

# Wydział Informatyki i Telekomunikacji

**Model rozpoznawania hieroglifów z użyciem konwolucyjnej sieci neuronowej (CNN)**

# Projekt z przedmiotu Programowanie Języka Naturalnego

## Praca zbiorowa, autorzy:

Maciej Adamczyk

Aleksandra Białas

Tomasz Bortacki

Krzysztof Kasprzak

Studia niestacjonarne, r. 2023/2024, semestr 8 (letni)

1. Cel i zakres pracy

Celem pracy jest opracowanie procesu trenowania konwolucyjnej sieci neuronowej (CNN) klasyfikującej egipskie hieroglify. Praca obejmuje przygotowanie zbioru hieroglifów, zdefiniowanie architektury sieci neuronowej, przeprowadzenie jej treningu oraz zastosowanie i sprawdzenie wpływu technik usprawniających trening modelu.

1. Opis problemu

W ciągu trzech tysiącleci istnienia kolejnych egipskich dynastii powstało blisko 6 tysięcy udokumentowanych hieroglifów. Mogły one oznaczać brzmienie spółgłosek (fonemy - wyróżnia się tutaj fonemy jedno- dwu i trzyliterowe), określniki (determinatywy, opisujące jak odczytywać znaczenia) i ideogramy (logogramy - przedstawiające pojęcia bez użycia liter czy znaków fonetycznych).

Język egipski należał do języków afroazjatyckich, w zapisie hieroglifami nawet z użyciem fonemów pomijano samogłoski.

Wybrany do programu zbiór danych z obrazkami hieroglifów opiera się głównie na ideogramach (obrazkach), z rzadka na określnikach pełniących często funkcje przyimków ( np. “among” - pośród, “at” - przy). Nie zawiera fonemów, które szczególnie przy tych dwu i trzy literowych wymagałyby implementacji dodatkowo łączenia kilku znaków w jeden. Nie zawiera też określników opisujących jak czytać cały zespół znaków (czyli np. kartuszy czyli “węzłów” oplatających całą sekwencję wskazujących na imię władcy).

Założeniem wyboru danych jest zasada “jeden hieroglif – jedno znaczenie”, tak by odczyt hieroglifów był jak najbardziej jednoznaczny.

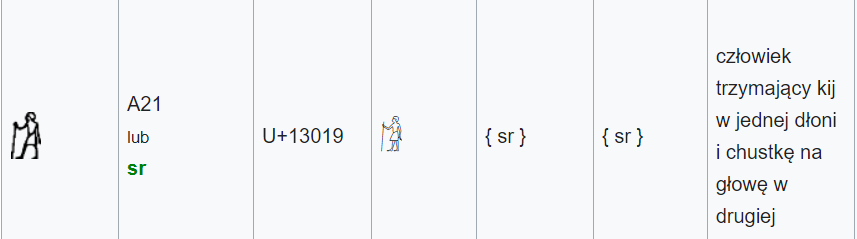
Innym wyzwaniem w odczytywaniu hieroglifów z oryginalnych transkrypcji jest kierunek odczytu pisma, który może się zmieniać w zależności od tego, w którą stronę zwracają twarz postacie przedstawione na hieroglifach (od lewej, od prawej lub z góry na dół). Dodatkowo niemal każdy znak w oryginalnych inskrypcjach miał swój lustrzany wariant zależny od tego, w którą stronę odczytuje się dany ciąg hieroglifów. W tym programie uproszczono kwestię kierunku odczytu hieroglifów, zakładając rozpoznawanie hieroglifów w trybie jeden do jeden, w kolejności wprowadzania, rezygnując z wieloznakowych hieroglifów.

Brytyjski egiptolog, sir Alan H. Gardiner spośród tysięcy udokumentowanych hieroglifów opracował w trzech wydaniach (1927, 1950, 1957) skróconą listę zawierającą 736 hieroglifów pogrupowanych w 26 kategorii, uważaną za jedną z najbardziej funkcjonalnych katalogów hieroglifów. Lista ta została później ujęta i poszerzona przy opracowaniu czcionki blokową *Unicode Egyptian Hierogliphs* (Unicode version 5.2, 2009) zawierającej 1071 znaków unicode.

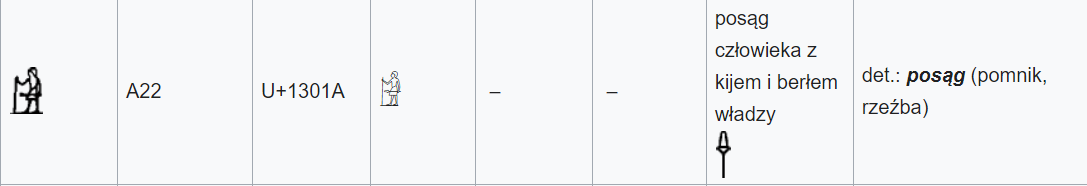
Lista Gardinera, mimo że stanowi ogromne uproszczenie i zawężenie zbioru 6 tysięcy hieroglifów, może w odczycie z obrazków tworzyć pewne dwuznaczności istotnie zmieniające znaczenie odczytywanego tekstu.

Przykład 1:

znaczenie : ***osoba starsza*** (najstarszy, starszy)

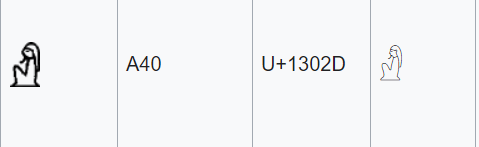
znaczenie : ***dostojnik*** (wysoki urzędnik), ***wielmoża, osoba wpływowa*** (człowiek wpływowy), ***możnowładca*** (człowiek z wyższych sfer) lub ***osoba zaufana*** (w tym: przyjaciel – bliski znajomy, dobry kolega; dworzanin – zausznik lub powiernik faraona)

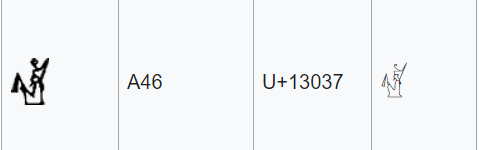
Przedstawienie postaci kroczącej z “chustką” na głowie (w praktyce dodatkowa jedna kreska) i brak rozdwojenia kija zmienia znaczenie słowa. Dodanie jeszcze jednej kreski na poziomie pasa postaci zmienia znaczenie hieroglifu na “rzeźba”.

