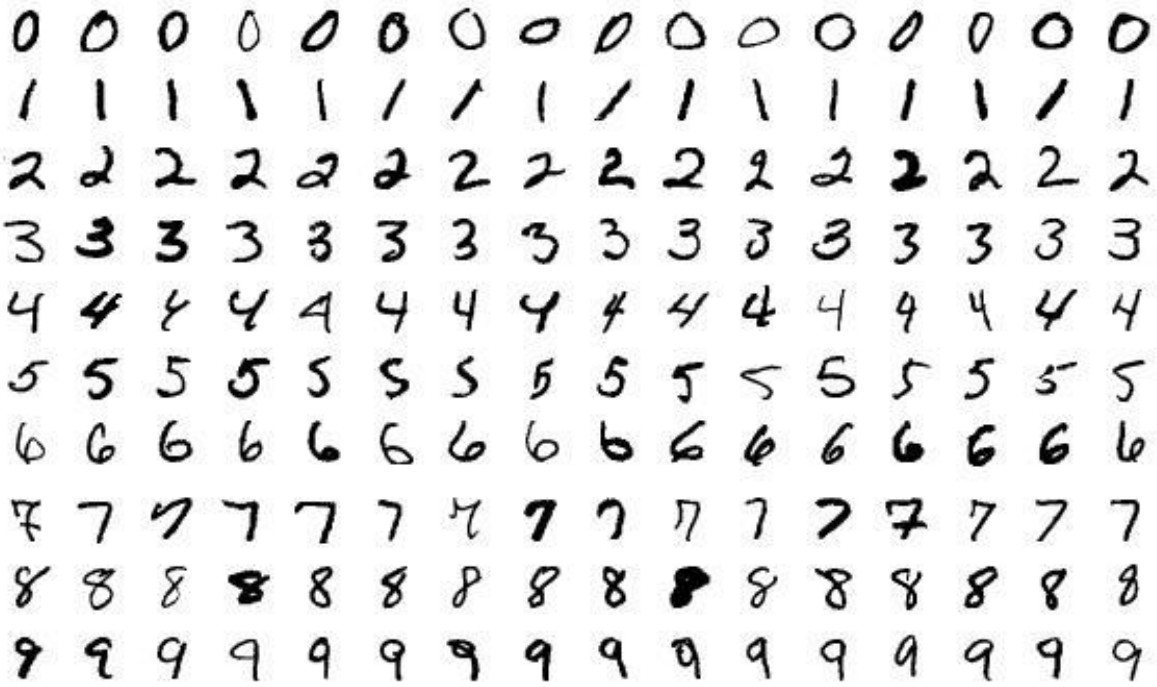
Sztuczne sieci neuronowe

Rozpoznawanie cyfr na podstawie zdjęć o wielkości 28x28px
przy wykorzystaniu zestawu danych Mnist

Wprowadzenie:

W tym projekcie budujemy model sztucznych sieci neuronowych do rozpoznawania cyfr z obrazów. Ze względu na niskie doświadczenie autorów w rozwiązywaniu tego typu problemów, zdecydowaliśmy się wykorzystać zbiór danych mnist, ponieważ został on wykorzystany w wielu podobnych pracach. , a następnie zostały delikatnie zmodyfikowane. Pliki z danymi zostały pobrane ze strony: https://git-disl.github.io/GTDL-Bench/datasets/mnist_datasets/ , a następnie zostały delikatnie zmodyfikowane. Owa modyfikacja polegała na przeniesieniu danych celem symetrycznego rozłożenia zapotrzebowania pamięci przez owe pliki. Podział był niezbędny celem rozwiązania problemów, które generował github. Następnie zmieniliśmy nazwy plikom na: „mnist1.csv” i



„mnist2.csv”.

W tym projekcie stosujemy tradycyjną sieć neuronową. Każdy neuron z konkretnej warstwy, łączy się z każdym neuronem sąsiadujących warstw. Dodatkowo, w naszym projekcie stosujemy także biasy, czyli stałe wartości, które są dodawane do wartości, która powstała w wyniku zsumowania iloczynu wag i wartości neuronu.

