Sztuczne Sieci Neuronowe

Daria Bartkowiak, Karolina Kawulska, Oliwia Wójcicka

Styczeń 2026

1 Implementacja perceptronu wielowarstwowego

1.1 Reprezentacja sieci i inicjalizacja parametrów

Architektura sieci jest parametryzowana listą

$$\texttt{layer\_sizes} = [n_{\text{in}}, n_1, \ldots, n_{\text{out}}],$$

co pozwala ustawić dowolną liczbę warstw i neuronów w każdej warstwie. Dla każdej pary sąsiednich warstw tworzona jest macierz wag

$$W^{(l)} \in \mathbb{R}^{n_l \times (n_{l-1}+1)}.$$

Dodatkowy wymiar (+1) odpowiada wyrazowi wolnemu (bias), zgodnie z modelem neuronu, gdzie do sumy ważonej dodawany jest stały składnik wolny.

Wagi inicjalizowane są losowo z rozkładu normalnego i skalowane czynnikiem zależnym od liczby wejść, aby ograniczyć zbyt duże wartości pre-aktywacji na starcie uczenia.

1.2 Propagacja w przód (`forward`)

Dla wejścia X (próbki w kolumnach) sieć oblicza kolejne warstwy zgodnie z modelem neuronu z wykładu. Najpierw sumę ważoną (pre-aktywację), a następnie aktywację:

$$Z^{(l)} = W^{(l)} \begin{bmatrix} A^{(l-1)} \\ \mathbf{1} \end{bmatrix}, \qquad A^{(l)} = \psi\Big(Z^{(l)}\Big),$$

gdzie $\mathbf{1}$ to wiersz jedynek realizujący bias, a ψ to wybrana funkcja aktywacji (np. sigmoidalna lub ReLU). Zaimplemenotwano dwie podane wyżej funkcje aktywacyjne, aczkolwiek implementacja sieci pozwala na wykorzystanie dowolnej po wcześniejszym dodaniu jej do słownika.

W implementacji zapisywane są listy $A^{(l)}$ (aktywacje) oraz $Z^{(l)}$ (pre-aktywacje), ponieważ są potrzebne w propagacji wstecznej.

1.3 Funkcja straty

Zaimplementowano jedną stratę MSE:

$$\mathcal{L} = \frac{1}{2m} \sum_{i=1}^{m} \left\| A_i^{(L)} - Y_i \right\|^2, \qquad \frac{\partial \mathcal{L}}{\partial A^{(L)}} = \frac{A^{(L)} - Y}{m},$$

natomiast implementacja sieci umożliwia wykorzystanie dowolnej funkcji straty, po wcześniejszym dodaniu jej do słownika.

1.4 Propagacja wstecz (`backward`)

Gradienty wag wyznaczane są metodą wstecznej propagacji gradientu (reguła łańcuchowa), zgodnie z podejściem omawianym na wykładzie.
Dla warstwy l (licząc od końca) obliczane są:

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial Z^{(l)}} = \frac{\partial \mathcal{L}}{\partial A^{(l)}} \circ \psi'(Z^{(l)}),$$

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} \left(\begin{bmatrix} A^{(l-1)} \\ \mathbf{1} \end{bmatrix} \right)^T, \qquad \frac{\partial \mathcal{L}}{\partial A^{(l-1)}} = \left(W^{(l)}_{\text{no-bias}} \right)^T \delta^{(l)}.$$

W praktyce oznacza to, że błąd jest propagowany od warstwy wyjściowej do wejściowej, a po drodze wyznaczane są gradienty wag dla każdej warstwy.

1.5 Uczenie: SGD, reshuffling i mini-batche

Parametry sieci są aktualizowane metodą stochastycznego najszybszego spadku (SGD):

$$W^{(l)} \leftarrow W^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial W^{(l)}},$$

gdzie α to learning rate. W implementacji obsłużono tryb:

- **full-batch**: aktualizacja na podstawie całego zbioru danych,
- **mini-batch**: podział danych na mini-pakiety oraz opcjonalne mieszanie próbek w każdej epoce (reshuffling), zgodnie z zaleceniami z wykładu o uczeniu SGD.