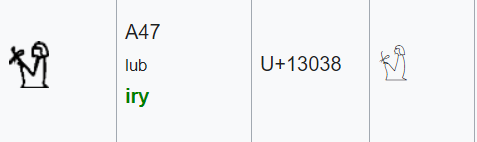


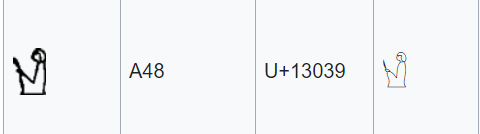
Przykład 2:

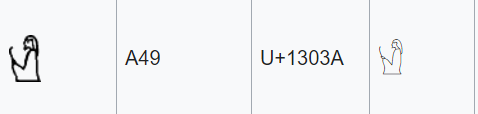
Jeszcze bardziej sprawy komplikują się przy ideogramach przedstawiających postacie siedzące:

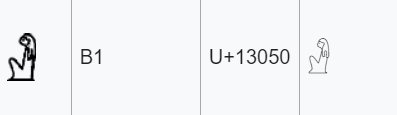
 znaczenie: **bóg**

 znaczenie: **król Dolnego Egiptu**

 znaczenie: **owczarz** **z patykiem z przydatkiem**

znaczenie: **Mężczyzna** ￼**z nożem**, ale też **“należeć, odnosić się”**

 znaczenie: **Syryjczyk z patykiem, obcokrajowiec**

 znaczenie: **kobieta**

Jak widać na wspomnianych wyżej grafikach, przy niestarannym zapisie hieroglifu albo naruszeniu inskrypcji przez czas czy niezbyt dokładnego zdjęcia, można by pomylić np. faraona z owczarzem, boga z Syryjczykiem a nożownika z kobietą.

Wniosek: mały element obszaru obrazka (kreska lub dwie) potrafią zmienić znaczenie hieroglifu, a liczba hieroglifów w liście Gardinera w pewnej mierze bierze się z odróżniania takich niuansów, z którymi program odróżniający obrazki może mieć problem (szczególnie przez zmienny styl hieroglifów, różniący się nie tylko epokami ale także indywidualnym “pismem” skrybów). W związku z tym należało wyszukać zbiór danych o możliwie najmniejszym ryzyku różnic w rozpoznawaniu znaków.

Jeszcze innym problemem jest zmiana znaczenia hieroglifów z czasem w trakcie tysiącleci istnienia kolejnych dynastii starożytnego Egiptu.

Przykład: hieroglify oznaczające odrębnie “uderzać” (człowiek trzymający kij) i “starzec” (człowiek oparty o laskę) były używane przez pewien czas zamiennie. 

Ze względu na liczne historyczne przesunięcia znaczeniowe w programie skupiono się na rozpoznawaniu hieroglifów przez program, a nie interpretacji i przypisywania im znaczenia.

Zamiast pełnej listy hieroglifów Gardiera zdecydowano się tej pracy znaleźć i wybrać gotowy zbiór danych będący wyciągiem z ww. listy znaków Gardiera. Poza selekcją podobnych do siebie znaków taki zbiór (tutaj zastosowany z 95 klasami zamiast 736) upraszczał działanie programu przez zawężenie liczby wzorców z którymi każdy obrazek (a raczej wektor z obrazka) miał być porównywany.

1. Opis i działanie programu

Jak wspomniano we wcześniejszym rozdziale, w programie zastosowano zbiór danych z ograniczoną liczbą klas hieroglifów, Egyptian Hieroglyphic Dataset, gdzie znaki wyselekcjonowali Sallar Khan z [Sir Syed University of Engineering and Technology SSUET](https://www.bing.com/ck/a?!&&p=c5dac84ab729065cJmltdHM9MTcxNzIwMDAwMCZpZ3VpZD0xZWU0YWY0ZC04OWNmLTY0ZDEtMWYzNi1iYjA1ODhiYzY1ZmUmaW5zaWQ9NTc3Ng&ptn=3&ver=2&hsh=3&fclid=1ee4af4d-89cf-64d1-1f36-bb0588bc65fe&psq=SSUET&u=a1aHR0cHM6Ly9wbC53aWtpcGVkaWEub3JnL3dpa2kvU2lyX1N5ZWRfVW5pdmVyc2l0eV9vZl9FbmdpbmVlcmluZ19hbmRfVGVjaG5vbG9neQ&ntb=1) i Waleed Bin Umer. Dane są dostępne na licencji serwisu Kaggle.com z dozwolonym użytkiem min. do zastosowań akademickich.

Rzeczony zbiór danych składa się z dwóch części: trenującej i testowej; w każdej z nich znajdywały się pliki graficzne (z nazwami będącymi znaczeniem danego hieroglifu) i odpowiadającymi im plikami xml będącymi zapisem metadanych.

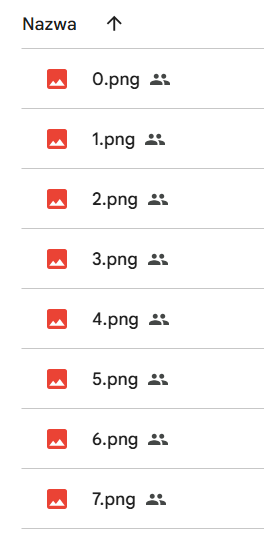
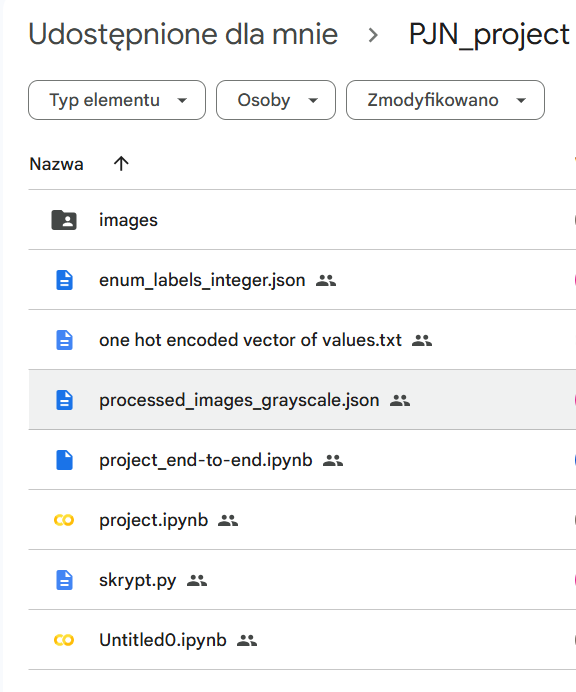
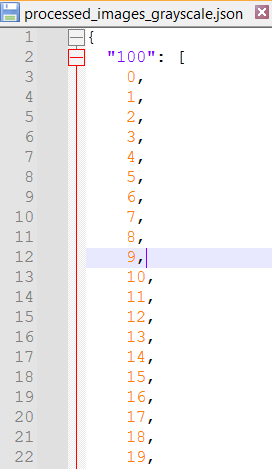
Pliki graficzne miały rozmiary 770 x 660 pikseli.

Przed zastosowaniem plików graficznych bezpośrednio w programie poddano je obróbce, sprowadzając do wspólnej wielkości 50x50, usuwając zbędne niezapisane pole oraz ograniczając głębię kolorów do skali szarości. Dzięki temu ograniczyliśmy rozmiar danych z 17 MB do 4 MB, zakładając szybsze działanie programu.

