

# Uczenie Maszynowe - opracowanie 2021/2022

<b>Uczenie ze wzmacnieniem</b>	<b>3</b>
Wstęp	3
Problem wielorękich bandytów	4
Definicja problemu	4
Metody oparte o wartościowanie akcji	5
Metody oparte o górną granicę ufności	6
Metody bandytów gradientowych	7
Inne warianty problemu	8
Skończone procesy decyzyjne Markowa	9
Definicja problemu	9
Rodzaje zadań	10
Polityka i wartościowania	11
Równania Bellmana	11
Ograniczenia praktyczne	13
Programowanie dynamiczne	14
Ewaluacja polityki	14
Poprawa polityki	15
Poprawa wartościowania	16
Inne warianty ulepszania polityki	17
Metody Monte Carlo	18
Definicja problemu	18
Monte Carlo prediction	19
Monte Carlo state-action evaluation	20
Sterowanie w podejściu Monte Carlo	20
Monte Carlo on-policy	22
Monte Carlo off-policy	23
Temporal Difference learning	25
Definicja problemu	25
Predykcja TD	27
SARSA	27
Q-learning	28
Expected SARSA	28
Maximization bias, Double Q-learning	29
n-step bootstrapping	30
Definicja problemu	30

n-step prediction	30
n-step on-policy control	31
n-step off-policy control	31
Planowanie, działanie i uczenie się	32
Definicja problemu	32
Q-planning	33
Dyna-Q	34
Dyna-Q+	35
Warianty aktualizacji	35
Próbkowanie trajektorii	36
Inne warianty planowania	36
Metody przybliżające wartościowanie stanów	38
Definicja problemu	38
Metody gradientowe i semi-gradientowe	39
Metody liniowe i nieliniowe	40
Sterowanie on-policy oparte o aproksymację - epizodyczne	41
Sterowanie on-policy oparte o aproksymację - ciągłe	41
Sterowanie off-policy oparte o aproksymację	43
Metody oparte o gradient polityki	44
Definicja problemu	44
Gradient polityki	44
REINFORCE	45
Metody z punktem odniesienia	46
Metody aktor-krytyk	47
<b>Metody interpretowalne</b>	<b>48</b>
Proste metody regresji	48
Regresja parametryczna i nieparametryczna	48
Zalety i wady ridge regression i lasso regression	49
Regresja liniowa i logistyczna jako narzędzia interpretowalne	50
ElasticNet regression	51
Kriging	52
Drzewa decyzyjne i lasy losowe	53
Drzewo decyzyjne, algorytm konstrukcji	53
Funkcja kosztu	54
Kryteria stopu	55
Zalety i wady drzew	55
Drzewa decyzyjne jako interpretatory	56
Lasy losowe	56
SVM	58
SVM - klasyfikacja	58
SVM - regresja	59
Zalety i wady SVMów	60
SVM a interpretowalność	60

Metody kompresji sieci neuronowych	61
Metody oparte o rozkład macierzy	61
Metody wycinania wag	62
Metody kwantyzacji sieci	65
Layers pruning, saturation	66
Destylacja wiedzy	67
Otwieranie czarnej skrzynki	68
Taksonomia problematyki otwierania czarnej skrzynki	68
Metody outcome explanation dla sieci konwolucyjnych	69
Metody inspekcji black boxu "od środka"	70
Metody globalne	70
LIME	71
SHAP	72
Wyjaśnienia kontrafaktualne	73
Przykłady adwersarialne i ataki na systemy ML	75
<b>Metody bayesowskie</b>	<b>77</b>
Podstawy statystyki bayesowskiej	77
Definicje i prawa prawdopodobieństwa	77
Twierdzenie Bayesa	78
Prawdopodobieństwo częstotliwościowe vs bayesowskie	79
Wielowymiarowy rozkład normalny	80
Bayesowska aproksymacja funkcji	81
Klasyczne dopasowanie krzywej	81
Dopasowanie krzywej maksymalną wiarygodnością	81
Bayesowskie dopasowanie krzywej - MAP	83
Bayesowska regresja liniowa	85
Bayesowska regresja liniowa online	87
Naiwny klasyfikator Bayesa	88
Problem predykci	88
Naiwny klasyfikator Bayesa	89
Kategoryczny naiwny klasyfikator Bayesa	90
Gaussowski naiwny klasyfikator Bayesa	90
Naiwny klasyfikator Bayesa - zalety, wady i ulepszenia	90
Procesy Gaussowskie	92
Procesy stochastyczne	92
Procesy Gaussowskie	92
Regresja procesem Gaussowskim	93
Kernele	96
Nauka parametrów kerneli	97
Klasyfikacja procesem Gaussowskim	98
Zalety i wady procesów Gaussowskich	101

# Uczenie ze wzmacnieniem

## Wstęp

**Uczenie ze wzmacnieniem (reinforcement learning):**

- uczenie nastawione na osiąganie celów i podejmowanie decyzji
- bezpośrednia interakcja ze środowiskiem i operowania na feedbacku, bez zewnętrznego nadzoru
- kompromis między eksploracją środowiska a eksploatacją zebranej wiedzy

**Agent:**

- postrzega w jakimś zakresie stan środowiska
- wykonuje akcje wpływające na środowisko i zmieniające stan agenta
- ma cele związane z osiąganiem konkretnych stanów

**Środowisko (environment):**

- rzeczywistość poza agentem
- może być znane lub nieznane, skończone lub nieskończone, stałe lub zmienne
- opcjonalnie może być znany **model środowiska**

**Polityka (policy):**

- definiuje sposób, w jaki sposób zachowuje się agent
- typowo mapuje postrzegane przez agenta stany na akcje do wykonania w ramach określonego stanu

**Sygnal nagrody (reward signal):**

- zwrotna informacja środowiska o pożądanym (lub nie) natychmiastowym skutku podjętej akcji
- związana z podjęciem danej akcji w danym stanie

**Funkcja wartościująca (value function):**

- informacja o długofalowych konsekwencjach znalezienia się w danej sytuacji
- (zazwyczaj) wartość oczekiwana sumy wszystkich przyszłych nagród

# Problem wielorękich bandytów

## Definicja problemu

### K-ręki bandyta (k-armed bandit):

- problem, w którym w każdym kroku czasowym należy wybrać jedną z  $k$  akcji
- każdy wybór skutkuje przyznaniem losowej nagrody, wybranej ze stacjonarnego rozkładu prawdopodobieństwa dla danej akcji
- cel: maksymalizacja wartości oczekiwanej nagród dla zadanej liczby kroków

Problem jest nietrywialny, bo nagrody są losowe - nie wystarczy wybrać akcji tylko raz, żeby stwierdzić, jak dobrze daje nagrody, tylko trzeba jeszcze wybrać ją wielokrotnie, żeby mieć rozsądnie dobrą estymatę nagród z danej akcji

### Funkcja wartościująca akcje (action value):

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a]$$

Po prostu wartość oczekiwana z nagród dla danej akcji.

Chcemy znaleźć **optimalną politykę (optimal policy)**, czyli taki sposób wyboru akcji, żeby dostać maksymalne nagrody. Gdybyśmy znali funkcję action value, to byłoby trywialnie - bierzemy funkcję o największej wartości oczekiwanej nagród. W praktyce znamy tylko jej estymatę  $Q_t(a)$ , wyznaczoną z naszej eksploracji środowiska przez ostatnie  $t-1$  akcji.

### Exploration-exploitation tradeoff w tym problemie jest oczywisty:

- exploitation: **zachłanny** wybór akcji o maksymalnym  $Q_t$ ; daje największą nagrodę krótkoterminowo
- exploration: wybór innej akcji; daje to lepszą estymatę ich nagród, więc potencjał na większą nagrodę długoterminowo

### Stacjonarność:

- problem stacjonarny - rozkłady dla poszczególnych akcji są niezmienne, np. faktyczne automaty do gry
- problem niestacjonarny - rozkłady dla poszczególnych akcji są zmienne w czasie, np. system rekommendacyjny piosenek (akcje - rekommendacje poszczególnych gatunków muzyki), bo gust użytkownika zmienia się z czasem

## Metody oparte o wartościowanie akcji

**Metody wartościowania akcji (action-value methods)** - algorytmy, w których szacujemy wartości akcji, a nasza polityka wynika bezpośrednio z tych estymat.

### Estymacja średnią z próbek (sample-average estimation):

- wzór:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

- wykonujemy akcje i po prostu bierzemy średnią z otrzymanych nagród
- zgodnie z prawem wielkich liczb gdy liczba prób dąży do nieskończoności, taka estymata dąży do prawdziwej funkcji action value
- można uczyć iteracyjnie, zgodnie ze wzorem (wyprowadzenie pominięte):

$$Q_{n+1} = Q_n + \frac{1}{n} [R_n - Q_n]$$

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} [\text{Target} - \text{OldEstimate}]$$

n - liczba razy, jaką wykonaliśmy daną akcję

$Q_n$  - estymata wartości akcji po wykonaniu jej n razy

$R_n$  - nagroda otrzymana za podjęcie danej akcji n-ty raz

$[R_n - Q_n]$  - błąd estymaty, jak bardzo nasza estymata różniła się od faktycznie otrzymanej nagrody

- powyższe jest utrzymywane dla każdej akcji osobno - potrzebujemy całego wektora wartości Q, takiej długości, ile mamy akcji
- w powyższej metodzie wielkość kroku uczącego ( $1/n$ ) maleje w kolejnych krokach, czyli nauka "wyżarza się", gwarantuje to też zbieżność (do prawdziwego wartościowania)

### Średnia wykładniczo ważona aktualnością (exponential recency-weighted average):

- w kolejnych krokach starsze przykłady wykładniczo tracą ważność

$$Q_{n+1} = (1 - \alpha)^n Q_1 + \sum_{i=1}^n \alpha (1 - \alpha)^{n-i} R_i$$

- $\alpha$  - stały krok uczący z zakresu  $(0, 1]$
- im większa  $\alpha$ , tym bardziej zwracamy uwagę tylko na ostatnie nagrody (tym mocniej zapominamy stare)
- przydatne przy problemach niestacjonarnych (zmiennych z czasem), bo stare, nieaktualne dla nowej sytuacji przykłady ważą bardzo mało
- nie jest zbieżne dla stałego  $\alpha$ , zawsze będą pewne fluktuacje - w problemach niestacjonarnych akurat pożądana cecha, bo optymalna polityka zmienia się z czasem

### Sposoby wyboru akcji (action selection rules):

- zachłanna (greedy):
  - bierzemy akcję o największej estymowanej nagrodzie
  - 100% exploitation, nie działa za dobrze w praktyce podczas nauki
  - używana na koniec, po wytrenowaniu algorytmu - chcemy wtedy zawsze używać tego, czego się nauczyliśmy
- $\epsilon$ -zachłanna (epsilon-greedy):
  - jak zachłanna, ale z małym prawdopodobieństwem  $\epsilon$  podejmujemy losową akcję
  - podczas nauki działa lepiej, bo wymusza eksplorację
  - $\epsilon$  to hiperparametr, który trzeba dostosować do problemu

Początkowe wartości wpływają na końcowe estymaty (powodują **bias**), bo jak trafimy coś bardzo dobrego na początku, to będziemy chcieli to ciągle wybierać. Intensywna eksploracja na początku pozwala "zbadać grunt" w każdej akcji, co często daje dobre wyniki. Można to zrobić **optymistycznymi początkowymi estymatami**, czyli ustawieniem na start dużej oczekiwanej nagrody dla każdej akcji. Spowoduje to wybór każdej akcji przynajmniej raz, co prawda po tym od razu spadnie, ale zawsze wymusimy pewną ilość eksploracji.

## Metody oparte o górną granicę ufności

**Upper Confidence Bound (UCB)** - górne ograniczenie na estymatę wartości danej akcji. Kiedy je dobrze oszacujemy, to możemy założyć, że estymata jest jakaś (jakkolwiek byśmy jej nie oszacowali), ale na pewno nie większa niż UCB.

### Selekcja UCB:

- wybieramy akcję o najwyższym UCB, czyli o potencjalnej największej nagrodzie
$$Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$$
$$a_t^{UCB} = \arg \max_{a \in \mathcal{A}} \hat{Q}_t(a) + \hat{U}_t(a)$$
- daje miarę niepewności co do estymaty - mało wybierane akcje mają pewnie mało dokładną estymatę, ale na pewno mają dużą niepewność, więc wysoką wartość UCB
- kiedy wybieramy daną akcję, to od razu estymata staje się bardziej dokładna, a niepewność maleje
- od razu balansuje poniekąd exploration-exploitation tradeoff:
  - akcje wybrane mało razy mają wysoką wartość (przez dużą niepewność), więc jest zachęta do ich wybierania
  - akcje wybrane dużo razy, ale z niską wartością mają niską estymatę, małą niepewność, więc suma jest mała i raczej ich nie wybierzemy
  - akcje wybrane dużo razy i z wysoką wartością mają wysoką estymatę, małą niepewność, ale suma pewnie i tak jest dość duża i spora szansa, że je wybierzemy

- górną granicę trzeba oszacować, robimy to z nierównością Hoeffdinga, mamy:

$$U_t(a) = \sqrt{\frac{-\log(p)}{2N_t(a)}}$$

$p$  - prawdopodobieństwo, że UCB okaże się nieprawdziwe, narzucone z góry przez nas

$N_t(a)$  - liczba razy, którą do tej pory (przed krokiem  $t$ ) wykonaliśmy akcję  $a$

- w praktyce  $p$  to bardzo mała liczba, ale malejąca z czasem, bo chcemy być coraz dokładniejsi, np.  $p = t^{-4}$

### Algorytm UCB1:

- wybiera akcje zgodnie z polityką:

$$A_t \doteq \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\ln(t)}{N_t(a)}} \right]$$

$t$  - aktualny krok, liczba łącznie wykonanych do tej pory akcji

$c$  - hiperparametr, waga niepewności, uogólnienie  $p$  z oszacowania niepewności

- niepewność maleje przy wybieraniu akcji  $a$ , bo wtedy  $N_t(a)$  rośnie
- zaleta: bardzo dobre wyniki w praktyce
- problemy:
  - dla wielu bardziej złożonych zadań ciężko oszacować niepewność
  - ciężko dostosować dla zadań niestacjonarnych
  - słabo się skaluje dla problemów z dużą liczbą możliwych stanów

## Metody bandytów gradientowych

Poprzednie metody zakładają, że najpierw estymujemy wartości akcji, wykorzystując wprost wartości otrzymywanych nagród - uczymy się funkcji wartościującej. W oparciu o nią wybieramy akcję zgodnie z wybraną zasadą (np.  $\epsilon$ -greedy).

Alternatywnie możemy zadziałać bezpośrednio i uczyć się preferencji poszczególnych akcji - im większa preferencja, tym większe prawdopodobieństwo wyboru danej akcji. Nie wykorzystujemy tutaj bezpośrednio nagrody, tylko po prostu porównujemy relatywne preferencje dla poszczególnych akcji.

Polityka to w tym wypadku wektor preferencji (prawdopodobieństw) poszczególnych akcji w czasie  $t$  po przekształceniu softmax (żeby był rozkład prawd.). Na początku (w momencie  $t=0$ ) wszystkie preferencje są równe. Akcja do podjęcia jest losowana z rozkładu.

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^k e^{H_t(b)}} \doteq \pi_t(a)$$

$H_t(a)$  - preferencja akcji  $a$  w kroku  $t$

Uczenie wykorzystuje **stochastyczne wspinanie się wzduż gradientu (stochastic gradient ascent, SGA)**. W praktyce negujemy wartości i robimy zwyczajne SGD, bo wszystkie narzędzia są do minimalizacji.

### Algorytm gradientowy:

- w kroku t wybieramy akcję  $A_t$  i otrzymujemy nagrodę  $R_t$
- aktualizacja preferencji dla wybranej akcji:

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t))$$

$\alpha$  - krok uczący

$\bar{R}^{\text{(bar)}}_t$  - średnia nagród do kroku t (ale nie wliczając tego kroku!), estymowana średnią albo eksponencjalnie ważoną średnią

$(\bar{R}_t - \bar{R}^{\text{(bar)}}_t)$  - błąd oszacowania, jeżeli wychodzi, że  $R_t$  jest większa od oczekiwanej (dotychczasowej średniej), to preferencja rośnie

$\pi_t(A_t)$  - prawdopodobieństwo wyboru akcji  $A_t$  zgodnie z aktualną polityką

- jednocześnie aktualizujemy pozostałe, nie wybrane akcje ( $a \neq A_t$ ):

$$H_{t+1}(a) \doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a)$$

- kierunek zmian innych akcji jest przeciwny do zmiany  $A_t$

## Inne warianty problemu

Powysze rozwiązania działają dla problemów **stacjonarnych** albo **niestacjonarnych z bardzo powolnym dryfem** - można je praktycznie przybliżać rozkładem stacjonarnym. Mamy też problem niezmienny w sensie naszych wyborów - cokolwiek zrobimy, nie zmienia to rozkładów akcji.

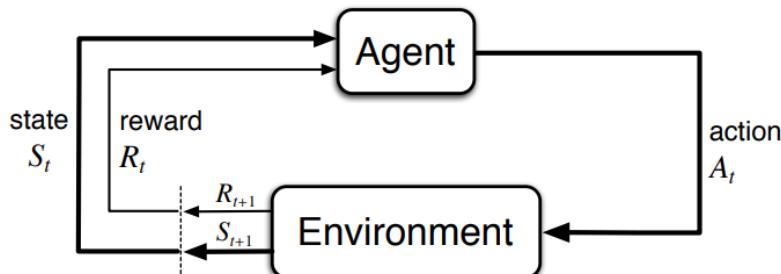
Gdy zmiany są bardziej dynamiczne, to jest ciężej. Są tu 3 warianty:

1. Nic nie wskazuje na zmianę - zmiany są częścią rozkładu
2. Coś sugeruje zmianę stanu - mamy kontekst, a więc problem bandytów kontekstowych
3. Zmiana jest skutkiem wcześniejszej wybranej akcji - czyli możemy zmieniać stan przez wybór akcji, czyli mamy pełny problem uczenia ze wzmacnieniem

# Skończone procesy decyzyjne Markowa

## Definicja problemu

Proces decyzyjny Markowa:



- Markov Decision Process (MDP)
- zakładamy, że środowisko to zestaw stanów  $S$  i akcji  $A$ ; zwykle dla uproszczenia zakłada się, że zestaw akcji jest taki sam dla wszystkich stanów
- agent w kroku  $t$ :
  - znajduje się w stanie  $S_t$
  - podejmuje akcję  $A_t$  (zgodnie z jakąś polityką)
  - przechodzi do stanu  $S_{t+1}$  i otrzymuje za to nagrodę  $R_{t+1}$

Skończony proces decyzyjny Markowa:

- Finite MDP
- zbiory stanów  $S$ , akcji  $A$  i nagród  $R$  są skończone
- $S_t$  i  $R_t$  mają dobrze zdefiniowane dyskretne rozkłady prawdopodobieństwa, zależne tylko od poprzedniego stanu i akcji - **własność Markowa**
- można jednoznacznie obliczyć prawdopodobieństwo wartości  $(s', r)$  w czasie  $t$ , jeżeli tylko znamy poprzedni stan i podjętą w nim akcję:
$$p(s', r | s, a) \doteq \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$$
- powyższa funkcja w 100% wystarcza do zdefiniowania zachowania środowiska
- jak zwykle chcemy maksymalizować otrzymywane nagrody

## Rodzaje zadań

### Zadania epizodyczne:

- dzielą się na epizody / gry / wykonania, gdzie w każdym mamy wykonujemy pewną ilość kroków  $T$  (zwykle różną dla różnych epizodów)
- krok  $T$  to **krok terminalny**, po którym "zerujemy" problem - wracamy do rozkładów początkowych i zaczynamy nowy epizod
- suma nagród to **zwrot (return)**:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

- przykłady: przechodzenie labiryntu (epizod = przejście), granie w gry (epizod = rozgrywka)

### Zadania ciągłe:

- trwają teoretycznie w nieskończoność, agent zaczyna pracę
- można oznaczyć  $T = \infty$
- 2 problemy:
  - proste sumowanie zwrotu dawałoby nieskończoność
  - nagrody w przyszłości są niepewne - im dalej w przyszłość, tym bardziej
- rozwiązanie - **zniżki (discount)**:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$G_t \doteq R_{t+1} + \gamma G_{t+1}$$

$\gamma$  - **współczynnik zniżki (discount rate)**, z zakresu  $[0, 1]$

- waga przyszłych nagród spada eksponencjalnie, przy czym im większa  $\gamma$ , tym większa waga przyszłych nagród (tym bardziej optymistyczni jesteśmy)

Można **zunifikować** powyższe przypadki:

- jeżeli  $\gamma=1$ , to mamy oczekiwany zwrot będący po prostu sumą zwrotów, czyli jak w przypadku epizodycznym
- po stanie terminalnym można dodać "stan pułapkę" z zerową nagrodą, który prowadzi tylko do samego siebie, co daje możliwość nieskończonego działania

## Polityka i wartościowania

**Polityka (policy)** - prawdopodobieństwo wyboru poszczególnych akcji w poszczególnych stanach. Oznaczana jako  $\pi$ , gdzie  $\pi(a|s)$  oznacza prawd. wybór akcji  $A_t=a$  w stanie  $S_t=s$ .

**Funkcja wartościująca stany (state-value function)** - funkcja  $v_\pi$  obliczająca oczekiwany zwrot za bycie w danym stanie, przy założeniu pewnej polityki  $\pi$ . Wartość tej funkcji dla stanu to **wartość stanu (state value)**, czyli  $v_\pi(s)$

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$$

**Funkcja wartościująca akcje (action-value function)** - funkcja  $q_\pi$  obliczająca oczekiwany zwrot za wybór danej akcji  $a$ , będąc w stanie  $s$ , przy założeniu pewnej polityki  $\pi$ . Wartość tej funkcji dla pary  $(s, a)$  to **wartość akcji (action value)**, czyli  $q_\pi(s, a)$ .

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

## Równania Bellmana

**Równanie Bellmana dla wartości stanów  $v_\pi$  (Bellman equation)** opisuje relację między wartością aktualnego stanu a wartościami kolejnych stanów.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \left[ r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \right] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) \left[ r + \gamma v_\pi(s') \right], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

Wzór sprowadza się do sumy ważonej po wszystkich możliwych wariantach przyszłości (rozważamy ostatnią linię).

Liczymy wartość oczekiwanej polityki, czyli bierzemy prawd. każdej akcji dla naszego stanu  $\pi(a|s)$  i mnożymy przez oczekiwany zwrot z wyboru tej akcji.

Ten oczekiwany zwrot liczymy, sumując nagrodę za przejście do każdego możliwego stanu, ważąc ją prawd. wykonania tego przejścia  $p(s', r | s, a)$ . Oczywiście żeby policzyć nagrodę trzeba znowu rekurencyjnie użyć równania Bellmana, żeby policzyć wartość stanu, do którego przechodzimy (mnożoną przez discount rate).

W praktyce mówi to tyle, że wartości stanów są powiązane ze sobą w taki sposób. Można to ułożyć w układ  $|S|$  równań ( $S$  - zbiór stanów).

Polityki mogą być **lepsze lub gorsze**. Polityka jest lepsza lub równa innej polityce, jeżeli jej oczekiwany zwrot dla każdego stanu jest większy lub równy od zwrotu dla drugiej polityki. **Dla finite MDP zawsze istnieje przynajmniej jedna optymalna polityka.** Może być ich więcej, ale jaką konkretnie wybierzemy, zwykle nie ma znaczenia. Optymalną politykę oznacza się  $\pi^*$ .

Wszystkie polityki optymalne współdzielą tę samą **optymalną funkcję wartościującą stany i optymalną funkcję wartościującą akcje**.

Opt. funkcja state-value opisuje najlepszą wartość, jaką można w ogóle osiągnąć z danego stanu, a opt. funkcja action-value najlepszą nagrodę, którą można uzyskać, jeżeli w danym stanie podejmiemy daną akcję. Mając tą parę, mamy zawsze precyzyjną informację, jaka akcja jest najlepsza w danym stanie, a więc co trzeba zrobić, żeby dostać największą możliwą nagrodę (w ogóle, nie tylko w tym kroku).

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s)$$

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a)$$

Są powiązane, bo żeby policzyć, która akcja jest najlepsza, musimy uwzględnić:

- aktualną nagrodę
- przyszłe nagrody, a one są związane z tym, jak dobre są stany, do których możemy przejść

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

**Równanie Bellmana dla wartości stanów  $v^*$  (Bellman optimality equation)** opisuje jak wartościować stany przy założeniu polityki optymalnej. Pozwala obliczyć, jaka jest maksymalna możliwa oczekiwana nagroda w danym stanie.

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi^*}(s, a) \\ &= \max_a \mathbb{E}_{\pi^*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_*(s')]. \end{aligned}$$

Bierzemy po prostu największy oczekiwany zwrot po wszystkich akcjach. Wagą akcji jest prawdopodobieństwo jej wyboru w danym stanie, a wartością oczekiwany zwrot, przy czym on też odwołuje się do optymalnych wartości stanów, do których przechodzimy daną akcją.

Można z tego też wyprowadzić wersję dla pary  $(s, a)$  - po prostu nie bierzemy max po akcjach, tylko

$$\begin{aligned} q_*(s, a) &= \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned}$$

Dla skończonych MDP równanie dla  $v_*$  ma unikalne rozwiązanie. W praktyce jest to układ  $|S|$  równań z  $|S|$  niewiadomymi, przy czym są to równania nieliniowe. Wystarczy użyć dowolnego solwera dla układów równań nieliniowych i dostaniemy optymalne wartościowania dla każdego stanu.

Jak już to mamy, to znalezienie optymalnej polityki jest łatwe. Dla każdego stanu znamy wtedy 1 lub więcej akcji o maksymalnym oczekiwany zwrocie, czyli  $v_*(s)$ . Dowolna polityka przypisująca niezerowe prawdopodobieństwo tylko tym akcjom jest optymalna. Innymi słowy, **polityka zachłanna względem optymalnej funkcji wartościowej  $v_*$  jest optymalna**.

Czyli dla skończonych MDP wystarczy rozwiązać układ równań nieliniowych z równania Bellmana dla  $v_*$  i dostajemy optymalną politykę.

## Ograniczenia praktyczne

Żeby w ogóle dało się ułożyć ten układ równań, to model środowiska, czyli zbiory stanów, akcji i nagród, musi być znany i to dokładnie. **Model środowiska często nie jest znany lub jest przybliżony (a więc błędny)**.

Rozwiązywanie układów równań nieliniowych nie jest wcale tanie obliczeniowo ani pamięciowo. W praktyce taki układ to odpowiednik przeszukiwania dokładnego (exhaustive search) wszystkich możliwych stanów i przejść. **Dla dużych zbiorów stanów lub zbiorów akcji obliczenie tego może być niemożliwe**.

Dodatkowo zakładamy tu właściwość Markowa, a często następny stan zależy nie tylko od bezpośrednio poprzedniego, ale też od jeszcze wcześniejszych.

Jak da się policzyć, to idealnie, ale w praktyce zwykle się nie da. Mamy wtedy 2 możliwości:

- aproksymować politykę optymalną
- wybrać podzbiór stanów i akcji, które rzeczywiście spotykamy w praktyce i dla nich liczyć jak najlepsze rozwiązanie

# Programowanie dynamiczne

**Programowanie dynamiczne (dynamic programming, DP)** - sposób efektywnego rozwiązywania problemów rekurencyjnych dających się rozbić na identyczne pod-problemy. W szczególności są użyteczne, gdy optymalne rozwiązania podproblemów dają optymalne rozwiązanie całości problemu (własność optymalnej podstruktury).

## Ewaluacja polityki

**Ewaluacja polityki (policy evaluation, prediction problem)** - zadanie obliczania funkcji wartościującej stany (state-value function) dla pewnej polityki  $\pi$ . Zgodnie z wcześniej podanymi równaniami mamy:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')] \end{aligned}$$

Mamy gwarancję, że  $v_\pi$  istnieje i jest unikalna, o ile  $\gamma < 1$  lub z każdego stanu kiedyś osiągniemy stan terminalny. Jeżeli znamy dokładnie model środowiska, to sprowadza się to do rozwiązania układu  $|S|$  równań liniowych z  $|S|$  niewiadomymi.

**Iteracyjna ewaluacja polityki (iterative policy evaluation)** polega na rozwiązaniu powyższego problemu iteracyjnym algorytmem rozwiązywania układów równań. Na początek zaczynamy z losowym  $v_0$  (dla stanów terminalnych 0); typowo od razu robimy cały wektor, dla wszystkich stanów. Potem aktualizujemy je iteracyjnie:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \end{aligned}$$

Takie rozwiązanie zbiega w nieskończoności do wartościowania  $v_\pi$ . W praktyce mamy więc rozwiązanie przybliżone, ale dość dobre, przy czym dla MDP często zbieżność jest całkiem szybka.

A po co nam właściwie dobre wartościowanie? Żeby znajdować lepsze polityki, czyli dające lepszy oczekiwany zwrot. Dla danego stanu chcemy wiedzieć, czy podążyć za obecną polityką, czy ją zmienić (a jeśli tak, to jak). Do tego trzeba umieć wartościować stany, żeby w ogóle móc stwierdzić jakość polityki.

## Poprawa polityki

**Twierdzenie o poprawie polityki (policy improvement theorem)** - jeżeli dla pary deterministycznych polityk  $\pi$  i  $\pi'$  dla dowolnego stanu zadziałanie zgodnie z nową polityką  $\pi'$  da lepszą oczekiwana nagrodę niż dla poprzedniej polityki  $\pi$ :

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

To polityka  $\pi'$  jest lepsza lub równie dobra względem  $\pi$ , czyli daje większe lub równe oczekiwane zwroty dla wszystkich stanów:

$$v_{\pi'}(s) \geq v_\pi(s)$$

Próbujemy wykonać najbliższą akcję w danym stanie  $s$  zgodnie z inną polityką, sprawdzając oczekiwany zwrot. Porównujemy to z obecnym  $v_\pi(s)$ , czyli najlepszym możliwym zwrotem przy aktualnej polityce dla tego stanu. Jeżeli wynik jest lepszy lub równy, to znaczy, że nowa polityka podejmuje równie dobre lub lepsze decyzje, czyli warto na nią przejść.

