Podstawy reprezentacji i analizy danych

Klasyfikacja cyfr pisanych odręcznie (baza MNIST)

Zespół nr. 8: Andrzej Czechowski, Franciszek Wysocki, Bartosz Zakrzewski

31.01.2021

Spis treści

Opis problemu

MNIST (Modified National Institute of Standards and Technology) to zbiór danych zawierający zdjęcia odręcznie pisanych cyfr i etykiety oznaczające, jaka to liczba.

Baza danych MNIST zawiera 60 000 elementów zbioru uczącego i 10 000 zbioru testowego (86% i 14%).

Istnieje wiele metod uczenia maszynowego zastosowanych do rozpoznawania tych zdjęć - na Wikipedii możemy znaleźć informację, że zanotowano maksymalny błąd 12% i minimalny równy 0.18%.

Zrealizowane kroki w procesie rozwiązywania problemu

Problem można rozwiązać, używając sieci neuronowej z wykorzystaniem deep learnigu.

W Internecie większość rozwiązań opiera się na TensorFlow - platformie open source do uczenia maszynowego,

Przykładowe rozwiązanie:
https://pythonprogramming.net/introduction-deep-learning-python-tensorflow-keras/

Następnie zmienialiśmy kolejne parametry i sprawdzaliśmy jak zmieniło się działanie sieci neuronowej - np. jak zmieniło się accuracy, loss, czas.

Wersja podstawowa

Używamy modułu tensorflow i tensorflow.keras.
Będziemy korzystać z prostego modelu sieci neuronowej Sequential() - czyli modelu, który ustawia warstwy sieci na sobie (jak stos) oraz każda warstwa ma jeden tensor/wektor/dane wejściowe i jeden tensor wyjściowy.

W podstawowej wersji używamy parametrów:

1. **Dwie warstwy ukryte sieci neuronowe**j:
   Każda warstwa ukryta posiada 128 neuronów.
   Są one połączone ze wszystkimi neuronami z poprzedniej warstwy (tf.keras.layers.Dense).
   Funkcją aktywacji każdej z nich (funkcja, która określa jak dane neurony wpływają na kolejne) jest tf.nn.relu.

2. Po **utworzeniu modelu** czyli:
   - Spłaszczeniu warstw, aby zdjęcia 28 x 28 pikseli użyć jako wektora 1x784 i “włożyć” go do modelu sieci neuronowej.
   - Dodaniu warstw ukrytych.
   - Dodaniu ostatniej warstwy (wynikowej).
   - Możemy skonfigurować model za pomocą model.compile()

   Możemy ustawić parametry **model.compile()**, czyli:
   - Optimizer - rodzaj algorytmu, który będzie optymalizował rezultaty naszego modelu.
     np. Optymalizator implementujący algorytm Adama, czyli algorytm oparty na stochastycznej aproksymacji metody gradientu prostego.
   - Loss - czyli funkcja straty, którą mamy minimalizować w procesie uczenia i ona powinna wpływać na to, że sieć maksymalizuje poprawność.
     np. Sparse_categorical_crossentropy
   - Metrics - oznacza czym jest nasz błąd.
     np. accuracy - oznacza jak często prognozy są równe etykietom.

3. Model.fit() oznacza **trenowanie modelu** i możemy ustawić ile razy, będzie się to odbywało.
   W podstawowej wersji epochs = 3.

Po ustawieniu naszego modelu sieci neuronowej otrzymujemy rezultaty:

Dla przykładowego uruchomienia (każde uruchomienie może się różnić tym jakie zdjęcia są poprawnie sklasyfikowane - czyli zmianą accuracy oraz tym jaki czas będzie trwało trenowanie - zależy to od możliwości komputera):

Czas treningu w poszczególnych epochach:
[**3.196532964706421, 2.7145726680755615, 2.7280514240264893**]
Loss dla danych testowych: **0.09123930335044861**
Accuracy dla danych testowych: **0.9718999862670898**
Loss dla danych trenowanych - uczących: **0.07381346821784973**
Accuracy dla danych trenowanych - uczących: **0.977400004863739**

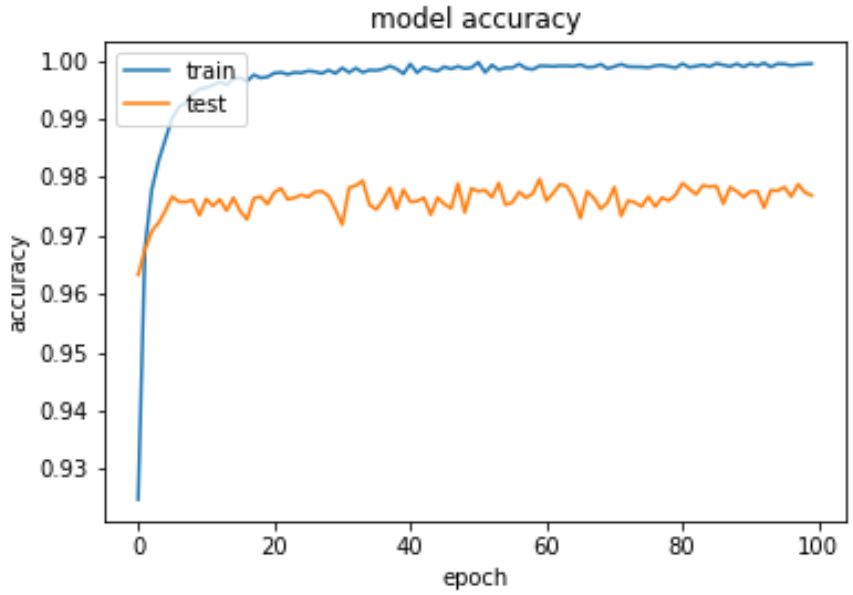
Możemy powiedzieć accuracy na poziomie 97.74 % jest całkiem dobre, ponieważ podczas poprowadzonych przez nas badań ciężko jest wyjść poza 98%.