Następnie wygenerowano plik JSON mapujący nazwy hieroglifów (etykiety, roboczo nazywając je potocznie “labelkami”, by nie mylić je z obrazkami hieroglifów) z nowymi nazwy obrazków (będące identyfikatorem z kolejnymi liczbami porządkowymi) na zasadzie “nazwa klasy hieroglifów” : [ nazwy id obrazków z hieroglifami z tej klasy].

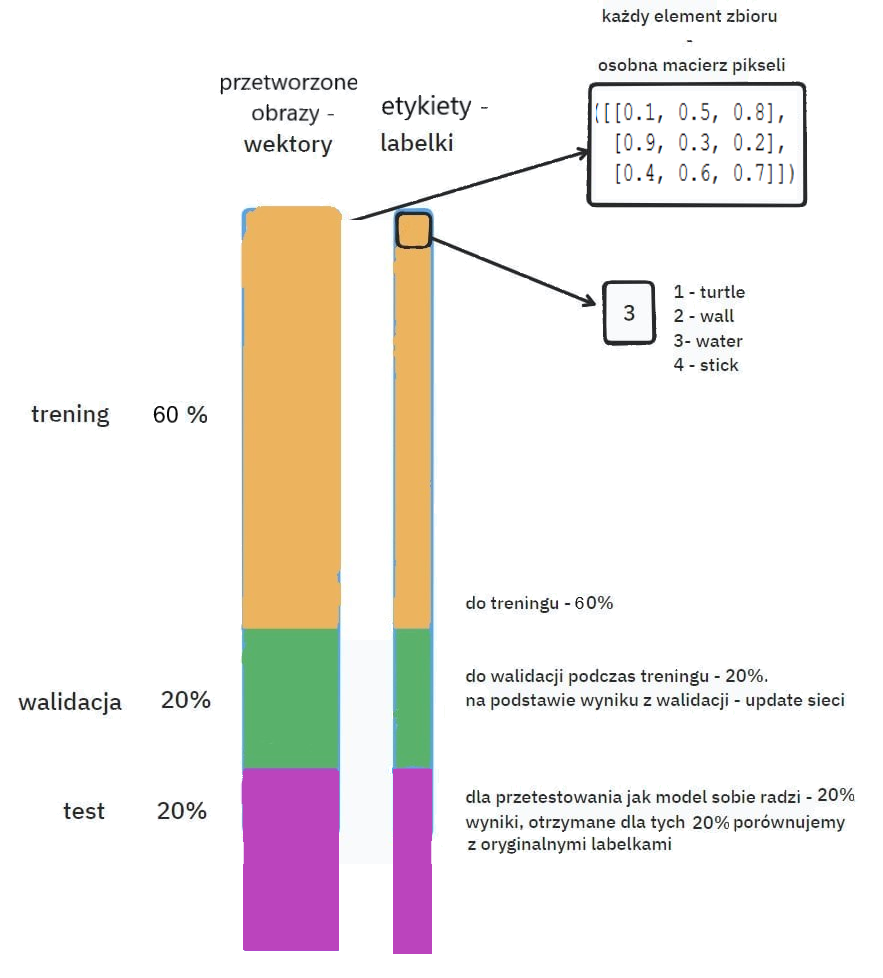
W projekcie obrazki reprezentujące hieroglify znajdują się w folderze \imgages

plik JSON mapujący nazwy klas (etykiet - “labelek”) z nazwami obrazków (identyfikatorami) nosi nazwę processed\_images\_grayscale.json

Z tak otrzymanym plik json przystąpiono do dalszego przekształcania danych obrazków hieroglifów i ich etykiet do takiej formy, by najbardziej nadawały się do uczenia maszynowego, tj. do ich dalszej zamiany w liczby.

Aby porównać zgodność zadanego obrazka ze wzorcem, obrazki (już uprzednio przekształcone do skali szarości), zostały odczytane za pomocą **funkcji io.imread** z biblioteki skimage a następnie przekształcone za pomocą funkcji img\_as\_float z biblioteki **skimage.util** do matrycy liczb zmiennoprzecinkowych (floatów)

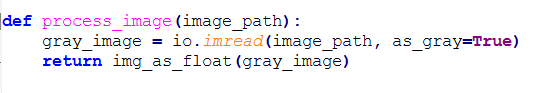


Wycinki kodu:

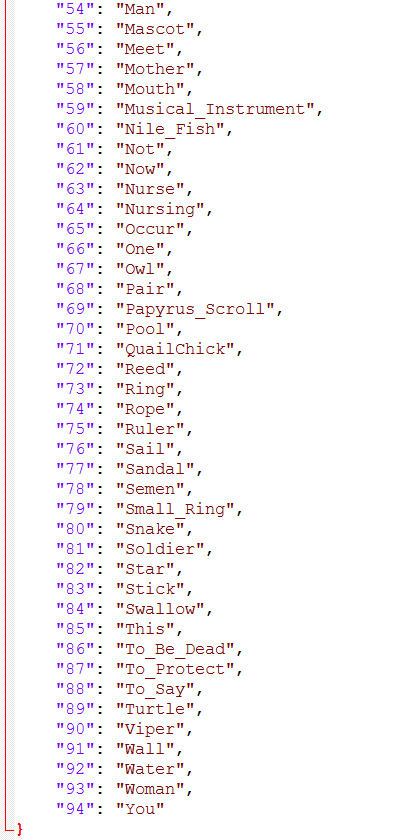
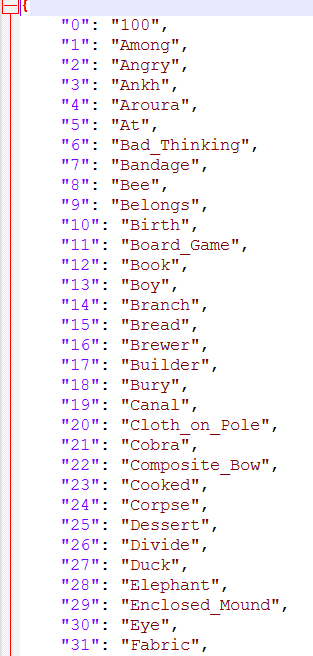
[...]



[...]



Następnie powyższej funkcji użyto wewnątrz funkcji do utworzenia sekwencji etykiet (nazw klas hieroglifów) przedstawionych w kodowaniu one-hot.

 Nazwom klas hieroglifów - etykietom - przyporządkowano kolejne numery porządkowe. Spis tych numerów zawarty jest w pliku o nazwie enum\_labels\_integer.json.

Następnie przyjęto, że nowe liczby porządkowe dla etykiet będą przekształcone w kodowanie, one hot wyglądające w ten sposób:

1 - [1,0,0,0 ...n]

2 - [0,1,0,0...n]

3 - [0,0,1,0...n]

[...]

n- [0,0,0,0 ...1]

Gdzie n oznaczało liczbę wszystkich wykorzystanych etykiet (nazw klas hieroglifów).