1.6 Uczenie: Adam

Parametry sieci mogą być aktualizowane także za pomocą algorytmu **Adam**, który łączy zalety metody momentów i adaptacyjnego uczenia się współczynników. Wagi są aktualizowane według wzorów:

$$\begin{aligned}
M_t^{(l)} &= \beta_1 M_{t-1}^{(l)} + (1-\beta_1) \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \\
V_t^{(l)} &= \beta_2 V_{t-1}^{(l)} + (1-\beta_2) \left(\frac{\partial \mathcal{L}}{\partial W^{(l)}} \right)^2, \\
\hat{M}_t^{(l)} &= \frac{M_t^{(l)}}{1-\beta_1^t}, \\
\hat{V}_t^{(l)} &= \frac{V_t^{(l)}}{1-\beta_2^t}, \\
W^{(l)} &\leftarrow W^{(l)} - \alpha \frac{\hat{M}_t^{(l)}}{\sqrt{\hat{V}_t^{(l)}} + \epsilon},
\end{aligned}$$

gdzie:

- $M_t^{(l)}$ – estymacja pierwszego momentu gradientu dla warstwy l,
- $V_t^{(l)}$ – estymacja drugiego momentu gradientu,
- $\hat{M}_t^{(l)}, \hat{V}_t^{(l)}$ – poprawione estymacje momentów,
- β_1, β_2 – współczynniki wygładzania, domyślnie $\beta_1 = 0.9$, $\beta_2 = 0.999$,
- ϵ – mała wartość numeryczna dla stabilności, domyślnie 10^{-8},
- α – learning rate

2 Klasyfikacja jakości wina z wykorzystaniem perceptronu wielowarstwowego

Celem tej części eksperymentu było wytrenowanie perceptronu wielowarstwowego do zadania klasyfikacji jakości wina na podstawie jego parametrów. Wykorzystano zbiór danych `data.csv`, zawierający 11 cech opisujących próbki wina. Zmienną decyzyjną była kolumna `quality`, przyjmująca wartości całkowite z zakresu 0–10, które potraktowano jako etykiety klas w zadaniu klasyfikacji wieloklasowej.

2.1 Podział danych i metodologia

Dane zostały podzielone na trzy rozłączne zbiory:

- zbiór treningowy (70%),
- zbiór walidacyjny (15%),
- zbiór testowy (15%).

Podział wykonano z zachowaniem proporcji klas (stratyfikacja). Aby ograniczyć wpływ losowości inicjalizacji wag oraz podziału danych, cały proces uczenia i walidacji powtórzono dla 10 różnych wartości ziarna generatora liczb losowych. Ostateczne wyniki przedstawiono jako średnią z tych powtórzeń.

Zbiór treningowy służył wyłącznie do uczenia modeli, zbiór walidacyjny do porównywania konfiguracji hiperparametrów, natomiast zbiór testowy wykorzystano jedynie do końcowej, niezależnej oceny jakości wybranego modelu.

2.2 Strojenie hiperparametrów

Przeprowadzono przeszukiwanie siatki hiperparametrów obejmującej:

- architekturę sieci (liczbę warstw ukrytych oraz neuronów w każdej warstwie),
- współczynnik uczenia (learning rate),
- rozmiar batcha,

- liczbę epok uczenia.

Początkowo rozważano szeroką siatkę hiperparametrów, obejmującą różne architektury sieci, szeroki zakres współczynnika uczenia, rozmiarów batcha oraz liczby epok. Takie podejście pozwalało na wstępne rozpoznanie wpływu poszczególnych parametrów na proces uczenia, jednak wiązało się z dużym kosztem obliczeniowym.

