

“NumPy Layer Lab” - Mateusz Walo

1. Opis projektu

Celem projektu było stworzenie lekkiego i edukacyjnego środowiska (laba) do tworzenia oraz trenowania prostych sieci neuronowych, bazującego wyłącznie głównie na bibliotece NumPy. Program umożliwia użytkownikowi znającemu podstawy algebry liniowej (operacji macierzowych – jest to warunek konieczny) poprzez interfejs wiersza poleceń zbudowanie własnej architektury sieci z podstawowych warstw (gęstych oraz aktywacyjnych), przeprowadzenie treningu na wbudowanym zbiorze danych (Digits 8×8), a także zapis wytrenowanego modelu i logów treningu do odpowiednich plików.

Głównym założeniem było zapewnienie maksymalnej przejrzystości i zrozumiałości działania każdej warstwy oraz procesu uczenia, co czyni projekt szczególnie przydatnym w celach dydaktycznych przy różnego rodzaju BootCampach. Narzędzie może być wykorzystywane przez osoby rozpoczynające naukę o uczeniu maszynowym, ponieważ pozwala zobaczyć „od kuchni”, jak działają kluczowe elementy takie jak propagacja w przód, wsteczna propagacja błędu czy funkcja kosztu.

Projekt rozwiązuje problem wysokiego poziomu abstrakcji w gotowych frameworkach ML (np. TensorFlow, PyTorch), które ukrywają istotne detale algorytmiczne. W NumPyLayer Lab wszystko od operacji macierzowych po aktualizację wag jest zaimplementowane ręcznie, z pełną kontrolą nad każdym etapem.

Do najważniejszych funkcjonalności projektu należą:

- możliwość definiowania architektury sieci przez dodawanie warstw (Dense, ReLU, LeakyReLU, Softmax),
- trenowanie modelu metodą SGD z funkcją kosztu Softmax + Cross-Entropy,
- zapis architektury do pliku .json oraz wag do .npz (oraz odczyt architektury),
- eksport przebiegu uczenia (loss/accuracy) do pliku .csv,
- obsługa CRUD na poziomie warstw (dodawanie, usuwanie, przedstawianie),
- pełna zgodność z zasadami programowania obiektowego.

2. Ogólna struktura projektu

Projekt został zorganizowany zgodnie z dobrymi praktykami inżynierii oprogramowania. Posiada wyraźnie wydzielone katalogi logiczne, a kod źródłowy znajduje się w katalogu *src*, co umożliwia łatwe utrzymanie, testowanie i rozwój labu. Taka struktura jest zgodna ze standardami stosowanymi w komercyjnych projektach programistycznych, również w środowiskach korporacyjnych m.in. w mojej pracy zawodowej w firmie Asseco, gdzie nacisk na klarowny podział warstw logicznych jest normą.

Całość projektu znajduje się w repozytorium GitHub ([LINK](#)), co umożliwia efektywne zarządzanie wersjami i historią zmian za pomocą systemu kontroli wersji Git. Repozytorium spełnia wymagania związane z przejrzystością kodu, automatycznym backupem, możliwością współpracy i późniejszego rozszerzania projektu, co najprawdopodobniej będzie miało miejsce w ramach Koła Uczenia Maszynowego ATLAS.

```

ROOT/
|
|- docs/           ← sprawozdanie PDF
|
|- logs/          ← wygenerowane historie treningu (.csv)
|
|- models/         ← zapisane modele (.json + .npz)
|
|- src/
    |- __init__.py
    |- cli.py
    |- model.py
    |- metrics.py
    |- utils.py
    |- layers/
        |- __init__.py
        |- base.py
        |- dense.py
        |- activation.py
            ← definicje warstw
            ← abstrakcja Layer
            ← Dense
            ← ReLU, LeakyReLU, Softmax
|
|- requirements.txt
|- README.md

```

3. Szczegółowa struktura klas i metod

Plik src/__init__.py : pełni funkcję inicjalizującą cały moduł aplikacji. Pozwala na wygodny import klas wysokiego poziomu bez konieczności eksplorowania struktury wewnętrznej katalogów. Dzięki temu użytkownik lub inny program może zimportować np. klasę Model. Plik ten nie zawiera funkcji ani klas jedynie przekierowuje importy, dzięki temu lab staje się pełnoprawnym pakietem Pythonowym.