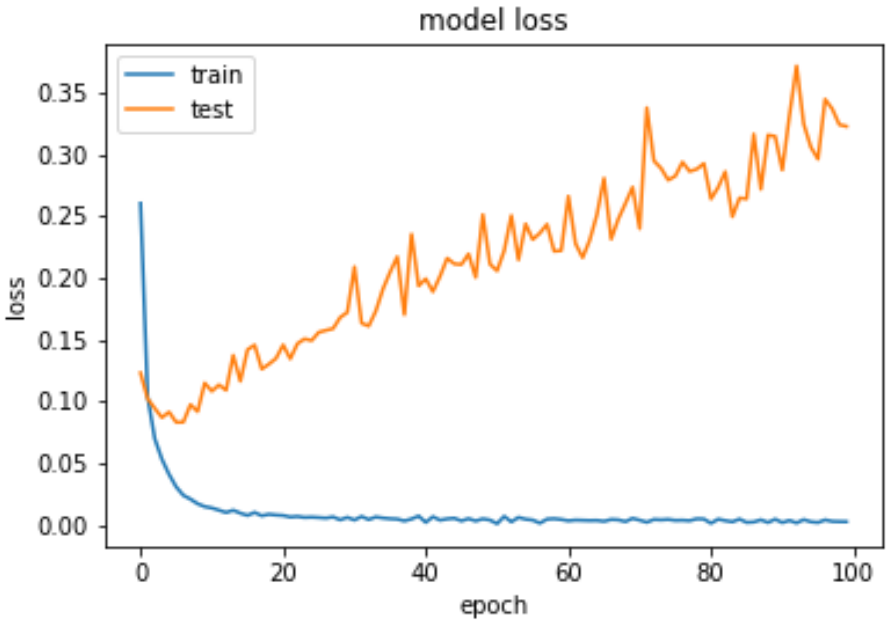
BADANIA

Val_acc oraz **val_loss** oznacza accuracy i loss zbioru testowego (e**val**uate) (10 000 elementów). Natomiast acc i loss oznaczają accuracy i loss zbioru uczącego (60 000 elementów).
Będziemy tymi skrótami posługiwać się w kodzie i w niektórych wykresach/tabelach.

Badanie ilości uczenia

Każde ponowne uczenie się sieci neuronowej, sprawa, że dla tych samych danych testowych wrasta accuracy oraz maleje loss. Widać to na poniższym wykresie.

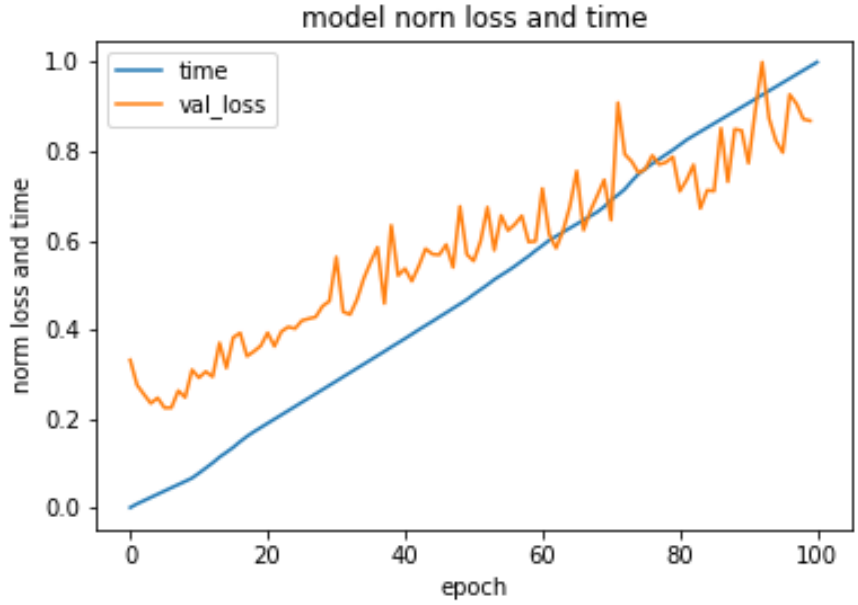




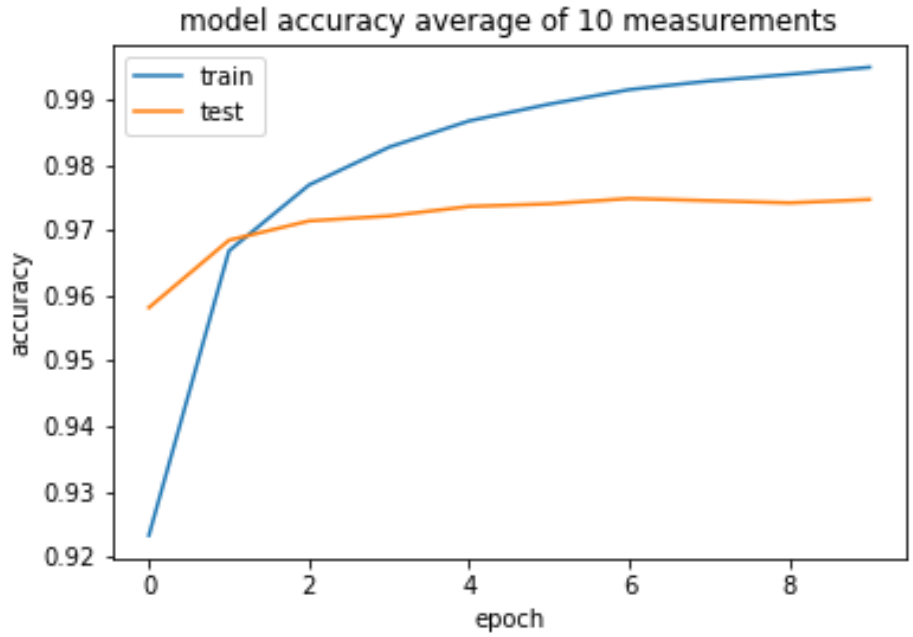
Z powyższych wykresów wynika, że ilość powtórzeń uczenia się wystarczy około 10, ponieważ większa ilość uczenia nie zwiększa dokładności danych testowych oraz treningowych, ale za to zwiększa się koszt (loss) danych testowych.