Przegląd literatury z zakresu danych MNIST:

Zbiór danych MNIST[1] to inaczej zestaw małych obrazów cyfr napisanych odręcznie przez uczniów szkół średnich i pracowników US Census Bureau (rządowa agencja odpowiedzialna za spisy ludności Stanów Zjednoczonych). Na przykładzie tych danych można wypróbować techniki uczenia się i metody rozpoznawania wzorów na danych ze świata rzeczywistego, jednocześnie poświęcając minimalny wysiłek na przetwarzanie i formatowanie danych[2].

Przykład wykorzystania danych MNIST przy pomocy biblioteki Scikit-Learn Python'a – na podstawie książki „Hands-on Machine Learning with Scikit Learn, Keras & TensorFlow"[3]:

Korzystając z biblioteki Scikit-Learn w python'ie pobierany jest zbiór danych MNIST. Dane to 70 000 obrazów z czego każdy ma 28 × 28 pikseli. Łącznie jeden obraz ma 784 cech z czego każda cecha reprezentuje po prostu intensywność jednego piksela, zaczynając od 0 (biały) do 255 (czarny). Na podstawie tych danych tworzony jest zbiór danych treningowych i testowych zgodnie z ich pierworodnie wyznaczonym podziałem w zbiorze danych (0-60000 dane treningowe, 60000-70000 dane testowe).

W pierwszej kolejności podejmowana jest próba klasyfikacji jednej cyfry – numer 5. Taka klasyfikacja odpowiada klasyfikatorowi binarnemu, który zdolny jest do rozróżnienia tylko dwóch klas – w tym przypadku na klasy 5 i nie-5. Do tego celu wybrany jest klasyfikator Stochastic Gradient Descent (SGD) z klasy SGDClassifier pakietu Scikit-Learn – testowany on jest na całym zestawie treningowym. Następnie wykonana została predykcja i ocena wydajności tego modelu.

Ocena klasyfikatora zajęła dość obszerną część tematu w tej literaturze. Przykładowo zaprezentowano użycie walidacji krzyżowej jako dobry sposób oceny modelu, z której funkcji również można skorzystać z pakietu Scikit-Learn. Innym efektywnym tutaj sposobem oceny wydajności klasyfikatoru jest przyjrzenie się macierzy pomyłek.

W dalszej kolejności podjęta została klasyfikacja wielu cyfr. Podane zostaje rozróżnienie algorytmów na takie, które nadają się do klasyfikacji wieloklasowej (klasyfikatory SGD, klasyfikatory RF (Random Forest), naiwny klasyfikator Bayesowski) jak i na te, które są klasyfikatorami ściśle binarnymi (regresja logistyczna, maszyna wektorów nośnych (działa jak klasyfikator)). W przypadku klasyfikatorów binarnych istnieją jednak rozwiązania, w których można stosować je do klasyfikacji wieloklasowej. Istnieje tutaj dowolność wyboru sposobu weryfikacji klasyfikatora.

Okazuję się , że w przypadku klasyfikacji wieloklasowej również istotną rzeczą jest przeanalizowanie rodzaju błędów w celu ulepszenia modelu. Choć jest to

---

[1] Https://en.wikipedia.org/wiki/MNIST_database#cite_note-1 [dostęp 09.01.24r.]

[2] http://yann.lecun.com/exdb/mnist/index.html [dostęp 09.01.24r.]

[3] Aurélien Géron „Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow. Concepts, Tools, and Techniques to Build Intelligent Systems", 2019 O'Reilly.

czasochłonne to analiza poszczególnych błędów może być też dobrym sposobem na uzyskanie wglądu w to, gdzie zawodzi klasyfikator.