Plik src/cli.py: Zawiera kompletny interfejs wiersza poleceń CLI umożliwiający użytkownikowi interaktywną pracę z projektem NumPyLayer Lab. Dzięki niemu można dynamicznie budować sieć neuronową, trenować ją, zapisywać, eksportować wyniki i zarządzać architekturą w wygodny sposób.

Zawartość pliku:

- Klasa CLI -Reprezentuje interaktywny interpreter poleceń.

Atrybuty:

- *model: Model | None* – aktualna instancja modelu.
- *history: list[dict[str, float]]* – lista słowników opisujących historię treningu (epoch, loss, accuracy).

Metody:

- *init(self)* - Inicjalizuje pusty model i historię.
- *_load_data(self)* - Wczytuje zbiór Digits 8×8 ze sklearn.datasets, normalizuje dane, dzieli na zbiór treningowy i testowy.
- *cmd_add(self, args: list[str])* - Dodaje warstwę do modelu.

- `cmd_list(self, _)` - Wyświetla wszystkie aktualnie dodane warstwy wraz z ich indeksem.
- `cmd_del(self, args: list[str])` - Usuwa warstwę o wskazanym indeksie z listy `self.model.layers`.
- `cmd_move(self, args: list[str])` - Przesuwa warstwę z indeksu `from` na pozycję `to`.
- `cmd_train(self, args: list[str])` - Trenuje aktualny model. Obsługuje argumenty pozycyjne: epochs (domyślnie: 5), lr -learning rate (domyślnie: 0.1), batch - batch size (domyślnie: 64). Dodatkowo ta metoda weryfikuje dopasowanie liczby cech wejściowych, automatycznie aktualizuje learning rate w warstwach gęstych i oczywiście po każdej epoce zapisuje loss, acc_train, acc_test do `self.history`.
- `cmd_save(self, args: list[str])` - Zapisuje aktualny model do katalogu `models/` jako `.json` i `.npz`.
- `cmd_load(self, args: list[str])` - Wczytuje model z katalogu `models/` na podstawie prefiku i odtwarza architekturę wraz z wagami.
- `cmd_export(self, args: list[str])` - Eksportuje historię treningu do pliku `.csv` w katalogu `logs/`.
- `cmd_help(self, _)` - Wyświetla dokumentację tekstową programu.
- `repl(self)` - Główna pętla programu uruchamia REPL czyta wejście, mapuje komendy na metody `cmd_*`, obsługuje quit, exit, EOF oraz KeyboardInterrupt.

Plik src/metrics.py: Zawiera pomocnicze funkcje ewaluacyjne oraz narzędzia do eksportowania wyników treningu. Jest modułem wspierającym, nie zawiera klas, lecz pełni kluczową rolę w ocenie jakości modelu oraz zapisie metryk do analizy po treningu.

Metody:

- `accuracy(preds: np.ndarray, labels: np.ndarray)` - Oblicza dokładność klasyfikatora tzn. odsetek poprawnych predykcji.
- `save_history_csv(history: List[dict[str, float]], path: Path)` - Zapisuje logi z treningu do `.csv`

Plik src/model.py: Zawiera klasę `Model`, która reprezentuje kompletną sieć neuronową jako sekwencję warstw (które są ustawione na stosie tak jak uczyła nas Pani w poprzednim semestrze 😊) Odpowiada za propagacje w przód, trening, dodawanie warstw oraz zapis i ładowanie modelu.

Zawartość pliku:

Atrybuty:

- `layers: list[Layer]` - sekwencyjna lista warstw tworzących architekturę.

Metody:

- `add_layer(self, layer: Layer) -> None` - Dodaje warstwę do modelu.
- `predict(self, x: np.ndarray) -> np.ndarray` - Przeprowadza pełną propagację w przód (forward pass).

