Sztuczna Inteligencja - Laboratorium

Robert Benke, Karol Draszawka, Szymon Olewniczak, Julian Szymański

czerwiec 2023

Laboratorium 1

Regresja

Regresja jest jednym z podstawowych problemów uczenia maszynowego, wpisującym się w szerszą kategorię uczenia nadzorowanego. W problemach regresji naszym zadaniem jest znalezienie funkcji $f(\vec{x})$, która na podstawie wejściowego wektora obserwacji $\vec{x}$, możliwie najdokładniej oszacuje wartość interesującej nas cechy wynikowej. Przykładem problemu regresji może być szacowanie ceny mieszkania na podstawie cech takich jak powierzchnia, dzielnica, odległość od centrum. W tym wypadku nasz wejściowy wektor obserwacji składa się z trzech cech, a cena mieszkania stanowi naszą cechę wynikową.

W regresji, podobnie jak w innych problemach z dziedziny uczenia nadzorowanego, nie jesteśmy w stanie określić dokładnego wzoru funkcji $f(\vec{x})$. Jedyne co posiadamy to pewien zbiór wektorów obserwacji X powiązanych z rzeczywistymi wartościami cech wynikowych Y. Na podstawie tej ograniczonej informacji, chcemy stworzyć model, który sprawdzałby się dobrze w ogólnym przypadku.

1.1 Regresja liniowa

Regresja liniowa jest jedną z najprostszych technik wyznaczania przybliżonego wzoru funkcji $f(\vec{x})$ na podstawie posiadanego zbioru obserwacji. W modelu tym zakładamy, że naszą cechę wynikową (zwaną zmienną zależną) można wyliczyć na podstawie liniowej kombinacji cech wejściowych (zwanych zmiennymi niezależnymi):

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \tag{1.1}$$

gdzie $\hat{y}$ jest naszą prognozowaną wartością cechy wyjściowej, n określa liczbę cech wejściowych modelu, x_i to wartość i-tej cechy, a θ_j to j-ty parametr modelu, gdzie θ_0 stanowi wyraz wolny, zwany punktem obciążenia (*bias term*). Równanie 1.1 możemy zapisać także w formie wektorowej:

$$\hat{y} = \vec{\theta} \circ \vec{x} \tag{1.2}$$

gdzie $\vec{x}$ stanowi rozszerzony wektor cech wejściowych $\vec{x} = [1, x_1, \cdots, x_n]$.

Zadaniem naszego algorytmu uczenia maszyn jest znalezienie wektora parametrów $\vec{\theta}$ dla którego błąd pomiędzy wartością oczekiwaną y, a oszacowaną $\hat{y}$ będzie możliwie najmniejszy. Proces ten nazywamy treningiem modelu.

1.2 Podział danych

Mając do dyspozycji zbiór obserwacji powiązanych z rzeczywistymi cechami wynikowymi, z reguły zanim przystąpimy do treningu modelu, dokonujemy jego podziału na dwa podzbiory. Pierwszy z nich nazywamy zbiorem treningowym, a drugi testowym. Podziału takiego najczęściej dokonuje się w sposób losowy, przydzielając większość danych do zbioru treningowego (popularną proporcją jest np. 80% dane treningowe, 20% dane testowe). Dane treningowe służą nam do treningu modelu, czyli na ich podstawie staramy się określić optymalne wartości wektora $\vec{\theta}$. Dane testowe wykorzystujemy natomiast w celu oszacowania rzeczywistej jakości modelu na danych, które nie były dla niego dostępne podczas treningu. Z reguły, jak na to wskazuje intuicja, wyniki modelu są lepsze dla danych treningowych, niż dla danych testowych.

1.3 Funkcja kosztu

Funkcja kosztu pozwala określić nam różnicę pomiędzy rzeczywistymi wartościami cech wyjściowych, a wartościami wyliczonymi przez model, czyli innymi słowy błąd naszego modelu. Dla problemu regresji często wykorzystywaną funkcją kosztu jest błąd średnio-kwadratowy (*mean square error, MSE*):

$$MSE(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\vec{\theta} \circ \vec{x}^{(i)} - y^{(i)})^2 \tag{1.3}$$

gdzie m - to liczba elementów w zbiorze obserwacji, $\vec{\theta} \cdot \vec{x}^{(i)}$ to predykcja naszego modelu dla i-tego wektora cech, a $y^{(i)}$ to rzeczywista wartość cechy wyjściowej dla i-tego wektora cech.

Funkcję kosztu wykorzystujemy podczas oceny jakości naszego modelu przy wykorzystaniu zbioru testowego, jak i również w niektórych algorytmach treningu.

1.4 Gradient funkcji

Gradient funkcji jest jednym z centralnych pojęć w dziedzinie uczenia maszynowego. Gradientem funkcji f nazywamy wektor pochodnych cząstkowych funkcji po wszystkich jej argumentach:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n}\right] \tag{1.4}$$

Zwrot wektora gradientu funkcji wskazuje nam kierunek najszybszego wzrostu wartości funkcji w danym punkcie.

W uczeniu maszynowym gradient funkcji wykorzystujemy najczęściej w celu wyznaczeniu wpływu poszczególnych wartości wektora parametrów $\vec{\theta}$ na wartość funkcji kosztu:

$$\nabla_\theta MSE(\theta) = \left[\frac{\partial}{\partial \theta_0} MSE(\theta), \cdots, \frac{\partial}{\partial \theta_n} MSE(\theta)\right] \tag{1.5}$$

W przypadku regresji liniowej $\frac{\partial}{\partial \theta_j} MSE(\theta)$ przyjmuje postać:

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\vec{\theta} \circ \vec{x}^{(i)} - y^{(i)}) x_j^{(i)} \tag{1.6}$$

gdzie $x_j^{(i)}$ to wartość j-tej cechy i-tego wektora wejściowego.
Równianie 1.5 możemy również zapisać w postaci macierzowej jako:

$$\nabla_\theta MSE(\theta) = \frac{2}{m} \dot{X}^T (\dot{X}\theta - y) \tag{1.7}$$

gdzie $\dot{X}$ to macierz obserwacji o wymiarach $(m \times n + 1)$:

$$\dot{X} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & \cdots & x_{mn} \end{bmatrix} \tag{1.8}$$

θ to wektor kolumnowy $(n + 1 \times 1)$ reprezentujący wagi naszego modelu:

$$\theta = \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_n \end{bmatrix} \tag{1.9}$$

a y to wektor kolumnowy $(m \times 1)$ reprezentujący wartości rzeczywiste odpowiadające poszczególnym wektorom obserwacji:

$$y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{1.10}$$

Gradient funkcji kosztu pozwala nam odpowiedzieć na bardzo ważne pytanie: jak powinniśmy zmienić wartości wektora parametrów θ, aby dla danego zbioru obserwacji $\dot{X}$ błąd pomiędzy wartościami rzeczywistymi y, a predykcją modelu $\hat{y}$ zmniejszył się.

1.5 Rozwiązanie jawne

Regresja liniowa jest jednym z nielicznych modeli uczenia maszynowego, który posiada tzw. rozwiązanie jawne (*closed-form solution*). Pozwala ono w sposób analityczny wyznaczyć optymalne wartości wektora parametrów θ, które minimalizują błąd pomiędzy wartościami wyliczonymi $\hat{y}$, a rzeczywistymi y.

Najczęściej stosowaną metodą wyznaczania rozwiązania jawnego dla regresji liniowej jest metoda najmniejszych kwadratów (*least squares*), której celem jest zminimalizowanie dystansu między poszczególnymi punktami danych, a linią regresji:

$$argmin_\theta (\dot{X}\theta - y)^T (\dot{X}\theta - y) \tag{1.11}$$

W celu znalezienia wektora θ rozwiązującego zadany problem optymalizacyjny wyliczamy gradient z minimalizowanej funkcji, a następnie przyrównujemy go do 0 w celu znalezienia minimum:

$$\dot{X}^T(\dot{X}\theta - y) = 0 \tag{1.12}$$

Ostatecznie otrzymujemy równanie, pozwalające wyznaczyć nam optymalny wektor parametrów θ:

$$\theta = (\dot{X}^T\dot{X})^{-1}\dot{X}^T y \tag{1.13}$$

1.6 Metoda gradientu prostego

Niestety dla większości modeli uczenia maszynowego nie jesteśmy w stanie podać wzoru na rozwiązanie jawne. Jedyną metodą jaka nam wówczas pozostaje to iteracyjne poszukiwanie parametrów optymalizujących naszą funkcję straty. Jedną z najczęściej stosowanych metod jest metoda gradientu prostego (*gradient descent*).

W metodzie gradientu prostego rozpoczynamy poszukiwanie optymalnego wektora parametrów θ od wypełnienia go losowymi wartościami (z reguły z przedziału $(0, 1)$). Następnie, w kolejnych iteracjach algorytmu, wyliczamy wartość gradientu dla wybranej funkcji kosztu i aktualizujemy wartość θ przeciwnie do kierunku gradientu:

$$\theta' = \theta - \eta\nabla_\theta MSE(\theta) \tag{1.14}$$

gdzie η to tzw. współczynnik uczenia (*learning rate*), określający tempo dokonywanych zmian. Zbyt niski współczynnik uczenia powoduje, że uczenie przebiega bardzo powoli, z kolei zbyt wysoki sprawi że algorytm nie będzie w stanie zbiec do minimum. Optymalna wartość współczynnika uczenia jest zależna od danych i w celu jego określenia musimy przeprowadzać treningi wiele razy, dla różnych jego wartości. Poszukiwania z reguły rozpoczynamy od wartości 0.1, a następnie próbujemy kolejne niższe potęgi 10: 0.01, 0.001 itd.

Liczba iteracji przez którą należy wykonywać algorytm, również zależy od problemu i jest ściśle powiązana z wartością współczynnika uczenia. Trening modelu kończymy kiedy kolejne iteracje nie zmniejszają już naszej funkcji kosztu.

1.7 Standaryzacja

Czasami kiedy różnice pomiędzy skalami wartości poszczególnych zmiennych są duże, trening z wykorzystaniem metody gradientu prostego staje się niemożliwy. Rozwiązaniem tego problemu jest normalizacja danych. Jedną z najczęściej spotykanych technik normalizacyjnych stanowi standaryzacja Z (*Z-score normalization*). W standaryzacji Z naszym celem jest przekształcenie zmiennych w taki sposób, aby wartość średnia z ich populacji wyniosła 0, a odchylenie standardowe 1. Dokonujemy tego w następujący sposób:

$$z = \frac{x - \mu}{\sigma} \tag{1.15}$$

gdzie x to pierwotna wartość zmiennej, μ to średnia z populacji, a σ to odchylenie standardowe populacji.