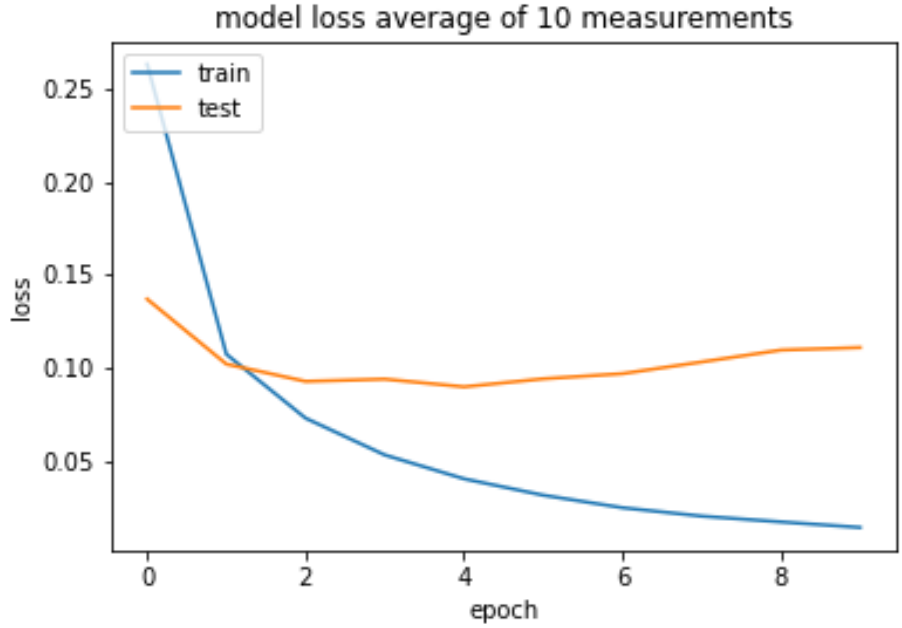
Również z wykresu model loss wynika, że od pewnej ilości epoch zwiększa się strata dla danych testowych, zatem przy projektowanie sieci neuronowej trzeba znaleźć odpowiednią epoch, przy którym loss jest najmniejsza dla danych testowych.

Złożoność algorytmu (naszej sieci neuronowej) jest liniowa i zależy od ilości uczenia się (epochs). Również widać, że val_lost ogólnie wzrasta liniowo w zależności od ilości epochs, ale to bardzo zależy od danej próby.



Wynik poszczególnego uczenia bardzo zależy od doboru zbioru testującego dlatego chcąc sprawdzić optymalną liczbę epoch, biorę średnia z 10 pomiarów.





Dla stworzonej sieci neuronowej najlepszą ilość uczenia jest 4, ponieważ dla tej wartości osiąga najlepsze wartości accuracy oraz los.

Badanie metryki

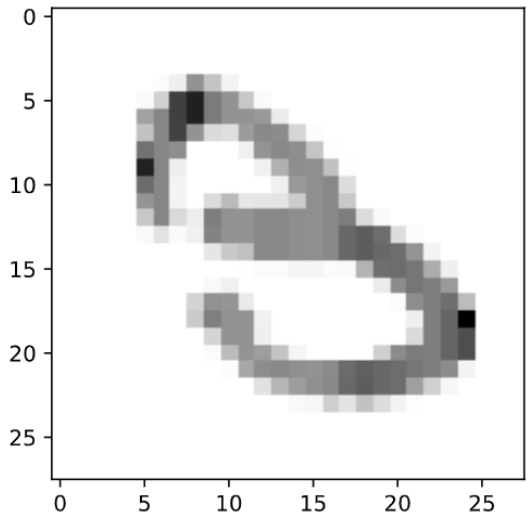
W wersji podstawowej metryką jest **accuracy**, czyli jak często prognoza jest równa etykietom.
Każde stworzenie i wytrenowanie modelu z parametrami “podstawowymi” daje różne rezultaty, ale accuracy zbioru testowego waha się w okolicach **0.98**, a zbioru uczącego **0.97**.

Oznacza to, że około **2%** zdjęć została źle sklasyfikowana.

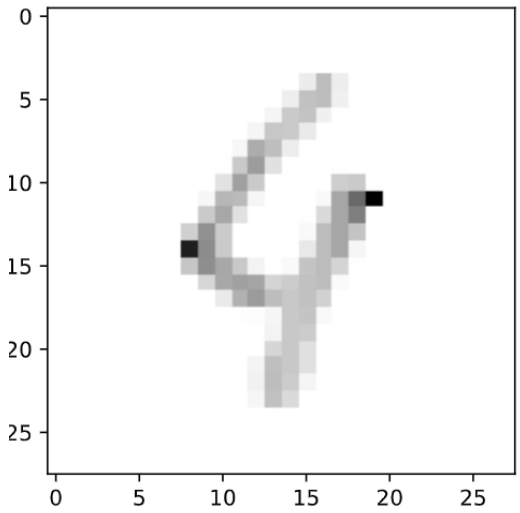
Możemy to sprawdzić, licząc ile predykcji zdjęć ze zbioru testowego, nie odpowiada ich etykietom i zobaczyć przykładowo źle sklasyfikowane zdjęcia.

Dla 500 zdjęć (używam pierwszych 500 zdjęć, aby szybciej zobaczyć rezultaty - sprawdzanie 10 000 zdjęć trwałoby o wiele dłużej), źle sklasyfikowanych dla przykładowego uruchomienia jest 12, czyli 12/500 = **0,024** co jest bliskie 0,02.