Przykład zastosowania Sztucznych Sieci Neuronowych z wykorzystaniem danych MNIST za pomocą biblioteki Keras Python'a – na podstawie projektu zaprezentowanego na stronie Kaggle[4]:

Keras to potężna i łatwa w użyciu bezpłatna biblioteka Pythona o otwartym kodzie źródłowym, służąca do uczenia maszynowego. Ponadto jest częścią biblioteki TensorFlow i pozwala definiować i trenować modele sieci neuronowych w zaledwie kilku linijkach kodu. Oferuje również proste interfejsy API.

W bibliotece Keras zbiór danych MNIST jest wstępnie załadowany w postaci czterech tablic Numpy. Podobnie jak w poprzednim przykładzie dane są początkowo podzielone na zbiór danych treningowych i testowych (x_train, y_train, x_test, y_test). W tym przypadku następuje jednak dodatkowa normalizacja danych (podzielenie przez 255, czyli zakres liczbowy w jakich obrazy są reprezentowane).

Budowa podstawowej sieci neuronowej w bibliotece Keras jest możliwa dzięki klasie Sequential. Metodą .add() można ustawić typy warstw modelu poprzez wskazanie liczby neuronów czy funkcji aktywacji. W przykładzie zostały ustawione dwie warstwy po 10 neuronów każda z funkcjami aktywacji kolejno sigmoid i softmax z czego w pierwszej warstwie zostało wskazane wejście 784 cech. Równie łatwo można ten model zweryfikować metodą .summary(). Dzięki właśnie tej metodzie zostaje w tym przykładzie wyjaśniona liczba parametrów wyjściowych, która jest skutkiem ustalonej liczby neuronów w warstwach i wejścia.

Inne wyodrębnione w tym przykładzie przydatne metody to metoda .compile() służąca do kompilacji modelu, metoda .fit() służąca do uczenia modelu oraz metoda .evaluate() służąca ocenie modelu. W kompilacji modelu zostały ustawione parametry na categorical_crossentropy jako funkcja straty oraz Stochastic Gradient Descent (SGD) jako optymalizator. W dalszej kolejności dane zostały trenowane i finalnie została dokonana ocena modelu.

Podobnie jak w poprzednim przykładzie również zostało zaproponowane przyjrzenie się macierzy błędów, która zlicza przewidywania w porównaniu z wartościami rzeczywistymi. Takie podejście umożliwia zrozumienie wydajności modelu. Dodatkowo w tym przykładzie zostały wykonane również przewidywania za pomocą metody .predict().

W tym przypadku ocena testu została uzyskana na poziomie ok 83%, co mówi nam, że ten model SSN poprawnie klasyfikuje cyfry w 83% przypadków.

[4] https://www.kaggle.com/code/prashant111/comprehensive-guide-to-ann-with-keras/notebook [dostęp 10.01.24r.]

Przykład rozpoznawania wzorów cyfr z danych MNIST na podstawie artykułu „Pattern recognition of Handwritten Digits MNIST Dataset”[5]:

Podobnie jak w poprzednim przykładzie jest zaproponowane tutaj podejście wykorzystania danych MNIST również przy pomocy biblioteki Karas Python’a (zapewne z tego powodu, że dzięki tej bibliotece w szybki i łatwy sposób można posługiwać się sztucznymi sieciami neuronowymi). Jednak dużą różnicą między tymi przykładami jest ustawienie warstw – w tym przypadku jest ustawiona większa ich ilość i wykorzystana jest głównie funkcja aktywacji relu. W tym przykładzie model uzyskuję również większą dokładność mianowicie na poziomie ok. 90% i wzwyż.