W przypadku uczenia maszynowego populację rozumiemy jako zbiór wszystkich wartości wybranej cechy w naszym zbiorze treningowym. Normalizacji danych z reguły dokonujemy zarówno dla zmiennych zależnych, jak i niezależnych. Należy pamiętać że w celu obliczenia średniej i odchylenia standardowego nie należy wykorzystywać zbioru treningowego ze względu na ryzyko wycieku danych testowych (*testing dataset leakage*). W celu pomiaru jakości modelu na zbiorze testowym, należy go znormalizować przy wykorzystaniu średniej i odchylenia standardowego wyliczonego na zbiorze treningowym.

Laboratorium 2

Algorytm genetyczny

Algorytm genetyczny (*genetic algorithm, GA*) to zbiorczy termin, którym określamy algorytmy optymalizacyjne inspirowane procesem doboru naturalnego. Nie stanowi on jednej konkretnej procedury, a zbiór różnych technik i praktyk, które mogą zostać wykorzystanie przy rozwiązywaniu konkretnego problemu. Algorytm genetyczny stanowi przykład metody heurystycznej, czyli nie dającej nam gwarancji znalezienia optymalnego rozwiązania, jednak próbującej się do niego jak najbardziej zbliżyć.

Aby zastosować algorytm genetyczny dla zadanego problemu optymalizacyjnego, muszą zostać spełnione dwa warunki:

1. dopuszczalne rozwiązania problemu muszą być możliwe do zakodowania w formie liczbowej,
2. musimy być w stanie zdefiniować tzw. funkcję dopasowania (*fitness*), która ocenia w sposób ilościowy jakość poszczególnych rozwiązań.

W przeciwieństwie do technik optymalizacyjnych wykorzystujących metodę gradientową, w algorytmie genetycznym nie wymagamy aby funkcja dopasowania była różniczkowalna. Umożliwia to między innymi na zastosowanie tego algorytmu do rozwiązywania problemów NP-trudnych, dla których z przyczyn wydajnościowych, często nie możemy wykorzystać algorytmów dokładnych.

2.1 Schemat działania

Algorytm genetyczny składa się z następujących kroków:

1. Tworzenie początkowej populacji rozwiązań
2. Wybór rodziców
3. Tworzenie kolejnego pokolenia
4. Mutacja
5. Aktualizacja populacji rozwiązań

Korki 2-5 powtarzamy aż do osiągnięcia zdefiniowanego kryterium stopu. W najprostszym przypadku może to być po prostu określona z góry liczba iteracji algorytmu.

2.1.1 Tworzenie początkowej populacji rozwiązań

W algorytmie genetycznym populacją nazywamy zbiór możliwych rozwiązań problemu optymalizacyjnego. Z kolei pojedyncze rozwiązanie nazywamy osobnikiem. W pierwszym kroku algorytmu generujemy początkowy zbiór możliwych rozwiązań problemu. Elementy do tego zbioru wybieramy w sposób losowy z przestrzeni wszystkich możliwych rozwiązań. Liczba osobników w początkowej populacji stanowi jeden z parametrów algorytmu.

2.1.2 Wybór rodziców

W algorytmie genetycznym nowe rozwiązania problemu generujemy na postawie łączenia ze sobą najlepszych dotychczasowych rozwiązań. Proces ten rozpoczynamy od wyboru z populacji najlepiej dopasowanych osobników (w sensie funkcji dopasowania), którzy następnie zostaną wykorzystani jako rodzice kolejnego pokolenia. Ten etap algorytmu nazywany jest selekcją. Liczba wybieranych rodziców stanowi parametr algorytmu.

Istnieje kilka popularnych algorytmów selekcji. Podstawowym problemem przed jakim one stają jest pogodzenie ze sobą dwóch sprzecznych wyzwań. Po pierwsze chcemy, aby jako rodzice kolejnego pokolenia wybierane były najlepiej dopasowane osobniki. Z drugiej strony, chcemy uniknąć sytuacji, w której zbyt szybko zawęzimy się jedynie do wąskiego grona obecnie najlepszych rozwiązań. Zbyt mała różnorodność populacji może w efekcie prowadzić do ugrzęźnięciem algorytmu w jednym z ekstremów lokalnych i uniemożliwić znalezienie globalnie najlepszego rozwiązania.

Selekcja ruletkowa

Selekcja ruletkowa (*roulette wheel selection*) stanowi jeden z najpopularniejszych algorytmów selekcji. W algorytmie tym każdemu osobnikowi w populacji przypisujemy prawdopodobieństwo wyboru, proporcjonalne do jego dopasowania:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{2.1}$$

gdzie f_i to wartość funkcji dopasowania i-tego osobnika, a n to wielkość naszej populacji. Wyliczone w ten sposób wartości tworzą nam rozkład prawdopodobieństwa. W celu przeprowadzenia selekcji losujemy wymaganą liczbę osobników z powstałego rozkładu. Należy zwrócić uwagę, że wyniku działania selekcji ruletkowej jeden osobnik może zostać wybrany wiele razy.

2.1.3 Tworzenie kolejnego pokolenia

Kolejnym etapem algorytmu genetycznego jest utworzenie kolejnego pokolenia osobników. W tym celu łączymy rodziców w pary. Na podstawie każdej z pary rodziców w tzw. procesie krzyżowania (*crossover*) tworzymy zbiór dzieci. Liczba tworzonych dzieci dla jednej pary rodziców jest zależna od konkretnej metody krzyżowania. Wynikiem tego etapu jest zbiór osobników kolejnego pokolenia.

2.1.4 Mutacja

Następny krok algorytmu genetycznego stanowi tzw. mutacja (*mutation*). W etapie tym wprowadzamy pewne drobne losowe zmiany w osobnikach z nowego pokolenia. Celem mutacji jest zwiększenie różnorodności naszej populacji, w celu rozszerzenia przestrzeni naszych poszukiwań.

2.1.5 Aktualizacja populacji rozwiązań

Ostatnim krokiem algorytmu genetycznego jest aktualizacja populacji rozwiązań. Możemy tutaj wykorzystać jeden z dwóch popularnych modeli: stanu ustalonego albo pokoleniowy. W modelu stanu ustalonego zachowujemy większość dotychczasowej populacji, zastępując nowymi osobnikami tylko część najsłabiej dopasowanych osobników. W celu odrzucenia najsłabiej dopasowanych możemy zastosować algorytm selekcji. Z kolei w modelu pokoleniowym zastępujemy całą dotychczasową populację nowymi osobnikami. Wybór konkretnego podejścia jest zależny od problemu.

Popularnym rozwinięciem etapu aktualizacji populacji jest wprowadzenie tzw. elityzmu. W aktualizacji z elityzmem do nowej populacji zawsze przenosimy pewną stałą liczbę najlepiej dopasowanych osobników z poprzedniej populacji.

2.2 Problem plecakowy

Problem plecakowy jest jednym z klasycznych problemów optymalizacyjnych. Wejściem dla problemu plecakowego jest maksymalny udźwig plecaka oraz lista przedmiotów z których każdy posiada swoją masę i wartość. Naszym zadaniem jest wybór podzbioru przedmiotów, który maksymalizował by sumaryczną wartość ładunku, jednocześnie nie przekraczając maksymalnego udźwigu plecaka.

Problem plecakowy jest problemem NP-trudnym, co oznacza, że prawdopodobnie nie istnieje algorytm, który rozwiązywałby go w czasie wielomianowym. Powoduje to, że znalezienie rozwiązania optymalnego szybko staje się niewykonalne już dla stosunkowo niewielkich rozmiarów listy wejściowej. Z tego względu z reguły musimy posiłkować się algorytmami heurystycznymi. Jedną możliwości jest wykorzystanie tutaj algorytmu genetycznego.

2.2.1 Kodowanie danych i funkcja dopasowania

Poszczególne rozwiązania dla problemu plecakowego można zakodować jako wektory binarne o długości równej wejściowej listy przedmiotów. Wektor taki posiada 1 na pozycjach odpowiadających przedmiotom załadowanym do plecaka, a 0 na pozostałych pozycjach. Funkcja dopasowania wygląda wówczas następująco:

$$fitness(x) = \begin{cases} \sum_{i=1}^{n} x_i v_i, & \text{if } \sum_{i=1}^{n} x_i w_i \leq W \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

gdzie $x_i \in \{0, 1\}$ oznacza decyzję o załadowaniu i-tego przedmiotu do plecaka, $v_i \in \mathbb{R}^+$ to wartość i-tego przedmiotu, $w_i \in \mathbb{R}^+$ to masa i-tego przedmiotu, $W \in \mathbb{R}^+$ to maksymalny udźwig plecaka, a n to liczba przedmiotów na

wejściu. W przytoczonym sformułowaniu problemu zakładamy możliwość pojawiania się rozwiązań, które przekraczają maksymalny udźwig plecaka, jednak przypisujemy im wówczas 0 wartość dopasowania. Powoduje to pewne zaszumienie zbioru możliwch rozwiązań, jednak z drugiej strony w znaczący sposób ułatwia zdefiniowanie operacji krzyżowania oraz mutacji.

2.2.2 Krzyżowanie

Najprostszym algorytmem krzyżowania jaki możemy zastosować dla problemu plecakowego jest tzw. krzyżowanie jednopunktowe. W tym algorytmie dzielimy wektory reprezentujące rodziców na dwie równe części. Następnie na ich podstawie tworzymy dwoje dzieci. Pierwsze z dzieci powstaje w wyniku połączenia pierwszej połówki wektora z pierwszego rodzica, z drugą połówką z drugiego rodzica. Drugie z dzieci powstaje w wyniku połączenia ze sobą dwóch niewykorzystanych połówek wektorów.

2.2.3 Mutacja

Jednym z wariantów mutacji w przypadku kodowania binarnego jest zamiana pojedynczego bitu w wektorze osobnika. W algorytmie tym wybieramy losowo jeden z elementów wektora, a następnie zastępujemy go negacją obecnej wartości.

Laboratorium 3

Algorytm min-max

Algorytm min-max służy wyszukiwaniu optymalnej strategii w dwuosobowych grach o sumie zerowej, czyli takich gdzie zysk jednego gracza oznacza stratę drugiego. Do tego typu gier można zaliczyć m.in. szachy, warcaby, czy go.