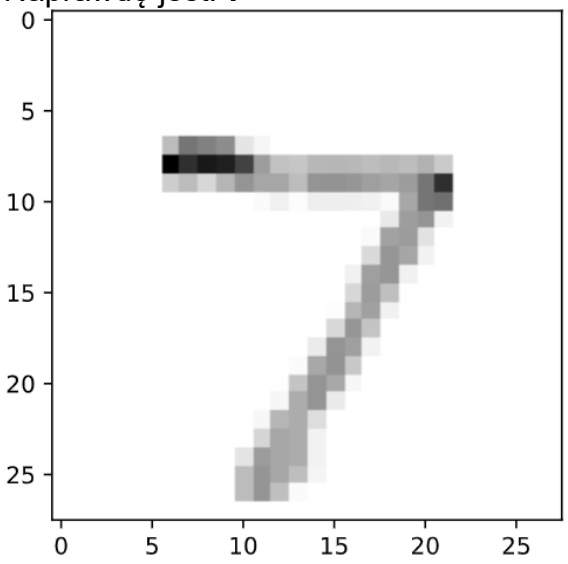
Przykładowo źle sklasyfikowane zdjęcie - zdjęcie ze zbioru testowego o indeksie 18:
Wykryto: **8**
Naprawdę jest: **3**



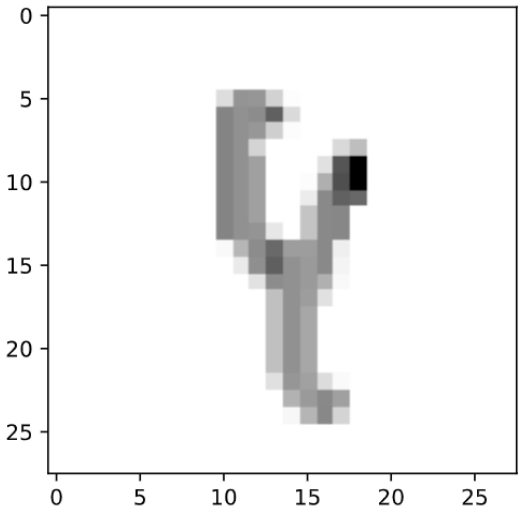
Przykładowo źle sklasyfikowane zdjęcie - zdjęcie ze zbioru testowego o indeksie 115:
Wykryto: **9**
Naprawdę jest: **4**



Przykładowo dobrze sklasyfikowane zdjęcie - zdjęcie ze zbioru testowego o indeksie 0:
Wykryto: **7**
Naprawdę jest: **7**



Przykładowo dobrze sklasyfikowane zdjęcie - zdjęcie ze zbioru testowego o indeksie 497:
Wykryto: **4**
Naprawdę jest: **4**



Szukając na stronie https://keras.io/api/metrics/ różnych metryk, możemy zauważyć, że “Accuracy metrics” jest kilka. Dodatkowo możemy zobaczyć tam inne grupy metryk.

Wiele metryk jest opartych na entropii krzyżowej.

Dla nas najbardziej zrozumiałą metryką jest accuracy - po prostu jak często prognozy są równe etykietom.

Dla przykładowych metryk możemy zobaczyć jak pozostałe parametry się zmieniają:

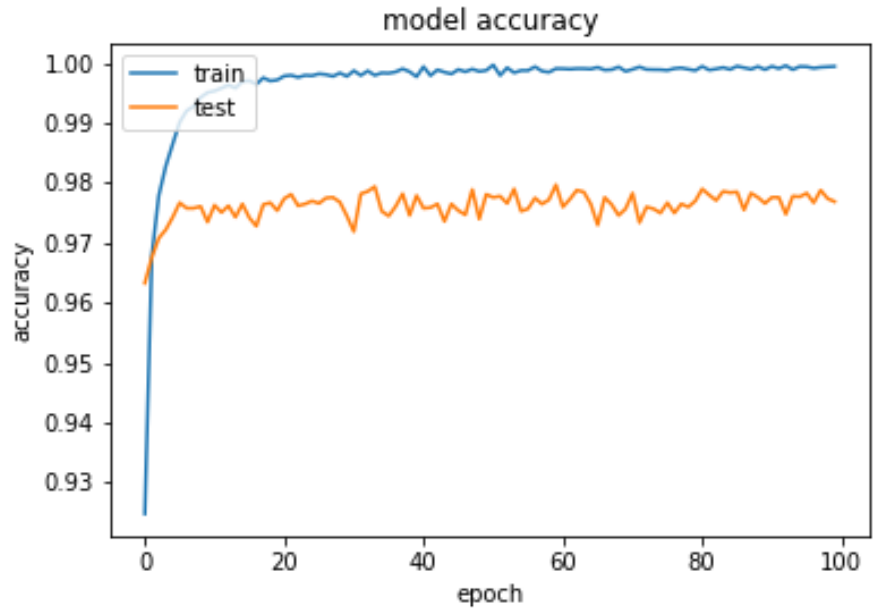
	metric	val_loss	val_acc	loss	acc	trains_time
0	accuracy	0.095869	0.970450	0.073604	0.976847	8.107486
1	binary_accuracy	0.097113	0.099545	0.072996	0.100090	8.339144
2	mean_squared_error	0.092442	27.336543	0.073182	27.390475	8.156726
3	sparse_categorical_crossentropy	0.092962	0.092962	0.072757	0.072757	8.017163

Każdy pomiar przeprowadzony był 10 razy - w tabeli znajdują się średnie tych pomiarów.

Dzięki uśrednieniu pomiarów możemy zobaczyć, że czas treningu jest podobny oraz, że val_loss i loss także są podobne. Oznacza to, że metryka nie wpływa nasz model, jedynie pokazuje w różny sposób jego poprawność.