Tak otrzymane nazwy etykiet w kodowaniu one-hot wstawiano do nowej macierzy, powielając nowe nazwy etykiet w formacie one-hot tyle razy, ile kolejno występowały obrazki reprezentujące daną klasę hieroglifu. Inaczej mówiąc, przykładowo dla 3889 obrazków, nowo utworzona tabela zawierała 41 pierwszych one hotów w postaci [1,0,0...n] powstałych z zamiany etykiety klasy hieroglifu o znaczeniu “100” (słownie sto), następnie tabela zawierała 41 one-hotów w postaci [0,1,0,0...n] powstałych z zamiany etykiety klasy hieroglifu o znaczeniu “among” (‘pośród”), następnie 41 one-hotow w postaci [0,0,1,0...n] powstałych z zaiany etykeity klasy hieroglifu o znaczeniu “ankh” (egipski znak-amulet w kształcie krzyża z pętlą zamiast górnej belki, symbol życia) itd.

{

[1,0,0,0....n], [1,0,0,0....n], [1,0,0,0....n], [1,0,0,0....n], [1,0,0,0....n], ... [1,0,0,0....n]

[0,1,0,0...n], [0,1,0,0...n], [0,1,0,0...n], [0,1,0,0...n], [0,1,0,0...n], .... [0,1,0,0...n]

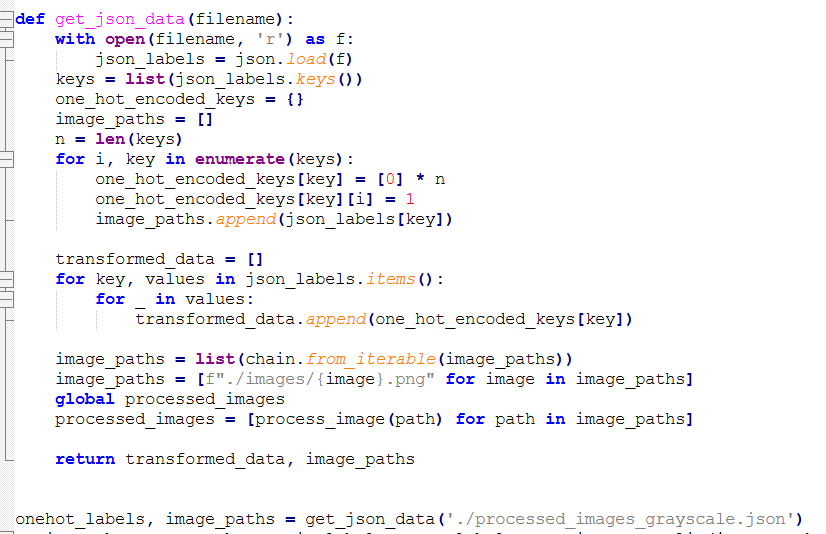
[0,0,1,0...n], [0,0,1,0...n], [0,0,1,0...n], [0,0,1,0...n], [0,0,1,0...n], .... [0,0,1,0...n],

[...]

[0,0,0,0 ...1], [0,0,0,0 ...1], [0,0,0,0 ...1], [0,0,0,0 ...1], [0,0,0,0 ...1], ... [0,0,0,0 ...1] }

N x M

Gdzie n – liczba etykiet, m – liczba obrazków dla tych etykiet



W kodzie wyżej opisana tabela one-hoterów zostaje zwrócona z funkcji get\_json\_data jako transformed\_data.

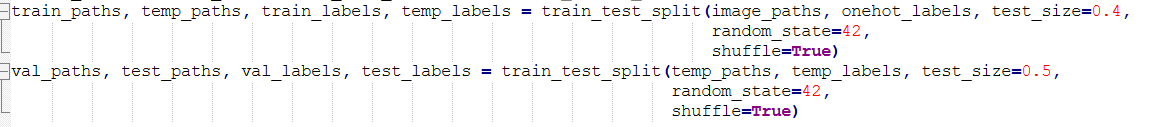
Wspomniane wcześniej macierze floatów są zapisywane w zmiennej processed\_images po użyciu funkcji process\_image.

Następnym krokiem było podzielenie zbioru danych dla celu przeszkolenia, przetestowania i walidacji.

W tym celu użyto funkcji **train\_test\_split** z biblioteki **sklearn.model\_selection.**

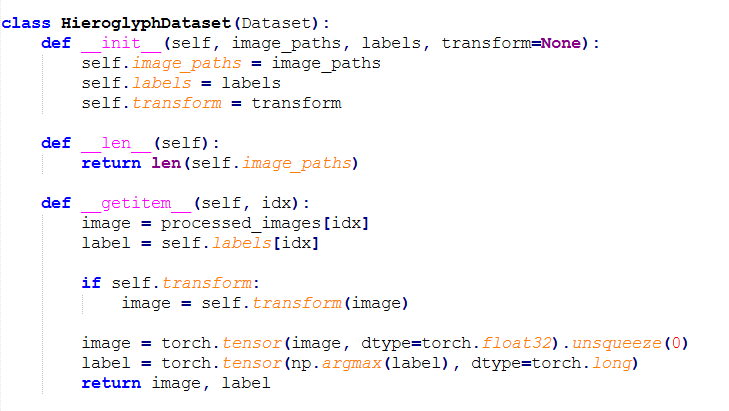


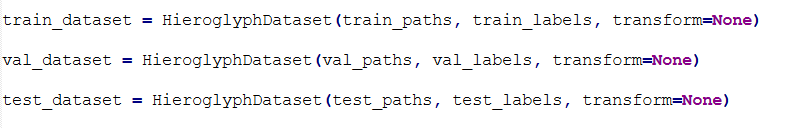
[...]



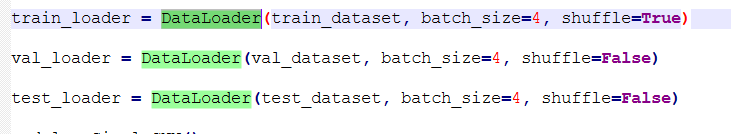
Na początku zwrócono ścieżki (z obrazkami) i etykiety do szkolenia modelu, z tymczasowymi ścieżkami i etykietami, które następnie zostały podzielone na ścieżki i etykiety odpowiednio do walidacji i testowania.

W obu przypadkach wybrano wymieszanie danych przed podziałem (parametr shuffle ustawiony na true).

[...]

 Następnie powiązano ze sobą ścieżki do obrazów i etykiety w zbiory danych odpowiednio do trenowania, walidacji i testowania. Jak widać, obrazki hieroglifów są tutaj reprezentowane jako matryce liczb zmiennoprzecinkowych.

Następnie przy użyciu biblioteki PyTorch tworzone są obiekty do ładowania zbiorów danych – **DataLoader** i iterowaniu po nich (po jednym DataLoaderze na każdy typ zbioru danych). Zbiór danych dla uczenia jest mieszany, pozostałe nie.



Następnym krokiem jest zainicjowanie modelu – prostej konwolucyjnej sieci neuronowej. W nowo utworzonej klasie **SimpleCNN** stworzono za pomocą **nn.Conv2d()** dwie warstwy splotowe (konwolucyjne) 2D, które przy użyciu tzw. jąder konwolucji (splotu), tj. filtrów poruszających się po obrazie celem wyszukiwania różnic.