Algorytm min-max odpowiada nam na pytanie z jakim wynikiem zakończy się gra, zakładając, że zarówno my jaki i przeciwnik będziemy grali optymalnie. W swoim podstawowym wariancie algorytm min-max można zdefiniować jako funkcję rekurencyjną:

$$m(s,x) = \begin{cases} 1, & \text{if } win(s) \\ -1, & \text{if } loose(s) \\ 0, & \text{if } tie(s) \\ max(m(s'_1,0),\ldots,m(s'_n,0)), & \text{if } x = 1 \\ min(m(s'_1,1),\ldots,m(s'_n,1)), & \text{otherwise} \end{cases} \tag{3.1}$$

gdzie s to aktualny stan gry, n to liczba możliwych do podjęcia decyzji w aktualnym stanie, s'_i to nowy stan gry po podjęciu przez nas i-tej decyzji, a x to flaga określająca czy na danym poziomie zagłębienia wybieramy maksimum, czy minimum z możliwych wyników rozgrywki. W przypadku kiedy w aktualnym stanie s oczekujemy na nasz ruch powinniśmy zastosować funkcję z flagą $x = 1$, co oznacza, że wybieramy ruch najkorzystniejszy dla nas. W przeciwnym przypadku, kiedy decyzja należy do naszego przeciwnika, flaga $x = 0$, czyli zakładamy, że przeciwnik wybierze ruch najbardziej dla nas niekorzystny. Funkcje $win(\cdot)$, $loose(\cdot)$ oraz $tie(\cdot)$ odpowiadają nam kolejno na pytania, czy dana rozgrywka zakończyła się naszą wygraną, przegraną, czy remisem.

W praktyce bardziej niż przewidywanym wynikiem zakończenia gry, jesteśmy zainteresowani informacją, jaki ruch powinniśmy wykonać, znajdując się w danym stanie s, tak aby zmaksymalizować naszą szansę na wygraną. W tym celu możemy wykorzystać decyzyjną wersję algorytmu min-max:

$$md(s) = argmax((m(s'_1,0),\ldots,m(s'_n,0)) \tag{3.2}$$

Kolejne wywołania rekurencyjne funkcji $m(\cdot)$ budują drzewo reprezentujące wszystkie możliwe przebiegi rozrywki, które rozrasta się w tempie wykładniczym z każdym kolejnym stopniem zagłębienia. W praktyce więc stosuje się zmodyfikowany wariant funkcji $m(\cdot)$, który uwzględnia maksymalny dopuszczalny poziom zagłębienia przy tworzeniu drzewa gry:

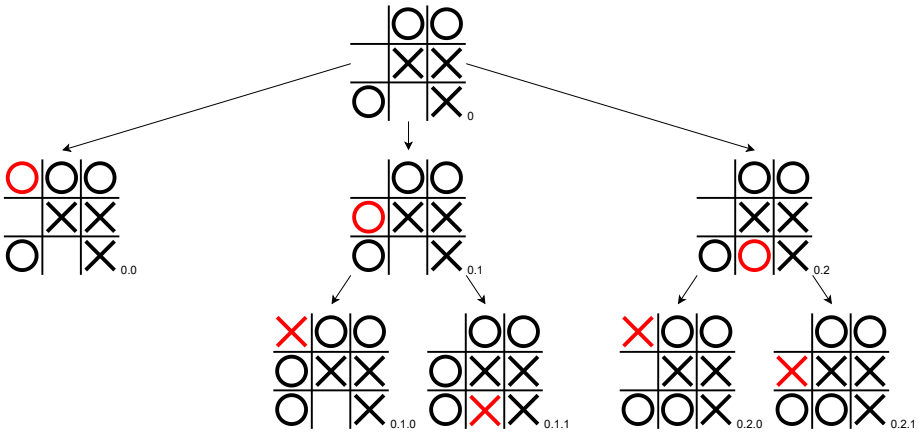
$$m'(s,x,d) = \begin{cases} 1, & \text{if } win(s) \\ -1, & \text{if } loose(s) \\ 0, & \text{if } tie(s) \\ 0, & \text{if } d = 0 \\ max(m'(s'_1,0,d-1),\ldots,m'(s'_n,0,d-1)), & \text{if } x = 1 \\ min(m'(s'_1,1,d-1),\ldots,m'(s'_n,1,d-1)), & \text{otherwise} \end{cases} \tag{3.3}$$

Funkcja $m'(\cdot)$ przyjmuje dodatkowy parametr d określający nam maksymalną głębokość wywołania rekurencyjnego. W przypadku kiedy po osiągnięciu maksymalnego dopuszczalnego poziomu zagłębienia wynik rozgrywki nie będzie ustalony, zwracamy wartość 0, podobnie jak w przypadku remisu. Analogicznie do (3.2), tutaj również możemy zdefiniować decyzyjną wersję algorytmu:

$$md'(s,d) = argmax((m'(s'_1,0,d-1),\ldots,m'(s'_n,0,d-1)) \tag{3.4}$$

3.1 Przykład min-max

Algorytm min-max możemy na przykład wykorzystać do wyznaczenia optymalnej strategii gry w kółko i krzyżyk.



Rysunek 3.1: Drzewo gry wygenerowane przez algorytm min-max dla zadanego stanu początkowego, przy maksymalnej głębokości $d = 2$.

Wyobraźmy sobie, że rozgrywamy partię w kółko i krzyżyk. Partię rozpoczęliśmy jako kółko i następnie w wyniku serii ruchów dotarliśmy do sytuacji przedstawionej na rysunku 3.1. W celu wyznaczenia optymalnego ruchu wykorzystajmy decyzyjną wersję algorytmu min-max. Załóżmy, że maksymalna głębokość na jakiej chcemy przeszukiwać nasze drzewo wynosi $d = 2$.

W wyniku kolejnych wywołań rekurencyjnych algorytm min-max tworzy nam drzewo gry przedstawione na rysunku 3.1. W celu wyboru optymalnego ruchu dla stanu s_0 musimy znaleźć wartości funkcji $m'(\cdot)$ dla dzieci węzła początkowego: $s_{0.0}$, $s_{0.1}$, $s_{0.2}$, a następnie wybrać węzeł o największej wartości (3.4). Węzeł $s_{0.0}$ reprezentuje stan końcowy, dla którego wartość funkcji $m'(s_{0.0},0,1) = 1$,

natomiast wartości funkcji $m'(\cdot)$ dla $s_{0.1}$, $s_{0.2}$ będzie wynosiła odpowiednio: $min(m'(s'_{0.1.0}, 1, 0), m'(s'_{0.1.1}, 1, 0))$ oraz $min(m'(s'_{0.2.0}, 1, 0), m'(s'_{0.2.1}, 1, 0))$. $s'_{0.1.0}$ jest stanem końcowym, dla którego $m'(s'_{0.2.0}, 1, 1) = -1$ (wygrana przeciwnika), z kolei dla $s'_{0.1.1}$ wartość funkcji $m'(s'_{0.1.1}, 1, 0) = 0$, ponieważ osiągnęliśmy maksymalny możliwy poziom zagłębienia. Ostatecznie $m'(s_{0.1}, 0, 1) = 1$. Postępując analogicznie możemy wyliczyć wartość funkcji $m'(\cdot)$ dla $s_{0.2}$. Po przeanalizowaniu całego drzewa gry, otrzymujemy końcowe wartości funkcji $m'(\cdot)$ dla $s_{0.0}$, $s_{0.1}$, $s_{0.2}$. Spośród nich wybieramy ruch $s_{0.0}$, który zapewnia nam natychmiastową wygraną.

3.2 Heurystyczna ocena stanu

Ponieważ w praktyce maksymalna głębokość analizowanego drzewa gry dla algorytmu min-max jest mocno ograniczona, może zdarzyć się tak, że zakończymy przeszukiwanie zanim dotrzemy do jakiegokolwiek stanu końcowego. W takiej sytuacji dla naszego algorytmu wszystkie możliwe do wykonania ruchy są nierozróżnialne. W rzeczywistości jednak wiemy, że pewne posunięcia są lepsze od innych, nawet jeśli nie prowadzą w krótkiej perspektywie do zwycięstwa bądź uniknięcia porażki. Naszą wiedzę dziedzinową na temat konkretnej gry, możemy przenieść do algorytmu min-max w postaci tzw. heurystycznej oceny stanu.

Heurystyczną ocenę stanu definiujemy jako funkcję $h(s) \in (-1, 1)$. Funkcja ta przyjmuje jako argument aktualny stan gry i zwraca nam liczbę rzeczywistą w przedziale od -1 do 1. Wartości od -1 do 0 interpretujemy jako wskazanie na przewagę naszego przeciwnika, natomiast od 0 do 1 jako wskazanie na naszą przewagę. Ważnie jest aby wartości zwracane przez funkcję heurystyczną nigdy nie przekroczyły -1 oraz 1, ponieważ chcemy aby nasz algorytm w pierwszej kolejności kierował się dążeniem do zwycięstwa oraz unikaniem porażki.

Wersja algorytmu min-max rozwinięta o heurystyczną ocenę stanu wygląda następująco:

$$mh(s, x, d) = \begin{cases} 1, & \text{if } win(s) \\ -1, & \text{if } loose(s) \\ 0, & \text{if } tie(s) \\ h(s), & \text{if } d = 0 \\ max(mh(s'_1, 0, d-1), \ldots, mh(s'_n, 0, d-1)), & \text{if } x = 1 \\ min(mh(s'_1, 1, d-1), \ldots, mh(s'_n, 1, d-1)), & \text{otherwise} \end{cases} \tag{3.5}$$

Definicja konkretnej funkcji heurystycznej zależy od rozważanej gry. W przypadku gry w kółko i krzyżyk możemy np. faworyzować zajmowanie pozycji na rogach planszy:

$$h_{tictactoe}(s) = 0,2 * \text{liczba zajętych przez nas narożników} \tag{3.6}$$

3.3 Algorytm alfa-beta

Algorytm alfa-beta jest rozszerzeniem standardowego algorytmu min-max o dodatkową optymalizację, wprowadzającą przerywanie przeszukiwania wybranych

ścieżek drzewa gry w sytuacjach kiedy nie ma to sensu. Algorytm alfa-beta wprowadza dwa dodatkowe parametry dla rekurencyjnej funkcji min-max: α oraz β. Parametr α przechowuje największą znalezioną wartość wierzchołka w drzewie gry podczas maksymalizacji (nasz ruch), parametr β natomiast przechowuje najmniejszą wartością znalezioną podczas minimalizacji (ruch przeciwnika). Oba parametry przekazywane są rekurencyjnie w dół drzewa przeszukiwań. Dysponując podczas minimalizacji wiedzą na temat tego jaką największą wartość wierzchołka (α) uzyskaliśmy na wyższym poziomie drzewa gry, możemy przerwać dalsze przeszukiwania, jeżeli dotychczasowa znaleziona wartość jest mniejsza niż α, ponieważ niezależnie od pozostałych wartości wierzchołków poziomu minimalizacji, na wyższym poziomie i tak wybierzemy wierzchołek dający nam wyższą wartość. Analogiczne rozumowanie możemy zastosować podczas maksymalizacji. Algorytm 3.1 przedstawia implementację algorytmu alfa-beta razem z odcinaniem na zadanej głębokości oraz heurystyczną oceną stanu.