**Poprawa polityki (policy improvement)** - proces zachłanego ulepszania polityki. Za każdym razem wybieramy politykę dla danego stanu, która daje największą nagrodę według wartościowania następnego stanu  $v_\pi(s')$ . Zgodnie z twierdzeniem o poprawie polityki nowa polityka wybrana w ten sposób jest co najmniej tak dobra jak poprzednia.

$$\begin{aligned} \pi'(s) &\doteq \arg \max_a q_\pi(s, a) \\ &= \arg \max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \arg \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')] \end{aligned}$$

Potrzebujemy wartościowania stanów, bo tylko dzięki niemu jesteśmy w stanie policzyć, jak dobre są poszczególne akcje. Poprawa jest z tego, że wybrane akcje mogą być inne, niż w dotychczasowej polityce, czyli dawać większe oczekiwane zwroty.

**Iteracyjna poprawa polityki (policy iteration)** - algorytm iteracyjnego ulepszania polityki:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \dots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

1. Zaczynamy od losowej polityki  $\pi_0$
2. Policy evaluation E - sprawdzamy wartościowania stanów w aktualnej polityce
3. Policy improvement I - poprawiamy politykę, wybierając dla każdego stanu najlepszą akcję

Algorytm zbiega w chwili, gdy akcje nie uległy zmianie w ramach policy improvement. Jako że MDP ma skończoną liczbę stanów, a każda iteracja ulepsza politykę, to zbiega do polityki optymalne (z równania Bellmana).

Warto zauważyć, że krok policy evaluation sam w sobie jest kosztownym procesem iteracyjnym, wymagającym iteracyjnego rozwiązywania układu równań.

## Poprawa wartościowania

**Iteracyjna poprawa wartościowania (value iteration)** - wariant policy iteration, w którym każdy krok policy evaluation robi tylko pojedynczą iterację, czyli każdy stan jest ewaluowany dokładnie raz. Po tym od razu następuje policy improvement.

Alternatywnie można to interpretować jako połączenie kroków E oraz I - zamiast najpierw po kolei robić dla wszystkich stanów iteracyjną ewaluację, a potem improvement, robimy jedną pętlę po stanach, dla każdego robiąc jedną ewaluację i od razu ulepszenie.

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \end{aligned}$$

Nazwa bierze się stąd, że podczas iteracji ulepszamy tylko wartościowanie, a dopiero na sam koniec na podstawie precyzyjnego wartościowania (blisko optymalnego) obliczamy politykę:

Loop:

```
| Δ ← 0
| Loop for each s ∈ S:
|   v ← V(s)
|   V(s) ← maxa ∑s',r p(s',r|s,a) [r + γV(s')]
|   Δ ← max(Δ, |v - V(s)|)
until Δ < θ
```

Output a deterministic policy,  $\pi \approx \pi_*$ , such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

Jest oparte o obserwację, że często wystarczy mało iteracji policy evaluation, żeby dostać dobre oszacowanie, więc tu ograniczamy się do jednego. Co prawda ogólnie iteracji będzie więcej, ale i tak jest szybciej, bo minimalizujemy najbardziej kosztowną część algorytmu.

Na podstawie równania Bellmana gwarantuje zbieżność do optymalnej polityki. Można jeszcze robić warianty value iteration, w których robi się kilka iteracji policy evaluation przed policy improvement, co przyspiesza zbieżność.

## Inne warianty ulepszania polityki

Kiedy mamy bardzo dużą liczbę stanów, to nawet pojedyncza ewaluacja polityki może być bardzo kosztowna. Rozwiązaniem jest **asynchroniczne aktualizowanie**, czyli aktualizacja **podzbioru stanów** w każdej iteracji. W szczególności może być to 1 stan naraz.

Daje to zbieżność, jeżeli w nieskończoności każdy stan aktualizowany jest nieskończonie wiele razy. Przy czym uwaga - nie zawsze jest nam to właściwie potrzebne, bo dzięki temu, że wybieramy tylko niektóre stany, to można ten wybór przeprowadzić inteligentnie, na przykład aktualizować częściej te stany, które agent częściej spotyka. W ten sposób dla tych ważnych stanów szybciej dostaniemy dobre wartościowanie, a zatem też dobrą politykę.

Można uogólnić te wszystkie podejścia jako **uogólnioną iteracyjną poprawę polityki (Generalized Policy Iteration, GPI)**.

Zakładamy w nim, że policy evaluation i policy improvement to dwa wpływające na siebie wzajemnie procesy, przy czym granularność i szczegółowe implementacyjne są niezależne. Ewaluacja polityki stabilizuje się, gdy jest zgodna z aktualną polityką, a sama polityka jest stabilna tylko wtedy, kiedy jest zachłanna względem aktualnego wartościowania. W takim wzajemnym procesie oba ustabilizują się jednocześnie tylko wtedy, kiedy znajdziemy politykę, która jest zachłanna wobec własnej funkcji ewaluacji, czyli spełnia równanie Bellmana, czyli jest polityką optymalną. W takim podejściu E oraz I "ciągną" każde w swoją stronę, ale oba zbiegają w stronę optymalności.

Zalety z takiego podejścia to możliwość niezależnego definiowania tych elementów. Pozwala to też w zunifikowany sposób definiować wiele problemów uczenia ze wzmacnieniem, w tym takich, w których środowisko nie jest dokładnie znane.

# Metody Monte Carlo

## Definicja problemu

### Metody Monte Carlo:

- wielokrotna symulacja doświadczeń
- często zbiór rzeczywistych doświadczeń jest niewielki - doświadczenia są drogie, pozwalają za to oszacować rozkłady i stworzenie symulacji procesu Markowa
- samo stworzenie modelu środowiska jest zbyt drogie i trudne, bo wymagałoby dużo doświadczeń, dokładnego oszacowania prawdopodobieństw przejść etc.
- **podejścia bezmodelowe (model-free approaches)**, bo nie budujemy explicite pełnego modelu środowiska z pełnymi rozkładami prawdopodobieństwa wszystkich możliwych przejść, a tylko uproszczoną symulację
- potrzebujemy wiedzy domenowej, żeby stworzyć symulację, ale często jest ona dość prosta

Przykładowo, gra w karty (np. blackjack) ma bardzo proste zasady i stworzenie symulatora do gry jest bardzo szybkie, samo używanie go też. Natomiast pełny model potrzebowałby na przykład potężnych macierzy rozmiaru (liczba kart) \* (liczba kart), żeby odwzorować prawdopodobieństwo, jakie karty zostaną zagrane jedna po drugiej, a co więcej trzeba by jeszcze zagrać mnóstwo gier, żeby dostać odpowiednie oszacowanie takich prawdopodobieństw.

W uczeniu ze wzmacnieniem działa dla **zadań epizodycznych**, bo pojedyncze uruchomienie symulacji to jeden epizod. Aktualizacje estymat wartościowań stanów i polityk są aktualizowane po pełnych zakończonych epizodach.

Aktualizacje opierają się o średnie zwroty dla par stan-akcja i średnie nagrody dla każdej akcji. Jest to dość podobne do problemu wielorékiego bandyty, ale każdy stan działa tutaj jak osobny bandyta, a co więcej są one ze sobą powiązane, bo wybór akcji w danym stanie wpływa na późniejsze stany w tym samym epizodzie. W związku z tym problem jest kontekstowy i niestacjonarny z tej perspektywy. Żeby z tym sobie poradzić, dodatkowo używa się Generalized Policy Iteration (GPI), przy czym tutaj nie możemy policzyć dokładnych wartości (bo nie mamy modelu środowiska), tylko się ich uczymy ze średnich zwrotów od MDP - w kroku E wykorzystujemy symulacje Monte Carlo i zbieramy doświadczenie, żeby w kroku I je wykorzystać i ulepszyć politykę.

## Monte Carlo prediction

Dla przypomnienia, **prediction** to zadanie ewaluacji polityki, czyli znalezienia wartościowań stanów używając jej.

W metodzie **first-visit Monte Carlo (MC)** uwzględniamy tylko **pierwszą wizytę (first visit)** stanu  $s$ , czyli pierwsze bycie w tym stanie w danym epizodzie. **Metoda every-visit MC** też działa, ale ma trochę inne własności teoretyczne. Obie metody są zbieżne dla nieskończonej liczby próbek z danego stanu do prawdziwej wartości  $v_{\pi}(s)$ .

Pseudokod dla **first-visit Monte Carlo prediction (policy evaluation)**:

Input: a policy  $\pi$  to be evaluated

Initialize:

$V(s) \in \mathbb{R}$ , arbitrarily, for all  $s \in \mathcal{S}$

$Returns(s) \leftarrow$  an empty list, for all  $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following  $\pi$ :  $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :

$G \leftarrow \gamma G + R_{t+1}$

Unless  $S_t$  appears in  $S_0, S_1, \dots, S_{t-1}$ :

Append  $G$  to  $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Generujemy epizod, czyli serię stan-akcja-nagroda. Dzięki temu, że aktualizujemy się na koniec epizodu, to mamy dane aż do ostatniej nagrody za przejście do stanu terminalnego  $R_T$ . W związku z tym dokładnie wiemy, jak duży zwrot przyniosła nam każda decyzja w tym epizodzie - wystarczy prześledzić go od końca.

Po wykonaniu (wygenerowaniu) epizodu aktualizujemy wartościowanie. Idziemy od końca po elementach epizodu, utrzymując sumę nagród ze zniżkami  $\gamma$ . Kiedy spotkamy historycznie pierwszą wizytę w danym stanie (czyli nie będzie jej już więcej podczas iteracji od końca do początku), to jej średni zwrot to po prostu średnia wartość z sumy zwrotów.

Ważna obserwacja: estymaty dla poszczególnych stanów **są niezależne**. Innymi słowy nie ma bootstrapowania, czyli powiązania estymat wartości stanów ze sobą (które było w równaniach Bellmana). W związku z tym **koszt obliczeniowy i pamięciowy nie są związane z ogólną liczbą stanów** - tylko od wybranej liczby sprawdzanych stanów. Można na przykład zastosować technikę, gdzie zaczynamy epizody od interesujących nas stanów i sprawdzamy średnie zwroty tylko dla nich.

## Monte Carlo state-action evaluation

Problem z estymowaniem tylko wartości stanów jest taki, że w podejściu model-free nie wiemy, gdzie mogą nas doprowadzić te stany, bo bez modelu nie znamy przejść stan-akcja-stan. W związku z tym musimy **estymować wartościowanie akcji**, czyli par stan-akcja, żeby dostać informacje przydatne do optymalizacji polityki.

**Problem ewaluacji polityki dla wartości akcji (policy evaluation for action values)** polega na estymowaniu  $q_{\pi}(s, a)$ , czyli oczekiwanej wartości zwrotu przy wykonaniu akcji  $a$  w stanie  $s$  dla polityki  $\pi$ . Algorytm jest właściwie identyczny z tym dla wartości, tylko zliczamy wystąpienia dla par, a nie dla samych stanów.

Jest tu jeden problem - możliwości jest znacznie więcej i jest spora szansa, że wiele par  $(s, a)$  nie zostanie nigdy odwiedzonych w danej polityce. Jest to problem **utrzymywania eksploracji (maintaining exploration)**, tak, żeby bez względu na aktualną politykę dalej eksplorować.

Rozwiązanie to **eksplorujące otwarcia (exploring starts)**, w których zaczynamy nie od konkretnego stanu, tylko od pary stan-akcja (narzucamy 2 pierwsze elementy epizodu), a przy generowaniu epizodów każda para ma niezerowe prawdopodobieństwo bycia wybraną.

Gorzej, kiedy nie da się tak zrobić, na przykład symulator jest zewnętrzny i nie ma takiej opcji, albo interesująca para stan-akcja nie jest dopuszczalna jako początkowa. Można to rozwiązać przez stosowanie polityk stochastycznych, czyli takich z niezerowym prawdopodobieństwem wyboru dowolnej akcji (np.  $\epsilon$ -greedy).

## Sterowanie w podejściu Monte Carlo

Wcześniejsze metody to tylko przybliżanie pewnych elementów. Chcemy natomiast **sterowania (control)**, czyli użycia Monte Carlo do przybliżania optymalnej polityki. Używa się tutaj Generalized Policy Iteration (GPI), gdzie mamy przybliżoną politykę i przybliżoną funkcję wartościującą (w tym wypadku pary stan-akcja), które wzajemnie na siebie wpływają. Finalnie taki układ zbiega jednak do polityki optymalnej.

Mogą być użyte wprost zwykłego GPI, jeżeli poczynimy 2 założenia:

- mamy nieskończoną liczbę epizodów
- epizody korzystają z eksplorujących startów

Przy tych założeniach estymacja Monte Carlo oszacowuje  $q_{\pi}(s, a)$  dokładnie, dla dowolnej polityki  $\pi$ .

Policy improvement jest prosty i polega na stworzeniu polityki zachłannej względem aktualnej funkcji wartościującej akcje. Skoro mamy estymaty dla każdej pary stan-akcja, to wystarczy jako politykę dla każdego stanu wybrać akcję o największym oczekiwany zwrocie:

$$\pi(s) \doteq \arg \max_a q(s, a)$$

Zgodnie z twierdzeniem o poprawie polityki dostaniemy w ten sposób lepszą politykę, co będzie ogółem zbiegać do polityki optymalnej.

W praktyce trzeba jednak pozbyć się obu tych założeń, żeby móc zaimplementować efektywny algorytm. Pozbycie się założenia o nieskończonej liczbie epizodów jest proste - po prostu robimy aktualizację co jeden epizod. Jest to podobne do value iteration, nazywa się **Monte Carlo with Exploring Starts (MC ES)**. Algorytm:

- wylosuj parę startową  $(s_0, a_0)$
- wygeneruj epizod, zaczynając od  $(s_0, a_0)$
- przejdź od końca po każdym kroku epizodu
- aktualizuj pary  $(s, a)$  przy first-visit i od razu wtedy aktualizuj politykę dla tego stanu (jeżeli ta akcja jest lepsza od dotychczasowej, to ją preferuj)

Co ważne - aktualizacje bierze się bez względu na aktualną politykę. W szczególności polityka może się zmieniać podczas iterowania po epizodzie, bo już wtedy robimy jej aktualizacje. Czy to jest zbieżne? Nie wiadomo, jest to problem otwarty, ale w praktyce wszystko wskazuje na to, że jest zbieżne do polityki optymalnej.

Założenie o eksplorujących startach jest niestety nierealne, bo nie zawsze jest możliwe i użyteczne. Pary stan-akcja nie zawsze mogą być na początku i bardzo rzadko coś takiego samo z siebie nastąpi, a chcemy się uczyć z otoczenia, nie narzucać coś arbitralnie algorytmowi. Zamiast tego chcemy wbudować w agenta taką inteligencję, żeby zawsze było prawdopodobieństwo wyboru par stan-akcja.

Są tutaj 2 rodziny metod

- **on-policy** - ewaluujemy i ulepszamy politykę używaną też do podejmowania decyzji
- **off-policy** - używana jest osobna polityka niż ta używana przy generowaniu danych (epizodów) i to ona jest ewaluowana/ulepszana

## Monte Carlo on-policy

Metody on-policy używają tylko jednej polityki, więc są prostsze niż off-policy. Często mają też mniejszą wariancję podczas treningu i zbiegają szybciej. Są jednak mniej ogólne.

Metody on-policy wymagają **polityk miękkich (soft policies)**, czyli takich, że wszystkie akcje mają niezerową szansę na bycie wybraną, np.  $\epsilon$ -greedy. Polityki oparte o metodę  $\epsilon$ -greedy należą do **grupy polityk  $\epsilon$ -soft**.

Algorytm **on-policy first-visit MC control  $\epsilon$ -soft**:

Algorithm parameter: small  $\epsilon > 0$

Initialize:

$$\begin{aligned}\pi &\leftarrow \text{an arbitrary } \epsilon\text{-soft policy} \\ Q(s, a) &\in \mathbb{R} \text{ (arbitrarily), for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \\ Returns(s, a) &\leftarrow \text{empty list, for all } s \in \mathcal{S}, a \in \mathcal{A}(s)\end{aligned}$$

Repeat forever (for each episode):

Generate an episode following  $\pi$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$$G \leftarrow 0$$

Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :

$$G \leftarrow \gamma G + R_{t+1}$$

Unless the pair  $S_t, A_t$  appears in  $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$ :

Append  $G$  to  $Returns(S_t, A_t)$

$$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$$

$$A^* \leftarrow \arg\max_a Q(S_t, a) \quad (\text{with ties broken arbitrarily})$$

For all  $a \in \mathcal{A}(S_t)$ :

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon / |\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon / |\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Z ważnych elementów, dalej mamy tylko jedną politykę (on-policy). Pod koniec nie tylko ewaluujemy parę  $(s, a)$ , ale też od razu aktualizujemy politykę. W aktualizacji jest element losowości (ten wzór pod koniec) - dla akcji nieoptimalnych (innych niż  $A^*$ ) przypisujemy niewielkie prawdopodobieństwo  $\epsilon / |\mathcal{A}(S)|$ .

Wcześniej w wyborze zachłannym przypisywalibyśmy pod koniec po prostu akcję o największym oczekiwany zwrocie, ta linijka z argmax, a wszystko pozostałe byłoby zerami.

Wybór  $\epsilon$ -zachłanny jest zbieżny, co pozwala się pozbyć założenia o eksplorujących otwarciach. Zatem w końcu mamy coś, co jest Monte Carlo, działa i nadaje się do praktycznej implementacji.

## Monte Carlo off-policy

Zawsze dążymy do optymalnej polityki. Im dalej w treningu jesteśmy, tym lepiej powinniśmy znać dla niej wartościowanie. Problem z metodami on-policy jest taki, że muszą one stosować sub-optymalną politykę, z pewną losowością wybierającą potencjalnie kiepskie akcje, żeby wymusić eksplorację.

Metody off-policy unikają tego problemu, utrzymując 2 osobne polityki:

- **polityka docelowa (target policy)** - tego chcemy się nauczyć, polityka zbieżna do optymalnej
- **polityka zachowania / behawioralna (behavior policy)** - polityka używana do wyboru akcji, może być suboptimalna dla eksploracji

Zalety to większa elastyczność i siła wyrazu, ale same metody są trudniejsze do formalnego opisu, a podczas treningu wariancja jest większa i zbieżność jest wolniejsza.

Na dobry początek trzeba dokonać wartościowania  $v_{\pi}$  albo  $q_{\pi}$ . W takim problemie i polityka docelowa  $\pi$ , i polityka zachowania  $\beta$  są stałe i znane, przy czym  $\beta \neq \pi$ .

Jako że mamy epizody wygenerowane z użyciem  $\beta$ , a chcemy estymować wartości dla  $\pi$ , to wymagamy, żeby każda akcja używana w  $\pi$  była też używana (przynajmniej czasem) przez  $\beta$ , czyli żądamy implikacji:

$$\pi(a|s) > 0 \rightarrow \beta(a|s) > 0$$

Jest to założenie **pokrywania (coverage)**. Wynika też z niego, że  $\beta$  musi być stochastyczne tam, gdzie nie jest identyczne z  $\pi$ , więc typowo jest po prostu  $\epsilon$ -zachłanne (przy okazji daje to eksplorację).

Estymowanie wartości oczekiwanych z jednego rozkładu mając próbki z drugiego wymaga techniki **importance sampling**. Dla danego zwrotu mamy jego **trajektorię**, czyli ciąg akcji i stanów, który od niego doprowadził. Stosunek prawdopodobieństw wystąpienia danej trajektorii w docelowej i behawioralnej polityce to **importance-sampling ratio**.

Prawdopodobieństwo trajektorii to po prostu iloczyn prawdopodobieństw danych stanów i akcji przy danej polityce:

$$\begin{aligned} \Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T | S_t, A_{t:T-1} \sim \pi\} \\ = \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\ = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k), \end{aligned}$$

Importance-sampling ratio to iloczyn tych wyrażeń dla dwóch polityk.

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

Co ważne, we wzorze powyżej same prawdopodobieństwa przejść MDP są takie same w obu politykach, bo działają w tym samym środowisku, więc się skracają. Dzięki temu nie trzeba modelu środowiska, a sam iloczyn zależy tylko od obu polityk i sekwencji stanów i akcji.

Żeby dostać estymaty wartości stanów dla polityki docelowej trzeba zważyć zwroty otrzymane z polityki behawioralnej tym iloczynem:

$$\mathbb{E}[\rho_{t:T-1} G_t | S_t = s] = v_\pi(s)$$

Kwestia jest taka, jak dokonywać tego próbkowania (w końcu to sampling).

**Ordinary importance sampling** bierze wszystkie wystąpienia danego stanu, skaluje ich zwroty importance-sampling ratio i zwraca średnią:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}$$

**Weighted importance sampling** liczy średnią ważoną tymi współczynnikami:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}$$

W praktyce wariant prosty jest unbiased, czyli zawsze daje dobrą estymatę wartości oczekiwanej. Niestety ma potencjalnie nieograniczoną wariancję, a w praktyce dużą (chociaż dającą do zera przy wzroście liczby próbek), bo współczynnik ważący potrafi się bardzo różnić między próbками i jedna może zdominować pozostałe.

Wariant ważony ma bias wynikający z różnicy między rozkładami, szczególnie dla małej liczby próbek. Wtedy współczynniki ważjące z licznika i mianownika praktycznie się znoszą, więc jeżeli w zwrotach  $G_t$  są różnice, to będą one praktycznie bezpośrednio powodowały bias. Natomiast wszystkie wagie sumują się do 1, bo to ułamki, więc wariancja jest ograniczona i w praktyce niewielka.

W praktyce częściej korzysta się z wariantu ważonego ze względu na brak problemów z wariancją.

# Temporal Difference learning

## Definicja problemu

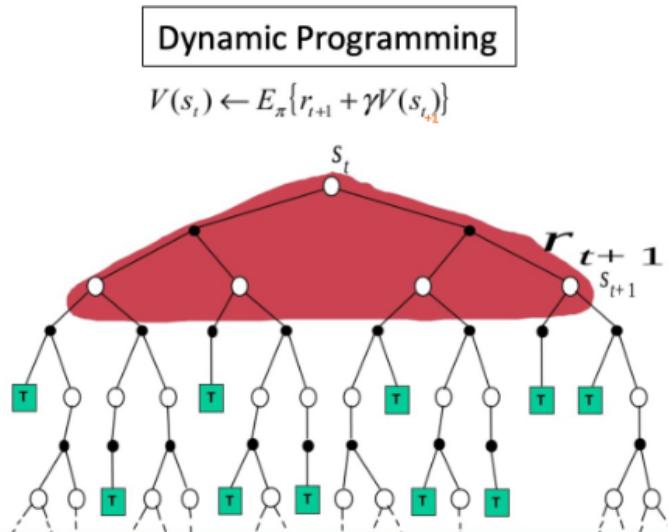
Nauka w oparciu o różnice czasowe (temporal-difference, TD):

- połączenie metod programowania dynamicznego i Monte Carlo
- model-free, podobnie do Monte Carlo
- możliwość działania w trybie ciągłym, bez czekania na koniec epizodu (bootstrapping), podobnie do DP
- nazwa bierze się z tego, że aktualizacje wartościowania opierają się o różnice między wcześniejszą a późniejszą estymatą

Różnica między DP, Monte Carlo i TD tkwi głównie w problemie predykcji (policy evaluation). Dla problemu kontroli (znajdowania optymalnej polityki) używają bardzo podobnych wariacji GPI.

Różnice i analogie dobrze widać na obrazkach.

W programowaniu dynamicznym aktualizujemy wartościowanie w każdym kroku (zadania ciągłe) za pomocą wartości oczekiwanej. Możemy to zrobić, bo robimy exhaustive search i wartość oczekiwana da się dokładnie policzyć.

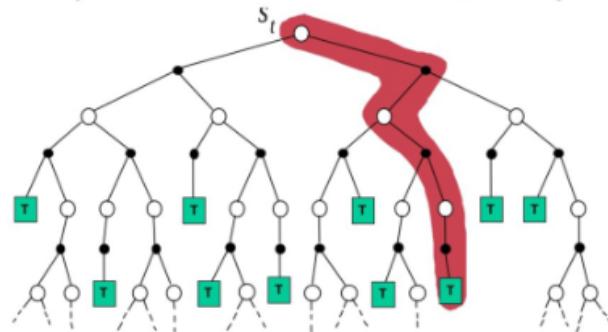


W metodach Monte Carlo aktualizujemy na koniec epizodu (zadania epizodyczne), uwzględniając trajektorię, którą przeszliśmy w ramach tego epizodu.

### Monte Carlo Learning

$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$

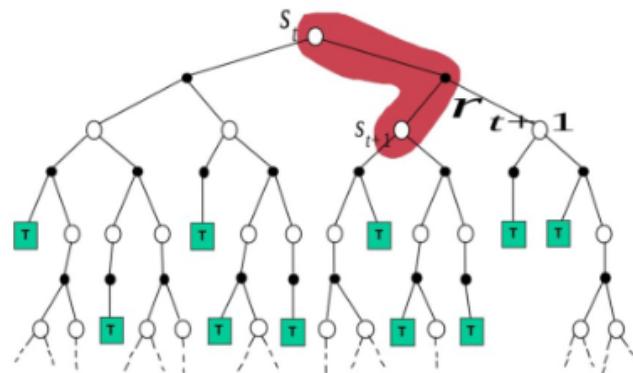
where  $R_t$  is the actual return following state  $s_t$ .



W metodach Temporal Difference aktualizujemy zgodnie z aktualną ścieżką, w oparciu o ostatnie kroki, w trakcie trwania (zadania ciągłe).

### Temporal Difference Learning

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$



#### Zalety metod TD:

- model-free - jak Monte Carlo, a przeciwnie do programowania dynamicznego
- online - uczą się w trakcie wykonywania akcji, jak programowanie dynamiczne, a przeciwnie do Monte Carlo
- dobre gwarancje zbieżności - takie same jak MC
- nie potrzeba epizodów - mogą być używane przy zadaniach ciągłych, jak DP, a przeciwnie do Monte Carlo
- dają lepsze wyniki w praktyce niż Monte Carlo, są wykonalne obliczeniowo tam gdzie DP nie jest

## Predykcja TD

**Predykcja TD** jest dość podobna do metod Monte Carlo, bo w TD też używamy doświadczenia zdobytego przez podążanie za pewną polityką  $\pi$  do ulepszenia oszacowania  $V$  wartości  $v_{\pi}$  dla tych stanów, które wystąpiły w tym doświadczeniu.

Najłatwiej porównać to z Monte Carlo. W every-visit MC aktualizacja to:

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$

MC czeka do końca epizodu, żeby zaktualizować dany stan, bo dopiero wtedy zna faktyczny zwrot  $G_t$ .

W metodach TD zamiast tego od razu, już w następnym kroku czasowym aktualizujemy estymatę wartości poprzedniego stanu:

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Czyli zamiast dokładnego zwrotu  $G_t$  bierzemy **przybliżenie zwrotu**, czyli sumę faktycznie otrzymanej nagrody i zdyskontowane oszacowanie wartości nowego stanu, do którego przeszliśmy:  $R_{t+1} + \gamma * V(S_{t+1})$ . Różnica między tym a poprzednim oszacowaniem  $V(S_t)$  to **błąd różnicowy czasowej (TD error)**.

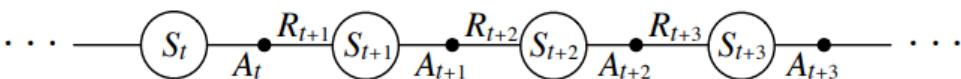
Powyzsze to **TD(0) / one-step TD**, bo wykonuje aktualizację już po jednym kroku. Istnieją warianty n-krokowe, które aktualizują dopiero po odbyciu kilku kroków. Metody TD tworzą więc płynne spektrum metod podobnych do Monte Carlo, używając samplingu. Używają przy teź **bootstrappingu**, tak jak DP, bo odwołują się do aktualnej estymaty wartości innego stanu.

## SARSA

**SARSA** to algorytm **on-policy temporal difference control**. Mamy zatem jedną politykę, której się uczymy i podejmujemy z jej pomocą decyzje, a uczymy ją algorytmami TD. Jako że dalej korzystamy z GPI, to TD jest używane w wartościowaniu (predykcji).

Uczymy się wartościowania par stan-akcja (action-value function), żeby on-policy działało. Innymi słowy uczymy się  $q_{\pi}(s, a)$  dla aktualnej polityki  $\pi$ , korzystając ze stanów i akcji z wygenerowanych epizodów.

Z pary  $(S_t, A_t)$  przechodzimy do pary  $(S_{t+1}, A_{t+1})$ , otrzymując za to nagrodę  $R_{t+1}$ . Uczymy się na błędzie wartościowania tych par.



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Nazwa algorytmu pochodzi od piątki SARSA, z której korzystamy podczas aktualizacji ewaluacji.

Jak już mamy powyższy algorytm wartościowania, to algorytm on-policy control już można łatwo zrobić, bo po prostu od razu po każdym wartościowaniu polityki robimy aktualizację polityki zachłannie względem tego wartościowania.

Zbieżność algorytmu SARSA zależy od tego, jak polityka korzysta z wartościowania Q. Dla polityki  $\epsilon$ -zachłannej mamy zbieżność dla nieskończonej liczby odwiedzin każdej pary  $(s, a)$  do optymalnego wartościowania i optymalnej polityki.

## Q-learning

**Q-learning** to algorytm **off-policy temporal difference control**. Uczymy się tutaj bezpośrednio aproksymować optymalną funkcję wartościującą  $q_*$ , bez względu na politykę definiującą zachowanie agenta.

Wzór:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Jedyna różnica względem algorytmu SARSA to sposób estymowania funkcji wartościującej (środkowy wyraz w nawiasie) - bierzemy akcję o największym oczekiwanym zwrocie.

Ważna różnica między algorytmami SARSA i Q-learning to fakt, że **SARSA bierze pod uwagę action selection, a Q-learning nie**. Bierze się to oczywiście z faktu, że pierwsze jest on-policy, więc mamy jedną politykę do podejmowania decyzji i optymalizacji, więc mamy połączoną całą wiedzę. Ważne jest jednak, że dla polityki  $\epsilon$ -zachłannej ze spadającym z czasem  $\epsilon$  obie metody będą zbiegać do polityki optymalnej.