- `fit(self, X: np.ndarray, y: np.ndarray, *, epochs: int = 5, lr: float = 0.1, batch_size: int = 64) -> list[dict]` - Trenuje model przez kilka epok z użyciem SGD. Zwraca historię strat. Dla każdej epoki robi shuffle danych i następuje podział na mini-batche. Dla każdego batcha następuje forward przez warstwy i Softmax (albo ręcznie albo przez ostatnio dodaną warstwę) następnie poropagacja gradientu wstecz i aktualizacja wag w warstwach Dense (o ile $lr > 0$), na koniec zapisuje epoch i loss do listy.
- `save(self, json_path: Path, npz_path: Path) -> None` – Zapisuje model do plików .json i .npz zapis odbywa się do katalogu models/.
- `@classmethod load(cls, json_path: Path, npz_path: Path) -> Model` – Wczytuje model z plików .json i .npz, kolejno odtwarza listę zapisanych warstw i odtwarza wagi

Plik src/utils.py: Zawiera zestaw pomocniczych funkcji wykorzystywanych przez główne komponenty projektu NumPyLayer Lab. Jego celem jest ułatwienie powtarzalnych operacji, takich jak losowość, kodowanie etykiet oraz tworzenie batchy danych. Plik nie zawiera klas, a jedynie funkcje narzędziowe.

Zawartość pliku:

Metody:

- `set_seed(seed: int = 42) -> None` – ustawia ziarno losowości do determinizmu procesu trenowania
- `one_hot(y: np.ndarray, num_classes: int | None = None) -> np.ndarray` - Zamienia etykiety klas na wektor OH, umożliwia to prawidłowy preprocessing danych do procesu uczenia maszynowego
- `batch_generator(X: np.ndarray, y: np.ndarray, batch_size: int = 64)` - Generator służący do dzielenia danych na mini-batche

Plik src/layers/__init__.py: Służy do agregacji i udostępnienia klas warstw z podmodułów dense.py oraz activation.py. Dzięki temu możliwe jest importowanie klas warstwowych z jednego miejsca, bez konieczności ręcznego sięgania do każdego podpliku.

Plik src/layers/activation.py: Zawiera podstawowe funkcje aktywacji używane w sieciach neuronowych. Każda klasa dziedziczy po abstrakcyjnej klasie Layer i implementuje metody forward i backward. Warstwy te nie mają parametrów uczących, więc params oraz grads zwracają puste słowniki. Warstwy te są wykorzystywane do wprowadzania nieliniowości w modelach oraz do końcowej klasyfikacji.

Zawartość pliku:

Atrybuty:

- `mask: np.ndarray | None` – przechowuje maskę wartości dodatnich używaną w ReLU i LeakyReLU.
- `alpha: float` – parametr w LeakyReLU, określający nachylenie dla wartości ujemnych.

- `out: np.ndarray | None` – wynik działania Softmax, wykorzystywany podczas propagacji wstecznej.

Metody:

- `forward(x: np.ndarray) -> np.ndarray` (z klasy ReLu) – ustawia maskę i zwraca wartości większe od zera.
- `backward(grad: np.ndarray) -> np.ndarray` (z klasy ReLu) – mnoży gradient przez maskę.
- `params, grads` – zwracają pusty słownik, działają tak samo w wszystkich klasach(brak parametrów uczących).
- `__str__()` (z klasy ReLu) – zwraca "ReLU()".
- `__init__(alpha=0.01)` (z klasy LeakyReLU) – domyślnie alpha = 0.01 jako mnożnik wartości mniejszych od zera.
- `forward(x: np.ndarray) -> np.ndarray` (z klasy LeakyReLU) – mnoży wartości ujemne przez alpha.
- `backward(grad: np.ndarray) -> np.ndarray` (z klasy LeakyReLU) – przekazuje gradient różnicując dla dodatnich i ujemnych.
- `__str__()` – zwraca "LeakyReLU(alpha=0.01)".
- `forward(x: np.ndarray) -> np.ndarray` (z klasy Softmax) – oblicza wartości softmax z zachowaniem stabilności numerycznej.
- `backward(grad: np.ndarray) -> np.ndarray` (z klasy Softmax) – oblicza gradient z wykorzystaniem macierzy Jacobiego (dla każdego przykładu osobno).
- `__str__()` (z klasy Softmax) – zwraca "Softmax()".

Plik src/layers/base.py: Zawiera definicję abstrakcyjnej klasy Layer, która stanowi wspólną bazę dla wszystkich warstw w sieci neuronowej. Klasa ta definiuje wymagane metody forward i backward, które muszą być zaimplementowane przez wszystkie warstwy dziedziczące. Ujednolica interfejs dla wszystkich typów warstw zarówno tych z parametrami uczącymi, jak i bez.