Algorytm 3.1 Algorytm alfa-beta

```
function ALPHABETA(s, x, d, α, β)
    if win(s) then
        return 1
    else if loose(s) then
        return −1
    else if tie(s) then
        return 0
    else if d=0 then
        return h(s)
    else if x=1 then
        v ← −∞
        for i = 1, ..., n do
            v ← max(v, ALPHABETA(s'_i, 0, d − 1, α, β))
            α ← max(α, v)
            if v ≥ β then
                break ▷ β cutoff
        return v
    else
        v ← ∞
        for i = 1, ..., n do
            v ← min(v, ALPHABETA(s'_i, 1, d − 1, α, β))
            β ← min(β, v)
            if v ≤ α then
                break ▷ α cutoff
        return v
```

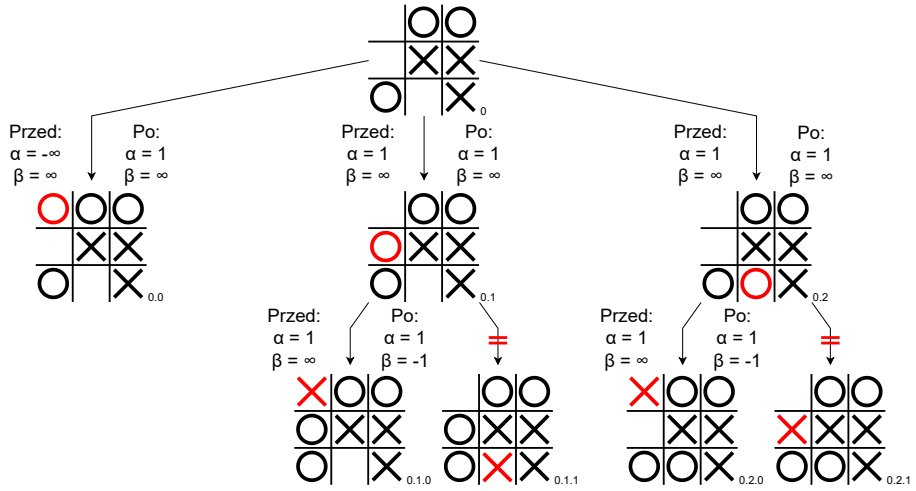
Analogicznie do algorytmu min-max, możemy również zdefiniować decyzyjną wersję alfa-beta (3.2).

3.4 Przykład alfa-beta

Działanie algorytmu alfa-beta możemy zaprezentować na przykładzie z rozdziału 3.1. Wygenerowane przez algorytm drzewo gry pokazane jest na rysunku 3.2.

Algorytm 3.2 Decyzyjna wersja algorytmu alfa-beta

function ALPHABETAD(s, x, d)
    $\alpha \leftarrow -\infty$
    $v \leftarrow -\infty$
    $y \leftarrow nothing$ ▷ *best possible move*
    for $i = 1, \ldots, n$ **do**
        $v_{s'_i} \leftarrow$ ALPHABETA$(s'_i, 0, d-1, \alpha, \infty)$
        if $v_{s'_i} > v$ **then**
            $v \leftarrow v_{s'_i}$
            $y \leftarrow i$
        $\alpha \leftarrow max(\alpha, v)$
    return y



Rysunek 3.2: Drzewo gry wygenerowane przez algorytm alfa-beta dla zadanego stanu początkowego, przy maksymalnej głębokości $d = 2$.

Analizę naszego drzewa gry rozpoczynamy od węzła 0.0. Ponieważ jest to węzeł końcowy funkcja ALPHABETA zwróci nam jego wartość równą 1. Następnie ponieważ 1 jest większa niż $-\infty$, aktualizujemy wartość α i przechodzimy do analizy kolejnego węzła. Węzeł 0.1 nie jest węzłem końcowym, także przed określeniem jego wartości, musimy przeanalizować wartości jego dzieci. W pierwszej kolejności schodzimy w dół do węzła 0.1.0. Węzeł 0.1.0 jest węzłem końcowym, którego wartość wynosi -1 (wygrana przeciwnika). Po przeanalizowaniu tego węzła aktualizujemy parametr β nową wartością i sprawdzamy warunek α odcięcia. Ponieważ $-1 \leq 1$ przerywamy dalsze przeszukiwanie na tym poziomie. Dzieje się tak, ponieważ po znalezieniu wartości -1, wiemy że na poziomie minimalizacji zwrócimy tę wartość lub jeszcze mniejszą. Ta wartość jednak i tak nie zostanie wybrana na wyższym poziomie maksymalizacji, ponieważ znaleźliśmy już tam wartość większą od niej, równą 1. Tym samym analiza dalszych gałęzi i tak nie zmieni finalnego wyniku. Z podobną sytuacją mamy do czynienia przy analizie węzła 0.2.

Laboratorium 4

Klasyfikacja

Klasyfikacja polega na przydzieleniu każdej z obserwacji prawdopodobieństwa z którym obserwacja ta przynależy do jeden z kilku predefiniowanych klas. Rozróżniamy klasyfikację binarną, w przypadku której mamy tylko dwie klasy i multiklasową, gdzie liczba klas jest skończona, ale większa niż dwie. Przykładem klasyfikacji binarnej może być odróżnienie zdjęć na których występują psy, od tych na których widzimy koty, pozytywnego i negatywnego sentymentu zdania dla danych tekstowych, czy przewidywanie prawdopodobieństwa rezygnacji klienta z jakiejś usługi.

Modele klasyfikacyjne, podobnie jak regresyjne, uczone są w sposób nadzorowany. Zakłada on, że posiadamy zbiór uczący zawierający obserwacje wraz z etykietami. W procesie uczenia chcemy aby nasz model przewidywał z jak najlepszą dokładnością dane ze zbioru treningowego, zachowując jednocześnie zdolności do generalizacji na niewidzianych wcześniej danych.

4.1 Ocena jakości modelu

Pełną informację o jakości modelu klasyfikacyjnym możemy uzyskać konstruując tablice pomyłek (ang. confusion matrix). Jest to macierz, dalej oznaczana literą C, o wielkości $K \times K$, gdzie K oznacza liczbę wszystkich klas. Tablice pomyłek tworzymy przy pomocy danych testowych wpisując w pole (i,j) liczbę obserwacji zaklasyfikowanych przez nasz model do klasy j, a dla których klasa prawdziwa to i. W ten sposób, na przekątnej takiej tablicy będzie znajdowała się liczba obserwacji poprawnie zaklasyfikowanych, a poza przekątną - błędnie. Dodatkowo, z takiej tablicy jesteśmy w stanie łatwo oczytać które klasy były najczęściej mylone i wykorzystać tę wiedzę do poprawy jakości modelu.

W oparciu o macierz pomyłek zostało zaproponowanych wiele innych metryk, które próbują wyrazić jakoś modelu w postaci skalaru. Jedną z najbardziej rozpowszechnionych jest dokładność (ang. accuracy). Accuracy mierzy się poprzez zsumowanie elementów na przekątnej tablicy pomyłek i podzielenia przez liczbę wszystkich obserwacji:

$$acc := \frac{\sum_{k=1}^{K} C[k,k]}{\sum_{i=1}^{K} \sum_{j=1}^{K} C[i,j]} \tag{4.1}$$

4.2 Drzewa decyzyjne

Drzewa decyzyjne są algorytmem wykorzystywanym zarówno do problemów regresji, jak i klasyfikacji. Wyuczone drzewo decyzyjne jest ukorzenionym drzewem binarnym, w którym każdy wierzchołek (ang. node) (poza liśćmi) jest stopnia 2. Wierzchołki drzewa zawierają warunek, który determinuje czy obserwacja powinna w kolejnym kroku trafić do lewego, czy do prawego dziecka. Wnioskowanie przy użyciu drzewa decyzyjnego odbywa się poprzez znalezienie drogi od korzenia do liścia i przydzielenie rozpatrywanej obserwacji etykiety najczęściej występującej w zbiorze treningowym w znalezionym liściu.

4.2.1 Tworzenie nowych węzłów

Proces uczenia drzewa decyzyjnego zaczyna się od korzenia i postępuje rekurencyjnie dla każdego kolejnego wierzchołka aż do osiągnięcia warunku stopu. Korzeń, mając do dyspozycji wszystkie dane treningowe, wybiera jedną zmienną X_i oraz jeden punkt x, który najlepiej rozdziela nam etykiety. Następnie, dane treningowe które spełniają warunek $X_i < x$ trafiają do lewego dziecka, a pozostałe do prawego. Proces ten jest kontynuowany aż do osiągnięcia żądanej głębokości lub do momentu, gdy liczba obserwacji treningowych w wierzchołku przekroczyła wartość minimalną do dalszego podziału.

Wybór miejsca podziału

W celu wybrania optymalnego miejsca podziału należy sprawdzić wszystkie możliwe podziały. Dla każdego wymiaru (cechy) danych treningowych należy posortować jej wartości i wybrać miejsca podziału znajdujące się pomiędzy dwoma następującymi po sobie wartościami. Następnie dzielimy dane zgodnie z tym podziałem i liczmy średnią (ważoną ilością obserwacji) wartość zanieczyszczenia (ang. impurity) etykiet, które znalazłyby się w lewym i prawym dziecku gdybyśmy dokonali takiego podziału.

Kryterium Giniego

Chcąc sprawdzić zysk informacyjny z proponowanego podziału można skorzystać z kryterium Giniego. Zakładając, że nasz proponowany podział sprawiłby, że w lewym dziecku byłoby $left\_pos$ obserwacji posiadających etykietę 1 i $left\_neg$ obserwacji z etykietą 0, oraz $right\_pos$ i $right\_neg$ analogicznie dla prawego dziecka, zysk informacyjny wyniósłby:

$$gini\_gain = 1 - \frac{left}{left + right} * gini\_left - \frac{right}{left + right} * gini\_right \quad (4.2)$$

$$gini\_left = 1 - (\frac{left\_pos}{left\_pos + left\_neg})^2 - (\frac{left\_neg}{left\_pos + left\_neg})^2 \quad (4.3)$$

$$gini\_right = 1 - (\frac{right\_pos}{right\_pos + right\_neg})^2 - (\frac{right\_neg}{right\_pos + right\_neg})^2 \quad (4.4)$$

, gdzie $left = left\_pos + left\_neg$ i $right = right\_pos + right\_neg$.

Wyższy Gini gain oznacza czystszy (lepszy) podział.

4.3 Lasy losowe

Drzewa decyzyjne posłużyły za podstawę wielu innych algorytmów uczenia maszynowego. Jednym z nich są lasy losowe (ang. random forest). W lasach losowych tworzymy wiele (kilkaset) drzew decyzyjnych niezależnie od siebie i używamy metody głosowania większościowego w procesie wnioskowania. Korzystanie z wielu drzew ma zapewnić większą stabilność (mniejszą wariancję) modelu, a tym samym poprawić jego jakość. Jest to możliwe tylko wtedy, gdy drzewa różnią się od siebie i w tym celu wprowadzamy pewną losowość w tworzenie każdego z nich. Obywa się to poprzez losowanie ze zwracaniem (bagging) podzbioru danych treningowych dla każdego z drzew oraz wprowadzeniem losowego ograniczenia na cechy z których wierzchołek może korzystać przy szukaniu najlepszego podziału.