## Expected SARSA

**SARSA z wartościami oczekiwanyimi (expected SARSA)** to coś jak połączenie algorytmów SARSA i Q-learning.

Algorytm działa prawie identycznie jak Q-learning. Jedyna różnica to taka, że zamiast brać akcję o największym wartościowaniu bierzemy wartość oczekiwana po następnych akcjach. Pozwala wziąć to pod uwagę, jak bardzo prawdopodobny w ogóle jest wybór tych akcji przy aktualnej polityce:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \mathbb{E}_\pi [Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \\ &= Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \end{aligned}$$

Algorytm ten deterministycznie porusza się w tym samym kierunku, w którym SARSA porusza się zgodnie z oczekiwaniem (in expectation), czyli zgodnie z aktualnym wartościowaniem.

Co prawda jest to trochę bardziej skomplikowane obliczeniowo, ale za to unika problemu losowego wyboru akcji w polityce  $\epsilon$ -zachłannej, czyli zmniejsza wariancję (bo bierzemy pod uwagę prawdopodobieństwa wyboru). **W praktyce expected SARSA działa lepiej od SARSA i Q-learningu**, w szczególności jest mniej czuła na dobór hiperparametrów.

Można zrealizować expected SARSA w wariantach on-policy i off-policy. Jest wtedy albo bardziej podobna do SARSA, albo do Q-learningu.

## Maximization bias, Double Q-learning

Powysze metody wykorzystują wprost maksymalizację podczas nauki docelowej polityki, czyli wybierają najlepszą akcję. Powoduje to **preferowanie maksymalizacji (maximization bias)**, czyli pewną "ślepotę" na to, że aktualne estymaty wcale nie muszą być prawidłowe. Algorytmy te będą wybierać te akcje, które przy aktualnej estymacji wydają im się najlepsze, nie biorąc pod uwagę tego, że to może być tylko błąd estymaty. Co gorsza, nawet przy dużej liczbie próbek np. Q-learning będzie dalej obciążony (biased) ze względu na to.

Obserwacja: te same próbki są używane do wyboru najlepszej akcji i do jej późniejszej oceny i to właśnie powoduje nadmierny optymizm.

Rozwiązanie: podzielić odpowiedzialność i przechowywać dwie niezależne estymaty.

Rozwiązanie to **Double Q-learning**. Mamy 2 osobne Q-learningi i za każdym razem jeden odpowiada za estymację wartości i wybór akcji o największym oczekiwany zwrocie, a drugi za aktualizację. W danym kroku aktualizowane są estymaty tylko tego drugiego.

Ma co prawda 2 razy większe wymagania pamięciowe (trzeba utrzymywać 2 tablice wartościowań), ale jako że tylko jedna jest aktualizowana, to złożoność obliczeniowa jest bez zmian.

W praktyce wyniki Double Q-learningu są lepsze niż zwykłego Q-learningu.

# n-step bootstrapping

## Definicja problemu

### n-step bootstrapping / TD(n):

- bierze pod uwagę ostatnie n kroków
- spektrum metod pomiędzy TD(0) a Monte Carlo
- rozdzielają krok czasowy bootstrappingu (patrzenia w czasie na możliwości) i aktualizacji; chcemy typowo częstych aktualizacji (żeby od razu uwzględnić zmiany, szybszą zbieżność), ale też odpowiedniego horyzontu obserwacji, żeby zaobserwować ciekawe zmiany (dłuższy bootstrapping)

Różnica tkwi zasadniczo w sposobie wartościowania - zamiast estymować tylko na podstawie jednego kroku, mamy dłuższą i bogatszą historię. W przeciwieństwie do DP mamy jednak kontrolę nad jej wielkością.

Często metody TD(n) dają lepsze wyniki, bo mamy dodatkowy hiperparametr do dostosowania do problemu. Kosztem jest właśnie koszt dodatkowego hiperparametru.

## n-step prediction

**Oczekiwany zwrot w TD(n)** szacujemy za pomocą ostatnich n nagród (je faktycznie znamy) i estymaty późniejszych zwrotów:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

**Wartościowanie stanów w TD(n)** polega na korekcie błędu, czyli różnicę między poprzednim wartościowaniem stanu i obliczonym z powyższego wzoru oczekiwany zwrotem:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha [G_{t:t+n} - V_{t+n-1}(S_t)]$$

Mamy tutaj n-krokowy bootstrapping, bo odwołujemy się do wartościowań innych stanów podczas wartościowania stanu (bootstrapping), ale używamy ostatnich n kroków.

Warto zauważyć, że potrzebujemy n kroków, żeby zrobić aktualizacje. W związku z tym pierwsze n-1 kroków nie jest uwzględniane. Żeby to zrekompensować, na koniec (po kroku terminalnym epizodu), na koniec robimy dodatkowe n-1 update'ów. Jako że wtedy nie mamy kolejnych kroków do dyspozycji, to używamy coraz więcej estymat (G estymuje mniej kroków, szacujemy je V).

n-krokowe zwroty mają własność **redukcię błędu (error reduction property)**, czyli gwarantują zmniejszanie błędu estymacji.

## n-step on-policy control

**n-krokowe sterowanie (n-step control)** jest bardzo podobne do sterowania TD(0), ale używamy n-krokowej estymacji do ewaluacji akcji.

**SARSA(n)** to przykład **sterowania on-policy** opartej o n-krokowy bootstrapping. Jest bardzo podobna do klasycznej Sarsy, różnica tkwi w wartościowaniu akcji.

**n-krokowe wartościowanie akcji** jest bardzo podobne do wartościowania stanów:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n})$$

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)]$$

Dalej mamy klasyczne elementy SARSA:

- wykorzystanie par stan-akcja (tylko ostatnich n)
- on-policy (tylko jedna polityka)
- aktualizacja polityki (żeby była zachłanna względem wartościowania) od razu po aktualizacji wartościowania

## n-step off-policy control

W **n-krokowym sterowaniu off-policy** też mamy elementy typowe dla off-policy:

- polityka docelowa  $\pi$ , która ma zachłannie coraz lepiej przybliżać politykę optymalną
- polityka zachowania  $b$ , która ma zachęcać do eksploracji (np.  $\epsilon$ -zachłanna)
- użycie importance sampling, żeby przenieść naukę z polityki  $b$  na politykę  $\pi$

**Off-policy SARSA(n)** jest bardzo podobna do wariantu on-policy, wystarczy dodać ważenie poszczególnych przykładów używając importance sampling ratio. Wersję on-policy można traktować po prostu jako nieważoną (wszystkie wag 1).

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h, T-1)} \frac{\pi(A_k | S_k)}{b(A_k | S_k)}$$

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)]$$

Warto zauważyć, że zaczynamy iloczyn wag krok później ( $t+1:t+n$ ), bo wartościujemy już konkretną wybraną akcję, więc nie ma sensu rozważać jej własnego prawdopodobieństwa.

W wariantie z wartością oczekiwanaą pomijalibyśmy też ostatni element ( $t+1:t+n-1$ ), bo jego "ogarnia" już średnia ważona z wartością oczekiwanej. Co więcej da się zrobić **wariant bez próbkowania ważności** tą samą ideą - liczymy w każdym kroku (nie tylko ostatnim) wartość oczekiwanaą w oparciu o prawdopodobieństwa z docelowej polityki.

# Planowanie, działanie i uczenie się

## Definicja problemu

Wcześniejsze podejścia dzieliły się wyraźnie na **oparte o model środowiska (model-based)** i **bezmodelowe (model-free)**.

Metody oparte o model (DP) polegają na **planowaniu**, bo znają model, więc mogą w oparciu o niego zaplanować przyszłe działania.

Metody bezmodelowe (Monte Carlo, Temporal Difference, metody n-krokowe) opierają się o **uczenie się**, bo nie znają modelu i muszą z praktycznej obserwacji (typowo w symulacjach) wywnioskować, co robić w konkretnych stanach.

Obie grupy mają duże podobieństwa:

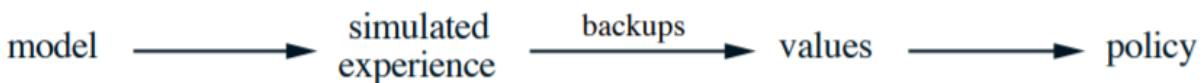
- główna część algorytmu to obliczanie funkcji wartościujących
- opierają się o patrzenie w przyszłość, estymowanie na podstawie tych doświadczeń wartości stanów/akcji i aktualizację na tej podstawie poprzednich estymat

Modele środowiska można podzielić na 2 grupy:

- **modele rozkładów (distribution models)** - budują pełny rozkład prawd. przejść z każdego stanu do innych przy danych akcjach, np. DP
- **modele próbujące (sample models)** - zgodnie z rozkładami prawd. (wiemy, że one są, ale ich nie znamy) generują pojedynczą próbkę i ją zwracają, np. MC

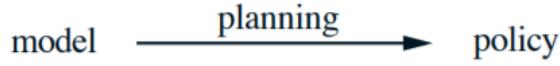
Oba rodzaje modeli pozwalają na zdobywanie **symulowanego doświadczenia (simulated experience)**, czyli uczenie się przez generowanie kolejnych kroków. Dla modeli rozkładów dostajemy dla danego stanu pełny rozkład dalszych możliwości, a dla modeli próbek po prostu kolejny stan. Można też generować całe epizody od razu (np. sampling w Monte Carlo).

Można to przedstawić jako podejście:

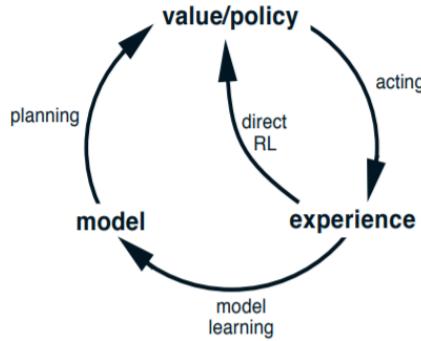


Czyli mamy model środowiska (albo jakąś symulację), zdobywamy doświadczenie na podstawie symulacji (np. generujemy epizody), zbieramy wartości które uzyskaliśmy, na ich podstawie robimy wartościowanie i z jego pomocą aktualizujemy politykę.

**Planowanie** to proces, w którym dostajemy model i na jego podstawie tworzymy lub ulepszamy politykę. Metody RL realizują **planowanie w przestrzeni stanów (state-space planning)**, czyli przeszukiwanie stanów w poszukiwaniu optymalnej polityki. Akcje powodują przejście w tej przestrzeni od stanu do stanu, a funkcje wartościujące są obliczane z wykorzystaniem stanów (samych stanów albo akcji w tych stanach). Samego modelu **można się uczyć (indirect RL, model learning)**.



Ważna różnica względem metod **uczących się** (np. MC) to fakt, że one uczą się z faktycznego wygenerowanego doświadczenia. Jest to typowe podejście trial-and-error, gdzie uczymy się bezpośrednio z doświadczenia, reaktywnie wyciągając wnioski (**direct RL**).



## Q-planning

**Random-sample one-step tabular Q-planning** to prosty algorytm planowania oparty o Q-learning. Zakładamy w nim, że na wejściu mamy model próbujący i będziemy korzystać z Q-learningu do nauki optymalnej polityki dla niego.

Loop forever:

1. Select a state,  $S \in \mathcal{S}$ , and an action,  $A \in \mathcal{A}(S)$ , at random
2. Send  $S, A$  to a sample model, and obtain  
a sample next reward,  $R$ , and a sample next state,  $S'$
3. Apply one-step tabular Q-learning to  $S, A, R, S'$ :  

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

Losowo wybieramy parę stan-akcja, wrzucamy do modelu próbującego, dostajemy dla nich następny stan i nagrodę za przejście. Potem wykonujemy jeden krok Q-learningu.

Jednokrokowy - czyli nie mamy n-krokkowego bootstrappingu. Tabular, bo po prostu robimy implementację tablicą dla par stan-akcja.

## Dyna-Q

**Tabular Dyna-Q** to algorytm do planowania dynamicznego (online), wykorzystujący tabelaryczną reprezentację modelu. Składa się z:

- planowania (planning)
- działania (acting)
- nauki modelu środowiska (model learning)
- bezpośredniej nauki wartościowania i polityki z doświadczenia (direct RL)

Do samej nauki wartościowania używa się random-sample one-step Q-learningu.

**Model środowiska** jest budowany w oparciu o tablicę. Dla wygenerowanego przejścia  $(S_t, A_t) \rightarrow (R_{(t+1)}, S_{(t+1)})$  w tablicy modelu dla współzewnętrznych  $(S_t, A_t)$  zapisujemy wartość  $(R_{(t+1)}, S_{(t+1)})$ . W związku z tym pamiętamy ostatnie przejście dla każdej pary, a model jest deterministyczny.

Zakładamy, że model to to, czego zdążyliśmy doświadczyć do tej pory - im więcej eksploracji, tym dokładniejszy jest nasz model.

Podczas **planowania** algorytmem Q-planning próbujemy jedną losową parę stan-akcja z tych doświadczonych do tej pory (które mamy w modelu).

Initialize  $Q(s, a)$  and  $Model(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$

Loop forever:

- $S \leftarrow$  current (nonterminal) state
- $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- Take action  $A$ ; observe resultant reward,  $R$ , and state,  $S'$
- $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
- $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)
- Loop repeat  $n$  times:  
 $S \leftarrow$  random previously observed state  
 $A \leftarrow$  random action previously taken in  $S$   
 $R, S' \leftarrow Model(S, A)$   
 $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

W zewnętrznej pętli ulepszamy model - bierzemy akcję zgodnie z eksplorującą polityką  $\varepsilon$ -zachłanną i dodajemy wynik do modelu. Przy okazji od razu ulepszamy wartościowanie Q-learningiem, bo czemu nie.

W wewnętrznej pętli n razy wykonujemy Q-planning, ulepszając wartościowanie próbami z naszego ulepszzonego modelu.

Większa liczba kroków planowania oznacza mniej kroków potrzebnych na epizod, bo mamy już pewne przewidywania co do kolejnych kroków i aktualizujemy więcej wartościowań w każdym kroku.

## Dyna-Q+

Algorytm Dyna-Q zakłada **niezmienne środowisko**. W praktyce środowiska są **stochastyczne**, czyli zmienne, a dodatkowo sam algorytm mógł nauczyć się niedokładnych przybliżeń modelu. Żeby to wykryć, trzeba zmusić model do eksploracji.

**Dyna-Q+** to modyfikacja Dyna-Q z zachętą do eksploracji. Zmieniamy w niej nagrodę za przejście:

$$r_+ = r + \kappa\sqrt{\tau}$$

$r$  - prawdziwa nagroda

$\tau$  - liczba kroków czasowych, która minęła od ostatniego takiego przejścia (z danego stanu, daną akcją, do danego nowego stanu)

$\kappa$  - niewielka stała

Realizujemy tutaj zachętą do eksploracji, premując realizację przejść, których dawno nie próbowaliśmy.

## Warianty aktualizacji

Grupa algorytmów Dyna teoretycznie mogłaby wykorzystać model prawdopodobieństwa środowiska, a nie model próbujący jak Dyna-Q. Wtedy zamiast Q-learningu można by użyć np. programowania dynamicznego i dostać optymalne rozwiązania, bo mamy możliwość szacowania wartości oczekiwanej.

Czemu tak nie robimy i dalej używamy aktualizacji z próbek? Koszt obliczeniowy - DP rozważa wszystkie możliwe scenariusze przyszłości, czyli możliwe stany i akcje, a tych w praktyce jest zawsze dużo. Wyznacza to **współczynnik rozgałęzienia (branching factor)** problemu, który często jest duży.

Co więcej, w praktyce warianty próbujące po prostu szybciej dostają dobre wyniki - co prawda każda aktualizacja wartościowań i modelu jest mniej dokładna, ale za to są bez porównania częstsze.

## Próbkowanie trajektorii

W typowym Dyna-Q podczas planowania losowo próbujemy akcję z już napotkanych akcji, z prawdopodobieństwem jednostajnym. To jest nieoptymalne, bo możemy próbować bardzo rzadkie stany, mało ważne etc.

Bardziej inteligentne jest **próbkowanie trajektorii (trajectory sampling)**, w którym symulowane doświadczenie to nie pojedyncza próbka stan-akcja, tylko od razu cały epizod, jak w Monte Carlo.

Czemu to działa? Bo używamy aktualnej polityki w tym epizodzie. Jeżeli aktualna polityka często wybiera jakieś stany i akcje, to z dużym prawdopodobieństwem wystąpią w epizodach. W końcu tylko początkowy stan jest tam losowy - reszta jest zgodna z polityką.

Konsekwencje: **szysza nauka, ale ryzyko zbiegnięcia się do gorszych wyników**. Wynika stąd, że ograniczamy eksplorację - jak polityka się zafiksuje na czymś, to już raczej z tego nie wyjdziemy

## Inne warianty planowania

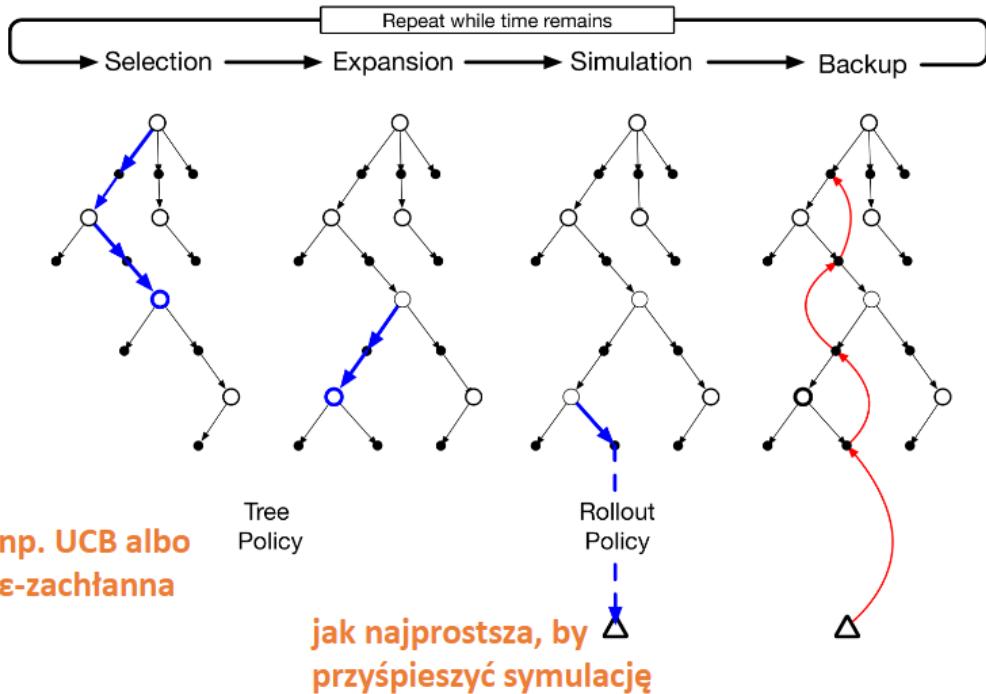
Planowanie w Dyna-Q polega na szukaniu optymalnej polityki na bazie aktualnego modelu. Mamy najpierw fazę aktualizacji modelu, a potem przy niezmiennym modelu środowiska mamy n kroków planowania.

Można przyjąć inną strategię - **planować podczas decyzji (decision-time planning)**, w którym skupiamy się na właściwym podjęciu najbliższej decyzji, będąc w określonym stanie. Dalej konstruujemy wartościowanie i politykę, ale przy każdej decyzji. Jest to bardzo adaptacyjne, ale kosztowne obliczeniowo - działa, gdy mamy dużo czasu na decyzję (z perspektywy maszyny), np. w grach planszowych jak szachy czy go.

**Monte Carlo Tree Search** to przykład tego rodzaju algorytmu. Wykonujemy w nim pętlę tak długo, jak mamy jeszcze czas na podjęcie decyzji:

- selection - wybieramy prostą, ale solidną politykę akcję do wykonania
- expansion - z tą polityką rozwijamy tę akcję na krótkie drzewo dalszych możliwości
- simulation - jak najszybszą i najprostszą (np. losową) polityką rozwijamy drzewo do końca, aż do końca epizodu
- backup - ewaluacja w oparciu o symulowanie doświadczenia

Idea jest taka, żeby najbliższe akcje ogarnąć jak najlepiej, a te dalsze nie wiadomo i tak jakie będą (np. bo otoczenie też reaguje, przeciwnik robi ruchy), więc tam marnujemy jak najmniej czasu, byle dojść do końca epizodu (wymaganie metod Monte Carlo).



Dobre dla problemów, w których mamy sporo czasu na każdą decyzję, ale każda decyzja ma duże konsekwencje i nie ma jak się z niej wycofać.

# Metody przybliżające wartościowanie stanów

## Definicja problemu

Wszystkie poprzednie metody bezpośrednio wartościowały stany z pewnego dyskretnego zbioru. W praktyce rodzi to problemy:

- przestrzeń stanów i akcji są zbyt duże
- stany są ciągłe, np. kąt nachylenia ramienia robota
- stany są ze sobą powiązane, podobne stany mają podobne wartościowania

### Metody przybliżające wartościowanie stanów (state-value approximation methods):

- dokonujemy aproksymacji funkcji wartościującej stany (lub pary stan-akcja), zamiast liczyć ją bezpośrednio
- przypisuje podobne wartościowania podobnym stanom, więc oblicza też implikite podobieństwo stanów; unika problemu potrzeby ewaluacji wszystkiego osobno
- polegają na **nadzorowanym uczeniu (supervised learning)** funkcji aproksymującej funkcję wartościującą:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$\mathbf{w}$  - wektor wag (parametrów)

- wymagają użycia metod uczenia maszynowego, które są nadzorowane, umieją się uczyć online i działają dla danych niestacjonarnych (rozkład zbioru uczącego zmienia się z czasem), czyli działają bez spełnienia własności i.i.d. (independent and identically distributed)

W praktyce chcemy jak najlepiej przybliżać **cel predykcyjny**, czyli minimalizować różnicę między estymacją modelu a prawdziwą funkcją wartościującą - **value error (VE)**. W praktyce jest to **błąd średniokwadratowy (mean squared error, MSE)**:

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[ v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2$$

$$\mu(s) \geq 0, \quad \sum_s \mu(s) = 1$$

Wagi  $\mu$  tworzą rozkład prawdopodobieństwa i wyznaczają, jak bardzo zależy nam na precyzyjnym wartościowaniu poszczególnych stanów. Bez tego by się nie dało, bo przestrzeń stanów jest bardzo duża i trzeba skupić się tylko na tych, które są faktycznie ważne. Ceną jest mniejsza dokładność estymacji pozostałych stanów, ale skoro są rzadkie i mało ważne, to nas tak nie boli.  $\mu(s)$  może być np. ułamkiem czasu spędzonym w danym stanie podczas treningu.

W praktyce na rysunkach etc. operuje się na pierwiastku kwadratowym z MSE (**RMSE, Root MSE**), bo ma on tę samą skalę jednostek co wartościowanie.

Nie ma gwarancji, że minimalizacja błędu predykcji to najlepszy sposób nauki, ale empirycznie to działa i co więcej działa dobrze, no i nie mamy nic lepszego. Dodatkowo trzeba pamiętać o tym, że to tylko aproksymacja - może zbiegać do optimów lokalnych, a nie globalnego.

## Metody gradientowe i semi-gradientowe

Metody gradientowe, oparte o poruszanie się w kierunku gradientu funkcji, w szczególności **stochastyczny spadek wzduż gradientu (stochastic gradient descent, SGD)**, szczególnie dobrze nadają się do nauki online, bo uczą się krok po kroku.

Wzór to klasyka - bierzemy poprzednie wagę, liczymy pochodne:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[ v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \\ &= \mathbf{w}_t + \alpha \left[ v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

Czyli po prostu poprawiamy poprzedni wektor wag o iloczyn kroku uczącego, błędu aproksymacji i wektora pochodnych po poszczególnych wagach.

Ważne jest, że **krok uczący musi być mały** - w RL mamy różne przykłady, zmienne środowisko, więc żaden przykład nie może zdominować pozostałych. Co więcej żeby było to zbieżne, nawet do minimum lokalnego, to krok uczący musi maleć z czasem.

Powyższe zakłada jednak, że perfekcyjnie dobrze znamy cel aproksymacji, czyli wartościowanie  $v_\pi(S_t)$ , co jest nierealne. Zamiast tego mamy pewne przybliżenie  $U_t$ , czyli na przykład wyniki z symulacji Monte Carlo. Wzór nie zmienia się minimalnie:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[ U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

Ważne tylko, żeby estymata była **nieobciążona (unbiased)**, czyli żeby jej wartością oczekwaną było prawdziwe wartościowanie. W połączeniu z warunkami co do stałej uczącej gwarantuje to zbieżność do minimum lokalnego.

$$\mathbb{E}[U_t | S_t = s] = v_\pi(S_t)$$

Dla metod Monte Carlo tak jest, bo zwrot  $G_t$  z epizodu jest nieobciążoną estymatą wartościowania. Natomiast nie jest tak dla metod z bootstrappingiem (DP, n-step TD), bo zależą one od aktualnych estymat, a zatem od aktualnego wektora wag. Innymi słowy, pochodna prawdziwej wagi nie znika jak dla metod gradientowych:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[ v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \\ &= \mathbf{w}_t + \alpha \left[ v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

Natomiast możemy radośnie zignorować ten fakt i dostaniemy wtedy **metody semi-gradientowe (semi-gradient methods)**, działające dla n-krokovego bootstrappingu. W praktyce takie zignorowanie oznacza, że uwzględniają one wpływ gradientu na estymatę wartości, ale nie na prawdziwą wartość. Brak formalnej gwarancji zbieżności, ale w praktyce są zbieżne dla ważnych klas problemów.

Niestety w praktyce takie metody oparte o TD(n) mają często gorszy punkt zbieżności od metod gradientowych, ale za to są szybsze, mają mniejszą wariancję i nie wymagają nauki epizodycznej.

Najprostsza aproksymacja funkcji to **agregacja stanów (state aggregation)** - grupujemy stany w paczki po np. 100, każda grupa dostaje jedną wspólną wagę w wektorze  $w$ , więc dla każdej grupy estymujemy jedną wartość. Aktualizujemy tylko ten jeden komponent, tak że gradient dla niego to 1, a dla pozostałych elementów w grupie 0.

## Metody liniowe i nieliniowe

**Metody liniowe (linear methods)** opisują stan jako wektor cech  $w(s)$ , a przybliżenie wartościowania to funkcja liniowa wag i wartości cech:

$$\hat{v}(s, w) \doteq w^\top x(s) \doteq \sum_{i=1}^d w_i x_i(s)$$

Aktualizacja jest prymitywnie prosta, bo pochodna to po prostu wektor cech:

$$\nabla \hat{v}(s, w) = x(s)$$

$$w_{t+1} \doteq w_t + \alpha [U_t - \hat{v}(S_t, w_t)] x(S_t)$$

Ze względu na taką prostotę mają bardzo ograniczoną siłę wyrazu, bo trzeba zrobić dobrą inżynierię cech dla stanu. Natomiast dowodzenie zbieżności jest bardzo proste, bo jest tylko jedno optimum, więc jest i lokalne, i globalne. Zbieżność jest gwarantowana i dla gradientowego MC, i dla semi-gradientowego TD(0).

Jak zrobić dobre cechy? To jest trudne. Po pierwsze muszą faktycznie reprezentować stan i pozwalać na generalizację między stanami, czyli wyróżnienie stanów podobnych, więc to np. odczyty z różnych sensorów. Liniowa aproksymacja nie wyłapuje jednak zależności między cechami, więc trzeba to rozwiązać na poziomie inżynierii cech, np. przez cechy wielomianowe, używając funkcji bazowych Fouriera albo innymi metodami tworzenia cech z interakcjami.

Dla takiego wektora cech można użyć dowolnego algorytmu przybliżania funkcji z uczenia maszynowego, np. k najbliższych sąsiadów.

**Metody nieliniowe** to głównie sieci neuronowe, modelujące właściwie dowolnie złożone zależności między cechami.

## Sterowanie on-policy oparte o aproksymację - epizodyczne

**Sterowanie on-policy z aproksymacją dla zadań epizodycznych** jest dość prostą modyfikacją poprzednich algorytmów on-policy, np. SARSA. Są 2 różnice: akcję wybieramy za pomocą wartościowania z wykorzystaniem aktualnych wag, a samych wag uczymy się korzystając z aproksymacji semi-gradientowej.

**Epizodyczna semi-gradientowa SARSA z aproksymacją** wygląda następująco:

Input: a differentiable action-value function parameterization  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$   
Algorithm parameters: step size  $\alpha > 0$ , small  $\varepsilon > 0$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Loop for each episode:

$S, A \leftarrow$  initial state and action of episode (e.g.,  $\varepsilon$ -greedy)

Loop for each step of episode:

Take action  $A$ , observe  $R, S'$

If  $S'$  is terminal:

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

Go to next episode

Choose  $A'$  as a function of  $\hat{q}(S', \cdot, \mathbf{w})$  (e.g.,  $\varepsilon$ -greedy)

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

$S \leftarrow S'$

$A \leftarrow A'$

Analogicznie działa dla wariantu z n-krokowym bootstrappingiem, tylko wariancja rośnie wraz z n.

## Sterowanie on-policy oparte o aproksymację - ciągłe

**Sterowanie on-policy z aproksymacją dla zadań ciągłych** wymaga pewnej modyfikacji. Do tej pory były 2 główne podejścia: epizodyczne i ciągłe ze zniżkami (discounting). Żeby użyć aproksymacji dla zadań ciągłych, trzeba jednak porzucić podejście ze zniżkami i zamiast tego zdefiniować **cel jako średnią nagrodę**. Jest to dość podobne do zniżek, ale po prostu liczymy średnią nagrodę, z taką samą wagą dla natychmiastowych i przyszłych nagród.