Na podstawie wyników wstępnych eksperymentów oraz obserwacji stabilności procesu uczenia zakres rozważanych hiperparametrów został zawężony do wartości zapewniających najbardziej efektywne i stabilne uczenie. W szczególności ograniczono zakres współczynnika uczenia, rozmiar batcha oraz liczbę epok do przedziałów, w których uzyskiwano najlepsze wyniki walidacyjne. Ostatecznie dalsze eksperymenty przeprowadzono z użyciem zredukowanej siatki hiperparametrów, co pozwoliło znacząco skrócić czas obliczeń bez utraty jakości uzyskanych rezultatów.

Najlepszą konfigurację wybrano na podstawie najwyższej średniej dokładności walidacyjnej:

- architektura: dwie warstwy ukryte o rozmiarach [32, 16],
- learning rate: 0.1,
- batch size: 32,
- liczba epok: 500.

Dla tej konfiguracji uzyskano średnią dokładność na zbiorze walidacyjnym równą około 0.5941, podczas gdy dokładność treningowa osiągnęła 0.6098, co świadczy o stabilnym procesie uczenia i ograniczonym stopniu przeuczenia.

2.3 Ocena na zbiorze testowym

Wybrany zestaw hiperparametrów wykorzystano następnie do wytrenowania modelu, który oceniono na zbiorze testowym. Średnia dokładność klasyfikacji na zbiorze testowym (uśredniona po 10 seedach) wyniosła około 0.5712, co potwierdza dobrą zdolność generalizacji wybranego modelu.

2.4 Analiza skrajnych konfiguracji

Dodatkowo przeanalizowano zachowanie sieci neuronowej dla skrajnych ustawień hiperparametrów w celu zilustrowania zjawisk niedouczenia i przeuczenia oraz zmiany pozostałych hiperparametrów.

Silne niedouczenie (underfitting). Zastosowano konfigurację o bardzo małej pojemności i krótkim czasie uczenia: architektura [8], learning rate = 0.001, uczenie full-batch oraz 50 epok. Dla tej konfiguracji uzyskano bardzo niską dokładność zarówno na zbiorze treningowym (0.1978), jak i testowym (0.1876). Wyniki te są zbliżone do losowej klasyfikacji i jednoznacznie wskazują na silne niedouczenie modelu, wynikające z niewystarczającej pojemności oraz zbyt krótkiego i wolnego procesu uczenia.

Duża architektura. W kolejnym eksperymencie zwiększono pojemność modelu do architektury [64], pozostawiając learning rate = 0.1, batch size = 32 oraz 500 epok. Uzyskano wzrost dokładności na zbiorze treningowym (0.6308), jednak średnia dokładność na zbiorze testowym wyniosła 0.5739 i nie uległa istotnej poprawie w porównaniu z konfiguracją optymalną. Oznacza to, że większa pojemność sieci zaczynała prowadzić do przeuczenia bez wyraźnych korzyści dla generalizacji.

Zbyt mała liczba epok. Dla architektury [32,16], learning rate = 0.1 i batch size = 32, przy liczbie epok równej 50 uzyskano dokładność 0.5367 na zbiorze treningowym oraz 0.5229 na zbiorze testowym. Otrzymane wyniki są wyraźnie niższe niż dla konfiguracji optymalnej, co wskazuje, że model nie osiągnął pełnej konwergencji i pozostał częściowo niedouczony.

Zbyt duża liczba epok. Zwiększenie liczby epok do 1500 przy tej samej architekturze [32,16], learning rate = 0.1 i batch size = 32 skutkowało wzrostem dokładności na zbiorze treningowym do poziomu 0.6624, przy jedynie nieznacznej poprawie dokładności testowej (0.5706). Wydłużenie czasu uczenia zwiększyło różnicę pomiędzy wynikami treningowymi i testowymi, co wskazuje na narastające przeuczenie oraz brak istotnych korzyści z dalszego trenowania modelu.