Możemy zauważyć, że tf.keras.metrics i tf.keras.losses posiadają te same nazwy klas np. SparseCategoricalCrossentropy. Oznacza to, że te same wzory i obliczenia może służyć do obliczania loss i metryki.

Patrząc na wykres zależności accuracy od ilości epochów:



Możemy zauważyć, że ogromna ilość uczenia doprowadza do tego, że prawie 100% zdjęć zostaje dobrze sklasyfikowanych **w zbiorze uczących**.

Jednak mimo tak zadowalających rezultatów w zbiorze uczącym, **zbiór testowy** jest klasyfikowany ze skutecznością nie większą niż około 98%.

Badanie funkcji aktywacji

Funkcja aktywacyjna/aktywacji to równania matematyczne, które określają wyjście pojedynczego neuronu, który przyjmuje znormalizowane wartości w zakresie od 1 do 0 lub od -1 do 1.

Równanie te polega na tym, że każdy neuron ma wagę, a pomnożenie liczby wejściowej przez tę wagę daje wynik neuronu, który jest przenoszony do następnej warstwy.

Badana przez nas sieć neuronowa to sieć gęstą (**Dense**), w której każdy neuron jest połączony z każdym neuronem z kolejnej warstwy, zatem każdy neuron jest zawsze aktywowany.

Wykorzystujemy w naszym modelu **deep learning**, dlatego sens ma wykorzystywanie jedynie funkcji nieliniowych, ponieważ dla funkcji liniowych nie ma znaczenia ilość warstw (zawsze i tak jest jedna warstwa).

W bibliotece keras są dostępne następujące funkcje aktywujące:
Relu, sigmoid, softplus, softsing, tanh, selu, elu, Swish.

Aby ocenić, która funkcja aktywująca jest najlepsza dla badanego modelu, została obliczona średnia z dziesięciu uruchomień programu, otrzymano następujące wynik:

	activation	val_loss	val_acc	loss	acc	trains_time
0	elu	0.112960	0.96444	0.106455	0.966818	6.337522
1	relu	0.097200	0.97012	0.073312	0.976762	6.170875
2	selu	0.128450	0.96046	0.125520	0.961248	6.382904
3	sigmoid	0.154221	0.95231	0.164776	0.950677	6.231506
4	softplus	0.130594	0.95919	0.138561	0.957287	6.595049
5	softsing	0.118347	0.96375	0.114616	0.965460	6.305855
6	swish	0.095859	0.97016	0.084753	0.973575	6.588647
7	tanh	0.113041	0.96524	0.110648	0.966272	6.458946

Dla przypomnienia val_ - zbiór testowy)

Jak widać z tabelki wyżej, funkcje aktywujące mają zbliżone wyniki, i aby ocenić, która funkcja aktywująca jest najlepsza, sprawdzono los oraz accuracy dla danych testowych.

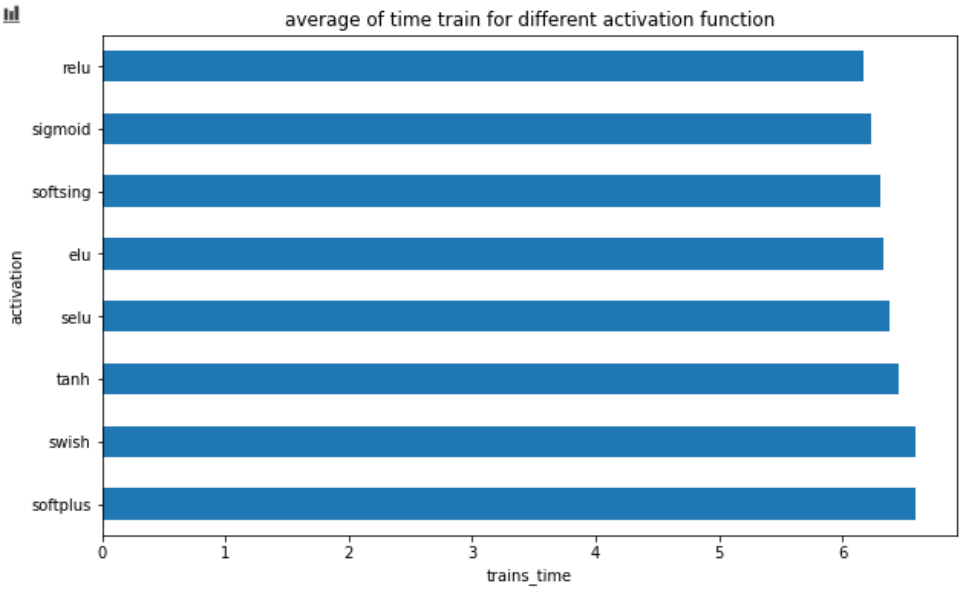
Z grafu wyżej wynika, że najlepszą funkcji aktywującą jest **swish** albo **relu**, bo mają największą poprawność (accuracy) oraz najmniejszą stratę (loss).

Również można zauważyć, że wszystkie funkcje aktywujące mają bardzo zbliżone wartości, dlatego dla naszego modelu nie to wielkiego znaczenia jaką funkcję aktywującą wybrać.

Warto jednak zauważyć, że mimo małej różnicy między poprawnościami w zbiorze testowym:
np. między **swish** i **relu** różnica wynosi **0,00004**,
a między **swish** a **sigmoid 0,01785**.

Takie wartości w teorii oznaczają, że dla np. **100 000** zdjęć, gdyby wybrano **swish** ponad **relu**, to tylko **4** zdjęcia więcej byłyby poprawnie sklasyfikowane, a dla **swish** nad **sigmoid** to aż **1785** zdjęć.

Jednakże, aby ocenić czy swish, czy relu jest najlepszą funkcją aktywującą dla naszego modelu, to porównano czas wykonania tych dwóch funkcji aktywujących.