4.3.1 Liczba drzew

Liczba drzew jest parametrem liniowo wpływającym na czas uczenia i wnioskowania lasów losowych. Nie ma ona wpływu na ekspresywność modelu, to znaczy, że większa liczba drzew nie pozwoli modelowi się lepiej dopasować do danych. Zazwyczaj możemy zaobserwować odwrotną zależność, czyli wraz ze wzrostem liczby drzew zwiększa się stabilność modelu i ciężej jest doprowadzić do nadmiernego dopasowania (ang. overfitting). Powyżej pewnej liczby drzew (zazwyczaj kilkuset) przestajemy zauważać wpływ dodawania kolejnych.

4.3.2 Bagging

Bagging ma zapewnić różnorodność danych, na których będą uczone poszczególne drzewa. Średnio każdy podzbiór wylosowany poprzez losowanie ze zwracaniem o wielkości oryginalnego zbioru będzie zawierał około 63% unikalnych obserwacji. Oznacza to, że jeśli każde z naszych drzew będzie odpowiednio ekspresywne żeby interpolować dane treningowe, to las losowy jest w stanie również dopasować się idealnie do zbioru treningowego. Uniknięcie takiej sytuacji jest możliwe poprzez wybranie odpowiednich parametrów drzewa decyzyjnego, które będą w stanie dobrze oddawać zależność pomiędzy cechami a etykietami i jednocześnie nie dopuszczając do nadmiernego dopasowania.

4.3.3 Wielkość podzbioru cech

Drugą techniką mającą zapewnić zróżnicowanie drzew decyzyjnych jest losowanie cech w wierzchołkach. Polega ona na usuwaniu na każdym etapie uczenia drzewa decyzyjnego pewnej liczby cech spośród których zostanie wybrana ta, która podzieli nam przestrzeń cech w wybranym wierzchołku. Liczba cech, które mogą zostać wybrane do stworzenia wierzchołka jest hiper-parametrem lasów losowych, ale jedną z heurystyk wykorzystywanych do jej inicjalizacji jest wartość pierwiastka kwadratowego z liczby wszystkich cech.

Laboratorium 5

Klasteryzacja

Klasteryzacja należy do kategorii uczenia nienadzorowanego, czyli grupy algorytmów uczenia maszynowego nie potrzebujących etykiet w procesie uczenia. Zadaniem klasteryzacji jest znalezienie takiego podziału zbioru danych, który połączy ze sobą obserwacje podobne. Definicje podobieństwa różnią się w zależności od wybranej metody, jednak w większosci wypadków będą opartę o miarę odległości w przestrzeni danych np. odległość euklidesową.

5.1 K-means

Jednym z podstawowych algorytmów klasteryzacji jest k-means. Wyróżnia się niską złożonością obliczeniową, co sprawia, że jest często wykorzystywanym algorytmem klasteryzacji w praktyce. W procesie uczenia, k-means stara się znaleźć punkty w przestrzeni obserwacji, które stanowią centra poszczególnych klastrów. Metoda ta wymaga podania przez użytkownika żadanej liczby klastrów i startując z losowych pozycji, iteracyjnie poprawia ich położenie. Cały algorytm k-means składa się z trzech częsci: zainicjalizowania wartości poczatkowych centroidów, przypisania wszystkich obserwacji do najbliższych centroidów oraz zaktualizowanie pozycji centroidów. Problem znalezienia pozycji centroidów, które będą minimalizowały odległość obserwacji od najbliższego centroida jest NP-trudny. Pierwszy krok (inicjalizacja) okazuje się bardzo istotny w znalezieniu dobrego minimum lokalnego k-means. Kolejne dwa kroki (przypisanie i aktualizacja) wykonywane są w pętli, do momentu zbiegnięcia do lokalnego minimum (żadna obserwacja nie zmieni swojej przynależności do klastra pomiędzy dwiema kolejnymi iteracjami).

5.1.1 Inicjalizacja - Forgy

Najprostszy sposób inicjalizacji k-means polega na wylosowaniu centroidów spośród obserwacji. Losowanie powinno odbyć się bez zwracania i wykorzystywać rozkład jednostajny (każda obserwacja ma takie samo prawdopodobieństwo bycia wylosowaną).

5.1.2 Inicjalizacja - kmeans++

Częstym problemem w algorytmie Forgy'a jest wylosowanie kilku centroidów w niewielkiej od siebie odległości. Jednym ze sposobów radzenia sobie z tym jest kmeans++, w którym kolejne centroidy wybierane są tak, aby znajdowały się jak najdalej od tych już wybranych. Algorytm inicjalizacji kmeans++ zaczyna się od wylosowania spośród obserwacji jednego centroida, a następnie dodawania kolejnych pojedyńczo. Dodawana jest zawsze ta obserwacja, której odległość od obecnych centroidów jest największa spośród wszystkich dostępnych obserwacji. W ten sposób zapewniamy dużo lepszy rozrzut początkowych centroidów i ułatwiamy algorytmowi zbieżność do lepszego minimum.

5.1.3 Aktualizacja centroidów

Aktualizacja centroidów polega na policzeniu średniej ze wszystkich obserwacji, dla których dany centroid był najbliższy.

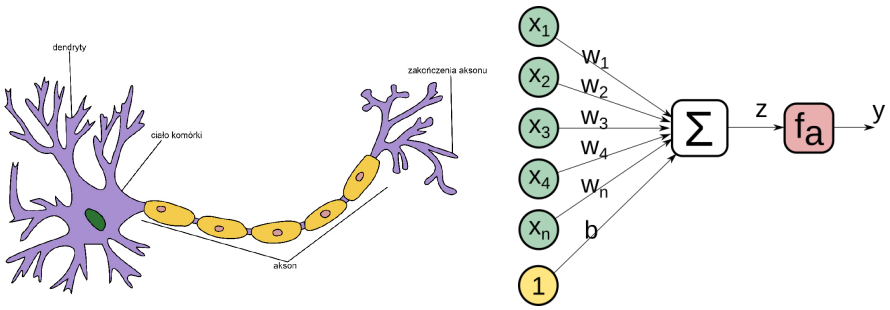
5.1.4 Ocena jakości podziału

K-means minimalizuje średnią odległość euklidesową obserwacji od ich najbliższych centroidów, dlatego tę metrykę powinniśmy kontrolować w procesie uczenia. W zależności od inicjalizacji centroidów możemy otrzymywać różne jej wartości. W celu ograniczenia ryzyka wylosowania złych punktów startowych, częstą praktyką jest uruchamianie k-means wielokrotnie i porównywanie średniej odległości od centroidów pomiędzy nimi.

Laboratorium 6

Sztuczne sieci neuronowe

Sztuczne sieci neuronowe to struktury danych wraz z towarzyszącymi im algorytmami treningu/uczenia, służące modelowaniu zależności występujących w danych. Z matematycznego punktu widzenia, sieci neuronowe stanowią pewną rodzinę parametryzowalnych funkcji mapujących punkty w przestrzeni danych wejściowych na punkty w przestrzeni danych wyjściowych, gdzie o konkretnej postaci funkcji decyduje architektura sieci oraz jej parametry – pewne wartości liczbowe typowo ustawiane w procesie uczenia sieci (choć, jak pokaże niniejsza instrukcja, w przypadku prostych sieci, parametry te można też wydedukować i ustawić "ręcznie"). Sieci mają budową modularną, tj. można je w dość swobodny sposób komponować z tzw. warstw neuronów lub nawet pojedynczych neuronów. Z tej racji, występują one w bardzo wielu wariantach - od bardzo prostych do bardzo złożonych, mających praktycznie nieograniczone możliwości modelowania - i znajdują zastosowanie w rozwiązywaniu bardzo wielu problemów praktycznych. W niniejszej instrukcji ograniczymy się jedynie do zaprezentowania podstawowej architektury sztucznej sieci neuronowej, stanowiącej punkt wyjścia, ale też często ważny element konstrukcyjny, dla innych, bardziej zaawansowanych architektur.



(a) Inspiracja: neuron biologiczny.

(b) Matematyczny model neuronu.

Rysunek 6.1: Pojedynczy neuron.

6.1 Model pojedynczego neuronu

Model pojedynczego sztucznego neuronu jest inspirowany sposobem działania biologicznego neuronu (rys. 6.1a), który, w ujęciu mocno uproszczonym, *odpala* sygnał wyjściowy swoim aksonem, jeśli sumaryczna wartość pobudzeń synaptycznych (o różnej wrażliwości) na jego wejściach (dendrytach) przekroczy pewną krytyczną wartość. Pojedynczy sztuczny neuron (rys.6.1b), stanowiący podstawowy budulec sztucznych sieci neuronowych, to bardzo prosty matematyczny model tego biologicznego zjawiska:

$$y = f_a(z) = f_a\left(\sum_i^n x_i w_i + b\right) = f_a(\mathbf{x}\mathbf{w} + b), \tag{6.1}$$

gdzie $\mathbf{x}$ to n-wymiarowe wejście neuronu:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \in \mathcal{R}^{[1\times n]},$$

$\mathbf{w}$ to *wagi* poszczególnych wejść neuronu:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \in \mathcal{R}^{[n\times 1]},$$

$b \in \mathcal{R}$, to tzw. *bias* (*obciążenie*) neuronu, a f_a to tzw. *funkcja aktywacji*. Przeważona suma wejść, która jest argumentem funkcji aktywacji, oznaczona jako z, nazywana jest często *logitem*. Wektor wag $\mathbf{w}$ oraz bias b to *parametry* neuronu, tj. wartości liczbowe, które, przeważnie, ustawiane są w trakcie uczenia w celu optymalizacji działania modelu w zadanym problemie (więcej o tym w sekcji 6.3).