Bardzo mały rozmiar batcha. Dla bardzo małego batcha (8), przy architekturze [32,16], learning rate = 0.1 i 500 epokach, uzyskano wysoką dokładność na zbiorze treningowym (0.6886), natomiast dokładność testowa wyniosła 0.5634. Zwiększona losowość procesu uczenia sprzyjała dopasowaniu modelu do danych treningowych, prowadząc do pogorszenia zdolności generalizacji.

Uczenie full-batch. Zastosowanie uczenia full-batch przy architekturze [32,16], learning rate = 0.1 i 500 epokach doprowadziło do stosunkowo niskiej dokładności zarówno na zbiorze treningowym (0.4910), jak i testowym (0.4686). Brak losowości w aktualizacji wag powodował mniej efektywną konwergencję i ograniczał zdolność modelu do uczenia się uogólnionych zależności.

Bardzo mały współczynnik uczenia. Przy bardzo niskim współczynniku uczenia (0.001) model uczony z architekturą [32,16] i batch size = 32 przez 500 epok osiągnął dokładność 0.4350 na zbiorze treningowym oraz 0.4340 na zbiorze testowym. Tak mała wartość learning rate prowadziła do bardzo wolnego procesu uczenia, skutkując niedostatecznym dopasowaniem modelu.

Bardzo duży współczynnik uczenia. Zastosowanie bardzo dużego współczynnika uczenia (0.6) przy architekturze [32,16], batch size = 32 oraz 500 epokach pozwoliło uzyskać wysoką dokładność na zbiorze treningowym (0.7468), przy dokładności testowej równej 0.5706. Największa różnica pomiędzy wynikami treningowymi i testowymi wskazuje na najsilniejsze przeuczenie spośród analizowanych konfiguracji.

Silne przeuczenie – duża sieć i długie uczenie. Dla bardzo złożonej architektury [64,32], learning rate = 0.1, batch size = 32 oraz 1500 epok uzyskano wysoką dokładność treningową (0.6660), przy dokładności testowej równej 0.5719. Pomimo dużej pojemności i długiego czasu uczenia nie uzyskano poprawy jakości generalizacji, co jednoznacznie wskazuje na przeuczenie modelu.

2.5 Wnioski

Przeprowadzone eksperymenty pokazują, że perceptron wielowarstwowy charakteryzuje się wyraźną wrażliwością na dobór hiperparametrów. Niedouczenie występowało dla modeli o zbyt małej pojemności, krótkim czasie uczenia lub bardzo niskim współczynniku uczenia, natomiast przeuczenie pojawiało się wraz ze wzrostem złożoności modelu, liczby epok, zmniejszaniem rozmiaru batcha oraz przy bardzo dużych wartościach learning rate.

Najlepszy kompromis pomiędzy dokładnością treningową i testową uzyskano dla sieci o umiarkowanej pojemności, uczonej z wykorzystaniem mini-batch SGD (batch size = 32) oraz odpowiednio dobranego współczynnika uczenia. Wyniki te potwierdzają, że dalsze zwiększanie złożoności modelu lub czasu uczenia nie prowadzi do poprawy generalizacji, a jedynie zwiększa ryzyko przeuczenia oraz koszt obliczeniowy.

3 Porównanie algorytmu SGD i Adam

3.1 Jak działa Adam

Algorytm **Adam** jest optymalizatorem stosowanym w uczeniu sieci neuronowych, który łączy zalety metody momentów i adaptacyjnego dostosowywania współczynników uczenia dla poszczególnych parametrów. W odróżnieniu od klasycznego SGD, Adam korzysta zarówno z pierwszego momentu gradientu , jak i z drugiego momentu, co pozwala na stabilniejsze i szybsze zbieganie w kierunku minimum funkcji straty. Ponadto, zastosowanie korekty biasu redukuje błędy wynikające z początkowej inicjalizacji momentów, szczególnie w pierwszych krokach uczenia.