Jak widać, z wykresu wyżej relu ma średnio szybszy czas wykonania niż swish, dlatego może uznać, że funkcja aktywująca relu jest najlepsza dla naszego modelu.

Dokładniejszy opis poszczególnych funkcji aktywujących można znaleźć na stronie: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/

Wnioski ogólne z badania funkcji aktywacji:

Dobór funkcji aktywacji dla tak prostych modeli jak nasz nie ma wielkiego znaczenia, najlepiej wybrać relu albo swish w zależności czy zależy na dokładności, czy na czasie.

Trwają badania znalezienia sposobu automatycznego wybór funkcji aktywacji, w celu uzyskania najwyższej dokładności. (np. Prowadzili je Franco Manessi o Alessandro Rozza.)

Badanie funkcji strat

Parametrem, który staramy się jak najbardziej zminimalizować podczas treningu sieci neuronowej, jest loss. Dzięki jej minimalizacji maksymalizujemy naszą “poprawność”.

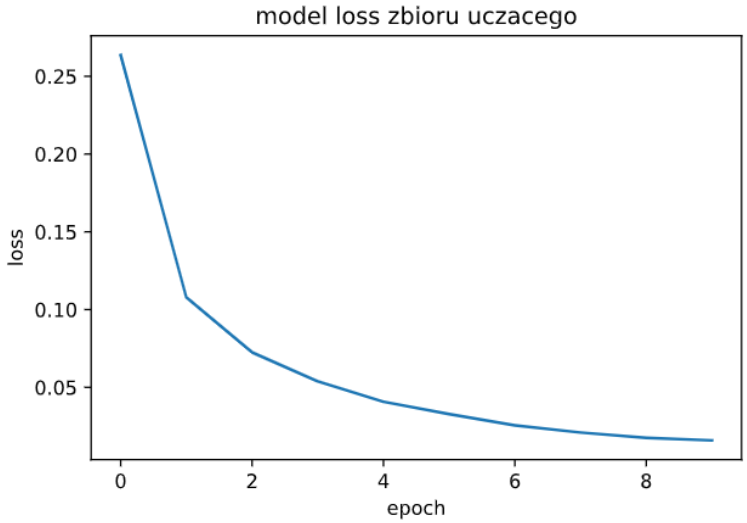
Możemy wybrać daną funkcję strat spośród wielu opisanych np. na stronie https://keras.io/api/losses/
czy https://www.tensorflow.org/api_docs/python/tf/keras/losses.

W wybranej wersji sieci neuronowej używamy **sparse_categorical_crossentropy**.

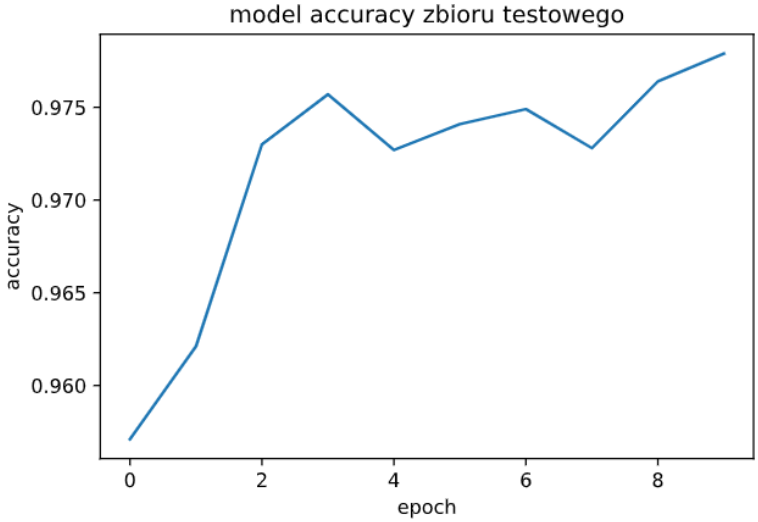
Szukając informacji na temat parametrów możemy często się spotkać z zagadnieniem entropii krzyżowej (pomiędzy etykietami i przewidywaniami). Jest ona związana z teorią informacji, która jest powiązana z prawdopodobieństwem, statystyką i matematyką.

Jak mogliśmy zauważyć podczas badania ilości uczenia, loss maleje do zera podczas trenowania zbioru uczącego.

Wykres dla loss = sparse_categorical_crossentropy i metrics = accuracy:



Natomiast accuracy naturalnie chcemy aby nam rosło do 1:

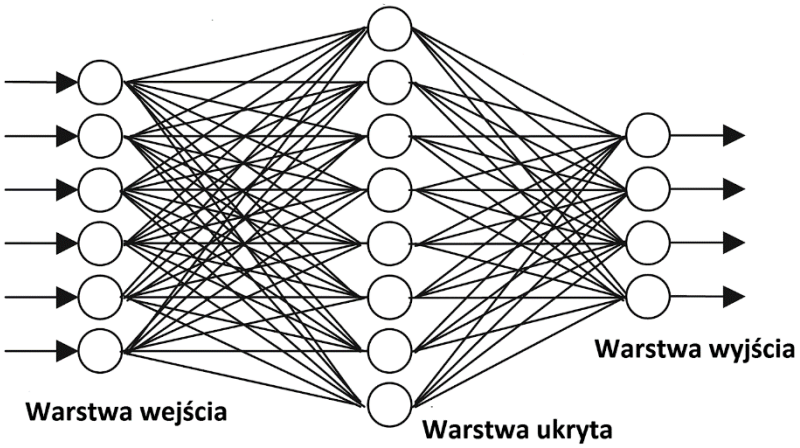


Okazuje się, że używając innej funkcji strat niż **sparse_categorical_crossentropy,** dla naszego modeli sieci neuronowej accuracy nie rośnie i pozostaje w okolicach **0.1** - praktycznie każde zdjęcie jest źle sklasyfikowane.