Zawartość pliku:

Atrybuty:6

- `params: Dict[str, np.ndarray]` – słownik trenowalnych parametrów warstwy (np. wagi, biasy). W klasach bez parametrów pozostaje pusty.
- `grads: Dict[str, np.ndarray]` – gradiensty odpowiadające parametrom, obliczane podczas propagacji wstecznej.

Metody:

- `forward(x: np.ndarray) -> np.ndarray` - Propagacja w przód — przekształca dane wejściowe do wyjściowe. Musi być zaimplementowana w każdej podklasie.
- `backward(grad: np.ndarray, lr: float) -> np.ndarray` - Propagacja wsteczna. Oblicza gradient względem wejścia oraz — jeśli warstwa posiada parametry — aktualizuje je na podstawie tempa uczenia lr.
- `zero_grad() -> None` - Zeruje wszystkie gradiensty. Przydatne przed kolejną iteracją treningową, zapobiega kumulacji gradientów.

- `__str__()` - Czytelna reprezentacja warstwy zawierająca liczbę parametrów i ich kształty.
- `__len__()` - Zwraca liczbę trenowalnych parametrów warstwy, umożliwiając np. użycie `len(layer)`.
- `__eq__(other: object) -> bool` - Porównuje dwie warstwy pod kątem typu i kształtów ich parametrów (np. przy testach lub serializacji).

Plik src/layers/dense.py: Zawiera implementację warstwy gęstej (Dense) – w pełni połączonej warstwy sieci neuronowej. Jest to podstawowy komponent w wielu architekturach sieciowych. Warstwa implementuje pełny przepływ danych (forward) oraz uczenie z użyciem algorytmu Stochastic Gradient Descent (SGD) w metodzie backward.

Zawartość pliku:

Atrybuty:

- `in_features: int` – liczba cech wejściowych.
- `out_features: int` – liczba cech wyjściowych.
- `lr: float | None` – tempo uczenia (opcjonalne, może być nadpisane przez `Model.fit`).
- `W: np.ndarray` – macierz wag, inicjalizowana losowo metodą He.
- `b: np.ndarray` – wektor biasów o kształcie `(1, out_features)`.
- `dW: np.ndarray | None` – gradient wag.
- `db: np.ndarray | None` – gradient biasów.
- `_x: np.ndarray | None` – cache z wejściem z metody forward, potrzebny w backward.

Metody:

- `forward(x: np.ndarray) -> np.ndarray`
Zwraca wynik $x @ W + b$. Przechowuje wejście `x` do użycia w metodzie backward.
- `backward(grad: np.ndarray) -> np.ndarray`
Oblicza gradienty względem wag (`dW`) i biasów (`db`) oraz — jeśli `lr` jest ustawione — wykonuje krok SGD aktualizując `W` i `b`. Zwraca gradient względem wejścia (`grad @ W.T`).
- `__str__()`
Czytelna reprezentacja warstwy, np. "Dense(128→64)".

4. Ogólny schemat działania programu

Program **NumPyLayer Lab** został wyposażony w interaktywny interfejs wiersza poleceń (CLI), który umożliwia użytkownikowi krok po kroku budowanie i trenowanie własnej sieci neuronowej. Obsługa odbywa się w pełni tekstowo – użytkownik nie musi pisać kodu źródłowego, a jedynie posługiwać się prostym językiem komend. Program uruchamia się za pomocą polecenia: **python -m src.cli**

Po starcie pojawia się kolorowy nagłówek oraz znak zachęty CLI. Użytkownik rozpoczyna pracę od budowy architektury sieci, korzystając z poleceń **add**, które dodają kolejne warstwy (np.

dense, relu, softmax) do aktualnego modelu w pamięci. Każde dodanie jest potwierdzane komunikatem. W każdej chwili użytkownik może podejrzeć zbudowaną architekturę, wpisując komendę **list**.

Po skonstruowaniu sieci użytkownik przystępuje do treningu, używając komendy **train**, która wykorzystuje wbudowany zbiór danych Digits 8×8 (pochodzący z `sklearn.datasets.load_digits()`). Dane te zawierają cyfry zapisane jako obrazy o wymiarach 8×8 pikseli.