Ponieważ współczesne komputery przystosowane są do wydajnego zrównoleglania obliczeń, ze względów wydajnościowych często oblicza się wyjście neuronu nie dla pojedynczego wektora wejściowego $\mathbf{x}$, ale dla *paczki* (ang. *batch*) danych:

$$\mathbf{y} = f_a\left(\mathbf{X}\mathbf{w} + broadcast(b)\right), \tag{6.2}$$

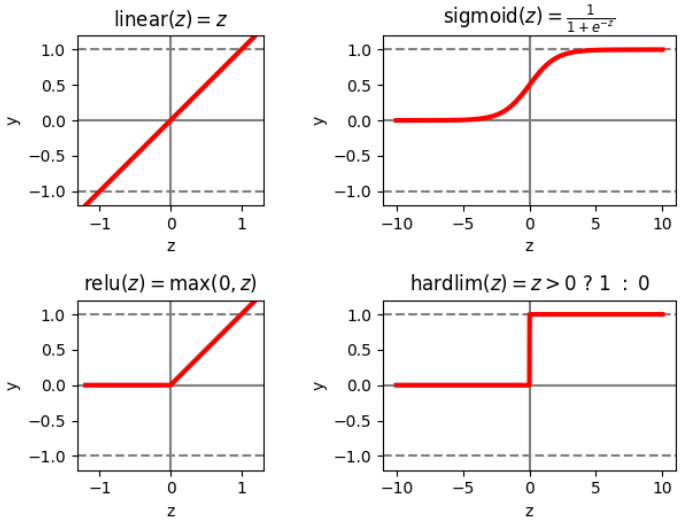
gdzie $\mathbf{X}$ to macierz paczki danych wejściowych o rozmiarze $BS\times n$ (BS – rozmiar paczki (ang. *batch size*)):

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \\ \vdots \\ \mathbf{x}^{(BS)} \end{bmatrix} \in \mathcal{R}^{[BS\times n]}.$$

Uzyskany wektor $\mathbf{y} \in \mathcal{R}^{[BS\times 1]}$ na i-tej pozycji zawiera odpowiedź neuronu na wejście $\mathbf{x}^{(i)}$.

6.1.1 Przykładowe funkcje aktywacji

Na rys. 6.2 przedstawiono kilka (z bardzo wielu) popularnych funkcji aktywacji:



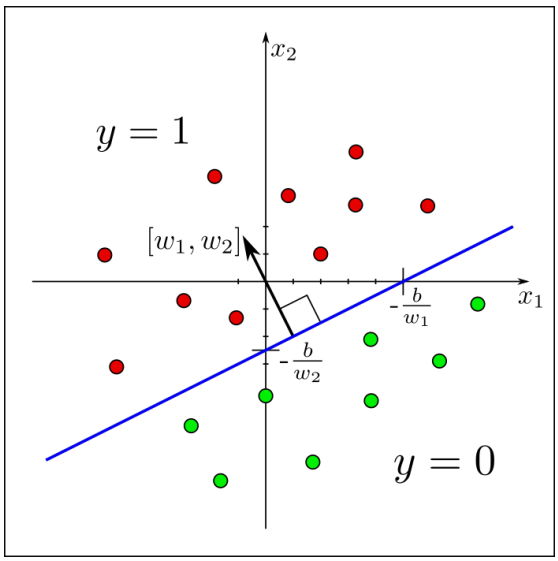
Rysunek 6.2: Przykładowe funkcje aktywacji - wykresy i równania.

- *liniowa* – oznacza, że wyjście neuronu jest równe logitowi, neuron oblicza jedynie kombinację liniową wejść i nie wykonuje żadnego nielinowego przekształcenia (można zauważyć, że taki neuron jest wtedy równoważny modelowi prostej regresji linowej, znanej z rozdziału 1);
- *sigmoid* – ogranicza wyjście neuronu do przedziału $(0, 1)$, dzięki czemu takie wyjście można interpretować jako znormalizowaną reakcję neuronu na wejście, gdzie 1 oznacza maksymalne "pobudzenie";
- *relu* – w szybki sposób dokonuje nieliniowego przekształcenia logitu;
- *hardlim* – używana dla ułatwienia analizy działania neuronów, ale nie w trenowanych sieciach neuronowych, gdyż jej pochodna, w każdym punkcie, wynosi 0.

6.1.2 Interpretacja geometryczna

Korzystając z równania 6.1, przy założeniu, że neuron ma dwa wejścia i funkcję aktywacji hardlim, łatwo można zauważyć, że pojedynczy neuron rozdziela swoją przestrzeń wejściową linią prostą (w ogólności, dla n-wymiarowego wejścia, jest to n-wymiarowa hiperpłaszczyzna), tak jak to pokazano na rys. 6.3. Jej kierunek jest wyznaczony przez wektor wag $\mathbf{w}$, jest ona zawsze ortogonalna do tego wektora, a punkty przecięcia z osiami wyznaczają wagi oraz *bias* neuronu (bez biasu, linia ta zawsze musiałaby przechodzić przez punkt początkowy układu współrzędnych).

Ta linia (hiperpłaszczyzna), to tzw. *linia decyzyjna*: na wszystkie wejścia, które w przestrzeni wejściowej są reprezentowane przez punkty leżące po tej stronie linii decyzyjnej, co wektor $\mathbf{w}$, neuron odpowie wartością $y = 1$, a na wejścia, które wpadają po drugiej stronie linii, odpowie wartością $y = 0$. Oznacza to, że może pojedynczy neuron jest w stanie z sukcesem realizować rolę binarnego



Rysunek 6.3: Interpretacja neuronu o dwóch wejściach i funkcji aktywacji hardlim: taki neuron rozwiązuje problemy liniowo-separowalne.

klasyfikatora w sytuacji, w której klasy dają się rozdzielić linią prostą. Mówimy wtedy, że jest to wtedy problem *liniowo-separowalny*.

6.2 Model wielowarstwowej sieci neuronowej

Oczywiście większość praktycznych problemów jest znacznie trudniejsza i należy do klasy problemów nieliniowo-separowalnych. Aby takie zadanie rozwiązać potrzebny jest bardziej złożony model: sieć neuronowa.

6.2.1 Warstwa gęsta neuronów

Ze względów praktycznych, większe modele buduje się najczęściej nie z pojedynczych neuronów, ale z *warstw* neuronów. Warstwa to grupa neuronów tego samego typu (mających taką samą liczbę wejść i taką samą funkcję aktywacji), operujących w sposób niezależny od siebie, przez co mogąca pracować w sposób zrównoleglony. Podstawową warstwą neuronową jest tzw. warstwa *gęsta*, przedstawiona na rys. 6.4, a realizująca formułę:

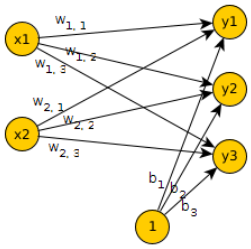
$$\mathbf{y} = f_a\left(\mathbf{x}\mathbf{W} + \mathbf{b}\right), \tag{6.3}$$

gdzie macierz $\mathbf{W}$ to macierz wag na połączeniach między n_{in} wejściami, a n_{out} neuronami tej warstwy:

$$\mathbf{W} = \left[\mathbf{w}_1\ \mathbf{w}_2\ \ldots\ \mathbf{w}_{n_{\text{out}}}\right] \in \mathcal{R}^{[n_{\text{in}} \times n_{\text{out}}]},$$

a $\mathbf{b}$ to wektor obciążeń neuronów:

$$\mathbf{b} = \left[b_1\ b_2\ \ldots\ b_{n_{\text{out}}}\right] \in \mathcal{R}^{[1 \times n_{\text{out}}]}.$$



Rysunek 6.4: Warstwa gęsta neuronów posiadająca 2 wejścia i 3 wyjścia.

Macierz $\mathbf{W}$ jest gęsta (o ile celowo nie zrównamy większości wag z zerem), co odpowiada gęstym połączeniom (każdy z każdym) pomiędzy wejściami, a wyjściami warstwy – stąd też nazwa warstwy. Można zauważyć, że łączna liczba parametrów warstwy gęstej to $(n_{\text{in}} + 1) \cdot n_{\text{out}}$, uwzględniając zarówno wagi, jak i obciążenia.

Podobnie, jak w przypadku pojedynczego neuronu, także tutaj, w celach wydajnościowych, najczęściej oblicza się jednocześnie wyjście warstwy nie dla pojedynczego przykładu wejściowego, ale dla paczki danych:

$$\mathbf{Y} = f_a\left(\mathbf{X}\mathbf{W} + broadcast(\mathbf{b})\right),$$

a macierz $\mathbf{Y} \in \mathcal{R}^{[BS \times n_{\text{out}}]}$ w i-tym wierszu zawiera odpowiedź warstwy na i-ty przykład podany na wejściu (będący i-tym wierszem macierzy $\mathbf{X}$).

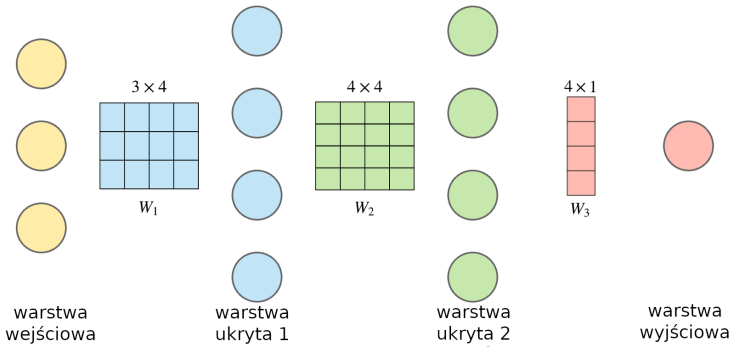
6.2.2 Sieć wielowarstwowa typu *Multi-Layer Perceptron* (MLP)

Podstawowym i bardzo uniwersalnym rodzajem sieci neuronowych jest sieć utworzona poprzez kaskadowe (sekwencyjne) połączenie warstw gęstych, tak, że wyjście warstwy poprzedniej jest wejściem warstwy następnej. Architektura taka nazywa jest często *wielowarstwowym perceptronem* – MLP (ang. *Multi-Layer Perceptron*). Przykładowo, na rys. 6.5 przedstawiono sieć, która jest złożona z kaskadowo połączonych trzech warstw gęstych. Pierwsza z nich przekształca 3-wymiarowe wejście w 4-wymiarową przestrzeń *ukrytą* (ang. *hidden space* lub *latent space*), tj. będącą wewnątrz modelu, normalnie niedostępną z zewnątrz. Druga warstwa transformuje tę 4-wymiarową przestrzeń w inną, także 4-wymiarową przestrzeń ukrytą. Ostatnia, trzecia warstwa oblicza skalarne wyjście modelu na podstawie tej ostatniej 4-wymiarowej reprezentacji podanego na wejściu przykładu.

Każda kolejna warstwa ukryta, o ile używa *nieliniowej* funkcji aktywacji, nieliniowo przekształca przestrzeń wejściową. W trakcie uczenia modelu, jednoczesna optymalizacja wszystkich parametrów modelu (wszystkich wag i biasów) sprawia, że te przekształcenia, z warstwy na warstwę, coraz bardziej upraszczają problem – tj. staje się on bliższy linowej separowalności.