[5] https://d1wqtxts1xzle7.cloudfront.net/67367555/PatternRecognition_Report_AutoRecovered_-libre.pdf?1621390409=&response-content-disposition=inline%3B+filename%3DPattern_Recognition_of_Handwritten_Digit.pdf&Expires=1704839199&Signature=M8ChQajt6elHtIoWNzlftd-mrKFEKI2mVKuhWgOM5K5uLwl3M1mliGfnE7wE~fwH-mePSZNWhIQGAHFPNTU-UUiFXCd0WF3a0ldWRc949Nh6np5laM7RmvH6BXR1SuAa1qUuqGTAbVm5nUse8o4QZK8FquiPZd-~d3nibzdKvvb3U~9TQTx1Q2wEQZHlBTxm-Zxq~ONc84dwwyo-flXuWrrV0uYqlyyV7L7cS0NzrCredE8QldxiuO5c2WyuiqmchkaJ-gpFVnyA-uD9Ayt~vreK0AUC7TnJ5pl03Ae80RxcpcxZ7tK2M9Qf0-fbZEFFDiVtB9ep2HvibukP-UafUlK1w__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA [dostęp 10.01.24r.]

Analiza

Stosunek danych testowych do uczących (test_sample_percent)

Pierwszym parametrem przyjmowanym przez naszą sieć neuronową, jest informacja, jaką część posiadanych danych, chcemy przeznaczyć na dane testowe. Wpisana liczba może być w przedziale od 0 do 1 (bez tych dwóch wartości). Dokładniejszy przedział definiuje wielkość zestawu danych. W naszym przypadku, zestaw mnist ma w sumie 70.000 danych, więc przyjmowane przez nasz program wartości muszą być z zakresu <1/70000 ; 69999/70000>. Pomimo tej teoretycznej wartości, przez niedoskonałości zaokrągleń komputerów, oraz możliwość wyboru niektórych zaimplementowanych systemów podziału danych, zalecamy wybierać liczby z przedziału od 0,005 do 0,995.

Parametr ten, cechuje się zależnością, że im większą liczbę użytkownik wybierze, tym więcej danych będzie weryfikowało skuteczność sieci neuronowej. W przypadku większej liczby danych testowych, sam proces uczenia się sieci jest istotnie szybszy, jednocześnie zbyt duża liczba powoduje brakiem uodpornienia się sieci na nietypowe przypadki. Analogicznie, spadek tej wartości, z powodu iż powiększa zbiór uczący, przygotowuje go na większą ilość nietypowych przypadków, co zwiększa jego skuteczność. Odbywa się to kosztem zbioru testowego, którego zbyt mały rozmiar zaowocuje wynikami zależnymi od ziarna losującego i brakiem wiarygodnej jakości predykcji.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus – braku zmiany wartości innych parametrów/czynników wpływających na wynik.

Wartość parametru	Wynik (w procentach)
0.005	86,479
0.2	88,796
0.6	88,213
0.995	58,526

Sposób dzielenia danych na zbiór testowy i uczący (type_of_split)

Drugim przyjmowanym parametrem jest enum oznaczający sposób dzielenia danych na zbiór uczący i treningowy. Posiadamy 4 takie metody:

- RANDOM - jest to metoda losowego podziału na grupy bez możliwości powtórzenia się wartości.
- STRATIFIEDSAMPLING - jest to metoda, losowego podziału na grupy, bez możliwości powtarzania się wartości, ale z zachowaniem proporcji podzbiorów (w tym przypadku zachowanie proporcji ilości cyfr).
- BOOTSTRAPPING - jest to metoda, która dla zbioru treningowego, wykonuje tyle losowań z całego zbioru danych, aż uzyska odpowiednią liczbę unikalnych wartości. W tym przypadku w zbiorze uczącym występują powtórzone wartości.
- RANDOMWITHIMPORTANCE - jest to metoda, która w sposób losowy przydziela dane do zbiorów treningowego i testowego, a następnie dla zbioru

uczącego, kopiuje losowe wartości podgrup (w tym przypadku konkretnych cyfr) tyle razy, aż uzyska taką samą proporcję danych jak przed podziałem na grupy.