Jakość polityki wyznacza w tym podejściu **średnia nagroda** otrzymywana podążając za nią (do nieskończoności, bo zadania ciągłe), czyli  $r(\pi)$ :

$$\begin{aligned} r(\pi) &\doteq \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \\ &= \lim_{t \rightarrow \infty} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi], \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r \end{aligned}$$

Powysze zakłada, że przejście z linijki 2 do 3 jest możliwe. Żeby tak było, problem musi być **ergodyczny (ergodic)**, co jest warunkiem koniecznym (ale nie wystarczającym). Oznacza to, że stan początkowy i wczesne decyzje agenta mają tylko tymczasowy efekt - w dłuższej perspektywie wartość oczekiwana stanu ma zależeć tylko od polityki i prawdopodobieństw przejść między stanami z MDP.

Zwroty w tym podejściu to **zwroty różnicowe (differential returns)**, czyli suma różnic między faktycznymi nagrodami a średnią nagrodą:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$

Funkcje wartościujące stany w tym podejściu to **różnicowe funkcje wartościujące (differential value functions)**.

Znowu trzeba też zaktualizować równania Bellmana, bo teraz nie mamy współczynnika zniżki  $\gamma$ , więc go usuwamy. Poza tym jedyna różnica to wrzucenie do wzorów różnicy między faktyczną nagrodą a średnią nagrodą:

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{r,s'} p(s',r|s,a) [r - r(\pi) + v_\pi(s')] \\ q_\pi(s, a) &= \sum_{r,s'} p(s',r|s,a) [r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s',a')] \\ v_*(s) &= \max_a \sum_{r,s'} p(s',r|s,a) [r - \max_\pi r(\pi) + v_*(s')] \\ q_*(s, a) &= \sum_{r,s'} p(s',r|s,a) [r - \max_\pi r(\pi) + \max_{a'} q_*(s',a')] \end{aligned}$$

Z uwzględnieniem tych równań można też prosto zdefiniować wartościowania stanów i akcji dla wariantu TD:

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)$$

Jak zaaplikujemy powyższe do algorytmu SARSA, to dostaniemy **algorytm różnicowego semi-gradientowego SARSA (differential semi-gradient SARSA)**.

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

Input: a differentiable action-value function parameterization  $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$   
Algorithm parameters: step sizes  $\alpha, \beta > 0$ , small  $\varepsilon > 0$   
Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )  
Initialize average reward estimate  $\bar{R} \in \mathbb{R}$  arbitrarily (e.g.,  $\bar{R} = 0$ )  
Initialize state  $S$ , and action  $A$   
Loop for each step:  
    Take action  $A$ , observe  $R, S'$   
    Choose  $A'$  as a function of  $\hat{q}(S', \cdot, \mathbf{w})$  (e.g.,  $\varepsilon$ -greedy)  
     $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$   
     $\bar{R} \leftarrow \bar{R} + \beta \delta$   
     $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla \hat{q}(S, A, \mathbf{w})$   
     $S \leftarrow S'$   
     $A \leftarrow A'$

## Sterowanie off-policy oparte o aproksymację

Są tutaj dwa problemy. Pierwszy to tradycyjny dla off-policy rozdziałek, że aktualizujemy politykę docelową próbami z polityki behawioralnej. To nam załatwia importance sampling i odpowiednie ważenie.

Potrzeba jeszcze drugiej, trudniejszej do realizacji korekty - **poprawy rozkładu częstotliwości aktualizacji**. W aproksymacji różne stany na siebie wpływają, bo są agregowane implicitem wektorem wag. W związku z tym musimy to oddać odpowiednim rozkładem częstotliwości aktualizacji poszczególnych stanów. Niestety nie wiadomo jak to zrobić tak, żeby zawsze dobrze działało.

Powoduje to **niestabilność** - możliwość zmian wag tak, że są rozbieżne, np. podobne stany "napędzają" wzajemnie swoje wartościowania i rosną do nieskończoności. Żeby sobie z tym radzić, trzeba uczyć się bardzo powoli

Daje to zabójczą mieszankę w postaci **śmiertelnej trójcy (the deadly triad)**:

- aproksymacja funkcji - konieczne, by generalizować i radzić sobie z potężnymi przestrzeniami stanów i akcji
- bootstrapping - konieczne, by przyspieszyć uczenie (możliwa nauka podczas epizodów) i zmniejszyć wariancję
- uczenie off-policy - konieczne, by odseparować politykę docelową (uczącą się optymalnej polityki) i behawioralną (mającą zachować eksplorację)

Potrzebujemy wszystkich 3, ale będzie to skutkowało nieuniknioną niestabilnością i problemami z uczeniem. Gdy wybierzemy dowolne 2 z 3, to tego problemu da się uniknąć.

# Metody oparte o gradient polityki

## Definicja problemu

Prawie wszystkie podejścia do tej pory to przedstawiciele **metod opartych o ewaluację akcji (action-value methods)**, które najpierw wyceniają akcję, a potem budują politykę z najlepszych akcji.

### Metody oparte o gradient polityki (policy gradient methods):

- uczymy się od razu bezpośrednio polityki, optymalizując pod nią, bez kroku pośredniego z ewaluacją akcji
- bardzo podobne do metod bandytów gradientowych, można to interpretować tak, że one są specyficznym przypadkiem gradientu polityki dla tylko 1 stanu
- mamy **politykę parametryzowaną (parameterized policy)**, opisywaną przez wektor parametrów  $\theta$
- optymalizacja wykorzystuje **gradient polityki**, maksymalizując efektywność (performance):

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$$

- we wzorach powyższe robi się wspinaniem wzdłuż gradientu (gradient ascent), w praktyce zmienia się znak i używa się SGD
- mogą też zawierać naukę funkcji przybliżającej wartościowanie stanów

### Metody aktor-krytyk (actor-critic methods):

- podgrupa metod opartych o gradient polityki
- robią i aproksymację polityki (policy gradient), i wartościowanie akcji (action-value)
- aktor - polityka, której się uczymy, działa w środowisku
- krytyk - funkcja wartościująca, której się uczymy, zwykle wartościuje stany (state-value function)

## Gradient polityki

Po co nam właściwie takie alternatywne podejście? Bo jest bardzo proste i daje bardzo bogate możliwości parametryzacji, czyli można użyć bardzo mocnych algorytmów, takich jak sieci neuronowe. Jak mamy dyskretny zbiór stanów, to polityka sprowadza się do wzoru:

$$\pi(a|s, \theta) = \frac{\exp(h(s, a, \theta))}{\sum_b \exp(h(s, b, \theta))}$$

$h(s, a, \theta)$  - funkcja wyznaczająca preferencje akcji dla poszczególnych stanów, parametryzowana wektorem  $\theta$ , np. sieć neuronowa

Dlatego to jest fajne? Bo nie mamy tutaj sztywnych i niezmiennych wartości dla poszczególnych akcji, tylko od razu ogólną politykę, która podczas treningu może zbiegać do polityki deterministycznej. Jest to duża przewaga nad zestawem action-value +  $\varepsilon$ -greedy, bo tam mamy do końca niewielkie prawdopodobieństwo wyboru słabych akcji  $\varepsilon$ .

W szczególności takie podejście może poradzić sobie z sytuacją, gdy mamy aproksymowanie stanów (niezbędne dla wielu problemów), a optymalne jest wybieranie z pewnej puli akcji. W takich sytuacjach optymalna jest polityka stochastyczna, czyli z pewnym rozkładem prawdopodobieństw wyboru optymalnych akcji. W gradiencie polityki jak najbardziej da się to zrobić (softmax nam to ogarnie), w poprzednich podejściach (które były zachłanne względem wartościowania, więc brały zawsze 1 akcję) nie.

Teraz przydałoby się nauczyć tej polityki, więc na dobry początek trzeba mieć miarę jej **efektywności (performance)**. W szczególności dla zadań epizodycznych można to zrobić prosto, przyjmując, że każdy epizod zaczyna się od pewnego stanu  $s_0$ . Efektywność definiujemy jako wartościowanie tego stanu:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

Jak policzyć gradient z takiej efektywności? Problem w tym, że zmiana polityki zmienia też rozkład późniejszych stanów w ramach epizodu. Na szczęście mamy do tego twierdzenie

**Twierdzenie o gradiencie polityki dla zadań epizodycznych** podaje wzór na gradient efektywności (performance) ze względu na parametry polityki, bez względu na pochodne stanów:

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_{\pi}(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \\ &= \mathbb{E}_{\pi} \left[ \sum_a q_{\pi}(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta}) \right] \end{aligned}$$

$\mu$  - rozkład on-policy dla polityki  $\pi$

## REINFORCE

**REINFORCE** to algorytm Monte Carlo policy gradient. Z twierdzenia o gradiencie polityki dla zadań epizodycznych wiemy, jak poruszać się w kierunku zgodnym z gradientem. Za to z metod Monte Carlo wiemy, jak dla zadań epizodycznych dostać estymatę wartości oczekiwanej. Pomijając wyprowadzenie, aktualizacja parametrów polityki to:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

Jak widać, każda zmiana to iloczyn zwrotu z epizodem  $G_t$  i wektora z gradientami prawdopodobieństw poszczególnych akcji podzielonych przez faktyczne prawd. tych akcji. Pierwszy element ciągnie w stronę akcji o dużym zwrocie, a drugi waży je tak, żeby częste akcje (pomimo potencjalnie małych zwrotów) nie dominowały.

Często powyższe zapisuje się jako gradient z logarytmu polityki, bo:

$$\nabla \ln x = \frac{\nabla x}{x}$$

Uwzględniając to, pełny algorytm REINFORCE jest krótki i prosty:

Input: a differentiable policy parameterization  $\pi(a|s, \theta)$

Algorithm parameter: step size  $\alpha > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  (e.g., to  $\mathbf{0}$ )

Loop forever (for each episode):

Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot| \cdot, \theta)$

Loop for each step of the episode  $t = 0, 1, \dots, T - 1$ :

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$$

$$\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t | S_t, \theta)$$

## Metody z punktem odniesienia

**Metody z punktem odniesienia (methods with baseline)** są oparte na obserwacji, że skoro we wzorze na gradient polityki jest znak proporcjonalności, to można go zmodyfikować, dodając dowolny punkt odniesienia (baseline):

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \left( q_\pi(s, a) - b(s) \right) \nabla_\theta \pi(a|s, \theta)$$

$$\sum_a b(s) \nabla_\theta \pi(a|s, \theta) = b(s) \nabla_\theta \sum_a \pi(a|s, \theta) = b(s) \nabla_\theta 1 = 0$$

Jaki może być taki punkt odniesienia? Dla metod MC najprościej po prostu wartościowanie stanu:

$$\theta_{t+1} \doteq \theta_t + \alpha \left( G_t - b(S_t) \right) \frac{\nabla \pi(A_t | S_t, \theta_t)}{\pi(A_t | S_t, \theta_t)}$$

Dobrze dobrany punkt odniesienia właściwie nic nie zmienia w samym algorytmie, ale potrafi zmniejszyć jego wariancję i przyspieszyć zbieżność.

Warto zwrócić uwagę - przywracamy tutaj do użytku wartościowanie stanów jako metodę pomocniczą do gradientu polityki.

## Metody aktor-krytyk

Mamy wersję epizodyczną z użyciem Monte Carlo. Z algorytmów tablicowych wiemy, że z bootstrappingiem jest stabilniej i szybciej. Jak zrobimy tutaj to samo, to dostaniemy algorytm **aktor-krytyk**:

```
Input: a differentiable policy parameterization  $\pi(a|s, \theta)$ 
Input: a differentiable state-value function parameterization  $\hat{v}(s, \mathbf{w})$ 
Parameters: step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$ 
Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $\mathbf{w} \in \mathbb{R}^d$  (e.g., to  $\mathbf{0}$ )
Loop forever (for each episode):
    Initialize  $S$  (first state of episode)
     $I \leftarrow 1$ 
    Loop while  $S$  is not terminal (for each time step):
         $A \sim \pi(\cdot|S, \theta)$ 
        Take action  $A$ , observe  $S', R$ 
         $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$            (if  $S'$  is terminal, then  $\hat{v}(S', \mathbf{w}) \doteq 0$ )
         $\mathbf{w} \leftarrow \mathbf{w} + \alpha^w \delta \nabla \hat{v}(S, \mathbf{w})$ 
         $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla \ln \pi(A|S, \theta)$ 
         $I \leftarrow \gamma I$ 
         $S \leftarrow S'$ 
```

Mamy tutaj 2 parametryzowalne rzeczy: politykę i funkcję wartościującą stany. Podczas generowania kolejnych epizodów zgodnie z obecnymi wartościami podejmujemy akcję i obliczamy błąd wartościowania. Potem gradientowo aktualizujemy parametry polityki i wartościowania.

Proste, pozwala użyć sieci neuronowych, zwykły spadek po gradiencie. Problemem jest duża niestabilność - trzeba używać bardzo małych stałych uczących, żeby to było zbieżne.

# Metody interpretowalne

## Proste metody regresji

### Regresja parametryczna i nieparametryczna

**Regresja** - zadanie uczenia nadzorowanego (supervised learning) polegającego na przewidywaniu wartości ciągłej. Wartość ta może być albo z pełnego zakresu liczb rzeczywistych, albo ograniczona z którejś strony (np. tylko nieujemne), albo z obu.

**Metody parametryczne** to takie, które mają stałą liczbę parametrów, niezmienną bez względu na ilość danych treningowych. Innymi słowy kiedy dodamy przykłady do zbioru treningowego, to liczba parametrów modelu pozostanie taka sama. Zakładają też z góry pewną postać funkcyjną dla naszych danych.

**Metody nieparametryczne** mają liczbę parametrów bezpośrednio zależną od wielkości zbioru treningowego. Nie zakładają one żadnej postaci funkcyjnej, tylko uczą się bezpośrednio z danych.

#### Metody regresji parametrycznej:

- regresja liniowa - wysoce inherentnie interpretowalna, model liniowy
- regresja logistyczna - wysoce inherentnie interpretowalna, model liniowy z prostym szacowaniem prawdopodobieństw
- regresja wielomianowa - dla stopnia 2 interpretowalna (proste iloczyny / kwadraty cech), wyżej nie
- stepwise regression - nieinterpretowalna
- ridge regression - podobnie interpretowalna jak liniowa
- lasso regression - bardziej interpretowalna od liniowej dla wielu cech, bo dąży do rzadkich przestrzeni cech (trzeba rozważać mniej cech)
- ElasticNet regression - połączenie ridge i lasso, podobnie do nich interpretowalna
- surrogate model (kriging, Gaussian process regression) - lokalna interpretowalność

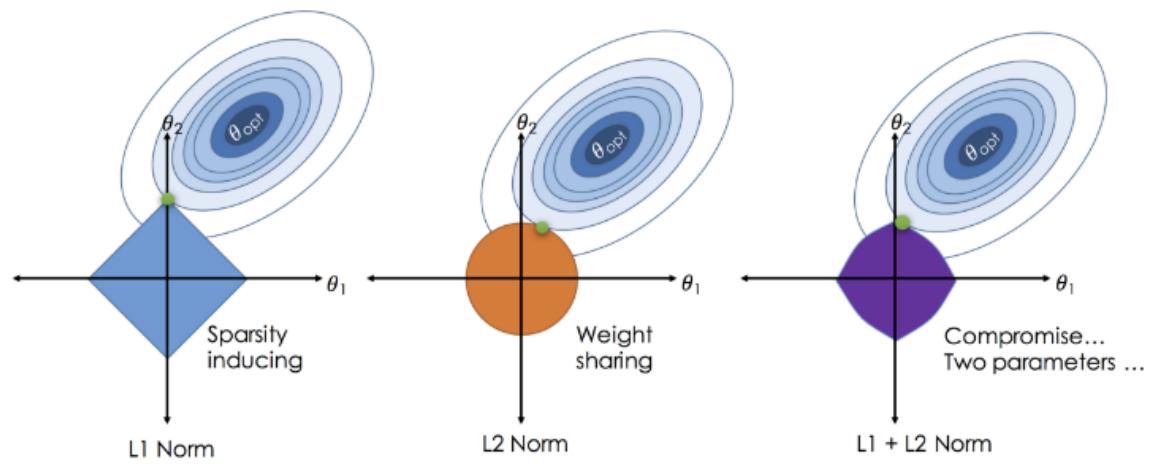
## Zalety i wady ridge regression i lasso regression

**Ridge regression** - regresja liniowa z regularizacją L2, która przeciwdziała overfittingowi przez penalizowanie dużych wag cech, zmniejszając je, ale do niezerowych wartości.

**LASSO regression** - regresja liniowa z regularizacją L1, która realizuje przy okazji selekcję cech, bo penalizuje wagi i szczególnie premiuje wagi równe dokładnie 0.

**ElasticNet regression** - mamy i L1, i L2 naraz.

Czemu tak jest? Słynny obrazek:



Każda funkcja regresji liniowej ma wypukły (convex) kształt funkcji kosztu RSS, taki kształt "tornada" z pewnym minimum. Dla regresji bez regularizacji po prostu wybieramy to minimum. Kiedy to zregularizujemy, to nie jest już tak prosto - trzeba zbalansować oba cele. Oznacza to wybór punktu styku między powierzchnią funkcji kosztu i regularizacji.

Norma L1 używana przez LASSO to wielowymiarowy kwadrat, który ma wierzchołki na osiach. Punkt styku wypada zatem typowo na osi, więc waga tej cechy ma wartość 0, więc nam znika i mamy mniej cech.

Norma L2 używana przez ridge to wielowymiarowy okrąg, więc minimum wypada bliżej środka układu współrzędnych (mniejsze wagi), ale nie przy zerach. Cechy nie znikają więc.

**Uwaga - każda regularizacja wymaga ustandaryzowanych zmiennych** (odjęta średnia, podzielone przez odchylenie standardowe). Wynika to z tego, że regularizacja opiera się o bezpośrednie wykorzystanie współczynników cech, które muszą być jednego rzędu wielkości - standaryzacja nam to zapewnia.

### Ridge regression:

- zalety:
  - lepsze do regularyzacji
  - prostsze i szybsze w treningu - różniczkowalne
  - mniejsze wagi cech - bardziej "ogarnialne" dla ludzkiego umysłu małe liczby i łatwiejsze do porównywania, więc bardziej interpretowalne
  - pozwala sobie radzić z cechami zależnymi liniowo
- wady:
  - nie zmniejsza liczby cech, jak było ich dużo na wejściu, to ciężko to porównywać

### LASSO regression:

- zalety:
  - feature selection - mniej cech do rozważania, więc bardziej interpretowalne
- wady:
  - słabsze wyniki w regularyzacji
  - cięższe w treningu - wymaga optymalizatorów działających dla funkcji nieróżniczkowalnych

## Regresja liniowa i logistyczna jako narzędzia interpretowalne

Na początek regresja liniowa. Wymaga **ustandardyzowanych (standardized)** cech (nawet bez regularyzacji), bo tylko wtedy można bezpośrednio porównywać ze sobą współczynniki poszczególnych cech.

Dodatkowo, jeżeli wszystkie zmienne są **wycentrowane (centered)**, czyli mają średnią zero (załatwione w standaryzacji), to wyraz wolny (bias) można traktować jako **średnią odpowiedź modelu**. Wynika to z tego, że jakby wszystkie zmienne miały wartość zero, to zwracamy sam bias, a że są wycentrowane, to średnia wartość wszystkich cech to właśnie zero.

Jeżeli dodamy do tego **selekcję cech (feature selection)** używając LASSO regression to przy okazji dostaniemy mniej cech do analizy.

**Interpretacja wzajemnych zależności wag jest prosta**, bo mamy model addytywny - dodajemy do siebie kolejne iloczyny wag i wartości cech, dzięki czemu wzrost cechy o  $x$  doda do wyniku  $waga \cdot x$ .

Mamy tu jednak bardzo konkretne wady. Po pierwsze jest to model liniowy, a zatem **nie wykrywa zależności nieliniowych** - wszystko tego typu trzeba podać explicite na wejście jako cechy. Dodatkowo **daje niezbyt dobre wyniki**, bo to w końcu bardzo prosty model, a ufanie interpretacji modelu, który niezbyt dobrze oddaje rzeczywistość, to raczej zły pomysł.

**Interpretacja wag jest nieintuicyjna.** Po pierwsze wagi to nie są ważności same w sobie, tylko liniowa korelacja danej cechy z celem regresji. Po drugie wagi uwzględniają inne cechy - w szczególności jeżeli mamy mocno skorelowane ze sobą cechy to te interakcje są mocne.

Wyniki i korelacje cech poprawia nieco ridge regression, ale same wady pozostają.

**Regresja logistyczna** polega na "wbiciu" siłą wyników regresji liniowej w zakres  $[0, 1]$  za pomocą funkcji softmax. Interpretuje się to jako prawdopodobieństwo klasy pozytywnej i używa do klasyfikacji.

Poszczególne wagi dalej mają współczynniki, ale **interpretacja wag cech jest nieliniowa** - wina funkcji softmax, wagi wpływają na prawdopodobieństwo nieliniowo. Natomiast regresja logistyczna to **model liniowy dla logarytmów szans (log-odds)**.

Wiele wad i zalet jest współdzielonych z regresją liniową, na przykład trzeba ręcznie dodać interakcje między cechami, a wyniki nie są zbyt dobre.

Zwraca **prawdopodobieństwo klas**, więc pod tym względem jest to bardzo interpretowalne. Co więcej to prawdopodobieństwo jest dobrze skalibrowane - faktycznie odpowiada rzeczywistemu, nie jest obciążone (biased) w żadną stronę. Dzięki temu można zaufać temu prawdopodobieństwu i traktować je jak faktyczny zakres ufności klasyfikatora.

Natomiast **wagi są niekoniecznie interpretowalne, bo wpływ wagi na wynik jest nieliniowy**. Mamy tu model multiplikatywny i nieliniowy, bo jest liniowy tylko w przestrzeni log-odds. Przykładowo, dla wagi (log-odds ratio) 0.7 zwiększenie cechy o 1 zwiększy szanse o  $e^{0.7}$ , co jest raczej niezbyt intuicyjne.

## ElasticNet regression

**Regresja ElasticNet** to po prostu połączenie regularizacji L1 i L2 w określonych proporcjach. Łączy modele ridge i LASSO.

Mamy połączenie zalet i wad obu podejść. Dodatkowo w praktyce zwykle daje lepsze wyniki od obu z nich oraz lepiej radzi sobie ze skorelowanymi cechami. Wymaga za to optymalizacji większej liczby hiperparametrów (siła regularizacji oraz proporcje L1 i L2).

## Kriging

**Regresja procesem Gaussowskim (Gaussian process regression) / kriging** - metoda regresji nieparametrycznej oparta o ważenie punktów za pomocą funkcji jądra. Przy rozsądnie dobranym priorze jest to best linear unbiased estimator (BLUE).

Można ją traktować po prostu jako wariant ważonej lokalnie regresji nieparametrycznej. Dla nowych punktów dokonujemy interpolacji wartości nowego punktu, ważąc punkty za pomocą Gaussowskiej funkcji jądra (kernel function).

Można to interpretować bayesowsko. Zaczynamy z pewnym założonym rozkładem a priori (prior) nad możliwymi funkcjami regresji. Tym priorem jest proces Gaussowski. Zakładamy więc, że wartości regresji dla naszego zbioru uczącego mają rozkład normalny, a macierz kowariancji jest wyznaczona za pomocą jądra rozkładu normalnego (Gaussian kernel).

Dla nowej wartości predykcji dokonujemy, szacując rozkład a posteriori (posterior) z twierdzenia Bayesa, łącząc prior (rozkład normalny) z Gaussowską funkcją wiarygodności (likelihood function). Wynikiem takiej regresji też jest rozkład normalny, ze średnią i odchyleniem standardowym oszacowanymi na zbiorze treningowym.

Dużą zaletą takiego podejścia jest **obliczanie przedziałów ufności (confidence intervals)**. Dzięki temu możemy dostać zakres, co do którego jesteśmy pewni, że mieści się tam np. 95% możliwych wartości. Takie oszacowanie niepewności zwiększa interpretowalność.

Dodatkowo taki model **daje bardzo dobre wyniki**.

Wadą jest **duży koszt obliczeniowy** - metoda jest nieparametryczna, złożoność to  $\mathcal{O}(n^3)$  dla n punktów w zbiorze uczącym.

# Drzewa decyzyjne i lasy losowe

## Drzewo decyzyjne, algorytm konstrukcji

**Drzewo decyzyjne** to nieparametryczny algorytm klasyfikacji/regresji oparty o rekurencyjny podział przestrzeni. Polega na utworzeniu drzewa, typowo binarnego, które podczas treningu w każdym węźle uczy się podziału, czyli prostej reguły if/else. Tworzenie optymalnych drzew jest problemem NP-zupełnym, dlatego w praktyce używa się zachłanego algorytmu top-down **recursive binary splitting**, optymalizującego każdorazowo dany podział.

Konstrukcja przebiega rekurencyjnie, tak samo dla korzenia i dalszych węzłów. Nauka węzła sprowadza się do wyznaczenia **punktu podziału (cut-off point)**, czyli wartości wybranej cechy, według której robi się wybór if/else. Rodzaj zależy od typu atrybutu:

- ciągły - reguła to " $\geq X$ ", czyli próg podziału to liczba rzeczywista; do lewego dziecka trafiają punkty o mniejszej wartości tej cechy, do prawego większej lub równej
- binarny/kategoryczny - reguła to " $=X$ "; do lewego dziecka trafiają punkty o wartości atrybutu różnej od X, a do prawego równej X

Jak wybrać taki punkt podziału? Rozważa się wszystkie możliwości. Idziemy po kolej po atrybutach, a dla każdego atrybutu po wszystkich jego wartościach. Dla każdej wartości dzielimy punkty na 2 grupy: o wartości mniejszej i większej lub równej (analogicznie równy/różny dla cech binarnych/kategorycznych) i sprawdzamy jakość tego podziału. Są tutaj do wyboru różne **metryki** do optymalizacji, opisane niżej. Wybieramy najlepszy split, dzielimy zbiór uczący, wywołujemy rekurencyjnie dla dzieci.

Atrybut może się też "wyczerpać", czyli nie ma sensu drugi raz używać do na dole drzewa do podziału. Przykładowo atrybut binarny jest używalny tylko raz - potem już wszystko jest podzielone. Wyczerpać się mogą tylko atrybuty kategoryczne i binarne.

Taki podział dokonuje się do momentu, kiedy węzeł jest **czysty (pure)**, czyli zawiera tylko przykłady z jednej klasy, lub tylko o jednej wartości regresji. Jest to jednoznaczne z perfekcyjnym overfittingiem na danych, więc typowo definiuje się dodatkowe **warunki stopu (stopping criteria)**, żeby zostawić nieczyste liście, ale zregularyzować drzewo.

Predykcja wymaga najpierw użycia wyuczonych reguł if/else i dojścia do liścia według cech nowego przykładu. Dla liścia dla klasyfikacji zwracamy albo najpopularniejszą klasę w liściu, albo procentowy udział poszczególnych klas (więc można dostać bez problemu prawdopodobieństwa klas). Dla regresji zwraca się wartość średnią celu regresji z punktów w liściu.

## Funkcja kosztu

### Information gain:

- klasyfikacja
- information gain to ilość informacji, którą przynosi nam podział, czyli zmniejszenie entropii zawartej w danych dzięki podziałowi
- idea: czystsze węzły, czyli zawierające mniej pomieszane klasy, są prostsze, więc wystarczy mniej bitów, żeby je opisać, zatem mają mniejszą entropię
- chcemy maksymalizować; maksymalizacja information gain jest równoważna minimalizacji entropii
- jest to różnica między sumą entropii dzieci przy danym podziale, ważona liczbą punktów, które trafiły do poszczególnych dzieci, a entropią rodzica:

$$H(X) = \sum_i^N p(x_i) \log p(x_i)$$

$$IG(X) = \frac{|X_{left}|}{|X|} * H(X_{left}) + \frac{|X_{right}|}{|X|} * H(X_{right}) - H(X)$$

$H(X)$  - entropia Shannona

- ważenie bierze się stąd, że mało punktów o dużej entropii boli nas dużo mniej, niż dużo punktów o dużej entropii

### Współczynnik Giniego (Gini coefficient):

- klasyfikacja
- oczekiwany błąd dla punktów, gdyby przypisywać im losową klasę zgodnie z proporcjami klas w zbiorze
- im bardziej jednolity klasowo zbiór, tym mniejszy błąd, bo po prostu trafiemy prawidłową klasę
- chcemy minimalizować
- wzór:

$$Gini(X) = \sum_{y \in Y} p_y * (1 - p_y) = 1 - \sum_{y \in Y} p_y^2$$

$$Gini_{coeff}(X) = Gini(X) - \frac{|X_{left}|}{|X|} * Gini(X_{left}) + \frac{|X_{right}|}{|X|} * Gini(X_{right})$$

Wada obu powyższych metod - **silnie preferują atrybuty o większej liczbie wartości**. W związku z tym wolą atrybuty ciągłe od kategorycznych, a kategoryczne od binarnych.

W praktyce jak chodzi o wyniki nie ma szczególnych różnic między nimi. Obliczanie współczynnika Giniego jest nieco szybsze, bo nie wymaga obliczania logarytmu (ze wzoru na entropię Shannona), ale to praktycznie pomijalne.

### **Suma kwadratów różnic (Residual Sum of Squares, RSS):**

- regresja
- suma kwadratów odległości między punktami a środkowym punktem z liścia

### **Kryteria stopu**

Możliwym kryterium stopu jest **minimalna poprawa metryki** - wymagamy, żeby poprawa wynosiła co najmniej jakieś X. Wymaga to jednak znajomości skali tych popraw dla naszego problemu. Nie wymaga tego natomiast **minimalna liczba przykładów** - nie dokonujemy podziału, jeżeli w którymkolwiek liściu byłoby mniej niż N przykładów.

Najprościej jest zdefiniować **maksymalną głębokość drzewa** - dokonujemy co najwyżej N podziałów. Można też analogicznie zdefiniować **maksymalną liczbę liści**.

Mogą zrobić bardziej wyrafinowany schemat regularyzacji, czyli najpierw robimy pełne drzewo, a potem je **przycinamy (pruning)**. Realizuje się tutaj **cost-complexity pruning**, który balansuje jakość (zawsze wyższa przy bardziej rozbudowanym drzewie) kontra liczbę parametrów (wielkość drzewa), zgodnie z pewnym współczynnikiem siły regularyzacji.