W czasie treningu program automatycznie dopasowuje wartość `in_features` do liczby cech wejściowych. Następnie wykonuje standardowe operacje w sieci neuronowej: propagację w przód, obliczenie funkcji straty (Softmax + CrossEntropy), propagację wsteczną błędu oraz aktualizację wag metodą SGD.

Po zakończeniu treningu użytkownik może:

- zapisać metryki (loss i accuracy) do pliku `.csv` za pomocą polecenia **export**,
- zapisać cały model (architekturę i wagi) do plików `.json` i `.npz` w katalogu `models/` za pomocą **save**.

W przypadku potrzeby edycji architektury można:

- usunąć konkretną warstwę komendą **del <index>**,
- przemieścić warstwę komendą **move <from> <to>**.

Sesję można zakończyć wpisując `quit`, lub alternatywnie naciskając skrót `Ctrl+C` (np. w systemie Linux).

5. Przykład użytkowania projektu

Poniżej przedstawiono pełny przykład działania programu **NumPyLayer Lab**, wraz z rzeczywistymi zrzutami ekranu prezentującymi interakcję użytkownika z systemem oraz wygenerowanymi plikami.

- Przykład uruchomienia projektu za pomocą polecenia **python -m src.cli** (w konsoli bashowej na Linuxie, wygląd będzie się różnił w zależności od systemu operacyjnego)

```
@ mateusz-walo .../projekt_numpylayerlab 🌱 main !?↑ 21:17:30 python -m src.cli
== NumPyLayer Lab CLI ==
Interfejs wiersza poleceń (CLI) dla **NumPyLayer Lab**.

Uruchomienie:
$ python -m src.cli

Polecenia (wpisz `help`, aby zobaczyć listę):
add <warstwa> dodaje warstwę (Dense, ReLU, ...)
list pokazuje aktualną architekturę
del <index> usuwa warstwę o podanym indeksie
move <from> <to> przestawia warstwę
train trenuje sieć na wbudowanym zbiorze Digits 8x8
save <prefix> zapisuje <prefix>.json + <prefix>.npz
load <prefix> wczytuje model z <prefix>.json/.npz
export <csv_path> zapisuje historię loss/accuracy do CSV
quit wyjście z programu

numl>
```

- Użytkownik krok po kroku buduje architekturę sieci neuronowej, dodając warstwy przez CLI. Każde dodanie jest potwierdzane.

```
numl> add Dense 64 32
Dodano warstwę: Dense(64→32)
numl> add ReLU
Dodano warstwę: ReLU()
numl> add Dense 32 10
Dodano warstwę: Dense(32→10)
numl> add softmax
Dodano warstwę: Softmax()
```

- Użytkownik może sprawdzić bieżący układ warstw i ich parametry.

```
numl> list
0: Dense(64→32)
1: ReLU()
2: Dense(32→10)
3: Softmax()
```

- Program trenuje sieć na zbiorze Digits. Po każdej epoce wypisywane są metryki.

```
numl> train 5 0.05 64
Epoka 1/5 loss=2.7725 acc_train=0.102 acc_test=0.104
Epoka 2/5 loss=2.7441 acc_train=0.102 acc_test=0.104
Epoka 3/5 loss=2.7215 acc_train=0.102 acc_test=0.104
Epoka 4/5 loss=2.7064 acc_train=0.102 acc_test=0.104
Epoka 5/5 loss=2.7011 acc_train=0.102 acc_test=0.104
numl>
```

- Program trenuje sieć na zbiorze Digits. Po każdej epoce wypisywane są metryki.

```
numl> export run1.csv
Zapisano historię do run1.csv
numl>
```

- Historia uczenia zostaje zapisana w postaci pliku .csv, który można analizować zewnętrznie przechodząc do folderu logs i znajdując konkretny plik .csv.

run1.csv	
logs >	run1.csv
1	epoch,loss
2	1,2.772487445137466
3	2,2.7440677062418857
4	3,2.7215129442242167
5	4,2.7063583526639934
6	5,2.7011016542359
7	