Każda metoda stara się odzwierciedlić w jakiś sposób proporcje podgrup/odpowiedzi. Metoda RANDOM posiada większe prawdopodobieństwo wylosowania wartości z liczniejszej podgrupy, więc teoretycznie podgrupy powinny zachować proporcje. I uśredniając tak właśnie jest, jednakże dosyć często zdarza się, że któraś podgrupa istotnie różni się proporcjami od początkowych danych. Takie zjawisko powoduje, że sieci neuronowe uczą się w niewystarczający sposób nieproporcjonalnie małych podgrup, co powoduje większy błąd predykcji.

Pozostałe metody starają się zminimalizować występowanie takiego problemu. Metoda STRATIFIEDSAMPLING wręcz wymusza konkretną ilość danych w podzbiorach, dzięki czemu dane uczące jak i testowe posiadają takie same proporcje. Ta metoda jest szczególnie dobra, kiedy wiemy, lub możemy założyć, że posiadane dane odzwierciedlają proporcje populacji, wśród której dana sieć neuronowa będzie działać. Jednakże, kiedy nie możemy tego stwierdzić, być może bezpieczniejszym założeniem będzie, że średnia kilkukrotnych powtórzeń obliczeń z metodą RANDOM będzie bliżej praktycznie osiąganych rezultatów. Obie te metody uważamy za najoptymalniejsze, gdyż przez to iż nie duplikują danych, uczenie jest najszybsze. Zostały zaczerpnięte z książki “Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow”.

Równie dobrze w przypadku nieznanych proporcji podgrup w populacji, może się spisać metoda BOOTSTRAPPING. Z powodu że losuje dużo więcej razy, staje się nieco bardziej proporcjonalna. Jednakże pojawia się tutaj także zwiększone ryzyko szybszego przeuczenia sieci do części danych, właśnie z powodu powtarzania się części danych w zbiorze treningowym. Powstały przy tej metodzie zbiór uczący jest istotnie większy i nie da się zbyt precyzyjnie określić jego wielkości przed podziałem. O tej metodzie dowiedzieliśmy się od chata GPT, jednakże zaimplementowaliśmy sami.

RANDOMWITHIMPORTANCE - jest to autorska metoda, która najpierw losuje określoną ilość unikalnych danych, a następnie poprawia proporcje podgrup duplikując losowe dane spośród najmniejszych podgrup, aż osiągnie ich równomierny rozkład. Stosujemy właśnie taką metodę, żeby zasymulować nieznanym rozkładem zbioru testowego, nieznany rozkład populacji do której model sieci neuronowej mógłby trafić. Jednocześnie przyjmujemy założenie, iż prawdopodobieństwo natrafienia na obraz każdej cyfry będzie zbliżone do równomiernego. Owy zabieg niweluje skutki nieproporcjonalności podgrup, jednakże zwiększa ryzyko wystąpienia przeuczenia sieci do części danych. Jednakże, należy tutaj podkreślić, że taka metoda doboru parametrów wolniej się przeucza względem metody BOOTSTRAPPING, ponieważ ilość dodawanych w ten sposób danych jest istotnie mniejsza. (Dla poniższego badania różnica wynosiła około 50 tyś. danych, a sam RANDOMWITHIMPORTANCE dodał trochę ponad 6 tyś.)

Typ metody	Wynik (w procentach)
RANDOM	76,771
STRATIFIEDSAMPLING	78,270
BOOTSTRAPPING	71,486
RANDOMWITHIMPORTANCE	77

Podane wyniki mogą zaskakiwać wyjątkowo niską wartością dla BOOTSTRAPPING, jednakże jest to najprawdopodobniej spowodowane częściowym przeuczeniem sieci, które pogorszyło wynik końcowy. Ukazuje to jednocześnie mniejszy problem takiego zjawiska dla metody RANDOMWITHIMPORTANCE.

Ilość warstw neuronów (layers_dims)

O ilości warstw neuronów decyduje ilość liczb zapisana w parametrze layers_dims. Każda kolejna warstwa sieci neuronowej wydłuża proces uczenia się sieci. O ile podczas naszych testów natrafiliśmy na nieistotne zmiany w skuteczności predykcji wraz ze wzrostem ilości warstw sieci neuronowych, o tyle nie natrafiliśmy na taką ilość sieci, która powodowałaby pogarszanie się wyników. Mimo tego faktu nie wykluczamy takiej możliwości.

