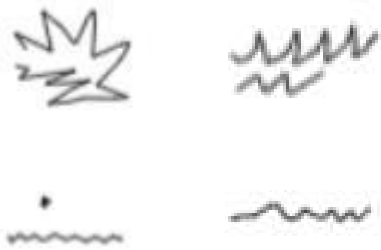
Rys. 4: Rozpoznawanie ręcznie namalowanych rysunków

Z wyników działania aplikacji można zauważyć, że importowana w środowisko Unity sieć neuronowa skutecznie rozpozna namalowane użytkownikiem rysunki, ale w szczególnych wypadkach mogą być niepowodzenia w klasyfikacji figur (rys. 5).



Rys. 5: Sieć niepoprawnie rozpoznała ocean

Możliwym powodem niepoprawnej klasyfikacji sieci jest podobieństwo obrazków w bazie danych «snowflake» i «ocean» (rys. 6).



Rys. 6: Podobieństwo obrazków dwóch klas
(po lewej stronie - klasa «snowflake», po prawej - «ocean»)

Mimo to, w przeważającej większości wypadków wytrenowana sieć neuronowa spełnia swoje zadanie i poprawnie rozpozna rysunki użytkownika.

3. PODSUMOWANIE

Dzięki realizacji tego projektu zdobyłam szeroki zakres wiedzy i umiejętności w dziedzinach takich jak:

- pisanie w języku C# i Python,
- tworzenie aplikacji w środowisku Unity,
- zapoznanie z uczeniem maszynowym,
- zapoznanie z uczeniem głębokim,
- zapoznanie z działaniem konwolucyjnych sieci neuronowych,
- rozwiązywanie problemów z dziedziny informatyki,
- organizacja pracy.

W przyszłości konwolucyjne sieci neuronowe mogą być zaimplementowane do gier (np. Do Pacmana)

4. ŹRÓDŁA

[1] https://bfirst.tech/konwolucyjne-sieci-neuronowe/

[2] https://github.com/googlecreativelab/quickdraw-dataset

[3] https://quickdraw.withgoogle.com/

[4] https://geek.justjoin.it/zalety-wady-korzystania-tensorflow-srodowisku-produkcyjnym