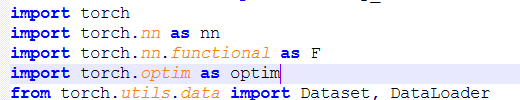
W pierwszej warstwie konwolucyjnej zastosowano 1 kanał wejścia, 16 kanałów wyjścia, rozmiar filtra 3 i dodawanie jednego dodatkowego piksela. W drugiej było 16 kanałów wejścia, 32 wyjścia, reszta bez zmian.

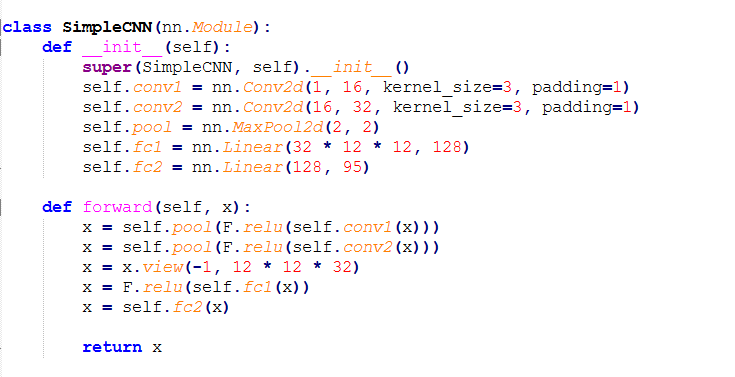
Następnie wywołana zostaje funkcja **MaxPool2d**() która wykonuje operację maksymalnego łączenia **(ang. max pooling)** na dwuwymiarowych danych wejściowych, takich jak obrazy. Tutaj **kernel size** (rozmiar okna poruszającego się po obszarze w poszukiwaniu maksymalnych wartości) ustawiono na 2 i krok też na 2.

Kolejnym krokiem było stworzenie dwóch w pełni połączonych warstw liniowych (gęstych) za pomocą funkcji **nn.Linear()**. W pierwszej warstwie liniowej stworzono 32 \* 12 \* 12 neuronów w warstwie wejściowej i 128 w wyjściowej, w drugiej – 128 na wejściu, 95 na wyjściu. Po każdym przejściu przez splot wykonywano **ReLU()**, czyli funkcję jednostki liniowej wyprostowanej elementarnie, która wyprowadza bezpośrednio sygnał wejściowy, jeśli jest dodatni, w przeciwnym razie wyświetli zero. Dzięki jej zastosowaniu model szybciej się uczy i jest łatwiejszy do wytrenowania, zerując wartości ujemne. Po tym zabiegu zastosowano **self.pool()** dla łączenia danych do danych wejściowych, co zmniejsza wymiar danych.

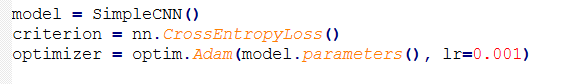
Za pomocą **X.view()** zwraca się tensor o zadanym kształcie bez zmiany jego danych.

Dane trafiają do pierwszej warstwy liniowej gdzie są traktowane funkcją jednostki liniowej wyprostowanej elementarnie, i do drugiej, już bez użycia **ReLu**.





[...]

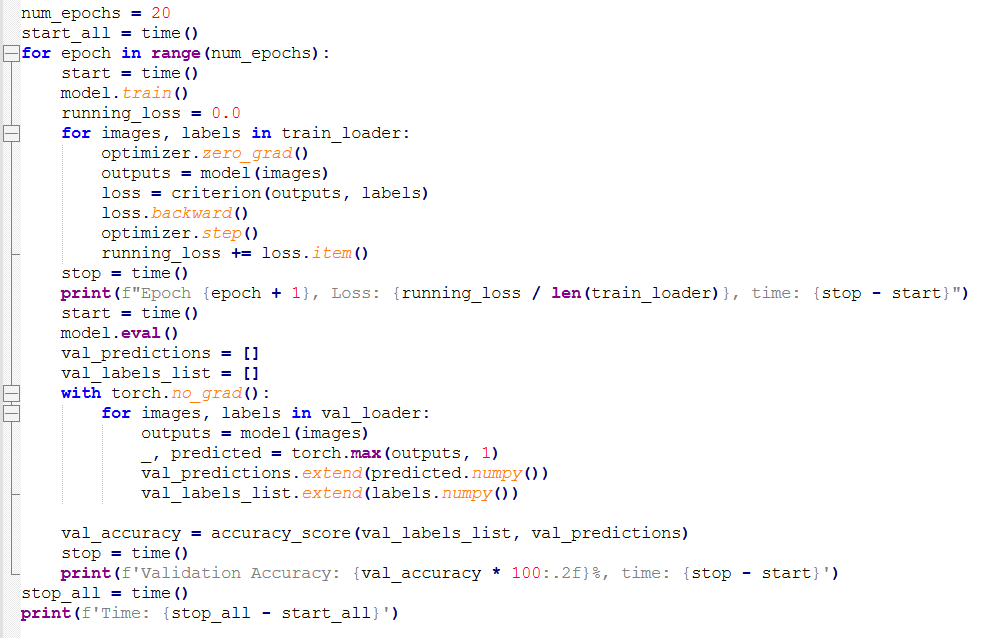


Po stworzeniu prostej warstwy konwolucyjnej zdefiniowano funkcję strat **CrossEntropyLoss()** i optymizator **Adam**.

Funkcja **CrossEntropyLoss()** przyjmuje surowe wartości wyników (logitów) przed funkcją softmax i prawdziwe etykiety klas, oblicza wartość prawdopodobieństwa należenia zadanych logitów do klas przy użyciu **softmax** (skalującej liczby do prawdopodobieństw), a następnie mierzy różnice między przewidywanymi prawdopodobieństwami a rzeczywistymi etykietami.

Zastosowano optymizator **optim.Adam(),** czyli **Adaptive Moment Estimation**, dostosowujący współczyniki uczenia się dla różnych parametrów na podstawie średnich gradientów i ich kwadratów.