Wyniki zmian tego parametru jest silnie skorelowana z dobieraną ilością neuronów w konkretnej warstwie. Z tego też powodu, dla analizy tego parametru, przeprowadziliśmy badanie, na zasadzie, że każda kolejna warstwa sieci neuronowej będzie posiadać połowę neuronów warstwy poprzedniej. Wyjątkami jest sieć pierwsza i ostatnia, które zawierają kolejno wartości 784 i 10.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus.

Ilości neuronów w warstwach	Wynik (w procentach)
784, 392, 10	73,957
784, 392, 196, 10	82,719
784, 392, 196, 98, 10	83,705
784, 392, 196, 98, 49, 10	88,646

Jak widać, wzrost liczby neuronów za każdym razem poprawił skuteczność programu.

Ilość neuronów w warstwach (layers_dims)

Kolejnym parametrem jest ilość neuronów w poszczególnych warstwach. Przyjmowane są wszystkie liczby całkowite od 1 do 2147483647, jednakże ze względu na charakterystykę zastosowanych danych, zalecane są wartości w przedziale od 10 do 784. Ważne, aby pierwsza sieć neuronowa posiadała 784 neurony, a ostatnia 10 i takie też wartości należy zapisać przy zmianie tego parametru.

Zwiększanie liczby neuronów wydłuża istotnie czas pracy programu. W przypadku skuteczności, za duża liczba neuronów nie pogarsza osiąganych wyników, ale może przyspieszyć proces przeuczenia danych. Przez specyfikę naszych danych, wiele z nadmiarowych neuronów będzie posiadało wartości oscylujące w okolicy 0, co pokazuje na ich niską, wręcz pomijalną, istotność. Większość pikseli cyfr jest scentralizowana, więc dla sieci neuronowej najistotniejsze są tylko te piksele, które najczęściej różnią się w zależności od interpretowanej cyfry.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus.

Ilości neuronów w warstwach	Wynik (w procentach)
784, 250, 150, 50, 20, 10	90,667
784, 300, 180, 80, 40, 10	87,418
784, 392, 196, 98, 49, 10	88,211
784, 500, 250, 120, 60, 10	82,598

Różnica w skuteczności być może polega na architekturze warstw, gdyż np. w przypadku ostatniego zauważamy znacząca spadek między ilością neuronów w ostaniej warstwie ukrytej, a ostatnią warstwą. Generalnie mniejszą skuteczność mają sieci o większej ilości neuronów, może to wynikać z faktu, że nasz problem nie jest zbyt złożony, a generalnie, gdy mamy sieć o większej ilości neuronów to model ma tendencje do zbytniego uszczegóławiania, co w tym wypadku może być gorsze.

Współczynnik uczenia (learning_rate)

Współczynnik uczenia jest parametrem wpływającym bezpośrednio na zmianę wartości wag i biasów. Zalecane przez nas wartości tego parametru, w naszym przypadku, zawierają się w zakresie od 0,001 do 0,05. Zbyt mały współczynnik uczenia, może spowodować bardzo powolny proces uczenia, co w konsekwencji spowoduje, że bardzo późno osiągniemy minimum kosztu oraz trzeba kazać programowi wykonać istotnie więcej iteracji. Jednocześnie zbyt duża wartość tego parametru, może spowodować, że nasza sieć neuronowa ominie globalne minimum funkcji kosztu.

Istotnym czynnikiem przy wyborze wartości współczynnika uczenia jest także wybrana funkcja aktywacji. W przypadku zdecydowania się na funkcję relu, większe wartości nie są wskazane. Owa funkcja, potrafi odmówić współpracy, kiedy dany parametr jest za duży, przez co nie będzie poprawiał sieci, a skuteczność całego programu w odczytywaniu obrazów będzie oscylować w okolicy 10%, co oznacza, że program równie dobrze mógłby zgadywać.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus.