### **Zalety i wady drzew**

Największą zaletą jest **prostota**, z której wynika też **bardzo wysoka interpretowalność**. Drzewa to proste reguły if/else połączone koniunkcją (schodzenie po drzewie robi AND), więc każdy jest w stanie je zinterpretować. Łatwo też sprawdzić same reguły i punkty podziału. Drzewo można też narysować i pokazać osobom nietechnicznym.

Mamy też miarę relatywną **ważności atrybutów** - wyżej w drzewie (wcześniej) są wybierane lepsze atrybuty, w sensie lepszego rozróżniania klas. Jest to też **selekcja cech**, bo cechy słabo rozróżniające klasy w ogóle nie zostaną wybrane.

**Obsługują różne typy danych**, w tym typy kategoryczne, nie wymagają też żadnego preprocessingu wartości. Zaletą jest też **obliczanie prawdopodobieństw** dla klasyfikacji.

Wady to **duża wariancja i overfitting** - nawet z regularyzacją jest ciężko, a na dodatek sama regularyzacja jest nietrywialna, jest to dużo hiperparametrów wpływających mocno na siebie nawzajem. Są też **czułe na niezbalansowanie klas** - metryki będą preferowały większe klasy przy optymalizacji (choć można temu przeciwdziałać ważeniem klas). W związku z tym **drzewa dają przeciętnie dobre wyniki**.

W przypadku regresji obliczana jest wartość średnia w liściu. Wynika z tego, że **drzewa nie potrafią ekstrapolować** - nigdy nie przewidzą wartości poza minimum i maksimum z danych treningowych.

Wadą typowych metryk jest też **preferowanie atrybutów o dużej liczbie unikalnych wartości**, chociaż można to poprawiać specjalnymi metrykami (np. gain ratio).

## Drzewa decyzyjne jako interpretatory

Najprościej można użyć **pojedynczego drzewa do interpretacji** - być może inne algorytmy dadzą lepsze wyniki, ale ludzie z drzewa decyzyjnego wyciągną więcej wiedzy, która może się też przydać przy tuningu innych algorytmów.

Ciekawą opcją jest też **użycie drzewa jako modelu surogatowego (surrogate model)**, czyli trening drzewa na wynikach przewidywanych przez inny klasyfikator. Polega to na tym, że wektor klas/celów regresji zastępujemy tym, co przewiduje dla tych przykładów bardziej złożony algorytm i na tym trenujemy drzewo decyzyjne. Wyjaśnia ono wtedy ten klasyfikator i jego predykcje - oczywiście z pewną dokładnością (wtedy metryki wyników drzewa dotyczą tego, jak dobrze nauczyło się ono wyjaśnianego modelu).

## Lasy losowe

**Lasy losowe (random forests)** to zbiory (ensembles) drzew decyzyjnych, typowo minimum 100 drzew.

Każde z nich jest uczone na **próbce bootstrapowej (bootstrap sample)** ze zbioru, czyli dla zbioru uczącego rozmiaru N losujemy ze zwracaniem N punktów i na tym uczymy drzewo. Będą powtórzenia, nawet sporo, ale się nie przejmujemy.

Dodatkowo przy każdym podziale rozważamy **losową podprzestrzeń (random subspace)**, czyli wybieramy losowo podzbior cech do rozważenia przy danym podziale, typowo  $\text{sqrt}(D)$  dla klasyfikacji i  $D/3$  dla regresji (dla D wymiarów/cech). Wymusza to eksplorację dostępnych cech przez drzewa.

Wynik zwracany przez las losowy to średni wynik dla poszczególnych drzew: dla klasyfikacji drzewa głosują i zwracamy najpopularniejszą klasę (można też zwrócić ich rozkład prawdopodobieństwa), dla regresji wartość średnią.

Las losowy ma **zmniejszoną wariancję** względem pojedynczych drzew decyzyjnych dzięki temu uśrednianiu, jest też więc dobrze **odporny na szum w danych**. **Nie można overfittować przez zwiększenie liczby drzew** - wynik będzie się polepszał, aż do pewnej asymptoty. Świeście się też uwspółbieźnia jego trening. Jest bardzo dobrym wyborem na **domyślny klasyfikator**, bo nie wymaga szczególnego tuningu hiperparametrów.

Z lasu losowego można dostać **duże lepsze oszacowanie ważności cech** niż z pojedynczego drzewa. Wynika to z tego, że wagę cech uśredniamy po wielu drzewach, które dzięki losowaniu podprzestrzeni są też zmuszane do eksploracji i sprawdzania różnych kombinacji cech.

Lasy losowe ze względu na uśrednianie mają **obciążone (biased) estymaty prawdopodobieństwa** - są słabo skalibrowane, bo ciągną w stronę wartości średnich, rzadko kiedy będą wyniki blisko 0% lub 100%. Ten sam problem tyczy się regresji - jest mocna tendencja do przewidywania uśrednionych wartości.

Są pod pewnymi względami **mniej interpretowalne od drzew**, bo nie da się takiego lasu narysować ani prześledzić pojedynczych decyzji.

# SVM

## SVM - klasyfikacja

Idea:

- oddzielmy klasy hiperpłaszczyzną
- dajmy sobie jak największy **margines** dla bezpieczeństwa
- hiperpłaszczyzna ma być jak najbardziej odległa od punktów, żeby zmaksymalizować margines
- typowo klasy nie są **separowalne liniowo**, więc dodatkowo dodajmy regularyzację, tak że pewna ilość punktów może się znaleźć po zlej stronie granicy decyzyjnej

SVM dla klasyfikacji rozwiązuje problem optymalizacji:

$$\text{minimize } \frac{1}{n} \sum_{i=1}^n \zeta_i + \lambda \|\mathbf{w}\|^2$$

subject to  $y_i(\mathbf{w}^T \mathbf{x}_i - b) \geq 1 - \zeta_i$  and  $\zeta_i \geq 0$ , for all  $i$

$$\zeta_i = \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}_i - b))$$

$\zeta$  to tzw. slack variables, każdy przykład uczący dostaje 1 taką zmienną. Odpowiadają one hinge loss, czyli funkcji kosztu dla każdego przykładu, której wartość to 0 (gdy jest po dobrej stronie marginesu) lub odległość od marginesu (jeżeli jest po zlej stronie). W związku z tym wartość slack variables jest nieujemna (w końcu to odległość)

Chcemy minimalizować średni hinge loss, dodając do tego czynnik regularyzujący (regularyzacja L2).

Warto zauważyć, że jest to metoda nieparametryczna, czyli koszt rośnie bezpośrednio wraz z liczbą punktów w zbiorze uczącym. Jednak margines zależy tylko od **wektorów wspierających (support vectors)**, czyli punktów przy granicy decyzyjnej - tylko one faktycznie wpływają na ułożenie hiperpłaszczyzny.

Powyższe sformułowanie - **primal problem** - jest fajne, ale mało użyteczne w praktyce, bo jest trudne w optymalizacji. Zamiast tego korzystamy z teorii optymalizacji matematycznej i robimy **problem dualny (dual problem)**:

$$\text{maximize } f(c_1 \dots c_n) = \sum_{i=1}^n c_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i c_i (\mathbf{x}_i^T \mathbf{x}_j) y_j c_j,$$

subject to  $\sum_{i=1}^n c_i y_i = 0$ , and  $0 \leq c_i \leq \frac{1}{2n\lambda}$  for all  $i$ .

Dodatkowo wielką zaletą jest tutaj to, że mamy wyróżniony w nawiasie iloczyn skalarny pomiędzy punktami. Iloczyn skalarny to najprostsza, liniowa funkcja jądra (linear kernel). Możemy ją zamienić na dowolną inną, nazywa się to **trikiem kernelowym (kernel trick)**:

$$\begin{aligned} \text{maximize } f(c_1 \dots c_n) &= \sum_{i=1}^n c_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i c_i (\varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_j)) y_j c_j \\ &= \sum_{i=1}^n c_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i c_i k(\mathbf{x}_i, \mathbf{x}_j) y_j c_j \end{aligned}$$

subject to  $\sum_{i=1}^n c_i y_i = 0$ , and  $0 \leq c_i \leq \frac{1}{2n\lambda}$  for all  $i$ .

To sformułowanie pozwala nam dokonać **implicite nieliniowego przekształcenia przestrzeni** i uwzględnienia nieliniowych interakcji między cechami. Przy okazji stąd też wynika potrzeba standaryzacji danych dla SVMów - cechy muszą mieć tę samą skalę, żeby dało się je porównywać funkcjami jądra.

Przy czym uwaga - SVM to zawsze klasyfikator liniowy! Po prostu hiperpłaszczyzna separująca klasy jest tworzona implicite w wysokowymiarowej przestrzeni wyznaczanej przez kernel. Po zrzutowaniu do niskowymiarowej przestrzeni daje to nieliniową granicę decyzyjną, ale przy klasyfikacji nowych przykładów znowu trzeba użyć kernela i przejść do przestrzeni wysokowymiarowej, bo tylko tam klasy są separowalne liniowo.

Jak rozwiązać w praktyce taki problem optymalizacji? Jest to problem programowania kwadratowego, do którego są efektywne solwery matematyczne. Ponadto został stworzony specjalny **algorytm SMO (Sequential Minimal Optimization)**, który robi to szczególnie szybko dla SVMów. Dalej jednak to minimalna złożoność obliczeniowa treningu to  $\mathcal{O}(n^2)$  ze względu na potrzebę liczenia odległości punkt-punkt.

## SVM - regresja

Dla regresji sformułowanie wygląda nieco inaczej:

$$\begin{aligned} \text{minimize } & \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{\ell} (\xi_i + \xi_i^*) \\ \text{subject to } & \begin{cases} y_i - \langle w, x_i \rangle - b \leq \varepsilon + \xi_i \\ \langle w, x_i \rangle + b - y_i \leq \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases} \end{aligned}$$

Idea jest tutaj taka, że chcemy znaleźć funkcję, jak najbardziej płaską (stąd minimalizujemy kwadrat normy wektora wag), gdzie żadna wartość regresji nie znajduje się dalej niż  $\xi$  niż

powierzchnia funkcji. Tak długo, jak wartości są w odległości mniejszej lub równej  $\xi$ , to jesteśmy zadowoleni:

$$|\xi|_\varepsilon := \begin{cases} 0 & \text{if } |\xi| \leq \varepsilon \\ |\xi| - \varepsilon & \text{otherwise} \end{cases}$$

Ponownie robimy sformułowanie dualne, znowu mamy kernel trick, dzięki czemu znowu mamy arbitralne nieliniowe zależności:

$$\begin{aligned} \text{maximize} \quad & \left\{ \begin{array}{l} -\frac{1}{2} \sum_{i,j=1}^{\ell} (\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*) \langle x_i, x_j \rangle \\ -\varepsilon \sum_{i=1}^{\ell} (\alpha_i + \alpha_i^*) + \sum_{i=1}^{\ell} y_i (\alpha_i - \alpha_i^*) \end{array} \right. \\ \text{subject to} \quad & \sum_{i=1}^{\ell} (\alpha_i - \alpha_i^*) = 0 \quad \text{and} \quad \alpha_i, \alpha_i^* \in [0, C] \end{aligned}$$

## Zalety i wady SVMów

SVM daje typowo **bardzo dobre wyniki**, szczególnie gdy feature engineering nie jest zbyt dobry - kernel sam wyciągnie odpowiednie interakcje cech. Ma też **niewiele hiperparametrów** - typowo silę regularyzacji C, kernel i jakiś parametr kernela, np. szerokość dla kernela Gaussowskiego.

Problemem jest **duża złożoność obliczeniowa**. Ponadto **słabo radzą sobie z klasami niebalansowanymi**, ale można dodać wagi klas.

## SVM a interpretowalność

SVM jest **metodą słabo interpretowalną**. Ich wektor wag to po prostu współczynniki hiperpłaszczyzny, ale sama hiperpłaszczyzna jest w wysokowymiarowej przestrzeni (potencjalnie nawet nieskończonymi) dla kernela Gaussowskiego), więc wyobrażenie sobie tego jest niemożliwe.

Natomiast jak użyje się liniowego SVMa (z domyślnym kernelem liniowym) i weźmie się wagi jako **ranking ważności cech**, to jest zaskakująco dobry. Działa to nawet dla przypadku, kiedy dane testowe są z trochę innych rozkładów.

Ważne jednak, że SVM jest rzadki - wagi tylko dla wektorów wspierających są niezerowe. Istnieją jednak algorytmy **ekstrakcji zasad (rule extraction)** z wysokowymiarowych reprezentacji kernelowych SVMów, oparte o wektory wspierające i klasteryzację.

# Metody kompresji sieci neuronowych

## Metody oparte o rozkład macierzy

**Singular Value Decomposition** - metoda liniowej redukcji wymiarowości. Dekomponujemy macierz  $W$  na iloczyn 3 macierzy:

$$W = USV^T, W \in \mathbb{R}^{m \times k}, U \in \mathbb{R}^{m \times m}, S \in \mathbb{R}^{m \times k}, V^T \in \mathbb{R}^{k \times k}$$

$U, V$  - macierze ortogonalne

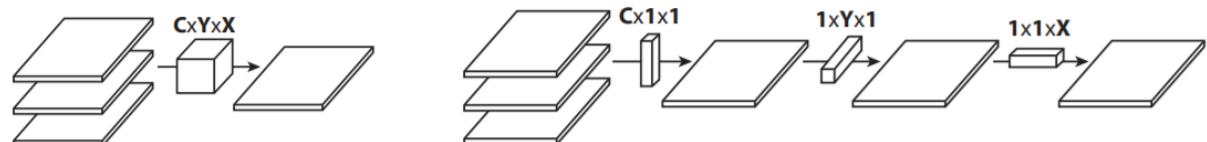
$S$  - macierz przekątniowa

Aby dokonać kompresji, wybiera się pierwsze  $t$  wektorów z  $U$  i  $V$ , lewy górny blok  $t \times t$  z macierzy  $S$  i mnoży, wychodzi mniejsza macierz:

$$\tilde{W} = \tilde{U}\tilde{S}\tilde{V}^T, \tilde{W} \in \mathbb{R}^{m \times k}, \tilde{U} \in \mathbb{R}^{m \times t}, \tilde{S} \in \mathbb{R}^{t \times t}, \tilde{V}^T \in \mathbb{R}^{t \times k}$$

Jakie konsekwencje ma taka operacja? Kompresja **zwiększa błąd**, ale minimalnie - można zmniejszyć sieć 3.5x, a błąd rośnie o 0.09%. Za to mamy **znacznie mniejsze wymagania pamięciowe**, a do tego późniejsze wykorzystanie takiej sieci **jest znacznie szybsze**, bo jest po prostu mniejsza. Ma to duże znaczenie np. w edge ML. Można też potraktować taką operację jak **regularyzację**, bo mamy mniej parametrów, więc zmniejszamy pojemność (capacity) modelu, więc go regularyzujemy.

Alternatywnie zamiast kompresować całą sieć rozkładem macierzowym można skupić się na rozkładzie samych warstw konwolucyjnych (zajmują najczęściej czasu). Dostajemy **flattened convolutions**, czyli zamiast pełnej konwolucji 3D robimy konwolucje 1D po poszczególnych wymiarach:



(a) 3D convolution

(b) 1D convolutions over different directions

Taki model ma mniejszą siłę wyrazu, bo nie każdy filtr 3D da się zdekomponować w taki sposób. Ale na szczęście większość użytecznych i ważnych filtrów się da, więc chociaż taka dekompozycja nie działa uniwersalnie, to spadki jakości klasyfikacji są akceptowalne (2-3%), a za to jest o wiele szybciej.

Oryginalna operacja konwolucji to:

$$F_f(x, y) = I * W_f = \sum_{c=1}^C \sum_{x'=1}^X \sum_{y'=1}^Y I(c, x - x', y - y') W_f(c, x', y')$$

Czyli mamy XYC parametrów na warstwę.

W wersji zdekomponowanej mamy 3 osobne, małe macierze wag:

$$\hat{F}_f(x, y) = I * \hat{W}_f = \sum_{x'=1}^X \left( \sum_{y'=1}^Y \left( \sum_{c=1}^C I(c, x - x', y - y') \alpha_f(c) \right) \beta_f(x') \right) \gamma_f(y')$$

Teraz jest w sumie X+Y+C parametrów.

Zysk to **szybkość treningu, szybkość inferencji** oraz **oszczędność pamięciowa**.

## Metody wycinania wag

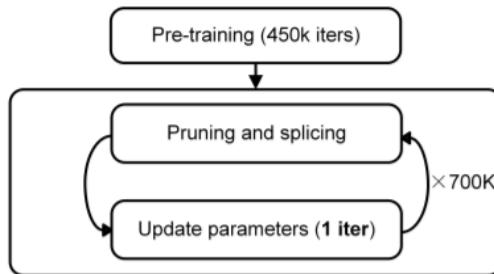
Istnieje szereg metod **wycinania wag (weight pruning)** z sieci neuronowych. Redukuje to liczbę parametrów.

Pierwsza to jednoczesny trening wag i w ogóle samych połączeń, tak, że uczymy się rzadziej połączonej sieci. **Metoda magnitude-based** polega na iteracyjnym pruningu i retrenowaniu sieci:

1. Wybierz architekturę, wytrenuj sieć, aż dostaniesz satysfakcyjujący wynik.
2. Usuń połączenia o wielkości wagi (magnitude, wartość bezwzględna) mniejszej niż próg  $\tau$ .
3. Dotrenuj sieć, aż dostaniesz satysfakcyjujący wynik.
4. Powtarzaj punkty 3-4, póki jakość predykcji nie spada.

Taka metoda jest **czasochłonna**, ale za to nie wymaga żadnych modyfikacji samej sieci - trenujemy ją zwyczajnie w kroku 1, możemy nawet wziąć pretrenowaną sieć i od niej zacząć. Potrafi to nawet **zmniejszyć błąd**, bo uczymy się, które połączenia są słabe i je usuwamy, a to one mogą wprowadzać szum do predykcji. Można skompresować tak sieć 9-13 razy przy porównywalnych albo ciut lepszych wynikach.

Alternatywną metodą jest **pruning z poprawką (pruning with rehabilitation)** tudzież **dynamic network surgery**, które stanowi bardziej wyrafinowaną wersję poprzedniej metody. Początek jest taki sam - bierzemy sieć i trenujemy. Różnica tkwi w tym, że tutaj zamiast brać jakiś z góry ustalony próg, **uczymy się** przycinania wag. Idea jest taka, że małe wagi wcale koniecznie nie są złe - sieć neuronowa to złożona rzecz, lepiej żeby algorytm sam nauczył się z danych, co najlepiej wyeliminować.



Uczymy się minimalizować błąd prunningu:

$$\min_{W_k, T_k} L(W_k \odot T_k) \quad s.t. \quad T_k^{(i,j)} = h_k(W_k^{(i,j)}), \forall (i,j) \in \mathfrak{T}$$

$W_k$  - macierz wag sieci w k-tej warstwie

$T_k$  - macierz indykatorowa (zera lub jedynki) w k-tej warstwie

Macierzy  $T_k$  uczymy się przez nauczenie się macierzy  $W_k$ , co robimy zwykłą propagacją wsteczną, a potem wrzuceniem wyników do funkcji:

$$h_k(W_k^{(i,j)}) = \begin{cases} 0 & \text{if } a_k > |W_k^{(i,j)}| \\ T_k^{(i,j)} & \text{if } a_k \leq |W_k^{(i,j)}| < b_k \\ 1 & \text{if } b_k \leq |W_k^{(i,j)}| \end{cases}$$

$a_k$  - pruning threshold, hiperparametr

$b_k$  - próg pozostawienia,

Pomiędzy  $a_k$  i  $b_k$  jest zakres pozostawiający aktualną wartość z macierzy  $T_k$

Co ciekawe w miarę nauki w kolejnych iteracjach macierz  $W_k$  się zmienia, więc **usunięta waga może zostać przywrócona**. Dlatego też ta metoda jest bardziej adaptacyjna od magnitude-based.

Precyzyjniejszą, ale znacznie mniej wydajną obliczeniowo i starszą metodą jest **Diagonal Hessian-based method** tudzież **optimal brain damage**. Idea jest taka, żeby zdefiniować **nasycenie (saliency)** dla parametrów, czyli wpływ konkretnego parametru na błąd treningowy. Pozwala to precyzyjnie oszacować, których wag się najlepiej pozbyć.

Jak zdefiniować saliency? Zauważmy, że błąd treningowy można przybliżyć szeregiem Taylora:

$$\delta E = \sum_i \frac{\partial E}{\partial u_i} \delta u_i + \frac{1}{2} \sum_i \frac{\partial^2 E}{\partial^2 u_i} \delta^2 u_i + \frac{1}{2} \sum_i \frac{\partial^2 E}{\partial u_i \partial u_j} \delta u_i \delta u_j + O(\|\delta U\|^3)$$

Podczas treningu backpropagation używamy gradientów i dla dobrze wytrenowanej sieci można założyć, że pierwszy człon to zero. Ignorujemy dalsze elementy, przybliżamy w ten sposób i zostaje nam:

$$\delta E = \frac{1}{2} \sum_i \frac{\partial^2 E}{\partial^2 u_i} \delta^2 u_i$$

Jak widać będą to elementy na diagonali Hesjanu, bo mamy same czyste drugie pochodne.

Wykorzystując tę definicję mamy algorytm:

1. Wybierz architekturę.
2. Wytrenuj sieć, aż dostaniesz satysfakcyjny wynik.
3. Oblicz wartości drugiej pochodnej dla każdego parametru.
4. Oblicz saliency dla każdego parametru:

$$S_k = \frac{\partial^2 E}{\partial^2 u_k} u_k^2$$

5. Posortuj parametry po saliency, wytnij parametry o najniższym saliency.
6. Powtórz kroki 2-4

Typowo taka metoda daje **wyniki o wiele lepsze od metody magnitude-based**. Jest natomiast o wiele kosztowniejsza obliczeniowo.

Optimal brain damage uszkadza sieć dość brutalnie, bo robi spore przybliżenia. Metoda **Hessian-based**, czyli **optimal brain surgeon** jest podobna, tylko zostawiamy pochodne częściowe mieszane. Poza tym jest identycznie.

Przybliżenie jest lepsze, więc mamy **lepsze wyniki** i bezpośrednią aktualizację wag z minimalną zmianą funkcji celu. Za to mamy **większy koszt obliczeniowy**.

Ostatnią metodą jest **Neural Net partitioning**. Jest on oparty na obserwacji, że kiedy mamy sprune'owaną sieć, to można ją potraktować jak ensemble mniejszych sieci działających równolegle i zbierających swoje wyniki na koniec.

Skoro tak, to możemy najpierw wytrenować zbiór kilku mniejszych sieci, a potem dołożyć łączącą je warstwę na koniec i ją dotrenować.

Ciekawostka - to podejście zostało zaprezentowane w pracy magisterskiej "SuperNet - An efficient method of neural networks ensembling", której prof. Dzwinel był promotorem.

## Metody kwantyzacji sieci

**Kwantyzacja** - podział wartości ciągłych na dyskretne fragmenty, kwanty. **Kwantyzacja wag sieci neuronowych** opiera się na obserwacji, że ML to nie obliczenia numeryczne i tu nie trzeba wielkiej precyzyji i pełnego zakresu floata, a jak zmniejszymy zakres i będzie szybciej, to wyniki nie ucierpią. Jest to szczególnie ważne dla edge ML, gdzie robi się nawet wagi całkowitoliczbowe albo wręcz binarne.

Dziedzina **kwantyzacji wektorów (vector quantization, VQ)** jest stara, bogata i rozbudowana, więc dużo zostało po prostu zaadaptowane do sieci neuronowych. Pierwszy i najprostszy algorytm to po prostu zaokrąglenie wag i użycie **liczb stałoprzecinkowych**, są tu 2 warianty:

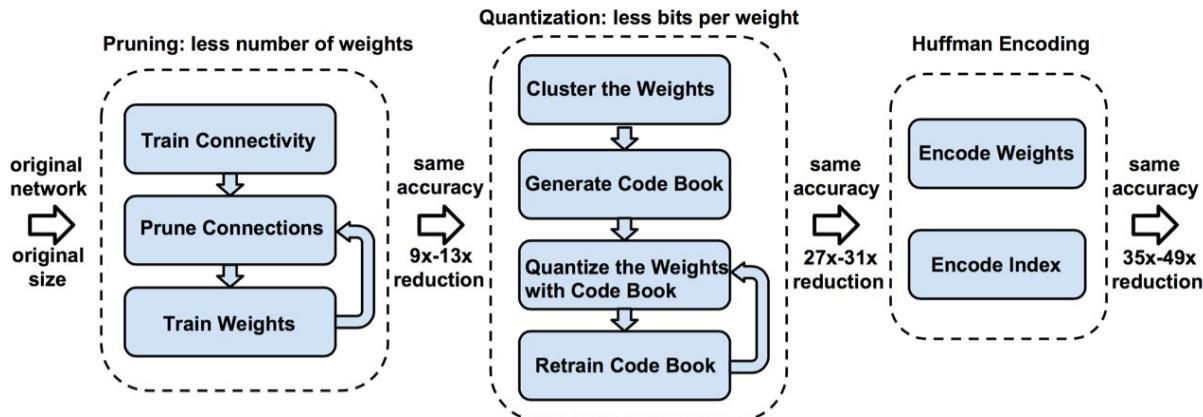
- **round-to-nearest** - po prostu zaokrąglamy matematycznie do zadanej liczby bitów po przecinku
- **stochastic rounding** - nieobciążone (unbiased), zaokrąglamy probabilistycznie, typowo daje dużo lepsze wyniki:

$$Round(x, \langle IL, FL \rangle) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{w.p. } \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases}$$

Klasyczne VQ korzysta z algorytmu k-means do sklastrowania najpierw danych, a potem zastąpienia każdego wektora (punktu) centroidem z jego klastra. Takie podejście nazywa się **code book**, bo wyuczona klasteryzacja działa jak słownik wektor -> kod (centroid).

Ekstremalna forma kwantyzacji to **binaryzacja**, czyli zamiana wszystkich wag na -1 lub +1. Można to najłatwiej zrobić kodowaniem Huffmana. Nie jest to jednak takie proste i wymaga wielokrotnego pipeline'u, żeby zachować dokładność sieci przy tak ostrej kompresji:

1. Wybieramy architekturę.
2. Trenujemy sieć do uzyskania zadowalających wyników.
3. Pruning - używamy typowo jakiejś metody magnitude-based
4. Kwantyzacja z k-means
5. Binaryzacja kodowaniem Huffmana

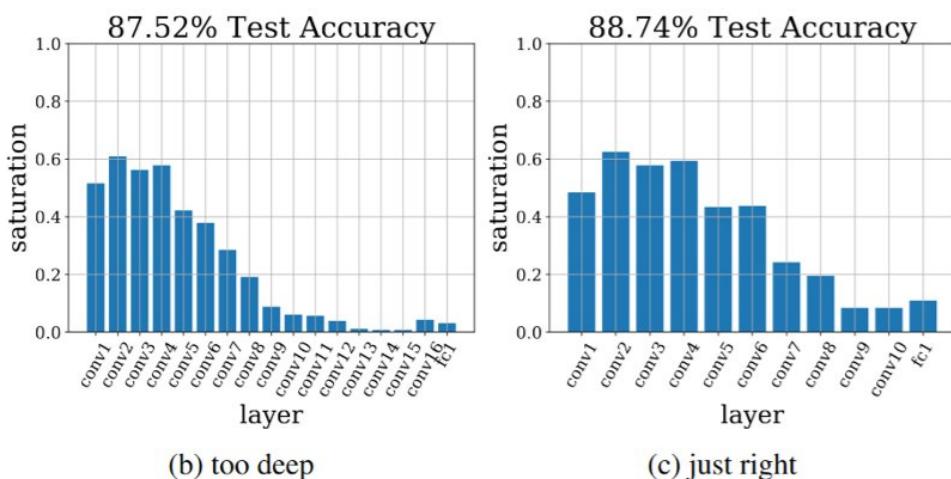


W wyniku otrzymujemy **ekstremalnie skompresowaną sieć dającą podobne wyniki**. Dodatkowo w praktycznym zastosowaniu taka sieć jest **wydajna energetycznie**, co w edge MLu (np. komórki, IoT) ma duże znaczenie.

## Layers pruning, saturation

**Obcinanie warstw (layer pruning)** polega na analizie całych warstw sieci i usuwaniu całych warstw, typowo ostatnich. Jest to oparte na obserwacji, że sieci są zbyt głębokie - zależności w danych nie są aż tak złożone, nie ma tam aż tak ciekawych hierarchicznych i nieliniowych zależności, żeby kolejne warstwy coś realnie dawały, zamiast tego robią tylko szum.

Jak to wykryć? Można badać **saturację (saturation)** poszczególnych warstw, czyli jak bardzo aktywne są dane warstwy w przekształcaniu danych. Saturacja warstwy jest wysoka, gdy prowadzi do złożonych i nieliniowych zmian w danych, które dostaje na wejście. Typowo saturacja maleje wraz z kolejnymi warstwami. Można z tego zrobić wykres i odciąć warstwy, w których saturacja już ostro zmalała.



## Destylacja wiedzy

**Destylacja wiedzy (knowledge distillation)** polega na wyekstrahowaniu kluczowej wiedzy z dużego i ciężkiego modelu, typowo sieci neuronowej ("wydestylowaniu" informacji) i wrzucenia jej do mniejszego modelu.