Matematycznie, taką sieć opisać można równaniem:

$$\hat{y} = f_{a_L}\left(f_{a_{L-1}}\left(\dots f_{a_1}\left(\mathbf{W}_1\mathbf{x}_i + \mathbf{b}_1\right)\dots\mathbf{W}_{L-1} + \mathbf{b}_{L-1}\right)\mathbf{W}_L + \mathbf{b}_L\right) = f_\theta(\mathbf{x}), \tag{6.4}$$



Rysunek 6.5: Przykładowa sieć typu MLP. Na rysunku, dla uproszczenia, pominięto wektory obciążeń każdej z warstw.

gdzie L to liczba warstw modelu, a θ to symbol reprezentujący wszystkie parametry modelu zebrane w jeden wektor.

6.2.3 Dobór topologii sieci do zadanego problemu

Sieć MLP jest uniwersalnym narzędziem, które może rozwiązać zarówno problemy regresji, jak i klasyfikacji (różnego typu). Typ danych oraz rodzaj problemu narzucają pewne ograniczenia na topologię modelu:

- liczba wejść modelu – określona jest przez liczbę cech występującą w danych, tj. wymiarowość oryginalnych danych;
- liczba wyjść modelu – w przypadku regresji (zwracana jedna liczba rzeczywista, jak np. poziom spalania auta w *mpg*) wystarcza jeden neuron wyjściowy (z liniową funkcją aktywacji), w przypadku klasyfikacji binarnej (np. czy na zdjęciu jest pies czy kot) także jeden neuron wyjściowy wystarcza, który powinien być typu *sigmoid*, co pozwala interpretować wyjście jako wartość prawdopodobieństwa p przynależności przykładu wejściowego do klasy oznaczonej jako 1 (a $1 - p$ do klasy komplementarnej), a w przypadku klasyfikacji K-klasowej, powinno być K neuronów wyjściowych z funkcją aktywacji *softmax* (nie przedstawionej w tym opracowaniu);
- liczba i wielkość warstw ukrytych – są to *hiper*parametry modelu, dobierane przed treningiem, które wpływają na możliwości modelujące sieci (im więcej warstw i im większe, tym zdolności modelujące większe), choć trening może być wolniejszy oraz istnieje groźba przeuczenia (ang. *overfitting*) modelu.

Podsumowując, liczba wejść i wyjść modelu (oraz typ neuronów na wyjściu), jest ściśle określona przez zadanie. Natomiast liczbę i wielkość warstw ukrytych należy dobrać tak, aby wielkość modelu była odpowiednia dla zadanego problemu.

6.3 Trening sieci neuronowej

Nadzorowany trening sieci neuronowych ma na celu znalezienie takich wartości parametrów modelu, aby wyjścia sieci $\hat{y}$ możliwie blisko odpowiadały oczekiwanym wyjściom y, zgodnie z oznaczeniami, jakie występują w danych treningowych. Rolą funkcji kosztu (zob. 1.3) jest pomiar tego dopasowania – im jest ono lepsze, tym wartość funkcji kosztu jest mniejsza. Tym samym, zadanie treningu sieci neuronowych sprowadza się do problemu minimalizacji funkcji kosztu. Ze względów praktycznych robi się to metodą gradientową (zob. 1.6).

6.3.1 Funkcje kosztu

W zależności od problemu, typowo, stosuje się następujące funkcje kosztu:

- regresja – średni błąd kwadratowy (ang. *mean squared error* - MSE), zob. 1.3;
- klasyfikacja binarna – binarna entropia krzyżowa (ang. *binary cross-entropy*);
- klasyfikacja K-klasowa – kategorialna entropia krzyżowa (ang. *categorical cross-entropy*) lub po prostu entropia krzyżowa.

6.3.2 Minimalizacja funkcji kosztu metodą gradientową z mini-paczkami

Na potrzeby uczenia dużych modeli (jakimi mogą być sieci neuronowe), a także z innych względów praktycznych (m.in. mniejsza podatność na utknięcie w minimum lokalnym), trening sieci odbywa się najczęściej nie w wersji *full-batch*, ale w wersji z tzw. *mini*-paczkami (ang. *mini-batch*). W wersji tej, w każdym obiegu pętli optymalizacji, liczona jest wartość funkcji kosztu, a następnie gradient, na podstawie *losowego* podzbioru danych treningowych (podzbiór zawiera jedynie BS danych). Ponieważ gradient szacowany jest na podstawie losowej próbki z danych, metoda ta nazywana jest często *stochastyczną metodą gradientową* (ang. stochastic gradient descent – SGD).

Znalezienie gradientu (tj. wektora pochodnych cząstkowych funkcji kosztu względem wszystkich parametrów modelu, zob. 1.4) dla skomplikowanego modelu, jakim może być sieć neuronowa nie jest trywialne i wykracza poza granice tego opracowania. Na potrzeby tego laboratorium przyjmujemy założenie, że algorytm wstecznej propagacji błędu (który to właśnie wyznacza gradient) jest zaimplementowany we *frameworku*, z którego korzystamy, i umożliwia łatwy dostęp do gradientu.

6.1 przedstawia uproszczoną wersję algorytmu SGD dla sieci neuronowej.

Algorytm 6.1 Algorytm SGD dla sieci neuronowej (wersja uproszczona).

function SGD($model, \mathbf{X}, \mathbf{y}, \eta, BS, M$)
  for $i = 1, \ldots, M$ **do**
    $\mathbf{X_{batch}}, \mathbf{y_{batch}} \leftarrow$ SAMPLE($\mathbf{X}, \mathbf{y}, BS$)
    $\hat{\mathbf{y}}_{\mathbf{batch}} \leftarrow$ FORWARDPASS($model, \mathbf{X_{batch}}$)
    $loss \leftarrow$ LOSSFCN($\hat{\mathbf{y}}_{\mathbf{batch}}, \mathbf{y_{batch}}$)
    $\nabla_\theta \leftarrow$ BACKWARDPASS($model, loss$)
    UPDATEPARAMS($model, \nabla_\theta, \eta$) ▷ $\theta \leftarrow \theta - \eta \cdot \nabla_\theta$
  return *model*

Laboratorium 7

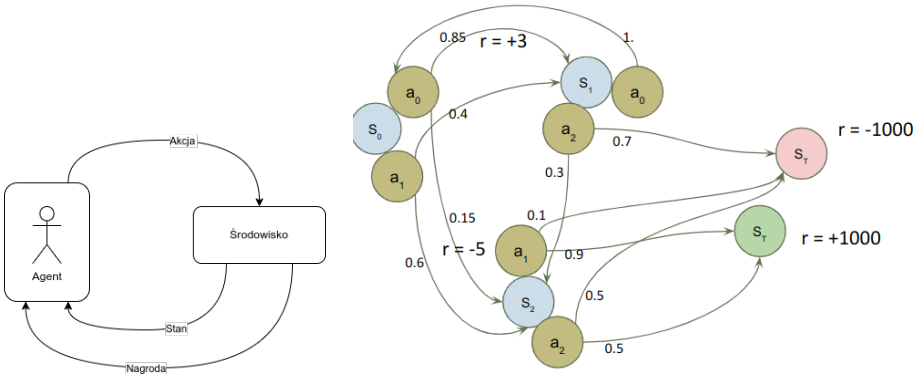
Uczenie ze wzmocnieniem

Uczenie ze wzmocnieniem to trzeci, obok uczenia nadzorowanego i nienadzorowanego, podstawowy rodzaj uczenia maszynowego. Stosowane jest ono tam, gdzie optymalizacji podlega pewien *wieloetapowy proces decyzyjny*, tj. szukana jest taka *strategia działania*, która maksymalizuje stosowny *zysk* wynikający z interakcji agenta ze środowiskiem. Przykładem takich problemów to np. nauka chodzenia, gdzie agent musi wykonać szereg odpowiednich ruchów, by osiągnąć zamierzony cel (przemieścić się), innego rodzaju systemy sterowania (np. wózkiem z odwróconym wahadłem) czy np. gry planszowe (szachy, go, shogi itd.) czy komputerowe (np. Mario, Breakout itd.), w których gracz musi podejmować odpowiednie akcje i odpowiednio reagować na zaobserwowane sytuacje, by w rezultacie osiągnąć zadowalający wynik końcowy.

Kluczowe jest tutaj *szukanie* optymalnego zachowania: w odróżnieniu od uczenia nadzorowanego, nie ma w uczeniu ze wzmocnieniem autorytatywnie danego zbioru uczącego, w którym podane są gotowe przykłady odpowiedniego działania (gdyby tak było, mielibyśmy do czynienia z *imitowaniem*, co jest niczym innym jak uczeniem zachowania w sposób nadzorowany). Przeciwnie, uczenie ze wzmocnieniem jest przydatne szczególnie tam, gdy nie jest z góry wiadomo, jakie jest najwłaściwsze działanie w zależności od zastanej sytuacji, tak by długoterminowo osiągnąć największe korzyści. Dzięki uczeniu ze wzmocnieniem możliwe jest odnalezienie innowacyjnych strategii działania w zadanych wieloetapowych procesach decyzyjnych.

Szukanie optymalnej strategii działania w uczeniu ze wzmocnieniem to rodzaj sformalizowanej *metody prób i błędów*, podczas której agent próbuje, często, szczególnie w początkowej fazie uczenia, zupełnie na ślepo, różnych działań, a następnie rewiduje ich użyteczność na podstawie doświadczenia zdobytego w interakcji ze środowiskiem. Ta ciągła interakcja między agentem, a środowiskiem, w którym się porusza, to centralny i niezbywalny rdzeń uczenia ze wzmocnieniem, nazywany *pętlą uczenia ze wzmocnieniem* (rys. 7.1a): agent, znajdując się w pewnej sytuacji w środowisku, podejmuje jakieś działanie, które w konsekwencji umiejscawia go w nowej sytuacji, z czym może się również wiązać pewna nagroda (pozytywne wzmocnienie) lub kara (wzmocnienie negatywne, osłabienie).

Pętla działania i doświadczania konsekwencji trwa tak długo, dopóki agent nie znajdzie się w jednym ze stanów końcowych (lub przerwiemy pętlę ze względu na przyjętą maksymalną liczbę kroków). Pojedynczy epizod nauki zostaje wtedy



(a) Podstawowa pętla uczenia ze wzmocnieniem

(b) Przykładowy proces decyzyjny Markowa: pokazano p-wa przejść od stanu do stanu zakładając podjęcie danej akcji. W przypadku niektórych przejść występuje niezerowa nagroda.

Rysunek 7.1: Podstawy uczenia ze wzmocnieniem.

zakończony. Takich epizodów agent musi wykonać wiele, zanim zgromadzone doświadczenie pozwoli mu na podejmowanie najwłaściwszych decyzji.