Wartość współczynnika uczenia	Wynik (w procentach)
0.05	9,826
0.01	91,452
0.005	88,653
0.001	60,468

W owym badaniu, relu odmówiło posłuszeństwa dla największej wartości. Najlepsza osiągnięta skuteczność predykcji została osiągnięta dla wartości współczynnika równego 0,01. Jest to spowodowane stosunkowo niewielką ilością iteracji które miały zostać powtórzone (200szt.). Nie do końca wiemy czy model w tym momencie zaczął się już przeuczać, czy może jednak jeszcze się poprawiał. Wiadomo natomiast widząc wyniki, że pozostałe dwie mniejsze wartości cały czas się poprawiały i dążyły do

osiągnięcia tego, lub nawet lepszego wyniku. Dla osiągnięcia lepszego wyniku, powinno odpowiednio modyfikować wartości parametru epoka, celem dopasowania wszystkich parametrów do siebie.

Epoka (epoka)

Parametr epoka w naszym programie pełni dwie funkcje, w zależności od parametru percent_of_validation_data.

Jeżeli parametr percent_of_validation_data posiada wartość równą 0, w tym momencie parametr epoka oznacza ilość powtórzeń procesu uczenia się sieci neuronowej. Wykona dokładnie taką ilość iteracji, ile napiszemy w tym parametrze.

Jednakże, jeżeli percent_of_validation_data będzie większe od 0, w tym momencie parametr będzie oznaczał minimalną ilość iteracji, jaką wykona program. W czasie wykonywania tych iteracji, nie będą sprawdzane dane walidacyjne, więc program zakończy obliczenia trochę wcześniej.

Dobranie odpowiedniej wartości parametru epoka jest trudne, gdyż zbyt mała wartość spowoduje, że sieć nie zdąży się nauczyć, natomiast zbyt duża wartość spowoduje przeuczenie sieci.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus przy parametrze percent_of_validation_data równym zero.

Ilość epok	Wynik (w procentach)
50	85,863
150	87,282
205	90,052
220	92,131
270	90,396

Występuje tutaj każde opisane zjawisko. Dane uczą się do około 220 epoki, po czym przeuczają się i ostatecznie dają gorsze rezultaty już przy 270 iteracji.

Procent danych walidacyjnych (percent_of_validation_data)

Zacznijmy od sposobu tworzenia przez nas zbioru walidacyjnego. Występuje to jedynie wtedy, kiedy parametr percent_of_validation_data jest większy od zera. Następnie zbiór uczący jest dzielony metodą STRATIFIEDSAMPLING na zbiór uczący i walidacyjny. Zbiór walidacyjny nie uczestniczy w ogóle w procesie uczenia. Jest wykorzystywany wyłącznie do sprawdzenia, jaką aktualny model powinien otrzymać skuteczność predykcji. Następnie jest on porównywany do ostatniego najlepszego modelu. W przypadku, kiedy jest gorszy już which_worse_prediction_stop_learning raz, wtedy kończymy uczenie zwracając najskuteczniejszy model.

Parametr ten musi mieć wartości od 0 do 1. W przeciwnym przypadku, najprawdopodobniej pojawi się błąd. W przypadku wpisania 0, zbiór walidacyjny nie powstanie.

Wpisywane wartości mają podobne problemy, jak stosunek danych testowych do uczących. Zbyt duża wartość spowoduje, że predykcja będzie niska, z powodu braku wystarczającej ilości danych. Jednakże jest to mimo wszystko wartość, która jest dosyć elastyczna do doboru, żeby uzyskiwała odpowiednie wartości. Warto także dodać, że im większą wartość wpiszemy, tym szybciej program będzie działał.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus.