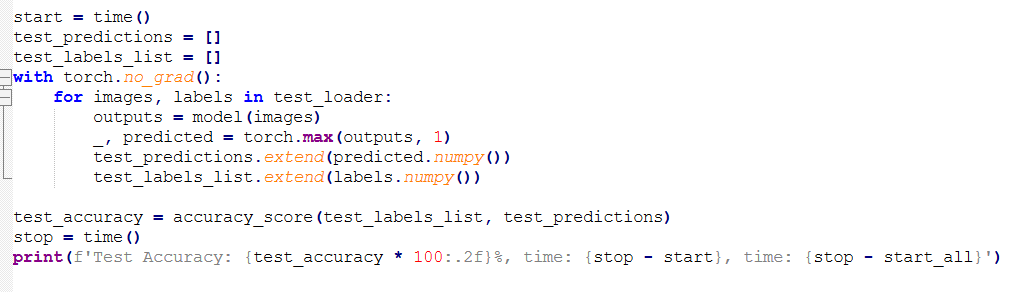
Liczba epok została ustawiona na 20, przy każdym kolejnym przebiegu zeruje się gradienty z optimizer.zero\_grad(), przepuszcza dane wejsciowe przez model, oblicza straty : **loss = criterion (output, labels)** jak i gradienty przez **loss.backward()** i aktualizuje się wagi – **optimizer.step()**. Straty sumuje się w zmienej **running\_loss** a następnie wypisuje się na ekran wynik dzielenia bieżącej straty przez długość **train\_loadera**. W każdej epoce po traningu następuje ewaluacja modelu i jego walidacja



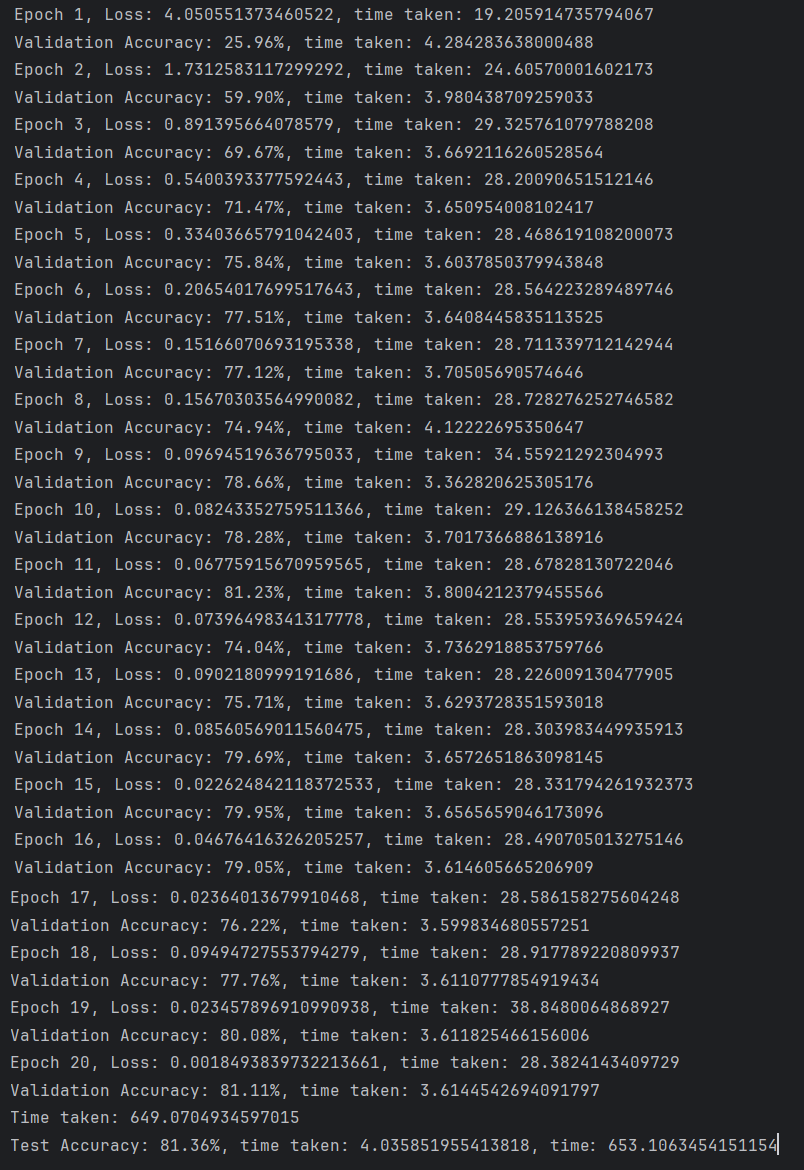
W walidacji stosuje się wyrażenie **torch.max(output, 1)** które służy do obliczania maksymalnej wartości oraz indeksu maksymalnej wartości wzdłuż osi zadanego tensora. Wartości maksymalne

Do obliczenia dokładności walidacji stosuje się tutaj funkcję **accuracy\_score** z modułu **sklearn.metrics**, która jako argumenty bierze liczbę wystąpień rzeczywistych etykiet **val\_labels\_list** oraz liczbę wystąpień spodziewanych etykiet **val\_predictions** i oblicza udział poprawnych przewidywań w całości przewidywań.

Dokładność walidacji jest przy każdym obiegu również wyświetlana użytkownikowi na ekranie.



Podobnie do walidacji zastosowano metodę test accuracy, z tym że tutaj dla danych testowych na test\_loaderze.



Powyżej przykładowy wydruk działania skryptu skrypt.py. Jak widać, z każdym obiegiem zmniejsza się strata i rośnie dokładność walidacji.

1. Wykorzystanie Callbacków w celu usprawnienia pętli treningowo-walidacyjnej

Aby zwiększyć wydajność trenowania modelu zastosowano callbacki. Wpływają znacząco na optymalizację wykorzystania zasobów obliczeniowych oraz skrócenie czasu treningu. Pomagają monitorować i kontrolować proces uczenia się poprzez dynamiczne dostosowywanie różnych aspektów trenowania.

Przetrenowanie występuje, gdy model zbyt dobrze dopasowuje się do danych treningowych, tracąc zdolność do generalizowania na nowe, niewidziane wcześniej dane.

Learning rate jest jednym z najważniejszych hiperparametrów w procesie trenowania modeli. Jego wartość wpływa na szybkość i stabilność uczenia się modelu. Duży learning rate powoduje szybkie uczenie się modelu, ale może prowadzić do niestabilnych wyników, ze względu na ryzyko przeskakiwania modelu przez minima funkcji kosztu, co może uniemożliwić osiągnięcie optymalnych rozwiązań. Mały learning rate przekłada się na powolne i dokładne uczenie modelu, co zwiększa szanse na znalezienie globalnego minimum, ale może znacząco spowolnić proces uczenia.

W projekcie funkcją kosztu jest **categorical cross-entropy**. Mierzy różnicę pomiędzy rozkładem prawdopodobieństwa przewidywanym przez model, a rzeczywistym rozkładem prawdopodobieństwa klas.

**EarlyStopping** to technika, monitorująca stratę na danych walidacyjnych i zatrzymująca trening, w momencie gdy metryka przestaje się poprawiać przez określoną liczbę epok. Oszczędza czas i zasoby obliczeniowe. Chroni model przed przetrenowaniem, czyli sytuacją, gdy model zbyt dobrze dopasowuje się do danych treningowych, tracąc zdolność do generalizowania na nowe dane. Jeśli trening zostaje przerwany, przywracana jest wersja modelu, która osiągnęła najniższą stratę na danych walidacyjnych. Technikę implementuje klasa EarlyStopping.