7.1 Wieloetapowy proces decyzyjny

Aby cel i metody uczenia ze wzmocnieniem ująć precyzyjnie, wieloetapowe procesy decyzyjne przedstawia się za pomocą modelu Procesu Decyzyjnego Markowa (ang. *Markov Decision Process* – MDP). Pierwszym elementem tego modelu jest $\mathcal{S}$, tj. zbiór wszystkich możliwych stanów, w jakich może się znaleźć agent, w tym wyróżniony stan inicjalny S_0, oraz stany końcowe S_{term}. Następnie mamy $\mathcal{A}$, czyli zbiór akcji, jakie agent może podejmować. $p(s'|s,a)$ to funkcja określająca prawdopodobieństwo przejścia stanu s w stan s' po wybraniu akcji a. Środowisko jest deterministyczne, gdy dla dowolnej kombinacji stanu s i akcji a, stan następny s' zachodzi z prawdopodobieństwem równym 1. $r(s,a,s')$ to funkcja nagrody: określa ona, jaką bezpośrednią nagrodę R agent otrzymuje po przejściu ze stanu s do stanu s' poprzez wykonanie akcji a. R może być dodatnie (faktyczna nagroda), ujemne (kara), a także, w wielu sytuacjach, może wynosić 0[1]. Wizualizację prostego MDP przedstawia rys. 7.1b.

Do określenia celu optymalizacji niezbędne jest zdefiniowanie dyskontowanego zwrotu G_t:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots + R = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{7.1}$$

gdzie γ to tzw. współczynnik dyskontowania, $0 \leq \gamma \leq 1$, będący parametrem

[1]W przypadku środowisk, w których R jest niezerowe sporadycznie (a czasami jedynie stan terminalny wiąże się z taką wartością) – mówimy wtedy o rzadko występującej nagrodzie – uczenie ze wzmocnieniem jest szczególnie trudne, bo sygnał uczący występuje rzadko. Przykładowo, bierki są grą w których nagrody zbiera się właściwie co akcję, ale w szachach liczącą się nagrodę zdobywa się jedynie za osiągnięcie stanu terminalnego (mat gracza, przeciwnika lub pat).

kontrolującym to, jak ważne mają być dla agenta nagrody, które w perspektywie ma do zdobycia w przyszłych krokach. Np. przyjęcie wartości $\gamma = 0$ oznacza, że oczekujemy agenta skrajnie krótkowzrocznego, który w kalkulacji G_t uwzględnia wyłącznie najbliższą bezpośrednią nagrodę nie dbając o przyszłe stany (i związane z nimi nagrody). W praktyce często spotykane wartości dla współczynnika dyskontowania to np. 0.9 czy 0.99.

Przez strategię działania $\pi(a|s)$ (ang. *policy*) rozumiana jest funkcja wyznaczająca akcję a na podstawie stanu s. Celem uczenia ze wzmocnieniem jest zatem znalezienie optymalnej strategii działania π^*, która maksymalizuje oczekiwany zwrot G_t, niezależnie od tego, w jakim stanie S_t w kroku t znajduje się agent. Nieformalnie: dobrze nauczony agent z każdej sytuacji, w jakiej się znajdzie, będzie starał się wybrnąć możliwie najlepiej.

7.2 Algorytm Q-learning

Spośród wielu algorytmów uczenia ze wzmocnieniem, na potrzeby tego laboratorium przedstawiony będzie algorytm *Q-learning*, w swojej *tabelarycznej*, podstawowej wersji. Może on być stosowany wtedy, gdy przestrzenie stanów oraz akcji danego problemu są dyskretne oraz nieduże.

7.2.1 Funkcja użyteczności Q

W algorytmie Q-learning, uczenie optymalnej strategii działania odbywa się pośrednio: algorytm stara się właściwie oszacować wartość *użyteczności* każdej możliwej akcji a w każdym możliwym stanie s. W tym celu wprowadza się funkcję użyteczności pary stan-akcja $Q(s, a)$:

$$Q(s,a) \doteq \mathop{\mathbb{E}}_{\pi}[G_t|S_t = s, A_t = a\ ] = \mathop{\mathbb{E}}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{7.2}$$

Zgodnie z powyższym wzorem, użyteczność Q można oszacować dla pewnej strategii działania π, estymując wartość oczekiwaną zwrotu, który jest do uzyskania za pomocą tej strategii. Okazuje się jednak, że jeśli *w czasie uczenia* strategia *wyboru akcji* π jest taka, że asymptotycznie pozwala ona na odwiedzenie wszystkich stanów i wykonanie wszystkich akcji w tych stanach, to oszacowane w ten sposób wartości użyteczności każdej pary stan-akcja będą optymalne. Dla takiej optymalnej użyteczności zachodzi wtedy tzw. równanie optymalności Bellmana dla funkcji użyteczności pary stan-akcja:

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a')|S_t = s, A_t = a\ \right] \tag{7.3}$$

Zakładając, że po intensywnych i długotrwałych doświadczeniach agenta, osiągnął on idealną wiedzę na temat jakości każdej możliwej pary stan-akcja (w przypadku tabelarycznego Q-learningu, wartości $Q(s, a)$ przechowywane są w tabeli o rozmiarze $|\mathcal{S}| \times |\mathcal{A}|$), optymalna strategia działania jest wtedy oczywista:

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a), \tag{7.4}$$

tj. agent wybierze zawsze taką akcję, która ma najlepszą jakość spośród dostępnych w danym stanie.

7.2.2 Uczenie metodą różnic czasowych

Ze względu na sposób, w jaki Q-learning stara się oszacować wartości funkcji użyteczności, algorytm ten zalicza się do tzw. metod *różnic czasowych* (ang. *temporal difference*). Różnica czasowa, o której tu mowa, to różnica między wartością użyteczności szacowaną na podstawie wyłącznie ostatnio zdobytego doświadczenia oraz wartością na jaką agent szacował daną użyteczność przed tym doświadczeniem. Ostatnio zdobyte doświadczenie, a więc obserwacja nagrody R i kolejnego stanu S', daje nowy szacunek użyteczności podjętej akcji A w stanie S równy $R + \gamma \max_a Q(S', a)$ (przy założeniu optymalnego działania w kolejnym stanie oraz przyjęciu współczynnika dyskontowania γ). A zatem różnica między oceną użyteczności akcji A w stanie S, którą właśnie agent doświadczył, a jego wcześniejszymi szacunkami tej użyteczności to:

$$\delta = \underbrace{R + \gamma \max_a Q(S', a)}_{\text{nowe doświadczenie}} - \underbrace{Q(S, A)}_{\text{dotychczasowa wiedza}} \tag{7.5}$$

Im większe $|\delta|$, tym większe *zaskoczenie* agenta. Właśnie zdobyte doświadczenie nie powinno zupełnie wymazywać poprzednich doświadczeń, tj. wcześniejszych estymat użyteczności, a jedynie delikatnie je korygować. Dlatego, nowa estymata użyteczności jest w metodzie różnic czasowych delikatną modyfikacją wcześniejszego oszacowania:

$$Q(S, A) \leftarrow Q(S, A) + \eta \cdot \delta \tag{7.6}$$

Hiperparametr η to współczynnik uczenia, który kontroluje wpływ ostatniego doświadczenia na modyfikację estymat. Wybór jego wartości jest kluczowy dla sukcesu uczenia: zbyt duża wartość η może sprawić, że uczenie nie będzie zbieżne (nie dojdzie do ustalenia stabilnej wartości $Q(s, a)$), natomiast zbyt mała wartość η przełoży się na powolne uczenie. Typowo, wartości tego parametru są w przedziale $[0.1, 0.01]$.

7.2.3 Ucząca strategia wyboru akcji

Jak wspomniano wyżej, strategia wyboru akcji π w trakcie szacowania wartości funkcji użyteczności powinna umożliwić asymptotyczne odwiedzenie wszystkich stanów i wykonanie w nich wszystkich akcji. W praktyce, ze względu na skończony czas uczenia, strategia taka powinna odpowiednio balansować między *eksploracją*, mającą na celu lepsze oszacowanie użyteczności danej akcji A w danym stanie S, a *eksploatacją*, tj. wyborem akcji maksymalizującej użyteczność w danym stanie po to, by zawęzić przeszukiwanie kolejnych stanów i akcji do najbardziej prosperujących. Wśród wielu mniej lub bardziej skomplikowanych strategii, które starają się znaleźć kompromis między eksploracją i eksploatacją, bardzo prostą, a dość efektywną i często wykorzystywaną strategią, jest strategia ϵ-zachłanna, przedstawiona jako Alg. 7.1

W strategii ϵ-zachłannej, parametr ϵ określa prawdopodobieństwo eksploracji. Jeśli jest stały, parametr ten przyjmuje wartości rzędu $1e^{-2} - 1e^{-5}$. W wielu aplikacjach korzystne jest rozpoczęcie nauki z silnym naciskiem na eksplorację, a następnie stopniowe przenoszenie go na eksploatację. W strategii ϵ-zachłannej takie zachowanie można osiągnąć stosując malejące ϵ, np. odejmując pewną małą wartość $\epsilon_{\text{decrement}}$, aż do osiągnięcia pewnego minimum eksploracji $\epsilon_{\min}$.

Algorytm 7.1 Strategia wyboru akcji ϵ-zachłanna.

function EpsGreedy(S – obecny stan, $\mathbf{Q}$ – tablica wartościQ, ϵ)
  $r \leftarrow \text{Uniform}(0, 1)$
  if $r > \epsilon$ **then**
    $A \leftarrow \text{argmax}_a \mathbf{Q}(S, a)$ ▷ *zachłanny wybór akcji (eksploatacja)*
  else
    $A \leftarrow \text{Uniform}(\mathcal{A})$ ▷ *losowa akcja (eksploracja)*
  return A

7.2.4 Algorytm

Pełny algorytm Q-learning dla problemu epizodycznego to uwzględnienie pętli uczenia ze wzmocnieniem, uczenia metodą różnic czasowych estymat jakości Q i strategii wyboru akcji π_{expl} w czasie uczenia (tu ϵ-zachłannej). Prezentuje go Alg. 7.2.

Algorytm 7.2 Podstawowa wersja algorytmu Q-learning.

function QLearning($\eta, \gamma, \pi_{expl}, q_{init}$)
  $\mathbf{Q} \leftarrow q_{init}$ ▷ *Inicjalizacja tabeli (najczęściej zerami, ale można też bardziej optymistycznie).*
  for $Epizod = 1, \ldots, M$ **do**
    $S_0 \leftarrow$ ResetEnv()
    $t \leftarrow 0$
    while $S_t \notin S_{Term}$ **and** $t < MaxSteps$ **do**
      $A \leftarrow$ ChooseAction(π_{expl}, S_t)
      (opcjonalnie) UpdatePolicy(π_{expl}) ▷ *Np. zredukuj ϵ*
      $S_{t+1}, R_t \leftarrow$ EnvStep(A)
      UpdateQ($\eta, \gamma, \mathbf{Q}, S_t, A, R_t, S_{t+1}$) ▷ *Wg wzoru 7.6*
      $t \leftarrow t + 1$
  return $\mathbf{Q}$