Może być to spowodowane tym jak zostały wybrane pozostałe parametry lub być może tym, że nasza warstwa wyjściowa ma funkcję aktywacji **tf.nn.softmax** (czyli rozkład prawdopodobieństwa i największe z nich (model.predict.argmax()) wskazuje sklasyfikowaną etykietę).

Badanie wpływu zmian liczby neuronów

W poniższym badaniu zostało zbadane w jaki sposób zmieniają się właściwości accuracy i loss zarówno zbioru uczącego, jak i testowego oraz czas uczenia w zależności od liczby neuronów w ukrytych warstwach.



Badanie było przeprowadzone na dwóch warstwach ukrytych, a liczba neuronów dotyczy liczby w pojedynczej warstwie (nie sumie).

	number_of_neurons	val_loss	val_acc	loss	acc	trains_time
0	16.0	0.217223	0.936317	0.220425	0.935336	12.083656
1	32.0	0.150558	0.954158	0.151649	0.954194	12.712865
2	48.0	0.125091	0.961533	0.118937	0.963543	14.918776
3	64.0	0.115433	0.964967	0.102178	0.968594	13.523852
4	80.0	0.105309	0.967708	0.091165	0.971708	14.148135
5	96.0	0.100429	0.969092	0.081968	0.974326	7.317846
6	112.0	0.098242	0.969500	0.077011	0.975890	7.017173
7	128.0	0.094132	0.971333	0.073306	0.976822	8.394929
8	144.0	0.091548	0.971750	0.070174	0.977990	9.437391
9	160.0	0.090374	0.972083	0.067641	0.978496	19.857723

Każdy pomiar przeprowadzony był (automatycznie) 10 razy - w tabeli znajdują się średnie tych pomiarów

Wnioski:
Z powyższej tabeli możemy zauważyć, że wraz ze wzrostem liczby neuronów w pojedynczych ukrytych warstwach zyskujemy dokładność.

Nie jest to jednak liniowa zależność. Najbardziej zauważalną różnicę można dostrzec podczas zmiany liczby neuronów z 16 na 32, gdyż zarówno dla zbioru uczącego, jak i testowego wynosiła ona około 2 punkty procentowe.

Wartym odnotowania jest fakt, że czasy uczenia również nie rosły proporcjonalnie - co widać szczególnie przy zmianie z 80 neuronów (około 14 sekund) do 96 (nieco ponad 7 sekund).

Najkorzystniejszymi liczbami neuronów mogą być zatem między innymi 96, 112, 128, ponieważ nie dość, że uczenie zajmuje krócej, to dokładność jest wysoka i nie zmienia się ona znacząco dodając kolejne neurony.

Badanie wpływu zmian liczby warstw

W poniższym badaniu zostało zbadane w jaki sposób zmieniają się accuracy i loss zarówno zbioru uczącego, jak i testowego w zależności od liczby warstw ukrytych.

	layer	val_loss	val_acc	loss	acc	trains_time
0	0.0	0.295018	0.91901	0.308779	0.913243	5.857893
1	1.0	0.108028	0.96765	0.098728	0.970538	7.992885
2	2.0	0.095370	0.97077	0.073327	0.976802	9.394294
3	3.0	0.097081	0.97069	0.072410	0.977358	10.678525
4	4.0	0.100682	0.97045	0.078392	0.975700	10.944129
5	5.0	0.109360	0.96829	0.085091	0.974068	12.138953
6	6.0	0.117771	0.96586	0.091571	0.973133	13.824395
7	7.0	0.126987	0.96572	0.098960	0.971787	13.821851
8	8.0	0.132818	0.96423	0.105772	0.970735	15.628236
9	9.0	0.141299	0.96309	0.113429	0.969398	17.551524

Każdy pomiar przeprowadzony był (automatycznie) 10 razy - w tabeli znajdują się średnie tych pomiarów.

Wnioski:
Z powyższej tabeli można zauważyć, że znacząca różnica w dokładności jest zauważalna jedynie przy dodaniu jakiejkolwiek warstwy ukrytej i sięga aż 6 punktów procentowych (5 dla zbioru testowego).

Każde kolejne warstwy zmieniają te dokładności nieznacznie, jednocześnie wydłużając czas trwania uczenia. Zatem najkorzystniej jest stosować jedną/dwie warstwy ukryte - różnica w wydajności z dodawania kolejnych jest znikoma.

Oczywiście brak różnic w wydajności może mieć związek z tym, że nasz model sieci neuronowej jest Sequential() albo z tym, że ustawieniem pozostałych parametrów.

Badanie różnych algorytmów optymalizacyjnych

W poniższym badaniu zostało zbadane w jaki sposób zmieniają się właściwości accuracy i loss zarówno zbioru uczącego, jak i testowego oraz czas uczenia w zależności od wykorzystanego algorytmu optymalizującego.

	optimizer	val_loss	val_acc	loss	acc	trains_time
0	Adadelta	2.162804	0.41194	2.191498	0.353350	9.135151
1	Adagrad	0.581595	0.86333	0.681549	0.841287	8.559567
2	Adam	0.094769	0.97006	0.072812	0.977230	8.755471
3	Adamax	0.137864	0.95821	0.147931	0.956478	9.387148
4	Ftrl	2.301982	0.11350	2.302062	0.112367	10.892714
5	Nadam	0.094860	0.97086	0.070892	0.977560	12.649051
6	RMSprop	0.107236	0.97017	0.085739	0.974687	10.067314
7	SGD	0.291232	0.91820	0.321590	0.907942	7.843386

Każdy pomiar przeprowadzony był (automatycznie) 10 razy, w tabeli znajdują się średnie tych pomiarów.