Trenujemy nową sieć - **ucznia (student)**, typowo sporo mniejszą od poprzedniej, korzystając z tego samego zbioru danych. Wykorzystujemy jednak **predykcje oryginalnej sieci**, bo oryginalna sieć **nauczyciel (teacher)** na wyjściu zwraca cały rozkład prawdopodobieństwa. Mamy więc złożoną funkcję kosztu:

$$L = L_{hard} + T^2 L_{soft}$$

$L_{hard}$  - "twardy" koszt, różnica predykcji modelu względem prawdziwej klasy, znane ze zbioru

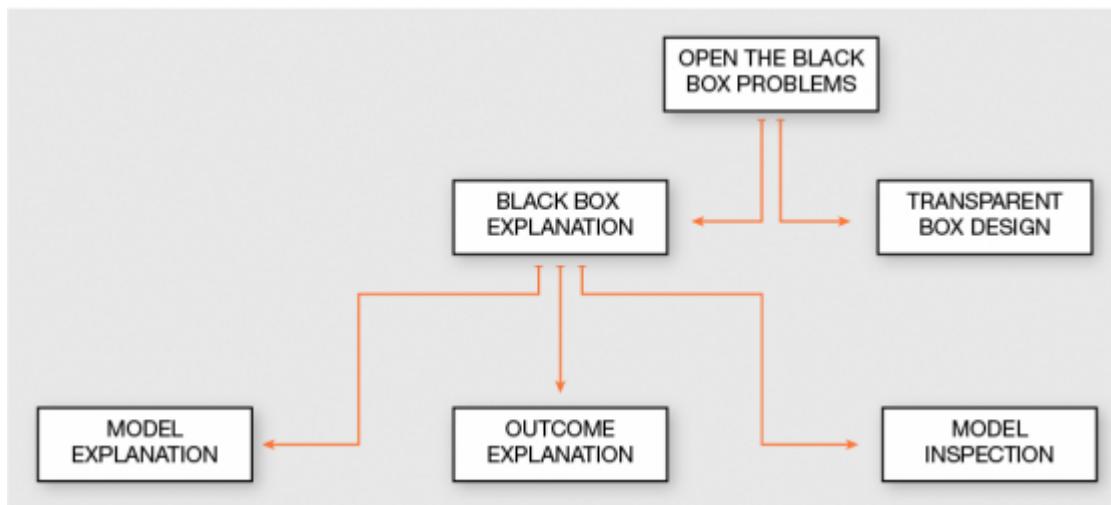
$L_{soft}$  - "miękkii" koszt, różnica predykcji modelu i predykcji nauczyciela, czyli cross-entropy między zwracanymi przez nie rozkładami prawdopodobieństwa klas  
 $T$  - temperatura, współczynnik jak bardzo interesuje nas prawda a jak nauczyciel, typowo ustawia się po prostu na 1

Można w ten sposób osiągnąć **wyniki praktycznie identyczne z nauczycielem**, a przy tym **drastycznie zmniejszyć czas predykcji**.

Szczególnym wariantem jest wytrenowanie ucznia o dokładnie takiej samej architekturze jak nauczyciel. W takich wypadkach **wydestylowana sieć często radzi sobie lepiej** i to rzędu kilku % lepiej. Hipoteza jest taka, że dodajemy w ten sposób informacje wynikające z treningu nauczyciela, wyciągnięte mimośrodem podczas treningu, tzw. **dark knowledge**. Oryginalna sieć daje na wyjście rozkład z softmaxu, czyli pewność co do klas, więc uczymy się dzięki temu podobieństwa klas do siebie, czego nie ma w oryginalnym zbiorze. Przykładowo, jeżeli model zwraca "80% cat, 20% dog", a klasa to "cat", to ucząc się, druga sieć dostaje implicitę informację "koty są podobne do psów".

# Otwieranie czarnej skrzynki

## Taksonomia problematyki otwierania czarnej skrzynki



Pierwszy podział to podział ze względu na ogólną interpretowalność:

- **transparent box** - w ogóle nie robimy czarnej skrzynki, tylko od razu robimy model interpretowalny, unikając problemu (ale typowo np. mając niższe wyniki), np. drzewo decyzyjne
- **black box explanation** - budujemy złożony model nieinterpretowalny, czyli czarną skrzynkę i próbujemy ją wyjaśniać, np. sieć neuronowa

Metody wyjaśniania czarnych skrzynek można podzielić na 3 grupy:

- **model explanation** - wyjaśniamy całe zachowanie złożonego modelu jako takiego, logiki sterujące jego ogólnym zachowaniem, np. rule extraction
- **outcome explanation** - wyjaśniamy wynik czarnej skrzynki dla konkretnej obserwacji, np. SHAP
- **model inspection** - pewna forma pośrednia między powyższymi, metody wyjaśniające jak czarna skrzynka zachowuje się przy określonych zmianach wejścia, np. sensitivity analysis, partial dependency plots

Jaka jest różnica między model explanation a inspection? Explanation wymaga stworzenia interpretowalnego globalnego explainera, który wyjaśni ogółem zachowanie modelu. Inspection natomiast koncentruje się na analizie modelu z perspektywy konkretnych własności, np. czułości na zmianę pewnego atrybutu, więc wymaga mniej ogólnych metod.

Można też wprowadzić drugą taksonomię, dzielącą techniki interpretowalności w 2 osiach: **specyfika modelu, zakres interpretacji**.

Podział ze względu na specyfikę modelu:

- **model specific** - działają dla konkretnych modeli lub klas modeli, mogą wykorzystywać wiedzę o ich konstrukcji i działaniu, wchodzą do wnętrza modelu, np. mnóstwo metod dla sieci neuronowych, np. feature visualization, pixel attribution
- **model agnostic** - działają dla dowolnych modeli, nie wchodzą do wnętrza modelu, np. SHAP, LIME

Podział ze względu na zakres interpretacji modelu:

- **local** - wyjaśnia się lokalnie predykcje modelu, dla konkretnych predykcji, w specyficznych przypadkach, np. LIME
- **global** - wyjaśnia się globalne zachowanie modelu, ogólną logikę, np. SAGE (Shapley Additive Global importancE)

Metody **model specific** realizują **mechanistic understanding**, czyli próbujemy zrozumieć, jaki mechanizm w środku modelu sprawia, że model w taki sposób rozwiązuje problem i dokonuje określonej predykcji. Jest to więc niski poziom abstrakcji, poznanie szczegółów w środku modelu.

W podejściu **model agnostic** mamy **functional understanding**, więc dostajemy zrozumienie modelu jako funkcji mapującej wejścia na wyjścia. Jest to bardziej ogólna metoda, ale nie wymaga znajomości tego, jak model dokładnie realizuje taką funkcję w środku.

Podejście lokalne vs globalne dobrze widać przy algorytmach model agnostic (functional understanding). Podejścia lokalne to **decision analysis** i wyjaśniają, co wpłynęło na konkretną decyzję modelu, czemu dane wartości wejścia dały takie wyjście funkcji, którą jest model. Natomiast globalnie mamy **model analysis** i chcemy opisać zmiany całej funkcji, którą jest model (jej kształtu) w zależności od całości danych.

## Metody outcome explanation dla sieci konwolucyjnych

Dla sieci neuronowych bardzo ciężko coś interpretować bardziej globalnie, bo działają na obrazach (ciężko wizualizować jakieś ogólne zależności), poza tym interesuje nas bardziej pytanie "na co sieć patrzy na obrazku", czyli **wyjaśnianie wyniku**.

**Layer-wise relevance propagation (LRP)** analizuje predykcję siedzi, idąc od końca do początku. Dla każdego piksela przypisywany jest relevance score, który jest pozytywny, jeżeli dany piksel wspiera daną klasę, a negatywny, jeżeli wspiera inną klasę:

$$f(x) \approx \sum_p R_p^{(1)}$$

Taka dekompozycja wyników dla każdego piksela jest możliwa na bardzo wiele sposobów. Pozwala to narysować **heatmapę** dla klasy. Z taką propagacją idzie się od końca sieci do

początku i nanosi się wyniki na wejściowy obraz. Takie metody nazywa się też **pixel attribution**, bo próbują wyjaśniać piksel po pikselu.

Typowa metoda tej grupy to **Grad-CAM**, czyli propagacja gradientów od końca zgodnie z powyższą zasadą. Waży on aktywacje 2D zgodnie ze średnią wartością gradientu.

**Activation maximization** wyjaśnia **prototyp** dla danej klasy, czyli jak sieć wyobraża sobie taki archetypiczny, typowy przykład dla danej klasy. Wynik tej metody to dość abstrakcyjny obraz tworzony na podstawie aktywacji neuronów maksymalizujących prawdopodobieństwo danej klasy.

## Metody inspekcji black boxu "od środka"

Główne chodzi tu o **wizualizację powierzchni funkcji kosztu** dla sieci neuronowych. Oczywiście funkcja jest bardzo wysokowymiarowa, więc wizualizacja (w 3D) wymaga znacznej redukcji wymiarowości i jest tylko pewnym przybliżeniem. Pozwala to jednak zauważać, jak bardzo problematyczne jest zastosowanie danej sieci do danego problemu.

Po co nam to? Na przykład pozwala **badać łatwość treningu sieci** - im bardziej wypukła funkcja kosztu i im mniej minimów lokalnych ma, tym prościej ją trenować. Można w ten sposób sprawdzić, czy dodanie/usunięcie jakiegoś elementu pomaga, np. skip connections w sieci ResNet albo gęsta sieć dodatkowych połączeń w DenseNet.

## Metody globalne

Są to 2 główne metody: **modele surogatowe (surrogate models)** i **destylacja wiedzy (knowledge distillation)**. Oba były opisywane wcześniej, więc tutaj tylko skrótowe przypomnienie.

Model surogatowy to prosty, interpretowalny model uczony tylko na predykcjach modelu czarnej skrzynki. Przykładowo, robimy predykcje używając SVMa dla całego zbioru i na jego przewidywanych klasach trenujemy drzewo decyzyjne. Pozwala to zrobić np. rule extraction z dowolnie złożonego modelu. Model uczony typowo nie jest wykorzystywany do predykacji, tylko do interpretacji. Metryką jakości jest tutaj jakość odwzorowania predykcji wyjaśnianego modelu.

W destylacji wiedzy uczymy model i na zbiorze, i predykcjach nauczyciela. Metryką jakości podczas treningu jest ważona funkcja kosztu, uwzględniająca i wynik na zbiorze, i odwzorowanie predykcji złożonego modelu. Model ucznia jest potem wykorzystywany w praktyce, bo jest mniejszy, a więc ma mniejszy koszt obliczeniowy i pamięciowy.

## LIME

**Locally Interpretable Model-agnostic Explanations (LIME)** to algorytm interpretacji. Jest, zgodnie z nazwą, model agnostic i local. Można go więc zastosować dla dowolnego algorytmu i wyjaśniać konkretne predykcje. Działa dla danych tabelarycznych, obrazów i tekstu.

LIME został zaprojektowany tak, aby miał 4 własności:

- **interpretowalny** - wyniki mają być ładnie zaprezentowane, łatwe w zrozumieniu, niekoniecznie oddawać model bardzo bezpośrednio (bo to może wymagać sporo wiedzy technicznej o modelu)
- **lokalnie wiernie przybliżać model (local fidelity)** - bardzo ciężko jest zrobić globalny interpreter tak, żeby dobrze oddawał model, natomiast chcemy przynajmniej, żeby dla konkretnych przykładów i ich otoczenia (podobnych sytuacji) interpretator bardzo dobrze oddawał działanie modelu
- **model agnostic** - większość najlepszych i najciekawszych modeli jest bardzo złożona, różnorodna i to właśnie one potrzebują takich interpretatorów
- **zapewnianie perspektywy globalnej** - nawet jak nie jesteśmy w stanie zapewnić globalnej interpretacji, to chcemy przynajmniej wybrać kilka reprezentatywnych przykładów tak, aby wyjaśnić działanie modelu w wybranych, ciekawych sytuacjach

LIME definiuje swój cel jako **minimalizację sumy różnicy interpretatora od danych i złożoności interpretatora**:

$$\xi(x) = \underset{g \in G}{\operatorname{argmin}} \quad \mathcal{L}(f, g, \pi_x) + \Omega(g)$$

$\mathcal{L}(f, g, \pi_x)$  - loss, różnica między predykcjami modelu  $f$  oraz interpretatora  $g$  dla otoczenia  $\pi_x$  punktu  $x$

$\Omega(g)$  - złożoność interpretatora  $g$

Chcemy żeby model był jak najwierniejszy lokalnie (w otoczeniu), a przy tym żeby interpretacje były jak najprostsze. **Złożoność interpretatora** to np. wielkość drzewa (gdy to drzewa decyzyjne) lub liczba niezerowych współczynników (dla modeli liniowych). LIME używa **rzadkich modeli liniowych (sparse linear models)** do interpretacji, typowo LASSO regression z narzuconą maksymalną liczbą cech.

Algorytm:

1. Dla punktu  $x$  wygeneruj jego otoczenie przez tworzenie **przemieszanych próbek (perturbed samples)**
2. Używając black boxa oblicz predykcje dla próbek
3. Dokonaj ważenia próbek w zależności od ich odległości od  $x$
4. Wytrenuj ważony model interpretowalny na próbках
5. Dokonaj interpretacji modelu

Jak generować próbki? To już zależy od rodzaju danych. Dzięki temu, że powyższy algorytm jest tak ogólny, to można go zastosować i dla danych tabelarycznych, i obrazów, i tekstu, różnica tkwi w generowaniu próbek.

Dla **danych tabelarycznych** generowanie próbek polega na modyfikacji poszczególnych cech. Dla cechy dodajemy wartość wylosowaną z rozkładu normalnego, ze średnią i odchyleniem standardowym obliczonym dla danej cechy (na zbiorze treningowym). Późniejsze obliczanie bliskich punktów używa exponential smoothing kernel do zdefiniowania bliskości i ważenia.

Dla **obrazów** nie miałoby sensu perturbowanie pojedynczych pikseli - w końcu na obiekt wpływają grupy pikseli, tworzące jakieś wizualne cechy. Obraz jest więc segmentowany na grupy połączonych ze sobą pikseli, czyli **superpiksele (superpixels)**. Generowanie nowych próbek polega na losowym wyłączaniu poszczególnych superpikseli, czyli zamianie tych obszarów na czarny lub szary kolor. Jak obliczać takie superpiksele? Trzeba do tego algorytmu segmentacji obrazu, właściwie dowolnego.

## SHAP

**SHapley Additive exPlanations (SHAP)** to model outcome explanation, który wyjaśnia wpływ poszczególnych cech na daną predykcję. W przeciwieństwie do LIME'a nie próbuje więc wyjaśniać lokalnie modelu, bo w ogóle nie definiuje lokalności czy otoczenia, tylko skupia się na wyjaśnianiu cech.

Idea SHAPa opiera się na obliczaniu **wartości Shapleya (Shapley values)**. Wywodzi się to z teorii gier kooperatywnych, bo zakładamy, że cechy to gracze, którzy dzielą się na 2 drużyny (koalicje): jedna chce zaklasyfikować przykład do klasy 0, druga do klasy 1. Każda cecha ma pewną wagę, czyli jak dużo wnosi dla swojej strony - to jest właśnie wartość Shapleya. Finalny model jest addytywny, czyli ważymy wartości cech przez wartości Shapleya, sumujemy i dostajemy wynik dla koalicji, czyli predykcję modelu dla danej klasy. Wygrywa oczywiście ta, która odpowiada klasie przewidywanej przez model, natomiast wartości Shapleya pozwalają wyznaczyć, jaki był udział poszczególnych cech w tej predykcji, czyli jak dużo dana cecha (gracz, koalicjant) wniosł dla swojej drużyny (koalicji).

SHAP został zaprojektowany tak, aby miał 3 własności:

- **lokalna celność (local accuracy)** - model wyjaśniający ma być wierny wyjaśnianemu modelowi przynajmniej dla danego przykładu
- **brak cechy (missingness)** - jeżeli uprościliśmy przykład  $\mathbf{x}$ , tworząc przykład  $\mathbf{x}'$  bez pewnych cech, to wpływ tych usuniętych cech będzie zerowy; daje to gwarancję, że SHAP zadziała prawidłowo w przypadku wartości brakujących w danych
- **spójność (consistency)** - jeżeli wyjaśniany model się zmieni, tak że wpływ danej cechy zwiększy się lub pozostanie bez zmian, to wartości Shapleya też wzrosną lub pozostaną bez zmian; gwarantuje zapewnienie różnych stricte matematycznych wymagań teorii gier i obliczalności wartości Shapleya

Jest jeden problem - **dokładne obliczanie wartości Shapleya ma złożoność eksponencjalną**, bo w uproszczeniu wymaga rozważenia wszystkich możliwych kombinacji cech. Dlatego SHAP definiuje metody przybliżania tych wartości metodami jądrowymi (KernelSHAP) oraz szczególnie efektywną implementację dla metod opartych o drzewa decyzyjne, bo tam każda cecha jest rozważana indywidualnie, bez wpływu innych (TreeSHAP).

## Wyjaśnienia kontrafaktualne

**Wyjaśnienia kontrafaktualne (counterfactual explanations)** polegają na wyjaśnianiu sytuacji przez zaprzeczenie, np. "If X had not occurred, Y would not have occurred". Wyznaczamy zatem w ten sposób warunki, które pozwoliłyby zmienić rzeczywistość tak, aby zmienił się oczekiwany wynik. W praktyce interesuje nas minimalna zmiana, czyli jak zmienić jak najmniejszą liczbę cech o jak najmniejszą wartość, aby zmienić predykcję modelu z nieprawidłowej klasy na poprawną. Jest też możliwy wariant ciągły: nie pytamy, co trzeba, żeby zmienić klasę (to może być trudne np. przy bardzo niezbalansowanych klasach), tylko co trzeba zrobić, żeby zwiększyć prawdopodobieństwo predykcji w % albo o określoną liczbę (dla regresji).

Jest to rodzina metod, które mogą być zarówno model-specific, jak i model-agnostic. Są oczywiście outcome explanation, bo wyjaśniają konkretne przykłady. Po co nam w ogóle takie wyjaśnianie modelu? Bo dostajemy krytyczne zmiany, które musiałyby nastąpić, żeby zmienić klasę, czyli wiemy, gdzie model najbardziej się wałał.

Zaletą takiego podejścia jest, że jest explicite **kontrastywne** oraz **selektywne**. Oznacza to, że mamy konkretne przykłady do porównania ze sobą, a ludzie dobrze myślą w kategoriach kontrastu (różnicy) między A i B. Selektyność oznacza ograniczenie się do podzbioru cech, bo ludziom łatwiej analizować mniej naraz.

Wadą jest to, że mamy **efekt Rashomon** - poszczególne przykłady kontrafaktualne mogą sugerować konflikujące, a nawet w pełni wykluczające się sposoby na poprawienie przykładu i nie wiadomo, co z tym zrobić. Można albo zwrócić wszystkie możliwości, albo w jakiś sposób je posortować i zwrócić najlepsze (ale wymaga to dobrze zdefiniowanej miary ich jakości).

Kryteria jakości przykładów kontrafaktualnych:

- **rzadkość (sparsity)** - liczba cech, którą trzeba zmienić, chcemy jak najmniej, bo wtedy łatwiej analizować
- **różnorodność (diversity)** - odległość między generowanymi przykładami kontrafaktualnymi, chcemy dość dużej, żeby sprawdzić różne możliwości
- **bliskość (proximity)** - odległość między oryginalnym punktem a przykładem kontrafaktualnym, chcemy jak najmniej, bo wtedy dostajemy minimalne zmiany, których trzeba dokonać

Przykład zastosowania: analiza ryzyka kredytowego. System oparty o ML odrzucił kredyt dla klienta, a bank chce udzielać jak najwięcej kredytów żeby mieć zysk, więc pytamy: "Co trzeba by zmienić u tego gościa, żeby jednak warto mu było udzielić kredytu?" i potem można się przyjrzeć bliżej wybranym cechom (np. dopytać o dodatkowe dokumenty). Jak system zwróci 2 przykłady "trzeba by zarobków wyższych o 100 zł miesięcznie" oraz "gdyby spłacił pożyczkę 10 lat temu", to możemy podejrzewać, że to jednak mało ryzykowny klient i można mu tego kredytu udzielić.

Inny przykład: wynajmujemy mieszkanie i chcemy ustalić cenę najmu. Mamy jego cechy, np. czy dopuszczały zwierzęta, powierzchnia i tym podobne, wrzucamy to do modelu. Wychodzi 1000 zł - mało, chcemy przynajmniej 1200. Przykład kontrafaktualny sugeruje zwiększenie powierzchni - nierealne - albo dopuszczenie zwierząt. To drugie jest realne, zmieniamy, mamy więcej pieniędzy.

Algorytm Wachter et al. generowania przykładów minimalizuje koszt:

$$L(x, x', y', \lambda) = \lambda \cdot (\hat{f}(x') - y')^2 + d(x, x')$$

$x$  - dany przykład

$x'$  - przykład kontrafaktualny

$y'$  - pożądany wynik, np. docelowa klasa przykładu kontrafaktualnego

$f(x')$  - przewidywany przez nasz algorytm wynik dla przykładu kontrafaktualnego

$d(x, x')$  - odległość między oryginalnym a kontrafaktualnym przykładem

$\lambda$  - współczynnik skalujący, hiperparametr

Odległość to odległość punktów w metryce L1 (Manhattan) znormalizowana przez Median Absolute Deviation (MAD):

$$d(x, x') = \sum_{j=1}^p \frac{|x_j - x'_j|}{MAD_j}$$

$$MAD_j = \text{median}_{i \in \{1, \dots, n\}} (|x_{i,j} - \text{median}_{l \in \{1, \dots, n\}} (x_{l,j})|)$$

Algorytm:

1. Narzucamy  $x$ ,  $y'$ , tolerancję  $\varepsilon$  i niewielką początkową wartość  $\lambda$
2. Wylosuj dowolny punkt  $x'$  ze zbioru jako początkowy przykład kontrafaktualny
3. Zoptymalizuj funkcję kosztu (podaną powyżej), otrzymując nowe  $x'$
4. Tak długo, jak  $|f(x') - y'| > \varepsilon$ :
  - a. Powiększ  $\lambda$
  - b. Zoptymalizuj funkcję kosztu, zaczynając od aktualnego  $x'$
  - c. Zwróć wygenerowany  $x'$  minimalizujący funkcję kosztu
5. Powtarzaj kroki 2-4 i zwróć listę wygenerowanych przykładów kontrafaktualnych albo ten o najmniejszym koszcie

# Przykłady adwersarialne i ataki na systemy ML

**Przykłady adwersarialne (adversarial examples)** to takie przykłady, które potrafią "oszukać" algorytm. W szczególności polegają na drobnej zmianie, która wykorzystuje czułość algorytmu i brak realnej inteligencji w nim, dzięki czemu można zmienić predykcję. Typowo bada się je w kontekście sieci neuronowych, bo np. na obrazach dobrze to widać.

Istnieją 3 hipotezy, czemu takie przykłady w ogóle istnieją.

**Hipoteza Szegedy'ego** mówi, że wynika to z istnienia niewielkich obszarów o niskim prawdopodobieństwie na powierzchni manifoldu tworzonego przez trening sieci neuronowych. Zbytnia nieliniowość i niedostateczna regularyzacja skutkują według niego bardzo złożonymi powierzchniami, przez co niektóre fragmenty w ogóle nie są pokrywane przez zbiór uczący i można tam właśnie wygenerować przykłady adwersarialne.

**Hipoteza Goodfellow'a** idzie w drugą stronę i twierdzi, że przykłady adwersarialne biorą się ze zbytniej liniowości w systemach ML, szczególnie głębokich sieciach neuronowych. Jego zdaniem funkcje aktywacji jak Softmax albo ReLU są w gruncie rzeczy linią prostą, a podczas treningu sieć w znacznej części uczy się po prostu zależności liniowych, które tylko trochę się zmieniają i kierują się w gruncie rzeczy w podobnym kierunku. Sieć nie potrafi się więc pozbyć niewielkich perturbacji na wejściu, są one akumulowane przez te liniowości i powodują drastyczne zmiany na wyjściu.

**Hipoteza przekrzywionej granicy (tilted boundary)** opiera się na obserwacji, że model nie ma szans nigdy dostosować się idealnie do danych - wtedy na danych testowych mielibyśmy dokładność 100%. Mamy więc pewną przestrzeń między granicą decyzyjną klasyfikatora a faktycznym manifoldem, na którym leżą dane. W tej właśnie przestrzeni leżą przykłady adwersarialne i są one nie da się wyeliminować faktu ich istnienia.

Ataki na systemy ML wykorzystują explicite słabości modeli i generują przykłady adwersarialne tak, aby je zmylić. Typowo są to **ataki gradientowe**, które wymagają znajomości gradientu obliczanego przez sieć neuronową. Patrzymy w nich, w jakim kierunku kieruje się typowo gradient dla danej klasy, np. sprawdzamy gradienty dla mnóstwa przykładów klasy "cat". Uśredniamy, bierzemy znak (-1/+1) dla każdego piksela, mnożymy przez mały ułamek i dostajemy coś, co wygląda jak szum, stąd też nazwa **atak szumem**. Dodajemy to do obrazu i sieć prawie na pewno uzna to za klasę, z której pochodzi gradient.

Można się przed czymś takim bronić - a nawet trzeba, skoro hipoteza tilted boundary mówi, że wyeliminować się nie da. Najprościej zrobić **trening adwersarialny** - samemu wygenerować dużo przykładów adwersarialnych i dodać je do zbioru treningowego, tak, aby sieć nauczyła się przypisywać prawidłowe klasy do perturbowanych wejść. Niestety w praktyce chroni to tylko przed najprostszymi atakami.

Można też wykorzystać destylację wiedzy, nazywa się w takim wypadku **destylacją obronną (defensive distillation)**. Wykorzystujemy tutaj drugi model, taki sam jak oryginalny, ale wykorzystujemy dark knowledge z treningu nauczyciela dzięki nauce na "soft labels". Odpowiada to wygładzaniu funkcji kosztu - gradienty stają się mniej ekstremalne, bo uczymy się podobieństw między klasami. Uodparnia to w znacznej mierze na ataki gradientowe.

Można też użyć osobnych algorytmów do identyfikacji przykładów adwersarialnych podczas predykcji. **Deep k Nearest Neighbors (DkNN)** to przykład takiego algorytmu.

Opiera się on na 3 definicjach:

- **pewność (confidence)** modelu można potraktować jako odległość między punktem testowym a punktami ze zbioru treningowego - jeżeli odległości są małe, to widzieliśmy już coś podobnego
- **interpretowalność** można zdobyć przez znalezienie punktów na manifoldzie treningowym, które wspierają daną predykcję
- sieć jest **solidna (robust)**, gdy jej predykcje są podobne we wszystkich warstwach, czyli finalna predykcja ma wysoką pewność

Żeby zapewnić te 3 własności, w DkNN robimy wyszukiwanie najbliższych sąsiadów (kNN) po każdej warstwie sieci neuronowej. Dla wyjścia danej warstwy (przetworzonego punktu testowego) szukamy najbliższych przykładów ze zbioru treningowego. Analizujemy klasę przewidywaną przez kNN dla tego przykładu - jeżeli się zmienia, to znaczy, że wyniki naszej sieci mocno różnią się od wyników dla podobnych przykładów, więc coś jest nie tak i możemy podejrzewać atak.

# Metody bayesowskie

## Podstawy statystyki bayesowskiej

### Definicje i prawa prawdopodobieństwa

**Prawdopodobieństwo łączne (joint probability)** to prawdopodobieństwo wystąpienia jednocześnie zestawu konkretnych wartości dla wielu zmiennych losowych:

$$p(X = x_i, Y = y_j) = \frac{n_{ij}}{N}$$

$n_{ij}$  - liczba punktów z wartościami  $x_i$  oraz  $y_j$

$N$  - łączna liczba punktów

**Prawdopodobieństwo brzegowe (marginal probability)** to prawdopodobieństwo wystąpienia konkretnej wartości dla danej zmiennej przy dowolnej wartości innych zmiennych:

$$p(X = x_i) = \frac{c_i}{N}$$

$c_i$  - liczba punktów z wartością  $x_i$

Można to też zdefiniować jako sumę prawdopodobieństw po wszystkich wartościach innych zmiennych:

$$p(X = x_i) = \sum_j p(X = x_i, Y = y_j)$$

**Prawdopodobieństwo warunkowe** to prawdopodobieństwo wystąpienia konkretnej wartości dla danej zmiennej dla podzbioru zdarzeń o konkretnej wartości innej zmiennej:

$$p(Y = y_j | X = x_i) = \frac{n_{ij}}{c_i}$$

$c_i$  - liczba punktów z wartością  $x_i$ , czyli spełniających warunek

$n_{ij}$  - liczba punktów z wartościami  $x_i$  oraz  $y_j$

Można to też zdefiniować jako iloczyn prawdopodobieństw dla warunku i wartości zmiennej:

$$p(X = x_i, Y = y_j) = p(Y = y_j | X = x_i)p(X = x_i)$$

Powyżej wykorzystano już 2 **prawa prawdopodobieństwa**, ale w specyficznej formie. W ogólności można je zdefiniować następująco.

#### **Prawo sumowania**

$$p(X) = \sum_Y p(X, Y)$$

#### **Prawo mnożenia**

$$p(X, Y) = p(Y|X)p(X)$$

#### **Prawo symetrii**

$$p(X, Y) = p(Y, X)$$

## **Twierdzenie Bayesa**

**Twierdzenie Bayesa** pozwala obliczyć wartość zmiennej pod warunkiem innej zmiennej, gdy umiemy obliczyć tylko relację odwrotną:

$$p(Y|X) = \frac{p(X, Y)}{p(X)} = \frac{p(Y, X)}{p(X)} = \frac{p(X|Y)p(Y)}{p(X)}$$

$$p(Y|X) = \frac{p(X|Y)p(Y)}{\sum_Y p(Y, X)} = \frac{p(X|Y)p(Y)}{\sum_Y p(X|Y)p(Y)}$$

Można to wyprowadzić krok po kroku z praw prawdopodobieństwa. Na początek mamy z prawa mnożenia:

$$p(X, Y) = p(Y|X)p(X) \Rightarrow p(Y|X) = \frac{p(X, Y)}{p(X)}$$

Potem z prawa przemienności zamieniamy miejscami w liczniku, a potem rozbijamy prawdopodobieństwo łączne na warunkowe, znowu z prawa mnożenia. To jest górną linijką.

Na dole z prawa sumowania dodajemy sobie zmienną  $\mathbb{Y}$ , explicite podkreślając, że w twierdzeniu Bayesa normalizujemy po wszystkich możliwych wartościach  $\mathbb{Y}$ . Potem rozbijamy prawdopodobieństwa łączne z prawa mnożenia.

Czemu to przekształcenie jest ważne? Bo wiemy, że możemy obliczyć  $p(\mathbb{Y}|\mathbb{X})$ , czyli typowo klasę mając dany pomiar, znając tylko prawdopodobieństwo klasy oraz prawdopodobieństwo tego punktu w tej klasie. Te wartości są znacznie łatwiej mierzalne.