- Architektura modelu i jego wagi zostają zapisane do katalogu models/

```
numl> save model_demo_1
Zapisano model pod prefiksem model_demo_1
numl>
```

```
1 [ { 2   "class": "Dense", 3     "config": { 4       "in_features": 64, 5       "out_features": 32 6     } 7   }, 8   { 9     "class": "ReLU", 10    "config": {} 11   }, 12   { 13     "class": "Dense", 14     "config": { 15       "in_features": 32, 16       "out_features": 10 17     } 18   }, 19   { 20     "class": "Softmax", 21     "config": {} 22   } 23 } 24 ]
```

- Zapisany model możemy załadować bez konieczności budowania go od zera

```
numl> load model_demo_1
Wczytano model z prefiku model_demo_1
numl> list
0: Dense(64-32)
1: ReLU()
2: Dense(32-10)
3: Softmax()
```

- Możliwa jest edycja architektury w dowolnym momencie – usuwanie i reorganizacja warstw.

```
numl> move 1 2
Przeniesiono warstwę na pozycję 2
numl> list
0: Dense(64→32)
1: Dense(32→10)
2: ReLU()
3: Softmax()
numl> █
```

```
numl> list
0: Dense(64→32)
1: Dense(32→10)
2: ReLU()
3: Softmax()
numl> del 2
Usunięto: ReLU()
numl> list
0: Dense(64→32)
1: Dense(32→10)
2: Softmax()
numl> █
```

- Opuszczenie programu po zakończeniu pracy.

```
numl> quit
Do zobaczenia!
```

- Obsługa pomocy, kiedy użytkownik zapomniał jak działa konkretna funkcja

```
numl> help
Interfejs wiersza poleceń (CLI) dla **NumPyLayer Lab**.

Uruchomienie:
$ python -m src.cli

Polecenia (wpisz `help`, aby zobaczyć listę):
add <warstwa>      dodaje warstwę (Dense, ReLU, ...)
list                  pokazuje aktualną architekturę
del  <index>         usuwa warstwę o podanym indeksie
move <from> <to>     przestawia warstwę
train                trenuje sieć na wbudowanym zbiorze Digits 8x8
save <prefix>        zapisuje <prefix>.json + <prefix>.npz
load <prefix>        wczytuje model z <prefix>.json/.npz
export <csv_path>    zapisuje historię loss/accuracy do CSV
quit                 wyjście z programu

numl>
```

6. Wnioski i podsumowanie

Realizacja projektu **NumPyLayer Lab** pozwoliła na stworzenie w pełni funkcjonalnego, edukacyjnego framework'a do tworzenia i trenowania prostych sieci neuronowych w czystym NumPy. Projekt ten od początku miał za zadanie nie tylko spełnić wymogi formalne związane z programowaniem obiektowym, ale również zapewnić praktyczne, zrozumiałe i możliwe do samodzielnie modyfikowania środowisko dla osób rozpoczynających naukę w dziedzinie uczenia maszynowego.

- Zrealizowano kompletny cykl życia modelu ML: od konstrukcji architektury, przez trening, aż po eksport i odczyt.
- Stworzono wygodny i przejrzysty interfejs CLI, który pozwala na korzystanie z systemu bez pisania własnego kodu.
- W pełni spełniono wymagania OOP: implementacja wielu klas, dziedziczenie, polimorfizm, metody specjalne i obsługa wyjątków.
- Warstwy i mechanizmy uczenia zostały napisane ręcznie – bez użycia frameworków takich jak TensorFlow czy PyTorch – co zwiększa transparentność algorytmów i edukacyjność rozwiązania.
- Projekt został przygotowany w sposób modularny, z myślą o dalszym rozwoju i testowalności.d
- Projekt pokazał, że z pomocą NumPy i odpowiedniej struktury klas możliwe jest zaimplementowanie kluczowych mechanizmów działania sieci neuronowej w sposób zrozumiały i prosty do rozwijania.
- Warstwowy design z użyciem Layer jako klasy bazowej sprawdził się bardzo dobrze w kontekście rozszerzalności.
- Interfejs CLI okazał się wygodnym narzędziem zarówno do szybkiego testowania, jak i do edukacji – użytkownik może bezproblemowo obserwować wpływ zmian architektury na jakość predykcji.