- **Adadelta** - algorytm osiągnął małą dokładność (ok. 41% dla zbioru testowego oraz ok. 35% dla zbioru uczącego) przy przeciętnym czasie uczenia;
- **Adamax** - algorytm osiągnął dobrą dokładność (ok. 86% dla zbioru testowego oraz ok. 84% dla zbioru uczącego) przy stosunkowo niskim czasie uczenia;
- **Adam** - algorytm ten osiągnął bardzo dobrą dokładność (ok. 97% dla zbioru testowego oraz prawie 98% dla zbioru uczącego) przy stosunkowo niskim czasie uczenia;
- **AdaMax** - algorytm ten osiągnął nieco niższe dokładności niż ,,Adam'' (ok. 96% zarówno dla zbioru testowego, jak i uczącego) przy czasie uczenia trochę dłuższym;
- **Ftrl** - algorytm ten osiągnął bardzo słabą dokładność (ok. 11% zarówno dla zbioru testowego, jak i uczącego) przy dłuższym czasie niż wcześniej wymienione algorytmy;
- **Nadam** - algorytm ten osiągnął bardzo dobrą dokładność (ok. 97% dla zbioru testowego oraz prawie 98% dla zbioru uczącego) przy stosunkowo wysokim czasie uczenia;

- **RMSprop** - algorytm ten osiągnął bardzo dobrą dokładność (ok. 97% zarówno dla zbioru testowego, jak i uczącego) przy przeciętnym czasie uczenia;
- **SGD** - algorytm ten osiągnął dobrą (lecz w porównaniu do np. Adama mniej korzystną) dokładność (ok. 91% zarówno dla zbioru testowego, jak i uczącego) przy najkrótszym czasie uczenia.

Badanie to pokazało, że w naszym przypadku najkorzystniej jest wybrać optymalizator Adam - bardzo wysoka dokładność przy niskim czasie uczenia. Alternatywnie można posłużyć się algorytmami RMSprop lub Adamax i osiągnąć niewiele gorsze rezultaty. Najmniej korzystne wyniki zaprezentował algorytm Ftrl.

Ogólne wnioski

- Bardzo duży wpływ na dokładność klasyfikacji ma dobrany algorytm optymalizacyjny (algorytm Adam osiąga accuracy blisko 98%, a dla kontrastu algorytm Ftrl tylko 11%);

- Przy tworzeniu sieci neuronowej opartej na deep learningu, najważniejsze jest ustawienie poprawnej pierwszej i ostatniej warstwy (co na input oraz output). Samo projektowanie sieci jest już mniej istotne, ponieważ z roku na roku powstają coraz lepsze algorytmy ułatwiające tworzenie sieci.

- Liczba warstw ukrytych znacząco nie wpływała na zmianę dokładności -- jedynie przypadek, gdy nie było żadnej warstwy znacząco obniżał dokładność. Dlatego nie ma potrzeby stosowania ich dużej liczby - zwłaszcza, że każda kolejna dodatkowo wydłuża czas uczenia;

- Liczba neuronów w warstwie ukrytej wpływa na zmianę dokładności - sprawdziliśmy różne wartości pomiędzy liczbą w pierwszej i ostatniej warstwie i dobraliśmy wyznaczyliśmy kilka, które dawały dużą dokładność przy stosunkowo niskim czasie;

- Dobór funkcji aktywującej nie jest bardzo istotny dla tak prostych modeli. Również w przyszłość powstanie automatyczne wybieranie funkcji aktywującej.

- Poprawnie wybór ilości uczenia się możemy ustalić na podstawie val_loss, która ciągnie maleje.

- Dla podstawowych parametrów nieważne ile będziemy naszą sieć neuronową Sequential uczyć, i tak nie otrzymamy accuracy większej niż 98%.

- Dla naszego modelu jedyną funkcją strat, która powoduje zwiększanie się poprawności jest sparse_categorical_crossentropy.

- Z powyższych punktów wytypowaliśmy najlepsze parametry dające najlepsze rezultaty i stworzyliśmy optymalny model – okazał się on taki sam jak w przykładowym rozwiązaniu.

  (alg. Adama, 128 neuronów w warstwie ukrytej, 2 warstwy ukryte, ilość uczenia = 3, funkcja aktywująca = relu).

  Model ten został zapisany w katalogu projektu, a jego użycie jest zaprezentowane w pliku reading_from_saved_model.ipynb.

  Osiągnęliśmy w ten sposób:

```
Accuracy dla danych testowych:  0.9700999855995178
Loss dla danych testowych:  0.09643306583166122
Accuracy dla danych uczących:  0.9764833450317383
Loss dla danych uczących:  0.0733737125992775
Czas uczenia:  8.313476800918579
```

  (Dane z pliku saving_default_model.ipynb)

Podział obowiązków

	Andrzej Czechowski	**Franciszek Wysocki**	**Bartosz Zakrzewski**
1. Epochs	Z	S	S
2. Wstęp	D, S	S	Z
3. Loss	S	S	Z
4. Metrics	S	S	Z
5. Liczba neuronów	S	Z	S
6. Ilość warstw sieci neuronowej	S	Z	D,S
7. Activation	Z	S	D, S
8. Optimizer	S	Z	S
9. Zapisywanie i odczytywanie optymalnego modelu	S	Z	Z
10. Wnioski ogólne	Z, S	Z, S	Z, S

Z – zrobił, D – dopisał, S - sprawdził

Źródła

- https://pythonprogramming.net/introduction-deep-learning-python-tensorflow-keras/
- https://en.wikipedia.org/wiki/MNIST_database
- https://www.youtube.com/watch?v=aircAruvnKk&feature=share&ab_channel=3Blue1Brown
- https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d
- https://machinelearningmastery.com/
- https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/
- https://stackoverflow.com/questions/43504248/what-does-relu-stand-for-in-tf-nn-relu
- https://www.tensorflow.org/
- https://keras.io/api/
- https://gombru.github.io/2018/05/23/cross_entropy_loss/
- https://towardsdatascience.com/cross-entropy-for-classification-d98e7f974451