## Prawdopodobieństwo częstotliwościowe vs bayesowskie

Powyższa definicja jest z perspektywy klasycznej, czyli **prawdopodobieństwa częstotliwościowego**. W nim prawdopodobieństwo to granica częstotliwości występowania zdarzenia w wielu próbach.

Na twierdzenie Bayesa można spojrzeć inaczej, z perspektywy **prawdopodobieństwa bayesowskiego**, w którym mierzmy niepewność, a pomiary to dowody, na których się opieramy, zmniejszające niepewność.

**Prawdopodobieństwo a priori (prior probability)** to prawdopodobieństwo zdarzenia przed zaobserwowaniem czegokolwiek, czyli po prostu  $p(X)$ . Jest to nasze pierwotne przekonanie o ogólnym prawdopodobieństwie na dane zdarzenie.

**Prawdopodobieństwo a posteriori (posterior probability)** to prawdopodobieństwo zdarzenia po obserwacji, czyli  $p(Y|X)$ . Oznacza korektę pierwotnego prawdopodobieństwa w świetle nowych dowodów, które albo wspierają prior (zwiększymy oszacowanie), albo są przeciwko niemu (zmniejszymy oszacowanie).

**Dowód (evidence)** to nowe pomiary, ze względu na które korygujemy swoje dotychczasowe poglądy.

**Wiarygodność (likelihood)** to prawdopodobieństwo zaobserwowania zdobytych dowodów przy aktualnym modelu, czyli  $p(X|Y)$ . Jest to miara dopasowania aktualnych przekonań do nowo zaobserwowanych dowodów. Uwaga - to **nie** jest funkcja gęstości prawdopodobieństwa (PDF), więc nie musi mieć wartości w zakresie  $[0, 1]$ !

W świetle podejścia bayesowskiego twierdzenie Bayesa zapewnia sposób na **obliczenie prawdopodobieństwa posterior na podstawie priora w świetle nowych dowodów**.

Po co to komu? Po pierwsze w podejściu bayesowskim mamy **oszacowanie niepewności** - algorytmy ML oparte o nie dadzą nie tylko taką liczbę, ale też miarę pewności co do wyniku, co często ma duże znaczenie. Dodatkowo sama definicja bayesowska podkreśla korygowanie prawdopodobieństw, a zatem **iteracyjność algorytmów**, które dobrze nadają się do podejścia online i dodawania na bieżąco nowych danych.

## Wielowymiarowy rozkład normalny

Rozkład normalny (Gaussowski) jest szczególnie ważny z perspektywy bayesowskiej, bo jest bardzo wygodny w użyciu. Można typowo przyjąć prior Gaussowski i wyniki wyjdą przyzwoite, bo mamy centralne twierdzenie graniczne, które uzasadnia częste występowanie rozkładów zbliżonych do normalnych.

**Wielowymiarowy rozkład normalny** ma wzór:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\boldsymbol{\Sigma}|^{1/2}} \exp \left\{ -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right\}$$

$\mathbf{x} \in \mathbb{R}^{D \times 1}$

$\boldsymbol{\mu} \in \mathbb{R}^{D \times 1}$  - mean vector

$\boldsymbol{\Sigma} \in \mathbb{R}^{D \times D}$  - covariance matrix

$|\boldsymbol{\Sigma}|$  - determinant of  $\boldsymbol{\Sigma}$

$\Delta^2 = (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$  - Mahalanobis distance

Macierz kowariancji wyznacza kowariancje poszczególnych cech. W szczególności jeżeli wszystkie są zerami, czyli jest przekątniowa, to rozkład jest **anizotropowy**, czyli taki sam w każdym kierunku.

Jest to rozkład **unimodalny**, czyli ma jedno maksimum, które jest w związku z tym modą (najczęstszą wartością danych). Jest ono równe wartości średniej.

Rozkład normalny to **rozkład sprzężony (conjugate prior)**. Są to takie rozkłady, że kiedy wybierzemy taki prior, to posterior we wnioskowaniu bayesowskim będzie z tej samej rodziny rozkładów. Jest z tym powiązana ważna własność: jeżeli operujemy na rozkładach normalnych, to prawdopodobieństwo warunkowe i brzegowe też mają rozkład normalny; jeżeli są dodatkowo niezależne, to jest też tak dla prawdopodobieństwa łącznego. Bardzo upraszcza to obliczenia.

Powyższe pozwala też na **podział (partitioning)** wektorów na fragmenty (macierzy kowariancji na bloki) i przetwarzanie ich osobno - mamy gwarancję, że wszędzie tam są rozkłady normalne:

$$\begin{aligned} \mathbf{x} &= \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix}, \boldsymbol{\mu} = \begin{bmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{bmatrix} \\ \boldsymbol{\Sigma} &= \begin{bmatrix} \boldsymbol{\Sigma}_{aa} & \boldsymbol{\Sigma}_{ab} \\ \boldsymbol{\Sigma}_{ba} & \boldsymbol{\Sigma}_{bb} \end{bmatrix}, \boldsymbol{\Sigma} = \boldsymbol{\Sigma}^T, \boldsymbol{\Sigma}_{ab} = \boldsymbol{\Sigma}_{ba}^T \\ \boldsymbol{\Lambda} &= \boldsymbol{\Sigma}^{-1}, \boldsymbol{\Lambda} = \begin{bmatrix} \boldsymbol{\Lambda}_{aa} & \boldsymbol{\Lambda}_{ab} \\ \boldsymbol{\Lambda}_{ba} & \boldsymbol{\Lambda}_{bb} \end{bmatrix} \end{aligned}$$

# Bayesowska aproksymacja funkcji

## Klasyczne dopasowanie krzywej

Regresję dla przypadku 1D można zdefiniować alternatywnie jako **aproksymację funkcji / dopasowanie krzywej**. Jest to zadanie analizy numerycznej podobne do interpolacji, ale dopuszczamy, żeby dopasowywana krzywa nie przechodziła dokładnie przez zmierzone punkty, tylko chcemy minimalizować sumę kwadratów różnic w tych punktach.

Mamy pomiary w punktach  $\mathbf{x}$  o wartościach  $t$ . Ich kształt aproksymujemy modelem liniowym z bazą wielomianową stopnia  $m$ .

$$\mathbf{x} = [x_1, \dots, x_n]^T, \mathbf{t} = [t_1, \dots, t_n]^T$$

$$y(x, \mathbf{w}) = \sum_{i=0}^m w_i x^i$$

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=0}^m (y(x_i, \mathbf{w}) - t_i)^2$$

Dodajemy regularyzację L2, żeby nie overfittować:

$$\tilde{E}(\mathbf{w}) = \frac{1}{2} \sum_{i=0}^m (y(x_i, \mathbf{w}) - t_i)^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Powyższe to klasyczna definicja z analizy numerycznej. Można natomiast podejść do problemu z perspektywy prawdopodobieństwa bayesowskiego.

## Dopasowanie krzywej maksymalną wiarygodnością

Mamy stały zbiór danych  $D$  i zmienny wektor wag  $\mathbf{w}$ , który parametryzuje nasz model.

Zakładamy pewien prior  $p(\mathbf{w})$  - jak nie mamy szczególnych przekonań co do wag, to może być chociażby jednostajny.

$p(D|\mathbf{w})$  to funkcja wiarygodności, która mierzy, jak bardzo prawdopodobny jest zbiór danych przy aktualnym wektorze wag  $\mathbf{w}$

$p(\mathbf{w}|D)$  to posterior, miara niepewności aktualnych wag w po zaobserwowaniu danych  $D$ . Z twierdzenia Bayesa znamy wzór na  $p(\mathbf{w}|D)$ , natomiast jako że dane są stałe, to możemy je opuścić i dostać samą proporcję posteriora

$$p(\mathbf{w}|D) = \frac{p(D|\mathbf{w})p(\mathbf{w})}{p(D)} \quad \text{posterior} \propto \text{likelihood} \times \text{prior}$$

**Funkcja wiarygodności (likelihood function)** mierzy jakość dostosowania modelu statystycznego (o pewnych ustalonych parametrach) do zaobserwowanych danych. Wykorzystuje prawdopodobieństwo łączne wygenerowania zbioru (wszystkich próbek).

**Metoda największej wiarygodności (maximum likelihood estimation)** polega na takim dostosowaniu parametrów modelu, aby zmaksymalizować prawdopodobieństwo wygenerowania zbioru danych. Chcemy więc dostać model najlepiej dopasowany do faktycznych danych. W praktyce maksymalizujemy logarytm z wiarygodności, czyli **log-likelihood**, bo jest prościej liczyć pochodne.

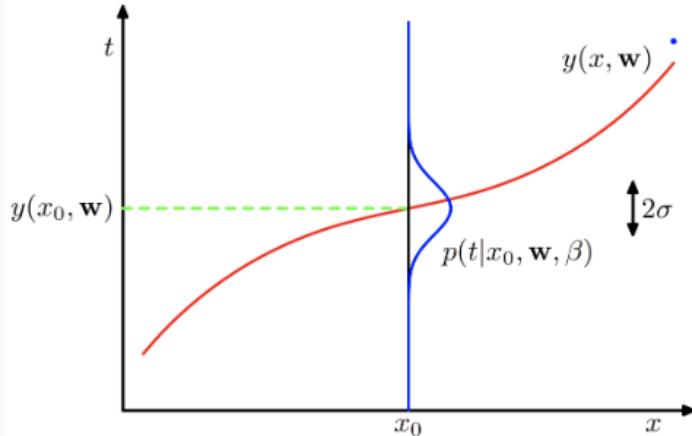
W bayesowskim dopasowaniu krzywej zakładamy, że prawdopodobieństwo otrzymania danej wartości jest dane przez rozkład normalny, mający swój środek w tym punkcie:

$$p(t|x, \mathbf{w}, \beta) = \mathcal{N}(t|y(x, \mathbf{w}), \beta^{-1})$$

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(x - \mu)^2\right\}$$

$$\text{Precision: } \beta^{-1} = \sigma^2$$

$\beta$  to precyza, czyli odwrotność wariancji, wyznacza dopuszczalną niepewność w góre i w dół. Mamy wysoką precyzję, czyli małą wariancję, to dzwon krzywej Gaussa wokół każdego punktu jest wąski, czyli krzywa musi przechodzić blisko punktów.



Chcemy teraz dostać takie wartości parametrów  $w$ , żeby zmaksymalizować wiarygodność, czyli stworzyć model, który z największym prawdopodobieństwem wygeneruje nasz zbiór danych.

Zakładamy **własność i.i.d. (independently and identically distributed)**, czyli każdy punkt był próbkowany niezależnie od pozostałych i w ten sam sposób. Pozwala nam to użyć tych samych parametrów dla całego modelu oraz mnożyć prawdopodobieństwa. W związku z tym prawdopodobieństwo całego zbioru to iloczyn prawdopodobieństw dla poszczególnych punktów. Mamy zatem likelihood:

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = \prod_{i=0}^n \mathcal{N}(t_i|y(x_i, \mathbf{w}), \beta^{-1})$$

Optymalizacja powyższego wyrażenia byłaby bardzo trudna - w końcu rozkład normalny ma eksponentę. Możemy zamiast tego wziąć logarytm, bo jest monotonicznie rosnący, więc

minimum będzie w tym samym miejscu. Przy okazji uprości nam to wzór. Jeżeli zanegowalibyśmy dodatkowo znaki, to dostalibyśmy **zanegowaną logarytmiczną funkcję wiarygodności (negative log likelihood, NLL)**. Zwykła log likelihood to:

$$\ln p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta) = -\frac{\beta}{2} \sum_{i=0}^n (y(x_i, \mathbf{w}) - t_i)^2 + \frac{n}{2} \ln \beta - \frac{n}{2} \ln (2\pi)$$

Ostatni element to stała, więc się nim nie przejmujemy. Pierwszy za to jest **sumą kwadratów błędów (RSS)**, czyli funkcją kosztu z klasycznej regresji liniowej. Widać więc, że przy założeniu szumu Gaussowskiego (niepewność wokół punktów to szum) przy powyższym rozumowaniu dostajemy po prostu klasyczną aproksymację średniokwadratową.

Przydałoby się jednak policzyć jeszcze same wagę i precyzję, korzystając oczywiście z danych:

$$\mathbf{w}_{ML} = \operatorname{argmin}_{\mathbf{w}} \sum_{i=0}^n (y(x_i, \mathbf{w}) - t_i)^2$$

$$\frac{1}{\beta_{ML}} = \sum_{i=1}^n (y(x_i, \mathbf{w}_{ML}) - t_i)^2$$

Warto zauważyć, że w związku z tym odwrotność precyzji jest związana z wariancją rezyduów naszego modelu - ma to sens, bo w związku z tym słabo dopasowany model miałby niską precyzję, a więc dużą wariancję rezyduów.

Zbierając to wszystko, mając już znane wagę i precyzję, dostajemy **rozkład predykcyjny (predictive distribution)**, czyli rozkład pozwalający robić regresję dla innych punktów:

$$p(t|x, \mathbf{w}_{ML}, \beta_{ML}) = \mathcal{N}(t|y(x, \mathbf{w}_{ML}), \beta_{ML}^{-1})$$

Powyższe rozumowanie pozwoliło wyprowadzić wzór na wagę zgodnie z zasadą **maximum likelihood (ML)**. Co więcej, jako że mamy pełen rozkład predykcyjny, to mamy nie tylko wartości dla tych punktów (średnia), ale też niepewność w nich (odchylenie standardowe).

## Bayesowskie dopasowanie krzywej - MAP

W poprzednim rozumowaniu nie użyliśmy twierdzenia Bayesa ani w ogóle wnioskowania bayesowskiego jako takiego - tylko powiązaliśmy regresję z wiarygodnością, co jest pojęciem też ze zwykłego prawdopodobieństwa częstotliwościowego.

Żeby użyć podejścia bayesowskiego, zdefiniować prior wag, na przykład jako rozkład normalny (wagi mogą się "odchylić" jednakowo w górę i w dół):

$$p(\mathbf{w}|\alpha) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \alpha^{-1} \mathbf{I}) = \left(\frac{\alpha}{2\pi}\right)^{(m+1)/2} \exp\left\{-\frac{\alpha}{2} \mathbf{w}^T \mathbf{w}\right\}$$

$\alpha$  - współczynnik wariancji wag, który zakładamy (bo to my narzucamy prior)

Z twierdzenia Bayesa wiemy, że posterior jest proporcjonalny do iloczynu wiarygodności (likelihood) i priora nad wagami:

$$p(\mathbf{w}|\mathbf{x}, \mathbf{t}, \alpha, \beta) \propto p(\mathbf{t}|\mathbf{x}, \mathbf{w}, \beta)p(\mathbf{w}|\alpha)$$

Jak mamy coś takiego, to żeby dostać w najbardziej prawdopodobne dla danych używamy metody **maximum a posteriori (MAP)**. Przy okazji od razu dodajemy logarytm:

$$\mathbf{w}_{MP} = \underset{\mathbf{w}}{\operatorname{argmin}} \left\{ \frac{\beta}{2} \sum_{i=0}^n (y(x_i, \mathbf{w}) - t_i)^2 + \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} \right\}$$

W odróżnieniu od wersji z ML pojawił nam się drugi człon, zależny od parametru  $\alpha$  z narzuconego przez nas priora. Jest to suma kwadratów wag, bo to iloczyn skalarny, a więc "wagi do kwadratu", czyli **regularyzacja L2**. Widać, że wyprowadzenie z podejściem bayesowskim naturalnie dodaje przeciwdziałanie overfittingowi do modelu.

Sam rozkład posterior jest jednak dla wag - pozwala je wyznaczyć, fajnie, ale my chcemy model pozwalający przewidywać wartości regresji dla nowych x. Jest to rozkład predykcyjny (powinny tu być jeszcze precyzie  $\beta$ , ale opuszczono dla czytelności):

$$p(t|x, \mathbf{x}, \mathbf{t}) = \int p(t|x, \mathbf{w})p(\mathbf{w}|\mathbf{x}, \mathbf{t})d\mathbf{w}$$

Mamy całkę po wagach z iloczynu prawdopodobieństwa warunkowego dla wag (założyliśmy na początku, że to rozkład normalny) i posteriora nad wagami w. Jest to w praktyce **konwolucja (convolution)** dwóch rozkładów normalnych, więc wynik to też rozkład normalny. Finalny rozkład predykcyjny to:

$$p(t|x, \mathbf{x}, \mathbf{t}) = \mathcal{N}(t|m(x), s^2(x))$$

Średnią i wariancję estymujemy z danych. Zakładamy cały czas bazę wielomianów kanonicznych.

$$\phi(x) = [1, x, \dots, x^m]^T$$

$$m(x) = \beta \phi(x)^T \mathbf{S} \sum_{i=0}^n \phi(x_i) t_i$$

$$s^2(x) = \beta^{-1} + \phi(x)^T \mathbf{S} \phi(x)$$

$\mathbf{S}$  to **macierz kowariancji** danych. Dla przypadku 1D odwrotność tej macierzy (bo tak łatwiej zapisać) to:

$$\mathbf{S}^{-1} = \alpha \mathbf{I} + \beta \sum_{i=0}^n \phi(x_i) \phi(x_i)^T$$

## Bayesowska regresja liniowa

Dopasowanie krzywej to dobra rzecz, ale działa w 1D. Chcielibyśmy pełnej regresji liniowej w wielu wymiarach, zakładając dla uproszczenia prostą bazę liniową i model postaci:

$$y(\mathbf{x}, \mathbf{w}) = \sum_{i=0}^{m-1} w_i \phi_i(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x})$$

$$\mathbf{w} = [w_0, \dots, w_{m-1}]^T$$

$$\phi = [\phi_0, \dots, \phi_{m-1}]^T$$

Wartości z danych to predykcje modelu plus pewna niedokładność. Wiarygodność dla tych danych to znowu rozkład normalny, z pewną precyzją:

$$t(\mathbf{x}) = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1}) = \mathcal{N}(t|\mathbf{w}^T \phi(\mathbf{x}), \beta^{-1})$$

Funkcja wiarygodności (likelihood) to znowu iloczyn prawdopodobieństw, czyli prawdopodobieństwo uzyskania całego zbioru. Warto jednak zauważyć, że teraz dane to pełna macierz  $\mathbf{X}$ , czyli punkty to  $D$ -wymiarowe wektory. Bierzemy oczywiście znowu logarytm i dostajemy log-likelihood, znowu z minimalizacją błędu kwadratowego:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{i=1}^n \mathcal{N}(t_i|\mathbf{w}^T \phi(\mathbf{x}_i), \beta^{-1})$$

$$\ln p(\mathbf{t}|\mathbf{w}, \beta) = -\frac{\beta}{2} \sum_{i=0}^n (t_i - \mathbf{w}^T \phi(\mathbf{x}))^2 + \frac{n}{2} \ln \beta - \frac{n}{2} \ln (2\pi)$$

Minimalizujemy licząc gradient i przyrównując do zera. Na podstawie tego dostajemy **rozwiązańe maximum likelihood**.  $\Phi$  to macierz wartości funkcji bazowych dla poszczególnych punktów i wymiarów.

$$\nabla \ln p(\mathbf{t}|\mathbf{w}, \beta) = 0$$

$$\mathbf{w}_{ML} = (\Phi^T \Phi)^{-1} \Phi^T \mathbf{t}$$

$$\frac{1}{\beta_{ML}} = \frac{1}{n} \sum_{i=1}^n (\mathbf{w}_{ML}^T \phi(\mathbf{x}_i) - t_i)^2$$

$$\Phi = \begin{bmatrix} \phi_0(\mathbf{x}_1) & \phi_1(\mathbf{x}_1) & \dots & \phi_{m-1}(\mathbf{x}_1) \\ \phi_0(\mathbf{x}_2) & \phi_1(\mathbf{x}_2) & \dots & \phi_{m-1}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(\mathbf{x}_n) & \phi_1(\mathbf{x}_n) & \dots & \phi_{m-1}(\mathbf{x}_n) \end{bmatrix}$$

Znowu mamy sformułowanie klasyczne, możliwości wrzucenia priora. Korzystając z podejścia bayesowskiego, możemy to zrobić. Zakładamy prior Gaussowski:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{m}_0, \mathbf{S}_0)$$

Rozkład posterior oczywiście też jest normalny i z twierdzenia Bayesa wynosi:

$$p(\mathbf{w} | \mathbf{t}) = \mathcal{N}(\mathbf{w} | \mathbf{m}_n, \mathbf{S}_n)$$

$$\mathbf{m}_n = \mathbf{S}_n (\mathbf{S}_0^{-1} \mathbf{m}_0 + \beta \Phi^T \mathbf{t})$$

$$\mathbf{S}_n^{-1} = \mathbf{S}_0^{-1} + \beta \Phi^T \Phi$$

MAP jest wyjątkowo proste - skoro mamy rozkład normalny, to jest to rozkład jednomodalny i najczęstsza wartość w danych to po prostu średnia:

$$\mathbf{w}_{MP} = \mathbf{m}_n$$

Powyższy problem można uprościć, czyniąc 2 założenia co do priora:

- wycentrowany prior, czyli średnia 0
- izotropowość rozkładu, czyli zerowe kowariancje (dane jednakowe w każdym kierunku)

$$p(\mathbf{w} | \alpha) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \alpha^{-1} \mathbf{I})$$

Posterior też jest prostszy:

$$p(\mathbf{w} | \mathbf{t}) = \mathcal{N}(\mathbf{w} | \mathbf{m}_n, \mathbf{S}_n)$$

$$\mathbf{m}_n = \beta \mathbf{S}_n \Phi^T \mathbf{t}$$

$$\mathbf{S}_n^{-1} = \alpha \mathbf{I} + \beta \Phi^T \Phi$$

MAP dostajemy z log-likelihood:

$$\ln p(\mathbf{w} | \mathbf{t}) = -\frac{\beta}{2} \sum_{i=1}^n (t_i - \mathbf{w}^T \phi(\mathbf{x}_i))^2 - \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} + const$$

Czyli mamy w tym uproszczonym wypadku regresję liniową z regularizacją L2.

## Bayesowska regresja liniowa online

Modele bayesowskie są szczególnie **wygodne do danych strumieniowych**, czyli działania online, bo pozwalają cyklicznie aktualizować dotychczasowe przekonania (model) na podstawie nowych dowodów.

W takim podejściu **posterior poprzedniej iteracji to prior następnej** (w tym np. współczynniki regularyzacji  $\alpha$  czy  $\beta$ ) i tak iteracyjnie to aktualizujemy.

Dzięki temu też **automatycznie się regularyzują**, bo same współczynniki regularyzacji też mogą się aktualizować w ten sposób

**Bayesowska regresja liniowa online** zakłada posterior proporcjonalny do poprzedniego posteriora (który teraz jest priorem) i wiarygodności wag:

$$p(\mathbf{w}_{i+1} | \mathbf{w}_i, \mathbf{x}_i, t_i) \propto p(t_i | \mathbf{x}_i, \mathbf{w}_i) p(\mathbf{w}_i)$$

Wzór na iteracyjny update posteriora:

$$p(\mathbf{w}_{i+1} | \mathbf{w}_i, \mathbf{x}_i, t_i) = \mathcal{N}(\mathbf{w}_{i+1} | \mathbf{m}_{i+1}, \mathbf{S}_{i+1})$$

$$\mathbf{S}_{i+1} = (\mathbf{S}_i^{-1} + \beta \mathbf{x}_i \mathbf{x}_i^T)^{-1}$$

$$\mathbf{m}_{i+1} = \mathbf{S}_{i+1} (\mathbf{S}_i^{-1} \mathbf{m}_i + \beta t_i \mathbf{x}_i)$$

Algorytm:

1. Przyjmij pewien początkowy prior, np. wyśrodkowany izotropowy rozkład normalny
2. Oblicz posterior na podstawie priora
3. Powtarzaj za każdym razem, kiedy przyjdzie nowy punkt:
  - a. Przyjmij poprzedni posterior jako prior
  - b. Oblicz nowy posterior

Oczywiście powyższe nie jest jeszcze użyteczne w praktyce - taki algorytm mógłby się kręcić w kółko dostając coraz lepsze wagi. Jak chcemy dostać z niego predykcje, to bierzemy aktualny posterior i robimy rozkład predykcyjny:

$$p(t_i | \mathbf{x}_i) = \mathcal{N}(t_i | \mu_i, \sigma_i)$$

$$\mu_i = \mathbf{w}_i^T \mathbf{x}_i$$

$$\sigma_i = \frac{1}{\beta} + \mathbf{x}_i^T \mathbf{S}_i \mathbf{x}_i$$

Ważna uwaga - wyprowadzenie jest względnie proste, a algorytm wydajny, bo prior to rozkład normalny, który jest rozkładem sprzężonym. Gdybyśmy chcieli inny prior, który nie jest sprzężony, to trzeba by wykorzystać kosztowne **próbkowanie Monte Carlo łańcuchami Markowa (Markov Chain Monte Carlo sampling, MCMC)**, które jest trudne i kosztowne.

# Naiwny klasyfikator Bayesa

## Problem predykcji

Nasze modele mają 3 możliwości nauki:

- budowanie łącznego rozkładu prawdopodobieństwa  $p(x, y)$ , czyli uczymy się ogólnej funkcji, z której pochodzą (przez którą są generowane) nasze dane, np. ukryty model Markowa (HMM), ograniczona maszyna Boltzmanna (RBM)
- budowanie warunkowego rozkładu prawdopodobieństwa  $p(y|x)$ , czyli uczymy się, jak wygląda granica decyzyjna pomiędzy klasami, np. naiwny klasyfikator Bayesa
- nie budować explicite rozkładu prawdopodobieństwa, tylko od razu samej granicy decyzyjnej, np. drzewa decyzyjne

Pierwsza grupa to **modele generatywne (generative models)**, bo uczą się rozkładu generującego dane. Często takie modele mogą faktycznie służyć do generacji danych. Są jednak cięższe do treningu, bo musimy oszacować "szerszy" rozkład warunkowy - nie mamy wygodnego ograniczenia w postaci prawdopodobieństwa warunkowego.

Druga grupa to **modele dyskryminatywne (discriminative models)**, które uczą się **funkcji dyskryminującej**, czyli funkcji opisującej **granicę decyzyjną (decision boundary)** oddzielającą klasy od siebie.

Typowo do tych drugich włącza się też algorytmy nie budujące explicite rozkładu prawdopodobieństwa, tylko samą funkcję dyskryminującą.

Modele dyskryminacyjne budujące rozkład posterior  $p(y|x)$  są lepsze, bo dają nam **więcej możliwości**:

- minimalizacja ryzyka - możemy precyjnie sterować finalną macierzą pomyłek, np. preferować fałszywie pozytywne od fałszywie negatywnych
- opcja reject - kiedy mamy dla decyzji tak/nie prawdopodobieństwo w okolicy 50%, możemy nie podejmować decyzji i zwrócić "nie wiem", bo wiemy, że model jest niepewny
- dobre dla zbiorów niebalansowanych - możemy wrzucić prior nad klasami, żeby uwzględnić ich niebalansowane i dać większą wagę bardziej interesującym klasom
- łączenie modeli - wykorzystujemy własność warunkowej niezależności (np. w naiwnym klasyfikatorze Bayesa) i możemy pomnożyć przez siebie prawdopodobieństwa z poszczególnych modeli

## Naiwny klasyfikator Bayesa

**Naiwny klasyfikator Bayesa** szacuje prawdopodobieństwo posterior k-tej klasy, zakładając naiwnie warunkową niezależność cech:

$$p(\mathcal{C}_k | x_1, \dots, x_n) = \frac{p(\mathcal{C}_k) \prod_{i=1}^n p(x_i | \mathcal{C}_k)}{\prod_{i=1}^n p(x_i)}$$

Czemu niezależność warunkowa? Bo zakładamy, że wartości cechy przy założeniu klasy  $p(x_i | \mathcal{C}_k)$  są niezależne.

$p(\mathcal{C}_k)$  to prior dla klasy, typowo szacuje się go przez sprawdzenie proporcji klas w zbiorze treningowym. Można tu też dodać inne informacje, np. ważenie klas (dla problemów niebalansowanych).

W mianowniku mamy dowód, przy czym ten evidence to po prostu prawdopodobieństwo wygenerowania zbioru przy danym rozkładzie każdej cechy. Jest więc stały dla poszczególnych klas i służy tylko za czynnik normalizujący prawdopodobieństwa, żeby sumowały się do 1. Jeżeli interesuje nas sama klasyfikacja, to można to pominąć.

Klasa jest wybierana zgodnie z MAP:

$$p(\mathcal{C}_k | x_1, \dots, x_n) \propto p(\mathcal{C}_k) \prod_{i=1}^n p(x_i | \mathcal{C}_k)$$
$$\mathcal{C} = \operatorname{argmax}_{\mathcal{C}_k} \left\{ p(\mathcal{C}_k) \prod_{i=1}^n p(x_i | \mathcal{C}_k) \right\}$$

Jeżeli chcemy cały rozkład posterior, to po prostu nie usuwamy dowodu z mianownika, tylko to liczymy i zwracamy cały rozkład.

Co ciekawe, powyższy podział na pojedyncze, jednowymiarowe podproblemy nie jest niezbędne. Można by podzielić problem na większe podzbiory cech i też je łączyć w identyczny sposób jak wyżej.

Pozostaje jeszcze kwestia tego, jak oszacować  $p(x_i | \mathcal{C}_k)$ . Ważne jest tutaj, czy cecha jest dyskretna, czy ciągła, bo w każdym wypadku trzeba wybrać inny rozkład prawdopodobieństwa do modelowania tego. Dla przypadku dyskretnego typowo zakłada się rozkład kategoryczny (wartości to dyskretny, nieuporządkowany zbiór), a dla ciągłego rozkład normalny. Są też inne możliwości, np. rozkład binomial, multinomial, albo w ogóle szacowanie z danych robiąc kernel density estimation (KDE)

## Kategoryczny naiwny klasyfikator Bayesa

Kiedy cecha jest dyskretna, a dodatkowo zakładamy, że ma rozkład kategoryczny, a więc jej wartości nie są ułożone po kolej, to naiwny klasyfikator Bayesa jest wyjątkowo prosty.