Wartość parametru	Wynik (w procentach)
0.05	89,539
0.15	90,76
0.22	89,003
0.35	89,846

Ilość gorszych predykcji, które pominiemy (which_worse_prediction_stop_learning)

Omawiany parametr jest istotny dla całego programu. Wpisanie zbyt małej liczby, wraz z niewielką ilością epok, może skutkować bardzo szybkim uzyskaniem odpowiedzi, ale z niezadowalającym rezultatem. Dzięki zastosowanemu zabezpieczeniu, zwracany zostaje zawsze najskuteczniejszy model dla danych walidacyjnych. Nie ma więc żadnego ryzyka, przy wpisaniu bardzo dużej liczby. Dodatkowo oferuje sporą dowolność we wpisaniu liczby całkowitej (w zakresie od 1 do 2147483647). Teoretycznie, im większa wartość zostanie wpisana, tym lepszy model nam zwróci, jednakże odbędzie się to kosztem czasu, jaki trzeba poświęcić na uczenie. W praktyce, wielkie liczby poprawiają model na tyle nieznacznie, że błąd losowy odgrywa tutaj istotniejszą rolę.

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus.

Wartość parametru	Wynik (w procentach)
1	89,082
3	90,96
5	90,824
8	90,774

Można tutaj zauważyć, że brak tolerancji na błędy faktycznie pogarsza wyniki, jednakże w przypadku wartości od 3 włącznie, różnice wartości są nieistotne.

Inicjalizacja parametrów (initzializing_method)

W projekcie zastosowaliśmy 3 rodzaje inicjalizacji parametrów:

- Inicjalizacja HE – wagi losowane są z rozkładu normalnego o średniej 0 i odchyleniu standardowym $\sqrt{\frac{2}{n}}$, gdzie n to liczba neuronów w warstwie wejściowej. Wg literatury inicjalizacja HE powinna dobrze współgrać z funkcją aktywacji Relu.

$$W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

Kaiming Initialization

- Inicjalizacja Xavier Glorot - wagi są losowane z rozkładu jednostajnego ciągłego z przedziału (-u,u), gdzie u liczone jest jako $\sqrt{\frac{6}{n_l+n_{l+1}}}$, gdzie nl to liczba neuronów w warstwie l
- Inicjalizacja Random – wagi są ustawiane losowo

Oto przykładowe zestawienie wartości parametru z osiągniętymi przez nie wynikami skutecznego zinterpretowania obrazów. Zastosowaliśmy tutaj zasadę ceteris paribus.

Metoda	Wartość (w procentach)
HE	88,778
RANDOM	11,255
XAVIER_GLOROT	89,954

Zaskakująco niską wartość wykazała tutaj metoda RANDOM, jednakże jest to spowodowane stosunkowo niską ilością iteracji i wartością learning_rate, gdzie metoda RANDOM wymaga istotnie większych obu tych wartości.

Uwagi

Projekt nie został wykonany w pełni samodzielnie. Uzyskaliśmy pomoc w wykonaniu projektu z dwóch źródeł: sztucznych sieci neuronowych tzw. Chat GPT, oraz osoby prowadzącej koło naukowe Mentor, którego część z nas jest aktywnymi uczestnikami.

Funkcje w pełni lub prawie w pełni niesamodzielne

extend_array – funkcja zamieniająca macierz np.array na macierz z zerami i jedynkami. Wykorzystywane do modyfikacji macierzy z cyframi definiującymi obraz na odpowiedź zerojedynkową.

save_array_as_csv – funkcja zapisująca np.array do pliku .csv. Wielokrotnie wykorzystana do debugowania.

translate_matrix_of_probabilities_to_matrix_of_answers – funkcja tłumacząca macierz prawdopodobieństw na macierz odpowiedzi.

matrix_comparison – funkcja porównująca macierze odpowiedzi celem obliczenia skuteczność predykcji.

Funkcje zoptymalizowane, upiększone lub w małym stopniu napisane niesamodzielnie

activation_function_forward – chat gpt dodał komentarze

linear_backward – udzielona pomoc przez prowadzącego koło

Autorzy:

Radosław Mocarski

Tomasz Zapart

Anna Rubin

Mateusz Strojek

Adrian Żyła