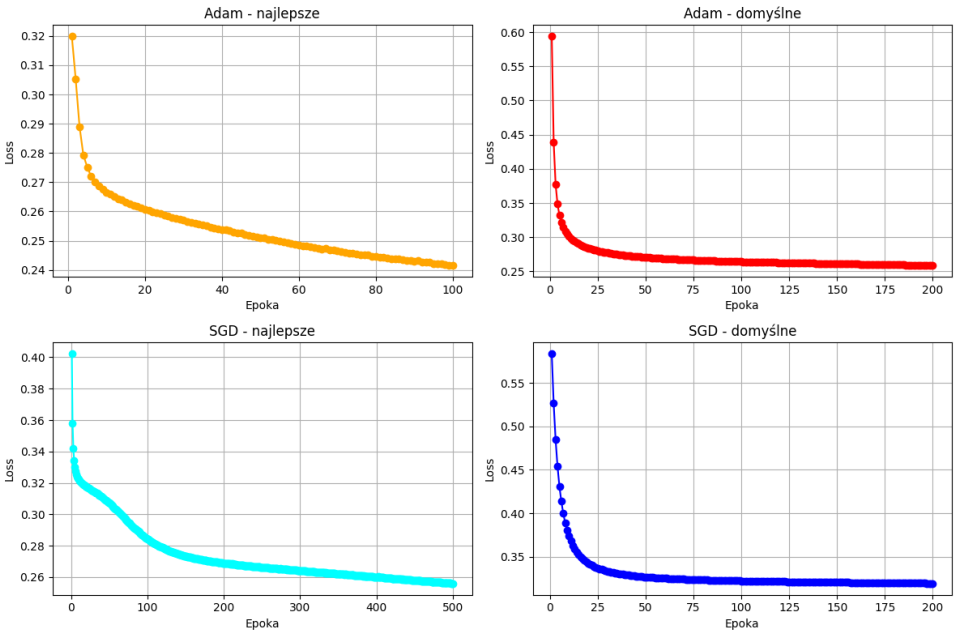
Algorytm automatycznie dostosowuje tempo uczenia dla każdego parametru, dzięki czemu jest mniej wrażliwy na dobór początkowej wartości współczynnika uczenia α w porównaniu do tradycyjnego SGD.

3.2 Wyniki

Analogicznie jak w przypadku SGD, przeprowadzono dopasowanie hiperparametrów dla obu algorytmów, a następnie wytrenowano cztery różne modele:

- **SGD default** – wartości domyślne,
- **SGD** – zoptymalizowane wartości hiperparametrów,
- **Adam default** – standardowe wartości: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$,
- **Adam** – zoptymalizowane wartości hiperparametrów.

Po wytrenowaniu modeli porównano ich wydajność na zbiorze testowym oraz analizowano zmiany funkcji straty w zależności od liczby epok.



Rysunek 1: Zmiana wartości funkcji straty w zależności od liczby epok dla różnych algorytmów optymalizacji.

Porównanie accuracy na zbiorze testowym:

- SGD domyślne: 0.4641 ± 0.0319
- SGD najlepsze: 0.5673 ± 0.0471
- Adam domyślne: 0.5804 ± 0.0369
- Adam najlepsze: 0.5784 ± 0.0288

3.3 Wnioski

Na podstawie przeprowadzonych eksperymentów można sformułować następujące wnioski:

- Algorytm **Adam** jest bardziej stabilny w trakcie uczenia i szybciej zbiega niż SGD, co widać po szybszym spadku funkcji straty na wykresie.
- Adam jest mniej wrażliwy na wartość learning rate w porównaniu do klasycznego SGD.
- Mimo że różnice w końcowej dokładności nie są duże, Adam osiąga lepsze wyniki średnie, co świadczy o większej powtarzalności wyników.
- **Funkcja straty jest lepiej minimalizowana przez Adama**, co oznacza, że w trakcie treningu model osiąga niższe wartości loss szybciej i bardziej stabilnie niż przy użyciu klasycznego SGD.