**ReduceLROnPlateau Callback** jest techniką dynamicznie zmniejszającą learning rate, gdy strata walidacyjna przestaje się poprawiać przez określoną liczbę epok. Umożliwia szybki start uczenia, a następnie bardziej precyzyjne dostrojenie, gdy model utknie w lokalnych minimach lub na płaskich odcinkach funkcji kosztu. Po każdej epoce monitorowana jest strata walidacyjna i jeśli nie stwierdzono poprawy przez określoną liczbę epok, następuje zmniejszenie learning rate o połowę. ReduceLROnPlateau został zaimportowany z biblioteki PyTorch.

Poniżej przestawiono fragmenty kodu odpowiadające za implementację callbacków w projekcie.

lr = 0.001

num\_epochs = 20

device = "cuda"

model = SimpleCNN()

model.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=lr)

scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(optimizer=optimizer, factor=0.5, patience=2)

stopper = EarlyStopping(patience=5, min\_delta=0.001, path="model\_checkpoint.pt", verbose=False)

Dla **EarlyStopping** ustawiono **Learning Rate** na 0.001 (lr), liczbę epok na 20 (num\_epochs), tolerancję modelu na brak poprawy straty walidacyjnej na 5 epok (patience), i parametr określający minimalną wymaganą zmianę w monitorowanej metryce uznawaną za poprawę na 0.001 (min\_delta).

W przypadku **ReduceLROnPlateau** ustawiono tolerancję modelu na brak poprawy straty walidacyjnej na 2 epoki (patience), mnożnik aktualnego współczynnika uczenia (learning rate) w przypadku braku poprawy uczenia na 0.5 (factor). Jako optymalizator wykorzystano algorytm Adam (Adaptive Moment Estimation).

class EarlyStopping:

def \_\_init\_\_(self, patience=5, min\_delta=0, path="model\_checkpoint.pt", verbose=False):

self.patience = patience

self.min\_delta = min\_delta

self.verbose = verbose

self.min\_validation\_loss = np.inf

self.counter = 0

self.early\_stop = False

self.best\_model = None

name\_salt = str(time()).split(".")[0][-4:] # add time salt

self.path = path + "\_" + name\_salt

def \_\_call\_\_(self, val\_loss, model):

if val\_loss < self.min\_validation\_loss:

self.save\_checkpoint(val\_loss, model)

self.min\_validation\_loss = val\_loss

self.counter = 0

elif val\_loss >= (self.min\_validation\_loss + self.min\_delta):

self.counter += 1

if self.counter >= self.patience:

self.early\_stop = True

Omówienie algorytmu:

Porównaj stratę walidacyjną z bieżącej epoki z historycznie najniższą stratą walidacyjną.

Jeśli bieżąca strata jest niższa niż historyczna strata:

* zapisz bieżący stan modelu
* zaktualizuj najniższą stratę
* wyzeruj licznik epok z wyższą stratą niż najniższa

Jeśli bieżąca strata jest większa niż historyczna strata:

* zwiększ licznik
* jeśli licznik osiągnie lub przekroczy parametr "patience",
* ustaw flagę "early\_stop" na True. To zakończy pętlę treningową.

W dalszej części klasy EarlyStopping zaimplementowano metodę **save\_checkpoint** zapisującą stan modelu do pliku binarnego i metodę **load\_best\_model** wczytującą stan modelu z pliku binarnego:

def save\_checkpoint(self, val\_loss, model):

if self.verbose:

print(f"Validation loss decreased ({self.min\_validation\_loss:.6f} --> {val\_loss:.6f}). Saving model ...")

torch.save(model.state\_dict(), self.path)

self.best\_model = model

def load\_best\_model(self):

if self.best\_model is not None:

self.best\_model.load\_state\_dict(torch.load(self.path))

self.best\_model.eval()

if self.verbose:

print("Best model has been loaded.")

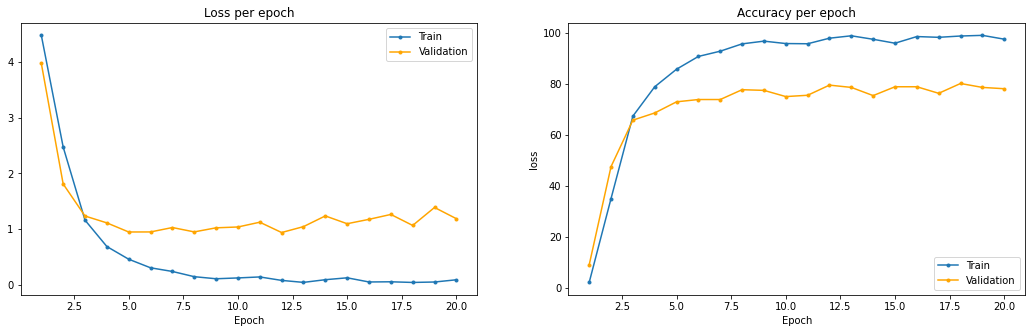
return self.best\_model

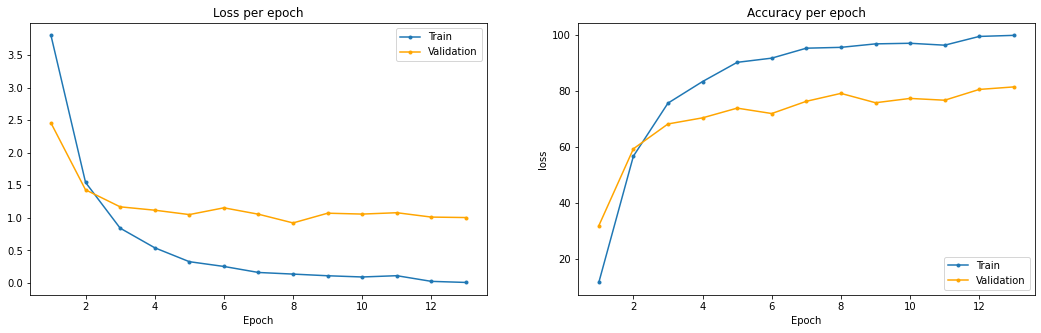
else:

print("There is no best model for current training")

return None

Przeprowadzono proces treningu modelu bez zastosowania callbacków i z ich wykorzystaniem. Rezultaty przedstawiono na wykresach 1.1 i 1.2.

*Wykres 1.1. Proces treningu bez wykorzystania callback’ów*



*Wykres 1.2. Proces treningu z wykorzystania callback’ów*

Na wykresie 1.1 po ósmej epoce strata na walidacyjnych danych zaczyna rosnąć, co może wskazywać na przetrenowanie modelu, natomiast trening trwa przez całe dwadzieścia epok. W treningu z wykorzystaniem callbacków (wykres 1.2) trening zostaje przerwany po trzynastej epoce, a model zostaje przywrócony do stanu z ósmej epoki.

1. Testowanie i omówienie metryk

Testowanie pozwala ocenić wydajność modelu na danych testowych, nie używanych w procesie treningu, co pozwala oszacować, jak dobrze model generalizuje wiedzę na nowych danych. Umożliwia wykrycie ewentualnych problemów lub błędów w modelu, takich jak przetrenowanie (**overfit**) i niedouczenie (**underfit**).