Robimy tabelkę, w której zliczamy wystąpienia każdej wartości w każdej klasie, dla każdej z cech po kolej. Żeby potem dostać wiarygodność  $p(x_i | C_k)$ , to wystarczy podzielić liczbę wystąpień wartości  $x_i$  dla danej cechy przez liczbę przykładów z danej klasy  $C_k$ .

Co więcej, atrybuty ciągłe też można sprowadzić do tego przypadku, robiąc **dyskretyzację (discretization / binning)**. Obsługuje to też łatwo wiedzę o wartościach brakujących, bo robimy dla nich osobny kubełek (bin). W praktyce daje to bardzo dobre wyniki, często nawet lepsze niż wersja Gaussowska (bo założenie o normalności może często nie być spełnione).

## Gaussowski naiwny klasyfikator Bayesa

Kiedy cecha jest ciągła i zakładamy, że ma rozkład normalny w obrębie klasy, to dostajemy Gaussowski naiwny klasyfikator Bayesa. Dla każdej cechy i klasy zakładamy rozkład normalny, z osobnymi parametrami średniej i odchylenia standardowego.

Robimy tabelkę wartości średnich i odchyleń standardowych dla każdej cechy i klasy. Wybieramy ze zbioru treningowego przykłady z danej klasy, szacujemy średnią i odchylenie standardowe, wrzucamy do tabelki. Żeby potem dostać wiarygodność, używamy wzoru:

$$p(x_i | C_k) = \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} \exp \left\{ -\frac{(x_i - \mu_{C_k})^2}{2\sigma_{C_k}^2} \right\}$$

## Naiwny klasyfikator Bayesa - zalety, wady i ulepszenia

### Zalety:

- działa bardzo dobrze, jeżeli założenie o niezależności cech jest spełnione
- wymaga stosunkowo niewielkiego zbioru uczącego
- szybki
- łatwy w implementacji
- działa dobrze dla wysokowymiarowych danych - jest odporny na klatwę wymiarowości, bo rozważa każdą cechę osobno
- działa dla wielu klas
- łatwa aktualizacja prawdopodobieństw przy nowych danych, można używać online
- może działać dla danych nie mieszczących się w pamięci (dzięki aktualizacji online)
- można użyć dodatkowej wiedzy dziedzinowej i bardziej złożonych rozkładów dla cech ciągłych, albo estymacji KDE

**Wady:**

- mocne założenie niezależności cech, właściwie nierealne (najwyżej mniej lub bardziej spełnione)
- problem braku wartości - jeżeli w zbiorze treningowym nie ma jakieś wartości dla danej cechy i klasy, to zeruje nam to prawdopodobieństwo
- jednakowe wagi cech
- ciężko radzić sobie z cechami ciągłymi - trzeba albo dyskretyzować (w jaki sposób? ile kubelków?), albo albo założyć jakiś rozkład (jaki?)
- nie działa domyślnie za dobrze dla klas niebalansowanych, trzeba sobie z tym radzić dodatkowo
- słaby estymator gęstości prawdopodobieństwa a posteriori, o ile stosunek prawdopodobieństw między klasami jest całkiem solidny, to sama wyjściowa estymacja prawdopodobieństwa jest nieprecyzyjna (tym bardziej, im bardziej zależne są od siebie cechy)

**Ulepszenia:**

- analiza korelacji i usunięcie mocno skorelowanych cech, lepiej spełniamy założenie o niezależności
- dobry feature engineering, tak aby zapewnić niezależne (ortogonalne) cechy
- dobranie odpowiedniego priora, użycie wiedzy dziedzinowej
- ensembling (bagging albo boosting) wielu klasyfikatorów, żeby zmniejszyć wariancję
- transformacja Box-Cox albo Yeo-Johnson dla cech ciągłych, żeby sprowadzić je bliżej rozkładów normalnych
- użycie wygładzania Laplace'a (Laplace smoothing), czyli dodanie +M (zwykle +1) do wystąpień każdej wartości, żeby uniknąć problemu braku wartości

# Procesy Gaussowskie

## Procesy stochastyczne

**Proces stochastyczny (stochastic process)** opisuje system losowo zmienny w czasie. W każdym kroku czasowym następuje zmiana aktualnych współrzędnych o losowe przesunięcie. Trajektoria (realizacja, ścieżka) dla danego punktu startowego będzie za każdym razem inna, bo mamy losowość w procesie.

Każda realizacja takiego procesu może być przy tym traktowana jako pewna funkcja, bo mamy kolejne wartości dla kolejnych argumentów. Cały proces stochastyczny może być zatem traktowany jak rozkład nad funkcjami.

Prosty przykład to **proces Wienera (ruch Browna)**, dla uproszczenia rozważmy przypadek 1D. Zaczynamy od współrzędnej zero i w każdym kroku dodajemy losową wartość z rozkładu normalnego:

$$d(t + \Delta t) = d(t) + \Delta d$$

$$\Delta d \sim \mathcal{N}(0, \Delta t)$$

Jest to też przy okazji proces Gaussowski.

## Procesy Gaussowskie

**Process Gaussowski (Gaussian process, GP)** to taki rodzaj procesu stochastycznego, w którym definiujemy rozkład normalny nad funkcjami  $f(\mathbf{x})$ , zdefiniowany przez:

- funkcję wartości średniej (mean function)  $m(\mathbf{x})$
- pozytywną funkcję kowariancji (funkcję jądra, kernel function)  $k(\mathbf{x}, \mathbf{x}')$

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

Dodatkowo funkcja  $f(\mathbf{x})$  dla dowolnego zestawu punktów  $\mathbf{x}$  ma łączny rozkład normalny opisany wektorem średnich  $m(\mathbf{x})$  i macierzą kowariancji  $k(\mathbf{x}, \mathbf{x})$ :

$$\mathbf{y} = f(\mathbf{X}) \sim \mathcal{N}(m(\mathbf{X}), k(\mathbf{X}, \mathbf{X}))$$

Macierz jest **dodatnio określona**, jeżeli jest symetryczna, rzeczywista i spełnia własność:

$$\mathbf{z}^T \mathbf{A} \mathbf{z} > 0 \text{ for } \mathbf{z} \in \mathbb{R}^{n \times 1} \text{ and } \mathbf{z} \neq \mathbf{0}$$

Jest to wymaganie, żeby macierz była w stanie opisywać jądro (kernel), jest związane z twierdzeniem Mercera (wyznacza warunki dla macierzy opisujących kernele).

Skoro proces Gaussowski definiuje rozkład, to możemy z niego próbować - jako próbki dostajemy funkcję. Potrzebujemy do tego funkcji średniej i kernela (funkcji kowariancji). Ta druga definiuje łączną zmienność zmiennych losowych tworzących proces Gaussowski i zwraca kowariancję dla każdej pary wejść  $k(\mathbf{x}, \mathbf{x}')$ . **Kernel stanowi prior**, więc możemy z jego pomocą dodać informacje na wejście co do rozkładu  $f(\mathbf{x})$ .

Skoro coś ma być priorem, to musimy być w stanie to próbować. Sam proces Gaussowski definiuje nieskończonymiarowy rozkład nad funkcjami, a my chcemy dostać skończonewymiarowy podzbiór jako próbę. Robi się to tak:

1. Próbkujemy N punktów (np. równoodległych) z jakiegoś przedziału
2. Obliczamy dla tych punktów kernel:

$$\boldsymbol{\Sigma} = k(\mathbf{X}, \mathbf{X})$$

3. Z procesu Gaussowskiego generujemy 5 próbek (realizacji) z N-wymiarowego rozkładu Gaussowskiego, używając średniej 0 i obliczonej macierzy kowariancji  $\Sigma$

W powyższym przykładzie można też podać wektor średnich jako dodatkowy prior, np. gdy jesteśmy bardziej zainteresowani próbłowaniem w jakimś obszarze.

Jakie mogą być takie funkcje kowariancji? Generalnie są to te same kernele co używane w kernel SVM dla obliczania odległości (bo idea jest dokładnie ta sama), na przykład:

- Radial Basis Function (RBF):

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2\sigma^2} \|\mathbf{x} - \mathbf{x}'\|^2\right)$$

- exponential kernel:

$$k(\mathbf{x}, \mathbf{x}') = \exp(-\theta \|\mathbf{x} - \mathbf{x}'\|)$$

Warto zauważyć, że kernele są parametryzowane, więc tu też mamy możliwość dodania wiedzy wstępnej (i konieczność optymalizacji hiperparametru).

## Regresja procesem Gaussowskim

Znowu chcemy robić regresję, tylko trochę potężniejszym narzędziem. Podstawy są takie same: kombinacja liniowa, wybieramy funkcje bazowe  $\phi$ , narzucamy **prior dla wag (wycentrowany izotropowy rozkład Gaussa)**.

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x})$$

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, \alpha^{-1} \mathbf{I})$$

Mamy ponadto łączne prawdopodobieństwo wartości funkcji dla modelu:

$$\mathbf{y} = [y(\mathbf{x}_1), \dots, y(\mathbf{x}_n)]^T \quad \mathbf{y} = \Phi \mathbf{w} \quad [\Phi]_{i,j} = \phi_j(\mathbf{x}_i)$$

$\Phi$  to macierz wartości funkcji bazowych dla poszczególnych punktów i wymiarów.

Mamy oznaczenia, to teraz żeby skonstruować proces Gaussowski potrzebujemy funkcji wartości średnich i kernela. Wartość średnia to wartość oczekiwana, więc liczymy, używając wzoru na  $y$  podanego powyżej:

$$\mathbb{E}[y] = \Phi \mathbb{E}[w] = 0$$

Skoro założyliśmy wycentrowany prior dla wag, no to wartość oczekiwana to 0.

Kowariancja:

$$\text{cov}[y] = \mathbb{E}[yy^T] = \Phi \mathbb{E}[ww^T] \Phi^T = \frac{1}{\alpha} \Phi \Phi^T = K$$

$$[K]_{i,j} = k(x_i, x_j) = \frac{1}{\alpha} \phi(x_i)^T \phi(x_j)$$

Najpierw robimy definicję kowariancji, potem podstawiamy za  $y$ , upraszczamy i dostajemy macierz. Jest to macierz symetryczna, rzeczywista, dodatnio określona, a więc w pełni nadaje się na kernel. Jest on obliczony z naszego priora dla wag, więc bezpośrednio wynika z naszej wiedzy wstępnej.  $K$  to przy okazji macierz Grama, czyli macierz iloczynów skalarnych.

Warto podkreślić tutaj ten związek - **prior dla wag, czyli baza wielomianów definiuje kernel**, albo możemy zrobić w drugą stronę i **zdefiniować bazę narzucając kernel**.

Z powyższego otrzymujemy rozkład posterior, który też jest wycentrowym (ale niekoniecznie izotropowym!) rozkładem normalnym:

$$p(y) = \mathcal{N}(y|0, K)$$

W regresji liniowej zakładaliśmy jednak, że prawdziwe wartości to wartości przewidywane przez nasz model oraz pewien szum. Jest on nieunikniony, proces Gaussowski też nie przewiduje idealnie, więc uwzględnijmy to. Szum ma oczywiście znowu rozkład normalny i jest wycentrowany w prawdziwych wartościach.

$$t_i = y_i + \epsilon_i$$

$$p(t_i|y_i) = \mathcal{N}(t_i|y_i, \beta^{-1})$$

$$p(\mathbf{t}_n|\mathbf{y}_n) = \mathcal{N}(\mathbf{t}_n|\mathbf{y}_n, \beta^{-1} I_n)$$

Powyższe to oczywiście sama wiedza, nic jeszcze użytecznego nie zaprojektowaliśmy. To, czego tak naprawdę potrzebujemy, to rozkład predykcyjny.

Rozkład predykcyjny nad zbiorem:

$$p(\mathbf{t}) = \int p(\mathbf{t}|\mathbf{y})p(\mathbf{y})d\mathbf{y} = \mathcal{N}(\mathbf{t}|\mathbf{0}, \mathbf{C})$$

$$[\mathbf{C}]_{i,j} = C(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) + \beta^{-1}\delta_{ij}$$

Warto zauważyć, że w powyższym wzorze kowariancja jest dana przez sumę kernela oraz szumu. Delta Kroneckera jest użyta dlatego, że szum ma być dodany tylko na przekątnej.

Chcemy natomiast rozkładu predykcyjnego dla nowych punktów:

$$p(t_{n+1}|\mathbf{t}_n, \mathbf{X}_n, \mathbf{x}_{n+1}) = p(t_{n+1}|\mathbf{t}_n, \mathbf{x}_1, \dots, \mathbf{x}_{n+1})$$

Zajmijmy się na początek rozkładem łącznym dla wartości:

$$p(\mathbf{t}_{n+1}) = p(t_1, \dots, t_{n+1}) = \mathcal{N}(\mathbf{t}_{n+1}|\mathbf{0}, \mathbf{C}_{n+1})$$

Macierz kowariancji musi "urosnąć" o 1 wymiar, żeby przewidzieć nowy punkt. Musimy do niej dodać kowariancje nowego punktu i tych ze zbioru treningowego, do czego używamy kernela:

$$\mathbf{C}_{n+1} \in \mathbb{R}^{(n+1) \times (n+1)} - \text{covariance matrix}$$

$$C(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) + \beta^{-1}\delta_{ij}$$

$$\mathbf{C}_{n+1} = \begin{bmatrix} \mathbf{C}_n & \mathbf{k} \\ \mathbf{k}^T & c \end{bmatrix}$$

$$c = k(\mathbf{x}_{n+1}, \mathbf{x}_{n+1}) + \beta^{-1}$$

$$\mathbf{k} = [k(\mathbf{x}_1, \mathbf{x}_{n+1}), \dots, k(\mathbf{x}_n, \mathbf{x}_{n+1})]^T$$

Możemy teraz obliczyć rozkład predykcyjny. Żeby to zrobić, wystarczy "odciąć" przewidywany punkt od pozostałych za pomocą prawdopodobieństwa warunkowego:

$$p(t_{n+1}|\mathbf{t}_n, \mathbf{X}_n, \mathbf{x}_{n+1}) = \mathcal{N}(t_{n+1}|m(\mathbf{x}_{n+1}), \sigma^2(\mathbf{x}_{n+1}))$$

$$m(\mathbf{x}_{n+1}) = \mathbf{k}^T \mathbf{C}_n^{-1} \mathbf{t}_n$$

$$\sigma^2(\mathbf{x}_{n+1}) = c - \mathbf{k}^T \mathbf{C}_n^{-1} \mathbf{k}_n$$

Algorytm regresji procesem Gaussowskim:

1. Zbieramy punkty treningowe
2. Definiujemy punkty testowe, czyli obszar, na którym będziemy obliczać funkcje próbkowane z procesu Gaussowskiego (bo proces zwraca funkcje), np. N punktów z przedziału [A, B]
3. Obliczamy średnią i wariancję posteriora w punktach testowych
4. Zwracamy rozkład posterior oraz próbki z posteriora

Intuicyjnie - proces Gaussowski pozwala nam próbować z niego realizacje funkcji dla zadanego zakresu punktów testowych. Zdobywamy zbiór funkcji i liczymy dla nich wartości średnie (czyli "średnią funkcję") oraz odchylenia standardowe.

## Kernele

W procesie Gaussowskim prior narzucamy przez wybór kernela. Co więcej, kernele można składać w bardziej złożone kernele, żeby dostać wiele własności naraz. Dwa dowolne kernele możemy **pomnożyć** (odpowiada operacji AND) lub **dodać** (odpowiada operacji OR).

**White noise kernel** to po prostu biały szum w danych. Używa się tego właściwie zawsze.

$$k(\mathbf{x}_a, \mathbf{x}_b) = \beta^{-1} \mathbf{I}_n$$

**Exponential quadratic kernel** jest funkcją gładką, do regularnych zmian w danych.

$$k(\mathbf{x}_a, \mathbf{x}_b) = \lambda^2 \exp\left(-\frac{\|\mathbf{x}_a - \mathbf{x}_b\|^2}{2\sigma^2}\right)$$

$\lambda^2$  - ogólny wariancji w danych, wpływa na maksymalne wartości kowariancji

$\sigma$  - skala długości, wpływa na rozrzut kowariancji, można definiować dopuszczalną szybkość zmian

**Rational quadratic kernel** to nieskończona ważona suma exponential quadratic kernels.

$$k(\mathbf{x}_a, \mathbf{x}_b) = \lambda^2 \left(1 + \frac{\|\mathbf{x}_a - \mathbf{x}_b\|^2}{2\alpha\sigma^2}\right)^{-\alpha}$$

$\alpha$  - wyznacza ważenie według kolejnych skal kerneli

**Periodic kernel** jest zbudowany z funkcji okresowej (sinus) i pozwala reprezentować cykliczne zależności.

$$k(\mathbf{x}_a, \mathbf{x}_b) = \lambda^2 \exp\left(-\frac{2}{\sigma^2} \sin^2\left(\pi \frac{\|\mathbf{x}_a - \mathbf{x}_b\|}{p}\right)\right)$$

$p$  - okres, odległość między powtórzeniami, pozwala definiować częstotliwość

**Local periodic kernel** to iloczyn periodic kernel oraz exponential quadratic kernel. Pozwala to na modelowanie długofalowych cykli.

$$k(\mathbf{x}_a, \mathbf{x}_b) = \lambda^2 \exp\left(-\frac{2}{\sigma_p^2} \sin^2\left(\pi \frac{\|\mathbf{x}_a - \mathbf{x}_b\|}{p}\right)\right) \exp\left(-\frac{\|\mathbf{x}_a - \mathbf{x}_b\|^2}{2\sigma_{eq}^2}\right)$$

## Nauka parametrów kerneli

Typowo w praktyce używa się złożonych kerneli. Najpierw sprawdzamy zależności w danych: trendy długoterminowe, zmiany cykliczne, wielkość szumu etc. Każdą z tych cech reprezentuje się jako odpowiedni kernel. Typowo potem zakładamy niezależność tych zmian i dodajemy kernele. Taki model ma bardzo dużą siłę ekspresji.

Pozostaje problem, jak dobrać takie parametry - jest już ich bardzo dużo, więc trzeba to inteligentnie zoptymalizować. Robimy to oczywiście maksymalizując wiarygodność, czyli minimalizując NLL. Bez utraty ogólności zakładamy, że dane są wycentrowane.

$$t(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

$$p(\mathbf{t}|\mathbf{X}, \theta) = \mathcal{N}(\mathbf{t}|\boldsymbol{\mu}, \boldsymbol{\Sigma}_{\theta}) = \mathcal{N}(\mathbf{t}|m(\mathbf{X}), \mathbf{C}_{\theta})$$

$$m(\mathbf{X}) = \mathbf{0}$$

$$\boldsymbol{\theta}_{ML} = \underset{\theta}{\operatorname{argmax}} \{p(\mathbf{t}|\mathbf{X}, \theta)\} = \underset{\theta}{\operatorname{argmin}} \{-\ln p(\mathbf{t}|\mathbf{X}, \theta)\}$$

$$\ln p(\mathbf{t}|\mathbf{X}, \theta) = -\frac{1}{2} \ln |\mathbf{C}_{\theta}| - \frac{1}{2} \mathbf{t}^T \mathbf{C}_{\theta}^{-1} \mathbf{t} - \frac{n}{2} \ln (2\pi)$$

Używa się tutaj optymalizacji gradientowej, jak w sieciach neuronowych. Wynika to z tego, że niestety **powierzchnia funkcji dla procesów Gaussowskich nie jest wypukła i ma wiele lokalnych maksimów**.

$$\begin{aligned} \frac{\partial}{\partial \theta_i} \ln p(\mathbf{t}|\mathbf{X}, \theta) &= -\frac{1}{2} \operatorname{Tr} \left( \mathbf{C}_{\theta}^{-1} \frac{\partial \mathbf{C}_{\theta}}{\partial \theta_i} \right) + \frac{1}{2} \mathbf{t}^T \mathbf{C}_{\theta}^{-1} \frac{\partial \mathbf{C}_{\theta}}{\partial \theta_i} \mathbf{C}_{\theta}^{-1} \mathbf{t} \\ \frac{\partial}{\partial \theta_i} \ln p(\mathbf{t}|\mathbf{X}, \theta) &= 0 \end{aligned}$$

W związku z powyższymi problemami używa się typowo **optymalizatora Adam**. Jest to zbiór ulepszeń klasycznego SGD, na przykład:

- adaptacyjny learning rate - zmienia się automatycznie z czasem
- osobny learning rate dla każdej cechy
- wykorzystywanie pierwszych i drugich pochodnych, w efektywny obliczeniowo sposób
- może wykorzystywać gradienty rzadkie, dobrze radzi sobie dla zaszumionych problemów

## Klasyfikacja procesem Gaussowskim

Założymy dla uproszczenia, że chcemy klasyfikacji binarnej. W takim wypadku nasz cel ma rozkład Bernoulliego, czyli 0 albo 1. Żeby dostać coś takiego wrzucamy wynik procesu Gaussowskiego do softmaxa i dostajemy proces stochastyczny (ale non-Gaussian!), z którego mamy  $y = \sigma(a)$ , gdzie  $a(x)$  to funkcja z procesu Gaussowskiego.

Mamy więc, że przynależność próbki do klasy pod warunkiem predykcji z procesu Gaussowskiego:

$$p(t|a) = \sigma(a)^t (1 - \sigma(a))^{(1-t)}$$

Chcemy przewidywać klasę kolejnego punktu, do czego potrzebujemy rozkładu predykcyjnego:

$$p(t_{n+1}|\mathbf{t}_n, \mathbf{X}_n, \mathbf{x}_{n+1})$$

Zakładamy dla uproszczenia pomiary dokładne (bez szumu) i robimy nad nimi prior procesu Gaussowskiego:

$$\mathbf{a}_{n+1} = [a(\mathbf{x}_1), \dots, a(\mathbf{x}_{n+1})]$$

$$p(\mathbf{a}_{n+1}) = \mathcal{N}(\mathbf{a}_{n+1} | \mathbf{0}, \mathbf{C}_{n+1})$$

$\mathbf{C}_{n+1} \in \mathbb{R}^{(n+1) \times (n+1)}$  - covariance matrix

$$[\mathbf{C}_{n+1}]_{i,j} = C(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) + \nu \delta_{ij}$$

Założyliśmy brak szumu, a jednak mamy tu deltę Kroneckera i dodawanie czegoś na przekątną. Tutaj jednak przyczyna jest inna - ulepsza to własności numeryczne macierzy, zapewnia jej dodatnią określoność.

Do klasyfikacji potrzebujemy rozkładu predykcyjnego nad danymi treningowymi, ale dla naszego procesu stochastycznego (non-Gaussian):

$$p(t_{n+1} = 1 | \mathbf{t}_n) = \int p(t_{n+1} = 1 | a_{n+1}) p(a_{n+1} | \mathbf{t}_n) da_{n+1}$$

$$p(t_{n+1} = 1 | a_{n+1}) = \sigma(a_{n+1})$$

Całka powyżej jest konwolucją sigmoidy z prawdopodobieństwem obliczonym z wyników procesu Gaussowskiego i prawdopodobieństw tych predykcji warunkowanych danymi. Problem jest taki, że taka całka jest nie do policzenia analitycznie i trzeba ją przybliżyć. Jest kilka takich metod, prostszych lub bardziej złożonych.

**Przybliżenie Laplace'a (Laplace approximation)** to prosty sposób przybliżania wartości całek niepoliczalnych analitycznie za pomocą szeregu Taylora. Przybliża ona funkcję gęstości prawdopodobieństwa  $p(z)$  definiowaną nad zbiorem zmiennych ciągłych za pomocą rozkładu Gaussowskiego  $q(z)$ .

Metoda działa też dla rozkładów multimodalnych, wtedy można dokonać wielu przybliżeń (po 1 na maksimum lokalne).

W przypadku pojedynczej zmiennej ciągłej aproksymacja taka jest wycentrowana w modzie wejściowego rozkładu, innymi słowy chcemy, żeby dominantą była osiągana dla tych samych argumentów.

Na początek znajdujemy modę oryginalnego rozkładu. Robi się to oczywiście numerycznie, dowolnym algorytmem optymalizacji.

$$\nabla f(z_0) = \mathbf{0}$$

Logarytmujemy naszą funkcję, przybliżamy logarytm szeregiem Taylora i przekształcamy, żeby przybliżyć oryginalną funkcję:

$$u(z) = \ln f(z)$$

$$u(z) \simeq u(z_0) - \frac{1}{2}(z - z_0)^T \mathbf{A}(z - z_0)$$

$$f(z) \simeq f(z_0) \exp \left\{ -\frac{1}{2}(z - z_0)^T \mathbf{A}(z - z_0) \right\}$$

Macierz A to hesjan. Przy okazji mamy tu też laplasjan (stąd nazwa metody):

$$\mathbf{A} = -\nabla^2 u(z) \Big|_{z=z_0} = -\Delta \ln f(z) \Big|_{z=z_0}$$

Normalizujemy, żeby był to poprawny rozkład prawdopodobieństwa:

$$q(z) = \frac{|\mathbf{A}|^{1/2}}{(2\pi)^{m/2}} \exp \left\{ -\frac{1}{2}(z - z_0)^T \mathbf{A}(z - z_0) \right\}$$

Żeby to działało, to macierz A musi być dodatnio określona, bo wtedy punkt stacjonarny  $z_0$  to maksimum.

Jak jesteśmy wyposażeni w przybliżenie Laplace'a, to możemy już wrócić do obliczania rozkładu predykcyjnego dla klasyfikacji. Mieliśmy:

$$p(t_{n+1} = 1 | \mathbf{t}_n) = \int \sigma(a_{n+1}) p(a_{n+1} | \mathbf{t}_n) da_{n+1}$$

Korzystając z przybliżenia konwolucji z sigmoidą za pomocą rozkładu Gaussowskiego mamy:

$$\int \sigma(a) \mathcal{N}(a | \mu, \sigma^2) da \simeq \sigma(\kappa(\sigma^2)\mu)$$

$$\kappa(\sigma^2) = (1 + \pi\sigma^2/8)^{-1/2}$$

Jest tylko pewien problem - drugi element z tej całki musi mieć rozkład normalny, żeby takie przybliżenie dało się zrobić. Z twierdzenia Bayesa mamy:

$$\begin{aligned} p(a_{n+1} | \mathbf{t}_n) &= \int p(a_{n+1}, \mathbf{a}_n | \mathbf{t}_n) d\mathbf{a}_n \\ &= \frac{1}{p(\mathbf{t}_n)} \int p(a_{n+1}, \mathbf{a}_n) p(\mathbf{t}_n | a_{n+1}, \mathbf{a}_n) d\mathbf{a}_n \\ &= \frac{1}{p(\mathbf{t}_n)} \int p(a_{n+1} | \mathbf{a}_n) p(\mathbf{a}_n) p(\mathbf{t}_n | \mathbf{a}_n) d\mathbf{a}_n \\ &= \underbrace{\int p(a_{n+1} | \mathbf{a}_n)}_{\text{GP regression}} \underbrace{p(\mathbf{a}_n | \mathbf{t}_n)}_{\Delta \text{approximation}} d\mathbf{a}_n \end{aligned}$$

Samo wyprowadzenie ma małe znaczenie, ale pod koniec mamy 2 ciekawe elementy: wynik z regresji procesem Gaussowskim i drugi element, który trzeba przybliżyć przybliżeniem Laplace'a. Pierwsze mamy, jest prosto:

$$p(a_{n+1} | \mathbf{a}_n) = \mathcal{N}(a_{n+1} | \mathbf{k}^T \mathbf{C}_n^{-1} \mathbf{a}_n, c - \mathbf{k}^T \mathbf{C}_n^{-1} \mathbf{k}_n)$$

Drugi element to wiarygodność wyników regresji procesem Gaussowskim przy danych klasach. Przy założeniu i.i.d.:

$$p(\mathbf{a}_n | \mathbf{t}_n) = \prod_{i=1}^n \sigma(a_i)^{t_i} (1 - \sigma(a_i))^{1-t_i} = \prod_{i=1}^n e^{a_i t_i} \sigma(-a_i)$$

Mamy prawdopodobieństwo z eksponentą - niewygodne. Logarytmujemy i używamy aproksymacji Laplace'a. Znajdujemy modę, obliczamy laplasjan. Dobre wieści - posterior ma jedno globalne maksimum, więc przybliżenie będzie ładne. Dostajemy:

$$\mathbf{a}_n^* = \mathbf{C}_n(\mathbf{t}_n - \boldsymbol{\sigma}_n)$$

$$q(\mathbf{a}_n) = \mathcal{N}(\mathbf{a}_n | \mathbf{a}_n^*, \mathbf{A}^{-1})$$

Podsumowując:

- najpierw robimy regresję procesem Gaussowskim (po prostu wartości regresji to 0 albo 1)
- przybliżamy rozkład posterior przybliżeniem Laplace'a
- rozkład prawdopodobieństwa dla predykcji dla nowego punktu obliczamy kolejnym przybliżeniem Laplace'a
- obliczamy przybliżenie rozkładu predykcyjnego jako konwolucji z sigmoidą

Pozostaje jeszcze pytanie, co z kernelem. Tutaj robimy MAP dla danych pod warunkiem parametrów kernela. Klasycznie - logarytmujemy wiarygodność, odwracamy znaki, używamy optymalizacji Adamem.

$$p(\mathbf{t}_n|\theta) = \int p(\mathbf{t}_n|\mathbf{a}_n)p(\mathbf{a}_n|\theta)d\mathbf{a}_n$$

$$\ln p(\mathbf{t}_n|\theta) = \Psi(\mathbf{a}_n^*) - \frac{1}{2} \ln |\mathbf{W}_n + \mathbf{C}_n^{-1}| + \frac{n}{2} \ln 2\pi$$

## Zalety i wady procesów Gaussowskich

### Zalety:

- tuning parametrów wbudowany w algorytm, nie trzeba robić osobno
- zwraca rozkład prawdopodobieństwa dla predykcji, mamy przedziały ufności
- szeroki wybór priorów (różne kernele i ich złożenia)
- model nieparametryczny, wygodny black box
- wszechstronność, działa dla wielu różnych problemów

### Wady:

- złożoność obliczeniowa  $\mathcal{O}(n^3)$
- gęstość - brak rzadkości, używamy zawsze pełnych danych
- tracą wydajność dla bardzo wysokowymiarowych przestrzeni ( $>10^5$ )