W procesie oceny modeli uczenia maszynowego stosowane są metryki, prezentujące ich wydajność w zadaniach klasyfikacyjnych. Metryki mają swoje specyficzne zastosowanie i znaczenie.

By obliczyć metryki wydajności modelu klasyfikacji wieloklasowej, w projekcie wykorzystano bibliotekę **sklearn**.**metrics**.

Dokładność (**accuracy**) obliczana jest za pomocą funkcji **accuracy\_score** z **sklearn**.**metrics**. Mierzy odsetek poprawnie sklasyfikowanych przypadków ze wszystkich przypadków.

**Precyzja** (**precision**) wykorzystuje funkcję **precision\_score z sklearn.metrics**. Mierzy odsetek przypadków przewidzianych jako pozytywne, które naprawdę są pozytywne.



**Czułość** (**recall**) wykorzystuje funkcję **recall**\_**score** z **sklearn**.**metrics** i mierzy odsetekpozytywnych przypadków przewidzianych poprawnie **spośród wszystkich prawdziwych pozytywnych przypadków.**



F1 Score obliczany jest za pomocą funkcji **f1\_score** z sklearn.metrics i jest **średnią harmoniczną precyzji i czułości.**

Parametr **average**=**'weighted'** oznacza, że miara F1, czułość i precyzja obliczane są jako **średnia ważona** dla wielu klas.

Każda z tych metryk jest użyteczna do oceny wydajności modelu klasyfikacji wieloklasowej, a ich wartości są określane w celu oceny skuteczności modelu.

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report

y\_pred = []

y\_test = []

with torch.no\_grad():

for images, labels in test\_loader:

images, labels = to\_device(images, labels, device=device)

outputs = model(images)

predicted = torch.argmax(outputs, 1).cpu().numpy()

labels = torch.argmax(labels, 1).cpu().numpy()

y\_pred.extend(predicted)

y\_test.extend(labels)

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred, average='weighted')

recall = recall\_score(y\_test, y\_pred, average='weighted')

f1 = f1\_score(y\_test, y\_pred, average='weighted')

Omówienie algorytmu testowania:

1. wyłączono obliczenia gradientów przez Pytorch
2. wczytano dane w batch’ach do GPU
3. przepuszczono batch’a obrazków przez model. Na wyjściu otrzymano 95-elementowe wektory dla każdej próbki danych
4. pozyskano ID najbardziej prawdopodobnych klas z predykcji i ID klas z prawdziwych danych
5. zapisano ID z predykcji i z prawdziwych danych
6. na podstawie danych z kroku 6 obliczono odpowiednie metryki z wykorzystaniem metod: accuracy\_score; precision\_score; recall\_score; f1\_score; przy pomocy biblioteki sklearn.metrics

Obliczone metryki porównano dla procesu z wykorzystaniem callbacków oraz bez ich wykorzystania. Z wykorzystaniem callbacków uzyskano wyniki:

Accuracy: 0.7931

Precision: 0.8304

Recall: 0.7931

F1 Score: 0.7918

Bez wykorzystania callbacków uzyskano wyniki:

Accuracy: 0.7763

Precision: 0.8207

Recall: 0.7763

F1 Score: 0.7717

Porównanie metryk treningu z wykorzystaniem callbacku i bez wskazuje na lepsze osiągnięcia modelu, gdy wykorzystywane są callbacki.

1. **Podsumowanie**

Celem niniejszej pracy jest opracowanie procesu trenowania konwolucyjnej sieci neuronowej (CNN) do klasyfikacji egipskich hieroglifów. Proces ten obejmuje kroki, poczynając od przygotowania zbioru danych zawierającego hieroglify, poprzez zdefiniowanie odpowiedniej architektury sieci neuronowej, przeprowadzenie treningu modelu, aż po zastosowanie i ocenę skuteczności różnych technik usprawniających proces trenowania.

W pierwszym etapie pracy, przeprowadzono staranne przygotowanie zbioru danych zawierającego hieroglify. Proces ten obejmował gromadzenie, czyszczenie i przetwarzanie danych w celu uzyskania spójnego i odpowiednio przygotowanego zestawu treningowego oraz testowego.

Następnie, zdefiniowano architekturę konwolucyjnej sieci neuronowej, uwzględniającą unikalne cechy strukturalne i wzorce hieroglifów. Architektura ta została starannie zaprojektowana w celu maksymalizacji skuteczności klasyfikacji hieroglifów, przy jednoczesnym minimalizowaniu ryzyka przetrenowania.

Po zdefiniowaniu architektury, przystąpiono do procesu treningu modelu. Wykorzystano odpowiednio dobrane zbiory: treningowy (60%); walidacyjny (20%); testowy (20%).

W ostatnim etapie pracy, przeprowadzono zastosowanie i ocenę wpływu różnych technik usprawniających trening modelu. Skoncentrowano technikach zastosowania callbacków, analizując ich wpływ z wykorzystaniem metryk: dokładność; czułość; precyzja; F1; na wydajność modelu pod względem dokładności klasyfikacji, szybkości uczenia się oraz odporności na przetrenowanie.

W rezultacie przeprowadzonych badań, udało się opracować efektywny proces trenowania konwolucyjnej sieci neuronowej do klasyfikacji egipskich hieroglifów.

Bibliografia

- Mark Collier, Bill Manley „How to read Egyptian hieroglyphs”, 1999

- zbiór danych z listy hieroglifów Gardinera ze strony kaggle com

[Egyptian Hieroglyphics Datasets (kaggle.com)](https://www.kaggle.com/datasets/waleedumer/egyptian-hieroglyphics-datasets) wyselekcjonowane przez: Sallar Khan ( asystent na [Sir Syed University of Engineering and Technology SSUET,](https://www.bing.com/ck/a?!&&p=c5dac84ab729065cJmltdHM9MTcxNzIwMDAwMCZpZ3VpZD0xZWU0YWY0ZC04OWNmLTY0ZDEtMWYzNi1iYjA1ODhiYzY1ZmUmaW5zaWQ9NTc3Ng&ptn=3&ver=2&hsh=3&fclid=1ee4af4d-89cf-64d1-1f36-bb0588bc65fe&psq=SSUET&u=a1aHR0cHM6Ly9wbC53aWtpcGVkaWEub3JnL3dpa2kvU2lyX1N5ZWRfVW5pdmVyc2l0eV9vZl9FbmdpbmVlcmluZ19hbmRfVGVjaG5vbG9neQ&ntb=1) Karachi, Pakistan) oraz Waleed Bin Umer (Data Analyst)

Artykuły na wikipedii

<https://pl.wikipedia.org/wiki/Lista_egipskich_hieroglifów>

<https://pl.wikipedia.org/wiki/Lista_znaków_Gardinera>

dokumentacje do bibliotek:

<https://scikit-learn.org/0.21/documentation.html>

<https://scikit-image.org/docs/stable/>

<https://pytorch.org/docs/stable/index.html>